Chapter 1

Transformations

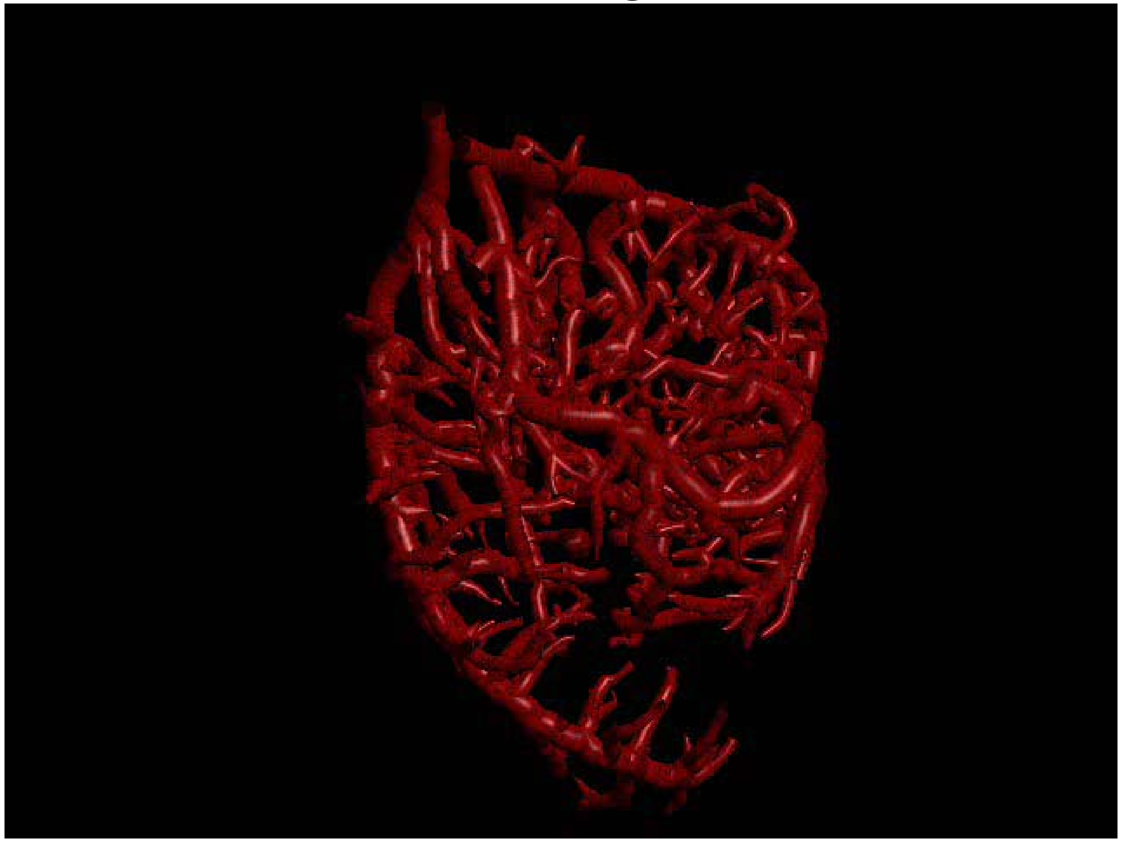
Chapter 1

Transformations are used within the entire viewing pipeline:

- Projection from world to view coordinate system
- View modifications:
  - Panning
  - Zooming
  - Rotation

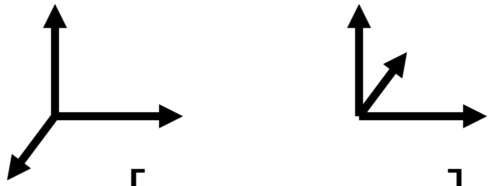
Chapter 1

Transformations can also be used for creating animations to better illustrate the 3D configuration of a model:



Chapter 1

Spaces and Transformations



Left-handed v. right handed

Homogeneous coordinates: $[x \quad y \quad z \quad 1]$

4x4 transformation matrix (TM)

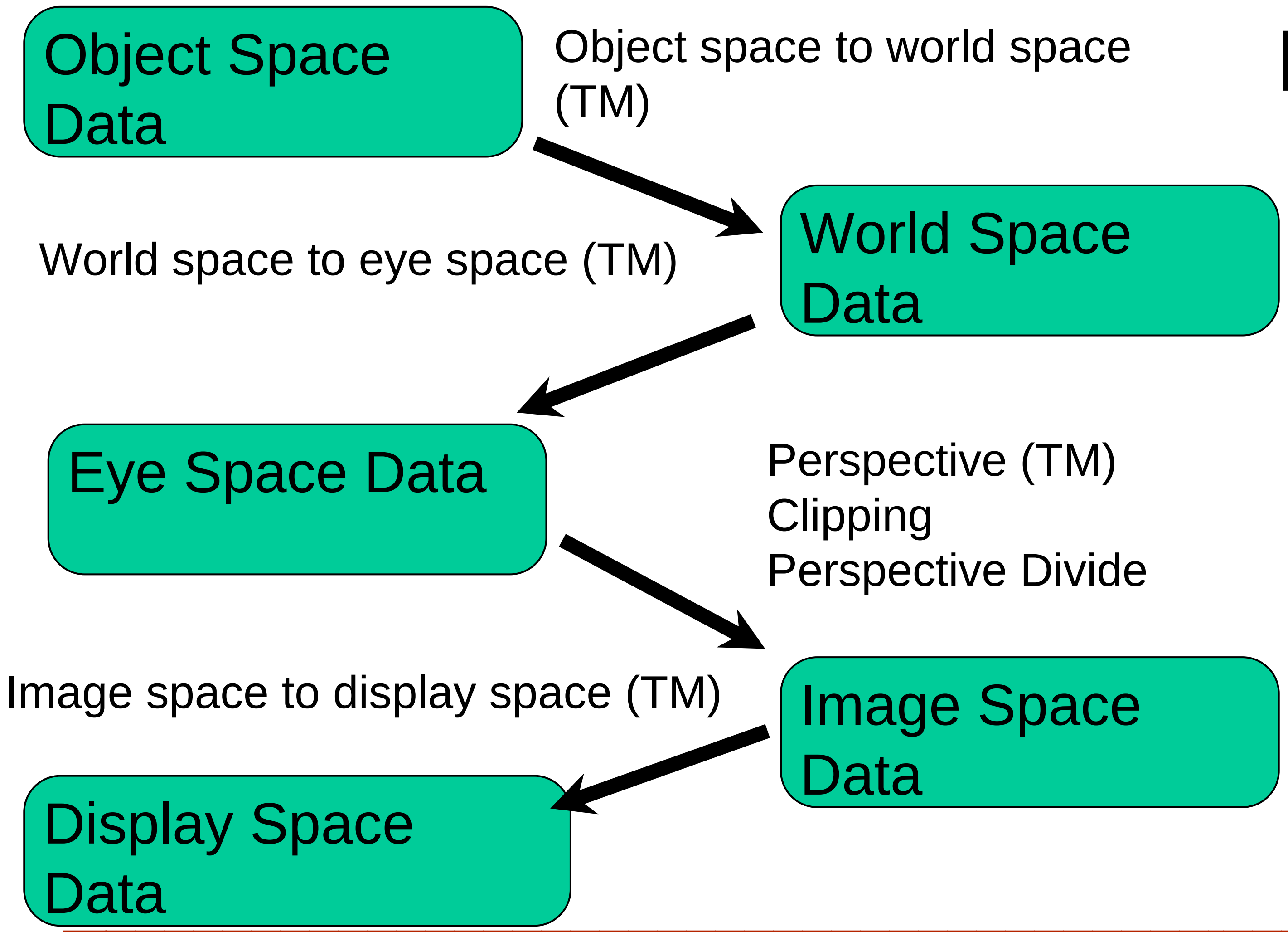
Concatenating TMs

Basic transformations (TMs)

Display pipeline

Chapter 1

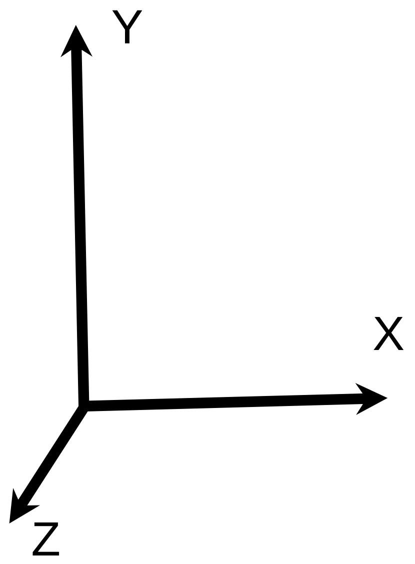
Display Pipeline



Chapter 1

Representing an orientation

Example: fixed angles - rotate around global axes

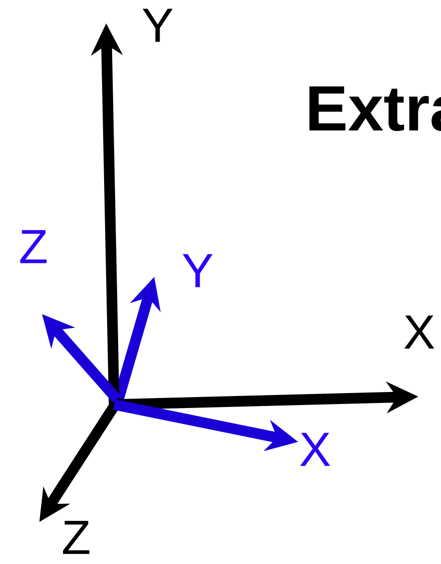


Orientation: $\begin{pmatrix}\alpha & \beta & \gamma\end{pmatrix}$

$$P' = R_z(\gamma)R_y(\beta)R_x(\alpha)P$$

Chapter 1

Working with fixed angles and Rotation Matrices (RMs)



Orthonormalizing a RM

Extracing fixed angles from an orientation

Extracing fixed angles from a RM

Making a RM from fixed angles

Making a RM from transformed unit coordinate system (TUCS)

Chapter 1

Transformations in pipeline

object -> world: often rigid transforms
world -> eye: rigid transforms
perspective matrix: uses 4th component of homo. coords
perspective divide: divide by homo. component
image -> screen: 2D map to screen coordinates
Clipping: procedure that considers view frustum

(rigid transformations preserve angles and distances)

Chapter 1

Error considerations

Accumulated round-off error - transform data:

transform world data by delta RM

update RM by delta RM; apply to object data

update angle; form RM; apply to object data

orthonormalization

rotation matrix: orthogonal, unit-length columns

iterate update by taking cross product of 2 vectors

scale to unit length

considerations of scale

miles-to-inches can exeed single precision arithmetic

Chapter 1

Orientation Representation

Rotation matrix
Fixed angles: rotate about global coordinate system
Euler angles: rotate about local coordinate system
Axis-angle: arbitrary axis and angle
Quaternions: mathematically handy axis-angle 4-tuple
Exponential map: 3-tuple version of quaternions

Chapter 1

Transformation Matrix

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

Chapter 1

Transformation Matrix

$$\begin{bmatrix} a & b & c & t_x \\ e & f & g & t_y \\ i & j & k & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Chapter 1

Rotation Matrices

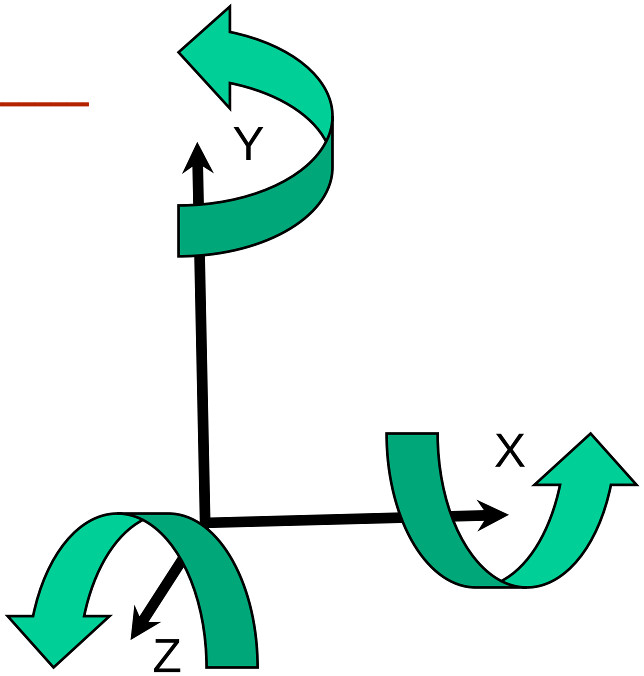
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Chapter 1

Fixed Angles



$$\begin{pmatrix}\alpha & \beta & \gamma\end{pmatrix} \longrightarrow P' = R_z(\gamma)R_y(\beta)R_x(\alpha)P$$

Fixed order: e.g., x, y, z; also could be x, y, x
Global axes

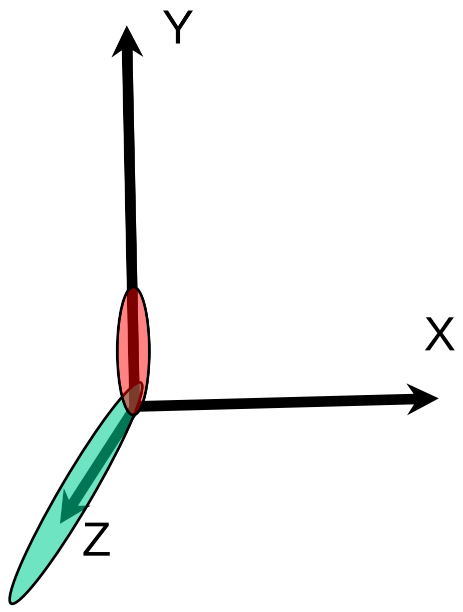
Chapter 1

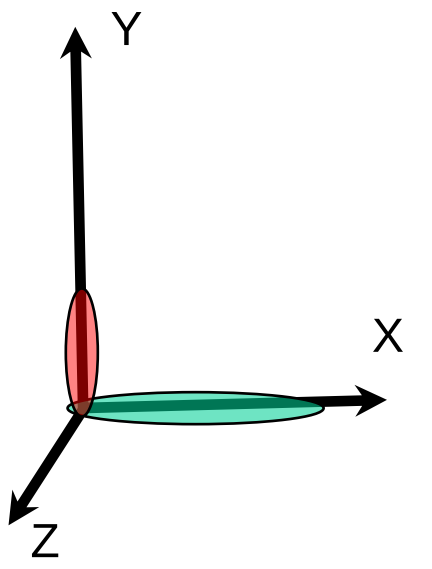
Gimbal Lock

Fixed angle: e.g., x, y, z

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

$$\begin{pmatrix} 0 & 90 & 0 \end{pmatrix}$$





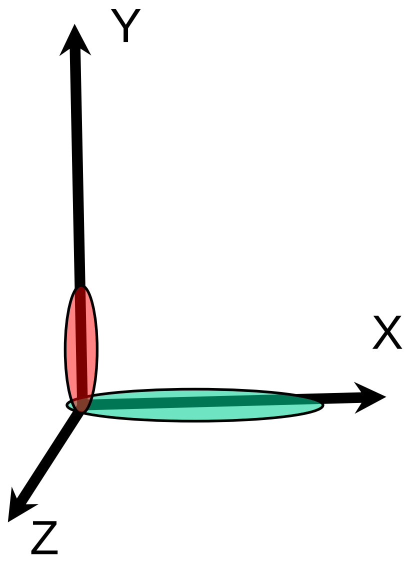
Chapter 1

Gimbal Lock

$$\begin{pmatrix} 0 & 90 & 0 \end{pmatrix}$$

Fixed order of rotations: x, y, z

What do these epsilon rotations do?



$$\begin{pmatrix} 0 \pm \varepsilon & 90 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 90 \pm \varepsilon & 0 \end{pmatrix}$$

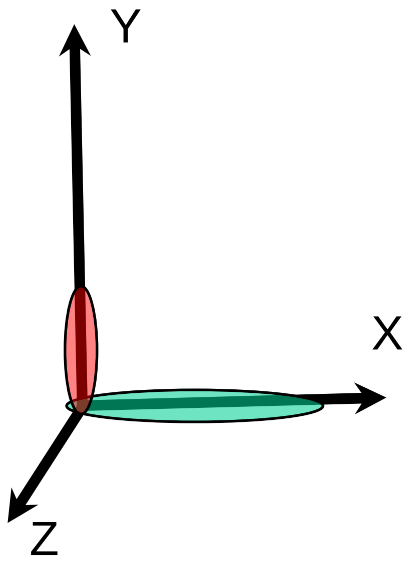
$$\begin{pmatrix} 0 & 90 & 0 \pm \varepsilon \end{pmatrix}$$

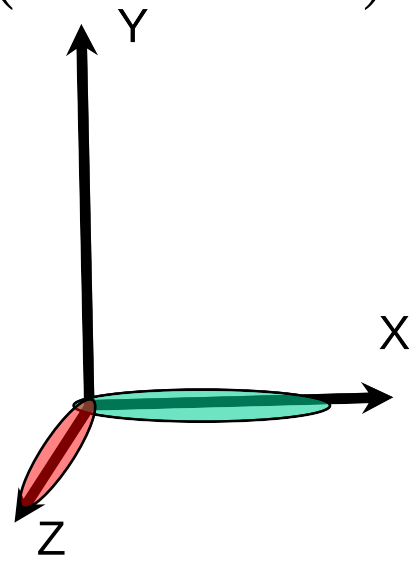
Chapter 1

Gimbal Lock

$\begin{pmatrix} 0 & 90 & 0 \end{pmatrix}$

$\begin{pmatrix} 90 & 0 & 90 \end{pmatrix}$



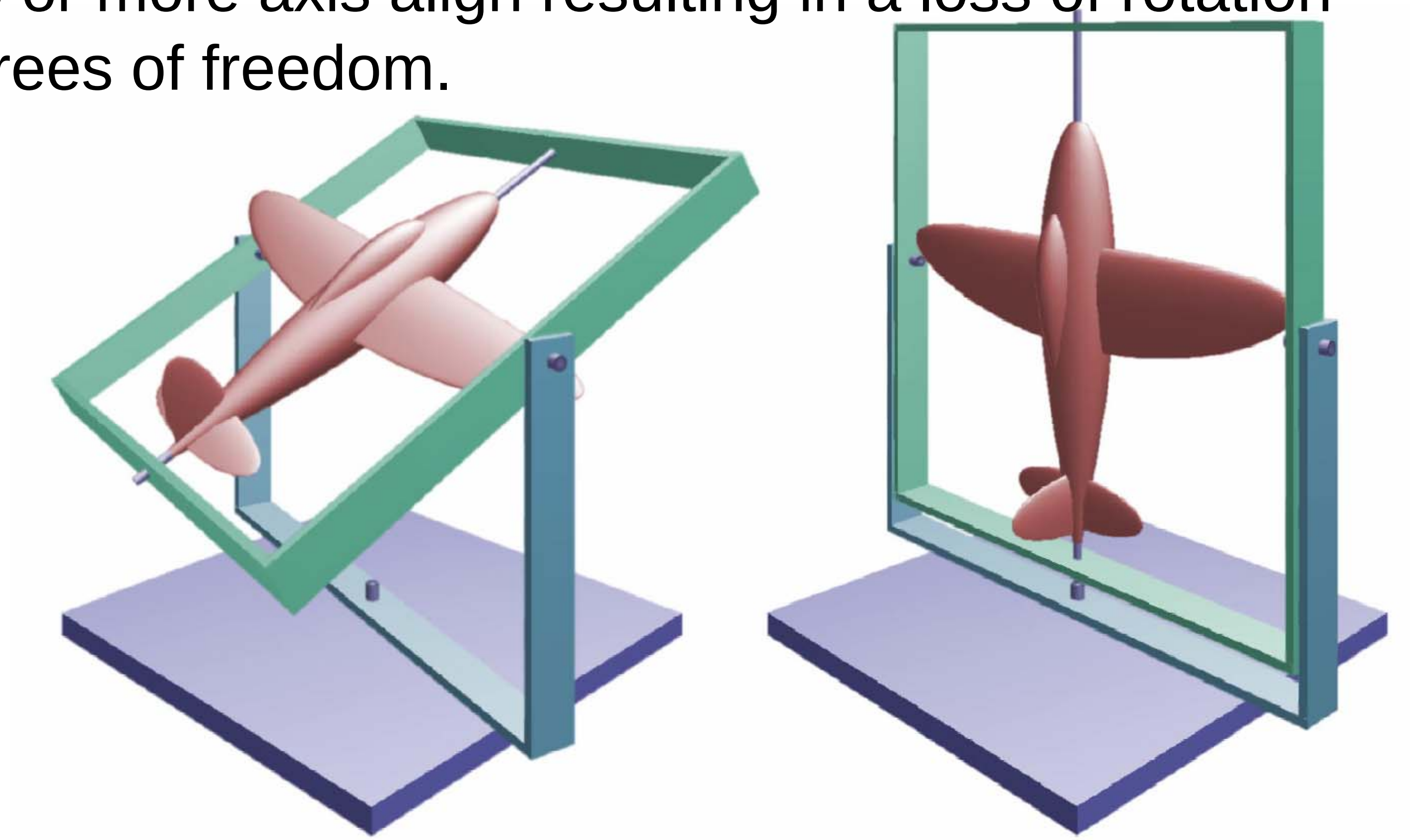


Interpolating FA representations does not produce intuitive rotation because of gimbal lock

Chapter 1

Gimbal Lock

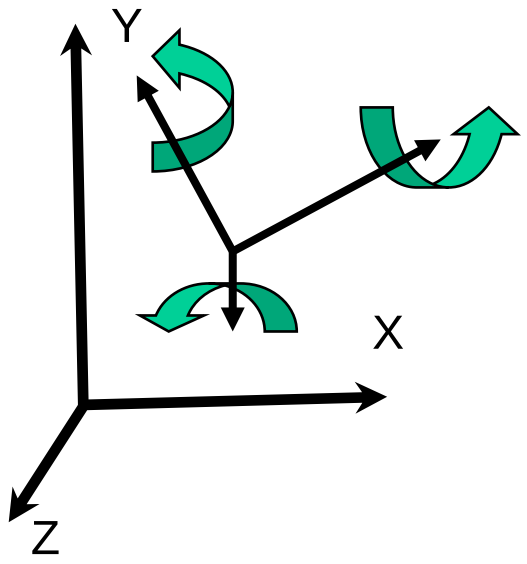
Two or more axis align resulting in a loss of rotation degrees of freedom.



Chapter 1

Euler Angles

$$\begin{pmatrix}\alpha & \beta & \gamma\end{pmatrix}$$



Prescribed order: e.g., x, y, z or x, y, x
Rotate around (rotated) local axes

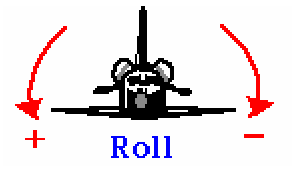
Note: fixed angles are same as Euler angles applied in reverse order and vice versa

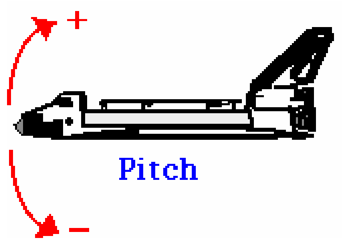
$$\begin{pmatrix}\alpha & \beta & \gamma\end{pmatrix} \longrightarrow P' = R_x(\alpha)R_y(\beta)R_z(\gamma)P$$

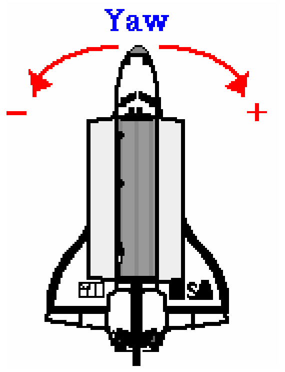
Chapter 1

Roll, pitch, and yaw (Euler angles)

The orientation of, for example, a space shuttle in space is defined as its **attitude**. Orbiter attitudes are specified using values for **pitch, yaw, and roll**. These are with respect to the local object's coordinate system and therefore represent Euler angles





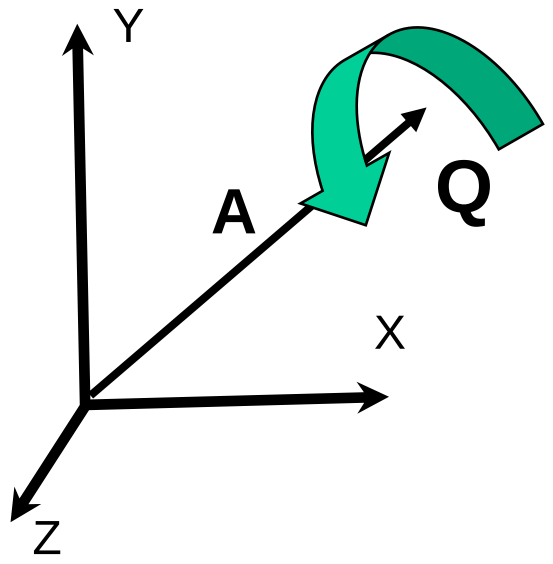


http://liftoff.msfc.nasa.gov/academy/rocket_sci/shuttle/attitude/pyr.html

Chapter 1

Axis-Angle

$$\begin{bmatrix}\theta & A\end{bmatrix}$$
$$\begin{bmatrix}\theta & \begin{pmatrix}x & y & z\end{pmatrix}\end{bmatrix}$$



Rotate about given axis

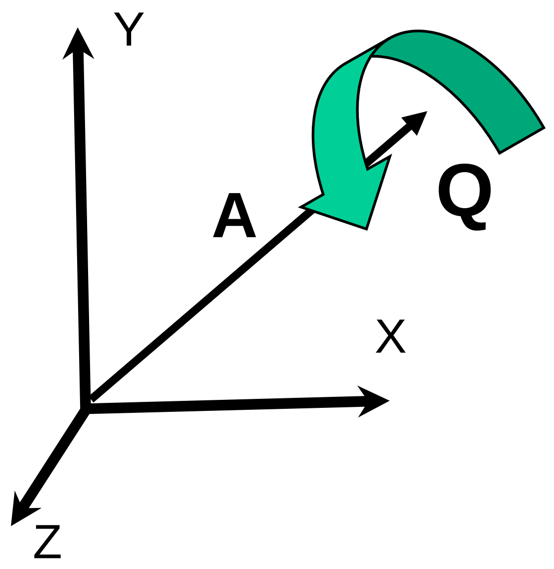
Euler's Rotation Theorem (orientation can be derived from another by a single rotation about an axis)

Fairly easy to interpolate between orientations

Difficult to concatenate rotations

Chapter 1

Axis-angle to rotation matrix



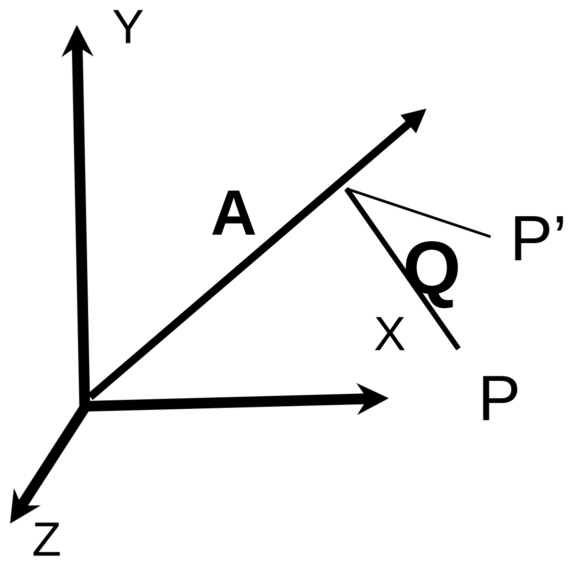
Concatenate the following:
Rotate A around z to x-z plane
Rotate A’ around y to x-axis
Rotate theta around x
Undo rotation around y-axis
Undo rotation around z-axis

Chapter 1

Axis-angle to rotation matrix

$$\hat{A} = \begin{bmatrix} a_x a_x & a_x a_y & a_x a_z \\ a_y a_x & a_y a_y & a_y a_z \\ a_z a_x & a_z a_y & a_z a_z \end{bmatrix}$$

$$A^* = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$



$$Rot_{(\theta \ (x \ \ y \ \ z))} = \hat{A} + \cos(\theta)(I - \hat{A}) + \sin(\theta) A^*$$

Chapter 1

Quaternions

A quaternion is a four-tuple of real numbers, *[s, x, y, z]* or, equivalently *[s, v]*, consisting of a scalar, *s*, and a three-dimensional vector, *v*.

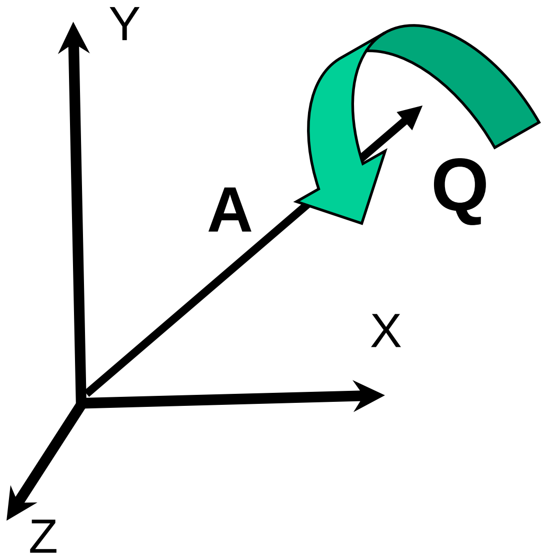
The quaternion is an alternative to the axis and angle representation in that it contains the same information in a different, nut mathematically convenient, form.

Importantly, it is a form that can be interpolated as well as used in concatenating a series of rotations into a single representation. The axis and angle information of a quaternion can be viewed as an orientation of an object.

Chapter 1

Quaternion

$$Rot_{(\theta \quad A)} = \left[\cos\left(\frac{\theta}{2}\right) \quad \sin(\frac{\theta}{2}) * A \right]$$



Same as axis-angle, but different form
Still rotate about given axis
Mathematically convenient form

$$q : \ [s \quad v]$$

Note: in this form v is a scaled version of the given axis of rotation, A

Chapter 1

Quaternion Arithmetic

Addition $[s_1 + s_2 \quad v_1 + v_2] = [s_1 \quad v_1] + [s_2 \quad v_2]$

Multiplication

$$q_1 q_2 = [s_1 s_2 - v_1 \cdot v_2 \quad s_2 v_1 + s_1 v_2 + v_1 \times v_2]$$

Inner Product $q_1 \cdot q_2 = s_1 s_2 + v_1 \cdot v_2$

Length $\|q\| = \sqrt{q \cdot q}$

Chapter 1

Quaternion Arithmetic

Inverse $$q^{-1} = \frac{1}{\|q\|^2} \begin{bmatrix} s & -v \end{bmatrix}$$

$$qq^{-1} = q^{-1}q = \begin{bmatrix} 1 & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \end{bmatrix}$$

$$(pq)^{-1} = q^{-1} p^{-1}$$

Unit quaternion $$\hat{q} = \frac{q}{\|q\|^2}$$

Chapter 1

Quaternion Represention

Rotating a vector using quaternions:

Vector $\begin{bmatrix} 0 & v \end{bmatrix}$

Transform $v' = Rot_q(v) = qvq^{-1}$

Chapter 1

Quaternion Geometric Operations

$$Rot_q(v) = Rot_{-q}(v)$$

$$Rot_q(v) = Rot_{kq}(v)$$

$$v'' = Rot_q(Rot_p(v)) = Rot_{qp}(v)$$

$$v'' = Rot_{q^{-1}}(Rot_q(v)) = q^{-1}(qvq^{-1})q = v$$

Chapter 1

Unit Quaternion Conversions

$$Rot_{[s\quad v]} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy-2sz & 2xz-2sy \\ 2xy-2sz & 1-2x^2-2z^2 & 2yz-2sx \\ 2xz-2sy & 2yz-2sx & 1-2x^2-2y^2 \end{bmatrix}$$

Axis-Angle

$$\begin{cases} \theta = 2\cos^{-1}(s) \\ (x, y, z) = v / \|v\| \end{cases}$$

Chapter 1

Rotations using Quaternions

Hence, a rotation of angle θ around the axis (x, y, z) can be described using the following quaternion:

$$[\cos(\theta/2), \sin(\theta/2)(x, y, z)]$$

Chapter 1

Quaternions

Avoids gimbal lock
Easy to rotate a point
Easy to convert to a rotation matrix
Easy to concatenate – quaternion multiply
Easy to interpolate – interpolate 4-tuples
How about smoothly (in both space and time) interpolate?

Chapter 1

Viewing in OpenGL

As seen before, viewing and projections is achieved by transforming from the world coordinate system to the display coordinate system using matrix multiplication. Hence, OpenGL provides several functions for modifying matrices.

Since OpenGL is a state machine, it has two different **matrix stacks** that can change the view onto a scene (set of objects). The first one is the projection stack, while the other one is the modelview stack. The projection transformation is responsible for the projection just like a lens for a camera. This transformation also determines the type of projection (e.g. perspective or orthographic).

Chapter 1

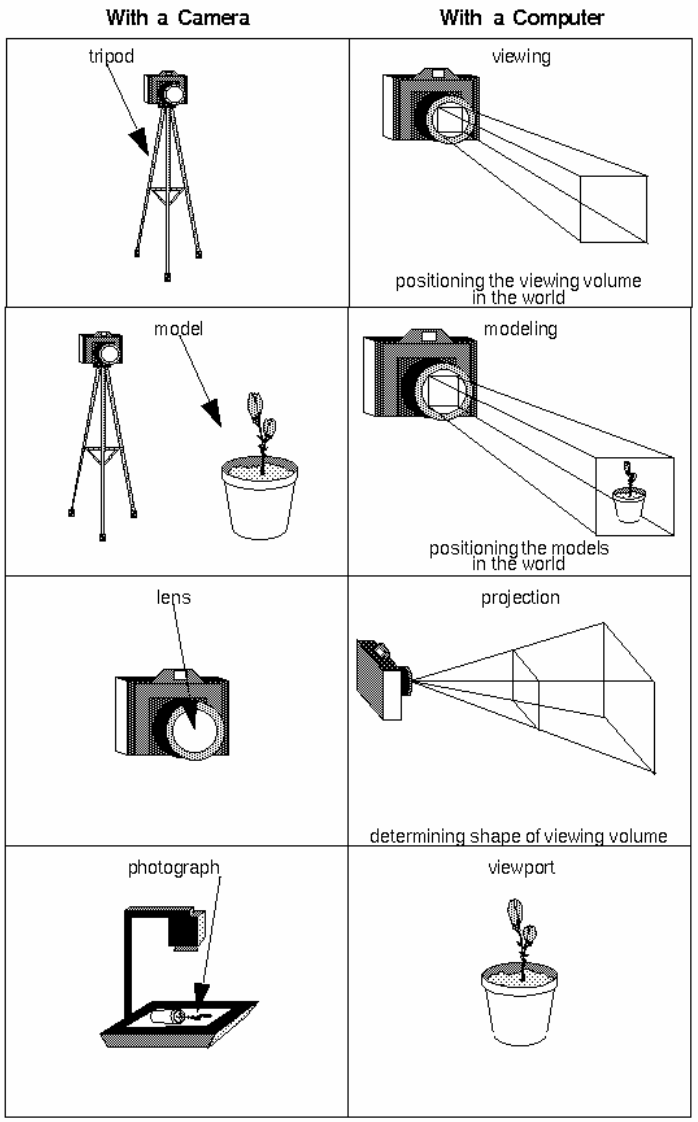
Viewing in OpenGL (continued)

The modelview transformation combines the view transformation and the model transformation onto the same stack. The view transformation indicates the shape of the available screen (width, height, ratio). The model transformation facilitates the change of the entire scene as a whole before mapping it onto the projection plane. For example, the model transformation can be used to rotate the entire scene or zoom in or out.

Chapter 1

Viewing in OpenGL (continued)

OpenGL mainly follows the analogy to a camera when creating an image on the display.


With a Camera
With a Computer
tripod
viewing
positioning the viewing volume in the world
model
modeling
positioning the models in the world
lens
projection
determining shape of viewing volume
photograph
viewport

Chapter 1

Viewing in OpenGL (continued)

To specify which stack you want to modify, OpenGL provides a method:

```
   glMatrixMode (GLenum mode);
```

The mode passed onto this function as the only argument can be specified as `GL_MODELVIEW` or `GL_PROJECTION`.

This then changes the state of OpenGL, so that all following matrix commands change that specific matrix only.

OpenGL uses homogenous coordinates to represent matrices, i.e. all matrices are 4x4 matrices.

Chapter 1

Viewing in OpenGL (continued)

To initialize a matrix stack with the identity matrix, the following functions can be used:

```
   glLoadIdentity ();
```

This then initializes the current matrix stack with the matrix

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

Usually, this function is used before any other matrix modification, since it just overwrites the current matrix.

Chapter 1

Viewing in OpenGL (continued)

Alternatively, you initialize the matrix with a specific matrix that was calculated before. The following functions overwrites the current matrix stack with the given matrix:

```
float mf[16];
double md[16];
glLoadMatrixf (mf);
glLoadMatrixd (md);
```

The elements of the matrices are specified as shown on the right:

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix}$$

Chapter 1

Viewing in OpenGL (continued)

To multiply a matrix onto the current one the following functions are useful:

```
glMatrixMultf (m);
glMatrixMultd (m);
```

The matrix is specified exactly the same as for the function `glLoadMatrix`.

Note that OpenGL multiplies the new matrix M to the current one C from the right, i.e. after applying the function `glMatrixMult` the matrix on the current stack will be $C \cdot M$.

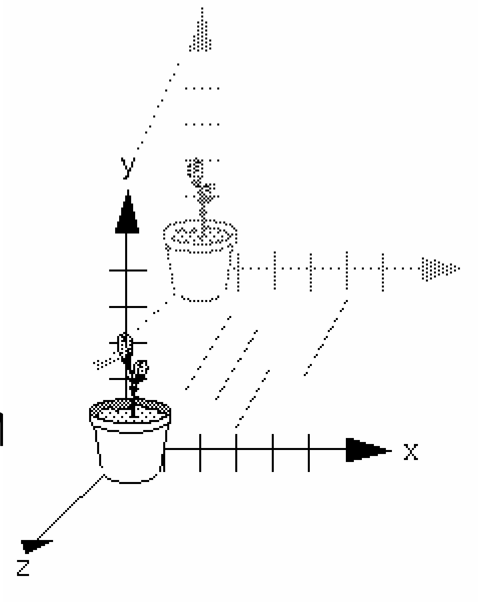
Chapter 1

Viewing in OpenGL (continued)

OpenGL also provides function for the basic types of transformation, i.e. translation, rotation, and scaling. The function

```
    glTranslatef (GLfloat x,
                  GLfloat y,
                  GLfloat z);
```

multiplies a translation matrix onto the current matrix stack using the translation vector *(x, y, z).*

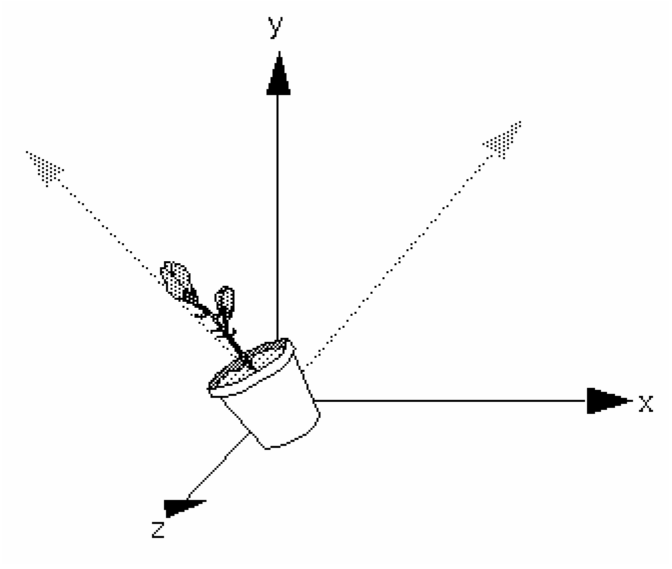


Chapter 1

Viewing in OpenGL (continued)

The function

```
   glRotatef (GLfloat angl
              GLfloat x,
              GLfloat y,
              GLfloat z);
```



multiplies a rotational matrix onto the current matrix stack,

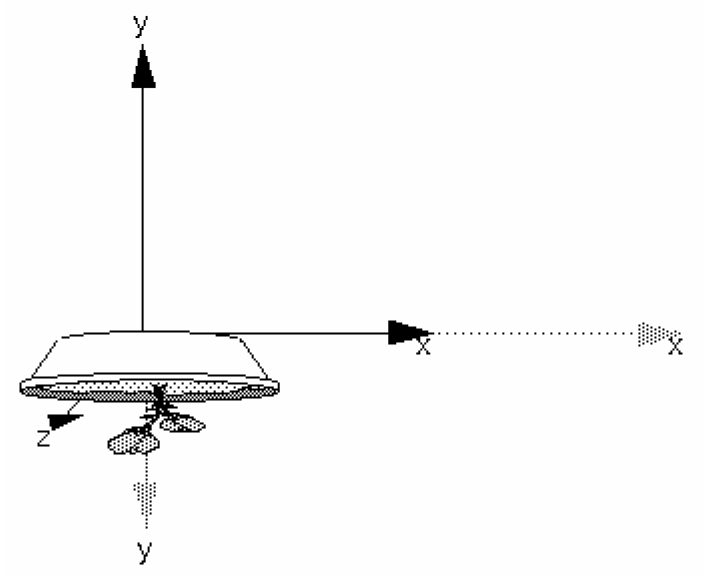
Chapter 1

Viewing in OpenGL (continued)

while the function

```
   glScalef (GLfloat x,
             GLfloat y,
             GLfloat z);
```

appends a scaling matrix to the matrix stack.



Chapter 1

Viewing in OpenGL (continued)

Note, that the corresponding functions that accept `double` values are also available. These use – according to the usual OpenGL convention – the suffix `d` instead of `f` to indicate the data type.

Using these matrix functions, both the projection as well as the modelview matrices can be specified.

OpenGL, however, provides some functions that are more convenient and intuitive.

Chapter 1

Viewing in OpenGL (continued)

The function `gluLookAt` can be used to specify the camera location and orientation:

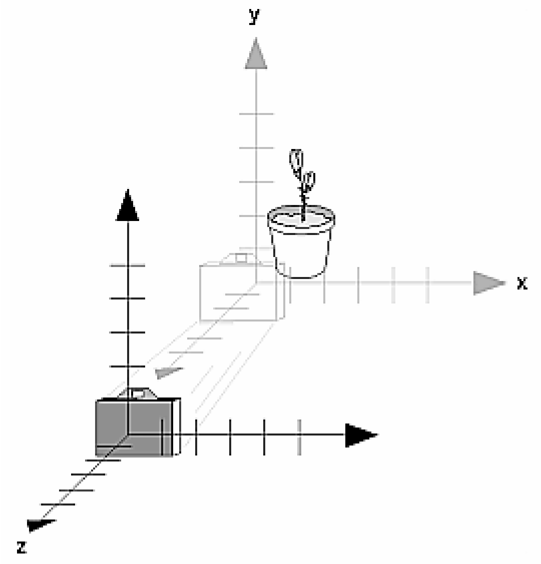
```
void gluLookAt (GLdouble eyex,
                GLdouble eyey,
                GLdouble eyez,
                GLdouble centerx,
                GLdouble centery,
                GLdouble centerz,
                GLdouble upx,
                GLdouble upy,
                GLdouble upz);
```

Chapter 1

Viewing in OpenGL (continued)

The arguments for the function `gluLookAt` specify the view coordinate with respect to the camera. The location of the camera or eye defines the origin, while the center point determines the direction the camera is pointing at.

Hence, eye-center determines the z-axis. The vector up identifies the y-axis, while the x-axis is orthogonal to the y- and z-axis.



The default it `gluLookAt (0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);`

Chapter 1

Projections in OpenGL

OpenGL provides built-in functions for perspective and orthogonal projections. These can be applied directly after changing the state to make the projection matrix the current matrix stack and initializaing:

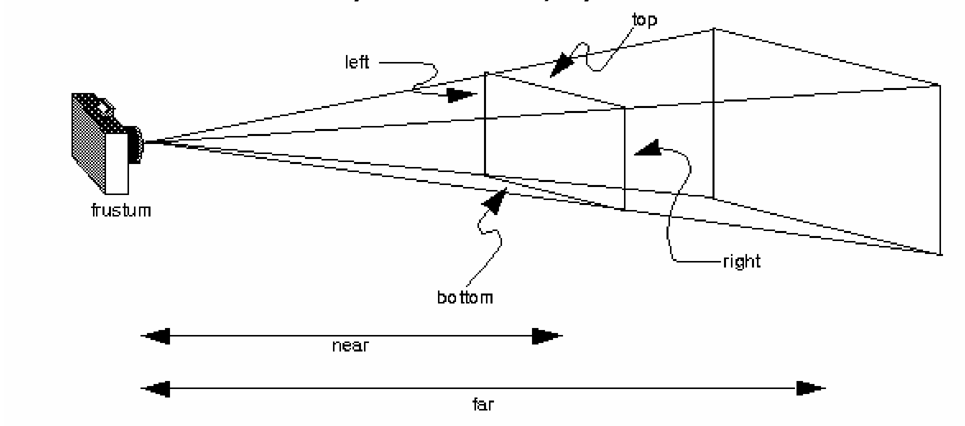
```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
```

Chapter 1

Projections in OpenGL (continued)

Using the function `glFrustum`, the view frustum can be declared using a perspective projection (all arguments are of the type `GLdouble`):

```
   glFrustum (left, right,
              bottom, top,
              near, far);
```

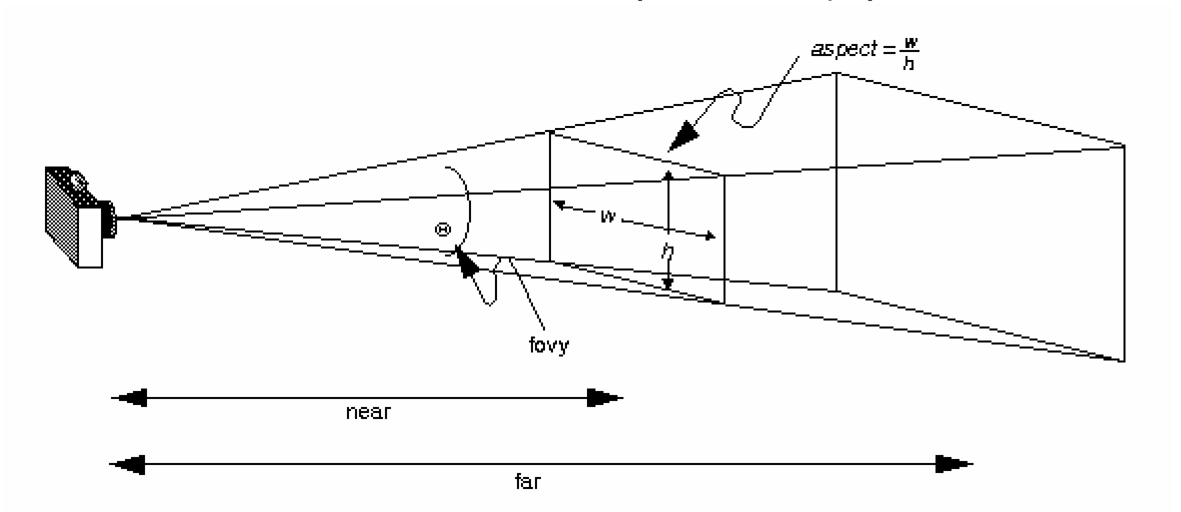


Chapter 1

Projections in OpenGL (continued)

Sometimes it is more convenient to specify the view frustum following the camera analogy more closely (all arguments are of type `GLdouble`):

```
gluPerspective (fovy, aspect,
                          near, far);
```

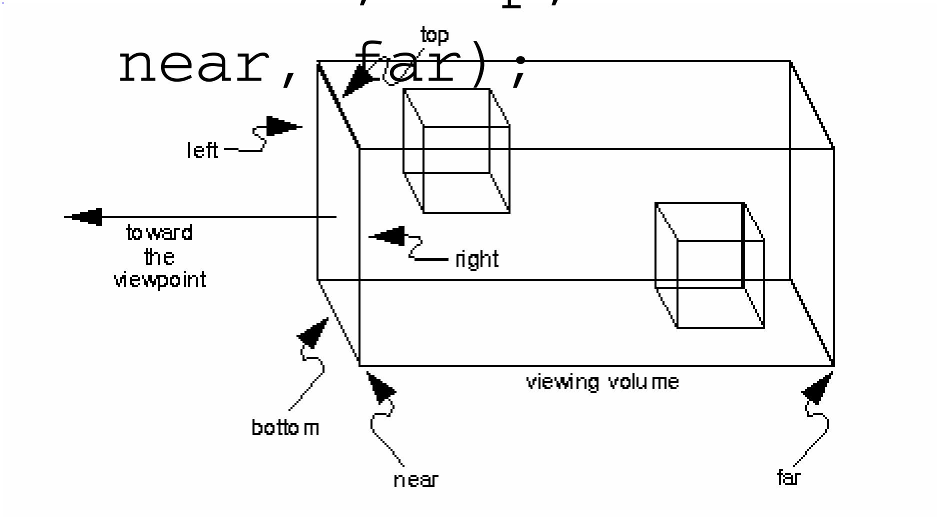


Chapter 1

Projections in OpenGL (continued)

If an orthogonal projection is desired, the following method can be used (all arguments are of type `GLdouble`):

```
   glOrtho (left, right,
                 bottom, top,
                 near, far);
```



Chapter 1

Projections in OpenGL (continued)

For a 2-D projection it does not make any difference if a perspective or orthogonal projection is used since the scene with all the objects does not have any depth.

Hence, there is only one function provided by OpenGL for a 2-D projection (all arguments are of type `GLdouble`):

```
glOrtho2D (left, right,
           bottom, top);
```

Chapter 1

Viewing in OpenGL

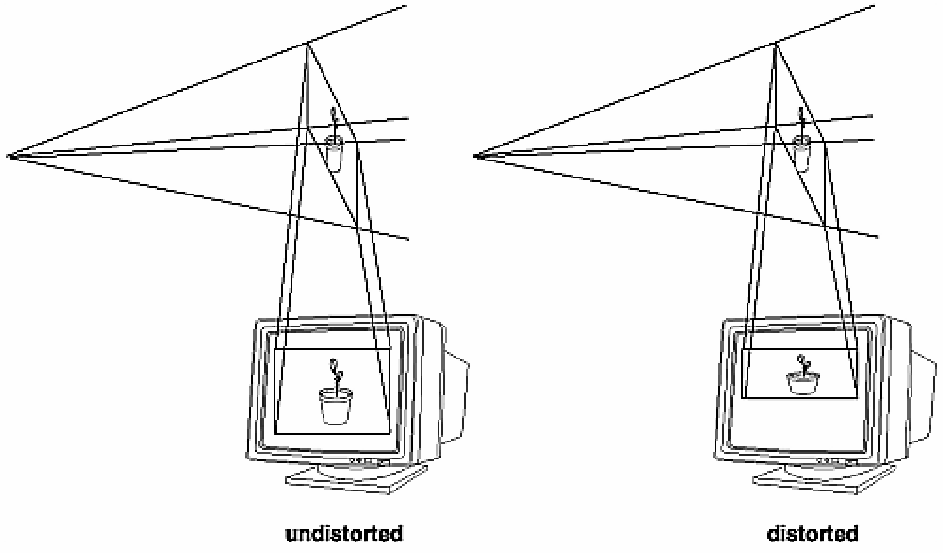
The last step of the process of creating an image on a computer display is the viewport transformation. Recalling the camera analogy, the viewport transformation corresponds to the stage where the size of the developed photography is chosen. The viewport is measured in window coordinates. By default, OpenGL uses the entire window provided. The following functions allows you to reduce the size of the image (all arguments are of type `GLint`):

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

Chapter 1

Viewing in OpenGL

The aspect ratio of a viewport should generally equal the aspect ratio of the viewing volume. If the two ratios are different, the projected image will be distorted as it's mapped to the viewport. Note that subsequent changes to the size of the window don't explicitly affect the viewport. Your application should detect window resize events and modify the viewport and projection appropriately.


undistorted
distorted