OpenGL Transformations, Hierarchical Transformations, Accelerations

**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
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.

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

Homogeneous Matrices

**Affine Transformations**

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

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

© Wolfgang Heidrich
Homogeneous Vectors

Representing vectors in homogeneous coordinates

- Column vectors with \( w = 0 \)

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

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
  - Even more complex transformations may be desirable, but are not available in hardware
    - Freeform deformations
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 a 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 both into a single matrix
- Saves computation time if many points are to be transformed
- Possible because the viewing transformation directly follows the modeling transformation without intermediate operations
Homogeneous Planes And Normals

Planes in Cartesian Coordinates:
\[(x, y, z)^T | nx \cdot x + ny \cdot y + nz \cdot z + d = 0\]

- \(nx, ny, nz\), and \(d\) are the parameters of the plane (normal and distance from origin)

Planes in Homogeneous Coordinates:
\[[x, y, z, w]^T | nx \cdot x + ny \cdot y + nz \cdot z + dw = 0\]

Homogeneous Planes And Normals

Planes in homogeneous coordinates are represented as row vectors

- \(E=[nx, ny, nz, d]\)
- Condition that a point \([x,y,z,w]^T\) is located in \(E\)
\[
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix} \in E = [nx, ny, nz, d] \Leftrightarrow [nx, ny, nz, d] \cdot 
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix} = 0
\]
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:
  \[ \mathbf{n} = \left[ \frac{-\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}, 0 \right] \]
- Distance of line from origin:
  \[ d = \frac{\sqrt{2}}{2} \]
- Thus:
  \[ \mathbf{L} = \left[ \frac{-\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}, \frac{\sqrt{2}}{2} \right] = [-1, -1, 1] \]

Example in 2D (cont.):

- Is \([1,0,1]^T\) on the line?
  \[ \begin{bmatrix} \frac{-\sqrt{2}}{2}, & \frac{-\sqrt{2}}{2}, & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \frac{-\sqrt{2}}{2} + 0 + \frac{\sqrt{2}}{2} = 0 \]
- What about \([0,0,1]^T, [1,1,1]^T\)?
  \[ \begin{bmatrix} \frac{-\sqrt{2}}{2}, & \frac{-\sqrt{2}}{2}, & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \frac{\sqrt{2}}{2} \];
  \[ \begin{bmatrix} \frac{-\sqrt{2}}{2}, & \frac{-\sqrt{2}}{2}, & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = -\frac{\sqrt{2}}{2} \]
Homogeneous Planes And Normals

Transformations of planes

\[
\begin{bmatrix}
  x \\
  y \\
  z \\
  w
\end{bmatrix}
\cdot \begin{bmatrix}
  n_x \\
  n_y \\
  n_z \\
  d
\end{bmatrix} = 0 \Leftrightarrow T([n_x,n_y,n_z,d]) \cdot (A \cdot \begin{bmatrix}
  x \\
  y \\
  z \\
  w
\end{bmatrix}) = 0
\]

- Works for \( T([n_x,n_y,n_z,d]) = [n_x,n_y,n_z,d]A^{-1} \)
- Thus: planes have to be transformed by the inverse of the affine transformation (multiplied from left as a row vector)!
Homogeneous
Planes And Normals

Homogeneous Normals
- The plane definition also contains its normal
- Normal written as a vector \([n_x, n_y, n_z, 0]^T\)

\[
\begin{bmatrix}
  n_x \\
  n_y \\
  n_z \\
  0
\end{bmatrix}
\begin{bmatrix}
  v_x \\
  v_y \\
  v_z \\
  0
\end{bmatrix}
= 0 \iff ((A^{-T} \cdot \begin{bmatrix}
  n_x \\
  n_y \\
  n_z \\
  0
\end{bmatrix}) \cdot (A \cdot \begin{bmatrix}
  v_x \\
  v_y \\
  v_z \\
  0
\end{bmatrix})) = 0
\]

Thus: the normal to any surface has to be transformed by the inverse transpose of the affine transformation (multiplied from the right as a column vector)!

Transforming Homogeneous
Normals

Back to 2D example:
- Before transformation

\[
L = \begin{bmatrix}
  -\sqrt{2} \\
  \sqrt{2} \\
  \sqrt{2}
\end{bmatrix} / 2
\]

\[
n = \begin{bmatrix}
  -\sqrt{2} \\
  \sqrt{2} \\
  \sqrt{2} \\
  0
\end{bmatrix} / 2
\]
Transforming Homogeneous Normals

**Scale by 1/2 in y direction**

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} = \begin{bmatrix} 1 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}
\]

\[
n' = M^{-T}n = \begin{bmatrix} 1 & 2 & 1 \\ -\sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ \sqrt{2} & \sqrt{2} & 0 \end{bmatrix}
\]

Transforming Homogeneous Normals

**Inverse Transpose of**

- Rotation by \( \alpha \)
  - *Rotation by* \( \alpha \)
- Scale by \( s \)
  - *Scale by* \( 1/s \)
- Translation by \( t \)
  - *Identity matrix!*
- Shear by \( a \) along x axis
  - *Shear by* \(-a\) along y axis
OpenGL Transformations, Hierarchical Transformations, Accelerations

**CPSC 314**

The Rendering Pipeline

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

Geometry Processing

- Scan Conversion
- Texturing
- Depth Test
- Blending

Fragment Processing

- Frame-buffer

© Wolfgang Heidrich
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
Rendering Geometry in OpenGL

Example:
```
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:**
- **gl Vertex 3 f**
  - OpenGL Prefix
  - Operation: gl
  - Dimensionality: 3
  - Type of parameters: float
  - Missing coordinates: 1 (w)
Matrix Operations in OpenGL

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

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

Matrix Operations in OpenGL

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

Specifying matrices (replacement)
- glEnable(GL_DITHER)
- glHint(GL_DITHER, GL_DITHER_AUTO)

Specifying matrices (multiplication)
- glMatrixMode(GL_MODELVIEW)
- glLoadIdentity()
- glMultMatrixf( GLfloat *m ) // 16 floats

Perspective Matrices (details next lecture):
- 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
Interpreting Composite Transformations

Example for last lecture: Rotation around arbitrary center

- E.g. rotate 30 degrees around (4,3)
- In OpenGL:
  vector3f center(4,3,0);
  glTranslatef( center );
  glRotatef( 30 );
  glTranslatef( -center );

Interpreting Composite Transformations

Interpretation 1: moving the coordinate system

- Read operations in forward order
  vector3f center(4,3,0);
  glTranslatef( center );
  glRotatef( 30 );
  glTranslatef( -center );
Interpreting Composite Transformations

*Interpretation 2: moving the object*

- Read operations in reverse order
  
  ```
  glTranslatef(4, 3);
  glRotatef(30);
  glTranslatef(-4, -3);
  ```

Compositing of Affine Transformations

*Example: Rotation around arbitrary center*

- Step 2: perform rotation
Compositing of Affine Transformations

Example: Rotation around arbitrary center

- Step 3: back to original coordinate system

Compositing of Affine Transformations

Example: Rotation around arbitrary center

- Step 1: translate coordinate system to rotation center
Compositing of Affine Transformations

Example: Rotation around arbitrary center

- Step 2: perform rotation

Example: Rotation around arbitrary center

- Step 3: back to original coordinate system
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
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

\[ D = C \text{ scale}(2,2,2) \text{ trans}(1,0,0) \]

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();
glTranslatef(45, 0, 0, 1);
glTranslatef(0, 2, 0);
glScalef(2, 1, 1);
glTranslatef(1, 0, 0);
glPopMatrix();
```
**Transformation Hierarchy Example 4**

```c
glTranslatef(x, y, 0);
glRotatef(\theta_1, 0, 0, 1);
DrawBody();
glPushMatrix();
  glTranslatef(0, 7, 0);
  DrawHead();
  glPopMatrix();
  glTranslatef(2.5, 5.5, 0);
  glRotatef(\theta_2, 0, 0, 1);
  DrawUArm();
  glTranslatef(0, -3.5, 0);
  glRotatef(\theta_3, 0, 0, 1);
  DrawLArm();
  glPopMatrix();
... (draw other arm)
```

---

**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 $s$

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

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:

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

![Rendering Pipeline Diagram](image-url)
Coming Up...

**Thursday, Sep 20:**
- Perspective transformations

**Tuesday, Sep 25:**
- Lighting