CIS367 - Computer Graphics

Shaders

Erik Fredericks - frederer@gvsu.edu

Overview

Basics of shaders

GLSL and shader programming

BUT FIRST

AN IN-CLASS ASSIGNMENT AAHHHHHH

How would you draw this with what you've learned so far

Shaders?

Vertex movement

- Fractal generation
- Vertex morphing

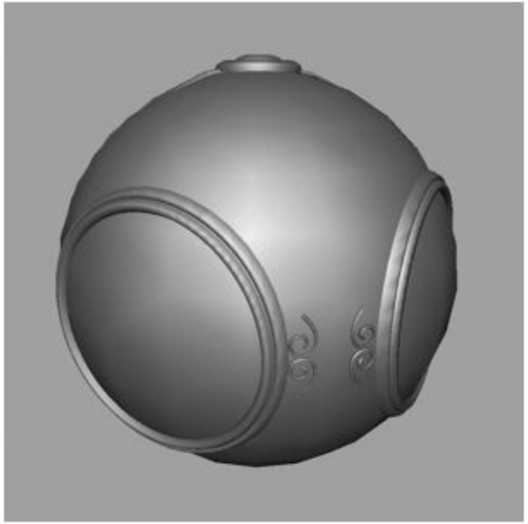
Lighting

- Realistic vs cartoon modeling

per vertex lighting

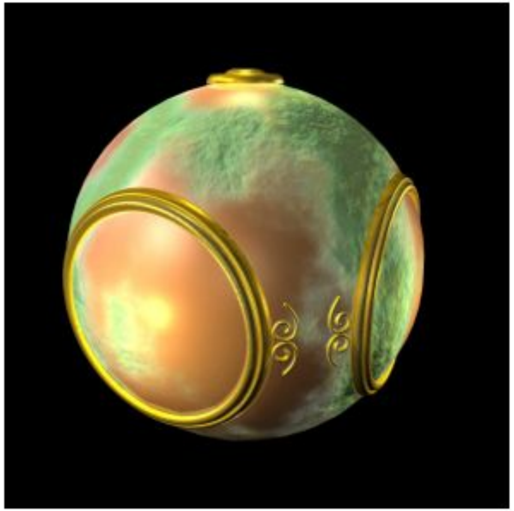
per fragment lighting

Texture mapping



smooth shading

environment mapping



bump mapping

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 - GLSL

- OpenGL 2.0 and up
- C-like
- Includes matrices, vectors, samplers

OpenGL 3.1 → shaders required!

Simple Vertex Shader

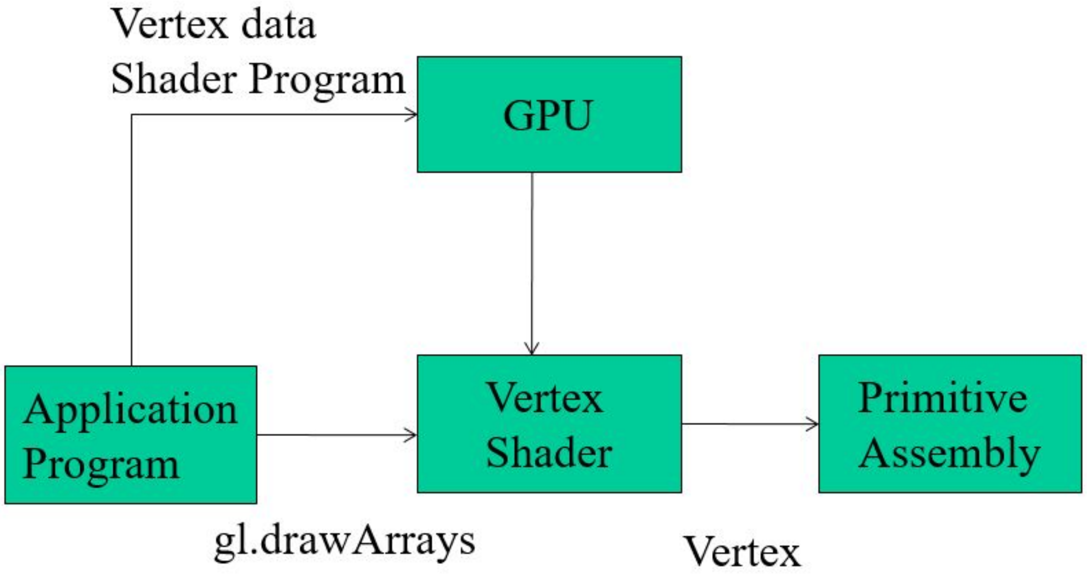
input from application

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

must link variable to application

built-in variable

Execution Model

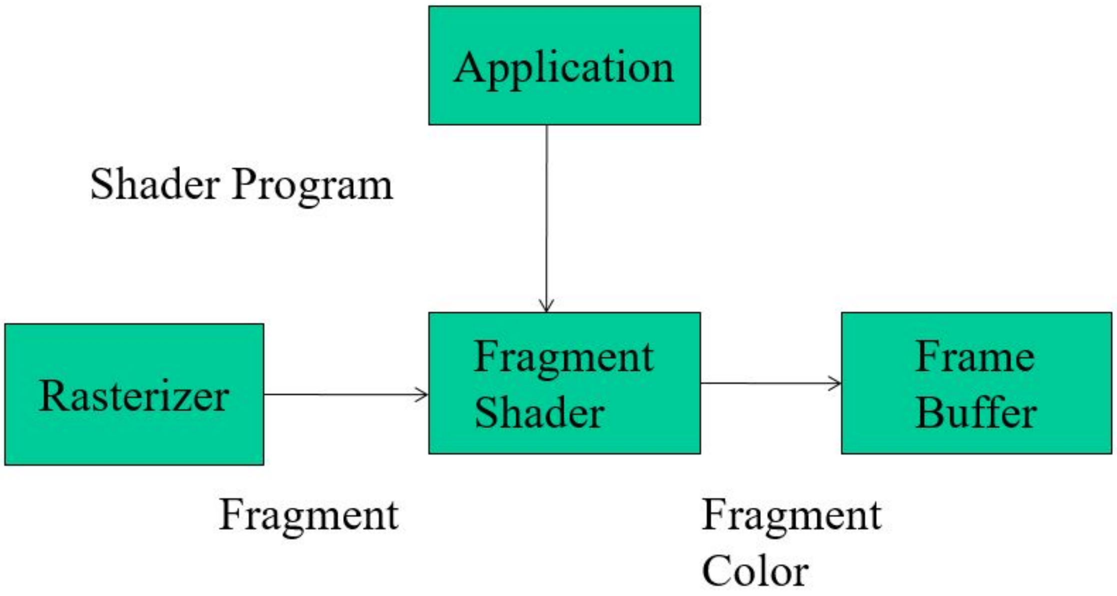


Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015

Simple Fragment Program

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

Execution Model (with Fragment Color)



Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015

Data Types

C: `int, float, bool`

Vectors:
```
float vec2, vec3, vec4
int (ivec), boolean (bvec)
```

Matrices: `mat2, mat3, mat4`

- Column storage
- Standard referencing ➔ m[row][column]

C++ constructors:
```
vec3 a = vec3(1.0, 2.0, 3.0);
vec2 b = vec2(a);
```

Pointers?

Nope!

C structs can be copied back/forth from functions

- Matrices/vectors are basic types

```
mat3 func1(mat3 a)
{
  ...
}
```

Qualifiers

Similar to C qualifiers, but with caveats

- Namely, to support execution model (shader model)

Variables change:

- Once per primitive
- Once per vertex
- Once per fragment
- Any time during application

Vertex attributes interpolated by rasterizer into fragment attributes

Attribute Qualifier

Change **at most** once per vertex

User defined (in application)

- `attribute float temperature`
- `attribute vec3 velocity`

Recent GLSL versions use `in` and `out` to get to/from shaders

Uniform Qualifier

Variables constant for **entire primitive**

Can change in application and be sent to shader

- Cannot be changed in shader

E.g., pass information to shader (time / bounding box) of primitive or transformed matrices

Varying Qualifier

Variables passed from vertex to fragment shader

- Automatically interpolated by rasterizer

Uses varying qualifier for both shaders

- `varying vec4 color;`

Recent versions of GLSL use `out` in vertex shader and `in` for fragment shader

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

Naming Convention

Helpful to follow (especially given various places you can write code...)

Passed to vertex shader start with **v** in application and shader

- `vPosition, vColor`
    - Different entities with the same name

Varying variables begin with **f** in both shaders

- `fColor`
    - Must have same name

Uniform variables have no specific convention -- same name in application and shaders

Vertex Shader Example

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

Fragment Shader

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

Send Color 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 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;
}
```

Shading a triangle

Make a copy of triangle.js/triangle.html

Rename them as triangle-shade.js / triangle-shade.html

Let's add some shading!

(1) We need to update our shaders and (2) add a new method for sending colors from the application to the shader!

Operators and Functions

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix}$$

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 (rearrange vectors)

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

Swizzling

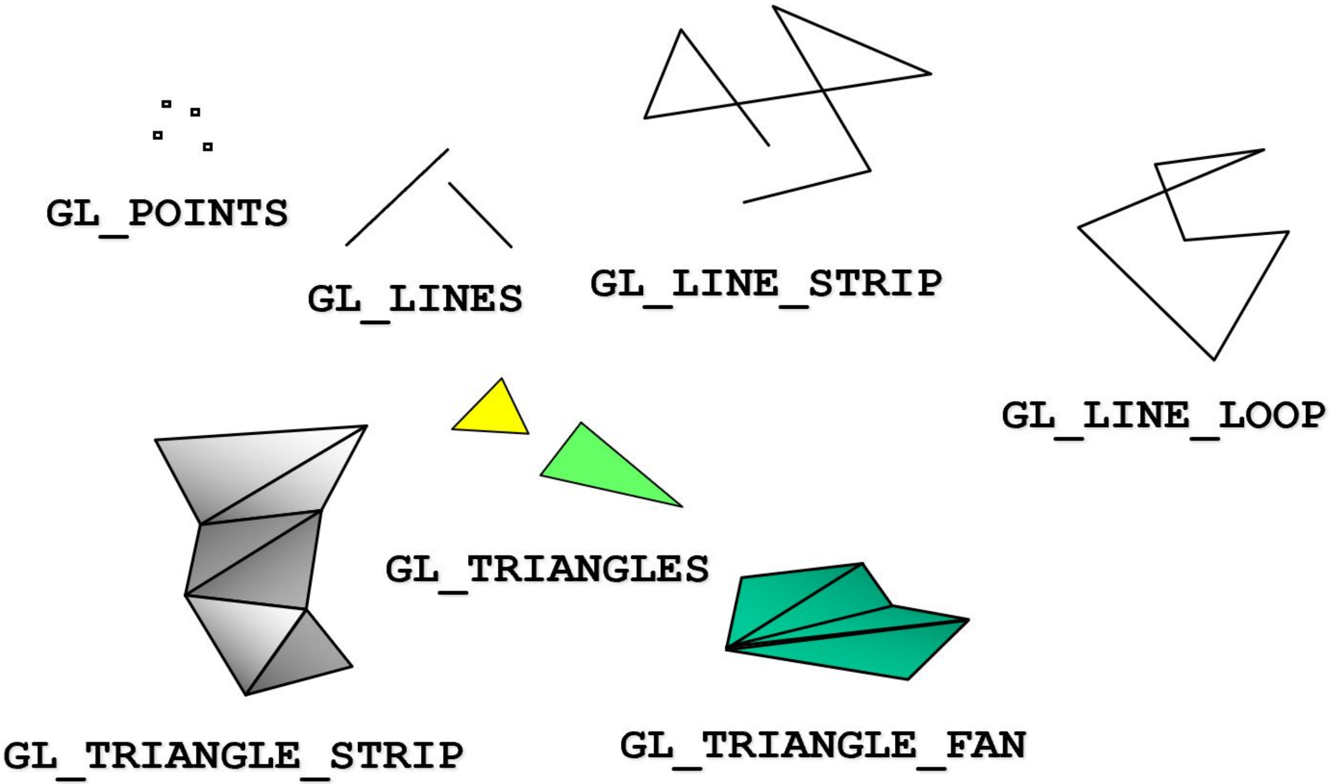
A = {1 2 3 4}

$$A.wwxy = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \\ 1 \\ 2 \end{bmatrix}$$

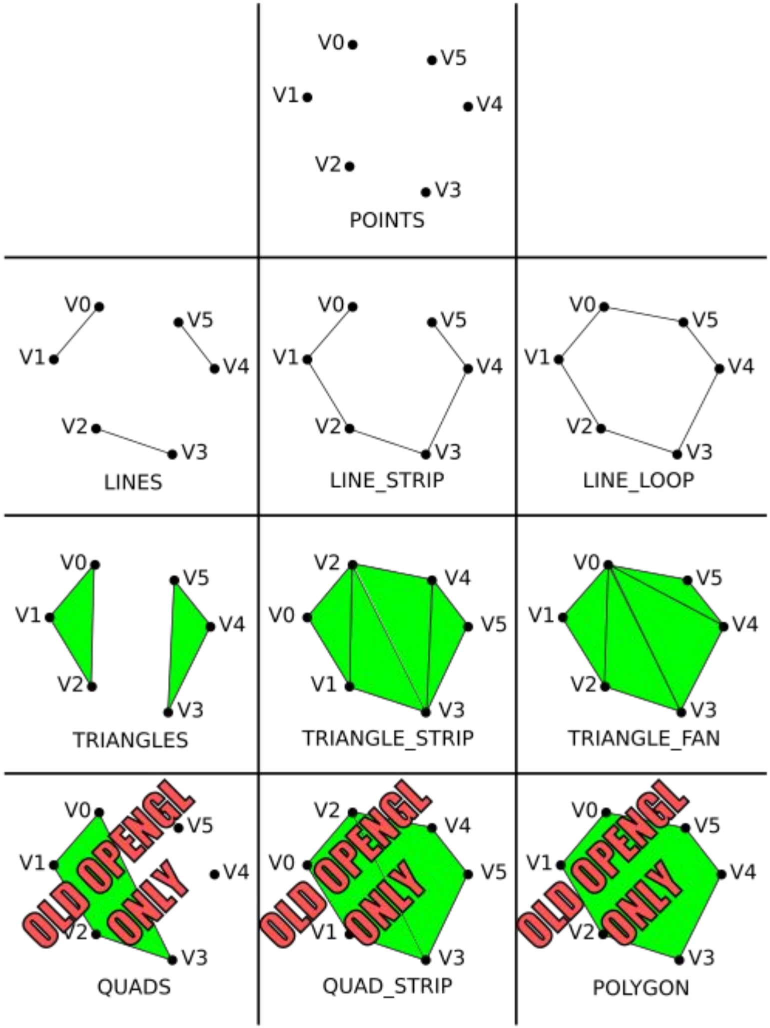
And now...

Primitives, color, attributes

WebGL Primitives



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



Polygon problems

WebGL only displays triangles

- ***Simple***: edges **cannot** cross
- ***Convex***: all points on line segment between two points in polygon **are also in** polygon
- ***Flat***: all vertices in same plane

Application program (your JS file) must tesselate polygon into triangles (triangulation)

OpenGL4.1 contains a tesselator, but WebGL does not!

non-simple polygon

non-convex polygon

Polygon Testing

Conceptually, simple to test for simplicity and convexity

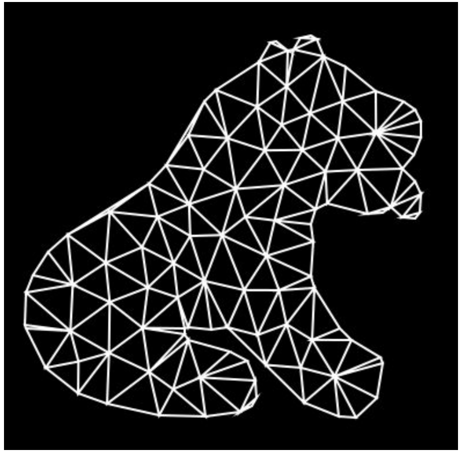
- Time consuming though!

Older versions of OpenGL left testing to application

Present version only renders triangles

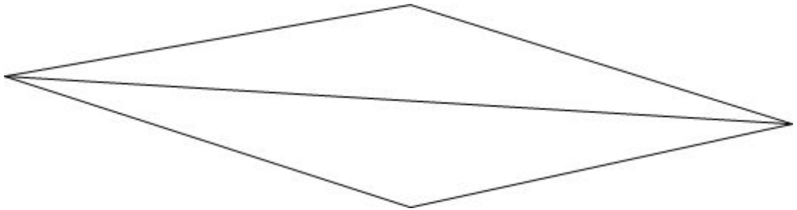
- Triangulation algorithm required instead!

https://mapbox.github.io/delaunator/demo.html



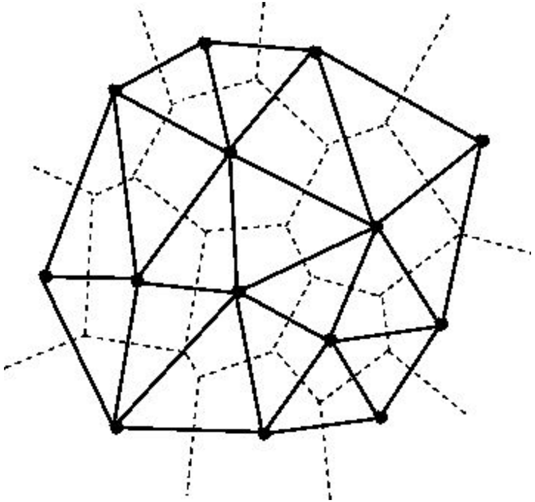
Good/Bad Triangles

Triangles that are **long** and **thin** tend to render poorly



Equilateral triangles render well

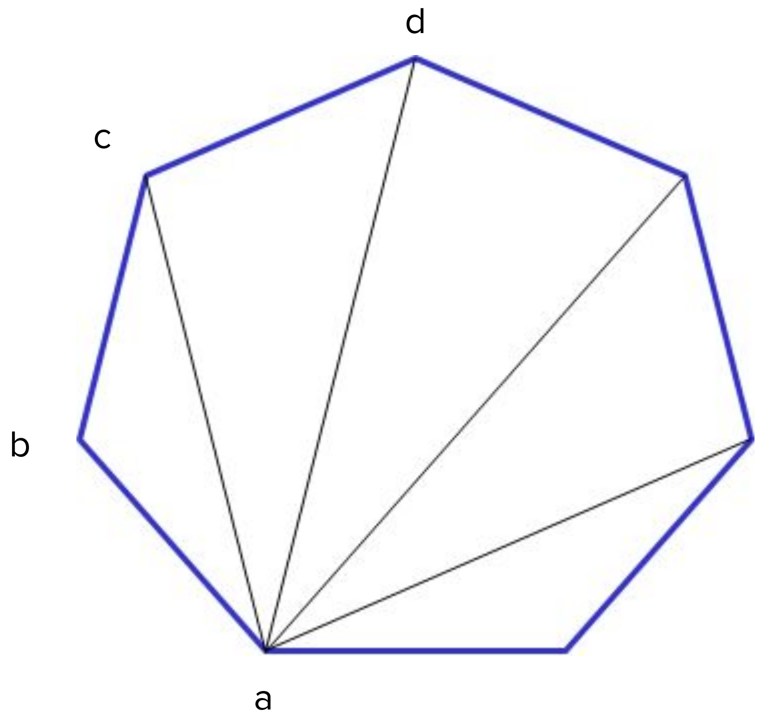
- Maximize minimum angle
- Various algorithms we'll get to for triangularization



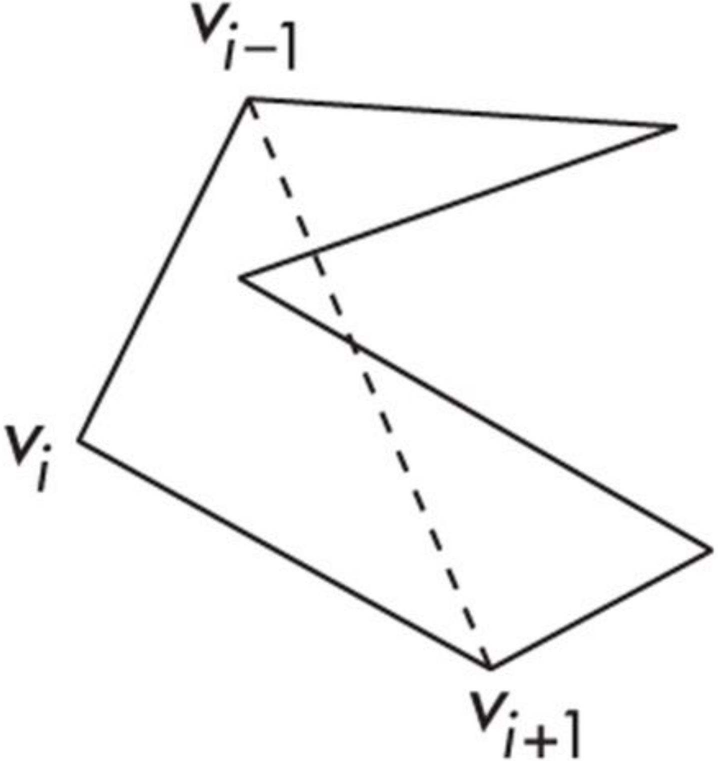
Triangularization

Convex polygons

- Start with triangle **abc**
  - Remove **b**
  - Then **acd** ...

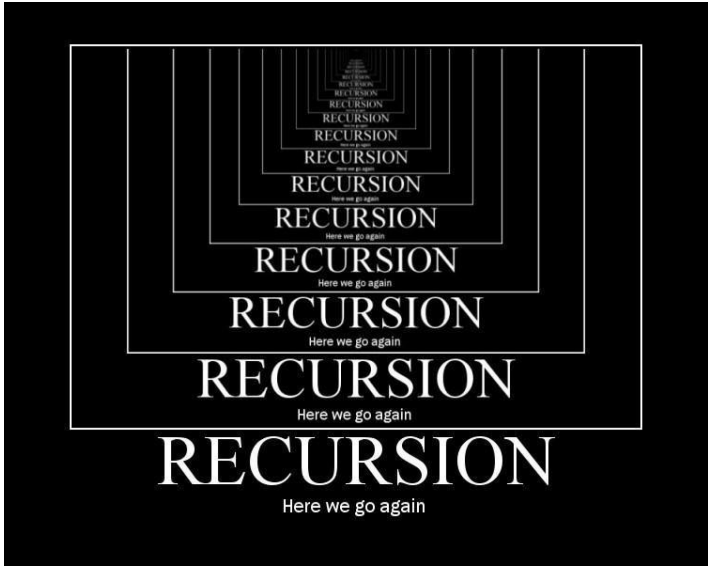


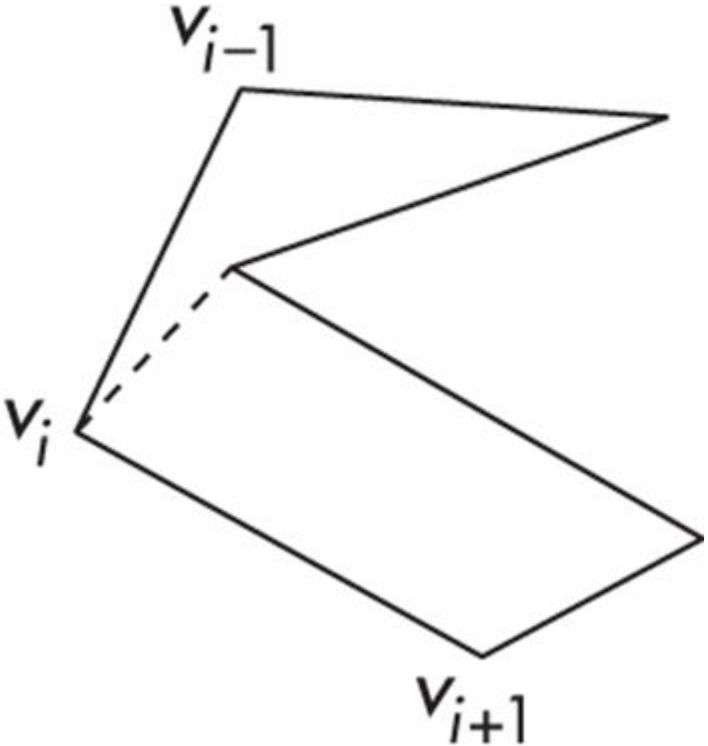
Non-convex (concave)



Recursive Division

Find leftmost vertex and split





Attributes

Determine object **appearance**

- Color (points, lines, polygons)
- Size/width (points, lines)
- Stipple pattern (lines, polygons)
- Polygon mode
  - Display as filled (solid or stipple)
  - Display edges
  - Display vertices

Only a few are supported by WebGL (e.g., gl_PointSize)

Can change **per vertex!**

Now, let's talk more GLSL

Things we care about here (right now anyway)

- Coupling shaders to applications (read/compile/link)
- Attributes
- Uniform variables

Linking shaders with application

Steps!

- 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?

Shader container!

- Can contain multiples
  - And other GLSL functions

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

gl.attachShader( program, vertex_shader );
gl.attachShader( program, fragment_shader );
gl.linkProgram( program );
```

Reading shaders

Added to program object and **compiled**

- Strange to say in the HTML world...

Multiple methods for reading/passing to program object

- Use null-terminated string (write all your code in a string)
  - Messy!
- Store in HTML and load with `getElementById`
  - We'll effectively use this
- Store in file and read into an object
  - May have issues with browser security!

Adding a **vertex** shader

Note: our setup makes this process a lot easier!

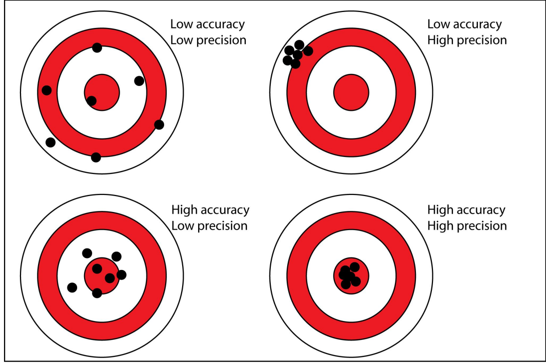
```
var vertex_shader;
ver vertex_element = document.getElementById( vertex_shader_id );

vertex_shader = gl.createShader( gl.VERTEX_SHADER );

gl.shaderSource( vertex_shader, vertex_element.text );
gl.compileShader (vertex_shader );

// after program object created…
gl.attachShader( program, vertex_shader );
```

Added a vertex shader (reader)

Again, this may be a security problem (cross-site request)

```
function getShader( gl, shader_name, type ) {
  var shader = gl.createShader( type );
  shader_script = loadFileAJAX( shader_name );
  if (!shader_script) {
    alert("Source not found: " + shader_name);
  }
}
```

Precision



Must specify type precision in **fragment shader** for GLSL

- Inherited from OpenGL ES
    - Must run on embedded systems that might not support 32 bit floats!
        - (less of an issue these days, but who knows)
- All implementations must support `mediump`
- No default for float in fragment shader!

We *can* use preprocessor directives (`#ifdef`) to check if high precision (`highp`) is supported

- Default to `mediump` if not

Precision / pass-through fragment shader example

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

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

BACK TO SIERPINSKI

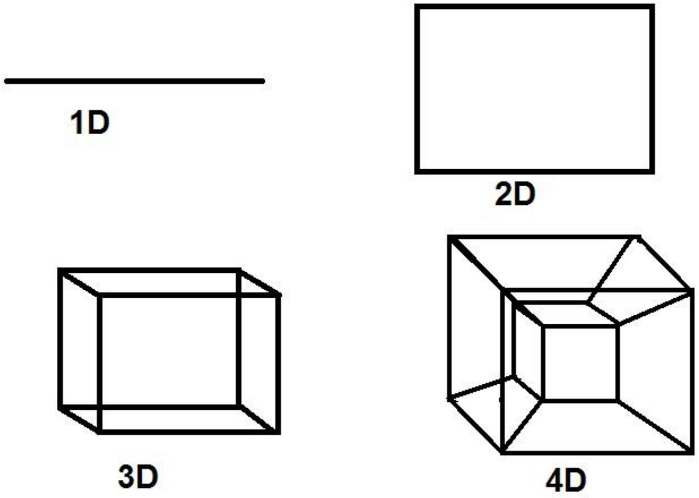
- Let's talk about the algorithm itself, changing it up a bit (the different flavors of gasket*n*.js), moving to 3D, and hidden surfaces

3D?

Just another parameter!

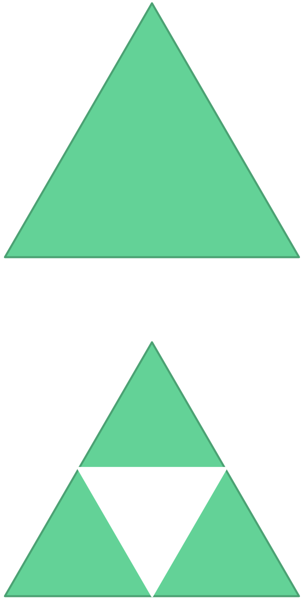
Instead of `vec2`, use `vec3`
Instead of `gl.uniform2f`, use `gl.uniform3f`

But, have to worry about rendering order for primitives and hidden surfaces



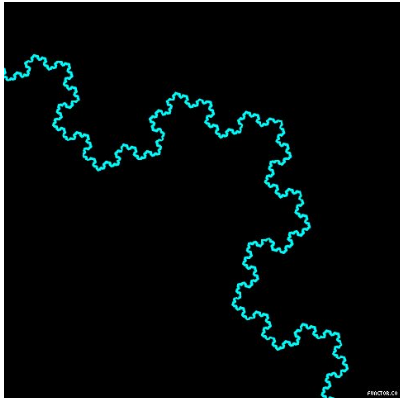
Sierpinski Gasket

Recursively drawn **fractal**



Connect bisectors of sides and remove central triangle

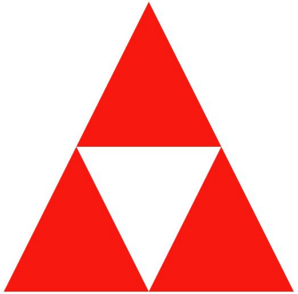
Repeat until done



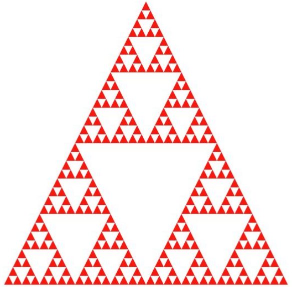
another fractal

Sierpinski with *n* subdivisions

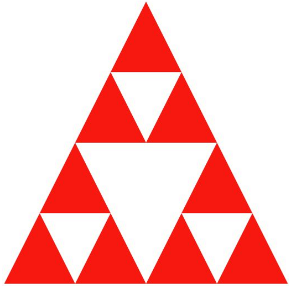
n = 1



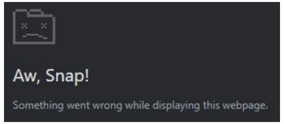
n = 5



n = 2



n = 100



Gasket as fractal

Consider: filled area (red) and perimeter (length of lines around filled triangles)

As we subdivide:

- Area goes to zero
- Perimeter goes to infinity!

Not an ordinary geometric object

- Neither 2 nor 3 dimensional

This is a fractal (fractional dimension) object

So our gasket program then (from WebGL)

HTML file (gasket*n*.html)

- Same as all other examples so far!
  - Pass-through vertex shader
  - Fragment shader sets the color
  - Reads in the application (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 );
```

Gasket program cont'd (draw 1 triangle)

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

Gasket program - 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 );
  }
}
```

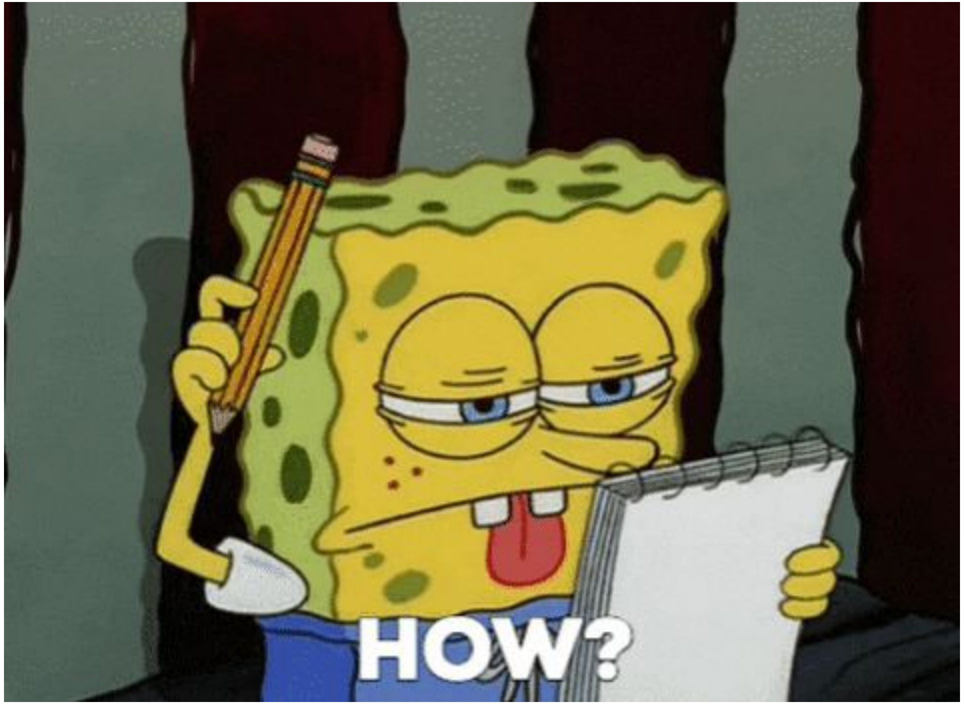
Gasket program -- init

```
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();
```

Gasket program -- render

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

But … what about 3 dimensions?



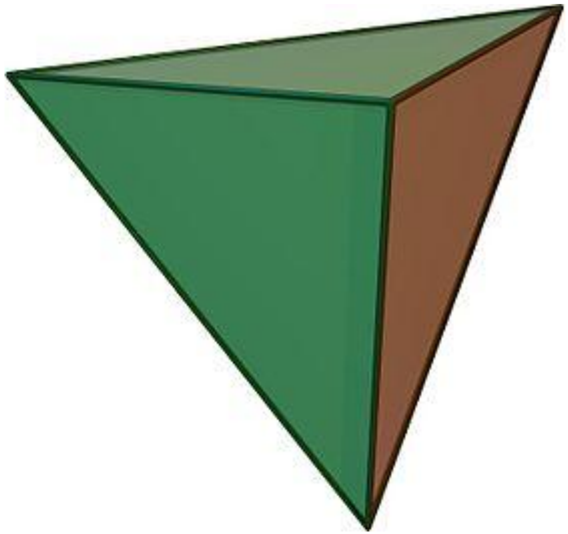
Moving to 3D

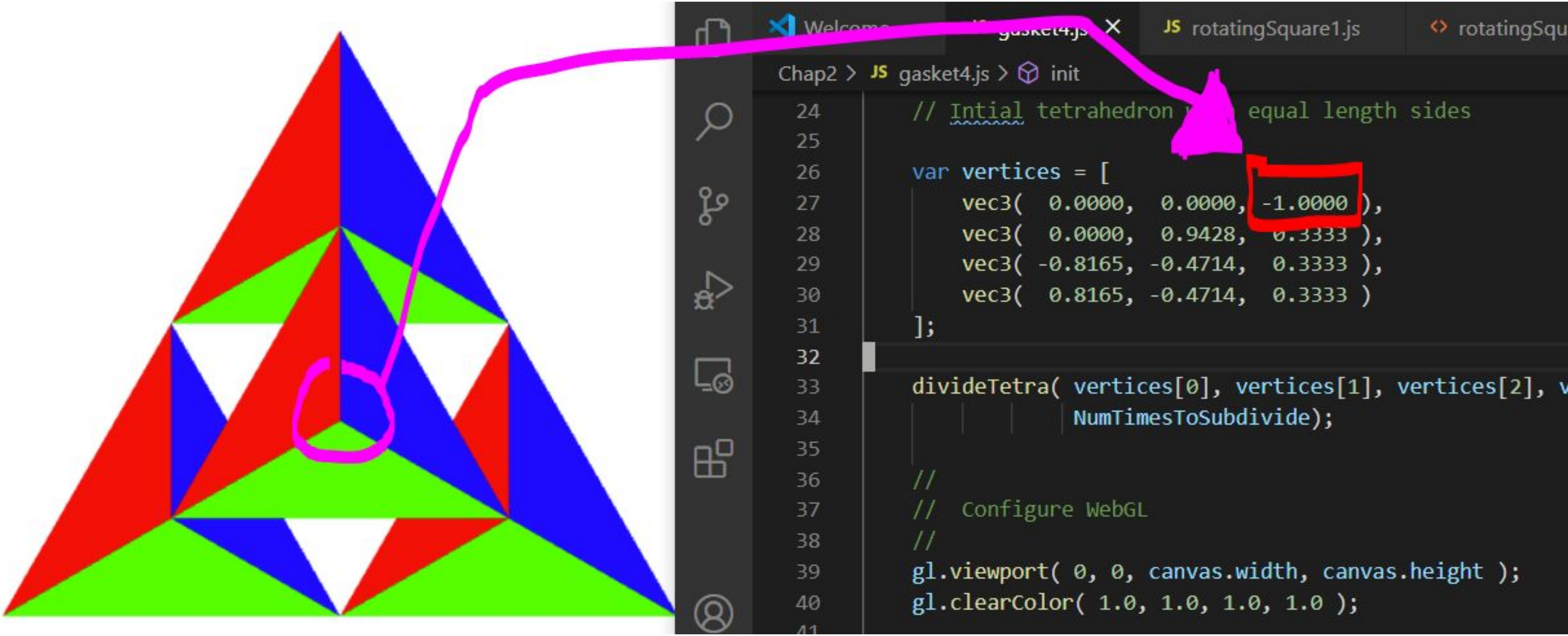
Use 3 dimensional points
(e.g., vec3)

Start with tetrahedron instead!

```
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 )
];
```

Then we sub-divide each *face*

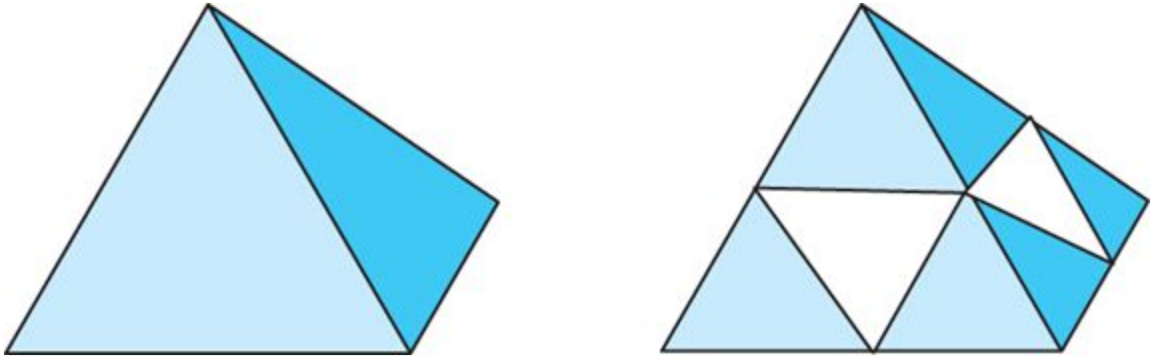




rotatingSquare1.js
Chap2 > JS gasket4.js > init
// Intial tetrahedron
equal length sides
var vertices = [
vec3(0.0000, 0.0000, -1.0000),
vec3(0.0000, 0.9428,
vec3(-0.8165, -0.4714, 0.3333),
vec3(0.8165, -0.4714, 0.3333)
];
divideTetra(vertices[0], vertices[1], vertices[2],
NumTimesToSubdivide);
//
// Configure WebGL
//
gl.viewport(0, 0, canvas.width, canvas.height);
gl.clearColor(1.0, 1.0, 1.0, 1.0);

3D gasket

Subdivide each of the 4 faces



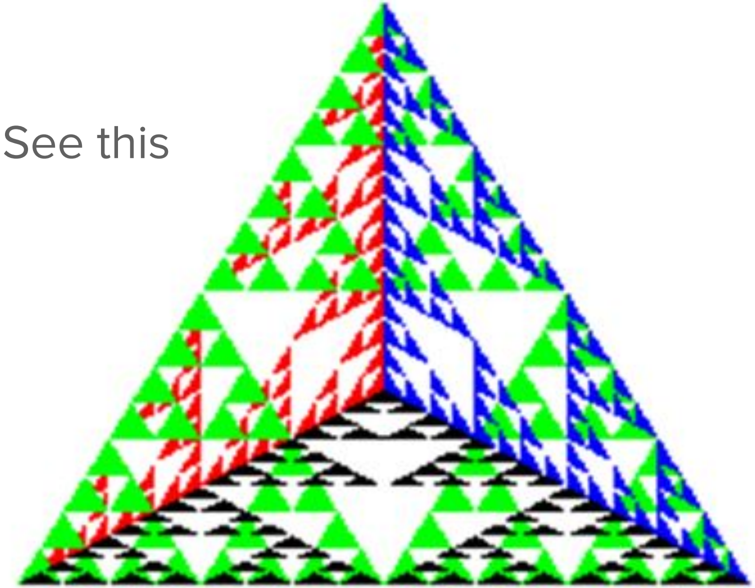
Looks like a solid tetrahedron removed from center!

- Code nearly identical to 2D example … nice!

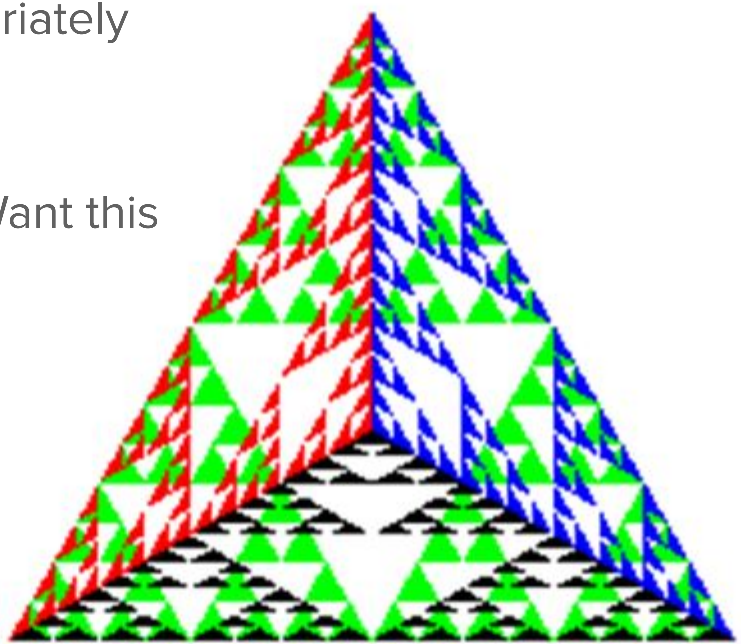
Triangles drawn *in order!*

- Front triangles not always rendered appropriately

See this



Want this



Hidden surface removal

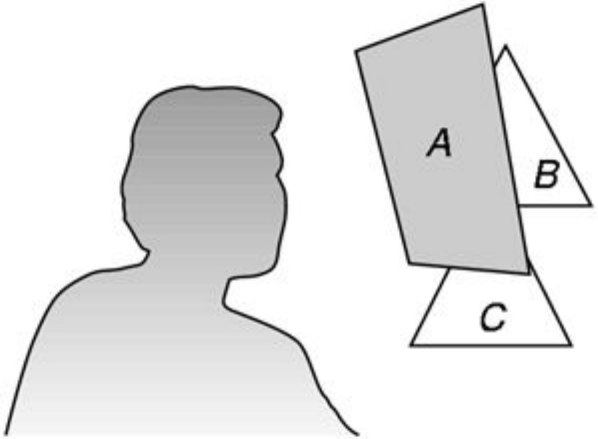
Only view surfaces in front of other surfaces

OpenGL uses the *z-buffer* algorithm (hidden surface method) that **saves depth information** as objects are rendered

- Stored as part of pipeline

For WebGL:

- Enable it:
    - `gl.enable(GL.DEPTH_TEST);`
- Clear for each render:
    - `gl.clear(GL.COLOR_BUFFER_BIT |`
    - `GL.DEPTH_BUFFER_BIT);`



Volume subdivision