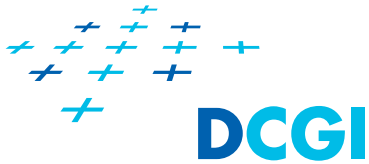


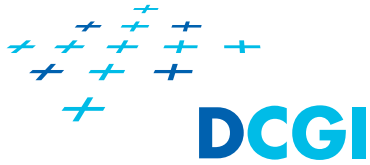
Struktura scény
Transformace (2)

Petr Felkel
Katedra počítačové grafiky a interakce, ČVUT FEL
místnost KN:E-413 (Karlovo náměstí, budova E)
E-mail: felkel@fel.cvut.cz

S použitím materiálů Bohuslava Hudce, Jaroslava Sloupa a Vlastimila Havrana

Poslední změna: 13.3.2013

Osnova



Struktura scény

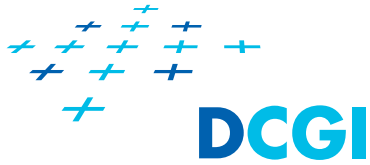
- Modelování a reprezentace scény
  - Lineární (nehierarchická) reprezentace
  - Hierarchická reprezentace
  - Vlastní graf scény – příprava pro cvičení

Transformace (2)

- Homogenní souřadnice
- Projekce a viewport
- Rotace podle Eulerových úhlů a Gimbal lock
- Virtuální trackball

Struktura scény

Modelování



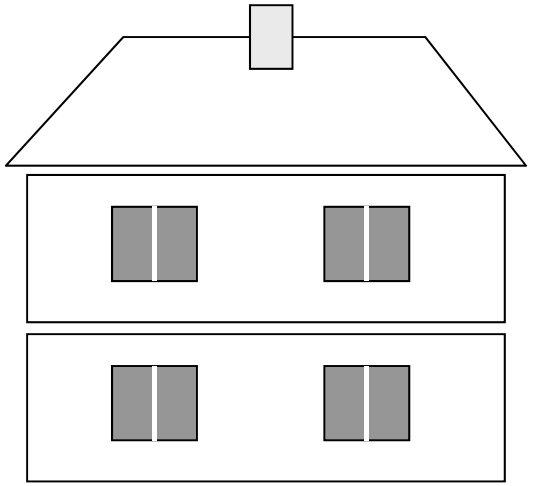
- **modely** = abstrakce virtuálních světů, které vytváříme svými programy
- v počítačové grafice modelujeme pomocí **geometrických objektů**
- většina API poskytuje naprosté minimum geometrických primitiv, složitější objekty z nich musí vytvořit uživatel
  (OpenGL pouze trojúhelník, čára a bod)
- **Složené objekty v modelech a vazby mezi nimi musíme nějak reprezentovat SAMI**
- Objekty se typicky načtou z externích souborů pomocí knihoven, například knihovna ASSIMP: http://assimp.sourceforge.net/

  Příklad použití:
  http://www.lighthouse3d.com/cg-topics/code-samples/importing-3d-models-with-assimp/
  (Pozn. Knihov¨na je pomalá v debug režimu na windows)

Modelování

geometrický objekt (dům)

lze rozdělit na nezávislé **segmenty** (detaily) – např. okno, komín,...

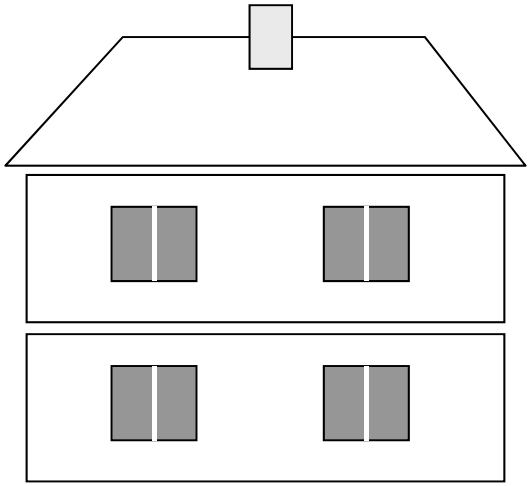


Nabízí se 3 způsoby reprezentace:

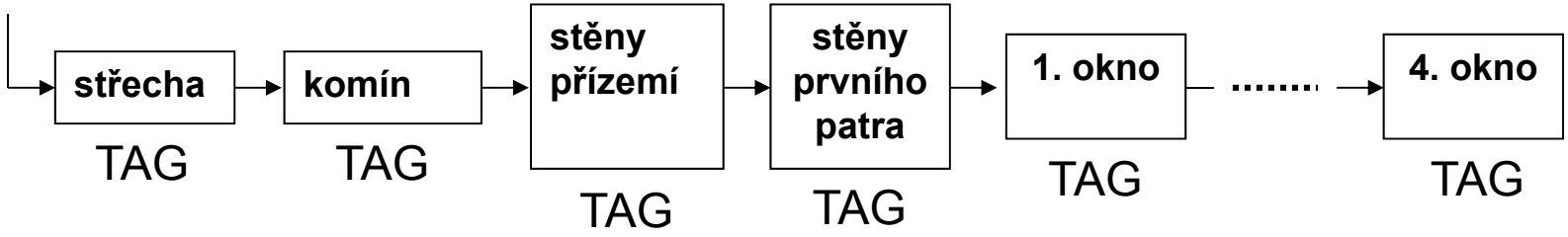
1. Posloupností segmentů
   (lineární datová struktura)
2. Hierarchický model – naivní strom
3. Hierarchický model – orientovaný acyklický graf

1. Lineární (nehierarchická) reprezentace

- objekt je reprezentován **posloupností segmentů** (lineární datová struktura)
- **segment** obsahuje definici grafických elementů a jejich atributů i transformací (př. levé okno v přízemí)

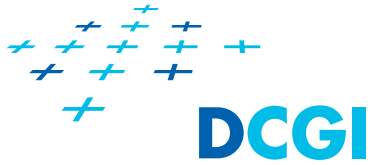


dům



- neexistuje informace o vzájemných vazbách objektů
  => snadno se manipuluje se segmenty,
  => ale těžko s komplexními strukturami (celé patro)

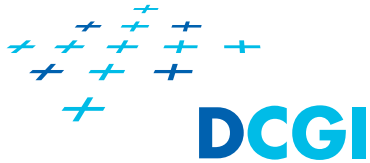
1. Lineární (nehierarchická) reprezentace



- Kdy použít lineární strukturu v programu?
- Pouze pro jednoduché objekty

```
void house(void) {        /* model celého domu */
   roof();                /* střešní segment */
   chimney();
   groundFloorWalls();
   firstFloorWalls();
   firstWindow();
   …
   fourthWindow();
}
```

2. a 3. Hierarchická reprezentace



- Vztahy mezi objekty modelu reprezentujeme **grafem,** kterému říkáme **GRAF SCÉNY ("scene graph")**

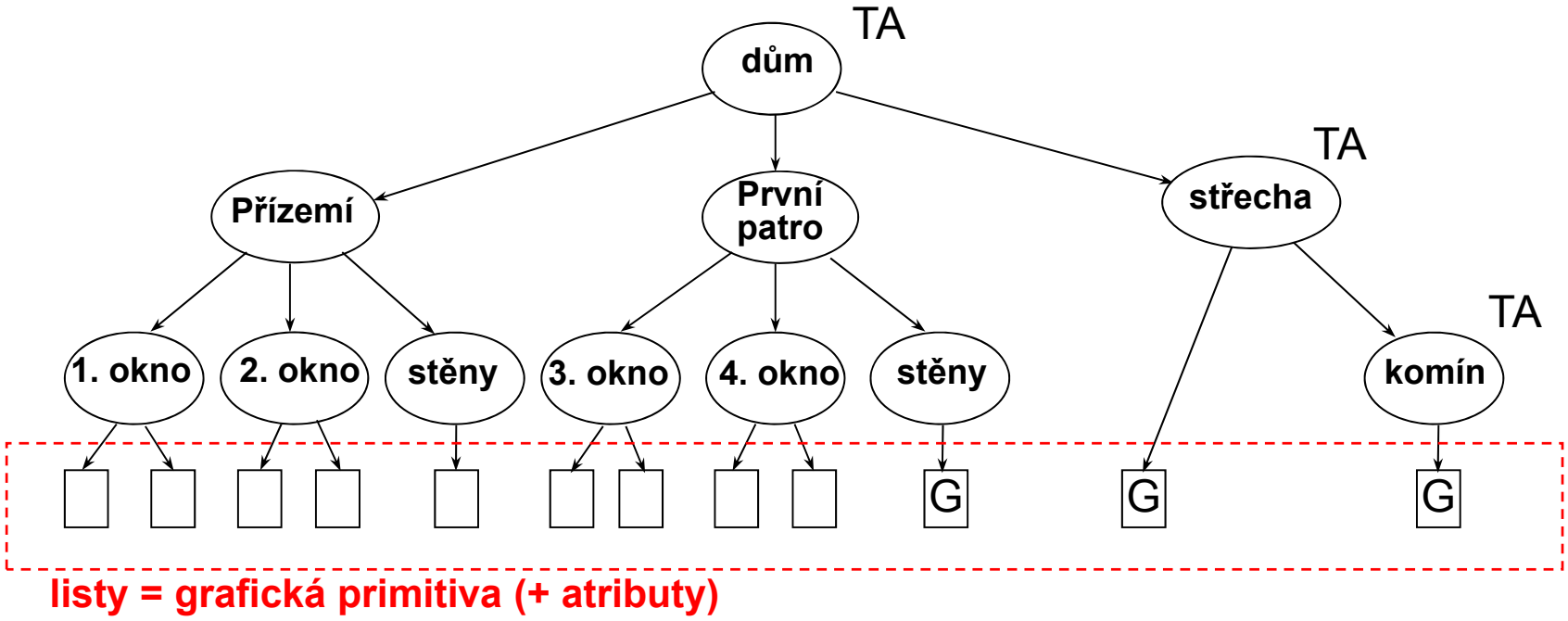
- Graf scény není konstruktem OpenGL, strukturu scény si v programu vytváříme sami.

- V grafu uložena (v závorce uvedeno ve které části grafu)
  - Geometrie G (listy),
  - transformace T
    atributy A (vnitřní uzly, hrany, listy)
  - odkazy na součásti (hrany)

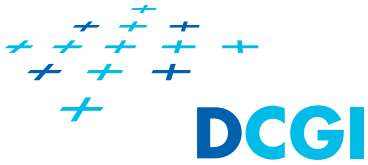
- Vykreslení objektu = systematické procházení (traverzování) grafu
  - do hloubky či do šířky
  - v programu zvolit jeden způsob traverzování a neměnit ho
  - atributy se dědí, nebo jsou v uzlu předefinovány

2. Hierarchický model – naivní strom

- Jednoduchý graf je **kořenový strom** (orientovaný graf bez uzavřených cest a smyček, každý uzel kromě kořene má předchůdce-rodiče)
- **Vnitřní uzly** reprezentují nadřazené segmenty (detaily)
- **Listy** reprezentují grafická primitiva – opakují se



2. Hierarchický model – naivní strom



Jak zakódovat strom v programu?

- **vnitřní uzly** – ukládají transformace, atributy a odkazy na následníky
- **listy** – reprezentovány kreslícími funkcemi (geometrie)

Př. Napevno v programu:

```
void house(void) {          /* model celého domu */
    // (posunutí do počátku)
    groundFloor(); // včetně transformace pro přízemí
    firstFloor();  // včetně transformace pro 1. patro
    roof();        // včetně transformace pro střechu
}
```

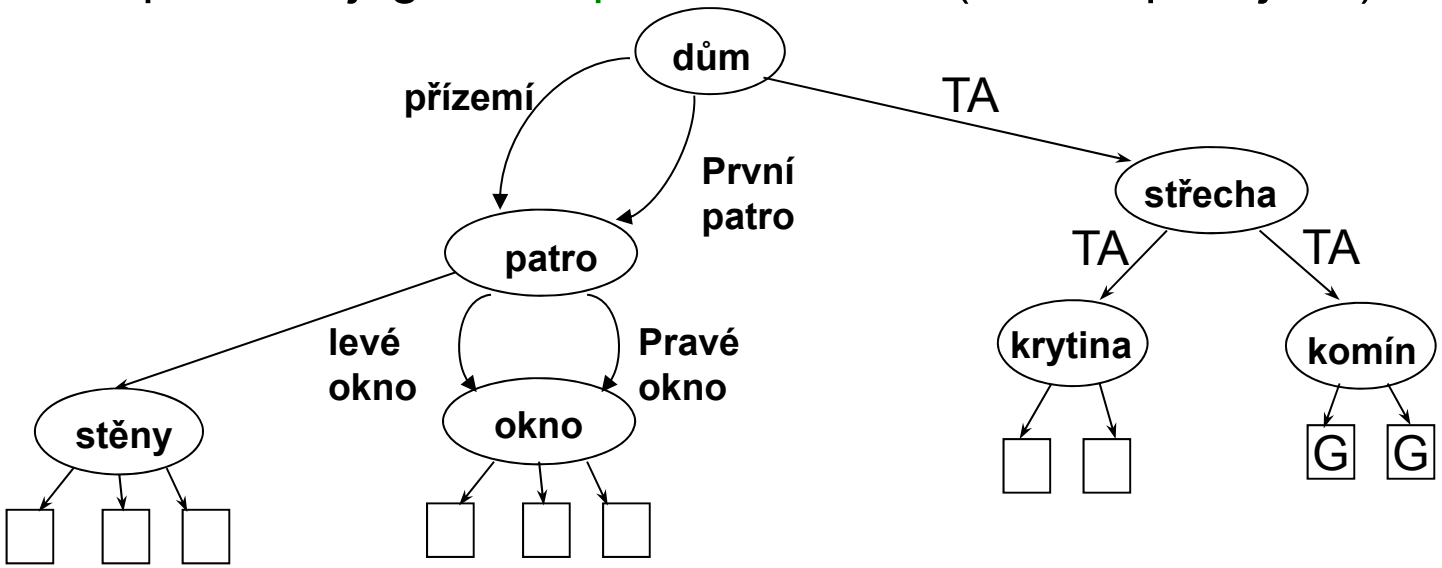
- Problémy reprezentace s použitím stromu:
  - opakuje se geometrie (listy)
    i stejné tvary (detaily) v různých polohách (vnitřní uzly)
  - transformace kódovány napevno v uzlech – obtížná manipulace
    (jeden celek (uzel) nejlze použít vícekrát na různých místech)

3. Orientovaný acyklický graf (DAG)

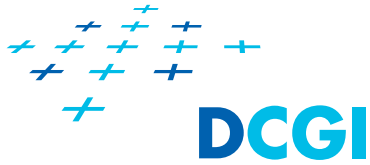
Též Collapsed tree (VRML)

Opakující se detaily se uloží jen jednou - (př. jedna definice komínu)

- **Vnitřní uzly –** reprezentují detaily dané úrovně modelu,
  – optimálně v počátku [0,0,0] (ale neopakují se)
- **Hrany –** reprezentují vazby (relace rodič-potomek)
  – nesou transformace, atributy
  (inkrementální změnu na cestě od rodiče k potomkovi)
- **Listy –** reprezentují grafická primitiva (ale neopakují se)



3. Orientovaný acyklický graf (DAG)



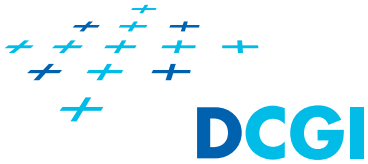
Jak zakódovat DAG v programu?

- **Vnitřní uzly** – ukládají odkazy na potomky v základní poloze
  případně „normalizační“ transformaci
  pozor na normály, viz. další lekce
- **Hrany** – transformace, atributy
- **Listy** – reprezentovány kreslícími funkcemi

Př. :

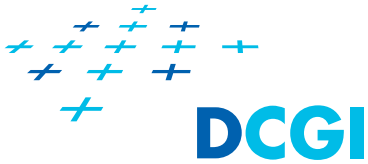
```
void house(void) {          /* model celého domu */
    /* nastav transformace pro přízemí */
    floor(); // v základní poloze
    /* nastav transformace pro první patro*/
    floor();                ;
    /* nastav transformace pro střechu */
    roof();
}
```

Reprezentace grafu



- specializovanou knihovnou – „grafem v paměti“ (OpenSceneGraph, OpenSG, OpenVRML, dcgiSceneGraph)
  - pro graf scény zvláštní třída a funkce
  - při vykreslení se traverzuje strom (transformují, volají kreslící procedury)
- skrytě
  - pomocí vlastní datové struktury reprezentující graf scény
  - pořadím volání vykreslovacích procedur
  - může být méně obecná a proto rychlejší

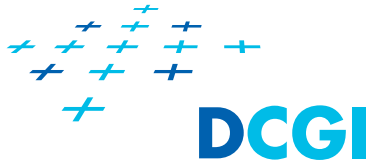
Hierarchická reprezentace - shrnutí



Výhody hierarchického modelování (DAG)

- hierarchické modely odrážejí strukturu objektů
- dobře se vytvářejí (top-down nebo bottom-up)
- dobře se editují a animují, dobře se manipuluje s podstrukturami (transformace jsou součástí hran)
- paměťově úsporná reprezentace modelu (bez redundance)

Atributy v grafech scény

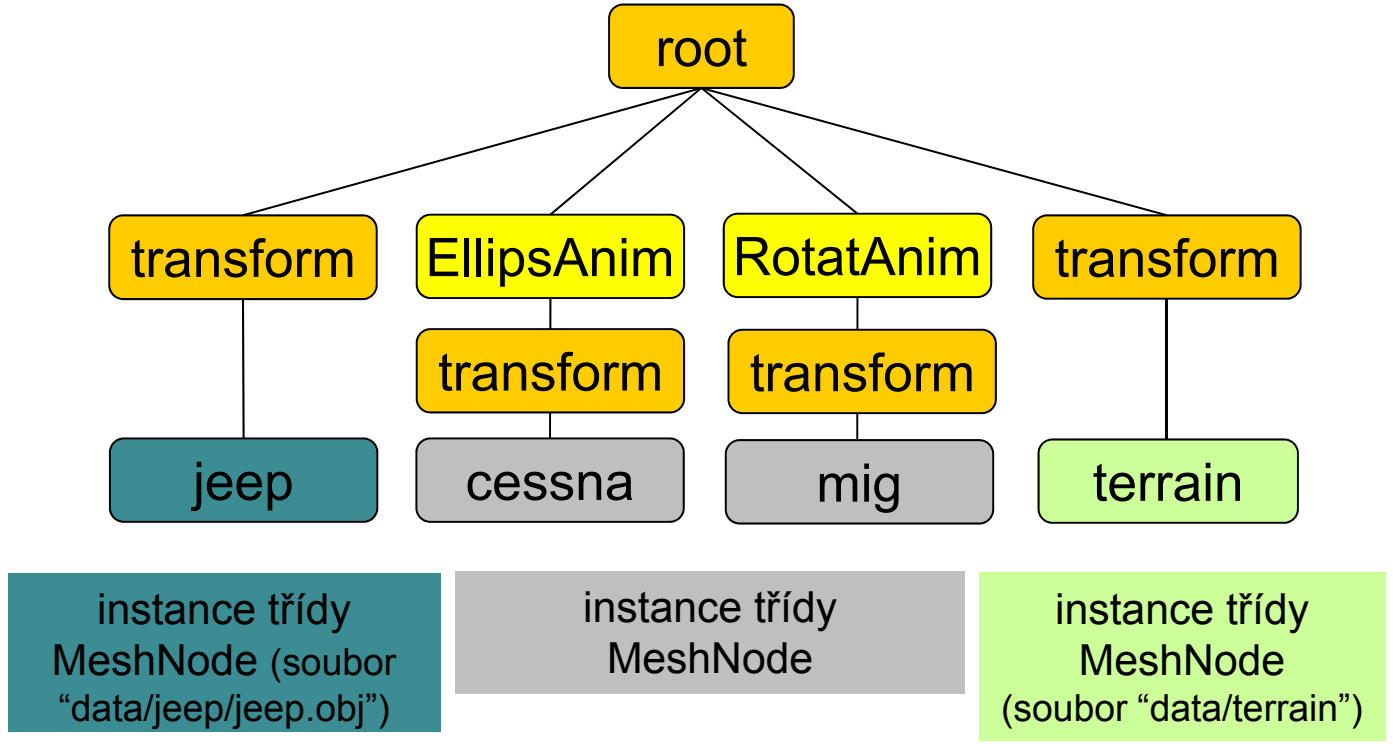


POZOR na atributy při traverzování grafu scény

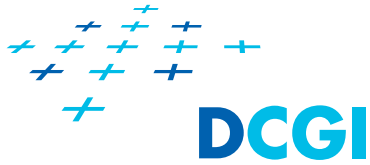
- atributy jsou stavové proměnné – proto zůstávají nastavené, dokud je nenastavíme jinak
  - to platí pro některé implementace
  - výsledek pak může záviset na pořadí traverzace
  - v našich příkladech to neplatí – na pořadí traverzace nezáleží
- pro animaci grafu (dynamické nastavování transformací) se používá několik postupů
- probereme jeden z možných způsobů, použitý pro náš graf scény

Vlastní graf scény

Jednoduchý graf scény pro účely předmětu PGR, viz cvičení.



Vlastní graf scény



Typy uzlů

- **SceneNode** – bázový typ
  – obsahuje vektor odkazů na potomky

  scene

  - **TransformNode** – ukládá transformační matici
    – vnitřní uzel grafu

    transform

  - **RotationAnimNode** – rotace kolem osy pro animaci
    – vnitřní uzel grafu

    RotatAnim

  - **MeshNode** – ukládá vykreslitelnou geometrii
    – list grafu

    terrain

    - TerrainNode – terén vygenerovaný programem terragen
    - OBJNode – objekt ve formátu .obj

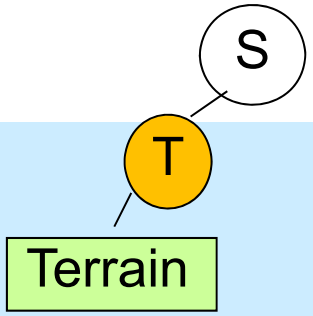
      OBJ

- **Metody** – update / pro animace
  – addChildNode/deleteChildNode - správa hierarchie

Vlastní graf scény - dcgiSceneGraph

DCGI

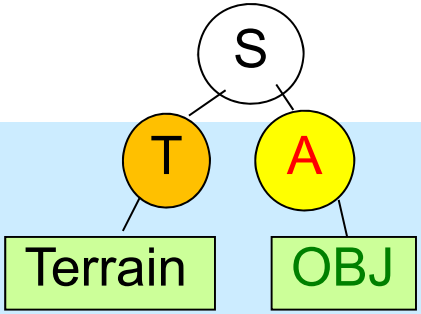
Ruční sestavení grafu scény 1




```
void init() {
   root_node = new SceneNode( "root" );

   // terrain transform node
   CTransformNode* transform_p =
                     new CTransformNode( root_node );
   // terrain node
   CTerrainNode *terrain_p =
                     new CTerrainNode( transform_p );
   terrain_p->load("terrain");

...
```

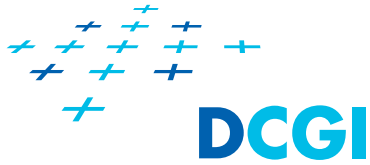
Vlastní graf scény - dcgiSceneGraph

Ruční sestavení grafu scény 2




```
....
....
// Model bude rotovat kolem osy procházející počátkem
souřadnic
 RotationAnimNode * prot
                 = new RotationAnimNode("py-rot2", root_node);
 prot->setAxis(Vec3f(1, 0, 0));
 prot->setSpeed(M_PI / 10);

  // Nyní nahraj model z formátu OBJ a učiň ho potomkem
uzlu s definicí
  // animované rotace, rodič je správně nastaven
  MeshNode *mesh = new MeshNode("cessna.obj", prot);
  mesh->loadMesh(); // load the data
  ...
```

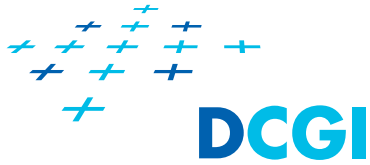
Vlastní graf scény - dcgiSceneGraph



Vykreslení grafu scény

```
void functionDraw(void) {
  // vymaž obrazovku
  glClearColor(0.3f, 0.3f, 0.3f, 1.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // nastavení matice projekce = perspektivy
  Matrix4f projection = Matrix4f::Perspective(M_PI / 4,
                        aspect_ratio, 0.1f, 100);
  // nastavení modelovací a pohledovou matici
  Matrix4f view = Matrix4f::Identity(); Matrix4f model=
  SendMatricesToShaders(projection, view, model)
  // vykresli celou scénu = traverzuj graf scény
  rootNode_p->draw();
}
....    glutSwapBuffers();
```

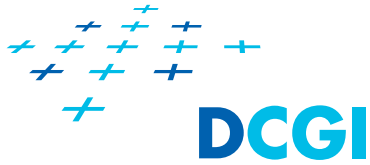
Vlastní graf scény - dcgiSceneGraph



Jednoduchá aktualizace transformace pomocí časovače

```
void FuncTimerCallback(int)
{
  double timed = 0.001 *
   (double)glutGet(GLUT_ELAPSED_TIME); // v milisecundách

   // animuj scénu pro zadaný čas
   if (root_node)
      root_node->update(timed);

   // nastav další volání časovače pro příště
   glutTimerFunc(33, FuncTimerCallback, 0);
   // vyvolej překreslení obrazu
   glutPostRedisplay();
}
```

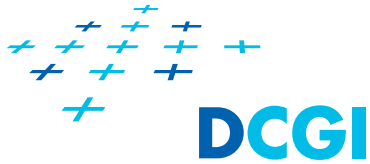
Vlastní graf scény - dcgiSceneGraph



Příklad: inkrementální rotace, viz třída `RotationAnimNode{ …`

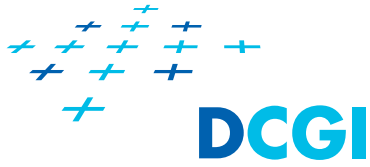
```
TransformNode * ptrans2 = new TransformNode("py-trans2",
root_node);
  ptrans2->translate(Vec3f(0.5, 0, -3));
  ptrans2->scale(Vec3f(0.05, 0.05, 0.05));

  RotationAnimNode * prot2 = new RotationAnimNode(
                                 "py-rot2", ptrans2);
  prot2->setAxis(Vec3f(1, 0, 0));
  prot2->setSpeed(M_PI / 10); // v radiánech za sekundu
  MeshNode * mesh1 = new MeshNode("data/cessna.obj",prot2);
  mesh1->loadMesh();
```

Skládání transformací v grafu scény



- Transformace se při postupu shora dolů skládají zleva doprava
  a) Vyhodnocují se během vykreslování
     výhodné použít zásobník: dolů – push / nahoru – pop
  b) Vyhodnotí se v samostatném průchodu – update()
     složené transformace se uloží v hranách
     (takto je to v dcgiSceneGraphu)

Homogenní souřadnice



- bod je reprezentován svými souřadnicemi $\mathbf{P} = [x, y, z]^t$
  v poč. grafice používáme **homogenní souřadnice** $\mathbf{P} = [x, y, z, w]^t$

- Kartézské souřadnice $\Rightarrow$ homogenní souřadnice

  $\mathbf{P} = [x, y, z]^t \Rightarrow$ zvolit $w \neq 0 \Rightarrow \mathbf{P} = [w.x, w.y, w.z, w]^t$

  příklad: bod v kartézských souřadnicích $[2, 3, 5]^t$.
  Jaké jsou jeho homogenní souřadnice?

  Pro body volíme w = 1

  $[w.2, w.3, w.5, w]^t$ a $w \neq 0 \Rightarrow$ např. $[2, 3, 5, 1]^t$, $[4, 6, 10, 2]^t$, atd.

- homogenní souřadnice $\Rightarrow$ Kartézské souřadnice

  $\mathbf{P} = [x, y, z, w]^t \Rightarrow \mathbf{P} = [x/w, y/w, z/w]^t \quad w \neq 0, \quad w \in R$

  příklad : bod v homogenních souřadnicích $[9, 3, 12, 3]^t$.
  Jaké jsou jeho kartézské souřadnice?

  ***P** = $[9/3, 3/3, 12/3]^t = [3, 1, 4]^t$*

Homogenní souřadnice ve 2D

Geometrická interpretace ve 2D:

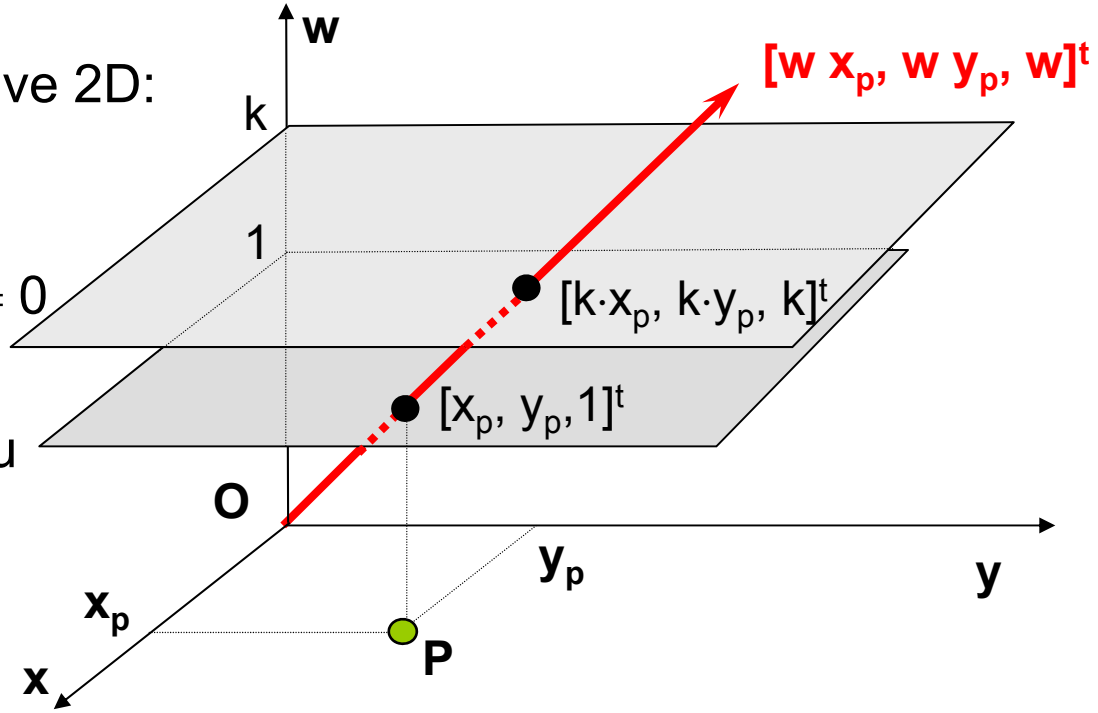
bodům s homogenními souřadnicemi

$\mathbf{P} = [w.x, w.y, w.z, w]^t$, $w \neq 0$

Tvoří přímku bez počátku,

která je celá obrazem bodu

$\mathbf{P} = [x, y, z]^t$



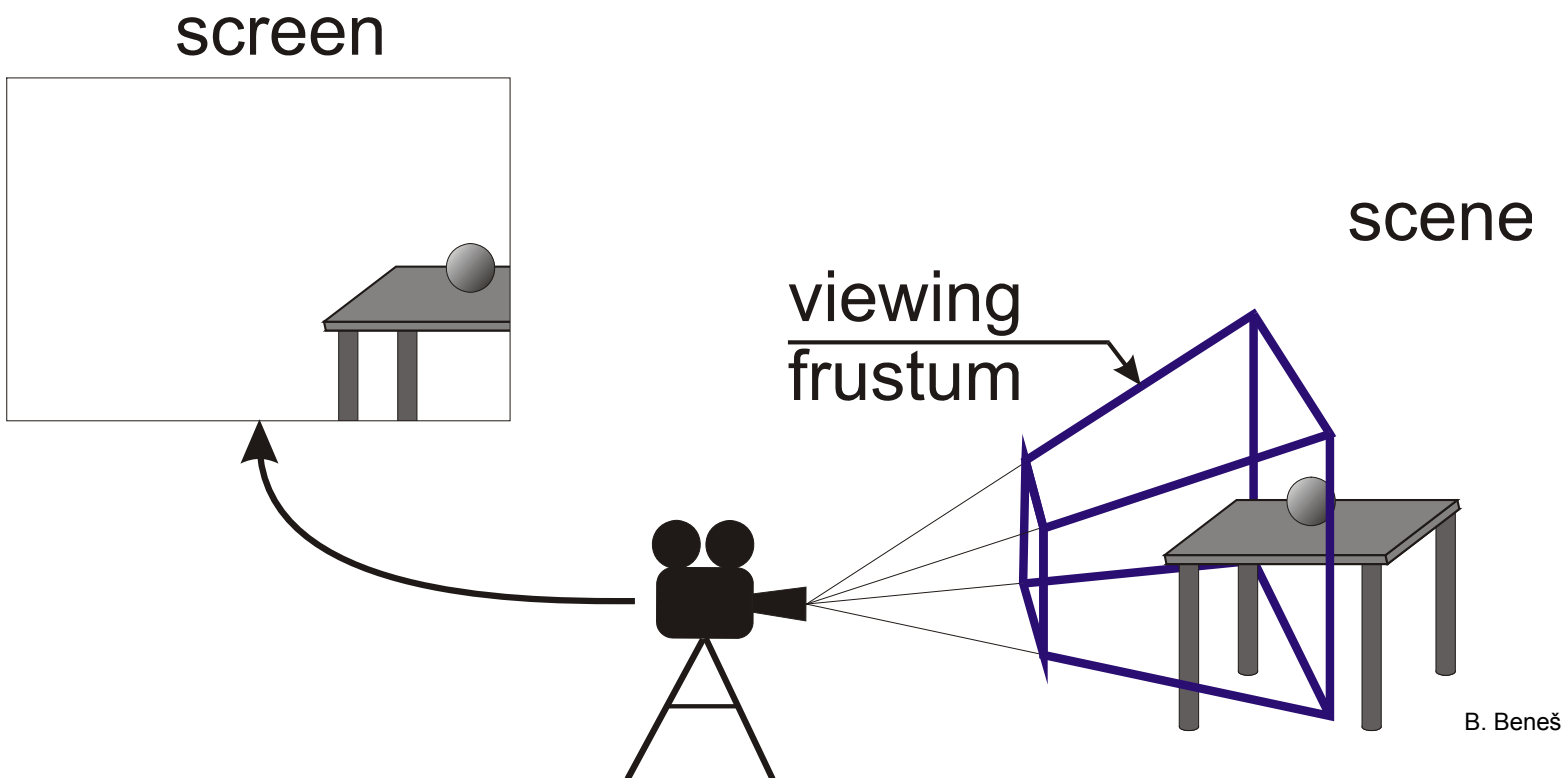
Výhody homogenních souřadnic

- Afinní transformace i projekce lze zapsat jednou maticí (ve 3D maticí 4x4)
- Skládání transformací jako násobení matic
- Kompaktní reprezentace bodů ($w \neq 0$) a vektorů ($w = 0$)

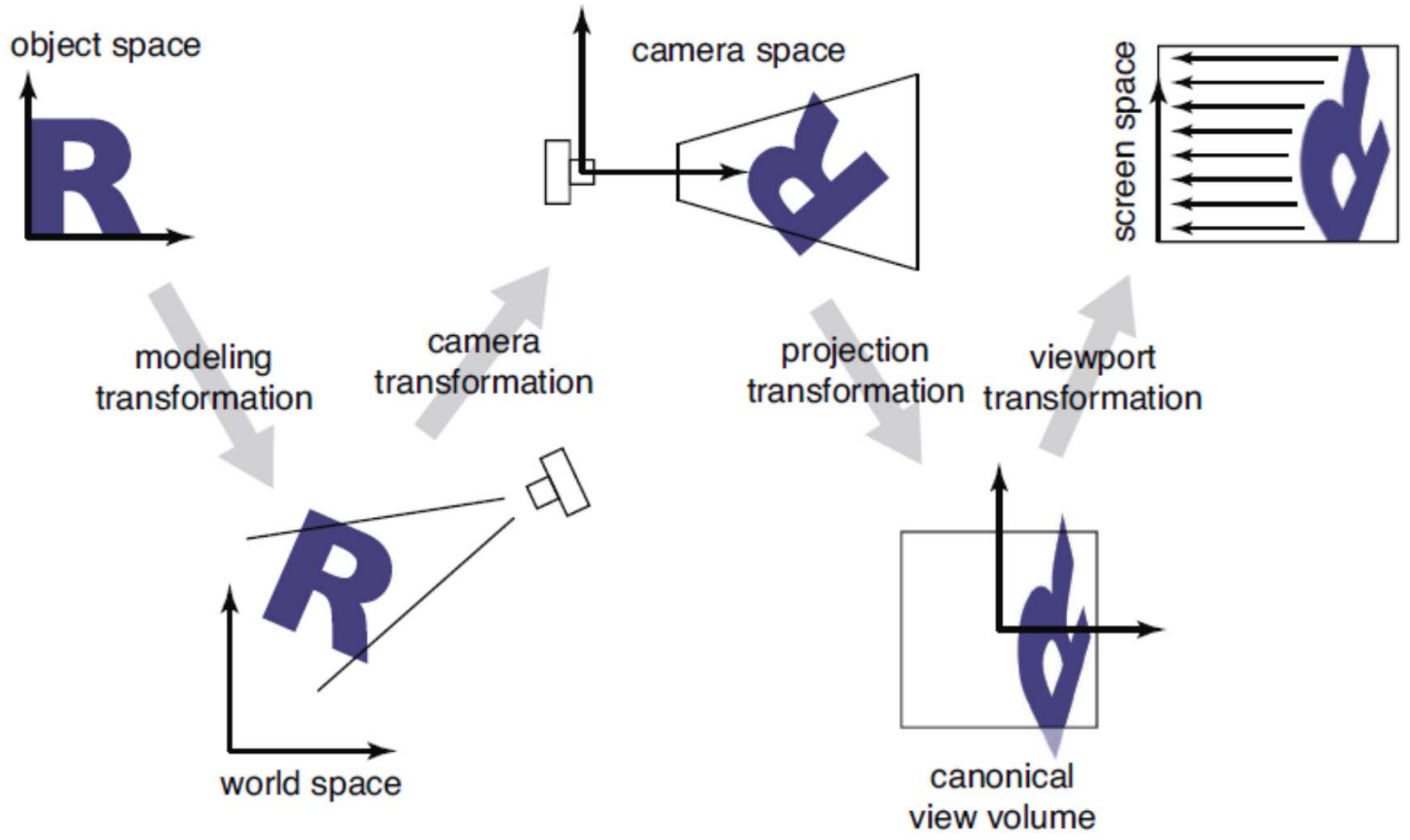
Pro vektory je w = 0

Transformace (2)

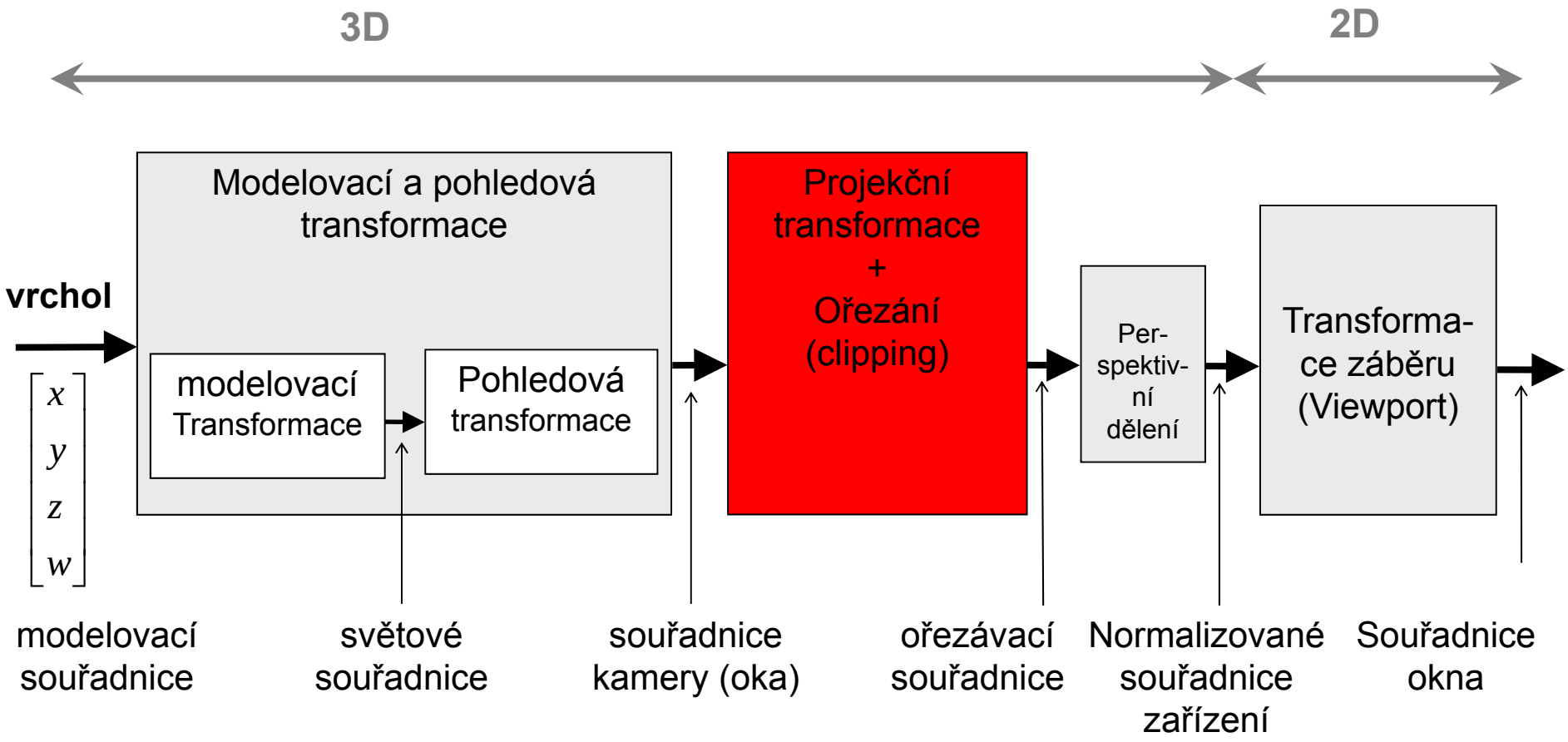
Analogie s fotoaparátem



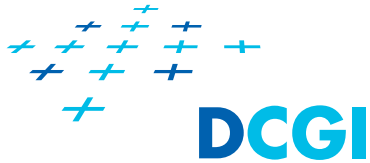
Vizualizace transformací - zopakování



Projekční transformace



Projekční transformace



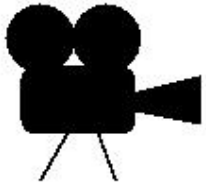
- definuje tvar pohledového objemu (**viewing volume, frustrum)**
- pohledový objem „komolý jehlan“
  - určuje, jak se objekt pomítá na průmětnu (*perspektivní* či *paralelní* projekce)
  - definuje polohu ořezávacích rovin, tj., které objekty či jejich části budou oříznuty (*clipping planes*)
  - zadává se v souřadnicích kamery

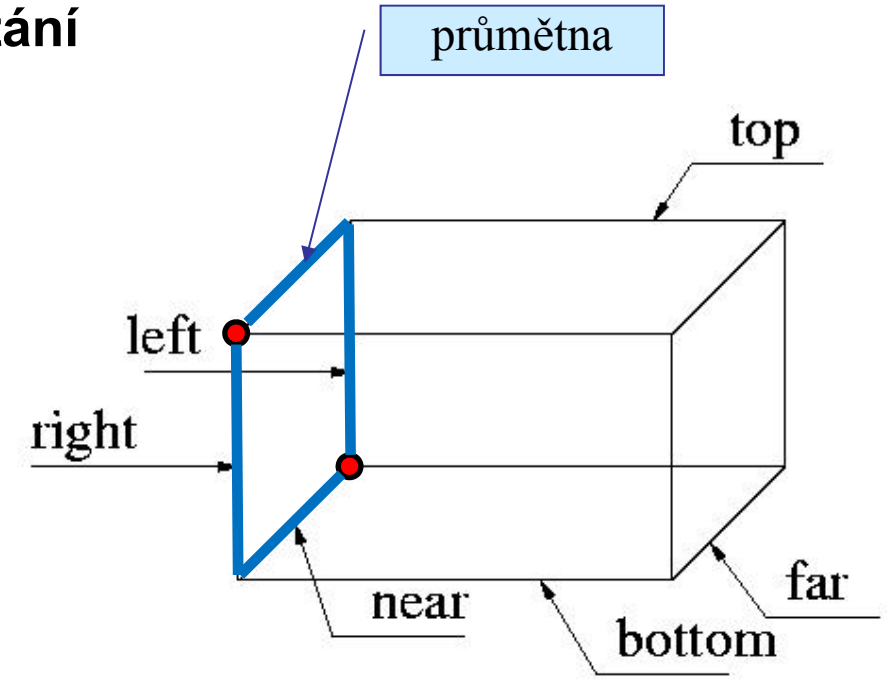
- OpenGL umožňuje jakoukoliv projekci definovanou uživatelem
- projekce NENÍ afinní transformací (projekční matice nemá poslední řádek ve tvaru 0 0 0 1)
- nutné jsou funkce vytvářející matice pro
  - ortografickou projekci (paralelní)
  - perspektivní projekci

Paralelní projekce

```
mat4 glm::ortho(  float left, float right,
                  float bottom, float top,
                  float near, float far);
```

- vytvoří matici pro **paralelní promítání**
- pohledový objem je kvádr
- [left, bottom, *] a [right, top, *] = body na blízké a vzdálené ořezávací rovině (* = near a far),
- jsou mapovány do dolního levého a pravého horního rohu formátu (viewport)

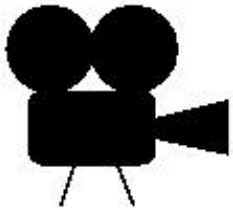


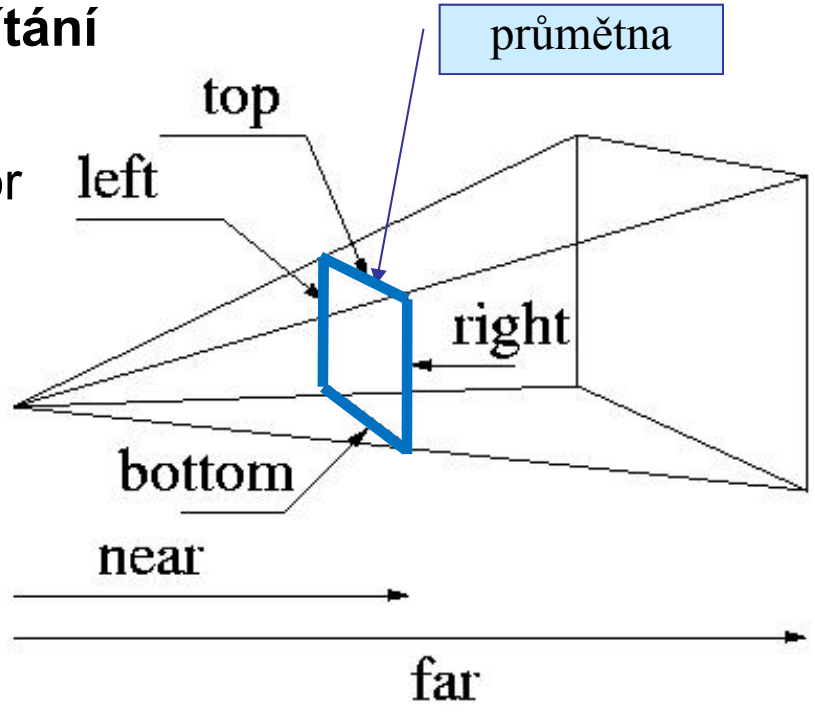


Perspektivní projekce

```
mat4 glm::frustum( float left, float right,
                   float bottom, float top,
                   float near, float far);
```

- vytvoří matici pro **perspektivní promítání**
- pohledový objem je komolý jehlan
- podstavy rovnoběžné, kolmé na vektor pohledu
- menší podstava = průmětna
- objekty blíže zvětšené (zaberou relativně větší část pohledového objemu)

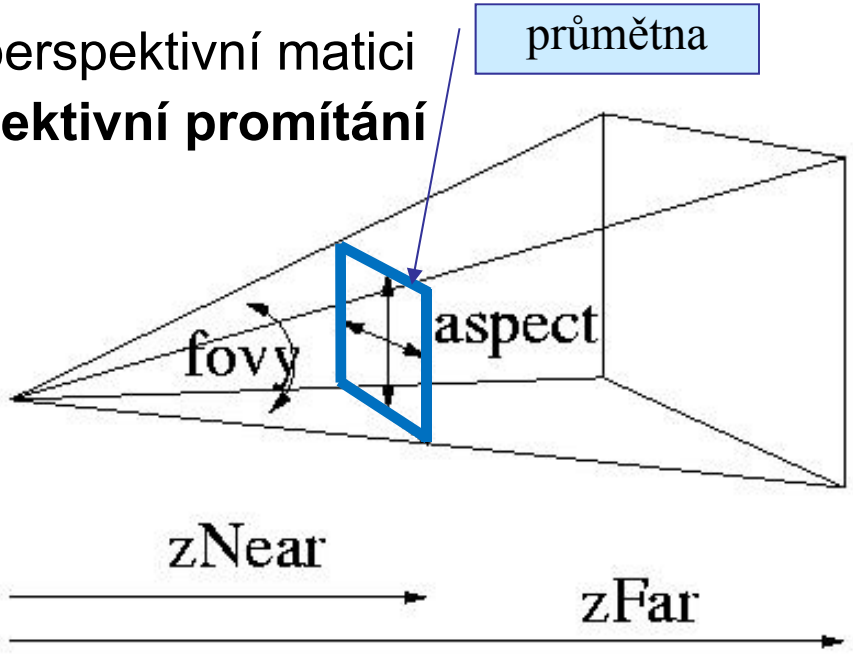




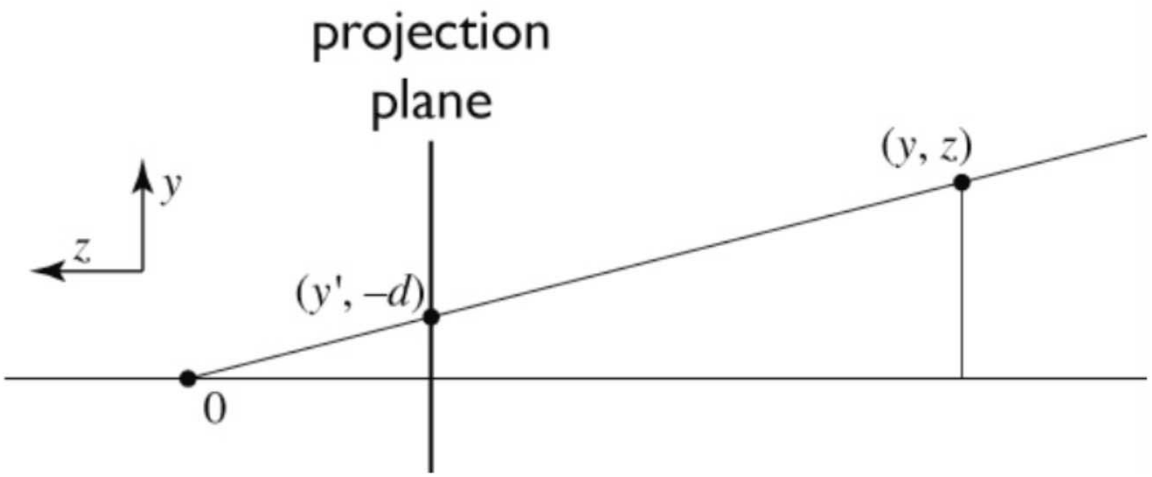
Perspektivní projekce

mat4 **glm::perspective (** float **fovy,** float **aspect,**
float **near,** float **far);**

- jiný způsob definice parametrů pro perspektivní matici
- vytvoří matici pro **symetrické perspektivní promítání**
- **fovy** = úhel záběru
  v rovině x-z,
  - rozsah ⟨0.0 °, 180.0°⟩
  - spolu s near určí h
- **aspect** je poměr šířky ku výšce pohledového objemu (w / h)
  w = aspect * h;
- hodnoty **near** a **far** musí být kladné (near > 0 !!!)



Základ perspektivního zobrazení



similar triangles:

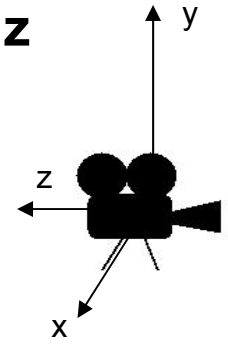
$$\frac{y'}{d} = \frac{y}{-z}$$

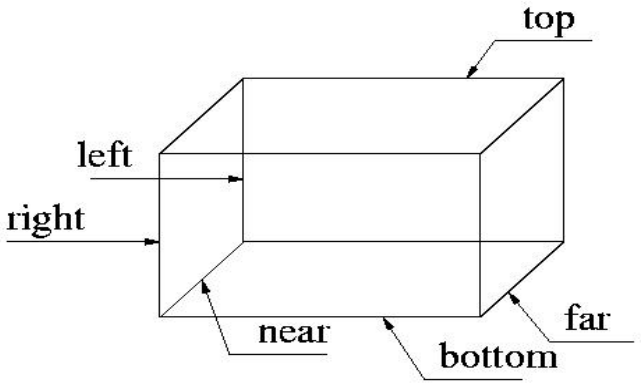
$$y' = -dy/z$$

Odvození paralelní projekční matice (Ortho)

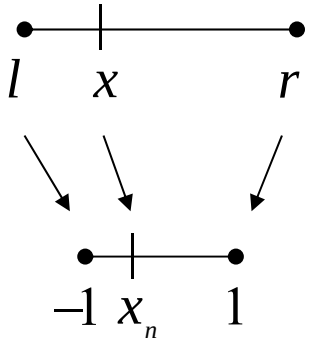
Princip: Zachová x a y. Ignoruje z

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{aligned} x' &= x \\ y' &= y \\ z' &= d \end{aligned}$$





Plus normalizace souřadnic: intervaly $\langle l, r\rangle, \langle t, b\rangle, \langle -n, -f\rangle \Rightarrow \langle -1, 1\rangle^3$



$$\frac{x-l}{r-l} = \frac{x_n-(-1)}{1-(-1)} = \frac{x_n+1}{2}$$

$$x_n = 2\frac{x-l}{r-l} - 1$$

$$x_n = \frac{2x}{r-l} - \frac{r-l+2l}{r-l}$$

$$x_n = \frac{2}{r-l}x - \frac{r+l}{r-l}$$

$$y_n = \frac{2}{t-b}y - \frac{t+b}{t-b}$$

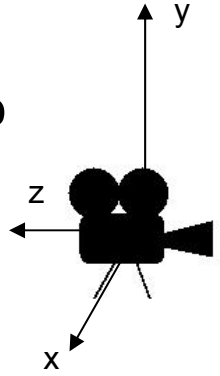
$$z_n = -1$$

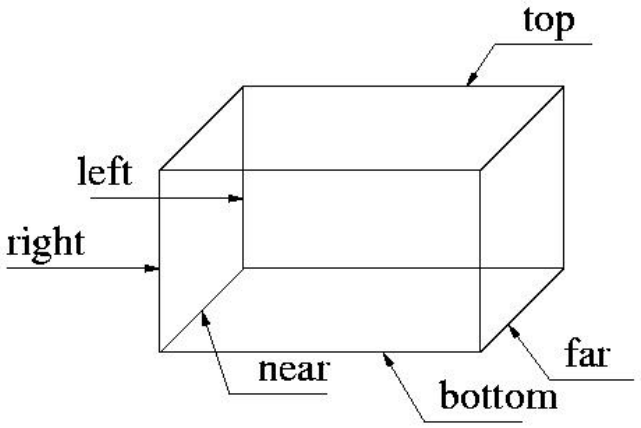
Hloubku ale potřebujeme na určení viditelnosti

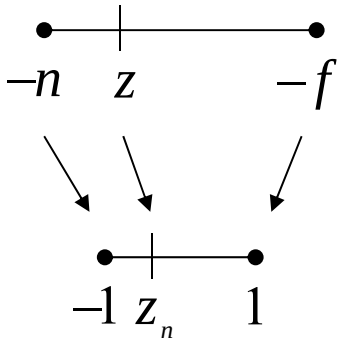
Odvození paralelní projekční matice (Ortho)

Odvození hloubky z_n

- near a far se zapisují v ortho kladně (vzdálenost od oka),
- jsou ale na záporné ose *z*,
- proto ve vzorcích záporné







$$\frac{z-(-n)}{-f-(-n)} = \frac{z_n-(-1)}{1-(-1)}$$

$$-\frac{z+n}{f-n} = \frac{z_n+1}{2}$$

$$z_n = -2\frac{z+n}{f-n} - \frac{f-n}{f-n}$$

$$z_n = \frac{-2z}{f-n} - \frac{f-n+2n}{f-n}$$

$$x_n = \frac{2}{r-l}x - \frac{r+l}{r-l}$$

$$y_n = \frac{2}{t-b}x - \frac{t+b}{t-b}$$

$$z_n = \frac{-2}{f-n}z - \frac{f+n}{f-n}$$

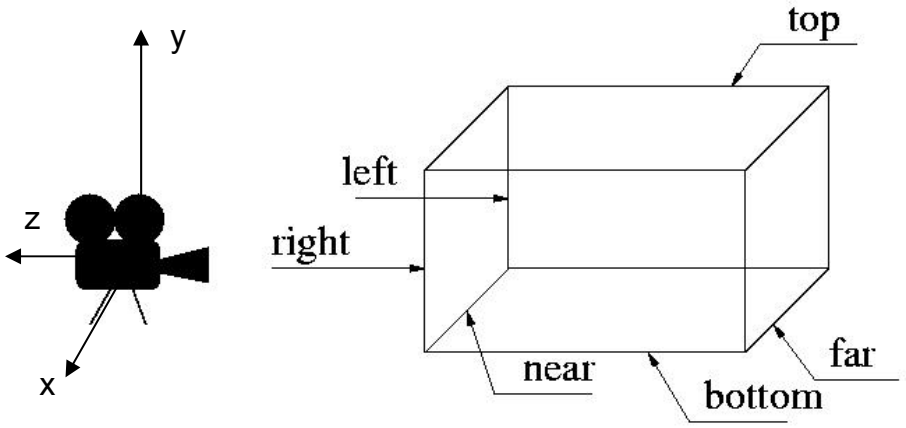
Platí, pokud : $l \neq r,\ t \neq b$ a $n \neq f$

Odvození paralelní projekční matice (Ortho)

Výsledná matice

$$R = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R^{-1} = \begin{pmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$x_n = \frac{2}{r-l}x - \frac{r+l}{r-l}$$

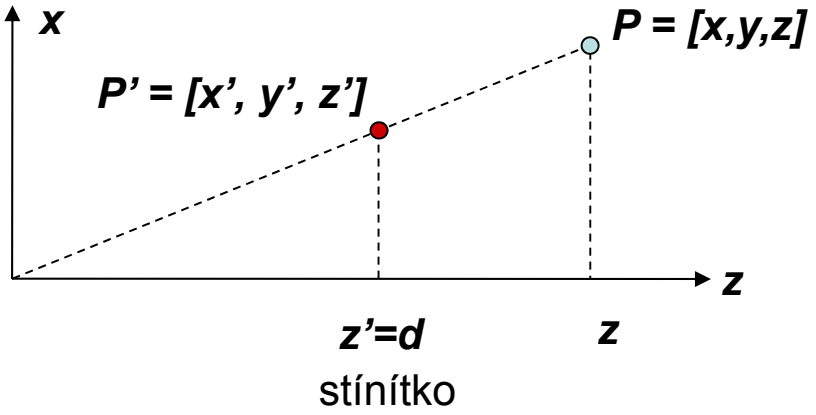
$$y_n = \frac{2}{t-b}x - \frac{t+b}{t-b}$$

$$z_n = \frac{-2}{f-n}z - \frac{f+n}{f-n}$$

Platí, pokud : $l \neq r,\ t \neq b$ a $n \neq f$

Odvození perspektivní projekční matice (Frustum)

Zmenšuje x a y dle z
(promítá je na stínítko)



$$\frac{x'}{d} = \frac{x}{z}, \qquad \frac{y'}{d} = \frac{y}{z}, \qquad z' = d$$

$$\Rightarrow \quad x' = \frac{d}{z}x, \qquad y' = \frac{d}{z}y, \qquad z' = d$$

$$[x', y', z', 1]^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = [x, y, z, z/d]^T$$

- Po transformaci z homogenních do kartézských souřadnic (po dělení w = z / d) dostáváne

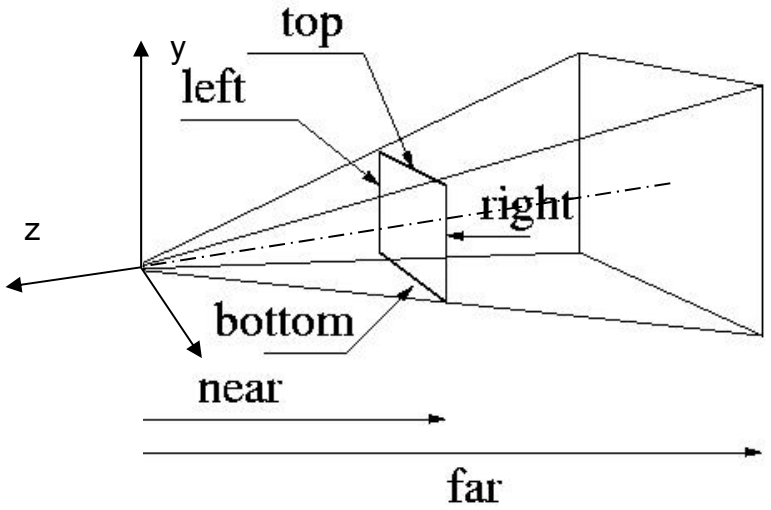
$$P' = [x.d / z, \quad y.d / z, \quad d]^T$$

Odvození perspektivní projekční matice (Frustum)

Odvození X a Y

- Dosadíme d = -n

$$x' = \frac{-n}{z}x, \quad y' = \frac{-n}{z}y$$



Plus normalizace souřadnic:

$$x_n = \frac{2}{r-l}x - \frac{r+l}{r-l} \qquad x_n = \frac{2n}{r-l}\frac{-1}{z}x - \frac{r+l}{r-l} \;\Big/ .(-z) \qquad -zx_n = \frac{2n}{r-l}x + \frac{r+l}{r-l}z$$

$$y_n = \frac{2}{t-b}y - \frac{t+b}{t-b} \qquad y_n = \frac{2n}{t-b}\frac{-1}{z}y - \frac{t+b}{t-b} \;\Big/ .(-z) \qquad -zy_n = \frac{2n}{t-b}y + \frac{t+b}{t-b}z$$

Odvození dále:

$$z_n = \frac{-2nf}{f-n}\frac{1}{(-z)} + \frac{f+n}{f-n} \;\Big/ (-z) \qquad -zz_n = -\frac{f+n}{f-n}z - \frac{2nf}{f-n}$$

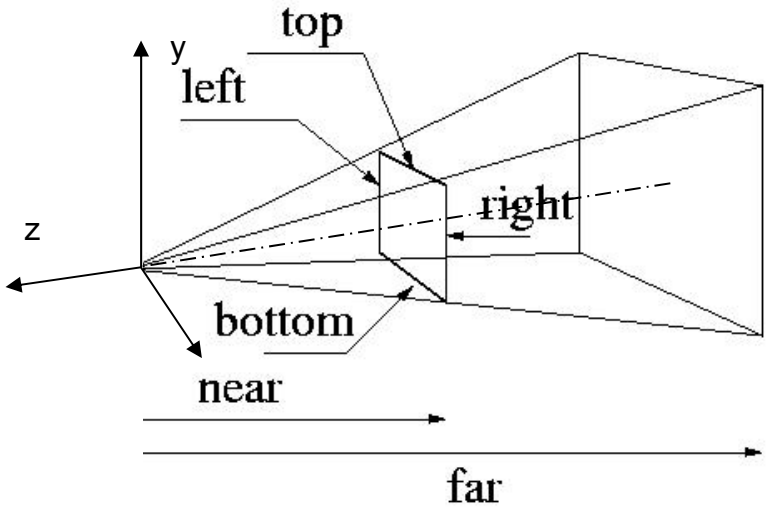
$$w = -z$$

Odvození perspektivní projekční matice (Frustum)

Mapování hloubky

- Interpoluje se 1/z
- Proto ve tvaru

$$z_n = \frac{A}{z} + B$$



Plus normalizace souřadnice *z*

- Dosadíme -n → -1, -f → 1

$$-1 = \frac{A}{-n} + B \quad \text{a} \quad 1 = \frac{A}{-f} + B$$

$$A = \frac{2nf}{f-n} \qquad B = \frac{f+n}{f-n}$$

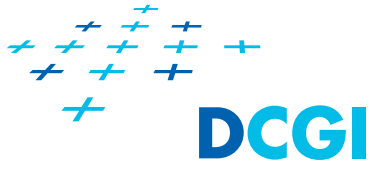
Odvození A a B na dalším slajdu

$$z_n = \frac{-2nf}{f-n}\frac{1}{(-z)} + \frac{f+n}{f-n}$$

$.(-z)$

$$-zz_n = -\frac{f+n}{f-n}z - \frac{2nf}{f-n}$$

Odvození perspektivní projekční matice (Frustum)



Odvození A a B: Dosadíme -n → -1, -f → 1

$$z_n = \frac{A}{z} + B$$

$$-1 = \frac{A}{-n} + B \quad /-1 \quad \text{a} \quad 1 = \frac{A}{-f} + B$$

$$1 = \frac{A}{n} - B$$

$$1 = \frac{A}{-f} + B$$

$$2 = \frac{A}{n} + \frac{-A}{f} = A\frac{f-n}{nf}$$

$$A = \frac{2nf}{f-n}$$

$$B = \frac{A}{n} - 1$$

$$A = \frac{2nf}{f-n} \Rightarrow B = \frac{\frac{2nf}{f-n}}{n} - 1 = \frac{2f}{f-n} - 1$$

$$B = \frac{2f}{f-n} - \frac{f-n}{f-n}$$

$$B = \frac{f+n}{f-n}$$

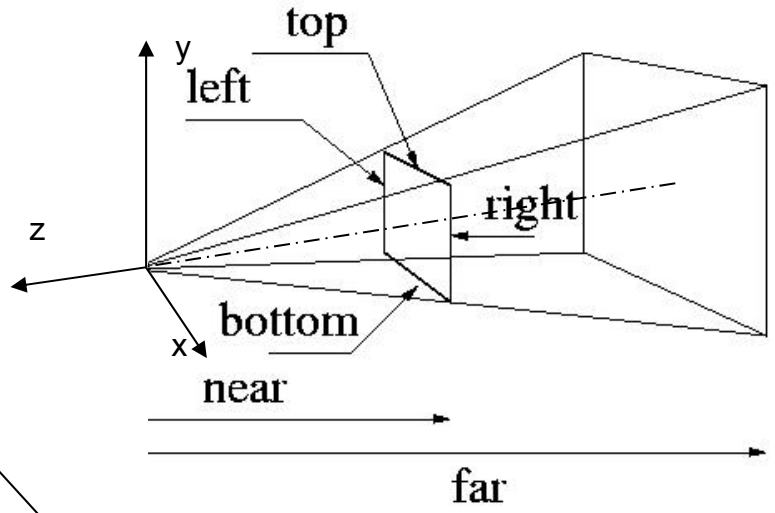
Odvození perspektivní projekční matice (Frustum)

Výsledná matice

Platí pokud : $l \neq r,\ t \neq b$ a $n \neq f$

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

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



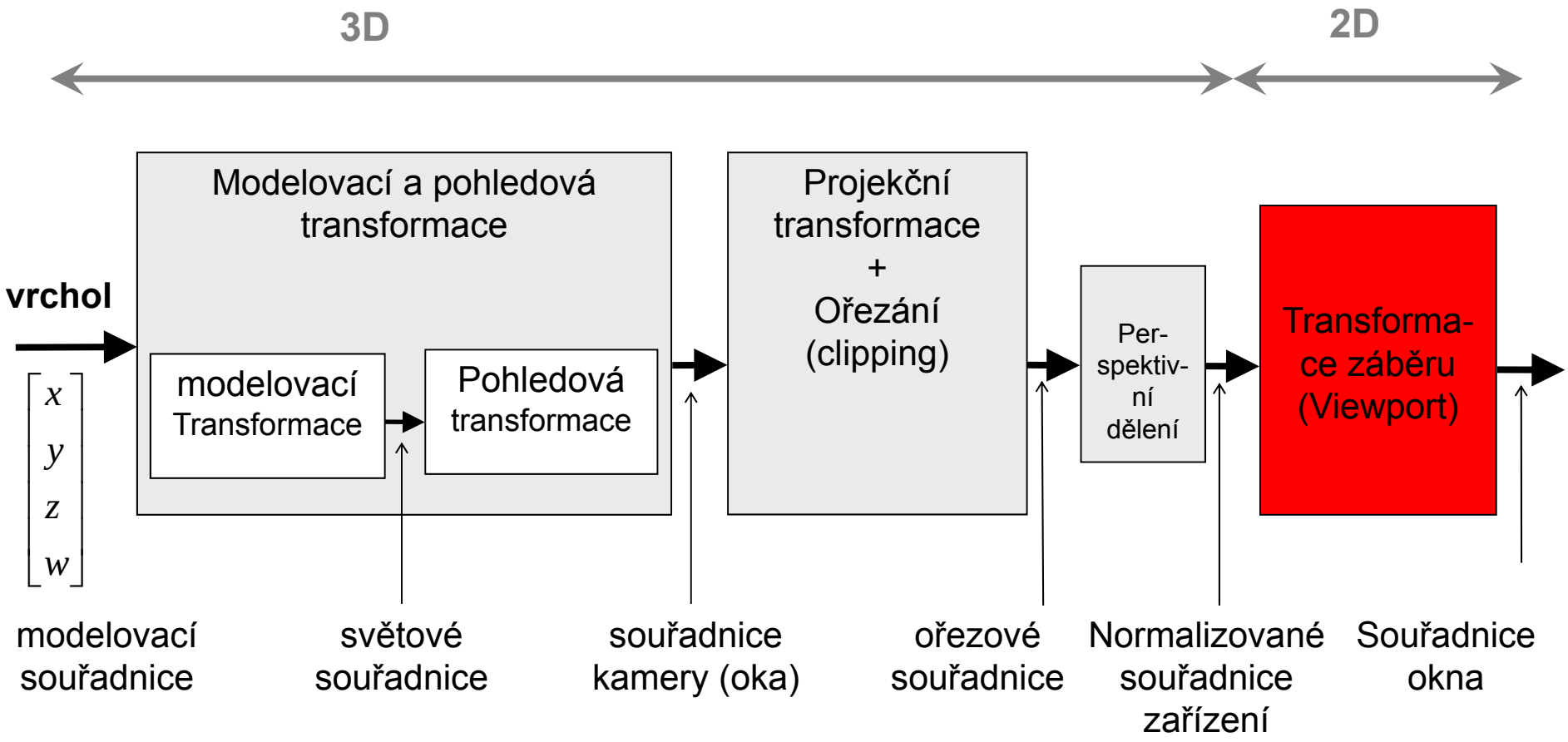
$$-zx_n = \frac{2n}{r-l}x + \frac{r+l}{r-l}z$$

$$-zy_n = \frac{2n}{t-b}y + \frac{t+b}{t-b}z$$

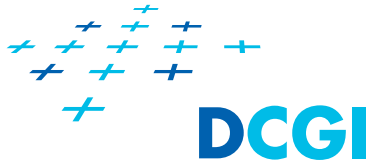
$$-zz_n = -\frac{f+n}{f-n}z - \frac{2nf}{f-n}$$

$$w = -z$$

Transformace záběru



Transformace pracoviště (Viewport)



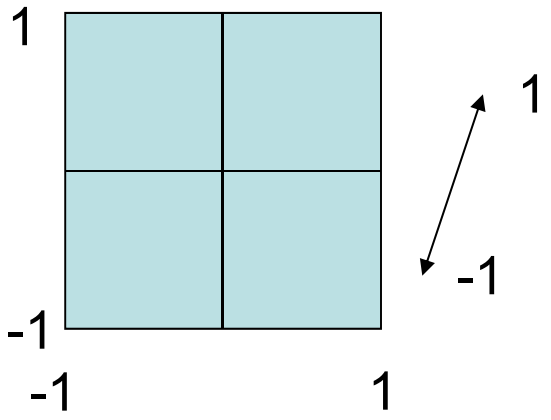
- Formát (viewport) = obdélníková oblast okna, do které se mapuje průmětna (= promítací rovina)

```
void glViewport(
        GLint x, GLint y, GLsizei width, GLsizei height);
```

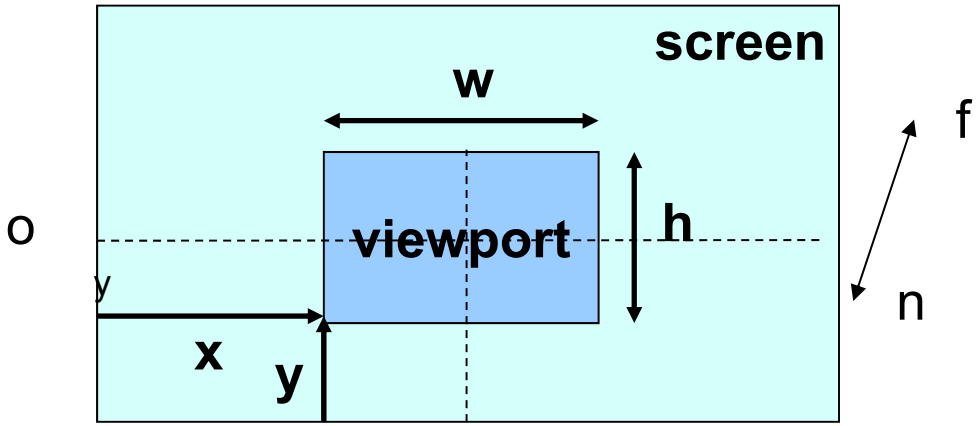
- [**x, y**] je levý dolní roh formátu (viewport),
- **width** a **height** je šířka a výška formátu …v souřadnicích okna – v pixelech
- pro knihovnu GLUT se nastavuje v callback funkci **reshape**
- poměr stran formátu by měl být stejný jako poměr stran průmětny
- jinak zkreslení obrazu
- GLUT: implicitně na celé okno (0,0,w, h)

Transformace pracoviště (Viewport)

Normalizované souřadnice zařízení $[x_d\ y_d\ z_d\ ]^t$



Souřadnice formátu na obrazovce $[x_w\ y_w\ z_w\ ]^t$



$x_w = (w / 2)\ x_d + o_x$

$y_w = (h / 2)\ y_d + o_y$

$z_w = [(f-n) / 2]\ z_d + (n+f) / 2$

z_w na viditelnost (Z-buffer)

$o_x = x + w/2$

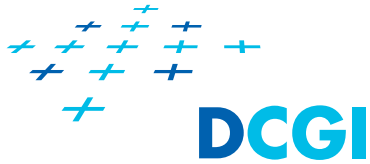
$o_y = y + h/2$

glDepthRange(n, f)

nastavení rozsahu hloubek, clamp(n,f)

n	blízká rovina	0.0
f	vzdálená rovina	1.0

Transformace pracoviště (Viewport)



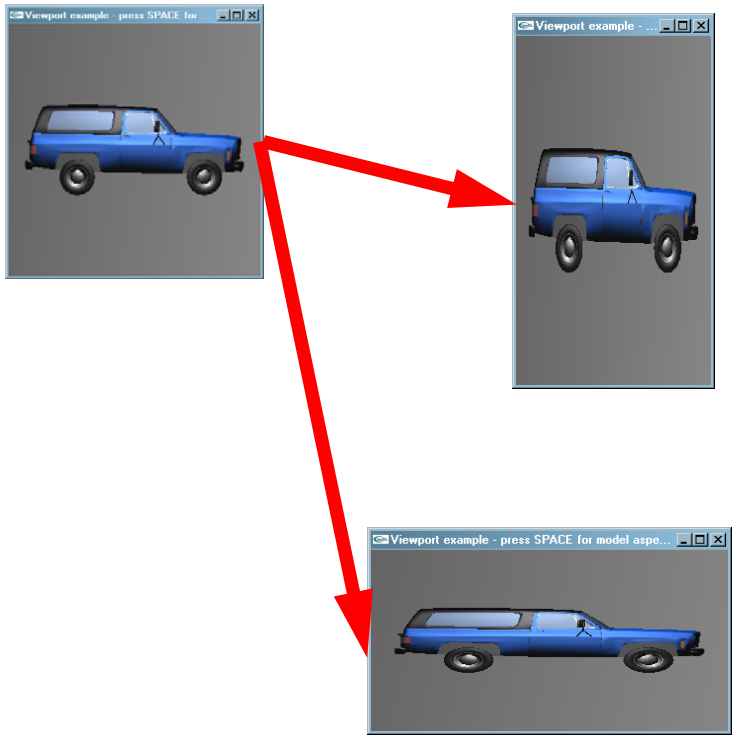
- Dotaz na aktuální formát (transformaci pracoviště)

```
int format[4];
glGetIntegerv( GL_VIEWPORT, format);
```

Význam: x, y, w, h x=format[0], y=format[1], …

Zachování poměru stran obrazu

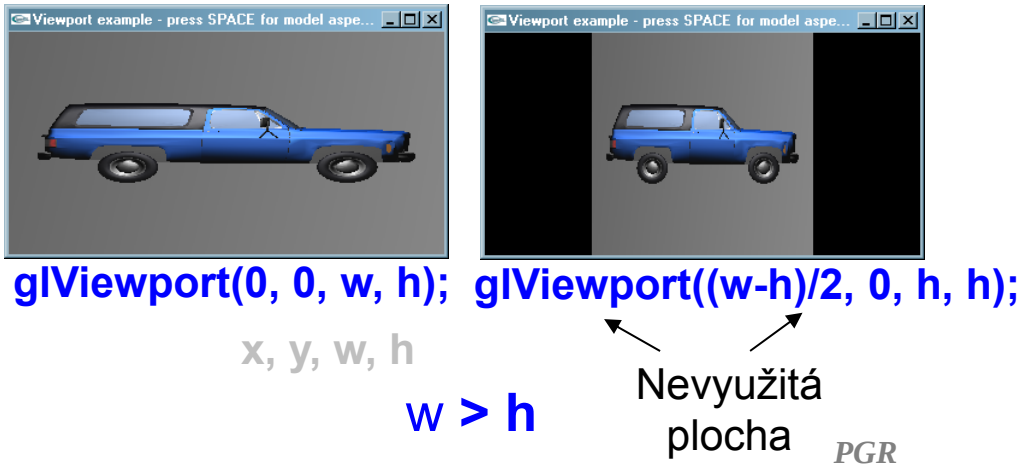
- … aby zůstala kružnice kulatá
- v proceduře reshape(int w, int h)
  registrována jako Callback
  …glutReshapeFunc(reshape);
- Dva způsoby
  a) zmenšením formátu
  b) protažením projekce

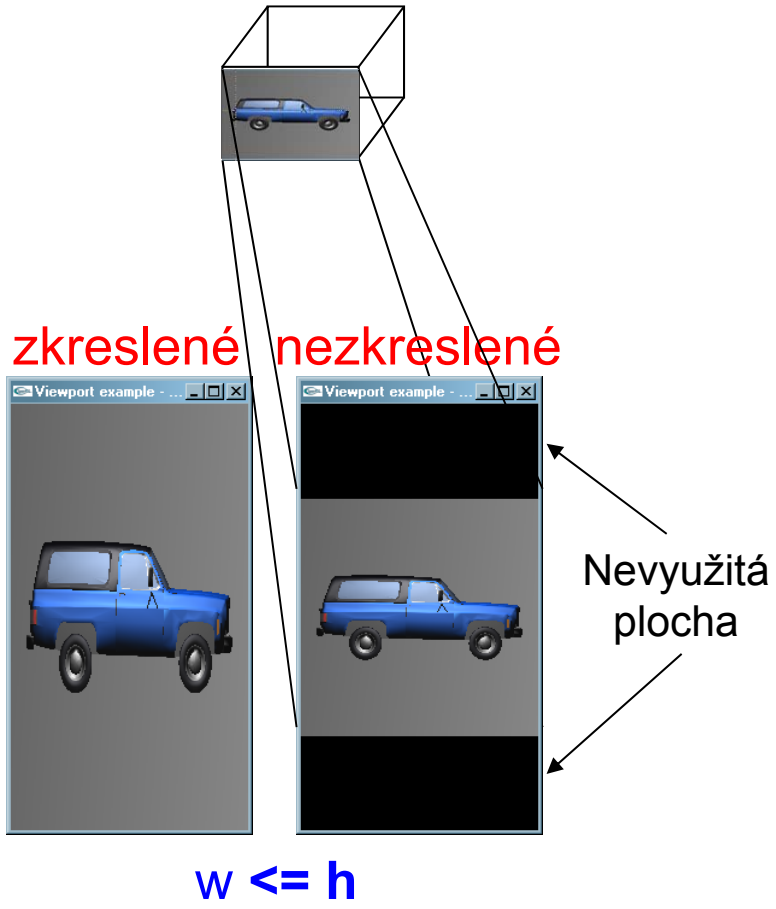


Zachování poměru stran obrazu

a) zmenšením formátu - zbude nevyužitá plocha okna

```
if( w > h )  // okno široké
    glViewport((w-h)/2, 0, h, h);
else         // okno vysoké
    glViewport(0, (h-w)/2, w, w);
```





Zachování poměru stran obrazu

b) protažením projekce (zvětší se pohledový objem)

```
GLfloat aspect = (GLfloat) w / (GLfloat) h;

if( aspect > 1.0 ) // w > h => okno široké
  m = glm::ortho(-1.15*aspect, 1.15*aspect, -1.15, 1.15, -10, 10);
else                          // left, right, bottom, top, near, far
  m = glm::ortho(-1.15, 1.15, -1.15/aspect, 1.15/aspect, -10, 10);
    glm::frustum(-1.15, 1.15, -1.15/aspect, 1.15/aspect, 0.1, 10);
```

Obdobně: `glm::perspective(60.0, aspect, 0.1, 10);`

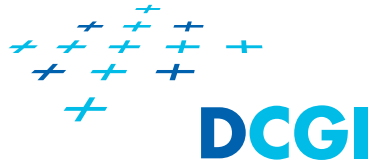
zkreslené

nezkreslené

w > h => okno široké

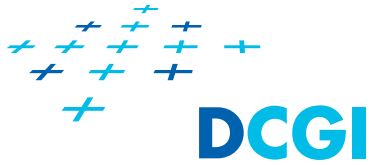
left*aspect

right*aspect



Rotace podle Eulerových úhlů a Gimbal lock

Rotace dle Eulerových úhlů



- Eulerovy úhly = pitch-yaw-roll (výška, kurs, rotace)
- komplexní otočení rozdělíme na otočení dle os X, Y, Z

X

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Rotate(θ, 1, 0, 0)

Y

$$\begin{vmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

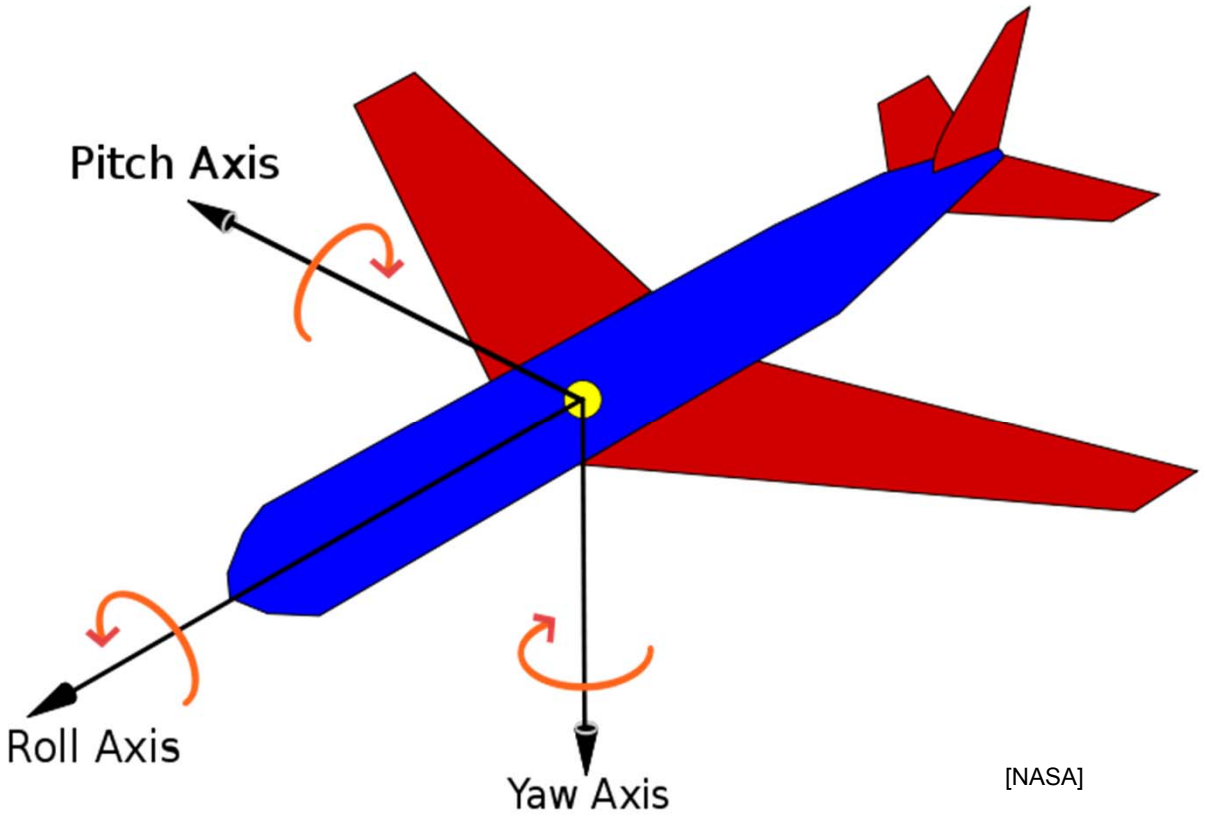
Rotate(θ, 0, 1, 0)

Z

$$\begin{vmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Rotate(θ, 0, 0, 1)

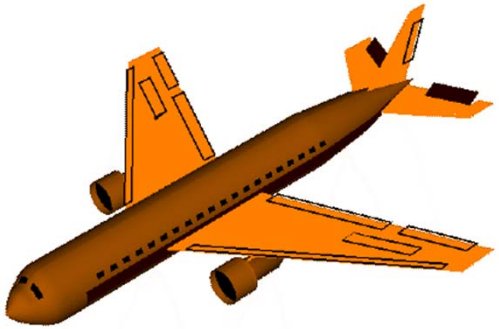
Základní orientace „Pitch-Yaw-Roll“



[NASA]

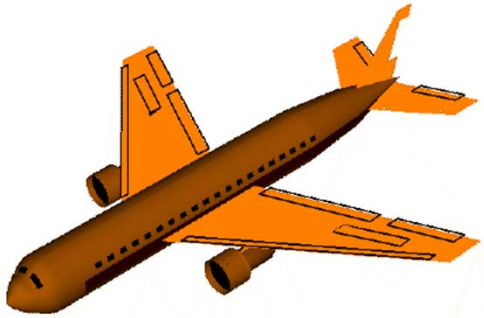
Základní orientace „Pitch-Yaw-Roll“

Pitch



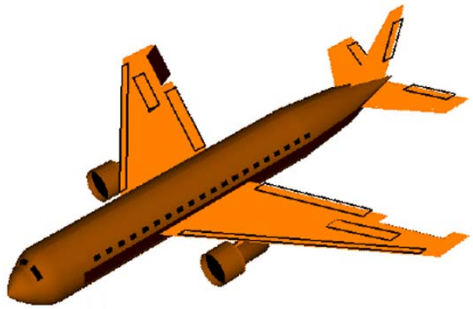
Výškovky

Yaw



Kormidlo

Roll

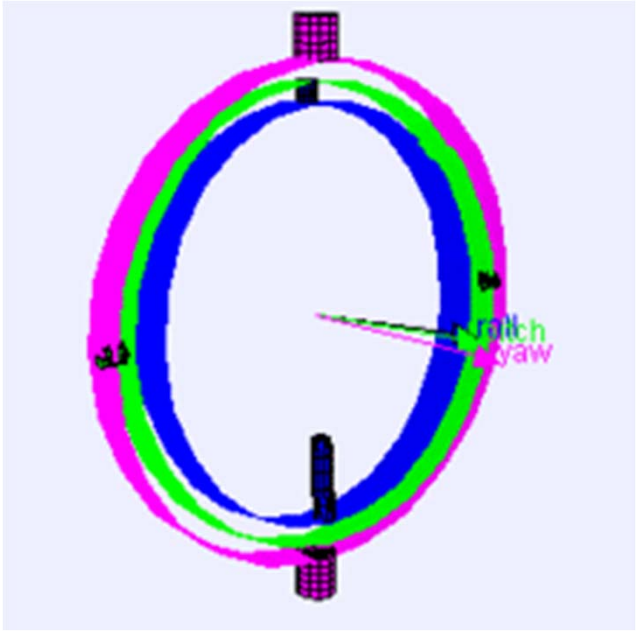


Křidélka

[NASA]

Gimbal [‘džimbl] – otočná podpěra

- Tři gimbals spojené dohromady + závaží či setrvačník

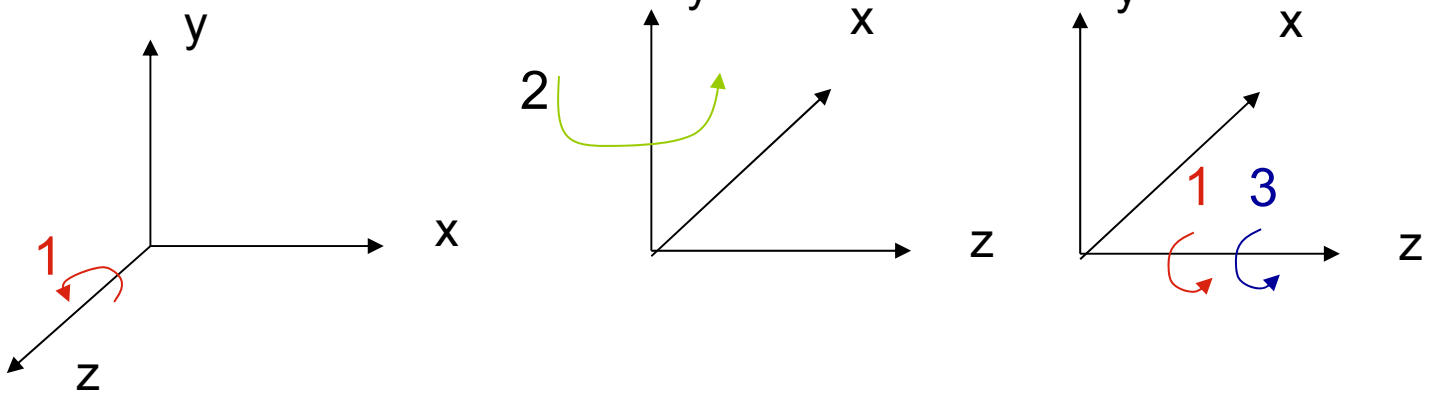


- Původní použití u gyroskopů a upevnění kompasů, kamen a sklenic na lodích od antiky.

[Wikipedia]

Rotace a „Gimbal lock“ [‘džimbl lok]

- Eulerovy úhly = pitch-yaw-roll (výška, kurs, rotace)
- komplexní otočení rozdělíme na otočení podle os X, Y, Z
- dělá se postupně => splynou-li osy, dojde ke ztrátě jednoho stupně volnosti (gimbal lock)



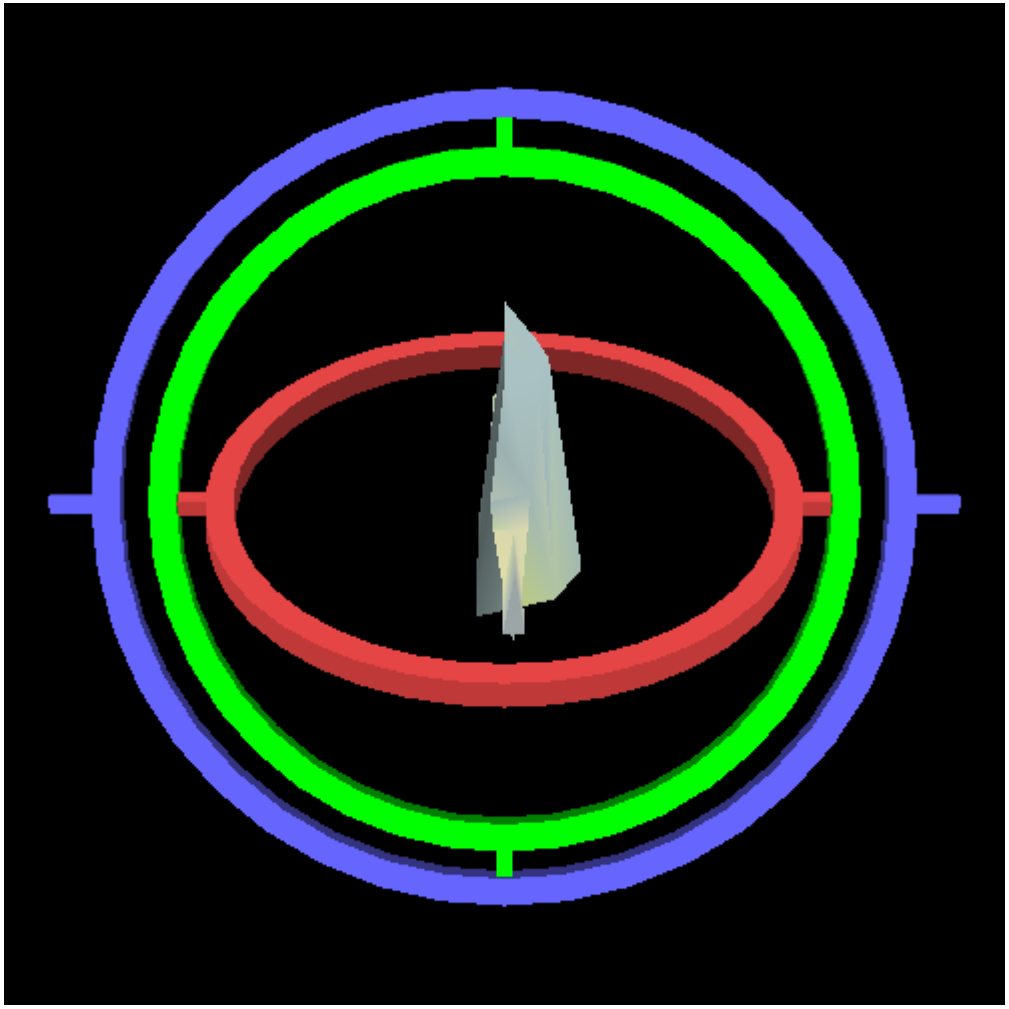
Pořadí rotací v programu:

```
Rotate(a,   1, 0, 0)  // X  3
Rotate(90, 0, 1, 0)  // Y  2
Rotate(c,   0, 0, 1)  // Z  1
```

Z splyne s X

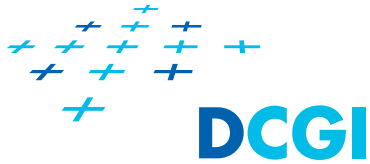
Rotace kolem X s modelem otočeným o 90°dle Y dělá totéž, co rotace kolem Z

Ztráta jednoho stupně „Gimbal Lock“



[Wikipedia]

Virtual trackball

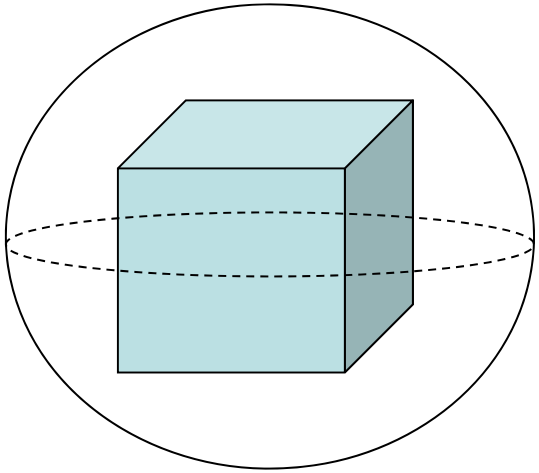


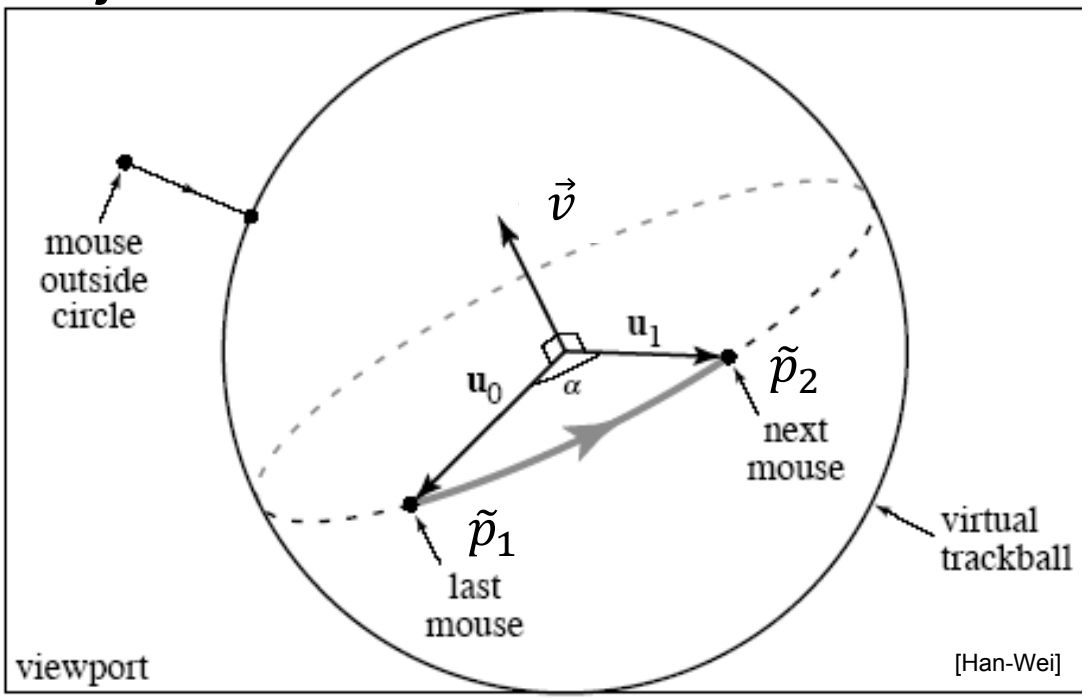
- Mapuje 2D plochu na 3D povrch polokoule
- Najde obecnou osu a úhel => nenastane gimbal lock!!!

- Kvaterniony
  - vhodné pro ukládání, sčítání, …
  - zatím se obejdeme bez nich

Virtual trackball - princip

- Představa: objekt obalen koulí se středem v počátku $\tilde{o}$ soustavy souřadnic objektu $\vec{o}^t$ a otáčí se vůči kameře $\vec{e}^t$ resp. obrazovce



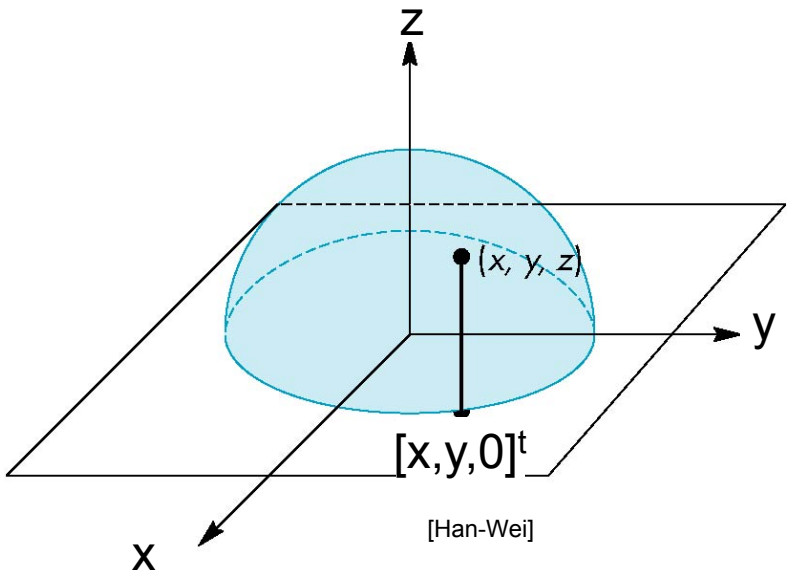


- Dva body $\tilde{p}_1$ a $\tilde{p}_2$ definují začátek a konec otáčení

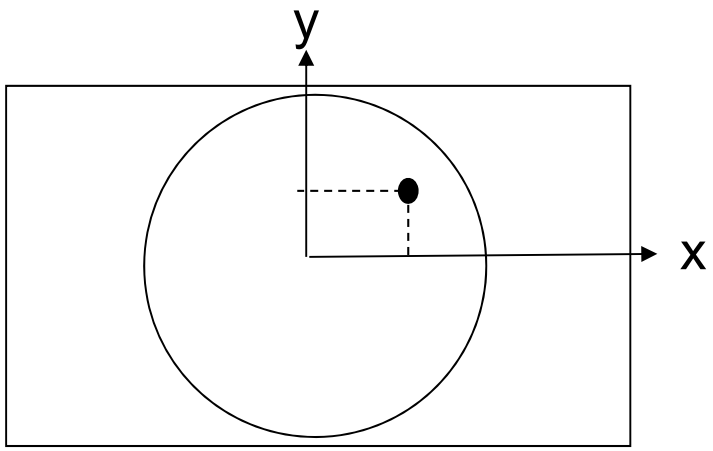
  => Úhel α a vektor obecné osy otáčení $\vec{v}$

Virtual trackball - princip

- Na záběr položíme polokouli
- Polokoule se na obrazovku promítne jako kružnice
- Myší zadáme x a y
- Promítneme polohu myši na povrch koule
  (pro x a y získáme z)



[Han-Wei]



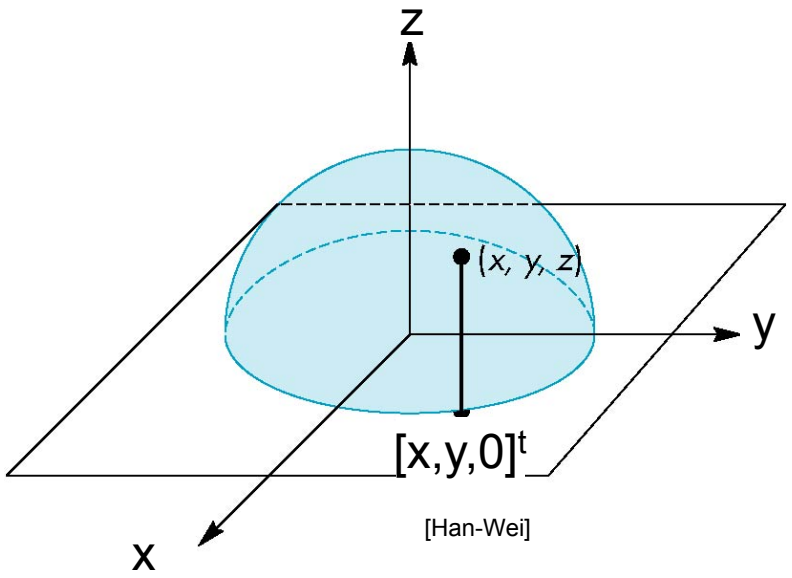
Virtual trackball - princip

Promítneme polohu myši na povrch koule
(pro x a y získáme z)

- Koule o poloměru r
- Bod na kouli se promítá na rovinu $z = 0$

  $[x \quad y \quad z]^t \Rightarrow [x \quad y \quad 0]^t$
- Pro známé $[x \quad y \quad 0]^t$ vypočteme z

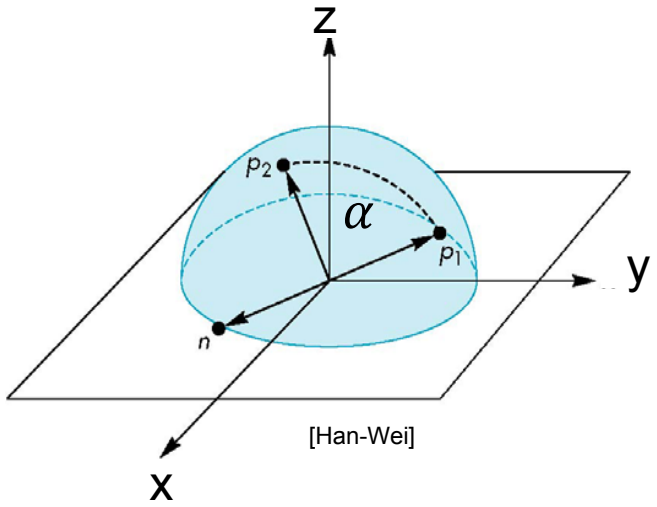
  $z = \sqrt{(r^2 - x^2 - y^2)}$



[Han-Wei]

Virtual trackball - princip

- Uložíme předchozí polohu myši $\tilde{p}_1$ a sledujeme aktuální $\tilde{p}_2$
- Normála $\vec{n}$ roviny $(\tilde{p}_1, \tilde{p}_2, \tilde{o})$ je osa otáčení okolo $\tilde{o}$
- Rotujeme objekt dle osy $\vec{n}$ o správný úhel α
  - Původní transformace O
  - Otočení $R(\alpha)$
  - Nová transformace $O' \leftarrow R(\alpha)O$

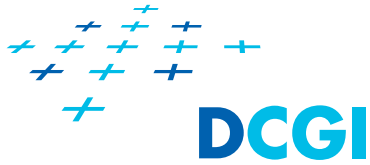


[Han-Wei]

$$\vec{\boldsymbol{o}}^t = \vec{\boldsymbol{w}}^t O \Rightarrow \vec{\boldsymbol{w}}^t R(\alpha) O = \vec{\boldsymbol{w}}^t O'$$

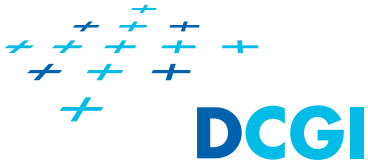
- Před interakcí si uchováme původní modelovou matici O
- Během interakce k ní akumulujeme nové otočení
- Po dokončení interakce ji aktualizujeme dle posledního α

Virtual trackball



1. Detekuj stisk myši a ulož si 3D souřadnice -> P1
2. Sleduj pohyb myši (drag) -> P2
   a) Promítni 2D body P1, P2 na kouli
   b) Urči osu rotace od P1 k P2 (normálu $\vec{n}$ roviny) a úhel α
   c) Přinásob pootočení k rotaci trackballu
   d) Překresli scénu
3. Ukonči sledování myši (uvolnění tlačítka)

Virtual trackball - implementace



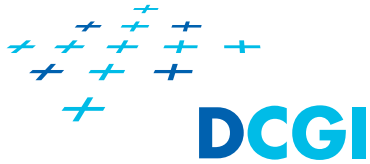
- Proměnná **`trackballRotationMatrix`** obsahuje celkovou rotaci trackballu (poskládanou z malých rotací od P1 k P2)
- Při vykreslování se skládá s aktuální modelovou maticí (násobí ji zleva) $O' \leftarrow R(\alpha)O$

```
glm::mat4 newModel = trackballRotationMatrix * modelMatrix;
```

a jako modelová matice používá se ta složená

```
// složená modelová matice
glUniformMatrix4fv( modelMatrixLocation, 1, GL_FALSE,
                    glm::value_ptr(newModel));
drawModel();
```

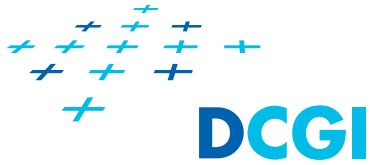
Virtual trackball - implementace



1. Detekce stisku myši a uložení výchozích souřadnic P1

```
//mouse button pressed within a window or released
void mouseCb(int button, int state, int x, int y) {

  if(button == GLUT_LEFT_BUTTON) {
    if(state == GLUT_DOWN) {
      startGrabX = x;                    // point P1
      startGrabY = y;
      glutMotionFunc(mouseMotionCb); // 2. start mouse tracking
      return;
    }
    else {  // GLUT_UP
      glutMotionFunc(NULL);           // 3. stop mouse tracking
    }
  }
}
```

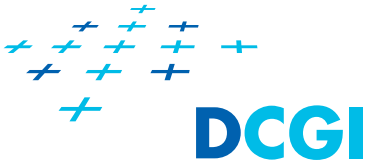
Virtual trackball - implementace



2. Sleduj pohyb myši -> P2

```
// mouse motion within the window with any button pressed (drag)
void mouseMotionCb(int x, int y) {
  endGrabX = x; // point P2
  endGrabY = y;

  if(startGrabX != endGrabX || startGrabY != endGrabY) {
    /* get rotation increment from trackball */
    trackball.addRotation(startGrabX, startGrabY, endGrabX,//2abc)
                          endGrabY, winWidth, winHeight);
    /* build rotation matrix from trackball rotation */
    trackball.getRotationMatrix(trackballRotationMatrix);
    startGrabX = endGrabX;  // move point P1 to current P2
    startGrabY = endGrabY;
    glutPostRedisplay(); // 2d)
  }
}
```

Virtual trackball - implementace

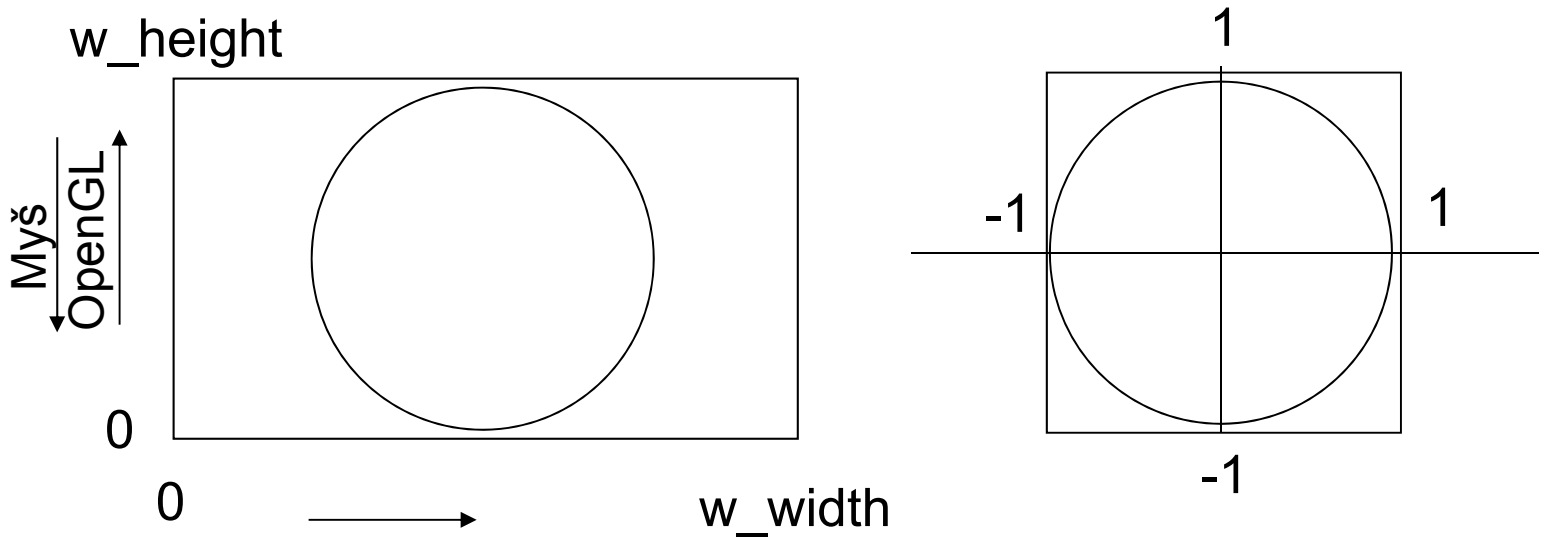


2abc) Přidání pootočení k celkové rotaci trackballu

```
void CClassicTrackball::addRotation(
                int startPointX, int startPointY,
                int endPointX,   int endPointY,
                int winWidth,    int winHeight ) {
 float endX, endY, startX, startY;
 if(startPointX == endPointX && startPointY == endPointY)
    return;   // no move => no rotation
 mapScreenCoords( startX, startY, startPointX, startPointY,// 2a)
                  winWidth, winHeight); //2D => 3D
 mapScreenCoords( endX, endY, endPointX, endPointY,        // 2a)
                  winWidth, winHeight); //2D => 3D
 glm::mat4 newRotation; // rotation increment
 computeRotation(newRotation, startX, startY, endX, endY); // 2b)

 // trackball rotation = rotation increment * previous rotation
 _rotationMatrix = newRotation * _rotationMatrix;          // 2c)
}
```

Virtual trackball – mapování na kouli

2a) Mapování souřadnice myši na normalizovanou kouli



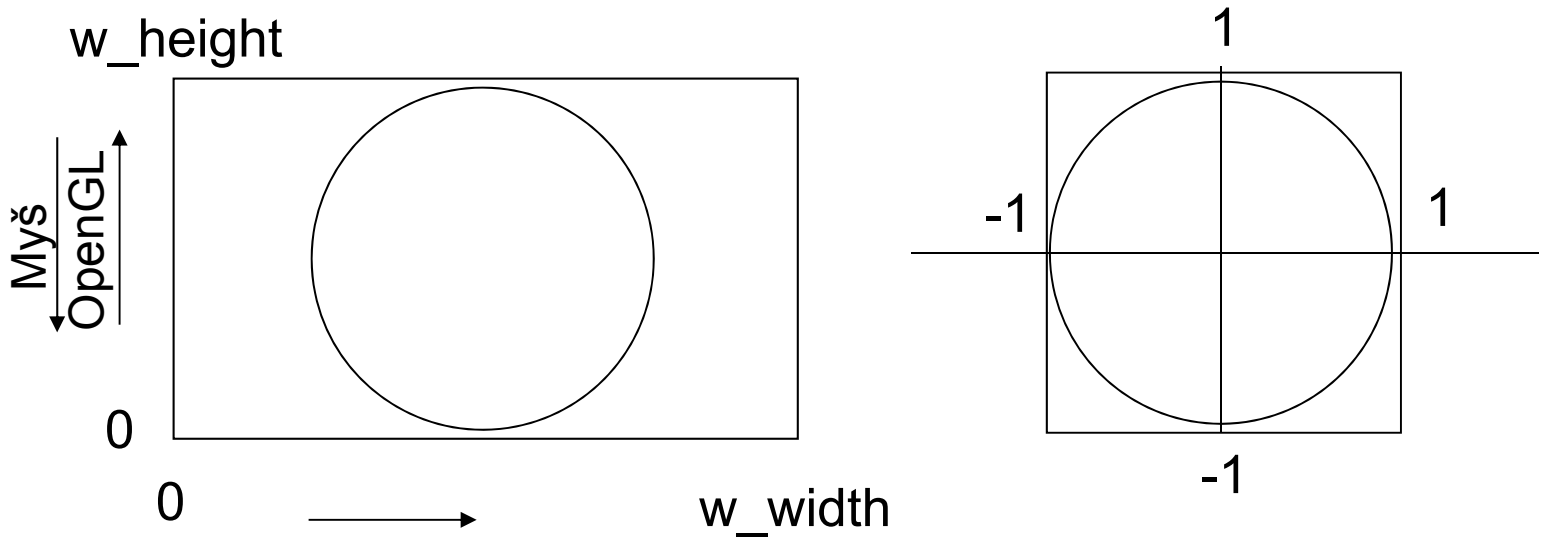
Převod intervalu ⟨0, w_width⟩ na rozsah ⟨-1, 1⟩

x' = – 1.0 + (screen_x / w_width) * 2.0

y' = 1.0 – (screen_y / w_height) * 2.0

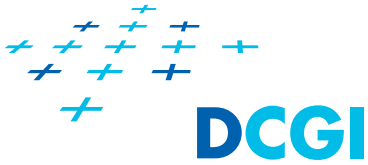
Virtual trackball - implementace

2a) Mapování souřadnice myši na plochu koule




```
void CTrackball::mapScreenCoords(float& outX, float& outY,
       int screenX, int screenY, int winWidth, int winHeight)
{
  outX = -1.0f + 2.0f * screenX / winWidth;
  outY =  1.0f - 2.0f * screenY / winHeight;
}
```

Virtual trackball - implementace



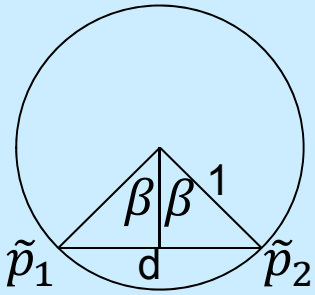
2b) Výpočet matice potočení trackballu $R(\alpha)$

```
void CClassicTrackball::computeRotation(glm::mat4& rotation,
float startPointX, float startPointY, float endPointX, float
endPointY)
{
  if(startPointX == endPointX && startPointY == endPointY) {
     rotation = glm::mat4(1.0); // Zero rotation
     return;
  }
  glm::vec3 axis;
  float angle;

  computeRotationAxisAndAngle(axis, angle,
              startPointX, startPointY, endPointX, endPointY);

  rotation = glm::rotate(glm::mat4(1.0f), angle, axis);
}
```

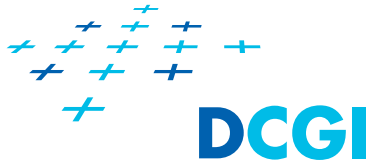
Virtual trackball - implementace

2b) Výpočet osy otáčení a úhlu α

```
 void CTrackball::computeRotationAxisAndAngle(
        glm::vec3& axis, float& angle, float startPointX,
        float startPointY, float endPointX, float endPointY)
{  // Compute z-coordinates for projection of P1, P2 to sphere
   glm::vec3 p1(startPointX, startPointY,
        projectToSphere(TRACKBALLSIZE, startPointX, startPointY));
   glm::vec3 p2(endPointX, endPointY,
        projectToSphere(TRACKBALLSIZE, endPointX, endPointY));
   axis = glm::normalize(glm::cross(p1, p2));   // AXIS
   // ALGLE to rotate around that axis.
   glm::vec3 d = p1 - p2; // chord (tětiva), t=sin(β), β = α/2
   double t = glm::length(d) / (2.0 * TRACKBALLSIZE);
   //* Avoid problems with out-of-control values...s
   if(t > 1.0)     t = 1.0;
   if(t < -1.0)    t = -1.0;
   angle = float(RADTODEG(2.0f * asin(t)));  // ANGLE
}                                            // (in degrees)
```



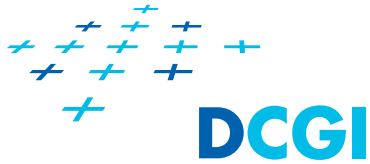
Virtual trackball - implementace



2a) Projekce 2D bodu na kouli (uvnitř) či hyperbolu (vně)

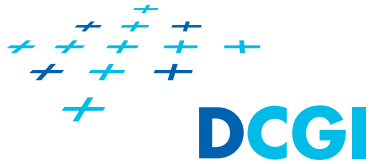
```
float CTrackball::projectToSphere(float radius,float x,float y)
{
  double d, t, z;

  d = sqrt(x*x + y*y);
  if(d < radius * 0.70710678118654752440) {    // Inside sphere
    z = sqrt(radius*radius - d*d);
  }
  else {                                       // On hyperbola
    t = radius / 1.41421356237309504880;
    z = t*t / d;
  }

  return (float)z;
}
```

Virtual trackball



- Roger Crawfis: “Implementing a Virtual Trackball or Examiner Viewer”, http://www.cse.ohio-state.edu/~crawfis/cis781/Slides/VirtualTrackball.html
  - pro starou verzi OpenGL, kterou se zde neučíme
  - Mírně odlišný postup – po celou dobu interakce se vztahuje k prvnímu bodu P1

Odkazy



- [NASA] *The beginners guide to aeronautics*. NASA Glenn Research Center, http://www.grc.nasa.gov/WWW/k-12/airplane/
- [Han-Wei] Han-Wei Shen. Quaternion and Virtual Trackball. CSE 781 Introduction to 3D Image Generation, 2007 http://www.cs.sunysb.edu/~mueller/teaching/cse564/trackball.ppt