-   -

Graphics Programming

Chapter 2

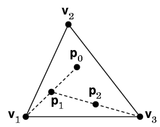
■ Introduction:
- Our approach is programming oriented.
- Therefore, we are going to introduce you to a simple but informative problem: the Sierpinski Gasket
- The functionality introduced in this chapter is sufficient to allow you to write sophisticated two-dimensional programs that do not require user interaction.

---

1. The Sierpinski Gasket

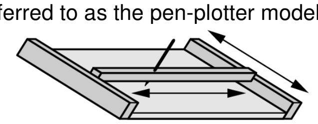
- This problem has a long history and is of interest in areas such as fractal geometry.
  - It can be defined recursively and randomly; in the limit, however, it has properties that are not at all random.
- Assume that we start with 3 points on the plane (a triangle)

- The construction proceeds as follows:
  - 1. Pick an initial point at random inside the triangle
  - 2. Select one of the three vertices at random
  - 3. Find the point halfway between the point and the vertex
  - 4. Mark/Draw that half-way point
  - 5. Replace the initial point with this new point
  - 6. Go to step 2



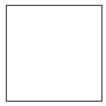
- So, what would our code look like?
  - initialize()
  - for(some_number_of_points)
  - {
  - pt=generate_a_point();
  - display_the_point(pt);
  - }
  - cleanup();
- Although our OpenGL code might look slightly different, it will almost be this simple.
- So, let's look at generating and displaying points.

- 1.1 The Pen-Plotter Model
  - Historically, most early graphics systems were two-dimensional systems. The conceptual model that they used is now referred to as the pen-plotter model.



  - Various API's - LOGO, GKS, and PostScript -- all have their origins in this model.

- The user works on a two-dimensional surface of some size
  - The following code could generate the first figure:
    - moveto(0,0);
    - lineto(1,0);
    - lineto(1,1);
    - lineto(0,1);
    - lineto(0,0)



(a)

- For certain applications, such as page layout in the printing industry, systems built on this model work well.
- We are more interested, however, in the three-dimensional world.

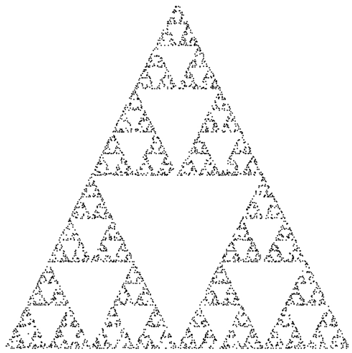
- As we saw in Chapter 1 we could do projections of the 3D points onto the 2D plane and plot with a pen.
- We prefer, however, to use an API that allows users to work directly in the domain of their problem, and have the computer carry out this projection process automatically.
- For two-dimensional applications, such as the Sierpinski gasket, we can start with a three-dimensional world, and regard two-dimensional systems as special cases.

- OpenGL has multiple forms for many functions.
  - The variety of forms allows the user to select the one best suited for their problem.
  - For a vertex function, we can write the general form
    - glVertex*
    - where * can be interpreted as two or three characters of the form nt or ntv
      - n signifies the number of dimensions (2, 3, or 4)
      - t denotes the data type (I for integer, f for float, d for double)
      - and v if present, indicates the variables are specified through a pointer to an array rather than through the argument list.

- In OpenGL, we often use basic OpenGL types, such as
  - Glfloat and Glint
  - rather than C types float and int
- So, in our application, the following are appropriate
  - glVertex2i(Glint xi, Glint yi)
  - GLVertex3f(Glfloat x, Glfloat y, Glfloat z)
- And if we use an array to store the information
  - Glfloat vertex[3];
  - glVertex3fv(vertex);

- Vertices can define a variety of geometric objects
  - A line segment can be defined as follows:
    - glBegin(GL_LINES)
    - glVertex2f(x1,y1);
    - glVertex2f(x2,y2);
    - glEnd();
  - A pair of points could be defined by:
    - glBegin(GL_POINTS)
    - glVertex2f(x1,y1);
    - glVertex2f(x2,y2);
    - glEnd();
  - Now on to the gasket.

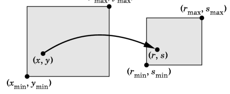

```
void display(void)
{
  point2 vertices[3] = {{0.0,0.0}, {250.0,500}, {500.0, 0.0}};
  static point2 p={75.0, 50.0};
  int j,k;
  for(k=0; k<5000;k++)
  {
    j=rand()%3;
    p[0]=(p[0]+triangle[j][0])/2;
    p[1]=(p[1]+triangle[j][1])/2;
    glBegin(GLPOINTS);
      glVertex2fv(p);
    glEnd();
  }
  glFlush();
}
```



- We have now written the core of the program. But we still have to worry about issues such as:
  - 1. In what color are we drawing?
  - 2. Where on the screen does our image appear?
  - 3. How large will the image be?
  - 4. How do we create an area on the screen - a window - for our image?
  - 5. How much of our infinite pad will appear on the screen?
  - 6. How long will the image remain on the screen?

■ 1.2 Coordinate Systems

- Originally, graphics systems required the user to specify all information, such as vertex locations, directly in units of the display device
- The advent of device independent graphics freed application programmers from worrying about the details of input and output devices.
- At some point the values in the world coordinates must be mapped into device coordinates. But the graphics system, rather than the user, is responsible for this task.



2. The OpenGL API

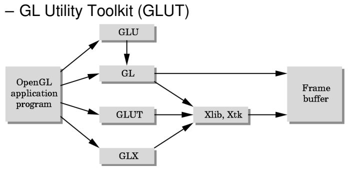
- Before completing our program, we describe the OpenGL API in more detail.
- In this chapter, we concentrate on how we specify primitives to be displayed;
  - We leave interaction to Chapter 3
- Note:
  - Our goal is to study computer graphics; we are using an API to help us attain that goal.
  - Consequently, we do not present all OpenGL functions

2.1 Graphics Functions

- We can divide the functions in the API into groups based upon their functionality:
  - 1. The primitive functions,
  - 2. Attribute functions,
  - 3. Viewing functions,
  - 4. Transformation functions,
  - 5. Input functions,
  - 6. Control functions.

2.2 The OpenGL Interface

- OpenGL function names begin with the letters gl and are stored in a library usually referred to as GL
- There are a few related libraries that we also use:
  - graphics utility library (GLU)
  - GL Utility Toolkit (GLUT)

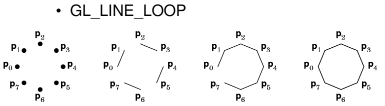


3. Primitives and Attributes

- Within the graphics community, there has been an ongoing debate:
  - API's should contain a small set of primitives (minimalist position) that ALL hardware can be expected to support.
  - API's should have everything hardware can support.

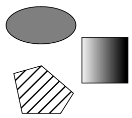
- OpenGL takes an intermediate position
  - The basic library has a small set of primitives.
  - GLU contains a richer set of objects (derived)
- The basic OpenGL primitives are specified via points in space. Thus, the programmer defined their objects with sequences of the form:
  - glBegin(type);
  - glVertex*(...);
  - ...
  - glVertex*(...);
  - glEnd();
  - The value of type specifies how OpenGL interprets the vertices

- If we wish to display line segments, we have a few choices in OpenGL.
- The primitives and their type specifications include:
  - Line Segments
    - GL_LINES
  - Polylines
    - GL_LINE_STRIP
    - GL_LINE_LOOP



■ 3.1 Polygon Basics

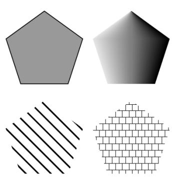
- Def: Polygon



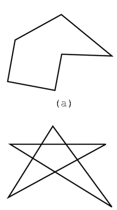
- Polygons play a special role in computer graphics because:
  - we can display them rapidly and
  - we can use them to approximate curved surfaces.

  - The performance of graphics systems is measured in the number o polygons per second that can be displayed

- We can display a polygon in a variety of ways.
  - Only its edges,
  - Fill its interior with a solid color
  - Fill its interior with a pattern.
  - We can display or not display the edges



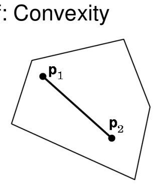
- Def: Simple Polygon



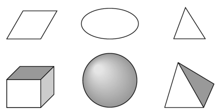
(a)

(b)

- Def: Convexity



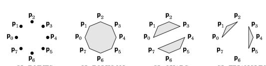
- In three dimensions polygons present a few more difficulties because they are not necessarily flat.
  - – 3 non-collinear points define a triangle ad a plane the triangle lies in.



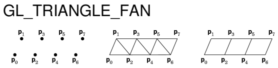
  - – Often we are almost forced to use triangles because typical rendering algorithms are guaranteed to be correct only if the vertices form a flat convex polygon.
    - In addition, hardware and software often support a triangle type that is rendered much faster than a polygon with three vertices.

■ 3.2 Polygon Types in OpenGL

- – Polygons
  - GL_POLYGON
- – Triangles and Quadrilaterals
  - GL_TRIANGLES
  - GL_QUADS



- – Strips and Fans
  - GL_TRIANGLE_STRIP
  - GL_QUAD_STRIP
  - GL_TRIANGLE_FAN



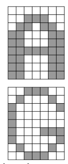
■ 3.3 Text

- Stroke Text
  - – Postscript -- font is defined by polynomial curves
  - – Requires processing power and memory
    - so printer typically has a CPU and memory

Computer

Graphics

- Raster Text
  - Simple and Fast



  - You can increase the size by replicating pixels

- OpenGL
  - Because stroke and bitmap characters can be created from other primitives, OpenGL does not have a text primitive
  - However, GLUT provides a few bitmap and stroke character sets that are defined in software.
    - glutBitmapCharacter(GLUT_BITMAP_8_BY_13, c)
  - We will return to text in Chapter 3.
  - There we shall see that both stroke and raster texts can be implemented most efficiently through display lists.

■ 3.4 Curved Objects

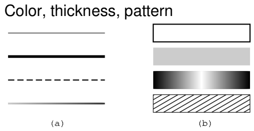
- The primitives in our basic set have all been defined through vertices.
- We can take two approaches to creating a richer set of objects.
  - 1. We can use the primitives that we have to approximate curves and surfaces.
    - If we want a circle, we can use a regular polygon of n surfaces.
    - If we want a sphere, we can approximate it with a regular polyhedron
    - More generally, we approximate a curved surface by a mesh of convex polygons (a tessellation).

– 2. The other approach, which we explore in Chapter 10, is to start with the mathematical definitions of curved objects, and then to build graphic functions to implement those objects.

– Most graphics systems provide aspects of both approaches.
  - We can use GLU for a collection of approximations to common curved surfaces.
  - And, we can write functions to define more of our own.

■ 3.5 Attributes

- In a modern graphics system, there is a distinction between what type of a primitive is and how that primitive is displayed
  - A red solid line and a green dashed line are the same geometric type, but each is displayed differently.
- An attribute is any property that determines how a geometric primitive is rendered.
  – Color, thickness, pattern

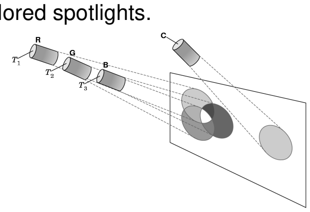


- Attributes may be associates with, or bound to, primitives at various points in the modeling rendering pipeline.
  - Bindings may not be permanent.
- In immediate mode, primitives are not stored in the system, but rather are passed through the system for possible display as soon as they are defined.
  - They are not stored in memory, and once erased from the screen, they are gone.

4. Color

- Color is one of the most interesting aspects of both human perception and computer graphics
- Color in computer graphics is based on what has become known as the three-color theory

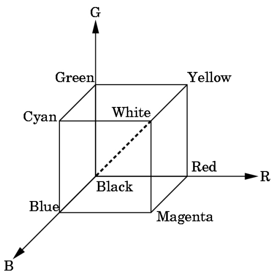
- A good analogy is to consider three colored spotlights.



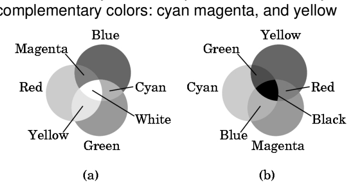
  - We can attempt to match any color by adjusting the intensities of the individual spotlights.
  - Although we might not be able to match all colors in this way, if we use red green and blue we can come close.

- The three colors stems from our eyes.
  - The color receptors in our eyes - the cones - are three different types.
  - Thus the brain perceives the color through a triplet, rather than a continuous distribution.
- The basic tenet of three-color theory:
  - if two colors produce the same tristimulus values, then they are visually indistinguishable.

- We can view a color as a point in a color solid as shown here:

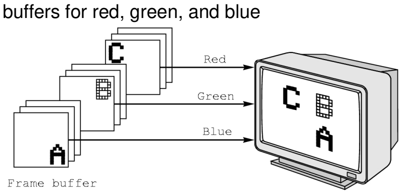


- We are looking at additive color systems because of the way computer display systems work.
  - There is also a subtractive color model which is typically used in commercial printing and painting.
  - In subtractive systems, the primaries are usually the complementary colors: cyan magenta, and yellow



■ 4.1 RGB Color

- Now we can look at how color is handled in a graphics system from the programmer's perspective -- that is, through the API
  - In the three-primary-color, additive-color RGB systems, there are conceptually separate frame buffers for red, green, and blue



- Because the API should be independent of the particulars of the hardware, we will use the color cube, and specify numbers between 0.0 and 1.0

- In OpenGL, we use the color cube as follows.
  - To draw in red, we issue the function call
    - glColor3f(1.0, 0.0, 0.0);

- Later, we shall be interested in a four-color (RGBA) system.
  - In Chapter 9, we shall see various uses of the Alpha channel, such as for creating fog effects or for combining images.
  - The alpha value will be treated by OpenGL as an opacity or transparency value.
- For now we can use it to clear our drawing window.
  - glClearColor(1.0, 1.0, 1.0, 1.0);
  - We can then use the function glClear to make the window solid and white.

■ 4.2 Indexed Color

- Many systems have frame buffers that are limited in depth.
  - If we choose a limited number of colors from a large selection, we should be able to create good quality images most of the time.

Input		Red	Green	Blue
0		0	0	0
1		2^m 2 1	0	0
.		0	2^m 2 1	0
.		.	.	.
.		.	.	.
2^k 2 1		.	.	.
		m bits	m bits	m bits

  - Historically color-index mode was important because it required less memory for the frame buffer.
  - For most of our code we will use a standard RGB model.

■ 4.3 Setting of Color Attributes

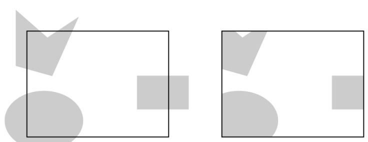
- The first color to set is teh clear clolr
  - glClearColor(1.0,1.0,1.0,1.0);
- We can select the rendering color for our points by setting the color variable
  - glColor3f(1.0,0.0,0.0);
- We can set the size of our rendered points to be 2 pixels wide, by using
  - glPointSize(2.0);
  - Note that attributes such as point size and line width are specified in terms of the pixel size.

5. Viewing

- Just as the casual photographer does not need to worry about how the shutter works or what are the details of the photochemical interaction of light and film is,
- So the application programmer only needs to worry about the specifications of the objects and the camera.

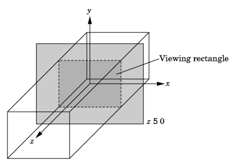
■ 5.1 Two-Dimensional Viewing

- taking a rectangular area of our two-dimensional world and transferring its contents to the display as shown:



(a) (b)

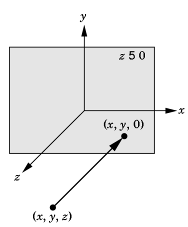
- Remember that two-dimensional graphics is a special case of three-dimensional graphics.
- Our viewing rectangle is the plane z=0 within a three-dimensional viewing volume.



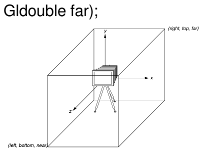
- If we do not specify a viewing volume, OpenGL uses its default, a 2x2x2 cube with the origin at the center.

■ 5.2 The Orthographic View

- This two-dimensional view is a special case of the orthographic projection (discussed more in Chapter 5)
  - points at (x,y,z) are projected to (x,y,0)



- In OpenGL, an orthographic projection is specified via
  - void glOrtho(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top, Gldouble near, Gldouble far);



  - Unlike a real camera, the orthographic projection can include objects behind the camera
    - void glOrtho2D(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top);
  - In Chapters 4 and 5 we will discuss moving the camera and creating more complex views.

■ 5.3 Matrix Modes

- The two most important matrices are
  - the model-view and
  - projection matrices.
- In Chapter 4 we will study functions to manipulate these matrices
- The following is common for setting a two-dimensional viewing rectangle:
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity();
  - gluOrtho2D(0.0,500.0, 0.0, 500.0);
  - glMatrixMode(GL_MODELVIEW);
  - This defines a 500x500 viewing rectangle, with the lower-left corner as the origin.

6. Control Functions

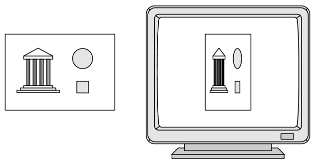
- We are almost done with our first program,
  - but we must still discuss interaction with the window and operating systems.
  - Rather than deal with these issues in detail we will look at the simple interface GLUT provides.
    - Applications produced using GLUT should run under multiple window systems.

■ 6.1 Interaction with the Window System

- Before we can open a window, there must be interaction between the windowing system and OpenGL.
  - glutInit(int *argcp, char **argv)
  - glutCreateWindow(char *title)
  - glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
  - glutInitWindowSize(480, 640);
  - glutInitWindowPosition(0,0);

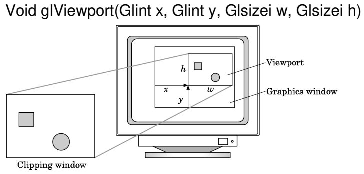
■ 6.2 Aspect Ratio and Viewports

- Def: Aspect Ratio
- If the ratio of the viewing rectangle (specified by glOrtho) is not the same as the aspect ratio specified by glutInitWindowSize, you can end up with distortion on the screen.



(a) (b)

- A viewport is a rectangular area of the display window.
  - By default, it is the entire window, but it can be set to any smaller size.
    - Void glViewport(Glint x, Glint y, Glsizei w, Glsizei h)



    - We will see further uses of the viewport in Chapter 3, where we consider interactive changes in the size and shape of the window

■ 6.3 The main, display, and myinit Functions

- In Chapter 3 we will discuss event processing, which will give us tremendous control in our programs. For now, we can use the GLUT function
  - void glutMainLoop(void);
- Graphics are sent to the screen through a function called a display callback.
  - This function is specified through the GLUT function
    - void glutDisplayFunc(void (*func)(void));

- #include <GL/glut.h>
-
- void main(int argc, char **argv)
- {
- glutInit(&argc, argv);
- glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
- glutInitWindowSize(500,500);
- glutInitWindowPosition(0,0);
- glutCreateWindow("simple OpeGL example);
- glutDisplayFunc(display);
- myinit();
- glutMainLoop();
- }

■ 6.4 Program Structure

- Every program we write will have the same structure as our gasket program.
  - We will always use the GLUT toolkit
  - The main function will then consist of calls to GLUT functions to set up our window(s)
  - The main function will also name the required callbacks
    - every program must have a display callback
    - most will have other callbacks to set up interaction.
  - The myinit will set up user options
    - (usually calls to GL and GLU library functions.)

7. The Gasket Program

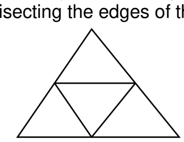
- Using the previous program as our base
  - We can now write the myinit function and the display function for our Sierpinski gasket
  - We will draw red points on a white background
  - all within a 500x500 square.

- void myinit(void)
- {
-   glClearColor(1.0,1.0,1.0,0.0;
-   glColor3f(1.0,0.0,0.0);
-   glMatrixMode(GL_PROJECTION);
-   gluLoadIdentity();
-   gluOrtho2D(0.0,500.0,0.0,500.0);
-   glMatrixMode(GL_MODELVIEW);
- }

- void display(void)
- {
-   typedef Glfoat point2[2];
-   point2 vertices[3]={{0.0,0.0},{250.0,500.0},{500.0,0,0}};
-   int i,j,k;
-   point2 p={75.0,50.0};
-   glClear(GL_COLOR_BUFFER_BIT);
-   for(k=0;k<5000;k++){
-     j=rand()%3;
-     p[0]=(p[0]+vertices[j][0])/2.0;
-     p[1]=(p[1]+vertices[j][1])/2.0;
-     glBegin(GL_POINTS)
-       glVertex2fv(p);
-     glEnd();
-   }
-   glFlush();
- }

8. Polygons and Recursion

- We can generate the gasket a different way bisecting the edges of the triangle



- and doing this over recursively until we reach the desired subdivision level

- Let us start our code with a simple function that draws a single triangular polygon with three arbitrary vertices.

```
• void triangle(point2 a, point2 b, point2 c)
• {
•   glBegin(GL_TRIANGLES);
•     glVertex2fv(a);
•     glVertex2fv(b);
•     glVertex2fv(c);
•   glEnd();
• }
```



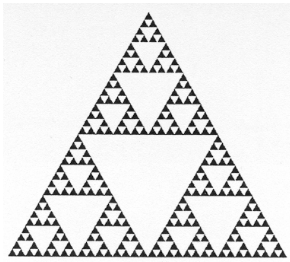
```
• void divide_triangle(point2 a, point2 b, point2 c, int k)
• {
•   point2 ab, ac bc;
•   int j
•   if(k>0){
•     // compute the midpoints of the sides
•     for(j=0;j<2;j++) ab[j]=(a[j]+b[j])/2;
•     for(j=0;j<2;j++) ac[j]=(a[j]+c[j])/2;
•     for(j=0;j<2;j++) bc[j]=(b[j]+c[j])/2;

•     // subdivide all but the inner triangle
•     divide_triangle(a,ab,ac,k-1);
•     divide_triangle(c,ac,bc,k-1);
•     divide_triangel(b,bc,ab,k-1);
•   }
•   else triangle(a,b,c);
• }
```


- The display function is now almost trivial. It uses global value of n determined by the main program to fix the number of subdivisional steps.

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

- Note:
  - often we have no convenient way to pass variables to OpenGL functions and callbacks other than through global parameters.
  - Although we prefer not to pass values in such a manner, because the form of these functions is fixed, we have no good alternative.

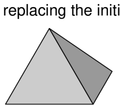
- Here is the triangle when there are 5 subdivisions.



CS 480/680 Chapter 2 -- Graphics Programming 64

9. The Three-Dimensional Gasket

- We have argued:
  - that two-dimensional graphics is a special case of three-dimensional graphics
  - But we have not yet seen a true three-dimensional program.
- So, lets convert the Gasket program to three-dimensions.
  - We start by replacing the initial triangle with a tetrahedron

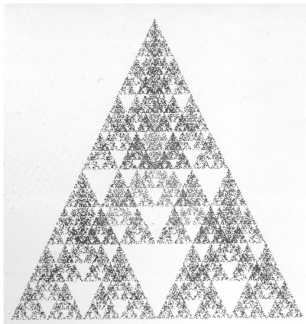


CS 480/680 Chapter 2 -- Graphics Programming 65

■ 9.1 Use of Three-Dimensional Points

- The required changes are primarily in the function display
  - typedef Glfloat point3[3];
  - point3 vertices[4]{{0.0,0.0,0.0}, {250.0,500.0,100.0}, {500.0,250.0,250.0}, {250.0,100.0,150.0)}};
  - point3 p={250.0, 100.0, 250.0};
- We will also color the points to help visualize its location.

CS 480/680 Chapter 2 -- Graphics Programming 66



■ 9.2 Use of Polygons in Three Dimensions

- Following our second approach, we note that the faces of a tetrahedron are the four triangles determined by its four vertices.
- Our triangle function changes to:

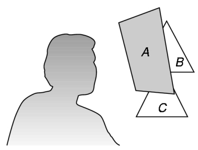
```
• void triangle(point3 a, point3 b, point3 c)
• {
•   glBegin(GL_POLYGON);
•     glVertex3fv(a);
•     glVertex3fv(b);
•     glVertex3fv(c);
•   glEnd();
• }
```


- Our divide triangle function just changes from point2 to point3 parameters.
- We then generate our subdivided tetrahedron

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

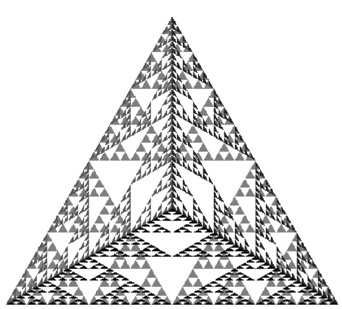
■ 9.3 Hidden-Surface Removal

- If you execute the code we just wrote, you might be confused
  - – the program draws the triangles in the order specified by the recursion, not by the geometric relationship between the triangles.
  - – Each triangle is drawn (filled) in a solid color and is drawn over those triangles already on the display.
- The issue is hidden surface removal



- For now, we can use the z-buffer algorithm supported by OpenGL
  - glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  - glEnable(GL_DEPTH_TEST);
- – we must also clear the Depth Buffer in the display function:

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   tetrahedron(n);
   glFlush();
}
```

10. Summary

- In this chapter, we introduced the OpenGL API
- The Sierpinski gasket provides a nontrivial beginning application
  - more details abut Fractal Geometry are given in Chapter 11.
- The historical development of graphics API's and graphical models illustrates the importance of starting in three dimensions.

11. Suggested Readings

- Pen Plotter API of Postscript and LOGO
- GKS, GKS-3D, PHIGS, and PHIGS+ API's
- The X Window System
- Renderman interface
- *OpenGL Programming Guide, OpenGL Reference Manual*

Exercises -- Due next class