Programming with OpenGL
Part 5: More GLSL

CS 432 Interactive Computer Graphics

Prof. David E. Breen
Department of Computer Science

Objectives

- Coupling shaders to applications
  - Reading
  - Compiling
  - Linking
- Vertex Attributes
- Setting up uniform variables
- Example applications

Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables

Program Object

- Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```
GLuint myProgObj;
myProgObj = glCreateProgram();
 /* define shader objects here */
glUseProgram(myProgObj);
glLinkProgram(myProgObj);
```

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function **`glShaderSource`**
- If the shader is in a file, we can write a reader to convert the file to a string

Shader Reader

```
#include <stdio.h>

static char*
readShaderSource(const char* shaderFile)
{
   FILE* fp = fopen(shaderFile, "r");

   if ( fp == NULL ) { return NULL; }

   fseek(fp, 0L, SEEK_END);
   long size = ftell(fp);
```

Shader Reader (cont)

```
    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);

    buf[size] = '\0';
    fclose(fp);

    return buf;
}
```

Adding a Vertex Shader

```
GLuint vShader;
GLchar vShaderfile[] = "my_vertex_shader";
GLchar* vSource =
        readShaderSource(vShaderFile);
vShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vShader, 1, &vSource, NULL);
glCompileShader(vShader);
glAttachShader(myProgObj, vShader);
```

Go to InitShader.cpp

Vertex Attributes

- Vertex attributes are named in the shaders
- Linker forms a table
- Application can get index from table and tie it to an application variable
- Similar process for uniform variables

Vertex Attribute Example

```
#define BUFFER_OFFSET( offset )
   ((GLvoid*) (offset))

GLuint loc =
  glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 2, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Uniform Variable Example

```
GLint angleParam;
angleParam = glGetUniformLocation(myProgObj,
     "angle");
/* angle defined in shader */

/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */

glUniform1f(angleParam, my_angle);
```

Double Buffering

- Updating the value of a uniform variable opens the door to animating an application
  - Execute glUniform in display callback
  - Force a redraw through glutPostRedisplay()
- Need to prevent a partially redrawn frame buffer from being displayed
- Draw into back buffer
- Display front buffer
- Swap buffers after updating finished

Adding Double Buffering

- Request a double buffer
  - glutInitDisplayMode(GLUT_DOUBLE)
- Swap buffers

```
void mydisplay()
{
        glClear(……);
        glDrawArrays();
        glutSwapBuffers();
}
```

Idle Callback

- Idle callback specifies function to be executed when no other actions pending
  - glutIdleFunc(myIdle);

```
void myIdle()
{
    // recompute display
    glutPostRedisplay();
}
```

Attribute and Varying Qualifiers

- Starting with GLSL 1.5 attribute and varying qualifiers have been replaced by in and out qualifiers
- No changes needed in application
- Vertex shader example:

```
#version 1.4
attribute vec3 vPosition;
varying vec3 color;
```

```
#version 1.5
in vec3 vPosition;
out vec3 color;
```

Adding Color

- If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes
- Let's associate a color with each vertex
- Set up an array of same size as positions
- Send to GPU as a buffer object

Setting Colors

```
typedef  vec3 color3;
color3 base_colors[4] = {color3(1.0, 0.0. 0.0), ….
color3 colors[NumVertices];
vec3 points[NumVertices];

//in loop setting positions

colors[i] = basecolors[color_index]
position[i] = …….
```

Setting Up Buffer Object

```
//need larger buffer

glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
   sizeof(colors), NULL, GL_STATIC_DRAW);

//load data separately

glBufferSubData(GL_ARRAY_BUFFER, 0,
   sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
   sizeof(colors), colors);
```

Second Vertex Array

```
// vPosition and vColor identifiers in vertex shader

loc = glGetAttribLocation(program, “vPosition”);
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0,
   BUFFER_OFFSET(0));

loc2 = glGetAttribLocation(program, “vColor”);
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
   BUFFER_OFFSET(sizeof(points)));
```

Coloring Each Vertex (deprecated)

```
attribute vec3 vPosition, vColor;
varying vec3 color;

void main()
{
  gl_Position = vec4(vPosition, 1);
  color = vColor;
}
```

Coloring Each Vertex

```
in vec3 vPosition, vColor;
out vec3 color;

void main()
{
  gl_Position = vec4(vPosition, 1);
  color = vColor;
}
```

Coloring Each Fragment (deprecated)

```
varying vec3 color;

void main()
{
  gl_FragColor = vec4(color, 1);
}
```

Coloring Each Fragment

```
in vec3 color;
out vec4 fragcolor;

void main()
{
  fragcolor = vec4(color, 1);
}
```

Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Wave Motion Vertex Shader

```
in vec4 vPosition;
out vec4 color;
uniform float xs, zs, // frequencies
uniform float h; // height scale
uniform float time; // time from app
void main()
{
  vec4 t = vPosition;
  t.y = vPosition.y
     + h*sin(time + xs*vPosition.x)
     + h*sin(time + zs*vPosition.z);
  gl_Position = t;
}
```

Particle System

```
in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 vel;
uniform float g, m, t;
void main()
{
 vec3 object_pos;
 object_pos.x = vPosition.x + vel.x*t;
 object_pos.y = vPosition.y + vel.y*t
                            + g/(2.0*m)*t*t;
 object_pos.z = vPosition.z + vel.z*t;
 gl_Position =
  ModelViewProjectionMatrix*vec4(object_pos,1);
}
```

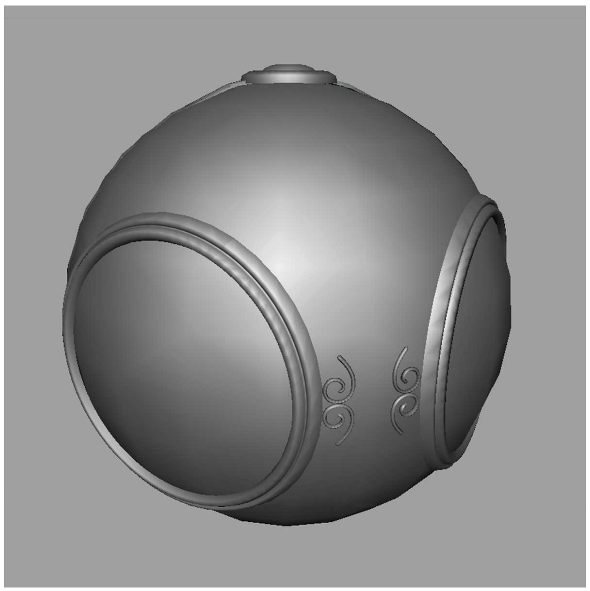
Vertex vs Fragment Lighting

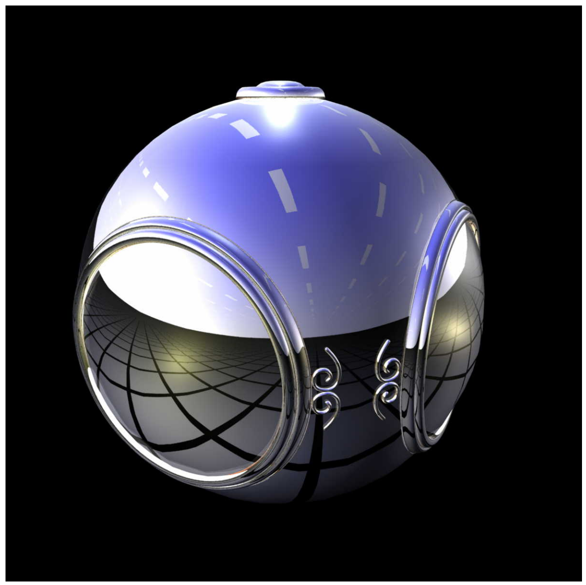
per vertex lighting
Goraud shading

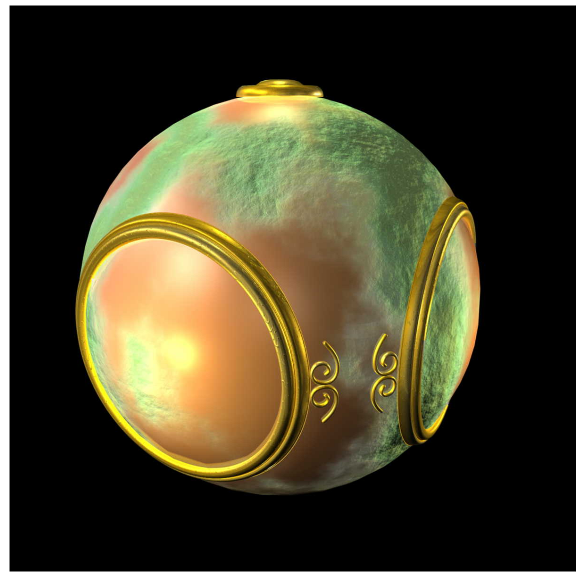
per fragment lighting
Phong shading

Fragment Shader Applications

Texture mapping







smooth shading

environment mapping

bump mapping

Programming with OpenGL Part 6: Three Dimensions

Objectives

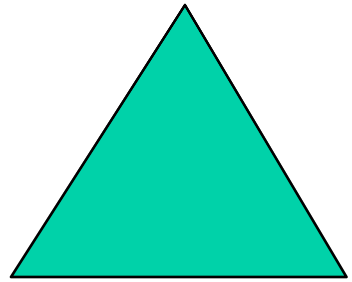
- Develop a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal

Three-dimensional Applications

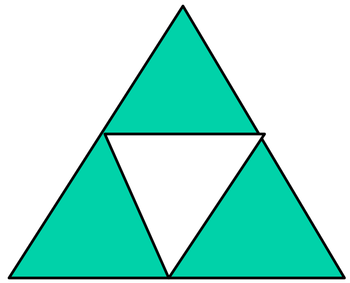
- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
  - Not much changes
  - Use **`vec3, glUniform3f`**
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal

Sierpinski Gasket (2D)

- Start with a triangle



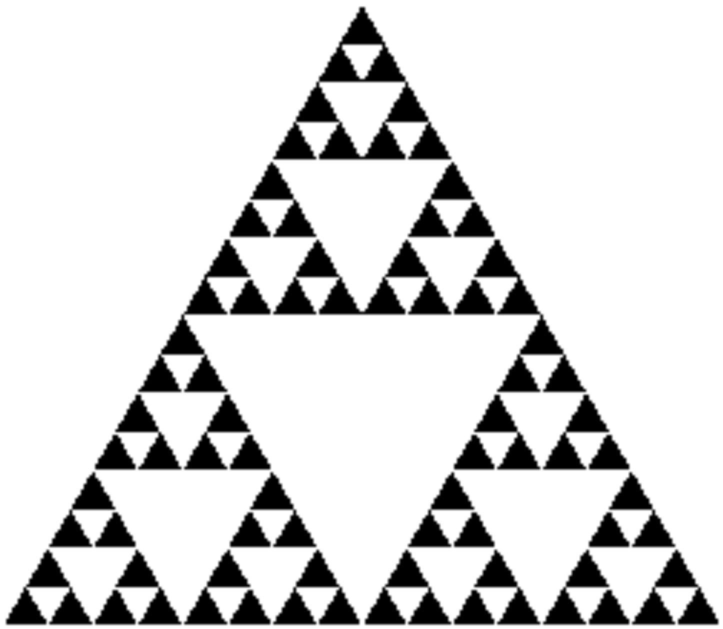
- Connect bisectors of sides and remove central triangle



- Repeat

Example

- Five 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
  - 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 */

point2 v[3] ={point2(-1.0, -0.58),
             point2(1.0, -0.58),
             point2 (0.0, 1.15)};

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

Draw one triangle

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

/* display one triangle  */
{
    //  static int i =0;  // This doesn't
                          // make sense
      points[i] = a;
      i++;
      points[i] = b;
      i++;
      points[i] = c;
      i++;
}
```

Triangle Subdivision

```
void divide_triangle(point2 a, point2 b, point2 c, int m)
{
/* triangle subdivision using vertex numbers */
    point2 ab, ac, bc;
    if(m>0)
    {
        ab = (a + b )/2;
        ac = (a + c)/2;
        bc = (b + c)/2;
        divide_triangle(a, ab, ac, m-1);
        divide_triangle(c, ac, bc, m-1);
        divide_triangle(b, bc, ac, m-1);
    }
    else(triangle(a,b,c));
 /* draw triangle at end of recursion */
}
```

display and init Functions

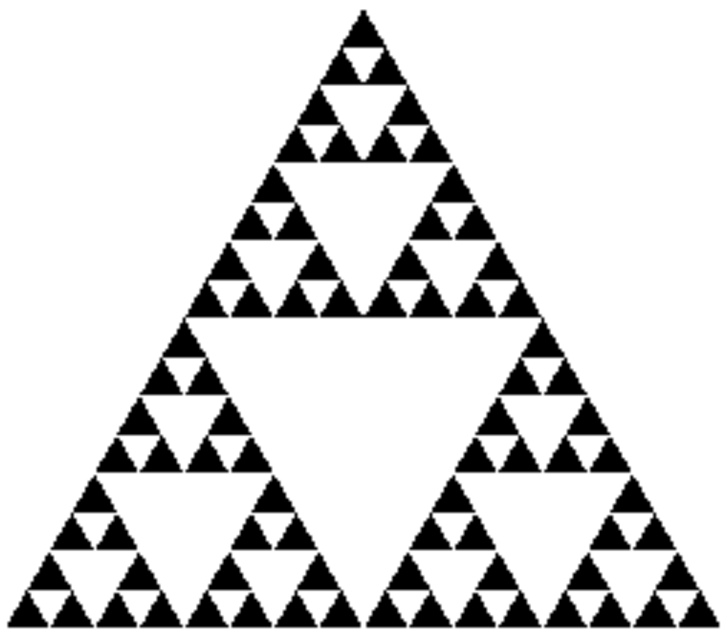
```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}

void myinit()
{
    vec2 v[3] = {point2(......
    .
    .
    divide_triangles(v[0], v[1], v[2], n);
    .
    .
}
```

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();
}
```

Example

- Five subdivisions



Moving to 3D

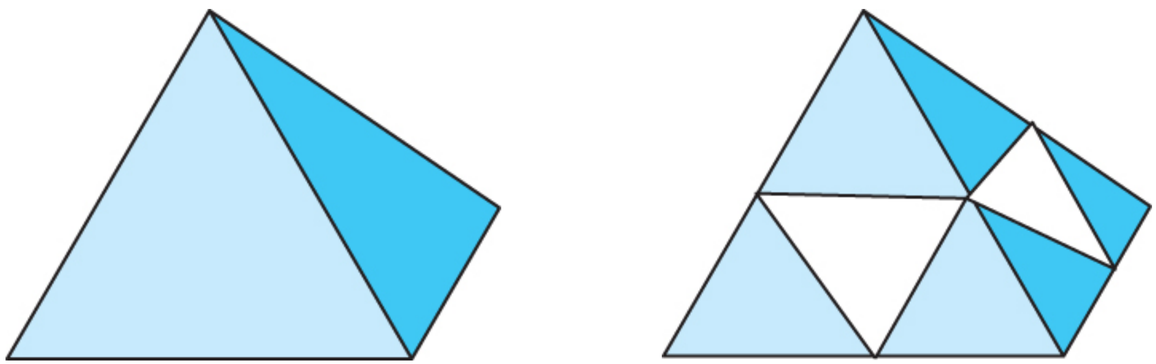
- We can easily make the program three-dimensional by using

```
point3 v[3]
```

and we start with a tetrahedron

3D Gasket

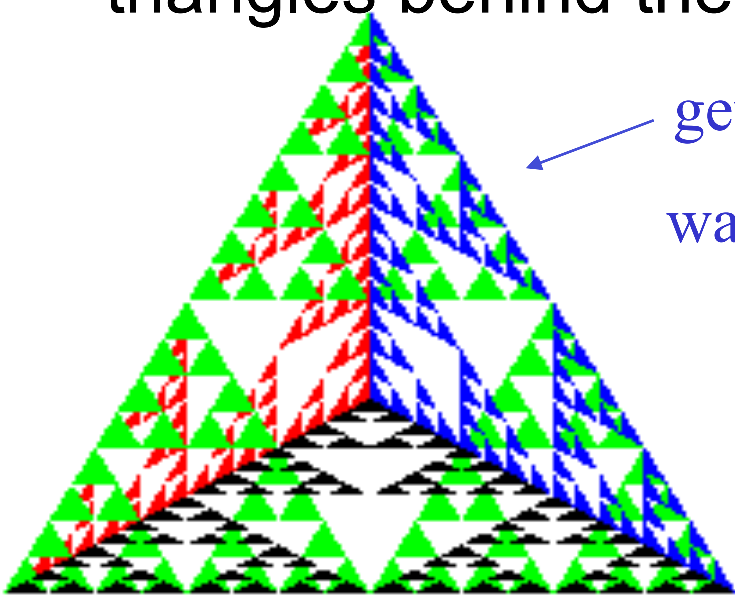
- We can subdivide each of the four faces



- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra
- Code almost identical to 2D example

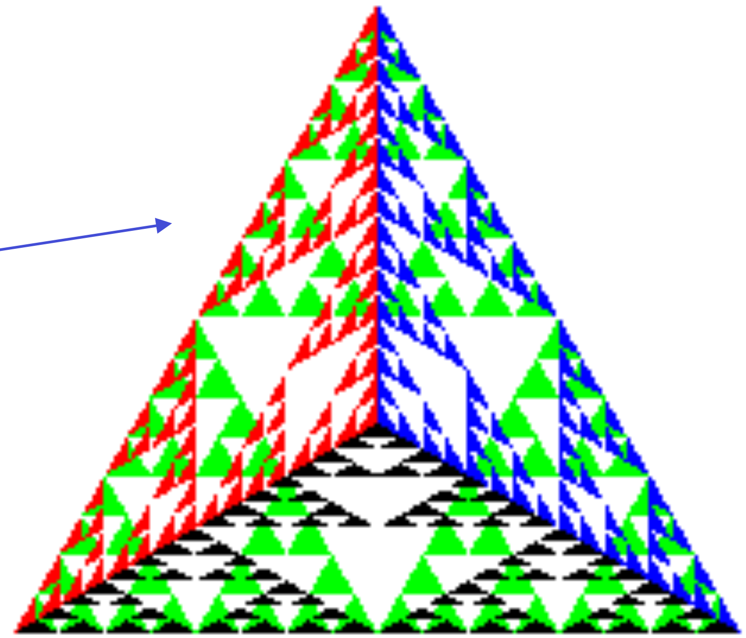
Almost Correct

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



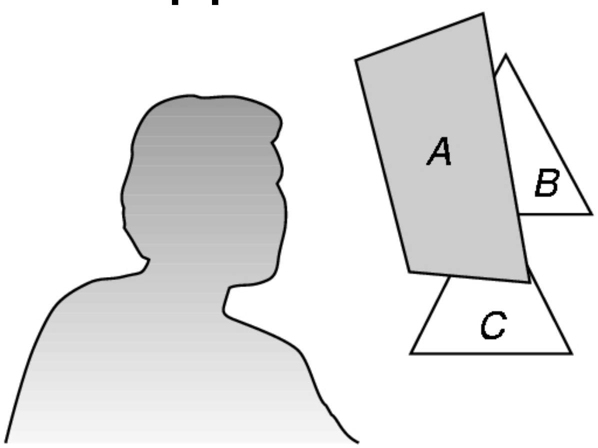
get this

want this



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



Z-buffering

- Z-buffering (depth-buffering) is a visible surface detection algorithm
- Implementable in hardware and software
- Requires data structure (z-buffer) in addition to frame buffer.
- Z-buffer stores values [0 .. ZMAX] corresponding to depth of each point.
- If the point is closer than one in the buffers, it will replace the buffered values

Z-buffering

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

+

5	5	5	5	5	5	5	
5	5	5	5	5	5		
5	5	5	5	5			
5	5	5	5				
5	5	5					
5	5						
5							

=

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
5	5	5	0	0	0	0	0
5	5	0	0	0	0	0	0
5	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
5	5	5	0	0	0	0	0
5	5	0	0	0	0	0	0
5	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

+

3							
4	3						
5	4	3					
6	5	4	3				
7	6	5	4	3			
8	7	6	5	4	3		

=

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
6	5	5	3	0	0	0	0
7	6	5	4	3	0	0	0
8	7	6	5	4	3	0	0
0	0	0	0	0	0	0	0

Z-buffering w/ front/back clipping

```
for (y = 0; y < YMAX; y++)
  for (x = 0; x < XMAX; x++) {
        F[x][y] = BACKGROUND_VALUE;
        Z[x][y] = -1;  /* Back value in NPC */
  }
for (each polygon)
  for (each pixel in polygon's projection) {
        pz = polygon's z-value at pixel coordinates (x,y)
        if (pz < FRONT && pz > Z[x][y]) {  /* New point is behind front
                                    plane & closer than previous point */
              Z[x][y] = pz;
              F[x][y] = polygon's color at pixel coordinates (x,y)
        }
  }
```

Using the *z*-buffer algorithm

- 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