

# Welcome to 02561: Computer Graphics

Jeppe Revall Frisvad

August 2023

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

- ▶ Three weeks for covering the basics.

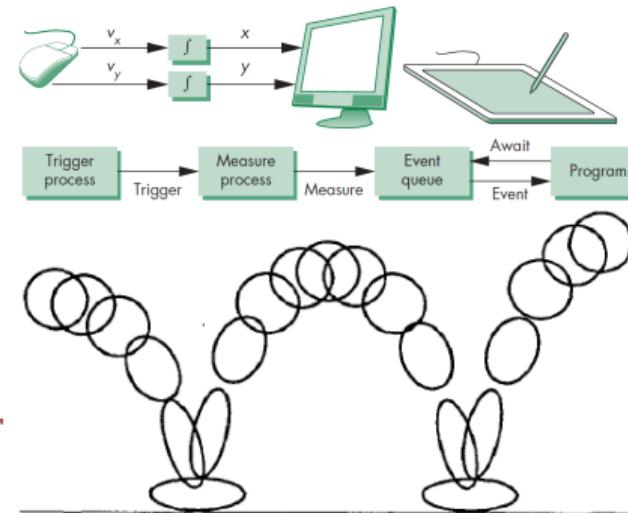
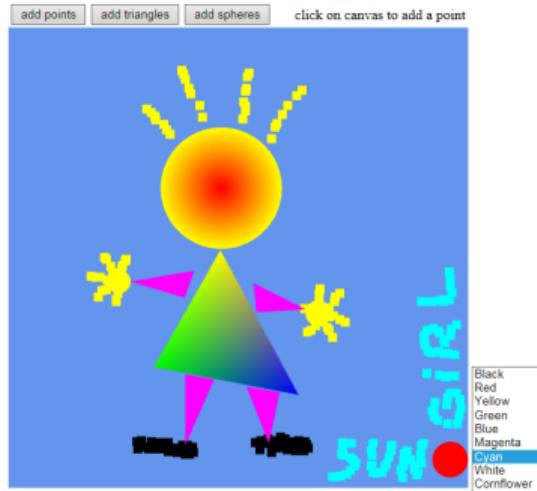


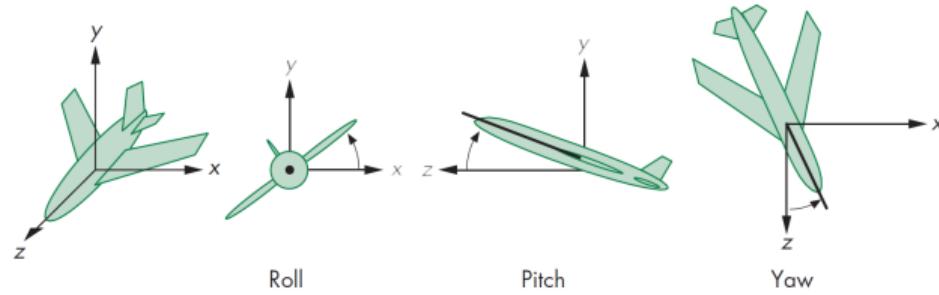
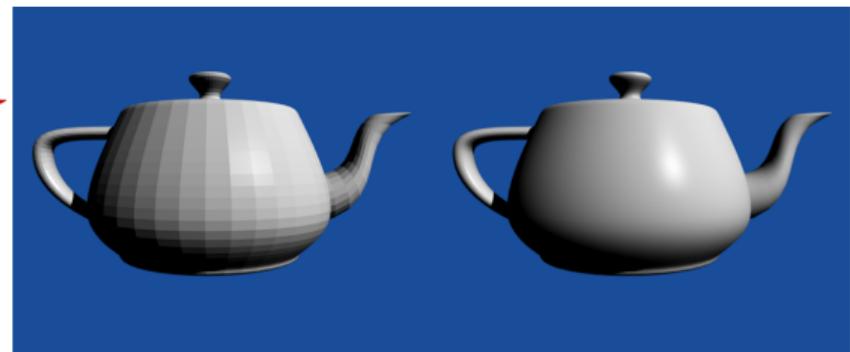
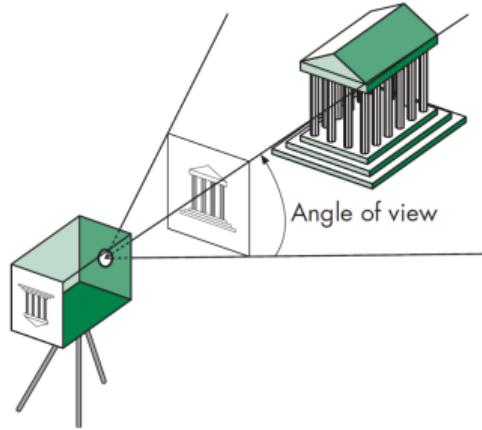
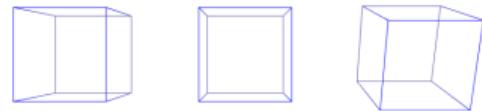
FIGURE 2. Squash & stretch in bouncing ball.

## 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.

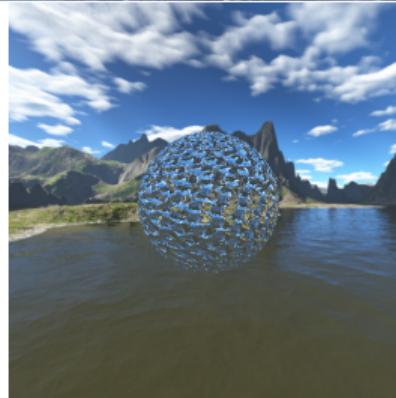
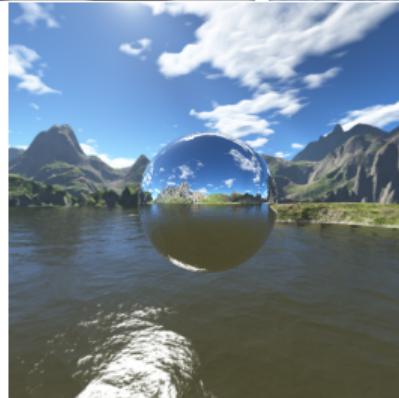
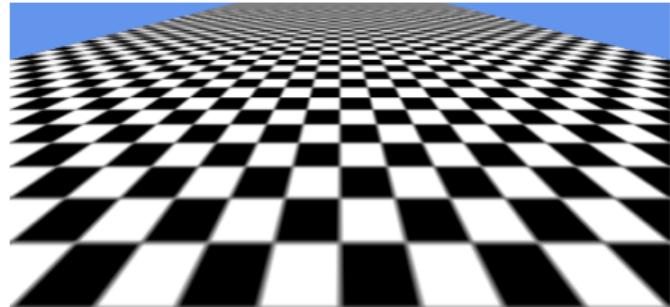
# Drawing in 3D, triangle meshes, viewing, lighting (weeks 4–6)

- ▶ Three weeks for adding depth.



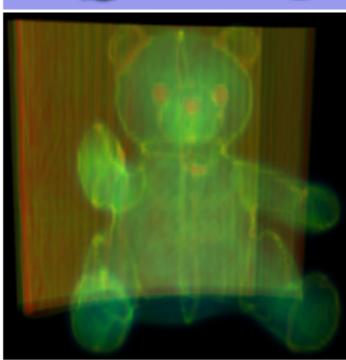
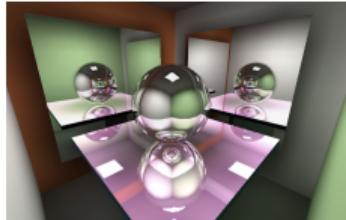
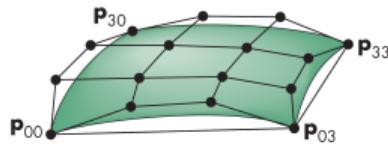
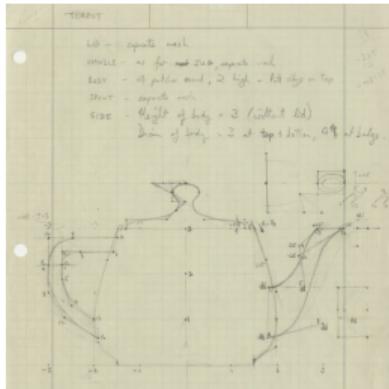
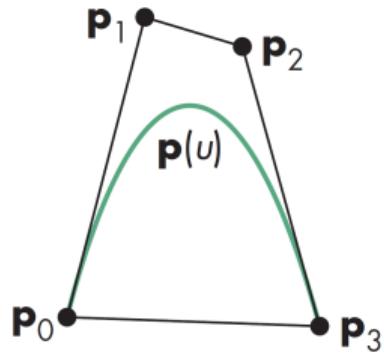
# Mapping, blending, off-screen buffers, multiple shaders (weeks 7–10)

- ▶ Four weeks for shading pixels.



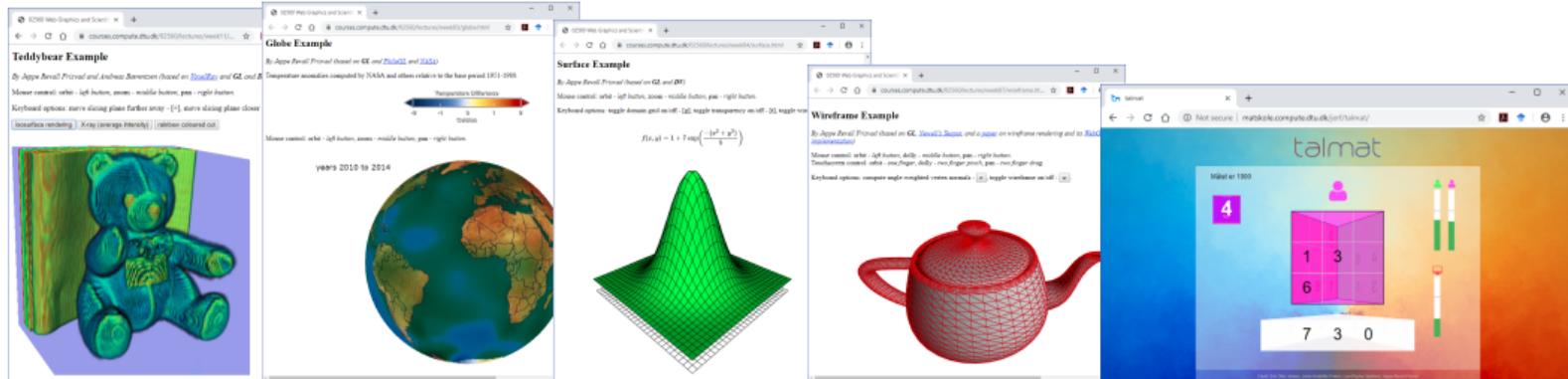
# Advanced techniques (weeks 11–13)

- ▶ 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: Minh Tran
- ▶ TA: to be announced
- ▶ Lectures and exercises interspersed (8:00-10:00)
- ▶ Exercises (10:00-12:00)
- ▶ 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: **17 December 2023 at 23:59**

# 02561 Computer Graphics

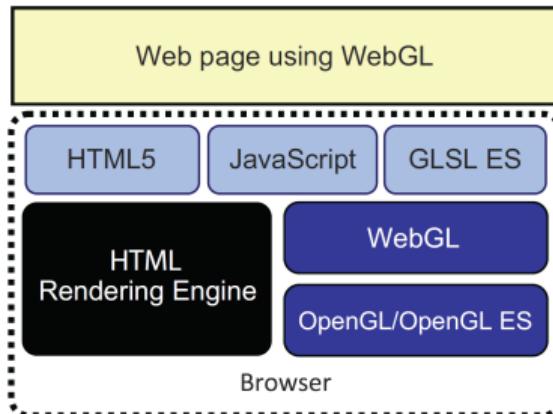
## WebGL and Graphics Pipeline

Jeppe Revall Frisvad

August 2023

# 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)



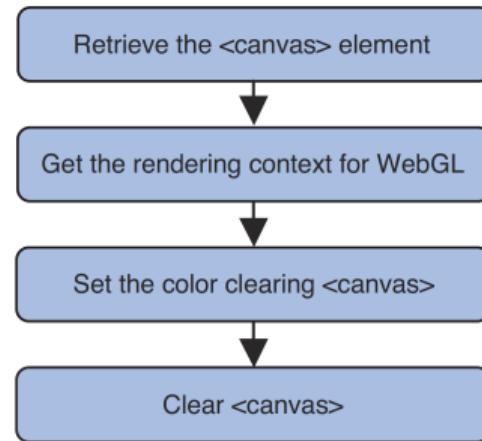
## Exercise: getting started (W01P1)

- ▶ 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>
```

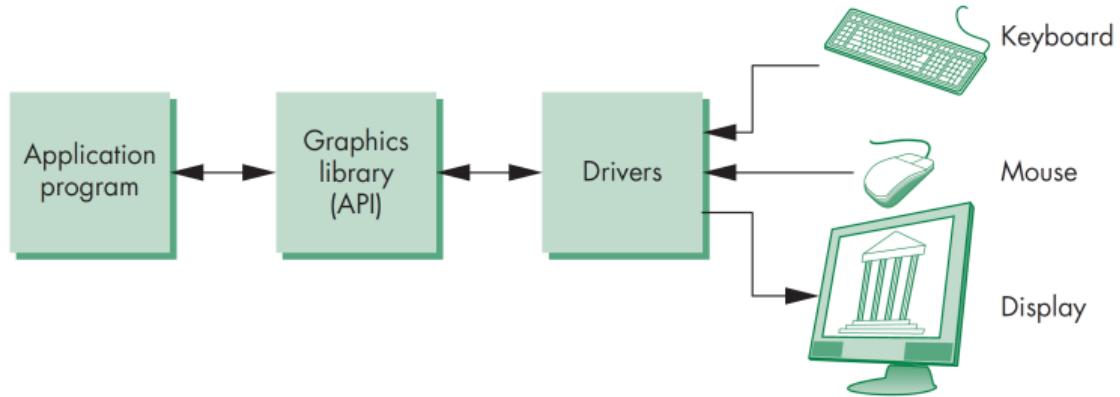
- ▶ Get the library files from angelCommon.zip on DTU Learn.
- ▶ Use **A**: 2.8 to set canvas size (this is the resolution of the rendered image) and an init function (in filename.js) and solve the first part of the first worksheet.

```
window.onload = function init()
{
    // My JavaScript code which runs when the webpage is loaded by the browser
}
```

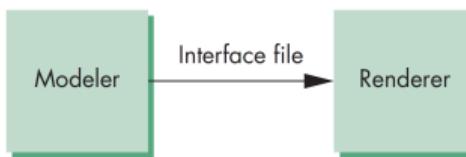


# Interactive computer graphics

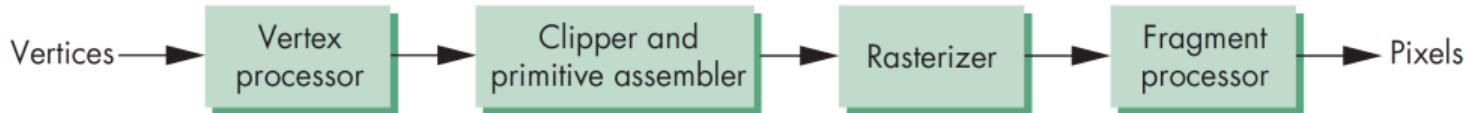
- ▶ Graphics system:



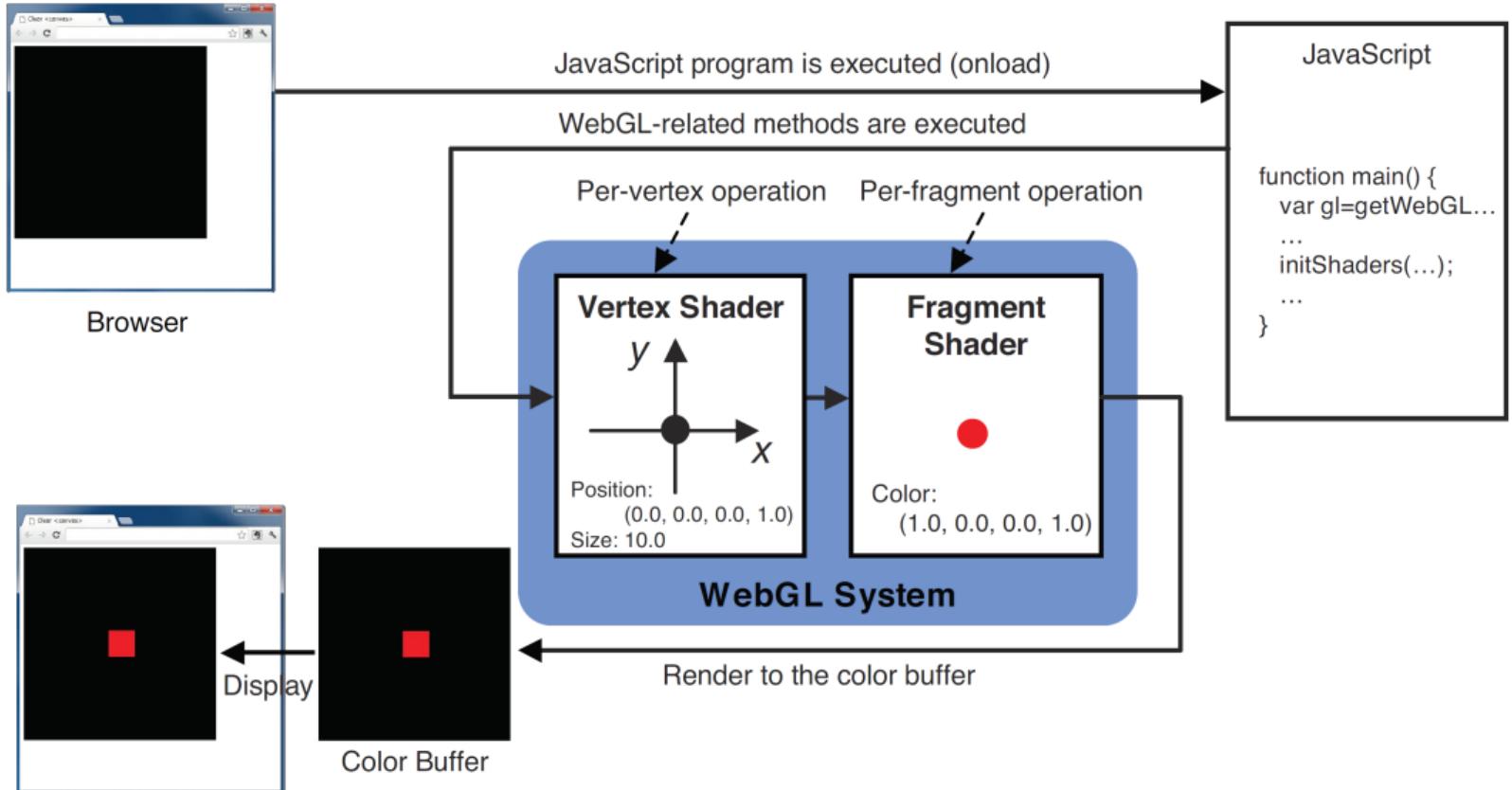
- ▶ Modeling-rendering paradigm:



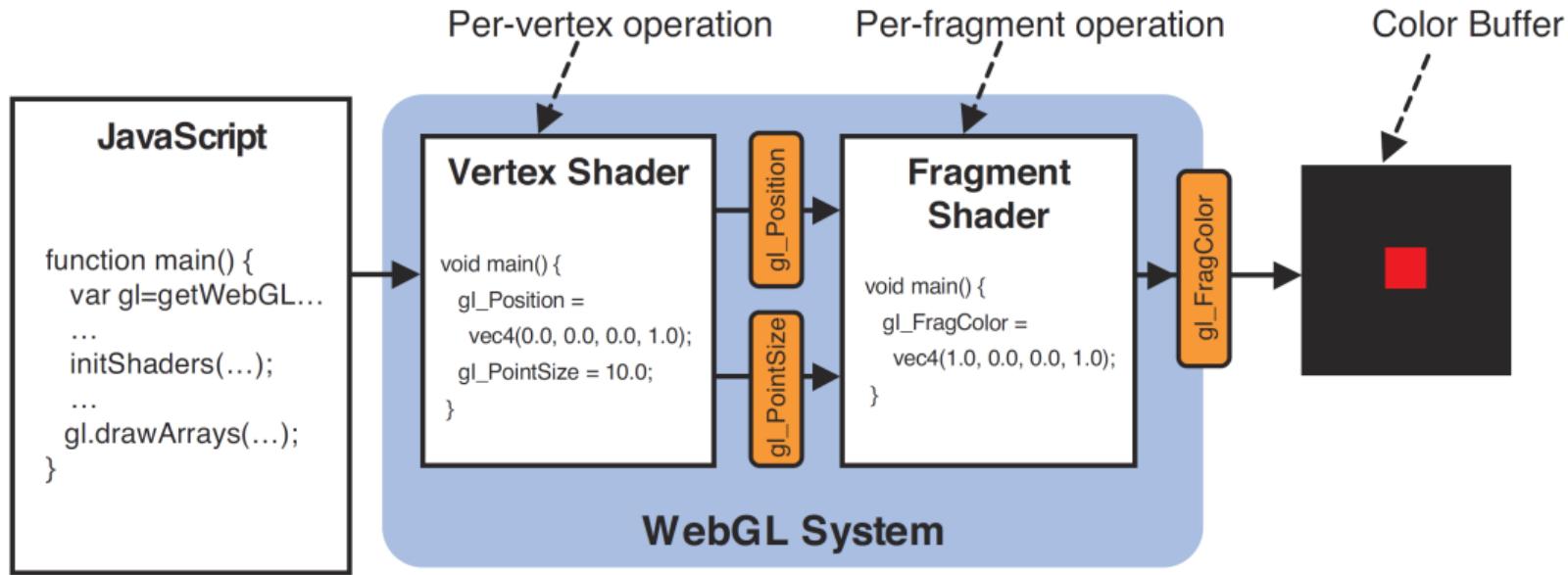
- ▶ Rasterization pipeline:



# Shaders and processing flow



# WebGL shaders in practice



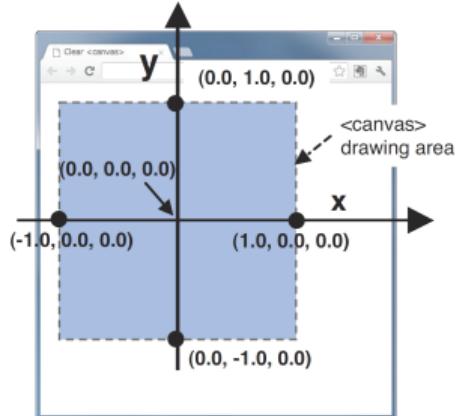
► A: 2.8

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

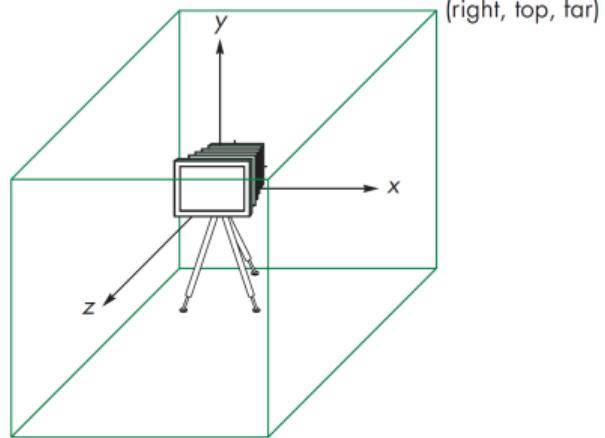
```
<script id="vertex-shader" type="x-shader/x-vertex">
    void main() {
        gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
        gl_PointSize = 10.0;
    }
</script>
```

```
<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>
```

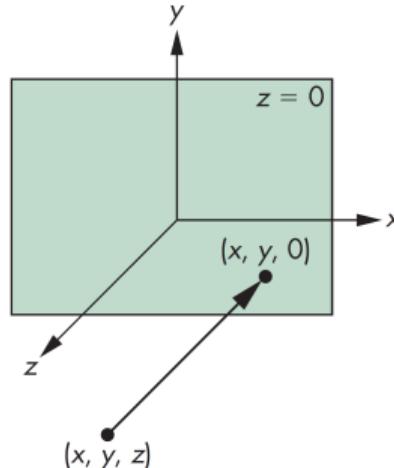
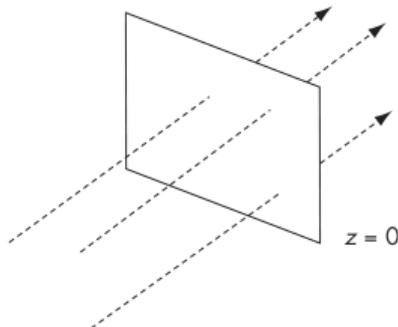
# Drawing in 2D



(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>
```

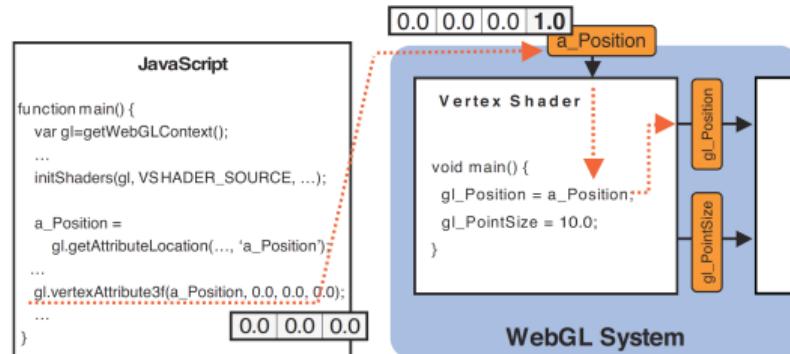
Storage Qualifier      Type      Variable Name  
attribute vec4 a\_Position;

- ▶ In JavaScript init/main function:

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

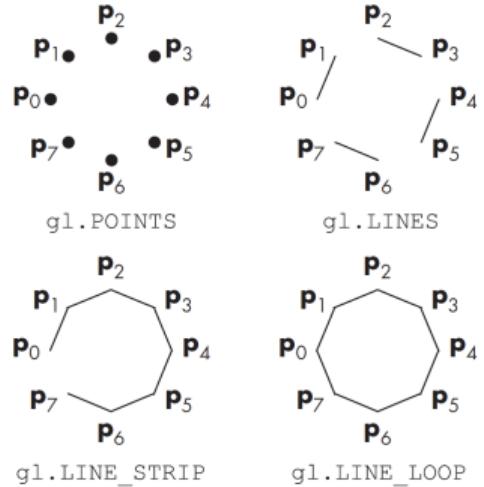
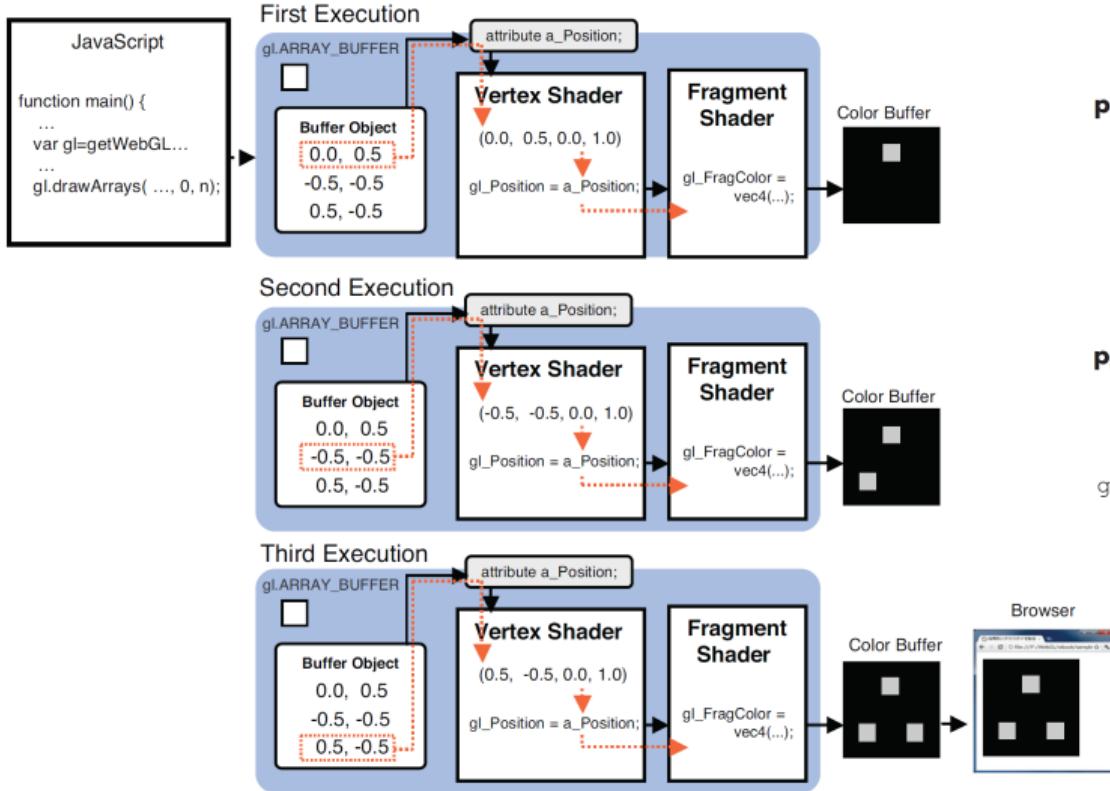
var vertices = [ vec2(0.0, 0.5), vec2(-0.5, -0.5), vec2(0.5, -0.5) ];
var vBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);

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



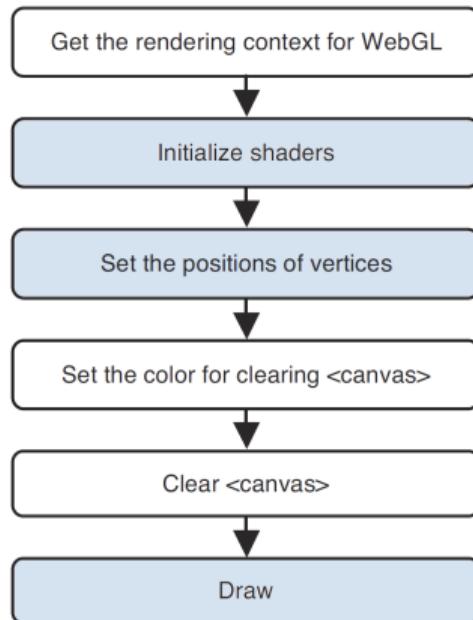
# Start drawing

- 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).

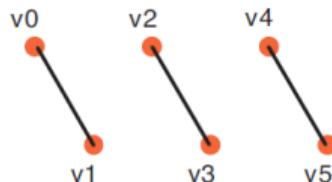


## Basic shapes

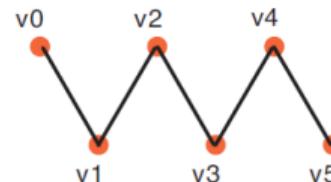
- Another draw call example: `gl.drawArrays(gl.TRIANGLES, 0, numVertices);`



`gl.POINTS`



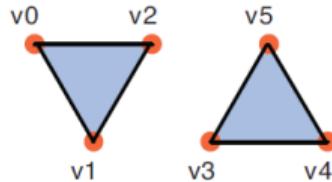
`gl.LINES`



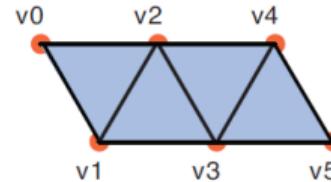
`gl.LINE_STRIP`



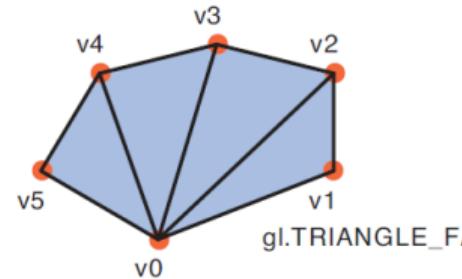
`gl.LINE_LOOP`



`gl.TRIANGLES`



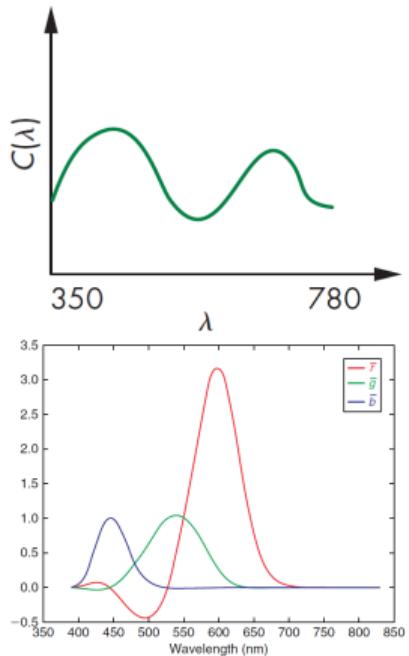
`gl.TRIANGLE_STRIP`



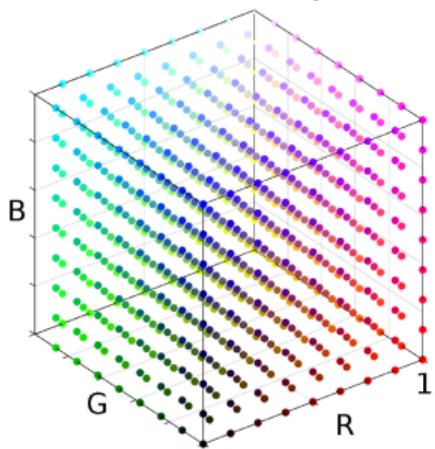
`gl.TRIANGLE_FAN`

# Colors

spectrum

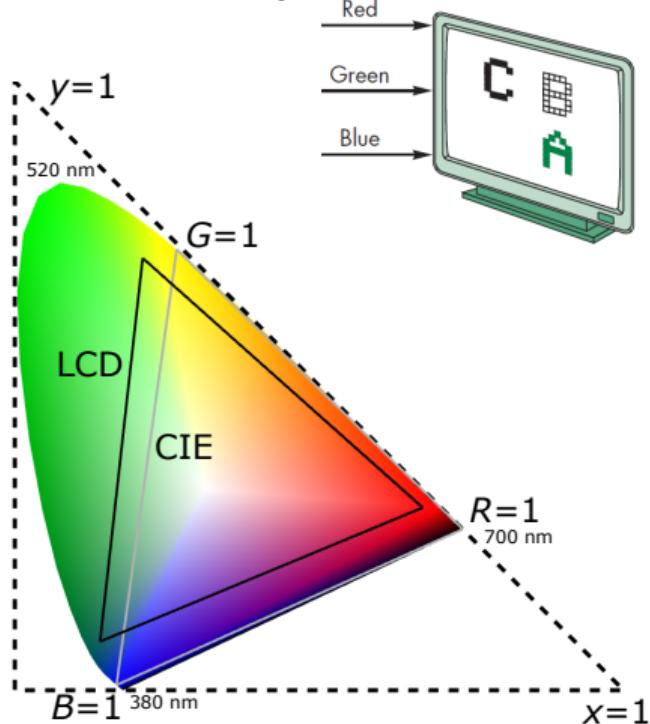


RGB color space



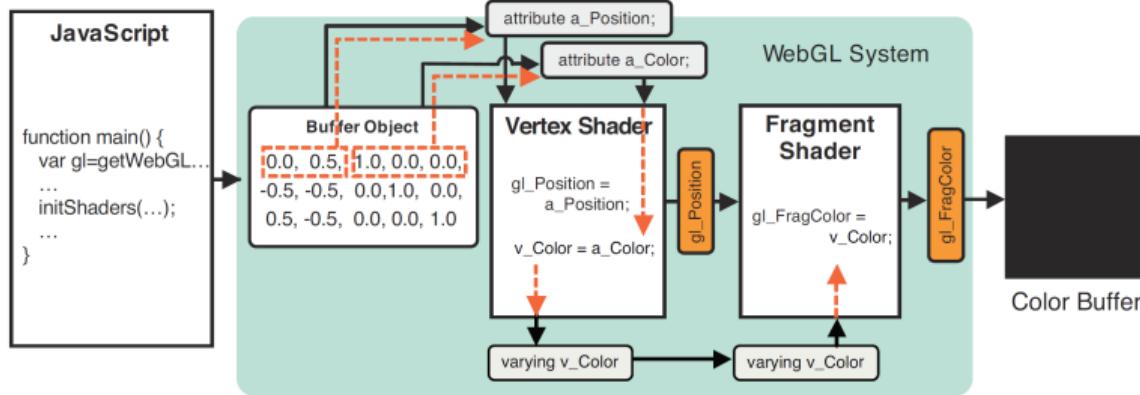
$$R = \int_{\gamma} C(\lambda) \bar{r}(\lambda) d\lambda$$
$$G = \int_{\gamma} C(\lambda) \bar{g}(\lambda) d\lambda$$
$$B = \int_{\gamma} C(\lambda) \bar{b}(\lambda) d\lambda$$

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.

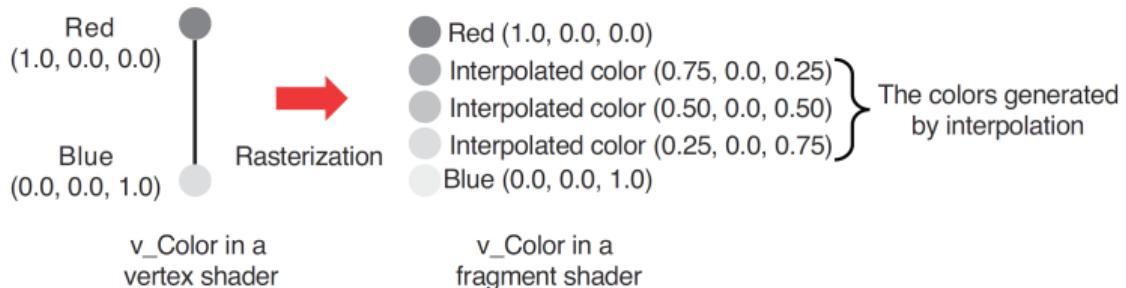
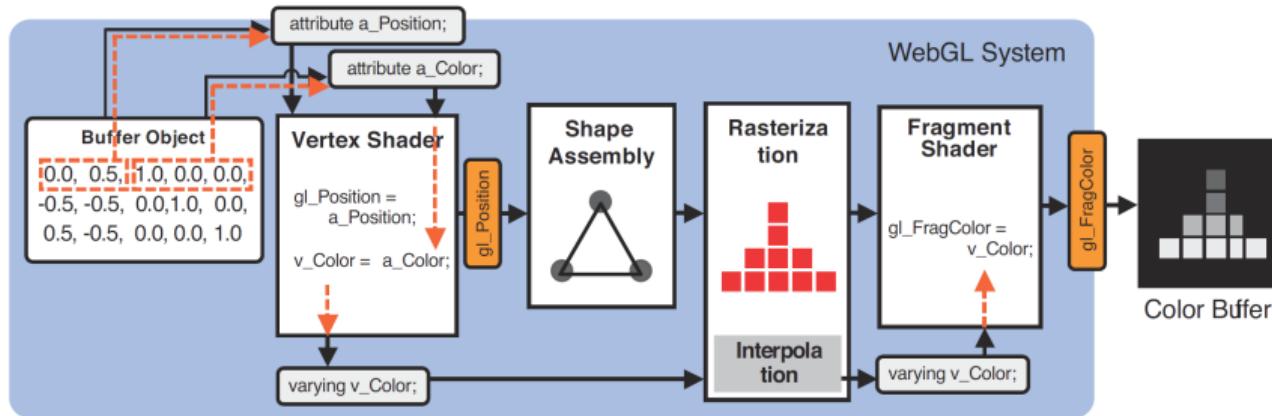
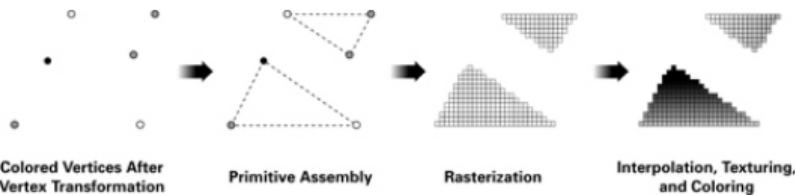


## ▶ A: 2.10

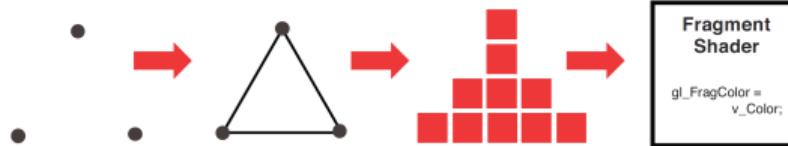
```
<script id="vertex-shader" type="x-shader/x-vertex">  
attribute vec4 a_Position;  
attribute vec4 a_Color;  
varying vec4 v_Color;  
void main() {  
    v_Color = a_Color;  
    gl_Position = a_Position;  
}  
</script>
```

```
<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)

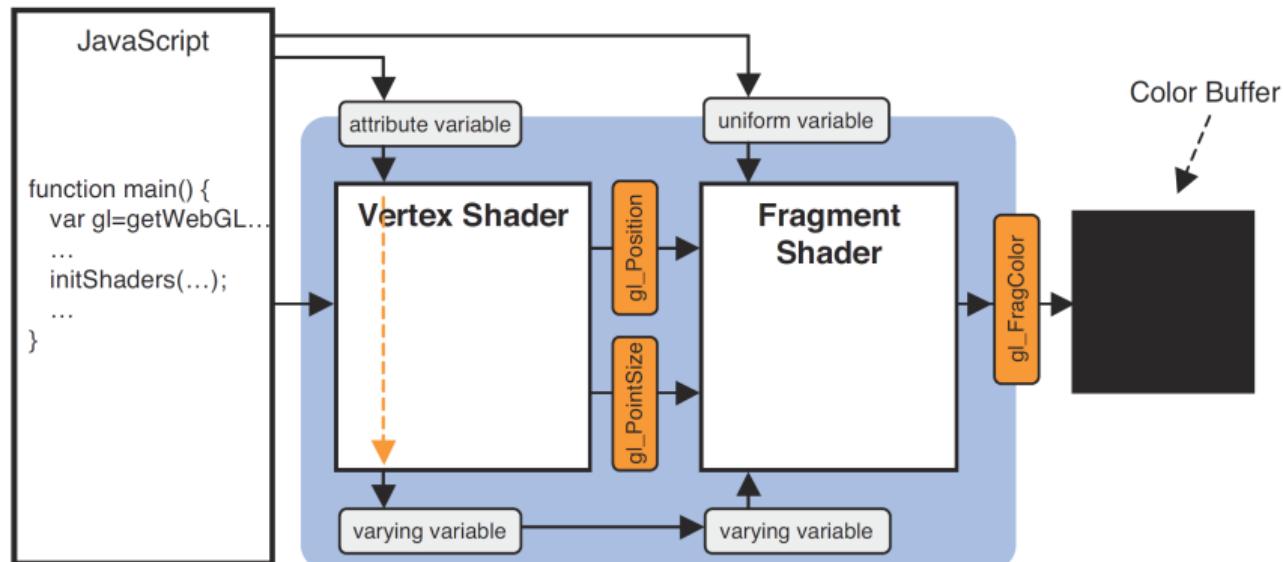


- ▶ Copy and extend your solution for Part 2.
- ▶ Update the vertex shader to include a color attribute that is passed to the fragment shader as a varying variable.
- ▶ Define colors (array of vectors).
- ▶ Connect color attribute variable to buffer:
  - ▶ Create a buffer.
  - ▶ Bind the buffer object 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.

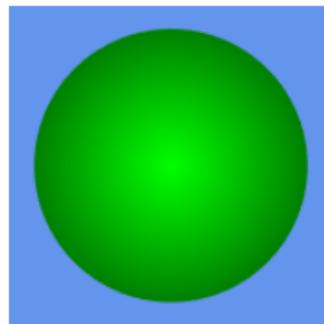
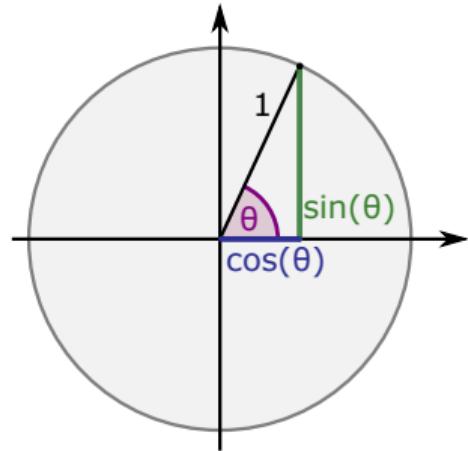
Storage Qualifier      Type      Variable Name  
**uniform vec4 u\_FragColor;**



```
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  $\theta = 2\pi i / n$  ( $\pi$  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.



# The WebGPU version (new API expected to supersede WebGL)

- ▶ The application consists of an HTML file and a JS file (just like WebGL)

```
<!DOCTYPE html>
<html>
<head>
    <title>W1P1</title>
    <script type="text/javascript" src="w1p1.js"></script>
</head>
<body>
    <canvas id="webgpu-canvas" width="512" height="512">
        Please use a browser that supports HTML5 canvas.
    </canvas>
</body>
</html>
```

make a canvas

- ▶ The background JavaScript needs an asynchronous main function:

```
"use strict";
window.onload = function () { main(); }
async function main()
{
    const gpu = navigator.gpu;
    const adapter = await gpu.requestAdapter();
    const device = await adapter.requestDevice();
    const canvas = document.getElementById("webgpu-canvas");
    const context = canvas.getContext("gpupresent") || canvas.getContext("webgpu");
    const canvasFormat = navigator.gpu.getPreferredCanvasFormat();
    context.configure({
        device: device,
        format: canvasFormat,
    });

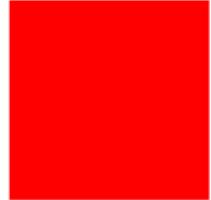
    // Create a render pass in a command buffer and submit it
    :
}
```

make a rendering context

# Clearing the canvas (W01P1)

- ▶ The render pass command buffer:

```
// Create a render pass in a command buffer and submit it
const encoder = device.createCommandEncoder();
const pass = encoder.beginRenderPass({
    colorAttachments: [{{
        view: context.getCurrentTexture().createView(),
        loadOp: "clear",
        clearValue: { r: 1.0, g: 0.0, b: 0.0, a: 1.0 },
        storeOp: "store",
    }}];
});
// Insert render pass commands here
pass.end();
device.queue.submit([encoder.finish()]);
```



- ▶ Writing WGSL shaders in the HTML file:

```
<script id="wgsl" type="x-shader">
@vertex
fn main_vs(@location(0) pos: vec2f) -> @builtin(position) vec4f
{
    return vec4f(pos, 0, 1);
}

@fragment
fn main_fs() -> @location(0) vec4f
{
    return vec4f(0.0, 0.0, 0.0, 1.0);
}
</script>
```

# Setting up a render pipeline

- ▶ Load the WGSL shaders from the HTML file and use them for the render pipeline:

```
const wgpu = device.createShaderModule({
    code: document.getElementById("wgsl").text
});

const pipeline = device.createRenderPipeline({
    layout: "auto",
    vertex: {
        module: wgpu,
        entryPoint: "main_vs",
        buffers: [vertexBufferLayout]
    },
    fragment: {
        module: wgpu,
        entryPoint: "main_fs",
        targets: [{ format: canvasFormat }]
    },
    primitive: {
        topology: "triangle-list",
        // GPUPrimitiveTopology { "point-list", "line-list", "line-strip", "triangle-list", "triangle-strip" };
    },
});
```

- ▶ Add the pipeline and a draw call to the render pass:

```
pass.setPipeline(pipeline);
pass.setVertexBuffer(0, vertexBuffer);
pass.draw(vertices.length);
```

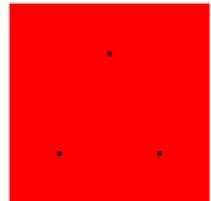
# Setting up a vertex buffer and drawing points (W01P2)

- ▶ Creating a buffer and connecting it to location 0 of the vertex shader:

```
const point_size = 10*(2/canvas.height);
var vertices = [];
add_point(vertices, vec2( 0.0,  0.5), point_size);
add_point(vertices, vec2(-0.5, -0.5), point_size);
add_point(vertices, vec2( 0.5, -0.5), point_size);

const vertexBuffer = device.createBuffer({
    size: flatten(vertices).byteLength,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(vertexBuffer, /*bufferOffset= */0, flatten(vertices));

const vertexBufferLayout = {
    arrayStride: sizeof["vec2"],
    attributes: [
        {
            format: "float32x2",
            offset: 0,
            shaderLocation: 0, // Position, see vertex shader
        },
    ],
};
```



- ▶ Point size is one and immutable in WebGPU. Use triangles to draw points:

```
function add_point(array, point, size)
{
    const offset = size/2;
    var point_coords = [ vec2(point[0] - offset, point[1] - offset), vec2(point[0] + offset, point[1] - offset),
                        vec2(point[0] - offset, point[1] + offset), vec2(point[0] - offset, point[1] + offset),
                        vec2(point[0] + offset, point[1] - offset), vec2(point[0] + offset, point[1] + offset) ];
    array.push.apply(array, point_coords);
}
```

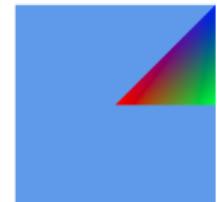
# Passing information from the vertex to the fragment shader (W01P3)

- ▶ Setting up a varying variable in WebGPU:

```
<script id="wgsl" type="x-shader">
    struct VSOut {
        @builtin(position) position: vec4f,
        @location(0)          color   : vec3f,
    };

    @vertex
    fn main_vs(@location(0) inPos: vec2f,
               @location(1) inColor: vec3f) -> VSOut
    {
        var vsOut: VSOut;
        vsOut.position = vec4f(inPos, 0.0, 1.0);
        vsOut.color   = inColor;
        return vsOut;
    }

    @fragment
    fn main_fs(@location(0) inColor: vec3f) -> @location(0) vec4f
    {
        return vec4f(inColor, 1.0);
    }
</script>
```



- ▶ A vertex color buffer is set up in the same way as a vertex position buffer.
- ▶ The color buffer layout needs to be added to the buffers array of the pipeline.
- ▶ The vertex color buffer needs to be added to the render pass at location 1.

## Setting up uniform variables

- ▶ Data that are the same for all vertices are called Uniforms.
- ▶ We can define them at the top of our WGSL code:

```
struct Uniforms {  
    theta: f32,  
};  
@group(0) @binding(0) var<uniform> uniforms : Uniforms;
```

- ▶ and use them like other structs:

```
let sin_theta = sin(uniforms.theta);
```

- ▶ But they require some infrastructure on the JavaScript side:

```
var theta = new Float32Array([0.0]);  
const uniformBuffer = device.createBuffer({  
    size: theta.byteLength,  
    usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,  
});  
const bindGroup = device.createBindGroup({  
    layout: pipeline.getBindGroupLayout(0),  
    entries: [{  
        binding: 0,  
        resource: { buffer: uniformBuffer }  
    }],  
});  
device.queue.writeBuffer(uniformBuffer, 0, theta);
```

- ▶ and a command needs to be added to the render pass before the draw call:  

```
pass.setBindGroup(0, bindGroup);
```