Computer Graphics

Minho Kim

Dept. of Computer Science

University of Seoul

Chapter 5: Using Colors and Texture Images

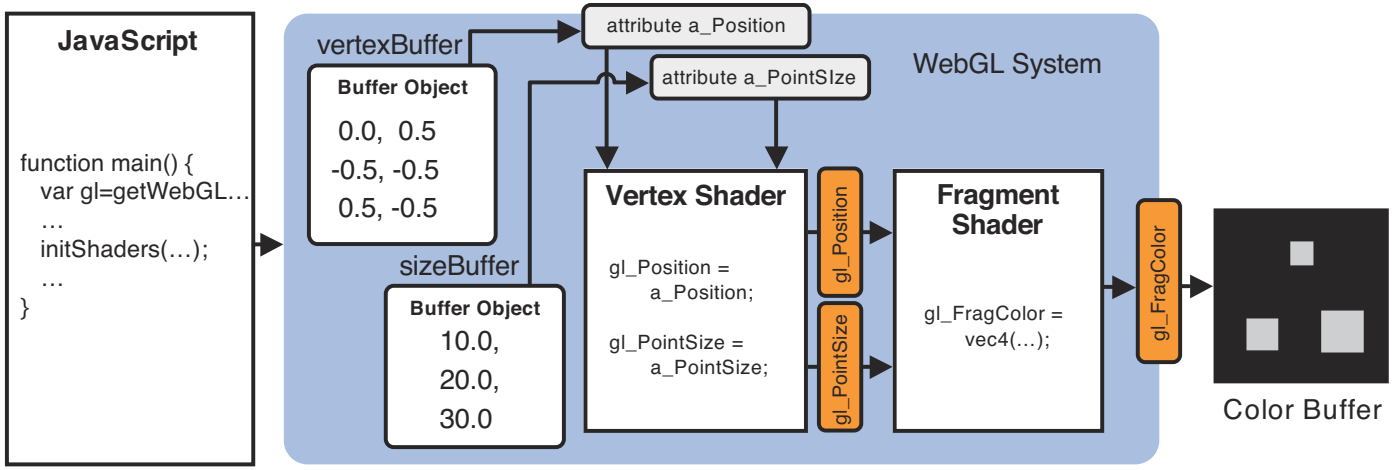
What to Learn

- Passing other data such as color information to the vertex shader
- The conversion from a shape to fragments that takes place between the vertex shader and the fragment shader, which is known as the **rasterization process**
- Mapping images (or textures) onto the surfaces of a shape or object

Example #1: `MultiAttributeSize`

Example #1: `MultiAttributeSize`

- http://rodger.global-linguist.com/webgl/ch05/MultiAttributeSize.html
- Drawing multiple points with different sizes
- What to learn
  - How to pass multiple attributes to the vertex shader
  - How to store multiple attributes in buffer objects
- In this example, a separate buffer object is created for each attribute



The `stride` and `offset` Parameters of `gl.vertexAttribPointer()`

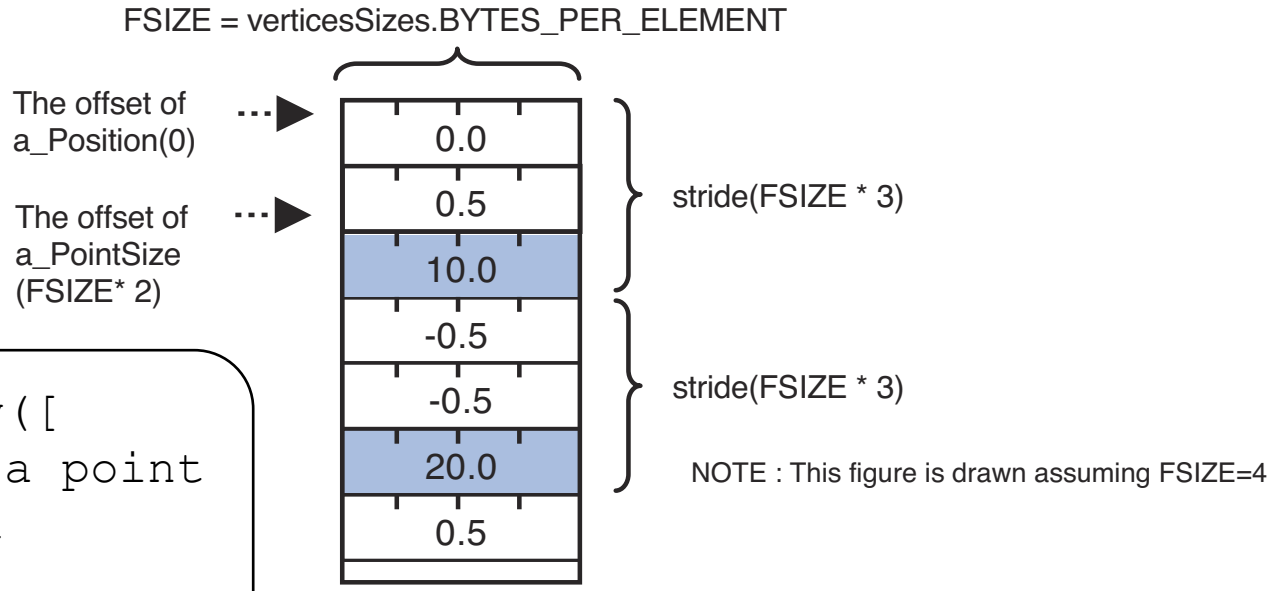
- We can pack several attributes (of the same type) in one buffer object → **interleaving**
- The layout for each attribute needs to be specified by the `stride` and `offset` parameters of `gl.vertexAttribPointer()`
  - `stride`: A `GLsizei` specifying the offset **in bytes** between the beginning of consecutive vertex attributes. Cannot be larger than 255.
  - `offset`: A `GLintptr` specifying an offset **in bytes** of the first component in the vertex attribute array. Must be a multiple of type.

Example #2:
`MultiAttributeSize_Interleaved`

Example #2: `MultiAttributeSize_Interleaved`

- http://rodger.global-linguist.com/webgl/ch05/MultiAttributeSize_Interleaved.html
- Same result as `MultiAttributeSize`, but the attributes are interleaved in one buffer object.
- What to learn
  - How to handle interleaved attributes

Handling Interleaved Attributes




```
var verticesSizes = new Float32Array([
  // Vertex coordinates and size of a point
   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 FSIZE = verticesSizes.BYTES_PER_ELEMENT;
…
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE * 3, 0);
…
gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT, false, FSIZE * 3, FSIZE * 2);
```

Lab Activities

- Set the buffer layout of `MultiAttributeSize_Interleaved` as follows and modify other parts to make it still work.
- “Array of structures” vs. “Structure of arrays”

```
var verticesSizes = new Float32Array([
    0.0,  0.5, // The coords of the 1st point
   -0.5, -0.5, // The coords of the 2nd point
    0.5, -0.5, // The coords of the 3rd point
    10.0,      // The size of the 1st point
    20.0,      // The size of the 2nd point
    30.0       // The size of the 3rd point
]);
```

Example #3: `MultiAttributeColor`

Example #3: `MultiAttributeColor`

- http://rodger.global-linguist.com/webgl/ch05/MultiAttributeColor.html
- Different color for each point
- The fragment color (`gl_FragColor`) can be set only in the fragment shader.
  → How to pass the color data to the fragment shader?
  → As uniform variables? Problem?
- What to learn
  - How to pass data from the vertex shader to the fragment shader

Varying Variables

- Variable types used to pass data from the vertex shader to the fragment shader
- Declared with `varying` keyword
- Only `float`/`vec*`/`mat*` types can be used
- Declared both in the vertex and fragment shader (with identical names and types)
- Drawing points → one vertex is rasterized into multiple fragments → same value for the varying variable is assigned to all the (rasterized) fragments
- **What about a line/triangle with different vertex attributes?**

Lab Activities

- Using the same vertex attributes, draw a triangle instead of three points. What happens?

Example #4: `ColoredTriangle`

Example #4: `ColoredTriangle`

- http://rodger.global-linguist.com/webgl/ch05/ColoredTriangle.html
- A triangle composed of three vertices with different colors
- Bug: In the fragment shader, "`#endif GL_ES`" → "`#endif`"
- What to learn
  - What happens in the rasterization process
  - How the varying variables of three vertices are assigned to the fragments

From the Vertex Shader to the Fragment Shader

- Questions in the app that “draws a triangle in a single color”
  - Who identifies that the vertex coords assigned to `gl_Position` are the vertices of a triangle?
  - Who decides which fragments have to be colored?
  - Who is responsible for invoking the fragment shader and how it handles processing for each of the fragments?

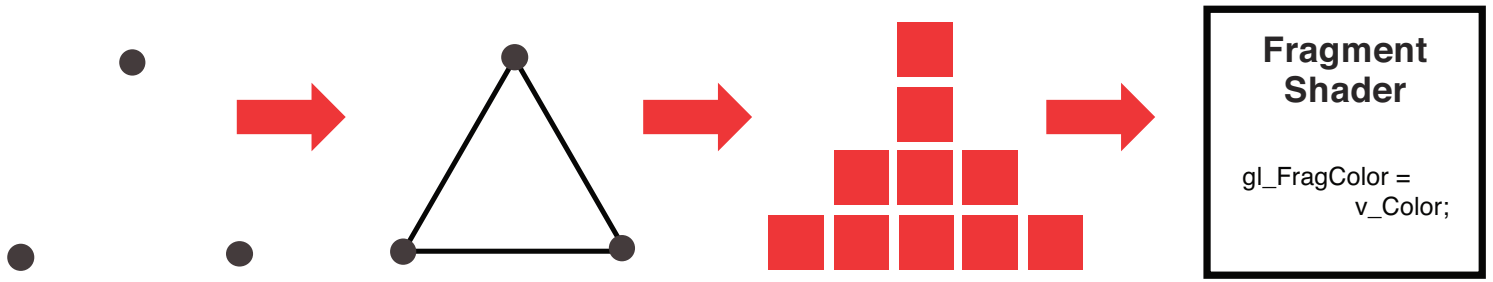
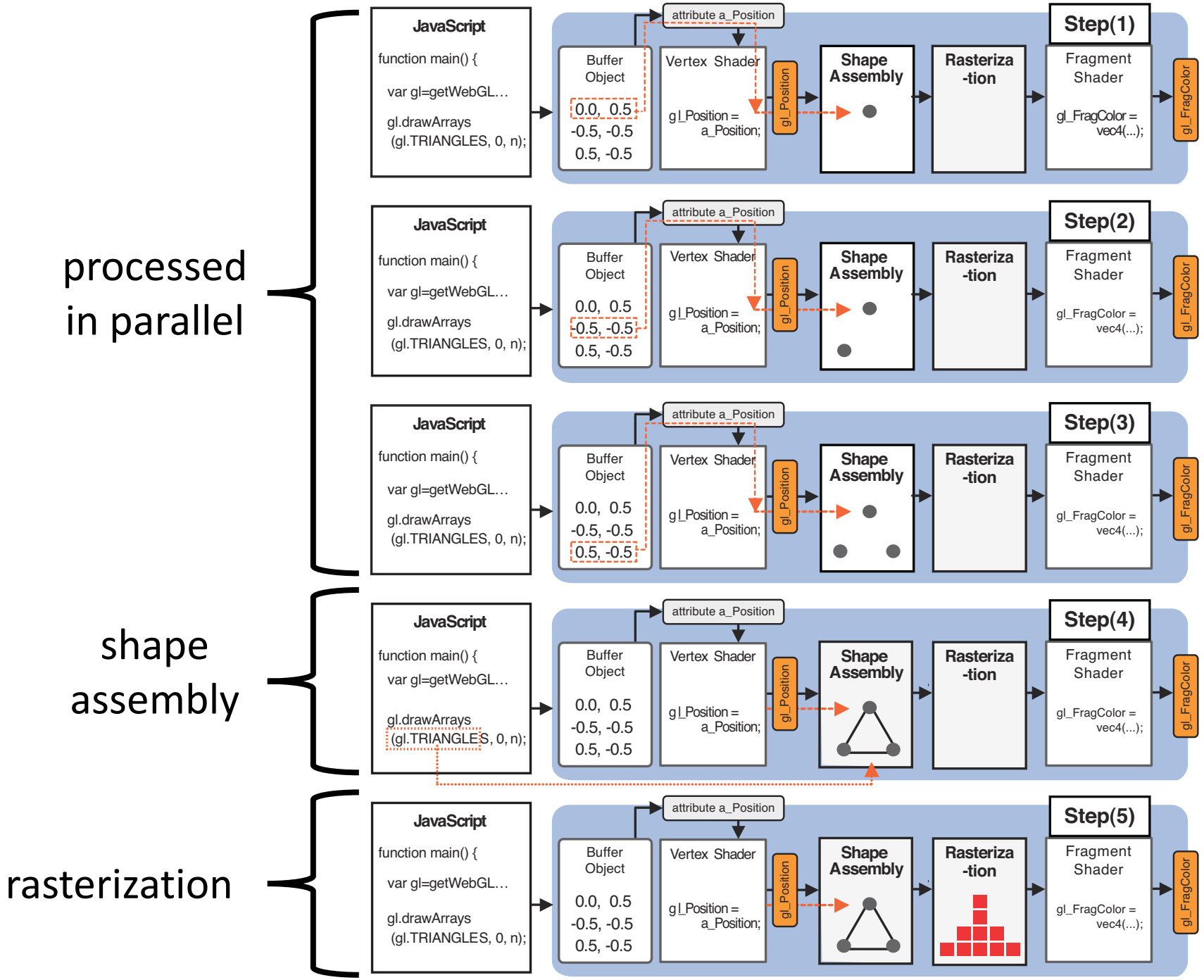


Figure 5.9 Vertex coordinate, identification of a triangle from the vertex coordinates, rasterization, and execution of a fragment shader

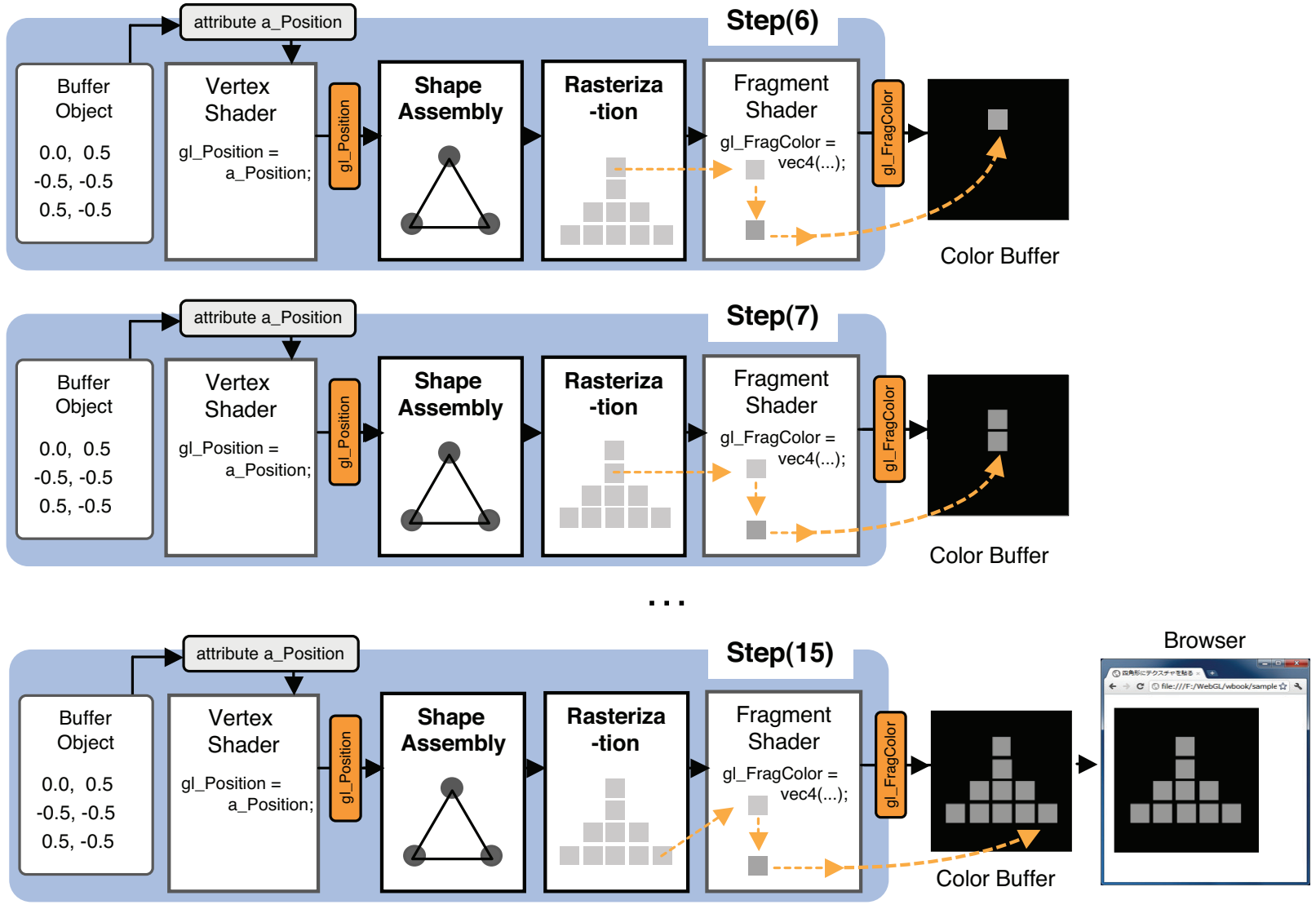
Shape Assembly and Rasterization

- Two stages between the vertex and fragment shaders
- The geometric shape assembly process
  - The geometric shape (line/triangle) is assembled from the specified vertex coords.
- The rasterization process
  - The geometric shape assembled in the geometric assembly process is converted into fragments.
  - Reading materials
    - Line drawing algorithm @Wikipedia
    - The Rasterization Stage @scratchapixel.com


processed in parallel
shape assembly
rasterization
JavaScript
function main() {
var gl=getWebGL…
gl.drawArrays
(gl.TRIANGLES, 0, n);
Buffer Object
0.0, 0.5
-0.5, -0.5
0.5, -0.5
attribute a_Position
Vertex Shader
gl_Position =
a_Position;
gl_Position
Shape Assembly
Rasteriza -tion
Fragment Shader
gl_FragColor =
vec4(...);
gl_FragColor
Step(1)
Step(2)
Step(3)
Step(4)
Step(5)

Fragment Shader Invocations

processed in parallel



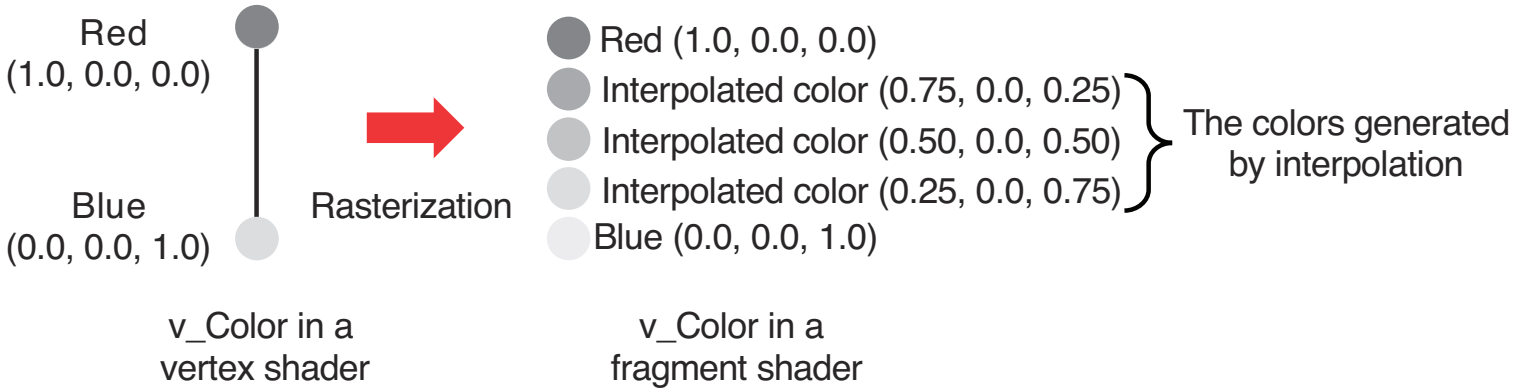
Example #5: `HelloTriangle_FragCoord`

Example #5: `HelloTriangle_FragCoord`

- http://rodger.global-linguist.com/webgl/ch05/HelloTriangle_FragCoord.html
- Sets the color of each fragment based on its location
- What to learn
  - How to access the coords of each fragment
- `gl_FragCoord`
  - built-in variable of type `vec4`
  - The 1st & 2nd component are the coords of the fragment in the <canvas> coords system (window coords system)
- The width & height of the `<canvas>` element can be accessed from `gl.drawingBufferWidth` & `gl.drawingBufferHeight`, respectively.
- The window coords system has its origin at the lower left corner.
  → Try https://xregy.github.io/webgl/src/HelloQuad_FragCoord.html

Functionality of Varying Variables and the Interpolation Process

- How `ColoredTriangle` example works? Why the fragments are assigned their colors in such a way?
- When three vertices (of a triangle) have different values for a varying variable, how the values are assigned to the fragments?
  → **(linearly) interpolated at the rasterization stage**



line example

Barycentric Coordinates

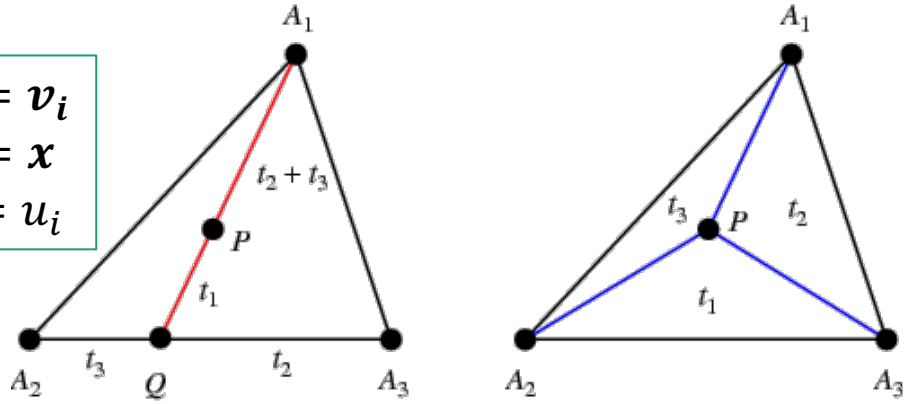
- A 2D point is represented by a three-tuple coordinates (u_1, u_2, u_3) with respect to a triangle composed of vertices $\boldsymbol{v_1}$, $\boldsymbol{v_2}$, and $\boldsymbol{v_3}$ whose Cartesian coords are $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, and $\boldsymbol{x_3}$, respectively.
- The Cartesian coords of the point whose barycentric coords is (u_1, u_2, u_3) is $u_1\boldsymbol{x_1} + u_2\boldsymbol{x_2} + u_3\boldsymbol{x_3}$.
- The three vertices $\boldsymbol{v_1}$, $\boldsymbol{v_2}$, and $\boldsymbol{v_3}$ have their barycentric coordinates $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, respectively.
- The sum of three components always becomes 1.
- Different signs depending on the area.
- Can be computed by signed area.

Barycentric Coordinates (cont'd)

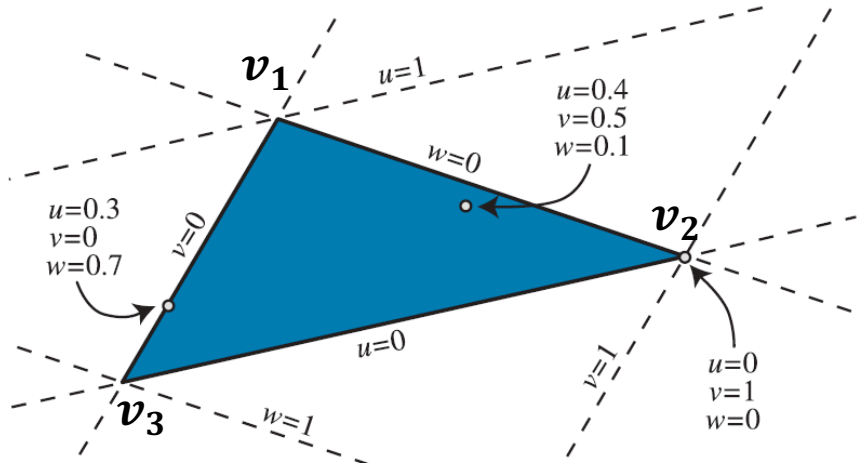
- Barycentric coords for a point $\boldsymbol{x} = (x, y)$ on the triangle with vertices $\boldsymbol{x_1} = (x_1, y_1), \boldsymbol{x_2} = (x_2, y_2), \boldsymbol{x_3} = (x_3, y_3)$

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$A_i = \boldsymbol{v_i}$
$P = \boldsymbol{x}$
$t_i = u_i$



(Mathworld)



$u = u_1$
$v = u_2$
$w = u_3$

(http://lib.znate.ru)

Linear Interpolation on a Triangle

- Any point on a triangle can be represented in the barycentric coordinates as
  (u_1, u_2, u_3) where $u_1, u_2, u_3 \geq 0$ and $u_1 + u_2 + u_3 = 1$.
- If three vertices, $\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}$ have values f_1, f_2, f_3 for a varying variable, the value at the point (u_1, u_2, u_3) can be computed as
  $$u_1 f_1 + u_2 f_2 + u_3 f_3$$
- Examples
  - The barycentric coords of the centroid of a triangle is $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ therefore it has the average value $\frac{f_1+f_2+f_3}{3}$.
  - The barycentric coords of the midpoint of $\boldsymbol{v_2}$ and $\boldsymbol{v_3}$ is $\left(0, \frac{1}{2}, \frac{1}{2}\right)$ therefore it has the value $\frac{f_2+f_3}{2}$.

Example #6: `TextureQuad`

Example #6: `TextureQuad`

- http://rodger.global-linguist.com/webgl/ch05/TexturedQuad.html
- Texture mapping example – gluing an image on a quad
- To run the example locally, you need to allow the browser to access local files from an HTML file. (blocked by default for security)
  - Chrome on MacOS

```
>> "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" --allow-file-access-from-files
```

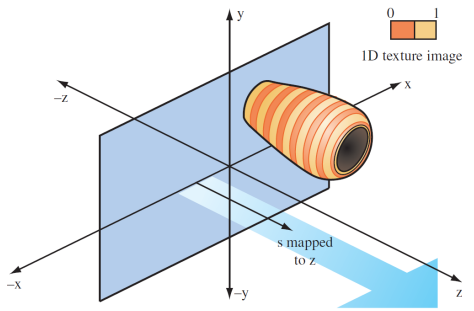
  - Chrome on Windows

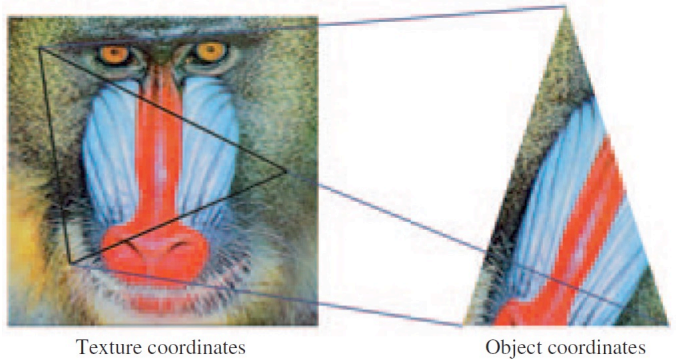
```
>> "C:\PathTo\Chrome.exe" --allow-file-access-from-files
```

Textures

- An OpenGL object that contains one or more images that all have the same image format (type + size + image format)
- 1D/2D/3D textures and their variants
- Sizes recommended to be powers-of-two for the best performance
- Not restricted to “images”, but can be used as “lookup tables” for various effects.
- Can be used in two ways
  - as a source of a texture access (texture mapping)
  - as a render target (render-to-texture)
- Reading material: https://www.khronos.org/opengl/wiki/Texture

Texture Mapping





- Proposed by Edwin Catmull (1974) ("Advanced Graphics Programming using OpenGL")
- To achieve a fine detail without using a large # of triangles by "pasting" an image on a surface (triangle)
- Not restricted to "gluing" → texture is a "lookup table" for various effects
- Done by assigning the texture image's pixel colors, called "texel (TEXture Element)", to the fragments in the fragment shader
- Procedure
  1. Prepare the image to be mapped on the geometric shape.
  2. Specify the image mapping method for the geometric shape.
  3. Load the texture image and configure it for use in WebGL.
  4. Extract the texels from the image in the fragment shader, and accordingly set the corresponding fragment.

Procedure for Texture Mapping

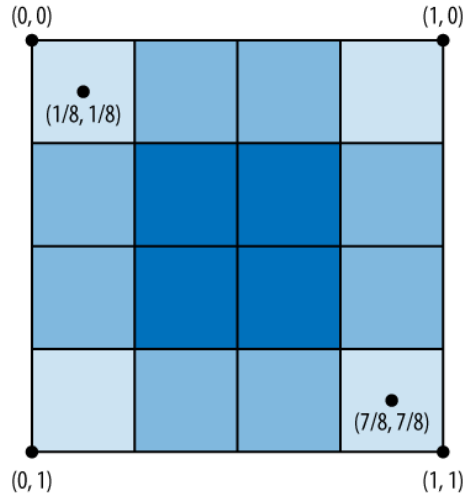
1. Create a **texture object** and load texel data into it.
2. Include **texture coordinates** with your vertices.
3. Associate a **texture sampler** with each texture map you intend to use in your shader.
4. Retrieve the texel values through the texture sampler from your shader.

Texture Objects

- `gl.createTexture()` / `gl.deleteTexture()`
- Needs to be “bound” to a “target” (texture type)
  - `gl.bindTexture()`
  - target: `gl.TEXTURE_2D`, `gl.TEXTURE_CUBE_MAP`
- Once bound to a target, it should not bound to a different target.
- In fact, the texture object is bound to **the target of the currently active “texture unit”** (more in later slides)

Texture Coordinates (Texcoords)

- How to specify the correspondence between the texture image and the triangle? → texture coordinates
- How to specify the texture coordinates system? Image width & height? Any problem?
- Texture coordinates are specified as floats in $[0,1]^2$ (`Float32Array`)
  → "normalized" → Can be specified regardless of the image size
- Passed from the vertex shader to the fragment shader as a varying variable → interpolated by the rasterizer



(iPhone 3D Programming)

Setting Up and Loading Images

- A texture object needs to be created by `gl.createTexture()`
  - Can be deleted by `gl.deleteTexture()`
- An image can be loaded using the JavaScript `HTMLImageElement` instance which can be created by `Image()` constructor.
- Since the **image loading is asynchronous**, we need to upload the image data in the following steps
  1. Create a JavaScript `Image` object.
  2. Register an event callback function where we upload the image data (e.g., `loadTexture()`)
  3. Specify the `src` property of the `Image` object. Then the (asynchronous) image loading starts. (The JavaScript program proceeds without waiting.)
- Due to the browser security restrictions, WebGL is not allowed to use images located in other domains for texture images.

Make the Texture Ready to Use in the WebGL System (`loadTexture()`)

- Procedure
  1. Flip an image's y-axis (`gl.pixelStorei()`)
  2. Make a texture unit active (`gl.activeTexture()`)
  3. Bind a texture object to a target (`gl.bindTexture()`)
  4. Set the texture parameters of the texture object (`gl.texParameteri()`)
  5. Upload the image data to a texture object (`gl.texImage2D()`)

1. Flip an Images' y-Axis

- `gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);`
- Not mandatory → Can be handled by changing the texcoords

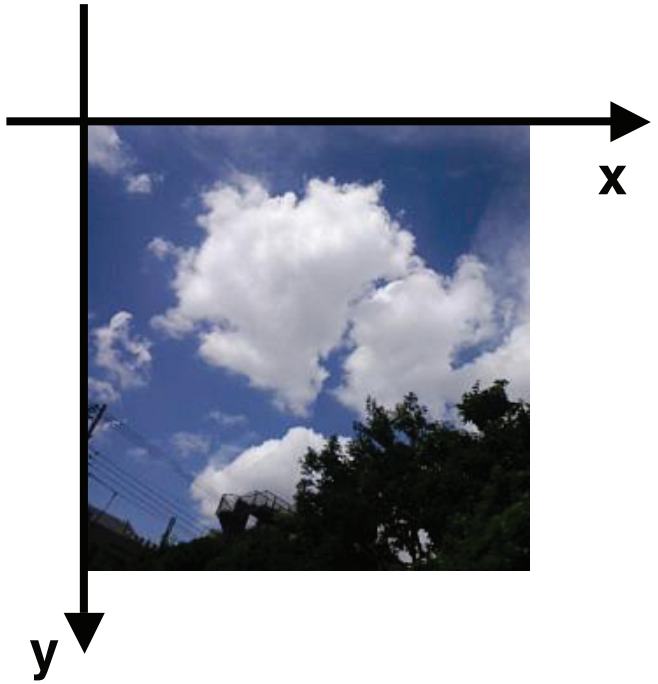
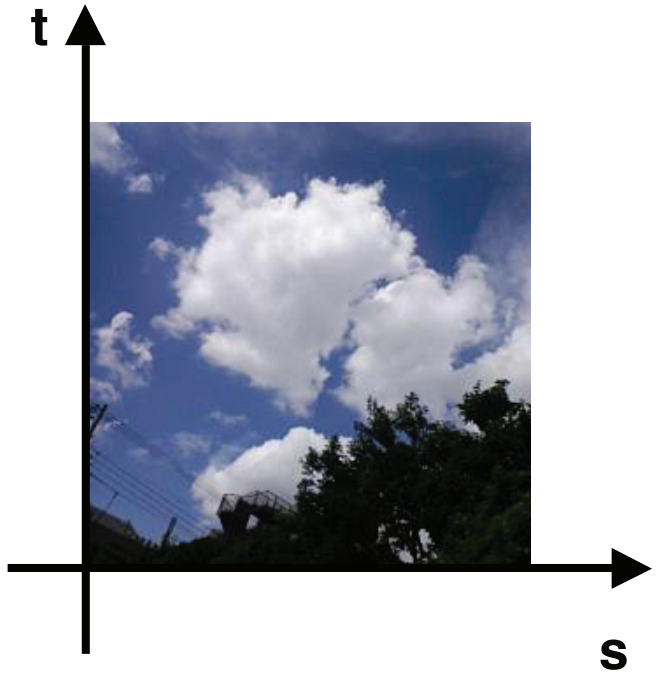


Image coordinate system



WebGL coordinate system

Texture Units

- Reading material (highly recommended): https://www.khronos.org/opengl/wiki/Texture#Texture_image_units
- Texture objects are accessed indirectly through “texture units”
- Each texture unit has its own “targets” – **We bind a texture object to one target of the currently active texture unit.**
- Texture binding for two reasons
  - To modify the texture object (setting parameters, uploading image data)
    - Since we can modify one texture object at a time, we can just bind it to the currently active texture unit when needed.
  - To access the texture object from the shaders
    - For multitexturing, since multiple texture objects need to be accessed at the same time, we need to bind each texture object to a separate texture unit
- Names: `gl.TEXTURE`*i* (e.g., `gl.TEXTURE3`) starting from `gl.TEXTURE0`.
  - At least 8 texture units
  - `gl.TEXTURE0==33984` and `gl.TEXTURE`*i*`==gl.TEXTURE0+`*i*
  - Maximum units can be queried by `gl.getParameter(gl.MAX_COMBINED_TEXTURE_IMAGE_UNITS)`

Texture Units (cont'd)

- Multitexturing is possible by binding multiple textures to separate tex units
- At any moment, one texture unit is active
  - "currently active texture unit"
  - Queried by `gl.getParameter(gl.ACTIVE_TEXTURE)`
  - By default, `TEXTURE0` is active.
- A texture object is bound to (the target of) a texture unit by calling
  - `gl.activeTexture(gl.TEXTURE`*i*`);`
    `gl.bindTexture(`*target*`, texture_object);`
- It seems that multiple texture objects with different types (targets) can be bound to the same texture unit. But this does not work. Read this.

2. Make a texture unit active (`gl.activeTexture()`)

- Multtexturing is supported through “texture units”
- All the textures to be accessed from the fragment shader at the same time need to be bound to separate texture unit.
- For single texturing, we do not need to call `gl.activeTexture()`. In this case, the default texture unit, `gl.TEXTURE0`, is used.

3. Bind a texture object to a target (`gl.bindTexture()`)

- A texture object needs to be “bound” to a target of specific type of the currently active texture unit
  - `gl.TEXTURE_2D` or `gl.TEXTURE_CUBE_MAP` (for cube map)
  - (WebGL2) `gl.TEXTURE_3D` and `gl.TEXTURE_2D_ARRAY` are added
- `gl.bindTexture()` performs two tasks
  - Enables the texture object and binds it to the target
  - Binds it to the (currently active) texture unit

4. Set the texture parameters of the texture object (`gl.texParameteri()`)

- To modify a texture object, it needs to be bound to the currently active texture unit
- Can be done before image loading is complete.
- Set the texture parameters that specify how the texture image will be processed when the texture image is mapped to shapes.
  - (WebGL 2.0) Sampler objects are used instead.
- Magnification (`gl.TEXTURE_MAG_FILTER`) – How to determine the color if a texel covers multiple fragments? → interpolation
- Minification (`gl.TEXTURE_MIN_FILTER`) – How to determine the color if multiple texels are mapped to a fragment? → Averaging, Mipmap
- Wrapping (`gl.TEXTURE_WRAP_*`)
  – How to handle the texcoords outside $[0,1]^2$?

5. Assigning a Texture Image to a Texture Object (`gl.texImage2D()`)

- Uploading the image data from the CPU memory to the VRAM
- Should be done after (asynchronous) image loading is complete
  → Should be called in the event callback function.
- `format` – specifies the format of the texel to be stored in.
- `internalformat` – should be the same as `format` parameter.
- `type` – specifies the data type of the texel data to be stored in.
- If the image data is defined as a typed array, `width`, `height`, and `format` should be set accordingly.
  - https://xregy.github.io/webgl/src/tex-typed-array-ubyte.html
  - https://xregy.github.io/webgl/src/tex-typed-array-float.html
  - `gl.pixelStorei(gl.UNPACK_ALIGNMENT, 1)` may need to be called.

Access the Texture from the Fragment Shader

- Procedure
  1. Pass the texture unit to the fragment shader (`gl.uniform1i()`)
     - A "sampler" of the appropriate type is declared as a uniform
     - **Pass `gl.TEXTURE`*i*`-gl.TEXTURE0` not `gl.TEXTURE`*i* to the sampler!**
     - `WebGLSampler` is supported in WebGL2.
  2. Pass texture coordinates from the vertex shader to the fragment shader
     - Initialized and passed to the vertex shader as vertex attributes
     - Passed to the fragment shader as a varying variable → interpolated by the rasterizer
  3. Retrieve the texel color in the fragment shader (`texture2D()`)
     - `vec4 texture2D(sampler2D sampler, vec2 coord)`
     - Always returns `vec4` → What if the internal format is NOT RGBA? What if ubyte?
     - Interpolated (for magnification) or averaged/mipmapped (for minification) depending on the setting.

Lab Activities

- Modify the texcoords (including outside $[0,1]^2$) and try various settings for `gl.texParameteri()`.
- http://rodger.global-linguist.com/webgl/ch05/TexturedQuad_Repeat.html
- http://rodger.global-linguist.com/webgl/ch05/TexturedQuad_Clamp_Mirror.html

Example #7: `MultiTexture`

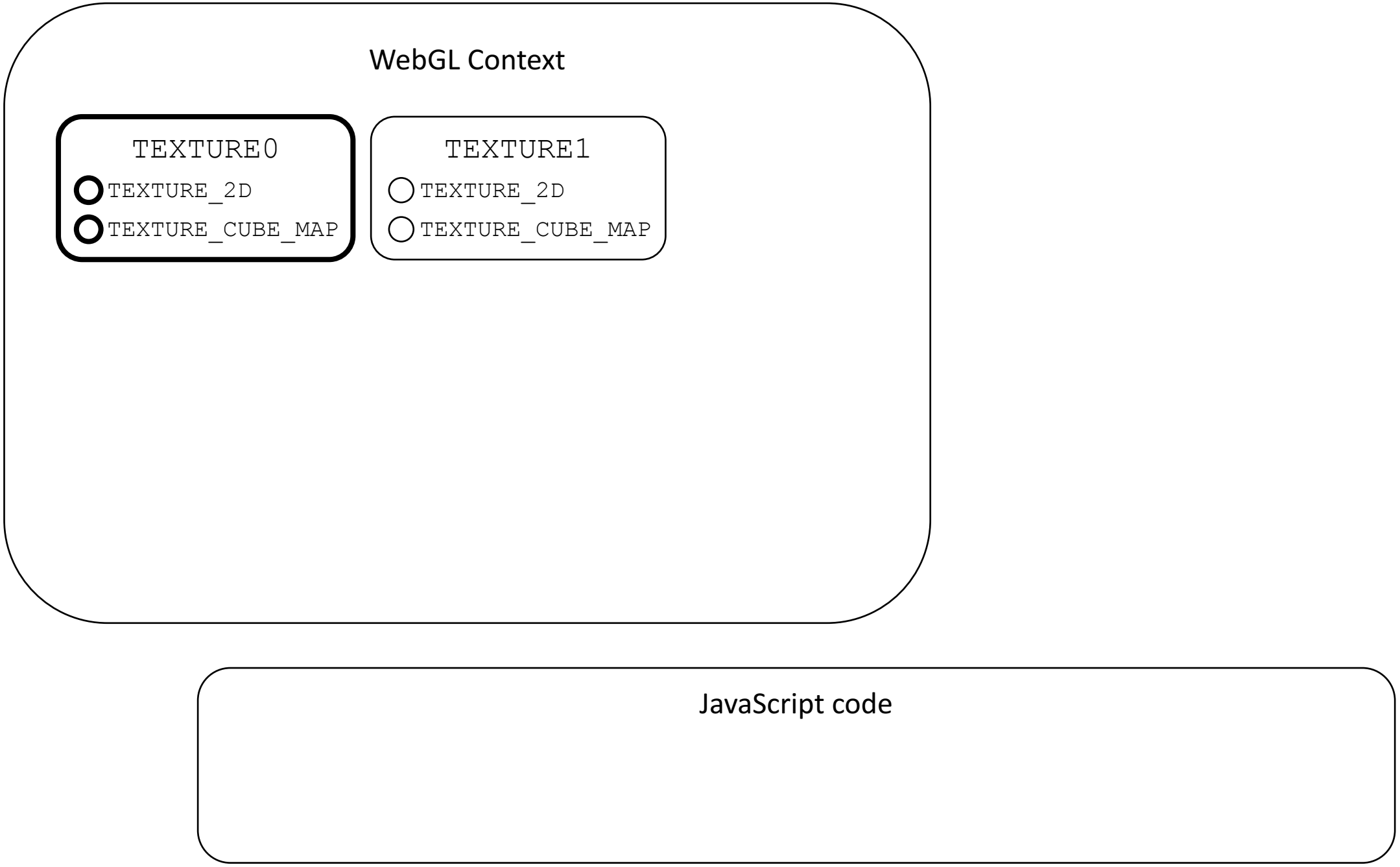
Example #7: `MultiTexture`

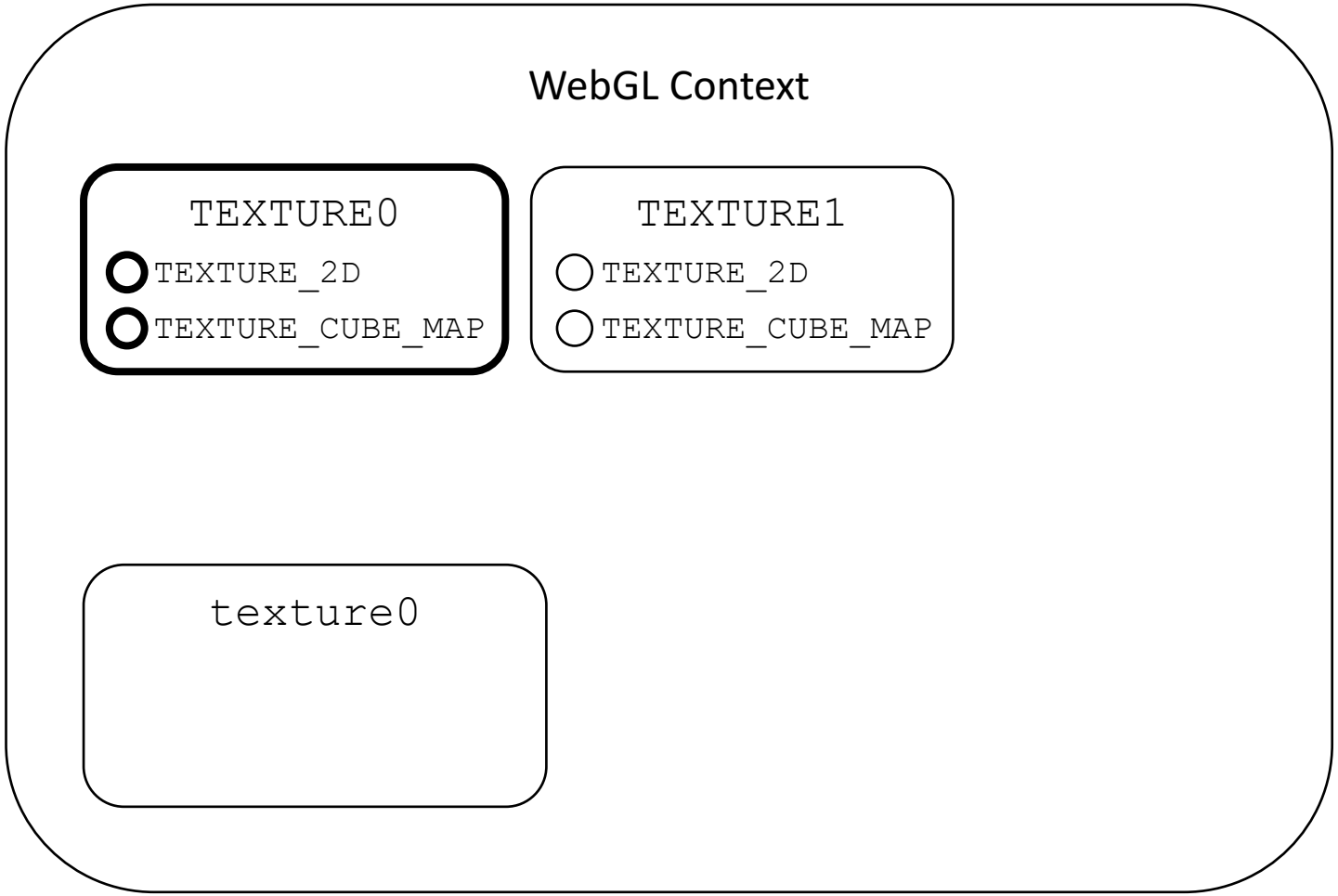
- http://rodger.global-linguist.com/webgl/ch05/MultiTexture.html
- Multitexturing
- Two texel values are merged in the fragment shader (component-wise multiplication)
  - Any combination of colors is possible

Extra Example: `tex-multi`

Extra Example: `tex-multi`

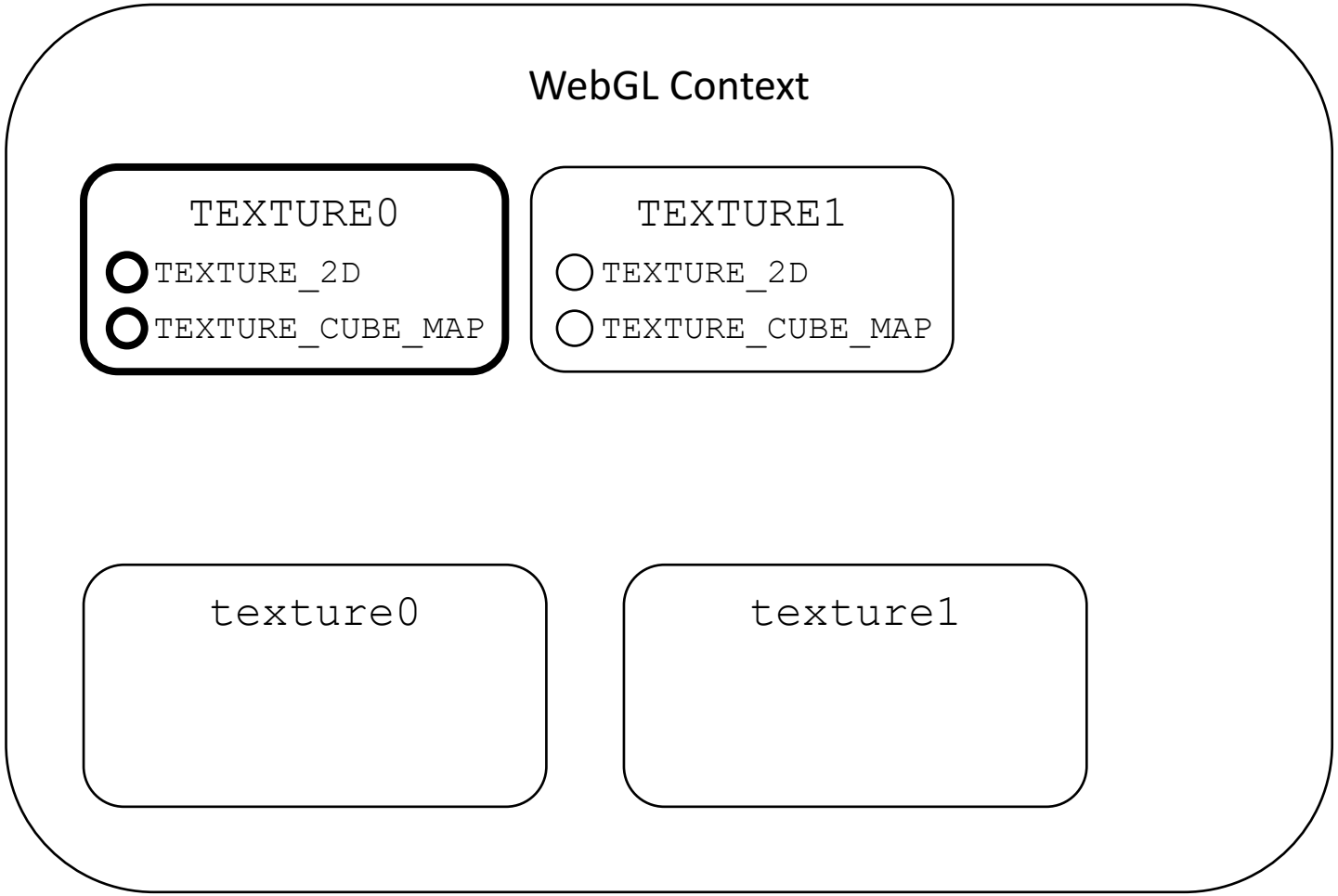
- https://xregy.github.io/webgl/src/tex-multi.html
- Multitexturing + animation
- Separate texcoords for each texture
- Two animation callback functions to handle asynchronous image loading
  - `tick_init()`: keeps checking if image loadings are complete
  - `tick()`: uploads textures and starts animation


WebGL Context
TEXTURE0
TEXTURE_2D
TEXTURE_CUBE_MAP
TEXTURE1
TEXTURE_2D
TEXTURE_CUBE_MAP
JavaScript code



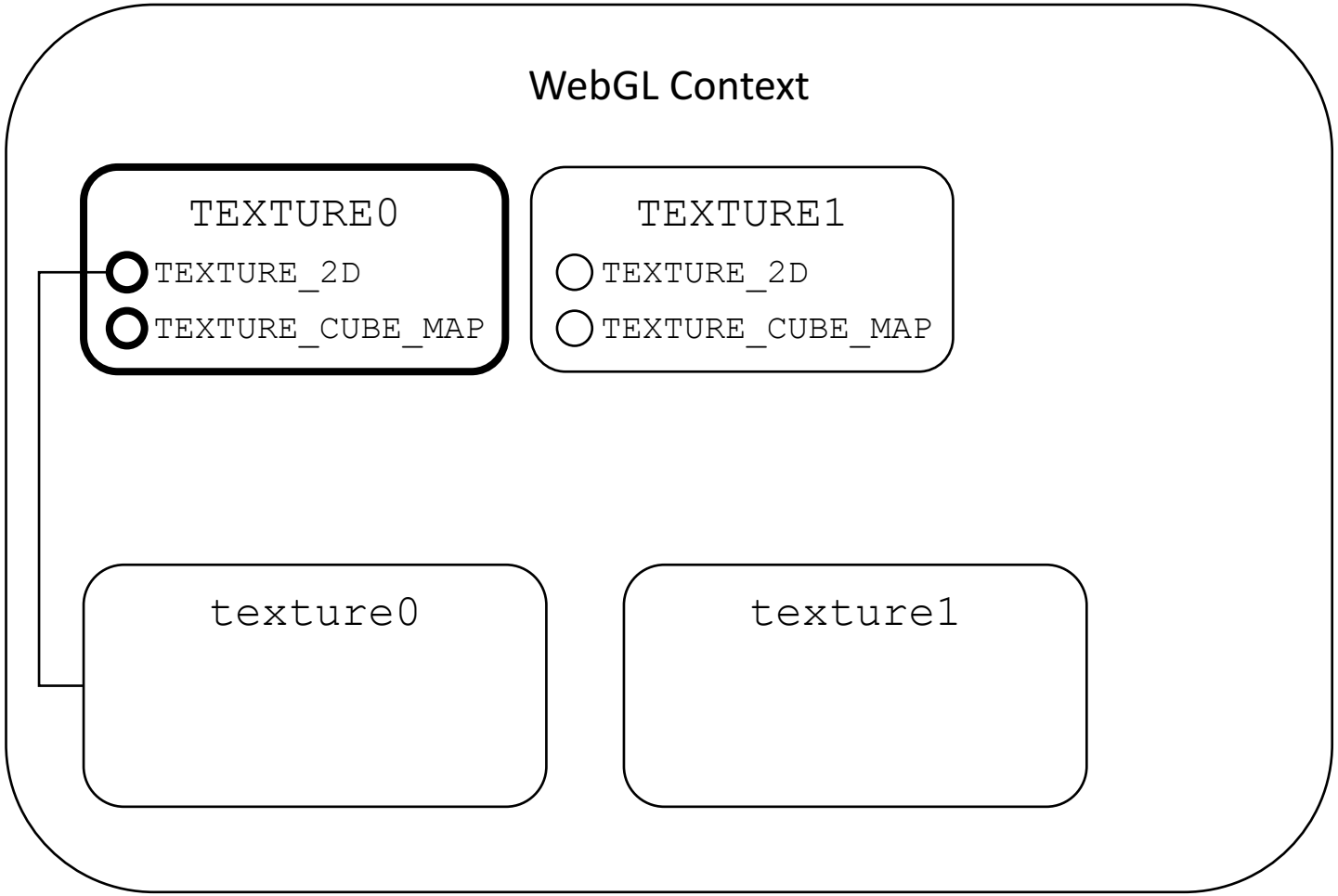
JavaScript code

```
var texture0 = gl.createTexture();
```



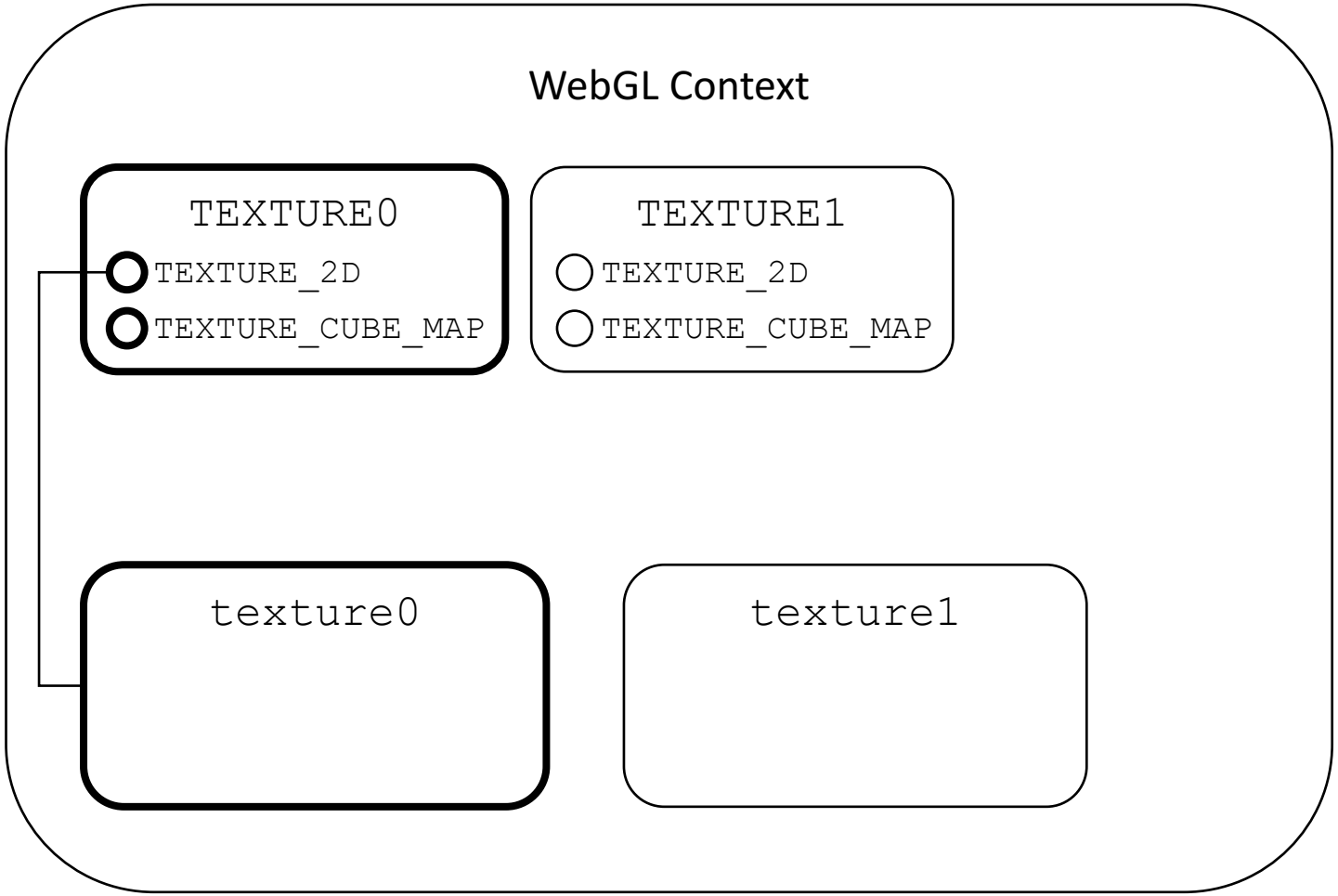
JavaScript code

```
var texture1 = gl.createTexture();
```



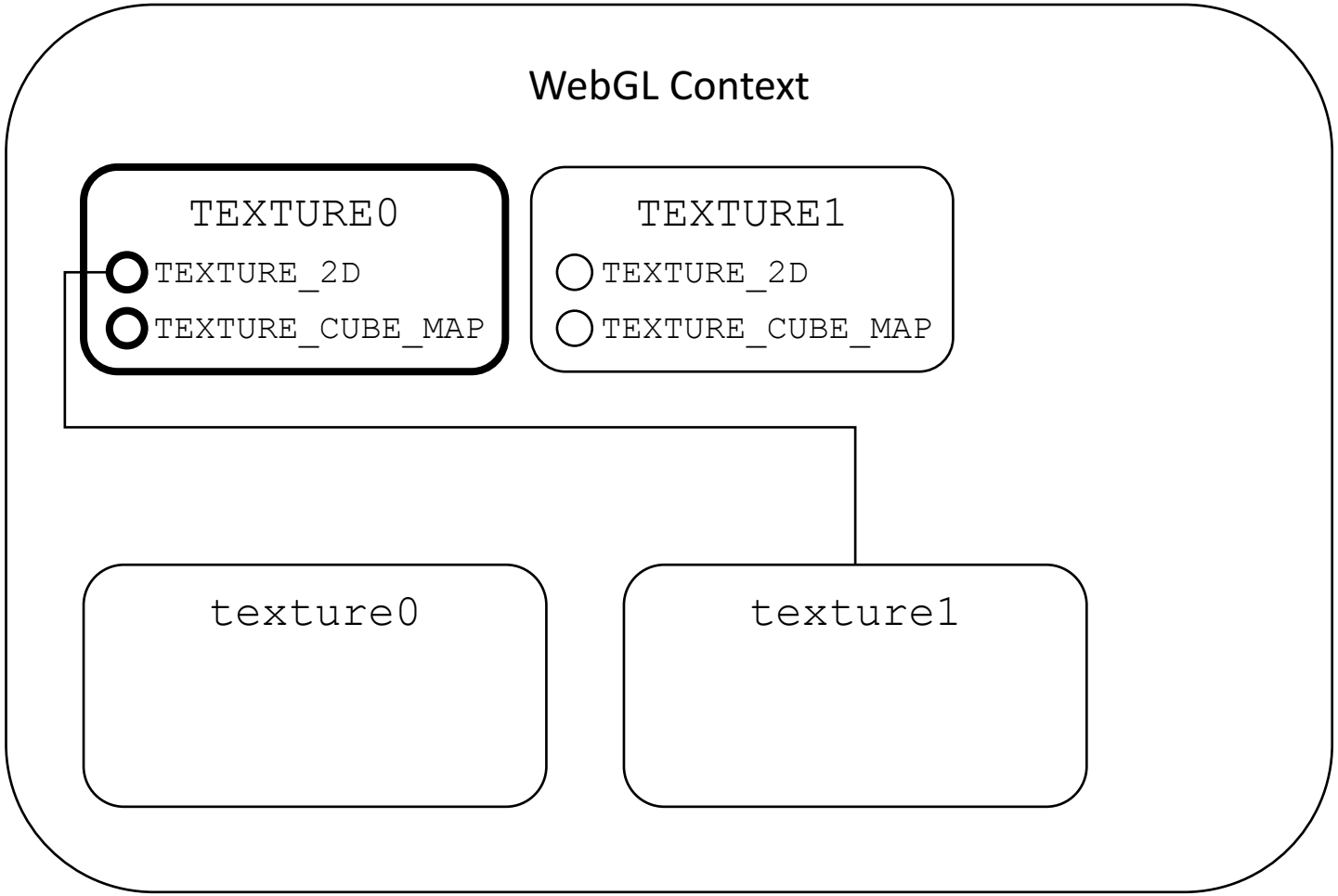
JavaScript code

```
gl.bindTexture(gl.TEXTURE_2D, texture0);
```



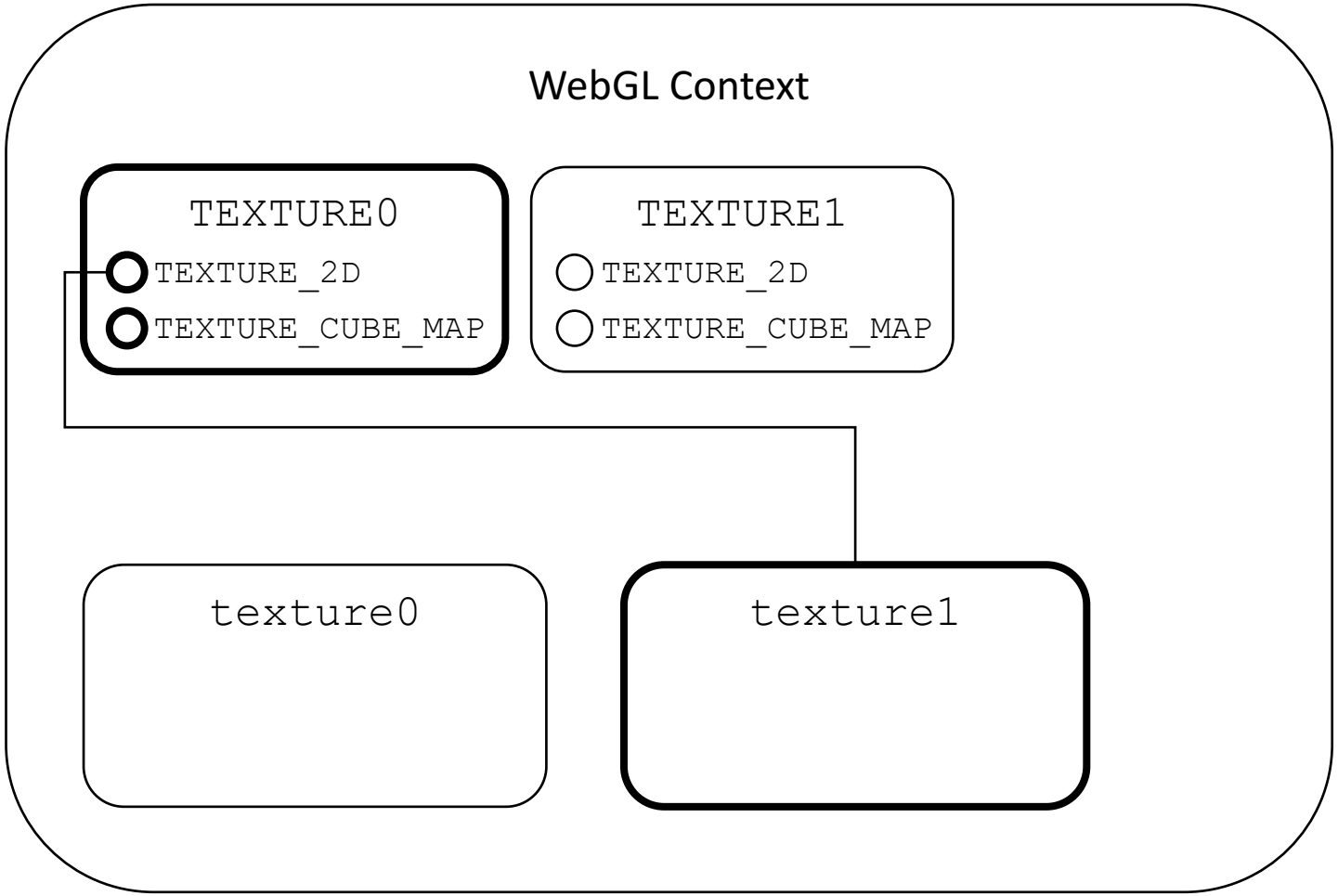
JavaScript code

```
gl.texParameteri(gl.TEXTURE_2D, …);
gl.texImage2D(gl.TEXTURE_2D, …);
```



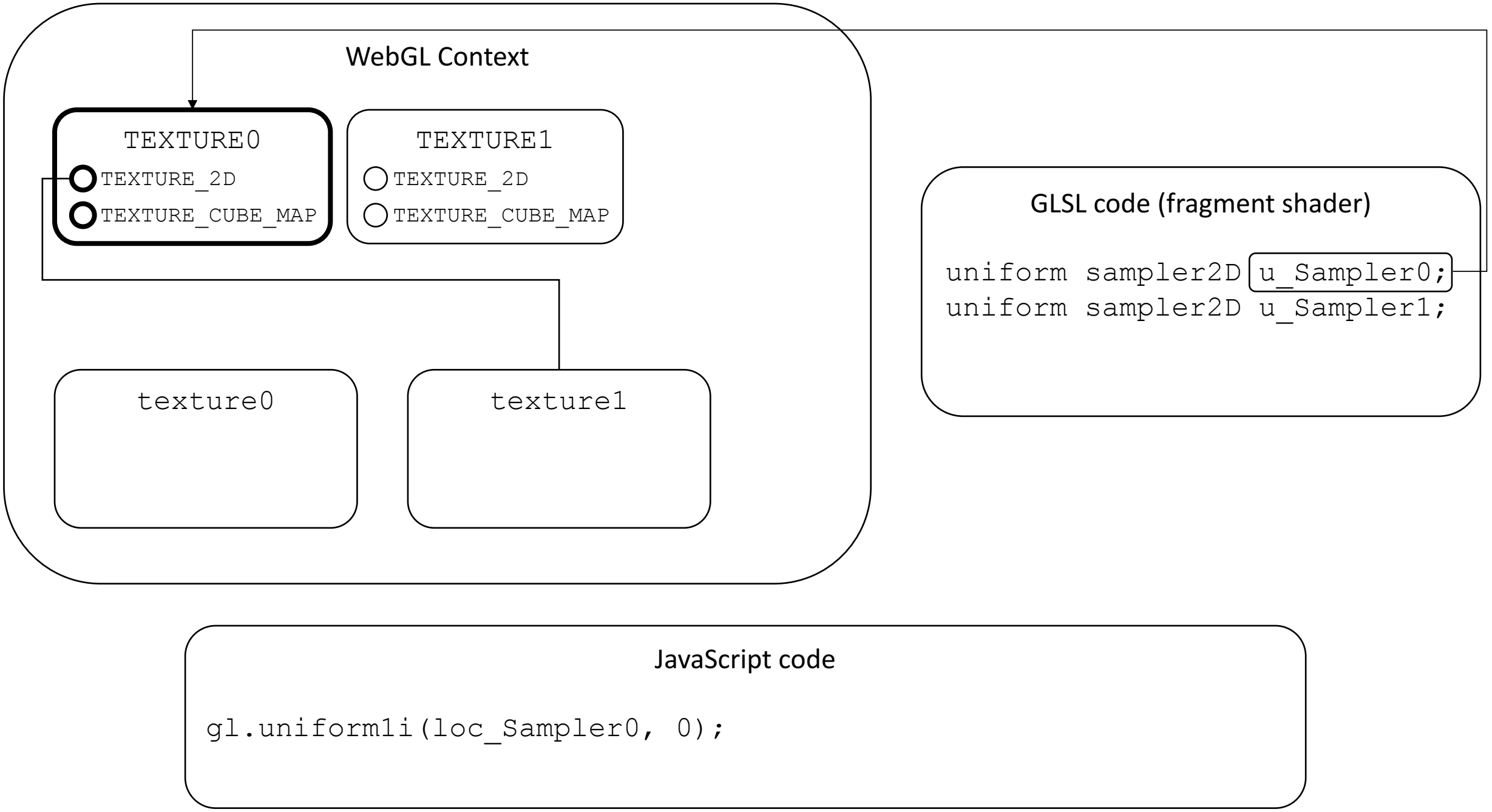
JavaScript code

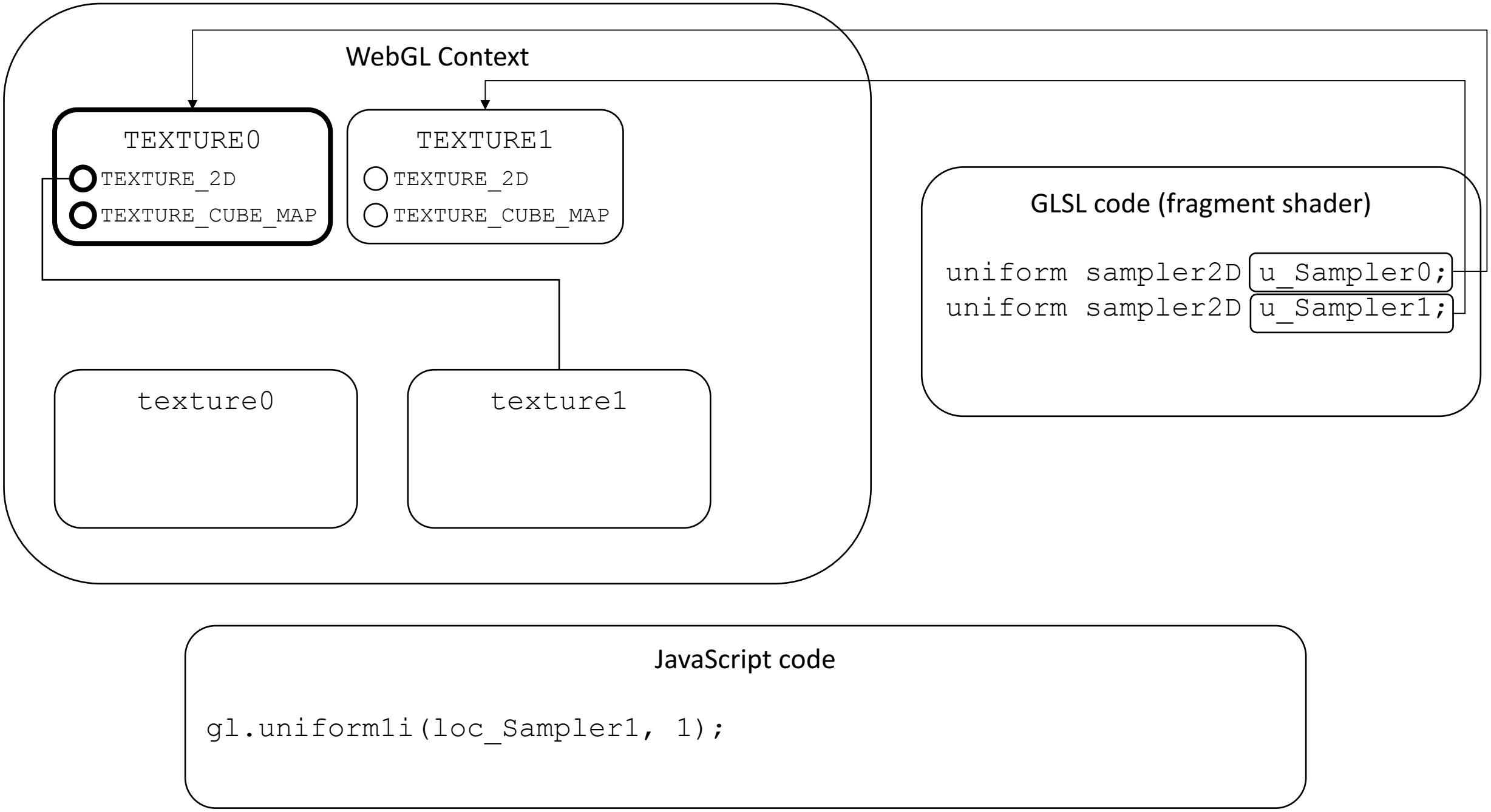
```
gl.bindTexture(gl.TEXTURE_2D, texture1);
```



JavaScript code

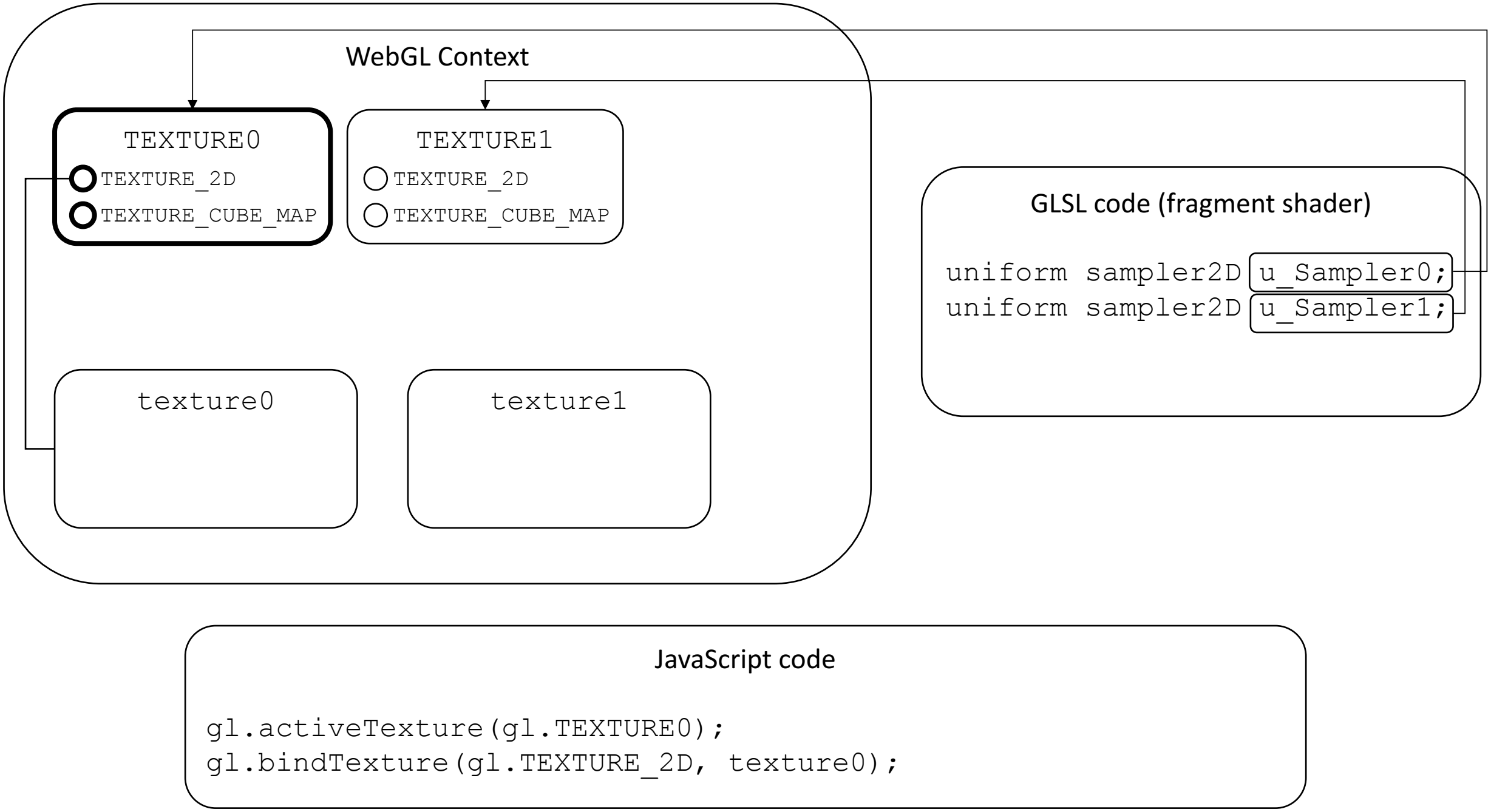
```
gl.texParameteri(gl.TEXTURE_2D, …);
gl.texImage2D(gl.TEXTURE_2D, …);
```


WebGL Context
TEXTURE0
TEXTURE_2D
TEXTURE_CUBE_MAP
TEXTURE1
TEXTURE_2D
TEXTURE_CUBE_MAP
texture0
texture1
GLSL code (fragment shader)
uniform sampler2D u_Sampler0;
uniform sampler2D u_Sampler1;
JavaScript code
gl.uniform1i(loc_Sampler0, 0);



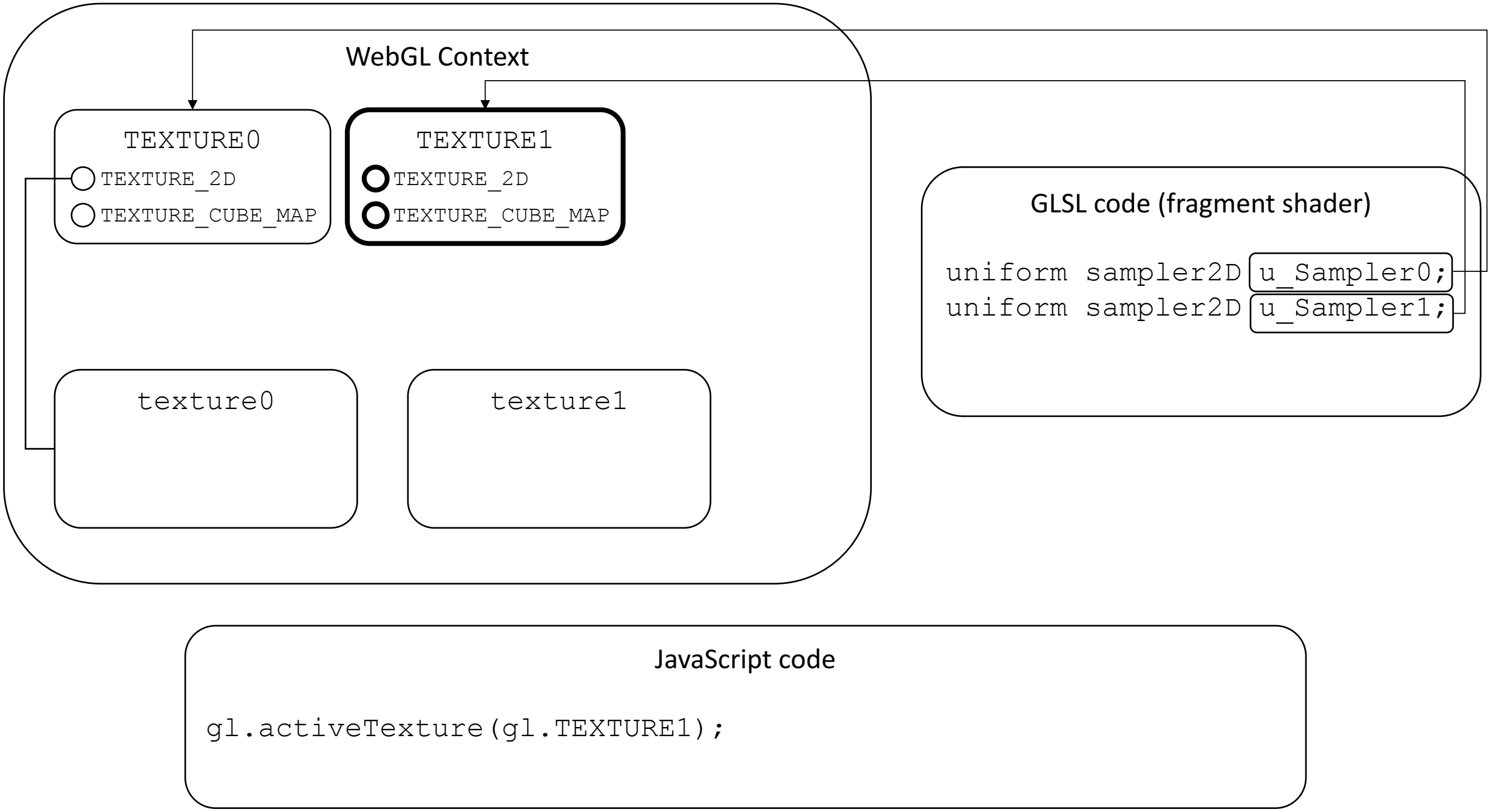
JavaScript code

```
gl.uniform1i(loc_Sampler1, 1);
```



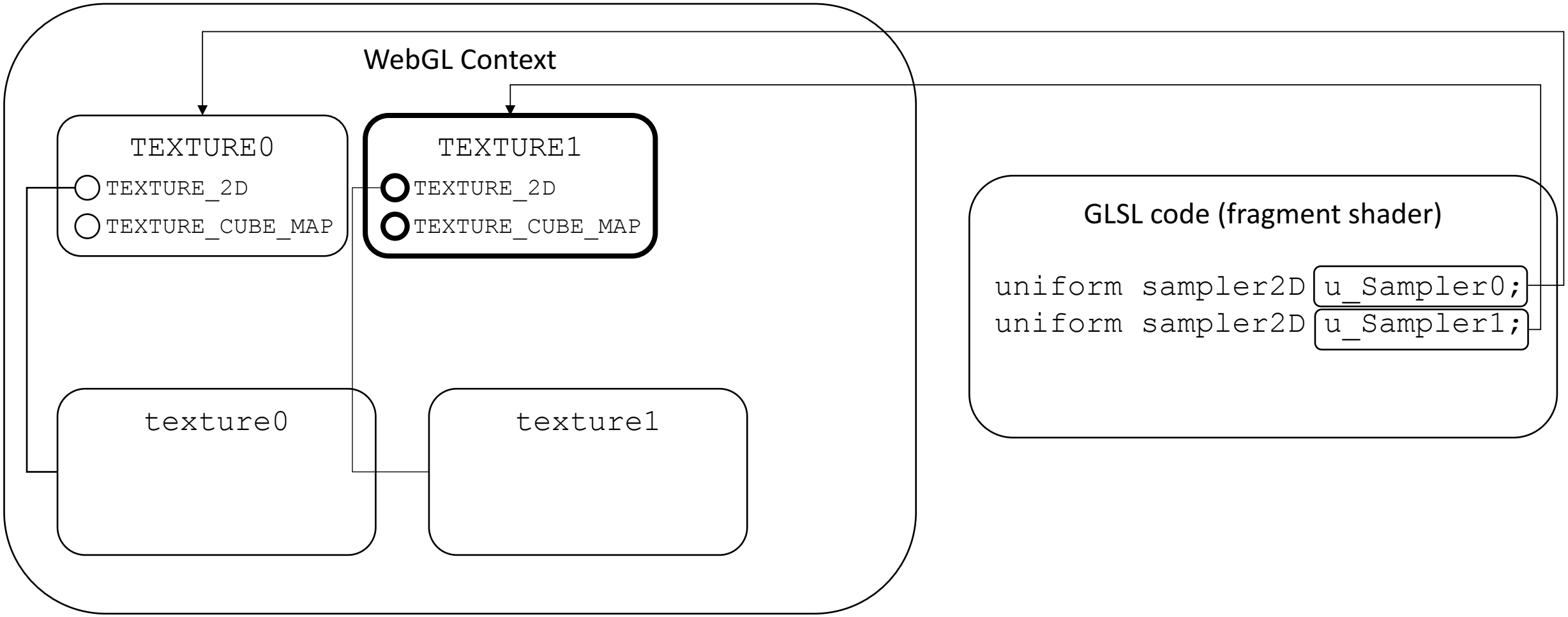
JavaScript code

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture0);
```



JavaScript code

```
gl.activeTexture(gl.TEXTURE1);
```



JavaScript code

```
gl.bindTexture(gl.TEXTURE_2D, texture1);
```

To be covered later…

- cube map
- mipmap