



## Table of contents

### Introduction

WebGL™ is an immediate mode 3D rendering API designed for the web. It is derived from OpenGL® ES 2.0, and provides similar rendering functionality, but in an HTML context. WebGL is designed as a rendering context for the HTML Canvas element. The HTML Canvas provides a destination for programmatic rendering in web pages, and allows for performing that rendering using different rendering APIs. The only such interface described as part of the Canvas specification is the 2D canvas rendering context, `CanvasRenderingContext2D`. This document describes another such interface, `WebGLRenderingContext`, which presents the WebGL API.

The immediate mode nature of the API is a divergence from most web APIs. Given the many use cases of 3D graphics, WebGL chooses the approach of providing flexible primitives that can be applied to any use case. Libraries can provide an API on top of WebGL that is more tailored to specific areas, thus adding a convenience layer to WebGL that can accelerate and simplify development. However, because of its OpenGL ES 2.0 heritage, it should be straightforward for developers familiar with modern desktop OpenGL or OpenGL ES 2.0 development to transition to WebGL development.

Many functions described in this document contain links to OpenGL ES man pages. While every effort is made to make these pages match the OpenGL ES 2.0 specification [[GLES20](#)], they may contain errors. In the case of a contradiction, the OpenGL ES 2.0 specification is the final authority.

The remaining sections of this document are intended to be read in conjunction with the OpenGL ES 2.0 specification (2.0.24 at the time of this writing, available from the [Khronos OpenGL ES API Registry](#)). Unless otherwise specified, the behavior of each method is defined by the OpenGL ES 2.0 specification. This specification may diverge from OpenGL ES 2.0 in order to ensure interoperability or security, often defining areas that OpenGL ES 2.0 leaves implementation-defined. These differences are summarized in the [Differences Between WebGL and OpenGL ES 2.0](#) section.

## Context Creation and Drawing Buffer Presentation

Before using the WebGL API, the author must obtain a `WebGLRenderingContext` object for a given `HTMLCanvasElement` as described below. This object is used to manage OpenGL state and render to the drawing buffer, which must also be created at the time of context creation. The author may supply configuration options for this drawing buffer, otherwise default values shall be used as specified elsewhere in this document. This drawing buffer is presented to the HTML page compositor immediately before

## The canvas Element

A WebGLRenderingContext object shall be created by calling the `getContext()` method of a given HTMLCanvasElement [\[CANVAS\]](#) object with the exact string 'webgl'. This string is case sensitive. When called for the first time, a WebGLRenderingContext object is created and returned. Also at this time a drawing buffer shall be created. Subsequent calls to `getContext()` with the same string shall return the same object.

The HTML Canvas specification [\[CANVAS\]](#) defines the behavior when attempting to fetch two or more incompatible contexts against the same canvas element.

A second parameter may be passed to the `getContext()` method. If passed, this parameter shall be a WebGLContextAttributes object containing configuration parameters to be used in creating the drawing buffer. See [WebGLContextAttributes](#) for more details.

The `depth`, `stencil` and `antialias` attributes are requests, not requirements. The WebGL implementation does not guarantee that they will be obeyed, but should make a best effort to honor them. Combinations of attributes not supported by the WebGL implementation or graphics hardware shall not cause a failure to create a WebGLRenderingContext. The attributes actually used to create the context may be queried by the `getContextAttributes()` method on the WebGLRenderingContext. The `alpha`, `premultipliedAlpha` and `preserveDrawingBuffer` attributes must be obeyed by the WebGL implementation.

On subsequent calls to `getContext()` with the 'webgl' string, the passed WebGLContextAttributes object, if any, shall be ignored.

## The Drawing Buffer

The drawing buffer into which the API calls are rendered shall be defined upon creation of the WebGLRenderingContext object. The table below shows all the buffers which make up the drawing buffer, along with their minimum sizes and whether they are defined or not by default. The size of this drawing buffer shall be determined by the `width` and `height` attributes of the HTMLCanvasElement. The table below also shows the value to which these buffers shall be cleared when first created, when the size is changed, or after presentation when the `preserveDrawingBuffer` context creation attribute is false.

Buffer	Clear value	Minimum size	Defined by default?
Color	(0, 0, 0, 0)	8 bits per component	yes

Depth	1.0	16 bit integer	yes
Stencil	0	8 bits	no



If the requested width or height cannot be satisfied, either when the drawing buffer is first created or when the `width` and `height` attributes of the `HTMLCanvasElement` are changed, a drawing buffer with smaller dimensions shall be created. The dimensions actually used are implementation dependent and there is no guarantee that a buffer with the same aspect ratio will be created. The actual drawing buffer size can be obtained from the `drawingBufferWidth` and `drawingBufferHeight` attributes.

By default, the buffers shall be the size shown below. The optional [WebGLContextAttributes](#) object may be used to change whether or not the buffers are defined. It can also be used to define whether the color buffer will include an alpha channel. If defined, the alpha channel is used by the HTML compositor to combine the color buffer with the rest of the page. The `WebGLContextAttributes` object is only used on the first call to `getContext`. No facility is provided to change the attributes of the drawing buffer after its creation.

WebGL presents its drawing buffer to the HTML page compositor immediately before a compositing operation, but only if the drawing buffer has been modified since the last compositing operation. Before the drawing buffer is presented for compositing the implementation shall ensure that all rendering operations have been flushed to the drawing buffer. By default, after compositing the contents of the drawing buffer shall be cleared to their default values, as shown in the table above.

This default behavior can be changed by setting the `preserveDrawingBuffer` attribute of the [WebGLContextAttributes](#) object. If this flag is true, the contents of the drawing buffer shall be preserved until the author either clears or overwrites them. If this flag is false, attempting to perform operations using this context as a source image after the rendering function has returned can lead to undefined behavior. This includes `readPixels` or `toDataURL` calls, or using this context as the source image of another context's `texImage2D` or `drawImage` call.

While it is sometimes desirable to preserve the drawing buffer, it can cause significant performance loss on some platforms. Whenever possible this flag should remain false and other techniques used. Techniques like synchronous drawing buffer access (e.g., calling `readPixels` or `toDataURL` in the same function that renders to the drawing buffer) can be used to get the contents of the drawing buffer. If the author needs to render to the same drawing buffer over a series of calls, a [Framebuffer Object](#) can be used.

Implementations may optimize away the required implicit clear operation of the Drawing Buffer as long as a guarantee can be made that the author cannot gain access to buffer contents from another process. For instance, if the author performs an explicit clear then the implicit clear is not needed.

## The WebGL Viewport

OpenGL manages a rectangular viewport as part of its state which defines the placement of the rendering in the drawing buffer. Upon creation of the WebGL context, the viewport is initialized to a rectangle with origin at (0, 0) and width and height equal to (canvas.width, canvas.height).

A WebGL implementation *shall not* affect the state of the OpenGL viewport in response to resizing of the canvas element.

Note that if a WebGL program does not contain logic to set the viewport, it will not properly handle the case where the canvas is resized. The following ECMAScript example illustrates how a WebGL program might resize the canvas programmatically.

```
var canvas = document.getElementById('canvas1');
var gl = canvas.getContext('webgl');
canvas.width = newWidth;
canvas.height = newHeight;
gl.viewport(0, 0, canvas.width, canvas.height);
```

*Rationale* : automatically setting the viewport will interfere with applications that set it manually. Applications are expected to use `onresize` handlers to respond to changes in size of the canvas and set the OpenGL viewport in turn.

## Premultiplied Alpha, Canvas APIs and `texImage2D`

The OpenGL API allows the application to modify the blending modes used during rendering, and for this reason allows control over how alpha values in the drawing buffer are interpreted; see the `premultipliedAlpha` parameter in the [WebGLContextAttributes](#) section.

The HTML Canvas APIs `toDataURL` and `drawImage` must respect the `premultipliedAlpha` context creation parameter. When `toDataURL` is called against a Canvas into which WebGL content is being rendered, then if the requested image format does not specify premultiplied alpha and the WebGL context has the `premultipliedAlpha` parameter set to true, then the pixel values must be de-multiplied; i.e., the color channels are divided by the alpha channel. **Note** that this operation is lossy.

Passing a WebGL-rendered Canvas to the `drawImage` method of `CanvasRenderingContext2D` may or may not need to modify the the rendered WebGL content during the drawing operation, depending on the premultiplication needs of the `CanvasRenderingContext2D` implementation.

When passing a WebGL-rendered Canvas to the `texImage2D` API, then depending on the setting of the `premultipliedAlpha` context creation parameter of the passed canvas and the `UNPACK_PREMULTIPLY_ALPHA_WEBGL` pixel store parameter of the destination WebGL context, the pixel data may need to be changed to or from premultiplied form.



## WebGL Resources

OpenGL manages several types of resources as part of its state. These are identified by integer object names and are obtained from OpenGL by various creation calls. In contrast WebGL represents these resources as DOM objects. Each object is derived from the WebGLObject interface. Currently supported resources are: textures, buffers (i.e., VBOs), framebuffers, renderbuffers, shaders and programs. The `WebGLRenderingContext` interface has a method to create a `WebGLObject` subclass for each type. Data from the underlying graphics library are stored in these objects and are fully managed by them. The resources represented by these objects are guaranteed to exist as long as the object exists. Furthermore, the DOM object is guaranteed to exist as long as the author has an explicit valid reference to it **or** as long as it is bound by the underlying graphics library. When none of these conditions exist the user agent can, at any point, delete the object using the equivalent of a delete call (e.g., `deleteTexture`). If authors wish to control when the underlying resource is released then the `delete` call can be made explicitly.

## Security

### Resource Restrictions

WebGL resources such as textures and vertex buffer objects (VBOs) must always contain initialized data, even if they were created without initial user data values. Creating a resource without initial values is commonly used to reserve space for a texture or VBO, which is then modified using `texSubImage` or `bufferSubData` calls. If initial data is not provided to these calls, the WebGL implementation must initialize their contents to 0; depth renderbuffers must be cleared to the default 1.0 clear depth. For example, this may require creating a temporary buffer of 0 values the size of a requested VBO, so that it can be initialized correctly. All other forms of loading data into a texture or VBO involve either `ArrayBuffers` or DOM objects such as `images`, and are therefore already required to be initialized.

When WebGL resources are accessed by shaders through a call such as `drawElements` or `drawArrays`, the WebGL implementation must ensure that the shader cannot access either out of bounds or uninitialized data. See [Enabled Vertex Attributes and Range Checking](#) for restrictions which must be enforced by the WebGL implementation.

### Origin Restrictions

In order to prevent information leakage, WebGL disallows uploading as textures:

- Image or video elements whose origin is not the same as the origin of the Document that contains the WebGLRenderingContext's canvas element
- Canvas elements whose *origin-clean* flag is set to false

If the `texImage2D` or `texSubImage2D` method is called with otherwise correct arguments and an `HTMLImageElement`, `HTMLVideoElement`, or `HTMLCanvasElement` violating these restrictions, a `SECURITY_ERR` exception must be raised.

WebGL necessarily imposes stronger restrictions on the use of cross-domain media than other APIs such as the 2D canvas rendering context, because shaders can be used to indirectly deduce the contents of textures which have been uploaded to the GPU.

WebGL applications may utilize images and videos that come from other domains, with the cooperation of the server hosting the media, using [Cross-Origin Resource Sharing \[CORS\]](#). In order to use such media, the application must explicitly request permission to do so, and the server must explicitly grant permission. Successful CORS-enabled fetches of image and video elements from other domains [cause the origin of these elements](#) to be set to that of the containing Document [\[HTML\]](#).

The following ECMAScript example demonstrates how to issue a CORS request for an image coming from another domain. The image is fetched from the server without any credentials, i.e., cookies.

```
var gl = ...;  
var image = new Image();  
  
// The onload handler should be set to a function which uploads the HTMLImageElement  
// using texImage2D or texSubImage2D.  
image.onload = ...;  
  
image.crossOrigin = "anonymous";  
  
image.src = "http://other-domain.com/image.jpg";
```

Note that these rules imply that the *origin-clean* using WebGL will never be set to false.

flag for a canvas rendered

For more information on issuing CORS requests for image and video elements, consult:

- [CORS settings attributes \[HTML\]](#)
- [The img element \[HTML\]](#)
- [Media elements \[HTML\]](#)

## Supported GLSL Constructs

A WebGL implementation must only accept shaders which conform to The OpenGL ES Shading Language, Version 1.00 [[GLES20GLSL](#)], and which do not exceed the minimum functionality mandated in Sections 4 and 5 of Appendix A. In particular:

- A shader referencing state variables or functions that are available in other versions of GLSL, such as that found in versions of OpenGL for the desktop, must not be allowed to load.
- `for` loops must conform to the structural constraints in Appendix A.
- Appendix A mandates certain forms of indexing of arrays; for example, within fragment shaders, indexing is only mandated with a *constant-index-expression* (see [[GLES20GLSL](#)] for the definition of this term).  
In the WebGL API, only the forms of indexing mandated in Appendix A are supported.

In addition to the reserved identifiers in the aforementioned specification, identifiers starting with "webgl\_" and "\_webgl\_" are reserved for use by WebGL. A shader which declares a function, variable, structure name, or structure field starting with these prefixes must not be allowed to load.

## Defense Against Denial of Service

It is possible to create, either intentionally or unintentionally, combinations of shaders and geometry that take an undesirably long time to render. This issue is analogous to that of long-running scripts, for which user agents already have safeguards. However, long-running draw calls can cause loss of interactivity for the entire window system, not just the user agent.

In the general case it is not possible to impose limits on the structure of incoming shaders to guard against this problem. Experimentation has shown that even very strict structural limits are insufficient to prevent long rendering times, and such limits would prevent shader authors from implementing common algorithms.

User agents should implement safeguards to prevent excessively long rendering times and associated loss of interactivity. Suggested safeguards include:

- Splitting up draw calls with large numbers of elements into smaller draw calls.
- Timing individual draw calls and forbidding further rendering from a page if a certain timeout is exceeded.
- Using any watchdog facilities available at the user level, graphics API level, or operating system level to limit

the duration of draw calls.

- Separating the graphics rendering of the user agent into a distinct operating system process which can be terminated and restarted without losing application state.

The supporting infrastructure at the OS and graphics API layer is expected to improve over time, which is why the exact nature of these safeguards is not specified.

## Out-of-Range Array Accesses

Shaders must not be allowed to read or write array elements that lie outside the bounds of the array. This includes any variable of array type, as well as vector or matrix types such as `vec3` or `mat4` when accessed using array subscripting syntax. If detected during compilation, such accesses must generate an error and prevent the shader from compiling. Otherwise, at runtime they may return a constant value (such as 0), or the value at any valid index of the same array.

See [Supported GLSL Constructs](#) for more information on restrictions which simplify static analysis of the array indexing operations in shaders.

## DOM Interfaces

This section describes the interfaces and functionality added to the DOM to support runtime access to the functionality described above.

### Types

The following types are used in all interfaces in the following section.

```
typedef events::Event Event;
typedef html::HTMLCanvasElement HTMLCanvasElement;
typedef html::HTMLImageElement HTMLImageElement;
typedef html::HTMLVideoElement HTMLVideoElement;
typedef html::ImageData ImageData;

typedef unsigned long GLenum;
typedef boolean GLboolean;
typedef unsigned long GLbitfield;
```

```
typedef byte          GLbyte;      /* 'byte' should be a signed 8 bit type. */
typedef short         GLshort;
typedef long           GLint;
typedef long           GLsizei;
typedef long long      GLintptr;
typedef long long      GLsizeiptr;
typedef unsigned byte  GLubyte;    /* 'unsigned byte' should be an unsigned 8 bit type. */
typedef unsigned short GLushort;
typedef unsigned long  GLuint;
typedef float          GLfloat;
typedef float          GLclampf;
```

## WebGLContextAttributes

The `WebGLContextAttributes` interface contains drawing surface attributes and is passed as the second parameter to `getContext`. A native object may be supplied as this parameter; the specified attributes will be queried from this object.

```
[Callback]
interface WebGLContextAttributes {
    attribute boolean alpha;
    attribute boolean depth;
    attribute boolean stencil;
    attribute boolean antialias;
    attribute boolean premultipliedAlpha;
    attribute boolean preserveDrawingBuffer;
};
```

## Context creation parameters

The following list describes each attribute in the `WebGLContextAttributes` object and its use. For each attribute the default value is shown. The default value is used either if no second parameter is passed to `getContext`, or if a native object is passed which has no attribute of the given name.

### alpha

*Default: true*

. If the value is true, the drawing buffer has an alpha buffer for the purposes of performing OpenGL destination alpha operations and compositing with the page. If the value is false, no alpha buffer is available.

## depth

*Default: true*

. If the value is true, the drawing buffer has a depth buffer of at least 16 bits. If the value is false, no depth buffer is available.

## stencil

*Default: false*

. If the value is true, the drawing buffer has a stencil buffer of at least 8 bits. If the value is false, no stencil buffer is available.

## antialias

*Default: true*

. If the value is true and the implementation supports antialiasing the drawing buffer will perform antialiasing using its choice of technique (multisample/ supersample) and quality. If the value is false or the implementation does not support antialiasing, no antialiasing is performed.

## premultipliedAlpha

*Default: true*

. If the value is true the page compositor will assume the drawing buffer contains colors with premultiplied alpha. If the value is false the page compositor will assume that colors in the drawing buffer are not premultiplied. This flag is ignored if the **alpha** flag is false.

See [Premultiplied Alpha](#) for more information on the effects of the premultipliedAlpha flag.

## preserveDrawingBuffer

*Default: false*

. If false, once the drawing buffer is presented as described in the [Drawing Buffer](#) section, the contents of the drawing buffer are cleared to their default values. All elements of the drawing buffer (color, depth and stencil) are cleared. If the value is true the buffers will not be cleared and will preserve their values until cleared or overwritten by the author.

*On some hardware setting the  
preserveDrawingBuffer  
to true can have significant  
performance implications.*

*flag*

Here is an ECMAScript example which passes a WebGLContextAttributes argument to `getContext`. It assumes the presence of a canvas element named "canvas1" on the page.

```
var canvas = document.getElementById('canvas1');
var context = canvas.getContext('webgl',
    { antialias: false,
      stencil: true });
```

The WebGLObject interface is the parent interface for all GL objects.

```
interface WebGLObject {  
};
```

## WebGLBuffer

The WebGLBuffer interface represents an OpenGL Buffer Object. The underlying object is created as if by calling glGenBuffers ( [OpenGL ES 2.0 §2.9](#), [man page](#) ) , bound as if by calling glBindBuffer ( [OpenGL ES 2.0 §2.9](#), [man page](#) ) and destroyed as if by calling glDeleteBuffers ( [OpenGL ES 2.0 §2.9](#), [man page](#) ) .

```
interface WebGLBuffer : WebGLObject {  
};
```

## WebGLFramebuffer

The WebGLFramebuffer interface represents an OpenGL Framebuffer Object. The underlying object is created as if by calling glGenFramebuffers ( [OpenGL ES 2.0 §4.4.1](#), [man page](#) ) , bound as if by calling glBindFramebuffer ( [OpenGL ES 2.0 §4.4.1](#), [man page](#) ) and destroyed as if by calling glDeleteFramebuffers ( [OpenGL ES 2.0 §4.4.1](#), [man page](#) ) .

```
interface WebGLFramebuffer : WebGLObject {  
};
```

## WebGLProgram

The WebGLProgram interface represents an OpenGL Program Object. The underlying object is created as if by calling glCreateProgram ( [OpenGL ES 2.0 §2.10.3](#), [man page](#) ) , used as if by calling glUseProgram ( [OpenGL ES 2.0 §2.10.3](#), [page](#) ) and destroyed as if by calling glDeleteProgram ( [OpenGL ES 2.0 §2.10.3](#), [man page](#) ) .

```
interface WebGLProgram {  
};
```

```
: WebGLObject {
```

## WebGLRenderbuffer

The WebGLRenderbuffer interface represents an OpenGL Renderbuffer Object. The underlying object is created as if by calling glGenRenderbuffers ([OpenGL ES 2.0 §4.4.3 man page](#)) , bound as if by calling glBindRenderbuffer ([OpenGL ES 2.0 §4.4.3 man page](#)) and destroyed as if by calling glDeleteRenderbuffers ([OpenGL ES 2.0 §4.4.3 man page](#)) .

```
interface WebGLRenderbuffer {  
};
```

```
: WebGLObject {
```

## WebGLShader

The WebGLShader interface represents an OpenGL Shader Object. The underlying object is created as if by calling glCreateShader ([OpenGL ES 2.0 §2.10.1 man page](#)) , attached to a Program as if by calling glAttachShader ([OpenGL ES 2.0 §2.10.3 man page](#)) and destroyed as if by calling glDeleteShader ([OpenGL ES 2.0 §2.10.1 man page](#)) .

```
interface WebGLShader {  
};
```

```
: WebGLObject {
```

## WebGLTexture

The WebGLTexture interface represents an OpenGL Texture Object. The underlying object is created as if by calling glGenTextures ([OpenGL ES 2.0 §3.7.13 man page](#)) , bound as if by calling glBindTexture ([OpenGL ES 2.0 §3.7.13 man page](#)) and destroyed as if by calling glDeleteTextures ([OpenGL ES 2.0 §3.7.13 man page](#)) .

```
interface WebGLTexture {
```

```
: WebGLObject {
```

## WebGLUniformLocation

The WebGLUniformLocation interface represents the location of a uniform variable in a shader program.

```
interface WebGLUniformLocation {  
};
```

## WebGLActiveInfo

The WebGLActiveInfo interface represents the information returned from the `getActiveAttrib` and `getActiveUniform` calls.

```
interface WebGLActiveInfo {  
    readonly attribute GLint size;  
    readonly attribute GLenum type;  
    readonly attribute DOMString name;  
};
```

### Attributes

The following attributes are available:

#### **size of type GLint**

The size of the requested variable.

#### **type of type GLenum**

The data type of the requested variable.

#### **name of type DOMString**

The name of the requested variable.

## ArrayBuffer and Typed Arrays

Vertex, index, texture, and other data is transferred to the WebGL implementation using the [ArrayBuffer](#) and [views](#) defined in the [Typed Array](#) specification [\[TYPEDARRAYS\]](#).

Typed Arrays support the creation of interleaved, heterogeneous vertex data; uploading of distinct blocks of data into a large vertex buffer object; and most other use cases required by OpenGL programs.

Here is an ECMAScript example showing access to the same ArrayBuffer using different types of typed arrays. In this case the buffer contains a floating point vertex position (x, y, z) followed by a color as 4 unsigned bytes (r, g, b, a).

```
var numVertices = 100; // for example

// Compute the size needed for the buffer, in bytes and floats
var vertexSize = 3 * Float32Array.BYTES_PER_ELEMENT +
    4 * Uint8Array.BYTES_PER_ELEMENT;
var vertexSizeInFloats = vertexSize / Float32Array.BYTES_PER_ELEMENT;

// Allocate the buffer
var buf = new ArrayBuffer(numVertices * vertexSize);

// Map this buffer to a Float32Array to access the positions
var positionArray = new Float32Array(buf);

// Map the same buffer to a Uint8Array to access the color
var colorArray = new Uint8Array(buf);

// Set up the initial offset of the vertices and colors within the buffer
var positionIdx = 0;
var colorIdx = 3 * Float32Array.BYTES_PER_ELEMENT;

// Initialize the buffer
for (var i = 0; i < numVertices; i++) {
    positionArray[positionIdx] = ...;
    positionArray[positionIdx + 1] = ...;
```

```
positionArray[positionIdx + 2] = ...;
colorArray[colorIdx] = ...;
colorArray[colorIdx + 1] = ...;
colorArray[colorIdx + 2] = ...;
colorArray[colorIdx + 3] = ...;
positionIdx += vertexSizeInFloats;
colorIdx += vertexSize;
}
```

## The WebGL context

The `WebGLRenderingContext` represents the API allowing OpenGL ES 2.0 style rendering into the `canvas` element.

```
interface WebGLRenderingContext {  
  
    /* ClearBufferMask */  
    const GLenum DEPTH_BUFFER_BIT          = 0x00000100;  
    const GLenum STENCIL_BUFFER_BIT        = 0x00000400;  
    const GLenum COLOR_BUFFER_BIT          = 0x00004000;  
  
    /* BeginMode */  
    const GLenum POINTS                  = 0x0000;  
    const GLenum LINES                   = 0x0001;  
    const GLenum LINE_LOOP               = 0x0002;  
    const GLenum LINE_STRIP              = 0x0003;  
    const GLenum TRIANGLES              = 0x0004;  
    const GLenum TRIANGLE_STRIP         = 0x0005;  
    const GLenum TRIANGLE_FAN           = 0x0006;  
  
    /* AlphaFunction (not supported in ES20) */  
    /* NEVER */  
    /* LESS */  
    /* EQUAL */
```

```
/* LEQUAL */
/* GREATER */
/* NOTEQUAL */
/* GEQUAL */
/* ALWAYS */

/* BlendingFactorDest */
const GLenum ZERO = 0;
const GLenum ONE = 1;
const GLenum SRC_COLOR = 0x0300;
const GLenum ONE_MINUS_SRC_COLOR = 0x0301;
const GLenum SRC_ALPHA = 0x0302;
const GLenum ONE_MINUS_SRC_ALPHA = 0x0303;
const GLenum DST_ALPHA = 0x0304;
const GLenum ONE_MINUS_DST_ALPHA = 0x0305;

/* BlendingFactorSrc */
/* ZERO */
/* ONE */
const GLenum DST_COLOR = 0x0306;
const GLenum ONE_MINUS_DST_COLOR = 0x0307;
const GLenum SRC_ALPHA_SATURATE = 0x0308;
/* SRC_ALPHA */
/* ONE_MINUS_SRC_ALPHA */
/* DST_ALPHA */
/* ONE_MINUS_DST_ALPHA */

/* BlendEquationSeparate */
const GLenum FUNC_ADD = 0x8006;
const GLenum BLEND_EQUATION = 0x8009;
const GLenum BLEND_EQUATION_RGB = 0x8009; /* same as
BLEND_EQUATION */
const GLenum BLEND_EQUATION_ALPHA = 0x883D;

/* BlendSubtract */
```

```
const GLenum FUNC_SUBTRACT          = 0x800A;
const GLenum FUNC_REVERSE_SUBTRACT = 0x800B;

/* Separate Blend Functions */
const GLenum BLEND_DST_RGB          = 0x80C8;
const GLenum BLEND_SRC_RGB          = 0x80C9;
const GLenum BLEND_DST_ALPHA        = 0x80CA;
const GLenum BLEND_SRC_ALPHA        = 0x80CB;
const GLenum CONSTANT_COLOR         = 0x8001;
const GLenum ONE_MINUS_CONSTANT_COLOR = 0x8002;
const GLenum CONSTANT_ALPHA         = 0x8003;
const GLenum ONE_MINUS_CONSTANT_ALPHA = 0x8004;
const GLenum BLEND_COLOR           = 0x8005;

/* Buffer Objects */
const GLenum ARRAY_BUFFER           = 0x8892;
const GLenum ELEMENT_ARRAY_BUFFER    = 0x8893;
const GLenum ARRAY_BUFFER_BINDING    = 0x8894;
const GLenum ELEMENT_ARRAY_BUFFER_BINDING = 0x8895;

const GLenum STREAM_DRAW            = 0x88E0;
const GLenum STATIC_DRAW            = 0x88E4;
const GLenum DYNAMIC_DRAW           = 0x88E8;

const GLenum BUFFER_SIZE            = 0x8764;
const GLenum BUFFER_USAGE           = 0x8765;

const GLenum CURRENT_VERTEX_ATTRIB   = 0x8626;

/* CullFaceMode */
const GLenum FRONT                  = 0x0404;
const GLenum BACK                   = 0x0405;
const GLenum FRONT_AND_BACK         = 0x0408;

/* DepthFunction */
```

```
/* NEVER */
/* LESS */
/* EQUAL */
/* LEQUAL */
/* GREATER */
/* NOTEQUAL */
/* GEQUAL */
/* ALWAYS */

/* EnableCap */
/* TEXTURE_2D */
const GLenum CULL_FACE          = 0x0B44;
const GLenum BLEND              = 0x0BE2;
const GLenum DITHER              = 0xBD0;
const GLenum STENCIL_TEST        = 0xB90;
const GLenum DEPTH_TEST          = 0xB71;
const GLenum SCISSOR_TEST        = 0xC11;
const GLenum POLYGON_OFFSET_FILL = 0x8037;
const GLenum SAMPLE_ALPHA_TO_COVERAGE = 0x809E;
const GLenum SAMPLE_COVERAGE     = 0x80A0;

/* ErrorCode */
const GLenum NO_ERROR            = 0;
const GLenum INVALID_ENUM         = 0x0500;
const GLenum INVALID_VALUE        = 0x0501;
const GLenum INVALID_OPERATION    = 0x0502;
const GLenum OUT_OF_MEMORY        = 0x0505;

/* FrontFaceDirection */
const GLenum CW                  = 0x0900;
const GLenum CCW                 = 0x0901;

/* GetPName */
const GLenum LINE_WIDTH           = 0x0B21;
const GLenum ALIASED_POINT_SIZE_RANGE = 0x846D;
```

```
const GLenum ALIASED_LINE_WIDTH_RANGE      = 0x846E;
const GLenum CULL_FACE_MODE                = 0x0B45;
const GLenum FRONT_FACE                   = 0x0B46;
const GLenum DEPTH_RANGE                  = 0x0B70;
const GLenum DEPTH_WRITEMASK              = 0x0B72;
const GLenum DEPTH_CLEAR_VALUE             = 0x0B73;
const GLenum DEPTH_FUNC                   = 0x0B74;
const GLenum STENCIL_CLEAR_VALUE           = 0x0B91;
const GLenum STENCIL_FUNC                 = 0x0B92;
const GLenum STENCIL_FAIL                 = 0x0B94;
const GLenum STENCIL_PASS_DEPTH_FAIL     = 0x0B95;
const GLenum STENCIL_PASS_DEPTH_PASS     = 0x0B96;
const GLenum STENCIL_REF                  = 0x0B97;
const GLenum STENCIL_VALUE_MASK            = 0x0B93;
const GLenum STENCIL_WRITEMASK             = 0x0B98;
const GLenum STENCIL_BACK_FUNC             = 0x8800;
const GLenum STENCIL_BACK_FAIL             = 0x8801;
const GLenum STENCIL_BACK_PASS_DEPTH_FAIL = 0x8802;
const GLenum STENCIL_BACK_PASS_DEPTH_PASS = 0x8803;
const GLenum STENCIL_BACK_REF              = 0x8CA3;
const GLenum STENCIL_BACK_VALUE_MASK       = 0x8CA4;
const GLenum STENCIL_BACK_WRITEMASK         = 0x8CA5;
const GLenum VIEWPORT                     = 0x0BA2;
const GLenum SCISSOR_BOX                  = 0x0C10;
/*   SCISSOR_TEST */
const GLenum COLOR_CLEAR_VALUE             = 0x0C22;
const GLenum COLOR_WRITEMASK               = 0x0C23;
const GLenum UNPACK_ALIGNMENT              = 0x0CF5;
const GLenum PACK_ALIGNMENT                = 0x0D05;
const GLenum MAX_TEXTURE_SIZE              = 0x0D33;
const GLenum MAX_VIEWPORT_DIMS             = 0x0D3A;
const GLenum SUBPIXEL_BITS                 = 0x0D50;
const GLenum RED_BITS                      = 0x0D52;
const GLenum GREEN_BITS                    = 0x0D53;
const GLenum BLUE_BITS                     = 0x0D54;
```

```
const GLenum ALPHA_BITS          = 0x0D55;
const GLenum DEPTH_BITS         = 0x0D56;
const GLenum STENCIL_BITS        = 0x0D57;
const GLenum POLYGON_OFFSET_UNITS     = 0x2A00;
/*    POLYGON_OFFSET_FILL */
const GLenum POLYGON_OFFSET_FACTOR   = 0x8038;
const GLenum TEXTURE_BINDING_2D      = 0x8069;
const GLenum SAMPLE_BUFFERS        = 0x80A8;
const GLenum SAMPLES               = 0x80A9;
const GLenum SAMPLE_COVERAGE_VALUE   = 0x80AA;
const GLenum SAMPLE_COVERAGE_INVERT   = 0x80AB;

/* GetTextureParameter */
/*    TEXTURE_MAG_FILTER */
/*    TEXTURE_MIN_FILTER */
/*    TEXTURE_WRAP_S */
/*    TEXTURE_WRAP_T */

const GLenum NUM_COMPRESSED_TEXTURE_FORMATS = 0x86A2;
const GLenum COMPRESSED_TEXTURE_FORMATS     = 0x86A3;

/* HintMode */
const GLenum DONT_CARE           = 0x1100;
const GLenum FASTEST             = 0x1101;
const GLenum NICEST              = 0x1102;

/* HintTarget */
const GLenum GENERATE_MIPMAP_HINT     = 0x8192;

/* DataType */
const GLenum BYTE                = 0x1400;
const GLenum UNSIGNED_BYTE       = 0x1401;
const GLenum SHORT               = 0x1402;
const GLenum UNSIGNED_SHORT      = 0x1403;
const GLenum INT                 = 0x1404;
```

```
const GLenum UNSIGNED_INT          = 0x1405;
const GLenum FLOAT                 = 0x1406;

/* PixelFormat */
const GLenum DEPTH_COMPONENT      = 0x1902;
const GLenum ALPHA                = 0x1906;
const GLenum RGB                  = 0x1907;
const GLenum RGBA                 = 0x1908;
const GLenum LUMINANCE            = 0x1909;
const GLenum LUMINANCE_ALPHA      = 0x190A;

/* PixelType */
/* UNSIGNED_BYTE */
const GLenum UNSIGNED_SHORT_4_4_4_4 = 0x8033;
const GLenum UNSIGNED_SHORT_5_5_5_1 = 0x8034;
const GLenum UNSIGNED_SHORT_5_6_5   = 0x8363;

/* Shaders */
const GLenum FRAGMENT_SHADER      = 0x8B30;
const GLenum VERTEX_SHADER         = 0x8B31;
const GLenum MAX_VERTEX_ATTRIBS    = 0x8869;
const GLenum MAX_VERTEX_UNIFORM_VECTORS = 0x8DFB;
const GLenum MAX_VARYING_VECTORS   = 0x8DFC;
const GLenum MAX_COMBINED_TEXTURE_IMAGE_UNITS = 0x8B4D;
const GLenum MAX_VERTEX_TEXTURE_IMAGE_UNITS = 0x8B4C;
const GLenum MAX_TEXTURE_IMAGE_UNITS   = 0x8872;
const GLenum MAX_FRAGMENT_UNIFORM_VECTORS = 0x8DFD;
const GLenum SHADER_TYPE           = 0x8B4F;
const GLenum DELETE_STATUS          = 0x8B80;
const GLenum LINK_STATUS            = 0x8B82;
const GLenum VALIDATE_STATUS        = 0x8B83;
const GLenum ATTACHED_SHADERS       = 0x8B85;
const GLenum ACTIVE_UNIFORMS        = 0x8B86;
const GLenum ACTIVE_ATTRIBUTES       = 0x8B89;
const GLenum SHADING_LANGUAGE_VERSION = 0x8B8C;
```

```
const GLenum CURRENT_PROGRAM = 0x8B8D;

/* StencilFunction */
const GLenum NEVER = 0x0200;
const GLenum LESS = 0x0201;
const GLenum EQUAL = 0x0202;
const GLenum LEQUAL = 0x0203;
const GLenum GREATER = 0x0204;
const GLenum NOTEQUAL = 0x0205;
const GLenum GEQUAL = 0x0206;
const GLenum ALWAYS = 0x0207;

/* StencilOp */
/*      ZERO */
const GLenum KEEP = 0x1E00;
const GLenum REPLACE = 0x1E01;
const GLenum INCR = 0x1E02;
const GLenum DECR = 0x1E03;
const GLenum INVERT = 0x150A;
const GLenum INCR_WRAP = 0x8507;
const GLenum DECR_WRAP = 0x8508;

/* StringName */
const GLenum VENDOR = 0x1F00;
const GLenum RENDERER = 0x1F01;
const GLenum VERSION = 0x1F02;

/* TextureMagFilter */
const GLenum NEAREST = 0x2600;
const GLenum LINEAR = 0x2601;

/* TextureMinFilter */
/*      NEAREST */
/*      LINEAR */
const GLenum NEAREST_MIPMAP_NEAREST = 0x2700;
```

```
const GLenum LINEAR_MIPMAP_NEAREST      = 0x2701;
const GLenum NEAREST_MIPMAP_LINEAR       = 0x2702;
const GLenum LINEAR_MIPMAP_LINEAR        = 0x2703;

/* TextureParameterName */
const GLenum TEXTURE_MAG_FILTER         = 0x2800;
const GLenum TEXTURE_MIN_FILTER         = 0x2801;
const GLenum TEXTURE_WRAP_S             = 0x2802;
const GLenum TEXTURE_WRAP_T             = 0x2803;

/* TextureTarget */
const GLenum TEXTURE_2D                 = 0x0DE1;
const GLenum TEXTURE                   = 0x1702;

const GLenum TEXTURE_CUBE_MAP          = 0x8513;
const GLenum TEXTURE_BINDING_CUBE_MAP   = 0x8514;
const GLenum TEXTURE_CUBE_MAP_POSITIVE_X = 0x8515;
const GLenum TEXTURE_CUBE_MAP_NEGATIVE_X = 0x8516;
const GLenum TEXTURE_CUBE_MAP_POSITIVE_Y = 0x8517;
const GLenum TEXTURE_CUBE_MAP_NEGATIVE_Y = 0x8518;
const GLenum TEXTURE_CUBE_MAP_POSITIVE_Z = 0x8519;
const GLenum TEXTURE_CUBE_MAP_NEGATIVE_Z = 0x851A;
const GLenum MAX_CUBE_MAP_TEXTURE_SIZE = 0x851C;

/* TextureUnit */
const GLenum TEXTURE0                  = 0x84C0;
const GLenum TEXTURE1                  = 0x84C1;
const GLenum TEXTURE2                  = 0x84C2;
const GLenum TEXTURE3                  = 0x84C3;
const GLenum TEXTURE4                  = 0x84C4;
const GLenum TEXTURE5                  = 0x84C5;
const GLenum TEXTURE6                  = 0x84C6;
const GLenum TEXTURE7                  = 0x84C7;
const GLenum TEXTURE8                  = 0x84C8;
const GLenum TEXTURE9                  = 0x84C9;
```

```
const GLenum TEXTURE10          = 0x84CA;
const GLenum TEXTURE11          = 0x84CB;
const GLenum TEXTURE12          = 0x84CC;
const GLenum TEXTURE13          = 0x84CD;
const GLenum TEXTURE14          = 0x84CE;
const GLenum TEXTURE15          = 0x84CF;
const GLenum TEXTURE16          = 0x84D0;
const GLenum TEXTURE17          = 0x84D1;
const GLenum TEXTURE18          = 0x84D2;
const GLenum TEXTURE19          = 0x84D3;
const GLenum TEXTURE20          = 0x84D4;
const GLenum TEXTURE21          = 0x84D5;
const GLenum TEXTURE22          = 0x84D6;
const GLenum TEXTURE23          = 0x84D7;
const GLenum TEXTURE24          = 0x84D8;
const GLenum TEXTURE25          = 0x84D9;
const GLenum TEXTURE26          = 0x84DA;
const GLenum TEXTURE27          = 0x84DB;
const GLenum TEXTURE28          = 0x84DC;
const GLenum TEXTURE29          = 0x84DD;
const GLenum TEXTURE30          = 0x84DE;
const GLenum TEXTURE31          = 0x84DF;
const GLenum ACTIVE_TEXTURE      = 0x84E0;
```

*/\* TextureWrapMode \*/*

```
const GLenum REPEAT              = 0x2901;
const GLenum CLAMP_TO_EDGE        = 0x812F;
const GLenum MIRRORED_REPEAT     = 0x8370;
```

*/\* Uniform Types \*/*

```
const GLenum FLOAT_VEC2          = 0x8B50;
const GLenum FLOAT_VEC3          = 0x8B51;
const GLenum FLOAT_VEC4          = 0x8B52;
const GLenum INT_VEC2            = 0x8B53;
const GLenum INT_VEC3            = 0x8B54;
```

```
const GLenum INT_VEC4          = 0x8B55;
const GLenum BOOL              = 0x8B56;
const GLenum BOOL_VEC2         = 0x8B57;
const GLenum BOOL_VEC3         = 0x8B58;
const GLenum BOOL_VEC4         = 0x8B59;
const GLenum FLOAT_MAT2        = 0x8B5A;
const GLenum FLOAT_MAT3        = 0x8B5B;
const GLenum FLOAT_MAT4        = 0x8B5C;
const GLenum SAMPLER_2D         = 0x8B5E;
const GLenum SAMPLER_CUBE        = 0x8B60;

/* Vertex Arrays */
const GLenum VERTEX_ATTRIB_ARRAY_ENABLED      = 0x8622;
const GLenum VERTEX_ATTRIB_ARRAY_SIZE          = 0x8623;
const GLenum VERTEX_ATTRIB_ARRAY_STRIDE         = 0x8624;
const GLenum VERTEX_ATTRIB_ARRAY_TYPE          = 0x8625;
const GLenum VERTEX_ATTRIB_ARRAY_NORMALIZED     = 0x886A;
const GLenum VERTEX_ATTRIB_ARRAY_POINTER        = 0x8645;
const GLenum VERTEX_ATTRIB_ARRAY_BUFFER_BINDING = 0x889F;

/* Shader Source */
const GLenum COMPILE_STATUS           = 0x8B81;

/* Shader Precision-Specified Types */
const GLenum LOW_FLOAT                = 0x8DF0;
const GLenum MEDIUM_FLOAT             = 0x8DF1;
const GLenum HIGH_FLOAT               = 0x8DF2;
const GLenum LOW_INT                 = 0x8DF3;
const GLenum MEDIUM_INT              = 0x8DF4;
const GLenum HIGH_INT                = 0x8DF5;

/* Framebuffer Object. */
const GLenum FRAMEBUFFER            = 0x8D40;
const GLenum RENDERBUFFER            = 0x8D41;
```

```
const GLenum RGBA4          = 0x8056;
const GLenum RGB5_A1        = 0x8057;
const GLenum RGB565         = 0x8D62;
const GLenum DEPTH_COMPONENT16 = 0x81A5;
const GLenum STENCIL_INDEX    = 0x1901;
const GLenum STENCIL_INDEX8   = 0x8D48;
const GLenum DEPTH_STENCIL    = 0x84F9;

const GLenum RENDERBUFFER_WIDTH      = 0x8D42;
const GLenum RENDERBUFFER_HEIGHT     = 0x8D43;
const GLenum RENDERBUFFER_INTERNAL_FORMAT = 0x8D44;
const GLenum RENDERBUFFER_RED_SIZE    = 0x8D50;
const GLenum RENDERBUFFER_GREEN_SIZE  = 0x8D51;
const GLenum RENDERBUFFER_BLUE_SIZE   = 0x8D52;
const GLenum RENDERBUFFER_ALPHA_SIZE  = 0x8D53;
const GLenum RENDERBUFFER_DEPTH_SIZE  = 0x8D54;
const GLenum RENDERBUFFER_STENCIL_SIZE = 0x8D55;

const GLenum FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE      = 0x8CD0;
const GLenum FRAMEBUFFER_ATTACHMENT_OBJECT_NAME       = 0x8CD1;
const GLenum FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL     = 0x8CD2;
const GLenum FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE = 0x8CD3;

const GLenum COLOR_ATTACHMENT0      = 0x8CE0;
const GLenum DEPTH_ATTACHMENT       = 0x8D00;
const GLenum STENCIL_ATTACHMENT     = 0x8D20;
const GLenum DEPTH_STENCIL_ATTACHMENT = 0x821A;

const GLenum NONE          = 0;

const GLenum FRAMEBUFFER_COMPLETE      = 0x8CD5;
const GLenum FRAMEBUFFER_INCOMPLETE_ATTACHMENT = 0x8CD6;
const GLenum FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT = 0x8CD7;
const GLenum FRAMEBUFFER_INCOMPLETE_DIMENSIONS      = 0x8CD9;
const GLenum FRAMEBUFFER_UNSUPPORTED           = 0x8CDD;
```

```
const GLenum FRAMEBUFFER_BINDING          = 0x8CA6;
const GLenum RENDERBUFFER_BINDING         = 0x8CA7;
const GLenum MAX_RENDERBUFFER_SIZE       = 0x84E8;

const GLenum INVALID_FRAMEBUFFER_OPERATION = 0x0506;

/* WebGL-specific enums */
const GLenum UNPACK_FLIP_Y_WEBGL        = 0x9240;
const GLenum UNPACK_PREMULTIPLY_ALPHA_WEBGL = 0x9241;
const GLenum CONTEXT_LOST_WEBGL         = 0x9242;
const GLenum UNPACK_COLORSPACE_CONVERSION_WEBGL = 0x9243;
const GLenum BROWSER_DEFAULT_WEBGL      = 0x9244;

readonly attribute HTMLCanvasElement canvas;
readonly attribute GLsizei drawingBufferWidth;
readonly attribute GLsizei drawingBufferHeight;

WebGLContextAttributes getContextAttributes();
boolean isContextLost();

DOMString[ ] getSupportedExtensions();
objectgetExtension(DOMString name);

void activeTexture(GLenum texture);
void attachShader(WebGLProgram program, WebGLShader shader);
void bindAttribLocation(WebGLProgram program, GLuint index, DOMString name);
void bindBuffer(GLenum target, WebGLBuffer buffer);
void bindFramebuffer(GLenum target, WebGLFramebuffer framebuffer);
void bindRenderbuffer(GLenum target, WebGLRenderbuffer renderbuffer);
void bindTexture(GLenum target, WebGLTexture texture);
void blendColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void blendEquation(GLenum mode);
void blendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);
void blendFunc(GLenum sfactor, GLenum dfactor);
```

```
void blendFuncSeparate(GLenum srcRGB, GLenum dstRGB,  
                      GLenum srcAlpha, GLenum dstAlpha);
```



```
void bufferData(GLenum target, GLsizeiptr size, GLenum usage);  
void bufferData(GLenum target, ArrayBufferView data, GLenum usage);  
void bufferData(GLenum target, ArrayBuffer data, GLenum usage);  
void bufferSubData(GLenum target, GLintptr offset, ArrayBufferView data);  
void bufferSubData(GLenum target, GLintptr offset, ArrayBuffer data);  
  
GLenum checkFramebufferStatus(GLenum target);  
void clear(GLbitfield mask);  
void clearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);  
void clearDepth(GLclampf depth);  
void clearStencil(GLint s);  
void colorMask(GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha);  
void compileShader(WebGLShader shader);  
  
void copyTexImage2D(GLenum target, GLint level, GLenum internalformat,  
                   GLint x, GLint y, GLsizei width, GLsizei height,  
                   GLint border);  
void copyTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset,  
                      GLint x, GLint y, GLsizei width, GLsizei height);  
  
WebGLBuffer createBuffer();  
WebGLFramebuffer createFramebuffer();  
WebGLProgram createProgram();  
WebGLRenderbuffer createRenderbuffer();  
WebGLShader createShader(GLenum type);  
WebGLTexture createTexture();  
  
void cullFace(GLenum mode);  
  
void deleteBuffer(WebGLBuffer buffer);  
void deleteFramebuffer(WebGLFramebuffer framebuffer);  
void deleteProgram(WebGLProgram program);
```

```
void deleteRenderbuffer(WebGLRenderbuffer renderbuffer);
void deleteShader(WebGLShader shader);
void deleteTexture(WebGLTexture texture);

void depthFunc(GLenum func);
void depthMask(GLboolean flag);
void depthRange(GLclampf zNear, GLclampf zFar);
void detachShader(WebGLProgram program, WebGLShader shader);
void disable(GLenum cap);
void disableVertexAttribArray(GLuint index);
void drawArrays(GLenum mode, GLint first, GLsizei count);
void drawElements(GLenum mode, GLsizei count, GLenum type, GLintptr offset);

void enable(GLenum cap);
void enableVertexAttribArray(GLuint index);
void finish();
void flush();
void framebufferRenderbuffer(GLenum target, GLenum attachment,
                           GLenum renderbuffertarget,
                           WebGLRenderbuffer renderbuffer);
void framebufferTexture2D(GLenum target, GLenum attachment, GLenum textarget,
                         WebGLTexture texture, GLint level);
void frontFace(GLenum mode);

void generateMipmap(GLenum target);

WebGLActiveInfo getActiveAttrib(WebGLProgram program, GLuint index);
WebGLActiveInfo getActiveUniform(WebGLProgram program, GLuint index);
WebGLShader[ ] getAttachedShaders(WebGLProgram program);

GLint getAttribLocation(WebGLProgram program, DOMString name);

any getParameter(GLenum pname);
any getBufferParameter(GLenum target, GLenum pname);
```

```
GLenum getError();

any getFramebufferAttachmentParameter(GLenum target, GLenum attachment,
                                     GLenum pname);
any getProgramParameter(WebGLProgram program, GLenum pname);
DOMString getProgramInfoLog(WebGLProgram program);
any getRenderbufferParameter(GLenum target, GLenum pname);
any getShaderParameter(WebGLShader shader, GLenum pname);
DOMString getShaderInfoLog(WebGLShader shader);

DOMString getShaderSource(WebGLShader shader);

any getTexParameter(GLenum target, GLenum pname);

any getUniform(WebGLProgram program, WebGLUniformLocation location);

WebGLUniformLocation getUniformLocation(WebGLProgram program, DOMString name);

any getVertexAttrib(GLuint index, GLenum pname);

GLsizeiptr getVertexAttribOffset(GLuint index, GLenum pname);

void hint(GLenum target, GLenum mode);
GLboolean isBuffer(WebGLBuffer buffer);
GLboolean isEnabled(GLenum cap);
GLboolean isFramebuffer(WebGLFramebuffer framebuffer);
GLboolean isProgram(WebGLProgram program);
GLboolean isRenderbuffer(WebGLRenderbuffer renderbuffer);
GLboolean isShader(WebGLShader shader);
GLboolean isTexture(WebGLTexture texture);
void lineWidth(GLfloat width);
void linkProgram(WebGLProgram program);
void pixelStorei(GLenum pname, GLint param);
void polygonOffset(GLfloat factor, GLfloat units);
```

```
void readPixels(GLint x, GLint y, GLsizei width, GLsizei height,
               GLenum format, GLenum type, ArrayBufferView pixels);

void renderbufferStorage(GLenum target, GLenum internalformat,
                       GLsizei width, GLsizei height);
void sampleCoverage(GLclampf value, GLboolean invert);
void scissor(GLint x, GLint y, GLsizei width, GLsizei height);

void shaderSource(WebGLShader shader, DOMString source);

void stencilFunc(GLenum func, GLint ref, GLuint mask);
void stencilFuncSeparate(GLenum face, GLenum func, GLint ref, GLuint mask);
void stencilMask(GLuint mask);
void stencilMaskSeparate(GLenum face, GLuint mask);
void stencilOp(GLenum fail, GLenum zfail, GLenum zpass);
void stencilOpSeparate(GLenum face, GLenum fail, GLenum zfail, GLenum zpass);

void texImage2D(GLenum target, GLint level, GLenum internalformat,
               GLsizei width, GLsizei height, GLint border, GLenum format,
               GLenum type, ArrayBufferView pixels);
void texImage2D(GLenum target, GLint level, GLenum internalformat,
               GLenum format, GLenum type, ImageData pixels);
void texImage2D(GLenum target, GLint level, GLenum internalformat,
               GLenum format, GLenum type, HTMLImageElement image)
raises (DOMException);
void texImage2D(GLenum target, GLint level, GLenum internalformat,
               GLenum format, GLenum type, HTMLCanvasElement canvas)
raises (DOMException);
void texImage2D(GLenum target, GLint level, GLenum internalformat,
               GLenum format, GLenum type, HTMLVideoElement video) raises (DOMException);

void texParameterf(GLenum target, GLenum pname, GLfloat param);
void texParameteri(GLenum target, GLenum pname, GLint param);

void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset,
```

```
    GLsizei width, GLsizei height,
    GLenum format, GLenum type, ArrayBufferView pixels);
void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset,
                  GLenum format, GLenum type, ImageData pixels);
void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset,
                  GLenum format, GLenum type, HTMLImageElement image)
raises (DOMException);
void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset,
                  GLenum format, GLenum type, HTMLCanvasElement canvas)
raises (DOMException);
void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset,
                  GLenum format, GLenum type, HTMLVideoElement video)
raises (DOMException);

void uniform1f(WebGLUniformLocation location, GLfloat x);
void uniform1fv(WebGLUniformLocation location, Float32Array v);
void uniform1fv(WebGLUniformLocation location, float[] v);
void uniform1i(WebGLUniformLocation location, GLint x);
void uniform1iv(WebGLUniformLocation location, Int32Array v);
void uniform1iv(WebGLUniformLocation location, long[] v);
void uniform2f(WebGLUniformLocation location, GLfloat x, GLfloat y);
void uniform2fv(WebGLUniformLocation location, Float32Array v);
void uniform2fv(WebGLUniformLocation location, float[] v);
void uniform2i(WebGLUniformLocation location, GLint x, GLint y);
void uniform2iv(WebGLUniformLocation location, Int32Array v);
void uniform2iv(WebGLUniformLocation location, long[] v);
void uniform3f(WebGLUniformLocation location, GLfloat x, GLfloat y, GLfloat z);
void uniform3fv(WebGLUniformLocation location, Float32Array v);
void uniform3fv(WebGLUniformLocation location, float[] v);
void uniform3i(WebGLUniformLocation location, GLint x, GLint y, GLint z);
void uniform3iv(WebGLUniformLocation location, Int32Array v);
void uniform3iv(WebGLUniformLocation location, long[] v);
void uniform4f(WebGLUniformLocation location, GLfloat x, GLfloat y, GLfloat z, GLfloat w);
void uniform4fv(WebGLUniformLocation location, Float32Array v);
void uniform4fv(WebGLUniformLocation location, float[] v);
```

```
void uniform4i(WebGLUniformLocation location, GLint x, GLint y, GLint z, GLint w);
void uniform4iv(WebGLUniformLocation location, Int32Array v);
void uniform4iv(WebGLUniformLocation location, long[] v);

void uniformMatrix2fv(WebGLUniformLocation location, GLboolean transpose,
                      Float32Array value);
void uniformMatrix2fv(WebGLUniformLocation location, GLboolean transpose,
                      float[] value);
void uniformMatrix3fv(WebGLUniformLocation location, GLboolean transpose,
                      Float32Array value);
void uniformMatrix3fv(WebGLUniformLocation location, GLboolean transpose,
                      float[] value);
void uniformMatrix4fv(WebGLUniformLocation location, GLboolean transpose,
                      Float32Array value);
void uniformMatrix4fv(WebGLUniformLocation location, GLboolean transpose,
                      float[] value);

void useProgram(WebGLProgram program);
void validateProgram(WebGLProgram program);

void vertexAttrib1f(GLuint indx, GLfloat x);
void vertexAttrib1fv(GLuint indx, Float32Array values);
void vertexAttrib1fv(GLuint indx, float[] values);
void vertexAttrib2f(GLuint indx, GLfloat x, GLfloat y);
void vertexAttrib2fv(GLuint indx, Float32Array values);
void vertexAttrib2fv(GLuint indx, float[] values);
void vertexAttrib3f(GLuint indx, GLfloat x, GLfloat y, GLfloat z);
void vertexAttrib3fv(GLuint indx, Float32Array values);
void vertexAttrib3fv(GLuint indx, float[] values);
void vertexAttrib4f(GLuint indx, GLfloat x, GLfloat y, GLfloat z, GLfloat w);
void vertexAttrib4fv(GLuint indx, Float32Array values);
void vertexAttrib4fv(GLuint indx, float[] values);
void vertexAttribPointer(GLuint indx, GLint size, GLenum type,
                        GLboolean normalized, GLsizei stride, GLintptr offset);
```

```
    void viewport(GLint x, GLint y, GLsizei width, GLsizei height);  
};
```



## Attributes

The following attributes are available:

### **canvas of type HTMLCanvasElement**

A reference to the canvas element which created this context.

### **drawingBufferWidth of type GLsizei**

The actual width of the drawing buffer. May be different from the `width` attribute of the `HTMLCanvasElement` if the implementation is unable to satisfy the requested width or height.

### **drawingBufferHeight of type GLsizei**

The actual height of the drawing buffer. May be different from the `height` attribute of the `HTMLCanvasElement` if the implementation is unable to satisfy the requested width or height.

## Getting information about the context

### **WebGLContextAttributes getContextAttributes()**

Returns the `WebGLContextAttributes` describing the current drawing buffer.

## Setting and getting state

OpenGL ES 2.0 maintains state values for use in rendering. All the calls in this group behave identically to their OpenGL counterparts unless otherwise noted.

**void activeTexture(GLenum texture)** ( [OpenGL ES](#)

[2.0 §3.7](#)

, [man page](#)



) ([OpenGL ES 2.0 §4.1.6](#), [man page](#))

**void blendEquation(GLenum mode)** ( [OpenGL ES 2.0](#)

[§4.1.6](#)

, [man page](#)

) ([OpenGL ES 2.0 §4.1.6](#), [man page](#))

**void blendFunc(GLenum sfactor, GLenum dfactor)** ( [OpenGL](#)

[ES 2.0 §4.1.6](#)

, [man page](#)

See [Blending With Constant Color](#) for limitations imposed by WebGL.

**void blendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha)** ( [OpenGL ES 2.0](#)

[§4.1.6](#)

, [man page](#)

See [Blending With Constant Color](#) for limitations imposed by WebGL.

**void clearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)** ( [OpenGL ES 2.0 §4.2.3](#)

, [man page](#)

) ([OpenGL ES 2.0](#)

[§4.2.3](#)

, [man page](#)

depth value is clamped to the range 0 to 1.

**void clearStencil(GLint s)** ( [OpenGL ES 2.0 §4.2.3](#)

, [man page](#)

) ([OpenGL ES 2.0 §4.2.2](#)

, [man page](#)

**void cullFace(GLenum mode)** ( [OpenGL ES 2.0 §3.5.1](#)

, [man page](#)

**void depthFunc(GLenum func)** ( [OpenGL ES 2.0](#) )

[§4.1.5](#), [man page](#)

**void depthMask(GLboolean flag)** ( [OpenGL ES 2.0](#) )

[§4.2.2](#), [man page](#)

**void depthRange(GLclampf zNear, GLclampf zFar)** ( [OpenGL](#) )

[ES 2.0 §2.12.1](#), [man page](#)

zNear and zFar values are clamped to the range 0 to 1 and zNear must be less than or equal to zFar; see [Viewport Depth Range](#).

**void disable(GLenum cap)** ( [man page](#) )

**void enable(GLenum cap)** ( [man page](#) )

**void frontFace(GLenum mode)** ( [OpenGL ES 2.0](#) )

[§3.5.1](#), [man page](#)

**any getParameter(GLenum pname)** ([glGet OpenGL ES 2.0](#))

[man page](#)) ([glGetString OpenGL ES 2.0](#))

[man page](#))

Return the value for the passed pname. The type returned is the natural type for the requested pname, as given in the following table:

pname	returned type
ACTIVE_TEXTURE	unsigned long
ALIASED_LINE_WIDTH_RANGE	Float32Array (with 2 elements)
ALIASED_POINT_SIZE_RANGE	Float32Array (with 2 elements)
ALPHA_BITS	long
ARRAY_BUFFER_BINDING	WebGLBuffer
BLEND	boolean
BLEND_COLOR	Float32Array (with 4 values)
BLEND_DST_ALPHA	unsigned long
BLEND_DST_RGB	unsigned long
BLEND_EQUATION_ALPHA	unsigned long

BLEND_EQUATION_RGB	unsigned long
BLEND_SRC_ALPHA	unsigned long
BLEND_SRC_RGB	unsigned long
BLUE_BITS	long
COLOR_CLEAR_VALUE	Float32Array (with 4 values)
COLOR_WRITEMASK	boolean[] (with 4 values)
COMPRESSED_TEXTURE_FORMATS	null
CULL_FACE	boolean
CULL_FACE_MODE	unsigned long
CURRENT_PROGRAM	WebGLProgram
DEPTH_BITS	long
DEPTH_CLEAR_VALUE	float
DEPTH_FUNC	unsigned long
DEPTH_RANGE	Float32Array (with 2 elements)
DEPTH_TEST	boolean
DEPTH_WRITEMASK	boolean
DITHER	boolean
ELEMENT_ARRAY_BUFFER_BINDING	WebGLBuffer
FRAMEBUFFER_BINDING	WebGLFramebuffer
FRONT_FACE	unsigned long
GENERATE_MIPMAP_HINT	unsigned long
GREEN_BITS	long
LINE_WIDTH	float
MAX_COMBINED_TEXTURE_IMAGE_UNITS	long
MAX_CUBE_MAP_TEXTURE_SIZE	long
MAX_FRAGMENT_UNIFORM_VECTORS	long
MAX_RENDERBUFFER_SIZE	long
MAX_TEXTURE_IMAGE_UNITS	long
MAX_TEXTURE_SIZE	long
MAX_VARYING_VECTORS	long
MAX_VERTEX_ATTRIBS	long

MAX_VERTEX_TEXTURE_IMAGE_UNITS	long
MAX_VERTEX_UNIFORM_VECTORS	long
MAX_VIEWPORT_DIMS	Int32Array (with 2 elements)
NUM_COMPRESSED_TEXTURE_FORMATS	long
PACK_ALIGNMENT	long
POLYGON_OFFSET_FACTOR	float
POLYGON_OFFSET_FILL	boolean
POLYGON_OFFSET_UNITS	float
RED_BITS	long
RENDERBUFFER_BINDING	WebGLRenderbuffer
RENDERER	DOMString
SAMPLE_BUFFERS	long
SAMPLE_COVERAGE_INVERT	boolean
SAMPLE_COVERAGE_VALUE	float
SAMPLES	long
SCISSOR_BOX	Int32Array (with 4 elements)
SCISSOR_TEST	boolean
SHADING_LANGUAGE_VERSION	DOMString
STENCIL_BACK_FAIL	unsigned long
STENCIL_BACK_FUNC	unsigned long
STENCIL_BACK_PASS_DEPTH_FAIL	unsigned long
STENCIL_BACK_PASS_DEPTH_PASS	unsigned long
STENCIL_BACK_REF	long
STENCIL_BACK_VALUE_MASK	unsigned long
STENCIL_BACK_WRITEMASK	unsigned long
STENCIL_BITS	long
STENCIL_CLEAR_VALUE	long
STENCIL_FAIL	unsigned long
STENCIL_FUNC	unsigned long
STENCIL_PASS_DEPTH_FAIL	unsigned long
STENCIL_PASS_DEPTH_PASS	unsigned long

STENCIL_REF	long
STENCIL_TEST	boolean
STENCIL_VALUE_MASK	unsigned long
STENCIL_WRITEMASK	unsigned long
SUBPIXEL_BITS	long
TEXTURE_BINDING_2D	WebGLTexture
TEXTURE_BINDING_CUBE_MAP	WebGLTexture
UNPACK_ALIGNMENT	int
UNPACK_COLORSPACE_CONVERSION_WEBGL	unsigned long
UNPACK_FLIP_Y_WEBGL	boolean
UNPACK_PREMULTIPLY_ALPHA_WEBGL	boolean
VENDOR	DOMString
VERSION	DOMString
VIEWPORT	Int32Array (with 4 elements)

The following *pname* arguments return a string describing some aspect of the current WebGL implementation:

VERSION	Returns a version or release number of the form WebGL<space>1.0<space><vendor-specific information>.
SHADING_LANGUAGE_VERSION	Returns a version or release number of the form WebGL<space>GLSL<space>ES<space>1.0<space><vendor-specific information>.
VENDOR	Returns the company responsible for this WebGL implementation. This name does not change from release to release.
RENDERER	Returns the name of the renderer. This name is typically specific to a particular configuration of a hardware platform. It does not change from release to release.

See [Extension Queries](#) for information on querying the available extensions in the current WebGL implementation.

## GLenum **getError()** ( [OpenGL ES 2.0 §2.5](#) )

[man page](#)

See [WebGLContextEvent](#) for documentation of a WebGL-specific return value from `getError`.

**void hint(GLenum target, GLenum mode)** ( [OpenGL ES](#)

[2.0 §5.2](#), [man page](#))

**GLboolean isEnabled(GLenum cap)** ( [OpenGL ES](#)

[2.0 §6.1.1](#), [man page](#))

**void lineWidth(GLfloat width)** ( [OpenGL ES 2.0 §3.4](#)

[man page](#))

**void pixelStorei(GLenum pname, GLint param)** ( [OpenGL ES](#)

[2.0 §3.6.1](#), [man page](#))

In addition to the parameters in the OpenGL ES 2.0 specification, the WebGL specification accepts the parameters UNPACK\_FLIP\_Y\_WEBGL, UNPACK\_PREMULTIPLY\_ALPHA\_WEBGL and UNPACK\_COLORSPACE\_CONVERSION\_WEBGL. See [Pixel Storage Parameters](#) for documentation of these parameters.

**void polygonOffset(GLfloat factor, GLfloat units)** ( [OpenGL ES](#)

[2.0 §3.5.2](#), [man page](#))

**void sampleCoverage(GLclampf value, GLboolean invert)** ( [OpenGL ES](#)

[2.0 §4.1.3](#), [man page](#))

**void stencilFunc(GLenum func, GLint ref, GLuint mask)** ( [OpenGL](#)

[ES 2.0 §4.1.4](#), [man page](#))

**void stencilFuncSeparate(GLenum face, GLenum func, GLint ref, GLuint mask)** ( [OpenGL ES 2.0 §4.1.4](#), [man page](#))

See [Stencil Separate Mask and Reference Value](#) for information on WebGL specific limitations to the allowable argument values.

**void stencilMask(GLuint mask)** ( [OpenGL ES 2.0 §4.2.2](#)

[man page](#))

See [Stencil Separate Mask and Reference Value](#) for information on WebGL specific limitations to the allowable mask values.

**void stencilMaskSeparate(GLenum face, GLuint mask)** ( [OpenGL ES](#)

**void stencilOp(GLenum fail, GLenum zfail, GLenum zpass)** ( [OpenGL](#)[ES 2.0 §4.1.4](#)[man page](#)

)

**void stencilOpSeparate(GLenum face, GLenum fail, GLenum zfail, GLenum zpass)**( [OpenGL ES 2.0 §4.1.4](#)[man page](#)

)

## Viewing and clipping

The viewport specifies the affine transformation of x and y from normalized device coordinates to window coordinates. The size of the drawing buffer is determined by the `HTMLCanvasElement`. The scissor box defines a rectangle which constrains drawing. When the scissor test is enabled only pixels that lie within the scissor box can be modified by drawing commands. When enabled drawing can only occur inside the intersection of the viewport, canvas area and the scissor box. When the scissor test is not enabled drawing can only occur inside the intersection of the viewport and canvas area.

**void scissor(GLint x, GLint y, GLsizei width, GLsizei height)** ( [OpenGL ES](#)[2.0 §4.1.2](#)[man page](#)

)

**void viewport(GLint x, GLint y, GLsizei width, GLsizei height)** ( [OpenGL](#)[ES 2.0 §2.12.1](#)[man page](#)

)

## Buffer objects

Buffer objects (sometimes referred to as VBOs) hold vertex attribute data for the GLSL shaders.

**void bindBuffer(GLenum target, WebGLBuffer buffer)** ( [OpenGL](#)[ES 2.0 §2.9](#)[man page](#)

)

Binds the given `WebGLBuffer` object to the given binding point (`target`), either `ARRAY_BUFFER` or `ELEMENT_ARRAY_BUFFER`. If the buffer is null then any buffer currently bound to this target is unbound. A given

WebGLBuffer object may only be bound to one of the ARRAY\_BUFFER or ELEMENT\_ARRAY\_BUFFER target for its lifetime. An attempt to bind a buffer object to the other target will generate an INVALID\_OPERATION error, and the current binding will remain untouched.



## **void bufferData(GLenum target, GLsizeiptr size, GLenum usage)**

( [OpenGL ES 2.0 §2.9](#) , [man page](#) )

Set the size of the currently bound WebGLBuffer object for the passed target. The buffer is initialized to 0.

## **void bufferData(GLenum target, ArrayBufferView data, GLenum usage)**

## **void bufferData(GLenum target, ArrayBuffer data, GLenum**

**usage)** ( [OpenGL ES 2.0 §2.9](#) , [man page](#) )

Set the size of the currently bound WebGLBuffer object for the passed target to the size of the passed data, then write the contents of data to the buffer object.

If the passed data is null then an INVALID\_VALUE error is generated.

## **void bufferSubData(GLenum target, GLintptr offset, ArrayBufferView data)**

## **void bufferSubData(GLenum target, GLintptr offset, ArrayBuffer**

**data)** ( [OpenGL ES 2.0 §2.9](#) , [man page](#) )

For the WebGLBuffer object bound to the passed target write the passed data starting at the passed offset. If the data would be written past the end of the buffer object an INVALID\_VALUE error is generated.

## **WebGLBuffer createBuffer()** ( [OpenGL ES 2.0 §2.9](#) ,

*similar to* [glGenBuffers](#) )

Create a WebGLBuffer object and initialize it with a buffer object name as if by calling glGenBuffers.

## **void deleteBuffer(WebGLBuffer buffer)** ( [OpenGL ES](#)

[2.0 §2.9](#) , *similar to* [glDeleteBuffers](#) )

Delete the buffer object contained in the passed WebGLBuffer as if by calling glDeleteBuffers. If the buffer has already been deleted the call has no effect. Note that the buffer object will be deleted when the WebGLBuffer object is destroyed. This method merely gives the author greater control over when the buffer object is destroyed.

## **any getBufferParameter(GLenum target, GLenum pname)** ( [OpenGL](#)

[ES 2.0 §6.1.3](#) , *similar to*

[glGetBufferParameteriv](#) )

Return the value for the passed pname. The type returned is the natural type for the requested pname, as given in the following table:

pname	returned type
BUFFER_SIZE	long
BUFFER_USAGE	unsigned long

## GLboolean isBuffer(WebGLBuffer buffer) ( [OpenGL ES 2.0](#) )

[§6.1.6](#), [man page](#))

### Framebuffer objects

Framebuffer objects provide an alternative rendering target to the drawing buffer. They are a collection of color, alpha, depth and stencil buffers and are often used to render an image that will later be used as a texture.

#### void bindFramebuffer(GLenum target, WebGLFramebuffer framebuffer) ( [OpenGL ES 2.0 §4.4.1](#) , [man page](#) )

Bind the given WebGLFramebuffer object to the given binding point (target), which must be FRAMEBUFFER. If framebuffer is null, the default framebuffer provided by the context is bound and attempts to modify or query state on target FRAMEBUFFER will generate an INVALID\_OPERATION error.

#### GLenum checkFramebufferStatus(GLenum target) ( [OpenGL](#) )

[ES 2.0 §4.4.5](#), [man page](#))

#### WebGLFramebuffer createFramebuffer() ( [OpenGL ES 2.0](#) )

[§4.4.1](#), similar to [glGenFramebuffers](#))

Create a WebGLFramebuffer object and initialize it with a framebuffer object name as if by calling glGenFramebuffers.

#### void deleteFramebuffer(WebGLFramebuffer buffer) ( [OpenGL ES](#) )

[2.0 §4.4.1](#), similar

to [glDeleteFramebuffers](#))

Delete the framebuffer object contained in the passed WebGLFramebuffer as if by calling glDeleteFramebuffers. If the framebuffer has already been deleted the call has no effect. Note that the framebuffer object will be deleted when the WebGLFramebuffer object is destroyed. This method merely gives the author greater control over

when the framebuffer object is destroyed.



**void framebufferRenderbuffer(GLenum target, GLenum attachment, GLenum renderbuffertarget, WebGLRenderbuffer renderbuffer)** ( [OpenGL ES 2.0 §4.4.3](#), [man page](#) )

**void framebufferTexture2D(GLenum target, GLenum attachment, GLenum textarget, WebGLTexture texture, GLint level)** ( [OpenGL ES 2.0 §4.4.3](#), [man page](#) )

**any getFramebufferAttachmentParameter(GLenum target, GLenum attachment, GLenum pname)** ( [OpenGL ES 2.0 §6.1.3](#), [similar](#) )

to [glGetFramebufferAttachmentParameteriv](#) )

Return the value for the passed pname given the passed target and attachment. The type returned is the natural type for the requested pname, as given in the following table:

pname	returned type
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	unsigned long
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME	WebGLRenderbuffer or WebGLTexture
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL	long
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE	long

**GLboolean isFramebuffer(WebGLFramebuffer framebuffer)** ( [OpenGL ES 2.0 §6.1.7](#), [man page](#) )

Return true if the passed WebGLFramebuffer is valid and false otherwise.

## Renderbuffer objects

Renderbuffer objects are used to provide storage for the individual buffers used in a framebuffer object.

**void bindRenderbuffer(GLenum target, WebGLRenderbuffer renderbuffer)** ( [OpenGL ES 2.0 §4.4.3](#), [man page](#) )

## WebGLRenderbuffer createRenderbuffer()

( [OpenGL ES 2.0 §4.4.3](#), similar to

[glGenRenderbuffers](#) )

Create a WebGLRenderbuffer object and initialize it with a renderbuffer object name as if by calling glGenRenderbuffers.

## void deleteRenderbuffer(WebGLRenderbuffer renderbuffer)

( [OpenGL ES 2.0 §4.4.3](#)

, similar to

[glDeleteRenderbuffers](#) )

Delete the renderbuffer object contained in the passed WebGLRenderbuffer as if by calling glDeleteRenderbuffers. If the renderbuffer has already been deleted the call has no effect. Note that the renderbuffer object will be deleted when the WebGLRenderbuffer object is destroyed. This method merely gives the author greater control over when the renderbuffer object is destroyed.

## any getRenderbufferParameter(GLenum target, GLenum pname)

( [OpenGL ES 2.0 §6.1.3](#)

, similar

to [glGetRenderbufferParameteriv](#) )

Return the value for the passed pname given the passed target. The type returned is the natural type for the requested pname, as given in the following table:

pname	returned type
RENDERBUFFER_WIDTH	long
RENDERBUFFER_HEIGHT	long
RENDERBUFFER_INTERNAL_FORMAT	unsigned long
RENDERBUFFER_RED_SIZE	long
RENDERBUFFER_GREEN_SIZE	long
RENDERBUFFER_BLUE_SIZE	long
RENDERBUFFER_ALPHA_SIZE	long
RENDERBUFFER_DEPTH_SIZE	long
RENDERBUFFER_STENCIL_SIZE	long

## GLboolean isRenderbuffer(WebGLRenderbuffer renderbuffer)

( [OpenGL ES 2.0 §6.1.7](#)

, [man page](#) )

Return true if the passed WebGLRenderbuffer is valid and false otherwise.

## Texture objects

Texture objects provide storage and state for texturing operations. If no WebGLTexture is bound (e.g., passing null or 0 to bindTexture) then attempts to modify or query the texture object shall generate an INVALID\_OPERATION error. This is indicated in the functions below.

**void bindTexture(GLenum target, WebGLTexture texture)** ( [OpenGL](#)

[ES 2.0 §3.7.13](#)

[man page](#)

**void copyTexImage2D(GLenum target, GLint level, GLenum internalformat,  
GLint x, GLint y, GLsizei width, GLsizei height, GLint border)**

( [OpenGL ES 2.0 §3.7.2](#)

[man page](#)

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

For any pixel lying outside the frame buffer, all channels of the associated texel are initialized to 0; see [Reading Pixels Outside the Framebuffer](#).

**void copyTexSubImage2D(GLenum target, GLint level, GLint xoffset,  
GLint yoffset, GLint x, GLint y, GLsizei width, GLsizei height)**

( [OpenGL ES 2.0 §3.7.2](#)

[man page](#)

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

For any pixel lying outside the frame buffer, all channels of the associated texel are initialized to 0; see [Reading Pixels Outside the Framebuffer](#).

**WebGLTexture createTexture()** ( [OpenGL ES 2.0](#)

[§3.7.13](#)

[man page](#)

Create a WebGLTexture object and initialize it with a texture object name as if by calling glGenTextures.

## **void deleteTexture(WebGLTexture texture)** ( [OpenGL ES](#) )

[2.0 §3.7.13](#)

[man page](#)

Delete the texture object contained in the passed WebGLTexture as if by calling glDeleteTextures. If the texture has already been deleted the call has no effect. Note that the texture object will be deleted when the WebGLTexture object is destroyed. This method merely gives the author greater control over when the texture object is destroyed.

## **void generateMipmap(GLenum target)** ( [OpenGL ES](#) )

[2.0 §3.7.11](#)

[man page](#)

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

## **any getTexParameter(GLenum target, GLenum pname)** ( [OpenGL](#) )

[ES 2.0 §6.1.3](#)

[man page](#)

Return the value for the passed pname given the passed target. The type returned is the natural type for the requested pname, as given in the following table:

pname	returned type
TEXTURE_MAG_FILTER	unsigned long
TEXTURE_MIN_FILTER	unsigned long
TEXTURE_WRAP_S	unsigned long
TEXTURE_WRAP_T	unsigned long

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

## **GLboolean isTexture(WebGLTexture texture)** ( [OpenGL ES 2.0](#) )

[§6.1.4](#)

[man page](#)

Return true if the passed WebGLTexture is valid and false otherwise.

## **void texImage2D(GLenum target, GLint level, GLenum internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, ArrayBufferView pixels)** ( [OpenGL ES 2.0](#) )

[§3.7.1](#)

[man page](#)

If pixels is null, a buffer of sufficient size initialized to 0 is passed.

If pixels is non-null, the type of pixels must match the type of the data to be read. If it is UNSIGNED\_BYTE, a Uint8Array must be supplied; if it is UNSIGNED\_SHORT\_5\_6\_5, UNSIGNED\_SHORT\_4\_4\_4\_4, or

UNSIGNED\_SHORT\_5\_5\_5\_1, a Uint16Array must be supplied. If the types do not match, an error is generated.



If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

See [Pixel Storage Parameters](#) for WebGL-specific pixel storage parameters that affect the behavior of this function.

**void texImage2D(GLenum target, GLint level, GLenum internalformat, GLenum format, GLenum type, ImageData pixels)**

**void texImage2D(GLenum target, GLint level, GLenum internalformat, GLenum format, GLenum type, HTMLImageElement image)**

**raises (DOMException)**

**void texImage2D(GLenum target, GLint level, GLenum internalformat, GLenum format, GLenum type, HTMLCanvasElement canvas)**

**raises (DOMException)**

**void texImage2D(GLenum target, GLint level, GLenum internalformat, GLenum format, GLenum type, HTMLVideoElement video) raises (DOMException)**

( [OpenGL ES 2.0 §3.7.1](#),  
[man page](#) )

Uploads the given element or image data to the currently bound WebGLTexture.

The source image data is conceptually first converted to the data type and format specified by the *format* and *type* arguments, and then transferred to the OpenGL implementation. If a packed pixel format is specified which would imply loss of bits of precision from the image data, this loss of precision must occur.

If the source image is an RGB or RGBA lossless image with 8 bits per channel, the browser guarantees that the full precision of all channels is preserved.

If the original image contains an alpha channel and the UNPACK\_PREMULTIPLY\_ALPHA\_WEBGL pixel storage parameter is false, then the RGB values are guaranteed to never have been premultiplied by the alpha channel, whether those values are derived directly from the original file format or converted from some other color format.

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

If this function is called with an HTMLImageElement or HTMLVideoElement whose origin differs from the origin of the containing Document, or with an HTMLCanvasElement whose *origin*-

`clean`

flag is set to false, a SECURITY\_ERR exception must be raised. See [OpenGL ES 2.0 §3.7.4](#)

See [Pixel Storage Parameters](#) for WebGL-specific pixel storage parameters that affect the behavior of this function.



## **void texParameterf(GLenum target, GLenum pname, GLfloat param)**

( [OpenGL ES 2.0 §3.7.4](#), [man page](#) )

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

## **void texParameteri(GLenum target, GLenum pname, GLint param)**

( [OpenGL ES 2.0 §3.7.4](#), [man page](#) )

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

## **void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLsizei width, GLsizei height, GLenum format, GLenum type, ArrayBufferView pixels)**

( [OpenGL ES 2.0 §3.7.2](#), [man page](#) )

See [texImage2D](#) for restrictions on the *format* and *pixels* arguments.

If an attempt is made to call this function with no WebGLTexture bound (see above), an INVALID\_OPERATION error is generated.

If *type* does not match the type originally used to define the texture, an INVALID\_OPERATION error is generated.

See [Pixel Storage Parameters](#) for WebGL-specific pixel storage parameters that affect the behavior of this function.

## **void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLenum format, GLenum type, ImageData pixels)**

## **void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLenum format, GLenum type, HTMLImageElement image)**

**raises (DOMException)**

## **void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLenum format, GLenum type, HTMLCanvasElement canvas)**

**raises (DOMException)**

## **void texSubImage2D(GLenum target, GLint level, GLint xoffset, GLint**

**yoffset, GLenum format, GLenum type, HTMLVideoElement video)**

**raises (DOMException)** ( [OpenGL ES 2.0 §3.7.2](#)

[man page](#) )



Updates a sub-rectangle of the currently bound WebGLTexture with the contents of the given element or image data.

See [texImage2D](#) for the interpretation of the *format* and *type* arguments.

If an attempt is made to call this function with no WebGLTexture bound (see above), an **INVALID\_OPERATION** error is generated.

If *type* does not match the type originally used to define the texture, an **INVALID\_OPERATION** error is generated.

If this function is called with an **HTMLImageElement** or **HTMLVideoElement** whose origin differs from the origin of the containing Document, or with an **HTMLCanvasElement** whose *origin-clean* flag is set to false, a **SECURITY\_ERR** exception must be raised. See [Origin Restrictions](#).

See [Pixel Storage Parameters](#) for WebGL-specific pixel storage parameters that affect the behavior of this function.

## Programs and Shaders

Rendering with OpenGL ES 2.0 requires the use of *shaders*, written in OpenGL ES's shading language, GLSL ES. Shaders must be loaded with a source string (*shaderSource*), compiled (*compileShader*) and attached to a *program* (*attachShader*) which must be linked (*linkProgram*) and then used (*useProgram*).

**void attachShader(WebGLProgram program, WebGLShader shader)**

( [OpenGL ES 2.0 §2.10.3](#)

, [man page](#) )

**void bindAttribLocation(WebGLProgram program, GLuint index, DOMString name)** ( [OpenGL ES 2.0 §2.10.4](#)

, [man page](#) )

See [Characters Outside the GLSL Source Character Set](#) for additional validation performed by WebGL implementations.

## **void compileShader(WebGLShader shader)** ( [OpenGL ES 2.0](#)

[§2.10.1](#)

[man page](#)



## **WebGLProgram createProgram()** ( [OpenGL ES 2.0](#)

[§2.10.3](#)

[man page](#)

Create a WebGLProgram object and initialize it with a program object name as if by calling glCreateProgram.

## **WebGLShader createShader(type)** ( [OpenGL ES](#)

[2.0 §2.10.1](#)

[man page](#)

Create a WebGLShader object and initialize it with a shader object name as if by calling glCreateShader.

## **void deleteProgram(WebGLProgram program)** ( [OpenGL ES](#)

[2.0 §2.10.3](#)

[man page](#)

Delete the program object contained in the passed WebGLProgram as if by calling glDeleteProgram. If the program has already been deleted the call has no effect. Note that the program object will be deleted when the WebGLProgram object is destroyed. This method merely gives the author greater control over when the program object is destroyed.

## **void deleteShader(WebGLShader shader)** ( [OpenGL ES](#)

[2.0 §2.10.1](#)

[man page](#)

Delete the shader object contained in the passed WebGLShader as if by calling glDeleteShader. If the shader has already been deleted the call has no effect. Note that the shader object will be deleted when the WebGLShader object is destroyed. This method merely gives the author greater control over when the shader object is destroyed.

## **void detachShader(WebGLProgram program, WebGLShader shader)** ( [OpenGL ES 2.0 §2.10.3](#)

[man page](#)

)

## **WebGLShader[ ] getAttachedShaders(WebGLProgram program)**

( [OpenGL ES 2.0 §6.1.8](#)

[man page](#)

)

Return the list of shaders attached to the passed program.

## **any getProgramParameter(WebGLProgram program, GLenum pname)** ( [OpenGL ES 2.0 §6.1.8](#)

, similar

to

[man page](#)

)

Return the value for the passed pname given the passed program. The type returned is the natural type for the requested pname, as given in the following table:

pname	returned type
-------	---------------

DELETE_STATUS	boolean
LINK_STATUS	boolean
VALIDATE_STATUS	boolean
ATTACHED_SHADERS	long
ACTIVE_ATTRIBUTES	long
ACTIVE_UNIFORMS	long

## DOMString getProgramInfoLog(WebGLProgram program) ( [OpenGL](#) )

[ES 2.0 §6.1.8](#)

[man page](#)

)

## any getShaderParameter(WebGLShader shader, GLenum pname) ( [OpenGL](#) )

[ES 2.0 §6.1.8](#)

, similar to

[man page](#)

)

Return the value for the passed pname given the passed shader. The type returned is the natural type for the requested pname, as given in the following table:

pname	returned type
SHADER_TYPE	unsigned long
DELETE_STATUS	boolean
COMPILE_STATUS	boolean

## DOMString getShaderInfoLog(WebGLShader shader) ( [OpenGL](#) )

[ES 2.0 §6.1.8](#)

[man page](#)

)

## DOMString getShaderSource(WebGLShader shader) ( [OpenGL ES](#) )

[2.0 §6.1.8](#)

[man page](#)

)

## GLboolean isProgram(WebGLProgram program) ( [OpenGL ES](#) )

[2.0 §6.1.8](#)

[man page](#)

)

Return true if the passed WebGLProgram is valid and false otherwise.

## GLboolean isShader(WebGLShader shader) ( [OpenGL ES 2.0](#) )

[§6.1.8](#)

[man page](#)

)

Return true if the passed WebGLShader is valid and false otherwise.

## void linkProgram(WebGLProgram program) ( [OpenGL ES](#) )

**void shaderSource(WebGLShader shader, DOMString source)** ( [OpenGL ES](#) )[ES 2.0 §2.10.1](#)[man page](#)

See [Supported GLSL Constructs](#) and [Characters Outside the GLSL Source Character Set](#) for additional constructs supported by and validation performed by WebGL implementations.

**void useProgram(WebGLProgram program)** ( [OpenGL ES](#) )[2.0 §2.10.3](#)[man page](#)**void validateProgram(WebGLProgram program)** ( [OpenGL ES](#) )[2.0 §2.10.5](#)[man page](#)**Uniforms and attributes**

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

**void disableVertexAttribArray(GLuint index)** ( [OpenGL ES](#) )[2.0 §2.8](#)[man page](#)**void enableVertexAttribArray(GLuint index)** ( [OpenGL ES](#) )[2.0 §2.8](#)[man page](#)

Enable the vertex attribute at `index` as an array. WebGL imposes additional rules beyond OpenGL ES 2.0 regarding enabled vertex attributes; see [Enabled Vertex Attributes and Range Checking](#).

**WebGLActiveInfo getActiveAttrib(WebGLProgram program, GLuint index)** ( [OpenGL ES 2.0 §2.10.4](#) , [man page](#) )

Returns information about the size, type and name of the vertex attribute at the passed index of the passed program object.

**WebGLActiveInfo getActiveUniform(WebGLProgram program, GLuint index)** ( [OpenGL ES 2.0 §2.10.4](#) , [man page](#) )

Returns information about the size, type and name of the uniform at the passed index of the passed program object.

## **GLint getAttribLocation(WebGLProgram program, DOMString name)** ( [OpenGL ES 2.0 §2.10.4](#) ) , [man page](#)

See [Characters Outside the GLSL Source Character Set](#) for additional validation performed by WebGL implementations.

## **any getUniform(WebGLProgram program, WebGLUniformLocation location)** ( [OpenGL ES 2.0 §6.1.8](#) ) , [man page](#) )

Return the uniform value at the passed location in the passed program. The type returned is dependent on the uniform type, as shown in the following table:

uniform type	returned type
boolean	boolean
int	long
float	float
vec2	Float32Array (with 2 elements)
ivec2	Int32Array (with 2 elements)
bvec2	boolean[] (with 2 elements)
vec3	Float32Array (with 3 elements)
ivec3	Int32Array (with 3 elements)
bvec3	boolean[] (with 3 elements)
vec4	Float32Array (with 4 elements)
ivec4	Int32Array (with 4 elements)
bvec4	boolean[] (with 4 elements)
mat2	Float32Array (with 4 elements)
mat3	Float32Array (with 9 elements)
mat4	Float32Array (with 16 elements)

## **WebGLUniformLocation getUniformLocation(WebGLProgram program, DOMString name)** ( [OpenGL ES 2.0 §2.10.4](#) ) , [man page](#) )

Return a WebGLUniformLocation object that represents the location of a specific uniform variable within a program object. The return value is null if name does not correspond to an active uniform variable in the passed program.

See [Characters Outside the GLSL Source Character Set](#) for additional validation performed by WebGL

**any getVertexAttrib(GLuint index, GLenum pname)** ( [OpenGL ES](#) )[2.0 §6.1.8](#)[man page](#)

Return the information requested in `pname` about the vertex attribute at the passed index. The type returned is dependent on the information requested, as shown in the following table:

<code>pname</code>	<code>returned type</code>
<code>VERTEX_ATTRIB_ARRAY_BUFFER_BINDING</code>	<code>WebGLBuffer</code>
<code>VERTEX_ATTRIB_ARRAY_ENABLED</code>	<code>boolean</code>
<code>VERTEX_ATTRIB_ARRAY_SIZE</code>	<code>long</code>
<code>VERTEX_ATTRIB_ARRAY_STRIDE</code>	<code>long</code>
<code>VERTEX_ATTRIB_ARRAY_TYPE</code>	<code>unsigned long</code>
<code>VERTEX_ATTRIB_ARRAY_NORMALIZED</code>	<code>boolean</code>
<code>CURRENT_VERTEX_ATTRIB</code>	<code>Float32Array (with 4 elements)</code>

**GLsizeiptr getVertexAttribOffset(GLuint index, GLenum pname)** ( [OpenGL](#) )[ES 2.0 §6.1.8](#)*, similar to*[man page](#)

```
void uniform[1234][fi](WebGLUniformLocation location, ...)
void uniform[1234][fi]v(WebGLUniformLocation location, ...)
void uniformMatrix[234]fv(WebGLUniformLocation location,
GLboolean transpose, ...) ( OpenGL ES 2.0
```

[§2.10.4](#)[man page](#)

Each of the uniform\* functions above sets the specified uniform or uniforms to the values provided. If the passed location is not null and was not obtained from the currently used program via an earlier call to `getUniformLocation`, an `INVALID_OPERATION` error will be generated. If the passed location is null, the data passed in will be silently ignored and no uniform variables will be changed.

If the array passed to any of the vector forms (those ending in V) has an invalid length, an `INVALID_VALUE` error will be generated. The length is invalid if it is too short for or is not an integer multiple of the assigned type.

**void vertexAttrib[1234]f(GLuint idx, ...)****void vertexAttrib[1234]fv(GLuint idx, ...)** ( [OpenGL ES](#) )[2.0 §2.7](#)[man page](#)

Sets the vertex attribute at the passed index to the given constant value. Values set via the `vertexAttrib` are

guaranteed to be returned from the `getVertexAttrib` function with the `CURRENT_VERTEX` even if there have been intervening calls to `drawArrays` or `drawElements`.



## **void vertexAttribPointer(GLuint indx, GLint size, GLenum type, GLboolean normalized, GLsizei stride, GLintptr offset)** ( [OpenGL ES](#)

[2.0 §2.8](#)

, [man page](#)

Assign the WebGLBuffer object currently bound to the `ARRAY_BUFFER` target to the vertex attribute at the passed index. Size is number of components per attribute. Stride and offset are in units of bytes. Passed stride and offset must be appropriate for the passed type and size or an `INVALID_OPERATION` error will be generated; see [Buffer Offset and Stride Requirements](#). If no WebGLBuffer is bound to the `ARRAY_BUFFER` target, an `INVALID_OPERATION` error will be generated. In WebGL, the maximum supported stride is 255; see [Vertex Attribute Data Stride](#).

### Writing to the drawing buffer

OpenGL ES 2.0 has 3 calls which can render to the drawing buffer: `clear`, `drawArrays` and `drawElements`. Furthermore rendering can be directed to the drawing buffer or to a Framebuffer object. When rendering is directed to the drawing buffer, making any of the 3 rendering calls shall cause the drawing buffer to be presented to the HTML page compositor at the start of the next compositing operation.

#### **void clear(GLbitfield mask)** ( [OpenGL ES 2.0 §4.2.3](#)

[man page](#)

#### **void drawArrays(GLenum mode, GLint first, GLsizei count)** ( [OpenGL](#)

[ES 2.0 §2.8](#)

, [man page](#)

If `first` is negative, an `INVALID_VALUE` error will be generated.

#### **void drawElements(GLenum mode, GLsizei count, GLenum type, GLintptr offset)** ( [OpenGL ES 2.0 §2.8](#)

, [man page](#)

Draw using the currently bound element array buffer. The given offset is in bytes, and must be a valid multiple of the size of the given type or an `INVALID_OPERATION` error will be generated; see [Buffer Offset and Stride Requirements](#). If `count` is greater than zero, then a non-null WebGLBuffer must be bound to the `ELEMENT_ARRAY_BUFFER` binding point or an `INVALID_OPERATION` error will be generated.

**void finish()** ( [OpenGL ES 2.0 §5.1](#), [man page](#) )

**void flush()** ( [OpenGL ES 2.0 §5.1](#), [man page](#) )

## Reading back pixels

Pixels in the current framebuffer can be read back into an ArrayBufferView object.

**void readPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, ArrayBufferView pixels)** ( [OpenGL ES 2.0 §4.3.1](#), [man page](#) )

Fills `pixels` with the pixel data in the specified rectangle of the frame buffer. The data returned from `readPixels` must be up-to-date as of the most recently sent drawing command.

The type of `pixels` must match the type of the data to be read. For example, if it is `UNSIGNED_BYTE`, a `Uint8Array` must be supplied; if it is `UNSIGNED_SHORT_5_6_5`, `UNSIGNED_SHORT_4_4_4_4`, or `UNSIGNED_SHORT_5_5_5_1`, a `Uint16Array` must be supplied. If the types do not match, an `INVALID_OPERATION` error is generated.

The following are the allowed format and type combinations:

format	type
RGBA	UNSIGNED_BYTE

If `pixels` is null, an `INVALID_VALUE` error is generated. If `pixels` is non-null, but is not large enough to retrieve all of the pixels in the specified rectangle taking into account pixel store modes, an `INVALID_OPERATION` value is generated.

For any pixel lying outside the frame buffer, the value read contains 0 in all channels; see [Reading Pixels Outside the Framebuffer](#).

Occurrences such as power events on mobile devices may cause the WebGL rendering context to be lost at any time and require the application to rebuild it; see [WebGLContextEvent](#) for more details. The following method assists in detecting context lost events.

### **boolean isContextLost()**

Returns true if the context is in the lost state.

## Detecting and enabling extensions

An implementation of WebGL must not support any additional parameters, constants or functions without first enabling that functionality through the extension mechanism. The `getSupportedExtensions` function returns an array of the extension strings supported by this implementation. Extension strings are case-insensitive. An extension is enabled by passing one of those strings to the `getExtension` function. This call returns an object which contains any constants or functions defined by that extension. The definition of that object is specific to the extension and must be defined by the extension specification.

Once an extension is enabled, no mechanism is provided to disable it. Multiple calls to `getExtension` with the same extension string shall return the same object. An attempt to use any features of an extension without first calling `getExtension` to enable it must generate an appropriate GL error and must not make use of the feature.

This specification does not define any extensions. A separate [WebGL extension registry](#) defines extensions that may be supported by a particular WebGL implementation.

### **DOMString[ ] getSupportedExtensions()**

Returns an array of all the supported extension strings. Any string in this list, when passed to `getExtension` must return a valid object. Any other string passed to `getExtension` must return null.

### **object getExtension(DOMString name)**

Returns an object if the passed extension is supported, or null if not. The object returned from `getExtension` contains any constants or functions used by the extension, if any. A returned object may have no constants or functions if the extension does not define any, but a unique object must still be returned. That object is used to indicate that the extension has been enabled.

## WebGLContextEvent

WebGL generates a `WebGLContextEvent` event in response to a status change to the WebGL rendering context associated with the `HTMLCanvasElement` which has a listener for this event. Events are sent using the [DOM Event System](#). Event types can include the loss or restoration of state, or the inability to create a context.

```
interface WebGLContextEvent : Event {  
    readonly attribute DOMString statusMessage;  
  
    void initWebGLContextEvent(DOMString typeArg,  
                               boolean canBubbleArg,  
                               boolean cancelableArg,  
                               DOMString statusMessageArg);  
};
```

### Attributes

The following attributes are available:

#### **statusMessage of type DOMString**

A string containing additional information, or the empty string if no additional information is available.

### Methods

The following methods are available:

```
void initWebGLContextEvent(DOMString typeArg, boolean  
                           canBubbleArg, boolean cancelableArg, DOMString statusMessageArg)
```

Initialize the event created through the [Event](#) interface. This method may only be called before the event is dispatched via the `dispatchEvent` method, though it may be called multiple times during the dispatching process if necessary. If called multiple times, the final invocation takes precedence.



## Parameters

### **typeArg** of type **DOMString**

Specifies the event type (see below).

### **canBubbleArg** of type **boolean**

Specifies whether or not the event can bubble.

### **cancelableArg** of type **boolean**

Specifies whether or not the event can be canceled.

### **statusMessageArg** of type **DOMString**

A string containing additional information, or the empty string if no additional information is available.

**No Return Value**

**No Exceptions**

## Event Types

### **webglcontextlost**

This event occurs when some system activity external to WebGL causes the rendering context associated with the `HTMLCanvasElement` receiving the event to lose all of its state. When this event type is delivered, the associated context is in a "lost" state. While the context is in the lost state:

- The context's `isContextLost` method returns `true`.
- All methods returning `void` return immediately.
- All methods returning nullable values or `any` return `null`.
- `checkFramebufferStatus` returns `FRAMEBUFFER_UNSUPPORTED`.
- `getAttribLocation` returns `-1`.
- `getError` returns `CONTEXT_LOST_WEBGL` the first time it is called while the context is lost. Afterward it will return `NO_ERROR` until the context has been restored.
- `getVertexAttribOffset` returns `0`.
- All `is` queries return `false`.

The default behavior of the `webglcontextlost` event is to prevent the context from being re-created.

The `statusMessage` attribute is always the empty string for this event.

The following code prevents the default behavior of the `webglcontextlost` event and enables the `webglcontextrestored` event to be delivered:

```
canvas.addEventListener("webglcontextlost", function(e) { e.preventDefault(); }, false);
```

## **webglcontextrestored**

This event occurs when the WebGL implementation determines that a previously lost context can be restored. The system only restores the context if the default behavior for the `webglcontextlost` event is prevented on the `HTMLCanvasElement` associated with the context.

Once the context is restored, WebGL resources such as textures and buffers that were created before the context was lost are no longer valid. The application must reinitialize the context's state and recreate all such resources.

The `statusMessage` attribute is always the empty string for this event.

## **webglcontextcreationerror**

This event occurs in response to a call to `getContext()` on the `HTMLCanvasElement` receiving the event, when some error occurs during that call. When such an error occurs the `getContext()` returns with a `null` value. Later, at the normal event delivery time, this event is delivered with the details of the failure. If the `HTMLCanvasElement` is listening for this event, it will be delivered whenever `getContext()` returns `null`.

The `statusMessage` attribute may contain a platform dependent string about the nature of the failure.

The following ECMAScript example shows how to register event listeners on a Canvas called `canvas1` which will receive context lost and restored events and restart the application. For completeness, it illustrates how an application performing asynchronous image loading would handle a context lost event at an arbitrary point in time.

```
window.onload = init;

var g_gl;
var g_canvas;
var g_intervalId;
var g_images = [];
var g_imgURLs = [
    "someimage.jpg",
    "someotherimage.jpg",
    "yetanotherimage.png"
];

function init() {
    g_canvas = document.getElementById("canvas1");
    g_canvas.addEventListener("webglcontextlost", contextLostHandler, false);
    g_canvas.addEventListener("webglcontextrestored", contextRestoredHandler, false);
    g_gl = canvas.getContext("webgl");

    for (var ii = 0; ii < g_imgURLs.length; ++ii) {
        // Create an image tag.
        var image = document.createElement('img');

        // Create a texture for this image.
        image.texture = g_gl.createTexture();

        // Mark the image as not loaded.
        image.loaded = false;

        // Setup a load callback.
        image.onload = (function(image) {
            return function() {
                imageLoadedHandler(image);
            };
        })(image));
    }
}
```

```
// Start the image loading.  
image.src = g_imgURLs[ii];  
  
        // Remember the image.  
        g_images.push(image);  
    }  
  
    g_intervalId = window.setInterval(renderHandler, 1000/60);  
}  
  
function renderHandler() {  
    // draw with textures.  
    // ...  
}  
  
function imageLoadedHandler(image) {  
    // Mark the image as loaded.  
    image.loaded = true;  
  
    // Copy the image to the texture.  
    updateTexture(image);  
}  
  
function updateTexture(image) {  
    if (!g_gl.isContextLost() && image.loaded) {  
        g_gl.bindTexture(g_gl.TEXTURE_2D, image.texture);  
        g_gl.texImage2D(g_gl.TEXTURE_2D, 0, image);  
    }  
}  
  
function contextLostHandler(event) {  
    // allow the context to be restored.  
    event.preventDefault();  
    // stop rendering.  
    window.clearInterval(g_intervalId);  
}
```

```
}
```

```
function contextRestoredHandler() {
    // create new textures for all images and restore their contents.
    for (var ii = 0; ii < g_images.length; ++ii) {
        g_images[ii].texture = g_gl.createTexture();
        updateTexture(g_images[ii]);
    }

    // Start rendering again.
    g_intervalId = window.setInterval(renderHandler, 1000/60);
}
```

## Differences Between WebGL and OpenGL ES 2.0

This section describes changes made to the WebGL API relative to the OpenGL ES 2.0 API to improve portability across various operating systems and devices.

### Buffer Object Binding

In the WebGL API, a given buffer object may only be bound to one of the `ARRAY_BUFFER` or `ELEMENT_ARRAY_BUFFER` binding points in its lifetime. This restriction implies that a given buffer object may contain either vertices or indices, but not both.

The type of a `WebGLBuffer` is initialized the first time it is passed as an argument to `bindBuffer`. A subsequent call to `bindBuffer` which attempts to bind the same `WebGLBuffer` to the other binding point will generate an `INVALID_OPERATION` error, and the state of the binding point will remain untouched.

### No Client Side Arrays

The WebGL API does not support client-side arrays. If `vertexAttribPointer` is called without a `WebGLBuffer` bound to the `ARRAY_BUFFER` binding point, an `INVALID_OPERATION` error is generated. If `drawElements` is called with a `count` greater than zero, and no `WebGLBuffer` is bound to the `ELEMENT_ARRAY_BUFFER` binding point, an `INVALID_OPERATION` error is generated.

## Buffer Offset and Stride Requirements



The offset arguments to `drawElements` and `vertexAttribPointer`, and the stride argument to `vertexAttribPointer`, must be a multiple of the size of the data type passed to the call, or an `INVALID_OPERATION` error is generated.

## Enabled Vertex Attributes and Range Checking

If a vertex attribute is enabled as an array via `enableVertexAttribArray` but no buffer is bound to that attribute via `bindBuffer` and `vertexAttribPointer`, then calls to `drawArrays` or `drawElements` will generate an `INVALID_OPERATION` error.

If a vertex attribute is enabled as an array, a buffer is bound to that attribute, and the attribute is consumed by the current program, then calls to `drawArrays` and `drawElements` will verify that each referenced vertex lies within the storage of the bound buffer. If the range specified in `drawArrays` or any referenced index in `drawElements` lies outside the storage of the bound buffer, an `INVALID_OPERATION` error is generated and no geometry is drawn.

If a vertex attribute is enabled as an array, a buffer is bound to that attribute, but the attribute is not consumed by the current program, then regardless of the size of the bound buffer, it will not cause any error to be generated during a call to `drawArrays` or `drawElements`.

## Framebuffer Object Attachments

WebGL adds the `DEPTH_STENCIL_ATTACHMENT` framebuffer object attachment point and the `DEPTH_STENCIL` renderbuffer internal format. To attach both depth and stencil buffers to a framebuffer object, call `renderbufferStorage` with the `DEPTH_STENCIL` internal format, and then call `framebufferRenderbuffer` with the `DEPTH_STENCIL_ATTACHMENT` attachment point.

A renderbuffer attached to the `DEPTH_ATTACHMENT` attachment point must be allocated with the `DEPTH_COMPONENT16` internal format. A renderbuffer attached to the `STENCIL_ATTACHMENT` attachment point must be allocated with the `STENCIL_INDEX8` internal format. A renderbuffer attached to the `DEPTH_STENCIL_ATTACHMENT` attachment point must be allocated with the `DEPTH_STENCIL` internal format.

In the WebGL API, it is an error to concurrently attach renderbuffers to the following combinations of attachment points:

- DEPTH\_ATTACHMENT + DEPTH\_STENCIL\_ATTACHMENT
- STENCIL\_ATTACHMENT + DEPTH\_STENCIL\_ATTACHMENT
- DEPTH\_ATTACHMENT + STENCIL\_ATTACHMENT



If any of the constraints above are violated, then:

- checkFramebufferStatus must return FRAMEBUFFER\_UNSUPPORTED.
- The following calls, which either modify or read the framebuffer, must generate an INVALID\_FRAMEBUFFER\_OPERATION error and return early, leaving the contents of the framebuffer, destination texture or destination memory untouched.
  - clear
  - copyTexImage2D
  - copyTexSubImage2D
  - drawArrays
  - drawElements
  - readPixels

## Pixel Storage Parameters

The WebGL API supports the following additional parameters to pixelStorei.

### **UNPACK\_FLIP\_Y\_WEBGL** of type boolean

If set, then during any subsequent calls to texImage2D or texSubImage2D, the source data is flipped along the vertical axis, so that conceptually the last row is the first one transferred. The default value is false.  
Any non-zero value is interpreted as true.

### **UNPACK\_PREMULTIPLY\_ALPHA\_WEBGL** of type boolean

If set, then during any subsequent calls to texImage2D or texSubImage2D, the alpha channel of the source data, if present, is multiplied into the color channels during the data transfer. The default value is false. Any non-zero value is interpreted as true.

### **UNPACK\_COLORSPACE\_CONVERSION\_WEBGL** of type unsigned long

If set to BROWSER\_DEFAULT\_WEBGL, then the browser's default colorspace conversion is applied during subsequent texImage2D and texSubImage2D calls taking HTMLImageElement. The precise conversions may be specific to both the browser and file type. If set to NONE, no colorspace conversion is applied. The default value is BROWSER\_DEFAULT\_WEBGL.

## Reading Pixels Outside the Framebuffer

In the WebGL API, functions which read the framebuffer (copyTexImage2D, copyTexSubImage2D,

## Stencil Separate Mask and Reference Value

In the WebGL API it is illegal to specify a different mask or reference value for front facing and back facing triangles in stencil operations. A call to `drawArrays` or `drawElements` will generate an `INVALID_OPERATION` error if:

- `STENCIL_WRITEMASK != STENCIL_BACK_WRITEMASK` (as specified by `stencilMaskSeparate` for the `mask` parameter associated with the FRONT and BACK values of `face`, respectively)
- `STENCIL_VALUE_MASK != STENCIL_BACK_VALUE_MASK` (as specified by `stencilFuncSeparate` for the `mask` parameter associated with the FRONT and BACK values of `face`, respectively)
- `STENCIL_REF != STENCIL_BACK_REF` (as specified by `stencilFuncSeparate` for the `ref` parameter associated with the FRONT and BACK values of `face`, respectively)

## Vertex Attribute Data Stride

The WebGL API supports vertex attribute data strides up to 255 bytes. A call to `vertexAttribPointer` will generate an `INVALID_VALUE` error if the value for the `stride` parameter exceeds 255.

## Viewport Depth Range

The WebGL API does not support depth ranges with where the near plane is mapped to a value greater than that of the far plane. A call to `depthRange` will generate an `INVALID_OPERATION` error if `zNear` is greater than `zFar`.

## Blending With Constant Color

In the WebGL API, constant color and constant alpha cannot be used together as source and destination factors in the blend function. A call to `blendFunc` will generate an `INVALID_OPERATION` error if one of the two factors is set to `CONSTANT_COLOR` or `ONE_MINUS_CONSTANT_COLOR` and the other to `CONSTANT_ALPHA` or `ONE_MINUS_CONSTANT_ALPHA`. A call to `blendFuncSeparate` will generate an `INVALID_OPERATION` error if `srcRGB` is set to `CONSTANT_COLOR` or `ONE_MINUS_CONSTANT_COLOR` and `dstRGB` is set to `CONSTANT_ALPHA` or `ONE_MINUS_CONSTANT_ALPHA` or vice versa.

The WebGL API does not support the GL\_FIXED data type.

## GLSL Constructs

Per [Supported GLSL Constructs](#), identifiers starting with "webgl\_" and "\_webgl\_" are reserved for use by WebGL.

## Extension Queries

In the OpenGL ES 2.0 API, the available extensions are determined by calling `glGetString(GL_EXTENSIONS)`, which returns a space-separated list of extension strings. In the WebGL API, the EXTENSIONS enumerant has been removed. Instead, `getSupportedExtensions` must be called to determine the set of available extensions.

## Implementation Color Read Format and Type

In the OpenGL ES 2.0 API, the `IMPLEMENTATION_COLOR_READ_FORMAT` and `IMPLEMENTATION_COLOR_READ_TYPE` parameters are used to inform applications of an additional format and type combination that may be passed to `ReadPixels`, in addition to the required `RGBA/UNSIGNED_BYTE` pair. In WebGL 1.0, the supported format and type combinations to `ReadPixels` are documented in the [Reading back pixels](#) section. The `IMPLEMENTATION_COLOR_READ_FORMAT` and `IMPLEMENTATION_COLOR_READ_TYPE` enumerants have been removed.

## Compressed Texture Support

WebGL does not support any compressed texture formats. Therefore the `CompressedTexImage2D` and `CompressedTexSubImage2D` functions are not included in this specification. Likewise the `COMPRESSED_TEXTURE_FORMATS` and `NUM_COMPRESSED_TEXTURE_FORMATS` parameters to `getParameter` will return the value null or zero.

## Maximum GLSL Token Size

The GLSL ES spec [\[GLES20GLSL\]](#) does not define a limit to the length of tokens. WebGL requires support of tokens up to 256 characters in length. Shaders containing tokens longer than 256 characters must fail to compile.

## Characters Outside the GLSL Source Character Set



The GLSL ES spec [\[GLES20GLSL\]](#) defines the source character set for the OpenGL ES shading language to ISO/IEC 646:1991, commonly called ASCII [\[ASCII\]](#). If a string containing a character not in this set is passed to any of the shader-related entry points `bindAttribLocation`, `getAttribLocation`, `getUniformLocation`, or `ShaderSource`, an `INVALID_VALUE` error will be generated. The exception is that any character allowed in an HTML DOMString [\[DOMSTRING\]](#) may be used in GLSL comments. Such use must not generate an error.

Some GLSL implementations disallow characters outside the ASCII range, even in comments. The WebGL implementation needs to prevent errors in such cases. The recommended technique is to preprocess the GLSL string, removing all comments, but maintaining the line numbering for debugging purposes by inserting newline characters as needed.

## String Length Queries

In the WebGL API, the enumerants `INFO_LOG_LENGTH`, `SHADER_SOURCE_LENGTH`, `ACTIVE_UNIFORM_MAX_LENGTH`, and `ACTIVE_ATTRIB_MAX_LENGTH` have been removed. In the OpenGL ES 2.0 API, these enumerants are needed to determine the size of buffers passed to calls like `glGetActiveAttrib`. In the WebGL API, the analogous calls (`getActiveAttrib`, `getActiveUniform`, `getProgramInfoLog`, `getShaderInfoLog`, and `getShaderSource`) all return `DOMString`.

## Texture Type in `TexSubImage2D` Calls

In the WebGL API, the `type` argument passed to `texSubImage2D` must match the type used to originally define the texture object (i.e., using `texImage2D`).

## References

### Normative references

#### [CANVAS]

[HTML5: The Canvas Element](#), World Wide Web Consortium (W3C).

#### [TYPEDARRAYS]

[Typed Array](#)

**[GLES20]**

[\*OpenGL® ES Common\*](#)

[\*Profile Specification\*](#)

[\*Version 2.0.25\*](#)

, A. Munshi, J. Leech, November 2010.

**[GLES20GLSL]**

[\*The OpenGL® ES\*](#)

[\*Shading Language Version 1.00\*](#)

R. Simpson, May 2009.

**[REGISTRY]**

[\*WebGL Extension Registry\*](#)

**[RFC2119]**

[\*Key words for use in RFCs\*](#)

[\*to Indicate Requirement\*](#)

[\*Levels\*](#), S. Bradner. IETF, March 1997.

**[CORS]**

[\*Cross-Origin Resource Sharing\*](#)

van Kesteren, July 2010.

, A.

**[HTML]**

[\*HTML\*](#), I. Hickson, June 2011.

**[WEBIDL]**

[\*Web IDL: W3C Editor\*](#)

*'s Draft*

C. McCormack, September 2009.

,

**[ASCII]**

*International Standard*

*ISO/IEC*

*646:1991.*

*Information technology -*

*ISO 7-bit coded*

*character set for*

*information interchange*

**[DOMSTRING]**

[\*Document Object Model\*](#)

[\*Core: The DOMString type\*](#)

, World Wide

Web Consortium (W3C).

## Other references

## Acknowledgments

This specification is produced by the Khronos WebGL Working Group.



Special thanks to: Arun Ranganathan (Mozilla), Jon Leech, Kenneth Russell (Google), Kenneth Waters (Google), Mark Callow (HI), Mark Steele (Mozilla), Oliver Hunt (Apple), Tim Johansson (Opera), Vangelis Kokkevis (Google), Vladimir Vukicevic (Mozilla), Gregg Tavares (Google)

Additional thanks to: Alan Hudson (Yumetech), Bill Licea Kane (AMD), Cedric Vivier (Zegami), Dan Gessel (Apple), David Ligon (Qualcomm), Glenn Maynard, Greg Roth (Nvidia), Jacob Strom (Ericsson), Kari Pulli (Nokia), Leddie Stenvie (ST-Ericsson), Neil Trevett (Nvidia), Per Wennersten (Ericsson), Per-Erik Brodin (Ericsson), Shiki Okasaka (Google), Tom Olson (ARM), Zhengrong Yao (Ericsson), and the members of the Khronos WebGL Working Group.