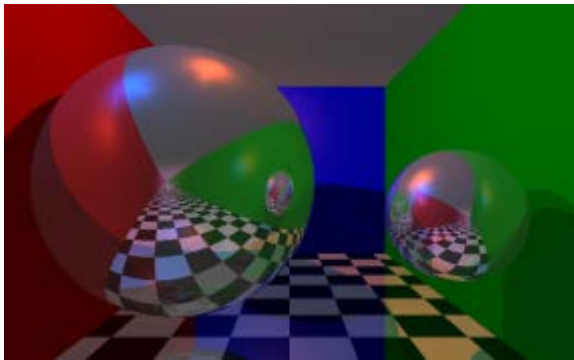
CSI 4105 Computer Graphics

Spring 2017

Lecture 2:
Programming with OpenGL (Part I)

Seon Joo Kim
Yonsei University



Objectives of this lecture

- Discuss the “anatomy” of an OpenGL application
- Discuss the OpenGL Architecture
  - OpenGL as a “state machine”
- OpenGL Functions and API
  - Types
  - Formats
- Example of a Simple program

Note: To get started we will introduce some concepts (such as the projection matrix) that we will talk about in more detail later.

This and Next lecture

- We will rely heavily on “GLUT”, an extension API built on top of OpenGL
- In **this lecture** we will focus on simple displaying only (i.e. not interactive!)
- In the **next lecture** we will talk about “input” and “user events”
- Good resources for both lectures:

https://users.cs.jmu.edu/bernstdh/web/common/lectures/slides_glut-basics.php

http://www.opengl.org/documentation/specs/glut/spec3/spec3.html

OpenGL Libraries (opengl32, glu32, glut32)

- OpenGL core library
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)
- OpenGL Utility Library (GLU)
  - Uses functions from OpenGL core to create more complex objects

GL Utility Toolkit (GLUT)

- OpenGL Utility Toolkit
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven

Download GLUT32 here – read instructions carefully on installation, esp where to copy files!:
http://www.opengl.org/resources/libraries/glut/glut_downloads.php

OpenGL Functions Types

- **Geometry Primitives**: draw primitives
  - Points
  - Line Segments
  - Polygons
- **Attributes**: change drawing attribute
- **Transformations:** change transformation
  - Viewing
  - Modeling
- **Control (**GLUT): setup application
- **Input** (GLUT): handle events
- **Query**: query attributes or window information

OpenGL State

An OpenGL implementation is often described as a state machine. That is, it has an inherent state represented by numerous internal variables, all of which remain in their default state until changed via an OpenGL call, and remain in that new state until changed again via an OpenGL call.

OpenGL will not display graphics properly, or at all, if it is not in the appropriate state, and many OpenGL programming errors are due to the programmer misunderstanding or making incorrect assumptions about the OpenGL state at some point.

The reason OpenGL operates like a state machine is to avoid having to pass many arguments on each function call.

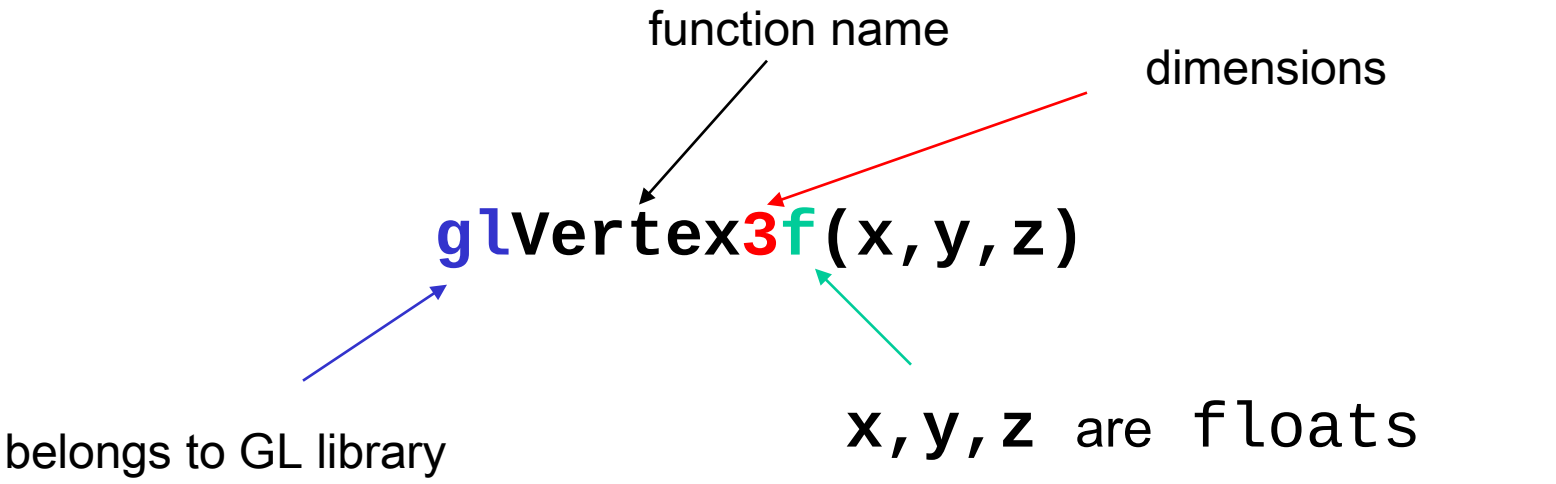
OpenGL State

- 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 of Object Orientation

- OpenGL is not C++ (object-oriented) so that there are multiple functions for a given logical function
  - **`glVertex3f`**
  - **`glVertex2i`**
  - **`glVertex3dv`**
- Could create overloaded functions in C++

OpenGL Function Format



glVertex3fv(p)

p is a pointer to an array

OpenGL Function Format Example

```
GLfloat x,y,z;
x=10.0; y=0; z=20.0;

GLfloat v[3];
v[0]=10.0; v[1]=0; z[2]=20.0;

glBegin(GL_POINTS);
      glVertex3f(x,y,z);      // both lines have the
      glVertex3fv(v);         // same result
glEnd();
```

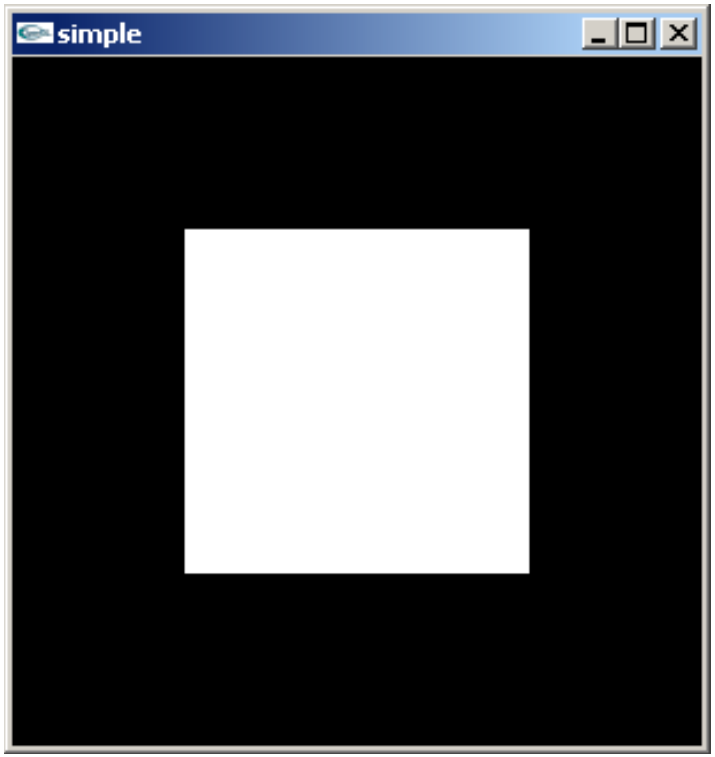
`Glfloat` is just a float. C/C++ allows "typedefs", something that Java doesn't allow. So, instead of using system float, you use the "OpenGL float". This is done to help keep OpenGL code compatible on different platforms, esp 64-bit vs. 32-bit compilers. See: http://en.wikipedia.org/wiki/Typedef

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



simple1.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();
}
```

This may look strange, the variable is the name of the function defined above!

See code “sample1.c”

Display Callback / Event Loop

- Note that the program defines a *display callback* function named **`mydisplay`**
  - Every glut program must have a display callback
  - glutDisplayFunc sets the display callback for the *current window*.
  - 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 **`glutMainLoop();`**
    - No more statements after `glutMainLoop()` will be executed, `glutMainLoop()` never terminates.

Event Loop

- glutMainLoop enters the GLUT event processing loop.
- This routine should be called at most once in a GLUT program.
- Once called, this routine will never return.
- It will call as necessary any callbacks that have been registered.

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

Programming with OpenGL
Part 2: Setting the States

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

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

See code “sample2.c”

simple2.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();
}
```

glut.h will includes **gl.h**

Not necessary unless you are using "X-widows" on Unix.

Defines app properties

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
  - RGB color
  - Single buffering (or double buffering, we will discuss later)
  - 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 func

```
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);
}
```

sets “black” to be the clear color
(it doesn’t clear the screen, only sets the color!)

opaque window
(opaque=solid)

set fill/draw
color to white
(R=1.0, B=1.0, G=1.0)

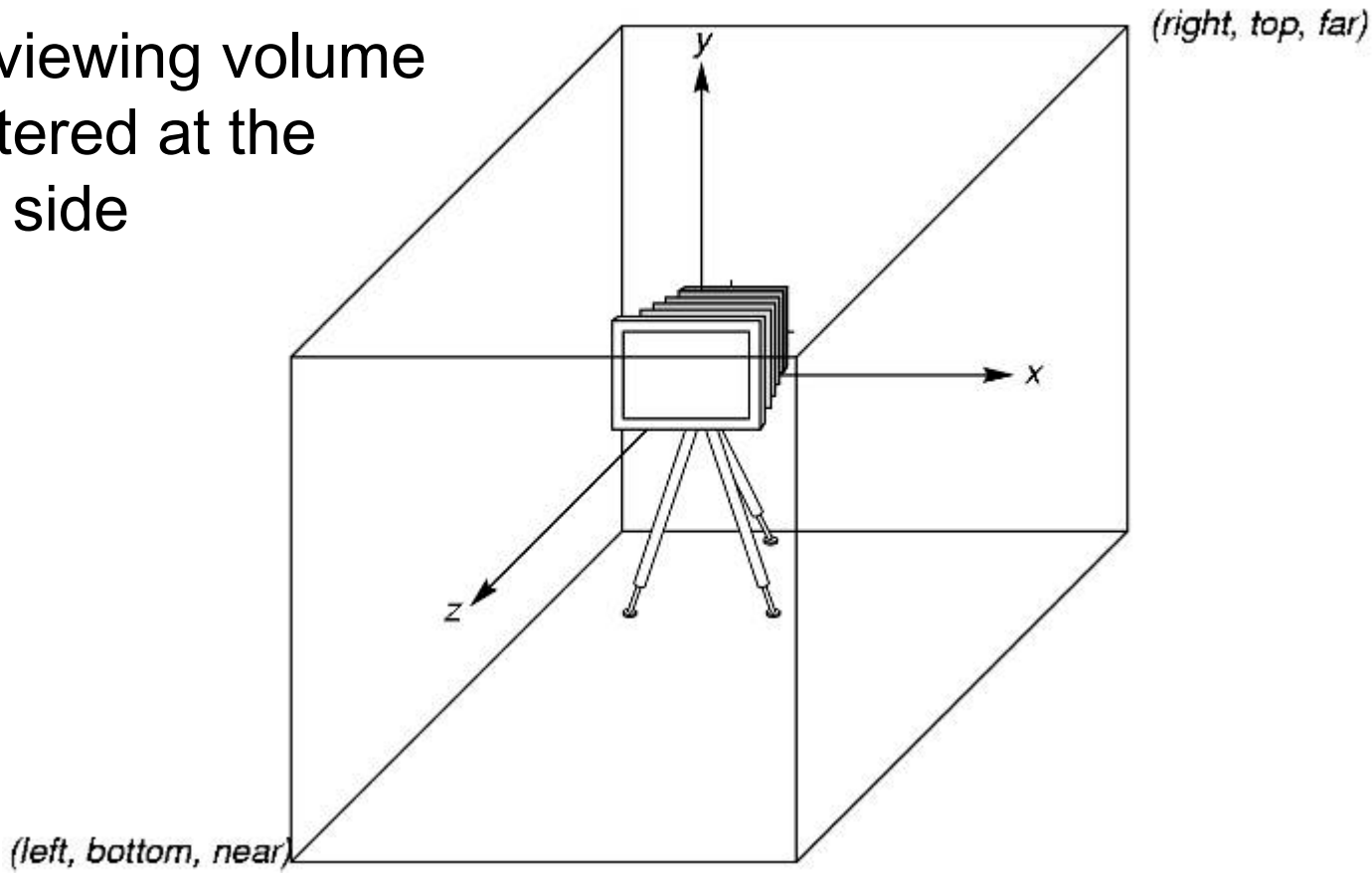
viewing volume

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

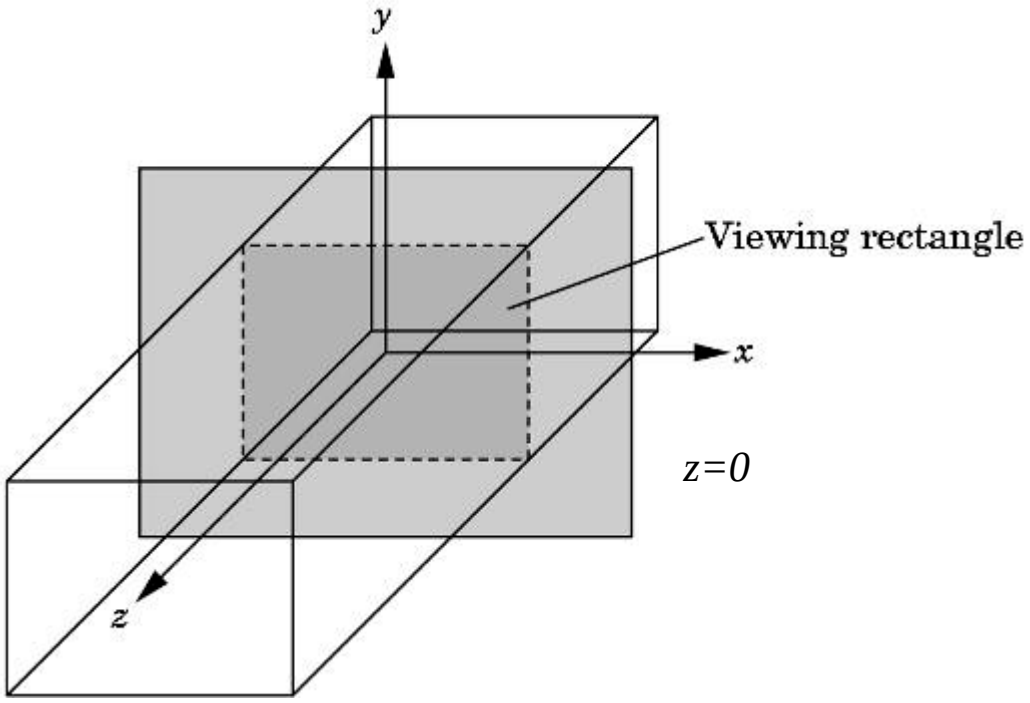
OpenGL Camera

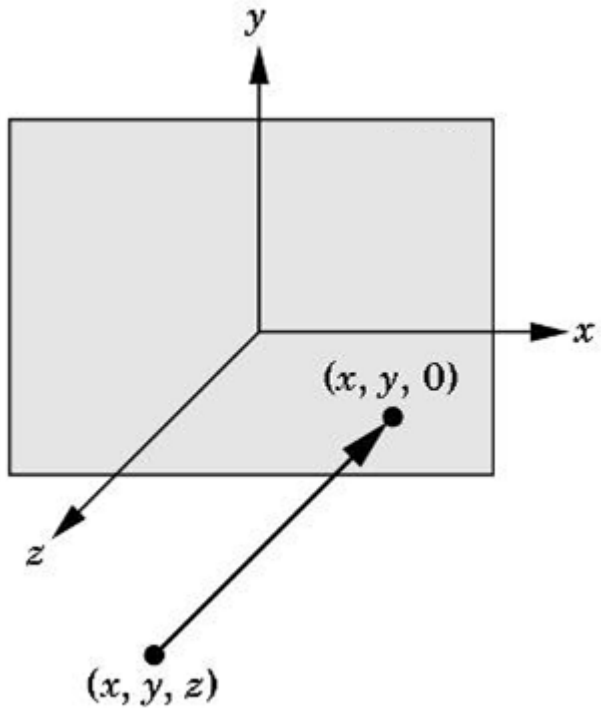
- OpenGL places a camera at the origin in object space pointing in the **negative z direction**
- The default viewing volume is a box centered at the origin with a side of length 2



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 from the camera
- Two-dimensional vertex commands place all vertices in the plane $z = 0$
- If the application is in two dimensions (i.e. all z values are 0), we can use the function

  `gluOrtho2D(left, right, bottom, top)`

- In two dimensions, the view or clipping volume becomes a *clipping window*

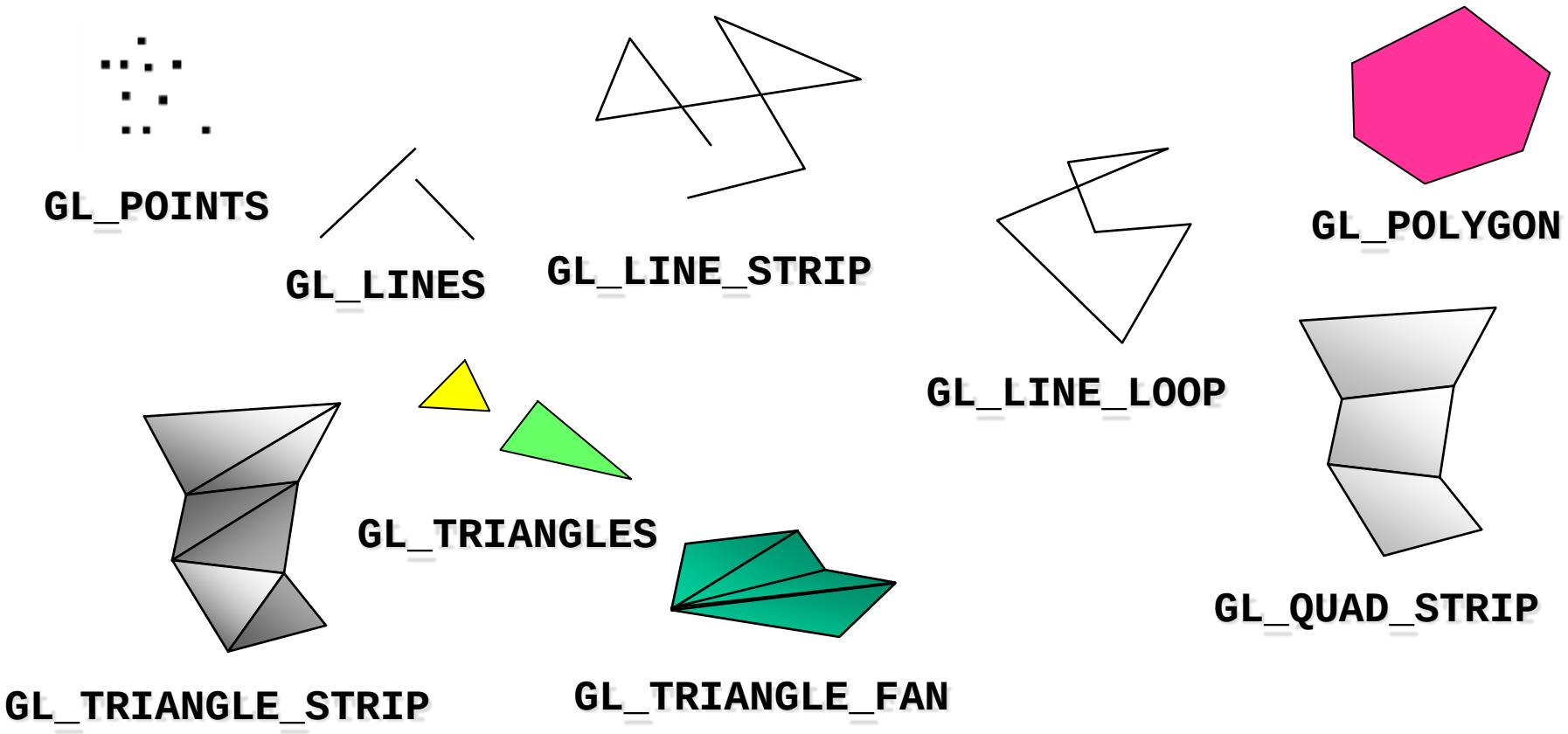
Display func (no change)

```
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();
}
```

This is the standard OpenGL command to begin geometry.

This ends the geometry.

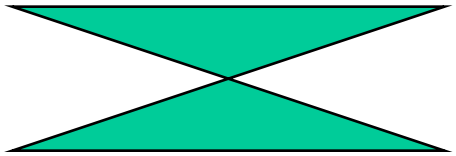
OpenGL Primitives



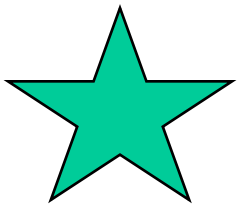
See: https://www.opengl.org/sdk/docs/man2/xhtml/glBegin.xml

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 the above true
  - OpenGL **will** produce output if these conditions are violated but it may not be what is desired!
- Triangles satisfy all conditions . . triangles are preferred!



non-simple polygon



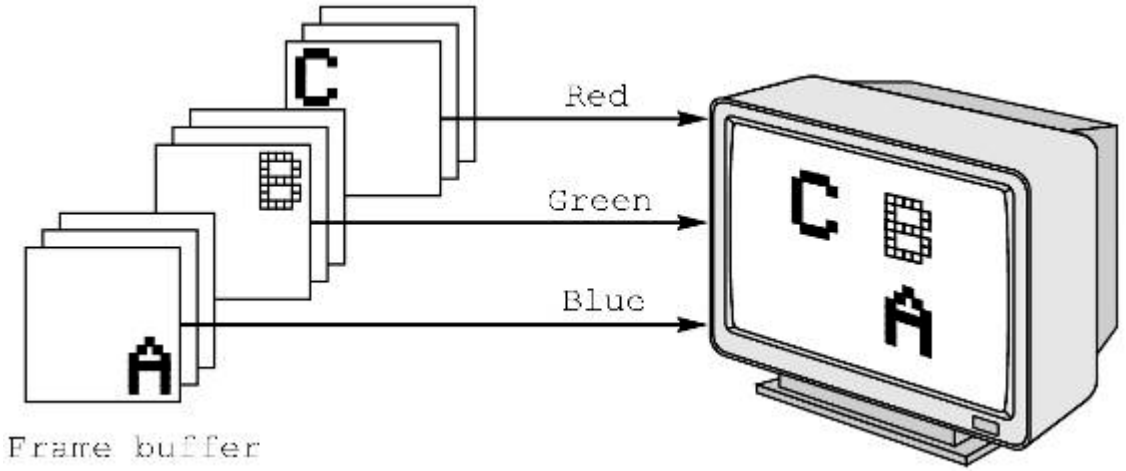
non-convex polygon

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 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



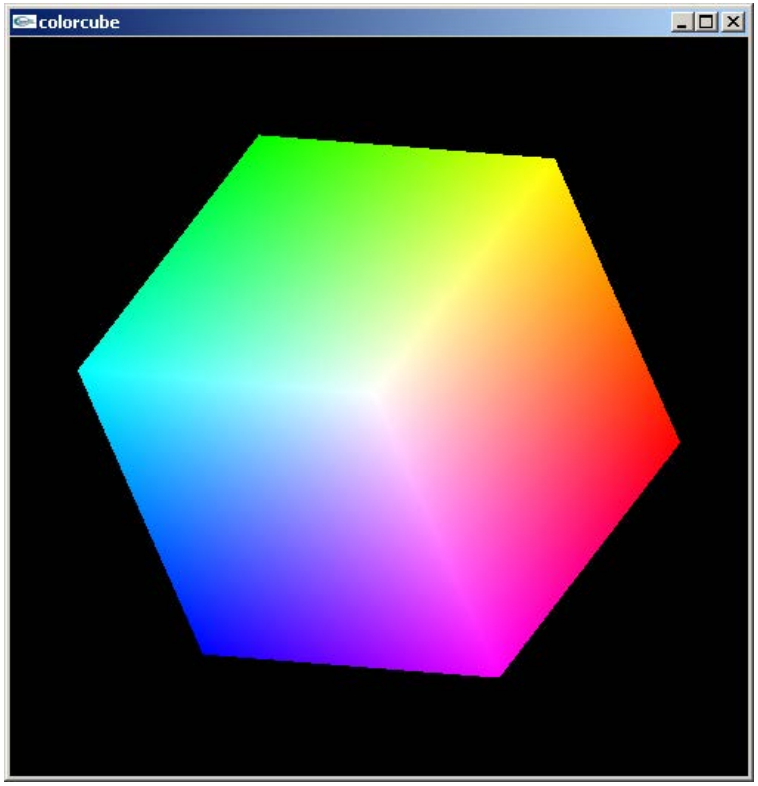
Color and State

- 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
  `glShadeModel(GL_FLAT)`



Display function with colors

```
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);    // clear the framebuffer
    glShadeModel(GL_SMOOTH);
    glBegin(GL_POLYGON);             // start drawing a polygon
        glColor3ub(255,0,0);         // Red
        glVertex2f(-0.5, -0.5);
        glColor3ub(0, 255,0);        // Green
        glVertex2f(-0.5, 0.5);
        glColor3ub(0,0,255);         // Blue
        glVertex2f(0.5, 0.5);
        glColor3ub(255,255,255);     // White
        glVertex2f(0.5, -0.5);

    glEnd();                         // end drawing polygon

    glFlush();
}
```

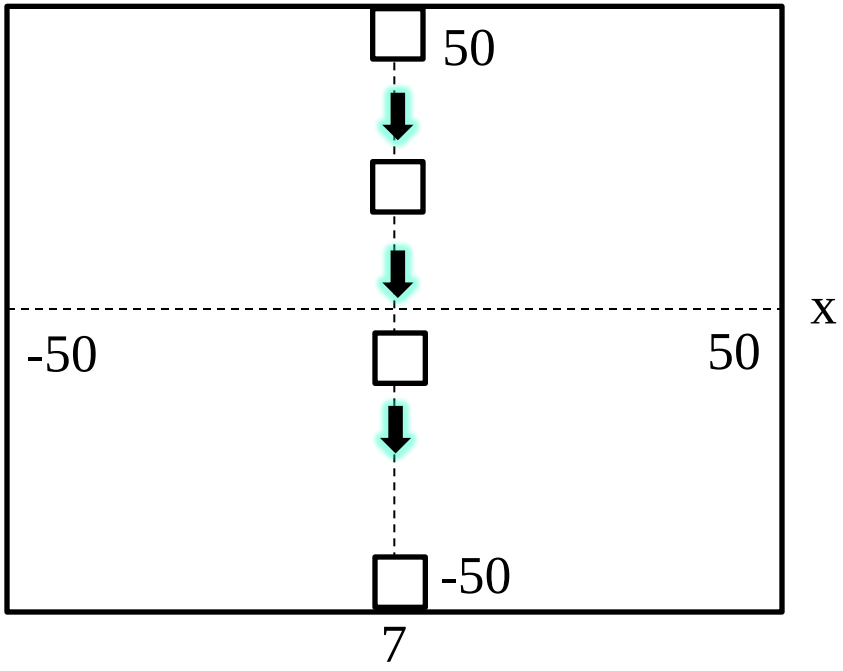
See code “sample3.c”

Programming with OpenGL
Example: BoxDrop

BoxDrop

- Lets consider a simple program that has some animation

The awesome "box drop" program!!!



World is glOrtho2D(-50,50,-50,50)
Box is sized only size 2.
Moves 1 unit at a time drop y-axis.

How to advance animation?

- Lets first consider using the keyboard
  - We will talk more about input callbacks in the next lecture
  - For now, anytime the space-key is pressed, we advance the box
- So, we will register three callback functions
  - Display
  - Keyboard
  - Reshape
- When the keyboard is pressed, we will post a display event. . .

BoxDrop1.c [init, keyboard]

```
#include <GL/glut.h>
void init(void) {
        glClearColor(0,0,0,1.0);
        glColor3f(1,1,1);
}
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case ' ':    /* Call display function */
         glutPostRedisplay();
         break;
      case 27:  /*  Escape Key  */
         exit(0);
         break;
        }
}
```

Set background/fore color

Keyboard function,
Records key, and even mouse location when pressed!

Key value is in ASCII code:

Note if ‘ ‘ (space) is pressed, we call “glutPostDisplay()” and not explicitly the “display()” function. This is the preferred glut application style.

To see ASCII codes for keys see: http://www.asciitable.com/

BoxDrop1.c [reshape]

```
void reshape(int w, int h) {
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D (-50.0, 50.0,
                  -50.0*(GLfloat)h/(GLfloat)w,
                   50.0*(GLfloat)h/(GLfloat)w);
  else
      gluOrtho2D (-50.0*(GLfloat)w/(GLfloat)h,
                  50.0*(GLfloat)w/(GLfloat)h,
                  -50.0, 50.0);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

Standard reshape. Note that the "w" and "h" are passed in from the "Window" system.

This is a common trick. We maintain the square ratio even when w≠h.

So, scale the longer dimension to make it appear the same.

BoxDrop1.c [drawBox]

```
void drawBox(double y_offset)
{
  glBegin(GL_LINE_LOOP);
    glVertex2f(1.0, 1.0+y_offset);    // v1    v4 --> v1
    glVertex2f(1.0, -1.0+y_offset);   // v2    ^      |
    glVertex2f(-1.0,-1.0+y_offset);   // v3    |      v
    glVertex2f(-1.0, 1.0+y_offset);   // v4   v3 ---  v2
  glEnd();

  glFlush();
}
```

User defined func. Draw a square using line loop. Pass in y_offset.

BoxDrop1.c [display]

```
void display(void)
{
    static double y_offset = 50;      /* 1 */

    glClear(GL_COLOR_BUFFER_BIT);     /* 2 */

    drawBox(y_offset);                /* 3 */

    y_offset-=1.0;                    /* 4 */
    if (y_offset < -50)
      y_offset = 50;

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

1. Setup a static variable (this is like a global – its value won't change after the function exits – but its scope is only inside this function.

2. Clear the color

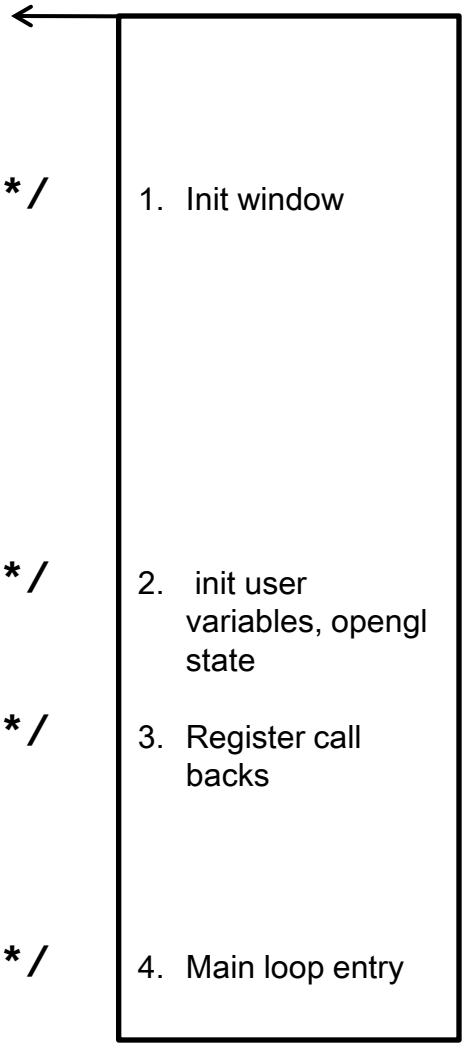
3. Draw Box with our offset.

4. Advance our offset by 1 unit (in –y direction). If we reach the "bottom", reset the position.

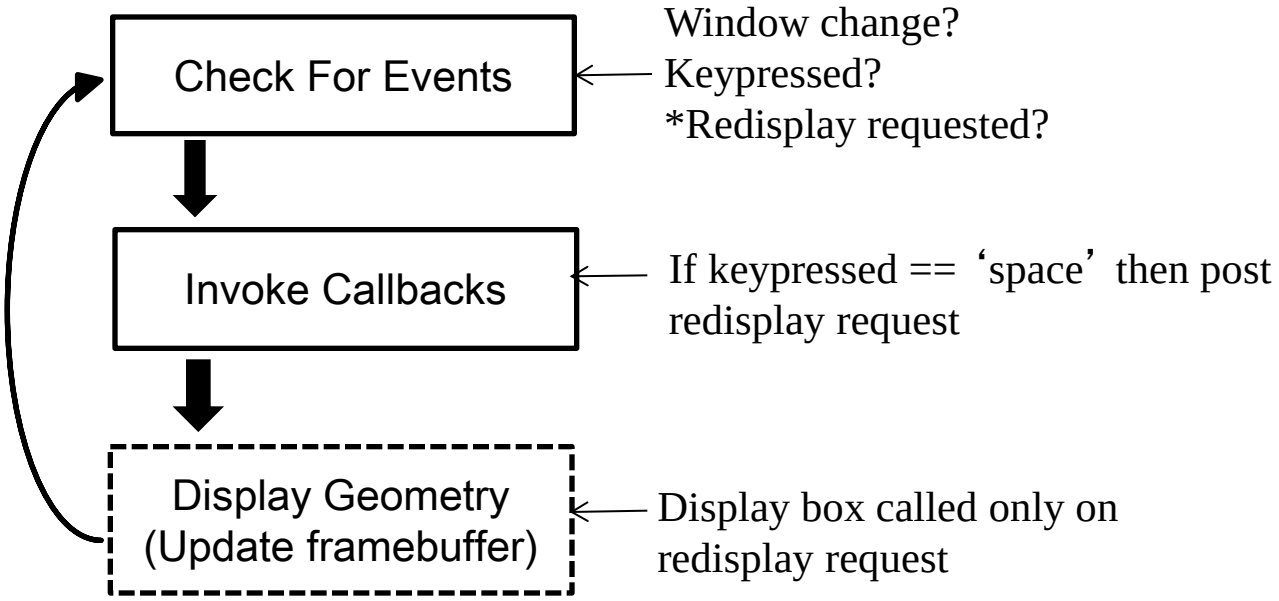
5. Swap the buffer (this avoids flashing)

BoxDrop1.c [main]

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);                          /* 1 */
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (512, 512);
   glutCreateWindow (argv[0]);

   init();                                         /* 2 */

   glutReshapeFunc (reshape);                      /* 3 */
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutMainLoop();                                 /* 4 */
   return 0;
}
```



Consider GLUT mainloop [BoxDrop1.c]



***What causes redisplay request?**

The window system can cause this if the window is considered “damaged.

We can call this in code using “glutPostReDisplay()” as done in our keyboard function.

How can we have animation? Idle Func

- GLUT is event driven
- When there are no events, no callbacks are invoked
  - This isn't good for animations
- Solutions? **`Idle`** callback function
  - This is a function you can register that is always invoked, even if there are no events (that is, when everything is "idle")
- Format of idle function (no return, no parameters):

```
void idle();
```

BoxDrop2.c [idle]

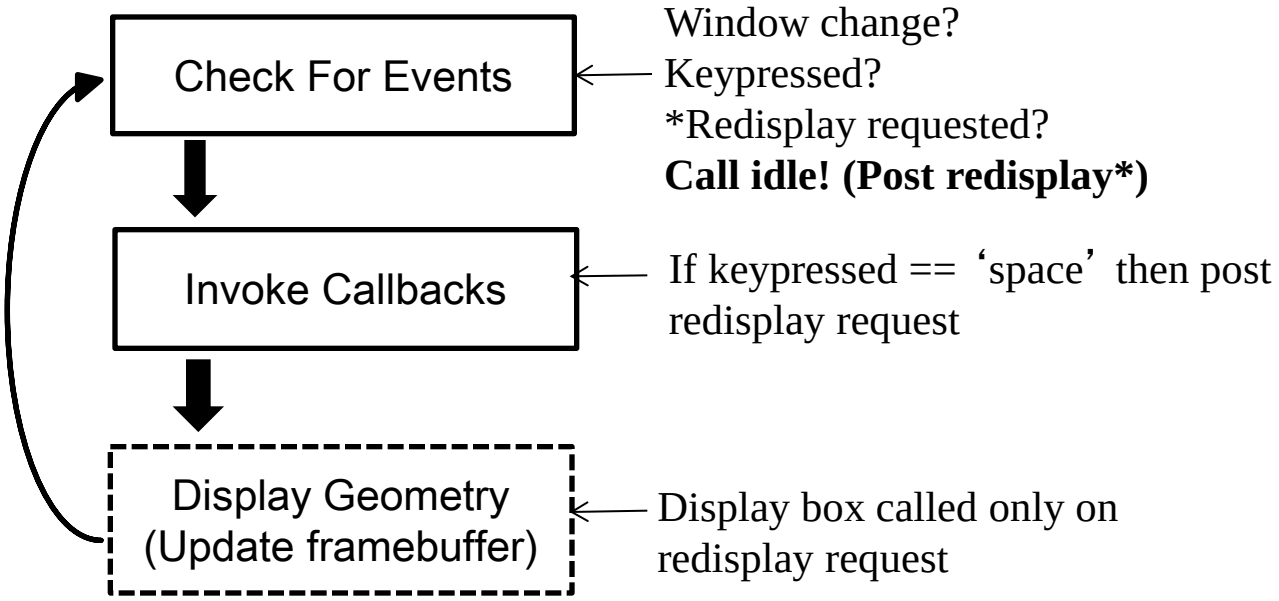
```
void idle(){
 glutPostReDisplay();
}
```

Add idle that simply calls "*glutPostRedisply()"

```
int main(int argc, char** argv)
{
   . . .
   glutCreateWindow (argv[0]);

   init();

   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutIdleFunc (idle);
   glutMainLoop();
   return 0;
}
```

Register idle

Consider GLUT mainloop [BoxDrop2.c]



What if they keyboard is pressed? This means that two glutPostRedisplay() has been called, one from the idle function and one from the keyboard! Will the display func be called twice? NO. It will only call display one time.

Programming with OpenGL
Part 3: Going “Cheap as free” 3D

Objectives

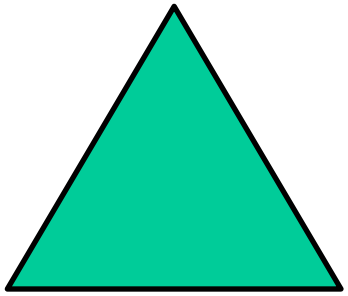
- Develop a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal
  - Introduce hidden-surface removal

Three-Dimensional Applications

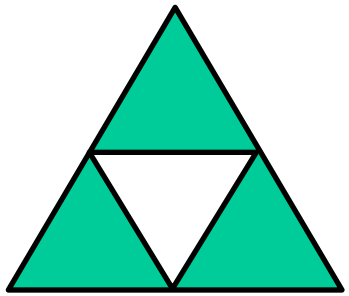
- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
  - Not much changes
  - Use **`glVertex3*()`**
  - Have to worry about the order in which polygons are drawn or use hidden-surface removal
  - Polygons should be simple, convex, flat

Example : Sierpinski * Gasket (2D)

- Start with a triangle



- Connect bisectors of sides and remove central triangle

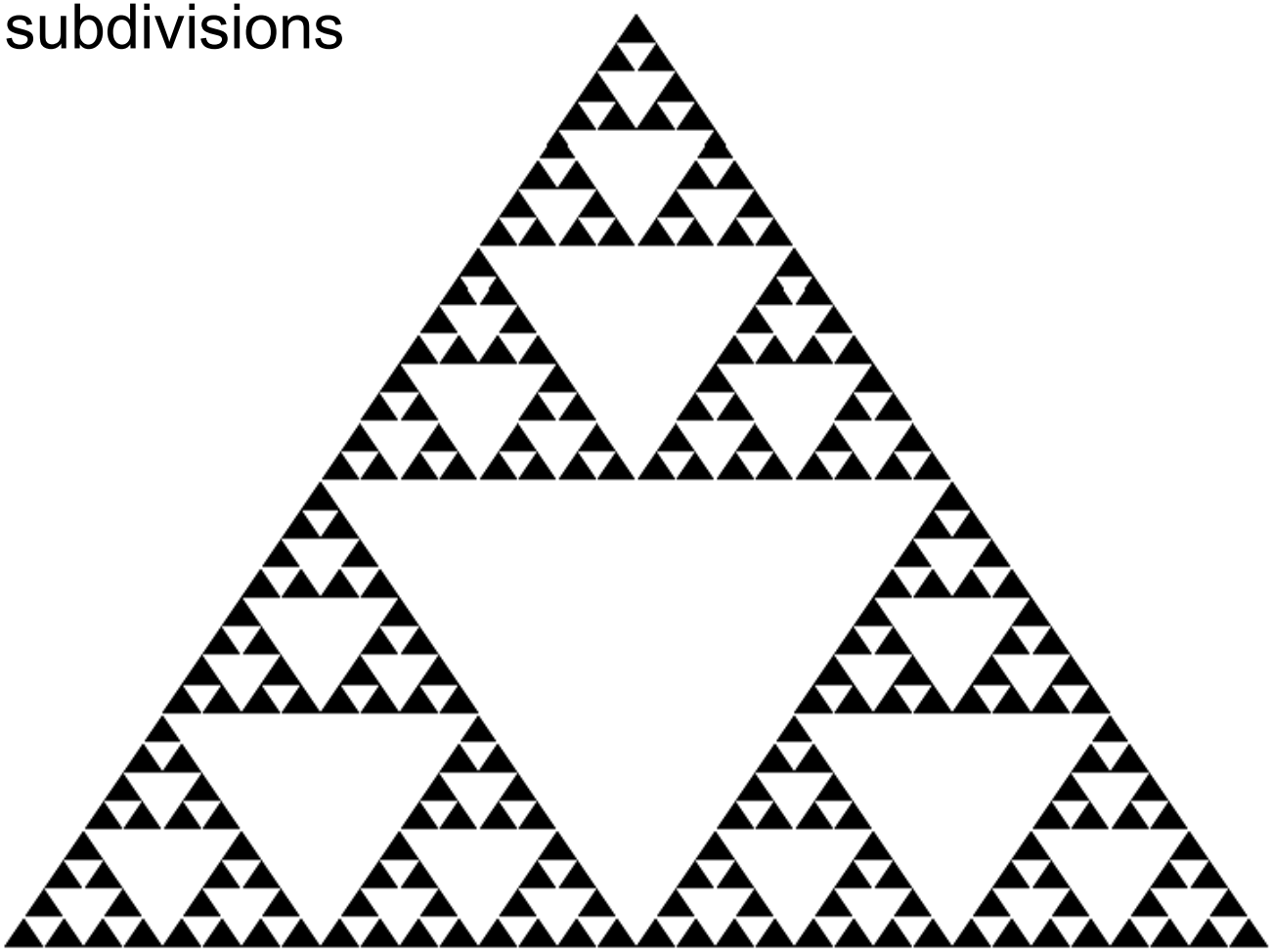


- Repeat

Pronounced: sir-pen-ski

Example

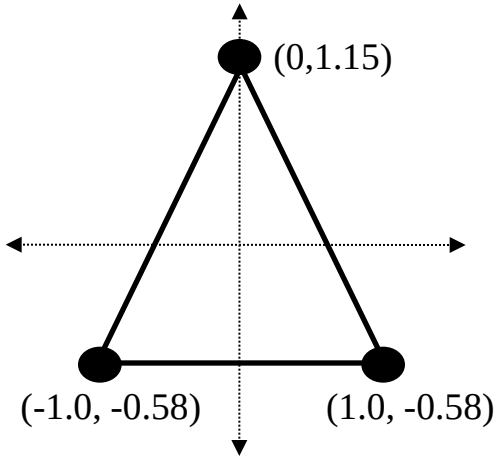
- Five subdivisions



Interesting fact: 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
  - Only exists in theory – but some nature reflects this property (think snow-flakes)

Gasket Program (2D version)




```
#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 */
```

See gasket2d.c

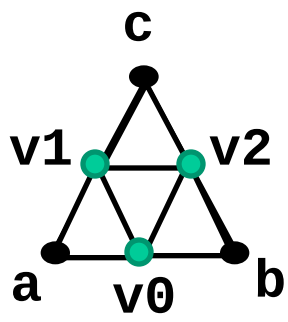
Draw One Triangle

```
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];
    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 */
}
```

Stopping condition.



`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);
}
```

Initial call to divide_triangle.

Stopping condition passed in also.

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

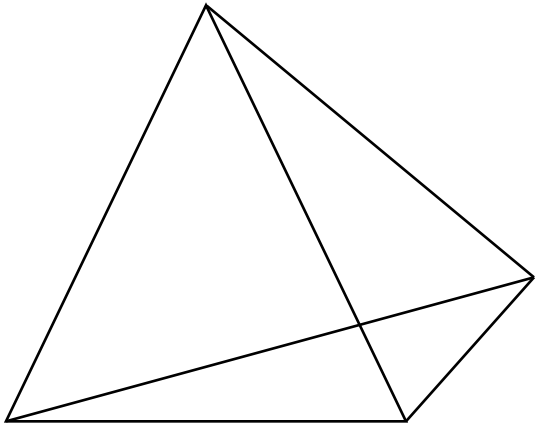
- By having the **`glBegin`** and **`glEnd`** in the display callback rather than in the function **`triangle`** and using **`GL_TRIANGLES`** rather than **`GL_POLYGON`** in **`glBegin`**, we call **`glBegin`** and **`glEnd`** only once for the entire gasket rather than once for each triangle!
  - This can be more efficient, allows all the triangles to be passed down in one continuous stream.

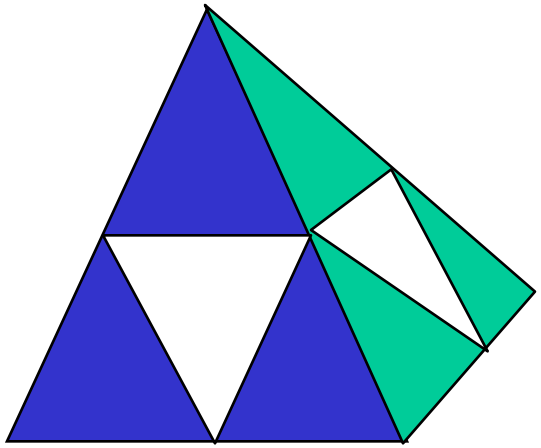
Moving to 3D

- 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

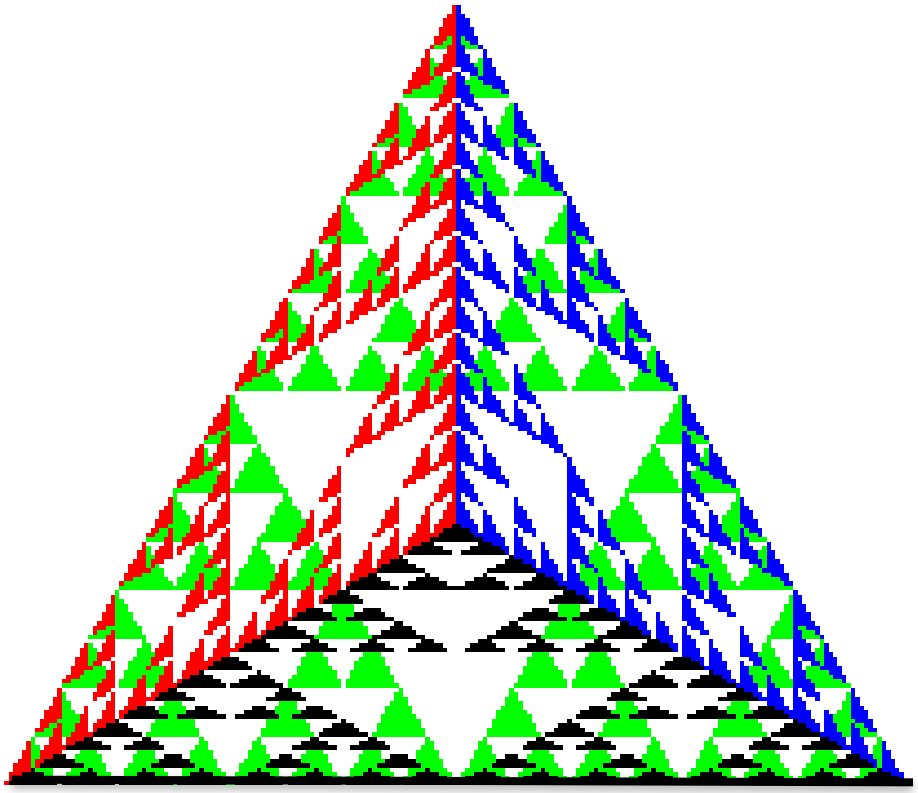




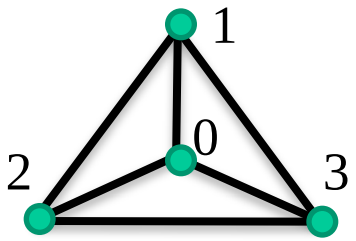
- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra

Example

After 5 iterations



triangle Code



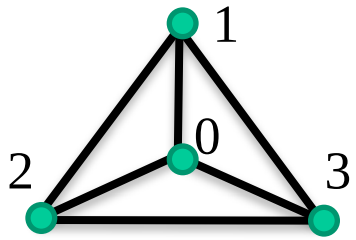

```
GLfloat v[4][3]={{0.0, 0.0, 1.0},                    0
                 {0.0, 0.942809, -0.33333},          1
                 {-0.816497, -0.471405, -0.333333},  2
                 {0.816497, -0.471405, -0.333333}};  3

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

See gasket3d_1.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);
}
```

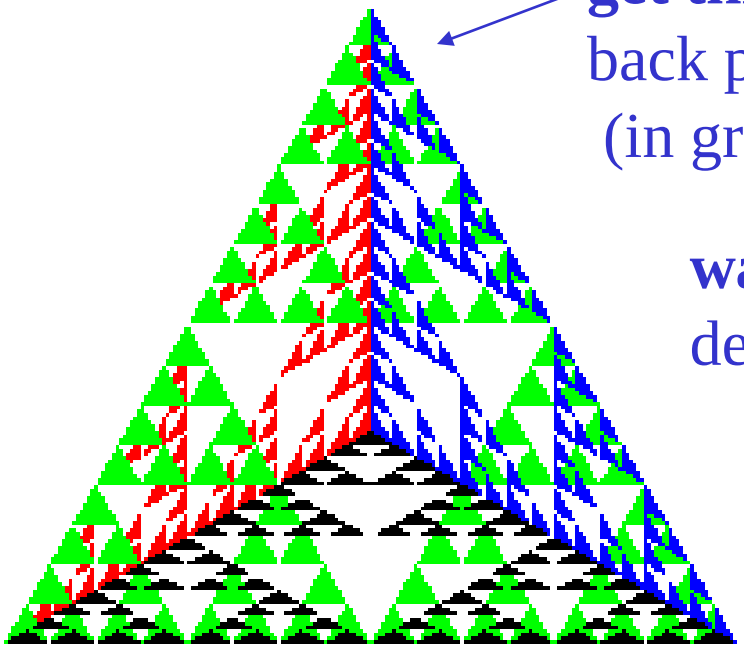
tetrahedron Code




```
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);
}
```

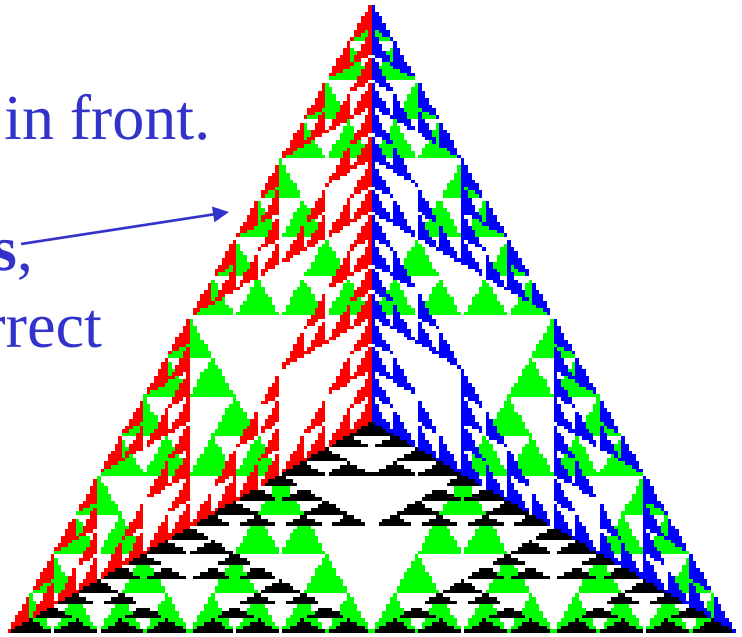
Almost Correct

- 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



get this –
back plane
(in green) is in front.

want this,
depth correct

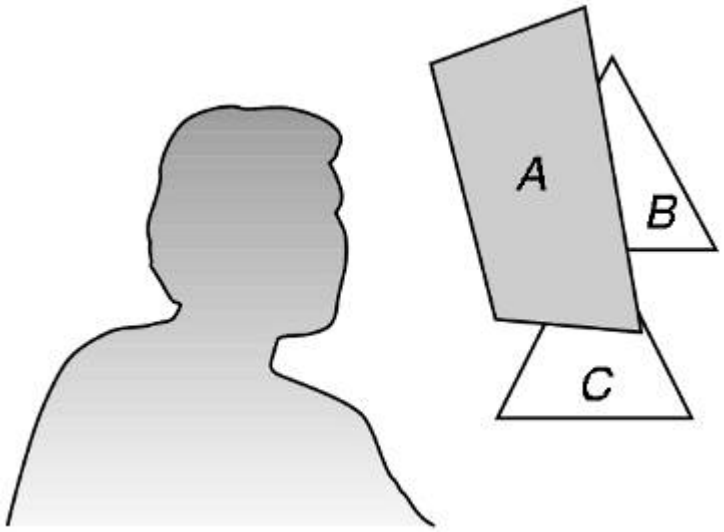


Incorrect depth ordering

Correct depth ordering

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

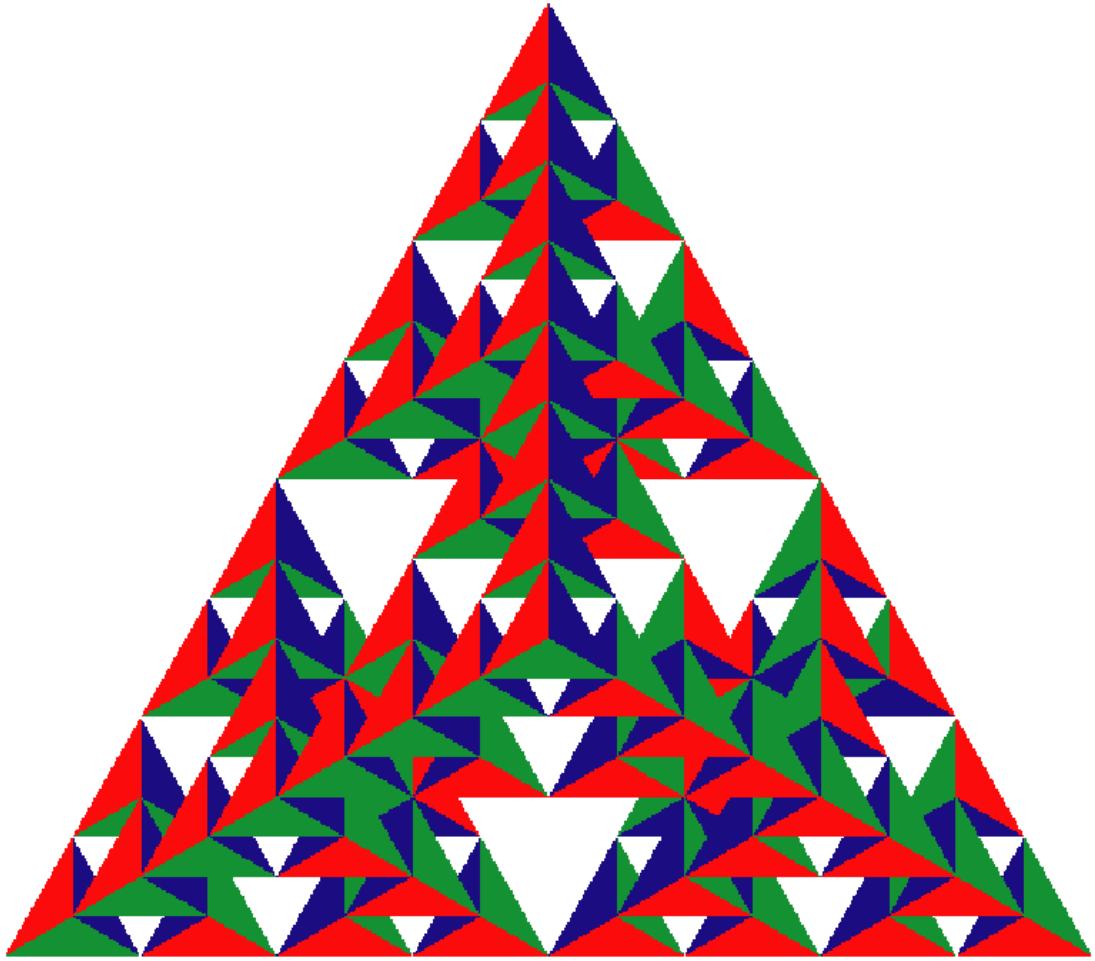
- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- It must be
  - Requested in **`main.c`**
    - **`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)`**
  - Enabled in **`init.c`**
    - **`glEnable(GL_DEPTH_TEST)`**
  - Cleared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

See gasket3d_2.c

Surface vs. Volume Subdivision

- 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
- Good programming exercise

Volume Subdivision



End of Lecture 2

-- next lecture GLUT Events