# Transforming Normals

- nonuniform scaling does not work
- \( x \times y = 0 \) plane
  - line \( x = y \)
  - normal: \([1, -1, 0]\)
  - direction of line \( x = y \)
  - (ignore normalization for now)

# Transforming Normals

- so if points transformed by matrix \( M \), can we just transform normal vector by \( M \) too?
  - translations OK: \( w = 0 \) means unaffected
  - rotations OK
  - uniform scaling OK
  - these all maintain direction

# Finding Correct Normal Transform

- transform a plane

\[
P \rightarrow P' = MP
\]

- \( N' = QN \)
- \( N' = QMP \)
- \( Q^T = I \)
- \( N'P = 0 \) if \( Q^T M = I \)

**thus the normal to any surface can be transformed by the inverse transpose of the modeling transformation**

# Computing Normals

- normal
  - direction specifying orientation of polygon
  - \( w = 0 \) means direction with homogeneous coords
  - vs. \( w = 1 \) for points/vectors of object vertices
  - used for lighting
  - must be normalized to unit length
  - can compute if not supplied with object

# Planes and Normals

- plane is all points perpendicular to normal
  - \( N \cdot P = 0 \) (dot product)

\[
N = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}
\]

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

\[
N = (P_2 - P_1) \times (P_1 - P_2)
\]

- explicit form: plane = \( ax + by + cz + d = 0 \)

# Transforming Geometric Objects

- lines, polygons made up of vertices
  - transform the vertices
  - interpolate between vertices
  - does this work for everything? no!
  - normals are trickier

# 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
      - interactive graphics: objects redrawn every frame from moving camera

# Review: Display Lists

- precompile/cache block of OpenGL code for reuse
- usually more efficient than immediate mode
- exact optimizations depend on driver
- good for multiple instances of same object
- but cannot change contents, not parametrizable
- good for static objects redrawn often
- display lists persist across multiple frames
- interactive graphics: objects redrawn every frame from new viewpoint from moving camera
- can be nested hierarchically

# Making Display Lists

GLint createDL()

```c
GLuint createDLGLuint createDL() {
// Create the id for the list
snowManDL = glGenLists(1);  
newList(snowManDL, GL_COMPILE);  
drawSnowMan();
return(snowManDL);  };
}
```
Rendering Pipeline

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

Convenient Camera Motion

- rotate/translate/scale versus
- eye point, gaze/lookat direction, up vector
- demo: Robins transformation, projection

Coordinate Systems

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

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
  - glMatrixMode(GL_MODELVIEW);
  - glMatrixMode(GL_PROJECTION);

Coordinate Systems

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

Basic Viewing

- starting spot - OpenGL
- camera at world origin
- probably inside an object
- y axis is up
- looking down negative z axis
- why? RNS 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

OpenGL Viewing Transformation

- gluLookAt(ex, ey, ez, lx, ly, lz, ux, uy, uz)
- postmultiplies current matrix, so to be safe:
  - glMatrixMode(GL_MODELVIEW);
  - glLoadIdentity();
  - gluLookAt(ex, ey, ez, lx, ly, lz, ux, uy, uz)
  - // now ok to do model transformations
- demo: Nate Robins tutorial projection

Viewing Transformation

- from world to view coordinates: W2V
Deriving W2V Transformation

- translate eye to origin
  \[ T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{bmatrix} \]

- rotate view vector (lookat – eye) to w axis
  \[ w = -\hat{g} = -\frac{g}{\|g\|} \]

W2V vs. V2W

- \( M_{W2V} = TR \)
- \( M_{V2W} = R^{-1}T^{-1} \)
- we derived position of camera in world
  - invert for world with respect to camera
  \[ M_{W2V} = (M_{W2V})^{-1} \]
- inverse is transpose for orthonormal matrices
- inverse is negative for translations

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 \( z = 7 \)
    - \( \)Note OpenGL models viewing in left-hand coordinates
    - camera stays put, but world moves to \( z = -7 \)
    - resulting image same either way
    - possible difference: are lights specified in world or view coordinates?

World vs. Camera Coordinates Example

\[ a = (1,1)_w \]
\[ b = (1,1)_c = (5,3)_w \]
\[ c = (1,1)_c = (1,3)_c = (5,5)_w \]