CGCG Category A

Notes by Vláďa Hořký with inputs from Jan Nekarda in the topics 5 and 6. Based on the lectures, slides and random stuff from the internet.

CGCG Category A 1
1 Transformations 2
Homogenní souřadnice 3
Příklady matic 3
Rotace 5
Euler angles 5
Gimball Lock 6
Quaternions 7
Příklad 8
Coordinate spaces 10
Viewing (LookAt) and perspective projection transformations 11
LookAt transformace 11
Perspective projection 11
2 Graphics pipeline 13
Why we need GPU? 13
Složitější architektura GPU 15
Graphics primitives 16
Textury 17
Filtrování 17
MIPMAPs 18
Graphics pipeline 18
Backface culling 19
Rasterizace 19
Pipeline 20
3 Shaders 22
Data types 22
Shader examples 23
Vertex shader 23
Tesselation shader 25
Geometry shader 27
Fragment shader 28
Compute shader 29
4 Game engine architecture 32
Engine, framework, middleware 32
Engine architecture 32
Game-specific subsystems 32
Game Agnostic Layer 33
Foundations 33

Asset pipeline.....34
Resource management.....34
Core layers.....35
Rendering Engine.....35
Event system.....36
Optimalizace.....36
Memory management.....36
String Handling.....38
Main Loop.....38
Single threaded loops.....38
Multi threaded loops.....39
5 Screen-space shading.....41
Forward rendering.....41
Deferred rendering.....41
GTA V notes.....41
Lights rendering.....42
Water rendering.....42
Ambient Occlusion.....43
6 Procedural content creation for games.....44
Terrain.....44
Vegetace.....46
V praxi (Ghost Recon).....47
Nedostatky PCG obsahu.....48
7 Animation.....49
Formats and compression.....49
Quake 2 .md2 format.....49
Komprese z Rise: Son of Rome.....50
Bone animation.....50
Problems.....51
Animation pipeline, animation graph.....51
Forward and inverse kinematics.....51
IK solvers.....52
Facial animations.....52
Motion capture.....53
Physically based animations.....54
Vlasy.....55
Anatomical body models.....56
Optimalizace.....56

1 Transformations

Vectors, matrices, homogeneous coordinates. Translation, scale, shear.
Rotation: Euler angles, gimbal lock, quaternions.
Coordinate spaces (model, world, view, clip, NDC).

Viewing (LookAt) and perspective projection transformations.

Homogenní souřadnice

Ve 3D můžeme pomocí matic dělat rotaci, zkosení (sheer), zrcadlení. Můžeme to skládat do sebe tím, že postupně zleva násobíme matice mezi sebou (první transformace je úplně vpravo). Ale NEJDE posunutí - to se musí řešit přičítáním.

Posunutí ale dost často potřebujeme (např. přechod mezi model space a world space, nebo world space a camera space…). Chceme ho rychlé a chceme ho skládat s ostatníma a invertovat. Proto:
Rozšíříme souřadnice na 4D (= homogenní). Díky tomu můžeme všechny transformace reprezentovat maticí! 4. homogenní složka se často nazývá w - a in general chceš, aby byla 1.

Když w není jedna, tak to tím celé vydělíš. To se pak hodí v projekci:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ y \end{bmatrix} \sim \begin{bmatrix} x/y \\ y/y \\ 1 \end{bmatrix}$$

A proto to takhle vlastně vypadá ta projekční matice (last row):

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

Příklady matic

https://www.gatevidyalay.com/3d-shearing-in-computer-graphics-definition-examples/

- posunutí/translace

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix}$$

- scale

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x v_x \\ s_y v_y \\ s_z v_z \\ 1 \end{pmatrix}$$

- zkosení/sheer

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ Sh_y & 1 & 0 & 0 \\ Sh_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Shearing Matrix

(In X axis)

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & Sh_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & Sh_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Shearing Matrix

(In Y axis)

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & Sh_x & 0 \\ 0 & 1 & Sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Shearing Matrix

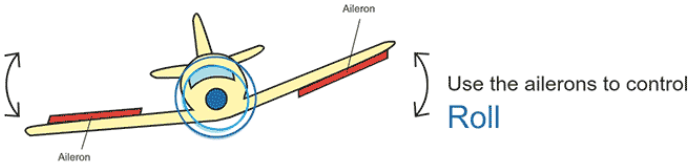
(In Z axis)

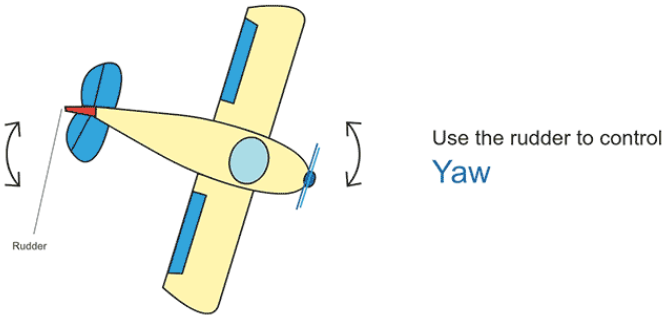
- reflection se dělá jednotkovou maticí, kde máš jednu -1

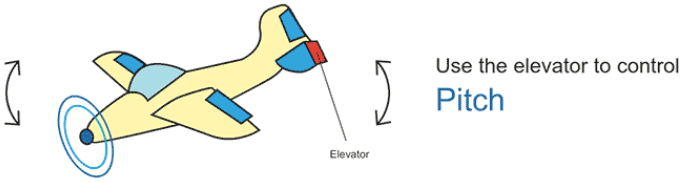
Rotace

Euler angles

reprezentuješ rotaci 3D objektů pomocí 3 úhlů (= 3 rotací podle 3 os). Přirozené pro nás. Záleží na pořadí, v jakém rotace aplikuješ a taky záleží na tom, jestli je aplikuješ v neměnném souřadnicovém systému, nebo v tom, který se rotuje (tj. kde 1. rotace změní osu další rotace). Často se používá “yaw, pitch, roll” třeba pro kameru (původně z letadel).

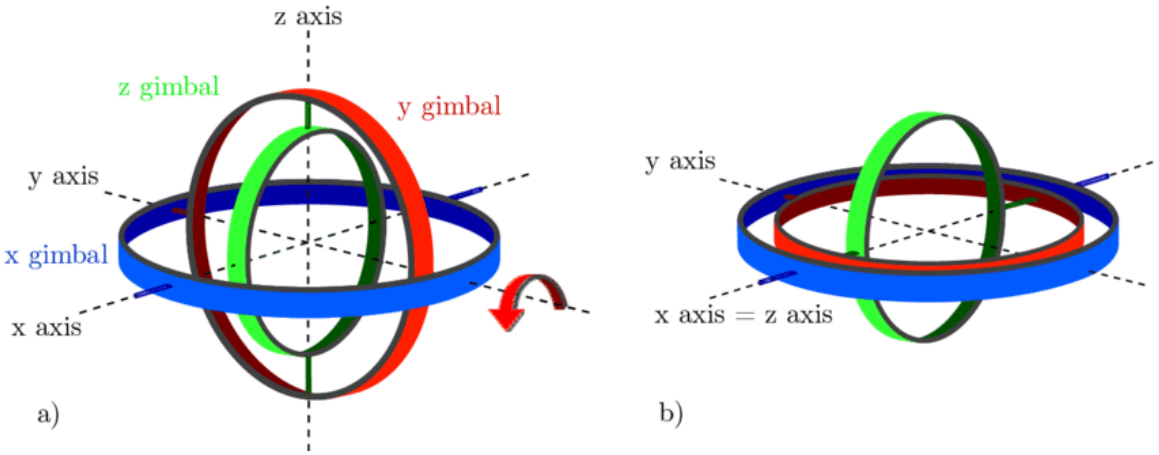




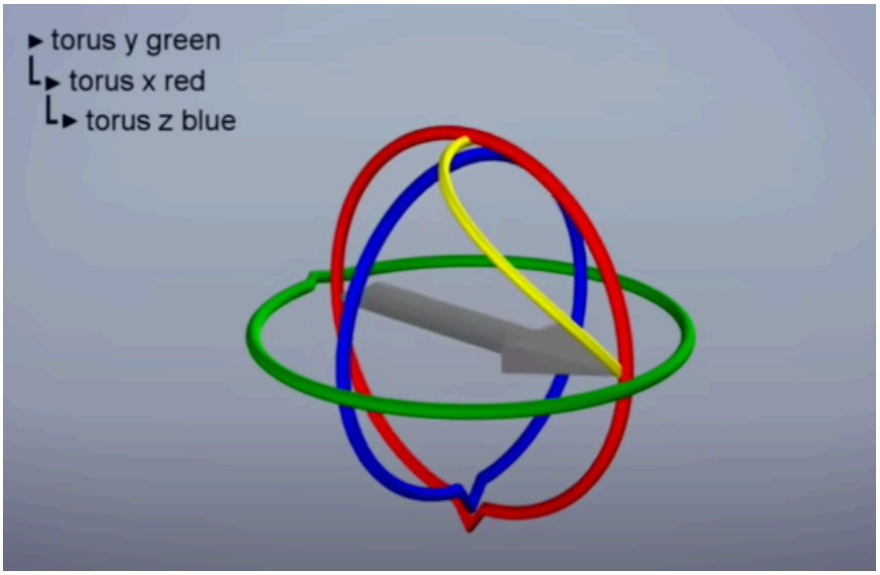


Gimball Lock

Když se ti to zrotuje tak, že máš 2 osy na sobě (= ztratíš 1D svobody)



Z locku to jde (vždy? jen někdy?) nějak odrotovat, ale musíš rotovat všechny kruhy (nebo min víc) A má to pak divnou trajektorii (chtěl jsi kameru třeba prostě jen otočit doprava).



Jak se locku vyhnout? Před rotováním změníš parenting structure torů. Ale pro každou parent structure může k locku nějak dojít. (máš 6 možných parenting struktur).

Quaternions

Každá rotace pomocí euler angles jde reprezentovat jedinou rotací o nějaký úhel podél fixed osy procházející fixed bodem! Kvaterniony tohle dobře kódují 🙂

4D rozšíření komplexních čísel, reprezentuje rotace
q = a + b**i** + c**j** + d**k**
q = w + x**i** + y**j** + z**k**
kde w,x,y,z jsou reálná čísla co tvoří kvaternion a i,j,k jsou imaginární složky. w je reálná složka

Multiplication of basis elements [edit]

The multiplication with 1 of the basis elements $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ is defined by the fact that 1 is a multiplicative identity, that is,

$$\mathbf{i}\,1 = 1\,\mathbf{i} = \mathbf{i}, \qquad \mathbf{j}\,1 = 1\,\mathbf{j} = \mathbf{j}, \qquad \mathbf{k}\,1 = 1\,\mathbf{k} = \mathbf{k}\,.$$

The products of other basis elements are

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1,$$
$$\mathbf{i}\,\mathbf{j} = -\mathbf{j}\,\mathbf{i} = \mathbf{k}, \qquad \mathbf{j}\,\mathbf{k} = -\mathbf{k}\,\mathbf{j} = \mathbf{i}, \qquad \mathbf{k}\,\mathbf{i} = -\mathbf{i}\,\mathbf{k} = \mathbf{j}.$$

Combining these rules,

$$\mathbf{i}\,\mathbf{j}\,\mathbf{k} = -1.$$

U kvaternionu nezáleží na pořadí rotací - 1 kvaternion je prostě jedna rotace.

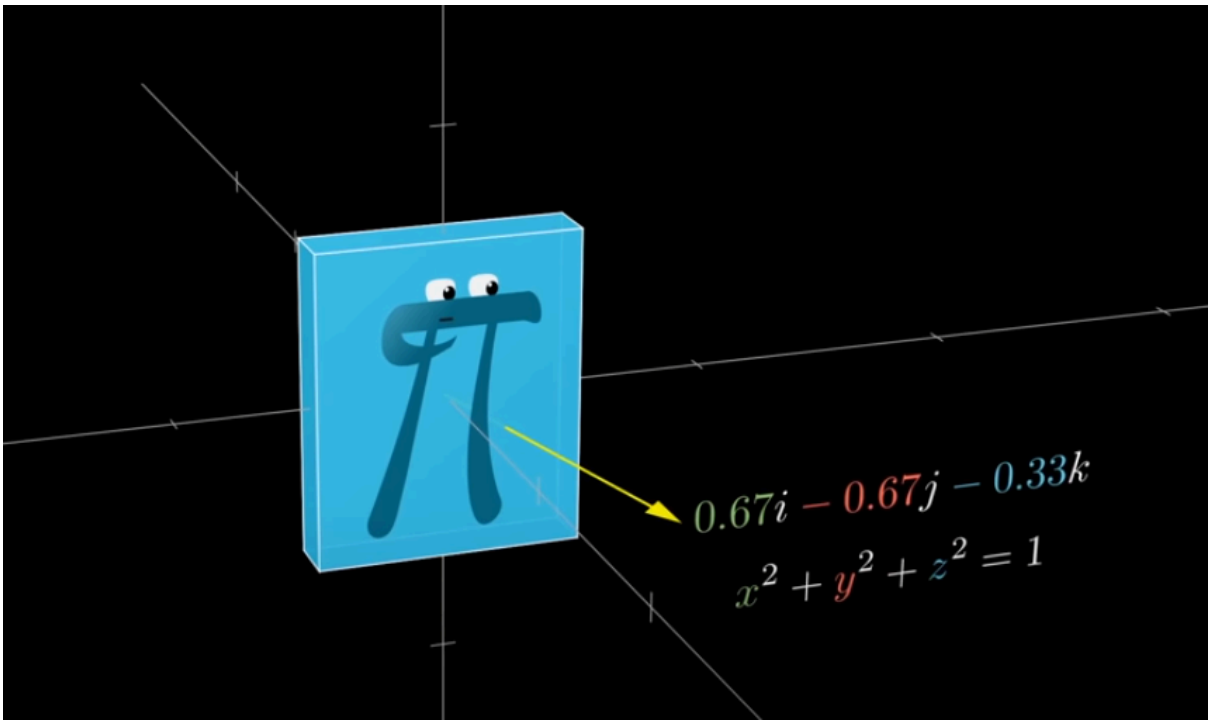
Kvaterniony nemají Gimball Lock.
Pomocí obojího můžeš udělat všechny rotace! Rozdíl je v tom, jak rotaci interpretuješ (= jak vypadá ta animace).
Kvaterniony nejsou ordered - je to smooth.
BACHA, kvaterniony berou nejkratší cestu - např. full backflip neudělají nic 😀 Musíš to rozdělit na několik rotací.
..Rotuješ pomocí SLERP (Spherical interpolation).

Pro rotaci používáš jednotkové kvaterniony (jejich norma je jedna).

Příklad

Chceš rotovat o úhel U = 40deg kolem osy O.

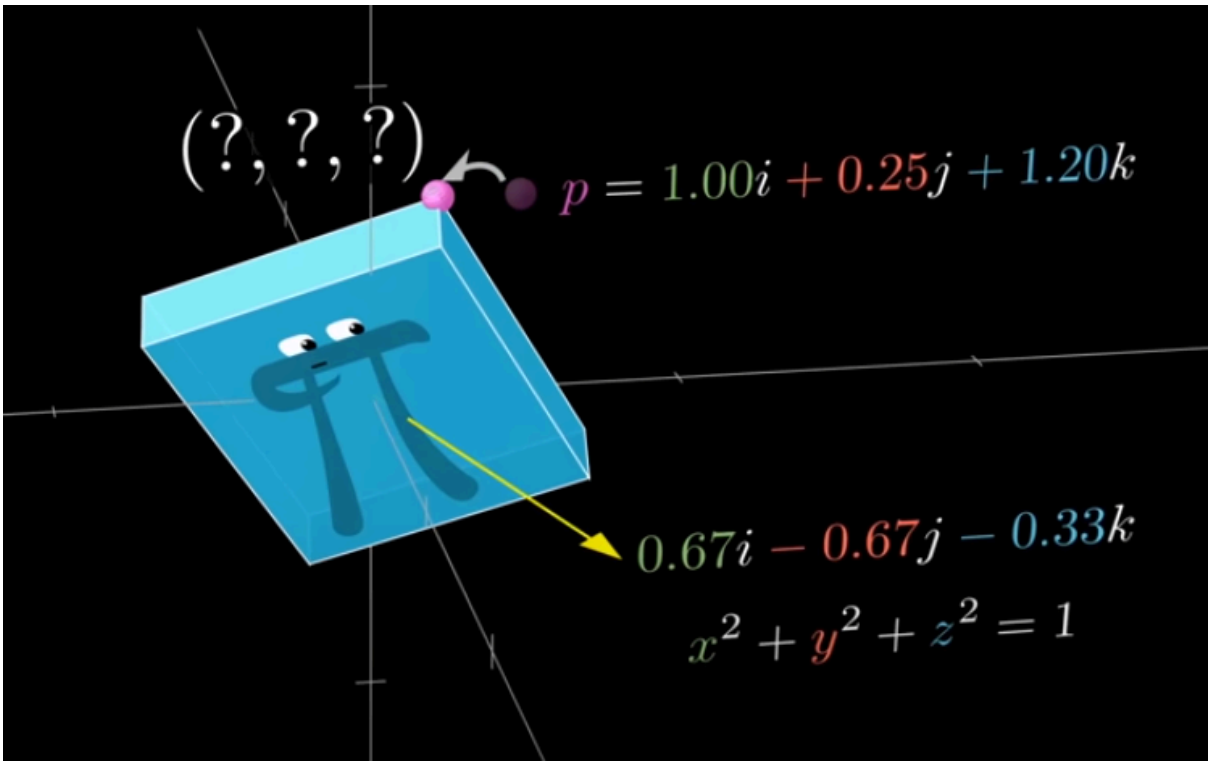
- definuješ osu O pomocí x,y,z a tak, abys měl součet 2. mocnin = 1 (složku w bereš jako 0)



- pak vyrobíš kvaternion (bereš ½ úhlu):

$$\underbrace{\left(\cos(40°/2) + \sin(40°/2)(0.67i - 0.67j - 0.33k)\right)}_{q}$$

- máš nějakej bod p (třeba vertex) a chceš vědět jeho výslednou pozici po rotaci



- uděláš “kvaternion sandwitch”

$$p \rightarrow q \cdot p \cdot q^{-1}$$

je potřebuješ znát pravidla pro násobení kvaternion

$$\underbrace{(w_0 + x_0 i + y_0 j + z_0 k)}_{q}\underbrace{(x_1 i + y_1 j + z_1 k)}_{p}\underbrace{(w_0 - x_0 i - y_0 j - z_0 k)}_{q^{-1}}$$

= výsledná pozice bodu p po rotaci

Násobení kvaternionů

A quaternion of the form a + 0 i + 0 j + 0 k, where a is a real number, is called **scalar**, and a quaternion of the form 0 + b i + c j + d k, where b, c, and d are real numbers, and at least one of b, c, or d is nonzero, is called a **vector quaternion**. If a + b i + c j + d k is any quaternion, then **a** is called its **scalar part** and **b i + c j + d k** is called its **vector part**.

If a quaternion is divided up into a scalar part and a vector part, that is,

$$\mathbf{q} = (r,\ \vec{v}),\ \ \mathbf{q} \in \mathbb{H},\ \ r \in \mathbb{R},\ \ \vec{v} \in \mathbb{R}^3,$$

then the formulas for addition, multiplication, and multiplicative inverse are

$$\begin{aligned}(r_1,\ \vec{v}_1) + (r_2,\ \vec{v}_2) &= (r_1 + r_2,\ \vec{v}_1 + \vec{v}_2)\,,\\ (r_1,\ \vec{v}_1)(r_2,\ \vec{v}_2) &= (r_1 r_2 - \vec{v}_1 \cdot \vec{v}_2,\ r_1\vec{v}_2 + r_2\vec{v}_1 + \vec{v}_1 \times \vec{v}_2)\,,\\ (r, \vec{v})^{-1} &= \left(\frac{r}{r^2 + \vec{v}\cdot\vec{v}}, \frac{-\vec{v}}{r^2 + \vec{v}\cdot\vec{v}}\right)\,,\end{aligned}$$

where "·" and "×" denote respectively the dot product and the cross product.

Coordinate spaces

(model, world, view, clip, NDC)

= souřadnicové systémy používané v rendering pipeline, these coordinate spaces help in positioning, orienting, and projecting 3D objects onto a 2D screen.

- Model space (local space)
    - lokální systém toho modelu (třeba v Blenderu), origin někde rozumně a tak
    - popisuje geometrii objektu nezávisle na jeho umístění ve scéně
- World space
    - systém, který definuje pozici a rotaci všech objektů ve scéně
    - globální a sdílený všemi objekty (nějaký origin a osy ve scéně) (=Transform v unity je ve world space)
    - z model space do world space transformuješ pomocí transformační matice modelu (obsahuje translaci, rotaci, scaling (sheer?))
- View space (camera space, eye space)
    - souřadnicový systém z perspektivy kamery;
      kamera je v počátku, osy (nejspíš) odpovídají yaw, pitch roll kamery; kamera se kouká ve směru -z
    - celý svět z world space pak transformuješ do camera space pomocí view transformation matrix
- Clip space
    - chápu to jako souřadnice pixelu na screenu + depth složka
    - dostaneme se tam pomocí projection matrix (různé druhy projekce: perspektivní, ortogonální)
- Normalized Device Coordinates (NDC)
    - všechny souřadnice normalizované do (-1,1) range (x,y i z)
    - obsahuje to perspective division - dělíš vše w (homogenní složkou, 4. dimenzí)
- Window - pozice pixelu relativní k poloze okna na obrazovce

- Pak to ještě mapneš na screen space (pomocí screen width a height, máš to už normalizované.

Viewing (LookAt) and perspective projection transformations

LookAt transformace

- potřebuje camera position, camera target (na co se dívá nebo forward vector), camera up vector
- LookAt udělá, aby se kamera koukala na cíl
- reálně používaš její inverz - ViewingMatrix

$$translationX = dot(positionOfCamera, rightVector);$$
$$translationY = dot(positionOfCamera, upVector);$$
$$translationZ = dot(positionOfCamera, forwardVector);$$

$$lookAt = \begin{bmatrix} R_x & R_y & R_z & t_x \\ U_x & U_y & U_z & t_y \\ F_x & F_y & F_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Magic LookAt Matrix

$$viewMatrix = \begin{bmatrix} R_x & U_x & F_x & 0 \\ R_y & U_y & F_y & 0 \\ R_z & U_z & F_z & 0 \\ -t_x & -t_y & -t_z & 1 \end{bmatrix}$$

The inverse LookAt matrix is the ViewMatrix

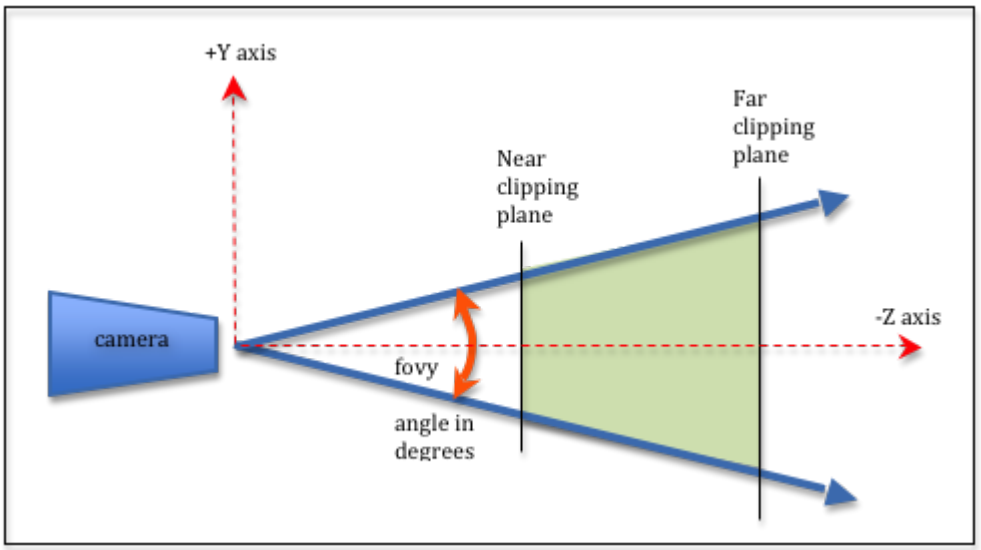
EXAMPLE: ORIENTING THE CAMERA. A common task in graphics is to orient the camera so that it looks at a certain position. Here we will present what `gluLookAt()` (from the OpenGL Utility Library, GLU for short) does. Even though this function call itself is not used much nowadays, the task remains common. Assume that the camera is located at $\mathbf{c}$, that we want the camera to look at a target $\mathbf{l}$, and that a given up direction of the camera is $\mathbf{u}'$, as illustrated in Figure 4.5. We want to compute a basis consisting of three vectors, $\{\mathbf{r}, \mathbf{u}, \mathbf{v}\}$. We start by computing the view vector as $\mathbf{v} = (\mathbf{c} - \mathbf{l})/||\mathbf{c} - \mathbf{l}||$, i.e., the normalized vector from the target to the camera position. A vector looking to the "right" can then be computed as $\mathbf{r} = -(\mathbf{v} \times \mathbf{u}')/||\mathbf{v} \times \mathbf{u}'||$. The $\mathbf{u}'$ vector is often not guaranteed to be pointing precisely up, so the final up vector is another cross product, $\mathbf{u} = \mathbf{v} \times \mathbf{r}$, which is guaranteed to be normalized since both $\mathbf{v}$ and $\mathbf{r}$ are normalized and perpendicular by construction. In the camera transform matrix, $\mathbf{M}$, that we will construct, the idea is to first translate everything so the camera position is at the origin, $(0, 0, 0)$, and then change the basis so that $\mathbf{r}$ is aligned with $(1, 0, 0)$, $\mathbf{u}$ with $(0, 1, 0)$, and $\mathbf{v}$ with $(0, 0, 1)$. This is done by

$$\mathbf{M} = \underbrace{\begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{change of basis}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{translation}} = \begin{pmatrix} r_x & r_y & r_z & -\mathbf{t} \cdot \mathbf{r} \\ u_x & u_y & u_z & -\mathbf{t} \cdot \mathbf{u} \\ v_x & v_y & v_z & -\mathbf{t} \cdot \mathbf{v} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.20}$$

Note that when concatenating the translation matrix with the change of basis matrix, the translation $-\mathbf{t}$ is to the right since it should be applied first. One way to remember where to put the components of $\mathbf{r}$, $\mathbf{u}$, and $\mathbf{v}$ is the following. We want $\mathbf{r}$ to become $(1, 0, 0)$, so when multiplying a change of basis matrix with $(1, 0, 0)$, we can see that the first row in the matrix must be the elements of $\mathbf{r}$, since $\mathbf{r} \cdot \mathbf{r} = 1$. Furthermore, the second row and the third row must consist of vectors that are perpendicular to $\mathbf{r}$, i.e., $\mathbf{r} \cdot \mathbf{x} = 0$. When applying the same thinking also to $\mathbf{u}$ and $\mathbf{v}$, we arrive at the change of basis matrix above. □

Perspective projection

- máš definované Viewing Volume (Frustum) = taková ta pyramida co vidí kamera
  - Field of View - FOV
  - Aspect Ratio
  - Near and Far planes (odkud už a kam až)



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

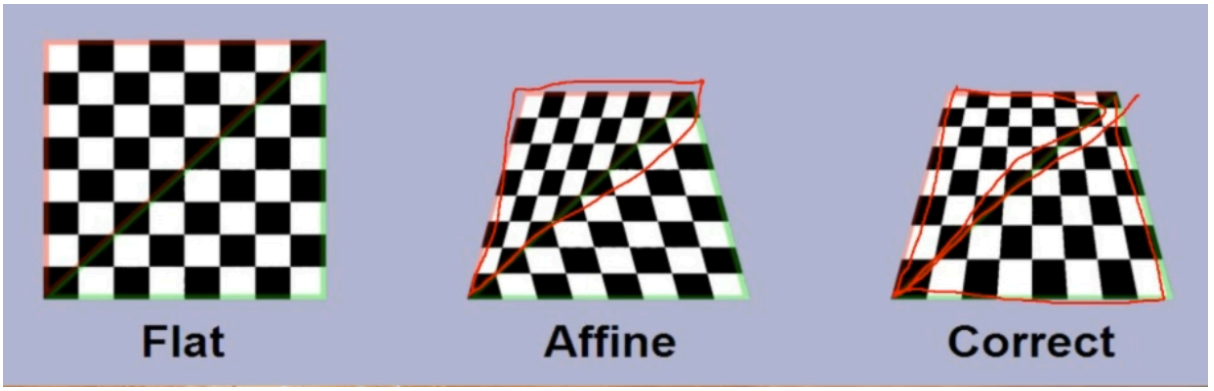
-

PERSPECTIVE MATRIX

$$\begin{pmatrix} \frac{1}{\text{ASPECT}\cdot\text{TAN}(\text{FOV}/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\text{TAN}(\text{FOV}/2)} & 0 & 0 \\ 0 & 0 & \frac{\text{NEAR}+\text{FAR}}{\text{NEAR}-\text{FAR}} & \frac{2\cdot\text{NEAR}\cdot\text{FAR}}{\text{NEAR}-\text{FAR}} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

-

- Musíš dělat perspective correction texture mapping



2 Graphics pipeline

General GPU architecture (Compare to CPU).
Types of graphics primitives.
Textures, sampling, and mip-mapping.
The processing chain of the graphics pipeline: vertex processing (vertex, tessellation, and geometry shaders), viewport transform, primitive assembly, rasterization, fragment processing.

Why we need GPU?

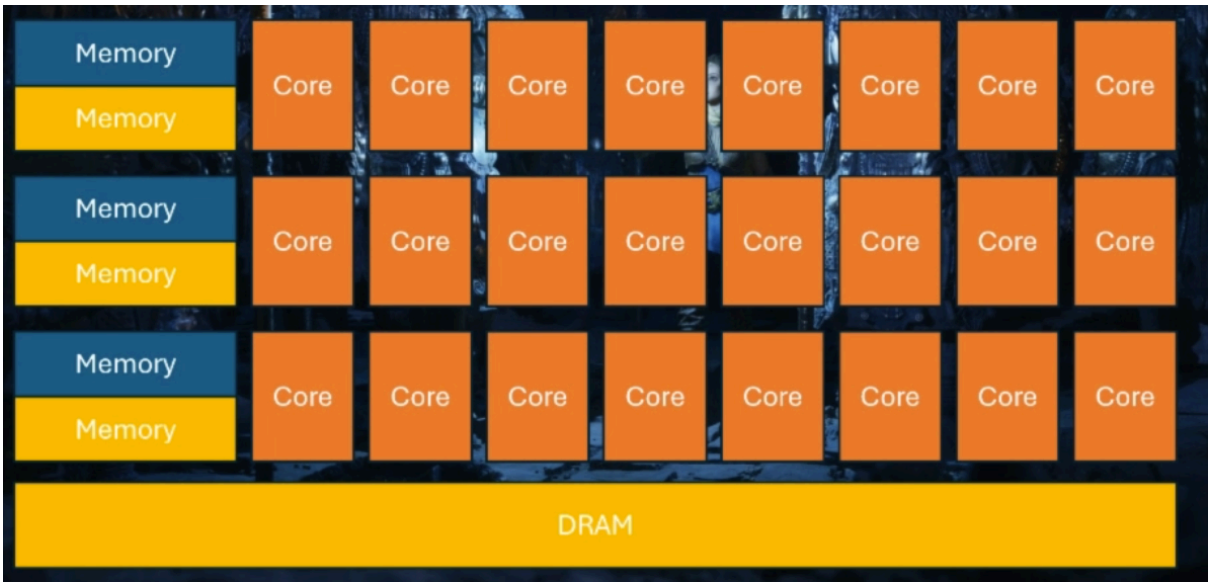
CPU

- hodně různých instrukcí
- složité operace (velké instruction sets, komplikované instrukce)
- dává požadavky (jako “GPU nakresli něco”)
- je omezené v tom co zvládne (počet jader)
- může mít efficient cores (šetří energii, ale zvládne přijmou email)
- hodně složitý design (na výrobu); jsou tam starý (asi unused?) obvody
    - nové cpus se snaží mít méně instrukcí a být více optimized
- Simplified architektura (asi 1 core)

    - ALU - Arithmetic unit
    - Control unit (to ovládá)
    - Cache = rychlá paměť; různé lvly (L1, L2, L3)

GPU

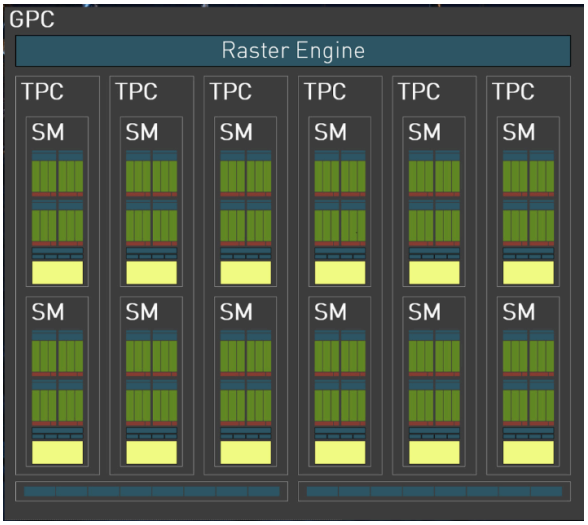
- máme fakt hodně (miliony) pixelů = hodně práce
- jednodušší instrukce
- hodně jader a memory levelů
- paralelizace (počítáš “to samé” pro každý pixel, nedopočítáš jeden a už rozjedeš další…)
- simple architektura

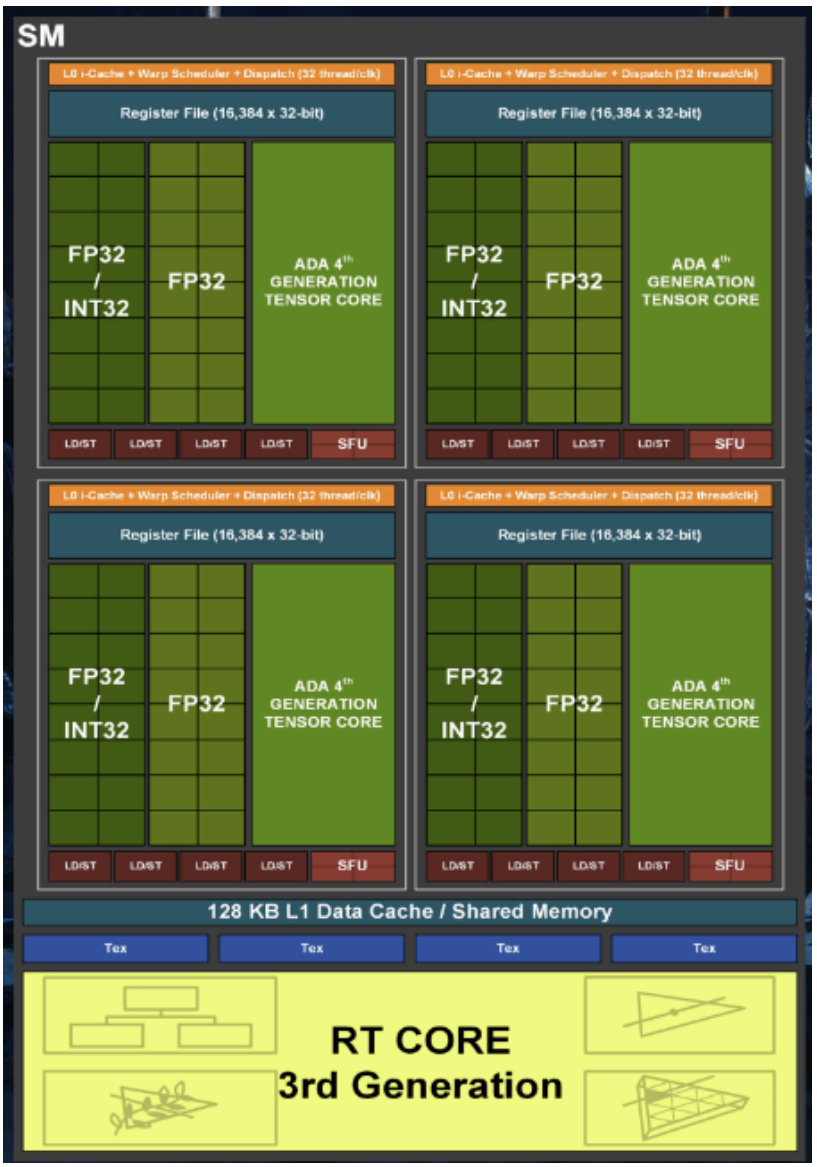


  - stovky tisíc jader
  - různé druhy paměti, např. skupina jader má přístup na ten a ten druh paměti
  - Overall myšlenka
    - hromada malých jader, paměť co je spojuje, pak poskládáš malé kousíčky co počítají jádra do sebe a máš výsledek
  - Nedostatky
    - branching (if else)
    - těžká matika (můžeš použít lookup tables)
    - nemá to rádo state changes ((un)binding buffers, changing textures)
    - vysoký polygon count
    - velký resolution (renderuješ na menší res a pak upscaluješ)
  - Už je to programovatelné, nemusíš renderovat (ale jen něco počítat), nejsou specialized bloky (až na ratracing…)

Složitější architektura GPU

- (zelené pixely jsou) jádra do skupinek, skupinky do skupinek atd…
- Vše spojené L2 cache přes kterou komunikují





= brain, co zpracovává shadery

- jsou tu (oranžové) caches
- spousta úrovní paměti - něco uděláš a uložíš, někdo to vezme a složí s něčím…

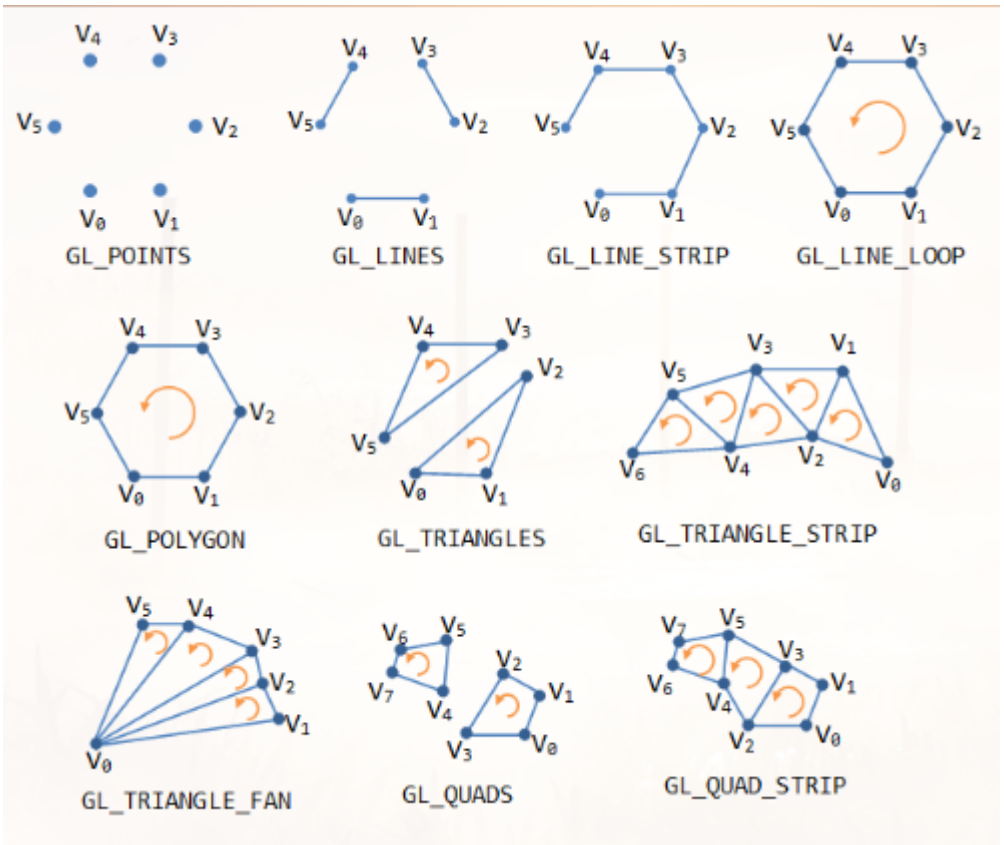
Graphics primitives

Vertex

- pozice, normála, texture coord (UVs)
- cokoliv dalšího (barva, teplota…)

Vrcholy načteš do arrays = Vertex Buffer Objects (VBO)

Jednotlivá primitiva:



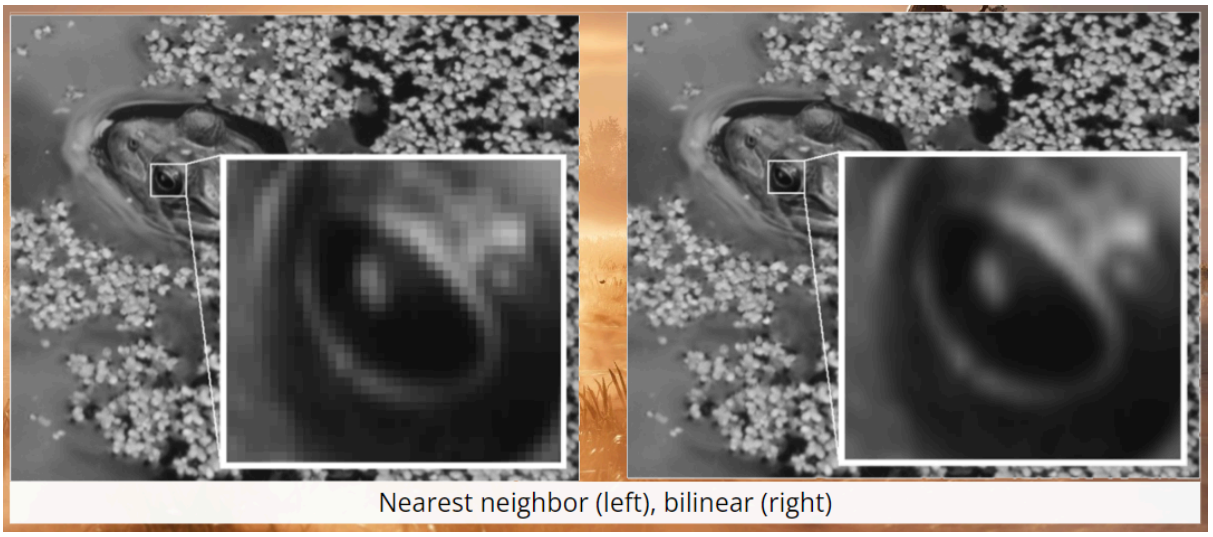
Textury

= 1D, 2D nebo 3D data

- propojené s modelem pomocí texture coordinates (UVs)
- načteš je do GPU a pak sampluješ
- často je filtruješ (Bilinear, bicubic, anisotropic)

Filtrování

- když máš objekt daleko, tak pixely na textuře objektu jsou mnohem menší než pixely na obrazovce; chceš se zbavit “jagged edges”

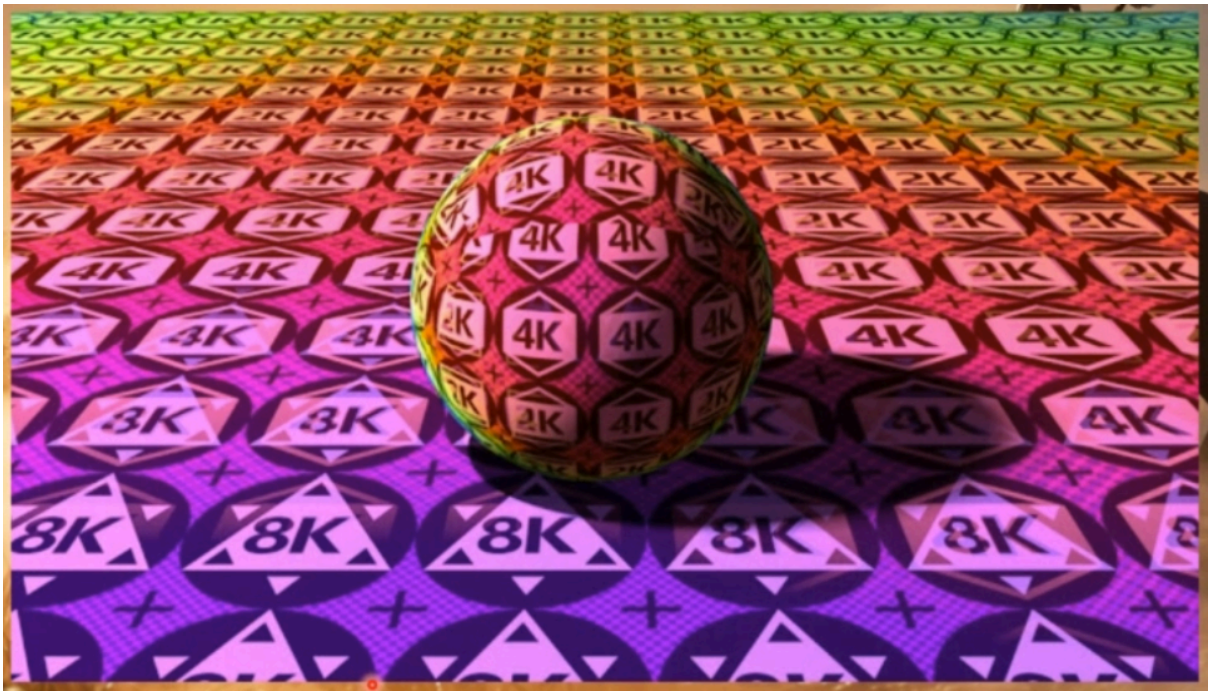


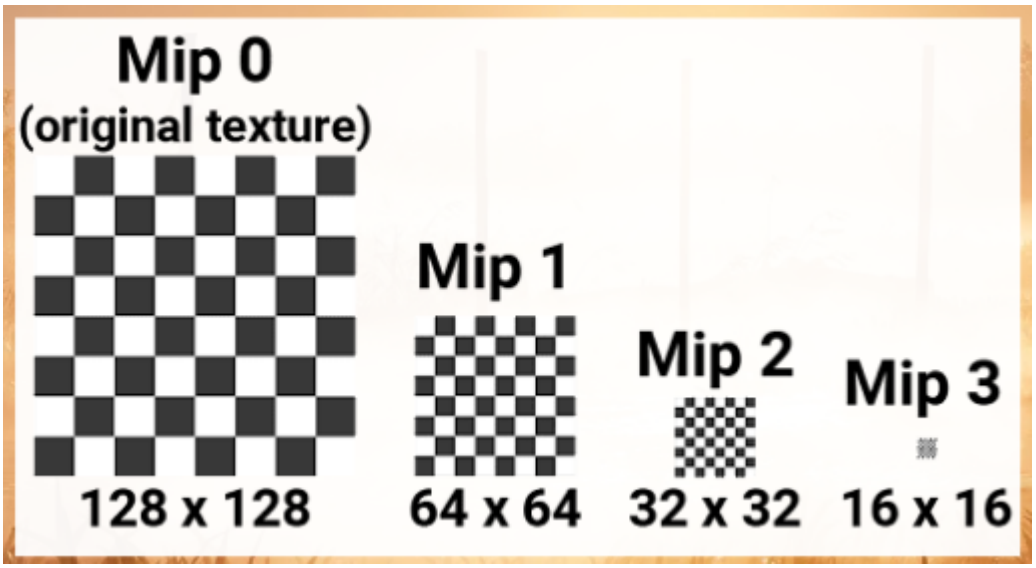
Nearest neighbor (left), bilinear (right)

MIPMAPs

= texture LODs

- čím jsi dál, tím menší textury používáš; nechceš samplovat 4k texturu na věc co je mega daleko
- nezabere to o tolik víc paměti
- MIPMAPs se ti dělají samy, polygon LODs musíš řešit
- menší textura se vejde lépe do cache; máš předpočítanou menší texturu dopředu, máš méně čtení v paměti





Graphics pipeline

- Vertex shader
- Tesselation (optional)
- Geometry (optional)
- Primitive assembly (face culling)

- Viewport transform
- Rasterization to pixels/fragments
- Pixel/fragment shader
- Sample processing

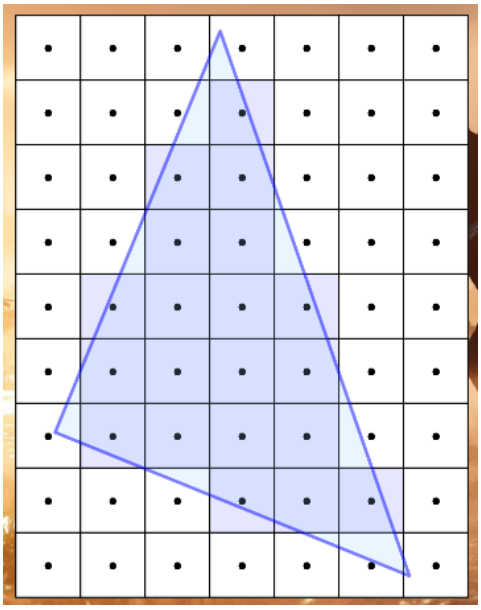
Nejvíce času většinou bere fragment shader.

Backface culling

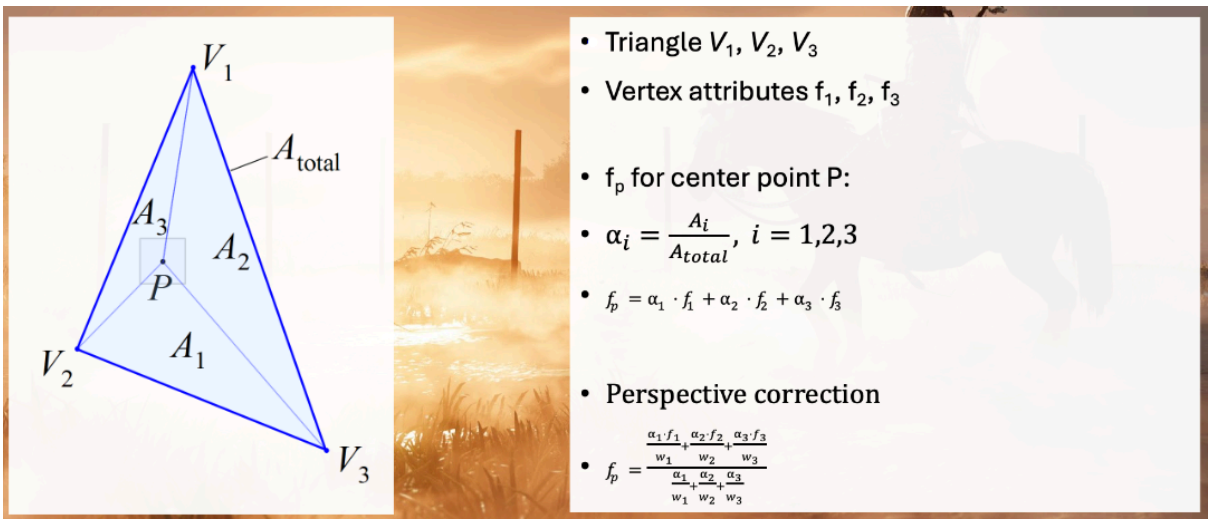
- nerenderuješ věci co vidíš “ze zadu”
- vypočteš normálu, koukneš jestli míří na kameru (ne = discard)

Rasterizace

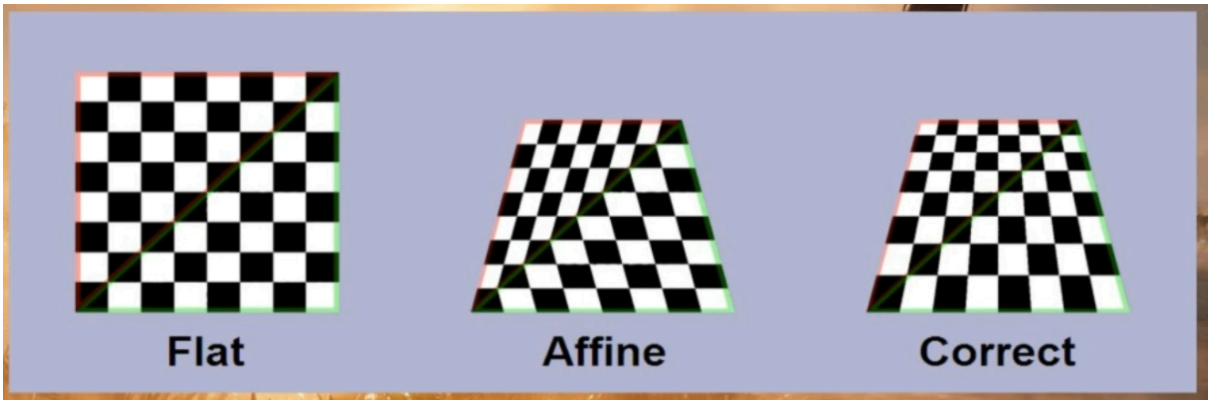
= potřebuješ to převést na pixely
jen vrcholy mají nějakou informaci (barvu); potřebuješ to zinterpolovat a mít ty hodnoty pro každý pixel



Interpolování slide (na přednášce ho skipnul)



Perspective correction
- nechceš s texturou pracovat naivně



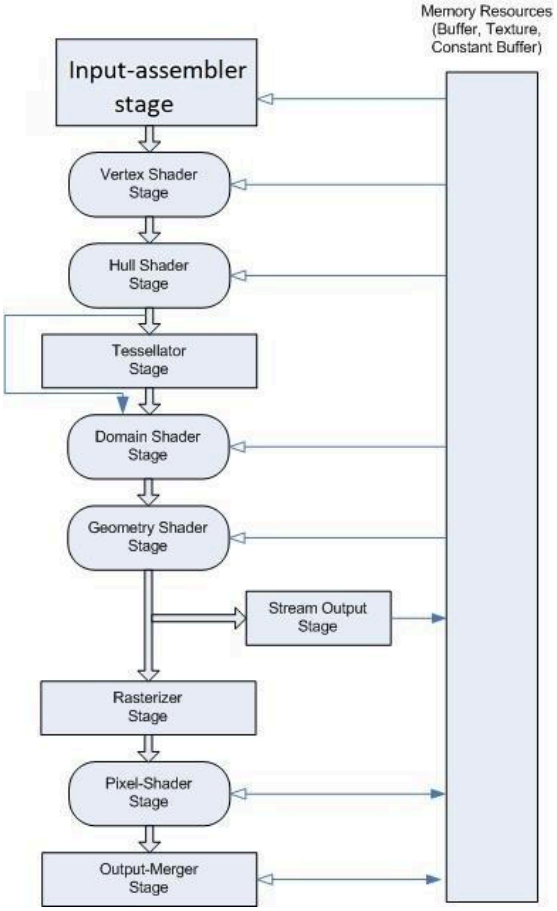
Per sample operations
- Multisampling (MSAA) - více samplů per pixel, ale fragment shader běží jen 1x
- Supersampling - tady běží celý fragment shader vícekrát

- Scissor test = discard fragments outside viewport
- Stencil a Depth test
  - pomocí Stencil a Depth bufferů
  - testuješ fragment value vs buffer value (a podle výsledku testu discarduješ, testy můžeš nastavit)
- Early fragment optimalizace
  - když FS nemění Z value (tak ho pustíš až po discardu)
  - když FS nediscarduje fragmenty
  - neděláš alpha blending

Pipeline

- Vertex Shader
  - Bere každý vrchol zvlášť
  - má za úkol ho transformovat z model/object space do clip space (pomocí model, view a projection matrix)

    - může se v něm počítat per vertex lightning
    - passuje data do dalších stages (barva, normály, UVs…)
- Tesselation (optional)
    - podrozděluje existující geometrii pro přidání detailu (např. víc smooth koule)
    - rozdělí geometrii (patches)
    - vypočte finální pozici podrozdělených vrcholů
- Geometry shader
    - zpracovává celá primitiva (trojúhelníky…)
    - může přidat, upravit i odebrat vrcholy, aby vytvořil novou geometrii
    - vhodný na shadow volumes, billboardy…
- Primitive assembly
    - dává dohromady vertexy do primitiv (trojúhelníků)



3 Shaders

Specific data types and constructs of shading languages (vectors, matrices, swizzling, etc.).
Types of shaders, their role in the processing pipeline.
Example use: vertex shader, tessellation shader, geometry shader, fragment/pixel shader, compute shaders.

- Shadery = programy běžící na GPU cores
- Hodně instancí běží souběžně na hodně jádrech

Shading languages
- GLSL v OpenGL a Vulkan
- HLSL - Direct3D
    - primární jazyk (pro cross compilaci = nemusíš to pak přepisovat)

Data types

GLSL	HLSL	Definition
bool	bool	Boolean true or false
float	float	FP-32
double	double	FP-64
int	int	Signed integer, 32-bits
unsigned int	uint	Unsinged integer, 32-bits

Type	Description	Examples
floatN	Represents a vector with N floating-point components	float4 (colors with alpha)
intN	Vectors with integer components	int3
uintN	Vectors with unsigned integer components	uint2
boolN	Vectors with boolean components	bool2
halfN	Half precision (FP-16) floating point number	half3
doubleN	Double precision (FP-64) floating point number	double2
floatMxN	Represents a matrix with M rows and N columns of floating-point values	float4x4

Swizzling

```
// Swizzling:
float3 bar = foo.xyz;
float3 all_x = foo.xxx;
float3 inv = foo.zyx;
```

Usage

```
shader.hlsl

// Explicit type:
float3 fVector = { 0.2f, 0.3f, 0.4f };
float2x2 fMatrix = { { 0.0f, 0.1f }, // row 1
                     { 2.1f, 2.2f } }; // row 2

// Using HLSL vector and matrix types directly:
float4 dVector = { 0.2, 0.3, 0.4, 0.5 };
float2x2 fMatrix2 = { { 0.0f, 0.1f }, // row 1
                      { 2.1f, 2.2f } }; // row 2

// Accessing matrix values
float element1 = fMatrix2[0][1]; // Using indices
float element2 = fMatrix2._01;   // Using the direct field access
```

Co tam máš:

- Matrices — arrays of column/row vectors
- Arrays (w/ defined length() method)
- C-like structures, conditionals, loops
- Lots of built-in functions for math, etc.
- Samplers for texture access
- Constants set from the main program

Debuguje se ne tak dobře jako na CPU. Můžeš použít např. RenderDoc - a koukat na jednotlivé draw cally, co obsahuje jaká textura…

Shader examples

Vertex shader

jediné required

- GLSL

```
shader.glsl

// OpenGL 2.1 - shader version 1.20
void main (void) {
  gl_Position = ftransform ( ) ;
}

// Since OpenGL 3.1 - shader version 1.40
#version 140
uniform Transformation {
  mat4 projection_matrix; // specified through
  mat4 modelview_matrix ; // uniform buffers
};

in vec3 vertex; // specified through vertex attribs (i.e. VBO)

void main (void) {
  gl_Position = projection_matrix
                * modelview_matrix
                * vec4(vertex , 1.0);
}
```

- HLSL

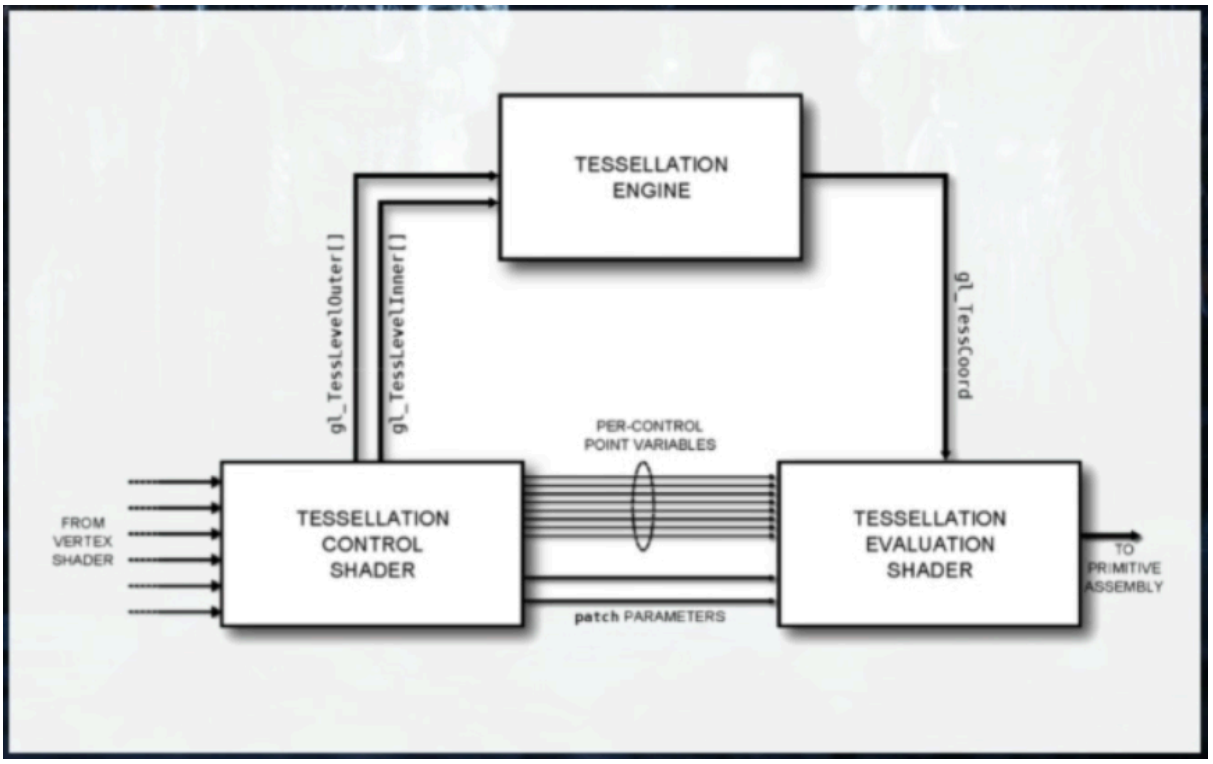
```
// Input stream from application
struct a2v {
  float4 Position : POSITION; // POSITION is an HLSL output semantic
};

// Output stream towards pixel shader
struct v2p {
  float4 Position : POSITION;
};

void main(in a2v IN, out v2p OUT, uniform float4x4 ModelViewMatrix)
{
  OUT.Position = mul(IN.Position, ModelViewMatrix);
}
```

Usage:
Displacement mapping (pohnes vrcholy na zaklade infa v texture (vodni hladina))
Mesh Distortion (vítr na stromy)
Asi taky:
Billboarding (orientuješ quads na kameru)
Skeletal animace

Tesselation shader

= dynamicky zvyšuješ geometry detail objektů;
podrozděluješ původní povrch na menší trojúhelníky;
nevytváříš nový polygon, jen podrozděluješ existující - nepridáváš vrcholy/polygony mimo původní polygon

Je to ze 2 částí



Control shader a evaluation shader; mezi tim je tesselator (=“fixed function”)

- Control Shader (= hull shader)
    - input: vertex data forming basic shapes (triangles, quads)
    - output: tessellation factor = jak moc to máš podrozdělit (např. based on camera distance)
- Tessellator
    - input: tessellation factor
    - output: subdivided mesh (nové vrcholy a primitiva)
- Evaluation Shader (=domain shader)
    - input: subdivided mesh
    - output: final pozice vrcholů (a jejich params)
    - běží pro každý tessellated vertex

- Treats vertices as control points
- Runs in batches, once per control point
- Determines tessellation parameters
- Passes parameters down the line

Usage:
“Dynamic LODs”
Terrain rendering (fine details when close)
- extra data máš v textuře

Example
Control Shader:

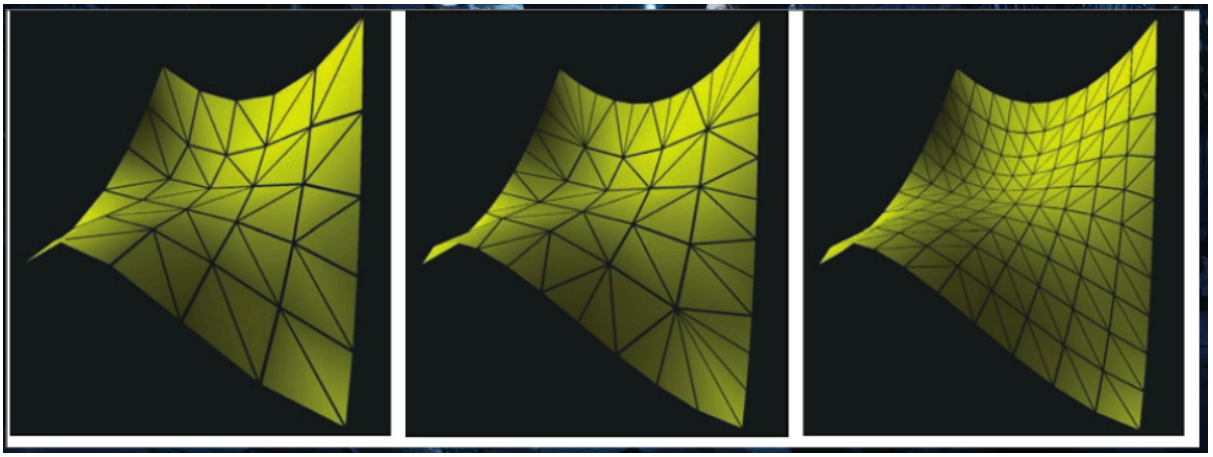
shader.glsl

```
#version 400

layout (vertices = 4) out; // each tessellated patch is a quad

void main(void) {
  // Set just once per patch
  if (gl_InvocationID == 0) {
    gl_TessLevelInner[0] = 9.0; gl_TessLevelInner[1] = 7.0;
    gl_TessLevelOuter[0] = 3.0; gl_TessLevelOuter[1] = 5.0;
    gl_TessLevelOuter[2] = 3.0; gl_TessLevelOuter[3] = 5.0;
  }

  // Pass through the unchanged input position
  gl_out[gl_InvocationID].gl_Position =
    gl_in[gl_InvocationID].gl_Position;
}
```

Evaluation shader:

shader.glsl

```
#version 400

// We want triangles, equal spacing, ccw winding order
layout(triangles,equal_spacing,ccw) in;

void main(void) {
  // Barycentric coords * real vertex positions
  gl_Position = gl_TessCoord.x * gl_in[0].gl_Position +
                gl_TessCoord.y * gl_in[1].gl_Position +
                gl_TessCoord.z * gl_in[2].gl_Position;
}
```



Geometry shader

- input je celé primitivum, může to vytvořit nové vrcholy, nebo nějaké odstranit
- zpomaluje to (GPU nemá rádo změny)

shader.glsl

```
// Simple pass-through geometry shader
#version 430 core

layout (triangles) in; // Take triangles
layout (points, max_vertices = 3) out;

void main(void) {
  for (int i = 0; i < gl_in.length(); ++i) {
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
  }
  // Explicit call, called by default at the end
  EndPrimitive();
}
```

Usage:
Soft particles rendering

- Single pass rendering into a cubemap
- Cascaded shadow maps in one pass
- Point sprite generation
- Shadow volume extrusion
- Pre-CS geometry processing using transform feedback
- ...

Fragment shader

= běží pro každý fragment;

- přiřadí mu barvu
  - světlo, stíny, ambient occlusion, odlesky
- může to být expensive (4k = 8M+ pixelů)
  - snaž se zabránit overdraw; NEdělat HEAVY computations

shader.glsl

```
#version 430 core

out vec4 fragColor; // Output color of the fragment

void main()
{
    fragColor = vec4(1.0, 0.5, 0.0, 1.0); // Output an orange color
}
```

Můžeme použít také pro postprocessing

- předrenderuješ si do render targetu (textury)
- pak to dáš na quad
- nový fragment shader s postprocessing effectem

Compute shader

Na přednášce se to nestihlo :/

- Form effectively a separate single-stage pipeline
- Perform arbitrary parallel computations
- Like CUDA or OpenCL
- Run in work groups over work items

Usage:

- Simulations (particles, water, ...)
- Image postprocessing
- Instance chain creation for indirect drawing
- GPU scene preparation (frustum culling, etc.)
- GPU occlusion culling
- Software rasterization
- ...

Workgroups: Compute shaders operate on groups of threads called workgroups. Each workgroup contains multiple threads that execute the shader code in parallel.

Invocations: Each thread (or invocation) within a workgroup processes a specific portion of the data. The total number of invocations is typically defined by the dimensions of the workgroups and the number of workgroups dispatched.

Example na blur:

Workflow

1. **Initialization**: Load the input image into `inputImage` and create an empty `outputImage`.
2. **Dispatch**: Call the compute shader with enough workgroups to cover the image dimensions.
3. **Execution**: Each thread processes a portion of the image, applying the blur filter.
4. **Output**: The blurred image is written to `outputImage`.

Shader Code (GLSL)

glsl Zkopírovat kód

```glsl
#version 430

layout(local_size_x = 16, local_size_y = 16) in;

layout(binding = 0, rgba8) uniform image2D inputImage;
layout(binding = 1, rgba8) uniform image2D outputImage;

void main() {
    ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
    vec4 color = vec4(0.0);

    // Simple blur kernel
    for (int x = -1; x <= 1; x++) {
        for (int y = -1; y <= 1; y++) {
            color += imageLoad(inputImage, pixelCoords + ivec2(x, y));
        }
    }
    color /= 9.0; // Average the color

    imageStore(outputImage, pixelCoords, color);
}
```

Example:

shader.glsl

```
// Image inversion compute shader
#version 430 core

layout (local_size_x = 32, local_size_y = 32,
  local_size_z = 1) in; // 32*32*1 local work group

layout (binding = 0, rgba32f) uniform image2D img_in;
layout (binding = 1) uniform image2D img_out;

void main(void) {
  // gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize
  // + gl_LocalInvocationID; i.e. get pixel position
  ivec2 p = ivec2(gl_GlobalInvocationID.xy);

  vec4 texel = vec4(1.0) - imageLoad(img_in, p);
  imageStore(img_out, p, texel);
}
```

Example CPU:

shader.glsl

```
// Bind input image
glBindImageTexture(0, tex_input, 0, GL_FALSE,
  0, GL_READ_ONLY, GL_RGBA32F);

// Bind output image
glBindImageTexture(1, tex_output, 0, GL_FALSE,
  0, GL_WRITE_ONLY, GL_RGBA32F);

// Dispatch the compute shader
// (IMAGE_WIDTH / 32) * (IMAGE_HEIGHT / 32) work groups
glDispatchCompute(IMAGE_WIDTH / 32, IMAGE_HEIGHT / 32, 1);
```

4 Game engine architecture

What is the difference between a game engine, game framework, and game middleware?
Game engine component layers and its functionalities.
What are common core layers and platform independence subsystems?
How are memory and resources handled in game engines?
Explain “main loop” and how it can behave in single vs. multi threaded code.

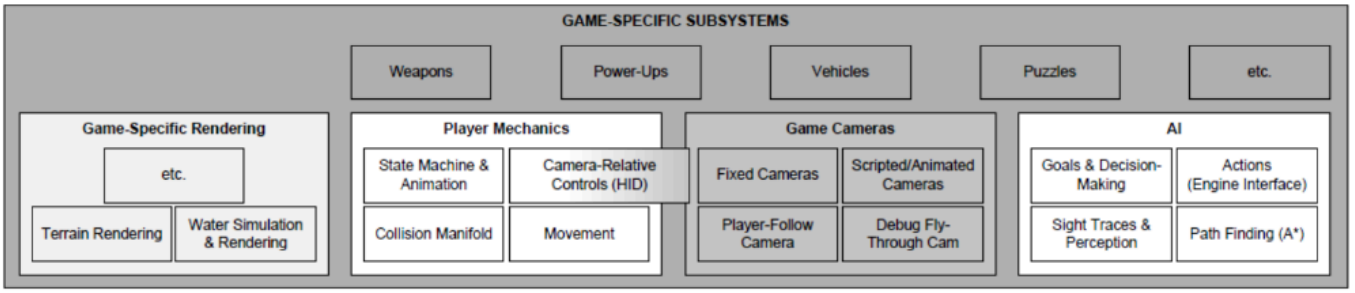
Engine, framework, middleware

- Engine
  - Comprehensive suite for game development with tools for graphics, physics, audio, scripting, etc.
  - Examples: Unity, Unreal Engine.
  - Používají se z více perspektiv (Engineer, artist, designer…); (framework a i middleware často oriented na programátory)
  - dříve engine byla ta hra (= neměl jsi engine)
  - game specific (pacman) vs game agnostic (Unity)
- Framework
  - Collection of libraries for core game development tasks, requiring more custom development.
  - “zjednodušený engine s méně features”; nemá integrované vývojářské prostředí
  - Základ pro hry nebo custom engine; nemusí být multiplatform…
  - Examples: MonoGame, LibGDX.
- Middleware
  - Specialized software/library for specific game development tasks, designed to be integrated into engines or frameworks.
  - žádný integrated enviroment
  - “one to a few subsystems”; “library on steroids”
  - Examples: Havok (physics), FMOD (audio).

Engine architecture

Game-specific subsystems

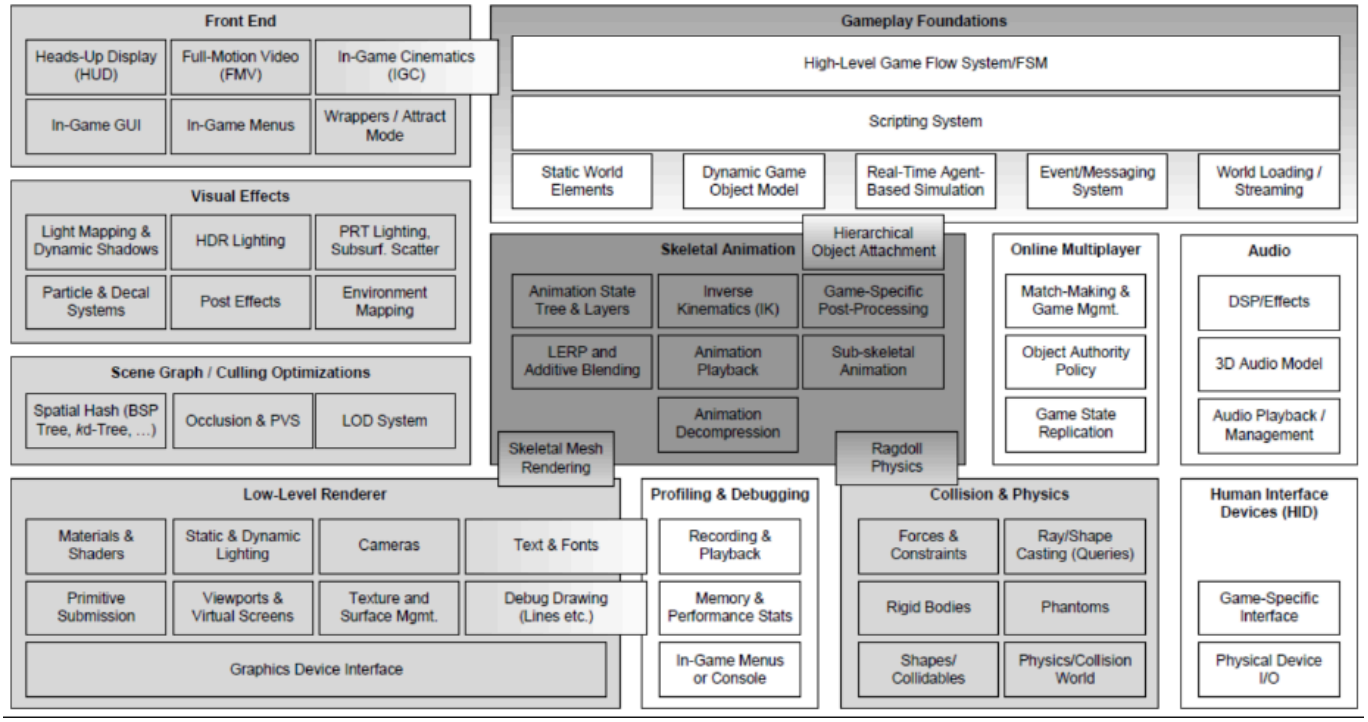
= věci specifické pro tvou hru
- herní AI, mechaniky, game-specific rendering (stylized voda…), …
- vlastně ohýbáš/rozšiřuješ engine pro svoje potřeby



Game Agnostic Layer

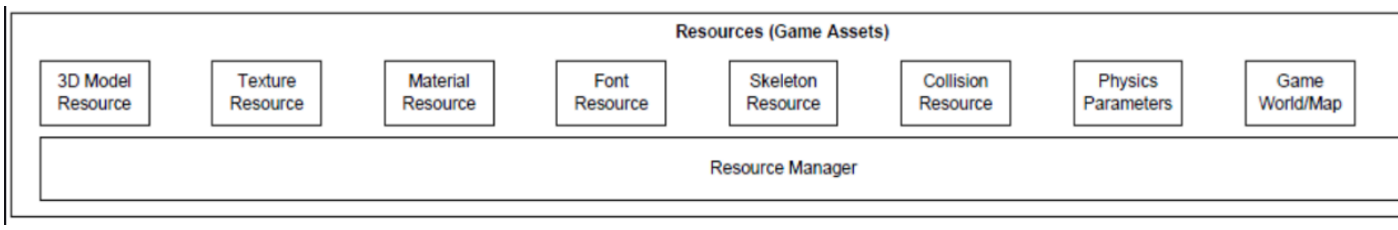
= same for almost all games

- **Gameplay foundations**
    - high level game flow system
    - scripting system
- scene graph
    - hierarchie v Unity… engine chytře využije
- front end/UI
- VFX
    - subsurface scattering,...
- (skeletal) animation
- multiplayer
- audio
- collision and physics
- low level rendering
    - multiplatform = must support many rendering APIs (DirectX, Vulcan, …)
    - jiné komprese textur, audia, … = v Unrealu “coocking”
- profiling and debugging
- Input/Output
- <u>Event system</u>
    - nechceš reference hell

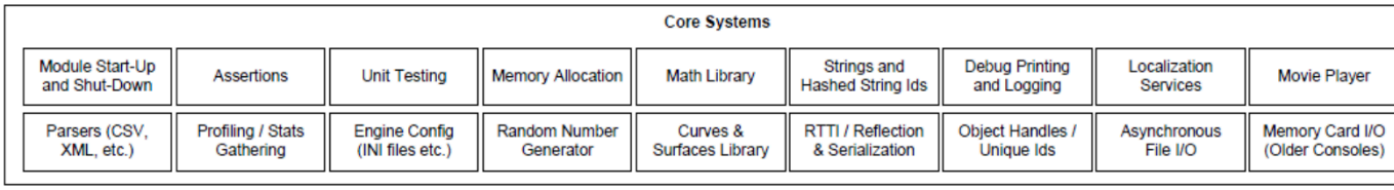


Foundations

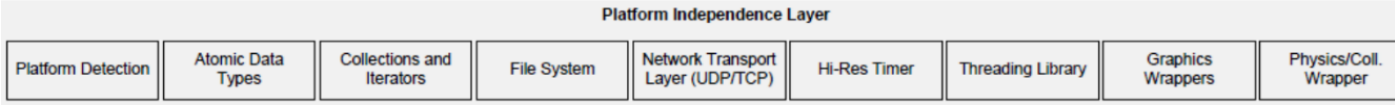
- game assets/resources
    - models, textures, materials, fonts, skeletons,
      <u>collision resources</u> (zjednodušené kolizní modely pro highly detailed charaktery = “collision wrappers”),
      physics materials…



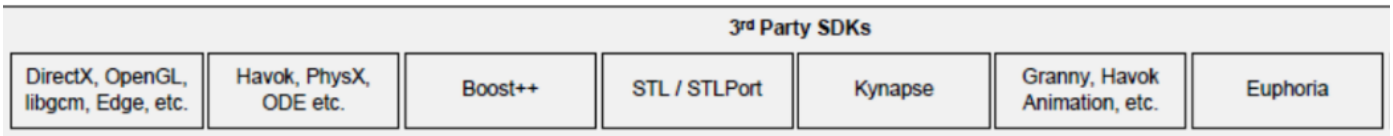
- Core systems (často platform specific logic)
  - math library (může být jiná na různých cpus…)
  - memory allocation
  - file loading/saving



- Platform independence layer
  - "interfacing with OS itself"
  - file system, threading, network transport layer…



- 3d party SDKs
  - rendering (DirectX)
  - physics (Havok, PhysX)



Pod tím máš pak OS, Drivery a nakonec Hardware.

Game Engine je v podstatě operační systém
- vlastní "jazyk", "memory management", "threading/synchronizace"...

Asset pipeline

- Resource database (TXT + SCM/SQL)
  - keeps track, co máš za assety ve hře a jejich dependencies
  - vztahy:
    model má 3 textury, ty jsou based na 3 materiálech, ty mají 5 variant podle síly PC
  - Chceš support pro version control (SCM); aby to rozumělo verzím
- Development data vs Cooked data
  - unoptimized v editoru vs final verze v buildu pro final platform

Resource management

= jak assety načítáš, procesuješ, upravuješ, mažeš (nejen) za runtime

Zodpovědnosti resource managera

- Engine musí zajistit, aby se každý asset do paměti načetl jen 1x.
    - auto load/unload podle potřeby
- Spravuje paměť
- streaming (= nahrávání dat asynchronně v pozadí)

Loading assetu

- Dictionary ID -> Resource
- je to pomalý
    - přednačteš vše na začátku
    - asynchronně v pozadí (streaming)

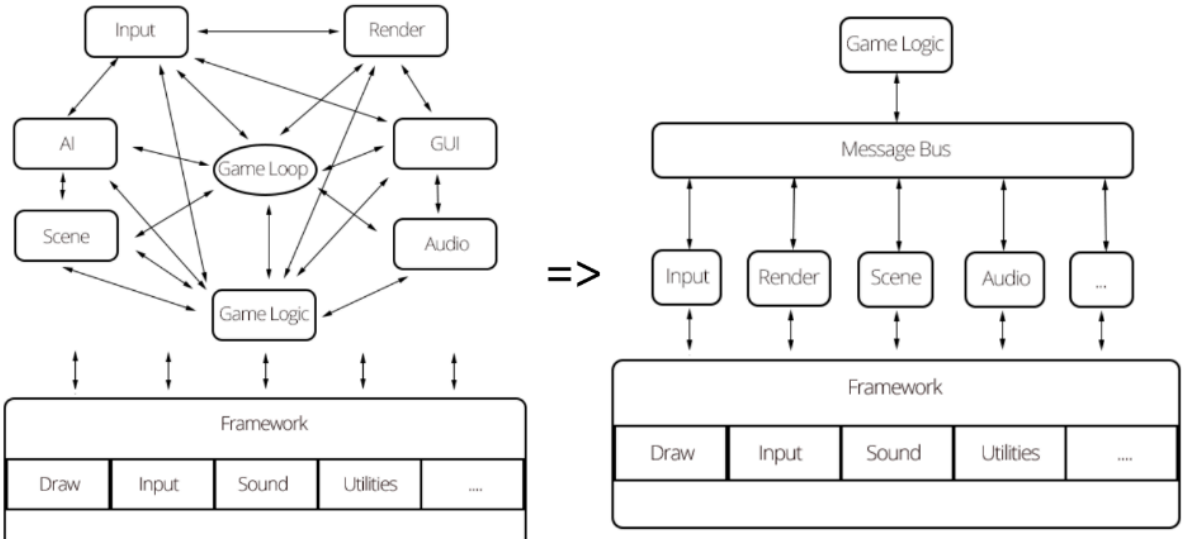
Core layers

- assertions
    - jsou v development stavu hry
      (hra má více stavů/verzí - v produkci už tyhle checky nechceš)
- Math
    - Optimalizované per platform
    - Abstrakce nad koncepty (Quaternions)
- Memory management
    - standard new/delete jsou slow - sys call = paměť ti dává OS (= čekáš)
    - (de)alokace nejsou instant

Rendering Engine

- GPU interface (DirectX, Vulcan…)
- Shader code abstraction - napíšeš 1 shader a on se pak přeloží do toho, co potřebuješ (hlsl vs glsl = Vulcan vs DirectX)
- Low level render
    - Primitives, materials, dynamic lightning, Camera, Viewport
- Scene graph, culling
- VFX
    - particles, decals, light mapping, HDR, …
- Front end/UI/HUD

Event system



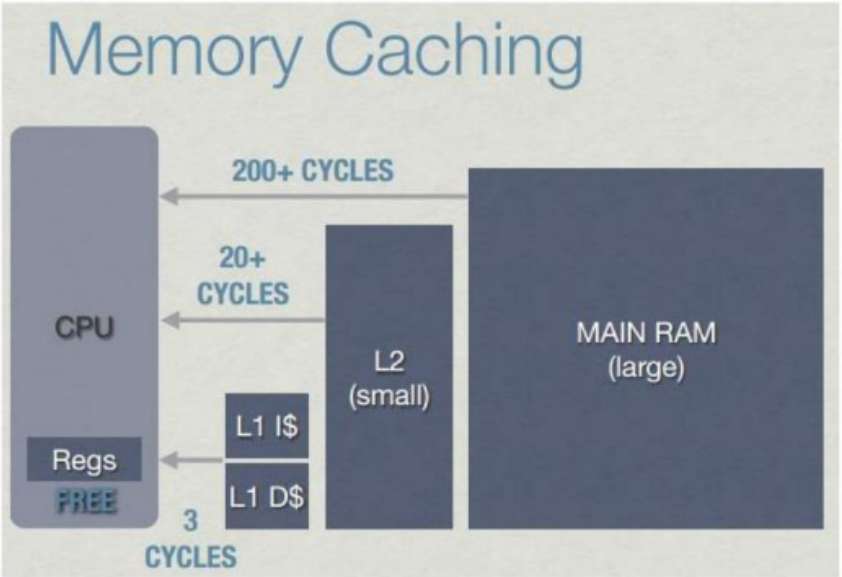
Optimalizace

Změna paradigmat

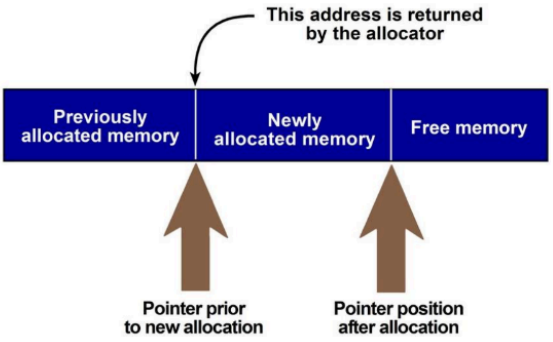
- dříve sequential - teď parallel
    - audio, gameplay, rendering každé na separate jádru
    - musíš to nějak synchronizovat (= pain)
- dříve CPU drahé, memory cheap - teď CPU levné, Memory drahá
- Musíš dělat profiling (odhady dnes nefungují)
- chceš vytížit CPU co nejvíc (jádra, pipeline instrukcí…)
- code dependencies, branching, cache misses
- precomputed tables, předpočítej vše co můžeš

Memory management

- CPU má multi lvl cache



- malloc a new jsou pomalé
    - je tam context switch do kernel mode OS
    - nejspíš se úplně vyhnout dynamické alokaci
- CHCEŠ continuous memory access
    - bacha na cache misses
    - zarovnávej dobře data
- engine má často vlastní <u>Allocator</u>
    - na začátku si řekne o obří kus paměti a pak si to managuje
- stack based memory management
    - basic verze
        - Stack-based memory management
            - Allocation simple
            - Freeing not arbitrary
            - Never fragments
            - Good for corridor games



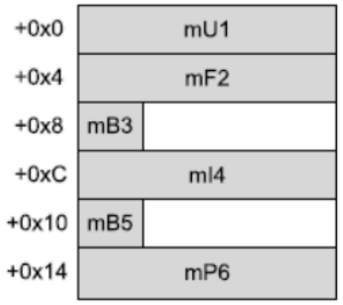
    - double sided
        - Double-side Stack-based memory management

            - Lower part -> Levels
            - Upper side -> Temporary data freed every frame
            - You must have well defined memory budget!
        - můžeš lépe mazat I guess?
    - Temporary per frame allocator
        - na začátku framu alokuješ, na konci zahodíš
        - můžeš mít i poslední 2 framy a předávat info v čase
    - Pool alocator (by data type)
        - máš předem pole nějakých instancí
    - Smart pointers
        - hlídá dealokaci sám
        - má index do memory bloku (easy defragmentace; rozlož ji mezi více frames)
    - dobré knihovny ti dovolí vlastní memory manager
    - aligning data structures

- vše musí být aligned na nějaké násobky = order matters

- The Problem

```
struct InefficientPacking
{
  U32 mU1;    // 32 bits
  F32 mF2;    // 32 bits
  U8 mB3;     // 8 bits
  I32 mI4;    // 32 bits
  bool mB5;   // 8 bits
  char* mP6;  // 32 bits
};
```



Compiler will not optimize this!

Because of efficient CPU Read/Write (CPU works only with properly aligned data).

String Handling

na přednášce to bylo - ale snad not needed

Main Loop

```
do      { play the game }
while (ExitNotRequested())
```

- chceš věci trochu posouvat, abys udělal iluzi pohybu
- máš minimálně render loop
- běžně máš více “game loops” které běží s různou frekvencí (třeba už jen Update a FixedUpdate v Unity)

Single threaded loops

- na slidech pong loop (prostě while true hraj pong)
- engine loop (abstrakce nad subsystems)

```
void main() // ENGINE
{
  initEngineAndLoadData();
  while (true) // game loop
  {
    inputSystem.Update ();
    if (isExit()) break; // exit the game
    physics.tick();
    gameplay.tick();
    rendering.tick();
  }
  destroyEngine();
}
```

- Windows message pump (pro win potřebuješ loop co něco dělá pro každé okno); “moje okno pořád funguje” (když ne, windows nahlásí stopped responding)

```
while (true)
{
  // Service any and all pending Windows messages.
  MSG msg;
  while (PeekMessage(&msg, NULL, 0, 0) > 0)
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  // No more Windows messages to process -- run one
  // iteration of our "real" game loop.
  RunOneIterationOfGameLoop();
}
```

- Můžeš mít různé loop architectures
  - Callback driven loop

```
while (true)
{
  for (each frameListener)
  {
    frameListener.frameStarted();
  }
  renderCurrentScene();
  for (each frameListener)
  {
    frameListener.frameEnded();
  }
  finalizeSceneAndSwapBuffers();
}
```

  - event driven loop
    - popular
    - vše po event busu
    - callbacky napodobíš peroidickými eventy, na konci eventu naplánuješ další
    - necheckuješ ve smyčce pro input, ale input ti vyvolá event

Multi threaded loops

- Fork Join loop
  - rozdělíš task na více procesů, čekáš, spojíš to
- thread per subsystem
  - animation thread, rendering thread…
  - Unity
- job model
  - joby můžeš dělat na všem co máš, i main thread když se nudí je může dělat
  - Unreal
- job
  - nesmí dělat race condition
  - jde vykonat v isolaci; málo data sharing a závislosti na jiných jobs
  - sám po sobě uklidí

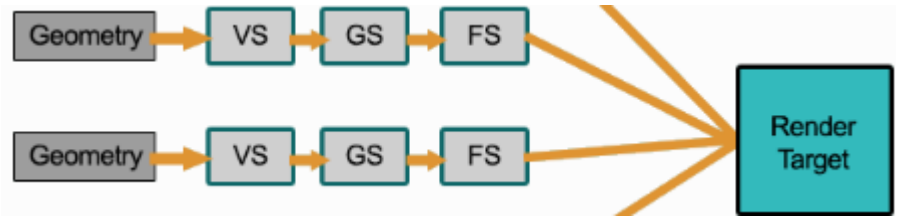
- False Sharing = můžou ti jádra zápasit o cache (to tě zpomalí)

5 Screen-space shading

Deferred shading: idea, pros and cons, comparison with forward rendering.
Ambient occlusion: definition, importance for rendering, general MC algorithm for computing AO, real-time screen-space AO.

Forward rendering

Every piece of geometry takes part in all stages of the pipeline including lighting.



<u>However</u> we need to iterate over every light in FS or render multiple times and blend.

Pros

+ easy to implement
+ simple transparency

Cons

- can be costly O(#lights * #fragments)
- some lightning computations are overwritten by closer geometry

Deferred rendering

- Decouples geometry from lighting
- Render scene once into G-Buffers (albedo, normal, specular, depth, irradiance, stíny, ambient occlusion…)
- Screen size pass to resolve lighting
    - využíváš věci stored v G-Buffers
    - world pos pixelu můžeš vypočítat díky depth bufferu reverzní transformací

Pros

+ better performance (lighting is costly)

Cons

- Memory bandwidth
- Transparency
- Antialiasing

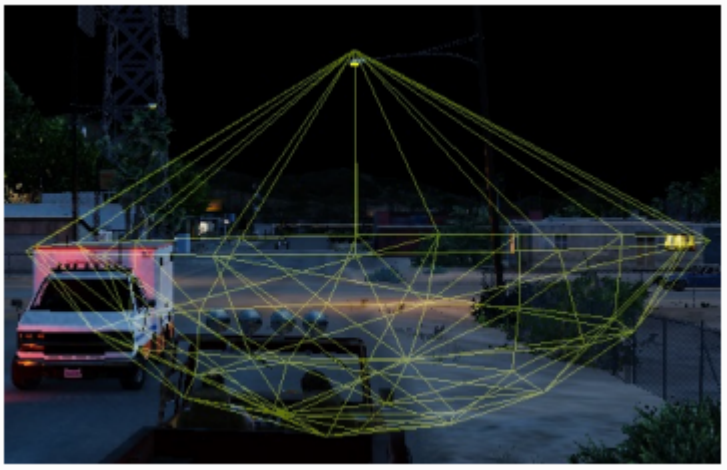
GTA V notes

- environment cube map
- uděláš g-buffers
- diffuse map
    - discarduješ něco - máš šachovnicový pattern (dithering)
- 4 shadow mapy, každá pro kus view frustrum
    - díky tomu různá kvalita stínů podle vzdálenosti
- screen space ambient occlusion
- kombinace G buffers

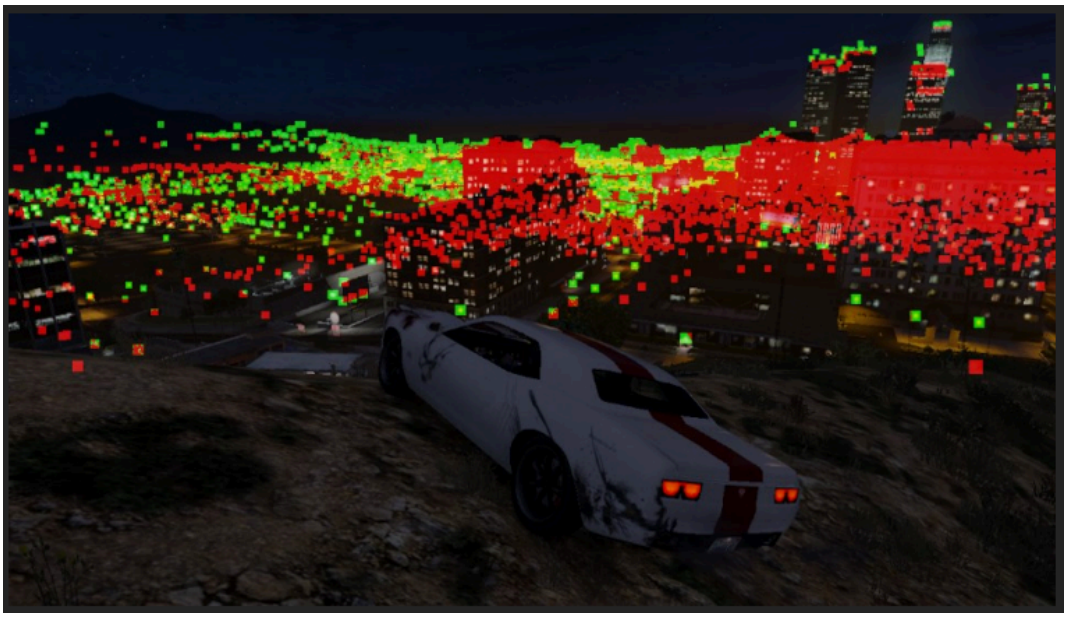
- subsurface scattering
- voda, atmosféra
- transparent pass
- odstranit dithering
- post processing (HDR tonemapping, bloom, lens distortion, chromatic aberration…)

Lights rendering

- hned po zkombinování G Buffers
- mesh kam dosvítí každé světlo; shaduješ jen pixely, kterých se to dotýká



- early z test (redukce overdraw)
- distant lights
    - instancuješ sprites + depth test



Water rendering

- vyrenderuješ scénu vzhůru nohama
- lower resolution rovina - reflaction map
- refraction map pro hladinu
- aplikuješ průhlednost vody

- kombinuješ

Ambient Occlusion

Goal: fake absence of light in occluded spaces.

If there is a lot of geometry around an object most of the ambient light will not reach it. Ideally you would cast multiple rays into the hemisphere given by the normal and use Monte Carlo to estimate how occluded the area is. This can be then used to modulate intensity of ambient light at each point.

The prominent technique is **Screen Space Ambient Occlusion (SSAO)**
It uses depth buffer data to estimate the occlusion by comparing depth of a pixel with its neighbors - if a pixel has a bigger depth than its neighbors it is likely more occluded. It trades performance for accuracy.

You can also precompute AO for static objects and store the data (in a texture)

Common Practices
Not sure jestli je to real - tohle jsem vyplodil s Chat GPT (ale snad ano, min vertex ano)

- Per-Vertex AO: Suitable for real-time applications where performance is critical, and the detail level of the AO is acceptable.
- Texture-Based AO (AO Maps): Commonly used in pre-rendered graphics, offline rendering, and in real-time applications where high detail is needed, such as in AAA games.
- (Voxel-Based AO: Used in scenarios requiring dynamic updates to AO, such as destructible environments or highly dynamic scenes.)

6 Procedural content creation for games

Top-down and bottom-up approach to procedural modeling.
Procedural terrain modeling: terrain representation, use of Perlin noise for terrain modeling.
Texturing terrains, tri-planar texture mapping.
Procedural vegetation: L-systems and their use for plant modeling.
Limitations of procedural modeling.

= We have some rules describing the desired object
= Repeatable randomness
= Saves work and memory

Two basic approaches:
- Top-down - qualitative description
- Bottom-up - simulation of natural processes

Common Techniques:
- Noise and fractals
- Rewriting system
- Voronoi diagrams
- Cellular automata
- genetic algs
- Discrete simulation
- Virtual agents
- …

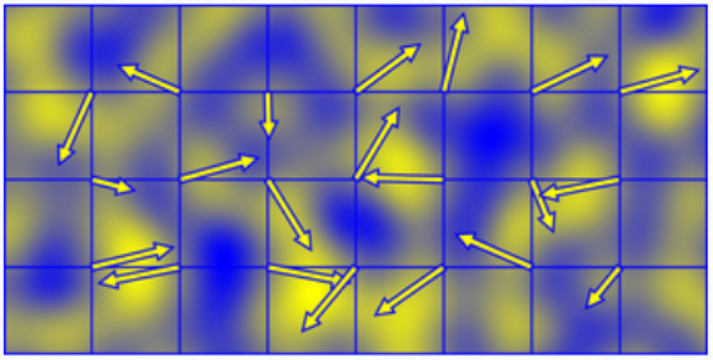
Applications:
- terrain and vegetation,
- roads,
- buildings,
- weapons,
- natural materials and phenomena - marble, wood, fire, decay

Terrain

- Heightmaps
    - array of elevation
    - sample step is important (0.5m/px is available)

- Perlin noise
    - persistent (repeatable), pseudo random
    - creation

        - na gridu si uděláš control points a z každého náhodný vektor



        - potom pro každý bod najdeš nejbližší body a interpoluješ je
    - Fractional brownian motion
      = dáš do sebe více frekvencí
        - $$h = \sum_{i=0}^{k} w_i \cdot f(\mathbf{v} \cdot n_i),$$
        - where
          - f is a noise function (e.g. Perlin noise)
          - w_i controls amplitude
          - n_i controls frequency
          - k is the number of "octaves"
          - $\mathbf{v}$ is the sample point

- texturování terénu
    - artist assignuje UVs (ok)
    - XY to UV projekce nefunguje moc dobře - na šikmých plochách se ti stretchne textura
    - Tri planar texturing
        - sampluješ texturu v XY, XZ a YZ projekcích
        - blenduješ na základě normál
        - stejné pro všechny normály
        - musíš per fragment 3x číst texturu

- voronoi diagramy
    - máš body, polchu rozdělíš na části podle toho, ke kterému bodu jsou nejblíže
    - noise based terén je nudnej (pořád stejnej, opakuje se)
    - ozvláštníš to tím, že to přenásobíš Voronoiem

$$h = c_1 \cdot d_1 + c_2 \cdot d_2 + c_3 \cdot d_3 + \ldots + c_n \cdot d_n,$$

where

- h denotes height of a point in the Voronoi cell
- d_i is distance to the i-th nearest seed
- c_i weight coefficient, e.g., $c_1 = -1$, $c_2 = 1$, rest 0

    -
- generování (ostrovů)
    - každý cell má výšku a biom, může být jezero
    - řeky podel hran
    - hodně noisu

- metody generování terénu
    - example based methods
        - máš v DB kusy terénu (heightmaps)
        - input: sketch terrain features
        - algoritmus pak poskládá kusy z databáze, aby vyhověl feature sketchi
    - sketch based methods
        - máš rough sketch; interpoluješ podle něj
        - direct noise editing
        - …
    - modelování řek
    - eroze a fyzikální simulace
    - fraktály

Vegetace

- L systémy
    - dobře zachycují podobnost kytek
    - jako gramatika (přepisování), jen chceš přepsat (=aplikovat pravidel) co nejvíc v 1 kroku
    - zastavíš po fixed počtu iterací
    - nakreslýš tím třeba to kapradí, stromy…
    - příklad

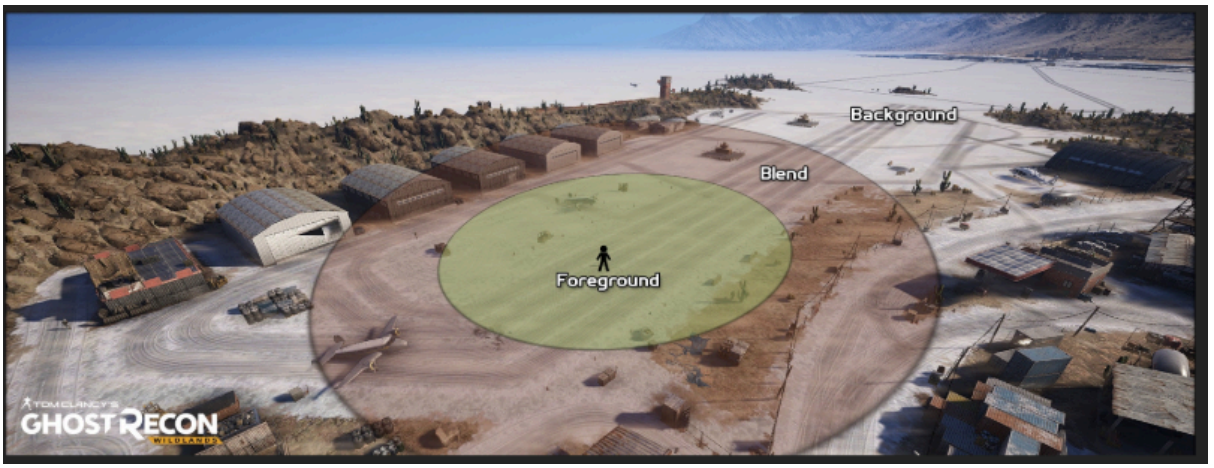
Non-terminals: F X
Terminals: + - [ ]
Axiom: X, angle = 25°
Production rules:
(X → F - [[X] + X] + F [+ F X] - X), (F → FF)

- F — draw forward
- X — no drawing, just branching
-], [— restore/store drawing state
- -, + — turn left/right by 25°

- populating the landscape
    - určíš biom na základě výšký, sklonu, vlhkosti, …
    - můžeš brát real world data; kombinovat to s noisem a biom rules
    - můžeš pustit simulaci (unpredictible)
    - artisti nejraději nástroje jako density brush
    - nebo placeneš na grid a dáš random offset

V praxi (Ghost Recon)

- na budovy shape grammars
- settlements
    - simuluješ historické události
    - connectuješ seeds
    - L-systémy
    - …
- overview
    - obří terén, 50cm resolutin + teselace, základ v reálném digital elevation modelu (DEM)
    - cesty, řeky, železnice, …
    - placement rules pro vegetaci, elektriku—
    - settlements a budovy
- v okolí hráče používáš nice 3D textury; pak je blend oblast, pak backround textury



- vegetace
    - máš podle blízkostí různé lvls of detail, jdeš až na simple 2D ish stromy
    - 3d model
    - proxy imposter
    - imposter
- cesty
    - konfiguruješ:
        - toleranci pro zatáčky
        - cenu existujících cest (jak moc stavět nové)
    - nejprve raw path
    - pak teraformující pásy
    - nakonec obarvíš

Nedostatky PCG obsahu

- limited controll
- moc random (trial and error)
- moc generic (nuda, opakuje se)
- neintuitivní ovládání/parametry
- top down metody (z popisu tvoříš) nemájí semantiku, často divné výsledky

Other notes

- inverse procedural modeling
  = snažíš se přijít na pravidla z hromady real letadel

7 Animation

Formats and compression.
Bone (or skeletal) animation; rigid bodies, joints.
Vertex blending and its problems.
Forward and inverse kinematics, solvers.
Facial animations; blend shapes, morph targets.
Motion capture.
Physically based animations; hair animation.

Formats and compression

- ve 2D máš sprites a nastavuješ podle stavu (animace) a času
    - limituje tě image size, memory size, čas
- ve 3D??

1. pamatuj si všechny vertex data pro každý frame
    a. insane drahý
    b. měl bys nějak kompresovat
    c. ok pro cutscenes
2. ulož si jen key frames a interpoluj
    a. starší hry (Quake 2…)
    b. pořád dost dat - kompresuj
    c. musíš vytvořit všechny animation poses 🙁

Quake 2 .md2 format

```
struct md2_frame_t {           // Single animation frame
  vec3_t scale;                // Scale factor for data decompression
  vec3_t translate;            // Translation for data decompression
  char name[16];               // Frame name
  struct md2_vertex_t *verts;  // List of vertices (num_vertices)
};
```

pak prostě interpoluješ mezi framy

```
// Drawing for each triangle
pframe1 = &mdl->frames[current];      // current anim. frame
pframe2 = &mdl->frames[current + 1]; // next anim. frame

pvert1 = &pframe1->verts[mdl->triangles[i].vertex[j]];
pvert2 = &pframe2->verts[mdl->triangles[i].vertex[j]];

// Decompress i-th vertex, pvert1->v is unsigned char
v_curr[i] = (pframe1->scale[i] * pvert1->v[i]) + pframe1->translate[i];
v_next[i] = (pframe2->scale[i] * pvert2->v[i]) + pframe2->translate[i];

// Interpolate between frames, interp is in range [0, 1]
v[i] = v_curr[i] + interp * (v_next[i] - v_curr[i]);
```

Komprese z Rise: Son of Rome

- kompresovali cutscenes aby to dostali z 50MB/s do 10MB/s

1. Quantizace (lossy step) - obětuješ přesnost (např. pozice budou jen z uint16)
2. Predikce - využiješ nějaký consistence
    a. kusy se vůbec nemění apod
    b. podobné jako MPEG (na video)
        i. I-frames = na intervalech storujеš celé frames
        ii. B-frames = ukládáš jen změny od předchozího framu
3. Entropy coding
   = prostě to pomocí entropy nějak zmenšit ještě
    a. kratší kódování pro časté znaky, delší pro vzácné…

Bone animation

= skeletal animation/skinning and rigging

Proč?

- dneska mají postavy uplně klidně 100k vrcholů
  = neni to real pro real time 😀
- frame to frame interpolace není gpu friendly
- Blender apod. stejně mají bones

Co?

- máš animovaný skeleton
- model je “skin” nad ním
- rigging = přiřadíš vrcholům kosti

Bones

- definuješ refference frame a pak transformaci
- origin má v jointu/kloubu
- transformuje k sobě přiřazené vrcholy (v model space)
- můžeš mít constraints pro transformaci (eg. loket)

Vertex blending

- vrchol ti transformuje více kostí
- každý bone-vertex pár má váhu
- blenduješ ty transforms ve vertex shaderu
- linear blending je fast
- formálně

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

    - w_i jsou váhy, nezáporne, sečtou se na 1
    - M_i je původní bone to model space transform
    - B_i(t) je bone to model space transform v čase t

- často konkatenace matic pro local bone transformace (podle toho jak pod sebou máš kosti)
- v praxi matice spojíš než to pošleš do VS
- animační systém musí hlídat bone constraints a každý frame updatovat B_i
- skinning matice pošleš do GPU jako constant/structured buffer
- každý vrchol má bone indices a váhy

Problems

- může to působit tuze
- visual artifacts
  - protínající se geometrie
  - collapsing volumes = např loket se ti zůží

  - texture stretching

Co s tím?

- joint rotation constraints
- víc vrcholů
- non linear blending (dual quaternions)
- přidání virtual bones

Animation pipeline, animation graph

skipuju, protože se na to otázka neptá (ale bylo na to pár slidů)

Forward and inverse kinematics

Forward

- postupně řetězíš transformace jednotlivých kostí
- snadno vypočítáš
- těžké dostat konkrétní výsledek
- v praxi musí animátor poses nastavit
- formálně
  - výsledná pozice p = f(O) kde O jsou skaláry kloubů O1..On
  - funkce f záleží na délce kostí a degrees of freedom kloubů (na ně min/max)

- simple 2D case

Simple 2D Case with 2 bones and 1-DOF, end point *P*:

$$\mathrm{P}_x = L_1 \cdot cos(\theta_1) + L_2 \cdot cos(\theta_1 + \theta_2)$$

$$\mathrm{P}_y = L_1 \cdot sin(\theta_1) + L_2 \cdot sin(\theta_1 + \theta_2)$$

Inverse

- máš cíl (např. kde má končit ruka) a constraints a nějak se to vypočte
- pomůže animátorovi dostat poses
- real time adjustments (nohy a terén, ruce a zvedání předmětů, rag dolls)
- řešíš numericky (Jacobian inverse)
- formálně
  - parametry O = f na mínu první (p) pro pozici p
  - jednoduché věci jsou vyřešit analyticky
  - v praxi numericky

IK solvers

- často iterativní lineární aproximace “iterative linear approximations of differential functions”
- někdy newtonova minimalizace: “newtonian minimization of constraint satisfying functionals”
- iterative inverse Jacobian solver

$$\mathrm{J}(\theta) = \frac{\partial f}{\partial \theta}$$

$$\Delta p = \mathrm{J}(\theta) \cdot \Delta\theta$$

$$\Delta\theta = \mathrm{J}^{-1}(\theta) \cdot \Delta p$$

  - počítej dokola delta O a p = f(O) dokud error = p - p¨ není dost malý
  - těžký je ten inverz jakobiánu J (SVD rozklad, nejmenší čtverce(?), …)

Facial animations

podobné jako keyframe interpolation

- vymodeluješ/zachytíš N facial expressions (Gollum má 675)
- blenduješ mezi nima na základě vah, předpokládáš stejnou topologii
- můžeš to uložit jako rozdíl vektorů pro localozed kontrolu
- můžeš rozdělit obličej na menší části pro localized kontrolu

Formálně:

- máš n blend shapes b_k každý z p vrcholů
- b_k je vektor p prvků s přiřazenou váhou w_k

- výsledek f je prostě lineární kombinace

$$\mathbf{f} = \sum_{k=0}^{n} w_k \cdot \mathbf{b}_k$$

- můžeš mít i difference notation kde b_0 je neutral face

$$\mathbf{f} = \mathbf{b}_0 + \sum_{k=1}^{n} w_k \cdot (\mathbf{b}_k - \mathbf{b}_0)$$

Vlastnosti
- dobře vyjádříš, ale hodně práce a artefakty
- musíš nějak získat blend shapes (sculpt, mocap)
- musíš iterovat než máš good shapes
- interpolace nemusí fachat - corrective shapes
- chceš ideálně moct retargetovat na podobné obličeje
- potřebuješ kompresi (1000 blend shapes s 10k vrcholy = 120MB)
    - delta notace a reordering = sparse matrices

Motion capture

- prostě nahraješ live performance
- nepotřebuješ animovat
- jen fyzicky možné věci by default
- facial expressions
- potřebuješ drahý gear

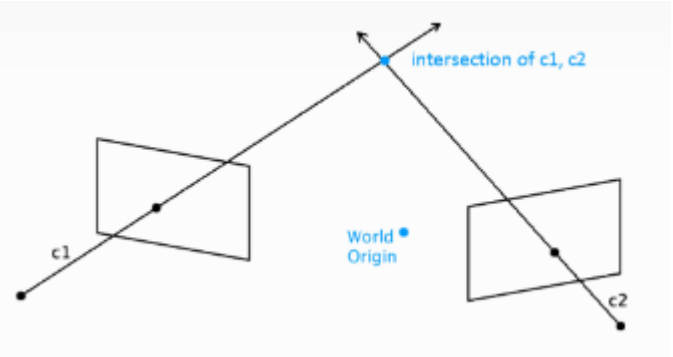
Metody
- optické systémy
    - velký space a setup, klidně 48 kamer
    - důležité světlo, kalibrace kamer, zakrývání
- ne-optické systémy
    - stačí méně místa
    - žádné zakrývání
    - méně přesné
- Magnetické systémy
    - žádné zakrývání
    - EM rušení

Druhy
- Marker-based systémy
    - máš markery na daných místech
    - svítíš na ně a trackuješ kamerou
    - aktivní
        - dráhe napájené obleky

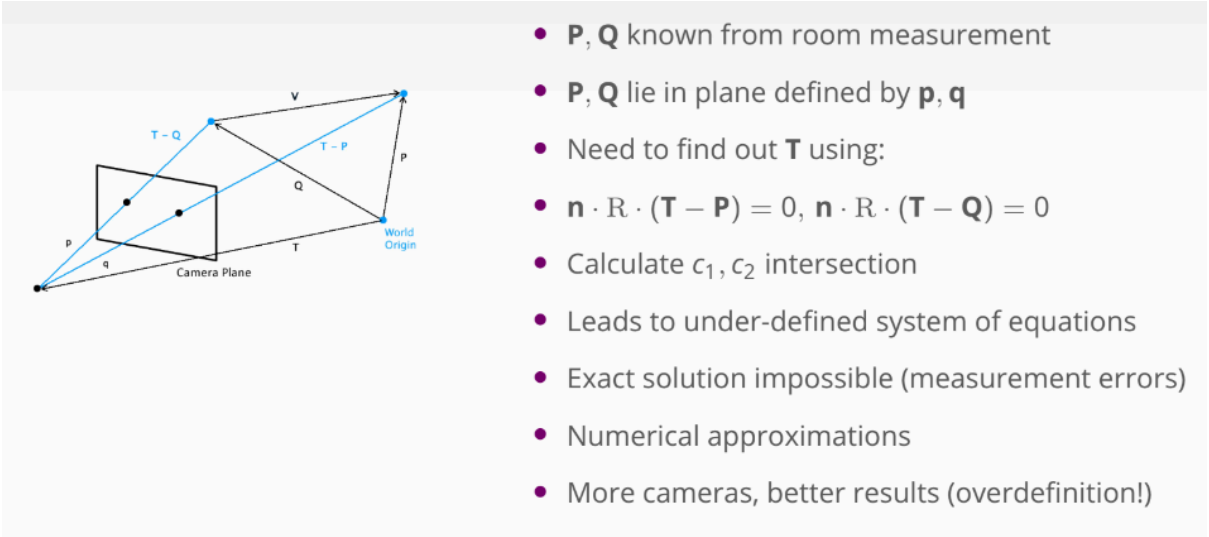
        - mega přesný
    - pasivní
        - reflective balls
        - může dojít k marker swappingu
- Marker-less systémy
    - třeba kinect
    - computer vision (optical flow, DL)
    - sensitive na camera setup
    - + na face capturing
    - často nepřesné

Matika

- nakalibruj kamery
- víš room dimensions
- víc kamer, abys trianguloval world point
- pro každou kameru: rotace a translace



- dont rly get this, prostě to nějak vypočteš



Physically based animations

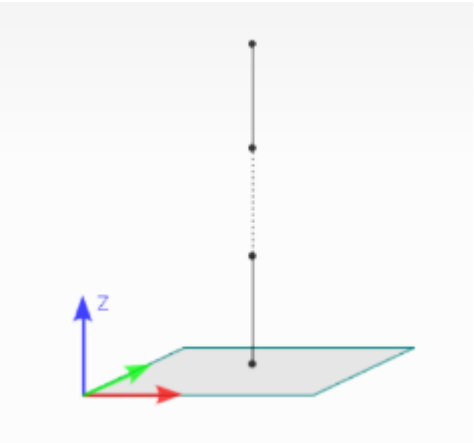
- rigid/soft body dynamics, springs
- particles, fluids, hair
- lze na CPU i GPU, často drahé
- NVidia PhysX, AMD TressFX, Havok…

Vlasy

- pozici a hustotu můžeš kontrolovat texturou

Animace vlasů

- vlas je vlastně tohle



- každá manipulace je IK (inverse kinematics) problém
  - musíš
    - zachovat délky segmentů
    - vyřešit kolize
    - podobné jako bone IK ale méně constraints

Solver

- máš zakódované inextensibility a collision constraints, iterativně po gradientu minimalizuješ, pak vyřešíš nějakou rovnici a máš correction vectory; dokola dokud ||C|| není dost malé - cca 5 iterací

- Hair constraints:
  - inextensibility: $c_i^e = \mathbf{e}_i \cdot \mathbf{e}_i - l^2,\ i = 0, 1, \ldots, n$
  - hair–geometry collisions: $c_j^c = (\mathbf{p}_j - \mathbf{v}_j) \cdot (\mathbf{p}_j - \mathbf{v}_j),\ j \in P, \mathbf{p},$
  - P contains all colliding vertices $\mathbf{v}_j$, $\mathbf{p}_j$ closest point on surface
- Constraint vector $\mathbf{C} = (c_1^e, \ldots, c_n^e, c_1^c, \ldots, c_k^c),\ k = |P|$
- Want to minimize $\|\mathbf{C}\|$, iteratively by going along largest gradient
- Construct $\nabla\mathbf{C}$ matrix containing $\frac{\partial c}{\partial x}, \frac{\partial c}{\partial y}, \frac{\partial c}{\partial z}$ for $c \in \mathbf{C}$
- Solve the following equation to calculate vertex corrections vector $\delta\mathbf{X}$:

$$\delta\mathbf{X} = -\nabla\mathbf{C}^T \cdot (\nabla\mathbf{C} \cdot \nabla\mathbf{C}^T)^{-1} \cdot \mathbf{C}$$

- Apply $\delta\mathbf{X}$ and iterate until $\|\mathbf{C}\|$ small enough ($\sim$ 5 iterations)

Standartní vlasy mají normalní skinning.
V Tomb Raideru 2013 použili TressFX knihovnu

- nějaké guide hairs a interpolace
- na GPU

Anatomical body models

= modely se svalama a tukem, physics driven - stahy svalů apod

- může být good na animation retargeting
    - dáš 3d scan člověka
    - fitneš ho na model
    - máš custom animovanej model
- hlavy
    - blendshapes kontroluješ stahem svalů
    - může to být fajn na automatic blendshapes

Optimalizace

- detaily jen u kamery
- méně bones
- menší frekvence animace u vzálených věcí
- ve VS animuj menší mesh, v teselaci dodělej detaily
- skining v compute shaderu, pak do bufferu a renreduj pomocí cheap VS

8. Physically-based rendering - Radiometry and Colorimetry

Physically-based rendering seeks to render photorealistic images using real optics. In order to do that, it is important to understand basics of radiometry, photometry and colorimetry.

8.1 Basic Principles of Radiometric Quantities

Radiometry is a field of science that focuses on measuring the electromagnetic radiation (**the flow of photons**, or light).

Photon

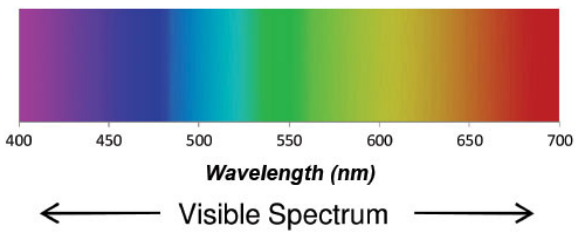
Photon is the smallest quantum of light that can be emitted from a light source. It can be described by its **wavelength** λ. It can also be described by its **frequency**: $\nu = c/\lambda$ (where c is the speed of light).

The **energy** of a photon is:

$$e_\lambda = h\nu,$$

where h is Plank's constant (6.6262×10^{-34} Js).

Photons visible to the human eye (**visible spectrum**) have wavelength within the range from 380nm to 780nm.



Radiant Flux

Radiant flux Φ $[W = Js^{-1}]$ is a quantity describing the amount of energy (**sum of photons**) emitted or received during a unit of time (second).

It can be calculated as follows (bonus):

$$\Phi = \frac{dQ}{dt}$$

Irradiance and Exitance

Irradiance E_L $[Wm^{-2}]$ is the amount of radiant flux received by a unit area. That is, from all incoming directions and all incoming light sources.

It can be calculated as follows (with A being the measured area):

$$E_L = \frac{d\Phi_i}{dA}$$

Total irradiance is a sum of all respective irradiances form all light sources:

$$E_L = \sum_{i=1}^{n} E_{L_i}$$

Surfaces however not only receive light but can also reflect light; in such cases the reflected light is called **exitance** (denoted M_L $[Wm^{-2}]$).

Irradiance depends on **distance** (**inverse-square law**) and **angle** (**Lambert's law**).

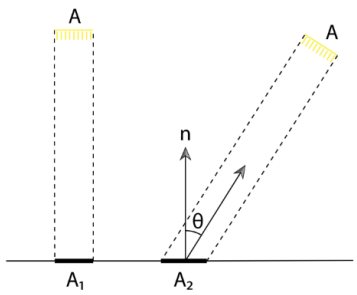
Inverse square law states that the observed "intensity" of a specified physical quantity is inversely proportional to the square of the distance from the source of that physical quantity. In the case of irradiance it is:

$$E_L = \frac{d\Phi}{dA} = \frac{\Phi}{A} = \frac{\Phi}{2\pi r^2} = \frac{\Phi}{2\pi} \cdot \frac{1}{r^2}$$

Because $\Phi/2\pi$ is constant we can write:

$$E_{L1}r_1^2 = E_{L2}r_2^2$$

Lambert's cosine law for irradiance goes as follows:



When $\Theta = 0$ the emmiting area A is the same as receiving area A_1. After tilting the surface (or the light source) we get new larger receiving area $A_2 = A_1/\cos\Theta$. After pluging this to the formula for irradiance we get:

$$E_{L2} = \frac{\Phi}{A_2} = \frac{\Phi \cdot \cos\Theta}{A_1} = E_{L1} \cdot \cos\theta$$

Radiant Intensity

Radiant intensity I $[Wsr^{-1}]$ is radiant flux emitted (reflected, transmitted or received) with respect to a solid angle:

$$I = \frac{d\Theta}{d\omega_L}$$

One way to look at radiant intensity is as a light source property that **does not change with distance**. If we imagine an ideal bulb that emits the same amount of energy in all directions, we can investigate some emitted light ray along its way. We an also cast a cone along the ray with some solid angle in steradians. Now, radiant flux Θ remains the same along the ray but the sphere area gets bigger and as such irradiance E_L gets smaller (r is radius of the considered sphere):

$$E_L(r) \propto \frac{1}{r^2}$$

Therefore, we can define radiant intensity as follows:

$$I = E_L(r)r^2$$

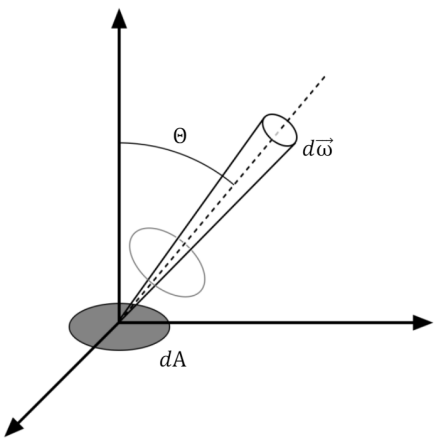
Now it's the radiant intensity that we are trying to capture by some sensor (camera sensor, a rod in the eye); however a capturing sensor also has an area.

Radiance

Radiance L $[Wm^{-2}sr^{-1}]$ measures the light power per unit projected area per unit solid angle. Formulas for the computation of radiance are:

$$L = \frac{dI}{dA \cdot \cos\Theta} = \frac{dE}{d\omega} = \frac{d^2\Phi}{d\omega \cdot dA \cdot \cos\Theta}$$

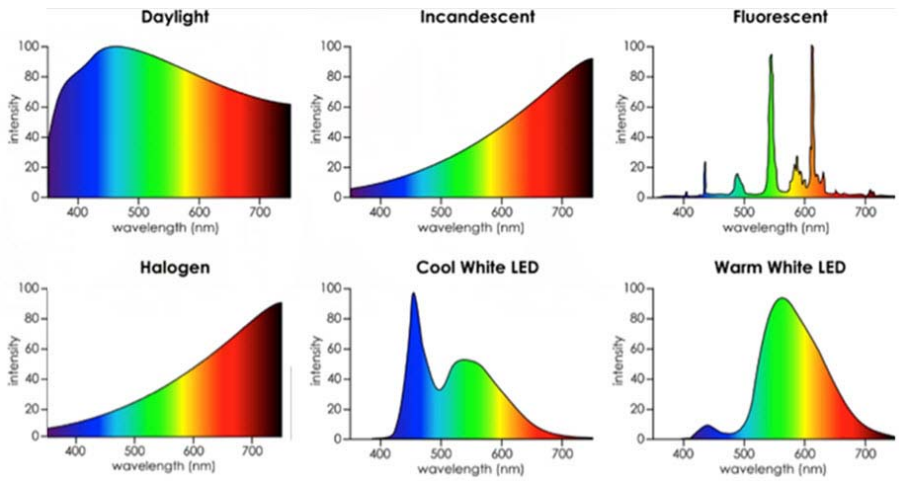
Radiance is very important quantity for the computer graphics as it **does not change with distance and rotation**. It is also proportional to perceived brightness.



Quick overview of a camera and its relation to radiance: Cameras in their very core consist of a small aperture, lens and a sensor. Aperture regulates the amount of incoming light (imaging squinting your eyes on a bright day) and lens focus the incoming light onto the sensor. This way the sensor receives only light from a small set of directions; each sensor is thus directionally specific. In rendering, we often use a "pinhole camera" model to avoid computing the influence of lens and aperture, but the overall concept stays similar.

8.2 Light Spectrum and Colors

Total radiant flux (radiant power) is the sum of all photon energies. However, different lights may have different amounts of photons at respective wavelengths. The plot of these amounts (radiant powers) is called **light spectrum (power distribution)**.



Spectral Radiance

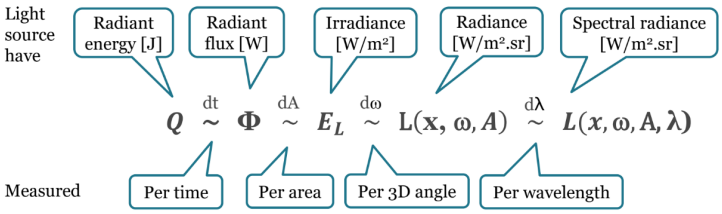
Spectral radiance (or specific intensity) is the radiance of a surface **per unit frequency or wavelength**, depending on whether the spectrum is taken as a function of frequency or of wavelength.

Spectral radiance in frequency is radiance per hertz:

$$L(x, \omega, A, \nu) = \frac{dL(x, \omega, A)}{d\nu} \quad [Wsr^{-1}m^{-2}Hz^{-1}]$$

Spectral radiance in wavelength is radiance per wavelength:

$$L(x, \omega, A, \lambda) = \frac{dL(x, \omega, A)}{d\lambda} \quad [Wsr^{-1}m^{-3}]$$



Human Eye

The main parts of the human eye that are good to know are: cornea, sclera, iris, lens, retina and optic nerve. What is important for now is that in **retina** there are two types of photoreceptors - **cones** ans **rods**.

Cones are gathered at the centre of the retina (and mainly in the fovea where there are only the cones present) and are responsible for acute and color vision. Rods are dominant in the peripheral parts of the retina and are responsible for vision in a dim light and for the wide field of vision. There are much more rods than cones.

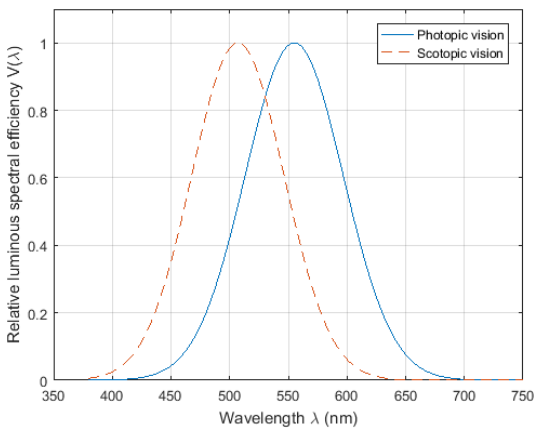
According to the luminance levels (brightness) the vision is mediated by cones (**photopic vision**), rods (**scotopic vision**), or by both at the same time (mesopic vision).

At low light, vision is mediated by more sensitive rods. There is only one type of rods (we see almost no color or we see more bluish as the rods are more sensitive to blueish colors - Purkinje effect). As the light gets brighter, the rods become saturated and the vision starts to be mediated by less sensitive cones. There are three types of cones (color vision).

Photometry

So far, we described physical quantities. What would be great however were some quantities that **take into account the human perception**. Thankfully such quantities exist, and the field of science dealing with them is called **photometry**.

In photometry, every radiometry quantity is weighted by sensitivity of the human eye to respective wavelengths, i.e., by human **luminosity function (luminous efficiency)**. It does not deal with the perception of colors, rather with the perception of brightness from light of various wavelengths.



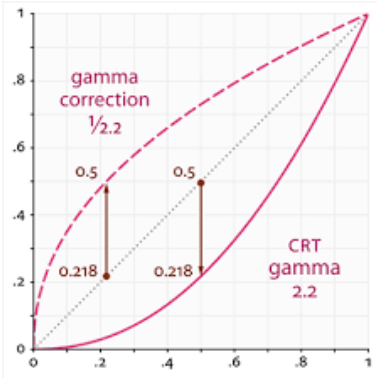
Luminosity function describes the average spectral sensitivity of human visual perception of brightness.

radiometry			photometry		
symbol	quantity	unit	symbol	quantity	unit
F_e	radiant flux	W (Watt)	F_v	luminous flux	lm (lumen)
I_e	radiant intensity	W/sr	I_v	luminous intensity	cd (candela)
E_e	irradiance	W/m^2	E_v	Illuminance	lx (lux)
L_e	radiance	$W/(sr?\cdot m^2)$	L_v	luminance	cd/m^2

Brightness and Gamma Correction

Human perception of brightness is **non-linear** to luminance - there is an exponential relationship, roughly 2.2. To alleviate this, CRTs (and LCDs) are **gamma correcting** the color signals to match human perception of brightness.

However, photon energy is linear. This means that, in a renderer, all light transport calculations must be done on linear values **before** the gamma correction is applied.



Colorimetry

Colorimetry is the science and technology used to quantify and describe physically the human color perception.

Humans have three types of cones in the retina. The cone types differ according to the their peak sensitivity to light of certain wavelength. We then have cones of type:

- S (short): 420-440 nm
- M (medium): 530-540 nm
- L (long): 560-580 nm

Based on this, CIE organized **experiment** that was to find how to form any color by combining red, green, and blue. The principle of this experiment was that a person was asked to find the intensities of red, green, and blue lights that, when combined, looked identical to the shown color spots. Participants could set the intensity of the three lights by turning three knobs in a range from -1 (light was added to the target color) to 1 (light fully on).

As the result of this experiment we got three functions describing spectral sensitivities. This is useful - when we have a known spectrum we can integrate this spectrum with the obtained three curves and get R, G, B values. With the known R, G, B values, we can then replicate any color by shining the 3 monochromatic lights with the given intensities.

Problem however is that the "red function" has **negative part**.

CIE then proposed three different hypothetical light sources that did not use monochromatic light to represent the (visible) light spectrum; and their corresponding color matching functions $\overline{x}(\lambda)$, $\overline{y}(\lambda)$ and $\overline{z}(\lambda)$ which have precise math models.

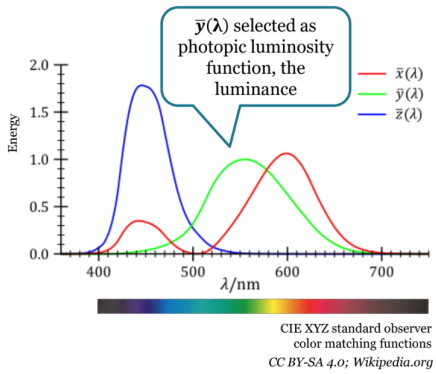
If we take a color function $C(\lambda)$, we can then compute the so-called **tristimulus values X, Y, Z**, which define the color in CIE XYZ space known as **CIE 1931 XYZ** color space.

$$\boldsymbol{X} = \int_{380}^{780} C(\lambda)\bar{x}(\lambda)\, d\lambda$$

$$\boldsymbol{Y} = \int_{380}^{780} C(\lambda)\bar{y}(\lambda)\, d\lambda$$

$$\boldsymbol{Z} = \int_{380}^{780} C(\lambda)\bar{z}(\lambda)\, d\lambda$$

Different $C(\lambda)$, i.e. different light spectra, can yield the same color! Such light spectra are called metamers.



CIE XYZ color space is 3D, we choose a plane such as X + Y + Z = 1. We define:

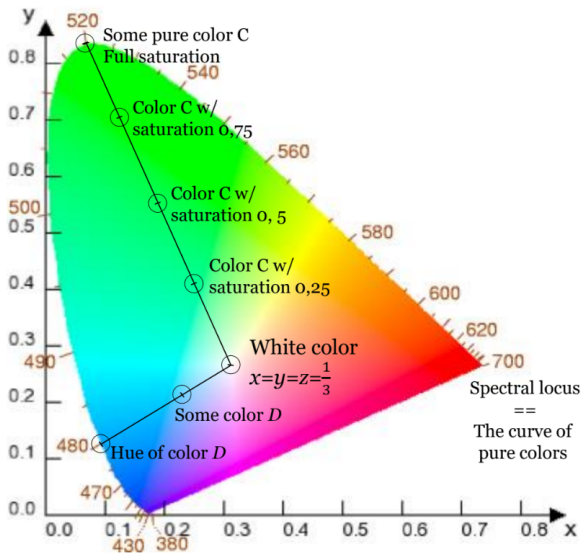
$$x = \frac{X}{X+Y+Z}; \quad y = \frac{Y}{X+Y+Z}; \quad z = \frac{Z}{X+Y+Z} = 1 - x - y$$

As a color space in 3D may be undesirable by some, next color space was obtained by dropping the z value (as it does not bear additional color value).

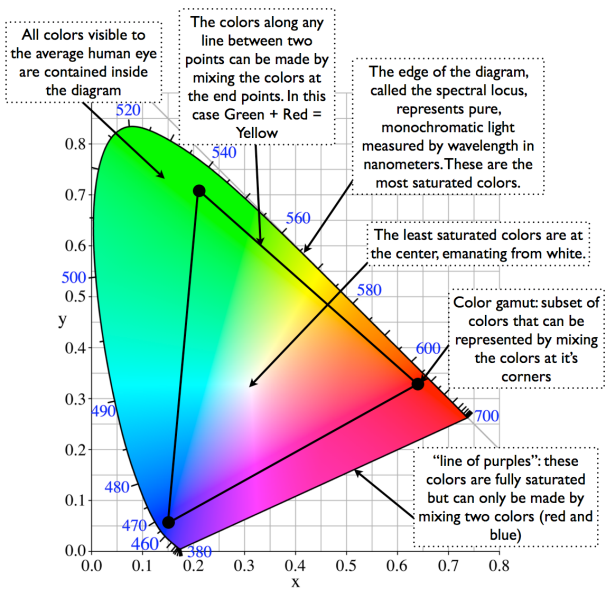
CIE xyY color space consisits od the xy plane of CIE XYZ color space and Y component - luminance. Conversion from xyY to XYZ exists:

$$X = \frac{Y}{y}x, \quad Z = \frac{Y}{y}(1 - x - y)$$

The xy plane of CIE XYZ color space also gives us **the chromaticity diagram**.The curve of this diagram is called *spactral locus* and consists of (pure) spectral colors. Diagram can be used to find **hue** and **saturation** of its colors.



Chromacity diagram; note there is no, e.g., brown color.
Public domain; Wikipedia.org



Anatomy of a CIE Chromaticity Diagram

sRGB

Color space used nowadays for monitors, printers and the internet is **sRGB**. There exists conversion between sRGB and XYZ. Color channels have 8-bit resolution.

In rendering, we represent the light via RGB values from sRGB color space; connection with CIE is connecting it with real-world and human perception.

sRGB defines a nonlinear transformation between the intensity of primary colors and the actual numbers stored (sRGB values are stored as gamma compressed). Contrary to widespread use, sRGB is not using 2.2 as gamma correction; the mapping is more complex.

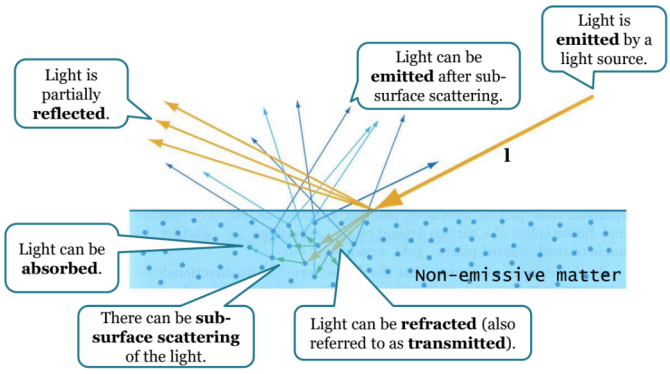
Importantly, sRGB values are in non-linear space but physically-based renderers are using linear transformations. This means that, when working with sRGB textures, you need to gamma decode values first, pass them through renderer and then gamma encode back.

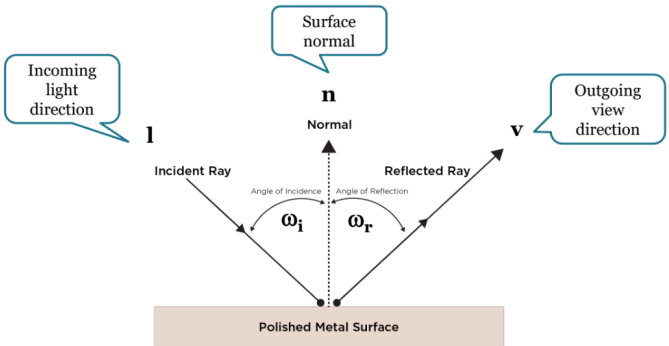
Gamut is the color triangle defined by the red, green and blue primaries. This triangle contains all the colors that we can get using the three primaries. Any other color outside this triangle is however impossible to obtain.

9. Physically-based rendering - BRDF

Shading as the process of using an equation to compute the outgoing radiance L_O along the view ray v based on material properties and light sources.

Light-matter Interactions





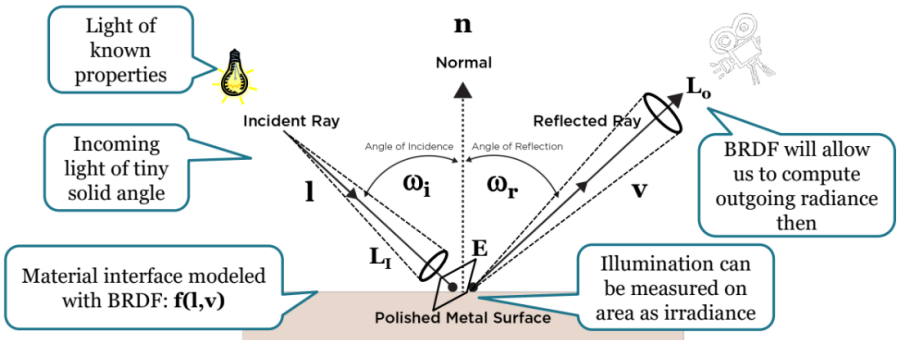
BRDF (Bidirectional Reflectance Distribution Functions)

The appearance of materials is determined by their interactions with light. These interactions can be characterized by functions known as **BxDFs (Bidirectional "x" Distribution Functions)**. In real-time rendering we are interested in **BRDF (R = reflection)** and **BTDF (T = transmission)**. Combining these two functions together we get **BSDF (S = scattering)**. BRDF is typically implemented in a fragment shader.

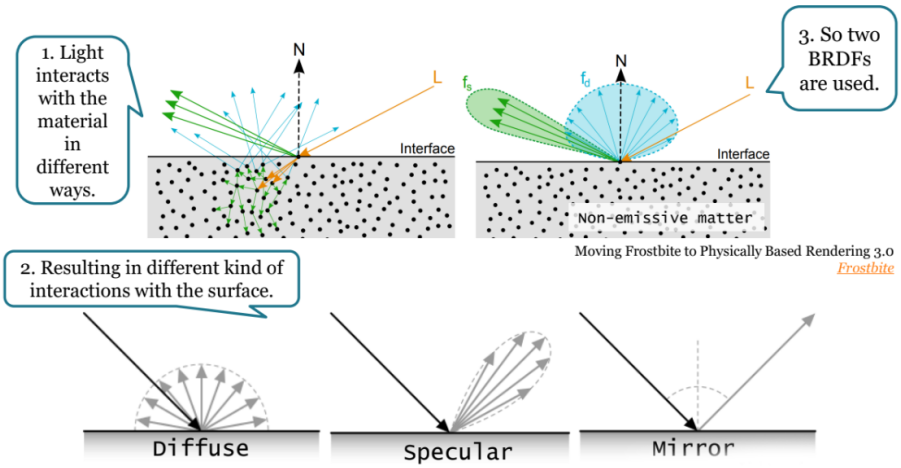
BRDF is a (black-box) function that, given incoming (l, or light) and outgoing (v, or view) directions, provides a ratio between incoming and outgoing light energies at some point on the material.

$$f(l,v) = \frac{dL_O(v)}{dE(l)} = \frac{L_O(v)}{E(l)\cdot\cos(\omega_i)} = \frac{L_O(v)}{L_I(l)\cdot\cos(\omega_i)\cdot dl} \quad [sr^{-1}]$$

All f, E, L_O are **RGB vectors** representing the spectrum, range [0;1]



There are **3 main kinds of BRDFs: diffuse, specular and mirror reflection**.



In practice, diffuse and specular components are modeled **separately**:

$$L_O(v) = L_{OD}(v) + L_{OS}(v) = f_D(l, v) \cdot E_D cos(\omega_i) + f_S(l, v) \cdot E_S cos(\omega_i)$$

Models for BRDFs are:

- Lambertian model (constant BRDF)
- Phong model
- Blinn-Phong model
- Torrance-Sparrow
- Cook-Torrance

The Rendering Equation

For purposes of shading we are the most interested in the **outgoing radiance** L_O. It can be computed as the the sum of emitted radiance L_e and reflected radiance L_r:

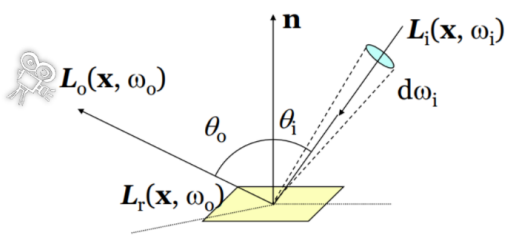
$$L_O(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_o)$$

Reflected radiance is then computed as an integral over the hemisphere (light can come from any direction) at the position on the surface x:

$$L_r(x, \omega_o) = \int_{\omega_i \in H(x)} L_I(x, \omega_i) \cdot f(x, \omega_i \to \omega_o) \cdot cos(\Theta_i) d\omega_i$$

As we can see, the equation consists of three parts: **reflected light (unknown)**, **BRDF (known)** and **consine of incident angle (known)**.

Now, in real-time (physically-based) rendering challenge is the computation of this integral.



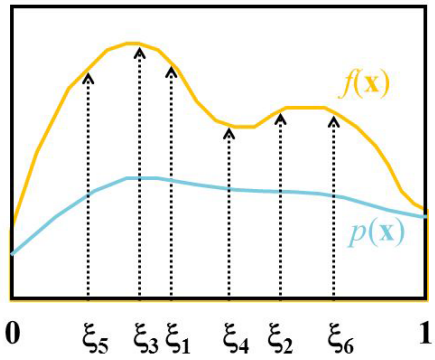
How to integrate the rendering equation: The integral is solved using **the Monte Carlo integration**, a general tool for estimating a definite integral such as:

$$I = \int g(x)dx$$

We choose a probability distribution p (e.g., uniform random distribution). Then we choose N random samples ξ_1, ... , ξ_N sampled proportionally to p. We estimate the integral I to be equal to:

$$\langle I \rangle = \frac{1}{N} \sum_{k=1}^{N} \frac{g(\xi_k)}{p(\xi_k)}$$

On average, the estimate $\langle I \rangle$ is equal to the actual integral: $E[\langle I \rangle] = I$



Another problem is that the rendering equation is **recursive** - in order to compute the reflected light we need to compute the incoming light, for which we need to again compute reflected light...

This is solved by reducing the number of light bounces in the scene - in real-time rendering we compute **only a single bounce**:

$$L_r(x, \omega_o) = \sum_{l \in L} L_{I_l}(x, \omega_{i_l}) \cdot f(x, \omega_{i_l} \rightarrow \omega_o) \cdot cos(\Theta_{i_l})$$

And as such, integral is reduced to a sum and Monte Carlo is not needed.

Basic Types of BRDF

Isotropic vs. Anisotropic Materials

BRDF of a given material may or may not depend on an azimuthal angle of the reflection. Materials where the viewing angle does affect the perceived reflection are called anisotropic. Examples of such materials are: brushed metal, fabrics and hair. This is caused by different microscopic roughness in different directions.

Dielectrics vs. Conductors

In dielectrics, a portion of incident light is transmitted; e.g., glass, mineral oil, water and air. **In conductors**, a small portion of incident light is transmitted and quickly absorbed; the rest is reflected; e.g., metals.

Bonus: One way to differentiate between a rendered plastic ball and a metal one, is that the metal one will have colorful highlight. On the other hand, the plastic one will always have white highlight.

Diffuse vs. Specular

Ideal diffuse reflection (Lambertian reflection) is when the light gets reflected to all directions with the same probability, irrespective of the direction it came from. This means that the BRDF function becomes a constant function as the angles are now redundant. For a given colour (albedo) value $\rho \in [0,1]$, the Lambertian BRDF formula is:

$$f_{BRDF} = \frac{\rho}{\pi}$$

For **mirrors** the reflectance equation is simplified as the reflected light is equal to the incoming light. However, different material reflect only a portion of light. This is determined by the **Fresnel equations**.

Fresnel Equations

For example, when viewing a surface of a table from above we observe no (depending on a varnish) reflections. As the viewing angle gets more shallow (or Θ_i gets larger), we start to observe that the surface becomes mirror-like. This phenomena is described by the **Fresnel reflectance** R_F:

$$L_r(x, v_i) = R_F(\Theta_i)L_i(x, v_r)$$

Light can also get **refracted**. This is computed using the **Snell's law**:

$$n_1 sin(\Theta_i) = n_2 sin(\Theta_r)$$

where n_1, n_2 are the **refractive indices** of the two materials (example air has $n_1 = 1$ and water $n_2 = 1.33$, the enumeration says that a ray went from air to water); Θ_i is the **angle of incidence** and Θ_r is the **angle of refraction**.

One way to represent these functions in a shader is using **Schlick's approximation**:

$$R_F(\Theta_i) = R_F(0) + (1 - R_F(0))(1 - cos(\Theta_i)^5$$

Because of this, we need only a single number to represent the Fresnel reflectance for the material.

Microfacet Model

In reality, surfaces are not completely smooth - on a microscopic level they consist of small imperfections. To solve this, we imagine the surface to consist of small "V" shaped valleys. These micro-surfaces (microfacets) then behave like ideal mirrors. We observe 3 phenomena: **masking**, **shadowing** and **refelction**.

BRDF using the microface model is as follows:

$$f = \frac{F(\Theta_i)G(\omega_i, \omega_r)D(\Theta_h)}{cos(\Theta_i)cos(\Theta_r)}$$

10. PBR – Function Representation and Spherical Harmonics

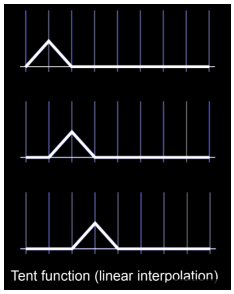
Representing a function as a linear combination of basis functions

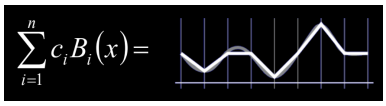
Let us assume that $G(x)$ is a function we wish to represent. Then If we have some basis functions $B_1(x), B_2(x), ...B_n(x)$, we can approximate $G(x)$ as **a linear combination** (weighting and summing) of these basis functions:

$$G(x) = \sum_{i=1}^{n} c_i B_i(x)$$

Where c_i are coefficients that we must store.

An example of some basis functions is sines and cosines (**Fourier basis**), or some shifted tent functions (linear interpolation).





Finding the coefficients

Now, we have the original function $G(x)$ and the basis functions. What we need to do is to **find the coefficients**. To do that, we utilize some error measure - in our case the L_2 error measure:

$$E_{L_2} = \int_I [G(x) - \sum_i c_i B_i(x)]^2$$

Minimizing E_{L_2} leads to:

$$\begin{bmatrix} \langle B_1 | B_1 \rangle & \langle B_1 | B_2 \rangle & \cdots & \langle B_1 | B_n \rangle \\ \langle B_2 | B_1 \rangle & \langle B_2 | B_2 \rangle & & \vdots \\ \vdots & & \ddots & \vdots \\ \langle B_n | B_1 \rangle & \langle B_n | B_2 \rangle & \cdots & \langle B_n | B_n \rangle \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle G | B_1 \rangle \\ \langle G | B_2 \rangle \\ \vdots \\ \langle G | B_n \rangle \end{bmatrix}$$

Where

$$\langle F | H \rangle = \int_I F(x) H(x) dx$$

The first matrix on the left side **B** does not depend on $G(x)$. **B** can be computed just once for a given basis.

The algorithm for finding the coefficients is then as follows:

1. Given a basis $\{B_i(x)\}$: compute matrix $\mathbf{B}$, compute its inverse $\mathbf{B}^{-1}$
2. Given a function $G(x)$: compute dot products

$$[\langle G|B_1\rangle\langle G|B_2\rangle \cdots \langle G|B_n\rangle]^T$$

3. Compute coefficients as:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \mathbf{B}^{-1} \begin{bmatrix} \langle G|B_1\rangle \\ \langle G|B_2\rangle \\ \vdots \\ \langle G|B_n\rangle \end{bmatrix}$$

Orthonormal basis functions

An **orthonormal set** is an orthogonal set with the additional condition that the inner product of any function in the set with itself is equal to 1. More formally, the condition for a set of functions $\{f_j()\}$ to be orthonormal is:

$$\langle f_i(), f_j()\rangle = \begin{cases} 0, & \text{where } i \neq j, \\ 1, & \text{where } i = j. \end{cases}$$

Note: **Inner product** is defined as:

$$\langle f_i(x), f_j(x)\rangle \equiv \int f_i(x) f_j(x) dx$$

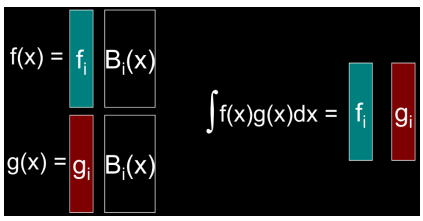
So, if basis is orthonormal, we get:

$$\mathbf{B} = \begin{bmatrix} \langle B_1|B_1\rangle & \langle B_1|B_2\rangle & \cdots & \langle B_1|B_n\rangle \\ \langle B_2|B_1\rangle & \langle B_2|B_2\rangle & & \vdots \\ \vdots & & \ddots & \vdots \\ \langle B_n|B_1\rangle & \langle B_n|B_2\rangle & \cdots & \langle B_n|B_n\rangle \end{bmatrix} = \begin{bmatrix} 1 & & & 0 \\ & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{bmatrix} = \mathbf{I}$$

Which simplifies the computation of coefficients to:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle G|B_1\rangle \\ \langle G|B_2\rangle \\ \vdots \\ \langle G|B_n\rangle \end{bmatrix}$$

Another advantage of having orthonormal basis is that the integral of product is simplified to **dot product of coefficients**:



This can be **applied to the rendering equation** - it is an integral of product $L_i(\omega_i)$ and $BRDF(\omega_i)cos(\Theta_i)$

Spherical harmonics

We need a way to represent the incoming radiance from any direction onto an object. To begin, we will consider the radiance to be a function of only the direction being integrated, not the surface position. Doing so works on the assumption that the lighting environment is infinitely far away.

Radiance arriving at a given point can be different for every incoming direction. These types of quantities can be represented by **spherical functions**, which are defined over the surface of the unit sphere, or over the space of directions in $\mathbb{R}^3$. We will denote this domain as S.

We will refer to these representations as **spherical bases**, as they are bases for vector spaces of functions defined over a sphere.

Converting a function to a given representation is called **projection**, and evaluating the value of a function from a given representation is called **reconstruction**.

Properties we might seek in a given basis are:

- Efficient encoding (projection) and decoding (lookup).
- The ability to represent arbitrary spherical functions with few coefficients and low reconstruction error.
- Rotational invariance of projection, which is the result of rotating the projection of a function is the same as rotating the function and then projecting it. This equivalence means that a function approximated with, e.g., spherical harmonics will not change when rotated.
- Ease of computing sums and products of encoded functions.
- Ease of computing spherical integrals and convolutions.

An example of a basis that can be used to approximate functions on the sphere are **spherical harmonics**.

Notation:

- Approximated function: $G(\Theta, \varphi)$
- Basis functions: $Y_i(\Theta, \varphi) = Y_{l,m}(\Theta, \varphi)$
- Indexing: $i = l(l+1) + m$

Representation of direction:

(Direction is **unit vector** in 3D.)

- Cartesian coordinates:

$$\omega = [x, y, z], \quad x^2 + y^2 + z^2 = 1$$

- Spherical coordinates:

$$\omega = [\Theta, \varphi], \quad \Theta \in [0, \pi], \quad \varphi \in [0, 2\pi]$$

$$\Theta = \arccos(z), \quad \varphi = \arctan(\frac{y}{x})$$

$$x = \sin(\Theta)\cos(\varphi), \quad y = \sin(\Theta)\sin(\varphi), \quad z = \cos(\Theta)$$

Where Θ is **polar agle** (angle from the z axis), and φ is **azimuth** (angle measured counter-clockwise from the x axis).

Definition of SH:

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2}K_l^m \cos(m\varphi)P_l^m(\cos\theta), & m > 0 \\ \sqrt{2}K_l^m \sin(-m\varphi)P_l^{-m}(\cos\theta), & m < 0 \\ K_l^0 P_l^0(\cos\theta), & m = 0 \end{cases}$$

- K … normalization constant
- P … Associated Legendre polynomial
  - Orthonormal polynomial basis on (0,1)
- In general: $Y_{l,m}(\theta,\varphi) = K \,.\, \Psi(\varphi) \,.\, P_{l,m}(\cos\theta)$
  - $Y_{l,m}(\theta,\varphi)$ is separable in θ and φ

Function approximation using SH:

We have:

$$G(\Theta,\varphi) = \sum_{l=0}^{n-1} \sum_{m=-l}^{m=l} c_{l,m} Y_{l,m}(\Theta,\varphi)$$

Where n is the **approximation order**. There are n^2 harmonics for order n.

Since, SH are **orthonormal**, we can compute the coefficients as follows:

$$\begin{aligned} c_{l,m} &= \langle G | Y_{l,m} \rangle \\ &= \int_S G(\omega)\, Y_{l,m}(\omega) d\omega \\ &= \int_0^{2\pi}\int_0^{\pi} G(\theta,\varphi)\, Y_{l,m}(\theta,\varphi) \sin\theta \, d\theta \, d\varphi \end{aligned}$$

This is then evaluated by numerical integration (Monte Carlo).

Product integral with SH:

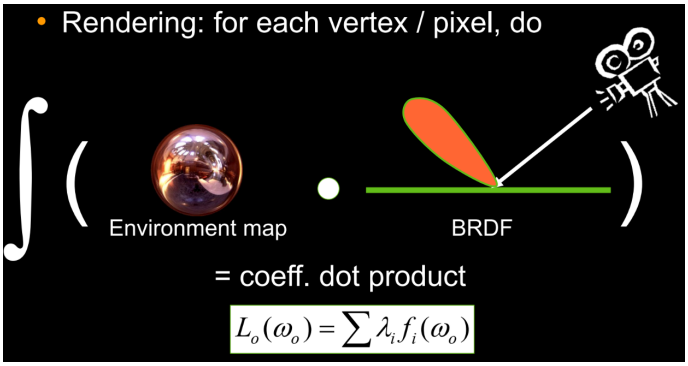
- Two functions represented by SH

$$F(\omega) = \sum_{i=0}^{n^2} f_i Y_i(\omega) \qquad G(\omega) = \sum_{i=0}^{n^2} g_i Y_i(\omega)$$

- Product integral

$$\int_S F(\omega)G(\omega)d\omega = \sum_{i=0}^{n^2} f_i g_i$$

Application:



- SH-based shading of arbitrary BRDFs
- SH irradiance environment mapping

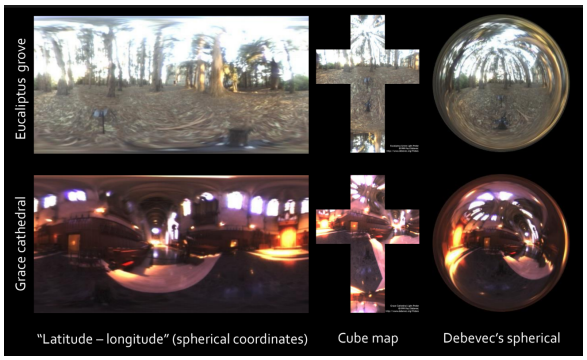
11. PBR - Image-based lighting (IBL; Environment Mapping)

What Is IBL Used For

Recording a spherical function in one or more images is called **environment mapping**, as we typically use texture mapping to implement the lookups in the table. This table is called **environment map** and stores the radiance values. We can allow to have just a 2D table as the incoming radiance L_i is dependent on only the direction.

The idea of illuminating a scene with texture data is also known as **image-based lighting (IBL)**, typically when the environment map is obtained from real-world scenes using cameras capturing 360 degree panoramic, high dynamic range images.

Having the radiance values stored in a texture means that we do not have to compute them in the rendering equation. Also, having an environment map yields richer, more realistic pictures.



Purpose and Assumptions of IBL; Advantages and Disadvantages

We have a basic assumption for any global spherical function, i.e., one used for all objects in the scene, that the incoming radiance L_i is **dependent on only the direction**. This assumption requires that the objects and lights being reflected are far away, and that **the reflector does not reflect itself**. For real-time rendering we also often assume **no shadowing**, **no interreflections**.

Advantages are that it is simple and fast to decode in real-time (compared to other spherical representations); by increasing the texture's resolution we can express spherical signals of arbitrarily high frequency and by increasing each channel's number of bits we can capture any range of environment radiance.

Disadvantages are however that it consumes more memory (compared to other spherical representations). The radiance values stored in environment maps usually have a high dynamic range. More bits per texel mean environment maps tend to take up more space than other textures and can be slower to access.

The Difference Between IBL and Global Illumination

IBL stores radiance values that were captured in a real-world scene. These values are then just put in the rendering equation. Global illumination however are algorithms that rigorously compute the rendering equation by following a ray (photon) and simulate its path around the scene. These computations are very heavy and are not used in real-time rendering (as they are).

The Role of Spherical Harmonics (SH)

Techniques of evaluating the BRDF tend to be slow, costly to compute and/or are costly to store. Because of this an approach using a spherical harmonics for lighting and (arbitrary) BRDF functions was developed.

Essentially, the BRDF is tabulated not in terms of response to directional lights, but to large lights covering the sphere via the orthonormal SH basis. The lighting environment also is projected into the SH basis using a fast, on-the-fly calculation.

Representation using the spherical harmonics is used. Environment map is represented as one set of coefficients. And scene BRDFs using one coefficient vector for each discretized viewing direction. BRDF coeffs are precomputed for all scene BRDFs (computing the coeffs is also called **SH-projection**). BRDF coefficient vector $[f_i]$ for a given ω_o is then looked up from a texture (use e.g. paraboloid mapping to map ω_o to a texture coordinate).

Computation of the rendering eequation using the environment map and SH then becomes a dot product:

$$L_o(\omega_o) = \sum \lambda_i f_i(\omega_o)$$

Where λ_i are the coefficients representing the EM and $f_i(\omega_o)$ are the coeff. vectors for s given viewing direction ω_o representing the BRDF.

The problem is however that the BRDF is in **local frame** and EM is in **global frame**. Because of this, we need to coordinate the frame alignment using **SH rotation**. SH are closed under rotation - we can use a rotation matrix. Fastest known procedure is the $zxzxz$-decomposition.

The algorithm pseudocode is:

- Preprocess
  - For each BRDF in the scene
    - For each viewing direction
      - Project the BRDF lobe onto SH basis (49-100 coefficients)
  - Whenever the EM changes
    - Project the EN onto SH basis (as many coefficients as for the "sharpest", i.e. most specular BRDF)
- Rendering in fragment shader
  - Fetch SH coefficients for the EM
  - Fetch SH coefficients for the BRDF (current viewing direction)
  - Bring the BRDF representation to the global frame using SH rotation
  - Calculate the dot product of coefficients = final pixel color

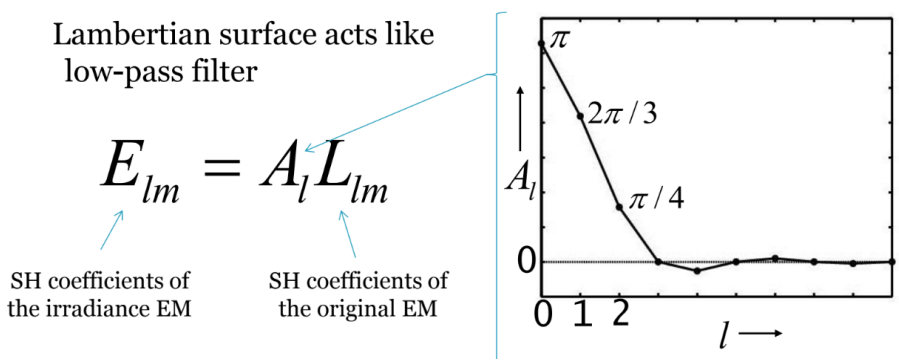
Irradiance Environment Mapping

Environment maps for **diffuse** reflections are indexed with just the surface normal n, and they contain **irradiance values**. For this reason they are called **irradiance environment maps**. Since irradiance environment maps are extremely blurred compared to the original illumination, they can be stored at significantly lower resolution.

Irradiance environment map is always a very smooth function over the sphere. As such, it can be well-represented using low-order SH expansion (order 2, 9 terms is good enough).

How to compute the coefficients:

- Project the input EM onto SH, get coeffs L_{lm}
- Multiply these coeffs by the factors A_l
- This gives the Irradiance EM SH coeffs E_{lm}



Rendering is done in **fragment shader**.
Widely used in games (AMPED for Microsoft Xbox) and movies (Pixar, Framestore CFC, ...).

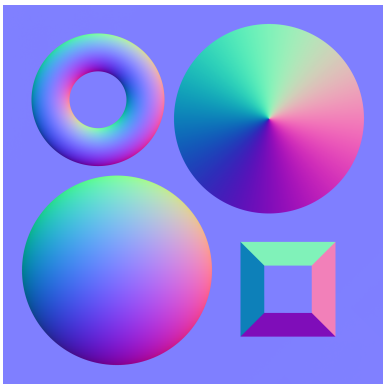
12. PBR - Pre-computed (baked) effects

Normal Mapping

Normal mapping is a texture mapping technique used for faking the lighting of bumps and dents – an implementation of **bump mapping**. It is used to add details without using more polygons. A common use of this technique is to greatly enhance the appearance and details of a low polygon model by generating a normal map from a high polygon model or height map.

Normal maps are commonly stored as regular RGB images where the RGB components correspond to the X, Y, and Z coordinates of the surface normal. X, Y and Z values are mapped to [-1, 1], for the 8-bit texture it means that value 0 represents -1.0 and 255 represents 1.0.

A problem with normal mapping is that the bumps never shift location with the view angle, nor ever block each other. For example, if you look along a real brick wall, at some angle you will not see the mortar between the bricks. A bump map of the wall will never show this type of occlusion, as it merely varies the normal. To aid this, **parallax mapping** is used.



Pre-computed Global Illumination

Global illumination

In real-time rendering, it is common to use just a local lighting model. Only the surface data at visible points are needed to compute the lighting. Primitives are processed and rasterized independently, after which they are discarded. Results of the lighting calculations at point a cannot be accessed when performing calculations at point b.

However, we know that the lights illuminating the scene produce photons which bounce around the scene and which upon every collision can be absorbed, reflected, and refracted in a variety of ways. And so, during rendering it would be nice to use information from other objects as well rather than just the one being illuminated.

Examples of algorithms (**global illumination** algorithms) that use this information from other objects are transparency, reflections and shadows.

Algorithms that solve the full rendering equation can generate stunning, photorealistic images. Those methods, however, are too computationally expensive for real-time applications. In static or partially static scenes, such algorithms can be run as a preprocess, storing the results for later use during rendering - **baked lighting**.

Baked lighting

Baked lightning is precomputed lighting information for static scenes and lights. It is typically baked to: **vertices**, **textures**, **light probe points** in space for relighting of dynamic objects. Lighting is baked for its quality, performance and lighting workflow.

Quality in a sense that we can now afford to compute multiple light bounces around the scene, for example, and do other heavy computing that would make the scene look better (and because the scene and lights are static we are allowed to make these computations just once). As the result of this we get nice global illumination affects and high quality direct lighting.

Performance also gets better as now it becomes independent of the light setup and global illumination algorithm. Baked lights also make the performance predictable and scalable (on low end platforms baked lighting is the most effective way to get good looking lighting).

Workflow is improved because it gives the artists an access to global illumination - there is no need for artificial fill lights and it decouples lighting from the geometry/materials. Baked Lighting also allows a richer set of light sources - physically based soft shadows.

Baked lighting has its **challenges** however: what do to with moving/deformed geometry?, what about memory usage? light rebuild times? changing light setups (day-night shift)?

Memory usage:
In comparison with material textures, lighting textures must be unique per instance, cannot be tiled, mirrored etc.. Possible to optimize resolution based on resolution requirements.

Another thing is that normal maps force the resolution of the light map to be the same as the normal map in order to capture the details.

Changing light setups:
It is possible to bake different times of days. Treat moving and intensity changing lights like ordinary runtime lights (good for explosions or for lights that flicker; no indirect lighting however).

Changing geometry:
For the moving characters, the naive approach is just adding direct lighting to these moving objects. This tends to make characters to look out of place. Also in regions where no direct light reaches, the characters are completely black. Light probes solve these problems elegantly and give a nice pipeline for lighting characters and other moving objects. A light probe in this case means a representation of the incoming light for an arbitrary point in space.

There also exist some **hybrid solutions**: bake only indirect lighting (saves on needed texture resolution); special treatment of the sun (bake indirect lighting only from the sun, add direct lighting as a runtime light).

Difference between IBL and baked global illumination:
IBL environment maps were captured in real world and these textures store real radiance values. Baked global illumination textures are a result of precomputed global illumination algorithms and contain colors that are than combined with the albedo of the scence objects. Also, light maps are limited to diffuse materials and have no high frequencies.

Radiosity Normal Mapping

Radiosity normal mapping **combines precomputed light maps and normal maps**.

While light mapping stores global, low-frequency illumination at sparsely sampled points in the scene, normal maps provide local, high-frequency shading variation at a high resolution. **The problem with combining the two methods** is that light maps store lighting information for only one normal direction (the base surface normal) and therefore cannot be evaluated using the normals stored in a normal map. Radiosity normal mapping aims to solve this issue.

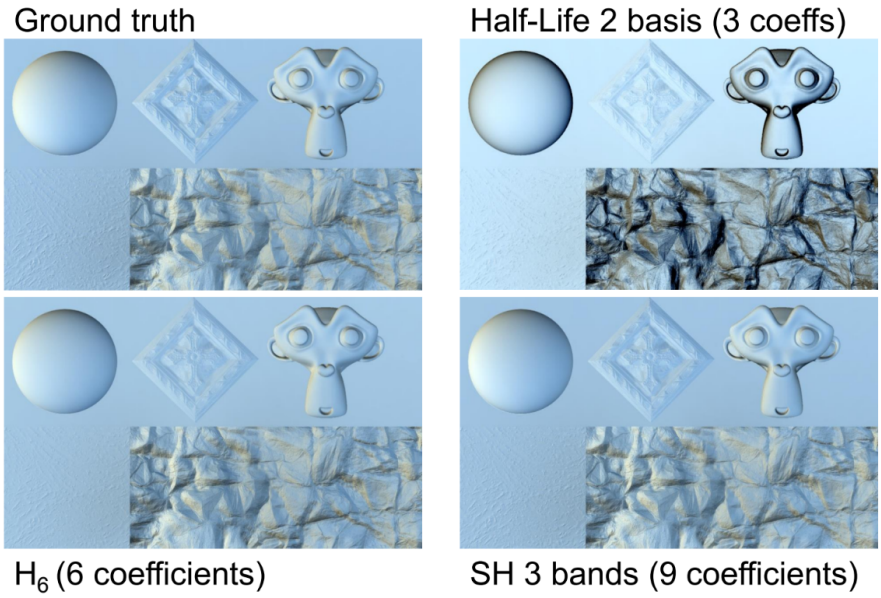
The idea is that in a radiosity light map we store directional distribution of incident light for each texel (essentially an irradiance EM for each texel).

We can represent the light map using: **irradiance EM** (9 SH coefficients (i.e. RGB triplets) per texel), **Half-life 2 basis** (3 coefficients per texel), **H-basis** (6 coefficients per texel).

SH irradiance environment maps (9 coeffs) are extremely close to reference but costly and bulky. Half-Life 2 basis is Efficient and light, but a bad approximation of reference. H6 is a good compromise.

Half-Life 2 basis represents hemispherical functions on surfaces by sampling three directions in tangent space. At the pixel level, transform the normal from a normal map into our basis; sample three light map colors, and blend between them based the transformed vector.

H-basis are spherical harmonics adapted to the hemisphere.



13. Shadow Algorithms I.

The Importance of Shadows

Shadows in a scene a important for various reasons:

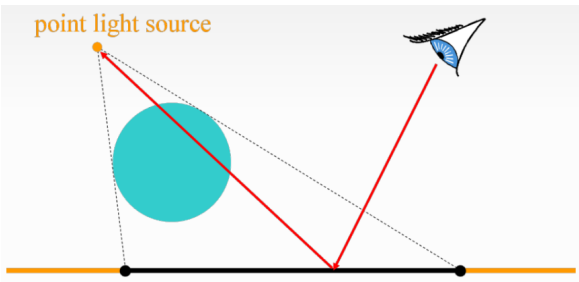
- **contact points** (it is hard to tell whether objects are floating in the air, or just placed further in a scene)
- **scene depth, depth cues** (shadows and scene lightning can help distinguish between foreground, middle ground and background)
- **realism**
- **mood** (example: in horror games many places are covered in shadow, more harsh shadows contrasting with bright light can bring a feeling of tension and unease, ...)

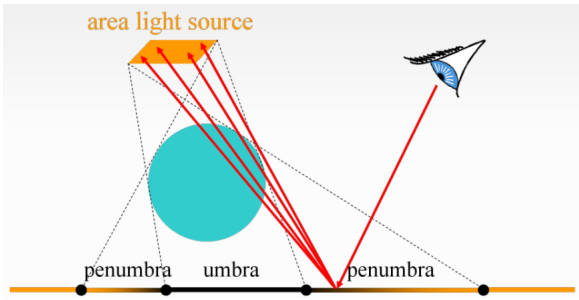
Shadow Calculation in Ray Tracing

To include shadows into a ray tracer is pretty simple. What needs to be added are **shadow rays**. These rays go from a points on a surface (which were hit by view rays) towards a light source (or all the light sources). If along the way they intersect an object, it means that that points on a surface are in shadow and light should not be applied to these points (except for maybe an ambient light).

Doing this for a **point light** means that all the shadow rays are targeted towards a single point and only a single ray needs to fired from one point on a surface. Point lights yield **hard shadows**. Sadly, point lights do not exist.

What we find in the real world however are **area lights**. On the surface of an area light we can determine several "target points" which will be targeted by the shadow rays. The points should be sufficiently randomized and should cover the whole surface equally. These points then can be viewed as several point lights whose sum of light intensity gets the intensity of the area light. This then means that instead of firing a single ray towards the light source, we fire several. Each shadow ray is targeting a different point of the area light. The resulting shadow (or light contribution) is then the sum of intensities of these "small point lights" that are not occluded by an object. This results in creating **soft shadows**, soft shadows consist of **umbra** (light completely occluded) and **penumbra** (light partially occluded)



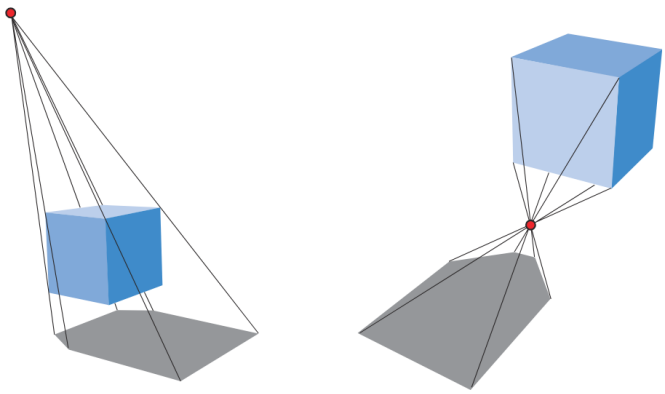


Planar shadow

Another way of making shadows is projecting them onto a plane. To render the shadow, we simply apply the projection matrix to the objects that should cast shadows on the plane, and render this projected object with a dark color and no illumination. In practice, we have to take measures to avoid allowing the projected triangles to be rendered beneath the surface receiving them. One method is to add some bias to the plane we project upon, so that the shadow triangles are always rendered in front of the surface.

If the ground plane has a limit, e.g., it is a rectangle, the projected shadows may fall outside of it, breaking the illusion. To solve this problem, we can use a **stencil buffer**. First, draw the receiver to the screen and to the stencil buffer. Then, with the z-buffer off, draw the projected triangles only where the receiver was drawn, then render the rest of the scene normally.

Another disadvantage of this method is, that all shadow casters **must be between** the light and the ground-plane receiver. If the light source is below the topmost point on the object, an **antishadow** is generated



Other disadvantages are: no self-shadowing, does not shadow other objects, problems with curved surfaces.

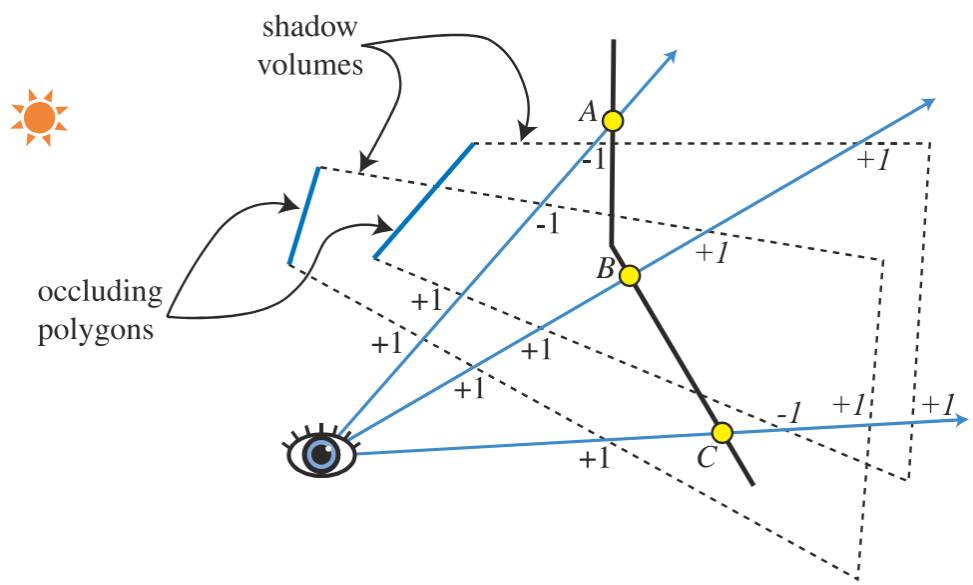
Shadow Volumes

To begin, let's imagine a point and a triangle. Extending the lines from a point through the vertices of a triangle to infinity yields an infinite three-sided pyramid. The part under the triangle, i.e., the part that does not include the point, is a truncated infinite pyramid, and the upper part is simply a pyramid. Now imagine that the point is actually a point light source. Then, any part of an object that is inside the volume of the truncated pyramid (under the triangle) is in shadow. This volume is called a **shadow volume**.

Say we view some scene and follow a ray from the eye through a pixel until the ray hits the object to be displayed on screen. While the ray is on its way to this object, we **increment** a counter each time it crosses a face of the shadow volume that is **frontfacing** (i.e., facing toward the viewer). Thus, the counter is incremented each time the ray goes into shadow. In the same manner, we **decrement** the same counter each time the ray crosses a **backfacing** face of the truncated pyramid. The ray is then going out of a shadow. We proceed, incrementing and decrementing the counter until the ray hits the object that is to be displayed at that pixel. If the counter is **greater than zero**, then that pixel is **in shadow**; otherwise it is not. This principle also works when there is more than one triangle that casts shadows.

Because doing this with rays is consuming, we use **stencil buffer** which does the counting for us. Also, doing a **scissor test** saves a fillrate.

There are two variants of this algorithm: **z-pass** and **z-fail**. Z-pass functions as described above. Z-fail works similarly, with the exception that the counting **starts beyond the surface**. Because of this, the shadow volumes **must be capped**. Z-fail provides the same counting result but works faster and solves problem when camera is in a shadow.



Shadow volumes however are **not used today**. Reason for that being that they are extremely variable - viewing toward the light can cause huge, unpredictable jumps in the cost of the algorithm, as can other scenarios. This variability is what makes shadow volumes unusable in interactive applications where a consistent frame rate is important. Another drawback is that they are impossible to do for complex geometry (e.g. trees)

Algorithm for the z-pass:

1. Depth pre-pass; disable z-write
2. Render the scene with ambient light only
3. For each light render shadow volumes:
   - Disable color write; enable stencil write
   - Do 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

Algorithm for the z-fail:

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

14. Shadow Algorithms II.

The Importance of Shadows

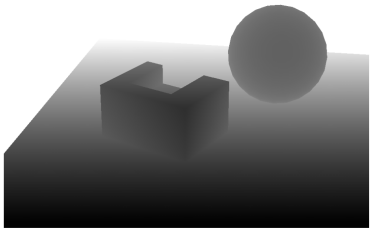
Shadows in a scene a important for various reasons:

- **contact points** (it is hard to tell whether objects are floating in the air, or just placed further in a scene)
- **scene depth, depth cues** (shadows and scene lightning can help distinguish between foreground, middle ground and background)
- **realism**
- **mood** (example: in horror games many places are covered in shadow, more harsh shadows contrasting with bright light can bring a feeling of tension and unease, ...)

Shadow Mapping

The idea is to render the scene, using **the z-buffer**, from the position of the light source that is to cast shadows. Whatever the light "sees" is illuminated, the rest is in shadow.

Each pixel in the z-buffer now contains the z-depth of the object closest to the light source. We call the entire contents of the z-buffer **the shadow map**, also sometimes known as **the shadow depth map** or **shadow buffer**. To use the shadow map, the scene is rendered a second time, but this time with respect to the viewer. As each drawing primitive is rendered, its location at each pixel is compared to the shadow map. If a rendered point is farther away from the light source than the corresponding value in the shadow map, that point is in shadow, otherwise it is not.



For **a distant directional light** such as the sun, the light's view is set to encompass all objects casting shadows into the viewing volume that the eye sees. The light uses an orthographic projection, and its view needs to be made wide and high enough in x and y to view this set of objects.

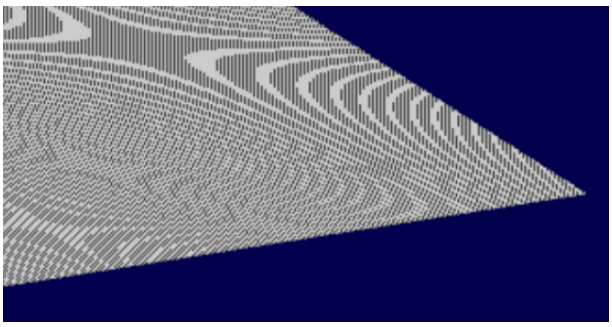
If the local light source is inside a scene and is surrounded by shadow-casters, a typical solution is to use a six-view cube, similar to cubic environment mapping. These are called **omnidirectional shadow maps**.

Not all objects in the scene need to be rendered into the light's view volume. First, only objects that can cast shadows need to be rendered. For example, if it is known that the ground can only receive shadows and not cast one, then it does not have to be rendered into the shadow map.

Rendering only relevant objects not only can save time rendering, but can also reduce the size required for the light's frustum and so increase the effective resolution of the shadow map, thus improving quality.

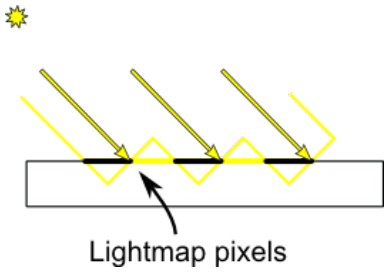
Shadow mapping is a popular algorithm because it is **relatively predictable**. The cost of building the shadow map is roughly linear with the number of rendered primitives, and access time is constant.

One disadvantage of shadow mapping is that the quality of the shadows depends on the resolution (in pixels) of the shadow map and on the numerical precision of the z-buffer. Since the shadow map is sampled during the depth comparison, the algorithm is susceptible to aliasing problems, especially close to points of contact between objects. A common problem is **self-shadow aliasing**, often called "surface acne" or **"shadow acne"**, in which a triangle is incorrectly considered to shadow itself.



This problem has two sources:

- One is simply the numerical limits of precision of the processor.
- The other source is geometric, from the fact that the value of a point sample is being used to represent an area's depth. That is, samples generated for the light are almost never at the same locations as the screen samples (e.g., pixels are often sampled at their centers). When the light's stored depth value is compared to the viewed surface's depth, the light's value may be slightly lower than the surface's, resulting in self-shadowing.



One common method to help avoid (but not always eliminate) various shadow-map artifacts is to introduce **a bias factor**. When checking the distance found in the shadow map with the distance of the location being tested, a small bias is subtracted from the receiver's distance.

This bias could be **a constant value**:

$$ShadowMap(x', y') + bias < z'$$

but doing so can fail when the receiver is not mostly facing the light.

A more effective method is to use a bias that *is proportional to the angle of the receiver to the light*. The more the surface tilts away from the light, the greater the bias grows, to avoid the problem. This type of bias is called **the slope scale bias**:

$$ShadowMap(x', y') + bias_{\Theta} < z'$$

Note that if a surface directly faces the light, it is not biased backward at all by slope scale bias. For this reason, a constant bias is used along with slope scale bias to avoid possible precision errors. Slope scale bias is also often clamped at some maximum, since the tangent value can be extremely high when the surface is nearly edge-on when viewed from the light.

Too much bias causes a problem called **light leaks** or **Peter Panning**, in which the object appears to float slightly above the underlying surface. This artifact occurs because the area beneath the object's point of contact, e.g., the ground under a foot, is pushed too far forward and so does not receive a shadow.

Resolution Enhancement

Similar to how textures are used, ideally we want one shadow-map texel to cover about one image pixel. If we have a light source located at the same position as the eye, the shadow map perfectly maps one-to-one with the screen-space pixels (and there are no visible shadows, since the light illuminates exactly what the eye sees). As soon as the light's direction changes, this per-pixel ratio changes, which can cause artifacts. The shadow is blocky and poorly defined because a large number of pixels in the foreground are associated with each texel of

the shadow map. This mismatch is called **perspective aliasing**. Single shadow-map texels can also cover many pixels if a surface is nearly edge-on to the light, but faces the viewer. This problem is known as **projective aliasing**. Blockiness can be decreased by increasing the shadow-map resolution, but at the cost of additional memory and processing.

Figure 7.16. On the left the light is nearly overhead. The edge of the shadow is a bit ragged due to a low resolution compared to the eye's view. On the right the light is near the horizon, so each shadow texel covers considerably more screen area horizontally and so gives a more jagged edge. *(Images generated by TheRealMJP's "Shadows" program on Github.)*

A good idea is adding more samples where the viewer is located, leading to algorithms that generate several shadow maps for a given view. Based on these algorithms emerged an lagorithm called **cascaded shadow maps**. The idea is to divide the view frustum's volume into a few pieces by slicing it parallel to the view direction. As depth increases, each successive volume has about two to three times the depth range of the previous volume. For each view volume, the light source can make a frustum that tightly bounds it and then generate a shadow map. This type of algorithm is straightforward to implement, can cover huge scene areas with reasonable results, and is robust. And there are no serious worst-case problems. Because of these strengths, cascaded shadow mapping is used in many applications.

Smaller view volumes for the closer shadow maps provide more samples where they are needed. Determining how the range of z-depths is split among the maps — a task called z-partitioning — can be quite simple or involved. One method is logarithmic partitioning, where the ratio of far to near plane distances is made the same for each cascade map.

A sudden change in a shadow's quality can occur when an object spans the boundary between two shadow maps. One solution is to have the view volumes slightly overlap. Samples taken in these overlap zones gather results from both adjoining shadow maps and are blended. Another problem is shimmering, which is prevented by rounding the texel values.

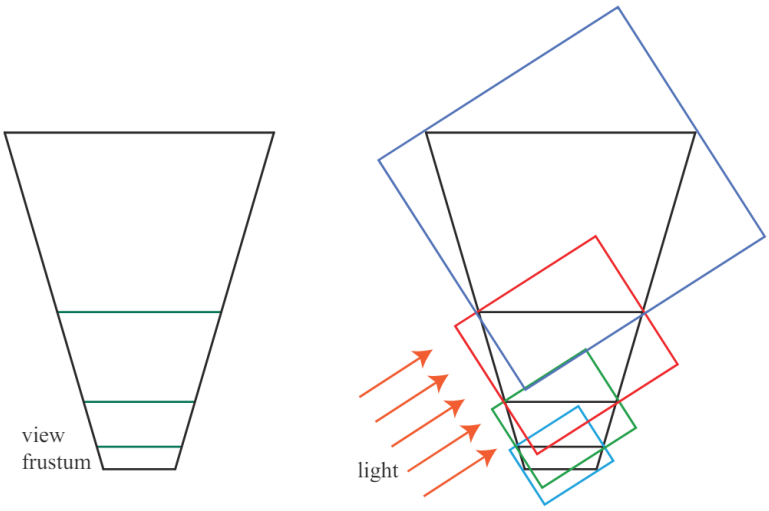


Figure 7.18. On the left, the view frustum from the eye is split into four volumes. On the right, bounding boxes are created for the volumes, which determine the volume rendered by each of the four shadow maps for the directional light. *(After Engel [430].)*

Percentage-Closer Filtering

A simple extension of the shadow-map technique can provide pseudo-soft shadows. This method can also help ameliorate resolution problems that cause shadows to look blocky when a single light-sample cell covers many screen pixels.

Instead of a single sample being taken off the shadow map, the four nearest samples are retrieved. The technique does not interpolate between the depths themselves, but rather the results of their comparisons with the surface's depth. That is, the surface's depth is compared separately to the four texel depths, and the point is then determined to be in light or shadow for each shadow-map sample. These results, i.e., 0 for shadow and 1 for light, are then bilinearly interpolated to calculate how much the light actually contributes to the surface location.

Consider 3x3 neigbourhood of pixel with depth of 49.8

Naive filtering

$$\begin{bmatrix} 50.2 & 50.0 & 50.0 \\ 50.1 & 1.2 & 1.1 \\ 1.3 & 1.4 & 1.2 \end{bmatrix} \Longrightarrow \mathit{filter} \Longrightarrow 22.9 < 49.8?$$

Percentage closer filtering

$$\begin{bmatrix} 50.2 & 50.0 & 50.0 \\ 50.1 & 1.2 & 1.1 \\ 1.3 & 1.4 & 1.2 \end{bmatrix} \Longrightarrow \mathit{cmp}(49.8) \Longrightarrow \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \Longrightarrow 0.55$$

Filter the result of the depth compare

This idea of retrieving multiple samples from a shadow map and blending the results is called **percentage-closer filtering (PCF)**. The name "percentage-closer filtering" refers to finding the percentage of the samples taken that are visible to the light. This percentage is how much light then is used to shade the surface.

Once the width of the area to be sampled is determined, it is important to sample in a way that avoids aliasing artifacts. Performing nearest neighbor sampling in a regular grid pattern can create noticeable artifacts. One solution to minimize grid patterns is to sample an area using a precomputed **Poisson distribution** pattern. This distribution spreads samples out so that they are neither near each other nor in a regular pattern.

Figure 7.23. The far left shows PCF sampling in a 4×4 grid pattern, using nearest neighbor sampling. The far right shows a 12-tap Poisson sampling pattern on a disk. Using this pattern to sample the shadow map gives the improved result in the middle left, though artifacts are still visible. In the middle right, the sampling pattern is rotated randomly around its center from pixel to pixel. The structured shadow artifacts turn into (much less objectionable) noise. *(Images courtesy of John Isidoro, ATI Research, Inc.)*

Self-shadowing problems and light leaks, i.e., acne and Peter Panning, can become worse with PCF.

A few different additional bias factors have been invented and used with some success to reduce the risk of self-shadowing. For example, it is assumed that nearby sample locations are on the same plane formed by the triangle. Referred to as **receiver plane depth bias** or other similar terms, this technique can be quite precise in many cases, as locations on this imaginary plane are indeed on the surface, or in front of it if the model is convex.

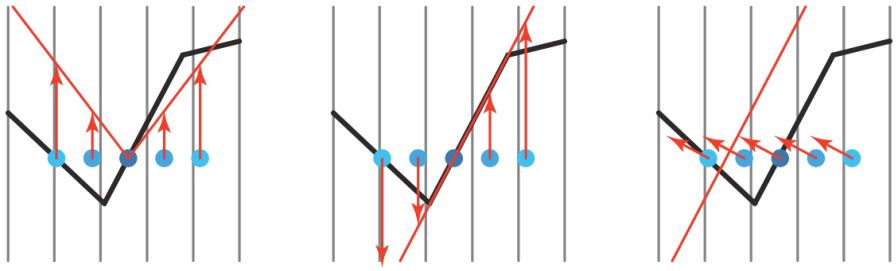
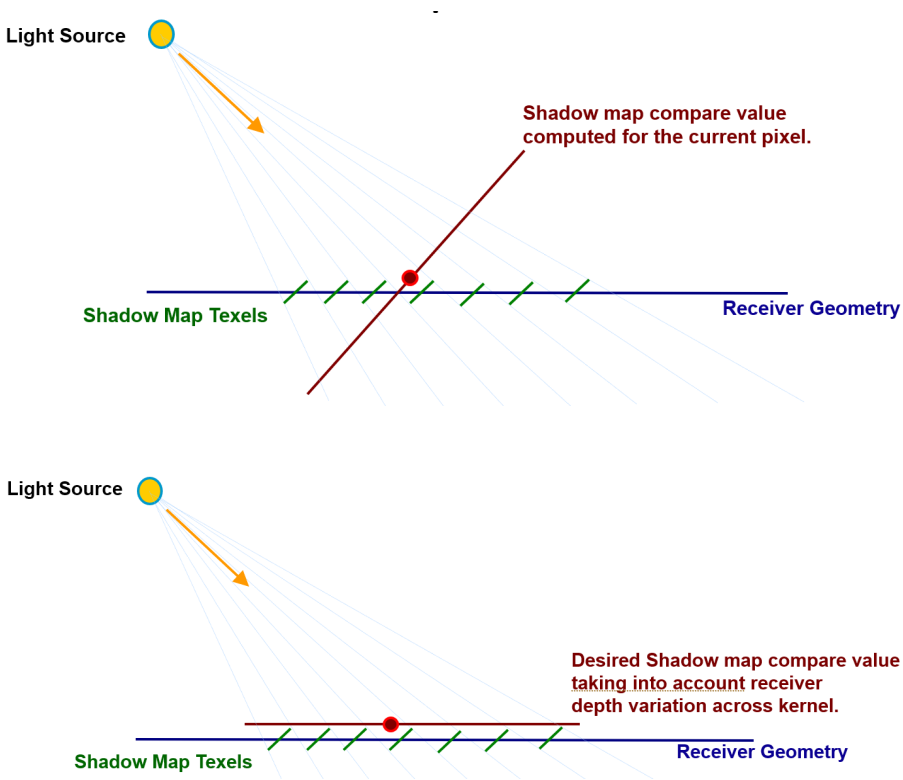


Figure 7.24. Additional shadow bias methods. For PCF, several samples are taken surrounding the original sample location, the center of the five dots. All these samples should be lit. In the left figure, a bias cone is formed and the samples are moved up to it. The cone's steepness could be increased to pull the samples on the right close enough to be lit, at the risk of increasing light leaks from other samples elsewhere (not shown) that truly are shadowed. In the middle figure, all samples are adjusted to lie on the receiver's plane. This works well for convex surfaces but can be counterproductive at concavities, as seen on the left side. In the right figure, normal offset bias moves the samples along the surface's normal direction, proportional to the sine of the angle between the normal and the light. For the center sample, this can be thought of as moving to an imaginary surface above the original surface. This bias not only affects the depth but also changes the texture coordinates used to test the shadow map.

Receiver Plane Depth Bias Details: As light hits surface at grazing angle, there is a large receiver depth variation across the kernel. This then may result in self shadowing.



In order to approximate the per-tap receiver depth, we need to know how much the depth changes with respect to shadow map coordinates. However, the derivative instruction in the pixel shader only computes derivatives with respect to screen space and we need it in texture coordinates. So, we compute it using Jacobian:

Derivative of distance to light source w.r.t. texture coordinates

Texture Space Jacobian (inverse-transpose)

Derivative of distance to light source w.r.t. screen coordinates

$$\begin{bmatrix} \frac{\partial d}{\partial u} \\ \frac{\partial d}{\partial v} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix}^{-T} \begin{bmatrix} \frac{\partial d}{\partial x} \\ \frac{\partial d}{\partial y} \end{bmatrix}$$

This tells us the change of Z with respect to UV.

One problem with PCF is that because the sampling area's width remains constant, shadows will appear uniformly soft, all with the same penumbra width. This may be acceptable under some circumstances, but appears incorrect where there is ground contact between the occluder and receiver.