Introduction to Computer Graphics

Graphics Programming

葉奕成 I-Cheng (Garrett) Yeh

Objectives

- In this lecture, we introduce how to write program and control graphics hardware.
  - A tutorial of OpenGL and glut
- We start with a historical introduction, too.

OpenGL

- A platform-independent API
  - Easy to use
  - Close enough to the hardware to get excellent performance
  - Focus on rendering
  - Omitted windowing and input to avoid window system dependencies

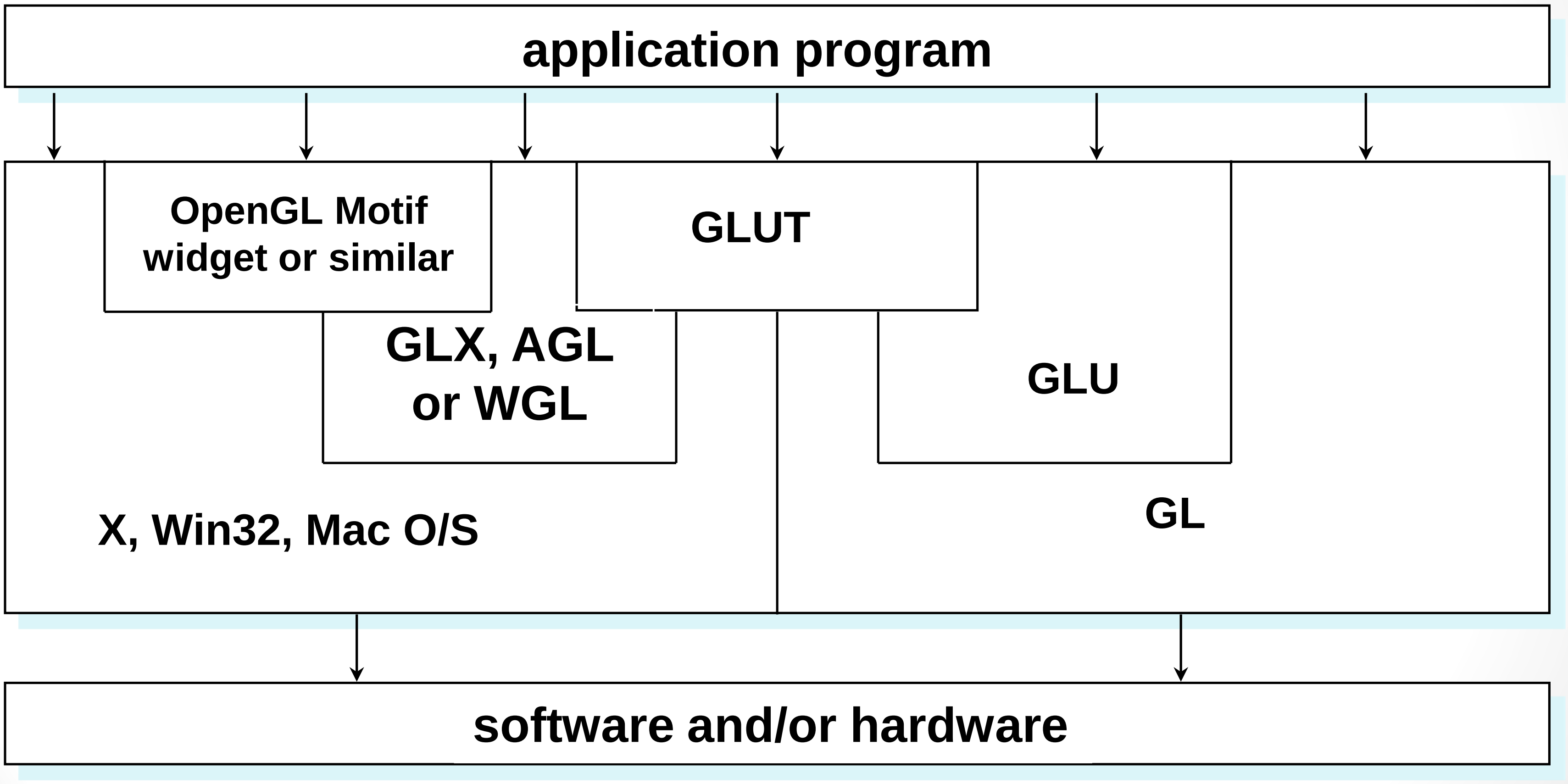
OpenGL Libraries

- 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

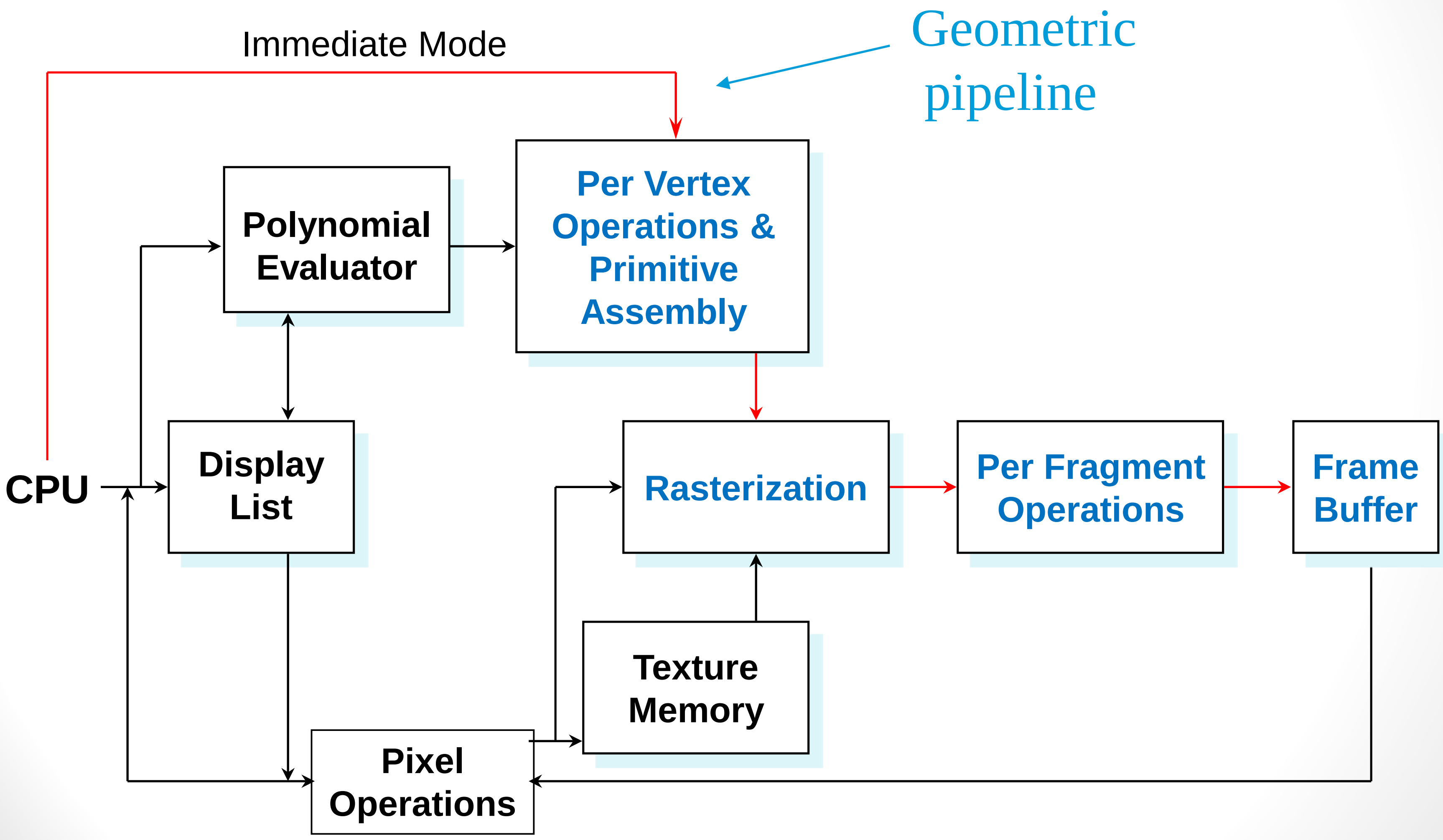
GLUT

- 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

Software Organization



OpenGL Architecture



OpenGL Functions

- Draw Primitives
  - Points
  - Line Segments
  - Polygons
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control
- Input (GLUT)

OpenGL State

- OpenGL is a state machine
- There are **two** types of OpenGL functions
  - Primitive generating
    - Can cause output if primitive is visible
    - How vertices are processes and appearance of primitive are controlled by the state
  - State changing
    - Transformation functions
    - Attribute functions

Lack of Object Orientation

- OpenGL is not object oriented so that there are multiple functions for a given logical function, e.g. glVertex3f, glVertex2i, glVertex3dv,.....
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

OpenGL Function Format

function name

`glVertex3f(x,y,z)`

belongs to GL library

`x,y,z` are `floats`

`glVertex3fv(p)`

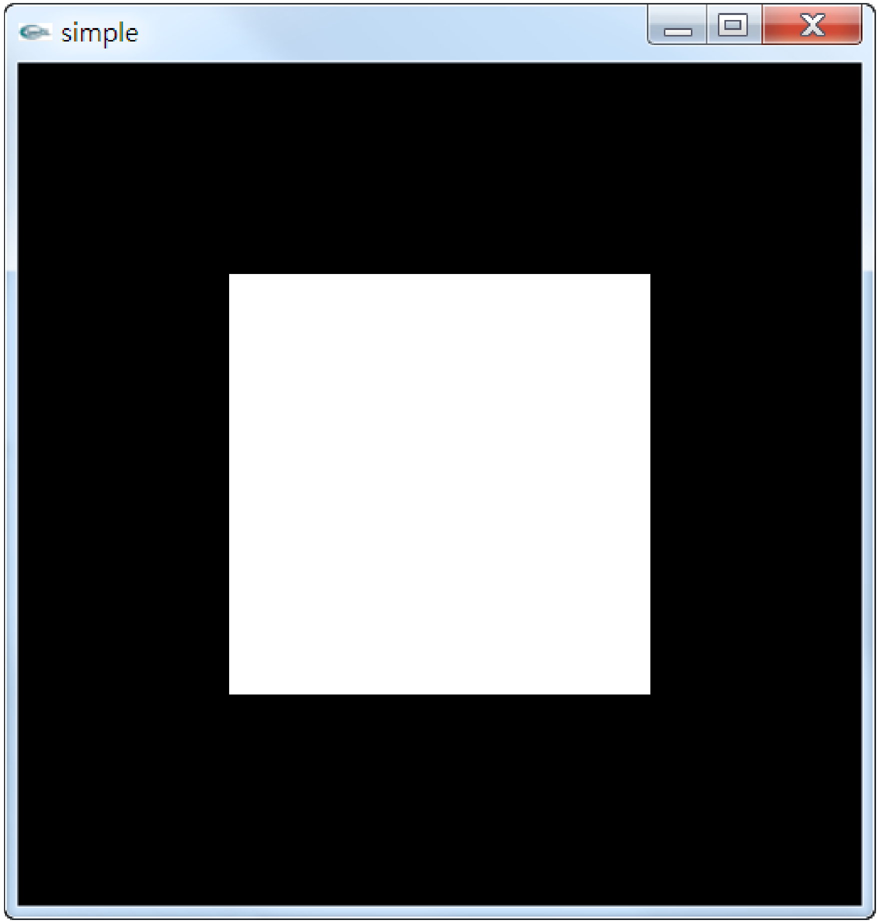
`p` is a pointer to an array

OpenGL #defines

- Most constants are defined in the include files gl.h, glu.h and glut.h
  - Note #include <glut.h> should automatically include the others
  - Examples
    - glBegin(GL_PLOYGON)
    - glClear(GL_COLOR_BUFFER_BIT)
- include files also define OpenGL data types: Glfloat, Gldouble,….

Basic Program

- Generate a square on a solid background



Color_square_basic.cpp

Basic Program

```
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
```

```
void mydisplay(){
    glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POLYGON);
            glVertex2f(-0.5, -0.5);
            glVertex2f(-0.5, 0.5);
            glVertex2f(0.5, 0.5);
            glVertex2f(0.5, -0.5);
        glEnd();
        glFlush();
}

int main(int argc, char** argv){
        glutCreateWindow("simple");
        glutDisplayFunc(mydisplay);
        glutMainLoop();
}
```

Event Loop

- Note that the program defines a display callback function named mydisplay
  - Every glut program must have a display callback
  - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened
  - The main function ends with the program entering an event loop

Defaults

- Basic program is too simple
- Makes heavy use of state variable default values for
  - Viewing
  - Colors
  - Window parameters
- Next version will make the defaults more explicit

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 (called by main func.)
    - viewing
    - Attributes
  - callbacks
    - Display function
    - Input and window functions

Basic Program Revisited

- In this version, we will see the same output but have defined all the relevant state values through function calls with the default values
- In particular, we set
  - Window properties
  - Colors (background and painter)
  - Viewing conditions
  - …

Main Function

```
#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, …)

GLUT functions

- glutWindowPosition from top-left corner of display
- glutCreateWindow create window with title “simple”
- glutDisplayFunc display callback
- glutMainLoop enter infinite event loop

Initialization

```
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

Paint primitive with white color

viewing volume

Coordinate Systems

- The units of in glVertex are determined by the application and are called world coordinates
- The viewing specifications are also in world coordinates and it is the size of the viewing volume that determines what will appear in the image
- Internally, OpenGL will convert to camera coordinates and later to screen coordinates

OpenGL Camera

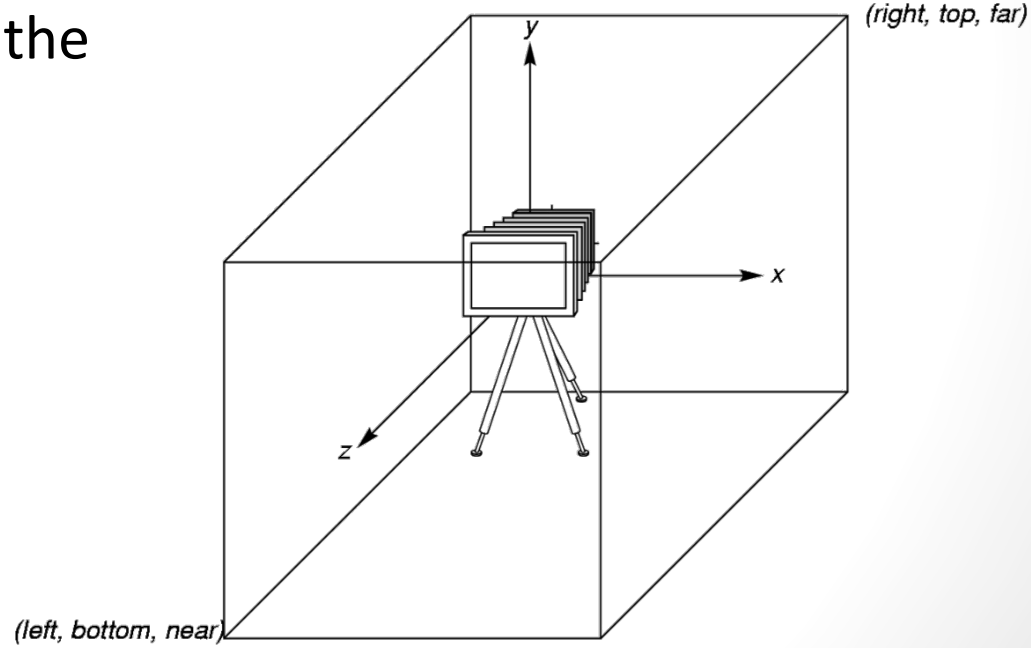
- OpenGL places a camera at the origin pointing in the negative z direction
- The default viewing volume

  is a box centered at the

  origin with a side of

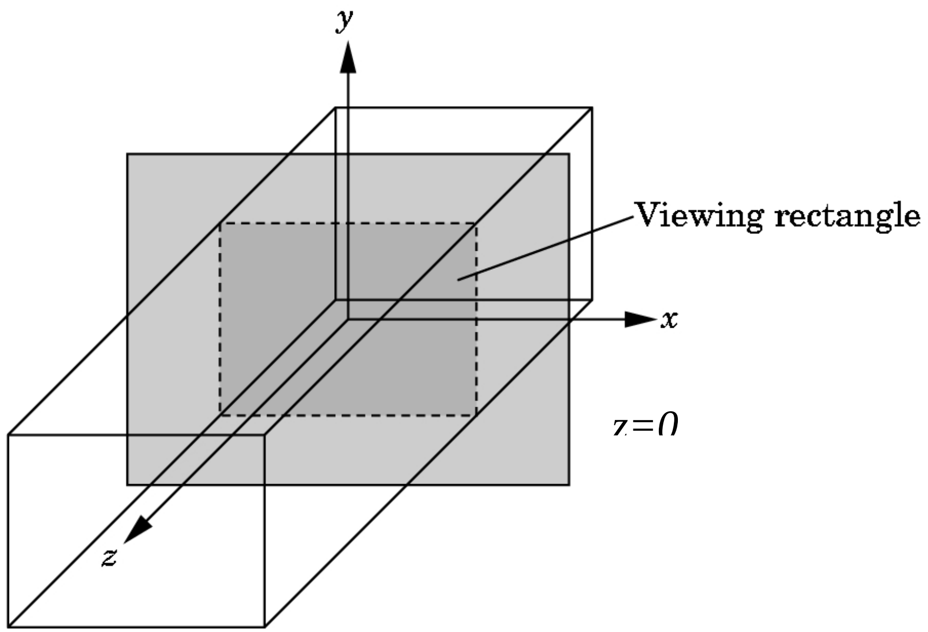
  length 2

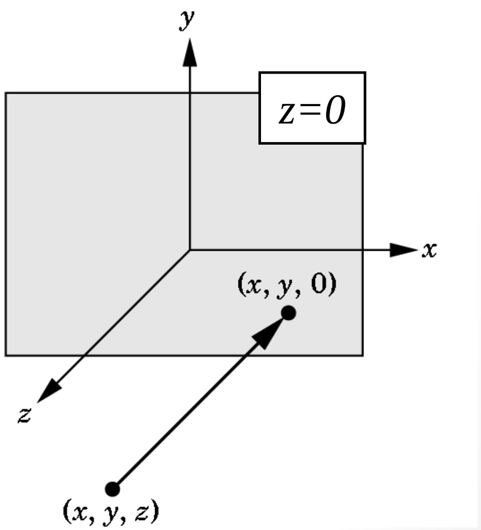
  glOrtho(-1.0, 1.0,
  -1.0, 1.0, -1.0, 1.0);



Orthographic Viewing

In the default orthographic view, points are projected forward along the z axis onto the plane $z=0$





Transformations and Viewing

- In OpenGL, the projection is carried out by a projection matrix (transformation)
- There is only one set of transformation functions so we must set the matrix mode first
  - glMatrixMode (GL_PROJECTION)
- Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume
  - glLoadIdentity ();
  - glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

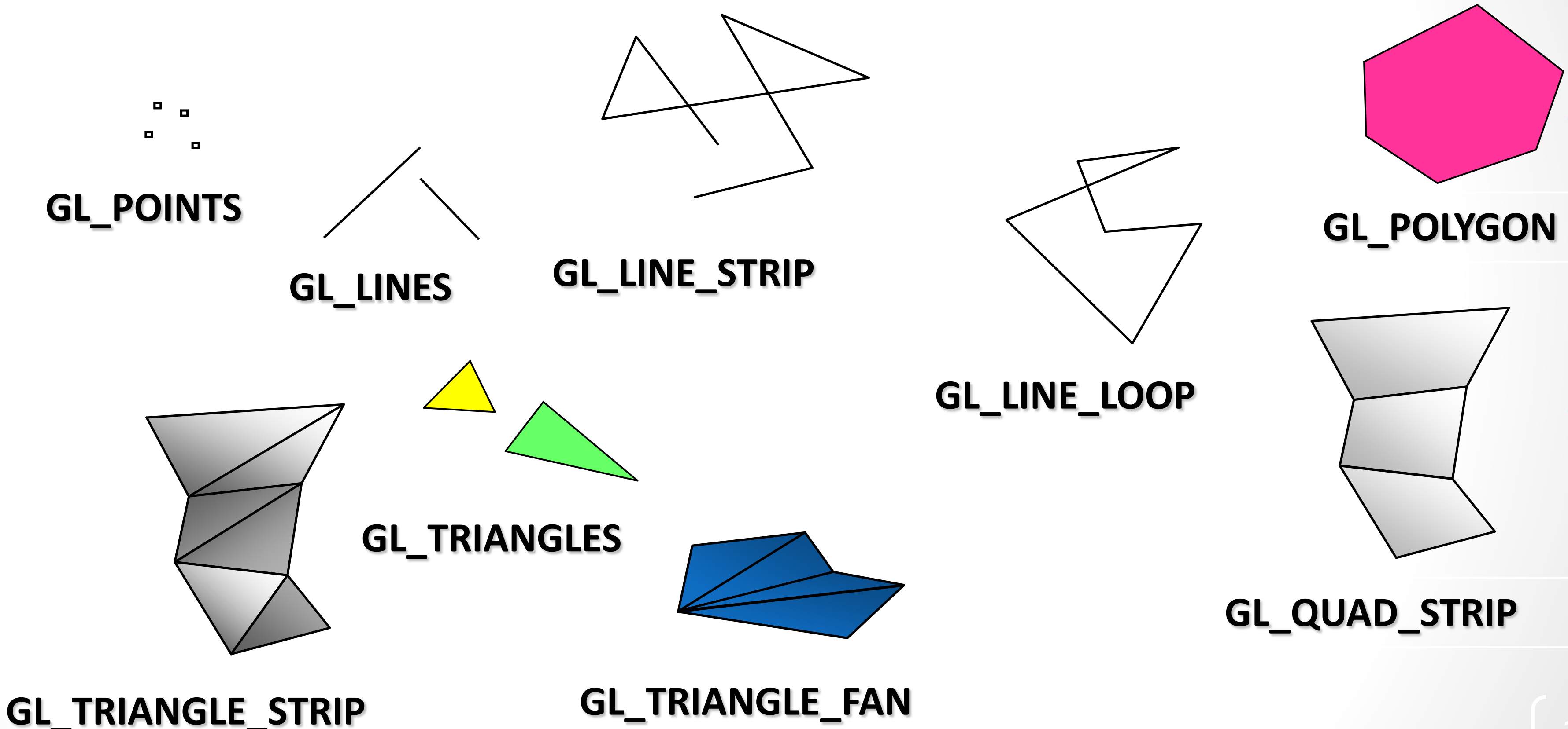
Two- and three-dimensional viewing

- In glOrtho(left, right, bottom, top, near, far) the near and far distances are measured from the camera
  - Two-dimensional vertex commands place all vertices in the plane z=0
- If the application is in two dimensions, we can use the function
  - gluOrtho2D(left, right,bottom,top)
- In two dimensions, the view or clipping volume becomes a clipping window

Display Function

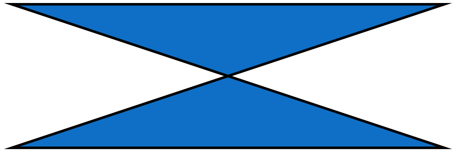
```
void mydisplay()
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POLYGON);
                glVertex2f(-0.5, -0.5);
                glVertex2f(-0.5, 0.5);
                glVertex2f(0.5, 0.5);
                glVertex2f(0.5, -0.5);
        glEnd();
        glFlush();
}
```

OpenGL Primitives



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

Restrictions on Using glBegin() and glEnd()

Valid Commands between glBegin() and glEnd()

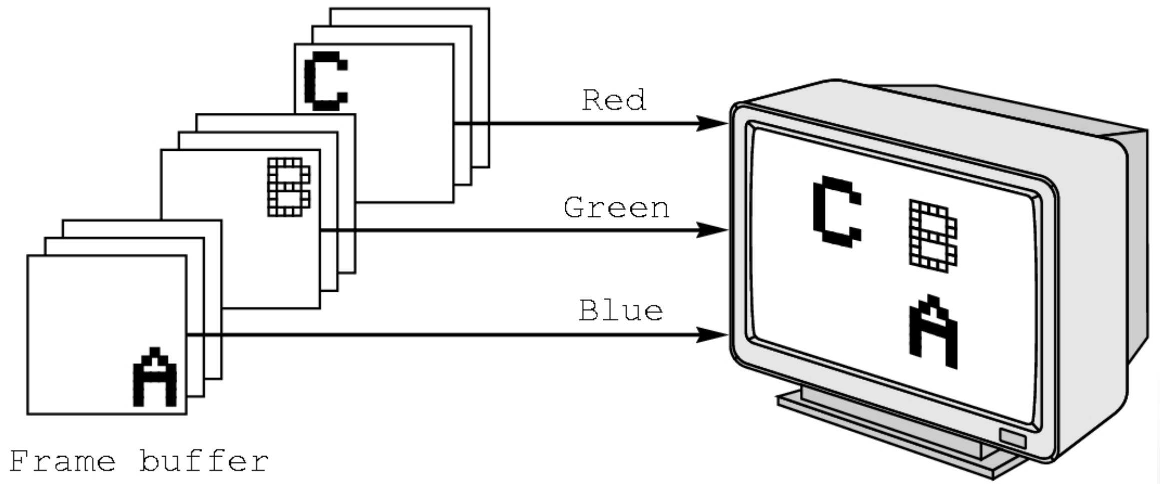
glColor*()	set current color
glIndex*()	set current color index
glNormal*()	set normal vector coordinates
glEvalCoord*()	generate coordinates
glCallList(), glCallLists()	execute display list(s)
glTexCoord*()	set texture coordinates
glEdgeFlag*()	control drawing of edges
glMaterial*()	set material properties

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

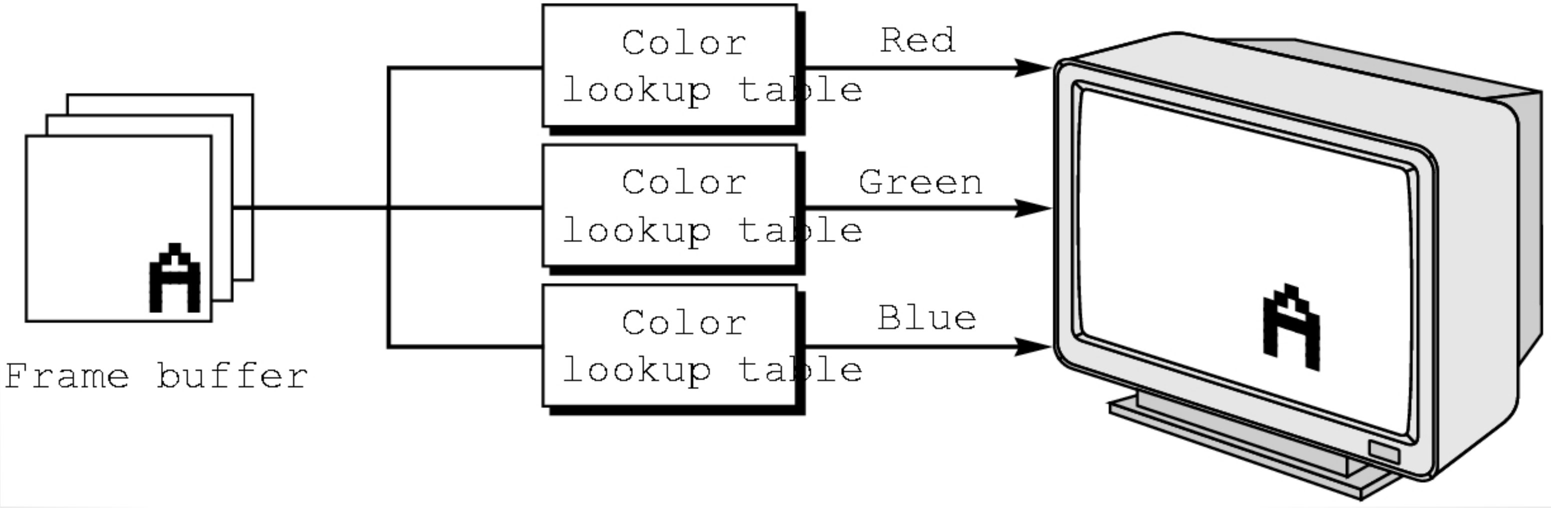
RGB color

- 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

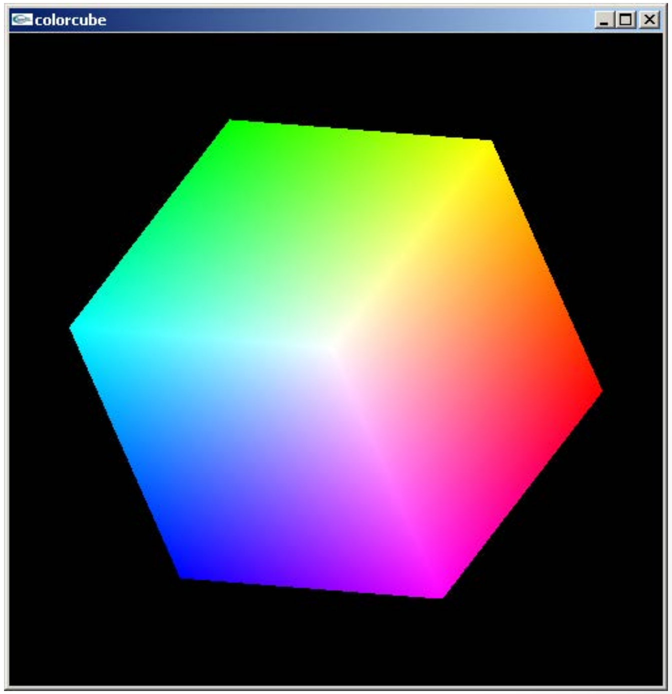


Color and State

- 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

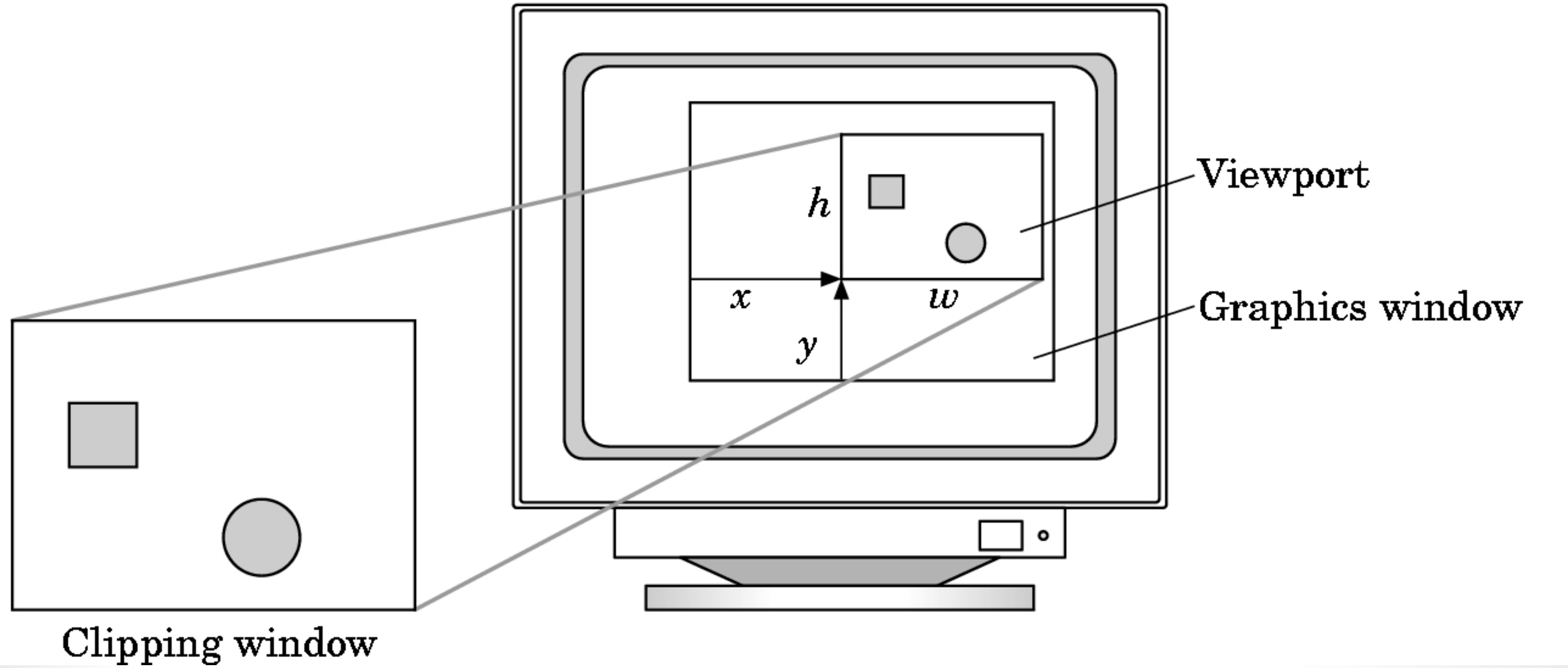
Smooth Color

- 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

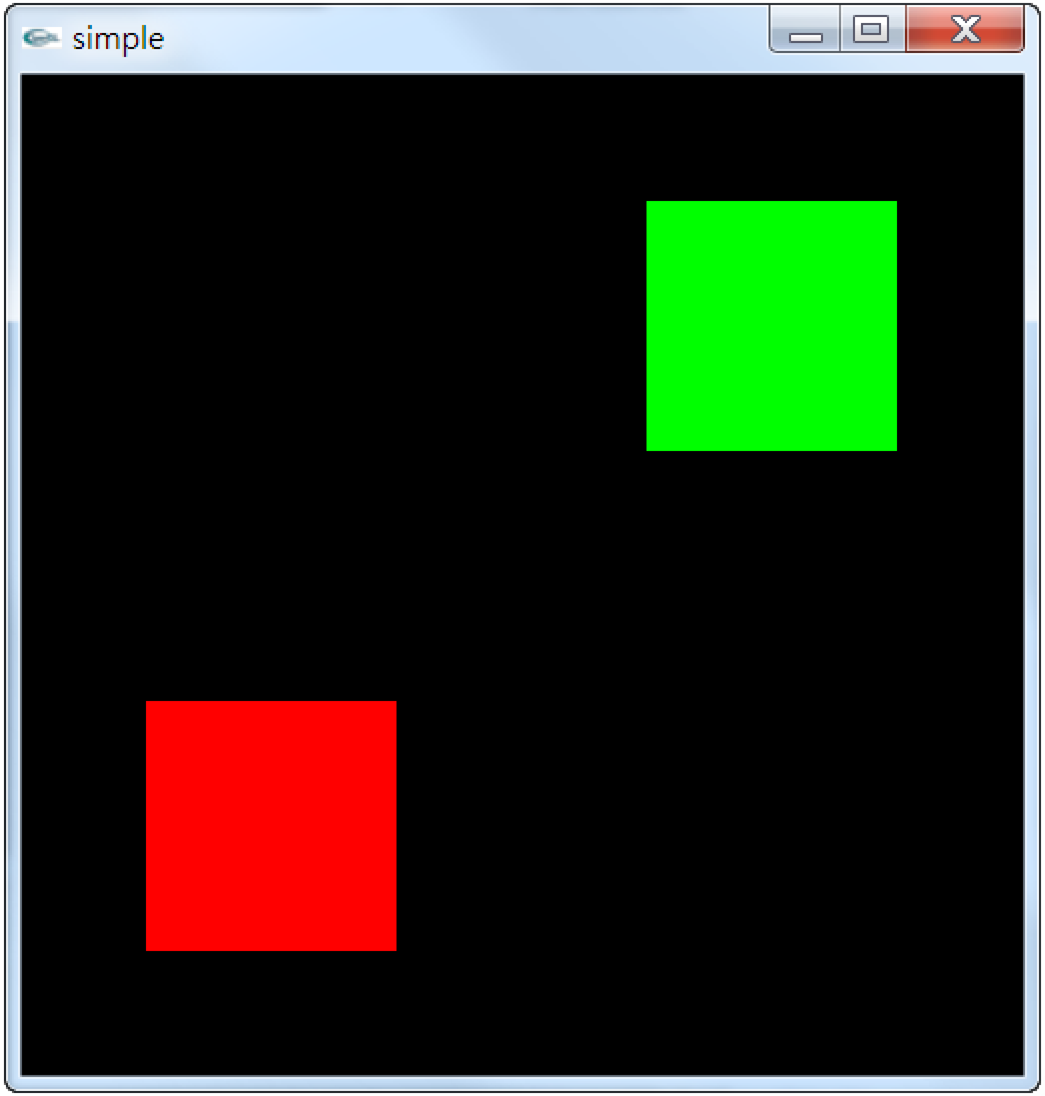


Viewports

- Do not have use the entire window for the image: glViewport(x,y,w,h)
- Values in pixels (screen coordinates)



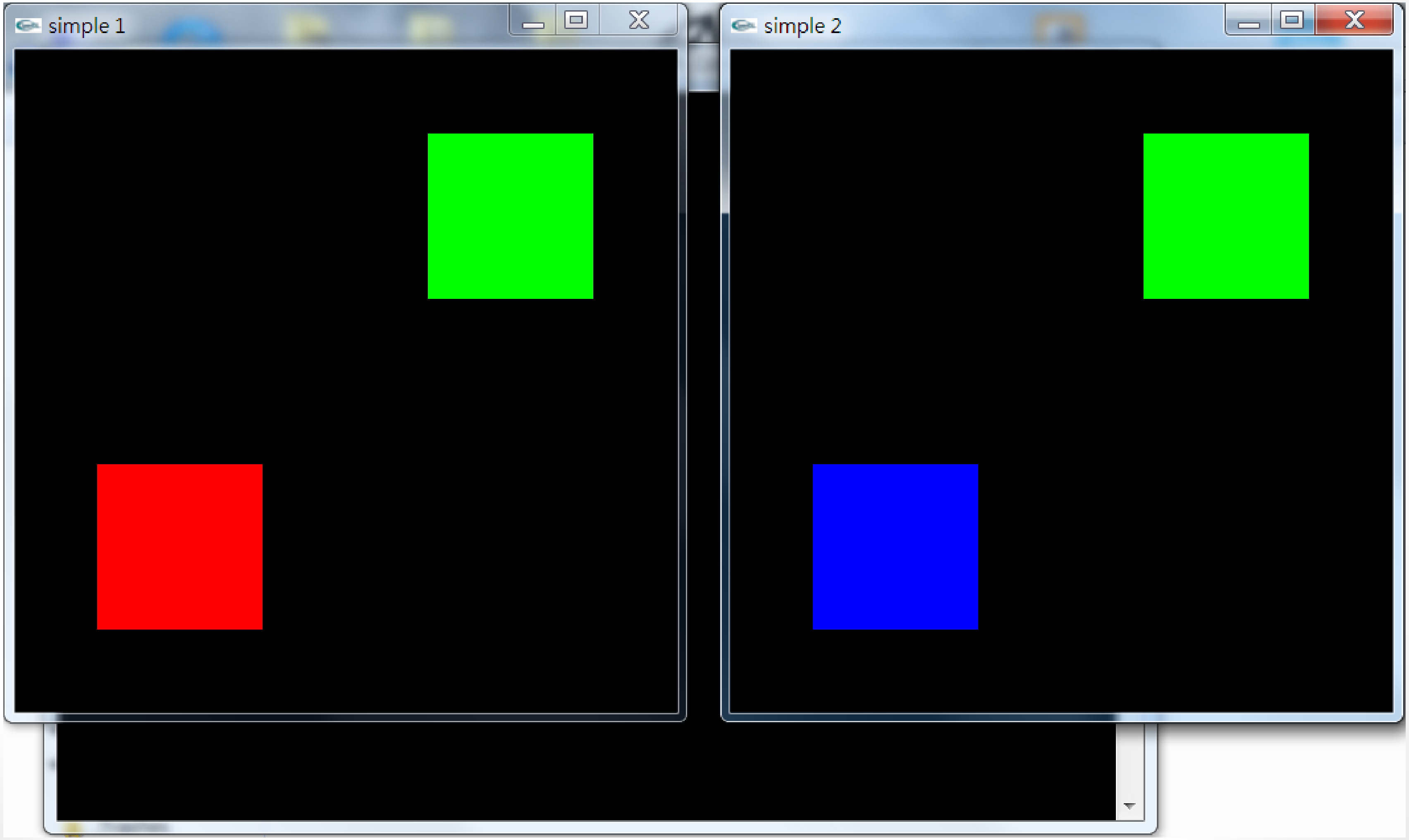
Multiple viewports? HOW?



2 Viewports

```
int width = 0, height = 0;
void myDisplay()
{
        std::cout << "display!";
        glClear(GL_COLOR_BUFFER_BIT);
        glViewport(0, 0, 0.5*width, 0.5*height);
        glColor3f(1.0, 0.0, 0.0);
        glBegin(GL_POLYGON);
                glVertex2f(-0.5, -0.5);
                glVertex2f(-0.5, 0.5);
                glVertex2f(0.5, 0.5);
                glVertex2f(0.5, -0.5);
        glEnd();
        glViewport(0.5*width, 0.5*height, 0.5*width, 0.5*height);
        glColor3f(0.0, 1.0, 0.0);
        …
        glFlush();
}
```

Multiple windows? HOW?



2 Windows

```
int WinID[] = {0,0};
int main(int argc, char** argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        WinID[0] = glutCreateWindow("simple 1");
        glutDisplayFunc(myDisplay);
        glutReshapeFunc(myReshape);
        glutInitWindowPosition(550,0);
        WinID[1] = glutCreateWindow("simple 2");
        glutDisplayFunc(myDisplay2);
        glutReshapeFunc(myReshape);
        init();
        glutMainLoop();
}
```

Take a rest

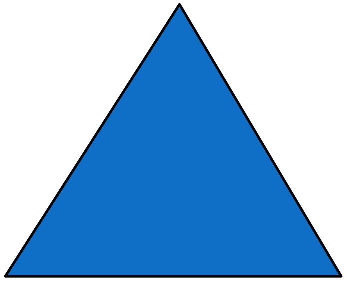
Try everything by yourself

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

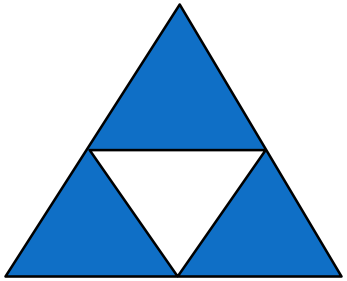
Sierpinski Gasket (2D)

- Start with a triangle



N = 0

- Connect bisectors of sides and remove central triangle

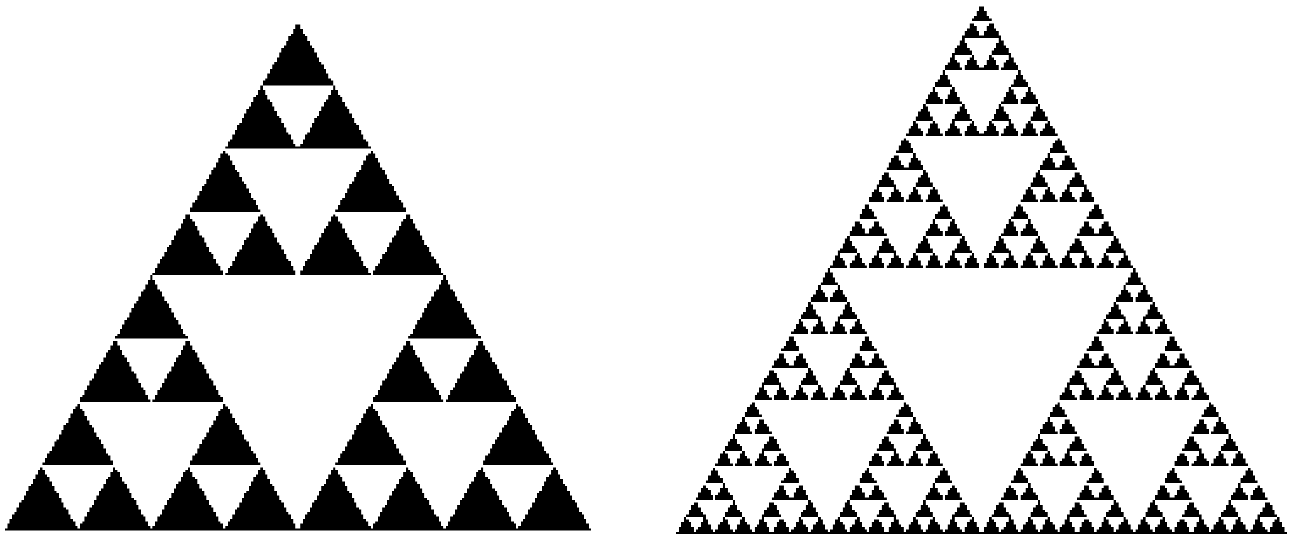


N = 1

- Repeat

Example

- N = 3 & 5



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 = 5; /* 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();
}
```

Gasket Program

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

/* initial triangle */
GLfloat v[3][2]={{-1.0, -0.58}, {1.0, -0.58}, {0.0, 1.15}};

int n = 3; /* number of recursive steps */
```

Draw a Triangle

```
void triangle( GLfloat *a, GLfloat *b, GLfloat *c)
{
/* draw one triangle */
    glVertex2fv(a);
    glVertex2fv(b);
    glVertex2fv(c);
}
```

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 Initialization 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

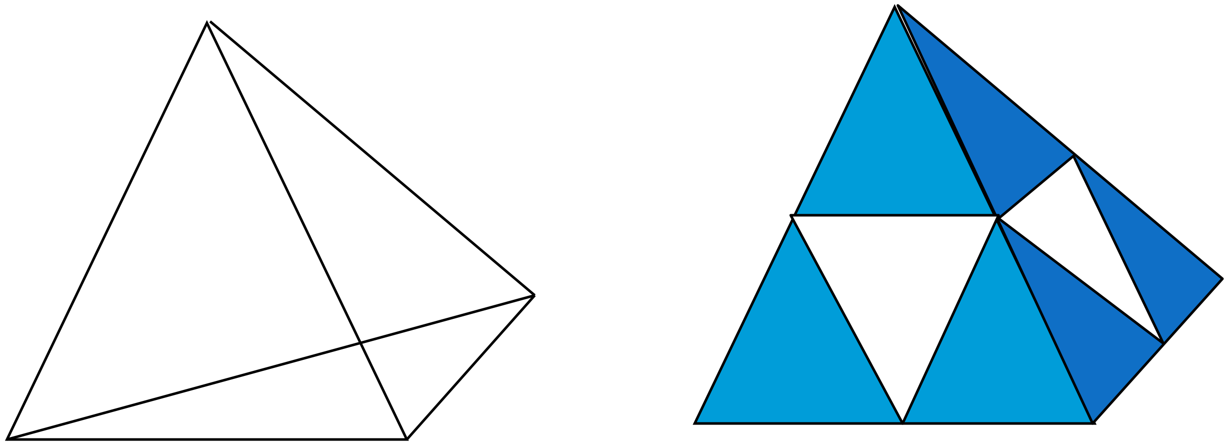
```
int main(int argc, char **argv)
{
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutCreateWindow("2D Sierpinski Gasket");
        glutDisplayFunc(myDisplay);
        myInit();
        glutMainLoop();
        return 0;
}
```

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

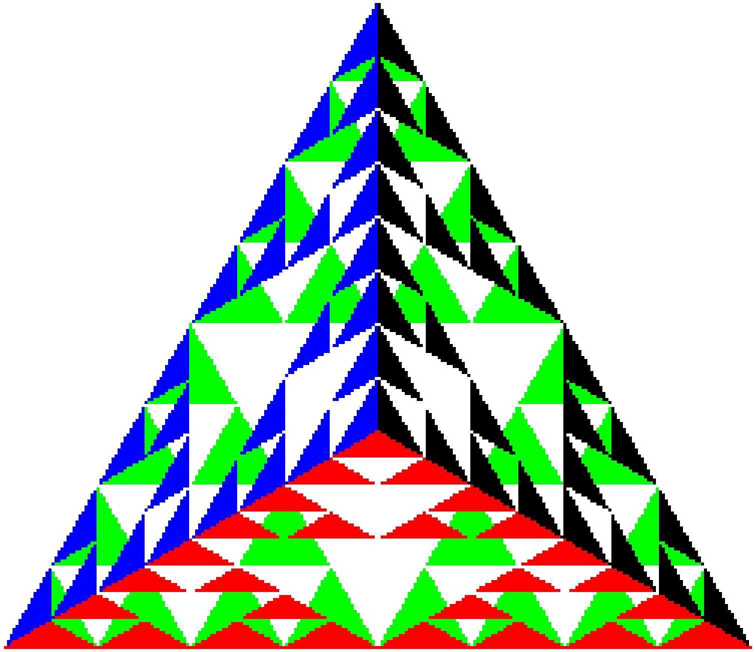


- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedtra

Example

- after 3 iterations

```
typedef GLfloat point3[3];
point3 v[]={
{0.0, 0.0, 1.15},
{-1.0, -0.58, -0.58},
{1.0, -0.58, -0.58},
{0.0, 1.15, -0.58}};
```

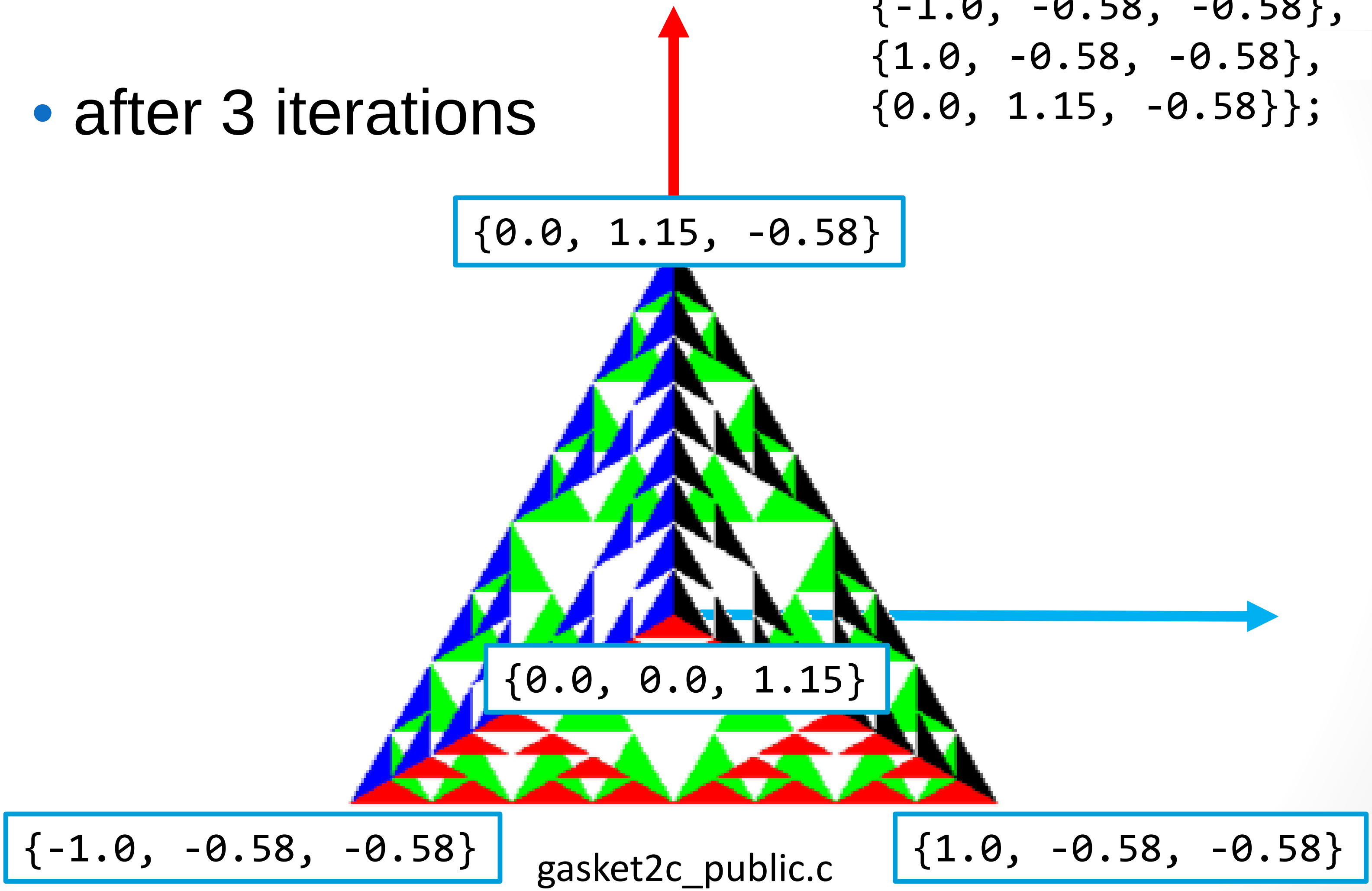


gasket2c_public.c

Example

- after 3 iterations

```
typedef GLfloat point3[3];
point3 v[]={
{0.0, 0.0, 1.15},
{-1.0, -0.58, -0.58},
{1.0, -0.58, -0.58},
{0.0, 1.15, -0.58}};
```



gasket2c_public.c

triangle code

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

subdivision code

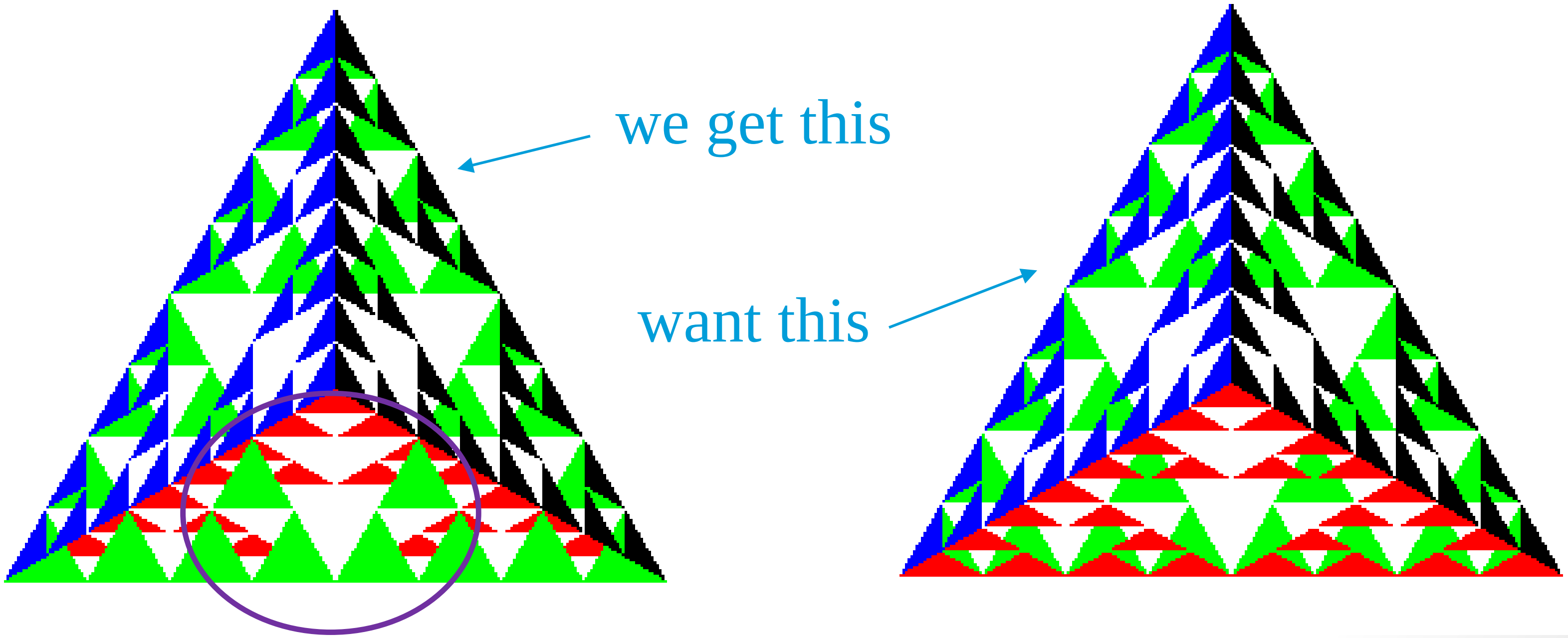
```
void divide_triangle(point3 a, point3 b, point3 c, int m) {
/* triangle subdivision using vertex numbers */
   point3 v0, v1, v2;
   int j;
   if(m>0)
   {
      for(j=0; j<3; j++) v0[j]=(a[j]+b[j])/2;
      for(j=0; j<3; j++) v1[j]=(a[j]+c[j])/2;
      for(j=0; j<3; j++) v2[j]=(b[j]+c[j])/2;
      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 */
}
```

tetrahedron code

```
void tetrahedron( int m) {
/* Apply triangle subdivision to faces of tetrahedron */
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);
}
```

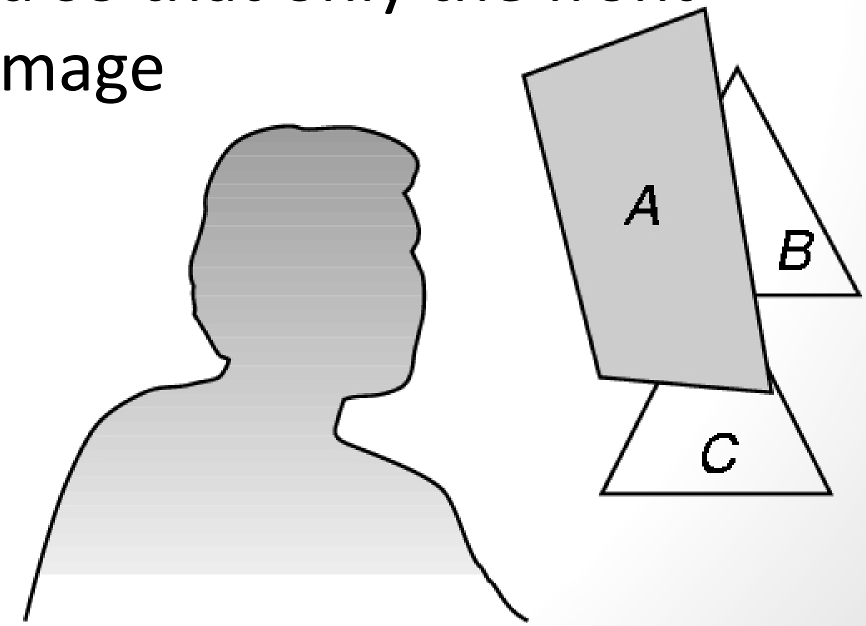
Almost Correct

- 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

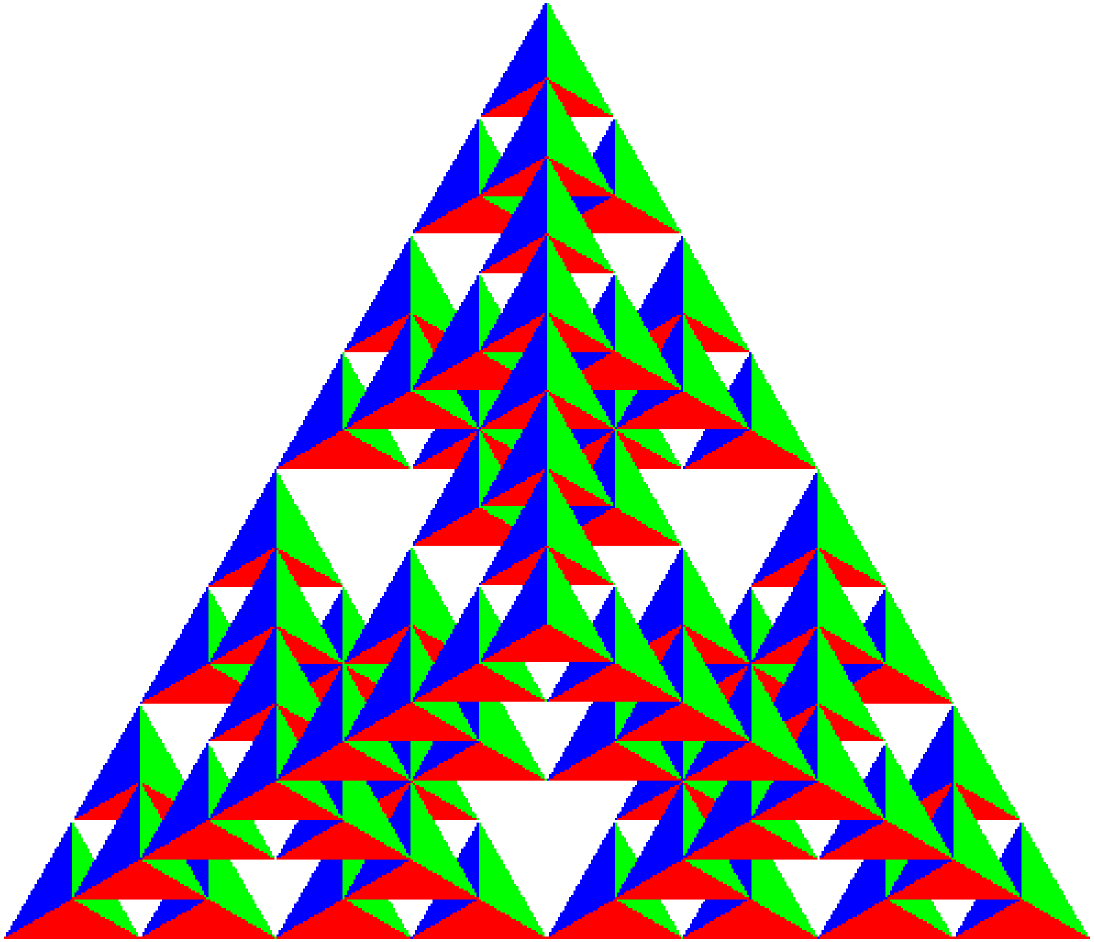
- We want to see only those surfaces in front of other surfaces
- OpenGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



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)

Volume Subdivision?



Surface vs Volume Subdvision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints (six points)
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a volume in the middle

Selected Extensive Readings

- OpenGL:The Industry's Foundation for High Performance Graphics
- GLUT - The OpenGL Utility Toolkit
- The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3
- NeHe Productions
- Direct3D vs. OpenGL: A Comparison

SUGGESTION!

OR

OBJECTION?

Let’s stop here,

TAKE A BREAK