Chapter 2: Graphics Programming Using OpenGL

李同益

成功大學資訊工程系

OpenGL and GLUT Overview

Part I

OpenGL and GLUT Overview

- **What is OpenGL & what can it do for me?**
- **OpenGL in windowing systems**
- **Why GLUT**
- **A GLUT program template**

What Is OpenGL?

- Graphics rendering API
  - high-quality color images composed of geometric and image primitives
  - window system independent
  - operating system independent

OpenGL as a Renderer

- Geometric primitives
  - points, lines and polygons
- Image Primitives
  - images and bitmaps
  - separate pipeline for images and geometry
    - linked through texture mapping
- Rendering depends on state
  - colors, materials, light sources, etc.

SGI and GL

- Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
- To use the system, application programmers used a library called GL
- With GL, it was relatively simple to program three dimensional interactive applications

OpenGL

- The success of GL lead to OpenGL (1992), a platform-independent API that was
  - Easy to use
  - Close enough to the hardware to get excellent performance
  - Focus on rendering
  - Omitted windowing and input to avoid window system dependencies

OpenGL Evolution

- Controlled by an Architectural Review Board (ARB)
  - Members include SGI, Microsoft, Nvidia, HP, 3DLabs,IBM,…….
  - Relatively stable (present version 1.4)
    - Evolution reflects new hardware capabilities
      - **3D texture mapping and texture objects**
      - **Vertex programs**
  - Allows for platform specific features through extensions

Related APIs

- AGL, GLX, WGL
  - glue between OpenGL and windowing systems
- GLU (OpenGL Utility Library)
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.
- GLUT (OpenGL Utility Toolkit)
  - portable windowing API
  - not officially part of OpenGL

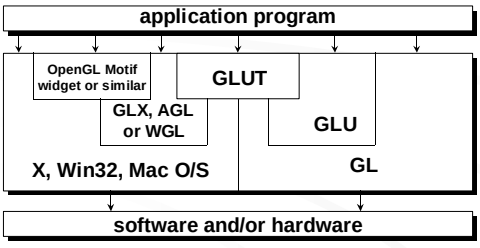
OpenGL Libraries

- OpenGL core library
  - OpenGL32 on Windows
  - GL on most unix/linux systems
- OpenGL Utility Library (GLU)
  - Provides functionality in OpenGL core but avoids having to rewrite code
- Links with window system
  - GLX for X window systems
  - WGL for Widows
  - AGL for Macintosh

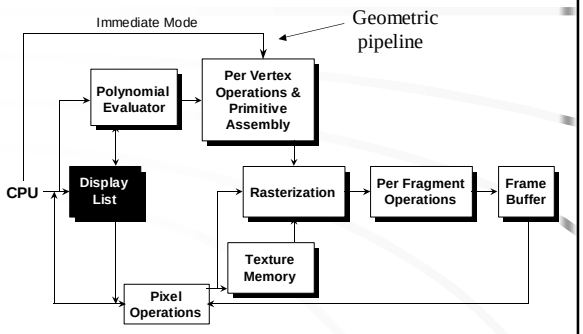
GLUT

- OpenGL Utility Library (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven
  - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - Slide bars

OpenGL and Related APIs



OpenGL Architecture



Preliminaries

- Headers Files
  - #include <GL/gl.h>
  - #include <GL/glu.h>
  - #include <GL/glut.h>
- Libraries
- Enumerated Types
  - OpenGL defines numerous types for compatibility
    - GLfloat, GLint, GLenum, etc.

Compilation on Windows

- Visual C++
  - Get glut.h, glut32.lib and glut32.dll from web
  - Create a console application
  - Add opengl32.lib, glut32.lib, glut32.lib to project settings (under link tab)
- Borland C similar

GLUT Basics

- Application Structure
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop

Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - **`main()`**:
    - defines the callback functions
    - opens one or more windows with the required properties
    - enters event loop (last executable statement)
  - **`init()`**: sets the state variables
    - viewing
    - Attributes
  - callbacks
    - Display function
    - Input and window functions

`main.c`

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

int main(int argc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
 glutCreateWindow("simple");
 glutDisplayFunc(mydisplay);

 init();

 glutMainLoop();
}
```

includes gl.h

define window properties

display callback

set OpenGL state

enter event loop

GLUT functions

- **`glutInit`** allows application to get command line arguments and initializes system
- **`gluInitDisplayMode`** requests properties of the window (the *rendering context*)
  - RGB color
  - Single buffering
  - Properties logically ORed together
- **`glutWindowSize`** in pixels
- **`glutWindowPosition`** from top-left corner of display
- **`glutCreateWindow`** create window with title "simple"
- **`glutDisplayFunc`** display callback
- **`glutMainLoop`** enter infinite event loop

`init.c`

```
void init()
{
  glClearColor (0.0, 0.0, 0.0, 1.0);

  glColor3f(1.0, 1.0, 1.0);

  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```

black clear color

opaque window

fill with white

viewing volume

Sample Program

```
void main( int argc, char** argv )
{
  int mode = GLUT_RGB|GLUT_DOUBLE;
  glutInitDisplayMode( mode );
  glutCreateWindow( argv[0] );
  init();
  glutDisplayFunc( display );
  glutReshapeFunc( resize );
  glutKeyboardFunc( key );
  glutIdleFunc( idle );
  glutMainLoop();
}
```

OpenGL Initialization

- Set up whatever state you're going to use

```
void init( void )
{
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );

  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
}
```

GLUT Callback Functions

- Routine to call when something happens
  - window resize or redraw
  - user input
  - animation
- "Register" callbacks with GLUT

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
```

Rendering Callback

- Do all of your drawing here
  - `glutDisplayFunc( display );`

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

Idle Callbacks

- Use for animation and continuous update
  - `glutIdleFunc( idle );`

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

User Input Callbacks

- Process user input
  - `glutKeyboardFunc( keyboard );`

```
void keyboard( unsigned char key, int x, int y )
{
  switch( key ) {
    case 'q' : case 'Q' :
      exit( EXIT_SUCCESS );
      break;
    case 'r' : case 'R' :
      rotate = GL_TRUE;
          glutPostRedisplay();
      break;
  }
}
```

Elementary Rendering

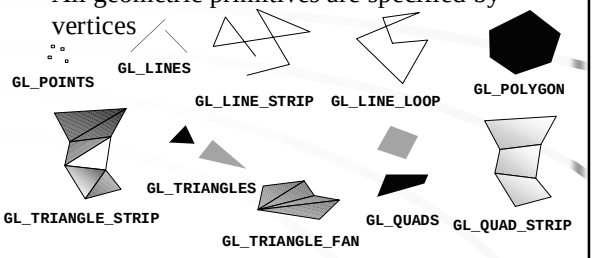
Part II

Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



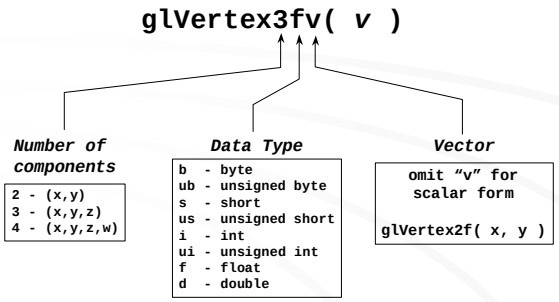
Polygon Issues

- OpenGL will only display polygons correctly that are
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- User program must check if above true
- Triangles satisfy all conditions

Simple Example

```
• 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();
• }
```

OpenGL Command Formats



Specifying Geometric Primitives

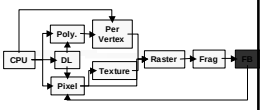
- Primitives are specified using

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

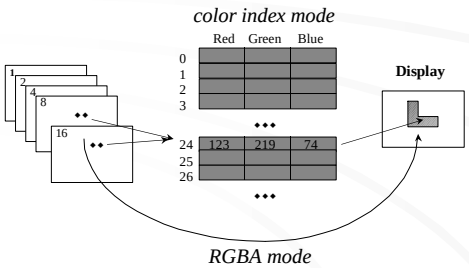
  - *primType* determines how vertices are combined

```
GLfloat red, green, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

OpenGL Color Models

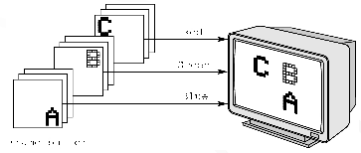


- RGBA or Color Index



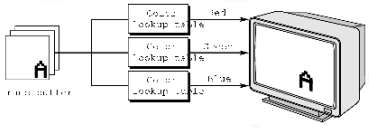
RGB color

- Each color component stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Note in **`glColor3f`** the color values range from 0.0 (none) to 1.0 (all), while in **`glColor3ub`** the values range from 0 to 255



Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive
    - Need more colors for shading



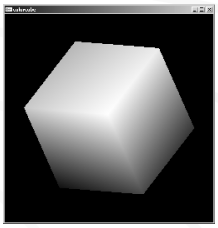
Color and State

- The color as set by **`glColor`** becomes part of the state and will be used until changed
  - Colors and other attributes are not part of the object but are assigned when the object is rendered
- We can create conceptual *vertex colors* by code such as

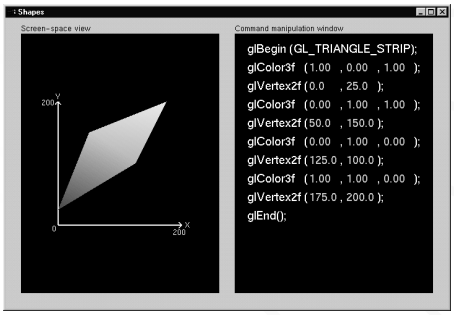
```
glColor
glVertex
glColor
glVertex
```

Smooth Color

- Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is *flat shading*
  - Color of first vertex determines fill color
- **`glShadeModel (GL_SMOOTH)`** or **`GL_FLAT`**

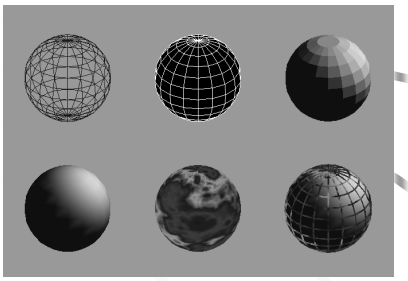


Shapes Tutorial



Controlling Rendering Appearance

- From
- Wireframe
- to Texture
- Mapped



OpenGL's State Machine

- All rendering attributes are encapsulated in the OpenGL State
  - rendering styles
  - shading
  - lighting
  - texture mapping

Manipulating OpenGL State

- Appearance is controlled by current state

```
for each ( primitive to render ) {
    update OpenGL state
    render primitive
}
```

- Manipulating vertex attributes is most common way to manipulate state

```
glColor*() / glIndex*()
glNormal*()
glTexCoord*()
```

Controlling current state

- Setting State

```
glPointSize( size );
glLineStipple( repeat, pattern );
glShadeModel( GL_SMOOTH );
```

- Enabling Features

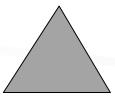
```
glEnable( GL_LIGHTING );
glDisable( GL_TEXTURE_2D );
```

Three-dimensional Applications

- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
  - Not much changes
  - Use **`glVertex3*( )`**
  - Have to worry about the order in which polygons are drawn or use hidden-surface removal
  - Polygons should be simple, convex, flat

Sierpinski Gasket (2D)

- Start with a triangle

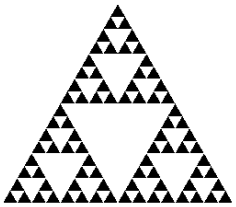


- Connect bisectors of sides and remove central triangle

- Repeat

Example

- Five subdivisions



The gasket as a fractal

- Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
- As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity
- This is not an ordinary geometric object
  - It is neither two- nor three-dimensional
- It has a *fractal* (fractional dimension) object

Gasket Program

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

/* a point data type
typedef GLfloat point2[2];

/* initial triangle */

point2 v[]={{-1.0, -0.58}, {1.0, -0.58},
  {0.0, 1.15}};

int n; /* number of recursive steps */
```

Draw a triangle

```
void triangle( point2 a, point2 b,
  point2 c)

/* display one triangle  */
{
    glBegin(GL_TRIANGLES);
      glVertex2fv(a);
      glVertex2fv(b);
     glVertex2fv(c);
    glEnd();
}
```

Triangle Subdivision

```
void divide_triangle(point2 a, point2 b, point2 c, int
  m)
{
/* triangle subdivision using vertex numbers */
    point2 v0, v1, v2;
    int j;
    if(m>0)
     {
        for(j=0; j<2; j++) v0[j]=(a[j]+b[j])/2;
        for(j=0; j<2; j++) v1[j]=(a[j]+c[j])/2;
        for(j=0; j<2; j++) v2[j]=(b[j]+c[j])/2;
        divide_triangle(a, v0, v1, m-1);
        divide_triangle(c, v1, v2, m-1);
        divide_triangle(b, v2, v0, m-1);
    }
    else(triangle(a,b,c));
 /* draw triangle at end of recursion */
}
```

Display and Init Functions

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    divide_triangle(v[0], v[1], v[2], n);
    glFlush();
}

void myinit()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-2.0, 2.0, -2.0, 2.0);
    glMatrixMode(GL_MODELVIEW);
    glClearColor (1.0, 1.0, 1.0,1.0)
    glColor3f(0.0,0.0,0.0);
}
```

main Function

```
int main(int argc, char **argv)
{
   n=4;
   glutInit(&argc, argv);

 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(500, 500);
   glutCreateWindow("2D Gasket");
   glutDisplayFunc(display);
   myinit();
   glutMainLoop();
}
```

Moving to 3D

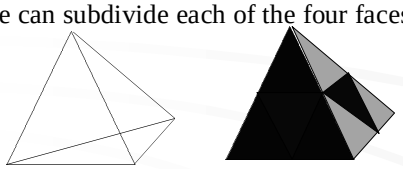
- We can easily make the program three-dimensional by using

`typedef Glfloat point3[3]`
`glVertex3f`
`glOrtho`

- But that would not be very interesting
- Instead, we can start with a tetrahedron

3D Gasket

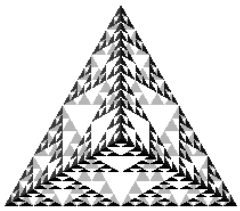
- We can subdivide each of the four faces



- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedtra

Example

after 5 interations



triangle code

```
void triangle( point a, point b, point
  c)
{
    glBegin(GL_POLYGON);
       glVertex3fv(a);
       glVertex3fv(b);
       glVertex3fv(c);
    glEnd();
}
```

subdivision code

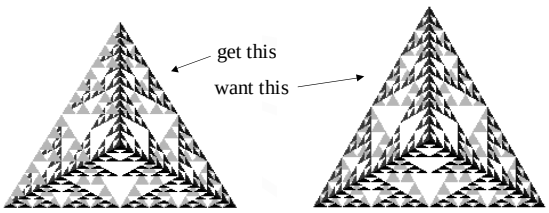
```
void divide_triangle(point a, point b, point
 c, int m)
{
    point v1, v2, v3;
    int j;
    if(m>0)
    {
        for(j=0; j<3; j++)
 v1[j]=(a[j]+b[j])/2;
        for(j=0; j<3; j++)
 v2[j]=(a[j]+c[j])/2;
        for(j=0; j<3; j++)
 v3[j]=(b[j]+c[j])/2;
        divide_triangle(a, v1, v2, m-1);
        divide_triangle(c, v2, v3, m-1);
        divide_triangle(b, v3, v1, m-1);
    }
    else(triangle(a,b,c));
}
```

tetrahedron code

```
void tetrahedron( int m)
{
    glColor3f(1.0,0.0,0.0);
    divide_triangle(v[0], v[1], v[2],
 m);
    glColor3f(0.0,1.0,0.0);
    divide_triangle(v[3], v[2], v[1],
 m);
    glColor3f(0.0,0.0,1.0);
    divide_triangle(v[0], v[3], v[1],
 m);
    glColor3f(0.0,0.0,0.0);
    divide_triangle(v[0], v[2], v[3],
 m);
}
```

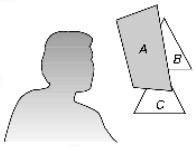
Almost Correct

- Because the triangles are drawn in the order they are defined in the program, the front triangles are not always rendered in front of triangles behind them



Hidden-Surface Removal (HSR)

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a *hidden-surface* method called the *z*-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



Using the *z*-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- It must be
  - Requested in **`main.c`**
    - **`glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)`**
  - Enabled in **`init.c`**
    - **`glEnable(GL_DEPTH_TEST)`**
  - Cleared in the display callback
    - **`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`**