Lecture 3:
Viewing

Topics:

1. Classical viewing
2. Positioning the camera
3. Perspective and orthogonal projections
4. Perspective and orthogonal projections in OpenGL
5. Perspective and orthogonal projection matrices

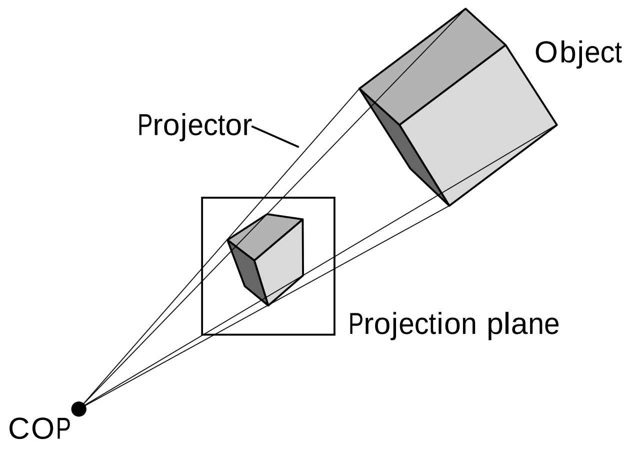
Chapter 5, Sections 5.1, 5.2, 5.3, 5.4, 5.7, 5.8.

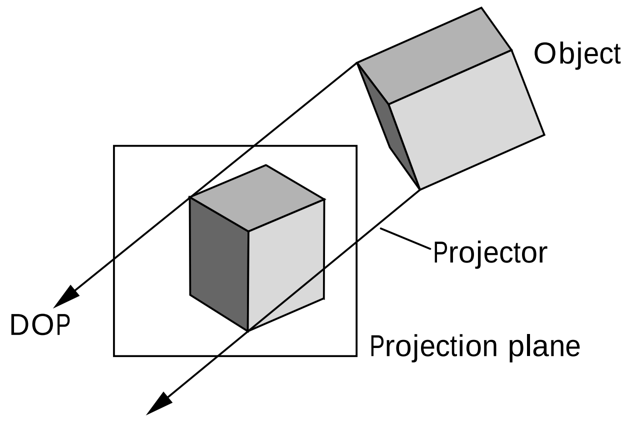
Classical and computer viewing

Both kinds of viewing involve: objects, a viewer, projectors, and a projection plane. The projectors meet at the **centre of projection** (**COP**). The COP is the centre of the lens or eye, or in computer graphics the origin of the **camera frame**.

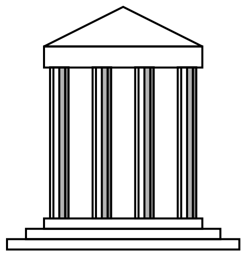
In both classical and computer viewing the viewer can be an infinite distance from the objects. As we move the COP to infinity the projectors become parallel and in the limit, the COP can be replaced by a **direction of projection** (**DOP**).

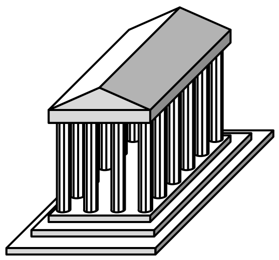
Views with a finite COP are called **perspective views** while those with a COP at infinity are called **parallel views**.

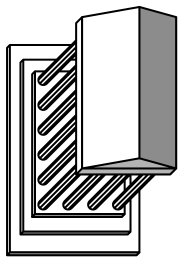


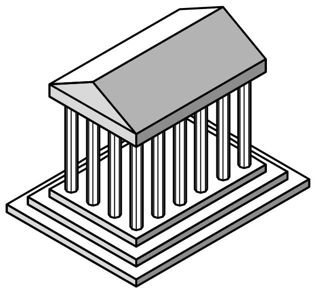


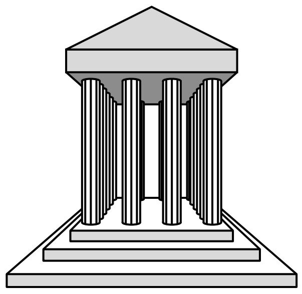
Classical viewing

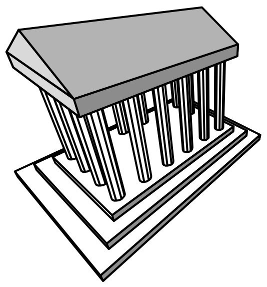

Front elevation


Elevation oblique


Plan oblique


Isometric


One-point perspective

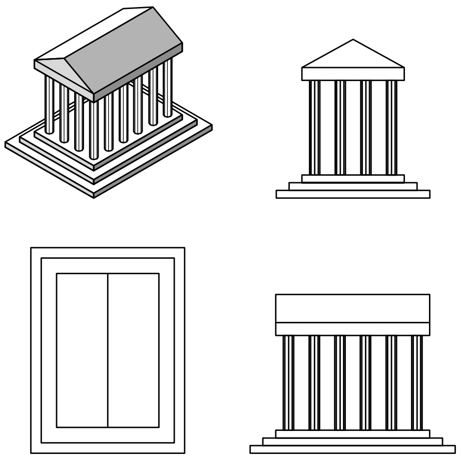

Three-point perspective

In classical viewing there is an underlying notion of a **principal face**. This is typical in architecture where objects are often buildings having planar faces: front, back, top, bottom, right and left faces. Such objects have three orthogonal directions associated with them.

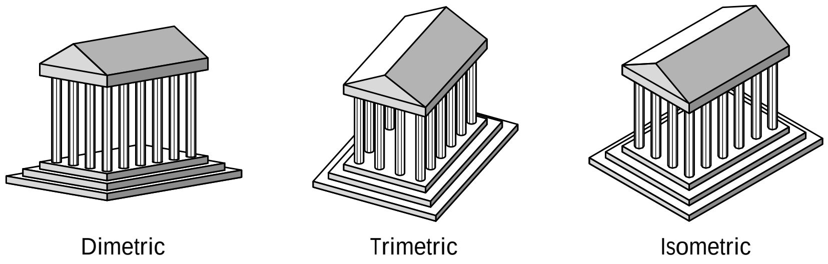
Parallel viewing

An **orthogonal projection** is one in which the projectors are not only parallel but also perpendicular to the projection plane.

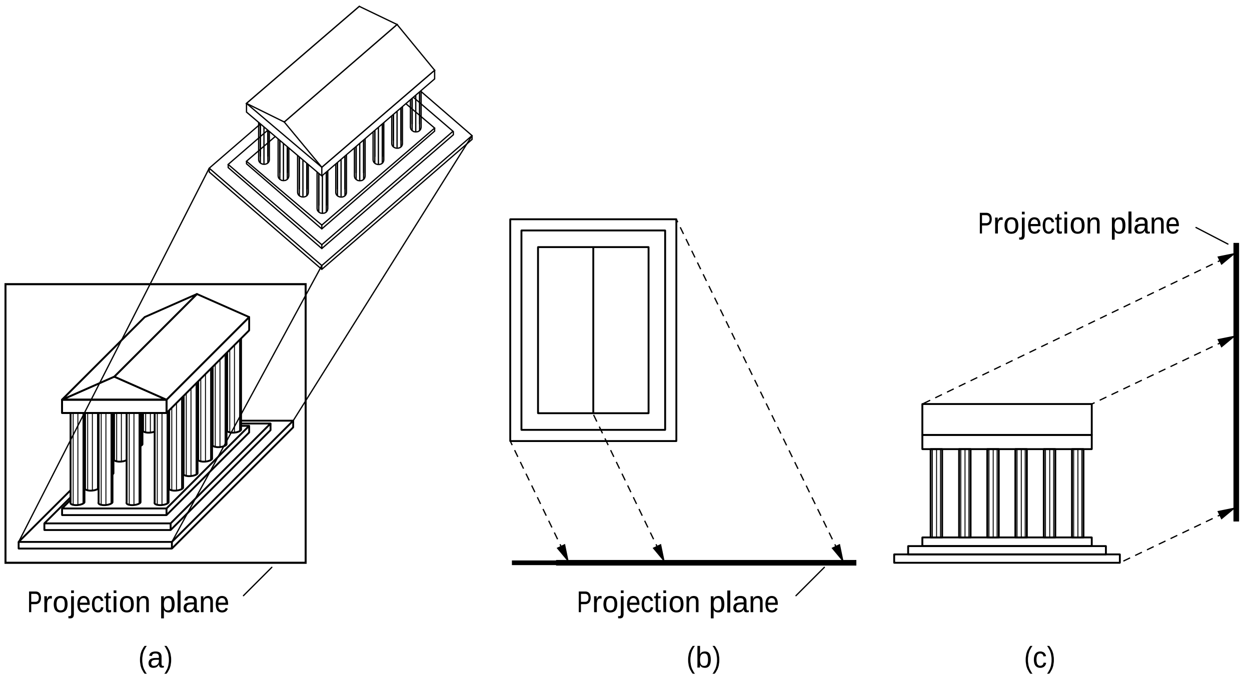
The classical **orthographic projection** is an orthogonal projection in which the projection plane is placed parallel to one of the principal faces of the object.



A more general orthogonal projection is the **axonometric projection**, in which the projection plane can have an arbitrary orientation.



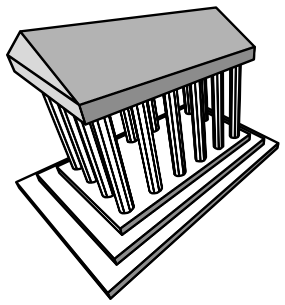
Oblique projections are parallel views in which the projection plane is not orthogonal to the projectors. The projection plane is "skew".

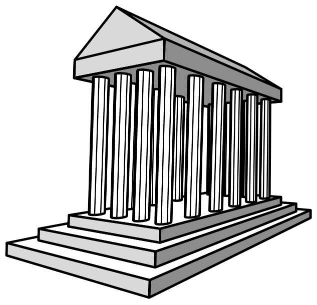


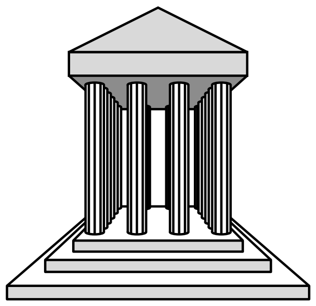
(a) construction
(b) top view
(c) side view

Perspective viewing

In perspective viewing, the image of an object gets smaller when it is moved away from the viewer. Therefore perspective views look more natural. However they cannot be used to make measurements, as with parallel views.


(a)


(b)

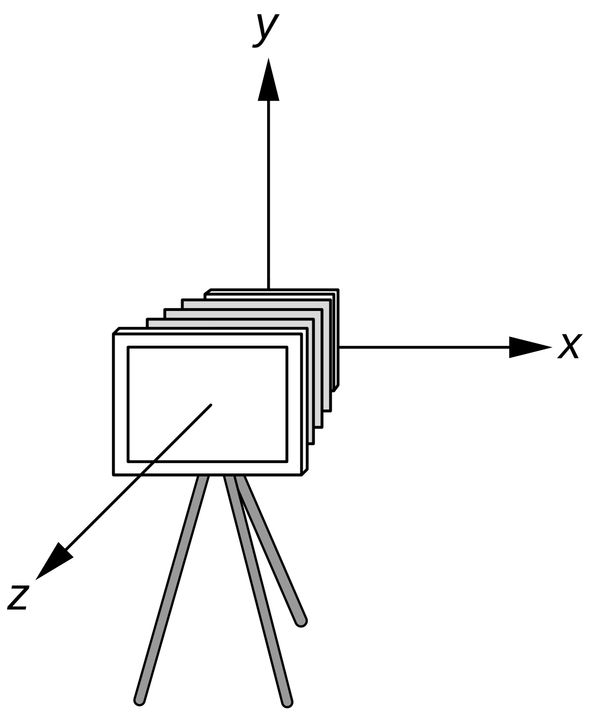

(c)

(a) **Three-point perspective**: three vanishing points
(b) **Two-point perspective**: two vanishing points
(c) **One-point perspective**: one vanishing point

Positioning the camera

How do we position the camera with respect to the objects? There are two points of view: (1) we move the camera with respect to the objects or (2) we move objects relative to a fixed camera. The two reciprocal movements will result in the same image.

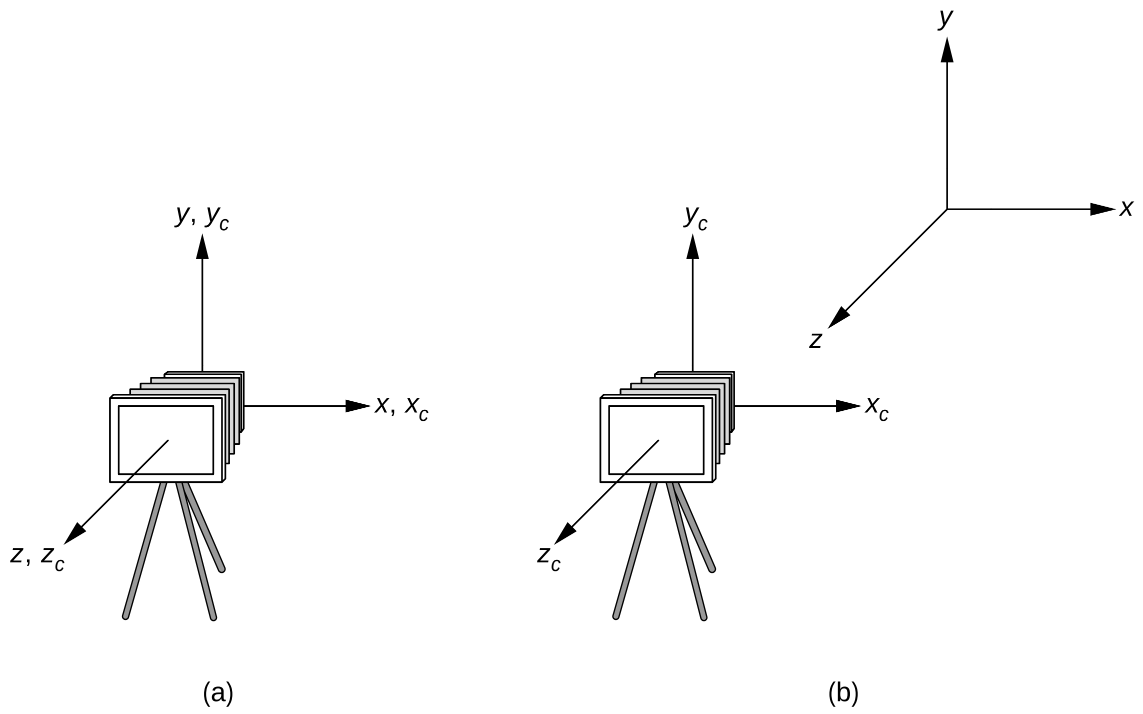
These movements are controlled by the **model-view matrix** which represents a transformation between the world frame and the camera frame. Initially (by default) the model-view matrix is set to the **identity**, in which case both frames are the same. The camera is at the origin pointing down the negative z axis.



Example 1. Objects are typically placed near the origin. So to view them, we need to move the camera back along the z axis. The correct transformation is therefore

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0,0, 0.0, -d);
```

for some positive number d.

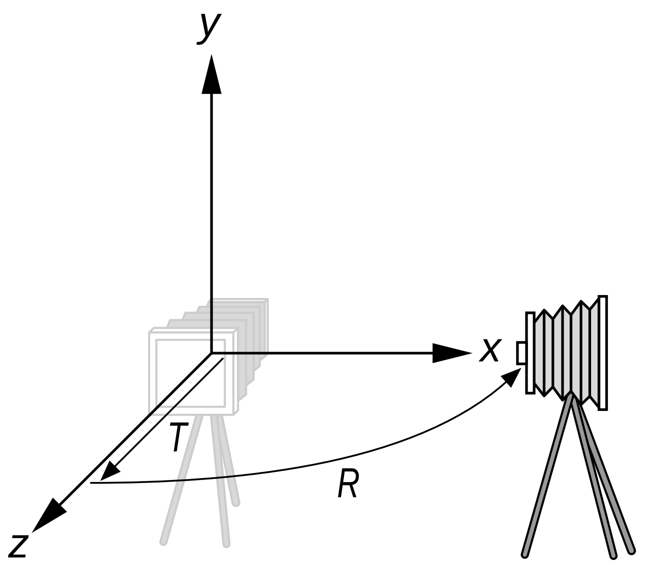


We can think of this transformation either as a translation of the objects through the vector $(0, 0, -d)$ or as a translation of the camera through the vector $(0, 0, d)$.

Example 2. We now wish to view the objects (which are again centred around the origin) from the positive x axis. To achieve this we need first to move the camera backwards along the positive z axis, a distance d, and then rotate the camera around the y axis by 90 degrees:

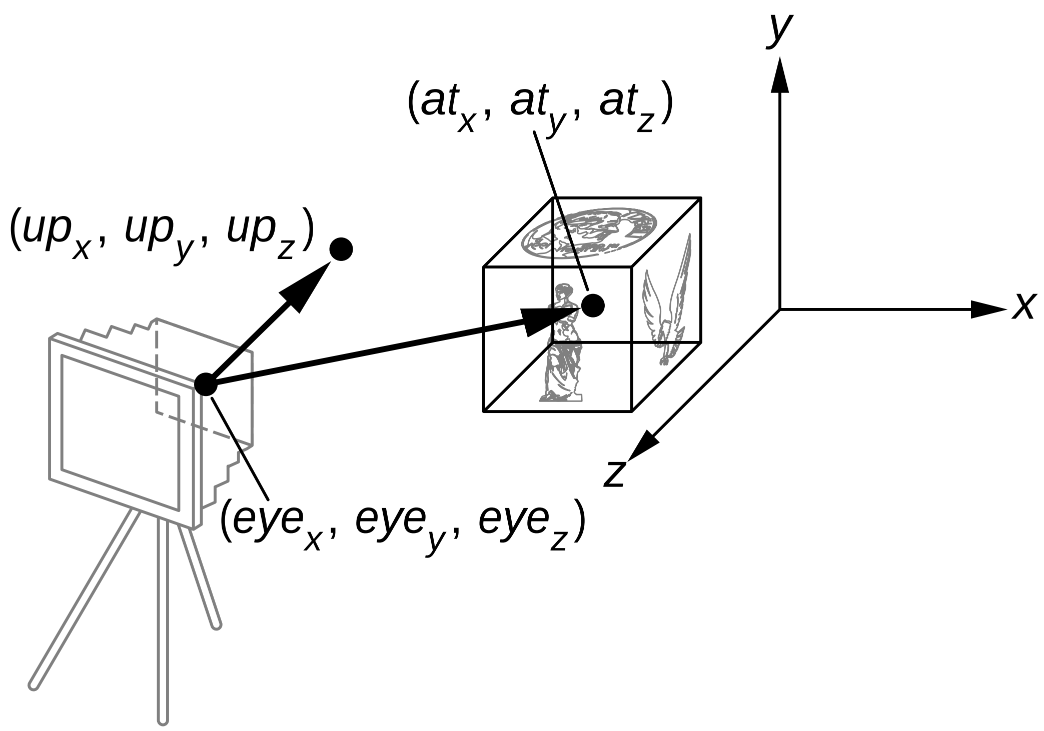
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0,0, 0.0, -d);
glRotate(-90.0, 0.0, 1.0, 0.0);
```

The equivalent viewpoint is to think of the model-view matrix being applied to the objects: they are first rotated through -90 degrees around the y axis, and then translated through $(0, 0, -d)$. The model-view matrix has the form $M = TR$, and M is applied to each object vertex p, moving it to Mp.



Example 3. To achieve more complicated orientations of the camera we can concatenate rotation matrices. However, it is easier to use a function provided in GLU:

```
    gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz);
```



This function sets the origin of the camera to be the **eyepoint e** and sets the camera frame axes **u,v,n** (playing the role of x, y, z) according to the point **at** and the vector **up**. It first sets $\mathbf{n} = \mathbf{e} - \mathbf{at}$, the normal to the projection plane. The vector **v** is then computed as the projection of **up** onto this plane. Finally **u** is taken to be the cross product of **v** and **n**.

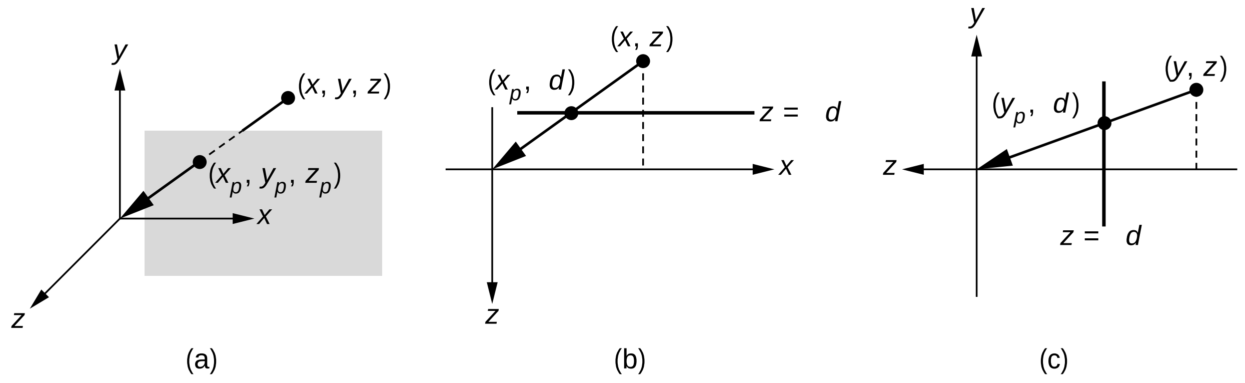
Perspective projections

Suppose the camera is at its default position at the origin, pointing along the negative z axis. How do we compute a perspective projection?
For simplicity, suppose the projection plane is parallel to the x-y plane, i.e.,

$$z = d,$$

for some negative number d. Each point (x, y, z) in space is projected towards the origin. Therefore the point where the projector hits the projection plane is (x_p, y_p, z_p), where

$$x_p = dx/z, \qquad y_p = dy/z, \qquad z_p = d.$$



The perspective transformation $(x, y, z) \to (x_p, y_p, z_p)$ is clearly non-affine. So how can we represent it using a 4×4 matrix?

The answer is to extend the notion of homogeneous coordinates. Recall that the point (x, y, z) has the homogeneous representation $(x, y, z, 1)$. We can extend this representation by agreeing that the same point can be represented by the vector (wx, wy, wz, w), for any $w \neq 0$. To recover the representation $(x, y, z, 1)$, we have only to divide the coordinates by w.

With this extended homogeneous representation of points, we can represent our perspective transformation by the matrix

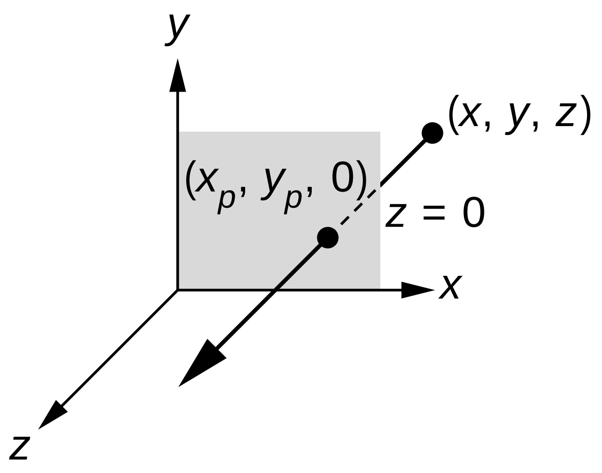
$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix}.$$

The matrix M transforms the point $p = (x, y, z, 1)^T$ into the point $q = (x, y, z, z/d)^T$, and after division by the last coordinate, z/d, we arrive at $q' = (x_p, y_p, z_p, 1)$. This latter division is called **perspective division**, and can be made part of the pipeline:

$$\textbf{model-view} \rightarrow \textbf{projection} \rightarrow \textbf{perspective division}$$

Orthogonal projections

Orthogonal projections can be viewed as the limiting case of perspective projections as the COP goes to infinity. However, in practice it is much simpler to derive the projection equation directly. We can assume that the projection plane is $z = 0$, in which case the point (x, y, z) is mapped to (x_p, y_p, z_p) where $x_p = x$, $y_p = y$, $z_p = 0$.



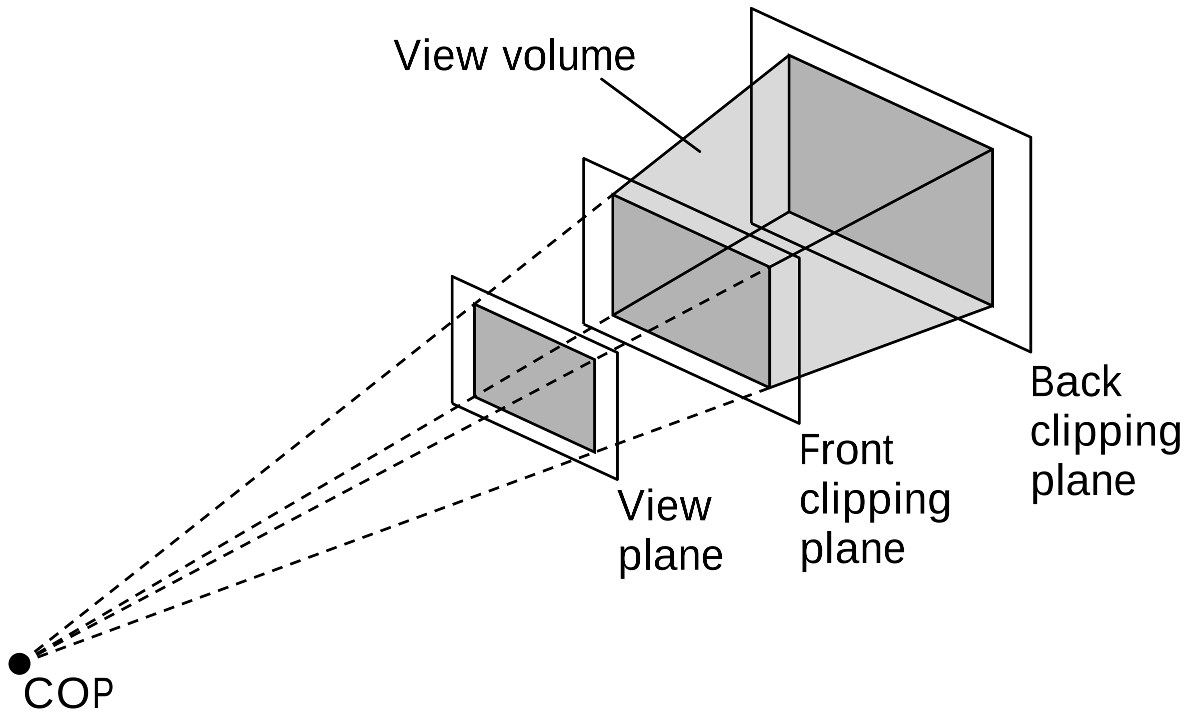
In homogeneous coordinates, the transformation is

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

Here no perspective division is necessary, but in hardware implementations we can use the same pipeline for both perspective and orthogonal projections.

Perspective projections in OpenGL

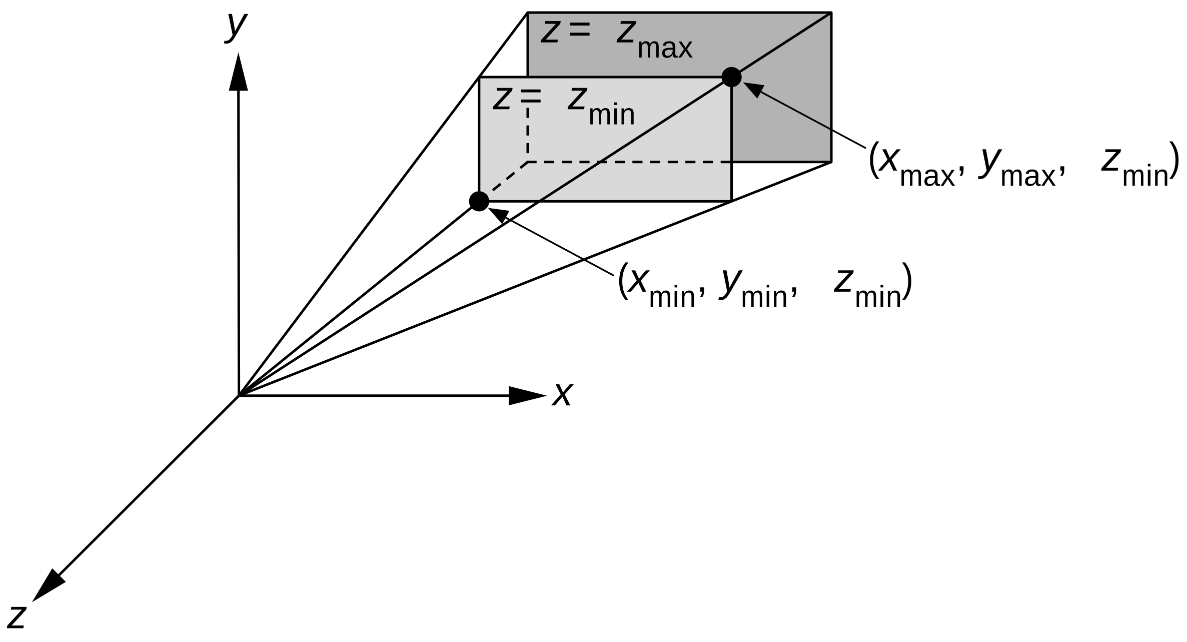
In addition to the projections themselves, we need to specify a **clipping volume** or **view volume**, by specifying front and back clipping frames. For perspective projections, the viewing volume becomes a **frustum**, a truncated pyramid.



In OpenGL, we can specify a perspective view by constructing and loading the matrix ourselves, or by using one of two special functions. The first is

```
glFrustum(xmin, xmax, ymin, ymax, near, far)
```

which sets the frustum shown in the following figure.

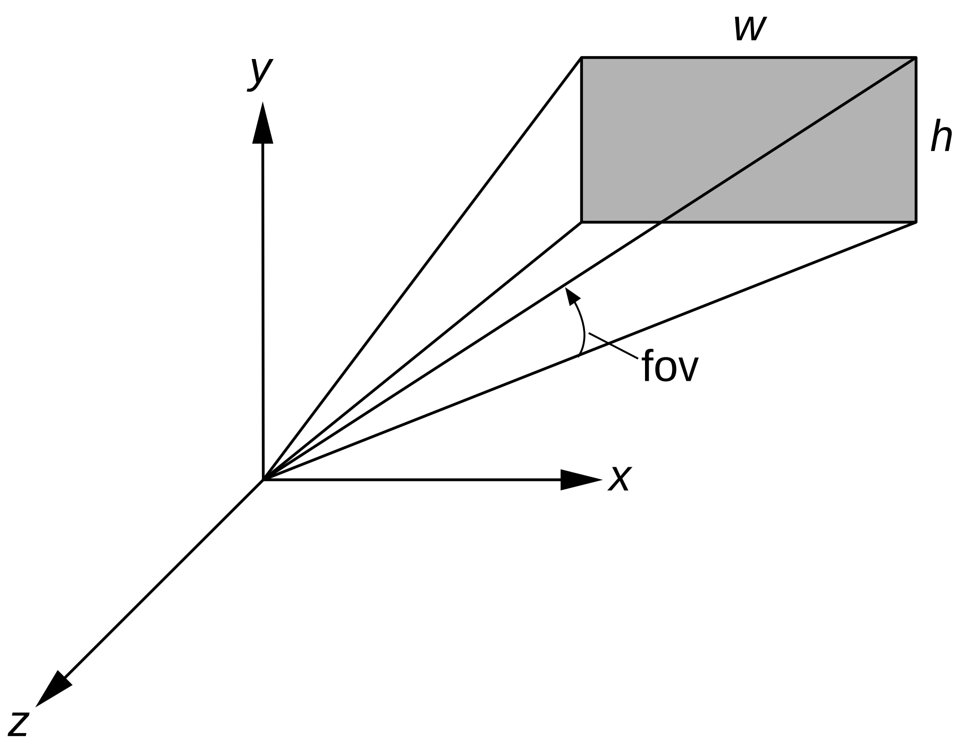


Here, z_{min} = −**near** and z_{max} = −**far** in the figure. The **near** and **far** distances must be positive, and so z_{min} and z_{max} are always negative. Note that **xmin**, **xmax**, **ymin**, **ymax** are the coordinates of the front clipping plane.

The second function for specifying a perspective projection with a clipping volume is

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

This function uses the angle of view **fovy** in the y direction and the aspect ratio — width divided by height — together with the near and far planes to completely determine the desired frustum.

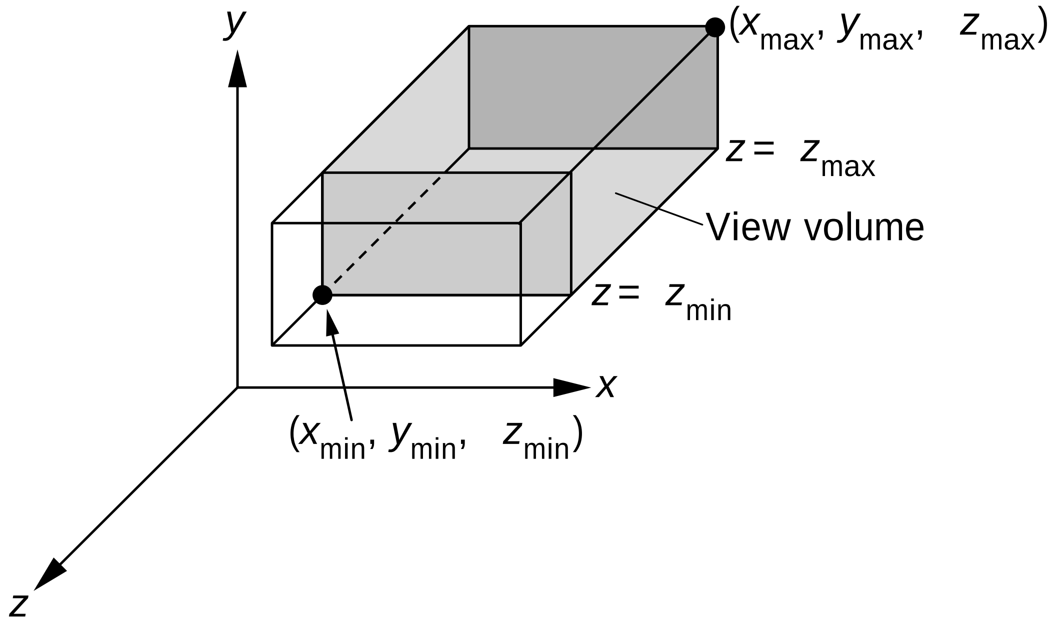


Orthogonal projections in OpenGL

To specify an orthogonal projection in OpenGL there is just one function:

```
glOrtho(xmin, xmax, ymin, ymax, near, far)
```

whose arguments are identical to those of `glFrustum`. The view volume is now a parallelepiped. We have set $z_{min} = -\mathbf{near}$ and $z_{max} = -\mathbf{far}$ again in the figure.


y
$(x_{\max}, y_{\max}, z_{\max})$
$z = z_{\max}$
View volume
$z = z_{\min}$
x
$(x_{\min}, y_{\min}, z_{\min})$
z

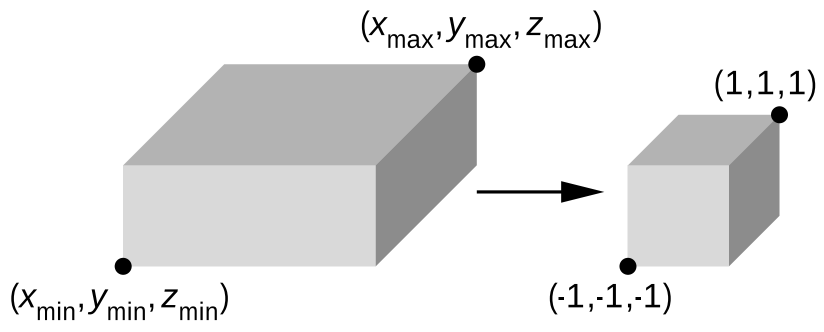
Unlike with perspective projections, the arguments **near** and **far** no longer have to be positive. The only restriction is that **near** < **far**.

Orthogonal projection matrices

The projection matrices used in OpenGL must also take account of the clipping volume. We begin with the projection matrix for the function

```
    glOrtho(xmin, xmax, ymin, ymax, near, far)
```

Set $z_{min} = -near$, $z_{max} = -far$. All OpenGL needs to do is to map the chosen view volume into the canonical one: $x = \pm 1$, $y = \pm 1$, $z = \pm 1$.



This can be achieved by concatenating a translation and a scaling:

$$T\Big(-(x_{min}+x_{max})/2, -(y_{min}+y_{max})/2, -(z_{min}+z_{max})/2\Big),$$

$$S\Big(2/(x_{max}-x_{min}), 2/(y_{max}-y_{min}), 2/(z_{max}-z_{min})\Big),$$

giving the projection matrix

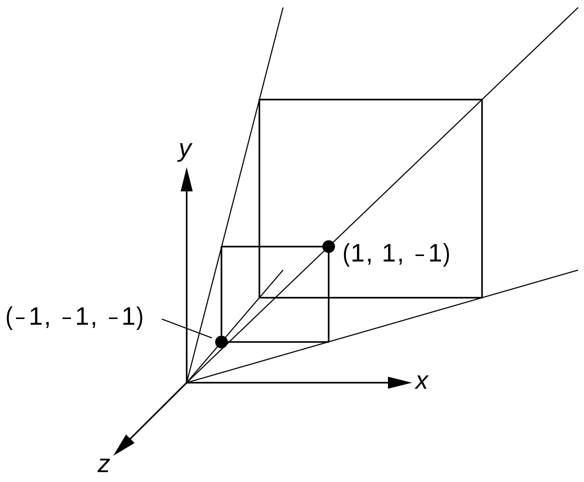
$$P = ST = \begin{pmatrix} \frac{2}{x_{max}-x_{min}} & 0 & 0 & -\frac{x_{min}+x_{max}}{x_{max}-x_{min}} \\ 0 & \frac{2}{y_{max}-y_{min}} & 0 & -\frac{y_{min}+y_{max}}{y_{max}-y_{min}} \\ 0 & 0 & \frac{2}{z_{max}-z_{min}} & -\frac{z_{min}+z_{max}}{z_{max}-z_{min}} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Perspective projection matrices

We now find the projection matrix in OpenGL specified by the function

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

Again set $z_{min} = -near$, $z_{max} = -far$. Assume for simplicity that $fovy = 90$ and $aspect = 1$:



From our previous discussion, the simple matrix

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

provides the correct perspective projectors but sends all points into the plane $z = -1$.

We need a more general matrix,

$$N = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{pmatrix},$$

in order to transform the front clipping plane $z = z_{min}$ into the canonical plane $z = 1$ and the back clipping plane $z = z_{max}$ into $z = -1$. The matrix N clearly maps the point $(x, y, z, 1)$ to the point (x', y', z', w') where

$$x' = x, \qquad y' = y, \qquad z' = \alpha z + \beta, \qquad w' = -z.$$

After dividing by w', we get

$$x'' = -x/z, \qquad y'' = -y/z, \qquad z'' = -(\alpha + \beta/z),$$

which means that the planes $x = \pm z$ and $y = \pm z$ are transformed into the planes $x'' = \pm 1$ and $y'' = \pm 1$, for any α, β. On the other hand, the planes $z = z_{min}$ and $z = z_{max}$ are mapped respectively into the planes

$$z'' = -(\alpha + \beta/z_{min}), \qquad z'' = -(\alpha + \beta/z_{max}).$$

So we have to solve the simultaneous equations

$$-(\alpha + \beta/z_{min}) = 1, \qquad -(\alpha + \beta/z_{max}) = -1,$$

for α, β, in order to map the front clipping plane $z = z_{min}$ into the canonical plane $z = 1$ and the back clipping plane $z = z_{max}$ into $z = -1$. The solution is

$$\alpha = -\frac{z_{max} + z_{min}}{z_{max} - z_{min}}, \qquad \beta = -\frac{2 z_{max} z_{min}}{z_{max} - z_{min}}.$$