**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

\[
\begin{pmatrix}
    x \\
    y \\
    z \\
    1
\end{pmatrix}
= \begin{pmatrix}
    m_{11} & m_{12} & m_{13} & t_x \\
    m_{21} & m_{22} & m_{23} & t_y \\
    m_{31} & m_{32} & m_{33} & t_z \\
    0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    w
\end{pmatrix}, \quad \forall w \neq 0
\]

**Homogeneous Vectors**

**Representing vectors in homogeneous coordinates**
- Column vectors with \( w = 0 \)

\[
\begin{pmatrix}
    x \\
    y \\
    z \\
    0
\end{pmatrix}
= \begin{pmatrix}
    m_{11} & m_{12} & m_{13} & t_x \\
    m_{21} & m_{22} & m_{23} & t_y \\
    m_{31} & m_{32} & m_{33} & t_z \\
    0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    0
\end{pmatrix}
\]

---

**The Rendering Pipeline**

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

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

**The Rendering Pipeline**

**Perspective Projection**

**Homogeneous Matrices**

**Affine Transformations**

\[
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    1
\end{pmatrix}
= \begin{pmatrix}
    m_{11} & m_{12} & m_{13} & 0 \\
    m_{21} & m_{22} & m_{23} & 0 \\
    m_{31} & m_{32} & m_{33} & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
    x \\
    y \\
    z \\
    1
\end{pmatrix}
\]

**Example:**

```c
glBegin( GL_TRIANGLES );
gColor3f( 1.0, 0.0, 0.0 );
gVertex3f( 1.0, 0.0, 0.0 );
gColor3f( 0.0, 0.0, 1.0 );
gVertex3f( 1.0, 0.0, 0.0 );
gColor3f(0.0, 0.0, 0.0 );
gVertex3f(0.0, 0.0, 0.0 );
gEnd();
```
Matrix Operations in OpenGL

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

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

Interpreting Composite Transformations

**Interpretation 1: moving the coordinate system**
- Read operations in forward order
  - `glTranslatef(4.0, 0.0, 0.0)`
  - `glRotatef(30.0);`  
  - `glTranslatef(-4.0, -3.0);`

**Interpretation 2: moving the object**
- Read operations in reverse order
  - `glTranslatef(-4.0, -3.0, 0.0)`
  - `glRotatef(30.0);`
  - `glTranslatef(4.0, 0.0, 0.0)`

Matrix Stacks

- `glPushMatrix()`
- `glPopMatrix()`

Transformation Hierarchy

Example 4

- `glTranslatef(x, y, 0);`
- `glRotatef(θ_x, θ_y, θ_z);`
- `DrawBody();`
- `glPushMatrix();`
- `glTranslatef(0.0, 0.0, 0.0);`
- `DrawHead();`
- `glPushMatrix();`
- `glTranslate(θ_x, θ_y, θ_z);`
- `glRotatef(θ_x, θ_y, θ_z);`
- `DrawLArm();`
- `glPopMatrix();`
- `glPushMatrix();`
- `glTranslate(θ_x, θ_y, θ_z);`
- `glRotatef(θ_x, θ_y, θ_z);`
- `DrawRArm();`
- `glPopMatrix();`
- `glPushMatrix();`
- `glTranslate(θ_x, θ_y, θ_z);`
- `glRotatef(θ_x, θ_y, θ_z);`
- `DrawOtherArm();`
- `glPopMatrix();`

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);`
**Display Lists**

**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 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**

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

---

**Triangle Fans**

**Similar concept:**
- All triangles share on 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)
  - GLfloat *points[3*vertices];
  - GLfloat *colors[3*vertices];
  - GLint *tris[nuntris];
  - {0,1, 3,2,4, ...};
  - glVertexPointer(..., points);
  - glColorPointer(..., colors);
  - glDrawArrays(GL_TRIANGLES, ...);

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 thrashing!)
    - Transform everything in software and just send results for individual triangles (bandwidth problem: multiple transfers of same vertex)

The Rendering Pipeline

Projective Rendering Pipeline

Rendering Pipeline
**Rendering Pipeline**

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

**result**

- Scene vertices in 3D view
  - (camera) coordinate system

**Perspective Transformation**

**Pinhole Camera:**

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

**Real Cameras**

- Pinhole camera has small aperture (lens opening)
  - Hard to get enough light to expose the film

- Real pinhole camera

- 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 4x4 matrix operations.

**Perspective Projection**

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

**Example:**
- Assume image plane at z=-1.
- A point \([x',y',z',1]^{T}\) projects to \([x/z, y/z, -z, 1]^{T} = [x, y, z]^{T}\).

**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:
    \[
    \begin{pmatrix}
    x' \\ y' \\ z' \\ 1
    \end{pmatrix} = \begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & -1 & 1
    \end{pmatrix} \begin{pmatrix}
    x \\ y \\ z \\ 1
    \end{pmatrix} = \begin{pmatrix}
    x' \\ y' \\ z' \\ 1
    \end{pmatrix} = \begin{pmatrix}
    x \\ y \\ z \\ 1
    \end{pmatrix}
    \]

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

**Convention:**
- Viewing frustum is mapped to a specific parallelepiped.
  - Normalized Device Coordinates (NDC)
- Only objects inside the parallelepiped get rendered.
- Which parallelepied 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 -1 and 1.
**Projective Transformations**

**OpenGL Convention**

Camera coordinates vs. NDC

- Frustum representation
- 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

**Asymmetric Viewing Frusta**

- Alternative specification of symmetric frusta
  - Field-of-view (fov) $\alpha$
  - $\text{Fov/2}$
  - Field-of-view in $y$-direction $f_{\text{y}} + \text{aspect ratio}$

**Demos**

- Tuebingen applets from Frank Hanisch
  - [Link](http://www.gts.uni-tuebingen.de/projects/gtsdev/doc/html/4_4/0_Applets.html#4_4/0_Homogeneous)
Projective Transformations

**Determining the matrix representation**
- Need to observe 5 points in general position, e.g.
  - [left,0,0,1]^T→[1,0,0,1]^T
  - [0, top, 0, 1]^T→[0,1,0,1]^T
  - [0,0,-1,1]^T→[0,0,1,1]^T
  - [left*fn, top*fn, -1, 1]^T→[1,1,1,1]^T
- Solve resulting equation system to obtain matrix

**Perspective Derivation**

\[
x' = \frac{Ex + Az}{w}, \quad y' = \frac{Ey + Bz}{w}, \quad z' = \frac{Cz + D}{w}
\]

\[
x = \text{left} \rightarrow x'/w' = 1, \quad y = \text{right} \rightarrow x'/w' = -1, \quad z = \text{bottom} \rightarrow y'/w' = 1, \quad z = \text{near} \rightarrow z'/w' = 1
\]

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

**Perspective Example**

**Orthographic Camera Projection**

- Camera's back plane parallel to lens
- Infinite focal length
- No perspective convergence

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

**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)
Projection Taxonomy

- planar projections
  - perspective: 1,2,3-point
  - parallel
  - oblique
  - cavalier
  - isometric
  - dimetric
  - trimetric
  - top, front, side

Perspective Projections

- classified by vanishing points
  - one-point perspective
  - two-point perspective
  - three-point perspective

Axonometric Projections

- projectors perpendicular to image plane
  - 3 Equal axes
  - 2 Equal axes
  - 0 Equal axes
  - 120°
  - 90°

View Volumes

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

View Volume

- 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

Perspective Matrices in OpenGL

- Perspective 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
**Projective Rendering Pipeline**

- **object OCS**
- **modeling transformation**
- **world WCS**
- **w2v viewing transformation**
- **w2v viewing transformation**
- **projection transformation**
- **v2c clipping transformation**
- **c2n perspective divide**
- **n2d normalized device NDCS**
- **viewport transformation**
- **device DCS**

**Window-To-Viewport Transformation**

**Generate pixel coordinates**
- Map $x, y$ from range $-1...1$ (normalized device coordinates) to pixel coordinates on the screen.
- Map $z$ from $-1...1$ to $0...1$ (used later for visibility).
- Involves 2D scaling and translation.

![Diagram of projective rendering pipeline and window-to-viewport transformation](image-url)