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
• snowman example
  http://www.lighthouse3d.com/opengl/displaylists
void drawSnowMan() {

    glColor3f(1.0f, 1.0f, 1.0f);

    // Draw Body
    glTranslatef(0.0f, 0.75f, 0.0f);
    glutSolidSphere(0.75f, 20, 20);

    // Draw Head
    glTranslatef(0.0f, 1.0f, 0.0f);
    glutSolidSphere(0.25f, 20, 20);

    // Draw Nose
    glColor3f(1.0f, 0.5f, 0.5f);
    glRotatef(0.0f, 1.0f, 0.0f, 0.0f);
    glutSolidCone(0.08f, 0.5f, 10, 2);
}

// Draw Eyes
glPushMatrix();
    glColor3f(0.0f, 0.0f, 0.0f);
    glTranslatef(0.05f, 0.10f, 0.18f);
    glutSolidSphere(0.05f, 10, 10);
    glTranslatef(-0.1f, 0.0f, 0.0f);
    glutSolidSphere(0.05f, 10, 10);
glPopMatrix();

// Draw Nose
    glColor3f(1.0f, 0.5f, 0.5f);
Instantiate Many Snowmen

// Draw 36 Snowmen
for(int i = -3; i < 3; i++)
    for(int j=-3; j < 3; j++) {
        glPushMatrix();
        glTranslatef(i*10.0, 0, j * 10.0);
        // Call the function to draw a snowman
        drawSnowMan();
        glPopMatrix();
    }

36K polygons, 55 FPS
GLuint createDL() {
    GLuint snowManDL;
    // Create the id for the list
    snowManDL = glGenLists(1);
    glNewList(snowManDL, GL_COMPILE);
    drawSnowMan();
    glEndList();
    return(snowManDL); }

snowmanDL = createDL();
for(int i = -3; i < 3; i++)
    for(int j=-3; j < 3; j++) {
        glPushMatrix();
        glTranslatef(i*10.0, 0, j * 10.0);
        glCallList(Dlid);
        glPopMatrix(); }
36K polygons, 153 FPS
Transforming Normals
Transforming Geometric Objects

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

\[
N = (P_2 - P_1) \times (P_3 - P_1)
\]
Transforming Normals

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

• so if points transformed by matrix \( \mathbf{M} \), can we just transform normal vector by \( \mathbf{M} \) too?
  • translations OK: \( w=0 \) means unaffected
  • rotations OK
  • uniform scaling OK

• these all maintain direction
Transforming Normals

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

- apply nonuniform scale: stretch along x by 2
  - new plane x = 2y
- transformed normal: [2,-1,0]

$$\begin{bmatrix}
2 \\
-1 \\
0 \\
0
\end{bmatrix} = \begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 \\
-1 \\
0 \\
0
\end{bmatrix}$$

- normal is direction of line x = -2y or x+2y=0
- not perpendicular to plane!
- should be direction of 2x = -y
Planes and Normals

• plane is all points perpendicular to normal
  • \( N \cdot P = 0 \) (with dot product)
  • \( N^T \cdot P = 0 \) (matrix multiply requires transpose)

\[
N = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}
\]

• explicit form: plane = \( ax + by + cz + d \)
Finding Correct Normal Transform

• transform a plane

\[
\begin{align*}
P & \quad \rightarrow \quad P' = MP \\
N & \rightarrow \quad N' = QN
\end{align*}
\]

given M, what should Q be?

stay perpendicular

substitute from above

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

\[
N^T P = 0 \text{ if } Q^T M = I
\]

\[
Q = \left( M^{-1} \right)^T
\]
Reading for This and Next 2 Lectures

- FCG Chapter 7 Viewing
- FCG Section 6.3.1 Windowing Transforms
- RB rest of Chap Viewing
- RB rest of App Homogeneous Coords
Viewing
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
    - projective transformation
Rendering Pipeline

Scene graph
Object geometry

Modelling
Transforms

Viewing
Transform

Projection
Transform
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

- result
  - 2D screen coordinates of clipped vertices
Viewing and Projection

- need to get from 3D world to 2D image
- projection: geometric abstraction
  - what eyes or cameras do
- two pieces
  - viewing transform:
    - where is the camera, what is it pointing at?
  - perspective transform: 3D to 2D
    - flatten to image
Rendering Pipeline

- Geometry Database
- Model/View Transform.
- Lighting
- Perspective Transform.
- Clipping
- Scan Conversion
- Texturing
- Depth Test
- Blending
- Frame-buffer
Rendering Pipeline
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
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

object transformation
O2W world transformation
W2V viewing transformation
V2C projection transformation
C2N clipping transformation
N2D normalized transformation
D2C device transformation

modeling transformation
Viewing Transformation

object OCS world WCS viewing VCS

modeling transformation modeling transformation viewing transformation

\[ M_{\text{mod}} \rightarrow M_{\text{cam}} \]

OpenGL ModelView matrix
Basic Viewing

• starting spot - OpenGL
  • camera at world origin
    • probably inside an object
  • y axis is up
  • looking down negative z axis
    • why? RHS with x horizontal, y vertical, z out of screen
• translate backward so scene is visible
  • move distance \( d = \text{focal length} \)

• where is camera in P1 template code?
  • 5 units back, looking down -z axis
Convenient Camera Motion

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

- rotate/translate/scale versus
  - eye point, gaze/lookat direction, up vector
From World to View Coordinates: W2V

- translate **eye** to origin
- rotate **view** vector (**lookat** – **eye**) to **w** axis
- rotate around **w** to bring **up** into **vw**-plane
Deriving W2V Transformation

- **translate eye to origin**

\[
T = \begin{bmatrix}
1 & 0 & 0 & e_x \\
0 & 1 & 0 & e_y \\
0 & 0 & 1 & e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Deriving W2V Transformation

- rotate **view** vector (**lookat** – **eye**) to **w** axis
  - **w**: normalized opposite of **view/gaze** vector **g**

\[ w = -\hat{g} = -\frac{g}{\|g\|} \]
Deriving W2V Transformation

- rotate around \( w \) to bring \( \text{up} \) into \( vw \)-plane
  - \( u \) should be perpendicular to \( vw \)-plane, thus perpendicular to \( w \) and \( \text{up} \) vector \( t \)
  - \( v \) should be perpendicular to \( u \) and \( w \)

\[
\begin{align*}
  u &= \frac{t \times w}{||t \times w||} \\
  v &= w \times u
\end{align*}
\]
Deriving W2V Transformation

- rotate from WCS $xyz$ into $uvw$ coordinate system with matrix that has columns $u$, $v$, $w$

$$u = \frac{t \times w}{|t \times w|} \quad v = w \times u \quad w = -\hat{g} = -\frac{g}{|g|}$$

$$R = \begin{bmatrix}
    u_x & v_x & w_x & 0 \\
    u_y & v_y & w_y & 0 \\
    u_z & v_z & w_z & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix} \quad T = \begin{bmatrix}
    1 & 0 & 0 & e_x \\
    0 & 1 & 0 & e_y \\
    0 & 0 & 1 & e_z \\
    0 & 0 & 0 & 1
\end{bmatrix} \quad M_{W2V} = TR$$

- reminder: rotate from $uvw$ to $xyz$ coord sys with matrix $M$ that has columns $u,v,w$
**W2V vs. V2W**

- \( M_{W2V} = TR \)

  \[
  T = \begin{bmatrix}
  1 & 0 & 0 & e_x \\
  0 & 1 & 0 & e_y \\
  0 & 0 & 1 & e_z \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

  \[
  R = \begin{bmatrix}
  u_x & v_x & w_x & 0 \\
  u_y & v_y & w_y & 0 \\
  u_z & v_z & w_z & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

- we derived position of camera in world
  - invert for world with respect to camera
- \( M_{V2W} = (M_{W2V})^{-1} = R^{-1}T^{-1} \)

  \[
  R^{-1} = \begin{bmatrix}
  u_x & u_y & u_z & 0 \\
  v_x & v_y & v_z & 0 \\
  w_x & w_y & w_z & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

  \[
  T^{-1} = \begin{bmatrix}
  1 & 0 & 0 & -e_x \\
  0 & 1 & 0 & -e_y \\
  0 & 0 & 1 & -e_z \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

- inverse is transpose for orthonormal matrices
- inverse is negative for translations
W2V vs. V2W

- \( M_{W2V} = TR \)
- \( T = \begin{bmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \)
- \( R = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \)
- we derived position of camera in world
  - invert for world with respect to camera
- \( M_{V2W} = (M_{W2V})^{-1} = R^{-1}T^{-1} \)

\[
M_{\text{view2world}} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & -e_x \\ v_x & v_y & v_z & -e_y \\ w_x & w_y & w_z & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]
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 = -3
      • Note OpenGL models viewing in left-hand coordinates
    • camera stays put, but world moves to -7
  • resulting image same either way
    • possible difference: are lights specified in world or view coordinates?
World vs. Camera Coordinates Example

- $a = (1,1)_{w}$
- $b = (1,1)_{c1} = (5,3)_{w}$
- $c = (1,1)_{c2} = (1,3)_{c1} = (5,5)_{w}$