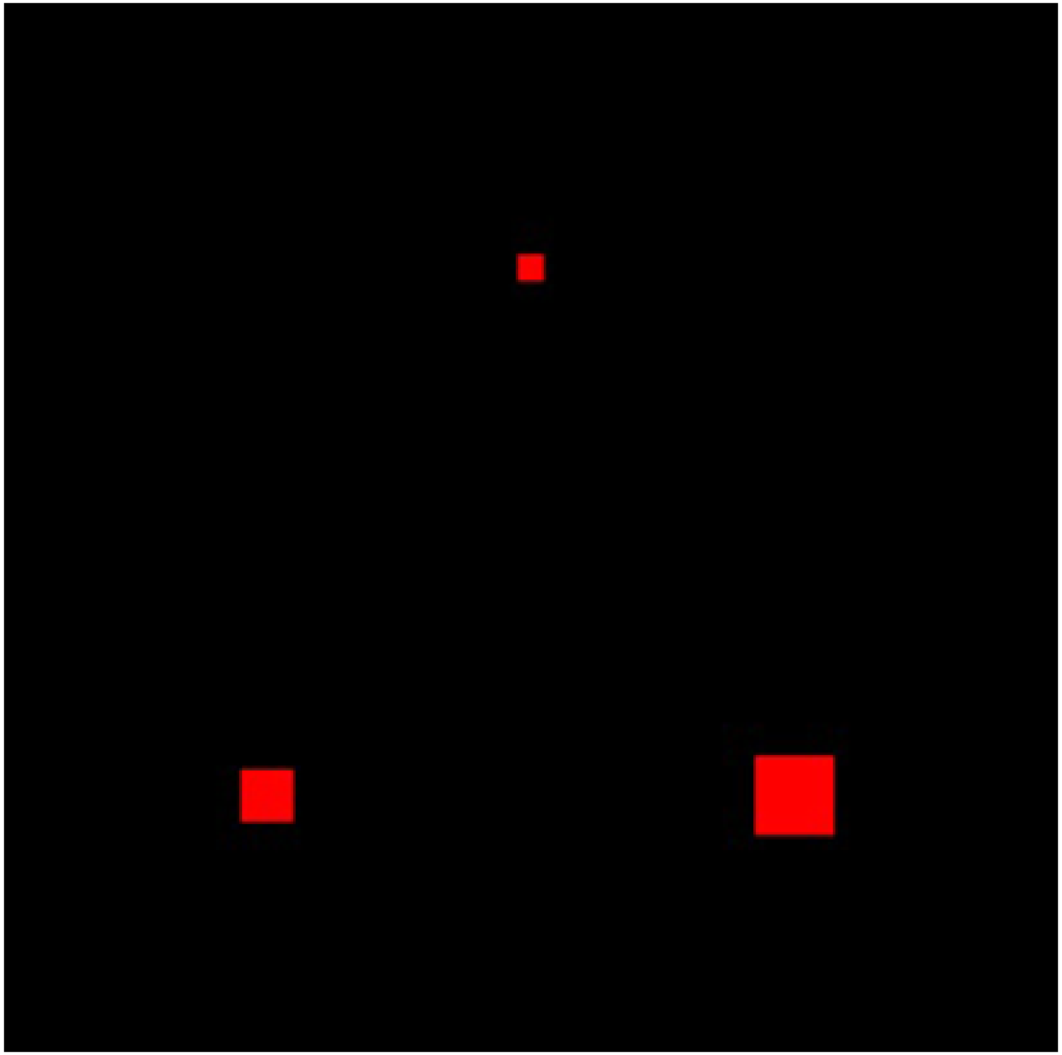
COSC 414/519I: Computer Graphics

2023W2

Shan Du

Multiple Attributes



Multiple Attributes

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute float a_PointSize;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '  gl_PointSize = a_PointSize;\n' +
  '}\n';
```

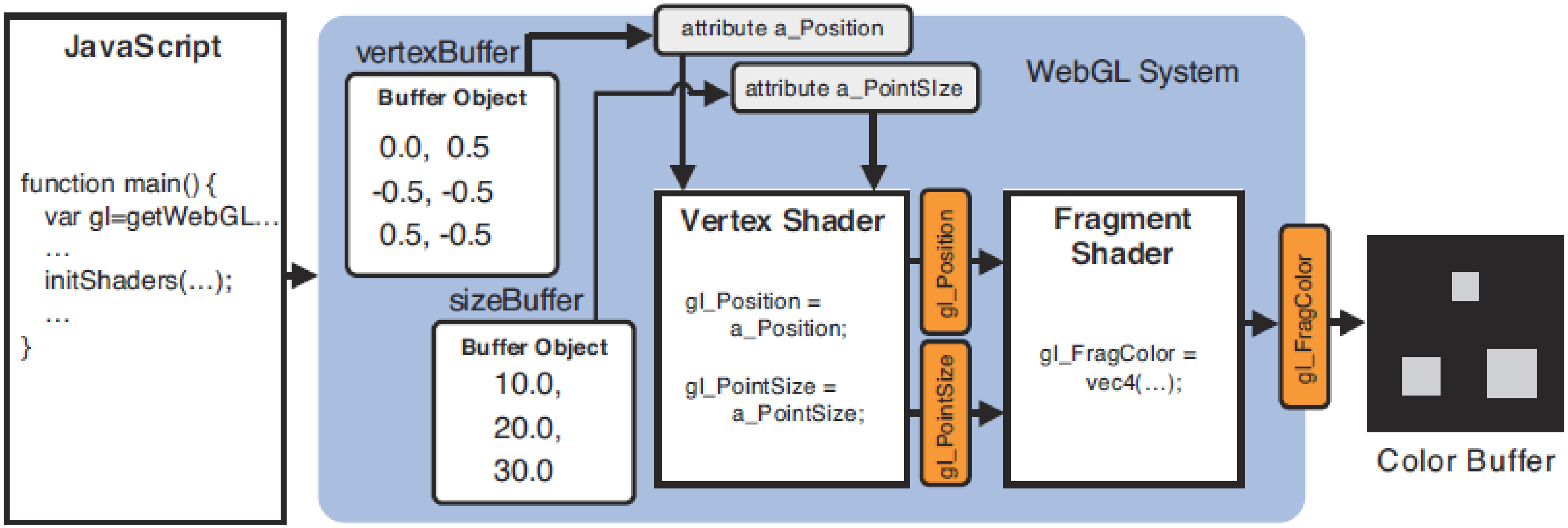
Multiple Attributes

```
……
var sizes = new Float32Array([
   10.0, 20.0, 30.0  // Point sizes
  ]);
……
 // Create a buffer object
  var vertexBuffer = gl.createBuffer();
  var sizeBuffer = gl.createBuffer();
```

Multiple Attributes

```
// Bind the point size buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER, sizeBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, sizes,
gl.STATIC_DRAW);
  var a_PointSize = gl.getAttribLocation(gl.program,
'a_PointSize’);
……
 gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT,
false, 0, 0);
  gl.enableVertexAttribArray(a_PointSize);
```

Multiple Attributes



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

```
gl_Position =
      a_Position;

gl_PointSize =
      a_PointSize;
```

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

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

Multiple Attributes

- One buffer object for multiple attributes

```
function initVertexBuffers(gl) {
 var verticesSizes = new Float32Array([
   // Coordinate and size of points
    0.0,  0.5,  10.0,  // the 1st point
   -0.5, -0.5,  20.0,  // the 2nd point
    0.5, -0.5,  30.0   // the 3rd point
 ]);
 var n = 3; // The number of vertices

 // Create a buffer object
 var vertexSizeBuffer = gl.createBuffer();
……
```

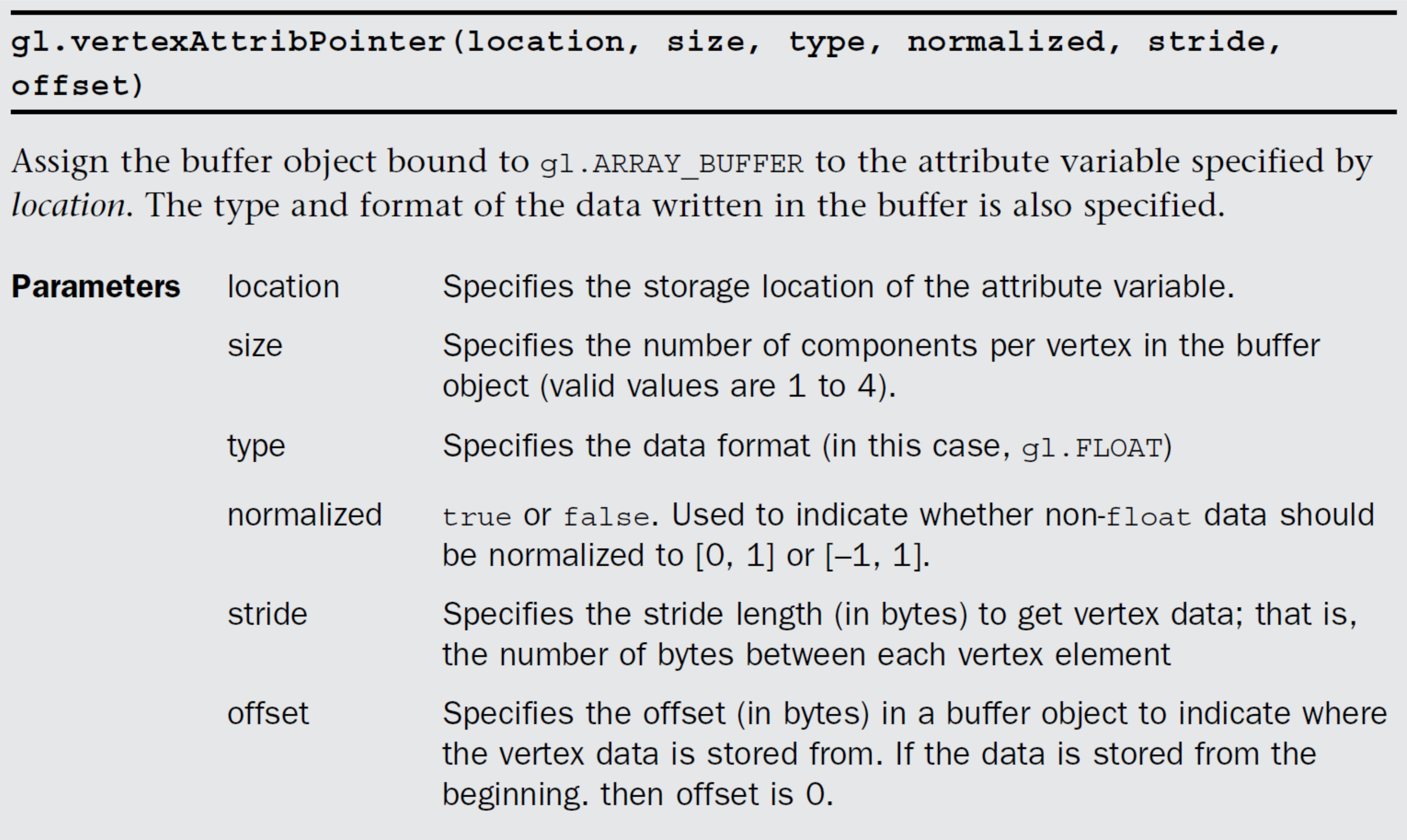
Multiple Attributes

```
// Bind the buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexSizeBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, verticesSizes,
gl.STATIC_DRAW);

  var FSIZE = verticesSizes.BYTES_PER_ELEMENT;
  //Get the storage location of a_Position, assign
and enable buffer
  var a_Position = gl.getAttribLocation(gl.program,
'a_Position');
……
```

Multiple Attributes

```
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE * 3, 0);
gl.enableVertexAttribArray(a_Position);  // Enable the assignment of the buffer object
  // Get the storage location of a_PointSize
  var a_PointSize = gl.getAttribLocation(gl.program, 'a_PointSize’);
……
gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT, false, FSIZE * 3, FSIZE * 2);
  gl.enableVertexAttribArray(a_PointSize);  // Enable buffer allocation
……
 return n;
}
```

Multiple Attributes



```
gl.vertexAttribPointer(location, size, type, normalized, stride,
offset)
```

Assign the buffer object bound to `gl.ARRAY_BUFFER` to the attribute variable specified by *location*. The type and format of the data written in the buffer is also specified.

Parameters	location	Specifies the storage location of the attribute variable.
	size	Specifies the number of components per vertex in the buffer object (valid values are 1 to 4).
	type	Specifies the data format (in this case, `gl.FLOAT`)
	normalized	`true` or `false`. Used to indicate whether non-`float` data should be normalized to [0, 1] or [–1, 1].
	stride	Specifies the stride length (in bytes) to get vertex data; that is, the number of bytes between each vertex element
	offset	Specifies the offset (in bytes) in a buffer object to indicate where the vertex data is stored from. If the data is stored from the beginning. then offset is 0.

Multiple Attributes

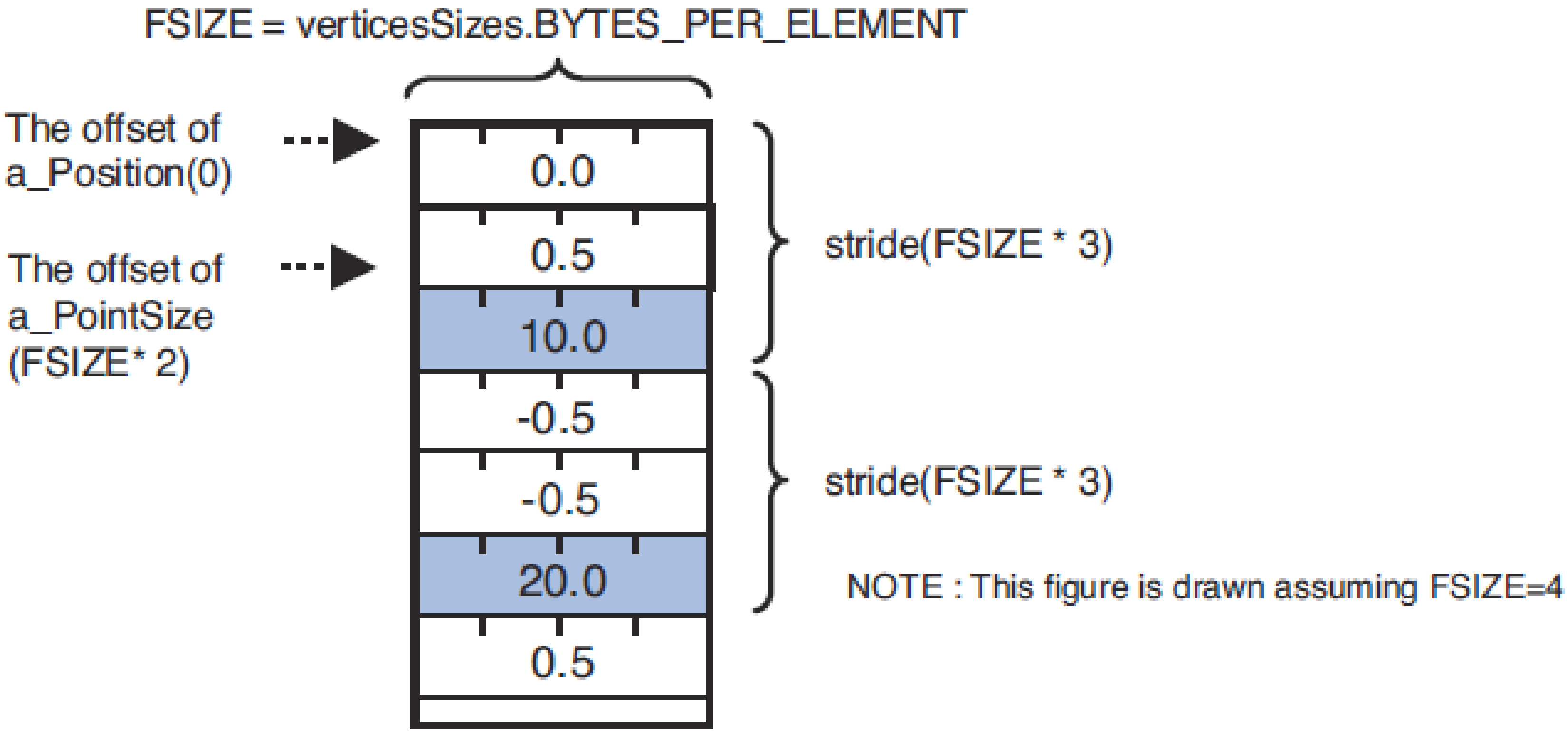

FSIZE = verticesSizes.BYTES_PER_ELEMENT
The offset of a_Position(0)
The offset of a_PointSize (FSIZE* 2)
0.0
0.5
10.0
-0.5
-0.5
20.0
0.5
stride(FSIZE * 3)
stride(FSIZE * 3)
NOTE : This figure is drawn assuming FSIZE=4

Figure 5.3 Stride and offset

Multiple Attributes

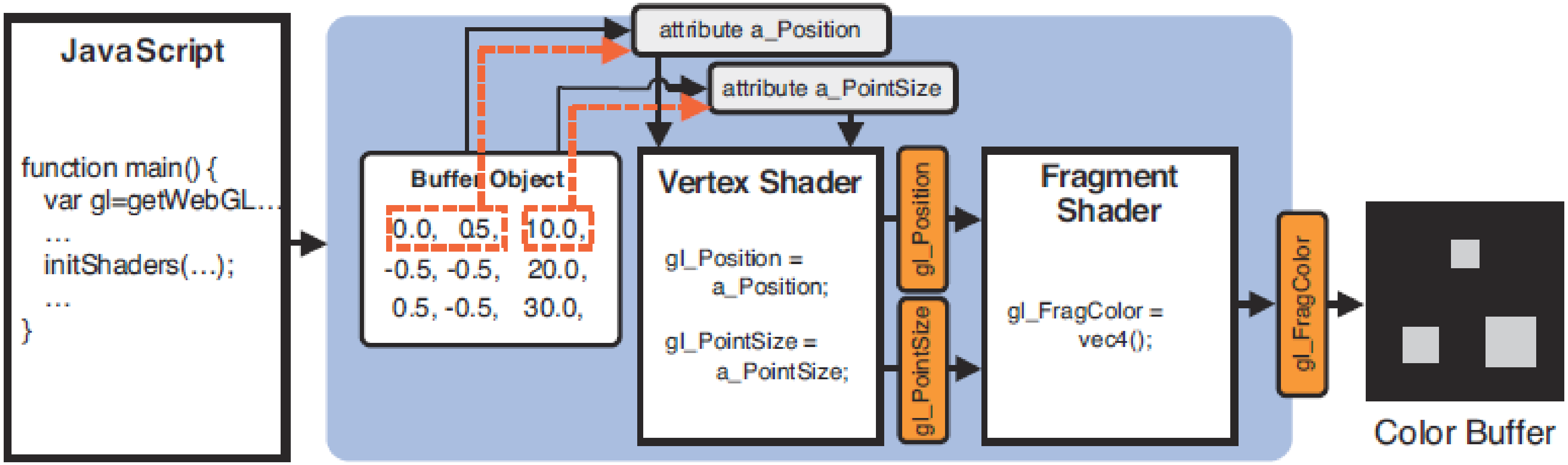
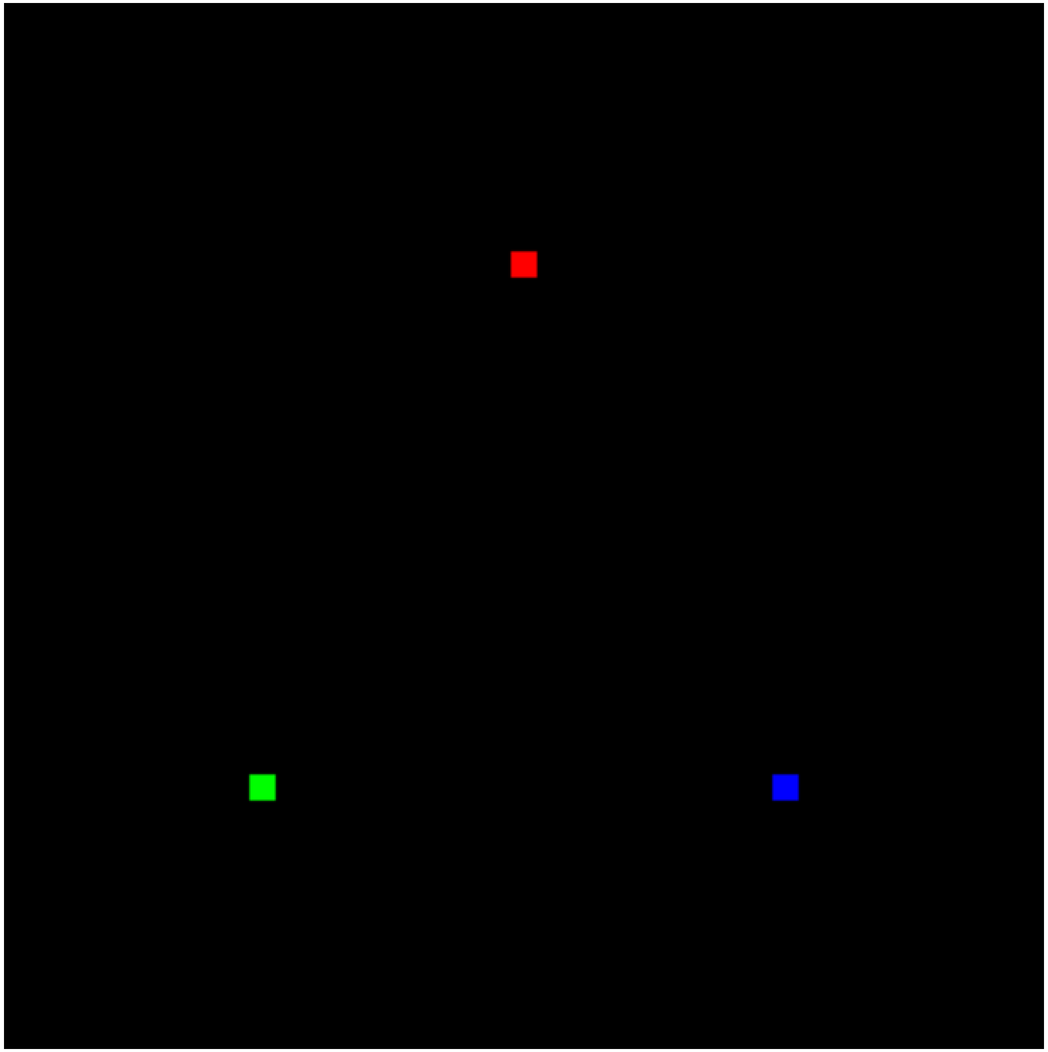


Figure 5.4 Internal behavior when stride and offset are used

Modifying the Color (Varying Variable)



Modifying the Color (Varying Variable)

- A uniform variable can be used to pass the color information to the fragment shader; however, because it is a “uniform” variable (not varying), it cannot be used to pass different colors for each vertex.
- We can send data from the vertex shader to the fragment shader: by using the varying variable.

Modifying the Color (Varying Variable)

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute vec4 a_Color;\n' +
  'varying vec4 v_Color;\n' + // varying variable
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '  gl_PointSize = 10.0;\n' +
  '  v_Color = a_Color;\n' +  // Pass the data to the fragment shader
  '}\n';

// Fragment shader program
var FSHADER_SOURCE =
  'precision mediump float;\n' + // Precision qualifier
  'varying vec4 v_Color;\n' +    // Receive the data from the vertex shader
  'void main() {\n' +
  '  gl_FragColor = v_Color;\n' +
  '}\n';
```

Modifying the Color (Varying Variable)

- A new varying variable *v_Color* is declared that will be used to pass its value to the fragment shader. Please note that you can only use *float* types (and related types *vec2* , *vec3* , *vec4* , *mat2* , *mat3* , and *mat4*) for varying variables.
- In WebGL, when varying variables declared inside the fragment shader have **identical names and types** to the ones declared in the vertex shader, the assigned values in the vertex shader are **automatically** passed to the fragment shader.

Modifying the Color (Varying Variable)

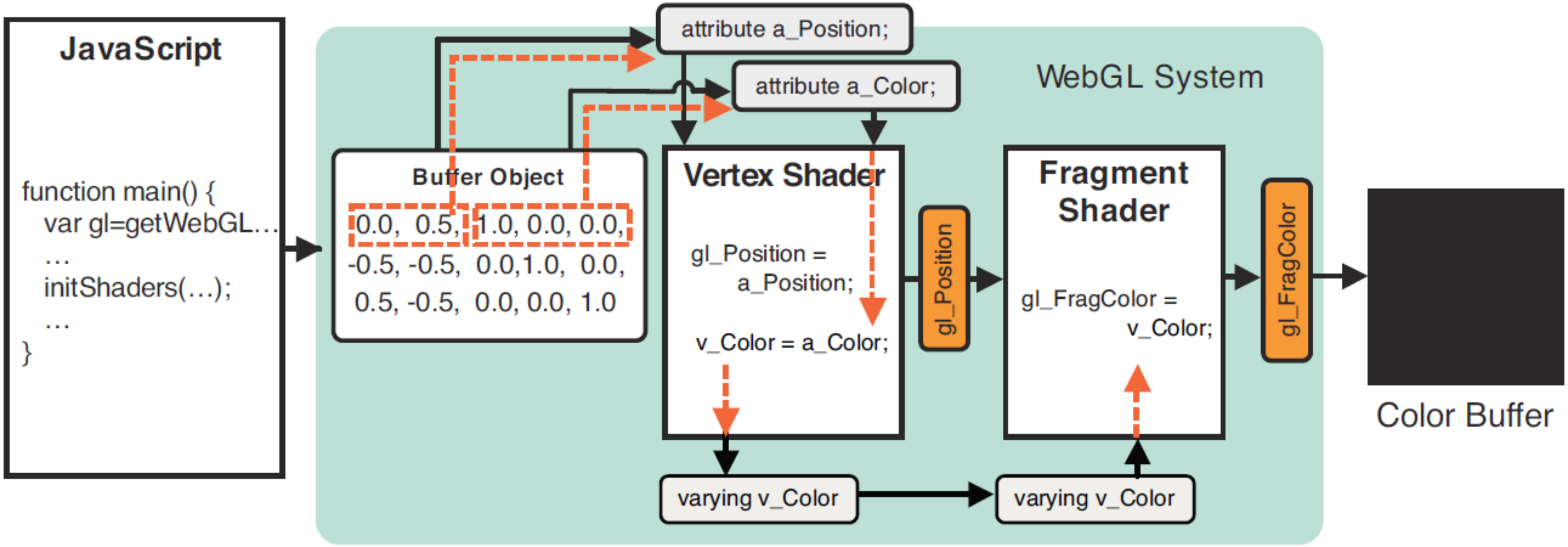


Figure 5.7 The behavior of a varying variable

Modifying the Color (Varying Variable)

```
var verticesColors = new Float32Array([
    // Vertex coordinates and color
     0.0,  0.5,  1.0,  0.0,  0.0,
    -0.5, -0.5,  0.0,  1.0,  0.0,
     0.5, -0.5,  0.0,  0.0,  1.0,
  ]);
……
// Create a buffer object
  var vertexColorBuffer = gl.createBuffer();
……
// Write the vertex coordinates and colors to the buffer object
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, verticesColors, gl.STATIC_DRAW);
var FSIZE = verticesColors.BYTES_PER_ELEMENT;
```

Modifying the Color (Varying Variable)

```
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE * 5, 0);
gl.enableVertexAttribArray(a_Position);  // Enable the assignment of the buffer object

  // Get the storage location of a_Color, assign buffer and enable
  var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
  if(a_Color < 0) {
    console.log('Failed to get the storage location of a_Color');
    return -1;
  }
  gl.vertexAttribPointer(a_Color, 3, gl.FLOAT, false, FSIZE * 5, FSIZE * 2);
  gl.enableVertexAttribArray(a_Color);  // Enable the assignment of the buffer object
```

Assembly and Rasterization

- There are actually two processes taking place between the vertex and the fragment shaders:
  - The geometric shape assembly process: In this stage, the geometric shape is assembled from the specified vertex coordinates. The first argument of *gl.drawArray()* specifies which type of shape should be assembled.
  - The rasterization process: In this stage, the geometric shape assembled in the geometric assembly process is converted into fragments.

Assembly and Rasterization

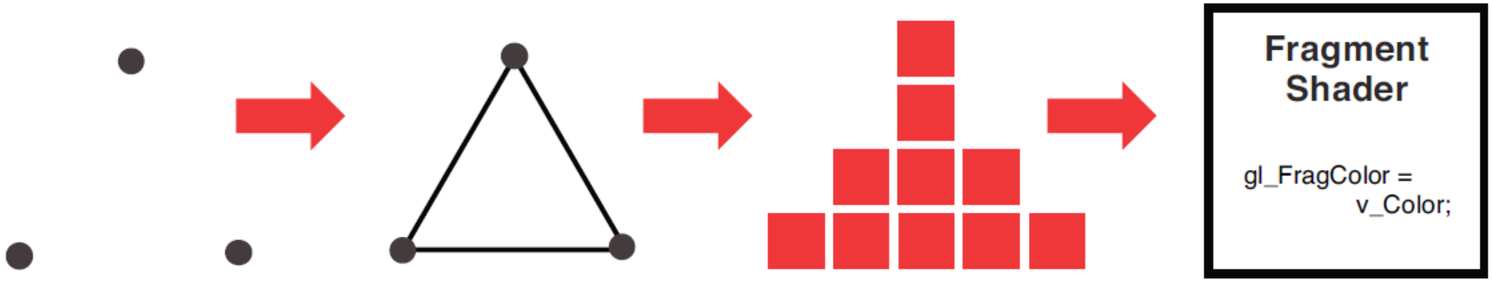


Figure 5.9 Vertex coordinate, identification of a triangle from the vertex coordinates, rasterization, and execution of a fragment shader

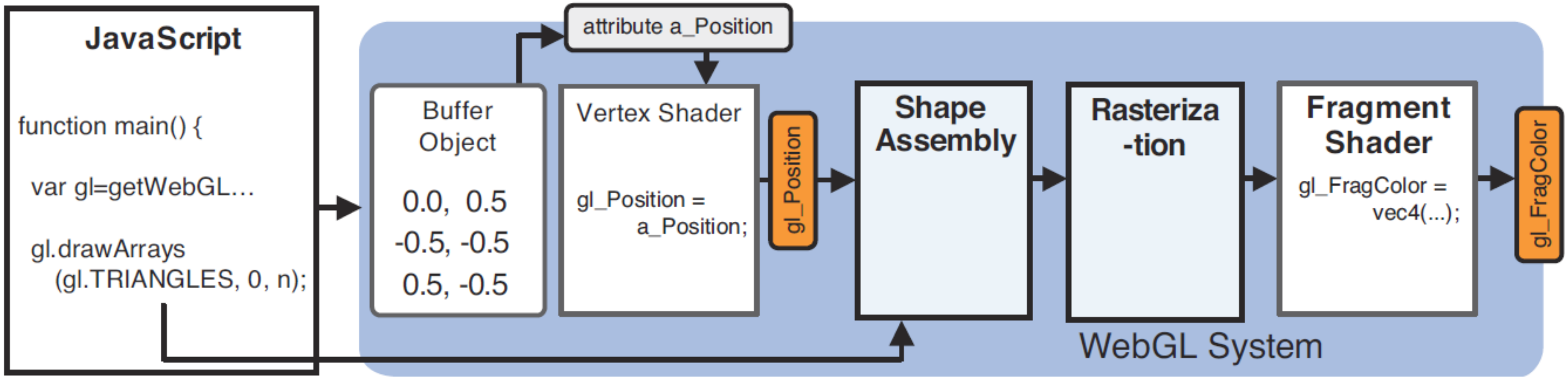


Figure 5.10 Assembly and rasterization between a vertex shader and a fragment shader

Basic Animation

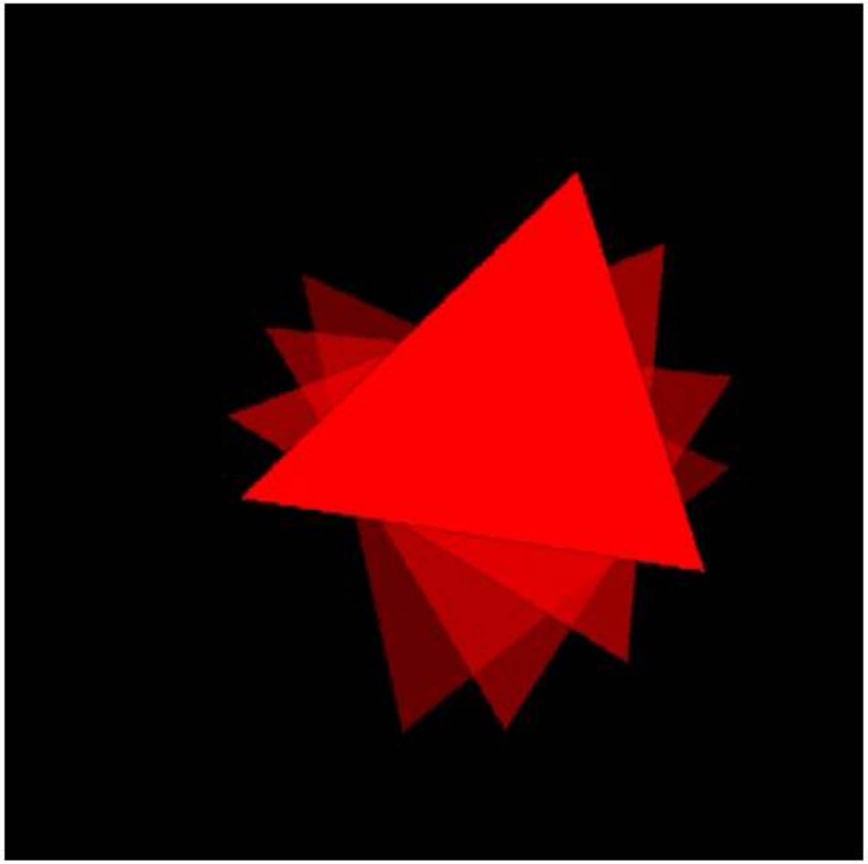


Figure 4.6 Multiple overlaid screenshots of RotatingTriangle

Continually rotates a triangle at a constant rotation speed (45 degrees/second)

Basic Animation

- To animate a rotating triangle, you simply need to redraw the triangle at a slightly different angle each time it draws. Of course, you need to clear the previous triangle before drawing a new one.

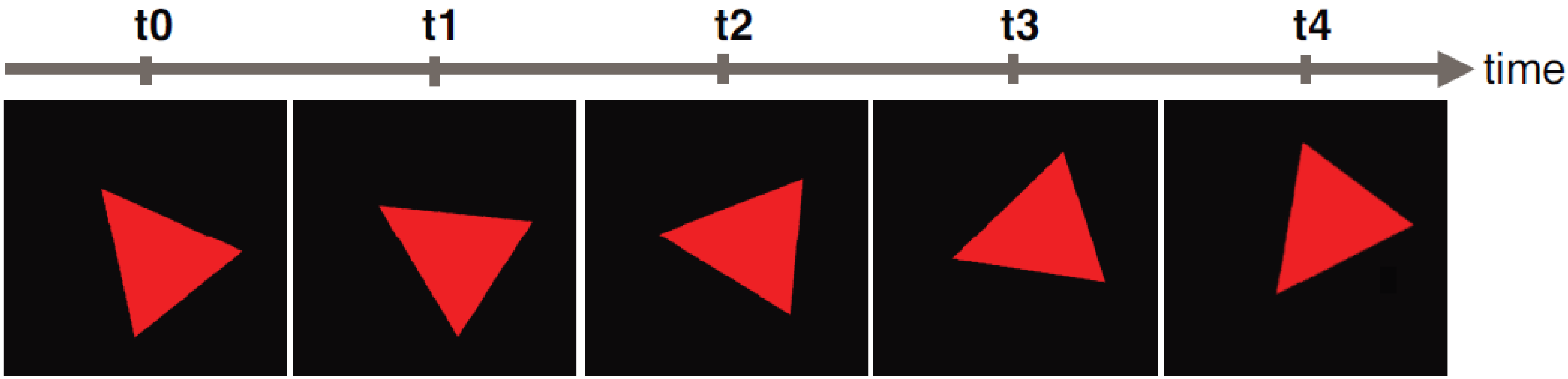


Figure 4.7 Draw a slightly different triangle for each drawing

Basic Animation

- Achieving animation requires two key mechanisms:
  - Mechanism 1: Repeatedly calls a function to draw a triangle at times t0, t1, t2, t3, and so on.
  - Mechanism 2: Clears the previous triangle and then draws a new one with the specified angle each time the function is called.

Basic Animation

```
// Rotation angle (degrees/second)
var ANGLE_STEP = 45.0;
function main() {
……
// Current rotation angle
  var currentAngle = 0.0;
  // Model matrix
  var modelMatrix = new Matrix4();

  // Start drawing
  var tick = function() {
    currentAngle = animate(currentAngle);  // Update the rotation angle
    draw(gl, n, currentAngle, modelMatrix, u_ModelMatrix);   // Draw the triangle
    requestAnimationFrame(tick, canvas); // Request that the browser calls tick
  };
  tick();
}
```

Basic Animation

```
function draw(gl, n, currentAngle, modelMatrix, u_ModelMatrix) {
 // Set the rotation matrix
 modelMatrix.setRotate(currentAngle, 0, 0, 1); // Rotation angle,
rotation axis (0, 0, 1)

 // Pass the rotation matrix to the vertex shader
 gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

 // Clear <canvas>
 gl.clear(gl.COLOR_BUFFER_BIT);

 // Draw the rectangle
 gl.drawArrays(gl.TRIANGLES, 0, n);
}
```

Basic Animation

```
// Last time that this function was called
var g_last = Date.now();
function animate(angle) {
  // Calculate the elapsed time
  var now = Date.now();
  var elapsed = now - g_last;
  g_last = now;
  // Update the current rotation angle (adjusted by the elapsed time)
  var newAngle = angle + (ANGLE_STEP * elapsed) / 1000.0;
  return newAngle %= 360;
}
```

Basic Animation

- Repeatedly Call the Drawing Function (tick())

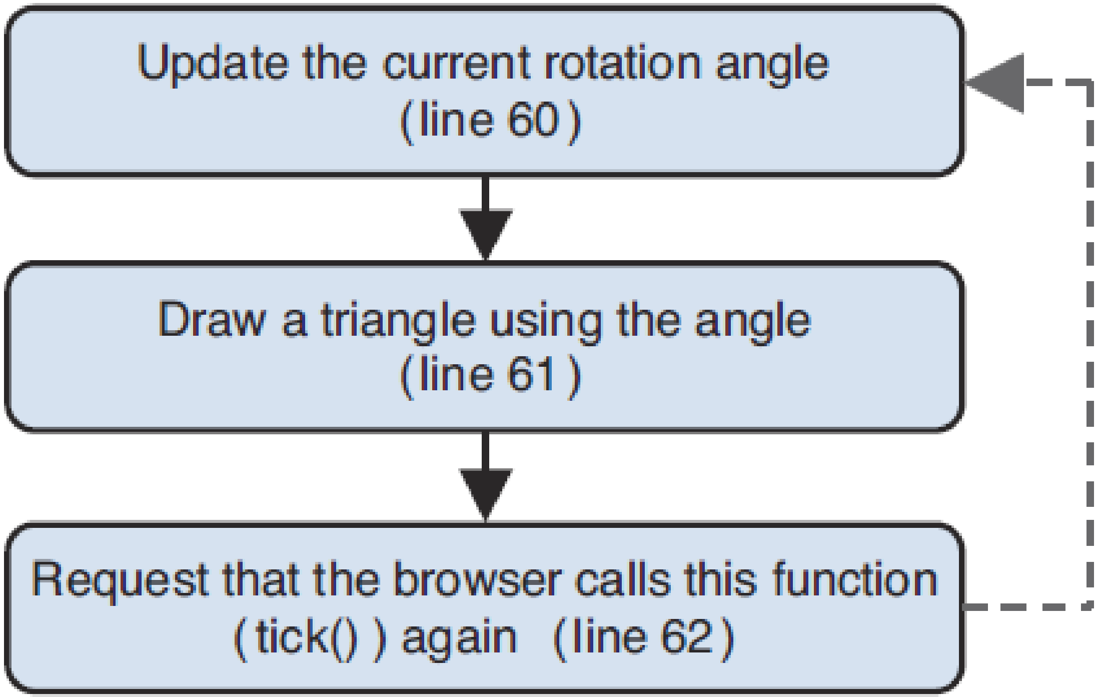

Update the current rotation angle
(line 60)
Draw a triangle using the angle
(line 61)
Request that the browser calls this function
(tick()) again (line 62)

Figure 4.8 The operations assigned to "tick"

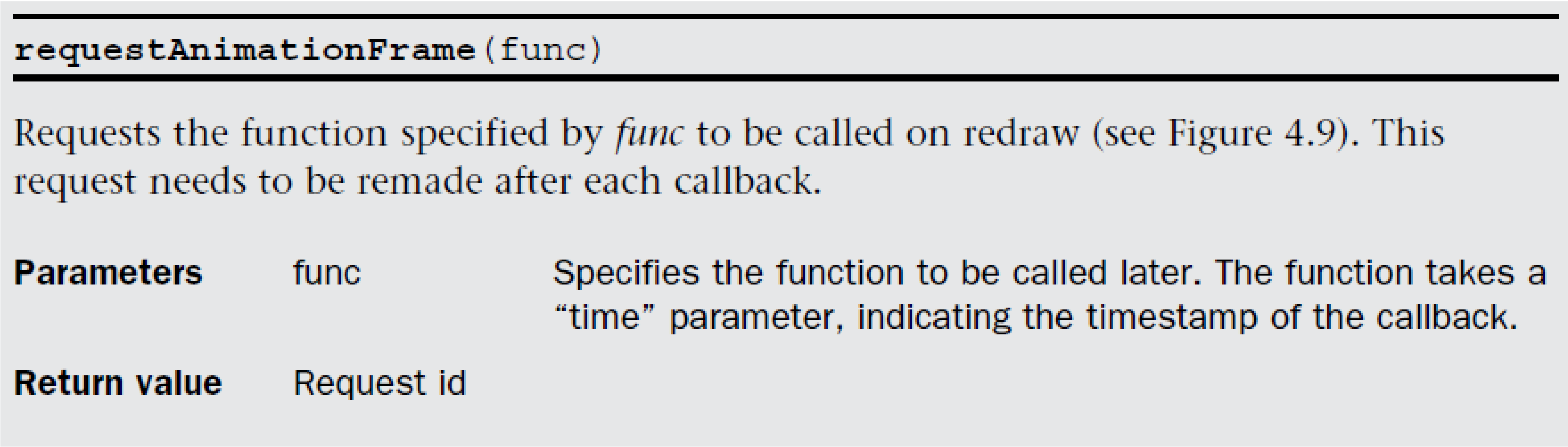
Basic Animation

- Draw a Triangle with the Specified Rotation Angle (draw())

```
function draw(gl, n, currentAngle, modelMatrix, u_ModelMatrix) {
  // Set the rotation matrix
  modelMatrix.setRotate(currentAngle, 0, 0, 1); // Rotation angle, rotation axis (0, 0, 1)

  // Pass the rotation matrix to the vertex shader
  gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

  // Clear <canvas>
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Draw the rectangle
  gl.drawArrays(gl.TRIANGLES, 0, n);
}
```

Basic Animation

- Request to Be Called Again (requestAnimationFrame())

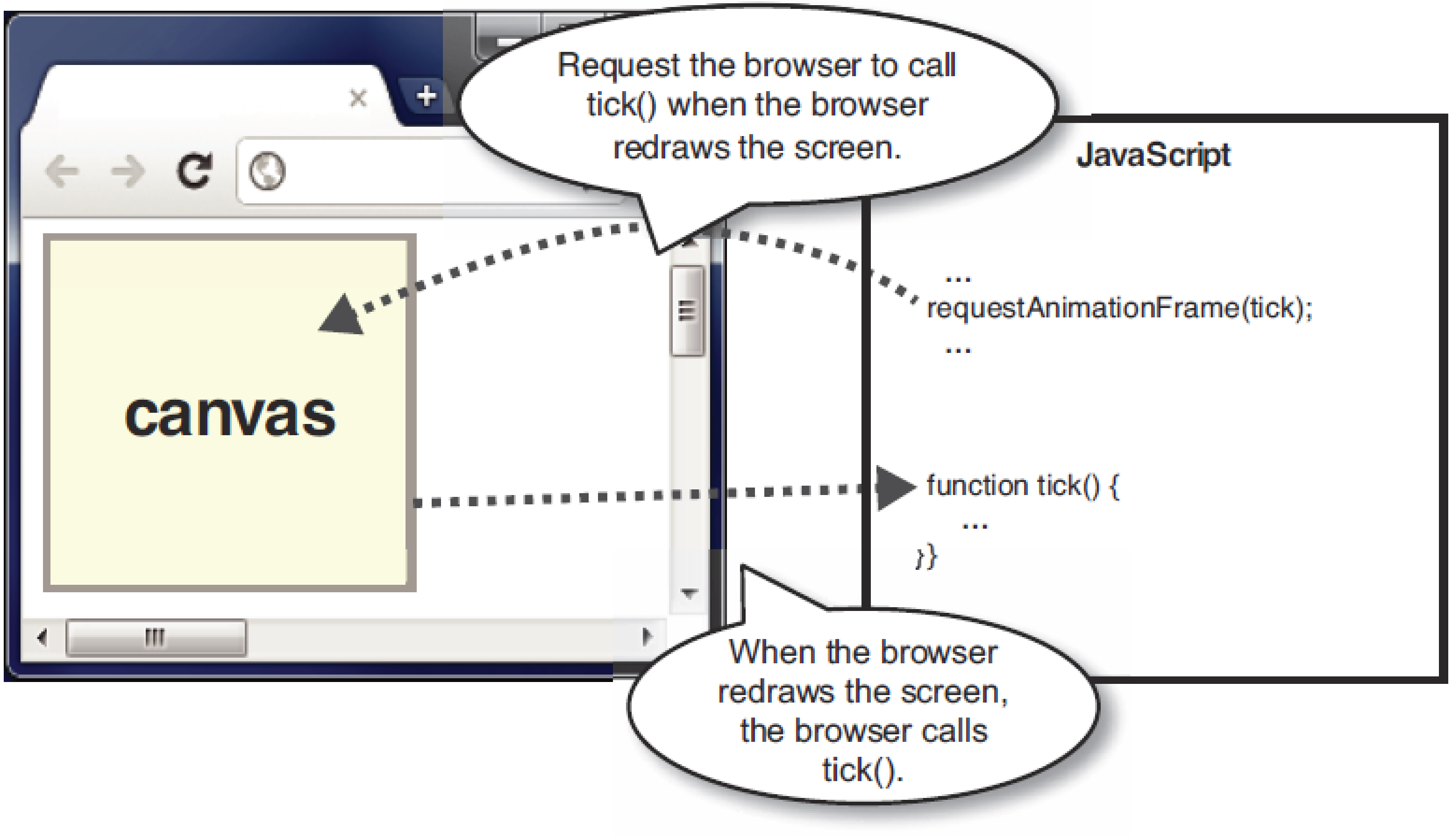


`requestAnimationFrame(func)`

Requests the function specified by *func* to be called on redraw (see Figure 4.9). This request needs to be remade after each callback.

Parameters	func	Specifies the function to be called later. The function takes a "time" parameter, indicating the timestamp of the callback.
Return value	Request id	

Basic Animation



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

Basic Animation

- Update the Rotation Angle (animate())

```
// Last time that this function was called
var g_last = Date.now();
function animate(angle) {
  // Calculate the elapsed time
  var now = Date.now();
  var elapsed = now - g_last;
  g_last = now;
  // Update the current rotation angle (adjusted by the elapsed time)
  var newAngle = angle + (ANGLE_STEP * elapsed) / 1000.0;
  return newAngle %= 360;
}
```