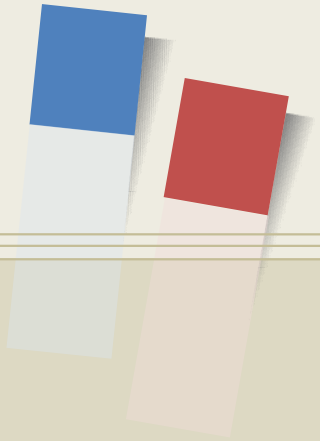


Graphics

Dept. of Software, Gachon University

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

gl.drawArrays & WebGLPrimitives

gl.drawArrays(primitive, first, count)

- The WebGLRenderingContext.drawArrays() method of the WebGL API renders primitives from array data.

Syntax

```
void gl.drawArrays(mode, first, count);
```

Parameters

mode

A `GLenum` specifying the type primitive to render. Possible values are:

- `gl.POINTS`: Draws a single dot.
- `gl.LINE_STRIP`: Draws a straight line to the next vertex.
- `gl.LINE_LOOP`: Draws a straight line to the next vertex, and connects the last vertex back to the first.
- `gl.LINES`: Draws a line between a pair of vertices.
- `gl.TRIANGLE_STRIP`
- `gl.TRIANGLE_FAN`
- `gl.TRIANGLES`: Draws a triangle for a group of three vertices.

first

A `GLint` specifying the starting index in the array of vector points.

count

A `GLsizei` specifying the number of indices to be rendered.

Return value

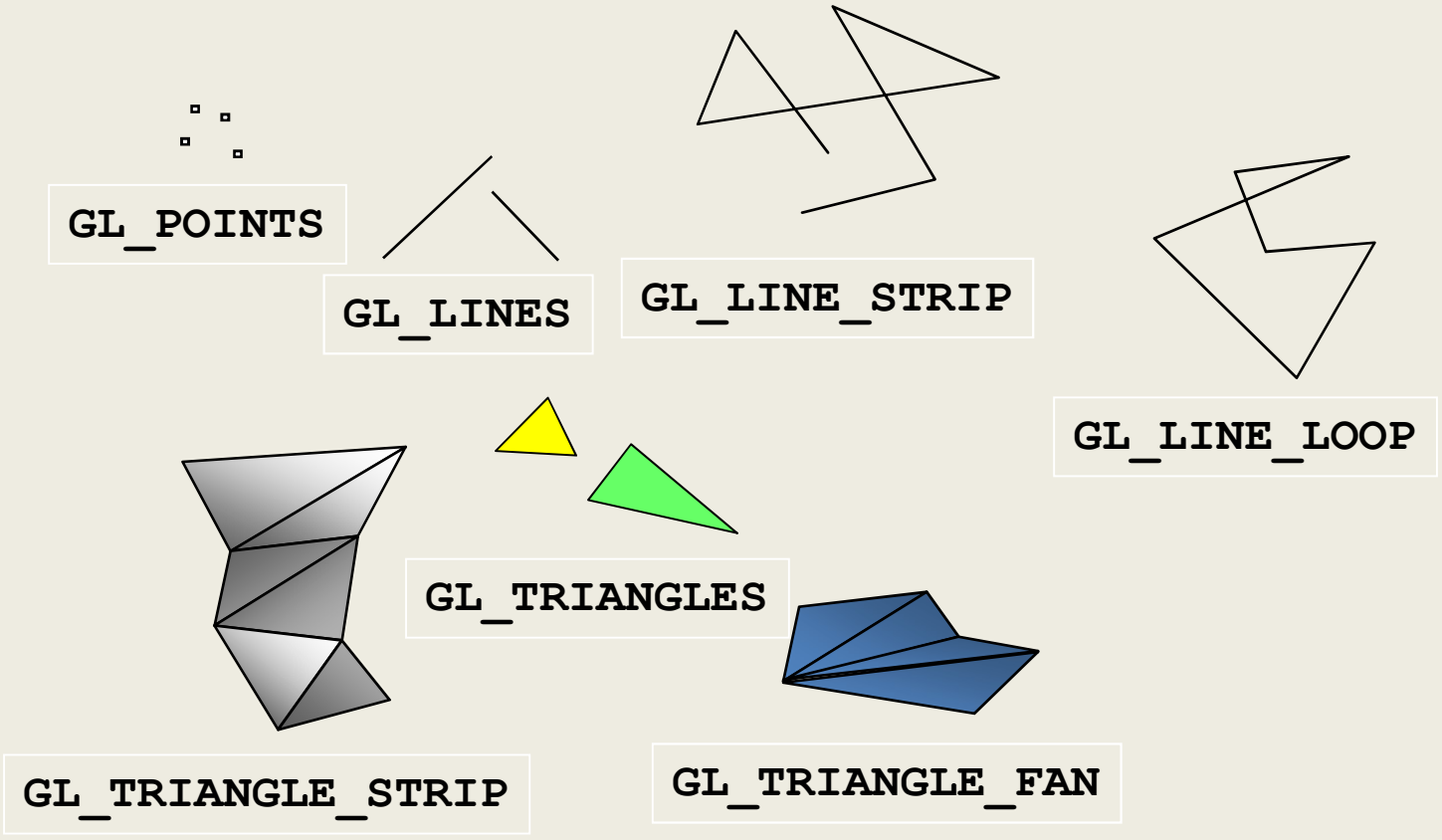
None.

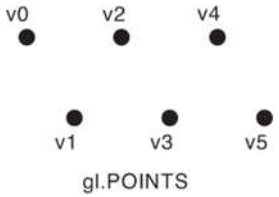
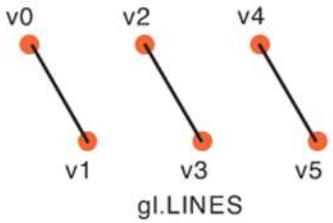
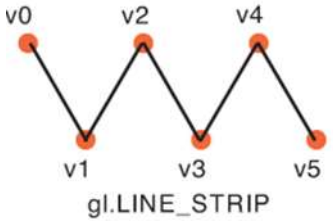
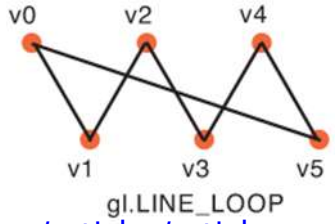
Exceptions

- If `mode` is not one of the accepted values, a `gl.INVALID_ENUM` error is thrown.
- If `first` or `count` are negative, a `gl.INVALID_VALUE` error is thrown.
- if `gl.CURRENT_PROGRAM` is `null`, a `gl.INVALID_OPERATION` error is thrown.

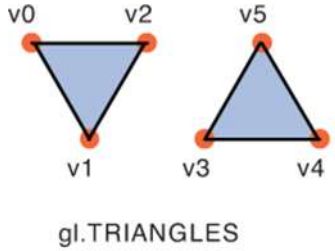
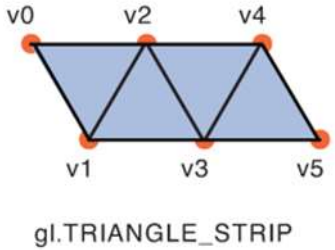
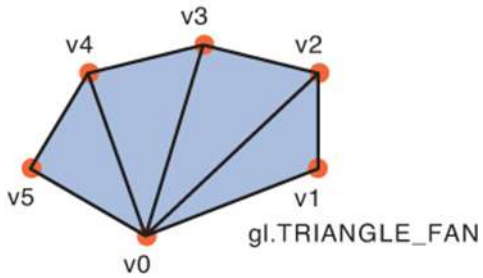
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/drawArrays

WebGLPrimitives



Basic Shape	Mode	Description
Points	gl.POINTS 	A series of points. The points are drawn at v0, v1, v2 …
Line segments	gl.LINES 	A series of unconnected line segments. The individual lines are drawn between vertices given by (v0, v1), (v2, v3), (v4, v5)… If the number of vertices is odd, the last one is ignored.
Line strips	gl.LINE_STRIP 	A series of connected line segments. The line segments are drawn between vertices given by (v0, v1), (v1, v2), (v2, v3), … The first vertex becomes the start point of the first line, the second vertex becomes the end point of the first line and the start point of the second line, and so on. The *i*-th (*i*> 1) vertex becomes the start point of the *i*-th line and the end point of the *i-1*-th line. (The last vertex becomes the end point of the last line.)
Line loops	gl.LINE_LOOP 	A series of connected line segments. In addition to the lines drawn by gl.LINE_STRIP, the line between the last vertex and the first vertex is drawn. The line segments drawn are (v0, v1), (v1, v2), …, and (v *n*, v0). v *n* is the last vertex.

http://www.informit.com/articles/article.aspx?p=2111395&seqNum=2

Basic Shape	Mode	Description
Triangles	gl.TRIANGLES 	A series of separate triangles. The triangles given by vertices (v0, v1, v2), (v3, v4, v5), … are drawn. If the number of vertices is not a multiple of 3, the remaining vertices are ignored.
Triangle strips	gl.TRIANGLE_STRIP 	A series of connected triangles in strip fashion. The first three vertices form the first triangle and the second triangle is formed from the next vertex and one of the sides of the first triangle. The triangles are drawn given by (v0, v1, v2), (v2, v1, v3), (v2, v3, v4) … (Pay attention to the order of vertices.)
Triangle fans	gl.TRIANGLE_FAN 	A series of connected triangles sharing the first vertex in fanlike fashion. The first three vertices form the first triangle and the second triangle is formed from the next vertex, one of the sides of the first triangle, and the first vertex. The triangles are drawn given by (v0, v1, v2), (v0, v2, v3), (v0, v3, v4), …

Example: gl.drawArrays(gl.TRIANGLES, 0, 3);

- In this case, the three vertices in the buffer object are no longer individual points, but become three vertices of a triangle.
- WebGL can draw only three types of shapes: a point, a line, and a triangle.
- However, spheres to cubes to 3D monsters to humanoid characters in a game can be constructed from small triangles. Therefore, you can use these basic shapes to draw anything.

http://www.informit.com/articles/article.aspx?p=2111395&seqNum=2

Drawing triangles to draw a rectangle.

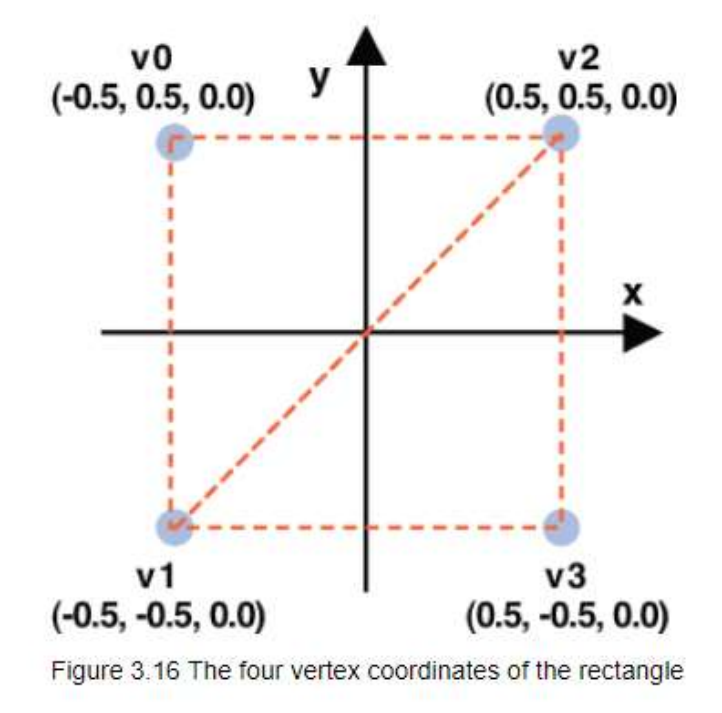


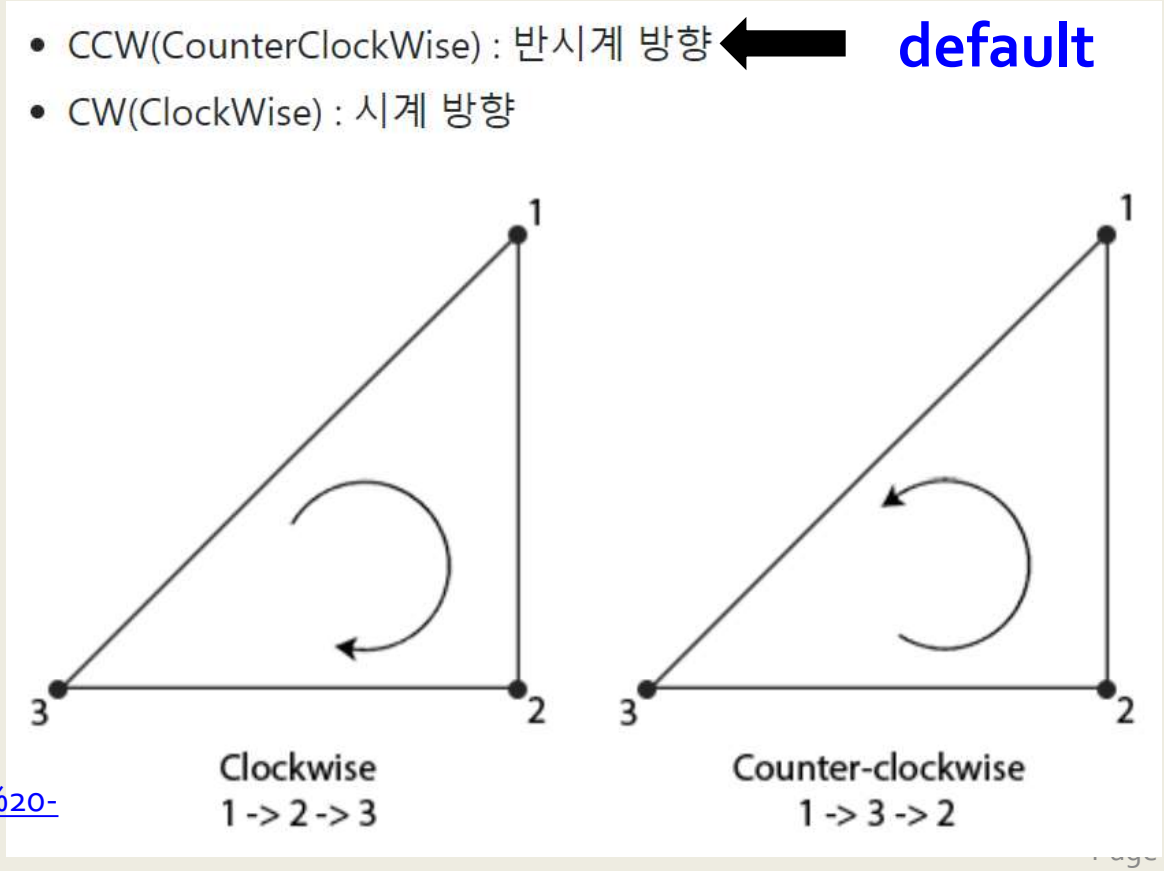
Figure 3.16 The four vertex coordinates of the rectangle

- WebGL cannot draw a rectangle directly, so you need to divide the rectangle into two triangles (v0, v1, v2) and (v2, v1, v3) and then draw each one using gl.TRIANGLES, gl.TRIANGLE_STRIP, or gl.TRIANGLE_FAN.
- In this example, you'll use gl.TRIANGLE_STRIP because it only requires you to specify four vertices.
- If you were to use gl.TRIANGLES, you would need to specify a total of six.

http://www.informit.com/articles/article.aspx?p=2111395&seqNum=2

Winding Order

- When you draw a triangle, you need three vertices, and there is a sequence (or direction) in which you draw the triangle. The order (or direction) in which a triangle is drawn is divided into two.



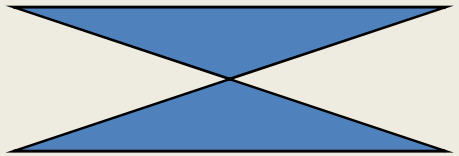
https://github.com/HomoEfficio/dev-tips/blob/master/WebGL%20-%2003%20-%20Basic%20Concepts.md

Why Is Winding Order Important?

- This is because the Winding Order determines whether a triangle is forward or backward, and if it is backward, the triangle is removed during the rasterizing phase to reduce the computational load.

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

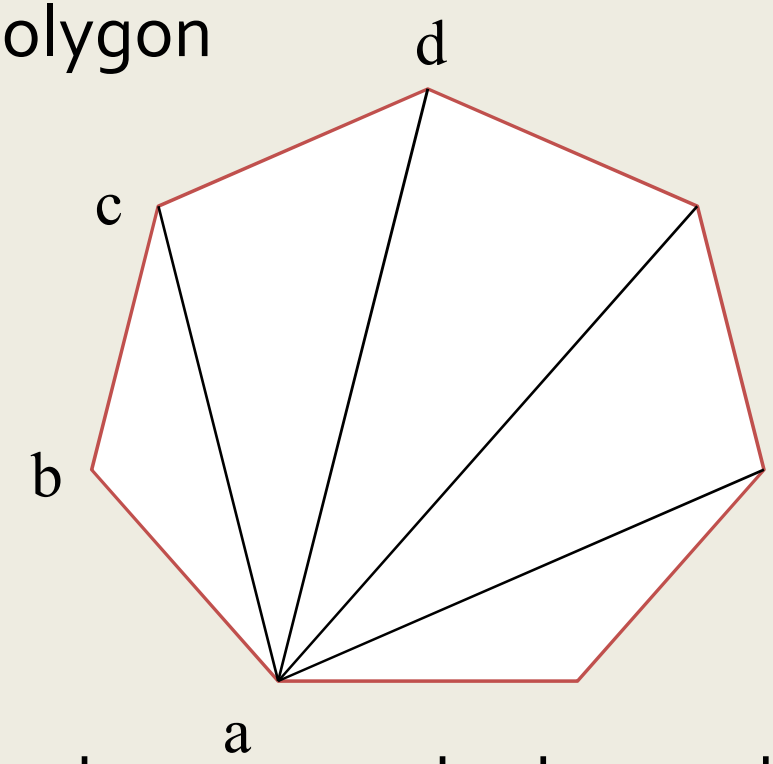


nonsimple polygon

nonconvex polygon

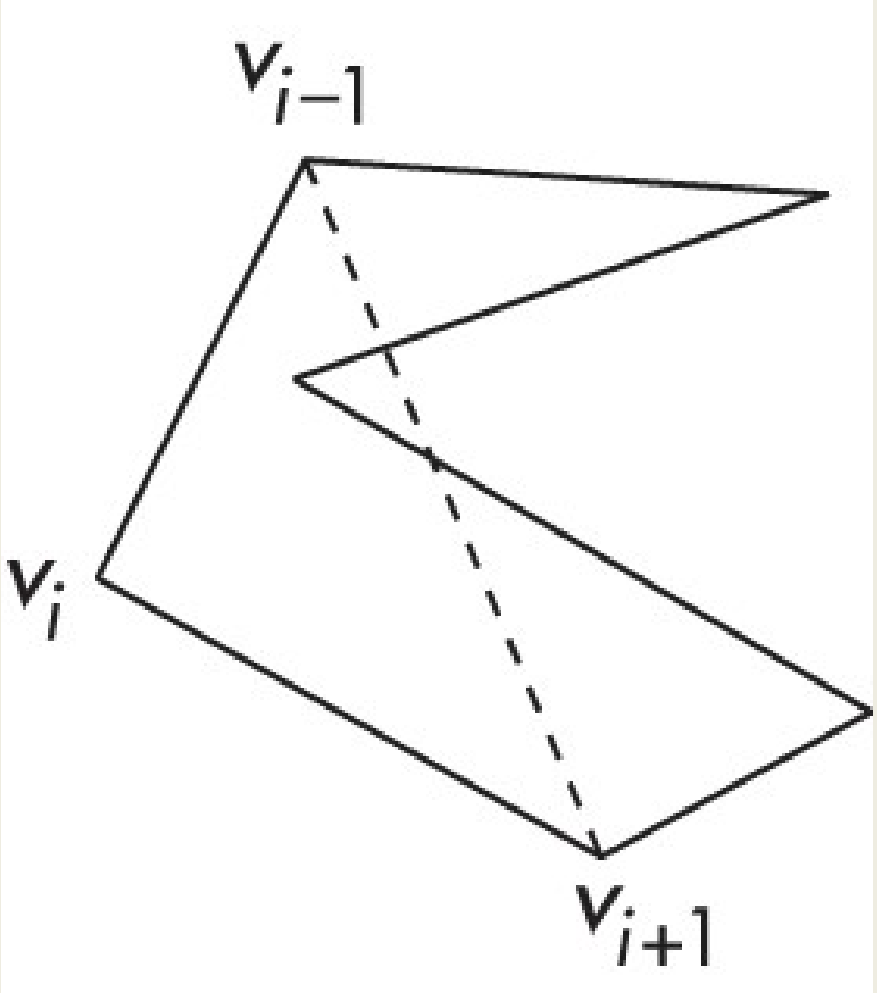
Triangularization

- Convex polygon



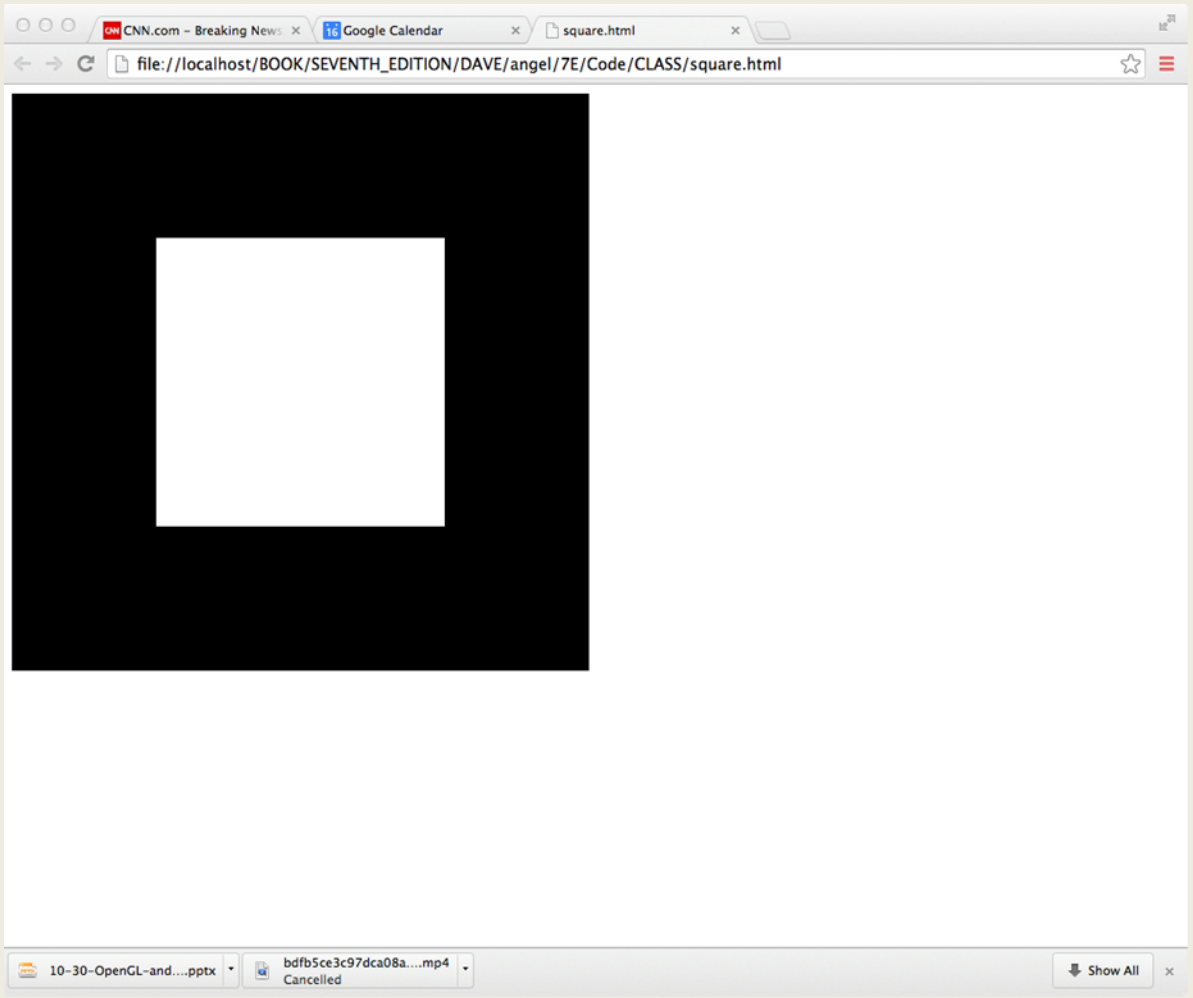
- Start with abc, remove b, then acd, ….

Non-convex (concave)



Programming with WebGL: Square Program

Square Program



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>
```

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>
```

square.js

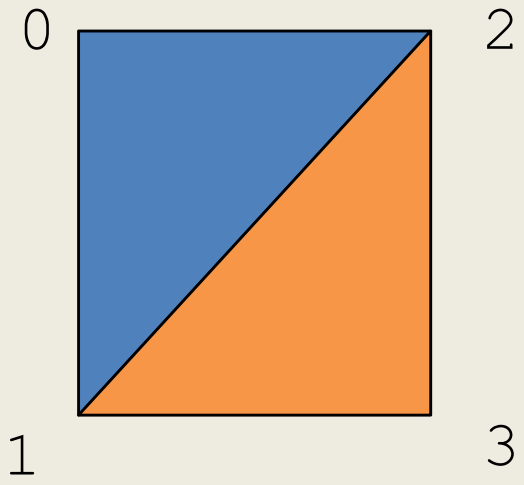
```
var gl;
var points;

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 ), // v0
        vec2( -0.5, -0.5 ), // v1
        vec2( 0.5, 0.5 ), // v2
        vec2( 0.5, -0.5) // v3
    ];
```

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 );
```

square.js (cont)

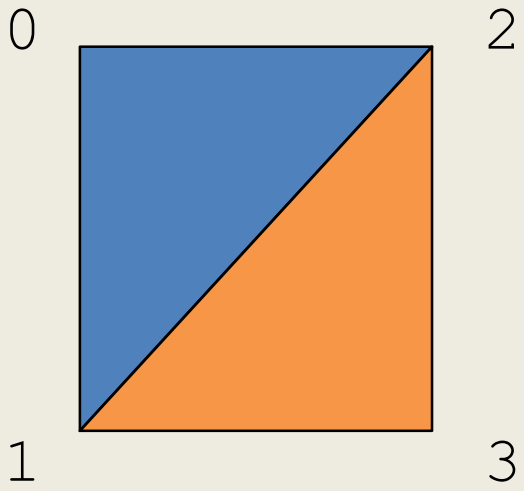
```
    render();
};

function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 ); // 0, 1, 2, 2, 1, 3
}
```



Triangles, Fans or Strips

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

```
var vertices = [
    vec2( -0.5, 0.5 ), // v0
    vec2( -0.5, -0.5 ), // v1
    vec2( 0.5, 0.5 ), // v2
    vec2( 0.5, -0.5) // v3
    vec2( 0.5, 0.5) // v4 = v2
    vec2( -0.5, -0.5) // v5 = v1
  ];
```




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

?

Exercise 2

Experimenting with the Sample Program

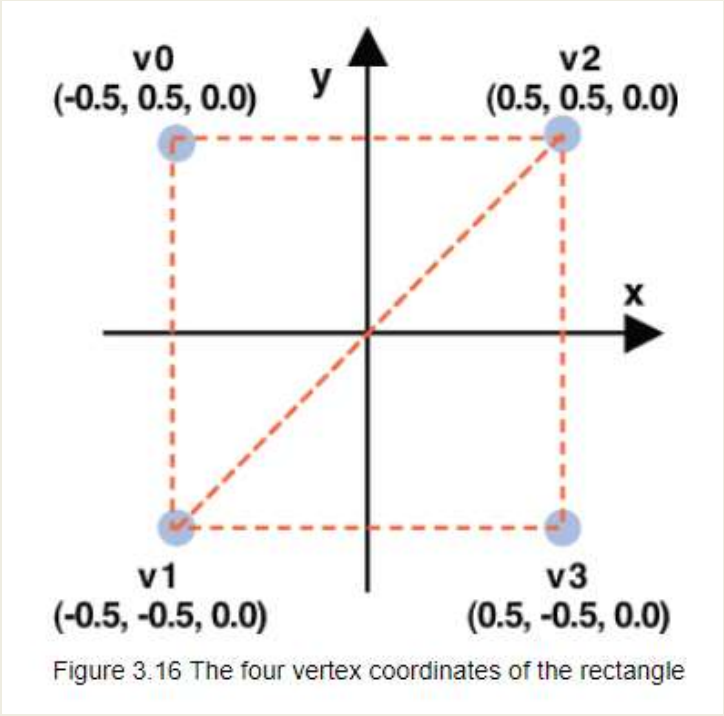
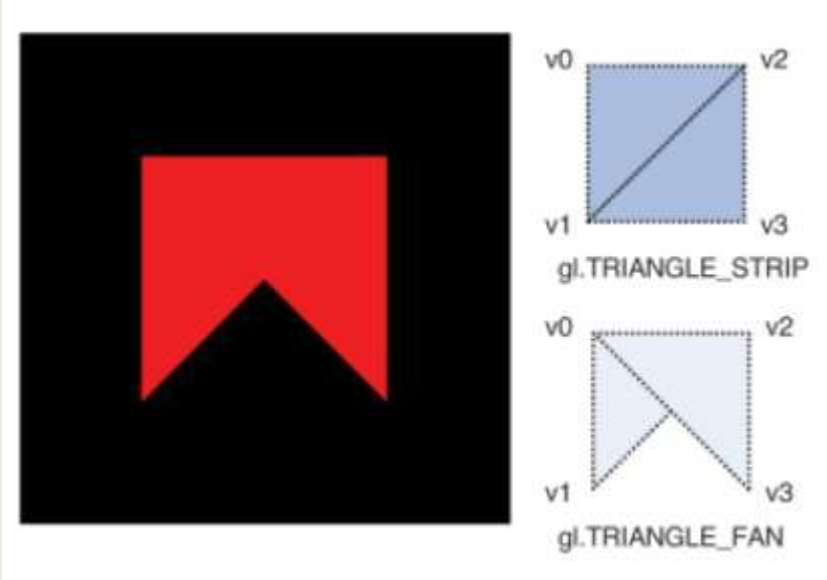


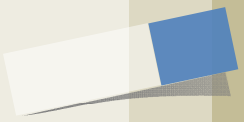
Figure 3.16 The four vertex coordinates of the rectangle



- gl.TRIANGLE_FAN causes WebGL to draw a second triangle that shares the first vertex (vo), and this second triangle overlaps the first, creating the ribbon-like effect.

Vertex & Fragment Shaders

Objectives



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

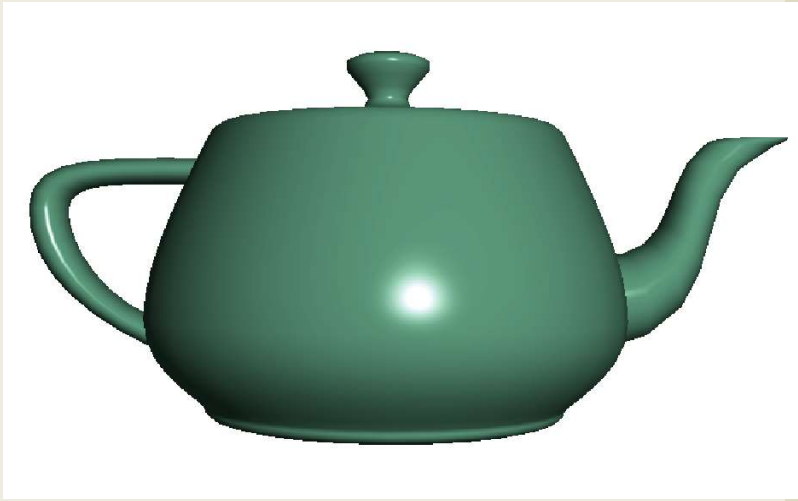
Vertex Shader Applications

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

Fragment Shader Applications

Per fragment lighting calculations

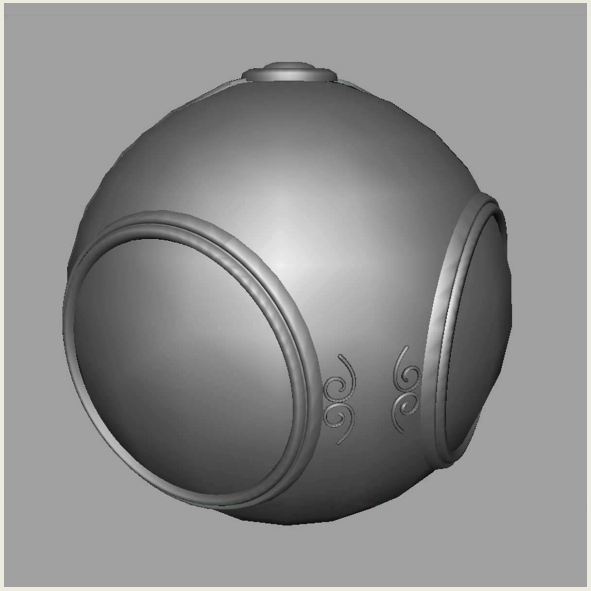
per vertex lighting



per fragment lighting

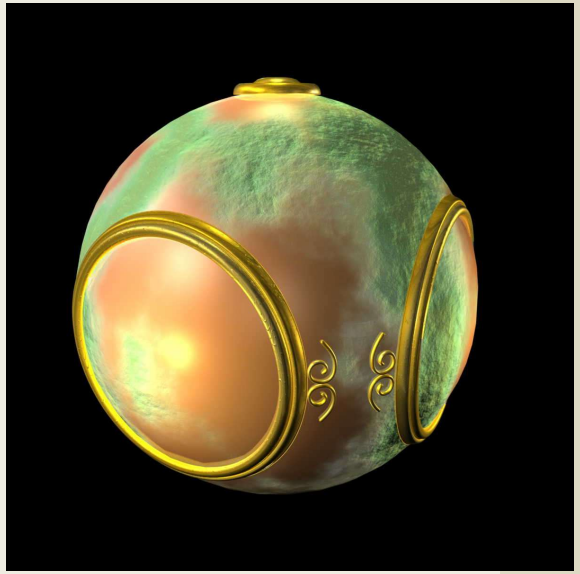
Fragment Shader Applications

Texture mapping



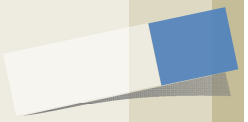
smooth shading

environment mapping



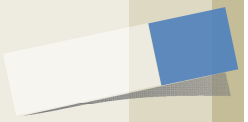
bump mapping

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

A Vertex Shader's job is to generate clipspace coordinates.

- gl.drawArrays(primitiveType, offset, count);
  ➔ Your shader is called once per vertex.
- Each time it's called you are required to set the the special global variable, gl_Position to some clipspace coordinates.

input from application

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

must link to variable in application

built in variable

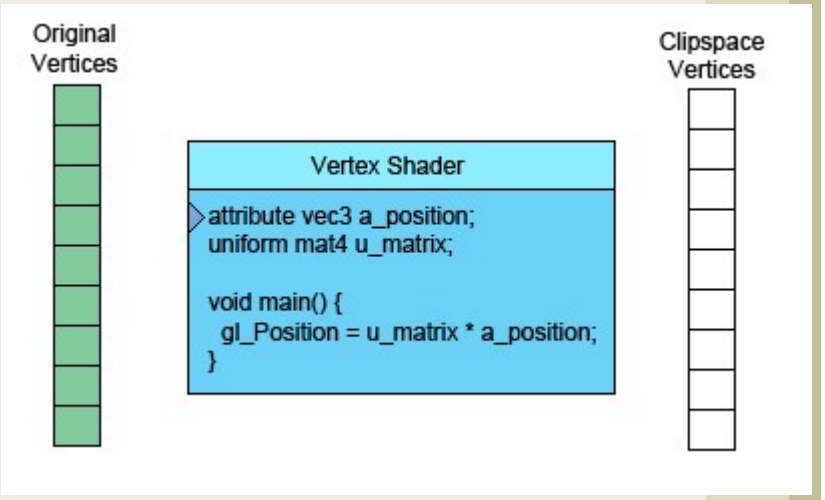
Simple Vertex Shader

input from application

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

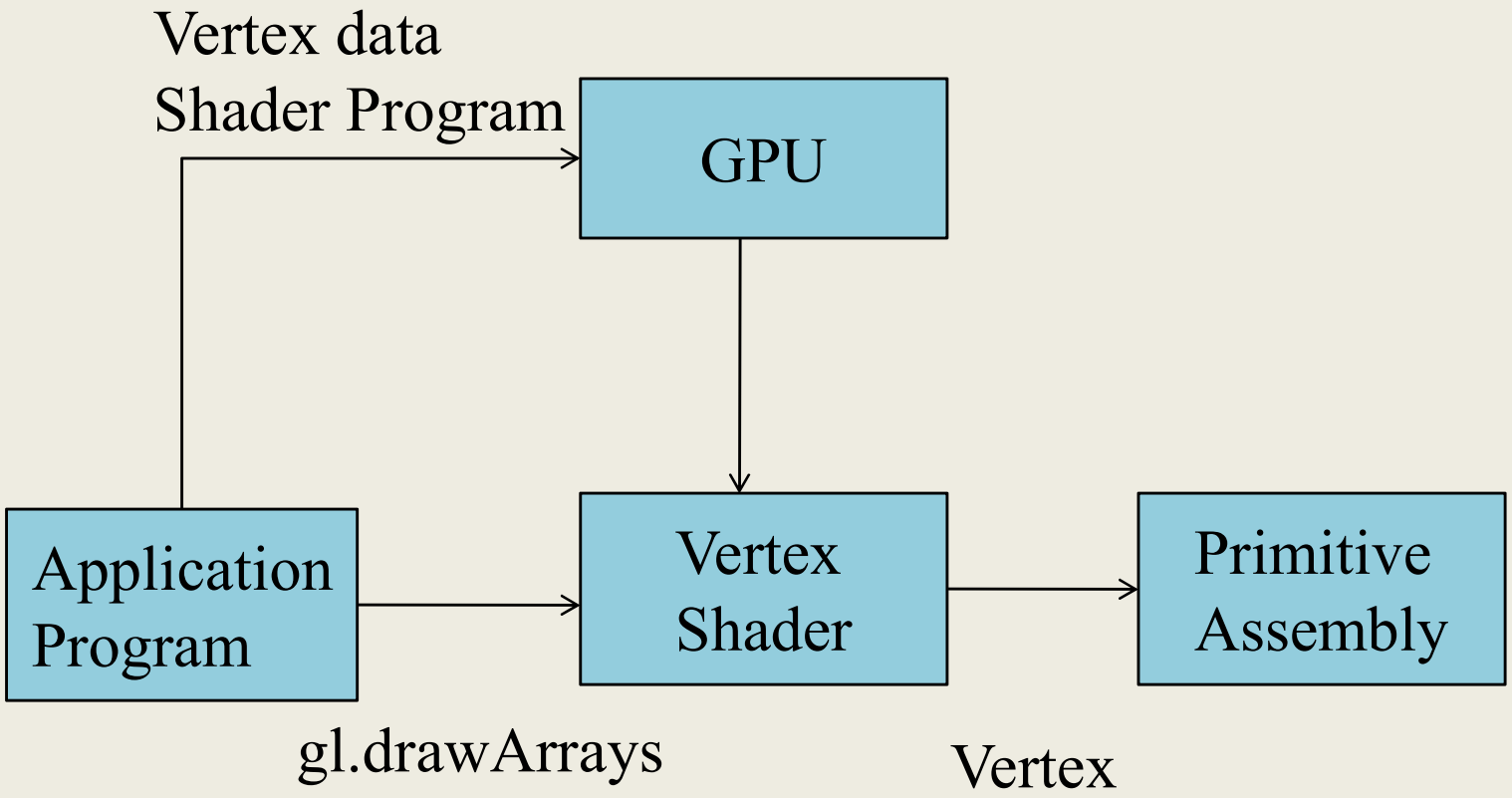
must link to variable in application

built in variable



https://webglfundamentals.org/webgl/lessons/webgl-how-it-works.html

Execution Model



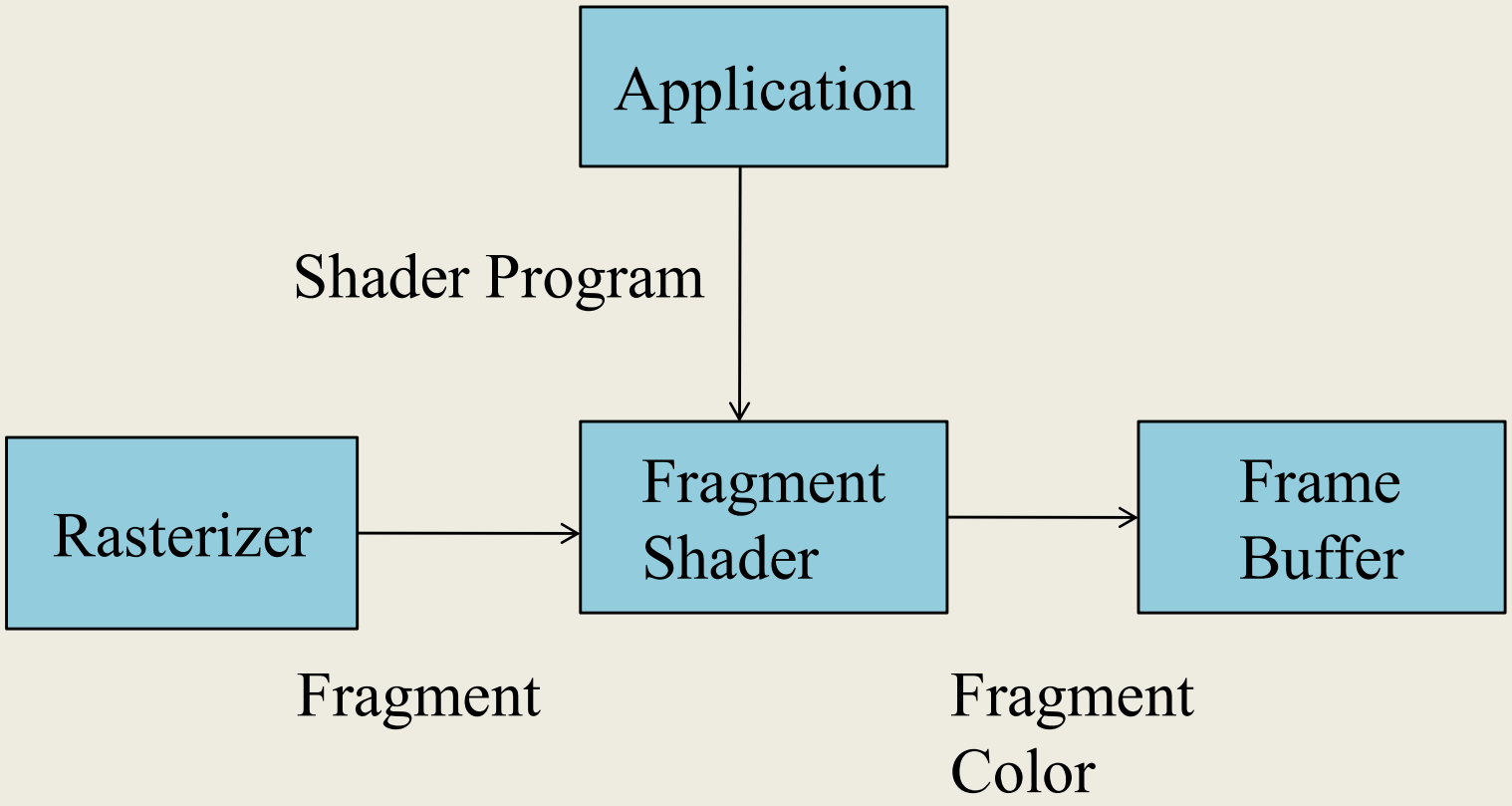
Simple Fragment Program

A Fragment Shader's job is to provide a color for the current pixel being rasterized.

- Your fragment shader is called once per pixel.
- Each time it's called you are required to set the special global variable, gl_FragColor to some color.

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

Execution Model



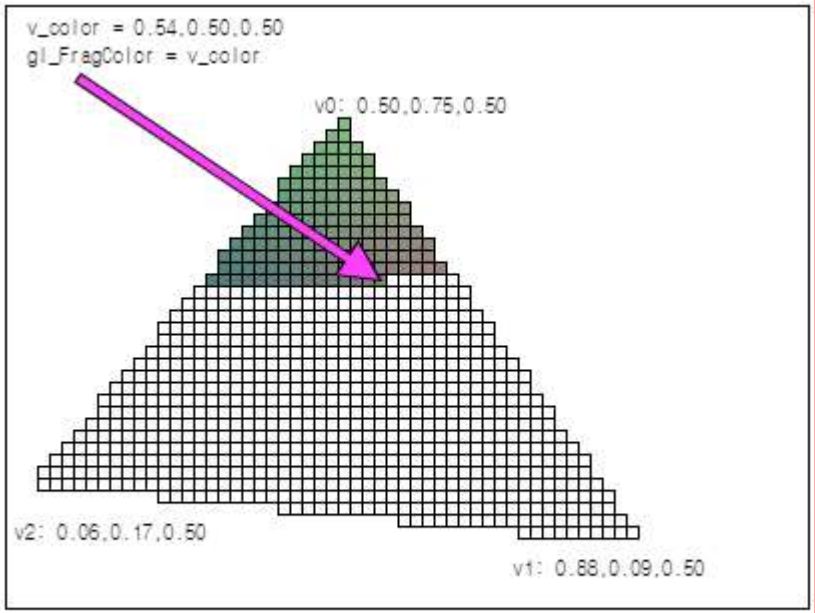
1) Vertex Shader

Vertex	
0	-100
150	125
-175	100

gl_Position에 작성된 값	
0.000	0.660
0.750	-0.830
-0.875	-0.660

v_color에 작성된 값		
0.5000	0.830	0.5
0.8750	0.086	0.5
0.0625	0.170	0.5

2) Fragment Shader



v_color is interpolated between v0, v1 and v2

1) Vertex Shader

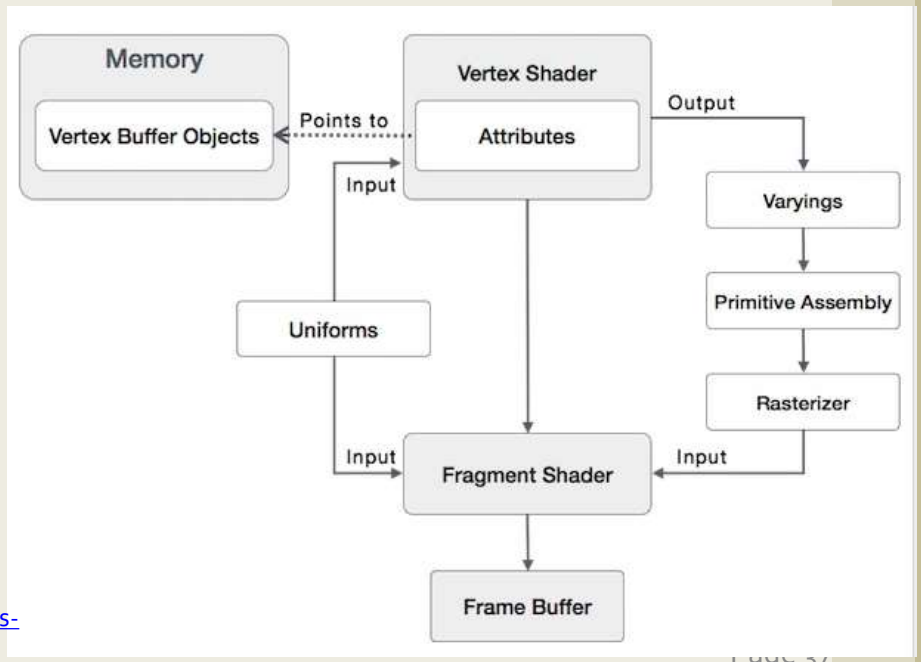
Vertex	
0	-100
150	125
-175	100

gl_Position에 작성된 값	
0.000	0.660
0.750	-0.830
-0.875	-0.660

v_color에 작성된 값		
0.5000	0.830	0.5
0.8750	0.086	0.5
0.0625	0.170	0.5

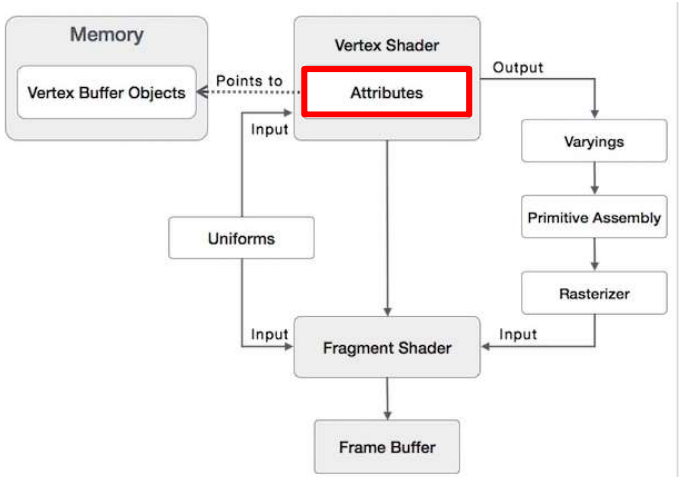
Qualifiers

- All the data that the functions need to access should be provided to the GPU.
- GLSL has many of the same qualifiers such as **`const`** as C/C++
- There are four ways for Shader to receive data:
  - Attribute
  - Uniform
  - Varying
  - Texture



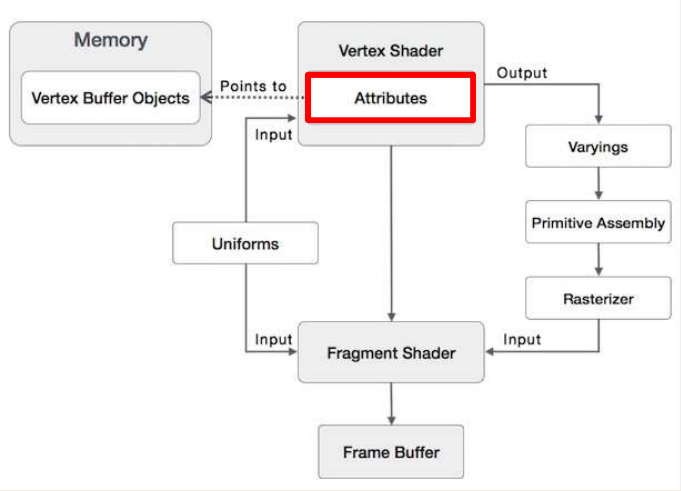
https://www.wisdomjobs.com/e-university/webgl-tutorial-1269/webgl-graphics-pipeline-16662.html

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

- Note that attributes passed to vertex shader are different entities with the same name in the application and the shader




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

Sending Vertices from Application

```
// 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 );
```

Sending Colors from Application

```
// Load the data into the GPU

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

// Associate out shader variables with our data buffer

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

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**

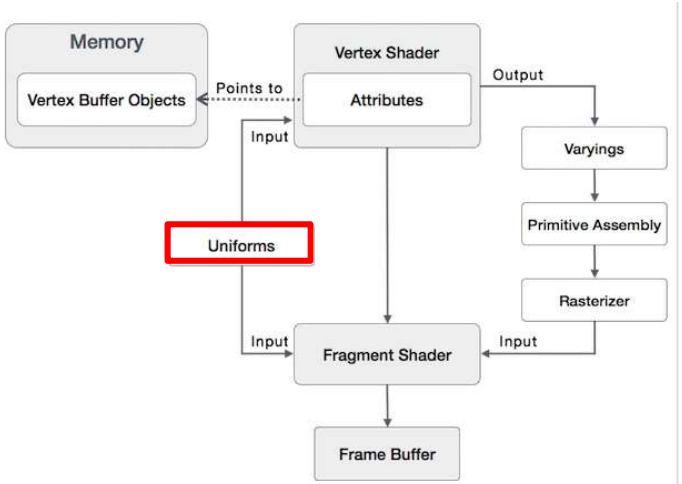
Sending Vertices from Application

```
// 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 );
```

Sending Colors from Application

```
// Load the data into the GPU

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

// Associate out shader variables with our data buffer

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

Uniform Qualified



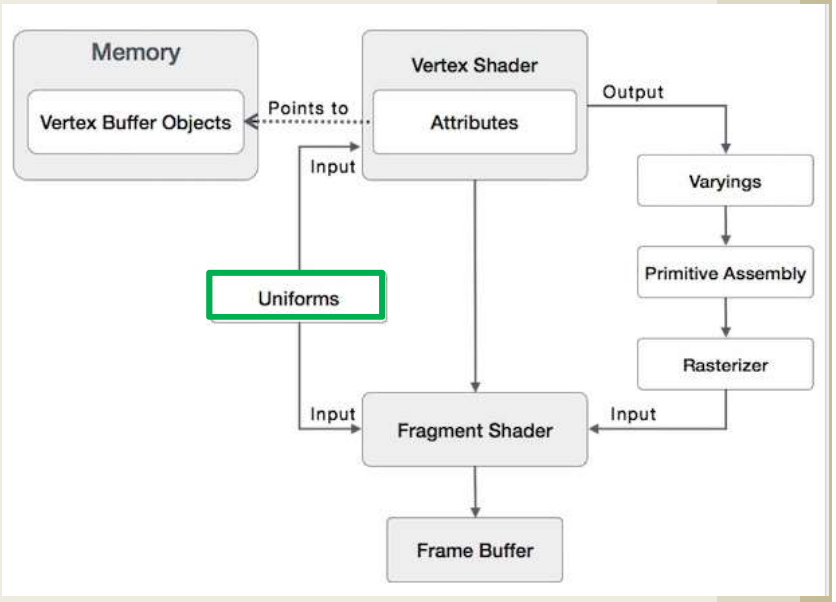
- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the time or a bounding box of a primitive or transformation matrices

Our Naming Convention

- Uniform variables are unadorned and can have the same name in application and shaders

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;
}
```



Example1

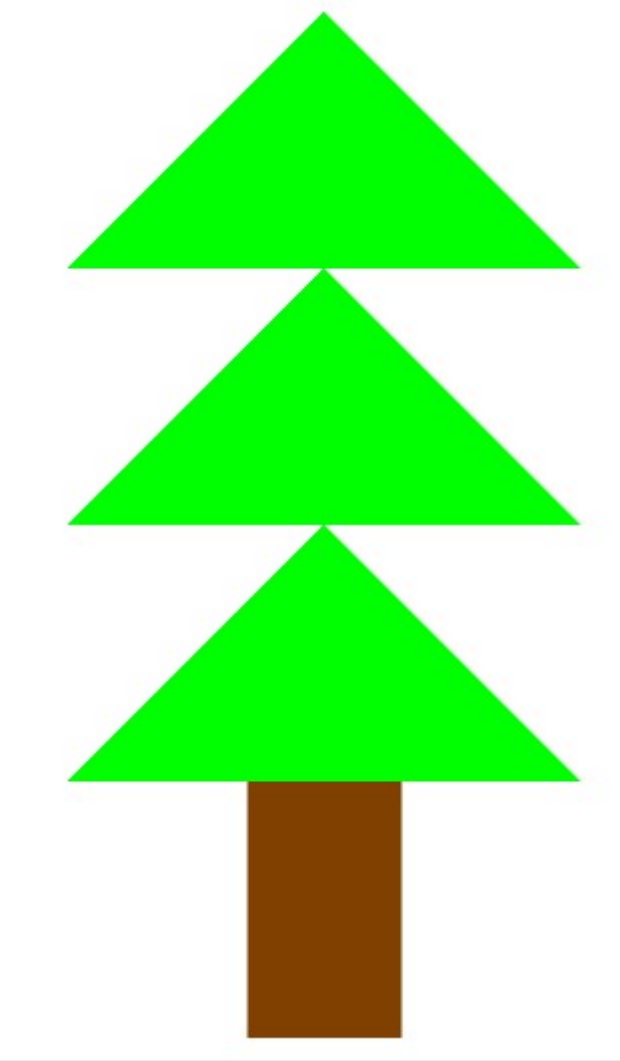
- As a very simple example we could add an offset.

```
attribute vec4 vPosition;
uniform vec4 vOffset;

void main() {
   gl_Position = vPosition + vOffset;
}
```

```
var vOffset = gl.getUniformLocation(someProgram, " vOffset ");
```

```
 // offset it to the right half the screen
gl.uniform4fv(vOffset, [1, 0, 0, 0]);
```

Exercise 3



- Draw a left tree with modifying triangle.html/js
- For using several colors, see 'sending a uniform variable' in page #42
- What should we modify a previous codes?
  - Repeat parts, which should be changed
  - Bufferdata, color, position, draw