

Outline

GLUT and Event Callbacks
Elementary Rendering
Points
Lines and Polylines
Polygons

A Simple Program

```
#include <GL/glut.h>

void display() {
    glClearColor(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glVertex2f(-0.5,-0.5);
        glVertex2f(-0.5,0.5);
        glVertex2f(0.5,0.5);
        glVertex2f(0.5,-0.5);
    glEnd();
    glFlush();
}

int main( int argc, char** argv ){
    glutInit( &argc, argv );
    glutCreateWindow( "My First Window" );
    glutDisplayFunc( display );
    glutMainLoop();
}
```

Simple

More Modifications

```
int main( int argc, char** argv ){
    glutInit( &argc, argv );
    → glutInitWindowSize(100, 200);
    → glutInitWindowPosition(100, 100);
    glutCreateWindow( "My First Window" );
    glutDisplayFunc( display );
    → glClearColor(1.0, 1.0, 1.0, 1.0);
    → glutMainLoop();
}
```

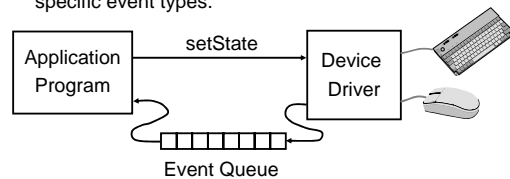
Try various values for Window size and position (including negative values)
Try various values for ClearColor
Try placing these function calls in different orders within the main loop

What do we notice?

The Event Model

Events from the input hardware are queued into an event queue.

Application routines (event handlers or callbacks) handle specific event types.



GLUT Callback Functions

"Register" callbacks with GLUT:

```
glutDisplayFunc( display );
glutReshapeFunc( reshape );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
glutMouseFunc( mouseButton );
glutMotionFunc( mouseMove );
```

Display Callback

Do all of your drawing here:

```
glutDisplayFunc( display );
```

```
void display( void )
{
    glClearColor( GL_COLOR_BUFFER_BIT );
    glBegin( GL_TRIANGLE_STRIP );
        glVertex3fv( v[0] );
        glVertex3fv( v[1] );
        glVertex3fv( v[2] );
        glVertex3fv( v[3] );
    glEnd();
    glutSwapBuffers();
}
```

Reshape Callback

Called whenever the window is resized:

```
    glutReshapeFunc( reshape );

void reshape( GLsizei w, GLsizei h )
{
    GW = w; // Set global width and height
    GH = h;
    glViewport( 0, 0, w, h ); // Set viewport
    glLoadIdentity();
    gluOrtho2D( -(float)w/h, (float)w/h, -1., 1.);
}
```

Another modification

```
#include <GL/glut.h>
int GH;
int GW;

void reshape(int w, int h) {
    GW = w;
    GH = h;

    glViewport(0, 0, w, h);
    glLoadIdentity();
    gluOrtho2D( -(float)w/h, (float)w/h, -1, 1);
}
```

More Modifications

```
int main(int argc, char** argv) {

    glutInit(&argc, argv);
    glutInitWindowSize(200, 200);
    glutCreateWindow("My First Window");
    glViewport(0, 0, 200, 100);
    GW = GH = 200;
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    → glClearColor(1.0, 1.0, 1.0, 0.0);
    gluOrtho2D(-2, 2, -2, 2);

    glutMainLoop();
}
```

Simple

What do we notice

What happens when you reshape the window?

Mouse Callbacks

GLUT lets you define several callback functions for mouse events:

```
glutMouseFunc( mouseButton );
- Mouse button press or release events.
glutMotionFunc( mouseMove );
- Mouse movement while button is pressed events.
```

Mouse Button Callback

Design a callback function mouseButton as follows:

```
void mouseButton( int button, int state, int x, int y );
- int button: GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON,
  GLUT_RIGHT_BUTTON
- int state: GLUT_UP, GLUT_DOWN
- int x, int y: mouse position
```

GLUT specifies the format of the callback functions, i.e. the variables that passed to the function

- What is the consequences of this???

ADD the Following

```
void mouse(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON) {
        if (state == GLUT_DOWN) { /* if the left button is
            clicked */
            printf("mouse clicked at %d %d\n", x, y);
        }
    }
}

void mouseMove(int x, int y) {
    printf("mouse moved at %d %d\n", x, y);
}
```

And to main:

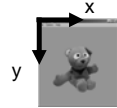
```
glutMouseFunc(mouse);
glutMotionFunc(mouseMove);
```

Simple

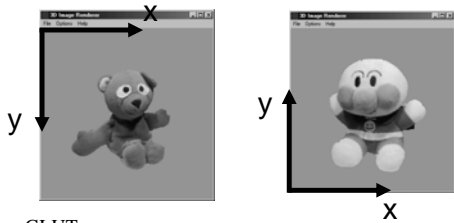
What do we notice

Click the mouse at the top of the window and drag it down
– What are the coordinates?

Important: the mouse position y coordinate is reversed from the usual OpenGL screen coordinate system!



Window Coordinate Systems



GLUT
Windows API
X Windows under Unix /
Linux
Apple QuickDraw

OpenGL

Instead Following

```
void mouse(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON) {
        if (state == GLUT_DOWN) {
            printf("mouse clicked at %d %d\n", x, GH-1-y);
        }
        glutPostRedisplay();
    }
}

void mouseMove(int x, int y) {
    printf("mouse moved at %d %d\n", x, GH-1-y);
}
```

Mouse Motion

Design a callback function mouseMove as follows:

```
void mouseMove( int x, int y );
– int x, int y: mouse position
```

This event only occurs if the mouse is moved while some button is held down.

Co-ordinate transforms

If I want to draw a point wherever I click the mouse - what do I need to do?

Recall that we learned

- to transform a point from world co-ordinates to image co-ordinates
- to transform a point from image co-ordinates to pixel co-ordinates
- What coordinates are (x,y) returned by the mouse callback functions?
- What coordinates do we draw in?

Simple

Keyboard Callback

Process keyboard input:

```
glutKeyboardFunc( keyboard );

void keyboard(unsigned char key, int x, int y){
    switch( key ) {
        case 'q': case 'Q' :
            exit( EXIT_SUCCESS );
            break;
        case 'h' : case 'H' :
            printf("hello!\n");
            break;
    }
}
```

Simple

Idle Callback

The idle callback is called whenever the event queue is empty:

```
glutIdleFunc( idle );
Use for animation and state update.

void idle( void )
{
    t += dt;
    glutPostRedisplay();
}
```

glutPostRedisplay() vs. display()

glutPostRedisplay() marks the current window as needing to be redisplayed.

- Similar to calling display() directly, but display only occurs once per event loop.

Imagine you have several event callbacks:

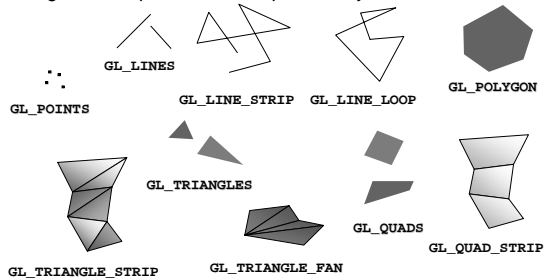
- 1) You call display() in every callback.
- 2) You call glutPostRedisplay() in every callback.
- Which is the smarter choice?

Outline

- GLUT and Event Callbacks
- Elementary Rendering
 - Points
 - Lines and Polylines
 - Polygons

OpenGL Geometric Primitives

All geometric primitives are specified by vertices



A Simple Example

Primitives are specified using

```
glBegin( primType );
glEnd();
```

- *primType* determines how vertices are combined

```
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
```

Outline

GLUT and Event Callbacks
Elementary Rendering
Points
Lines and Polylines
Polygons

Drawing Points

Use `glBegin` to tell OpenGL what kind of primitive you want to specify with the vertices (e.g., `GL_POINTS`).

Use `glEnd` to end the list of vertices.

```
glBegin( GL_POINTS );
glVertex2d( 0.3, 0.3 );
glVertex2d( 0.5, 0.5 );
glEnd();
```

Note the indentation!

Points

Setting the Point Size

The size of a point is 1 pixel by default.

You can change this by calling:

```
void glPointSize( GLfloat size );
```

To find the range of point sizes that are supported use:

```
GLfloat sizes[2]; // Supported point size range
GLfloat step; // Supported point size increments
// Get supported point size range and step size
glGetFloatv( GL_SMOOTH_POINT_SIZE_RANGE, sizes );
glGetFloatv( GL_SMOOTH_POINT_SIZE_GRANULARITY, &step );
```

The largest or smallest point size is used if you use a non-supported point size.

Pointsz

Drawing Lines

We can specify lines with two vertices:

```
glBegin( GL_LINES );
glVertex2d( 0.2, 0.2 );
glVertex2d( 0.5, 0.5 );
glEnd();
```

For every two vertices in `GL_LINES` a line is drawn.

If you specify an odd number of vertices the last vertex is ignored.

Lines Fanned in a Circle

We can use this feature to save calls to

`glBegin/glEnd`.

```
glClear( GL_COLOR_BUFFER_BIT );
glBegin( GL_LINES );
for ( GLfloat angle=0.0f; angle<=GL_PI*3.0f; angle+=0.5f )
{
    // First endpoint of line
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);
    glVertex2d( x, y );

    // Second endpoint of line
    x = 50.0f*sin(angle + 3.1415f);
    y = 50.0f*cos(angle + 3.1415f);
    glVertex2d( x, y );
}
glEnd();
glFlush();
```

Lines

Line Strips and Loops

`GL_LINE_STRIP` draws a polyline from one vertex to the next in a continuous segment.

```
glBegin( GL_LINE_STRIP );
glVertex2d( 0, 0 );
glVertex2d( 50, 50 );
glVertex2d( 50, 100 );
glEnd();
```

`GL_LINE_LOOP` behaves just like

`GL_LINE_STRIP`, except that a final line is drawn between the last and the first vertex specified.

LStrips

Setting Line Width

You can set line widths by using:

```
void glLineWidth( GLfloat width );
```

To enable / disable line antialiasing:

```
glEnable( GL_LINE_SMOOTH );
```

To find the range of line widths that are supported

```
GLfloat sizes[2]; // Supported line width range
GLfloat step; // Supported line width increments
// Get supported point size range and step size
glGetFloatv( GL_SMOOTH_LINE_WIDTH_RANGE, sizes );
glGetFloatv( GL_SMOOTH_LINE_WIDTH_GRANULARITY, &step );
```

The OpenGL specification only requires that a line width of 1.0 be supported.

Linesw

Drawing Polygons

`GL_POLYGON` draws a convex polygon between a number of vertices.

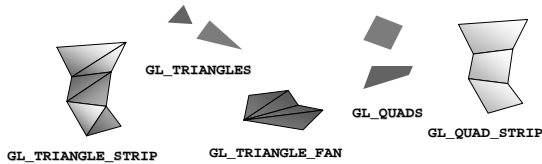
```
glBegin( GL_POLYGON );
glVertex2d( x0, y0 );
glVertex2d( x1, y1 );
:
glVertex2d( xn, yn );
glEnd();
```

Star

The polygon will be filled with the current color.

OpenGL Polygon Shapes

Use any of the following constants in `glBegin()` to draw these five basic polygon shapes:



2D Rectangles

The axis aligned 2D rectangle has its own OpenGL function:

```
glRecti( GLint x1, GLint y1, GLint x2, GLint y2 );
```

(x1, y1)



(x2, y2)

Of course, you could also draw a rectangle using:
`glBegin(GL_POLYGON);`