CIS367 - Computer Graphics

A full program (and some shaders)

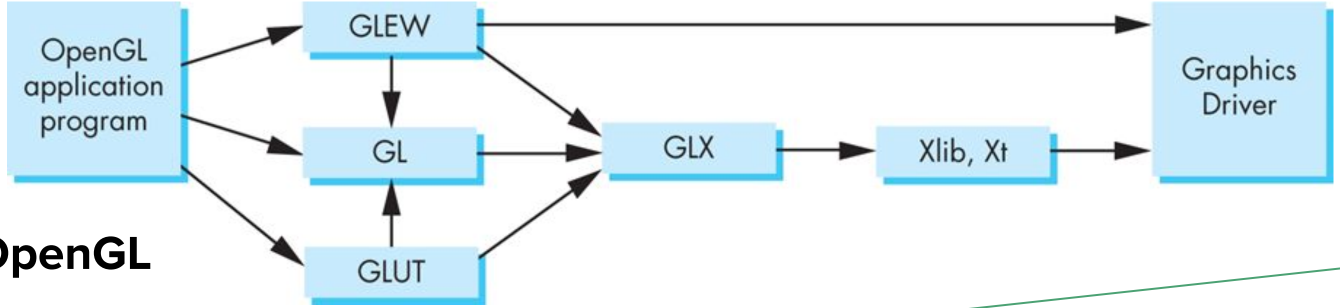
Erik Fredericks - frederer@gvsu.edu

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

Overview

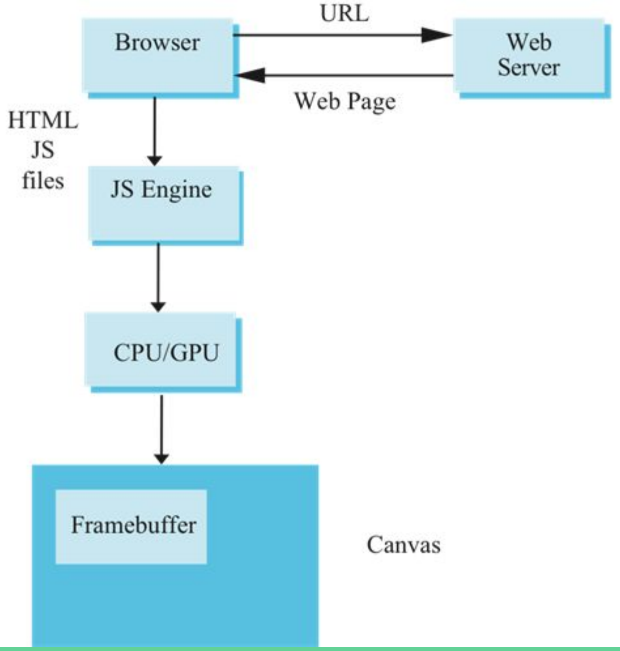
The full thing, from start to finish, including:

- GLSL introduction
- Using triangles to draw a square



OpenGL

WebGL



Angel and Shreiner: Interactive Computer Graphics 7E © A

Event Loop

Program specifies a **`render` function**

- Actually an event listener / callback

Static programs (meaning, no change)

- Call `render` once

Dynamic programs (animations, user interaction, etc.)

- `render` called recursively
    - Redrawing triggered by some event

WebGL function format

function name

dimension

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

belongs to
WebGL canvas

`x`, **`y`**, **`z`** are float variables

`gl.uniform3fv(p)`

`p` is a vector (array)

Constants

Constants generally defined in canvas object (`gl`)

- OpenGL ➜ came from #includes

Examples

- OpenGL (desktop)
    - `glEnable(GL_DEPTH_TEST);`

- WebGL (what we're doing)
    - `gl.enable(gl.DEPTH_TEST);`
    - `gl.clear(gl.COLOR_BUFFER_BIT);`

WebGL and GLSL

Shader-driven programming a requirement

- Focus on data flow instead of state flow model (i.e., not vanilla OpenGL)
- All action happens in shaders

Purpose of application?

- Package up data and send to GPU

GLSL

GLSL ➔ OpenGL Shading Language

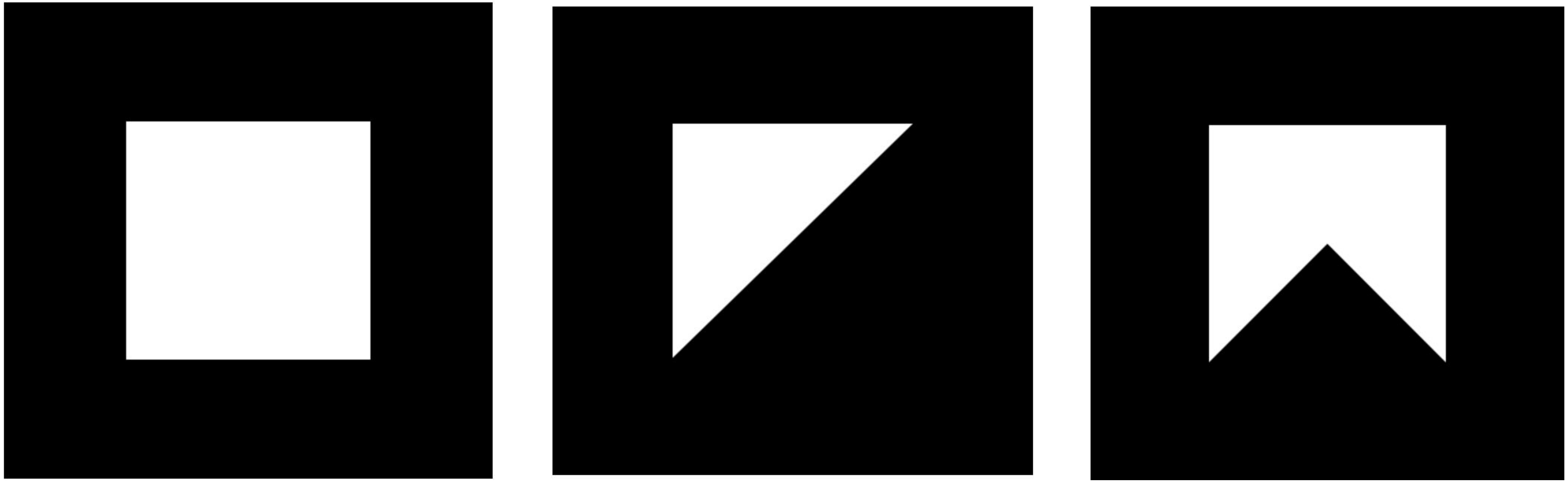
C-"like"

- Matrices and vectors (2 -- 4 dimensions)
- Operator overloads
- C++-"like" constructors

Code sent to shader as source code

- WebGL functions compile, link, get information to shaders

Square drawing



https://efredericks.github.io/gvsu-cis367/demos/square.html

5 steps for program

Setup webpage (HTML)

- Request WebGL canvas
- Read files

Define shaders (HTML)

- Can be separate as well

Compute/specify data (JavaScript)
Send data to GPU (JavaScript)
Render data (JavaScript)

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

Shaders are assigned names to be used in our JavaScript files

Considered "trivial pass-through" (do nothing) with two **required** built-in variables

- `gl_Position`
- `gl_FragColor`

Both shaders are **full programs**

Vector type: `vec2`

Precision set in **fragment shader**

square.html continued

```
<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>
```

Included files

../Common/webgl-utils.js

- Standard utilities for setting up WebGL (% Angel/Shreiner)

../Common/initShaders.js

- Matrix-vector package (% Angel/Shreiner)

../Common/MV.js

- Matrix-vector package (% Angel/Shreiner)

square.js

- Our application

Always include these!

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 on JS/WebGL

`onload`:

- Where execution starts **after** all code is loaded into memory

`canvas` element receives WebGL context from HTML file

Vertices are using vec2 type from the MV.js library

JS array is different from C/Java arrays

- object with methods and properties
  - vertices.length ➔ 4

Values in clip coordinates

square.js continued

```
//  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 our shader variables with our data buffer
var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
```

More notes

`initShaders`

- Load/compile/link shaders to form a **program object**

Data loaded into GPU by creating **vertex buffer object**

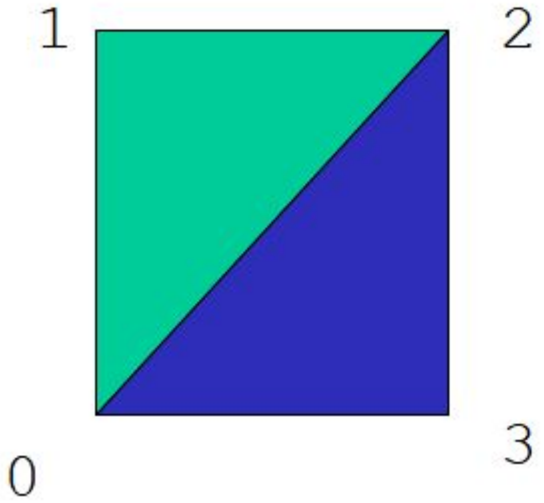
- `flatten()` converts a JS array to array of `float32`

Connect program variable to shader variable

- Requires name/type/location in buffer

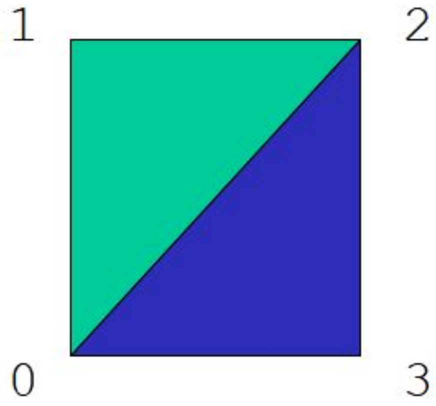
square.js (finalized)

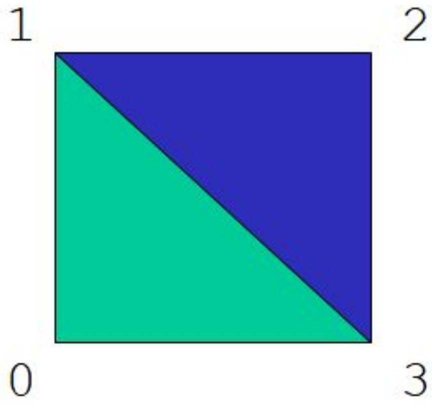
```
     render();
};
function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 );
}
```



Triangles, Fans, Strips (pick one in lecture-square.js)

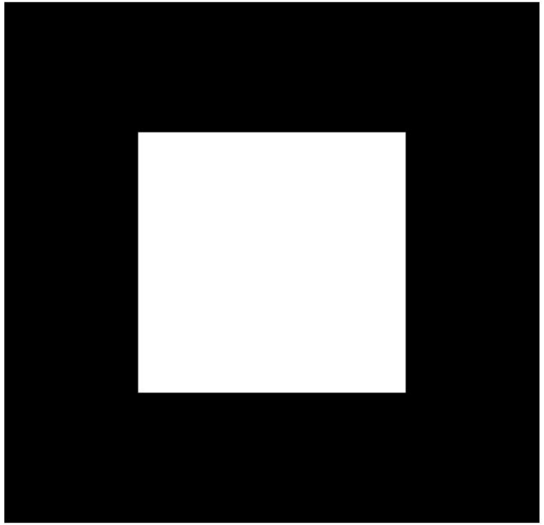
```
gl.drawArrays( gl.TRIANGLES, 0, 6 );      // 0, 1, 2, ...
gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 );   // 0, 1, 2, 3
gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 ); // 0, 1, 3, 2
```



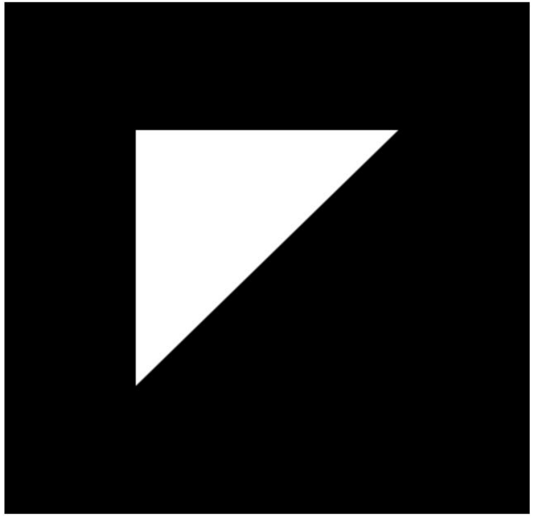


Square drawing

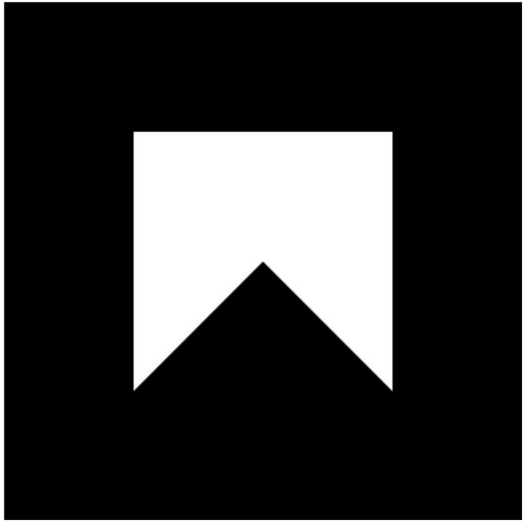
`gl.TRIANGLE_FAN`



`gl.TRIANGLES`



`gl.TRIANGLE_STRIP`



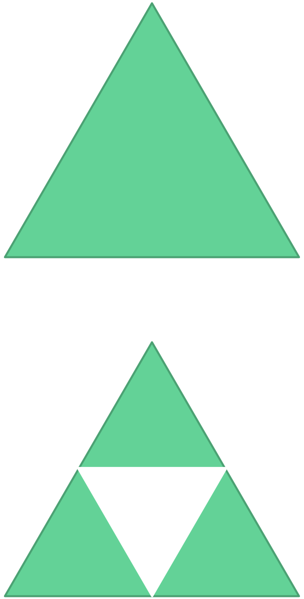
Sierpinski Gasket

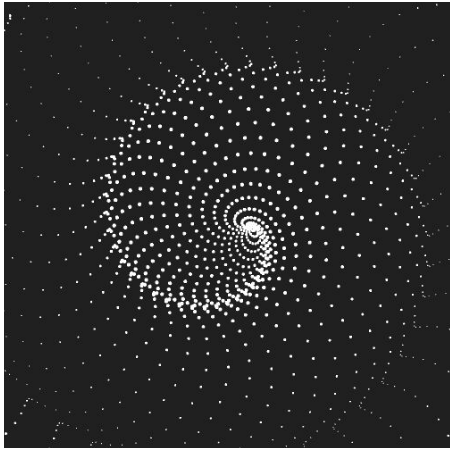
Recursively drawn fractal

Connect bisectors of sides and remove central triangle

Repeat until done

- We'll come back to this!

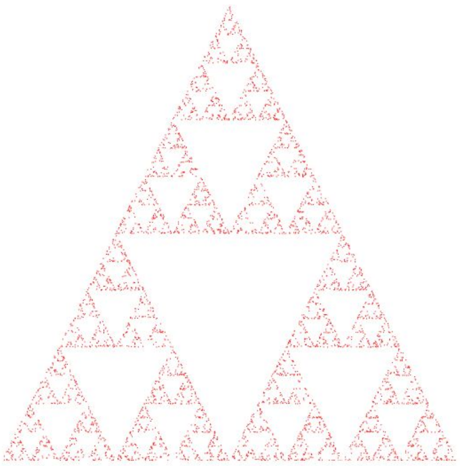




a fractal

https://efredericks.github.io/gvsu-cis367/demos/gasket1.html

The HTML



```
<!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>
<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>
```

More HTML!

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

gasket.js

```
"use strict";

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" ); }

  // First, initialize the corners of our gasket with three points.
  var vertices = [
      vec2( -1, -1 ),
      vec2(  0,  1 ),
      vec2(  1, -1 )];

  // 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 );
}

//  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 );
```

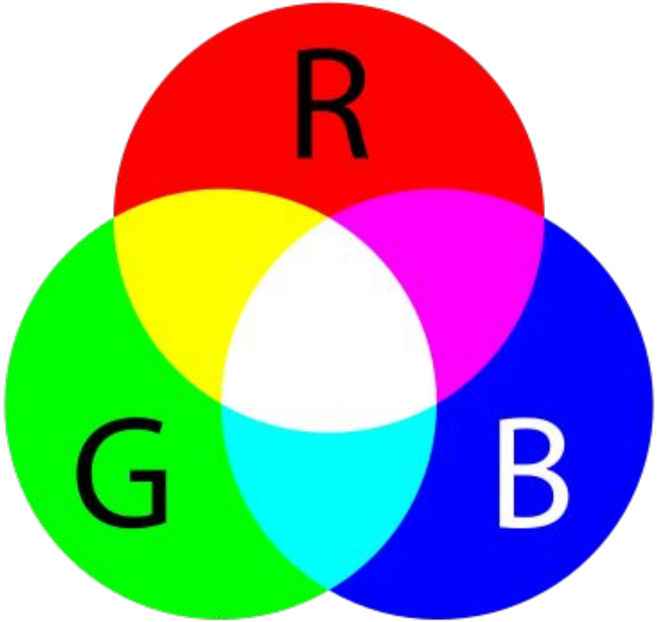
```
  // Load the data into the GPU
  var bufferId = gl.createBuffer();
  gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
  gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

  // 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();
};


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

http://glslsandbox.com/

https://www.shadertoy.com/

More on Colors

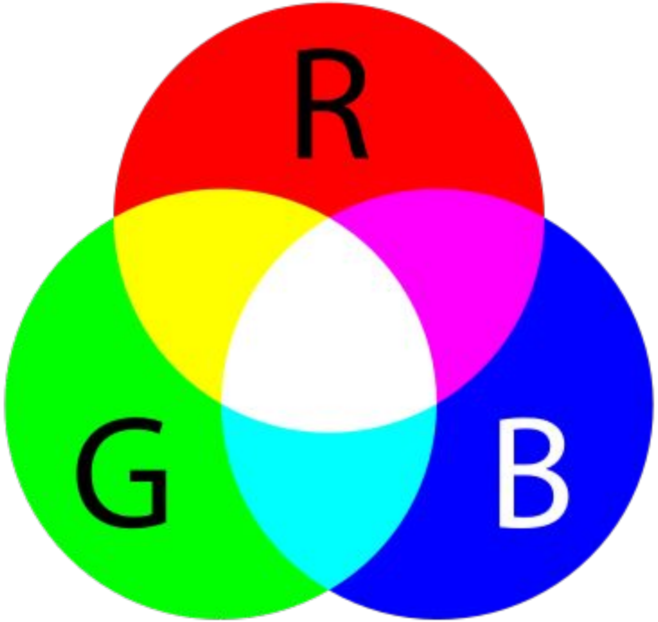
Colors can be represented in multiple ways




```
R: [0, 255]
G: [0, 255]
B: [0, 255]
```
A: [0, 255]?

https://www.rapidtables.com/web/color/RGB_Color.html

More on Colors



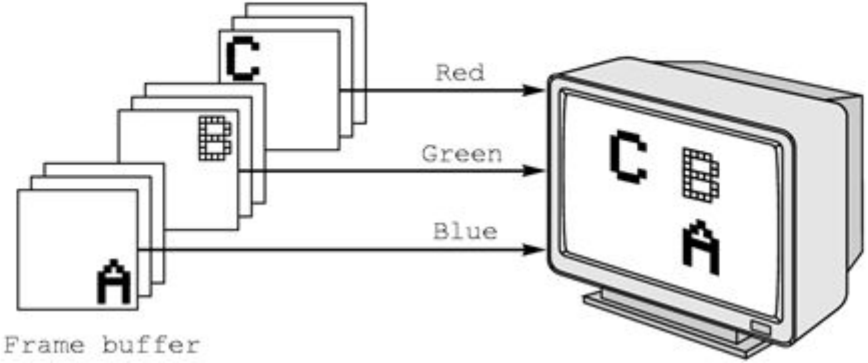

```
RGB red:    (255, 0, 0)

WebGL red: vec3(1.0, 0.0, 0.0)
```

What if we want to add transparency?

RGB Color

Each component stored separately in frame buffer

- Generally, 8 bits per component
  - 2^8 → 256 possible values

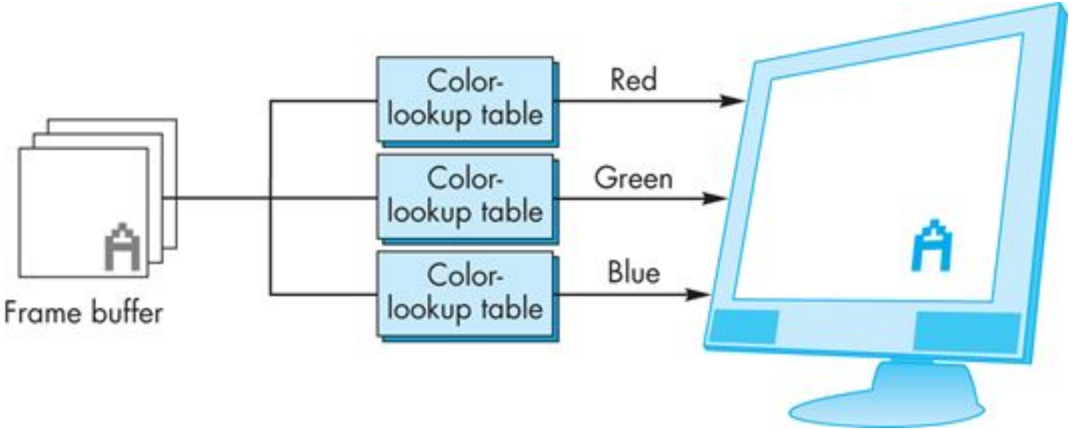


- Values range from
  - 0.0 → 1.0 using floats
  - 0 → 255 using unsigned bytes

Indexed Color

Colors are actually indices into RGB table values

- Requires less memory
  - Index ➔ 8 bits



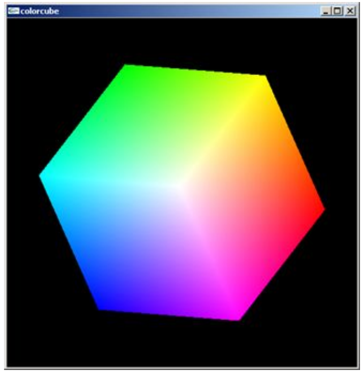
Smooth Color

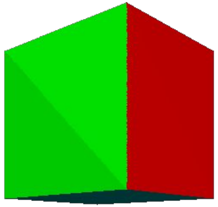
Default shading in WebGL is *smooth*

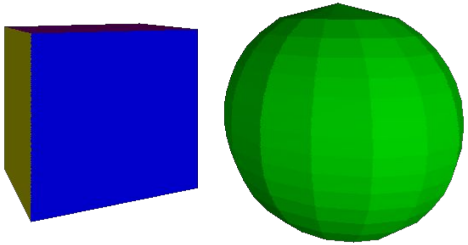
- Rasterizer interpolates vertex colors across the *visible* polygons

Alternative is *flat shading*

- Color of first vertex determines face color
- Handled by shader







Setting Colors

Set in fragment shader

- Can be determined by **either** shader or in application

Application color

- Pass to *vertex shader* as **uniform variable** or as **vertex attribute**

Vertex shader color

- Pass to *fragment shader* as **varying variable**

Fragment color

- Can alter in shader code

Setting the background color

Let's say we want our canvas background (drawing window) to white.

- ➜ na, that's boring, how about hot pink?
- ➜ that will be 0xFF00FF
  - ➜ or (255, 0, 255)
    - ➜ or (1.0, 0.0, 1.0)

Anybody know why this is special?

```
gl.clearColor(1.0, 0.0, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

Generally...

We'll be storing them as either a `vec3` or a `vec4`

`vec3` ➜ don't care about alpha

`vec4` ➜ care about alpha

Sample colors definition ➜ index into array for later lookup!

```
var colors = [
    vec3(1.0, 0.0, 0.0), // red
    vec3(0.0, 1.0, 0.0), // green
    vec3(0.0, 0.0, 1.0), // blue
    vec3(1.0, 1.0, 1.0), // white
    vec3(0.0, 0.0, 0.0)  // black
];
```

More on the WebGL API

IF YOU ARE AT ALL CONFUSED ABOUT THE CONNECTION BETWEEN WEBGL AND "THE PIPELINE," THEN READ AND READ AND READ CHAPTER 2!!!

(mainly 2.8)

WebGL API (for our implementation at least)

HTML file

- **Canvas**
  Viewport -- drawing area
  <canvas>

- **Vertex shader**
  Initially, location information

- **Fragment shader**
  Initially, color information

JavaScript file
"The **application**"

- Connectors from shaders to pull in data
- E.g., define a rotation angle theta in shaders
  - Theta updated by application
  - We'll see this when we do animation

Things to keep in mind

Our application's **main** function (separate from vertex/fragment shader main) is the **`window.onload`** command.

- This is the JavaScript function triggered after everything is pulled into memory (code-wise)