Transformations
3D is just like taking a photograph (lots of photographs!)
Transformations take us from one “space” to another

- All of our transforms are $4 \times 4$ matrices

Diagram:

- Vertex Data
- Model-View Transform
- Projection Transform
- Perspective Division ($w$)
- Viewport Transform
- 2D Window Coordinates

Spaces:

- World Coords.
- Eye Coords.
- Clip Coords.
- Normalized Device Coords.
Camera Analogy and Transformations

- **Projection transformations**
  - adjust the lens of the camera

- **Viewing transformations**
  - tripod—define position and orientation of the viewing volume in the world

- **Modeling transformations**
  - moving the model

- **Viewport transformations**
  - enlarge or reduce the physical photograph
3D Transformations

- A vertex is transformed by $4 \times 4$ matrices
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
    - this is opposite of what “C” programmers expect

- matrices are always post-multiplied
- product of matrix and vector is $\mathbf{Mv}$

$$\mathbf{M} = \begin{bmatrix}
m_0 & m_4 & m_8 & m_{12} \\
m_1 & m_5 & m_9 & m_{13} \\
m_2 & m_6 & m_{10} & m_{14} \\
m_3 & m_7 & m_{11} & m_{15}
\end{bmatrix}$$
Set up a viewing frustum to specify how much of the world we can see

Done in two steps

- specify the size of the frustum (projection transform)
- specify its location in space (model-view transform)

Anything outside of the viewing frustum is clipped

- primitive is either modified or discarded (if entirely outside frustum)
OpenGL projection model uses \textit{eye coordinates}

- the “eye” is located at the origin
- looking down the -z axis

Projection matrices use a six-plane model:

- near (image) plane and far (infinite) plane
  - both are distances from the eye (positive values)
- enclosing planes
  - top & bottom, left & right
Position the camera/eye in the scene
  - place the tripod down; aim camera
To “fly through” a scene
  - change viewing transformation and redraw scene

\[ \text{LookAt}( \text{eye}_x, \text{eye}_y, \text{eye}_z, \]
\[ \text{look}_x, \text{look}_y, \text{look}_z, \]
\[ \text{up}_x, \text{up}_y, \text{up}_z ) \]
  - up vector determines unique orientation
  - careful of degenerate positions
Move the origin to a new location

$$T(t_x, t_y, t_z) = \begin{pmatrix}
1 & 0 & 0 & t_x \\
0 & 1 & 0 & t_y \\
0 & 0 & 1 & t_z \\
0 & 0 & 0 & 1
\end{pmatrix}$$
Scale

Stretch, mirror or decimate a coordinate direction

\[ S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

Note, there’s a translation applied here to make things easier to see.
Rotate coordinate system about an axis in space

Note, there’s a translation applied here to make things easier to see
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
}
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0, 0.0, 0.0, 0.0,  
                0.0, c.x, s.x, 0.0,  
                0.0, -s.x, c.x, 0.0,  
                0.0, 0.0, 0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,  
                0.0, 1.0, 0.0, 0.0,  
                s.y, 0.0, c.y, 0.0,  
                0.0, 0.0, 0.0, 1.0 );
mat4 rz = mat4(c.z, -s.z, 0.0, 0.0, 
s.z, c.z, 0.0, 0.0, 
0.0, 0.0, 1.0, 0.0, 
0.0, 0.0, 0.0, 1.0);

color = vColor;
gl_Position = rz * ry * rx * vPosition;
// compute angles using mouse and idle callbacks
GLuint theta;  // theta uniform location
vec3 Theta;   // Axis angles

void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, Theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );

    glutSwapBuffers();
}