Perspective Projection

**CPSC 314**

The Rendering Pipeline

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

- Scan Conversion
- Texturing
- Depth Test
- Blending
- Frame-buffer

Geometry Processing

Rasterization

Fragment Processing

© Wolfgang Heidrich
Homogeneous Coordinates

**Homogeneous representation of points:**
- Add an additional component \( w=1 \) to all points
- All multiples of this vector are considered to represent the same 3D point
- All points are represented as column vectors

\[
\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix}, \quad \forall w \neq 0
\]

Homogeneous Matrices

**Affine Transformations**

\[
\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & 0 \\ m_{2,1} & m_{2,2} & m_{2,3} & 0 \\ m_{3,1} & m_{3,2} & m_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \\ 0 \end{bmatrix}
\]

\[
= \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & 0 \\ m_{2,1} & m_{2,2} & m_{2,3} & 0 \\ m_{3,1} & m_{3,2} & m_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & t_x \\ 0 & 0 & 0 & t_y \\ 0 & 0 & 0 & t_z \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}
\]
Homogeneous Vectors

Representing vectors in homogeneous coordinates

- Column vectors with $w=0$

$$T \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & t_x \\ m_{2,1} & m_{2,2} & m_{2,3} & t_y \\ m_{3,1} & m_{3,2} & m_{3,3} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

Homogeneous Planes And Normals

Planes in homogeneous coordinates are represented as row vectors

- $E = [n_x, n_y, n_z, d]$
- $[n_x, n_y, n_z, 0]^T$ is normal of plane with $n_x^2 + n_y^2 + n_z^2 = 1$
- Normal points towards side of plane on which origin is located

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \in E = [n_x, n_y, n_z, d] \iff [n_x, n_y, n_z, d] \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = 0$$
### Homogeneous Planes and Normals

#### Example in 2D (lines instead of planes):

- **Line** $L$: $y = 1 - x$
- Implicit definition: $-x - y + 1 = 0$
- Unit-length normal of that line:
  $$\mathbf{n} = \left[\frac{-\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}, 0\right]^T$$
- Distance of line from origin:
  $$d = \frac{\sqrt{2}}{2}$$
- Thus:
  $$L = \left[\frac{-\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right] = [-1, -1, 1]$$

---

### The Rendering Pipeline

The rendering pipeline consists of several stages:

1. **Geometry Database**
2. **Model/View Transform.**
3. **Lighting**
4. **Perspective Transform.**
5. **Clipping**
6. **Scan Conversion**
7. **Texturing**
8. **Depth Test**
9. **Blending**
10. **Frame-buffer**

Each stage processes the data in a specific way, leading to the final rendered image.
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

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

Result
- 2D screen coordinates of clipped vertices

Perspective Transformation

*Pinhole Camera:*

- Light shining through a tiny hole into a dark room yields upside-down image on wall
Perspective Transformation

Pinhole Camera

Real Cameras

- pinhole camera has small aperture (lens opening)
  - hard to get enough light to expose the film
- lens permits larger apertures
- lens permits changing distance to film plane without actually moving the film plane

price to pay: limited depth of field
Real Cameras - Depth of Field

**Limited depth of field**
- Can be used to direct attention
- Artistic purposes

Perspective Transformation

*In computer graphics:*
- Image plane is conceptually *in front* of the center of projection
- Perspective transformations belong to a class of operations that are called *projective transformations*
- Linear and affine transformations also belong to this class
- *All* projective transformations can be expressed as $4 \times 4$ matrix operations
**Perspective Projection**

**Synopsis:**
- Project all geometry through a common *center of projection* (eye point) onto an *image plane*

![Diagram of perspective projection]

**Example:**
- Assume image plane at \( z = -1 \)
- A point \([x, y, z, I]^T\) projects to \([-x/z, -y/z, -z/z, I]^T = [x, y, z, -z]^T\)
Perspective Projection

Analysis:
- This is a special case of a general family of transformations called *projective transformations*.
- These can be expressed as 4x4 homogeneous matrices!
  - *E.g. in the example:*

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

Projective Transformations

Transformation of space:
- Center of projection moves to infinity.
- Viewing frustum is transformed into a parallelepiped.
Projective Transformations

**Convention:**
- Viewing frustum is mapped to a specific parallelepiped
  - Normalized Device Coordinates (NDC)
- Only objects inside the parallelepiped get rendered
- Which parallelepiped is used depends on the rendering system

**OpenGL:**
- Left and right image boundary are mapped to $x=-1$ and $x=+1$
- Top and bottom are mapped to $y=-1$ and $y=+1$
- Near and far plane are mapped to 0 and 1

---

Projective Transformations

**OpenGL Convention**

Camera coordinates

- Frustum

NDC

- $x = 1$
- $x = -1$
- $z = 1$
- $z = -1$
Projective Transformations

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
Projective Transformations

Alternative specification of symmetric frusta

- Field-of-view (fov) $\alpha$
- Fov/2
- Field-of-view in y-direction (fovy) + aspect ratio

Demos

Tuebingen applets from Frank Hanisch

- http://www.gris.uni-tuebingen.de/projects/grdev/doc/html/etc/AppletIndex.html#Transformationen
Projective Transformations

Properties:
- All transformations that can be expressed as homogeneous 4x4 matrices (in 3D)
- 16 matrix entries, but multiples of the same matrix all describe the same transformation
  - 15 degrees of freedom
  - The mapping of 5 points uniquely determines the transformation

Determining the matrix representation
- Need to observe 5 points in general position, e.g.
  - \([\text{left}, 0, 0, 1]^T \rightarrow [1, 0, 0, 1]^T\)
  - \([0, \text{top}, 0, 1]^T \rightarrow [0, 1, 0, 1]^T\)
  - \([0, 0, -f, 1]^T \rightarrow [0, 0, 1, 1]^T\)
  - \([0, 0, -n, 1]^T \rightarrow [0, 0, 0, 1]^T\)
  - \([\text{left}*f/n, \text{top}*f/n, -f, 1]^T \rightarrow [1, 1, 1, 1]^T\)
- Solve resulting equation system to obtain matrix
### Perspective Derivation

#### Derivation for the Perspective Transformations

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

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

\[
y' = Fy + Bz, \quad \frac{y'}{w'} = \frac{Fy + Bz}{w'}, \quad 1 = \frac{Fy + Bz}{w'}, \quad 1 = \frac{Fy + Bz}{-z},
\]

\[
1 = \frac{y}{-z} + B \frac{z}{-z}, \quad 1 = \frac{y}{-z} - B, \quad 1 = \frac{\text{top}}{-(-\text{near})} - B,
\]

\[
1 = \frac{\text{top}}{\text{near}} - B
\]

### Similarly for Other 5 Planes

**6 Planes, 6 Unknowns**

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

© Wolfgang Heidrich
**Perspective Example**

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

*view volume*

- left = -1, right = 1
- bot = -1, top = 1
- near = 1, far = 4

---

**Projective Transformations**

**Properties**

- Lines are 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)*
Orthographic Camera Projection

- Camera’s back plane parallel to lens
- Infinite focal length
- No perspective convergence
- Just throw away z values

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

Projection Taxonomy

- Planar projections
  - Perspective: 1,2,3-point
  - Parallel
  - Oblique
  - Cabinet
  - Cavalier
  - Orthographic
  - Axonometric: isometric, dimetric, trimetric

http://ceprofs.tamu.edu/kraker/ENGR%20111/5.1/20

© Wolfgang Heidrich
Perspective Projections
classified by vanishing points

- one-point perspective
- two-point perspective
- three-point perspective

Axonometric Projections

- projectors perpendicular to image plane

- A. Isometric
- B. Dimetric
- C. Trimetric

http://ceprofs.tamu.edu/tkramer/ENGR%20111/5.1.20 © Wolfgang Heidrich
**View Volumes**

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

**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
**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

**Window-To-Viewport Transformation**

*Generate pixel coordinates*

- Map \( x, y \) from range \(-1...1\) (normalized device coordinates) to pixel coordinates on the screen
- Involves 2D scaling and translation
OpenGL Transformations, Hierarchical Transformations, Accelerations

CPSC 314

The Rendering Pipeline
Rendering Geometry in OpenGL

```c
glBegin( GL_TRIANGLES );
    glVertex3f( x1, y1, z1 ); // vertex 1 of triangle 1
    glVertex3f( x2, y2, z2 ); // vertex 2 of triangle 1
    glVertex3f( x3, y3, z3 ); // vertex 3 of triangle 1
    glVertex3f( x4, y4, z4 ); // vertex 1 of triangle 2
    glVertex3f( x5, y5, z5 ); // vertex 2 of triangle 2
    glVertex3f( x6, y6, z6 ); // vertex 3 of triangle 2
...

glEnd();
```

Additional attributes

- `glColor3f`: RGB color value (0...1 per component)
- `glNormal3f`: normal vector
- `glTexCoord2f`: texture coordinate (explained later)

OpenGL is state machine:

- Every vertex gets color, normal etc. that corresponds to last specified value

---

© Wolfgang Heidrich
Rendering Geometry in OpenGL

Example:

```c
glBegin( GL_TRIANGLES );
    glColor3f( 1.0, 0.0, 0.0 );
    glVertex3f( 1.0, 0.0, 0.0 );
    glColor3f( 0.0, 0.0, 1.0 );
    glVertex3f( 0.0, 0.0, 0.0 );
    glVertex3f(1.0, 0.0, 0.0 );
    glEnd();
```

OpenGL Naming Scheme

Function names:

- **OpenGL Prefix**: gl
- **Operation**: Vertex 3f
- **Dimensionality**: 1-4
- **Type of parameters**:
  - e.g. f (float)
  - d (double)
  - i (integer)

- Missing coordinates are 0 (x,y,z) or 1 (w)
Matrix Operations in OpenGL

2 Matrices:
- Model/view matrix \( M \)
- Projective matrix \( P \)

Example:
```c
glMatrixMode( GL_MODELVIEW );
glLoadIdentity(); // M=Id
glRotatef( angle, x, y, z ); // M=Id*R(\( \alpha \))
glTranslatef( x, y, z ); // M= Id*R(\( \alpha \))*T(x,y,z)
glMatrixMode( GL_PROJECTION );
glRotatef( ... ); // P= ...
```

Matrix Operations in OpenGL

Semantics:
- \( \text{glMatrixMode} \) sets the matrix that is to be affected by all following transformations (multiplication from the right)
- Transformations that affect a vertex \textit{first} have to be specified \textit{last}
- Whenever primitives are rendered with \( \text{glBegin}() \), the vertices are transformed with whatever the current model/view and perspective matrix is
  - \textit{Normals are transformed with the inverse transpose}
Matrix Operations in OpenGL

Specifying matrices (replacement)
- `glLoadIdentity()`
- `glLoadMatrixf( GLfloat *m ) // 16 floats`

Specifying matrices (multiplication)
- `glMultMatrixf( GLfloat *m ) // 16 floats`
- `glRotatef( GLfloat angle, GLfloat x, GLfloat y, GLfloat z ) // angle and axis`
- `glScalef( GLfloat x, GLfloat y, GLfloat z )`
- `glTranslatef( GLfloat x, GLfloat y, GLfloat z )`

More Complicated Matrices:
- `glFrustum( left, right, bottom, top, near, far )`
  - Specifies perspective xform (near, far are always positive)
- `glOrtho( left, right, bottom, top, near, far )`

Convenience Functions:
- `gluPerspective( fovy, aspect, near, far )`
  - Another way to do perspective
- `gluLookAt( eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ )`
  - Useful for viewing transform
Transformation Hierarchies

Scene may have a hierarchy of coordinate systems

- Stores matrix at each level with incremental transform from parent’s coordinate system

Scene graph

Transformation Hierarchy Example 1

trans(0.30,0,0) rot(z,\theta)
Transformation Hierarchies

- Hierarchies don’t fall apart when changed
- transforms apply to graph nodes beneath

Brown Applets

http://www.cs.brown.edu/exploratories/freeSoftware/catalogs/scenegraphs.html

- Have a look later
Transformation Hierarchy
Example 2

- Draw same 3D data with different transformations: instancing

Matrix Stacks

**Challenge of avoiding unnecessary computation**

- Using inverse to return to origin
- Computing incremental $T_1 \rightarrow T_2$
Matrix Stacks

- `glPushMatrix()`
- `glPopMatrix()`
- `D = C \text{ scale}(2,2,2) \text{ trans}(1,0,0)`
- `DrawSquare()`
- `glPushMatrix()`
- `glScalef(2,2,2)`
- `glTranslatef(1,0,0)`
- `DrawSquare()`
- `glPopMatrix()`

Modularization

**Drawing a scaled square**

- Push/pop ensures no coord system change

```c
void drawBlock(float k) {
    glPushMatrix();
    glScalef(k,k,k);
    glBegin(GL_LINE_LOOP);
    glVertex3f(0,0,0);
    glVertex3f(1,0,0);
    glVertex3f(1,1,0);
    glVertex3f(0,1,0);
    glEnd();
    glPopMatrix();
}
```
Matrix Stacks

**Advantages**
- No need to compute inverse matrices all the time
- Modularize changes to pipeline state
- Avoids incremental changes to coordinate systems
  - Accumulation of numerical errors

**Practical issues**
- In graphics hardware, depth of matrix stacks is limited
  - (typically 16 for model/view and about 4 for projective matrix)

Transformation Hierarchy

**Example 3**

```c
glLoadIdentity();
glTranslatef(4,1,0);
glPushMatrix();
glRotatef(45,0,0,1);
glTranslatef(0,2,0);
glScalef(2,1,1);
glTranslatef(1,0,0);
glPopMatrix();
```
Transformation Hierarchy
Example 4

Hierarchical Modeling

Advantages
- Define object once, instantiate multiple copies
- Transformation parameters often good control knobs
- Maintain structural constraints if well-designed

Limitations
- Expressivity: not always the best controls
- Can't do closed kinematic chains
  - *Keep hand on hip*
Single Parameter: simple

Parameters as functions of other params

- Clock: control all hands with seconds $s$

$$m = s/60, \ h = m/60,$$
$$\theta_s = (2 \pi \ s) / 60,$$
$$\theta_m = (2 \pi \ m) / 60,$$
$$\theta_h = (2 \pi \ h) / 60$$

Single Parameter: complex

Mechanisms not easily expressible with affine transforms

http://www.flying-pig.co.uk
Display Lists

Concept:

- If multiple copies of an object are required, it can be compiled into a display list:

  ```
  glNewList( listId, GL_COMPILE );
  glBegin( ... );
  ... // geometry goes here
  glEndList();
  // render two copies of geometry offset by 1 in z-direction:
  glCallList( listId );
  glTranslatef( 0.0, 0.0, 1.0 );
  glCallList( listId );
  ```

Advantages:

- More efficient than individual function calls for every vertex/attribute
- Can be cached on the graphics board (bandwidth!)
- Display lists exist across multiple frames
  - Represent static objects in an interactive application
**Shared Vertices**

**Triangle Meshes**

- Multiple triangles share vertices
- If individual triangles are sent to graphics board, every vertex is sent and transformed multiple times!
  - Computational expense
  - Bandwidth

![Triangle Meshes Diagram](image)

**Triangle Strips**

**Idea:**

- Encode neighboring triangles that share vertices
- Use an encoding that requires only a constant-sized part of the whole geometry to determine a single triangle
- N triangles need n+2 vertices

![Triangle Strips Diagram](image)
Triangle Strips

**Orientation:**
- Strip starts with a counter-clockwise triangle
- Then alternates between clockwise and counter-clockwise

Triangle Fans

**Similar concept:**
- All triangles share one center vertex
- All other vertices are specified in CCW order
Triangle Strips and Fans

Transformations:
• n+2 for n triangles
• Only requires 3 vertices to be stored according to simple access scheme
• Ideal for pipeline (local knowledge)

Generation
• E.g. from directed edge data structure
• Optimize for longest strips/fans

Vertex Arrays

Concept:
• Store array of vertex data for meshes with arbitrary connectivity (topology)

```c
GLfloat *points[3*nvertices];
GLfloat *colors[3*nvertices];
Glint *tris[numtris]=
  {0,1,3, 3,2,4, ...};
glVertexPointer( ..., points );
glColorPointer( ...,colors );
glDrawElements( 
  GL_TRIANGLES, ...,tris );
```
Vertex Arrays

**Benefits:**
- Ideally, vertex array fits into memory on graphics chip
- Then all vertices are transformed exactly once

**In practice:**
- Graphics memory may not be sufficient to hold model
- Then either:
  - Cache only parts of the vertex array on board (may lead to cache trashing!)
  - Transform everything in software and just send results for individual triangles (bandwidth problem: multiple transfers of same vertex!)

---

The Rendering Pipeline

[Diagram of the rendering pipeline showing steps from geometry database to frame-buffer including stages like Model/View Transform, Lighting, Perspective Transform, Clipping, Scan Conversion, Texturing, Depth Test, Blending, and fragment processing.]
Coming Up...

**Wednesday, May 17:**
- Shading and Lighting

**Friday, May 19:**
- Clipping
- Quiz 1!