Computer Graphics

Minho Kim

Dept. of Computer Science

University of Seoul

Chapter 2: Your First Step with WebGL

What to Learn

- How WebGL uses the `<canvas>` element and how to draw on it
- The linkage between HTML and WebGL using JavaScript
- Simple WebGL drawing functions
- The role of shader programs within WebGL

- Examples can be found at https://sites.google.com/site/webglbook/home/chapter-2

What Is a Canvas?

- Pre-HTML5
  - Static images only using `<img>` tag → plug-ins are used such as Flash Player
- HTML5
  - Dynamic graphics in `<canvas>` using JavaScript
  - Points, lines, rectangles, circles, etc.
  - Drawing apps: Canvas Painter, Sketch Toy, Muro, SketchPad, and more

Example #1: `DrawRectangle`

Example #1: `DrawRectangle`

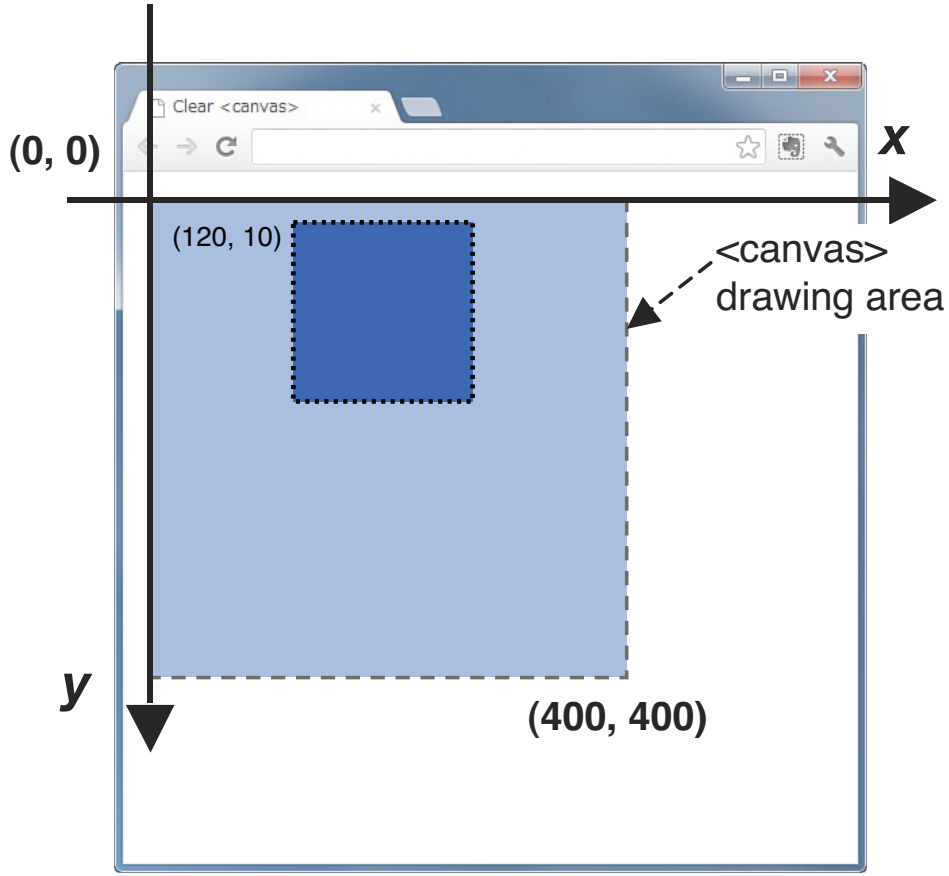
- Drawing in `<canvas>` using JavaScript
- 2D drawing (not using WebGL)
- http://www.minho-kim.com/courses/17fa71033/data/DrawRectangle.html
- Procedure for 2D drawing
  1) Retrieve the `<canvas>` element.
  2) Request the rendering “context” for the 2D graphics from the element.
  3) Draw the 2D graphics using the methods that the context supports.

Procedure for 2D Drawing

1) Retrieve the `<canvas>` element.
   - Using the method `document.getElementById()`
   - `null` returned if failed
   - Uses `console.log()` to display an error message.
     → Needs to turn on the console window
2) Request the rendering "context" for the 2D graphics from the element.
   - `canvas.getContext('2d')`
   - No error checking in this example

Procedure for 2D Drawing (cont'd)

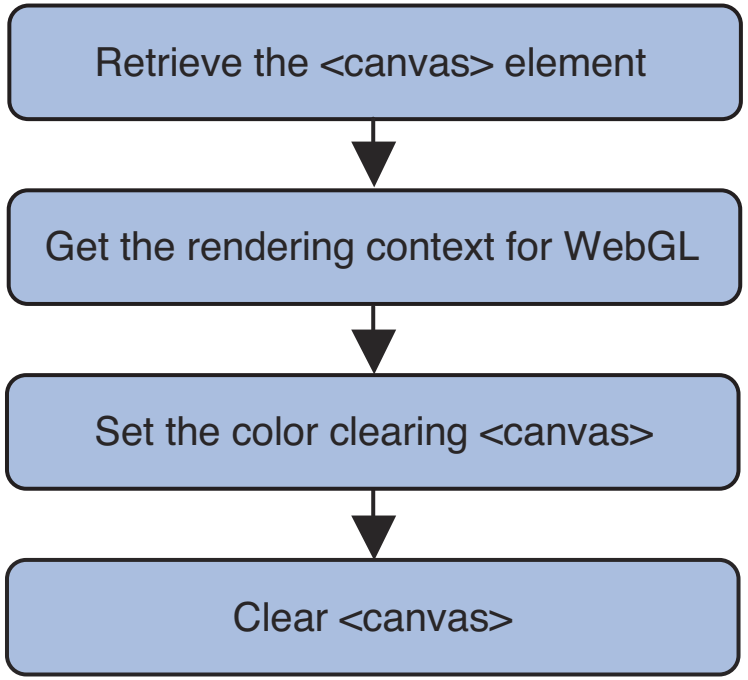
3) Draw the 2D graphics using the methods that the context supports.
   - RGBA color format
     - Each color channel as an unsigned byte (0~255)
     - alpha (transparency) as a float in [0,1]
   - Coordinate system
     - origin at the top-left corner



Example #2: `HelloCanvas`

Example #2: `HelloCanvas`

- The world's shortest WebGL program
  - Nothing is drawn, No shader used
- http://rodger.global-linguist.com/webgl/ch02/HelloCanvas.html
  - Without helper library: https://xregy.github.io/webgl/minimal.html
- Uses helper JavaScript library files
  - `webgl-utils.js` (by Google)
  - `webgl-debug.js` (by Ken Rusell)
  - `cuon-utils.js`, `cuon-matrix.js` (by the authors)



Step 1: Retrieve the `<canvas>` Element

- Using `document.getElementById()` with the id of the `<canvas>` element

Step 2: Get the Rendering Context for WebGL

- Using `HTMLCanvasElement.getContext()`
- The argument varies between browsers
  - "`experimental-webgl`" or "`webgl`"
  - `WebGLUtils.setupWebGL()` in `webgl-utils.js` handles it for us. (called by `getWebGLContext()` in `cuon-utils.js`)
- Stored in the variable "`gl`"

`getWebGLContext(element [, debug])`

Get the rendering context for WebGL, set the debug setting for WebGL, and display any error message in the browser console in case of error.

Parameters	element	Specifies `<canvas>` element to be queried.
	debug (optional)	Default is `true`. When set to `true`, JavaScript errors are displayed in the console. Note: Turn off after debugging; otherwise, performance is affected.
Return value	non-null	The rendering context for WebGL.
	null	WebGL is not available.

Step 3: Set the Color for Clearing the `<canvas>`

- Using `gl.clearColor()`
- Each color channel specified as a float in [0,1]
- Retained in the WebGL system and not changed until specified again.

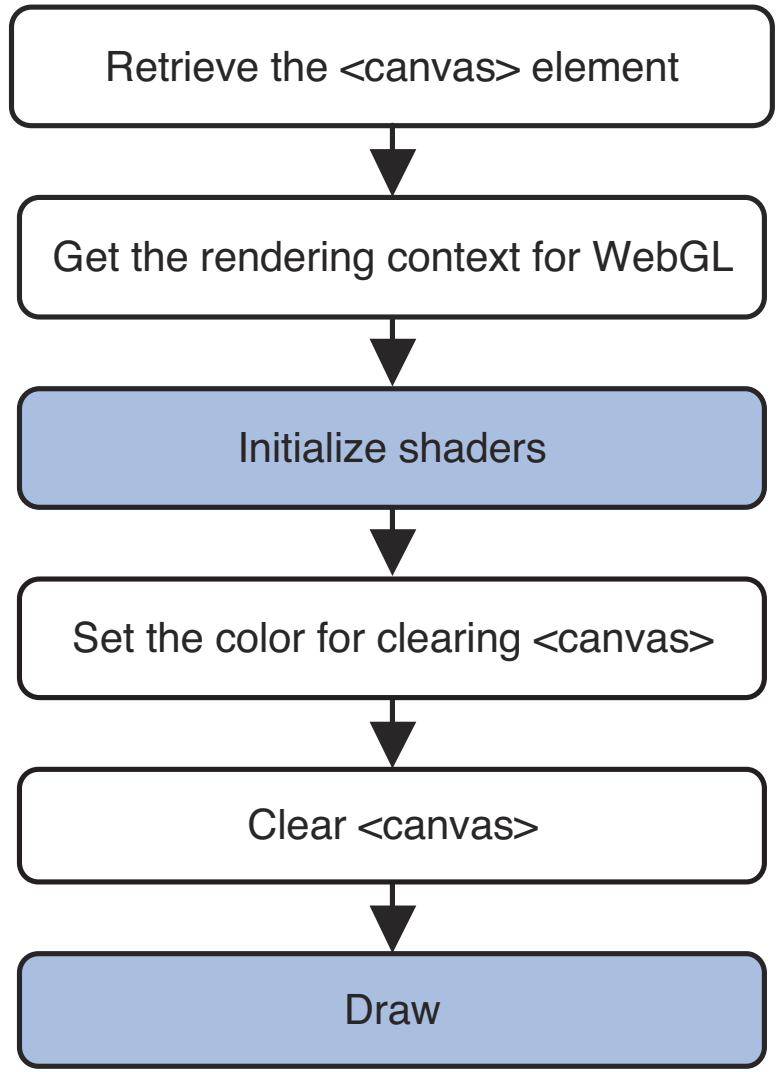
Step 4: Clear `<canvas>`

- Using `gl.clear()`
- Multiple buffers can be cleared (color/depth/stencil)
  - Specified by bitwise OR → (potential) performance improvement by parallel execution
  - Default values are used if not specified

Example #3: `HelloPoint1`

Example #3: `HelloPoint1`

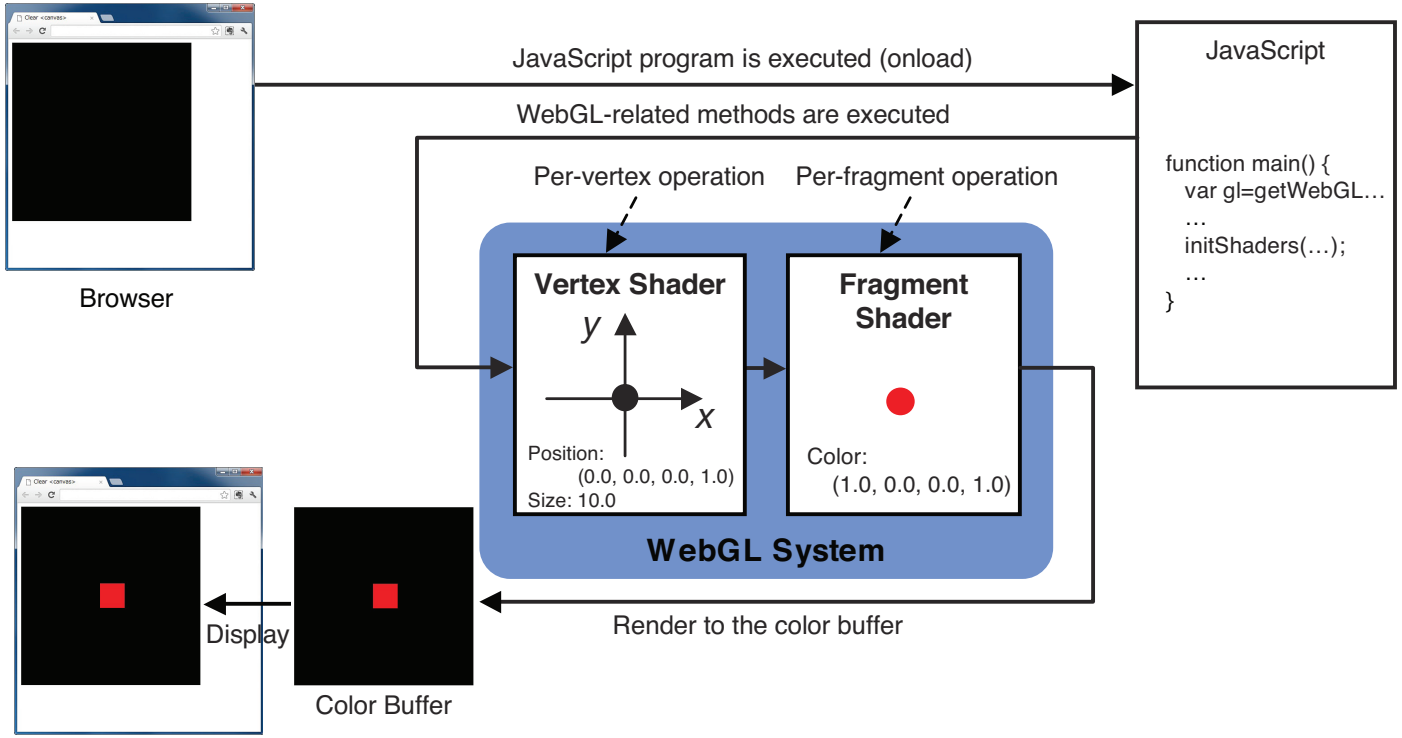
- http://rodger.global-linguist.com/webgl/ch02/HelloPoint1.html
- Drawing a (rectangular) point at the center
- Not using any “buffer object”
  - No data pass between JS & shaders – hard-coded
  - Not a practical example (We have to use buffer objects!)
- No animation – No re-drawing, drawing only once
- What to learn
  - How to specify the color of the point?
  - How to specify the position (coordinates) of the point?
  - How to implement shader programs?

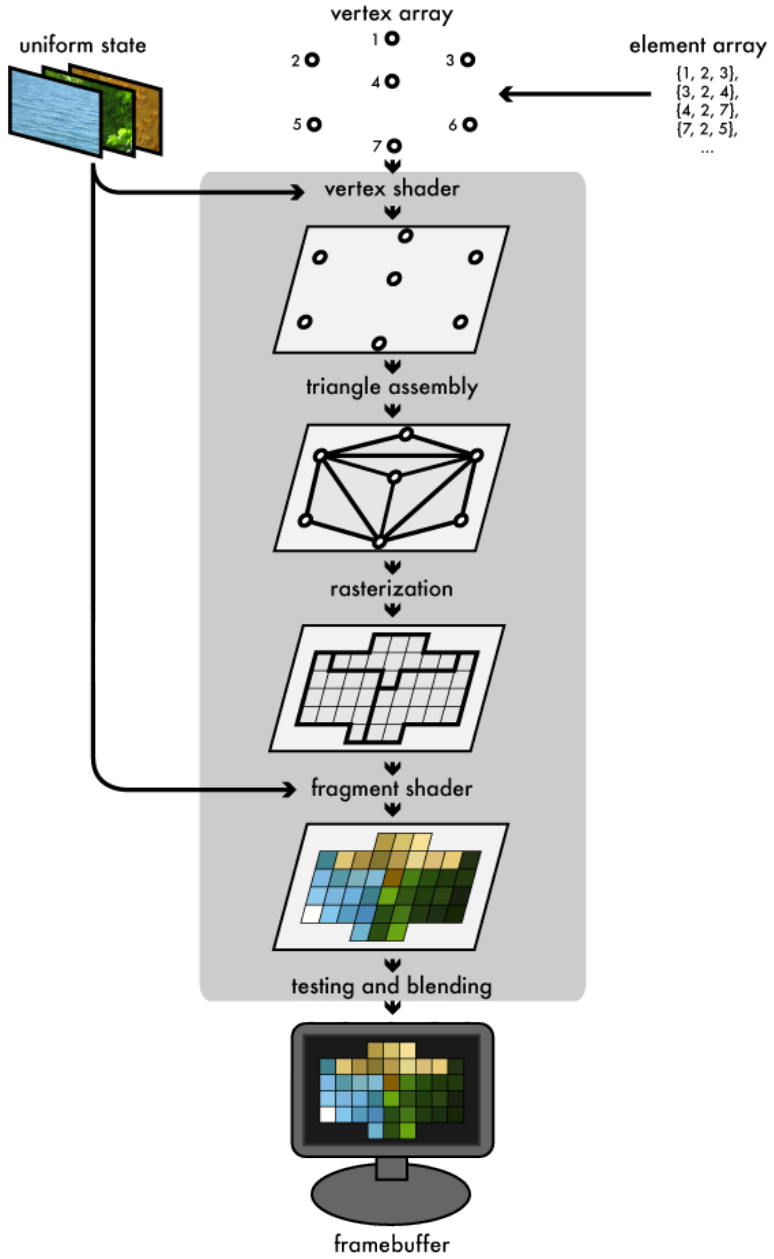


What Is a Shader?

- A small program running on GPU cores in parallel
- WebGL app = JS (exec'ed by browser) + shaders (exec'ed by WebGL system)
- Two types
  - Vertex shader
    - Describes the traits (position, colors, etc.) of a vertex
    - Per-vertex operations – transformation, etc.
    - Vertices – points composing primitives, such as points, lines and triangles
  - Fragment shader
    - Deals with per-fragment processing
    - Per-fragment operations – lighting, etc.
    - Fragments – potential pixels composed of various values
- Written in GLSL (OpenGL Shading Language) ES

WebGL Rendering Pipeline





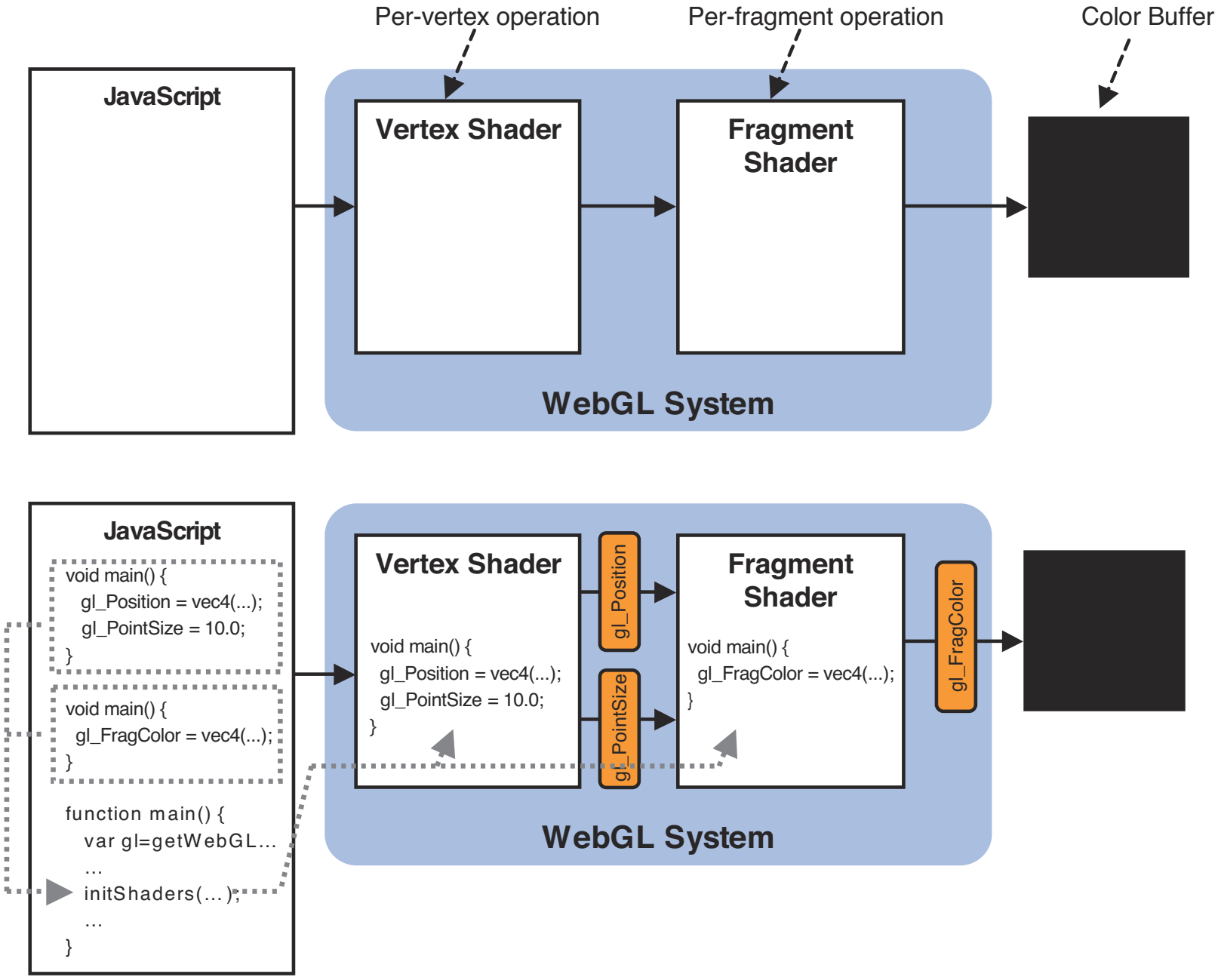
(Joe Groff)

The Structure of a WebGL Program that Uses Shaders

- Consists of three parts
  - Vertex shader program (written in GLSL ES)
  - Fragment shader program (written in GLSL ES)
  - Main program (written in JavaScript)
- GLSL ES
  - C-like language
  - Passed to (as a string) and compiled in run-time by the WebGL system
- Where to store shader source codes
  - As JavaScript strings – easy to program, hard to manage
  - Embedded in HTML tag https://xregy.github.io/webgl/shaders_in_html.html
  - In separate files and loaded in run-time using (Appendix F) – tricky, not working locally
  - Embedded in server side (e.g. Server Side Includes)

Initializing Shaders

- Procedure
  - Create a program object (`gl.createProgram()`)
  - For each shader type (vertex, fragment)
    - Create a shader object (`gl.createShader()`)
    - Pass the source code as a string (`gl.shaderSource())`
    - Compile the shader (`gl.compileShader()`)
    - Attach the shader (`gl.attachShader())`
  - Link the program (`gl.linkProgram()`)
- The whole procedure is wrapped in `initShaders()` in `cuon-utils.js`, including error checking.
- More in Chapter 8


Per-vertex operation
Per-fragment operation
Color Buffer
JavaScript
Vertex Shader
Fragment Shader
WebGL System
JavaScript
void main() {
gl_Position = vec4(...);
gl_PointSize = 10.0;
}
void main() {
gl_FragColor = vec4(...);
}
function main() {
var gl=getWebGL...
...
initShaders(...);
...
}
Vertex Shader
void main() {
gl_Position = vec4(...);
gl_PointSize = 10.0;
}
gl_Position
gl_PointSize
Fragment Shader
void main() {
gl_FragColor = vec4(...);
}
gl_FragColor
WebGL System

Vertex Shader

```
void main() {
  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
  gl_PointSize = 10.0;
}
```

- Per-vertex operation
- Here, it specifies the position of a point and its size
- Built-in variables
  - `vec4 gl_Position` – should always be written
  - `float gl_PointSize`
- “Typed” language
- `vec4`
  - Specifies coordinates, color, etc.
  - 3D coordinates specified in homogeneous coordinates system
  - The 4th component has default 1.0

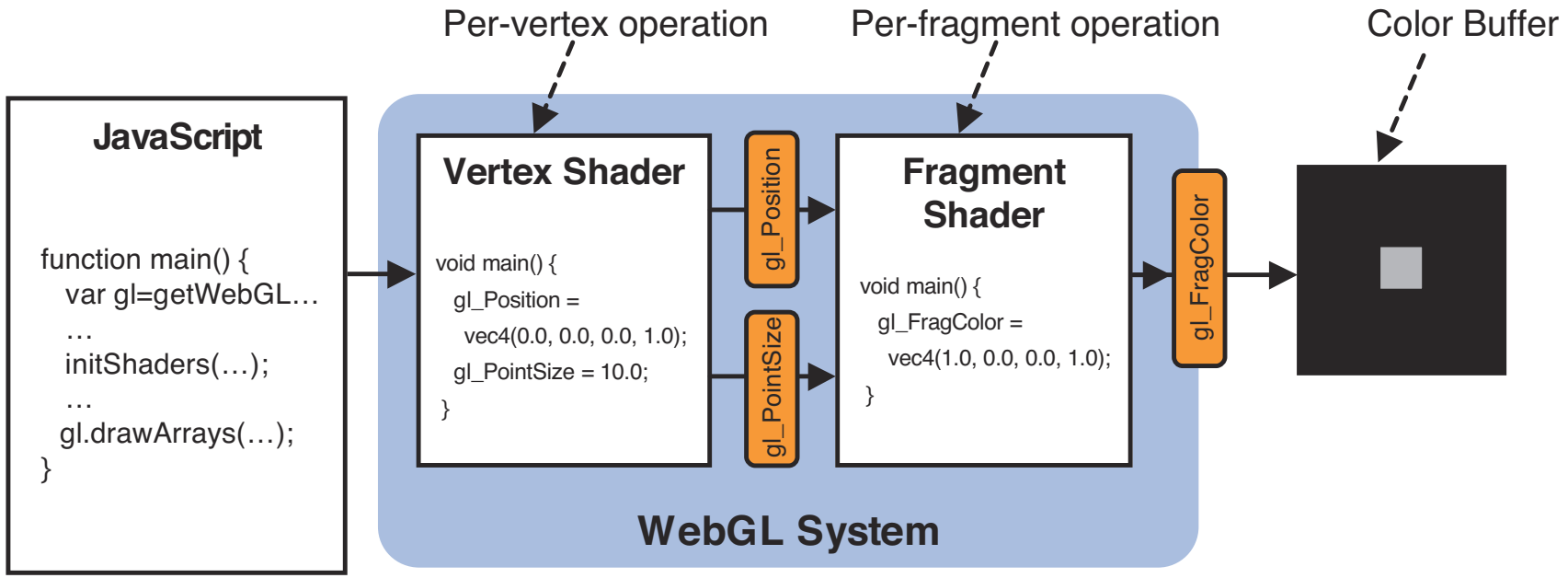
Fragment Shader

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

- Per-fragment operation
  - Fragment – A (potential) pixel with a collection of values
- Here, it specifies the color of the point
- `gl_FragColor`
  - Built-in variable specifying the fragment color
  - Available only in a fragment shader
  - Removed in WebGL 2.0

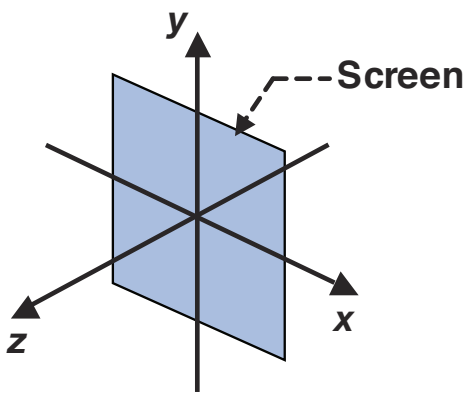
The Draw Operation

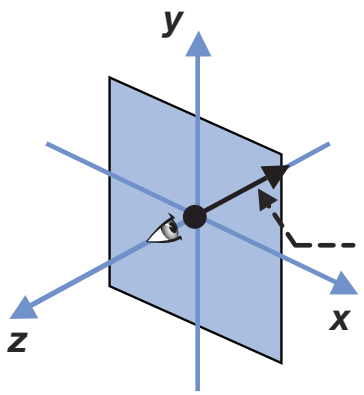
- `gl.drawArrays()`
- `gl.drawElements()`
- More draw functions in WebGL 2.0

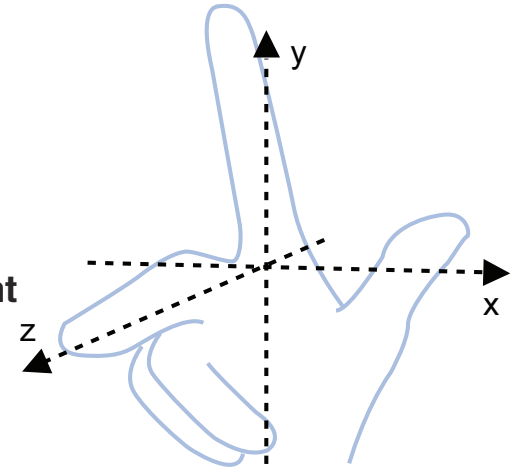


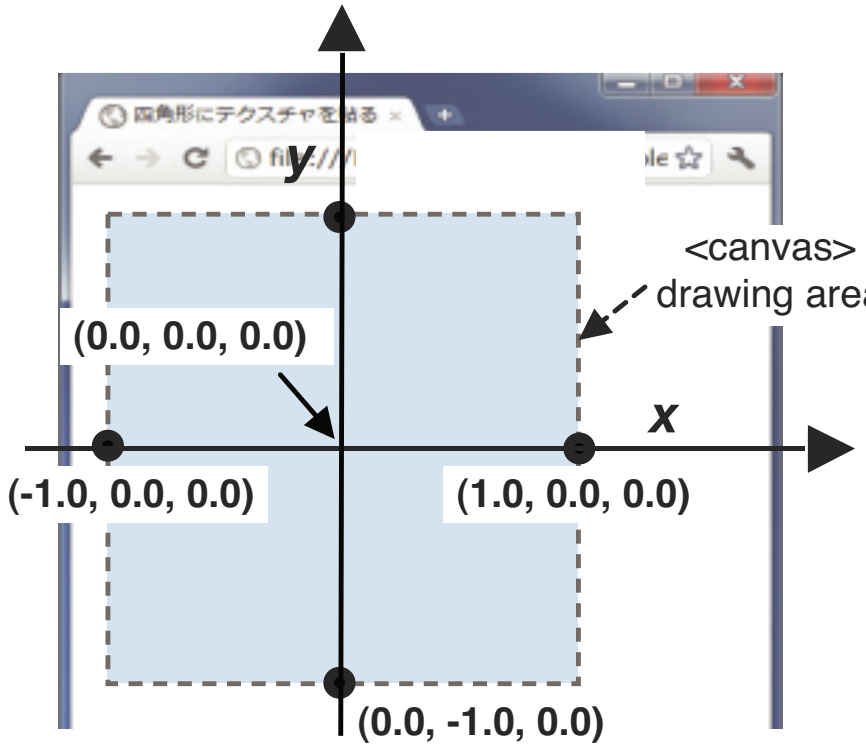
The WebGL Coordinate System

- Right-handed coordinate system
  - Convention – WebGL doesn't force any coordinate system (Appendix D)
- By default, the <canvas> area is mapped to $[-1,1]^3$









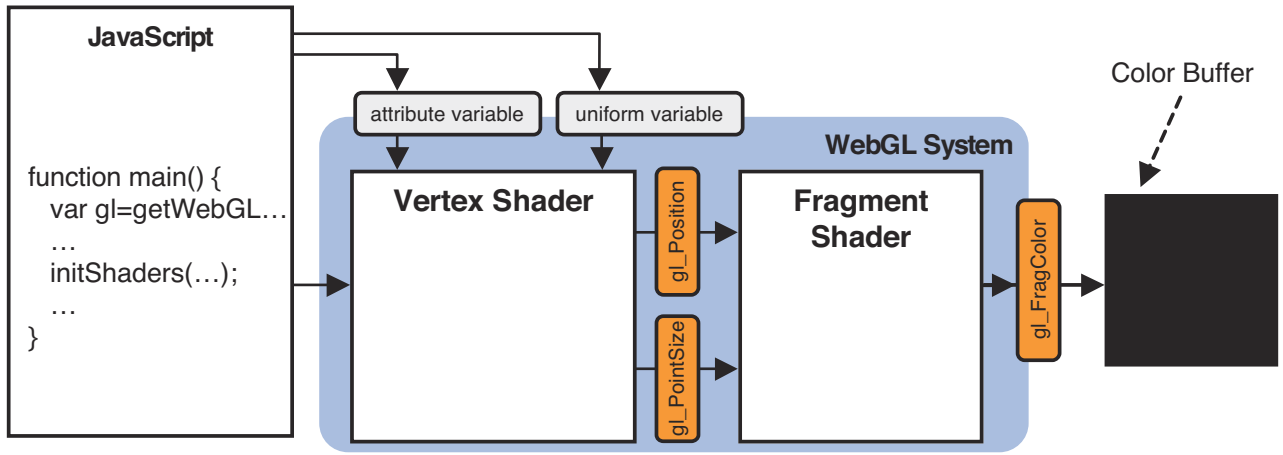
Example #4: `HelloPoint2`

Example #4: `HelloPoint2`

- http://rodger.global-linguist.com/webgl/ch02/HelloPoint2.html
- What to learn
  - How to pass data (point position) between JS & shaders using **vertex attributes** → flexibility

Using Attribute Variables

- Two ways to pass data to a vertex shader
  - attribute variables – per-vertex. position, texcoords, etc.
  - uniform variables – same (uniform) in each vertex
- Procedure to use attribute variables
  1. Prepare the attribute variable for the vertex position in the vertex shader.
  2. Assign the attribute variable to the `gl_Position` variable.
  3. Pass the data to the attribute variable.



HelloPoint2.js

1. Prepare the attribute variable for the vertex position in the vertex shader.

```
attribute vec4 a_Position;
```

   - `attribute`: storage qualifier
2. Assign the attribute variable to the `gl_Position` variable.

```
gl_Position = a_Position;
```

3. Pass the data to the attribute variable.
   1. Get the storage location of an attribute variable.
   2. Assign a value to an attribute variable.

Getting the Storage Location of an Attribute Variable

- Locations of attribute variables are determined by the WebGL system when compiling & linking the shader programs. → should be called after linking

```
var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
if (a_Position < 0) {
  console.log('Fail to get the storage location of a_Position');
  return;
}
```

- JavaScript (not GLSL) – Don't' confuse `a_Position` with the one in the GLSL code!
- `gl.program` holds the program object (not originally in `WebGLRenderingContext`, but assigned dynamically by `initShaders()`.)

Assigning a Value to an Attribute Variable

- Assigns a “static” vertex attribute

```
gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);
```

  - OpenGL function naming rules
- Not practical. We won’t assign values in this way…
- In practice, we will use vertex attribute array enabled by `gl.enableVertexAttribArray()`.
- `vertexAttrib3f` for `vec4`? → Default values assigned (0,0,0,1)

Try Yourself

- Try to set the point size using an attribute variable.

Example #5: `ClickedPoint`

Example #5: `ClickedPoint`

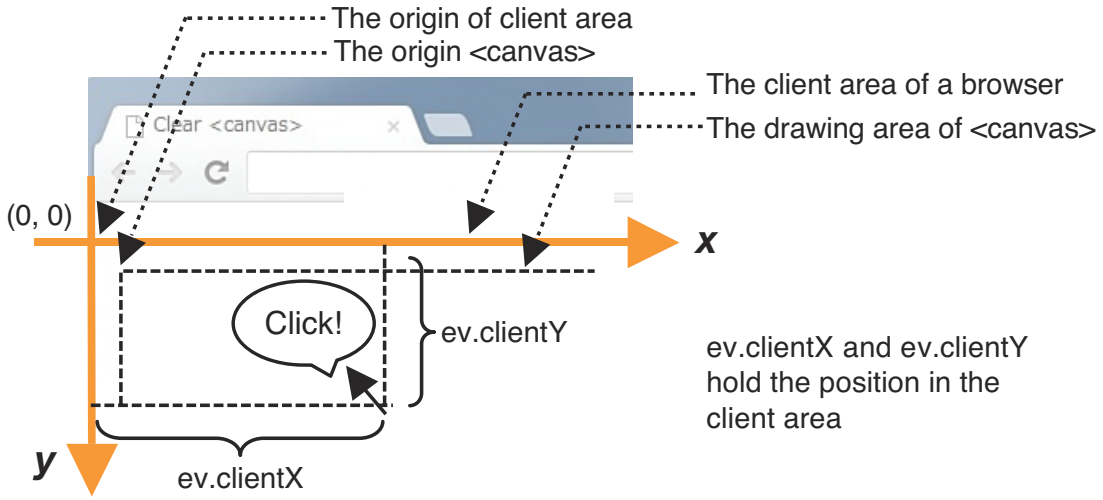
- http://rodger.global-linguist.com/webgl/ch02/ClickedPoints.html
- Multiple points are added according to mouse clicks
- What to learn
  - How to handle JavaScript mouse events
  - How to re-draw a 3D scene
- How it works
  - Whenever the mouse is clicked,
    - a new point is added to the list and
    - all the recorded points are drawn
- Each point is drawn by calling `gl.drawArrays()` separately
  → bad design
  → We should draw them all at once using a “buffer object” (Chapter 3)

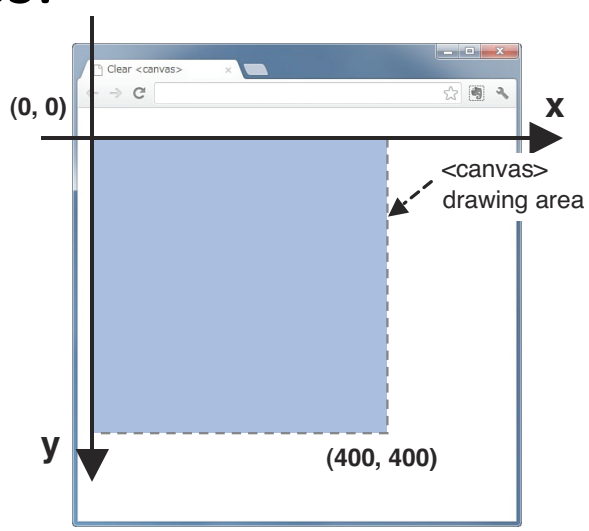
Register Event Handlers

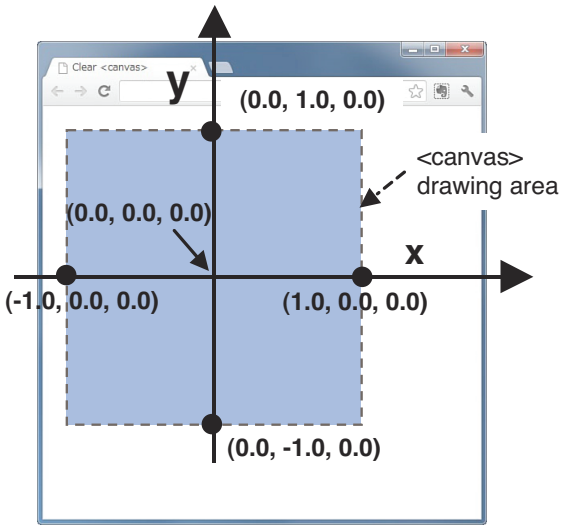
- How to handle events? → DOM on-event handlers (by MDN)
- How to specify the event handler?
  - In <canvas> tag
    - https://xregy.github.io/webgl/ClickedPoints_html.html
    - Values (`gl`, `canvas`, `a_Position`) should be passed to the handler as global variables
  - Dynamically set to an JS anonymous function (`ClickedPoints.html`)
    - Local values can be passed → better design

Handling Mouse Click Events

- Procedure
  1. Retrieve the position of the mouse click and then store it in an array.
  2. Clear `<canvas>`.
  3. For each position stored in the array, draw a point.
- The mouse position stored in the `mousedown` event needs to be converted to the WebGL coordinates.







Handling Mouse Click Events

- Mouse positions (points) are appended to the JS array dynamically by push() method.
- Since we're setting the "static" vertex attributes, each point should be drawn separately by calling `gl.drawArrays()`
  - Not practical
  - We have to draw them using only one function call using a buffer object. (Chapter 3)

Experimenting with the Sample Program

- What if we do not clear (gl.clear()) at each drawing?
  - The color buffer is reinitialized to (0,0,0,0) hence the canvas becomes completely transparent.
- Try to store the (x,y) coordinates in one array as “array of arrays”

Example #6: `ColoredPoints`

Example #6: `ColoredPoints`

- http://rodger.global-linguist.com/webgl/ch02/ColoredPoints.html
- Different color assigned to the points depending on their position.
- What to learn
  - How to pass data to a fragment shader using uniform variables.
- Procedure (to pass data color to the frag shader as a uniform variable)
  1. Prepare the uniform variable for the color in the fragment shader.
  2. Assign the uniform variable to the `gl_FragColor` variable.
  3. Pass the color data to the uniform variable from the JavaScript program.
     1. Retrieve the storage location of a uniform variable.
     2. Assign a value to a uniform variable.

Uniform Variables

- Available to both vertex & fragment shaders
  - c.f. attribute variables, varying variables (Chapter 5)
- Declared as “<storate qualifier> <type> <variable name>”
  - e.g. `uniform vec4 u_FragColor;`
- “Precision qualifier” will be covered in Chapter 5.

Retrieving the Storage Location of a Uniform Variable

- Using `gl.getUniformLocation()`
- `null` returned if failed
  - c.f. -1 returned by `gl.getAttribLocation()`

Assigning a Value to a Uniform Variable

- Using `gl.uniform*()` functions