Exercises Computer Graphics

Prof. Dr.sc.hum. Hans-Heino Ehricke

Exercise: Geometrical Transformations, 3D Viewing

The focus of this exercise is on geometrical transformations, necessary in order to transform model data from the three-dimensional world coordinate system to the viewing plane of the viewport in the context of 3D viewing:

1. modeling transformations
2. viewing transformations
3. projection transformations
4. viewport transformation

Modeling Transformations

Modeling transformations, e.g. translation, rotation and scaling, are used to modify geometric properties of model data within the world coordinate system and to generate complex from simple objects.

All transformations may be tackled in the same manner, if homogeneous coordinates are used. Transformations are defined as 4×4-matrices. Tey are applied to the model data by multiplication of the matrix with each object point (vertex). The world coordinate system is right-handed, three-dimensional and orthogonal.

The **translation T** with the distance vector d = (dx, dy, dz) is defined as:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

The **rotation R** rotates an object in a counterclockwise direction about one of the coordinate axes.

Rotation about x-axis with angle φ:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about y-axis with angle φ:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation about z-axis with angle φ:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Since a universal rotation matrix is rather confusing, complex rotations are defined with a series of the above metioned elementary rotations.

Scaling S changes the size of an object along the axes with the scaling factor vector s = (sx, sy, sz). Values $|s_i|>1$ (i = x, y, z) stretches the object, values $|s_i|<1$ shrinks it. Negative values additionally reflect objects across the corresponding axis.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

By the sequential application of different transformations arbitrary complex transformations may be carried out. If the task is e.g. to scale, rotate and translate an object the resulting transformation matrix is built by multiplication of the elementary matrices:

$M_{all} = T * R * S$

Please note: the multiplication of matrices is *not* commutative.

For the definition and multiplication of transformation matrices OpenGL provides two functions:

void **glLoadMatrix**{fd}(const TYPE *m)
Sets the sixteen values of the current matrix to those specified by m.

void **glMultMatrix**{fd}(const TYPE *m)
Multiplies the matrix specified by the sixteen values pointed to by m by the current matrix

and stores the result as the current matrix.

The following code example draws a single point using two modeling transformations:

```
glMatrixMode(GL_MODELVIEW);    // switches to model view mode (OpenGL is a
state machine)
glLoadIdentity();              // loads the identity matrix as current
matrix
glMultMatrixf(N);              // apply transformation N
glMultMatrixf(M);              // apply transformation M
glBegin(GL_POINTS);
glVertex3f(v);                 // draw transformed vertex v
glEnd();
```

The order of transformation is as follows: First, v is transformed by M, then (Mv) is transformed by N. Effectively, v is transformed by NM.

As a more comfortable alternative modeling transformations may be defined with the OpenGL functions
void **glTranslate**{fd}(TYPE x, TYPE y, TYPE z);
void **glRotate**{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
void **glScale**{fd}(TYPE x, TYPE y, TYPE z);

Viewing Transformations

A viewing transformation changes the position and orientation of the *eyepoint* (in the literature sometimes also called *viewpoint*). The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera. The position of the camera or anologously of the *eyepoint eye* is defined in the context of the world coordinate system. It defines the origin of the eyepoint or camera coordinate system U, V, N. (frequently the left-handed system U, V, W with w = -n is used). Just as you move the camera to some position and rotate it until it points in the desired direction, viewing transformations are generally composed of translations and rotations.

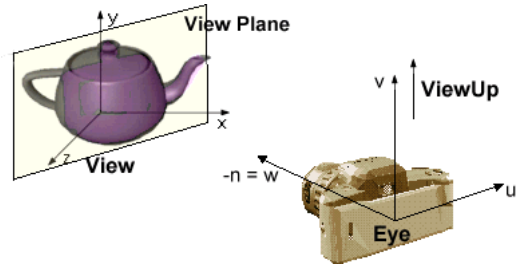


Fig. 1: Viewing transformations.

The camera, positioned at point *eye*, is directed towards the *reference point view*. By default this is the origin of the world coordinate system. The vector *viewup* defines the upward direction of the camera system.

Initially the viewpoint is located in the origin of the world coordinate system and the camera points down the negative z-axis. Transformation of model data into the camera coordinate system is carried out by two steps:

1. Translation of the viewpoint backward, away from the objects.

2. Rotation of the camera coordinate system such, that the objects are viewed from the desired side.

In OpenGL these transformations are defined by

```
glTranslate{fd}(...)
glRotate{fd}(...);
```

Note: The order in which the translate and rotate commands have to be issued is crucial: (1) translate, (2) rotate.

Alternatively, the *gluLookAt()* utility routine may be used:

void **gluLookAt**(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

It defines a viewing matrix and multiplies it to the right of the current matrix. The desired eyepoint is specified by *eyex, eyey, eyez*. The reference point *centerx, centery, centerz* specifies any point along the desired line of sight. The *upx, upy, upz* arguments indicate which direction is up.

Note: Before you issue any of the modeling or viewing transformation commands, remember to call
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

Projection Transformations

The transformation of model data from 3D space to the two-dimensional viewing plane is called projection transformation. There are two basically different projection techniques:

1. Perspective projection
2. Orthographic projection

Fig. 2: Result of perspective (right) and orthographic (left) projection of the teapot model.

A simple orthographic projection is the *orthogonal* one. In this case projection rays are orthogonal to the viewing plane. The transformation matrix for a projection onto the uv-plane at n=0 is given below:

$$\begin{pmatrix} Vx \\ Vy \\ * \\ Vw \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} u \\ v \\ n \\ 1 \end{pmatrix}$$

V_x and V_y are the coordinates of a point in the viewing plane.

In OpenGL an orthographic projection may be defined by the function

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

Fig. 3 explains the meaning of the parameters.

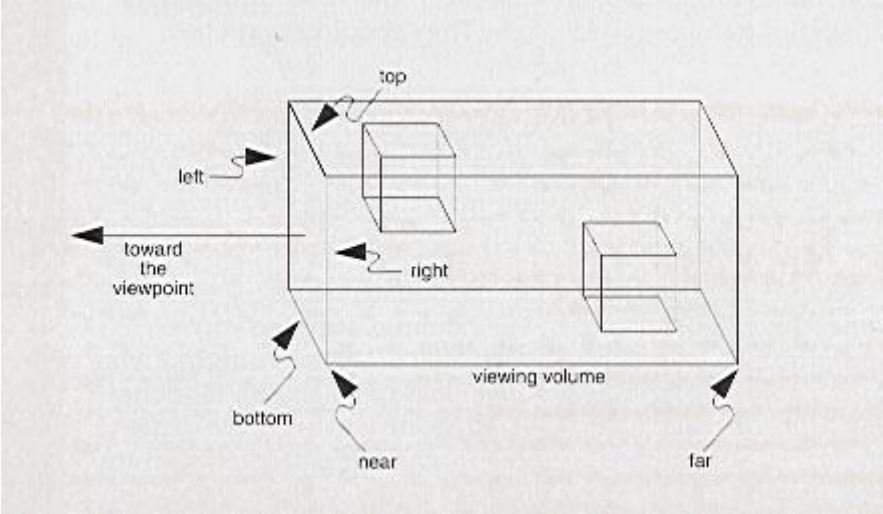


Fig. 3: Orthographic projection and its parameters.

In the context of a perspective projection rays are not parallel, but converge to a center point. The effect is foreshortening: the farther an object is from the camera, the smaller it appears in the final image. This occurs because the viewing volume for a perspective projection is a frustum of a pyramid (a truncated pyramid whose top has been cut off by a plane parallel to its base.

A simple perspective projection is given by the transformation matrix below:

$$\begin{pmatrix} Vx \\ Vy \\ * \\ Vw \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/nd & 1 \end{pmatrix} * \begin{pmatrix} u \\ v \\ n \\ 1 \end{pmatrix}$$

In this case the projection plane is parallel to the uv-plane and cuts the n-axis at n=0. The center of projection is $-n_d$. For $n_d \rightarrow \infty$ the transformation matrix converges to the above mentioned orthographic projection matrix.

In OpenGL a perspective transformation may be defined with the function

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

or the utility library routine

void **gluPerspective**(GLdouble fovy, GLdouble aspect, Gldouble zNear, GLdouble zFar);
The parameters are explained in fig. 4.

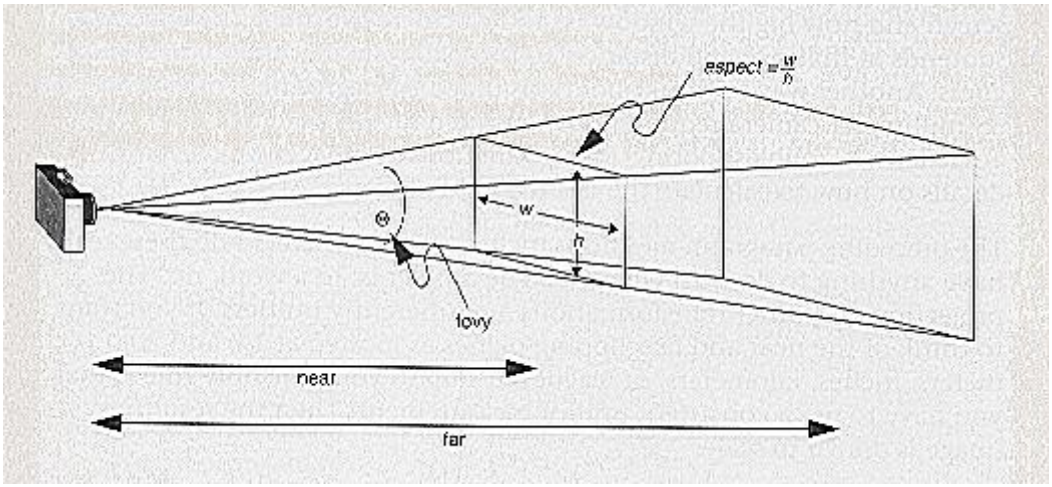


Fig. 4: Definition of a perspective projection (gluPerspective()).

Note: Do not forget to switch to projection matrix mode before issuing projection transformation commands:

glMatrixMode(GL_PROJECTION);
glLoadIdentity();

Viewport Transformation

After the projection transformation the model data have already been mapped to a two-dimensional coordinate system, the viewing plane. In the next step the image has to be mapped to the output window by linear transformations:
Scaling (adapt the image size to the size of the output window)
Translation (shift the image to an arbitrary position within the output window).

With OpenGL these transformations are carried out with the function

void **glViewport**(GLint x, GLint y, GLsizei width, GLsizei height);
It defines a pixel rectangle in the output window into which the final image is mapped. The *(x, y)* parameters specify the lower left corner of the viewport, *width* and *height* specify the size.

3D Viewing Pipeline

All transformations necessary to view three-dimensional model data on a two-dimensional screen (viewport) make up the viewing pipeline. In order to save computing power transformation matrices are combined by multiplication into a single matrix:

- Viewing transformations
    1. Translation of the viewpoint backward
    2. Rotation of the camera coordinate system
- Projection Transformations

    3. Orthographic or perspective projection

- Viewport Transformation

    4. Scaling
    5. Translation

$P = T_{vp} * S_{vp} * Proj * R_{vpn} * T_{view}$

Since the multiplication of transformation matrices is not commutative, the order in which the transformations are applied is crucial. The resulting matrix *P* is applied to all vertices of the three-dimensional model data.

Preparation

1. Recapture your knowledge of homogenous coordinates, geometrical transformations, 3D viewing.
2. Make yourself familiar with the OpenGL graphics standard.
3. Analyze structure and functionality of the example program transform.c.
4. Prepare and note the transformation matrices necessary for the following exercises.
5. Answer the following questions:
   Which transformations have to be transformed in order to display a 3D scene on a monitor? What is the correct logical order?
   Why do we use homogenous coordinates in 3D computer graphics and how are they used in order to reduce computation overhead?
   Which two kinds of projection are distinguished?
   How does the display change with parallel projection, when the eyepoint is moved away from the object?
   Which parameters have be specified in order to define the viewing transformation, and how is the camera coordinate system constructed?
   The display shows a rotating object. How can this be achieved? Explain three different approaches.

Exercises

1. Complete the example program transform.c by implementing and applying matrices for the following modelling transformations:

   Rotation of the original cube (at 2.0,2.0,2.0) with an angle of 30 degrees around the x-axis.

   Translation of the original cube such, that its left lower front corner arrives at (1, -1, 3).

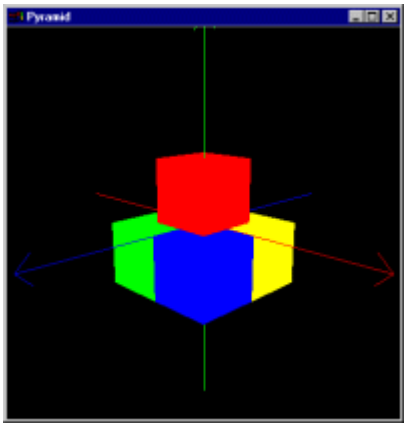
   Distort the original cube such, that a square with length 6, height 0.7 and width 3 is generated.

2. Implement the following combined transformations by (1) matrix operations and (2)OpenGL modelling transformation functions (glScale, glTranslate, glRotate):

   Rotate a cube with lower left front corner at (2, 2, 2) around an axis through the (2,2,2) vertex and parallel to the x-axis with a rotation angle of 45 degrees.

Demonstrate with an arbitrary example, that the multiplication of transformation matrices is not commutative.

3. Construct a pyramid from 5 cubes. Place the eyepoint at (5, 2, 5). View the scene with (1) orthographic and (2) perspective projection. Increase the distance between camera and model by moving the eyepoint. Describe and explain the effect on the resulting image for the two projection techniques. Draw the scene with orthographic projection as front, side and top view.



4. (Optional) Implement the following combined transformation by matrix operations: Rotate a cube with center at (2, 2, 2) around the axis through (-1,0,1),(4,4,-2) with 10 degrees.