

CS3241

Computer Graphics

Semester 1, 2019/2020

Lecture 2

Elementary OpenGL Programming

School of Computing
National University of Singapore

Programming with OpenGL

Part 1: Background

Outline

- 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 ISO 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 IRIS GL

- Silicon Graphics Inc. (SGI) revolutionized the graphics workstation by implementing the raster graphics rendering pipeline in hardware (1982)
- To access the system, application programmers used a library called IRIS GL
- With IRIS GL, it was relatively simple to program three dimensional interactive applications

OpenGL

- The success of IRIS GL led 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

OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
 - Original members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM, etc.
- Relatively stable (present version 4.x)
 - Evolution often reflects new hardware capabilities
 - 3D texture mapping and texture objects
 - Vertex programs
- Allows for platform specific features through **extensions**
- OpenGL ARB is now part of **The Khronos Group**

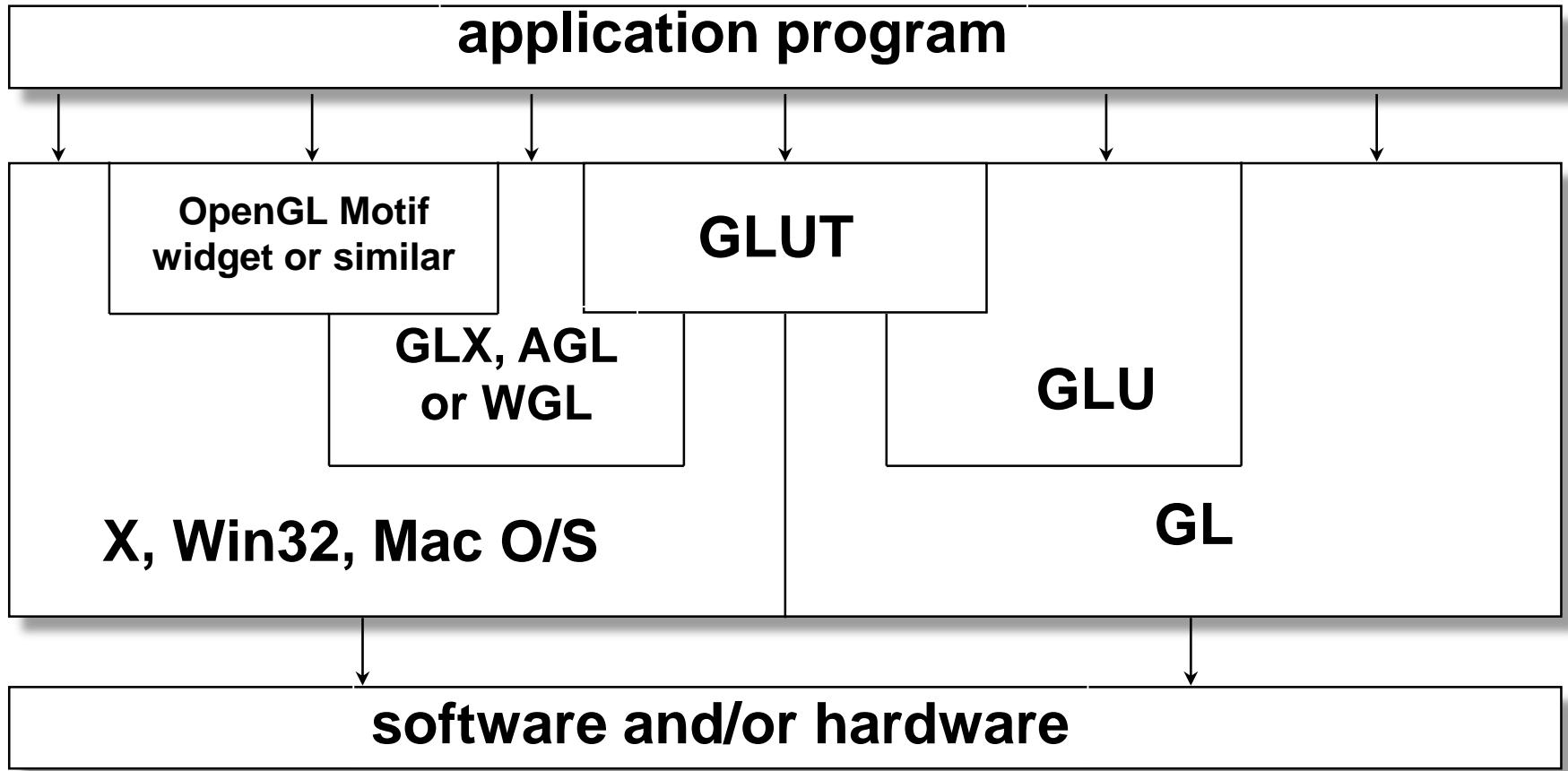
OpenGL Libraries

- 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 / FreeGLUT Libraries

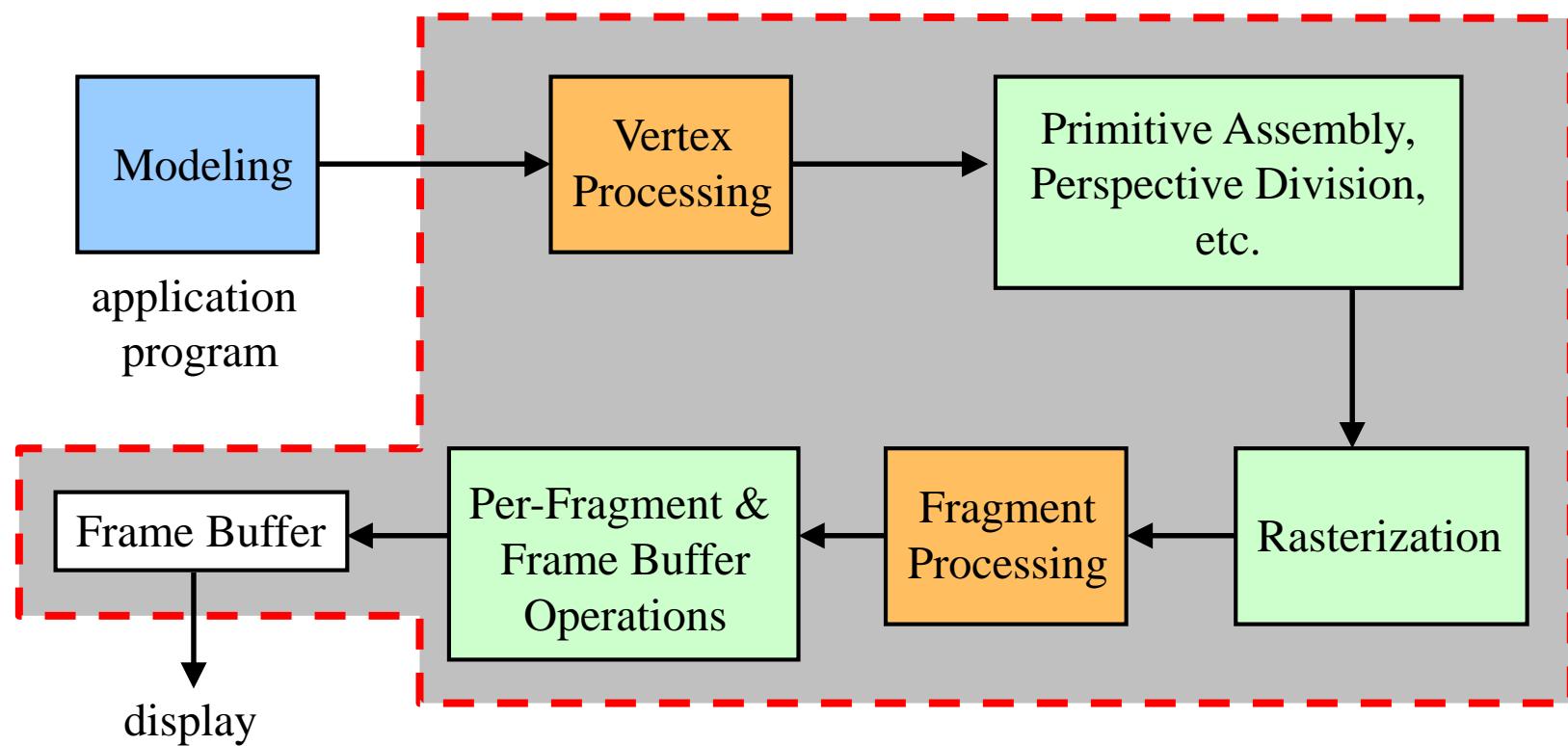
- GLUT = OpenGL Utility Toolkit
 - Not part of OpenGL
 - Provides functionality common to all window systems
 - Open a window
 - Get input from mouse and keyboard
 - Menus
 - Event-driven
 - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
 - No slide bars
- FreeGLUT: <http://freeglut.sourceforge.net/>

Software Organization



Basic OpenGL 3D Rendering Pipeline

- To render a primitive using OpenGL, the primitive goes through the following main stages
 - Turning primitive into pixels



OpenGL Functions

- Primitives

- Points
- Line Segments
- Polygons

- Attributes

- Transformations

- Viewing
- Modeling

- Control (GLUT)

- Input (GLUT)

- Query

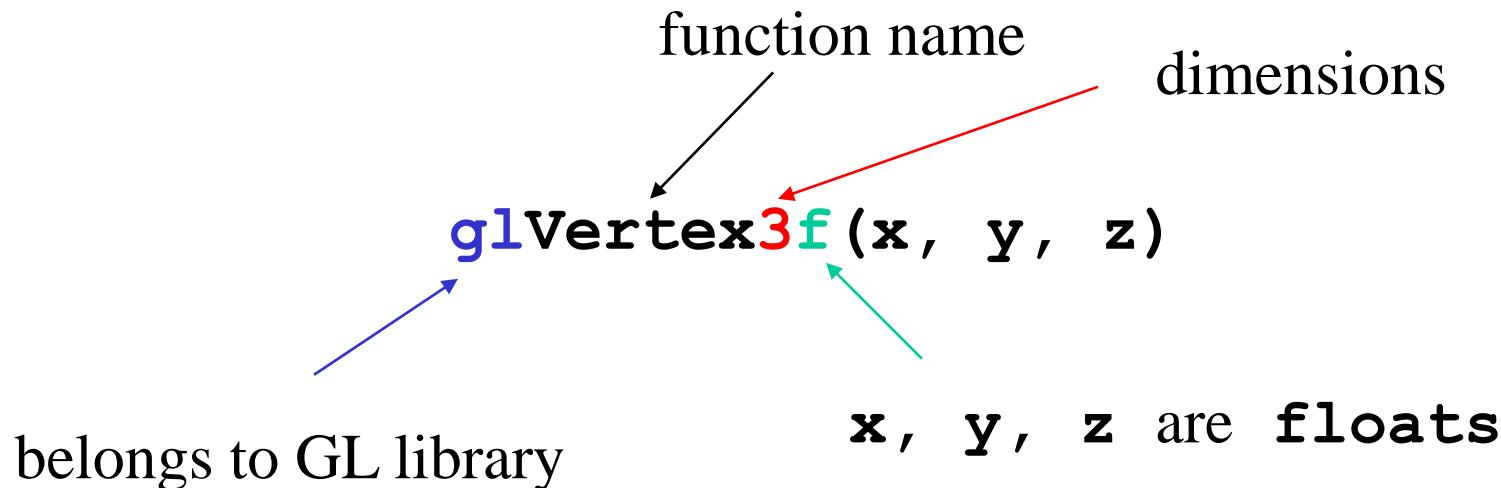
OpenGL State

- 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

Lack OOP Support

- OpenGL is not object-oriented so that 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



The diagram illustrates the components of an OpenGL function call with a pointer parameter:

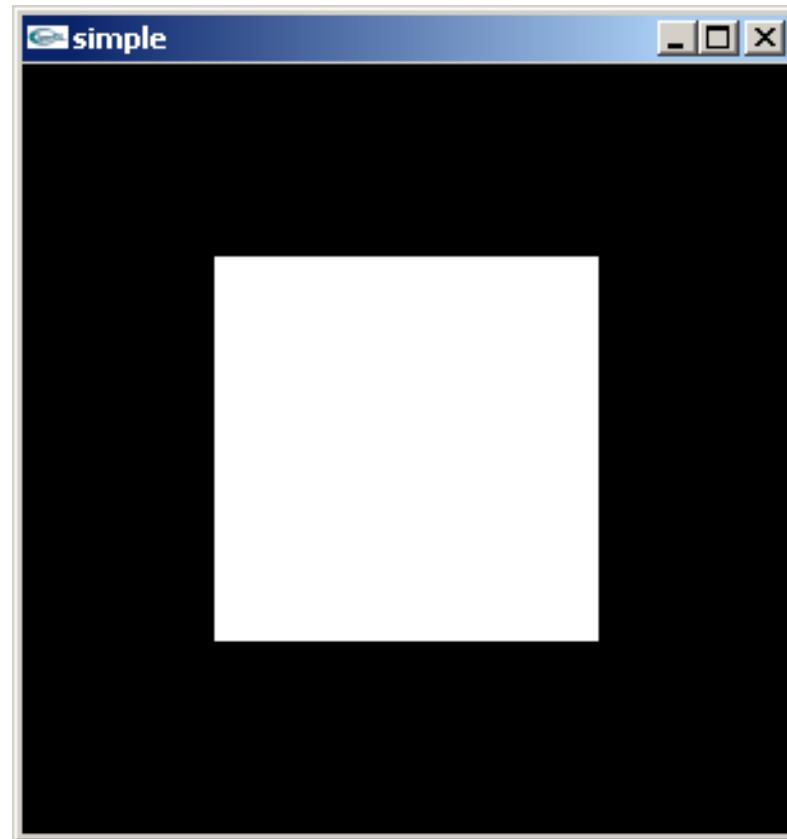
- function name:** `glVertex3fv` (highlighted in black)
- parameters:** `(p)` (highlighted in green)
- p is a pointer to an array:** A green arrow points from this text to the parameter `p`.

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)`
- The include files also define OpenGL data types: `GLfloat`, `GLdouble`,

A Simple Program

Generate a square on a solid background



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();
}
```

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

Compilation on Windows

■ Visual C++

- Get glut.h, glut32.lib and glut32.dll from web
- Create a console application
- Add opengl32.lib, glu32.lib, glut32.lib to project settings
(under link tab)

■ Cygwin (Linux under Windows)

- Can use gcc and similar makefile to Linux
- Use -lopengl32 -lglu32 -lglut32 flags

Programming with OpenGL

Part 2: Complete Programs

Outline

- Refine the first program
 - Alter the default values
 - Introduce a standard program structure
- Simple viewing
 - 2D viewing as a special case of 3D viewing
- Fundamental OpenGL primitives
- Attributes

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 callback function
 - Input and window functions

simple.c Revisited

- In the second version, we shall see the same output
- But we will define all the relevant state values explicitly through function calls using the default values
- In particular, we set
 - Colors
 - Viewing conditions
 - Window properties

main()

```
#include <GL/glut.h>           ← includes gl.h

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);   ← define window properties
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);   ← set display callback

    init();                      ← set OpenGL state

    glutMainLoop();              ← 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
- **glutWindowSize** — in pixels
- **glutWindowPosition** — from top-left corner of display
- **glutCreateWindow** — create window with title “simple”
- **glutDisplayFunc** — display callback
- **glutMainLoop** — enter infinite event loop

init()

```
void init()
```

```
{
```

```
    glClearColor(0.0, 0.0, 0.0, 1.0);
```

black clear color

opaque window

```
    glColor3f(1.0, 1.0, 1.0);
```

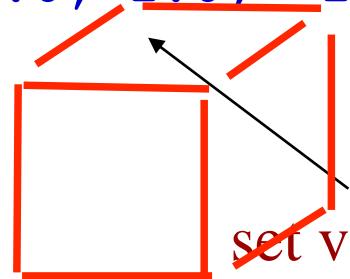
set fill/draw
color to white

```
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
```

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

```
}
```



set viewing volume

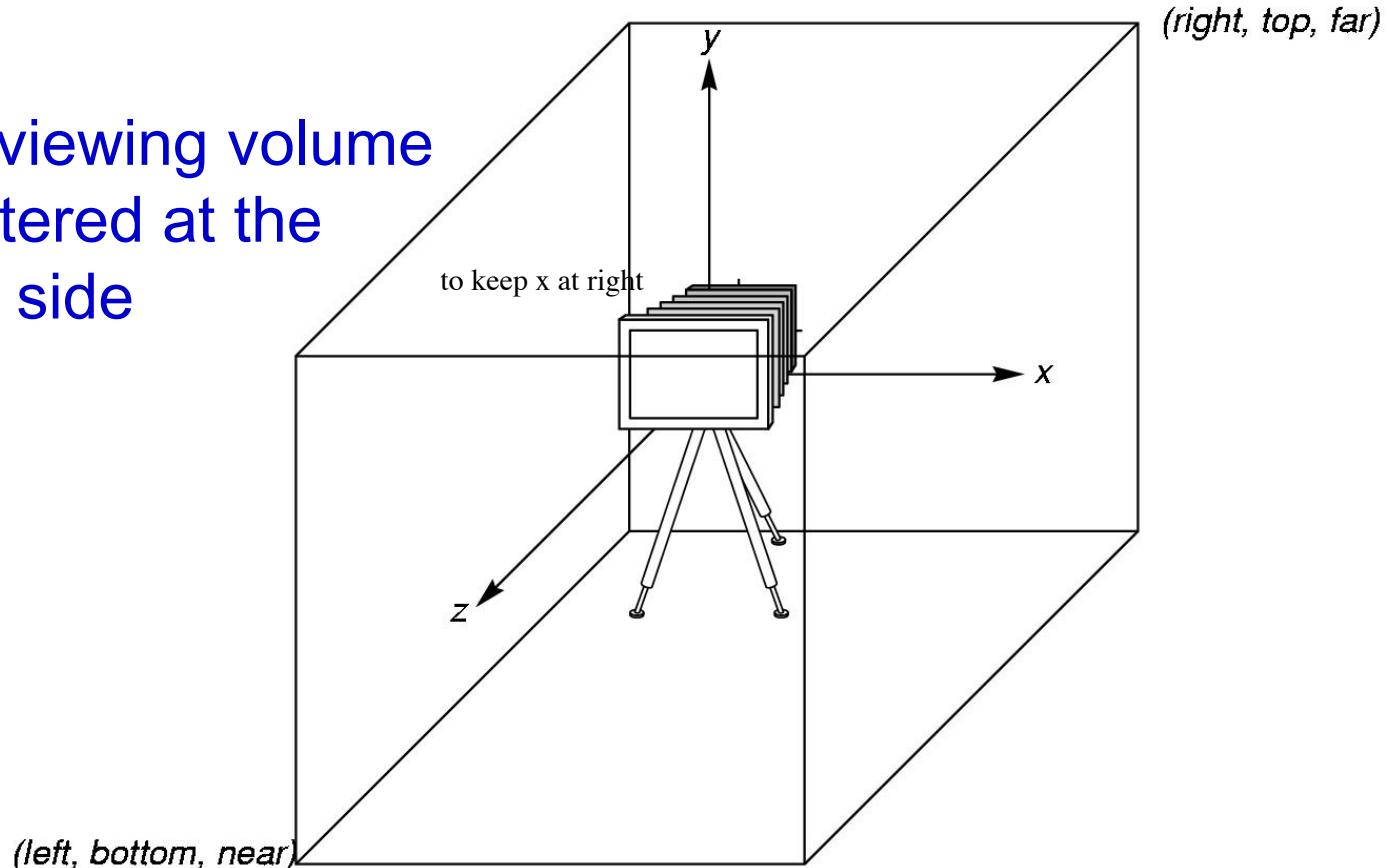
Coordinate Systems

- The units in `glVertex` are determined by the application and are called **object coordinates**
 - In this case, the **object coordinates** are the same as the **world coordinates**
- The **camera** is positioned in the **world coordinates**
- The viewing specifications are specified in **camera (eye) coordinates**
 - The size of **viewing volume** determines what will appear in image
- Internally, OpenGL will convert vertices to **camera (eye) coordinates** and later to **window coordinates**

OpenGL Camera

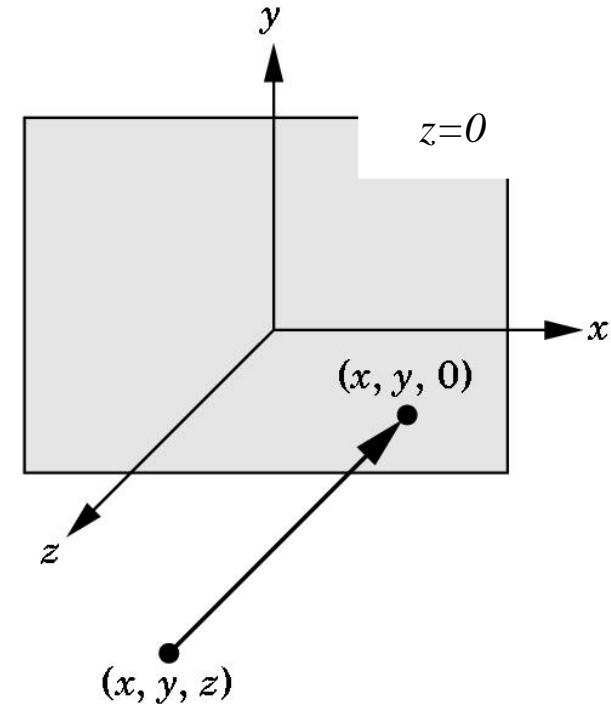
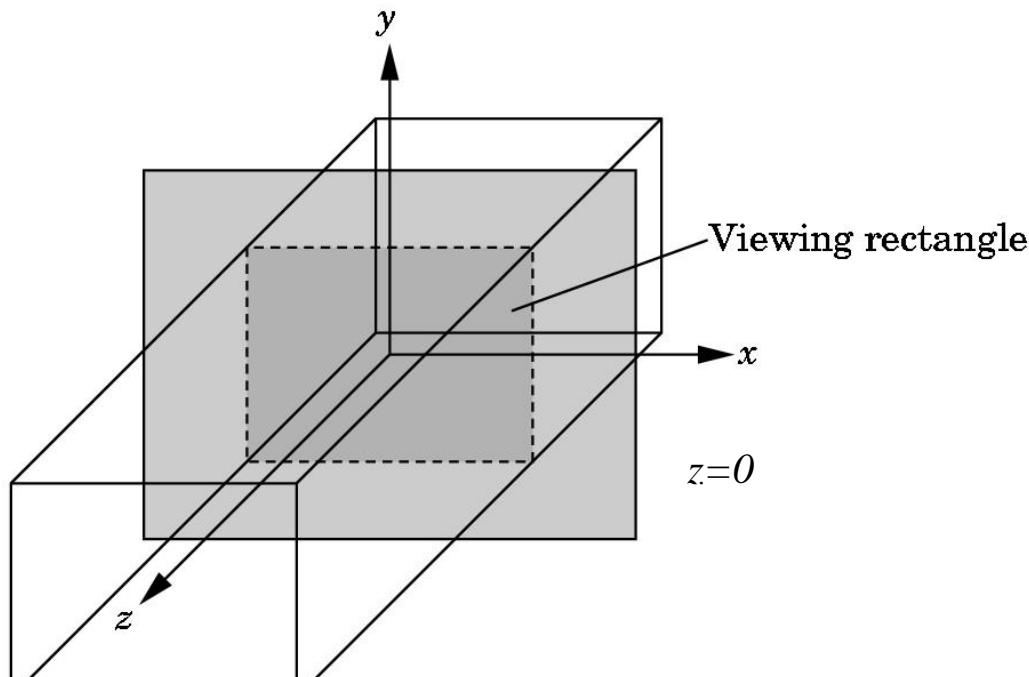
- By default, OpenGL places a camera at the origin in world space, looking in the negative z direction

- The default viewing volume is a box centered at the origin with a side of length 2



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, 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);
```

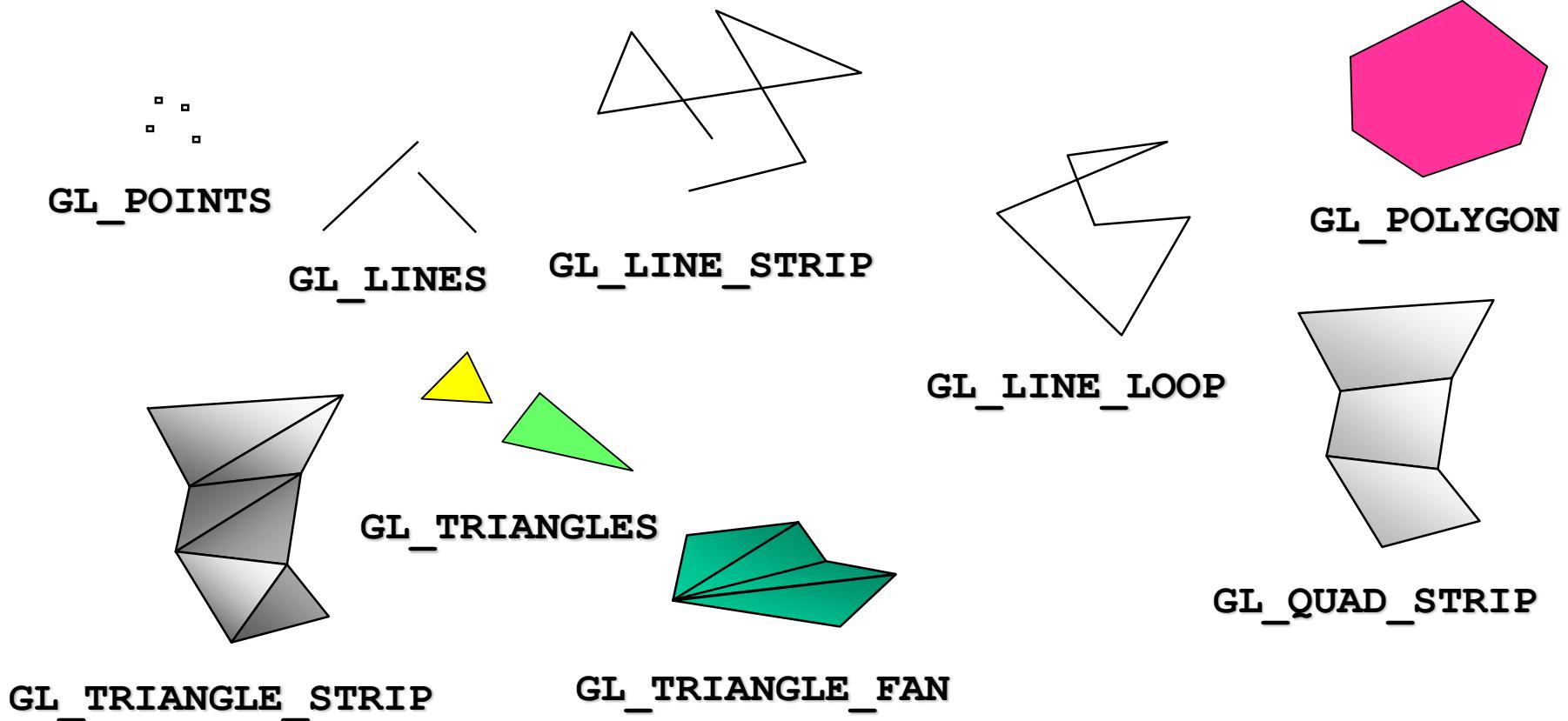
2D and 3D Viewing

- In `glOrtho(left, right, bottom, top, near, far)`
 - The **near** and **far** distances are measured, in the $-z$ direction, from the $z = 0$ plane of the camera frame
- Two-dimensional vertex commands place all vertices in the plane $z = 0$
- If the application is in 2D, we can use the function
 - `gluOrtho2D(left, right, bottom, top)`
- In 2D, the view or clipping volume becomes a **clipping window**

mydisplay()

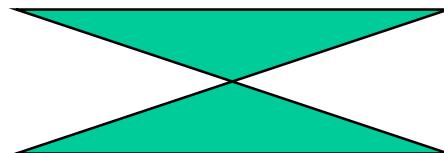
```
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    glFlush();
}
```

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 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



non-simple polygon



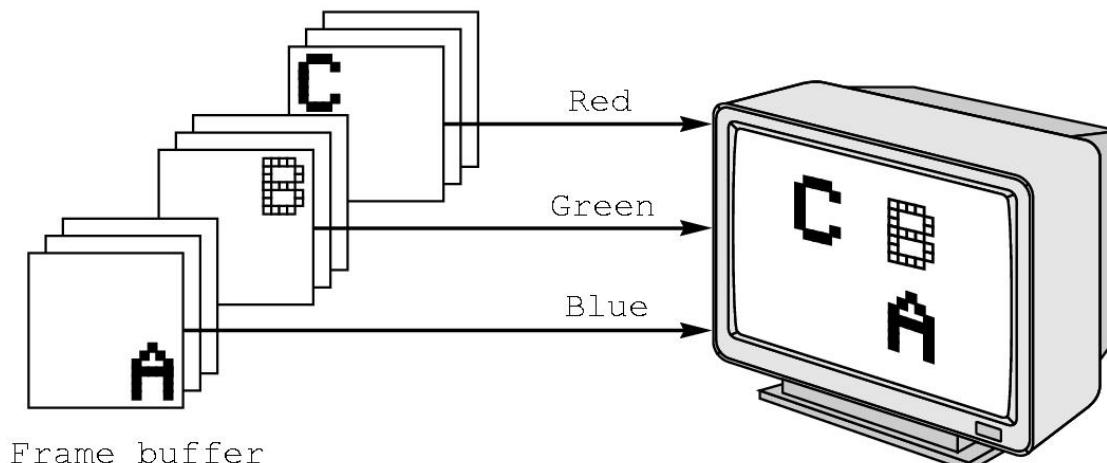
non-convex polygon

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

RGB Color

- Each color component/channel is stored separately in the frame buffer
- Usually 8 bits per component/channel 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



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

`glColor...`

`glVertex...`

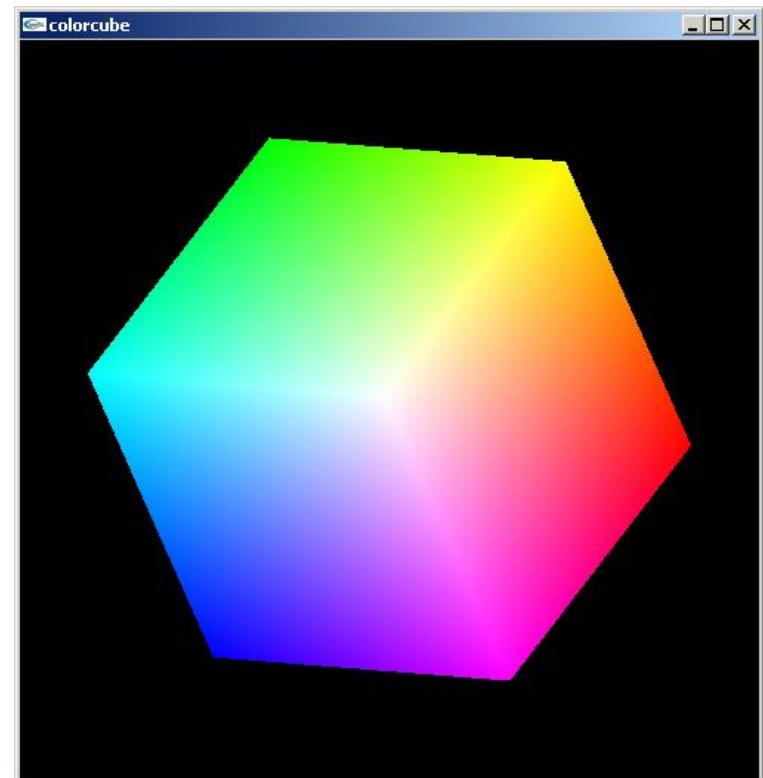
`glColor...`

`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
glShadeModel (GL_FLAT)

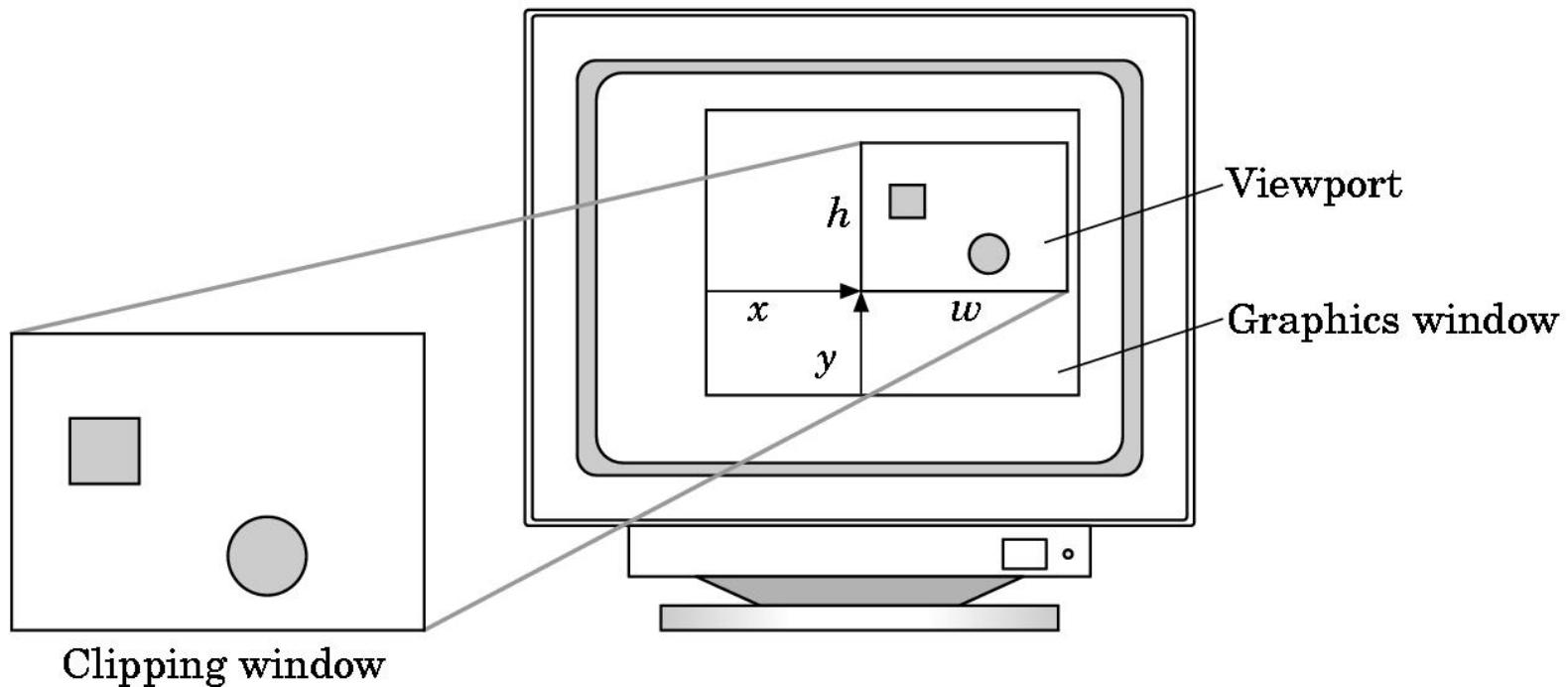


Viewports

- Do not have to use the entire window for the image:

glViewport(x, y, w, h)

- Values in pixels (window coordinates)



Programming with OpenGL

Part 3: Three Dimensions

Outline

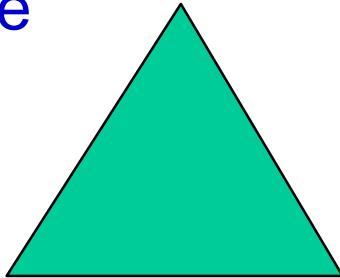
- Develop a more sophisticated three-dimensional example
 - Sierpinski gasket: a **fractal**
- Introduce **hidden-surface removal**

Three-Dimensional Applications

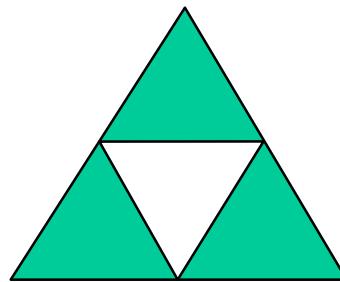
- In OpenGL, 2D applications are a special case of 3D graphics
- Going to 3D
 - Not much changes
 - Use `glVertex3*` ()
 - Have to worry about the order in which polygons are drawn or use hidden-surface removal
 - Polygons should be simple, convex, flat

Sierpinski Gasket (2D)

- Start with a triangle



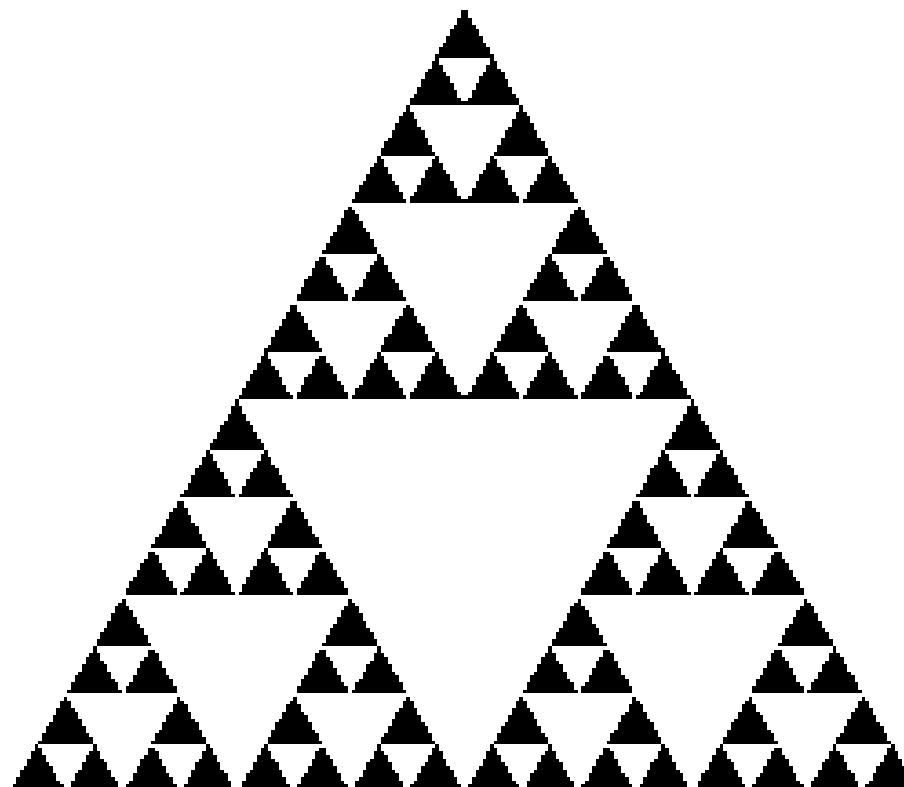
- Connect bisectors of sides and remove central triangle



- Repeat

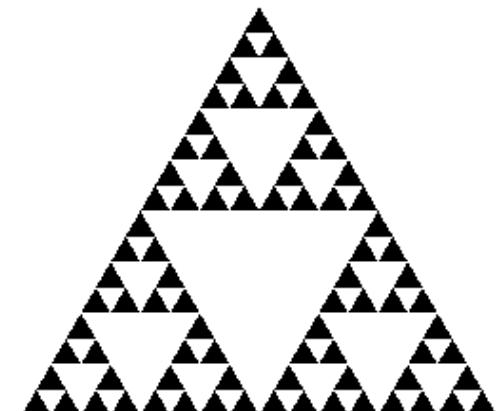
Example

- 4 levels of subdivisions



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 (< 2D)
 - but the perimeter goes to infinity (> 1D)
- This is not an ordinary geometric object
 - It is neither one- nor two-dimensional
- It is a **fractal** (fractional dimension) object
 - Approximately 1.585 D



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 */
```

Draw One Triangle

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

Triangle Subdivision

```
void divide_triangle(GLfloat *a, GLfloat *b, GLfloat *c, int m)
{
    /* triangle subdivision using vertex numbers */
    GLfloat v0[2], v1[2], v2[2];

    if (m > 0)
    {
        for (int j = 0; j < 2; j++) v0[j] = (a[j]+b[j])/2;
        for (int j = 0; j < 2; j++) v1[j] = (a[j]+c[j])/2;
        for (int 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 myinit()

```
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);
}
```

main()

```
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

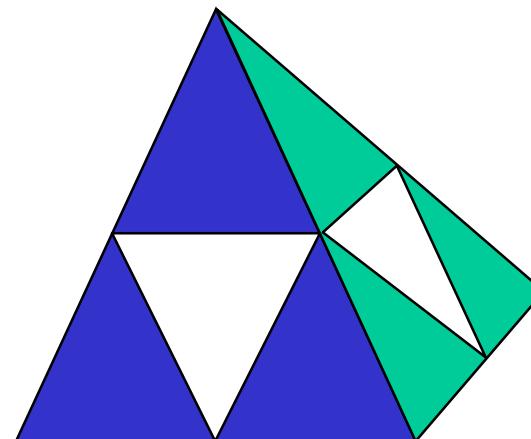
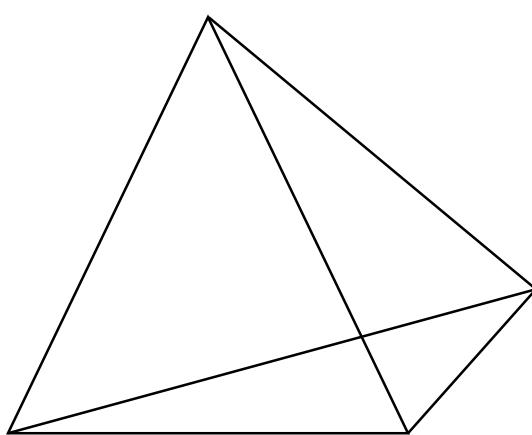
- By having the `glBegin` and `glEnd` in the display callback rather than in the function `triangle` and using `GL_TRIANGLES` rather than `GL_POLYGON` in `glBegin`, we call `glBegin` and `glEnd` only once for the entire gasket rather than once for each triangle

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**

3D Gasket

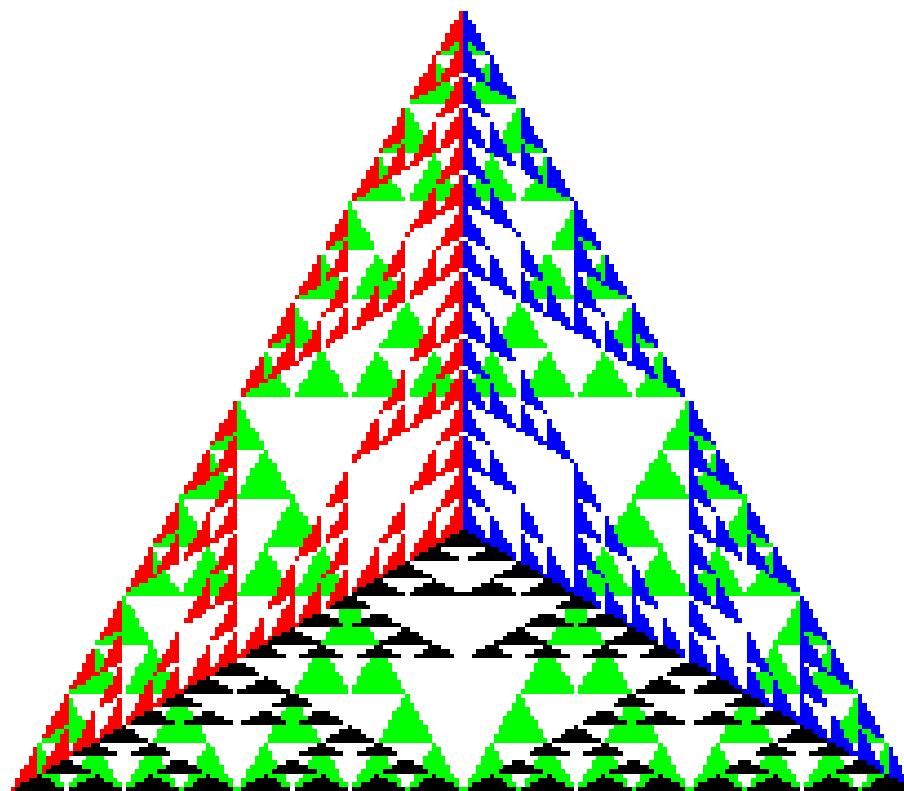
- We can subdivide each of the four faces



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

Example

After 4 iterations



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 v0[3], v1[3], v2[3];

    if (m > 0)
    {
        for (int j = 0; j < 3; j++) v0[j]=(a[j]+b[j])/2;
        for (int j = 0; j < 3; j++) v1[j]=(a[j]+c[j])/2;
        for (int 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);
}
```

tetrahedron Code

```
void tetrahedron(int m)
{
    glColor3f(1.0, 0.0, 0.0); // red
    divide_triangle(v[0], v[1], v[2], m);

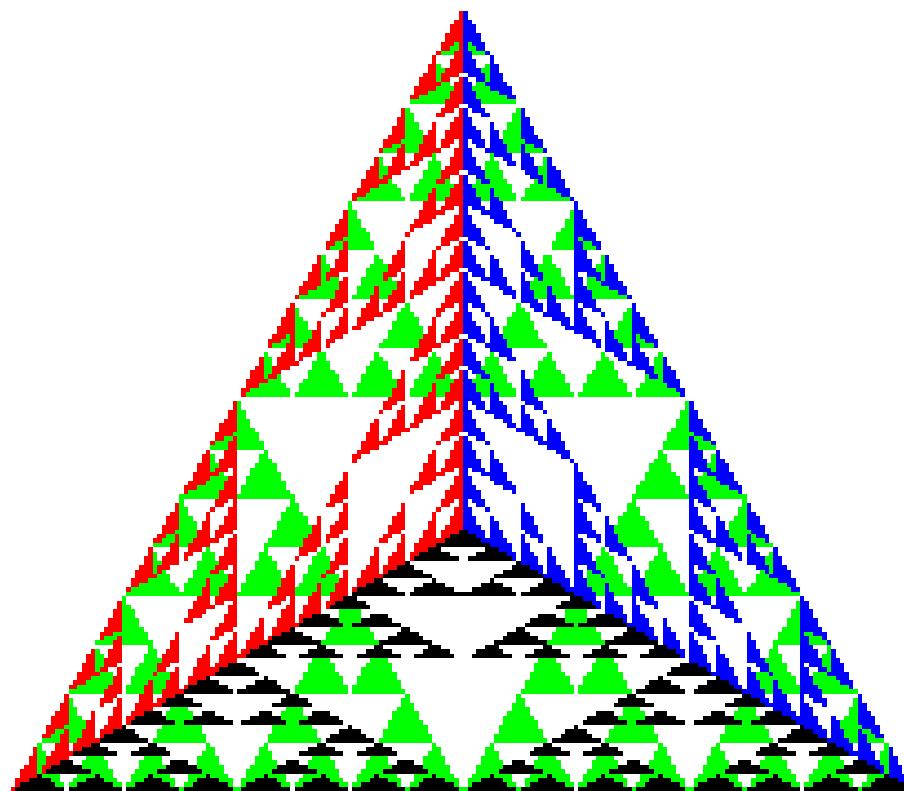
    glColor3f(0.0, 1.0, 0.0); // green
    divide_triangle(v[3], v[2], v[1], m);

    glColor3f(0.0, 0.0, 1.0); // blue
    divide_triangle(v[0], v[3], v[1], m);

    glColor3f(0.0, 0.0, 0.0); // black
    divide_triangle(v[0], v[2], v[3], m);
}
```

Example

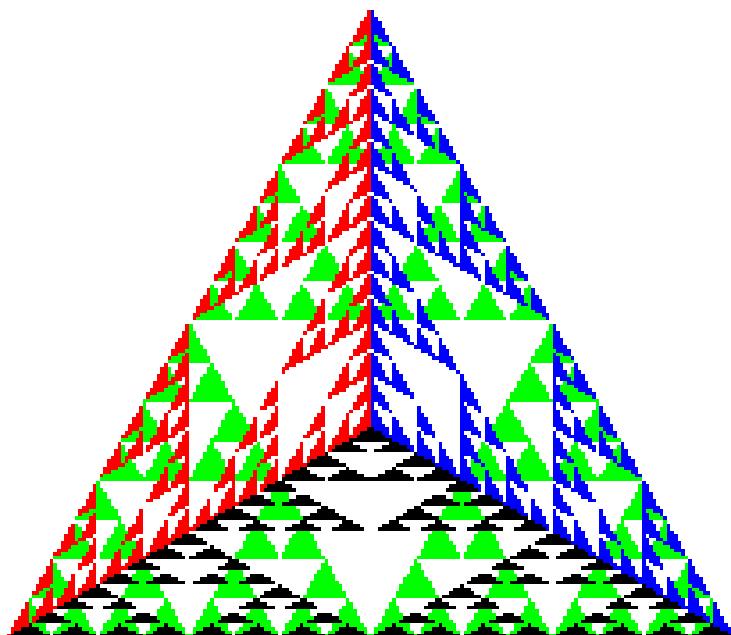
After 4 iterations



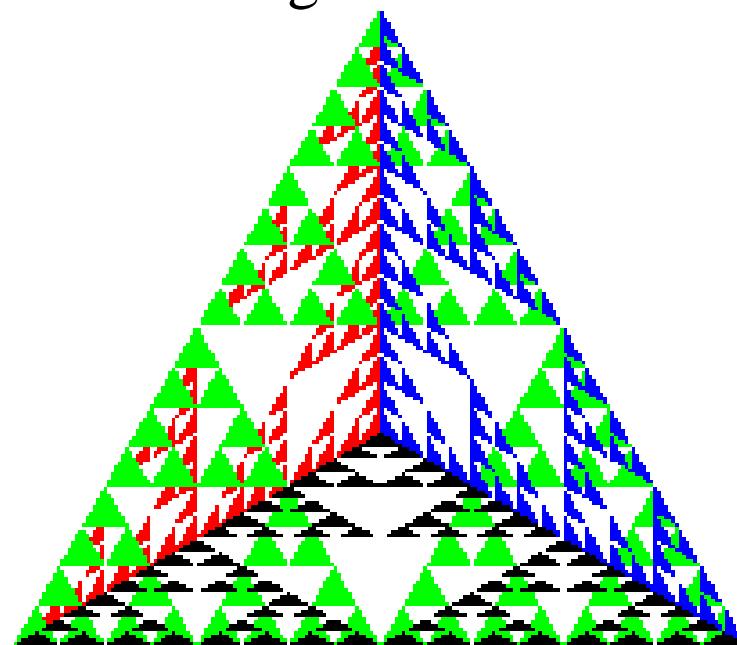
Almost Correct

- 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

want this

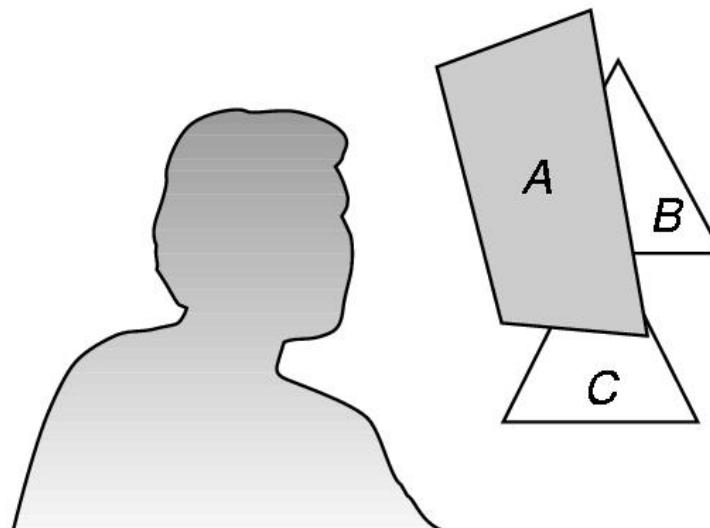


get this



Hidden-Surface Removal

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



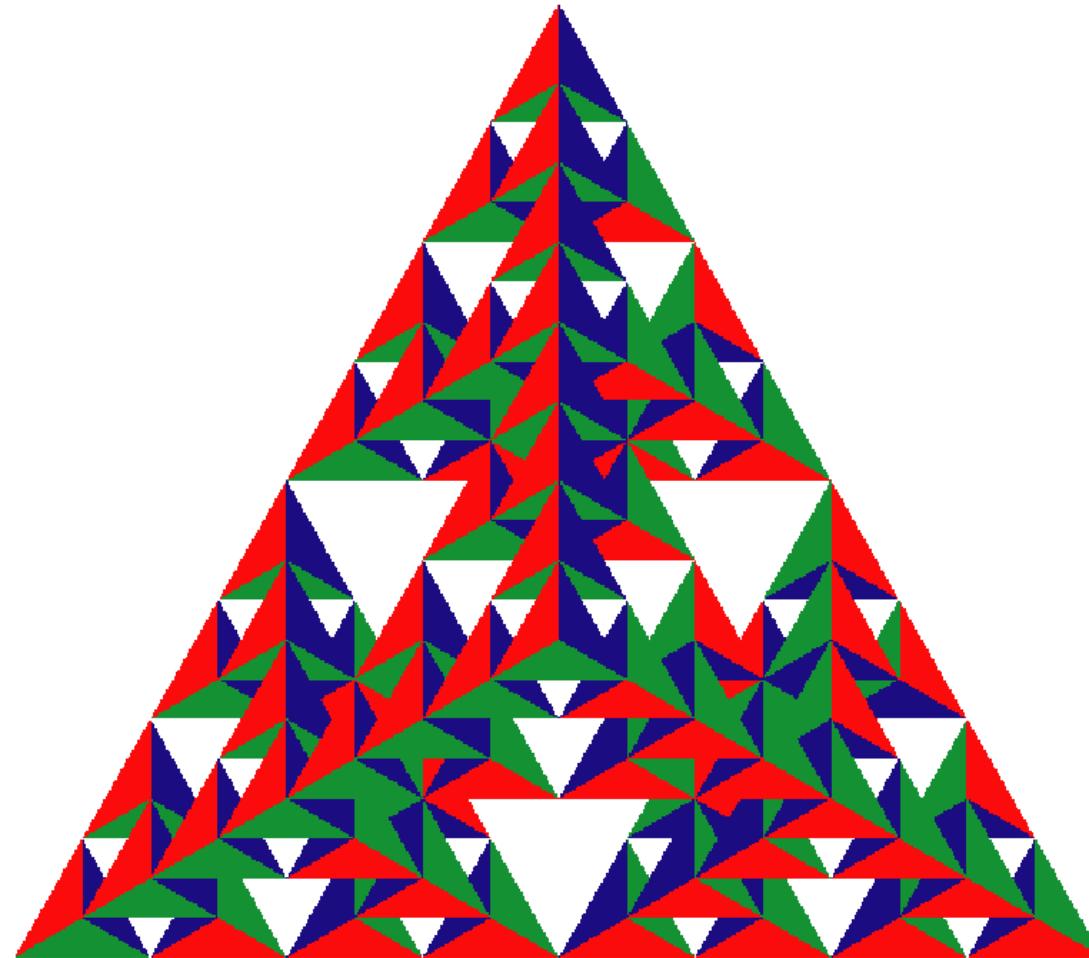
Using the z-buffer Algorithm

- The algorithm uses an extra buffer, the **z-buffer**, to store **depth information** as geometry travels down the pipeline
- It is
 - Requested in `main()`
 - `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)`
 - Enabled in `init()`
 - `glEnable(GL_DEPTH_TEST)`
 - Cleared in the display callback
 - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`

Surface vs. Volume Subdivision

- In our example, we subdivided the **surface** of each face
- We could also subdivide 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
- Good programming exercise

Volume Subdivision



End of Lecture 2