Projection: Mapping 3-D to 2-D

Our scene models are in 3-D space and images are 2-D
- so we need some way of projecting 3-D to 2-D

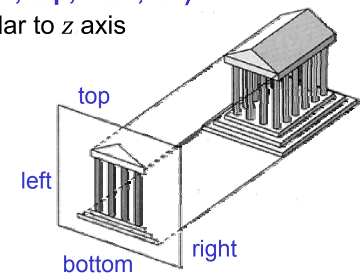
The fundamental approach: planar projection
- first, we define a plane in 3-D space
  - this is the image plane (or film plane)
- then project scene onto this plane
- and map to the window viewport

Need to address two basic issues
- how to define plane
- how to define mapping onto plane

Orthographic Projection

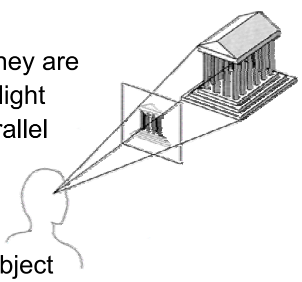
Arguably the simplest projection
- image plane is perpendicular to one of the coordinate axes
- project onto plane by dropping that coordinate
- $(x, y, z) \rightarrow (x, y)$ or $\rightarrow (x, z)$ or $\rightarrow (y, z)$

OpenGL — glOrtho(left, right, bottom, top, near, far)
- assumes image plane perpendicular to z axis
  - in other words, it's the xy-plane
- projects points $(x, y, z) \rightarrow (x, y)$
- also defines viewport mapping
  - defines rectangle on xy-plane
  - this gets mapped to window

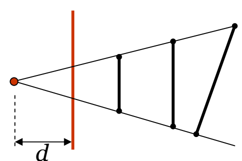


Perspective Projection

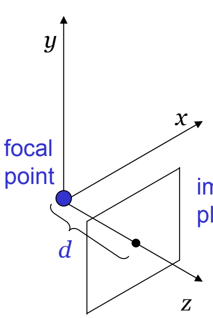
But we naturally see things in perspective
- objects appear smaller the farther away they are
- lenses bend (and hence focus) incoming light
- in orthographic projection, all rays are parallel



We've been using pinhole camera models
- draw rays thru focal point and points on object
- some of these lines will intersect the image plane
- this defines our projection into 2-D
- all points along a ray project to same point
- can project lines by projecting endpoints



The Canonical Camera Configuration



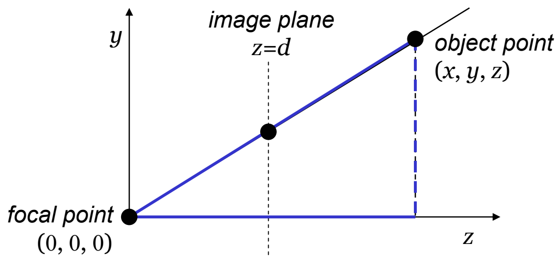
Want to derive perspective transformation
- in particular, a matrix representation

First, we fix a canonical camera
- focal point at origin
- looking along z axis
- image plane parallel to xy plane
- located distance d from origin
  - called the focal length

Effect of Perspective Projection on Points

We project points thru the line connecting them to the focal point

- given a point, we want to know where this line hits the image plane

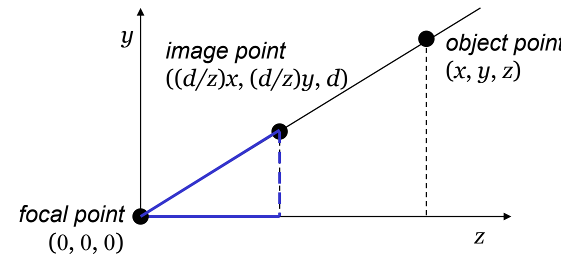


Effect of Perspective Projection on Points

We project points thru the line connecting them to the focal point

- given a point, we want to know where this line hits the image plane

Can easily compute this using similar triangles



Perspective Projection as a Transformation

This homogeneous matrix performs perspective projection

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

It's operation on any given point is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective Projection as a Transformation

This homogeneous matrix performs perspective projection

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

And when we do the homogeneous division

- we get exactly the point we want
- only keep x and y coordinates

$$\begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \Rightarrow \begin{bmatrix} \left(\frac{d}{z}\right)x \\ \left(\frac{d}{z}\right)y \\ d \\ 1 \end{bmatrix}$$

Completing the Projection

The image plane itself is infinite

- must map a rectangular region of it to the viewport
- defined by (*left, right, top, bottom*) coordinates

We also customarily define near & far clipping planes

- these are expressed as distances from the viewpoint
- they should always be positive
- nothing nearer than *near* will be drawn
  - don't want to draw things behind the image plane
- nothing further than *far* will be drawn
- distance far-near should be small
  - use fixed precision numbers to represent depth between them

OpenGL — glFrustum(left, right, bottom, top, near, far)

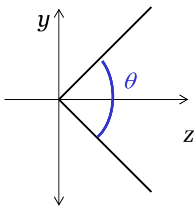
More Convenient Perspective Specification

Could always use glFrustum(left, right, bottom, top, near, far)

- this is certainly sufficient
- but it's inconvenient

Generally want to use: gluPerspective(fovy, aspect, near, far)

- viewport is always centered about z axis
- specifies the field of view along the y axis
  - the angle θ made by the sides of the frustum
- and the aspect ratio of the viewport
  - this is just (*width / height)*



Viewing Volumes

The sides of the viewport define an infinite pyramid

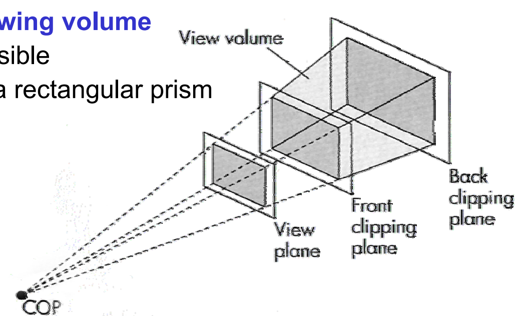
- focal point at apex, extending outward through space

Adding in the clipping planes, we get a truncated pyramid

- this is called a frustum

Can think of this as the viewing volume

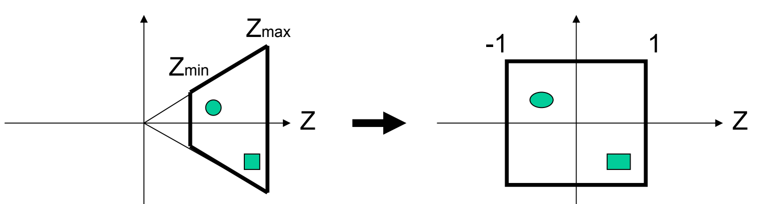
- nothing outside of it is visible
- projection warps this to a rectangular prism



Transformation for Viewing Volumes

$$\mathbf{P} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & \dfrac{z_{\max} + z_{\min}}{z_{\max} - z_{\min}} & \dfrac{-2 z_{\max} z_{\min}}{z_{\max} - z_{\min}} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Preserves depth order



We Need More General Cameras

So far, we've assumed a "canonical" camera configuration

- focal point at the origin
- image plane parallel to xy-plane

This is pretty limited, we want greater flexibility

- deriving general projection matrices is painful
- but we can transform world so camera is canonical
- typically called the viewing transformation

Naturally, there are several ways of setting this up

- we'll focus on the OpenGL supported mechanism
- the one in the book is gratuitously complex

Specifying General Camera Configurations

First, we want to allow focal point to be anywhere in space

- call this position *lookFrom*, or just *from*

Next, we need to specify the orientation of the camera

- define what it's pointing at: *lookAt*
  - *lookAt*–*lookFrom* will define the axis of projection
- define vertical axis of image: *vUp*
  - essentially a twist parameter about the *lookAt* axis

Converting Camera to Canonical Form

Our camera is parameterized by three vectors

- *lookFrom*, *lookAt*, and *vUp*

We want to transform into canonical camera position

1. translate *lookFrom* to the origin — translate by –*lookFrom*
2. rotate *lookAt*–*lookFrom* to the z axis

   Axis: $\mathbf{u} = (\mathit{lookAt} - \mathit{lookFrom}) \times \mathbf{z}$

   Angle: $\theta = \sin^{-1}(\|\mathbf{u}\| / L)$ where $L = \|\mathit{lookAt} - \mathit{lookFrom}\| \, \|\mathbf{z}\|$

3. rotate about z so that *vUp* lies inside the y-z plane

OpenGL Transformation Matrices

OpenGL maintains two different matrices

- one to hold the camera projection matrix
- and one to hold everything else
- select "current matrix" with glMatrixMode(*which*)
  - *which* is GL_MODELVIEW or GL_PROJECTION

glFrustum() and friends multiply the current matrix

- just like glTranslate(), glScale(), glRotate()

Vertices are transformed in the following manner

→ Model-view → Projection → Perspective division →

OpenGL Viewing Transformations

Specify camera configuration with

gluLookAt(ex, ey, ez, ax, ay, az, ux, uy, uz)

These are our three camera vectors

- *lookFrom* (ex, ey, ez)
- *lookAt* (ax, ay, az)
- *vUp* (ux, uy, uz)

Typical Transformation Setup:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fovy, aspect, zNear, zFar);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookat(ex, ey, ez, ax, ay, az, 0, 1, 0);
```

Demo

See "Links" web page for link to OpenGL tutors