



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



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
- `gl.shaderSource(fragShdr, fragElem.text);`
- If shader is in HTML file, we can get it into application by `getElementById` method
- If the shader is in a file, we can write a reader to convert the file to a string



Adding a Vertex Shader

```
var vertShdr;  
var vertElem =  
    document.getElementById( vertexShaderId );  
  
vertShdr = gl.createShader( gl.VERTEX_SHADER );  
  
gl.shaderSource( vertShdr, vertElem.text );  
gl.compileShader( vertShdr );  
  
// after program object created  
gl.attachShader( program, vertShdr );
```



Adding a Fragment Shader

```
var fragShdr;  
var fragElem =  
    document.getElementById( fragmentShaderId );  
  
fragShdr = gl.createShader( gl.FRAGMENT_SHADER );  
  
gl.shaderSource( fragShdr, fragElem.text );  
gl.compileShader( fragShdr );  
  
// after program object created  
gl.attachShader( program, fragShdr );
```



Shader File Reader

- Following code may be a security issue with some browsers if run locally

```
function getShader(gl, shaderName, type) {  
    var shader = gl.createShader(type);  
    shaderScript = loadFileAJAX(shaderName);  
    if (!shaderScript) {  
        alert("Could not find shader source:  
            "+shaderName);  
    }  
}
```



Program Object

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

```
var program = gl.createProgram();
```

```
gl.attachShader( program, vertShdr );  
gl.attachShader( program, fragShdr );  
gl.linkProgram( program );
```

Go to `initShaders.js`



Attribute and Varying Qualifiers

- Starting with OpenGL GLSL 1.4 *attribute* and *varying* qualifiers have been replaced by *in* and *out* qualifiers
- No changes needed in application
- Vertex shader example:

```
# WebGL version 1.0  
# GLSL ES version 1.0  
# GLSL version 1.2  
attribute vec3 vPosition;  
varying vec3 color;
```

```
# WebGL version 2.0  
# GLSL ES version 3.0  
# GLSL version 1.4  
in vec3 aPosition;  
out vec3 vColor;
```



Vertex Attributes

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



Adding Color

- If we set a color in the application, we can send it to the shaders as a per 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



Sending Vertices from Application

```
var vBuffer = gl.createBuffer();  
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );  
gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices),  
               gl.STATIC_DRAW );
```

```
var aPosition = gl.getAttribLocation( program, "aPosition" );  
gl.vertexAttribPointer( aPosition, 3, gl.FLOAT, false, 0, 0 );  
gl.enableVertexAttribArray( aPosition );
```



Sending Colors from Application

```
var cBuffer = gl.createBuffer();  
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );  
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),  
               gl.STATIC_DRAW );  
  
var aColor = gl.getAttribLocation( program, "aColor" );  
gl.vertexAttribPointer( aColor, 3, gl.FLOAT, false, 0, 0 );  
gl.enableVertexAttribArray( aColor );
```



Sending a Uniform Variable

```
// in application  
vec4 color = vec4(1.0, 0.0, 0.0, 1.0);  
colorLoc = gl.getUniformLocation( program, "color" );  
gl.uniform4fv( colorLoc, color );
```

```
// in fragment shader (similar in vertex shader)  
uniform vec4 color;  
out vec4 fcolor;  
void main()  
{  
    fColor = color;  
}
```



Coloring Each Vertex (New)

```
in vec3 aPosition, aColor;  
out vec3 vColor;
```

```
void main()  
{  
    gl_Position = vec4(aPosition, 1);  
    vColor = aColor;  
}
```



Coloring Each Fragment (New)

```
precision highp float;  
in vec3 vColor;  
out vec4 fColor;  
  
void main()  
{  
    fColor = vec4(vColor, 1);  
}
```




Drawing Data in Buffers

```
function render() {  
  
    gl.clear( gl.COLOR_BUFFER_BIT );  
    gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 );  
  
}
```



Review Code

Go to `HW2.html`

`HW2.js`

`MV.js`



Programming with WebGL

Part 6: Going to Three Dimensions



Objectives

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

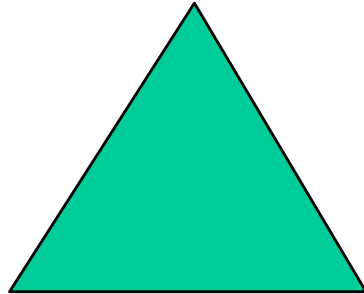


Three-dimensional Applications

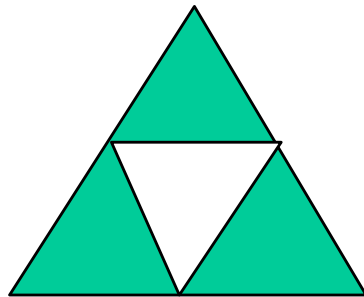
- In Open/WebGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
 - Not much changes
 - Use `vec3`, `gl.uniform3f`
 - Have to worry about the order in which primitives are rendered or use hidden-surface removal

Sierpinski Gasket (2D)

- Start with a triangle



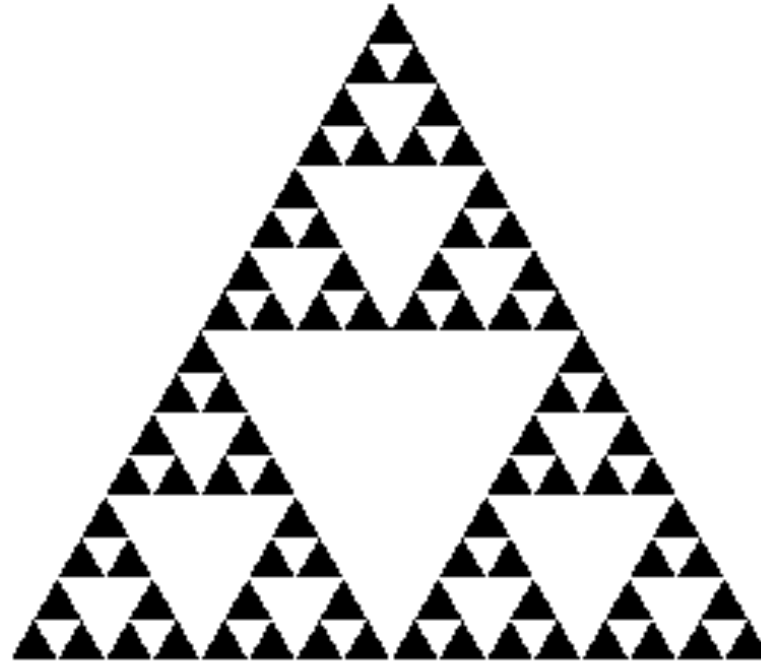
- Connect bisectors of sides and remove central triangle



- 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

- HTML file
 - Same as in other examples
 - Pass through vertex shader
 - Fragment shader sets color
 - Read in JS file



Gasket Program

```
var points = [];  
var NumTimesToSubdivide = 5;  
  
/* initial triangle */  
  
var vertices = [  
    vec2( -1, -1 ),  
    vec2(  0,  1 ),  
    vec2(  1, -1 )  
];  
  
divideTriangle( vertices[0], vertices[1],  
                vertices[2], NumTimesToSubdivide);
```



Draw one triangle

```
/* display one triangle */  
  
function triangle( a, b, c ) {  
    points.push( a, b, c );  
}
```



Triangle Subdivision

```
function divideTriangle( a, b, c, count ){
  // check for end of recursion
  if ( count === 0 ) {
    triangle( a, b, c );
  }
  else {
    //bisect the sides
    var ab = mix( a, b, 0.5 );
    var ac = mix( a, c, 0.5 );
    var bc = mix( b, c, 0.5 );
    --count;
    // three new triangles
    divideTriangle( a, ab, ac, count-1 );
    divideTriangle( c, ac, bc, count-1 );
    divideTriangle( b, bc, ab, count-1 );
  }
}
```



init()

```
var program = initShaders(gl, "vertex-shader",  
                           "fragment-shader");  
  
gl.useProgram( program );  
var bufferId = gl.createBuffer();  
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );  
gl.bufferData( gl.ARRAY_BUFFER, flatten(points),  
               gl.STATIC_DRAW );  
var aPosition = gl.getAttribLocation( program,  
                                       "aPosition" );  
gl.vertexAttribPointer( aPosition, 2, gl.FLOAT,  
                        false, 0, 0 );  
gl.enableVertexAttribArray( aPosition );  
render();
```

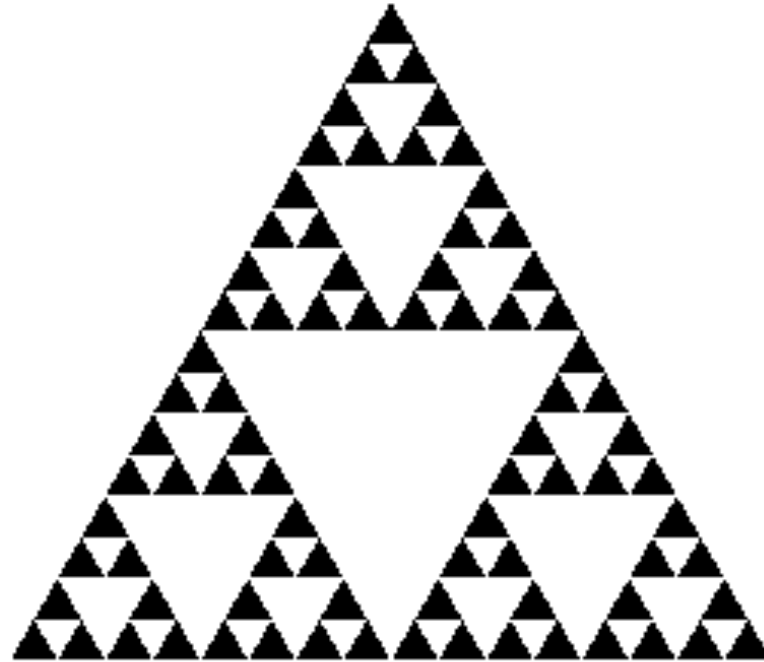


Render Function

```
function render() {  
    gl.clear( gl.COLOR_BUFFER_BIT );  
    gl.drawArrays( gl.TRIANGLES, 0,  
                  points.length );  
}
```

Example

- Five subdivisions





Moving to 3D

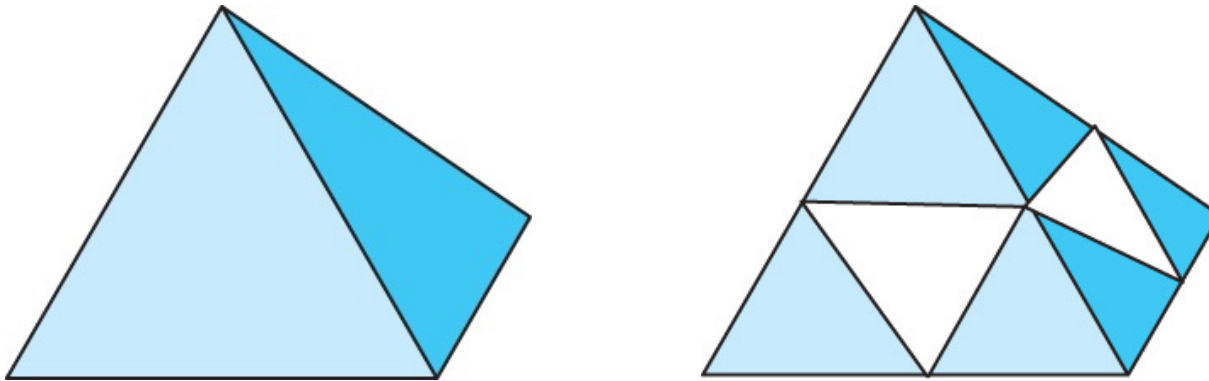
- We can easily make the program three-dimensional by using

```
point3 v[3]
```

and we start with a tetrahedron

3D Gasket

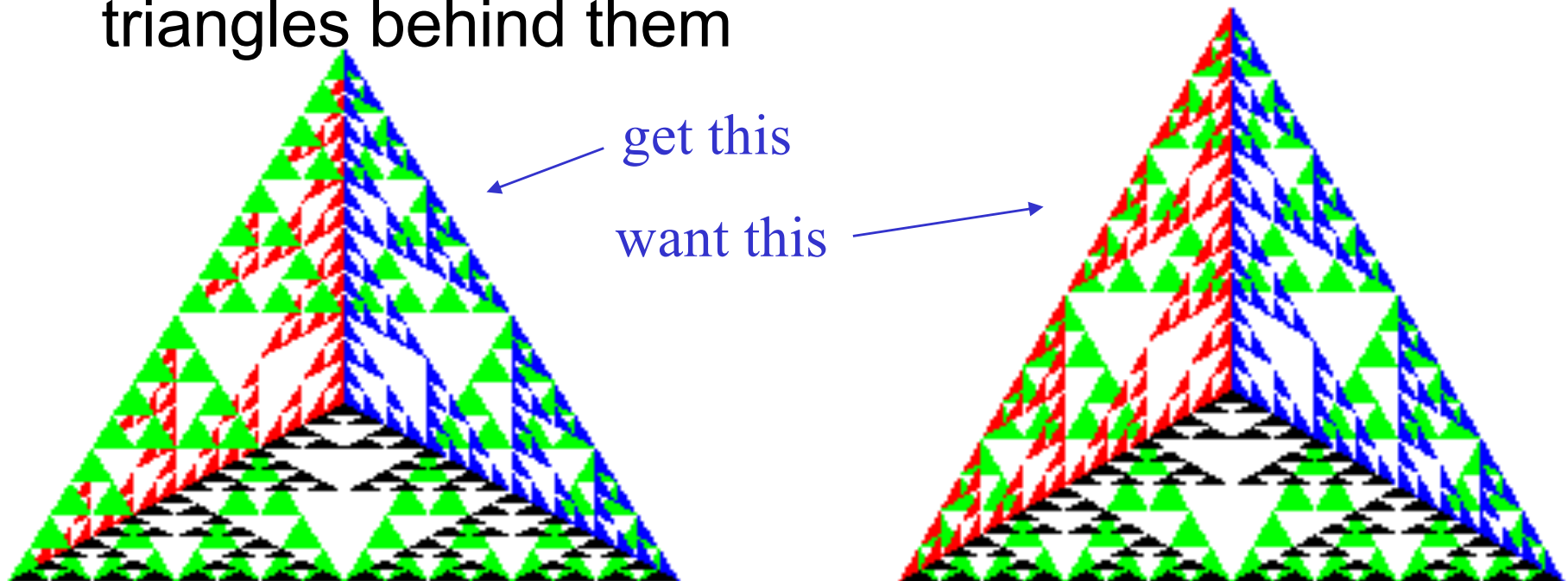
- We can subdivide each of the four faces



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

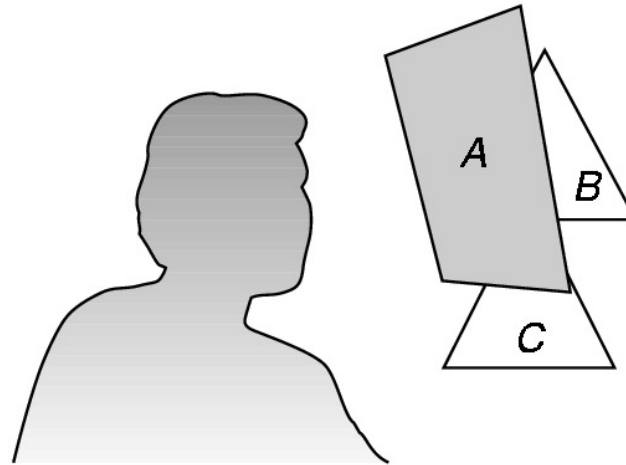
Almost Correct

- 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



Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- Open/WebGL 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 $[0 \dots ZMAX]$ 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

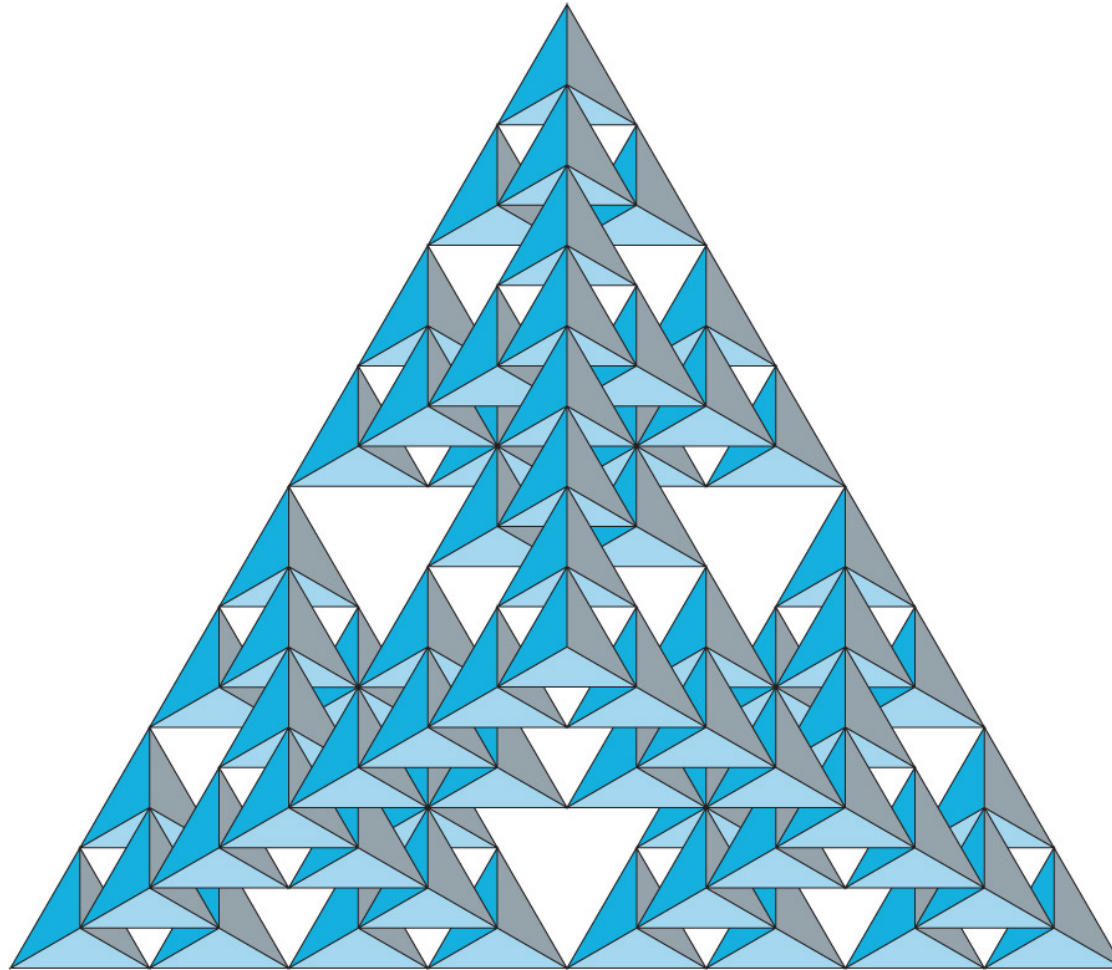
- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- Depth buffer is required to be available in WebGL
- It must be
 - Enabled
 - `gl.enable(gl.DEPTH_TEST)`
 - Cleared in for each render
 - `gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)`



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
- See text for code

Volume Subdivision





Swapping Shader Programs

- May need different shader programs for different objects or situations in a graphics application
- Could pass a flag to one set of shader programs to execute different code for different objects or situations
- This strategy might produce performance degradation
- Alternative: Dynamically change shader programs



Swapping Shader Programs

During initialization

- Create shaders

```
program1_ID = initShaders(gl, "vertex_shader1", "fr  
program2_ID = initShaders(gl, "vertex_shader2", "fr
```

- Get locations of uniform and attrib variables

```
thetaLoc1      = gl.getUniformLocation(program1_  
positionLoc1  = gl.getAttribLocation(program1_ID,
```

```
thetaLoc2      = gl.getUniformLocation(program2_  
positionLoc2  = gl.getAttribLocation(program2_ID,
```



Swapping Shader Programs

During initialization

- Create and fill buffers

```
positionBuffer1 = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer1);  
gl.bufferData(gl.ARRAY_BUFFER, flatten(positions1), g
```

```
positionBuffer2 = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer2);  
gl.bufferData(gl.ARRAY_BUFFER, flatten(positions2), g
```



Swapping Shader Programs

In the render() function

- Use first shader program

```
gl.useProgram(program1_ID);
```

- Set uniform value

```
gl.uniform1f(thetaLoc1, rotateAngle1);
```

- Bind a buffer and define its attributes

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer1);  
gl.vertexAttribPointer(positionLoc1, 2, gl.FLOAT, false, 0, 0);  
gl.enableVertexAttribArray(positionLoc1);
```

- Draw data in the buffer

```
gl.drawArrays(gl.TRIANGLE_FAN, 0, numOfVertices1);
```



Swapping Shader Programs

In the render() function

- Use second shader program

```
gl.useProgram(program2_ID);
```

- Set uniform value

```
gl.uniform1f(thetaLoc2, rotateAngle2);
```

- Bind a buffer and define its attributes

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer2);  
gl.vertexAttribPointer(positionLoc2, 2, gl.FLOAT, false,  
gl.enableVertexAttribArray(positionLoc2);
```

- Draw data in the buffer

```
gl.drawArrays(gl.TRIANGLE_FAN, 0, numOfVertices2);
```