CLASSICAL VIEWING

Objectives

- Introduce the classical views
- Compare and contrast image formation by computer with how images have been formed by architects, artists, and engineers
- Learn the benefits and drawbacks of each type of view

Introduction

A painting [the projection plane] is the intersection of a visual pyramid [view volume] at a given distance, with a fixed center [center of projection] and a defined position of light, represented by art with lines and colors on a given surface [the rendering]. (Alberti, 1435)

Classical Viewing

- Viewing requires three basic elements
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface
- Classical views are based on the relationship among these elements
  - The viewer picks up the object and orients it how she would like to see it
  - Each object is assumed to constructed from flat principal faces
    - Buildings, polyhedra, manufactured objects

Planar Geometric Projections

- Standard projections project onto a plane
- Projectors are lines that either
  - converge at a center of projection
  - are parallel
- Such projections preserve lines
  - but not necessarily angles
- Nonplanar projections are needed for applications such as map construction
Perspective vs Parallel

- Classical viewing developed different techniques for drawing each type of projection
- For graphics all projections done same way using a single pipeline
- Fundamental distinction between parallel and perspective viewing
  - even though mathematically parallel viewing is the limit of perspective viewing

Taxonomy of Planar Geometric Projections

- planar geometric projections
  - parallel
  - perspective
  - multiview
    - orthographic
  - axonometric
  - oblique
    - 1 point
    - 2 point
    - 3 point
  - isometric
  - dimetric
  - trimetric

Classification of Planar Projections

- many historical names...
- same math in the end 😊

Parallel Projections

- cop at infinity
- projectors are parallel
- no foreshortening
- distant objects do not look smaller
- parallel lines remain parallel
- parallel projection subtypes
  - depending on angles between vpn and dop
  - depending on dop and coordinate axes

Orthographic Projection

- Projectors are orthogonal to projection surface
- Special Cases:
  - top/side/front projections: view plane is perpendicular to one of the X,Y,Z axes
  - dop is identical to one of the X,Y,Z axes
  - angles are maintained used in architecture & civil engineering

Multiview Orthographic Projection

- Projection plane parallel to principal face
- Usually form front, top, side views
  - isometric (not multiview orthogonal view)
  - CAD & architecture often display 3 views plus isometric
Advantages and Disadvantages

- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
  - Building plans
  - Manuals
- Cannot see what object really looks like because many surfaces hidden from view
  - Often we add the isometric

Axonometric Projections

Allow projection plane to move relative to object

classify by how many angles of a corner of a projected cube are the same

none: trimetric
two: dimetric
three: isometric

Types of Axonometric Projections

- Lines are scaled (foreshortened) but can find scaling factors
- Lines preserved but angles are not
- Projection of a circle in a plane not parallel to the projection plane is an ellipse
- Some optical illusions possible
- Parallel lines appear to diverge
- Does not look real because far objects are scaled the same as near objects
- Used in CAD applications

Oblique Projection

Arbitrary relationship between projectors and projection plane

Oblique Parallel Projections Subtypes

- Cavalier projection: 45° angle between dop and vpn
- Cabinet projection: 63.4° (arctan(2)) angle between dop and vpn; z-lengths shorter by ½
- these are not perspective projections!

Advantages and Disadvantages

- Can pick the angles to emphasize a particular face
  - Architecture: plan oblique, elevation oblique
- Angles in faces parallel to projection plane are preserved while we can still see “around” side
- In physical world, cannot create with simple camera; possible with bellows camera or special lens (architectural)
**Perspective Projection**

- Projectors converge at center of projection (camera)
- COP not at infinity
- Parallel lines are not preserved
  - The meet at vanishing points
    - Drawing simple perspectives by hand uses these vanishing point(s)

**Three-Point Perspective**

- No principal face parallel to projection plane
- Three vanishing points for cube

**Two-Point Perspective**

- On principal direction parallel to projection plane
- Two vanishing points for cube

**One-Point Perspective**

- One principal face parallel to projection plane
- One vanishing point for cube

**Advantages and Disadvantages**

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*nonuniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

**Perspective Projection: Vanishing Points**

- how many vanishing points maximum?
Perspective Projection: Vanishing Points

- Consider a simple object: cube with three sets of parallel lines
- Consider the view plane position w.r.t. X, Y, Z axes

Perspective Projections

- For our cube example:
  - Three vanishing points if view plane intersects all coordinate axes
  - Two if only two axes are intersected
  - One if only one axis is intersected

- For a general object
  - As many vanishing points as sets of parallel lines (for a natural scene, infinitely many)

Objectives

- Introduce the mathematics of projection
- Introduce viewing in OpenGL
- Look at alternate viewing APIs

Model-View Transformation

- Input:
  - Object definitions (including lights, etc.)
  - Object transformations
  - Camera position & parameters
- Problem:
  - How to arrange objects in space, where we want them to be?
Model-View Transformation

1st idea: transform all objects and camera into one world coordinate system

Model-View Transformation

2nd idea: transform objects directly into camera (also called eye) coordinates

combine model and view transformation in a single model-view transformation

Perspective Projections: Camera Model

- mimics real cameras
- real camera parameters
  - position, orientation
  - aperture
  - shutter time
  - focal length
  - type of lens (zoom vs. wide)
  - depth of field
  - size of resulting image
  - aspect ratio (4:3, 16:9, 1.85:1, 2.35:1, 2.39:1)
  - resolution (digital cameras)

Perspective Projections: Camera Model

- simplified camera model in CG
  - position: 3D point (= cop)
  - view direction: 3D vector (= vpn)
  - image area on view plane: viewport
  - clipping planes for removing near and far objects

The OpenGL Camera

- In OpenGL, initially the object and camera frames are the same
  - Default model-view matrix is an identity
  - The camera is located at origin and points in the negative z direction
  - OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity

Default Projection

Default projection is orthogonal
Moving the Camera Frame

- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction
  - Translate the camera frame
  - Move the objects in the negative z direction
  - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - Want a translation \( \text{Translate}(0.0, 0.0, -d) \)
  - \( d > 0 \)

Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - Rotate the camera
  - Move it away from origin
  - Model-view matrix \( C = TR \)

OpenGL code

- Remember that last transformation specified is first to be applied

```cpp
// Using mat.h
// gets passed to shader
mat4 t = Translate(0.0, 0.0, -d);
mat4 ry = RotateY(90.0);
mat4 m = t * ry;
```

Other Viewing APIs

- The LookAt function is only one possible API for positioning the camera
- Others include
  - View reference point, view plane normal, view up (PHIGS, GKS-3D)
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles
The default projection in the eye (camera) frame is orthogonal.

Center of projection at the origin

d = focal distance

view plane placed into XY

Find projected point (x', y')

Given a point (x, y, z) to be projected

All other views are converted to the default view by transformations

determine the projection matrix

Allows use of the same pipeline for all views

Most graphics systems use projectors are parallel to Z

Allows use of the same pipeline for all views

For points within the default view volume

d is focal distance

Homogeneous Coordinate Representation

default orthographic projection

Consider top and side views

Similar triangles

In practice, we can let \( M = I \) and set the \( z \) term to zero later

Perspective Projections: Math

• take the projection formulas...

\[
(x', y', z') = \left( \frac{x}{z/d}, \frac{y}{z/d}, \frac{z}{d} \right)
\]

• write them as matrix in homogeneous coordinates

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & \frac{1}{d}
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
x \\
y \\
z \\
\frac{1}{d}
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
x \\
y \\
z \\
\frac{1}{d}
\end{bmatrix}
\]

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

Parallel Projections: Math

• view plane placed into XY plane

• projectors are parallel to Z axis

• hence, z values project to 0

• parallel projection matrix is thus:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

NEVER set the distance of front clipping plane to camera position to zero!!!
Other Parallel Projections

- **oblique projection matrix**

\[
P = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
- \begin{bmatrix}
\frac{\cos(\alpha)}{\tan(\beta)} & 0 & 0 & 0 \\
\frac{\sin(\alpha)}{\tan(\beta)} & 1 & 0 & 0 \\
\frac{-\sin(\beta)}{\sin(\beta)} & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Projections Summary

- encode all projection types as matrices \(M\)
- given a camera specification (location, focal distance \(d\))
- 3D point \(p\) in the eye coordinate system
- compute the projection \(p' = M \cdot p\)
- obtain the final 2D coordinates \((x, y)\) of the projected point by dividing by the 4th element of the homogeneous vector \(p'\)
- can merge projection and modelview seamlessly (matrix multiplication)
- extra elements
  - clipping, hidden surface removal

OpenGL Orthogonal Viewing

\[
\text{Ortho}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far})
\]

- near and far measured from camera
- \(\text{glm::mat4 glm::ortho(} \text{float left, float right, float bottom, float top, float zNear, float zFar)\;}\)
- \(\text{glm::mat4 projection = glm::ortho(GLfloat(} 0.0, \text{ w, h, 0.0, 1.0, -1.0)\;)}\)

OpenGL Perspective Viewing

\[
\text{Frustum}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far})
\]

- \(\text{glm::mat4 glm::frustum(}\ \text{float left, float right, float bottom, float top, float zNear, float zFar)\;}\)

Using Field of View

- With \text{Frustum} it is often difficult to get the desired view
- \text{Perspective(fovy, aspect, near, far)} often provides a better interface
- \(\text{glm::mat4 perspective(} \text{float fovy, float aspect, float zNear, float zFar)\;}\)
```
Example (Similar to rotating Cube)

// generate 12 tris 36 verts & 36 colors
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```
Display function
void display( void ) {
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity();
    point4  eye( radius*sin(theta)*cos(phi),
                radius*sin(theta)*sin(phi),
                radius*cos(theta),
                1.0 );
    point4  at( 0.0, 0.0, 0.0, 1.0 );
    vec4    up( 0.0, 1.0, 0.0, 0.0 );
    mat4  mv = LookAt( eye, at, up );
    glUniformMatrix4fv( model_view, 1, GL_TRUE, mv );
    mat4  p = Ortho( left, right, bottom, top, 
zNear, zFar );
    glUniformMatrix4fv( projection, 1, GL_TRUE, p );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}

Keyboard func – camera interaction
void keyboard( unsigned char key, int x, int y ) {
    switch( key ) {
        case 033: // Escape Key
            case 'q': case 'Q': exit( EXIT_SUCCESS );
            break;
        case 'x': left *= 1.1; right *= 1.1; break;
        case 'X': left *= 0.9; right *= 0.9; break;
        case 'y': bottom *= 1.1; top *= 1.1; break;
        case 'Y': bottom *= 0.9; top *= 0.9; break;
        case 'z': zNear *= 1.1; zFar *= 1.1; break;
        case 'Z': zNear *= 0.9; zFar *= 0.9; break;
        case 'r': radius *= 2.0; break;
        case 'R': radius *= 0.5; break;
        case 'o': theta += dr; break;
        case 'O': theta -= dr; break;
        case 'p': phi += dr; break;
        case 'P': phi -= dr; break;
        case ' ':  // reset values to their defaults
            left = -1.0; right = 1.0;
            bottom = -1.0; top = 1.0;
            zNear = 0.5; zFar = 3.0;
            radius = 1.0; theta = 0.0; phi = 0.0;
            break;
    }
    glutPostRedisplay();
}

Reshape function
void reshape( int width, int height ) {
    glViewport( 0, 0, width, height );
}

Vertex Shader
#version 150
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform mat4 model_view;
uniform mat4 projection;
void main() {
    gl_Position = projection*model_view*vPosition/vPosition.w;
    color = vColor;
}

Pixel Shader
#version 150
in vec4 color;
out vec4 fColor;
void main() {
    fColor = color;
}

The reshape function is a really good place to put this, unless want user interaction as in this program.
Objectives

- Derive the projection matrices used for standard OpenGL projections
- Introduce oblique projections
- Introduce projection normalization

Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

Notes

- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
- Both these transformations are nonsingular
- Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until end
  - Important for hidden-surface removal to retain depth information as long as possible

Pipeline View

Normalization

\[ \text{Ortho}(left, right, bottom, top, near, far) \]
Orthogonal Matrix

- Two steps
  - Move center to origin
    \[ T\left(\frac{-\text{left}+\text{right}}{2}, \frac{-\text{top}+\text{bottom}}{2}, \frac{\text{near}+\text{far}}{2}\right) \]
  - Scale to have sides of length 2
    \[ S\left(\frac{2}{\text{left}-\text{right}}, \frac{2}{\text{top}-\text{bottom}}, \frac{2}{\text{near}-\text{far}}\right) \]

\[ P = ST = \begin{bmatrix} 2 & 0 & 0 & -\frac{\text{right}-\text{left}}{\text{top}-\text{bottom}} \ 0 & 2 & 0 & -\frac{\text{top}-\text{bottom}}{\text{top}-\text{bottom}} \ 0 & 0 & 2 & -\frac{\text{near}-\text{far}}{\text{far}-\text{near}} \ 0 & 0 & 0 & 1 \end{bmatrix} \]

Final Projection

- Set \( z = 0 \)
- Equivalent to the homogeneous coordinate transformation

\[ M_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- Hence, general orthogonal projection in 4D is

\[ P = M_{\text{orth}} S T \]

Oblique Projections

- The OpenGL projection functions cannot produce general parallel projections such as

- However if we look at the example of the cube it appears that the cube has been sheared
- Oblique Projection = Shear + Orthogonal Projection

Shear Matrix

\[ \begin{bmatrix} 1 & 0 & -\cot \theta & 0 \\ 0 & 1 & -\cot \phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

General case:

\[ P = M_{\text{orth}} H(\theta, \phi) \]

Equivalency
Effect on Clipping

- The projection matrix $P = STH$ transforms the original clipping volume to the default clipping volume.

![Diagram showing the effect of projection matrix on clipping volume]

Perspective Matrices

Simple projection matrix in homogeneous coordinates:

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

Note that this matrix is independent of the far clipping plane.

Generalization

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

After perspective division, the point $(x, y, z, 1)$ goes to

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

which projects orthogonally to the desired point regardless of $\alpha$ and $\beta$.

Picking $\alpha$ and $\beta$

If we pick

$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

and

$$\beta = \frac{2\text{near} \cdot \text{far}}{\text{near} - \text{far}}$$

the near plane is mapped to $z = -1$

the far plane is mapped to $z = 1$

and the sides are mapped to $x = \pm 1, y = \pm 1$

Hence the new clipping volume is the default clipping volume.

Normalization Transformation

- The COP at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes $x = \pm z, y = \pm z$.

- After perspective division, the point $(x, y, z, 1)$ goes to $x'' = x/z, y'' = y/z, z'' = -(\alpha + \beta)/z$ which projects orthogonally to the desired point regardless of $\alpha$ and $\beta$.
Normalization and Hidden-Surface Removal

- Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if \( z_1 > z_2 \) in the original clipping volume then for the transformed points \( z_1' > z_2' \).
- Thus hidden surface removal works if we first apply the normalization transformation.
- However, the formula \( z'' = -\frac{\alpha + \beta}{z} \) implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small.

OpenGL Perspective

- \texttt{glFrustum} allows for an unsymmetric viewing frustum (although \texttt{Perspective} does not).

OpenGL Perspective Matrix

- The normalization in \texttt{Frustum} requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthogonal transformation.

\[
P = \text{NSH}
\]

our previously defined perspective matrix  
shear and scale

Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing.
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading.
- We simplify clipping.