6 Camera Models

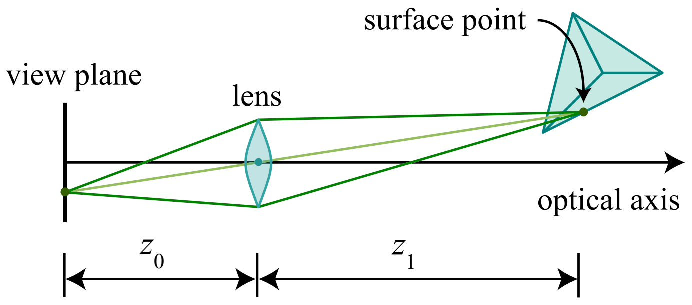
Goal: To model basic geometry of projection of 3D points, curves, and surfaces onto a 2D surface, the **view plane** or **image plane**.

6.1 Thin Lens Model

Most modern cameras use a lens to focus light onto the view plane (i.e., the sensory surface). This is done so that one can capture enough light in a sufficiently short period of time that the objects do not move appreciably, and the image is bright enough to show significant detail over a wide range of intensities and contrasts.

Aside:
In a conventional camera, the view plane contains either photoreactive chemicals; in a digital camera, the view plane contains a charge-coupled device (CCD) array. (Some cameras use a CMOS-based sensor instead of a CCD). In the human eye, the view plane is a curved surface called the *retina*, and and contains a dense array of cells with photoreactive molecules.

Lens models can be quite complex, especially for compound lens found in most cameras. Here we consider perhaps the simplist case, known widely as the thin lens model. In the thin lens model, rays of light emitted from a point travel along paths through the lens, convering at a point behind the lens. The key quantity governing this behaviour is called the *focal length* of the lens. The focal length,, $|f|$, can be defined as distance behind the lens to which rays from an infinitely distant source converge in focus.

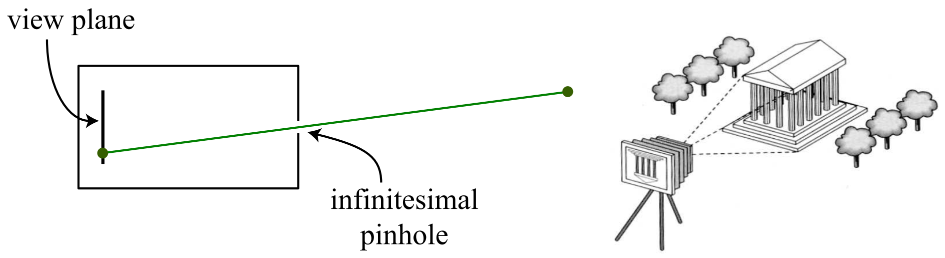


More generally, for the thin lens model, if z_1 is the distance from the center of the lens (i.e., the nodal point) to a surface point on an object, then for a focal length $|f|$, the rays from that surface point will be in focus at a distance z_0 behind the lens center, where z_1 and z_0 satisfy the thin lens equation:

$$\frac{1}{|f|} = \frac{1}{z_0} + \frac{1}{z_1} \tag{1}$$

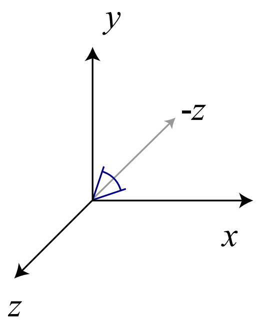
6.2 Pinhole Camera Model

A **pinhole** camera is an idealization of the thin lens as aperture shrinks to zero.

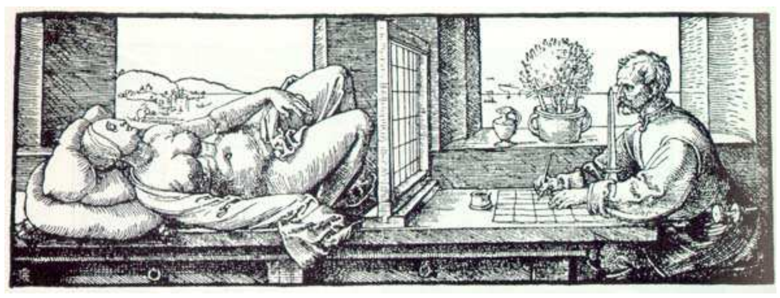


Light from a point travels along a single straight path through a pinhole onto the view plane. The object is imaged upside-down on the image plane.

Note:
We use a right-handed coordinate system for the camera, with the x-axis as the horizontal direction and the y-axis as the vertical direction. This means that the optical axis (gaze direction) is the negative z-axis.

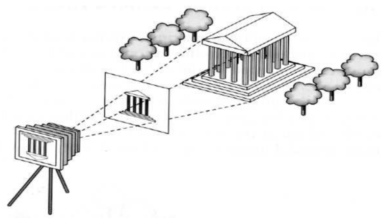


Here is another way of thinking about the pinhole model. Suppose you view a scene with one eye looking through a square window, and draw a picture of what you see through the window:



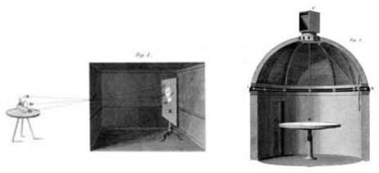
(Engraving by Albrecht Dürer, 1525).

The image you'd get corresponds to drawing a ray from the eye position and intersecting it with the window. This is equivalent to the pinhole camera model, except that the view plane is in front of the eye instead of behind it, and the image appears rightside-up, rather than upside down. (The eye point here replaces the pinhole). To see this, consider tracing rays from scene points through a view plane behind the eye point and one in front of it:



For the remainder of these notes, we will consider this camera model, as it is somewhat easier to think about, and also consistent with the model used by OpenGL.

Aside:
The earliest cameras were room-sized pinhole cameras, called *camera obscura*s. You would walk in the room and see an upside-down projection of the outside world on the far wall. The word *camera* is Latin for "room;" *camera obscura* means "dark room."



18th-century camera obscuras. The camera on the right uses a mirror in the roof to project images of the world onto the table, and viewers may rotate the mirror.

6.3 Camera Projections

Consider a point $\bar{p}$ in 3D space oriented with the camera at the origin, which we want to project onto the view plane. To project p_y to y, we can use similar triangles to get $y = \frac{f}{p_z} p_y$. This is **perspective projection**.

Note that $f < 0$, and the focal length is $|f|$.

In perspective projection, distant objects appear smaller than near objects:

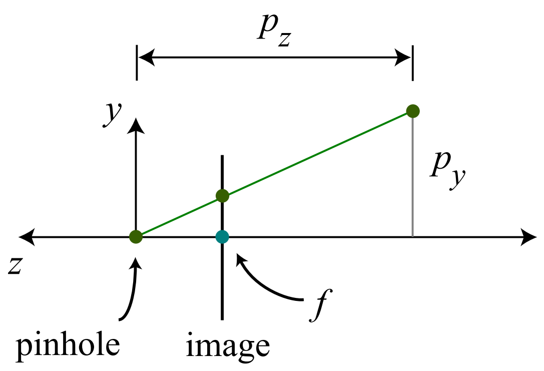
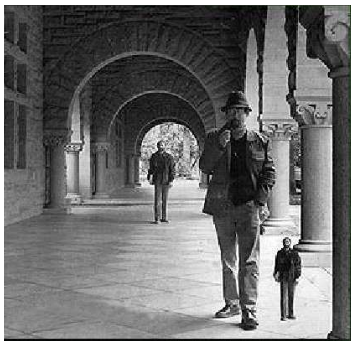


Figure 1: *

Perspective projection



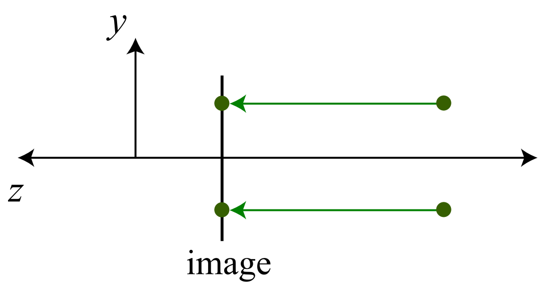
The man without the hat appears to be two different sizes, even though the two images of him have identical sizes when measured in pixels. In 3D, the man without the hat on the left is about 18 feet behind the man with the hat. This shows how much you might expect size to change due to perspective projection.

6.4 Orthographic Projection

For objects sufficiently far away, rays are nearly parallel, and variation in p_z is insignificant.

Here, the baseball players appear to be about the same height in pixels, even though the batter is about 60 feet away from the pitcher. Although this is an example of perspective projection, the camera is so far from the players (relative to the camera focal length) that they appear to be roughly the same size.

In the limit, $y = \alpha p_y$ for some real scalar α. This is **orthographic projection**:



6.5 Camera Position and Orientation

Assume camera coordinates have their origin at the "eye" (pinhole) of the camera, $\bar{e}$.

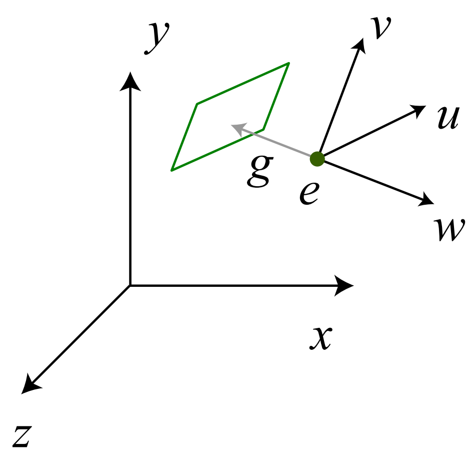


Figure 2:

Let $\vec{g}$ be the gaze direction, so a vector perpendicular to the view plane (parallel to the camera z-axis) is

$$\vec{w} = \frac{-\vec{g}}{\|\vec{g}\|} \tag{2}$$

We need two more orthogonal vectors $\vec{u}$ and $\vec{v}$ to specify a camera coordinate frame, with $\vec{u}$ and $\vec{v}$ parallel to the view plane. It may be unclear how to choose them directly. However, we can instead specify an "up" direction. Of course this up direction will not be perpendicular to the gaze direction.

Let $\vec{t}$ be the "up" direction (e.g., toward the sky so $\vec{t} = (0, 1, 0)$). Then we want $\vec{v}$ to be the closest vector in the viewplane to $\vec{t}$. This is really just the projection of $\vec{t}$ onto the view plane. And of course, $\vec{u}$ must be perpendicular to $\vec{v}$ and $\vec{w}$. In fact, with these definitions it is easy to show that $\vec{u}$ must also be perpendicular to $\vec{t}$, so one way to compute $\vec{u}$ and $\vec{v}$ from $\vec{t}$ and $\vec{g}$ is as follows:

$$\vec{u} = \frac{\vec{t} \times \vec{w}}{\|\vec{t} \times \vec{w}\|} \qquad\qquad \vec{v} = \vec{w} \times \vec{u} \tag{3}$$

Of course, we could have use many different "up" directions, so long as $\vec{t} \times \vec{w} \neq 0$.

Using these three basis vectors, we can define a **camera coordinate system**, in which 3D points are represented with respect to the camera's position and orientation. The camera coordinate system has its origin at the eye point $\bar{e}$ and has basis vectors $\vec{u}$, $\vec{v}$, and $\vec{w}$, corresponding to the x, y, and z axes in the camera's local coordinate system. This explains why we chose $\vec{w}$ to point away from the image plane: the right-handed coordinate system requires that z (and, hence, $\vec{w}$) point away from the image plane.

Now that we know how to represent the camera coordinate frame within the world coordinate frame we need to explicitly formulate the rigid transformation from world to camera coordinates. With this transformation and its inverse we can easily express points either in world coordinates or camera coordinates (both of which are necessary).

To get an understanding of the transformation, it might be helpful to remember the mapping from points in camera coordinates to points in world coordinates. For example, we have the following correspondences between world coordinates and camera coordinates: Using such correspondences

Camera coordinates (x_c, y_c, z_c)	World coordinates (x, y, z)
$(0, 0, 0)$	$\bar{e}$
$(0, 0, f)$	$\bar{e} + f\vec{w}$
$(0, 1, 0)$	$\bar{e} + \vec{v}$
$(0, 1, f)$	$\bar{e} + \vec{v} + f\vec{w}$

it is not hard to show that for a general point expressed in camera coordinates as $\bar{p}^c = (x_c, y_c, z_c)$, the corresponding point in world coordinates is given by

$$\begin{aligned} \bar{p}^w &= \bar{e} + x_c\vec{u} + y_c\vec{v} + z_c\vec{w} && (4)\\ &= \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix}\bar{p}^c + \bar{e} && (5)\\ &= M_{cw}\bar{p}^c + \bar{e}. && (6) \end{aligned}$$

where

$$M_{cw} = \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix} = \begin{bmatrix} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{bmatrix} \tag{7}$$

Note: We can define the same transformation for points in homogeneous coordinates:

$$\hat{M}_{cw} = \begin{bmatrix} M_{cw} & \bar{e} \\ \vec{0}^T & 1 \end{bmatrix}.$$

Now, we also need to find the inverse transformation, i.e., from world to camera coordinates. Toward this end, note that the matrix M_{cw} is orthonormal. To see this, note that vectors $\vec{u}$, $\vec{v}$ and, $\vec{w}$ are all of unit length, and they are perpendicular to one another. You can also verify this by computing $M_{cw}^T M_{cw}$. Because M_{cw} is orthonormal, we can express the inverse transformation (from camera coordinates to world coordinates) as

$$\begin{aligned} \bar{p}^c &= M_{cw}^T(\bar{p}^w - \bar{e}) \\ &= M_{wc}\bar{p}^w - \bar{d}\,, \end{aligned}$$

where $M_{wc} = M_{cw}^T = \begin{bmatrix} \vec{u}^T \\ \vec{v}^T \\ \vec{w}^T \end{bmatrix}$. (why?), and $\bar{d} = M_{cw}^T \bar{e}$.

In homogeneous coordinates, $\hat{p}^c = \hat{M}_{wc}\hat{p}^w$, where

$$\begin{aligned} \hat{M}_v &= \begin{bmatrix} M_{wc} & -M_{wc}\bar{e} \\ \vec{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} M_{wc} & \vec{0} \\ \vec{0}^T & 1 \end{bmatrix} \begin{bmatrix} I & -\bar{e} \\ \vec{0}^T & 1 \end{bmatrix}. \end{aligned}$$

This transformation takes a point from world to camera-centered coordinates.

6.6 Perspective Projection

Above we found the form of the perspective projection using the idea of similar triangles. Here we consider a complementary algebraic formulation. To begin, we are given

- a point $\bar{p}^c$ in camera coordinates (uvw space),
- center of projection (eye or pinhole) at the origin in camera coordinates,
- image plane perpendicular to the z-axis, through the point $(0, 0, f)$, with $f < 0$, and

- line of sight is in the direction of the negative z-axis (in camera coordinates),

we can find the intersection of the ray from the pinhole to $\bar{p}^c$ with the view plane.

The ray from the pinhole to $\bar{p}^c$ is $\bar{r}(\lambda) = \lambda(\bar{p}^c - \bar{0})$.

The image plane has normal $(0,0,1) = \vec{n}$ and contains the point $(0,0,f) = \bar{f}$. So a point $\bar{x}^c$ is on the plane when $(\bar{x}^c - \bar{f}) \cdot \vec{n} = 0$. If $\bar{x}^c = (x^c, y^c, z^c)$, then the plane satisfies $z^c - f = 0$.

To find the intersection of the plane $z^c = f$ and ray $\vec{r}(\lambda) = \lambda\bar{p}^c$, substitute $\vec{r}$ into the plane equation. With $\bar{p}^c = (p^c_x, p^c_y, p^c_z)$, we have $\lambda p^c_z = f$, so $\lambda^* = f/p^c_z$, and the intersection is

$$\vec{r}(\lambda^*) = \left(f\frac{p^c_x}{p^c_z}, f\frac{p^c_y}{p^c_z}, f \right) = f\left(\frac{p^c_x}{p^c_z}, \frac{p^c_y}{p^c_z}, 1 \right) \equiv \bar{x}^*. \tag{8}$$

The first two coordinates of this intersection $\bar{x}^*$ determine the image coordinates.

2D points in the image plane can therefore be written as

$$\begin{bmatrix} x^* \\ y^* \end{bmatrix} = \frac{f}{p^c_z} \begin{bmatrix} p^c_x \\ p^c_y \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \frac{f}{p^c_z} \bar{p}^c.$$

The mapping from $\bar{p}^c$ to $(x^*, y^*, 1)$ is called **perspective projection**.

Note:
Two important properties of perspective projection are:

- Perspective projection preserves linearity. In other words, the projection of a 3D line is a line in 2D. This means that we can render a 3D line segment by projecting the endpoints to 2D, and then draw a line between these points in 2D.
- Perspective projection does not preserve parallelism: two parallel lines in 3D do not necessarily project to parallel lines in 2D. When the projected lines intersect, the intersection is called a **vanishing point**, since it corresponds to a point infinitely far away. Exercise: when do parallel lines project to parallel lines and when do they not?

Aside:
The discovery of linear perspective, including vanishing points, formed a cornerstone of Western painting beginning at the Renaissance. On the other hand, defying realistic perspective was a key feature of Modernist painting.

To see that linearity is preserved, consider that rays from points on a line in 3D through a pinhole all lie on a plane, and the intersection of a plane and the image plane is a line. That means to draw polygons, we need only to project the vertices to the image plane and draw lines between them.

6.7 Homogeneous Perspective

The mapping of $\bar{p}^c = (p^c_x, p^c_y, p^c_z)$ to $\bar{x}^* = \frac{f}{p^c_z}(p^c_x, p^c_y, p^c_z)$ is just a form of scaling transformation. However, the magnitude of the scaling depends on the depth p^c_z. So it's not linear.

Fortunately, the transformation can be expressed linearly (ie as a matrix) in homogeneous coordinates. To see this, remember that $\hat{p} = (\bar{p}, 1) = \alpha(\bar{p}, 1)$ in homogeneous coordinates. Using this property of homogeneous coordinates we can write $\bar{x}^*$ as

$$\hat{x}^* = \left(p^c_x, p^c_y, p^c_z, \frac{p^c_z}{f}\right).$$

As usual with homogeneous coordinates, when you scale the homogeneous vector by the inverse of the last element, when you get in the first three elements is precisely the perspective projection. Accordingly, we can express $\hat{x}^*$ as a linear transformation of $\hat{p}^c$:

$$\hat{x}^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \hat{p}^c \equiv \hat{M}_p \hat{p}^c.$$

Try multiplying this out to convince yourself that this all works.
Finally, $\hat{M}_p$ is called the homogeneous perspective matrix, and since $\hat{p}^c = \hat{M}_{wc}\hat{p}^w$, we have $\hat{x}^* = \hat{M}_p \hat{M}_{wc} \hat{p}^w$.

6.8 Pseudodepth

After dividing by its last element, $\hat{x}^*$ has its first two elements as image plane coordinates, and its third element is f. We would like to be able to alter the homogeneous perspective matrix $\hat{M}_p$ so that the third element of $\frac{p^c_z}{f}\hat{x}^*$ encodes depth while keeping the transformation linear.

Idea: Let $\hat{x}^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1/f & 0 \end{bmatrix} \hat{p}^c$, so $z^* = \frac{f}{p^c_z}(ap^c_z + b)$.

What should a and b be? We would like to have the following two constraints:

$$z^* = \begin{cases} -1 & \text{when } p^c_z = f \\ 1 & \text{when } p^c_z = F \end{cases},$$

where f gives us the position of the **near plane**, and F gives us the z coordinate of the **far plane**.

So $-1 = af + b$ and $1 = af + b\frac{f}{F}$. Then $2 = b\frac{f}{F} - b = b\left(\frac{f}{F} - 1\right)$, and we can find

$$b = \frac{2F}{f - F}.$$

Substituting this value for b back in, we get $-1 = af + \frac{2F}{f-F}$, and we can solve for a:

$$\begin{aligned} a &= -\frac{1}{f}\left(\frac{2F}{f-F} + 1\right) \\ &= -\frac{1}{f}\left(\frac{2F}{f-F} + \frac{f-F}{f-F}\right) \\ &= -\frac{1}{f}\left(\frac{f+F}{f-F}\right). \end{aligned}$$

These values of a and b give us a function $z^*(p_z^c)$ that increases monotonically as p_z^c decreases (since p_z^c is negative for objects in front of the camera). Hence, z^* can be used to sort points by depth.

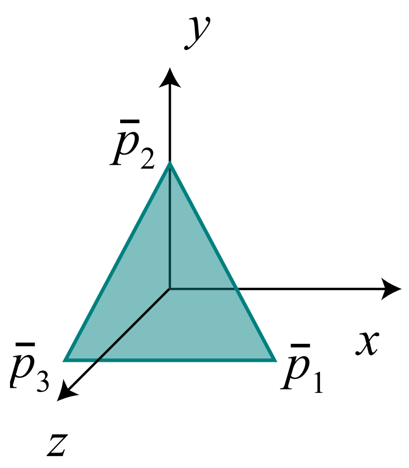
Why did we choose these values for a and b? Mathematically, the specific choices do not matter, but they are convenient for implementation. These are also the values that OpenGL uses.

What is the meaning of the near and far planes? Again, for convenience of implementation, we will say that only objects between the near and far planes are visible. Objects in front of the near plane are behind the camera, and objects behind the far plane are too far away to be visible. Of course, this is only a loose approximation to the real geometry of the world, but it is very convenient for implementation. The range of values between the near and far plane has a number of subtle implications for rendering in practice. For example, if you set the near and far plane to be very far apart in OpenGL, then Z-buffering (discussed later in the course) will be very inaccurate due to numerical precision problems. On the other hand, moving them too close will make distant objects disappear. However, these issues will generally not affect rendering simple scenes. (For homework assignments, we will usually provide some code that avoids these problems).

6.9 Projecting a Triangle

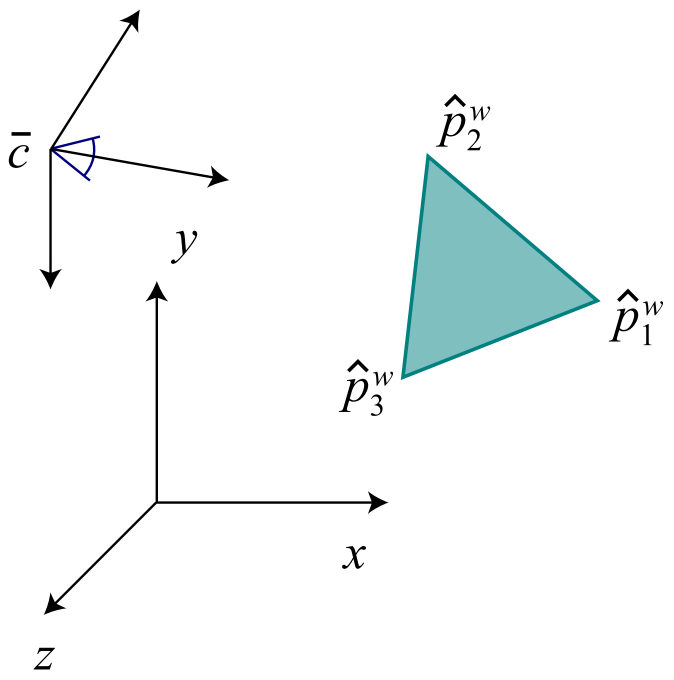
Let's review the steps necessary to project a triangle from object space to the image plane.

1. A triangle is given as three vertices in an object-based coordinate frame: $\bar{p}_1^o$, $\bar{p}_2^o$, $\bar{p}_3^o$.



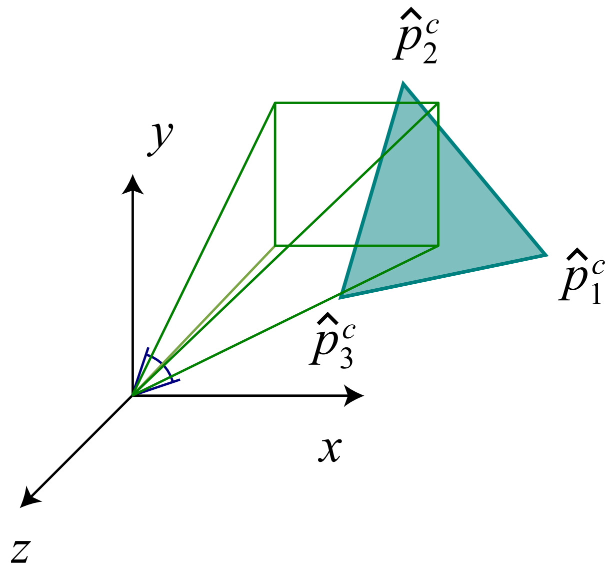
A triangle in object coordinates.

2. Transform to world coordinates based on the object's transformation: $\hat{p}_1^w$, $\hat{p}_2^w$, $\hat{p}_3^w$, where $\hat{p}_i^w = \hat{M}_{ow}\hat{p}_i^o$.



The triangle projected to world coordinates, with a camera at $\bar{c}$.

3. Transform from world to camera coordinates: $\hat{p}_i^c = \hat{M}_{wc}\hat{p}_i^w$.



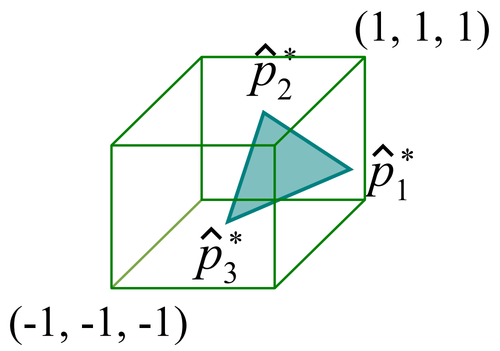
The triangle projected from world to camera coordinates.

4. Homogeneous perspective transformation: $\hat{x}_i^* = \hat{M}_p \hat{p}_i^c$, where

$$\hat{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1/f & 0 \end{bmatrix}, \text{ so } \hat{x}_i^* = \begin{bmatrix} p_x^c \\ p_y^c \\ ap_z^c + b \\ \frac{p_z^c}{f} \end{bmatrix}.$$

5. Divide by the last component:

$$\begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} = f \begin{bmatrix} \frac{p_x^c}{p_z^c} \\ \frac{p_y^c}{p_z^c} \\ \frac{ap_z^c+b}{p_z^c} \end{bmatrix}.$$



The triangle in normalized device coordinates after perspective division.

Now (x^*, y^*) is an image plane coordinate, and z^* is pseudodepth for each vertex of the triangle.

6.10 Camera Projections in OpenGL

OpenGL's modelview matrix is used to transform a point from object or world space to camera space. In addition to this, a *projection matrix* is provided to perform the homogeneous perspective transformation from camera coordinates to *clip coordinates* before performing perspective division. After selecting the projection matrix, the `glFrustum` function is used to specify a viewing volume, assuming the camera is at the origin:

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

For orthographic projection, `glOrtho` can be used instead:

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

The GLU library provides a function to simplify specifying a perspective projection viewing frustum:

```
gluPerspective(fieldOfView, aspectRatio, near, far);
```

The field of view is specified in degrees about the x-axis, so it gives the vertical visible angle. The aspect ratio should usually be the viewport width over its height, to determine the horizontal field of view.