WebGL® is a software interface for accessing graphics hardware from within a web browser. Based on OpenGL ES 2.0, WebGL allows a programmer to specify the objects and operations in producing high-quality graphical images, specifically color images of 3D objects.

- [n,n] refers to sections in the WebGL 1.0 specification, available at www.khronos.org/webgl
- Content marked in purple does not have a corresponding function in OpenGL ES. The OpenGL ES 2.0 specification is available at www.khronos.org/registry/gles

WebGL function calls behave identically to their OpenGL ES counterparts unless otherwise noted.

### Interfaces

Interfaces are optional requests and may be ignored by an implementation. See getContextAttributes for actual values.

### WebGLContextAttributes [5.5.2]

This interface contains requested drawing surface attributes and is passed as the second parameter to getContext.

**Attributes:**

- **alpha**
  - Default: true
- **depth**
  - Default: true
- **stencil**
  - Default: false
- **antialias**
  - Default: true
- **premultipliedAlpha**
  - Default: true
- **preserveDrawingBuffer**
  - Default: false

### ArrayBuffer and Typed Arrays [5.12]

Data is transferred to WebGL using ArrayBuffer and views. Buffers represent unstructured binary data, which can be modified using one or more typed array views.

**Buffers**

- **ArrayBuffer**(ulong byteLength)**
  - byteLength: read-only, length of view in bytes.
  - Returns a new buffer.
  - To modify the data, create one or more views referencing it.

**Views**

- **Uint8Array**, **Uint8ClampedArray**, **Uint16Array**, **Uint32Array**, **Float32Array**, **Float64Array**

### Whole Framebuffer Operations [5.13.3]

void clearColor(float red, float green, float blue, float alpha)

void clearDepth(float depth)

### Buffer Objects [5.13.5]

- **bufferData**(enum target, long offset, long length, long usage, Object data)**
  - length: number of elements in this view.
  - Creates a new buffer.

- **bufferSubData**(Object data)**
  - length: number of elements in this view.
  - Modifies the data.

### View and Clip [5.13.3 - 5.13.4]

The viewport specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Drawing buffer size is determined by the HTMLCanvasElement.

void depthRange(float zNear, float zFar)**
- zNear: Clamped to the range 0 to 1
- zFar: Clamped to the range 0 to 1

void scissor(int x, int y, long width, long height)**
- x, y: Clamped to the range 0 to 1

### Detect context lost events [5.13.13]

- **isContextLost()**
Textures Objects [5.1.3.8]  
Texture objects provide storage and state for texturing operations. WebGL adds an error for operations relating to the currently bound texture if no texture is bound.

```plaintext
void activeTexture(enum target)  
texture: TEXTURE_2D, TEXTURE_CUBE_MAP

void copyTexImage2D(enum target, int level, enum internalformat, int x, int y, long width, long height, int border)  
texture: TEXTURE_2D, TEXTURE_CUBE_MAP

void copyTexSubImage2D(enum target, int level, int xoffset, int yoffset, int x, int y, int width, int height)  
```

Shaders must be loaded with a source string (Object) which must be linked (Object) and attached to a program (Object) and then used (Object).

### Special Functions [5.1.3.9]

- **contextStr**
  - **getContextAttributes**

```plaintext
void disable(enum cap)  
cap: BLEND, CULL, FACE, DEPTH, TEST, DITHER, POUSON, TEXTURE, SAMPLE_COVERAGE, SAMPLE_COVERAGE_STENCIL, TEST_STENCIL

void enable(enum cap)  
cap: See cap for disable
```

### Uniforms and Attributes [5.1.10]

Values used by the shaders are passed in as uniform or vertex attributes.

```plaintext
void disableVertexAttribArray(uint index)  
index: [0, MAX_VERTEX_ATTRIBS - 1]

void enableVertexAttribArray(uint index)  
index: [0, MAX_VERTEX_ATTRIBS - 1]
```

Object getActiveArray(Object program, uint index)

Object getActiveUniform(Object program, uint index)

ulong getActiveUniformLocation(Object program, program, string name)

any getAttribLocation(Object object, Object shader)

void getVertexAttribOffset(Object object, int index, long offset, long stride)

void getVertexAttribPointerv(Object object, int index, Object pointer)

void pixelStorei(enum target, int param)
```

### Framebuffer Objects [5.1.3.6]

Framebuffer objects provide an alternative rendering target to the drawing buffer.

```plaintext
Object framebuffer()

void bindFramebuffer(Object buffer)

target: FRAMEBUFFER

enum checkFramebufferStatus(enum target)
```

Object createFramebuffer()

```
void disableFramebufferObject(Object buffer)
```

void framebufferRenderbuffer(enum target, enum attachment, enum renderbufferTarget, Object renderbuffer)

Object framebufferTarget = Object renderbuffer

bool isFramebuffer(Object framebuffer)

Object createContext()

```
void deleteFramebufferObject(Object buffer)
```

Object getActiveTexture(Object texture)

```
void getFramebufferAttachmentParameter(Object object, int target, enum cap, int location)
```

Object getVideoSource(Object video)

```
void getVideoParameter(Object object, int target, enum param, Object param)
```

Framebuffer textureObject to use when drawing

```plaintext
void pixelStorei(enum target, int param)
```

Object pixels\[\]

```plaintext
void texImage2D(enum target, int level, enum internalformat, int xoffset, int yoffset, int x, int y, int width, int height)
```

```
void texSubImage2D(enum target, int level, int xoffset, int yoffset, int x, int y, int width, int height)
```

Writing to the Draw Buffer [5.1.3.11]

When rendering is directed to drawing buffer, OpenGL ES 2.0 rendering calls cause the drawing buffer to be presented to the HTML page composer at start of next compositing operation.

```plaintext
void drawArrays(enum mode, int first, long count)
```

```plaintext
void drawElements(enum mode, long count, enum type, Object index)
```

```plaintext
void drawBuffers(enum mode, long count, enum type, Object index)
```

```plaintext
void drawElements(enum mode, long count, enum type, long offset)
```

```plaintext
void drawBuffers(enum mode, long count, enum type, long offset)
```

```plaintext
void invalidateFramebufferRange(Object framebuffer, Object buffer, Object offset, Object size)
```

```plaintext
void invalidateRenderbuffers(Object renderbuffers, Object buffer, Object offset, Object size)
```

```plaintext
void invalidateRenderbufferRange(Object renderbuffer, Object buffer, Object offset, Object size)
```

```plaintext
void pixelStorei(enum target, int param)
```

```plaintext
void pixelSubBuffer(int target, Object buffer)
```

```plaintext
void pixelTransferi(enum target, int param)
```

```plaintext
void pixelStorei(enum target, int param)
```
Types [4.1]
A shader can aggregate these using arrays and structures to build more complex types. There are no pointer types.

Basic Types
- void: no function return value or empty parameter list
- bool: Boolean
- int: signed integer
- float: floating scalar
- vec2, vec3, vec4: n-component floating point vector
- bvec2, bvec3, bvec4: Boolean vector
- mat2, mat3, mat4: 2x2, 3x3, 4x4 float matrix
- sampler2D: access a 2D texture
- samplerCube: access cube mapped texture
- struct-name: successions of variables separated by ;
  struct-name
    { type-name members
      }
  // optional variable declaration,
  // optionally an array

Structures and Arrays [4.1.8, 4.1.9]
- float foo[3]: * structures and blocks can be arrays
- * only 1-dimensional arrays supported
- * structure members can be arrays

Operators and Expressions
Operators [5.1] Numbered in order of precedence. The relational and equality operators > < <= >= == != evaluate to a Boolean. To compare vectors component-wise, use functions such as leqThan(), equal(), etc.

Operators
- parenthesis grouping N/A
- array subscript function call & constructor structure field or method selector, swizzler
- prefix increment and decrement
- prefix increment and decrement
- multiplicative
- additive
- relational
- equality
- logical and
- logical or
- logical inclusive or
- logical inclusive or
- selection (selects one entire operand. Use mix() to select individual components of vectors.)
- assignment arithmetic assignments
- sequence

Vector Components [5.5]
In addition to array numeric suffix syntax, names of vector components are denoted by a single letter. Components can be swizzled and replicated, e.g.: pos.xx, pos.zy

[36x775]OpenGL ES Shading Language 1.0 Quick Reference Card - Page 3

Preprocessor [3.4]

Preprocessor Directives
The number sign (#) can be immediately preceded or followed in its line by spaces or horizontal tabs.
# define #undef #if #ifdef #ifndef #else
#else if #endif #error #pragma #extension #version #line

Examples of Preprocessor Directives
- "extension_name" is a version specifier that states that the program is written in GLSL version 1.00. It is optional. If used, it must occur before anything else in the program other than whitespace or comments.
- "extension_name" is behavior, where behavior behavior can be require, enable, warn, or disable; and where extension_name is the extension supported by the compiler

Structured Arrays [4.1.8, 4.1.9]

Structures: struct type-name { members
    }

Arrays: float foo[3]: * structures and blocks can be arrays
- * only 1-dimensional arrays supported
- * structure members can be arrays

Structures and Arrays
- struct-name
    { type-name members
      }

Qualifiers
Storage Qualifiers [4.3]
Variable declarations may be preceded by one storage qualifier.

- none (Default) local read/write memory, or input parameter
- const: Compile-time constant, or read-only function parameter
- attribute: Linkage between a vertex shader and OpenGL ES for per-vertex data
- uniform: Value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL ES, and the application
- varying: Linkage between a vertex shader and fragment shader for interpolated data

Uniform [4.3.4]
Use to declare global variables whose values are the same across the entire primitive being processed. All uniform variables are read-only. Use uniform qualifiers with any basic data types, to declare a variable whose type is a structure, or an array of any of these. For example:

```
uniform vec4 lightPosition;
```

Varying [4.3.5]
The varying qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, mat4, or arrays of these. Structures cannot be varying. Varying variables are required to have global scope. Declaration is as follows:

```
varying vec3 normal;
```

Parameter Qualifiers [4.4]
Input values are copied in at function call time, output values are copied out at function return time.

- none: [Default] local read/write memory, or input parameter
- inout: For function parameters passed back out of a function, but not initialized for use when passed in
- inout: For function parameters passed both into and out of a function

Aggregate Operators and Constructors
Matrix Constructor Examples [5.4]
mat2(float); // identity
mat2(vec2, vec2); // column-major order
mat2(float, float, float, float); // column-major order

Structure Constructor Example [5.4]
struct light {float intensity; vec3 Color;};
```
m = matrixCompMult(m, m);          // component-wise multiply
v = cross(v, v);  // vector cross product
m = m * v;       // matrix * column vector linear algebraic multiply
m = v * m;           // row vector * matrix linear algebraic multiply
m = m * m; // linear algebraic multiply
m = m +/- m; // matrix component-wise addition/subtraction

Matrix Components [5.6]
Access components of a matrix with array subscript syntax. For example:
```
mat4 m; // m represents a matrix
m[3][3] = 2.0; // sets 4th element of 3rd column to 2.0
```

Examples of operations on matrices and vectors:
```
m = m + v; // scalar vector-component-wise
m = m * v; // scalar * matrix-component-wise
```

Array Operations [4.1.9]
Array elements are accessed using the array subscript operator [ ] . For example:
```
```
Built-In Inputs, Outputs, and Constants [7]
Shaders programs use Special Variables to communicate with fixed-function parts of the pipeline. Output Special Variables may be read back after writing. Input Special Variables are read-only. All Special Variables have global scope.

Vertex Shader Special Variables [7.1]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>highp vec4 gl_Position;</td>
<td>transformed vertex position</td>
<td>clip coordinates</td>
</tr>
<tr>
<td>mediump float gl_PointSize;</td>
<td>transformed point size (point rasterization only)</td>
<td>pixels</td>
</tr>
</tbody>
</table>

Built-In Inputs, Outputs, and Constants [7]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediump vec4 gl_FragCoord;</td>
<td>fragment position within frame buffer</td>
<td>window coordinates</td>
</tr>
<tr>
<td>bool gl_FrontFacing;</td>
<td>fragment belongs to a front-facing primitive</td>
<td>Boolean</td>
</tr>
<tr>
<td>mediump vec4 gl_PointCoord;</td>
<td>fragment position within a point (point rasterization only)</td>
<td>0.0 to 1.0 for each component</td>
</tr>
</tbody>
</table>

Fragment Shader Special Variables [7.2]
Fragment shaders may write to gl_FragColor or to one or more elements of gl_FragData[], but not both. The size of the gl_FragData array is given by the built-in constant gl_MaxDrawBuffers.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediump vec4 gl_FragCoord;</td>
<td>fragment color</td>
<td>RGBA color</td>
</tr>
<tr>
<td>mediump vec4 gl_FragData[n];</td>
<td>fragment color for color attachment n</td>
<td>RGBA color</td>
</tr>
</tbody>
</table>

Angle & Trigonometry Functions [8.1]

- radians() degrees to radians
- degrees() radians to degrees
- sin() sine
- cos() cosine
- tan() tangent
- asin() arc sine
- acot() arc cosine
- atan() arc tangent
- atan2() arctangent

Exponential Functions [8.2]

- pow() x^y
- exp() e^x
- log() log(x)
- log2() log2(x)
- sqrt() square root
- inversesqrt() inverse square root

Common Functions [8.3]

- abs() absolute value
- sign() returns -1.0, 0.0, or 1.0
- floor() nearest integer = x
- ceil() nearest integer > x
- fract() x - floor(x)
- mod() modulus
- min() minimum value
- max() maximum value
- clamp() clamp(x, min, max)
- mix() linear blend of x and y
- steep() 0.0 if x < edge, else 1.0
- smoothstep() linear blend of 0.0 to 1.0

Geometric Functions [8.4]

- float length(T) length of vector
- float distance(T, p0, p1) distance between points
- float dot(T, x, y) dot product
- vec3 cross(T, x, y) x cross product
- T normalize(T) normalize vector to length 1
- T faceforward(T, n, T, T) returns N if dot(N, f) < 0, else -N
- T refraction(T, N, T, T) reflection direction = - 2 * dot(N, f) * N
- T refracted(T, N, T, T) refraction vector

Matrix Functions [8.5]

- mat2x2() Returns a 2x2 matrix
- mat3x3() Returns a 3x3 matrix
- mat4x4() Returns a 4x4 matrix
- mat3x2() Returns a 3x2 matrix
- mat2x3() Returns a 2x3 matrix
- mat2x1() Returns a 2x1 matrix
- mat3x1() Returns a 3x1 matrix
- mat1x3() Returns a 1x3 matrix
- mat1x2() Returns a 1x2 matrix
- mat1x1() Returns a 1x1 matrix
- mat4() Returns a 4x4 matrix
- mat3() Returns a 3x3 matrix
- mat2() Returns a 2x2 matrix
- mat1() Returns a 1x1 matrix

Vector Relational Functions [8.6]

- bool lessThanEqual(T, T) T < y
- bool lessThanOrEqual(T, T) T <= y
- bool greaterThanEqual(T, T) x > y
- bool greaterThan(T, T) x > y
- bool equal(T, T) x = y
- bool notEqual(T, T) x != y
- bool all(T) true if any component of T is true
- bool any(T) true if all components of T are true
- bool not(T) logical complement of x

Texture Lookup Functions [8.7]

- vec4 texture2Dlod(sampler2D sampler, vec2 coord, float lod)
- vec4 texture2DProj(sampler2D sampler, vec2 coord, float lod)
- vec4 textureCube(samplerCube sampler, vec3 coord, float lod)
- vec4 textureCubeProj(samplerCube sampler, vec3 coord, float lod)

Sample Program
A shader pair that applies diffuse and ambient lighting to a textured object.

Vertex Shader
uniform mat4 mvp_matrix; // model-view-projection matrix
uniform mat3 normal_matrix; // normal matrix
uniform vec3 light_dir; // light direction in eye coords
attribute vec3 a_normal; // vertex normal
attribute vec4 a_vertex; // vertex position
varying vec3 v_normal; // texture coordinates
void main (void)
{
  // put vertex normal into eye coords
  vec3 v_normal = normalize(normal_matrix * a_normal);
  // emit diffuse scale factor, texcoord, and position
  vec3 diffuse = max(dot(light_dir, v_normal), 0.0); // vertex normal
  v_texcoord = a_texcoord; gl_Position = mvp_matrix * a_vertex;
}

Fragment Shader
precision mediump float;
uniform sampler2D t_reflectance;
uniform vec4 ambient;
attribute vec3 a_vertex;
attribute vec4 a_vertex;
void main (void)
{
  vec4 color = texture2D(t_reflectance, v_texcoord);
  gl_FragColor = color * (vec4(v_diffuse) + ambient);
}