

Počítačová grafika pro hry

Státnice 2021

kód	Předmět	Kredity	ZS	LS
NPGR003 ↗	Základy počítačové grafiky	5	2/2 Z+Zk	—
NPGR004 ↗	Počítačová grafika II	5	—	2/1 Z+Zk
NPGR019 ↗	Realtime grafika na GPU	5	—	2/1 Z+Zk
NPGR033 ↗	Grafika pro počítačové hry	6	—	2/2 Z+Zk

Za případné chyby může Mikuláš Hrdlička, jenž ještě autorem.

1. Obsah

[1. Obsah](#)

[2. Základní matika a pojmy v grafice](#)

[2.1. Homogenní souřadnice](#)

[2.2. Afinní a projektivní transformace v rovině a v prostoru](#)

[2.3. Kvaterniony](#)

[2.4. Spline funkce](#)

[2.5. Interpolace kubickými spliny](#)

[2.6. Bézierovy křivky](#)

[2.7. Catmull-Rom spliny](#)

[2.8. B-spliny](#)

[2.9. de Casteljau a de Boor algoritmus](#)

[3. Animace postav, skinning, rigging](#)

[4. Detekce kolizí](#)

[5. Obrázky, textury a práce s nimi](#)

[5.1. 2D Fourierova transformace a konvoluce](#)

[5.2. Vzorkování a kvantování obrazu](#)

[5.3. Anti-aliasing](#)

[5.4. Textury](#)

[5.5. Změna kontrastu a jasu](#)

[5.6. Kompozice polopřhledných obrázků](#)

[5.7. Principy komprese rastrové 2D grafiky](#)

[5.8. Komprese videosignálu](#)

[5.9. Časová predikce \(kompenzace pohybu\)](#)

[5.10. Standardy JPEG a MPEG.](#)

[6. Realtime grafika, vykreslování scén, světel a stínů](#)

[6.1. Reprezentace 3D scén](#)

[6.2. Výpočet viditelnosti](#)

[6.3. Výpočet vržených stínů a měkké stíny,](#)

[6.4. Rozptyl světla pod povrchem](#)

[6.5. Modely osvětlení a stínovací algoritmy](#)

[6.6. Rekurzivní sledování paprsku](#)

[6.7. Fyzikální model šíření světla \(radiometrie, zobrazovací rovnice\)](#)

[6.8. Algoritmus sledování cest](#)

[6.9. Předpočítané globální osvětlení](#)

[6.10. Výpočet globálního osvětlení v reálném čase](#)

[6.11. Stínování založené na sférických harmonických funkcích](#)

[6.12. Předpočítaný přenos radiance](#)

[7. Architektura, shadery a práce s GPU](#)

[7.1. Architektura grafického akcelerátoru](#)

[7.2. Předávání dat do GPU](#)

[7.3. Textury a GPU buffery](#)

[7.4. Programování GPU - shadery](#)

[7.5. Základy OpenGL, GLSL, CUDA a OpenCL](#)

[7.6. Pokročilé techniky práce s GPU](#)

[7.7. Architektura herní engine.](#)

Zdroje a videopřednášky

kód	Předmět	Kredity	ZS	LS
NPGR003 ↗	Základy počítačové grafiky	5	2/2 Z+Zk	—
NPGR004 ↗	Počítačová grafika II	5	—	2/1 Z+Zk
NPGR019 ↗	Realtime grafika na GPU	5	—	2/1 Z+Zk
NPGR033 ↗	Grafika pro počítačové hry	6	—	2/2 Z+Zk

- [Základy PC grafiky přednášky - YT Playlist](#) (20 hours, 32 minutes, 23 seconds)
- [Počítačová Grafika II](#) (Jmenuje se jinak předmět, ale kód sedí, a sajdy taky zmiňovali správnej název. Videa se mi nepovedlo proklikat se na)
- [Realtime Grafika na GPU přednášky](#) (4 hodiny, chybí ale 6/12 přednášek)
- [Grafika pro PC Hry přednášky -](#) (13 hodin, chybí 2 přednášky)

2. Základní matika a pojmy v grafice

Super, půlka přednášek používá vektory jako sloupce, druhá používá vektory jako řádky. Podle toho se pak mění jestli třeba u translační matice je translace poslední řádek, nebo poslední sloupec, a je v tom prostě mega bordel. To, co je anglicky vypadá, že bere vektory jako sloupce... Český přednášky ze základů to zas berou jako řádky a násobí zprava. Většina na netu je bere jako sloupce, takže pak vypadaj jinak. Radost to je.

Takže, snažím se většinu slajdů držet podle zápisu z Základů Grafiky, takže máme řádkový vektory, a násobíme zprava:

Násobení vektoru souřadnic maticí zprava

- à la DirectX (OpenGL to má obráceně)
- kartézské souřadnice bodu $[x, y]$ tvoří řádkový vektor
- transformační matice je čtvercová (v rovině má rozměr 2×2)

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}$$

Třeba v první podčásti o zakladních operacích to ale jsou slajdy z Grafiky pro GPU, který sou samozřejmě opačně, a používají násobení sloupcem, a zleva. Takže třeba translační matice pak vypadá takhle:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

$$: [x, y, w] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Oproti

Není to moc těžký převádět, vypadá to, že stačí prohodit poslední sloupec za poslední řádek v matici/matici transponovat. Kdoví, jestli to tak ale funguje vždycky. (Třeba

rotace jsou stejný, takže předpokládám že to asi bude tak, že je jenom prohozená translace s projekcí)

Některý základní operace se kterejma se počítá:

Skalární součin vektorů (dot product), je definován jako součet součinů jednotlivých složek vektoru. Tj $(a_1, b_1) \cdot (a_2, b_2) = (a_1 \cdot b_1) + (a_2 \cdot b_2)$. Výsledek je teda číslo. Dá se počítat i jako součin délek vektorů, vynásobené cosinem uhlí mezi nimi:

SCALAR (DOT) PRODUCT

► Definition:

$$\mathbf{p} \cdot \mathbf{q} = \sum_i p_i q_i$$

► Value:

$$\mathbf{p} \cdot \mathbf{q} = \|\mathbf{p}\| \|\mathbf{q}\| \cos \alpha$$



► Matrix notation:

$$\mathbf{p} \cdot \mathbf{q} = \mathbf{p}^T \mathbf{q} = [p_0, \dots, p_{n-1}] \begin{bmatrix} q_0 \\ \vdots \\ q_{n-1} \end{bmatrix}$$

Taky se dobře počítá maticovým násobením a to tak, že vezmu jeden vektor jako řádkové, a vynásobím ho tím druhým jako by byl sloupové. Pokud sou na sebe vektory **kolmý**, tak jejich dot product je **0**.

Používá se třeba při projekcích, když potřebuju zjistit jakou složku vektoru má daná osa, což se hodí třeba při změně báze do jinejch součadnic.:

VECTOR PROJECTION

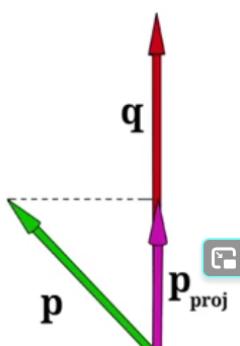
► Projection on another vector:

$$\mathbf{p}_{proj} = \frac{\mathbf{p} \cdot \mathbf{q}}{\|\mathbf{q}\|} \mathbf{q}$$

Tj když jako q dám osu X, zjistím tím x souřadnici/složku p.

► Matrix notation ($\mathbf{q}\mathbf{q}^T$):

$$\mathbf{p}_{proj} = \frac{1}{\|\mathbf{q}\|^2} \begin{bmatrix} q_x^2 & q_x q_y & q_x q_z \\ q_x q_y & q_y^2 & q_y q_z \\ q_x q_z & q_y q_z & q_z^2 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$



► Useful for repeated projections, embedding in matrix expressions

Další je **vektorové součin** (funguje jenom ve 3D a v 7D, v

ostatních dimenzích vektorů neexistuje). Jak se počítá v 7d netuším.

CROSS PRODUCT

► Definition:

$$\mathbf{p} \times \mathbf{q} = [p_y q_z - p_z q_y, p_z q_x - p_x q_z, p_x q_y - p_y q_x]$$

► As formal determinant:

$$\mathbf{p} \times \mathbf{q} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ p_x & p_y & p_z \\ q_x & q_y & q_z \end{vmatrix}$$

► Matrix formulation:

$$\mathbf{p} \times \mathbf{q} = \begin{bmatrix} 0 & -p_z & p_y \\ p_z & 0 & -p_x \\ -p_y & p_x & 0 \end{bmatrix} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

Záleží u něj na pořadí. Blbě se pamatuje. Záleží na pořadí, tj $\mathbf{p} \times \mathbf{q} \neq \mathbf{q} \times \mathbf{p}$.

Dobře se pamatuje tak, že to je determinant z 3x3 matice, kde dám do prvního řádku $\mathbf{i}, \mathbf{j}, \mathbf{k}$, a do dalších složky vektoru.

Conversion to matrix multiplication [\[edit\]](#)

The vector cross product also can be expressed as the product of a [skew-symmetric matrix](#) and a vector.^[16]

$$\mathbf{a} \times \mathbf{b} = [\mathbf{a}]_{\times} \mathbf{b} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$
$$\mathbf{a} \times \mathbf{b} = [\mathbf{b}]^T_{\times} \mathbf{a} = \begin{bmatrix} 0 & b_3 & -b_2 \\ -b_3 & 0 & b_1 \\ b_2 & -b_1 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix},$$

where superscript T refers to the [transpose](#) operation, and $[\mathbf{a}]_{\times}$ is defined by:

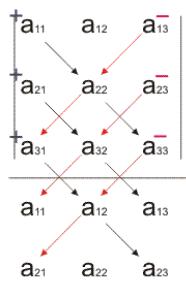
$$[\mathbf{a}]_{\times} \stackrel{\text{def}}{=} \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}.$$

Determinant matice se taky počítá a pamatuje blbě, na 3x3 matice je ale **Sarrusovo** pravidlo. Když se do něj dosadí výše uvedená matice, výjde nám číslo, který se zjednoduší na $<\text{něco}> * \mathbf{i} + <\text{něco}> * \mathbf{j} + <\text{něco}> * \mathbf{k}$. Ty jednotlivé části sou stejný jako složky vektoru co hledáme, tj jenom umázneme $\mathbf{i}, \mathbf{j}, \mathbf{k}$ a jejich násobek použijeme jako tu danou složku vektoru (|) zančí u matice determinant).

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix}$$

$$\det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}a_{22}a_{33} + a_{21}a_{32}a_{13} + a_{31}a_{12}a_{23} - (a_{13}a_{22}a_{31} + a_{23}a_{32}a_{11} + a_{33}a_{12}a_{21})$$

Pro lepší zapamatování si můžeme přepsat první dva řádky pod determinant a jednotlivé součiny dostaneme z jednotlivých diagonál. Tomuto postupu se říká Sarrusovo pravidlo. Sarrusovo pravidlo můžeme aplikovat i horizontálně, tak, že si přepíšeme první dva sloupečky na pravou stranu.



Nebo si můžu pamatovat tohle:

b_x	c_x	b_y	c_y	b_z	c_z	$b_y c_z - c_y b_z$	$c_x b_z - b_x c_z$	$b_x c_y - c_x b_y$
-------	-------	-------	-------	-------	-------	---------------------	---------------------	---------------------

Mnemonic to calculate a cross product in vector form

K čemu je - je to vektor, co je kolmej na rovinu tvořenou těma dvouma vektorama.

CROSS PRODUCT II

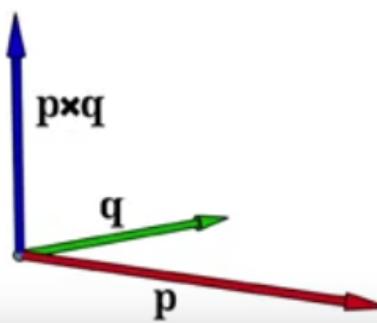
- Perpendicular to \mathbf{p}, \mathbf{q} :

$$(\mathbf{p} \times \mathbf{q}) \cdot \mathbf{p} = (\mathbf{p} \times \mathbf{q}) \cdot \mathbf{q} = 0$$

- Size:

$$\|\mathbf{p} \times \mathbf{q}\| = \|\mathbf{p}\| \|\mathbf{q}\| \sin \alpha$$

- Follows *right hand rule*



Násobení matic je celkově další základní věc, která se hodí připomenout. Aby matice šla násobit, musí mít první stejně sloupců jako druhá řádků (nebo naopak, podle toho jak násobíme).

Když násobíme matici o velikosti $K \times N$ a $N \times K$, vznikne matice velikosti $N \times N$.

Pro její prvky platí, že prvek na i,j pozici výsledné matice je **skalární součin** i ho řádku první matice, s j ím sloupcem druhé matice. Tj:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj},$$

Prostě:

$$\begin{array}{|c|c|c|} \hline & & \\ \hline i & \text{j} \rightarrow & \\ \hline & & \\ \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \bullet \begin{array}{|c|c|c|} \hline & & k \\ \hline & & j \downarrow \\ \hline & & \\ \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline & & k \\ \hline & & \\ \hline i & & \text{i} \\ \hline & & \\ \hline \end{array}$$

2.1. Homogenní souřadnice

Homogenní souřadnice popisují způsob, jak popisovat body v momentě, kdy přejdeme z affiní geometrie do geometrie projekční. Jsou to vlastně obdoba kartézských souřadnic v projekční geometrii.

Affiní geometrie je zjednodušení geometrie, když vynecháš úhly a vzdálenosti. Tj pracuješ jenom s bodama, vektorama a přímkami.

Projekční geometrie je geometrie, která zkoumá vlastnosti co se nemění u projektivních transformací. Většinou se jako model používá projektivní rovina, pro kterou platí tyhle axiomy:

- Každé dva různé body leží na právě jedné přímce
- Každé dvě různé přímky se protínají právě v jednom bodě
- Existují alespoň 4 různé body, z nichž žádné tři neleží na přímce
- Existují alespoň 4 různé přímky, z nichž žádné tři se neprotínají v bodě.

Toho se dosáhne tak, že se prostě přidá bod v nekonečnu, aby se dal nějak zjistit průnik rovnoběžek.

Tl;dr, homogení souřadnice získáme tak, že ke kartézskejm přidáme další složku, kterou dáme rovnu 1. Pokud tedy máme bod $[a,b,c]$, v homogeních souřadnicích to bude $[a,b,c,1]$.

Pro převedení bodu nazpátek do kartézskejch, prostě vdělíme poslední souřadnicí:

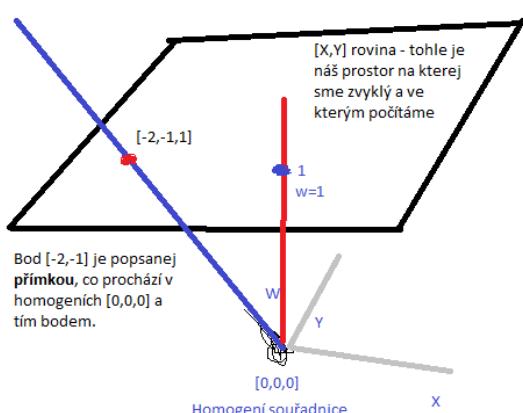
$[a/w, b/w, c/w]$

To nám umožňuje provádět další operace a transformace za pomocí násobení matic (třeba translaci), a celkově jednodušeji pracovat s bodama a vektorama.

Snažil jsem se hrozně dlouho přijít na to, jaký to dává smysl a jaká myšlenka za tím je. Myšlenka je následující. Pokusím se odlišit o kterém souřadnicovém systému mluvím tím, že když se bavím o přímkách a bodech v prostoru co popisujeme, označím to 2D, a pokud to bude v homogením/projekčním prostoru, označím to 3D.:

Místo toho, aby jsme 2D prostor popisovali jako body v X a Y ose ve 2D, budeme se na ně dívat jako na přímky ve 3D, který prochází počátkem souřadnic, tj $[0,0,0]$. To, co nás zajímá, je způsob, jakým se promítou na **projekční rovinu**, což je v našem případě rovina se souřadnicí $w = 1$. Náš prostor, kterej byly předtím osy X a Y se teda posunul, a teď je celej v $[X, Y, 1]$. V podstatě to funguje jako CRTčko, který z bodu $[0,0,0]$ posílá paprsky, a to, kde se

promítnout do roviny s $W=1$ je to, kam se vykreslí v našich původních souřadnicích. A homogení souřadnice popisují ten paprsek. **Bod** ve 2D je tedy **přímka** ve 3D/projekční geometrii. **Přímka** ve 2D je potom **rovina** v projekční/3D



Pro popsání přímky je to stejný - přímku ve 2D popisuje **rovina** v homogeních souřadnicích, která prochází bodem $[0,0,0]$, a tam, kde protne projekční rovinu ($W=1$) tak tam je přímka.

Proč to vzniklo? Bylo potřeba nějak zajistit, aby každá dvojice 2D přímek měla jenom bod, ve kterém se protíná, a zároveň aby každýma dva body procházela nějaká přímka. V podstatě sme potřebovali přidat 2D bod v nekonečnu, který se prohlásí za průnik 2D rovoběžek, aby nám platili axiomu. Z obrázku vejš je vidět, že se nám to povedlo - jediný přímky ve 3D (což popisuje bod v projekční rovině), který rovinu neprotnou (tudíž nejdou zobrazit a jsou v nekonečnu) jsou ty, přímky, které mají souřadnici $W=0$.

Zároveň sme zajistili, jak zjistit průnik dvou 2D rovoběžek - v homogených je to průnik dvou 3D **rovin** co obě prochází bodem $[0,0,0]$, a který na sebe sou kolmý. Tím pádem ale jejich průnik je 3D přímka, co má souřadnici $W=0$:

<https://www.geogebra.org/3d/v7mr4jgx>

Jediný co teda zbývá je převod z Homogených do Kartézských: prostě vydělíme všechno souřadnicí W .

$$\begin{bmatrix} x & y & z \end{bmatrix} \rightarrow \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \rightarrow \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix} \quad (w \neq 0)$$

2.2. Afinní a projektivní transformace v rovině a v prostoru

Transformace popisují způsob, kterým je pracováno s body/objekty v souřadnicovém prostoru. Například pokud chci posunout bod o kus vedle, je to transformace.

Transformace se dělí na dva typy:

- **Afinní transformace** jsou transformace v prostoru, které zachovávají dimenze podprostorů (tj, přímky na přímky, body na body), a paralelismus (tj, rovnoběžný přímky jsou stále rovnoběžný, a poměr délek *rovnoběžných* úseček). Co už **nemusí** nutně zachovávat jsou úhly, a vzdálenosti jednotlivých bodů. (poměr vzdáleností bodů na stejně úsečce se ale zachová)
 - Posunutí
 - Otočení kolem osy souřadnic
 - Změna měřítka (scale)
 - Zkosení

- Zrcadlení
- **Projektivní** - transformace, který se většinou snaží zobrazit body v souřadnicích nižší dimenze, na průmětnu (což je teda rovina/přímka na kterou promítám). Co zachovávají záleží na typu projekce.

Jedním z pojmu jsou **Rigid-body transform** - což sou transformace co méně polohu ale né tvar, tj jenom posunutí a rotace.

Matice operací - jsou to matice, kterejma násobíme vektor, když chci provést danou operaci.

Nejdřív jsou operace v rovině, kde **řádkovej vektor souřadnic** násobíme **zprava** maticí transformace:

Homogenní transformační matice



Posunutí
("translation")

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Otočení ("rotation")
kolem počátku

$$R(\alpha) = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Zmenšení / zvětšení
("scale")

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Zkosení
("shear")

$$Sh(a, b) = \begin{bmatrix} 1 & a & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Jednotlivý transformace se dají skládat tak, že nejdřív vynásobím mezi sebou matice, a teprve výslednou maticí vynásobím ten bod. Můžu tím ušetřit spousty času, když např. Dělám sekvenci transformací několikrát:

Složené transformace

$$\left(\left([x, y, w] \cdot T_1 \right) \cdot T_2 \right) \cdot T_3 = [x, y, w] \cdot (T_1 \cdot T_2 \cdot T_3)$$

Otočení o úhel α kolem bodu $[x, y]$

$$R(x, y, \alpha) = T(-x, -y) \cdot R(\alpha) \cdot T(x, y)$$

U skládání a transformací ale může dojít k problému, pokud potřebuju zachovat normálové vektor tak, aby byl pořád kolmej. Transformace co nejsou **Rigid body transform** (rigidbody transform je translace a rotace), takže zkosení a škálování, nemůžu na normálové vektor použít, protože by ho rozhodily. Řešením je, vyrobit si speciální matici pro transformaci normálového vektoru:

Normal vector transformation

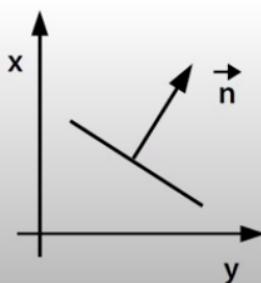
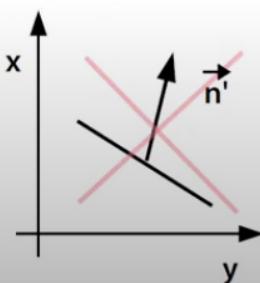
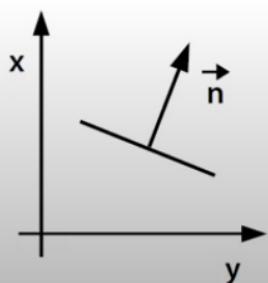


Normal vectors must not be transformed by regular matrices
(like point positions are)

- exception: M is rotational (orthonormal)

Normal-vector transformation matrix N :

$$N = (M^{-1})^T$$



Kde M je původní transformační matice kterou sme transofrmovali objekt. Prostě si seženu inverzi, a tu transponuji. Tím získám transformační matici pro Normálovej vektor, kterej tím můžu transformovat a zachová se mi kolmost. Důkaz jak se dá tohle [odvodit je tady](#). (Bacha, používá sloupce a násobení zleva)

Co se transformací týče, pro 3D je to všechno podobný:

Posunutí

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

Zkosení

$$Sh(a, b, c, d, e, f) = \begin{bmatrix} 1 & a & b & 0 \\ c & 1 & d & 0 \\ e & f & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

U otáčení si vyberu kolem které osy se točím, a ty vynuluju s 1 na diagonále, zbytek doplňím normálně 2D rotační maticí:

Otočení
kolem osy y

$$R_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Otočení
kolem osy z

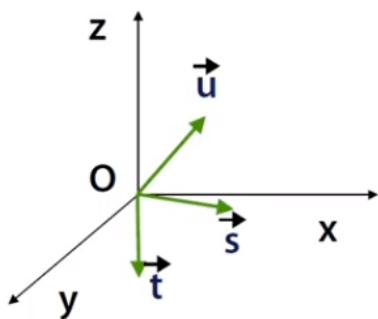
$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Speciálním pojmem je **rigid-body transform**. Jedná se o transformace, které zachovávají tvar, a mění jenom rotaci a směr. Taky se za ně považuje konverze mezi souřadnicovýma systémama.

Další usefull postup je konverze mezi souřadnicovýma systémama. Je na to několik postupů, například stačí složením Translace a několika rotací posunout střed jednoho systému na ten druhý, a pak ho několika rotacemi odrotovat tak, aby byly totožné. Co je ale jednodušší, je tohle:

Conversion between two orientations



Coordinate system has an origin **O** and is defined by three unit vectors **[s, t, u]**

$$M_{stu \rightarrow xyz} = \begin{bmatrix} s_x & s_y & s_z \\ t_x & t_y & t_z \\ u_x & u_y & u_z \end{bmatrix}$$


$$[1, 0, 0] \cdot M_{stu \rightarrow xyz} = s$$

$$[0, 1, 0] \cdot M_{stu \rightarrow xyz} = t$$

$$[0, 0, 1] \cdot M_{stu \rightarrow xyz} = u$$

$$M_{xyz \rightarrow stu} = M_{stu \rightarrow xyz}^T$$

Mám teda world souřadnicové systém X,Y,Z, začínající v bodě [0,0,0].

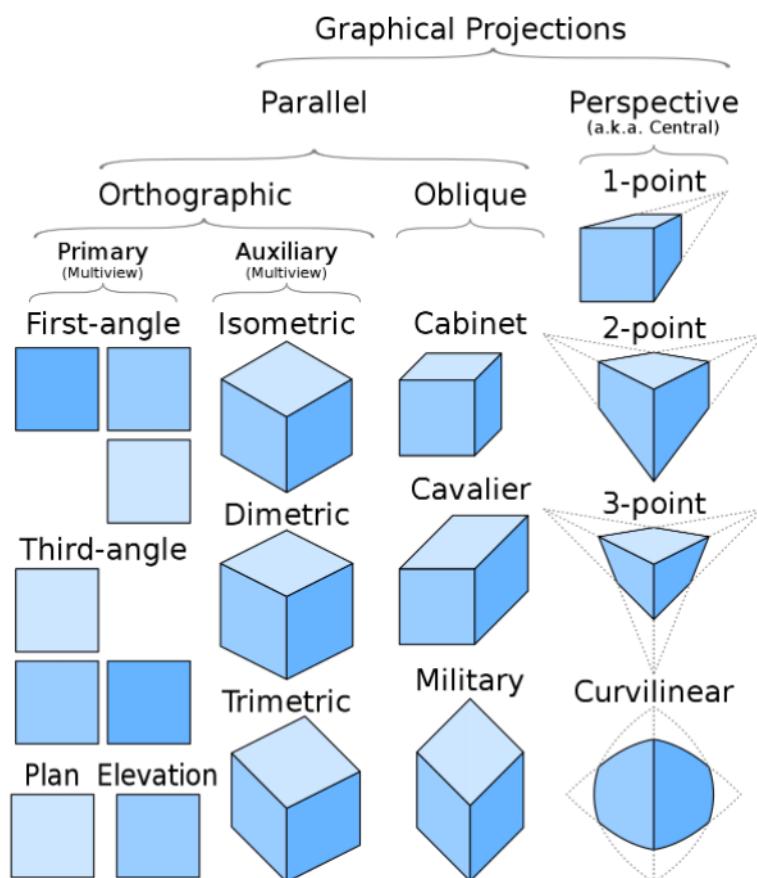
Pak mám druhý systém, který má počátek v bodě O, a osy daný jednotkovými vektorama [s,t,u]. Abych vytvořil transformační matici, která jako vstup dostane polohu bodu v systému S,T,U, a převede ho na polohu v systému X,Y,Z, musím provést následující:

Nejdřív si bod posunu tak, aby seděli počátky, tj translace o -O. Tím se dostanu do situace, co je popsána na slajdu vejš. Potom potřebuju udělat matici, která bod "otočí". Tu udělám tak, že vezmu hodnoty **jednotkových** vektorů S,T,U (po tom co sou posunutý do počátku), a udělám z nich řádky transformační matice (slad vpravo dole). Funguje to proto, že vím, že pokud mám vektor co má 1 v jedný z os (vlevo dole), a vynásobím ho tou maticí, tak tím přeče dostanu hodnotu jednoho z těch vektorů z S, T, U. (Protože sou jednotkový. Takže [1,0,0] v S,T,U bude [s,0,0] v X,Y,Z. A na základě toho je postavená ta matice vpravo).

([Video](#) to asi popíše líp)

Projekční maticy

O týhle matici všude jenom mluví, a nikde jí nechtějí odvodit. K čemu přesně je? Používá se k přepočtu souřadnic do projekce, většinou o dimenzi nižší. Nejčastěji teda z 3D do 2D - kam na monitor se vykreslí který objekt. Projekce jsou různý, podle toho, co zachovávají za vlastnosti:



Rovnoběžné projekce

- promítací paprsky jsou navzájem rovnoběžné

Kolmé projekce

- promítací paprsky jsou kolmé na průmětnu
- Mongeova projekce, půdorys, nárys, bokorys
- axonometrie (obecná kolmá projekce)

Kosoúhlé projekce

- kabinetní projekce (zkrácení měřítka osy z na 1/2)
- kavalírní projekce (stejné měřítko na všech osách)

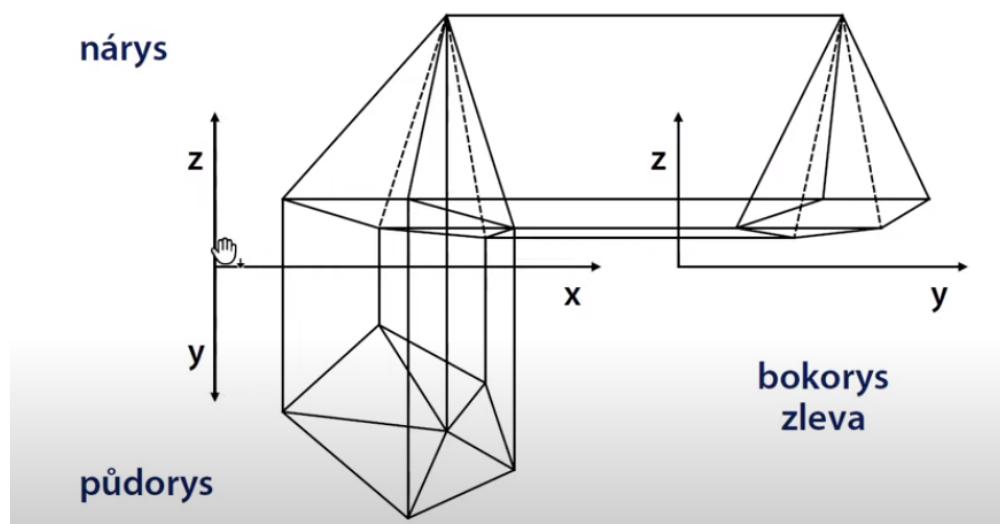


V podstatě popisují co zachovávám, a jak posílám paprsky. Ten slajd je trochu matoucí, má bejt takhle:

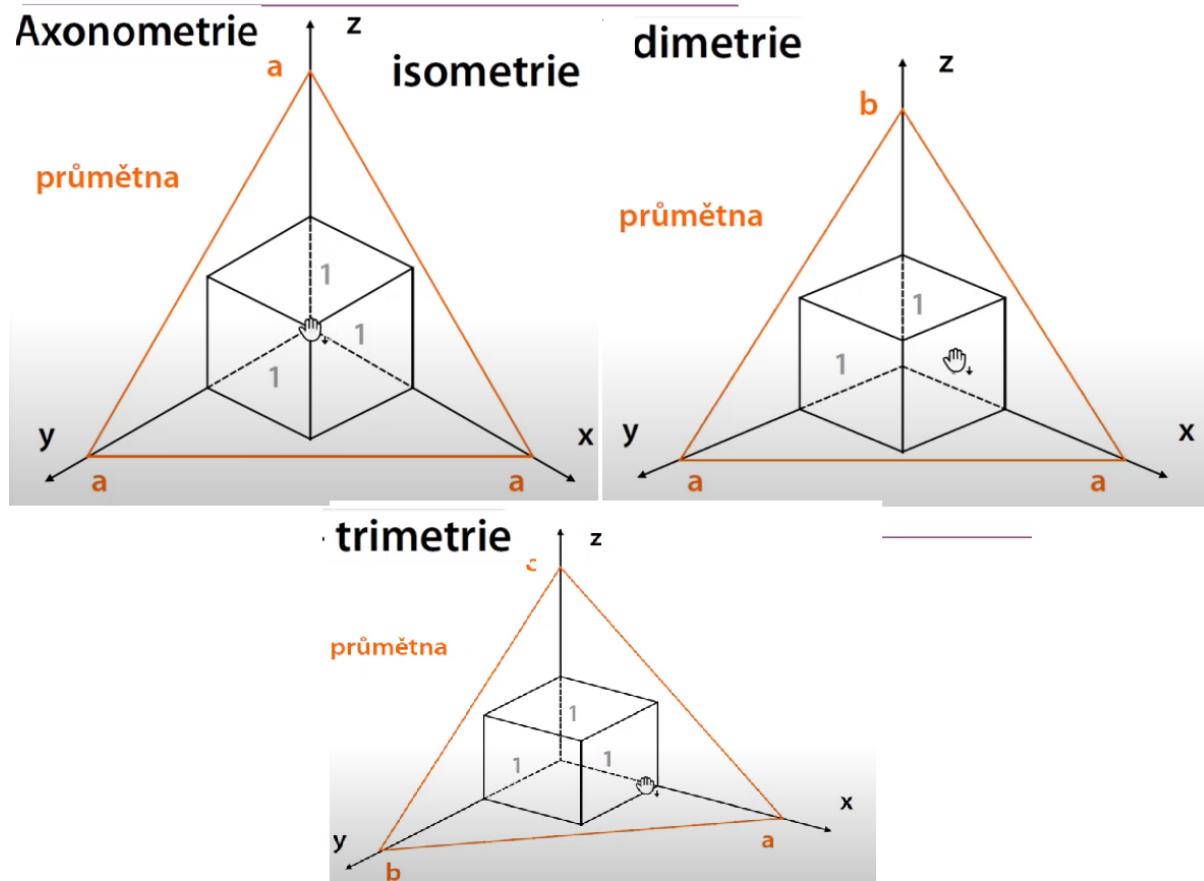
- Rovnoběžný projekce - paprsky jsou rovnoběžný navzájem
 - Rovnoběžné kolmé - jsou rovnoběžný, a zároveň kolmý na průmětnu.
 - Rovnoběžné kosoúhlé - paprsky jsou rovnoběžný, ale už né nutně kolmý. Tím dojde ke zkrácení měřítek některých os, protože se promítnout jako menší.

Rovnoběžné kolmé:

Mongeova projekce



Uplně klasická, prostě mám průmětnu kolmo na jednu ze tří os, podle toho na kterou tak ten typ to je.



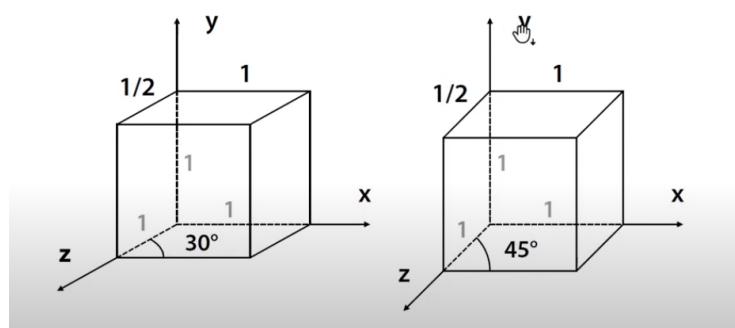
Axonometrie - obecné kolmé projekce. Mám určenou průmětnu tím, že se určí její vzdálenost na jednotlivých osách (tj body a, b, c). Rovina, která vede těma třema bodama, je průmětna, na kterou pak promítám kolmý rovnoběžný paprsky. Podle toho jak jsou body a, b, c , různý mám různý projekce: isometrie, všechny stejný. Dimetrie, dvě stejný a trimetrie je uplně obecná, kde a, b a c jsou různý. Dobře se v tom odečítají vzdálenosti.

Rovnoběžné kosoúhlé:

Kabinetní projekce



průmětna = xy

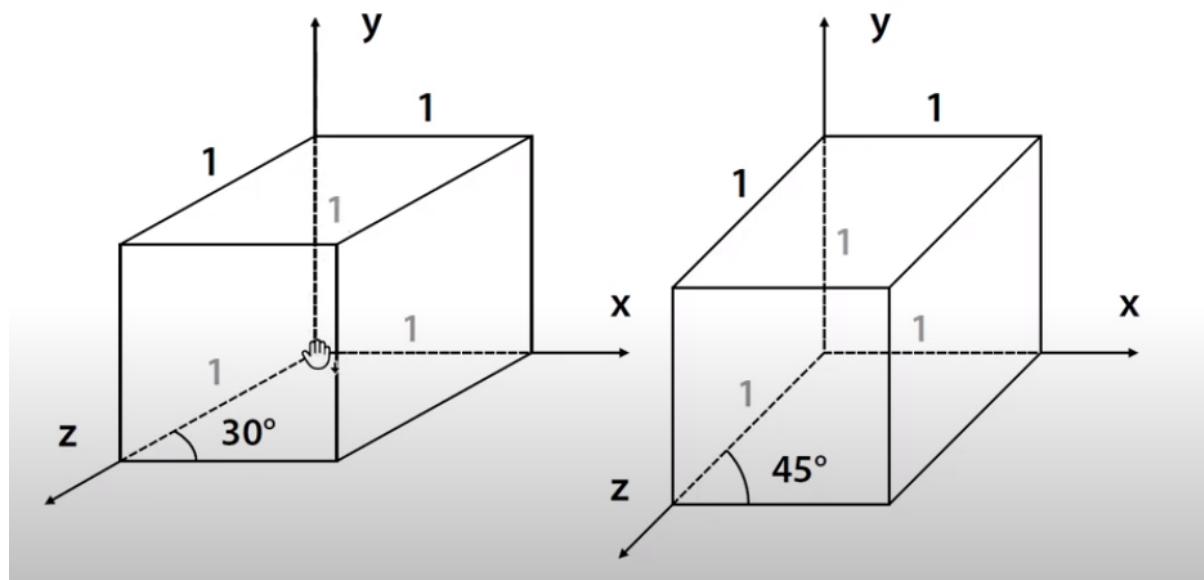


Průmětna je xy, ale nepromítám kolmě - zkrátím měřítko osy Z o polovinu, a promítnu jí pod určeným úhlem (většinou 30 nebo 45 stupňů)

Kavalírní projekce



průmětna = xy



To samý jako kabinetní, jenom nesnižuju měřítko osy Z.

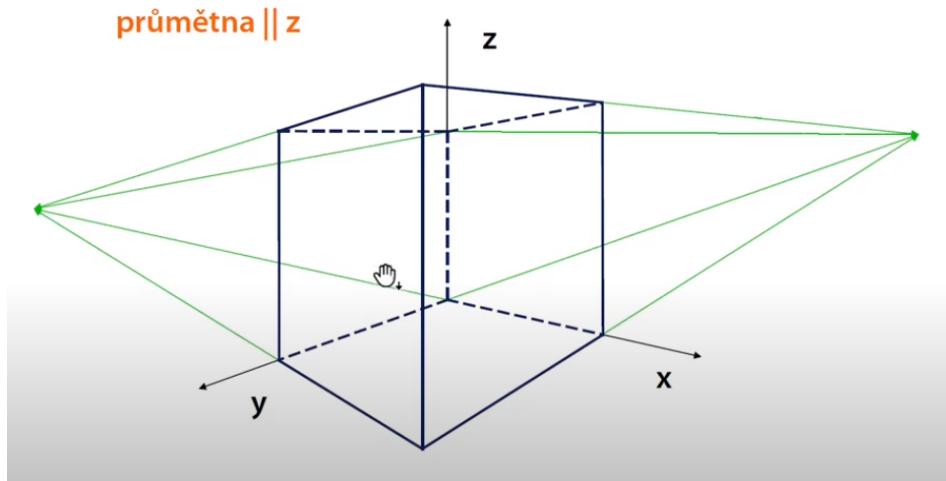
Perspektivní (středové) projekce:

Ty už nezachovávají rovnoběžky.

Promítací paprsky tvoří svazek, co prochází jedním bodem (středem projekce).

Dělí se na 1, 2 a 3 bodovou, podle toho, kolik os má střed projekce:

Dvoubodová perspektiva

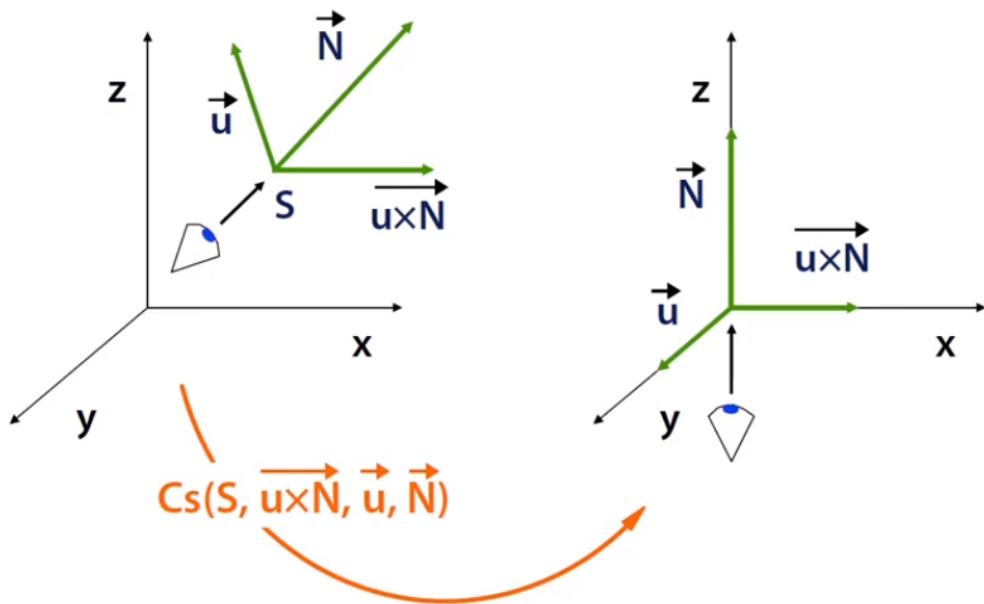


Bod určuje směr, kterým směřují rovnoběžky v daný ose. Ve dvoubodový je průmětna kolmá na třetí osu (tudíž se zachovají a nepotkají se). Ve tříbodový není kolmá ani na jednu, a v jednobodový je kolmá na dvě osty, a bod projekce je jenom jeden.

Projektivní transformace:

Obecná kolmá:

Obecná kolmá projekce



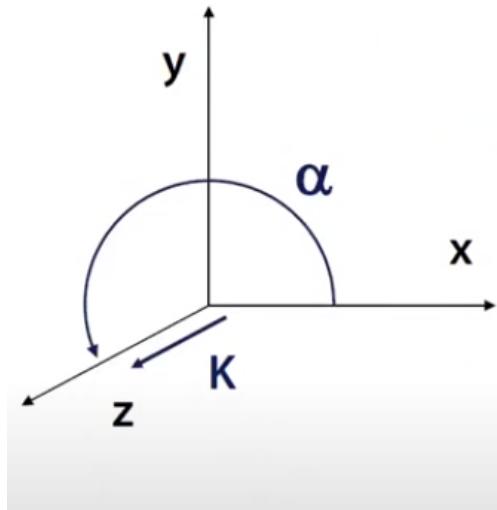
Máme zadáno následující:

- S - pozice pozorovatele - kde se nachází 0,0 průmětny

- N - vektor, určující směr, kterým se divá pozorovatel (tj osa Z)
- u - vektor, určující kde je naoře (up vector, vlastně osa y)
- uxN - right vektor, vektor kolmý na N a u. Určuje osu x.

Pro převedení souřadnic nejdřív provedu translaci do bodu [0,0,0] (tj, posun o $-Sx, -Sy, -Sz$), a pak si vytvořím matici, která převede souřadnice z x,y,z do uxN, u, N. Pak prostě vezmu $[x,y]$ bodů, a mám kolmou projekci.

Kosoúhlá projekce:



průmětna: xy
koeficient zkrácení: K
úhel průmětu osy z: α

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ K \cdot \cos \alpha & K \cdot \sin \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X a Y nechám bejt, jenom zkrátím osu z podle úhlu co očekávám.

Středová projekce:

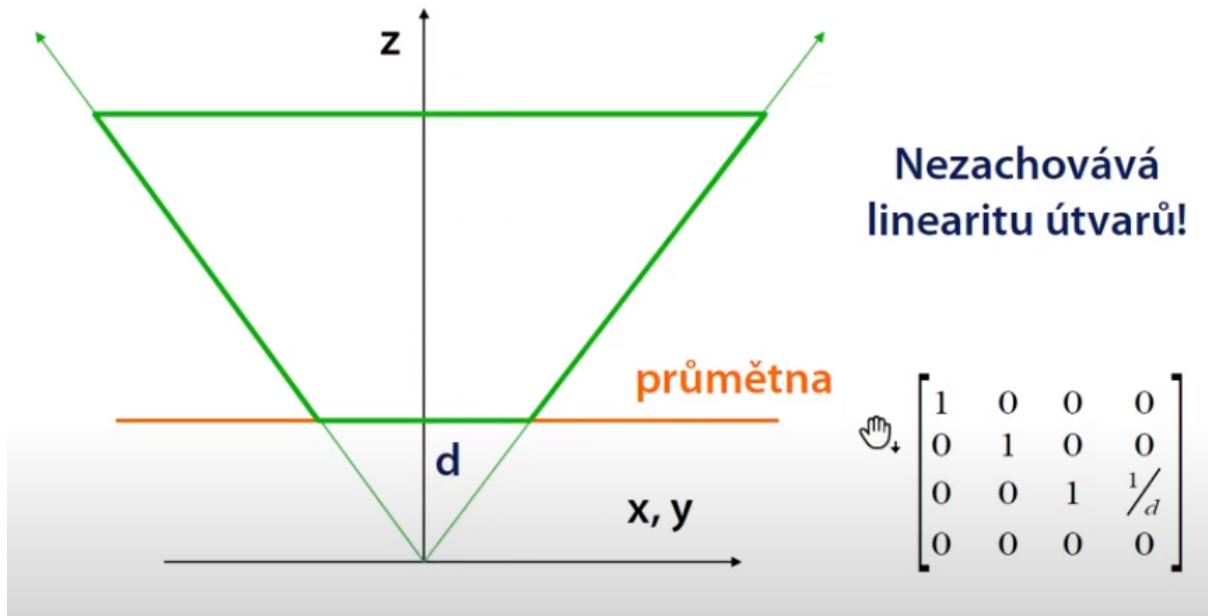
Obecná středová projekce

- střed projekce: S
- směr pohledu (normálový vektor průmětny): \vec{N}
- vzdálenost průmětny od středu projekce: d
- svislý vektor: \vec{u}

Promítací transformace

- převedení do **základní polohy** (střed projekce do počátku, směr pohledu do osy z): $C_s(S, \vec{u}, \vec{xN}, \vec{u}, \vec{N})$
- **perspektivní projekce**: např. $[x \cdot d/z, y \cdot d/z, z]$

Postup je podobný - nejdřív přesunu střed, tj převedu si souřadnice z báze X,Y,Z do báze uXN, u, N (viz konverze mezi souřadnicovýma systémama). Potom je potřeba zavést perspektivu, kde mi pomohou vlastnosti homogenních souřadnic:



Projekční matice OpenGL

V praxi se ale používá trošku jiná:

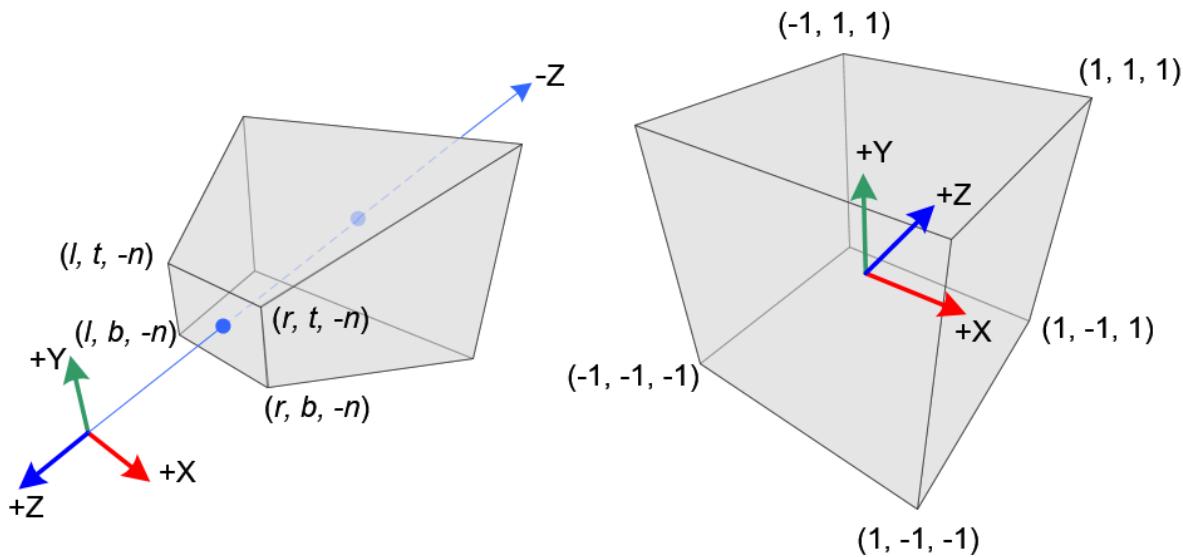
$$\left[\begin{array}{cccc} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & \frac{f+n}{f-n} & 1 \\ 0 & 0 & -\frac{2fn}{f-n} & 0 \end{array} \right] \quad \begin{array}{c} xy \\ n \\ [-]z \\ f \end{array}$$

Proměnný sou taky jiný než nahoře:

- n - Near, vzdálenost průmětny od středu projekce (předtím d). Taky vzdálenos co nejblíž vykreslju.
- r a l - right a left, vzdálenost od bodu [0,0,n] ke kraji projekce
- t a b - top a bottom, vzdálenost od 0,0,n ke kraji
 - Tj průmětna/okno do kterého promítám má velikost [r+l,t+b]
- f - Far - vzdálenost, kterou nejdál vykreslju

Cílem projekce je nejenom převést souřadnice mezi souřadnicovýma systémama (jako v projekci předtím), ale taky přeskálovat tak, aby dostal kostku jasně omezující co chci

vykreslovat. Chci převést všechny body co jsou v tom frustru (žlutá na obrázku) tak, aby byly v rozmezí souřadnic $[-1, 1]$ pro všechny osy. Jde mi teda o tenhle převod:



Toho dosáhnu přes podobnost trojuhelníků a škálování. Výsledná matice je tahle:

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & \frac{f+n}{f-n} & 1 \\ 0 & 0 & -\frac{2fn}{f-n} & 0 \end{bmatrix}$$

a dá se odvodit tímhle způsobem. Je to docela dlouhý, a relativně intuitivní, jenom bacha, ve článku dál mají opačnou notaci - násobí zprava sloupcovým vektorem, místo zleva řádkem jako my, tj výsledek bude transponovaný.
<https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix>

2.3. Kvaterniony

Kvaterniony jsou v podstatě 4D imaginární čísla, a používají se hlavně pro zápis rotace. Základní idea je následující:

Quaternions



Sir William Rowan **Hamilton**, 16 Oct 1843 (Dublin)

- $i^2 = j^2 = k^2 = ijk = -1$
- usage in graphics since 1985 (Shoemake)
- generalization of complex numbers in 4D space

$$\mathbf{q} = (\mathbf{v}, w) = i \mathbf{x} + j \mathbf{y} + k \mathbf{z} + w = \mathbf{v} + w \quad \text{sometimes } (\mathbf{w}, \mathbf{v})!$$

Imaginary part $\mathbf{v} = (x, y, z) = i \mathbf{x} + j \mathbf{y} + k \mathbf{z}$

$$i^2 = j^2 = k^2 = -1, \quad jk = -kj = i, \quad ki = -ik = j, \quad ij = -ji = k$$

V imaginárních platí, že $i^2 = -1$. Tohle rozšiřuje ještě o další 2 písmenka, pro který platí že $i^*j^*k = -1$, a každý 2 je -1 . Nejčastěji se zapisujou jako $\mathbf{q} = (\mathbf{v}, w)$, kde \mathbf{v} je vektor imaginárních částí, a w je reálný číslo, tj hodnota.

Taky platí $i^*j = k$, a $j^*i = -k$. Násobení teda není komutativní.

Operace s kvaternionama:

Součet je easy, po složkách. Násobení je bullshit:

Addition

$$- (\mathbf{v}_1, w_1) + (\mathbf{v}_2, w_2) = (\mathbf{v}_1 + \mathbf{v}_2, w_1 + w_2)$$

Multiplication

$$- \mathbf{q} \mathbf{r} = (\mathbf{v}_q \times \mathbf{v}_r + w_q \mathbf{v}_r + w_r \mathbf{v}_q, w_q w_r - \mathbf{v}_q \cdot \mathbf{v}_r)$$

$$i(q_y r_z - q_z r_y + r_w q_x + q_w r_x),$$

$$j(q_z r_x - q_x r_z + r_w q_y + q_w r_y),$$

$$k(q_x r_y - q_y r_x + r_w q_z + q_w r_z),$$

$$q_w r_w - q_x r_x - q_y r_y - q_z r_z$$



$$(\mathbf{q} \mathbf{q}')_{xyz} = \mathbf{q}_{xyz} \times \mathbf{q}'_{xyz} + \mathbf{q}_w \mathbf{q}'_{xyz} + \mathbf{q}'_w \mathbf{q}_{xyz} \quad [2.5]$$

$$(\mathbf{q} \mathbf{q}')_w = \mathbf{q}_w \mathbf{q}'_w - (\mathbf{q}_{xyz} \cdot \mathbf{q}'_{xyz}).$$

To druhý se asi líp pamatuje. Jak se k tomu dá dojít? Je to vlastně relativně easy dovození:

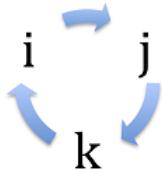
$$\mathbf{q} \mathbf{q}' = (\mathbf{q}_w + \mathbf{q}_x i + \mathbf{q}_y j + \mathbf{q}_z k)(\mathbf{q}'_w + \mathbf{q}'_x i + \mathbf{q}'_y j + \mathbf{q}'_z k).$$

Prostě si to rozepíšu, vynásobím, a potom zkrátím podle pravidel $i^*j^*k = -1$ a dalších vlastností, který ty čísla i, j, k mezi sebou mají, což se dobře pamatuje podle směru hodinových ručiček:

Multiplying two quaternions then involves using the relations $i^2 = j^2 = k^2 = -1$ and

$$\begin{aligned} ij &= k, \quad jk = i, \quad ki = j, \\ ji &= -k, \quad kj = -i, \quad ik = -j. \end{aligned}$$

This is typically abbreviated with a cycle mnemonic:



Conjugation

- $(v, w)^* = (-v, w)$

Norm (squared absolute value)

- $\|q\|^2 = n(q) = q q^* = x^2 + y^2 + z^2 + w^2$

Unit

- $i = (0, 1)$

Reciprocal

- $q^{-1} = q^* / n(q)$



Multiplication by a scalar

- $s q = (0, s) (v, w) = (s v, s w)$

Jednotkový quaternion je ten, pro kterej platí, že $n(q)$ (délka, absolutní hodnota na druhou), je rovna 1. Tj platí $x^2+y^2+z^2+w^2 = 1$.

Pokud mám jednotkové quaternion, dá se zapsat vždycky jako $q=(uq * \sin \phi, \cos \phi)$.

Tj, existuje úhel ϕ , a vektor z,y,z zapíšu jako jednotkové vektor $u * \sin \phi$, s reálnou $w=\cos \phi$.

- $q = (u_q \sin \phi, \cos \phi)$



- for some unit 3D vector u_q

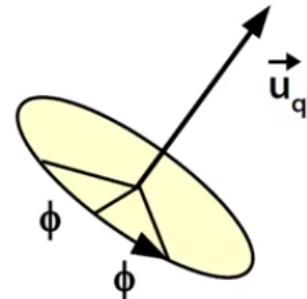
Reprezentujeme tím rotaci o úhel ϕ kolem osy U_q .

Power, exponential, logarithm

- $\mathbf{q} = \mathbf{u}_q \sin \phi + \cos \phi = \exp(\phi \mathbf{u}_q)$, $\log \mathbf{q} = \phi \mathbf{u}_q$
- $\mathbf{q}^t = (\mathbf{u}_q \sin \phi + \cos \phi)^t = \exp(t\phi \mathbf{u}_q) = \mathbf{u}_q \sin t\phi + \cos t\phi$

Unit quaternion

- $\mathbf{q} = (\mathbf{u}_q \sin \phi, \cos \phi)$
- \mathbf{u}_q ... axis of rotation, ϕ ... angle



Vector (point) in 3D: $\mathbf{p} = [p_x, p_y, p_z, 0]$

Rotation of vector (point) \mathbf{p} around \mathbf{u}_q by angle 2ϕ

$$\mathbf{p}' = \mathbf{q} \mathbf{p} \mathbf{q}^{-1} = \mathbf{q} \mathbf{p} \mathbf{q}^*$$

K čemu to hlavně je - dá se tím naprosto v pohodě interpolovat mezi dvouma rotacemi:

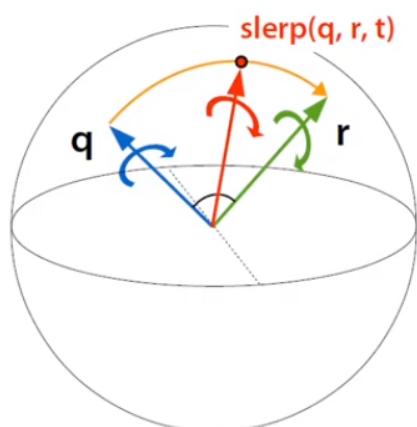
Spherical linear interpolation (slerp)



Two quaternions \mathbf{q} and \mathbf{r} ($\mathbf{q} \cdot \mathbf{r} \geq 0$, else take $-\mathbf{q}$)

Real parameter $0 \leq t \leq 1$

Interpolated quaternion $\text{slerp}(\mathbf{q}, \mathbf{r}, t) = \mathbf{q} (\mathbf{q}^* \mathbf{r})^t$



$$\text{slerp}(\mathbf{q}, \mathbf{r}, t) = \frac{\sin(\phi(1-t))}{\sin \phi} \cdot \mathbf{q} + \frac{\sin(\phi t)}{\sin \phi} \cdot \mathbf{r}$$

$$\cos \phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$$

The shortest spherical arc
between \mathbf{q} and \mathbf{r}
(quaternion splines will be explained later)

Kde q a r jsou kvaterniony, a t je čas mezi 0,1, a určuje jak daleko na cestě sem (tj, v 0 to bude na začátku, q , v 1 bude na konci v r , a 0.5 bude přesně uprostřed.)

Vzoreček je relativně easy, umocnění kvaternionu na t se dělá tak, že jenom vynásobím úhel tčkem - $q^*t = (v^*\sin t^*F_i, \cos t^*F_i)$, q^* se dělá tak že prohodím znaménko u $(v,w)^* = (-v,w)$.

Tj $\text{slerp}(q,r,t) = q (q^*r)^t = (qv, qw) ((-qv, qw) (rv, rw))^t$ atd.

Pokud potřebuju rotovat jenom mezi dvouma vektorama (tj, nezajímá mě rotace objektu okolo vlastní osy), můžu použít tohle:

Rotation between two vectors



Two vectors s and t

1. normalization of s , t

$$2. \text{ unit rotation axis} \quad \mathbf{u} = (s \times t) / \|s \times t\|$$

$$3. \text{ angle between } s \text{ and } t \quad e = s \cdot t = \cos 2\phi$$

$$\|s \times t\| = \sin 2\phi$$

$$4. \text{ final quaternion} \quad \mathbf{q} = (\mathbf{u} \cdot \sin \phi, \cos \phi)$$

$$q = (q_v, q_w) = \left(\frac{1}{\sqrt{2(1+e)}} (s \times t), \frac{\sqrt{2(1+e)}}{2} \right)$$

Rozdíl oproti předchozímu vzorci je to, že na začátku a na konci mám jenom směr, a nezajímá mě už ta rotace.

Nevýhody kvaternionů

Jsou hrozně náročný pro transformaci bodů/vektorů, v porovnání s rotační maticí. Posunutí bodu kvaternionem obsahuje asi 16 operací násobení, kdežto rotační matice má většinou HW support v GPU.

Matice se zase ale nedaj rozumě interpolovat, takže pro slerpy jsou lepší kvaterniony. Další nevýhodou rotací podle eulerovských úhlů (tj, podle 3 os) je gimbal lock, což je situace, která může nastat v momentě, kdy se ti dvě osy otočí tak, že jsou totožné - a tím se ztratí možnost otáčet se v jednom směru.

Obrázek to ukáže nejlíp -

https://upload.wikimedia.org/wikipedia/commons/4/49/Gimbal_Lock_Plane.gif

Tj, když vymýslím jak přesně budu interpolovat, tak použiju kvaternion, nakonec z něj ale stejně udělám rotační matici, kterou pak masivně posunu ty miliony bodů.

2.4. Spline funkce

Jedná se o funkce, které používám v momentě, kdy chci vykreslovat nějakou spojitou křivku, na základě několika určených kontrolních bodů a parametrů. Mám dva typy:

- Aproximaci - výsledná křivka bodama nemusí procházet
- Interpolaci - křivka bodama prochází

Parametrický křivky - jedná se o lepší způsob zápisu křivek. Normálně sme zvyklý zapisovat křivky jako graf funkce, tj $y = f(x)$. Tím, že zadám x , dostanu pro něj y . To má ale dvě nevýhody - nemůžu nakreslit křivku co má více stejných souřadnic y (tj třeba kruh), a neumím spočítat tangent v případě, že křivka jde dolů přímo. Řeší se to použitím parametrických křivek:

Křivku si definuju jako vektorovou funkci $f(t)$, která vrátí vektor souřadnic křivky v čase t , tj $f(t) = [x(t), y(t)]$. Například teda pro kruh by to bylo $f(t) = [\sin(t), \cos(t)]$.

Polynomiální křivky se většinou pak zapisujou takhle: $f(t) = a_0 + a_1*t + a_2*t^2 + \dots + a_n * t^n$, kde a_i je vektor koeficientů. Tj třeba funkce

$$x(t) = 6t - 9t^2 + 4t^3,$$

$$y(t) = 4t^3 - 3t^2.$$

se může zapsat jako

$$f(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} 6 \\ 0 \end{pmatrix} t + \begin{pmatrix} -9 \\ -3 \end{pmatrix} t^2 + \begin{pmatrix} 4 \\ 0 \end{pmatrix} t^3.$$

Taky mě u nich zajímají dvě různé vlastnosti spojitosti:

Geometrická spojitost - jestli ta křivka vizuálně je spojitá, a jestli jsou její tečny spojité. Příše se jako G^n , kde n určuje řád ve kterém mě spojitost zajímá. G^0 je prostě spojitost, G^1 je tangent - tj jestli tečny jsou kolinearne - tady pozor, nejde o rovnost, ale kolinearitu, tj platí:

Nta derivace $P(x)$ v bodě $x_0 = k * Nta$ derivace $Q(x)$ v bodě x_0 .

Tj prostě mají stejný směr, ale už nemusí mít stejnou velikost.

Analytická/Parametrická spojitost - zajímá mě i jak by se po křivce pohyboval bod. Jestli je rychlosť spojita, tj se skokově nemění, to samý u akcelerace atd. Tam požaduju aby nta derivace byla spojita, a zároveň se rovnala, tj požaduju totální tečný vektory

Curve continuity

- G^n – geometric continuity of the n^{th} order (G^0 – simple continuity, G^1 – tangent, G^2 – curvature...)
- C^n – analytical continuity of the n^{th} order, n^{th} derivative continuity (C^1 – speed, C^2 – acceleration), superior to geometric continuity

Většinu teore vymysleli lidi v automobilkách, aby dal dohromady hezký karoseire aut :D Beziér je z Renaultu, de Castelau z Citroenu, Ferguson pro Boing, de Boor pro General Motors...

Pointa je teda taková, že mám zadanejch několik kontrolních bodů, a program mi jima kreslí křivku tak, aby splňovala parametry co chci. Většinou se kromě bodů určuje i tangent v bodě, a chceme jistou lokalitu - změna v kontrolním bodu změní jenom jeho bezprostrední okolí.

Většinou jsou zadávány takhle:

Parametric expression ($0 \leq t \leq 1$)

$$P(t) = \sum_{i=0}^{N-1} w_i(t) P_i$$

Kde t je "čas", kterej bod křivky to je. Tj 0 je na začátku, 1 na konci křivky. Wi je váha bodu Pi v daném bodě, a Pi je kontrolní bod. Tohle je obecná definice free-form křivek, spliny mají níž konkrétnější.

Většinou požadujeme, aby platila **Cauchyho** podmínka - pokud otočím křivku nějakou transformací bod po bodu, tak dostanu stejný výsledek jako když otočím kontrolní body a křivku znova spočítám. Platí pokud

$$\sum_{i=0}^{N-1} w_i(t) = 1$$

A teda zásadní jsou takový kachničky (ducks anglicky) co sou mega těžký, a používají se k ukotvení pravítka s názvem Spline, který se používá na kreslení křivek co prochází body co chci:



Definice spline funkcí je teda tohle:

Named after elastic ruler used in ship design (pinned in several points by "ducks")

Definition: **spline function of degree n**

- piece-wise **polynomial** (of degree **n**)
- **maximum-smoothness connection:**
 C^{n-1} – continuity of **n-1th** derivative (polynomial of degree **n**)
- **global parametrization** u , $u_0 \leq u \leq u_N$ $[u_0, u_1, \dots u_N]$
- individual parts are often uniformly parametrized – **uniform spline** $t_i = (u - u_i) / (u_{i+1} - u_i)$, $0 \leq t_i \leq 1$

Spline řádu **n** je funkce co je **po částech** polynomiální (tj jednotlivý části jsou různý polynomy, ale né stejný - pak by to byl prostě polynom, a né spline), a je maximálně monžně spojitá v bodech spojení.

Maximálně možně neznámená absolutně (tj, všechny derivace C^0 až C^n jsou spojité), protože pak by byly oba spolejší polynomy střené (a to nechceme, chceme různý), ale znamená to C^{n-1} spojitost

Když definujeme Spline, tak ho definujeme na intervalu $[a,b]$ (většinou 0-1), kterej rozdělíme na kpodintervalů $[t_0-t_1, t_1-t_2, t_2-t_3, \dots, t_{k-1}-t_k]$, kde $t_0 < t_1 < t_2 \dots < t_k$

A pro každý $[t_i, t_{i+1}]$ definujeme Polynom P_i , kterej ho určuje. Řád spline funkce je pak rád polynomu s největším řádem, můžu ale používat i menší, pokud dodržím dpomínky spojitosti $n-1$ derivace.

Spline je pak definovanej takhle. Prostě vezmu ten polynom do kterého intervalu to spadá, a kouknu na jeho hodnotu.

$S(t) = P_i(t)$, kde $t_i < t < t_{i+1}$. Tj takhle:

$$\begin{aligned}S(t) &= P_0(t), \quad t_0 \leq t < t_1, \\S(t) &= P_1(t), \quad t_1 \leq t < t_2, \\&\vdots \\S(t) &= P_{k-1}(t), \quad t_{k-1} \leq t \leq t_k.\end{aligned}$$

2.5. Interpolace kubickými spliny

Protože pracuju s polynomama (který se používají hlavně proto, že se dobře derivujou a počítaj), tak to ve výsledku znamená, že vlastně řeším tuhle soustavu rovnic (v případě kubickejch polynomů):

určen, ta jimi však procházet nemusí. Parametricky

$$\begin{aligned}x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x \\y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y \\z(t) &= a_z t^3 + b_z t^2 + c_z t + d_z\end{aligned}$$

zapisováno je v maticovém tvaru:

$$Q(t) = \mathbf{T}\mathbf{C} = [t^3 \ t^2 \ t \ 1] \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}.$$

me derivací vektoru \mathbf{T}

Což si můžu přepsat jako matici vejš. Zároveň protože většinou pracuju s nějakýma bodama, tak si je tam přihodím jako poslední sloupce vektor:

Matrix notation of a curve



$$\mathbf{P}(t) = \mathbf{T} \mathbf{C} = \mathbf{T} \mathbf{M} \mathbf{G}$$

- separation of a parameter vector (\mathbf{T}) from polynomial basis (\mathbf{M}) and geometric control conditions/points (\mathbf{G})
- differentiation (tangent, curvature) restricted to \mathbf{T}
- control polynomial $\mathbf{T}\mathbf{M}$ times “geometry” \mathbf{G}

Cubic: $n = 3$, $k = 4$

$$Q(t) = [t^3, t^2, t, 1] \cdot \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \cdot \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{bmatrix}$$

Matice m určuje bázový parametry křivky, což jsou konstanty které určujou jak se křivka bude chovat. Třeba když hodím $m_{41}=1$, $m_{31}=-1$ a $m_{32} = 1$ a zbytek dám na 0, tak po vynásobení dostanu: $Q(t) = (1-t)*G1 + t*G2$, což je rovnice přímky.

Nejjednodušší křivkou která jde dá takhle učlat je Fergusova kubika (resp, říká se jí taky Hermitovská křivka):

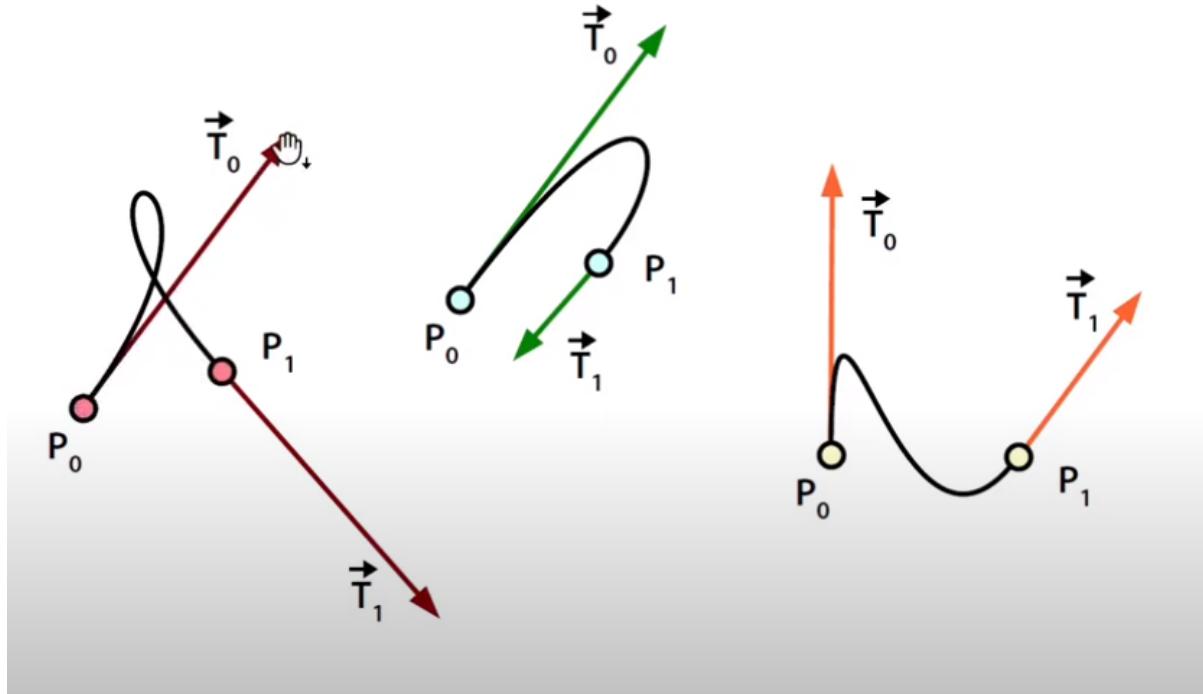
Ferguson curve (cubic)

Geometry: endpoints and tangent vectors

- beginning (\mathbf{P}_0) and end (\mathbf{P}_1) of a curve
- tangents in beginning (\mathbf{T}_0) and ending (\mathbf{T}_1) points

$$F(t) = [t^3, t^2, t, 1] \cdot \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ T_0 \\ T_1 \end{bmatrix}$$

Do pravé matice jsou první dva řádky body, a další dva vektory tečen v těch bodech. Vypadá nějak takhle:



Pokud je chci na sebe navazovat, tj mít dvě co nějak pokračujou, dejme tomu $Q(x)$ a $R(x)$, tak mi stačí dát : $RP0 = QP1$ a $RT0 = k * QT1$ - tj mají stejnej jeden bod a v něm tečnu stejným směrem. (ta je potřeba pro geometrickou spojitost)

Další oblíbený křivky vycházející z hermitovkých jsou **Kochanek-Bartels křivky** občas se jím říká TCB křivky, protože u nich definuju tři parametry pro danou bod - **tension**, **continuity** a **bias** (napětí, spojitost a šikmost).

Křivka je zadána posloupností bodů $P_0 \dots P_n$, s tím že začíná v bodě P_1 a končí v bodě P_{n-1} . První a poslední sou teda řídící. V každém z vnitřních bodů má koeficienty (ti, ci a bi). Defaultně jsou 0, a v tom případě je to **Catum-roll spline**.

Je potřeba si spočítat tangenty v bodě, ten se dělí na levoj a pravej (ve smyslu rostoucího parametru t):

Left and right tangent (T_0 and T_1 in local sense):

$$L_i = \frac{(1-t)(1-c)(1+b)}{2} \cdot (P_i - P_{i-1}) + \frac{(1-t)(1+c)(1-b)}{2} \cdot (P_{i+1} - P_i)$$

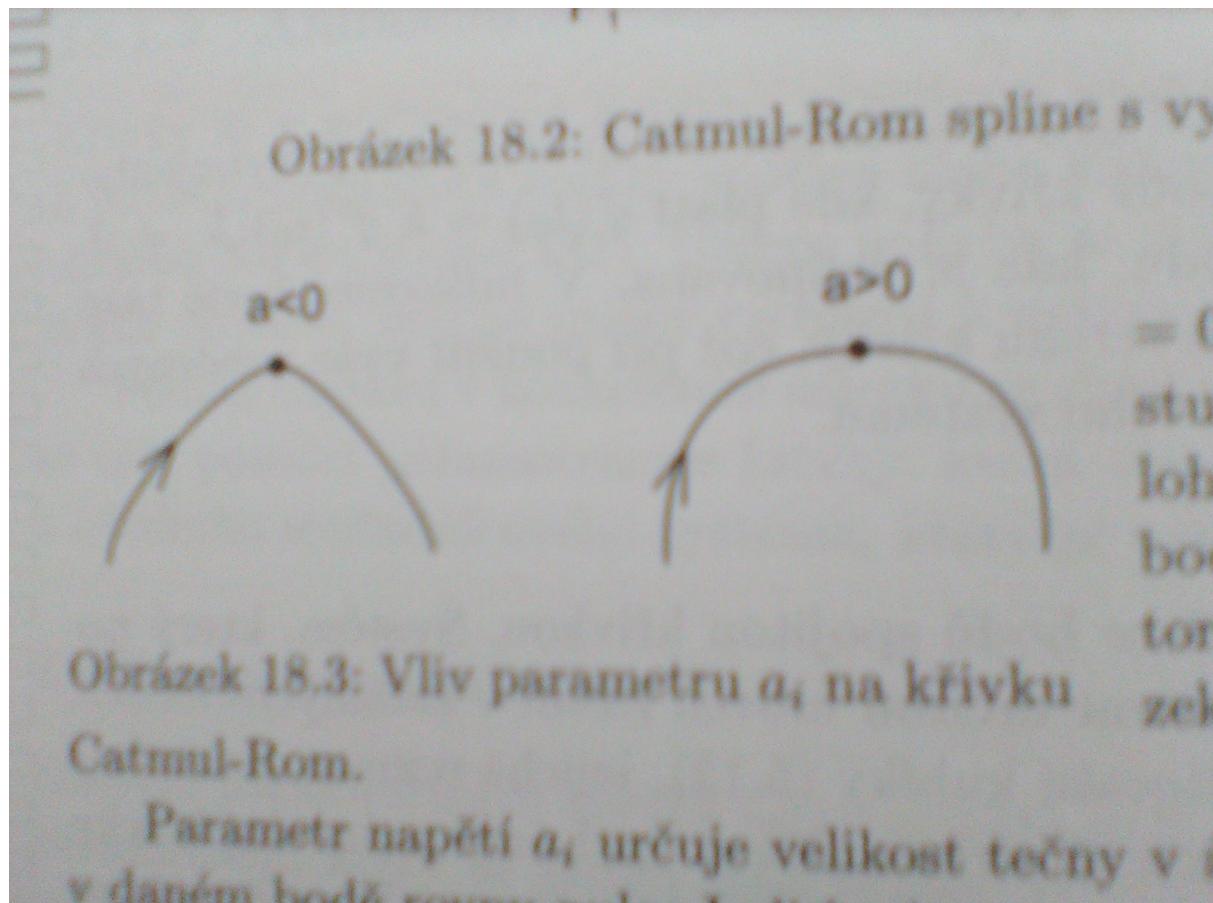
$$R_i = \frac{(1-t)(1+c)(1+b)}{2} \cdot (P_i - P_{i-1}) + \frac{(1-t)(1-c)(1-b)}{2} \cdot (P_{i+1} - P_i)$$

Parameter t_i ovlivňuje velikost tečny v bodě i - určuje jestli bude víc vlevo nebo víc vpravo.
Pokud si dosadím jako $t_i=0$ a spočítám rovnici pro jeden z tangentů, dostanu po zkrácení:

$$T_i = \frac{1}{2} \cdot (P_{i+1} - P_{i-1}) \quad (\text{což je taky catmull-roll splina}).$$

Pokud jsou parametry c a $b = 0$, tak parameter t_i určuje $T_i = (1-t)/2 * (P(i+1) - P(i-1))$

Projeví se takhle: (Na obrázku je $a=t$, knížka má jinak značení)



Obrázek 18.3: Vliv parametru a_i na křivku Catmull-Rom.

Parametr napětí a_i určuje velikost tečny v daném bodě rozdílu mezi T_i a T_{i+1} .

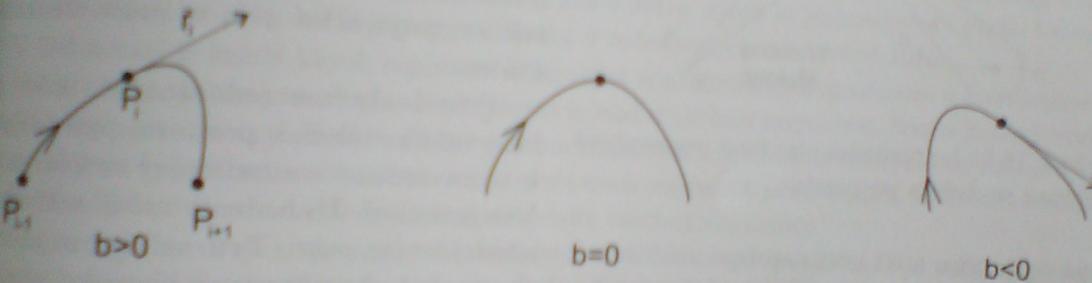
Parameter b určuje šikmost - směr a délku tečného vektoru. Pro $b_i > 0$ bude tečna blíž k levé části křivky, pro $b_i < 0$ víc k pravé části křivky.

Projeví se takhle:

a náhle změní svoji dráhu. Pro $c = -1$ je $\vec{l}_i = P_i - P_{i-1}$ a prvního řádu a je pouze C^0 . Pohybující se objekt způsobí změnu směru pohybu.

Jednou z nevýhod TCB křivek je, že předpokládají konstantní interval $t_{i+1} - t_i = 1$ mezi uzelky. Pro jinou změnu parametru $\Delta_i = t_{i+1} - t_i$ se upraví hodnoty vektorů \vec{l}_i a \vec{r}_i

$$\vec{l}'_i = \vec{l}_i \frac{2\Delta_i}{\Delta_{i-1} + \Delta_i} \quad \vec{r}'_i = \vec{r}_i \frac{2\Delta_{i-1}}{\Delta_{i-1} + \Delta_i}.$$



Obrázek 18.4: Vliv parametru b_i na křivku Kochanek–Bartels

2 Vysokoúrovňová počítačová animace

Vysokoúrovňové animace uvedeme stručný úvod do přímé a inverzní kinematiky. Algoritmy, které se používají pro detekci kolizí, jsou uvedeny v části 14.3.

Ještě vysokoúrovňové grafiky, o které budeme dále hovořit, patří do oblasti *kinematiky*. Kinematika

Parameter c_i určuje spojitost v bodě P_i . Pokud je $c_i = -1$, tak skončí levá tečna jako $P_i - P_{i-1}$ a pravá tečna $P_{i+1} - P_i$, což znamená že křivka není spojitá v C^1 , protože se prostě zlomí - tečna je rovná směru křivky a je jedno kam ten směr vede. Pro $c_i=1$ nastane podobná situace, akorát se prohodí levá a pravá tečna.

Cardinal spline:

cardinal spline

- parameter a only (in fact relates to "t", $c = b = 0$)

$$T_i = a \cdot (P_{i+1} - P_{i-1}) \quad 0 \leq a \leq 1$$

Tady nevím jestli není chyba - v knížce píšou, že t je v rozmezí -1 a 1 , a pokud to dosadíš do vzorku pro levou a pravou tečnu vejš v TCB křivce, tak ti výjde že to tak má být (pro $a=0$ výjde $\frac{1}{2}$). Třeba má ale jinak značení.

Ještě víc rigidní jsou velmi často používaný **Catmull-Rom spliny**, který sou definovaný jako cardinal spline s $a=\frac{1}{2}$. (Podle přednášek, podle knížky je to $a=0$)

Catmull-Rom spline $T_i = \frac{1}{2} \cdot (P_{i+1} - P_{i-1})$

– $a = t = 1/2$

$$MG = \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

Kubický spline - co se týče přímo kubicýho splinu, tak má takovouhle formu:

Interpolating cubic spline

- in knot points x_0, x_1, \dots, x_n function values y_0, y_1, \dots, y_n are prescribed

$$S(x) = S_k(x) = s_{k,0} + s_{k,1}(x-x_k) + s_{k,2}(x-x_k)^2 + s_{k,3}(x-x_k)^3$$

$$x \in [x_k, x_{k+1}], \quad k=0, 1, \dots, n-1$$

Condition A: $S(x_k) = y_k \quad k=0, 1, \dots, n$

$S(x)$ mi počítá hodnotu v bodě. $S_k(x)$ je kubickej polynom co odhaduje jednu z částí, konkrétně mezi bodama x_k a x_{k+1} . Mám několik podmínek A,B,C,D,E, který řeším jako soustavu rovnic, čímž získám jednoznačný konstanty/parametry $s_{k,i}$, kde $k=0\dots n$ a i jde $0\dots 3$. Z nichž nejzajímavější je podmínka A - která určuje, jakou hodnotu y_k má funkce mít v bodě X_k , což jsou ty knot pointy co na sebe navazujou polynomy.

Podmínky jsou:

Condition A: $S(x_k) = y_k \quad k=0, 1, \dots, n$

Condition B (C^0 continuity):

$$S_k(x_{k+1}) = S_{k+1}(x_{k+1}) \quad k=0, 1, \dots, n-2$$

Condition C (C^1 continuity):

$$S'_k(x_{k+1}) = S'_{k+1}(x_{k+1}) \quad k=0, 1, \dots, n-2$$

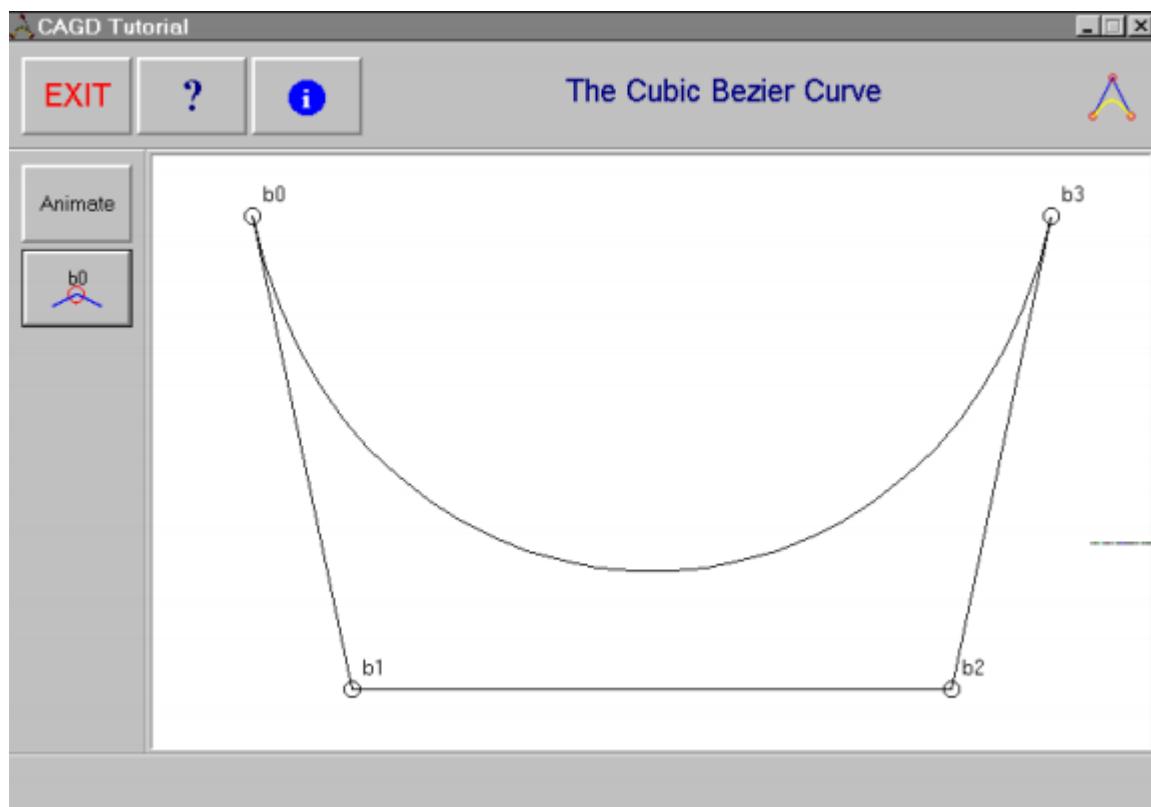
Condition D (C^2 continuity):

$$S''_k(x_{k+1}) = S''_{k+1}(x_{k+1}) \quad k=0, 1, \dots, n-2$$

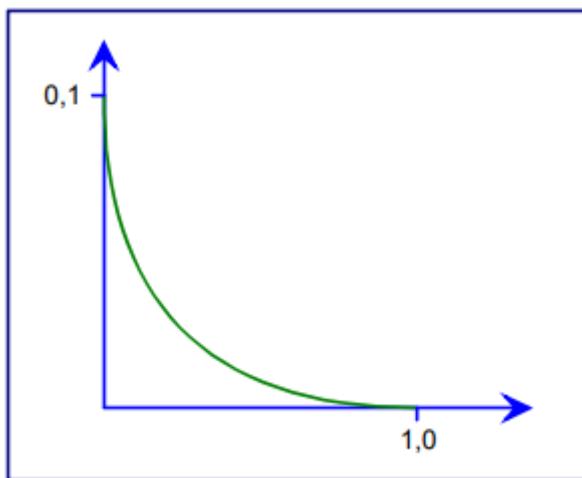
Natural cubic spline has an additional condition E:

$$S''(x_0) = S''(x_n) = 0$$

2.6. Bézierovy křivky



Když si vezmeme parabolu, co prochází bodama [1,0] a [0,1], a je v těch bodech kolmá na osy x a y, vypadá nějak takhle:



a její rovnice bude

$$\mathbf{f}(t) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} t^2 + \begin{pmatrix} -2 \\ 0 \end{pmatrix} t + \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Ta se ale dá přepsat takhle:

$$f(t) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}(1-t)^2 + \begin{pmatrix} 0 \\ 0 \end{pmatrix}2t(1-t) + \begin{pmatrix} 0 \\ 1 \end{pmatrix}t^2.$$

A takhle si to přepisuji proto, že když se podívám na vektory a, b, c, tak mám body [1,0],[0,0],[0,1], což je přesně trojice kontrolních bodů co sem používal - prochází body 01 a 10, a je v nich kolmá na polygon 01, 00, 10. A to je přesně to, co chceme od Beziérové křivky -

- Prochází krajiními kontrolními body
- Je v nich tečnou na kontrolní polygon z ostatních bodů.

Obecně zapsaná bude tedy kvadratická beziérova křivka vypadat takhle:

$$f(t) = b_0 (1-t)^2 + b_1 2t(1-t) + b_2 t^2.$$

Kde b0,b1 a b2 jsou kontrolní body.

Tímhle způsobem jde pracovat i s většíma stupněma polynomů. Pro obecný stupeň n beziérové křivky je vzoreček následující:

$$f(t) = \sum_{i=0}^n b_i B_i^n(t)$$

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$

B_n je Bernsteinova funkce:
kombinační číslo, b(i) je i-tej kontrolní bod (tj vektor).

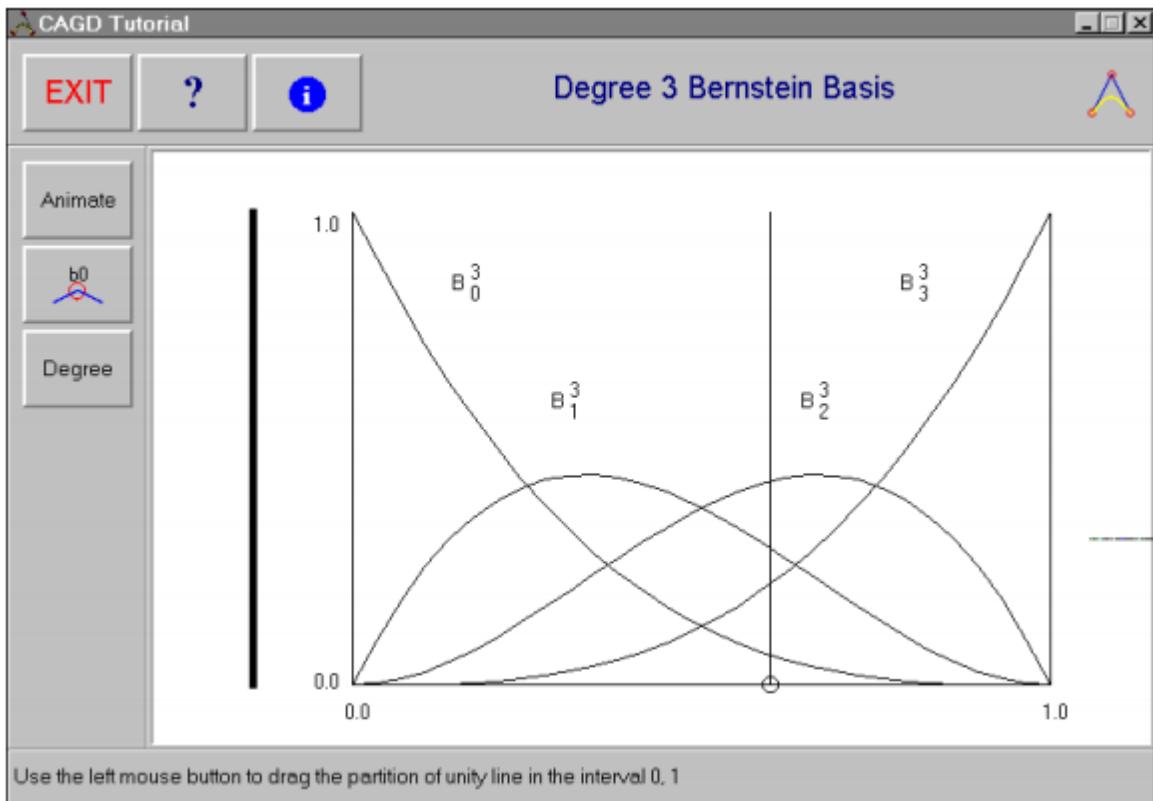
ve který (n nad i) je

Několik hlavních výhod Bernsteinový funkce je "partition of unity", tj když sečtu všechny polynomy z bernsteinové báze (pro řád n je bernsteinova báze všechny polynomy B(n,i) kde i = 0, 1, .. n), dostanu 1:

$$\sum_{i=0}^n B_i^n(t) = 1$$

- což se mi skvěle hodí do **Cauchihho** podmínky.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$



4 polynomy patřící do Bernsteinovy báze. Tím, že je sčítám dohromady z různějma koeficientama, sem schopnej aproximovat různý křivky, podobně, jako když sčítám sinusovky.

Její maticové zápis vypadá takhle:

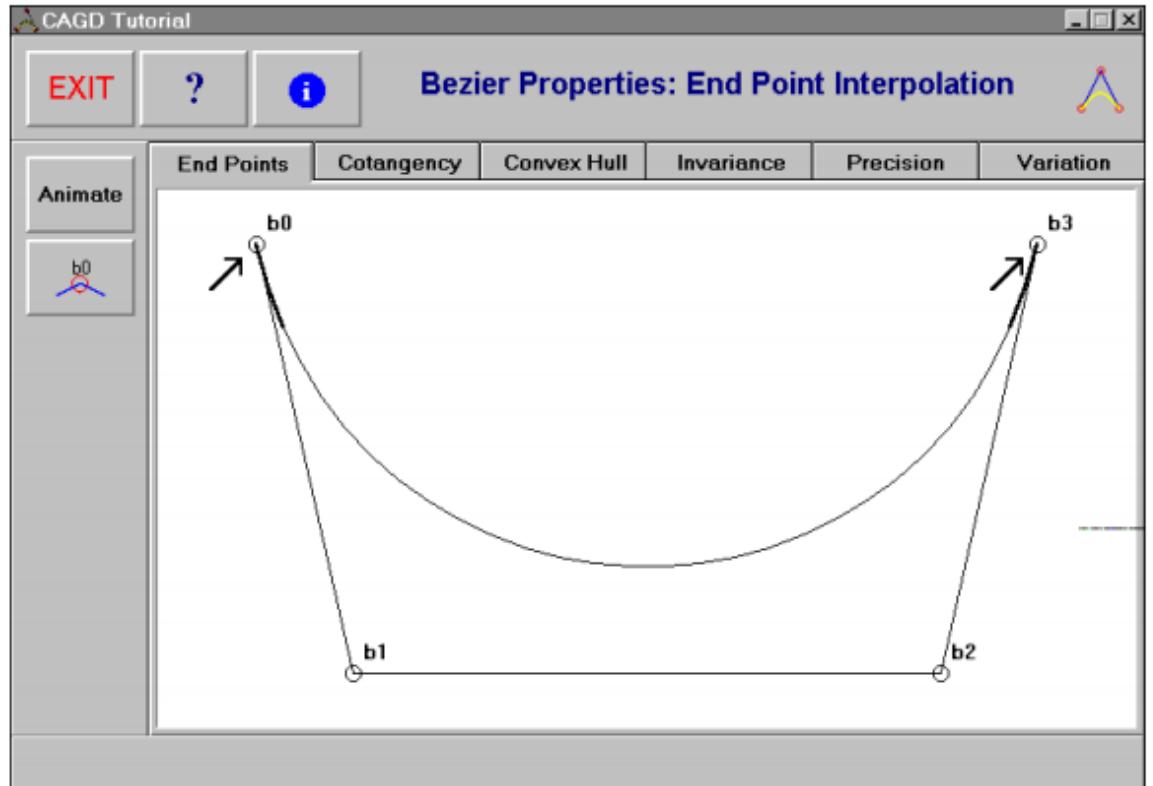
$$\mathbf{MG} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Vlastnosti beziérové křivky:

- Interpolace koncových bodů - Platí $f(0) = b(0)$ a $f(1) = b(n)$,

- Může za to bernsteinova funkce, protože všechny funkce výjdou na nulu, kromě

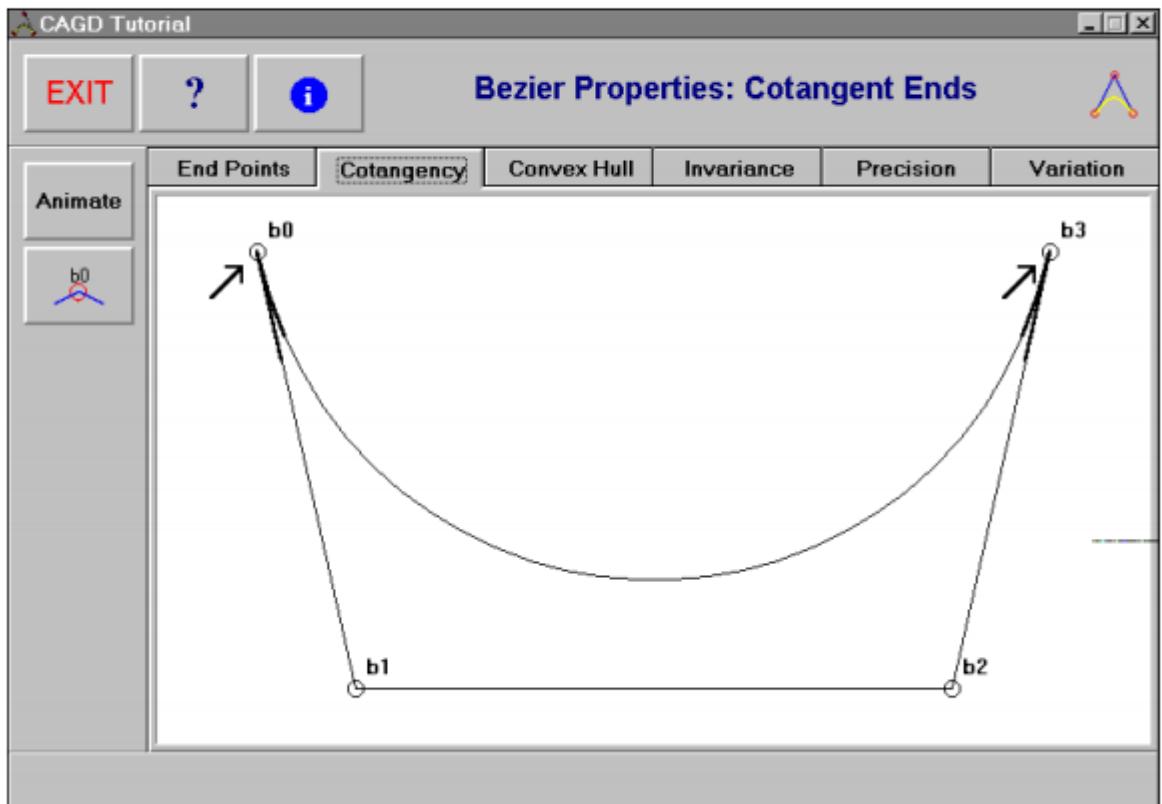
At \mathbf{b}_0 , $B_0^3 = 1$; at \mathbf{b}_3 , $B_3^3 = 1$



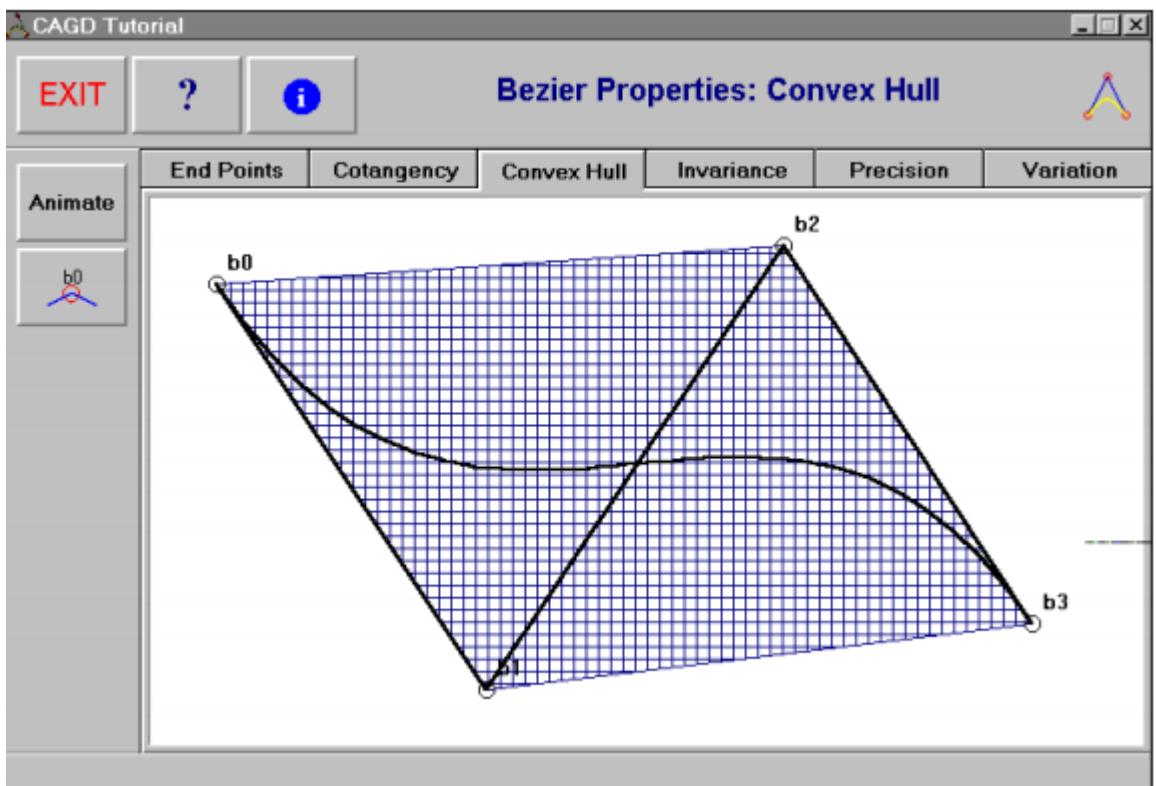
- Tangent - beziérovka je tečna na první a poslední segment kontrolního polygonu

$$f'(0) = (\mathbf{b}_1 - \mathbf{b}_0)n \text{ and } f'(1) = (\mathbf{b}_n - \mathbf{b}_{n-1})n,$$

where n is a constant.



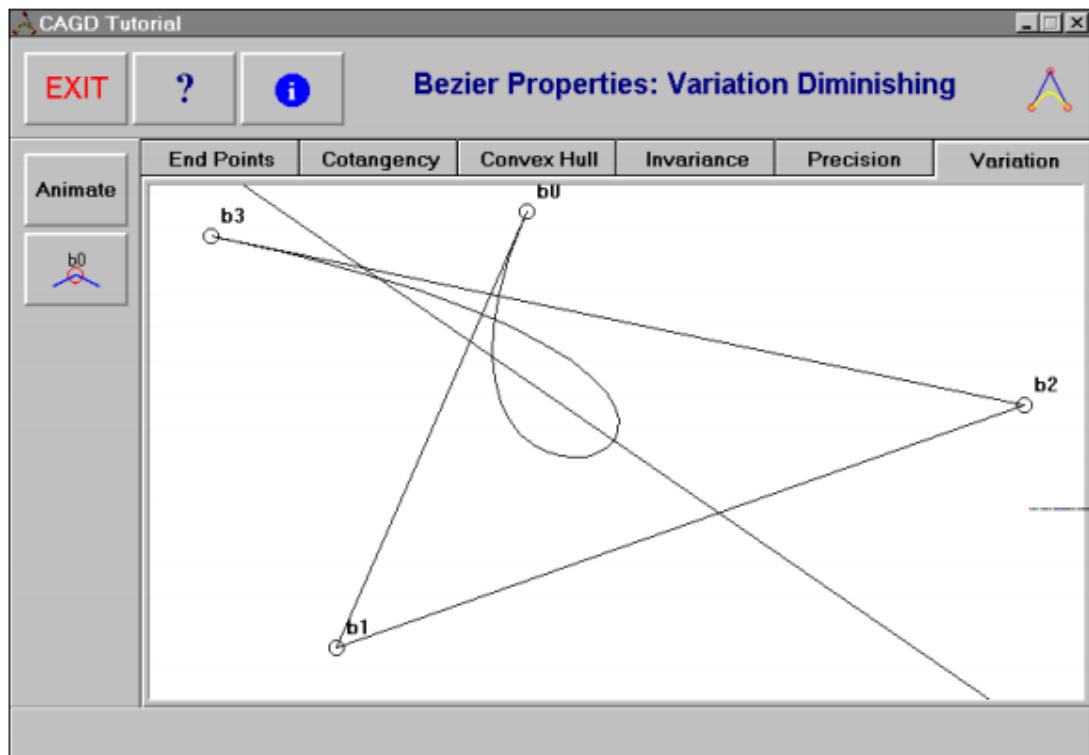
- Pro $0 \leq t \leq 1$ je uvnitř konvexního obalu kontrolních bodů



- Affiní invariance - **Cauchiho** podmínka - pokud aplikuju translaci nebo rotaci na každej od křivky, je to to samý jako je aplikovat na kontrolní body.
- Lineárně precizní - pokud jsou všechny body na přímce, bude to ta přímka.
- Variatin diminishing? - nevlní se víc než kontrolní body:

- Variation Diminishing

The Bézier curve is variation diminishing. It does not wiggle ³ any more than its control polygon; it may wiggle less. In this figure, notice that the straight line intersects the convex hull three times and also intersects the Bézier curve three times.



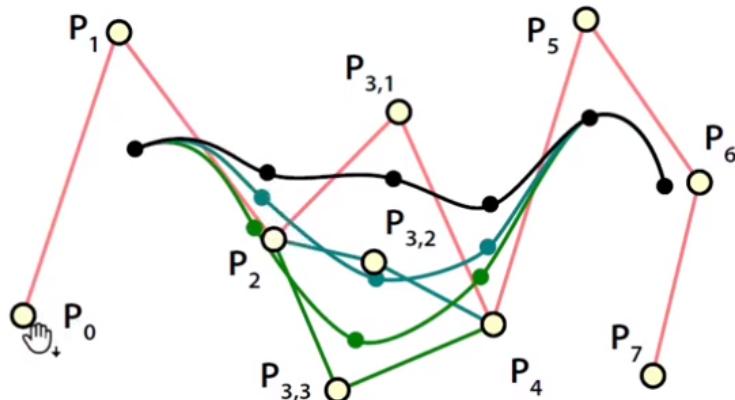
-

2.7. Catmull-Rom spliny

Viz podkapitolka o kubickejch splinech, je tam na konci.

2.8. B-spliny

Tzv. Coonsova kubika, neboli uniformní neracionální b-spline.



- continuity C^2
- sharing 3 CP between neighbours
- altering one CP induces change in closest 4 segments

$$MG = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

Určují ho 4 body, a maticce co je na obrázku. Křivka nezačíná v počátcích kontrolních bodů, ale začíná v tzv. Antitěžišti - je to $\frac{1}{3}$ těžnice trojuhelníko tvořenýho body P0, P1 a P2, která začíná v bodě P1 - antitěžiště proto, že těžiště je ve $\frac{2}{3}$ tý samý těžnice.

Počítá se $Q(0) = (P_0 + 4P_1 + P_2)/6$

Pokud chci stavět coonsův kubický B-spline, tak na sebe jednotlivý části navazuji tak, že když mám uzel P0,P1,P2,P3, tak pro další vezmu P1,P2,P3,P4. Tj, vezmu 3 body z toho předchozího, a přidám si další bod navíc.

2.9. de Casteljau a de Boor algoritmus

De Boor a De Casteljau je stejný název pro ten samej algoritmus, akorát ho vymysleli 2 lidi najednou a nezávisle na sobě, takže jak se mu říká záleží na tom, ke komu si blíž.

De Castlejau algoritmus se váže k Bézierově křivkám, a dá se použít k rozdělení nebo přesnějšímu vykreslení křivky. Není nutně efektivnější než Beziérovu křivku prostě přímo vynásobit, problém násobení je ale v přesnosti - dostávam moc malý čísla na velký mocniny, čímž narazím na limity s přesností floatů a doublů. De Casteljau to řeší, a funguje zhruba takhle:

The curve at point t_0 can be evaluated with the recurrence relation

$$\beta_i^{(0)} := \beta_i, \quad i = 0, \dots, n$$

$$\beta_i^{(j)} := \beta_i^{(j-1)}(1 - t_0) + \beta_{i+1}^{(j-1)}t_0, \quad i = 0, \dots, n-j, \quad j = 1, \dots, n$$

Then, the evaluation of B at point t_0 can be evaluated in $\binom{n}{2}$ operations. The result $B(t_0)$ is given by

$$B(t_0) = \beta_0^{(n)}.$$

Moreover, the Bézier curve B can be split at point t_0 into two curves with respective control points:

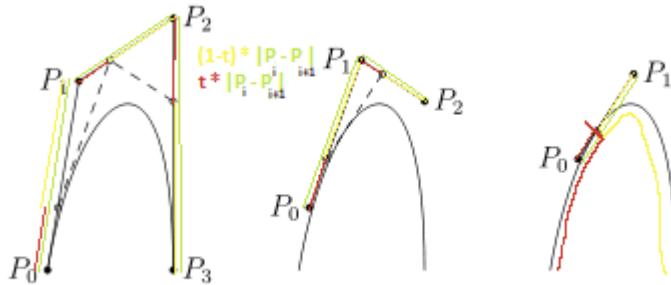
$$\beta_0^{(0)}, \beta_0^{(1)}, \dots, \beta_0^{(n)}$$

$$\beta_0^{(n)}, \beta_1^{(n-1)}, \dots, \beta_n^{(0)}$$

Jde o to, že abych dostal bod křivky v čase t, provedu tohle:

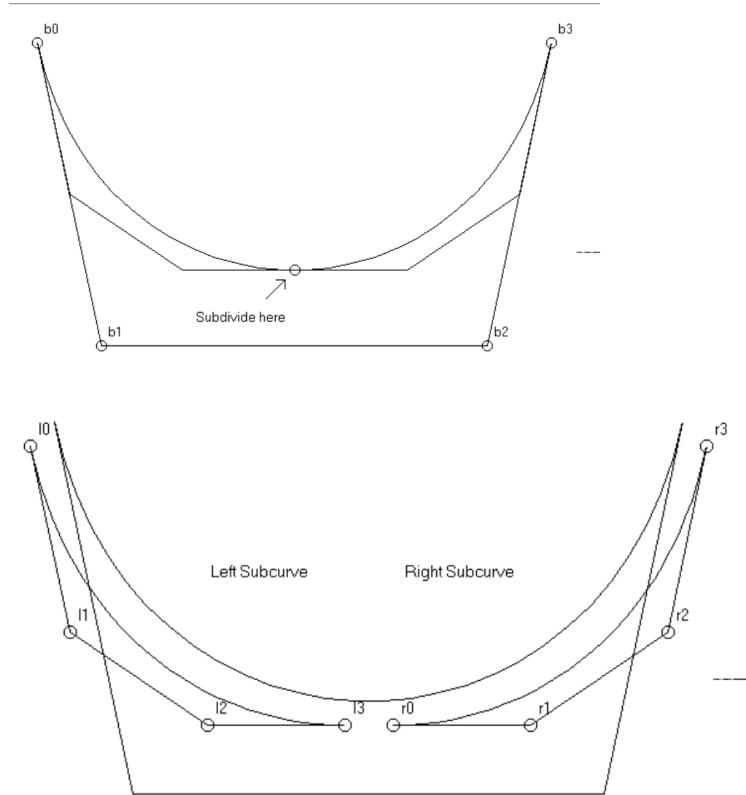
1. Pospojuji kontrolní body křivky úsečkama v jejich pořadí, tj $P_0 \rightarrow P_1 \rightarrow P_2 \dots \rightarrow P_n$
2. Na každý úsečce vyznačím body tak, aby jí rozdělila v poměru $t:(1-t)$
3. Nově vyznačený body pospojuji, takže mi vznikne o úsečku míř.
4. Opakuji kroky 2.-4. Pro nově vzniklý úsečky tak dlouho, dokud mi nezbyde jedna úsečka, a na ní jeden bod.

Vizuálně je to dost jasné:



Tady je vizualizace: <https://www.malinc.se/m/DeCasteljauAndBezier.php>

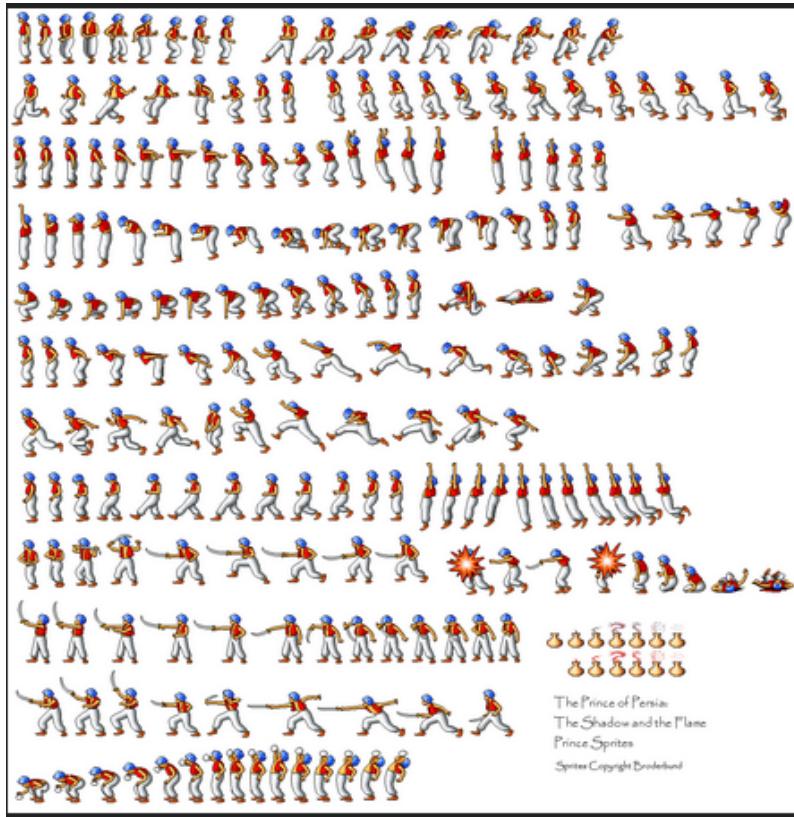
Co je na algoritmu ještě super je to, že když vezmu nově nakreslený body a rozdělím je půlce:



3. Animace postav, skinning, rigging

Animace - snaha vytvořit iluzi pohybu tím, že máme framy animace, ať už obrázky nebo pózy, a mezi nima interpolujeme/přeskakujeme v čase.

2D Animace jsou jednoduchý - mám Sprite Sheet, co má jednotlivý framy animace, a Animation engine prostě jenom musí vykreslit správnej obrázek v závislosti na čase a stavu animace.



Ve 3D je to složitější, ale teoreticky podobný.

Full Vertex Animation - mám “spritesheet”, který určuje kde má být který vertex v daném čase animace. Prostě framy animace vertex po vertexu. Moc se nepoužívá, ale má nejlepší možnost a největší přesnost, zabírá ale moc místa. Tisíce framů jsou potřeba.

Keyframes a interpolovat vertexy - Máme uložených několik framů - keyframes, které určujou pozice vertexů, a mezi nimi interpoluju za běhu. Taky zabírá moc místa. Komprese se používala, ale zas je pak potřeba je načíst do paměti, takže si tolik nepomůžeš

Bone Animation - vytvoří se “rig”, což je vlastně kostra modelu, a pak se animace tvoří tak, že máš uložený keyframey těch jednotlivých kostí, a zbytek modelu se dointerpoluje podle pozice kostí. Kosti jsou body (tj. ne kosti), a je uložená jejich pozice vzhledem k předchozí kosti (matice pozice)

Bones - definují reference frame a transformaci, mají počátek v kloubu, a transformují jemu přiřazený vertices v modelu spacu. Většinou na to mají nějaké constraints

Rigging - přiřazení vertex; jednotlivým kostem, jak daleko jsou od ní, s čím sou spojený a tak.

Vertex blending - vertex nepatří jenom k jedné bone, ale mají přiřazenejch několik (max 4 většinou, protože float4) blízkých kostí a váhu jak moc je ovlivňuje. Většinou to vypadá pak takhle:

Vertex blending

- Formally: vertex $\mathbf{u}(t)$ influenced by n bones from the rest pose vertex \mathbf{p} :

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}, \text{ where}$$

- Weights w_i for each bone: $\sum_{i=0}^{n-1} w_i = 1, w_i \geq 0$
- \mathbf{M}_i denotes initial bone to model space transformation
- $\mathbf{B}_i(t)$ denotes bone to model space transformation at time t
 - Typically concatenation of hierarchy of matrices and local bone transformation
 - In practice $\mathbf{B}_i(t) \mathbf{M}_i^{-1}$ concatenated together when passed to VS

Rest pose - póza ve které je model vymodelován, bejval T pose, teď se spíš dělá A pose protože vypadá líp v některých případech.



How to Get a Girlfriend



Vertex blending má pár problémů, animace vypadají rigidně, mají pár visuálních artefaktů (překrejvající se geometrie, roztaování textur, překrucování atd)

Dá se nějak řešit - pořádně definovat constrainty jointů, přidat více vertices, používat nelineární blending (třeba dual-quaternion), přidat virtuální bones (vezmeš si optimální

nelineární řešení, a přidáš virtuální bones tak, aby .lineární blending dosáhl podobného výsledku)

Komprese animací - V Ryse Son of Rome používali vertex animace pro cutscény. Třeba pozadí v jedný scéně mělo 30k vertices, což je zhruba 50MB/s dat pokud chci dosáhnout na 30 FPS bez komprese. Měli budget 10MBs, tak kompresi dělali takhle:

1. Quantizace (mapování vstupu do menšího prostoru, tj třeba zaokrouhlování atd), v tomhle případě použili převod na 3x uint16 pro pozici
2. Predikce - místo toho, aby se ukládaly pozice, tak se ukládají vektory mezi framama. Zneužívá se koherence, funguje to v podstatě uplně stejně jako při kompresi framů videa.
3. Komprese - na data (který sou hrozně zmenšený predikcí, a obsahují spoustu opakujících se věcí) půjde v klidu použít nějaké DEFLATE nebo LZ77

Animation pipeline

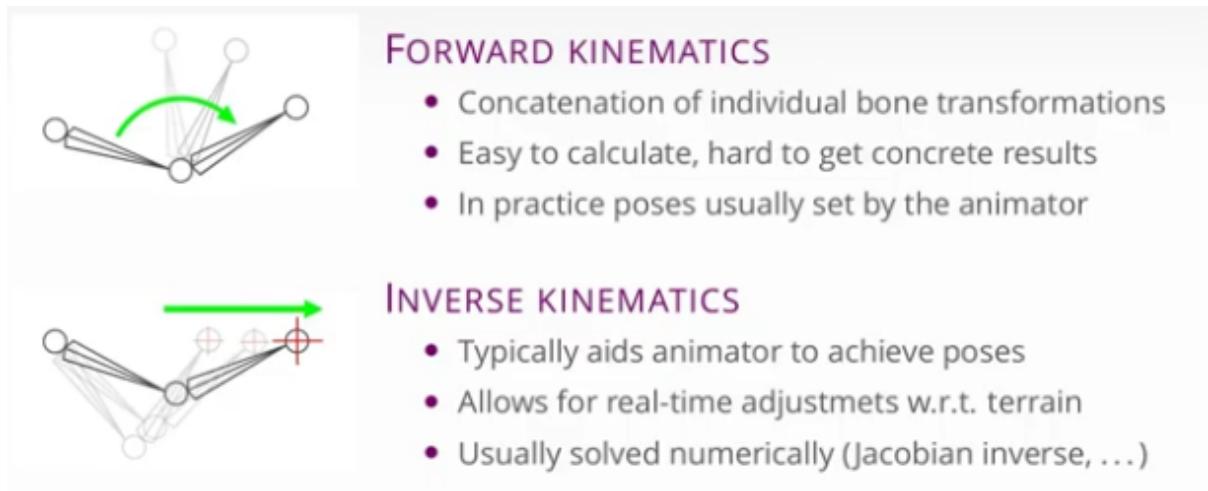
- Začneme s rigem, co má několik kostí (rigging - tvorba "kostlivce")
-)
- Animace se vytvoří z keyframů, křivek, a přes kosti - každá bone má svůj channel (většinou x,y,z a Euler), a ten určuje její pozici v keyframu. Mezi tím se pak interpoluje.
- Pak se exportuje do formátu to čeká engine
 - Channely se převedou, třeba do quaternionů nebo do jiný báze)
 - Bakne se - vytvoří se in-between frames (tj když mám třeba 3 keyframy na vteřinu, tak se jich dopočítá 27 mezi nima, aby šlo lineárně interpolovat), aby se dalo lineárně interpolovat mezi nima a animace nezrychlovala
 - Komprese nějaká
- Samplování animací, v enginu
 - Z poza rigu (rest pose) se musí naskinnovat na model, dopočítat local to model prostorová matice.
 - Za běhu se pak vyhodnotí animation graph, podle toho co chce animator/hra.
 - Graph vybere animaci a frame, tak si zakumuluju bone matice, vynásobím inverzí A-pose matice, a mám skinning transformaci co nacpu do Vertex Shaderu.

Animační graf - Bezstavověj přímej acyklickej graf animačních nodů

Do vstupu mu přijdou vstupy, delta času, kontexty. Animace bejvaj v listech. Průchod grafem na základě dat co přišli určí, co se bude dít. Několik nodů:

- Blend nodes — blend between 2 and more animation samples
- Logical nodes — switch between several animation routes
- State machines — perform complex animation switching (state passed in through context)

Dalším pojmem je Inverse Kinematics:



Chci třeba aby se postava držela žebříku, takže chci aby bone ruky byla na daný pozici. IK dopočítá procedurálně pozici ostatních kostí tak, aby dodržela constrainy.

Forward Kinematic - Jdu od root bony, nastavím její poholu, a jdu ke všem dětem, těm taky nastavím plohu (na základě polohy rootu), atd.. Pro 2 Bony co mají jeden joint mezi sebou a hýbou se v jednom směru, vypadá třeba takhle: (PxPy je pozice ruky, tj poslední bony)

use matrix bones and their end points

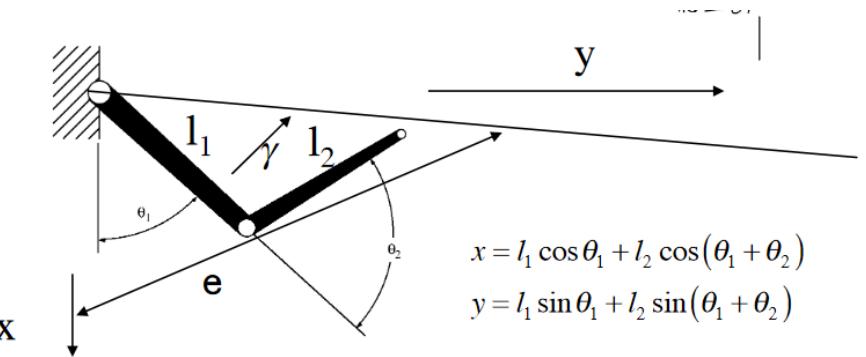
$$P_x = L_1 \cdot \cos(\theta_1) + L_2 \cdot \cos(\theta_1 + \theta_2)$$

$$P_y = L_1 \cdot \sin(\theta_1) + L_2 \cdot \sin(\theta_1 + \theta_2)$$

Inverse - chci tomuhle inverzí funkci. Chci, aby Px a Py skončili na určitým místě.

Problémy -

- Ně vždycky má řešení
- Většinou má neomezeně moc řešení
- Neexistuje vždycky analytický řešení (tj, použitelný vzoreček, co to přesně spočítá, musí se odhadovat)
 - Pokud mám stejně constrainů jako degrees-of-freedom, tak existuje.
 - Prostě invertuju matematický operace pro Forward Kinematic



$$x = l_1 \cos \theta_1 + l_2 \cos(\theta_1 + \theta_2)$$

$$y = l_1 \sin \theta_1 + l_2 \sin(\theta_1 + \theta_2)$$

- Inverse Kinematics:

$$l = \sqrt{x^2 + y^2}$$

$$l_2^2 = l_1^2 + l^2 - 2l_1 l \cos \gamma$$

$$\Rightarrow \gamma = \arccos\left(\frac{l^2 + l_1^2 - l_2^2}{2l_1 l}\right)$$

$$\frac{y}{x} = \tan \varepsilon \quad \Rightarrow \quad \theta_1 = \arctan \frac{y}{x} - \gamma$$

$$\theta_2 = \arctan\left(\frac{y - l_1 \sin \theta_1}{x - l_1 \cos \theta_1}\right) - \theta_1$$

○

Iterativní řešení - Nejčastěji se k tomu používá Jacobian. Skvěle popsáný je to tady:

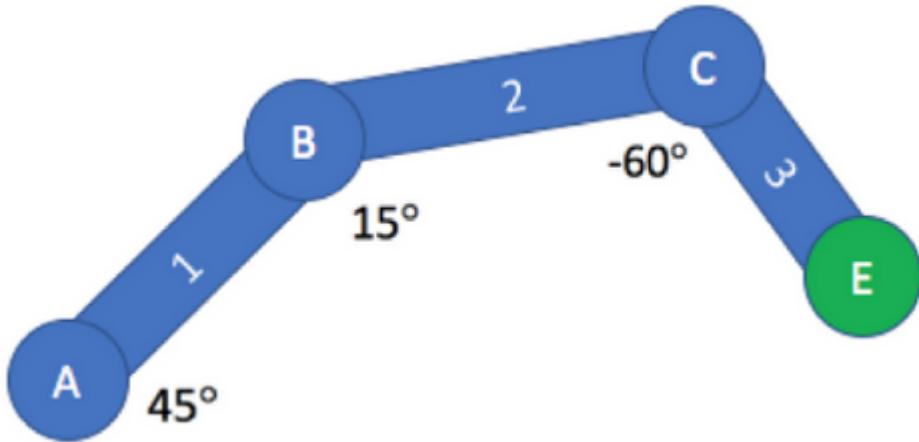
<https://medium.com/unity3d-animation/overview-of-jacobian-ik-a33939639ab2>

Jacobian je matici, která určuje jak moc se změní soustava když pohnu z jednou ze složek.

$$J = \begin{bmatrix} \frac{\partial p_x}{\partial \theta_A} & \frac{\partial p_x}{\partial \theta_B} & \frac{\partial p_x}{\partial \theta_C} \\ \frac{\partial p_y}{\partial \theta_A} & \frac{\partial p_y}{\partial \theta_B} & \frac{\partial p_y}{\partial \theta_C} \\ \frac{\partial p_z}{\partial \theta_A} & \frac{\partial p_z}{\partial \theta_B} & \frac{\partial p_z}{\partial \theta_C} \end{bmatrix}$$

Je to matici, která se staví takhle:

Tohle je příklad, kdy mám ve 3D 3 bony, kde každá je otočena o úhel Theta. P(x,y,z) je pozice bodu E, co mě zajímá (tj. rukou na konci soustavy):



V Jacobianu je teda pro každej sloupec derivace funkce P, která počítá výslednej bod, složkou jednoho z úhlů - určuje to teda o kolik se změní pozice bodu E, když trošku hnu s úhlem (tj, derivace).

Pozice bodu E je tedy počítaná forward kinematik způsobem funkcí: $P(O) = p$, kde p je bod a O je vektor všech úhlů [Th1, Th2, Th3]. V případě jenom 2 ramen by to bylo třeba

$$P_x = L_1 \cdot \cos(\theta_1) + L_2 \cdot \cos(\theta_1 + \theta_2)$$

$$P_y = L_1 \cdot \sin(\theta_1) + L_2 \cdot \sin(\theta_1 + \theta_2)$$

(ted' máme ale 3).

Abych zjistil Inverzní Kinematiku, potřebuju najít funkci $P^{-1}(p) = O$.

Můžeme si to rozdělit, a brát to tak, že $p = P(O+Od)$, kde Od je nějaká změna vektoru úhlů, kterou sem se hnul u každý úhlu za frame, a O je výchozí pozice. Jakmile zjistím Od , tak vím o kolik hnout s jakýma úhlama a mám vyhráno.

Máme ale matici Jacobianu J , která nám říká, o kolik mě hne kterej z úhlů. Tím pádem platí tohle: Cesta mezi E a T , tj $T-E = J * Od$, kde E je ruka před posunutím, a T je cíl kam se chceme dopočítat.

Tím mám vyhráno, stačí mi zjistit inverzi matice J , tou vynásobit $T-E$ a získám Od :

$$Od = J^{-1} * T-E$$

Bohužel, J nemusí být čtvercová, pokud mám víc úhlů než souřadnic, tudíž nemusí mít inverzi. Na to ale máme řešení, nesrat se s tím a prostě použít Transponaci J matice, která to trošku odhadne. Protože ale není uplně nejpřesnější, je potřeba algoritmus provádět iterativně a po malejch kouscích, a postupně se přibližovat. IK tedy počítám takhle:

C = cílovej bod kam chci s rukou

O = počáteční úhly

$E = P(O)$ - počáteční pozice ruky

$h = 0.0001$ - iterativní step velikost, je potřeba protože approximuj transpozicí, a jinak by to nemuselo vyjít

```

while( abs(vzdalenost C a E) > tolerance)
    Spocitam Od = JT * (C - E)
    O = O+Od * h - přičtu malinkej kousek toho co sem nepřesně spočítal, ať
neprestelím)
    E = P(O)

```

Motion Capture:

Nahrávám lidi co mají na sobě HW, a z toho jak hrajou vytáhnu animace.

- Optical - používají tuny kamer a barevný značky. Potřeba calibrovat, hrozně prostoru, světlo moc nedává, musí vidět na značky.
- Non-optical - používá různý neoptický senzory
 - Inertial system - nepotřebuje prostor ani snezory kolem, na herce se nalepí pár gyroskopů (hodně zjednodušeně). Je potřeba fakt dobře kalibrovat a spíš nemá moc dobrý výsledek
 - Magnetický systémy - Používají magnetický pole a sledují značky, podobně jako Optical systemy. Nevadí, že nevidí na značky. 6Degrees of Freedom. Problémem je cena a interference Elektrovln kolem. (*Když nahrávali Maffi, tak museli rekalibrovat systém pokaždý když kolem na ulici projela tramvaj. Což někde v kanclu na holešovivých bylo docela často*)

Další dělení je

- Marker-based systems
 - Aktivní
 - Třeba ledky, co blikaj různě podle toho de a jak se hejbou. Přesný, ale drahý
 - Pasivní
 - Reflexní tečky. Nic moc přesnost, často se rozkalibrujou
- Marker-less
 - Nepotřebuješ marky, používá se třeba Deep learning, nebo podobný computer vision techniky.
 - Levný, nepřesný, potřeba mít fakt dobře postavený kamery.
 - Face motion capture je docela ok v tomhle

Matika fuguje zhruba tak, že mám přesný pozice kamer, vím jak sou otočený, vím kde jsou vůdci středu světa, a promítám paprsky na jednotlivý sledovaný body. Spojím data z více kamer, a podle toho se pokusím odhadnout kde je bod a jak se pohnul. Nejde řešit přesně kvůli measurement errorům, takže se většinou approximuje.

Physically-based animation - počítám animace za běhu na základě nějaké fyziky. Třeba vlasy, cloth, Nvidia PhysX. Existují i animační SW co jsou založeny na anatomicky přesných modelech, a animace generují na základě kontrakcí svalů - používá se třeba pro lepší generování animací obličeje, ale pro runtime použití je pomalý a zbytečný.

4. Detekce kolizí

Není v žádný z těch dporučenejch přednášek, ale je tady -
<https://cgg.mff.cuni.cz/~pepca/lectures/pdf/2d-07-collisions.pdf>

Potřebuju zjistit, který tělesa ve scéně spolu kolidují (tj mají neprázdný průnik), a chci:

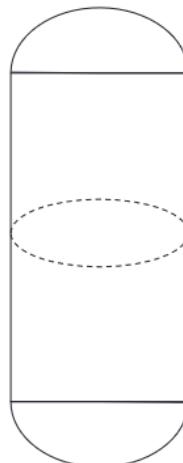
- Čas a místo kdy ke kolizi došlo k prvnímu kontaktu
- Hloubka vnoření
- Vektor separace - přibližně nejkratší směr kudy šoupnout obejky tak, aby nekolidovali. Například když se mi spawnou dvě věci do sebe, tak je oddělit.

V naprostý většině řešení se kolize počítá na základě collideru, tj jednoduchého tvaru, u kterého se dá jednodušeji počítat kolize, než to dělat vertex po vertextu:

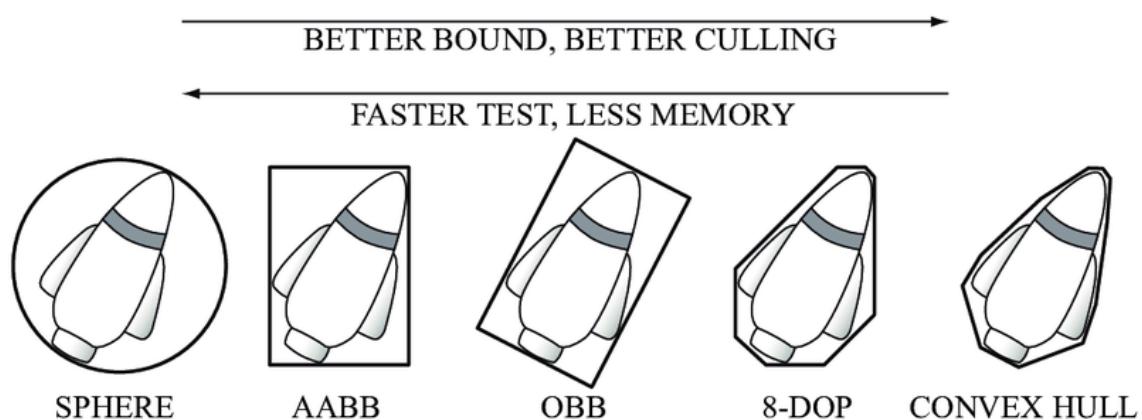
- Geometry used for rendering:



- Geometry used for collision detection:



Používá se převážně několik tvarů, který mají jednoduchý detekce:



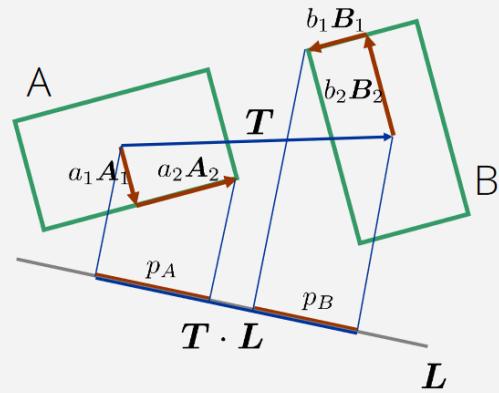
Sphere - prostě mám kruh/kouli určenou středem a průměrem

AABB - Axis Aligned Bounding Box - Mám zadaný Min a Max hodnoty v každý ose světa - když se objekt točí, musím přepočítat - neotáčí se totiž.

OBB - otočenej bounding box - určují mi ho na sebe kolmý vektory, každej o velikosti poloměru jedný se stran, a můžu s ním otáčet

OBB Overlap Test in 2D

- $\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2$ are normalized axes of A and B
- a_1, a_2, b_1, b_2 are radii of A and B
- \mathbf{L} is a normalized direction
- \mathbf{T} is the distance of centers of A and B
- $p_A = a_1 \mathbf{A}_1 \mathbf{L} + a_2 \mathbf{A}_2 \mathbf{L}$
- $p_B = b_1 \mathbf{B}_1 \mathbf{L} + b_2 \mathbf{B}_2 \mathbf{L}$
- A and B do not overlap in 2D if $\exists \mathbf{L} : \mathbf{T} \cdot \mathbf{L} > p_A + p_B$

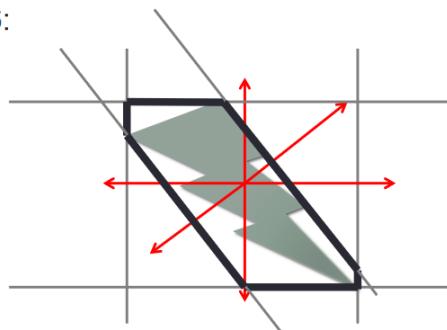


V knížce Grafiky k tomu ještě psali, že se test ve 3D provádí tím, že si promítnu oba OBB na jednu z os (třeba Xová, tj vezmu největší a nejmenší X bod obou tvarů), a když se neprotnou, tak určitě nemaj kolizi. Když se protnou, tak se musí podívat na jinou osu.

Je dokázaný, že stačí udělat tenhle test maximálně v 15 přesně určených osách (jenom už nepsali jaký), aby jsi vyloučil kolizi OBB ve 3D. Provést se to dá pod 100 instrukcí, porovnání 2 OBB.

Collision Shapes: k-DOPs

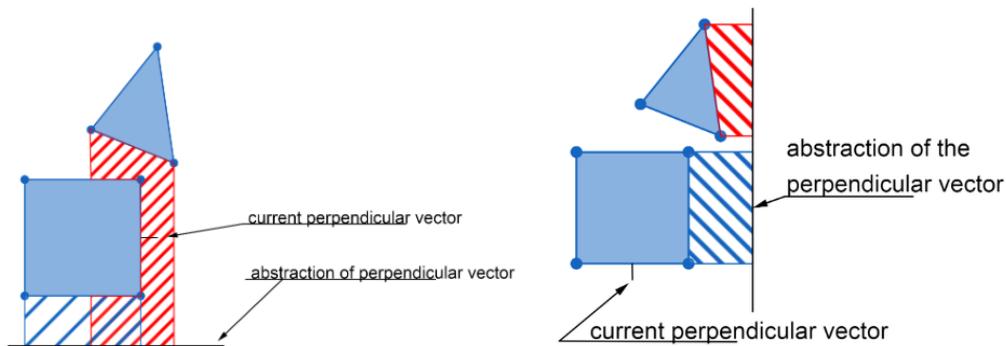
- Discrete Oriented Polytopes
- Given k vectors (each determining an orientation)
- Move a plane perpendicular to each vector as close as possible to the object at hand
- Ex, k = 6:



DOP - Jedná se o kolizní tvar kterej si určím tak, že zadám k vektorů. Pro každej vektor vytvořím rovinu která je na něj kolmá, a připlácnu jí co nejbližši na objekt (tj najdu nejbližší

kolmou rovinu na vektor, která nekoliduje s meshem co popisuji). Dají se tím dobře popisovat Meshes - normála každého vertexu se stane vektorem pro DOP.

Když mám počet a směr vektorů DOP mám pevně daný pro celou scénu (tj všechny objekty popisují stejný vektor), tak se pak porovnává relativně v pohodě - vyzkouším k/2 os, a pro každou se podívám, jestli se překrejvají. **Separating Axis Theorem** říká, že když najdu jednu osu, na kterou když promítnu objekty, tak se nepřekrejvají, tak spolu nekolidujou. V podstatě to totiž říká, že sem schopnej mezi nima nakreslit čáru. Platí ale jenom pro konvexní tvary.



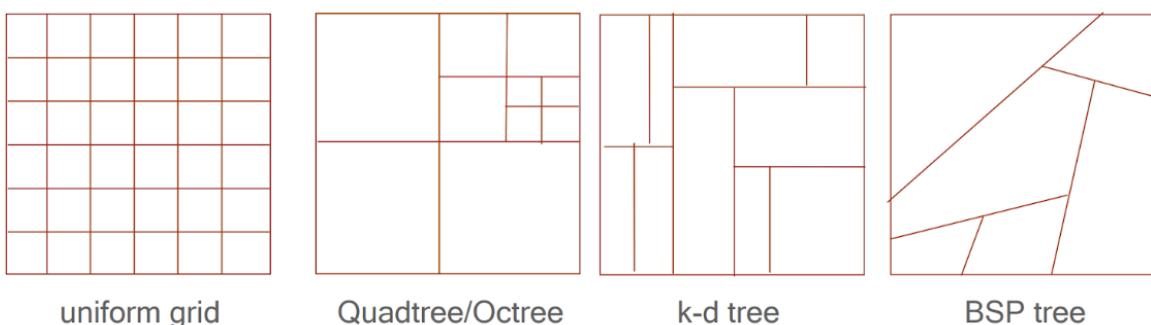
Optimalizace

Dalším problémem je složitost - pokud chci kolize mezi všemi objektama, musel bych kontrolovat každý z každým. Scény mívají tisíce objektů, proto musím nějak filtrovat kdo už je dost blízko. Dělá se to pomocí různých přístupů, obecně ale mají dvě fáze:

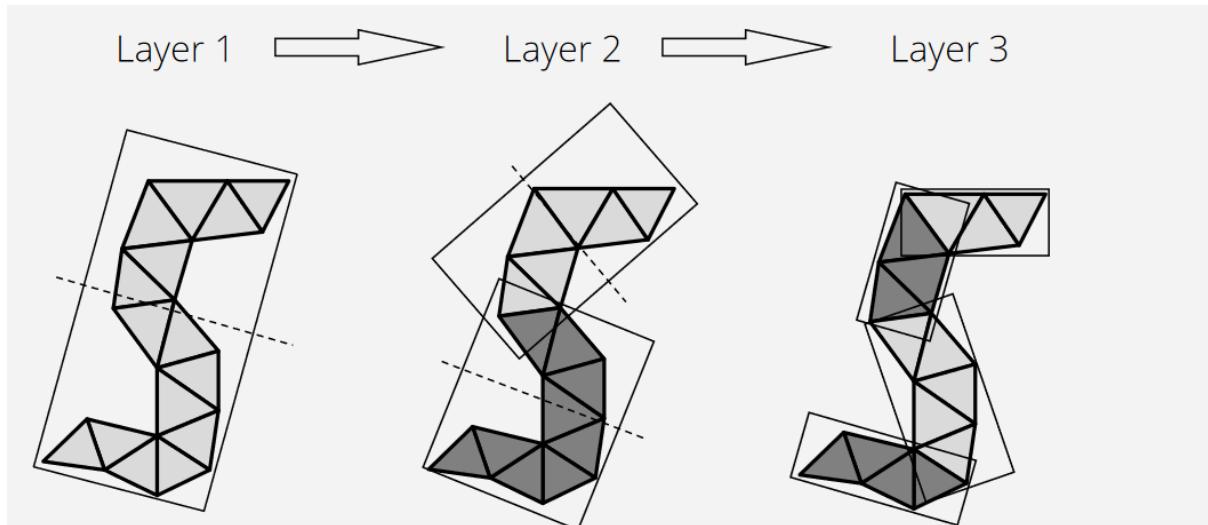
- Broad Phase - detekuu který objekty jsou blízko u sebe, třeba pomocí gridu, nebo obřích bounding boxů, nebo rozdelením objektů do větších skupinek
- Narrow phase - upřesňuji kolize co jsem detekoval že by se mohly stát

Přístupů na optimalizaci je několik:

- Brute force - prostě dělám každej s každým, a mám málo objektů
- Spacial Partitioning
 - Rozdělím si world na menší kusy, u každý vím který objekt do ní patří, a kolize kontroluji jenom v rámci skupiny
 - BSP-trees, Q-trees, Grids, K-D Trees...
 - BSP je super pro uzavřený prostory, můžu pro objekt zjistit jestli je v legální pozici nebo ne.
 - Sphere-trees je varianta BSP dobrá pro dynamický objekty



Můžu taky dělit objekty na menší, pro přesnější kolize:



5. Obrázky, textury a práce s nima

5.1. 2D Fourierova transformace a konvoluce

Fourierova transformace je transformace signálu z domény času do domény frekvencí. Jde o to, že mám nějaký periodický signál, a zajímá mě, z jakých frekvencí je složený. Existuje i její spojitá verze, nám ale stačí ta diskrétní (tj. omezená nějakým sampling ratem):

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

Xk - poměr frekvence k v signálu

co se nasamploval na body x0 - xn

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N}$$

- xn - nasamplovaná hodnota signálu v bodě n. Je to inverzní funkce k transformaci vejš.

Tak jako zevrubně popsána ta idea je taková, že se podívám na všechny možné frekvence sinusovek, a vydělím tou sinusovkou svůj singál. Tím získám graf, kterej mi určuje, jakou součástí se ta testovaná sinusovka projevuje na výsledku.

Nejde ale o sinusovky, místo toho se používá Eulerova formule k tomu, aby se samplovalo po kruhu. To je tahle:

$$e^{i\pi} = -1$$

"It is **absolutely paradoxical**; we cannot understand it, and we don't know what it means, but we have proved it, and therefore we know it must be the truth."

— BENJAMIN PEIRCE, 1759

To nám v podstatě říká, že pokud posunu e^{ix} o π , tak dostanu půlkruh. Celkově, když si nakreslím graf e^{ix} , dostanu kruh. Taky se přepisuje jako $e^{ix} = \cos(x) + i\sin(x)$
A na tom staví fourierova transformace:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

Pro zjištění poměru jedné frekvence (X_k), prostě točíme signálem po kruhu na ty frekvenci, a průměrujeme body na té cestě |

Hrozně hezky to popisuje třeba

tady: <https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/>

K čemu se používá? Na fourirově transformaci signálu je hezky vidět, které frekvence se v něm vyskytujou, a tudíž se dá třeba filtrovat noise. Když vím, že mi tam něco hučí stabilní frekvencí, podívám se na FT a kouknu kde vyskočí do vysokých hodnot, tj koeficient je vysoký:

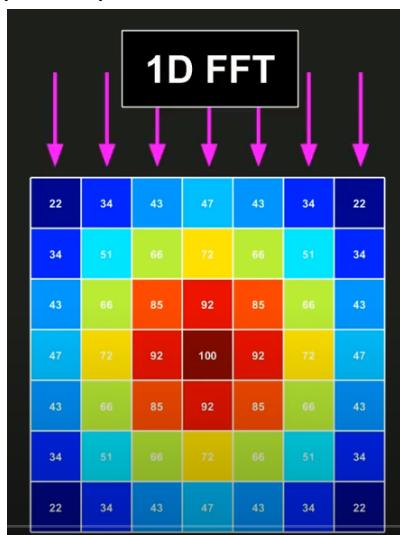
https://upload.wikimedia.org/wikipedia/commons/7/72/Fourier_transform_time_and_frequency_domains_%28small%29.gif

2D Fourierova transformace dělá to samý, akorát přidává další dimenzi:

$$F(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) e^{-j2\pi(x\frac{m}{M}+y\frac{n}{N})}$$

$$f(m,n) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F(x,y) e^{j2\pi(x\frac{m}{M}+y\frac{n}{N})}$$

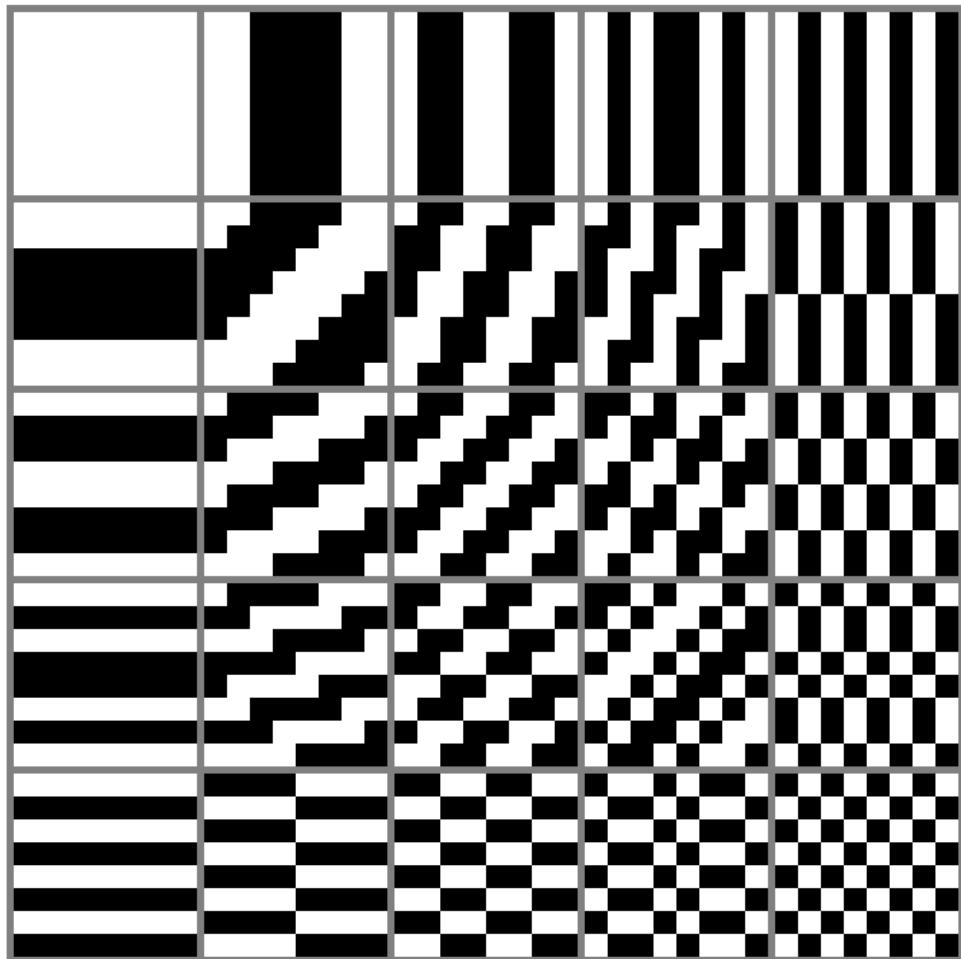
Mám obrázek MxN. a postupně skládám vertikální a horizontální sinusovky. Nejdřív nabere po sloupcích několik 1D fourier transformací



ke kterejm přičtu FT po řádcích:



Výsledkem je matice koeficientů, která určuje zastoupení jednotlivých kombinací frekvencí horizontálních a vertikálních: (Ono, frekvence není správný slovo, protože už jde o 2D věc. Spíš



Tenle obrázek ukazuje, jak zhruba vypadají jednotlivé komponenty, kterých se snažím zjistit poměr v obrázku. Používá se třeba v JPEG komprese (Kde je teda sinusová transformace, což je podobná věc, akorát používám jenom sin a né cos, tj vynechám imaginární část).

Konvoluce

Konvoluce popisuje další způsob, jakým se dají skládat dohromady dvě funkce. Dobře intuitivní způsob jak o tom přemýšlet co jsem viděl je na příkladu s ohňostrojem:

- Funkce $f(t)$ popisuje, kolik ohňostrojů vybuchlo v čase t
- Funkce $S(t)$ popisuje, kolik kouře zbylo po výbuchu jednoho ohňostroje v čase t po výbuchu. (Tj, bude to hodně v t_0 , a klesat k nule)
- Zajímá mě, kolik je celkem ve vzduchu kouře v čase T .

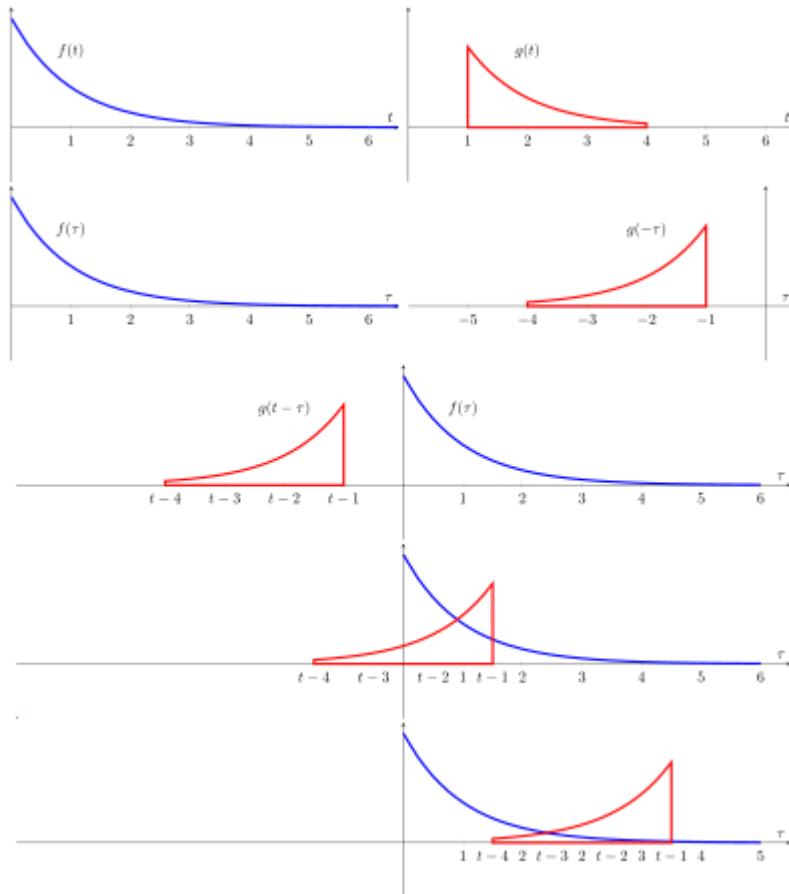
Vzoreček vypadá takhle:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau.$$

popřípadě diskrétní takhle:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m],$$

Vizuálně si to jde představit tak, že prostě vezmu funkci g , tu otočím kolem osy Y (tj beru pozpátku), posunu jí počátek tak, aby byl v bodě t (nebo n), funkce mezi sebou vynásobím a vezmu integrál/plochu pod grafem:



S tím, že výsledek je právě ta plocha pod grafem, tj jejich průsečík v podstatě.

Animace:

https://upload.wikimedia.org/wikipedia/commons/b/b9/Convolution_of_spiky_function_with_box2.gif

2D varianta je v podstatě uplně stejná, ale používá se na 2D funkce:

$$y[i, j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} h[m, n] \cdot x[i - m, j - n]$$

Idea je tam úplně stejná. Hlavní použití je ve filtrování obrázků - definuju si malou mřížku (**kernel**) několika hodnot, třeba pro jednoduchý blur to je:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

, což je moje funkce $x[i,j]$ z předchozí rovnice. (Aby funkce vejš fungovala, je potřeba ten kernel nejdřív otočit, což když je identita je jedno, ale obecně není. Tj třeba místo

A,B

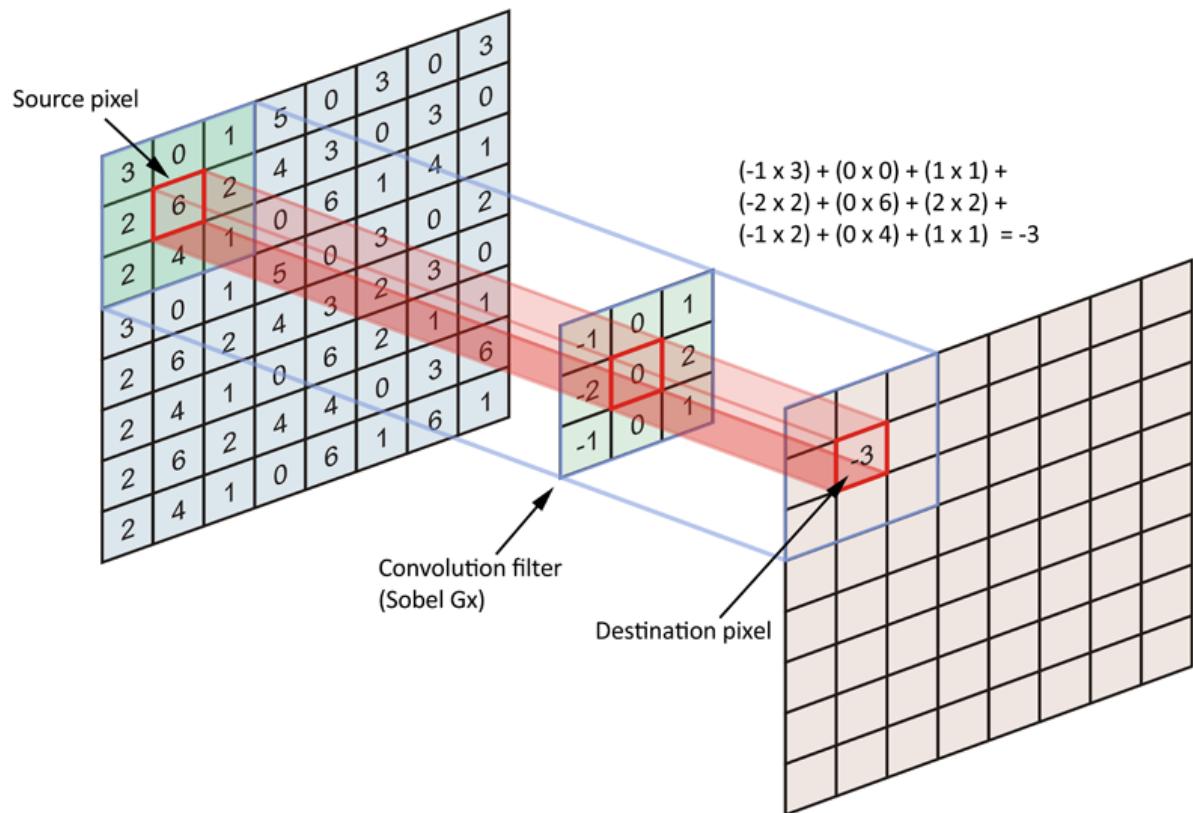
C,D

Vzít

D,C

$B,A.)$

Jako funkce $h[i,j]$ se použije obrázek. Pak spočítám konvoluci $y[i,j]$ podle vzorečku co je vejš, a když si to rozpočítám a rozkleslím, dojdu k tomu, že to dělá přesně to co chci - Postupně projede obrázek od 0,0 až do m,n , a pro kažej bod i,j vezme z obrázku okolí bodu i,j a do přenásobí hodnotama ve mřížce:



Používají se různý kernely (tj filtrační matice), a podle toho dostanu výsledek. Třeba tím jde detekce han, blur, ostření a podobně.

Convolute a Fourier - Convolution Theorem - ještě je tu jedna důležitá vlastnost -

Convolution theorem

$$r(x) = \{g * h\}(x) = \mathcal{F}^{-1}\{G \cdot H\}, \quad \text{where } \cdot \text{ denotes point-wise multiplication}$$

Prostě- Konvoluce r(x) fukncí g(x) a h(x) se rovná inverzní FT z násobků FT obou funkcí.

Prostě, abych získal funkci konvoluce, nemusím si to složitě rozpočítávát - můžu si spočítat Fourierovu Transformaci funkcí g a h, a FT konvoluce se rovná součinu FT funkcí g a h.
 $\text{FT}(g*h) = \text{FT}(g) \cdot \text{FT}(h)$, kde tečka je normální bodový násobení.

Shrnutí:

Spojité verze [\[editovat zdroj\]](#)

- Dopředná Fourierova transformace: $F(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) e^{-2\pi i(ux+vy)} dx dy$
- Zpětná Fourierova transformace: $f(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(u, v) e^{2\pi i(ux+vy)} du dv$
- Konvoluce: $(f * g)(x, y) = (g * f)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(a, b) g(x-a, y-b) da db$ Vlastnosti:
 - komutativní $f * g = g * f$
 - asociativní $f * (g * h) = (f * g) * h$
 - distributivní $f * (g + h) = f * g + f * h$
 - asociativita při násobení skalárem: $a(f * g) = (af) * g = f * (ag)$
 - Existence jednotky: $f * \delta = \delta * f = f$ (δ je diracova delta fce)

Vlastnosti [\[editovat zdroj\]](#)

- Convolution theorem: $\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$
- Linearita: $\mathcal{F}\{a \cdot f + b \cdot g\} = a \cdot \mathcal{F}\{f\} + b \cdot \mathcal{F}\{g\}$
- Shift theorem: $\mathcal{F}\{f(x-x_0, y-y_0)\}(u, v) = e^{-2\pi i(ux_0+vy_0)} F(u, v)$
- Rotace: $\mathcal{F}\{Rot(f)\} = Rot(\mathcal{F}\{f\})$

Diskrétní verze [\[editovat zdroj\]](#)

- Dopředná Fourierova transformace: $F_{n,m} = \frac{1}{\sqrt{MN}} \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} f_{k,l} e^{-2\pi i(\frac{km}{M} + \frac{ln}{N})}$
- Zpětná Fourierova transformace: $f_{k,l} = \frac{1}{\sqrt{MN}} \sum_{m=0}^{N-1} \sum_{n=0}^{M-1} F_{n,m} e^{2\pi i(\frac{km}{M} + \frac{ln}{N})}$
- Konvoluce: $(f * g)[m, n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j) g(m-i, n-j)$

5.2. Vzorkování a kvantování obrazu

Matematický model vzorkování, Shannon theorem [\[editovat zdroj\]](#)

$f(x, y) \cdot s(x, y) = d(x, y)$, kde f je původní funkce, s je vzorkovací fce (pole delta funkci) a d je navzorkovaný obraz.

- $F(u, v) * S(u, v) = D(u, v)$
- $s(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \delta(x - i\Delta x, y - j\Delta y)$
- $S(u, v) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \delta(u - i \frac{1}{\Delta x}, v - j \frac{1}{\Delta y})$

Fourierův obraz navzorkované funkce ($D(u, v)$) je tvořen do mřížky poskládanými spektry původní funkce s roztečemi $\frac{1}{\Delta x}$ a $\frac{1}{\Delta y}$.

Dokážeme zrekonstruovat původní funkci pouze pokud se nám jednotlivá spektra neslijí a to platí jen pokud je původní funkce frekvenčně omezená a vzorkujeme s dostatečnou frekvencí:

$$\Delta x \leq \frac{1}{2W_u} \text{ a } \Delta y \leq \frac{1}{2W_v}, \text{ kde } W_u \text{ a } W_v \text{ jsou maximální frekvence v základních směrech.}$$

Potřebujeme dvakrát vyšší frekvenci než je maximální přítomná frekvence v původní fci.

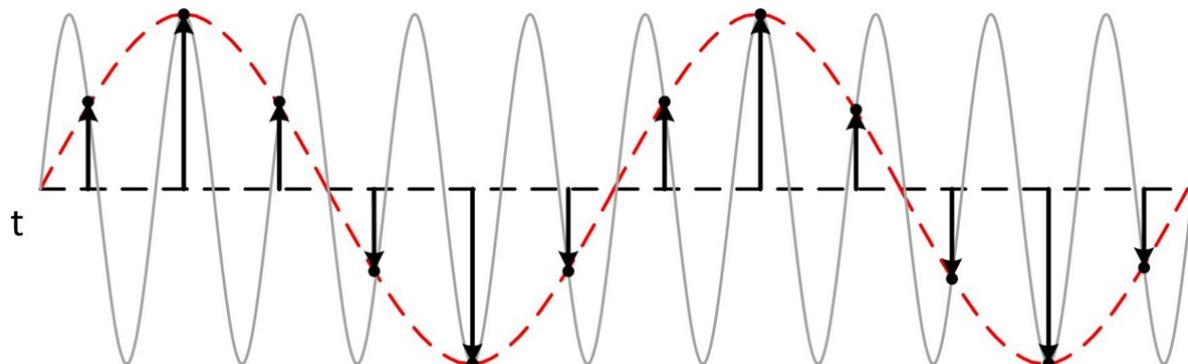
Negativní projevy podvzorkování [\[editovat zdroj\]](#)

- aliasing (stráta vysoko frekvenčnej informacie - hrany, detaily)
- Moiré efekt - falešné nízké frekvence

Kvantování [\[editovat zdroj\]](#)

- Diskretizace oboru hodnot signálu - vždy ztrátové.
- Často se kvantizér navrhuje tak aby využíval vlastnosti lidského oka - např. nerozlišitelným jasovým úrovním se přiřazuje stejná hodnota

Nejdůležitější je tady Shannon theorem, ten nám říká, že potřebujeme alespoň 2x větší frekvencí samplování než je nejvyšší frekvence co se v signálu vyskytuje, aby se povedla zrekonstruovat. Je to hezky vidět na obrázku, co se stane, když se v signálu objeví nějaká frekvence která je vyšší než jak sampluju:



This Photo by Unknown Author is licensed under CC BY-SA

U obrázků to platí stejně, jde akorát o to, jak se mění barvy a jas mezi jednotlivými pixelama. Viz 2D fourierova transformace, která to v podstatě popisuje. Tj, nejmenší detail v digitálním obrazu musí být minimálně dvojnásobkem vzorkovacího intervalu.

Nyquistova hodnota - jedná se o frekvenci, kterou potřebuju, abych mohl rekonstruovat celej obraz. Tj, aby se nestal Shannonův theorém. Takže potřebuju samplovat aspoň takhle:

$$\Delta x \leq \frac{1}{2W_u} \text{ a } \Delta y \leq \frac{1}{2W_v}, \text{ kde } W_u \text{ a } W_v \text{ jsou maximální frekvence v základních směrech.}$$

určuje vzdálenost mezi bodama co sampluju.

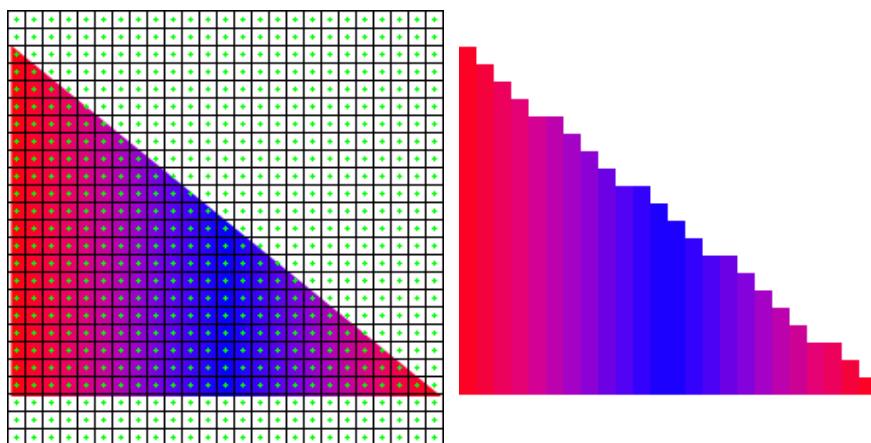
Digitalizace obrazu - kvantování a vzorkování. Máme obrazovou funkci $f(x,y)$, která nám popisuje jakou barvu má obraz v bodě x,y . Ve světě je to spojitá funkce s neomezenou úrovní detailů, a obor hodnot je taky spojitej a neomezenej. Převedení spojité funkce obrazu do diskrétní funkce (tj, mám omezený rozlišení a mám omezený počet barev) se jmenuje digitalizace obrazu. A samozřejmě, přechod ze spojité do diskrétní nám zanáší artefakty a aliasing.

Kvantování popisuje krok, při kterém si musím obor hodnot obrazové funkce, tj barvy, který naměruju, a který mají nejspíš spojitou hodnotu, nějak převést do omezeného prostoru barev co můžu zobrazit. Už to není úplně aktuální problém při použití HDR, kde mám barev víc než potřebuji, ale předtím bylo potřeba nějakým způsobem zobrazit těchhle neomezeně barev do rozmezí 255 u každý barevný složky.

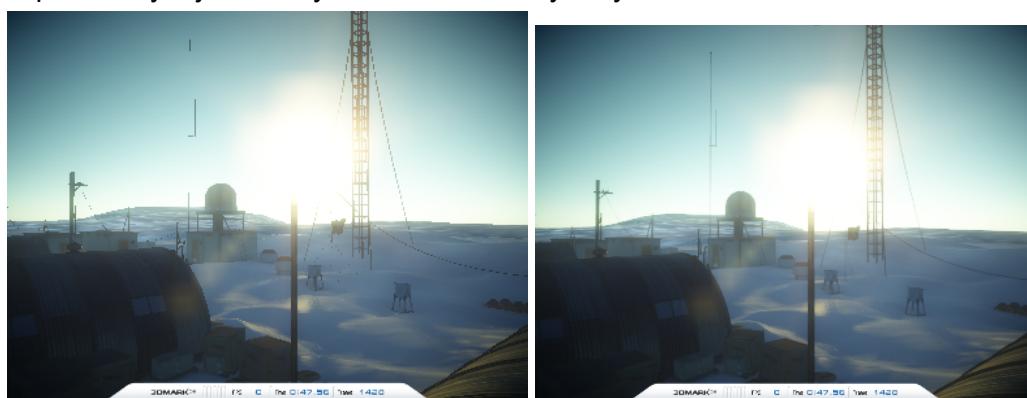
Vzorkování - sampling obrazu, popisuje krok, ve kterém převádíme definiční obor funkce obrazu. Jde o zaznamenání hodnot obrazové funkce v daných intervalech. Vzdálenost mezi vzorky je perioda vzorkovacího signálu, jejich převrácená hodnota je vzorkovací frekvence,

5.3. Anti-aliasing

Co je aliasing - alias popisuje nežádoucí artefakt, kterej vzniká v případě, že musím něco diskretizovat a mám na to malý rozlišení. Nemusí jít nutně o obrázky, může bejt např. i ve zvuku. V grafice jde převážně o problém s hranama při rasterizaci obrázu do nějakho rozlišení:



Tohle vznikne, pokud beru jako barvu střed pixelu. Problémy nejsou ale jenom zuby, může jít např. i o chybějící detaily nebo mizící tenký čáry:

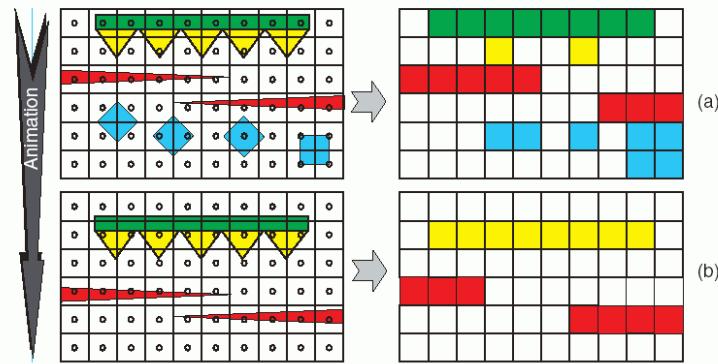


Může za to Nyquist–Shannon sampling theorem - „přesná rekonstrukce spojitého, frekvenčně omezeného signálu z jeho vzorků je možná tehdy, pokud byla vzorkovací frekvence vyšší než dvojnásobek nejvyšší harmonické složky vzorkovaného signálu.“. No, a cokoliv co má frekvenci vyšší než polovina samplovací frekvence (tj. rozlišení), tak se vykreslí blbě a je to alias.

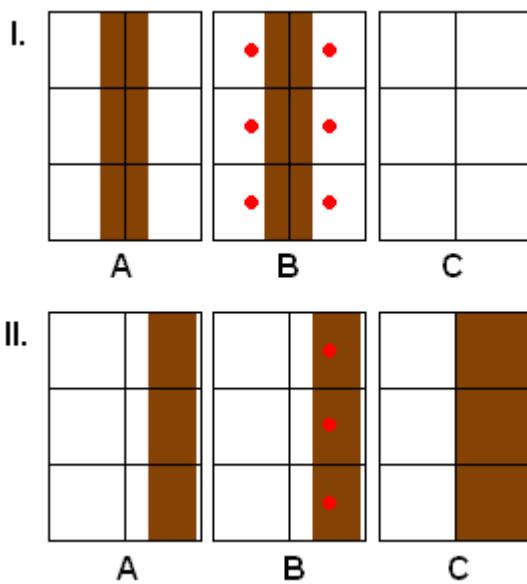
Máme několik typů, první jsou na hranách, další na plochách textur:

- **Hrany**

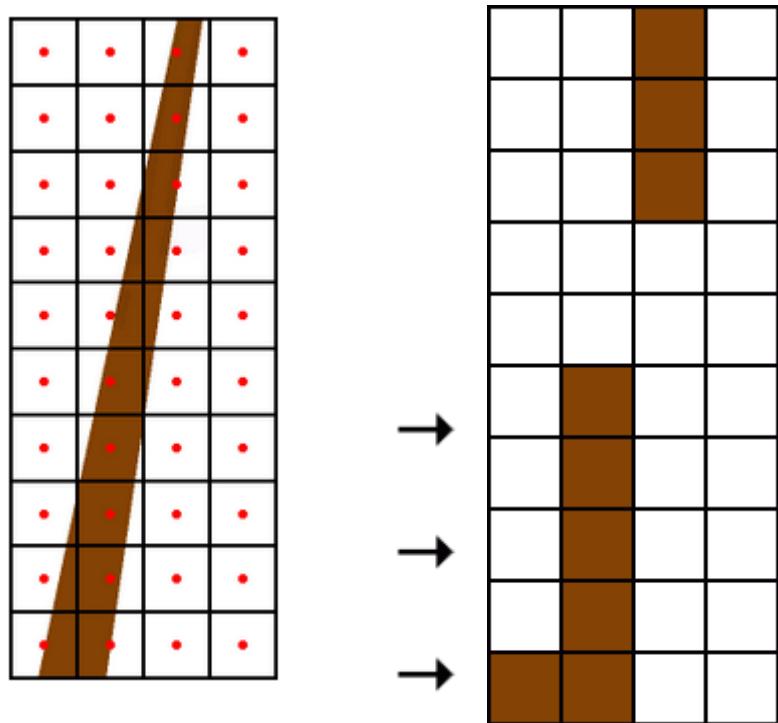
- -jaggies (=stair steps) - viz trojúhelník vejš. Hnusný hrany, za který může špatně zvolená barva pixelu
- -crawling
 - Popisuje jev, jak se jaggies hrany hejbou když hejbu kamerou. Určitě to znáte.
- -pixel popping
 - Když máme objekt co je zhruba velikosti pixelu, zobrazuje se blbě když se pohybuje. Občas jeden pixel, občas dva, občas nemusí být vidět vůbec (první modrá koule zleva):



- -(sub)pixel flickering
 - Podobná situace jako nahoře, akorát v momentě, kdy je jeden z rozměrů moc malej, tj třeba tyč v délce:



- Nespojitý čáry

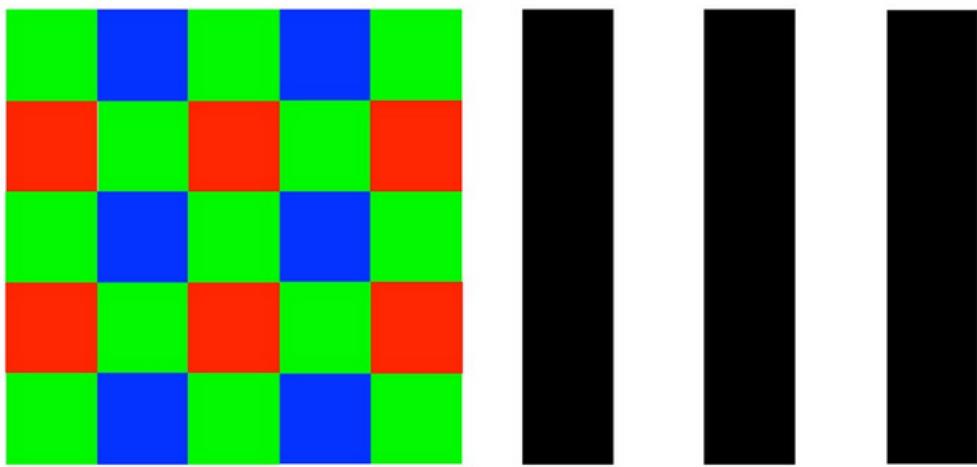


- - Plochy Textur - vlastně stejný problém jako předtím.
 - -moire
 - Když mám nějakou texturu se vzorem, tak se mi začne rozbíhat

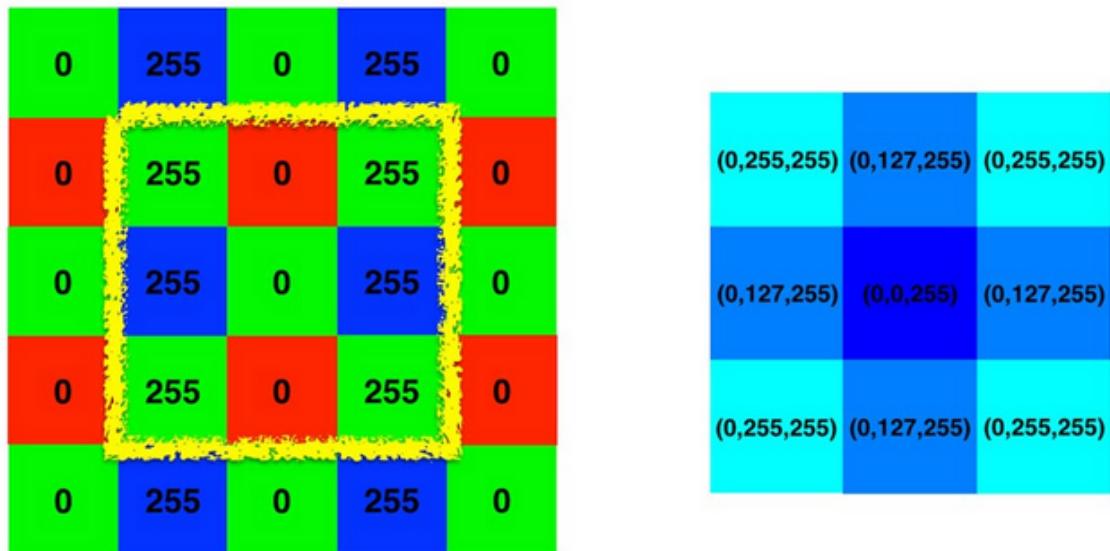


- -jaggies (=stair steps)
 - -unconnected lines

V digitální fotografii je stejný problém, kterej ještě souvisí s Bayerovým filtrem (že senzory používají v gridu pixelů 2x víc zelený, tj řádky RGRGRG a GBGBGBBG:



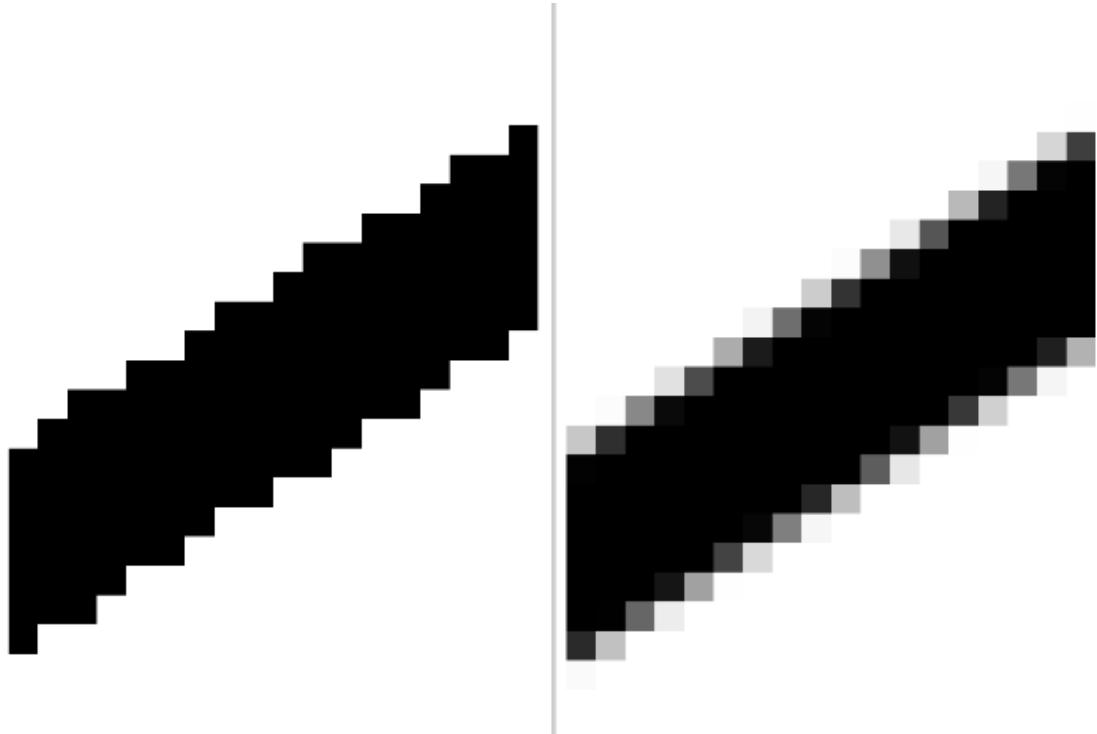
Moiré nastane v momentě, kdy se snažím takovýmhle senzorem zachytit obrázek vpravo, tj černý čáry stejně široký jako řádek pixelů. Dostanu zhruba něco takovyhleho:



Což moc nesedí, a vzniká tím Moiréé.

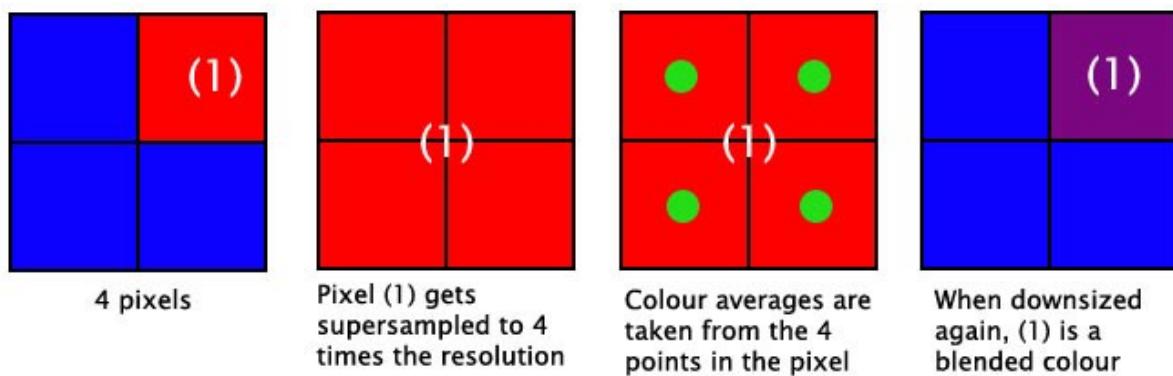
Anti-Aliasing

Anti-aliasing se pokusí řešit výše zmíněný problémy tím, že „rozmaďává“ pixely, který uplně přesně nesedí -



Algoritmů je na to několik. Prvních pár fungují tak, že zvětšují sample rate:
SSAA (Supersampling anti-aliasing)

Nejstarší metoda, prostě si vydělám obrázek výrazně větším rozlišením, a pak spočítám hodnotu barvy pixelu tím, že downsampleuju:



Multi-Sampling Anti-Aliasing (MSAA)

Vylepšený supersampling. V SSAA je problém v tom, že protože sem downsampleoval, tak se volal i pixel shader 4x navíc. MSAA tohleto řeší. Funguje skoro stejně, ale pixel shader se volá už jenom jednou per pixel, a né per subpixel (a to u všech pixelů, jejichž subpixeli dají trojuhelník co vykrelují překrývá. Musíme si teda uchovávat coverage bitmapu subpixelů, abych věděl, v kterých všech pixelech můj objekt je).

EQAA(Enhanced Quality AA) and CSAA(Coverage Sample AA)

Jedná se o Nvidií a AMDí pokus o to vylepšit MSAA tak, aby se zachoval počet samplů, ale kvalita byla o něco lepší. Dělají to v podstatě tak, že color/z sampling je drahej a dělat

nechceme, ale coverage sampling, tj jestli trojuhelník co kreslím na daném místě je, je vlastně docela cheap. EQAA a CSAA fungují stejně, jenom jedno je od Nvidie a druhý od AMD, a dělají to, že samplují coverage 4x větším rozlišením než barvu, a pak ten výsledek blendnou na základě toho, kolik přesněji ten objekt zabírá místa na daném pixelu. Tady to popisují docela rychle a srozumitelně:

<https://www.anandtech.com/show/4061/amds-radeon-hd-6970-radeon-hd-6950/10>

Další část AA algoritmů je tzv post-AA, protože jsou to postprocess filtry co zlepšují až obrázek:

MLAA(Morphological AA)

V podstatě funguje tak, že používá reconstruction filtry k tromu, aby odhadlo výsledek SSAA aniž by bylo potřeba to počítat přímo. Funguje v podstatě metodou pattern matchingu - najde hrany, a podívá se na okolí pixelu, a podle "zubů" co vidí odhadne jaké asi má směr ta čára:

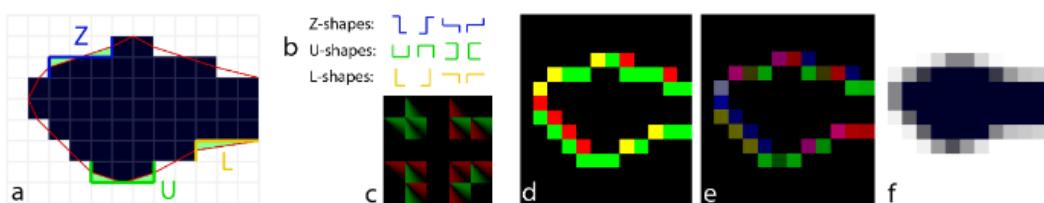


Figure 3: MLAA overview. (a) Input image, with the intended approximation outlined by red lines and the coverage areas shown in green. (b) Predefined patterns in the original algorithm [Res09]. (c) Precomputed areas texture in Jimenez's GPU implementation [JME*11]. (d) Detected edges. (e) Calculated coverage areas. (f) Final blending. Our SMAA algorithm overhauls

FXAA(Fast Approximate AA)

Je super rychlý, a funguje v podstatě jako MLAA. prostě detektne hrany a blurne je. MLAA se snaží tomu extra blurru předejít tím pattern matchingem. Popsané je docela fajn tady - https://en.wikipedia.org/wiki/Fast_approximate_anti-aliasing

SMAA (Enhanced Subpixel Morphological AntiAliasing)

<https://iryoku.com/smaa/downloads/SMAA-Enhanced-Subpixel-Morphological-Antialiasing.pdf>

V podstatě funguje stejně jako MLAA, akorát má o něco víc vzorů pro pattern recognition, používá i barvy a nejenom jas pro edge detection, a taky se dá kombinovat s multi/supersamplingem pro ještě lepší výsledky. Je to prostě taková upgraded verze.

TXAA - Temporal Anti Aliasing

Temporal Anti-Aliasing by měl bejt z definice slova Motion Blur, protože Temporal Aliasing se vážě na pohyb a nemá nic společného s hranama,

Nvidia ale vydala TXAA, který s hranama něco společného má, a blbě se zjišťuje co že zkratka znamená. Funguje to v podstatě jako kombinace výše zmíněnejch, a ještě proprietární pro Nvidii. Víc infa je tady

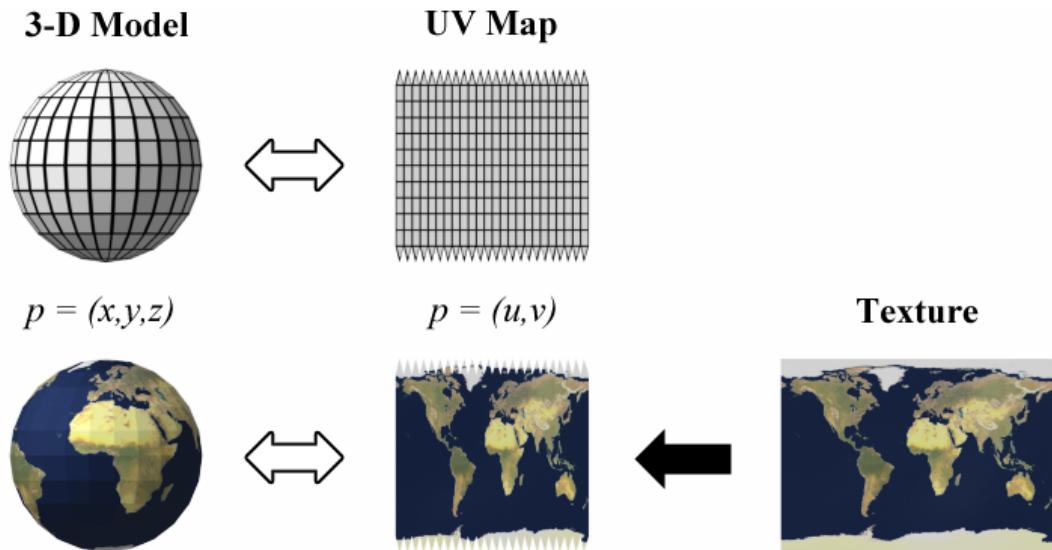
<https://www.nvidia.com/en-gb/geforce/technologies/txaa/technology/>

Pod anti-aliasing se taky dá řadit filtrování textur, který je v kapitole Textury.

5.4. Textury

Textury jsou v podstatě 1D, 2D nebo 3D pole, který se používají pro barvení a nastavování dalších vlastností objektů. Nemusí nutně jít jenom o obrázky (i když se tak nejčastěji zobrazujou), ale jejich hodnota může určovat i další věci. Říká se jim Texture Mapy.

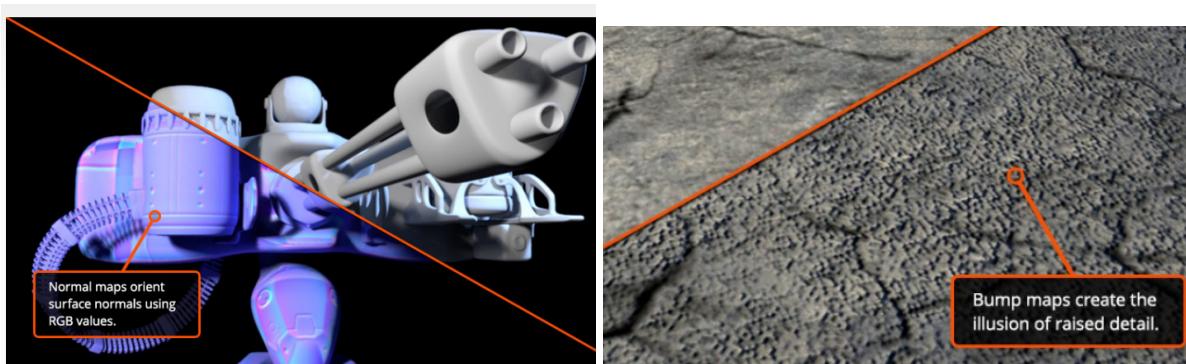
Textury se nejčastěji mapují na objekty, čehož je dosaženo UV mappingem:

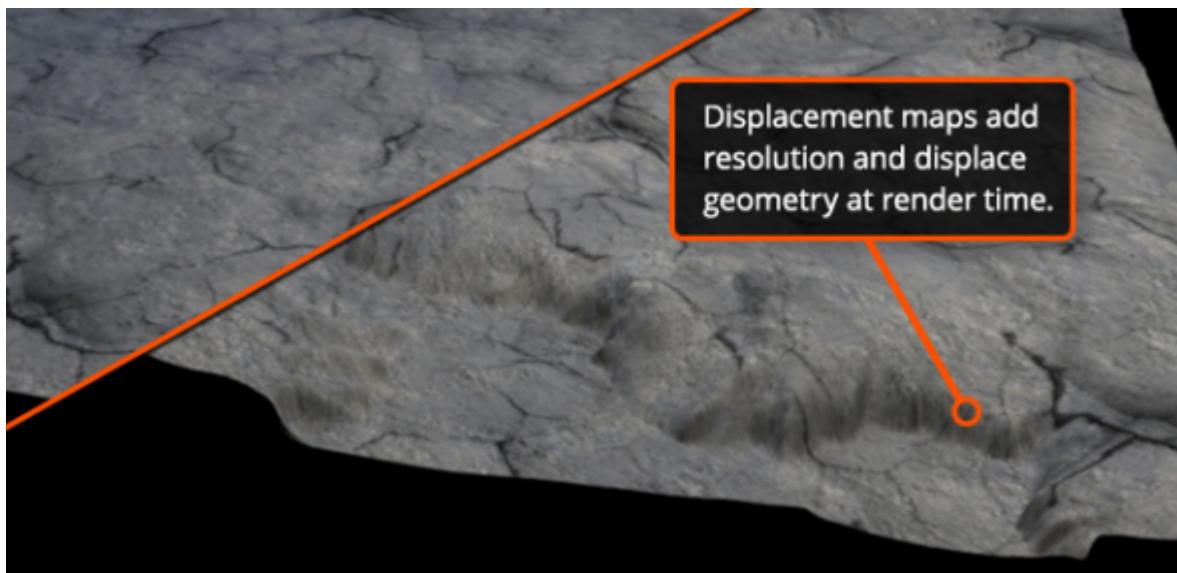


UV mapa nám definuje způsob, kterým se jednotlivý body na textuře promítají na vertextu na objektu. V podstatě jde o zobrazení z model spacu do 2D texture spacu. UV coordinates sou většinou v rozmezí $[(0,0),(1,1)]$

Textury se používají pro spoustu věcí. Například:

[height mapping](#), [bump mapping](#), [normal mapping](#), [displacement mapping](#), [reflection mapping](#), [specular mapping](#), [occlusion mapping](#). V tom případě nám hodnota textury neurčuje barvu, ale nějakoujinou vlastnost objektu v daném bodě, jako třeba jestli svítí, koeficienty pro odrážení světla, normály, displacement vertexů pro vertex shader, nastavení vlastností pro physical based rendering a podobně.





Multitexturing - většinou mám na objektu více jak jednu texturu, většinou různých typů co každá určí nějakou jinou vlastnost. Máme teda jednu co určí barvu, další co určí displacement. Moderní pipeliny mívají třeba až 10 různých textur na objektu.

Texel - texel je pixel na textuře. Nemusí nutně odpovídat tomu, co se zobrazí, třeba proto, že má textura větší rozlišení než objekt co sleduju, třeba proto, že je daleko.

Texture filtering - je právě způsob, kterým si vyberu barvu jakou bude mít zobrazený pixel na objektu, na základě jeho textury. Nemusí se mi totiž stát, že se přesně trefím do jednoho pixelu, třeba když je objekt daleko nebo pod blbým úhlem. Je tu několik problémů, a algoritmů jak vše řešit. Celkově máme dva případy - texture minification a texture magnification. Když máme texel větší než pixel na obrazovce (tj. textura je blízko) tak je to texture magnification. Pokud je ale textura daleko, tak jeden pixel obsahne více jak jeden texel, a říkáme tomu texture minification. Algoritmu je několik:

Nearest-neighbor interpolation - nejjednodušší, prostě vezmi texel kterej je nejbližší místu, kam dopadne tvůj pixel. Má to spousty artefaktů, a při minifikaci to má docela performance loss. Může se použít s mipmappingem, čímž se problém s performancem minifikace zlepší, a zbaví se trocha artefaktů.

Linear mipmap filtering - to samý jako předtím, akorát vezmu nejbližší body ze dvou nejbližších mipmap, a lineárně interpoluju mezi nima

Bilinear filtering - vezmu 4 nejbližší texely, a jejich hodnoty dám dohromady s váhou rovnou vzdálenosti k danému texelu. Takže texel co je blíž, se více projeví. Většinou se pojí s mipmapou. Vzniká ale problém v momentě, kdy se mění level detailů mipmapy, protože se pak na textuře skokově změní barva. To řeší **trilinear filtering**, kterej si bilineárně najde texel na dvou nejbližších mipmapách, a ty váženě spojí na základě vzdálenosti od mipmapy.

Tj. pokud mám jeden level mipmapy pro $Z=50$, a další pro $Z=100$, a mám pixel na $Z=60$, tak si bilineárně najde texel na obou mipmapách, a pak je seče s váhama 0.2 a 0.8.

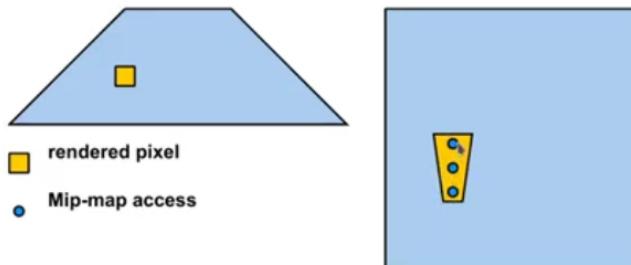
Anisotropic filtering - Anisotropic filtering je asi nejlepší způsob, který je ale docela hardwarově náročnej. Idea je taková, že pokud samplujeme plochy které sou hrozně moc pod úhlem, tak je špatně samplovat texturu čtverečkem (pixelem), protože se do textury promítne přece jinak. Vznikají proto artefakty. A to se snaží vyřešit anisotropic filtering, který sampluje texely v podstatě pod úhlem. Může si ty úhly předpočítat předem,



čemuž se říká **RIP-Mapa** - . Tady by ale třeba byl problém v tom, že to vyřeší jenom osy kolmo, a když dojde k rotaci trochu šikmo, tak už RIP mapa nepomůže. Když to předpočítaný nemám, tak si prostě sampluju tak, že si promítну pixel na texturu:

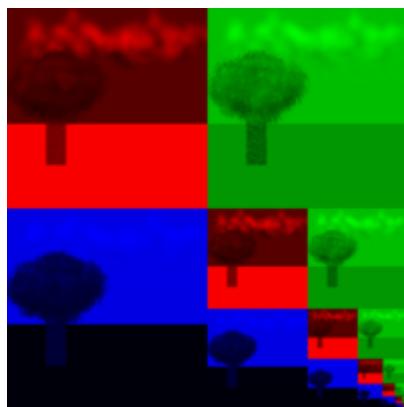
ANISOTROPIC FILTERING

- ▶ back-projected screen pixel = deformed quadrangle
- ▶ MIP-map level according the higher sub-sampling (shorter size)
- ▶ multi-sampling (averaging) along the longer side



Algoritmy na to jsou většinou proprietární, nebo to odhadují nějak magicky.

Mipmapping - není to přímo algoritmus, ale technika kterou se ukládají textury tak, aby se vyřešil problém s minifikací. Pokud je textura daleko, může se stát, že pixel zakrývá hodně texelů na textuře, což je problém, protože musíme načíst všechny jejich hodnoty pro texture filtering. Dokonce se může stát, že jeden pixel zabere celou texturu, takže musím průměrovat všechny texely, což je preformance hit. Řešením je mipmapping, kdy si v paměti vygeneruju předem texturu v menších velikostech - půlením tak dlouho, dokud se nedostanu na velikost 1 pixelu. Podle vzdálenosti od kamery potom sampluju tu menší texture poodle toho, co zrovna potřebuju. Celkově mě uložení výjde na 33% ztrátu, tj výsledná textura bude o 33% větší. Když si jí uložím po RGB složkách, hezky se to skládá do čtverce:



Texture Atlas - Protože jakákoliv změna dat co se musí tahat z disku je pro grafárnu nepříjemná, protože to prostě zpomaluje, může se používat Texture atlas. Je to v podstatě moderní obdoba Tile Map, kdy si spojím spousty textur do jedné, kterou pak mám v paměti, a tu správnou vyberu prostě jenom tak, že přidám offset tý textury co chci v momentě, kdy hledám bod na textuře.

Texture mapping unit - Hardware který pracuje s texturama. Grafarna jich má spousty, a každej zvládá jednu texturu. Má na starosti addressing textur (tj. vybrat na základě coordinate správnej texel), a texture filtering.

5.5. Změna kontrastu a jasu

Tady to můžou bejt dvě věci, buď jde o hýbání s jasem/kontrastem obrázku přes histogram, nebo jde o nastavování jasu/kontrastu na monitoru a teorie kolem toho.

Pro první případ, základní věc jak poznám, jak jsem na tom s jasem a kontrastem na obrázku, je **Histogram**.

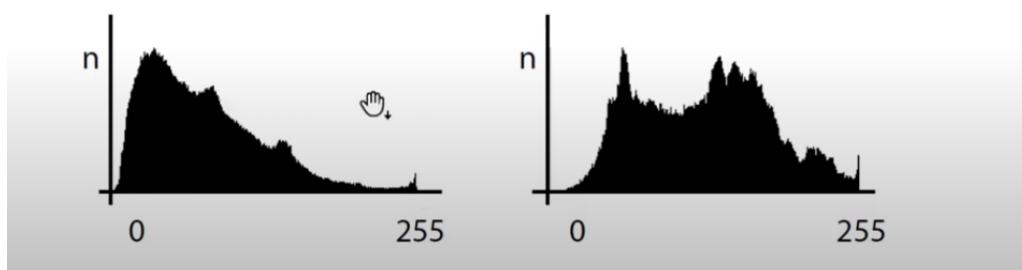
Histogram obrázku



Tabulka četností jednotlivých jasových (barevných) hodnot

– spojity případ – hustota pravděpodobnosti

Přímé použití – fotografie



Je to prostě graf, co ukazuje, kolik pixelů má jakou hodnotu jasu. Ideálně chci, aby byl graf co nejvíce rozprostřený a bez skoků, protože to značí špatný kontrast, nebo nedostatečně využitý prostor.

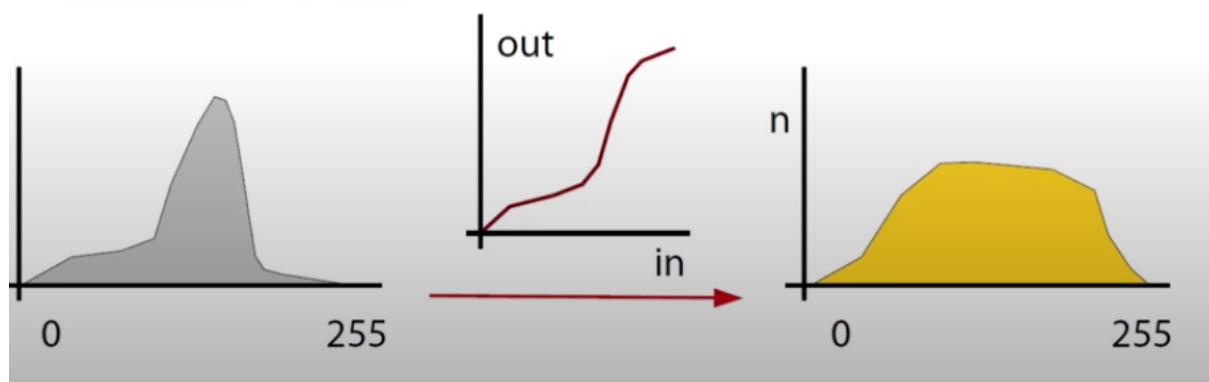
Řešit se to může transformační funkcí, která prostě vezme pixel a vyplivne jeho novou hodnotu:

Převodní funkce („transfer function“) mezi jasy na vstupu a výstupu

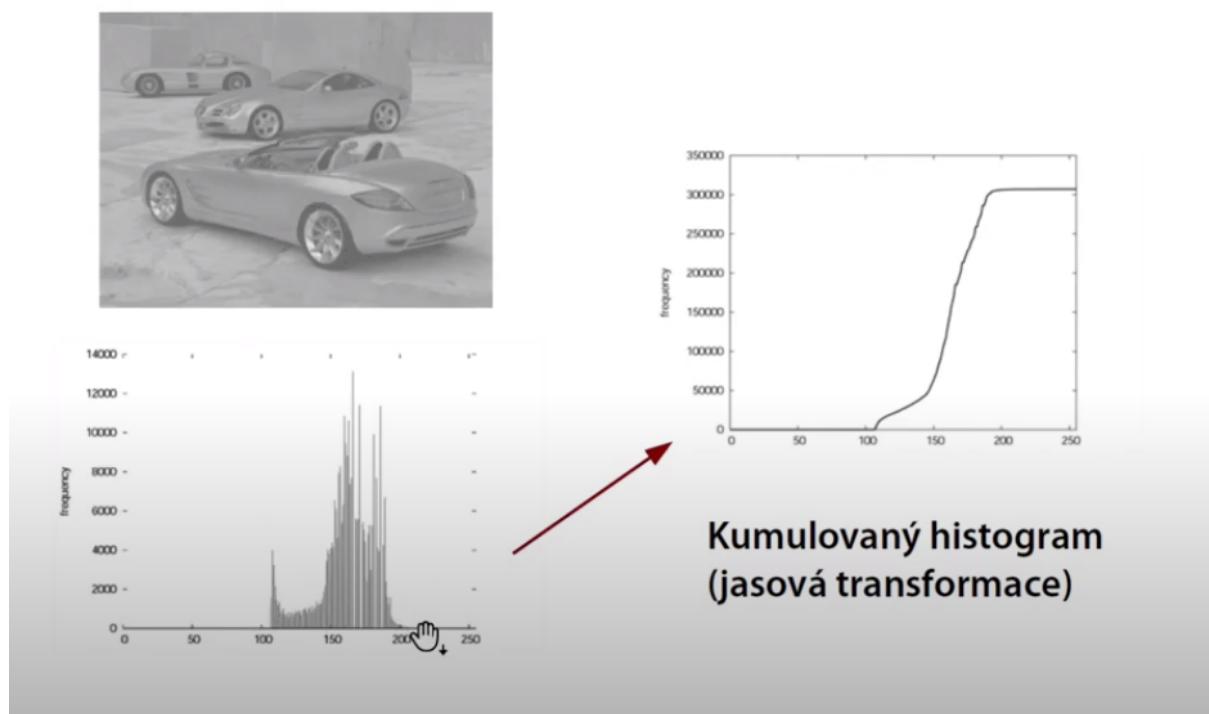
- $t: \mathbb{R} \rightarrow \mathbb{R}$ (obyčejně $[0, 1] \rightarrow [0, 1]$)

Gamma-korekce

Zvětšování kontrastu



Potřebujeme nějak ekvalizovat tu distribuci jasu, a jeden ze způsobů je například následující ekvalizace:



Postavím si kumulovaný histogram (je to histogram, který ukazuje kolik pixelů má stejnou nebo nižší hodnotu). Ten histogram pak vydělíme celkovým počtem pixelů, a tím dostanu převodní funkci kterou použiju na ekvalizaci hodnot.

Tj. například pro pixely hodnoty 110 mám hodnotu kumulovaného histogramu 50000, celkem mám 3000000 pixelů, takže transformační funkce v bodě 110 bude mít hodnotu $50000/300000$, takže se hodí na hodnotu 0.016. Tu je ještě potřeba převést nazpátek z 0-1 do 256, vynásobením 256, a tím získám ekvalizovanou hodnotu pixelu.

Druhá možnost o který může v otázce jít, je přímo gamma a vnímání jasu z výstupních zařízení.

Když jde o nějakou barvu zobrazovanou, máme dvě veličiny:

Intenzitu, což jsou přímo hustota fotonů, fyzická vlastnost

Jas, kterej určuje subjektivní vnímání člověka jak vidí "brightness"

Vztah mezi intenzitou a jasem není lineární, lidský oko to vnímá relativně, a aby se daly rozdíly pořádně popsat, je určitě potřeba nějaká logaritmická stupnice intenzit. Oko pozná zhruba 1% rozdíl.

Jde o to, že např. V noci vidím i nejslabší světlo docela v pohodě, a intenzitu má rozhodně výrazně jinou než když se podívám do slunce. Tedy tam nemůže být lineární závislost.

S gammou a gamma korekcí je dost nedorozumění, a vznikla v podstatě nuceně historicky - na CRT monitorech se vykreslovalo katodovým dělem, co rozsvěcelo pixely. A vzorek intenzity mělo takovějhle:

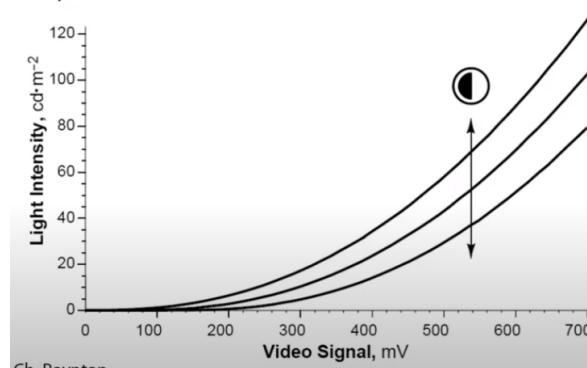
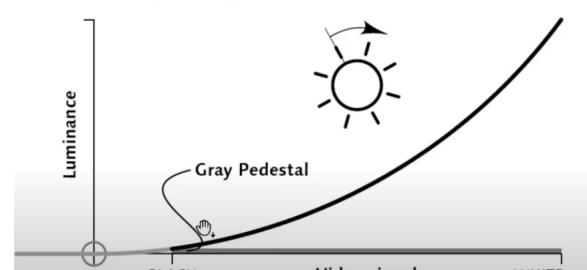
$$I = K(V + \varepsilon)^\gamma$$



- » V – napětí přiváděné do CRT (hodnota pixelu)
- » K – proměnná – ovládací prvek kontrast („picture“)
- » ε – proměnná – ovládací prvek „jas“ („black level“)
- » γ – konstanta – gamma exponent (2.35 až 2.55)



Posunutí vstupního argumentu ε



Brightness určuje posunutí startu. Tím, že to trochu šoupnu, může zamezit tomu aby úplně tmavé barvy byly moc tmavé, protože ta funkce stoupá na začátku docela pomalu.

Kontrast určuje multiplikativní koeficient, jak rychle to stoupá.

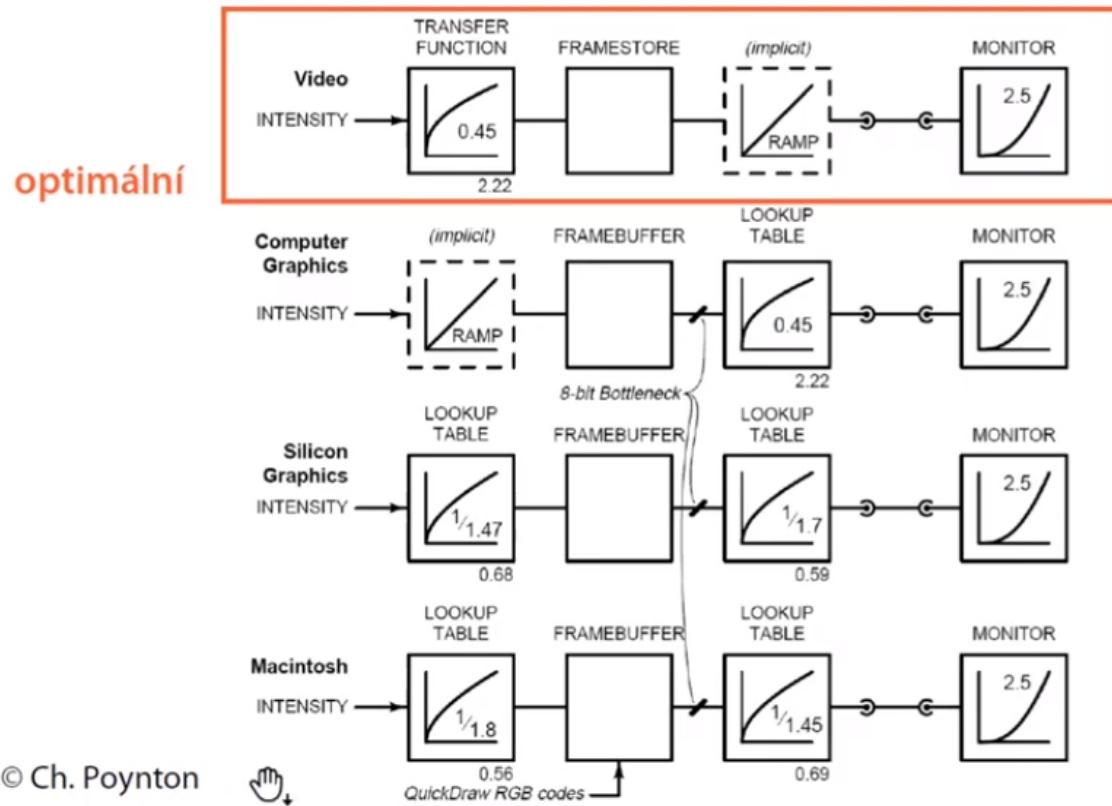
Hlavní problém s gama korekcí je ale ten, že vzniklo zmatení v tom, jak a kdy jí provádět a jak ukládat barvy. Například při generování barev a scén je s tím třeba počítat, protože pak by monitor a nelineární zobrazovací funkce monitoru způsobila, že by barvy vypadaly zkresleně.

Nejlíp to vyřešili hardwareáři od videa, který prostě data na videopásky ukládají už konvertovaný - před uložením se provede kompenzační korekce (tj. použije se na to opačná transformační funkce než je gama monitoru), a monitor to pak vykreslí správně.

(První řádek níž),

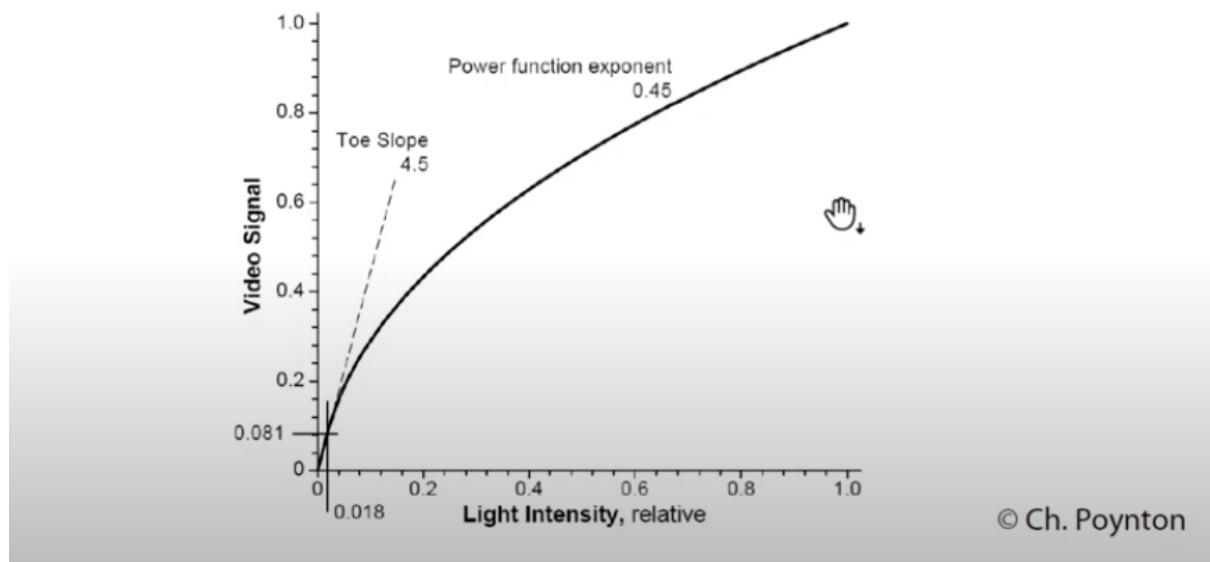
O něco hůř už to vyřešili v CG, protože tam se data do frame bufferu ukládají jak sou, a o konverzi pro gammu se stará lookup tabulka, většinou na GPU těsně před vykreslením.

No a pak tu je ještě Silicon Graphics a Macintosh, který se prostě rozhodli, že z nějakého důvodu udělají půlku předtím, a půlku potom.



Transformace intenzity před uložením do RAM

~ inverzní k nelinearitě CRT monitoru



© Ch. Poynton

Na LCD displayích už gamma problém nebyla, většiny to ale simulují kvůli zpětný kompatibilitě. Zároveň taky ty konverzní matice a převody už nejsou pevně daný, a jde s nimi hýbat softwarově.

5.6. Kompozice poloprůhledných obrázků

Přidám 4. Alpha kanál, který určuje průhlednost.

Procentuální pokrytí pixelu neprůsvitnou barvou

- doplněk průhlednosti
- $\alpha = 0$... zcela průhledný pixel (nemá vliv na výsledek)
- $\alpha = 1$... neprůhledný pixel („nic za ním neprosvítá“)

Ukládání hodnoty α v každém pixelu

- často celočíselná reprezentace ($0 \div 255$)
- čtveřice $[R, G, B, \alpha]$
- ještě častější reprezentace $[R\alpha, G\alpha, B\alpha, \alpha]$

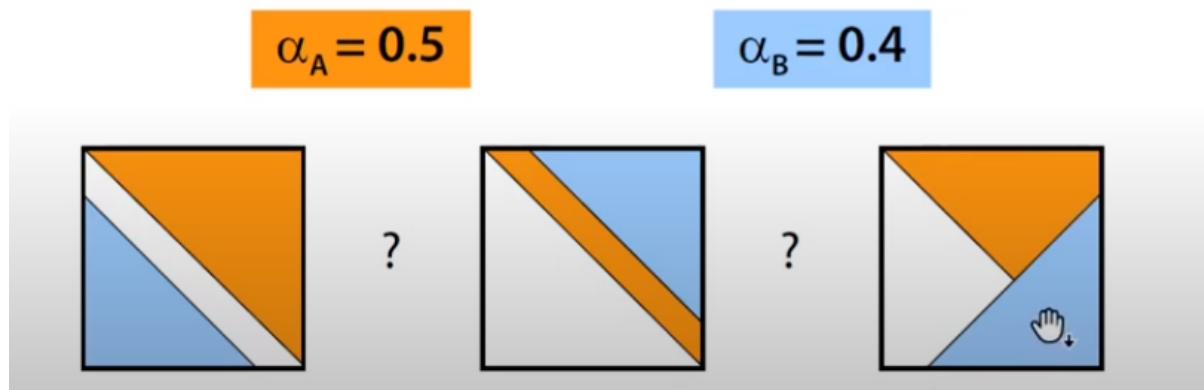
Častěji se preferuje druhý formát, kdy se každá barva vynásobí alphou už při uložení.

Největší problém je, když potřebuju přijít na to, jak skládat pixel v případě, že mi přes něj kouká více objektů.

Dva skládané pixely $[A, \alpha_A]$ resp. $[B, \alpha_B]$

- potřebuji určit výslednou hodnotu $[C, \alpha_C]$

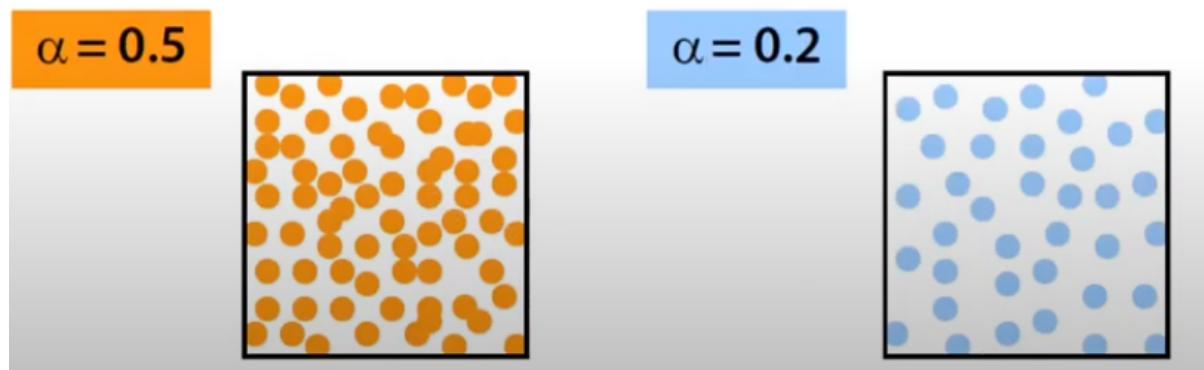
? model pro skládání pixelů ?



Mám problém v tom, že prostě nevím, v jaký jsem situaci - slajd nahoře ukazuje, co může nastat - buď jsou obě barvy disjunktní, nebo je jedna pod druhou, nebo jsou kolmý. Protože to nevím, tak se většinou předpokládá ten třetí případ, který je aritmeticky nejuniverzálnější - dá se představit tak, že prostě random poplivu pixel tou barvou podle jeho průhlednosti:

Pixel $[A, \alpha_A]$ je náhodně pokryt barvou A
s rovnoměrně rozloženou pravděpodobností α_A

- skládání geometricky nezávislých tvarů
- vyhovuje ve většině případů



Další co je potřeba vyřešit je, kolik existuje různých možných případů kompozic obrázků:

oblast	plocha	vybarvení
1	nic	$(1 - \alpha_A)(1 - \alpha_B)$
2	A	$\alpha_A(1 - \alpha_B)$
3	B	$\alpha_B(1 - \alpha_A)$
4	A i B	$\alpha_A\alpha_B$



Mám celkem 4 oblasti co potřebuju rozhodnout, podle toho kolik barev na nich může být.
Plochy 2-4 mají víc možností jak je vybarvit, z čehož celkem získám 12 možností.

Skládání dvou pixelů: Mám pixel

$[A, \alpha_A]$

a

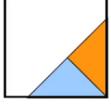
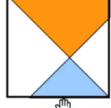
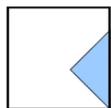
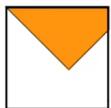
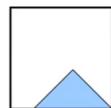
$[B, \alpha_B]$

. V realitě se mohou různě překrývat, ale mám pouze číslo alfa. Model pokrytí pixelu, kde je pixel pokryt barvou A s s rovnoměrně rozloženou pstí alfa_A. Nezáleží tak na skutečném tvaru a vyhovuje ve většině případů. Překrývání dvou pixelů v realitě má 4 oblastí:

- Oblast bez A i bez B
- Oblast pouze s A
- Oblast pouze s B
- Oblast s A i B

V závislosti na způsobu překryvu je 12 možností, jak je zkombinovat

A přes B A over B (0,A,B,A) 1 $(1 - \alpha_A)$	B přes A B over A (0,A,B,B) 1 $(1 - \alpha_B)$	A v B A in B (0,0,0,A) α_B 0	barvy nic clear F_A F_B	(0,0,0,0) 0 0	(0,A,0,A) 1 0 1

					
A na povrchu B A atop B	B na povrchu A B atop A	A xor B	B v A B in A	A mimo B A out B	B mimo A B out A
(0,0,B,A)	(0,A,0,B)	(0,A,B,0)	(0,0,0,B)	(0,A,0,0)	(0,0,B,0)
α_B	$(1 - \alpha_B)$	$(1 - \alpha_B)$	0	$(1 - \alpha_B)$	0
$(1 - \alpha_A)$	α_A	$(1 - \alpha_A)$	α_A	0	$(1 - \alpha_A)$

F_A a F_B je kolik procent plochy, která by měla mít barvu A/B se má použít při směsi (ne poměr na velikost pixelu, ale na plochu A/B pixelu).

Pokud chci teda sloučit dva pixely, vezmu na to následující vzorec podle toho, co za mod/operaci slučování chci dělat.

$$[F_A R_A + F_B R_B, F_A G_A + F_B G_B, F_A B_A + F_B B_B, F_A \alpha_A + F_B \alpha_B]$$

$$\text{darken} (A, \rho) = [\rho R_A, \rho G_A, \rho B_A, \alpha_A]$$

$$\text{fade} (A, \delta) = [\delta R_A, \delta G_A, \delta B_A, \delta \alpha_A]$$

$$\text{opaque} (A, \omega) = [R_A, G_A, B_A, \omega \alpha_A]$$

5.7. Principy komprese rastrové 2D grafiky

Ze základu je cílem komprese to, se nějak zbavit redundantních dat který jsou navíc a nejdou moc důležitý. Matematicky je popsáno tohle:

Komprese dat [\[editovat zdroj\]](#)

- Máme kódovanou abecedu $S = x_1, x_2, \dots, x_n$
- prst. výskytu symbolu x_i je p_i
- kód symbolu x_i má délku d_i
- zpráva (kódovaný řetězec) $X = x_{i_1}, x_{i_2}, \dots, x_{i_k}$
- entropie (informační hodnota) symbolu x_i je $E_i = -\log_2 p_i$ bitů
- průměrná entropie symbolu je $AE = \sum_{i=1}^n p_i E_i$ bitů
- entropie zprávy je $E(X) = \sum_{j=0}^k p_{i_j} E_{i_j} = -\sum_{j=0}^k p_{i_j} \log_2 p_{i_j}$ bitů
- délka zprávy $L(X) = \sum_{j=1}^k d_{i_j}$ bitů
- redundance zprávy $R(X) = L(X) - E(X)$ bitů
- Výběrová střední kvadratická odchylka $RMS^2 = \frac{1}{NM} \sum \sum (u_{i,j} - u'_{i,j})^2$ (původním obrazem vs rekonstrukce)
- Peak Signal to Noise ratio $PSNR = 10 \log_{10} \frac{P^2}{RMS^2} dB$, kde P^2 je interval hodnot originálu, např 255^2

Je spousty různých formátů rastrový grafiky, a liší se převážně tímhle:

Formát uložení barev

- barevná paleta, šedá škála, „true-color“, kanál „ α “

Komprese

- bezzáratová / [zzáratová](#)
- RLE: TGA, BMP; LZ*: PNG, GIF, TIFF; [JPEG](#): JFIF, TIFF

Rozklad obrázku

- prokládané/progresivní režimy (PNG, GIF, TGA, JFIF ...)

Negrafické info – metadata

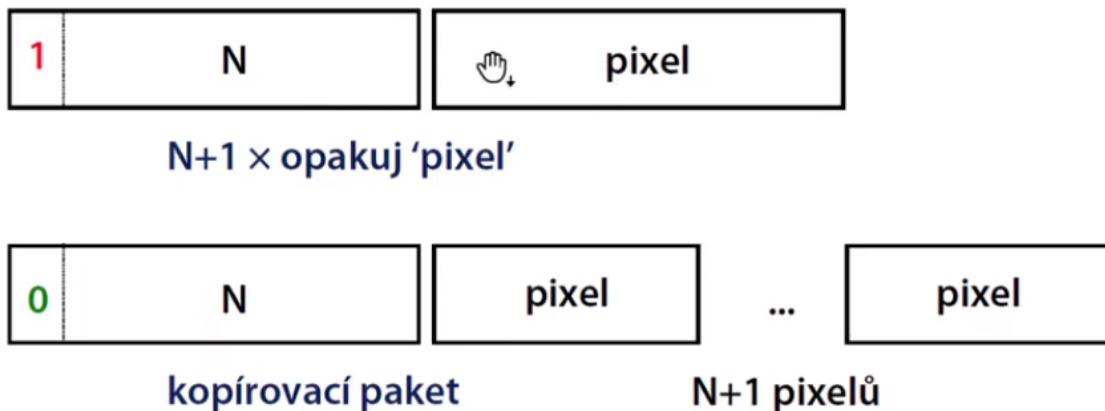
- všechny moderní formáty (TIFF, PNG, GIF, JFIF ...)

Rozklad obrázku/prokládání značí, jestli je možné nahrávat prokládaně, tj stylem “nejdřív nahraju ultra malý rozlišení obrázku, pak postupně zlepšuju. Používalo se při starších dobách mega pomalých internetů, teď už není moc potřeba.

Komprese

Run-length Encoding (RLE) komprese, která se používá v BMP a TGA je bezzáratová, a docela jednoduchá.

RLE komprese v TGA



Maximální délka paketu je 128 pixelů

- prodloužení je v nejhorším případě 0.8 % délky souboru

Začnu tím, že mám kontrolní byte, co mi určí v prvním bytu typ, a pak počet dalších pixelů. Pokud je typ 1, tak vezmu další pixel a opakuji ho tolikrát, kolik je v počtu. Pokud je typ 0, tak nic nekopíruju, a přečtu tolik dalších pixelů. Protože mám jenom 7 bitů, max počet je 128. Po tom co projedu N pixelů, si zase přečtu další bit, který bude kontrolní, a mám typ a počet dalšího bloku.

Další používaná komprese je LZW - Lempel-Ziv-Welch.

LZW komprese (Lempel-Ziv-Welch)

Slovníková kompresní metoda (1984)

- vychází z LZ78 (Lempel-Ziv, 1978)
- slovník obsahuje přiřazení „fráze → kód“
- fráze = posloupnost pixelů („znaků abecedy“)
- kód = n-bitové číslo ($3 \leq n \leq 12$)



V průběhu kódování se mění

- **slovník**
 - » adaptivní přizpůsobení kódovaným datům
- **délka kódového slova „n“**
 - » zvětšuje se po jedné až do 12

LZW komprese je slovníková komprese založená na LZ78 metodě, která se používala ke komprimování souborů.

Idea algoritmu je taková, že postupně stavím slovník frází (co jsou složený z několika pixelů) co sem potkal, a nahrazuji je identifikátorem fráze ve slovníku.

Výhodou LZW oproti LZ78 je, že komprimovat trvá zhruba stejně dlouho jako číst, komprese totiž probíhá jedním průchodem.

Idea algoritmu je následující:

1. inicializace
 - do slovníku všechny jednopixelové (jednoznakové) fráze
 - $\text{Act} := []$ (prázdná posloupnost)
2. přečti další pixel ze vstupu do K
3. je fráze „ $\text{Act} + \text{K}$ “ ve slovníku?
 - Ano: $\text{Act} := \text{Act} + \text{K}$
 - Ne: zapiš na výstup kód fráze Act
 - přidej $\text{Act} + \text{K}$ do slovníku
 - $\text{Act} := \text{K}$
4. dokud neskončí vstup, opakuj kroky 2. a 3.
5. zapiš na výstup kód fráze Act

Prostě - jedu po znacích od začátku souboru, a postupně si stavím Act , tedy frázi. Vždycky se podívám na další znak. Pokud je ta kombinace znaků ve slovníku, tak se podívám na další, jesli tam náhodu nemám delší frázi. Jamile narazím na něco co ještě nemám, tak vypíšu do výstupu kód tý co jsem zatím našel, a přidám do slovníku novou, která je o ten jeden pixel delší. Takhle pokračuju až na konec souboru.

Hlavní výhoda je, že si nemusím slovník nikam zapisovat - sem schopnej si ho postavit úplně stejně i při dekomprezii prostě tak, že použiju v podstatě stejný algoritmus jako pro komprezii. Ten dictionary si stavím znova, ze znaků co potkám na začátku souboru (kde byl ještě slovník malej), a v podstatě v momentě kdy potkám první kompresnutý slovo, tak už ho mám postavený ve slovníku z těch předechozích kroků.

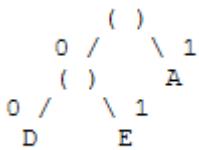
Algoritmus i s příkladem průchodu je ukázanej tady: <https://youtu.be/j2HSd3HCpDs?t=310>

PNG používá komprezi DEFLATE, která je založená na LZ77 spolu s Huffman kódem. Hrozně hezky je popsáný tady: <https://zlib.net/feldspar.html>

Huffmanův kód je prefixový kód, kterej se staví na základě četnosti znaků/prvků v celém dokumentu. Staví se následovně:

- Udělám si tabulku četnosti jednotlivých prvků (třeba písmen, nebo pixelů)
- Seřadím jí podle četnosti. Vezmu dva prvky co jsou nejméně krát v souboru.
- Ty dva prvky spojím jako listy stromu.
 - Prvek s nižší četností dám do pravého childu. Pokud mají stejně, berou se podle abecedy/nějakého řazení, aby šel strom rekonstruovat z tabulky jenom jeden.
 - Prvek vlevo má hodnotu cesty 0, prvek vpravo 1.
- Nově vytvořený strom přidám jako další prvek do seznamu, a sečtu jejich četnosti.

- Takhle pokračuju, dokud nemám postavený celej strom. Kód jednotlivýho prvku potom získám tak, že dám zasebe hodnoty na cestě od rootu stromu:



- - A má kód 1, D má kód 00, E 01.
- Zároveň by takhe vypadalo, kdybych stavěl tabulku pro prvky A s četností 40, D četnosí 8 a E četností 8.

Tím získám komprimovaný kódy pro jednotlivý prvky, a protože vždycky mají unikátní prefix a strom mám, tak tím můžu komprimovat soubor. Třeba příklad:

AAEAAAADD, tj 64 bitů zkompresuji na 1 1 01 1 1 1 00 00 - 11 bitů.

Další částí je **LZ77** komprese:

Ta je zase obře popsaná tady

<https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c09>

a teda i na tom linku vejš. Příklad na druhém linku je ale lepší.

LZ77 komprese (Lempel-Ziv)



Bezeztrátová slovníková kompresní metoda

- s posuvným oknem (typicky desítky KB)
- data v okně bývají zpracována do vhodné datové struktury pro velmi rychlé vyhledávání
- nesymetrický algoritmus (dekódování je rychlejší)

Kóduje se sekvence dat

- fráze = posloupnost znaků (pixelů)

Kódem je trojice [offset, délka, znak]

- offset = relativní poloha začátku fráze (v okně)
- délka = délka fráze v pixelech
- znak = pixel, který následuje za frází

Komprese probíhá zhruba takhle:

- Komprese začíná na začátku, popíšu ale rovnou obecně příklad uprostřed souboru:
- Komprimuju znak na pozici n v souboru.
 - Udržuju si lookahead buffer. Do něj přidám znak na pozici n+1
 - Podívám se, jestli se posloupnost znaků v lookahead bufferu nachází někde mezi znakama co už sem prošel (tj, mezi znakama na pozicích 0 - n).

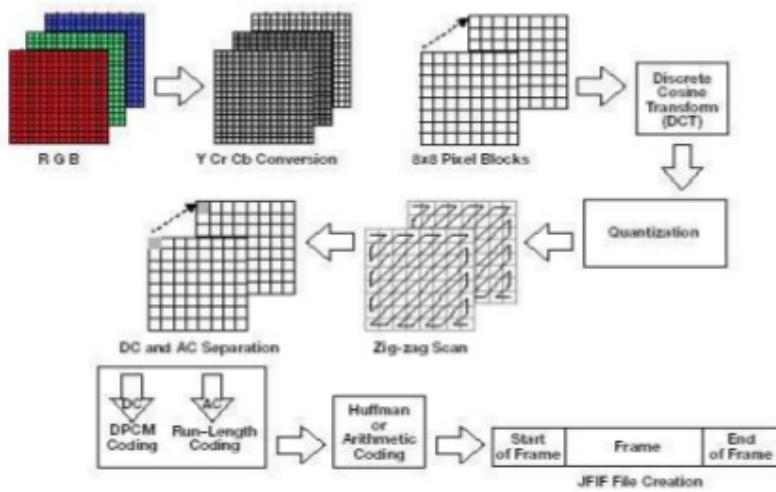
- c. Pokud ho najdu (tj už sem to co je v lookaheadu někde použil), tak do lookahead bufferu přidám další znak co následuje., a znova se podívám na krok b.
- d. Takhle pokračuju dokud se mi nestane, že v lookahead je posloupnost kterou sem ještě nepoužil.
- e. Do komprimovaného souboru zapíšu trojici znaků: [offset, délka, znak]
 - Offset mi určuje, jak zpátky musím jít, abych našel tu posloupnost
 - Délka je délka lookahead bufferu, bez posledního znaku (tj délka poslední posloupnosti co sem našel)
 - Znak je poslední znak lookahead bufferu, tj první co sem nenašel.
 - Je tam proto, aby šlo vypsat znaky bez matche
 - AABC se zakoduje jako (0,0,A)(1,1,B)(0,0,C)
- f. Posunu pozici znaku co komprimuju o velikost lookaheadu.

Takhle funguje v teorii - prostě se dívám, jestli sem už neviděl posloupnost znaků co následuje někde předtím, najdu tu nejdělsí co sem viděl, a místo ní napíšu instrukci [podívej se o offset znaků zpátky, přečti dalších délka znaků, potom vypiš znak]. Časová komplexita je ale blbá - musel by kontrolovat všechny předchozí znaky a hledat match, tak se z toho důvodu zavádí posuvné okénko - tj buffer o kolik se maximálně koukám zpátky. To samý se občas dělá pro lookahead buffer,

DEFLATE - a celý dohromady se to pojí tak, že nejdřív provedu LZ77, a pak jí hodím do Huffman kódování. Je několik různých módů jak, jestli použiju předdefinovaný huffman trees který specifikuje Deflate (a ušetřím tak místo), nebo jestli si zapíšu vlastní.

Další je **JPEG** komprese, která se používá v JFIF file formátu.

JPEG compression pipeline



Je založená na Diskrétní Cosine Transformaci, a na tom, že lidi moc nevnímaj drobný rozdíly změn v barvách, a všimaj si jenom těch větších.

Funguje zhruba takhle:

- Převedu si obrázek z RGB do YCbCr

- Y Cb Cr je kódování barev co používá televize - Y je jas (celková luminosita), Cb je podíl modrý, Cr je podíl červený. Z těch tří si dopočítám zelenou.
 - Je to proto, že lidi vnímají změny jasu výrazně více než barev. Můžu potom v rámci komprese snížit rozlišení Cb a Cr klidně 4x, a uchovat jenom Y (tj mít 4 Y pro jedno Cb Cr)
- Rozdělím si ho na 8x8 pixel bloky
- Provedu diskrétní kosinovou transformaci.
 - V podstatě jde o to, popsat koeficienty jak složit 64 sinusovek tak, aby mi vyšel celkového obrázek.

6.1917	-0.3411	1.2418	0.1492	0.1583	0.2742	-0.0724	0.0561
0.2205	0.0214	0.4503	0.3947	-0.7846	-0.4391	0.1001	-0.2554
1.0423	0.2214	-1.0017	-0.2720	0.0789	-0.1952	0.2801	0.4713
-0.2340	-0.0392	-0.2617	-0.2866	0.6351	0.3501	-0.1433	0.3550
0.2750	0.0226	0.1229	0.2163	-0.2583	-0.0742	-0.2042	-0.5906
0.0553	0.0428	-0.4721	-0.2905	0.4745	0.2875	-0.0284	-0.1611
0.3169	0.0541	-0.1033	-0.0225	-0.0056	0.1017	-0.1650	-0.1500
-0.2570	-0.0627	0.1960	0.0644	-0.1436	-0.1034	0.1887	0.1444

- - Tohle jsou ty sinusovky co skládám. Výsledkem je tabulka koeficientů, co určuje kolikrát vynásobit jednotlivé z bloků v tomhle obrázku, a pak všechny sečtu přes sebe a tím dostanu výslednou podobu jedný ze složek.

- Máme teda tabulku 8*8 číselných hodnot, co určují koeficienty co víceméně lossless popisují hodnotu pixelů.
- Další krok je Kvantizace, ve který probíhá základ komprese - sinusovky co jsou vlevo dole (tj mají větší frekvence) nejsou moc vidět, a nevadí, když se jich zbavíme. Proto máme kvantizační tabulku, která určuje, jakým číslem mám vydělit jednotlivý

koeficienty, a zároveň zaokrouhlím, na integer, abych dostal co nejvíce nul.

Step 4: Quantization

- This is where the data is compressed in great extent. After performing DCT, we apply quantization on it to remove the high frequency components. This is the real data compression part of jpeg.
- For quantizing the DCT transformed data, jpeg use a predefined quantization table. The amount of compression is dependent on the choice of quantization table.
- This quantization is non reversible meaning that we can't recover the original data without loss.

$\begin{array}{ccccccccc} 313 & 56 & -27 & 18 & 78 & -69 & 27 & -27 \\ -38 & -27 & 13 & 44 & 32 & -1 & -24 & -10 \\ -29 & -17 & 10 & 33 & 21 & -6 & -16 & -9 \\ -10 & -8 & 9 & 17 & 9 & -10 & -13 & 1 \\ -6 & 1 & 6 & 4 & -3 & -7 & -5 & 5 \\ 2 & 3 & 0 & -3 & -7 & -4 & 0 & 3 \\ 4 & 4 & -1 & -2 & -9 & 0 & 2 & 4 \\ 3 & 1 & 0 & -4 & -2 & -1 & 3 & 1 \end{array}$	$\begin{array}{ccccccccc} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 35 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 36 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{array}$	$+$	$\begin{array}{ccccccccc} 20 & 5 & -3 & 1 & 3 & -2 & 1 & 0 \\ -3 & -2 & 1 & 2 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$	$=$	Result		
8 x 8 DCT Terms	Quantization table (Matrix)						

FIGURE 27.13
JPEG quantization tables. These are two example quantization tables that might be used during compression. Each value in the DCT spectrum is divided by the corresponding value in the quantization table, and the result rounded to the nearest integer.

10

- Tím pádem mi zůstane hrozně moc nul v dolních částech, kde v naprostý většinově jsou minimální koeficienty který jsem ještě dělil obrovským číslem.
- Použiju na to huffman encoding, a hodnoty beru zigzag od začátku, takže mi tam komprimuje úplně obrovské řetězec nul.

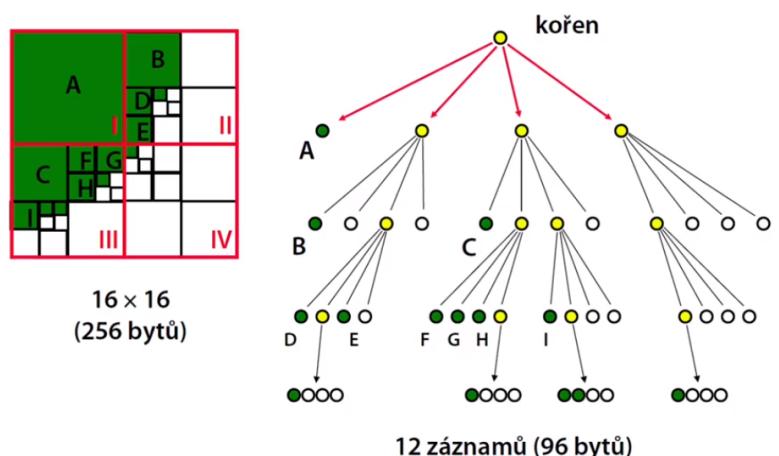
Naprosto skvěle to vysvětluje tohle video:

<https://www.youtube.com/watch?v=Q2aEzeMDHMA>

V další části jsou kódování, který jsou dobrý pro práci s bitovou maskou - pokud chci třeba průnik, nebo doplněk dvou tvarů, tak je dobrý mít je uložený takhle, protože pak můžu dobře počítat:

Quadtree.

Kvadrantový strom („quadtree“)



Dělím obrázek na čtvrtiny (tj mám strom co má 4 childy), a podívám se, jestli náhodou ty childi nemají stejnou hodnotu. Pokud mají, tak si jí zapíšu. Pokud ne, tak rekurzivně dělím na menší a menší, dokud nemám všechny listy určený. Funguje jenom na N^N obrázky. Jeho zásadní výhoda je v tom, že některý bitový operace se dají provádět na něm, a tím se hrozně moc zrychlit.

Lepší možnost může být začít na čtvercích 2x2, a stavět ten strom od spoda a spojovat stejný pixely. Pak čtu každej pixel jenom jednou, místo toho abych četl každej několikrát.

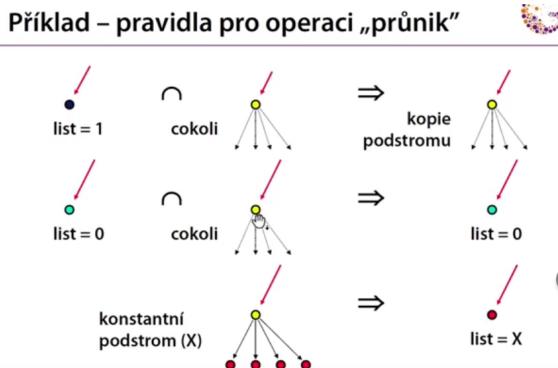
Hybridní implementace - hlídám si, jestli podstrom quadtree není větší než velikost syrové bitmapy té části, tak si prostě uložím binárně bitmapu do listu a dál neřeším.

Když na ně provádím množinový operace, tak se dá pracovat relativně rekurzivně, tj prostě jdu po jednotlivých větvích. Zároveň se ale může stát, že jeden strom jde v nějakém směru hlloubš, a ten další tam má list - tím pádem ale vím, že ten s listem má dál pro porovnání furt stejnou hodnotu. Taky mi to zjednoduší některý operace.

Abych si zdefinoval nějakou booleovskou operaci na quadtree, stačí když definuju co dělat ve 4 případech, když porovnávám podstrom a list:

- List s hodnout 1 OP podstrom
- Podstrom OP List s hodnotou 1
- List s hodnotou 0 OP podstrom
- Podstrom OP List s hodnotou 0

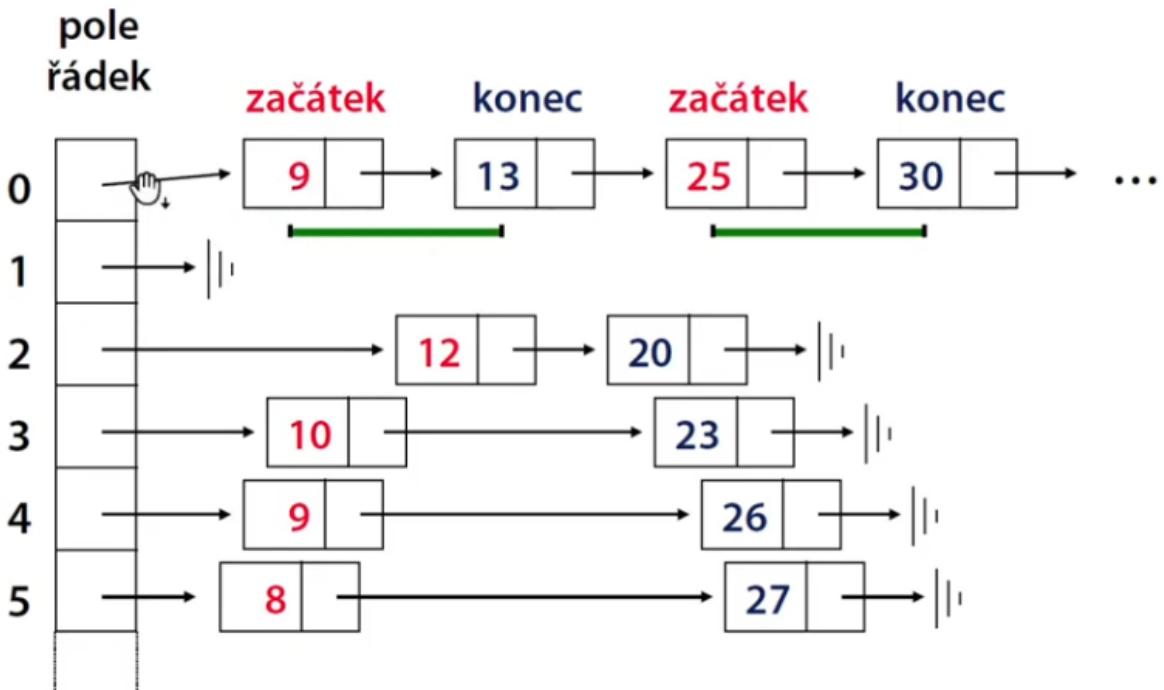
Jakmile si tohle definuju, tak potom provést operaci není problém. Třeba průnik: (Je symetrická operace, takže je jedno jestli je list vlevo nebo vpravo):



Pak tu máme **X-Transition List (řádkový seznam změn)**:

Ukládám si data po řádcích, a pamatuju si jenom body změn. (Funguje teda pro bitovou masku jenom, tj 0 a 1).

Každej řádek je jedno pole, který obsahuje čísla ve kterech se mění hodnota. Tj, když mám třeba 0 0 0 0 1 1 0 0 0 1, tak dostanu zakodováno jako pole [5,7,10]. Protože na 5 pozicy začíná řetězec 1, pak na 7 zase nuly, a na 10 zase jedničky.



Tady bych měl 9×0 , pak 4×1 , pak $12 \times 1 \dots$ na prvním řádku.

Dobrě se v tom dělají množinové operace, stačí totiž tu operaci udělat po řádcích na těc dvou zeznamech, a mernout je dohromady.

5.8. Komprese videosignálu

Nejjednodušší mám Motion JPEG - prostě slideshow JPEG obrázků. Malá komprese - jenom cca 16:1, což na videa moc nestačí. Používá se ale ve videoeditingu profi, protože každý frame má data v bitstreamu, a nic se neztratí a můžeš precizně provádět frame-by-frame edit.

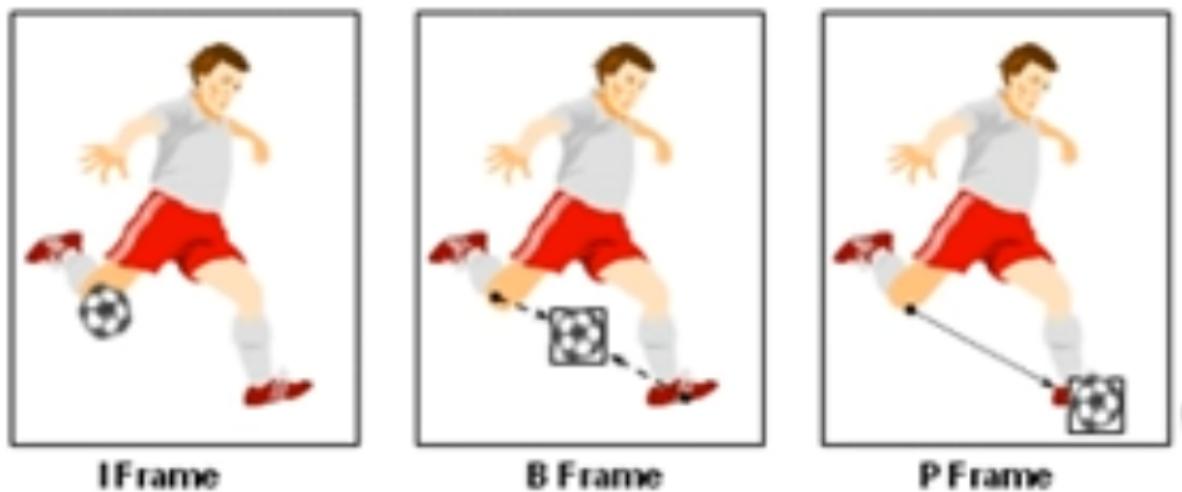
Kodek - kodek není způsob komprese, ale je to její konkrétní implementace. Tj formát je H.264, a kodek je OpenH264. Je to stejný rozdíl jako je Jazyk C (formát/specifikace) a GCC (implementace compileru).

Kontejnery - spousta známých formátů nejsou pro kompresi videa, ale kontejnery co spojujou video kompresní a audio kompresní formát dohromady. Tj třeba MP4 není formát videa, ale kontejner co obsahuje video a audio v různých možných formátech, který podporuje. Co podporuje je tady: <https://mp4ra.org/#/codecs#>

Většinou v MP3 bejvá H.264 video a Mpeg Audio Layer 3 audio. Může ale mít jiný. .AVI a .MOV jsou taky kontejnery, a proto může být takový problém je rozchudit - může v nich být obskurní kodek který jen tak nic nepustí, třeba Infinop Lightning Strike.

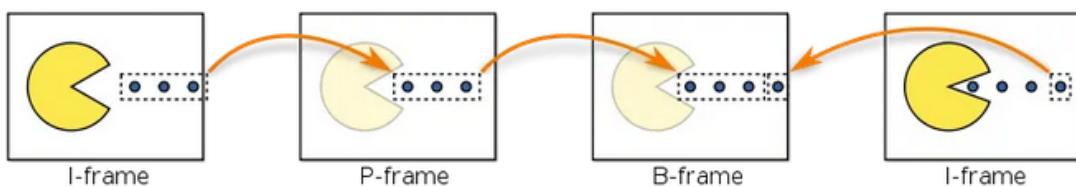
Většina lepších formátů s větší kompresí potom používá využívá ke kompresi temporal redundancy, tj časovou. Jde v podstatě o to, že když mám 3 framy:

Temporal Redundancy



Sem schopnej frame B postavit tak, že vezmu IFrame, trochu posunu míč, a vzniklou mezeru vyplním z Pframu. A takhle funguje MPEG - uloží si pár keyframů jako JPEG, a pak má uložený data o pohybu pixelů po obrazovce, a z nich skládá zbytek...

I-frames, P-frames, and B-frames



I-frame je fully encoded JPEG, tj má prostě celej obrázek.

P-frame je Predicted z Iframu podle toho, jak se co změnilo.

B-frame je bi-directionally predicted z předchozích framů, ale i z budoucího Iframu. B-framy nejsou ve všech implementacích/formátech, většinou mají jenom P-framy.

Jak to hledají? Cílem je, sehnat motion vector a difference pro každej z makrobloků z nějakého Iframu. Shání se to zhruba takhle:

- the difference between two macroblocks can then be measured by their Mean Absolute Difference (MAD):

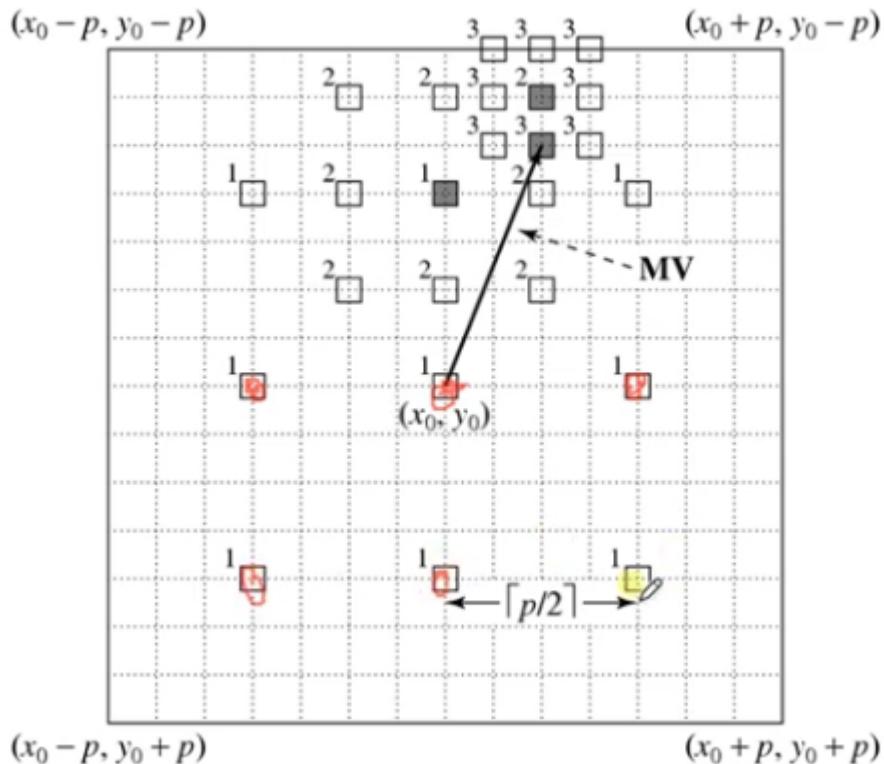
$$MAD(i, j) = \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} |C(x+k, y+l) - R(x+i+k, y+j+l)|$$

- the goal of the search is to find
 - a vector (i, j) as the motion vector $MV = (u, v)$ /
 - $MAD(i, j)$ is minimum:

$$(u, v) = [(i, j) \mid MAD(i, j) \text{ is minimum}, i \in [-p, p], j \in [-p, p]]$$

Prostě provedu součet všech rozdílů mezi jednotlivými pixely nějakého bloku, což mi dá Mean Absolute Difference, tj o kolik se liší. Ten si spočítám u pixelů v okolí, a vezmu ten, co má nejmenší MAD.

Většinou to zkouším jenom u bloků v menším okolí. Můžu ale taky použít 2D logaritmický hledání, nebo downsampling:

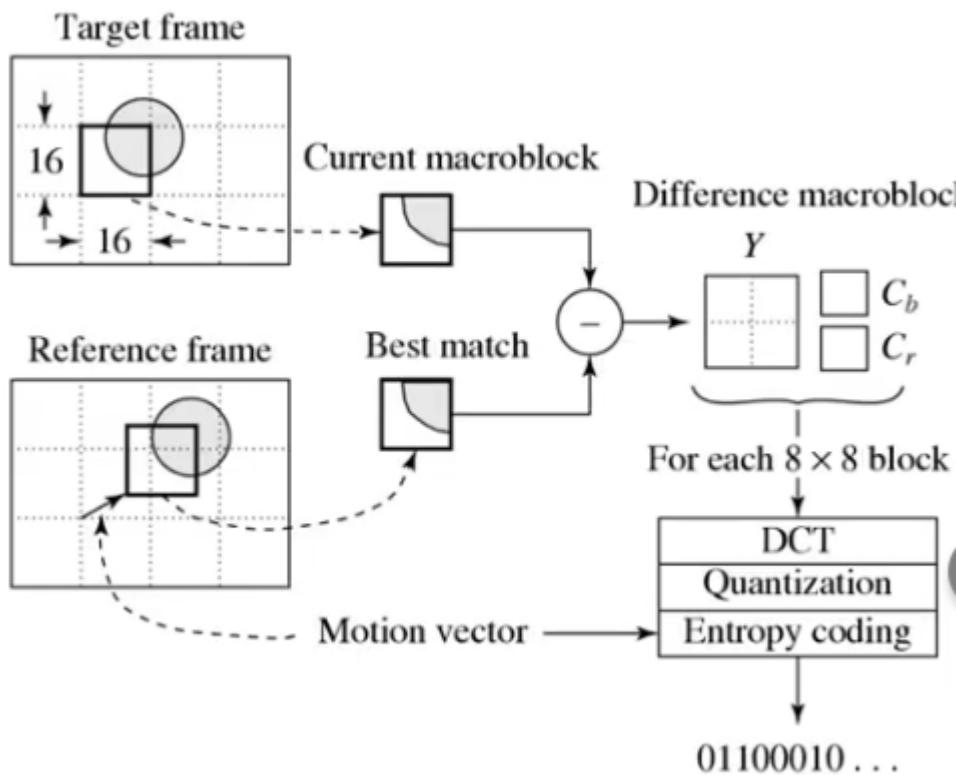


2D Log Search - checknu nejdřív 9 okolo (ty z 1čkou), z nich vezmu nejmenší, a kolem něj kouknu na dalších 9 s poloviční vzdáleností, a takhle jdu až jak hluboko chci.

Downsampling funguje podobně, akorát si místo hledání kolem zmenším resolution obrázku a pak zvětšuju.

Když teda stavím p-framy, tak to dávám dohromady nějak takhle:

N	size of MB
$k & l$	indices for pixels in MB
$i & j$	hor. & vert. displacements
$C(x+k, y+l)$	pixels in MB in TF
$R(x+i+k, y+j+l)$	pixels in MB in RF



DCT je Discrete Cosine Transform (fourier jenom s cosinama misto sinu), quantizace je lossy step co zahodí moc velký frekvence, entropy coding je Hummel nebo arithmetic.

AVC (taky známej jako **H.264 or MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC)**) - rozdělí frame na bloky 16×16 , a má rozestup iframů. Místo toho, aby si další (tj ty co nejsou iframy ukládal celý, tak se podívá do iframů a najde stejně 16×16 blok (makrobloky), a uloží si jenom vektor pohybu, tj o kolik se blok posunul. Je nejpoužívanější formát vůbec, pár let zpátky měl 91% market share. Pak stejně jako JPEG provede Diskrétní Cosine Transformaci a uloží to.

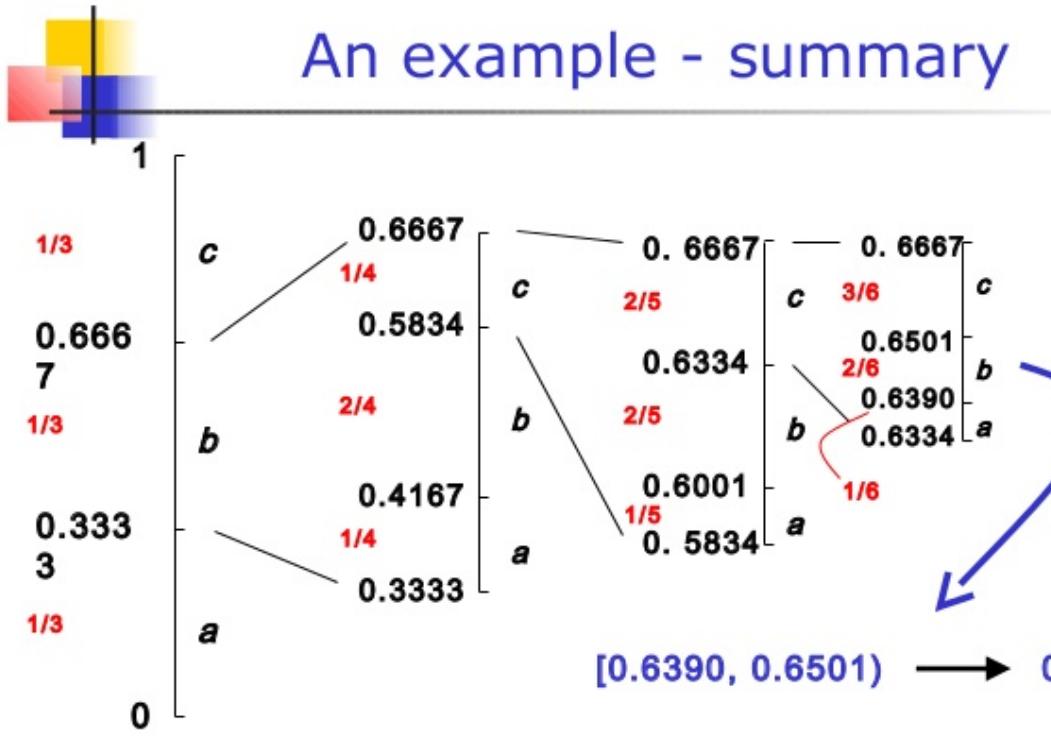
Používá taky ale **Context-adaptive binary arithmetic coding** -

Arithmetic coding je kódování, který řeší problémy Huffmanova a je o něco lepší. Huffman encoding má problém v tom, že jak data zapisuje po symbolech, tak i když máš fakt nízkou entropii, a ti moc nepomůže - Máme abecedu $\{0,1\}$ a 95% 1 a 5% 0, Huffman encoding stejně bude stejně dlouhý jako je ten řetězec, i když je entropie informací výrazně nižší.

Arithmetic coding to řeší tak, že celou zprávu co kóduju zapíše jako jedno desetinný (binární) číslo v intervalu $[0,1]$ a to tak, že interval postupně dělí podle znaku ve zprávě, dokud jí nemá celou. Postup je takovéhle:

1. Máme tabulku pravděpodobnosti výskytu znaků v abecedě. Třeba $A=0.5$, $B=0.3$, $C=0.2$ a zprávu **AABC**
2. Rozdělím si interval $[0,1]$ podle pravděpodobností, tj $A=[0;0.5]$, $B=[0.5;0.8]$, $C=[0.8;1]$
3. Podívám se na další znak zprávy, a kouknу do kterého spadá intervalu (V příkladu je to **A**, tj $[0;0.5]$)

4. Interval $[0;0.5]$ si rozdělím podle pravděpodobností poměrně,
 $A=[0;0.25], B=[0.25;0.4], C=[0.4;0.5]$
5. Podívám se na další znak ve zprávě, a vyberu ten podinterval. (Zase, A, tj $[0;0.25]$)
6. Takhle pokračuju dál, dokud nemám zakódovanou celou zprávu. Zůstane mi teda nějaký interval $[A,B]$.
7. Najdu jakýkoliv desetinné číslo, který do něj spadá, a to si uložím
8. To číslo je výslednej kód.



14

Context adaptive tomu ještě přidává to, že pravděpodobnosti nemám pevně daný, ale mění se mi během kódování v závislosti na to, co jsem kódoval za znak předtím. Tj pokud třeba kóduju angličtinu, a vím, že poslední znak byl e, tak vím podle tabulek jaký je nejčastější znak po e, a podle toho rozdělím ten další interval.

High Efficiency Video Coding (HEVC), (taky známej jako **H.265**, nebo **MPEG-H Part 2**) je jeho nástupce, relativně novej (2013), a používá se hlavně pro streaming a podobně. D8vá docela zabrat na HW, ale umí dosáhnout klidně o 50% větší komprese oproti AVC.

Místo 16×16 makrobloků používá CTU - coding tree unit, kterej může mít různý velikosti, a obraz teda dělí nelineárně.

5.9. Časová predikce (kompenzace pohybu)

Jedná se o predikci co je popisovaná v kapitole o komprezi videosignálu, s i a p framama.

5.10. Standardy JPEG a MPEG.

Moving Pictures Experts Group a Joint Photographic Experts Group.

Standardy formátů JPEG a MPEG byly popsány v předchozích kapitolách u kódování obrázků nebo videa, a řekl bych že to je to co chtějí slyšet.

Ale jenom samotná MPEG a JPEG komprese není to jediný, o čem ty standardy mluví, to je jenom jedna ze součástí. Jinak obsahují hodně věcí okolo media-delivery chainu.

Většina textů sou definice standardů, třeba u videa je to do kolika milisekund se musí vejít dekódování, aby šlo poutívat pro hovory, nebo úroveň komprese, nebo jaký musí podporovat metadata a funkce jako třeba titulky a podobně má podporovat, binární formát ve kterém má být zapisovaný tak, aby se dal dekódovat obecně. Ve většině případů se nezabývají přímo implementací, pouze definují postupy a standardy co by měla implementace splňovat, a jak k tomu už jednotlivý implementace dojdou je jedno.

Třeba MPEG má následující. Každá z jednotlivých kapitol má ještě několik (desítek občas i) částí, třeba H.265 je MPEG-H Part 2 a H.264 formát je MPEG-4 Part 10). Víc info maj na <https://www.mpegstandards.org/standards/>

- [**MPEG-I**](#) A collection of standards to digitally represent immersive media
- [**MPEG-DASH**](#) DASH is a suite of standards providing a solution for the efficient and easy streaming of multimedia using existing available HTTP infrastructure (particularly servers and CDNs, but also proxies, caches, etc.).
- [**MPEG-H**](#) Suite of standards for heterogeneous environment delivery of audio-visual information compressed with high efficiency
- [**MPEG-G**](#) A suite of standards to provide new effective and interoperable solutions for genomic information processing applications
- [**MPEG-4**](#) A suite of standards for multimedia for the fixed and mobile web.
- [**MPEG-5**](#) Standard to collect video coding parts
- [**MPEG-2**](#) A suite for standards for digital television
- [**MPEG-1**](#) A suite of standards for audio-video and systems particularly designed for digital storage media
- [**MPEG-IoMT**](#) APIs for communicating media devices (called Media Things)
- [**MPEG-CICP**](#) A suite of standards to specify code points for non-standard specific media formats
- [**MPEG-U**](#) Specification of the exchange, the display, the control and the communication of widgets with other entities and for data format for advanced user interaction (AUI) interfaces to support various advanced user interaction devices.. MPEG-U provides a general purpose technology with innovative functionality that enable its use in heterogeneous scenarios such as broadcast, mobile, home network and web domains
- [**MPEG-M**](#) MPEG-M is a suite of standards to enable the easy design and implementation of media-handling value chains whose devices interoperate because they are all based on the same set of technologies, especially MPEG technologies accessible from the middleware and multimedia services
- [**MPEG-V**](#) MPEG-V outlines an architecture and specifies associated information representations to enable interoperability between virtual worlds (e.g., digital content provider of a virtual world, gaming, simulation), and between real and virtual worlds(

e.g., sensors, actuators, vision and rendering, robotics). Please see <http://wg11.sc29.org/mpeg-v/> for a detailed specification.

- **MPEG-E** A standard for an Application Programming Interface (API) of Multimedia Middleware (M3W) that can be used to provide a uniform view to an interoperable multimedia middleware platform.
- **MPEG-C** A suite of video standards that do not fall in other well-established MPEGVideo standards
- **MPEG-B** A suite of standards for systems technologies that do not fall in other well-established MPEG standards
- **MPEG-A** A suite of standards specifying application formats that involve multiple MPEG and, where required, non MPEG standards
- **MPEG-21** A suite of standard that define a normative open framework for end-to-end multimedia creation, delivery and consumption that provides content creators, producers, distributors and service providers with equal opportunities in the MPEG-21 enabled open market, and also be to the benefit of the content consumers providing them access to a large variety of content in an interoperable manner.
- **MPEG-MAR** A Mixed and Augmented Reality Reference Model developed jointly with SC 24/WG 9
- **MPEG-7** A suite of standards for description and search of audio, visual and multimedia content
- **MPEG-D**
- A suite of standards for Audio technologies that do not fall in other MPEG standards

JPEG má to samý. Info je na <https://jpeg.org>

- [JPEG 1 -](#)
- [JPEG 2000](#)
- [JPEG AI](#)
- [JPEG AIC](#)
- [JPEG Pleno](#)
- [JPEG Systems](#)
- [JPEG XL](#)
- [JPEG XR](#)
- [JPEG XS](#)
- [JPEG XT](#)
- JPEG LS
- JPEG SEARCH
- JBIG

Part 1: Requirements and guidelines	Specifies the core coding system, consisting of the well-known Huffman-coded DCT based lossy image format, but also including the arithmetic coding option, lossless coding and hierarchical coding.	Part 2: Compliance testing	Specifies conformance testing, and as such provides test procedures and test data to test JPEG 1 encoders and decoders for conformance.	Part 3: Extensions	Specifies various extensions of the JPEG 1 format, such as spatially variable quantization, tiling, selective refinement and the SPIFF file format.
Part 4: Registration authorities	Registers known application markers, SPIFF tags profiles, compression types and registration authorities.	Part 5: File Interchange Format	Specifies the JPEG File Interchange Format (JFIF) which includes the chroma upsampling and YCbCr to RGB transformation.	Part 6: Application to printing systems	Specifies markers that refine the colour space interpretation of JPEG 1 codestreams, such as to enable the embedding of ICC profiles and to allow the encoding in the CMYK colour model.
Part 7: Reference Software	Provides JPEG 1 Reference Software implementations.				

Celkově historicky těch požadavků ve standardu bylo tak hrozně moc, že na to každej kašlal, a prostě si nějak implementoval JPEG.

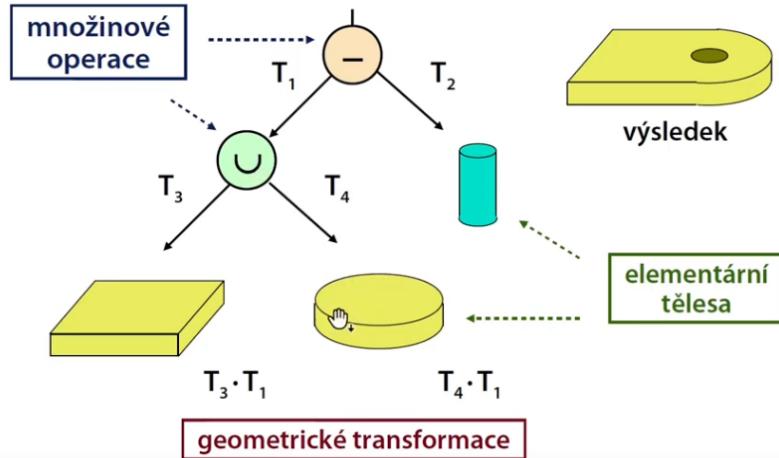
6. Realtime grafika, vykreslování scén, světel a stínů

6.1. Reprezentace 3D scén

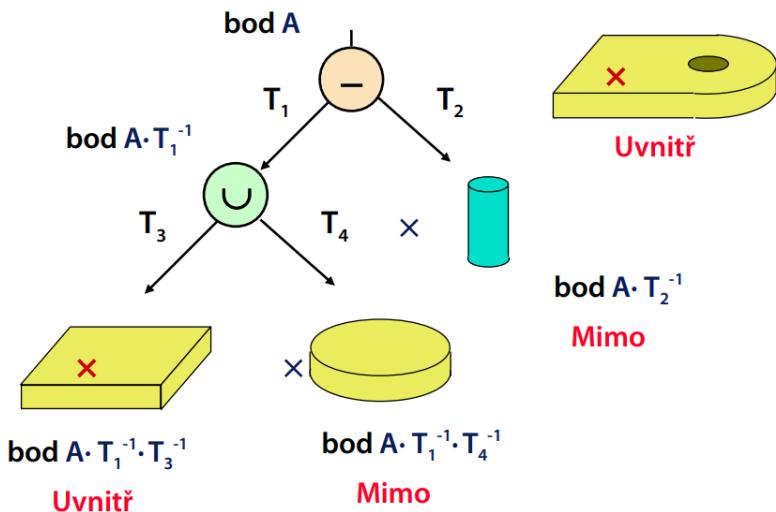
Dvě metody zobrazování:

- **Objemová** - přesně vím jaké jsou objemy těles, tj mám uložený info i o bodech vevnitř tělesa. Dobře se zjišťuje, jestli je bod v tělesu. Zobrazování už je ale horší. Nepoužívá se skoro vůbec.
 - **Výčtové** - mám uložený přímo výčet obsazeného prostoru, bod po bodu (tj 3D grid a voxely místo pixelů - tzv. Buněčný model). Používá se hlavně spíš jako pomocný metody třeba pro rychlejší vyhledávání, třeba když potřebuju rozdělit prostor. Je to třeba oktanový strom. (Což je 3D quadtree)
 - Když vykresluju quadtree, tak prostě začnu těma částma co jsou nejdál a překresluju je, tj odzadu dopředu, a určitě ho vykreslím správně.
 - **Constructive Solid Geometry** - mám definovaný elementární tělesa a transformace, a mezi nima dělám množinový operace - třeba vyříznou do sploštělé koule menší kouli, a mám donut. Velmi přesný, ale blbě se zobrazuje

CSG strom



- Transformace v hranách - T1, T2 atd, určují jak transformuju to, co je níž, předtím než to použiju v operátoru co je vejš. Tj třeba u sjednocení na obrázku vezmu) kvadr * T3) a sjednotím s (valeckem * T4)
- Všechny základní tělesa mám uložený jako jednotkový co jsou v počátku souřadnicového systému, a pak si je prostě naškáluju/otočím/posunu tam, kam je potřebuju v operaci tou transformační maticí. Takže si nemusím ukládat spousty tvarů, ale stačí od každého jeden.
- Check bod x CSG strom, tj leží bod v tělesu:
Stačí když vezmu inverzní matice transformací, co vedou ke každému z listů (co obsahuje normovaný tvar), a pak provedu check jestli ten transformovaný bod leží v tom tvaru. A tohle provedu pro všechny cesty, a výsledky porovnám bool ekvivalentem množinových operací.



Tj spočítám se tyhle 2 výsledky inverzníma transformacema, a pak se podívám jestli platí (bodCtverec OR bodKruh) AND NOT (bodValec)

- **Povrchová** - mám info o povrchu/plášti těles, tj jeho hrany a stěny. Hůř se zjišťuje, jestli daný bod je v tělesu, zobrazuje se ale výrazně líp.

Povrchové reprezentace



„Drátový model“

- pseudo-povrchová reprezentace
- pouze **vrcholy** a **hrany** těles (nelze použít pro výpočet viditelnosti)

VHS(T) reprezentace

- kompletní topologická informace: seznamy **vrcholů**, **hran**, **stěn** (a **těles**)

„Okřídlená hrana“ („winged-edge“)

- redundantní informace pro **rychlé vyhledávání** sousedních objektů (hrany incidentní s vrcholem...)

○

VHS(T) reprezentace je takovej přirozený způsob, jak si ukládat data. Mám uloženo následující:

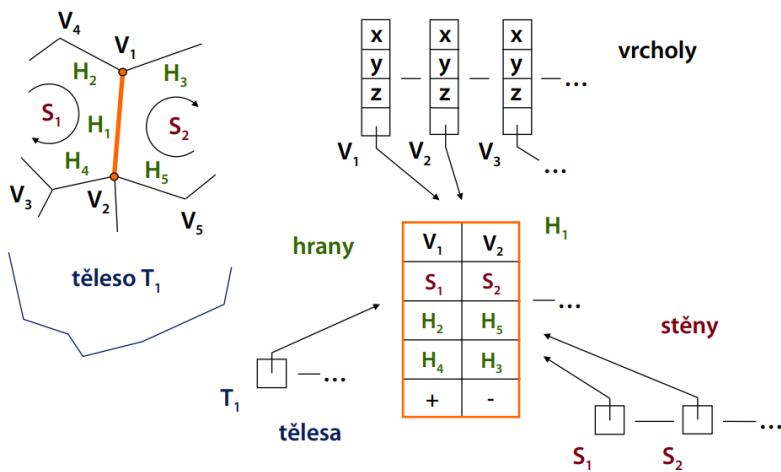
- Pole všech vrcholů
- Pole všech hran (tj, dvojice vrcholů)
- Pole všech stěn (tj, jaké hrany tvoří stěnu)
- Můžu mít i pole všech těles (tj, co za stěny dá dohromady těleso)

V praxi je blbý, protože se hrozně blbě zjišťují informace o lokálnosti - například když potřebuju zjistit, co za stěny spolu sousedí. Nebo zjistit, ve kterých stěnách je danej vrchol - abych to zjistil, musím si projet pole hran, a zjistit, ve kterých je ten vrchol. A pak si projet pole stěn, atd...

Místo toho se používá o něco lepší reprezentace **okřídlený hrany**. Reprezentace to není tak kompaktní, ale má výhodu v tom, že se v ní líp hledá.

Mám pole vrcholů, a pole hran. U každý hrany mám ale odkaz na informace kolem:

Okřídlená hrana („winged-edge“)

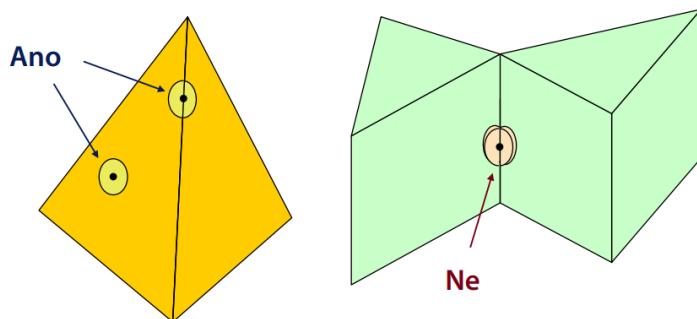


Vím teda se kterýma stěnami a hranama sousedí. Vím teda zhruba tohle:

- Pole vrcholů
- Pole hran, který obsahuje:
 - Pointer na vrcholy který spojuje
 - Pointer na levou a pravou stěnu vedle hrany
 - Pointer na sousední hrany též stěny
 - Info o pořadí v jakém jsou vrcholy pro danou stěnu (tj jesti jí kreslím po nebo proti směru)
 - V příkladu nahoře je to + u S1, protože ta stěna je kreslená v pořadí V4 -> V1 -> V2 -> V3 ..
 - U S2 je mínus, protože je kreslená V5 -> V2 -> V1 ->...

Podmínka aby šla použít winged edge - musím mít 2-manifoly:

Def: Pro každý povrchový bod existuje okolí, které je topologicky ekvivalentní s rovinou



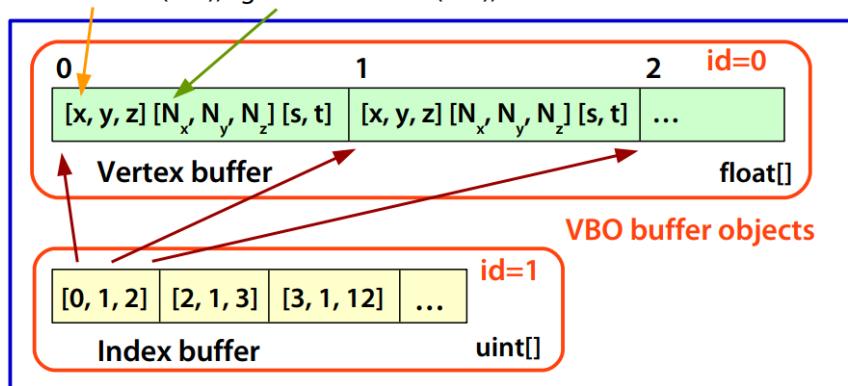
V podstatě to znamená, že by se nemělo stát, že se v jednom místě dotýká hrany 3 a více stěn

V OpenGL se objekty reprezentují skrz Vertex a Index Buffer. Vertex buffer mi popisuje co mám všechno za vertexty, a všechny data k nim patří (tj třeba barvu, materiál, normálu a další), a Index buffer obsahuje indexy vertexů ve vertex bufferu který skládají požadovaný tvary (většinou prostě obsahují trojice bodů, co dělají trojúhelníky).

Vertex Buffer Objects (Buffers)



```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glVertexAttribPointer( ... ); glNormalPointer( ... ); ...
```

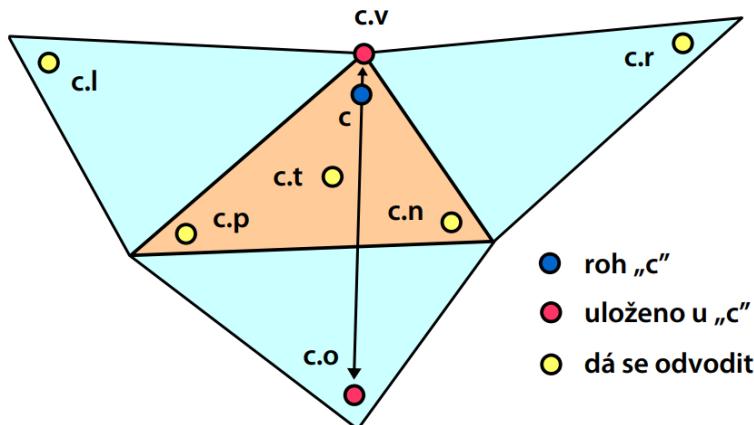


```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 1);
glDrawElements(GL_TRIANGLES, ...);
```

GPU memory

Existují další reprezentace, například docela zajímavá je **Corner table** - ukládám si jenom pole vrcholů, a pak si ukládám pole bodů v rozích trojuhelníku. U každého rohu mám uloženou vrchol kterej u něj je, a protější roh sousedního trojuhelníku.

Rohy mám za sebou proti směru hodinových ručiček:



Díky tomu si už zbytek můžu odvodit:

- Další vrcholy trojúhelníku - je to roh co je v poli rohů před nebo za tím rohem, modulo 3. (Protože rohy sou po trojicích)
- Sousední trojuhelník - protější mám uloženou u sebe, a levej a pravej trojuhelník zjistím tak, že se podívám na protější roh u dalších rohů mýho trojuhelníku.

Povedlo se jim tím, ještě za použití kompresí, dosáhnout až na 9bitů per vertex.

Pozice vertexů zkomprimovaný až na 7bpv, za použití predikce

Topologie sítě (rohy) zkomprimovali (ztrátově) na 1bpv až 2bpv.

Tj celkem je to asi 9 bitů na vrchol, což je dost málo (třeba winged edge má 3*float + 3* uint)

6.2. Výpočet viditelnosti

Pro výpočet viditelnosti přednášky popisovali 2 algoritmy-

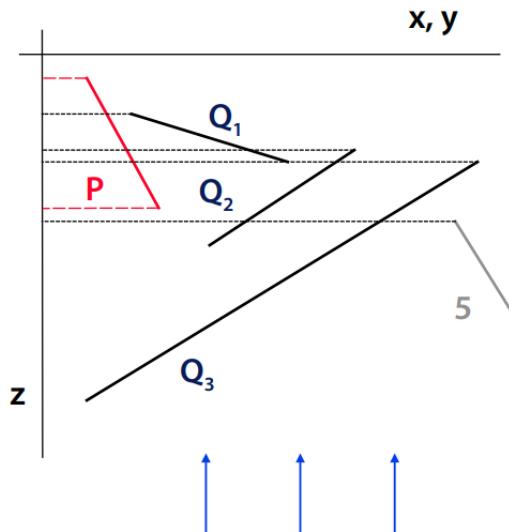
Malířův, který maluje odzadu a překresluje

Z-buffer, který maluje odpředu na základě z-souřadnice bodu.

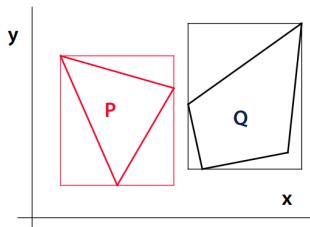
Idea malířova algoritmu je taková, že prostě tím, že budu kreslit odzadu, tak postupně překreslím vše co jsou dál věcmi co jsou blíž ke kamere. Na to je potřeba ale správně poznat, který trojuhelník co vykresluji překrývá co. Dělá to v několika krocích:

1. Trojuhelníky seřadí podle nejvzdálenějšího bodu trojuhelníku od kamery.
2. Vezme dosud nevykreslenej trojuhelník co má nejvzdálenější bod. Říkejme mu trojuhelník P
3. Zkontroluje, jestli ten trojuhelník nepřekrývá nějaké jinej ještě nevykreslenej trojuhelník. Kontrola se dělá v několika krocích:
 - a. Vezmu nejbližší a nejvzdálenější bod trojuhelníku co vykresulu, a podívám se, který nejvzdálenější body jinejch trojuhelníku do toho intervalu spadají (Tj,

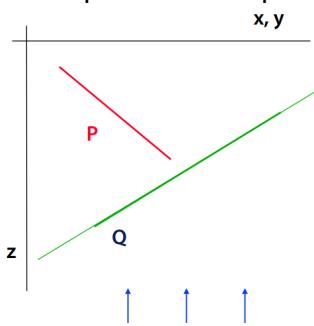
může je potenciálně překrývat)



- b. Pro každý trojúhelník co sem spadá provedu následující kroky (Qi je trojúhelník co zrovna zkouším):
 - i. Udělám AABB box v rovině kamery, a podívám se, jestli se překrývají. Když ne, tak můžu končit, protože vím, že se ani trojúhelníky nemohou překrývat:

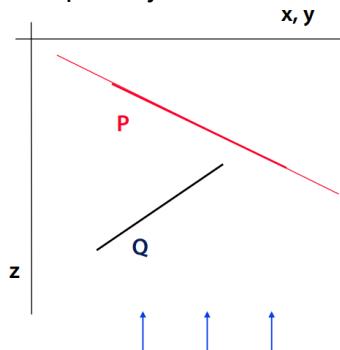


- ii. Pokud AABB čtverce překrývají, spočítám souřadnice vertexů stěny P v rovině stěny Q (tj. jestli jsou vrcholy P za Q). Pokud ano, můžu končit protože se nepřekvýrají



- iii. Pokud stěna P protne stěnu Q, tak pokračuju opačným testem - jestli jsou souřadnice Q v rovině stěny P. Pokud ano, můžu končit protože

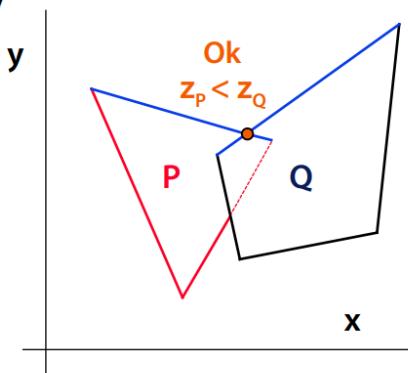
se nepřekrejou.



- iv. Pokud se i ten test nezdaří, musím prostě udělat úplnější test překrytí, a podívat se přesně, jestli se teda překrývají nebo ne.

Testujeme proti sobě **všechny**
hrany P a Q

- najdeme-li průsečíky, porovnáme v nich souřadnice z
- je-li vždy P za Q , test Q končí úspěšně (pass)
- v opačném případě **nelze P nakreslit jako první!**

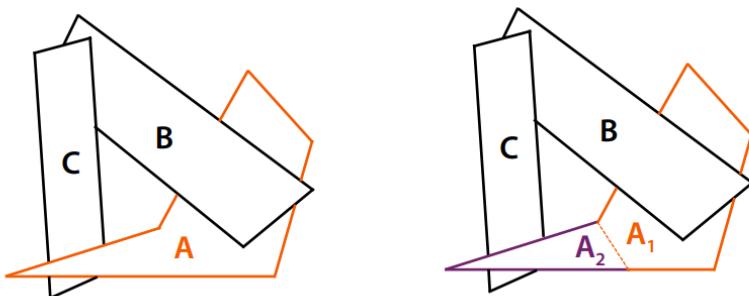


Taky se musím podívat, jestli neleží P celý uvnitř Q (tj hrany nemají průsečíky, ale stejně P překrývá Q)

- v. Pokud se i v tomhle testu ukázalo, že se překrývají, tak prohodím pořadí P a Q v seřezaném seznamu, a provedu znova testy pro stěnu Q , která se ukázalo je dál než původně testovaná stěna P , i když nemá nejvzdálenější bod dál.

4. Vykreslí ten trojúhelník, u který skončil nejdál (tj prošel testama vejš se všem potencinálníma kolizema)

Může se mi ale stát, že se při prohazování bodů zacyklím



Jestliže je testován některý kandidát podruhé, došlo k **zacyklení**

Cyklus lze odstranit **rozdelením** některé stěny

- správné pořadí: A_1, B, C, A_2

Takhle postupně vykreslím všechny trojúhelníky.

Z-Buffer - algoritmus, kterej naopak funugje líp, když kreslí zepředu. Je mega primitivní -

Jde po pixelech, a né po trojuhelnících. U každýho pixelu co vykresluju si pamatuju ještě navíc jeho hloubku, tj Z souřadnici - jak je daleko od kamery. Mám FrameBuffer, což je buffer pixelů co se pak vykreslí a obrazovku, a Zbuffer, kterej určuje, jak daleko je pixel co je zrovna vykreslenej ve VideoBufferu.

Na začátku je ZBuffer nekonečno, a VideoBuffer má barvu pozadí.

Potom pixely vykresluju takhle:

```
void writePixel (int x, int y, float z, Color color)
{
    if (z < zbuf[x, y])
    {
        zbuf[x, y] = z;
        videoRAM[x, y] = color;
    }
}
```

Prostě se podívám, jestli pixel co chci vykreslit je blíž než ten, co už je vykreslenej, Když je, tak ho překreslím, protože je blíž, a pamatuju si, jak daleko byl. Když není blíž, tak ho zahodím.

ZBUFFER - Může pro pixely zbytečně počítat uvnitř **fragment shaderu**

Taky je ale potřeba zjistit, co přesně za fragmenty ten paprsek protne. Většinou se to dělá podobně jako v případě kolizí - reprezentuju si scénu v nějakém hierarchickém modelu, třeba octantree, nebo BPS, nebo kdTree, a netestuju průsečík se včem polygonama, ale jenom s těma, ke kterejm dojdu postupně ve stromu, že to má smysl. Je to stejně princip jako u kolizí.

Hezkej algoritmus je na procházení BSP stromu -

1. Do zásobníku vložím kořen BSP stromu
2. Vyndám ze zásobníku uzel U
3. Pokud je U list, tak:
 - a. Pro všechny tělesa zjistím, jestli je paprsek protne, a vezmu to nejbližší a mám hotov
 - b. Pokud nic nenašel, vrátím se na krok 2 a beru další uzel.
4. Pokud je U vnitřní uzel (tj. má potomky), tak si vypočítám průsečík **P** s řeznou rovinou toho Uzlu (každej uzel v BSP má určenou rovinu, která ho dělí na 2 části, a tím je půlnej ten prostor)
5. Určí se průsečíky paprsku s obalem uzlu U. Obal uzlu je čtverec co obsahuje oboje půlky uzlu, prostě ta plocha co je rozdelená řeznou rovinou. Průsečík co je blíž počátku paprsku se označí **Pn** a ten co je dál je **Pf**
6. Potomka uzlu U (tj. jednu z půlek) co je blíž počátku paprsku označ jako bližší uzel, tu druhou jako vzdálenější. Formální popis je: Bližší potomek je potomek reprezentující poloprostor určený řeznou rovinou co obsahuje počátek paprsku. (Tj,

udělám poloprostor podle řezný roviny (co už není omezený obalem uzlu ale je nekonečnej), a ten ve kterém je počátek paprsku je blíž).

7. Podle polohy bodu P (průsečík paprsku a řezný roviny) vzhledem k Pn a Pf se na zásobník uloží:

- Pokud je P za (dál než) Pf, ulož bližší uzel
- Pokud je P blíž než Pn, ulož vzdálenější uzel
- Jinak ulož oba dva uzly.
- (Tohle v podstatě popisuje situaci, kdy už si zanořenej ve stromu, a paprsek ti protíná jenom jendu z těch půlek toho stromu. To pozáš podle toho kde je průsečík)

Nevýhody Z-bufferu



Mazání bufferu na začátku zpracování snímku

Některé pixely ve Video-RAM se **několikanásobně překreslují**

- zbytečné vyhodnocování jejich barvy (fragment shader)

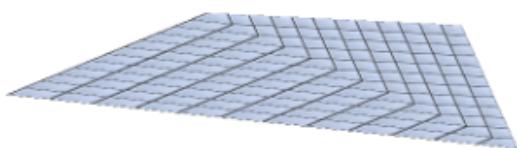
„Z-fighting“

- nepříjemné artefakty při kreslení objektů ve stejné rovině
- velmi rušivé při animacích
- je to často chybou modelu nebo špatně nastavených parametrů projekce

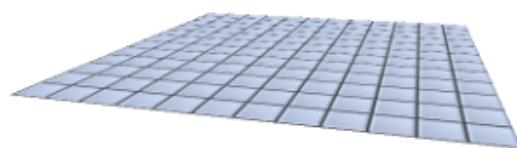
Neřeší jednoduše **poloprůhledné 3D scény**

- nutnost předzpracování (3D třídění primitivů)

Je potřeba dát si bacha na korektní interpolaci hloubky -



affinní



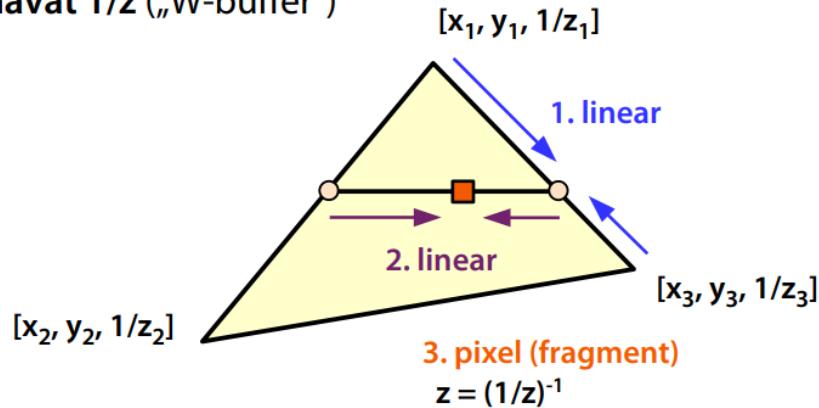
perspektivně správné

Perspektivně korektní interpolace hloubky



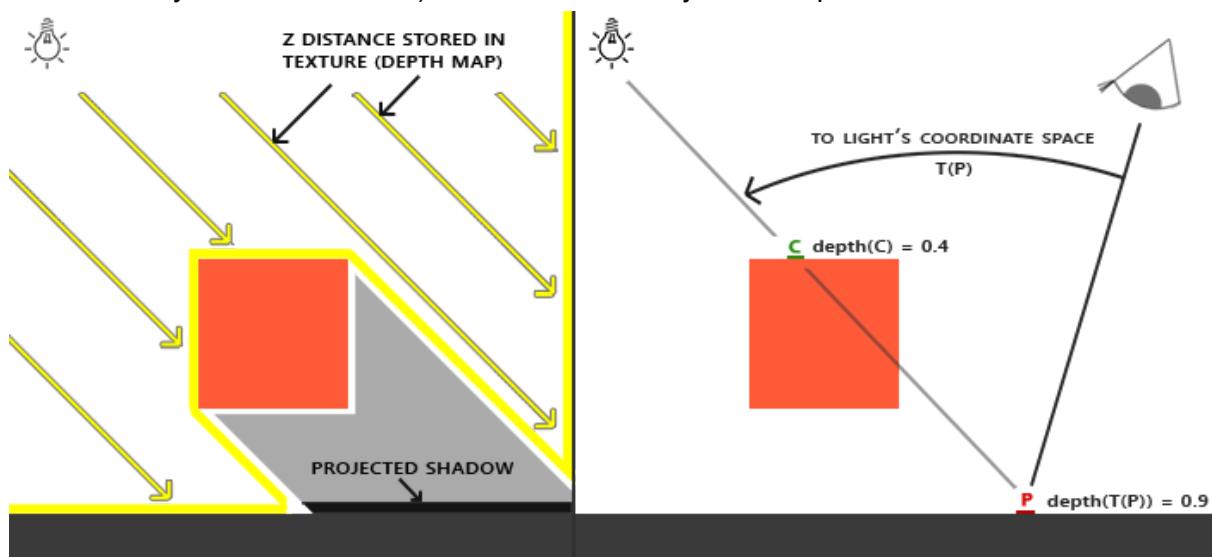
V perspektivě nemůže interpolátor (rasterizer) na ploše průmětny přímo lineárně interpolovat hloubku z

- místo toho lze interpolovat převrácené hodnoty $1/z$
- v každém pixelu můžeme z spočítat dělením nebo přímo porovnávat $1/z$ („W-buffer“)



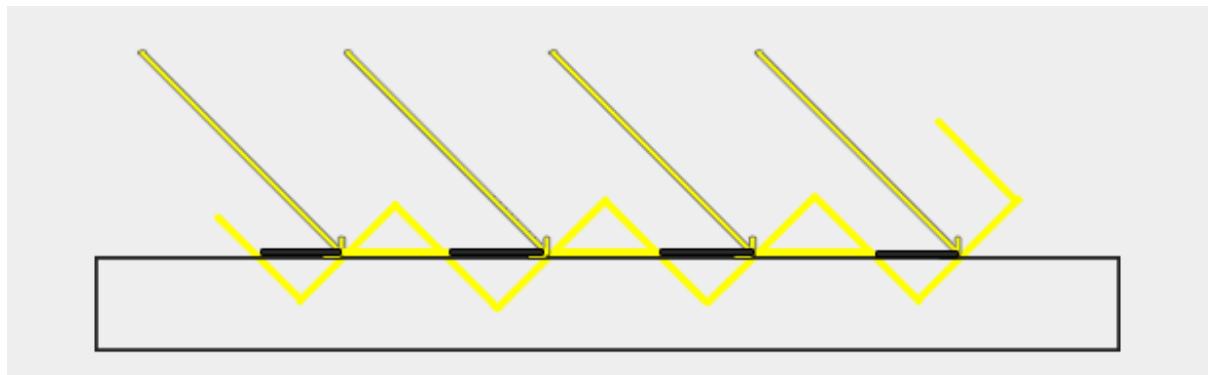
6.3. Výpočet vržených stínů a měkké stíny

Pro výpočet vržených stínů se používá relativně jednoduchý algoritmus - vyrenderujeme si scénu (stačí zbuffer) tak, že položíme kameru do směru světelného zdroje, pro kterou kreslíme stín, a postavíme si buffer Z hodnot (shadow map), který nám popisuje co ta kamera vidí nejblíž. Když potom počítáme zbytek scény pro normální kameru, tak se jenom podíváme jestli pixel co vykreslujeme leží blíž nebo dál než jakou hodnotu máme uloženou v shadow mapě (musíme si převést souřadnice, tj zjistit z hodnotu v zobrazení této stínové kamery). Když je dál, víme, že není vidět a tudíž světlo nepočítáme. Pokud je stejně (nebo blíž? Ale to by se asi dít nemělo), tak víme, že na něj světlo dopadá.

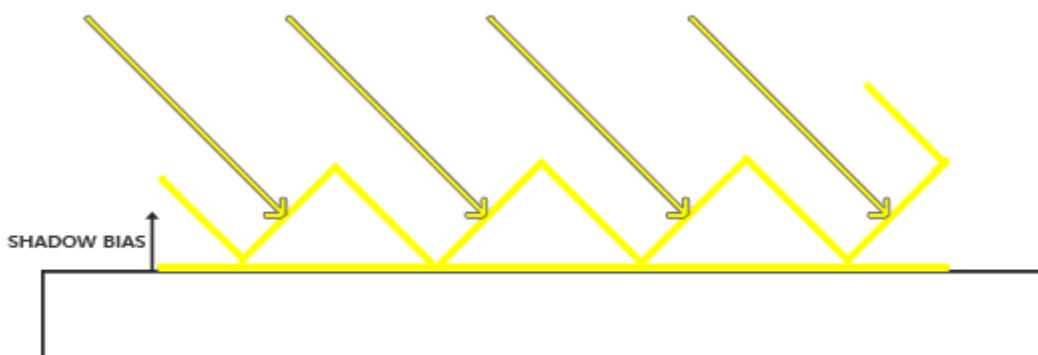


Jsou tu dva problémy - nepřesnost shadow mapy, a problémy s aliasingem.

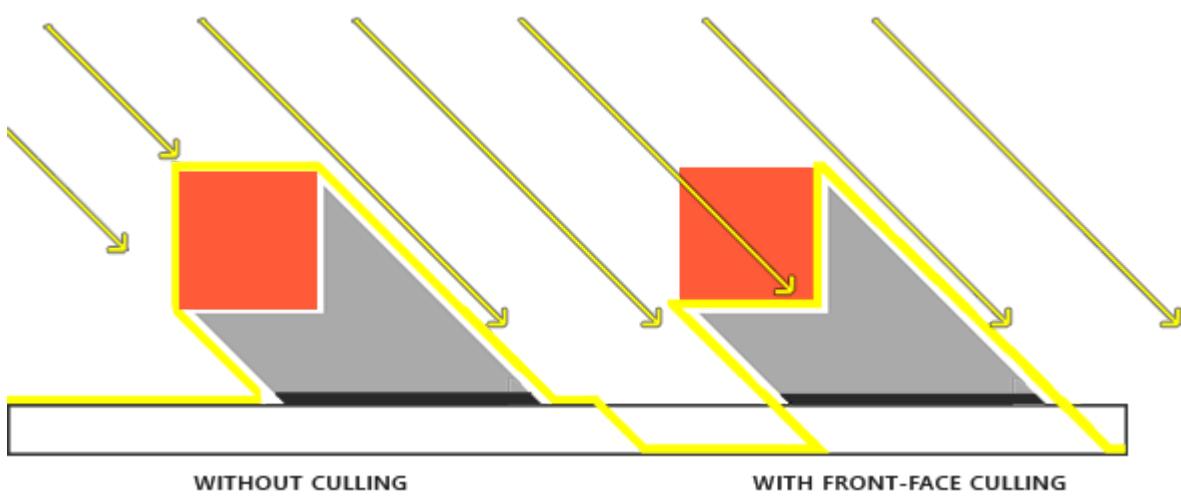
Nepřesnost shadowmapy může způsobit, že se bod kvůli přenosti sám zastíní (tj vyjde nám, že je dál než je, protože se ty čísla přesně nerovnají. To se řeší tím, že když počítám stíny do shadowbufferu, tak všem bodům trošku zvětším vzdálenost (tj, zapíšu si o něco větší hodnotu), abych zajistil, že se mi nestane že kvůli nepřesnosti se pixel sám zastíní. Říká se tomu shadow acne:



A řeší se biasem:

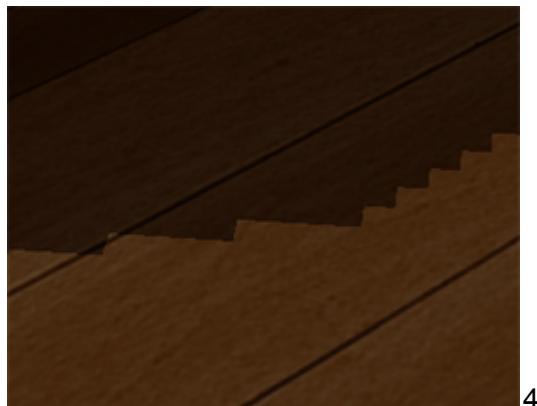


Bohužel to ale může přidat tzv peter panning, kdy stín nesedí s hranou objektu, a věci co maj bejt spojité nejsou (přesně kvůli tomu biasu). Místo toho jde použít **front face culling** - prostě při renderování shadow mapy nepoužívám polygony co jdou ven z objektu, ale použiju ty vnitřní (co normálně nejsou vidět):



Problém ale nastáv třeba u ploch co nemaj vnitřní faces, jako je třeba plane.

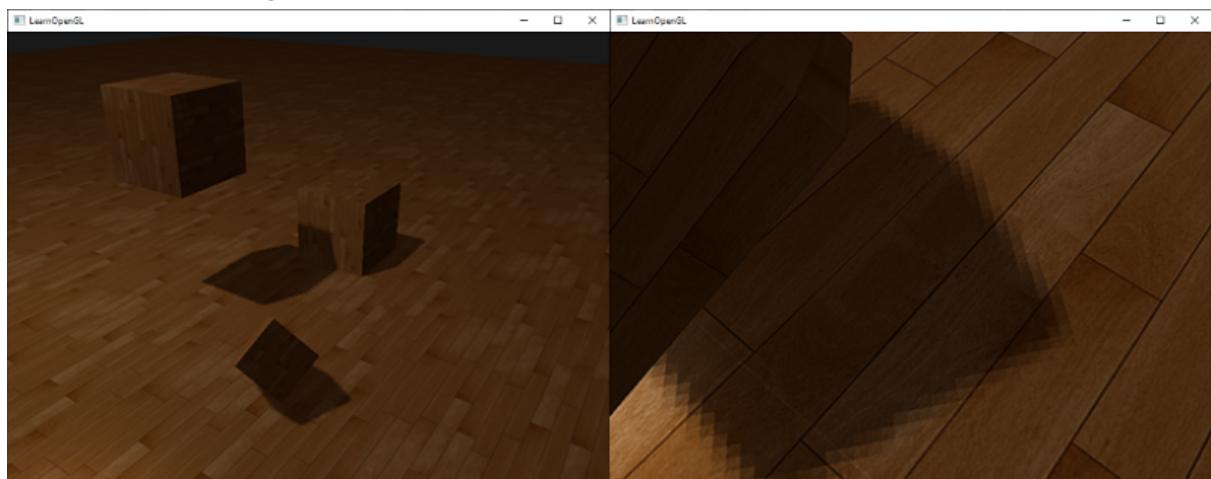
Další problém je aliasing, kterej se řeší prostě tak, že místo toho abych bral stín jenom z jednoho pixelu (kde vzniká aliasing), tak si ho spočítám z několika sousedních:



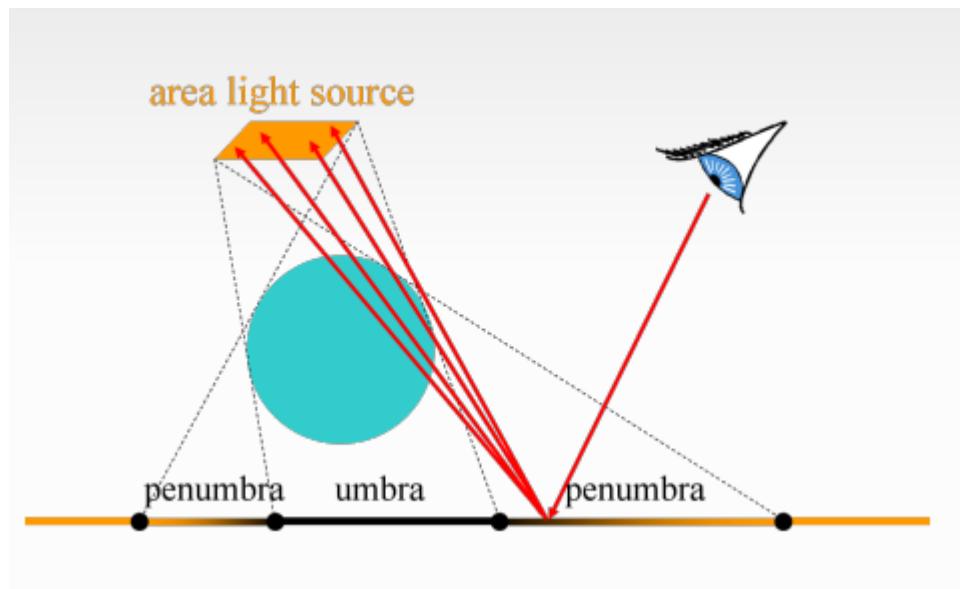
4

Používá se na to **percentage close filtering**, a je spoustu různejch algoritmů jak to implementovat:

<https://developer.nvidia.com/gpugems/gpugems/part-ii-lighting-and-shadows/chapter-11-shadow-map-antialiasing>

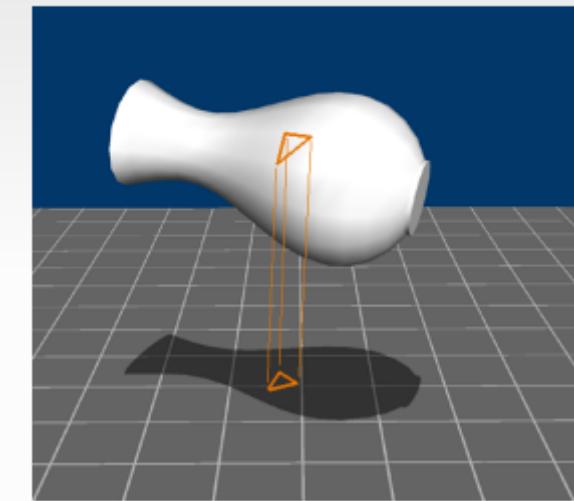


Další možností je raytracovat stíny - vrhnu si paprsek z pixelu který zrovna počítám směrem ke světlu (nebo klidně i několik paprsků), a podle toho kolik jich nedopadne do světla spočítám stíny:



Pokud mě zajímají jenom stíny na zemi, můžu si v dalším passu promítat polygony do

Planar shadows

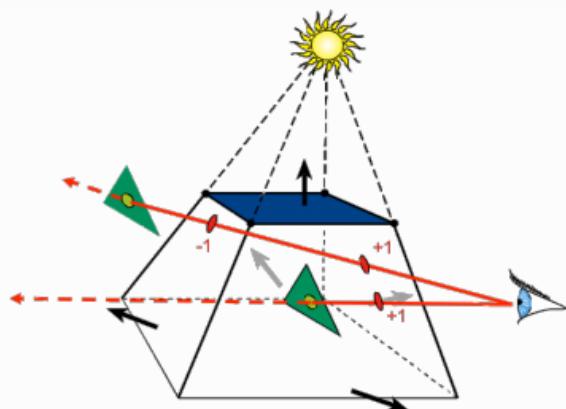


roviny země, a prostě je vykreslit jako stín:

Pak tu máme podobný algoritmus, který se jmenuje **Shadow Volumes**. Ten se dělá tak, že si pro každý objekt vytvořím jeho shadow volume, což je nekonečný objem určující co všechno je v jeho stínu - cokoliv co je v shadow volume bude zastínovaný tím objektem. Vytváří se třeba tak, že si pomůžu compute shaderem a polygon promítu do nekonečka ze směru světla:

Shadow volumes

- Shadow volumes for each light
- Project silhouette edges to infinity
- Create top/bottom caps
 - Prevent light leaks
 - Camera inside shadow
- Test objects for shadows
- Utilize stencil buffer
 - Count front/back faces
 - 0 = not in shadow



Při renderování potom použiju stencil buffer (je to prostě 8bit buffer), a když vykresluju nějaký bod, tak si cestou k němu počítám v kolika shadow volumes jsem - když vlezu do nový shadow volume (tj procházím její venkovní stěnou dovnitř), zvětším si ho o jedna, když vycházím zadní stěnou ven, zmenším ho o jedna. Když narazím na pixel, tak podle toho jestli je 0 nebo větší poznám v kolika jsem shadow volumes.

Algoritmus na vykreslování je tenhle:

- 1 Depth pre-pass, *disable* z-write
- 2 Render the scene with ambient only
- 3 For each light render shadow volumes:
 - *Disable* color write, *enable* stencil write
 - Stencil OP on **depth pass**:
 - front face **increment**
 - back face **decrement**
 - *Enable* color write, stencil test, *disable* stencil write
- 4 Render the scene again, light where stencil = 0

Tj - depth pass - vyrenderuju si jenom hloubku scény.

Pak vystínuji objekty jenom s ambient světlem.

Přepnu si blend-mode na additive (tj, když v dalším passu kreslím do pixelu, tak přidávám barvu k tomu co už tam je, místo toho abych přepisoval).

Znova renderuju, ale tentokrát přes shadow volumes, a pamatuju si kolika sem prošel volumes než narazím na pixel, a zapíšu si do stencil bufferu hodnotu na pixelu

Pak renderuju ještě jednou, a tam kde je stencil 0 kreslím světlo.

Bíbý je, když začnu uvnitř shadow volume. Pak je lepší z-fail, kdy fungují opačně - stencil OP změním na depth fail, front face decrement a back face increment, a počítám kolik volumes je za objektem.

Poslední relativně novou možností je **Raytraced Signed Distance Field**, kde si předrenderuješ v podstatě voxelovou distance mapu, kde máš pro každý bod ve scéně uložený jak daleko je nejbližší bod nějaké geometrie. Potom víš, že když počítáš ray ke světlu z nějakého pixelu, tak jak daleko můžeš skočit aniž bys na něco narazil, takže můžeš ušetřit pár testů.

6.4. Rozptyl světla pod povrchem

Subsurface scattering je jev který přichází z radiometrie, a popisuje způsob jakým se chová část světla, která se neodrazila ale proletěla dovnitř do objektu. Tady záleží na vlastnostech objektu, konkrétně jestli je homogenní nebo ne.

Homogenní hmotou prostě paprsek projde. Občas se změnou barvy, když pohlcuje nějakou vlnovou délku:



Jakmile ale není homogenní, tak tam světlo naráží na různý partikly a začíná se odrážet uvnitř média

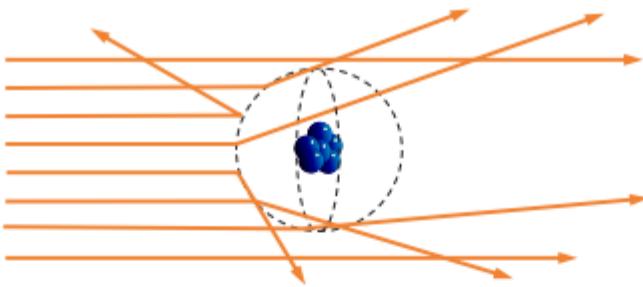
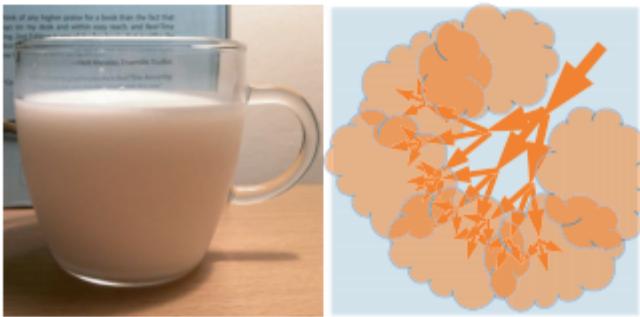


Figure 6: Particles cause light to scatter in all directions.

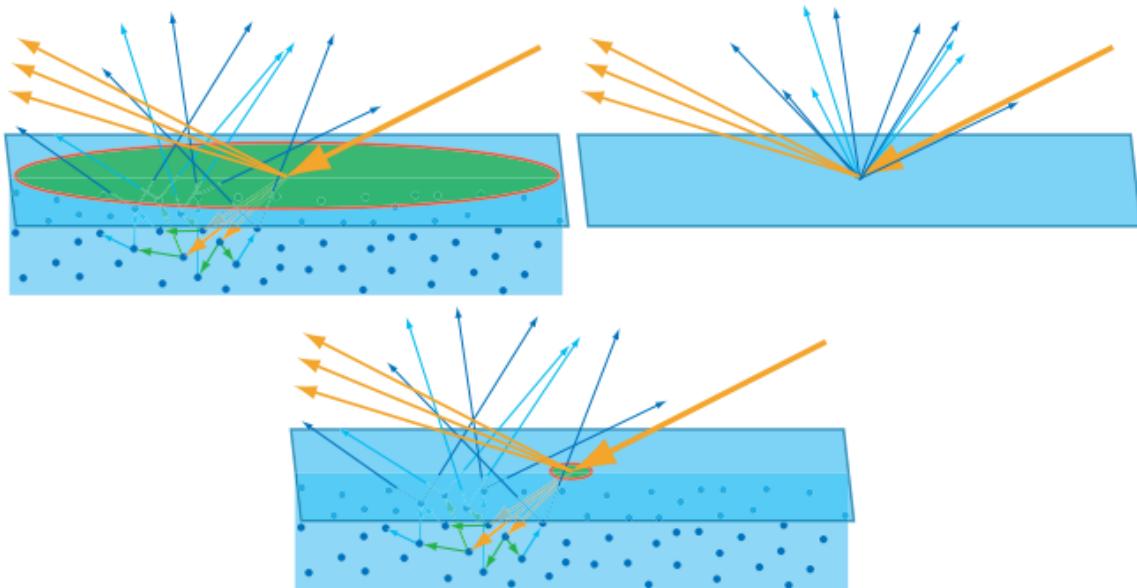


Light in cloudy media (left) has its direction somewhat randomized as it propagates (right).



A eventuálně se odrazí zase nazpátek ven, čemuž se říká subsurface scattering. Protože i ty paprsky se nějak promítnou na barvě.

Vlastností objektů je tzv distance-to-exit, což je vzdálenost jak daleko od bodu vstupu se paprsky zase dostanou ven. Pokud je takhle veličina menší než pixel, tak nemusím subsurface scattering řešit, protože se nijak neprojeví:



Zelená barva je velikost mýho samplingu, tj pixelu. Modré čáry sou subsurface scatterernutý paprský. Když se mi vejdu do pixelu, tak můžu počítat jako by prostě vylezli v bodě vstupu a nemusím řešit jak se scatterovali, protože se to na pixelu stejně neprojeví. Pokud ale mám pixel menší, tak musím použít jiné algoritmus.

Počítá se bohužel docela blbě, zvlášť v tom druhém případě. Jednou z možností je spočítat si hodnoty jak se světlo propaguje v daném bodě, a udělat si z toho SSS texturu, kterou pak za pomoci konvoluce přidávám při renderování. Ríká se tomu diffusion profile, a získám to tak, kdybych střelil přesně bodový laser do jednoho místa povrchu. Třeba pro kůži vypadá takhle:



Jenže konvoluce furt trvá, takže to v realitu nejde pořádně dělat.

Několik approachů na to ale je, a sou skvěle popsáný tady:

<https://therealmjp.github.io/posts/ssss-intro/>

TSSSS - Texture-Space SubSurface Scattering - Místo toho abych si počítal scanning ve 3D, tak ho počítám jenom per texture, kterou nějak rozbluruju a tím dostanu efekt SSS.

SSSSS - ScreenSpace SubSurface Scattering - Nehraju si s texturama, ale rovnou to počítám ve screen-spacu, tj až na základě hotově vyrenderovaného framu.

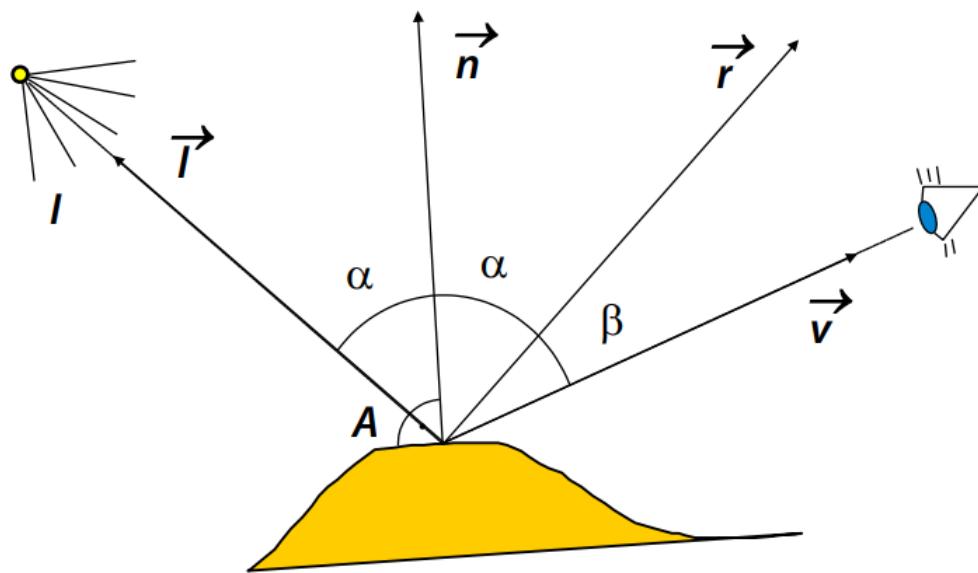
Pre-integrated Subsurface Scattering - SSS je předpočítanej a integrovanej do materiálu. Bohužel ale funguje jenom když je povrch koule, takže si pak musím popsat model/texturu objemama a lokacemi koulí pro který pak spočítám SSS.

Víc info o těchhle metodách je na link vejš.

6.5. Modely osvětlení a stínovací algoritmy

Nejpoužívanější model osvětlení a stínování je Phongův model osvětlení, kterej je empirickou approximací PBR - physical based renderingu.

Docela častej Phongův model. Pracuje tak, že pokud chci získat barvu fragmentu na který se zrovna koukám, tak si ho spočítám podle světla, co na něj dopadá:



Mám tu několik důležitých úhlů a vektorů - n je normála povrchu, l je paprsek světla, kterej dopadá na fragment (počítá se prostě tak, že odečtu souřadnice fragmentu a světla) a v je vektor pozorovatele, tedy odkud koukám na fragment. r je potom vektor odrazu, tj kam by se správně světlo nejvíce odrazilo.

Pozor - odraz r bude vždycky ve stejný rovině jako l , kdežto v už v rovině bejt nemusí.

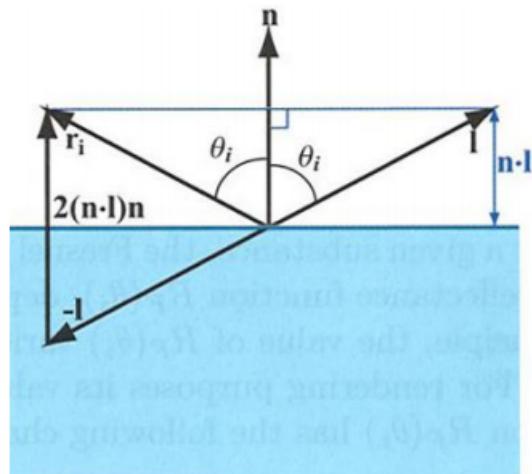
Calculating the reflection vector

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

R se počítá jako

Vector for reflected light can be computed according to the Law of Reflection:

$$r_i = 2(n \cdot l)n - l$$



IMG: Real-time Rendering Third Edition (2008)
T. Akenin-Moller, E. Haines, N. Hoffman

Barvu pixelu spočítám jako: $C_p = \text{SUM}[i = \text{světla ve scéně}](E_{is} + E_{id}) + E_a$

Tj, Barva pixelu je rovná součtu Spektrálních (odraz) a Diffuzních(surface scattering a podobně) složek všech světel plus approximace ambientní složky (což je konstanta, a představuje ambientní světlo)

Jednotlivé složky jsou pak takhle:

$$E_s = I_i \cdot C_s \cdot k_s \cdot \cos^h \beta$$

Globally constant lighting

C_s ... highlight color (RGB)

Approximates / replaces indirect light

k_s ... specular coefficient (0.0 to 1.0)

$$E_A = C_D \cdot k_A$$

$\cos \beta = \vec{r} \cdot \vec{v}$... dot product of unit vectors

C_D ... diffuse color (RGB)

h ... glossiness / specularity (5 ... 500)

k_A ... ambient coefficient (0.0 to 1.0)

$$E_d = I_i \cdot C_d \cdot k_d \cdot \cos \alpha$$

I_i ... light source intensity

C_d ... diffuse color (RGB)

k_d ... diffuse coefficient (0.0 to 1.0)

$\cos \alpha = l \cdot n$... scalar product of light direction and surface normal

Pokud mám pozorovatele a světlo v nekonečnu (tj. directional světlo a paralelní projekce), můžu si to zjednodušit tím, že beru vektor pozorovatele a vektor světla jako konstantu. Můžu pak approximovat za použití half vektoru na čtvrtou, čemuž se říká **Blinn-Phong model**. Což si teda vůbec nezaslouží, protože tomu Blinn fakt nic nepřidal, jenom tomu dodal ze základní algebry co se stane když se ti nemění v a h.

If both previous conditions are met, we can use $(\vec{h}_i \cdot \vec{n})^{4h}$
instead of $(\vec{r}_i \cdot \vec{v})^h$

Half-way vector $\vec{h}_i = (\vec{l}_i + \vec{v}) / |\vec{l}_i + \vec{v}|$

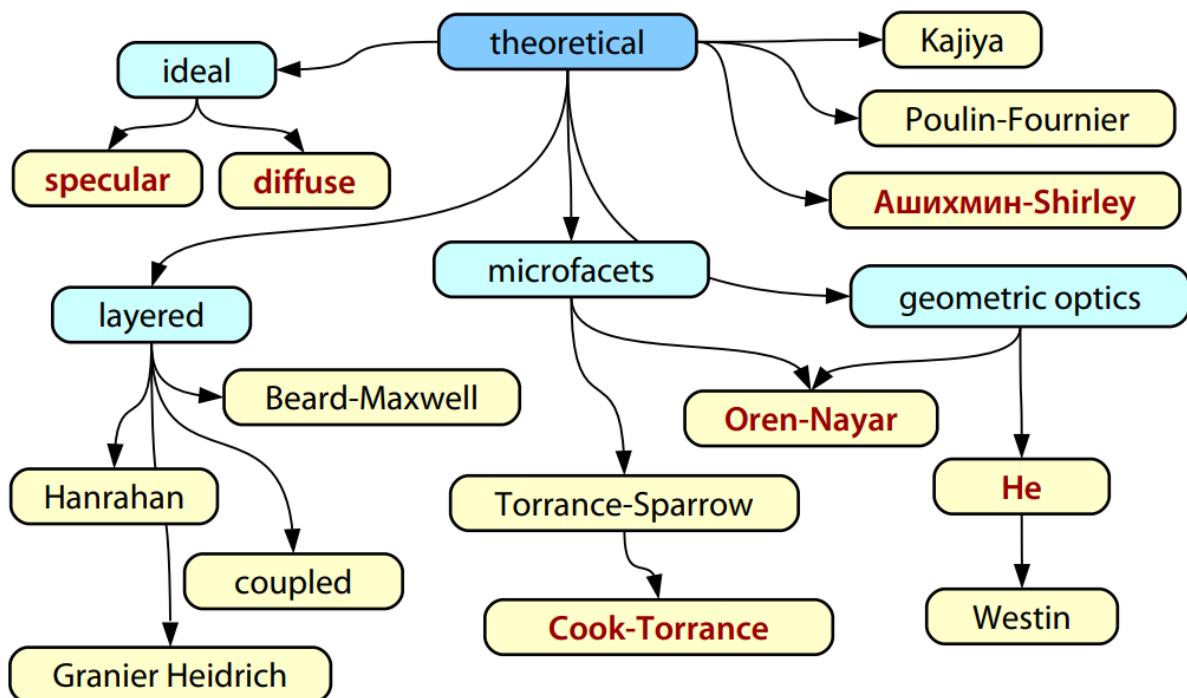
– \vec{h}_i is constant everywhere

Sometimes this simplification is called „Blinn-Phong model“

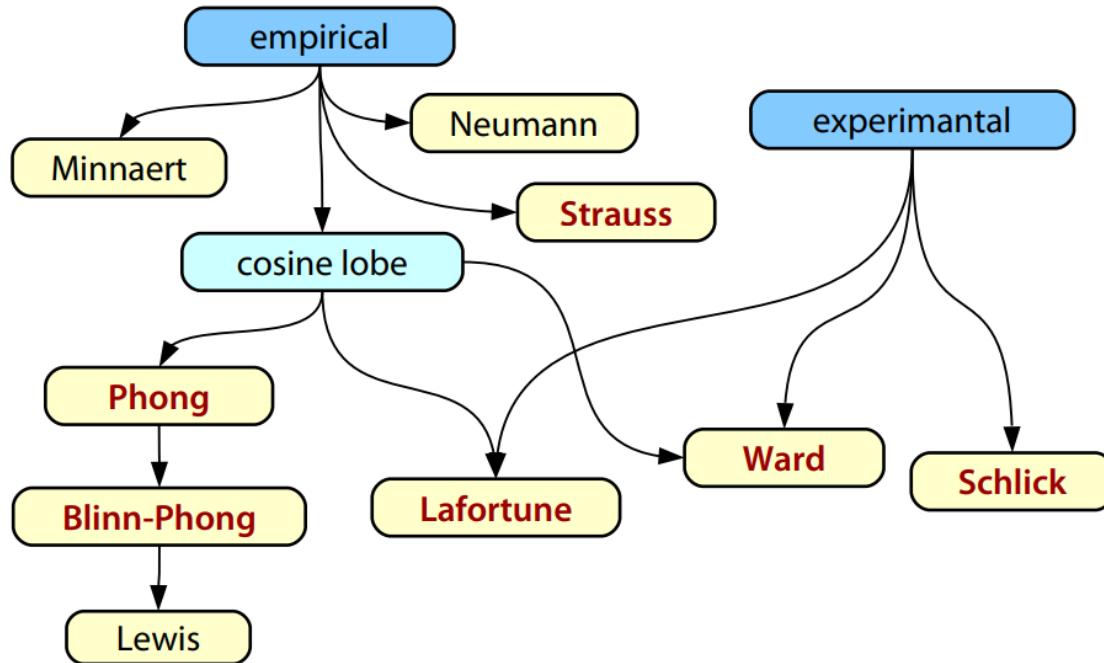
Takže nemusím počítat Reflection vektor, a ve spektrální složce použiju místo $\cos^h \beta = (r * v)^h$ dám $\cos^h(\beta) = (\vec{h} * \vec{n})^{4h}$

Jako další možností je physical-based rendering, založené na BRDF. Tam už je ale více možností jak ho počítat. Existují následující modely, který se dělí na teoretický a empirický:

BRDF ZOO I



BRDF ZOO II



Další možností je **direct light** algoritmus, kdy si integrál po polokouli z rendering equation (vykreslovací rovnice), což je ten největší problém, prostě změním na sumu, a sčítám jenom přímo odrazy od zdrojů světla, a ignoruji odrazy ve scéně

Stínovací algoritmy určujou, jakým způsobem sou ve výsledku obarvovány polygony.

Nejčastější jsou tyhle tři:

- Flat shading
 - Vezmu normálu polygonu a spočítám jeho normálu v těžišti (nebo jí mám uloženou), a v ní spočítám barvu podle osvětlení. Tou barvou pak obarvím úplně celej polygon.
 - Je to rychlý, protože nemusím počítat osvětlení pro každý pixel, ale stačí jenom pro každou stěnu jednou.
- Gouraud shading
 - Spočítám si normálu v každém vertexu tak, že vyrobím vážený průměr normál všech ploch se kterejma bod sousedí. Potom si v každém vertexu spočítám barvu modem osvětlení, a když pak dopočítávám barvu pixelu na stěně tak bilineárně interpoluji mezi barvami. To funguje zhruba tak, že jedu po scanlinách, tam kde mi scanlina protne hrany polygonu si spočítám barvu na hraně z barev dvou vrcholu co spojuje, a pak si jednotlivý pixel spočítám podle těch dvou nových bodů:

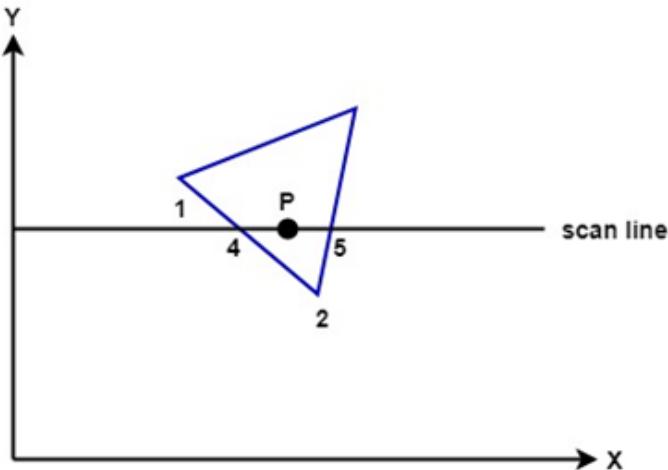
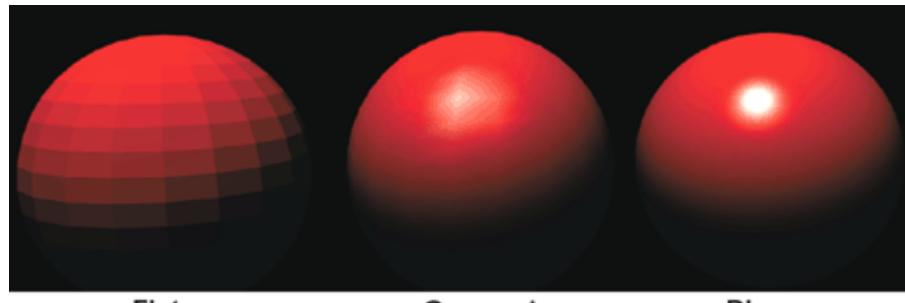
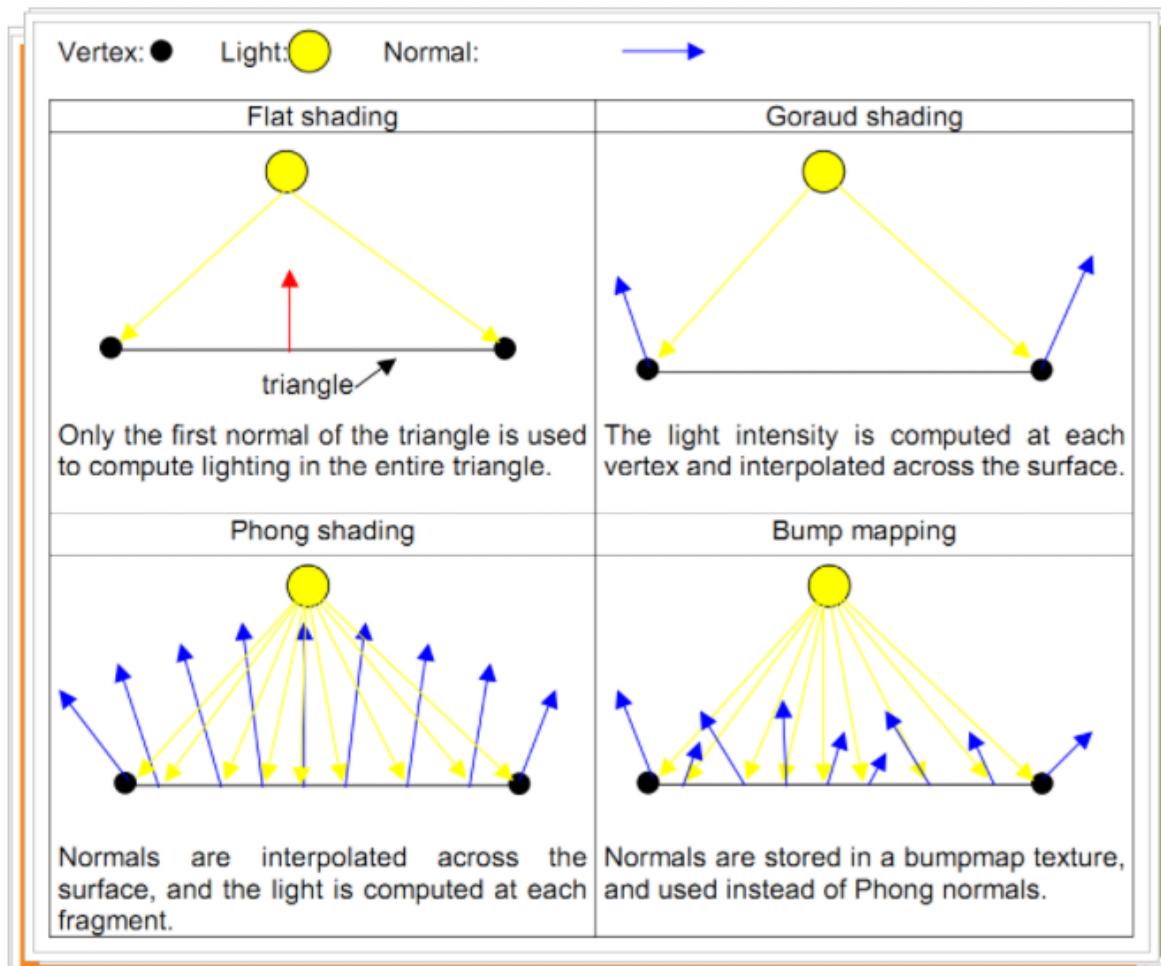


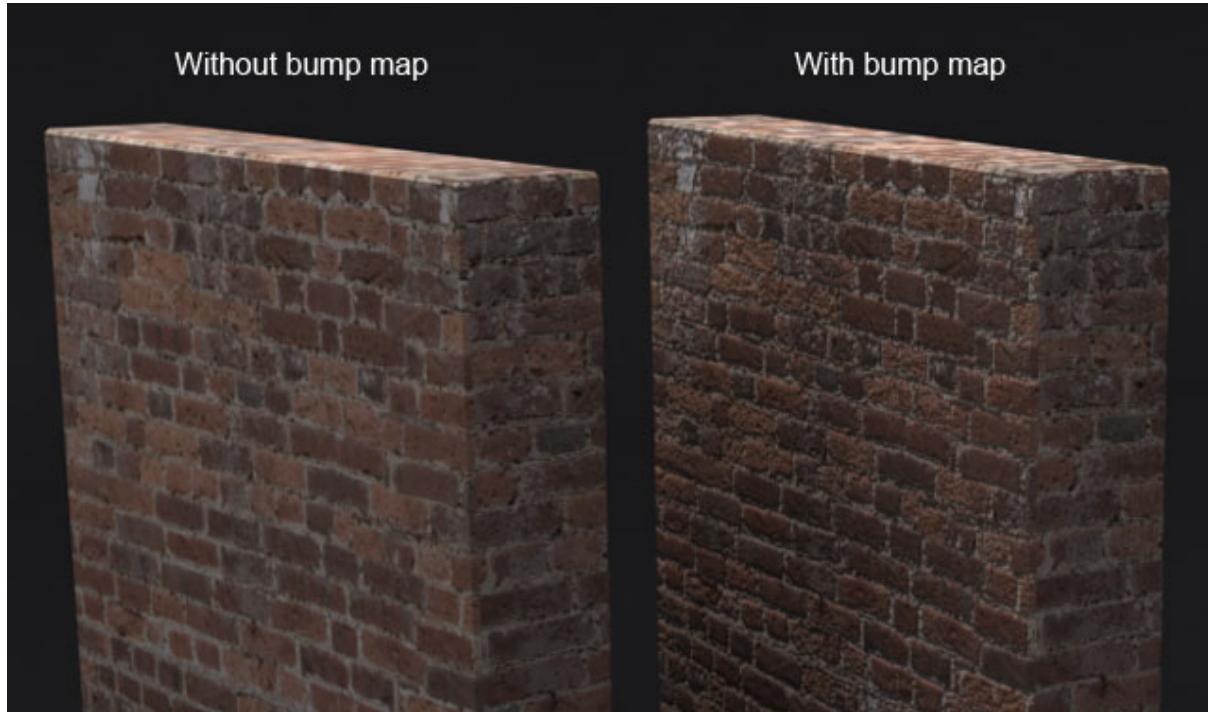
Fig: For Gouraud Shading, the intensity at point 4 is linearly interpolated from the intensities at vertices 1 and 2. The intensity at point 5 is linearly interpolated from intensities at vertices 2 and 3. An interior point P is then assigned an intensity value that is linearly interpolated from intensities at position 4 and 5.

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2$$

-

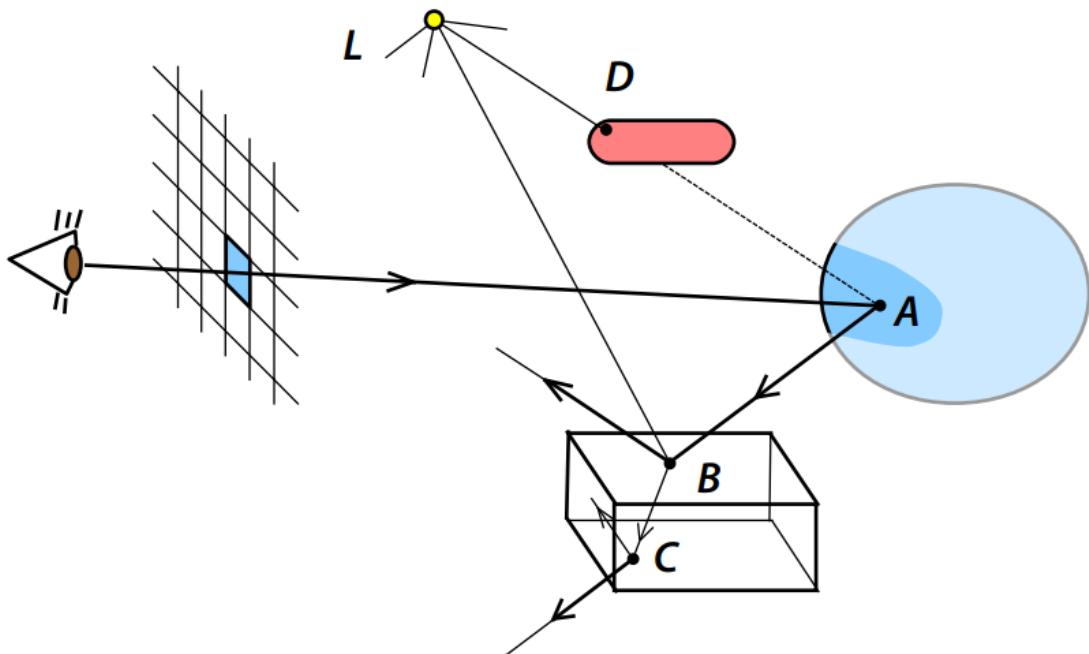
- Phong shading
 - Normály si interpoluji pro každý pixel, a počítám světlo každým pixelu zvlášť.
Vypadá to pak líp, ale zas musím výrazně víckrát počítat světlo.
- Bump/Normal mapping
 - Podobný jako phong shading, akorát mám bumpmapu, tj texturu která mi určuje směr normály v daném bodě.
 - Bump mapa je v grayscalu, a určuje jenom rozdíl výšky bodu, podle kterého si pak spočítáš normálu na základě jeho okolí.
 - Normal mapa je v RGB a máš přímo uložený vektor v daném bodě.



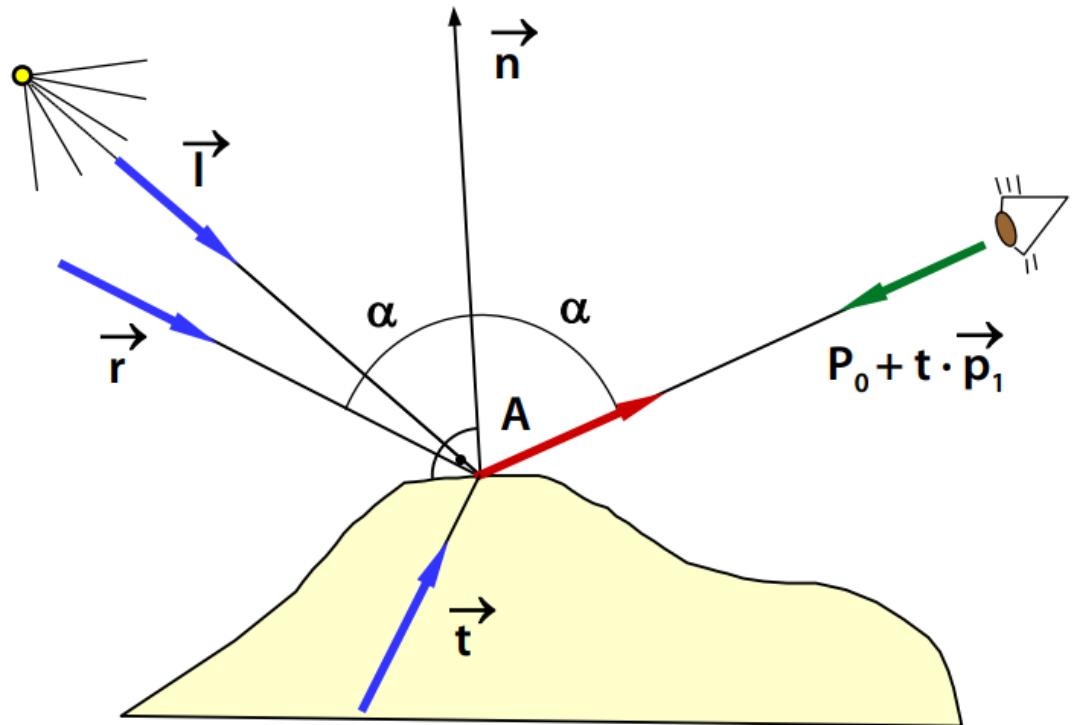


6.6. Rekurzivní sledování paprsku (Raytracing)

Idea je taková, že paprsky nesleduju od světla, protože je fakt malá šance, že se mi trefí do pozorovatele, ale udělám to opačně - začnu od oka, dorazím na předmět, a z něj si spočítám paprsek ke světlu. Ale taky rekurzivně pokračuju dál, a kouknu kam dál se odrazí, a pokračuju rekurzivně.



Tj když počítám jeden bod, podívám se takhle:



A spočítám teda kolik mi posvítí světlo na bod A, třeba Phongem, a pak se rekursivně zavolám a přičtu k tomu kolik světla přijde ze směru r a t (odraz a při průhlednejsch materiálu i transmision zevnitř). Protože počítání světla z A do oka se počítá stejně jako z r do A, tak to prostě můžu volat rekursivně.

Rekurzivní implementace



```
int maxDepth = 10;           // Maximum recursion level.  
RayScene scene;             // Global scene object (geometry, materials, light sources...)  
  
RGB shade (Vector3d p0, Vector3d p1, int depth)  
    // p0 - ray origin, p1 - ray direction, depth - interactions so far  
{  
    Vector3d A = intersection(scene, p0, p1);  
    if (!isValid(A)) return scene.background; // No intersection at all.  
  
    RGB color{0};           // Result color.  
    for (const auto& light : scene.lightSources)  
        if (!isValid(intersection(scene, A, light.point - A)))  
            color += scene.kL(A) * light.contribution(A, -p1, scene.material(A), scene.normal(A));  
  
    if (++depth >= maxDepth) return color;  
  
    if (scene.isGlossy(A))      // Recursion - reflection.  
    {  
        Point3d r = reflection(p1, scene.normal(A));  
        color += scene.kR(A) * shade(A, r, depth);  
    }  
  
    if (scene.isTransparent(A)) // Recursion - refraction.  
    {  
        Point3d t = refraction(p1, scene.normal(A), scene.index(A));  
        color += scene.kT(A) * shade(A, t, depth);  
    }  
  
    return color;  
}
```

Protože teoreticky můžu rekurzivně počítat donekonečna, tak je potřeba to nějak omezit, většinou se k tomu používá hloubka, tj třeba max 10 odrazů. Jak to je po částech: (Kód je jenom pseudokód)

P0 je pozorovatel, p1 je směr, hloubka je pomocná co počítá kolikátý počítám odraz.

Podívám se kam mi paprsek doletí. Pokud nedoletěl nikam, prostě vrátím barvu pozadí scény:

```
Vector3d A = intersection(scene, p0, p1);  
if (!isValid(A)) return scene.background; // No intersection at all.
```

Pokud někam narazil, pošlu paprsek z toho bodu do všech světel ve scéně, podívám se jestli na světlo vidí (tj není ve stínu), a spočítám třeba Phonga, tj jak ten bod vybarví světla:

```
RGB color{0};           // Result color.  
for (const auto& light : scene.lightSources)  
    if (!isValid(intersection(scene, A, light.point - A)))  
        color += scene.kL(A) * light.contribution(A, -p1, scene.material(A), scene.normal(A));
```

Pokud jsem už v limitu odrazů, tak tu barvu vrátím: (a taky si zvětším depth o +1, abych pak dál na to nezapomněl)

```
if (++depth >= maxDepth) return color;
```

Pokud je materiál glossy, tj aspoň trochu zrcadlovej, tak odrazím paprsek a rekurzivně znova zavolám funkci shade, akorát s větší hloubkou, a výsledek uložím do barvy (tj tam pak mám barvu z odrazu, i ze světel. kR je konstanta která nějak škáluje příspěvek z odrazu, v závislosti na materiálu:

```

if (scene.isGlossy(A))      // Recursion - reflection.
{
    Point3d r = reflection(p1, scene.normal(A));
    color += scene.kR(A) * shade(A, r, depth);
}

```

A nakonec, pokud je materiál i průhlednej, tak si spočítám na základě indexu lomu materiálu barvu která prochází skrz ten předmět, tj prostě pošlu další paprsek skrz a zase zavolám rekurzivně funkci shade a spočítám, jak přispěje výsledku:

```

if (scene.isTransparent(A)) // Recursion - refraction.
{
    Point3d t = refraction(p1, scene.normal(A), scene.index(A));
    color += scene.kT(A) * shade(A, t, depth);
}

return color;

```

A nakonec vrátím barvu.

Řešení hloubky rekurze, tj jak počítat kdy mám skončit.

Je na to několik přístupů:

- Staticky, tj zastavím po 10 odrazech.
- Dynamicky - mám nějakou škálu významu, třeba 100-0 procent, a s každou rekurzí tu škálu zvednu podle toho, jak moc ten paprsek přispívá výsledku.
- Kombinovaně - kombinuju předchozí přístupy.

Nejjednodušší je asi ten dynamický. Funguje to tak, že místo toho, abych hloubku zvýšil konstantě o +1, tak začnu na 100%, a vždycky když volám rekurzi, tak jí zmenším o koeficient kterej určuje jak moc ten paprsek přispívá výsledku. V algoritmu nahoře to jsou konstanty scene.kR(A) a scene.kT(A) (což budou čísla co se nasčítají na 1-scene.kL(A), což je příspěvek světla), a budou v rozmezí 0-1. A zastavím rekurzi v momentě, kdy je příspěvek dost malej (třeba pod 2%).

To vede k tomu, že pokud mám třeba materiál kterej je super odrazivej a trochu průhlednej, takže z odrazu přispěje 95% barvy pixelu a z průhlednosti jenom 5%, tak skrz průhlednost nemusím počítat tak daleko, protože se to stejně neprojeví. Tj transparentností půjdou třeba 2-3 hloubku rekurze (podle toho jak maj koeficienty ty místa kam pak dorazí), ale odrazem jich udělám celejch 10.

A u kombinovaného přístpu si počítám i počet odrazů, a když tak to zastavím po X odrazech, kdyby se ten význam moc nesnížuje. Tj třeba když v muzeu zrcadel pošlu odrazy, tak se význam nebude snižovat skoro vůbec (protože kT bude 0, kL bude třeba 0.03, a kR bude 0.97) a odrážel bych se donekonečna.

6.7. Fyzikální model šíření světla (radiometrie, zobrazovací rovnice)

<https://www.youtube.com/watch?v=j-A0mwsJRmk> dobrá přednáška o tom.

Důležitou součástí počítání radiometrie je pochopit, co je prostorovej úhel (solid angle). Je to v podstatě přenesení radiánů do 3. Rozměru. Co určuje radián? Jeden radián je středový úhel, který přísluší oblouku o stejné délce, jako je poloměr kružnice. Což zní hrozně debilně,

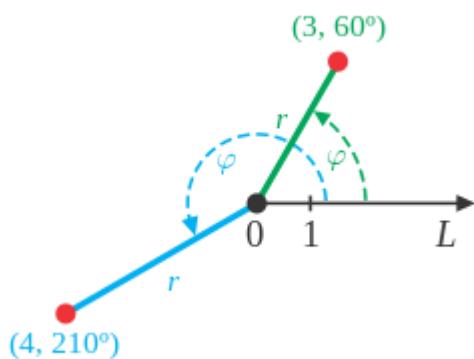
ale jde o to, že když si vezmu poloměr kružnice, a začnu ho obmotávat kolem té kružnice, tak to, kolikrát se na kružnici vejde aby obepsal velikost toho úhlu, tak tolik radiánů to je.

Animace to vysvětlí naprosto jednoduše:

<https://www.mathsisfun.com/app.html?folder=geometry&file=radian&p=rad>

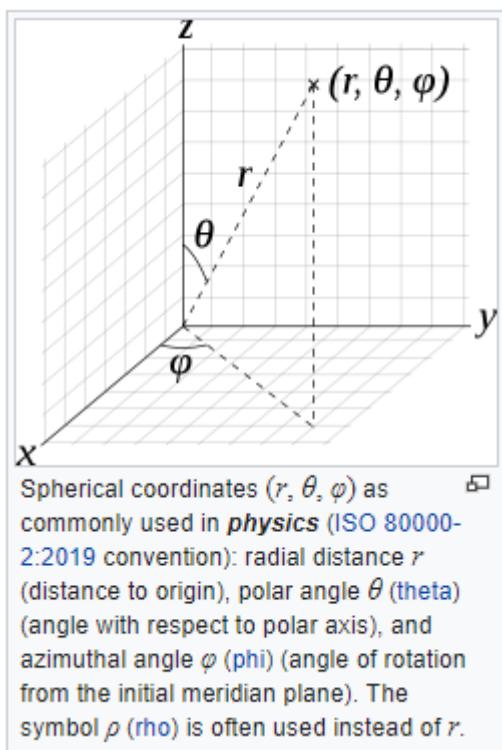
Pokud mám radiány třeba 2, tak prostě ten poloměr nalepím 2x za sebe, a vezmu úhel co to popisuje. Proto je 180 stupňů Pi radiánů. V podstatě jde o to, že radián je poměr délky poloměru kružnice k obvodu co úhel vykrájí na obalu kružnice - $r = \theta r$

No, a teď se dostáváme k solid angle, prostorovým úhlům. První co musím popsát jsou **polární souřadnice**. Používaj se k popsání souřadnic na základě vzdálenosti a úhlů. Ve 2D je to easy:



mám prostě úhel a vzdálenost.

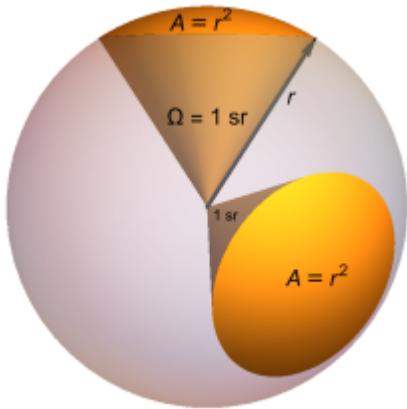
Ve 3D k tomu akorát přidáme jeden úhel, a popisuje to:



Jakmile mám polární souřadnice, můžu se podívat, co je prostorové úhel. Je to v podstatě 3D radián - poměr obsahu plochy co promítne kužel na kouli, značí se Omega:

$$\Omega = \frac{A}{r^2}$$

, kde r je poloměr koule, a A je plocha co promítne:



Říká se tomu seradián, a používá se hodně právě v radiometrii, jako jednotka - třeba při definici luminence se používá jeho derivace, aby se určilo kolik světla vyzáří na nekonečně malej bod.

Derivace seradiánu se ve sférickejch souřadnicích počítá takhle:

$$d\Omega = \sin \theta d\theta d\varphi$$

kde Theta a Phi sou souřadnice ve sférickejch (polárních) souřadnicích směru, kterým směruje, a dTheta a dPhi je právě ten nekonečně malej úhlík.

Fyzikální veličiny

Základem radiometrie je právě popisování světla a jeho šíření/chování.

Z pohledu fyziky jsou důležitý následující pojmy:

$$E = \frac{hc}{\lambda}$$

Where

E is photon energy (Joules),

λ is the photon's wavelength (metres),

c is the speed of light in vacuum - 3×10^8 metres per second

h is the Planck constant - $6.62606957 \times 10^{-34} \text{ (m}^2\text{kgs}^{-1}\text{)}$

Energie fotonu -

Radiantní energie určuje energii, kterou mají fotony v určitém jednom bodu. Počítá se jednoduše jako integrál přes všechny vlnový délky - $Q = \int_{0-\infty} n(l) * e(l) dl$

Kde $n(l)$ je počet fotonů s vlnovou délkou l a $e(l)$ je energie fotonu délky l (E vejš). Prostě vezmu všechny vlnový délky a počty fotonu s tou délkou a sečtu je (integruju)

Zářivý výkon - značí se Φ , a rovná se $\Phi = Q/t$, tj radiantní energie na jednotku času Nerozlišuje se, jestli je to výkon co přichází (incident) nebo se je vyzařován (reflected), proto se odlišují značením **Phi** a **Phi**

Irradiance - Tok, co dopadne na nějakou plochu, se počítá jako $E = d\Phi / dA$, kde A je obsah plochy

Zářivá intenzita - Kolik vyzařuje světla v daném prostorovém úhlu. Je to hustota světla v oblasti prostorového úhlu, a $I = d\Phi / dw$, kde w je prostorový úhel.

Nejdůležitější veličinou je ale **radiance** - udává přijímaný či vyzářený výkon na jednotkovém prostorovém úhlu na jednotku kolmo promítnuté plochy. Dá se to představit tak, že mám bod, ze kterého vychází světlo (at' už odrazem nebo jinak), a zajímá mě kolik fotonů přichází nebo je vyzářeno v určitém směru $d\Omega$ za jednotku času procházející průmětem diferenciální plošky dA který je kolmý na ten směr:

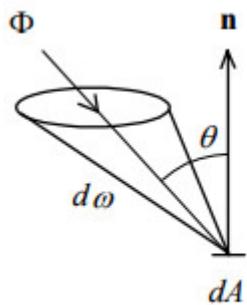


Figure 3a

Radiance (arriving)

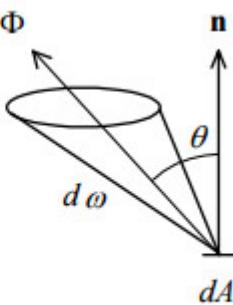


Figure 3b

Radiance (leaving)

(na obrázku je dw místo omegy)

$$L_e(\Omega) = \frac{d^2\Phi_e}{dA d\Omega \cos \theta} = \frac{I_e(\Omega) E_e(A)}{\cos \theta}$$

kde

L je zář (W na m^2 na steradián),

Φ_e je radiometrický zářivý tok zdroje (W),

θ je úhel mezi kolmici na uvažovanou rovinu a měřeným směrem,

A je plocha povrchu (m^2),

Ω je prostorový úhel (steradián).

$I_e(\Omega)$ je zářivost.

$E_e(A)$ je ozářenost.

Z téhle rovnice se dá odvodit všechny ty předchozí:

- Radiantrní energie - integrál radiance přes čas T, polokouli Omega nad bodem x přes všechny body plochy A
- Zářivý tok - integrál přes polokouli Omega přes všechny body plochy A
- Irradiance je zase integrál přes polokouli.
- Intenzita je zase integrace přes plochu A

Nejdůležitější pojem z toho všeho je BRDF (Bidirectional reflectance distribution function), což je označení pro obousměrnou distribuční funkci odrazu světla. Pro matematické vyjádření vlastností povrchu. Udává subkritickou hustotu pravděpodobnosti (její integrál smí být menší než 1), že se světlo, které na povrch dopadne, odrazí daným směrem. Parametry funkce jsou příchozí (w_i) a odchozí (w_o) směr, oba definované vůči normále povrchu.

Návratová hodnota funkce se udává ve sr⁻¹ a vyjadřuje poměr odražené diferenciální záře (radiance) vůči ozáření (irradiance) povrchu. Je to teda rozdelení pravděpodobnosti. V podstatě to říká, když na tenhle bod dopadne světlo v úhlu ω_i , tak má každé foton šanci, že se odrazí do směru ω_o , $f(x, \omega_i \rightarrow \omega_o)$. Tj statisticky vzato se mi odrazí $f(x, \omega_i \rightarrow \omega_o)$ procent paprsků.

BRDF

$$f_r(\mathbf{x}, \omega_i \rightarrow \omega_o) = \frac{dL_r(\omega_o)}{dE(\omega_i)} = \frac{dL_r(\omega_o)}{L_i(\omega_i) \cos \theta_i d\omega_i} [sr^{-1}]$$

Kde

$dL_r(\omega_o)$ značí odraženou diferenciální zář (radiance) [$Wm^{-2}sr^{-1}$],

$dE(\omega_i)$ diferenciální ozáření povrchu (irradiance) [Wm^{-2}],

$L_i(\omega_i)$ je zář dopadající ze směru ω_i a

θ_i odpovídá sklonu dopadajícího světla od normály.

Má několik vlastností co by měla dodržovat - reciprocity (v obou směrech je stejná, tj když se prohodí světlo a pozorovatel výsledek se nezmění), a nesmí porušovat fyzikální zákon, tj pro všechny směry světla platí, že když sečtu kolik se celkem světla odrazí z jednoho směru do všech směrů (tj integruju v místo I), tak součet není větší než 1 - prostě se neodrazí více než 100% světla.

Tím získávám zásadní člen do Zobrazovací rovnice. Hodnota BRDF je v podstatě konstanta FUNKCE (co se zjišťuje různýma metodama, třeba skládáním metalických barev u kterých víme, jaká mají hodnotu, nebo měřením různýma přístrojema), a tu pak používáme k počítání barvy pixelu

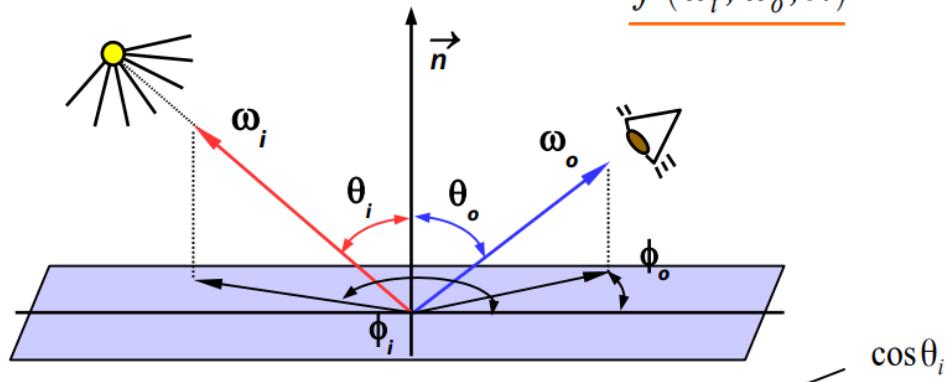
Lokální osvětlovací model

$$L_r(x, \omega_r) = \int_{H^2} f_r(x, \omega_i \rightarrow \omega_r) L_i(x, \omega_i) \cos \theta'_i d\omega_i$$

L_r (odražené světlo do směru ω_r) se rovná integrál přes všechny směry v polokouli, a pro každé spočítám BRDF * L_i (intenzita příchozího světla z toho směru), zmenšenou úhlem ze kterého přichází:

BRDF function: $\mathbf{R}^5 \rightarrow \mathbf{R}$

$$f(\omega_i, \omega_o, \lambda)$$



$$\underline{L_o(\omega_o)} = \int_{\Omega} \underline{f(\omega_i, \omega_o)} \cdot \underline{L_i(\omega_i)} \cdot \underline{(n \cdot \omega_i)} \frac{d\omega_i}{\cos \theta_i}$$

- Lr počítám, Li a Thetai se integrují, tj vezmu všechny směry z celé polokoule.

Tím se dostaneme k **zobrazovací rovnici** (rendeing equation):

$$L(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_H L(r(\mathbf{x}, \omega_i), -\omega_i) \cdot f_r(\mathbf{x}, \omega_i \rightarrow \omega_o) \cdot \cos(\theta_i) d\omega_i$$

Kde

$L(\mathbf{x}, \omega_o)$ značí celkovou vyslanou **zář** (radiance) z bodu \mathbf{x} podél paprsku ve směru ω_o ,

$L_e(\mathbf{x}, \omega_o)$ značí **zář** (radiance) emitovanou zdrojem z bodu \mathbf{x} ve směru ω_o ,

H hemisféru ve směru normály se středem v \mathbf{x} , přes kterou integrujeme (viz obr.),

ω_i směr příchozího paprsku

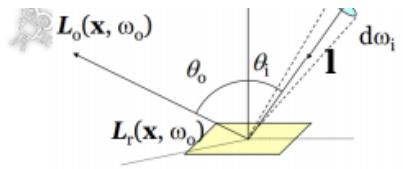
θ_i je velikost úhlu mezi ω_i a normálou plochy

$f_r(\mathbf{x}, \omega_i \rightarrow \omega_o)$ je distribuční funkce odrazu (BRDF) v bodě x ze směru ω_i do ω_o .

Z toho plyne pár věcí - pokud mám dokonale difuzní materiál, tj ze všech směrů vypadá stejně protože světlo rovnoměrně rozptýlý do všech směrů, bude BRDF $f_r(x) = \text{konst}$, a můžu si L napsat jako

$$L(x, \omega_0) = f_r(x) * L_e(x, \omega_0).$$

Lambertian BRDF model



- Assumption: perfectly diffuse material independent on view vector

$$f(x, \omega_i \rightarrow \omega_o) = \frac{\text{albedo}}{\pi}$$

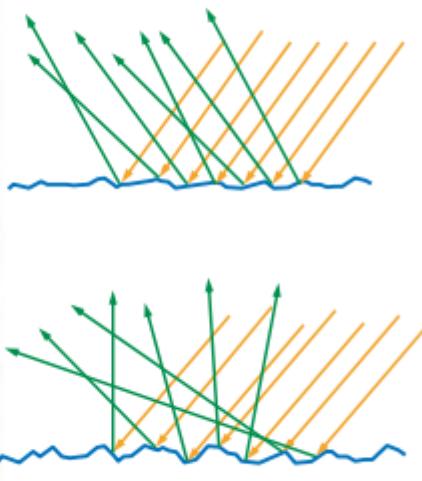
Derivation <https://sakibsaikia.github.io/graphics/2019/09/10/Deriving-Lambertian-BRDF-From-First-Principles.html>

- For single light

$$\begin{aligned} L_r(x, \omega_o) &= \int_{\omega_i \in H(x)} L_i(x, \omega_i) \cdot \frac{\text{albedo}}{\pi} \cdot \cos \theta_i d\omega_i = \\ &= \frac{\text{albedo}}{\pi} L_i \cdot \cos \theta_i = \frac{\text{albedo}}{\pi} L_i \cdot n \cdot l \end{aligned}$$

(Albedo se dělí π proto, aby se zachovala nerovnost že odrazím do všech směrů v součtu mří nebo stejně světla jako jsem přijal. Kdyby tam π nebylo, mohlo by se stát, že odrazím víc.)

Tohle všechno ale počítá s tím, že je materiál perfektně rovná plocha, což bohužel není. Povrch je totiž složenej z microfacets, což jsou miniaturní různě nakloněný plošky, který vlastně skládají ten povrch:



Což popisuje difuzní složku světla, tj světlo co jde do všech směrů:

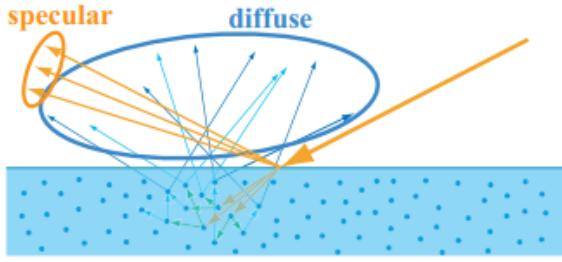
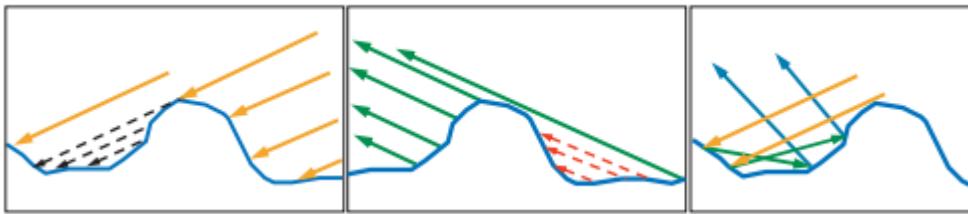


Figure 21: BRDF specular terms are typically used for surface reflection, and BRDF diffuse terms for subsurface scattering. (image from "Real-Time Rendering, 3rd edition" used with permission from A K Peters).

Počítá se relativně složitě, protože se může stát, že né všechny odrazy dojdou k povrchu - může je blokovat jiný plošky v cestě:



Celkové se to celý odhaduje zase statisticky, a vzoreček je na to tenhle:

$$f_{\mu\text{facet}}(\mathbf{l}, \mathbf{v}) = \frac{F(\mathbf{l}, \mathbf{h})G(\mathbf{l}, \mathbf{v}, \mathbf{h})D(\mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})} \quad (4)$$

We will go into each of the terms in more detail, but first a quick summary. $F(\mathbf{l}, \mathbf{h})$ is the Fresnel reflectance of the active microfacets as a function of the light direction \mathbf{l} and the active microfacet normal $\mathbf{m} = \mathbf{h}$. $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ is the proportion of microfacets (of the ones with $\mathbf{m} = \mathbf{h}$) which are *not* shadowed or masked, as a function of the light direction \mathbf{l} , the view direction \mathbf{v} , and the active microfacet normal $\mathbf{m} = \mathbf{h}$. $D(\mathbf{h})$ is the microfacet normal distribution function evaluated at the active microfacet normal $\mathbf{m} = \mathbf{h}$; in other words, the concentration of microfacets with normals equal to \mathbf{h} . Finally, the denominator $4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})$ is a correction factor which accounts for quantities being transformed between the local space of the microfacets and that of the overall macrosurface.

Tohle zase určuje distribuci pravděpodobnosti, tj kolik světla se odrazí difuzně. $F(\mathbf{l}, \mathbf{h})$ je fresnel reflectance, což určuje kolik světla odrazí jedna miniploška natočená do směru \mathbf{h} , $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ popisuje kolik procent mikroplošek s normálou (\mathbf{h}) **není** blokováno (viz předchozí obrázek), a $D(\mathbf{h})$ určuje kolik procent mikroplošek má normálu \mathbf{h} .

Tj je to relativně easy, prostě abych zjistil kolik světla se mi odrazí od plošek, tak vynásobím světlo na plošku šancí, že ploška bude vidět a počtem plošek. Akorát pracuju v statistice a né v konkrétních číslech - většina jsou data co seženu třeba tím, že si je prostě změřím u reálných materiálů. Většinou se to celý udává prostě jako konstanta, který se říká difuzní složka - a Phongovi je to přesně ta difuzní složka.

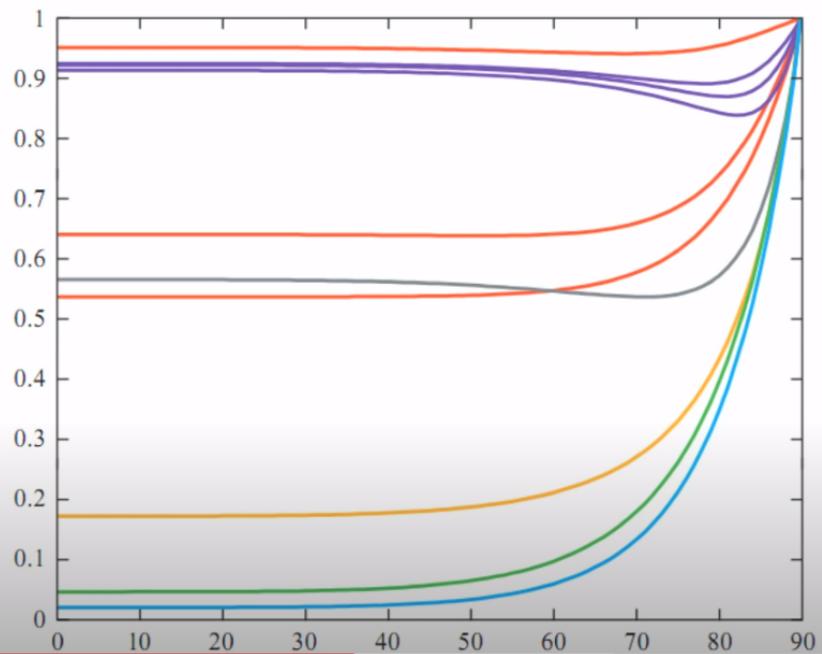
Fresnel reflectance je grafik, který se drží pro většinu úhlů konstatní (na základě materiálu), a potom prudce roste k 1 jak se blížíme k 90 stupňům. Tj, odráží všechno v 90 stupních ale jinak je plus minus stejně. A proto se approximuje touhle hodnotou:

$$F_{\text{Schlick}}(F_0, \mathbf{l}, \mathbf{n}) = F_0 + (1 - F_0)(1 - (\mathbf{l} \cdot \mathbf{n}))^5$$

F_0 je hodnota odrazivosti pro 0 stupňů pro daný materiál.

Reálně pro materiály vypadá nějak takhle:

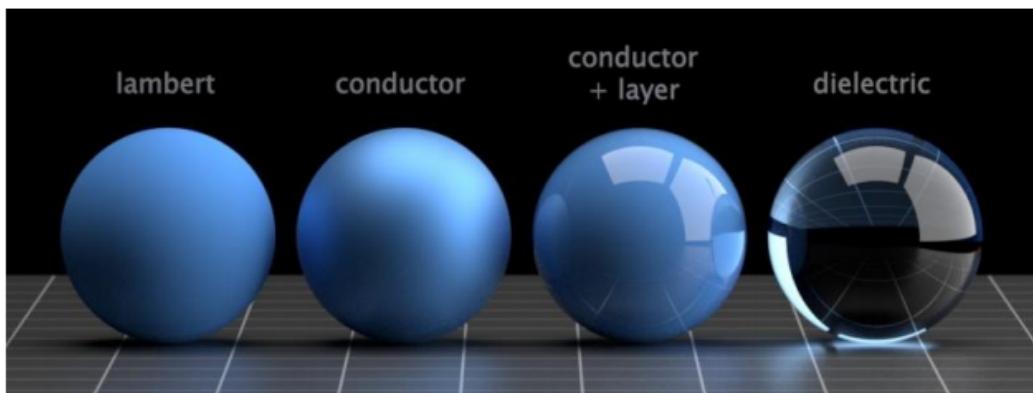
Fresnel Reflectance



Taky to, kolik se odrazí světla, záleží na tom, co za typ materiálu to je:

Dielectrics vs. Conductors

- Dielectrics transmit a portion of incident light, e.g., glass, mineral oil, water and air.
- Conductors transmit only a small portion of incident light, which is quickly absorbed by the material, the rest is reflected, e.g., metals
- Semiconductors have more complex interactions than



6.8. Algoritmus sledování cest (pathtracing)

Globální osvětlení

Tady je o tom hezký počtení -

<https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing/introduction-global-illumination-path-tracing>

Algoritmus sledování cest je něco jiného než raytracing, a používá se k lepšímu počítání globálního osvětlení. Protože raytracing posílá paprsky jenom ve směru odrazu, nepovede se mu zachytit globální osvětlení z ostatních směrů. Pathtracing tohle řeší.

Funguje na principu Monte Carlo metody odhadu hodnoty integrálu, která v podstatě počítá s tím, že když vezmu několik náhodných vzorků funkce $f(x)$ a spočítám jejich střední hodnotu, čímž odhadnu docela dobře přesnou hodnotu integrálu.

Odhaduji integrál $I = \int_{\Omega} f(x) dx$, , a jako primární estimátor si určím

$$F_{\text{prim}} = \frac{f(X)}{p(X)},$$

kde X je libovolná náhodná veličina s hustotou pravděpodobnosti $p(x)$, pro kterou platí

$$\forall x \in \Omega : f(x) \neq 0 \implies p(x) \neq 0.$$

$$E X = \int_R x dP(x).$$

Potom z definice střední hodnoty () platí, že střední hodnota prim. Estimátoru se rovná hodnotě integrálu:

$$E[F_{\text{prim}}] = \int_{\Omega} \frac{f(x)}{p(x)} p(x) dx = I.$$

Co mi z toho plyne - čím víc vzorků si vezmu, tím blíž se dostanu pravý hodnotě integrálu, pokud budu počítat střední hodnotu postupně:

Jelikož použití pouze jednoho vzorku v primárním estimátoru nezaručuje dostatečně malý rozptyl odhadu, používá se estimátor sekundární. Ten využívá N nezávislých náhodných veličin $Y_i = f(X_i)/p(X_i)$ a jako výsledek se vezme jejich průměr, tedy:

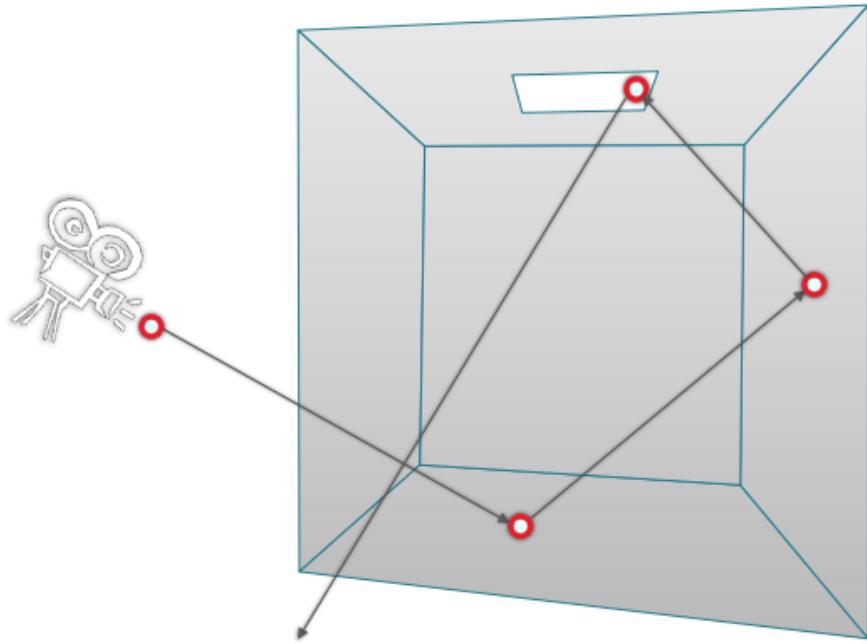
$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}.$$

Tento estimátor je opět nestranný, jelikož platí:

$$E[F_N] = E \left[\frac{1}{N} \sum Y_i \right] = \frac{1}{N} \sum E[Y_i] = \frac{1}{N} NI = I.$$

Toho se dá využít pro path tracing, kdy počítám hodnotu pixelu, kde potřebuji odhadnout integrál v rendering quation, prostě tak, že pošlu náhodně několik paprsků, počkám si kam dopadnou, a z toho poskládám výsledek. Základem je poslání jednoho paprsku:

Basic path tracing



66

Kde pro každej dopad prostě spočítám s rovnoměrnou distribucí náhodnej směr, kterým ho pošlu dál. Takhle pokračuju až dokud nevyletí mimo scénu, nebo dokud neprovedu určitej počet skoků. Základní algoritmus vypadá takhle:

getLi (\mathbf{x}, ω):

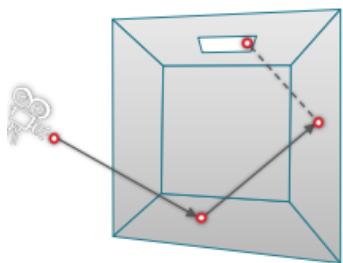
```
 $\mathbf{y} = \text{traceRay}(\mathbf{x}, \omega)$ 
return
     $L_e(\mathbf{y}, -\omega) +$  // emitted radiance
     $L_r(\mathbf{y}, -\omega)$  // reflected radiance
```

Lr(\mathbf{y}, ω):

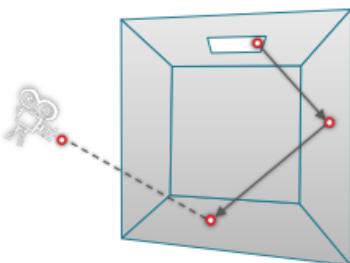
```
 $\omega' = \text{genUniformRandomDir( } \mathbf{n}(\mathbf{y}) \text{ )}$ 
return getLi ( $\mathbf{y}, \omega'$ ) * brdf( $\mathbf{y}, \omega, \omega'$ ) * dot( $\omega', \mathbf{n}(\mathbf{y})$ ) *  $2\pi$ 
```

Existuje několik způsobů jak to počítat:

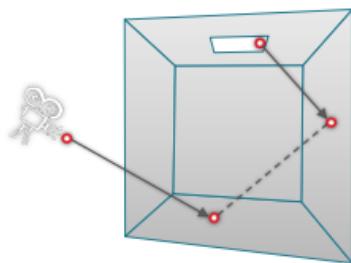
Path tracing



Light tracing



Bidirectional path tracing



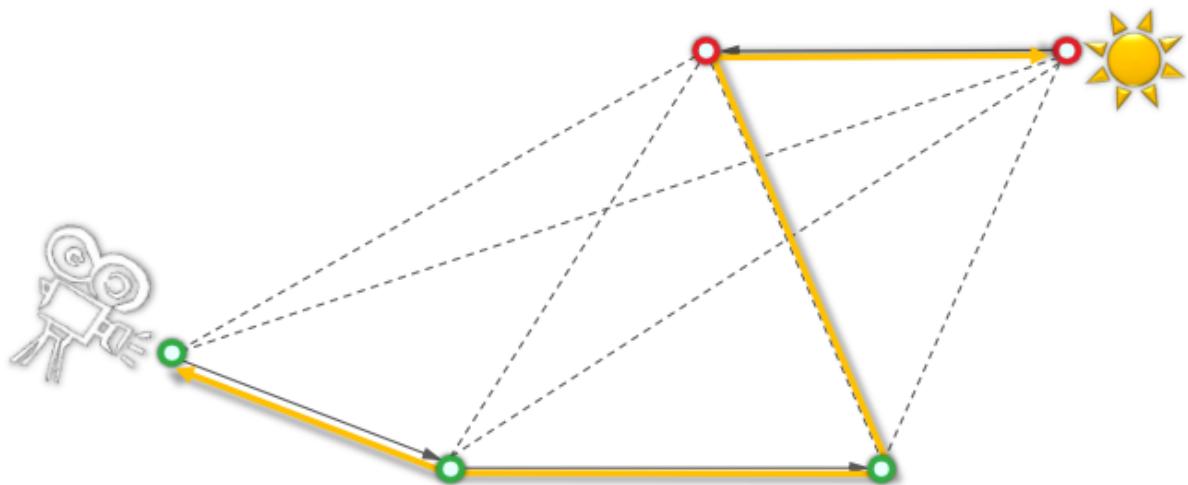
Path tracing a light tracing se liší v tom, že kterého směru posílám paprsek. V originálním popisu path tracingu, a pokud chci mít opravdu reálný osvětlení bez jakýchkoliv biasů, tak musím brát paprsky opravdu náhodně. Což bohužel není moc použitelný, protože většina paprsků nikdy netrefí světlo nebo nějaké jiné emisivní materiál, což mi moc nepomůže. Což se neší Next Event Estimationem - prostě na konci cesty pošlu paprsek né náhodně, ale směrem k důležitým věcem - světlu nebo kameře, podle toho ze který strany jedu. **Už ale to má svoje nevýhody - zaprvé to trochu biasuje, ale hlavně to nepomůže v případě, že mám ve scéně spekulární materiály, protože ty odráží světlo jenom v omezeném úhlu, a může se mi stát, že mi paprsek neodrazí nic.**

To se řeší zase **importance sampling** metodou, kdy si na základě BRDF vygeneruju náhodnou distribuci ze které sampluji pro Monte Carlo tak, aby měli větší pravděpodobnost směry, které více přispívají k výsledné barvě.

Další možností jak řešit problém s netrefováním světel je v každém bodě kam dopadnu poslat jeden rekursivní paprsek náhodně dál, a jeden ke světlu - je to vlastně kombinace raytracingu a path tracingu.

Z každým směrem posílám více než jenom jeden paprsek, většinou se posílají tisíc per pixel. Vezmu pak jejich průměr, který je potřeba vážit distribuční funkcí pravděpodobnosti co jsem použil pro generování náhodného samplu.

Bidirectional path tracing pak funguje tak, že pošlu paprsek jak ze světla tak z kamery, a po určitém počtu bounců je spojím - tj pošlu jejich poslední skok směrem k bodu kde skončil paprsek toho druhého. Tohle můžu dělat jak teprve po konci, nebo ještě líp - spojovat každou dvojici bodů:



Výsledky to má o něco lepší:



BPT, 25 samples per pixel



PT, 56 samples per pixel

Images: Eric Veach

Je tam rozdíl hlavně v tom, že líp zvládá věci jako caustics (když nějaký předmět láme paprsky tak, že padají do jednoho místa, třeba lupa) a spekulární odrazy. U spekulárních povrchů je totiž problém v tom, že odráží světlo jenom z malého rozsahu úhlů, tudíž většina paprsků kterou náhodně pošlu se netrefí do specular lobu, a počítám je zbytečně protože se vůbec neprojeví na výsledku.

6.9. Předpočítané globální osvětlení

Na tohle sem nenašel jedinou přednášku. Vůbec nevím co si pod tím představit, nebo co pod to zařadit.

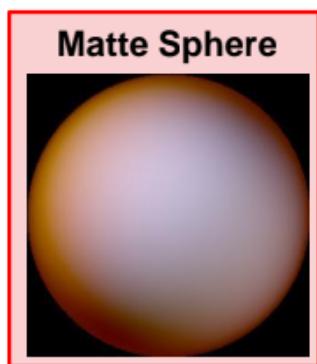
Jedna věc co mě napadla jsou lightmapy - předpočítáš si globální iluminaci na statickou scénu předem, a bakneš jí do light mapy, což je v podstatě textura co má v sobě předpočítanou hodnotu pro GI.

Další možností je použití předpočítaného přenosu radiance nebo stínování aloženého na spherical harmonics, co je popsáný níž. V podstatě je idea taková, že si do pousta míst ve scéně dás Light probes (aspoň tak tomu říká Unity), což je bod pro kterej se ti offline spočítá jak jím **prochází** světlo v daném bodě. To je rozdíl mezi lightmapou - lightmapa počítá jak na danej povrch dopadá světlo, light probe ukládá jjakým světlo prochází prázdným prostorem v daném bodě. To si předpočítám nějakým offline způsobem globální illuminace, a reprezentuju sférickem harmonikama, tj mi z toho padne několik koeficinetů ze kterých můžu jednoduše spočítat funkci, co pro danej vektor vrátí číslo jak moc osvícenej ten bod má bejt. No a to použiju při počítání realtime GI pro dynamický objekty - prostě se podívám na neblížší light probu a na jejím základě spočítám kolk a odkud na postavu má dopadat GI. Samozřejmě to spíš nefunguje s dynamickejma světlama.

Další možností může bejt enviromental map filtering:

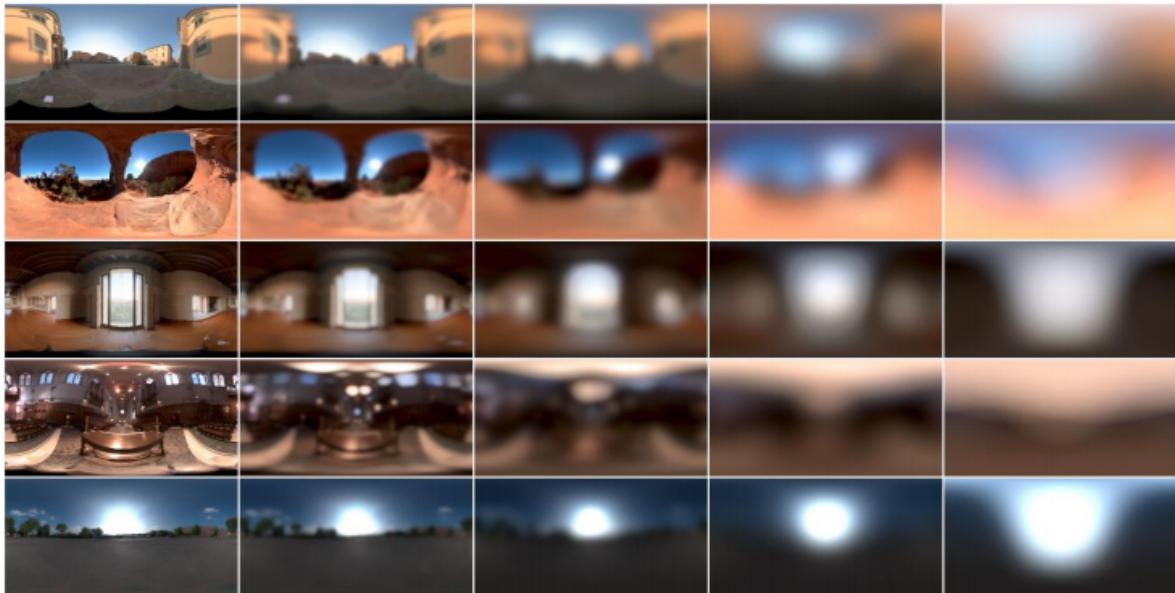
Environment map pre-filtering

- **Phong model** for rough surfaces
 - Illumination function of reflection direction R
- **Lambertian diffuse** surface
 - Illumination function of surface normal N



- Pre-filter (= blur) the EM [Miller and Hoffman, 1984]
 - **Irradiance (indexed by N)** and **Phong (indexed by R)**

Prostě si předem zblurred enviromental map, a pak beru reflections podle toho, jak moc difuzí materiál je



6.10. Výpočet globálního osvětlení v reálném čase

Na tohle sem taky nenašel přednášku. Možná pod to patří spherical harmonics a předpočítanej přenos radiance, ale o tom vůbec žádná s přednášek nemluví, nebo sem jí jenom nenašel.

Jedna z možností je Real-time Voxel Cone Tracing.

<https://wickedengine.net/2017/08/30/voxel-based-global-illumination/>

Idea je taková, že si spočítám quadtree voxelovou 3D volume, kterou postavím podle scény geometrie co mám před sebou zneužitím rasterizační pipeliny, geometry a vertex shaderů, a dál pracuju jenom s ní, místo toho, abych musel řešit posílaní paprsků a další věci přímo ve scéně.

V geometry shaderu si pro každej trojuhelník vyberu, do který z os se promítně nejvíce, aby věděl jak si ho zvoxelizovat. A v pixel shaderu si všechno zvoxelizuju, spočítám si barvu přímýho osvětlení (nejlíp tak, že si pro každý světlo vykreslím kam vidí a voxely na který dopadá jim uložím barvu světla), a uložím si strukturu do 3D textury. (Ještě tam v implementaci dělaj nějaký triky s bufferama, ale to asi není důležitý).

Když se vykreslí voxel grid, vypadá nějak takhle:

<https://turanszkij.files.wordpress.com/2017/08/voxelgi.gif?w=809>

Každej voxel teda obsahuje kolik světla na něj dopadá z kterého zdroje, plus nějaký info navíc (třeba normály, materiál a tak), a pak je potřeba vygenerovat MIP mapy tý 3D textury ve který jsou voxely uložený. Na to třeba DirectX umí přímo funkci, takže to není takovej problém.

No a nakonec pro každej pixel co vykresluju ve scéně pustím cone-ray marching, tj vygeneruju si několik směrů (třeba 5) tak, aby pokryli co největší plochu, a z nich pošlu marchovat voxel po voxelu ten cone. Podle toho jak daleko dorazí, tak takovej level MIPmapy voxel textury budu samplovat, a takhle pokračuju dokud nemám nabráno dost.

6.11. Stínování založené na sférických harmonických funkcích

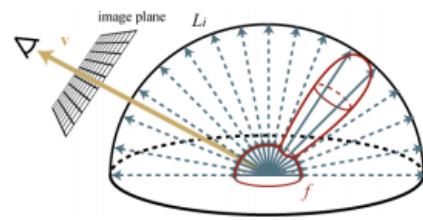
V některých případech se mi hodí mít osvětlení založený na enviromenálnm osvětlení, tj místo flat barvy jako okolního osvětlení použiju nějakou environmental mapu. Třeba si na fotím katedrálu kolem. A chci, aby se mi odrážela v materiálech.

Základní idea je taková, že při stínování pošlu paprsek do environmentální mapy, která obklopuje celou scénu a určuje osvětlení který je v daném bodě:

Shading with Environment Map

- The incident radiance L_i is due to the env. map. emission L_{em} modulated by the EM visibility V_{em}

$$L_i(x, \omega_o) = L_{em}(x, \omega_i) \cdot V_{em}(x, \omega_i)$$



Shading with Environment Map

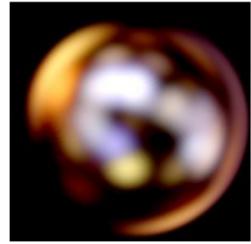
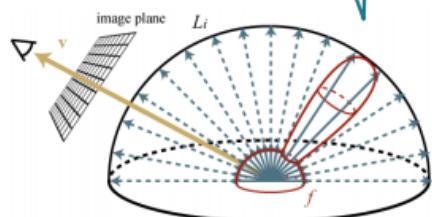
$$L_r(x, \omega_o) = \int_{\omega_i \in H(x)} L_i(x, \omega_i) \cdot f(x, \omega_i \rightarrow \omega_o) \cdot \cos \theta_i d\omega_i$$

1. We integrate over hemisphere.

2. Which is sampling a lot from EM.

3. Weighted according to the material's BRDF.

4. Which result in "blurred" EM reflection.



A revoluční řešení za pomocí sférických harmonik je takový, že si předpočítám ze vzoru 9 parametrů pro sférický harmoniky, a potom pro zjištění iluminace v libovolném ze směru stačí spočítat tu harmoniku, což jde jako jednoduchej polynomiál:

Irradiance approximated by quadratic polynomial

$$E(n) = c_4 L_{00} + 2c_2 L_{11}x + 2c_2 L_{1-1}y + 2c_2 L_{10}z + c_5 L_{20}(3z^2 - 1) + \\ 2c_1 L_{2-2}xy + 2c_1 L_{21}xz + 2c_1 L_{2-1}yz + c_1 L_{22}(x^2 - y^2)$$

$$E(n) = n^t M n$$

4x4 matrix
(depends linearly
on coefficients L_{lm})

Surface Normal vector
column 4-vector

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

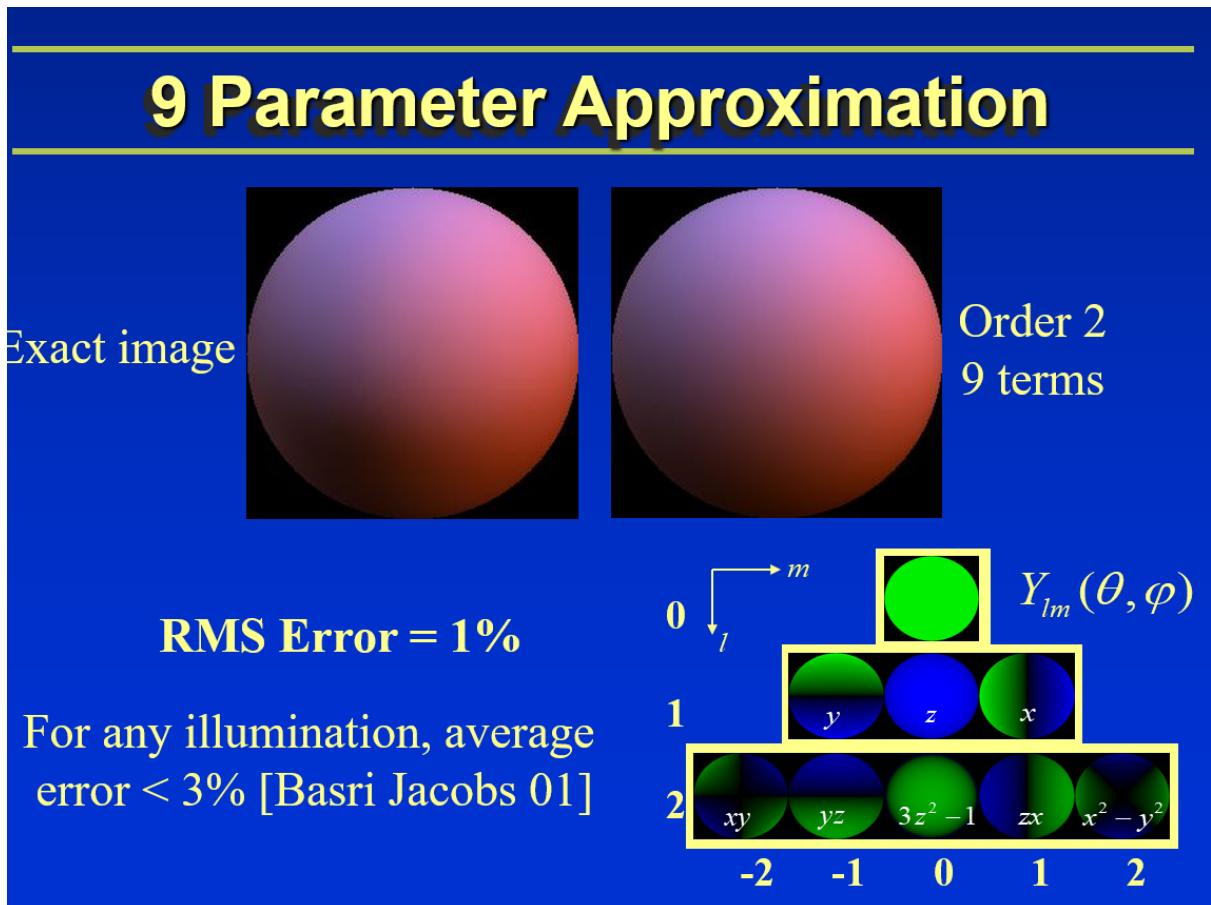
Lmn jsou spočítaný hodnoty/koeficienty co mám předpočítaný níž z enviro mapy, a c jsou pevně daný konstanty, který sou odvozený z převedení spherical harmonics na polynomiály a ještě k tomu nějaký normalizační konstanty, tj se nemění.

$$M = \begin{pmatrix} c_1 L_{22} & c_1 L_{2-2} & c_1 L_{21} & c_2 L_{11} \\ c_1 L_{2-2} & -c_1 L_{22} & c_1 L_{2-1} & c_2 L_{1-1} \\ c_1 L_{21} & c_1 L_{2-1} & c_3 L_{20} & c_2 L_{10} \\ c_2 L_{11} & c_2 L_{1-1} & c_2 L_{10} & c_4 L_{00} - c_5 L_{20} \end{pmatrix}$$

$$c_1 = 0.429043 \quad c_2 = 0.511664$$

$$c_3 = 0.743125 \quad c_4 = 0.886227 \quad c_5 = 0.247708.$$

V podstatě je to fourierova transformace ve 3D, tj místo toho abych si odhadoval skládáním sinusovek, tak skládám sférický funkce:



A jenom si uložím koeficient kolikrát jakou použiju. Sférická funkce dostane jako vstup dva úhly, a vrátí výsledek.

Parametry získám tak, že si prostě spočítám původní integrál polokoule (pro enviro mapu), a zvážím to funkci Y_{lm} (což je báze sferické harmoniky):

Compute 9 lighting coefficients L_{lm}

- 9 numbers instead of integrals for every pixel
- Lighting coefficients are moments of lighting

$$L_{lm} = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} L(\theta, \phi) Y_{lm}(\theta, \phi) \sin \theta d\theta d\phi$$

- Weighted sum of pixels in the environment map

$$L_{lm} = \sum_{pixels(\theta, \phi)} envmap[pixel] \times basisfunc_{lm}[pixel]$$

Horní pulka ukazuje co počítám teoreticky, ten dolní ukazuje co to ve skutečnosti znamená - basisfunc lm je bázová funkce spherical harmonky Ylm:

$$\begin{aligned} N_{lm} &= \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} \\ Y_{lm}(\theta, \phi) &= N_{lm} P_{lm}(\cos \theta) az_m(\phi), \end{aligned}$$

Asi líp napsaný takhle:

$$y_l^m(\theta, \phi) = \begin{cases} \sqrt{2} K_l^m \cos(m\varphi) P_l^m(\cos \theta), & m > 0 \\ \sqrt{2} K_l^m \sin(-m\varphi) P_l^{-m}(\cos \theta), & m < 0 \\ K_l^0 P_l^0(\cos \theta), & m = 0 \end{cases}$$

- K ... normalization constant
- P ... Associated Legendre polynomial
 - Orthonormal polynomial basis on $(0,1)$
- In general: $Y_{l,m}(\theta, \varphi) = K \cdot \Psi(\varphi) \cdot P_{l,m}(\cos \theta)$
 - $Y_{l,m}(\theta, \varphi)$ is separable in θ and φ

No, a protože je sférická funkce funkce který dám dva úhly (tj, polární souřadnice vektoru ve 3d), a vrátí mi nějaké výsledky (tj, hondotu env mapy), tak sem tím dal dohromady mega cheap způsob jak si předpočítat a uložit všechny odrazy environmental mapy. Protože v každém bodě je už předpočítaný odraz uplně celý env. Mapy, tj integrál po celý polokouli, což bych jinak musel dělat per pixel.

Nějaká matika a odvození je v poslední kapitole slajdů tady, str 52+:

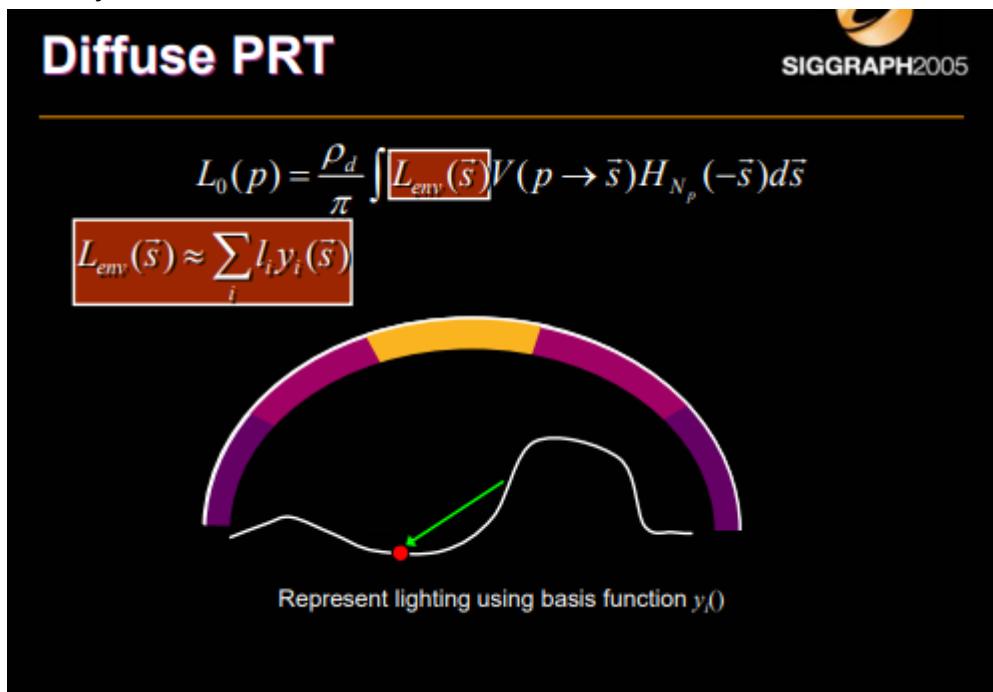
<https://drive.google.com/file/d/1ddSd4p2KqsDOytAR6LxWr0vHs0nFTUYe/view>

6.12. Předpočítaný přenos radience

Předpočítaný přenos radience je v podstatě stejný nápad jako u stínování založeného na sférických harmonikách, akorát si předpočítávám něco jiného.

Místo toho, abych měl uložený koeficienty Environmental mapy podle který řeším okolní osvětlení, tak mám pro každý objekt uložený transfer koeficienty, který určuje, jak přesně se světlo z dané mapy přesune v daném směru. V podstatě mám předem integrovanou visibility a cosine mapu.

Matika je zatím takováhle:



Mám rendering equation pro difuzní matroš (tj, ρ_d je albedo barva), $L_{env}(s)$ určuje světlo přicházející z daného směru, V určuje binárně viditelnost (je/není ve stínu), $H_{N_p}(-s)$ je cosinus příchozího úhlu, resp N^*V z rendering rovnice.

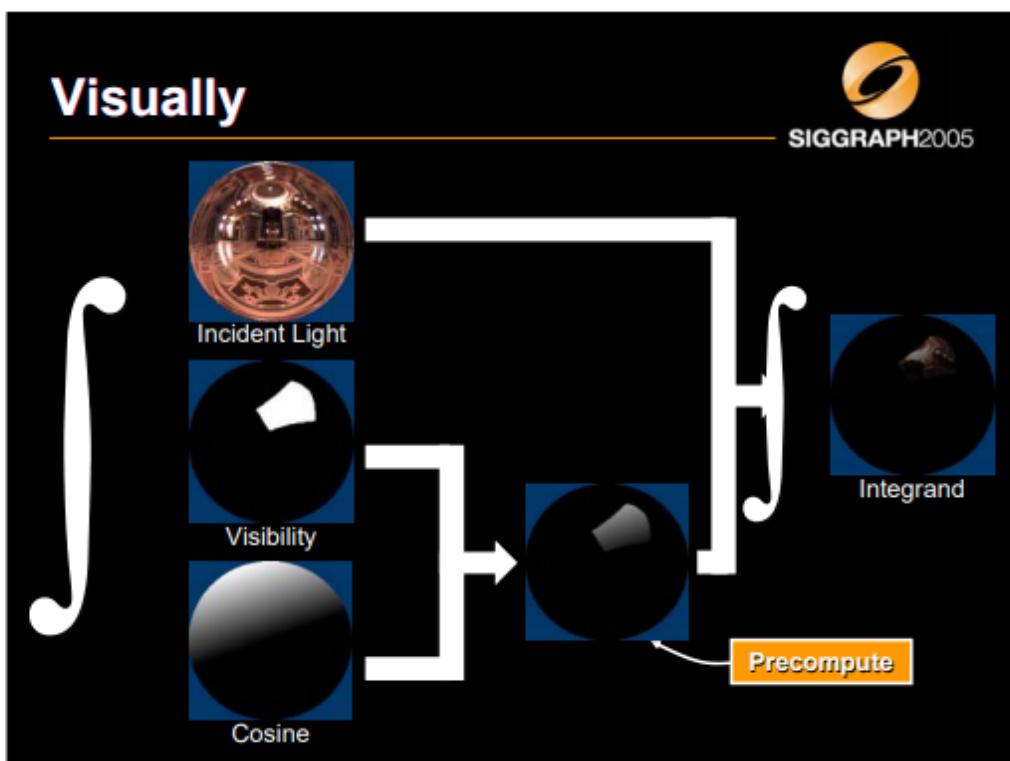
Nejdřív si přepíšu L_{env} , a odhadnu si ho jako součet několika sférických funkcí. Li je koeficient co sem spočítal, a y_i jsou bázové sférické funkce ve kterých to odhaduju (stejný koncept jako u spherical harmonics).

Pak ten odhad hodím místo L_{env} do rendering rovnice, kde můžu koeficienty li vyhodit před integrál, protože suma integrálů je integrál sum:

$$L_0(p) = \frac{\rho_d}{\pi} \int_{\Omega} L_{env}(\vec{s}) V(p \rightarrow \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$

$$L_0(p) = \frac{\rho_d}{\pi} \sum_i l_i \left[\int_{\Omega} y_i(\vec{s}) V(p \rightarrow \vec{s}) H_{N_p}(-\vec{s}) d\vec{s} \right]$$

Tím se mi vyhodí z rovnice ven li (tj první půlka odhadu Lenv funkce), a zůstane mi tam integrál přes bázový funkce odhadu * viditelnost * cosinus. Co je na tom ale super je to, že už nezávisí na světle co do něj přichází (to popisuje suma přes li koeficienty), takže si ho můžu obecně předpočítat dopředu nějakou offline metodou, a pak provádět jenom jednoduchý násobení v momentě, kdy to počítám v realtimu. V podstatě jde o tohle:

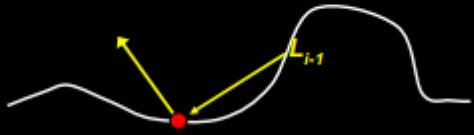


The main trick we are going to use for precomputed radiance transfer (*PRT*) is to combine the visibility and the cosine into one function (*cosine-weighted visibility or transfer function*), which we integrate against the lighting.

Nejlepší na tom je, že tím můžu jednoduchým skládáním počítat víc bounců světla. Rekurzivní výpočet světla si můžu rozepsat jako Neumannovu expanzi:

$$L(p \rightarrow \vec{d}) = L_0(p \rightarrow \vec{d}) + L_1(p \rightarrow \vec{d}) + \dots$$

$$L_i(p \rightarrow \vec{d}) = \int_{\Omega} f_r(p, \vec{s} \rightarrow \vec{d}) [L_{i-1}(p \leftarrow \vec{s}) (1 - V(p \rightarrow \vec{s})) H_{N_p}(-\vec{s})] ds$$



All paths from source that take i bounces

Kde L_0 je přímý světlo, L_1 je světlo co příšlo s jedním bouncem atd...
A když mám transfer funkce, tak se mi to změní v následující výpočet:

$$L(p \rightarrow \vec{d}) = L_0(p \rightarrow \vec{d}) + L_1(p \rightarrow \vec{d}) + \dots$$

$$L(p) = \sum_i l_i [t_{pi}^0 + t_{pi}^1 + \dots]$$

$$L(p) = \sum_i l_i [t_{pi}]$$

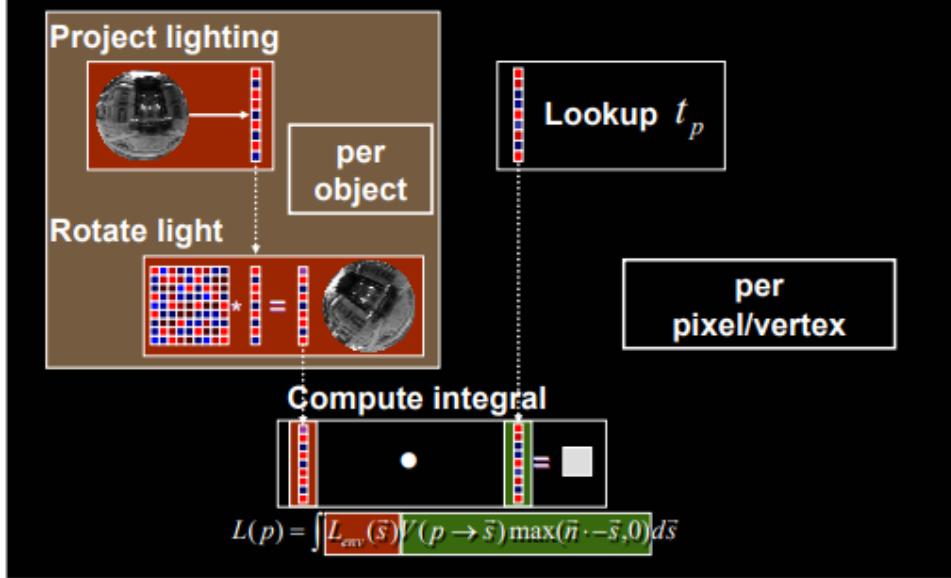
We do this for every bounce and fold everything into a final transfer vector

Celý to probíhá při renderování pak takhle:

Diffuse PRT



SIGGRAPH2005



This shows the rendering process.

We project the lighting into the basis (integral against basis functions). If the object is rotated wrt. to the lighting, we need to apply the inverse rotation to the lighting vector (in case of SH, use rotation matrix).

At run-time, we need to lookup the transfer vector at every pixel (or vertex, depending on implementation). A (vertex/pixel)-shader then computes the dot-product between the coefficient vectors. The result of this computation is the outgoing radiance at that point.

7. Architektura, shadery a práce s GPU

7.1. Architektura grafického akcelerátoru

Grafický akcelerátor (synonymum pro GPU), je kus hardware dedikovanéj pro masivně paralelizovaný procesování vizuálních dat.

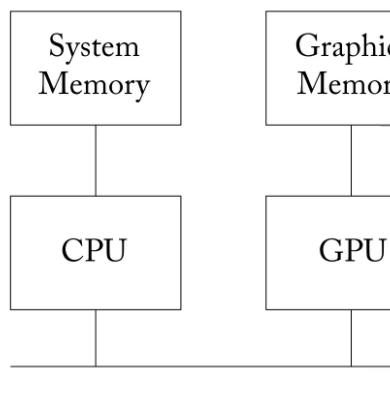
GPU architecture



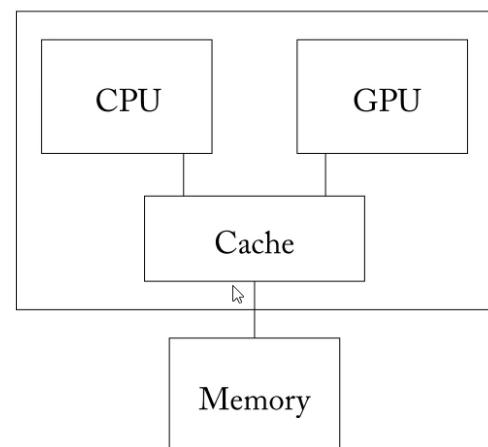
Narozdíl od procesoru (CPU) má GPU výrazně víc jader, s obrovským množstvím ALU (aritmeticko-logickejch jednotek), který se používají na počítání výpočtů. Znamená to, že je fakt dobrej v tom, řešit matiku která se děje na pozadí při vykreslování, ale je hrozně špatnej ve věcech okolo změny stavů, jako jsou třeba ify, změna textur, změna shaderů ,branching kódu, a podobný rozhodování.

GPU je tedy extrémně dobrý na paralelní algoritmy, při kterých se řeší spousty nezávislejch problémů který se nakonec zkombinujou, a moc nedává sekvenční algoritmy, který se pouští za sebou a po jednom.

Z high-level pohledu jsou nejčastější dvě různé spojení CPU/GPU architektury:



(a) System with discrete GPU



(b) Integrated CPU and GPU

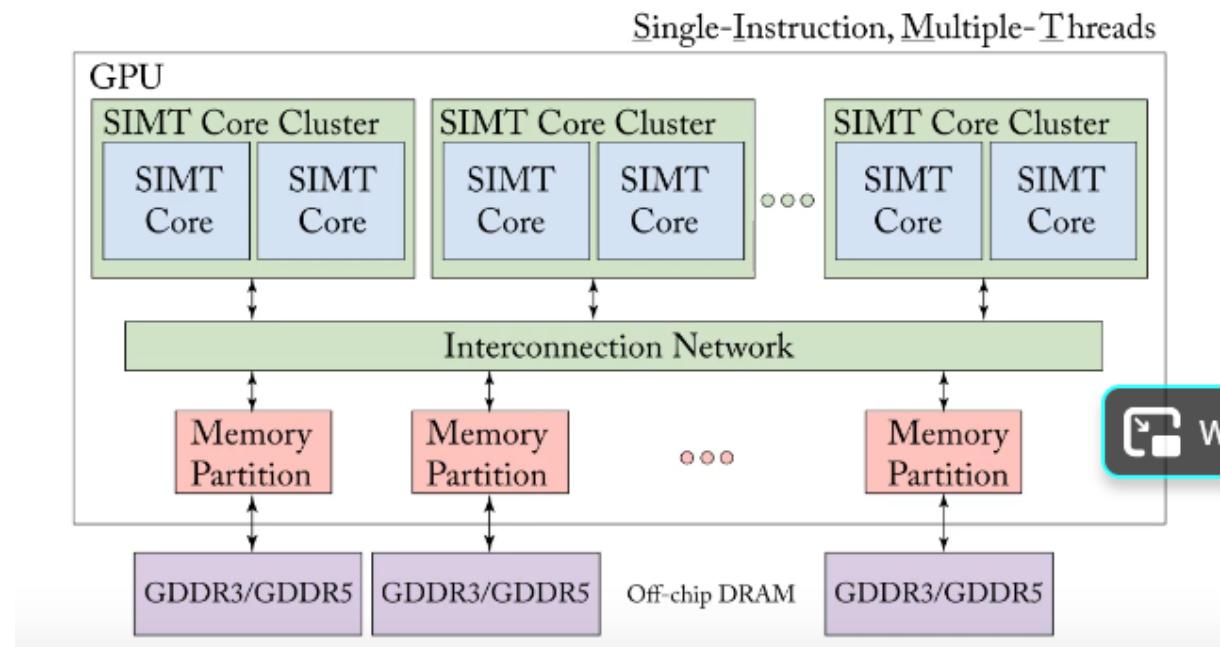
Paměti nebývají stejné - CPU paměť potřebuje nízkou latenci, kdežto GPU naopak vysokou propustnost.

Architektura - popisuje způsob, kterým je všechno vymyšlený a implementovaný, jak na fyzický, tak na programátorský hladině. Popisuje procesy, instruction set, hardware a tak.

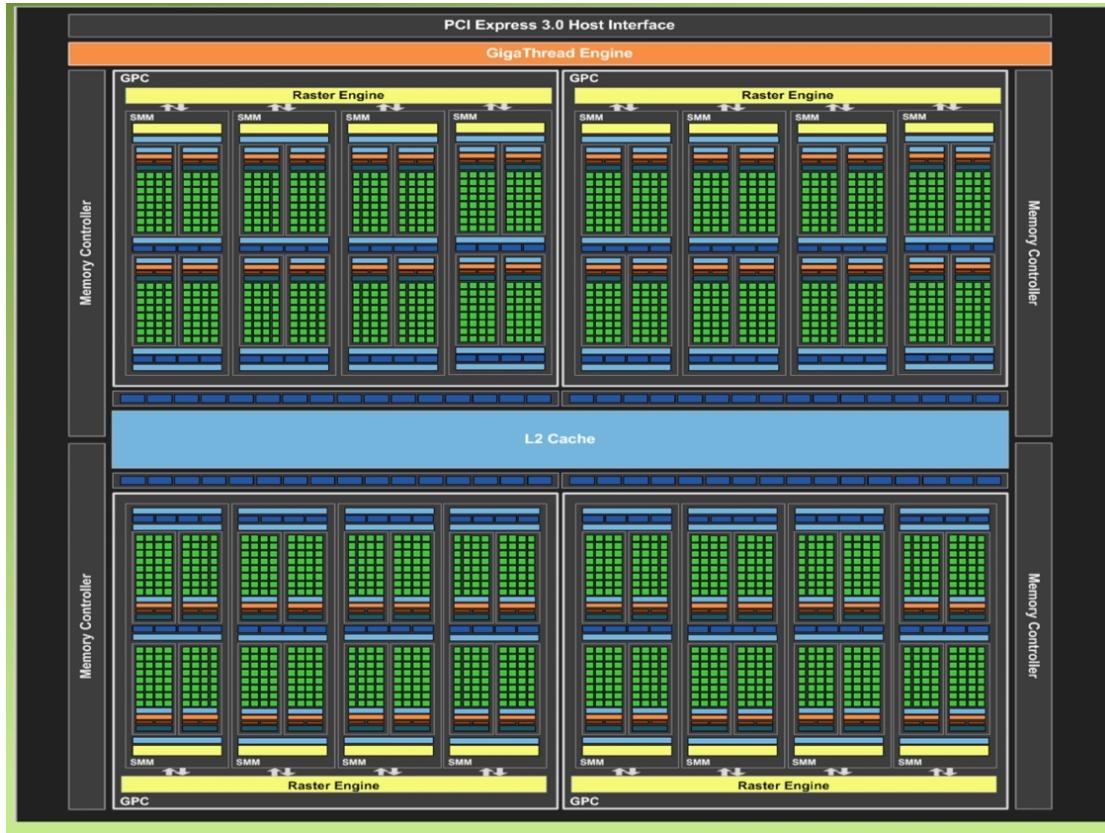
Podsoučástí architektury je **Mikroarchitektura**, která popisuje low-level pohled na to, jak jsou jednotlivé instrukce prováděny v procesorech, jak řeší branch prediction, pipelining, datapathy a podobně.

Základní idea stavby GPU je zhruba takhle:

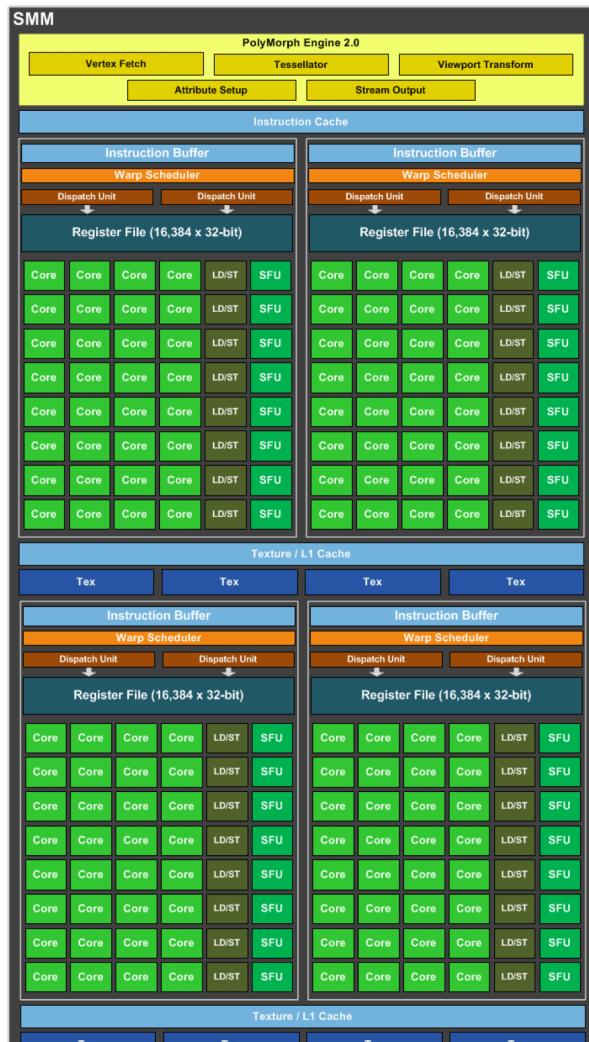
GPU jsou poskládaný z několika jader (cores), kterým NVIDIA říká Streaming Multiprocessors, a AMD Compute Units. Základní idea je SIMT-execution - Single Instruction, Multiple Thread, kde jednotlivý jádro (ze desítkama ALU) spouští jednu instrukci, na několika tisících vláken per jádro.



Nejznámější a nejčastěji popisované jsou mikroarchitektury od NVIDIA, který se v rámci let hodně mění. Mezi jednotlivýma architekturama dochází k zásadním změnám, tj. nejde o drobné zlepšeníčko, ale občas i dost velký overhaul. V přednáškách se hodně mluví o Maxwellu, což je mikroarchitektura generace kolem GeForce 900, tj. cca rok 2014. Maxwell vypadá zhruba takhle: (GTX 980)



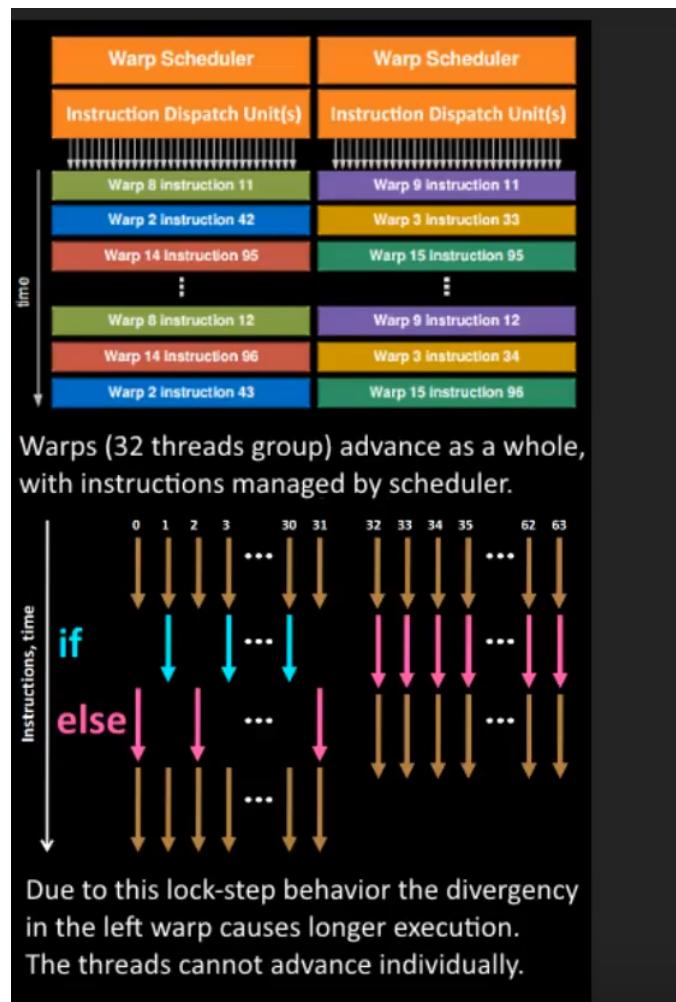
Podle modelu obsahuje různěj počet GPCs, tady jsou 4. GPC je graphics processing cluster, a každej z clusterů v sobě má v tomhle případě 4 Streaming multiprocessor. V těch se děje ta hlavní magie:



Je rozdělenej do 4 stejnejch bloků, opět, a každej má warp scheduler, 2 dispatch jednotky, 32 CUDA jader (int float operace), 8 special function jader (cos, sin atd), 8 Load/Store jader, registry a nějaký cache kolem.

SIMT funguje tak, že groupuje vlákna (vertices, pixely nebo primitivy) do Warps, který spouští zároveň stejnou instrukci. Tím, že se spouští tolik instrukcí najednou, zůstává víc místa pro ALUs místo kontrolních jednotek (warp schedulerů). Nastává ale problém v případě, že je potřeba branching kódu. (Viz další obrázek)

Problémem bývá taky sdílená paměť - pokud je potřeba změna stavu, většinou se musí zahodit pro všechny ty jádra a začít znova.

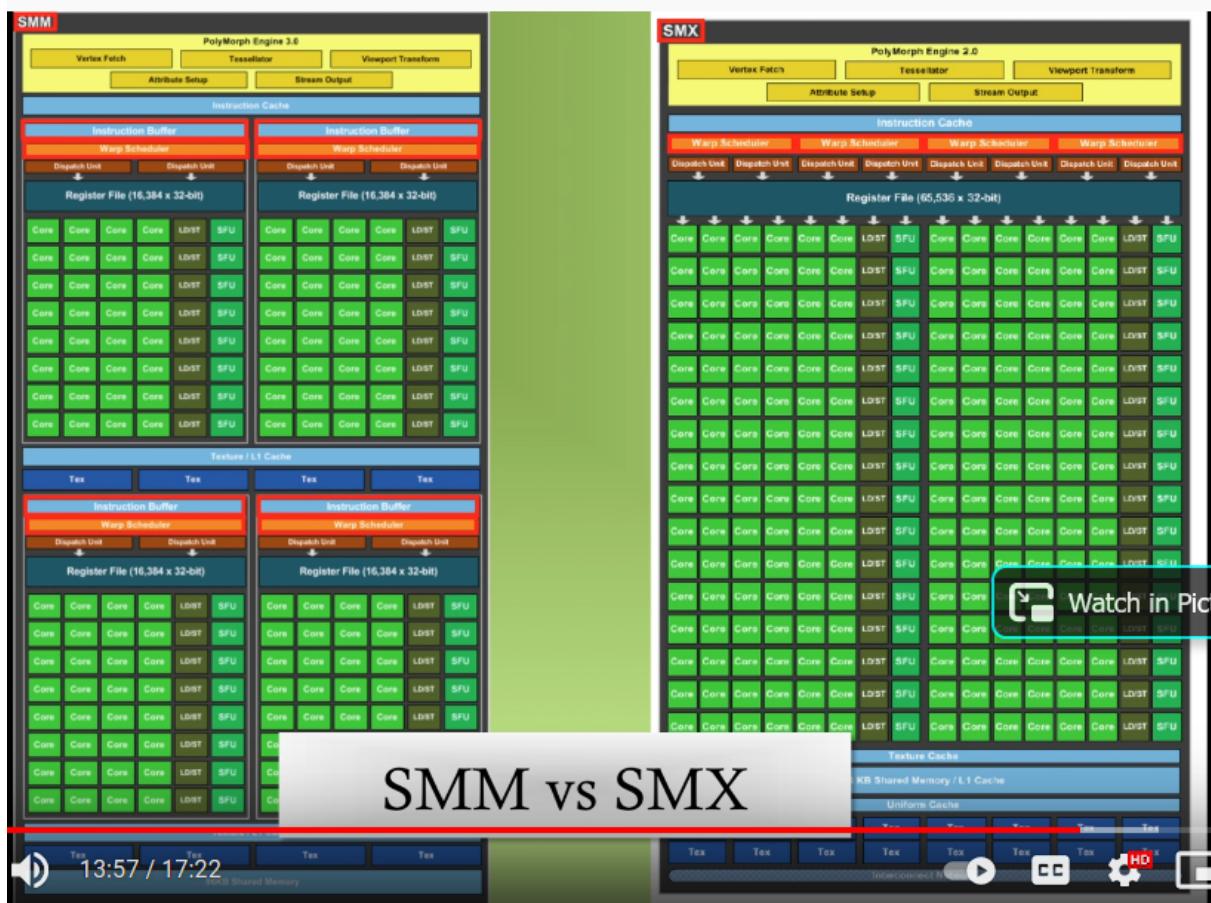


Tady je vidět, jakým způsobem jsou warpy (grupa několika vláken, co má provádět úplně stejnou instrukci, ale na různá data. Tj třeba shader kód na různý pixely) dispatchovány. Aby se zamaskovala latence paměti, tak warpy co čekají na data prostě uvolní místo dalším warpům.

Druhá půlka ukazuje, co se děje v případě, když dojde ke branchingu (tj shader má if, takže najednou chce půlka jinou instrukci). Máme dvě možnosti - buď warp execute jeden branch a doufá, že nedojde k rozdelení a je to správně. Pokud ne, musí zahazovat. Další možnost je, že executne oboje možnosti, a zamaskuje ty výsledky, který ho nezajímají. Obojoe má svoje nevyhody, proto je architektura tan náchylná na branching.

Samotný počty různých jader, bufferů, dispatcherů a tak se mění mezi různými generacemi architektur, ta základní idea warpů a masivní paralelizace je ale furt stejná. Například třeba mezi Keplerem (jedny z prvních NVIDIA generací) a Maxwellem (ta další, 2014) je zásadní

změna to, že dali každému warp scheduleru jeho vlastní instruction buffer, což předtím měli sharovanej:



A například pro porovnání, takhle vypadá Streaming Multiprocessor z NVIDIA Ampere, což je generace z minulého roku:



Navíc je tam vidět například Tensor jádra, který se používají k počítání maticových operací/násobení jednou instrukcí (aspoň, to sem pochopil). Taky mega zvětlí počet jader.

Samozřejmě, tohle je jenom jeden Streaming Multiprocessor. Samotná Aphere grafárna vypadá takhle:



7.2. Předávání dat do GPU

Předávání dat do GPU je řešeno primárně skrz API programovatelný pipeliny. Nejčastěji používaný API jsou OpenGL a DirectX. Většina přednášek funguje na OpenGL, takže se držím toho.

Předávání dat do GPU funguje v naprostý většině případů přes buffery (viz další podkapitoly), konkrétně přes Vertex Buffer, Index Buffer a Texture Mapping Units. (Jsou i další způsoby, třeba jak posílat attributy do shaderů před uniform variably, ale ty se moc často nedělají).

Vertex Buffer obsahuje pole všech vertexů co chci zobrazovat. U každého vertexu můžu mít kolik dat chci, většinou ale určitě budu mít aspoň polohu.

```
GLfloat vertices[] = {
    -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, // Top-left - formát X, Y, R, G, B
    0.5f, 0.5f, 0.0f, 1.0f, 0.0f, // Top-right
    0.5f, -0.5f, 0.0f, 0.0f, 1.0f, // Bottom-right
    -0.5f, -0.5f, 1.0f, 1.0f, 1.0f // Bottom-left
};
```

```
glGenBuffers(1, &VBO);
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
 Potom určím který data znamenají co přes VertexAttribPointer:
```

The function `glVertexAttribPointer` specifies how OpenGL should interpret the vertex buffer data whenever a drawing call is made. The interpretation specified is stored in the currently bound vertex array object saving us all quite some work.

The parameters of `glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer)` are as follows:

- **index:** Specifies the index of the vertex attribute.
- **size:** Specifies the number of components per vertex attribute. Must be 1, 2, 3, 4.
- **type:** Specifies the data type of each component in the array.
- **normalized:** Specifies whether data should be normalized (clamped to the range -1 to 1 for signed values and 0 to 1 for unsigned values).
- **stride:** Specifies the byte offset between consecutive vertex attributes. If stride is 0, the generic vertex attributes are understood to be tightly packed in the array.
- **pointer:** Specifies an offset of the first component of the first vertex attribute in the array.

Example usage

	VERTEX 1	VERTEX 2	VERTEX 3
	X Y Z	X Y Z	X Y Z
<code>GLfloat data[] = {</code>			
// Position // Color Z // TexCoords			
1.0f, 0.0f, 0.5f, 0.5f, 0.0f, 0.5f,	16	20	24
0.0f, 1.0f, 0.2f, 0.8f, 0.0f, 0.0f, 1.0f			28
<code>};</code> STRIDE: 12			32
<code>glEnableVertexAttribArray(0);</code>			
<code>glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,</code>			
7 * sizeof(GLfloat), (GLvoid*)0);			
<code>glEnableVertexAttribArray(1);</code>			
<code>glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,</code>			
7 * sizeof(GLfloat), (GLvoid*)(2 * sizeof(GLfloat)));			
<code>glEnableVertexAttribArray(2);</code>			
<code>glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,</code>			
7 * sizeof(GLfloat), (GLvoid*)(5 * sizeof(GLfloat)));			

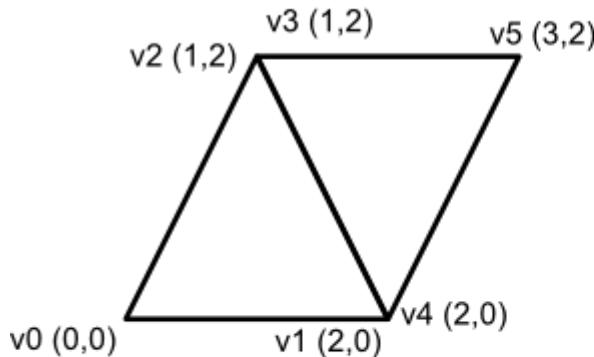
Teoreticky můžu vertexbuffer prostě vykreslit tak, že řeknu `glDrawArray`, a on je vykreslí tak, že vezme trojice vertexu a udělá z nich trojúhelník. Tím toho ale moc neudělám, protože abych udělal tříčetku čtverec, musel bych opakovat společný vertexy trojúhelníku 2x. A od toho je **Index buffer**:

```
unsigned int indices[] = { // note that we start from 0!
    0, 1, 3, // first triangle
    1, 2, 3 // second triangle
};

unsigned int EBO;
 glGenBuffers(1, &EBO);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
 GL_STATIC_DRAW);
```

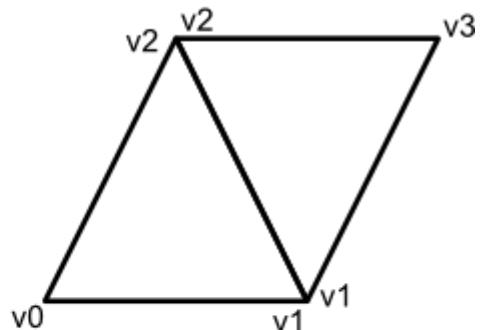
Tohle mi říká, že má vykreslit 2 trojuhelníky - 0,1,3 a 1,2,3, kde ty čísla jsou indexy ve Vertex bufferu. Nemusím potom duplikovat vertexy:

Without indexing



[0,0, 2,0, 1,2, 1,2, 2,0, 3,2]

With indexing



[0,1,2, 2,1,3]
[0,0, 2,0, 1,2, 3,2]

Vertices
reused
twice

Tím se do GPU dostávají data o scéně, a objekty co má zobrazit. Další důležitou částí jsou Textury, který se předávají přes TMU:

```
unsigned int texture;
glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);
// set the texture wrapping/filtering options (on the currently bound texture object)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// load and generate the texture
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
if (data)
{
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
    std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(data);
```

Kód shaderů a jak se předává je popsáný dál v kapitole o shaderech. Důležitý ale je, že se do něj dají předávat parametry přes Uniform variables (ty jsou globální pro celou scénu stejný) a vertex attributy (ty má každej vertex jiný):

Tohle nastaví *float4 uniform ourColor* variable v aktivních shaderech na *greenValue*:

```
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
```

```
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

A tohle použije data z VertexArraye, aby je bindlo na parameter 0 a 1:

```
// position attribute  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);  
glEnableVertexAttribArray(0);  
// color attribute  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));  
glEnableVertexAttribArray(1);
```

Který v shaderu použiju takhle:

```
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0  
layout (location = 1) in vec3 aColor;
```

7.3. Textury a GPU buffery

OpenGL pracuje v podstatě převážně jenom s bufferama, a většina workflow je přes to, že do nich jsou posílaný data. Co je ale ještě nadřazený, je Vertex Array Object, **VAO**. Ten si musím vytvořit na začátku, předtím než si začnu vytvářet buffery. Je to pole, který si ukládá informace o jiných bufferech a jejich atributech co jsem si potom vytvořil, abych mohl když tak rychle swapovat mezi kontextama, nebo to nemusel definovat pokaždý znova.

Hrozně hezký popsáný je to tady:

<https://stackoverflow.com/a/57258564>

Co se týče bufferů, tak většinou je workflow následující:

- Vygeneruju si buffer - glGenBuffers(n,&p) - vytvoří n bufferů a pointer na ně dá do p
- Bindnu si buffer, čímž ho nastavím v kontextu jako aktivní a další další volání s ním počítá. Zároveň mu určím i jeho typ. - glBindBuffer(type, pointer)
 - Typo je hodně:

Specifies the target to which the buffer object is bound. The symbolic constant must be GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_TEXTURE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, or GL_UNIFORM_BUFFER.
- Nahraju do něj data - (GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage)
- Nastavím attributy, který popisujou jaký data v něm mám a co je kde:

```

GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
 glEnableVertexAttribArray(posAttrib);
 glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE,
                      5*sizeof(float), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
 glEnableVertexAttribArray(colAttrib);
 glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
                      5*sizeof(float), (void*)(2*sizeof(float)));

```

Například pokud chci vytvořit Vertex Buffer, tak na to musím takhle:

```

// This will identify our vertex buffer
GLuint vertexbuffer;
// Generate 1 buffer, put the resulting identifier in vertexbuffer
 glGenBuffers(1, &vertexbuffer);
// The following commands will talk about our 'vertexbuffer' buffer
 glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
// Give our vertices to OpenGL.
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);

```

Specifický druhý bufferů jsou následující:

GL_ARRAY_BUFFER

The buffer will be used as a source for vertex data, but the connection is only made when glVertexAttribPointer is called. The pointer field of this function is taken as a byte offset from the beginning of whatever buffer is currently bound to this target.

GL_ELEMENT_ARRAY_BUFFER

All rendering functions of the form gl*Draw*Elements* will use the pointer field as a byte offset from the beginning of the buffer object bound to this target. The indices used for indexed rendering will be taken from the buffer object. Note that this binding target is part of a Vertex Array Objects state, so a VAO must be bound before binding a buffer here.

GL_COPY_READ_BUFFER and GL_COPY_WRITE_BUFFER

These have no particular semantics. Because they have no actual meaning, they are useful targets for copying buffer object data with glCopyBufferSubData. You do not have to use these targets when copying, but by using them, you avoid disturbing buffer targets that have actual semantics.

GL_PIXEL_UNPACK_BUFFER and GL_PIXEL_PACK_BUFFER

These are for performing asynchronous pixel transfer operations. If a buffer is bound to GL_PIXEL_UNPACK_BUFFER, glTexImage*, glTexSubImage*, glCompressedTexImage*, and glCompressedTexSubImage* are all affected. These functions will read their data from the bound buffer object instead of where a client pointer points. Similarly, if a buffer is bound to GL_PIXEL_PACK_BUFFER, glGetTexImage, and glReadPixels will store their data to the bound buffer object instead of where a client pointer points.

GL_QUERY_BUFFER

These are for performing direct writes from asynchronous queries to buffer object memory. If a buffer is bound to GL_QUERY_BUFFER, then all glGetQueryObject[ui64v] function calls will write the result to an offset into the bound buffer object.

GL_TEXTURE_BUFFER

This target has no special semantics.

GL_TRANSFORM_FEEDBACK_BUFFER

An indexed buffer binding for buffers used in Transform Feedback operations.

GL_UNIFORM_BUFFER

An indexed buffer binding for buffers used as storage for uniform blocks.

GL_DRAW_INDIRECT_BUFFER

The buffer bound to this target will be used as the source for the indirect data when performing indirect rendering. This requires OpenGL 4.0 or ARB_draw_indirect.

GL_ATOMIC_COUNTER_BUFFER

An indexed buffer binding for buffers used as storage for atomic counters. This requires OpenGL 4.2 or ARB_shader_atomic_counters

GL_DISPATCH_INDIRECT_BUFFER

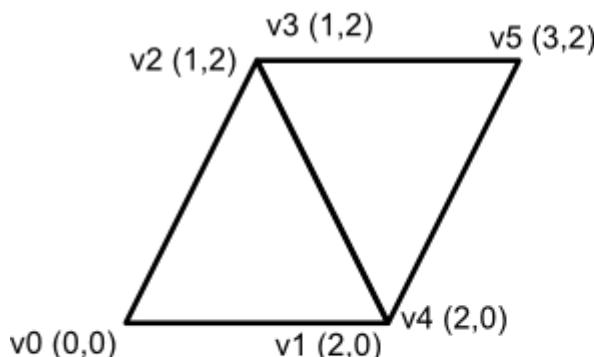
The buffer bound to this target will be used as the source for indirect compute dispatch operations, via glDispatchComputeIndirect. This requires OpenGL 4.3 or ARB_compute_shader.

GL_SHADER_STORAGE_BUFFER

An indexed buffer binding for buffers used as storage for shader storage blocks. This requires OpenGL 4.3 or ARB_shader_storage_buffer_object.

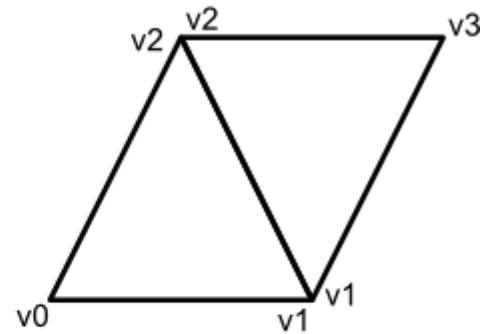
Nejdůležitější z nich jsou ARRAY a ELEMENT_ARRAY buffer, protože se používají jako vertex (**VBO**) a index (**IBO**) buffer. Tj, v ARRAY bufferu jsou jednotlivý data o vertexech, jejich barva a vzhled, a v ELEMENT bufferu je zase uložené index buffer, tj odkaz na indexy ve vertex bufferu který určuje jak je má pospojovat do stěn:

Without indexing



[0,0, 2,0, 1,2, 1,2, 2,0, 3,2]

With indexing



[0,1,2, 2,1,3]
[0,0, 2,0, 1,2, 3,2]

Vertices
reused
twice

(Obrázek ukazuje, jak se vykresluje VBO bez ELEMENT bufferu, a vpravo jak se vykresluje s element bufferem. Zeleně označený je právě ELEMENT buffer, který ukazuje co za indexy spojit. Bez něj je potřeba duplikovat vertex data, protože prostě spojuje po trojicích)

Mimo výše uvedený buffer objekty má OpenGL taky přímo předdefinovaný **Framebuffery**, který se používají k tomu, že do nich jde přímo kreslit. Dá se definovat si vlastní, se kterejma se potom pracuje za použití attachmentů.

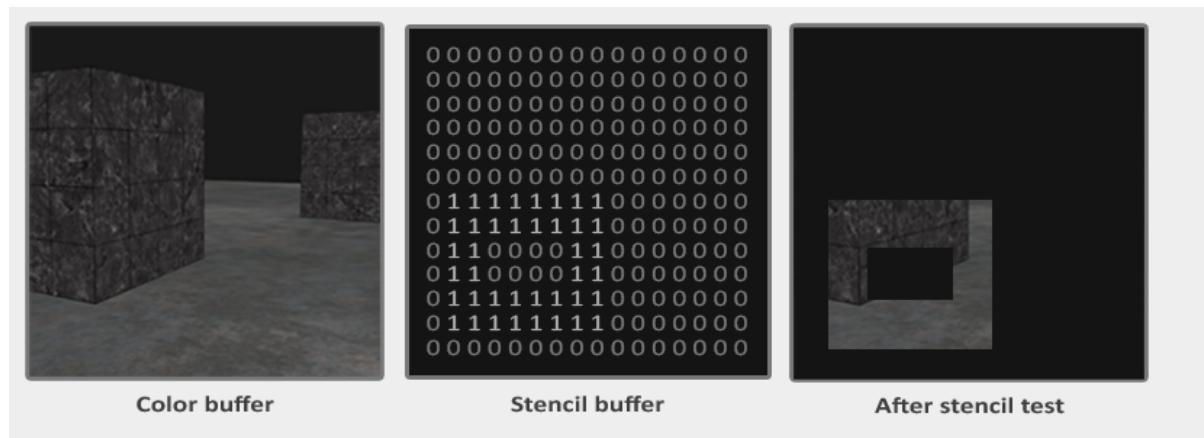
Od základu jsou ale definovaný následující buffery:

GL_FRONT_LEFT (vykreslený), GL_BACK_LEFT (záložní na switch), GL_FRONT_RIGHT (VR), and GL_BACK_RIGHT (VR) - **color buffery**. Je to screen buffer, tj co je vidět na obrazovce, barvy jednotlivých pixelů.. FRONT a BACK slouží pro double buffering - FRONT je vidět na obrazovce, do BACK renderuju.

Používají se jenom levý buffery - tj GL_FRONT_LEFT a GL_BACK_LEFT. Pravý hodí error, pokud karta nepodporuje 3D stereo zobrazování.

Další je **GL_DEPTH** buffer (Z-buffer), kterej se používá pro výpočet viditelnosti při renderování.

Poslední je **GL_STENCIL**, což je stencil buffer. Používá se pro stencil testy, kterýma se dá dělat jak culling tak třeba počítat stíny. Je to v podstatě custom depth buffer, protože se používá pro podobnou věc, akorát si můžeš nastavit jak se přesně chová. V podstatě jde o to, že definuju Stencil Test, ve kterém určím hodnoty stencil bufferu pro daný vertex, a ten se vykreslí podle toho jestli prošel nebo neprošel. Např. takhle:



Nastavuje se přes `glStencilFunc`:

The `glStencilFunc(GLenum func, GLint ref, GLuint mask)` has three parameters:

- `func`: sets the stencil test function that determines whether a fragment passes or is discarded. This test function is applied to the stored stencil value and the `glStencilFunc`'s `ref` value. Possible options are: `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL` and `GL_ALWAYS`. The semantic meaning of these is similar to the depth buffer's functions.
- `ref`: specifies the reference value for the stencil test. The stencil buffer's content is compared to this value.
- `mask`: specifies a mask that is ANDed with both the reference value and the stored stencil value before the test compares them. Initially set to all 1s.

A `glStencilOp`:

But `glStencilFunc` only describes whether OpenGL should pass or discard fragments based on the stencil buffer's content, not how we can actually update the buffer. That is where `glStencilOp` comes in.

The `glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass)` contains three options of which we can specify for each option what action to take:

- `sfail`: action to take if the stencil test fails.
- `dpfail`: action to take if the stencil test passes, but the depth test fails.
- `dppass`: action to take if both the stencil and the depth test pass.

Then for each of the options you can take any of the following actions:

Action	Description
<code>GL_KEEP</code>	The currently stored stencil value is kept.
<code>GL_ZERO</code>	The stencil value is set to 0.
<code>GL_REPLACE</code>	The stencil value is replaced with the reference value set with <code>glStencilFunc</code> .
<code>GL_INCR</code>	The stencil value is increased by 1 if it is lower than the maximum value.
<code>GL_INCR_WRAP</code>	Same as <code>GL_INCR</code> , but wraps it back to 0 as soon as the maximum value is exceeded.
<code>GL_DECR</code>	The stencil value is decreased by 1 if it is higher than the minimum value.
<code>GL_DECR_WRAP</code>	Same as <code>GL_DECR</code> , but wraps it to the maximum value if it ends up lower than 0.
<code>GL_INVERT</code>	Bitwise inverts the current stencil buffer value.

Dá se použít třeba pro kreslení outline u jednotek docela easy způsobem:

1. Enable stencil writing.
2. Set the stencil op to `GL_ALWAYS` before drawing the (to be outlined) objects, updating the stencil buffer with 1s wherever the objects' fragments are rendered.
3. Render the objects.
4. Disable stencil writing and depth testing.
5. Scale each of the objects by a small amount.
6. Use a different fragment shader that outputs a single (border) color.
7. Draw the objects again, but only if their fragments' stencil values are not equal to 1.
8. Enable depth testing again and restore stencil func to `GL_KEEP`.

Framebuffers - výše zmíněný color, depth a stencil buffery jsou defaultní, který si založila knihovna při tvorbě výchozího openGL kontextu. Můžu si ale definovat flastní **FBO** - framebuffer objekty, který slouží jako render targets a se kterejma se dají dělat spousty dalších věcí - třeba odrazy zrcadel, stíny a podobně.

Definují se stejně jako je zvyk u OenGL:

```
GLuint frameBuffer;  
glGenFramebuffers(1, &frameBuffer);  
 glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer); - tohle ho už ale bindne na zápis,  
 takže když pak volám glDraw volání, tak to kreslí do aktivních bufferu.  
(glBindFramebuffer(GL_FRAMEBUFFER, 0) to zase vrátí na GL_BACK_LEFT)
```

Aby šel použít, musí splnit následující podmínky:

- At least one buffer has been attached (e.g. color, depth, stencil)
- There must be at least one color attachment (OpenGL 4.1 and earlier)
- All attachments are complete (For example, a texture attachment needs to have memory reserved)
- All attachments must have the same number of multisamples

Nejčastěji se jako attachmenty přidávají textury, nebo RenderBuffery. Tenhle kód vytvoří nové renderbuffer, a připojí ho na aktivní framebuffer jako DEPTH a STENCIL buffer:

```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600);
glBindRenderbuffer(GL_RENDERBUFFER, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
GL_RENDERBUFFER, rbo); - tohle je připojení na framebuffer jako attachment do Depth a Stencilu.
```

Renderbuffer je buffer, který je optimalizovaný na zápis a nedá se z něj přímo číst. Jediný jak z něj jde číst je pomocí pomalého glReadPixels, ale narození od textury má výhody v tom, že se data ukládají přímo a nesnaží se o kompresi nebo zpracování jako u textur. A protože ví, že je read-only, tak může být optimalizovanější. Používá se proto hodně třeba na depth a stencil buffer, protože z něj číst nepotřebujeme my, ale až rendering (který na něj už šáhat může). Stejně tak se dá prostě přepnout jako aktivní v obrazovce a rovnou zobrazit.

Další jsou **Textury**, a pracuje se s nima v podstatě stejně jako s předchozíma bufferama:

TEXTURING API

- Texture handle creation:

```
unsigned int texture;
glGenTextures(1, &texture);
```

- Texturing unit activation and texture binding:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
```

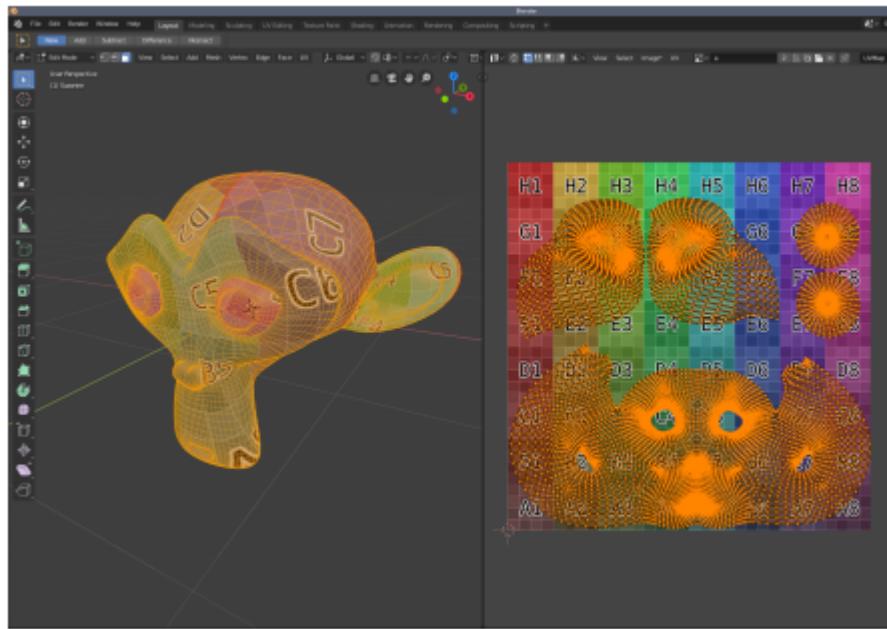
- Data upload:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE
, data);
```

- Texturing parameters:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
...;
```

Důležitou myšlenkou jsou UV souřadnice a UV mapping, které určují pro jednotlivé vertexy jejich polohu na 2D textuře. Na základě toho se potom texturuje v pixel shaderu.

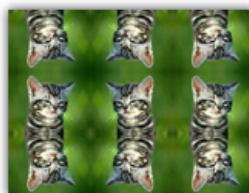


Doména textury je definovaná pro UV souřadnice v $[0,1]$ rozsahu. V případě, že se pokusím načíst texturu co je mimo rozsah, záleží na tom jak mám buffer nastavený, a to se stane. To se nastavuje skrz parameter `GL_TEXTURE_WRAP`:

- `GL_REPEAT`: The integer part of the coordinate will be ignored and a repeating pattern is formed.
- `GL_MIRRORED_REPEAT`: The texture will also be repeated, but it will be mirrored when the integer part of the coordinate is odd.
- `GL_CLAMP_TO_EDGE`: The coordinate will simply be clamped between `0` and `1`.
- `GL_CLAMP_TO_BORDER`: The coordinates that fall outside the range will be given a specified border color.



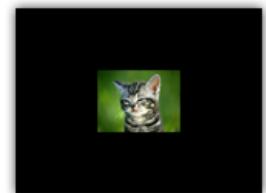
GL_REPEAT



GL_MIRRORED_REPEAT

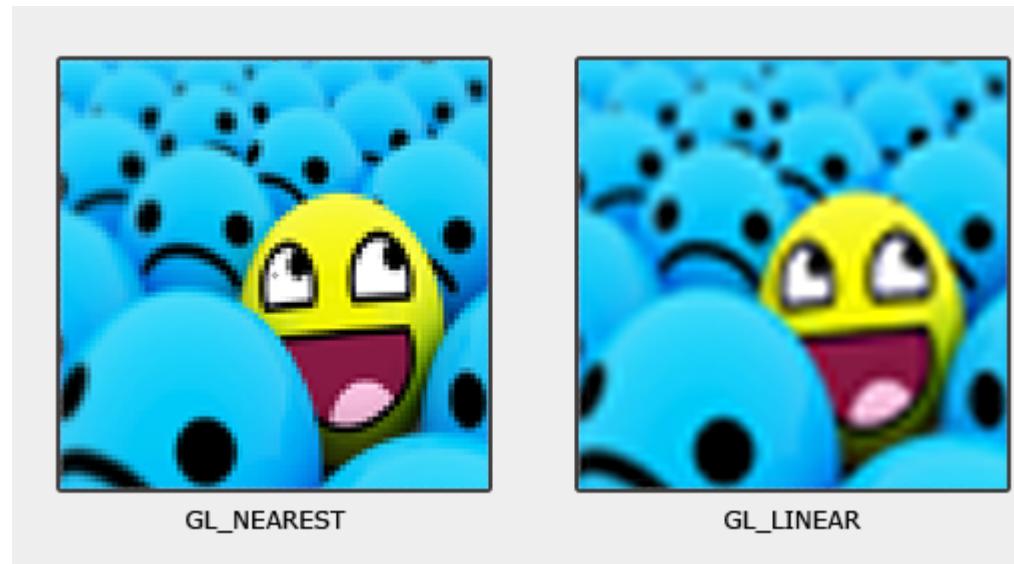


GL_CLAMP_TO_EDGE



GL_CLAMP_TO_BORDER

Taky se stará o sampling/filtering, a ten se nastavuje skrz GL_TEXTURE_MIN_FILTER a GL_TEXTURE_MAG_FILTER (podle toho jesli je blíž nebo dál):



O textry se sarají Texture Mapping Units, TMU, v grafické kartě - je to hardware. Pracuje se s nima přes Sampler2D v GLSL:

- ▶ `glBindSampler()` + `glBindTexture()` – bind to a texture unit

```
#version 330

in vec2 v_tex;
out vec4 f_color;

uniform sampler2D u_texture;

void main() {
    f_color = texture(u_texture, v_tex);
}
```

Pokud chci teda využít 2 texture units, tak to provedu takhle:

```
glActiveTexture(GL_TEXTURE0); aktivuju TMU 0
glBindTexture(GL_TEXTURE_2D, texture1); a nahraju do ní texturu 1
glActiveTexture(GL_TEXTURE1); Aktivuju TMU 1
glBindTexture(GL_TEXTURE_2D, texture2); nahraju do ní texturu 2
glUniform1i(glGetUniformLocation(ourShader.ID, "texture1"), 0);
glUniform1i(glGetUniformLocation(ourShader.ID, "texture2"), 1);
A nastavím uniform Sampler2D attribut.
```

7.4. Programování GPU - shadery

Shadery jsou miniprográmky psaný v jednom ze shading languages, jako je GLSL (opengl) nebo HLSL (directx). Ve vykreslovací pipelině zaujímají důležitý místa, a určují jak a kde se zobrazí který objekt a pixel.

Celkem máme následující shadery:

- **Vertex Shaders:** `GL_VERTEX_SHADER`
 - Dostane vertex, musí vyplivnout jeho polohu. Tj dělá transformace vertexu.
- **Tessellation Control and Evaluation Shaders:** `GL_TESS_CONTROL_SHADER` and `GL_TESS_EVALUATION_SHADER`. (requires GL 4.0 or `ARB_tessellation_shader`)
 - Tessellation umí dělit plochu na detailnější, tj přidávat vertexy
- **Geometry Shaders:** `GL_GEOMETRY_SHADER`
 - Dostane primitiva (tj trojuhelník), a musí vyplivnout jeden nebo více primitivů. Může zahazovat!?
- **Fragment Shaders:** `GL_FRAGMENT_SHADER = PIXEL_SHADER!`
 - Dostane pixel, a musí dát dohromady jeho barvu.
- **Compute Shaders:** `GL_COMPUTE_SHADER`. (requires GL 4.3 or `ARB_compute_shader`)
 - Může si počítat co chce. Je relativně mimo pipelinu, a používá se na počítání dat kolem.

Nejdůležitější je Vertex a Fragment (Pixel) shader, bez kterých se nejde obejít. Vypadají zhruba takhle:

Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0

out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color
}
```

Fragment shader

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)

void main()
{
    FragColor = vertexColor;
}
```

Nejzákladnější mechanika je jejich IN a OUT parametry. Vertex shader je v tom trochu speciální, protože IN parametry dostává z Vertex Bufferu, kde se voláním `glVertexAttribPointer` určí, kde má ty data brát (resp. jakou strukturu mají ve vertex bufferu).

Další možnosti jsou Uniform parametry, který se bindují přímo voláním a jsou pro každý vertex stejný.

Další věc co má shader definovanou jsou OUT parametry. U Fragment Shaderu to je Vec4, který určuje jakou barvu bude mít pixel na obrazovce. U vertex shaderu to ale může být úplně cokoliv, co si definuje. V příkladu vejš je to vec4 vertexColor, ale klidně k tomu může přidat vec4 normal, vec4 lightColor a podobně. Jediný co je důležitý je, aby se OUT parametry Vertex shaderu rovnali IN parametrům ve Fragment shaderu.

Přímo grafická pipeline potom prochází tak, že zavolá pro každý vertex vertex shader, tomu naplní z vertex bufferu a z kódu in parametry, a potom vezme hodnoty z OUT parametrů a linkne je do Fragment shaderu, který s nimi může dál počítat, a nakonec vyplivne výslednou barvu pixelu.

Úkolem vertex shaderu je teda spočítat, kde na obrazovce se ten vertex vyskytuje v souřadnicích clip spacu. To znamená, že by mělo dojít k nějaké transformaci, tj v naprostý většině případu dostane jako vstup MVP matici: model - view - projection, tj transformace ze souřadnic v prostoru modelu (tj, tenhle trojuhelník je horní stěna kostky), přes prostor světa (kostka stojí v zadním rohu scény) do prostoru kamery (kamera kouká na scénu z levého rohu v tým perspektivě), a vypadá to nějak takhle:

Vertex shader:

```
#version 330 core
layout (location = 0) in vec3 aPos;
...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // note that we read the multiplication from right to left
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Matice se generují třeba takto:

```
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
glm::mat4 view = glm::mat4(1.0f);
// note that we're translating the scene in the reverse direction of where we want to move
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

A pak se posílají takto:

```
int modelLoc = glGetUniformLocation(ourShader.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
```

Přímo používání shaderů není nijak extra složitý, jenom je potřeba je zkompilovat, spojit do programu, a ten použít. `&vertexShaderSource`, je string s kódem shaderu.

```
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

```

glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
unsigned int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);

```

A potom v rendering loopy stačí jenom program použít a vykreslovat:

```

glClear(GL_COLOR_BUFFER_BIT);
// draw our first triangle
glUseProgram(shaderProgram);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glfwSwapBuffers(window);

```

Fragment Shader má zase za úkol vymyslet jakou barvu bude mít bod na obrazovce, na základě dat o jeho pozici co dostal od Vertex Shaderu. Většinou k tomu používá textury, a modely osvětlení.

Takhle si nahraju 2 textury v OpenGL:

```

GLuint textures[2];
 glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, textures[0]);
 image = SOIL_load_image("sample.png", &width, &height, 0, SOIL_LOAD_RGB);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
              GL_UNSIGNED_BYTE, image);
 SOIL_free_image_data(image);
 glUniform1i(glGetUniformLocation(shaderProgram, "texKitten"), 0);

 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, textures[1]);
 image = SOIL_load_image("sample2.png", &width, &height, 0, SOIL_LOAD_RGB);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
              GL_UNSIGNED_BYTE, image);
 SOIL_free_image_data(image);
 glUniform1i(glGetUniformLocation(shaderProgram, "texPuppy"), 1);

 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

```

Který potom ve Fragment Shaderu sampluju a mixuju pomocí texture a Texcoord attribute, který mi přišel od Vertex shaderu. (UV souřadnice kde ten pixel je)

```
...
uniform sampler2D texKitten;
uniform sampler2D texPuppy;

void main()
{
    vec4 colKitten = texture(texKitten, Texcoord);
    vec4 colPuppy = texture(texPuppy, Texcoord);
    outColor = mix(colKitten, colPuppy, 0.5);
}
```

Zbytek už je dost magie. V teorii je to všechno jednoduchý, a nic víc to neumí. Veškerý "multipass" shader věci co jsou v Unity shaderech, dopočítávání světla a dalších věcí nejsou v OpenGL, protože to všechno implementuje Unita navíc. Samotný shadery prostě dostanou vertex/fragment, a vyplivnou pozici/barvu. Takže se do něj musí implementovat veškerý pozice světla (předávají se jako maticy do shaderu), osvětlující modely se počítají přímo v shaderu, stejně jako animace a transformace bodů - se řeší tím, že si změním transformační matici ve vertex shaderu.

Pokud chci multipass (tj, volat víc shaderů, třeba v prvním passu vykreslit světla, ve druhém průhledný objekty), tak se toho dosáhne tak, že si prostě vykreslím co chci s jedním programem, pak změním program na druhý, a vykreslím zbytek.

7.5. Základy OpenGL, GLSL, CUDA a OpenCL

Tl;dl - CUDA - massivně paralelní programování multithreaded aplikací na GPUčku, od Nvidie. Prostě posíláš do grafárny vlastní thready a bloky a on ti je megarychle spočítá. Můeš si alokovat paměť na GPUčku.

OpenCL - Opensource verze CUDY, je to C api pro počítání věcí na grafárně, steně jako CUDA. Funguje ale stejně blbě jako OpenGL, tj musíš si bindovat buffery a pošílat data syntaxem podobným OpenCLku. Což CUDA to má relativně rozumějí.

7.6. Pokročilé techniky práce s GPU

Mezi pokročilý techniky práce s GPU může být dost cokoliv. Od použití stencil bufferu pro Shadow volumes, přes používání frame-bufferů na off-screen rendering třeba pro deferred rendering (nejdřív vykreslím do bufferu geometrii, a až pak v dalším passu barvy a světla)

Deferred shading - popisuje přístup, kdy si rozdělím renderování na více passů. Nejdřív si vytváří do G-bufferu (render target textury) s úplně jednoduchým fragment shaderem, který mi do něj uloží pozici, normálu, albedo a všechny data co potřebuju pro vykreslení světla. Potom renderuju znova, a vezmu si z té textury ty data a podle nich spočítám světla.

Výhodu to mám v tom, že při prvním passu proběhne depth-test, takže vím že nebudu nikdy počítat světla zbytečně.

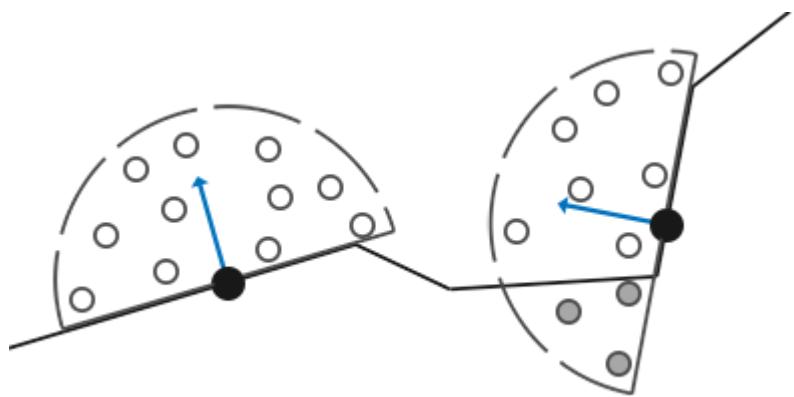
The short answer is, if you are using many dynamic lights then you should use deferred rendering. However, there are some significant drawbacks:

- This process requires a video card with multiple render targets. Old video cards don't have this, so it won't work on them. There is no workaround for this.
- It requires high bandwidth. You're sending big buffers around and old video cards, again, might not be able to handle this. There is no workaround for this, either.
- You can't use transparent objects. (Unless you combine deferred rendering with Forward Rendering for just those transparent objects; then you can work around this issue.)
- There's no anti-aliasing. Well, [some engines](#) would have you believe that, but there are solutions to this problem: [edge detection](#), [FXAA](#).
- Only one type of material is allowed, unless you use a modification of deferred rendering called [Deferred Lighting](#).
- Shadows are still dependent on the number of lights, and deferred rendering does not solve anything here.

Další možností je **SSAO** - Screen Space Ambient Occlusion.



Je to stínování podél hran. Dělá se tak, že si přidám SSAO rendering pass, a v něm na základě depth mapy ztmavím místa, který jsou moc v rohu.



Ztmavení počítám podle toho, že si samplnu pár pixelů kolem, a podle toho kolik procent jich je v sample polokouli, podle toho se ztmavím.

Nějaký další nápady jsou tady, v sekci Advanced Lightning a Advanced OpenGL:
<https://learnopengl.com/Advanced-OpenGL/Advanced-Data>

Popřípadě techniky jako:

- [Bump mapping](#)
- [Parallax mapping](#)
- [Advanced parallax mapping](#)
- [Sobel outline](#)
- [Toon shading](#)
-

7.7. Architektura herní engine.

Framework vs Engine - Framework je jenom knihovna co zjednodušuje jednu část vývoje, většinou nemá vlastní execute loop ani IDE. Třeba XNA nebo Havoc je ideální příklad

V podstatě se to dá vyimprovizovat z

<http://hightalestudios.com/wordpress/wp-content/uploads/2017/03/jaKUP.png>

A ze základní idey Game Loopu:

```

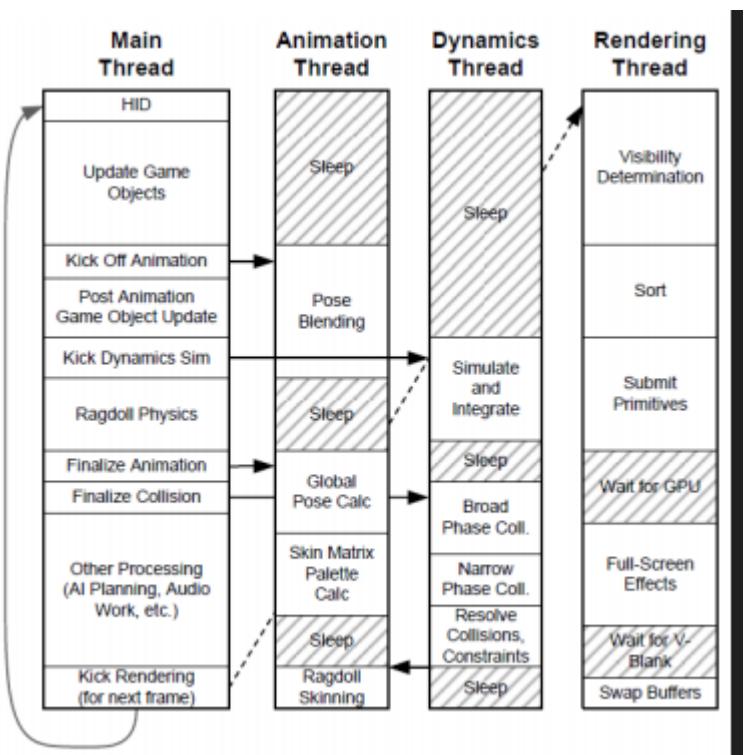
void main() // ENGINE
{
    initEngineAndLoadData();
    while (true) // game loop
    {
        inputSystem.Update();
        if (isExit()) break; // exit the game
        physics.tick();
        gameplay.tick();
        rendering.tick();
    }
    destroyEngine();
}

```

▀ (ještě se většinou do .tick přidává deltaTime, což je čas od posledního framu, abych měl podle čeho škálovat jak posouvám a updatuju věci. Taky může být kromě výše ukázanýho několik typů game loopů:

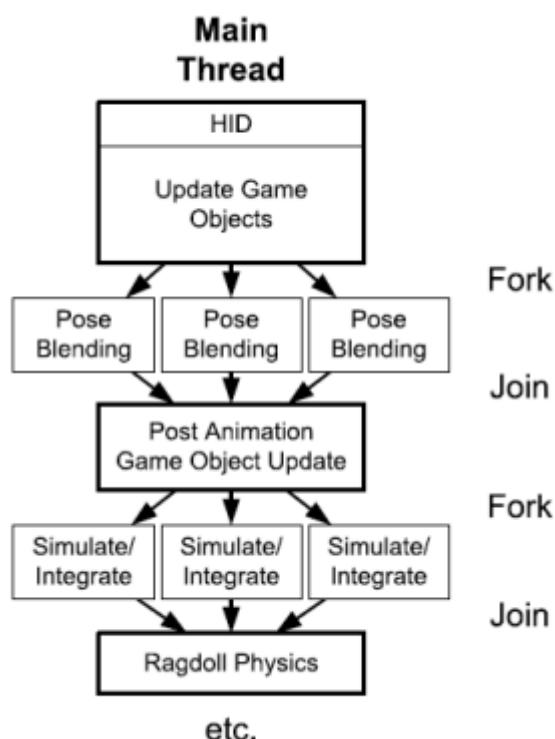
- Callback-driven zavolám frameBegin() event všem co poslouchají, vyrenderuju frame, zavolám frameEnd().
- Event-driven - mám periodický event (třeba timer ve winforms) co volám často, a v kterém probíhá gameloop. Další možností je volat event na konci loopu co ho spustí znova.

Pak tu sou i multi-threaded game loopy:



Per-system multithreading, není úplně ideální, ale dobrý pokus. Občas na sebe musí čekat.

Fork-join loop:

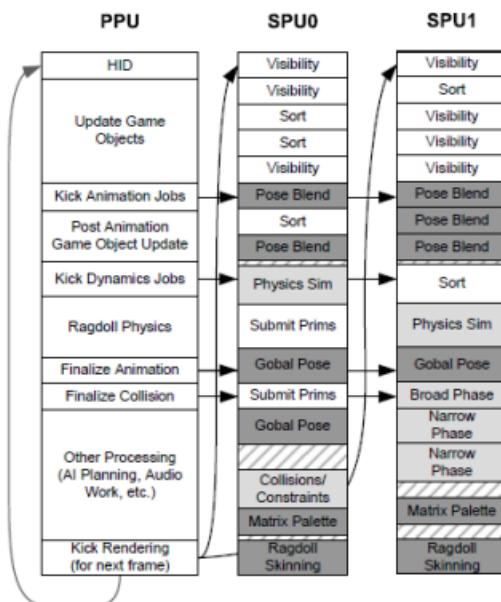


Nejstarší pokusy, nic moc, ale dobrý způsob jak naroubat multithread na původně single thread engine.

Lepším vylepšením je Job Model:

■ Job model

- Advocated in PS3 with SPURS library
- Jobs are fine-grained and can be distributed between as many cores as one have
- Whenever main thread has nothing to do, it crunches jobs as well!
- E.g. Unreal Engine 4



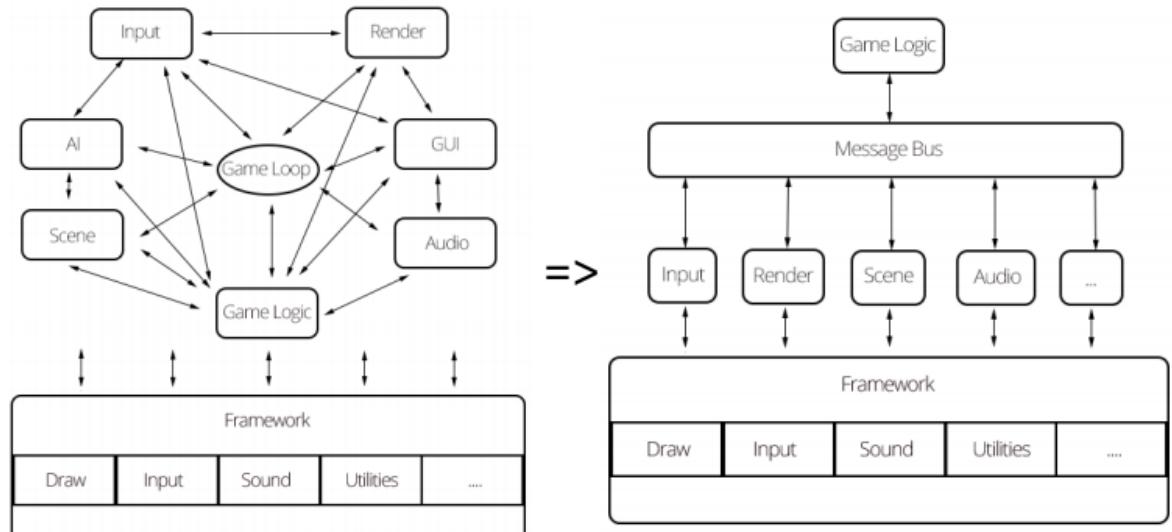
Má několik problémů, ale idea je, že mám job scheduler systém, do kterého házím jednotlivý ohraničený tasky, a chopí se jich jakýkoliv jádro co má zrovna volno. Je tam docela problém se sdílením dat, false sharing (když si navzájem invalidujou cache - <https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html>) a tak. (**Když oba modifikovali data najednou, způsobí cache invalidation a je třeba znova flushnout a reloadnout**)

Taky důležitou součástí enginu je práce s resourcama a assetama, jak je ukládá, jak je referencuje a jak s nima dovoluje pracovat tak, aby to nebylo moc náročný. Nejlepší je ukládat skrz nějaký GUIDčka, který jsou ale číselný a né textový, protože s tím se rychleji pracuje.

Velký problémy bejvaj s memory-managementem, protože tam se ztrácí hrozně moc času. Alokace paměti je drahá, a často zvlášť ve hrách je jednoduchý spadnout k tomu, že člověk alokuje novou paměť každej frame, a pak Garbage Collector hrozně zaseká hru v náhodných intervalech.

Třeba hezký řešení je mít **“per-frame” blok paměti**, která obsahuje data z minulého framu, a na konci se smaže. Nealokuje se nic novýho, blok je to furt stejnej, jenom vím že to je místo co můžu použít proto, abych si poslal info do dalšího framu. Clean po N framech.

Dalším zajímavým architektonickým řešením je třeba jak řešit posílaní zpráv sem tam. Určitě lepší přes message systém:



http://www.gamasutra.com/blogs/MichaelKissner/20151027/257369/Writing_a_Game_Engine_from_Scratch__Part_1_Messaging.php