2. Programming with OpenGL

Lecture Overview

- Recap of Lecture I
- Programming with OpenGL
  - Background
  - Simple programs
  - Intro to 3D
- Note: only immediate mode covered today

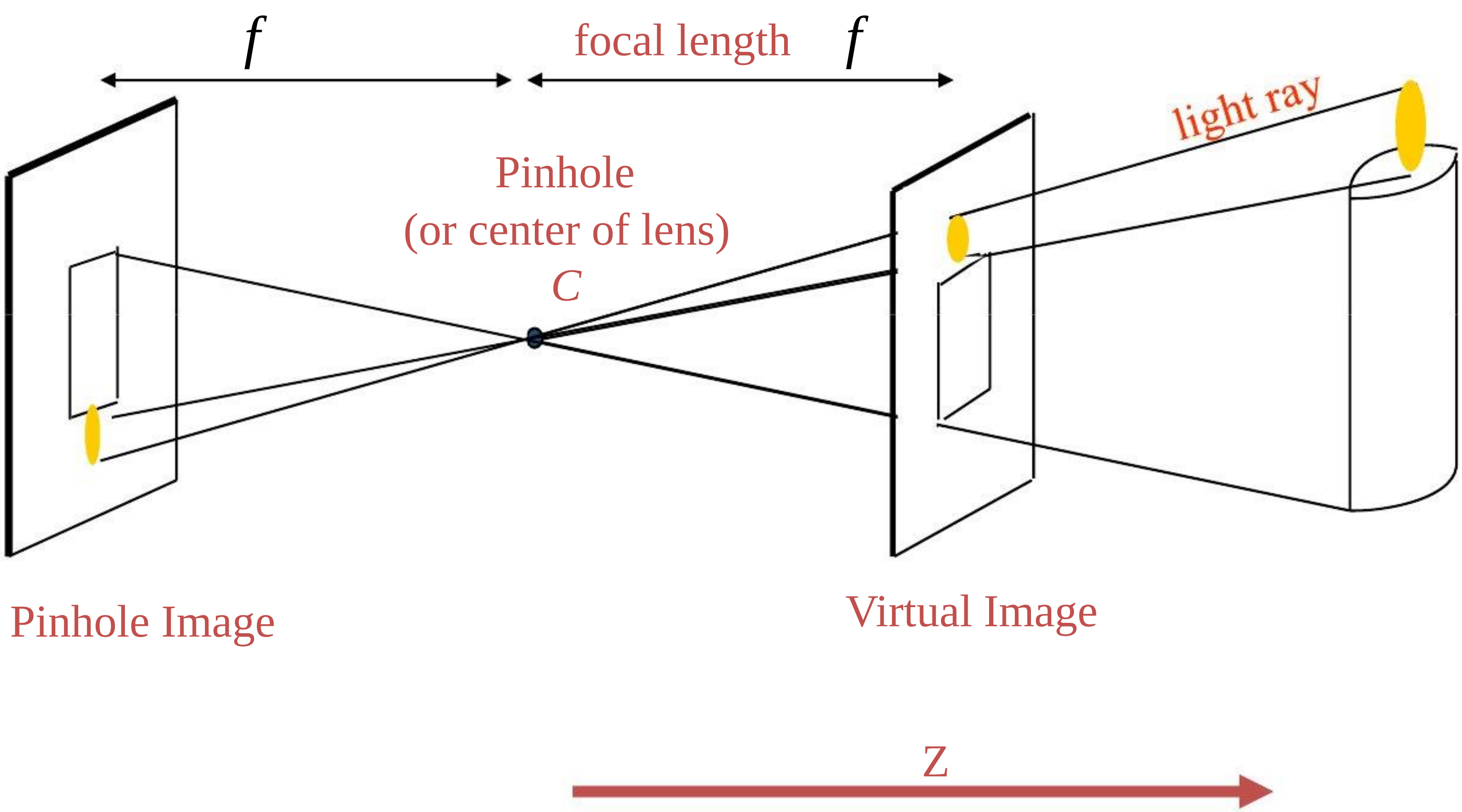
- Reading: ANG Ch. 2, except 2.11

Recap

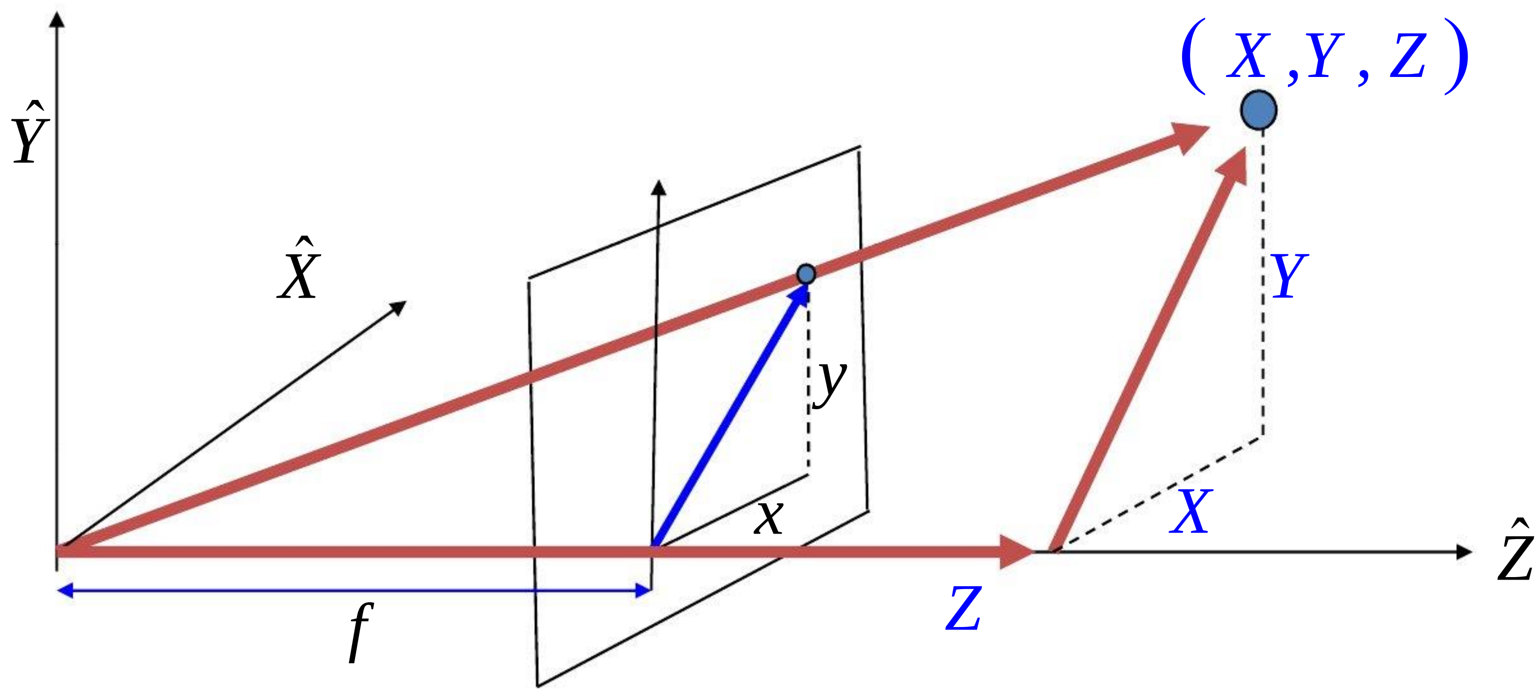
- Image formation
- The rendering pipeline

Perspective Projection

- Pinhole camera ➔ Virtual image



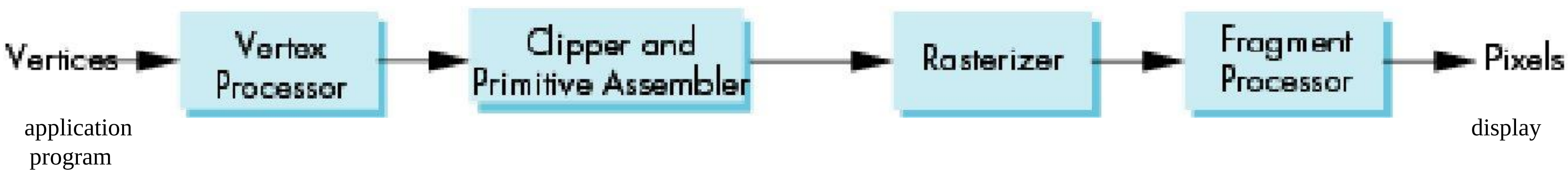
Projection Equation



Similar triangles: $\frac{x}{X} = \frac{y}{Y} = \frac{f}{Z}$ → $\boxed{(x, y) = \frac{f}{Z}(X, Y)}$

Rendering Pipeline

- Process objects one at a time in the order they are generated by the application
  - Can consider only local lighting
- Pipeline architecture



- All steps can be implemented in hardware on the graphics card

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

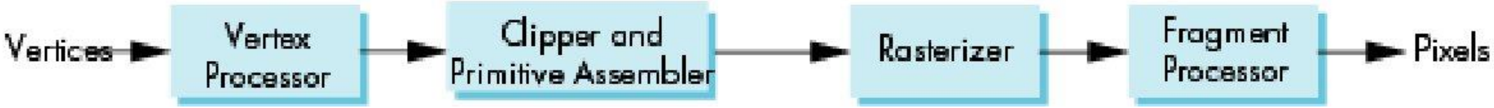
Vertex Processing

- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates
- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Projection

- Projection is the process that combines the 3D viewer with the 3D objects to produce the 2D image
  - Perspective projections: all projectors meet at the center of projection
  - Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection

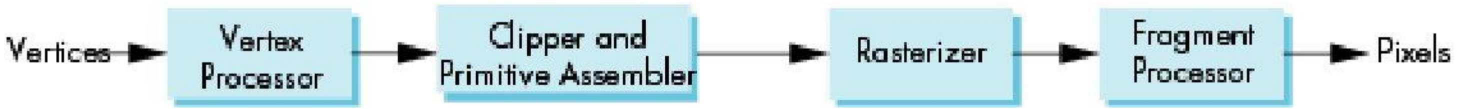


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Primitive Assembly

Vertices must be collected into geometric objects before clipping and rasterization can take place

- Line segments
- Polygons
- Curves and surfaces

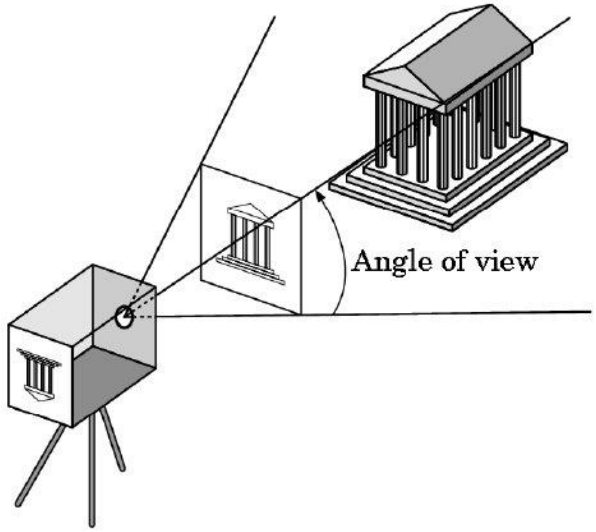


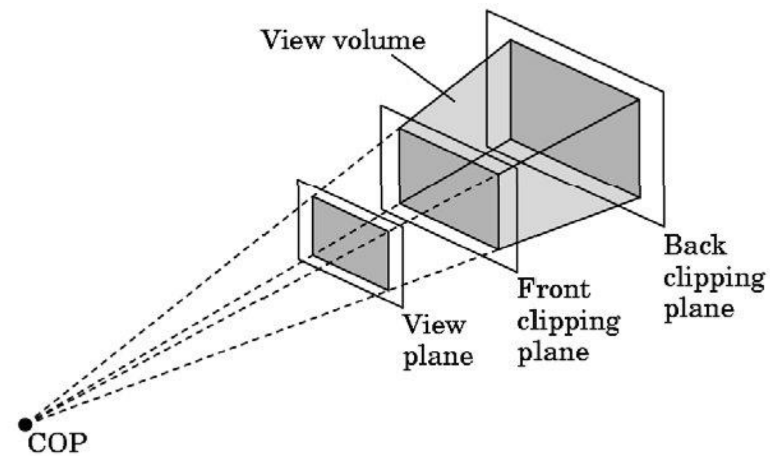
Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Clipping

Just as a real camera cannot "see" the whole world, the virtual camera can only see part of the world or object space

– Objects that are not within this volume are said to be clipped out of the scene

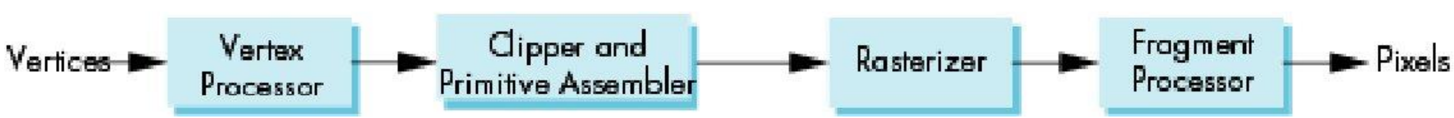




Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Rasterization

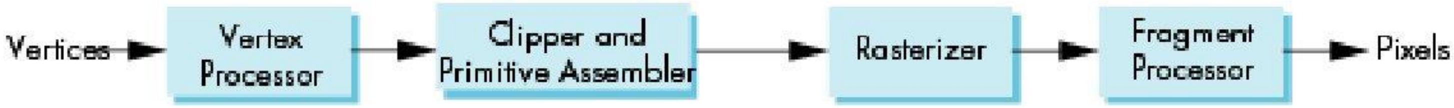
- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- The rasterizer produces a set of fragments for each object
- Fragments are "potential pixels"
  - Have a location in frame bufffer
  - Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Fragment Processing

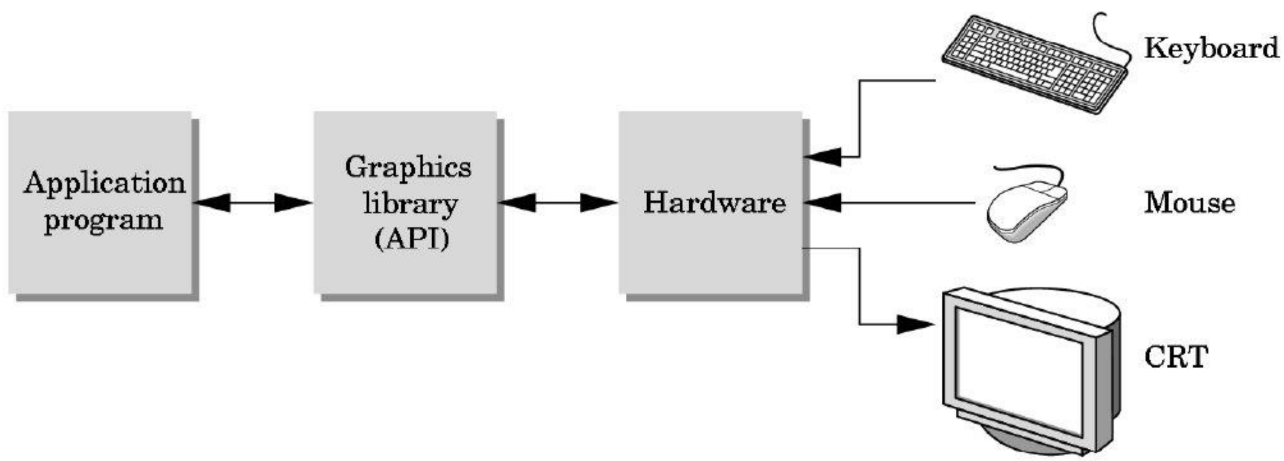
- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

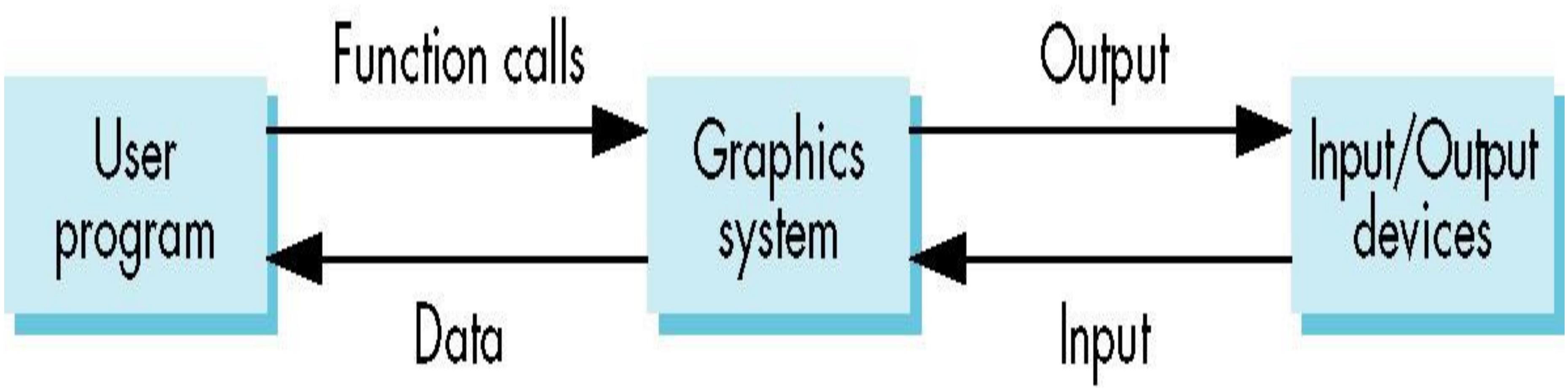
The Programmer's Interface

- Programmers see the graphics system through a software interface: the Application Programmer Interface (API)



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Graphics system as a black box



Programming with OpenGL
Part 1: Background

Objectives

- Development of the OpenGL API
- OpenGL Architecture
  - OpenGL as a state machine
- Functions
  - Types
  - Formats
- Simple program

Early History of APIs

- IFIPS (1973) formed two committees to come up with a standard graphics API
  - Graphical Kernel System (GKS)
    - 2D but contained good workstation model
  - Core
    - Both 2D and 3D
  - GKS adopted as IS0 and later ANSI standard (1980s)
- GKS not easily extended to 3D (GKS-3D)
  - Far behind hardware development

PHIGS and X

- Programmers Hierarchical Graphics System (PHIGS)
  - Arose from CAD community
  - Database model with retained graphics (structures)
- X Window System
  - DEC/MIT effort
  - Client-server architecture with graphics
- PEX combined the two
  - Not easy to use (all the defects of each)

SGI and GL

- Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
- To access 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

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,…….
  - Relatively stable
    - Evolution reflects new hardware capabilities
      - 3D texture mapping and texture objects
      - Vertex programs
  - Allows for platform specific features through extensions
  - ARB replaced by Khronos

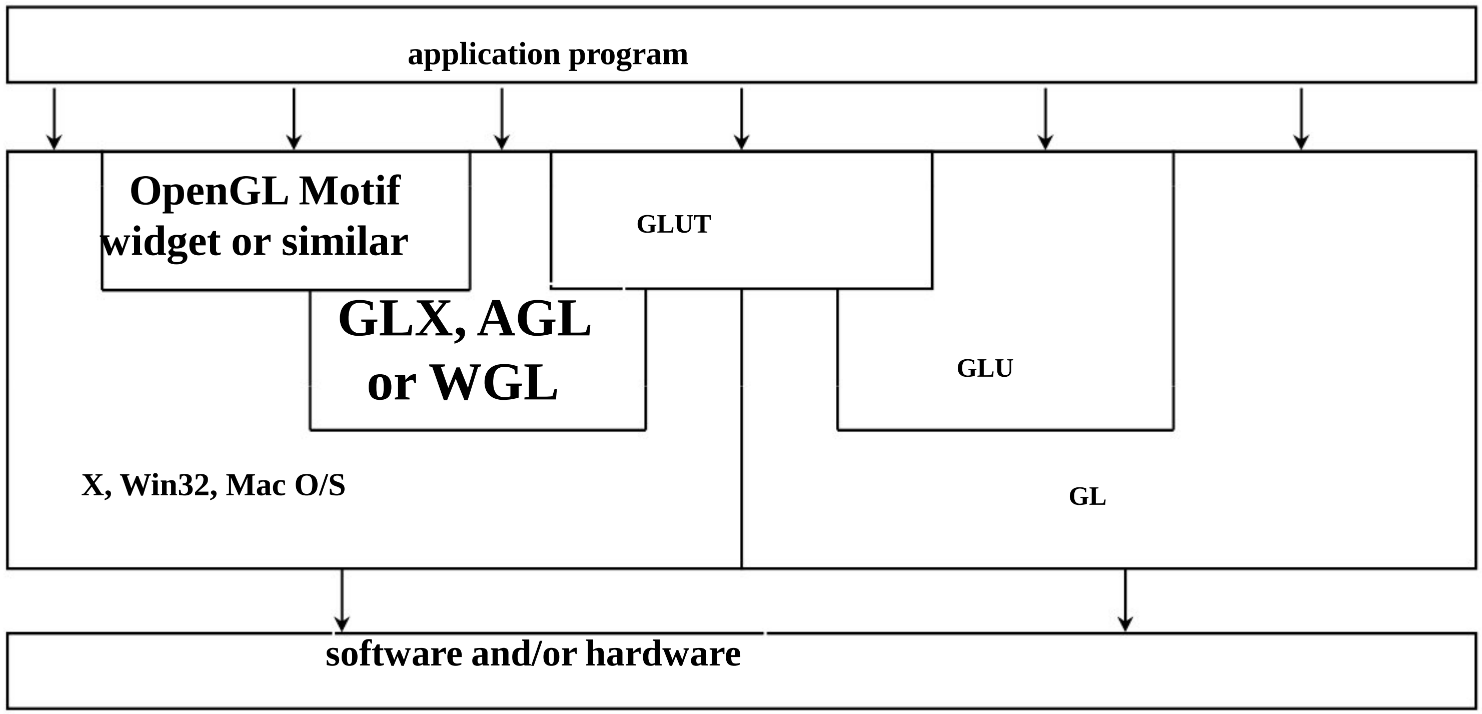
OpenGL Libraries

- OpenGL core library
  – OpenGL32 on Windows
  – GL on most unix/linux systems (libGL.a)
- 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 Windows
  – AGL for Macintosh

GLUT

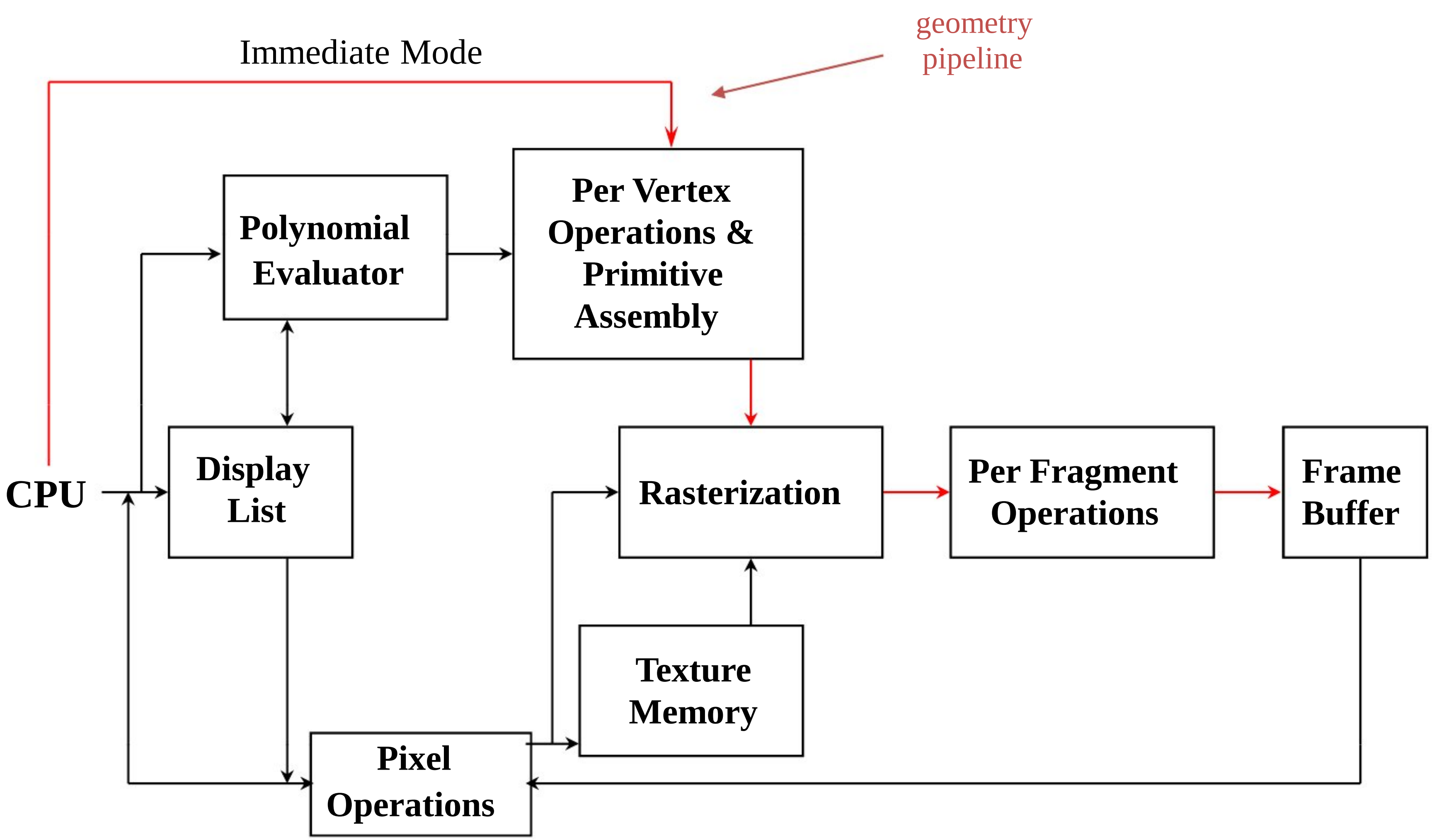
- OpenGL Utility Toolkit (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven
  - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - No slide bars

Software Organization



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

OpenGL Architecture



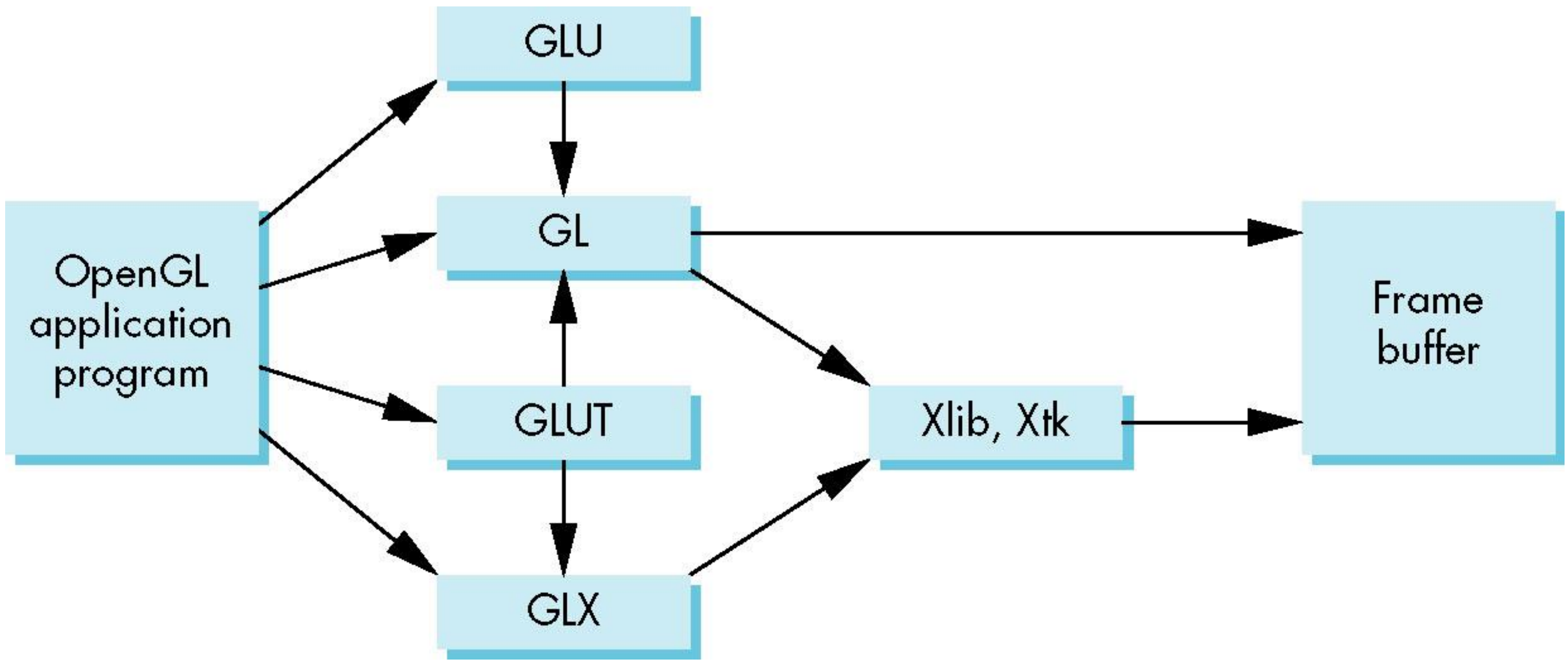
OpenGL Functions

- Primitives
  - Points
  - Line Segments
  - Polygons
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (GLUT)
  - Interaction with window system
- Input (GLUT)
- Query

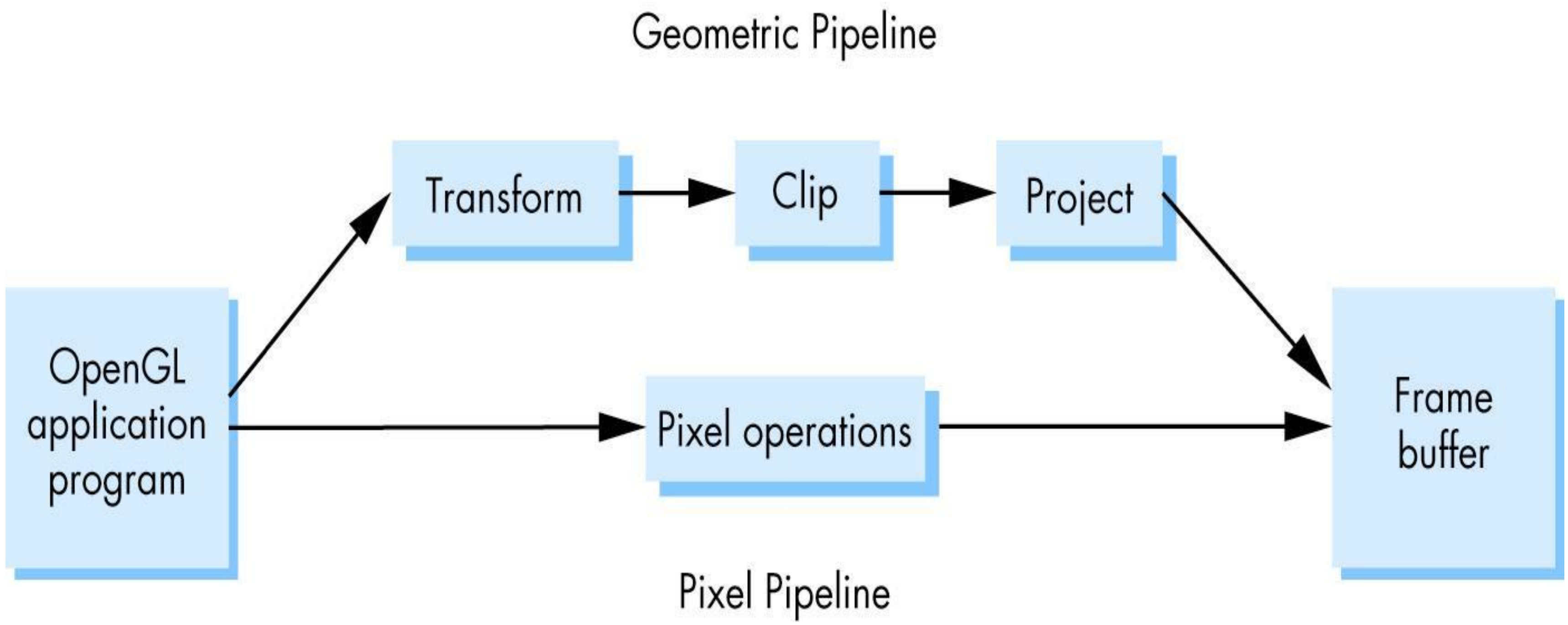
OpenGL State

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

Library Organization



Simplified OpenGL Pipeline



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Not Object Oriented

- OpenGL is not object oriented, so there are multiple functions for a given logical function
  - **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

dimensions

`glVertex3f(x,y,z)`

belongs to GL library

`x,y,z` are `floats`

`glVertex3fv(p)`

`p` is a pointer to an array

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

OpenGL #defines

- Most constants are defined in the include files **gl.h**, **glu.h** and **glut.h**
  - Note **#include <GL/glut.h>** should automatically include the others
  - Examples

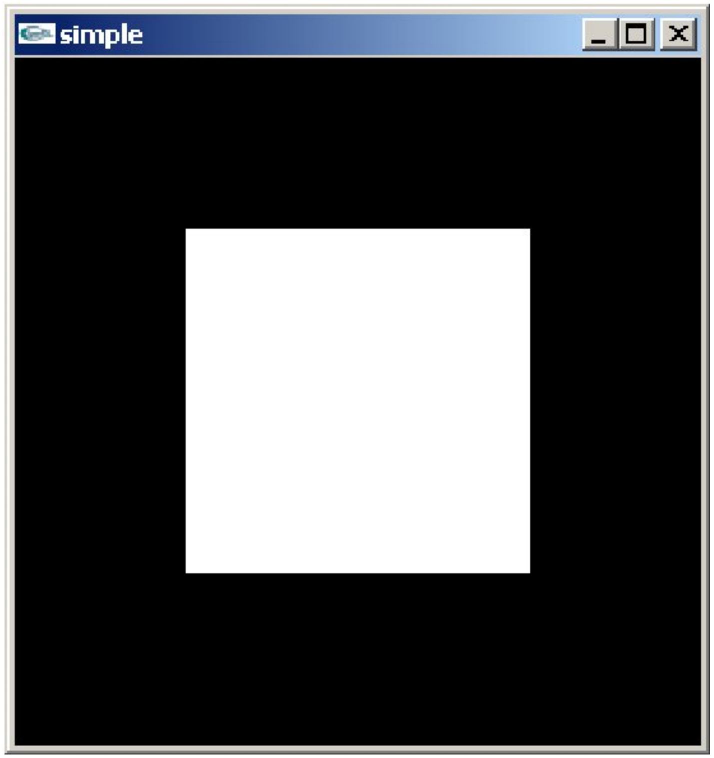
    glBegin(GL_POLYGON)

    glClear(GL_COLOR_BUFFER_BIT)
- include files also define OpenGL data types: **GLfloat, GLdouble,….**

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A Simple Program

Generate a square on a solid background



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

simple.c

```
#include <GL/glut.h>
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();
}
```

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

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

- **simple.c** is too simple
- Relies on state variable default values for
  – Viewing
  – Colors
  – Window parameters
- Next version will make the defaults more explicit

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Notes on compilation

- See website and ftp for examples
- Unix/linux
  - Include files usually in …/include/GL
  - Compile with –lglut –lglu –lgl loader flags
  - May have to add –L flag for X libraries
  - Mesa implementation included with most linux distributions
  - Check web for latest versions of Mesa and glut

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)
- Cygwin
  – Can use gcc and similar makefile to linux
  – Use –lopengl32 –lglu32 –lglut32 flags

於Dev Cpp使用OpenGL之設定

1. 先將glut.dll & glut32.dll丟至windows根目錄 (C:\WINDOWS or C:\WINNT)
2. 將glut.h 丟至<Dev-cpp 安裝路徑>\include\GL\glut.h
3. 再將glut.lib & glut32.lib 丟到
   <Dev-cpp 安裝路徑>\lib\glut.lib
   <Dev-cpp 安裝路徑>\lib\glut32.lib
4. 開啟Dev C++ 選擇 工具->編譯器選項 切換到第一頁編譯器下(預設也是這頁啦)
   然後新增一個編譯器組態
   接著勾選下面的第二個選項"在連結器命令列中加入以下命令“
   最後在下面輸入參數 -lglut32 -lglu32 -lopengl32 -lwinmm -lgdi32

Programming with OpenGL Part 2: Complete Programs

Objectives

- Refine the first program
  - Alter the default values
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Fundamental OpenGL primitives
- Attributes

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

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

simple.c revisited

- In this version, we shall see the same output but we have defined all the relevant state values through function calls using the default values
- In particular, we set
  – Colors
  – Viewing conditions
  – Window properties

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();
}
```

include glut.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 for the window (the rendering context)
  - RGB color
  - Single buffering
  - Properties logically ORed together
- **glutInitWindowSize** in pixels
- **glutInitWindowPosition** from top-left corner of display
- **glutCreateWindow** create window with title "simple"
- **glutDisplayFunc** display callback
- **glutMainLoop** enter infinite event loop

init.c

black clear color

opaque window

```
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);
}
```

fill/draw with white

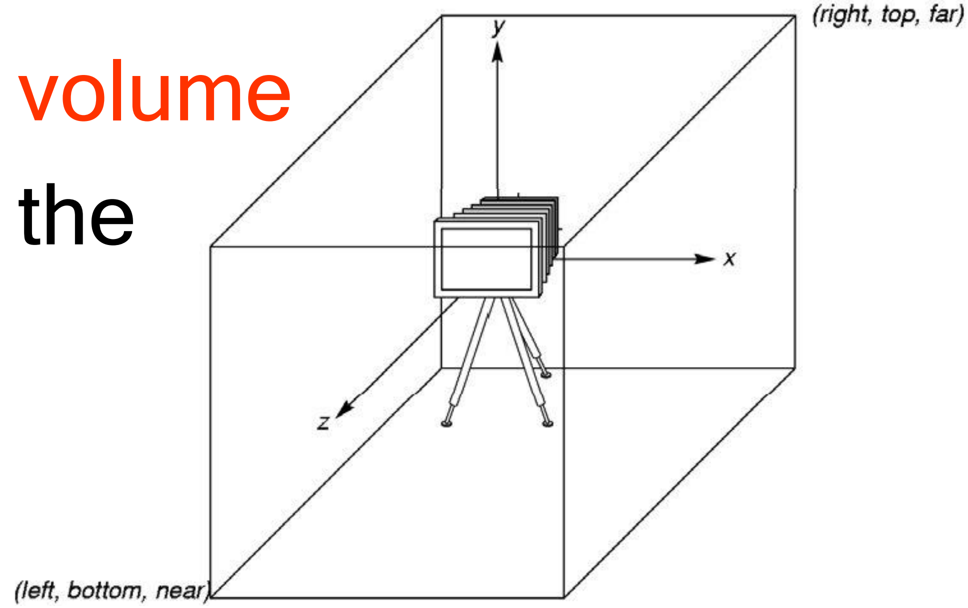
viewing volume

Coordinate Systems

- The units in **glVertex** are determined by the application and are called object or problem coordinates
- The viewing specifications are also in object coordinates and it is the size of the viewing volume that determines what will appear in the image
- Internally, OpenGL will convert to camera (eye) coordinates and later to screen coordinates
- OpenGL also uses some internal representations that usually are not visible to the application

OpenGL Camera

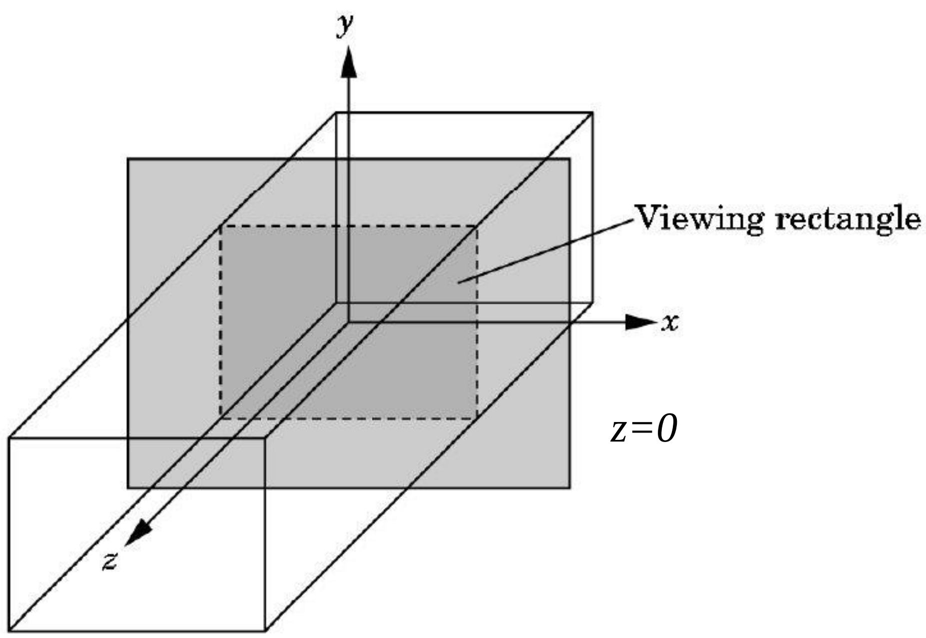
- OpenGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with a side of length 2

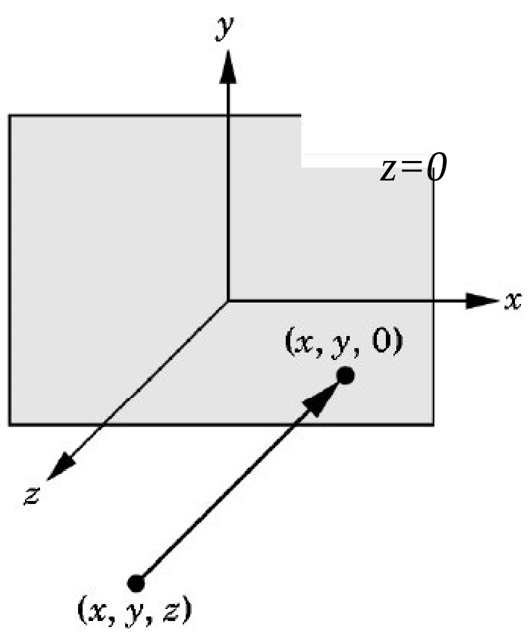


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Orthographic Viewing

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

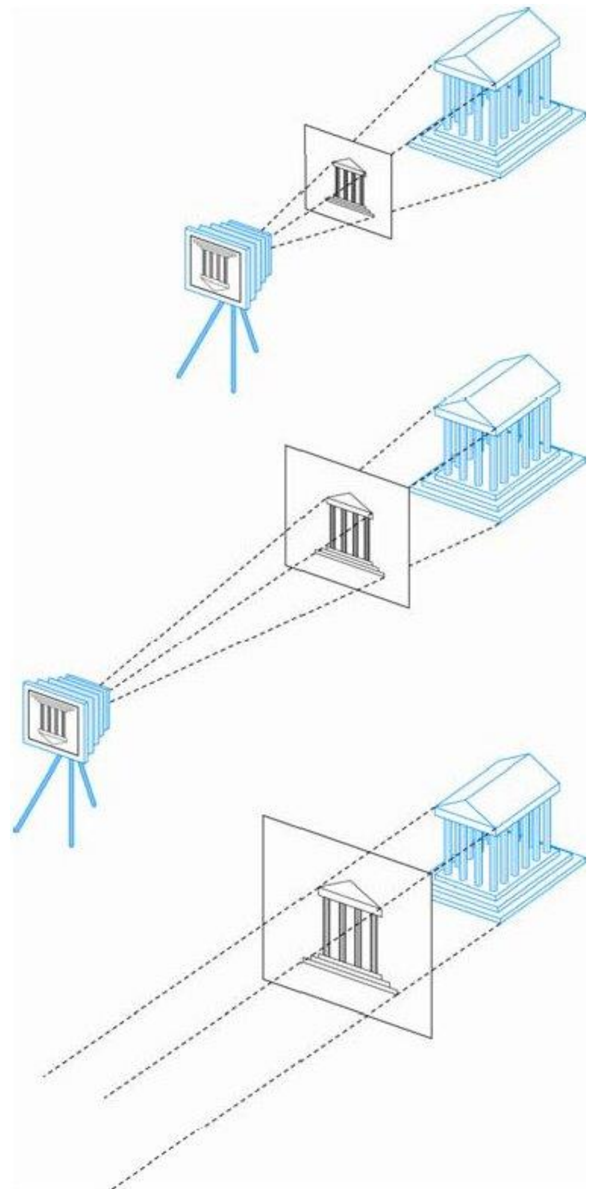




Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Orthographic Viewing

Imagine a camera infinitely far away from image plane



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Transformations and Viewing

- In OpenGL, 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);
```

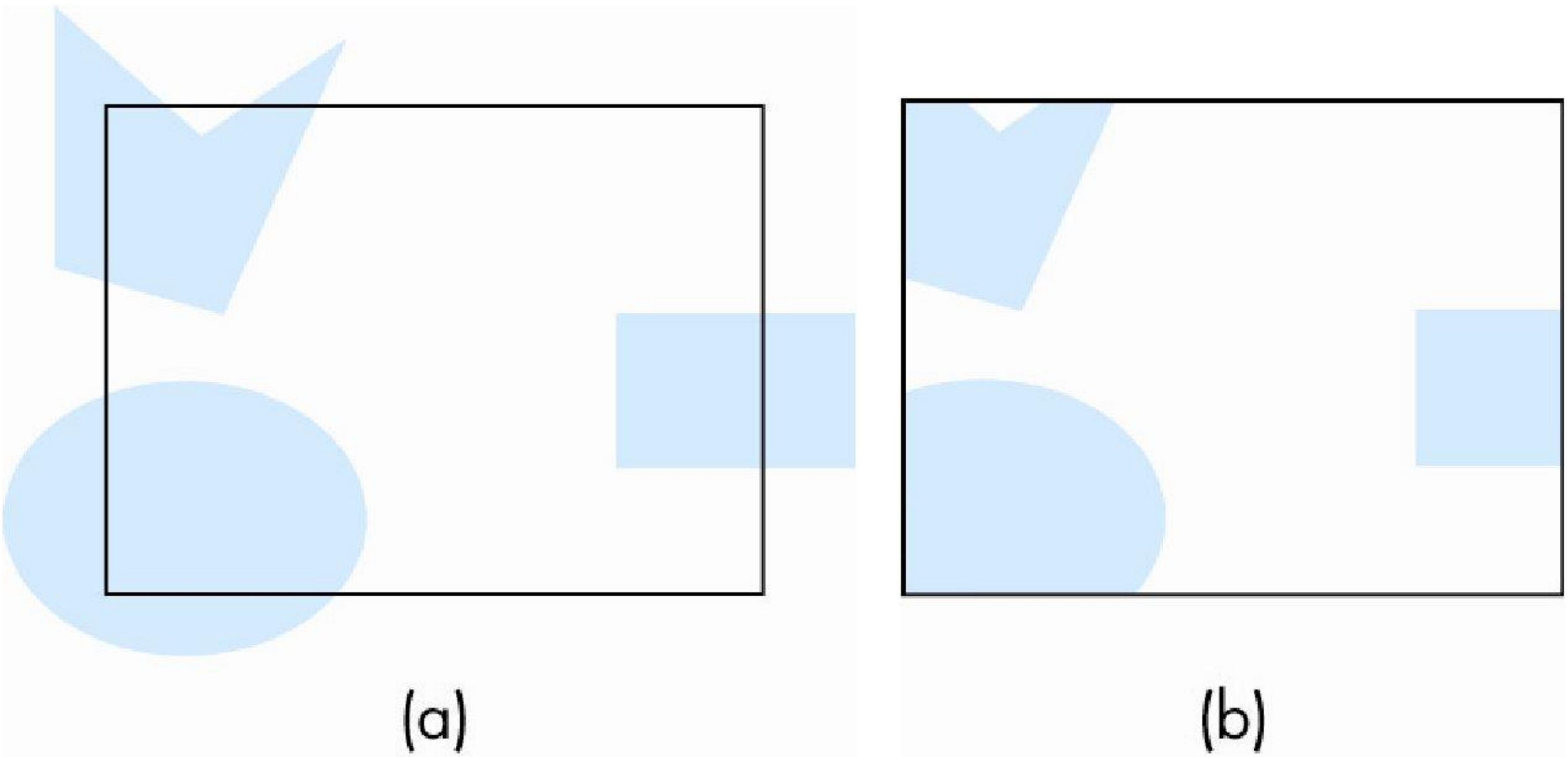
Two- and three-dimensional viewing

- 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

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Clipping



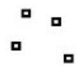
(a) Objects before clipping
(b) Image after clipping

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

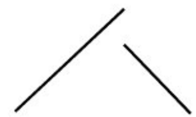
mydisplay.c

```
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();
}
```

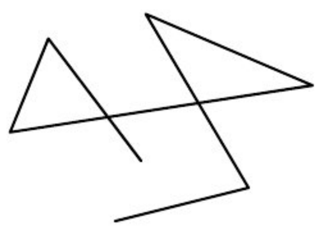
OpenGL Primitives



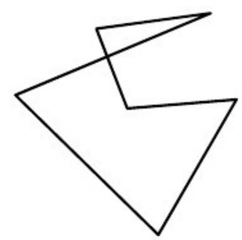
GL_POINTS



GL_LINES

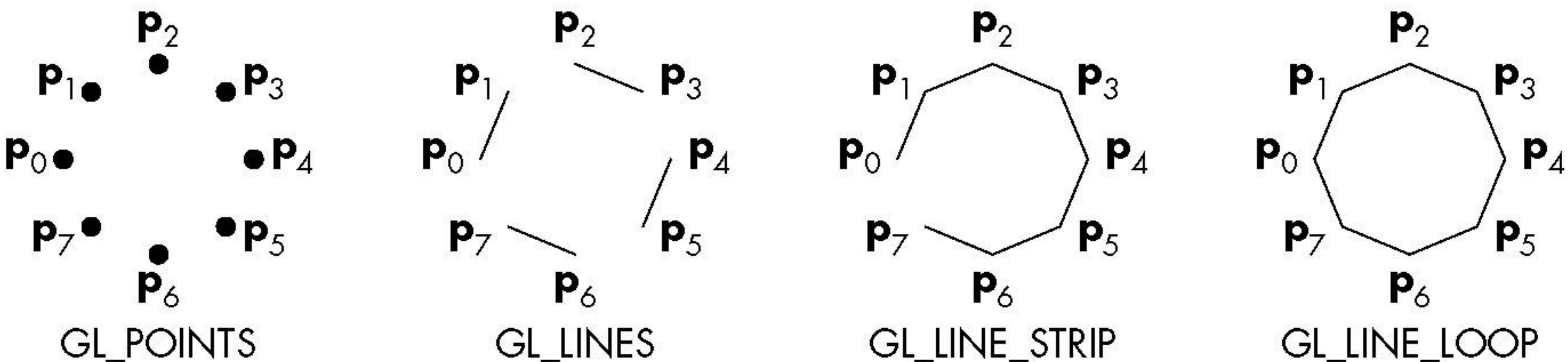


GL_LINE_STRIP



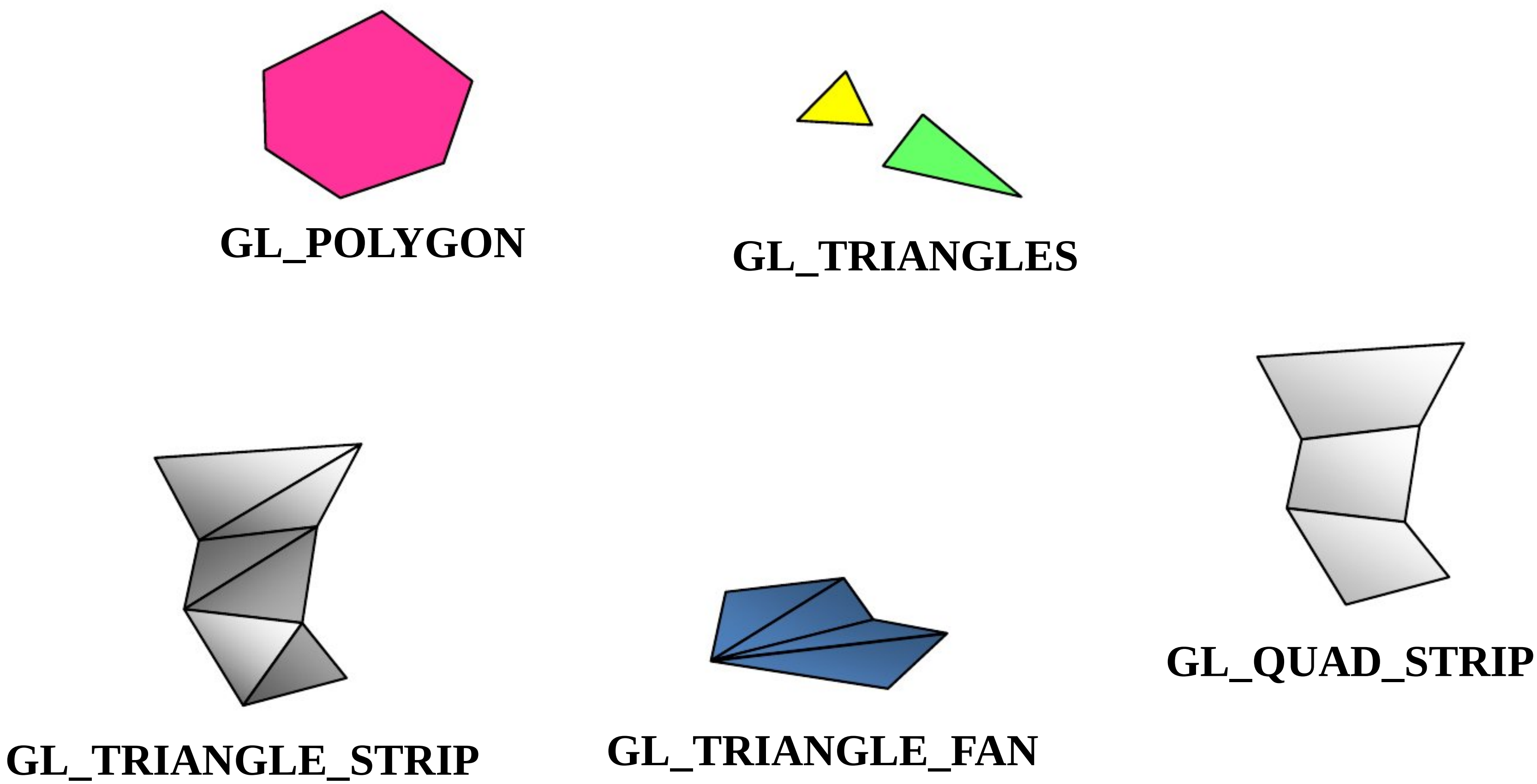
GL_LINE_LOOP

Point and Line Segment Types in OpenGL



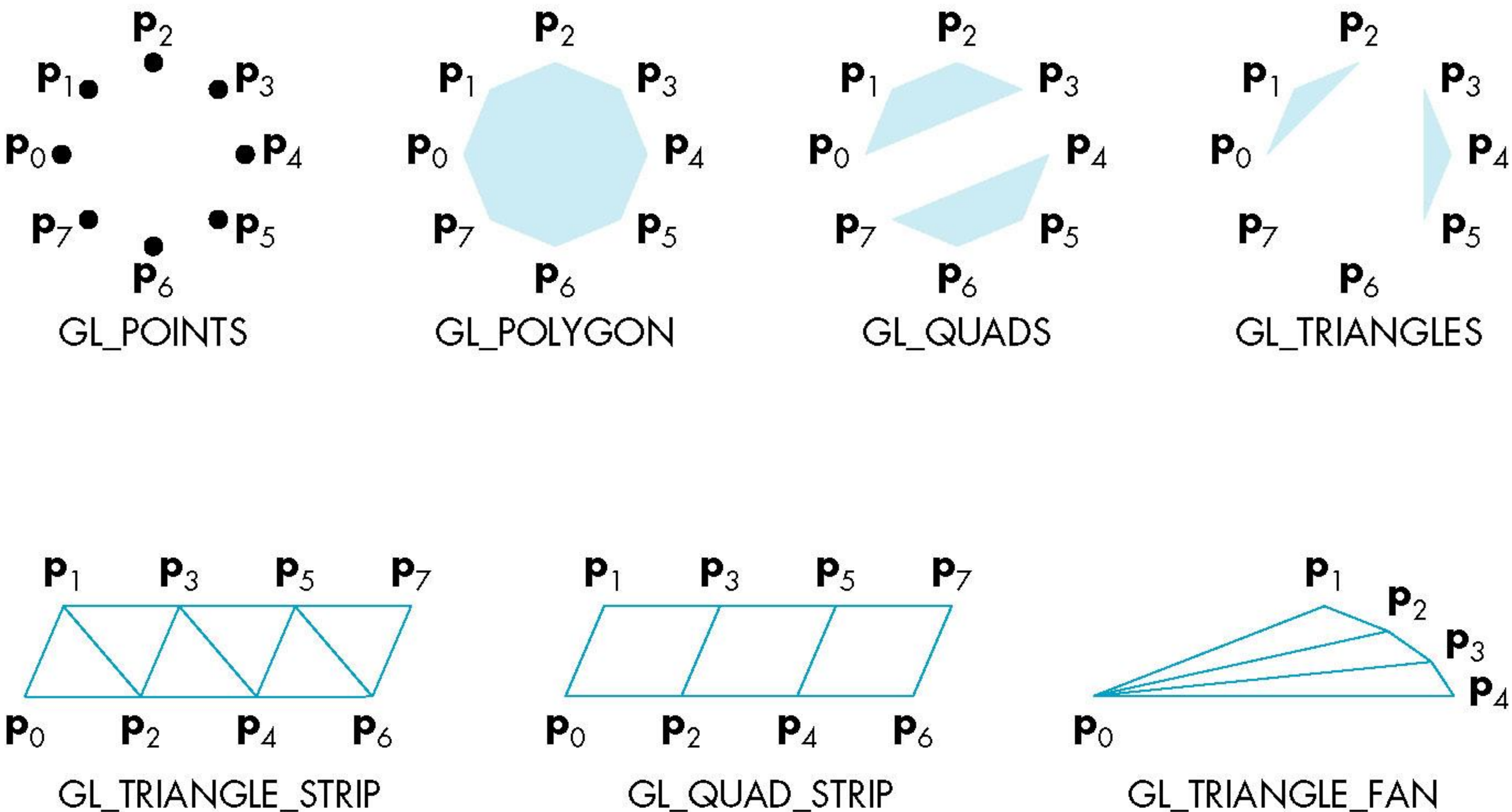
Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

OpenGL Primitives



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

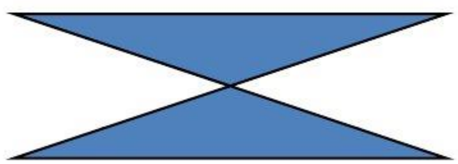
Polygon Types in OpenGL



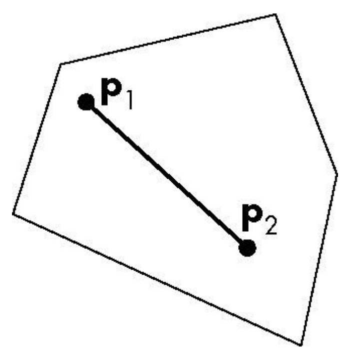
Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

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. (Easier test?)
  - Flat: all vertices are in the same plane
- User program can check if above true
  - OpenGL will produce output if these conditions are violated but it may not be what is desired
- Triangles satisfy all conditions



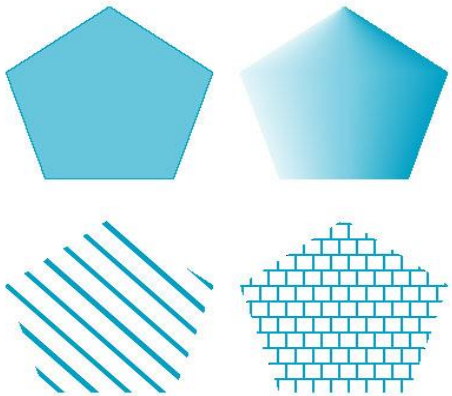
nonsimple polygon



convex polygon

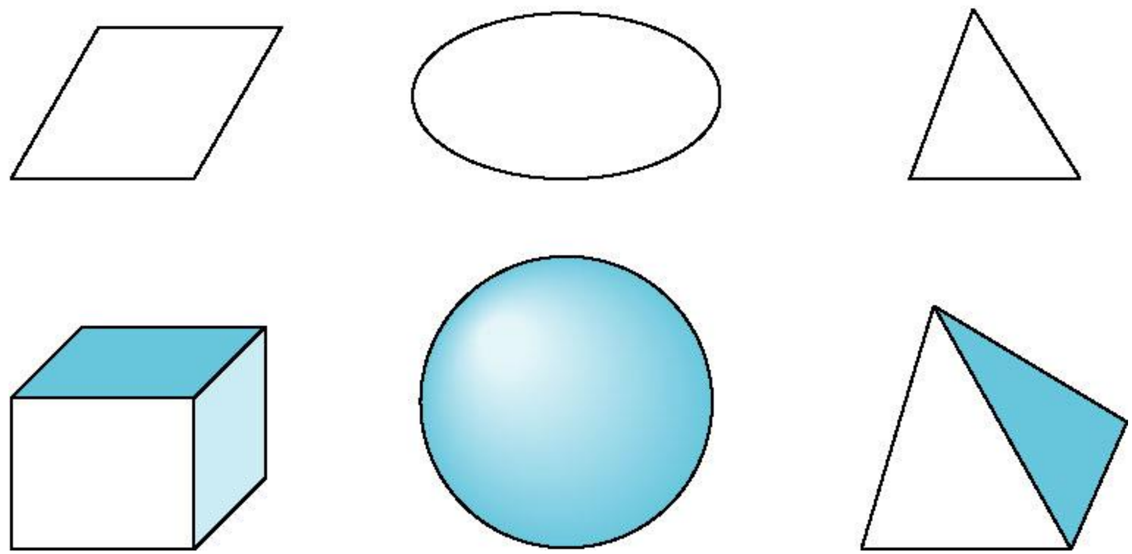
nonconvex polygon

Methods of displaying polygons



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

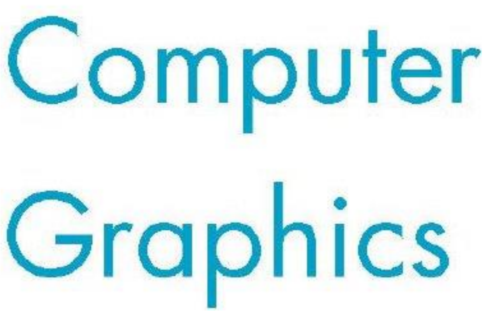
Convex Objects



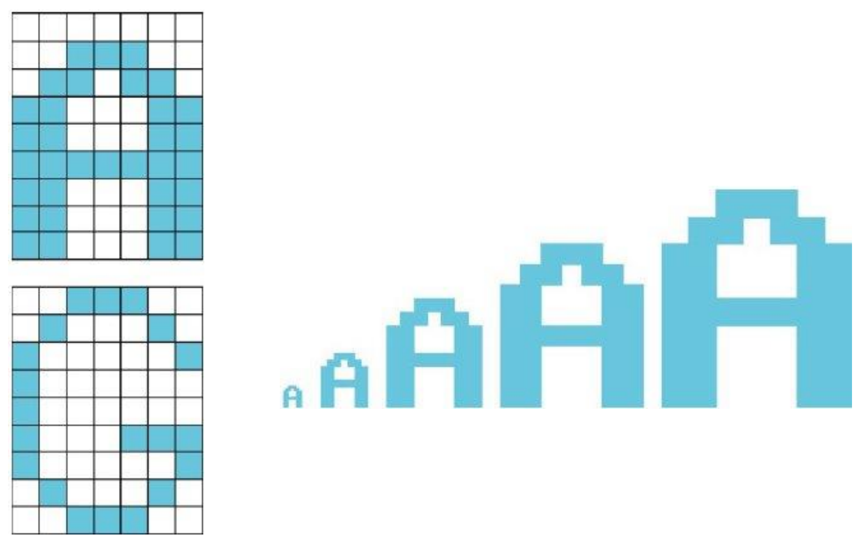
Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Text

- Stroke vs. Raster Text
- Stroke text: vertices that define lines or curves
- Raster text: bit blocks



Stroke text (Postscript font)

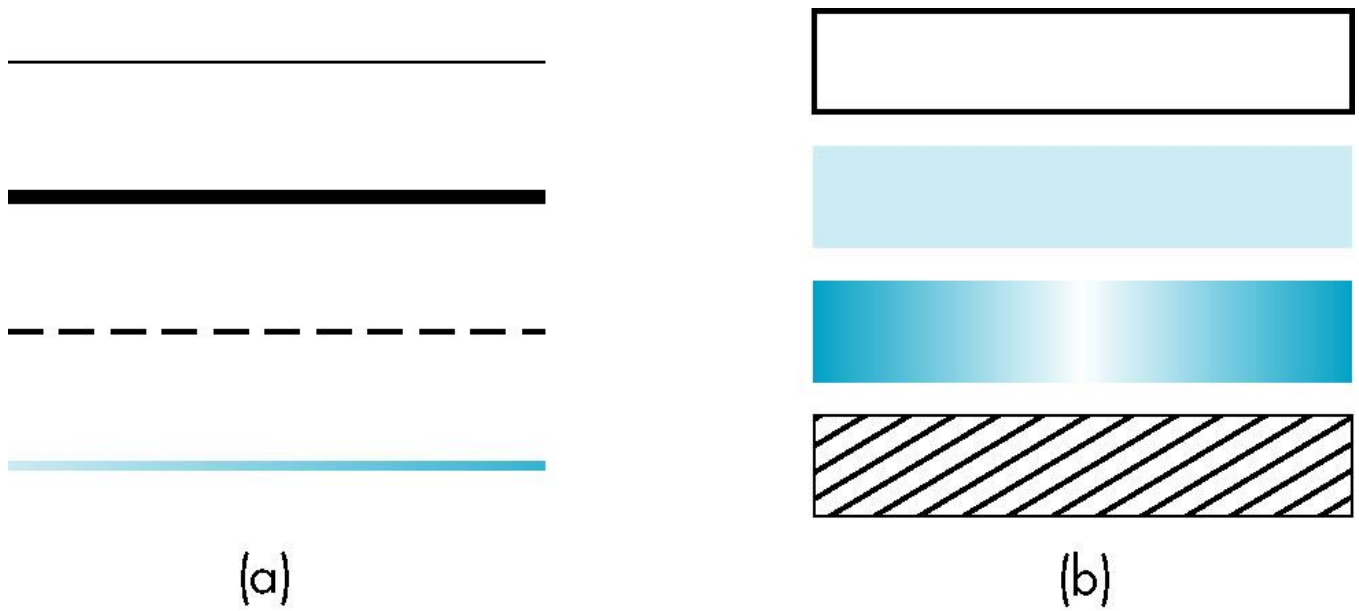


Raster text

Attributes

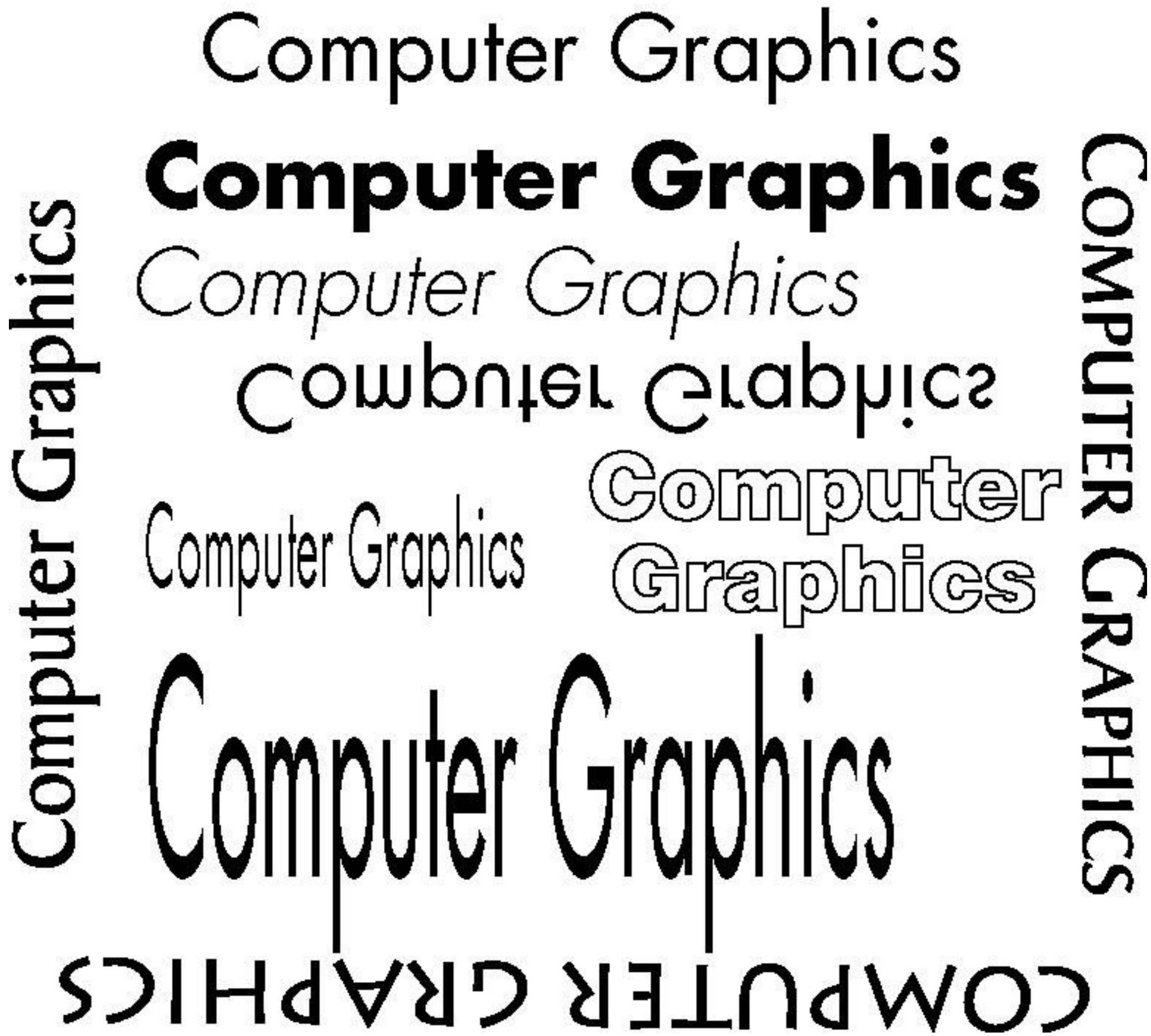
- Attributes are part of the OpenGL state and determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges
    - Display vertices

Attributes for lines and polygons



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

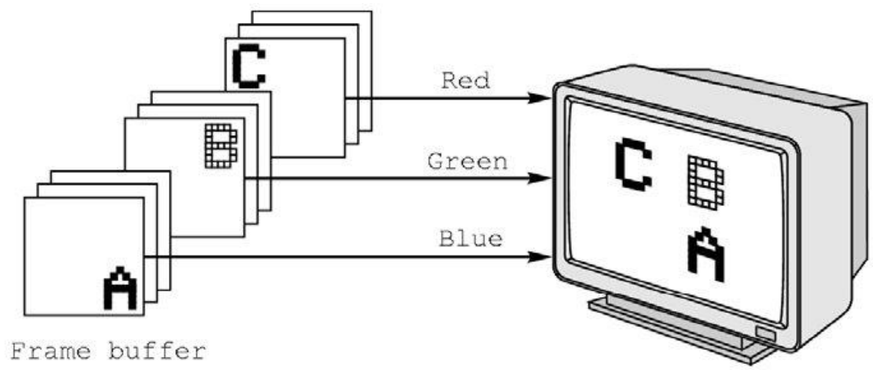
Stroke-text Attributes



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

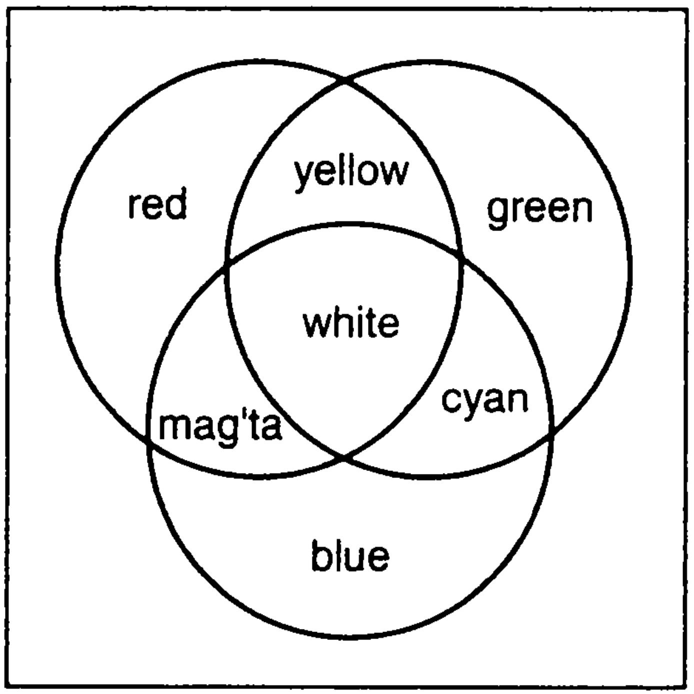
RGB color

- Each color component is 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), whereas in **glColor3ub** the values range from 0 to 255

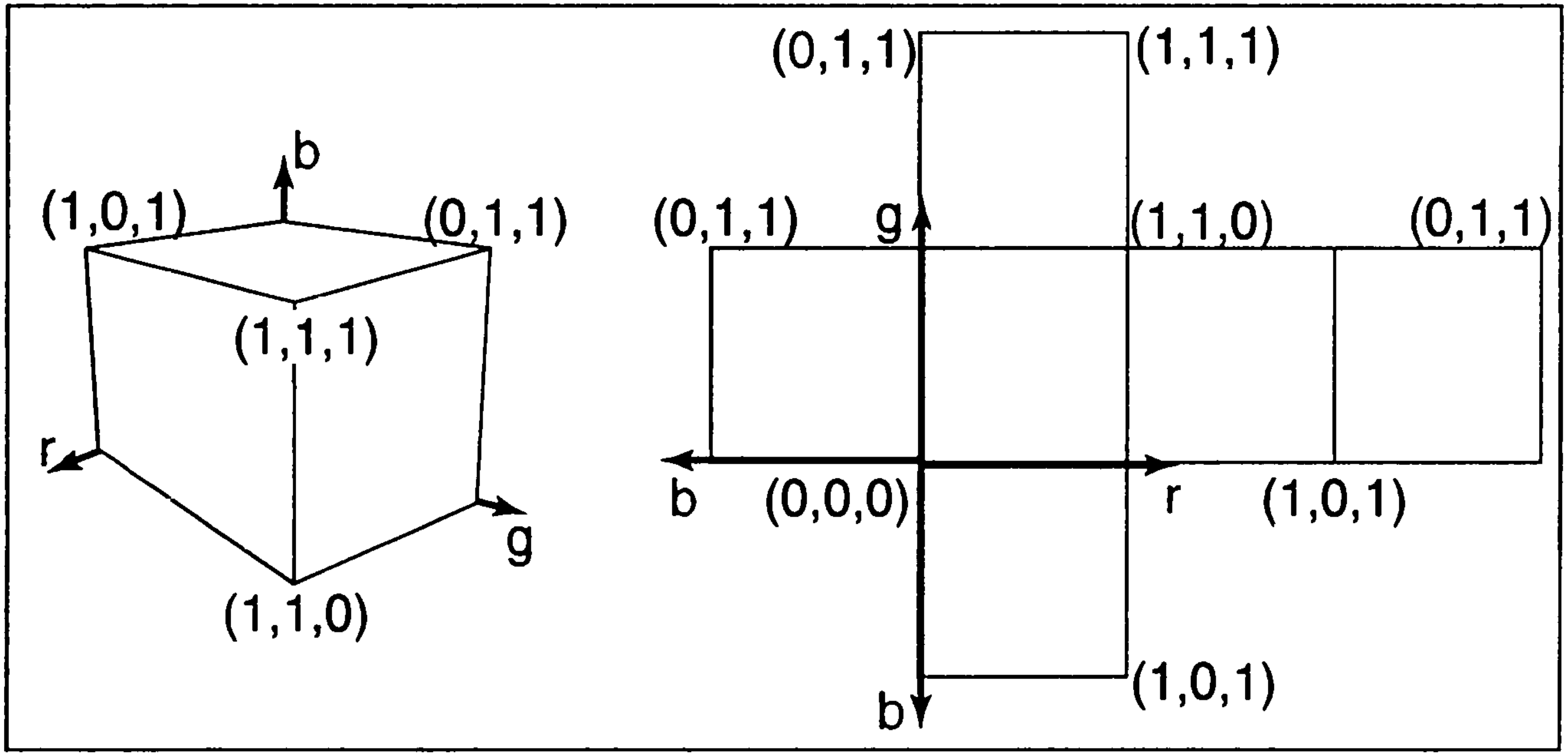


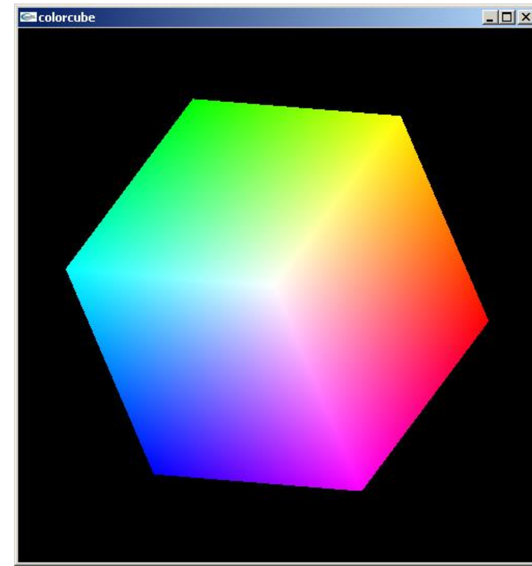
RGB Color

- Additive Colors
  - The additive mixing rules for colors red/green/blue.



The RGB color cube



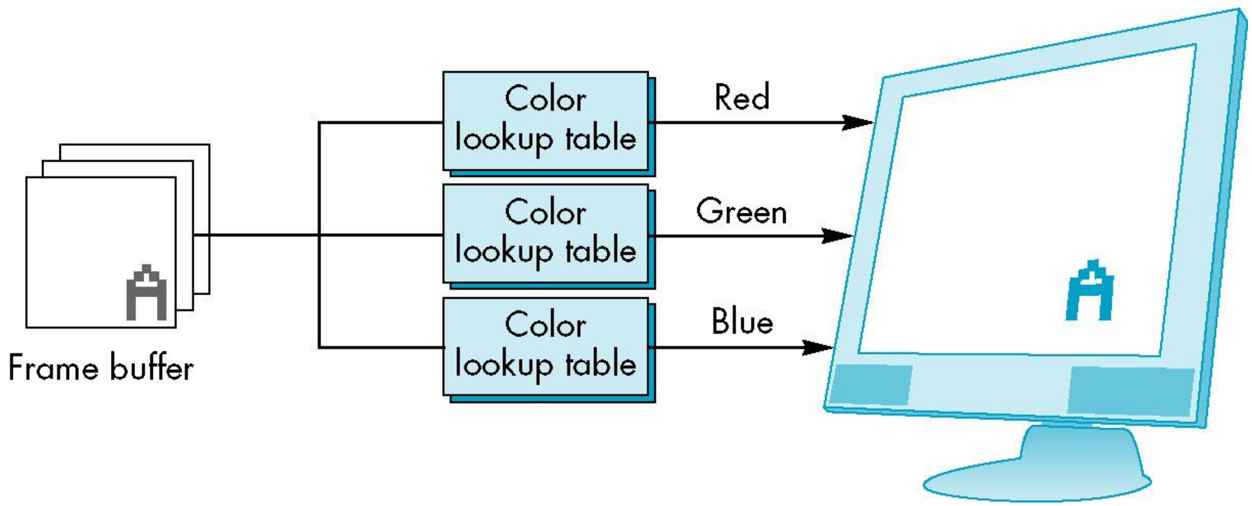


black = (0,0,0)
red = (1,0,0)
green=(0,1,0)
blue= (0,0,1)

yellow = (1,1,0)
magenta = (1,0,1)
cyan = (0,1,1)
white = (1,1,1)

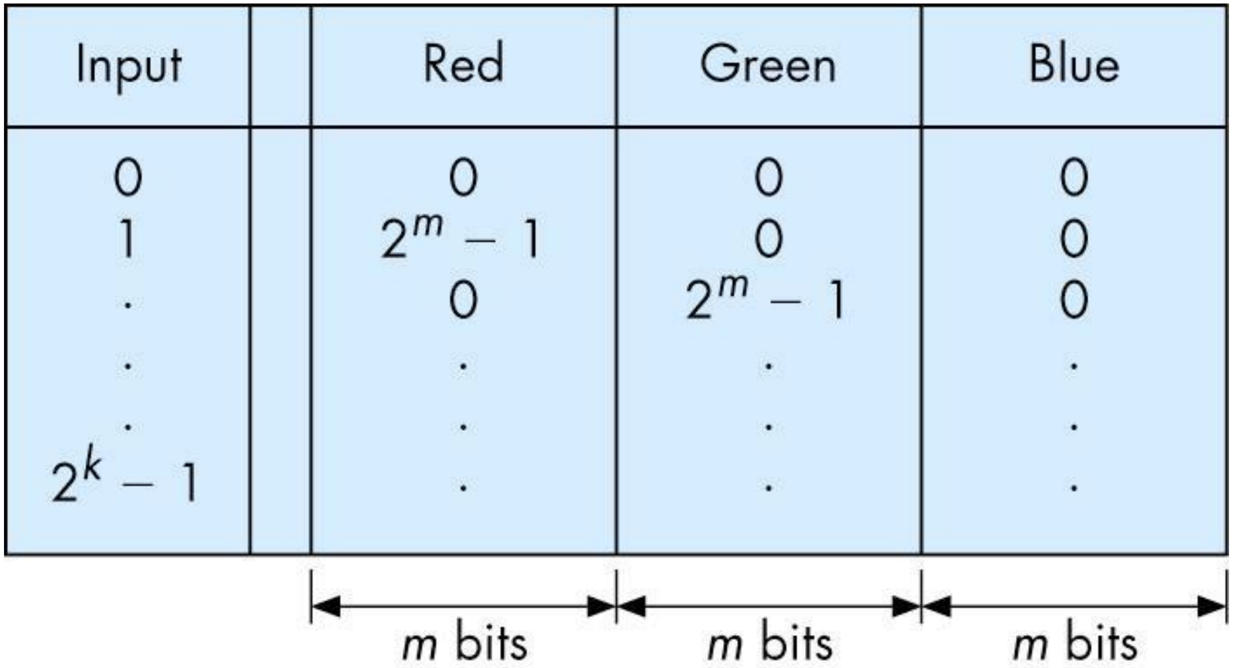
Indexed Color

- 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



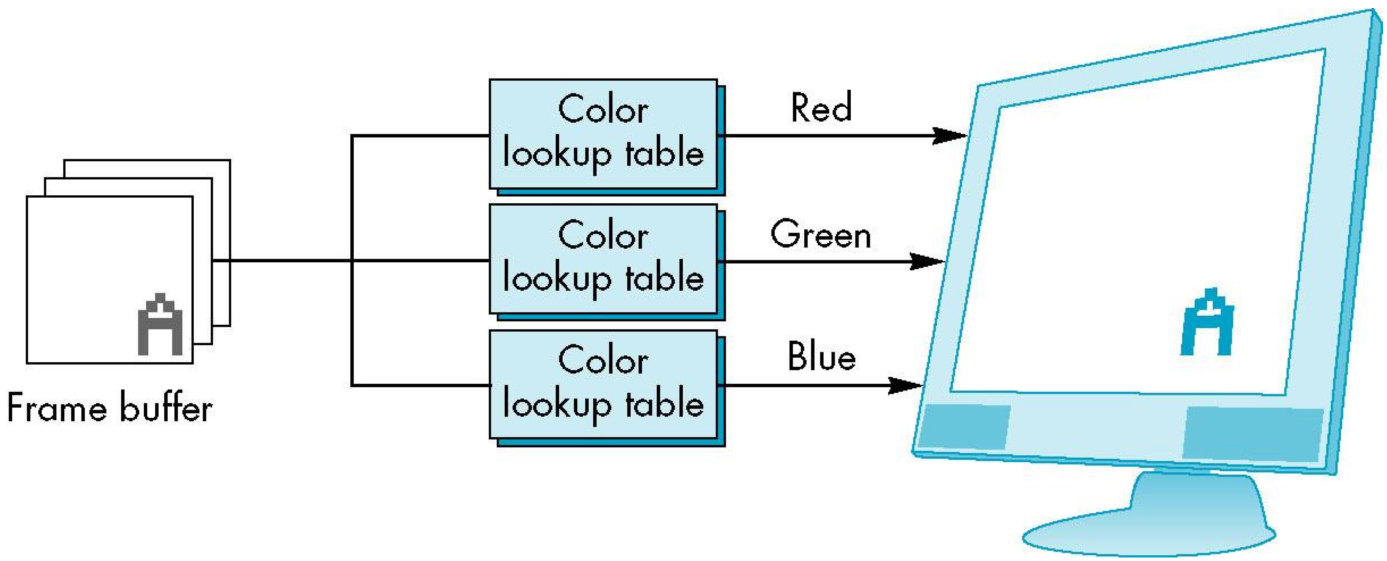
Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Color-lookup Table



Input		Red	Green	Blue
0		0	0	0
1		$2^m - 1$	0	0
.		0	$2^m - 1$	0
.		.	.	.
.		.	.	.
$2^k - 1$		.	.	.

Indexed color



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

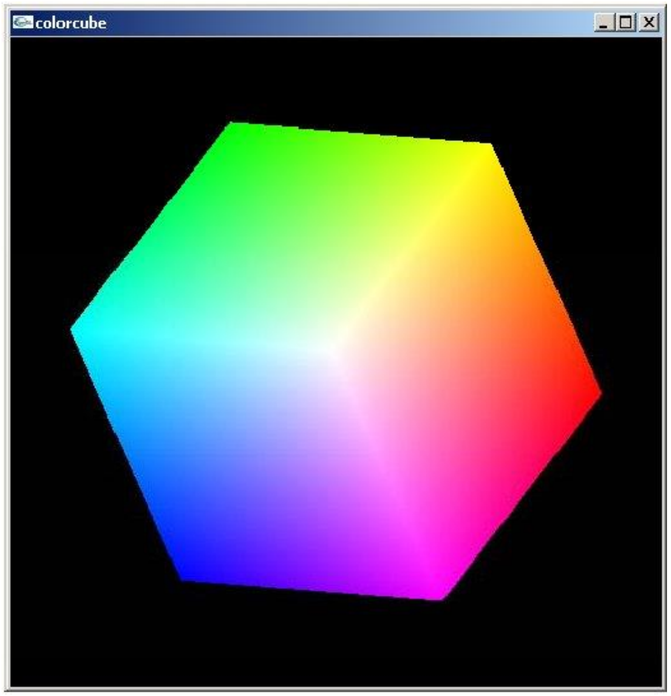
Color and State

- 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

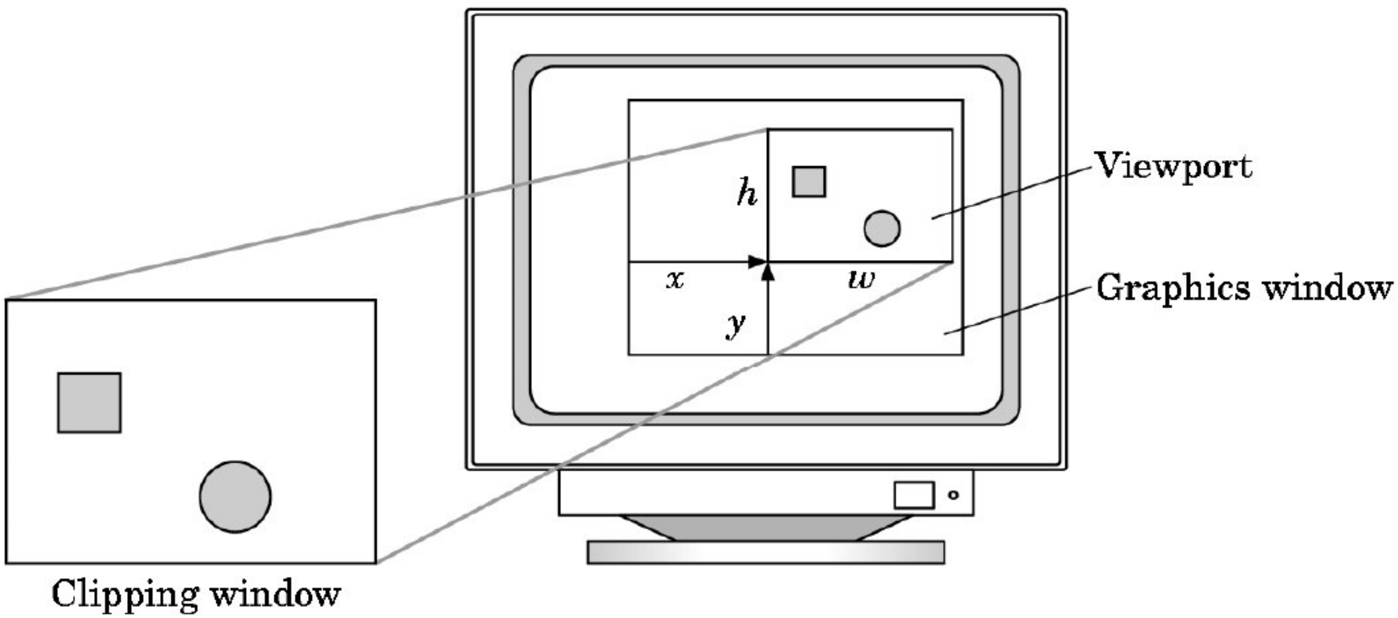
- Default is smooth shading
  – OpenGL *interpolates* vertex colors across visible polygons
- Alternative is flat shading
  – Color of first vertex determines fill color
- **glShadeModel** **(GL_SMOOTH)** or **GL_FLAT**



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

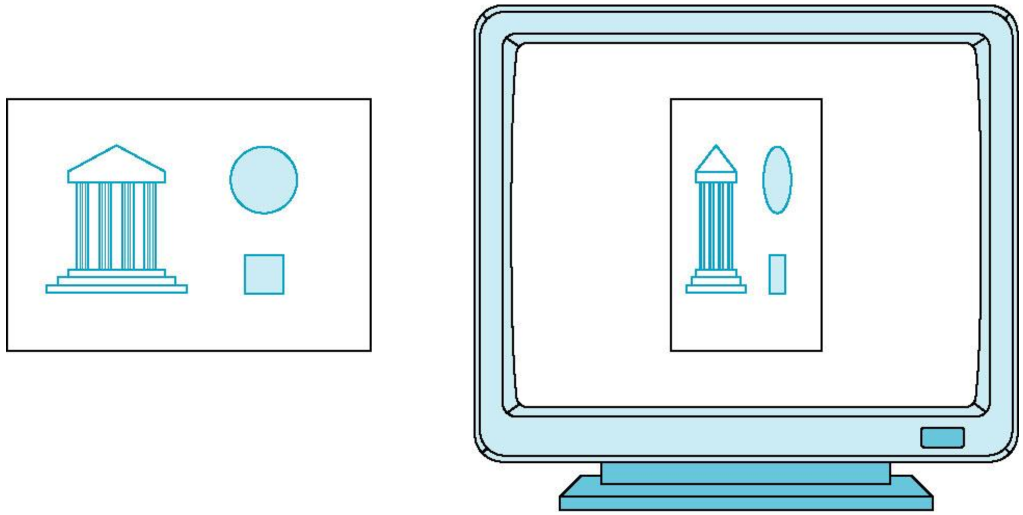
Viewports

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



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Aspect-Ratio Mismatch

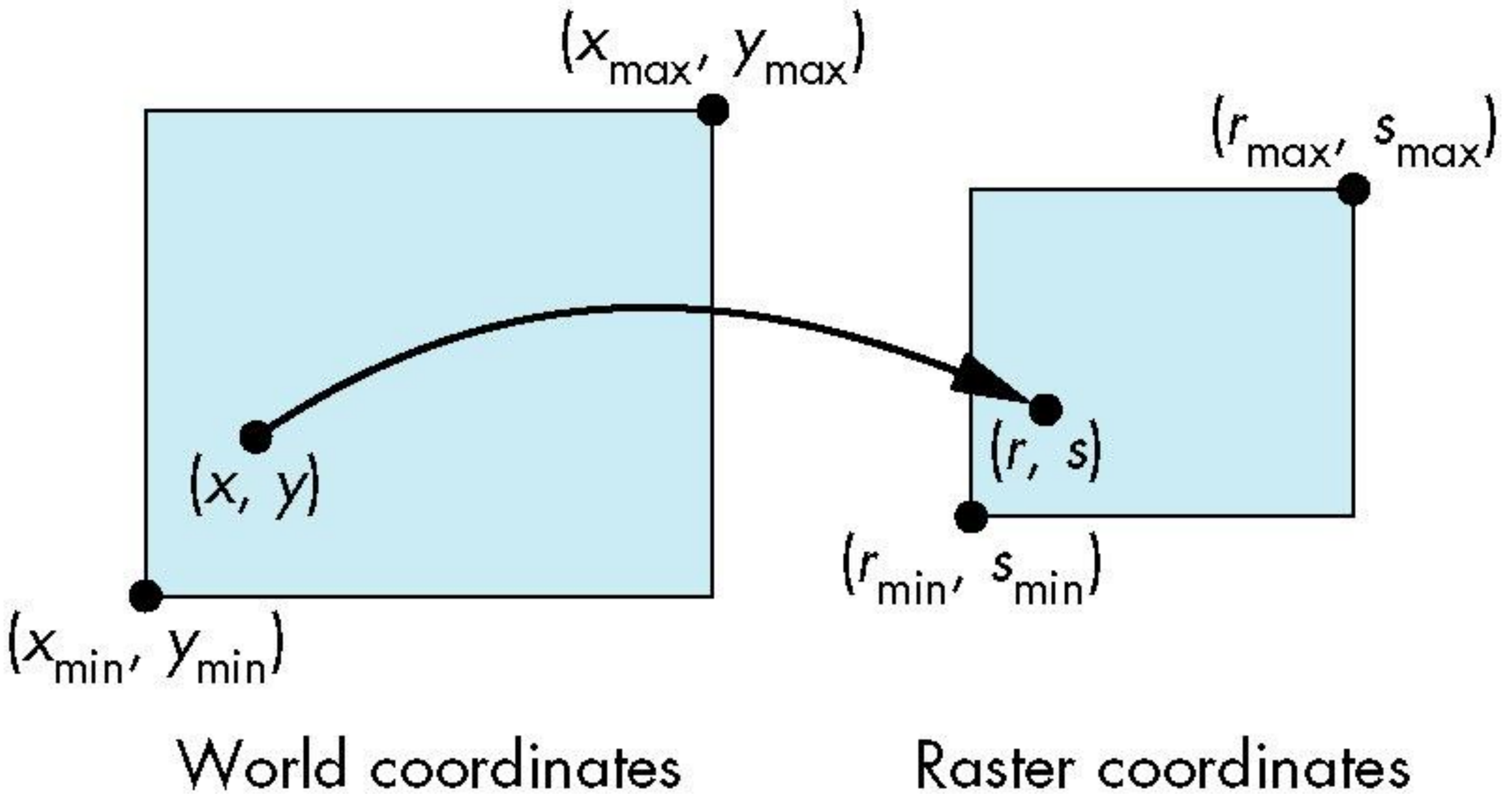


(a) (b)

Viewing rectangle Screen

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Mapping from world coordinates to screen coordinates



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Programming with OpenGL Part 3: Three Dimensions

Objectives

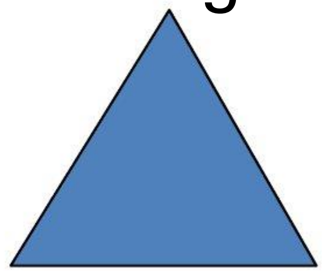
- Develop a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal

Three-dimensional Applications

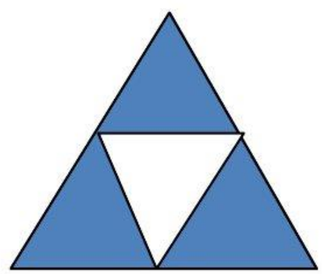
- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
  – Not much changes
  – Use **glVertex3*( )**
  – Have to worry about the order in which polygons are drawn or use hidden-surface removal
  – Polygons should be simple, convex, flat

Sierpinski Gasket (2D)

- Start with a triangle



- Connect bisectors of sides and remove central triangle

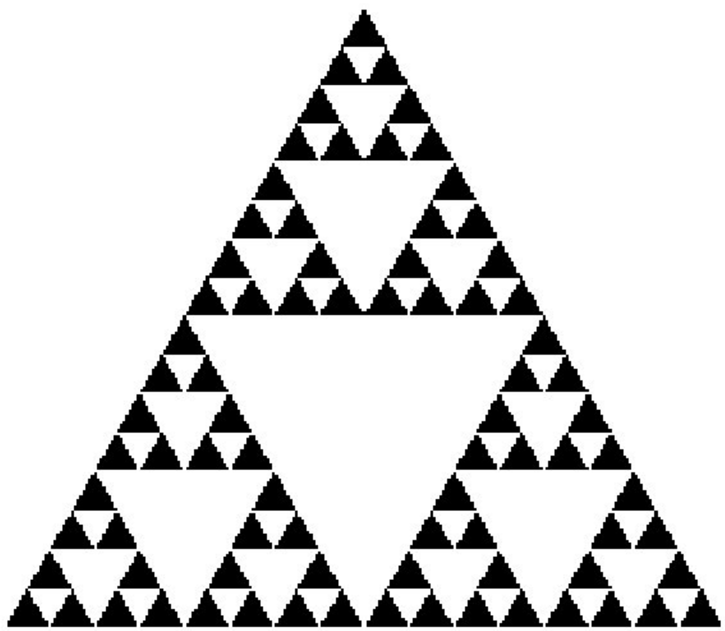


- Repeat

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Example

- Five subdivisions



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

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 is a fractal (fractional dimension) object

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

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Draw one triangle

```
void triangle( GLfloat *a, GLfloat *b,
   GLfloat *c)

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

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Triangle Subdivision

```
void divide_triangle(GLfloat *a, GLfloat *b, GLfloat *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 */
}
```

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

display and init Functions

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

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

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();
}
```

Efficiency Note

By having the **glBegin** and **glEnd** in the display callback rather than in the function **triangle** and using **GL_TRIANGLES** rather than **GL_POLYGON** in **glBegin,** we call **glBegin** and **glEnd** only once for the entire gasket rather than once for each triangle

Moving to 3D

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

GLfloat v[3][3]

glVertex3f

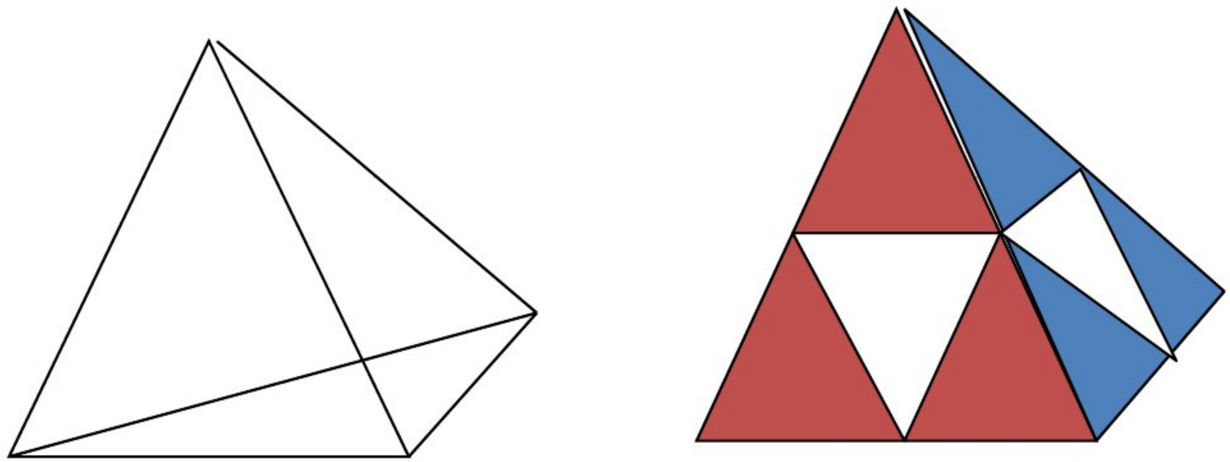
glOrtho

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

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

3D Gasket

- We can subdivide each of the four faces

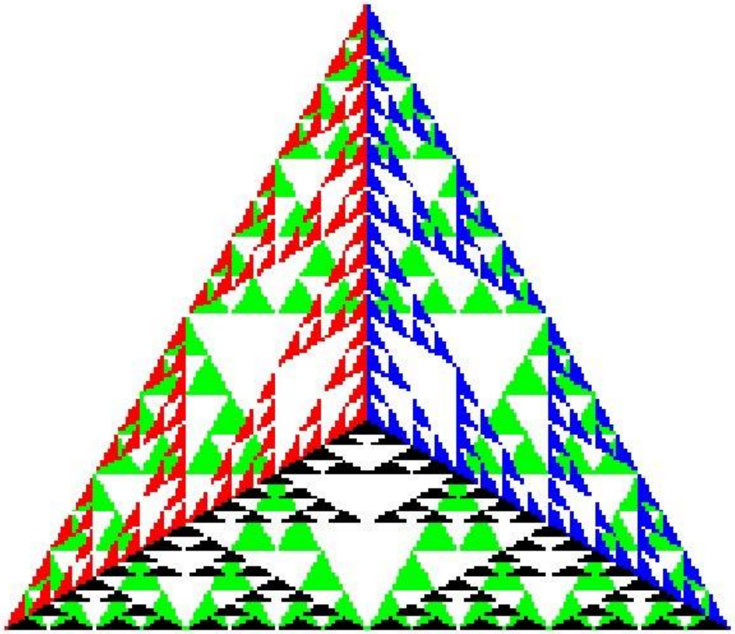


- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Example

after 5 iterations



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

triangle code

```
void triangle( GLfloat *a, GLfloat *b,
   GLfloat *c)
{
        glVertex3fv(a);
        glVertex3fv(b);
        glVertex3fv(c);
}
```

subdivision code

```
void divide_triangle(GLfloat *a, GLfloat *b, GLfloat
    *c, int m)
{
        GLfloat v1[3], v2[3], v3[3];
        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));
}
```

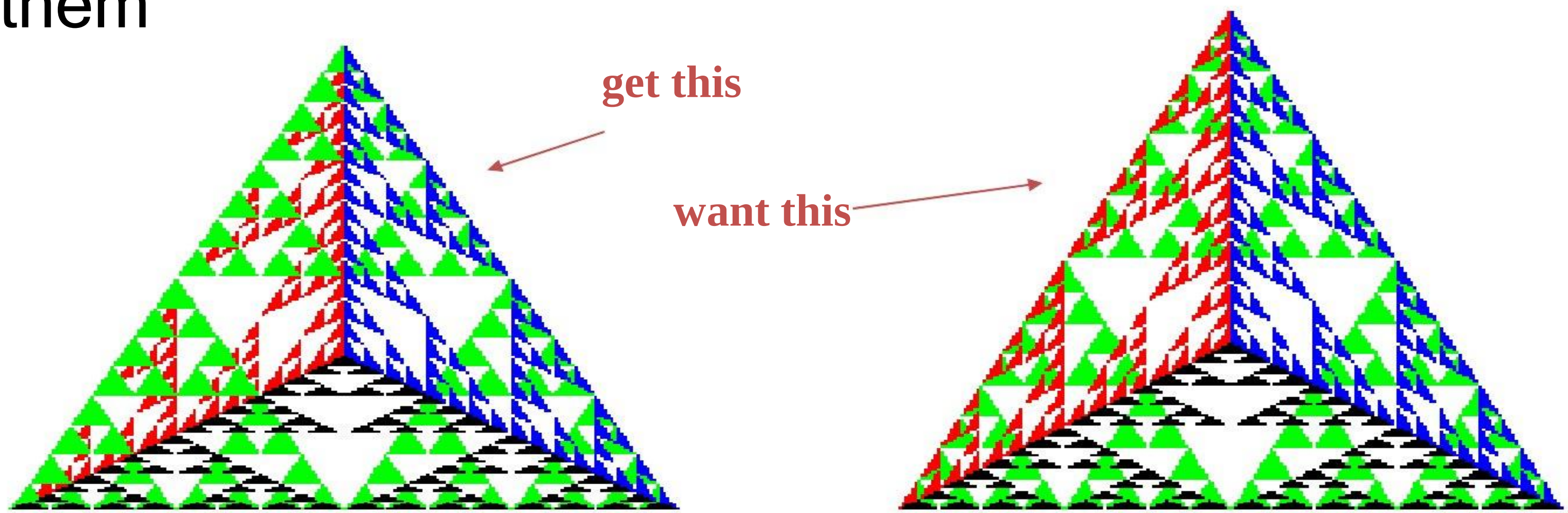
Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

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

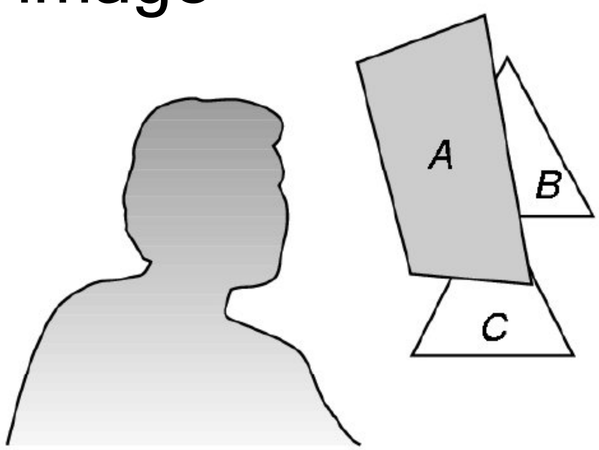
- Because the triangles are drawn in the order they are defined in the program, the front triangles are not always rendered in front of triangles behind them



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

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



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

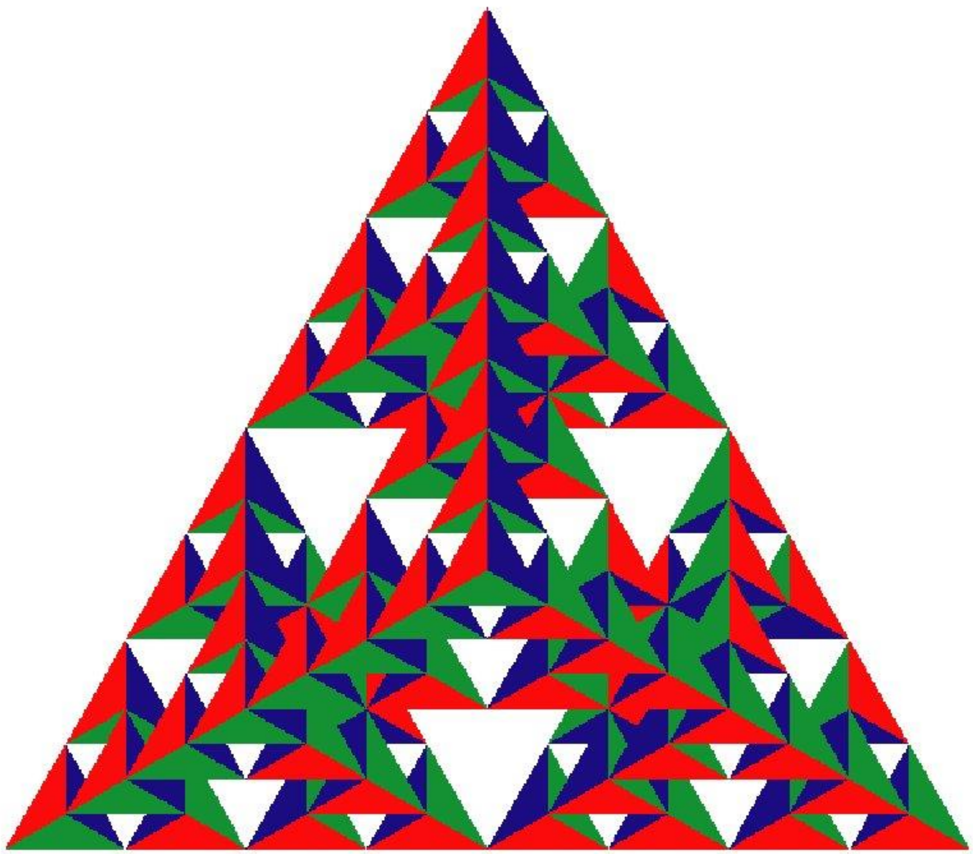
Using the z-buffer algorithm

- 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)**

Surface vs Volume Subdvision

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

Volume Subdivision



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Exercise 2.2

- Consider recursive version of Sierpinski gasket construction
  - What percentage of the area of the triangle remains after the center triangle is removed at each subdivision?
  - What percentage of the perimeter remains?

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Exercise 2.4

- Turtle graphics
  - State x, y, theta
  - Turtle turns and moves to draw line
- How to implement:

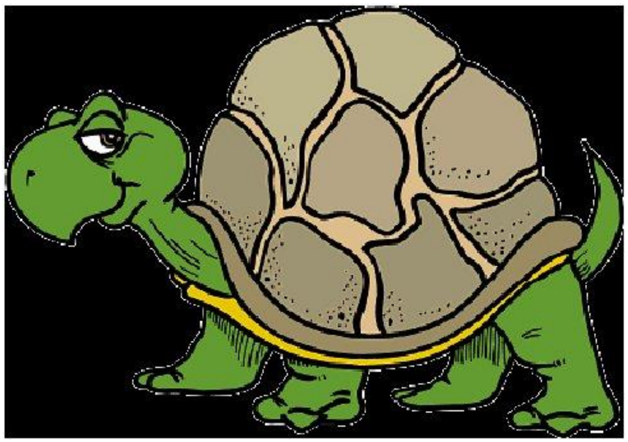
init(x,y,theta);

forward(distance);

right(angle);

left(angle);

pen(up_down);



Exercise 2.9

- Map a point (x,y) from an orthographic clipping rectangle to the viewport of a window on the screen

glViewport(u,v,w,h);

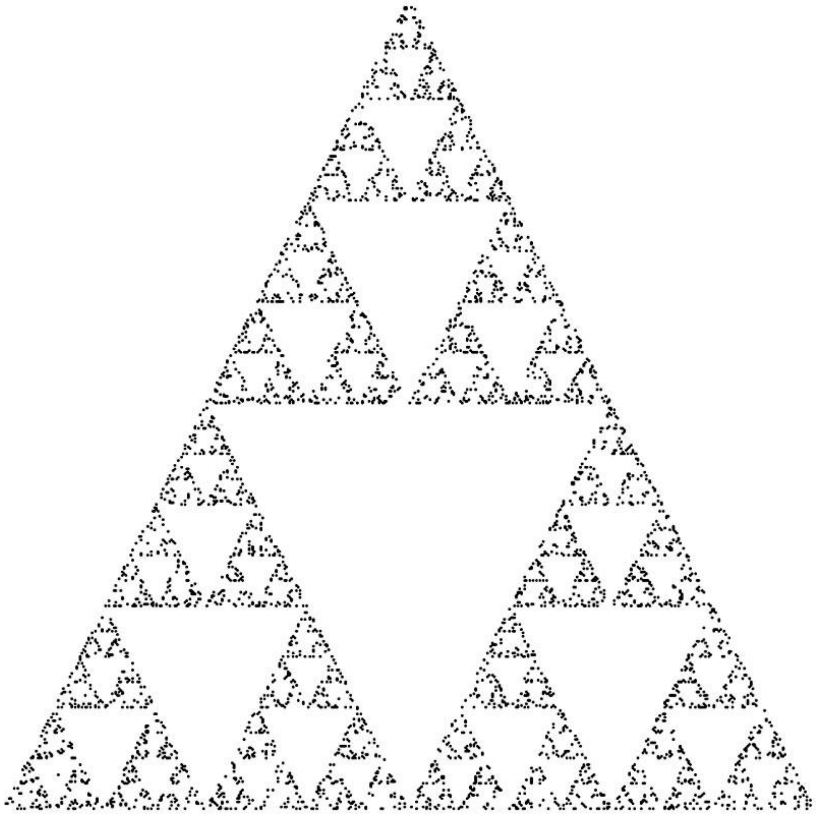
gluOrtho2D(x_min,x_max,y_min, y_max);

Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

Sample Programs

- A.1 gasket.c
- A.2 gasket2.c
- A.3 tetra.c
- A.3’ gasket3.c

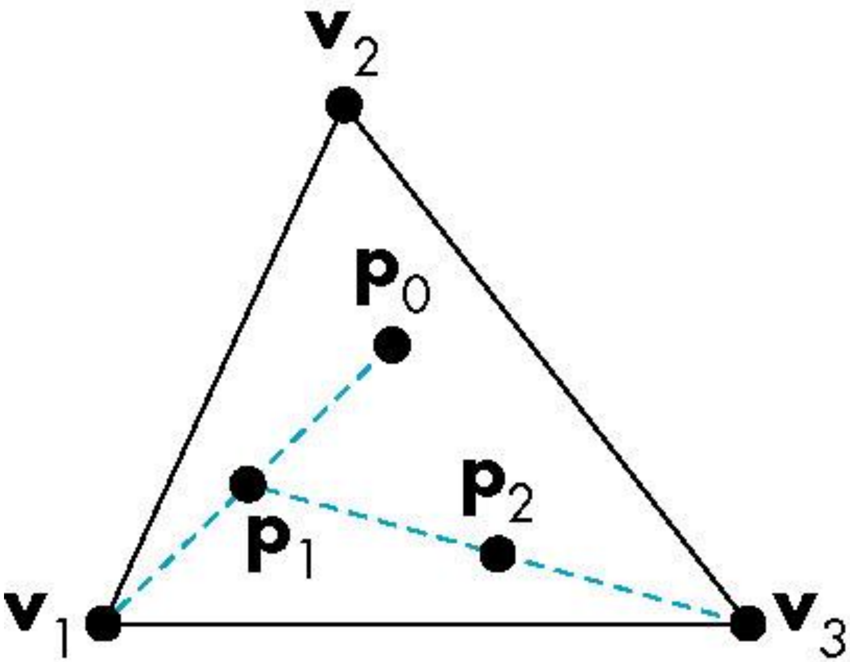
A.1 gasket.c

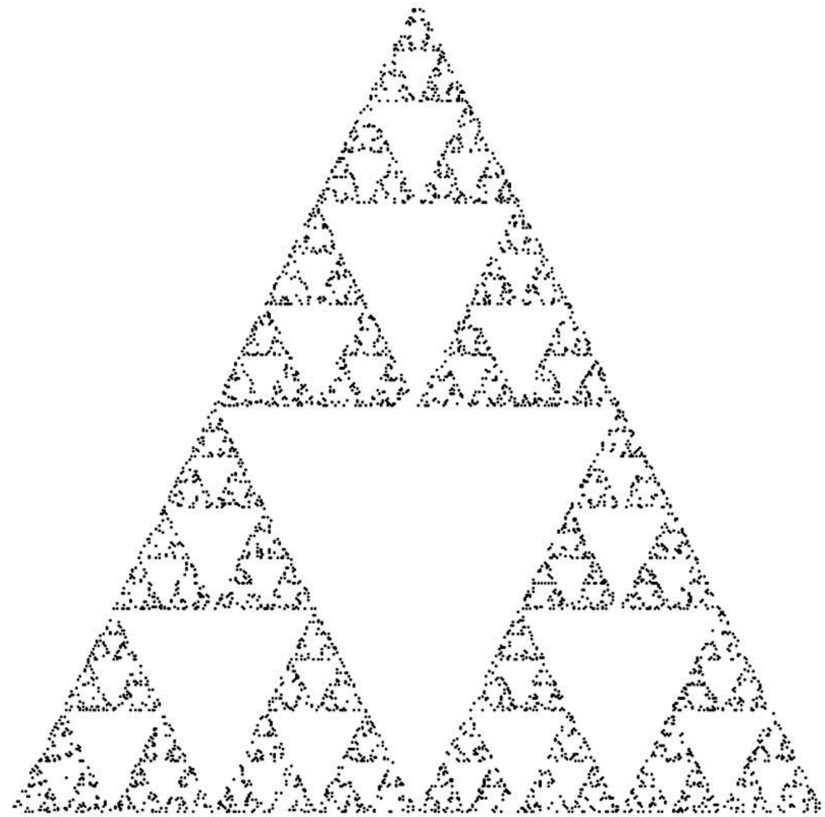


A.1 The Sierpinski Gasket

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 initial point and the randomly selected vertex.
4. Display this new point by putting some sort of marker, such as a small circle, at its location.
5. Replace the initial point with this new point.
6. Return to step 2.

A.1 The Sierpinski Gasket





Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.1 gasket.c (1/3)

```
/* Two-Dimensional Sierpinski Gasket         */
/* Generated Using Randomly Selected Vertices */
/* And Bisection                             */

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

void myinit()
{

/* 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 */
/* 500 x 500 window with origin lower left */

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 50.0, 0.0, 50.0);
    glMatrixMode(GL_MODELVIEW);
}
```


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.1 gasket.c (2/3)

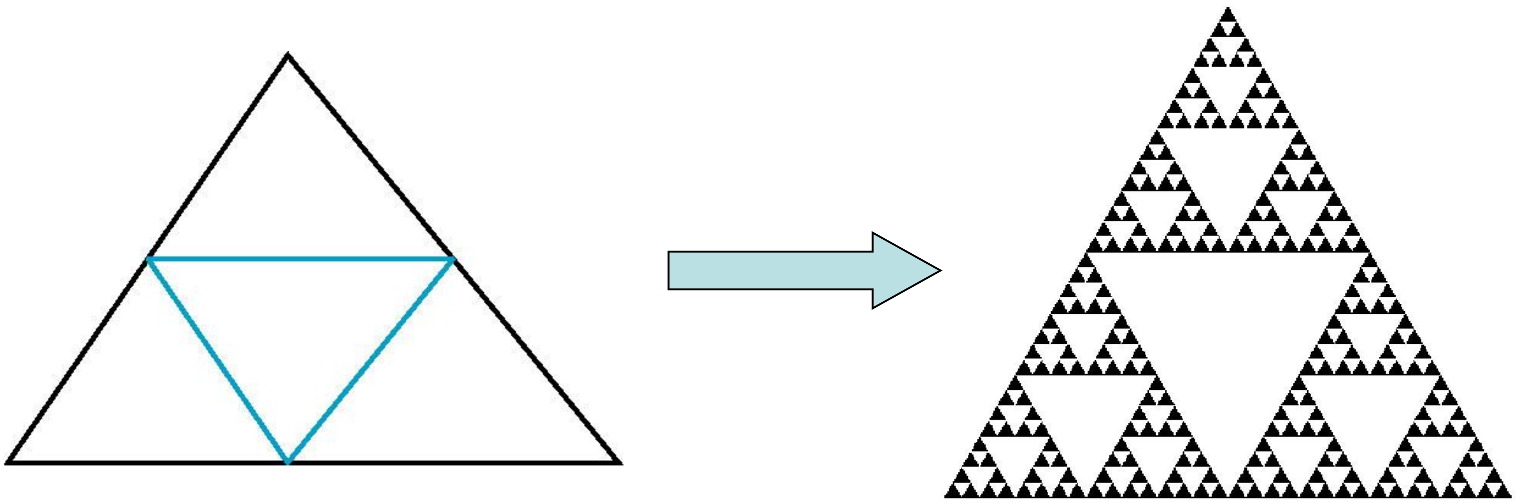
```
void display( void )
{
    GLfloat vertices[3][2]={{0.0,0.0},{25.0,50.0},{50.0,0.0}}; /* A triangle */

    int j, k;
    int rand();       /* standard random number generator */
    GLfloat p[2] ={7.5,5.0};  /* An arbitrary initial point inside traingle */
    glClear(GL_COLOR_BUFFER_BIT);  /*clear the window */
    /* compute and plots 5000 new points */
            glBegin(GL_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 */
        glVertex2fv(p);
     }
             glEnd();
     glFlush(); /* clear buffers */
 }
```


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.1 gasket.c (3/3)

```
void main(int argc, char** argv)
{

/* Standard GLUT initialization */

    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB); /* default, not needed */
    glutInitWindowSize(500,500); /* 500 x 500 pixel window */
    glutInitWindowPosition(0,0); /* place window top left on display */
    glutCreateWindow("Sierpinski Gasket"); /* window title */
    glutDisplayFunc(display); /* display callback invoked when window opened */

    myinit(); /* set attributes */

    glutMainLoop(); /* enter event loop */
}
```

A.2 gasket2.c



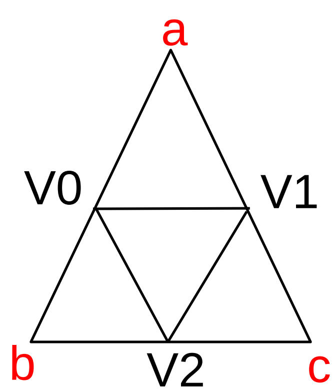
A.2 gasket2.c (1/3)

```
/* gasket2.c   */

/* E. Angel, Interactive Computer Graphics */
/* A Top-Down Approach with OpenGL, Third Edition */
/* Addison-Wesley Longman, 2003 */
/* Recursive subdivision of triangle to form Sierpinski gasket */

#include <GL/glut.h>

typedef float point2[2];

/* initial triangle */
point2 v[]={{-1.0, -0.58}, {1.0, -0.58}, {0.0, 1.15}};
int n;

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

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


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.2 gasket2.c (2/3)

```
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 */
}
```



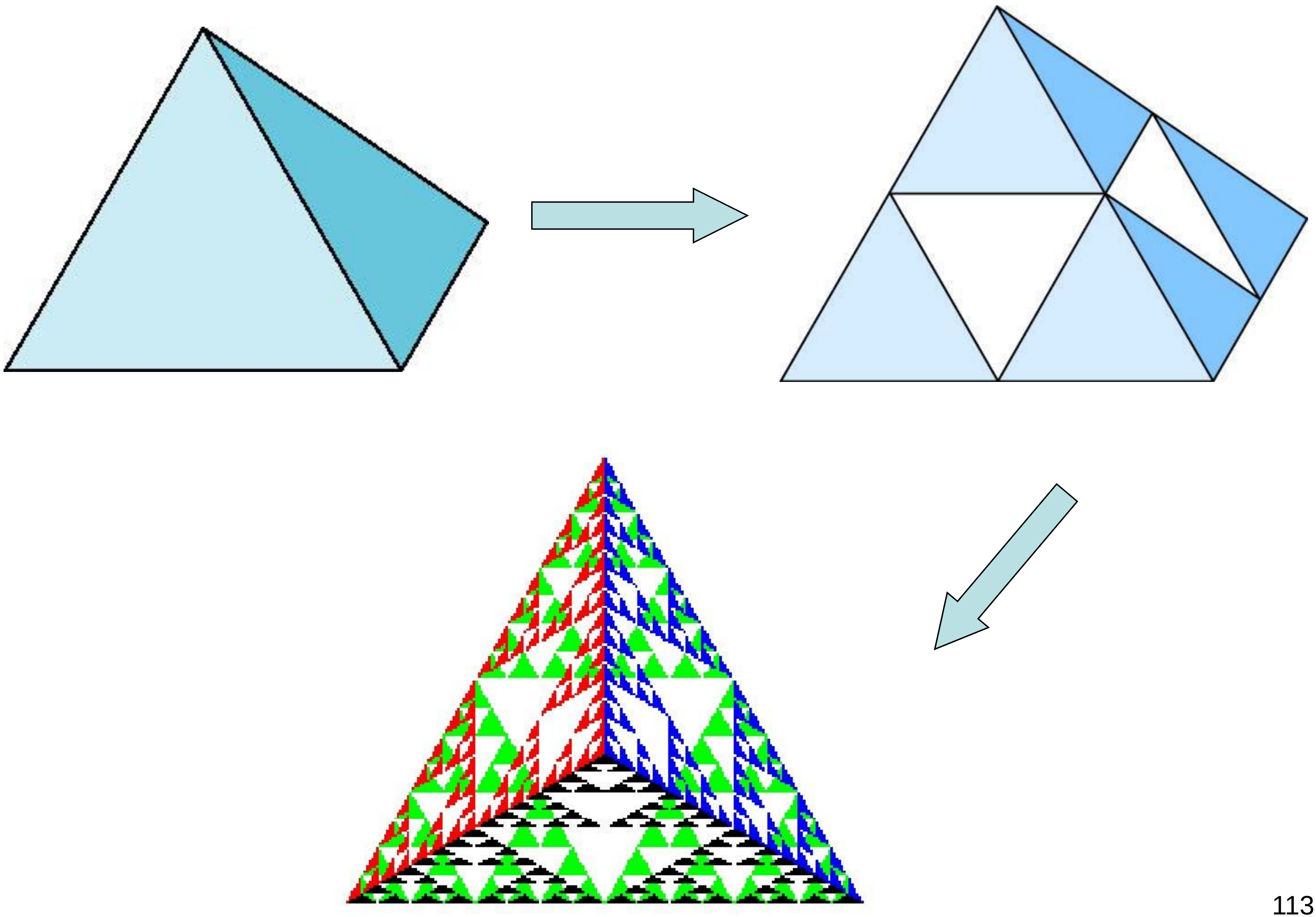

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    divide_triangle(v[0], v[1], v[2], n);
    glFlush();
}
```


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.2 gasket2.c (3/3)

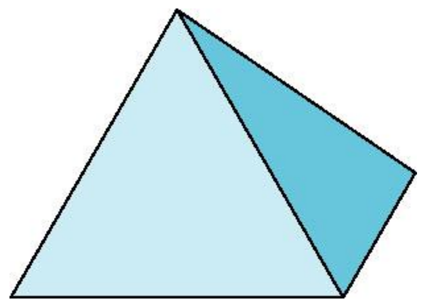
```
void myinit()
{

    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);
}

void
main(int argc, char **argv)
{
    n=4;
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
    glutInitWindowSize(500, 500);
    glutCreateWindow("3D Gasket");
    glutDisplayFunc(display);
    myinit();
    glutMainLoop();
}
```

A.3 tetra.c



A.3 tetra.c (1/5)

```
/* Recursive subdivision of tetrahedron to form
  3D Sierpinski gasket */

#include <stdlib.h>
#ifdef __APPLE_
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

typedef float point[3];

/* initial tetrahedron */

point v[]={{0.0, 0.0, 1.0}, {0.0, 0.942809, -0.33333},
        {-0.816497, -0.471405, -0.333333}, {0.816497, -0.471405, -0.333333}};

static GLfloat theta[] = {0.0,0.0,0.0};

int n;
```



$(0.0,\ 0.0,\ 1.0)$

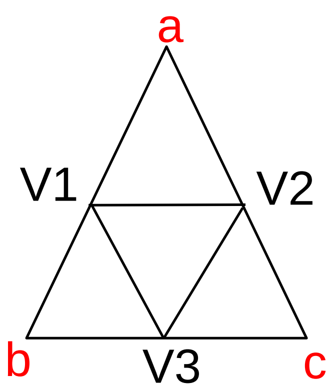
$(0.0, 2\sqrt{2}/3,\ \text{-}1/3)$

$(-\sqrt{6}/3, -\sqrt{2}/3,\ \text{-}1/3)$

$(\sqrt{6}/3, -\sqrt{2}/3,\ \text{-}1/3)$

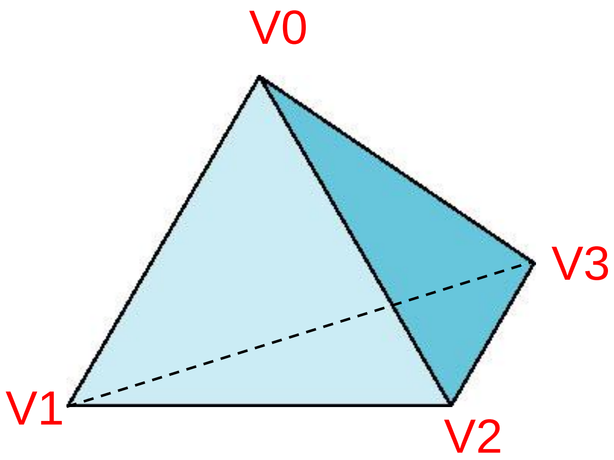
A.3 tetra.c (2/5)

```
void triangle( point a, point b, point c)
/* display one triangle using a line loop for wire frame, a single
normal for constant shading, or three normals for interpolative shading */
{
    glBegin(GL_POLYGON);
       glNormal3fv(a);
       glVertex3fv(a);
       glVertex3fv(b);
       glVertex3fv(c);
    glEnd();
}
void divide_triangle(point a, point b, point c, int m)
{
/* triangle subdivision using vertex numbers
righthand rule applied to create outward pointing faces */
    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)); /* draw triangle at end of recursion */
}
```



A.3 tetra.c (3/5)

```
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);
}

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



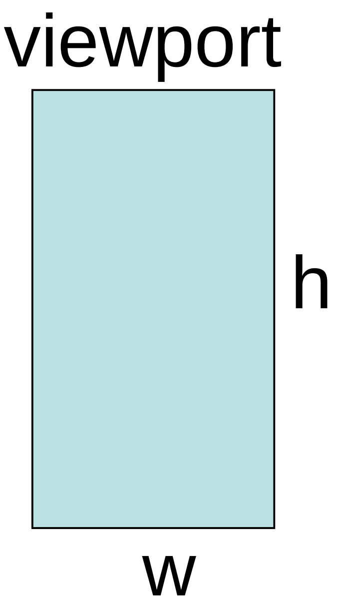
A.3 tetra.c (4/5)

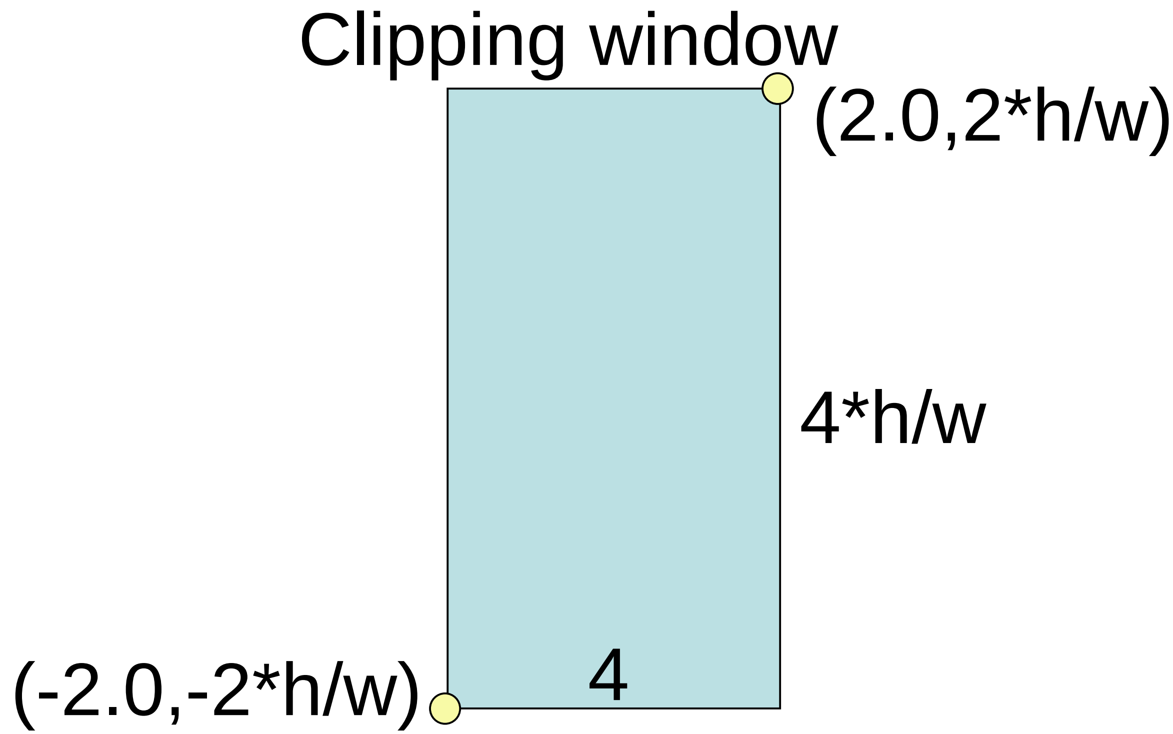
```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
            2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
            2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();
}
void  main(int argc, char **argv)
{
    n=4;
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("3D Gasket");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glEnable(GL_DEPTH_TEST);
    glClearColor (1.0, 1.0, 1.0, 1.0);
    glutMainLoop();
}
```

Reshape Function

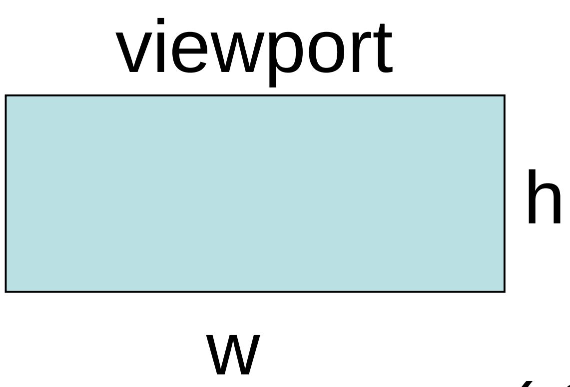
A.3 tetra.c (5/5)

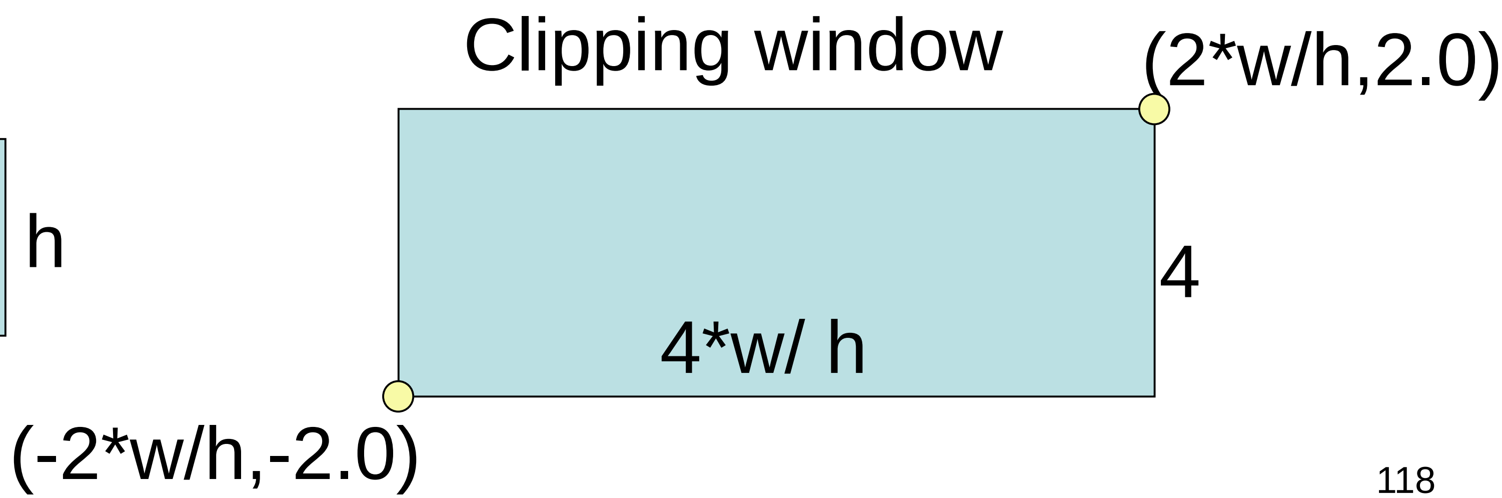
Case 1: w ≤ h





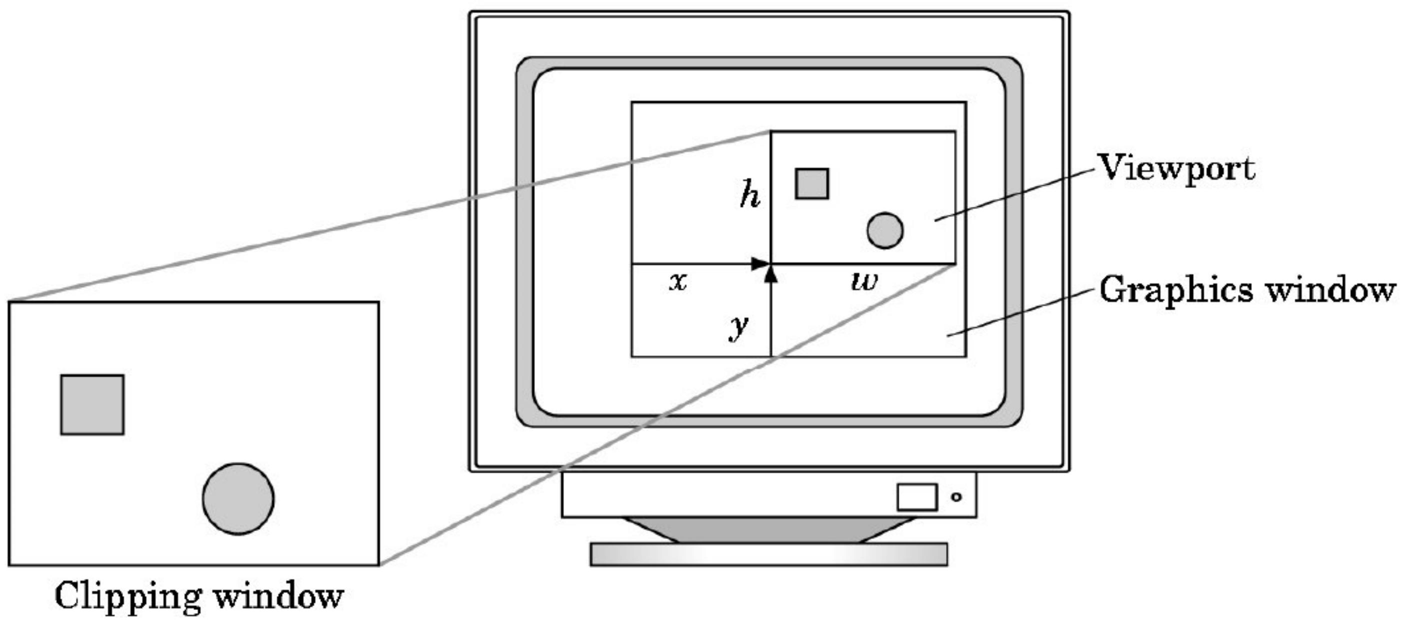
Case 2: w > h





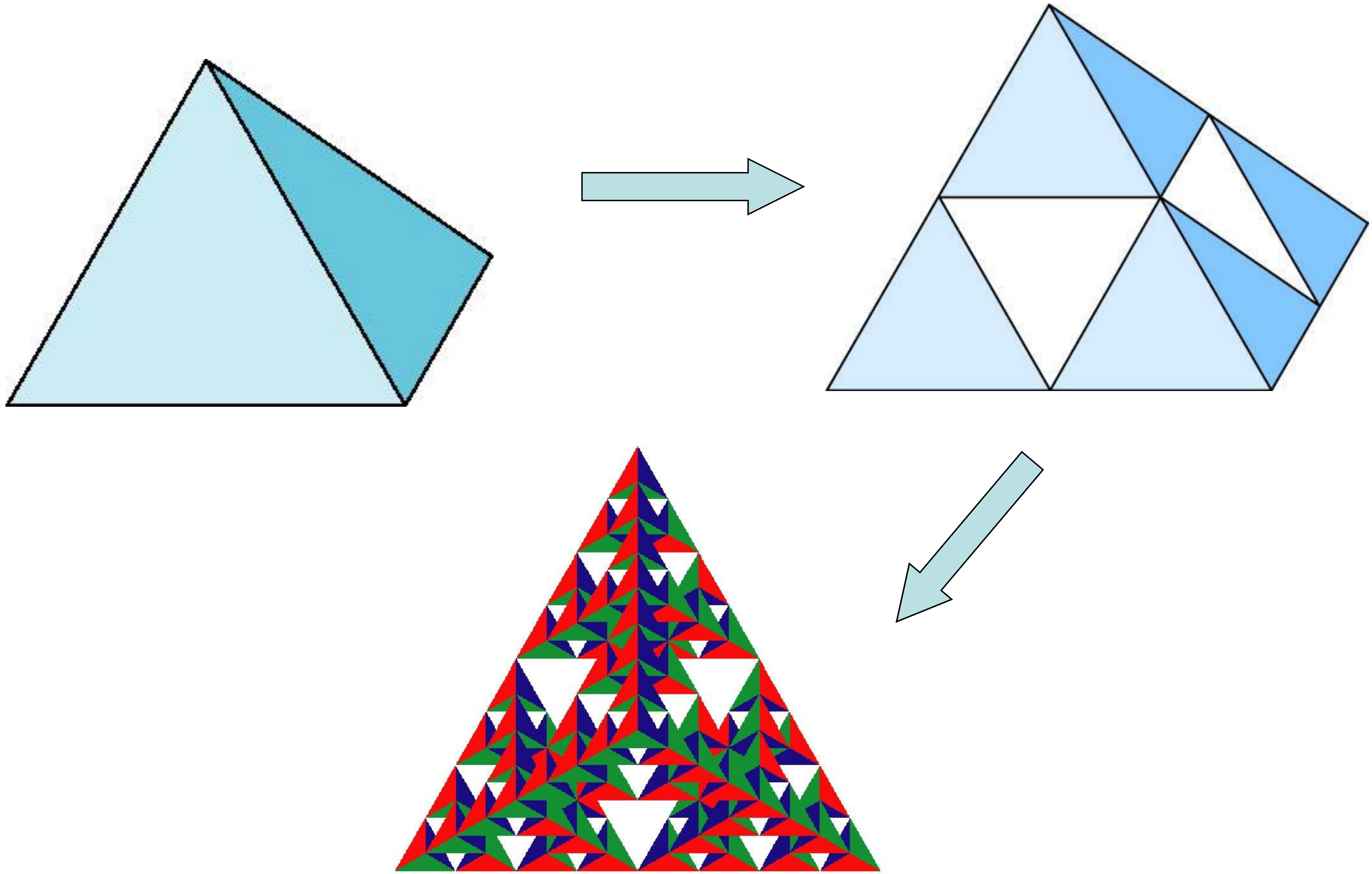
Viewports

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



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.3’ gasket3.c

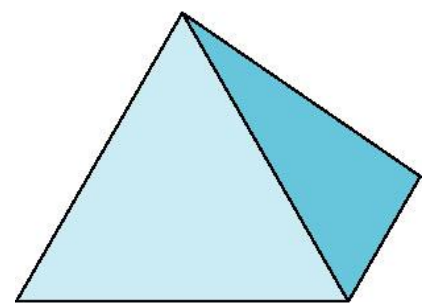


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.3’ gasket3.c (1/5)

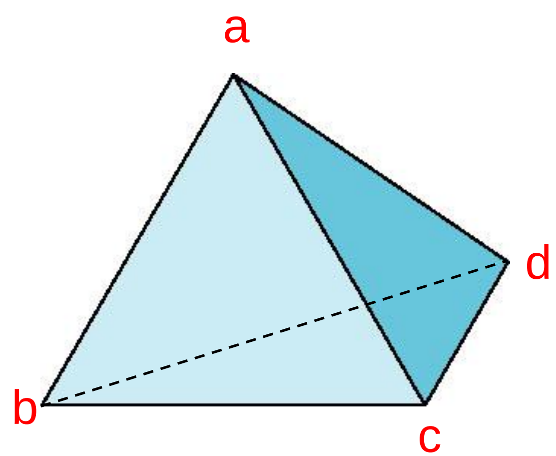
```
/* recursive subdivision of a tetrahedron to form 3D Sierpinski gasket */
/* number of recursive steps given on command line */

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

/* initial tetrahedron */

GLfloat v[4][3]={{0.0, 0.0, 1.0}, {0.0, 0.942809, -0.33333},
    {-0.816497, -0.471405, -0.333333}, {0.816497, -0.471405, -0.333333}};

GLfloat colors[4][3] = {{1.0, 0.0, 0.0}, {0.0, 1.0, 0.0},
                        {0.0, 0.0, 1.0}, {0.0, 0.0, 0.0}};

int n;

void triangle(GLfloat *va, GLfloat *vb, GLfloat *vc)
{
      glVertex3fv(va);
      glVertex3fv(vb);
      glVertex3fv(vc);
}
```

$$\begin{cases} (0.0,\ 0.0,\ 1.0) \\ (0.0, 2\sqrt{2}/3,\ -1/3) \\ (-\sqrt{6}/3, -\sqrt{2}/3,\ -1/3) \\ (\sqrt{6}/3, -\sqrt{2}/3,\ -1/3) \end{cases}$$



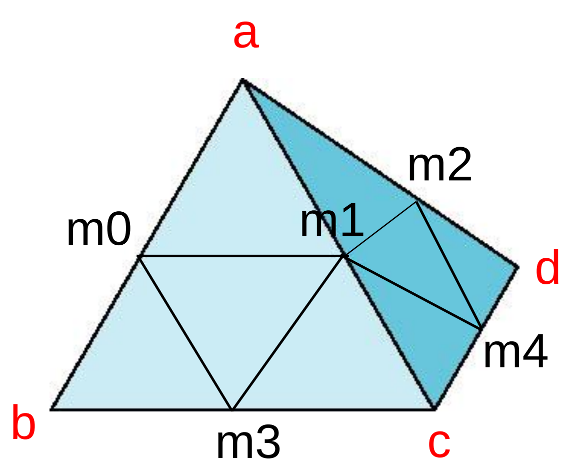
A.3' gasket3.c (2/5)

```
void tetra(GLfloat *a, GLfloat *b, GLfloat *c, GLfloat *d)
{
   glColor3fv(colors[0]);
   triangle(a, b, c);
   glColor3fv(colors[1]);
   triangle(a, c, d);
   glColor3fv(colors[2]);
   triangle(a, d, b);
   glColor3fv(colors[3]);
   triangle(b, d, c);
}
```



A.3' gasket3.c (3/5)

```
void divide_tetra(GLfloat *a, GLfloat *b, GLfloat *c, GLfloat *d, int m)
{

    GLfloat mid[6][3];
    int j;
    if(m>0)
    {
        /* compute six midpoints */

        for(j=0; j<3; j++) mid[0][j]=(a[j]+b[j])/2;
        for(j=0; j<3; j++) mid[1][j]=(a[j]+c[j])/2;
        for(j=0; j<3; j++) mid[2][j]=(a[j]+d[j])/2;
        for(j=0; j<3; j++) mid[3][j]=(b[j]+c[j])/2;
        for(j=0; j<3; j++) mid[4][j]=(c[j]+d[j])/2;
        for(j=0; j<3; j++) mid[5][j]=(b[j]+d[j])/2;

        /* create 4 tetrahedrons by subdivision */

        divide_tetra(a, mid[0], mid[1], mid[2], m-1);
        divide_tetra(mid[0], b, mid[3], mid[5], m-1);
        divide_tetra(mid[1], mid[3], c, mid[4], m-1);
        divide_tetra(mid[2], mid[4], d, mid[5], m-1);

    }
    else(tetra(a,b,c,d)); /* draw tetrahedron at end of recursion */
}
```



Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.3' gasket3.c (4/5)

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glBegin(GL_TRIANGLES);
   divide_tetra(v[0], v[1], v[2], v[3], n);
   glEnd();
   glFlush();
}


void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
         2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
   else
      glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
         2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glutPostRedisplay();
}
```


Angel: Interactive Computer Graphics 5E © Addison-Wesley 2009

A.3’ gasket3.c (5/5)

```
int main(int argc, char **argv)
{
   n=atoi(argv[1]); /* or enter number of subdivision steps here */
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("3D Gasket");
   glutReshapeFunc(myReshape);
   glutDisplayFunc(display);
   glEnable(GL_DEPTH_TEST);
   glClearColor (1.0, 1.0, 1.0, 1.0);
   glutMainLoop();
}
```