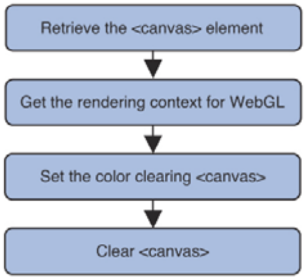

Retrieve the <canvas> element
Get the rendering context for WebGL
Set the color clearing <canvas>
Clear <canvas>

Figure 2.8 The processing flow of the main() function

WebGL

- Main –
  - JavaScript
- Vertex Shader
  - GLSL
- Fragment Shader
  - GLSL
- Event driven input HTML5

- GLSL
- ~ C
- + vector and Matrices basic types
- + C++ features
    - Operator overloading
- Angel: MV.js
  - Support
    - Graphics functions
    - Types, Operations in GLSL

WebGL

- DOM API
  - Creating 3D graphics in Web browser
- Based on OpenGL ES 2.0
- Use GLSL - OpenGL Shading Language
- HTML
  - JavaScript infrastructur
  - Document Object Model (DOM)
- WebGL another rendering context on <canvas> element
- Combines with HTML (and other web content) layered

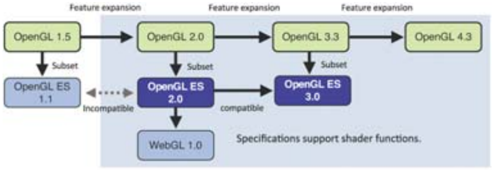


Figure 1.4 Relationship among OpenGL, OpenGL ES 1.1/2.0/3.0, and WebGL

Angel Common files

- webgl-utils.js: standard utilities from google to set up a webgl context
- MV.js: our matrix/vector package. Documentation on website
- initShaders.js: functions to initialize shaders in the html file
- initShaders2.js: functions to initialize shaders that are in separate files

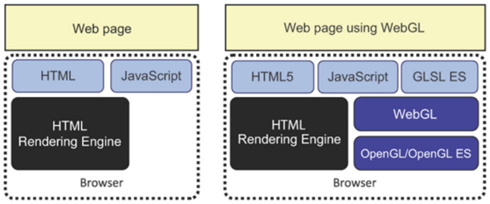
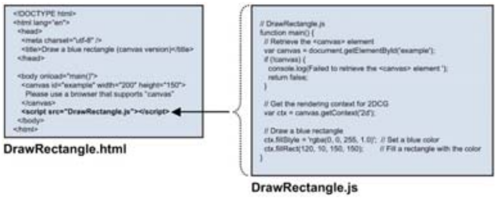


Figure 1.5 The software architecture of dynamic web pages (left) and web pages using WebGL (right)



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Draw a blue rectangle (canvas version)</title>
  </head>

  <body onload="main()">
    <canvas id="example" width="200" height="150">
      Please use a browser that supports "canvas"
    </canvas>
    <script src="DrawRectangle.js"></script>
  </body>
</html>
```

DrawRectangle.html

```
// DrawRectangle.js
function main() {
  // Retrieve the <canvas> element
  var canvas = document.getElementById('example');
  if (!canvas) {
    console.log('Failed to retrieve the <canvas> element ');
    return false;
  }

  // Get the rendering context for 2DCG
  var ctx = canvas.getContext('2d');

  // Draw a blue rectangle
  ctx.fillStyle = 'rgba(0, 0, 255, 1.0)'; // Set a blue color
  ctx.fillRect(120, 10, 150, 150);        // Fill a rectangle with the color
}
```

DrawRectangle.js

Figure 2.3 DrawRectangle.html and DrawRectangle.js

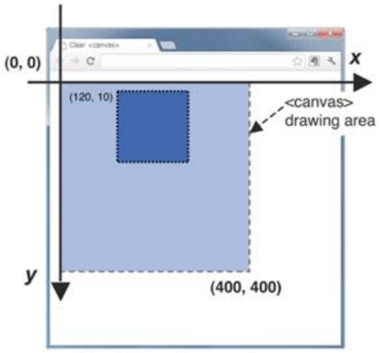
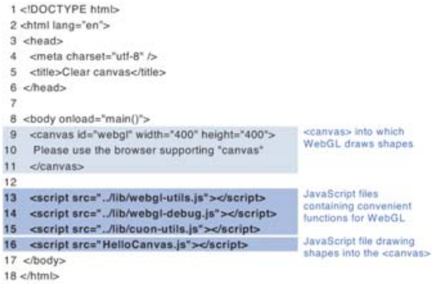


Figure 2.5 The coordinate system of <canvas>



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Clear canvas</title>
</head>

<body onload="main()">
  <canvas id="webgl" width="400" height="400">
  Please use the browser supporting "canvas"
  </canvas>

  <script src="../lib/webgl-utils.js"></script>
  <script src="../lib/webgl-debug.js"></script>
  <script src="../lib/cuon-utils.js"></script>
  <script src="HelloCanvas.js"></script>
</body>
</html>
```

<canvas> into which WebGL draws shapes

JavaScript files containing convenient functions for WebGL

JavaScript file drawing shapes into the <canvas>

Figure 2.7 HelloCanvas.html

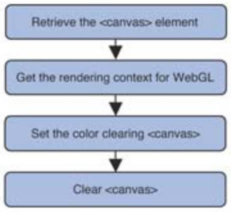

Retrieve the <canvas> element
Get the rendering context for WebGL
Set the color clearing <canvas>
Clear <canvas>

Figure 2.8 The processing flow of the main() function

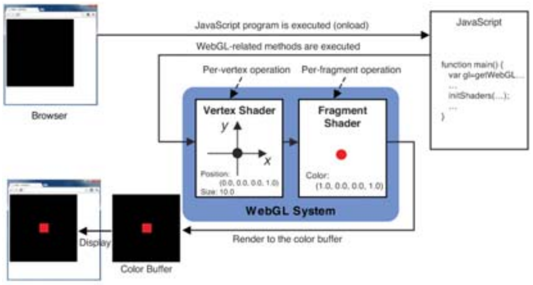
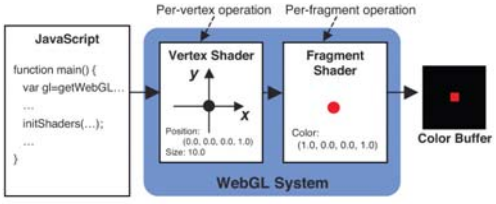
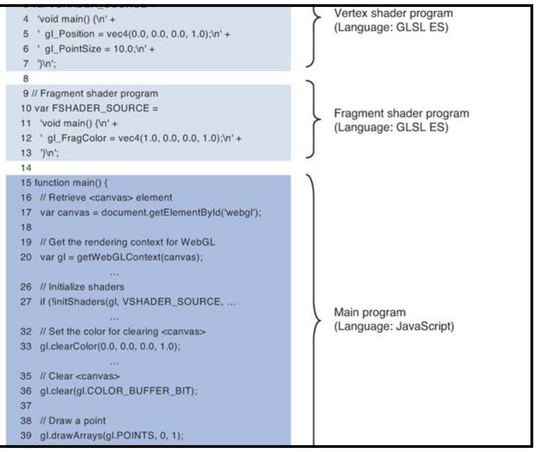


Figure 2.10 The processing flow from executing a JavaScript program to displaying the result in a browser



```
function main() {
  var gl=getWebGL...
  ...
  initShaders(...);
  ...
}
```

Figure 2.11 The simplified version of Figure 2.9



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

// Fragment shader program
var FSHADER_SOURCE =
 'void main() {\n' +
 '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
 '}\n';

function main() {
 // Retrieve <canvas> element
 var canvas = document.getElementById('webgl');

 // Get the rendering context for WebGL
 var gl = getWebGLContext(canvas);
                    ...
 // Initialize shaders
 if (!initShaders(gl, VSHADER_SOURCE, ...
                    ...
 // Set the color for clearing <canvas>
 gl.clearColor(0.0, 0.0, 0.0, 1.0);
                    ...
 // Clear <canvas>
 gl.clear(gl.COLOR_BUFFER_BIT);

 // Draw a point
 gl.drawArrays(gl.POINTS, 0, 1);
```

Vertex shader program
(Language: GLSL ES)

Fragment shader program
(Language: GLSL ES)

Main program
(Language: JavaScript)

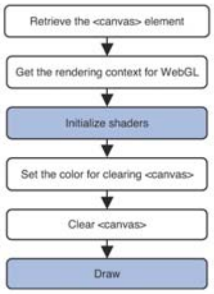
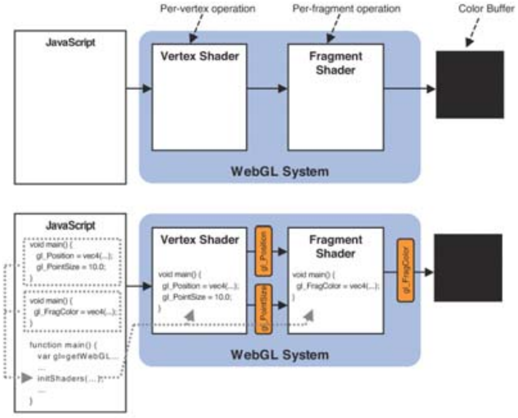

Retrieve the <canvas> element
Get the rendering context for WebGL
Initialize shaders
Set the color for clearing <canvas>
Clear <canvas>
Draw

Figure 2.13 The processing flow of a WebGL program

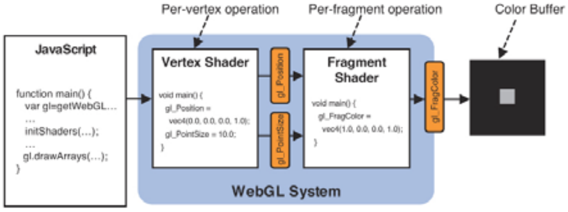


```
void main() {
  gl_Position = vec4(...);
  gl_PointSize = 10.0;
}
```

```
void main() {
  gl_FragColor = vec4(...);
}
```

```
function main() {
  var gl=getWebGL...
  ...
  initShaders(...);
  ...
}
```

Figure 2.14 Behavior of initShaders()



```
function main() {
  var gl=getWebGL...
  ...
  initShaders(...);
  ...
 gl.drawArrays(...);
}
```

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

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

Figure 2.15 The behavior of shaders

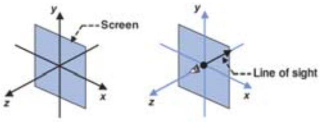


Figure 2.16 WebGL coordinate system

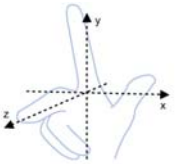


Figure 2.17 The right-handed coordinate system

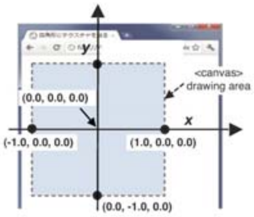
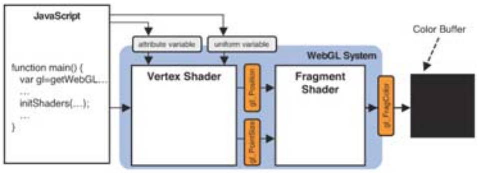
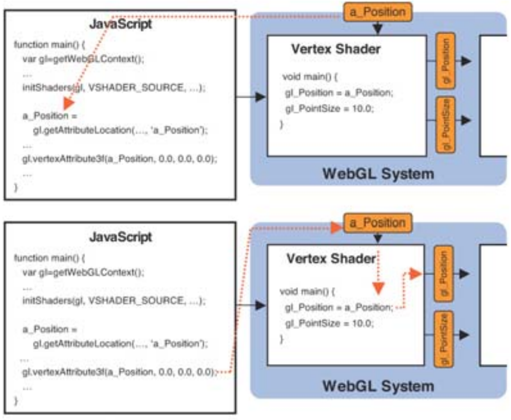


Figure 2.18 The <canvas> drawing area and WebGL coordinate system



```
function main() {
  var gl=getWebGL...
  ...
  initShaders(...);
  ...
}
```

Figure 2.20 Two ways to pass data to a vertex shader



```
function main() {
  var gl=getWebGLContext();
  ...
  initShaders(gl, VSHADER_SOURCE, ...);

  a_Position =
    gl.getAttributeLocation(..., 'a_Position');
  ...
  gl.vertexAttribute3f(a_Position, 0.0, 0.0, 0.0);
  ...
}
```

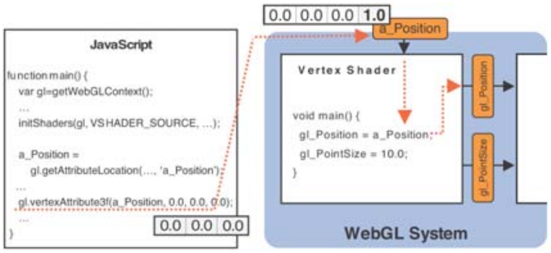
```
void main() {
  gl_Position = a_Position;
  gl_PointSize = 10.0;
}
```

```
function main() {
  var gl=getWebGLContext();
  ...
  initShaders(gl, VSHADER_SOURCE, ...);

  a_Position =
    gl.getAttributeLocation(..., 'a_Position');
  ...
  gl.vertexAttribute3f(a_Position, 0.0, 0.0, 0.0);
  ...
}
```

```
void main() {
  gl_Position = a_Position;
  gl_PointSize = 10.0;
}
```

Figure 2.22 Getting the storage location of an attribute variable and then writing a value to the variable



```
function main() {
  var gl=getWebGLContext();
  ...
  initShaders(gl, VSHADER_SOURCE, ...);

  a_Position =
    gl.getAttributeLocation(..., 'a_Position');
  ...
  gl.vertexAttribute3f(a_Position, 0.0, 0.0, 0.0);
  ...
}
```

```
void main() {
  gl_Position = a_Position;
  gl_PointSize = 10.0;
}
```

Figure 2.23 The missing data is automatically supplied

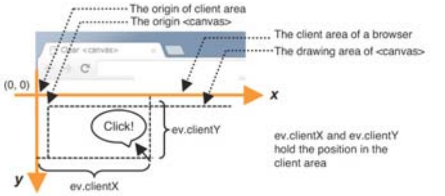

The origin of client area
The origin <canvas>
The client area of a browser
The drawing area of <canvas>
Clear <canvas>
(0, 0)
x
y
Click!
ev.clientY
ev.clientX
ev.clientX and ev.clientY hold the position in the client area

Figure 2.26 The coordinate system of a browser's client area and the position of the <canvas>

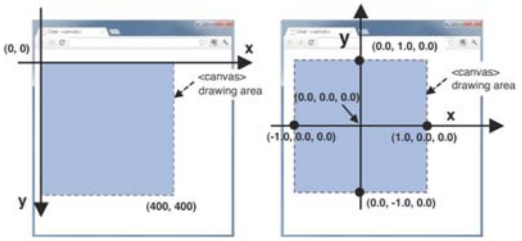
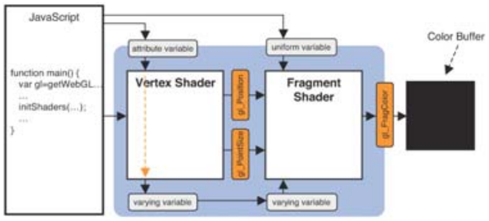
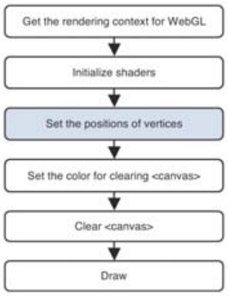


Figure 2.27 The coordinate system of <canvas> (left) and that of WebGL on <canvas> (right)



```
function main() {
  var gl=getWebGL...
  ...
  initShaders(...);
  ...
}
```

Figure 2.30 Two ways of passing a data to a fragment shader



Get the rendering context for WebGL

Initialize shaders

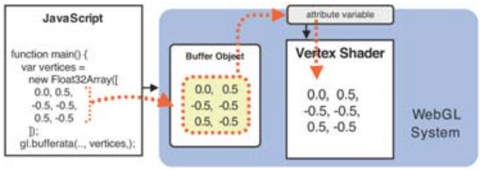
Set the positions of vertices

Set the color for clearing <canvas>

Clear <canvas>

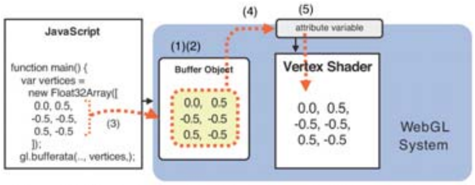
Draw

Figure 3.3 Processing flowchart for MultiPoints.js



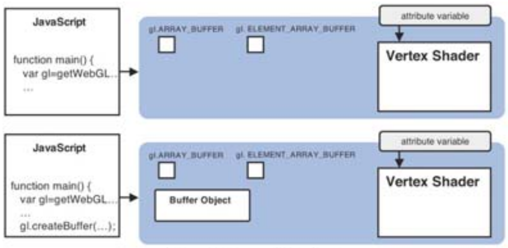
```
function main() {
  var vertices =
    new Float32Array([
      0.0, 0.5,
      -0.5, -0.5,
      0.5, -0.5
    ]);
  gl.bufferata(.., vertices,);
```

Figure 3.4 Passing multiple vertices to a vertex shader by using a buffer object



```
function main() {
  var vertices =
    new Float32Array([
      0.0, 0.5,
      -0.5, -0.5,
      0.5, -0.5
    ]);
  gl.bufferata(.., vertices,);
```

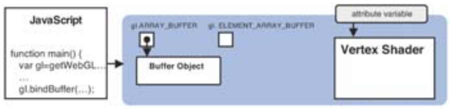
Figure 3.5 The five steps to pass multiple data values to a vertex shader using a buffer object



```
function main() {
  var gl=getWebGL...
  ...
```

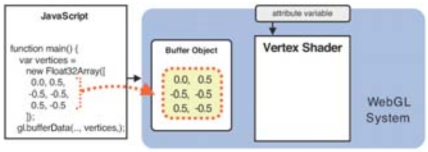
```
function main() {
  var gl=getWebGL...
  ...
  gl.createBuffer(....);
```

Figure 3.6 Create a buffer object



```
function main() {
  var gl=getWebGL...
  ...
  gl.bindBuffer(...);
```

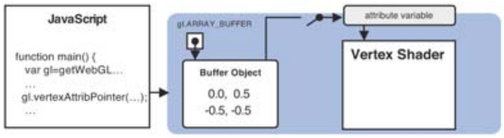
Figure 3.7 Bind a buffer object to a target



```
function main() {
  var vertices =
    new Float32Array([
     0.0, 0.5,
     -0.5, -0.5,
     0.5, -0.5
   ]);
  gl.bufferData(..., vertices,);
```

```
 0.0,  0.5
-0.5, -0.5
 0.5, -0.5
```

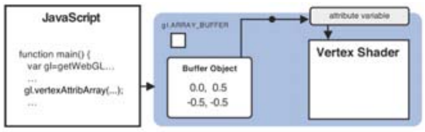
Figure 3.8 Allocate storage and write data into a buffer object



```
function main() {
  var gl=getWebGL...
  ...
  gl.vertexAttribPointer(...);
  ...
```

```
 0.0,  0.5
-0.5, -0.5
```

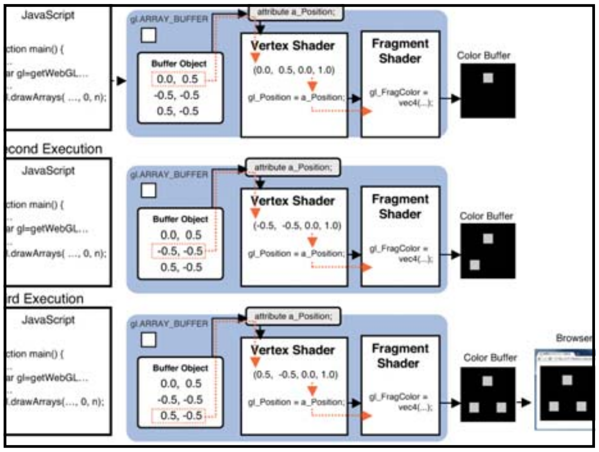
Figure 3.9 Assign a buffer object to an attribute variable



```
function main() {
  var gl=getWebGL...
  ...
  gl.vertexAttribArray(...);
  ...
```

```
 0.0,  0.5
-0.5, -0.5
```

Figure 3.10 Enable the assignment of a buffer object to an attribute variable



```
ction main() {
 ...
ar gl=getWebGL...
 ...
l.drawArrays( ..., 0, n);
```

```
 0.0,  0.5
-0.5, -0.5
 0.5, -0.5
```

```
(0.0,  0.5, 0.0, 1.0)
gl_Position = a_Position;
```

```
gl_FragColor =
      vec4(...);
```

```
(-0.5,  -0.5, 0.0, 1.0)
gl_Position = a_Position;
```

```
(0.5,  -0.5, 0.0, 1.0)
gl_Position = a_Position;
```

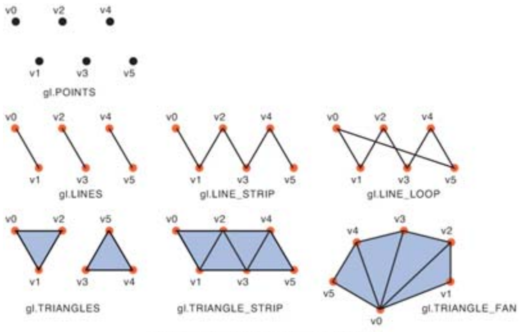


Figure 3.13 Basic shapes available in WebGL

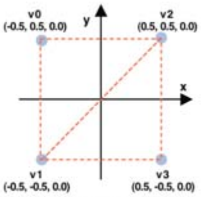


Figure 3.16 The four vertex coordinates of the rectangle

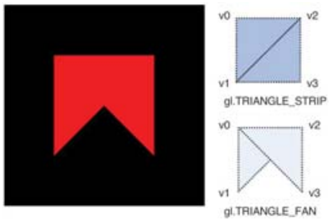


Figure 3.17 HelloQuad_FAN

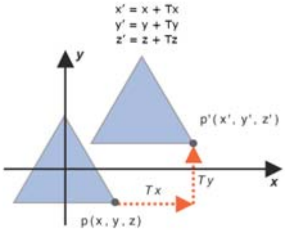


Figure 3.19 Calculating translation distances

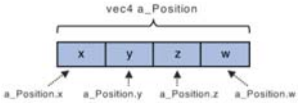


Figure 3.24 Access methods for each component in a vec4

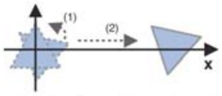


Figure 3.25 Rotate first and then translate a triangle

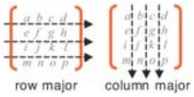


Figure 3.27 Row major order and column major order

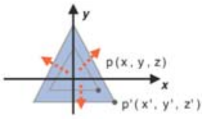


Figure 3.28 A scaling transformation

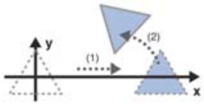


Figure 4.3 The triangle translated and then rotated

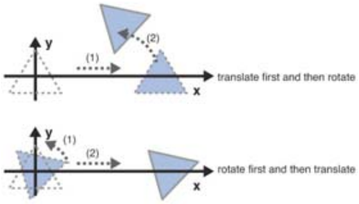

y
(1)
(2)
translate first and then rotate
x
y
(1)
(2)
rotate first and then translate
x

Figure 4.5 The order of transformations will show different results

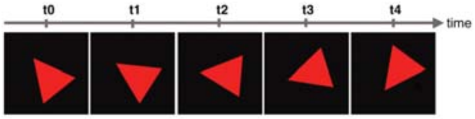
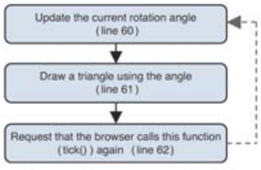


Figure 4.7 Draw a slightly different triangle for each drawing

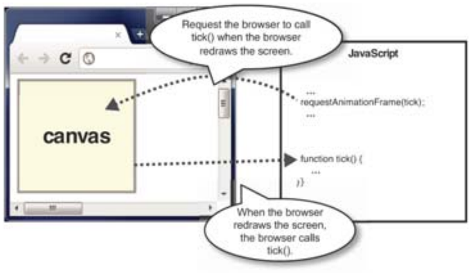


Update the current rotation angle (line 60)

Draw a triangle using the angle (line 61)

Request that the browser calls this function (tick()) again (line 62)

Figure 4.8 The operations assigned to "tick"


Request the browser to call tick() when the browser redraws the screen.
JavaScript
canvas
When the browser redraws the screen, the browser calls tick().

```
...
requestAnimationFrame(tick);
...

function tick() {
  ...
}}
```

Figure 4.9 The requestAnimationFrame () mechanism

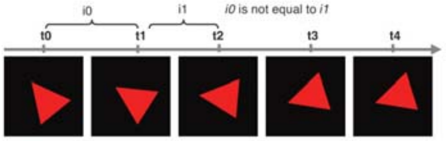


Figure 4.10 The interval times between each tick() vary

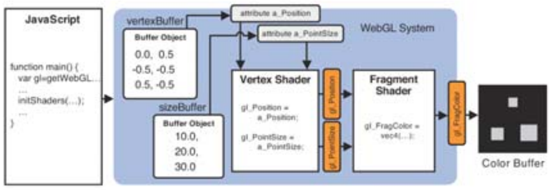


Figure 5.2 Using two buffer objects to pass data to a vertex shader

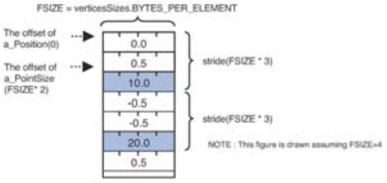


Figure 5.3 Stride and offset

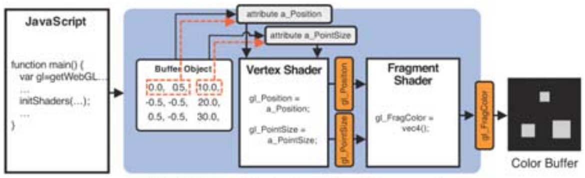


Figure 5.4 Internal behavior when stride and offset are used

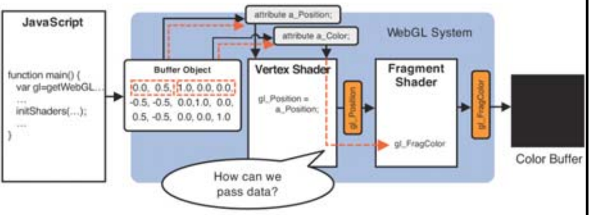


Figure 5.6 Passing data from a vertex shader to a fragment shader

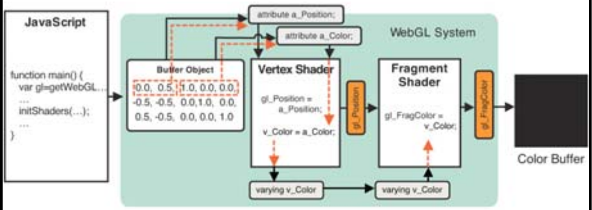
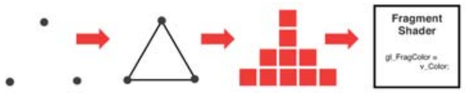


Figure 5.7 The behavior of a varying variable



.9 Vertex coordinate, identification of a triangle from the vertex coordinates, rasterization, and execution of a fragment shad

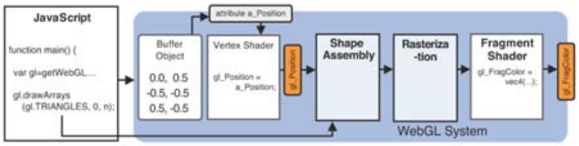
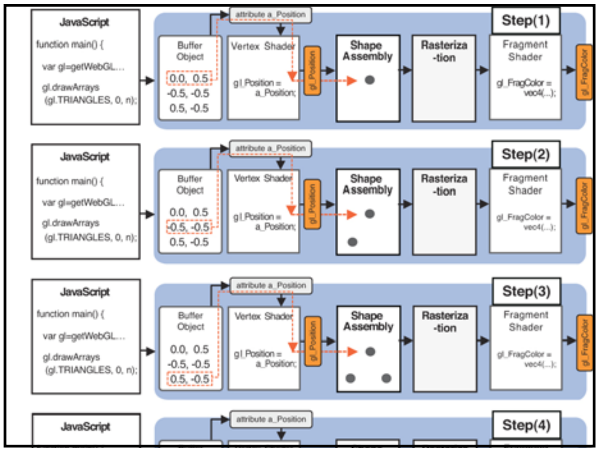


Figure 5.10 Assembly and rasterization between a vertex shader and a fragment shader



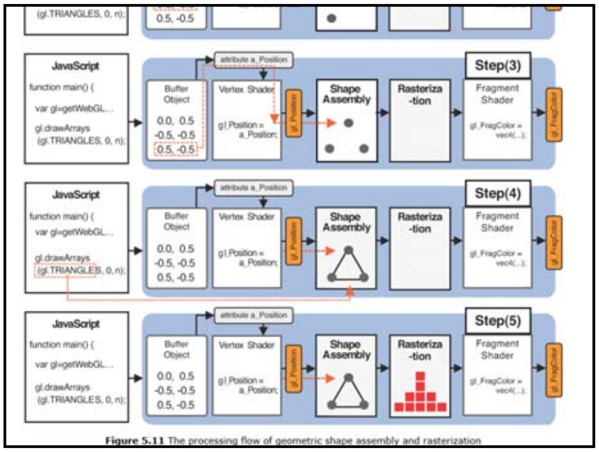


Figure 5.11 The processing flow of geometric shape assembly and rasterization

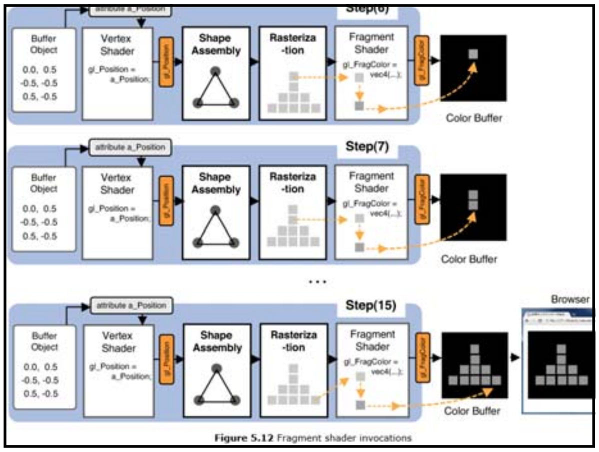


Figure 5.12 Fragment shader invocations