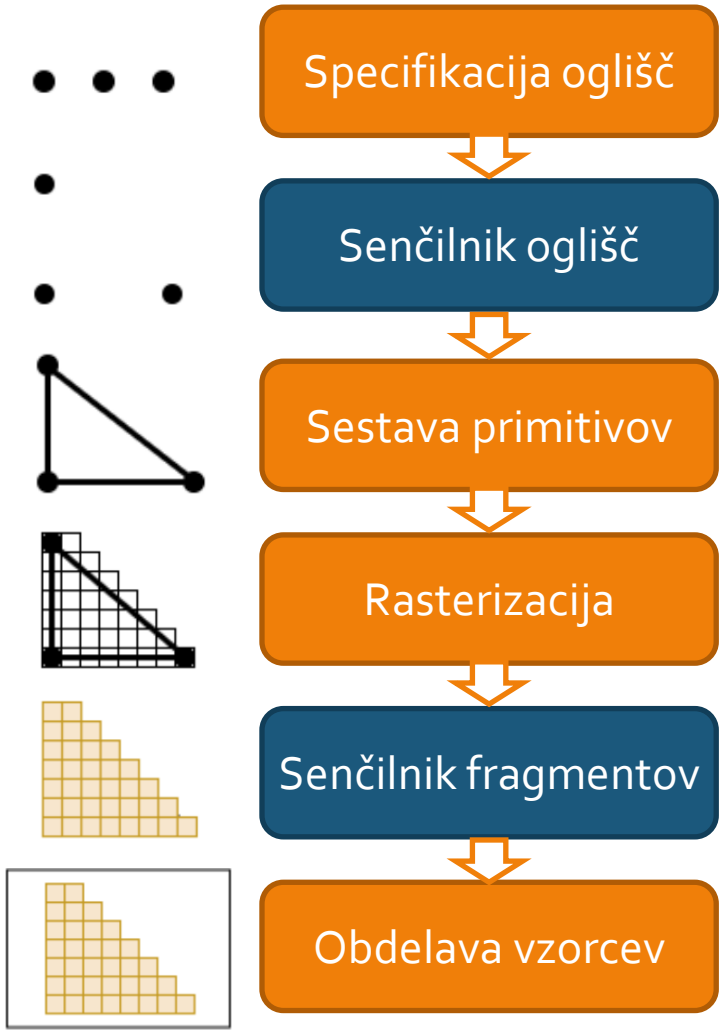
BARVANJE

Kako pobarvati poligone?

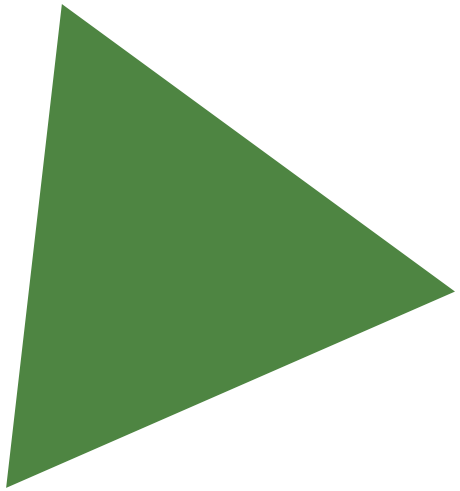
- Transformacija v k.s. pogleda in projekcija
  - v **senčilniku oglišč**
  - rezultat so trikotniki v normaliziranih koordinatah naprave (NDC)
- **Rasterizacija**
  - določi katere piksle/fragmente pokriva vsak trikotnik
  - interpolira vrednosti v ogliščih na nivo piksla/fragmenta
- Določanje barve pikslov/fragmentov
  - **senčilnik fragmentov**

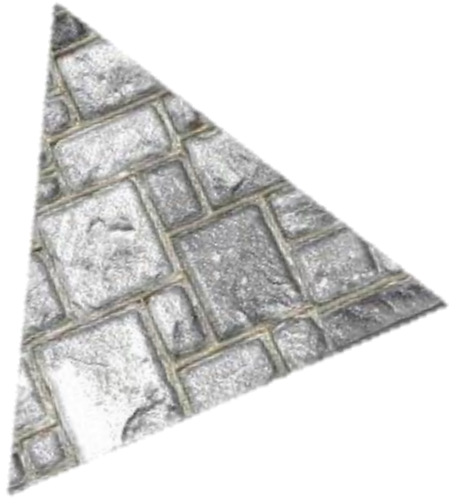


Kako pobarvati poligone?

- Barvo piksla/fragmenta izračunamo, npr. glede na barvo v ogliščih

- Barvo določimo glede na teksturo
  - 2D - tipično slika, ki jo nalepimo na trikotnike
  - 3D – zaporedje slik ali proceduralna tekstura



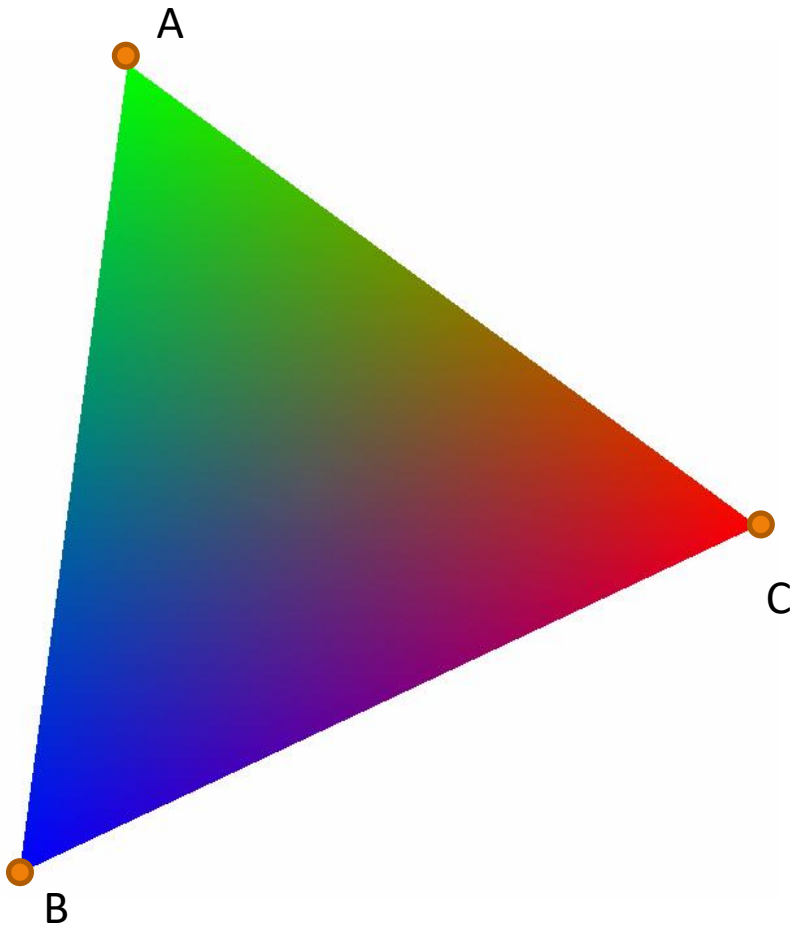


Interpolacija

Bilinearna interpolacija

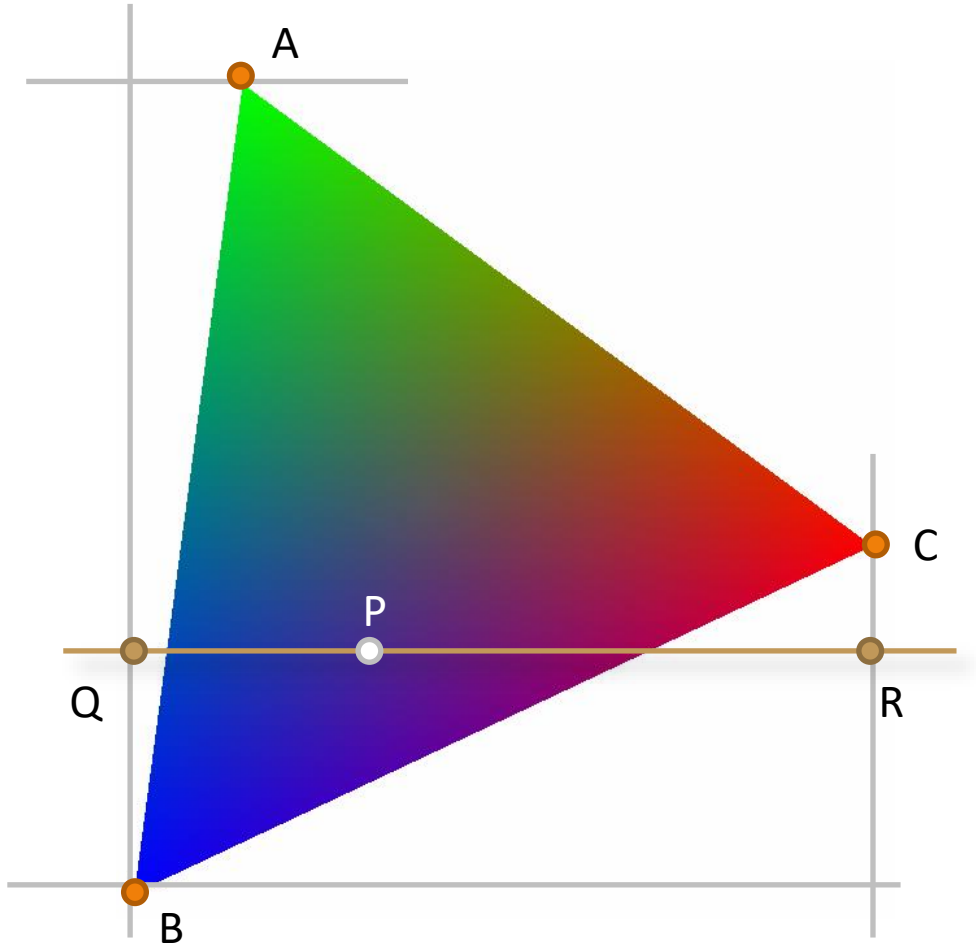
- Z **interpolacijo** določimo vrednosti v notranjosti trikotnika na podlagi vrednosti v ogliščih
  - gre za kakršnekoli lastnosti – barva, normale, teksturne koordinate, globina ipd.

- **Bilinearna interpolacija**
  - linearno interpolacijo izvedemo najprej po eni, nato po drugi osi



Bilinearna interpolacija

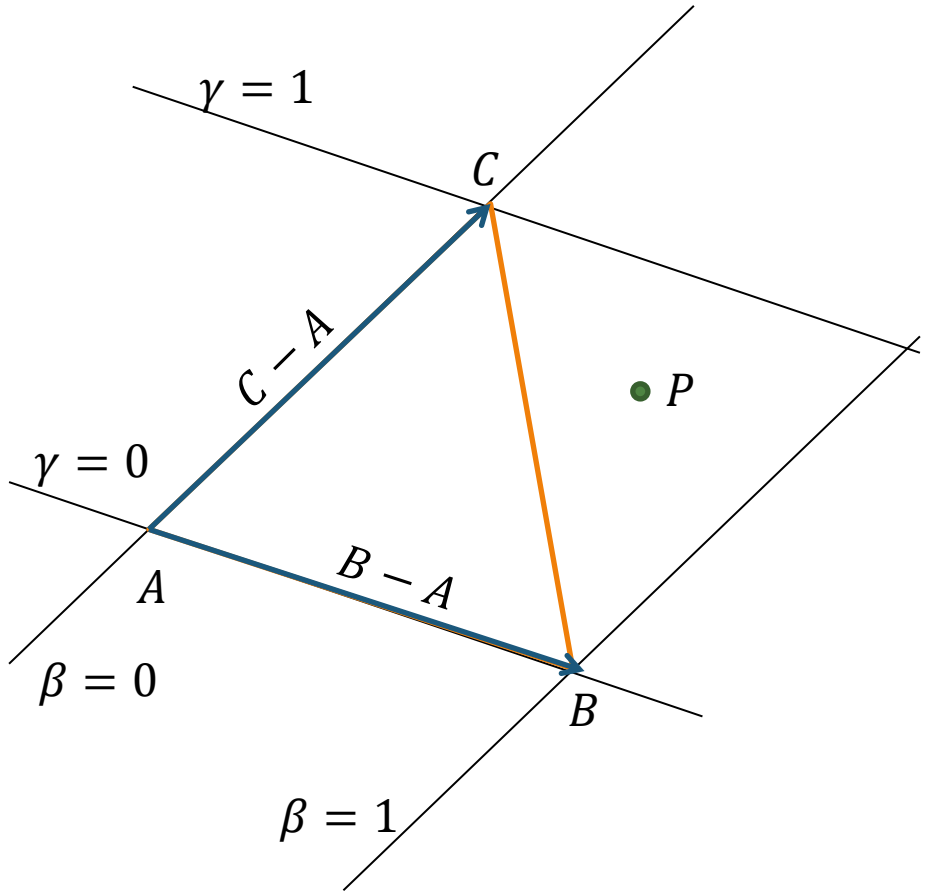
- Imamo trikotnik z oglišči A, B, C in vrednostmi V_A, V_B, V_C, ki jih želimo interpolirati v točki P
- Najprej interpoliramo po eni osi - npr. y, da dobimo vrednosti v Q in R
  - $V_Q = \frac{y_P - y_B}{y_A - y_B} V_A + \frac{y_A - y_P}{y_A - y_B} V_B$
  - $V_R = \frac{y_P - y_B}{y_C - y_B} V_C + \frac{y_C - y_P}{y_C - y_B} V_B$
- Potem interpoliramo po osi x, da dobimo končno vrednost v P
  - $V_P = \frac{x_C - x_P}{x_C - x_B} V_Q + \frac{x_P - x_B}{x_C - x_B} V_R$
- Izračun v zaporednih točkah lahko pohitrimo (sprememba je konstantna vzdolž smeri *x*)



Težiščne koordinate

- Pri trikotnikih lahko definiramo tudi t.i. **težiščni** (baricentrični) koordinatni sistem
  - koordinatni osi sta stranici trikotnika
- Točko v tem prostoru predstavimo kot:

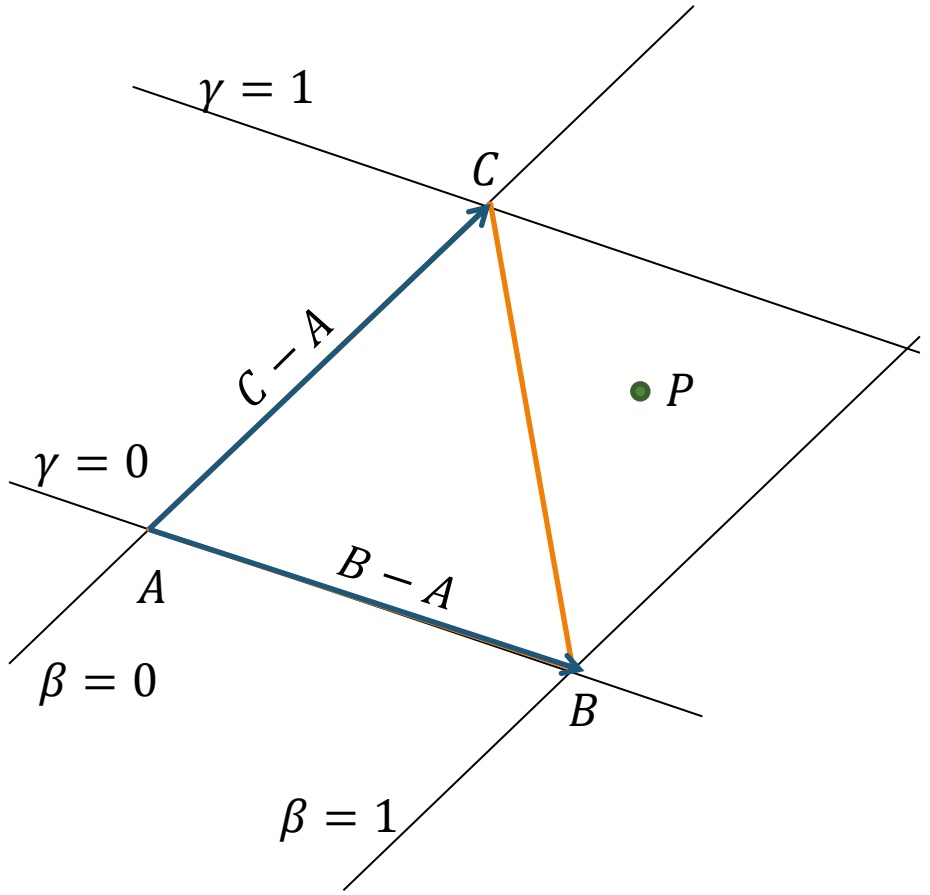
  $P = A + \beta(B - A) + \gamma(C - A)$, oz.
  $P = \alpha A + \beta B + \gamma C$
  $\alpha + \beta + \gamma = 1$



Izračun težiščnih koordinat v 2D

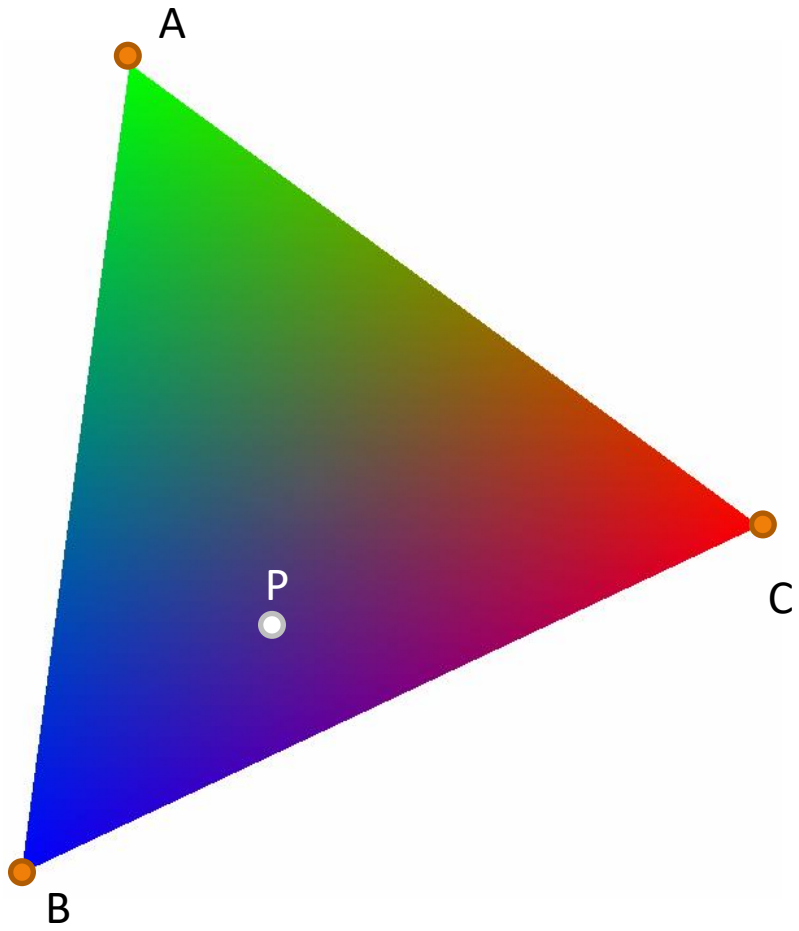
- V 2D prostoru (npr. zaslonskih koordinatah) lahko zapišemo:
  - $\beta(B-A)+\gamma(C-A)=P-A$
  - $\begin{bmatrix} x_B-x_A & x_C-x_A \\ y_B-y_A & y_C-y_A \end{bmatrix}\begin{bmatrix}\beta\\ \gamma\end{bmatrix}=\begin{bmatrix}x_P-x_A\\ y_P-y_A\end{bmatrix}$
- in dobimo:
  - $\beta=\frac{(x_A-x_C)(y_P-y_C)-(y_A-y_C)(x_P-x_C)}{(x_B-x_A)(y_C-y_A)-(x_C-x_A)(y_B-y_A)}$
  - $\gamma=\frac{(x_B-x_A)(y_P-y_A)-(y_B-y_A)(x_P-x_A)}{(x_B-x_A)(y_C-y_A)-(x_C-x_A)(y_B-y_A)}$
  - $\alpha=1-\beta-\gamma$

- Izračun v 3D



Težiščne koordinate

- Če poznamo predstavitev točke P s težiščnimi koordinatami α, β, γ, lahko:
  - enostavno računamo **bilinearno interpolacijo** lastnosti V oglišč v točki
    - $V_P = \alpha V_A + \beta V_B + \gamma V_C$
  - enostavno ugotovimo, če je točka **znotraj trikotnika**:
    - $0 < \alpha, \beta, \gamma < 1$: točka je znotraj trikotnika
    - $0 \leq \alpha, \beta, \gamma \leq 1$: točka je znotraj ali na robu trikotnika
    - sicer je točka izven trikotnika

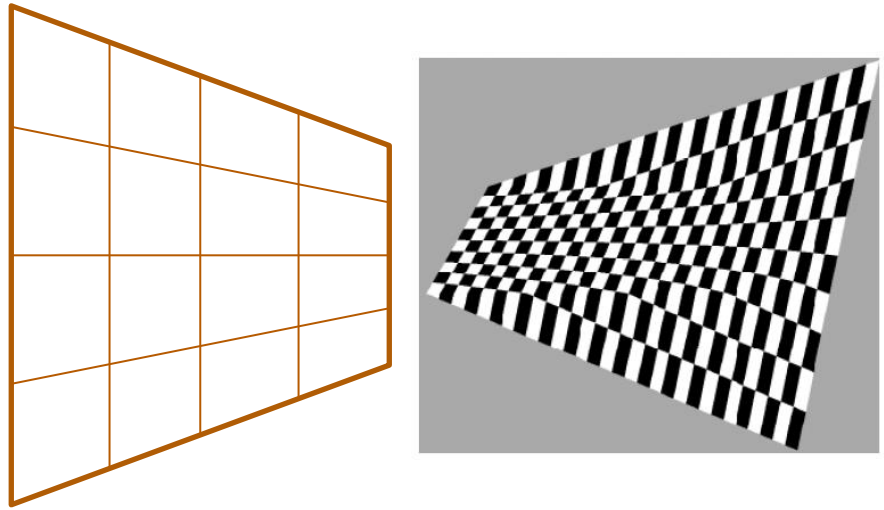


Perspektivno pravilna interpolacija

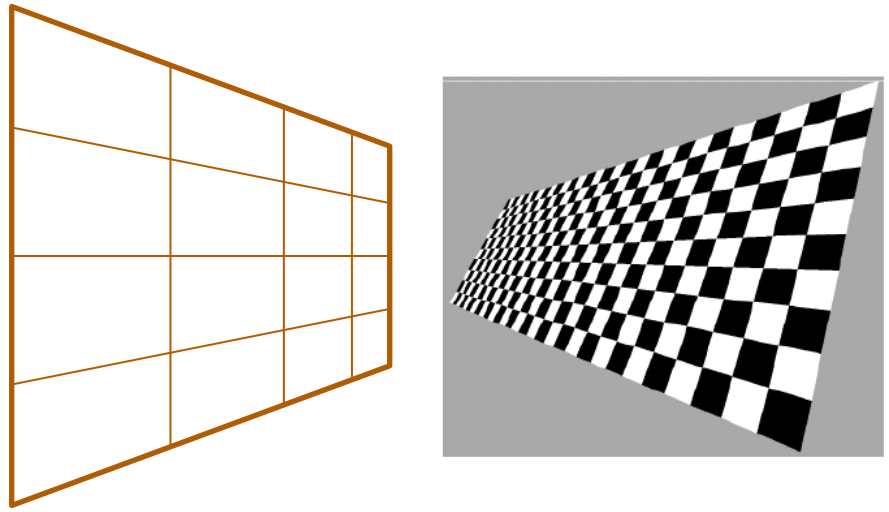
- Če uporabljamo **perspektivno** projekcijo, ta **popači** predmete
  - perspektiva ni linearna
  - delimo s homogeno koordinato *w* (*z/d*)
  - če izberemo *d=1:*

$$P' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

- Navadna bilinearna interpolacija po projekciji ne upošteva pravilno tega popačenja



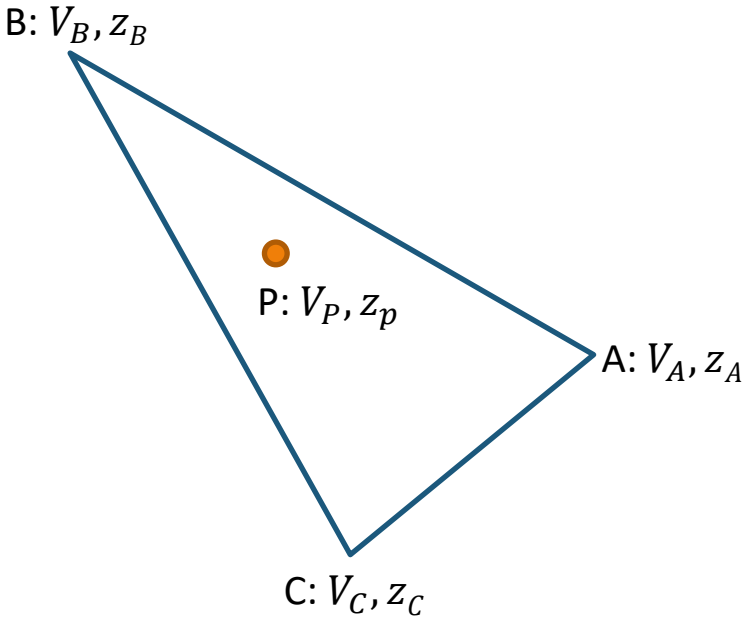
linearna interpolacija v koordinatah zaslona



perspektivno pravilna interpolacija

Perspektivno pravilna interpolacija

- Tudi vrednosti, ki jih interpoliramo *homogeniziramo* s faktorjem $w = \frac{z}{d}$ oz. $w = z$, če izberemo *d=1*
  - $V'_A = \frac{1}{z_A} V_A, V'_B = \frac{1}{z_B} V_B, V'_C = \frac{1}{z_C} V_C$
- Izvedemo bilinearno interpolacijo homogeniziranih vrednosti:
  - $V'_P = lerp(V'_A, V'_B, V'_C)$
- Bilinearno interpoliramo tudi faktorje, s katerimi smo množili ($\frac{1}{z}$):
  - $\frac{1}{z_P} = lerp\left(\frac{1}{z_A}, \frac{1}{z_B}, \frac{1}{z_C}\right)$
- Nazaj popravimo interpolirane homogenizirane vrednosti:
  - $V_P = z_P V'_P$

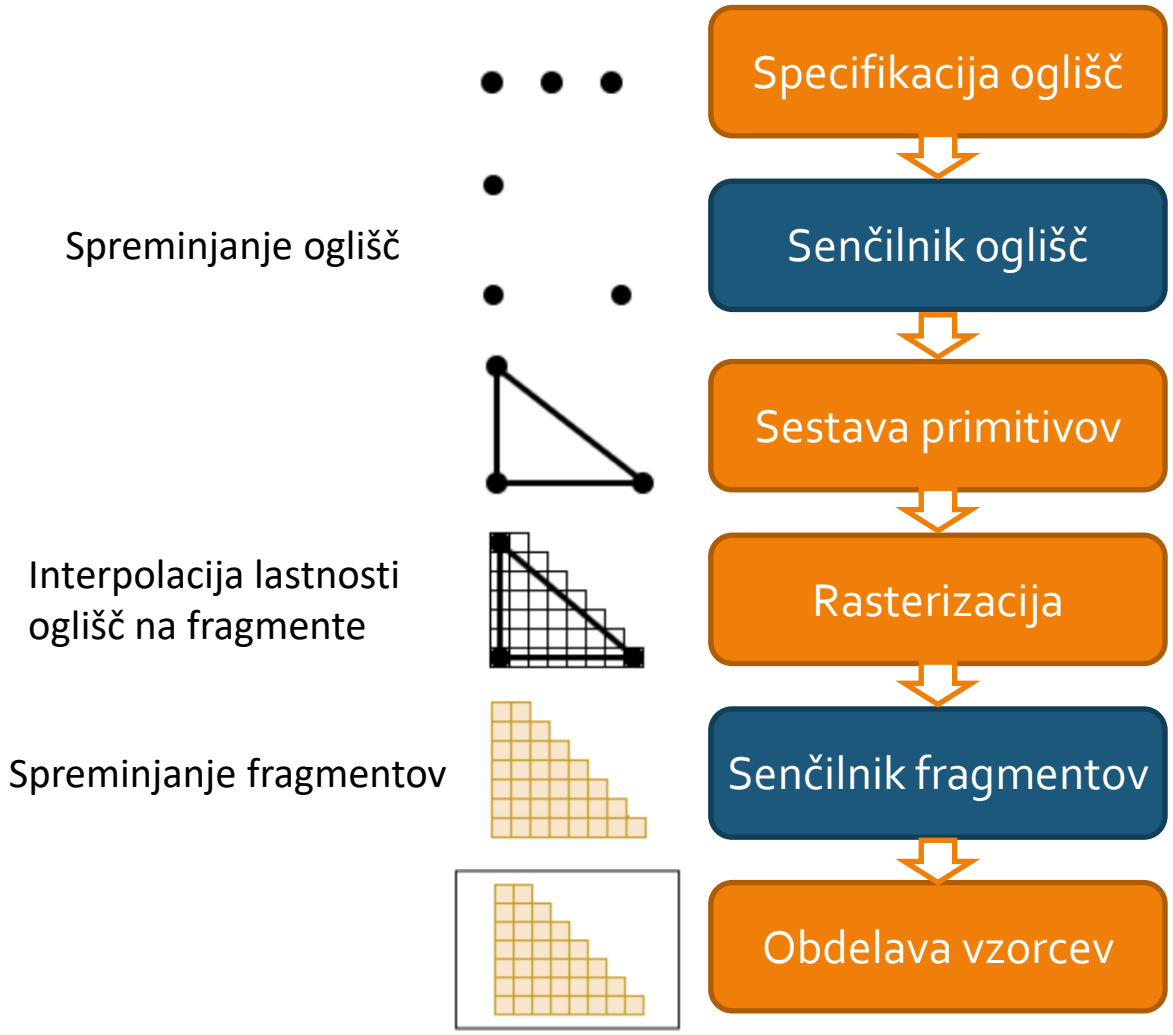


Interpolacija v cevovodu

- Novejše GPE interpolirajo preko **težiščnih** koordinat
  - **interpolacija** lastnosti
  - test **ali je točka v trikotniku**
  - enostavna **paralelizacija**
- Interpolacija lastnosti oglišč na posamezne fragmente se izvede med **rasterizacijo**
- Spremenljivke **interpoliranke**, ki jih so izhodi in vhodi senčilnikov oglišč in fragmentov, se **avtomatsko** interpolirajo
  - z @interpolate lahko dolčimo ali je interpolacija perspektivno pravilna (perspective), navadna (linear) ali je ni (flat)

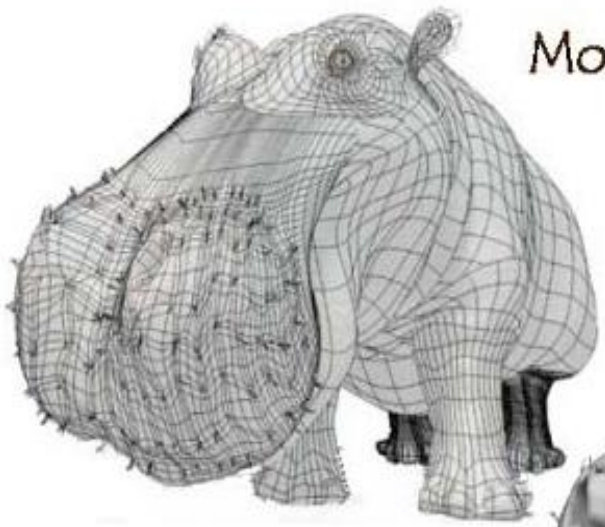
```
struct VertexOutput {
    @builtin(position) position : vec4f,
    @location(0) @interpolate(flat) color : vec4f, // flat linear perspective
}

struct FragmentInput {
    @location(0) @interpolate(flat) color : vec4f, // flat linear perspective
}
```

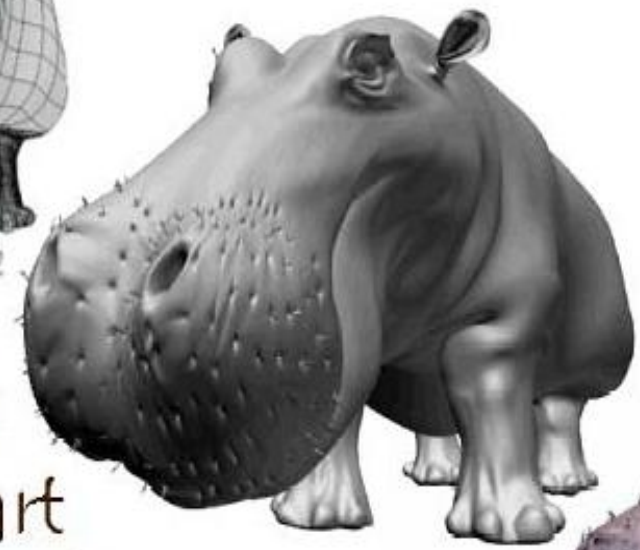


Teksture

The Quest for Visual Realism



Model



Model + Shading

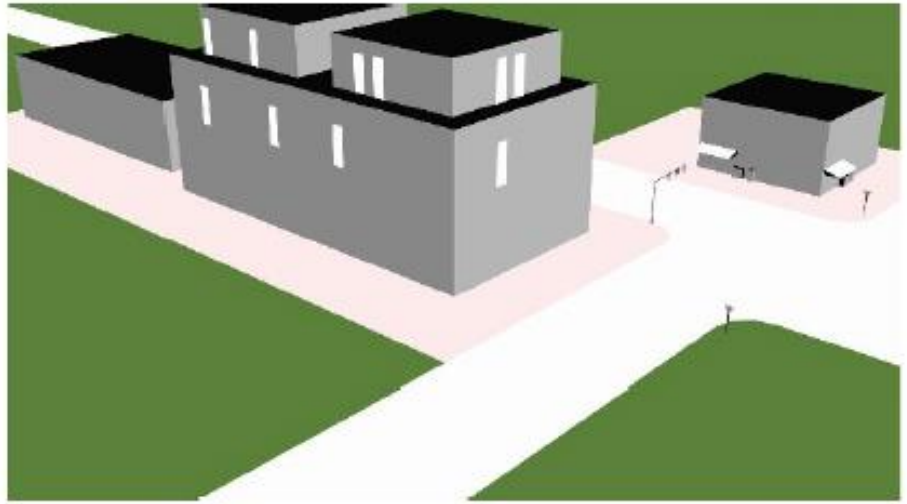
Model + Shading
+ Textures

At what point
do things start
looking real?

- For more info on the computer artwork of Jeremy Birn see http://www.3drender.com/jbirn/productions.html

Teksture

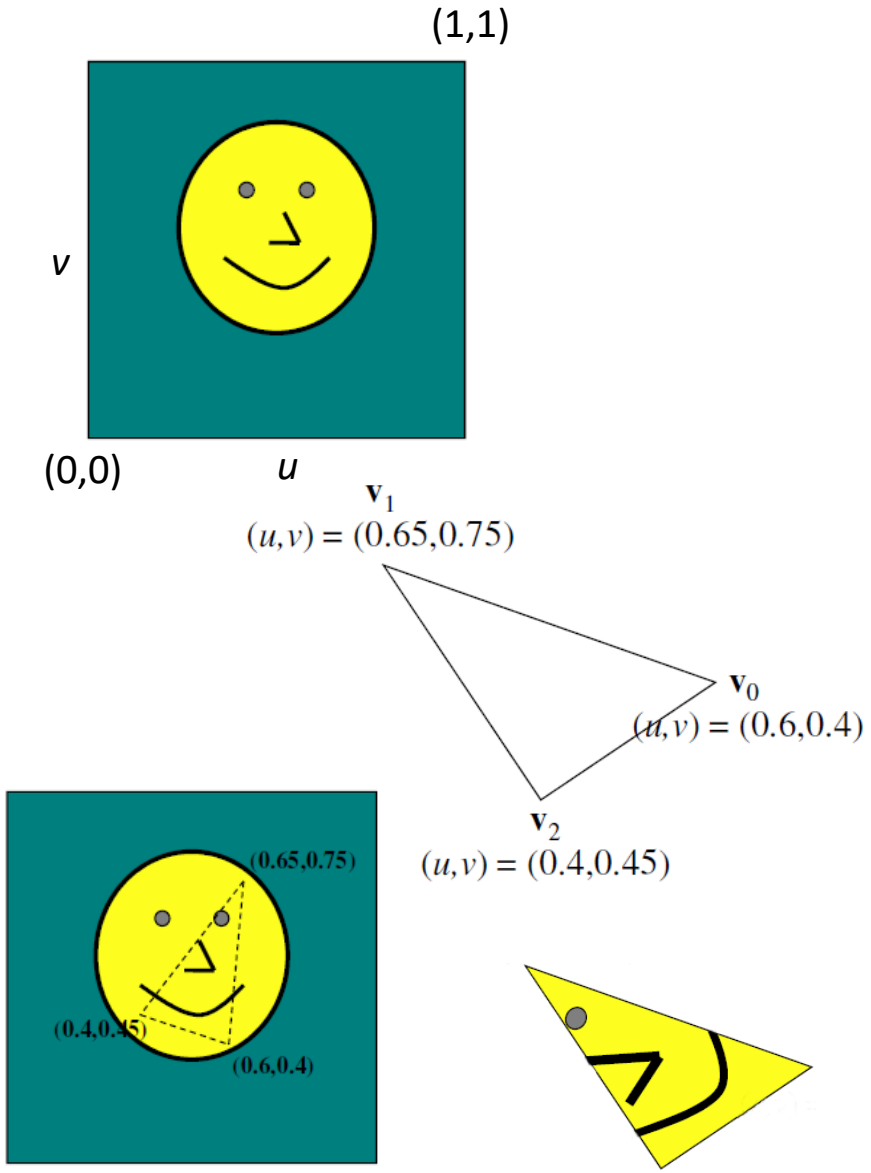
- Enobarvni poligoni – enostavni, hitri, nezanimivi
- Želimo več podrobnosti – na ploskve nalepimo teksture
  - 2D slike (npr. fotografije ali umetno ustvarjene)
  - 3D teksture (serija slik ali proceduralno ustvarjene)
- Tekstura lahko vpliva na barvo, pa tudi na druge parametre – normale, položaj oglišč itn.
  - simulacija materialov
  - zmanjševanje geometrijske kompleksnosti
  - odsevi




2D Teksture

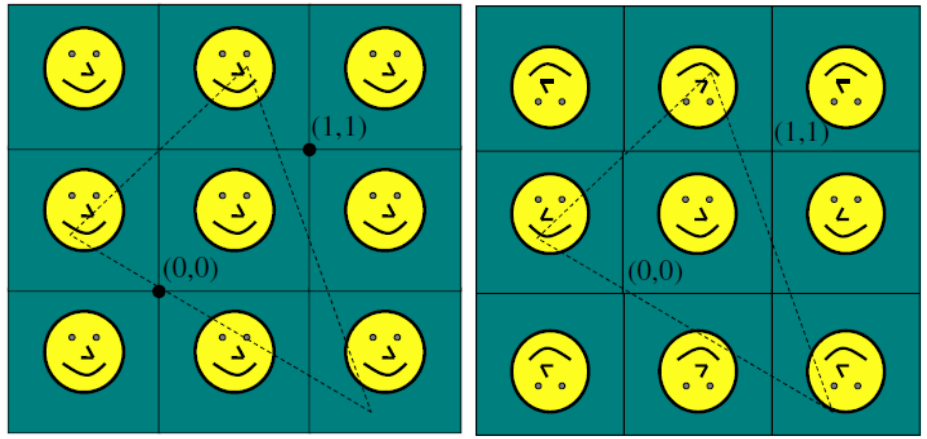
2D teksture

- Kako se 2D tekstura – **bitna slika**
  - preslika na predmet?
- **uv koordinatni prostor** teksture:
  - pikslom teksture (texels) določimo koordinate u, v na območju $[0,1]$
  - levo spodaj je $(0,0)$, desno zgoraj $(1,1)$
- Vsako oglišče trikotnika hrani u, v koordinate dela teksture, ki se nanj preslika

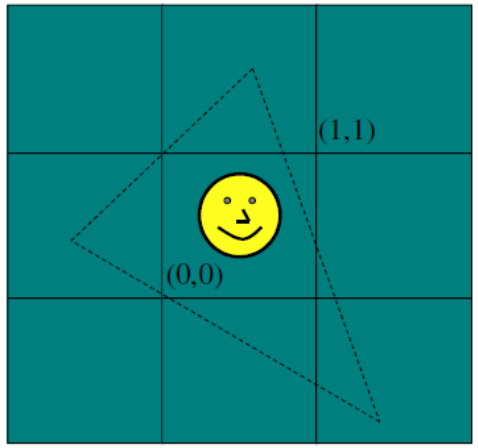


Vrednosti izven območja

- Slika je definirana na intervalu
  - $u, v \in [0,0] \times [1,1]$
- Če u, v koordinate padejo izven tega območja:
  - teksturo **ponavljamo** (*tiling*) ali **zrcalimo** (*mirroring*)
    - pri ponavljanju je smiselno, da so robovi teksture enaki, sicer je prehod viden
  - uporabimo **barvo roba teksture** (*clamping*) ali teksture ne upoštevamo



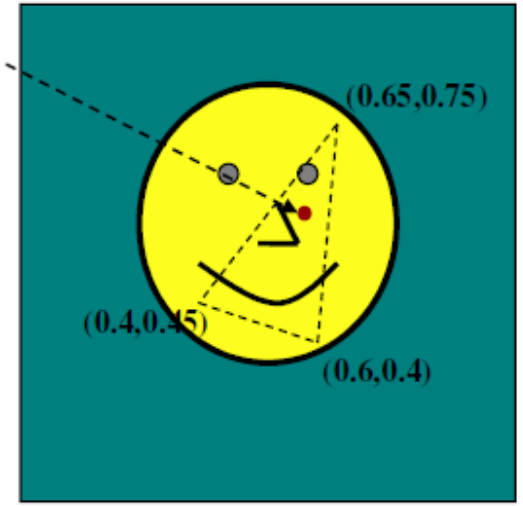
ponavljanje zrcaljenje

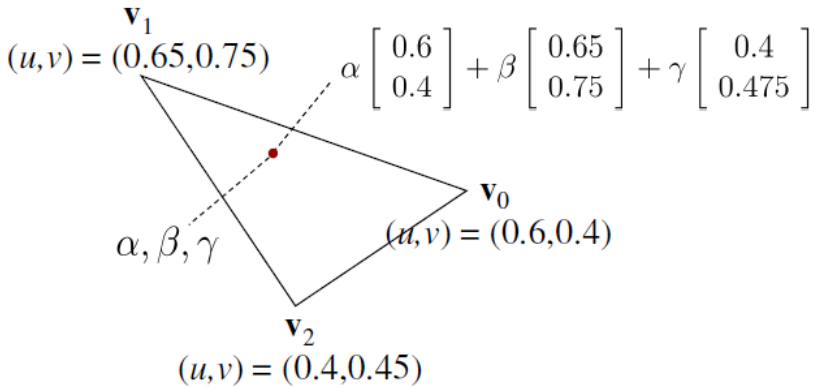


barva roba

Izris 2D tekstur

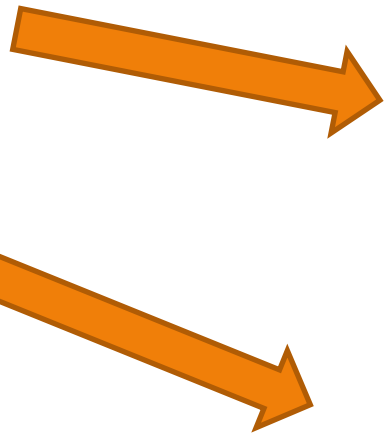
- Za vsako točko (fragment oz. piksel) v trikotniku lahko z **interpolacijo** *uv* koordinat izračunamo v katero točko teksture se preslika
- Interpolacija je standardna **perspektivno pravilna bilinearna** interpolacija
  - uv koordinate oglišč prenesemo v senčilnik fragmentov v interpolirankah

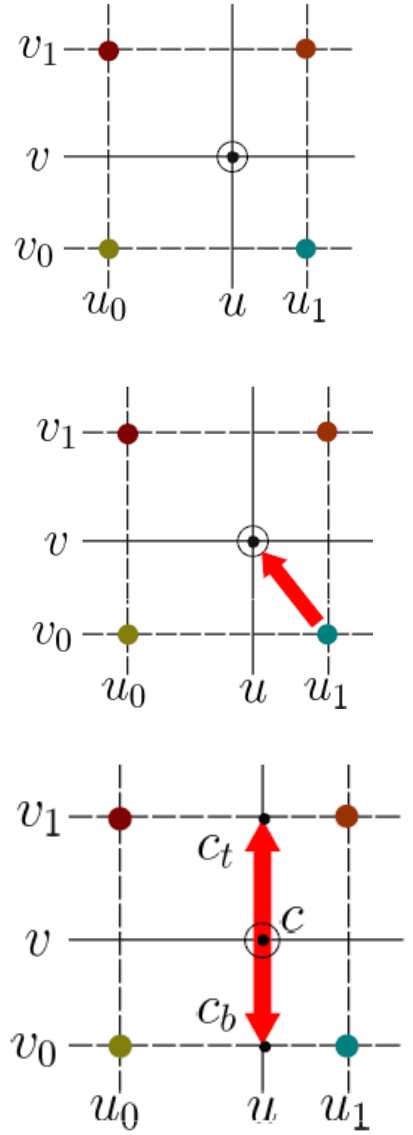




Izris 2D tekstur

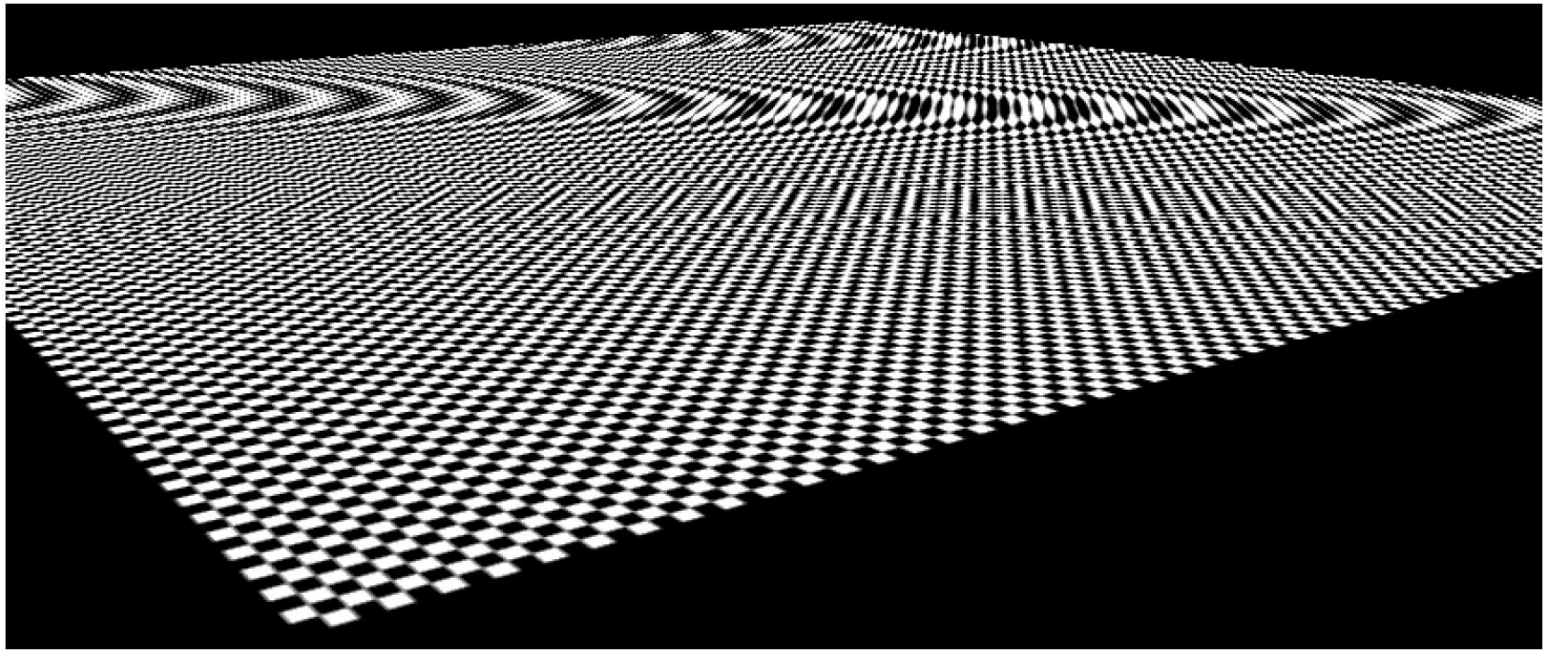
- Dobljene interpolirane u, v koordinate navadno ne padejo “na” teksel
- Kako izračunamo barvo na u, v , če poznamo barvo štirih najbližjih tekslov?
  - **najbližji sosed** (hitro, slaba kvaliteta)
  - **bilinearna interpolacija**
    - najprej izračunamo c_t in c_b, potem c





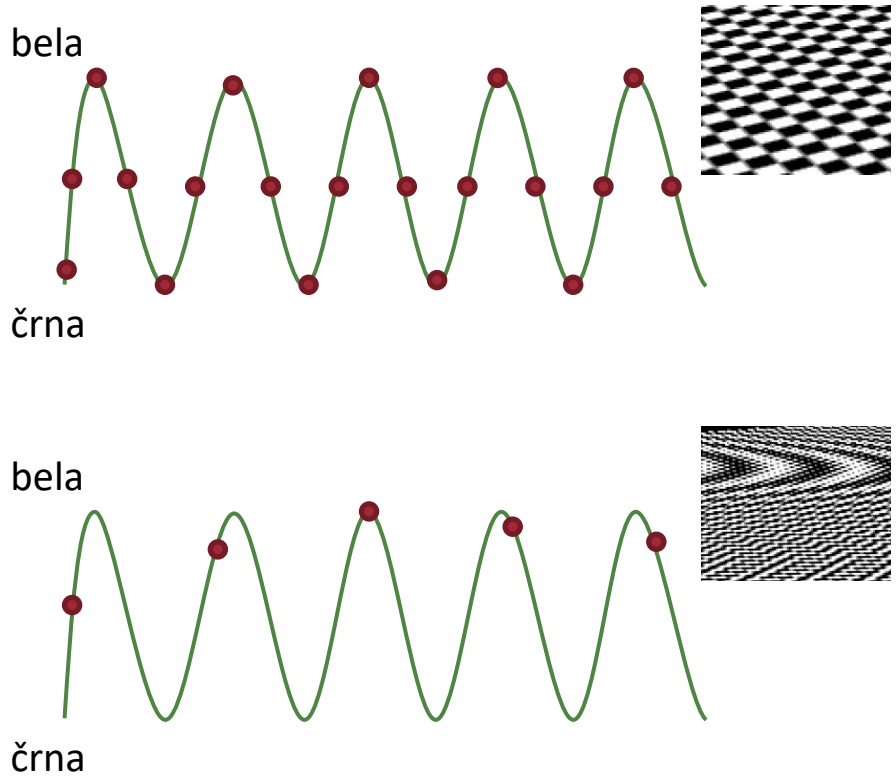
Problem

- Uporabimo perspektivno pravilno interpolacijo za izračun u, v in bilinearno interpolacijo za izračun barve piksla, in …



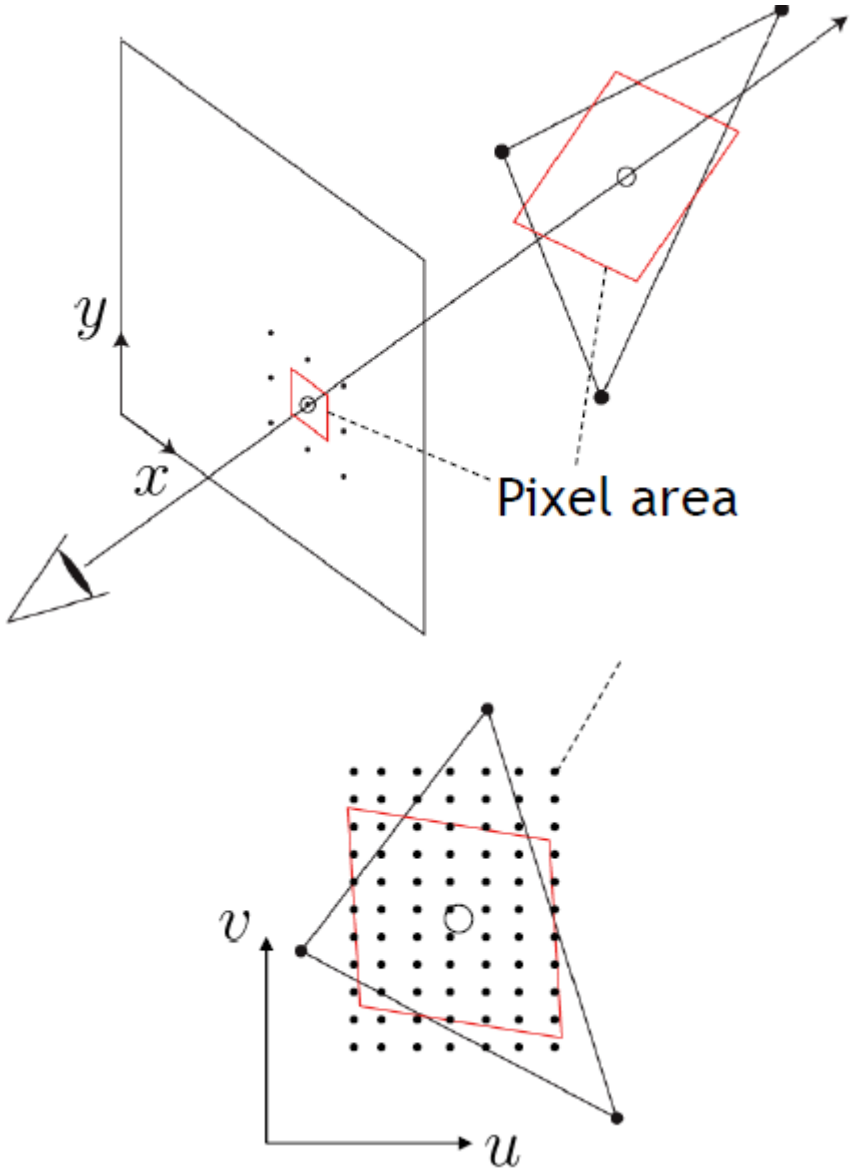
Prekrivanje - aliasing

- Na teksturo gledamo kot na signal:
  - če vzorčimo (izbiramo teksle) dovolj pogosto, je vse v redu
  - če ne vzorčimo dovolj pogosto, hitro menjanje barv (visoke frekvence) postane počasno menjanje barv (nizke frekvence)
    - pojavu, ki ga dobimo z vzorčenjem pri prenizki frekvenci, rečemo **prekrivanje** (*aliasing*)



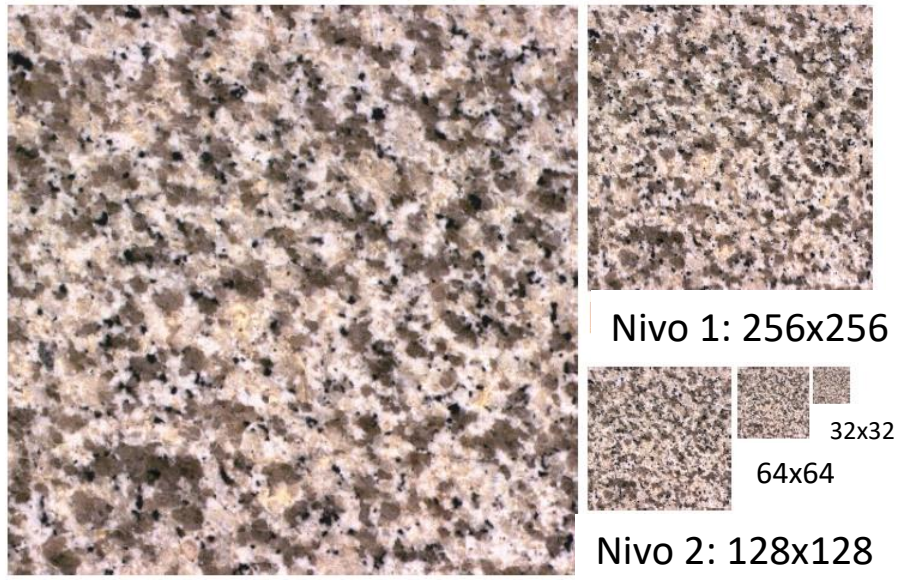
Prekrivanje - aliasing

- Do prekrivanja pride, ker lahko piksel na sliki pokrije **velik kos teksture**
- Izognemo se ga lahko s **povprečenjem tekslov** (nizkoprepustni filter)
  - je pa to počasno – potencialno je potrebno povprečiti veliko tekslov



Prekrivanje - rešitev

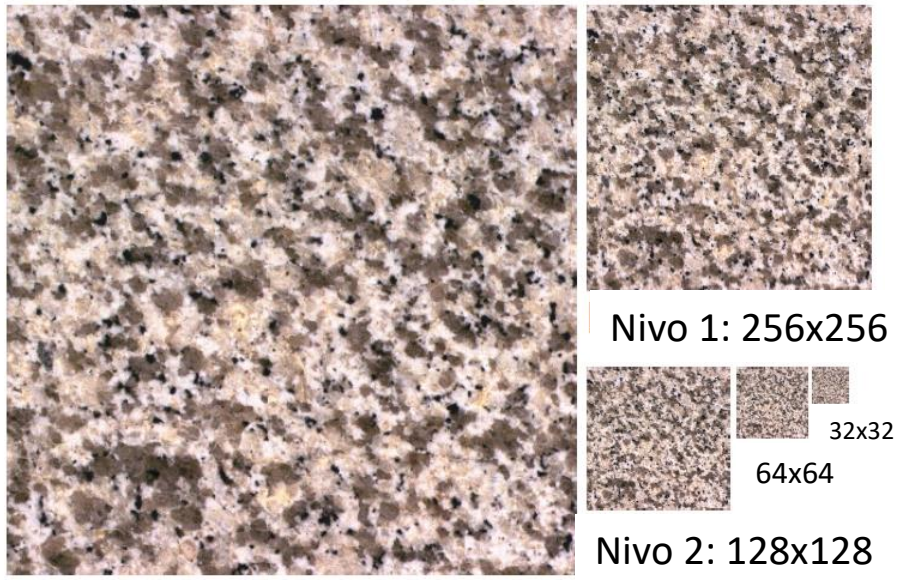
- Povprečenje izvedemo vnaprej - **mipmapping**
  - vnaprej izračunamo več verzij teksture različnih velikosti – mipmaps
    - vsaka verzija je 2x manjša od prejšnje
  - mip = *multum in parvo* = veliko v malem
- Porabimo 1/3 več pomnilnika;
  - $\frac{1}{3} = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \cdots$
- Mipmapping je strojno podprt



1 teksel na nivoju 1 je povprečje
2x2=4 pikslov nivoja 0
1 teksel na nivoju 4 je povprečje
4^4=256 tekslov na nivoju 0

Mipmapping

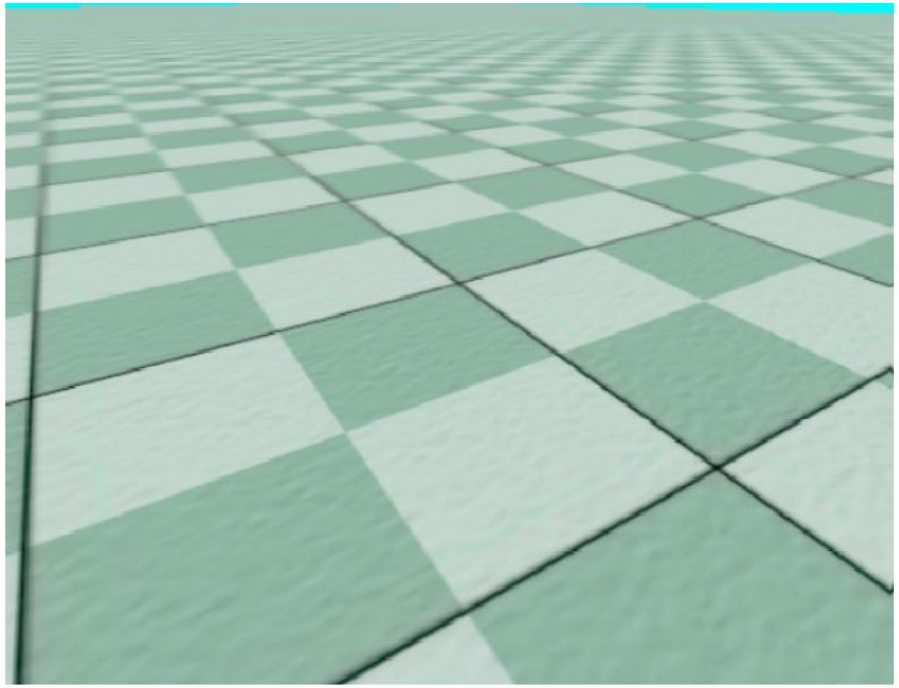
1. Izračunamo koordinate u, v
2. Izračunamo približno **velikost piksla** v teksturnem prostoru
3. Barvo z **bilinearno interpolacijo** izberemo iz ustrezno velike teksture
   - npr. če piksel pokriva 10x10 tekslov, izberemo teksturo tretjega nivoja (8x8)



1 teksel na nivoju 3 je povprečje 4^3=64 tekslov na nivoju 0

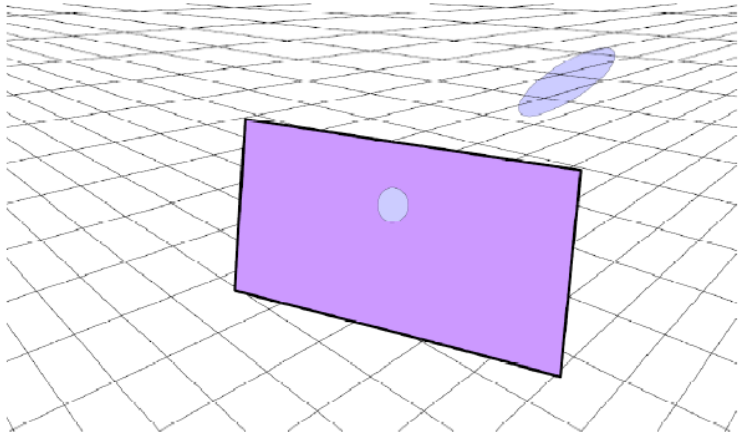
Mipmapping

- Za bolj mehek prehod med nivoji lahko uporabimo dva nivoja – **trilinearna interpolacija**:
  - vzamemo teksturi dveh najbližjih nivojev – npr. za 10x10 vzamemo tretji (8x8) in četrti (16x16) nivo
  - naredimo bilinearno interpolacijo, da dobimo barvo piksla v obeh nivojih
  - linearno interpoliramo še med nivojema

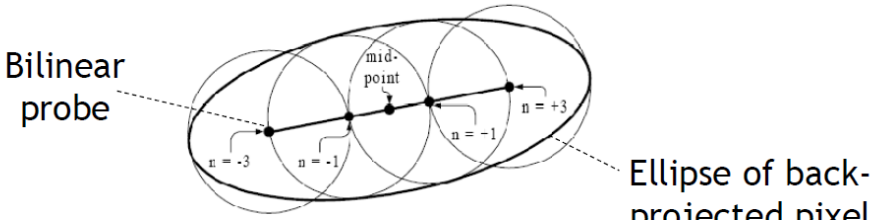


Anizotropično filtriranje

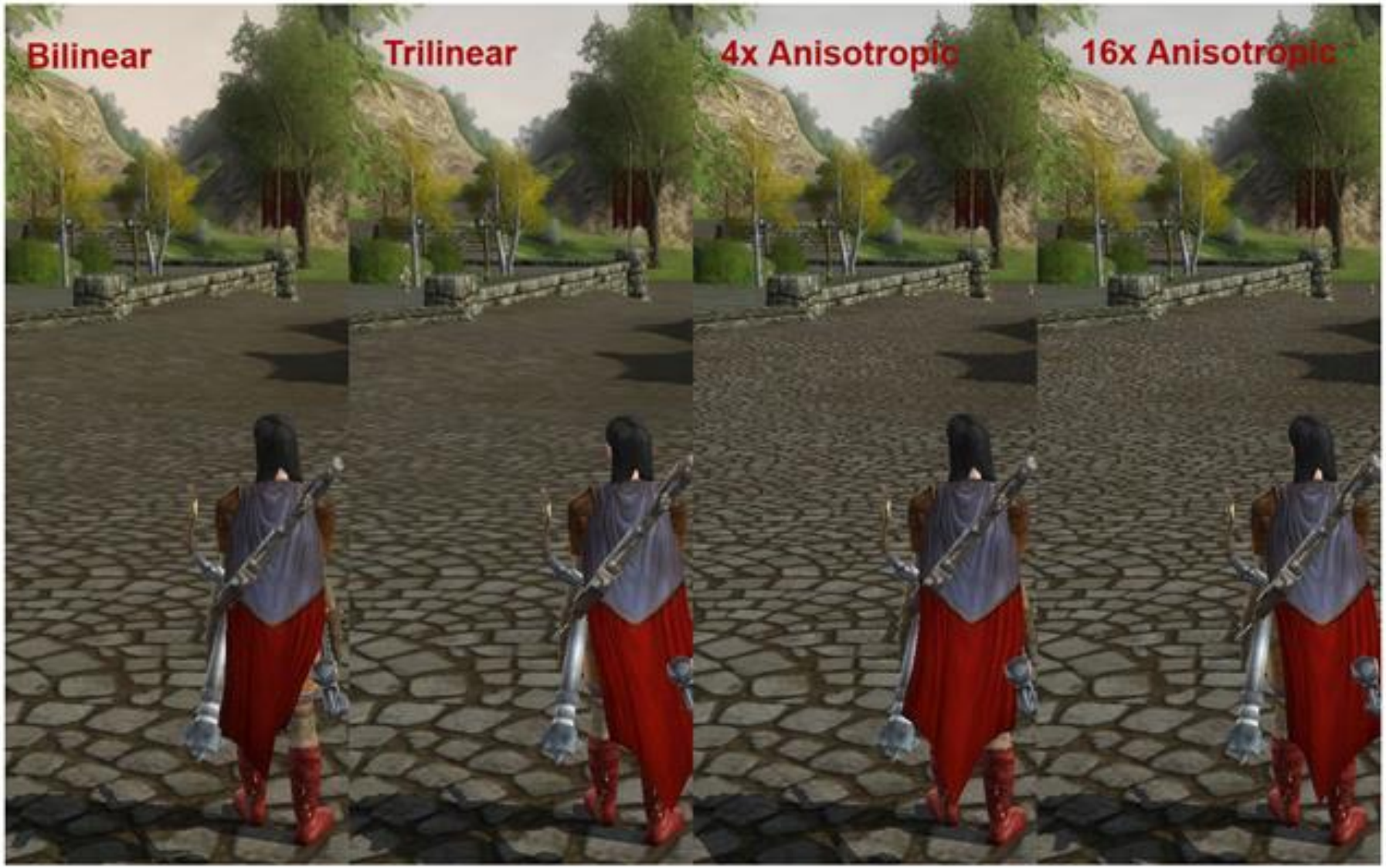
- Mipmapping povpreči teksturo v vseh smereh enako, kot da je piksel kvadrat
  - piksel pa navadno **ni kvadrat** v teksturnem prostoru
    - kadarkoli ne gledamo naravnost, ampak pod kotom
  - mipmapping “preveč” povpreči v teh primerih
- Anizotropično filtriranje povpreči teksturo v tekslih znotraj **okna, izračunanega za vsak piksel**
  - okno je odvisno od orientacije ploskve in je boljši približek realnosti
  - povprečimo določeno število tekslov znotraj tega okna (npr. 16)
  - boljša kvaliteta od mipmappinga, bolj zahtevno, predvsem glede pretoka podatkov (večkrat dostopamo do tekslov)
    - npr. 16x trilinearno anizotropično filtriranje dostopa do 128 tekslov (16 vzorcev * 4 mipmap * 2 nivoja)
  - je strojno podprto



piksel ni nujno kvadrat v teksturnem prostoru



Anizotropično filtriranje



http://www.geforce.com/whats-new/guides/aa-af-guide

WebGPU in teksture

- *sampler* določa način ponavljanja teksture, interpoliranja in nastavitve mipmappinga in anisotropičnega filtriranja

```
const sampler = device.createSampler({
   magFilter: 'linear’,  // nearest, linear
   minFilter: 'linear’,  // nearest, linear
   addressModeU: "repeat",  // repeat, mirror-repeat
   addressModeV: "repeat",  // repeat, mirror-repeat
   mipmapFilter: "linear",  // nearest, linear
   lodMinClamp: 0,  // min mipmap level
   lodMaxClamp: 32,  // max mipmap level
   maxAnisotropy: 4,  // number of anisotropy samples
});
```

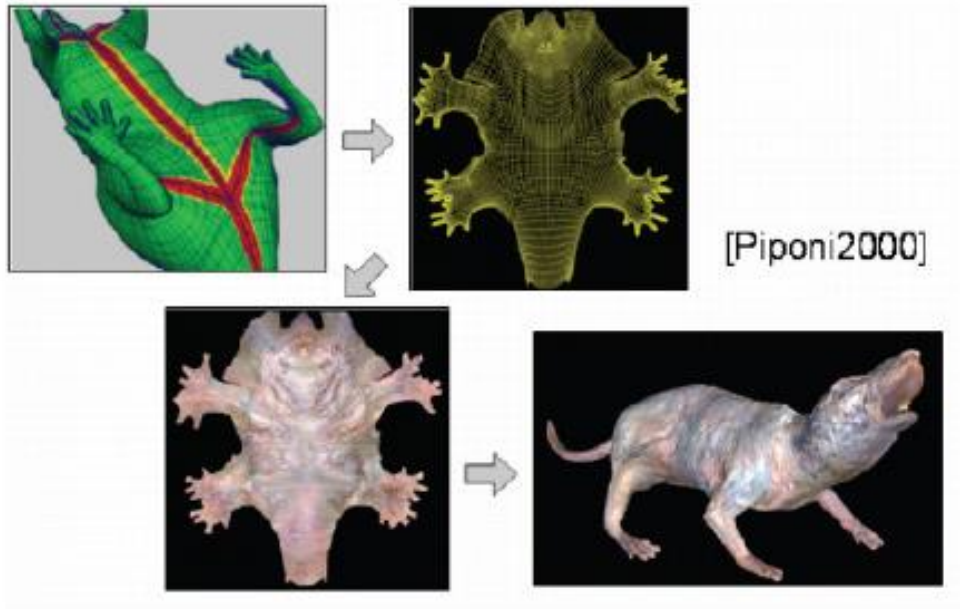
Lepljenje tekstur

Lepljenje tekstur

- Kako dobimo u, v koordinate za izris tekstur, če imamo model?
  - preslikava med 3D položajem teksture na predmetu in 2D koordinatami tekstur
- Oglišča 3D modela **razprostremo** v u, v prostor (kot kožo)
  - (pol)avtomatsko ali čisto ročno
- Teksturo lahko **projiciramo** na 3D model in izračunamo u, v koordinate

Kožno lepljenje

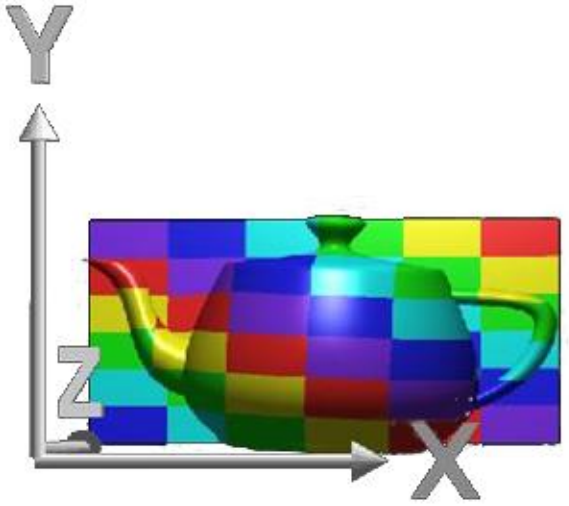
- Površino predmeta razvijemo v 2D – kožo
  - kompleksen problem
- Ročno določimo položaj delov modela v u, v prostoru
- Kožo pobarvamo, nalepimo nazaj

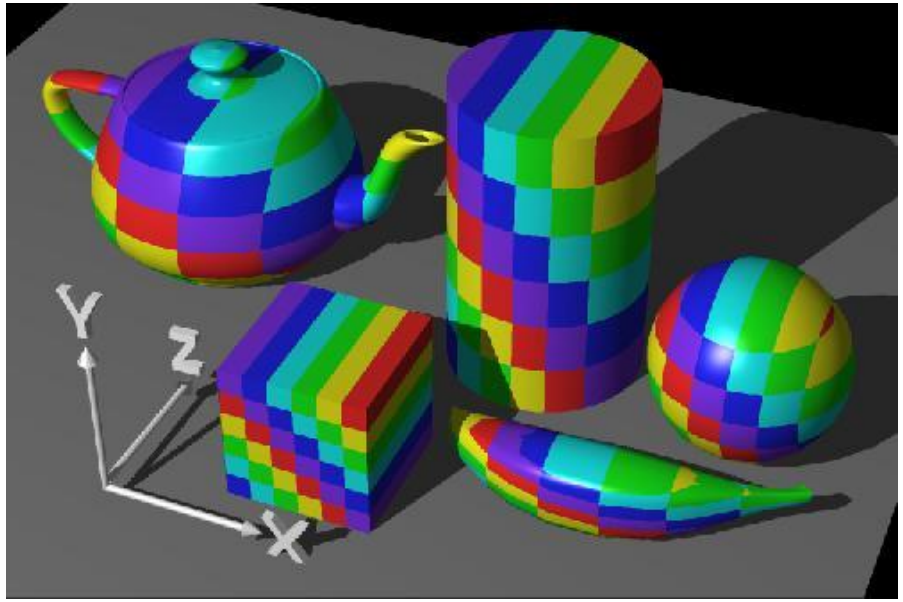


Vzporedno (planarno) lepljenje

- Uporabimo **linearno transformacijo** xyz koordinat predmeta
- Primer:

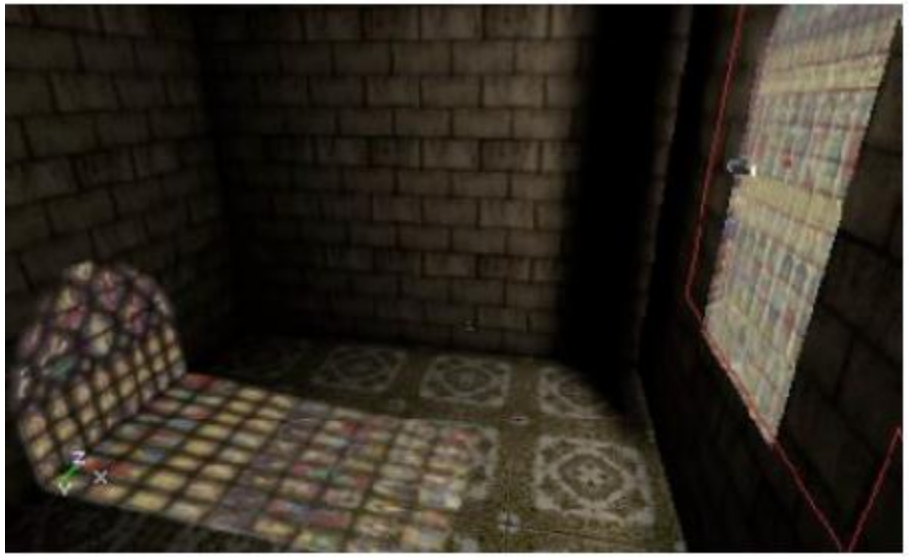
$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$





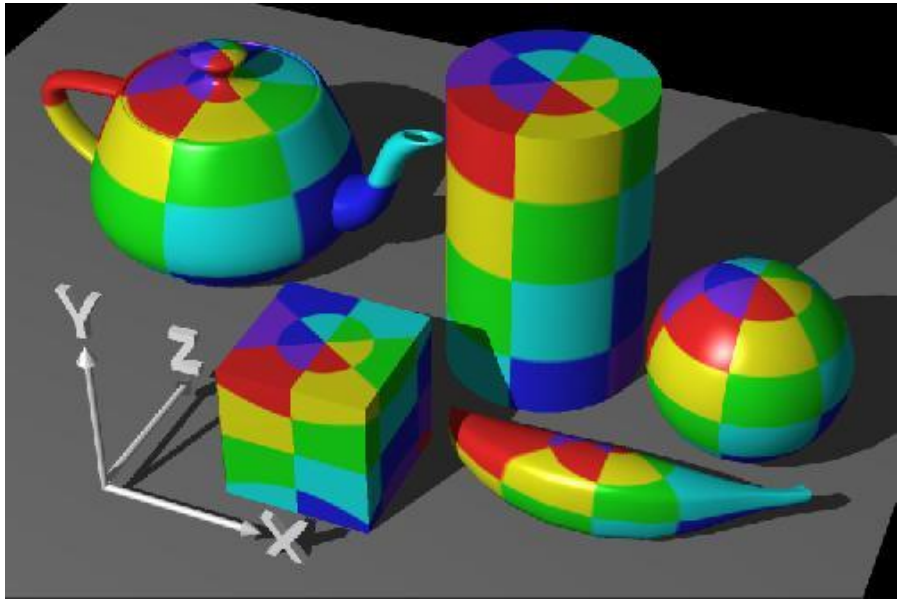
Perspektivno lepljenje

- Uporabimo **perspektivno projekcijo** *xyz* koordinat predmeta
- Uporabno npr. za efekte osvetlitve



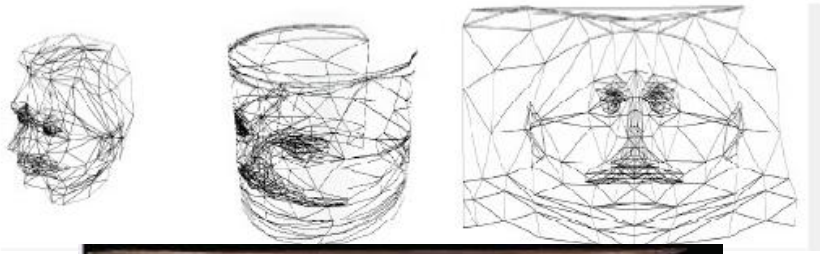
Sferično lepljenje

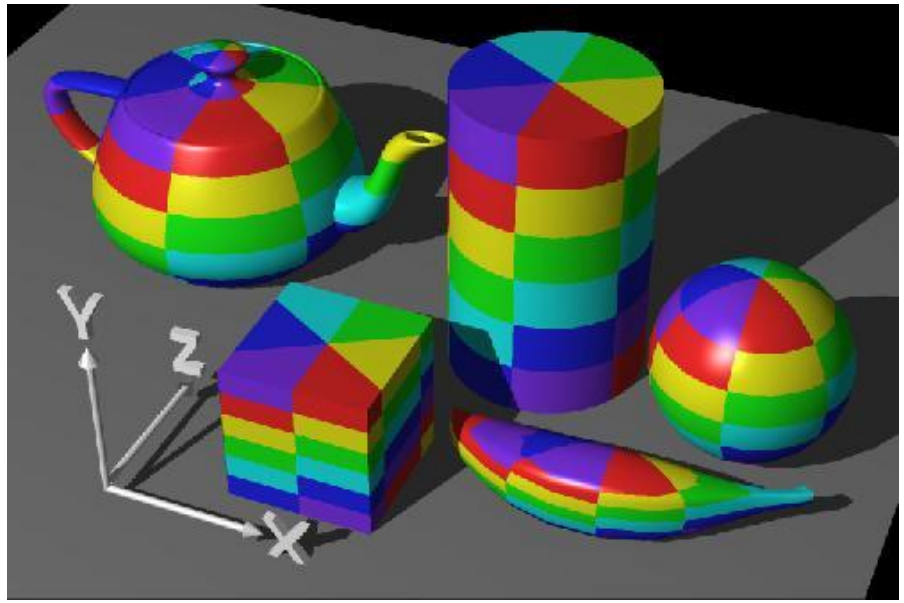
- Uporabimo **sferične koordinate**
- Kot bi predmet postavili v kroglo in teksturo s krogle nalepili na predmet
- Izračun u, v v točki **p**, če je krogla z radijem R v središču **c** :
  - krogelne koordinate p-ja:
    - $\cos\theta = \frac{p_y - c_y}{R}, \tan\phi = \frac{p_z - c_z}{p_x - c_x}$
  - u,v coordinate točke p:
    - $u = \frac{\phi}{2\pi}, \; v = \frac{\pi - \theta}{\pi}$



Cilindrično lepljenje

- Podobno kot sferično, teksturo mapiramo preko valja
  - $\tan\phi = \frac{p_z - c_z}{p_x - c_x}, u = \frac{\phi}{2\pi}$
  - $v \simeq y$

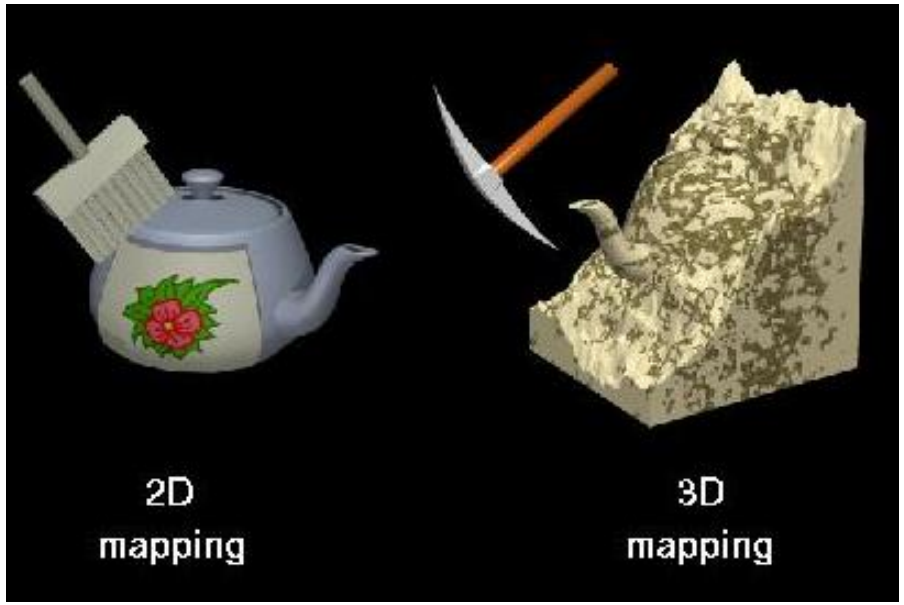





3D teksture

3D teksture

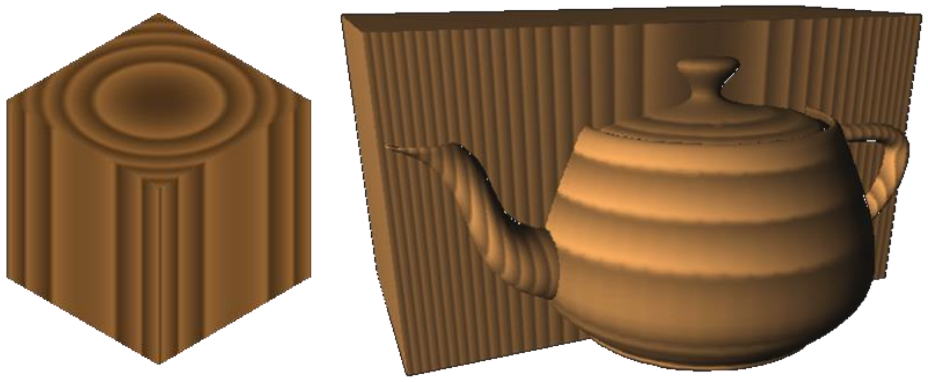
- Simuliramo predmet narejen **iz nekega materiala**
- 3D tekstura – definirana v treh dimenzijah
  - serija slik (ena nad drugo)
  - velikokrat je specificirana **proceduralno**
- Nanjo lahko gledamo kot na funkcijo
  - $f(s,t,r)$, ki vrne barvo v neki 3D točki
- Lahko uporabimo **neposredno 3D** lepljenje
  - $(x,y,z) = (s,t,r)$
  - če je funkcija f definirana le v diskretnih točkah, naredimo interpolacijo (npr. trilinearno)



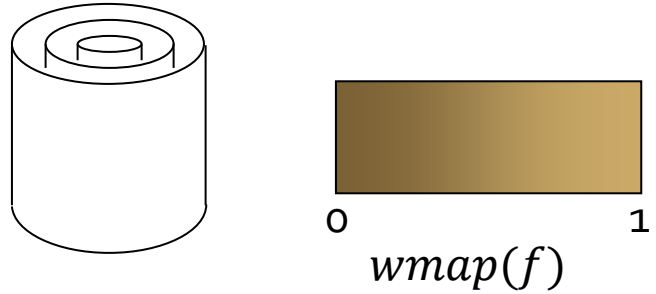
Proceduralne 3D teksture

- Uporabljamo za naravne materiale – les, kamen, oblaki, dim …
- Osnova je **funkcija *f(s,t,r)***
  - funkcija lahko vrne barvo, prosojnost, normale …
- Preprost primer: les
  - barva se spreminja glede na oddaljenost od središča - $(s^2 + t^2)$
  - $f(s,t,r) = wmap((s^2 + t^2)\ mod\ 1)$
    - barvna tabela $wmap$

      0 = temno rjava, 1 = svetlo rjava



$wood(s,t,r)$

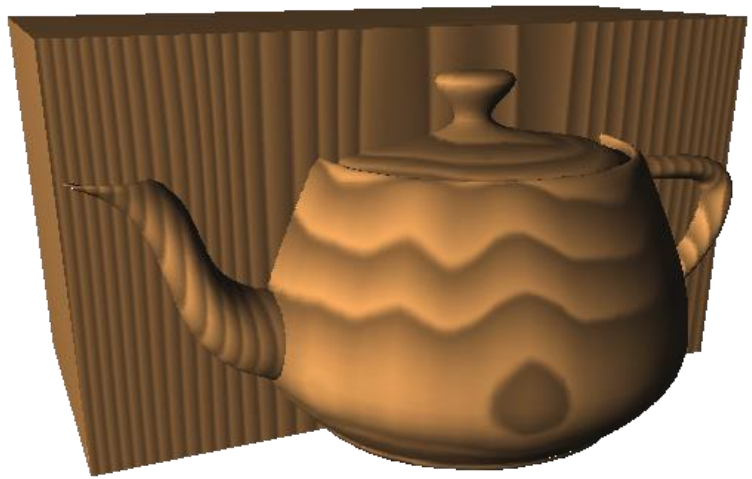


$wmap(f)$

$f(s,t,r) \approx (s^2 + t^2)\ mod\ 1$

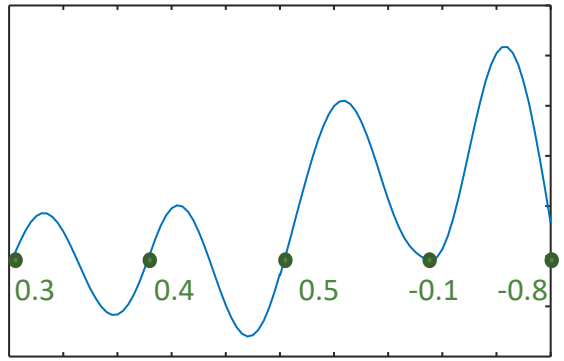
Proceduralne 3D teksture

- Dodamo **šum**, izboljšamo realizem
  - $wmap\left(\left(s^2 + t^2 + N(s,t,r)\right) mod\ 1\right)$
  - N je šumna funkcija

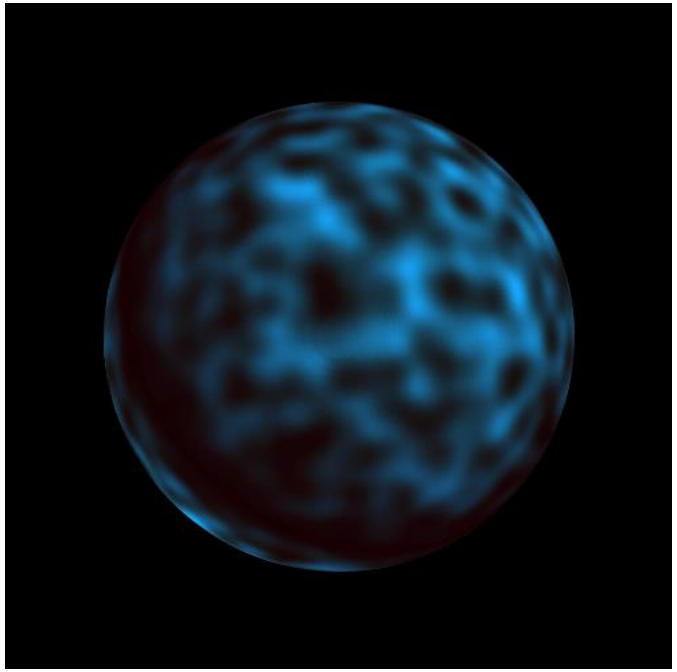


Šum

- Proceduralne teksture navadno vsebujejo komponento šuma
  - večji realizem
- **Perlinov šum** (Perlin noise)
  - $\mathbb{R}^n$ šumna funkcija
  - **gradientni šum** – interpolacija med naključnimi gradienti, mehki prehodi
  - veliko se uporablja, ker je “mehka” funkcija, omogoča enostaven nadzor nad **frekvenco** in **fazo**



ustvarimo naključne gradiente, interpoliramo vmes



Perlinov šum na krogli

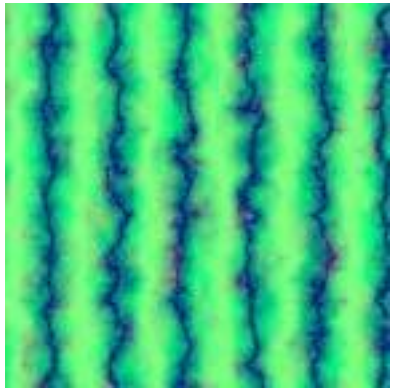
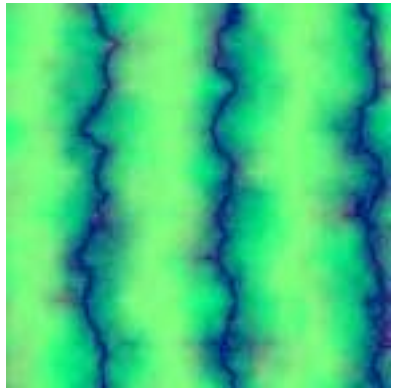
Turbulenca

- V naravnih teksturah velikokrat srečamo neko regularnost, ki se pojavlja na več velikostnih nivojih
  - lahko jo dobimo s seštevanjem skaliranih šumov
- **Turbulenca** = vsota skaliranih Perlinovih šumov
  - oz. vsota šumnih *oktav*:
  - $noise(f) + \frac{1}{2}\, noise(2f) + \frac{1}{4}\, noise(4f) + \cdots$
    - f: frekvenca šuma

$$\Sigma\left(\frac{1}{n^i} f(n^i)\right)$$

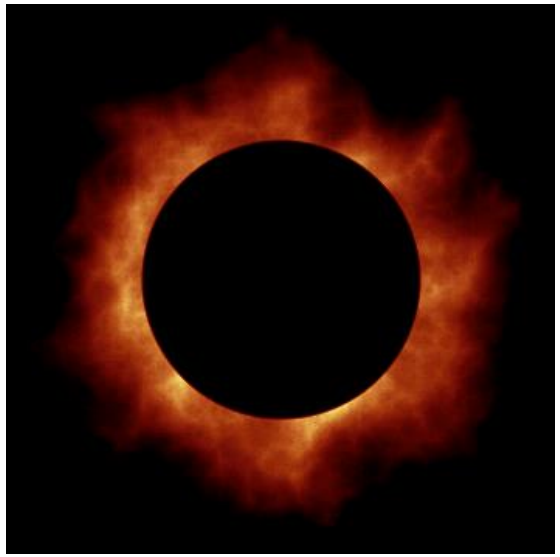
$$\Sigma\left(\frac{1}{n^i} f(|n^i|)\right)$$

Turbulenca - primeri

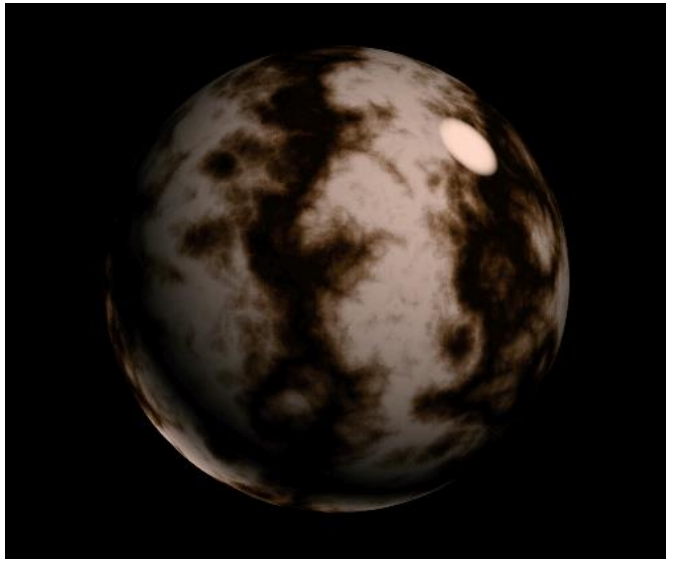



marmor


oblaki



korona



marmor – $\sin(f * (x + A * Turb(x, y, z))$

REFERENCE

- Ken Perlin: Making Noise
- J.P. Schulze: Introduction to Computer Graphics (slides)
- N. Guid: Računalniška grafika, FERI Maribor
- J.D. Foley, A. Van Dam et al.: Computer Graphics: Principles and Practice in C, Addison Wesley