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

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

Homogeneous Matrices

**Affine Transformations**

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

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 aspect ratio

**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:**
\[(x, y, z) \mid n_x x + n_y y + n_z z + d = 0\]
- \(n_x, n_y, n_z\) and \(d\) are the parameters of the plane (normal and distance from origin)

**Planes in Homogeneous Coordinates:**
\[(x, y, z, w) \mid n_x x + n_y y + n_z z + d w = 0\]

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:
  \[n = \left[ \frac{-\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}, 0 \right]\]
- Distance of line from origin:
  \[d = \sqrt{2} / 2\]
- Thus:
  \[L = \left[ \frac{-\sqrt{2}}{2}, \frac{-\sqrt{2}}{2}, \frac{\sqrt{2}}{2} \right] = \left[ -1, -1, 1 \right]\]

**Example in 2D (cont.):**
- Is \([1,0,1]^T\) on the line?
  \[\begin{bmatrix} -\sqrt{2}/2 & -\sqrt{2}/2 & 0 \\ 0 & \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -\sqrt{2}/2 \end{bmatrix}\]
- What about \([0,0,1]^T, [1,1,1]^T\)?
  \[\begin{bmatrix} -\sqrt{2}/2 & -\sqrt{2}/2 & 0 \\ 0 & \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2 \end{bmatrix}\]
Homogeneous Planes And Normals

Transformations of planes

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

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 \([n_x, n_y, n_z, 0]^T\)

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

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

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

\[
n = \frac{1}{2} \begin{bmatrix} -2 \\ 2 \\ 2 \end{bmatrix}
\]

Transforming Homogeneous Normals

Scale by 1/2 in y direction

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

\[
n' = M^{-1}n = \frac{1}{2} \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}
\]

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
Open GL Transformations, Hierarchical Transformations, Accelerations

**CPSC 314**

---

**Rendering Geometry in Open GL**

```c
glBegin(GL_TRIANGLES);
gVertex3f(x1, y1, z1); // vertex 1 of triangle 1
gVertex3f(x2, y2, z2); // vertex 2 of triangle 1
gVertex3f(x3, y3, z3); // vertex 3 of triangle 1
gVertex3f(x4, y4, z4); // vertex 1 of triangle 2
gVertex3f(x5, y5, z5); // vertex 2 of triangle 2
gVertex3f(x6, y6, z6); // vertex 3 of triangle 2
...
gEnd();
```

---

**Rendering Geometry in Open GL**

*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

---

**OpenGL Naming Scheme**

*Function names:

<table>
<thead>
<tr>
<th>OpenGL Prefix</th>
<th>Operation</th>
<th>Dimensionality</th>
<th>Type of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gl</td>
<td>3</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>glVertex</td>
<td>1</td>
<td>(double)</td>
</tr>
<tr>
<td></td>
<td>3f</td>
<td>1</td>
<td>(integer)</td>
</tr>
</tbody>
</table>
Matrix Operations in OpenGL

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

Example:
```c
void f(size_t frame, const float *v)
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity(); // M = \text{Id}
    glRotatef(angle, x, y, z); // M = \text{Id} \cdot R(\alpha)
    glTranslatef(x, y, z); // M = \text{Id} \cdot R(\alpha) \cdot T(x,y,z)
    glMatrixMode(GL_PROJECTION);
    glRotatef(\ldots); // P = \ldots
}
```

Matrix Operations in OpenGL

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

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

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:
  ```c
glTranslatef(4,3);
glRotatef(30);
glTranslatef(-4,-3);
```

Interpreting Composite Transformations

Interpretation 1: moving the coordinate system
- Read operations in forward order
  ```c
glTranslatef(4,3);
glRotatef(30);
glTranslatef(-4,-3);
```
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 2: perform rotation

Compositing of Affine Transformations

*Example: Rotation around arbitrary center*
- Step 1: translate coordinate system to rotation 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 Hierarchies

- Hierarchies don't fall apart when changed
- Transforms apply to graph nodes beneath

Transformation Hierarchy Example 1

Transformation Hierarchy Example 2

- Draw same 3D data with different transformations: instancing

Brown Applets

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

- Have a look later

Matrix Stacks

Challenge of avoiding unnecessary computation

- Using inverse to return to origin
- Computing incremental \( T_1 \rightarrow T_2 \)
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

Modularization

**Drawing a scaled square**
- Push/pop ensures no coord system change

```c
void drawBlock(float x) {
  glPushMatrix();
  glScalef(x, x, x);
  glBegin(GL_LINE_LOOP);
  glVertex2f(0, 0);
  glVertex2f(1, 0);
  glVertex2f(1, 1);
  glVertex2f(0, 1);
  glEnd();
  glPopMatrix();
}
```

Transformation Hierarchy

**Example 3**

```c
void transformExample3() {
  glLoadIdentity();
  glTranslatef(4, 1, 0);
  glPushMatrix();
  glTranslatef(45, 0, 0.1);
  glTranslatef(0, 1, 0);
  glScalef(2, 2, 2);
  glTranslatef(1, 0, 0);
  glPopMatrix();
  glPopMatrix();
}
```

**Example 4**

```c
void transformExample4() {
  glLoadIdentity();
  glTranslatef(x, y, z);
  glRotatef(θ1, 0, 1, 0);
  glRotatef(θ2, 1, 0, 0);
  glRotatef(θ3, 0, 0, 1);
  glBegin(GL_LINES);
  glVertex2f(x, y);
  glVertex2f(x + 2, y);
  glEnd();
  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
**Single Parameter: simple**

**Parameters as functions of other params**
- Clock: control all hands with seconds

\[
m = s / 60, \ h = m / 60, \\
\theta_s = (2 \pi s) / 60, \\
\theta_m = (2 \pi m) / 60, \\
\theta_h = (2 \pi h) / 60
\]

**Display Lists**

**Concept:**
- If multiple copies of an object are required, it can be compiled into a display list:

```c
glNewList( listId, GL_COMPILE );
gBegin( ... );
... // geometry goes here
gLEndList();
// render two copies of geometry offset by 1 in z-direction:
gCallList( listId );
gTranslate( 0.0, 0.0, 1.0 );
gCallList( 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 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

http://www.flying-pig.co.uk
**Triangle Strips**

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

**Similar concept:**
- All triangles share on center vertex
- All other vertices are specified in CCW order

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

**The Rendering Pipeline**

*Geometry Database* → *ModelView Transform* → *Lighting* → *Perspective Transform* → *Clipping* → *Fragment Processing* → *Frame-buffer*
Coming Up…

Thursday, Sep 20:
• Perspective transformations

Tuesday, Sep 25:
• Lighting