Computer Graphics

Gerda Kamberova

Outline

- OpenGL
  - Overview
  - main loop, program structure
  - Interaction supported through GLUT
  - Setting up display window
  - 2D viewing, setting up viewport
- Program structure
  - Sierpinski algorithm for generating the gasket
  - Recursive algorithm for generating the gasket
- OpenGL
  - Geometric primitives and attributes
  - Text
- GLUT and GLU shapes

OpenGL

- High performance, window independent graphics API to graphics hardware
- Designed by SGI, 1982
- No commands for windowing tasks and user interaction
- Only low-level commands
- A shape/object is modeled from geometric primitives
- Objects are arranged in a 3D world
- Objects are assigned various attributes
- Renderer

OpenGL

- Portable, device independent
- Uses 2D and 3D coordinates (internally all 4D)
- Graphics functions for specifying
  - Primitives and their attributes
  - Geometric/Modeling transformations
  - Viewing transformations
  - Input functions (usually developed in a separate library) to support and process input to interactive graphics
  - System, control functions

OpenGL Overview

- C library of about 350 functions
- All function names begin with `gl`
- All constant names begin with `GL_`
- World coordinate system: `(x,y,z)` is right-handed, x-to-y (counter clockwise), z-towards viewer (direction of thumb)
- Graphics objects are sent to display in two modes
  - Immediate mode: send object for display as soon as the command defining it is executed. The object is not retained in the memory, just the image of the object is in the FB.
  - Retained mode: object description is defined once, the description is put in a display list. Display lists are good when objects are not changing too rapidly.

OpenGL: Matrix Modes

- Two states of the system characterized by matrix modes : model-view and projection

```
glMatrixMode(GL_PROJECTION)
glLoadIdentity();
gluOrtho2D(0.0, 500.0, 0.0, 500.0)
glMatrixMode(GL_MODELVIEW)
```

- Viewing rectangle with lower-left corner at the origin, size 500x500. Defined in viewing coordinates.
- OpenGL is a state machine. Various states remain in effect until you change them

Control Functions, GLUT

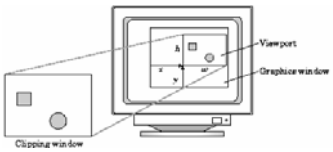
- Heavy use of OpenGL Utility Toolkit (GLUT) to
  - Interface with window system
  - Window management (create,destroy, position,resize)
  - Interaction
    - menu management
    - register callback functions
    - Color model management
  - Simple shapes (cube, sphere, cone, torus)
- (Display) window is the rectangle area on our display
- Screen coordinates are measured in pixels
- Our window exists within a window system
- Reference is to top left corner of the screen

Display Window Management

```
glutInit(&argc,argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB); /* default */
glutInitWindowSize(500,500); /* 500 x 500 pixel window */
glutInitWindowPosition(0,0); /* place window top left on display */
glutCreateWindow("Sierpinski Gasket"); /* window title */
```

- Five routines necessary for initializing a display window
- Viewport is mapped into the whole display window, by default.

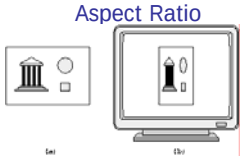
Viewports



Viewports

- Viewport is a rectangular area of the display window. Default is the entire window
- Can set to a smaller size to avoid distortion
  void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)
  where (x,y) is the position of lower left corner of viewport in the display window.
- The arguments w an h are width and height of the viewport in pixels.

Aspect Ratio



- Clipping window (left) aspect ratio (set by glOrtho() or gluOrtho2D())

  gluOrtho2D(0.0, 600.0, 0.0, 400);
- If you want to avoid distortion:

  glutInitWinowSize(300, 200); //same AR

  keep same AR of display window and clipping window, or

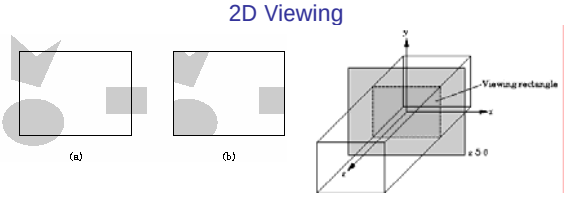
  if those are different set a viewport with AR as clipping window

  glViewport(0,0,300,200);

Viewing

- Viewing volume – the volume in 3D world that is being viewed.
- By default, an orthographic projection is assumed with a viewing volume a cube size 2x2x2 centered at the origin of the camera (I.e. the camera is at the center of the viewing volume)
- For 2D viewing, by default the viewing (clipping) window is 2x2 square in the plane z=0
- For 2D viewing, gluOrtho2D(), sets the coordinates of the vertices of the viewing window.
- For 3D othrographic projection, the viewing volume is a parallelepiped, glOtrho() is used to specify it.

2D Viewing

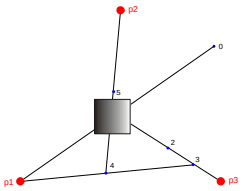


(x,y,z) is the camera coordinate system

By default, world and camera coordinate systems are aligned, but their z axes are in opposite directions

How thus this effect the way you model 2 objects and select a viewing rectangle?

Example



- The model we have in mind are the three points P1(-5,-4), P2(0, 7), P3(6,-4)
- Default view in 2x2 shaded rectangle centered at (0,0)
- P1 , P2, P3 are outside of the viewing rectangle, nothing can be imaged
- Need to adjust the projection parameters, set new viewing rectangle such that the points are in view
- gluOrto2D(-10,10, -5, 10)
- But in order for gluOrtho to work you need to be in Projection mode:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrto2D(-10,10, -5, 10);
glMatrixMode(GL_MODELVIEW);
```


Getting Things On the Screen

- Immediate Mode Graphics – primitives are displayed as soon as they are defined
- Interactive Graphics respond to events
- `glutMainLoop()` processes events as they occur. Currently, the only event we have seen is the need to redraw the display. That event is processed be a display callback function.
- No events, then wait forever. Kill with "`Cntrl C`"

Display Callback Function

- Graphics are sent to the screen through this function:

```
void glutDisplayFunc(void (*func)(void))
```

- Called whenever GLUT determines that the contents of the window needs to be redisplayed
- All routines that need to redraw must go or be called directly or indirectly from the display callback function

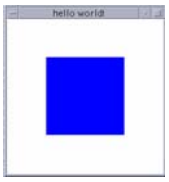
Program Structure

- **`main`** function consists of calls to initialize GLUT functions. Names required callbacks
- **`Display callback function`** every program must have this. Contains primitives to be redrawn. So far you have seen only name **`display`** , but you may give it any name, just keep it informative, and register it with **`gutDisplayFunc()`**
- **`myinit`** put here user initializer options. This is housekeeping.


```
/* hello.c * This is a simple, introductory OpenGL program. */
#include <GL/glut.h>
void display(void);
void init(void);

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT); /* clear all pixels  */

 /* draw blue polygon (square)  */
   glColor3f (0.0, 0.0, 1.0);
   glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
/* don't wait! start processing buffered OpenGL routines  */
   glFlush ();
}
```

```
void init (void)
{
/* select clearing color     */
   glClearColor (1.0, 1.0, 1.0, 1.0);
/* set type of camera projection used, orthographic in this example */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
}
int main(int argc, char** argv)
{
/* GLUT negotiates with the window system:
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("hello world!");

   init ();   /* you write this function, set up camera and view parameters */
   glutDisplayFunc(display); /* you write this, put all drawing commands here */
   glutMainLoop();   /* waits for events, display redraws in this case, and process*/
   return 0;   /* ANSI C requires main to return int. */
}
```


Hello World!



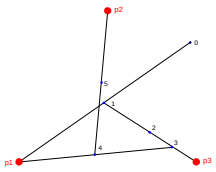
Homework

- Read Angel Ch 2, OpenGL primer Ch 2

Sierpinski Algorithm

- Given 3 vertices of a triangle
- Starting at any point P on the plane, initial point, choose a vertex V randomly
- Draw a point M half way P and V
- M now becomes the current point, P
- Repeat this process, each time drawing a point halfway between the current point and a randomly chosen vertex.

Sierpinski Algorithm




```
Clear window.
Set the 3 vertices.

Pick up initial point P.

Repeat:
  choose at random vertex V.
  P = midpoint of V..
  Draw P.
Until done
```


Sierpinski Gasket

- There are "large" areas where points will never be drawn

Example: Serpinski's algorithm implementation

```
#include <GL/glut.h>
void myinit(void);
void display(void);
void main(int argc, char** argv)
{
/* Standard GLUT initialization */
    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB); /* default */
    glutInitWindowSize(500,500); /* 500 x 500 pixel window */
    glutInitWindowPosition(0,0); /* place window top left corner */
    glutCreateWindow("Sierpinski Gasket"); /* window title */
    glutDisplayFunc(display); /* display callback registered */
    myinit(); /* set attributes (state variables) */
    glutMainLoop(); /* enter event loop */
}
```



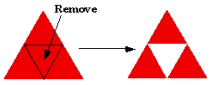
```
void myinit(void)
{

/* attributes */

      glClearColor(1.0, 1.0, 1.0, 1.0); /* white background */
      glColor3f(1.0, 0.0, 0.0); /* draw in red */

/* set up viewing, camera */
/* 500 x 500 clipping window with lower left coner at (0.0,0.0), world
coordinates */
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0.0, 500.0, 0.0, 500.0);
      glMatrixMode(GL_MODELVIEW);
}
```



```
void display( void )
{
    /* define a point data type */
    typedef GLfloat point2[2];    // you could define a class point2
    point2 vertices[3]={{0.0,0.0},{250.0,500.0},{500.0,0.0}}; /* A triangle */
    int i, j, k;
    int rand();             /* standard random number generator */
    point2 p ={75.0,50.0};  /* An arbitrary initial point inside triangle */
    glClear(GL_COLOR_BUFFER_BIT);  /*clear the window */
    /* compute and plots 5000 new points */
    for( k=0; k<5000; k++)
    {
        j=rand()%3; /* pick a vertex at random */
      /* Compute point halfway between selected vertex and old point */
        p[0] = (p[0]+vertices[j][0])/2.0;
        p[1] = (p[1]+vertices[j][1])/2.0;
       /* plot new point */
         glBegin(GL_POINTS);
              glVertex2fv(p);
         glEnd();
      }
    glFlush(); /* show buffer */
}
```

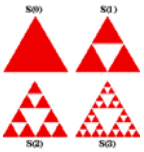

Sierpinski Gasket via Recursive Subdivision

- Start with solid triangle **S(0)**
- Divide this into 4 smaller equilateral triangles using the midpoints of the three sides of the original triangle as the new vertices
- Remove the interior of the middle triangle (that is, do not remove the boundary) to get **S(1)**



Sierpinski Gasket

- Now repeat this procedure on each of the three remaining solid equilateral triangles to obtain **S(2)**.



- Continue to repeat the construction to obtain a decreasing sequence of sets

$$S(0) \supseteq S(1) \supseteq S(2) \supseteq S(3) \cdots$$

Recursive Approach

- Use polygons and fill solid areas
- No need for random number generating
- Recursive program called from display()

```
divide_triangle(a,b,c,m)
//a,b,c are the vertices,m controls depth
   if m==0 draw triangle(a,b,c)
else
   find the midpoints of each side.
   call divide_triangle for each of the 3
     smaller triangles, with depth m-1
```

- Only the smallest triangles will be drawn

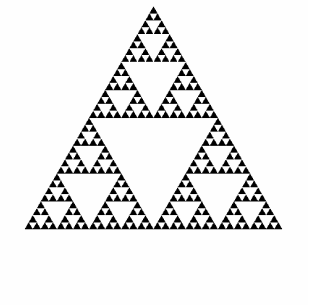
Recursive Sierpinski

```
void triangle(point2 a, point2 b, oint2 c)
/* display one triangle  */
{
    glBegin(GL_TRIANGLES);
       glVertex2fv(a);
       glVertex2fv(b);
       glVertex2fv(c);
    glEnd();
}
```

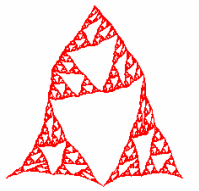


```
void divide_triangle(point2 a, point2 b, point2 c, int m)
{
/* triangle subdivision using vertex numbers */
    point2 v0, v1, v2;
    int j;
    if (m>0)
    {   /* generate mid points of the sides */
        for(j=0; j<2; j++) v0[j]=(a[j]+b[j])/2;
        for(j=0; j<2; j++) v1[j]=(a[j]+c[j])/2;
        for(j=0; j<2; j++) v2[j]=(b[j]+c[j])/2;
        /* make recursive calls for each of the smaller triangles */
        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 */
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    divide_triangle(v[0], v[1], v[2], n);
    glFlush();
}
```


After 5 Subdivisions



Randomized



Graphics Standard: primitives and attributes

- Primitives may include
  - Point
  - Line/polyline
  - Text
  - Marker
  - Polygon
  - Rectangle
  - Circle/arc
  - Curve, etc.
- Attribute: any property that determines how a geometric primitive is to be rendered

Graphics Standard

- Primitives and attributes, examples

Primitive: Attribute	Line	Text	Marker	polygon
color	X	X	X	X
line style	X			
line width	X			
pen	X	X	X	
font		X		
size		X		
Fill style				X
Edge style				X

OpenGL Primitives

- Point: 2D or 3D Vertex (internally 4 coord)
- Command suffixes specify data type

Suffix	Number bits	C- type	OpenGL-type
b	8	char	GLbyte
s	16	short int	GLshort
i	32	long int	GLint
f	32	float	Glfloat
d	64	double	GLdouble
ub	8	unsigned char	GLubyte
us	16	unsigned short	GLushort
ui	32	unsigned long	GLuint

OpenGL primitives: Point

- Points are referred as vertices
- For 2D vertex, z-coord is 0

glVertex{234}{sifd}[v](coordinates)

Examples:

```
glVertex2s(2,3)
glVertex3i(10, -5, 100)
glVertex4f(4.0, 6.0,21.5, 2.0)

GLdouble dpt[3]={5.0, 9.0, 11.6};
glVertex3dv(dpt)
```


OpenGL primitives: lines and ploygons

- Primitive object is defined as a sequence of vertices between glBegin() and glEnd()

```
glBegin(GLenum_mode );
   define primitives here
glEnd();
```

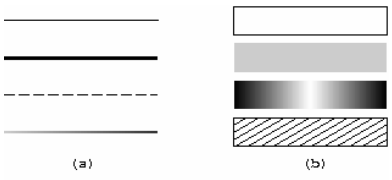
The mode above could be:
GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
GL_POLYGON, GL_TRIANGLES, GL_QUADS,
GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN

OpenGL permits only simple, convex polygons

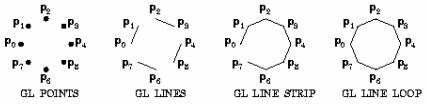
Approximating curves

```
#define PI 3.1415926535897

GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
    for (i = 0; i < circle_points; i++) {
       angle = 2*PI*i/circle_points;
       glVertex2f(cos(angle), sin(angle));
    }
glEnd();
```


Attributes



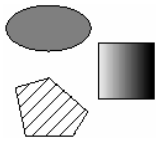
OpenGL: Primitives



- Points (GL_POINTS)
- Line Segments (GL_LINES) – successive pairs of vertices interpreted as the endpoints of individual segments
- Polylines (GL_LINE_STRIP) – successive vertices (and line segments) are connected. Can be a closed path.

Polygons Basics

- Polygon – area object that has a border that can be described by a single line loop.
- Used to approximate curved surfaces



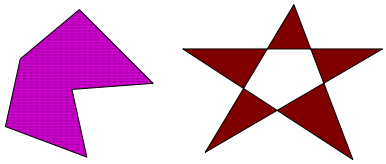
Polygonal approximation of a surface

"Canonical" Polygon

- Simple – no pair of edges cross each other
- Convex – the line segment between any two pints on the polygon is entirely inside the polygon
- Flat – all vertices are coplanar

Polygon Examples

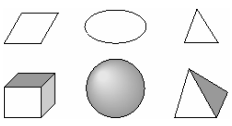
- Simple – no pair of edges cross each other



Simple **Nonsimple**

Convex Shapes

- Convex – all points on the line segment between any two points inside the object , or on its boundary, are inside the object
- Here are some convex shapes



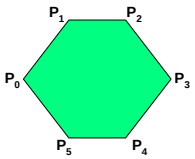
OpenGL Polygon Examples

convex

Non-convex.
Left is also nonsimple

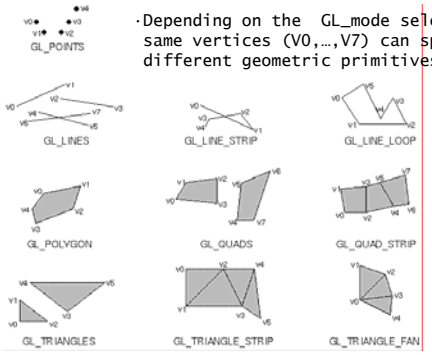
Polygons

- GL_POLYGON - the edges are the same as they would be if we used line loops
- Inside, Outside separated by "widthless" edge
- Have front and back side: when walking counter clockwise along the boundary front is towards viewer:
- In OpenGL the order of the vertices of the polygon on the right must be given as: P0,P5,P4,P3,P2,P1



GL_POLYGON

OpenGL Primitives



·Depending on the GL_mode selected same vertices (V0,…,V7) can specify different geometric primitives

Attributes of the OpenGL geometric primitives

- Attributes are defined by the current state of the system, i.e., objects are drawn with the current attributes
- Point: point size
  - `glPointSize(GLfloat size)`

Attributes of the OpenGL geometric primitives

- Line
  - Line width
    - `glLineWidth(Glfloat size)`
  - Enable antialiasing
    - `glEnable(GL_LINE_SMOOTH), not in Mesa`
  - Enable line stipple
    - `glEnable(GL_LINE_STIPPLE)`
    - `glLineStipple(GLint factor, Glushort ptrn)` `ptrn` 16-bit binary sequence of 0'a and 1's, `factor` scales the pattern.

Attributes of the OpenGL geometric primitives

- Polygon
  - Polygon face: front/back, which face to draw
    - `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`
  - Polygon mode for drawing
    - `GL_POINT, GL_LINE, GL_FILL`
  - Enable polygon stipple
    - `glEnable(GL_POLYGON_STIPPLE)`
    - `glPolygonStipple(Glubyte *mask)` `mask` is a 2D binary pattern
    - `glEdgeFlag(Glboolen flag)` applies only to triangles, quads, and polygons.

Text

- Fonts – families of type faces:

Times Roman - ABCDabcd123

Courier - ABCDabcd123

Arial - ABCDabcd123

Century - ABCDabcd123

Comic Sans MS - ABCDabcd123

Stroke Text

- Constructed from other graphics primitives
- Vertices define line segments; curves outline each character
- Can be manipulated by transformations (translate, scale, rotate)

Stroke Text

Raster Text

- Characters defined as rectangles of bits called bit blocks
- Placed in frame buffer by bitblt operation
- Increase size only by replicating pixels
- Cannot be scaled gracefully, look blocky
- glutBitmapCharacter(GLUT_BITMAP_8_BY_13, c)

Raster Text

GLUT (GL Utility Toolkit)

- Interface with window system
- Window management (create,destroy, position,size)
- Interaction
  - menu management
  - register callback functions
  - Color model management
- GLUT shapes
  - `glutSolidCube, glutWireCube`
  - `glutSolidSphere, GlutWireSphere`

  given radius and #slices (in z axis)
  - `Cone:` solid and wire versions
  - `Torus:` solid and wire versions
  - `Teapot:` solid and wire versions

GLU

- GLU (Graphics Library Utility)
- Higher level functions
- High-level transformations
  - Projection transformations
  - World-to-viewing coordinates transformations
- Functions for simple 3D objects
  - `Spheres`
  - `Open cylinders`
  - `Disks`
  - `Polygon tessalation, meshing`
  - `Quardics, splines and surfaces`
  - `NURBS curves and surfaces`
  - `Manipulation of images for texture mapping`

Open Inventor

- Built on top of OpenGL
- 3D toolkit for interactive 3D graphics
- Scene database; hierarchical representation of 3D scenes using scene graphs
- Primitive object: a node with fields for various values (shape, camera, light source, transformation)
- Manipulator , used to manipulate objects
- Scene manipulator, e.g., material editor, viewer
- 3D interchange file format for 3D objects
- Animator

IRIS Performer

- Combines OpenGL and Open Inventor
- Toolkit for visual simulation and virtual reality
- Supports graphics and database operations
  - Optimized graphics primitives
  - Shared memory
  - Database hierarchy
  - Multiprocessing
  - Morphing