Chapter 4: Transformations- Transforming Normals, Hierarchies and OpenGL, Assignment 2

Transforming Normals
Computing Normals

- polygon:
  \[ N = (P_2 - P_1) \times (P_3 - P_1) \]
  
- assume vertices ordered CCW when viewed from visible side of polygon
  
- normal for a vertex
  - used for lighting
  - supplied by model (i.e., sphere), or computed from neighboring polygons

Transforming Normals

- What is a normal?
  - **Vector**
    - Orthogonal (perpendicular) to plane/surface

- Do standard transformations preserve orthogonality?
Planes and Normals

- Plane - all points where \( N \cdot P = 0 \)

\[
P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \quad N = \begin{bmatrix} A \\ B \\ C \\ 0 \end{bmatrix}
\]

- Implicit form

\[
\text{Plane} = A \cdot x + B \cdot y + C \cdot z + D
\]

Finding Correct Normal Transform

- transform a plane

\[
P \quad N \quad P' = MP \quad N' = QN
\]

\[
N'^T \cdot P' = 0 \quad \text{Given M, find Q}
\]

\[
(QN)^T \cdot (MP) = 0 \quad \text{stay perpendicular}
\]

\[
N^T \cdot Q^T \cdot MP = 0 \quad \text{substitute from above}
\]

\[
Q^T \cdot M = I \quad \text{(AB)}^T = B^T \cdot A^T
\]

\[
Q = (M^{-1})^T \quad \text{Normal transformed by transpose of the inverse of the modeling transformation}
\]

Copyright 2011, A. Sheffer, UBC
Transformations in OpenGL

The Rendering Pipeline

Geometric Content → Model/View Transform. → Lighting → Perspective Transform. → Clipping → Geometry Processing

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer → Rasterization, Fragment Processing
Modeling Transformation

- Purpose:
  - Map geometry from local object coordinate system into a global world coordinate system
  - Same as placing objects
- Transformations:
  - Arbitrary affine transformations are possible
    - More complex transformations may be desirable
    - Freeform deformations
  - Not available in hardware

Viewing Transformation

- Purpose:
  - Map geometry from world coordinate system into camera coordinate system
  - Camera coordinate system is right-handed, viewing direction is negative z-axis
  - Same as placing camera
- Transformations:
  - Usually only rigid body transformations
    - Rotations and translations
  - Objects have same size and shape in camera and world coordinates
Model/View Transformation

- Combine modeling and viewing transform
  - Combine into single matrix
  - Saves computation time
    - if many points are to be transformed
  - Possible because viewing transformation directly follows modeling transformation without intermediate operations

Transformations in OpenGL

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_LINE_LOOP);
  glVertex2f(0,0);
  glVertex2f(2,0);
  glVertex2f(2,2);
  glVertex2f(1,3);
  glVertex2f(0,2);

DrawHouse()

```

Copyright 2011, A. Sheffer, UBC
Transformations in OpenGL

\[
\begin{bmatrix}
    x \\
    y \\
    z \\
    w
\end{bmatrix} = \begin{bmatrix}
    2 & 0 & 0 & 3 \\
    0 & 2 & 0 & 1 \\
    0 & 0 & 2 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1_{obj}
\end{bmatrix}
\]

GLfloat T[16] = { 2,0,0,0, 0,2,0,0, 0,0,2,0, 3,1,0,1};

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(3,1,0);
glScalef(2,2,2);

DrawHouse();

Transformations in OpenGL

- An easier way to do the same thing....

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(3,1,0);
glScale(2,2,2);

DrawHouse();
Matrix Operations in OpenGL

- 2 Matrices:
  - Model/view matrix $M$
  - Projective matrix $P$
- Example:

```c
glMatrixMode( GL_MODELVIEW );
glLoadIdentity(); // $M=\text{Id}$
glRotatef( angle, x, y, z ); // $M= R(\alpha)\text{Id}$
glTranslatef( x, y, z ); // $M= T(x,y,z)*R(\alpha)\text{Id}$
glMatrixMode( GL_PROJECTION );
glRotatef( … ); // $P= \text{…}$
```

Composing Transformations

Suppose we want

- Rotate($z, -90$)
- Translate($2, 3, 0$)

$P_A = \text{Rot}(z, -90) P_h$

$P_W = \text{Trans}(2,3,0) P_A$

$P_W = \text{Trans}(2,3,0) \text{Rot}(z, -90) P_h$
Composing Transformations

\[ P_W = Trans(2,3,0)Rot(z,-90)P_h \]

- R-to-L: interpret operations wrt fixed coords
  - moving object
- L-to-R: interpret operations wrt local coords
  - changing coordinate system
- OpenGL (L-to-R, local coords)

\[ \begin{align*}
    M_{MV} &= Trans(2,3,0) \cdot M_{MV} \\
    M_{MV} &= Rot(z,-90)M_{MV}
\end{align*} \]

Post Multiplication

- Composite transformation is now just the product of a few matrixes
- Rather than multiply each point sequentially with 3 matrices, first multiply the matrices, then multiply each point with only one (composite) matrix
  - Much faster for large # of points!
  - Same reason to use homogeneous coordinates
Interpreting Composite OpenGL Transformations

- Example from earlier lectures:
  - Rotation around arbitrary center
  - In OpenGL:

```c
// initialization of matrix
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(4, 3);
glRotatef(30, 0.0, 0.0, 1.0);
glTranslatef(-4, -3);
glBegin(GL_TRIANGLES);
// specify object geometry...
```

Top-to-bottom: transf. of coordinate frame
Bottom-to-top: transf. of object

Transformation Hierarchies
Transformation Hierarchies

- scene may have a hierarchy of coordinate systems
  - Multiple objects, multiple joint links, ...
  - stores matrix at each level with incremental transform from parent's coordinate system

Transformation Hierarchies

world

- torso
  - LLeg
  - RLeg
  - LLarm
  - RLarm
  - Lfoot
  - Rfoot
  - Lhand
  - Rhand
  - head

rot(z,θ) trans(0.30,0,0)
Check out: Brown Applets

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

Matrix Stacks

- Avoiding unnecessary computation when incremental processing makes no sense
- Using inverse to return to origin costs.

Copyright 2011, A. Sheffer, UBC
Matrix Stacks

```plaintext
Matrix Stacks

D = C scale(2,2,2) trans(1,0,0)

```
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 ~4 for projective matrix

Transformation Hierarchy Examples

```c
glLoadIdentity();
glLoadIdentity();
glTranslatef(4,1,0);
glTranslatef(4,1,0);
glPushMatrix();
glPushMatrix();
glRotatef(45,0,0,1);
glRotatef(45,0,0,1);
glTranslatef(0,2,0);
glTranslatef(0,2,0);
glScalef(2,1,1);
glScalef(2,1,1);
glTranslate(1,0,0);
glTranslate(1,0,0);
glPopMatrix();
glPopMatrix();
glLoadIdentity();
glTranslatef(4,1,0);
glPushMatrix();
glRotatef(45,0,0,1);
glTranslatef(0,2,0);
glScalef(2,1,1);
glTranslatef(1,0,0);
glPopMatrix();
```
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
Assignment 2

- Out last week, due **4pm Fri Oct 14, 2011**
- Start very soon!
- Build and animate a giraffe made out of spheres and 4x4 matrices
  - think cartoon, not beauty
- Template code - program shell, Makefile
Advice

- **Draw one section at a time**
  - Ensure you’re constructing hierarchy correctly
  - Use body as scene graph root
  - Continue with attached parts
  - Finish all required parts before...
    - ...Adding extra links or DOFs
    - ...Going for extra credit

- Visual debugging
  - Draw the current coord system

Advanced transformations example

- Deformation Transfer [Sumner’05]
  - Use transformation gradients (transformation without translation) as per-triangle encoding of motion
Advanced transformations example

- Deformation Transfer