Reading for This Module

• FCG Chapter 7 Viewing
• FCG Section 6.3.1 Windowing Transforms
• RB rest of Chap Viewing
• RB rest of App Homogeneous Coords
• RB Chap Selection and Feedback
• RB Sec Object Selection Using the Back Buffer
  • (in Chap Now That You Now )
Viewing
Using Transformations

• three ways
  • modelling transforms
    • place objects within scene (shared world)
    • affine transformations
  • viewing transforms
    • place camera
    • rigid body transformations: rotate, translate
  • projection transforms
    • change type of camera
    • projective transformation
Rendering Pipeline

- Scene graph
- Object geometry
- Modelling
  - Transforms
- Viewing
  - Transform
- Projection
  - Transform
Rendering Pipeline

- result
  - all vertices of scene in shared 3D world coordinate system
Rendering Pipeline

• result
  • scene vertices in 3D view (camera) coordinate system
Rendering Pipeline

- result
  - 2D *screen* coordinates of clipped vertices
Viewing and Projection

• need to get from 3D world to 2D image
• projection: geometric abstraction
  • what eyes or cameras do
• two pieces
  • viewing transform:
    • where is the camera, what is it pointing at?
  • perspective transform: 3D to 2D
    • flatten to image
Rendering Pipeline

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Geometry Database → Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer
Rendering Pipeline

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer
OpenGL Transformation Storage

• modeling and viewing stored together
  • possible because no intervening operations
• perspective stored in separate matrix

• specify which matrix is target of operations
  • common practice: return to default modelview mode after doing projection operations
    ```c
    glMatrixMode(GL_MODELVIEW);
    glMatrixMode(GL_PROJECTION);
    ```
Coordinate Systems

- result of a transformation
- names
  - convenience
    - animal: leg, head, tail
  - standard conventions in graphics pipeline
    - object/modelling
    - world
    - camera/viewing/eye
    - screen/window
    - raster/device
Projective Rendering Pipeline

OCS - object/model coordinate system
WCS - world coordinate system
VCS - viewing/camera/eye coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device/display/screen coordinate system

OCS $\rightarrow$ O2W
WCS $\rightarrow$ W2V
VCS $\rightarrow$ V2C

perspective divide

viewport transformation

C2N
N2D

clipping
CCS
normalized
device
NDCS
device
DCS
Viewing Transformation

Object → World → Viewing

OCS → WCS → VCS

Modeling transformation: $M_{mod}$

Viewing transformation: $M_{cam}$

OpenGL ModelView matrix
Basic Viewing

- starting spot - OpenGL
  - camera at world origin
    - probably inside an object
  - y axis is up
  - looking down negative z axis
    - why? RHS with x horizontal, y vertical, z out of screen
- translate backward so scene is visible
  - move distance d = focal length

- where is camera in P1 template code?
  - 5 units back, looking down -z axis
Convenient Camera Motion

- rotate/translate/scale versus
  - eye point, gaze/lookat direction, up vector

- demo: Robins transformation, projection
OpenGL Viewing Transformation

\[ \text{gluLookAt}(ex, ey, ez, lx, ly, lz, ux, uy, uz) \]

- postmultiplies current matrix, so to be safe:

\[
\begin{align*}
\text{glMatrixMode}(\text{GL_MODELVIEW}); \\
\text{glLoadIdentity}(); \\
\text{gluLookAt}(ex, ey, ez, lx, ly, lz, ux, uy, uz) \\
// \text{now ok to do model transformations}
\end{align*}
\]

- demo: Nate Robins tutorial \textit{projection}
Convenient Camera Motion

• rotate/translate/scale versus
  • eye point, gaze/lookat direction, up vector
Placing Camera in World Coords: V2W

- treat camera as if it’s just an object
  - translate from origin to eye
  - rotate view vector (lookat – eye) to w axis
  - rotate around w to bring up into vw-plane
Deriving V2W Transformation

- translate origin to eye

\[ T = \begin{bmatrix}
1 & 0 & 0 & e_x \\
0 & 1 & 0 & e_y \\
0 & 0 & 1 & e_z \\
0 & 0 & 0 & 1
\end{bmatrix} \]
Deriving V2W Transformation

- rotate view vector ($\text{lookat} - \text{eye}$) to $w$ axis

- $w$: normalized opposite of view/gaze vector $g$

$$w = -\hat{g} = -\frac{g}{\|g\|}$$
Deriving V2W Transformation

• rotate around \( w \) to bring \( \text{up} \) into \( vw \)-plane
  • \( u \) should be perpendicular to \( vw \)-plane, thus perpendicular to \( w \) and \( \text{up} \) vector \( t \)
  • \( v \) should be perpendicular to \( u \) and \( w \)

\[
\begin{align*}
  u &= \frac{t \times w}{\|t \times w\|} \\
  v &= w \times u
\end{align*}
\]
Deriving V2W Transformation

- rotate from WCS \( xyz \) into \( uvw \) coordinate system with matrix that has columns \( u, v, w \)

\[
\begin{align*}
\mathbf{u} &= \frac{\mathbf{t} \times \mathbf{w}}{|\mathbf{t} \times \mathbf{w}|} \\
\mathbf{v} &= \mathbf{w} \times \mathbf{u} \\
\mathbf{w} &= -\hat{\mathbf{g}} = -\frac{\mathbf{g}}{|\mathbf{g}|}
\end{align*}
\]

\[
\mathbf{T} = \begin{bmatrix}
1 & 0 & 0 & e_x \\
0 & 1 & 0 & e_y \\
0 & 0 & 1 & e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\mathbf{R} = \begin{bmatrix}
u_x & v_x & w_x & 0 \\
u_y & v_y & w_y & 0 \\
u_z & v_z & w_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\mathbf{M}_{V2W} = \mathbf{T}\mathbf{R}
\]

- reminder: rotate from \( uvw \) to \( xyz \) coord sys with matrix \( \mathbf{M} \) that has columns \( u,v,w \)
V2W vs. W2V

- $M_{V2W} = TR$

- we derived position of camera as object in world
  - invert for `gluLookAt`: go from world to camera!
- $M_{W2V} = (M_{V2W})^{-1} = R^{-1} T^{-1}$

$$
\begin{bmatrix}
1 & 0 & 0 & e_x \\
0 & 1 & 0 & e_y \\
0 & 0 & 1 & e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_x & v_x & w_x & 0 \\
u_y & v_y & w_y & 0 \\
u_z & v_z & w_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & -e_x \\
0 & 1 & 0 & -e_y \\
0 & 0 & 1 & -e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

- inverse is transpose for orthonormal matrices
- inverse is negative for translations
V2W vs. W2V

- $M_{W2V} = (M_{V2W})^{-1} = R^{-1}T^{-1}$

\[
M_{\text{world2view}} = \begin{bmatrix}
    u_x & u_y & u_z & 0 \\
    v_x & v_y & v_z & 0 \\
    w_x & w_y & w_z & 0 \\
    0   & 0   & 0   & 1
\end{bmatrix} \begin{bmatrix}
    1 & 0 & 0 & e_x \\
    0 & 1 & 0 & e_y \\
    0 & 0 & 1 & e_z \\
    0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
    u_x & u_y & u_z & -e \cdot u \\
    v_x & v_y & v_z & -e \cdot v \\
    w_x & w_y & w_z & -e \cdot w \\
    0   & 0   & 0   & 1
\end{bmatrix}
\]

\[
M_{\text{W2V}} = \begin{bmatrix}
    u_x & u_y & u_z & -e_x \cdot u_x + -e_y \cdot u_y + -e_z \cdot u_z \\
    v_x & v_y & v_z & -e_x \cdot v_x + -e_y \cdot v_y + -e_z \cdot v_z \\
    w_x & w_y & w_z & -e_x \cdot w_x + -e_y \cdot w_y + -e_z \cdot w_z \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]
Moving the Camera or the World?

• two equivalent operations
  • move camera one way vs. move world other way
• example
  • initial OpenGL camera: at origin, looking along -z axis
  • create a unit square parallel to camera at z = -10
  • translate in z by 3 possible in two ways
    • camera moves to z = -3
      • Note OpenGL models viewing in left-hand coordinates
    • camera stays put, but world moves to -7
• resulting image same either way
  • possible difference: are lights specified in world or view coordinates?
World vs. Camera Coordinates Example

\[ a = (1,1) \_{w} \]
\[ b = (1,1) \_{C1} = (5,3) \_{w} \]
\[ c = (1,1) \_{C2} = (1,3) \_{C1} = (5,5) \_{w} \]
Projections I
Pinhole Camera

- ingredients
  - box, film, hole punch
- result
  - picture

www.kodak.com
www.pinhole.org
www.debevec.org/Pinhole
Pinhole Camera

- theoretical perfect pinhole
- light shining through tiny hole into dark space yields upside-down picture

![Diagram of pinhole camera with light ray and film plane]
Pinhole Camera

- non-zero sized hole
- blur: rays hit multiple points on film plane
Real Cameras

- pinhole camera has small aperture (lens opening)
  - minimize blur

- problem: hard to get enough light to expose the film

- solution: lens
  - permits larger apertures
  - permits changing distance to film plane without actually moving it
    - cost: limited depth of field where image is in focus

Graphics Cameras

- real pinhole camera: image inverted

- computer graphics camera: convenient equivalent
GeneralProjection

• image plane need not be perpendicular to view plane
Perspective Projection

- our camera must model perspective
Perspective Projection

- our camera must model perspective
Projective Transformations

• planar geometric projections
  • planar: onto a plane
  • geometric: using straight lines
  • projections: 3D -> 2D

• aka projective mappings

• counterexamples?
Projective Transformations

• properties
  • lines mapped to lines and triangles to triangles
  • parallel lines do NOT remain parallel
    • e.g. rails vanishing at infinity

• affine combinations are NOT preserved
  • e.g. center of a line does not map to center of projected line (perspective foreshortening)
Perspective Projection

- project all geometry
  - through common center of projection (eye point)
  - onto an image plane
Perspective Projection

- Projection plane
- Center of projection (eye point)
- How tall should this bunny be?
Basic Perspective Projection

similar triangles

$\frac{y'}{d} = \frac{y}{z} \Rightarrow y' = \frac{y \cdot d}{z}$

$x' = \frac{x \cdot d}{z}$

but $z' = d$

- nonuniform foreshortening
- not affine
Perspective Projection

- desired result for a point $[x, y, z, 1]^T$ projected onto the view plane:

$$ \frac{x'}{d} = \frac{x}{z}, \quad \frac{y'}{d} = \frac{y}{z} $$

$$ x' = \frac{x \cdot d}{z}, \quad y' = \frac{y \cdot d}{z}, \quad z' = d $$

- what could a matrix look like to do this?
Simple Perspective Projection Matrix

\[
\begin{bmatrix}
\frac{x}{z/d} \\
\frac{y}{z/d} \\
\frac{z/d}{d}
\end{bmatrix}
\]
Simple Perspective Projection Matrix

\[
\begin{bmatrix}
\frac{x}{z/d} \\
\frac{y}{z/d} \\
\frac{z/d}{d}
\end{bmatrix}
\]

is homogenized version of

\[
\begin{bmatrix}
x \\
y \\
z \\
z/d
\end{bmatrix}
\]

where \( w = z/d \)
Simple Perspective Projection Matrix

\[
\begin{bmatrix}
\frac{x}{z/d} \\
\frac{y}{z/d} \\
\frac{z}{z/d} \\
d
\end{bmatrix}
\]

is homogenized version of

where \( w = \frac{z}{d} \)

\[
\begin{bmatrix}
x \\
y \\
z \\
\frac{z}{d}
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]
Perspective Projection

- expressible with 4x4 homogeneous matrix
  - use previously untouched bottom row
- perspective projection is irreversible
  - many 3D points can be mapped to same (x, y, d) on the projection plane
  - no way to retrieve the unique z values
Moving COP to Infinity

• as COP moves away, lines approach parallel
• when COP at infinity, orthographic view
Orthographic Camera Projection

- camera’s back plane parallel to lens
- infinite focal length
- no perspective convergence
- just throw away z values

\[
\begin{bmatrix}
    x_p \\
    y_p \\
    z_p \\
    1
\end{bmatrix}
= 
\begin{bmatrix}
    1 & 0 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]
Perspective to Orthographic

- transformation of space
  - center of projection moves to infinity
  - view volume transformed
    - from frustum (truncated pyramid) to parallelepiped (box)
View Volumes

- specifies field-of-view, used for clipping
- restricts domain of $z$ stored for visibility test

perspective view volume

orthographic view volume
Canonical View Volumes

- standardized viewing volume representation

**Perspective**

**Orthographic**

- orthogonal
- parallel

\[ x \text{ or } y = \pm z \]
Why Canonical View Volumes?

• permits standardization
  • clipping
    • easier to determine if an arbitrary point is enclosed in volume with canonical view volume vs. clipping to six arbitrary planes

• rendering
  • projection and rasterization algorithms can be reused
Normalized Device Coordinates

• convention
  • viewing frustum mapped to specific parallelepiped
    • Normalized Device Coordinates (NDC)
      • same as clipping coords
    • only objects inside the parallelepiped get rendered
  • which parallelepiped?
    • depends on rendering system
Normalized Device Coordinates

left/right $x = +/- 1$, top/bottom $y = +/- 1$, near/far $z = +/- 1$

Camera coordinates

NDC

Frustum

$z = -n$  $z = -f$

$x = 1$

$x = -1$

$z = -1$  $z = 1$
Understanding Z

- z axis flip changes coord system handedness
- RHS before projection (eye/view coords)
- LHS after projection (clip, norm device coords)
Understanding Z

near, far always positive in OpenGL calls

```c
glOrtho(left, right, bot, top, near, far);
glFrustum(left, right, bot, top, near, far);
glPerspective(fovy, aspect, near, far);
```

perspective view volume

![Perspective View Volume Diagram](image)

orthographic view volume

![Orthographic View Volume Diagram](image)
Understanding Z

• why near and far plane?
  • near plane:
    • avoid singularity (division by zero, or very small numbers)
  • far plane:
    • store depth in fixed-point representation (integer), thus have to have fixed range of values (0…1)
    • avoid/reduce numerical precision artifacts for distant objects
Orthographic Derivation

- scale, translate, reflect for new coord sys
Orthographic Derivation

- scale, translate, reflect for new coord sys

\[ y' = a \cdot y + b \]

\[ y = \text{top} \rightarrow y' = 1 \]

\[ y = \text{bot} \rightarrow y' = -1 \]
Orthographic Derivation

- scale, translate, reflect for new coord sys

\[ y' = a \cdot y + b \]
\[ y = \text{top} \rightarrow y' = 1 \quad 1 = a \cdot \text{top} + b \]
\[ y = \text{bot} \rightarrow y' = -1 \quad -1 = a \cdot \text{bot} + b \]

\[ b = 1 - a \cdot \text{top}, b = -1 - a \cdot \text{bot} \]
\[ 1 - a \cdot \text{top} = -1 - a \cdot \text{bot} \]
\[ 1 - (-1) = -a \cdot \text{bot} - (-a \cdot \text{top}) \]
\[ 2 = a(-\text{bot} + \text{top}) \]
\[ a = \frac{2}{\text{top} - \text{bot}} \]

\[ 1 = \frac{2}{\text{top} - \text{bot}} \cdot \text{top} + b \]
\[ b = 1 - \frac{2 \cdot \text{top}}{\text{top} - \text{bot}} \]
\[ b = \frac{(\text{top} - \text{bot}) - 2 \cdot \text{top}}{\text{top} - \text{bot}} \]
\[ b = \frac{-\text{top} - \text{bot}}{\text{top} - \text{bot}} \]
Orthographic Derivation

- scale, translate, reflect for new coord sys

\[ y' = a \cdot y + b \]

\[ y = \text{top} \rightarrow y' = 1 \]
\[ y = \text{bot} \rightarrow y' = -1 \]

\[ a = \frac{2}{\text{top} - \text{bot}} \]
\[ b = -\frac{\text{top} + \text{bot}}{\text{top} - \text{bot}} \]

same idea for right/left, far/near
Orthographic Derivation

- scale, translate, reflect for new coord sys

\[
P = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & \frac{-\text{right} + \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bot}} & 0 & \frac{-\text{top} + \text{bot}}{\text{top} - \text{bot}} \\
0 & 0 & \frac{-2}{\text{far} - \text{near}} & \frac{-\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Orthographic Derivation

- **scale**, translate, reflect for new coord sys

\[
P = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} + \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bot}} & 0 & -\frac{\text{top} + \text{bot}}{\text{top} - \text{bot}} \\
0 & 0 & -\frac{2}{\text{far} - \text{near}} & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Orthographic Derivation

- scale, **translate**, reflect for new coord sys

\[
P = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & \frac{-\text{right} + \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bot}} & 0 & \frac{-\text{top} + \text{bot}}{\text{top} - \text{bot}} \\
0 & 0 & \frac{-2}{\text{far} - \text{near}} & \frac{-\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Orthographic Derivation

- scale, translate, reflect for new coord sys

\[ P = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} + \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bot}} & 0 & -\frac{\text{top} + \text{bot}}{\text{top} - \text{bot}} \\
0 & 0 & -\frac{2}{\text{far} - \text{near}} & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix} \]
Orthographic OpenGL

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bot, top, near, far);
Demo

• Brown applets: viewing techniques
  • parallel/orthographic cameras
  • projection cameras

• http://www.cs.brown.edu/exploratories/freeSoftware/catalogs/viewing_techniques.html
Projections II
Asymmetric Frustums

• our formulation allows asymmetry
• why bother?

Frustum

\[ z = -n \quad \text{and} \quad z = -f \]
Asymmetric Frusta

- our formulation allows asymmetry
- why bother? binocular stereo
  - view vector not perpendicular to view plane
Simpler Formulation

• left, right, bottom, top, near, far
  • nonintuitive
  • often overkill
• look through window center
  • symmetric frustum
• constraints
  • left = -right, bottom = -top
Field-of-View Formulation

• FOV in one direction + aspect ratio (w/h)
  • determines FOV in other direction
  • also set near, far (reasonably intuitive)
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

glFrustumum(left, right, bot, top, near, far);
or

glPerspective(fovy, aspect, near, far);
Demo: Frustum vs. FOV

- Nate Robins tutorial (take 2):
Projective Rendering Pipeline

OCS - object/model coordinate system
WCS - world coordinate system
VCS - viewing/camera/eye coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device/display/screen coordinate system
Projection Warp

• warp perspective view volume to orthogonal view volume
  • render all scenes with orthographic projection!
  • aka perspective warp
Perspective Warp

- perspective viewing frustum transformed to cube
- orthographic rendering of cube produces same image as perspective rendering of original
Predistortion
Projective Rendering Pipeline

OCS - object/model coordinate system
WCS - world coordinate system
VCS - viewing/camera/eye coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device/display/screen coordinate system

O2W - modeling transformation
W2V - viewing transformation
V2C - projection transformation
C2N - perspective divide
N2D - viewport transformation

object world viewing
OCS WCS VCS

80
Separate Warp From Homogenization

- warp requires only standard matrix multiply
  - distort such that orthographic projection of distorted objects is desired persp projection
    - w is changed
  - clip after warp, before divide
- division by w: homogenization
Perspective Divide Example

• specific example
• assume image plane at $z = -1$
• a point $[x,y,z,1]^T$ projects to $[-x/z,-y/z,-z/z,1]^T \equiv [x,y,z,-z]^T$
Perspective Divide Example

\[
T\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z \end{pmatrix} = \begin{pmatrix} -x/z \\ -y/z \\ -1 \end{pmatrix}
\]

- after homogenizing, once again \( w=1 \)
Perspective Normalization

- matrix formulation

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{d}{d-\alpha} & \frac{-\alpha \cdot d}{d-\alpha} \\
0 & 0 & \frac{1}{d} & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
= \begin{bmatrix}
\frac{x}{(z-\alpha) \cdot d} \\
\frac{y}{d} \\
\frac{z}{d} \\
1
\end{bmatrix}
\]

- warp and homogenization both preserve relative depth (z coordinate)
Demo

- Brown applets: viewing techniques
  - parallel/orthographic cameras
  - projection cameras

Perspective To NDCS Derivation

VCS

- x = left
- y = top
- z = -near
- y = bottom
- z = -far
- x = right

NDCS

- (1, 1, 1)
- (-1, -1, -1)
Perspective Derivation

simple example earlier:

\[
\begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{bmatrix} =
\begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 1/d & 0
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

complete: shear, scale, projection-normalization

\[
\begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{bmatrix} =
\begin{bmatrix}
    E & 0 & A & 0 \\
    0 & F & B & 0 \\
    0 & 0 & C & D \\
    0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]
Perspective Derivation

earlier:

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  w'
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 & | & x \\
  0 & 1 & 0 & 0 & | & y \\
  0 & 0 & 1 & 0 & | & z \\
  0 & 0 & 1/d & 0 & | & 1
\end{bmatrix}
\]

complete: shear, scale, projection-normalization

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  w'
\end{bmatrix} = \begin{bmatrix}
  E & 0 & A & 0 & | & x \\
  0 & F & B & 0 & | & y \\
  0 & 0 & C & D & | & z \\
  0 & 0 & -1 & 0 & | & 1
\end{bmatrix}
\]
Perspective Derivation

earlier:

\[
\begin{bmatrix}
  x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 1/d & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

complete: shear, scale, projection-normalization

\[
\begin{bmatrix}
  x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
  E & 0 & A & 0 \\
  0 & F & B & 0 \\
  0 & 0 & C & D \\
  0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]
Recorrection: Perspective Derivation

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  w'
\end{bmatrix} =
\begin{bmatrix}
  E & 0 & A & 0 \\
  0 & F & B & 0 \\
  0 & 0 & C & D \\
  0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

\[x' = Ex + Az \quad x = \text{left} \rightarrow x' / w' = -1\]
\[x' = Ex + Az \quad x = \text{right} \rightarrow x' / w' = 1\]
\[y' = Fy + Bz \quad y = \text{top} \rightarrow y' / w' = 1\]
\[y' = Fy + Bz \quad y = \text{bottom} \rightarrow y' / w' = -1\]
\[z' = Cz + D \quad z = -\text{near} \rightarrow z' / w' = -1\]
\[z = -\text{far} \rightarrow z' / w' = 1\]

\[y' = Fy + Bz, \quad \frac{y'}{w'} = \frac{Fy + Bz}{w'}, \quad 1 = \frac{Fy + Bz}{w'}, \quad 1 = \frac{Fy + Bz}{-z},\]
\[1 = F \frac{y}{-z} + B \frac{z}{-z}, \quad 1 = F \frac{y}{-z} - B, \quad 1 = F \frac{\text{top}}{-(-\text{near})} - B,\]
\[1 = F \frac{\text{top}}{\text{near}} - B\]
**Perspective Derivation**

- similarly for other 5 planes
- 6 planes, 6 unknowns

\[
\begin{bmatrix}
2n & r + l & 0 \\
\frac{r - l}{r - l} & 0 & 0 \\
\frac{2n}{t - b} & \frac{r - l}{t + b} & 0 \\
\frac{0}{t - b} & \frac{(f + n)}{-(f + n)} & \frac{-2fn}{-2fn} \\
\frac{0}{f - n} & \frac{-1}{f - n} & 0 \\
\end{bmatrix}
\]
Projective Rendering Pipeline

- **OCS** - object/model coordinate system
- **WCS** - world coordinate system
- **VCS** - viewing/camera/eye coordinate system
- **CCS** - clipping coordinate system
- **NDCS** - normalized device coordinate system
- **DCS** - device/display/screen coordinate system

**Transformations:**
- O2W (object to world)
- W2V (world to viewing)
- V2C (viewing to clipping)
- C2N (clipping to normalized device)
- N2D (normalized device to device)

**Mentioned Techniques:**
- Modeling transformation
- Viewing transformation
- Projection transformation
- Viewport transformation

**Perspective Operations:**
- Perspective divide
NDC to Device Transformation

- map from NDC to pixel coordinates on display
  - NDC range is $x = -1...1$, $y = -1...1$, $z = -1...1$
  - typical display range: $x = 0...500$, $y = 0...300$
    - maximum is size of actual screen
    - $z$ range max and default is $(0, 1)$, use later for visibility

```c
glViewport(0, 0, w, h);
glDepthRange(0, 1); // depth = 1 by default
```
Origin Location

• yet more (possibly confusing) conventions
  • OpenGL origin: lower left
  • most window systems origin: upper left
• then must reflect in y
• when interpreting mouse position, have to flip your y coordinates
N2D Transformation

• general formulation
  • reflect in y for upper vs. lower left origin
  • scale by width, height, depth
  • translate by width/2, height/2, depth/2
  • FCG includes additional translation for pixel centers at (.5, .5) instead of (0,0)
N2D Transformation

\[
\begin{bmatrix}
  x_D \\ y_D \\ z_D \\ 1
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & \frac{width}{2} - 1 & \frac{2}{width} & 0 & 0 & 0 & 1 \\
  0 & 1 & 0 & \frac{height}{2} - 1 & 0 & \frac{2}{height} & 0 & 0 & 0 \\
  0 & 0 & 1 & \frac{2}{depth} & 0 & 0 & \frac{2}{depth} & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1
\end{bmatrix} \begin{bmatrix}
  x_N \\ y_N \\ z_N \\ 1
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 & \frac{width(x_N + 1) - 1}{2} & \frac{2}{height(-y_N + 1) - 1} & 0 & 0 & 0 \\
  0 & -1 & 0 & 0 & 0 & \frac{2}{depth(z_N + 1)} & 0 & 0 & 1 \\
  0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1
\end{bmatrix}
\]

Display z range is 0 to 1. 
glDepthRange(n,f) can constrain further, but depth = 1 is both max and default

reminder: NDC z range is -1 to 1
Device vs. Screen Coordinates

- viewport/window location wrt actual display not available within OpenGL
  - usually don’t care
    - use relative information when handling mouse events, not absolute coordinates
  - could get actual display height/width, window offsets from OS
- loose use of terms: device, display, window, screen...
Projective Rendering Pipeline

OCS - object coordinate system
WCS - world coordinate system
VCS - viewing coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device coordinate system

modeling transformation
O2W

viewing transformation
W2V

projection transformation
V2C

viewport transformation
N2D

glVertex3f(x,y,z)
glTranslatef(x,y,z)
glRotatef(a,x,y,z)
....
gluLookAt(...)
glFrustum(...)
glutInitWindowSize(w,h)
glViewport(x,y,a,b)

alter w / w

object world viewing

clipping

normalized

device

device
Coordinate Systems

viewing
(4-space, W=1)

projection matrix

clipping
(4-space parallelepiped, with COP moved backwards to infinity

divide by w

normalized device
(3-space parallelepiped)

scale & translate

device
(3-space parallelepiped)

framebuffer
tracks in VCS:
left $x=-1$, $y=-1$
right $x=1$, $y=-1$

view volume
left = -1, right = 1
bot = -1, top = 1
near = 1, far = 4
Perspective Example

view volume
• left = -1, right = 1
• bot = -1, top = 1
• near = 1, far = 4

\[
\begin{bmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & \frac{t+b}{r-l} & 0 \\
0 & 0 & -\frac{(f+n)}{t-b} & \frac{-2fn}{r-l} \\
0 & 0 & -1 & \frac{f-n}{r-l}
\end{bmatrix}
 \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -\frac{5}{3} & -\frac{8}{3} \\
0 & 0 & -1 & 0
\end{bmatrix}
\]
Perspective Example

\[
\begin{bmatrix}
1 \\
-1 \\
-5z_{VCS}/3 - 8/3 \\
-z_{VCS}
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
1 \\
-5/3 \\
-1
\end{bmatrix}
\begin{bmatrix}
x_{NDCS} \\
y_{NDCS} \\
z_{NDCS}
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
-1
\end{bmatrix}
\]

\[
x_{NDCS} = -1/z_{VCS}
\]

\[
y_{NDCS} = 1/z_{VCS}
\]

\[
z_{NDCS} = \frac{5}{3} + \frac{8}{3z_{VCS}}
\]
OpenGL Example

```
# In CCS
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45, 1.0, 0.1, 200.0);

# In VCS
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -5.0);
glPushMatrix();
glTranslatef(4, 4, 0);
glutSolidTeapot(1);
glPopMatrix();
glTranslatef(2, 2, 0);
glutSolidTeapot(1);
```

Transformations that are applied to object first are specified last.
Reading for Next Time

• RB Chap Color

• FCG Sections 3.2-3.3

• FCG Chap 20 Color

• FCG Chap 21.2.2 Visual Perception (Color)
Viewing: More Camera Motion
Fly "Through The Lens": Roll/Pitch/Yaw
Viewing: Incremental Relative Motion

- how to move relative to current camera coordinate system?
  - what you see in the window
- computation in coordinate system used to draw previous frame is simple:
  - incremental change $I$ to current $C$
  - at time $k$, want $p' = I_k l_{k-1} l_{k-2} ... l_5 l_4 l_3 l_2 l_1 C p$
- each time we just want to premultiply by new matrix
  - $p' = I C p$
  - but we know that OpenGL only supports postmultiply by new matrix
    - $p' = C I p$
Viewing: Incremental Relative Motion

• sneaky trick: OpenGL modelview matrix has the info we want!
  • dump out modelview matrix from previous frame with glGetDoublev()
    • C = current camera coordinate matrix
  • wipe the matrix stack with glLoadIdentity()
  • apply incremental update matrix I
  • apply current camera coord matrix C
• must leave the modelview matrix unchanged by object transformations after your display call
  • use push/pop
• using OpenGL for storage and calculation
  • querying pipeline is expensive
    • but safe to do just once per frame
Caution: OpenGL Matrix Storage

• OpenGL internal matrix storage is columnwise, not rowwise
  a e i m
  b f j n
  c g k o
  d h l p
  • opposite of standard C/C++/Java convention
  • possibly confusing if you look at the matrix from glGetDoublev()!
Viewing: Virtual Trackball

• interface for spinning objects around
  • drag mouse to control rotation of view volume
    • orbit/spin metaphor
    • vs. flying/driving

• rolling glass trackball
  • center at screen origin, surrounds world
  • hemisphere “sticks up” in z, out of screen
  • rotate ball = spin world
Clarify: Virtual Trackball

- know screen click: \((x, y, 0)\)
- want to infer point on trackball: \((x, y, z)\)
  - ball is unit sphere, so \(||x, y, z|| = 1.0\)
  - solve for \(z\)
Clarify: Trackball Rotation

- user drags between two points on image plane
  - mouse down at $i_1 = (x, y)$, mouse up at $i_2 = (a, b)$
- find corresponding points on virtual ball
  - $p_1 = (x, y, z)$, $p_2 = (a, b, c)$
- compute rotation angle and axis for ball
  - axis of rotation is plane normal: cross product $p_1 \times p_2$
  - amount of rotation $\theta$ from angle between lines
    - $p_1 \cdot p_2 = |p_1| |p_2| \cos \theta$
Clarify: Trackball Rotation

- finding location on ball corresponding to click on image plane
  - ball radius $r$ is 1

![Diagram of tracking ball rotation](image)
Trackball Computation

• user defines two points
  • place where first clicked \( p_1 = (x, y, z) \)
  • place where released \( p_2 = (a, b, c) \)
• create plane from vectors between points, origin
  • axis of rotation is plane normal: cross product
    • \((p_1 - o) \times (p_2 - o)\): \(p_1 \times p_2\) if origin = \((0,0,0)\)
  • amount of rotation depends on angle between lines
    • \( p_1 \cdot p_2 = |p_1| |p_2| \cos \theta \)
    • \(|p_1 \times p_2| = |p_1| |p_2| \sin \theta \)
• compute rotation matrix, use to rotate world
Picking
Reading

• Red Book
  • Selection and Feedback Chapter
    • all
  • Now That You Know Chapter
    • only Object Selection Using the Back Buffer
Interactive Object Selection

- move cursor over object, click
  - how to decide what is below?
  - inverse of rendering pipeline flow
    - from pixel back up to object
- ambiguity
  - many 3D world objects map to same 2D point
- four common approaches
  - manual ray intersection
  - bounding extents
  - backbuffer color coding
  - selection region with hit list
Manual Ray Intersection

• do all computation at application level
  • map selection point to a ray
  • intersect ray with all objects in scene.

• advantages
  • no library dependence

• disadvantages
  • difficult to program
  • slow: work to do depends on total number and complexity of objects in scene
Bounding Extents

• keep track of axis-aligned bounding rectangles

• advantages
  • conceptually simple
  • easy to keep track of boxes in world space
Bounding Extents

- disadvantages
  - low precision
  - must keep track of object-rectangle relationship
- extensions
  - do more sophisticated bound bookkeeping
    - first level: box check.
    - second level: object check
Backbuffer Color Coding

- use backbuffer for picking
  - create image as computational entity
  - never displayed to user
- redraw all objects in backbuffer
  - turn off shading calculations
  - set unique color for each pickable object
    - store in table
  - read back pixel at cursor location
    - check against table
Backbuffer Color Coding

- **advantages**
  - conceptually simple
  - variable precision

- **disadvantages**
  - introduce 2x redraw delay
  - backbuffer readback *very* slow
Backbuffer Example

```c
for(int i = 0; i < 2; i++)
for(int j = 0; j < 2; j++) {
    glPushMatrix();
    switch (i*2+j) {
        case 0: glColor3ub(255,0,0);break;
        case 1: glColor3ub(0,255,0);break;
        case 2: glColor3ub(0,0,255);break;
        case 3: glColor3ub(250,0,250);break;
    }
    glTranslatef(i*3.0,0,-j * 3.0)
    glCallList(snowman_display_list);
    glPopMatrix();
}
```

```c
for(int i = 0; i < 2; i++)
for(int j = 0; j < 2; j++) {
    glPushMatrix();
    glTranslatef(i*3.0,0,-j * 3.0);
    glColor3f(1.0, 1.0, 1.0);
    glCallList(snowman_display_list);
    glPopMatrix();
}
```

http://www.lighthouse3d.com/opengl/picking/
Select.Hit

- use small region around cursor for viewport
- assign per-object integer keys (names)
- redraw in special mode
- store hit list of objects in region
- examine hit list

- OpenGL support
Viewport

• small rectangle around cursor
  • change coord sys so fills viewport

• why rectangle instead of point?
  • people aren’t great at positioning mouse
    • Fitts’ Law: time to acquire a target is function of the distance to and size of the target
  • allow several pixels of slop
Viewport

- nontrivial to compute
  - invert viewport matrix, set up new orthogonal projection

- simple utility command
  - gluPickMatrix(x, y, w, h, viewport)
    - x, y: cursor point
    - w, h: sensitivity/slop (in pixels)
  - push old setup first, so can pop it later
Render Modes

- `glRenderMode(mode)`
  - `GL_RENDER`: normal color buffer
    - default
  - `GL_SELECT`: selection mode for picking
  - `(GL_FEEDBACK: report objects drawn)`
Name Stack

- again, "names" are just integers
  - glInitNames()
- flat list
  - glLoadName(name)
- or hierarchy supported by stack
  - glPushName(name), glPopName
- can have multiple names per object
Hierarchical Names Example

for(int i = 0; i < 2; i++) {
    glPushName(i);
    for(int j = 0; j < 2; j++) {
        glPushMatrix();
        glPushName(j);
        glTranslatef(i*10.0,0,j * 10.0);
        glPushName(HEAD);
        glCallList(snowManHeadDL);
        glLoadName(BODY);
        glCallList(snowManBodyDL);
        glPopName();
        glPopName();
        glPopMatrix();
    }
    glPopName();
}

http://www.lighthouse3d.com/opengl/picking/
Hit List

• `glSelectBuffer(buffersize, *buffer)`
  • where to store hit list data
• on hit, copy entire contents of name stack to output buffer.
• hit record
  • number of names on stack
  • minimum and maximum depth of object vertices
    • depth lies in the NDC z range \([0,1]\)
    • format: multiplied by \(2^{32} - 1\) then rounded to nearest int
Integrated vs. Separate Pick Function

- **integrate**: use same function to draw and pick
  - simpler to code
  - name stack commands ignored in render mode
- **separate**: customize functions for each
  - potentially more efficient
  - can avoid drawing unpickable objects
Select/Hit

• advantages
  • faster
    • OpenGL support means hardware acceleration
    • avoid shading overhead
  • flexible precision
    • size of region controllable
  • flexible architecture
    • custom code possible, e.g. guaranteed frame rate

• disadvantages
  • more complex
Hybrid Picking

- select/hit approach: fast, coarse
  - object-level granularity
- manual ray intersection: slow, precise
  - exact intersection point
- hybrid: both speed and precision
  - use select/hit to find object
  - then intersect ray with that object
OpenGL Precision Picking Hints

• **gluUnproject**
  - transform window coordinates to object coordinates given current projection and modelview matrices
  - use to create ray into scene from cursor location
  - call gluUnProject twice with same (x,y) mouse location
    - \( z = \text{near}: (x,y,0) \)
    - \( z = \text{far}: (x,y,1) \)
    - subtract near result from far result to get direction vector for ray
  - use this ray for line/polygon intersection
Projective Rendering Pipeline

Following pipeline from top/left to bottom/right: moving object POV

OCS - object coordinate system
WCS - world coordinate system
VCS - viewing coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device coordinate system

glVertex3f(x,y,z)

modeling transformation

O2W

w

W2V

WCS

viewing transformation

glTranslatef(x,y,z)
glRotatef(a,x,y,z)

...-

viewing

VCS

gluLookAt(...)

V2C

projection transformation

C2N / w

perspective division

glFrustum(...)

CCS

normalization

CCS

glutInitWindowSize(w,h)

N2D

glViewport(x,y,a,b)

viewport

transformation

device

DCS

135
OpenGL Example

go back from end of pipeline to beginning: coord frame POV!

object world viewing clipping
OCS O2W WCS W2V VCS V2C CCS

modeling transformation viewing transformation projection transformation

CCS
- glMatrixMode(GL_PROJECTION);
- glLoadIdentity();
- gluPerspective(45, 1.0, 0.1, 200.0);

VCS
- glMatrixMode(GL_MODELVIEW);
- glLoadIdentity();
- glTranslatef(0.0, 0.0, -5.0);

WCS
- glPushMatrix();
- glTranslatef(4, 4, 0);

OCS1
- glutSolidTeapot(1);
- glPopMatrix();
- glTranslatef(2, 2, 0);

OCS2
- glutSolidTeapot(1);

• transformations that are applied to object first are specified last
Coord Sys: Frame vs Point

read down: transforming between coordinate frames, from frame A to frame B
read up: transforming points, up from frame B coords to frame A coords

OpenGL command order

D2N  DCS display  glVertex3f(x,y,z)
N2V  NDCS normalized device  glFrustum(...)  glRotatef(a,x,y,z)
V2W  VCS viewing  gluLookAt(...)  glVertex3f(x,y,z)
W2O  WCS world  glFrustum(...)  glVertex3f(x,y,z)
O2W  OCS object  glVertex3f(x,y,z)
Coord Sys: Frame vs Point

- is **gluLookat** viewing transformation **V2W** or **W2V**? depends on which way you read!
  - coordinate frames: **V2W**
    - takes you from view to world coordinate frame
  - points/objects: **W2V**
    - point is transformed from world to view coords when multiply by **gluLookAt** matrix

- **H2** uses the object/pipeline POV
  - Q1/4 is **W2V** (**gluLookAt**)
  - Q2/5-6 is **V2N** (**glFrustum**)
  - Q3/7 is **N2D** (**glViewport**)