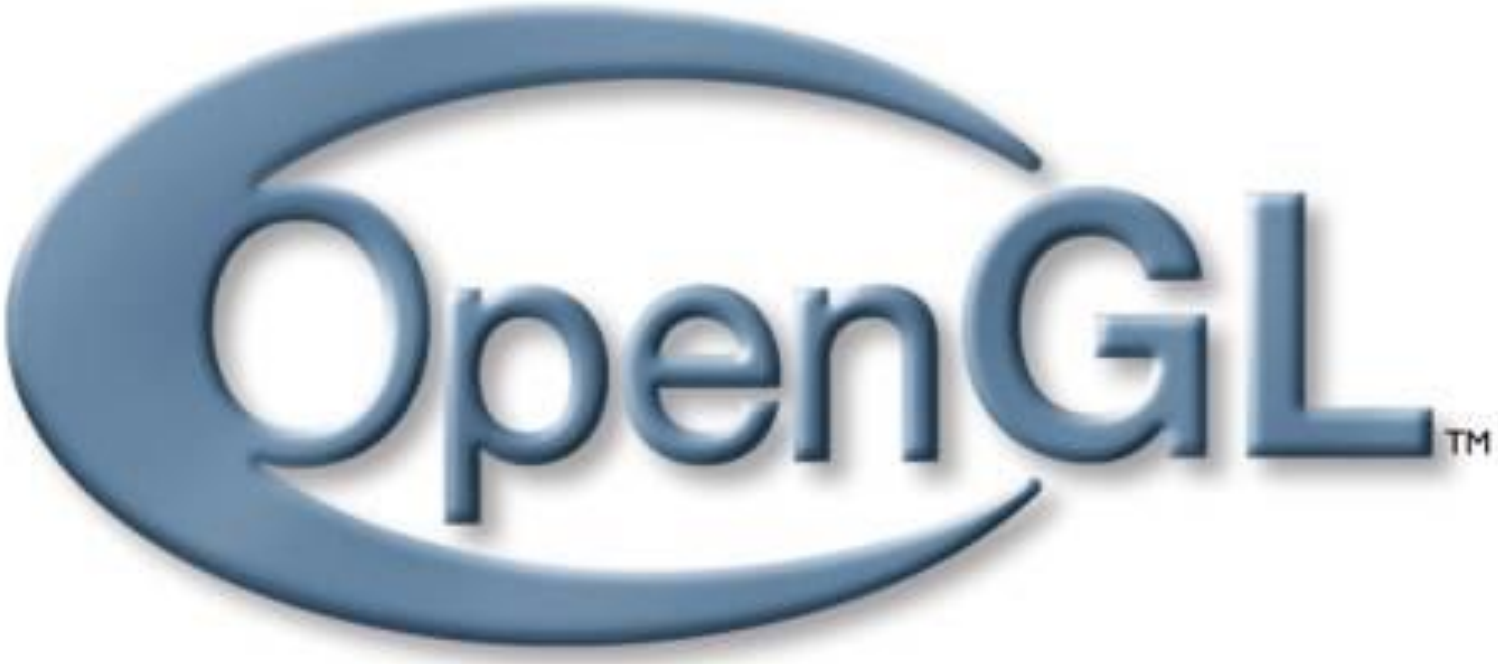
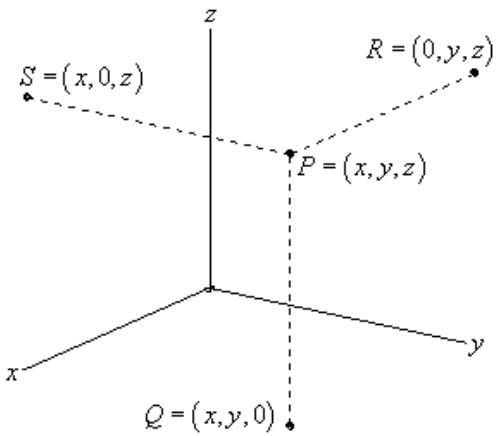


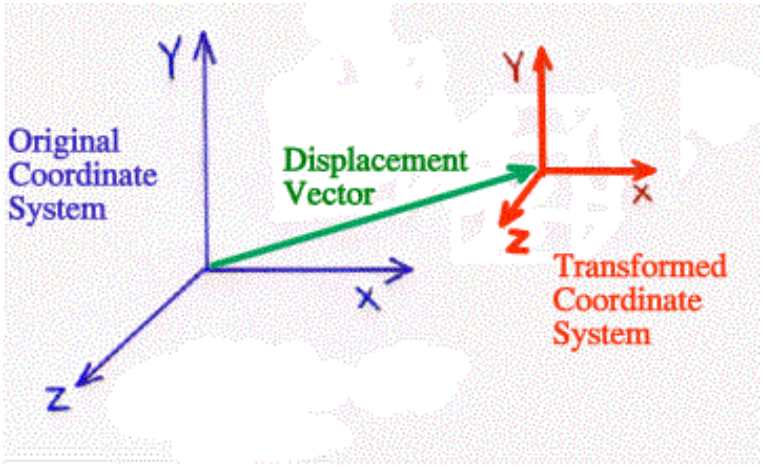
part 3

Martin Samuelčík
http://www.sccg.sk/~samuelcik
Room I4

Vertex coordinates

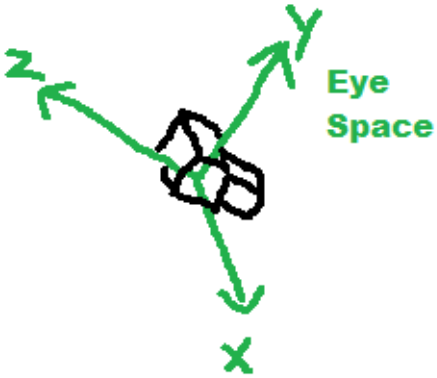
- From input coordinates to window coordinates
- Coordinates are always related to coordinates system, space, frame
- Transforming vertex = changing its coordinates = changing vertex coordinates from one system to another

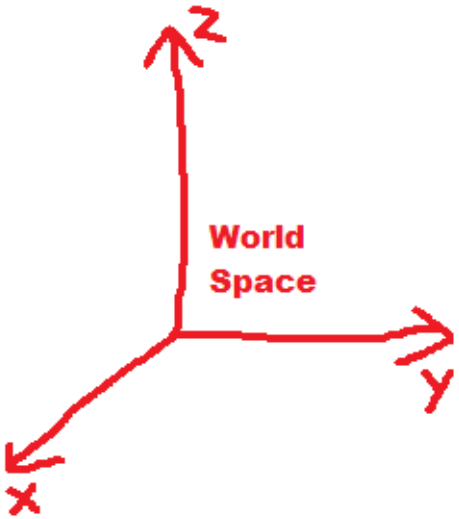


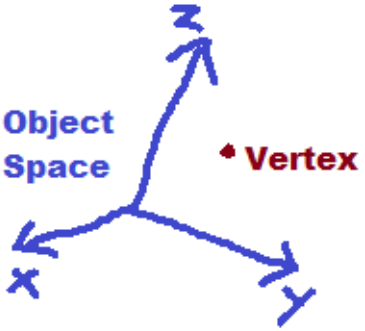


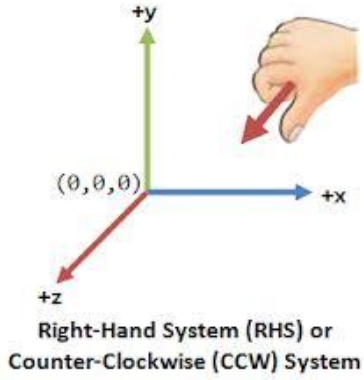
Coordinate systems

- When working with vertex coordinates, it is always in some space
- Object space – this is input space for all vertices
- World space – main coordinate system of whole scene
- Eye (camera) space – system where camera is always in point (0, 0, 0) heading direction –z



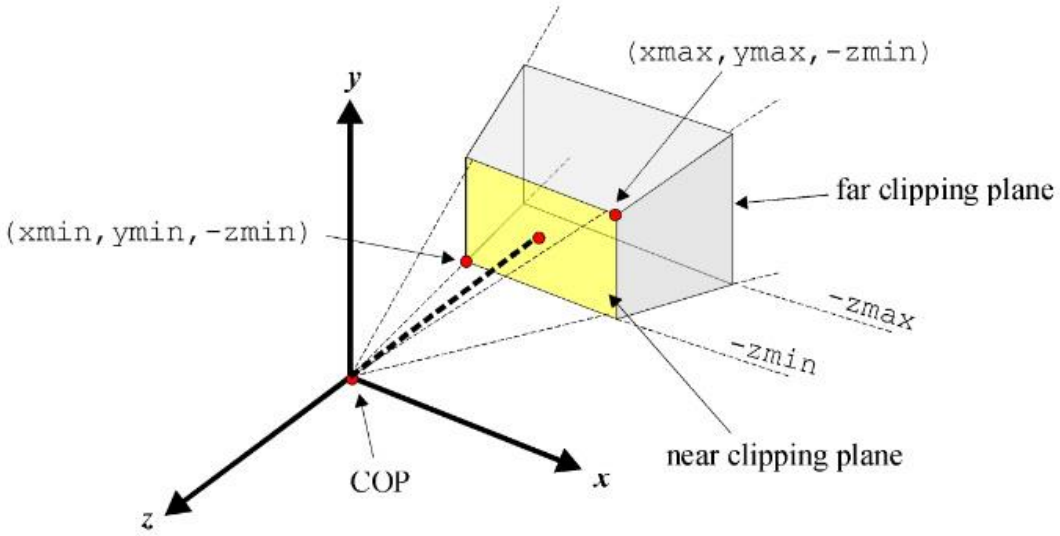


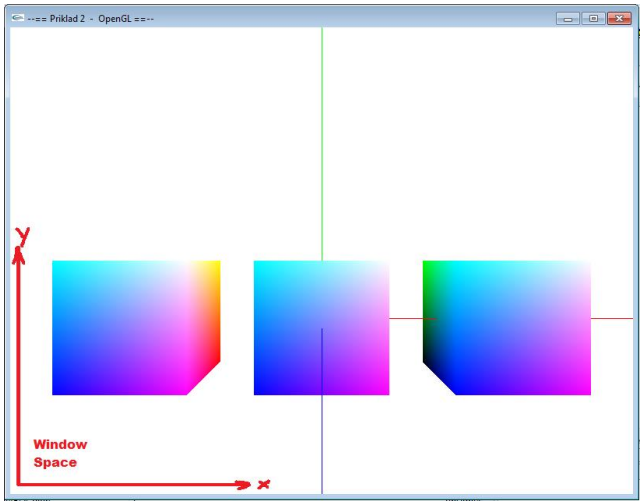




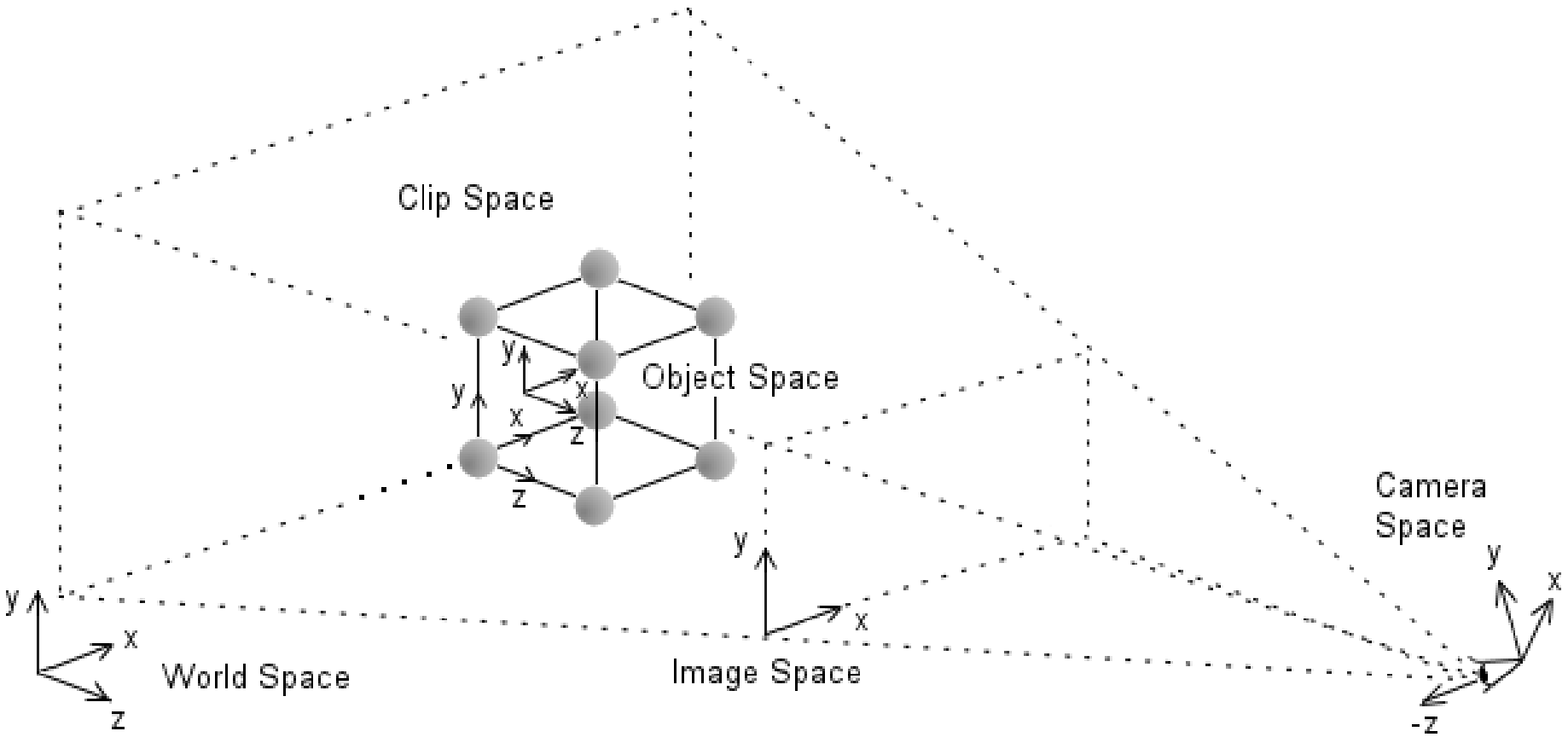
Coordinate systems

- Clip space – system after projection (3D to 2D) transformation is applied
- Normalized device coordinates – intermediate system used for computation of window coordinates, all coordinates of visible vertices are in the range <-1,1>
- Window (image) space – system inside window





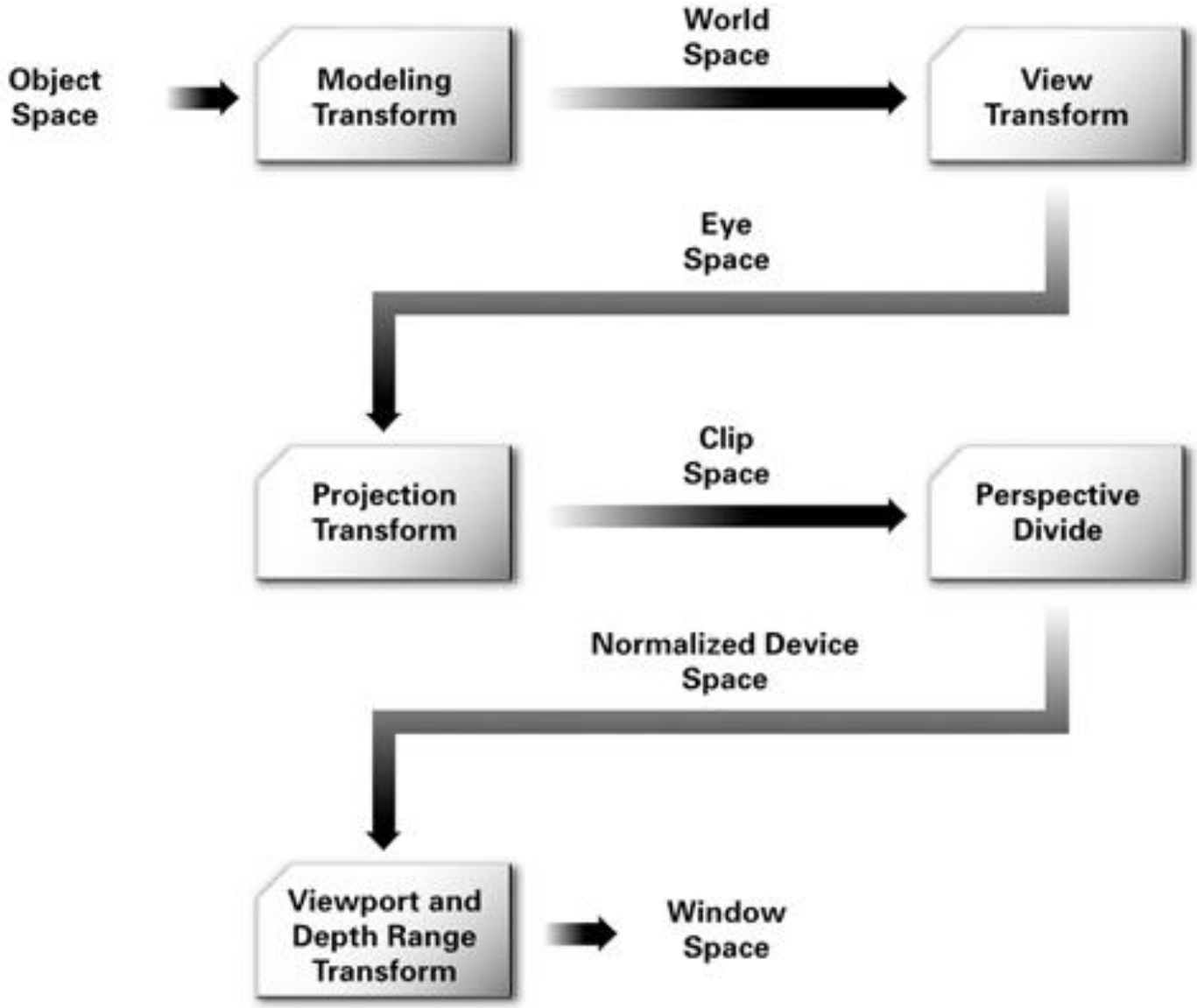
Coordinate systems



Transformations

- Applying for vertices, normals, texture coordinates
- Changing coordinates of the same vertex between coordinate systems
  - Modeling transformation – from object to world system
  - Viewing transformation – from world to eye system
  - Projection transformation – from eye to clip system
  - Perspective divide – from clip to normalized device system
  - Viewport transform – from normalized devide to window system

Transformations & systems



Transformations

Transorfmation – 4x4 matrix

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

Vertex – 4x1 matrix

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Applying transformation to coordinates = matrices multiplication

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Transformations

- Order of applying transformations is important

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} * \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} \neq \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} * \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

- Transformations are applied in reverse order as specified

$$\left[\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} * \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} \right] * \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} * \left[\begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \right]$$

OpenGL transformations

- Model and view transformation in one matrix
- 3 types of transformations, state variables Modelview, Projection, Texture
- One active type of transformation
- All operations with matrix are applied to current type of matrix
- **void glMatrixMode(GLenum mode)**
  - mode = GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE

Working with matrices

- **void glLoadIdentity(void)**
  - set current matrix as identity matrix
- **void glLoadMatrix[fd](const TYPE *m)**
  - set current matrix as matrix m
  - m is array of floats or doubles with the count at least 16
- **void glMultMatrix[fd](const TYPE *m)**
  - multiply current matrix with matrix m and set current matrix to result
  - m is array of floats or doubles with the count at least 16

Example

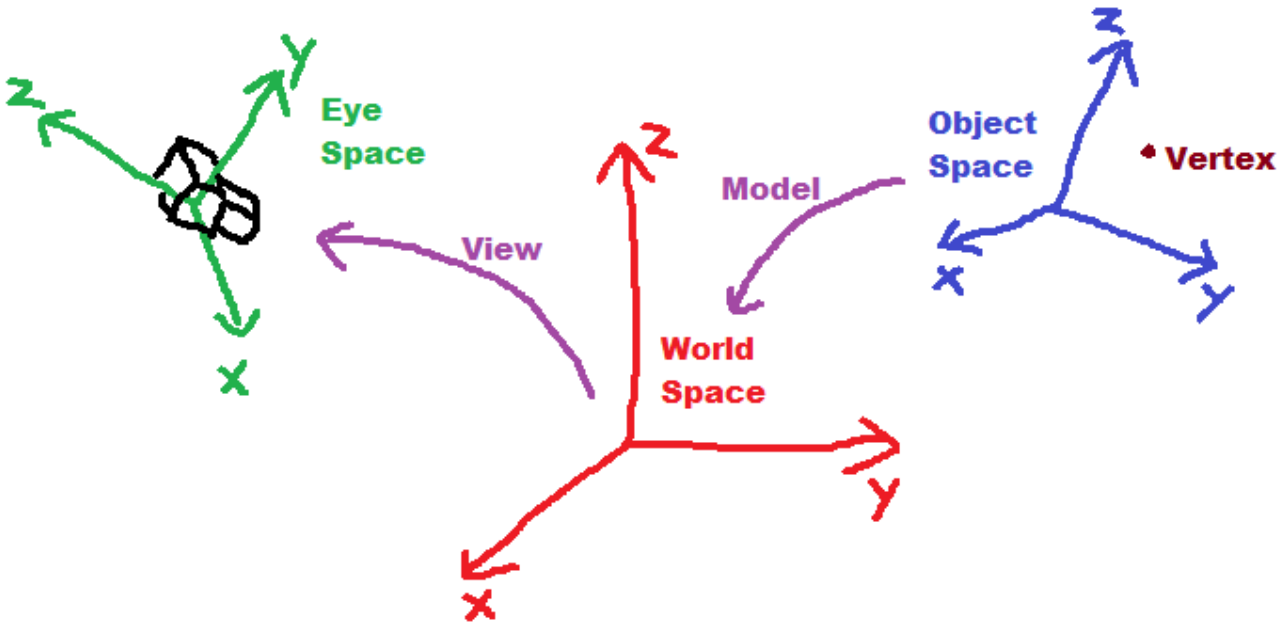
- glMatrixMode(GL_MODELVIEW);
- glLoadIdentity();
- glMultMatrixf(m_1);
- glMultMatrixf(m_2);
- glMultMatrixf(m_3);

- glBegin(GL_POINTS);
- glVertex3fv(v);
- glEnd();

- Vertex v is transformed using matrix m_3, then m_2, and then m_1

Modelview matrix

- Modeling and view transformation together
- Model – how object is positioned in the world space
- View – how the camera is viewing world space
- Modelview matrix = View * Model
- Each modelview matrix represents some local coordinate system – identity matrix is eye space



Model matrix

- **void glTranslate{fd}(TYPE x, TYPE y, TYPE z)**
  - Compute translation matrix and mutiplies it with current matrix
- **void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z)**
  - Compute rotation matrix and multiplies it with current matrix
- **void glScale{fd}(TYPE x, TYPE y, TYPE z)**
  - Compute rotation matrix and multiplies it with current matrix

Model matrix

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

translation

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

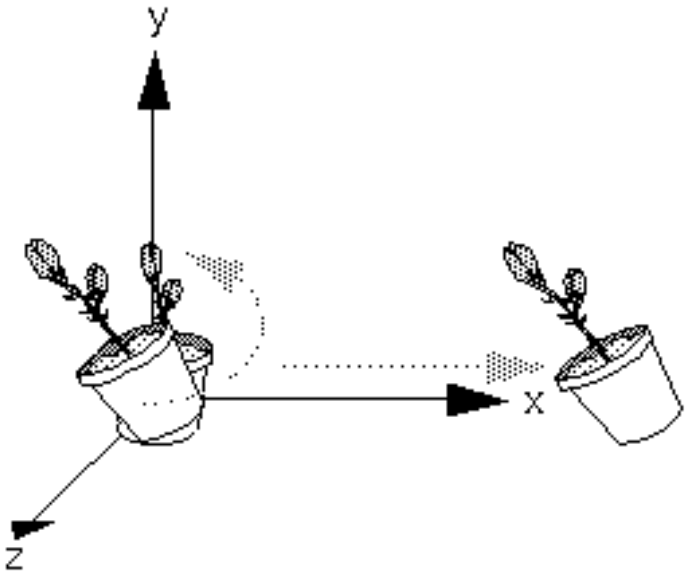
scaling

$$\begin{pmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ xy(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

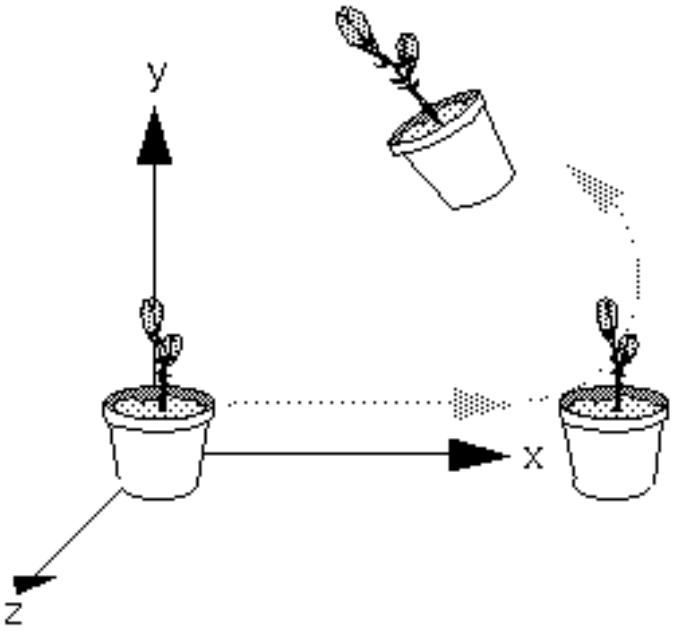
where $c = \cos\theta, \quad s = \sin\theta$

rotation

Order matters



- glTranslate
- glRotate
- DrawFlowerpot



- glRotate
- glTranslate
- DrawFlowerpot

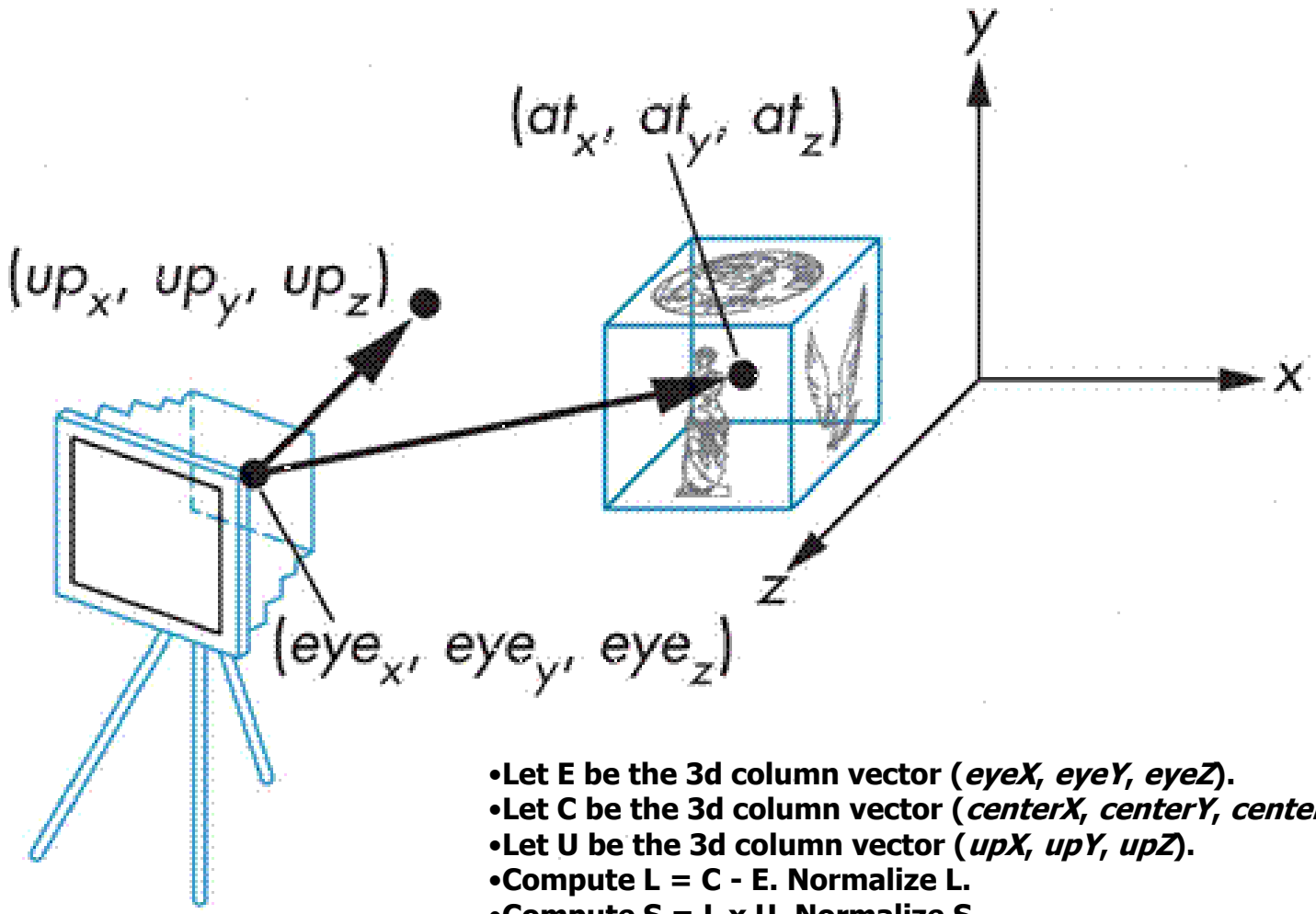
View matrix

- Transformation from world to eye space
- Specifies how camera views virtual space
- In eye space, camera has coordinates [0,0,0] and is heading in -z direction
- If view matrix is identity, then eye space = world space, vertices near [0,0,0] are not visible
- Basic view matrix – glTranslatef(0,0,-5)

View matrix

- Function for setting view matrix from GLU library
- Computes view matrix and multiplies it with current matrix
- **void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz)**
  - eye – camera position in world space
  - center – interest position in world space
  - up – up vector in world space

View matrix



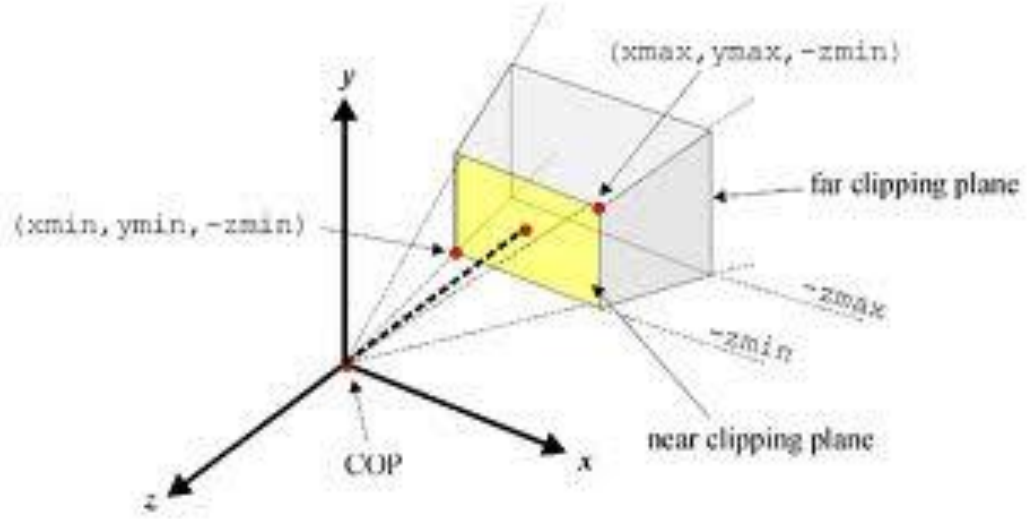
- Let E be the 3d column vector (*eyeX*, *eyeY*, *eyeZ*).
- Let C be the 3d column vector (*centerX*, *centerY*, *centerZ*).
- Let U be the 3d column vector (*upX*, *upY*, *upZ*).
- Compute L = C - E. Normalize L.
- Compute S = L x U. Normalize S.
- Compute U' = S x L.
- M is the matrix whose columns are, in order:
- (S, 0), (U', 0), (-L, 0), (-E, 1) (all column vectors)

Modelview example

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// here is eye space
gluLookAt(5,4,3, 0,0,0, 0,1,0);
// here is world space
glTranslatef(5,5,0);
// here is first object space
DrawObject1();
glTranslatef(-5,0,2);
glRotatef(45, 0,1,0);
// here is second objects space
DrawObject2();
```

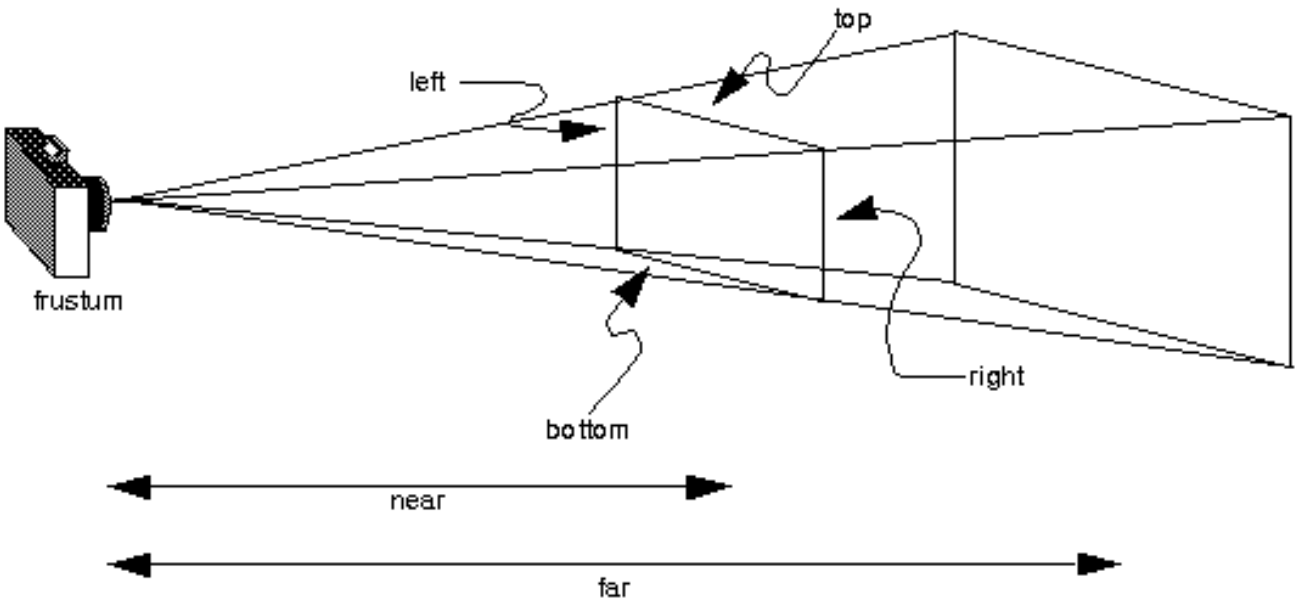
Projection matrix

- Transforming 3D to 2D
- Again 4x4 matrix – result of projection is 4-component vector
- Still remembering also z coordinate
- Defining viewing volume – only vertices inside volume will be rendered



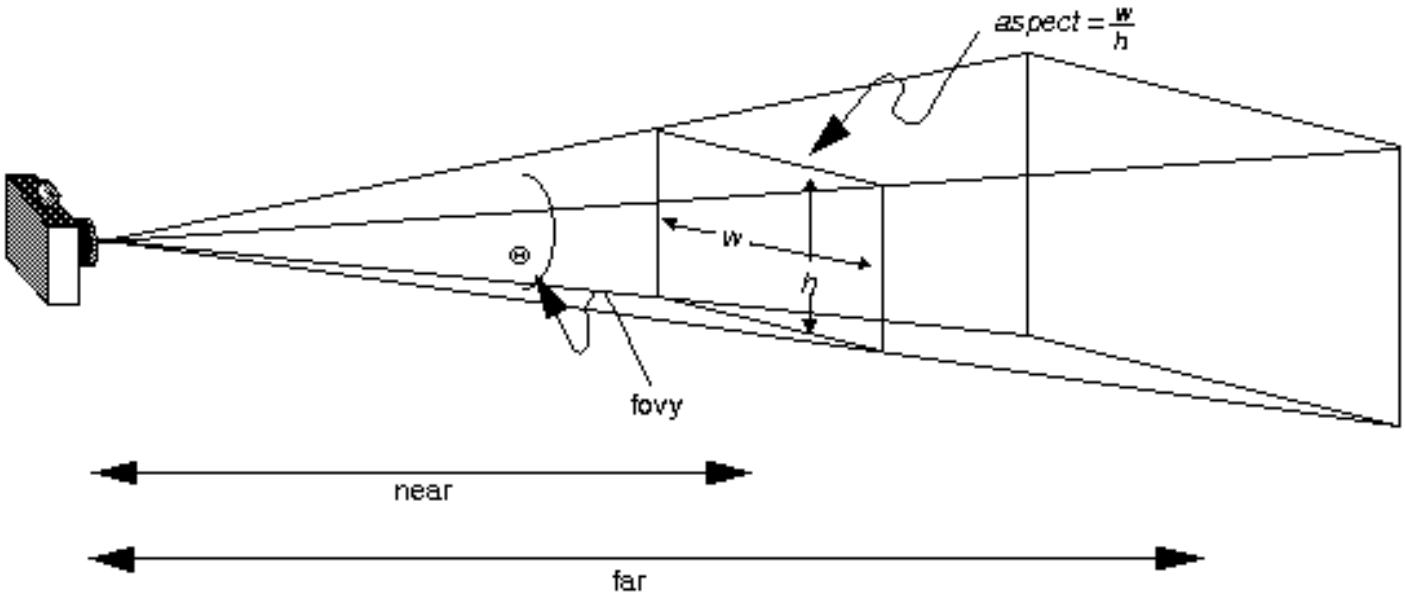
Perspective projection

- **void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)**



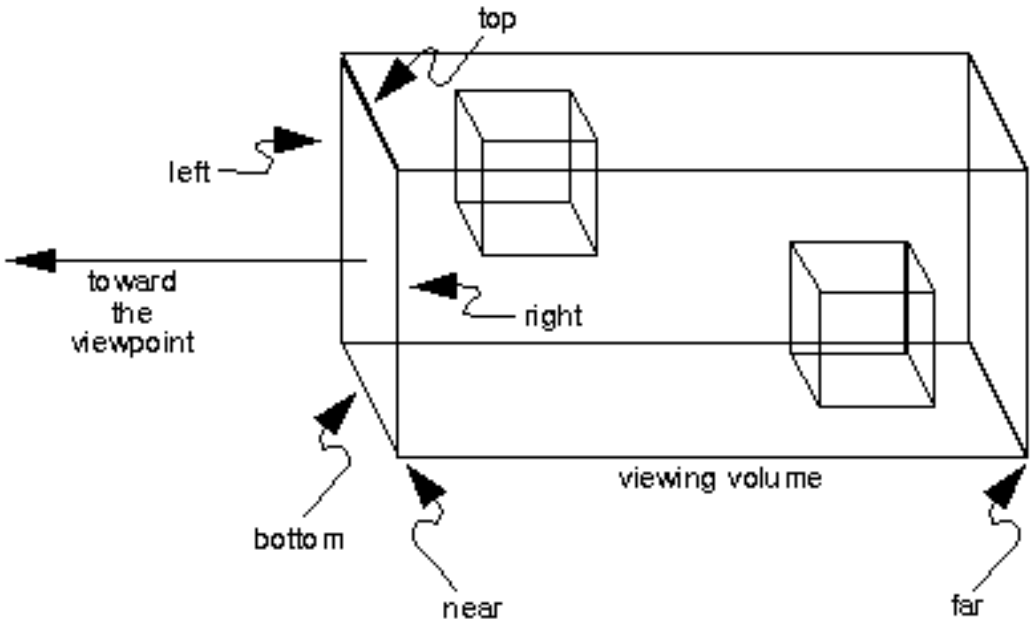
Perspective projection

- **void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)**



Orthographic projection

- **void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)**
- **void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)**



Projection matrices

$$\begin{bmatrix} \frac{2\,\text{zNear}}{\text{right}-\text{left}} & 0 & A & 0 \\ 0 & \frac{2\,\text{zNear}}{\text{top}-\text{bottom}} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$A = \frac{\text{right}+\text{left}}{\text{right}-\text{left}}$$

$$B = \frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}}$$

$$C = -\frac{\text{zFar}+\text{zNear}}{\text{zFar}-\text{zNear}}$$

$$D = -\frac{2\,\text{zFar}\,\text{zNear}}{\text{zFar}-\text{zNear}}$$

glOrtho

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{far-near} & t_z \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

where

$$t_x = -\frac{right+left}{right-left}$$

$$t_y = -\frac{top+bottom}{top-bottom}$$

$$t_z = -\frac{far+near}{far-near}$$

glFrustum

Clip & normalized space

- Result of projection transformation are coordinates of vertex in clip space
- In clip space, if x,y,z coordinates of vertex are in range <-w,w>, then are not clipped
- Normalized space – x,y,z are divided by w
- In normalized space, all remaining vertices have coordinates in range <-1,1>
- Normalized space is used for better mapping to window

Window space

- Pixel coordinates of vertex together with its depth – passed to rasterizer
- Viewport - specifies part of window where image plane of projection is mapped
- **void glViewport(GLint x, GLint y, GLsizei width, GLsizei height)**
- Aspect ratio of viewing volume and viewport should be equal
- Depth range specifies interval for z coordinate
- **glDepthRange(n, f)**

Projection & viewport

```
void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glDepthRange(0, 1);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (h == 0)
        h = 1;
    gluPerspective(80, (float)w/(float)h, 1.0, 5000.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

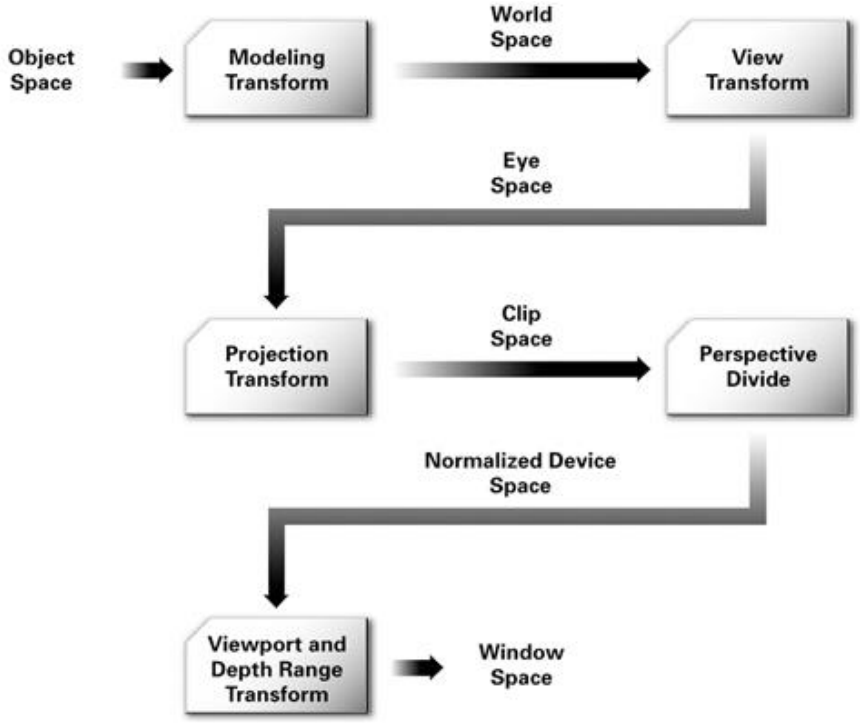
Transformations conclusion

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelView} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{w}}{2} x_{ndc} + (\mathrm{x} + \frac{\mathrm{w}}{2}) \\ \frac{\mathrm{h}}{2} y_{ndc} + (\mathrm{y} + \frac{\mathrm{h}}{2}) \\ \frac{\mathrm{f}-\mathrm{n}}{2} z_{ndc} + \frac{\mathrm{f}+\mathrm{n}}{2} \end{pmatrix}$$



OpenGL®

Retrieving transformations

- GLdouble model[4*4];
- glGetDoublev(GL_MODELVIEW_MATRIX, model);

- GLdouble proj[4*4];
- glGetDoublev(GL_PROJECTION_MATRIX, proj);

- GLint view[4];
- glGetIntegerv(GL_VIEWPORT, view);

Matrix stack

- Each type of matrix (MODELVIEW, PROJECTON, TEXTURE) has its own stack
- Matrix on top of stack is current
- Working with current matrix
- **void glPushMatrix(void)**
- **void glPopMatrix(void)**
- Pushing – duplicate matrix on top
- Popping – delete matrix from top
- Use to remember matrices for future use, for example view matrix

Matrix stack example

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// here is eye space
gluLookAt(5,4,3, 0,0,0, 0,1,0);
// here is world space
// remember it for future use
glPushMatrix();
glTranslatef(5,5,0);
// here is first object space
DrawObject1();
// return to world space
glPopMatrix();
// here is world space again
glTranslatef(-5,0,2);
glRotatef(45, 0,1,0);
// here is second objects space
DrawObject2();
```

The End!

Questions?