Scientific Visualization

OpenGL

- Matrices
- MODELVIEW stack
- Conventions
- Simple use
- Drawing geometric objects
- Examples

Intro to OpenGL

- OpenGL operates as an infinite loop
  - Put things in the scene (points, colored lines, textured polys)
  - Describe the camera (location, orientation, field of view)
  - Listen for keyboard events
  - Render – draw the scene

Intro to OpenGL

- OpenGL has a “state”
  - There are a lot of ways your OpenGL program can be configured
  - The current configuration is stored in OpenGL’s state
  - Be aware that OpenGL commands affect the program’s state rather than redirect its logical execution

Intro to OpenGL

- OpenGL uses matrices
  - Matrix describes camera type
  - Matrix describes current configuration of the 3D space
    - Explanation…

Intro to OpenGL

- OpenGL coordinate system
  - right-handed
    - Hold out your right hand and hold your thumb, index, and middle fingers orthogonal to one another
    - Call your thumb the x-axis, index = y-axis, and middle = axis
    - This is the OpenGL coordinate system
  - The camera defaults to look down negative z-axis

Intro to OpenGL

- So…
  - X-axis = thumb = 1, 0, 0
  - Y-axis = index = 0, 1, 0
  - Z-axis = middle = 0, 0, 1

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  - Camera defaults to look down negative z-axis
  - Let's say we want it to look down x-axis

Intro to OpenGL

- Coordinate system transformation so camera looks down x-axis
  - If x-axis → negative z-axis
    - x → -z
    - y → y
    - z → x

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Intro to OpenGL

- The a→ i matrix defines the transformation

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

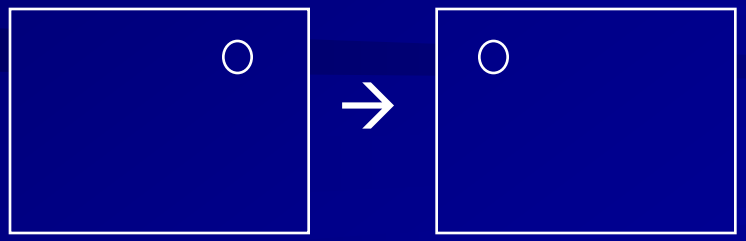
- Why store the transformation matrix and not the final desired matrix?

Intro to OpenGL

- The transformation will be applied to many points
  - If the following transformation moves the axes

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

  - The same transformation moves vertices
    - Example: (1, 1, -1) → (-1, 1, -1)



$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i' \\ j' \\ k' \end{bmatrix}$$

Intro to OpenGL

- This important matrix is stored as the GL_MODELVIEW matrix
  - The MODELVIEW matrix is so important OpenGL maintains a stack of these matrices
  - You have control of this stack with the glPushMatrix and glPopMatrix commands
  - The matrix is actually 4x4.
- Note OpenGL preserves a similar matrix to describe the camera type and this is called the GL_PROJECTION

Modeling Transformations

- glTranslate (x, y, z)
  - Post-multiplies the current matrix by a matrix that moves the object by the given x-, y-, and z-values
- glRotate (theta, x, y, z)
  - Post-multiplies the current matrix by a matrix that rotates the object in a counterclockwise direction about the ray from the origin through the point (x, y, z)

Modeling Transformations

- glScale (x, y, z)
  - Post-multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes

Modeling Transformations

- It is important that you understand the order in which OpenGL concatenates matrices

```
glMatrixMode
(GL_MODELVIEW);
glLoadIdentity();
glMultMatrix(N);
glMultMatrix(M);
glMultMatrix(L);
glBegin(POINTS);
glVertex3f(v);
glEnd();
```

Modelview matrix successively contains:
I(dentity), N, NM, NML

The transformed vertex is:
NMLv = N(M(Lv))

Manipulating Matrix Stacks

- Observation: Certain model transformations are shared among many models
- We want to avoid continuously reloading the same sequence of transformations
- **`glPushMatrix ( )`**
  - push all matrices in current stack down one level and copy topmost matrix of stack
- **`glPopMatrix ( )`**
  - pop the top matrix off the stack

Matrix Manipulation - Example

- Drawing a car with wheels and lugnuts

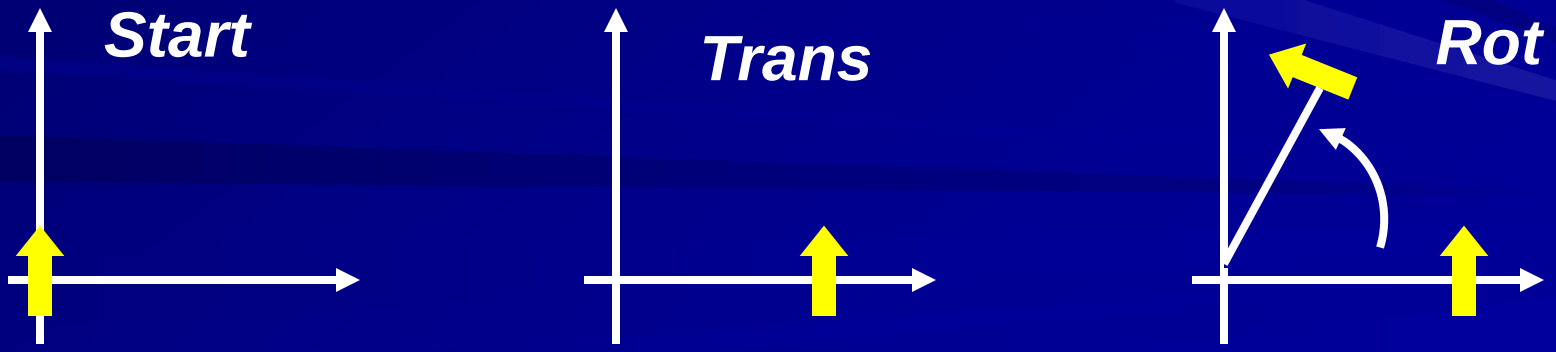
```
draw_wheel( );
for (j=0; j<5; j++) {
      glPushMatrix ();
            glRotatef(72.0*j, 0.0, 0.0, 1.0);
            glTranslatef (3.0, 0.0, 0.0);
            draw_bolt ( );
      glPopMatrix ( );
}
```

Matrix Manipulation – Example
Grand, fixed coordinate system

```
draw_wheel( );
for (j=0; j<5; j++) {
      glPushMatrix ();
            glRotatef(72.0*j, 0.0, 0.0, 1.0);
            glTranslatef (3.0, 0.0, 0.0);
            draw_bolt ( );
      glPopMatrix ( );
```

R
RT
RTv

Global – Bottom Up

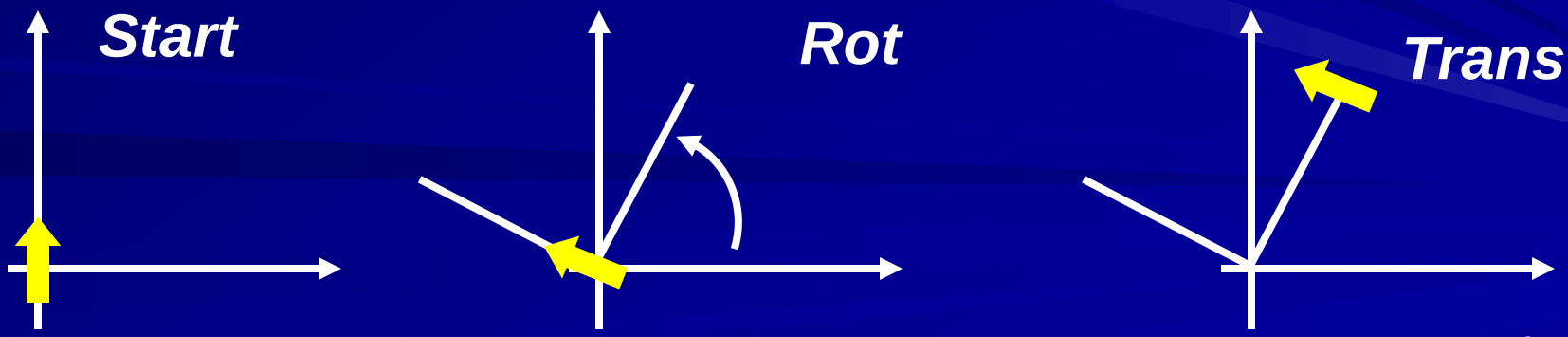


Matrix Manipulation – Example
Local coordinate system

```
draw_wheel( );
for (j=0; j<5; j++) {
      glPushMatrix ();
            glRotatef(72.0*j, 0.0, 0.0, 1.0);
            glTranslatef (3.0, 0.0, 0.0);
            draw_bolt ( );
      glPopMatrix ( );
```

R
RT
RTv

Local – Top Down



OpenGL: Conventions

- Functions in OpenGL start with **`gl`**
  - Most functions just **`gl`** (e.g., **`glColor()`**)
  - Functions starting with **`glu`** are utility functions (e.g., **`gluLookAt()`**)
  - Functions starting with **`glx`** are for interfacing with the X Windows system (e.g., in gfx.c)

OpenGL: Conventions

- Function names indicate argument type and number
  - Functions ending with **`f`** take floats
  - Functions ending with **`i`** take ints
  - Functions ending with **`b`** take bytes
  - Functions ending with **`ub`** take unsigned bytes
  - Functions that end with **`v`** take an array.
- Examples
  - **`glColor3f()`** takes 3 floats
  - **`glColor4fv()`** takes an array of 4 floats

OpenGL: Conventions

- Variables written in CAPITAL letters
  - Example: GLUT_SINGLE, GLUT_RGB
  - usually constants
  - use the bitwise or command (x | y) to combine constants

OpenGL: Simple Use

- Open a window and attach OpenGL to it
- Set projection parameters (e.g., field of view)
- Setup lighting, if any
- Main rendering loop
  - Set camera pose with **`gluLookAt()`**
    - Camera position specified in world coordinates
  - Render polygons of model
    - Simplest case: vertices of polygons in world coordinates

OpenGL: Simple Use

- *Open a window and attach OpenGL to it*
  - glutCreateWindow() or FLTK window method

OpenGL: Perspective Projection

- *Set projection parameters (e.g., field of view)*
- Typically, we use a *perspective projection*
  - Distant objects appear smaller than near objects
  - Vanishing point at center of screen
  - Defined by a *view frustum* (draw it)
- Other projections: *orthographic*, *isometric*

Setting up Camera

- **glMatrixMode(GL_MODELVIEW);**
- **glLoadIdentity();**
- **gluLookAt(eyeX, eyeY, eyeZ,**
  lookX, lookY, lookZ,
  upX, upY, upZ);
  - eye[XYZ]: camera position in world coordinates
  - look[XYZ]: a point centered in camera's view
  - up[XYZ]: a *vector* defining the camera's vertical
- **Creates a matrix that transforms points in world coordinates to *camera coordinates***
  - Camera at origin
  - Looking down -Z axis
  - Up vector aligned with Y axis

OpenGL: Perspective Projection

- In OpenGL:
  - Projections implemented by *projection matrix*
  - `gluPerspective()` creates a perspective projection matrix:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity(); //load an identity matrix
gluPerspective(vfovy, aspect, near, far);
```

- Parameters to `gluPerspective()`:
  - `vfovy`: field of view angle
  - `aspect`: field of view width/height
  - `near`, `far`: distance to near & far clipping planes

OpenGL: Lighting

- *Setup lighting, if any*
- Simplest option: change the *current color* between polygons or vertices
  - glColor() sets the current color
- Or OpenGL provides a simple lighting model:
  - Set parameters for light(s)
    - Intensity, position, direction & falloff (if applicable)
  - Set *material* parameters to describe how light reflects from the surface
- Won't go into details now; check the red book if interested

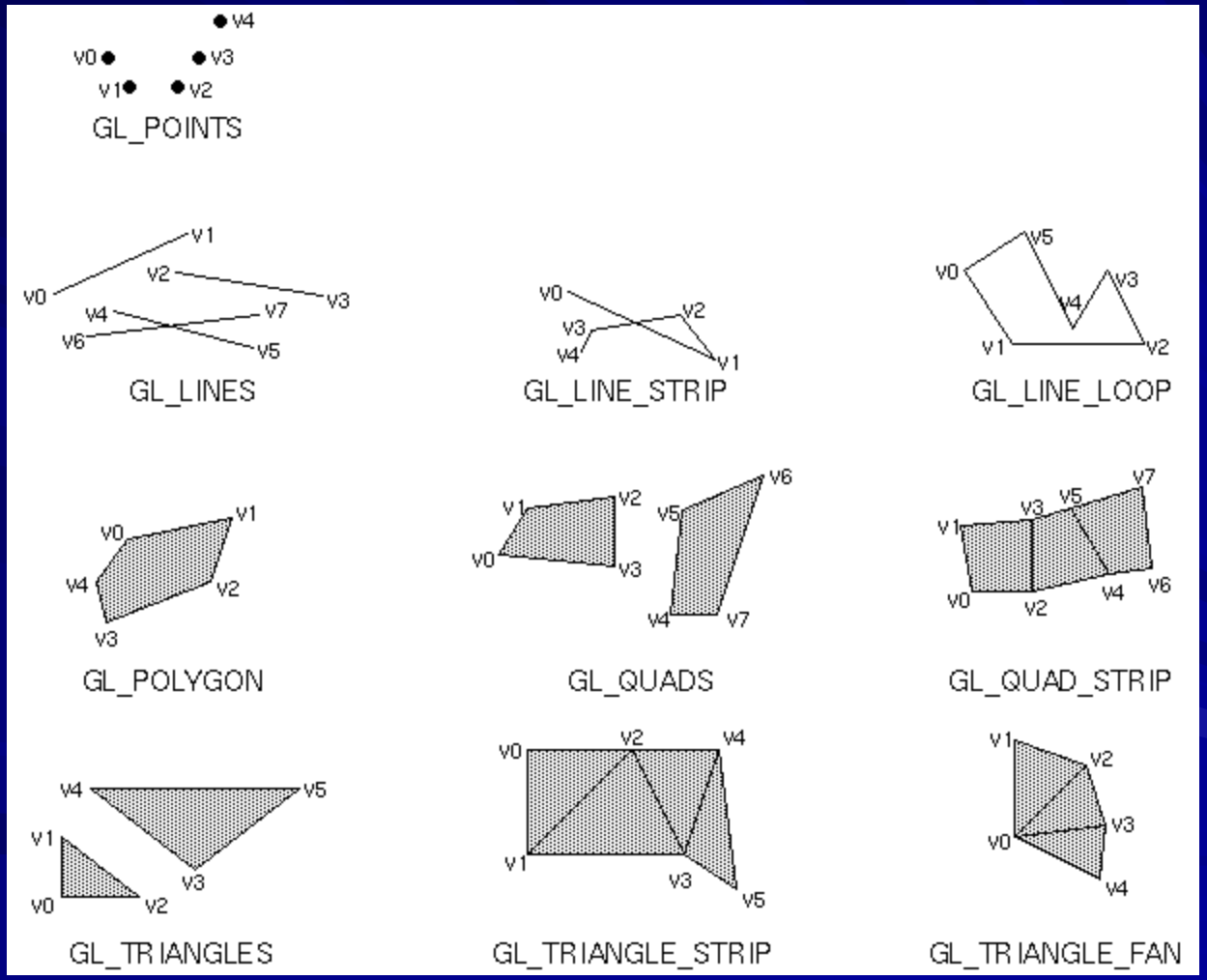
OpenGL: Specifying Geometry

- Geometry in OpenGL consists of a list of vertices in between calls to **`glBegin()`** and **`glEnd()`**
  - A simple example: telling GL to render a triangle

```
glBegin(GL_POLYGON);
  glVertex3f(x1, y1, z1);
  glVertex3f(x2, y2, z2);
  glVertex3f(x3, y3, z3);
glEnd();
```

  - Usage: **`glBegin(`***geomtype***`)`** where geomtype is:
    - Points, lines, polygons, triangles, quadrilaterals, etc...

Primitive Types



Points

- Object of zero dimension (infinitely small)
- Specified by a set of floating-point numbers (coordinates) called a **vertex**
- Displayed as a single pixel on screen
- void **glPointSize**(Glfloat *size*);
  - Sets the size of a rendered point in pixels

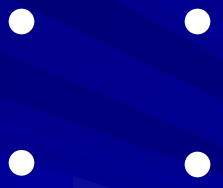
Specifying Vertices

- void **glVertex**{234}{sifd}[v](TYPE *coords*);
  - Specifies a vertex for use in describing a geometric object
    - **glVertex2s**(2,4);
    - **glVertex4f**(2.3, 1.0, -2.2, 2.0);
    - GLdouble dvect[3] = {5.0, 9.0, 4.0);
    - **glVertex3dv**(dvect);
- OpenGL works in homogeneous coordinates
    - **vertex:: (*x, y, z, w*)**
    - ***w* = 1 for default**

Displaying Vertices

- Bracket a set of vertices between a call to **glBegin**() and a call to **glEnd**() pair
  - The argument GL_POINTS passed to glBegin() means drawing vertices in the form of the points
    - **glBegin**(GL_POINTS);

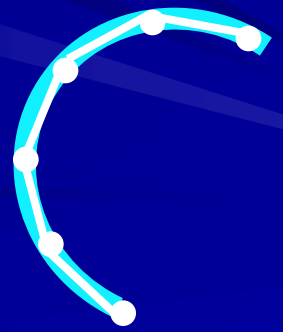
      glVertex2f(0.0, 0.0);
      glVertex2f(4.0, 0.0);
      glVertex2f(4.0, 4.0);
      glVertex2f(0.0, 4.0);
    - **glEnd();**



  - Other drawing options for vertex-data list
    - Lines (GL_LINES)
    - Polygon (GL_POLYGON)

Lines

- The term *line* refers to a *line segment*
- Specified by the vertices at their endpoints
- Displayed solid and one pixel wide
- Smooth curves from line segments

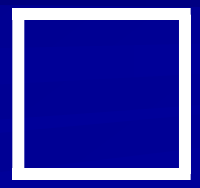


Drawing Lines

- To draw a vertex-data list as lines
  - **glBegin**(GL_LINES);
  - **glVertex2f**(0.0, 0.0);
  - **glVertex2f**(4.0, 0.0);
  - **glVertex2f**(4.0, 4.0);
  - **glVertex2f**(0.0, 4.0);
  - **glEnd();**

- GL_LINE_STRIP
  - A series of connected lines

- GL_LINE_LOOP
  - A closed loop



Wide and Stippled Lines

- void **glLineWidth**(GLfloat *width*);
  - Sets the width in pixels for rendered lines
- void **glLineStipple**(GLint *factor*, GLshort *pattern*);
  - Sets the current stippling pattern (dashed or dotted) for lines
  - *Pattern* is a 16-bit series of 0s and 1s
    - 1 means one pixel drawing, and 0 not drawing
  - *Factor* stretches the pattern multiplying each bit
  - Trun on and off stippling
    - **glEnable(**GL_LINE_STIPPLE**)**
    - **glDisable(**GL_LINE_STIPPLE**)**

Example of Stippled Lines

- **glLineStipple**(1, 0x3F07);
  - *Pattern* **0x3F07** translates to **0011111100000111**
  - Line is drawn with 3 pixels on, 5 off, 6 on, and 2 off

- **glLineStipple**(2, 0x3F07);

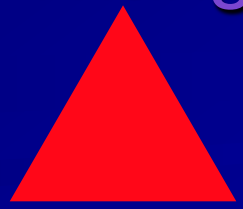
  Factor is 2

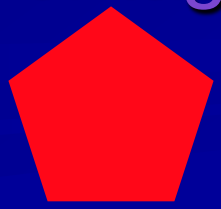
  Line is drawn with 6 pixels on, 10 off, 12 on, and 4 o

Polygon

- Areas enclosed by singled closed loops of line segments
- Specified by vertices at the corners
- Displayed as solid with the pixels in the interior filled in

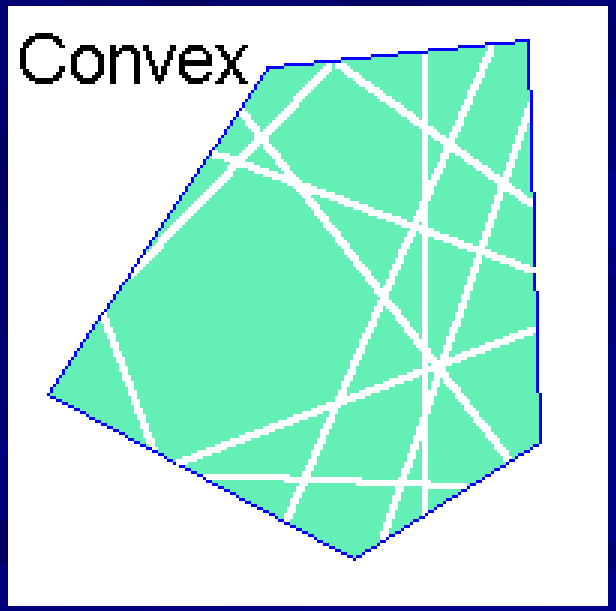
  Examples: Triangle and Pentagon

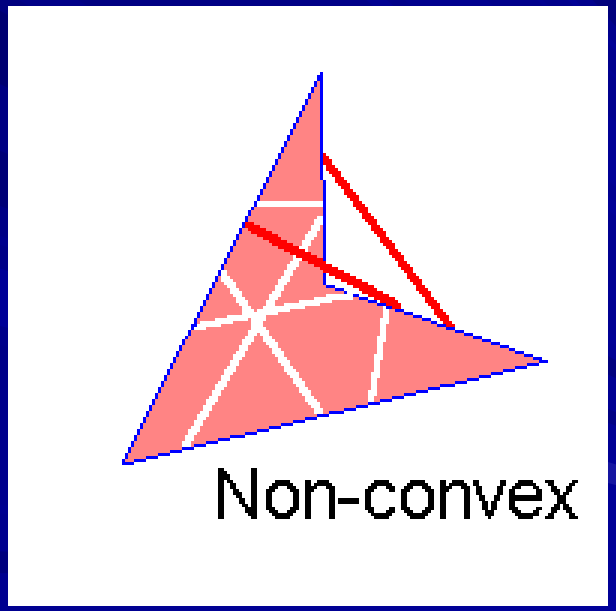




GL_POLYGON

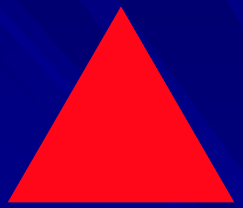
- List of vertices defines polygon edges
- Polygon must be convex



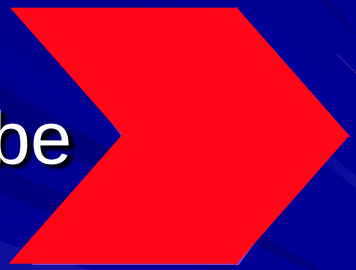


Polygon Tessellation

- Simple and convex polygon
  - Triangle
  - Any three points always lie on a plane



- Polygon tessellation
  - Nonsimple or nonconvex polygons can be represented in the form of triangles



- Curved surfaces can be approximated by polygons

Non-planar Polygons

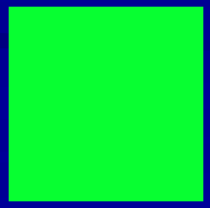
- Imagine polygon with non-planar vertices
- Some perspectives will be rendered as concave polygons
- These concave polygons may not rasterize correctly

Drawing Polygon

- Draw a vertex-data list as a polygon
  - **glBegin**(GL_POLYGON);
  - **glVertex2f**(0.0, 0.0);
  - **glVertex2f**(4.0, 0.0);
  - **glVertex2f**(4.0, 4.0);
  - **glVertex2f**(0.0, 4.0);
  - **glEnd();**

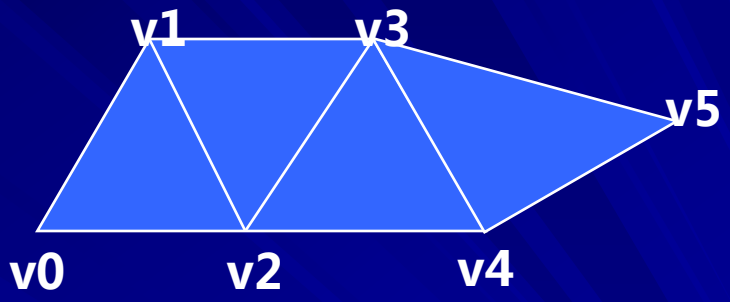
- GL_TRIANGLES
  - Draws first three vertices as a triangle

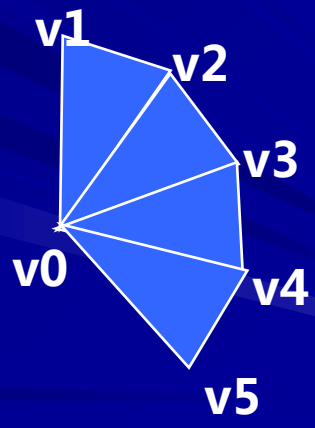
- GL_QUADS
  - Quadilateral is a four-sided polygon



Drawing Polygons

- GL_TRIANGLE_STRIP
  - Draws a series of triangles using vertices in the order
    - v0,v1,v2; v2,v1,v3
    - v2,v3,v4; v4,v3,v5
  - All triangles are drawn with the same orientation (clockwise order)
- GL_TRIANGLE_FAN
  - One vertex is in common to all triangles
  - Clockwise orientation
- GL_QUAD_STRIP
  - Draws a series of quadrilaterals





Polygons as Points and Outlines

- void **glPolygonMode**(GLenum *face*, Glenum *mode*);
  - Controls the drawing mode for a polygon's front and back faces
    - glPolygonMode(GL_FRONT, GL_FILL);
    - glPolygonMode(GL_BACK, GL_LINE);
    - glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
- By convention, polygons whose vertices appear in counterclockwise order are front-facing
    - GL_CCW

Deciding Front- or Back Facing

- Decision based the sign of the polygon's area, ***a*** computed in window coordinates

$$a = \frac{1}{2}\sum_{i=0}^{n-1}\left[x_i y_{i\oplus 1} - x_{i\oplus 1} y_i\right]$$

- For GL_CCW, if a>0 means the polygon be front-facing, then a<0 means the back-facing
- For GL_CW, if a<0 for front-facing, then a>0 for back-facing

Reversing and Culling Polygons

- void **glFrontFace**(GLenum *mode*);
  - Controls how front-facing polygons are determined
  - Default mode is GL_CCW (vertices in counterclockwise order)
  - Needs to be enabled: glEnable(GL_CULL_FACE)
- void **glCullFace**(GLenum *mode*);
  - Indicates which polygons (back-facing or front-facing) should be discarded (culled)
  - Needs to be enabled: glEnable(GL_CULL_FACE)

Stippling Polygons

- Void **glPolygonStipple**(const GLbyte **mask*);
  - Defines the current stipple pattern for the filled polygons
  - The argument is a pointer to a 32x32 bitmap (a mask of 0s and 1s)

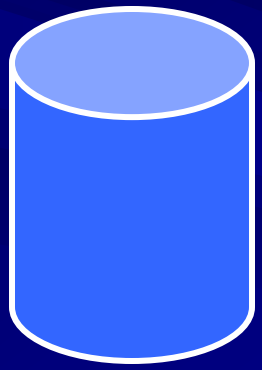
- Needs to be enabled and disabled
  - glEnable(GL_POLYGON_STIPPLE);
  - glDisable(GL_POLYGON_STIPPLE);

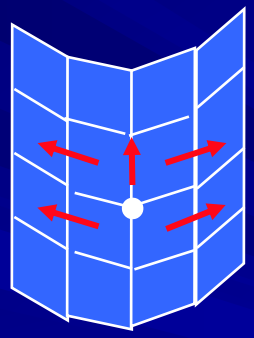
Normal Vectors

- Points in a direction that's perpendicular to a surface
  - The normal vectors are used in lighting calculations
- void **glNormal3**(bsidf)(*TYPE nx, TYPE ny, TYPE nz*);
  - Sets the current normal vector as specified by the arguments
- void **glNormal3**(bsidf)**v**(const *TYPE *v*);
  - Vector version supplying a single array ***v*** of three element

Finding Normal Vector

- Surfaces described with polygonal data
  - Calculate normal vectors for each polygonal facet
  - Average these normals for neighboring facets
  - Use the averaged normal for the vertex that the neighboring facets have in common

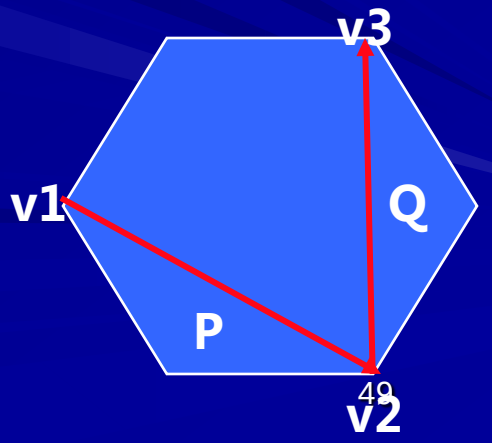




- Using normal vectors in lighting model to make surface appear smooth rather than facet

Finding Normal Vector

- Make two vectors from any three vertices v1, v2 and v3
  - *P = v1 - v2; Q = v2 - v3*
- Cross product of these vectors is perpendicular to polygonal surface
  - *N = P x Q = [Px Py Pz] x [Qx Qy Qz]*
  - *= [PyQz-QyPz) (QxPz-PxQz) (PxQy-QxPy]*
  - *= [Nx Ny Nz]*
- Normalize the vector
  - n = [*nx ny nz*] = [*Nx/L Ny/L Nz/L*]
  - where L is length of the vector [*Nx Ny Nz*]



Building Polygonal Models of Surfaces

- You can approximate smooth surfaces by polygons

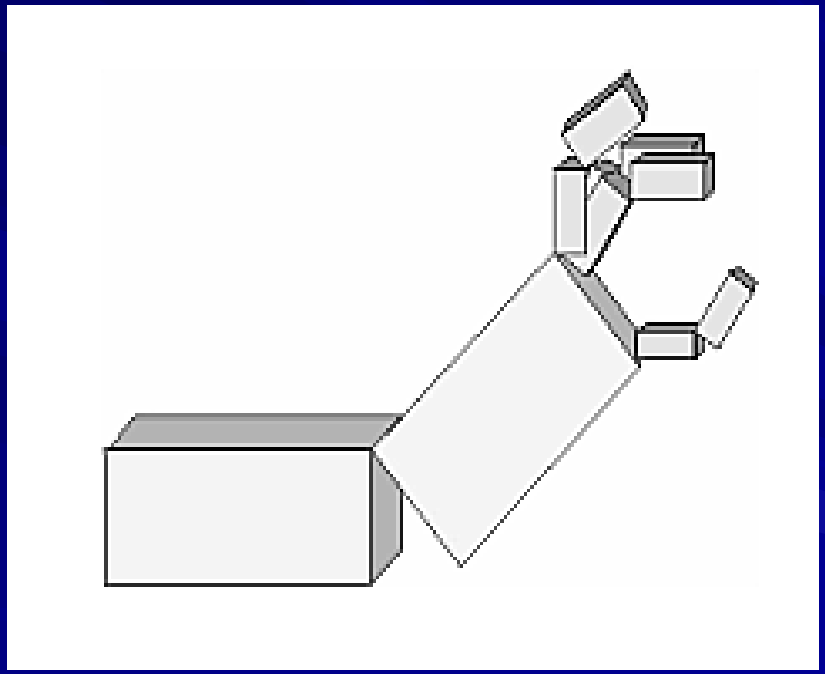
- Important points
  - Polygon orientations consistency (all clockwise or all anticlockwise)
  - Caution at non-triangular polygons
  - Trade-off between display speed and image quality

Double Buffering

- Avoids displaying partially rendered frame buffer
- OpenGL generates one raster image while another raster image is displayed on monitor
- glxSwapBuffers (Display *dpy, Window, w)
- glutSwapBuffers (void)

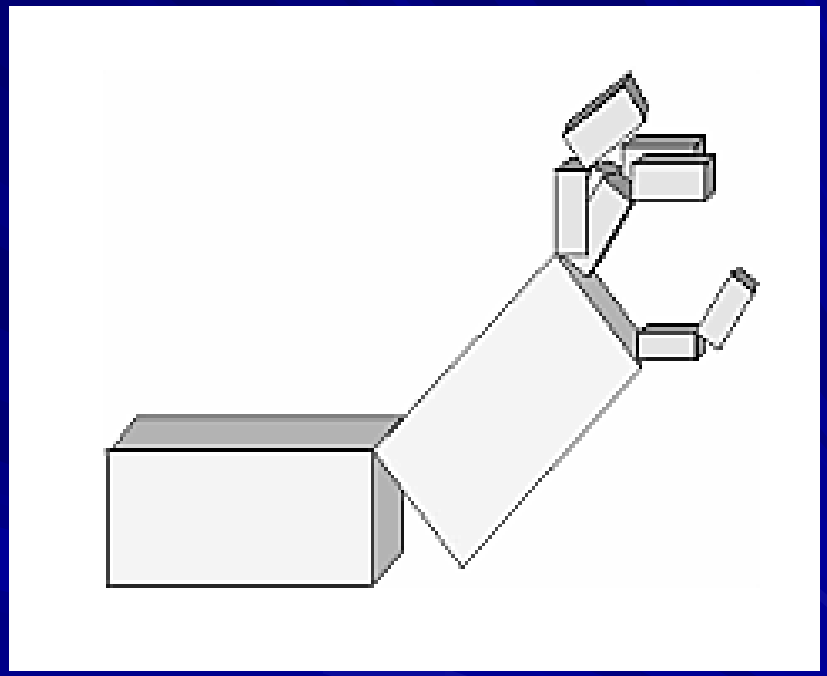
Learn OpenGL by example

- robot.c from the OpenGL Programming Guide



Learn OpenGL by example

- Two bodies
  - Upper arm
  - Lower arm
- Major tasks
  - Position
  - Orientation



Learn OpenGL by example

- Both bodies originally at origin

Learn OpenGL by example

- Headers
  - #include <GL/gl.h>
  - #include <GL/glu.h>
  - #include <GL/glut.h>

Learn OpenGL by example

```
int main(int argc, char** argv) {
        glutInit(&argc, argv);
        glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowSize (500, 500);
        glutInitWindowPosition (100, 100);
        glutCreateWindow (argv[0]);
        init ();
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutKeyboardFunc(keyboard);
        glutMainLoop();
        return 0;  }
```

Learn OpenGL by example

```
void init(void) {
        glClearColor (0.0, 0.0, 0.0, 0.0);
        glShadeModel (GL_FLAT);

}
```

Learn OpenGL by example

```
void display(void){
        glClear (GL_COLOR_BUFFER_BIT);
        glPushMatrix();
                glTranslatef (-1.0, 0.0, 0.0);
                glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
                glTranslatef (1.0, 0.0, 0.0);
                glPushMatrix();
                        glScalef (2.0, 0.4, 1.0);
                        glutWireCube (1.0);
                glPopMatrix();
Continued…
```

Learn OpenGL by example

```
            glTranslatef (1.0, 0.0, 0.0);
            glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
            glTranslatef (1.0, 0.0, 0.0);
            glPushMatrix();
                  glScalef (2.0, 0.4, 1.0);
                  glutWireCube (1.0);
            glPopMatrix();
      glPopMatrix();
      glutSwapBuffers();
}
```