

**OpenGL® ES** is a software interface to graphics hardware. The interface consists of a set of procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

- **[n.n.n]** refers to sections and tables in the OpenGL ES 2.0 specification.
- **[n.n.n]** refers to sections in the OpenGL ES Shading Language 1.0 specification.

Specifications are available at [www.khronos.org/registry/gles](http://www.khronos.org/registry/gles)

OpenGL ES Command Syntax [2.3]

Open GL ES commands are formed from a return type, a name, and optionally a type letter i for 32-bit int, or f for 32-bit float, as shown by the prototype below:

```
return-type Name{1234}{i}{v} ([args,] T arg1, . . . , T argN [, args]);
```

The arguments enclosed in brackets ([args,] and [, args]) may or may not be present. The argument type T and the number N of arguments may be indicated by the command name suffixes. N is 1, 2, 3, or 4 if present, or else corresponds to the type letters. If “v” is present, an array of N items is passed by a pointer.

For brevity, the OpenGL documentation and this reference may omit the standard prefixes. The actual names are of the forms: glFunctionName(), GL\_CONSTANT, GLtype

Buffer Objects [2.9]

Buffer objects hold vertex array data or indices in high-performance server memory.

```
void GenBuffers(sizei n, uint *buffers);
void DeleteBuffers(sizei n, const uint *buffers);
```

**Creating and Binding Buffer Objects**  
void **BindBuffer**(enum target, uint buffer);  
target: ARRAY\_BUFFER, ELEMENT\_ARRAY\_BUFFER

**Creating Buffer Object Data Stores**  
void **BufferData**(enum target, sizeiptr size, const void \*data, enum usage);  
usage: STATIC\_DRAW, STREAM\_DRAW, DYNAMIC\_DRAW

Updating Buffer Object Data Stores

void **BufferSubData**(enum target, intptr offset, sizeiptr size, const void \*data);  
target: ARRAY\_BUFFER, ELEMENT\_ARRAY\_BUFFER

**Buffer Object Queries [6.1.6, 6.1.3]**  
boolean **IsBuffer**(uint buffer);  
void **GetBufferParameteriv**(enum target, enum value, T data);  
target: ARRAY\_BUFFER, ELEMENT\_ARRAY\_BUFFER  
value: BUFFER\_SIZE, BUFFER\_USAGE

Viewport and Clipping

**Controlling the Viewport [2.12.1]**  
void **DepthRangef**(clampf n, clampf f);  
void **Viewport**(int x, int y, sizei w, sizei h);

Texturing [3.7]

Shaders support texturing using at least MAX\_VERTEX\_TEXTURE\_IMAGE\_UNITS images for vertex shaders and at least MAX\_TEXTURE\_IMAGE\_UNITS images for fragment shaders.

void **ActiveTexture**(enum texture);  
texture: [TEXTURE0..TEXTUREi] where i = MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS-1

Texture Image Specification [3.7.1]

void **TexImage2D**(enum target, int level, int internalformat, sizei width, sizei height, int border, enum format, enum type, void \*data);  
target: TEXTURE\_2D, TEXTURE\_CUBE\_MAP\_POSITIVE\_{X,Y,Z}, TEXTURE\_CUBE\_MAP\_NEGATIVE\_{X,Y,Z}  
internalformat: ALPHA, LUMINANCE, LUMINANCE\_ALPHA, RGB, RGBA  
format: ALPHA, RGB, RGBA, LUMINANCE, LUMINANCE\_ALPHA  
type: UNSIGNED\_BYTE, UNSIGNED\_SHORT\_5\_6\_5, UNSIGNED\_SHORT\_4\_4\_4\_4, UNSIGNED\_SHORT\_5\_5\_5\_1

**Conversion from RGBA pixel components to internal texture components:**

Base Internal Format	RGBA	Internal Components
ALPHA	A	A
LUMINANCE	R	L
LUMINANCE_ALPHA	R, A	L, A
RGB	R, G, B	R, G, B
RGBA	R, G, B, A	R, G, B, A

Alt. Texture Image Specification Commands [3.7.2]

Texture images may also be specified using image data taken directly from the framebuffer, and rectangular subregions of existing texture images may be respecified.

void **CopyTexImage2D**(enum target, int level, enum internalformat, int x, int y, sizei width, sizei height, int border);  
target: TEXTURE\_2D, TEXTURE\_CUBE\_MAP\_POSITIVE\_{X, Y, Z}, TEXTURE\_CUBE\_MAP\_NEGATIVE\_{X, Y, Z}  
internalformat: See **TexImage2D**

Reading Pixels [4.3.1]

void **ReadPixels**(int x, int y, sizei width, sizei height, enum format, enum type, void \*data);  
format: RGBA type: UNSIGNED\_BYTE  
Note: ReadPixels() also accepts a queriable implementation-defined format/type combination, see [4.3.1].

void **TexSubImage2D**(enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, enum type, void \*data);  
target: TEXTURE\_CUBE\_MAP\_POSITIVE\_{X, Y, Z}, TEXTURE\_CUBE\_MAP\_NEGATIVE\_{X, Y, Z}  
format and type: See **TexImage2D**  
void **CopyTexSubImage2D**(enum target, int level, int xoffset, int yoffset, int x, int y, sizei width, sizei height);  
target: TEXTURE\_2D, TEXTURE\_CUBE\_MAP\_POSITIVE\_{X, Y, Z}, TEXTURE\_CUBE\_MAP\_NEGATIVE\_{X, Y, Z}  
format and type: See **TexImage2D**

Compressed Texture Images [3.7.3]

void **CompressedTexImage2D**(enum target, int level, enum internalformat, sizei width, sizei height, int border, sizei imageSize, void \*data);  
target and internalformat: See **TexImage2D**  
void **CompressedTexSubImage2D**(enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, sizei imageSize, void \*data);  
target and internalformat: See **TexImage2D**

Texture Parameters [3.7.4]

void **TexParameterf**(if)(enum target, enum pname, T param);  
void **TexParameterfv**(enum target, enum pname, T params);  
target: TEXTURE\_2D, TEXTURE\_CUBE\_MAP  
pname: TEXTURE\_WRAP\_{S, T}, TEXTURE\_{MIN, MAG}\_FILTER

Manual Mipmap Generation [3.7.11]

void **GenerateMipmap**(enum target);  
target: TEXTURE\_2D, TEXTURE\_CUBE\_MAP

Texture Objects [3.7.13]

void **BindTexture**(enum target, uint texture);  
void **DeleteTextures**(sizei n, uint \*textures);  
void **GenTextures**(sizei n, uint \*textures);

Enumerated Queries [6.1.3]

void **GetTexParameterf**(if)(enum target, enum value, T data);  
target: TEXTURE\_2D, TEXTURE\_CUBE\_MAP  
value: TEXTURE\_WRAP\_{S, T}, TEXTURE\_{MIN, MAG}\_FILTER

Texture Queries [6.1.4]

boolean **IsTexture**(uint texture);

Errors [2.5]

enum **GetError**( void ); //Returns one of the following:

INVALID_ENUM	Enum argument out of range
INVALID_FRAMEBUFFER_OPERATION	Framebuffer is incomplete
INVALID_VALUE	Numeric argument out of range
INVALID_OPERATION	Operation illegal in current state
OUT_OF_MEMORY	Not enough memory left to execute command
NO_ERROR	No error encountered

GL Data Types [2.3]

GL types are not C types.

GL Type	Minimum Bit Width	Description
boolean	1	Boolean
byte	8	Signed binary integer
ubyte	8	Unsigned binary integer
char	8	Characters making up strings
short	16	Signed 2's complement binary integer
ushort	16	Unsigned binary integer
int	32	Signed 2's complement binary integer
uint	32	Unsigned binary integer
fixed	32	Signed 2's complement 16.16 scaled integer
sizei	32	Non-negative binary integer size
enum	32	Enumerated binary integer value
intptr	ptrbits	Signed 2's complement binary integer
sizeiptr	ptrbits	Non-negative binary integer size
bitfield	32	Bit field
float	32	Floating-point value
clampf	32	Floating-point value clamped to [0; 1]

Vertices

Current Vertex State [2.7]

void **VertexAttrib(1234){f}{v}**(uint index, T values);  
void **VertexAttrib(1234){f}{v}**(uint index, T values);

Vertex Arrays [2.8]

Vertex data may be sourced from arrays that are stored in application memory (via a pointer) or faster GPU memory (in a buffer object).

void **VertexAttribPointer**(uint index, int size, enum type, boolean normalized, sizei stride, const void \*pointer);  
type: BYTE, UNSIGNED\_BYTE, SHORT, UNSIGNED\_SHORT, FIXED, FLOAT  
index: [0, MAX\_VERTEX\_ATTRIBS - 1]

If an ARRAY\_BUFFER is bound, the attribute will be read from the bound buffer, and pointer is treated as an offset within the buffer.

void **EnableVertexAttribArray**(uint index);  
void **DisableVertexAttribArray**(uint index);  
index: [0, MAX\_VERTEX\_ATTRIBS - 1]

void **DrawArrays**(enum mode, int first, sizei count);  
void **DrawElements**(enum mode, sizei count, enum type, void \*indices);  
mode: POINTS, LINE\_STRIP, LINE\_LOOP, LINES, TRIANGLE\_STRIP, TRIANGLE\_FAN, TRIANGLES  
type: UNSIGNED\_BYTE, UNSIGNED\_SHORT

If an ELEMENT\_ARRAY\_BUFFER is bound, the indices will be read from the bound buffer, and indices is treated as an offset within the buffer.

Rasterization [3]

Points [3.3]

Point size is taken from the shader builtin gl\_PointSize and clamped to the implementation-dependent point size range.

Line Segments [3.4]

void **LineWidth**(float width);

Polygons [3.5]

void **FrontFace**(enum dir);  
dir: CCW, CW  
void **CullFace**(enum mode);  
mode: FRONT, BACK, FRONT\_AND\_BACK  
**Enable/Disable**(CULL\_FACE)  
void **PolygonOffset**(float factor, float units);  
**Enable/Disable**(POLYGON\_OFFSET\_FILL)

Pixel Rectangles [3.6, 4.3]

void **PixelStorei**(enum pname, int param);  
pname: UNPACK\_ALIGNMENT, PACK\_ALIGNMENT

# OpenGL ES 2.0 API Quick Reference Card

## Shaders and Programs

### Shader Objects [2.10.1]

```
uint CreateShader(enum type);
    type: VERTEX_SHADER, FRAGMENT_SHADER

void ShaderSource(uint shader, sizei count,
    const char **string, const int *length);

void CompileShader(uint shader);

void ReleaseShaderCompiler(void);

void DeleteShader(uint shader);
```

### Loading Shader Binaries [2.10.2]

```
void ShaderBinary(sizei count, const uint *shaders,
    enum binaryformat, const void *binary, sizei length);
```

### Program Objects [2.10.3]

```
uint CreateProgram(void);

void AttachShader(uint program, uint shader);

void DetachShader(uint program, uint shader);

void LinkProgram(uint program);

void UseProgram(uint program);

void DeleteProgram(uint program);
```

### Shader Variables [2.10.4]

#### Vertex Attributes

```
void GetActiveAttrib(uint program, uint index,
    sizei bufSize, sizei *length, int *size, enum *type,
    char *name);
    *type returns: FLOAT, FLOAT_VEC[2,3,4], FLOAT_MAT[2,3,4]

int GetAttribLocation(uint program, const char *name);
```

```
void BindAttribLocation(uint program, uint index,
    const char *name);
```

#### Uniform Variables

```
int GetUniformLocation(uint program, const char *name);

void GetActiveUniform(uint program, uint index,
    sizei bufSize, sizei *length, int *size, enum *type,
    char *name);
    *type: FLOAT, FLOAT_VEC[2,3,4], INT, INT_VEC[2,3,4], BOOL,
    BOOL_VEC[2,3,4], FLOAT_MAT[2,3,4], SAMPLER_2D,
    SAMPLER_CUBE

void Uniform{1234}{if}(int location, T value);
void Uniform{1234}{if}v(int location, sizei count, T value);
void UniformMatrix{234}fv(int location, sizei count,
    boolean transpose, const float *value);
    transpose: FALSE
```

### Shader Execution (Validation) [2.10.5]

```
void ValidateProgram(uint program);
```

## Shader Queries

### Shader Queries [6.1.8]

```
boolean IsShader(uint shader);

void GetShaderiv(uint shader, enum pname, int *params);
    pname: SHADER_TYPE, DELETE_STATUS, COMPILE_STATUS,
    INFO_LOG_LENGTH, SHADER_SOURCE_LENGTH

void GetAttachedShaders(uint program, sizei maxCount,
    sizei *count, uint *shaders);

void GetShaderInfoLog(uint shader, sizei bufSize,
    sizei *length, char *infoLog);

void GetShaderSource(uint shader, sizei bufSize,
```

```
sizei *length, char *source);

void GetShaderPrecisionFormat(enum shadertype,
    enum precisiontype, int *range, int *precision);
    shadertype: VERTEX_SHADER, FRAGMENT_SHADER
    precision: LOW_FLOAT, MEDIUM_FLOAT, HIGH_FLOAT, LOW_INT,
    MEDIUM_INT, HIGH_INT

void GetVertexAttribfv(uint index, enum pname,
    float *params);
    pname: CURRENT_VERTEX_ATTRIB, VERTEX_ATTRIB_ARRAY_x
    (where x may be BUFFER_BINDING, ENABLED, SIZE, STRIDE, TYPE,
    NORMALIZED)

void GetVertexAttribiv(uint index, enum pname,
    int *params);
    pname: CURRENT_VERTEX_ATTRIB, VERTEX_ATTRIB_ARRAY_x
    (where x may be BUFFER_BINDING, ENABLED, SIZE, STRIDE, TYPE,
    NORMALIZED)

void GetVertexAttribPointerv(uint index, enum pname,
    void **pointer);
    pname: VERTEX_ATTRIB_ARRAY_POINTER

void GetUniformfv(uint program, int location,
    float *params)

void GetUniformiv(uint program, int location,
    int *params)
```

### Program Queries [6.1.8]

```
boolean IsProgram(uint program);

void GetProgramiv(uint program, enum pname, int *params);
    pname: DELETE_STATUS, LINK_STATUS, VALIDATE_STATUS,
    INFO_LOG_LENGTH, ATTACHED_SHADERS,
    ACTIVE_ATTRIBUTES, ACTIVE_ATTRIBUTE_MAX_LENGTH,
    ACTIVE_UNIFORMS, ACTIVE_UNIFORM_MAX_LENGTH

void GetProgramInfoLog(uint program, sizei bufSize,
    sizei *length, char *infoLog);
```

## Per-Fragment Operations

### Scissor Test [4.1.2]

```
Enable/Disable(SCISSOR_TEST)

void Scissor(int left, int bottom, sizei width, sizei height );
```

### Multisample Fragment Operations [4.1.3]

```
Enable/Disable(cap)
    cap: SAMPLE_ALPHA_TO_COVERAGE, SAMPLE_COVERAGE

void SampleCoverage(clampf value, boolean invert);
```

### Stencil Test [4.1.4]

```
Enable/Disable(STENCIL_TEST)

void StencilFunc(enum func, int ref, uint mask);

void StencilFuncSeparate(enum face, enum func, int ref,
    uint mask);

void StencilOp(enum sfail, enum dpfail, enum dppass);

void StencilOpSeparate(enum face, enum sfail, enum dpfail,
    enum dppass);
    face: FRONT, BACK, FRONT_AND_BACK
    sfail, dpfail, and dppass: KEEP, ZERO, REPLACE, INCR, DECR, INVERT,
    INCR_WRAP, DECR_WRAP
    func: NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GREATER, GEQUAL,
    NOTEQUAL
```

### Depth Buffer Test [4.1.5]

```
Enable/Disable(DEPTH_TEST)

void DepthFunc(enum func);
    func: NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GREATER, GEQUAL, NOTEQUAL
```

### Blending [4.1.6]

```
Enable/Disable(BLEND) (applies to all draw buffers)

void BlendEquation(enum mode);

void BlendEquationSeparate(enum modeRGB,
    enum modeAlpha);
    mode, modeRGB, and modeAlpha: FUNC_ADD, FUNC_SUBTRACT,
    FUNC_REVERSE_SUBTRACT

void BlendFuncSeparate(enum srcRGB, enum dstRGB,
    enum srcAlpha, enum dstAlpha);

void BlendFunc(enum src, enum dst);
    dst, dstRGB, and dstAlpha: ZERO, ONE, [ONE_MINUS_]SRC_COLOR,
    [ONE_MINUS_]DST_COLOR, [ONE_MINUS_]SRC_ALPHA,
    [ONE_MINUS_]DST_ALPHA, [ONE_MINUS_]CONSTANT_COLOR,
    [ONE_MINUS_]CONSTANT_ALPHA
    src, srcRGB, srcAlpha: same for dst, plus SRC_ALPHA_SATURATE

void BlendColor(clampf red, clampf green, clampf blue, clampf alpha);
```

### Dithering [4.1.7]

```
Enable/Disable(DITHER)
```

### Clearing the Buffers [4.2.3]

```
void Clear(bitfield buf);
    buf: Bitwise OR of COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT,
    STENCIL_BUFFER_BIT

void ClearColor(clampf r, clampf g, clampf b, clampf a);

void ClearDepthf(clampf d);

void ClearStencil(int s);
```

## Whole Framebuffer Operations

### Fine Control of Buffer Updates [4.2.2]

```
void ColorMask(boolean r, boolean g, boolean b, boolean a);

void DepthMask(boolean mask);

void StencilMask(uint mask);

void StencilMaskSeparate(enum face, uint mask);
    face: FRONT, BACK, FRONT_AND_BACK
```

## Framebuffer Objects

### Binding & Managing Framebuffer Objects [4.4.1]

```
void BindFramebuffer(enum target, uint framebuffer);
    target: FRAMEBUFFER

void DeleteFramebuffers(sizei n, uint *framebuffers);

void GenFramebuffers(sizei n, uint *framebuffers);
```

### Renderbuffer Objects [4.4.2]

```
void BindRenderbuffer(enum target, uint renderbuffer);
    target: RENDERBUFFER

void DeleteRenderbuffers(sizei n, const uint *renderbuffers);

void GenRenderbuffers(sizei n, uint *renderbuffers);

void RenderbufferStorage(enum target,
    enum internalformat, sizei width, sizei height);
    target: RENDERBUFFER
    internalformat: DEPTH_COMPONENT16, RGBA4, RGB5_A1,
    RGB565, STENCIL_INDEX8
```

### Attaching Renderbuffer Images to Framebuffer

```
void FramebufferRenderbuffer(enum target,
    enum attachment, enum renderbuffertarget,
    uint renderbuffer);
    target: FRAMEBUFFER
    attachment: COLOR_ATTACHMENT0, DEPTH_ATTACHMENT,
    STENCIL_ATTACHMENT
    renderbuffertarget: RENDERBUFFER
```

### Attaching Texture Images to a Framebuffer

```
void FramebufferTexture2D(enum target,
    enum attachment, enum textarget, uint texture,
    int level);
    textarget: TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE[X, Y, Z],
    TEXTURE_CUBE_MAP_NEGATIVE[X, Y, Z],
    target: FRAMEBUFFER
    attachment: COLOR_ATTACHMENT0, DEPTH_ATTACHMENT,
    STENCIL_ATTACHMENT
```

### Framebuffer Completeness [4.4.5]

```
enum CheckFramebufferStatus(enum target);
    target: FRAMEBUFFER
    returns: FRAMEBUFFER_COMPLETE or a constant indicating which
    value violates framebuffer completeness
```

### Framebuffer Object Queries [6.1.3, 6.1.7]

```
boolean IsFramebuffer(uint framebuffer);

void GetFramebufferAttachmentParameteriv(enum target,
    enum attachment, enum pname, int *params);
    target: FRAMEBUFFER
    attachment: COLOR_ATTACHMENT0, DEPTH_ATTACHMENT,
    STENCIL_ATTACHMENT
    pname: FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE,
    FRAMEBUFFER_ATTACHMENT_OBJECT_NAME,
    FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL,
    FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE
```

### Renderbuffer Object Queries [6.1.3, 6.1.7]

```
boolean IsRenderbuffer(uint renderbuffer);

void GetRenderbufferParameteriv(enum target,
    enum pname, int *params);
    target: RENDERBUFFER
    pname: RENDERBUFFER_x (where x may be WIDTH, HEIGHT,
    RED_SIZE, GREEN_SIZE, BLUE_SIZE, ALPHA_SIZE, DEPTH_SIZE,
    STENCIL_SIZE, INTERNAL_FORMAT)
```

## Special Functions

### Flush and Finish [5.1]

Flush guarantees that commands issued so far will eventually complete. Finish blocks until all commands issued so far have completed.

```
void Flush(void);

void Finish(void);
```

### Hints [5.2]

Hint controls certain aspects of GL behavior.

```
void Hint(enum target, enum hint);
    target: GENERATE_MIPMAP_HINT
    hint: FASTEST, NICEST, DONT_CARE
```

## State and State Requests

A complete list of symbolic constants for states is shown in the tables in [6.2].

### Simple Queries [6.1.1]

```
void GetBooleanv(enum value,
    boolean *data);

void GetIntegerv(enum value, int *data);

void GetFloatv(enum value, float *data);

boolean IsEnabled(enum value);
```

### Pointer and String Queries [6.1.5]

```
ubyte *GetString(enum name);
    name: VENDOR, RENDERER, VERSION,
    SHADING_LANGUAGE_VERSION,
    EXTENSIONS
```

The OpenGL® ES Shading Language is two closely-related languages which are used to create shaders for the vertex and fragment processors contained in the OpenGL ES processing pipeline.

[n.n.n] and [Table n.n] refer to sections and tables in the OpenGL ES Shading Language 1.0 specification at [www.khronos.org/registry/gles](http://www.khronos.org/registry/gles)

## Types [4.1]

A shader can aggregate these using arrays and structures to build more complex types. There are no pointer types.

### Basic Types

<b>void</b>	no function return value or empty parameter list
<b>bool</b>	Boolean
<b>int</b>	signed integer
<b>float</b>	floating scalar
<b>vec2, vec3, vec4</b>	n-component floating point vector
<b>bvec2, bvec3, bvec4</b>	Boolean vector
<b>ivec2, ivec3, ivec4</b>	signed integer vector
<b>mat2, mat3, mat4</b>	2x2, 3x3, 4x4 float matrix
<b>sampler2D</b>	access a 2D texture
<b>samplerCube</b>	access cube mapped texture

### Structures and Arrays [4.1.8, 4.1.9]

<b>Structures</b>	<pre>struct type-name {     members } struct-name[];</pre> <p>// optional variable declaration, // optionally an array</p>
<b>Arrays</b>	<pre>float foo[3];</pre> <p>* structures and blocks can be arrays * only 1-dimensional arrays supported * structure members can be arrays</p>

## Operators and Expressions

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

	Operator	Description	Associativity
1.	()	parenthetical grouping	N/A
2.	[], (), ., ++, --	array subscript function call & constructor structure field or method selector, swizzler postfix increment and decrement	L - R
3.	++, --, unary	prefix increment and decrement unary	R - L
4.	*	multiplicative	L - R
5.	+	additive	L - R
7.	<, <=, >, >=	relational	L - R
8.	==, !=	equality	L - R
12.	&&	logical and	L - R
13.	^^	logical exclusive or	L - R
14.		logical inclusive or	L - R
15.	?:	selection (Selects one entire operand. Use mix() to select individual components of vectors.)	L - R
16.	=, +=, -=, *=, /=	assignment arithmetic assignments	L - R
17.	,	sequence	L - R

### Vector Components [5.5]

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

<b>{x, y, z, w}</b>	Use when accessing vectors that represent points or normals
<b>{r, g, b, a}</b>	Use when accessing vectors that represent colors
<b>{s, t, p, q}</b>	Use when accessing vectors that represent texture coordinates

## Preprocessor [3.4]

### Preprocessor Directives

The number sign (#) can be immediately preceded or followed in its line by spaces or horizontal tabs.

#	#define	#undef	#if	#ifdef	#ifndef	#else
#elif	#endif	#error	#pragma	#extension	#version	#line

#### Examples of Preprocessor Directives

- "#version 100" in a shader program specifies that the program is written in GLSL ES version 1.00. It is optional. If used, it must occur before anything else in the program other than whitespace or comments.
- #extension extension\_name : behavior, where behavior can be require, enable, warn, or disable; and where extension\_name is the extension supported by the compiler

### Predefined Macros

__LINE__	Decimal integer constant that is one more than the number of preceding new-lines in the current source string
__FILE__	Decimal integer constant that says which source string number is currently being processed.
__VERSION__	Decimal integer, e.g.: 100
GL_ES	Defined and set to integer 1 if running on an OpenGL-ES Shading Language.
GL_FRAGMENT_PRECISION_HIGH	1 if highp is supported in the fragment language, else undefined [4.5.4]

## Qualifiers

### Storage Qualifiers [4.3]

Variable declarations may be preceded by one storage qualifier.

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

### Uniform [4.3.4]

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

```
uniform vec4 lightPosition;
```

### Varying [4.3.5]

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

```
varying vec3 normal;
```

### Parameter Qualifiers [4.4]

Input values are copied in at function call time, output values are copied out at function return time.

<b>none</b>	(Default) same as in
<b>in</b>	For function parameters passed into a function
<b>out</b>	For function parameters passed back out of a function, but not initialized for use when passed in
<b>inout</b>	For function parameters passed both into and out of a function

## Aggregate Operations and Constructors

### Matrix Constructor Examples [5.4]

```
mat2(float)           // init diagonal
mat2(vec2, vec2);      // column-major order
mat2(float, float,    // column-major order
    float, float);
```

### Structure Constructor Example [5.4.3]

```
struct light {float intensity; vec3 pos; };
light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));
```

### Matrix Components [5.6]

Access components of a matrix with array subscripting syntax.

For example:

```
mat4 m;           // m represents a matrix
m[1] = vec4(2.0); // sets second column to all 2.0
m[0][0] = 1.0;    // sets upper left element to 1.0
m[2][3] = 2.0;    // sets 4th element of 3rd column to 2.0
```

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

### Precision and Precision Qualifiers [4.5]

Any floating point, integer, or sampler declaration can have the type preceded by one of these precision qualifiers:

<b>highp</b>	Satisfies minimum requirements for the vertex language. Optional in the fragment language.
<b>mediump</b>	Satisfies minimum requirements for the fragment language. Its range and precision is between that provided by lowp and highp.
<b>lowp</b>	Range and precision can be less than mediump, but still represents all color values for any color channel.

For example:

```
lowp float color;
varying mediump vec2 Coord;
lowp ivec2 foo(lowp mat3);
highp mat4 m;
```

Ranges & precisions for precision qualifiers (FP=floating point):

	FP Range	FP Magnitude Range	FP Precision	Integer Range
<b>highp</b>	$(-2^{62}, 2^{62})$	$(2^{-62}, 2^{62})$	Relative $2^{-16}$	$(-2^{16}, 2^{16})$
<b>mediump</b>	$(-2^{14}, 2^{14})$	$(2^{-14}, 2^{14})$	Relative $2^{-10}$	$(-2^{10}, 2^{10})$
<b>lowp</b>	$(-2, 2)$	$(2^{-8}, 2)$	Absolute $2^{-8}$	$(-2^8, 2^8)$

A precision statement establishes a default precision qualifier for subsequent int, float, and sampler declarations, e.g.:

```
precision highp int;
```

### Invariant Qualifiers Examples [4.6]

#pragma STDGL invariant(all)	Force all output variables to be invariant
invariant gl_Position;	Qualify a previously declared variable
invariant varying mediump vec3 Color;	Qualify as part of a variable declaration

### Order of Qualification [4.7]

When multiple qualifications are present, they must follow a strict order. This order is as follows.

invariant, storage, precision  
storage, parameter, precision

```
v = v * v;           // vector * vector component-wise
m = m +/- m;         // matrix component-wise addition/subtraction
m = m * m;           // linear algebraic multiply
m = v * m;           // row vector * matrix linear algebraic multiply
m = m * v;           // matrix * column vector linear algebraic multiply
f = dot(v, v);        // vector dot product
v = cross(v, v);       // vector cross product
m = matrixCompMult(m, m); // component-wise multiply
```

### Structure Operations [5.7]

Select structure fields using the period (.) operator. Other operators include:

.	field selector
== !=	equality
=	assignment

### Array Operations [4.1.9]

Array elements are accessed using the array subscript operator "[ ]". For example:

```
diffuseColor += lightIntensity[3] * NdotL;
```



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

Shader programs use Special Variables to communicate with fixed-function parts of the pipeline. Output Special Variables may be read back after writing. Input Special Variables are read-only. All Special Variables have global scope.

**Vertex Shader Special Variables [7.1]****Outputs:**

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

**Fragment Shader Special Variables [7.2]**

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

**Inputs:**

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

**Outputs:**

Variable	Description	Units or coordinate system
mediump vec4 <code>gl_FragColor</code> ;	fragment color	RGBA color
mediump vec4 <code>gl_FragData[n]</code>	fragment color for color attachment <i>n</i>	RGBA color

**Built-In Constants With Minimum Values [7.4]**

Built-in Constant	Minimum value
const mediump int <code>gl_MaxVertexAttribs</code>	8
const mediump int <code>gl_MaxVertexUniformVectors</code>	128
const mediump int <code>gl_MaxVaryingVectors</code>	8
const mediump int <code>gl_MaxVertexTextureImageUnits</code>	0
const mediump int <code>gl_MaxCombinedTextureImageUnits</code>	8
const mediump int <code>gl_MaxTextureImageUnits</code>	8
const mediump int <code>gl_MaxFragmentUniformVectors</code>	16
const mediump int <code>gl_MaxDrawBuffers</code>	1

**Built-In Uniform State [7.5]**

Specifies depth range in window coordinates. If an implementation does not support high precision in the fragment language, and state is listed as highp, then that state will only be available as mediump in the fragment language.

```
struct gl_DepthRangeParameters {
    highp float near;    // n
    highp float far;     // f
    highp float diff;    // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;
```

**Built-In Functions****Angle & Trigonometry Functions [8.1]**

Component-wise operation. Parameters specified as *angle* are assumed to be in units of radians. T is float, vec2, vec3, vec4.

T <code>radians</code> (T <i>degrees</i> )	degrees to radians
T <code>degrees</code> (T <i>radians</i> )	radians to degrees
T <code>sin</code> (T <i>angle</i> )	sine
T <code>cos</code> (T <i>angle</i> )	cosine
T <code>tan</code> (T <i>angle</i> )	tangent
T <code>asin</code> (T <i>x</i> )	arc sine
T <code>acos</code> (T <i>x</i> )	arc cosine
T <code>atan</code> (T <i>y</i> , T <i>x</i> )	arc tangent
T <code>atan</code> (T <i>y_over_x</i> )	

**Exponential Functions [8.2]**

Component-wise operation. T is float, vec2, vec3, vec4.

T <code>pow</code> (T <i>x</i> , T <i>y</i> )	$x^y$
T <code>exp</code> (T <i>x</i> )	$e^x$
T <code>log</code> (T <i>x</i> )	ln
T <code>exp2</code> (T <i>x</i> )	$2^x$
T <code>log2</code> (T <i>x</i> )	$\log_2$
T <code>sqrt</code> (T <i>x</i> )	square root
T <code>inversesqrt</code> (T <i>x</i> )	inverse square root

**Common Functions [8.3]**

Component-wise operation. T is float, vec2, vec3, vec4.

T <code>abs</code> (T <i>x</i> )	absolute value
T <code>sign</code> (T <i>x</i> )	returns -1.0, 0.0, or 1.0
T <code>floor</code> (T <i>x</i> )	nearest integer $\leq x$
T <code>ceil</code> (T <i>x</i> )	nearest integer $\geq x$
T <code>fract</code> (T <i>x</i> )	$x - \text{floor}(x)$
T <code>mod</code> (T <i>x</i> , T <i>y</i> )	modulus
T <code>mod</code> (T <i>x</i> , float <i>y</i> )	
T <code>min</code> (T <i>x</i> , T <i>y</i> )	minimum value
T <code>min</code> (T <i>x</i> , float <i>y</i> )	
T <code>max</code> (T <i>x</i> , T <i>y</i> )	maximum value
T <code>max</code> (T <i>x</i> , float <i>y</i> )	
T <code>clamp</code> (T <i>x</i> , T <i>minVal</i> , T <i>maxVal</i> )	
T <code>clamp</code> (T <i>x</i> , float <i>minVal</i> , float <i>maxVal</i> )	<code>min(max(x, minVal), maxVal)</code>
T <code>mix</code> (T <i>x</i> , T <i>y</i> , T <i>a</i> )	linear blend of <i>x</i> and <i>y</i>
T <code>mix</code> (T <i>x</i> , T <i>y</i> , float <i>a</i> )	
T <code>step</code> (T <i>edge</i> , T <i>x</i> )	0.0 if $x < \text{edge}$ , else 1.0
T <code>step</code> (float <i>edge</i> , T <i>x</i> )	
T <code>smoothstep</code> (T <i>edge0</i> , T <i>edge1</i> , T <i>x</i> )	
T <code>smoothstep</code> (float <i>edge0</i> , float <i>edge1</i> , T <i>x</i> )	clip and smooth

**Geometric Functions [8.4]**

These functions operate on vectors as vectors, not component-wise. T is float, vec2, vec3, vec4.

float <code>length</code> (T <i>x</i> )	length of vector
float <code>distance</code> (T <i>p0</i> , T <i>p1</i> )	distance between points
float <code>dot</code> (T <i>x</i> , T <i>y</i> )	dot product
vec3 <code>cross</code> (vec3 <i>x</i> , vec3 <i>y</i> )	cross product
T <code>normalize</code> (T <i>x</i> )	normalize vector to length 1
T <code>faceforward</code> (T <i>N</i> , T <i>I</i> , T <i>Nref</i> )	returns <i>N</i> if <code>dot(Nref, I) &lt; 0</code> , else $-N$
T <code>reflect</code> (T <i>I</i> , T <i>N</i> )	reflection direction $I - 2 * \text{dot}(N, I) * N$
T <code>refract</code> (T <i>I</i> , T <i>N</i> , float <i>eta</i> )	refraction vector

**Matrix Functions [8.5]**

Type mat is any matrix type.

mat `matrixCompMult`(mat *x*, mat *y*) multiply *x* by *y* component-wise

**Vector Relational Functions [8.6]**

Compare *x* and *y* component-wise. Sizes of input and return vectors for a particular call must match. Type bvec is bvecn; vec is vecn; ivec is ivec $n$  (where *n* is 2, 3, or 4). T is the union of vec and ivec.

bvec <code>lessThan</code> (T <i>x</i> , T <i>y</i> )	$x < y$
bvec <code>lessThanEqual</code> (T <i>x</i> , T <i>y</i> )	$x \leq y$
bvec <code>greaterThan</code> (T <i>x</i> , T <i>y</i> )	$x > y$
bvec <code>greaterThanEqual</code> (T <i>x</i> , T <i>y</i> )	$x \geq y$
bvec <code>equal</code> (T <i>x</i> , T <i>y</i> )	$x == y$
bvec <code>equal</code> (bvec <i>x</i> , bvec <i>y</i> )	
bvec <code>notEqual</code> (T <i>x</i> , T <i>y</i> )	$x \neq y$
bvec <code>notEqual</code> (bvec <i>x</i> , bvec <i>y</i> )	
bool <code>any</code> (bvec <i>x</i> )	true if any component of <i>x</i> is true
bool <code>all</code> (bvec <i>x</i> )	true if all components of <i>x</i> are true
bvec <code>not</code> (bvec <i>x</i> )	logical complement of <i>x</i>

**Texture Lookup Functions [8.7]**

Available only in vertex shaders.

```
vec4 texture2DLod(sampler2D sampler, vec2 coord, float lod)
vec4 texture2DProjLod(sampler2D sampler, vec3 coord, float lod)
vec4 texture2DProjLod(sampler2D sampler, vec4 coord, float lod)
vec4 textureCubeLod(samplerCube sampler, vec3 coord, float lod)
```

Available only in fragment shaders.

```
vec4 texture2D(sampler2D sampler, vec2 coord, float bias)
vec4 texture2DProj(sampler2D sampler, vec3 coord, float bias)
vec4 texture2DProj(sampler2D sampler, vec4 coord, float bias)
vec4 textureCube(samplerCube sampler, vec3 coord, float bias)
```

Available in vertex and fragment shaders.

```
vec4 texture2D(sampler2D sampler, vec2 coord)
vec4 texture2DProj(sampler2D sampler, vec3 coord)
vec4 texture2DProj(sampler2D sampler, vec4 coord)
vec4 textureCube(samplerCube sampler, vec3 coord)
```

**Statements and Structure****Iteration and Jumps [6]**

Function Call	call by value-return
Iteration	for (;;) { break, continue } while ( ) { break, continue } do { break, continue } while ( );
Selection	if ( ) { } if ( ) { } else { }
Jump	break, continue, return discard // Fragment shader only
Entry	void main()

**Sample Program**

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

**Vertex Shader**

```
uniform mat4 mvp_matrix; // model-view-projection matrix
uniform mat3 normal_matrix; // normal matrix
uniform vec3 ec_light_dir; // light direction in eye coords

attribute vec4 a_vertex; // vertex position
attribute vec3 a_normal; // vertex normal
attribute vec2 a_texcoord; // texture coordinates
```

```
varying float v_diffuse;
varying vec2 v_texcoord;
```

```
void main(void)
{
    // put vertex normal into eye coords
    vec3 ec_normal = normalize(normal_matrix * a_normal);
```

```
    // emit diffuse scale factor, texcoord, and position
    v_diffuse = max(dot(ec_light_dir, ec_normal), 0.0);
    v_texcoord = a_texcoord;
    gl_Position = mvp_matrix * a_vertex;
}
```

**Fragment Shader**

```
precision mediump float;
```

```
uniform sampler2D t_reflectance;
uniform vec4 i_ambient;

varying float v_diffuse;
varying vec2 v_texcoord;
```

```
void main (void)
{
    vec4 color = texture2D(t_reflectance, v_texcoord);
    gl_FragColor = color * (vec4(v_diffuse) + i_ambient);
}
```

OpenGL ES is a registered trademark of Silicon Graphics International, used under license by Khronos Group.

The Khronos Group is an industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics and dynamic media on a wide variety of platforms and devices.

See [www.khronos.org](http://www.khronos.org) to learn more about the Khronos Group.

See [www.khronos.org/opengles](http://www.khronos.org/opengles) to learn more about OpenGL ES.