COSC 414/519I: Computer Graphics

2023W2

Shan Du

Viewing

- Specify exactly which objects should appear on the screen – where we point the camera and what lens we use.
- The specification of the objects in our scene is completely independent of our specification of the camera. Once we have specified both the scene and the camera, we can compose an image.

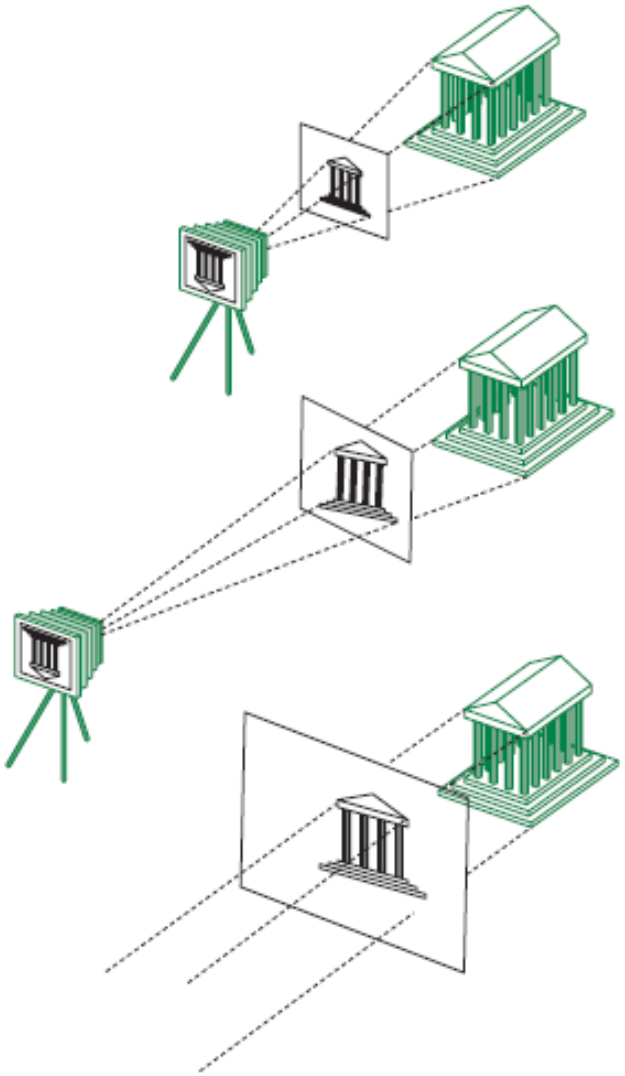
Viewing

- With APIs, the application developer just needs to worry about the specification of the parameters for the objects and the camera.

Orthographic View

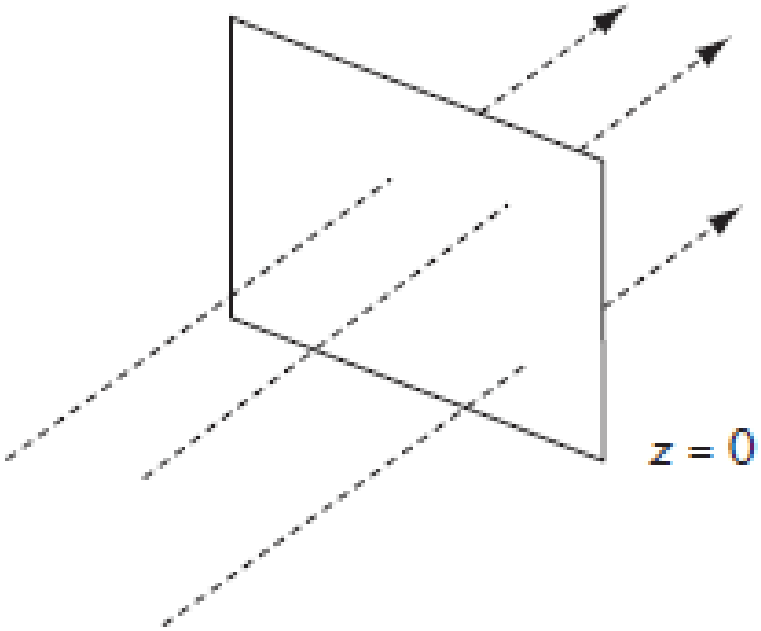
- The simplest view is the orthographic projection.
- Mathematically, the orthographic projection is what we would get if the camera in our synthetic camera model had an infinitely long lens and then we could place the camera infinitely far from our objects.
- In the limit, all the projectors become parallel and the center of projection is replaced by a direction of projection.

Orthographic View

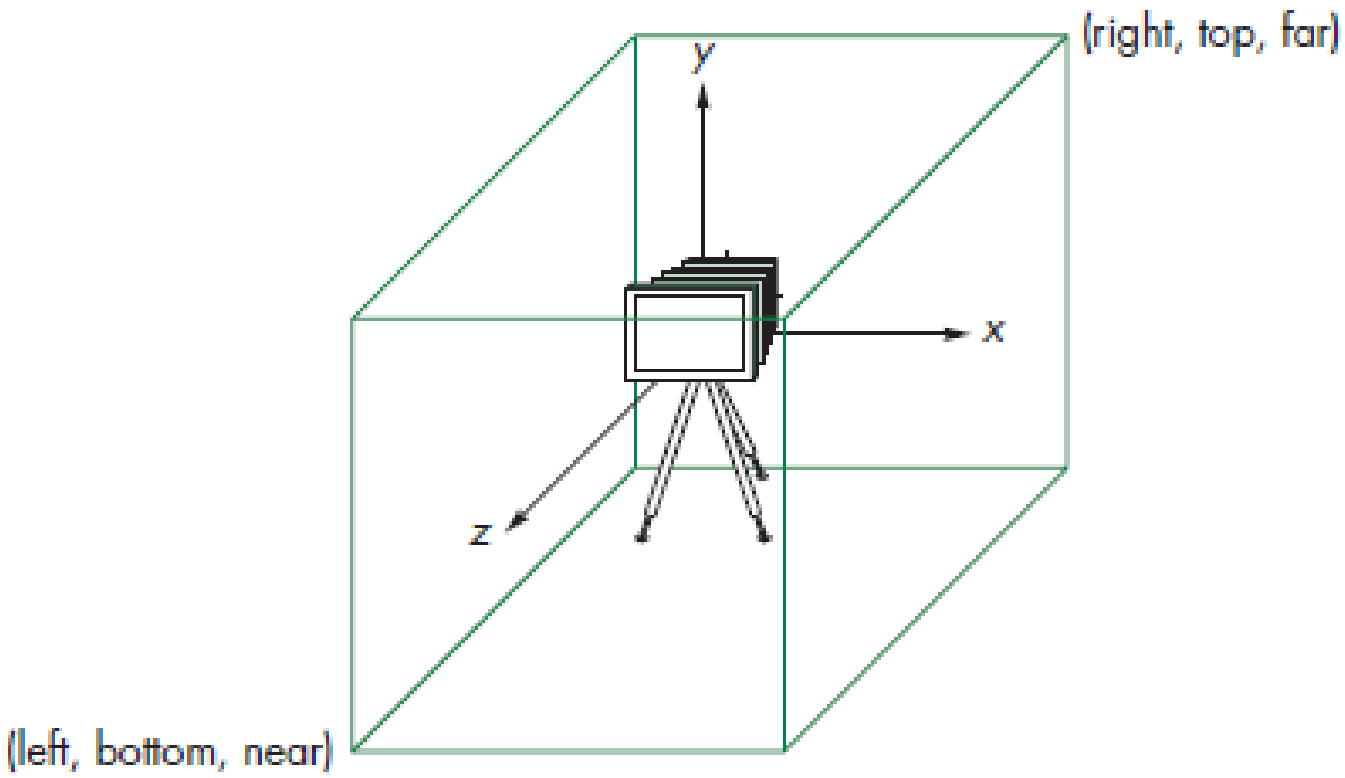


Orthographic View

- Suppose we start with projectors that are parallel to the positive *z*-axis and the projection plane at *z=0*, we can slide the projection plane along the z-axis without changing where the projectors intersect this plane.



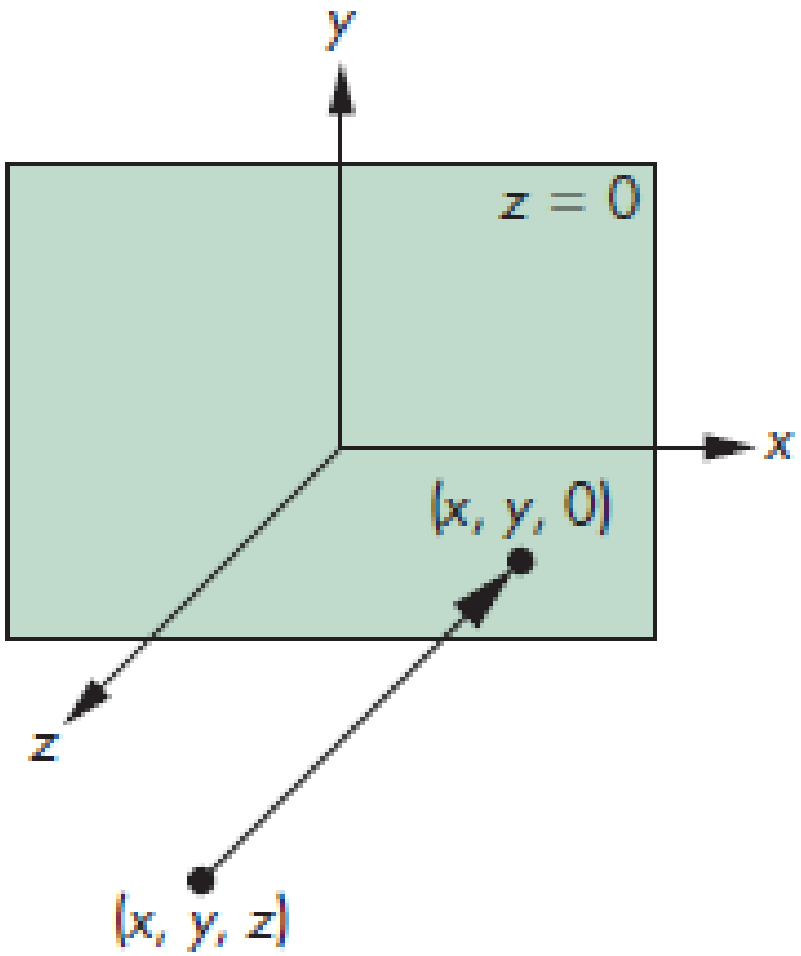
Orthographic View Volume



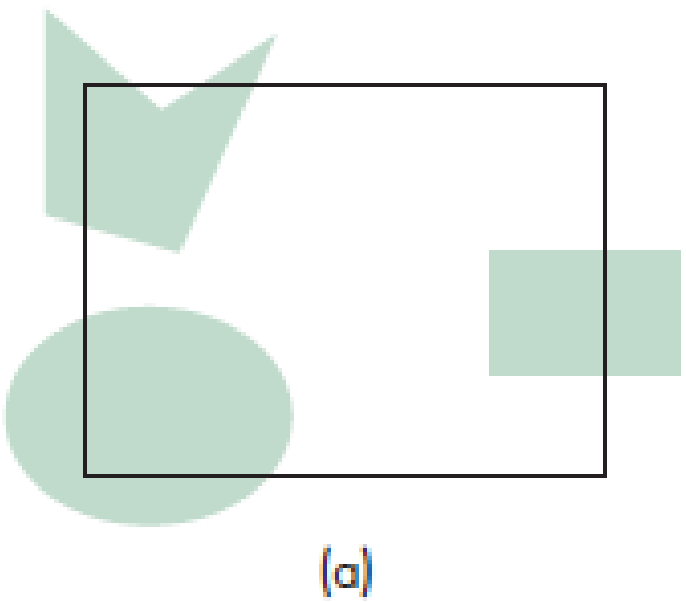
In WebGL, an orthographic projection with a right-parallelpiped viewing volume is the default. The volume is the cube defined by the planes $x = \pm 1 \quad y = \pm 1 \quad z = \pm 1$.

Orthographic Projection

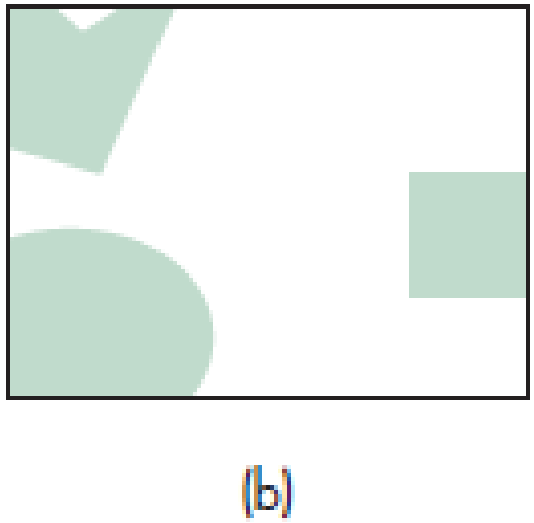
- The orthographic projection “sees” the objects in the volume specified by the viewing volume.
- Unlike a real camera, the orthographic projection can include objects behind the camera.



2D Viewing



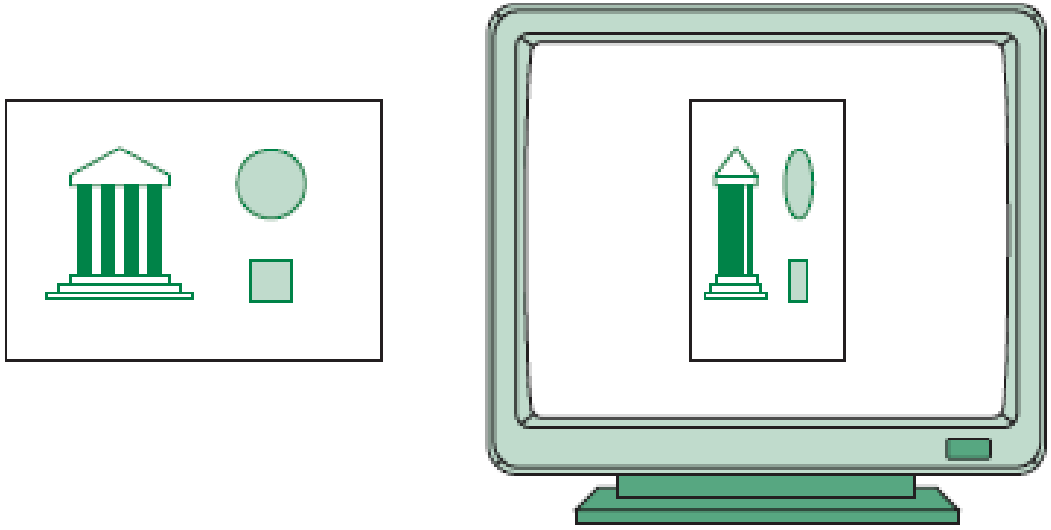
Before clipping



After clipping

Aspect Ratio and Viewports

- The aspect ratio of a rectangle is the ratio of the rectangle's width and height.



Aspect ratio mismatch: (a) viewing rectangle (b) display window

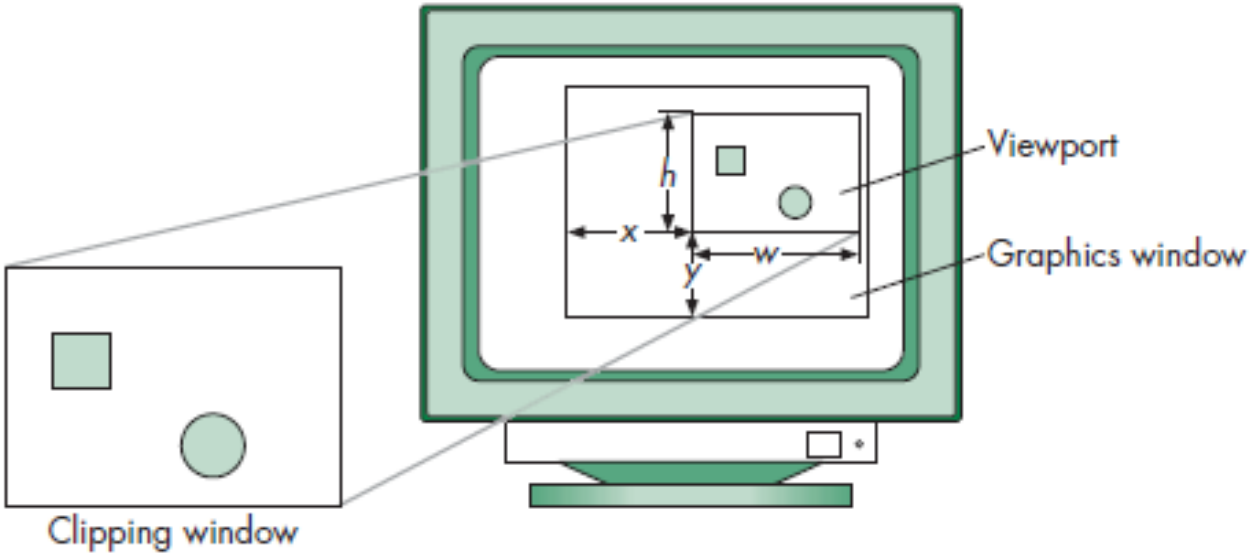
Aspect Ratio and Viewports

- Solution: use the concept of a viewport.
- A viewport is a rectangle area of the display window. By default, it is the entire window, but it can be set to any smaller size in pixels via the function

```
gl.viewport(x, y, w, h);
```

where *(x,y)* is the lower-left corner of the viewport, and *w* and *h* give the width and height, respectively.

Aspect Ratio and Viewports



- The values should be in pixels. For a given window, we can adjust the height and width of the viewport to match the aspect ratio of the clipping rectangle, thus preventing object distortion in the image.

Polygons and Recursion

Polygons and Recursion

Polygons and Recursion

- If we run the gasket program with many iterations, then the randomness in the image would disappear.
- No matter how many points we generate, there are no points in the middle.
- If we draw line segments connecting the midpoints of the sides of the original triangle, then we divide the original triangle into four triangles and the middle one contains no points.

Polygons and Recursion

- Look at the other three triangles, we see that we can apply the same to each of them.
- This structure suggests a second generation method of Sierpinski gasket by using polygons instead of points and does not require the use of a random number generator.
- Strategy: Start with a single triangle, subdivide it into four smaller triangles by bisecting the sides, and then remove the middle triangle from further consideration.

Polygons and Recursion

- Define a triangle

```
function triangle(a, b, c)
{
  points.push(a);
  points.push(b);
  points.push(c);
}
```

- Find the midpoints of the sides

```
var ab = mix(a, b, 0.5);
var ac = mix(a, c, 0.5);
var bc = mix(b, c, 0.5);
```

Polygons and Recursion

- Divide

```
divideTriangle(a, b, c, count);
```

```
function divideTriangle(a, b, c, count)
{
  if (count == 0) {
    triangle(a, b, c);
  }
  else {
    var ab = mix(0.5, a, b);
    var ac = mix(0.5, a, c);
    var bc = mix(0.5, b, c);

    --count;

    divideTriangle(a, ab, ac, count);
    divideTriangle(c, ac, bc, count);
    divideTriangle(b, bc, ab, count);
  }
}
```

Drawing Multiple Points

- For shapes that use multiple vertices, you need a way to simultaneously pass multiple vertices to the vertex shader so that you can draw shapes constructed from multiple vertices, such as triangles, rectangles, and cubes.
- WebGL provides a convenient way to pass multiple vertices and uses something called a **buffer object** to do so. A buffer object is a memory area that can store multiple vertices in the WebGL system. It is used both as a staging area for the vertex data and a way to simultaneously pass the vertices to a vertex shader.

Drawing Multiple Points

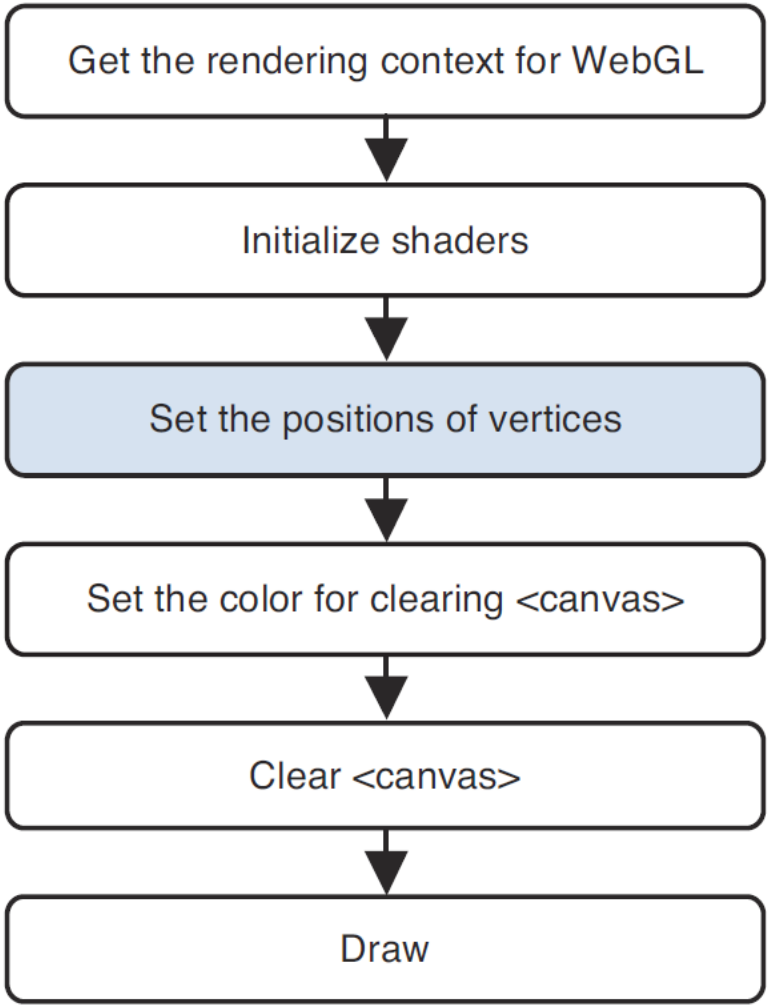


Figure 3.3 Processing flowchart for MultiPoints.js

Drawing Multiple Points

```
// Write the positions of vertices to a vertex shader
 var n = initVertexBuffers(gl);
 if (n < 0) {
   console.log('Failed to set the positions of the vertices');
   return;
 }
……
// Draw three points
 gl.drawArrays(gl.POINTS, 0, n);
```

Drawing Multiple Points

```
function initVertexBuffers(gl) {
  var vertices = new Float32Array([ 0.0, 0.5,   -0.5, -0.5,   0.5, -0.5 ]);
  var n = 3; // The number of vertices

  // Create a buffer object
  var vertexBuffer = gl.createBuffer();
  if (!vertexBuffer) {
    console.log('Failed to create the buffer object');
    return -1;
  }

  // Bind the buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  // Write date into the buffer object
  gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
……
  // Assign the buffer object to a_Position variable
  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);

  // Enable the assignment to a_Position variable
  gl.enableVertexAttribArray(a_Position);

  return n;
}
```

Drawing Multiple Points

- A buffer object is a mechanism provided by the WebGL system that provides a memory area allocated in the system that holds the vertices you want to draw.
- By creating a buffer object and then writing the vertices to the object, you can pass multiple vertices to a vertex shader through one of its attribute variables.

Drawing Multiple Points

- There are five steps needed to pass multiple data values to a vertex shader through a buffer object.
  1. Create a buffer object (*gl.createBuffer()*).
  2. Bind the buffer object to a target (*gl.bindBuffer()*).
  3. Write data into the buffer object (*gl.bufferData()*).
  4. Assign the buffer object to an attribute variable (*gl.vertexAttribPointer()*).
  5. Enable assignment (*gl.enableVertexAttribArray()*).

Drawing Multiple Points

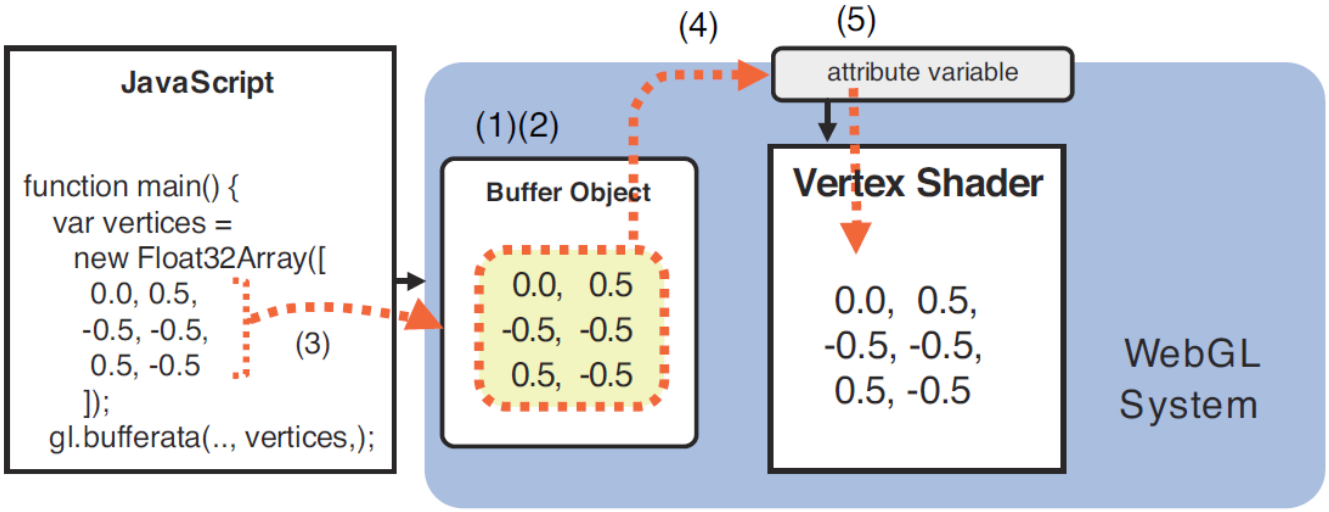


Figure 3.5 The five steps to pass multiple data values to a vertex shader using a buffer object

Drawing Multiple Points

- Create a Buffer Object (*gl.createBuffer()*)

```
// Create a buffer object
 var vertexBuffer = gl.createBuffer();
 if (!vertexBuffer) {
   console.log('Failed to create the buffer
object');
   return -1;
 }
```

Drawing Multiple Points

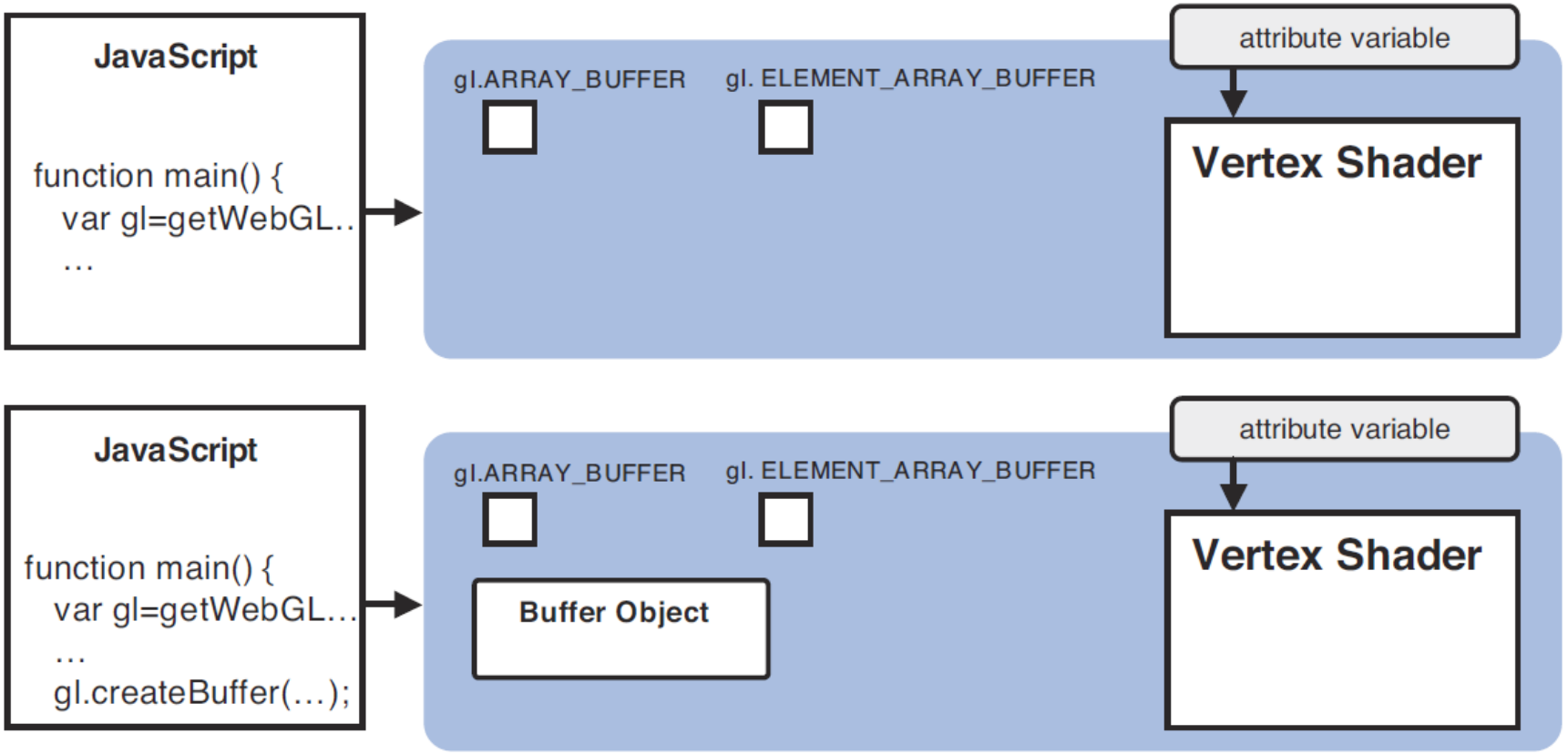


Figure 3.6 Create a buffer object

Drawing Multiple Points

- After creating a buffer object, the second step is to bind it to a “target.” The target tells WebGL what type of data the buffer object contains, allowing it to deal with the contents correctly.

Drawing Multiple Points

- Bind a Buffer Object to a Target (*gl.bindBuffer()*)

// Bind the buffer object to target
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

Drawing Multiple Points

`gl.bindBuffer(target, buffer)`

Enable the buffer object specified by *buffer* and bind it to the *target*.

Parameters Target can be one of the following:

`gl.ARRAY_BUFFER`	Specifies that the buffer object contains vertex data.
`gl.ELEMENT_ARRAY_BUFFER`	Specifies that the buffer object contains index values pointing to vertex data. (See Chapter 6, "The OpenGL ES Shading Language [GLSL ES].)
buffer	Specifies the buffer object created by a previous call to `gl.createBuffer()`. When `null` is specified, binding to the *target* is disabled.

Return Value None

Errors

INVALID_ENUM	*target* is none of the above values. In this case, the current binding is maintained.

Drawing Multiple Points

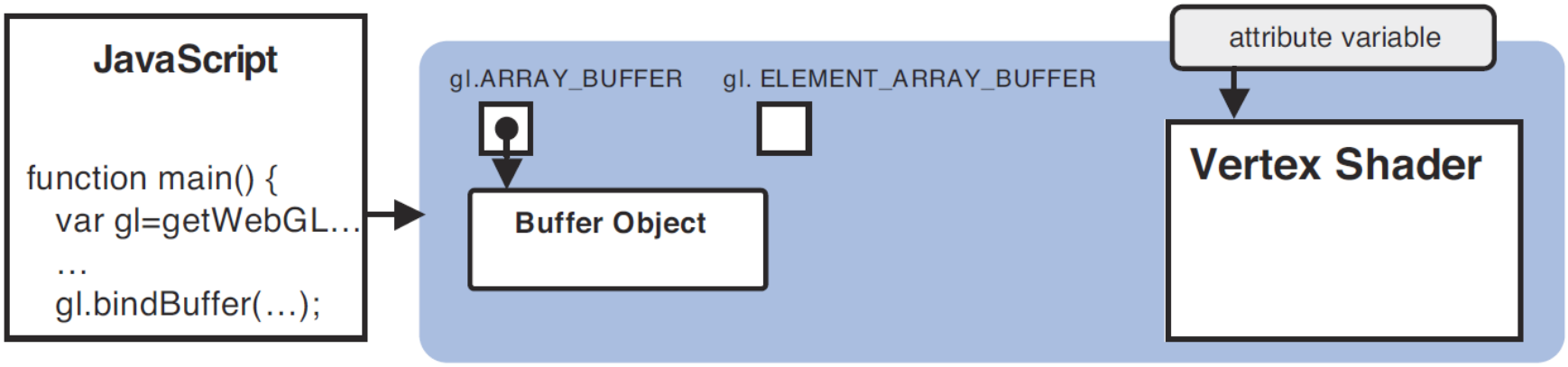


Figure 3.7 Bind a buffer object to a target

Drawing Multiple Points

- Write Data into a Buffer Object (*gl.bufferData())*

// Write date into the buffer object
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

Drawing Multiple Points

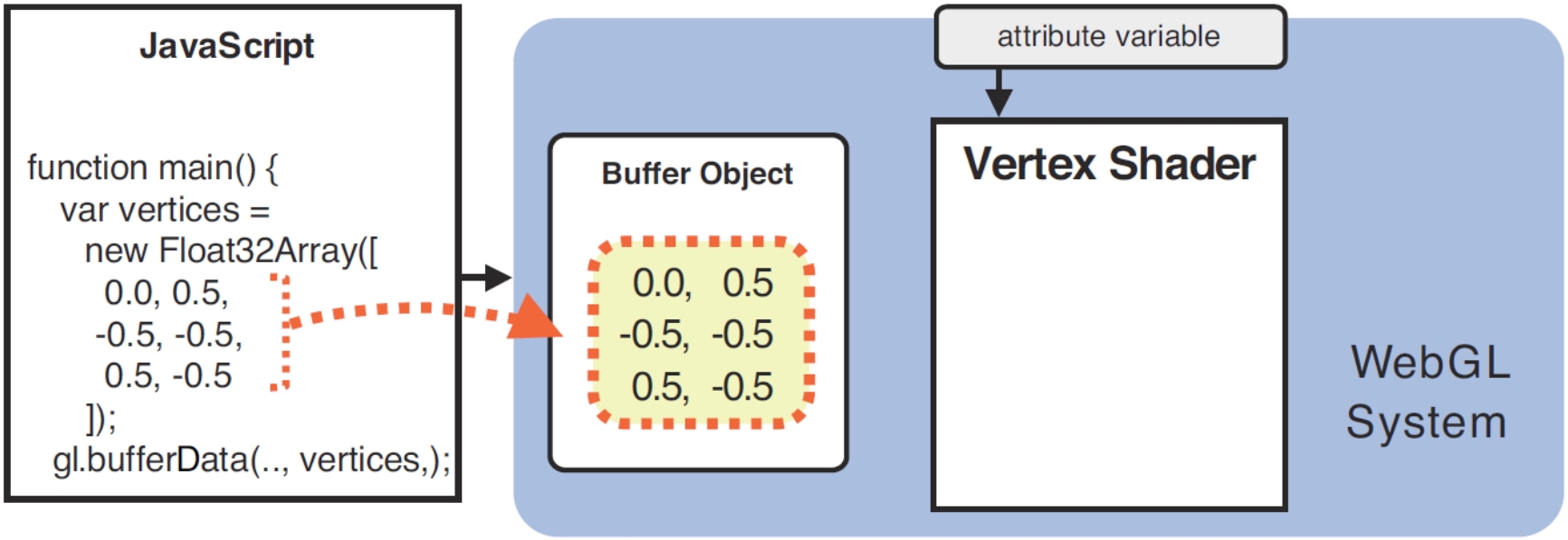


Figure 3.8 Allocate storage and write data into a buffer object

Drawing Multiple Points

`gl.bufferData(target, data, usage)`

Allocate storage and write the data specified by *data* to the buffer object bound to *target*.

Parameters	target	Specifies `gl.ARRAY_BUFFER` or `gl.ELEMENT_ARRAY_BUFFER`.
	data	Specifies the data to be written to the buffer object (typed array; see the next section).
	usage	Specifies a hint about how the program is going to use the data stored in the buffer object. This hint helps WebGL optimize performance but will not stop your program from working if you get it wrong.
		`gl.STATIC_DRAW` — The buffer object data will be specified once and used many times to draw shapes.
		`gl.STREAM_DRAW` — The buffer object data will be specified once and used a few times to draw shapes.
		`gl.DYNAMIC_DRAW` — The buffer object data will be specified repeatedly and used many times to draw shapes.
Return value	None	
Errors	INVALID_ENUM	*target* is none of the preceding constants

Drawing Multiple Points

- **Typed Arrays**

WebGL often deals with large quantities of data of the same type, such as vertex coordinates and colors, for drawing 3D objects. For optimization purposes, a special type of array (typed array) has been introduced for each data type. Because the type of data in the array is known in advance, it can be handled efficiently.

Drawing Multiple Points

Table 3.1 Typed Arrays Used in WebGL

Typed Array	Number of Bytes per Element	Description (C Types)
`Int8Array`	1	8-bit signed integer (signed char)
`Uint8Array`	1	8-bit unsigned integer (unsigned char)
`Int16Array`	2	16-bit signed integer (signed short)
`Uint16Array`	2	16-bit unsigned integer (unsigned short)
`Int32Array`	4	32-bit signed integer (signed int)
`Uint32Array`	4	32-bit unsigned integer (unsigned int)
`Float32Array`	4	32-bit floating point number (float)
`Float64Array`	8	64-bit floating point number (double)

Drawing Multiple Points

Table 3.2 Methods, Property, Constant of Typed Arrays

Methods, Properties, and Constants	Description
`get(index)`	Get the *index*-th element
`set(index, value)`	Set *value* to the *index*-th element
`set(array, offset)`	Set the elements of *array* from *offset*-th element
`length`	The length of the array
`BYTES_PER_ELEMENT`	The number of bytes per element in the array

- Like standard arrays, the *new* operator creates a typed array and is passed the array data.
- Unlike the *Array* object, the [] operator is not supported.
- An empty typed array can be created by specifying the number of elements of the array as an argument. For example:

var vertices = new Float32Array(4);

Drawing Multiple Points

- Assign the Buffer Object to an Attribute Variable (*gl.vertexAttribPointer()*)

// Assign the buffer object to a_Position variable

gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);

Drawing Multiple Points

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

Assign the buffer object bound to `gl.ARRAY_BUFFER` to the attribute variable specified by *location*.

Parameters

location	Specifies the storage location of an attribute variable.
size	Specifies the number of components per vertex in the buffer object (valid values are 1 to 4). If *size* is less than the number of components required by the attribute variable, the missing components are automatically supplied just like `gl.vertexAttrib[1234]f()`. For example, if *size* is 1, the second and third components will be set to 0, and the fourth component will be set to 1.
type	Specifies the data format using one of the following:

`gl.UNSIGNED_BYTE`	unsigned byte	for `Uint8Array`
`gl.SHORT`	signed short integer	for `Int16Array`
`gl.UNSIGNED_SHORT`	unsigned short integer	for `Uint16Array`
`gl.INT`	signed integer	for `Int32Array`
`gl.UNSIGNED_INT`	unsigned integer	for `Uint32Array`
`gl.FLOAT`	floating point number	for `Float32Array`

normalized	Either `true` or `false` to indicate whether nonfloating data should be normalized to [0, 1] or [–1, 1].
stride	Specifies the number of bytes between different vertex data elements, or zero for default stride (see Chapter 4).
offset	Specifies the offset (in bytes) in a buffer object to indicate what number-th byte the vertex data is stored from. If the data is stored from the beginning, *offset* is 0.

Return value None

Errors

INVALID_OPERATION	There is no current program object.
INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8, by default). *stride* or *offset* is a negative value.

Drawing Multiple Points

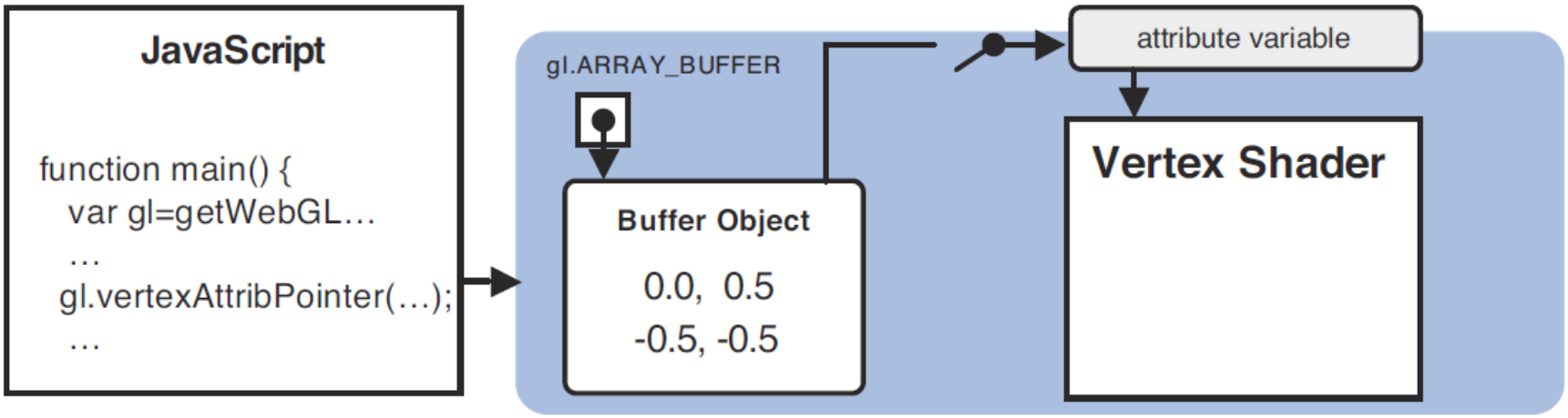


Figure 3.9 Assign a buffer object to an attribute variable

Drawing Multiple Points

The fifth and final step is to enable the assignment of the buffer object to the attribute variable.

- Enable the Assignment to an Attribute Variable (*gl.enableVertexAttribArray()*)

// Enable the assignment to a_Position variable

gl.enableVertexAttribArray(a_Position);

Drawing Multiple Points

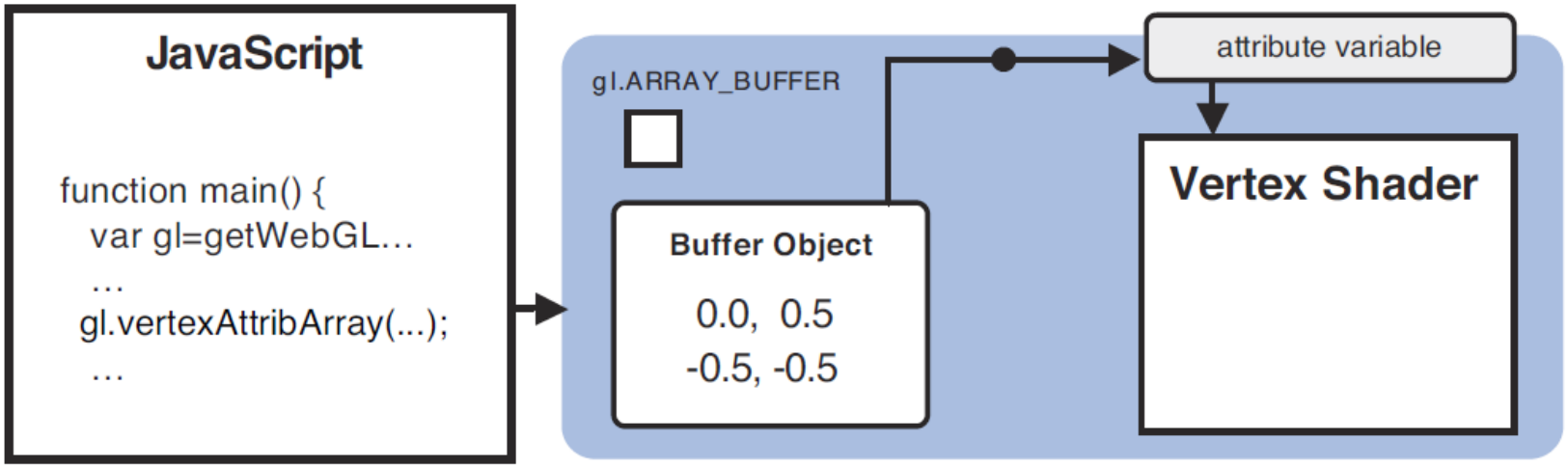


Figure 3.10 Enable the assignment of a buffer object to an attribute variable