Graphics Programming using OpenGL and GLUT

Graphics programs are finite state machines. The API has functions that define objects that flow through the graphics pipeline, and those that change the state of the machine to cause varied visible output.

```
main()
{
    initialize_state_machine();
    for(some set of objects)
    {
        obj = generate_object();
        display_object(obj);
    }
    cleanup();
}
```

The API is broken down into:

- *Primitive Functions:* Describes low-level objects (e.g. points, line segments, polygons, pixels, text, ...) that system can display.
- *Attribute Functions:* Govern the way the primitives appear on the display. (e.g. line segment color, pattern for polygon, ...)
- *Viewing Functions:* Specify different views that the synthetic camera can provide.
- *Transformation Functions:* Rotation, Translation, Scaling of Objects
- *Input Functions:* For diverse input devices (e.g. keyboards, mice, data tablets)
- *Control Functions:* Communicate with window system, to initialize programs, deal with execution errors etc.
- *Inquiry Functions:* To determine display device parameters,

Primitive Functions

OpenGL types are used rather than C types, and defined in header files

#define GLfloat float

glVertexNT

where N = 2,3,4 /*number of dimensions */

and T = i,f,d /* datatype (integer,float,double) */

glVertex2i(GLint x, GLint y)

glVertex3f(GLfloat x, GLfloat y, GLfloat z)

glvertexNTv

where additionally v indicates that variables are specified through a pointer to an array, rather than via an argument list.

GLfloat vertex[3] glVertex3fv(vertex)

Begin/End allow the definition of the geometric type a collection of vertices

```
glBegin(GL_POINTS);
   glVertex3f(x1, y1, z1);

   glVertex3f(x2, y2, z2);

   glVertex3f(x3, y3, z3);
glEnd();

glBegin(GL_LINES);
   glVertex3f(x1, y1, z1);

   glVertex3f(x2, y2, z2);
glEnd();
```

Strings such as GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_POLYGON, GL_TRIANGLE, GL_QUADS, GL_QUAD_STRIP, etc. are defined in .h files

So include lines are needed #$include < GL/glut.h >$ to read in glut.h, gl.h, glu.h etc.

Attribute Functions

Attributes such as point size and line width are specified in terms of the pixel size.

glPointSize(2.0) allows rendering points to be 2 pixels wide

glClearColor (1.0,1.0,1.0,1.0); is specifying RGBA opaque white color.

The use of indexed color and color lookup table allow for color palettes.

In color-index mode, the present color is selected by the function glIndexi(element); which pulls out a particular color out of the color lookup table.

glutSetColor (int color, GLfloat red, Glfloat blue, GLfloat green) allows the setting of entries in a color table for each window.

Viewing / Transformation Functions

right-parallelpiped viewing / camera volume is specified via

glOrtho(GLdouble left, GLdouble right, Gldouble bottom, Gldouble top. GLdouble near, GLdouble far);

near and far distances are measured from the camera (eye) position. The camera starts off at the origin pointing in the negative z direction.

The orthographic projection displays only those objects inside the viewing volume, and unlike a real camera, can also include objects behind the camera (eye) position, as long as viewing volume contains the eye position.

Two important matrices in OpenGl are are the ModelView and the Projection Matrices. At any time the state machine has values for both of these matrices. Typically these are initialized to identity matrices. The transformation function sequence modifies these matrices. To select the matrix to which the transformation is to be applied, one first sets the MatrixMode.

The sequence below defines a 500x500 viewing region with the lower left corner at the origin of a 2D viewing system.

```
glMatrixMode(GL_PROJECTION);

GlLoadIdentity();

gluOrtho2D(0.0,500.0, 0.0, 500.0);

glMatrixMode(GL_MODELVIEW);
```

Control Functions

The OpenGL Utility Toolkit (GLUT) is a library of functions that provides a simple interface between the graphics subsystem and the operating and window systems of the computer platform.

Window or screen window denotes a rectangular are of our device display, and displays the contents of the frame buffer. The window has a height and width and measured in window/screen coordinates, with units as pixels.

Interaction between windowing system and OpenGL is initialized by the GLUT function

```
glutInit(int *argcp, char **argv)
```

The two arguments allow the user to pass command-line arguments, as in the standard C main function.

Opening an OpenGL window can be accomplished by glutCreateWindow(char *title) where the string title is displayed on the top of the window. The window has a default size, a position on the screen, and characteristics such as use of RGB color. GLUT functions allow specification of these parameters

glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE); /*parameters OR-ed*/

RGB color (default) instead of indexed (GLUT_INDEX) color

Depth buffer (not default) for hidden-surface removal

double buffering rather than the default single (GLUT_SINGLE) buffering.

glutWindowSize(480,640);

glutInitWindowPosition(0,0);

Example OpenGL and GIUT program

A simple example with straightforward interaction, single window display. More user interaction is the topic of the next lecture.

```
#include<GL/glut.h>
    void main(int argc, char **argv) /*calls to GLUT to set up
    windows, display properties, and event processing callbacks*/
    {
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT\_SINGLE $|$ GLUT\_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Fractal example");
        glutDisplayFunc(display); /*display callback that sends graphics to the s
        myinit(); /*set up user options*/
        glutMainLoop(); /* event processing loop for interactive graphics progra
    }
```

The Initialization and Display Functions

```
void myinit(void)
{
    /*attributes*/
    glClearColor(1.0, 1.0, 1.0, 0.0); /*white background*/
    glColor3f(1.0, 0.0, 0.0); /*draw in red*/

    /*set up viewing*/
        glMatrixMode(GL\_Projection);
        glLoadIdentity();
        gluOrtho2D(0.0, 500.0, 0.0, 500.0);
        glMatrixMode(GL\_ModelVIEW);
}

void display(void)
{
    typedef GLfloat point2[2]; /*a point data type */

        point2 vertices[3] = {{0.0,0.0},{250.0, 500.0},{500.0,0.0}} ; /*defi
        int i,j,k;
        int rand(); /*random number generator */
        point2 p = {75.0, 50.0);  /*point inside triangle */
    /*compute*/
    glClear(GL\_COLOR\_BUFFER\_BIT); /*clear the window*/

    for(k=0;k < 5000;k++)
    {
    j=rand()%3; /*randomly select a vertex */
    p[0] =(p[0]+vertices[j][0])/2.0;
    p[1] =(p[1]+vertices[j][1])/2.0;
    /*plot points*/
        glBegin(GL\_Points);
        glVertex2fv(p);
        glEnd();
    }
glFlush(); /*forces system to plot points on display as soon as possible*/
}
```

Basic Graphical User Interface Concepts

- Physical input devices used in graphics
- Virtual devices
- Polling vs event processing
- UI toolkits for generalized event processing

Physical Devices

Actual, physical input devices include:

- Dials (potentiometers)
- Selectors
- Pushbuttons
- Switches
- Keyboards (collections of push buttons called “keys”)
- Trackballs (relative motion)
- Mice (relative motion)
- Joysticks (relative motion, direction)
- Tablets (absolute position)
- etc.

Virtual Devices

Devices can be classified according to the kind of value they return:

- Button: returns a Boolean value; can be *depressed* or *released*.
- Key: returns a character; can also be thought of as a button.
- Selector: returns an integral value (in a given range).
- Valuator: returns a real value (in a given range).
- Locator: returns a position in (2D/3D) space (usually several ganged valuators).

Each of the above is called a *virtual device*.

Polling and Sampling

In *polling*, the value of an input device is constantly checked in a tight loop:

- To record the motion of a valuator
- To wait for a change in status

If a record is taken, this input mode may be called *sampling*.

Generally, polling is inefficient and should be avoided, particularly in time-sharing systems.

Event Queries

- Device is monitored by an asynchronous process.
- Upon change in status of device, this process places a record into an *event queue*.
- Application can request read-out of queue:
  - number of events
  - 1st waiting event
  - highest priority event
  - 1st event of some category
  - all events
- Application can also
  - specify which events should be placed in queue
  - clear and reset the queue
  - etc.
- Queue reading can be blocking or non-blocking.

- The cursor is usually *bound* to a pair of valuators, typically `MOUSE_X` and `MOUSE_Y`.
- Events can be restricted to particular areas of the screen, based on the cursor position.
- Events can be very general or specific:
  - a mouse button or keyboard key is depressed
  - a mouse button or keyboard key is released
  - the cursor enters a window
  - the cursor has moved more than a certain amount
  - an `Expose` event is triggered under X which a window becomes visible
  - a `Configure` event is triggered when a window is resized
  - a timer event may occur after a certain interval
- Simple event queues just record a code for event (Iris GL).
- Better event queues record extra information such as time stamps (X windows).

GUI Toolkits

Event-loop processing can be generalized

1. Instead of a `switch`, use table lookup.
2. Each table entry associates an event with a *callback* function.
3. When the event occurs, the *callback* is invoked.
4. Provide an API to make and delete table entries.
5. Divide screen into parcels, and assign different callbacks to different parcels (X windows does this).

Modular UI functionality is provided through a collection of *widgets*.

- Widgets are parcels of the screen that can respond to events.
- A widget has a graphical representation that suggests its function.
- Widgets may respond to events with a change in appearance, as well as issuing callbacks.
- Widgets are arranged in a parent/child hierarchy.
- Widgets may have multiple parts, and in fact may be composed of other widgets in a hierarchy.

Some UI toolkits: Xm, Xt, SUIT, FORMS, Tk, GLUT, GLUI, QT, ...

GUI Programming using OpenGL and GLUT

Last time we looked at a simple example with straightforward interaction, single window display.

```
#include<GL/glut.h>
    void main(int argc, char **argv) /*calls to GLUT to set up
    windows, display properties, and event processing callbacks*/
    {
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT\_SINGLE $|$ GLUT\_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Fractal example");

        /*display callback that sends graphics to the screen */
        glutDisplayFunc(display);
        /*set up user options*/
        myinit();
    /* enter event processing loop for interactive graphics programs*/
    glutMainLoop();
}
```

Extensions to GUI using GLUT -I

We shall look at event-driven input that uses the callback mechanism in GLUT. X-11 window system is more general.

- `Move` event is generated when the mouse is moved with one of the buttons depressed
- `Passive Move` event is generated when the mouse is moved without a button being held down
- `Mouse` event is generated when one of the mouse buttons is either depressed or released.

The mouse callback function is specified usually in the main function by means of the GLUT function

glutMouseFunc(mouse)

The mouse callback has the form below, where we specify the action(s) that we want to take place on the occurrence of the event

```
void mouse(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
            exit(); /* causes termination of the program*/
}
```

As long as no other callbacks are defined and registered with the Window system, no response action will occur if any other mouse event occurs.

Extensions to GUI using GLUT - II

```
void main(int argc, char **argv) /*draw a small box at each
location where mouse is located when left button is pressed*/
    {
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT\_SINGLE $|$ GLUT\_RGB);

        glutCreateWindow("Square");
        myinit();

        glutReshapefunc(myReshape);
        glutMouseFunc(mouse);

        glutDisplayFunc(display);
        /* enter event processing loop for interactive graphics programs*/
        glutMainLoop();
    }
```

```
void mouse(int button, int state, int x, int y)
    {
        if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
                drawSquare(x,y);
        if(button == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
                exit();
    }
```

Extensions to GUI using GLUT - II

The Initialization and Display Functions

```
/*globals*/
GLsizei wh = 500, ww = 500; /*initial window size*
GLfloat size = 3.0; /*one-half of side length of square */

void myinit(void)
{
    /*set viewing conditions*/
    glViewport(o,0,ww,wh);
    glMatrixMode(GL_Projection);
    glLoadIdentity();
    gluOrtho2D(0.0, ww, 0.0, wh);
    glMatrixMode(GL_ModelVIEW);

    glClearColor(10.0, 0.0, 0.0, 0.0); /*black background*/
    glClear(GL_COLOR_BUFFER_BIT); /*clear window*/
    glFlush();

}

void drawsquare(int x, int y)
{
    /*flip as window system has its origin at top left*/
    y = wh -y;
    /*pick a random color*/
    glColor3ub((char) rand()%256, (char) rand()%256, (char) rand()%256);
    glBegin(GL_POLYGON);
        glVertex2fv(x+size,y+size);
        glVertex2fv(x-size,y+size);
        glVertex2fv(x-size,y-size);
        glVertex2fv(x+size,y-size);
    glEnd();
    glFlush();
}
```

Picking and 3D Selection using OpenGL and GLUT

- *Pick:* Select an object by positioning mouse over it and clicking
- *Question:* How do we decide what was picked?
  - We could do the work ourselves:
    - Map selection point to a ray
    - Intersect with all objects in scene
  - Let OpenGL/graphics hardware do the work
- *Idea:* Draw entire scene, and "pick" anything drawn near the cursor
  - Only "draw" in a small viewport near the cursor
  - Just do clipping, no shading or rasterization
  - Need a method of identifying "hits"
  - OpenGL uses a *name stack* managed by `glInitNames()`, `glLoadName()`, `glPushName()`, and `glPopName()`
  - "Names" are short integers
  - When hit occurs, copy entire contents of stack to output buffer
- *Example:*

```
glSelectBuffer(size, buffer);          /* initialize */
glRenderMode(GL_SELECT);
glInitNames();
glGetIntegerv(GL_VIEWPORT, viewport);  /* set up pick view */
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glIdentity();
gluPickMatrix(x, y, w, h, viewport);
glMatrixMode(GL_MODELVIEW);

ViewMatrix();
glLoadName(1);
Draw1();
glLoadName(2);
Draw2();

glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
hits = glRenderMode(GL_RENDER);
```

- What you get back:
  - If you click on Item 1 only:
    `hits = 1,`
    `buffer = 1, min(z1), max(z1), 1.`
  - If you click on Item 2 only:
    `hits = 1,`
    `buffer = 1, min(z2), max(z2), 2.`
  - If you click over both Item 1 and Item 2:
    `hits = 1,`
    `buffer = 1, min(z1), max(z1), 1, 1, min(z2), max(z2), 2.`

- More complex example:

```
/* initialization stuff goes here */
glPushName(1);
    Draw1();                    /* stack: 1 */
    glPushName(1);
        Draw1_1();              /* stack: 1 1 */
        glPushName(1);
            Draw1_1_1();        /* stack: 1 1 1 */
        glPopName();
        glPushName(2);
            Draw1_1_2();        /* stack: 1 1 2 */
        glPopName();
    glPopName();
    glPushName(2);
        Draw1_2();              /* stack: 1 2 */
    glPopName();
glPopName();
glPushName(2);
    Draw2();                    /* stack: 2 */
```

```
glPopName();
/* wrap-up stuff here */
```

- What you get back:
  - If you click on Item 1:

    `hits = 1,`

    `buffer = 1, min(z1), max(z1), 1.`
  - If you click on Items 1:1:1 and 1:2:

    `hits = 2,`

    `buffer = 3, min(z111), max(z111), 1, 1, 1, 2, min(z12), max(z12), 1, 2.`
  - If you click on Items 1:1:2, 1:2, and 2:

    `hits = 3,`

    `buffer = 3, min(z112), max(z112), 1, 1, 2, 2, min(z12), max(z12), 1, 2, 1, min(z2), max(z2), 2.`
- In general, if h is the number hits, the following is returned.
  - `hits` $= h$.
  - h *hit records*, each with four parts:
    1. The number of items q on the name stack at the time of the hit (1 int).
    2. The minimum z value among the primitives hit (1 int).
    3. The maximum z value among the primitives hit (1 int).
    4. The contents of the hit stack, deepest element first (q ints).

- Important Details:
  - Make sure that projection matrix is saved with a `glPushMatrix()` and restored with a `glPopMatrix()`.
  - `glRenderMode(GL_RENDER)` returns negative if buffer not big enough.
  - When a hit occurs, a flag is set.
  - Entry to name stack only made at next `gl*Name(s)` or `glRenderMode` call. So, each draw block can only generate at most one hit.

Reading Assignment and News

Chapter 2 pages 39 - 95, and Chapter 3 pages 99- 153, of Recommended Text.

Project 1 has been posted.

Please also track the News section of the Course Web Pages for the most recent Announcements related to this course.

(http://www.cs.utexas.edu/users/bajaj/graphics25/cs354/)