Outcomes

- Understand the 3 basic transformations
- See and use basic matrix math underlying transformations
- Understand basic methods of animation control

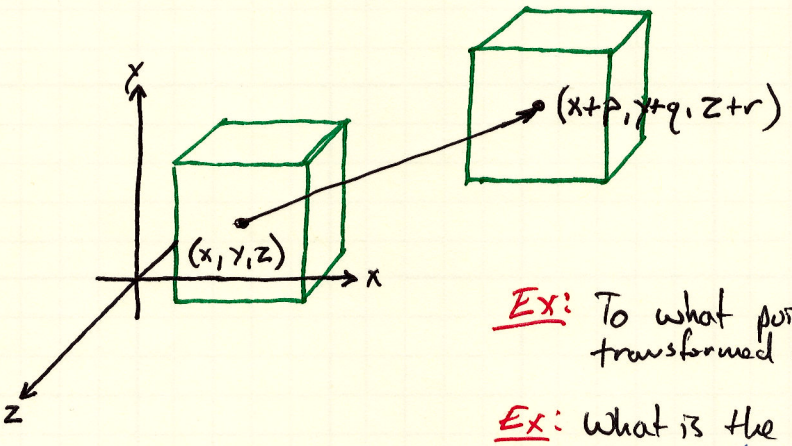
Modeling Transformations

- Transformations → Change object orientation, location, and shape
- 3 Types
  - Translation
  - Scaling
  - Rotation

Translation

- glTranslatef(p, q, r) ≡ translates an object p units in the x direction, q units in the y direction and r units in the z direction
  - Thus each point (x, y, z) of an object is mapped to (x+p, y+q, z+r)
  - This is the result of vector addition $\vec{p} + \vec{p'}$, where

$$\vec{p} = (x, y, z) \quad \vec{p'} = (p, q, r)$$



Ex: To what point is (-2.0, 3.0, 9.0) transformed by glTranslatef(3.0, 5.0, 9.0)?

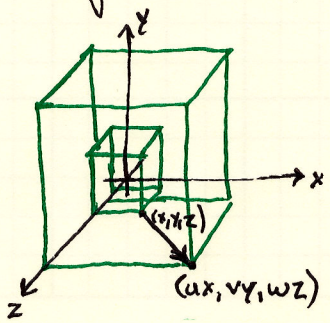
Ex: What is the OpenGL translation that takes (3.0, -1.0, 2.0) to (3.0, 5.0, 9.0)?

Scaling

- glScalef(u, v, w) ≡ map each point (x, y, z) of an object to the point (ux, vy, wz).
  - This has the effect of stretching objects
  - A scaling transform where one or more of the scaling factors is 0 is said to be **degenerate**

Ex: To what point is (-2.0, 3.0, 9.0) transformed by glScalef(3.0, 1.0, -8.0)?

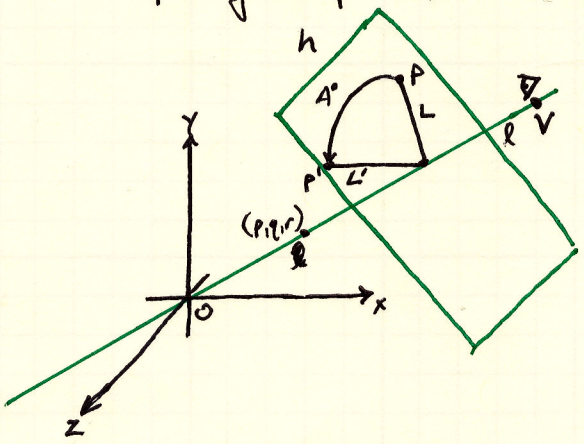
Ex: What is the OpenGL scaling that transforms (3.0, -1.0, 2.0) to (3.0, 5.0, 9.0)?



Modeling Transformations: Rotation

- The rotation command `glRotatef(A, p,q,r)` rotates an object about an axis from the origin point $O(0,0,0)$ to the point (p,q,r).
  - The amount of rotation is $A°$ CCW looking from (p,q,r) to the origin
  - The purpose of point (p,q,r) is to define axis l along the origin O through which the rotation occurs
  - Note: if $(p,q,r) = O$, then the rotation is invalid.
- Rotations are mapped as follows: Assum P does not lie on l
  1.) Drop perpendicularly from P to the point Q on l. Denote L as the line segment PQ.

  L lies on the plane h perpendicular to l through Q
  2.) Locate a viewer at V far enough along l, on the side of (p,q,r) as to be able to see h when looking towards O
  3.) Rotate the segment L about Q, on the plane h, an angle $A°$ CCW as measured by the viewer
  4.) If L' is the new position of L after rotation, then P is mapped to the corresponding endpoint P' of L'



Composing Modeling Transformations

- When we are composing multiple transforms we need to understand how they will be applied to an object.
- If we look at some examples and think through them it becomes apparent that they are applied in backwards order as they appear in code.
- Why is this?
  - First we need to understand that in OpenGL a vertex $V = (x, y, z)$ is represented as the vector

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

which is really the representation of V's homogeneous coordinate

  - Every modeling transformation t - regardless of it being a rotation, translation, or scaling - is represented as the 4x4 matrix:

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

  - Applying this transformation to V consists of multiplying V from the left by M:

$$t(V) = MV = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + a_{13}z + a_{14} \\ a_{21}x + a_{22}y + a_{23}z + a_{24} \\ a_{31}x + a_{32}y + a_{33}z + a_{34} \\ a_{41}x + a_{42}y + a_{43}z + a_{44} \end{bmatrix}$$

  - Example: $t_1 = glTranslatef(5.0, 0.0, 0.0) = M_1$

$$M_1 = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + 5.0 \\ y \\ z \\ 1 \end{bmatrix}$$

$$t_2 = glTranslatef(0.0, 10.0, 0.0) = M_2$$

$$M_2 = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 10.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

If we apply t_2 followed by t_1 to a vertex V, then V is mapped as follows

$$V \mapsto t_1(t_2(V)) = M_1(M_2V) = (M_1M_2)V$$

Composing Modeling Transformations (cont'd)

$$(M_1 M_2)V = \left(\begin{bmatrix} 1.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 10.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 1.0 & 0.0 & 10.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+5.0 \\ y+10.0 \\ z \\ 1 \end{bmatrix}$$

- If one applies successively the transformations $t_n, t_{n-1}, \cdots, t_1$ (in that order, t_n being first) to a vertex V, then it is mapped to

$$t_1(t_2(\cdots t_n(V)\cdots)) = M_1(M_2(\cdots(M_n V)\cdots)) = (M_1 M_2 \cdots M_n)V$$

  where matrix M_i corresponds to transformation t_i, $1 \le i \le n$

- Initially the modelview matrix is the identity matrix
- As each transformation is processed, this updates the modelview matrix by multiplication from the left
- An object drawing statement is processed by multiplying the object's vertices from the left by the current modelview matrix
- Thus, when an object is encountered all of its vertices are multiplied by the current modelview matrix
  - Thus all transformations encountered thus far are applied (in backwards order)
- Note: The Trick
  - Rotations happen about the origin, ~~thus in order to~~ and not about an object's center. Thus to rotate an object about its center, we must do the following
    1.) Translate the object to the origin
    2.) Rotate it about the origin
    3.) Translate it back

- Scale Transformation Matrix

`glScalef(2.0, 1.0, 3.0)`

$$\begin{bmatrix} 2.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

- Rotation Transformation Matrix

`glRotatef(90.0, 0.0, 1.0, 0.0)`

$$\begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ -1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Placing Multiple Objects

- Placing objects individually using transformations is quite straight-forward.
- Placing multiple objects requires understanding the relativistic nature of the application of OpenGL transformations
- for example

```
glTranslatef(0.0, 0.0, -15.0);
glTranslatef(5.0, 0.0, 0.0);
glutWireCube(5.0);
glTranslatef(0.0, 10.0, 0.0)
glutWireSphere(2.0, 10, 8);
```

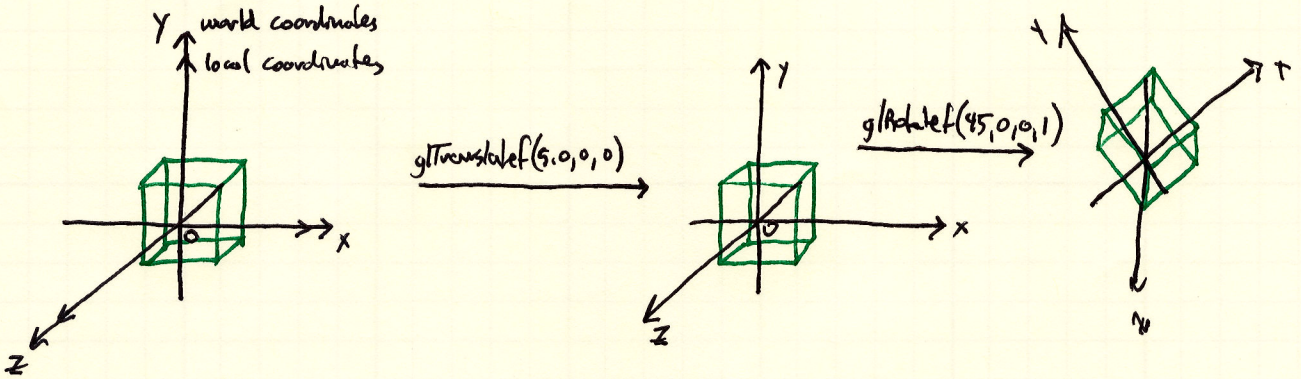
This positions a 5.0 size cube centered at (5.0, 0, -15.0)

and then positions the sphere centered at (5.0, 10.0, -15.0)

- These ? are apparent due to the understanding that we must work backwards from the last object to the first transformation
  - we also know that glut Objects always start at (0,0,0)
- Proposition: If object1 precedes object2 in code, then the location of object2 in the local coordinate system of object1 is determined by the transformations defined between the two and nothing else

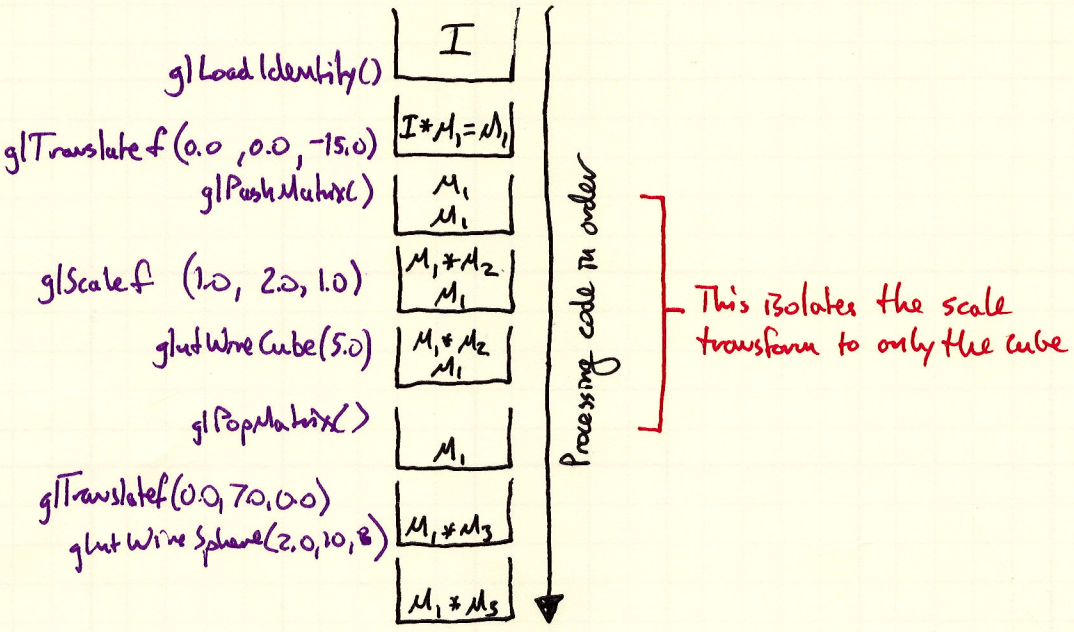
Local vs. World Coordinates

- Initially when an object is created its local coordinate system is aligned with and calibrated to the world coordinate system at the Origin.
- This connection is modified by transformations



ModelView Matrix Stack and Isolating Transformations

- The modelview matrix which is modified by modeling transformations is actually the top most matrix on a modelview matrix stack
- OpenGL maintains 3 separate matrix stacks:
  - Modelview
  - Projection
  - Texture
  - glMatrixMode(mode) ≡ determines which stack is currently active.
    - mode: one of GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE
- Because a series of transformations apply to all objects, the question comes up about how to prevent this?
  - The Answer: Isolate the set of transforms within a push-pop pair
    - glPushMatrix() ≡ creates a copy of the current matrix and places it on top of the current stack
    - glPopMatrix() ≡ deletes the top-most matrix on the stack
  - Example



- Best Practice ≡ Enclose all transforms in the drawing routine in one giant push-pop pair. This way when complete the modelview matrix stack is guaranteed to return to the identity matrix upon completion

Animation: Technicals

- Controlling Animation
  - Three Simple Methods
    - Interactively via mouse or keyboard input
    - Automatically by specifying an idle_function through glutIdleFunc(idle_func) which is called whenever no OpenGL event is pending
    - Semi-automatically, by specifying timer_function through glutTimerFunc(period, timer_func, value), where period is the period in milliseconds after the call, and value is the value passed to timer_func
  - The speed of animation, or, frame rate, is the rate at which the screen is redrawn
    - Cannot be increased arbitrarily because redrawing the scene takes some minimum time. This depends on scene complexity and GPU speed.
    - Frame rate cannot exceed the monitor's refresh rate which is the rate at which it can update all the pixels on the screen.
      - If the refresh rate is n Hz, then there can be n refreshes per second, then n is the maximum achievable fps (frames per second) as well.
  - Note: The number of frames drawn per second by the GPU to the color buffer. That is, the number of completions of drawScene() may not equal, exactly, the number of frames drawn to the screen per second → The true fps.
  - In this class our exclusive method for animation will be via glutTimerFunc() using time elapsed since last frame to guide the animation
  - You can calculate the time difference between frames by calling glutGet(GLUT_ELAPSED_TIME), which returns the milliseconds elapsed since glutInit() was called.

For Next Time

- Read Ch. 4 Sections 5-9
- Finish HW 1 and Demo it to me or TA
- Come to Class
- Review prior lectures