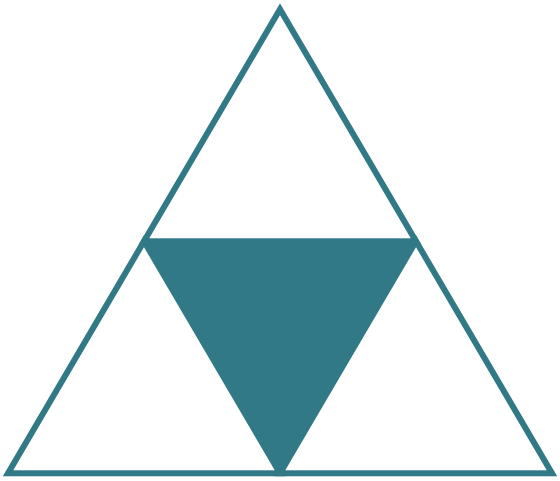
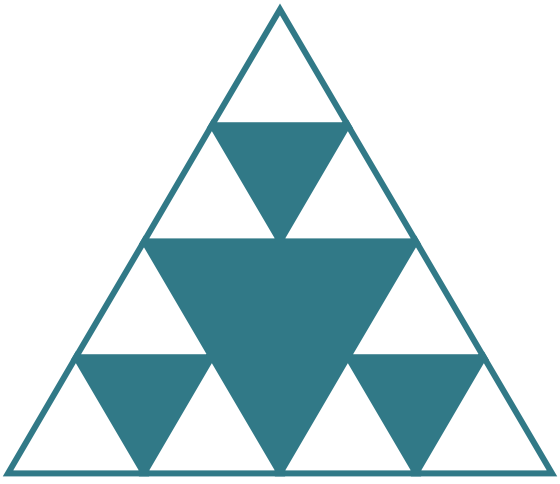
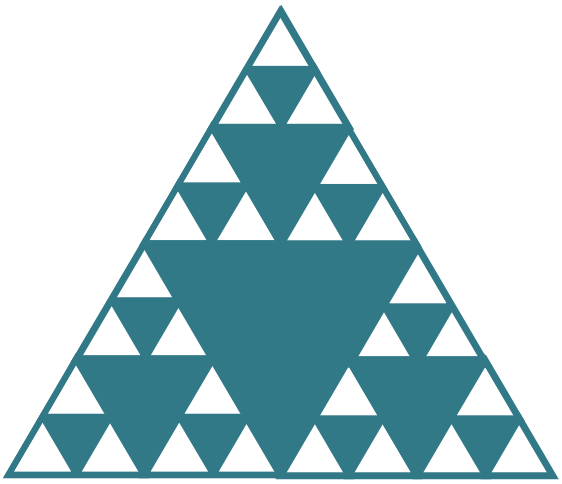
Example: 2D Sierpinksi Gasket



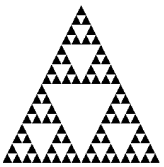
Sierpinksi Gasket



Sierpinksi Gasket



Gasket Program



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

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

/* initial triangle - global variables */

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

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

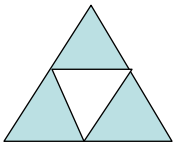
main Function

```
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();
}
```

Display and Init Functions

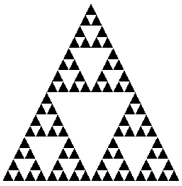
```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    divide_triangle(v[0], v[1], v[2], n);
    glFlush();
}

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);
}
```

Triangle Subdivision



```
void divide_triangle(point2 a, point2 b, point2 c, int m)
{
/* triangle subdivision using vertex coordinates */
    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 */
}
```

Draw a triangle

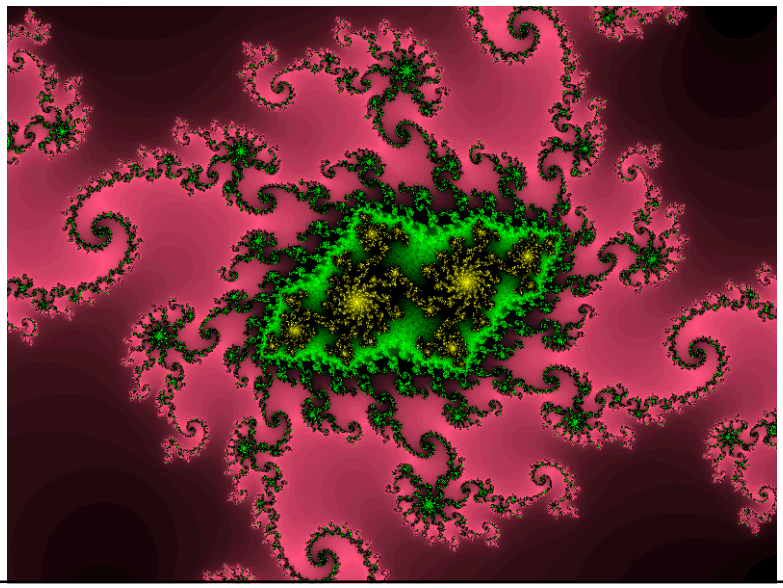


```
void triangle( point2 a, point2 b, point2 c)

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

Fractals -- Interesting Properties

- No matter how small you get there's still detail
  - Great for special effects
  - Commonly used for landscapes and mountains
  - Good for clouds

Fractal Example: Mandelbrot Set

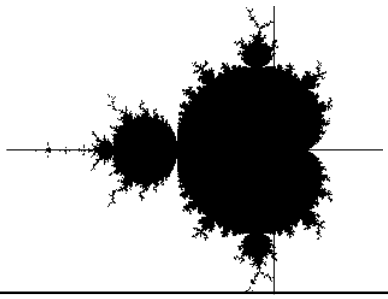


Mandelbrot Sets

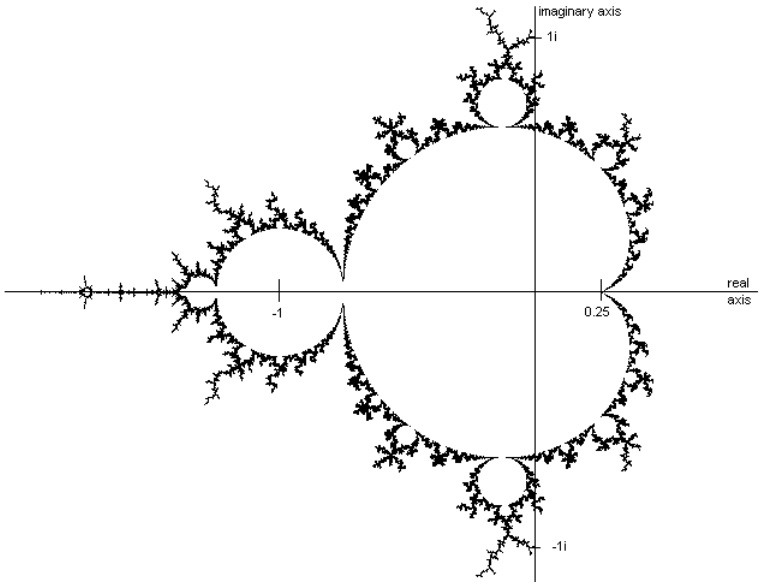
- Suppose we have the following complex function:
  - $f(x)=x^2+c$
  - Let $c=0+0.5i$ and $x_0=0$
  - Define $x_{i+1}=f(x_i)$
    - $x_0=0+0i$
    - $x_1=0+0.5i$
    - $x_2=-0.25+0.5i$
    - etc…

Convergence

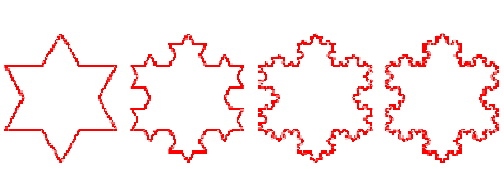
- What if we try different values of c?
  - Some tend to get smaller
  - Some tend to get grow without bound
- If we plot all c values (complex) that converge, we get the Mandelbrot Set

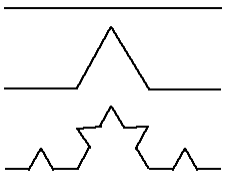


Mandelbrot Set Outline



Koch Snowflakes and Geometrical Programming





- Assignment I: Koch snowflake (due in 2 weeks)
  - Like the 2D Sierpinski gasket, start with an equilateral triangle
  - but use different generation rules
- General steps for geometric computing:
  - Understand geometric concepts and operations
  - Formalize them into algebraic computation
  - Implement them in code

Next

- Develop a more sophisticated three-dimensional example
  - –Sierpinski gasket: a fractal
- Introduce hidden-surface removal

Moving to 3D

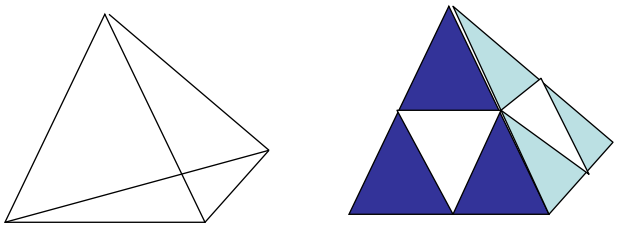
- We can easily make the program three-dimensional by using

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

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

3D Gasket

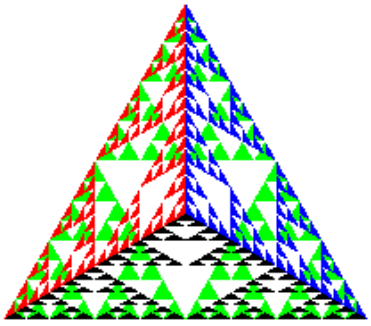
- We can subdivide each of the four faces



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

Example

after 5 interations



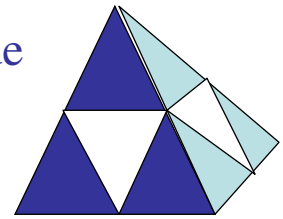
triangle code

```
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));
}
```

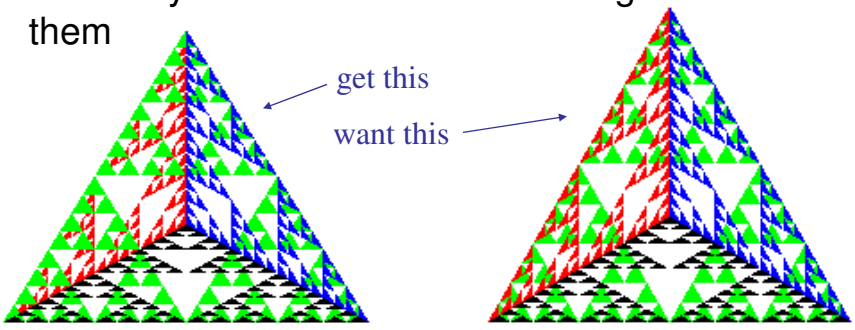
tetrahedron code



```
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);
}
```

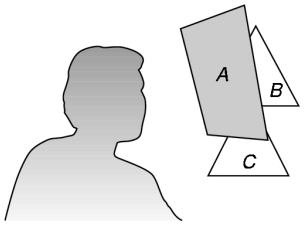
Almost Correct

- 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

- 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

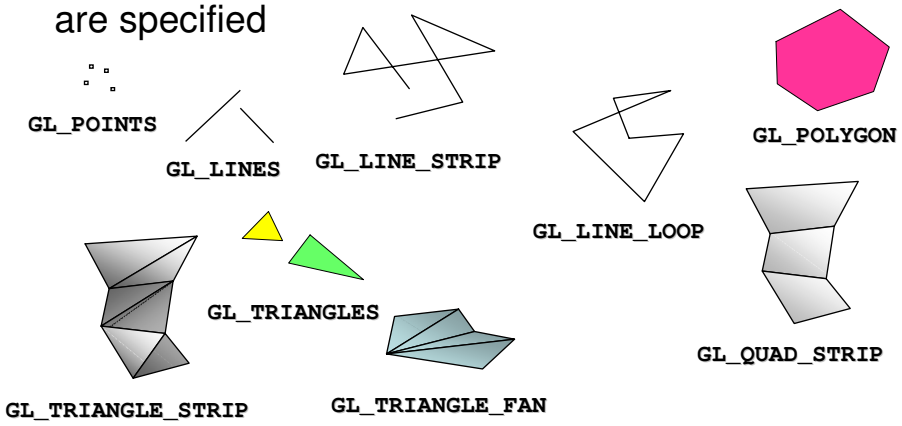


Using the *z*-buffer algorithm

- 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)`

OpenGL Primitives

- pay attention to the order in which the vertices are specified



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

nonsimple polygon

nonconvex polygon

Attributes

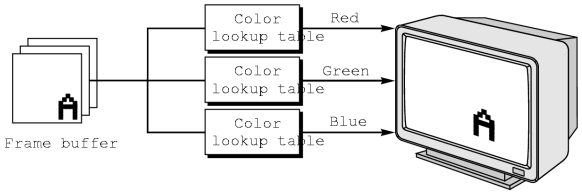
- Attributes are part of the OpenGL and determine the appearance of objects
  - –Color (points, lines, polygons)
  - –Size and width (points, lines)
  - –Stipple pattern (lines, polygons)
  - –Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges

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`**