2. Graphics Programming

Programming with WebGL

- Part 1: Background
- Part 2: Complete Programs
- Part 3: Shaders
- Part 4: Color and Attributes
- Part 5: More GLSL
- Part 6: Three Dimensions
- Sample Programs

Programming with WebGL
Part 1: Background

Objectives

- Development of the OpenGL API
- OpenGL Architecture
  - OpenGL as a state machine
  - OpenGL as a data flow 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 IS0 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 GL

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

OpenGL

The success of GL lead to OpenGL (1992), a platform-independent API that was

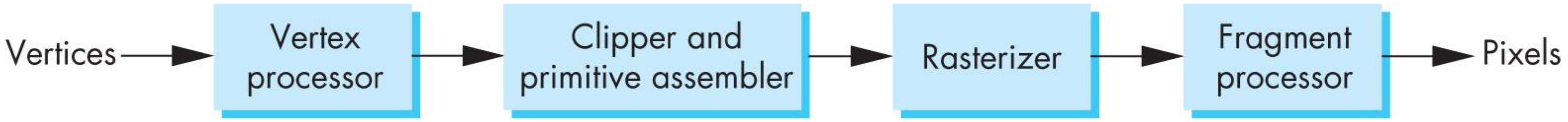
- Easy to use
- Close enough to the hardware to get excellent performance
- Focus on rendering
- Omitted windowing and input to avoid window system dependencies

OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,…….
  - Now Kronos Group
  - Was relatively stable (through version 2.5)
    - Backward compatible
    - Evolution reflected new hardware capabilities
      - 3D texture mapping and texture objects
      - Vertex and fragment programs
  - Allows platform specific features through extensions

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

OpenGL 3.1

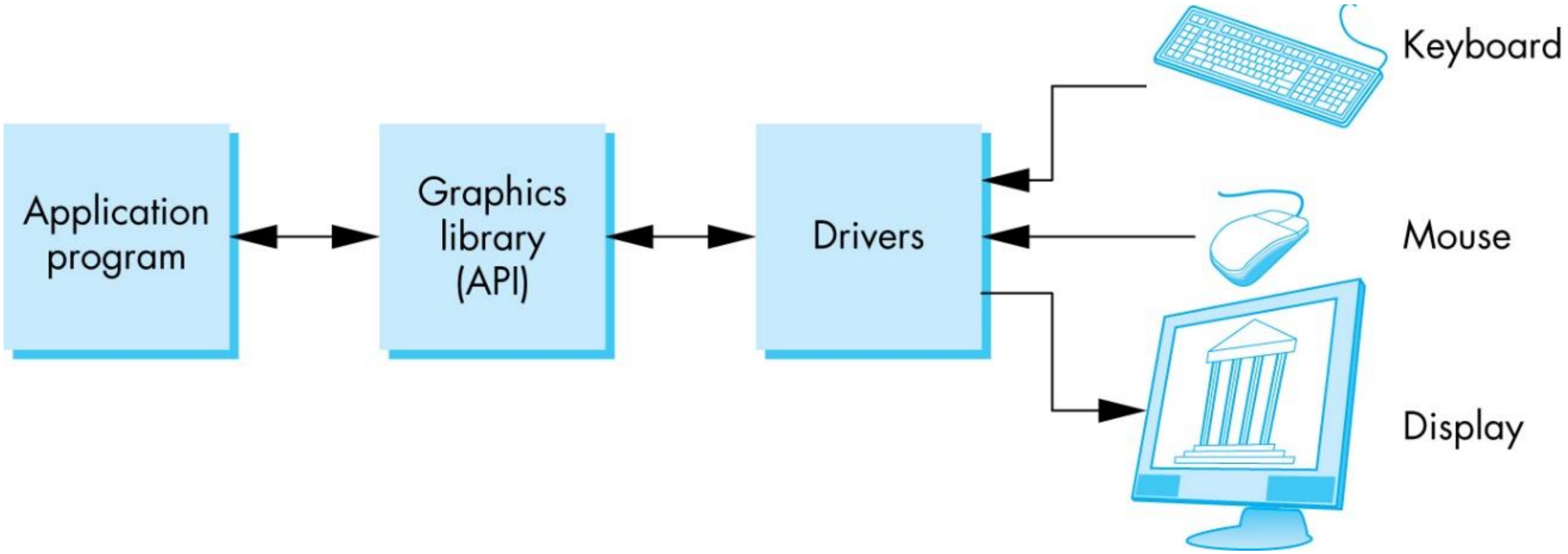
- Totally shader-based
  - No default shaders
  - Each application must provide both a vertex and a fragment shader
- No immediate mode
- Few state variables
- Most 2.5 functions deprecated
- Backward compatibility not required
  - Exists a compatibility extension

Other Versions

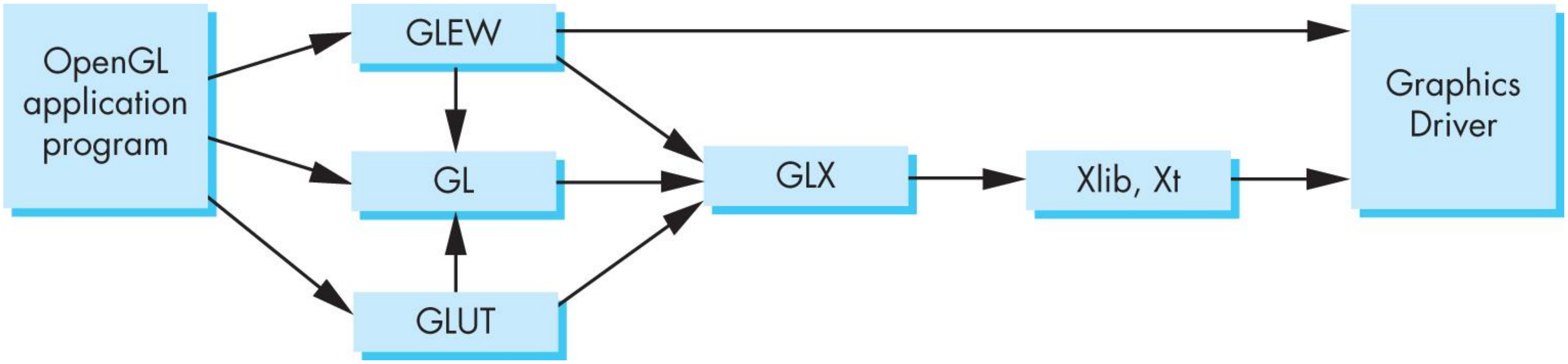
- OpenGL ES
  - Embedded systems
  - Version 1.0 simplified OpenGL 2.1
  - Version 2.0 simplified OpenGL 3.1
    - Shader based
- WebGL
  - Javascript implementation of ES 2.0
  - Supported on newer browsers
- OpenGL 4.1, 4.2, …..
  - Add geometry, tessellation, compute shaders

Programming with WebGL
Part 1: Background

OpenGL Architecture

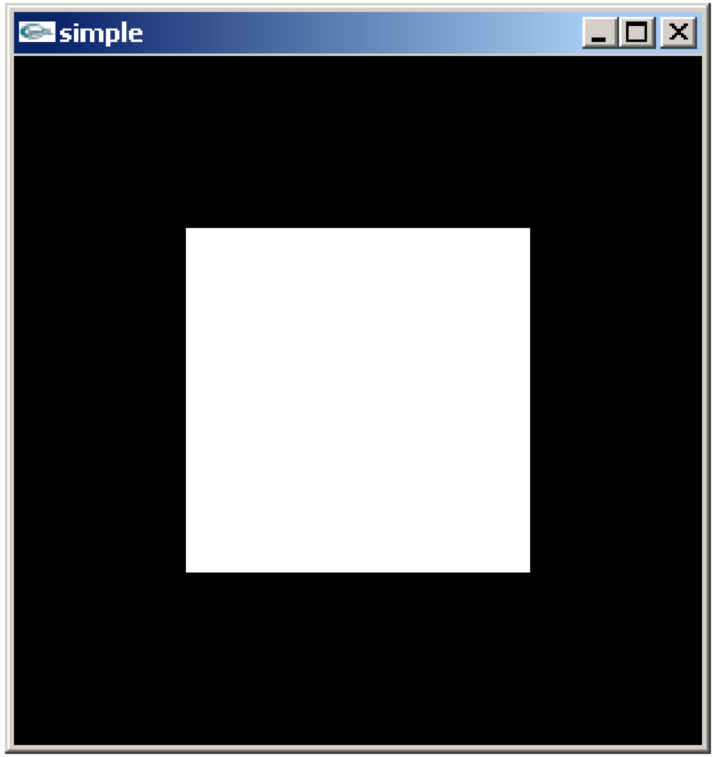


Software Organization



A OpenGL Simple Program

Generate a square on a solid background



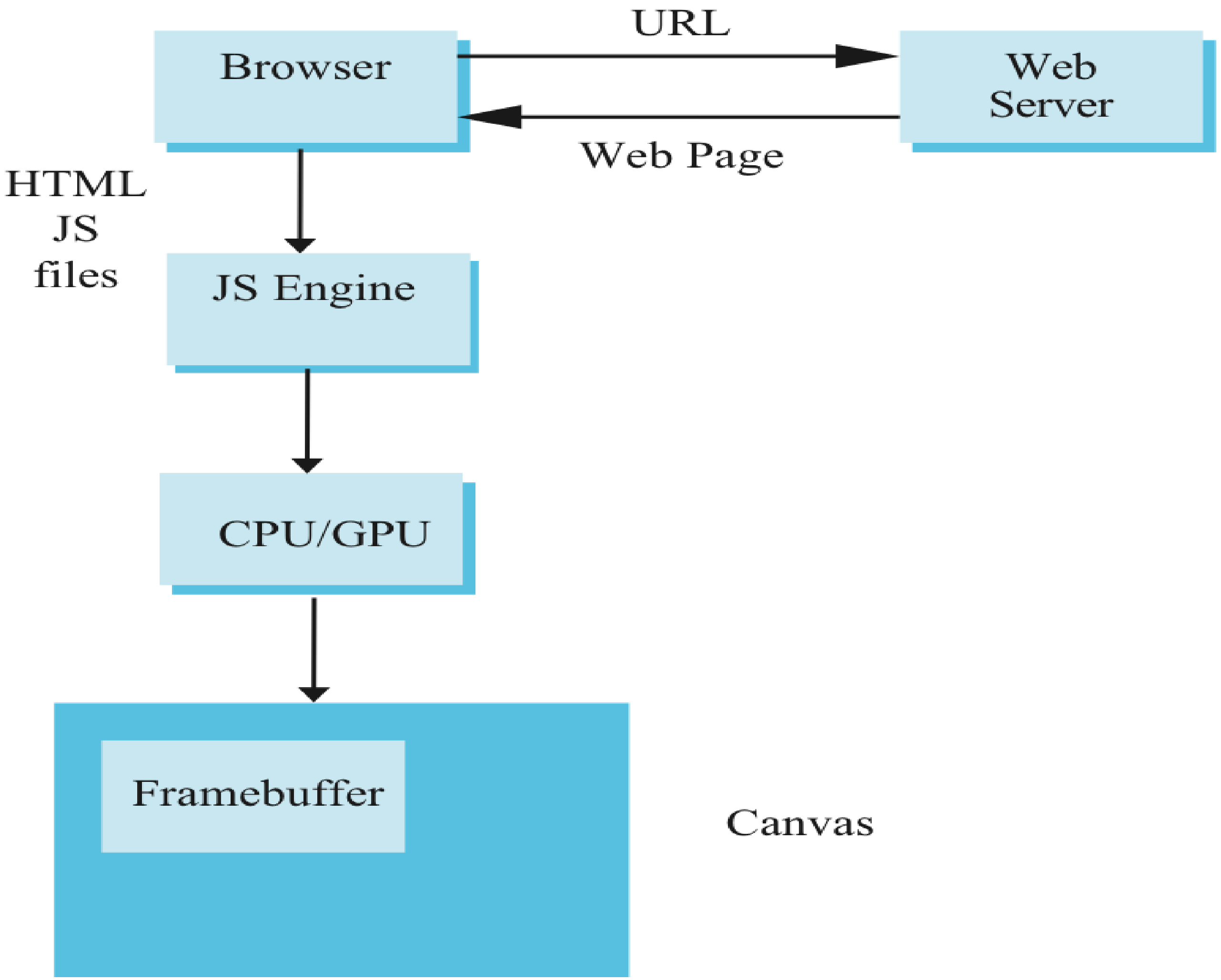
It used to be easy

```
#include <GL/glut.h>
void mydisplay(){
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_QUAD);
                glVertex2f(-0.5, -0.5);
                glVertex2f(-0,5,  0,5);
                glVertex2f( 0.5,  0.5);
                glVertex2f( 0.5, -0.5);
        glEnd()
}
int main(int argc, char** argv){
        glutCreateWindow("simple");
        glutDisplayFunc(mydisplay);
        glutMainLoop();
}
```

What happened?

- Most OpenGL functions deprecated
  - immediate vs retained mode
  - make use of GPU
- Makes heavy use of state variable default values that no longer exist
  - Viewing
  - Colors
  - Window parameters
- However, processing loop is the same

Execution in Browser



Event Loop

- Remember that the sample program specifies a render function which is an *event listener* or *callback* function
  - Every program should have a render callback
  - For a static application we need only execute the render function once
  - In a dynamic application, the render function can call itself recursively but each redrawing of the display must be triggered by an event

Lack of Object Orientation

- All versions of OpenGL are not object oriented so that there are multiple functions for a given logical function
- Example: sending values to shaders
  - `gl.uniform3f`
  - `gl.uniform2i`
  - `gl.uniform3dv`
- Underlying storage mode is the same

WebGL function format

function name

dimension

```
gl.uniform3f(x,y,z)
```

belongs to WebGL canvas

x,y,z are variables

```
gl.uniform3fv(p)
```

p is an array

WebGL constants

- Most constants are defined in the canvas object
  - In desktop OpenGL, they were in #include files such as **`gl.h`**
- Examples
  - **`desktop OpenGL`**
    - **`glEnable(GL_DEPTH_TEST);`**
  - **`WebGL`**
    - **`gl.enable(gl.DEPTH_TEST)`**
  - **`gl.clear(gl.COLOR_BUFFER_BIT)`**

WebGL and GLSL

- WebGL requires shaders and is based less on a state machine model than a data flow model
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job of application is to get data to GPU

GLSL

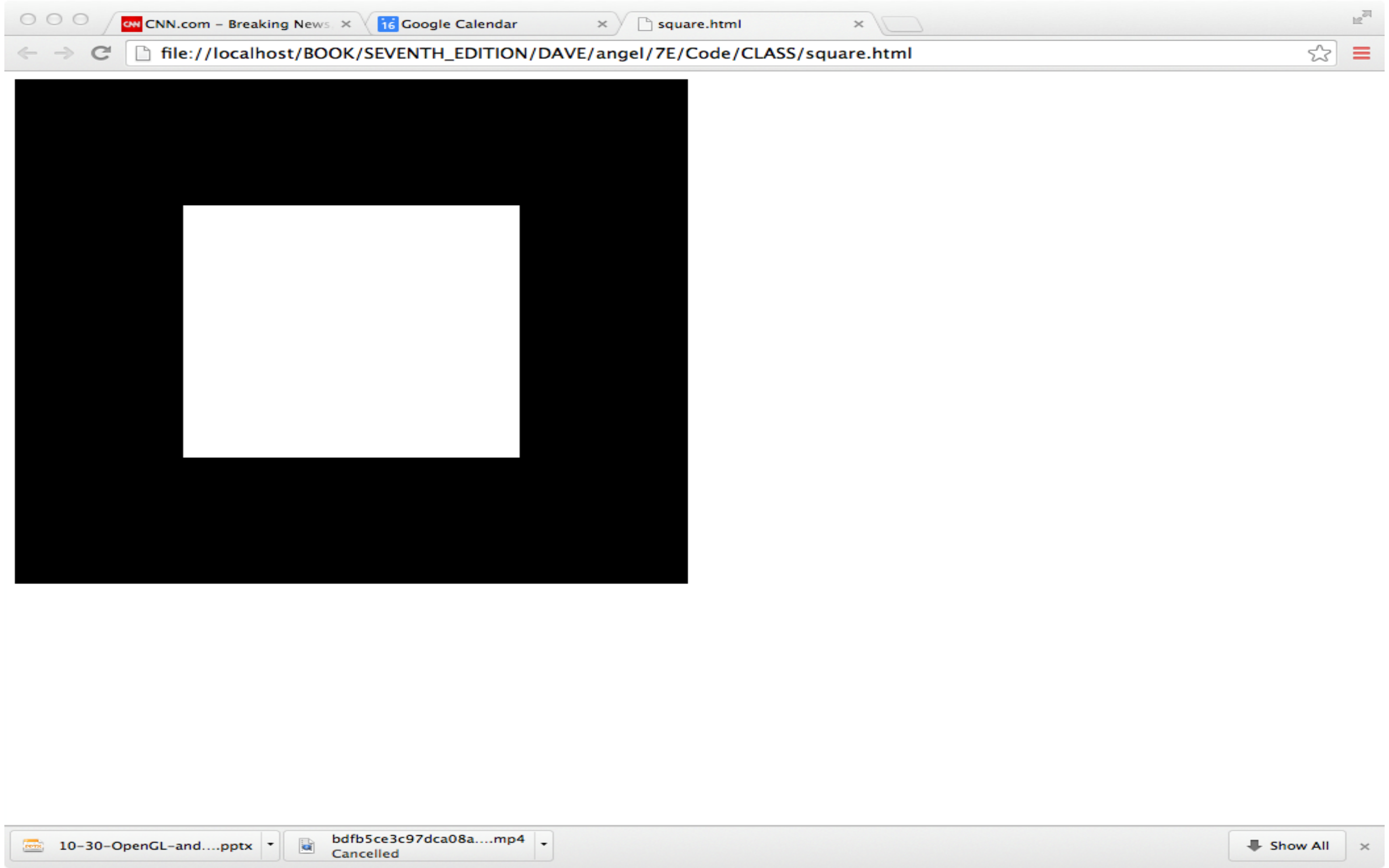
- OpenGL Shading Language
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Similar to Nvidia's Cg and Microsoft HLSL
- Code sent to shaders as source code
- WebGL functions compile, link and get information to shaders

Programming with OpenGL
Part 2: Complete Programs

Objectives

- Build a complete first program
  - Introduce shaders
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Initialization steps and program structure

Square Program



WebGL

- Five steps
  - Describe page (HTML file)
    - request WebGL Canvas
    - read in necessary files
  - Define shaders (HTML file)
    - could be done with a separate file (browser dependent)
  - Compute or specify data (JS file)
  - Send data to GPU (JS file)
  - Render data (JS file)

square.html

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">

attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>

<script id="fragment-shader" type="x-shader/x-fragment">

precision mediump float;

void main()
{
    gl_FragColor = vec4( 1.0, 1.0, 1.0, 1.0 );
}
</script>
```

Shaders

- We assign names to the shaders that we can use in the JS file
- These are trivial pass-through (do nothing) shaders that which set the two required built-in variables
  - gl_Position
  - gl_FragColor
- Note both shaders are full programs
- Note vector type vec2
- Must set precision in fragment shader

square.html (cont)

```
<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="square.js"></script>
</head>

<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

Files

- **`../Common/webgl-utils.js`**: Standard utilities for setting up WebGL context in Common directory on website
- **`../Common/initShaders.js`**: contains JS and WebGL code for reading, compiling and linking the shaders
- **`../Common/MV.js`**: our matrix-vector package
- **`square.js`**: the application file

square.js

```
var gl;
var points;

window.onload = function init(){
    var canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }

    // Four Vertices

    var vertices = [
        vec2( -0.5, -0.5 ),
        vec2( -0.5,  0.5 ),
        vec2(  0.5,  0.5 ),
        vec2(  0.5, -0.5 )
    ];
```

Notes

- **`onload`**: determines where to start execution when all code is loaded
- canvas gets WebGL context from HTML file
- vertices use vec2 type in MV.js
- JS array is not the same as a C or Java array
  - object with methods
  - vertices.length // 4
- Values in clip coordinates

square.js (cont)

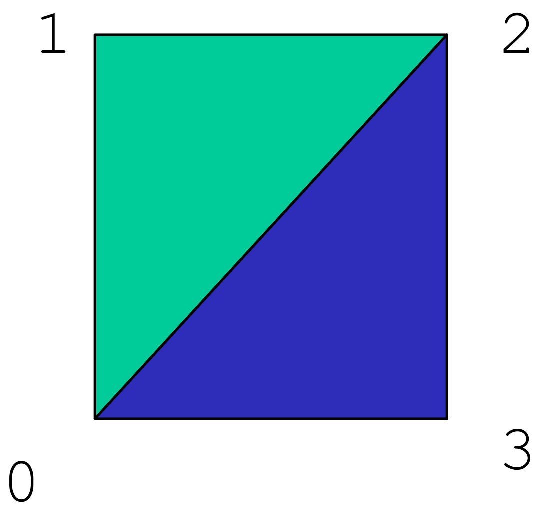
```
//  Configure WebGL

gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 0.0, 0.0, 0.0, 1.0 );

 //  Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

 // Load the data into the GPU

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

 // Associate out shader variables with our data buffer

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

Notes

- `initShaders` used to load, compile and link shaders to form a program object
- Load data onto GPU by creating a **vertex buffer object** on the GPU
  - Note use of flatten() to convert JS array to an array of float32's
- Finally we must connect variable in program with variable in shader
  - need name, type, location in buffer

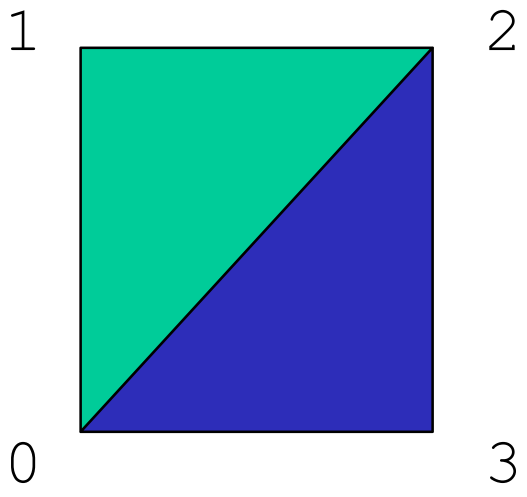
square.js (cont)

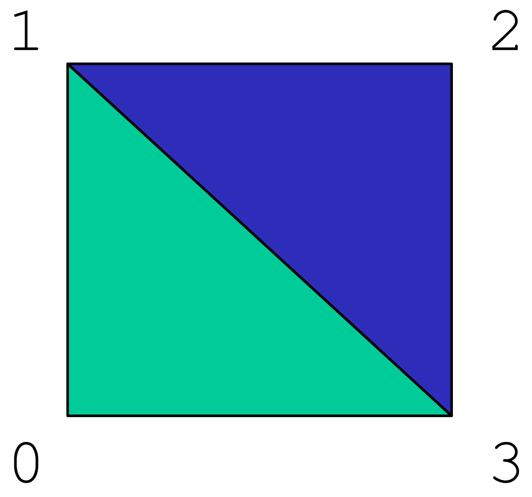
```
    render();
};

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



Triangles, Fans or Strips

```
gl.drawArrays( gl.TRIANGLES, 0, 6 ); // 0, 1, 2, 0, 2, 3

gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 ); // 0, 1 , 2, 3
```






```
gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 ); // 0, 1, 3, 2
```

Programming with OpenGL
Part 2: Complete Programs

Objectives

- Build a complete first program
  - Introduce shaders
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Initialization steps and program structure

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

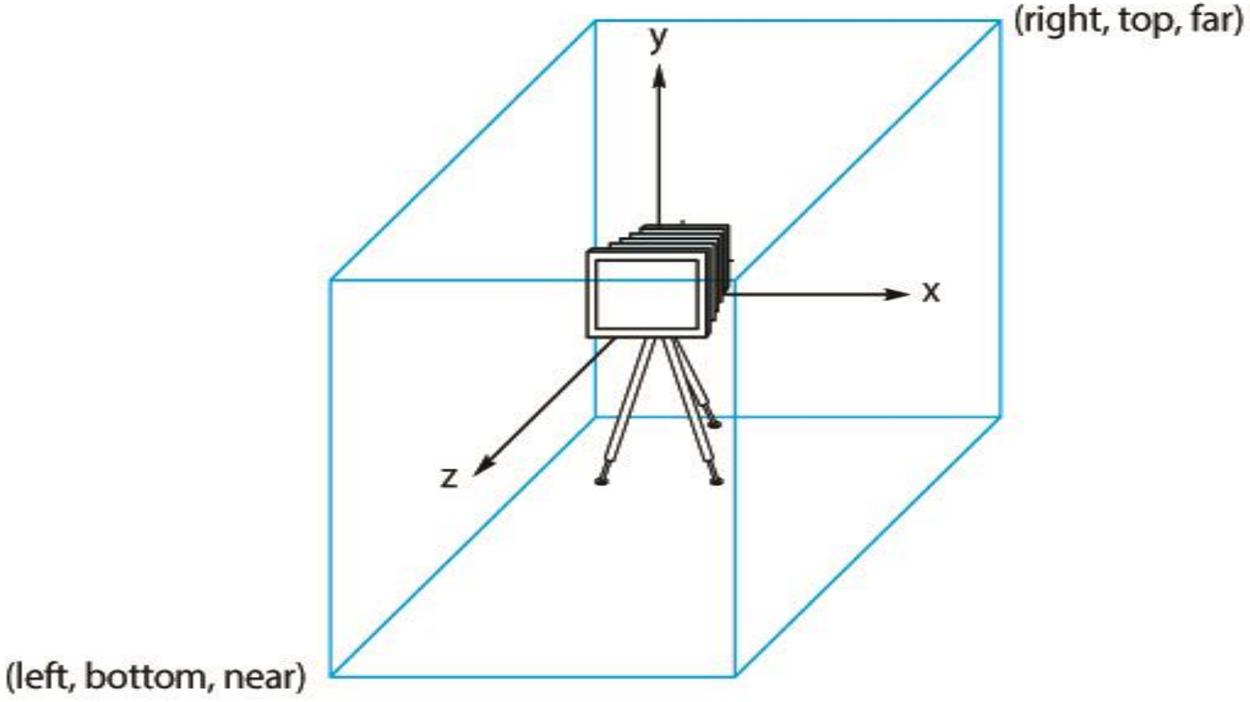
- The units in **`points`** are determined by the application and are called *object, world, model* or *problem coordinates*
- Viewing specifications usually are also in object coordinates
- Eventually pixels will be produced in *window coordinates*
- WebGL also uses some internal representations that 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

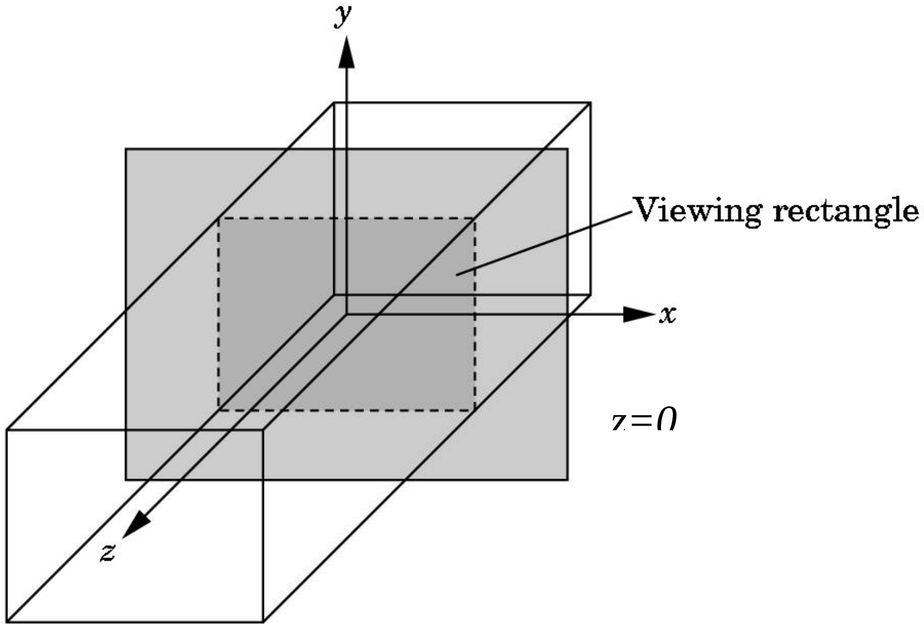
WebGL Camera

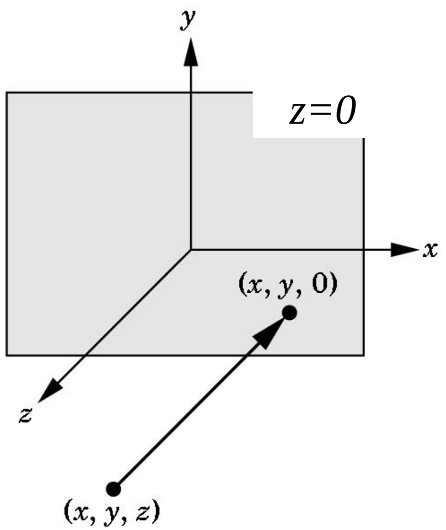
- WebGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with sides of length 2



Orthographic Viewing

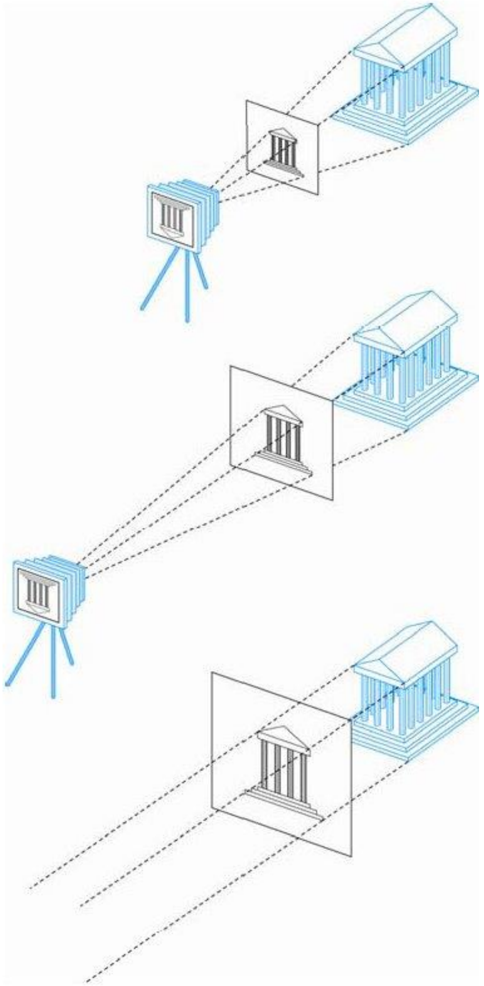
In the default orthographic view, points are projected forward along the z axis onto the plane $z=0$





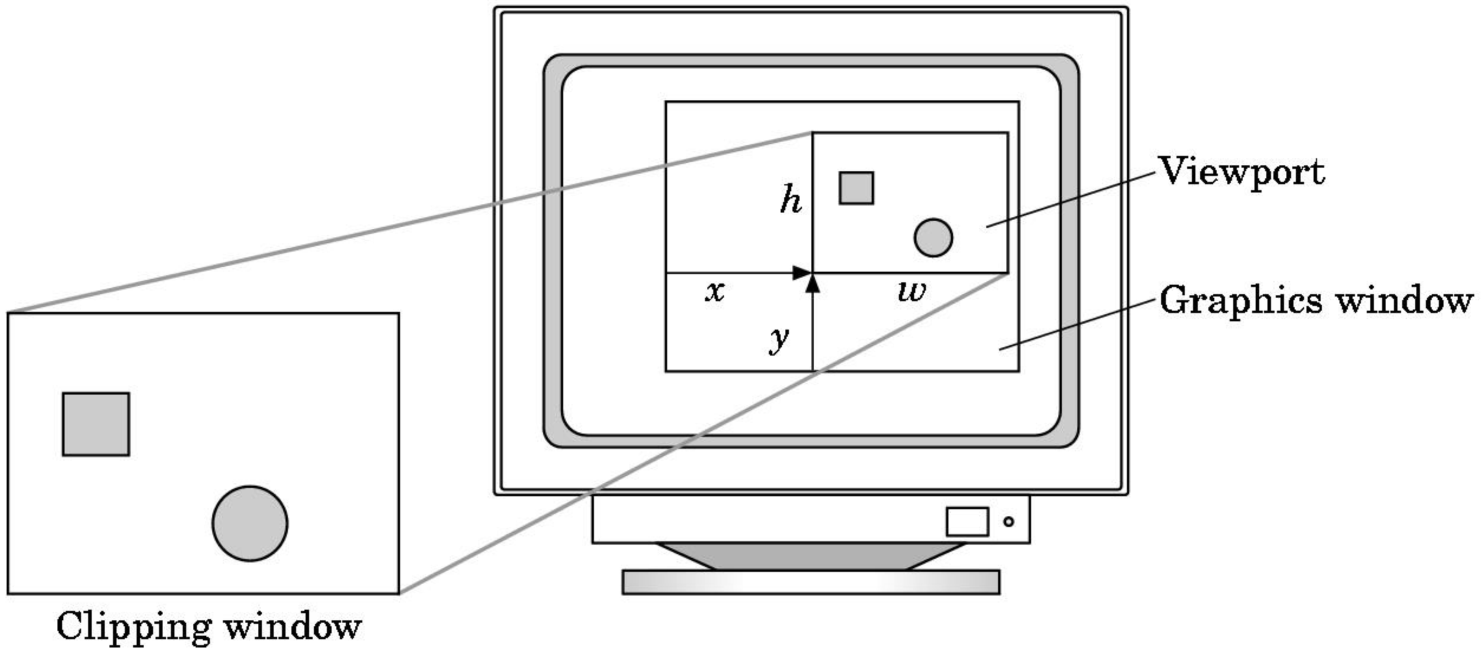
Orthographic Viewing

Imagine a camera infinitely far away from image plane

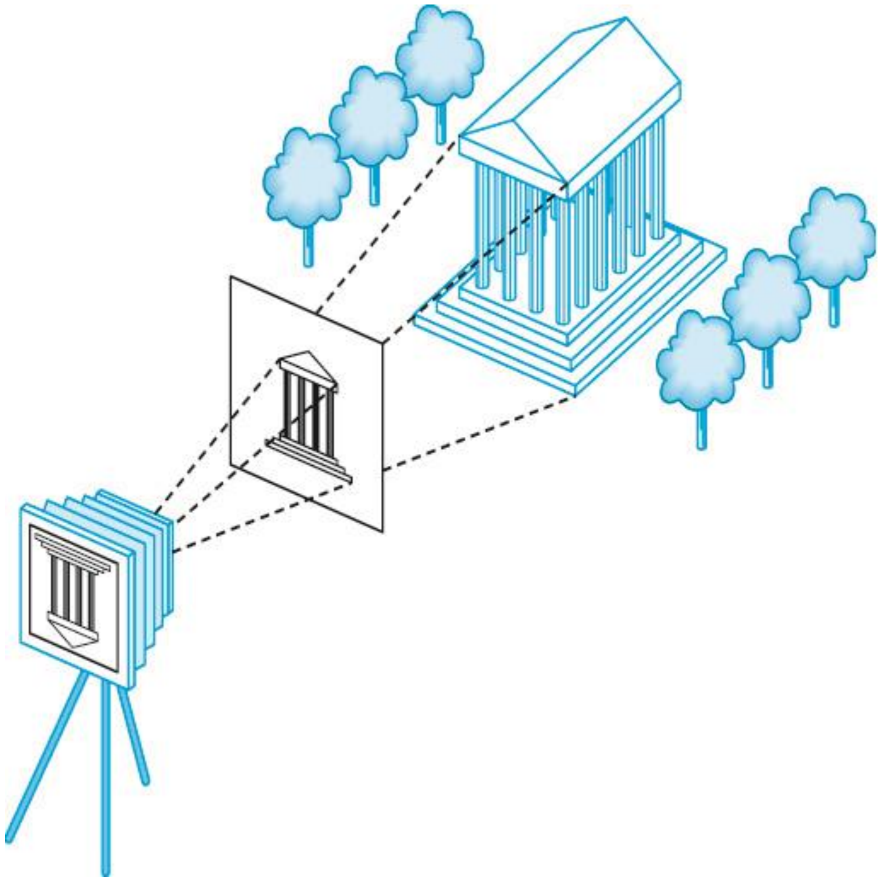


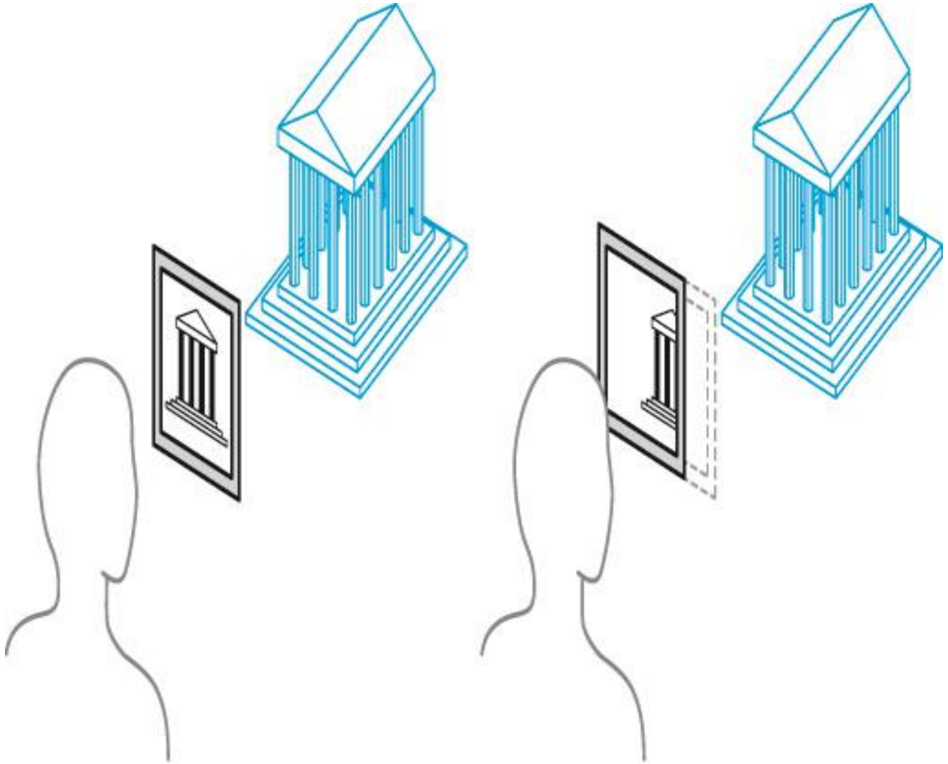
Viewports

- Do not have use the entire window for the image: **`gl.viewport(x,y,w,h)`**
- Values in pixels (window coordinates)

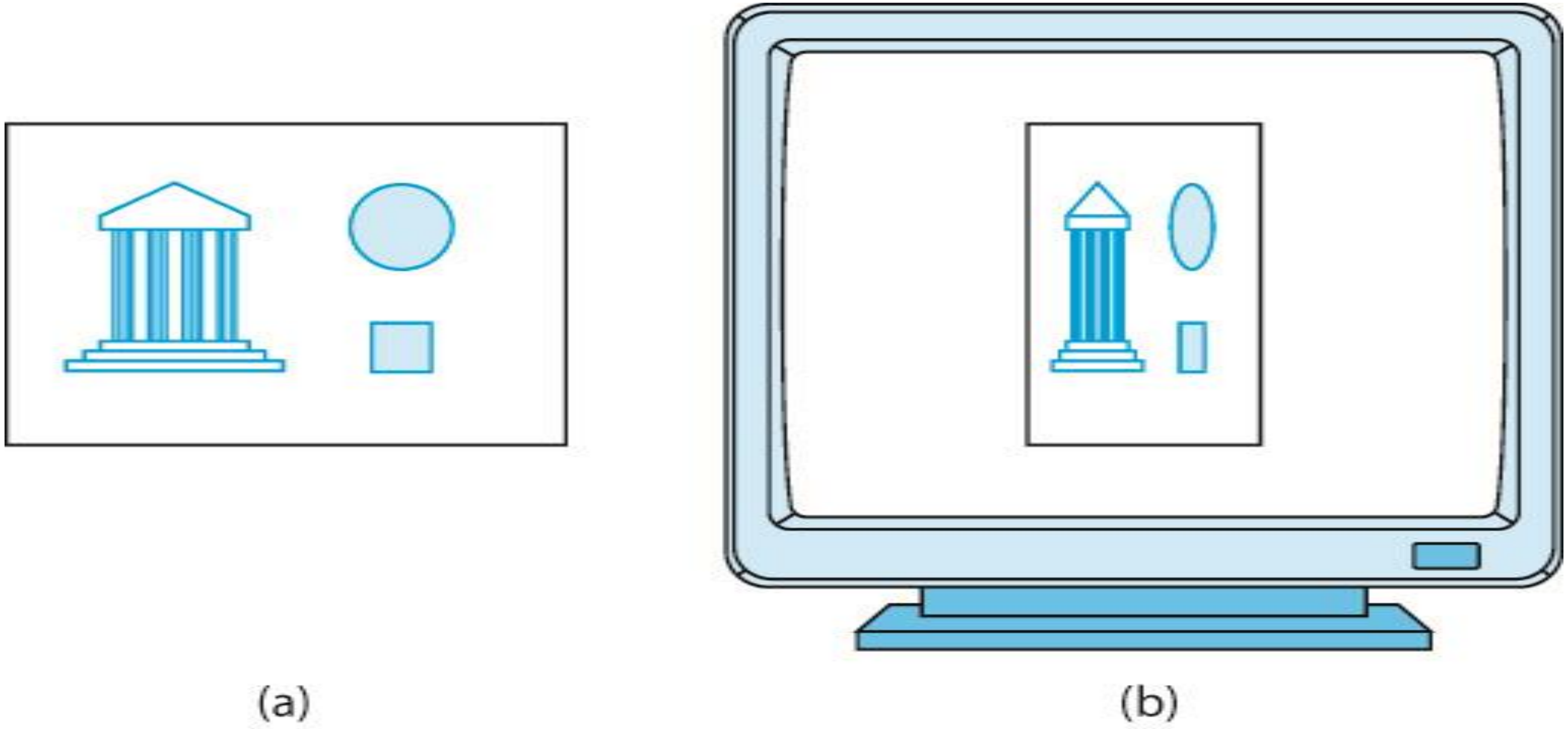


Clipping



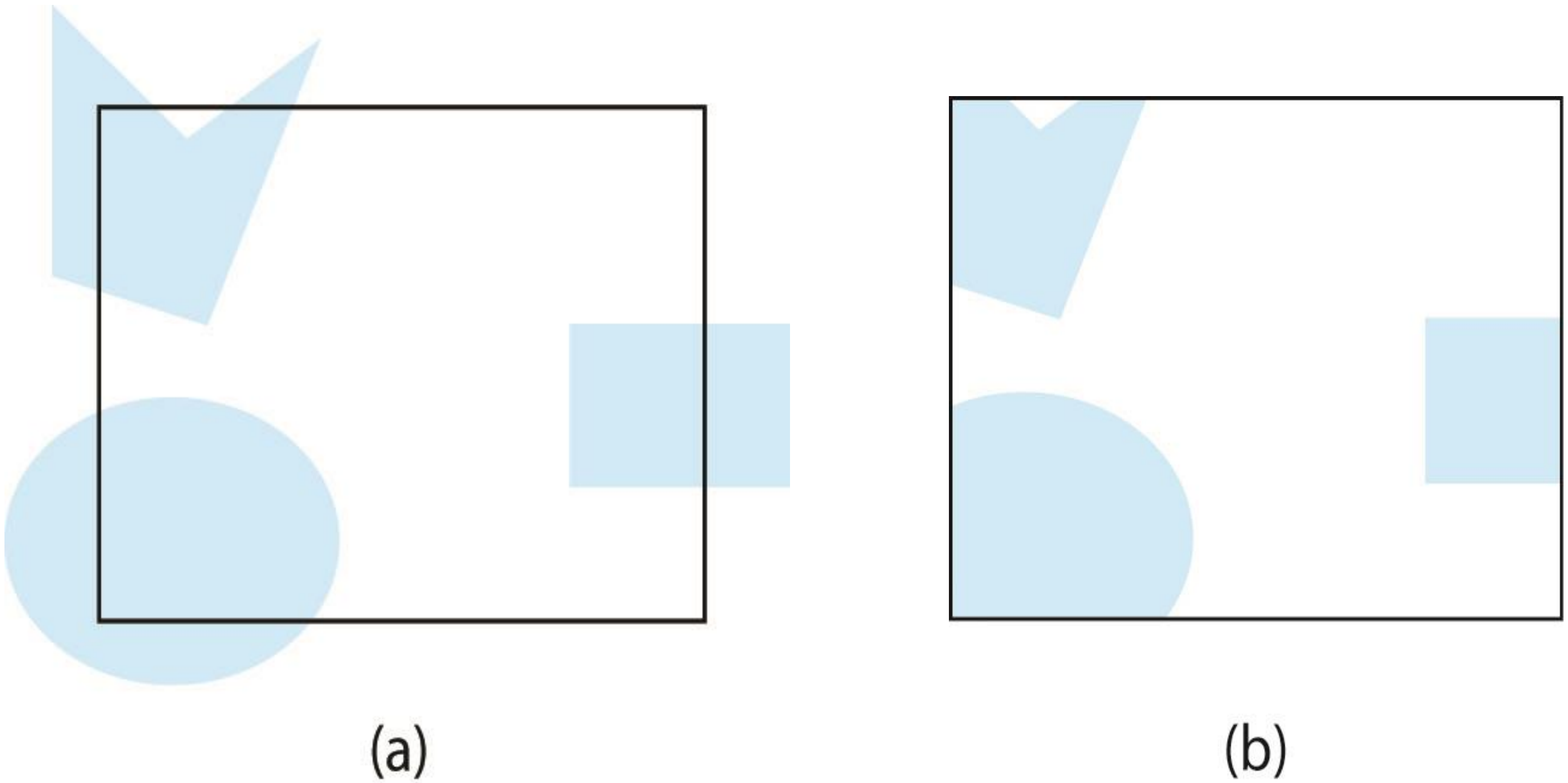


Aspect-Ratio Mismatch



Clipping window Display window

Two-dimensional Viewing



Objects before clipping Image after clipping

Transformations and Viewing

- In WebGL, we usually carry out projection using a projection matrix (transformation) before rasterization
- Transformation functions are also used for changes in coordinate systems
- Pre 3.1 OpenGL had a set of transformation functions which have been deprecated
- Three choices in WebGL
  - Application code
  - GLSL functions
  - MV.js

First Assignment: Tessellation and Twist

- Consider rotating a 2D point about the origin

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

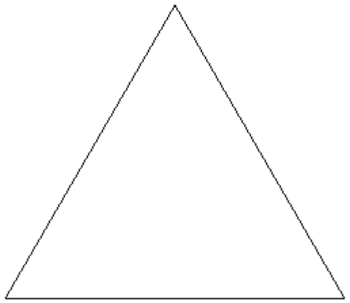
- Now let amount of rotation depend on distance from origin giving us **twist**

$$x' = x\cos(d\theta) - y\sin(d\theta)$$

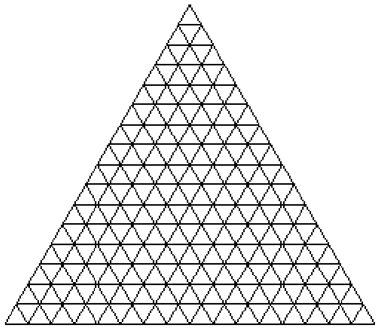
$$y' = x\sin(d\theta) + y\cos(d\theta)$$

$$d \propto \sqrt{x^2 + y^2}$$

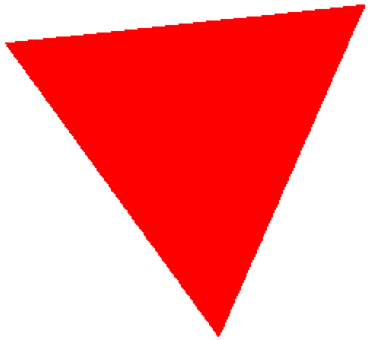
Example



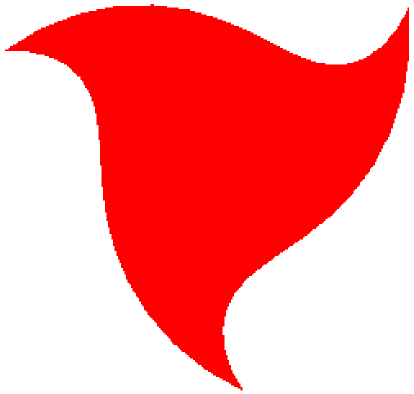
triangle



tessellated triangle



twist without tessellation



twist after tessellation

Programming with WebGL
Part 3: Shaders

Objectives

- Simple Shaders
  - Vertex shader
  - Fragment shaders
- Programming shaders with GLSL
- Finish first program

Vertex Shader Applications

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

Fragment Shader Applications

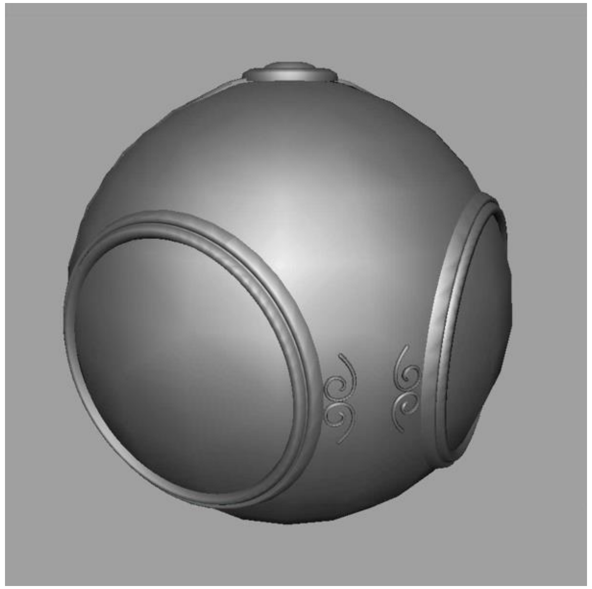
Per fragment lighting calculations

per vertex lighting

per fragment lighting

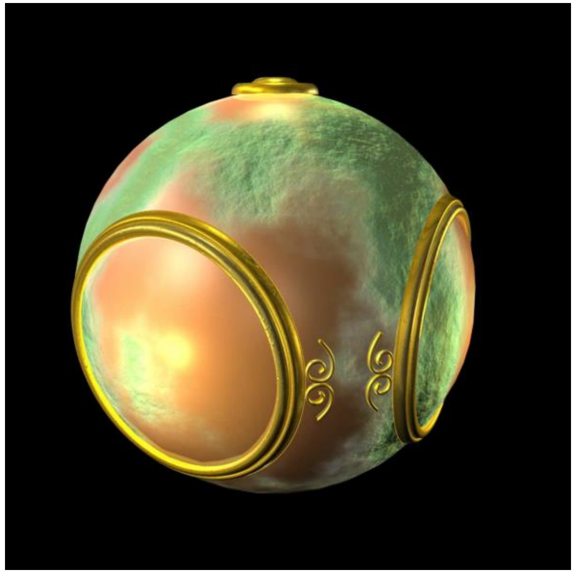
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping



bump mapping

Writing Shaders

- First programmable shaders were programmed in an assembly-like manner
- OpenGL extensions added functions for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
- OpenGL Shading Language (GLSL)

GLSL

- OpenGL Shading Language
- Part of OpenGL 2.0 and up
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- As of OpenGL 3.1, application must provide shaders

Simple Vertex Shader

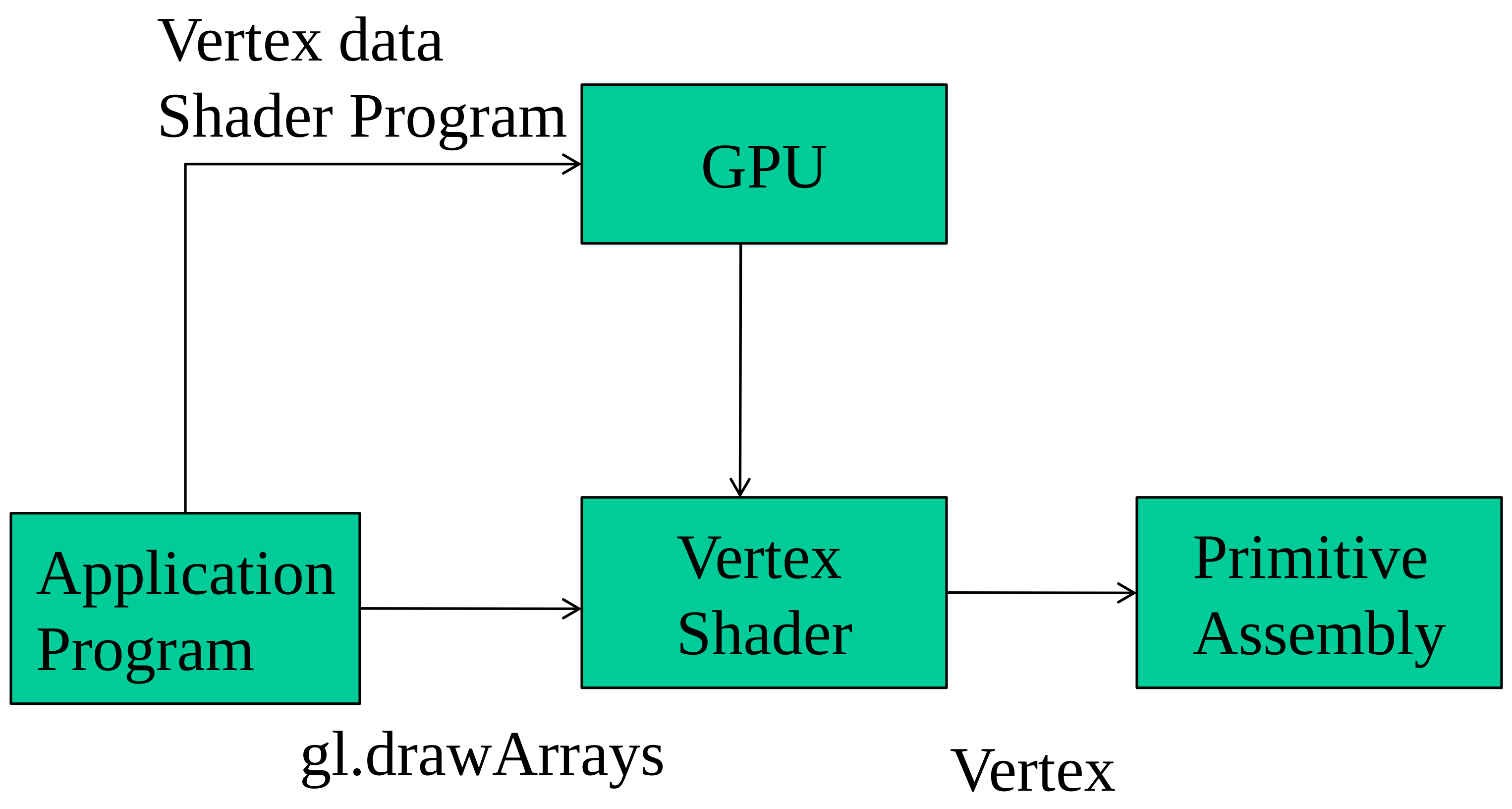
input from application

```
attribute vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

must link to variable in application

built in variable

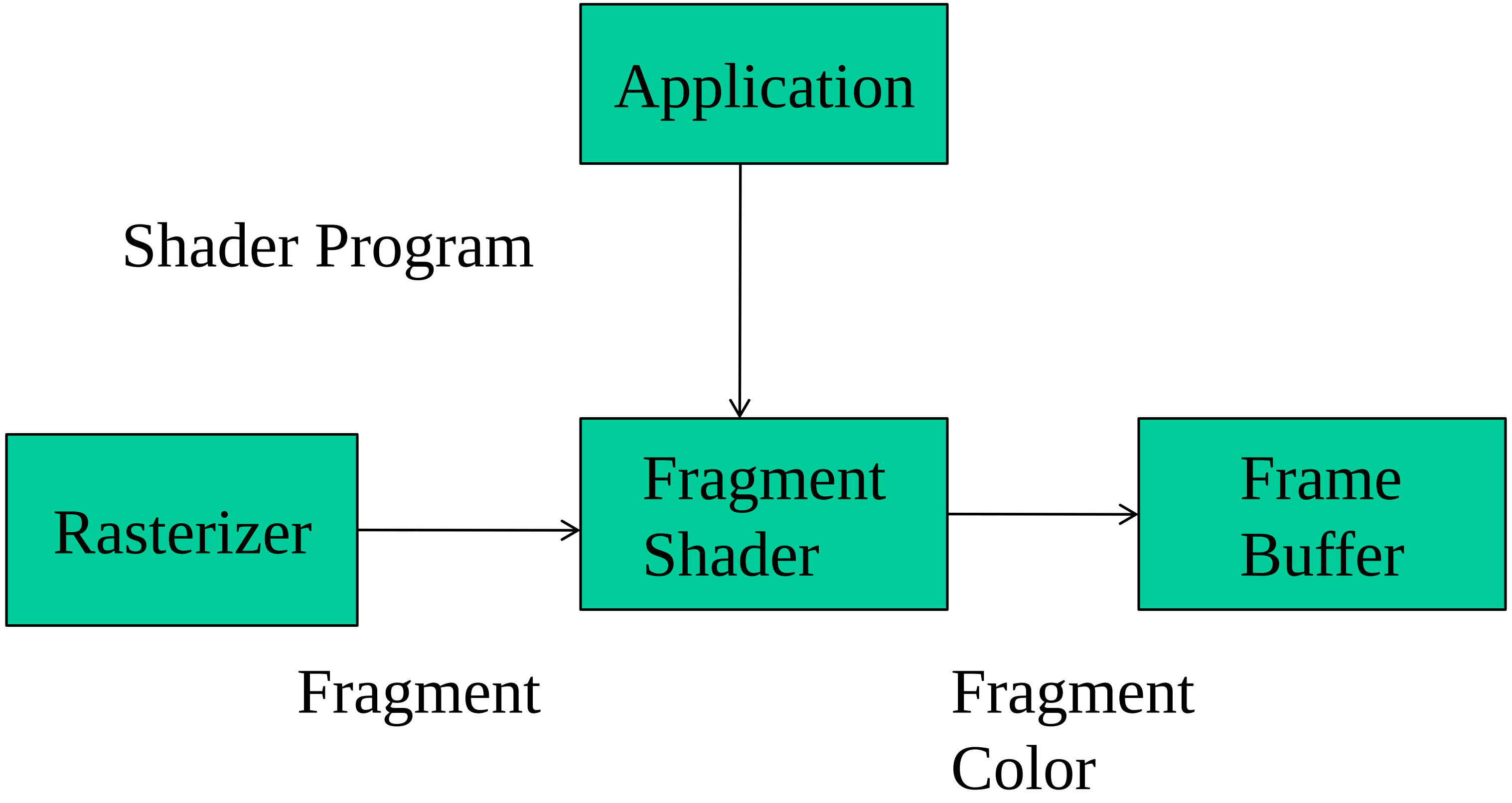
Execution Model



Simple Fragment Program

```
precision mediump float;
void main(void)
{
   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Execution Model



Programming with WebGL
Part 3: Shaders

Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a =vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

No Pointers

- There are no pointers in GLSL
- We can use C structs which

  can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

  mat3 func(mat3 a)
- variables passed by copying

Qualifiers

- GLSL has many of the same qualifiers such as `const` as C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as gl_Position but most have been deprecated
- User defined (in application program)
  - `attribute float temperature`
  - `attribute vec3 velocity`
  - recent versions of GLSL use **`in`** and **`out`** qualifiers to get to and from shaders

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the time or a bounding box of a primitive or transformation matrices

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- With WebGL, GLSL uses the varying qualifier in both shaders

```
varying vec4 color;
```

- More recent versions of WebGL use **out** in vertex shader and **in** in the fragment shader

```
out vec4 color; //vertex shader
in vec4 color;  // fragment shader
```

Our Naming Convention

- attributes passed to vertex shader have names beginning with v (vPosition, vColor) in both the application and the shader
  - Note that these are different entities with the same name
- Variable variables begin with f (fColor) in both shaders
  - must have same name
- Uniform variables are unadorned and can have the same name in application and shaders

Example: Vertex Shader

```
attribute vec4 vColor;
varying vec4 fColor;
void main()
{
  gl_Position = vPosition;
  fColor = vColor;
}
```

Corresponding Fragment Shader

```
precision mediump float;

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

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

Sending a Uniform Variable

```
// in application

vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation( program, ”color" );
gl.uniform4f( colorLoc, color);

// in fragment shader (similar in vertex shader)

uniform vec4 color;

void main()
{
   gl_FragColor = color;
}
```

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - `a[2], a.b, a.z, a.p` are the same
- **Swizzling** operator lets us manipulate components

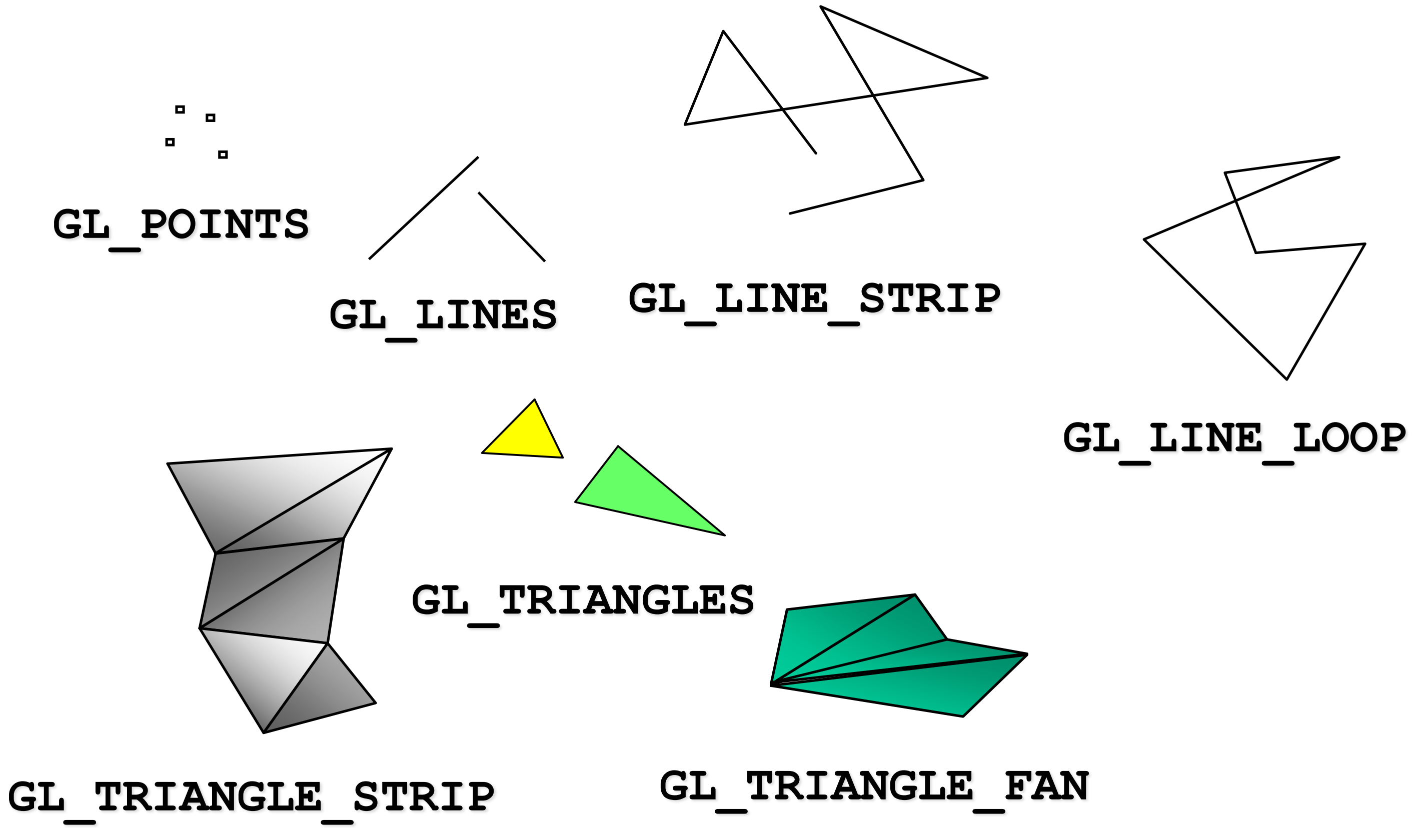
```
vec4 a, b;
a.yz = vec2(1.0, 2.0, 3.0, 4.0);
b = a.yxzw;
```

Programming with WebGL
Part 4: Color and Attributes

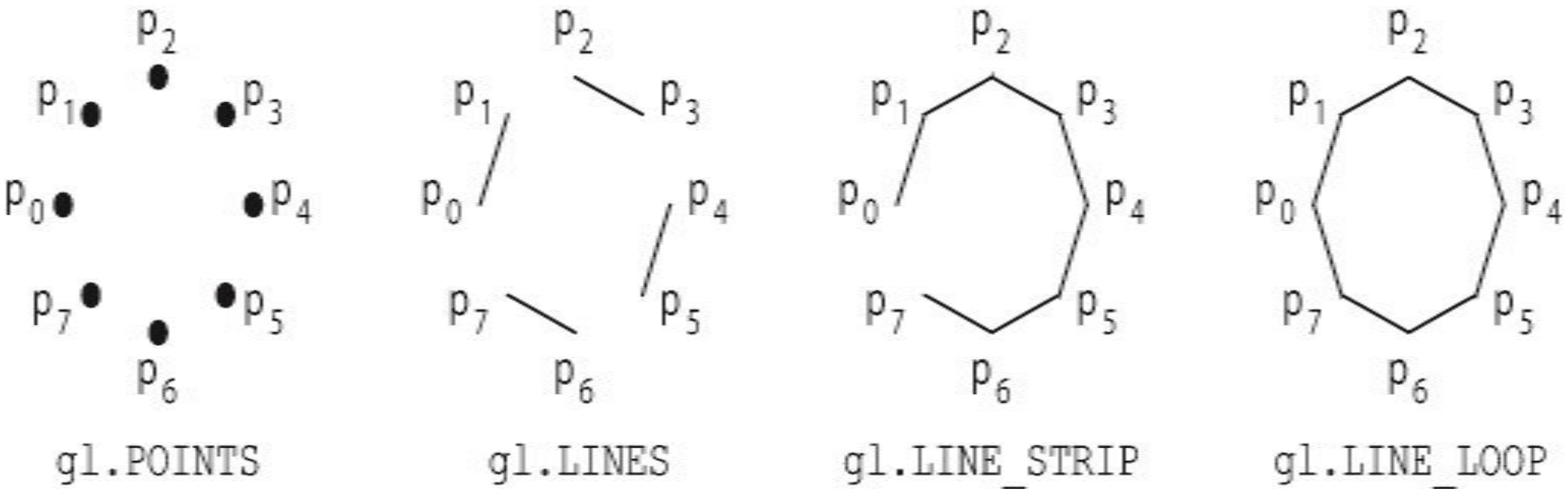
Objectives

- Expanding primitive set
- Adding color
- Vertex attributes

WebGL Primitives

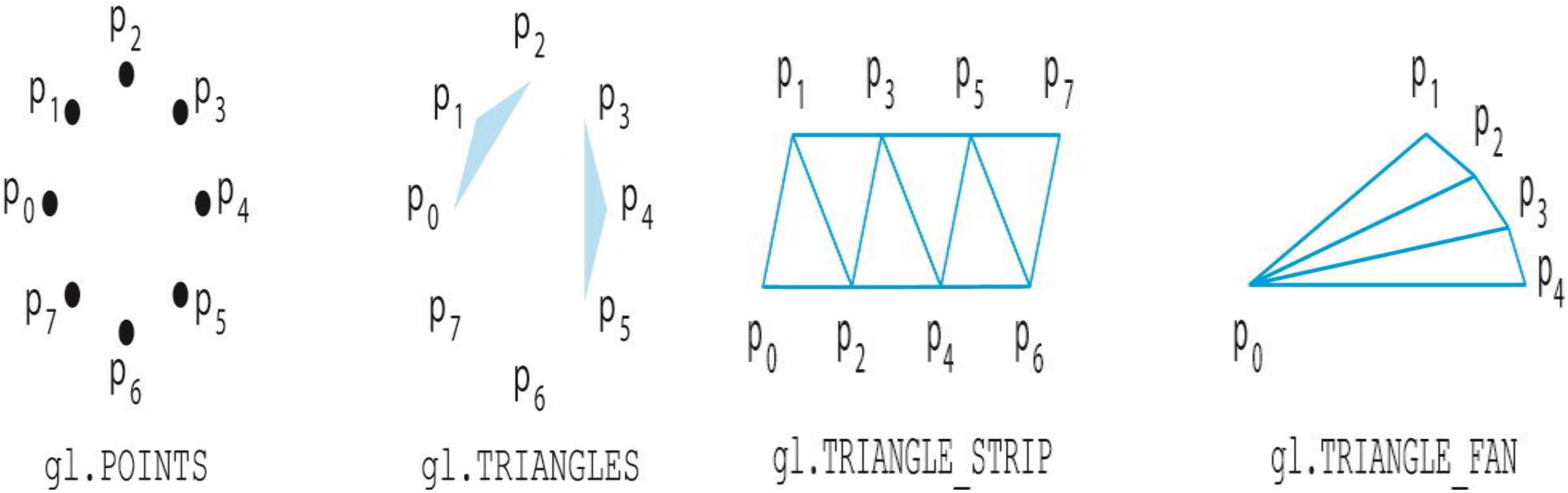


WebGL Primitives



Point and line-segment types

WebGL Primitives

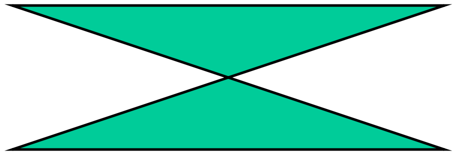


Point and triangle types

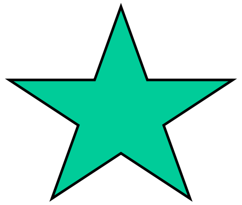
Triangle stripe and triangle fan

Polygon Issues

- WebGL will only display triangles
  - 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
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator but not WebGL



nonsimple polygon



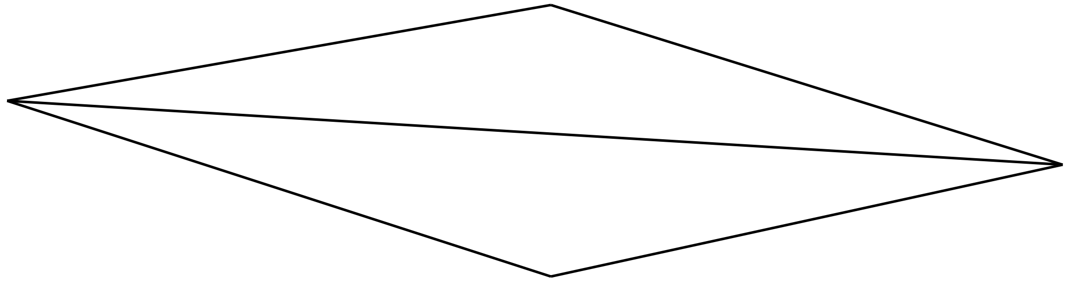
nonconvex polygon

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

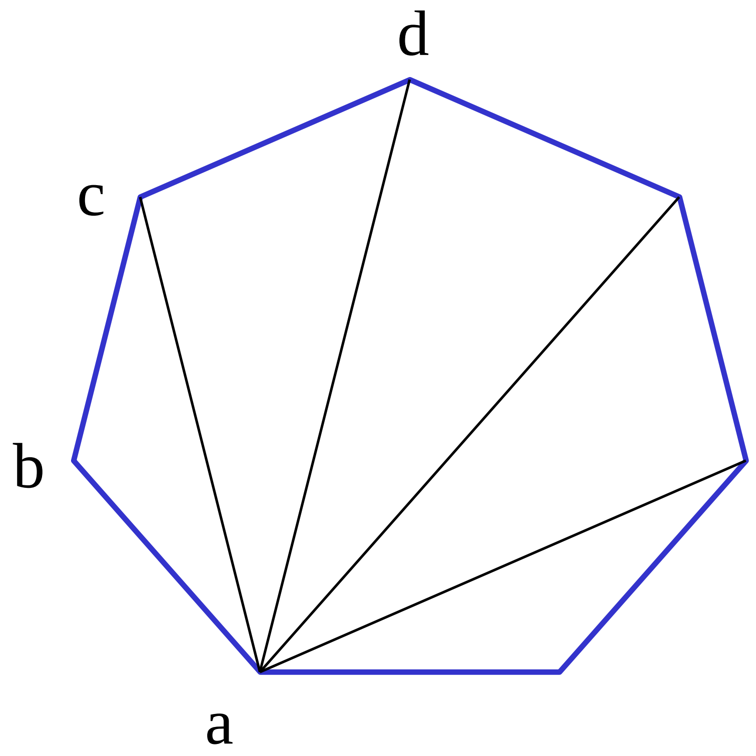
- Long thin triangles render badly



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

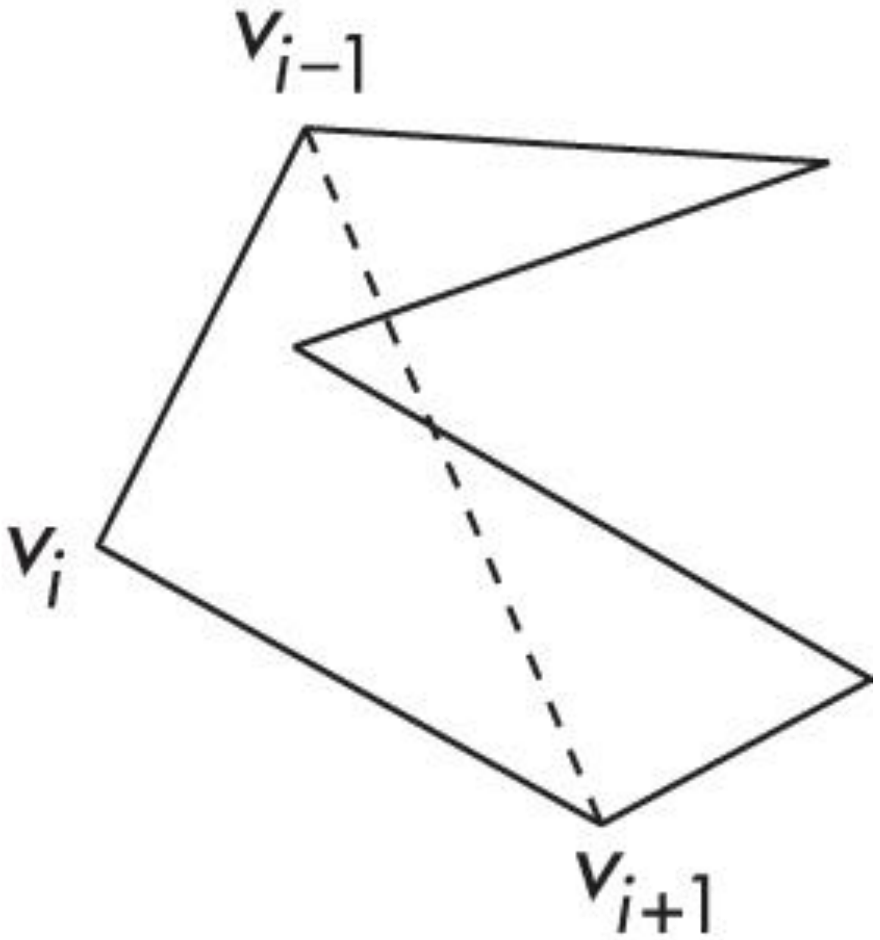
Triangularization

- Convex polygon



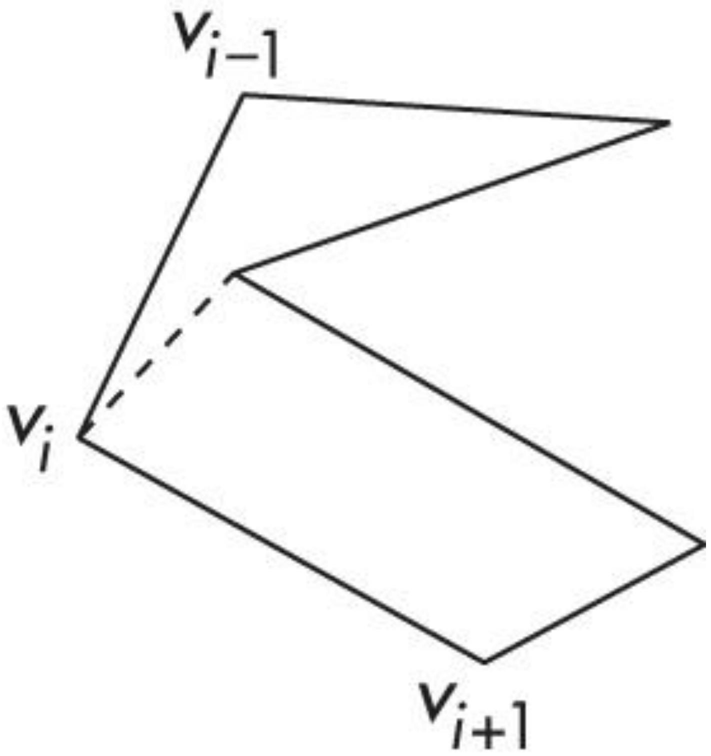
- Start with abc, remove b, then acd, ….

Non-convex (concave)



Recursive Division

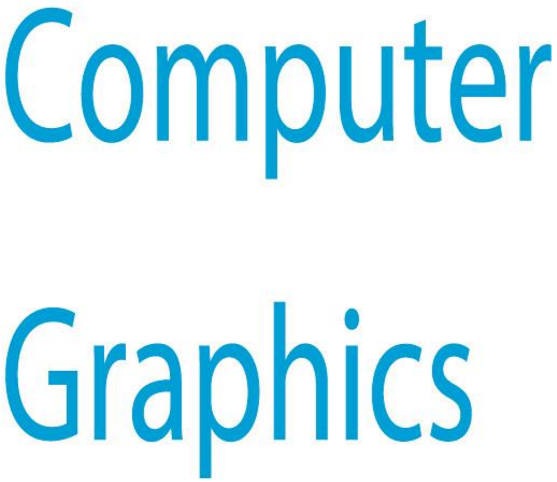
- Find leftmost vertex and split



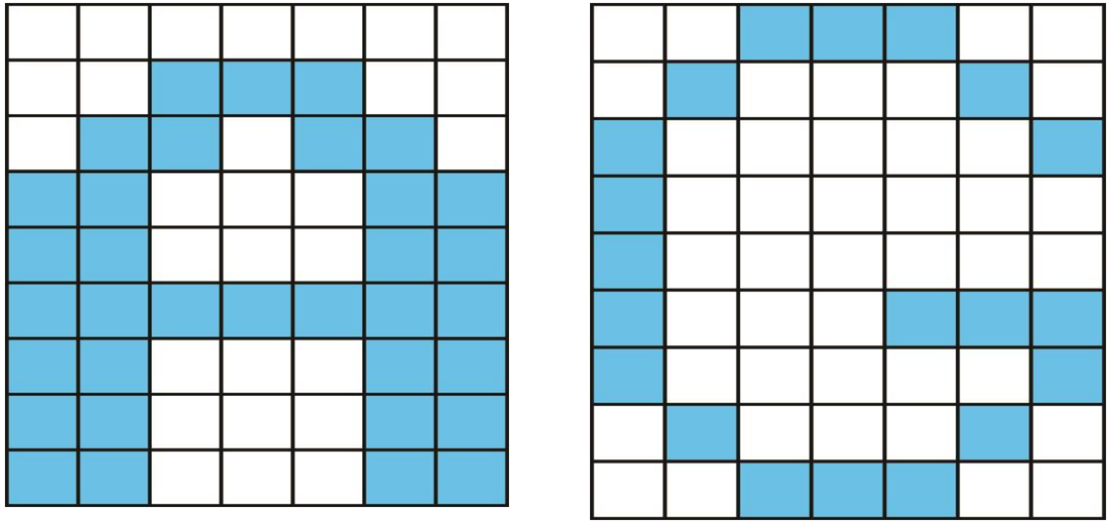
Attributes

- Attributes 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
- Only a few (gl_PointSize) are supported by WebGL functions

Attributes



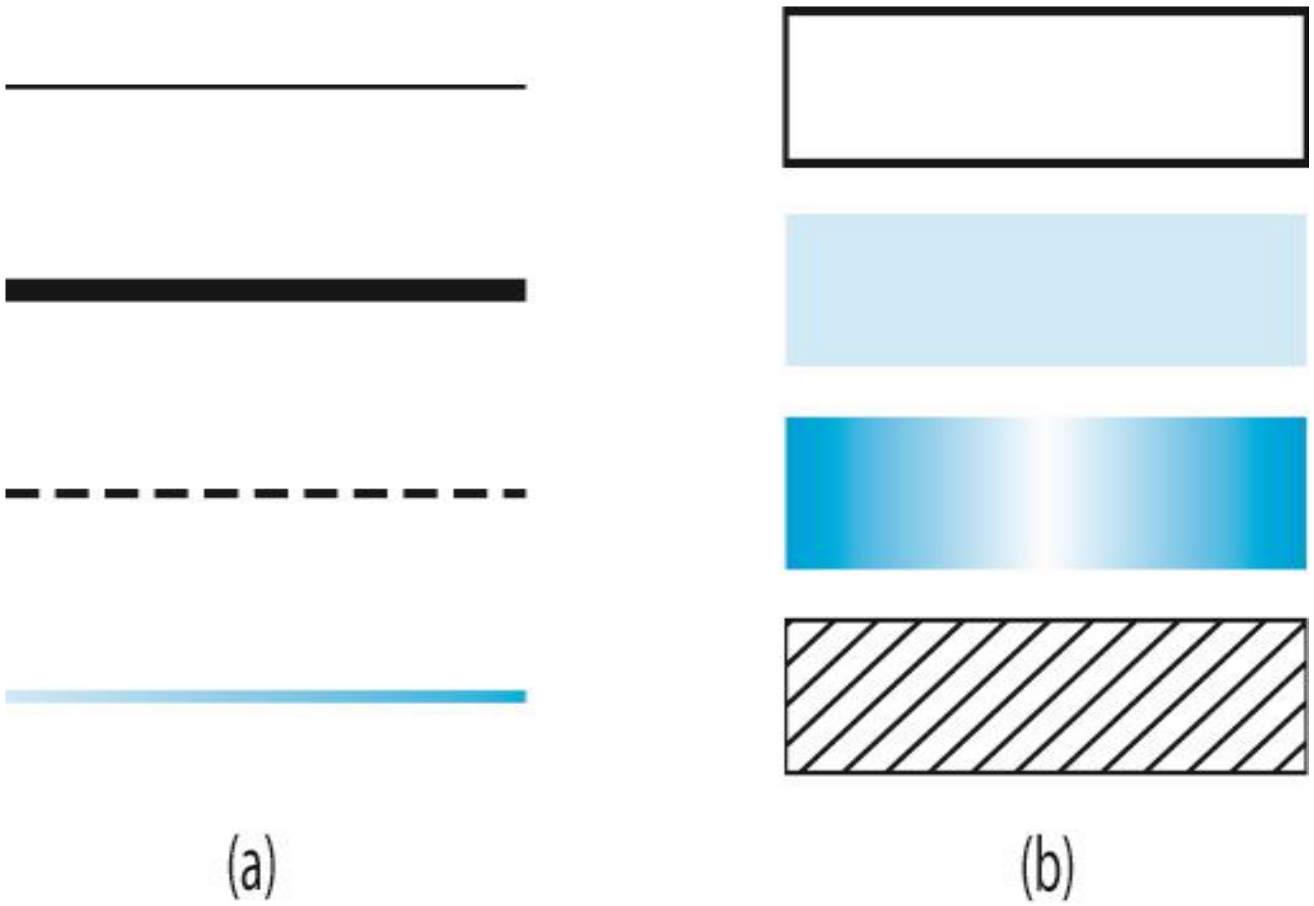
Stroke text



Raster text

Raster-character replication

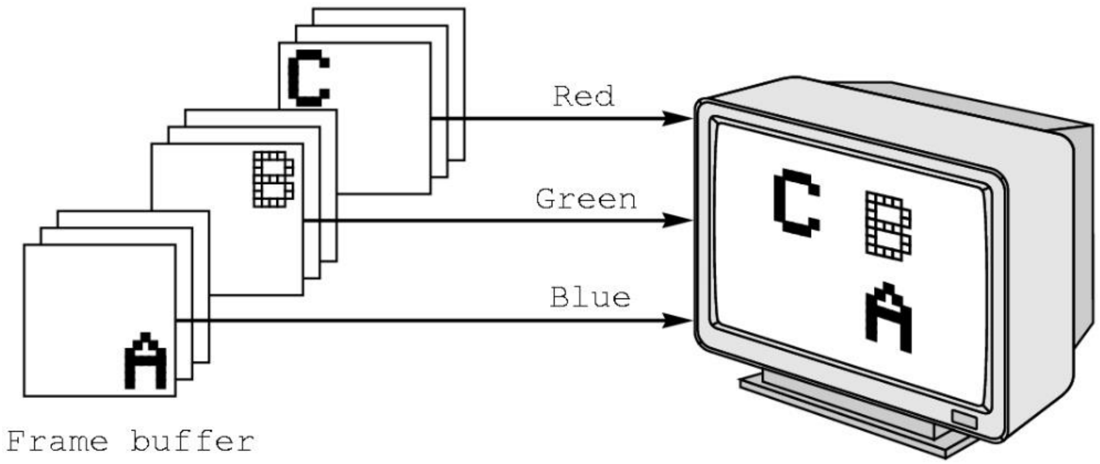
Attributes



Lines Polygons

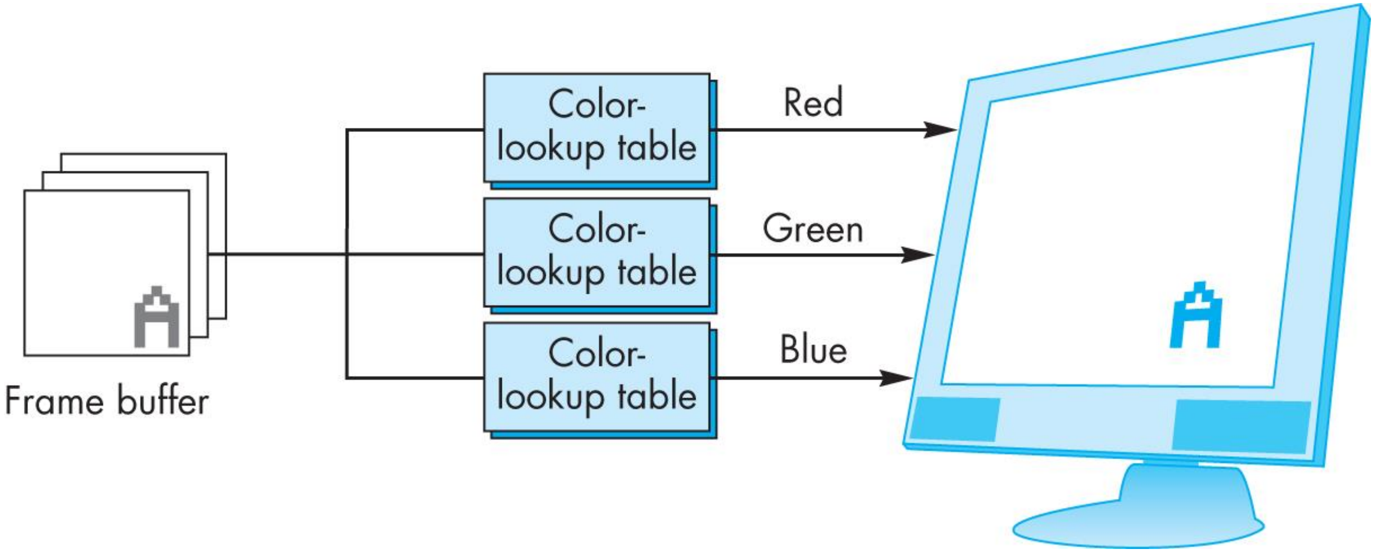
RGB color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes



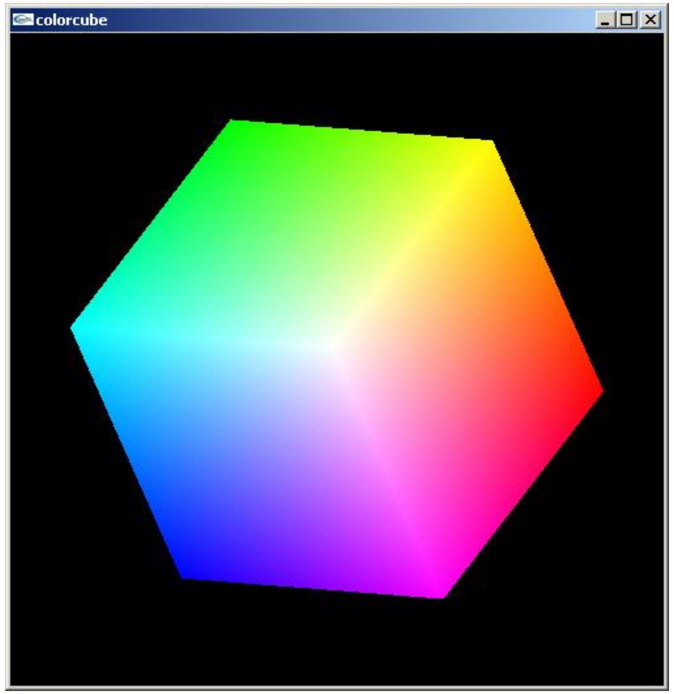
Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive
    - Need more colors for shading



Smooth Color

- Default is *smooth* shading
  - Rasterizer interpolates vertex colors across visible polygons
- Alternative is *flat shading*
  - Color of first vertex determines fill color
  - Handle in shader



Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
- Application color: pass to vertex shader as a uniform variable or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable
- Fragment color: can alter via shader code

Programming with WebGL
Part 6: Three Dimensions

Objectives

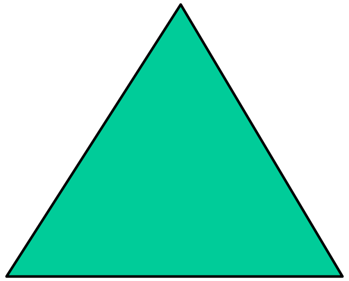
- Develop a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal

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

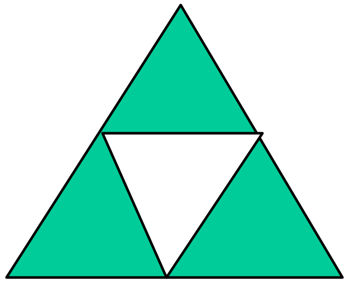
Sierpinski Gasket (2D)

- Start with a triangle



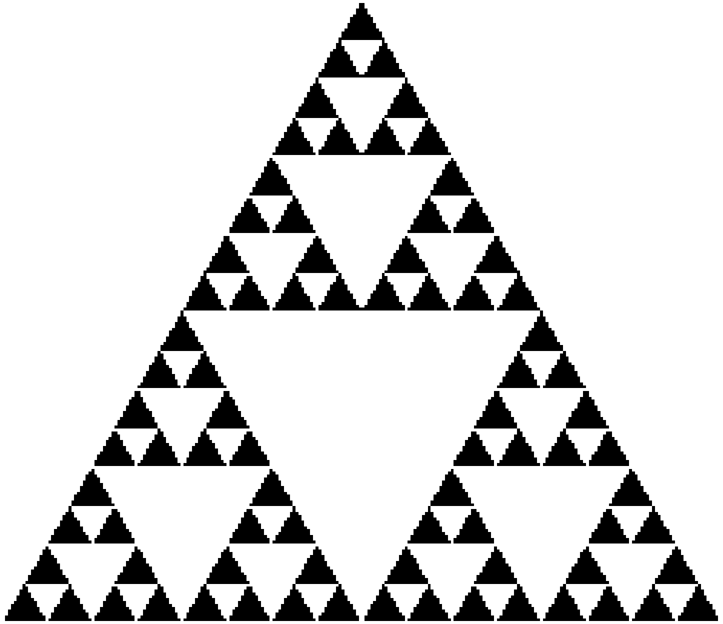
- Connect bisectors of sides and remove central triangle

- 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

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 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 );
}
```

Programming with WebGL
Part 6: Three Dimensions

Moving to 3D

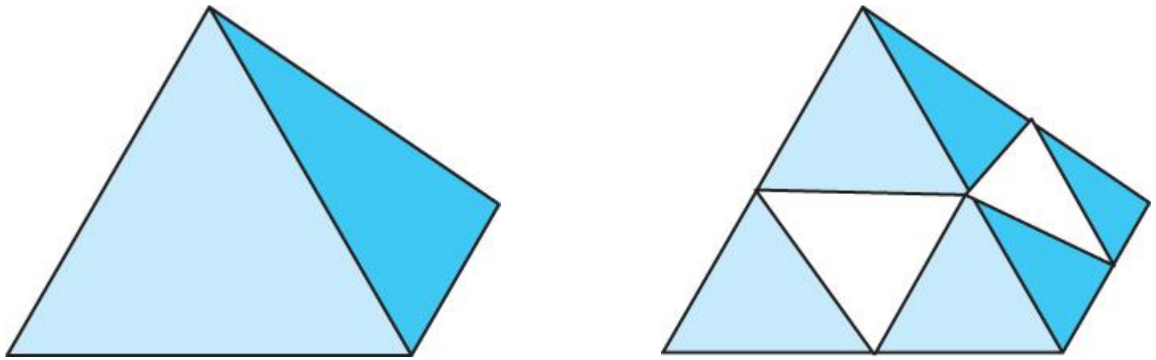
- We can easily make the program three-dimensional by using three dimensional points and starting with a tetrahedron

```
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

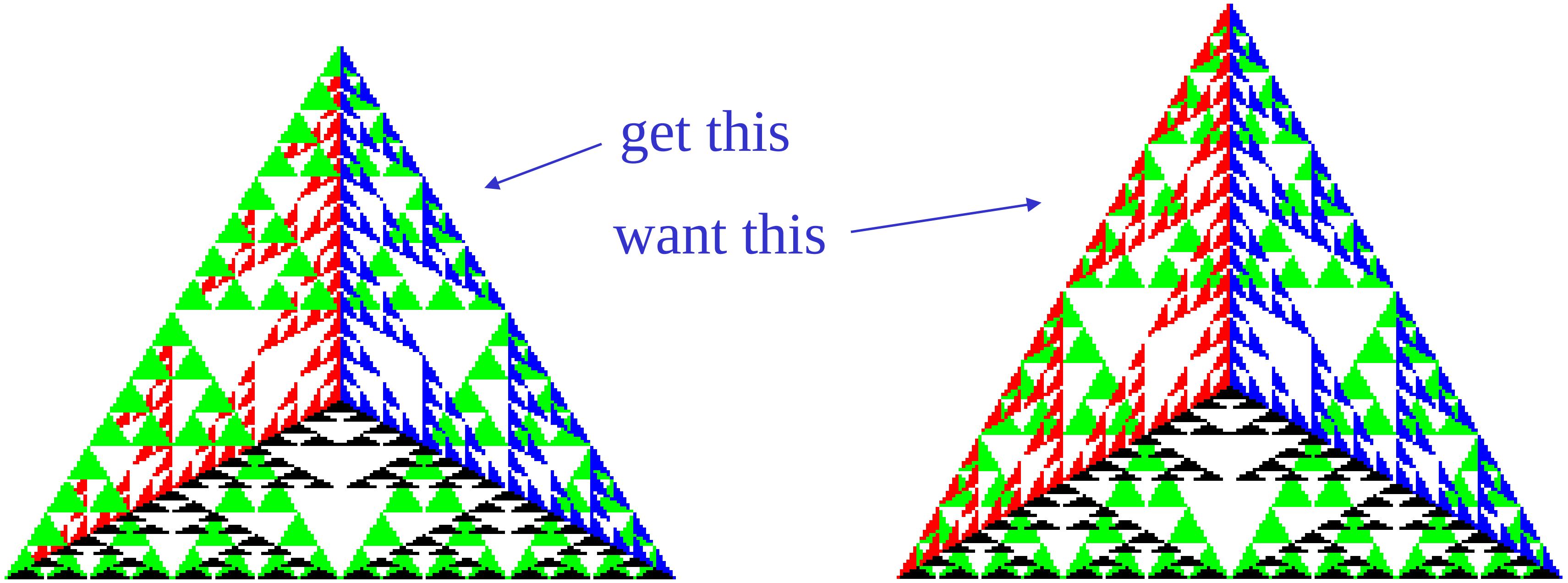
- We can subdivide each of the four faces



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

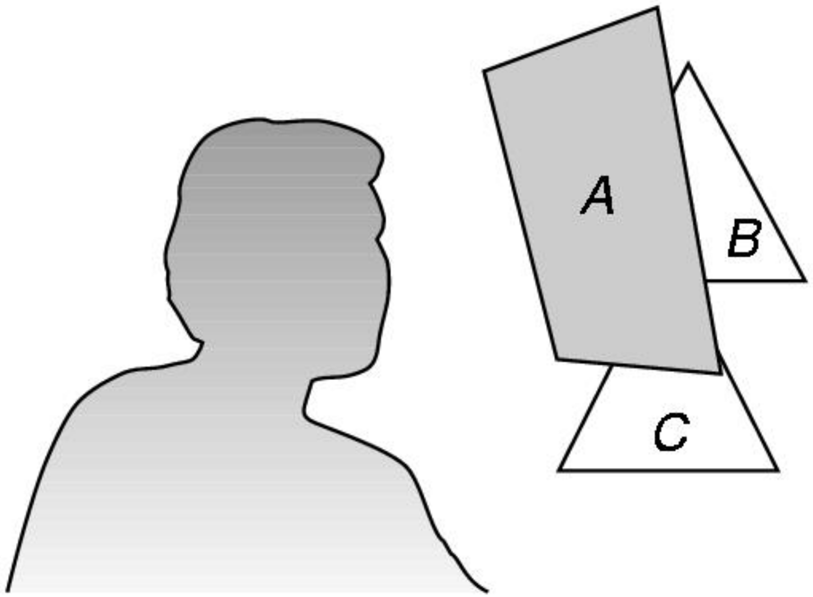
Almost Correct

- 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



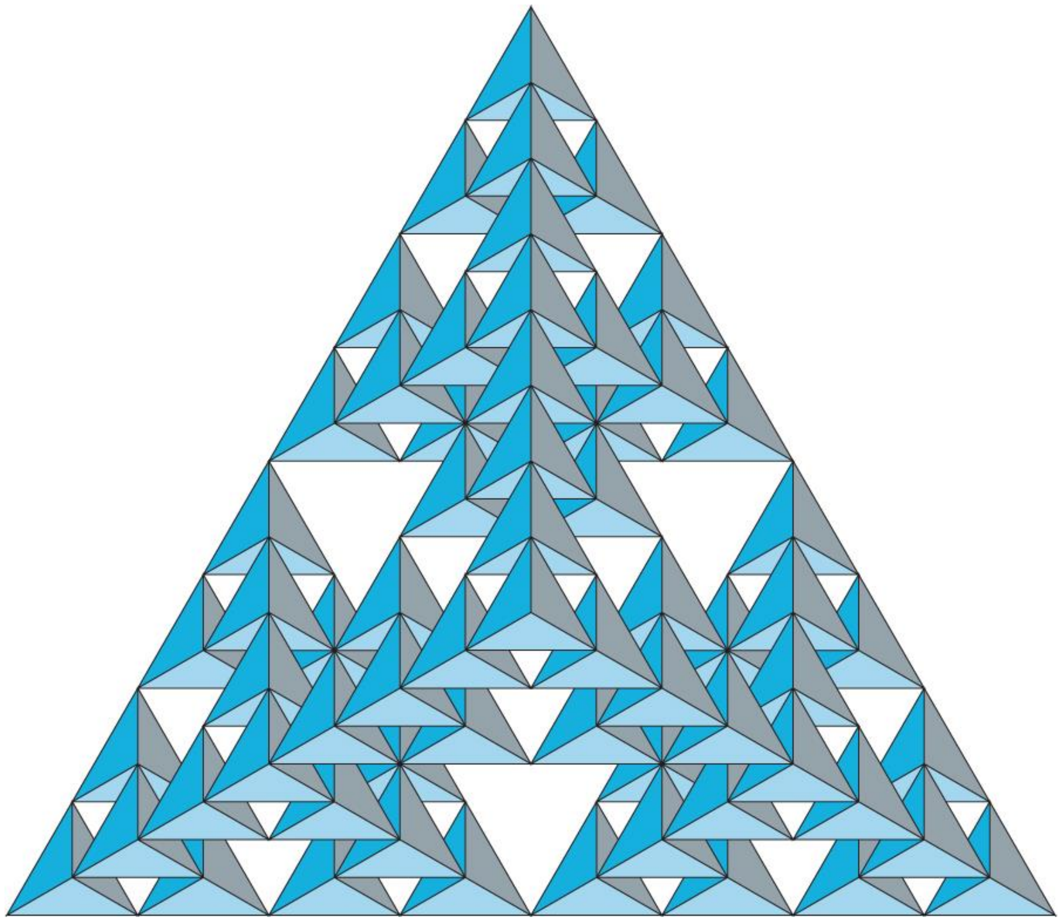
Using the *z*-buffer algorithm

- 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)`

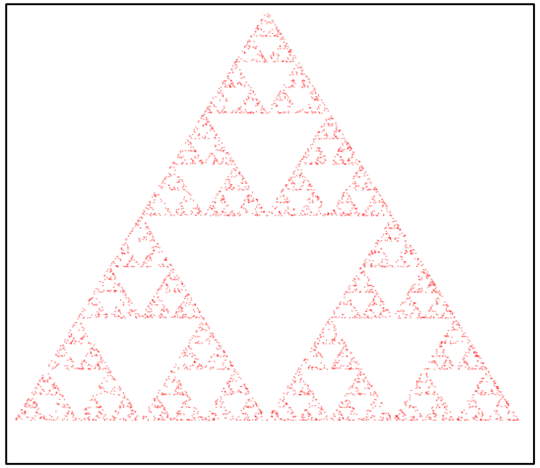
Surface vs Volume Subdvision

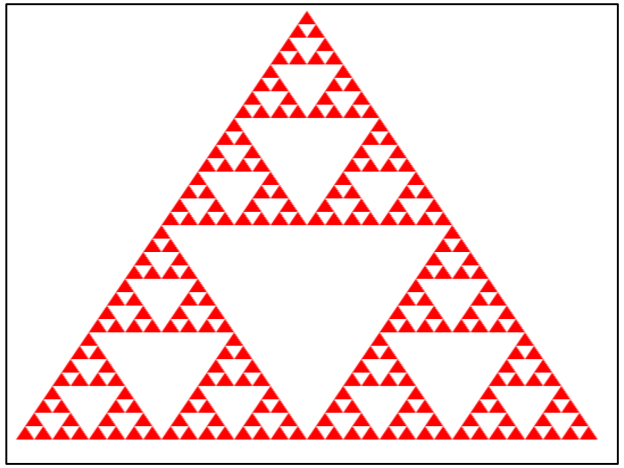
- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a *volume* in the middle
- See text for code

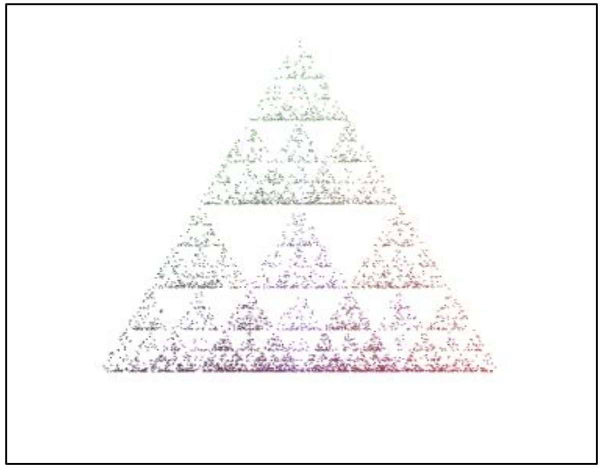
Volume Subdivision

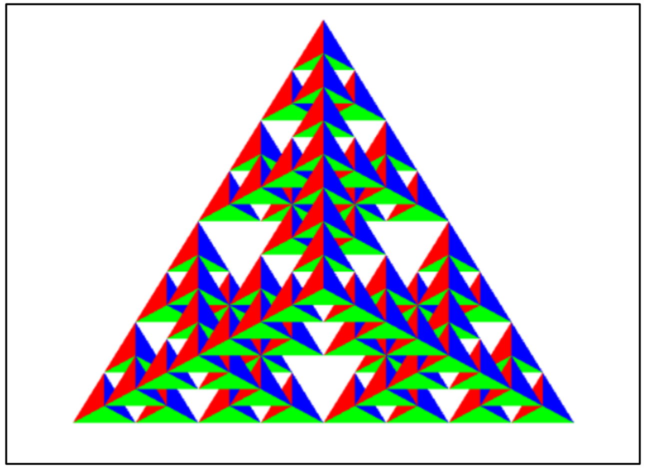


Programming with WebGL
Sample Programs

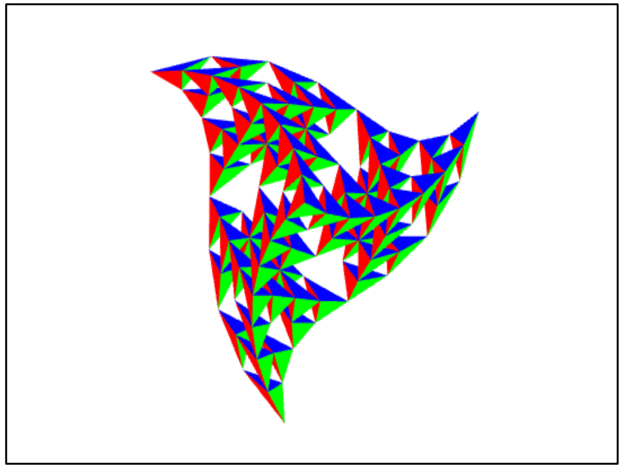

gasket1


gasket2


gasket3

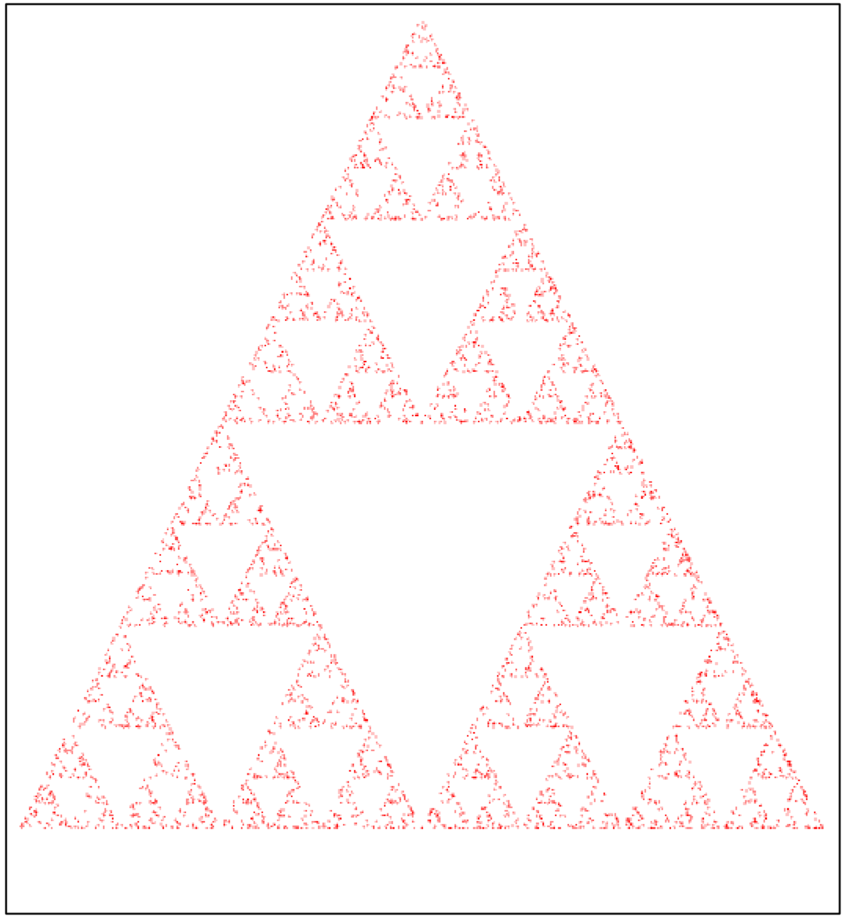

gasket4

Programming with WebGL
Sample Programs



gasket6

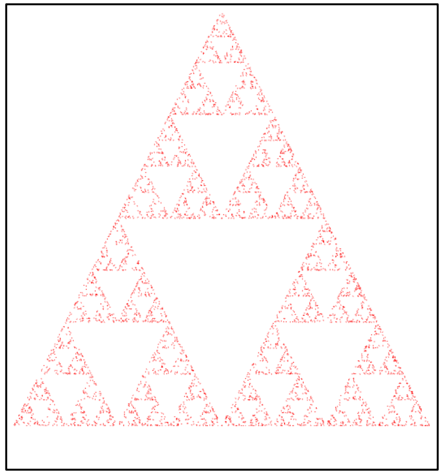
Sample Programs: gasket1.html, gasket1.js



Generates Sierpinski Gasket using 5000 points generated by random algorithm

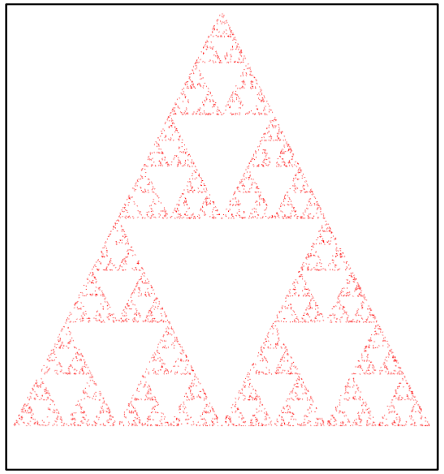
gasket1.html (1/3)

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
<title>2D Sierpinski Gasket</title>

<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;

void
main()
{
    gl_PointSize = 1.0;
    gl_Position = vPosition;
}
</script>
```



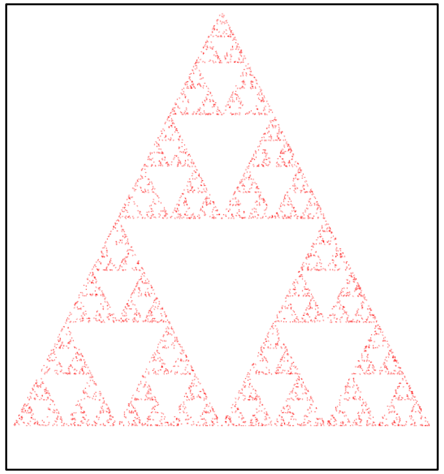
gasket1.html (2/3)

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

void
main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>

<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="gasket1.js"></script>
</head>
```



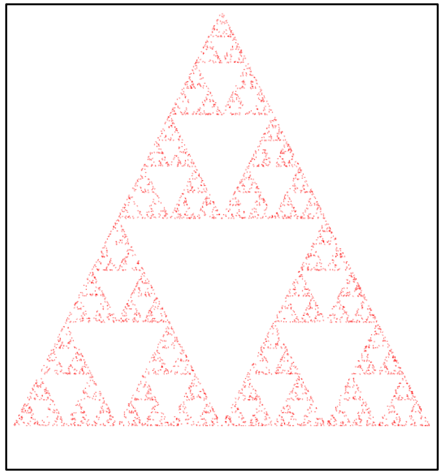
gasket1.html (3/3)

```
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

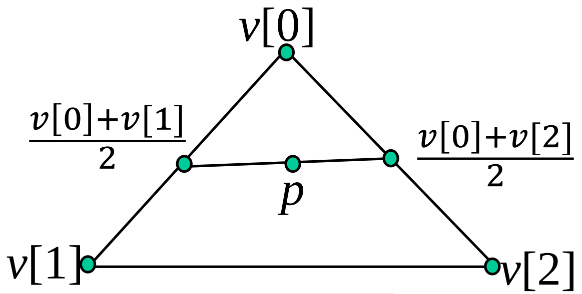


gasket1.js (1/4)

```
var gl;
var points;

var NumPoints = 5000;

window.onload = function init()
{
    var canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }

    //  Initialize our data for the Sierpinski Gasket

    // First, initialize the corners of our gasket with three points.

    var vertices = [ vec2( -1, -1 ), vec2(  0,  1 ), vec2(  1, -1 ) ];
```

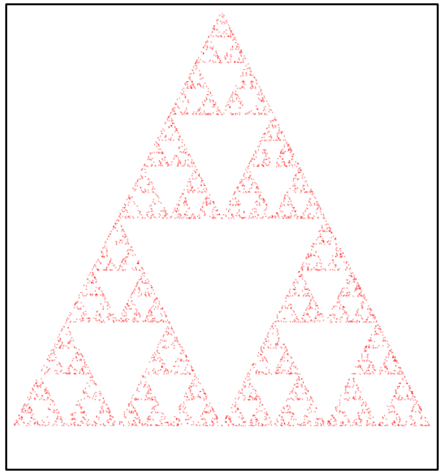


gasket1.js (2/4)




```
// Specify a starting point p for our iterations
    // p must lie inside any set of three vertices
    var u = add( vertices[0], vertices[1] );
    var v = add( vertices[0], vertices[2] );
    var p = scale( 0.25, add( u, v ) );

    // And, add our initial point into our array of points
    points = [ p ];

// Compute new points
    // Each new point is located midway between
    // last point and a randomly chosen vertex

    for ( var i = 0; points.length < NumPoints; ++i ) {
        var j = Math.floor(Math.random() * 3);
        p = add( points[i], vertices[j] );
        p = scale( 0.5, p );
        points.push( p );
    }
```

$$p = \frac{(v[0]+v[1]) + (v[0]+v[2])}{4} = \frac{\frac{v[0]+v[1]}{2} + \frac{v[0]+v[2]}{2}}{2}$$



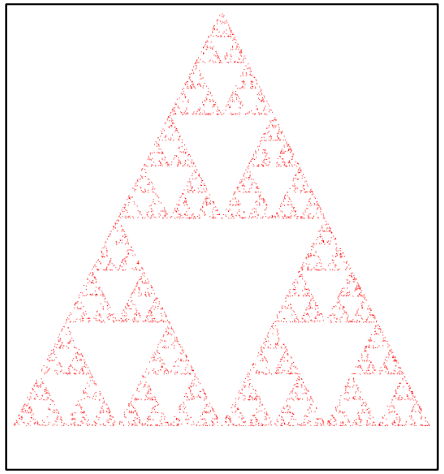
Math.random(): random-number generator that produces a random number in 0~0.9999999.

Math.random()*3): produce a number in 0~2.9999999

Math.floor(Math.random()*3): produce integers 0, 1, and 2

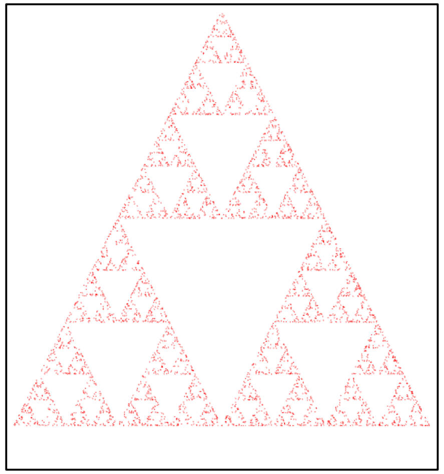
gasket1.js (3/4)

```
//
//  Configure WebGL
//
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

//  Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

// Load the data into the GPU

var bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );
```

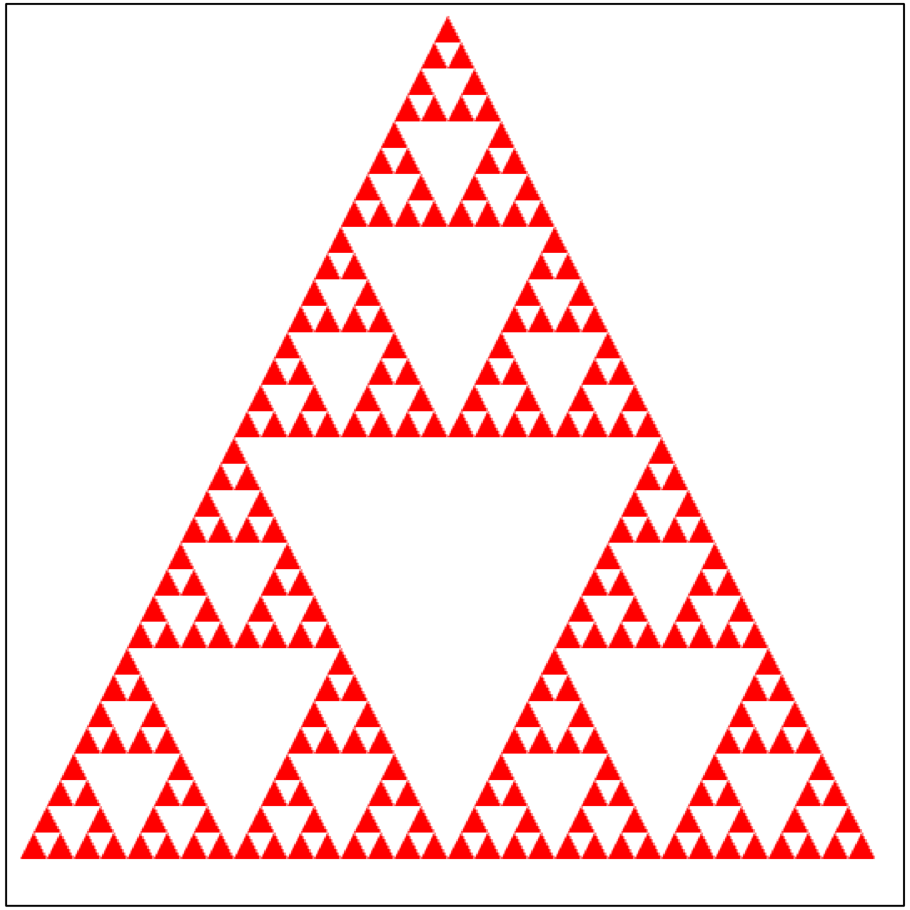


gasket1.js (4/4)

```
    // Associate out shader variables with our data buffer

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

    render();
};  // end of window.onload

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



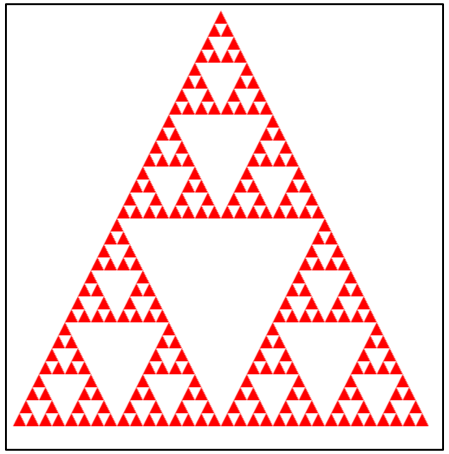
Sample Programs: gasket2.html, gasket2.js



Generating Sierpinski Gasket by recursion

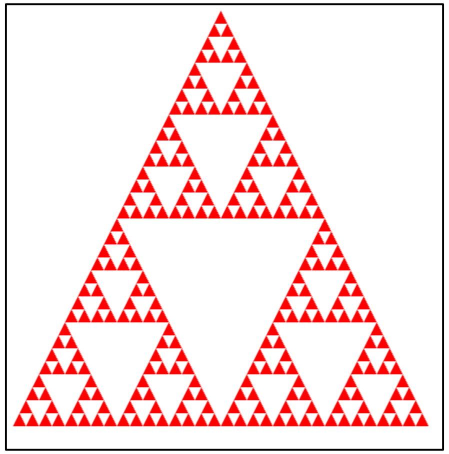
gasket2.html (1/3)

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
<title>2D Sierpinski Gasket</title>


<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;

void
main()
{
    gl_Position = vPosition;
}
</script>
```



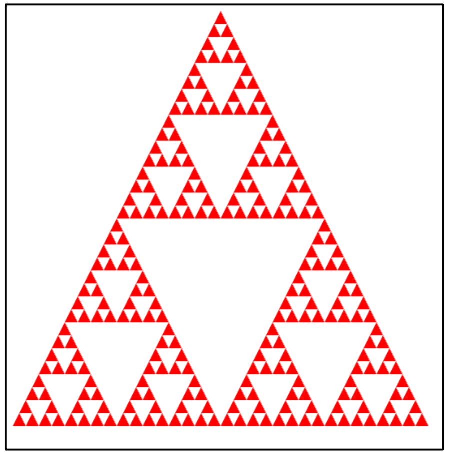
gasket2.html (2/3)

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

void
main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>

<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="gasket2.js"></script>
</head>
```

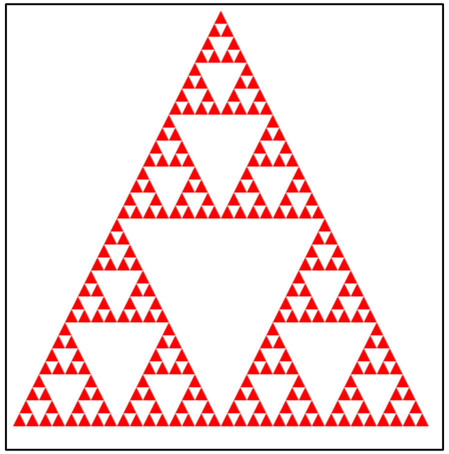


gasket2.html (3/3)

```
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```



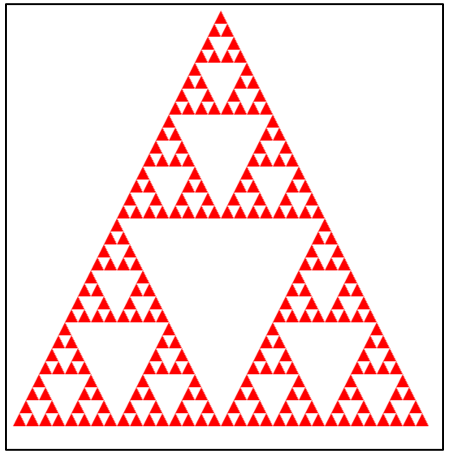
gasket2.js (1/4)



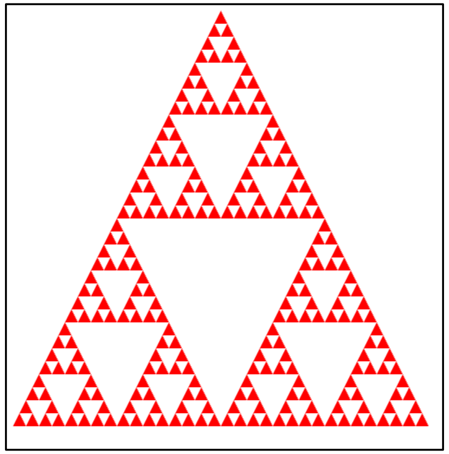
```
var canvas;
var gl;

var points = [];

var NumTimesToSubdivide = 5;

window.onload = function init()
{
    canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }
```

gasket2.js (2/4)

```
//  Initialize our data for the Sierpinski Gasket

    // First, initialize the corners of our gasket with three points.

    var vertices = [ vec2( -1, -1 ), vec2(  0,  1 ), vec2(  1, -1 ) ];

    divideTriangle( vertices[0], vertices[1], vertices[2], NumTimesToSubdivide);

    //  Configure WebGL

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

    //  Load shaders and initialize attribute buffers

    var program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program );
```



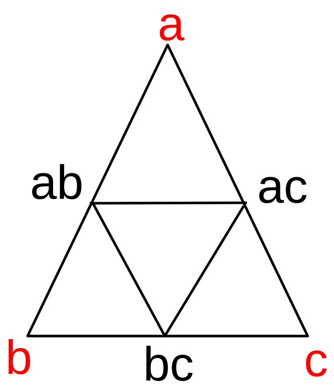
gasket2.js (3/4)



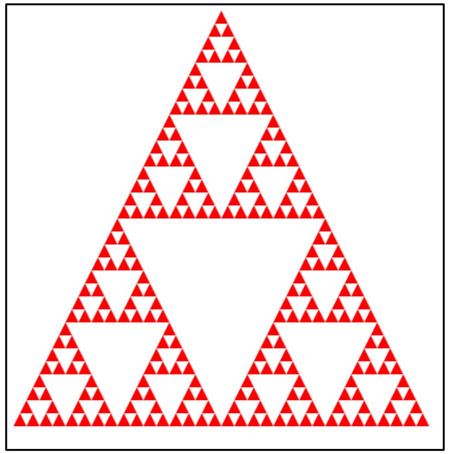
```
// Load the data into the GPU

    var bufferId = gl.createBuffer();
    gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

    // Associate out shader variables with our data buffer

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

    render();

};  //  end of window.onload
```

gasket2.js (4/4)

```
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 );
            divideTriangle( c, ac, bc, count );
            divideTriangle( b, bc, ab, count );
      }
}
```



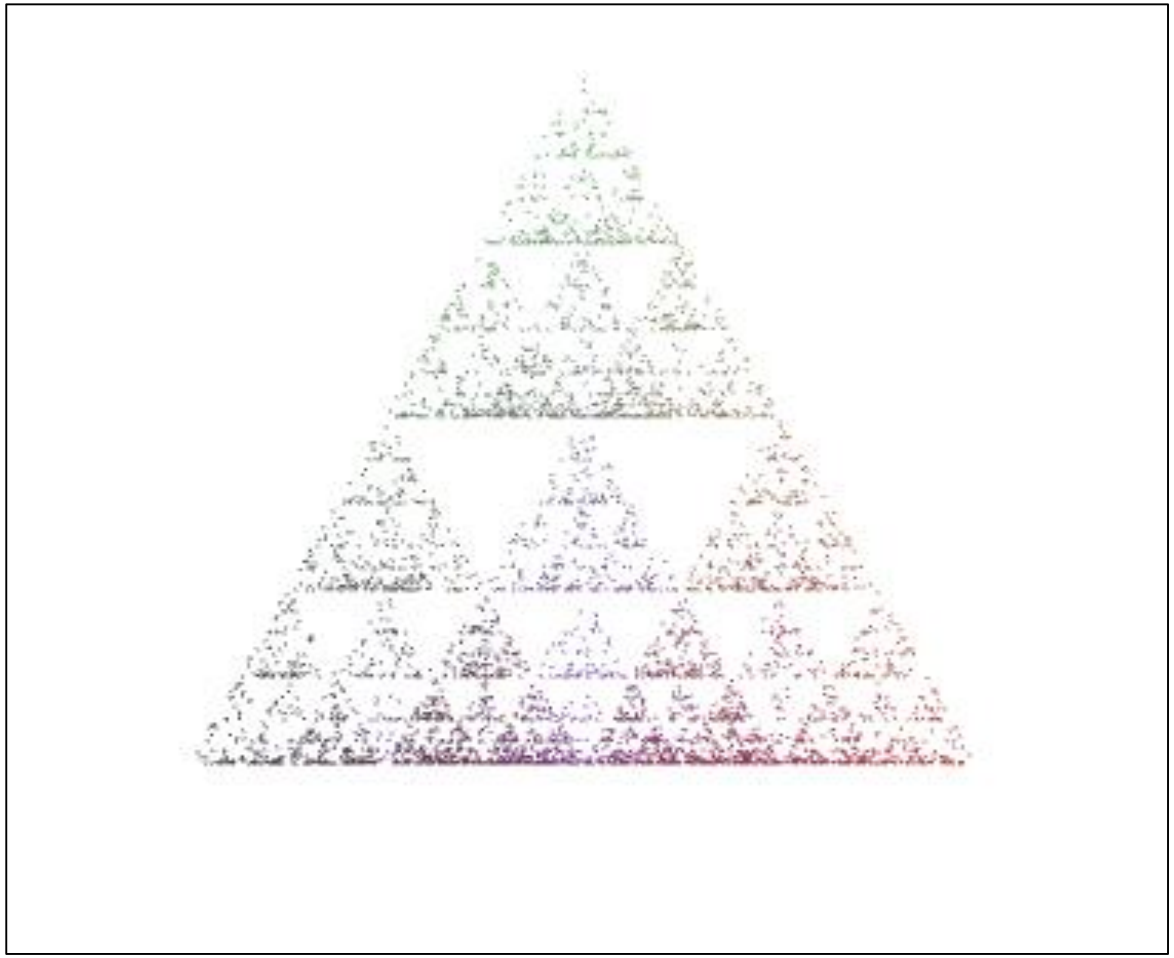

```
function triangle( a, b, c )
{
    points.push( a, b, c );
}

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



$$\boxed{\text{mix}(a, b, s) = s * a + (1-s) * b}$$

$$\Rightarrow \text{mix}(a, b, 0.5) = \frac{a + b}{2}$$

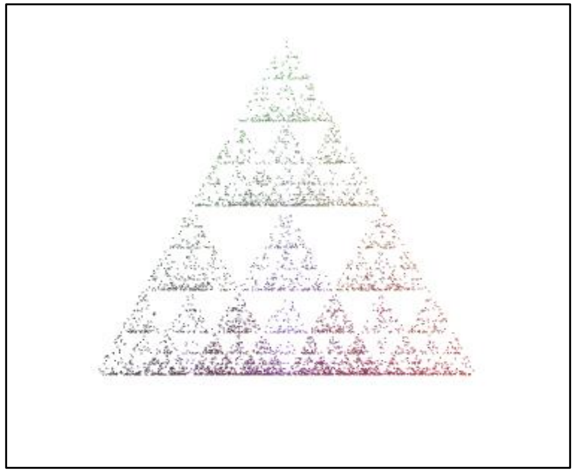
Sample Programs: gasket3.html, gasket3.js



Generating 3D Sierpinski Gasket by random algorithm

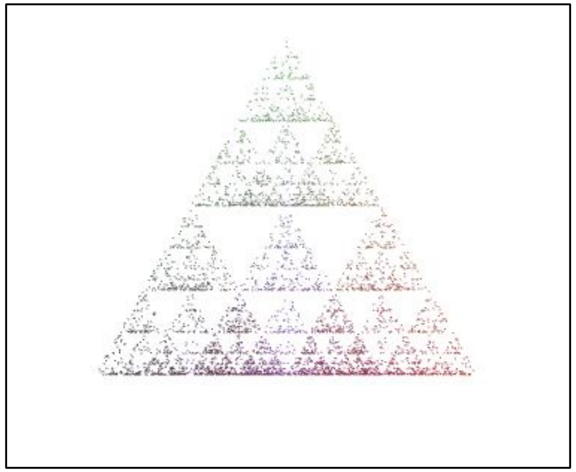
gasket3.html (1/3)

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
<title>3D Sierpinski Gasket</title>

<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
varying   vec4 vColor;

void
main()
{
    gl_PointSize = 1.0;
    vColor = vec4((1.0+vPosition.xyz)/2.0, 1.0);
    gl_Position = vPosition;
}
</script>
```



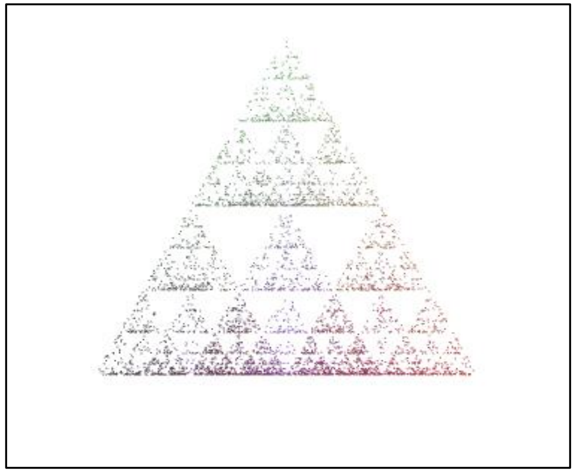
gasket3.html (2/3)

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

varying vec4 vColor;

void
main()
{
    gl_FragColor = vColor;
}
</script>

<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="gasket3.js"></script>
</head>
```



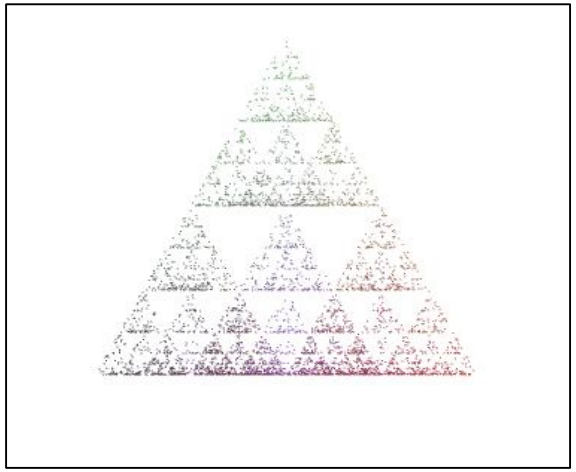
gasket3.html (3/3)

```
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

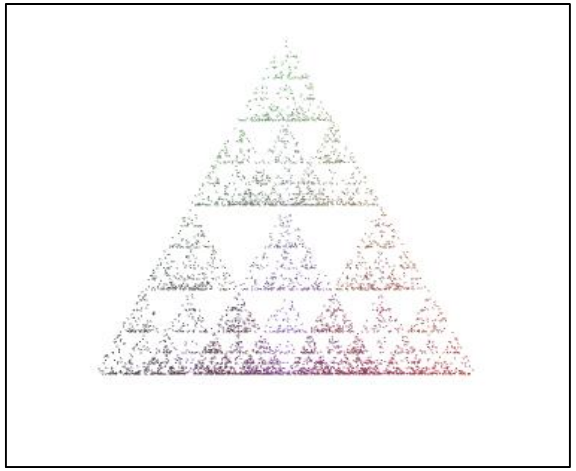


gasket3.js (1/4)

```
var canvas;
var gl;
var points;

var NumPoints = 5000;

window.onload = function init()
{
    canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }
```



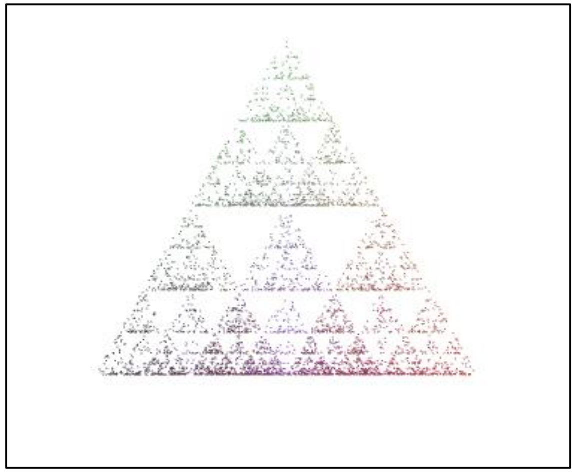
gasket3.js (2/4)



```
//
//  Initialize our data for the Sierpinski Gasket
//

// First, initialize the vertices of our 3D gasket

var vertices = [ vec3( -0.5, -0.5, -0.5 ), vec3(  0.5, -0.5, -0.5 ), vec3(  0.0,  0.5,  0.0 ), vec3(  0.0, -0.5, 0.5 ) ];

points = [ vec3( 0.0, 0.0, 0.0 ) ];

for ( var i = 0; points.length < NumPoints; ++i ) {
    var j = Math.floor(Math.random() * 4);

    points.push(mix(points[i], vertices[j], 0.5) );
}
```

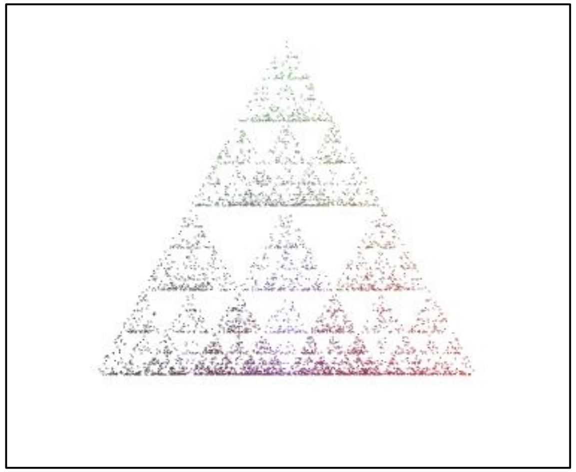
gasket3.js (3/4)

```
//  Configure WebGL

gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
gl.enable( gl.DEPTH_TEST );

//  Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

// Load the data into the GPU

var bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );
```

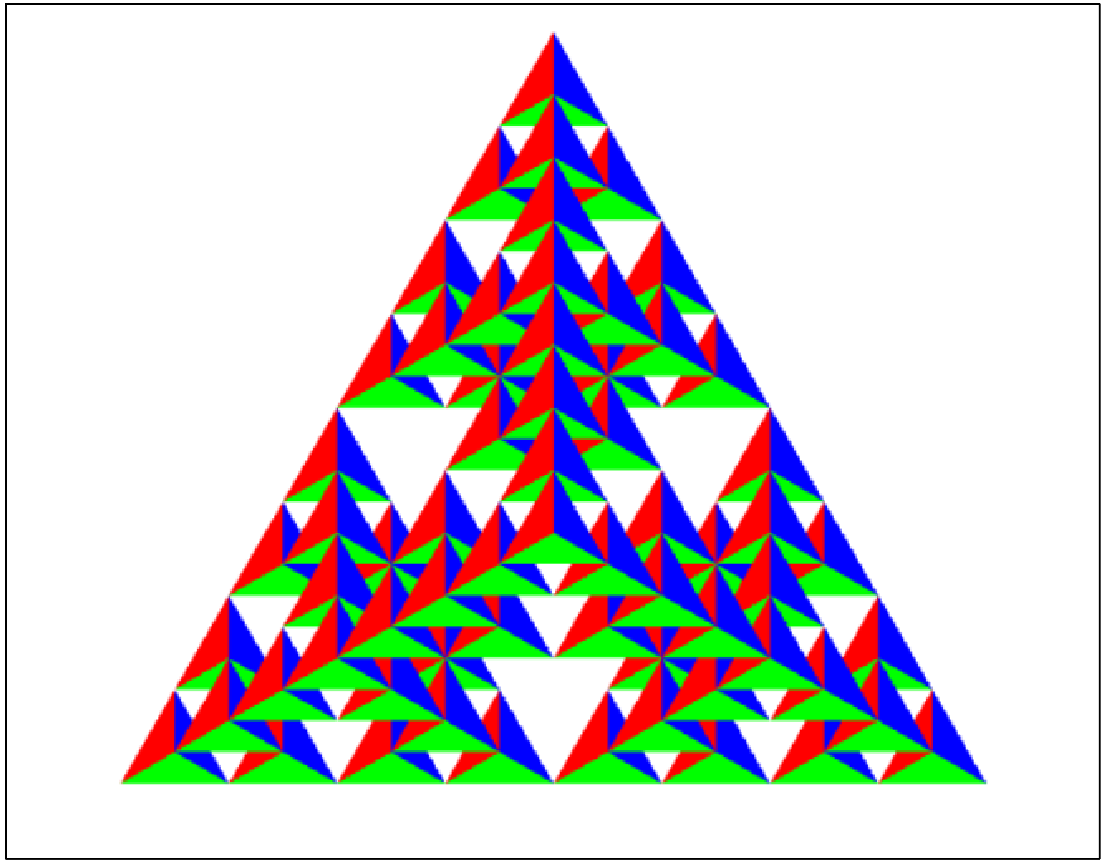


gasket3.js (4/4)

```
// Associate out shader variables with our data buffer

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

    render();
};  // end of window.onload

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



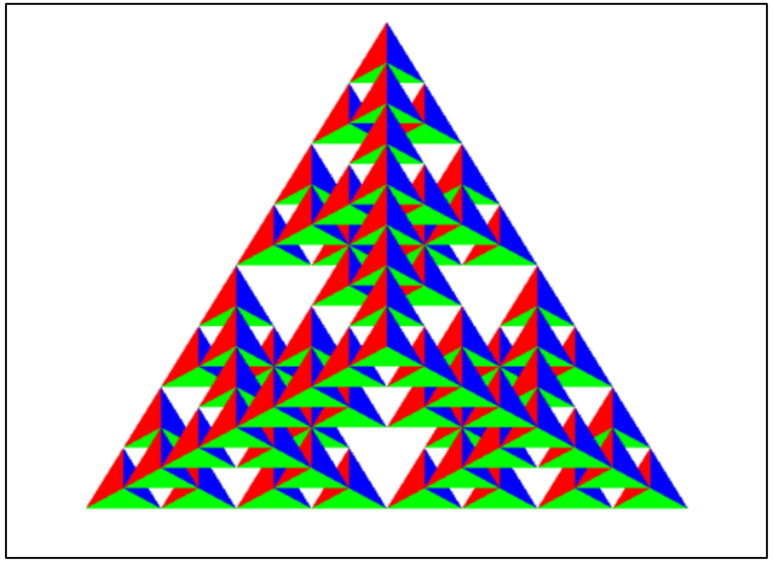
Sample Programs: gasket4.html, gasket4.js



Generating 3D Sierpinski Gasket using subdivision of tetrahedra

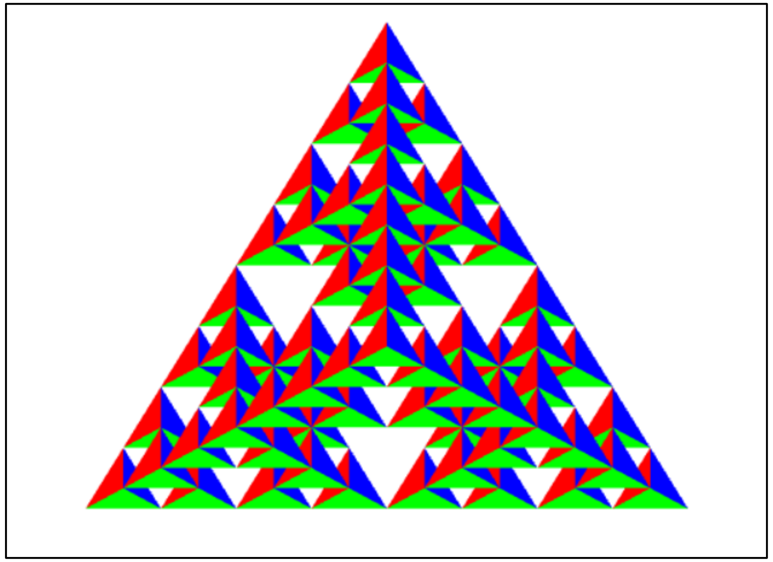
gasket4.html (1/3)

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
<title>3D Sierpinski Gasket</title>

<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec3 vPosition;
attribute vec3 vColor;
varying   vec4 color;

void
main()
{
    gl_Position = vec4(vPosition, 1.0);
    color = vec4(vColor, 1.0);
}
</script>
```



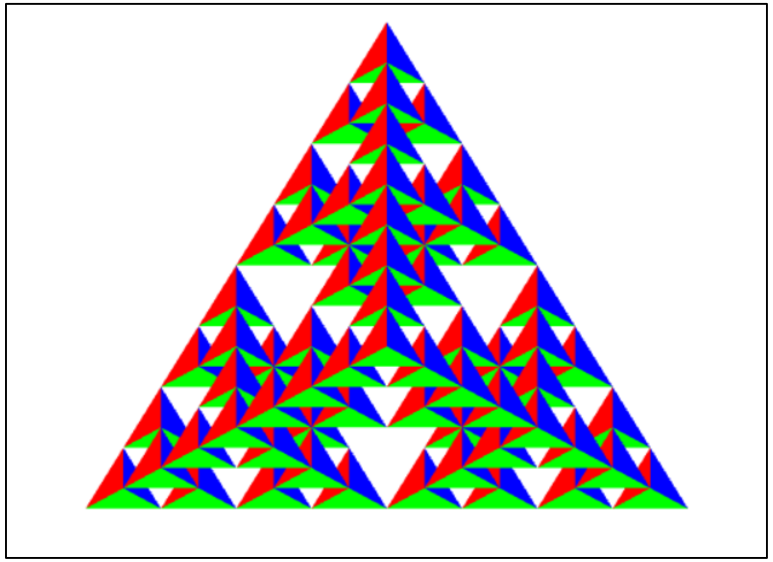
gasket4.html (2/3)

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

varying vec4 color;

void
main()
{
    gl_FragColor = color;
}
</script>

<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="gasket4.js"></script>
</head>
```



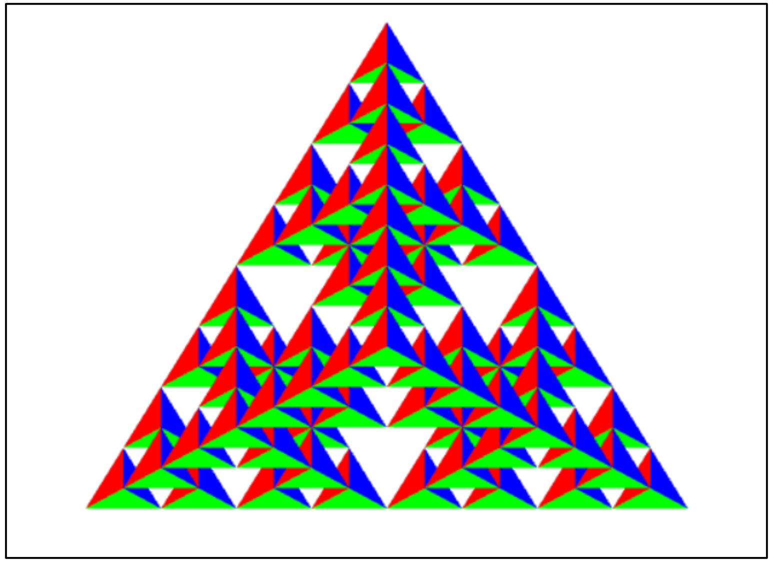
gasket4.html (3/3)

```
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```



gasket4.js (1/8)

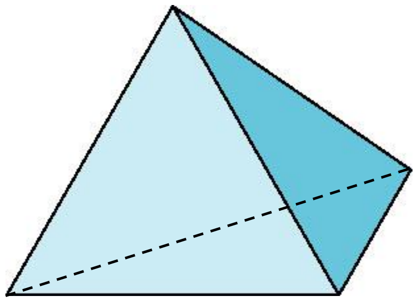
```
var canvas;
var gl;

var points = [];
var colors = [];

var NumTimesToSubdivide = 3;

window.onload = function init()
{
    canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }
```

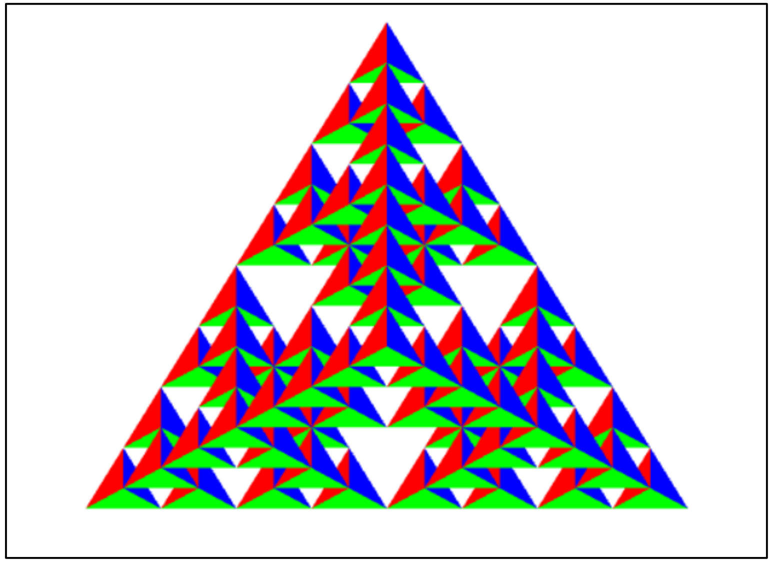


gasket4.js (2/8)

```
//
//  Initialize our data for the Sierpinski Gasket
//

// First, initialize the vertices of our 3D gasket
// Four vertices on unit circle
// Initial tetrahedron with equal length sides

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 )
];

divideTetra( vertices[0], vertices[1], vertices[2], vertices[3],  NumTimesToSubdivide);
```

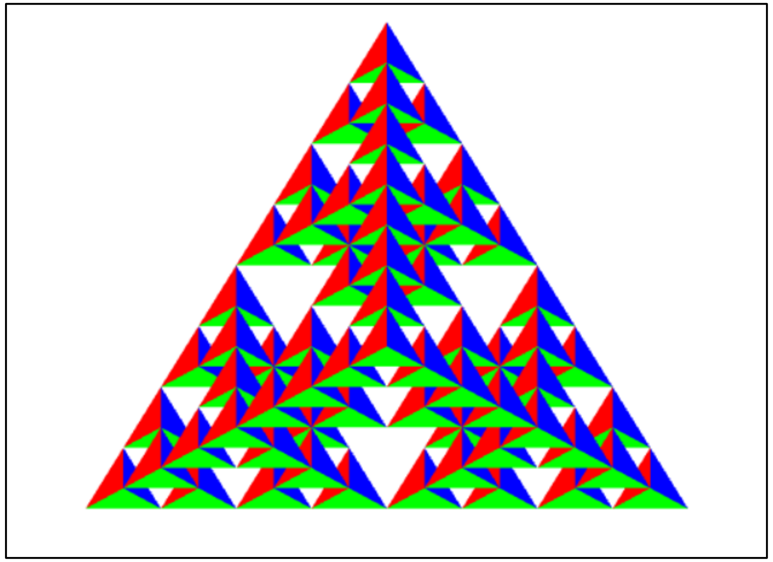
$$\left\{\begin{array}{l}(0.0, 0.0, \text{-}1.0)\\ (0.0, 2\sqrt{2}/3, 1/3)\\ (-\sqrt{6}/3, -\sqrt{2}/3, 1/3)\\ (\sqrt{6}/3, -\sqrt{2}/3, 1/3)\end{array}\right.$$





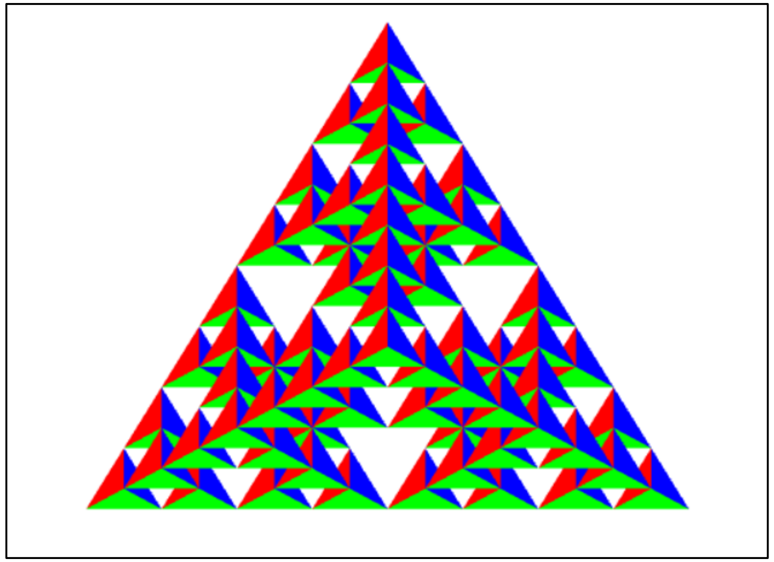
gasket4.js (3/8)

```
//
//  Configure WebGL
//
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

// enable hidden-surface removal

gl.enable(gl.DEPTH_TEST);

//  Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );
```



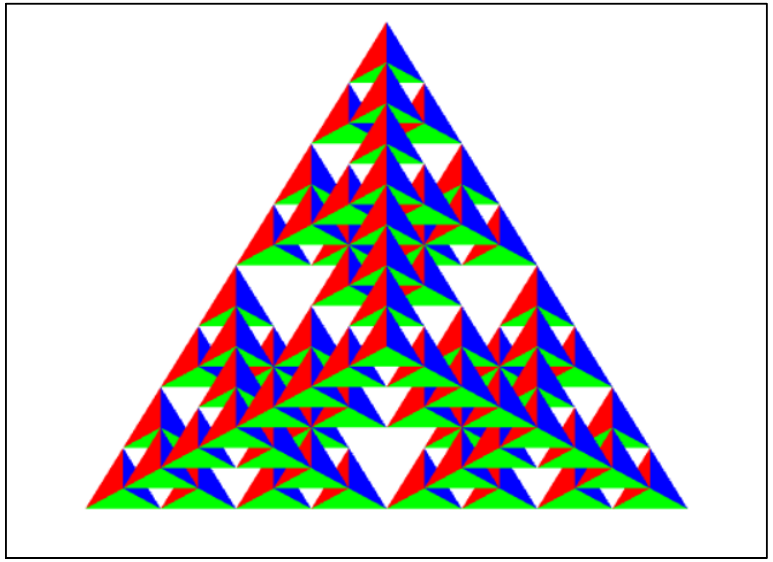
gasket4.js (4/8)

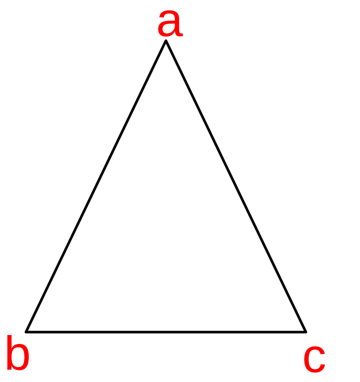
```
// Create a buffer object, initialize it, and associate it with the
// associated attribute variable in our vertex shader
    var cBuffer = gl.createBuffer();
    gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW );

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

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

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

    render();
};    // end of window.onload
```



gasket4.js (5/8)

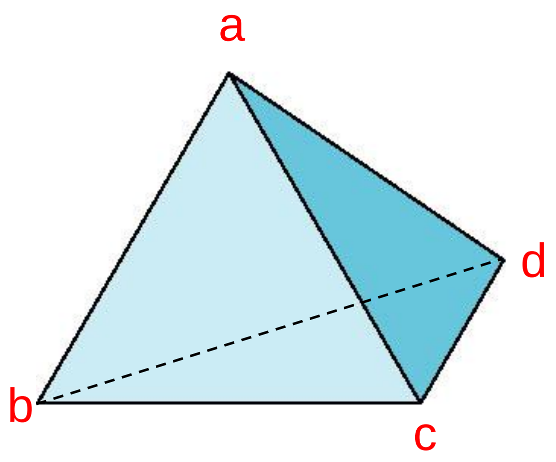
```
function triangle( a, b, c, color )
{

    // add colors and vertices for one triangle

    var baseColors = [
        vec3(1.0, 0.0, 0.0),
        vec3(0.0, 1.0, 0.0),
        vec3(0.0, 0.0, 1.0),
        vec3(0.0, 0.0, 0.0)
    ];

    colors.push( baseColors[color] );
    points.push( a );
    colors.push( baseColors[color] );
    points.push( b );
    colors.push( baseColors[color] );
    points.push( c );
}
```

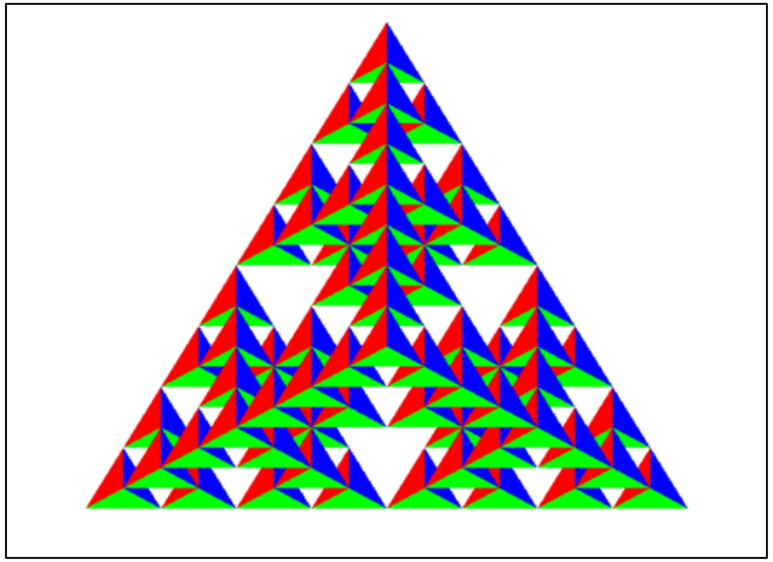




gasket4.js (6/8)

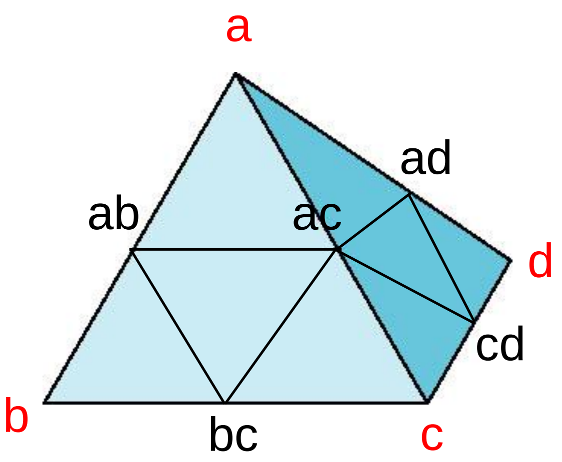
```
function tetra( a, b, c, d )
{
    // tetrahedron with each side using
    // a different color

    triangle( a, c, b, 0 );
    triangle( a, c, d, 1 );
    triangle( a, b, d, 2 );
    triangle( b, c, d, 3 );
}
```

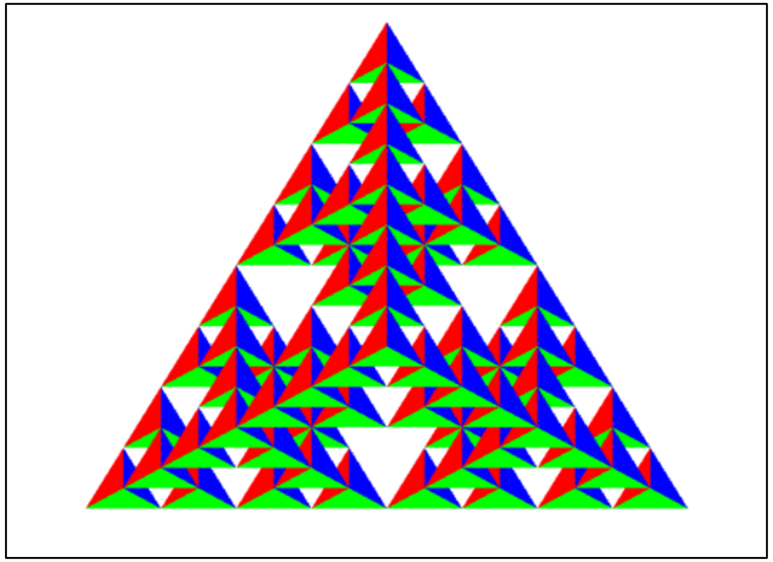




gasket4.js (7/8)

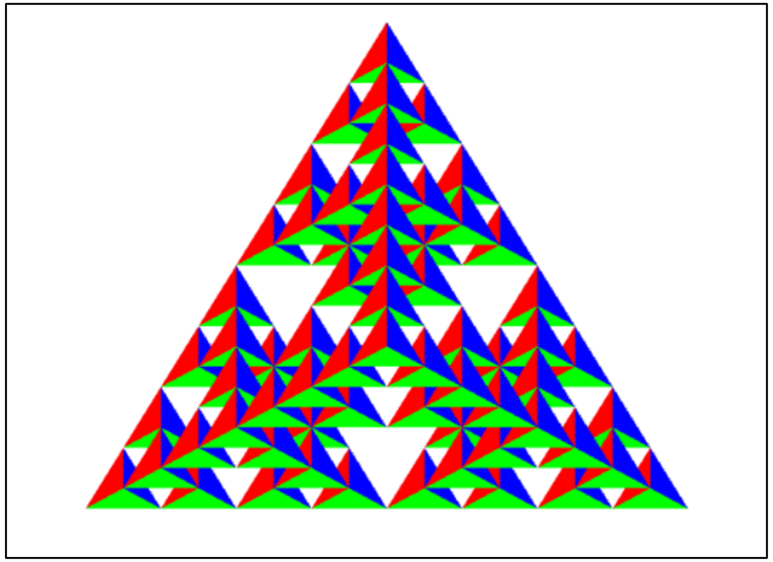
```
function divideTetra( a, b, c, d, count )
{   // check for end of recursion
    if ( count === 0 ) { tetra( a, b, c, d )  }
        // find midpoints of sides
        // divide four smaller tetrahedra
    else {   var ab = mix( a, b, 0.5 );
             var ac = mix( a, c, 0.5 );
             var ad = mix( a, d, 0.5 );
             var bc = mix( b, c, 0.5 );
             var bd = mix( b, d, 0.5 );
             var cd = mix( c, d, 0.5 );
             --count;
            divideTetra(   a, ab, ac, ad, count );
            divideTetra( ab,   b, bc, bd, count );
            divideTetra( ac, bc,   c, cd,  count );
            divideTetra( ad, bd, cd,  d,  count );
        }
}
```





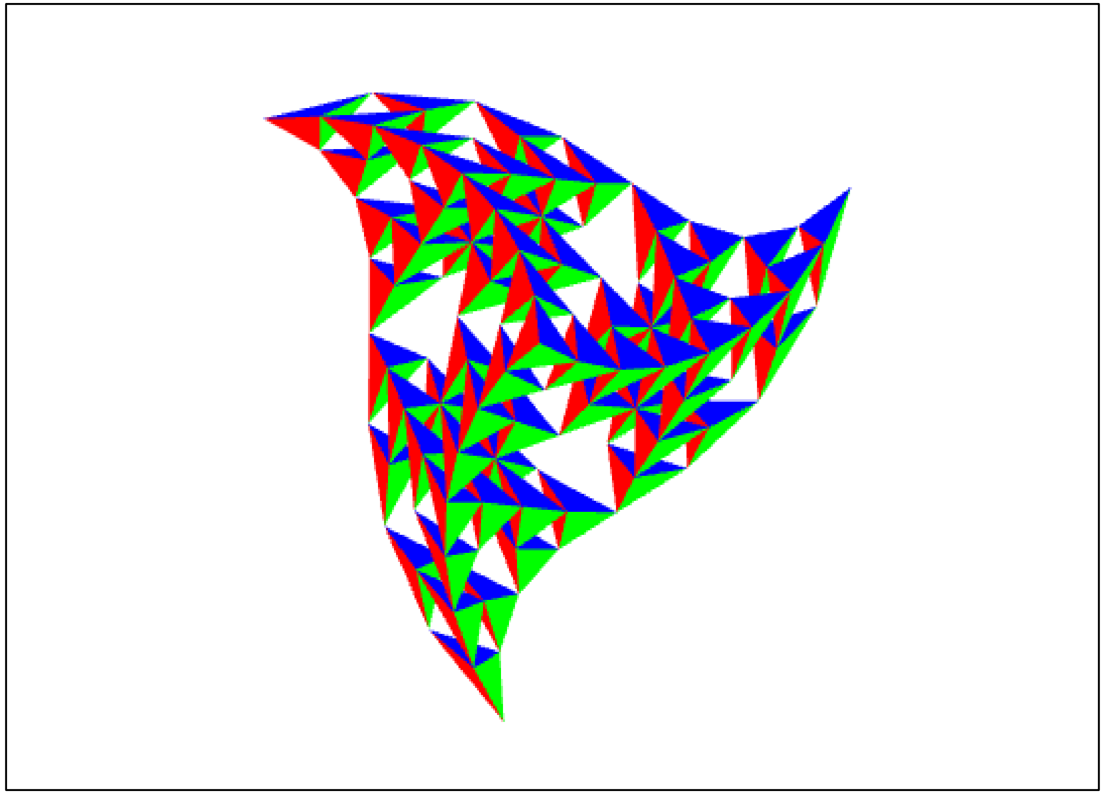
gasket4.js (8/8)

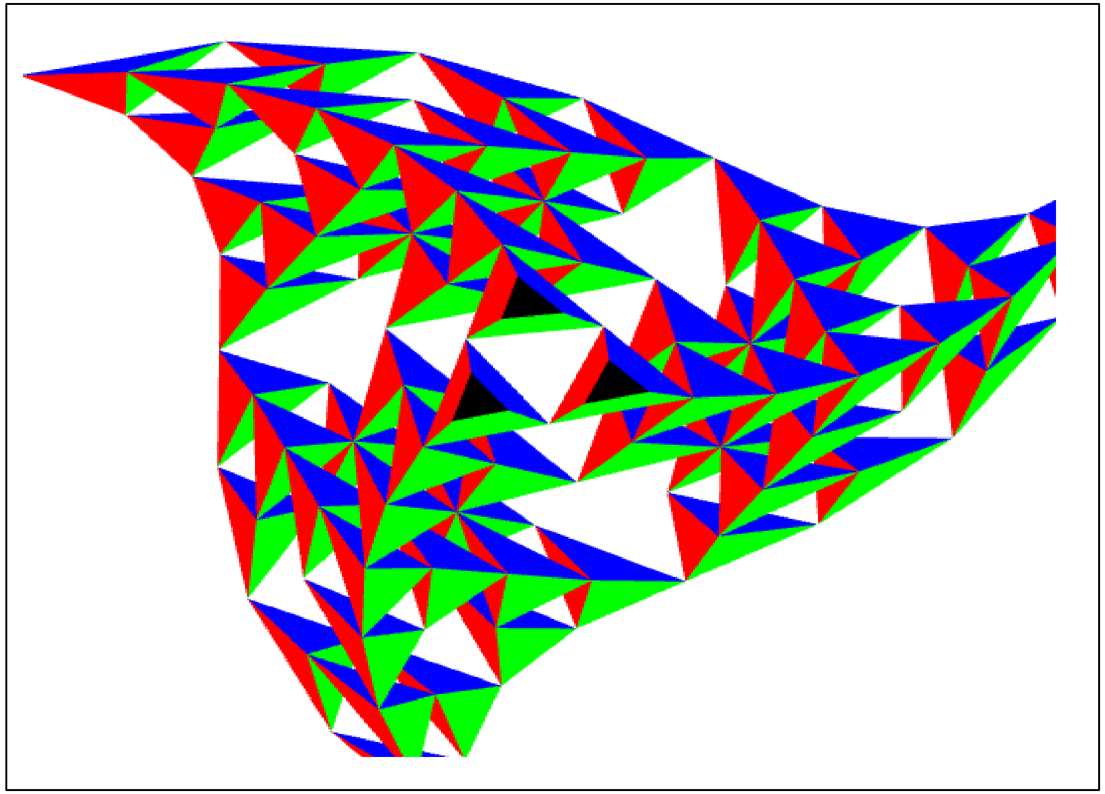
```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT |   gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, points.length );
}
```



Sample Programs: gasket6.html, gasket6.js

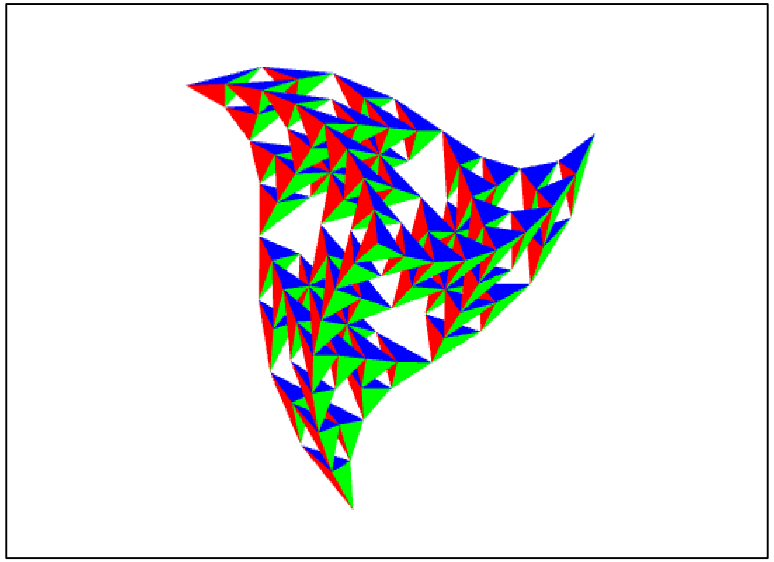
Generating 3D Sierpinski Gasket using subdivision of twisted tetrahedra





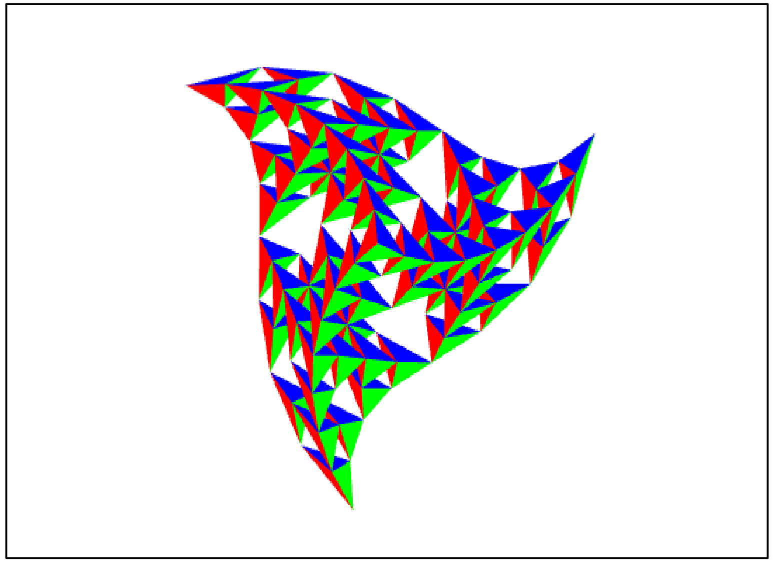
gasket6.html (1/4)

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
<title>3D Sierpinski Gasket</title>


<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec3 vPosition;
attribute vec3 vColor;
varying  vec4 color;
uniform float time;
```



gasket6.html (2/4)

```
void main()
{
    float d = sqrt(vPosition.x*vPosition.x+vPosition.y*vPosition.y);
    vec3 Pos;
    float theta = 1.0;
    Pos.x = vPosition.x*cos(d*theta)-vPosition.y*sin(d*theta);
    Pos.y = vPosition.y*cos(d*theta)+vPosition.x*sin(d*theta);
    Pos.z = vPosition.z;
    //gl_Position = vec4((1.0+0.5*sin(time))*vPosition, 1.0);
    gl_Position = vec4((1.0+0.5*sin(time))*Pos, 1.0);
    color = vec4(vColor, 1.0);
}
</script>
```

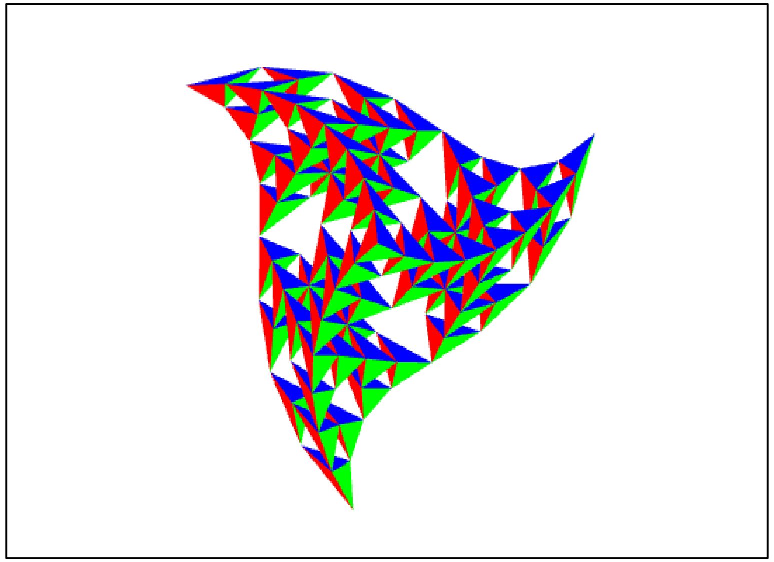


$$\begin{bmatrix} \cos(d\theta) & -\sin(d\theta) \\ \sin(d\theta) & \cos(d\theta) \end{bmatrix} \begin{bmatrix} vPosition.x \\ vPosition.y \end{bmatrix} = [Pos.x \quad Pos.y]$$

where $d = \sqrt{vPosition.x^2 + vPosition.y^2}$

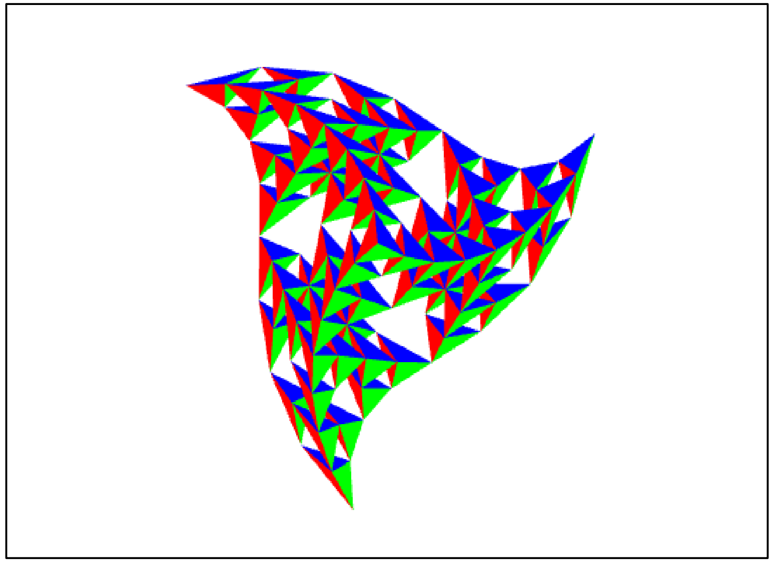
gasket6.html (3/4)

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

varying vec4 color;

void main()
{
    gl_FragColor = color;
}
</script>

<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="gasket6.js"></script>
</head>
```



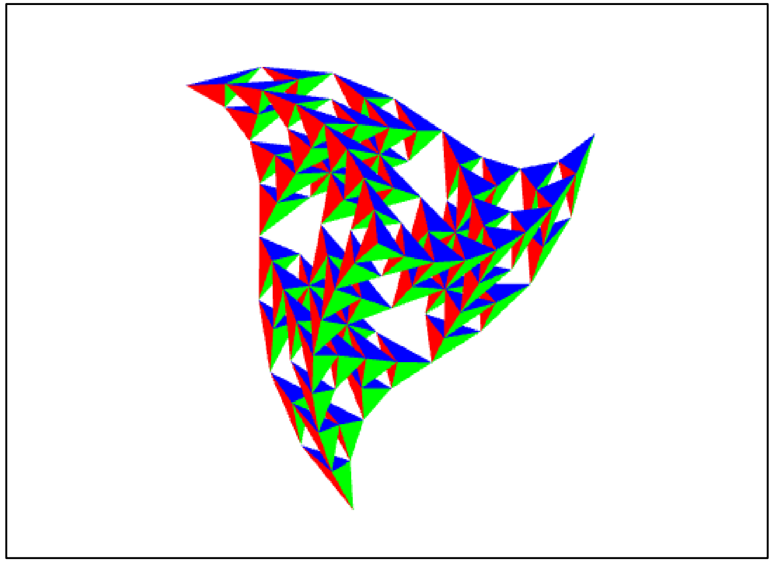
gasket6.html (4/4)

```
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```



gasket6.js (1/9)

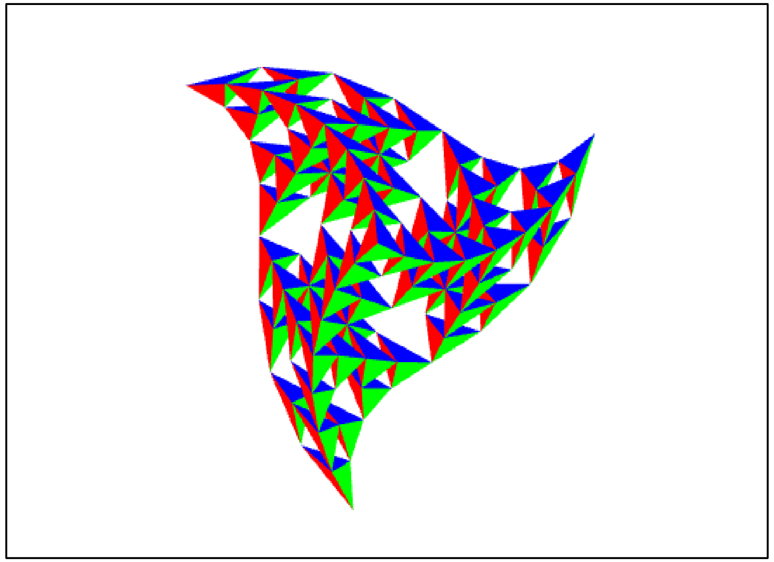
```
var canvas;
var gl;

var points = [];
var colors = [];

var NumTimesToSubdivide = 3;

var time = 0;
var dt = 1.0/60.0;
var timeLoc;
```

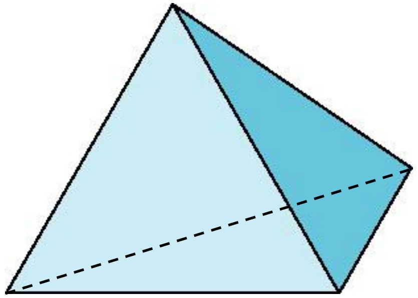


gasket6.js (2/9)

```
window.onload = function init()
{
    canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }

    //  Initialize our data for the Sierpinski Gasket

    // First, initialize the vertices of our 3D gasket
    // Four vertices on unit circle
    // Initial tetrahedron with equal length sides

    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 ) ];

    divideTetra( vertices[0], vertices[1], vertices[2], vertices[3], NumTimesToSubdivide);
```

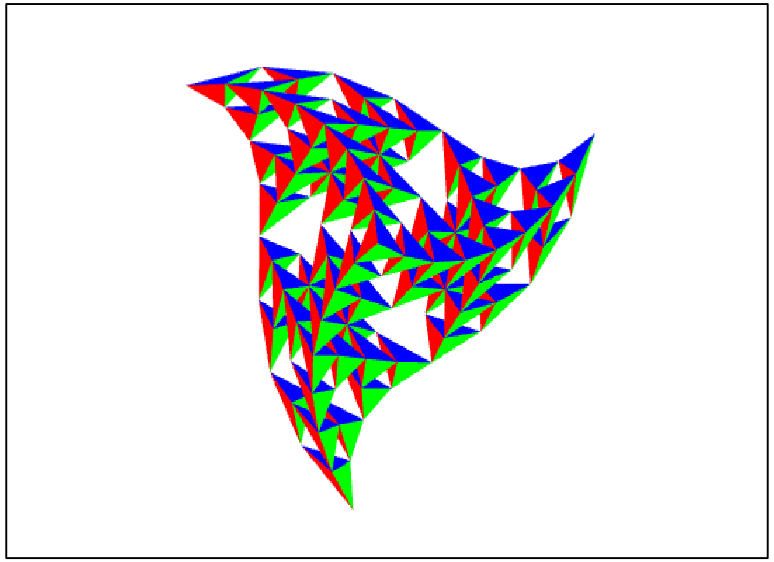
$(0.0, 0.0, -1.0)$
$(0.0, 2\sqrt{2}/3, 1/3)$
$(-\sqrt{6}/3, -\sqrt{2}/3, 1/3)$
$(\sqrt{6}/3, -\sqrt{2}/3, 1/3)$





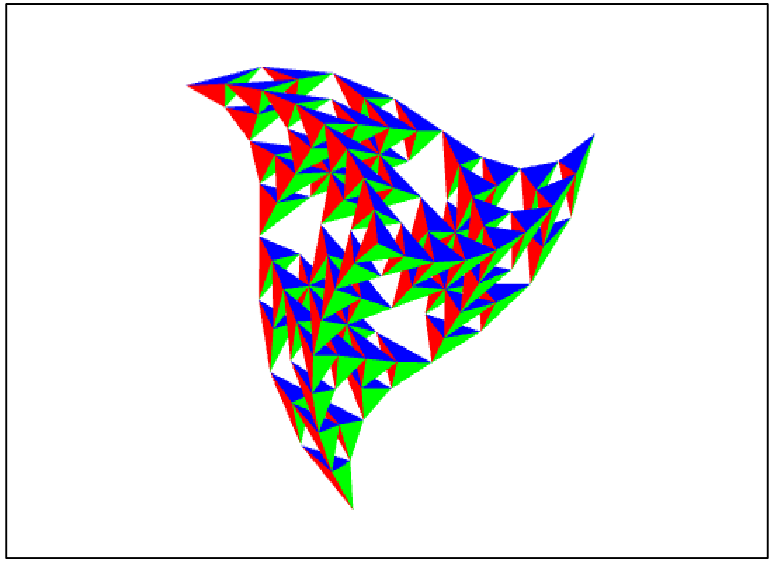
gasket6.js (3/9)

```
//
//  Configure WebGL
//
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

// enable hidden-surface removal

gl.enable(gl.DEPTH_TEST);

//  Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );
```



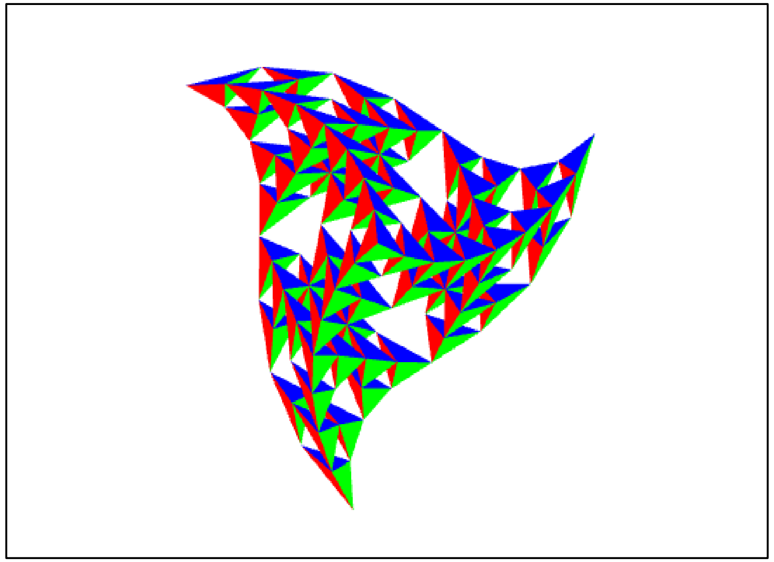
gasket6.js (4/9)

```
// Create a buffer object, initialize it, and associate it with the
//  associated attribute variable in our vertex shader

var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW );

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



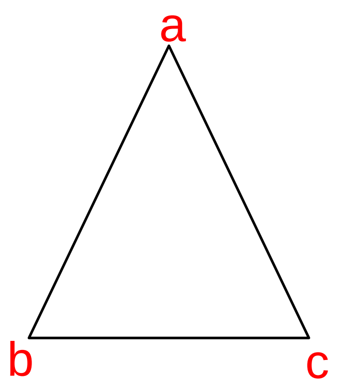
gasket6.js (5/9)

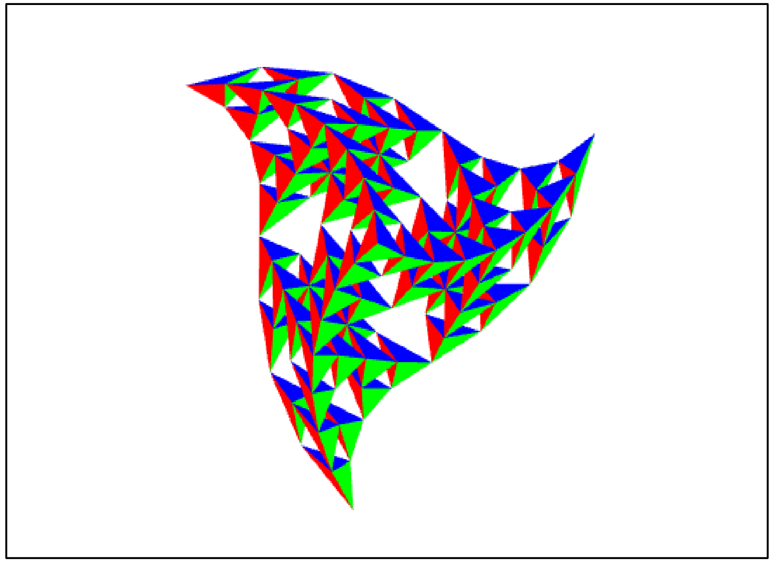
```
var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

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

timeLoc = gl.getUniformLocation(program, "time");

render();

}; // end of window.onload
```



gasket6.js (6/9)

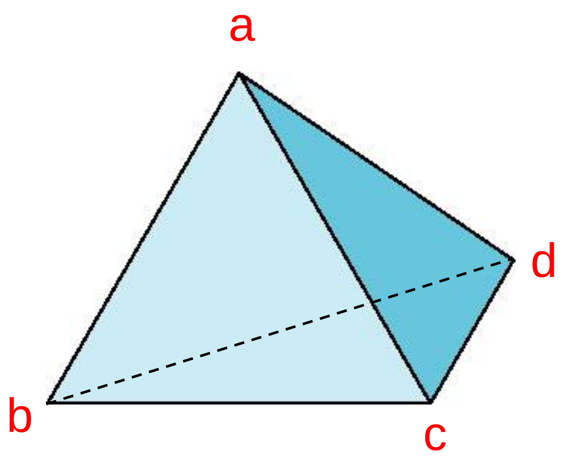
```
function triangle( a, b, c, color )
{
    // add colors and vertices for one triangle

    var baseColors = [
        vec3(1.0, 0.0, 0.0),
        vec3(0.0, 1.0, 0.0),
        vec3(0.0, 0.0, 1.0),
        vec3(0.0, 0.0, 0.0)
    ];
    colors.push( baseColors[color] );
    points.push( a );
    colors.push( baseColors[color] );
    points.push( b );
    colors.push( baseColors[color] );
    points.push( c );
}
```

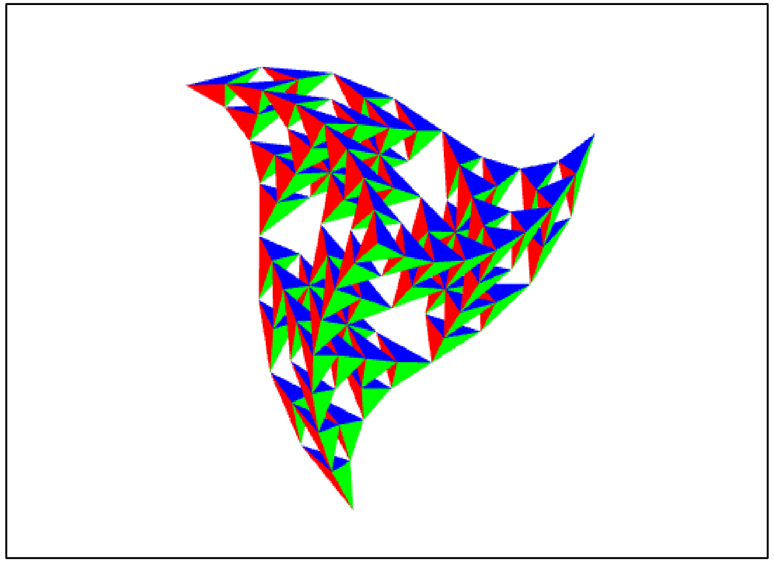




gasket6.js (7/9)

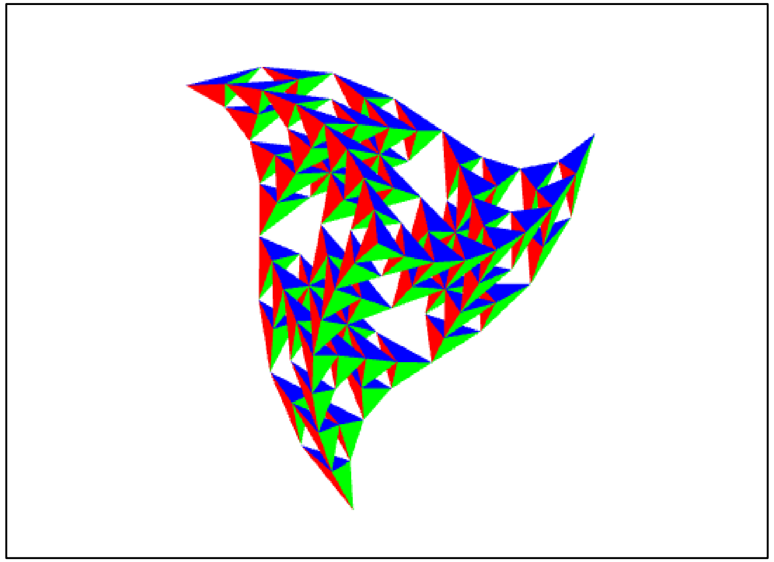
```
function tetra( a, b, c, d )
{
    // tetrahedron with each side using
    // a different color

    triangle( a, c, b, 0 );
    triangle( a, c, d, 1 );
    triangle( a, b, d, 2 );
    triangle( b, c, d, 3 );

}
```

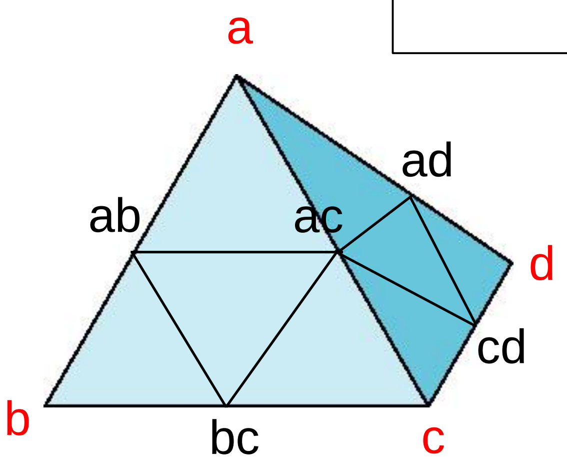




gasket6.js (8/9)

```
function divideTetra( a, b, c, d, count )
{
    // check for end of recursion
    if ( count === 0 ) { tetra( a, b, c, d ); }

    // find midpoints of sides
    // divide four smaller tetrahedra

    else {
        var ab = mix( a, b, 0.5 );  var ac = mix( a, c, 0.5 );
        var ad = mix( a, d, 0.5 );  var bc = mix( b, c, 0.5 );
        var bd = mix( b, d, 0.5 );  var cd = mix( c, d, 0.5 );
        --count;
        divideTetra(  a, ab, ac, ad, count );
        divideTetra( ab,  b, bc, bd, count );
        divideTetra( ac, bc,  c, cd, count );
        divideTetra( ad, bd, cd,  d, count );
    }
}
```





gasket6.js (9/9)

```
function render()
{
    time+=dt;
    gl.uniform1f(timeLoc, time);
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, points.length );
    requestAnimFrame(render);
}
```