Računalniška Grafika in Tehnologija Iger

1. Geometrija

1.1. Transformacija modela

- Linearno algebro potrebujemo za opis položaja, orientacije, gibanje objektov, položaj glede na kamero, projekcijo 3D scene na 2D ravnino
- **Linearna kombinacija** je vsota n skalarnih vektorjev
- Vektorji so linearno odvisni, če lahko enega izmed njih zapišemo kot linearno kombinacijo ostalih (vsota skalarnih vektorjev)
- Bazni vektorji so linearno neodvisni vektorji dolžine 1, ki so pravokotni drug na drugega
- Z določitvijo izhodišča in baznih vektorjev lahko določamo položaje točk v prostoru (za bazne vektorje v R^n uporabimo n linearno neodvisnih vektorjev)
- Imamo dve varianti baznih vektorjev – **levosučni** (DirectX) in **desnosučni** (OpenGL, pravilo roke)
- Točka plus vektor je premaknjena točka, razlika dveh točk pa je vektor
- Velikost vektorja

$$|v| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

- **Enotski vektor** je velikosti 1, dobimo ga z normiranjem (delimo vektor z njegovo dolžino)
- Skalar med dvema vektorja nam pove **kot** med njima in **dolžino** projekcije na vektor

$$a \cdot b = \sum_i a_i b_i$$

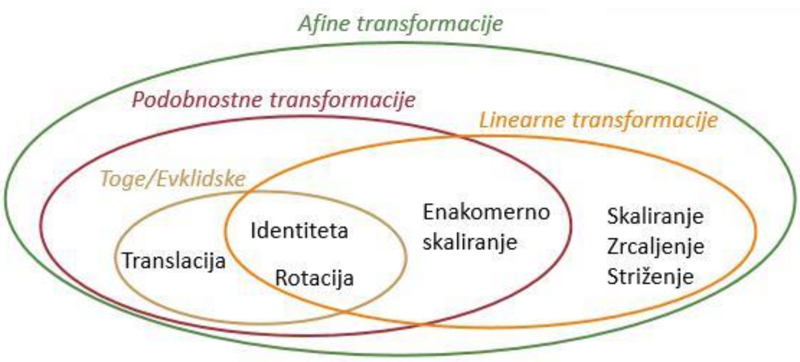
$$a \cdot b = |a||b| \cos\theta$$

Skalarni produkt potrebujemo za računanje osvetlitve (kot vpadne svetlobe), za detekcijo trkov (pod kakšnim kotom se zaletita), katera stran poligona gleda proti kameri...

- **Vektorski produkt** nam da pravokoten vektor na a in b v desnosučnem k. s. in je pomemben pri iskanju normal (ni komutativen)

$$a \times b = \begin{bmatrix} a_y b_z & a_z b_y \\ a_z b_x & a_x b_z \\ a_x b_y & a_y b_x \end{bmatrix}$$

- Na primer imamo poligon z točkami a, b in c, za izračun normale uporabimo: n = a X b
  Smer postavitve točk določa smer normale
- **Transformacija** je funkcija, ki preslika eno konfiguracijo točk v drugo – translacija, rotacija, skaliranje. Spremenijo usmerjenost, velikost, položaj v prosotru. Transformacija vzame točko ali vektor in jo preslika v neko drugo točko ali vektor.
  P = (x, y) v originalu gre v p' = (x', y') v transformiranem prostoru
  Kamera ima položaj in usmeritev (translacija, rotacija)
- Poznamo projekcijske (ohranjajo črte – afine transformacije) in nelinearne transformacije



1.2. Afine 2D transformacije

- **Translacija** je preprosto prištevanje odmika **P' = P+T**

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- **Skaliranje** je množenje koordinat s faktorji skaliranja, poteka okoli središča sistema, če je s_x različen od s_y dobimo neenakomerno skaliranje **P' = S*P**
  Za skaliranje okoli poljubne točke moramo transformacije sestaviti – premaknemo točko skaliranja v središče KS, skaliramo in nato premaknemo nazaj, saj se v nasprotnem primeru predmet lahko premakne
- **Rotacija** okoli središča KS je enako, kot da bi rotirali KS v obratni smeri – pozitivni koti so v nasprotni smeri urinega kazalca (desnosučno) **P'=R*P**

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

  Za rotacije okoli poljubne točke je enako kot pri skaliranju – središče.
- Problem pri teh operacijah je, da so heterogene (seštevanje in množenje) – želimo imeti samo množenje, da jih lahko združujemo. Zaradi tega problema smo uvedli matematični konstrukt: homogene koordinate
- Da lahko vse afine transformacije pretvorimo v množenje matrik uvedemo **dodatno koordinato** za zapis točk – za točke postavimo w=1 in za vektorje w=0.
  Afine transformacije ne spreminjajo vrednosti w, medtem ko projekcijske jo
- Za pretvorbo iz točke v homogenih koordinatah v običajne koordinate dobimo z deljenjem z w
- **Translacija postane množenje**, skaliranju in rotaciji pa le dodamo stolpec in vrstico z enico na diagonali, drugod pa ničle

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x+dx \\ y+dy \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- To omogoča enostavno sestavljanje transformacij kjer je važen vrstni red (množenje ni komutativno) – vrstni red od **desne proti levi**; Sestavljene afine transformacije predstavimo z množenjem matrik; Tipična sestava osnovnih treh transformacij je **T*R*S*x = M*x**
- Za **zrcaljenje** preko osi KS uporabimo skaliranje z negativnimi koeficienti; Za zrcaljenje preko poljubne osi pa naredimo najprej rotacijo, nato zrcaljenje in na koncu ponovno rotacijo
- **Striženje** raztegne predmet v odvisnosti od oddaljenosti od osi in je mešanica skaliranj in rotacij. h je faktor striženja v smeri x, g pa faktor striženja v smeri y.

1.3. Afine 3D transformacije

- Pri uvedbi transformacij v **3. dimenzijo** je edini problem pri rotaciji, kjer so možne tri matrike – za rotacijo okoli vsake osi; Rotacija predmeta je tako rotacijo preko vseh treh osi
- Pri translaciji in skaliranju dodamo še eno vrstico 0 0 0 1 in z pri rotaciji pa

$$X: \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} & & & \\ & \cos\theta & -\sin\theta & 0 \\ & \sin\theta & \cos\theta & 0 \\ & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad Y: \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & & \sin\theta & 0 \\ & & & \\ -\sin\theta & & \cos\theta & 0 \\ 0 & & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad Z: \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & & 0 \\ \sin\theta & \cos\theta & & 0 \\ & & & \\ 0 & 0 & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- **Eulerjevi koti** predstavljajo kote rotacij okrog koordinatnih osi, da dosežemo želeno usmeritev – celotno matriko rotacije zapišemo kot produkt matrik (pomemben vrstni red)

- Pri rotaciji z Eulerjevimi koti lahko pride do **kardanske zapore – Gimbal Lock**, ki je problem, ki nastane, če se dve osi poravnata med seboj – tako izgubimo prostostno stopnjo in se predmet ne bo vrtel, kot smo si zamislili. Nastal bo problem pri spremembah in animaciji rotacij.
- Kot alternativo Eulerjevim kotom pa lahko rotacijo predstavimo z osjo okoli katere je predmet rotiran in kotom rotacije. Ločeno premikamo os in kot. Pri animaciji si lahko pomagamo s **kvaternioni** (lahko pretvarjamo z Eulerjevimi koti). Prehod iz začetne v končno rotacijo naredimo z interpolacijo (najboljši rezultat je sferična linearna interpolacija na krogli).
  Kvaternioni so 4D kompleksno število. Splošna oblika je: $q = s + xi + yj + zk$
  $i^2 = j^2 = k^2 = ijk = -1$
  $ij = k, jk = i, ki = j,$
  $ji = -k, kj = -i, ik = -j$
  Rotacijo okoli n predstavimo kot:
  Točko p kot kvaternion pa predstavimo z (0, p) in jo zavrtimo kot:
- **Normale** so definirane kot vektor pravokoten na drug vektor oz. ploskev in se ne transformirajo enako kot točke – če točko transformiramo z M moramo normalne transformirati z $(M^{-1})^T$

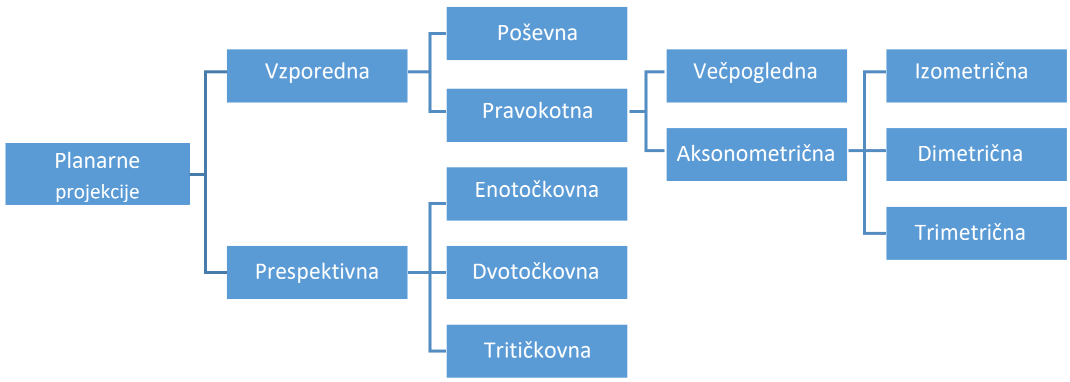
$$M = \begin{bmatrix} 1 & 1.5 \\ 0 & 1 \end{bmatrix} \quad \Rightarrow \quad (M^{-1})^T = \begin{bmatrix} 1 & 0 \\ -1.5 & 1 \end{bmatrix} \qquad \text{nova normala: } (\mathbf{M}^{-1})^T\mathbf{n} = \begin{bmatrix} 1 & 0 \\ -1.5 & 1 \end{bmatrix}\begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1.5 \end{bmatrix}$$

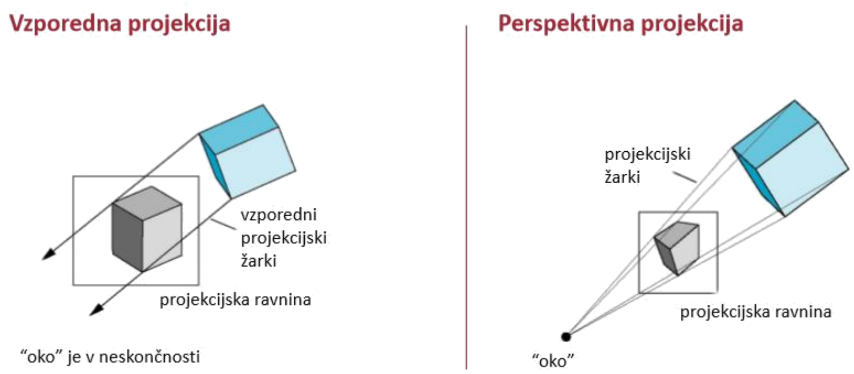
1.4. Koordinatni sistem

- 3D scena je sestavljena iz predmetov v prostoru, na katerega gledamo skozi kamero, ki je tudi postavljena v prostoru
- Poznamo **lokalne koordinate, koordinate sveta in koordinate kamere**
- Posamezen predmet je narejen relativno glede na svoj lokalni koordinatni sistem, ki je navadno v središču predmeta; Lokalne koordinate nato določajo položaj točk glede na središče lokalnega KS
- Predmeti so preko **transformacije modela M** postavljeni v svet – točke se iz lokalnih koordinat transformirajo v koordinate sveta s pomočjo afinih transformacij (M =T*R*S)
- Koordinatni sistem kamere je navadno X-Y ravnina vzporedna z ravnino projekcije – to je ravnina, na katero se projicira slika, ki jo vidimo, če pogledamo skozi kamero.
  Predmete iz koordinat sveta v koordinate pogleda preslikamo s **transformacijo pogleda V**
- Položaj kamere je določen s **transformacijo kamere** ($T_K * R_K$); Za izračun transformacije pogleda lahko uporabimo obratno transformacijo transformacije kamere ($T_k^{-1} * R_k^{-1}$)
- Da preslikamo predmet iz lokalnih koordinat v koordinate pogleda moramo torej matriko točk P preslikati s transformacijo modela, nato pa še z transformacijo pogleda
  $P' = VMP = R_k^{-1} * T_k^{-1} * T_m * R_m * S_m * P = X*P$
- V WebGL izračunamo transformacijo pogleda kot premik kamere z inverznima translate in rotate - lookAt definira položaj kamere, točko pogleda in nagib kamere.

2. Projekcije

- Ko predmet preslikamo v koordinate kamere je še vedno 3D – naredimo planarno projekcijo, ki je preslikava iz 3D v 2D
- Planarna projekcija je projekcija na ravnino, kjer se ohranijo črte, ne pa tudi nujno koti
- Glede na namen jih delimo na vzporedne, kjer so projekcijski žarki vzporedni, in na perspektivne, kjer projekcijski žarki konvergirajo v točko.





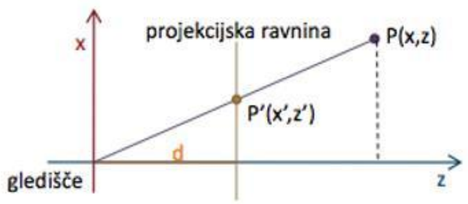
2.1. Vzporedne projekcije

- Pri **pravokotni (ortografski)** projekciji so žarki pravokotni na ravnino projekcije. Predmeti daleč izgledajo enako veliki kot tisti blizu (ni efektov perspektive)
- Pri **večpogledni** projekciji je projekcijska ravnina vzporedna z osnovnimi pogledi na predmet (tloris, naris in stranski ris). Ohranja (+) razdalje, kote in oblike, (–) vendar se ne vidi kako predmet zares izgleda, saj je veliko ploskev zakritih. Lahko jo uporabljamo za meritve.
- Pri **aksonometričnih** projekcijah je projekcijska ravnina premaknjena glede na predmet, glede na število kotov, ki so na projicirani kocki enaki jih delimo na izometrične (trije), dimetrične (dva) in trimetrične (nič).
  (+) Črte so skalirane vendar faktor skaliranja poznamo,
  (+) razmerja črt se ohranijo,
  (+) vidimo vse tri osnovne stranice kockastih predmetov,
  (–) koti se ne ohranijo
  Uporabljajo se za tehnične risbe, ilustracije, CAD, igre (enako vidimo tudi oddaljene predmete – optične iluzije)
- Pri projiciranju v **koordinate pogleda**, kjer z kaže v 3D sceno (levosučni) oziroma stran (desnosučni) lahko z enostavno zanemarimo (odstranimo vrstico iz matrike). Pri aksonometričnih in večpoglednih projekcijah moramo kamero še rotirati

2.2. Perspektivne projekcije

- Pri perspektivni projekciji žarki niso vzporedni, temveč se stikajo v točki (oko-gledišče)
- Črte, ki so na predmetu vzporedne, in niso na ravnini, ki je vzporedna projekcijski ravnini (ta je nekje med glediščem in sceno), se sekajo v **ponornih točkah**
- Perspektivne poglede lahko **delimo** glede na število ponornih točk – če je ena glavna ploskev predmeta vzporedna s projekcijsko ravnino imamo eno ponorno točko – enotočkovna, če je ena glavno stranica vzporedna s projekcijsko ravnino imamo dve ponorni točki - dvotočkovna, drugače pa imamo tritočkovno perspektivo - tritočkovna
- (+) predmeti, ki so dlje so manjši od tistih bližje (realističen izgled)
  (–) enake razdalje niso projicirane v enake razdalje na projekciji,
  (–)predmeti so popačeni,
  (–)koti se ohranijo le na ravninah, ki so vzporedne s projekcijsko
- Z d označimo razdaljo od kamere do projekcijske ravnine

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



- Razdalja med projekcijsko ravnino ter glediščem in velikost izseka na projekcijski ravnini določa **zorni kot** (field of view)
- Z njim lahko npr. poudarimo efekt perspektive
- V WebGL je vzporedna projekcija z ortho, perspektiva s perspective
- <u>Kamera ima tipično naslednje parametre</u>:
  - Položaj in orientacijo (določajo KS pogleda)
  - Slikovno razmerje (razmerje med širino in višino slike)
  - Zorni kot
  - Prednja in zadnja ravnina rezanja (določata vidno prisekano piramido, kjer so vidni le predmeti znotraj te piramide)
  - Fokusna razdalja (za simuliranje globinske ostrine)

3. Poligoni in graf scene

3.1. Poligoni

- 3D predmet predstavimo z množico poligonov (mnogokotnikov) – želimo učinkovit prikaz in geometrijske operacije (ali je točka znotraj poligona), učinkovita povpraševanja (kateri so sosednji poligoni) in učinkovito porabo pomnilnika
- Najprimernejši so **trikotniki**, če so vedno **planarni** in **konveksni** (vsak notranji kot < 180 stopinj, vsaka črta med dvema robovoma v celoti leži znotraj trikotnika)
- Definiramo jim lahko tudi težiščni koordinatni sistem, kjer sta koordinati osi stranici trikotnika
- Točka je predstavljena z $\mathbf{p = \alpha * a + \beta * b + \gamma * c}$ $\mathbf{\alpha + \beta + \gamma = 1}$
  $0 < \alpha, \beta, \gamma < 1$: točka je znotraj trikotnika
  $0 \leq \alpha, \beta, \gamma \leq 1$: točka je izven trikotnika
  Sicer je točka izven trikotnika
- Če poznamo predstavitev točke p s težiščnimi koordinatami in lahko enostavno računamo **interpolacijo** lastnosti oglišč v točki: npr. barva je b_p $\mathbf{= \alpha * b_a + \beta * b_b + \gamma * b_c}$
- Splošno lahko poljubno točko in trikotnik α, β, γ izračunamo kot:

3.2. Triangulacija

- Če nimamo trikotnikov naredimo triangulacijo. Če je poligon konveksen eno oglišče povežemo z vsemi drugimi (ni najbolj optimalen) WebGL ne podpira večkotnikov.
  Triangulacija je način določanja triangulacijske točke s pomočjo trikotniških pravil in 2 točk z znanima koordinatama.
- **Garey-Johnson-Preparata-Tarjan**, 1978 (plane sweep pometanje po ravnini)
  Poligon razbijemo v y-monotone poligone da nimajo dolin ali vrhov v y smeri
  1. trapezoidacija
  - dodamo vodoravne odseke (pometanje)
  - vsak trapezoid, kjer oglišče ni povezano z manjšim (po y), razbijemo (povežemo)
  - odstranimo odseke

  2. triangulacija dobljenih y-monotonih poligonov
  - od spodaj navzgor zapiramo trikotnike

3.3. Graf scene

- Ker so predmeti v RG tipično sestavljeni iz več delov, ki so med seboj odvisni jih lahko **združujemo v skupine** (matrike hranimo na skladu, izris je rekurziven sprehod skozi drevo)
- Transformacijska matrika je produkt matrik od korena do predmeta
- Vsaka skupina ima **seznam članov v hierarhiji** in si deli lastnosti. Povezave nosijo podatke o relativnem položaju glede na starša (transformacijske matrike) – iz drevesa dobimo usmerjen acikličen graf. Predmeti imajo poleg položaja še druge lastnosti: barva, lastnost za senčenje, natančnost, animacijski parametri, …
  Hierarhija predmetov **omogoča** tudi lažje hierarhično deljenje prostora, izločanje predmetov, ki jih ne vidimo še pred izrisom, izračun nivojev podrobnosti
- En predmet lahko pripada več skupinam (ima več instanc), vsako sicer z drugačno transformacijsko matriko (npr. kolesa pri avtomobilu).

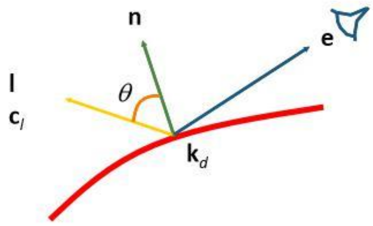
4. Osvetljevanje in senčenje

4.1. Osvetljevanje

- Pri **osvetljevanju** računamo osvetlitev neke površine, pri **senčenju** pa kako jo izrisati
- Ločimo **lokalno** - poenostavljeno, le en odboj med izvorom in gledalcem, računanje osvetlitve ene ploskve neodvisno od ostalih, ni odbojev svetlobe in
  globalno osvetljevanje - več odbojev svetlobe od predmetov, računsko zahtevno, več interakcije med površinami

4.2. Lokalno osvetljevanje

- Odbita svetloba je vsota treh komponent: **razpršeni** odboj, **zrcalni** odboj in **ambientna** svetloba
- **Razpršeni odboj** (diffuse) predstavlja material, ki odbija svetlobo enakomerno na vse strani (npr. papir, les, kamen), torej ni važno s katere strani ga gledamo. Odbita svetloba je proporcionalna s kosinusom kota med vpadno svetlobo in normalo na površino. Večji kot je kot, manj je svetlobe. Pri 90 stopinjah svetloba izginja. Temu se reče Lambertov kosinusni zakon.

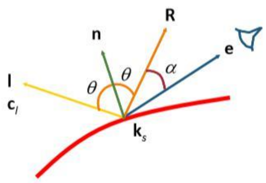


- n ... normala na površino (normirana)
- l ... smer svetlobe (normirana)
- k_d ... razpršena odbojnost (RGB vektor barve)
- cl ... intenziteta vpadne svetlobe (RGB vektor)
- cd ... intenziteta odbite svetlobe (RGB vektor)

$$\mathbf{c}_d = \mathbf{c}_l \mathbf{k}_d (\mathbf{n} \cdot \mathbf{l}) = \mathbf{c}_l \mathbf{k}_d \cos\theta$$

- **Zrcalni odboj** (specular) je zabrisan odsev vira svetlobe in je odvisen od smeri gledanja. Pri ogledalu imamo idealen odboj, kjer je kot vpada enak kotu odboja. Svetloba se pri zrcalnem odboju odbija približno v smeri idealnega odboja.
  Phongov model: kot med idealnim odbojem in smerjo pogleda določa količino odbite svetlobe; parameter zrcalnega odboja p določa velikost razpršitve
  Blinnov model je podoben Phongu, vendar ne potrebujemo izračuna odboja – imamo nek kompromisni vektor h. Če sta luč in gledalec daleč od površine lahko predpostavimo, da je h konstanten – hitrejši izračun.

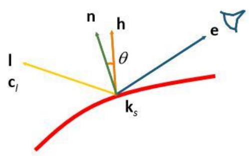
Phongov



$$\mathbf{c}_s = \mathbf{c}_l \mathbf{k}_s (\mathbf{R} \cdot \mathbf{e})^p$$

$$\mathbf{R} = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} - \mathbf{l}$$

Blinnov



$$\mathbf{c}_s = \mathbf{c}_l \mathbf{k}_s (\mathbf{h} \cdot \mathbf{n})^s$$

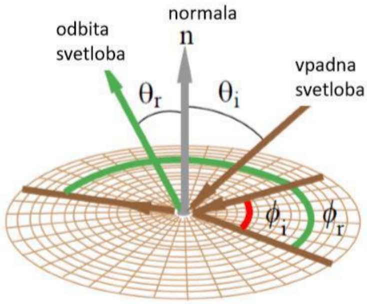
$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{e}}{|\mathbf{l} + \mathbf{e}|}$$

- **Ambientna svetloba** v realnosti je del svetlobe povsod, ker se odbija od ostalih predmetov – to aproksimiramo z ambientno svetlobo (povsod dodamo konstantno osvetlitev) **$c = c_a * k_a$**

- Ker imamo lahko več virov svetlobe, seštejemo vse prispevke in tako dobimo **model lokalnega osvetljevanja**

- Ker se gostota svetlobnega toka zmanjšuje s kvadratom razdalje lahko uvedemo tudi funkcijo zmanjševanja

- Posplošeno lahko odboj zapišemo s funkcijo **BRDF** – za vsak par smer luči l in smer gledalca e se določi, koliko svetlobe se odbije do gledalca. Je štiridimenzionalna funkcija vpadne in odbite svetlobe

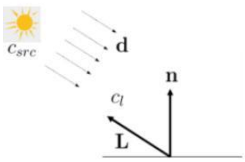


$$\mathbf{c} = \mathbf{c}_a \mathbf{k}_a + \sum_i f(d_i)\, \mathbf{c}_i \,\mathrm{BRDF}(\theta_i, \phi_i, \theta_r, \phi_r) \cos\theta_i$$

4.3. Luči

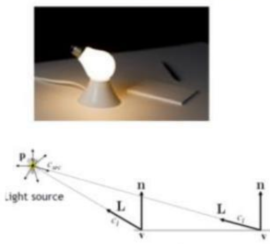
- Imajo razne parametre: barva, površina, usmerjenost – tudi odbojne površine so vir svetlobe
- **Usmerjena** luč ima izvor zelo daleč (npr. Sonce), kjer so si žarki vzporedni – določa jo barva in smer
- **Točkasta** luč (žarnica) seva svetlobo v vse strani enako – kot vpadne svetlobe je odvisen od položaja, jakost pada s kvadratom razdalje
- **Reflektor** seva v neko smer in ima poleg položaja torej tudi usmeritev – kot vpadne svetlobe je odvisen od položaja in smeri, jakost pa je odvisna od širine stožca in pada proti robu stožca s potenco f

Usmerjena luč



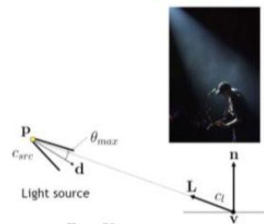
$$\mathbf{L} = -\mathbf{d}$$
$$c_l = c_{src}$$

Točkasta luč



$$\mathbf{L} = \frac{\mathbf{p}-\mathbf{v}}{\|\mathbf{p}-\mathbf{v}\|}$$
$$c_l = \frac{c_{src}}{\|\mathbf{p}-\mathbf{v}\|^2}$$

Reflektor



$$\mathbf{L} = \frac{\mathbf{p}-\mathbf{v}}{\|\mathbf{p}-\mathbf{v}\|}$$
$$c_l = \begin{cases} 0 & \text{if} \quad -\mathbf{L}\cdot\mathbf{d} \le \cos(\theta_{max}) \\ c_{src}(-\mathbf{L}\cdot\mathbf{d})^f & \text{otherwise} \end{cases}$$

4.4 Senčenje

- Za barvanje poligonov lahko računamo osvetlitev *celega poligona*, osvetlitev *na ogliščih* ali pa v *vsaki posamezni točki*
- Pri **ploskem** (konstantnem) senčenju osvetlitev računamo za cel poligon in ga pobarvamo z eno barvo (npr. izračunamo osvetlitev v enem oglišču). Najbolj primitiven algoritem, je hiter in da nezvezen izgled
- Pri **Gouraudovem** senčenju računamo osvetlitev v vsakem oglišču in nato v notranjosti interpoliramo med barvami oglišča – za izračun osvetlitve potrebujemo normale v ogliščih, ki jih lahko računamo s povprečenjem normal ploskev, ki se stikajo v oglišču za čim bolj mehke prehode.
  Problematično je pri odsevih, ko je število poligonov majhno (OpenGL notranjost se interpolira)
  Interpoliramo z <u>bilinearno interpolacijo</u>, kjer za neko točko v notranjosti izračunamo osvetlitev na robovih poligona v dveh točkah na primer Q in R z interpolacijo po y, nato pa izračunamo osvetlitev v točki P z interpolacijo po x
- Pri **Phongovem** senčenju osvetlitev računamo v vsaki točki poligona – za izračun osvetlitve v točki potrebujemo normalo v točki, ki jo tudi dobimo z bilinearno interpolacijo. Najprej izračunamo normali v robnih točkah Q in R z interpolacijo po y, potem pa normalo v P izračunamo z interpolacijo po x. Je boljše kvalitete kot Gouraud vendar počasnejše.
- V OpenGL implementiramo z senčilnikom

$$\mathbf{c} = \mathbf{c}_a \mathbf{k}_a + \mathbf{k}_e + \sum_i f(d_i) \mathbf{c}_i \left(\mathbf{k}_d (\mathbf{l}_i \cdot \mathbf{n}) + \mathbf{k}_s (\mathbf{h}_i \cdot \mathbf{n})^s \right)$$

5. Teksture

5.1. Lepljenje tekstur

- Teksture so lahko 2D ali 3D in lahko vplivajo na barvo, normale, položaj...
- **Pikslom teksture (texels)** se določijo koordinate **u** in **v** na območju [0,1] – vsako oglišče trikotnika pa hrani u,v koordinate dela teksture, ki se nanj preslika (za vsako točko v trikotniku lahko računamo z interpolacijo u,v koordinat). Teksturni prostor sega od (0,0) – levo spodaj do (1,1) – desno zgoraj
- Pri **vzporednem** (planarnem) lepljenju uporabimo linearno transformacijo xyz koordinat predmeta
- Pri **perspektivnem** lepljenju uporabimo perspektivno projekcijo xyz koordinat predmeta
- Pri **sferičnem** lepljenju uporabimo sfreične koordinate – kot bi predmet postavili v kroglo in teksturo s krogle nalepili na predmet
- **Cilindrično** lepljenje je podobno kot sferično, le da mapiramo preko valja
- Najbolj kompleksno je **kožno** lepljenje, kjer površino predmeta razvijemo v 2D kožo, jo pobarvamo in nalepimo nazaj
- V primeru, da u,v koordinate padejo izven območja [0,0] x [1,1] lahko teksturo **ponavljamo**, **zrcalimo** ali pa uporabimo **barvo roba** teksture ali teksture ne upoštevamo

5.2. Izris tekstur

- Za vsako točko v trikotniku lahko z interpolacijo u, v koordinat izračunamo v katero točko teksture se preslika
- Interpolacija je lahko standardna **bilinearna**, lahko uporabimo tudi težiščne koordinate točk α, β, γ
- Problem nastane, ker se teksture izrišejo po preslikavi v koordinate zaslona in zato ne da pravilnih rezultatov (**perspektivno popačenje**) – rešitev je hiperbolična interpolacija
- Ker dobljene interpolirane u,v koordinate navadno ne padejo na teksel, se barva lahko računa glede na najbližjega soseda (hitro a slaba kvaliteta) ali pa z bilinearno interpolacijo med 4 sosednjimi teksli (teksturni piksli)
- Pri bolj oddaljenih nalepljenih teksturah pride do problema **prekrivanja** (aliasing), ker ne vzorčimo dovolj pogosto – vsak piksel na sliki pokrije velik kos teksture; Izognemo se s **povprečenjem tekslov** (sicer počasno opravilo)
- Boljša rešitev je povprečenje vnaprej (**mipmapping**), kjer vnaprej izračunamo več verzij teksture različnih velikosti (vsaka 2x manjša) – najprej izračunamo koordinate u,v in približno velikost piksla v teksturnem prostoru, nato z bilinearno interpolacijo izberemo barvo iz ustrezno velike teksture Če piksel pokriva 10x10 tekslov izberemo teksturo 8x8 – 3. nivo 1 teksel na nivoju 3 je povprečje 4^3=64 tekslov na nivoju 0
- Za še mehkejši prehod med nivoji se uporablja dva najbližja nivoja – **trilinearna interpolacija** (naredimo bilinearno interpolacijo, da dobimo barvo piksla v obeh nivojih nato linearno interpoliramo še med nivojema)
- **Anizotropično filtriranje** ne obravnava piksla kot kvadrat in ne povprečij teksture v vseh smereh enako – povprečij jo znotraj okna izračunanega za vsak piksel (bolj zahtevno, bolj realno)
- WebGL – najbolje da je velikost potenca 2, minification (teksel < piksel) in obratno - magnification

5.3. 3D teksture

- 3D tekstura je definirana v **treh dimenzijah** – navadno je specificirana proceduralno
- Osnova je **funkcija f(s,t,r)** funkcija lahko vrne barvo, prosojnost, normale.
- Uporabljajo se za naravne materiale, kjer imamo osnovno funkcijo in šum za dodan realizem
- Poznamo **Perlinov šum**, ki je gradientni šum z interpolacijo med naključnimi gradienti – enostaven nadzor nad frekvenco in fazo. **Turbulenca** je vsota skaliranih Perlinovih šumov – regularnost
- V naravnih teksturah velikokrat srečamo neko regularnost, ki se pojavlja na več velikostnih nivojih; lahko jo dobimo s seštevanjem skaliranih šumov

6. Grafični cevovod

- Pretvori predmet oz. sceno iz računalniškega zapisa v bitno sliko
- Implementiran v strojni opremi (grafična kartica)
- Za predstavitev za grafični cevovod uporabljamo poligone – največkrat trikotnike (vedno planarni in konveksni)
- Poligonski model je zapisan s 3D koordinatami oglišč, kako se oglišča povezujejo v poligone, barve in teksture ter normale
- Za **vhod** v cevovod se potrebuje geometrijski model, transformacije, podatki za osvetljevanje, položaj in parametri kamere in ločljivost slike

  Izhod je **bitna slika** (npr. 24 bitna RGB vrednost za vsak piksel)
- Vsaka faza **posreduje rezultate** naslednji fazi
- Faze so lahko implementirane **strojno** ali **programsko** – vstavljanje dodatnih funkcionalnosti v cevovod se imenuje **senčilniki** (shaders), ki se izvajajo na GPU in zamenjajo funkcionalnost delov cevovoda
- **Vertex** shaders se izvajajo na posameznih ogliščih (transformacije oglišč in normal, osvetljevanje...)
- **Geometry** shaders lahko ustvarjajo novo geometrijo
- **Tesselation** control se uporabljajo za kontrolo nivojev podrobnosti in lepljenje odmika
- **Fragment** shaders izvajajo operacije na pikslih fragmentov (računanje barve, osvetljevanje...)

Specifikacija oglišč

- Priprava podatkov za upodabljanje (Vertex Array Objects, Vertex Buffer Objects)
- Določimo oglišča in definiramo primitive, ter povezave med njimi

Senčilnik oglišč

- Obdava posameznih oglišč, program specificira uporabnik (je obvezen)
- Transformacije oglišč/normal, osvetlitve, projekcija
- Oglišča le spreminjamo, ne dodajamo novih

Teselacija

- Deljenje poligonov na več manjših
- Dva senčilnika, ki jih specificira uporabnik
- control: stopnja deljenja in evaluation: interpolira lastnosti oglišč

Senčilnik geometrije

- Dodaja nove primitive/ briše obstoječe
- Spreminja obstoječe in pretvarja primitive

Postprocesiranje oglišč

- Oglišča lahko shranimo nazaj v VBO
- Rezanje in perspektivno deljenje – normalizirane koordinate naprave
- Transformacija v koordinate zaslona

Sestava primitivov

- Zaporedje oglišč gre v zaporedje primitivov
- Izvede se izločanje zadnjih ploskev

Rasterizacija

- Primitivi se pretvorijo v diskretne elemente, glede na del končne slike, ki ga pokrivajo
- Najmanj en fragment na piksel
- Interpolacija lastnosti oglišč v pikslih in senčilnik fragmentov nad posameznim pikslom/delom piksla

Senčilnik fragmentov

- Program za vsak fragment (vsaj 1 fragment na piksel)
- Izračuna barvo, globino, šablono

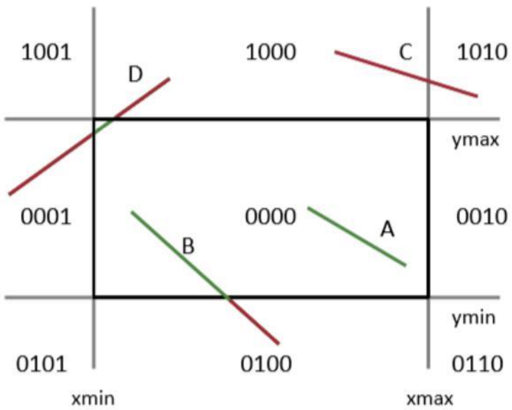
Obdelava vzorcev

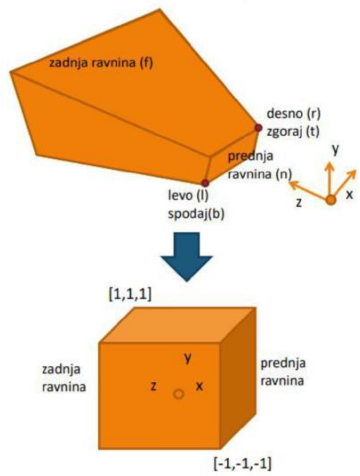
- Sestavi končno sliko
- Test škarij, mehčanje robov, test šablone, test globine, zlivanje, zapisovanje vrednosti

6.1. Rezanje

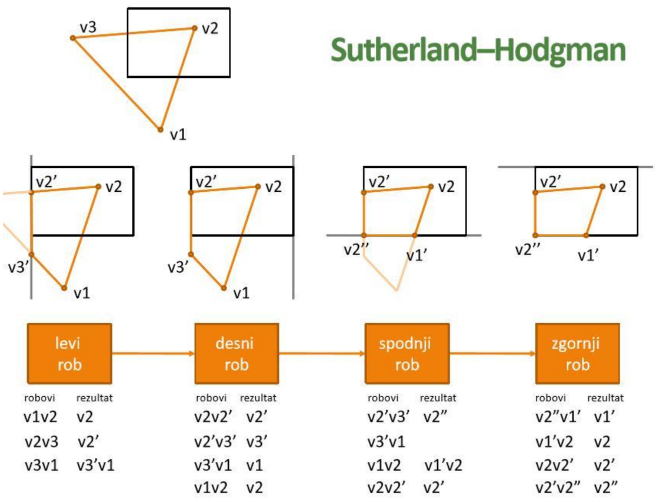
- **Odstranimo** vse ploskve, ki niso v vidnem stožcu in **porežemo** ploskve, ki so delno v, delno pa izven tega stožca
- Režemo v fazi post procesiranja ali med rasterizacijo. Režemo vse kar je izven homogenega dela (točke odstranimo, črte in trikotnike odstranimo če so celotni izven drugače režemo)
- Rezanje je enostavneje, če vidno polje pretvorimo v **normalizirano kocko**, s koordinatami med [1,1,1] in [-1,-1,-1] – normalizirane koordinate naprave
  Koordinate rezanja nastanejo pri perspektivni projekciji pred perspektivnim deljenjem, po množenju še ne opravimo perspektivnega deljenja. Vidno je vse kar: $-w_c <= x_c, y_c, z_c, <= w_c$
  Režemo torej vse, kar je izven homogenega dela w_c
- **Cohen-Sutherland algoritem za rezanje črt:**
  2D ravnino razdelimo na 9 regij in v vsaki označimo s 4 biti ali smo znotraj ali izven x in y

  Gledamo te 4 bite za začetno in končno točko črte, če:
  - **o_1&o_2 = 0** je točka znotraj in ne režemo
  - **o_1&o_2 nista 0** jo v celoti režemo
  - **o_1 = 0** in **o_2 <> 0** jo režemo, bit pove s katerim robom režemo
  - **o_1&o_2 nista 0** in **na različnih straneh**, režemo do prvega roba in zračunamo na novo ponovno za obe polovici
- (+) Deluje tudi na 3D - dodamo 2 bita,
  (+) Je hiter
  (--) Rekurzivna obravnava črt v zadnjem primeru





- **Sutherland-Hodgman algoritem za rezanje poligonov**:
  Rezanje po vrsti po robovih poligona.
  Za vsak rob pogledamo, ali ga je potrebno rezati z vsako od štirih stranic (px,py):
  Če je **px zunaj, py znotraj**, je rezultat presečišče in py
  Če sta **obe oglišči znotraj** je rezultat py
  Če je **px znotraj, py zunaj** je rezultat presečišče
  Če sta **obe zunaj** je rezultat prazen



6.1.1 Perspektivno deljenje

Iz koordinat rezanja s perspektivnim deljenjem (x, y, z delimo z w) pretvorimo v normalizirane koordinate naprave. Prisekana piramida pogleda postane kocka med [1,1,1] in [-1,-1,-1]. Z koordinata ni vel linearna 1/z.

6.1.2 Koordinate zaslona

Iz NDC pretvorimo v koordinate zaslona – viewport coordinates. Ustrezajo ločljivosti zaslona. Pretvorba je le ustrezno skaliranje iz NDC. X in y se pretvorita glede na ločljivost slike, z pa se preslika na (pod)interval [0,1].

6.2. Izločanje

- Izločiti želimo čim več predmetov, ki **ne bodo vidni** – še preden pošljemo v cevovod in tudi znotraj
- Poznamo izločanje predmetov izven vidnega polja, izločanje zadnjih ploskev in izločanje zakritih ploskev
- Izločanje **zadnjih ploskev** se izvede na nivoju poligonov v cevovodu, kjer predpostavimo, da so poligoni **enostranski** – če proti **kameri gleda zadnja stran poligona, ga izločimo** (gleda se vrstni red pri izrisu trikotnika ali pa predznak komponente normale po transformaciji pogleda)
- Izločanje predmetov **izven vidnega polja** se izvede na nivoju predmetov, preden jih pošljemo v cevovod – izločimo predmete izven prisekane piramide. Ker imajo predmeti kompleksno geometrijo uporabimo za hitro izločanje tehnike razdelitve prostora
- Izločanje **zakritih ploskev** je koristno v primerih, ko je velik del scene zakrit
- V Unity določimo kaj so ploskve, ki zakrivajo lahko le statični predmeti. Na točkah znotraj scene se preračuna globino tega kar kamera vidi nato določimo ločljivost in najmanjša velikost predmeta ki zakriva. Podatki se zapečejo - izračunajo v posebne teksture pred izrisom in če se scena spreminja, jih je potrebno osvežiti.
- Imamo veliko algoritmov, npr. Hierarhični **Z-Buffer**: V fazi oblikovanja določimo predmete, ki zakrivajo (veliki predmeti). Te predmete izrišemo. Izračunamo hierarhijo globinskih slik. Poiščemo preseke očrtanih predmetov z globinskimi slikami glede na očrtan kvader

6.3. Rasterizacija

- Gledamo katere piksle pokriva poligon in kakšne barve so ti piksli
- Rasterska slika je 2D diskretno polje pikslov, koordinate pikslov so cela števila
- Koordinate predmetov so zvezne
- Rasterizacija: učinkovit izračun pokritja pikslov

- Navadno delamo s trikotniki – če imamo mnogokotnike, jih pretvorimo v trikotnike pred rasterizacijo
- **Stara šola - malo velikih trikotnikov**, izračunamo robove, zapolnimo notranjost
- **Bresenhamov** algoritem za risanje črt (primer za 0 < m < 1, y = mx + b)

  Vsakič izberemo piksel vertikalno najbližji črti. Naslednja točka je v E(x+1, y) ali NE(x+1, y+1). E izberemo, če je segment pod ali na sredinski točki med piksloma, NE pa izberemo, če je segment nad sredinsko točko
- Nekateri algoritmi uvajajo tudi mehčanje robov (antialiasing) z dodajanjem pikslov pod in nad črto
- **Scan-line rasterizacija** procesira poligon po vrsticah – v vsaki vrstici najdemo presek z vsemi robovi poligonov in jih uredimo po naraščajočem x, nato zapolnimo piksle med robovi

  (–) Izračun seznama robov je lahko počasen
  (–) težko je tudi rezanje, če imamo poligone delno izven zaslona
  (–) težka paralelizacija
- **Nova šola - več manjših trikotnikov**

  Groba sila: za vsak piksel izračunamo, ali je znotraj trikotnika. Računamo za piksle znotraj očrtanega pravokotnika, rezanje je enostavno.
  Da ne računamo preveč nepotrebnih pikslov očrtan pravokotnik razbijemo na pod področja. Če je celo pod področje izven trikotnika, ga izpustimo

  Računa se tudi mehčanje robov, kar pomeni, da računamo na višji ločljivosti in nato povprečimo rezultate.

6.4. Obdelava vzorcev

Izvede se za vsak fragment in izračuna barvo, globino in šablono

- **Test škarij** - odstrani fragmente, če so izven pravokotnika
- **Mehčanje robov** - piksle delimo na fragmente/vzorce, končno barvo piksla povprečimo, na robovih poligonov (MSAA), senčilnik fragmentov se izvede za vsak vzorec (SSAA). Celotna scena se izriše na višji ločljivosti, slika zmanjša (FSAA).
- **Test šablone -** vsak fragment ima lahko vrednost šablone. Vrednost se primerja s trenutno vrednostjo v medpomnilniku šablone – glede na operacijo lahko fragment obdržimo ali zavržemo Uporaba za omejitev izrisa na področje, sence, mehke prehode, poudarjanje robov itd.
- **Test globine -** določa vidnost (kateri poligoni bodo vidni). Vsak fragment ima globino, fragment zavržemo, če je globina večja od globine v *medpomnilniku globine*

6.5. Določanje vidnosti

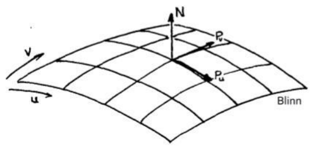
- Določiti je potrebno, kateri poligoni so pred drugimi
- **Slikarjev algoritem:** Izrišemo vse predmete od zadnjega proti prvemu. Problem je z urejanjem poligonov (npr. BSP drevo) in z urejanjem ciklov
- **Z-Buffer:** Za vsak piksel hranimo z vrednost (globino) trenutno najbližjega izrisanega poligona v medpomnilniku globine. Če je z vrednost večja od vrednosti v medpomnilniku globine jo zavržemo. Za računanje globine v pikslu uporabimo interpolacijo glede na globino oglišč trikotnika. Problem nastane, če je prednja ravnina preblizu očesu, saj bomo porabili preveč bitov za predstavitev predmetov blizu le-te. Z fighting – če imata dva predmeta isto globino, utripanje
- Problem je tudi, da hrani le najbližji piksel – ni možna transparenca, mehčanje robov…
  - odgovor so A buffer, K buffer in podobni
- Pri Z bufferju lahko fragment zavržemo, čim vemo njegovo globino. (pohitritev – preskočimo lahko senčilnik fragmentov,… Smiselno če izvedba senčilnika ne vpliva na zakrivanje
- **Zlivanje** - zlivanje barve fragmenta z barvo slike - privzeto se prepiše. Največ za transparenco.
- **Zapisovanje vrednosti** - če je fragment izrisan, se vrednosti zapišejo v končne medpomnilnike - sliko (barva, alfa), globinski medpomnilnik, medpomnilnik šablone

7. Senčilniki

7.2. Lepljenje normal

- Pri lepljenju normal (normal mapping) komponente v teksturi RGB predstavljajo **XYZ normale**
- Tekstura ne spreminja barve, temveč vpliva na **izračun osvetlitve**
- Vrednosti teksture definirajo normale na površino (s tem modeliramo hrapavo ploskev)
- Ne spreminjamo geometrije - rezultati so **vidni zaradi modela osvetljevanja**
- Dobro deluje, ko je predmet oddaljen; ko je blizu, opazimo, da predmet ni 3D, problemi pri sencah
- Izboljšamo izgled predmeta z malo poligoni
- Potek: tekstura RGB vsebuje normalo v tangentnem prostoru, skaliranje normal iz tekstur [0,255] na [-1,1]. Sledi pretvorba prostora torej vektorja luči, pogleda v tangentni prostor **ali** normale v prostor pogleda. Na koncu pa še izračun osvetlitve z novo normalo

  - *Tangent space*
  - Definiran z
    - N: normala
    - P_u: tangenta (T)
      - vzporedna s smerjo teksture u
    - P_v: bitangenta (B) (tudi binormala)
      - vzporedna s smerjo teksture v
    - $N = P_u \times P_v$



7.3. Lepljenje izboklin

- Pri lepljenju izboklin (bump mapping) tekstura ne spreminja barve, temveč spreminja normale – vpliva na **izračun osvetlitve**
- Tekstura je navadno sivinska slika – **višinska slika**
- Geometrija se ne spreminja
- Za izračun izboklin vzamemo vrednosti v višinski teksturi b(u,v)
- Točko P na predmetu bi navidezno spremenili v smeri normale
- $$P' (u,v) = P(u,v) + b(u,v)\, n$$
- Razlika v višini pove, koliko moramo spremeniti normalo, novo normalo pa nato zračunamo tako:
- $$N' = N + bv\, (Pu * n) + bu\, (n * Pv)$$

7.4. Lepljenje odmika

- Pri lepljenju odmika (displacement mapping) je tako kot pri lepljenju izboklin tekstura sivinska slika, le da tu dejansko **spreminja geometrijo** – **položaj poligonov**;
- **Teselacija** je GPU podpora za povečanje ločljivosti poligonov - deljenje črt in poligonov na manjše dele. Določi kolikokrat delimo poligon/črto – zmanjšamo kompleksnost pčoligonov
- Uporaba: nivoji podrobnosti (manjša podrobnost bolj oddaljenih modelov), lepljenje odmika, risanje parametričnih krivulj
  Tesselation Control Shader – inner: število gnezdnih primitivov
  outer: kolikokrat se deli vsak rob
  Tesselation Evaluation Shader – dobi položaj novega oglišča v težiščnih koordinatah in lahko interpolira med vrednostmi v ogliščih – položaj, normala ter jih premika – lepljenje odmika
- Senčilnik geometrije lahko ustvari novo geometrijo

- Primer: Billboarding - poligon s teksturo vedno obrnjen proti kameri – kamor se obrne kamera se tudi poligon. V programu generiramo točke, v senčilniku geometrije pa točke spremenimo v štirikotnik s teksturo
- Implementacija:
  - senč. oglišč: točko pretvorimo v prostor kamere,
  - geom.: ustvarimo štirikotnik okoli točke in UV koordinate teksture,
  - frag.: izrišemo štirikotnik

7.5. Lepljenje okolice

- Z lepljenjem okolice (environment/reflection mapping) lahko simuliramo **odboj svetlobe** – smer odbite svetlobe določa teksle na teksturi
- Cubemap: tekstura predstavlja 6 smeri pogleda in sicer okolico nalepljeno na krogli ali kocki (tudi Skybox)
- Barvo predmeta računamo kot odboj žarka iz kamere na skybox teksturo
- Upodabljanje: izračunamo odboj vektorja pogleda preko normale pretvorimo v koordinate sveta, preberemo teksturo

7.6. Lepljenje senc

- Sence imamo lahko že vnaprej izračunane za statične predmete in zapečene v teksture – *lightmaps* ali pa jih računamo med izrisovanjem – *shadow mapping*
- Z lepljenjem senc lahko dodamo občutek **globine** – če je točka **vidna** iz luči je **osvetljena**, kar lahko izračunamo s **postavitvijo kamere na položaj luči**
- Najprej torej izrišemo sceno s postavitvijo kamere na položaj luči in shranimo globinsko sliko, torej razdaljo do luči – **zemljevid senc** (shadow map). V drugem koraku pa izrišemo sceno iz položaja kamere in na vsakem pikslu razdaljo do luči primerjamo z razdaljo shranjeno v zemljevidu senc – če je daljša smo v senci, drugače pa je piksel osvetljen
- Implementacija 2. koraka:

  Senčilnik oglišč: vsako oglišče pretvorimo tudi v prostor luči s projekcijsko matriko luči

  Senčilnik fragmentov: primerjamo oglišče z zemljevidom senc in upoštevamo korekcijski faktor

7.7. Ambientno zastiranje

- Deli predmetov so **zastrti zaradi bližine drugih delov** - posledično so temnejši
- V zunanjem okolju lahko ocenimo na primer glede na to kako vidno je nebo za vsako točko. Znotraj ne primerjamo vseh točk prostora. Izvor ambientne svetlobe so večinoma stene. Rezulti niso ostre sence, temveč zatemnjeni deli/mehke sence.
- Ideja **Screen space AO**: izračun faktorja zastiranja v vsakem fragmentu. V vsakem fragmentu vzorčimo okolico npr. v polkrogli usmerjeni kot normala. Odstotek točk, ki je pod površino je faktor zastiranja, ugotovimo iz globinske slike. Z njim potemnimo sliko.
- **Odloženo upodabljanje** : V prvem koraku v medpomnilnike shranimo vsaj barvo, normale in globino. V drugem koraku za vsak viden piksel vemo globino in normalo, izračunamo osvetlitev, končno barvo in lahko celo sceno izrišemo na pravokotnik.
  Prednost je hitrost, osvetljevanje je ločeno od geometrije. Osvetlitev računamo na koncu, za vsako luč le za piksle, ki jih osvetli. Težave so, da različni materiali zahtevajo več medpomnilnikov. Robove težje mehčamo, ni transparence.
- **Upodabljanje v igri**: Cubemap okolja; Pretvorba v dve polkrogli; Izločanje – level of detail (odvisna od razdalje je tudi ločljivost predmeta); Odloženo upodabljanje G buffers; Sence (4 zemljevidi senc: natančni blizu, manj daleč); Odsev v vodi – scena izrisana obrnjeno v teksturo, ki bo nalepljena na vodo kot odsev; Ambientno zastiranje; Sestava slike; Boljša koža; Voda – odsev, lom svetlobe, lepljenje odmika; Atmosfera; Prosojni predmeti; HDR v LDR, Mehčanje robov in popačenje leče; UI (minimap

8. Povpraševanja

8.1. Poligonske predstavitve

- **Povpraševanja** so sestavni del operacij pri izrisovanju, interakciji in spreminjanju 3D modelov: kateri poligoni se stikajo z danim poligonom, katera dva poligona se stikata v robu, kateri poligoni se stikajo v oglišču...?
- Potrebujemo ustrezno predstavitev poligonov. Pri predstavitvah poligonov ločimo dve vrsti informacij – **topologijo** (kako so poligoni povezani) in **geometrijo** (kje so poligoni v 3D prostoru)
- Pri **indeksiranih** poligonih so le-ti definirani z indeksi oglišč, torej imamo eno tabelo z oglišči in eno tabelo z poligoni
  (+) topologija in geometrija sta ločeni
  (–) sosedje niso dobro definirani
- Za boljšo predstavitev topološke informacije uvedemo **sosednostne sezname**, ki so razširitev indeksiranih poligonov z dodatnimi topološkimi informacijami. Za določanje sosednosti se uporablja trikotnike, kjer vsak trikotnik kaže na tri sosednje trikotnike, vsako oglišče pa kaže na en sosednji trikotnik.
  (+) Omogoča **enostavno iskanje** sosednjih trikotnikov in trikotnikov okoli oglišča; Imamo dve dodatni tabeli – ena vsebuje podatke, na kateri trikotnik kaže določeno oglišče, druga pa vsebuje sosede vsakega trikotnika
- Alternativa je **Winged-edge** predstavitev, ki za določanje sosednosti uporablja robove (torej dela na poljubnih poligonih). Vsak usmerjen rob kaže na levi in desni sprednji rob, levi in desni zadnji rob, prednje in zadnje oglišče ter levi in desni poligon. Vsak poligon in oglišče kažeta na en rob.
  (+) Enostavno iskanje robov poligona in oglišča ter sosednjih oglišč in poligonov
  (–) Porabi več prostora kot sosednostni seznami
- **Half- edge** predstavitev. Podobno kot winged-edge, le da rob razdelimo na dva. En rob kaže naprej, en nazaj. Dodamo še kazalec na drug pol-rob, lahko izpustimo kazalec nazaj.

8.2. Tehnike razdelitve prostora

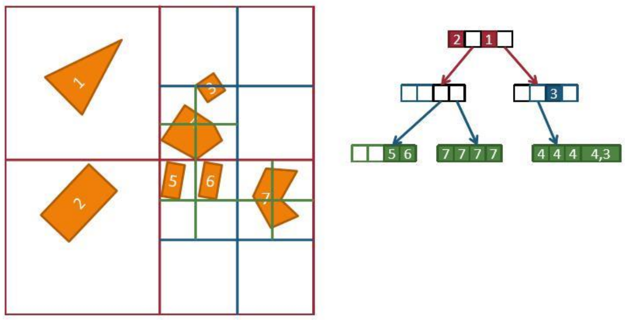
- Razdelijo **3D prostor** na več delov
- Uporabljamo za bolj **zgoščeno predstavitev** in **hitrejše povpraševanje** po predmetih (izločanje, detekcija trkov, iskanju sosedov, sledenje žarku)

  Poznamo: **prostorsko usmerjene**: delijo prostor – predstavljen tudi prazen prostor, točka v prostoru je v eni veji drevesa, predmet je lahko v več razdelkih
  predmetno usmerjene: delijo prostor glede na očrtan obseg predmetov – ni praznih prostorov, točka lahko spada v več vej dreves, predmet spada v eno vejo
- Prostor razdelimo z **mrežo kvadrov – vokslov**
- Posamezen kvader je prazen ali vsebuje seznam predmetov
- (–) Problem v ločljivosti mreže (mreža lahko pregroba ali prefina)

8.2.1. Osmiško drevo

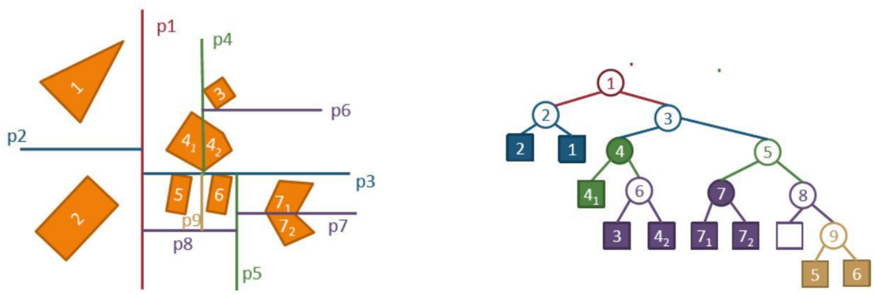
- **Hierarhična** drevesna predstavitev 3D prostora (v 2D štiriško drevo)
- Rekurzivno **deli prostor na 8 delov** do neke globine – kriterij odvisen od aplikacije (npr. max število poligonov v celici ali homogenost regije)
- Gradnja je **enostavna**: Če prostor (vozlišče) vsebuje več kot N poligonov ga razdelimo na 8 podvozlišč. Poiščemo in označimo kateri poligoni pripadajo posameznim podvozliščem (enostavno, ker so meje podprostorov poravnane z osmi). Če poligon sodi v več podprostorov, ga vključimo v vse. Nadaljujemo rekurzivno z vsemi narejenimi podvozlišči

- Primer štiriškega drevesa z pogojem max 4 poligoni v enem delu:



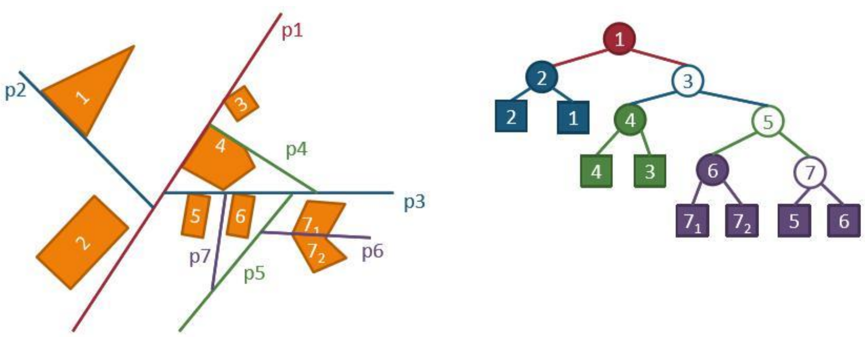
8.2.3. KD drevo

- So posplošitev osmiških dreves - binarna drevesa
- Vsakič (pod)**prostor razdelimo na dva dela** glede na ravnino (najprej x, nato y in z)
- V prostor postavljamo ravnine **vzporedne z osmi** in delimo ciklično po vrsti po oseh
- Razdelitev je tako bolj fleksibilna in optimalna, vendar je **gradnja zahtevnejša**



8.2.2. BSP drevo

- BSP (Binary Space Partitions) drevesa predstavljajo **hierarhično** razdelitev prostora
- **Z ravninami prostor rekurzivno delimo** – vsakič na dva dela.
- Pogoj za ustavitev je odvisen od namena (npr. detekcija trkov: dokler vsebina v listih ni dovolj enostavna; vidnost: dokler vsebina v listih ni konveksno področje z enim predmetom; predstavitev polnih teles: dokler ni vsebina v listih polno konveksno področje)
- Vsako vozlišče vsebuje podatke o ravnini in seznam predmetov, ki v celoti ležijo na ravnini – vsak list je prazen ali ima seznam predmetov, ki jih vsebuje
- Če ravnina seka predmet, ga razdelimo na dva predmeta – predmet je vedno v eni veji
- Gradnja je **zahtevna:** Izbira ravnine v vsakem koraku je odvisna od uporabe (npr. na vsaki strani čim bolj enako št. poligonov). Cilj je čim bolj uravnoteženo drevo. Po izbiri ravnine je potrebno ugotoviti, kateri predmeti so levo in desno od nje in katere je potrebno deliti na pol.
- Primer za sceno s poligoni z pogojem max 4 poligoni (polno pobarvana vozlišča vsebujejo tudi seznam predmetov):

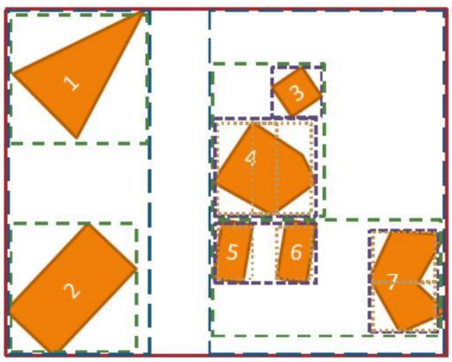


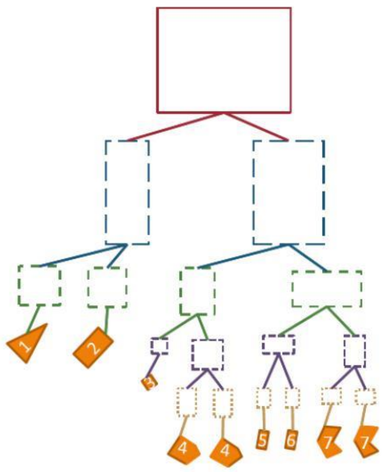
8.2.4. Očrtana telesa in BHV

- So **poenostavitve kompleksnih teles** in pomagajo pri računanju trkov, sledenju žarkov...
- Naj bi se čim bolje **prilegalo predmetu**, da po nepotrebnem ne računamo presekov
- Poznamo **krogle** (slabo prileganje a hitre operacije), **valje** (primeren za osebke v igrah, hitre operacije), z **osmi poravnane kvadre** (boljše prileganje od krogle, še vedno hitre operacije) in **usmerjene kvadre** (dobro prileganje a počasnejše operacije)
- Očrtana telesa postavimo v **drevo** – na vrhu očrtana celotna scena. Vsako vozlišče je očrtano telo celotnega poddrevesa, kjer listi vsebujejo **geometrijo**
- Uporabno za računanje presekov začnemo pri korenu drevesa (celotna scena) in nadaljujemo proti predmetom v listih.

8.2.5. AABB drevo

- Binarno drevo z **osmi poravnanih očrtanih kvadrov**
- Gradimo **rekurzivno od zgoraj navzdol** ali od spodaj navzgor – pogoji so lahko različni, očrtani kvadri naj bi se čim manj prekrivali.
- Na vsakem koraku izračunamo najmanjši očrtan kvader vseh predmetov. Vzdolž najdaljše stranice izberemo ravnino, ki kvader razdeli in predmete razdelimo v obe polovici
- Primer za sceno s poligoni, pogoj je max 4 poligoni





8.2.6. R drevo

- Podobno AABB drevesu, AABB kot očrtano telo
  Ni binarno - analogno B drevesom, le za prostorske podatke
- Optimizirano za shranjevanje na disku

8.2.6. OBB drevo

- Binarno drevo **usmerjenih očrtanih kvadrov**
- Podobno kot AABB le da se bolje prilega, gradnja in računanje presekov pa je bolj zapletena
- Več možnosti kako razdeliti geometrijo
- Navadno izračunamo lastne vektorje kovariančne matrike znotraj območja. Dobimol osi z maksimalno in minilano varianco položajev točk, na podlagi katerih izračunamo usmerjenost kvadra.

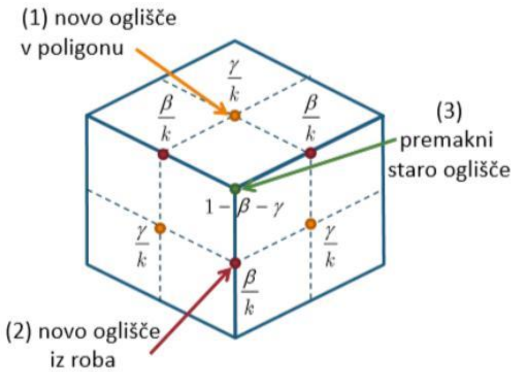
9. Detekcija trkov

- Vprašanje **kdaj** predmeti med seboj trčijo in **kako** se obnašajo **po trku**. Težak problem ker moramo preverjati trk vsakega predmeta z vsakim in to v vsakem koraku. Poznamo dva pristopa:
- **Preverjanje prekrivanja**: preveri ali je že prišlo do trka, preveri v eni točki, uporablja se v igrah in se računa **na koncu** koraka simulacije – če je do trka prišlo moramo najprej izvedeti **čas trka**, nato premaknemo čas simulacije nazaj v to točko in izračunamo rezultat trka in izvedemo simulacijo do konca.
  Problem je, če se predmeti **premikajo prehitro** – hitrost najhitrejšega predmeta krat časovni korak morata biti manjša kot najtanjši predmet (omejitev hitrosti, velikosti objektov, čas. koraka)
- **Preverjanje križanj**: preveri ali se bosta poti predmetov križali v prihodnosti in bo prišlo do trka, računamo na začetku koraka simulacije, predmet ekstrapoliramo po poti. Če bo do trka prišlo premaknemo simulacijo na čas trka in izračunamo rezultat ter izvedemo simulacijo do konca. Poznati moramo natančno stanje sistema v času t, ter hitrost mora biti konstantna. Lahko uporabljamo tudi za preverjanje v kateri predmet se bomo zaleteli najprej, če iščemo v neki smeri
- Za **pohitritev** preverjanja trkov **poenostavimo** kompleksne modele in **zmanjšamo** število primerjav – oboje dosežemo s tehnikami razdelitve prostora, detekcijo razdelimo na več korakov
- V interaktivni grafiki detekcijo trkov delimo na dva podproblema **groba** detekcija in **fina** detekcija
- **Groba detekcija:** Je hiter test**.** Uporabimo očrtana telesa za predstavitev predmetov in delitev prostora največkrat za statično geometrijo.
  Sweep and prune algoritem da najdemo pare ki se prekrivajo: Kvader predstavimo s 3 intrevali. Najprej imamo n kvadrov in tri 2n dolge urejene sezname začetkov in koncev intervalov, za vsako os enega. Nato v vsakem seznamu poiščemo vse hkrati aktivne intervale in jih shranimo v n*n matriki. In vsi kvadri, ki imajo vse tri intervale hkrati aktivne, se prekrivajo.
  Za predmete ki se **ne premikajo**, lahko vnaprej izračunamo razdelitev prostora z enim od algoritmov in nato enostavno preverjamo ali so predmeti prišli v notranjost ovir. Prav tako lahko tehniko uporabimo za **dinamične predmete** - omejevanje s katerimi računamo trke.
  Primer: navadna mreža za omejevanje sosedov, s katerimi računamo trke - če je celica večja od predmeta, vemo, da moramo gledati kvečjemu še sosednje celice
- **Fina detekcija:** Ko najdemo pare teles, ki se verjetno prekrivajo, sledi bolj natančno preverjanje. Telesa so lahko kompleksna - uporabimo lahko hierarhije očrtanih teles, AABB, OBB drevesa,… Ko pridemo do konca, lahko testiramo še prekrivanje poligonov
  Iskanje **presekov dveh BVH dreves** poteka rekurzivno, primerjamo **pare vozlišč** obeh dreves:
  1. če se očrtani telesi obeh vozlišč ne sekata, vrnemo false
  2. če sta obe vozlišči lista, izračunamo presek predmetov v listih in vrnemo rezultat
  3. če je eno vozlišče list, drugo pa ne, računamo presek lista z vsemi nasledniki drugega vozlišča
  4. če sta obe vozlišči notranji, računamo presek vozlišča z manjšim volumnom z nasledniki vozlišča z večjim volumnom
- **Prekrivanje OBB**: Izrek o **ločitveni osi** narekuje da, če imamo dve konveksni telesi in najdemo os, na kateri se projekciji obeh teles ne prekrivata, se telesi ne prekrivata.
- Na najnižjem nivoju lahko preverjamo sekanje trikotnikov. Veliko pristopov, na **primer interval overlap method**: preveri prekrivanje na normalah. Če je nek trikotnik popolnoma na eni strani ravnine drugega, ni preseka. Izračunaj premico preseka obeh ravnin $\mathbf{L(t) = P + t(n_1 \times n_2)}$**.** Projiciramo oba trikotnika na - če se preseka sekata, se trikotnika prekrivata
- **Odziv na trk** tipično pomeni določitev novih položajev in hitrosti predmetov, ki so trčili (lahko pa tudi deformacije) – izračunati moramo torej **normalo trka** (predmeta se odbijeta okoli normale trka), **točen čas trka** in nato **novo gibanje** predmetov. Normala trka vpliva na izračun odziva. Za izračun normale lahko uporabimo položaj predmetov tik pred trkom - najdemo najbližji točki na obeh predmetih, tam je normala trka (krogle – normala je razlika med središčema). Hitrost po krtu v smeri normale trka se lahko zmanjša. Gibalna količina se spremeni: ob trku generiramo impulz moči j v smeri normale trka n.

10. Predstavitve predmetov

- Predstavitve 3D predmetov v RG:
  Ploskve: poligonske mreže, deljene ploskve, parametrične, implicitne
  Polna telesa: osmiška, BSP, CSG
  Neposredno z zajetimi podatki: voksli, oblaki točk, globinske slike
  Višje-nivojske strukture: graf scene
- Želimo predstavitev, ki bo čim bolj natančna, zgoščena, intuitivna, invariantna za afine transformacije, podpirala poljubno topologijo, zvezna/gladka, učinkovita za prikaz, učinkovita za operacije kot je računanje presekov
- Problem poligonskih mrež je **nezveznost** – mreža ni gladka. To lahko odpravimo z **delitvijo posameznih odsekov**, kjer grobo poligonsko mrežo iterativno razdeljujemo (limitira k zvezni ploskvi)
  Uporabljajo se v **animaciji** in **filmih**, zaradi enostavnega opisa kompleksnih ploskev, poljubne topologije, podpore lokalnim spremembam in zagotovljene zveznosti.
- Glavna razlika s poligonskimi mrežami je, da so deljene ploskve **bolj natančne, zgoščene in zvezne.**
- Algoritmi za deljenje se delijo glede na:
  - delitev (poligoni ali oglišča),
  - tip mreže (trikotniki ali štirikotniki),
  - **zveznost limite** (zvezen prvi odvod C1 ali drugi odvod C2)
  - **premik oglišč** (aproksimacija premakne originalna oglišča, interpolacija pa jih ne)
- **Deljenje Catmull-Clark** deli štirikotnike z aproksimacijo in zagotavlja C2 zveznost v notranjosti;
- Algoritem dodaja/spreminja oglišča kot uteženo vsoto obstoječih oglišč:
  1. Dodaj nova oglišča v središča vseh štirikotnikov
  2. Dodaj novo oglišče za vsak rob kot središče starih oglišč roba in oglišč sosednjih štirikotnikov
  3. Premakni originalna oglišča z uteženo vsoto sosednjih novih oglišč in originalnih oglišč
  4. Naredi nove štirikotnike iz izračunanih oglišč

Korake ponavljamo do želene globine, vsakič je izgled predmeta bolj zvezen.



novo oglišče iz roba: $\beta = \frac{3}{2k}$

novo oglišče v poligonu: $\gamma = \frac{1}{4k}$

staro oglišče: $1 - \beta - \gamma$

- **Deljenje Loop** je podobno Catmull-Clark, le da je za trikotnike:
  - Dodaj novo oglišče za vsak rob
  - Premakni originalna oglišča kot uteženo vsoto novih oglišč

  Različne variante obstajajo za izbor uteži ß

11. Parametrične predstavitve

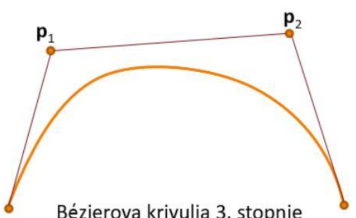
- Predmete lahko neposredno predstavimo **krivuljami in ploskvami**, ki jih krivulje definirajo. Veliko se uporabljajo v CAD aplikacijah in animacijah ker dosežemo gladkost.
- Poznamo:
  - **eksplicitne**
    (–) enkratne vrednosti in odvisnost od KS
    (+) enostavno vemo kdaj točka leži na krivulji
  - **implicitne**
    (–)odvisnost od KS
    (+) večkratne vrednosti in enostavno vemo ali je točka na krivulji
  - **parametrične predstavitve krivulj**
    (–) težko vemo ali točka leži na krivulji
    (+) imamo večkratne vrednosti in neodvisnost od KS
- Krivulja naj omogoča definicijo **poljubnih oblik** (večkratne vrednosti), **enostavno spreminjanje** z majhnim številom parametrov (točke na krivulji – interpolacijo, točke ob krivulji – aproksimacija) in **lepljenje krivulj naj bo gladko** (C1 ali C2 zveznost)

- Obliko in položaj parametričnih krivulj določajo **kontrolne točke** p_i– te določajo točke ob/skozi katere naj bi krivulja potekala:
  Za vsako točko lahko izberemo **mešalno** (bazično) funkcijo $B_i(t)$, ki določa kakšen je vpliv točke i na krivuljo za podan t. **Mešalne funkcije določajo** tip krivulje in jih izberemo tako, da omogočajo lokalen vpliv (kontrolna točka naj ima največji vpliv v svoji okolici). Želimo da so enostavne, zvezne, lokalne – interval na katerem so različne od 0 je majhen in omogočajo interpolacijo – končne točke naj ležijo na krivulji.
- Največkrat so mešalne funkcije polinomi (stopnja določa kakšna je oblika krivulje). Število kontrolnih točk je stopnja+1. Navadno uporabljamo **kubične polinome** (3. stopnja) ker so najnižji ne planaren red v 3D, ki omogoča C1 zvezne zlepke. Polinomi nižjih redov ne oogočajo fleksibilnosti pri oblikovanju, višji pa povzročajo nezaželena nihanja v krivulji in so računsko zahtevnejši

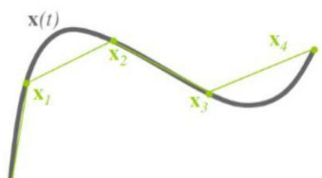
11.1. Bézierjeve krivulje

- So parametrične krivulje, ki se veliko uporabljajo v računalniški grafiki (2D orodja, TrueType fonti…) kjer so mešalne funkcije **Bernsteinovi polinomi**. Navadno je stopnja poljubna vendar največkrat uporabljamo kubične krivulje. Velja naslednje:
  - Prva in zadnja kontrolna točka sta na krivulji (interpolacija) notranje pa ne (aproksimacija)
  - Premik prve točke ne vpliva na tangento v zadnji in obratno
  - Krivulja v celoti leži znotraj konveksne ovojnice kontrolnih točk (enostavno računanje presekov)
  - Na ravnini nobena črta ne seka krivulje večkrat kot njenih kontrolnih črt
  - Je afino invariantna: enak rezultat dobimo s transformacijo kontrolnih točk
  - Za risanje imamo več načinov:
  - **Enakomerno vzorčenje**: krivuljo izrišemo z ravnimi črtami med točkami na krivulji z N ravnimi segmenti. N izberemo vnaprej, izračunamo N točk na krivulji. Če imamo malo točk je slab približek, če pa preveč pa je počasno, pride do prekrivanja segmentov.
  - **Adaptivno vzorčenje**: število segmentov se prilagaja ukrivljenosti. Segmentov je malo kjer je krivulja bolj ravna. Obstajajo različne sheme za določanje števila in lokacije segmentov.
  - **Rekurzivno (De Casteljaujevo) deljenje**: vsak del kubične krivulje je tudi kubična krivulja. Bezierovo krivuljo lahko razdelimo v več manjših Bezierovih krivulj, ki so bolj gladke kot celota. Kontrolne črte rekurzivno razdelimo na polovice, da dobimo dve Bézierjevi krivulji in to ponavljamo dokler ne dobimo skoraj ravnih segmentov, ki jih izrišemo kot črte. Kdaj je dovolj ravna? Ko je kot med segmenti dovolj majhen.

<u>Bezierove krivulje:</u>



Bézierova krivulja 3. stopnje



- Bernsteinovi polinomi stopnje n
  - $B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$
- Stopnja 3:
  - $B_{03}(t) = (1-t)^3$
  - $B_{13}(t) = 3t(1-t)^2$
  - $B_{23}(t) = 3t^2(1-t)$
  - $B_{33}(t) = t^3$

Izračun vrednosti krivulje:

$$\boldsymbol{p}(t) = \sum_{i=0}^{3} \boldsymbol{p}_i B_{i,3}(t) = \boldsymbol{p}_0(1-t)^3 + \boldsymbol{p}_1 3t(1-t)^2 + \boldsymbol{p}_2 3t^2(1-t) + p_3 t^3$$

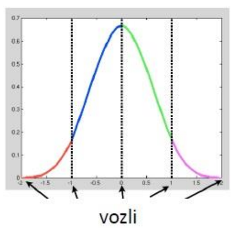
$$\mathbf{p}(t) = \begin{bmatrix} p_{0x} & p_{1x} & p_{2x} & p_{3x} \\ p_{0y} & p_{1y} & p_{2y} & p_{3y} \\ p_{0z} & p_{1z} & p_{2z} & p_{3z} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \mathbf{C} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

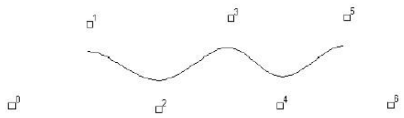
- Za daljše krivulje navadno kombiniramo krivulje 3. stopnje z **zlepki** (C^0 – se samo dotikajo, C^1 – enaka usmerjenost ali C^2 – enaka ukrivljenost)
- **Bezierovi zlepki** so lepljenje Bezierove krivulje 3. stopnje. Lepimo N segmentov, vsak je parametriziran med [0, 1]. Končna točka prejšnjega je začetna naslednjega, rabimo 3N+1 kontrolnih točk.
  Pri Bézierjevih zlepkih težko dosežemo C^2 zveznost (enak drugi odvod v stiku). C^1 zveznost pa zagotovimo z dodatno omejitvijo: tangenta v stični točki obeh krivulj naj bo enaka
  Slabosti Bézierovih zlepkov:
  - potrebujemo 3N+1 kontrolnih točk za N segmentov,
  - težko dosežemo C2 zveznost,
  - del kontrolnih točk je na krivulji, del ne
- **B zlepki** so posplošenje Bézierjevih zlepkov, kjer kontrolne točke niso več na krivulji. Na ekvivalenten zlepek potrebujemo manj kontrolnih točk kot pri Bezieru. Omogočajo lokalno kontrolo. Ohranjajo lastnost, da krivulja v celoti leži znotraj konveksne ovojnice kontrolnih točk. Ohranimo afino invariantnost.
  Njihove mešalne funkcije so rekurzivno definirane, kjer je u vektor, ki določa meje med polinomi, ki jih imenujemo **vozli** (vozel določa območje vpliva mešalne funkcije).

B zlepki s funkcijami $B_{k,d}$

- določa jih *n* kontrolnih točk in *n* mešalnih funkcij
- stopnja polinoma je *d*-1, omogočajo C^{d-2} zveznost
- vsaka mešalna funkcija je definirana na *d* podintervalih
- celotno območje t je torej razdeljeno na *n*+*d*-1 podintervalov, ki jih določa **vektor vozlov** (dolžine n+d)
  - kontrolna točka *k* je definirana na območju vozlov od *k do k+d*
- **lokalnost**
  - na vsak odsek krivulje vpliva *d* kontrolnih točk
  - vsaka kontrolna točka vpliva na največ *d* odsekov krivulje

$$\mathbf{p}(t) = \sum_{k=0}^{n-1} \mathbf{p}_k \mathbf{B}_{\mathbf{k,d}}(t)$$

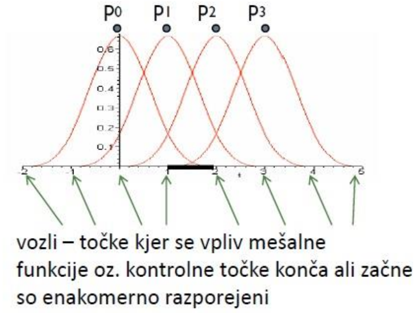




- Pri **enakomernih kubičnih B zlepkih** so vozli enakomerno razporejeni. Mešalna funkcija je enaka pri vseh kontrolnih točkah.

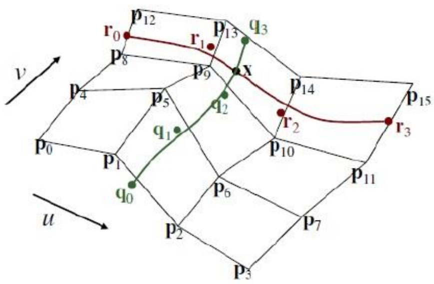
Mešalna funkcija: $B_{k,d}(t) = B_d(t-k)$

Krivulja: $$\mathbf{p}(t) = \sum_{k=0}^{n} \mathbf{p}_k \mathrm{B}_d(t-i)$$



$$\mathbf{S}_i(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}_{i-1} \\ \mathbf{p}_i \\ \mathbf{p}_{i+1} \\ \mathbf{p}_{i+2} \end{bmatrix}$$

- Pri neenakomernih racionalnih B zlepkih (**NURBS**), ki so najbolj standardne krivulje v 3D oblikovanju pa vozli niso nujno enakomerno razdeljeni – vektor vozlov je tako v NURBS pomemben saj določa bazne funkcije. Vsaka kontrolna točka ima utež, ki določa njen vpliv na krivuljo. Imamo lahko večkratne vozle. Začetna in končna točka NURBS imata navadno večkratni vozel enak redu krivulje. C2 zveznost je možna s kubičnimi zlepki, ki so lokalni. Krivulja v celoti leži znotraj konveksne ovojnice kontrolnih točk. Poleg afine invariantnosti je še projekcijsko invariantna. Predstavitev lahko zapišemo v homogenih koordinatah – dobimo racionalno funkcijo ko delimo z w komponento.
- **Parametrične ploskve** so razširitev krivulj v 2D. Namesto enega parametra f(t) imamo dva f(u, v) in 2D množico kontrolnih točk. Ploskve lepimo iz posameznih parametričnih krp, ki so sestavljene iz krivulj – uporabljamo za natančne matematične modele (avtomobili), vendar so problematične za splošno modeliranje zaradi težavnega sestavljanja različnih ploskev. V animacijah in filmih se največkrat uporabljajo deljene ploskve



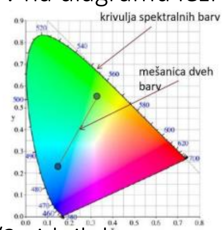
$$\mathbf{s}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{p}_{i,j} \mathrm{B}_i(u) \mathrm{B}_j(v)$$

12. Upodabljanje

- Slika nastane kot **interakcija** med objekti v sceni, lučmi in gledalcem
- **Bitna slika** je 2D tabela slikovnih elementov pikslov, ki je najmanjši element slike. Piksel vsebuje zapis barve (črno-bele, sivinske, barvne, večspektralne slike). Glede na število bitov za zapis barve pa ločimo 8, 16, 24, 32, … bitno globino slike.

12.1. Barve

- Vidna svetloba je elektromagnetno valovanje, vidimo nekje 390 – 750 nm. Fotoni nosijo optično informacijo, energija fotona pa je linearno povezana z valovno dolžino.
- V RG je pogled na svetlobo **poenostavljen** – ne gledamo posameznih fotonov posebej ampak presek čez čas. Histogram energij fotonov, ki predstavlja svetlobni spekter zaznamo kot barvo.
- V očesu imamo celice, ki delujejo kot fotoreceptorji. Dva tipa: čepki, ki so za vid ob dobri svetlobi in barve in paličice, ki so za svetlost in vid v slabi svetlobi (L valovna dol. - red, M - yellow, S - blue) **Metamerizem** - različne spektre zaznamo kot isto barvo – omogoča, da lahko zaznamo enake barve reproducirane s tiskalniki, TV zasloni, monitorji, čeprav so njihovi spektri dejansko različni
- Percepcija barv je kompleksna - dejansko se v vizualnem sistemu v možganih barve kodirajo kot razlike med odzivi čepkov (črna-bela, rdeča-zelena, modra-rumena)
- Pri **tridražljajski teoriji** lahko katerokoli barvo predstavimo s kombinacijo treh osnovnih barv. Nabor parametrov, ki določajo barve imenujemo **barvi prostor. Tridražljajski eksperiment** je bil eksperiment, s katerim so želeli ugotoviti, kakšne so jakosti treh osnovnih barv RGB za predstavitev poljubne monokromatske barve. Eksperiment je naredila CIE, dobljeni prostor barv imenujemo **CIE RGB**. Nekaterih barv niso mogli predstaviti oz. bi rabili negativne vrednosti RGB.
- Najbolj razširjen umetno ustvarjen barvni prostor je **CIE 1931 XYZ** prostor in lahko predstavi vse vidne barve – uporablja se za pretvorbe med različnimi prostori. Je umetno ustvarjen prostor, ki nima negativnih vrednosti. Ima tri barvne vrednosti, Y je bolj povezan s svetlostjo
- **CIE XYZ barvni prostor** lahko obravnavami kot 3D koordinate v prostoru, vendar veliko XYZ vrednosti ne predstavlja vidne svetlobe
- **Barvni diagram CIE** – za lažjo predstavitev so uvedli prostor xyY (Y – svetlost, xy – deleža barvnih vrednosti). Če prostor izrišemo so vidne barve v podkvasti obliki. Krivulja se imenuje krivulja spektralnih barv in predstavlja enobarvne svetlobe. Mešanica dveh barv na diagramu leži na črti, ki povezuje barvi.
- **Barvni obseg** vsake naprave, ki deluje na mešanju primarnih barv, je nabor vseh barv, ki jih naprava lahko reproducira. Velja, da z mešanjem osnovnih barv lahko ustvarimo le barve, ki ležijo znotraj konveksne ovojnice osnovnih barv na barvnem diagramu CIE. Točke izven konveksne ovojnice bi ustrezale negativnim vrednostim



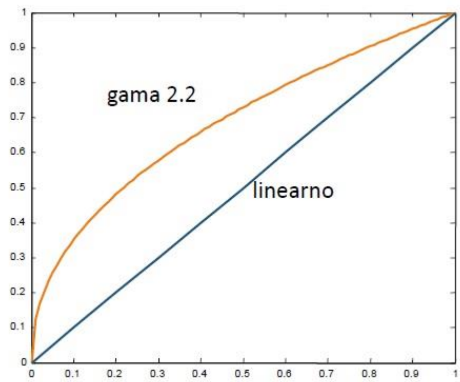
- **sRGB** je najpogostejši barvni prostor (monitorji, fotoaparati). Pokrije 1/3 vidnih barv
- **AdobeRGB** pokrije cca. 50% vidnih barv
- **PhotoPro** ima imaginarni modri in zeleni komponenti. Pokrije ccs. 90% vidnih barv, vendar je redundanten – nekatere kombinacije niso vidne.
- Ker je XYZ umeten veliko vrednosti v tem prostoru ne predstavlja nobene vidne barve (prostorsko potraten) – zato se uporablja veliko drugih prostorov (npr. sRGB, AdobeRGB...) XYZ pa se uporablja za pretvorbe med prostori, saj je neodvisen od reprodukcijskih naprav, predstavi pa lahko vse vidne barve.
  Pretvorbe med prostori gredo prek XYZ, matematično so pretvorbe množenje matrik. Barve, ki zaradi omejitev naprav pri konverziji padejo izven barvnega obsega naprave, se preslikajo glede na rendering intent. **Sistemi za upravljanje z barvami** se uporabljajo za konverzije med barvnimi prostori (vmesni prostor je XYZ) Če želimo recimo natisniti sRGB sliko posneto z mobilnim fotoaparatom: CMS pretvori sliko iz sRGB prostora v XYZ prostor za tisk pretvori XYZ v CMYK s tiskalnikovim ICC profilom (sRGB => XYZ => prostor tiskalnika)
- **RGB** prostori temeljijo na **seštevanju** barv, za tiskalnike pa rabimo drug prostor, saj se črnilo odšteva, npr. rumeno črnilo na belem listu odšteje modro, odbije rdečo in zeleno. Tak prostor se imenuje **CMY(K)**. Komponento K imamo, saj brez te ne moremo reproducirati temnih delov.

- Ker RGB ni intuitiven za določanje barv v risarskih programih imamo še **HSL in HSB**, ki sta bolj intuitivna – nista barvni spekter ampak le alternativni zapis nekega RGB prostora
  Hue – barvni odtenek
  Saturation – nasičenost
  Lightness/brightness – svetlost
  Iz HLS/HSB lahko pretvorimo v RGB in obratno.
- **Weberjev zakon**: Razmerje med obstoječo energijo in energijo, ki jo moramo obstoječi energiji dodati, da zaznamo razliko, je konstantno. Pri majhnih intenzitetah smo bolj občutljivi na majhne spremembe kot pri velikih.

$$\frac{\Delta I}{I} = k$$

Zato je nesmiselno barve kodirati linearno ampak z **gama korekcijo**.

- **Gama korekcija** pomanjkljivost odpravlja s tem, da sliko ob zajemu zakodiramo $(1/\gamma)$
  - $s = cr^{1/\gamma}$
  - porabimo več bitov za nižje intenzitete
  - tipična gama je 2.2
- Digitalni fotoaparati, kamere zajeto sliko že gama zakodirajo $(1/\gamma)$
  - jpg, mpeg, dvd ... vsebujejo gama zakodirane vrednosti
- Z gama korekcijo v računalnikih upravlja sistem za upravljanje z barvami
  - ob prikazu se slika ustrezno dekodira glede na napravo, ki jo prikazuje
- Barvni prostori lahko definirajo svoje načine gama kodiranja, npr. sRGB

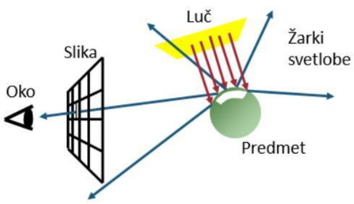


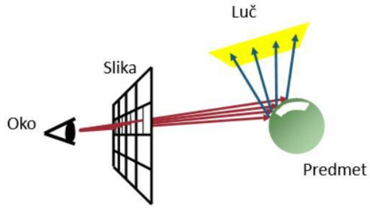
$$C_{\mathrm{srgb}} = \begin{cases} 12.92 C_{\mathrm{linear}}, & C_{\mathrm{linear}} \le 0.0031308 \\ (1+a) C_{\mathrm{linear}}^{1/2.4} - a, & C_{\mathrm{linear}} > 0.0031308 \end{cases}$$

13. Globalno osvetljevanje

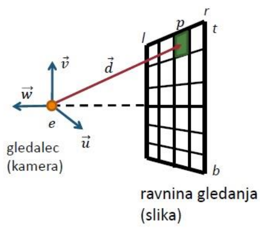
13.1. Sledenje žarku – Ray tracing

- Je metoda **globalnega osvetljevanja**, kjer **simuliramo svetlobne žarke** (naravna osvetlitev, odboji, lom svetlobe, mehke sence, globinsk ostrina, zabrisano gibanje)
  Uporaba v animaciji, filmih, simulacijah...
  V resnici žarke generirajo svetlobni viri in ti potujejo do očesa, vendar bi za tak izračin morali slediti veliko žarkov, ki sploh ne pridejo do očesa – **zato obrnemo situacijo in sledimo žarkom od očesa**





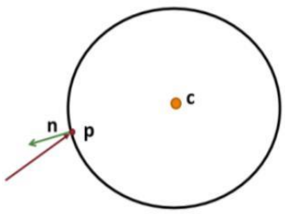
- Prvi del algoritma je **metanje žarka** (ray casting), kjer sledimo žarku svetlobe od očesa do presečišča s prvim predmetom, kjer gre en žarek skozi vsak piksel v končni sliki – v presečišču s predmetom se izračuna barva z osvetlitvenim modelom, če pa žarek ne seka nobenega predmeta je piksel črn.
  Uporablja se pri prikazu polnih teles, prikazu vokslov, odstranjevanju zakritih površin.
  Konstrukcija žarka:
  - Kako za nek piksel slike izračunati **smer žarka**
  - Konstrukcija žarka:
    - koordinatni system kamere je $[\vec{u}, \vec{v}, \vec{w}]$
    - slika (velikosti $n_x \times n_y$) je pravokotna na $\vec{w}$ kamere in na razdalji d od kamere
    - koordinate (u, v, w) piksla (i, j) v koordinatah kamere so:
      - $u = l + (r - l)\frac{i+0.5}{n_x}$
      - $v = b + (t - b)\frac{j+0.5}{n_y}$
      - $w = -d$
    - izvor žarka je e
    - smer žarka: $\vec{d} = u\vec{u} + v\vec{v} - d\vec{w}$
    - piksel na sliki: $p = e + \vec{d}$
  - Parametrična **enačba žarka**:
    - $r(t) = e + t(p - e) = e + t\vec{d}$

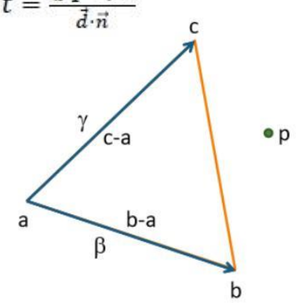


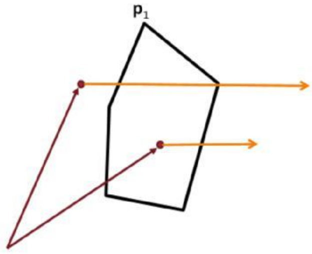
- Žarek je predstavljen kot $p(t) = e + td$ Iščemo torej vrednost t pri kateri žarek preseka predmet. Najmanjši t > 0 bo najbližji presek. Iskanje presekov je najbolj časovno zahteven del pri tej metodi.
- Velikokrat uporabljamo **preseke s kroglo** zaradi enostavnosti – pri dveh realnih pozitivnih ničlah je manjša prvi presek. Pri dvojni ničli imamo tangento pri eni pozitivni in eni negativni ničli se žarek začenja v krogli in gre ven. Če imamo kompleksi ničli, žarek ne seka krogle. Dovolj je, da pogledamo, če je izraz pod korenom negativen, da vemo ali žarek seka kroglo ali ne. Za izračun osvetlitve v točko preseka potrebujemo tudi normalo.
  $\vec{n} = \frac{p-c}{|p-c|}$
- Pri **preseku s trikotnikom** dobimo sistem treh enačb (x,y,z) in treh neznank (t, β , γ), ki ga rešimo. Če velja $t > 0$ in $0 < \gamma < 1$ in $0 < \beta < 1-\gamma$, je točka znotraj trikotnika
  $$p = a + \beta(b - a) + \gamma(c - a)$$
  Točko predstavimo kot: $p = \alpha a + \beta b + \gamma c, \quad \alpha + \beta + \gamma = 1$
  Vstavimo enačbo žarka: $e + t\vec{d} = a + \beta(b - a) + \gamma(c - a)$
- Pri preseku s **planarnim poligonom** najprej računamo presek z ravnino na kateri je poligon, nato pa ostane **point in polygon** problem, kjer ne vemo ali je točka v poligonu? Štetje kolikokrat vodoraven žarek iz točke preseka poligon, liho št. – točka je v poligonu.

Enačba ravnine: $(p - p_1) \cdot \vec{n} = 0$

Vstavimo enačbo žarka in dobimo t: $t = \frac{(p_1 - e) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}$







- Po izračunu preseka moramo izračunati še **osvetlitev** – računamo torej v vsaki točki (podobno lokalni osvetlitvi). Osvetlitveni model je lahko Phongov.

$$I = k_a I_a + \boldsymbol{V_i} I_i \left(k_d (\vec{L} \cdot \vec{N}) + k_s (\vec{V} \cdot \vec{R})^p \right)$$

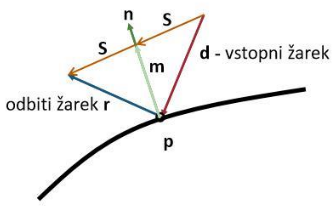
- Metanje žarka upošteva tudi sence - v vsaki točki preseka pošljemo **senčni žarek** (shadow ray) proti vsaki luči. Če je na poti kak predmet, je točka v senci (V = 0), sicer ni (V = 1)
- Za **sledenje žarku** nadgradimo metanje žarka tako, da žarku sledimo tudi po prvem dotiku s predmetom v dve smeri:
  - **popolni odboj** za materiale, ki imajo zrcalno komponento
  - v primeru prozornega materiala tudi **prepuščeni žarek**;

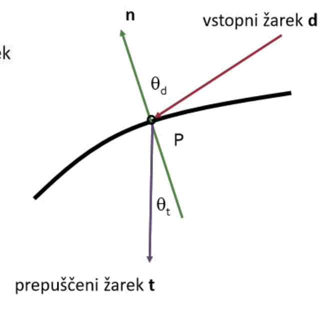
  Na obeh žarkih ponovimo postopek sledenja rekurzivno. Rekurzijo pri nekem žarku ustavimo, ko:
  - Žarek zadane luč (dobi barvo luči)
  - Žarek ne zadane ničesar (tema)

  Ta dva pogoja nista dovolj. Omejiti moramo globino rekurzije:
  - Koliko nivojev rekurzije rabimo? Odvisno od kompleksnosti scene npr. 4
  - Z večjo globino raste tudi kompleksnost, saj moramo slediti vse več žarkom in računati vse več presekov
- Odboj: odbiti žarek računamo kot popolni odboj.
- Lom svetlobe: prosojni materiali (steklo, voda) prepuščajo svetlobo in jo tudi lomijo. Svetloba se upočasni, ko preide bolj gost medij. Velja Snellov zakon:
  - $\eta_d \sin \theta_d = \eta_t \sin \theta_t$
    - η_d - lomni količnik v snovi vstopnega žarka
    - η_t - lomni količnik v snovi prepuščenega žarka

  $$\eta = \frac{hitrost\ svetlobe\ v\ vakuumu}{hitrost\ svetlobe\ v\ mediju}$$

  - $\eta_{zrak} \sim 1, \eta_{voda} += 1{,}33, \eta_{steklo} = 1{,}5$



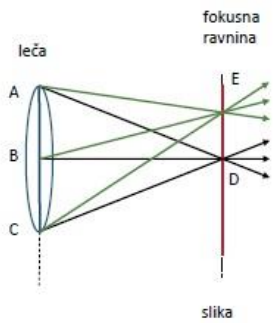


- **Osvetlitev** se računa kot pri metanju žarka, le da prištejemo še deleže svetlobe, ki pridejo od obeh novih žarkov po rekurzivnem sledenju, tako da imamo dejansko osvetlitev izračunano šele po koncu rekurzije.

$$I = k_a I_a + I_i \left(k_d (\vec{L} \cdot \vec{N}) + k_s (\vec{V} \cdot \vec{R})^p \right) + \boxed{k_r I_r + k_t I_t}$$

- Pomanjkljivost algoritma za sledenje žarkom je precej **umeten izgled** slik (ostri robovi, trde sence, vse v fokusu...). To odpravlja sledenje porazdeljenim žarkom kjer namesto enega žarka delamo v vsaki fazi z več žarki
- Za **mehčanje robov** pošljemo skozi vsak piksel namesto enega več žarkov (lahko jih porazdelimo enakomerno ali naključno – jitter). Število žarkov je lahko fiksno ali adaptivno. Končna barva je tako uteženo povprečje žarkov. Dobimo bolj mehke prehode med ostrimi kontrasti.
- Za **mehke sence** naj luči zajemajo večjo površino. V luč pošljemo več naključno porazdeljenih senčnih žarkov in nato seštejemo doprinose.
  Število zadetkov/števil žarkov = % osvetlitve
- Pri standardnem sledenju žarkov so odboji preveč podobnimi zrcalu – idealni. Zaradi grobosti materialov so v realnosti precej bolj zabrisani. Idejo prenesemo tudi na odbite žarke, namesto enega jih ustvarimo več v nekoliko naključni smereh (uporabimo lahko lastnosti materiala za določanje števila smeri).
- Kot za odbite žarke, lahko tudi pri prepuščenih žarkih ustvarimo več naključno porazdeljenih žarkov okoli idealnega žarka. Dobimo efekt prosojnosti oz. pol-prosojnosti.
- Globinska ostrina je področje okoli fokusne razdalje, v katerem je slika še sprejemljivo ostra.
- RG kamera:
  - idealna, vsi žarki gredo iz ene točke (leča velikosti 0), vse je ostro
  - ena točka v sceni = ena točka na sliki
- Leča: bolj realističen model, žarki razporejeni po leči
  - ena točka na sceni = krožec na sliki, razen za točke v fokusni ravnini

  Globinsko ostrino simuliramo s porazdelitvijo žarkov po površini leče. Slika je postavljena na fokusno ravnino.



- Zabrisano gibanje je gibanje, ko žarke porazdelimo po času in povprečimo. Predmeti, ki se premikajo bodo zamegljeni.
- Ker je sledenje žarkom počasna metoda (računanje presekov) moramo uporabiti **metode za pohitritev**
  <u>Hierarhije očrtanih teles</u>
  Kompleksne predmete postavimo v telesa s katerim enostavno izračunamo preseke. Če žarek ne seka očrtanega telesa, ne seka predmeta v njem. Očrtana telesa postavimo v drevo. Vozlišče drevesa je očrtano telo celotnega poddrevesa, v listih pa so predmeti. Ko sledimo žarkom, začnemo pri korenu drevesa in nadaljujemo proti predmetom v listih.
  <u>Razdeljevanje prostora (navadna mreža, octree, BSP drevo)</u>
  Prostor razdelimo na podprostore, hierarhično izračunamo preseke žarka dokler ne pridemo do posameznih primitivov, s katerimi izračunamo presek.
- Sledenje žarka (upoštevamo le zrcalne odboje) : prava globalna osvetlitev (upoštevamo tudi difuzne odboje – sledenje poti)
- Z sledenjem žarkom tako pridobimo:
  (+) prosojnost, odboje, sence, realizem, dobra hitrost s primerno razdelitvijo prostora
  (–) za fotorealistične efekte velika količina žarkov in je počasno, ni popolna globalna osvetlitev (ni difuznih odbojev, puščanja barva in kavstike) .