Graphics

Jungchan Cho
Dept. of Software, Gachon University

Many slides from Edward Angel and Dave Shreine
Many examples are from https://webglfundamentals.org/

Quick Review of WebGL

Program Execution

- WebGL runs within the browser
  - complex interaction among the operating system, the window system, the browser and your code (HTML and JS)
- Simple model
  - Start with HTML file
  - files read in asynchronously
  - start with onload function
    - event driven input

Coordinate Systems

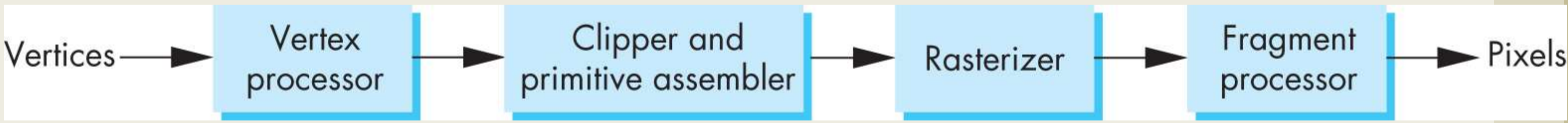
- The units in **`points`** are determined by the application and are called *object*, *world*, *model* or *problem coordinates*
- Viewing specifications usually are also in object coordinates
- Eventually pixels will be produced in *window coordinates*
- WebGL also uses some internal representations that usually are not visible to the application but are important in the shaders
- Most important is *clip coordinates*

Coordinate Systems and Shaders

- Vertex shader must output in clip coordinates
- Input to fragment shader from rasterizer is in window coordinates
- Application can provide vertex data in any coordinate system but shader must eventually produce **gl_Position** in clip coordinates
- Simple example uses clip coordinates

Modern OpenGL

- Performance is achieved by using GPU rather than CPU
- **Control GPU through programs called shaders**
- **Application's job is to send data to GPU**
- GPU does all rendering



Immediate Mode Graphics

- Geometry specified by vertices
  - Locations in space(2 or 3 dimensional)
  - Points, lines, circles, polygons, curves, surfaces
- Immediate mode
  - Each time a vertex is specified in application, its location is sent to the GPU
  - Old style uses **`glVertex`**
  - **Creates bottleneck between CPU and GPU**
  - Removed from OpenGL 3.1 and OpenGL ES 2.0

Retained Mode Graphics

- Put all vertex attribute data in array
- Send array to GPU to be rendered immediately
- Almost OK but problem is we would have to send array over each time we need another render of it
- **Better to send array over and store on GPU for multiple renderings**

Programming with WebGL: More GLSL

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

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

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

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

https://en.wikipedia.org/wiki/Null-terminated_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 );
```

Shader Reader

- Following code may be a security issue with some browsers if you try to run it locally
  - Cross Origin Request

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

Precision Declaration

- In GLSL for WebGL we must specify desired precision in fragment shaders
  - artifact inherited from OpenGL ES
  - ES must run on very simple embedded devices that may not support 32-bit floating point
  - All implementations must support mediump
  - No default for float in fragment shader
- Can use preprocessor directives (#ifdef) to check if highp supported and, if not, default to mediump

Pass Through Fragment Shader

```
#ifdef GL_FRAGMENT_SHADER_PRECISION_HIGH
  precision highp float;
#else
  precision mediump float;
#endif

varying vec4 fcolor;
void main(void)
{
     gl_FragColor = fcolor;
}
```

Programming with WebGL: Three Dimensions

Objectives

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

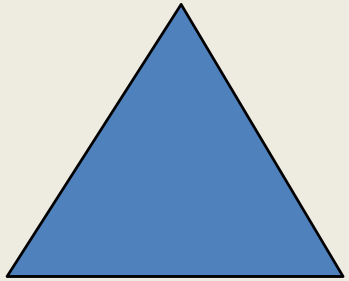
Three-dimensional Applications

- In WebGL, two-dimensional applications are a special case of three-dimensional graphics

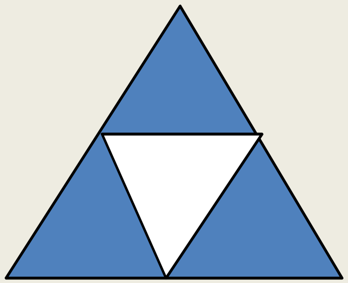
- Going to 3D
  - Not much changes
  - Use `vec3, gl.uniform3f, …`
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal

Sierpinski Gasket (2D)

- Start with a triangle



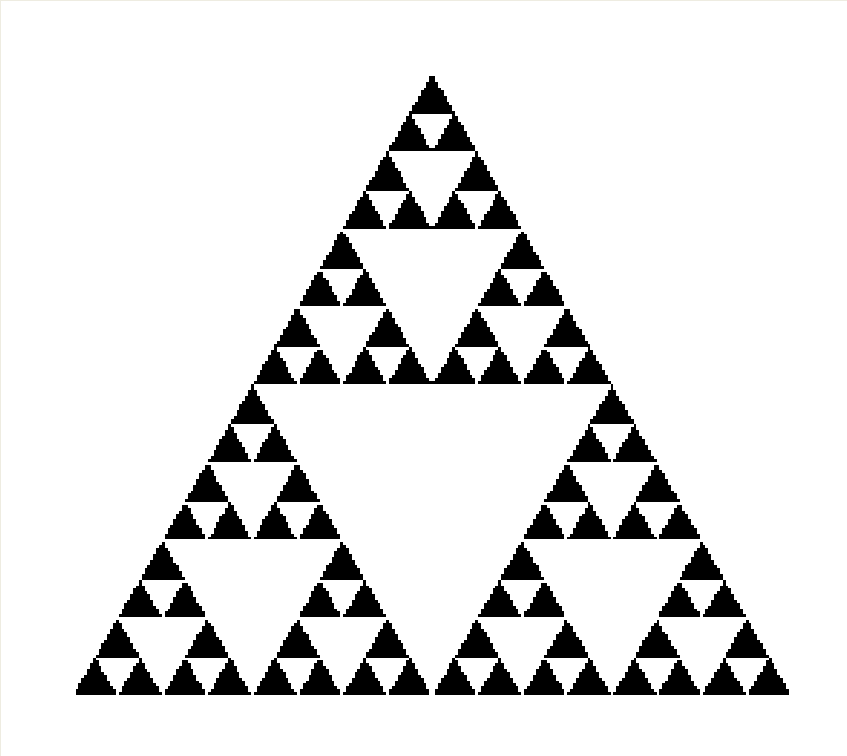
- Connect bisectors of sides and remove central triangle



- Repeat

Example

- Five subdivisions



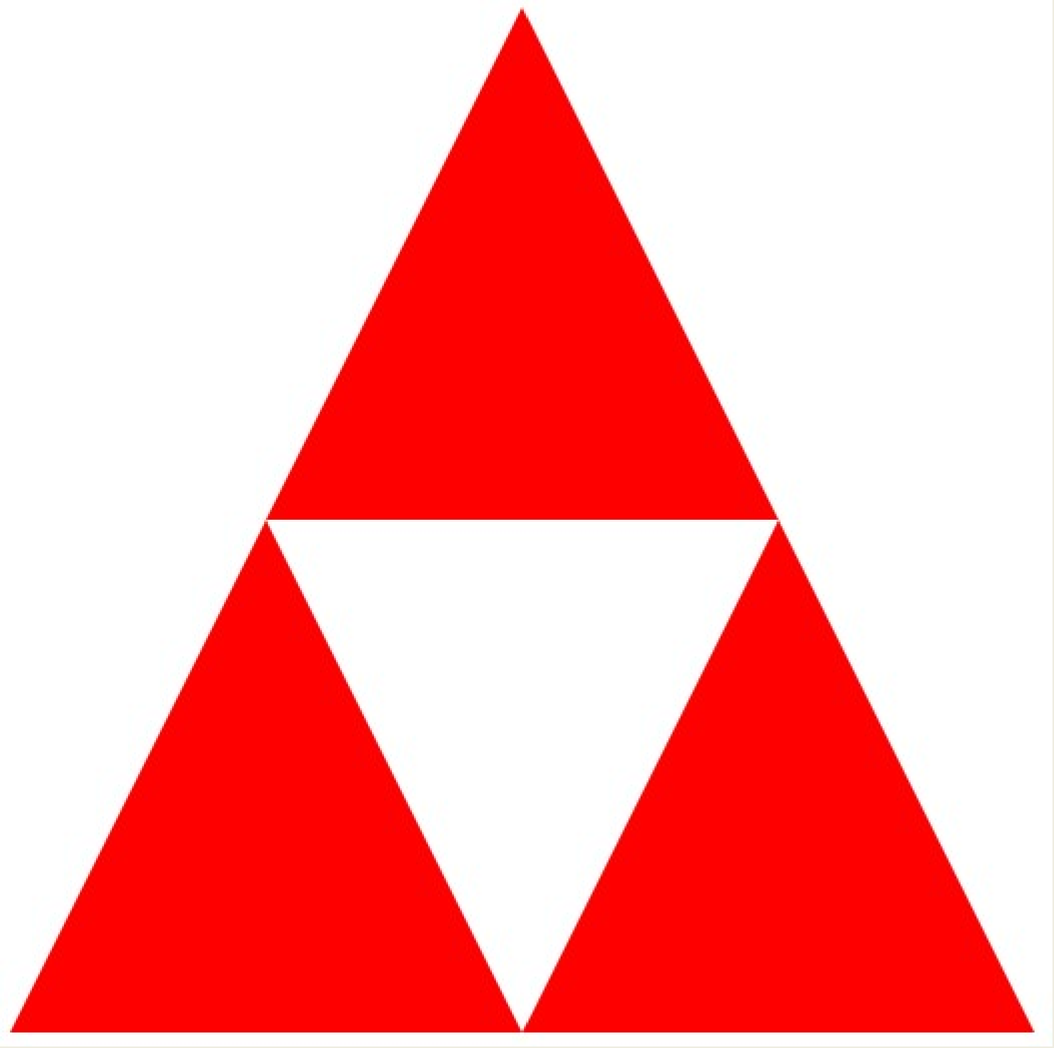
The gasket as a fractal

- 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

gasket2.html



Gasket Program (gasket2.js)

```
var points = [];
var NumTimesToSubdivide = 1;

/* 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 );
    }
}
```

```
/* display one triangle  */

function triangle( a, b, c ){
    points.push( a, b, c );
}
```

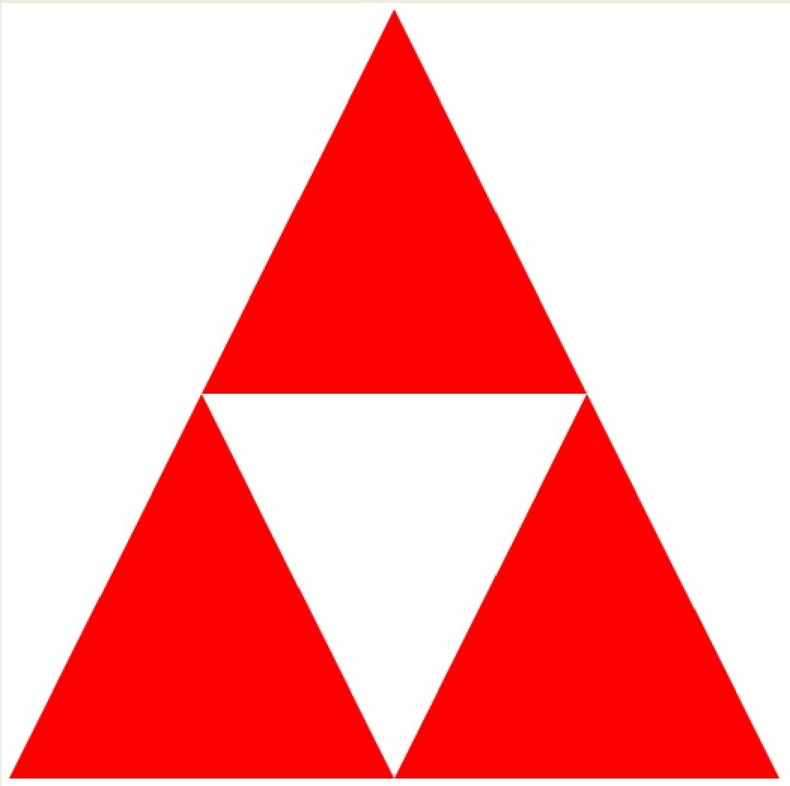
init()

```
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 vPosition = gl.getAttribLocation( program,
  "vPosition" );
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false,
  0, 0 );
gl.enableVertexAttribArray( vPosition );
render();
```

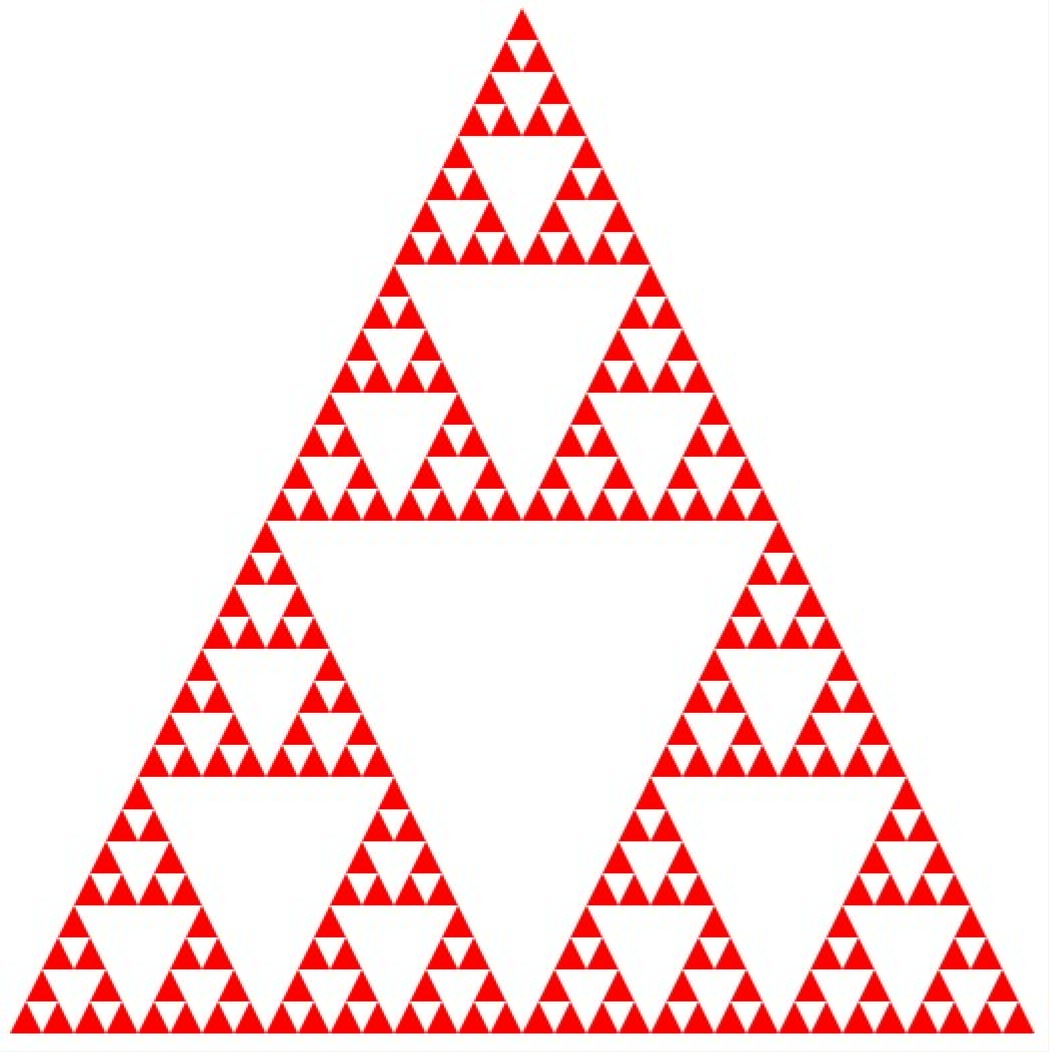
Render Function

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



gasket2.html

```
var NumTimesToSubdivide = 5;
```



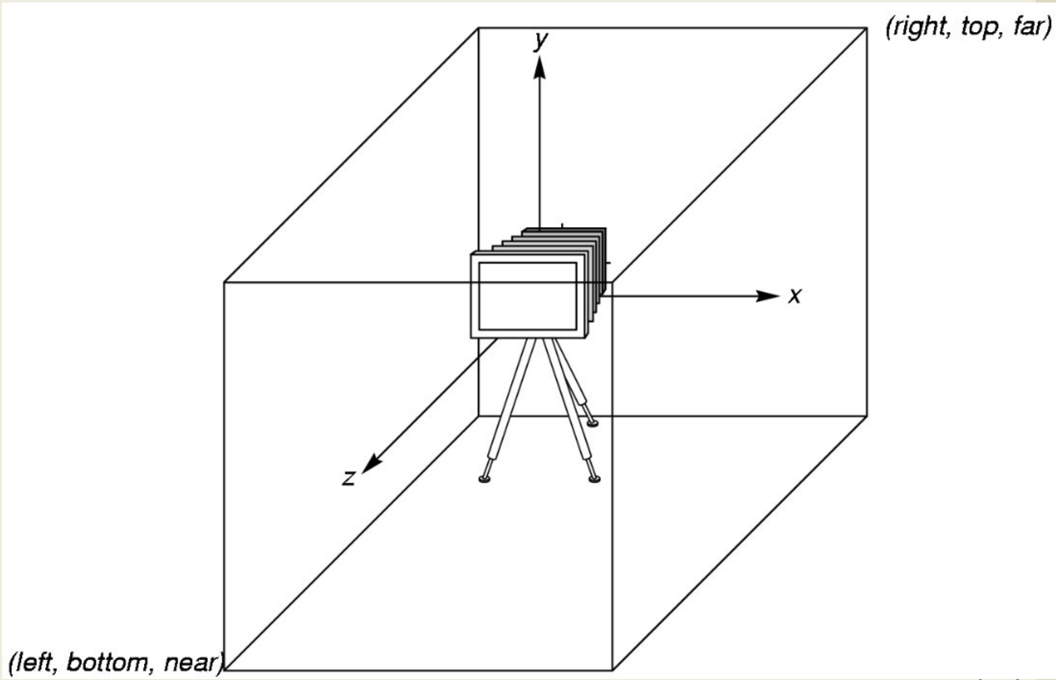
WebGL Camera

- WebGL 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 sides of

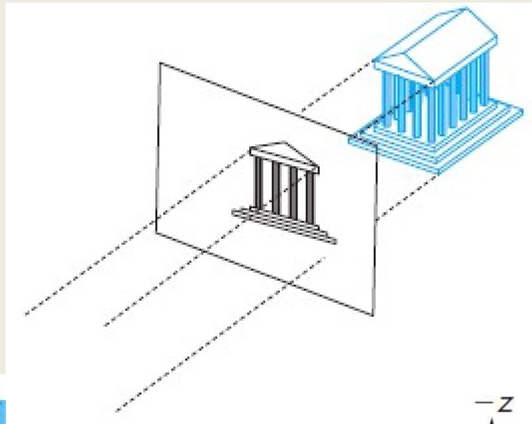
length 2

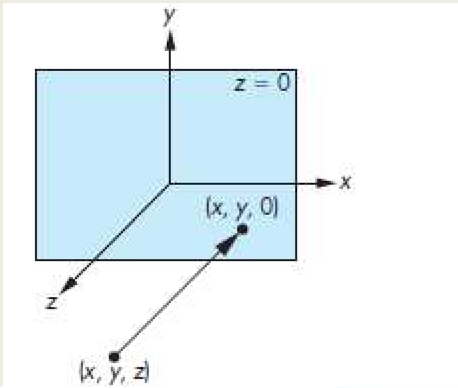


The Orthographic Viewing

$(x, y, z) \rightarrow (x, y, 0)$

- The simplest and OpenGL's default view is the orthographic projection.
  - Unlike a real camera, the orthographic projection can include objects behind the camera. Thus, because the plane $z = 0$ is located between -1 and 1, the two-dimensional plane intersects the viewing volume.
  - We discuss this projection and others in detail in Chapter 4.





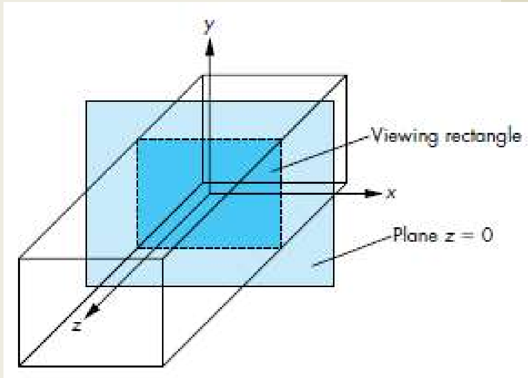
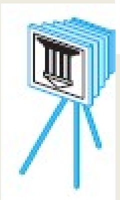
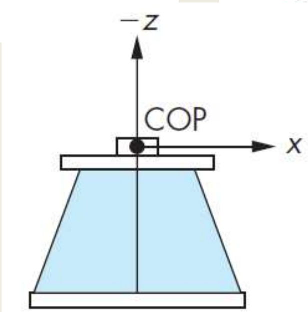


FIGURE 2.34 Viewing volume.





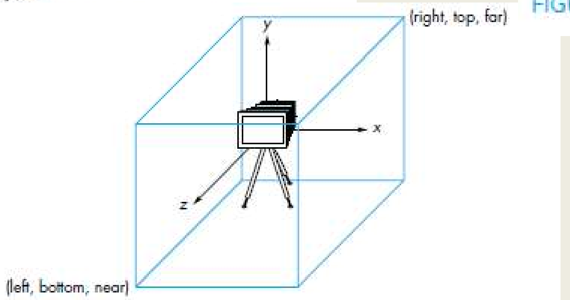


FIGURE 2.32 The default camera and an orthographic view volume.

Moving to 3D

- We can easily make the program three-dimensional by using three dimensional points and starting with a tetrahedron

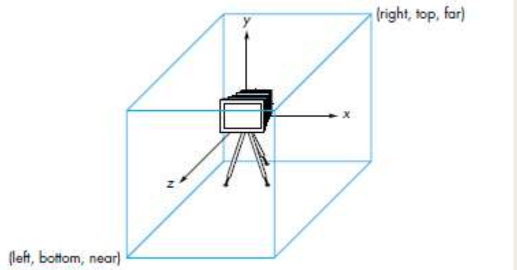


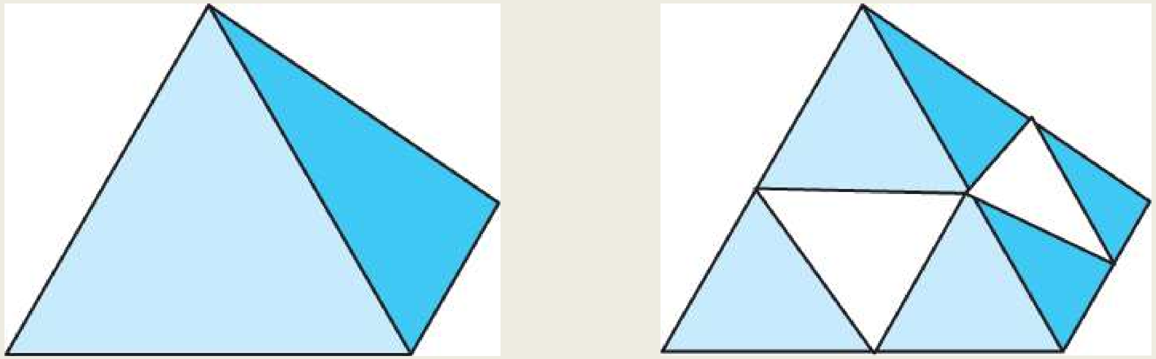
FIGURE 2.32 The default camera and an orthographic view volume.

```
var vertices = [
    vec3(  0.0000,  0.0000, -1.0000 ),
    vec3(  0.0000,  0.9428,  0.3333 ),
    vec3( -0.8165, -0.4714,  0.3333 ),
    vec3(  0.8165, -0.4714,  0.3333 )
];
```

subdivide each face

3D Gasket

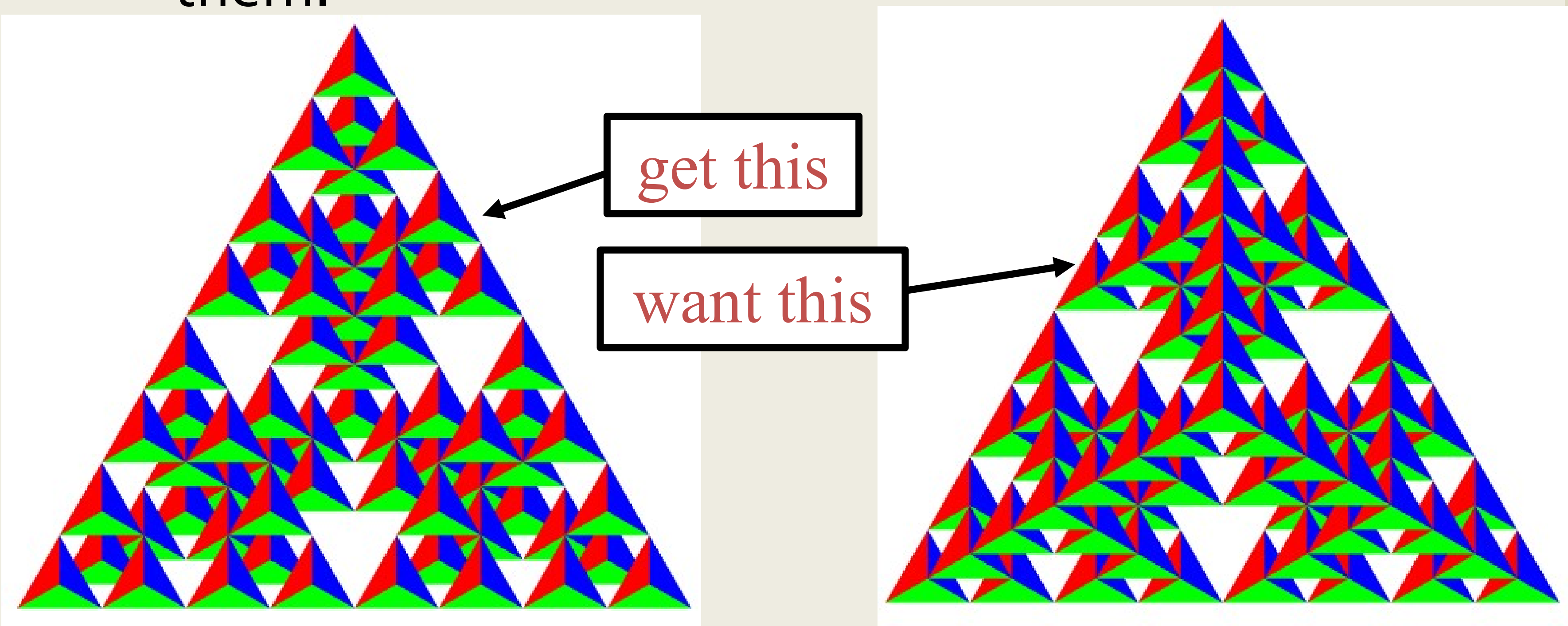
- We can subdivide each of the four faces



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

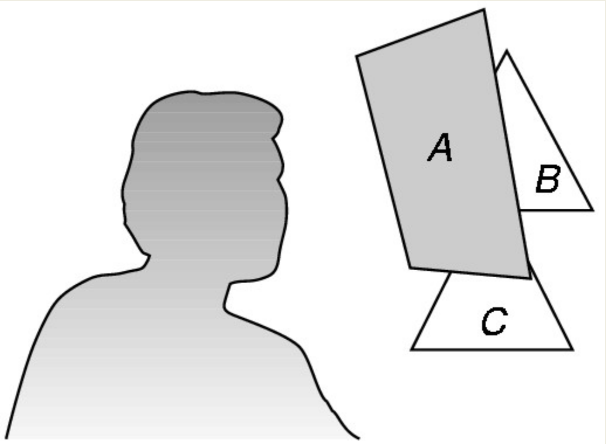
Almost Correct

- 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
- 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

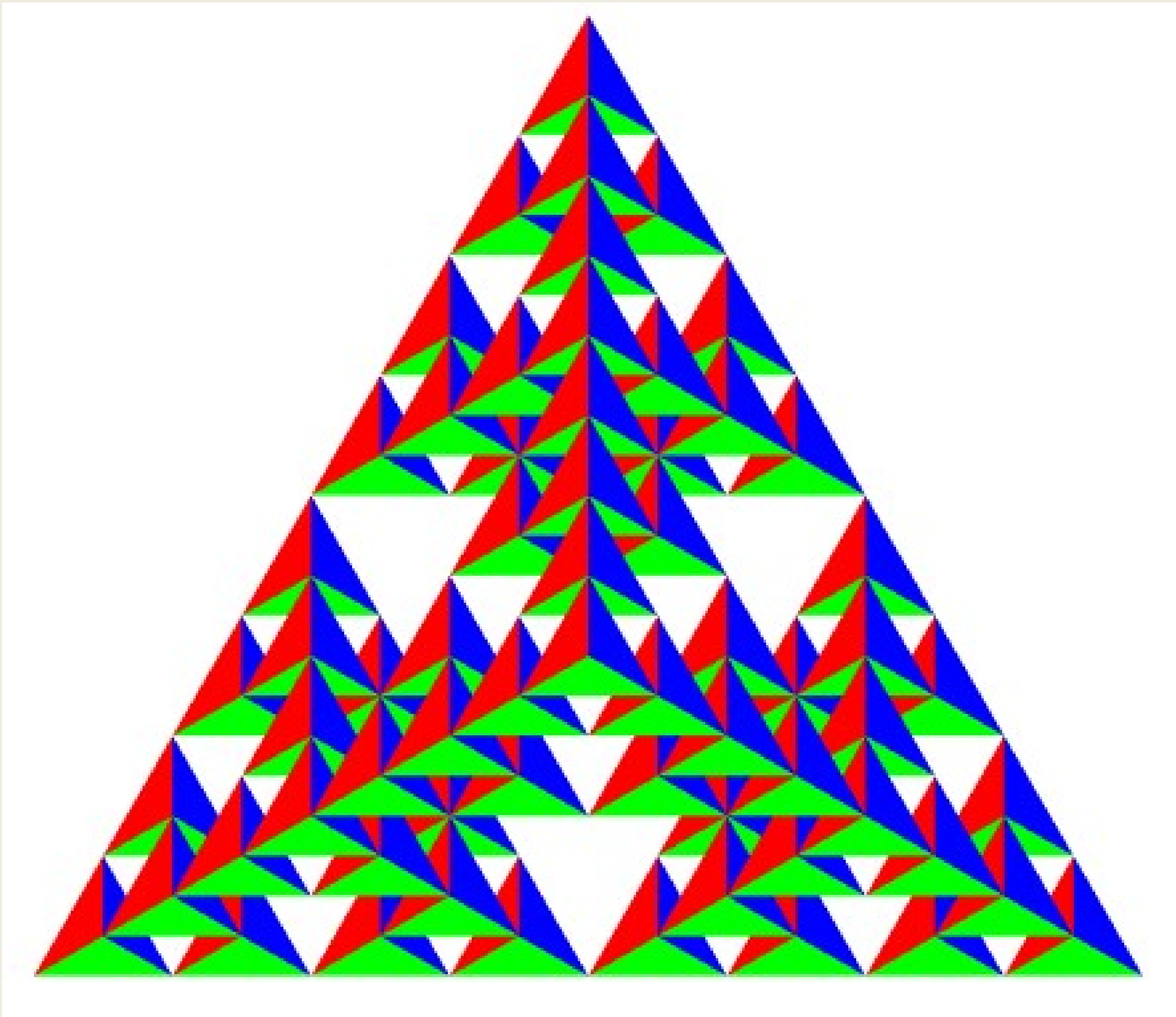
- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- Depth buffer is required to be available in WebGL

- 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 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(gasket4.html)



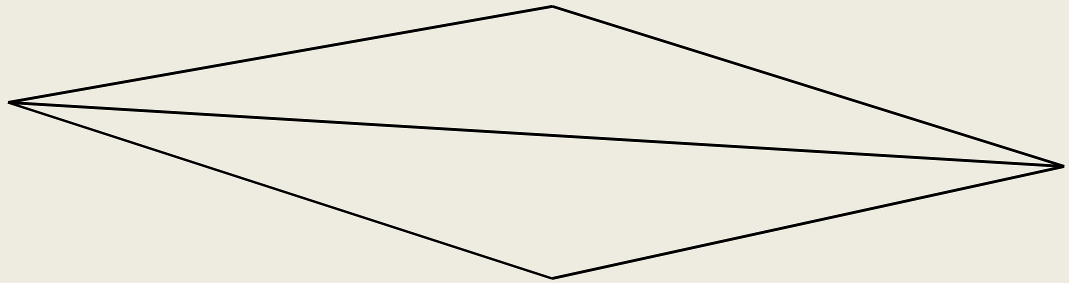
(Optional) Polygon + α

Polygon Testing

- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assumed both and left testing to the application
- Present version only renders triangles
- Need algorithm to triangulate an arbitrary polygon

Good and Bad Triangles

- Long thin triangles render badly



- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points

Recursive Division

- Find leftmost vertex and split