OpenGL
OpenGL

- What it is:
  - Software interface to graphics hardware
  - ~ 120 C-callable routines for 3D graphics
  - Hardware independent
  - When running with X (with GLX extension)
    - Client-server model
    - Network transparent
OpenGL

- What it is not:
  - Not a windowing system (no window creation)
  - Not a UI system (no keyboard and mouse routines)
  - Not a 3D modeling system (Open Inventor, VRML, Java3D, 3DMax, Blender, etc.)
OpenGL Functionality

- Simple geometric objects (e.g. lines, polygons, rectangles, etc.)
- Transformations, viewing, clipping
- Hidden line & hidden surface removal
- Color, lighting, texture
- Bitmaps, fonts, and images
- Immediate- & retained- mode graphics
- Etc.
- But no shadow, raytracing, radiosity
OpenGL Convention

- Functions:
  - prefix `gl` + capital first letter (e.g. `glClearColor`)

- Constants:
  - prefix `GL` + all capitals (e.g. `GL_COLOR_BUFFER_BIT`)
Many variations of the same functions

```c
void glClearColor[2,3,4][b,s,i,f,d,ub,us,ui](v)
```

- `[2,3,4]`: dimension
- `[b,s,i,f,d,ub,us,ui]`: data type
- `(v)`: optional pointer (vector) representation
OpenGL Basic Concepts

- OpenGL as a state machine
  - Graphics primitives going through a “pipeline” of rendering operations
  - OpenGL controls the state of the pipeline with many state variables (fg & bg colors, line thickness, texture pattern, eyes, lights, surface material, etc.)
  - Binary state: glEnable & glDisable
  - query: glGet[Boolean,Integer,Float,Double]v
Rendering Pipeline

graphics primitives → modeling transform → viewing transform → clipping → shading & texture

transform matrix → Eye, lookat, headup → Parallel or Perspective volume → material, lights, surface color

images on screen → viewport transform → images in Internal buffer → projection

viewport location
Points, Lines, Polygons

- Specified by a set of vertices
  - void glVertex[2,3,4][s,i,f,d](v) (TYPE coords)

- Polygons:
  - simple, convex, no holes

- Grouped together by glBegin() & glEnd()
  - glBegin(GL_POLYGON)
    - glVertex3f( … )
    - glVertex3f( … )
  - glEnd
Points, Lines, Polygons Details

- Points:
  - size: void glPointSize(GLfloat size)

- Lines:
  - width: void glLineWidth(GLfloat width)
  - stippled lines:
    - glEnable(GL_LINE_STIPPLE)
    - glLineStipple()
Points, Lines, Polygons Details (cont.)

- void glPolygonMode(face, mode)
  - face: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
  - mode: GL_POINT, GL_LINE, GL_FILL
  - default: both front and back as filled
Points, Lines, Polygons Details (cont.)

- face culling
  - void glEnable(GL_CULL_FACE)
  - void glCullFace(mode)
    - mode: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
    - outside: back-facing polygon not visible
    - inside: front-facing polygon not visible
Other Primitives

- Many
  - GL_TRIANGLES
  - GL_TRIANGLE_STRIP
  - GL_QUADS
  - GL_QUAD_STRIP
  - Etc.
Other Primitives

GL_TRIANGLE_STRIP
012, 213, 234, 435

GL_QUAD_STRIP
0132, 2354, 4576
Hint on Homework

- Only two primitives are needed
  - A cube (with GL_POLYGON) for modeling “straight” pieces
  - A flexible quad or triangular strip, allowing:
    - Any angular extent
    - Different width
    - Sampling rate
    - Etc.
Rendering Pipeline

- **Graphics primitives**
  - Transform matrix

- **Modeling transform**
  - Eye, lookat, headup

- **Viewing transform**
  - Parallel or Perspective volume

- **Clipping**
  - Material, lights, surface color

- **Shading & texture**

- **Projection**
  - Images on screen

- **Viewport transform**
  - Viewport location

- **Images in Internal buffer**
Geometric Transform

- Step 1: Modeling transform
  - A global “world” coordinate system where one constructs and manipulates models

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>glTranslate<a href="x,y,z">f,d</a></td>
<td>Translation in the x, y, or z direction</td>
</tr>
<tr>
<td>glRotate<a href="x,y,z">f,d</a></td>
<td>Rotation around the x, y, or z axis</td>
</tr>
<tr>
<td>glScale<a href="x,y,z">f,d</a></td>
<td>Scaling along the x, y, or z axis</td>
</tr>
</tbody>
</table>
Geometric Transform

- Step 2: Viewing transform
  - Select the eye pos, look-at dir, head-up dir, and view volume
**Geometric Transform**

- Step 3: Clipping
  - Remove primitives that are not in the view volume
Geometric Transform

- Step 4: Projection
  - Map from 3D into 2D
Geometric Transform

- Step 5: Viewport transform
  - Map 2D images onto screen
Transform in OpenGL

- OpenGL uses stacks to maintain transformation matrices (MODELVIEW stack is the most important)
- You can load, push and pop the stack
- The current transform is applied to all graphics primitive until it is changed
General Transform Commands

- Specify current matrix
  - void glMatrixMode(GLenum mode)
    - GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE

- Initialize current matrix
  - void glLoadIdentity(void)
  - void glLoadMatrix[f,d](const TYPE *m)
General Transform Commands (cont.)

- Concatenate current matrix
  - `void glMultMatrix(const TYPE *m)`
  - `C = CMv` (remember: GL uses a stack)

- Caveat: OpenGL matrices are stored in column major (this is different from C convention)

\[
M = \begin{bmatrix}
  m_1 & m_5 & m_9 & m_{13} \\
  m_2 & m_6 & m_{10} & m_{14} \\
  m_3 & m_7 & m_{11} & m_{15} \\
  m_4 & m_8 & m_{12} & m_{16}
\end{bmatrix}
\]

Best use utility functions `glTranslate`, `glRotate`, `glScale`
Transformations

- Translation
- Scaling

- Scaling is w.r.t the origin
- Apparent movement if object is not zero centered
Rotation

- Six DOFs (things you can specify)
  - Axis Location (3)
  - Axis Orientation (2)
  - Angle (1)

\[
(n_x, n_y, n_z), n_x^2 + n_y^2 + n_z^2 = 1
\]

\[(x, y, z)\]
Rotation in OpenGL

- Axis location
  - Always go through the origin
- Axis orientation
  - X, Y, or Z
- Angle of rotation
- Again, rotation is w.r.t the origin
- Apparent movement occurs if the object is not zero-centered
Rotation
Euler Angle Rotation

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix}
= 
\begin{bmatrix}
    \cos \theta & -\sin \theta & 0 & 0 \\
    \sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z
\end{bmatrix}
\]

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix}
= 
\begin{bmatrix}
    \cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z
\end{bmatrix}
\]

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z
\end{bmatrix}
\]
Step 1: Modeling Transform

\[ \text{glTranslate}[f,d](x,y,z) \]
\[ \text{glRotate}[f,d](\text{angle},x,y,z) \]
\[ \text{glScale}[f,d](x,y,z) \]

- Order is important

Rotate \textit{then} Translate \hspace{1cm} \text{Translate} \textit{then} Rotate
**Step 1: Modeling Transform (cont.)**

\[ \text{glMatrixMode(GL\_MODELVIEW)}; \]
\[ \text{glLoadIdentity();} \]
\[ \text{glMultiMatrixf(T);} \]
\[ \text{glMultiMatrixf(R);} \]
\[ \text{draw\_the\_object(v);} \]
\[ v' = ITRv \]

\[ \text{glMatrixMode(GL\_MODELVIEW)}; \]
\[ \text{glLoadIdentity();} \]
\[ \text{glMultiMatrixf(R);} \]
\[ \text{glMultiMatrixf(T);} \]
\[ \text{draw\_the\_object(v);} \]
\[ v' = IRTv \]
Modeling Transform

- Usually, think there is a global “world” coordinate system where
  - all objects are defined
  - rotation, translation, scaling of objects in the world system
  - order is reverse (backward)

- Can also be thought in a “local” coordinate system where
  - x, y, z are fixed w.r.t the reviewer
  - Coordinate system moves with the viewer
  - Order is forward
Two Different Views

- As a global system
  - object moves but coordinates stay the same
  - apply in the *reverse* order

- As a local system
  - object moves and coordinates move with it
  - applied in the *forward* order

*The code and OpenGL operation are identical!!!*

*The difference is in how you (human) interpret what happens!!!*

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(T_1);
glMultiMatrixf(T_2);
...
glMultiMatrixf(T_n);
draw_the_object(v);
v' = IT_1 T_2 T_n v
```

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(T_1);
glMultiMatrixf(T_2);
...
glMultiMatrixf(T_n);
draw_the_object(v);
v' = IT_1 T_2 T_n v
```
Global view
• Rotate object
• Then translate

Local view
• Translate object (and coordinate)
• Then rotate

```c
glLoadIdentity();
glMultiMatrixf(T);
glMultiMatrixf(R);
draw_the_object(v);
```
- Global view
- Translate object
- Then rotate

- Local view
- Rotate object (and coordinate)
- Then translate

```c
glLoadIdentity();
glMultiMatrixf(R);
glMultiMatrixf(T);
draw_the_object(v);
```
glLoadIdentity();
glRotate(0,0,90);
glTranslate(1,0,0);
glRotate(0,0,45);
glTranslate(1,0,0);
draw_the_object(v);
glLoadIdentity();
glRotate(0,0,90);
glTranslate(0,1,0);
glScale(2,0.5,1);
glTranslate(1,0,0);
draw_the_object(v);

Caveats: scale in local view may distort coordinate systems!!

A movement of 2 instead of 1!
```c
glLoadIdentity();
glTranslate(-4,0,0);
glScale(2,1,1);
glTranslate(1,0,0);
draw_the_object(v);
```
Hierarchical Transform

- Used very frequently for building complex objects in a modular manner
- Cf. subroutine calls
- Be able to push and pop transform matrices as needed
- OpenGL provides two stacks
  - at least 32 4x4 model view matrices
  - at least 2 4x4 projection matrices
- `glLoadMatrix()`, `glMultMatrix()`, `glTrans(rotate, scale, etc.)` affect top-most (current) one (the others are not affected!)
Hierarchical Transform (cont.)

- void glPushMatrix(void)
  - topmost matrix is *copied* (top and second-from-top)
- void glPopMatrix(void)
  - topmost is *gone* (second-from-top becomes top)

- Very important
  - For OpenGL beginner
  - Transformation ordering
  - Transformation grouping
Hierarchical Transform (cont.)

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(T);
draw_car_body();
glPushMatrix();
glTranslatef(T1);
draw_wheel();
glPopMatrix();
glPushMatrix();
glTranslatef(T2);
draw_wheel();
glPopMatrix();
```

```
... 
glPushMatrix();
```
Hierarchical Transform

- Try this!

- Translate
  - Sun

- Translate
  - Earth

- Translate
  - Earth about the sun
  - Revolve (year)

- Rotate (day)
  - Earth about itself

- Moon
The Sun

- Suppose the Sun is the center of the universe
  - No translation
  - Turn about itself

```cpp
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glPushMatrix();
glRotate(Rs);
draw_sun();
glPopMatrix();
```
The Earth

- **DOFs**
  - Translation away from the sun
  - Rotate about the sun
  - Rotate about itself

- **Which DOFs to isolate?**
- **Viewer centered or global view?**
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glPushMatrix();
  glRotate(Rs);
  draw_sun();
glPopMatrix();
glRotate(Rs->e);
glTranslatef(Ts->e);
glPushMatrix();
  glRoate(Re);
  draw_earth();
glPopMatrix();

Revolve (year)

Rotate (day)

Translate

Sun

Earth

Re
The Moon

- `glMatrixMode(GL_MODELVIEW);`
- `glLoadIdentity();`
- `glPushMatrix();`
- `glRotate(Rs);`  
  `draw_sun();`  
  `glPopMatrix();`
- `glRotate(Rs>e);`
- `glTranslatef(Ts>e);`
- `glPushMatrix();`
- `glRotate(Re);`  
  `draw_earth();`  
  `glPopMatrix();`
- `glRotate(Re>m);`
- `glTranslatef(Re>m);`
- `glPushMatrix();`
- `glRotate(Rm);`  
  `draw_moon();`  
  `glPopMatrix();`

Revolve (year)

Translate

Rotate (day)
HomeWork
Viewing Transform (Extrinsic)

- Default: eyes at origin, looking along -Z

- Important parameters:
  - where is the observer (camera)?
    - origin of the viewing system
  - What is the look-at direction?
    - -z direction
  - What is the head-up direction?
    - y direction
void gluLookAt (GLdouble eyex, eyey, eyez,
GLdouble centerx, centery, centerz,
GLdouble upx, upy, upz)
Viewing Transform (cont.)

- **eye and center**: local \( w(z) \) direction
- **up and local \( w(z) \)**: local \( v(y) \) direction
- **local \( v(y) \) and \( w(z) \)** directions: local \( u(x) \) direction
Viewing Transform

- Occur after modeling transform
- Usually involves only translation + rotation (no scaling)
- Best done by gluLookAt function
- Can also be done using glTranslate + glRotate (need to think about moving the camera instead of object in opposite way)
Viewing Transform - the hard way

- Use gluLookAt if possible
- Think in an object-centered way (forward)
- Camera is at the origin pointing along -z
- Rotate and translate objects to expose the right view
- `glMatrixMode`
- `glLoadIdentity`
- `glRotateZ(roll)`
- `glRotateY(pitch)`
- `glRotateX(heading)`
- `glTranslate(-px,-py,-pz)`
- Other modeling transform
- `glBegin … glEnd`
Shading and Texturing

- A BIG topic in graphics
- For photo realistic rendering
- Two aspects: geometry (location and orientation) and appearance (color, shading, texture)
- Here we concentrate on geometry only
Perspective Projection (Intrinsic)

- glMatrixMode(GL_PROJECTION);
- glLoadIdentity();
- void glFustrum(GLDouble left, right, bottom, top, near, far);

\[
\begin{align*}
\text{v(y)}
& \quad \text{far} \\
& \quad \text{near} \\
\text{left} & \quad \text{top} \\
\text{bottom} & \quad \text{right} \\
& \quad \text{(left, bottom, -near)} \\
& \quad \text{(left, top, -near)} \\
& \quad \text{(right, top, -near)} \\
& \quad \text{(right, bottom, -near)} \\
\end{align*}
\]
Perspective Projection (cont.)

- void gluPerspective(GLdouble fovy, aspect, near, far) -- for symmetric view volume

\[
\begin{align*}
\theta &< 180^\circ \\
\text{aspect} & = \frac{w}{h}
\end{align*}
\]
Perspective Projection (cont.)

\[ v(y) \]

\[ \theta \approx 2 \tan^{-1} \left( \frac{h}{2 \cdot \text{near}} \right) \]

\[ \text{aspect} \approx \frac{w}{h} \]

\[ w(z) \]

far

near

bounding box
Example
Example

One-point perspective

Two-point perspective
Vanishing points, horizon lines

- Parallel lines in the scene intersect at the *horizon line*
  - Each pair of parallel lines meet at a *vanishing point*
  - The collection of vanishing points for all sets of parallel lines *in a given plane* is collinear, called the *horizon line* for that plane
Vanishing points, horizon lines
Parallel (Orthographical) Projection

- No perspective foreshortening
  - sizes and angles can be measured and compared
- Useful for engineering drawing
  - top, front, side views

\[
x = X
y = Y
\]
Parallel (Othographic) Projection

- void glOrtho(GLdouble left, right, bottom, top, near, far)

(right, top, -near)

(left, bottom, -near)

(right, top, -far)

(left, bottom, -far)

v(y)

-left

-bottom

-top

-v(y)

-near

-far

-w(z)
Clipping

- Get rid of the things that are not seen
- To do things efficiently require some mathematical twiddling (details later)
Viewport Transform

- void glViewport(GLint x, y, GLsizei width, height);
- The internal buffer is mapped to the rectangle specified by (x,y) lower left corner of size width and height

\[
\begin{align*}
    i &= \frac{(x + a) \times \text{width}}{2a} \\
    j &= \frac{(y + b) \times \text{height}}{2b}
\end{align*}
\]
Viewport Transform

- Multiple buffers can be mapped to a single window (if they have different viewports)
- Distortion may occur if viewport does not have the right aspect ratio

```c
gluPerspective(fovy, 1.0, near, far)  
glViewport(0,0,400,400)              
gluPerspective(fovy, 1.0, near, far)  
glViewport(0,0,400,200)
```
Image Plane?

- Q: Where it is?
- A: It doesn’t really matter (conveniently set at $z=-1$)

- Q: What is the film resolution?
- A: Depend on the real window resolution
  - In reality, to get smooth (anti-aliased) display, rendering is done at sub-pixel accuracy (A-buffer)
OpenGL-Related Libraries

- GLU (prefix glu-)
  - utility library
- GLX (prefix glX-)
  - OpenGL extension to X
- Programming Guide aux library (prefix aux-)
  - windowing, input, simple objects (also try ToGL)
- GLUT (prefix glut-)
  - windowing, input, simple objects (OpenGL1.1 and later, replacing aux- )
OpenGL-Related Libraries

- Open Inventor
  - objects + methods of interaction
  - creating + editing 3D scenes
  - data format exchange
(Most) Basic OpenGL

- Create a drawing buffer (*not* a screen window)
- Clear buffer
- Draw to buffer
- Link buffer to screen window display
- Interaction (expose, resize, mouse, keyboard input, etc).
OpenGL Buffers

- Rectangular arrays of pixels
- Color buffers: front-left, front-right, back-left, back-right
  - At least one, color indexed or RGBA
  - Stereoscopic systems have left and right
  - Doubled buffered systems have front and back

![Graph showing location of pixel (3.0, 2.0)]
OpenGL Buffers

- Color Indexed Buffer

2^n

n bits

24

R G B
OpenGL Buffers

- RGBA Buffer

- 8 bits
- 8 bits
- 8 bits
- 8 bits

- 24
- 256
Other OpenGL Buffers

- Depth buffers:
  - for determining hidden surface effects
- Stencil buffers:
  - acts like a cardboard stencil ("dirty windshield effect")
- Accumulation buffers:
  - for accumulating multiple images into one (e.g. for anti-aliasing, motion blur)
OpenGL Buffer Operations

- **Clear**
  - `void glClear[Color,Index,Depth,Stencil,Accum]`
    - E.g. `glClearColor(0.0,0.0,0.0,0.0);`
    - `glClearDepth(1.0);`
    - Set clear color, depth values
  - `void glClear (GLbitfield mask)`
    - `GL_COLOR_BUFFER_BIT`
    - `GL_DEPTH_BUFFER_BIT`
    - `GL_STENCIL_BUFFER_BIT`
    - `GL_ACCUM_BUFFER_BIT`
  - Etc.
OpenGL Buffer Operations (cont.)

- **Draw**
  - void glDrawBuffer(GLenum mode)
    - enabled for writing or clearing
    - GL_FRONT, GL_BACK, GL_RIGHT, GL_LEFT
    - GL_FRONT_RIGHT, GL_FRONT_LEFT,
    - GL_BACK_RIGHT, GL_BACK_LEFT
    - GL_AUXI, GL_FRONT_AND_BACK,
    - GL_NONE
Misc. OpenGL Functions (cont.)

- **Color**
  - void glColor3f(r, g, b) \(0 \leq r, g, b \leq 1\)
  - “flat” color with no variation
  - affects subsequent primitives

- **Proper depth cue**
  - void glEnable(GL_DEPTH_TEST)
  - void glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
Misc. OpenGL Functions

- Forced completion
  - void glFlush(void)
    - asynchronous
  - void glFinish(void)
    - synchronous
- One of them should be called (glFlush()) at the end of each frame
Helper Libraries

- Remember that OpenGL does not do
  - windowing, GUI, and modeling
- A real application will need all the above
- At least three choices
  - GLUT (GL Utility Library) or aux (obsolete)
    - simple windowing, GUI and models
    - /fs/contrib/src/mesa/current/sample_executable/glut for GLUT examples (aux examples very similar)
  - Togl
    - allow OpenGL to work with Tcl/Tk for a much more sophisticated GUI
GL Utility Library (glut)

- Convenient and easy-to-use
- For
  - specifying the display mode
  - creating window (size and location)
  - handling window and input events
  - convenient objects
  - Menu and buttons
- Replaced aux library after version 1.1
- Use X Window callback mechanism, ported to both MS Windows and Mac
GUI Programming

- Have little control of what user will do
  - Self (resize, minimize, maximize, reshape, key, mouse)
  - Others (overlap, pop up or down)
- Window manager is the master (UI policy)
- Prepare for all contingency (events) in advance and register with glut
- At run time, window manager delivers events and data to glut to invoke the right "callback"
GL Utility Library (glut)

- Void glutInit(int argc, char **argv)
  - initialize glut, process command line arguments such as -geometry, -display, etc.
- void glutInitDisplayMode(unsigned int mode)
  - Mode for later glutCreateWindow() call
  - GLUT_RGBA or GLUT_INDEX
  - GLUT_SINGLE or GLUT_DOUBLE
  - Associated GLUT_DEPTH, GLUT_STENCIL, GLUT_ACCUM buffers
  - default: RGBA & SINGLE (use RGBA, DOUBLE, and DEPTH)
GL Utility Library (glut)

- `void glutInitWindowPosition(int x, int y)`
- `void glutInitWindowSize(int width, int height)`
  - Window location and size
  - These are hints to the underlying window system and may not be honored
- `Void glutPositionWindow(int x, int y)`
- `Void glutReshapeWindow(int width, int height)`
- `Void glutFullScreen(void)`
  - For both top-level and subwindows
  - For top-level windows, hints to the underlying windowing system (may not be honored)
GL Utility Library (glut)

- int glutCreateWindow(char *name)
  - after Init, Displaymode, Position, and Size calls
  - will not appear until glutMainLoop
  - WinID starts at 1

- int glutCreatSubWindow(int win, int x, int y, int width, int height)
  - Hierarchical (nested windows)
  - (x,y) relative to the parent (win)
GL Utility Library (glut)

- void glutSetWindow (int win)
- int glutGetWindow(void)
  - Set and get current window
- void glutDestroyWindow (int win)
GL Utility Library (glut)

- Windows may have layers (normal, back for double buffered windows)

- `void glutSwapBuffers(void)`
  - Swap the *layer in use* of the current buffer
  - No effect if not doubled buffered
GL Utility Library (glut)

- void glutDisplayFunc(void (*func) (void))
  - display function for initial display, de-iconfy and expose

- void glutReshapeFunc(void (*function) (width, height))
  - called when window is resized or moved
  - default glViewport(0,0,width,height)
**GL Utility Library (glut)**

- `void glutKeyboardFunc(void *(func) (unsigned int key, int x, int y))`
  - ASCII code for key
  - `(x,y)` for window location when the key was pressed

```c
switch (key) {
    case 'z':
        // action
        break;
    case 'x':
        // action
        break;
    default:
        exit(0);
}
```
void glutMouseFunc(void *(func) (int button, int state, int x, int y))

- button: GLUT_{LEFT,MIDDLE,RIGHT}_BUTTON
  - Be careful of GLUT_MIDDLE_BUTTON (3 for 4)
- mode: GLUT_UP, GLUT_DOWN

```c
if (button==GLUT_LEFT_BUTTON) {
    if (state==GLUT_DOWN) { // left mouse button down
    } else if (state==GLUT_UP) {
    }
} else if(button==GLUT_RIGHT_BUTTON) {
    if (state==GLUT_DOWN) { // right mouse button down
    } else if (state==GLUT_UP) {
    }
} else if (button==3) { // mouse wheel scroll up
} else if (button==4) { // mouse wheel scroll down
```
GL Utility Library (glut)

- void glutMotionFunc(void *(func) (int x, int y))
  - mouse pointer move while one or more mouse buttons is pressed
GL Utility Library (glut)

- glut{Wire,Solid}Sphere()
- glut{Wire,Solid}Cube()
- glut{Wire,Solid}Box()
- glut{Wire,Solid}Torus()
- glut{Wire,Solid}Cylinder()
- glut{Wire,Solid}Cone()
- glut{Wire,Solid}Teapot()
  - centered at the origin
- glut{Wire,Solid}(Icosahedron,Octahedron,Tetrahedron,dodecahedron)
GL Utility Library (glut)

- void glutMainLoop (void)
  - GLUT main loop, never returns
Sample Programs

- Close to a hundred of them under http://www.cs.ucsb.edu/~cs180/sampleprograms.tar.gz
- Note that they are all C functions
- You can use C++ for sure
- Caveats:
  - Missing data/images files (may not compile)
  - Have to fix Makefile manually (depending on your OS)
  - Work one year but not next
    - Must have GL (/usr/include/GL)
    - Must have glut (/usr/include/GL)
    - Must have X (/usr/X11R6/ or /usr/)
    - Libraries got split/merged (/usr/lib -> /usr/lib64)
GLUT Example

#include <GL/glut.h>
#include <stdlib.h>
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glLoadIdentity ();             /* clear the matrix */
    /* viewing transformation */
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glScalef (1.0, 2.0, 1.0);      /* modeling transformation */
    glutWireCube (1.0);
    glFlush ();
}
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
    glMatrixMode (GL_MODELVIEW);
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
    case 27:
        exit(0);
        break;
    }
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
ToGL

- A special widget (like the Canvas widget) that allows OpenGL to draw to it
- Tcl+Tk for GUI and OpenGL for 3D graphics
- Need togl.c togl.h and tkInit4.0.h
- Otherwise, very easy to use
- /fs/contrib/src/Togl/current/
**Togl Widget**

- **Initialization**
  - `Togl_Init(Tcl_Interp *interp)`
  - `main` calls `Tk_Main` calls `your_main` calls `Togl_Init`

- **Callbacks**
  - `void Togl_CreateFunc(void (*function) (struct Togl*))`
  - `void Togl_DisplayFunc(void (*function) (struct Togl*))`
  - `void Togl_ReshapeFunc(void (*function) (struct Togl*))`
  - `void Togl_DestroyFunc(void (*function) (struct Togl*))`
    - called when Togl widget is created, redrawn, resized and destroyed
Togl Widget

- Tcl/Tk commands for Togl
  - Togl_CreateCommand("tcl_name", c_name)
    ```c
    int c_name(struct Togl*, int argc, int argv**)
    {
      ...
      return TCL_OK or TCL_ERROR
    }
    ```

- Check Togl homepage for more functions