Chap. 5
3D Viewing and Projections
4BA6 - Topic 4
Dr. Steven Collins
References

- “Computer graphics: principles & practice”, Foley, vanDam, Feiner, Hughes, S-LEN 500.1644 M23*1;1-6 (has a good appendix on linear algebra)
- “Advanced Animation and Rendering Techniques”, Watt and Watt, S-LEN 500.18 N26;2-5
- “The OpenGL Programming Guide”, Woo, Neider & Davis, S-LEN 500.18 N72;0-2
- “Interactive Computer Graphics”, Edward Angel
OpenGL® Geometry Pipeline

original vertex

MODELVIEW matrix

vertex in the eye coordinate space

PROJECTION matrix

2D projection of vertex onto viewing plane

perspective division

normalised device coordinates (foreshortened)

viewport transformation

final window coordinates

\[
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_{eye} \\
y_{eye} \\
z_{eye} \\
w_{eye}
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_{proj} \\
y_{proj} \\
z_{proj} \\
w_{proj}
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_{dev} \\
y_{dev} \\
w_{dev}
\end{bmatrix}
\]
The Camera System

- To create a view of a scene we need:
  - a description of the scene geometry
  - a camera or view definition
- Default OpenGL camera is located at the origin looking down the \(-z\) axis.
- The camera definition allows projection of the 3D scene geometry onto a 2D surface for display.
- This projection can take a number of forms:
  - *orthographic* (parallel lines preserved)
  - *perspective* (foreshortening): 1-point, 2-point or 3-point
  - *skewed orthographic*
Camera Types

- Before generating an image we must choose our viewer:
  - The *pinhole camera model* is most widely used:
    - infinite *depth of field* (everything is in focus)
  - Advanced rendering systems model the camera
    - *double gauss lens* as used in many professional cameras
    - model depth of field and non-linear optics (including *lens flare*)
  - *Photorealistic rendering systems* often employ a physical model of the eye for rendering images
    - model the eyes response to varying *brightness* and *colour* levels
    - model the internal optics of the eye itself (*diffraction* by lens fibres etc.)
Pinhole Camera Model

- Imaging Surface
- Pinhole Aperture
- Viewing Frustum
- Scene
Modeling the Eye’s Response

Adaptation

Glare & Diffraction
Camera Systems

Double Gauss Lens

Viewing System

- We are only concerned with the *geometry* of viewing at this stage.
- The camera’s position and orientation define a *view-volume* or *view-frustrum*.
  - objects completely or partially within this volume are potentially visible on the viewport.
  - objects fully outside this volume cannot be seen ⇒ clipped
Camera Models

- Each vertex in our model must be projected onto the 2D camera viewport plane in order to be displayed on the screen.
- The $CTM$ is employed to determine the location of each vertex in the camera coordinate system:
  \[
  \bar{x}' = M_{CTM} \bar{x}
  \]
- We then employ a projection matrix defined by `GL_PROJECTION` to map this to a 2D viewport coordinate.
- Finally, this 2D coordinate is mapped to device coordinates using the viewport definition (given by `glViewport()`).
Camera Modeling in OpenGL

camera coordinate system \rightarrow \text{viewport coordinate system} \rightarrow \text{device/screen coordinate system}

glMatrixMode(GL_MODELVIEW)
...  
glViewport(0,0,xres,yres)

\( (0,0) \rightarrow (xres,yres) \)

glMatrixMode(GL_PROJECTION)
...
3D → 2D Projection

- Type of projection depends on a number of factors:
  - location and orientation of the viewing plane (viewport)
  - direction of projection (described by a vector)
  - projection type:
    - Perspective
      - 1-point
      - 2-point
      - 3-point
    - Parallel
      - Orthographic
      - Axonometric
    - Oblique
Parallel Projections

axonometric

orthographic

oblique
Perspective Projections

3-point perspective

1-point perspective

2-point perspective
Orthogonal Projections

- The simplest of all projections, *parallel project* onto view-plane.
- Usually view-plane is *axis aligned* (often at $z=0$)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\Rightarrow \bar{P} = MP \text{ where } M =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Multiple Projections

- It is often useful to have *multiple projections* available at any given time
  - usually: plan (top) view, front & left or right elevation (side) view
Orthogonal Projections

- The result is an orthographic projection if the object is axis aligned, otherwise it is an axonometric projection.
- If the projection plane intersects the principle axes at the same distance from the origin the projection is isometric.
Parallel Projections in OpenGL

\texttt{glOrtho}(xmin, xmax, ymin, ymax, zmin, zmax);

\textbf{Note}: we always view in -z direction need to transform world in order to view in other arbitrary directions.
Perspective Projections

- Perspective projections are more complex and exhibit *foreshortening* (parallel appear to converge at points).

- Parameters:
  - centre of projection (COP)
  - field of view \((\theta, \phi)\)
  - projection direction
  - up direction
Perspective Projections

Consider a perspective projection with the viewpoint at the origin and a viewing direction oriented along the positive -z axis and the view-plane located at \( z = -d \)

\[
\frac{y}{z} = \frac{y_p}{d} \Rightarrow y_p = \frac{y}{z/d}
\]

a similar construction for \( x_p \) \( \Rightarrow \)

\[
\begin{bmatrix}
  x_p \\
  y_p \\
  z_p \\
  1
\end{bmatrix} = \begin{bmatrix}
  \frac{x}{z/d} \\
  \frac{y}{z/d} \\
  \frac{z}{d} \\
  1
\end{bmatrix} \leftrightarrow \begin{bmatrix}
  x \\
  y \\
  -z \\
  z/d
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 & x \\
  0 & 1 & 0 & 0 & y \\
  0 & 0 & -1 & 0 & z \\
  0 & 0 & 1/d & 0 & 1
\end{bmatrix}
\]

divide by homogenous ordinate to map back to 3D space
Perspective Projections Details

Flip $z$ to transform to a left handed co-ordinate system $\Rightarrow$ increasing $z$ values mean increasing distance from the viewer.
Perspective Projection

- Depending on the application we can use different mechanisms to specify a perspective view.
- **Example:** the *field of view* angles may be derived if the distance to the viewing plane is known.
- **Example:** the viewing direction may be obtained if a point in the scene is identified that we wish to look at.
- OpenGL supports this by providing different methods of specifying the perspective view:
  - `gluLookAt`, `glFrustum` and `gluPerspective`
Perspective Projections

\[ \text{glFrustum}(x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, z_{\text{min}}, z_{\text{max}}); \]
glFrustum

- Note that all points on the line defined by \((x_{\text{min}}, y_{\text{min}}, -z_{\text{min}})\) and COP are mapped to the *lower left* point on the viewport.
- Also all points on the line defined by \((x_{\text{max}}, y_{\text{max}}, -z_{\text{min}})\) and COP are mapped to the upper right corner of the viewport.
- The viewing direction is always parallel to \(-z\).
- It is not necessary to have a symmetric frustrum like:

  \[
  \text{glFrustum}(-1.0, 1.0, -1.0, 1.0, 5.0, 50.0);
  \]

- Non symmetric frustrums introduce *obliqueness* into the projection.
- \(z_{\text{min}}\) and \(z_{\text{max}}\) are specified as positive distances along \(-z\).
Perspective Projections

\[ \text{gluPerspective}(\text{fov}, \text{aspect}, \text{near}, \text{far}); \]

\[
\begin{align*}
\text{aspect} &= \frac{w}{h} \\
\text{fov} &= \theta
\end{align*}
\]

\[
\frac{h}{2 \text{near}} = \tan \frac{\theta}{2} \Rightarrow h = 2 \text{near} \tan \frac{\theta}{2}
\]
**gluPerspective**

- A utility function to simplify the specification of perspective views.
- Only allows creation of symmetric frustrums.
- Viewpoint is at the origin and the viewing direction is the -z axis.
- The *field of view* angle, $\text{fov}$, must be in the range $[0..180]$.
- `aspect` allows the creation of a view frustum that matches the *aspect ratio* of the viewport to eliminate distortion.
Perspective Projections

- Typical
- Large aspect
- Large fov (or small near)
- Small fov (or large near)
Lens Configurations

10mm Lens (fov = 122°)  
20mm Lens (fov = 84°)

35mm Lens (fov = 54°)  
200mm Lens (fov = 10°)
Positioning the Camera

The previous projections had limitations:
- usually fixed origin and fixed projection direction

To obtain arbitrary camera orientations and positions we manipulate the MODELVIEW matrix prior to creation of the models. This positions the camera w.r.t. the model.

We wish to position the camera at (10, 2, 10) w.r.t. the world

Two possibilities:
- transform the world prior to creation of objects using translatef and rotatef: `glTranslatef(-10, -2, -10);`
- use `gluLookAt` to position the camera with respect to the world co-ordinate system: `gluLookAt(10, 2, 10, ... );`

Both are equivalent.
Positioning the Camera

\texttt{gluLookAt(eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz);} 

\texttt{equivalent to:}

\texttt{glTranslatef(-eyex, -eyey, -eyez);} 
\texttt{glRotatef(theta, 1.0, 0.0, 0.0);} 
\texttt{glRotatef(phi, 0.0, 1.0, 0.0);}
Projection window

- The projection matrix defines the mapping from a 3D world co-ordinate to a 2D viewport co-ordinate.
- The window extents are defined as a parameter of the projection:
  - `glFrustum(l,r,b,t,n,f)`
  - `gluPerspective(f,a,n,f) ⇒ h = n · tan(f/2)`
    
    \[
    w = h · a
    \]
Projection window

- We need to associate the 2D *window co-ordinate system* with the *viewport co-ordinate system* in order to determine the correct pixel associated with each vertex.

![Projection window diagram]
Window to Viewport

Transformation: review

- An affine planar transformation is used.
- After projection to the window, all points are transformed to normalised device co-ordinates: \([-1, 1] \times [1, 1]\)

\[
x_n = 2 \left( \frac{x_p - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \right) - 1
\]

\[
y_n = 2 \left( \frac{y_p - y_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} \right) - 1
\]

- `glViewport` used to relate the co-ordinate systems:

  `glViewport(int x, int y, int width, int height);`
Window to Viewport Transformation: review

- \((x, y) = \text{location of bottom left of viewport within the window}\)
- \(\text{width, height} = \text{dimension in pixels of the viewport} \Rightarrow \)
  \[
  x_w = (n_\text{w} + 1) \left( \frac{\text{width}}{2} \right) + x \quad y_w = (n_\text{h} + 1) \left( \frac{\text{height}}{2} \right) + y
  \]
- Normally we re-create the window after a window resize event to ensure a correct mapping between window and viewport dimensions:

```java
static void reshape(int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(85.0, 1.0, 5, 50);
}
```
Aspect Ratio

- The aspect ratio defines the relationship between the width and height of an image.
- Using `gluPerspective` an viewport aspect ratio may be explicitly provided, otherwise the aspect ratio is a function of the supplied viewport width and height.
- The aspect ratio of the window (defined by the user) must match the viewport aspect ratio to prevent unwanted affine distortion:

  \[
  \text{aspect ratio} = 1.25 \\
  \text{aspect ratio} = 0.5
  \]
// top left: top view
glViewport(0, win_height/2, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-3.0, 3.0, -3.0, 3.0, 1.0, 50.0);
gluLookAt(0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glCallList(object);

// top right: right view
glViewport(win_width/2, win_height/2, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-3.0, 3.0, -3.0, 3.0, 1.0, 50.0);
gluLookAt(5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glCallList(object);

// bottom left: front view
glViewport(0, 0, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-3.0, 3.0, -3.0, 3.0, 1.0, 50.0);
gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glCallList(object);

// bottom right: rotating perspective view
glViewport(win_width/2, 0, win_width/2, win_height/2);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(70.0, 1.0, 1, 50);
gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(30.0, 1.0, 0.0, 0.0);
glRotatef(Angle, 0.0, 1.0, 0.0);
glCallList(object);