Chapter 3

Transformation and projection

3.1 Overview

The basis for generating images on a display or other output devices are coordinate transformation in 2-D or 3-D.

Generally, we distinguish between the coordinate system of the objects (world coordinate system) and the coordinate system of the output device (display coordinate system).

The world coordinate system is often defined by geometric properties for objects, e.g. special directions, symmetries, …

The display coordinate system is defined by the output device itself (e.g. origin at lower left corner, axes parallel to the edges of boundary).

3.1 Overview

Using coordinate transformations (translation, scaling, rotation), we can transform the world coordinate system into the display coordinate system.

Generally, we assume that the coordinate systems are orthonormalized (Cartesian) coordinate systems.

Usual way of finding a coordinate transformation:

1. Define a coordinate system for the output device (usually based on the pixel grid)
2. Define a world coordinate system for the objects
3. Project the world coordinates onto an image plane using parallel or perspective projection

3.1 Overview

4. The display coordinate system usually is not identical to the projection of the world coordinate system, therefore requiring another coordinate transformation

Assume we have two coordinate systems S (e.g. display coordinate system) and S' (world coordinate system) as $S=(O, x_1, x_2)$ and $S'=(O', x_1', x_2')$.

3.2 Planar transformations

Translation

The simplest transformation between the coordinate systems S and S' is a translation. We presume that the coordinate axes of the two systems are mutually parallel.

To transform from one coordinate system to the other, every point is moved along a constant vector.

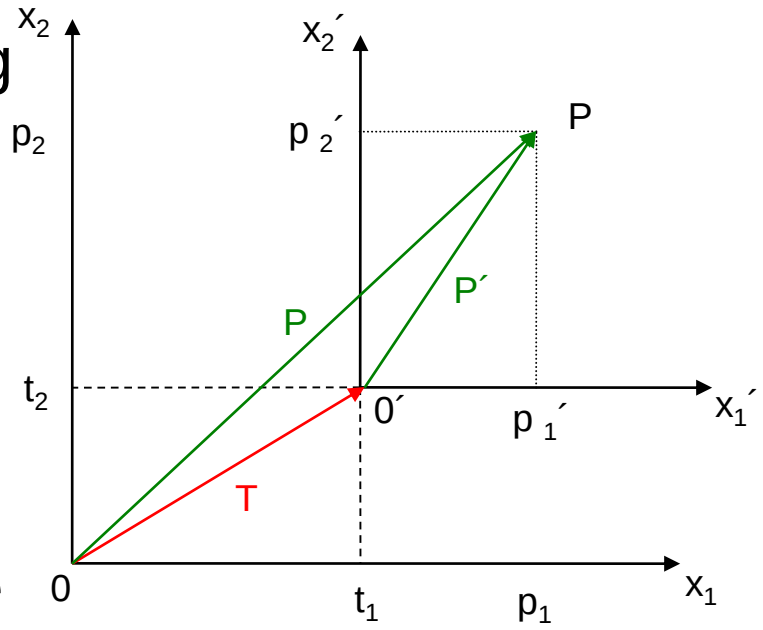
3.2 Planar transformations

Translation (continued)

Thus, for every point the following equation holds:

$$p_1 = t_1 + p_1'$$
$$p_2 = t_2 + p_2'$$

Here, the translation vector (t_1, t_2) corresponds to the origin of S' represented in coordinates of S. Hence, the point P defined as the point (p_1', p_2') with respect to S' also has the coordinates $(t_1 + p_1', t_2 + p_2')$ in S:



$$\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} + \begin{pmatrix} p_1' \\ p_2' \end{pmatrix} \quad \text{or short} \quad P = T + P'$$

3.2 Planar transformations

Rotation

Assuming that the two coordinate systems S and S' have the same origin, a rotation with angle results in the following equations:

$$\frac{l}{p_2'} = \sin\varphi \quad \text{and} \quad \frac{L}{p_1'} = \cos\varphi$$

$$\text{Hence}: \quad p_1 = L - l = p_1' \cdot \cos\varphi - p_2' \cdot \sin\varphi$$

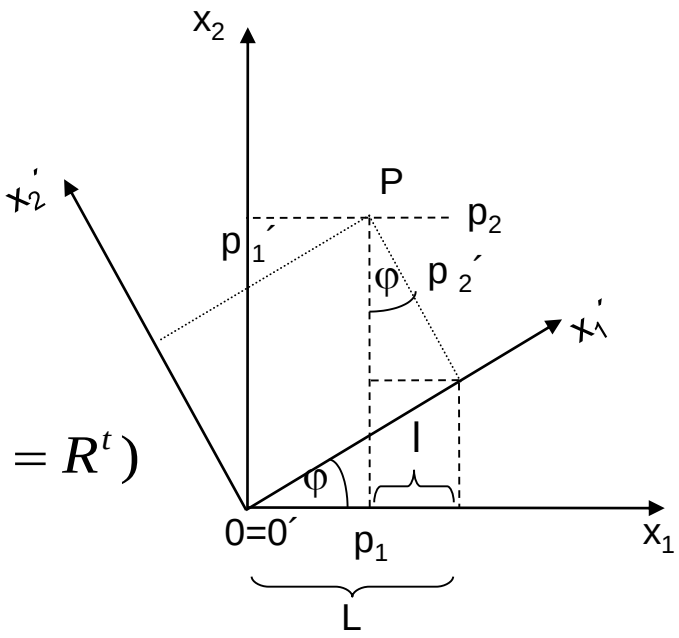
$$\text{and} \quad p_2 = p_1' \cdot \sin\varphi + p_2' \cdot \cos\varphi$$

Using vector - matrix notation, we get :

$$P = R \cdot P'$$

where R is an orthonormal matrix (i.e. $R^{-1} = R^t$)

$$R = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix}$$



3.2 Planar transformations

Rotation (continued)

Note: different interpretations of $P=R\cdot P'$:

- R transforms the coordinates (p_1', p_2') of a point with respect to the coordinate system S' into a representation with respect to S. This corresponds to interpreting S as a global coordinate system where applying the matrix R to (p_1', p_2') rotates a point.
- The coordinate system S is transformed (by rotating mathematically positively) into the coordinate system S'. A given point's coordinates (p_1', p_2') are then represented with respect to the coordinate system S'.

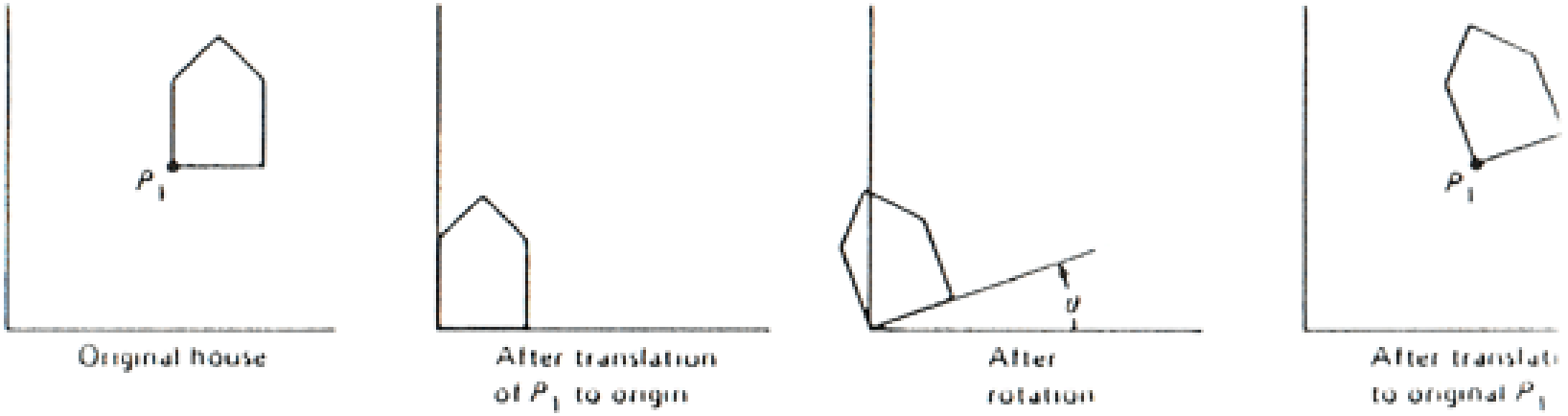
The same interpretations are valid for other types of transformations.

3.2 Planar transformations

Rotation (continued)

When rotating with an arbitrary point as rotational center, we need to include two additional translations:

1. Translate P_1 onto the origin
2. Rotate around the origin
3. Translate back to P_1 using the inverse translation from step 1.



3.2 Planar transformations

Rotation (continued)

Comments:

Multiplying matrices is generally not commutative, i.e. $A \cdot B \neq B \cdot A$.

Special care has to be taken to ensure that the order in which the matrices are applied in the correct order when using more than one rotations.

3.2 Planar transformations

Scaling

If the coordinate system S' is to be "enlarged" or "downsized" we can scale it accordingly:

$$p_1 = \lambda_1 \cdot p_1'$$

$$p_2 = \lambda_2 \cdot p_2'$$

Using vector - matrix notation, we get :

$$P = \tilde{S} \cdot P'$$

with a scaling matrix $\tilde{S}$

$$\tilde{S} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

3.2 Planar transformations

Shearing

A shearing transformation can be achieved by applying:

$$p_1 = p_1' + s_1 \cdot p_2'$$

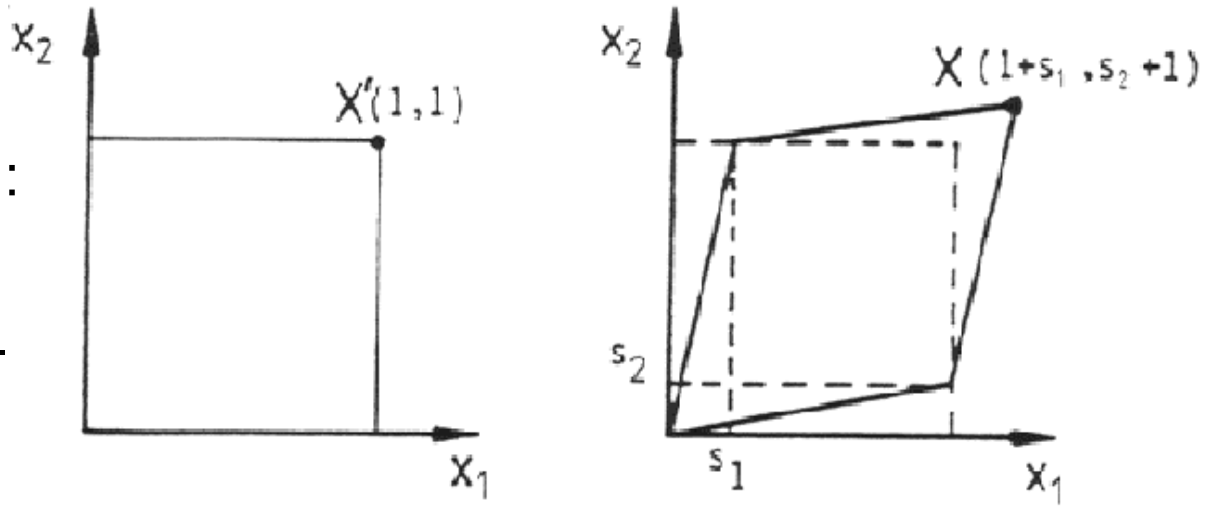
$$p_2 = s_2 \cdot p_1' + p_2$$

Using vector - matrix notation:

$$P = S \cdot P'$$

with the shear - ing matrix S

$$S = \begin{pmatrix} 1 & s_1 \\ s_2 & 1 \end{pmatrix}$$



Scherung in der Ebene

3.2 Planar transformations

Affine transformations

Affine transformations can be described as a linear mapping combined with a translation:

The previous transformations (translation, rotation, scaling, and shearing are examples of affine transformations.

Affine invariance of subdivisions

For every affine transformation F and points P and Q, the following equation holds:

$$F(\lambda \cdot P+(1-\lambda) \cdot Q)= \lambda \cdot F(P)+(1-\lambda) \cdot F(Q) \text{ for } 0\leq \lambda \leq 1$$

3.2 Planar transformations

Affine transformations (continued)

This equation shows, that the image of a straight-line connecting P and Q remains a straight-line after applying F and that the ratios of subdivisions $\lambda:(1-\lambda)$ are preserved.

Thus, it is sufficient to just map the points P and Q and then interpolate in between $F(P)$ and $F(Q)$.

Note that parallel lines remain parallel after applying affine transformations.

3.2 Planar transformations

Affine transformations (continued)

More affine transformations:

Mirroring along a straight-line x=y: $F = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Mirroring along a straight-line x=-y: $F = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$

Mirroring along the x-axis: $F = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$

Mirroring along the y-axis: $F = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

Mirroring with respect to the origin: $F = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$

3.2 Planar transformations

Homogeneous coordinates

Homogeneous coordinates emerged from projective geometry. Here, however, we will introduce a different motivation:

When concatenating rotation, translation, and scaling, we get the following equation: $P = \tilde{S}(T + R \cdot P')$

If we want to combine several of these transformations, the addition in this equation complicates things.

Since matrix multiplications are supported by the graphics hardware nowadays, it would be beneficial to represent all transformations by matrices, i.e.:

$$P = M_n \cdot \ldots \cdot M_3 \cdot M_2 \cdot M_1 \cdot P'$$

3.2 Planar transformations

Homogeneous coordinates (continued)

This can be achieved by transitioning to a higher dimension:

The triple (x, y, w), $w \neq 0$ represents the homogeneous coordinates of the point $(x/w, y/w)$.

Since there are infinitely many representations of the same point, we use the normalized representation with $w = 1$.

Hence, a point $P = (x, y)$ is represented in homogeneous coordinates by $(x, y, 1)$.

Note: we can achieve the same in 3-D by adding another dimension

3.2 Planar transformations

Homogeneous coordinates (continued)

This then allows for a different representation of the translation:

Translation of a point *(x', y')* by a vector (t_1, t_2):

$$\begin{pmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x'+t_1 \\ y'+t_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Rotation of a point *(x', y')* at angle φ:

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

3.2 Planar transformations

Homogeneous coordinates (continued)

Scaling of a point (x', y') using the factors λ_1 and λ_2:

$$\begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \lambda_1 \cdot x' \\ \lambda_2 \cdot y' \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Rotation of a point (x', y') around an arbitrary point P_1 at angle φ:

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

3.3 Transformations in 3-D

Translation

To move a point (x', y', z') along a translation vector (t_x, t_y, t_z) results in the point (x, y, z) when applying the following matrix:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{T(t_x, t_y, t_z)} \cdot \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} x' + t_x \\ y' + t_y \\ z' + t_z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

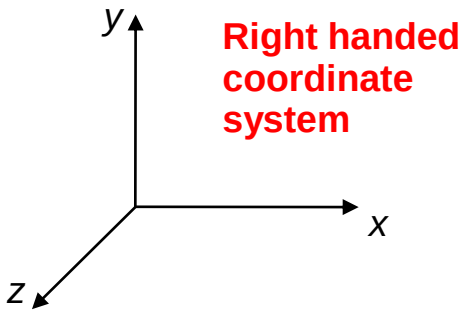
3.3 Transformations in 3-D

Scaling

Scaling using the three scaling factors s_1, s_2, and s_3 can be achieved using the following matrix:

$$\begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_1 \cdot x' \\ s_2 \cdot y' \\ s_3 \cdot z' \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

3.3 Transformations in 3-D



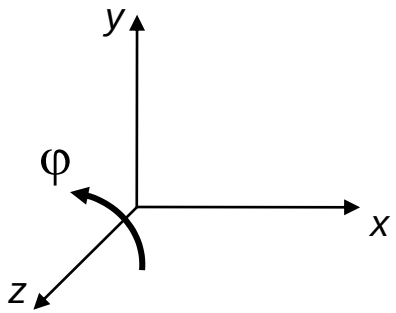
Rotation

- All rotations are mathematically positive, i.e. counter-clockwise
- The observer “sits” along one of the coordinate axis and looks towards the origin of the coordinate system
- We first consider rotations around a single coordinate axis at angle φ

  → transformation matrix $R_x(\varphi)$, $R_y(\varphi)$, $R_z(\varphi)$
- Think of a fixed coordinate system where points are transformed (rotated).

3.3 Transformations in 3-D

Rotation around the z-axis



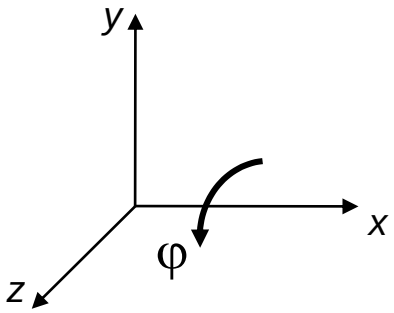
When rotating a point (x', y', z') around the z-axis at angle φ, we get the point (x, y, z):

$$\underbrace{\begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{R_z(\varphi)} \cdot \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} x' \cdot \cos\varphi - y' \sin\varphi \\ x' \sin\varphi + y' \cos\varphi \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Note that this rotation is similar to the 2-D case with a fixed z-coordinate.

3.3 Transformations in 3-D

Rotation around the x-axis



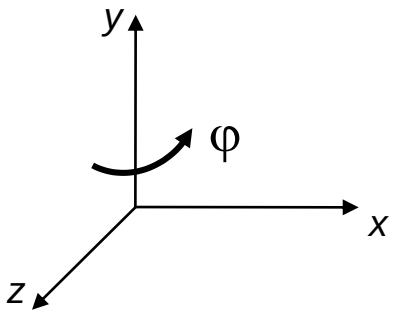
When rotating a point (x', y', z') around the x-axis at angle φ, we get the point (x, y, z):

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{R_x(\varphi)} \cdot \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \cdot \cos\varphi - z' \sin\varphi \\ y' \sin\varphi + z' \cos\varphi \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Here, the x-coordinate is left unchanged so that the matrix describes a rotation within the y-z-plane.

3.3 Transformations in 3-D

Rotation around the y-axis



When rotating a point (x', y', z') around the y-axis at angle φ, we get the point (x, y, z):

$$\underbrace{\begin{pmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{R_y(\varphi)} \cdot \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} x'\cos\varphi + z'\sin\varphi \\ y' \\ -x'\sin\varphi + z'\cos\varphi \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

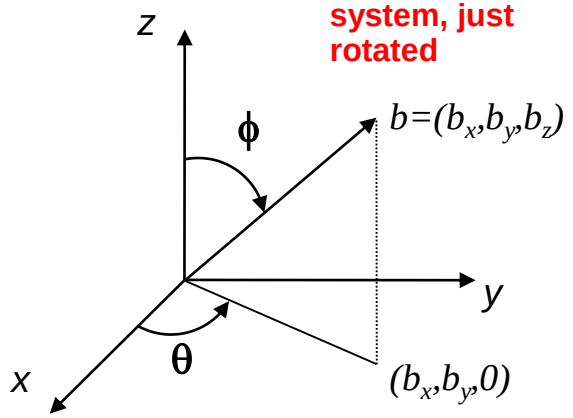
Here, the y-coordinate is left unchanged so that the matrix describes a rotation within the z-x-plane.

3.3 Transformations in 3-D

Note: Still right-handed coordinate system, just rotated

Rotation around an arbitrary axis

Every rotation around an arbitrary axis can be composed of rotations around the coordinate axes (see Euler).



$$b_x = \sin\varphi \;\; \cos\theta$$
$$b_y = \sin\varphi \;\; \sin\theta$$
$$b_z = \cos\varphi$$

We now develop such a rotation $R_G(\alpha)$ that rotates around a point P using an arbitrary axis of rotation G at an angle α.

First, we consider a special case where the axis of rotation intersects with the origin and is defined by the vector $b = (b_x,b_y,b_z)$ with $||b||=1$, i.e. $G:\lambda \cdot b$.

3.3 Transformations in 3-D

Rotation around an arbitrary axis (continued)

We now need to find the coordinates of a point P after rotating around the axis G at an angle α.

Idea:

We transform the point P in such a way that the rotational axis is identical to the z-axis. Then, we apply a rotation around the z-axis at angle φ using the rotation matrix $R_z(\varphi)$. After that, we reverse the temporary rotation which rotated G onto the z-axis.

Particularly, we follow the following five steps:

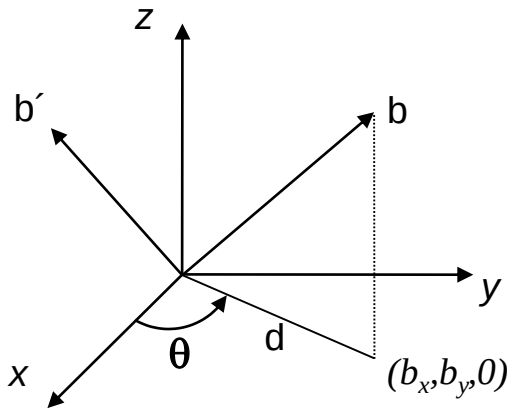
3.3 Transformations in 3-D

Rotation around an arbitrary axis (continued)

Step 1:

We first rotate in such a way that the vector b is located within the z-x-plane (b'). This transforms P onto $P'=R_z(-\theta)P$:

$$R_z(-\theta)=\begin{pmatrix}\cos\theta & \sin\theta & 0 & 0\\ -\sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}=\frac{1}{d}\begin{pmatrix}b_x & b_y & 0 & 0\\ -b_y & b_x & 0 & 0\\ 0 & 0 & d & 0\\ 0 & 0 & 0 & d\end{pmatrix}$$



$$d^2=b_x^2+b_y^2$$

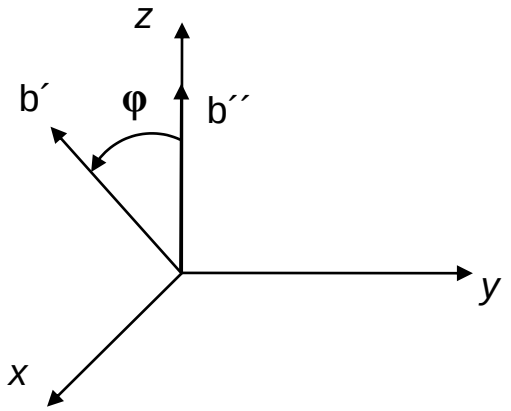
3.3 Transformations in 3-D

Rotation around an arbitrary axis (continued)

Step 2:

We now rotate in such a way that the vector b' is identical with the z-axis (b''). Then we get $P'' = R_y(-\varphi)P'$.

$$R_y(-\varphi) = \begin{pmatrix} \cos\varphi & 0 & -\sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} b_z & 0 & -d & 0 \\ 0 & 1 & 0 & 0 \\ d & 0 & b_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



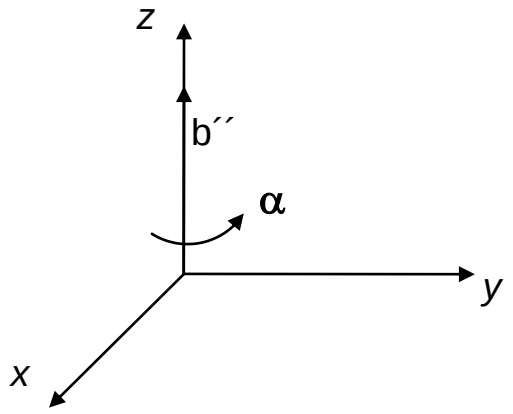
3.3 Transformations in 3-D

Rotation around an arbitrary axis (continued)

Step 3:

Now we can rotate around the z-axis at angle α. This then transforms the point P'' onto P''' with $P''' = R_z(\alpha)P''$.

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



3.3 Transformations in 3-D

Rotation around an arbitrary axis (continued)

Steps 4 and 5:

Finally, we apply the inverse rotation of steps 1 and 2 in reverse order to reverse this temporary rotation. This then maps the point P''' onto the desired point Q by applying $Q = R_z(\theta)\, R_y(\varphi) P'''$.

$$R_y(\varphi) = \begin{pmatrix} b_z & 0 & d & 0 \\ 0 & 1 & 0 & 0 \\ -d & 0 & b_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\theta) = \frac{1}{d} \begin{pmatrix} b_x & -b_y & 0 & 0 \\ b_y & b_x & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 0 & d \end{pmatrix}$$

3.3 Transformations in 3-D

Rotation around an arbitrary axis (continued)

Result:

The entire transformation can then be realized by concatenating all these transformations:

$$M_b(\alpha) = R_z(\theta) \cdot R_y(\varphi) \cdot R_z(\alpha) \cdot R_y(-\varphi) \cdot R_z(-\theta)$$

General case:

If the rotational axis is an arbitrary straight-line (i.e. does not intersect with the origin)

$$G: a+\lambda \cdot b \quad (\lambda \in \mathbb{R}, ||b||=1, a = (a_x, a_y, a_z))$$

then we need to perform a translation first and reverse it again after applying all the above rotations:

$$M_G(\alpha) = T(a_x,a_y,a_z) \cdot R_z(\theta) \cdot R_y(\varphi) \cdot R_z(\alpha) \cdot R_y(-\varphi) \cdot R_z(-\theta) \cdot T(-a_x,-a_y,-a_z)$$

3.4 Projections

A projection generally is a map from an n-dimensional space onto a space with dimension less than n.

Since a display (CRT, LCD) is often a two-dimensional output device, three-dimensional objects need to be rendered using a two-dimensional view. To achieve this, a point in 3-D space is mapped along a projection ray (projector) onto a pre-defined projection plane.

The projection ray is defined by the center of projection and the point itself. The intersection between the projection ray and the projection plane determines the projected point.

3.4 Projections

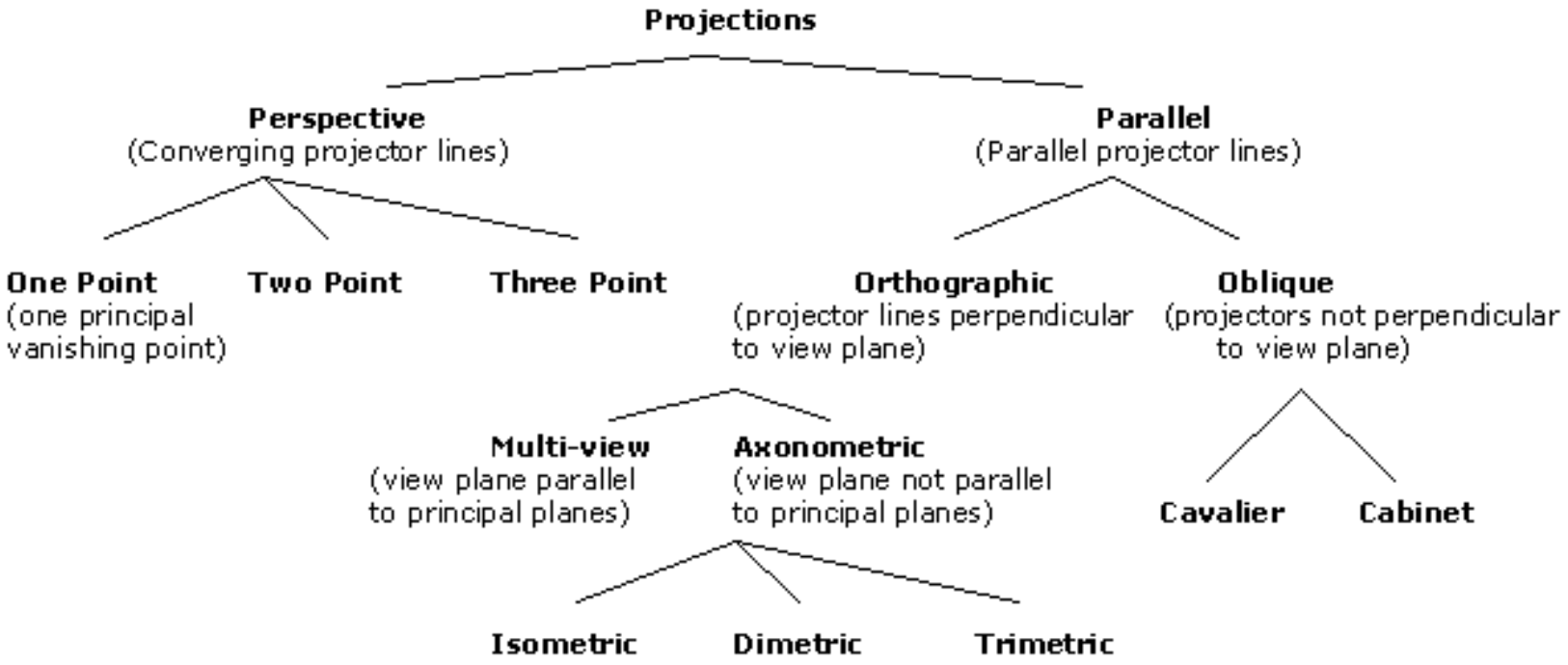
Examples for geometric, planar projections are perspective and parallel projection.

For parallel projections, the center of projection is located at a point that is infinitely far away. When using projective geometry, the parallel projection is therefore just a special case of the perspective projection.

The following image shows a classification of the most common projections:

3.4 Projections

Most common projections:

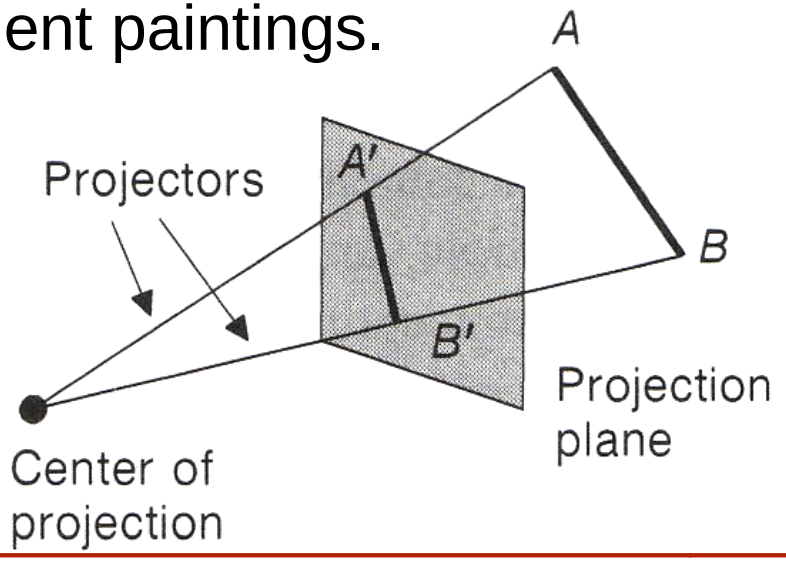


3.4 Projections

Perspective projection

For the perspective projection, all projector rays intersect with the center of projection. We can think of the center of projection as the location of the eye.

This projection creates an optical impression of depth and can already be found in ancient paintings.



3.4 Projections

Perspective projection (continued)

Properties:

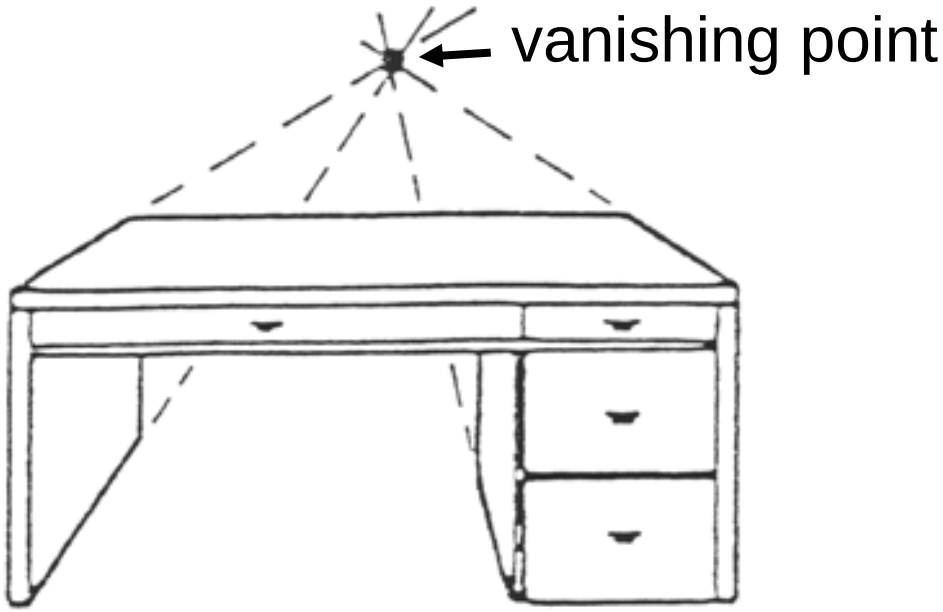
Every pair of straight-lines, which are not parallel to the projection plane, intersect at a point, the so called vanishing point. There may be infinitely many vanishing points, one for every set of lines that are parallel. Often times, one vanishing point is used for lines parallel to a single coordinate axis, e.g. all lines parallel to the x-axis intersect at the same vanishing point.

Perspective projection are classified according to the number of coordinate axes that intersect the projection plane. This results in one point, two point, and three point perspectives.

3.4 Projections

Perspective projection (continued)

Example:

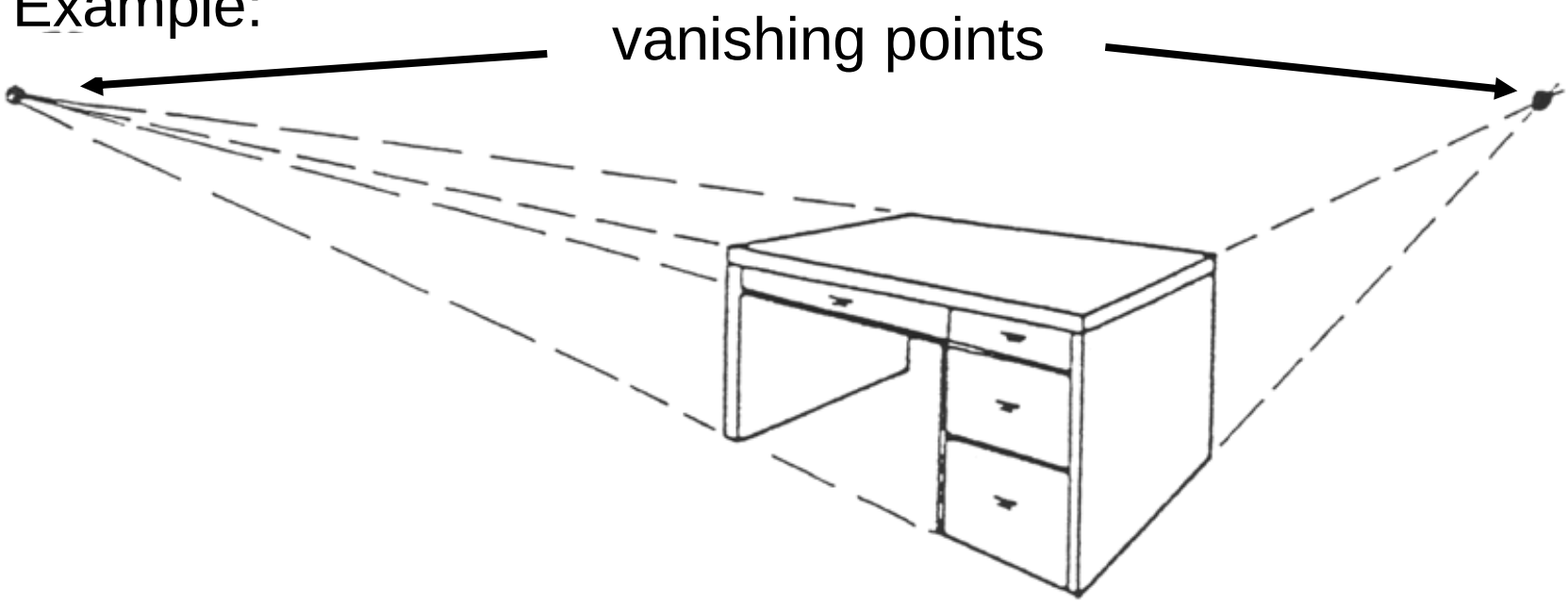


one point perspective

3.4 Projections

Perspective projection (continued)

Example:



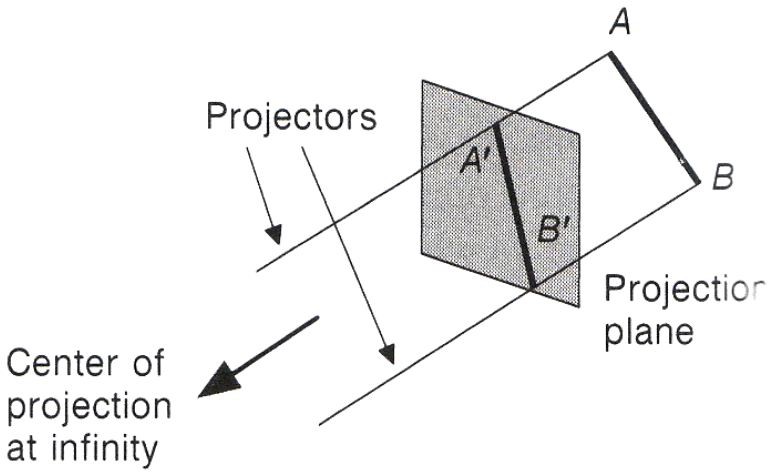
two point perspective

3.4 Projections

Parallel projection

For a parallel projection, the center of projection is infinitely far away. Thus, all projection rays are parallel to each other.

The parallel projection is less realistic. It is, however, easier to estimate measurements from the projected image.



3.4 Projections

Parallel projection (continued)

The projection rays may intersect the projection plane orthogonally (orthographic projection) or at any other angle (oblique projection).

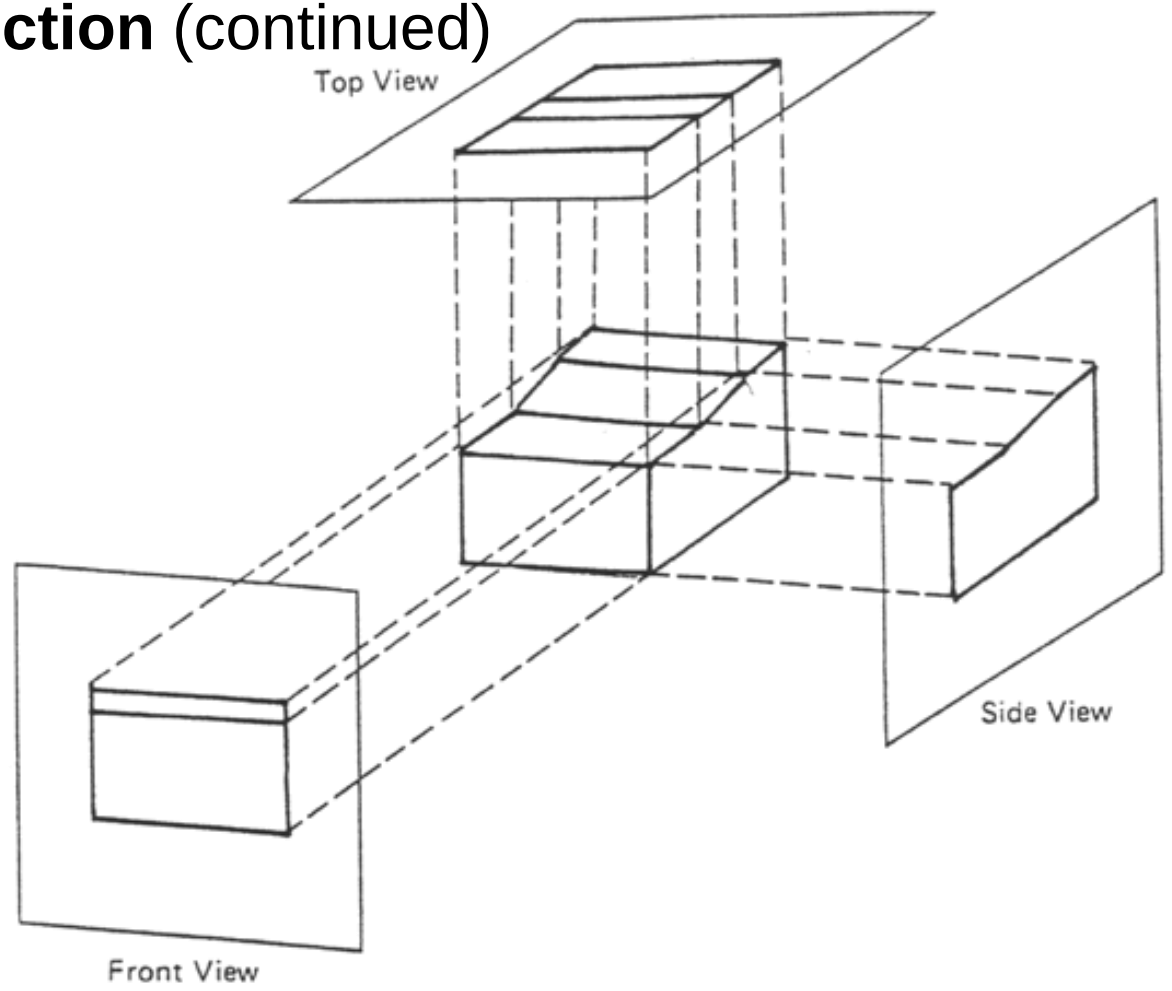
Orthographic Projection

For a orthographic projection, the direction of projection is identical to the normal of the projection plane. We distinguish between multi-view and axonometry.

The multi-view projection uses projection planes that are parallel to the coordinate axes resulting in three different views: top view, front view, and side view.

3.4 Projections

Parallel projection (continued)



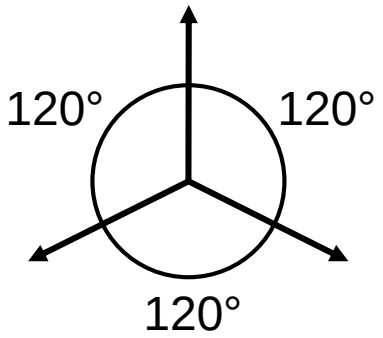
3.4 Projections

Parallel projection (continued)

When using axonometry, the direction of projection is not necessarily parallel to one of the coordinate axes.

Parallel lines are mapped onto parallel lines. Angles, however, are not preserved. Distances can be measured along the coordinate axes, the scale may be different for each coordinate axis, though.

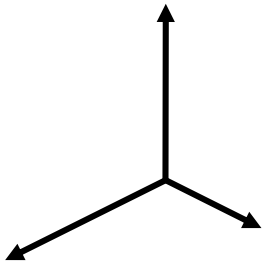
The most common case is the **isometric** axonometry. This projection maps the coordinate axes in such a way that each pair of axes forms the same angle.



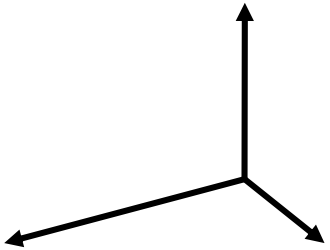
3.4 Projections

Parallel projection (continued)

The **dimetric** projection maps the coordinate axes in such a way that two of them form the same angle; the scale for two of the three coordinate axes is identical:



The **trimetric** projection maps the coordinate axes in such a way that they all form different angles, the scale is different for each coordinate axis:



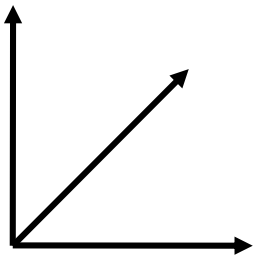
3.4 Projections

Parallel projection (continued)

Oblique projections occur if the direction of projection is different from the normal of the projection plane. The most common examples of oblique projections are the **cavalier** and **cabinet projection**.

Cavalier projection

The angle between the direction of projection and the projection plane is 45°. The length of a line that is orthogonal to the projection plane remains the same.



3.4 Projections

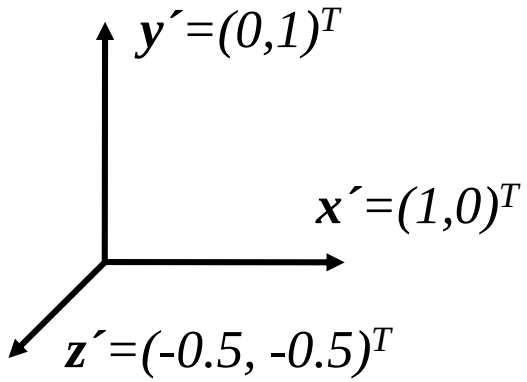
Parallel projection (continued)

Cabinet projection

In this case, the length of the projection of a line that is orthogonal to the projection plane is supposed to be cut in half after projecting.

The angle between the direction of projection and the projection plane therefore is $arctan\ 2 = 63.4°$.

Example:

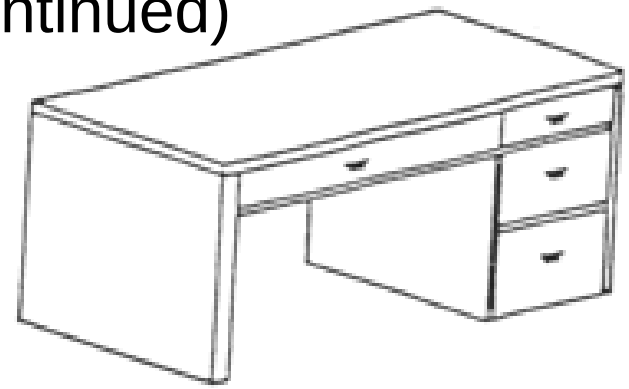


projected unit vectors

3.4 Projections

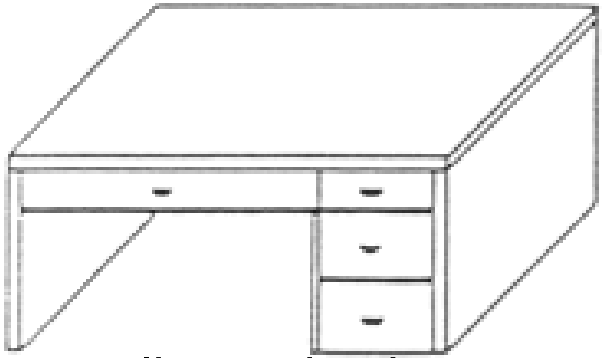
Parallel projection (continued)

Examples:



isometric: 1:1:1

cabinet projection



cavalier projection

3.4 Projections

Perspective projection – implementation

The perspective projection depends on the application and can be realized using different configurations using appropriate transformations of the coordinate system.

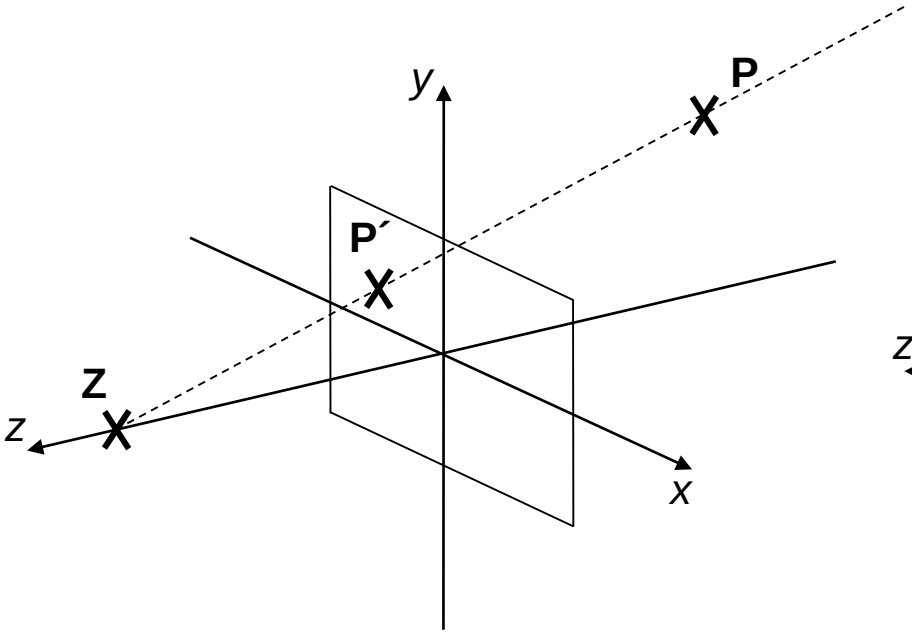
In this example, we choose the following setup:

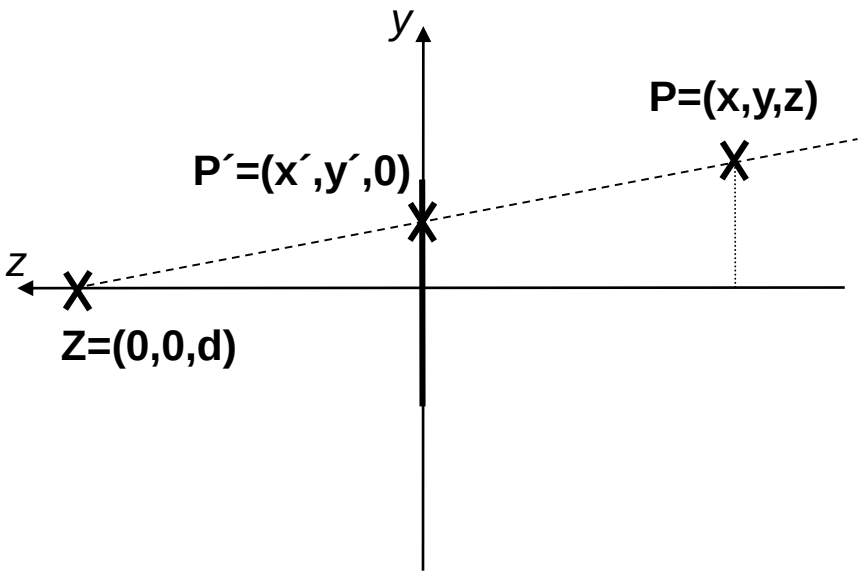
- The center of projection Z and the location of the eye are identical and are positioned on the positive z-axis at distance $d>0$ from the origin, i.e. $Z = (0, 0, d)$.
- The view direction points towards the negative z-axis.
- The projection plane is identical to the x-y-plane.

3.4 Projections

Perspective projection – implementation (continued)

Setup:





3.4 Projections

Perspective projection – implementation (continued)

According to the theorem on intersecting lines:

$$\frac{y'}{d} = \frac{y}{d-z} \quad \text{and} \quad \frac{x'}{d} = \frac{x}{d-z}$$

$$\text{Hence :} \quad y' = y \cdot \frac{d}{d-z} = y \cdot \left(1 - \frac{z}{d}\right)^{-1}$$

$$x' = x \cdot \frac{d}{d-z} = x \cdot \left(1 - \frac{z}{d}\right)^{-1}$$

Thus, the perspective projection can be described by the following matrix :

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -d^{-1} & 1 \end{pmatrix}$$

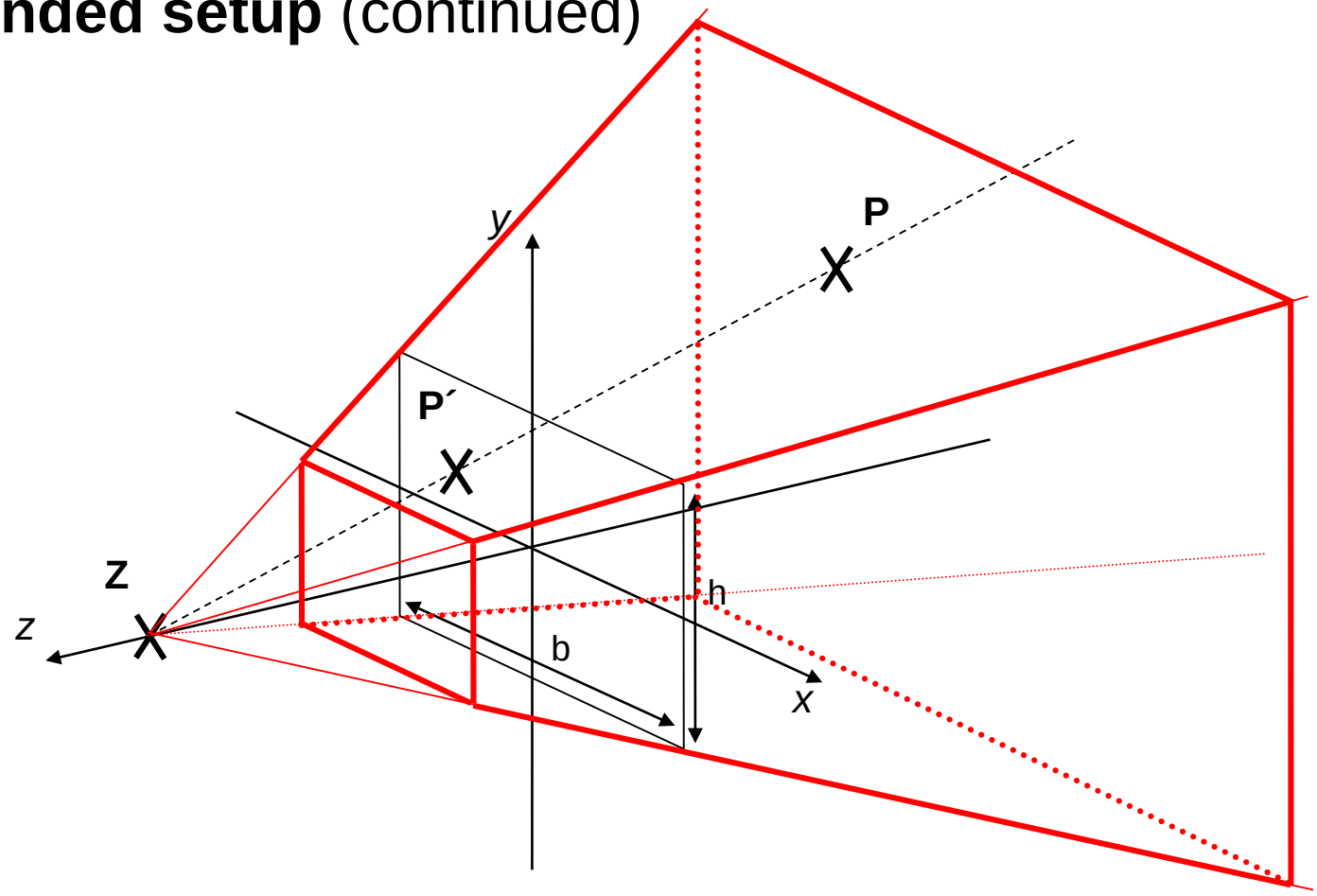
3.4 Projections

Extended setup

- Within the projection plane, a **view window** is specified (width, height, ratio); the view window is symmetrically placed around the origin.
- The projections intersecting the vertices of the view window define the **view frustum** (view volume).
- Two additional planes, which are parallel to the projection plane, (**front** and **back clipping plane**) limit the view frustum in z-direction.
- The view frustum limits the space that is displayed (→clipping).

3.4 Projections

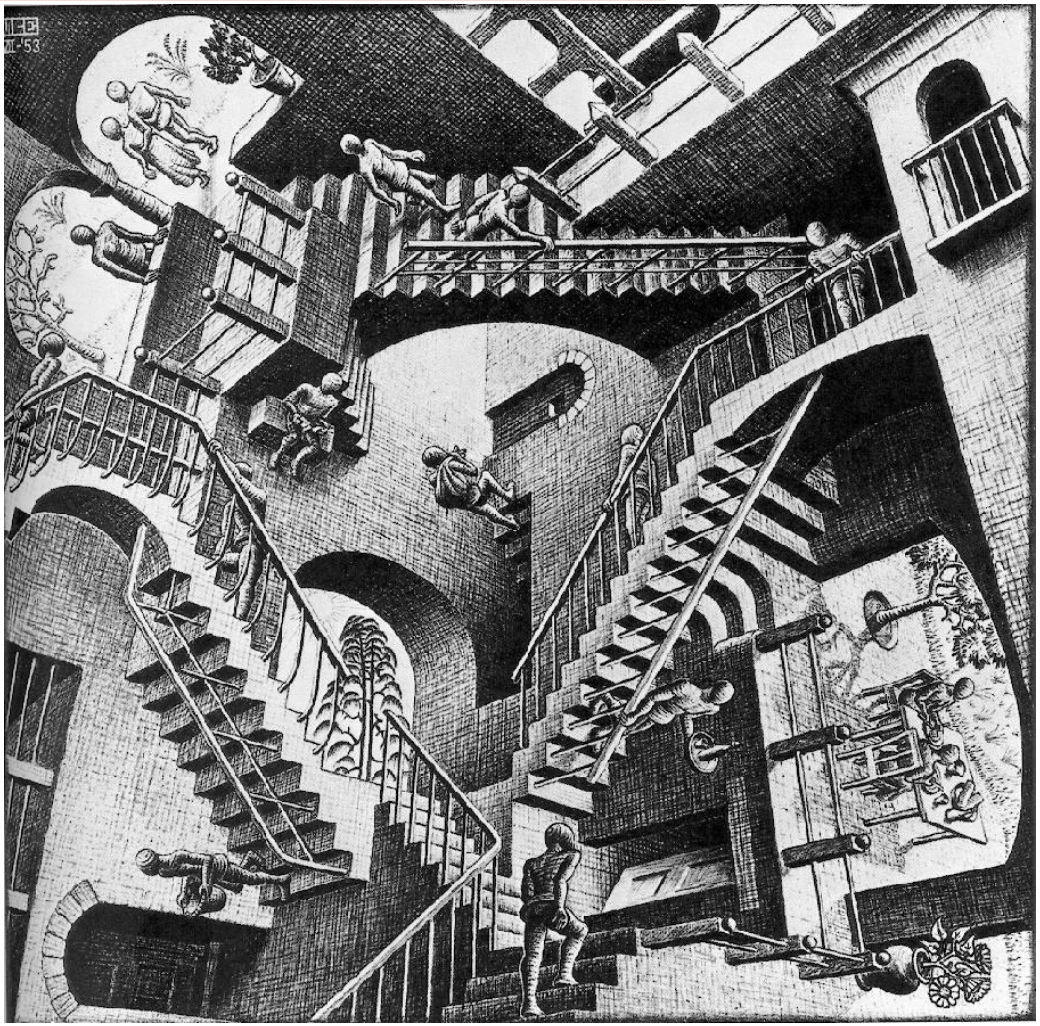
Extended setup (continued)



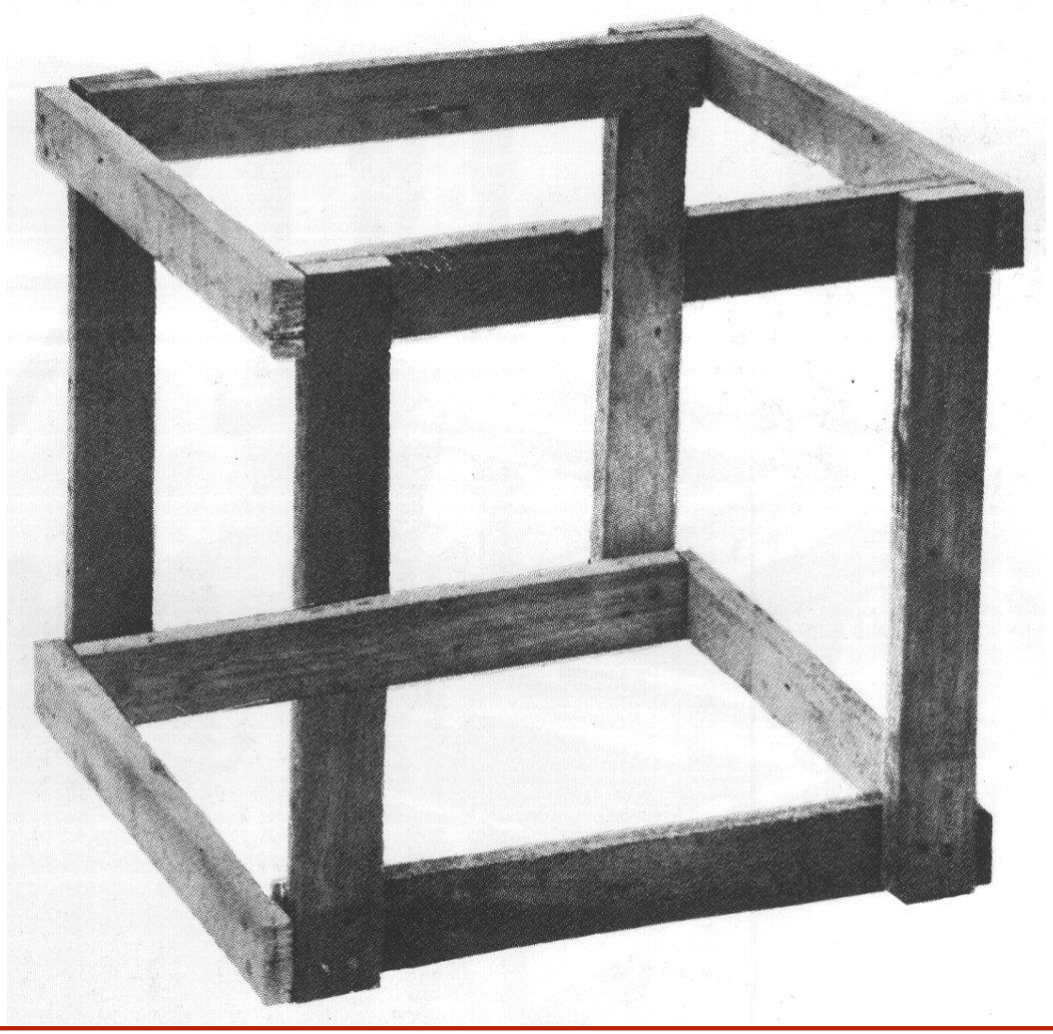
3.4 Projections

3.4 Projections

3.4 Projections



3.4 Projections



3.4 Projections

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).

3.4 Projections

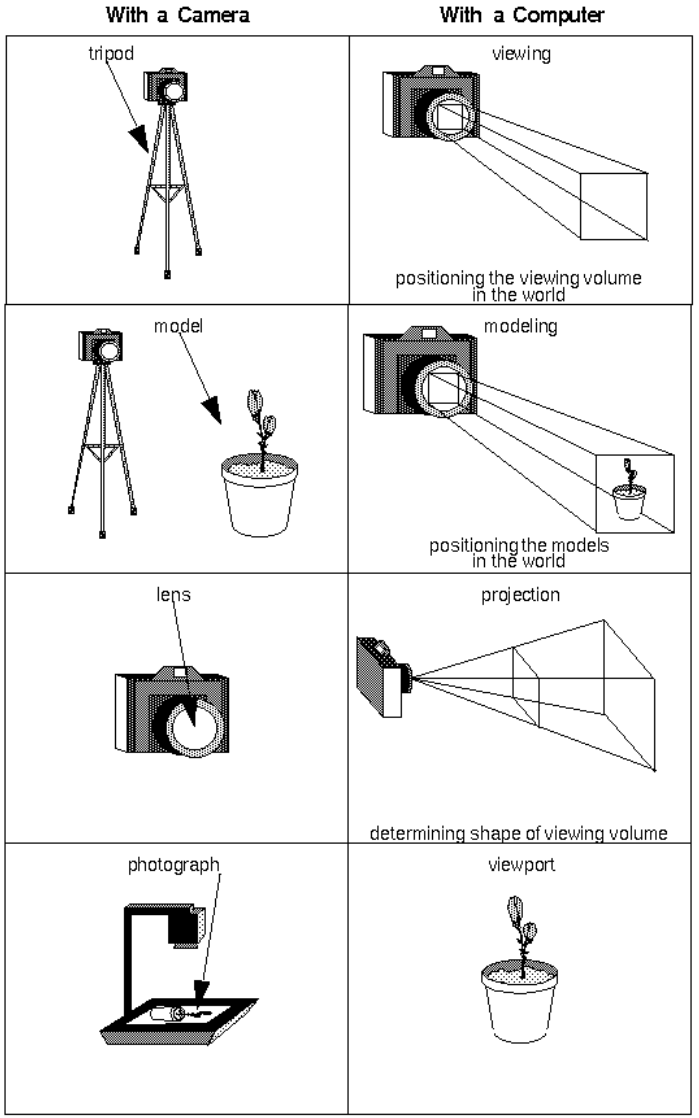
Viewing in OpenGL (continued)

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.

3.4 Projections

Viewing in OpenGL (continued)

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



3.4 Projections

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 homogeneous coordinates to represent matrices, i.e. all matrices are 4x4 matrices.

3.4 Projections

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.

3.4 Projections

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:

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

3.4 Projections

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 $C \cdot M$.

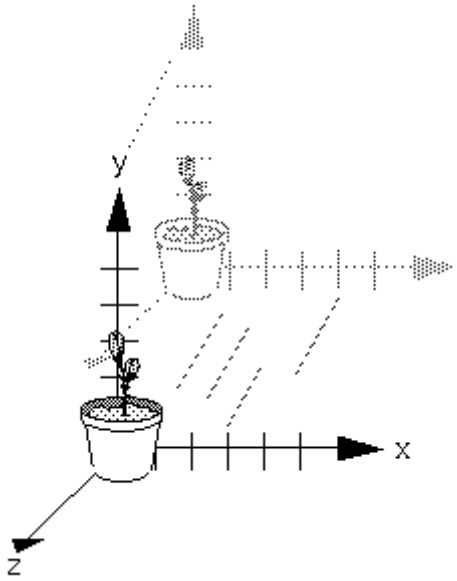
3.4 Projections

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)*.

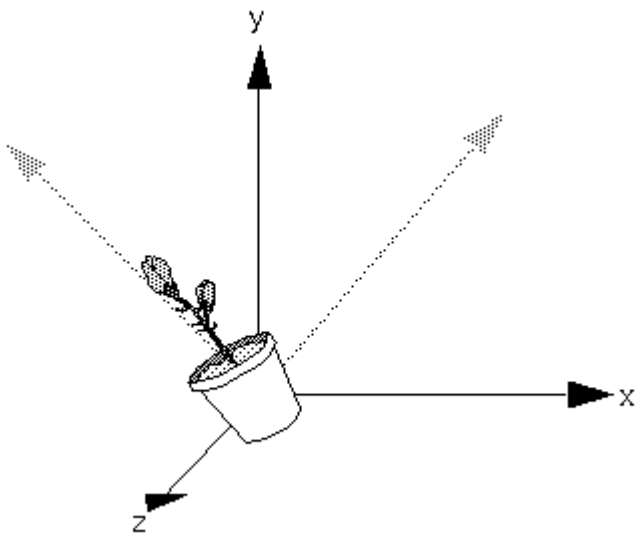


3.4 Projections

Viewing in OpenGL (continued)

The function

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



multiplies a rotational matrix onto the current matrix stack,

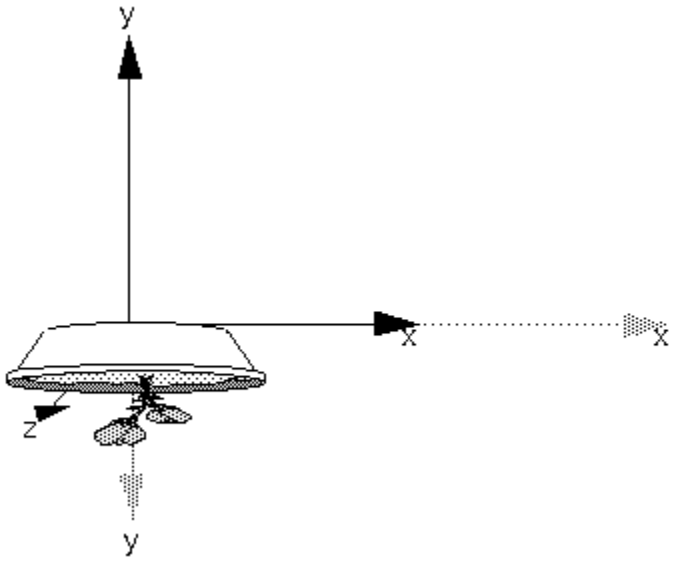
3.4 Projections

Viewing in OpenGL (continued)

while the function

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

appends a scaling matrix to the matrix stack.



3.4 Projections

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.

3.4 Projections

Viewing in OpenGL (continued)

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

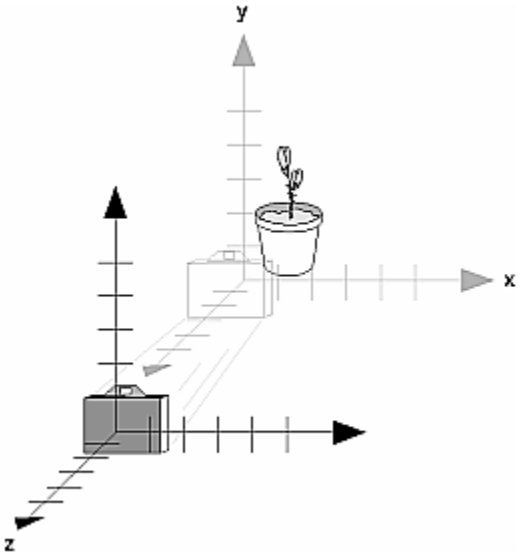
```
void gluLookAt (GLdouble eyex,
                GLdouble eyey,
                GLdouble eyez,
                GLdouble centerx,
                GLdouble centery,
                GLdouble centerz,
                GLdouble upx,
                GLdouble upy,
                GLdouble upz);
```

3.4 Projections

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);`

3.4 Projections

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:

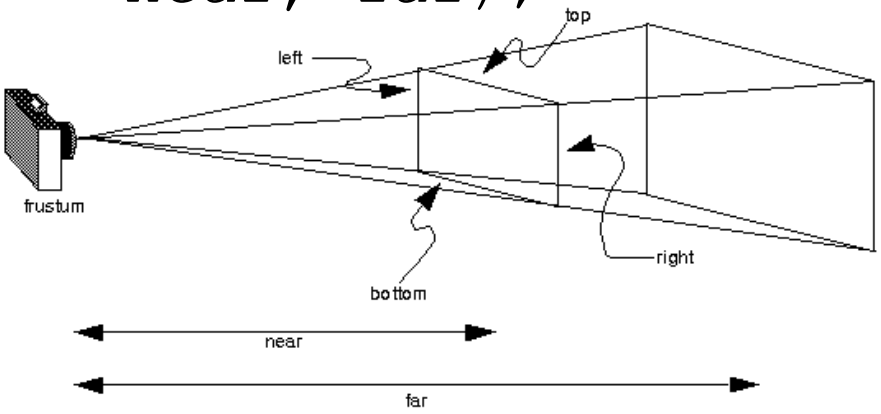
```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
```

3.4 Projections

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);
```

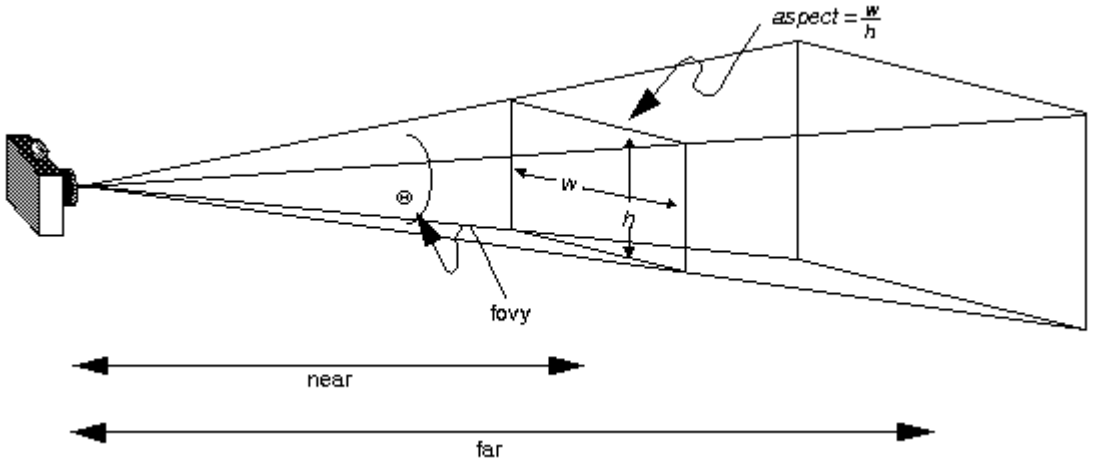


3.4 Projections

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);
```

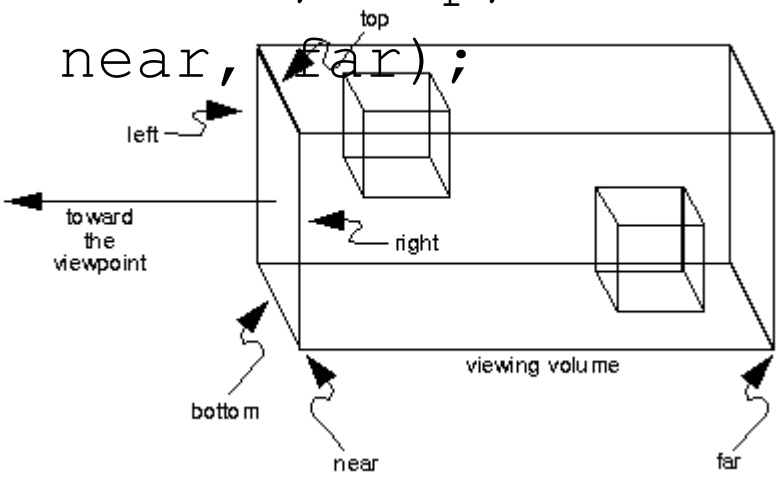


3.4 Projections

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);
```



3.4 Projections

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);
```

3.4 Projections

Viewing in OpenGL

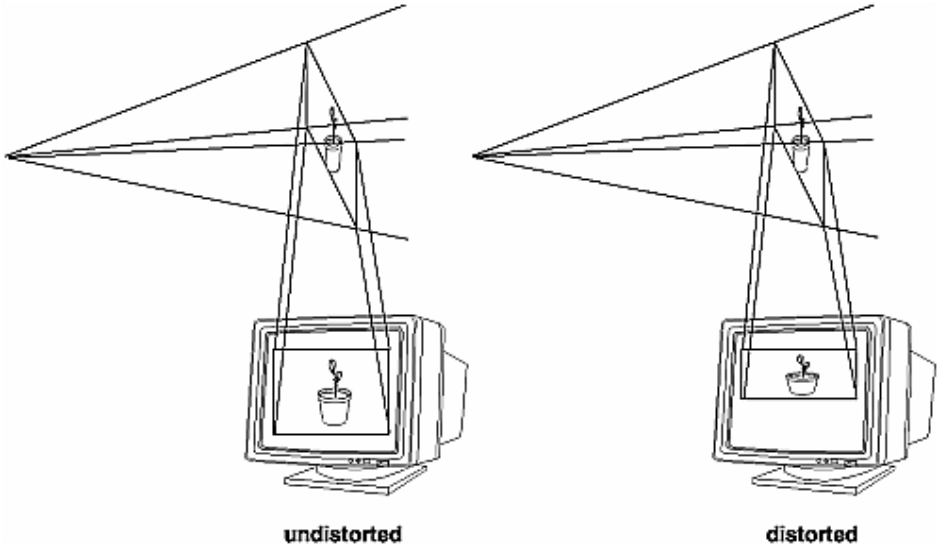
The last step of the process of creating an image on a computer display is the viewport transformation. Recalling the camera analogy, the viewport transformation corresponds to the stage where the size of the developed photography is chosen. The viewport is measured in window coordinates. By default, 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);
```

3.4 Projections

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.



3.4 Projections

As of OpenGL 3.0 many of these functions were declared deprecated including glLoadMatrix, glRotate, etc. While these are still supported (and probably will be for quite a while), the OpenGL 3.0 specifications (and later versions) you wants you to handle matrices and view settings manually yourself.

While using glRotate etc. is easier, especially for the beginner, there are ways to be conform with the latest OpenGL specifications without having to do everything manually: use the **glm library (OpenGL Mathematics)**.

3.4 Projections

OpenGL Mathematics

The glm library supports most vector and matrix algebra to specify any type of transformation:

```
#include <glm/glm.hpp>
#include <glm/transform.hpp>

glm::mat4 myMatrix = glm::translate(10.0f, 0.0f, 0.0f);
glm::vec4 myVector (10.0f, 10.0f, 10.0f, 0.0f);
glm::vec4 transformedVector = myMatrix * myVector;
```

3.4 Projections

OpenGL Mathematics

Specifying a scaling matrix:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform.hpp>
glm::mat4 myScalingMatrix =
   glm::scale(2.0f, 2.0f ,2.0f);
```

3.4 Projections

OpenGL Mathematics

Specifying a rotational matrix:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform.hpp>
glm::vec3 myRotationAxis( 1.0, 0.0, 0.0);
glm::rotate( angle_in_degrees,
   myRotationAxis );
```

3.4 Projections

OpenGL Mathematics

Convenience functions for view settings are also available:

```
glm::mat4 CameraMatrix =
 glm::LookAt(
     cameraPosition,
     cameraTarget,
     upVector);
```

3.4 Projections

OpenGL Mathematics

Convenience functions for projection:

```
glm::mat4 projectionMatrix =
   glm::perspective(
      FoV,
      4.0f / 3.0f, // Aspect Ratio
      0.1f,        // Near clipping plane.
      100.0f       // Far clipping plane.
      );
```

3.4 Projections

OpenGL Mathematics

You could then feed the results into the different matrix stacks. However, that is deprecated as well (even though it is very useful).

Instead, you are according to the latest OpenGL specs supposed to feed those matrices directly into your own shader programs and multiply they vertices within the vertex shader yourself (manually as the built-in variables for these matrices are deprecated as well).

We will look into shader program during the last chapter of this class.