Visual Computing Exercise 6: Introduction to WebGL

Nikola Kovacevic

nikolak@inf.ethz.ch

Schedule

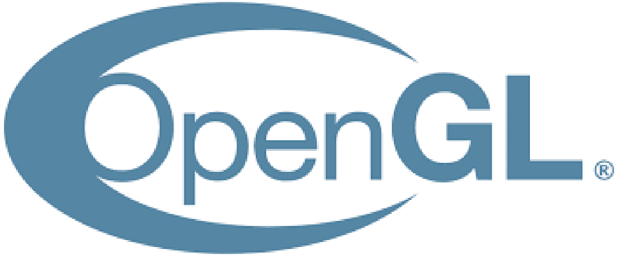
Lecture	Exercise	TA
Nov. 7/9	Ex. 6: WebGL Rendering	Nikola Kovacevic
Nov. 14/16	Ex. 7: Light and Colors	Yingyan Xu
Nov. 21/23	Ex. 8: Transformations	Nikola Kovacevic
Nov. 28 / 30	Ex. 9: Shaders in WebGL	Yingyan Xu
Dec. 5/7	Ex. 10: Lighting and Shading	Philine Witzig
Dec. 12/14	Ex. 11: Curves and Surfaces	Lingchen Yang
Dec. 19/21	Q&A Session	All TA's

Overview

- Introduction to WebGL
- Graphics Pipeline
- Code Template
  - Initialization
  - Shaders
  - Drawing a Triangle
- Exercise 6

What is WebGL?

- JavaScript API for rendering graphics in the web
- Used for 2D and 3D computer graphics applications
- Hardware independent – no plug-ins



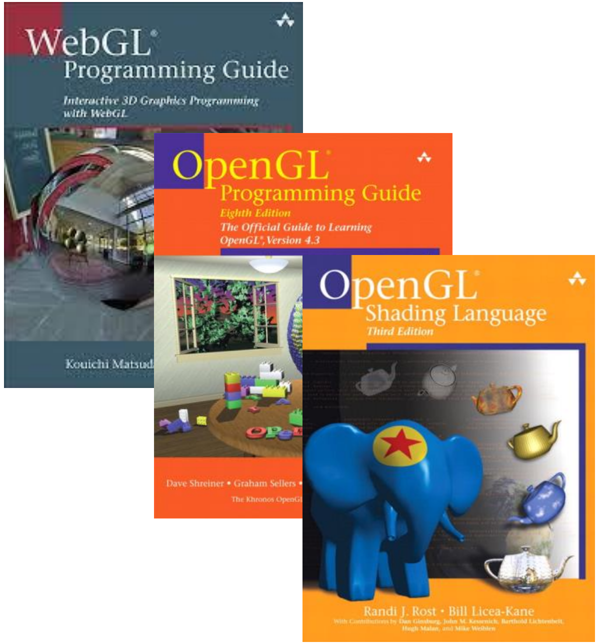
Documentation

- WebGL:

  The gray book
- OpenGL:

  The red book
- OpenGL shading language:

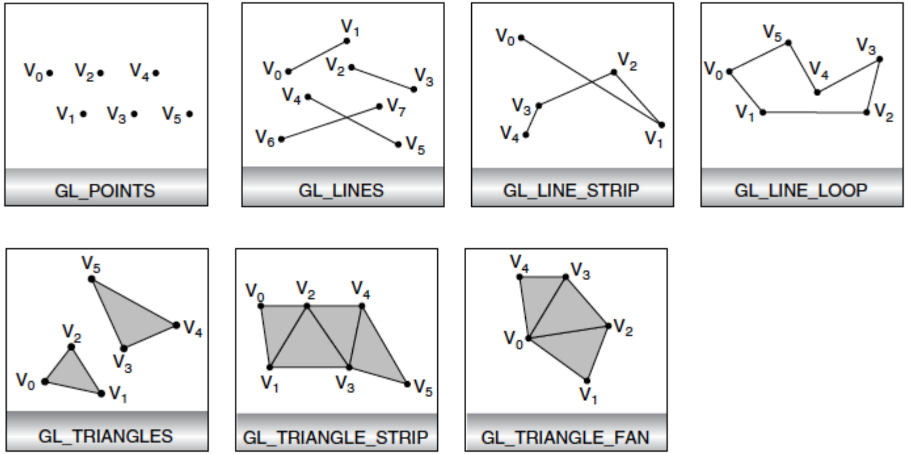
  The orange book

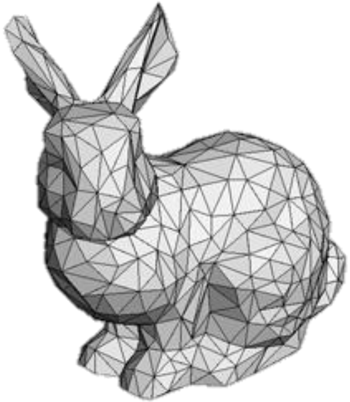

WebGL
Programming Guide
Interactive 3D Graphics Programming with WebGL
Kouichi Matsud
OpenGL
Programming Guide
Eighth Edition
The Official Guide to Learning OpenGL, Version 4.3
Dave Shreiner • Graham Sellers •
The Khronos OpenGl
OpenGL
Shading Language
Third Edition
Randi J. Rost • Bill Licea-Kane

- Useful links: WebGL API and glmatrix API

Rendering with WebGL

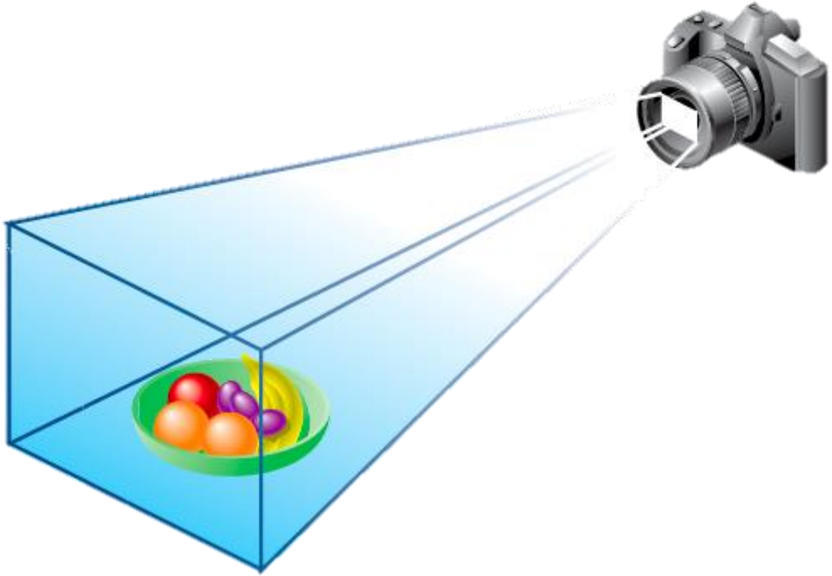
- Construct shapes from geometric primitives





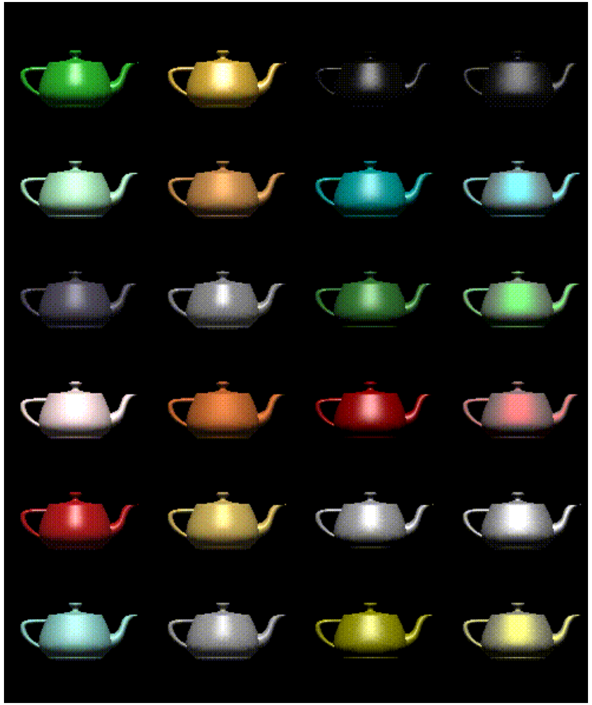
Rendering with WebGL

- Arrange objects in 3D space
- Specify viewpoint



Rendering with WebGL

- Calculate colors of objects
  - textures, materials, lighting
- Colors explicitly controlled with shaders



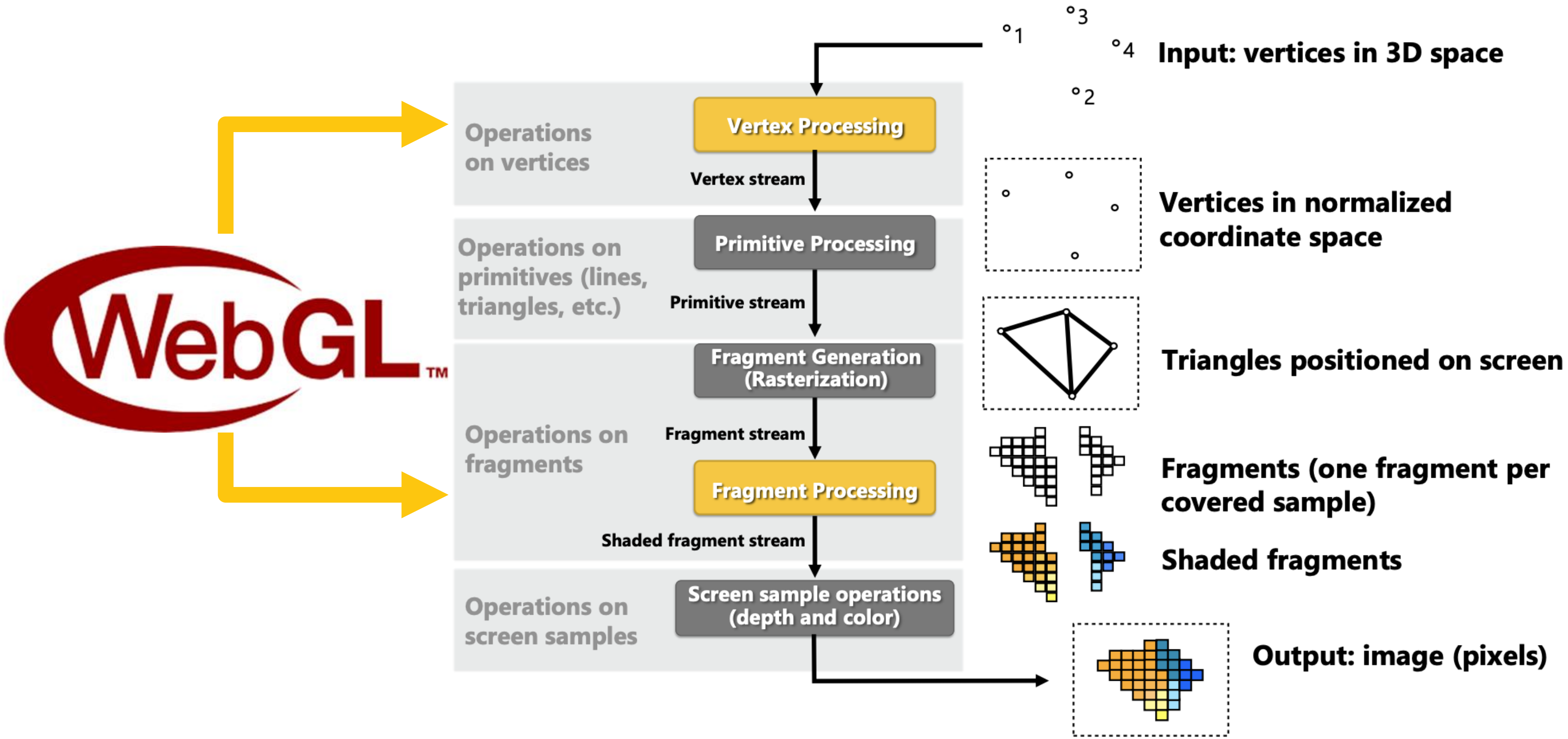
WebGL is a state machine

- WebGL can be put into various states or modes
- Settings remain in effect until changed again
- Examples: drawing color, characteristics of lights, viewing parameters

Overview

- Introduction to WebGL
- Graphics Pipeline
- Code Template
  - Initialization
  - Shaders
  - Drawing a Triangle
- Exercise 6

Graphics Pipeline



WebGL™

Operations on vertices

Vertex Processing

Vertex stream

Operations on primitives (lines, triangles, etc.)

Primitive Processing

Primitive stream

Fragment Generation (Rasterization)

Operations on fragments

Fragment stream

Fragment Processing

Shaded fragment stream

Operations on screen samples

Screen sample operations (depth and color)

Input: vertices in 3D space

Vertices in normalized coordinate space

Triangles positioned on screen

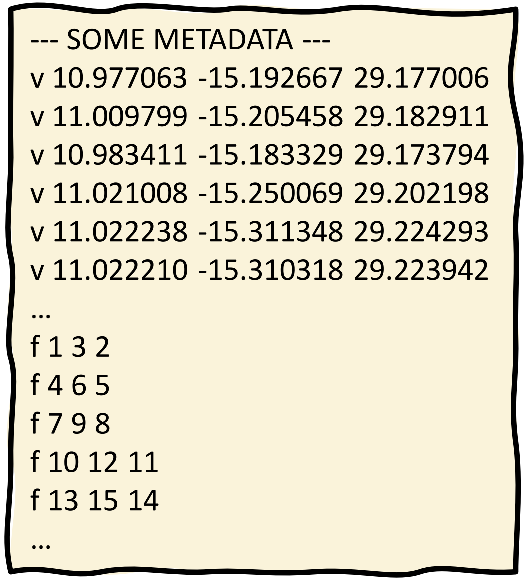
Fragments (one fragment per covered sample)

Shaded fragments

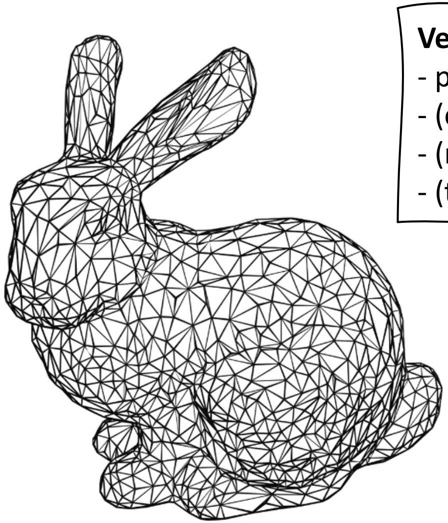
Output: image (pixels)

Input Mesh

- How is a mesh represented?
  - Take an .OBJ file and open it in a text editor

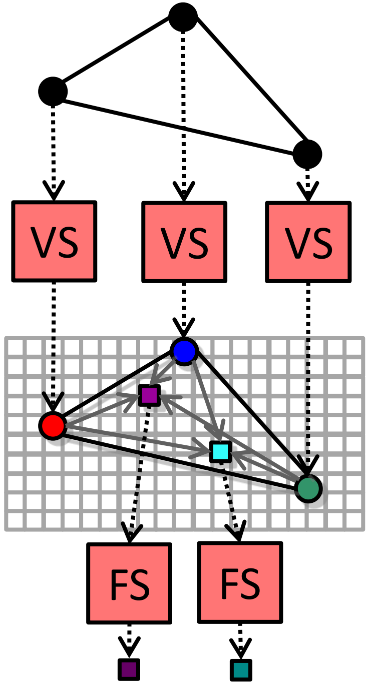


```
--- SOME METADATA ---
v 10.977063 -15.192667 29.177006
v 11.009799 -15.205458 29.182911
v 10.983411 -15.183329 29.173794
v 11.021008 -15.250069 29.202198
v 11.022238 -15.311348 29.224293
v 11.022210 -15.310318 29.223942
…
f 1 3 2
f 4 6 5
f 7 9 8
f 10 12 11
f 13 15 14
…
```



Vertex
- position
- (color)
- (normal)
- (texture)

Vertex and fragment shaders



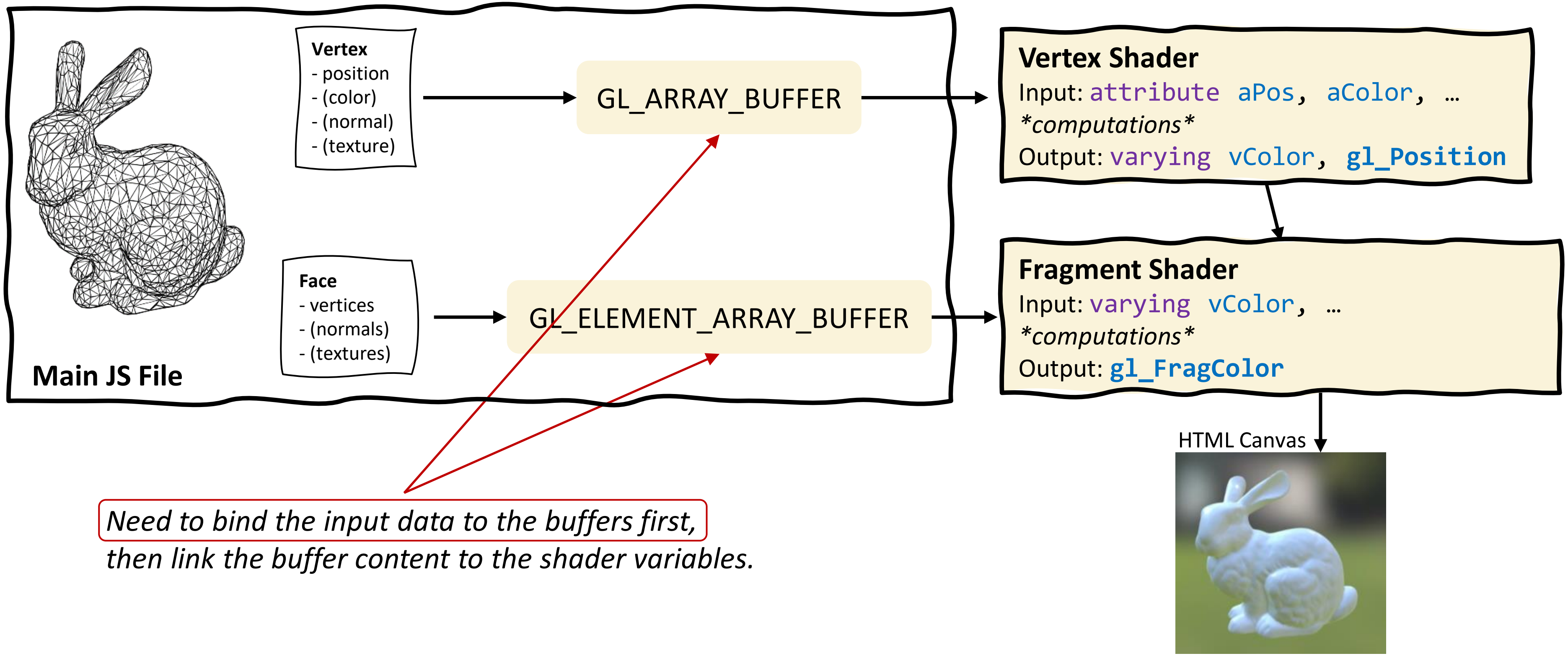
`attribute`(s) given per vertex

Vertex shader computes `varying`

Interpolation of varying values

Fragment shader computes pixel color

GL Pipeline



Vertex
- position
- (color)
- (normal)
- (texture)

Face
- vertices
- (normals)
- (textures)

Main JS File

GL_ARRAY_BUFFER

GL_ELEMENT_ARRAY_BUFFER

Vertex Shader
Input: `attribute aPos, aColor, …`
computations
Output: `varying vColor, gl_Position`

Fragment Shader
Input: `varying vColor, …`
computations
Output: `gl_FragColor`

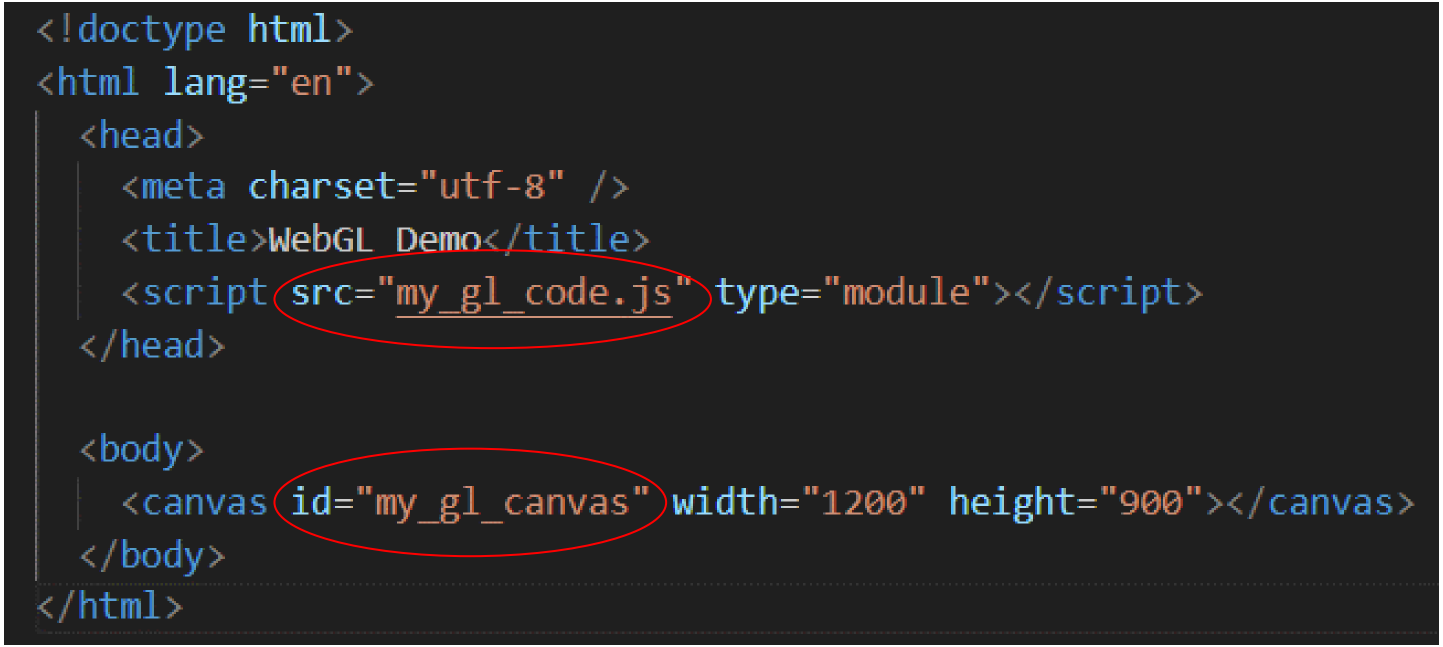
HTML Canvas

Need to bind the input data to the buffers first,
then link the buffer content to the shader variables.

Overview

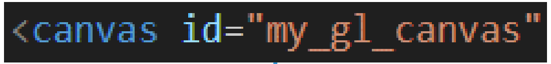
- Introduction to WebGL
- Graphics Pipeline
- Code Template
  - Initialization
  - Shaders
  - Drawing a Triangle
- Exercise 6

Initialization (HTML)



```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>WebGL Demo</title>
    <script src="my_gl_code.js" type="module"></script>
  </head>

  <body>
    <canvas id="my_gl_canvas" width="1200" height="900"></canvas>
  </body>
</html>
```

Initialization (JS)



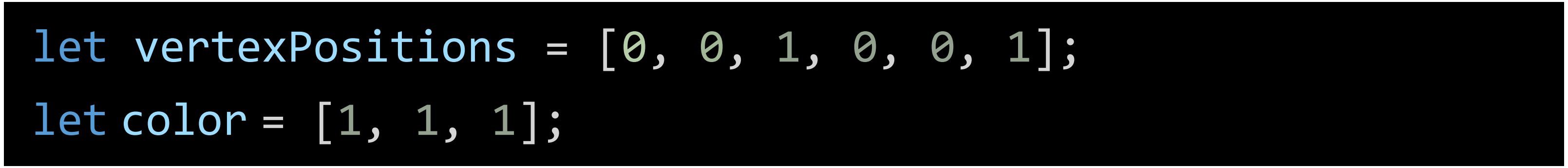
```
<canvas id="my_gl_canvas"
```

```
var canvas = document.getElementById('my_gl_canvas');
var gl = canvas.getContext('webgl');
```

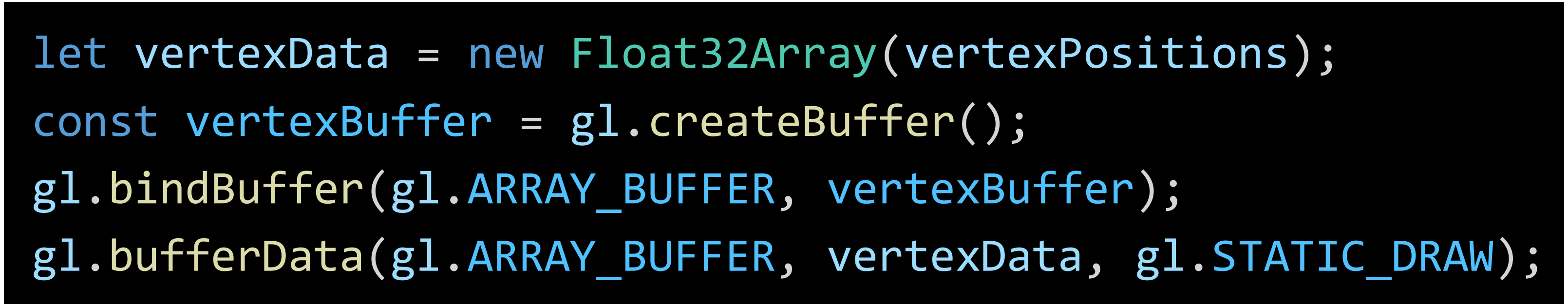
```
gl.viewport(0, 0, canvas.width, canvas.height);
gl.clearColor(0., 0., 0., 1.); // black (rgba)
```

```
// depth test is by default disabled.
gl.enable(gl.DEPTH_TEST);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Bind Input Data



```
let vertexPositions = [0, 0, 1, 0, 0, 1];
let color = [1, 1, 1];
```



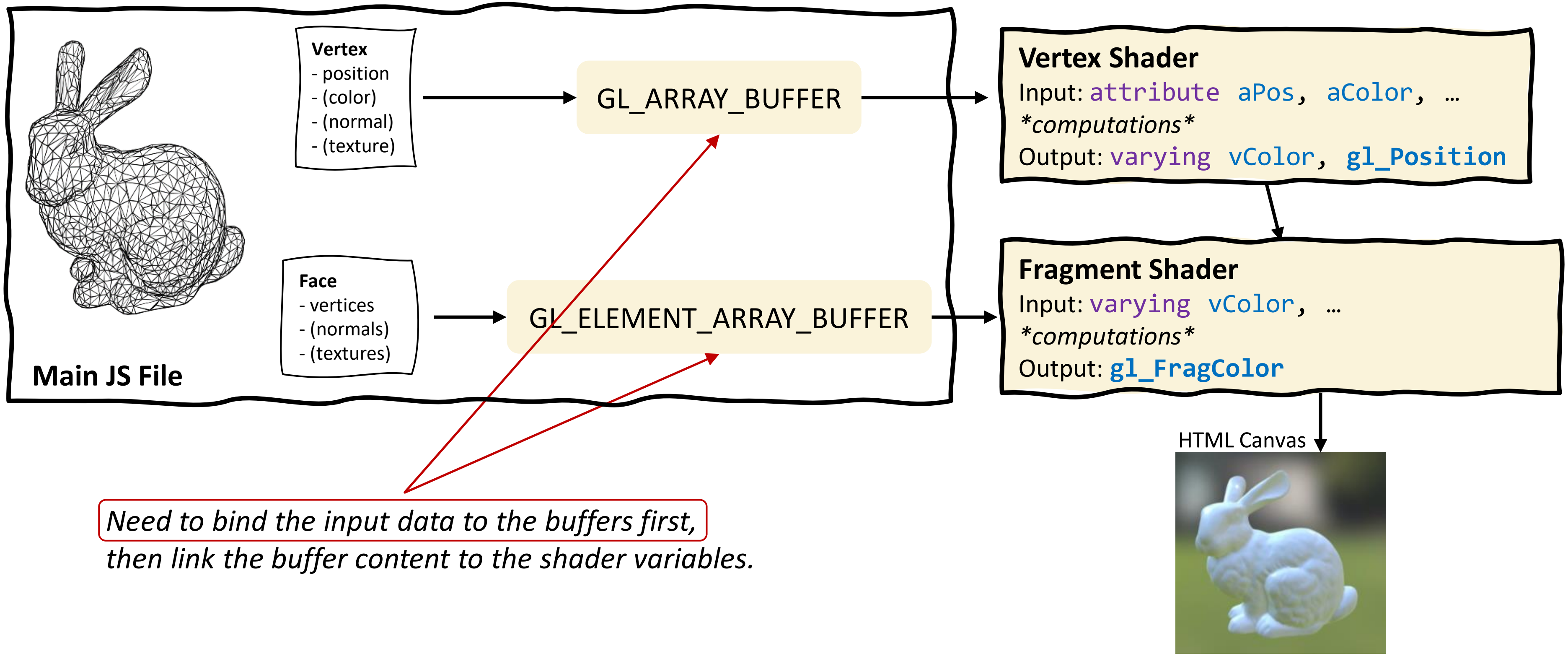
```
let vertexData = new Float32Array(vertexPositions);
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);
```

Repeat for other input data (e.g., color, normal, texture coordinates)

Create separate buffers and bind them to `gl.ARRAY_BUFFER`

For the face data use `gl.ELEMENT_ARRAY_BUFFER`

GL Pipeline



Vertex
- position
- (color)
- (normal)
- (texture)

Face
- vertices
- (normals)
- (textures)

Main JS File

GL_ARRAY_BUFFER

GL_ELEMENT_ARRAY_BUFFER

Vertex Shader
Input: attribute aPos, aColor, …
computations
Output: varying vColor, **gl_Position**

Fragment Shader
Input: varying vColor, …
computations
Output: **gl_FragColor**

HTML Canvas

Need to bind the input data to the buffers first,
then link the buffer content to the shader variables.

Vertex shader example

```
#version 130

uniform vec4 uLightPos;
uniform mat4 uProjectViewMatrix;

attribute vec4 aPosition;
attribute vec4 aColor;
attribute vec3 aNormal;

varying vec4 vColor;

void main(void) {

   // Lighting
   vec3 vecToLight = normalize(uLightPos.xyz - aPosition.xyz);
   float diffuseIntensity = dot(aNormal, vecToLight);
   diffuseIntensity = clamp(diffuseIntensity, 0.0, 1.0);
   vColor = aColor * diffuseIntensity;

   // Project vertex coordinates to screen
   gl_Position = uProjectViewMatrix * aPosition;
}
```

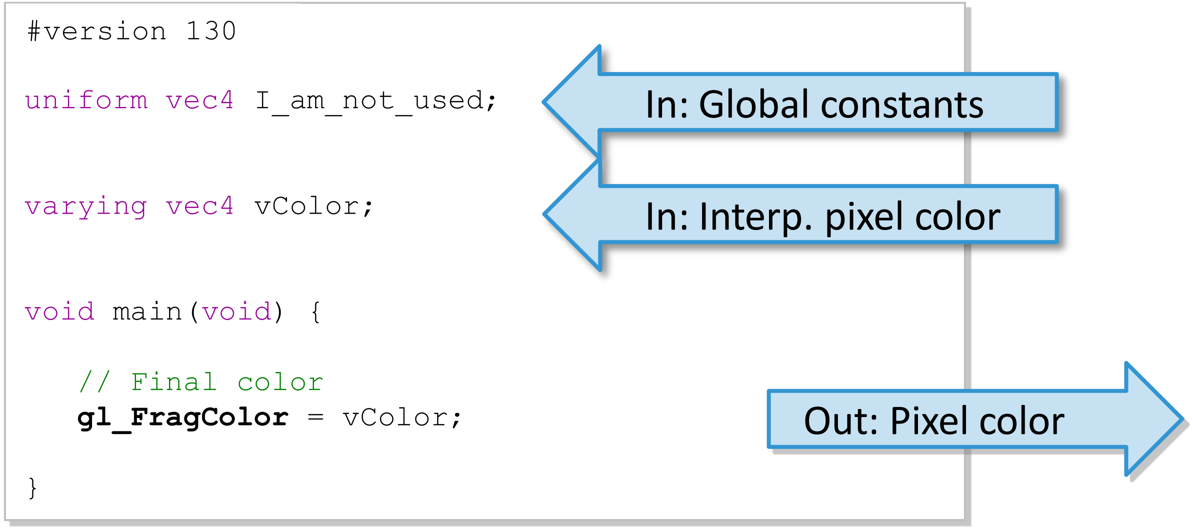
In: Global constants

In: Per-vertex attribs

Out: Vertex color

Out: Vertex pos.

Fragment shader example



```
#version 130

uniform vec4 I_am_not_used;


varying vec4 vColor;


void main(void) {

   // Final color
   gl_FragColor = vColor;

}
```

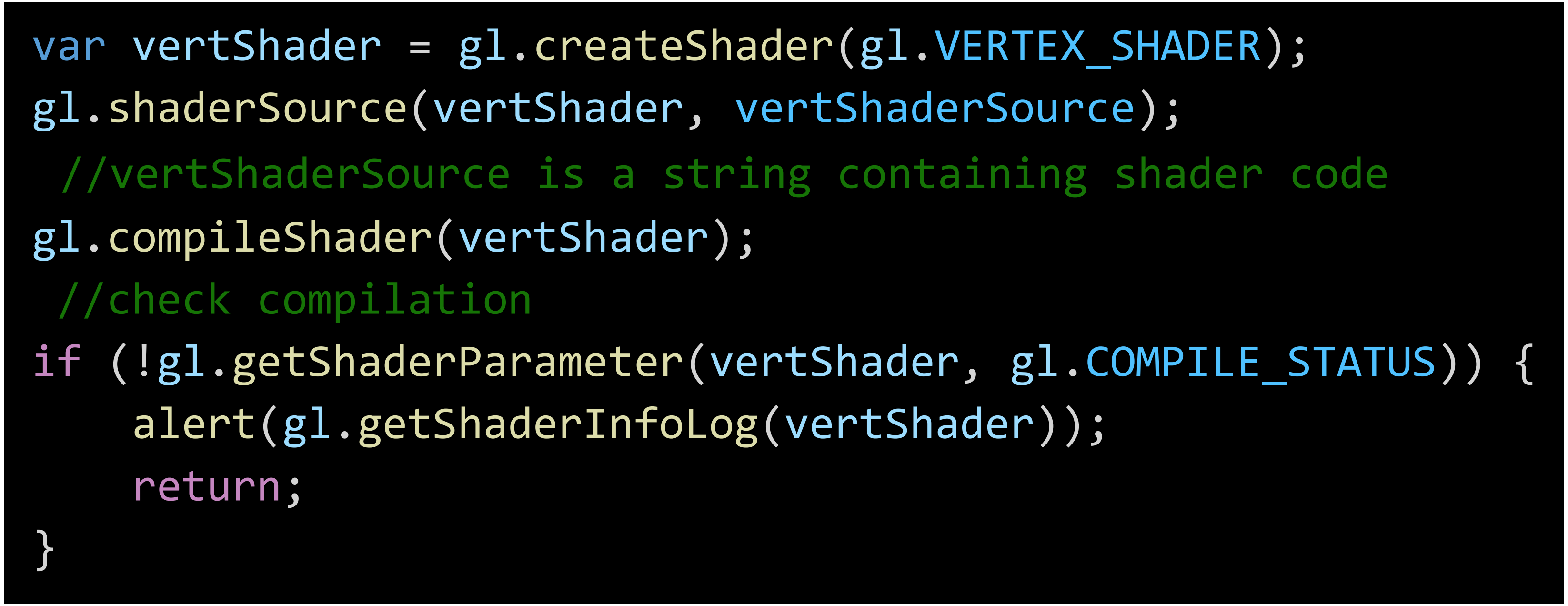
In: Global constants

In: Interp. pixel color

Out: Pixel color

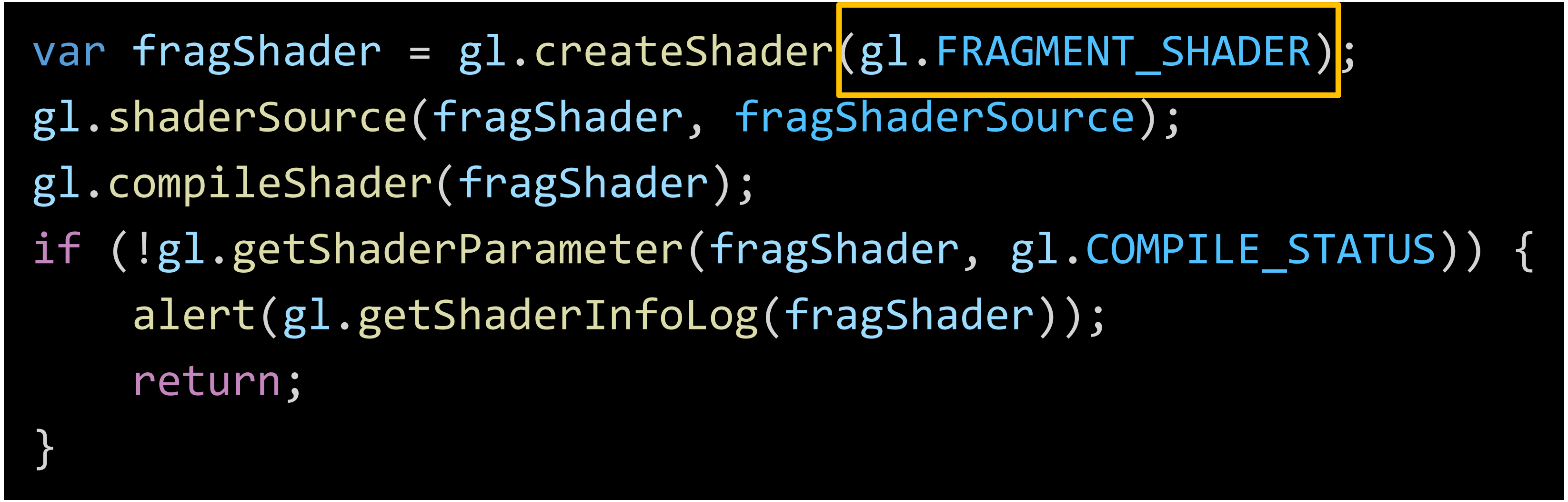
(Trivial shader)

Link Shader Programs



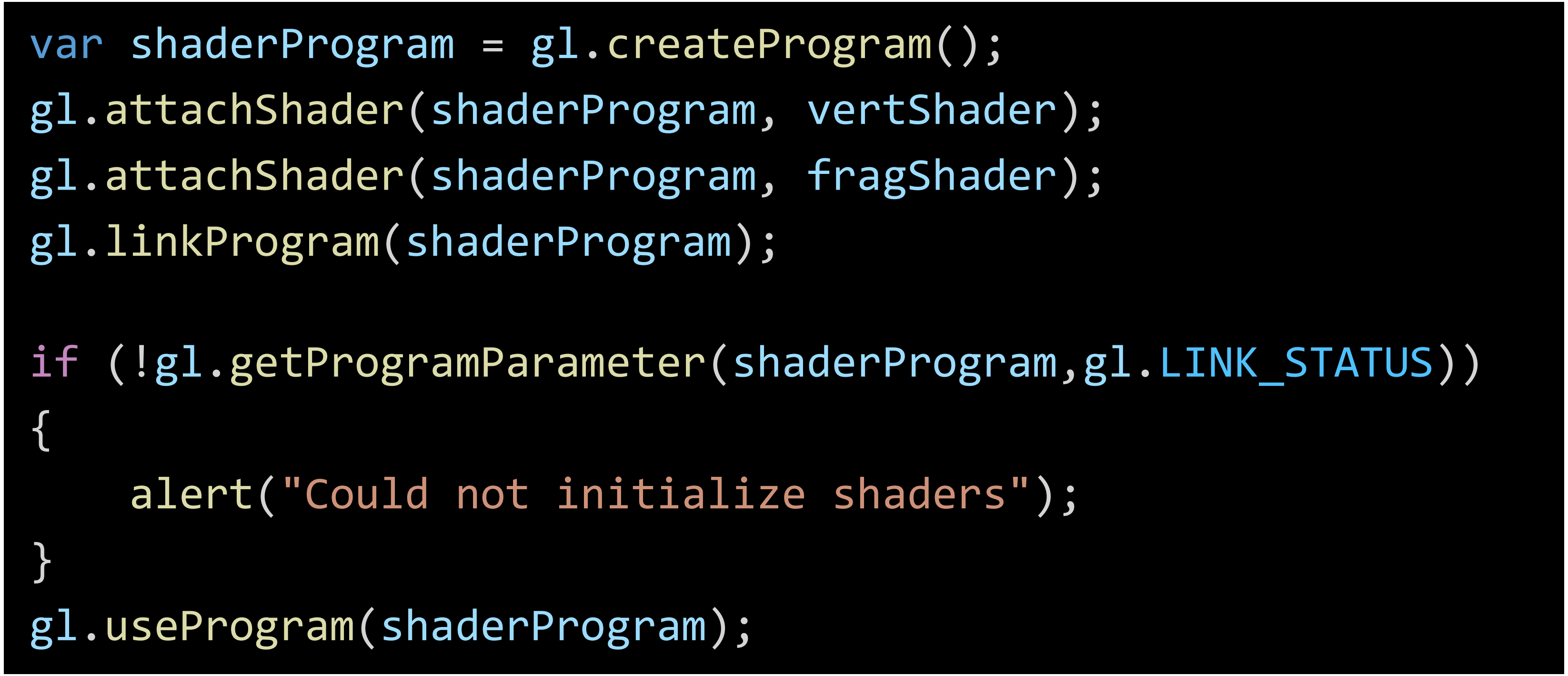
```
var vertShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertShader, vertShaderSource);
 //vertShaderSource is a string containing shader code
gl.compileShader(vertShader);
 //check compilation
if (!gl.getShaderParameter(vertShader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(vertShader));
    return;
}
```

Link Shader Programs



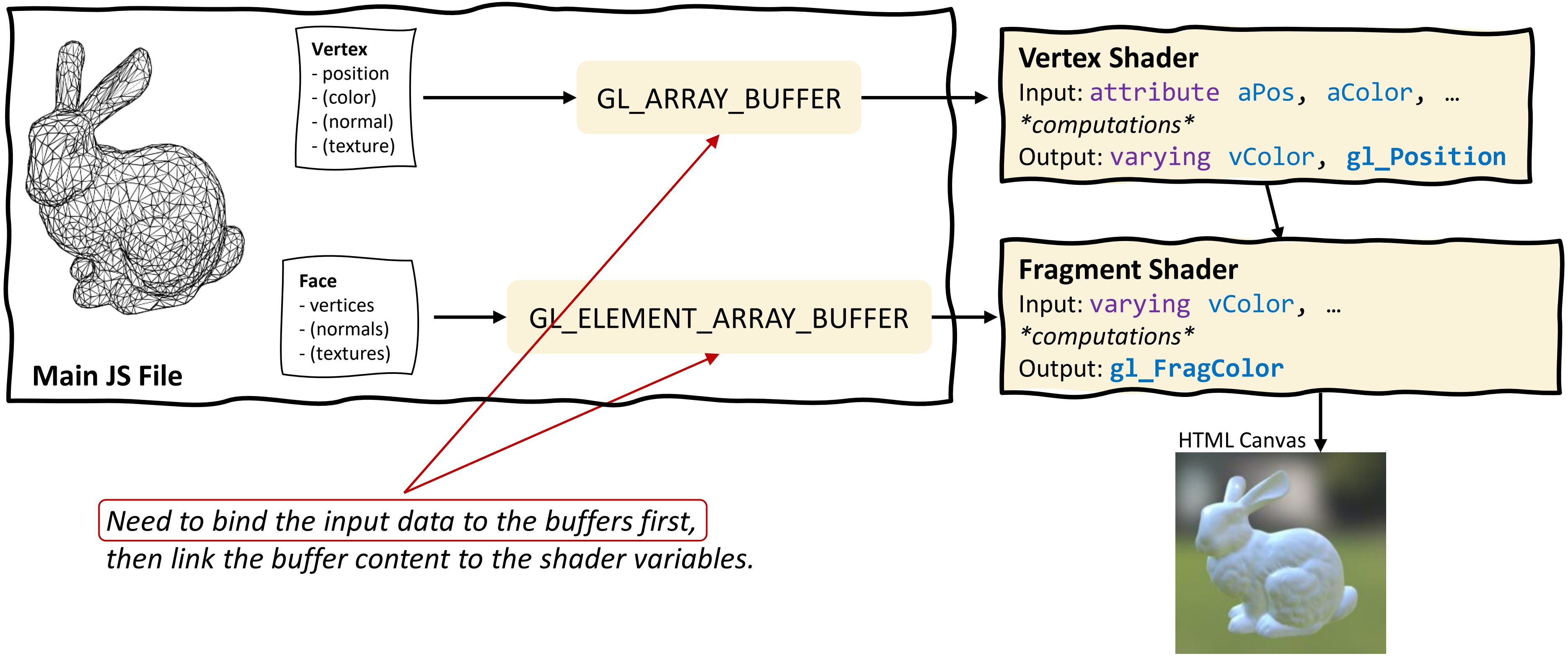
```
var fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, fragShaderSource);
gl.compileShader(fragShader);
if (!gl.getShaderParameter(fragShader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(fragShader));
    return;
}
```

Link Shader Programs



```
var shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertShader);
gl.attachShader(shaderProgram, fragShader);
gl.linkProgram(shaderProgram);

if (!gl.getProgramParameter(shaderProgram,gl.LINK_STATUS))
{
    alert("Could not initialize shaders");
}
gl.useProgram(shaderProgram);
```

GL Pipeline



Main JS File

Vertex
- position
- (color)
- (normal)
- (texture)

Face
- vertices
- (normals)
- (textures)

GL_ARRAY_BUFFER

GL_ELEMENT_ARRAY_BUFFER

Vertex Shader

Input: `attribute aPos, aColor,` …

computations

Output: `varying vColor, gl_Position`

Fragment Shader

Input: `varying vColor,` …

computations

Output: `gl_FragColor`

HTML Canvas

Need to bind the input data to the buffers first, then link the buffer content to the shader variables.

Link Uniform Variables

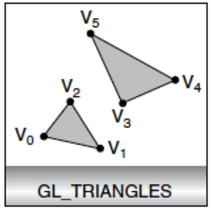
```
gl.useProgram(shaderProgram); // can have multiple
const uForegroundColor = gl.getUniformLocation(shaderProgram,
                              'uForegroundColor’);


let col = vec3.fromValues(1., 1., 1.);
gl.uniform3f(uForegroundColor, col[0], col[1], col[2]);


gl is from the webgl canvas, 3 dimension, f float
```

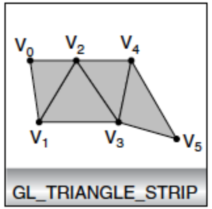
Link Attribute Variables

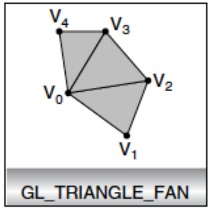
```
Remember this?
    let vertexData = new Float32Array(vertexPositions);
    const vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);
Add this to link the attribute
    const aVertexPosition = gl.getAttribLocation(shaderProgram,
    "aVertexPosition");
    // 2 entries, float, no normalization, no stride, no offset
    gl.vertexAttribPointer(aVertexPosition, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(aVertexPosition);
```

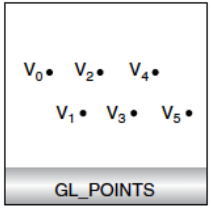
Render it!

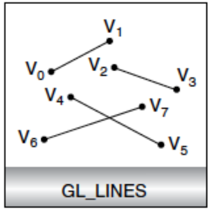
```
gl.drawArrays(gl.TRIANGLES, 0, 3);
// or
gl.drawElements(gl.TRIANGLES, nVertices, gl.UNSIGNED_SHORT, 0);
```

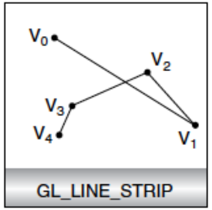


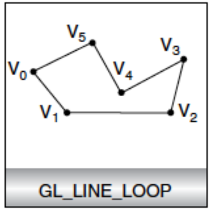










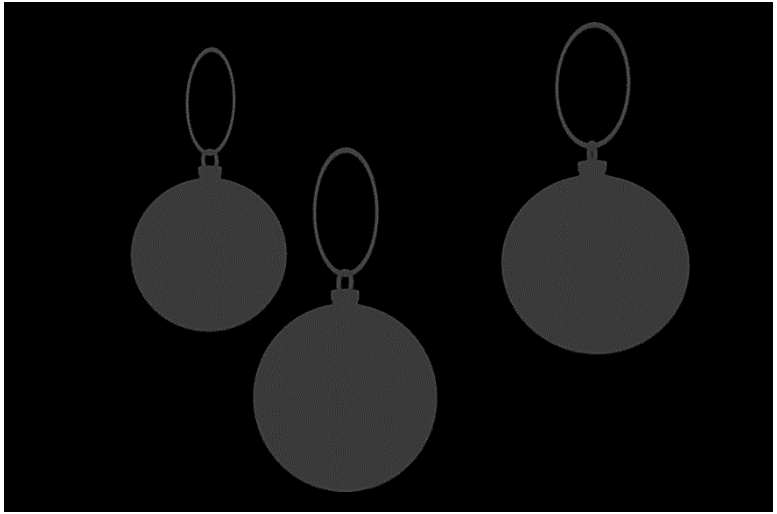


Overview

- Introduction to WebGL
- Graphics Pipeline
- Code Template
  - Initialization
  - Shaders
  - Drawing a Triangle
- Exercise 6

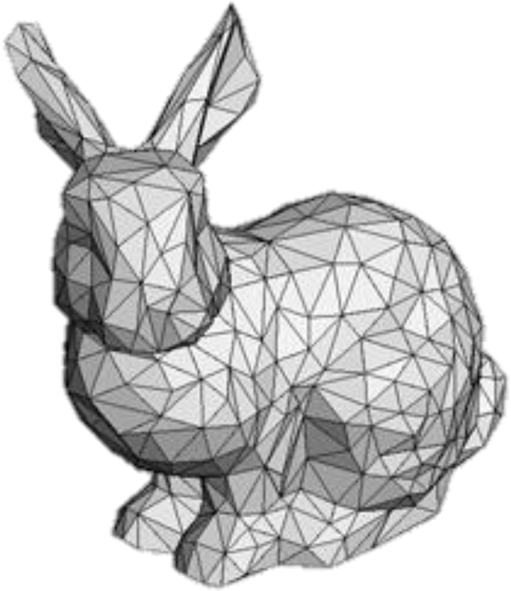
1) Mesh setup and initialization

- Setup vertex buffer and index buffer
- Pass data to vertex shader



Mesh representations

- Focus on triangle meshes
- 3D mesh consists of:
  - vertices
  - faces
- Information stored:
  - vertex: position, color, normal, …
  - face: links to vertices, surface normal, …

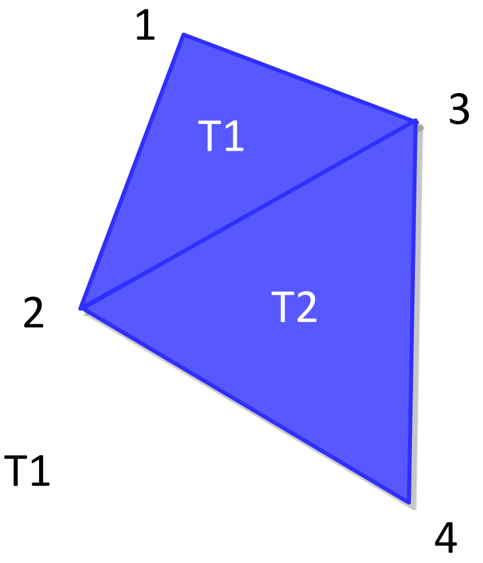


Mesh representations

- Indexed triangle list
  - Stores vertices only once
  - Define triangles by indexing

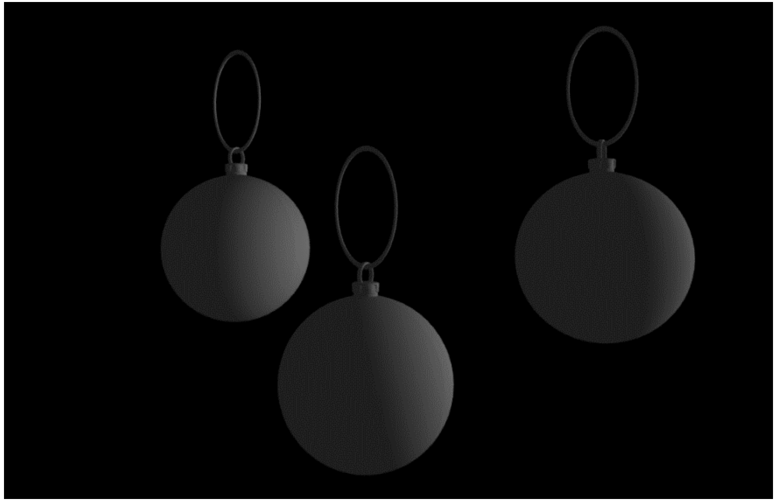
Vertex list
1: (x1, y1, z1)
2: (x2, y2, z2)
3: (x3, y3, z3)
4: (x4, y4, z4)

Index list	
1	T1
2	
3	
4	T2
3	
2	



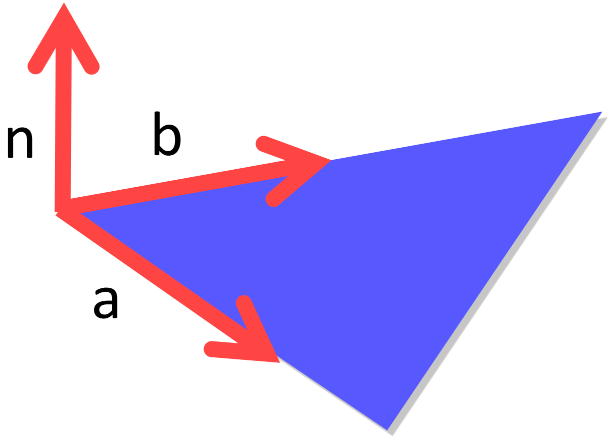
2) Normals for lighting

- Calculate face and vertex normals



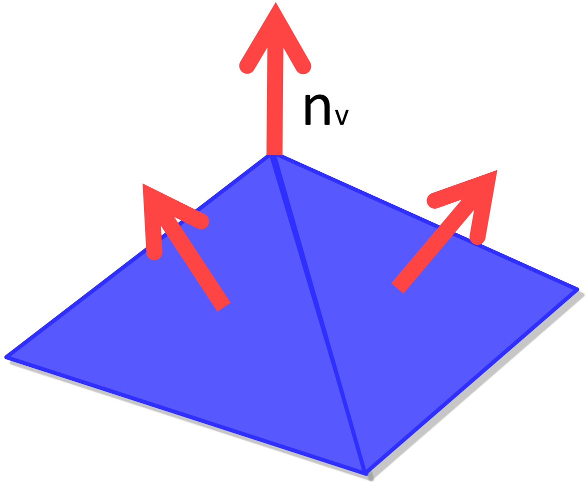
Normals

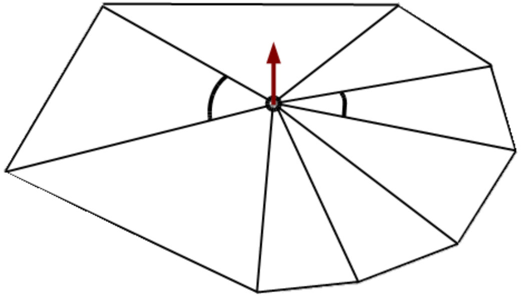
- Face normals
  - Normalized cross product of a and b



Normals

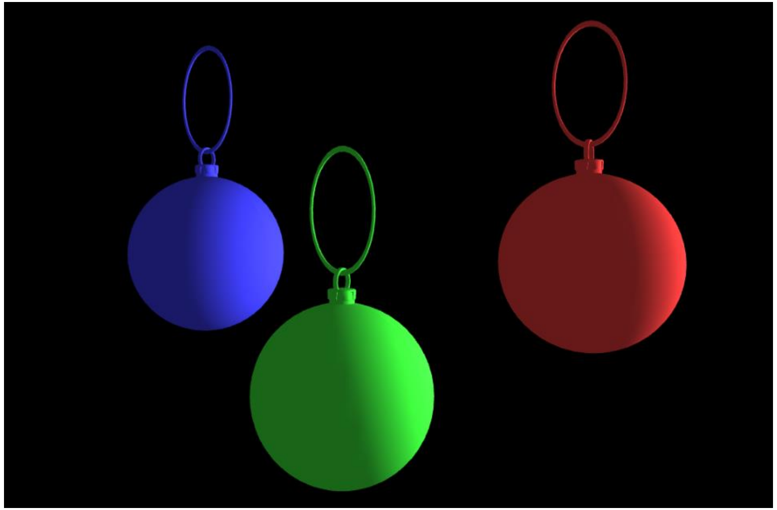
- Vertex normal
  - Average of surrounding face normals
  - Actually, better to weight according to angles (optional)





3) Coloring the mesh

- Color vertices depending on position
- Rotate the scene



Questions

?

nikolak@inf.ethz.ch