Computer Graphics

Minho Kim

Dept. of Computer Science

University of Seoul

Chapter 3: Drawing and Transforming Triangles

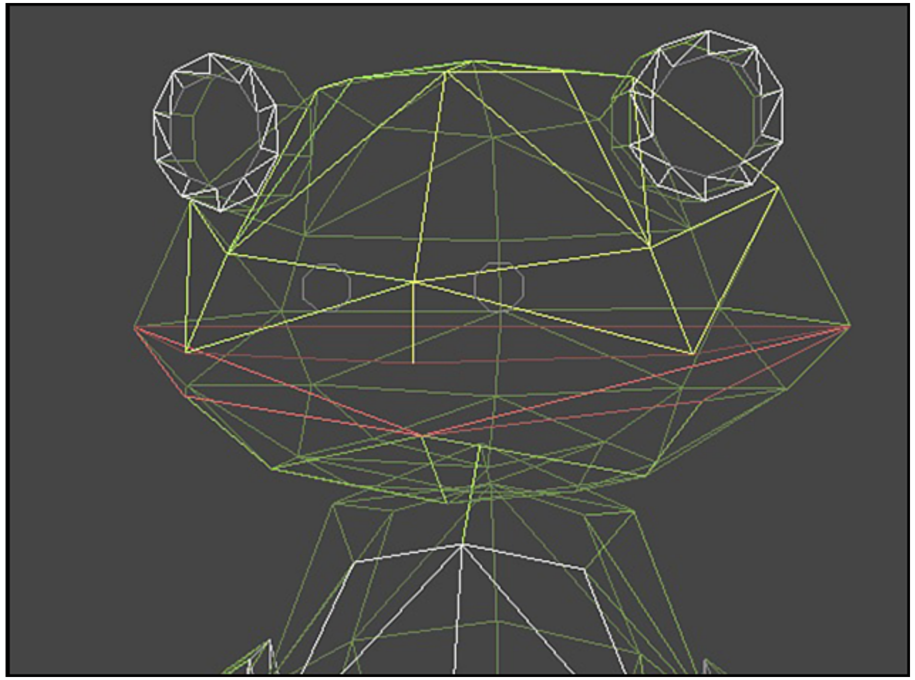
What to Learn

- The critical role of **triangles** in 3DCG and WebGL's support for drawing triangles
- Using multiple triangles to draw other basic shapes
- Basic **transformation** that move, rotate, and scale triangles using simple equations
- How **matrix operations** make transformations simple

- Examples can be found at https://sites.google.com/site/webglbook/home/chapter-3

Drawing Multiple Points

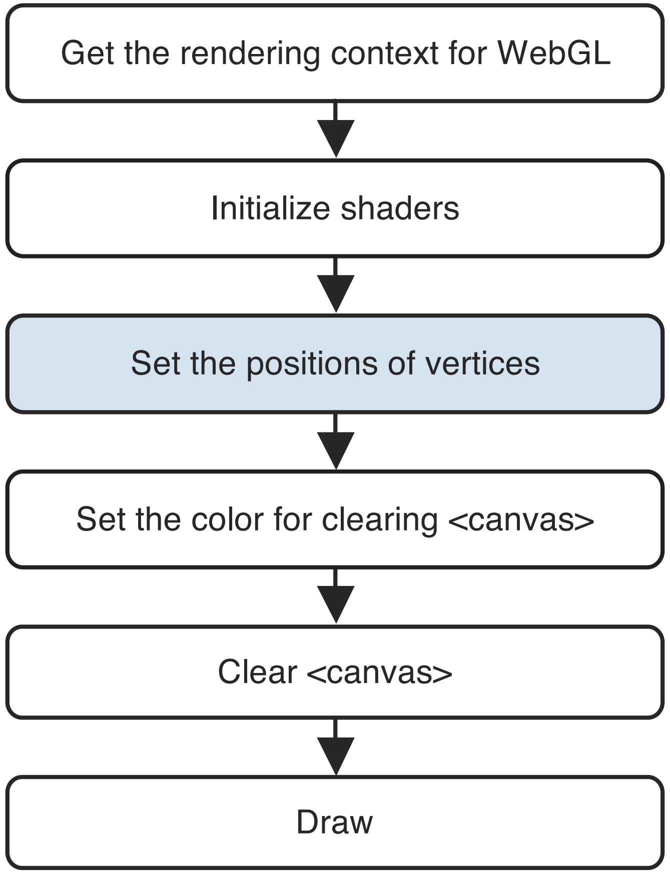
- 3DCG is all about drawing millions of triangles fast. Really, really fast.
- Multiple triangles are defined by multiple vertices (points)
  → How to pass multiple vertices to WebGL with low overhead?



Example #1: `Multipoint`

Example #1: `Multipoint`

- http://rodger.global-linguist.com/webgl/ch03/MultiPoint.html
- Draws three red points on the screen
- We need to call `gl.drawArrays()` only once thanks to “buffer object” – All vertex shaders in Chapter 2 do not run in parallel. (one vertex at a time)
- What to learn
    - How to use a “buffer object” to pass data (attributes) for multiple vertices to the vertex shader at once



Get the rendering context for WebGL
Initialize shaders
Set the positions of vertices
Set the color for clearing <canvas>
Clear <canvas>
Draw

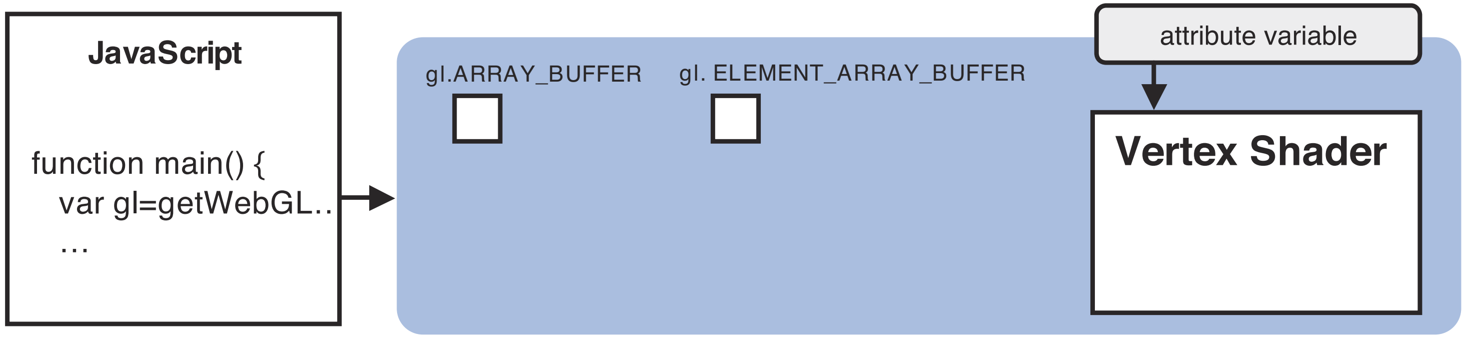
Using Buffer Objects

- A mechanism provided by the WebGL system that provides a memory area allocated in the system that holds the vertices you want to draw
- Enables us to pass multiple vertices to a vertex shader through one of its attribute variables.
- Procedure
  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()`)

1. Create a Buffer Object (gl.createBuffer())

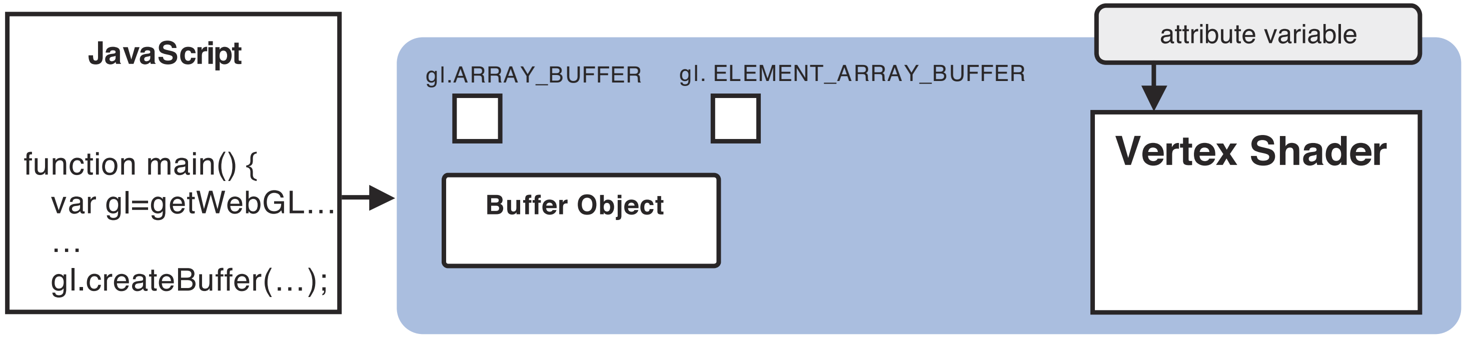
- A buffer object is created and stored in the WebGL system
  - Refer to OpenGL objects
- Can be deleted (released) by calling gl.deleteBuffer()

before gl.createBuffer()



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

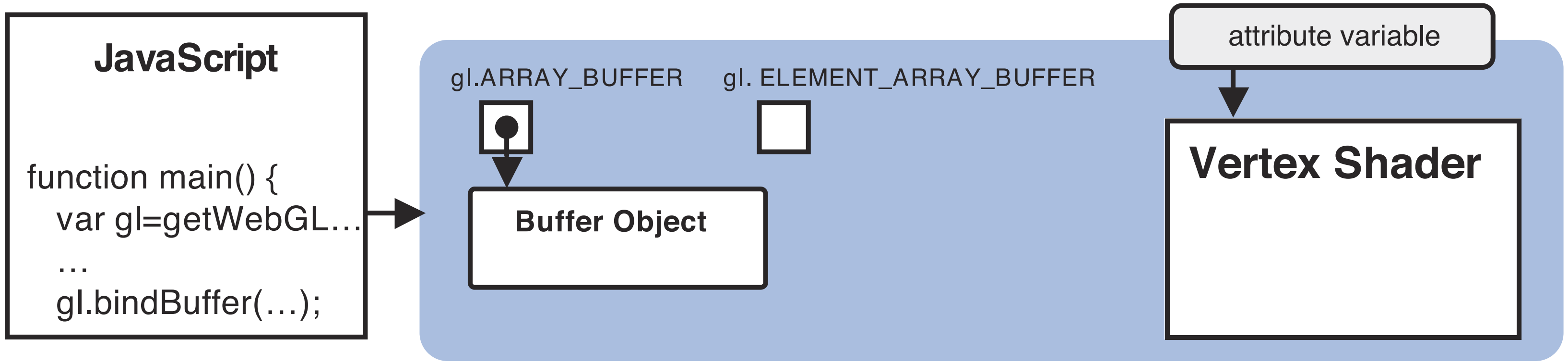
after gl.createBuffer()



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

2. Bind a Buffer Object to a Target (`gl.bindBuffer()`)

- To manipulate the buffer object just created, we need to “bind” it to a **target**
- Target
  - One of the internal states of the WebGL system
  - Either `gl.ARRAY_BUFFER` or `gl.ELEMENT_ARRAY_BUFFER` for `gl.bindBuffer()`



3. Write Data into a Buffer Object (`gl.bufferData()`)

- Allocates storage and writes data to the buffer
- “Writing” by sending data from CPU memory to GPU memory
- Destination (target) buffer should have been specified by `gl.bindBuffer()`
- Data in CPU memory are defined as JavaScript typed arrays, e.g., `Float32Array`
- `usage` parameter for a “hint”
    - For “potential” performance improvement
    - The BO needs not to be used as specified

3. Write Data into a Buffer Object (`gl.bufferData()`) (cont'd)

- Allocates storage and writes data to the buffer
  - Allocation only (without initialization) if `srcData==null`
- "Writing" by sending data from CPU memory to GPU memory
- Destination (target) buffer should have been specified by `gl.bindBuffer()`
- Data in CPU memory are defined as JavaScript typed arrays, e.g., `Float32Array`
- `usage` parameter for a "hint"
  - For "potential" performance improvement
  - The BO needs not to be used as specified
- More in Chapter 5 (p.140)

Typed Arrays

- Standard JavaScript arrays
  - For general-purpose data structure
  - Able to hold both numeric data & strings
  - Not optimized for large quantities of data of the same type
- Typed arrays
  - Arrays with their types known in advance → more efficient
  - `push()` and `pop()` are not supported
  - The only way to create a typed array is by using the `new` operator
  - An empty typed array can be created by specifying the number of elements as an argument, e.g., `var vertices = new Float32Array(4);`

4. Assign the Buffer Object to an Attribute Variable (`gl.vertexAttribPointer()`)

- Currently, the BO data are in the GPU memory. For the WebGL system to "decode" the data in the BO correctly, what kind of information do we need to tell?

4. Assign the Buffer Object to an Attribute Variable (`gl.vertexAttribPointer()`)

- Currently, the BO data are in the GPU memory. For the WebGL system to "decode" the data in the BO correctly, what kind of information do we need to tell?
    - Which attribute do the OB specify?
    - Where do the data begin?
    - What is the type of the data?
    - How many values are specified for each vertex? (e.g., `float`, `vec2`, `vec3`, etc.)
    - How far are the data of adjacent vertices apart in the memory?
- More in Chapter 5 (p.140)

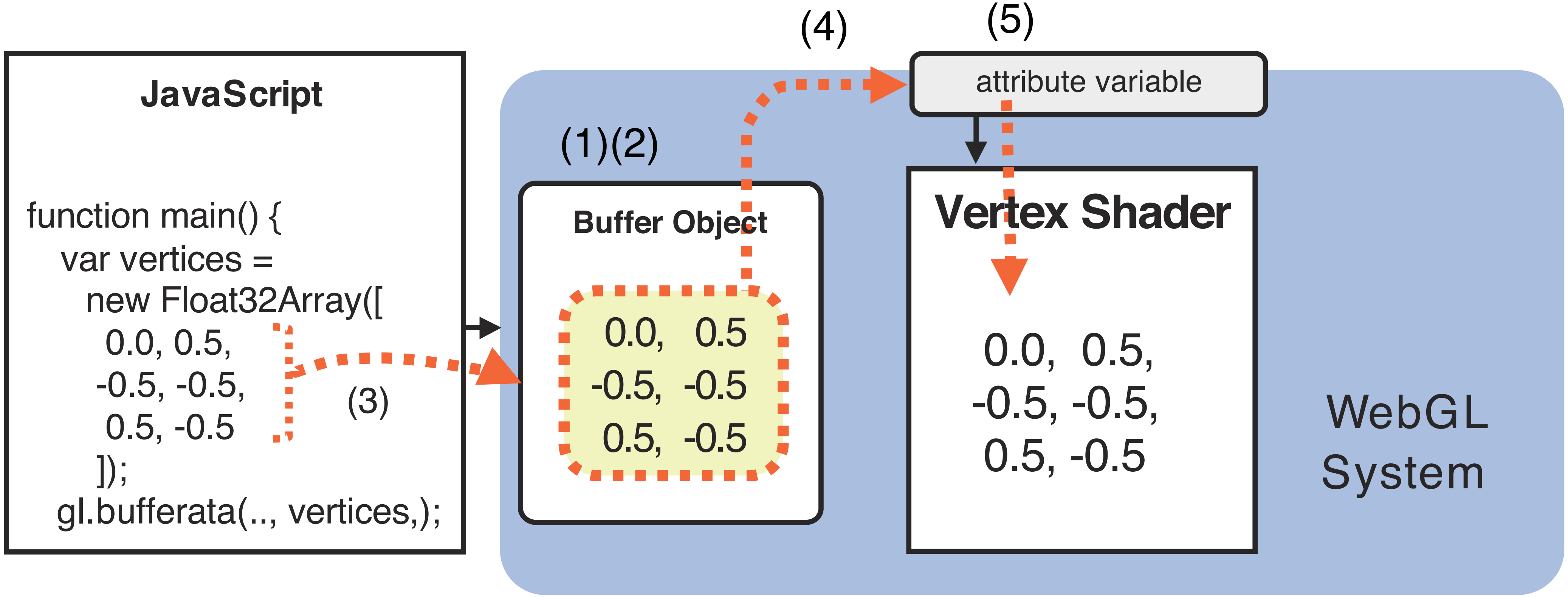
5. Enable the Assignment to an Attribute Variable (`gl.enableVertexAttribArray()`)

- If not called, the “static” vertex attribute values (specified by `gl.vertexAttrib*()` as in Chapter 2) are used instead.
- One of the top rookie mistakes
- Can be disabled by `gl.disableVertexAttribArray()`

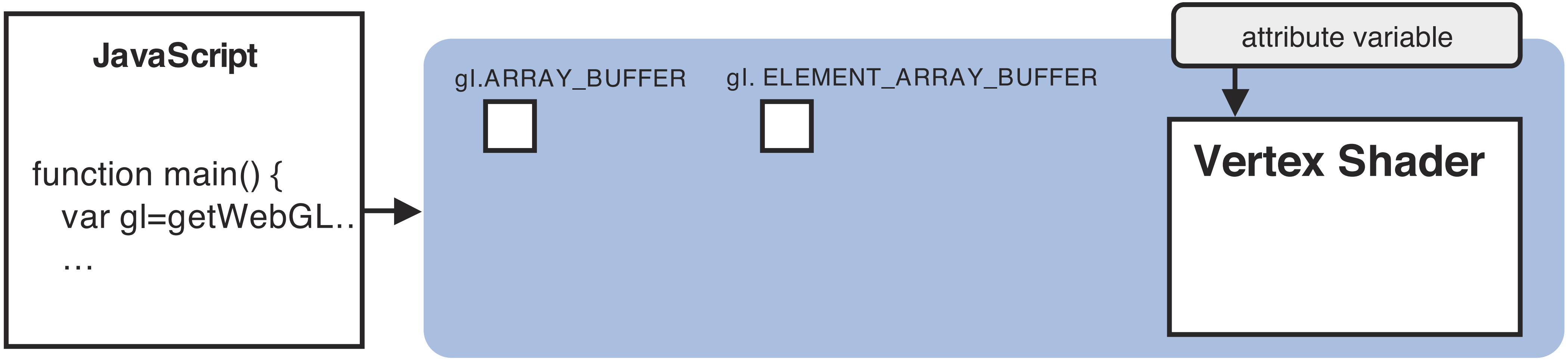
The 2nd and 3rd Parameters of `gl.drawArrays()`

- With `first` and `count` parameters, we can draw only part of the (contiguous) vertices in the BO

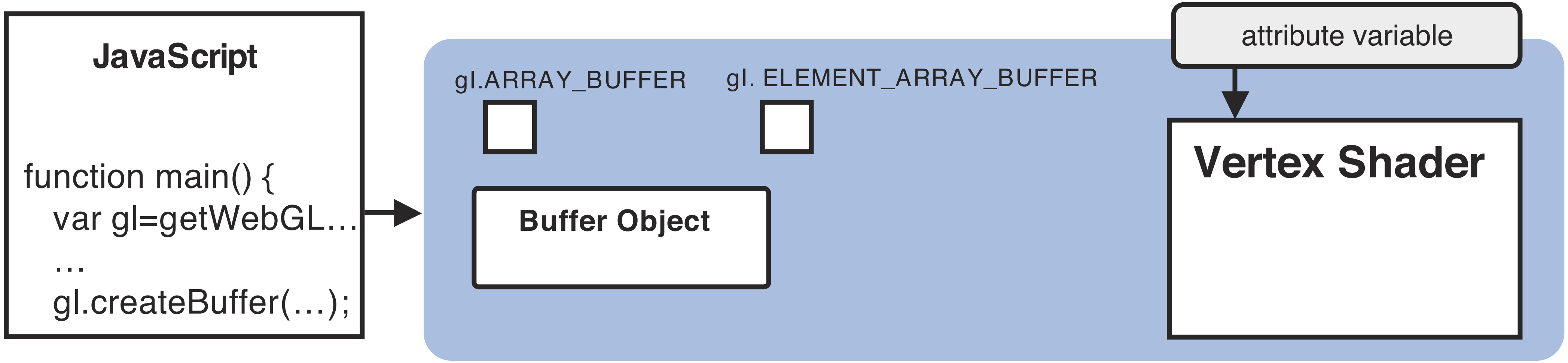
Wrap-Up


JavaScript
function main() {
var vertices =
new Float32Array([
0.0, 0.5,
-0.5, -0.5,
0.5, -0.5
]);
gl.bufferata(.., vertices,);
(3)
(1)(2)
Buffer Object
0.0, 0.5
-0.5, -0.5
0.5, -0.5
(4)
(5)
attribute variable
Vertex Shader
0.0, 0.5,
-0.5, -0.5,
0.5, -0.5
WebGL
System

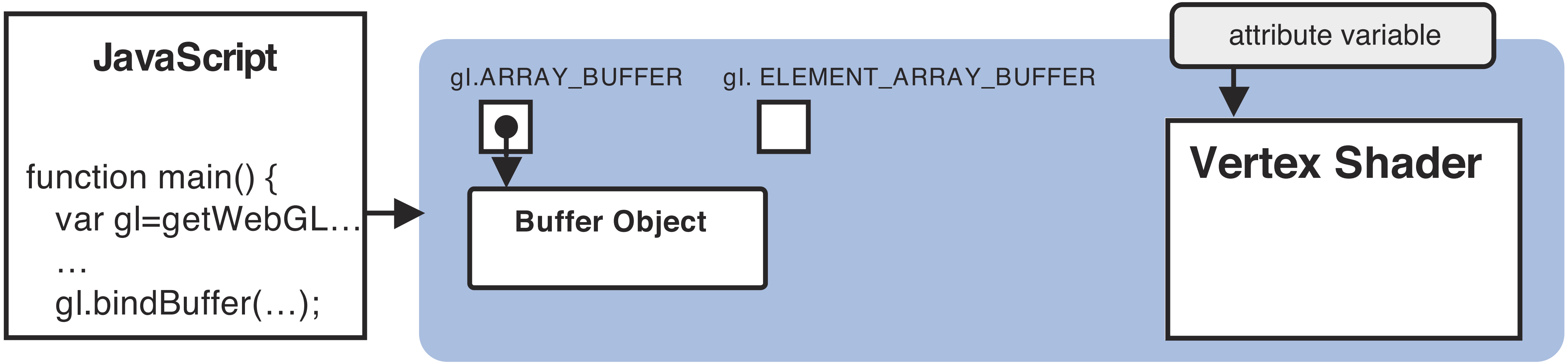
0. Before `gl.createBuffer()`



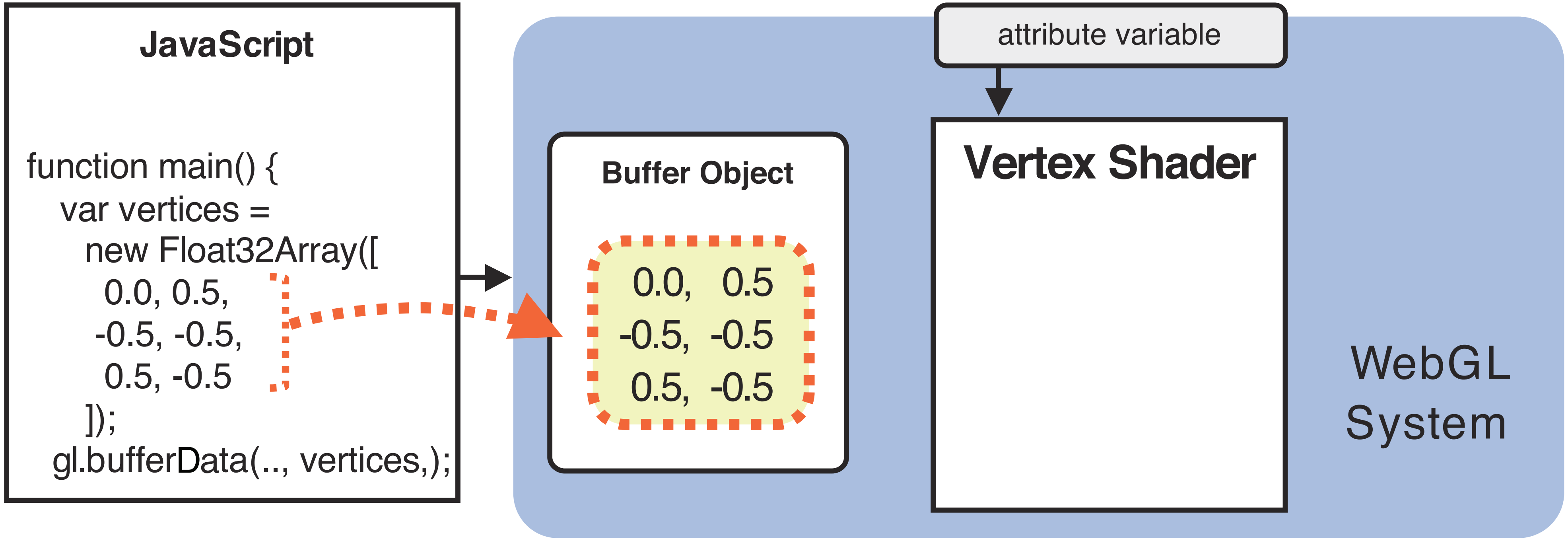
1. Create a Buffer Object (`gl.createBuffer()`)



2. Bind a Buffer Object to a Target (`gl.bindBuffer()`)



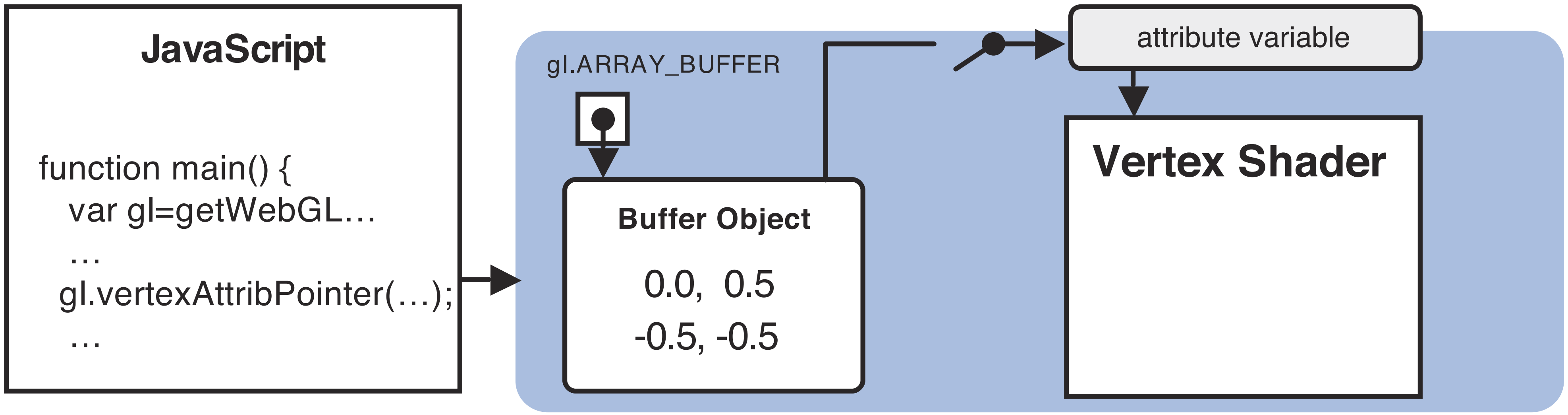
3. Write Data Into a Buffer Object (`gl.bufferData()`)



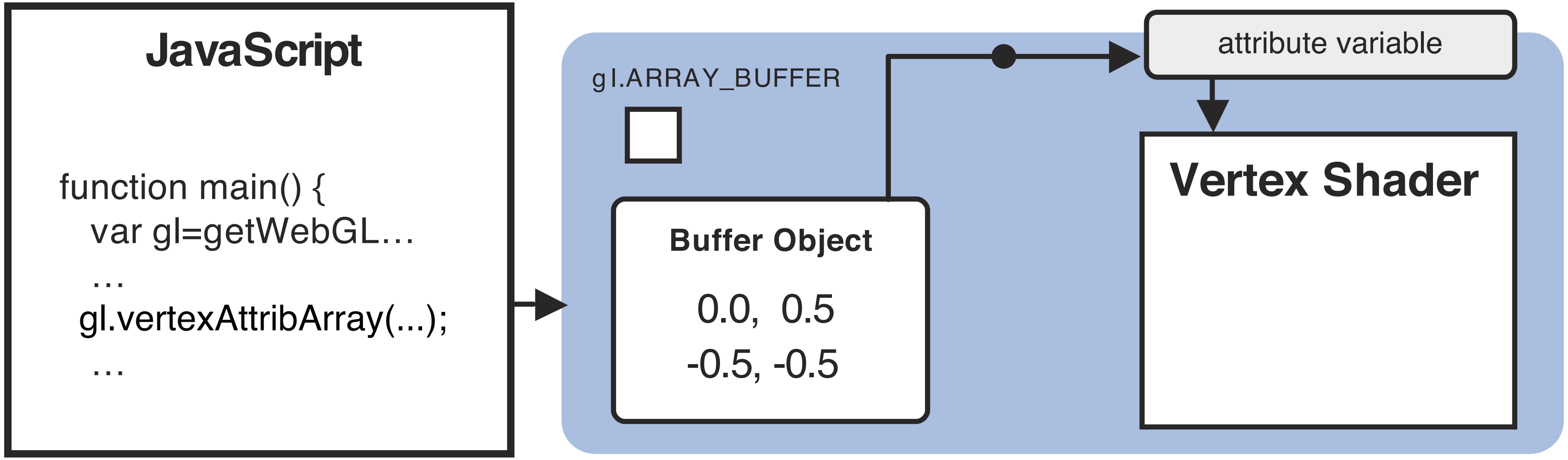
```
function main() {
  var vertices =
    new Float32Array([
      0.0, 0.5,
     -0.5, -0.5,
      0.5, -0.5
    ]);
  gl.bufferData(.., vertices,);
```

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

4. Assign the Buffer Object to an Attribute Variable (`gl.vertexAttribPointer()`)



5. Enable the Assignment to an Attribute Variable (`gl.enableVertexAttribArray()`)



Lab Activities

- Play with the `first` and `count` parameters of `gl.drawArrays()`

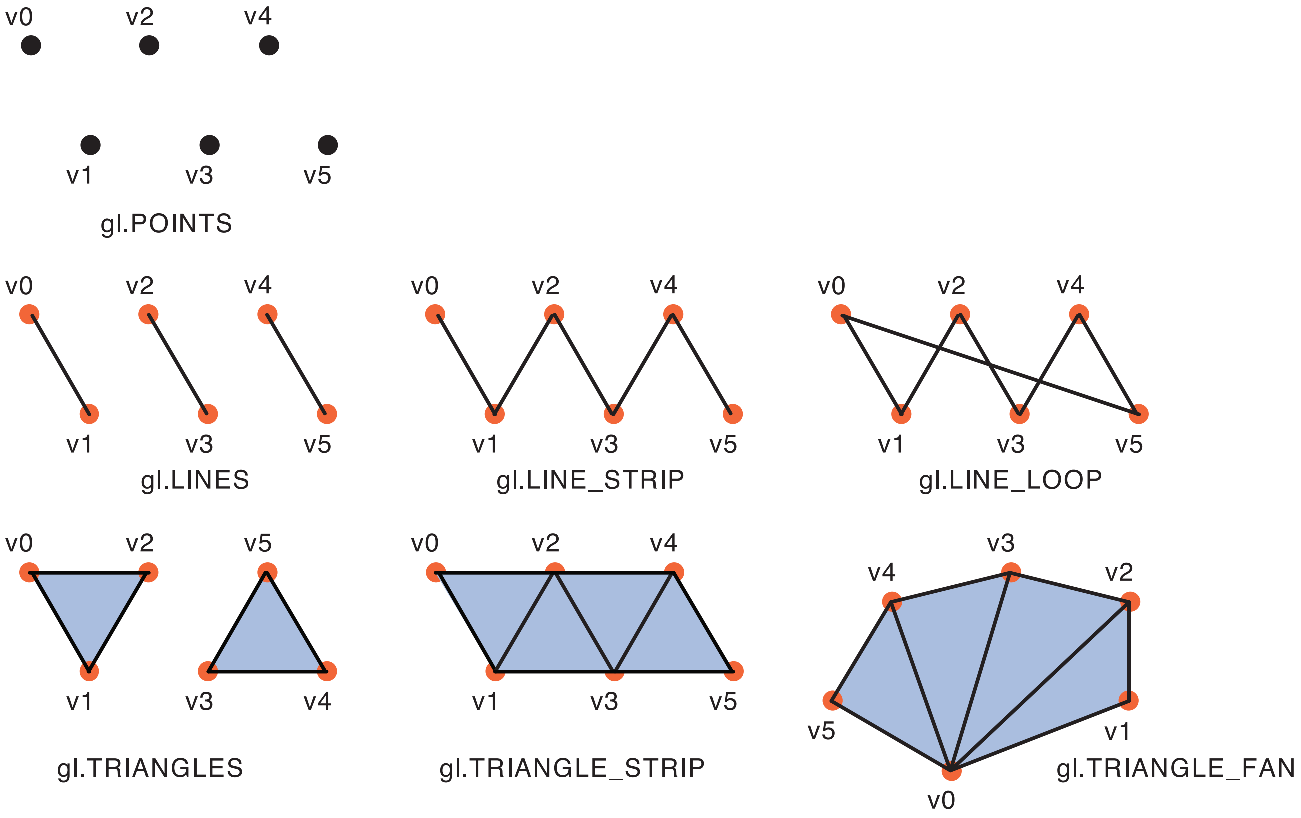
Example #2: `HelloTriangle`

Example #2: `HelloTriangle`

- http://rodger.global-linguist.com/webgl/ch03/HelloTriangle.html
- What to learn
  - How to draw triangles using `gl.drawArrays()`

Basic Shapes

- Seven basic shapes can be drawn using `gl.drawArrays()`



Lab Activities

- Play with the mode parameter of `gl.drawArrays()` to draw various shapes
- http://rodger.global-linguist.com/webgl/ch03/HelloTriangle_LINES.html
- http://rodger.global-linguist.com/webgl/ch03/HelloTriangle_LINE_STRIP.html
- http://rodger.global-linguist.com/webgl/ch03/HelloTriangle_LINE_LOOP.html

Example #3: `HelloQuad`

Example #3: `HelloQuad`

- http://rodger.global-linguist.com/webgl/ch03/HelloQuad.html
- How to draw a quad using triangles?
- You need to be careful with the order of the vertices! – One of the top rookie mistakes
    - http://rodger.global-linguist.com/webgl/ch03/HelloQuad_FAN.html

Lab Activities

- Try to draw a quad using different shapes
  - `gl.TRIANGLE_STRIP` (Example #3)
  - `gl.TRIANGLE_FAN`
  - `gl.TRIANGLES`

Transformations

Translation

- Adding scalar values (offset) to each of the coordinates

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x + \alpha \\ y + \beta \\ z + \gamma \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

- Adding a “vector” to a “point”
- Where should a translation be done? – Host? VS? FS?
- How do we need to pass the “offset”? – attrib var? uniform var?

Example #4: `TranslatedTriangle`

Example #4: `TranslatedTriangle`

- http://rodger.global-linguist.com/webgl/ch03/TranslatedTriangle.html
- A “3D model” is composed of many triangles and we need to transform all the triangles in each frame.
- Where do we need to update the vertex coordinates?
  - Host – The whole BO needs to be updated and uploaded in each frame
  - Vertex shader – A good choice
  - Fragment shader – There are too many fragments. Moreover, we cannot update fragment positions. (We can update z value only)
- What to learn
  - How to translate vertices in a vertex shader

Example #4: `TranslatedTriangle` (cont'd)

- The offset (translation distances)
  - Needs to be passed as a uniform variable
  - The w component should be 0. (What if not?)
  - What if we do not specify the w component?

Example #5:
`RotatedTriangle`

Rotation

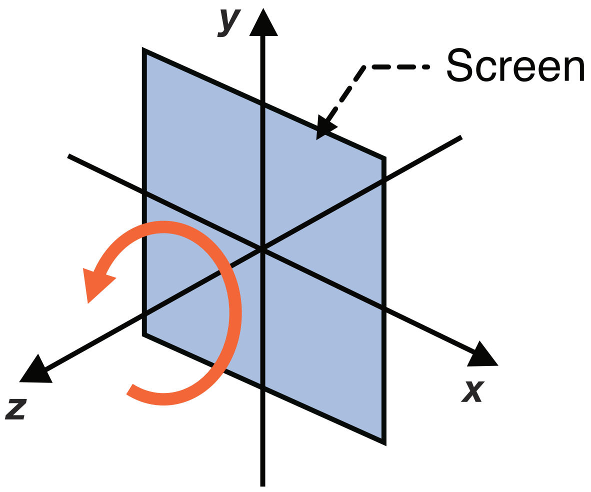
- Defined by
    - Rotation axis – a line through the origin
    - Rotation direction – clockwise or counterclockwise
        - convention: right-hand-rule rotation
    - Rotation angle

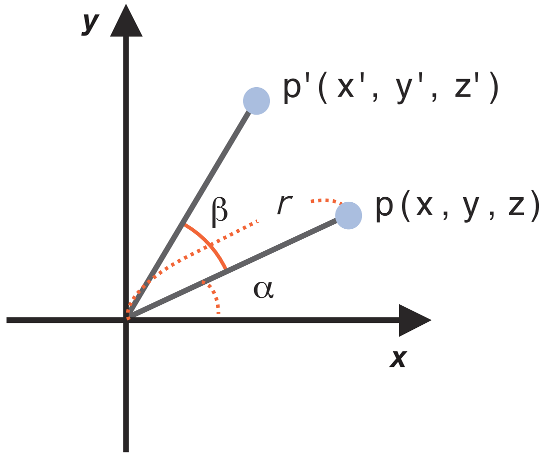
Rotation About the z-axis

- Before rotation: $\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r\cos\alpha \\ r\sin\alpha \\ z \end{bmatrix}$
- After rotation: $\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} r\cos(\alpha+\beta) \\ r\sin(\alpha+\beta) \\ z \end{bmatrix}$

$$= \begin{bmatrix} r(\cos\alpha\cos\beta - \sin\alpha\sin\beta) \\ r(\sin\alpha\cos\beta - \cos\alpha\sin\beta) \\ z \end{bmatrix} = \begin{bmatrix} x\cos\beta - y\sin\beta \\ x\sin\beta - y\cos\beta \\ z \end{bmatrix}$$

- Angle sum and difference identities (trigonometric identities)





Example #5: `RotatedTriangle`

- http://rodger.global-linguist.com/webgl/ch03/RotatedTriangle.html
- What to learn
  - How to compute trigonometric functions in JavaScript
  - How to access each component of `vec*` type variables
- JavaScript built-in object `Math` is required to compute trigonometric functions – Note that the arguments are in radians not degrees!
  - `var radian = Math.PI * ANGLE / 180.0;`
- Where do we need to compute the trigonometric functions? Host? VS?
- To pass multiple `float` values, it's more efficient to pass them as one `vec*` value at once.

Transformation Matrix: Rotation

- Rotation around the z-axis

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \cos\beta - y \sin\beta \\ x \sin\beta - y \cos\beta \\ z \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- In homogeneous coordinates

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 & 0 \\ \sin\beta & \cos\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- More rotation matrices

Transformation Matrix: Translation

- $\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x+\alpha \\ y+\beta \\ z+\gamma \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$
- Cannot be done using a 3×3 matrix!
- Homogeneous coordinate system required

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \alpha \\ 0 & 1 & 0 & \beta \\ 0 & 0 & 1 & \gamma \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+\alpha \\ y+\beta \\ z+\gamma \\ 1 \end{bmatrix}$$

Example #6: `RotatedTriangle_Matrix`

Example #6: `RotatedTriangle_Matrix`

- http://rodger.global-linguist.com/webgl/ch03/RotatedTriangle_Matrix.html
- Rotation as a matrix-vector multiplcation
- What to learn
  - How to compute matrix-vector multiplication in a vertex shader
  - How to use a matrix type in GLSL ES
  - How to pass a matrix as a uniform variable

Rotation About an Arbitrary Axis

- Can be derived using quaternions
- The rotation matrix around $\boldsymbol{u} \coloneqq (\alpha, \beta, \gamma)$, where $\|\boldsymbol{u}\| = 1$, by the angle θ

$$\begin{bmatrix} \alpha^2(1-\cos\theta) \ +\cos\theta & \alpha\beta(1-\cos\theta)-\gamma\sin\theta & \alpha\gamma(1-\cos\theta)+\beta\sin\theta \\ \alpha\beta(1-\cos\theta)+\gamma\sin\theta & \beta^2(1-\cos\theta)+\cos\theta & \beta\gamma(1-\cos\theta)-\alpha\sin\theta \\ \alpha\gamma(1-\cos\theta)-\beta\sin\theta & \beta\gamma(1-\cos\theta)+\alpha\sin\theta & \gamma^2(1-\cos\theta)+\cos\theta \end{bmatrix}$$

Matrices in WebGL

- In GLSL ES, matrix types are supported natively.
  - `mat3`, `mat4`, etc.
- Matrix-matrix and matrix-vector multiplications are supported natively by `*` operator
- Matrices are defined as a JavaScript typed array
- Uniform matrix variable are set by `gl.uniformMatrix*()`
  - Row-major / column-major (default) order → Be careful!
  - **The `transpose` parameter should be false all the time!** (column-major only)

Transformation Matrix: Scaling

- $\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \alpha x \\ \beta y \\ \gamma z \end{bmatrix}$
- In homogeneous coordinates

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Non-uniform scaling is hardly used in practice.