

Welcome to 02561: Computer Graphics

Jeppe Revall Frisvad

September 2020

Web graphics, drawing in 2D, animation, interaction (weeks 1–3)

- ▶ Three weeks for covering the basics.

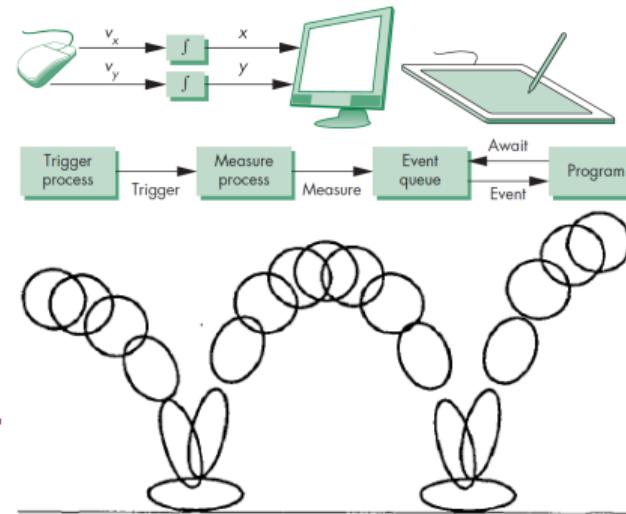
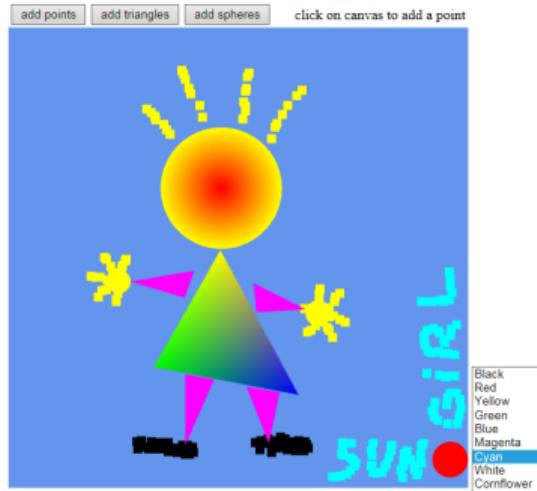


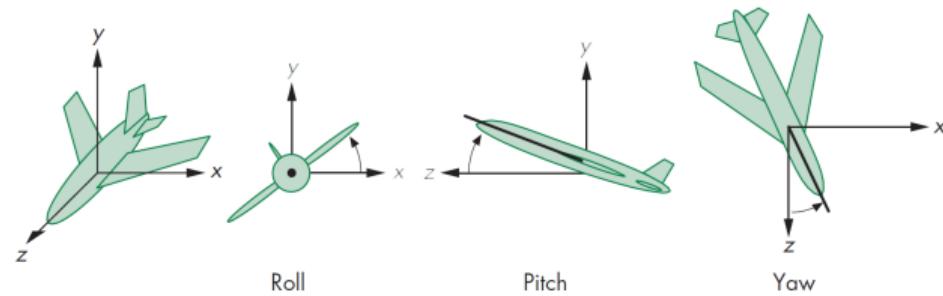
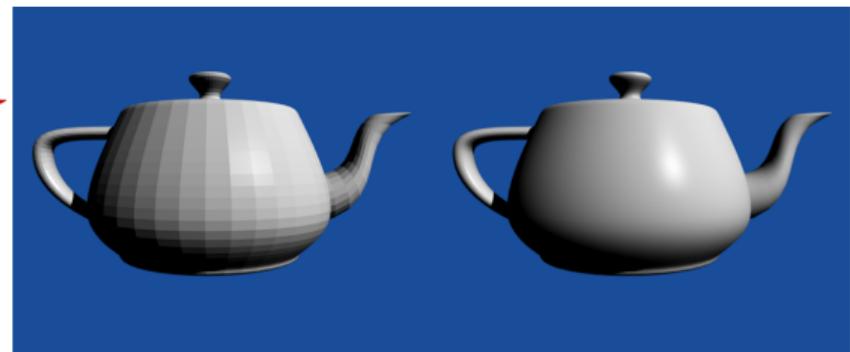
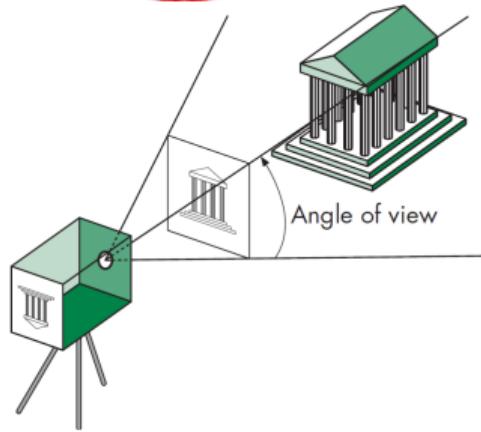
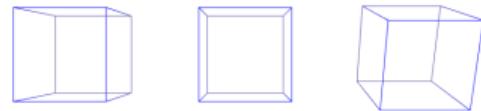
FIGURE 2. Squash & stretch in bouncing ball.

Figure references

- Worksheet 2 and Wikimedia Commons.
- Angel, E., and Shreiner, D. *Interactive Computer Graphics: A Top-Down Approach with WebGL*, seventh edition. Pearson, 2015.
- Lasseter, J. Principles of traditional animation applied to 3D computer animation. *Computer Graphics (SIGGRAPH '87) 21(4):35-44*, 1987.

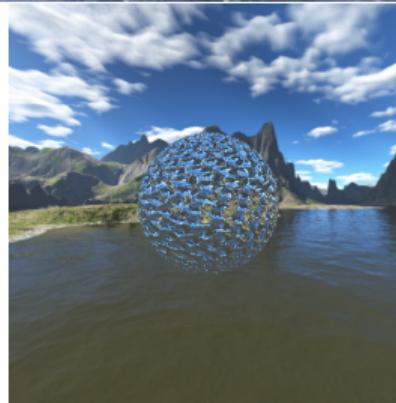
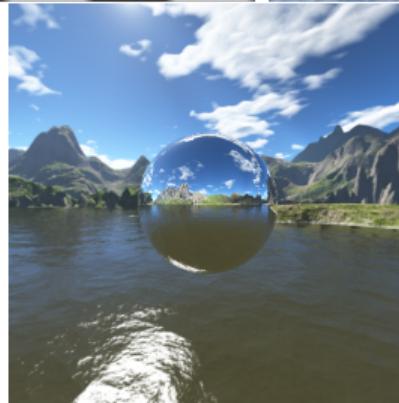
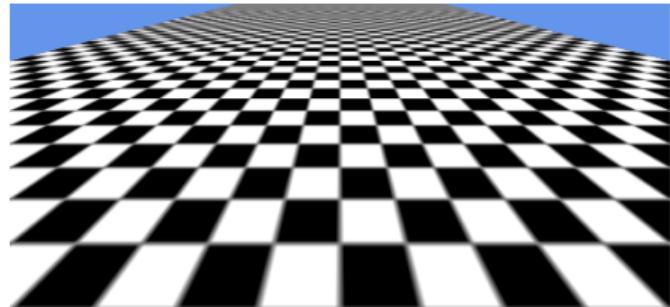
Drawing in 3D, triangle meshes, viewing, lighting (weeks 4–6)

- ▶ Three weeks for adding depth.



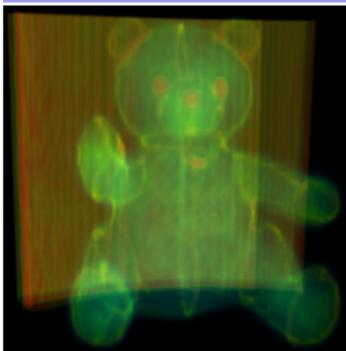
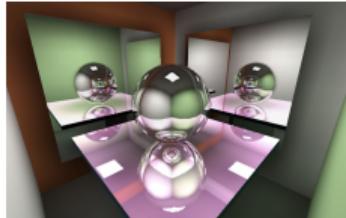
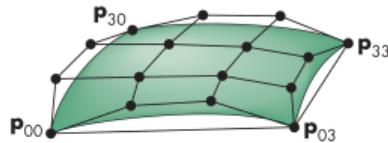
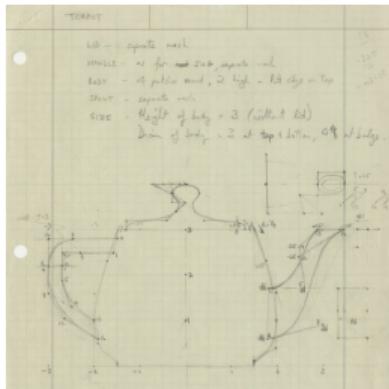
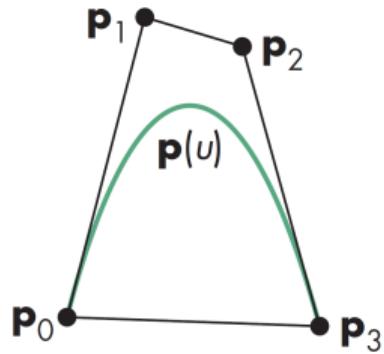
Mapping, blending, off-screen buffers, multiple shaders (weeks 7–10)

- ▶ Four weeks for shading pixels.



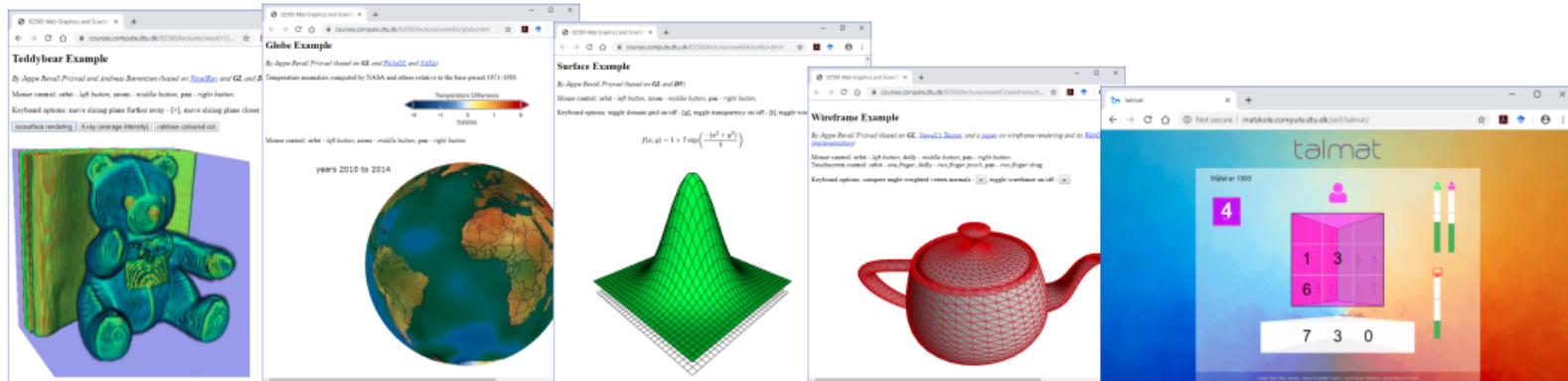
Advanced techniques (weeks 11–13)

- ▶ Three weeks for curves, surfaces, volumes, and visual effects.



Why web graphics?

- ▶ Develop 3D graphics applications using only a text editor and a browser.
- ▶ Easy sharing of 3D graphics on all platforms (mobile included).
- ▶ Combine scientific reporting with interactive visualization in a webpage.
- ▶ Effortless integration of graphics with browser UI and image file I/O.



Instructors and teaching style

- ▶ Course responsible: Jeppe Revall Frisvad
- ▶ TA (online): Rasmus Borstrøm, s174298@student.dtu.dk
- ▶ Lectures and exercises interspersed (8:30-10:30)
- ▶ Exercises (10:30-12:30)
- ▶ Lab journal: Save a set of solution files for each part of each worksheet.
- ▶ Project: Select, study and implement, describe in a report.
- ▶ Hand-in: HTML and/or PDF (include all code in a ZIP file)
- ▶ Deadline: **20 December 2020 at 23:59**

02561 Computer Graphics

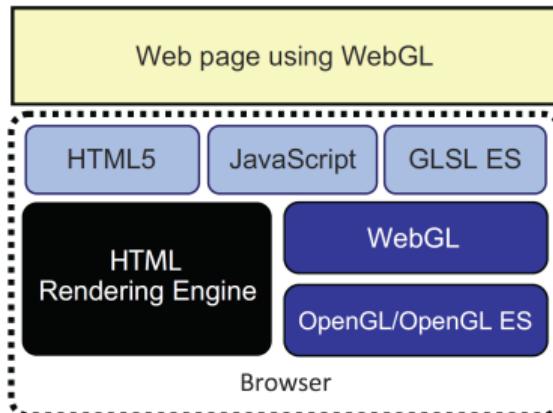
WebGL and Graphics Pipeline

Jeppe Revall Frisvad

September 2020

How to work with WebGL

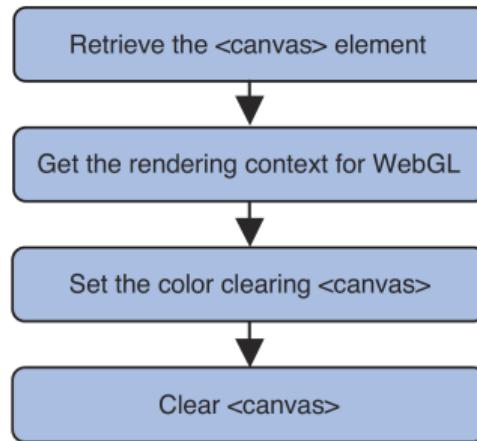
- ▶ Select a browser and check its compatibility:
<https://webglreport.com/>
- ▶ Select an editor. Some provide more code completion help than others.
- ▶ Create a webpage (HTML file) with
 - ▶ **HTML5** canvas element
 - ▶ **JavaScript** (in a separate JS file)
 - ▶ vertex shader script (**GLSL ES**)
 - ▶ fragment shader script (**GLSL ES**)
- ▶ Load utility libraries from existing JS files (webgl-utils.js, initShaders.js, MV.js) [A: 2.8]
- ▶ Open HTML file in browser and debug using “**Inspect [Element]**” <F12>
- ▶ Re-run the program after an edit by **reloading** the webpage <F5>
- ▶ Save your graphic as an image by right-clicking the canvas (not all browsers)



Exercise: getting started (W01P1)

- ▶ Minimal program (quick reference card)

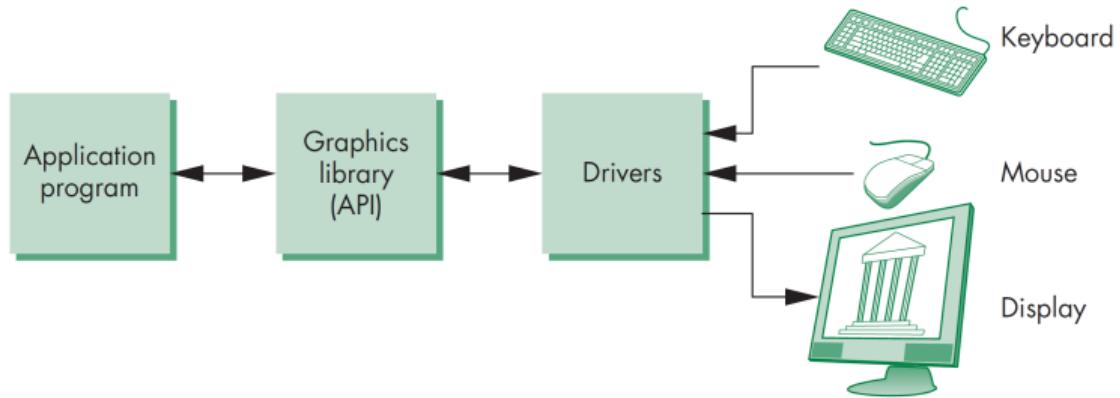
```
<!DOCTYPE html>
<html><body>
<canvas id="c" />
<script type="text/javascript">
    var canvas = document.getElementById("c");
    var gl = canvas.getContext("webgl");
    gl.clearColor(1.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
</script>
</body></html>
```



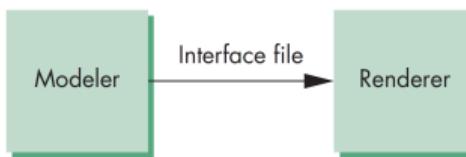
- ▶ Get the library files from angelCommon.zip on DTU Inside File Sharing.
- ▶ Use **A**: 2.8 to set canvas size (this is the resolution of the rendered image) and an init function (in filename.js) and solve the first part of the first worksheet.

Interactive computer graphics

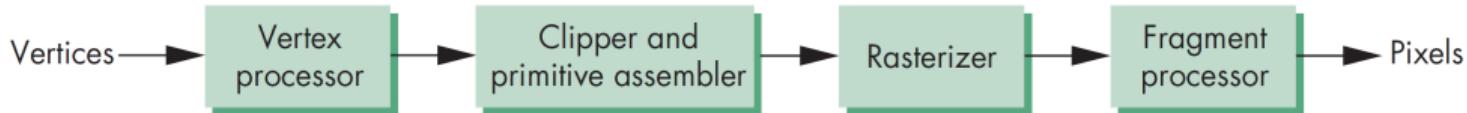
- ▶ **Graphics system:**



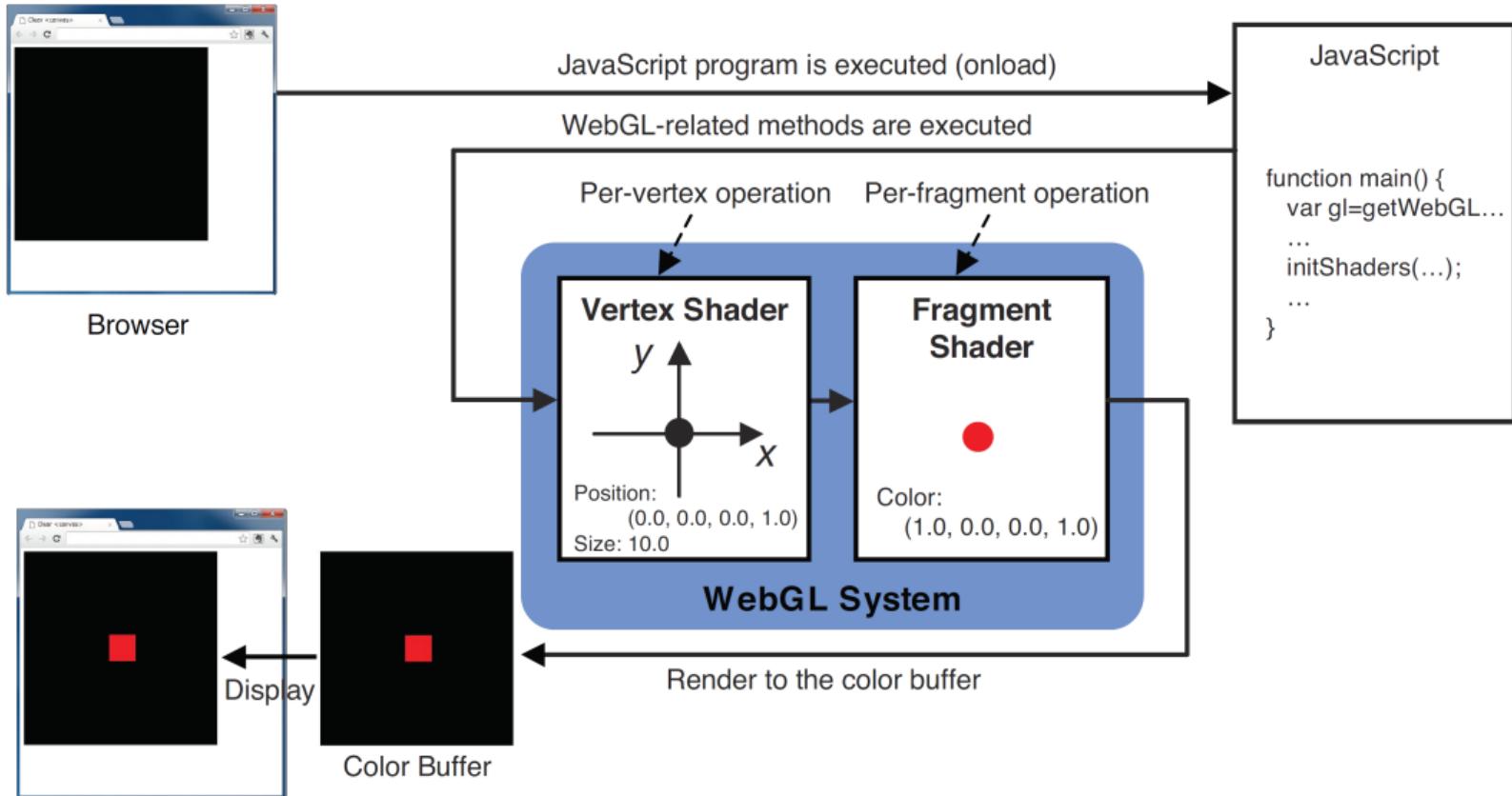
- ▶ **Modeling-rendering paradigm:**



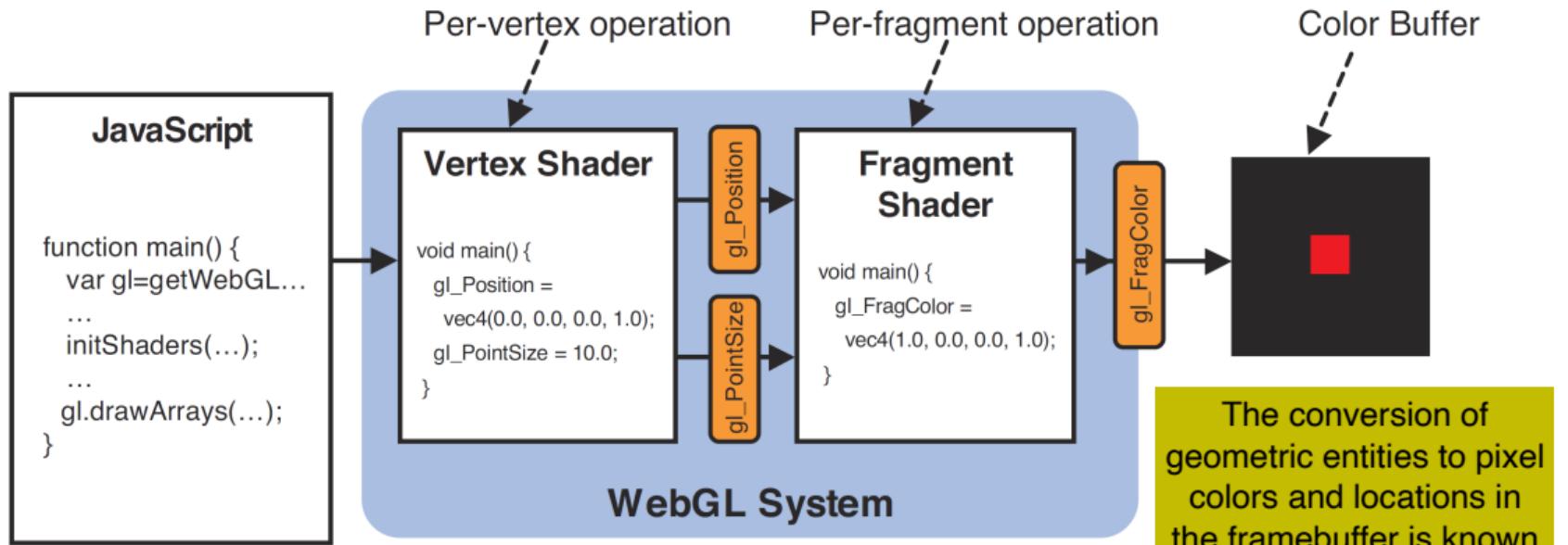
- ▶ **Rasterization pipeline:**



Shaders and processing flow



WebGL shaders in practice



The conversion of geometric entities to pixel colors and locations in the framebuffer is known as rasterization or scan con- version.

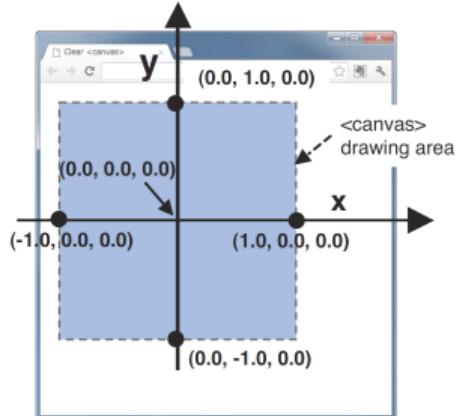
► A: 2.8

```
<script type="text/javascript" src="../common/initShaders.js"></script>
```

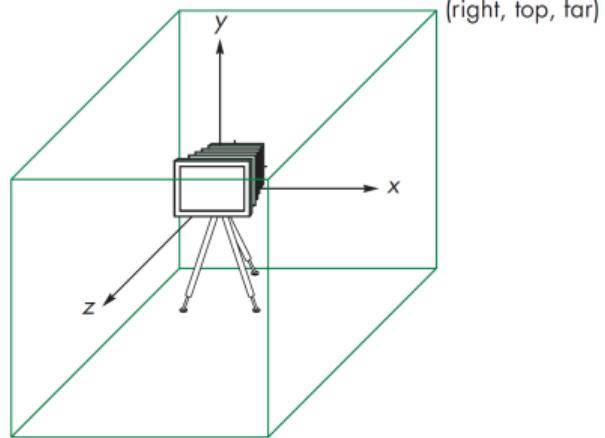
```
<script id="vertex-shader" type="x-shader/x-vertex">  
void main() {  
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);  
    gl_PointSize = 10.0;  
}  
</script>
```

```
<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>
```

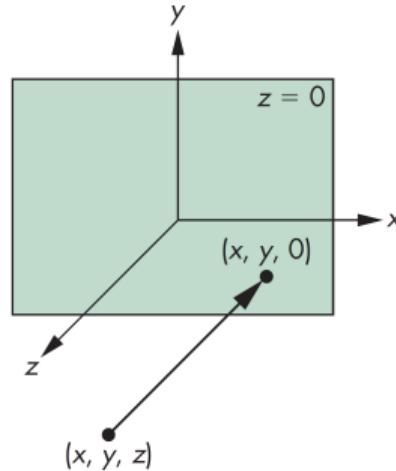
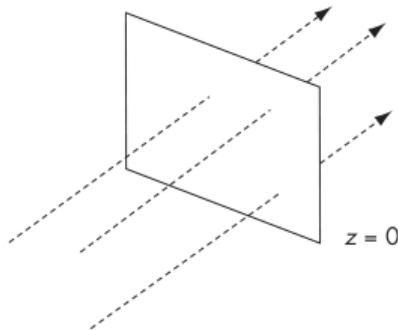
Drawing in 2D



(left, bottom, near)



- ▶ The default camera: $x, y, z \in [-1, 1]$.
- ▶ View volume and orthographic projection.



Attribute variables and data flow

- ▶ Stream data from the CPU to draw a number of points using the same shader:

```
<script id="vertex-shader" type="x-shader/x-vertex">
    attribute vec4 a_Position;
    void main() {
        gl_Position = a_Position;
        gl_PointSize = 10.0;
    }
</script>
<script type="text/javascript" src="../common/MV.js"></script>
```

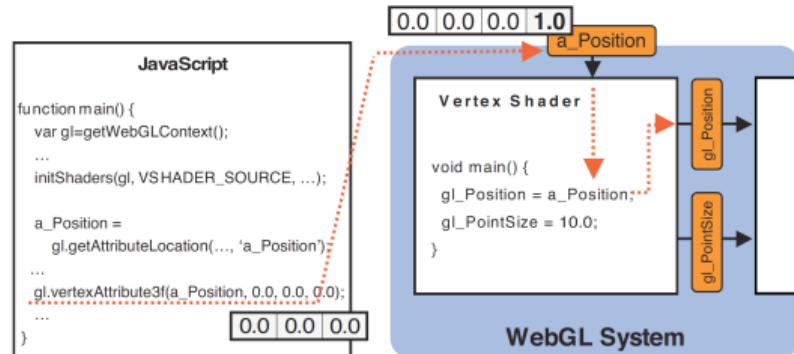
Storage Qualifier Type Variable Name
attribute vec4 a_Position;

- ▶ In JavaScript init/main function:

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

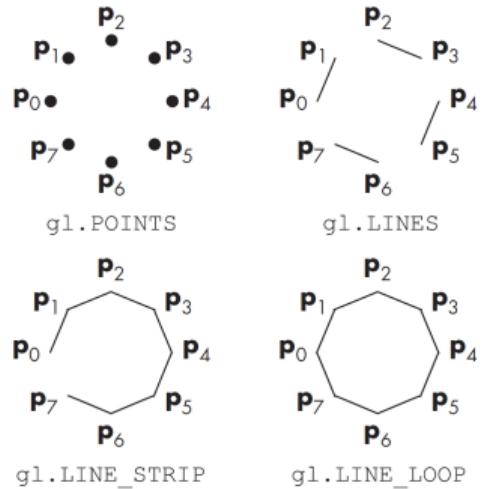
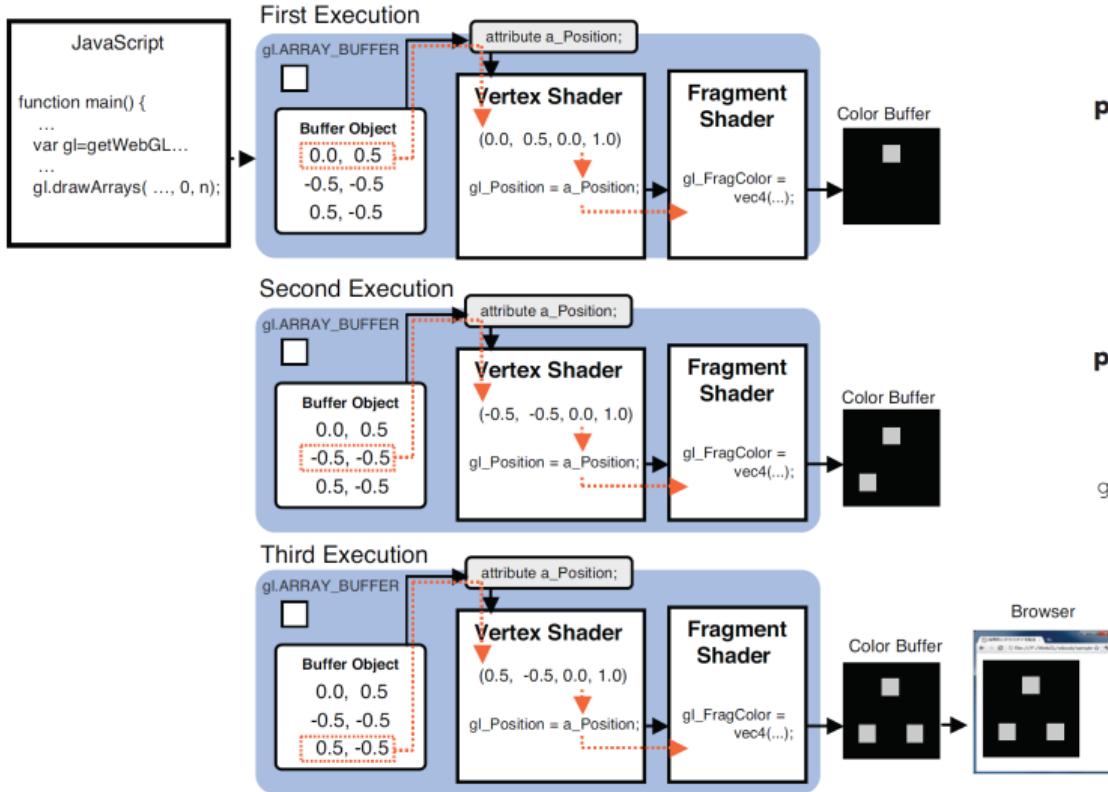
var vertices = [ vec2(0.0, 0.5), vec2(-0.5, -0.5), vec2(0.5, -0.5) ];
var vBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);

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



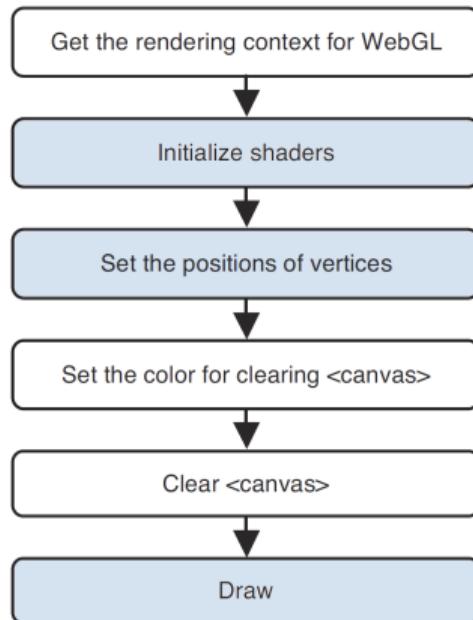
Start drawing

- Execution starts at the draw call. Example: `gl.drawArrays(gl.POINTS, 0, numPoints);`



Exercise: hello points (W01P2)

- ▶ Copy and extend your solution for Part 1.
- ▶ Write, load, and use a shader program (**A**: 2.8.3-2.8.7).
- ▶ Set point size in the vertex shader (**A**: 2.5.3).
- ▶ Define point coordinates (array of vectors).
- ▶ Connect attribute variable to buffer (**A**: 2.8):
 - ▶ Create a buffer.
 - ▶ Bind the buffer object to a target.
 - ▶ Submit data to the buffer.
 - ▶ Assign the buffer object to an attribute variable.
 - ▶ Enable assignment.
- ▶ Draw the points after clearing (**A**: 2.4, 2.8.2).

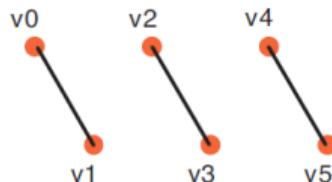


Basic shapes

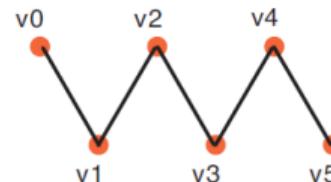
- Another draw call example: `gl.drawArrays(gl.TRIANGLES, 0, numVertices);`



`gl.POINTS`



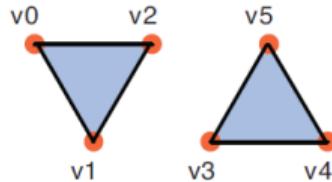
`gl.LINES`



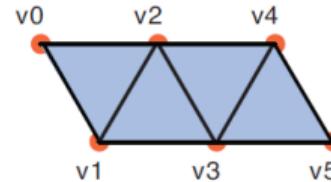
`gl.LINE_STRIP`



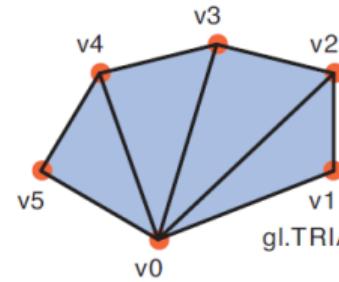
`gl.LINE_LOOP`



`gl.TRIANGLES`



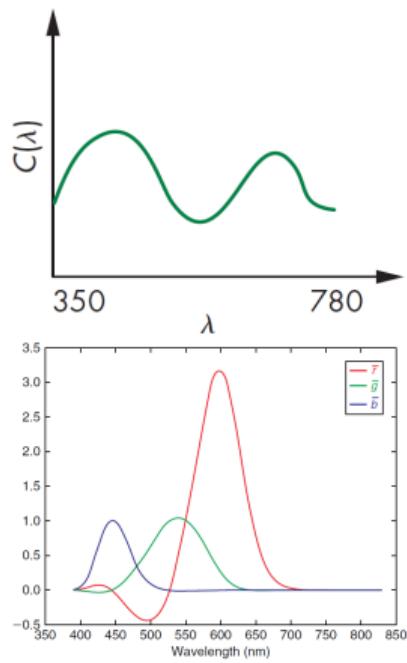
`gl.TRIANGLE_STRIP`



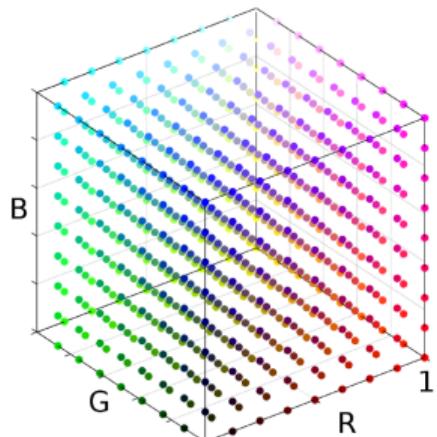
`gl.TRIANGLE_FAN`

Colors

spectrum

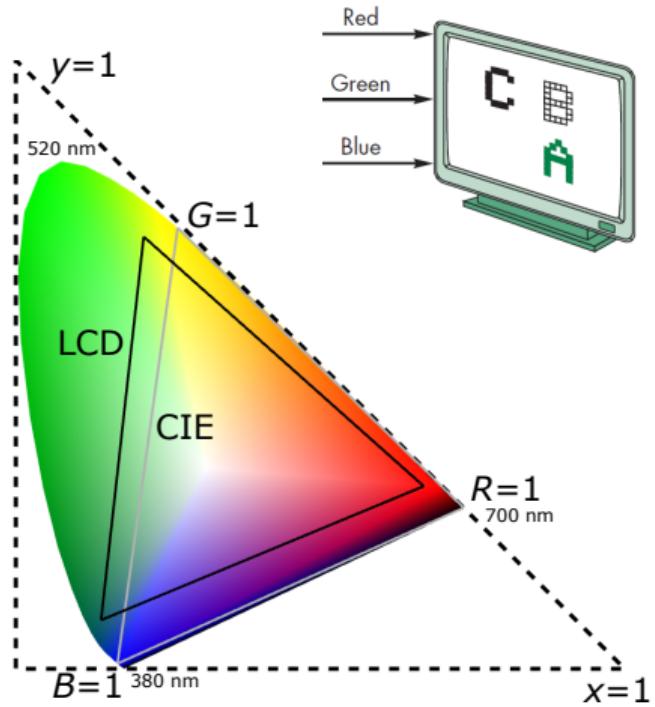


RGB color space



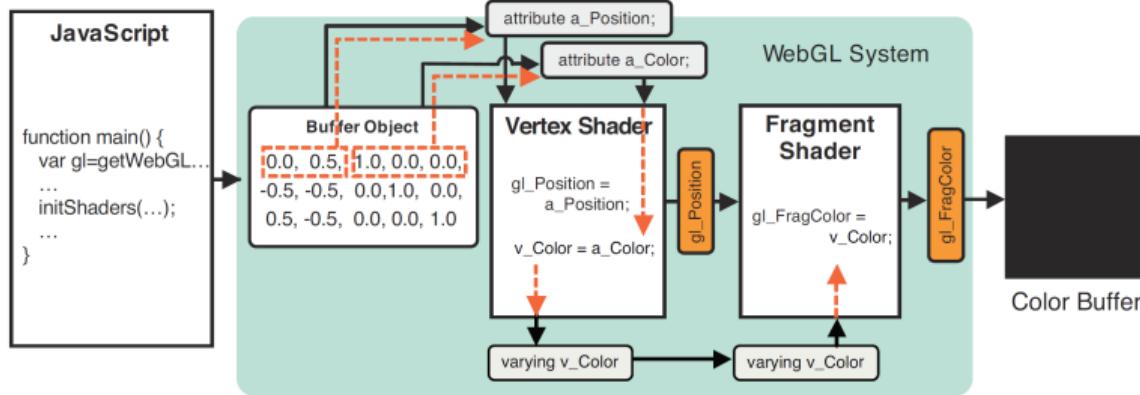
$$R = \int_{\gamma} C(\lambda) \bar{r}(\lambda) d\lambda$$
$$G = \int_{\gamma} C(\lambda) \bar{g}(\lambda) d\lambda$$
$$B = \int_{\gamma} C(\lambda) \bar{b}(\lambda) d\lambda$$

chromaticity and gamut



From vertex to fragment

- ▶ We can attach color as a vertex attribute just like position (see above).
- ▶ Pass data from vertex to fragment shader using a varying variable.

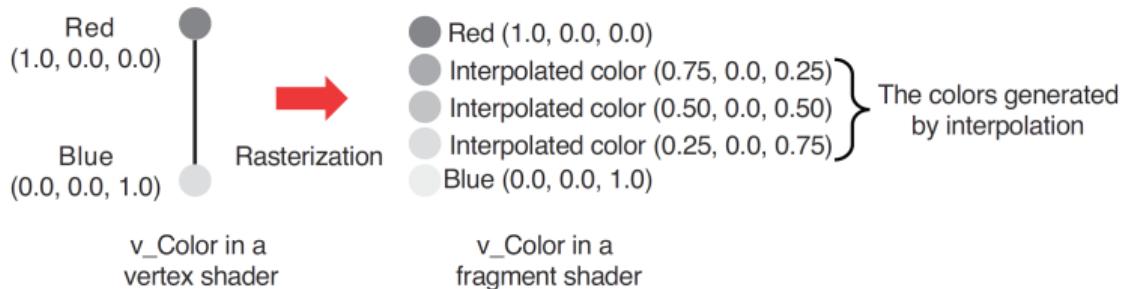
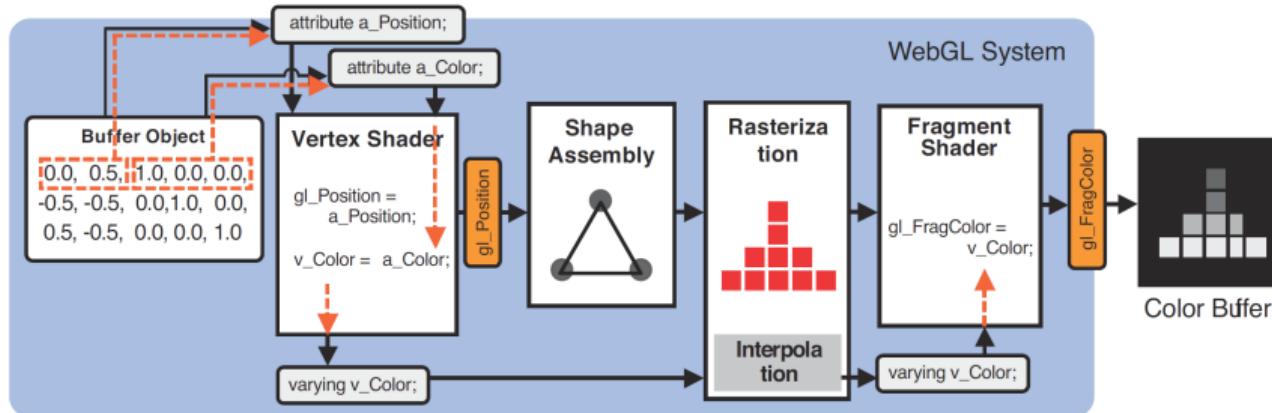
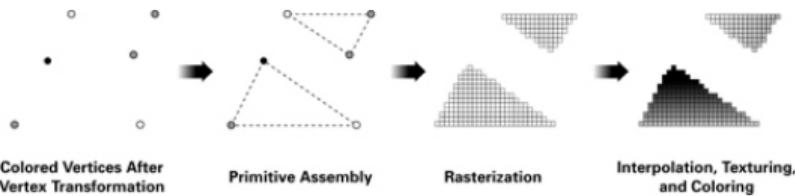


- ▶ A: 2.10

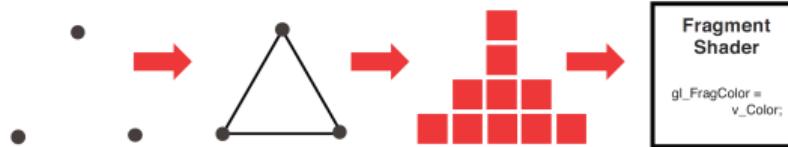
```
<script id="vertex-shader" type="x-shader/x-vertex">  
attribute vec4 a_Position;  
attribute vec4 a_Color;  
varying vec4 v_Color;  
void main() {  
    v_Color = a_Color;  
    gl_Position = a_Position;  
}  
</script>
```

```
<script id="fragment-shader" type="x-shader/x-fragment">  
precision mediump float;  
varying vec4 v_Color;  
void main() {  
    gl_FragColor = v_Color;  
}  
</script>
```

Rasterization and interpolation



Exercise: hello triangle (W01P3)



Fragment
Shader

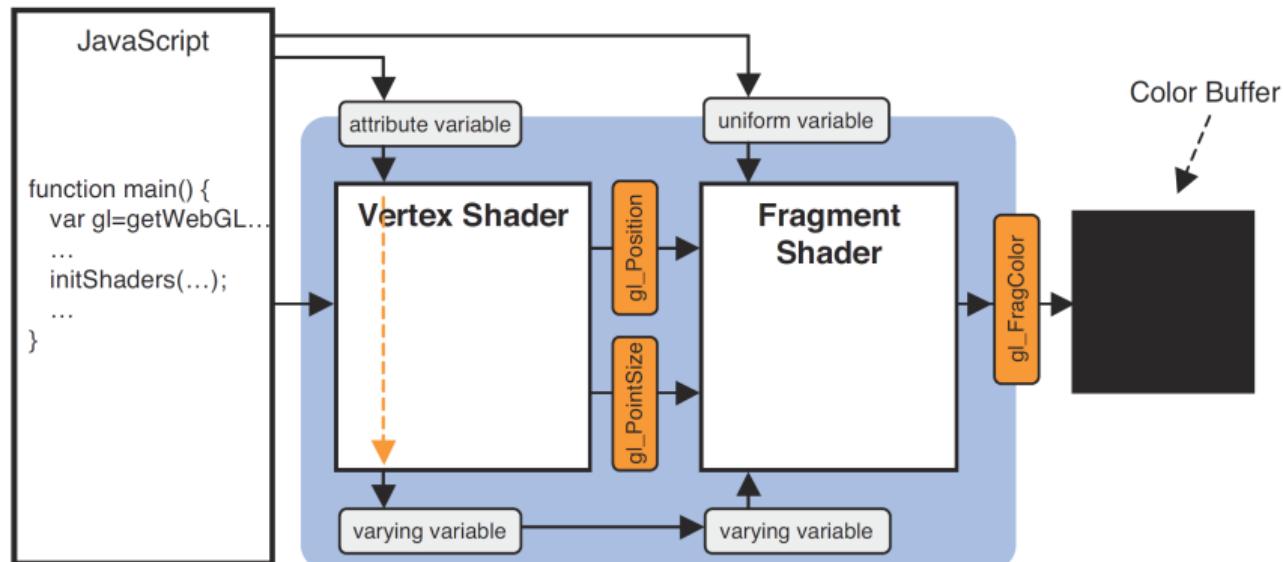
```
gl_FragColor =  
v_Color;
```

- ▶ Copy and extend your solution for Part 2.
- ▶ Update the vertex shader to include a color attribute that is passed to the fragment shader as a varying variable.
- ▶ Define colors (array of vectors).
- ▶ Connect color attribute variable to buffer:
 - ▶ Create a buffer.
 - ▶ Bind the buffer object to a target.
 - ▶ Submit data to the buffer.
 - ▶ Assign the buffer object to an attribute variable.
 - ▶ Enable assignment.
- ▶ Draw a triangle after clearing.

Uniform variables

- ▶ Use uniform variables to pass parameters that do not change per vertex.
- ▶ Uniforms can be used in both vertex and fragment shader.

Storage Qualifier Type Variable Name
uniform vec4 u_FragColor;



```
var u_FragColorLoc = gl.getUniformLocation(program, "u_FragColor");  
gl.uniform4f(u_FragColorLoc, rgba[0], rgba[1], rgba[2], rgba[3]);
```

Drawing a circle

- ▶ Create a position array and a color array containing the center vertex only.
- ▶ Select radius r and number of points n to represent the circle.
- ▶ Loop from $i = 0$ to $i = n$:
 - ▶ Calculate angle $\theta = 2\pi i / n$ (π is available as `Math.PI`).
 - ▶ Use radius, sine (`Math.sin`), and cosine (`Math.cos`) to calculate a position $(r \cos \theta, r \sin \theta)$ on the circle.
 - ▶ Push position and color to corresponding arrays.
- ▶ Submit data to position and color attribute buffers.
- ▶ Draw as a triangle fan with $n + 2$ vertices.

