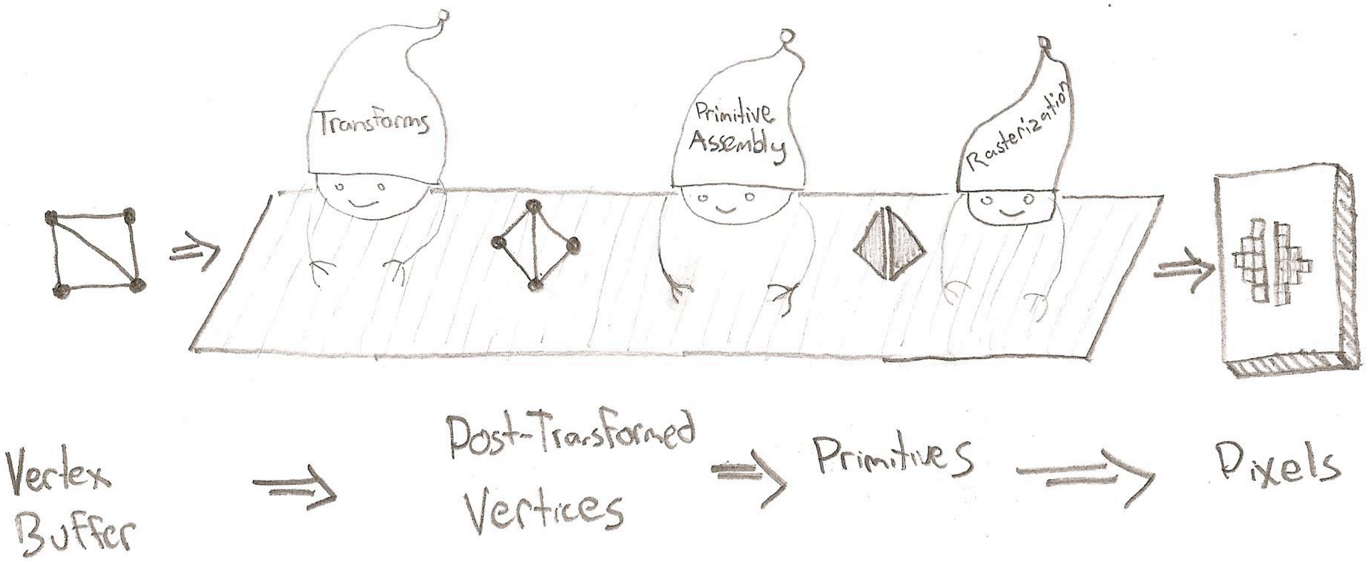
OpenGL (under the hood): Matrix Stacks



O'Reilly's "iPhone 3D Programming"

There's a pizza place near where I live that sells only slices. In the back you can see a guy tossing a triangle in the air.

--Stephen Wright, Comedian

OpenGL Matrix Types

- Matrices in graphics – purpose:
  - Geometric Transformations
  - Normalizing/Viewing Transformation
  - Textures/Pixmaps

- Correspondingly, 3 OpenGL matrix “categories”:
  - GL_MODELVIEW
  - GL_PROJECTION
  - GL_TEXTURE

  Note: viewport-mapping transformation handled separately through glViewport

- OpenGL matrix:
  - a 4 x 4 matrix of single- or double-precision floating-point values stored in column-major order. That is, the matrix is stored as follows:

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

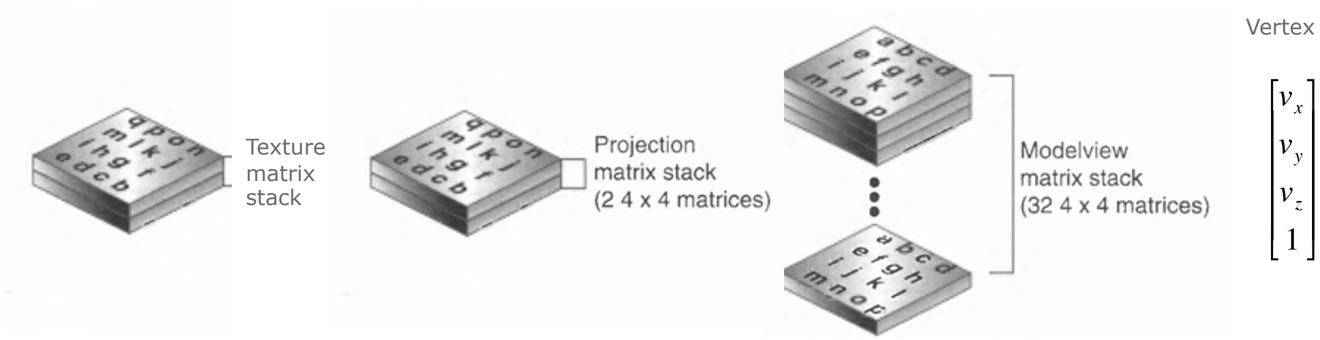
In C, can define the matrix as:

GLfloat my_matrix[4][4];

GLdouble my_dbl_matrix[4][4];

Matrix Stacks

- OpenGL maintains 3 stacks of matrices, one stack for each matrix type
  - to specify which matrix stack to work with, use glMatrixMode(<matrix_type>)
  - by convention, the default mode is GL_MODELVIEW (most commonly used)


Texture matrix stack
Projection matrix stack (2 4 x 4 matrices)
Modelview matrix stack (32 4 x 4 matrices)
Vertex
$\begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$

http://what-when-how.com/opengl-programming-guide/

- Each stack top is *automatically* applied to *every* vertex
  - think analogy with drawing attributes (glColor3f)

- Follows automated transformation pipeline
  - apply top of each stack, pizza-pipeline style; inflexible but makes gfx cards super-fast
  - Note: OpenGL transformations do not alter the state of the object, only their rendering!

"What if I do *not* want the current transformation to be applied to some object?"

- Answer: "Tough luck".
- No exceptions other than commands acting directly on the viewport
- To avoid application of the current transformation on an object, need to:
  - load identity matrix on top of stack
  - do your drawing
  - pop the stack
- Or define your own transformations and never load anything on the stack
  - slower if gazillion verts using same transform

General Stack Ops

Once matrix mode is set, we can perform various operations on the stack:

- glLoadIdentity() – sets current matrix to the identity matrix
- glLoadMatrix*(M) – loads (copies) a given matrix M over the current matrix
  - * can be either 'f' or 'd', depending on the type of M
- glMultMatrix*(M) – replaces the current matrix CTM with the result of CTM*M
  - * can be either 'f' or 'd', depending on the type of M
- glPushMatrix() – pushes a copy of the current matrix on top of the stack (thus stack has now two copies of the top matrix)
- glPopMatrix() – pops the current matrix off the stack

ModelView-Stack Specific Ops

- Translate
  - `glTranslatef(dx,dy,dz);`
  - Replace stacktop M by M*T
- Scale
  - `glScalef(sx,sy,sz);`
  - Replace stacktop M by M*S
- Rotate
  - `glRotatef(angle,lx,ly,lz);`
  - Replace stacktop M by M*R,

    where $(lx,ly,lz,0)^T$ defines the rotational axis: 1, 0, 0 is the X axis, 0,1,0 is Y etc.
- Transformation order matters: note that stack transformations are multiplied to the *right!!!*
  - *what does this mean re: transformation order?*
- These functions are deprecated in newer versions of OpenGL – so don't count on them

Example: What Happens If...?

```
my_display() {
  … // usual init stuff
  glTranslatef(1,3,0);
  glScalef(0.5,0.5,0.5);

  make_cube(); //see example code
  glRotatef(30,0,0,1);
  glutSwapbuffers();
}
```

Where Does My Camera Go?

- We know that the world-to-film transform can be broken up into component matrices (M_{pp}, S, M_{rot}, T_{trans})

- The (T, M) matrices are responsible for translating and rotating the world s.t. the viewer is positioned at the origin and looking down the –Z axis. Let's call their concatenation the **View** matrix (think "rigid camera")

- the (S, and optional M_{pp}) matrices are responsible for projecting the world onto the film plane and performing a homogeneous divide to create perspective. Let's call their concatenation the **Projection** matrix (think "lens of camera")

- **View** goes on the Model**View** stack; **Projection** goes on the Projection stack.

Parallel Camera in OpenGL

- Align camera coordinate system (u, v, w) with canonical coordinate system (x, y, z)
  - transfo built automatically by calling **gluLookAt**

```
//…assuming: glMatrixMode(GL_MODELVIEW)
glLoadIdentity();
gluLookAt(posX, posY, posZ, lookAtX, lookAtY, lookAtZ,
    upX, upY, upZ);
//… all other ModelView transformations follow this
```

Note!!: Here *lookAt* is the point we're looking at, not a vector

But *up* is a vector, nevertheless. Sigh.

- Squeeze camera view volume into canonical view volume, clip and project
  - specify viewing volume and projection type: **glOrtho**

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
//if parallel
glOrtho(left, right, bottom, top, near, far);
glMatrixMode(GL_MODELVIEW);
```

Perspective Camera in OpenGL

Viewing process separated in two steps

- Align camera coordinate system (u, n, v) with canonical coordinate system (x, y, z)
  - transfo built automatically by calling gluLookAt

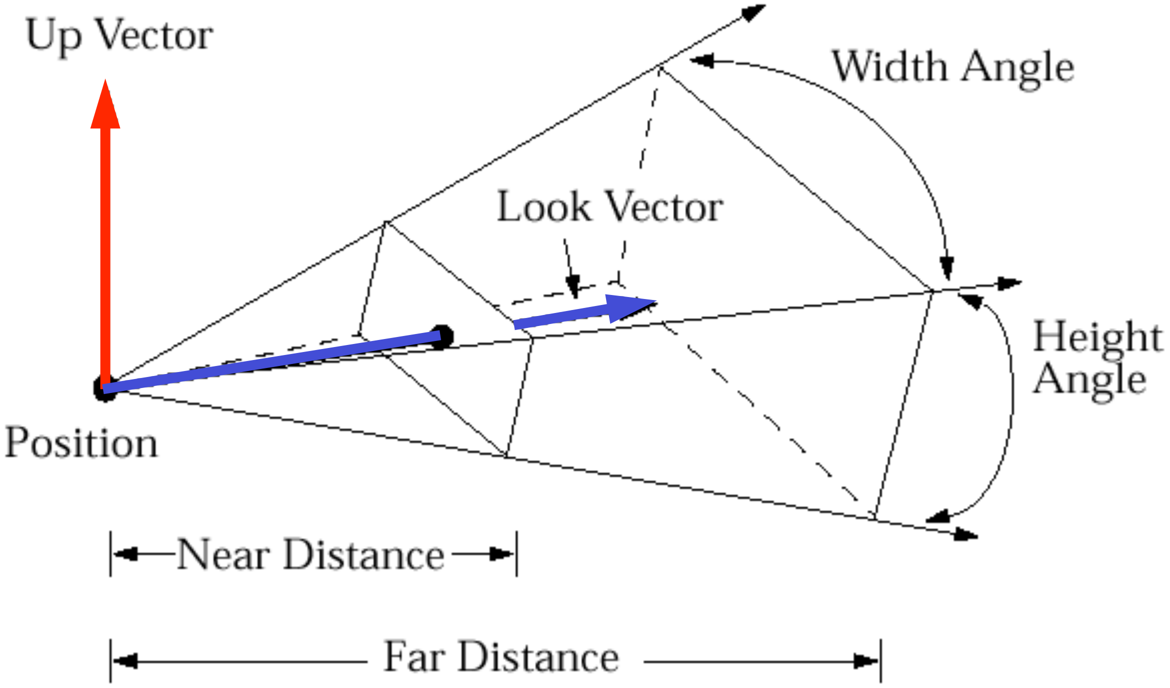
```
//…assuming: glMatrixMode(GL_MODELVIEW)
glLoadIdentity();
gluLookAt(posX, posY, posZ, lookAtX, lookAtY, lookAtZ,
    upX, upY, upZ);
//… all other ModelView transformations follow this
```

- Squeeze camera view volume into canonical view volume, clip and project
  - specify viewing volume and projection type

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
//if perspective
gluPerspective(fovy, aspect, near, far);
glMatrixMode(GL_MODELVIEW);
```

gluPerspective

- gluPerspective(fovy, aspect, near, far);
  - fovy – field of view (angle) in the y direction
  - aspect ratio – width/height


Up Vector
Width Angle
Look Vector
Height Angle
Position
Near Distance
Far Distance