

# Základy počítačové grafiky

## Přednáška 9

Martin Němec

VŠB-TU Ostrava

2024

# Nahradí programátory v budoucnu umělá inteligence?



Jaký význam má funkce `dot` v glsl ?



V jazyce GLSL (OpenGL Shading Language), funkce `dot` slouží k výpočtu skalárního součinu dvou vektorů. Tato funkce má významný vliv v mnoha grafických operacích, zejména při práci s osvětlením a stínováním v 3D grafice.



Představme si, že máme dva vektory v prostoru: `'vec3 A'` a `'vec3 B'`. Skalární součin (`'dot product'`) těchto vektorů můžeme získat pomocí funkce `'dot'` následovně:

glsl



Copy code

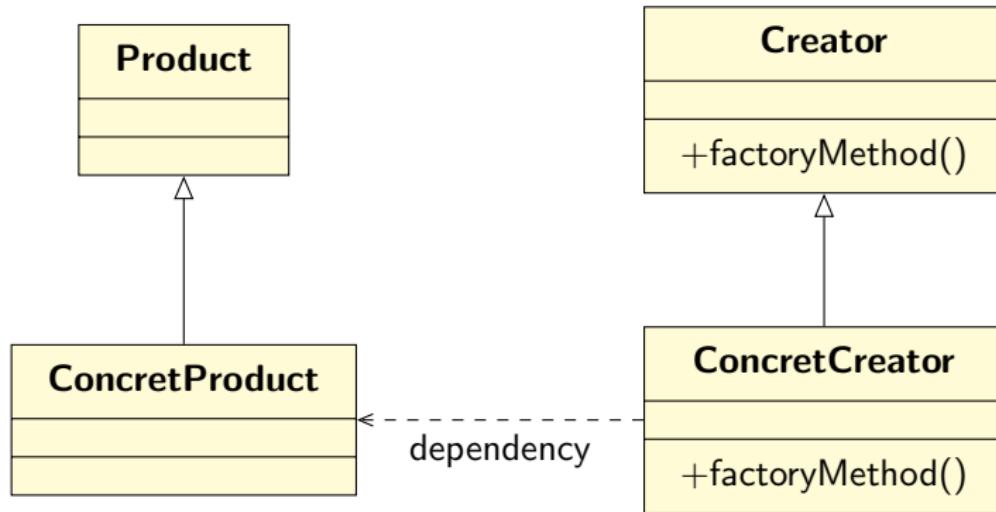
```
float result = dot(A, B);
```

Skalární součin nám poskytuje informaci o tom, jak jsou tyto vektory orientovány relativně sobě. Pokud jsou vektory kolmé, skalární součin je nula. Jestliže jsou vektory podobně orientovány, je skalární součin kladný, a pokud jsou opačně orientovány, je skalární součin záporný.

V praxi se v GLSL může funkce `'dot'` použít v mnoha situacích, zejména v osvětlovacích modelovačích technikách. Například při výpočtu odlesku (specular reflection) ve fázovém modelu Phong, může být funkce `'dot'` použita k určení úhlu mezi normálou povrchu a směrem pohledu nebo směrem světla. Tato informace pak může ovlivnit intenzitu odlesku na povrchu.

## Je něco, co byste chtěli zopakovat?

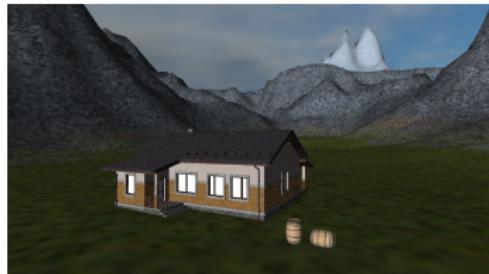
# Factory Method



# Načítání externích modelů

Více možnosti jak modely načíst

- vlastní načítání modelů (obvykle OBJ);
- využití projektů jako je tinyobjloader;
- použití importeru jako je Assimp.



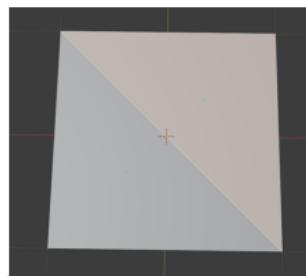
Máme více možností vyberte si libovolnou. Důležité je to, v jakém formátu máme data (model, textury atd.).

# OBJ

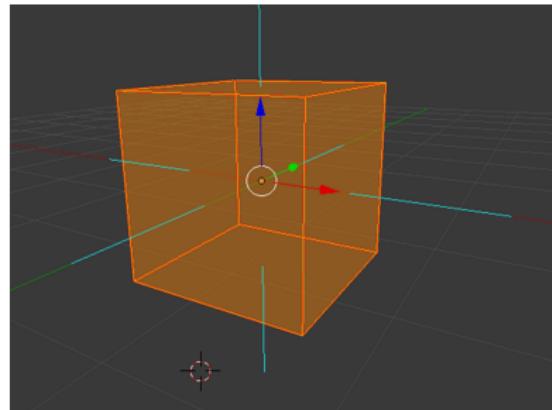
OBJ je textový formát pro ukládání geometrie (Wavefront Technologies).

- # - komentář
- v - vrchol
- o název objektu
- vt - souřadnice textury
- vn - normála
- f - plocha (face)
- s - smooth
- mtllib - materiál

```
# Blender 3.3.1
# www.blender.org
mtllib untitled.mtl
o Plane
v -1.000000 0.000000 1.000000
v 1.000000 0.000000 1.000000
v -1.000000 0.000000 -1.000000
v 1.000000 0.000000 -1.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 0.000000 0.000000
vt 1.000000 1.000000
vn 0.0000 1.0000 0.0000
usemtl Material
s off
f 2/1/1 3/2/1 1/3/1
f 2/1/1 4/4/1 3/2/1
```

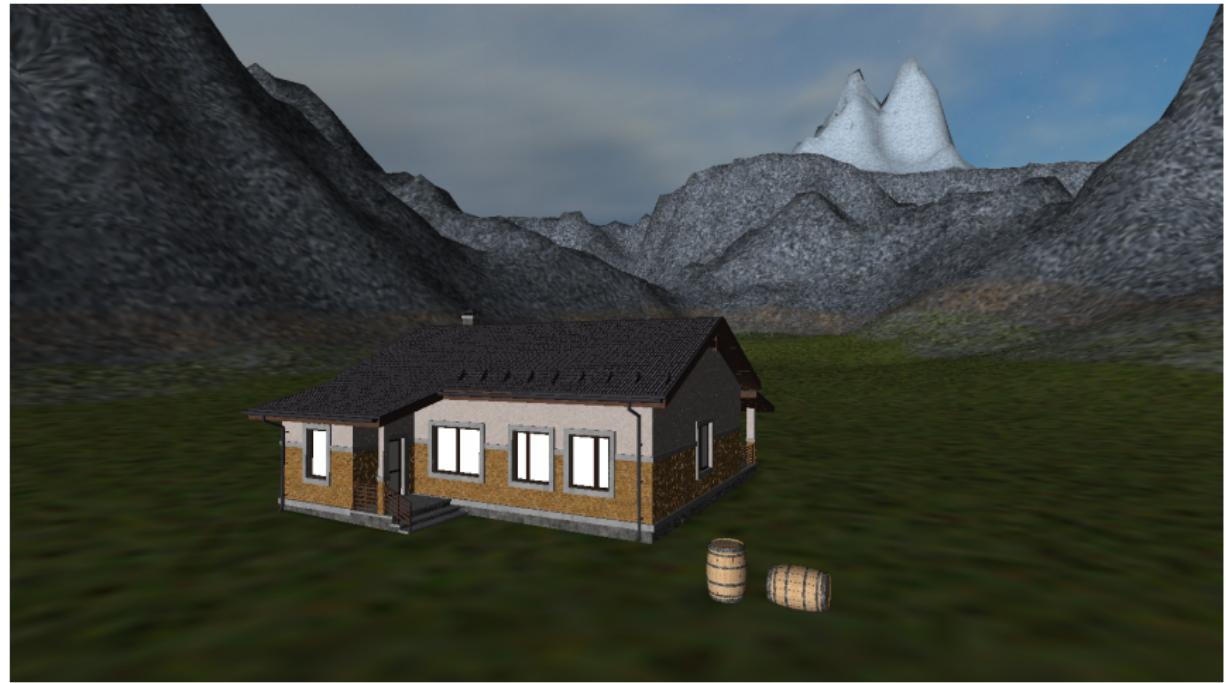


# OBJ



```
# Blender 3.3.1
# www.blender.org
mtllib untitled.mtl
o Cube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -0.999999
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 1.000000 1.000000
vt 0.000000 1.000000
vt -0.000000 0.000000
vt 1.000000 -0.000000
vn 0.000000 1.000000 0.000000
vn 1.000000 0.000000 0.000000
vn -0.000000 -0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 -1.000000 0.000000
usemtl Material
s 0
f 5/1/1 8/2/1 7/3/1 6/4/1
f 1/4/2 5/1/2 6/2/2 2/3/2
f 2/4/3 6/1/3 7/2/3 3/3/3
f 4/4/4 3/3/4 7/2/4 8/1/4
f 5/4/5 1/1/5 4/2/5 8/3/5
f 1/4/6 2/1/6 3/2/6 4/3/6
```

## Ukázka



# Assimp - Supported file formats

Assimp (Open Asset Import Library) - C++ knihovna určená k načítání scény (modely, materiály, nastavení atd.) z různých formátu (<http://www.assimp.org/>).

- Wavefront Object (\*.obj);
- Filmbox (\*.fbx);
- Collada (\*.dae; \*.xml);
- Biovision BVH (\*.bvh);
- 3D Studio Max 3DS (\*.3ds);
- Stanford Polygon Library (\*.ply);
- Autodesk DXF (\*.dxf);
- Ogre (\*.mesh.xml, \*.skeleton.xml, \*.material);
- atd.

Načte scénu z různých modelů, následně je to už na nás.

# Assimp - Importer

```
#include <assimp/Importer.hpp> //C++ importer interface
#include <assimp/scene.h> //aiScene output data structure
#include <assimp/postprocess.h> //Post processing flags

Assimp::Importer importer;

unsigned int importOptions =
| aiProcess_OptimizeMeshes
| aiProcess_JoinIdenticalVertices
| aiProcess_Triangulate
| aiProcess_CalcTangentSpace
| ...
//aiProcess_GenNormals, ai_Process_GenSmoothNormals,
//aiProcess_FlipUVs, aiProcess_RemoveRedundantMaterials

const aiScene* scene =
    importer.ReadFile(fileName, importOptions);
```

# Assimp - aiScene

aiScene - obsahuje načtená data (modely, materiály, textury, světla, kamery atd.).

```
aiScene* scene; // test objekty
    bool HasMeshes()
    bool HasTextures()
    bool HasMaterials()
    // atd.

if (scene->HasMeshes){ ... }

scene->mNumMeshes;
scene->mNumMaterials;
scene->mNumTextures;
scene->mNumLights;
scene->mNumCameras;
```

# Assimp - aiScene

Procházení načtených částí.

```
//Mesh telesa
for (int i=0; i<scene->mNumMeshes; i++)
    aiMesh* mesh = scene->mMeshes[i];
```

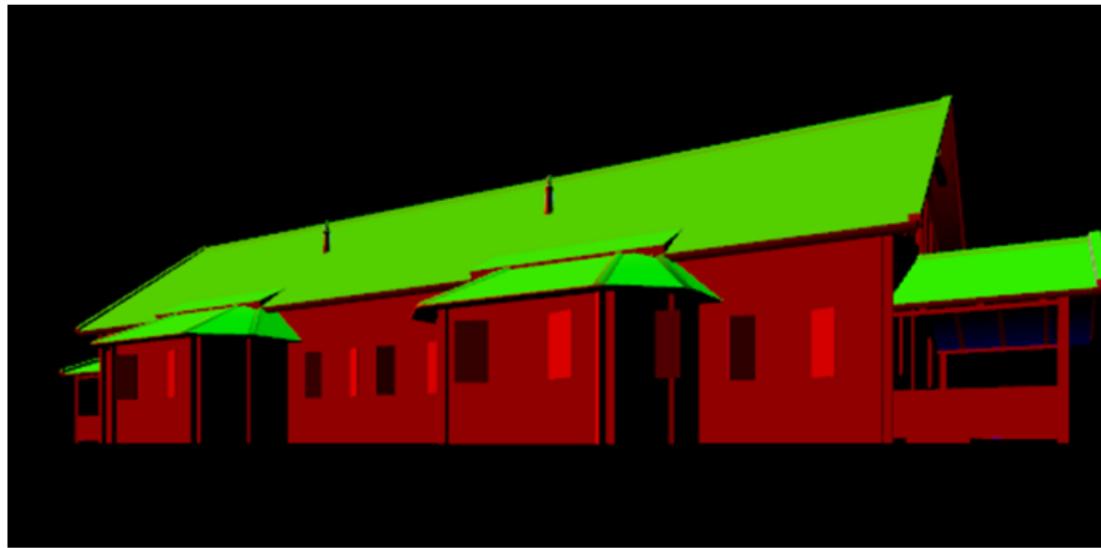
# Assimp - aiMesh

```
aiMesh* mesh; //Mesh telesa
if (scene) {
    aiMesh* m = scene->mMeshes[0];
    count = m->mNumFaces * 3;
    for (unsigned int i = 0; i < m->mNumFaces; i++){
        for (unsigned int j = 0; j < 3; j++){
            data.push_back(m->mVertices[m->mFaces[i].mIndices[j]].x);
            data.push_back(m->mVertices[m->mFaces[i].mIndices[j]].y);
            data.push_back(m->mVertices[m->mFaces[i].mIndices[j]].z);
            data.push_back(m->mNormals[m->mFaces[i].mIndices[j]].x);
            data.push_back(m->mNormals[m->mFaces[i].mIndices[j]].y);
            data.push_back(m->mNormals[m->mFaces[i].mIndices[j]].z);
            data.push_back(m->mTextureCoords[0][m->mFaces[i].mIndices[j]].x);
            data.push_back(m->mTextureCoords[0][m->mFaces[i].mIndices[j]].y);
        }
    }
}

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
    8 * sizeof(float), (GLvoid*)0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    8 * sizeof(float), (GLvoid*)(sizeof(float) * 3));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
    8 * sizeof(float), (GLvoid*)(sizeof(float) * 6));
```

# Ukázka



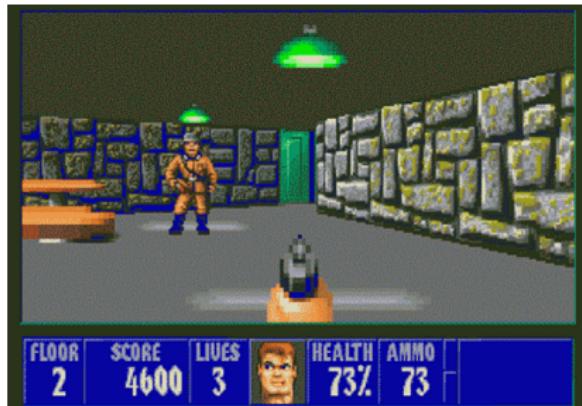
MGA - Blender + Unreal Engine 5



# Offline vs. real-time rendering

Se vznikajícím výkonem počítačů se otevírají nové možnosti jak se přiblížit realitě.

- Offline rendering (realistický) - cílem je se co nejblíže přiblížit realitě.
- Real-time rendering - omezující je čas renderování (obvykle minimum frame/s).



# Lokální vs. globální zobrazovací metody

## Lokální zobrazovací metody

- Pouze přímé osvětlení.
- Objekty se navzájem neovlivňují z hlediska osvětlení (vidíme jen to, co je osvětlené).
- Počítáme pouze s jedním odrazem paprsku od povrchu objektu. Každý objekt ve scéně má individuálně (lokálně) vyhodnocen osvětlovací model nezávisle na ostatních objektech.
- Jediný případ, kdy se pozice objektů bere v potaz, je při vypočtu viditelnosti (Z-buffer).

## Globální zobrazovací metody

- Výpočet nepřímého osvětlení.
- světelnými zdroji mohou být objekty samy sobě. Metody pracují s vícenásobnými odrazy světla.

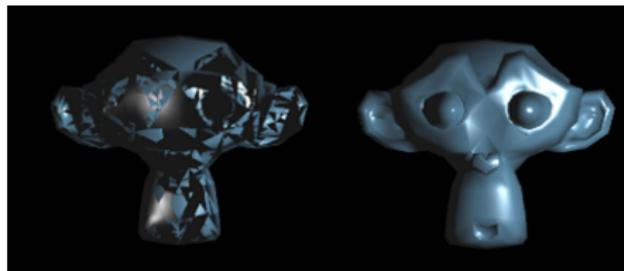
# Algoritmy viditelnosti

## Podle výsledných dat

- Vektorové algoritmy - geometrické prvky vrcholy, hrany a stěny. Výstupem vektorové řešení.
- Rastrové algoritmy - výsledkem je rastrový obraz (jednotlivé pixely obsahující barvu), většina současných metod.

## Podle místa řešení

- Řešení v prostoru objektů - porovnávání vzájemné polohy těles, složitost  $O(n^2)$ .
- Řešení v prostoru obrazu - pracujeme s promítnutými a rasterizovanými objekty. Pro pixely hledáme nejbližší objekty.



# Odstraňování neviditelných hran

Mesh tělesa (objekty) jsou "oplátována" rovinami. Vyjdeme z rovnice roviny

$$\alpha : Ax + By + Cz + D = 0$$

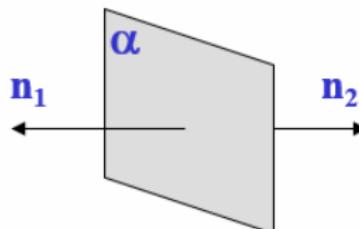
pokud je bod  $P[x_p, y_p, z_p]$  bodem roviny  $\alpha$ , potom  $k = 0$

$$\alpha : Ax_p + By_p + Cz_p + D = k, k = 0$$

Pokud je  $k > 0$ , leží bod "nad" rovinou  $\alpha$ .

Pokud je  $k < 0$ , leží bod "pod" rovinou  $\alpha$ .

V PG domluva, normálový vektor míří z tělesa ven.



# Rastrové algoritmy

Mezi základní algoritmy patří:

- Malířův algoritmus (Painter's algorithm)
- Dělení obrazovky (Warnock subdivision)
- Plovoucí horizont (Floating Horizon Algorithm)
- Z-buffer (Depth-buffer)
- atd.

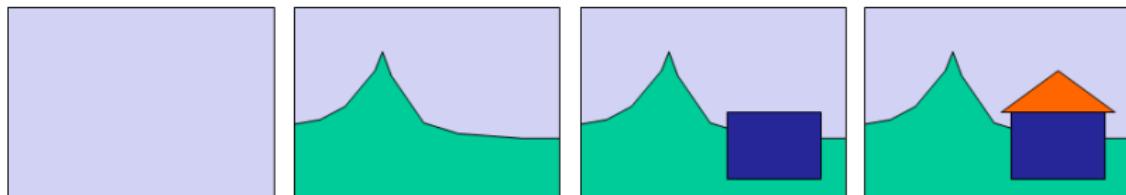
V OpenGL je nejčastěji používaným algoritmem Z-buffer neboli paměť hloubky (depth-buffer).

# Malířův algoritmus

Algoritmus a jeho chování:

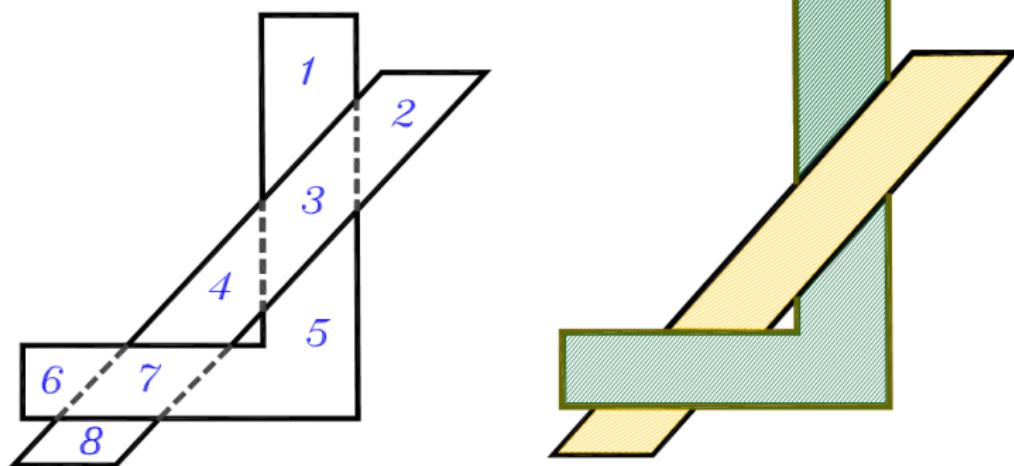
- Porovnej jednotlivé plochy z hlediska intervalu jejich z-tových souřadnic. Pokud lze rozhodnout, tak se plocha se vzdálenější z-tovou souřadnici vykreslí první.
- Jestliže se plochy v rovině xy nepřekrývají, potom na pořadí vykreslení nezáleží.
- Pokud nelze rozhodnout, musíme rozdělit na nepřekrývající plochy.

Závislé na třídících algoritmech.



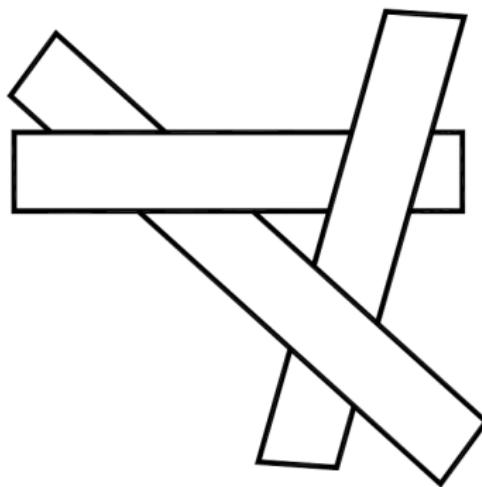
# Malířův algoritmus

Pokud nelze určit které těleso vykreslíme jako první, musíme těleso rozdělit. Problémy s nekonvexními tělesy.



# Malířův algoritmus

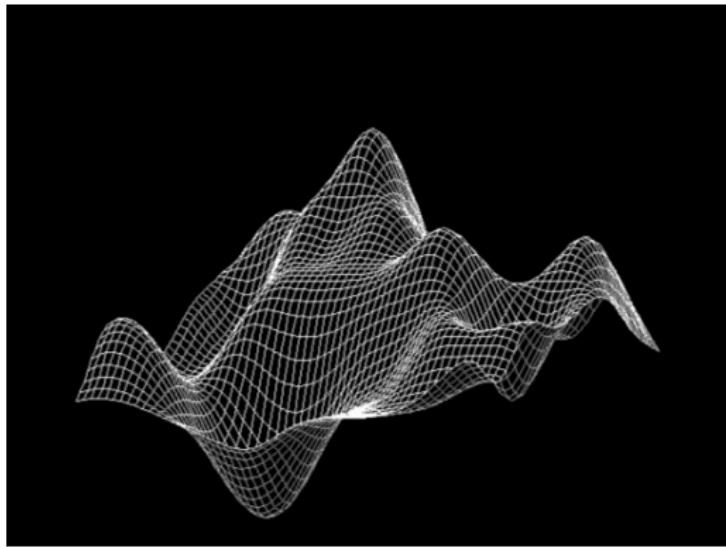
Ani rozdelení ploch na konvexní objekty nezaručuje správnost (možnost zacyklení).



Při pootočení ve scéně se může skokově měnit rychlosť vykreslování.

# Plovoucí horizont

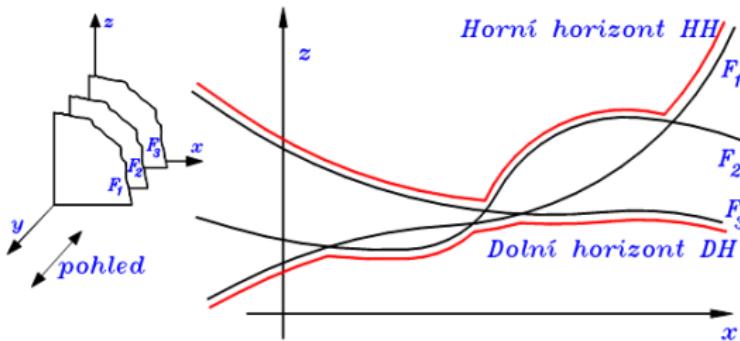
Používáno pro zobrazení funkci dvou proměnných vyjádřenou v explicitním tvaru  $z = F(x, y)$ .



# Plovoucí horizont

Postup:

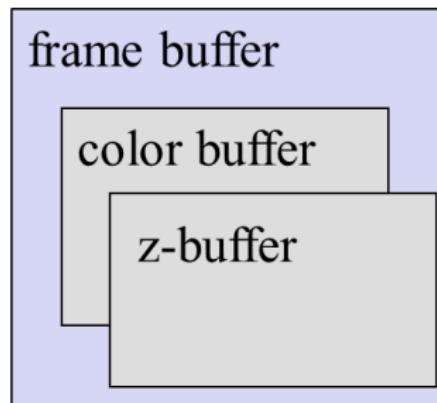
- 1 Inicializujeme masky horizontu (horní a dolní hranice).
- 2 Vykreslujeme postupně řezy plochy mimo hranice nastavené masky (nad horní a pod dolní horizont).
- 3 Upraveni obou horizontů.
- 4 Pokračujeme dalším řezem bodem 2.



# Z-buffer

Algoritmus z-buffer, neboli paměť hloubky. Připisováno Edwinu Catmullovi. Myšlenku popsal v roce 1974 ve své práci Wolfgang Straßer.

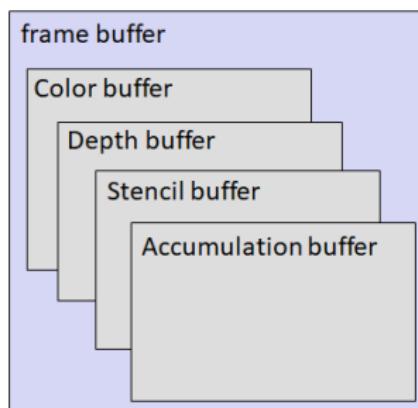
- Vyplň color buffer barvou pozadí.
- Vyplň z-buffer – nekonečnem.
- Pro každou plochu najdi její průměr a jednotlivé fragmenty  $[x_i, y_i, z_i]$  (rasterizace).
- Porovnej hloubku  $z_i$  a případně zapiš do paměti.



# Frame buffer

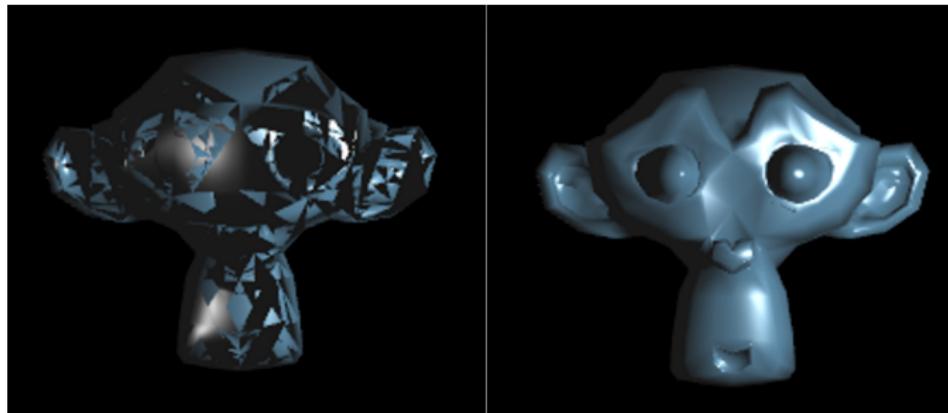
Frame buffer slouží pro ukládání výsledku zobrazovacího řetězce

- color buffer - buffer pro barvu;
- depth buffer - paměť hloubky (z-buffer);
- stencil buffer - paměť šablony;
- accumulation buffer - akumulační buffer.



## Z-buffer - paměť hloubky

- Nejznámější a nejfektivnější.
  - Každá plocha se zpracovává pouze jednou.
  - Doba zpracování roste s počtem ploch lineárně (záleží i na velikosti ploch).
  - Není potřeba žádné třídění nebo pomocné datové struktury.
  - Velkou výhodou je možnost paralelizace.



# Z-buffer

Mezi základní příkazy pro práci se z-bufferem patří: Zapnutí,

vypnutí z-bufferu:

```
glEnable(GL_DEPTH_TEST); glDisable(GL_DEPTH_TEST);
```

Vyčistění z-bufferu:

```
glClear(GL_DEPTH_BUFFER_BIT);
```

Obvykle i s barevným bufferem:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Zapnutí a vypnutí zapisování do z-bufferu:

```
glDepthMask(bool write); //GL_FALSE, GL_TRUE
```

Nastavení srovnávací funkce *glDepthFunc()*:

GL\_ALWAYS, GL\_NEVER, GL\_EQUAL, GL\_LESS, GL\_GREQUAL,  
GL\_GREATER atd.

# Z-buffer - průhledné plochy

Algoritmus z-buffer, neboli paměť hloubky.

- Inicializuj color buffer a depth buffer.
- Postupně načti všechny plochy, neprůhledné zpracuj, průhledné si zapamatuj a odlož pro následné zpracování.
- Po zpracování neprůhledných ploch setříd průhledné plochy podle vzdálenosti.
- Zpracuj průhledné plochy s použitím alfa míchání (blending).

# Globální zobrazovací metody

Tělesa se ovlivňují navzájem (zastínění, odraz, průhlednost, lom na rozhraní dvou prostředí atd.). Jedná se o metody:

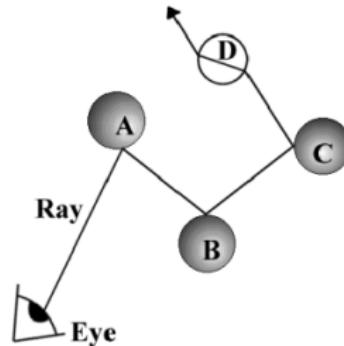
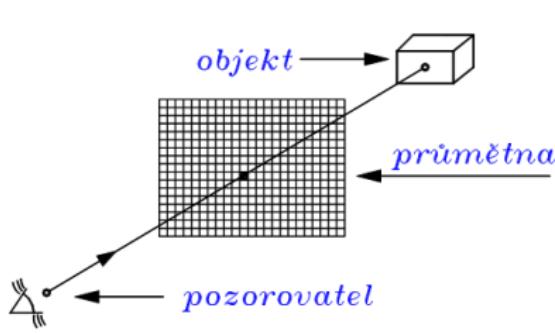
- sledování paprsku (ray tracing);
- sledování cesty (path tracing);
- radiozitní metoda (radiosity);
- atd.



# Ray tracing

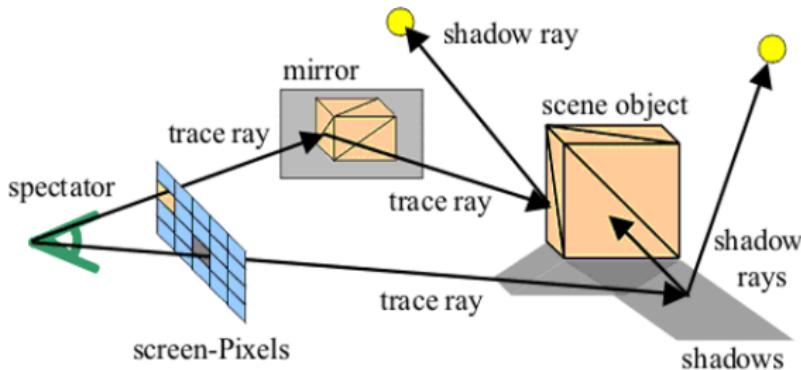
Základem je výpočet průsečíku paprsku a těles ve scéně. Zpětné sledování.

- **Primární paprsek** (primary ray) - od pozorovatele do scény.
- **Sekundární paprsek** (secondary ray) - odražené nebo paprsky procházející průhlednými tělesy.
- **Stínový paprsek** (shadow ray) - hledám, zda je nějaký objekt stínící aktuální pozici (stačí jeden, nehledám nejbližší).



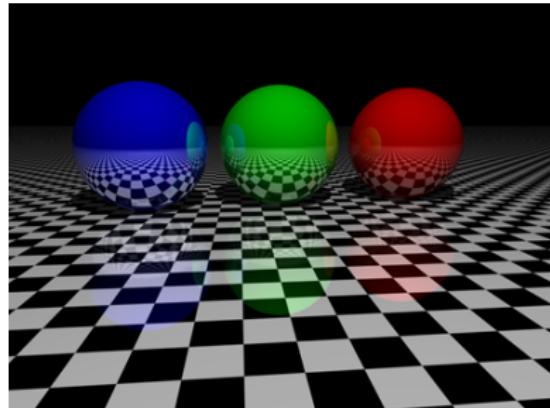
# Ray tracing

- Primární paprsek (primary ray) - od pozorovatele do scény.
- Sekundární paprsek (secondary ray) - odražené nebo paprsky procházející průhlednými tělesy.
- Stínový paprsek (shadow ray) - hledám, zda je nějaký objekt stínící aktuální pozici (stačí jeden, nehledám nejbližší).



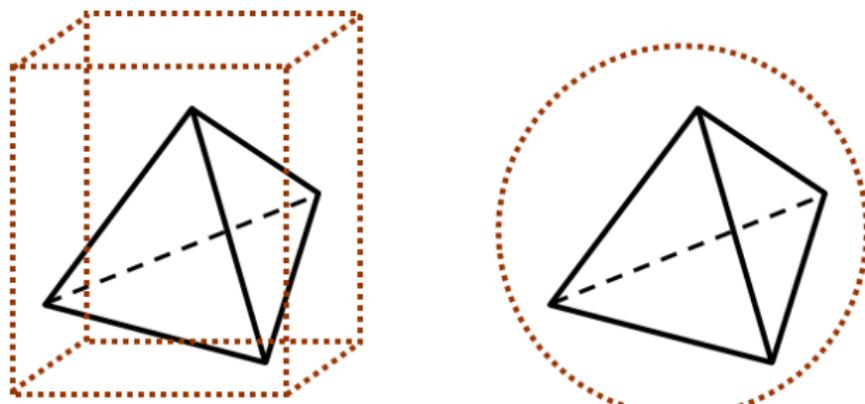
# Ray tracing - nevýhody

- Zobrazování ostrých stínů.
- Lesklé plochy (zrcadla) sice odrážejí své okolí, ale neodráží světlo do okolí (sekundární zdroj světla).
- Při změně scény (místo pozorovatele, nové světlo, nový objekt atd.) se musí vyhodnotit znova.
- Klasická metoda pouze pro bodové světla.



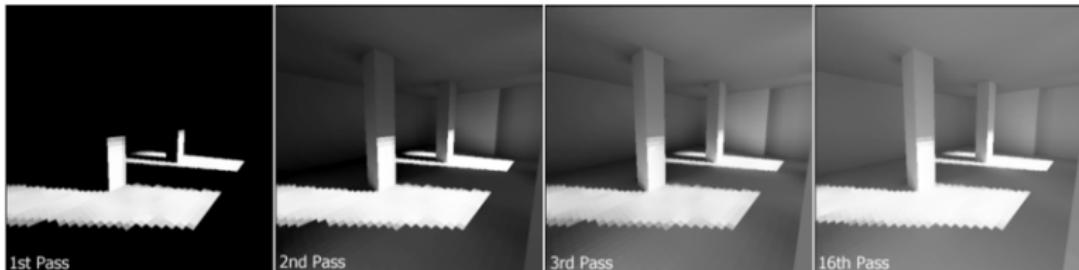
# Ray tracing - urychlení výpočtu

"Obalování" složitých těles - koule, krychle, hranoly atd.



Renderovací metoda představena 1984 výzkumníky na Cornell University. Polygony rozděleny na malé plošky, výpočet konfiguračního faktoru (vliv každé plošky na každou jinou plošku ve scéně).

- Časově náročné.
- Vychází ze zákona zachování energie (vyžaduje energeticky uzavřenou scénu).
- Nedokáže pracovat s průhlednými objekty, zrcadly.



Výpočet může probíhat:

- iteračně (progresivně);
- řešením soustavy rovnic (maticové řešení).



# Počítačová grafika 1

Více o globálních metodách v předmětu Počítačová grafika 1  
(PG1, path-tracing).



**Dotazy?**