Name

abs — return the absolute value of the parameter

Declaration

```
genType abs (x);
genType x;
genIType abs (x);
genIType x;
```

Parameters

x Specify the value of which to return the absolute.

Description

`abs` returns *x* if $x \geq 0$, otherwise returns $-x$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
abs (genType)	✔	✔
abs (genIType)	-	✔

See Also

sign

Copyright

Name

acos — return the arccosine of the parameter

Declaration

```
genType acos (x);
```

```
genType x;
```

Parameters

x Specify the value whose arccosine to return.

Description

`acos` returns the angle whose trigonometric cosine is x. The range of values returned by `acos` is $[0, \pi]$. The result is undefined if $\left| x \right| > 1$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
acos	✔	✔

See Also

sin, cos, tan, asin, atan

Copyright

Name

acosh — return the arc hyperbolic cosine of the parameter

Declaration

```
genType acosh (x);

genType x;
```

Parameters

x Specify the value whose arc hyperbolic cosine to return.

Description

`acosh` returns the arc hyperbolic cosine of x; the non-negative inverse of cosh. The result is undefined if $x < 1$.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
acosh	-	✔

See Also

sin, cos, sinh, cosh

Copyright

Name

all — check whether all elements of a boolean vector are true

Declaration

```
bool all (x);

bvec x;
```

Parameters

x Specifies the vector to be tested for truth.

Description

`all` returns true if all elements of *x* are true and false otherwise. It is functionally equivalent to:

```
bool all(bvec x)       // bvec can be bvec2, bvec3 or bvec4
{
    bool result = true;
    int i;
    for (i = 0; i < x.length(); ++i)
    {
        result &= x[i];
    }
    return result;
}
```

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
all	✔	✔

See Also

any, not

Copyright

Name

any — check whether any element of a boolean vector is true

Declaration

`bool **any** (*x*);`

`bvec *x*;`

Parameters

x Specifies the vector to be tested for truth.

Description

`any` returns true if any element of *x* is true and false otherwise. It is functionally equivalent to:

```
bool any(bvec x)       // bvec can be bvec2, bvec3 or bvec4
{
    bool result = false;
    int i;
    for (i = 0; i < x.length(); ++i)
    {
        result |= x[i];
    }
    return result;
}
```

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
any	✔	✔

See Also

all, not

Copyright

Name

asin — return the arcsine of the parameter

Declaration

```
genType asin (x);
```

```
genType x;
```

Parameters

x Specify the value whose arcsine to return.

Description

`asin` returns the angle whose trigonometric sine is x. The range of values returned by `asin` is $[-{\pi \over 2}, {\pi \over 2}]$. The result is undefined if $\left| x \right| > 1$.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
asin	✔	✔

See Also

sin, cos, tan, acos, atan

Copyright

Name

asinh — return the arc hyperbolic sine of the parameter

Declaration

```
genType asinh (x);
genType x;
```

Parameters

x Specify the value whose arc hyperbolic sine to return.

Description

`asinh` returns the arc hyperbolic sine of ; the inverse of sinh.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
asinh	-	✔

See Also

sin, cos, sinh, cosh

Copyright

Name

atan — return the arc-tangent of the parameters

Declaration

```
genType atan (y, x);

genType y;
genType x;

genType atan (y_over_x);

genType y_over_x;
```

Parameters

y	Specify the numerator of the fraction whose arctangent to return.
x	Specify the denominator of the fraction whose arctangent to return.
y_over_x	Specify the fraction whose arctangent to return.

Description

`atan` returns either the angle whose trigonometric arctangent is $y \over x$ or *y_over_x*, depending on which overload is invoked. In the first overload, the signs of y and x are used to determine the quadrant that the angle lies in. The value returned by `atan` in this case is in the range $[-\pi,\pi]$. The result is undefined if $x = 0$.

For the second overload, `atan` returns the angle whose tangent is *y_over_x*. The value returned in this case is in the range $[-{\pi \over 2 },{\pi \over 2}]$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
atan	✔	✔

See Also

sin, cos, tan, asin, acos

Copyright

Name

atanh — return the arc hyperbolic tangent of the parameter

Declaration

```
genType atanh (x);
```

```
genType x;
```

Parameters

x Specify the value whose arc hyperbolic tangent to return.

Description

`atanh` returns the arc hyperbolic tangent of x; the inverse of tanh. The result is undefined if $\left| x \right| > 1$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
atanh	-	✓

See Also

sin, cos, sinh, cosh

Copyright

Name

ceil — find the nearest integer that is greater than or equal to the parameter

Declaration

```
genType ceil (x);
```

```
genType x;
```

Parameters

x Specify the value to evaluate.

Description

`ceil` returns a value equal to the nearest integer that is greater than or equal to *x*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
ceil (genType)	✔	✔

See Also

floor, round

Copyright

Name

clamp — constrain a value to lie between two further values

Declaration

```
genType clamp (x, minVal, maxVal);

genType x;
genType minVal;
genType maxVal;

genType clamp (x, minVal, maxVal);

genType x;
float minVal;
float maxVal;

genIType clamp (x, minVal, maxVal);

genIType x;
genIType minVal;
genIType maxVal;

genIType clamp (x, minVal, maxVal);

genIType x;
int minVal;
int maxVal;

genUType clamp (x, minVal, maxVal);

genUType x;
genUType minVal;
genUType maxVal;

genUType clamp (x, minVal, maxVal);

genUType x;
uint minVal;
uint maxVal;
```

Parameters

x	Specify the value to constrain.
minVal	Specify the lower end of the range into which to constrain *x*.
maxVal	Specify the upper end of the range into which to constrain *x*.

Description

`clamp` returns the value of *x* constrained to the range *minVal* to *maxVal*. The returned value is computed as min(max(*x*, *minVal*), *maxVal*). The result is undefined if *minVal* $\geq$ *maxVal*.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
clamp (genType)	✔	✔
clamp (genIType)	-	✔
clamp (genUType)	-	✔

See Also

min, max

Copyright

Name

cos — return the cosine of the parameter

Declaration

```
genType cos (angle);
genType angle;
```

Parameters

angle Specify the quantity, in radians, of which to return the cosine.

Description

`cos` returns the trigonometric cosine of *angle*.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
cos	✔	✔

See Also

sin, tan, acos, asin, atan

Copyright

Name

cosh — return the hyperbolic cosine of the parameter

Declaration

```
genType **cosh** (*x*);

genType *x*;
```

Parameters

x Specify the value whose hyperbolic cosine to return.

Description

`cosh` returns the hyperbolic cosine of . The hyperbolic cosine of is computed as .

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
cosh	-	✔

See Also

sin, cos, sinh

Copyright

Name

cross — calculate the cross product of two vectors

Declaration

```
vec3 cross (x, y);

vec3 x;
vec3 y;
```

Parameters

x	Specifies the first of two vectors
y	Specifies the second of two vectors

Description

`cross` returns the cross product of two vectors, *x* and *y*, i.e. $\begin{pmatrix} { x[1] \times y[2] - y[1] \times x[2] } \\ { x[2] \times y[0] - y[2] \times x[0] } \\ { x[0] \times y[1] - y[0] \times x[1] } \end{pmatrix}$.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
cross (vec3)	✔	✔

See Also

dot

Copyright

Name

dFdx, dFdy — return the partial derivative of an argument with respect to x or y

Declaration

```
genType dFdx (p);

genType p;

genType dFdy (p);

genType p;
```

Parameters

p Specifies the expression of which to take the partial derivative.

Description

Available only in the fragment shader, `dFdx` and `dFdy` return the partial derivative of expression *p* in x and y, respectively. Deviatives are calculated using local differencing. Expressions that imply higher order derivatives such as `dFdx(dFdx(n))` have undefined results, as do mixed-order derivatives such as `dFdx(dFdy(n))`. It is assumed that the expression *p* is continuous and therefore, expressions evaluated via non-uniform control flow may be undefined.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
dFdx	-	✔
dFdy	-	✔

See Also

fwidth

Copyright

Name

degrees — convert a quantity in radians to degrees

Declaration

```
genType degrees (radians);

genType radians;
```

Parameters

radians Specify the quantity, in radians, to be converted to degrees.

Description

`degrees` converts a quantity, specified in radians into degrees. That is, the return value is .

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
degrees	✔	✔
radians	✔	✔

See Also

radians

Copyright

Name

determinant — calculate the determinant of a matrix

Declaration

```
float determinant (m);

mat2 m;

float determinant (m);

mat3 m;

float determinant (m);

mat4 m;
```

Parameters

m Specifies the matrix of which to take the determinant.

Description

`determinant` returns the determinant of the matrix *m*.

Version Support

Function Name	**OpenGL ES Shading Language Version**	
	1.00	**3.00**
determinant (float)	-	✔

See Also

transpose, inverse

Copyright

Name

distance — calculate the distance between two points

Declaration

```
float distance (p0, p1);

genType p0;
genType p1;
```

Parameters

p0	Specifies the first of two points
p1	Specifies the second of two points

Description

`distance` returns the distance between the two points *p0* and *p1*. i.e., `length(p0 - p1);`

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
distance (genType)	✔	✔

See Also

length, normalize

Copyright

Name

dot — calculate the dot product of two vectors

Declaration

```
float dot (x, y);

genType x;
genType y;
```

Parameters

x Specifies the first of two vectors

y Specifies the second of two vectors

Description

`dot` returns the dot product of two vectors, *x* and *y*. i.e.,

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
dot (genType)	✔	✔

See Also

cross

Copyright

Name

equal — perform a component-wise equal-to comparison of two vectors

Declaration

```
bvec equal (x, y);

vec x;
vec y;

bvec equal (x, y);

bvec x;
bvec y;

bvec equal (x, y);

ivec x;
ivec y;

bvec equal (x, y);

uvec x;
uvec y;
```

Parameters

x Specifies the first vector to be used in the comparison operation.

y Specifies the second vector to be used in the comparison operation.

Description

`equal` returns a boolean vector in which each element *i* is computed as *x*[*i*] == *y*[*i*].

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
equal (vec)	✔	✔
equal (bvec)	✔	✔
equal (ivec)	✔	✔
equal (uvec)	-	✔

See Also

lessThan, lessThanEqual, greaterThan, greaterThanEqual, notEqual, any, all, not

Copyright

Name

exp — return the natural exponentiation of the parameter

Declaration

```
genType exp (x);

genType x;
```

Parameters

x Specify the value to exponentiate.

Description

`exp` returns the natural exponentiation of *x*. i.e., .

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
exp	✔	✔

See Also

exp2, pow

Copyright

Name

exp2 — return 2 raised to the power of the parameter

Declaration

```
genType exp2 (x);
```

```
genType x;
```

Parameters

x Specify the value of the power to which 2 will be raised.

Description

`exp2` returns 2 raised to the power of *x*. i.e., .

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
exp2	✔	✔

See Also

exp, log, log2

Copyright

Name

faceforward — return a vector pointing in the same direction as another

Declaration

```
genType faceforward (N, I, Nref);

genType N;
genType I;
genType Nref;
```

Parameters

N	Specifies the vector to orient.
I	Specifies the incident vector.
Nref	Specifies the reference vector.

Description

`faceforward` orients a vector to point away from a surface as defined by its normal. If dot(*Nref*, *I*) < 0 `faceforward` returns *N*, otherwise it returns -*N*.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	3.00
faceforward (genType)	✔	✔

See Also

reflect, refract

Copyright

Name

floatBitsToInt — produce the encoding of a floating point value as an integer

Declaration

```
genIType floatBitsToInt (x);

genType x;

genUType floatBitsToUint (x);

genType x;
```

Parameters

x Specifies the value whose floating point encoding to return.

Description

`floatBitsToInt` and `floatBitsToUint` return the encoding of their floating-point parameters as highp `int` or `uint`, respectively. The floating-point bit-level representation is preserved. For mediump and lowp, the value is first converted to highp floating point and the encoding of that value is returned.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
floatBitsToInt	-	✔
floatBitsToUInt	-	✔

See Also

intBitsToFloat, uintBitsToFloat

Copyright

Name

floor — find the nearest integer less than or equal to the parameter

Declaration

```
genType floor (x);

genType x;
```

Parameters

x Specify the value to evaluate.

Description

`floor` returns a value equal to the nearest integer that is less than or equal to *x*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
floor (genType)	✔	✔

See Also

trunc, round

Copyright

Name

fract — compute the fractional part of the argument

Declaration

```
genType fract (x);
genType x;
```

Parameters

x Specify the value to evaluate.

Description

`fract` returns the fractional part of *x*. This is calculated as *x* - floor(*x*).

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
fract (genType)	✔	✔

See Also

floor, round

Copyright

Name

fwidth — return the sum of the absolute derivatives in x and y

Declaration

```
genType fwidth (p);

genType p;
```

Parameters

p Specifies the expression of which to take the partial derivative.

Description

Available only in the fragment shader, `fwidth` returns the sum of the absolute derivatives in x and y using local differencing for the input argument *p*. It is equivalent to `abs(dFdx(p)) + abs(dFdy(p))`.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
fwidth	-	✔

See Also

abs, dFdx, dFdy

Copyright

Name

glActiveTexture — select active texture unit

C Specification

```
void glActiveTexture (texture);

GLenum texture;
```

Parameters

texture	Specifies which texture unit to make active. The number of texture units is implementation-dependent, but must be at least 32. *texture* must be one of `GL_TEXTURE`*i*, where *i* ranges from zero to the value of `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS` minus one. The initial value is `GL_TEXTURE0`.

Description

`glActiveTexture` selects which texture unit subsequent texture state calls will affect. The number of texture units an implementation supports is implementation-dependent, but must be at least 32.

Errors

`GL_INVALID_ENUM` is generated if *texture* is not one of `GL_TEXTURE`*i*, where *i* ranges from zero to the value of `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS` minus one.

Associated Gets

glGet with argument `GL_ACTIVE_TEXTURE`, or `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glActiveTexture	✔	✔

See Also

glGenTextures, glBindTexture, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glDeleteTextures glIsTexture, glTexImage2D, glTexImage3D, glTexSubImage2D, glTexSubImage3D, glTexParameter,

Copyright

Name

glAttachShader — Attaches a shader object to a program object

C Specification

```
void glAttachShader (program, shader);

GLuint program;
GLuint shader;
```

Parameters

program	Specifies the program object to which a shader object will be attached.
shader	Specifies the shader object that is to be attached.

Description

In order to create a complete shader program, there must be a way to specify the list of things that will be linked together. Program objects provide this mechanism. Shaders that are to be linked together in a program object must first be attached to that program object. `glAttachShader` attaches the shader object specified by *shader* to the program object specified by *program*. This indicates that *shader* will be included in link operations that will be performed on *program*.

All operations that can be performed on a shader object are valid whether or not the shader object is attached to a program object. It is permissible to attach a shader object to a program object before source code has been loaded into the shader object or before the shader object has been compiled. It is not permissible to attach multiple shader objects of the same type. It is permissible to attach a shader object to more than one program object. If a shader object is deleted while it is attached to a program object, it will be flagged for deletion, and deletion will not occur until glDetachShader is called to detach it from all program objects to which it is attached.

Errors

`GL_INVALID_VALUE` is generated if either *program* or *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_OPERATION` is generated if *shader* is already attached to *program*.

`GL_INVALID_OPERATION` is generated if a shader of the same type as *shader* is already attached to *program*.

Associated Gets

glGetAttachedShaders with the handle of a valid program object

glGetShaderInfoLog

glGetShaderSource

glIsProgram

glIsShader

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glAttachShader	✔	✔

See Also

glCompileShader, glCreateShader, glDeleteShader, glDetachShader, glLinkProgram, glShaderSource

Copyright

Name

glBeginQuery — delimit the boundaries of a query object

C Specification

```
void glBeginQuery (target, id);

GLenum target;
GLuint id;

void glEndQuery (target);

GLenum target;
```

Parameters for `glBeginQuery`

`target`	Specifies the target type of query object established between `glBeginQuery` and the subsequent `glEndQuery`. The symbolic constant must be one of `GL_ANY_SAMPLES_PASSED`, `GL_ANY_SAMPLES_PASSED_CONSERVATIVE`, or `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`.
`id`	Specifies the name of a query object.

Parameters for `glEndQuery`

`target`	Specifies the target type of query object to be concluded. The symbolic constant must be one of `GL_ANY_SAMPLES_PASSED`, `GL_ANY_SAMPLES_PASSED_CONSERVATIVE`, or `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`.

Description

`glBeginQuery` and glEndQuery delimit the boundaries of a query object. *`query`* must be a name previously returned from a call to glGenQueries. If a query object with name *`id`* does not yet exist it is created with the type determined by *`target`*. *`target`* must be one of `GL_ANY_SAMPLES_PASSED`, `GL_ANY_SAMPLES_PASSED_CONSERVATIVE`, or `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`. The behavior of the query object depends on its type and is as follows.

If *`target`* is `GL_ANY_SAMPLES_PASSED`, *`id`* must be an unused name, or the name of an existing boolean occlusion query object. When `glBeginQuery` is executed, the query object's samples-passed flag is reset to `GL_FALSE`. Subsequent rendering causes the flag to be set to `GL_TRUE` if any sample passes the depth test. When `glEndQuery` is executed, the samples-passed flag is assigned to the query object's result value. This value can be queried by calling glGetQueryObjectuiv with *`pname`* `GL_QUERY_RESULT`.

If *`target`* is `GL_ANY_SAMPLES_PASSED_CONSERVATIVE`, *`id`* must be an unused name, or the name of an existing boolean occlusion query object. When `glBeginQuery` is executed, the query object's samples-passed flag is reset to `GL_FALSE`. Subsequent rendering causes the flag to be set to `GL_TRUE` if any sample passes the depth test. The implementation may choose to use a less precision version of the test, which can additionally set the samples-passed flag to `GL_TRUE` in some other implementation-dependent cases. When `glEndQuery` is executed, the samples-passed flag is assigned to the query object's result value. This value can be queried by calling glGetQueryObjectuiv with *`pname`* `GL_QUERY_RESULT`.

If *`target`* is `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`, *`id`* must be an unused name, or the name of an existing primitive query object previously bound to the `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` query binding. When `glBeginQuery` is executed, the query object's primitives-written counter is reset to 0. Subsequent rendering will increment the counter once for every vertex that is written into the bound transform feedback buffer(s). If transform feedback mode is not activated between the call to `glBeginQuery` and `glEndQuery`, the counter will not be incremented. When `glEndQuery` is executed, the primitives-written counter is assigned to the query object's result value. This value can be queried by calling glGetQueryObjectuiv with *pname* `GL_QUERY_RESULT`.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not one of the accepted tokens.

`GL_INVALID_OPERATION` is generated if `glBeginQuery` is executed while a query object of the same *`target`* is already active. Note: `GL_ANY_SAMPLES_PASSED` and `GL_ANY_SAMPLES_PASSED_CONSERVATIVE` alias to the same target for the purposes of this error.

`GL_INVALID_OPERATION` is generated if `glEndQuery` is executed when a query object of the same *`target`* is not active.

`GL_INVALID_OPERATION` is generated if *`id`* is 0.

`GL_INVALID_OPERATION` is generated if *`id`* not a name returned from a previous call to `glGenQueries`, or if such a name has since been deleted with `glDeleteQueries`.

`GL_INVALID_OPERATION` is generated if *`id`* is the name of an already active query object.

`GL_INVALID_OPERATION` is generated if *`id`* refers to an existing query object whose type does not does not match *`target`*.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glBeginQuery`	-	✔
`glEndQuery`	-	✔

See Also

glDeleteQueries, glGenQueries, glGetQueryiv, glGetQueryObjectuiv, glIsQuery

Copyright

Name

glBeginTransformFeedback — start transform feedback operation

C Specification

`void` **`glBeginTransformFeedback`** `(`*`primitiveMode`*`);`

`GLenum` *`primitiveMode`*`;`

`void` **`glEndTransformFeedback`** `(`*`void`*`);`

`void``;`

Parameters for `glBeginTransformFeedback`

`primitiveMode`	Specify the output type of the primitives that will be recorded into the buffer objects that are bound for transform feedback.

Description

Transform feedback mode captures the values of varying variables written by the vertex shader. Transform feedback is said to be active after a call to `glBeginTransformFeedback` until a subsequent call to `glEndTransformFeedback`. Transform feedback commands must be paired. An implicit glResumeTransformFeedback is performed by `glEndTransformFeedback` if the transform feedback is paused. Transform feedback is restricted to non-indexed `GL_POINTS`, `GL_LINES`, and `GL_TRIANGLES`.

While transform feedback is active the *`mode`* parameter to glDrawArrays must exactly match the *`primitiveMode`* specified by `glBeginTransformFeedback`.

Errors

`GL_INVALID_OPERATION` is generated if `glBeginTransformFeedback` is executed while transform feedback is active.

`GL_INVALID_ENUM` is generated by `glBeginTransformFeedback` if *`primitiveMode`* is not one of `GL_POINTS`, `GL_LINES`, or `GL_TRIANGLES`.

`GL_INVALID_OPERATION` is generated if `glEndTransformFeedback` is executed while transform feedback is not active.

`GL_INVALID_OPERATION` is generated by glDrawArrays and glDrawArraysInstanced if transform feedback is active and *`mode`* does not exactly match *`primitiveMode`*.

`GL_INVALID_OPERATION` is generated by glDrawElements, glDrawElementsInstanced, and glDrawRangeElements if transform feedback is active and not paused.

`GL_INVALID_OPERATION` is generated by `glBeginTransformFeedback` if any binding point used in transform feedback mode does not have a buffer object bound. In interleaved mode, only the first buffer object binding point is ever written to.

`GL_INVALID_OPERATION` is generated by `glBeginTransformFeedback` if no binding points would be used, either because no program object is active of because the active program object has specified no varying variables to record.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
`glBeginTransformFeedback`	-	✔
`glEndTransformFeedback`	-	✔

See Also

glPauseTransformFeedback, glResumeTransformFeedback

Copyright

Name

glBindAttribLocation — Associates a generic vertex attribute index with a named attribute variable

C Specification

```
void glBindAttribLocation (program, index, name);

GLuint program;
GLuint index;
const GLchar *name;
```

Parameters

program	Specifies the handle of the program object in which the association is to be made.
index	Specifies the index of the generic vertex attribute to be bound.
name	Specifies a null terminated string containing the name of the vertex shader attribute variable to which *index* is to be bound.

Description

`glBindAttribLocation` is used to associate a user-defined attribute variable in the program object specified by *program* with a generic vertex attribute index. The name of the user-defined attribute variable is passed as a null terminated string in *name*. The generic vertex attribute index to be bound to this variable is specified by *index*. When *program* is made part of current state, values provided via the generic vertex attribute *index* will modify the value of the user-defined attribute variable specified by *name*.

If *name* refers to a matrix attribute variable, *index* refers to the first column of the matrix. Other matrix columns are then automatically bound to locations *index+1* for a matrix of type `mat2`; *index+1* and *index+2* for a matrix of type `mat3`; and *index+1*, *index+2*, and *index+3* for a matrix of type `mat4`.

This command makes it possible for vertex shaders to use descriptive names for attribute variables rather than generic variables that are numbered from zero to the value of `GL_MAX_VERTEX_ATTRIBS` minus one. The values sent to each generic attribute index are part of current state. If a different program object is made current by calling glUseProgram, the generic vertex attributes are tracked in such a way that the same values will be observed by attributes in the new program object that are also bound to *index*.

Attribute variable name-to-generic attribute index bindings for a program object can be explicitly assigned at any time by calling `glBindAttribLocation`. Attribute bindings do not go into effect until glLinkProgram is called. After a program object has been linked successfully, the index values for generic attributes remain fixed (and their values can be queried) until the next link command occurs.

Any attribute binding that occurs after the program object has been linked will not take effect until the next time the program object is linked.

Notes

`glBindAttribLocation` can be called before any vertex shader objects are bound to the specified program object. It is also permissible to bind a generic attribute index to an attribute variable name that is never used in a vertex shader.

If *name* was bound previously, that information is lost. Thus you cannot bind one user-defined attribute variable to multiple indices, but you can bind multiple user-defined attribute variables to the same index.

Applications are allowed to bind more than one user-defined attribute variable to the same generic vertex attribute index. This is called *aliasing*, and it is allowed only if just one of the aliased attributes is active in the executable program, or if no path through the shader consumes more than one attribute of a set of attributes aliased to the same location. The compiler and linker are allowed to assume that no aliasing is done and are free to employ optimizations that work only in the absence of aliasing. OpenGL implementations are not required to do error checking to detect aliasing.

Active attributes that are not explicitly bound will be bound by the linker when glLinkProgram is called. The locations assigned can be queried by calling glGetAttribLocation.

OpenGL copies the `name` string when `glBindAttribLocation` is called, so an application may free its copy of the `name` string immediately after the function returns.

Generic attribute locations may be specified in the shader source text using a `location` layout qualifier. In this case, the location of the attribute specified in the shader's source takes precedence and may be queried by calling glGetAttribLocation.

Errors

`GL_INVALID_VALUE` is generated if *`index`* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`GL_INVALID_OPERATION` is generated if *`name`* starts with the reserved prefix "gl_".

`GL_INVALID_VALUE` is generated if *`program`* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`program`* is not a program object.

Associated Gets

glGet with argument `GL_MAX_VERTEX_ATTRIBS`

glGetActiveAttrib with argument *`program`*

glGetAttribLocation with arguments *`program`* and *`name`*

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBindAttribLocation	✔	✔

See Also

glDisableVertexAttribArray, glEnableVertexAttribArray, glUseProgram, glVertexAttrib, glVertexAttribPointer

Copyright

Name

glBindBuffer — bind a named buffer object

C Specification

```
void glBindBuffer (target, buffer);

GLenum target;
GLuint buffer;
```

Parameters

target	Specifies the target to which the buffer object is bound. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.
buffer	Specifies the name of a buffer object.

Description

`glBindBuffer` binds a buffer object to the specified buffer binding point. Calling `glBindBuffer` with *target* set to one of the accepted symbolic constants and *buffer* set to the name of a buffer object binds that buffer object name to the target. If no buffer object with name *buffer* exists, one is created with that name. When a buffer object is bound to a target, the previous binding for that target is automatically broken.

Buffer object names are unsigned integers. The value zero is reserved, but there is no default buffer object for each buffer object target. Instead, *buffer* set to zero effectively unbinds any buffer object previously bound, and restores client memory usage for that buffer object target (if supported for that target). Buffer object names and the corresponding buffer object contents are local to the shared object space of the current GL rendering context; two rendering contexts share buffer object names only if they explicitly enable sharing between contexts through the appropriate GL windows interfaces functions.

glGenBuffers may be used to generate a set of unused buffer object names.

The state of a buffer object immediately after it is first bound is an unmapped zero-sized memory buffer with `GL_READ_WRITE` access and `GL_STATIC_DRAW` usage.

While a non-zero buffer object name is bound, GL operations on the target to which it is bound affect the bound buffer object, and queries of the target to which it is bound return state from the bound buffer object. While buffer object name zero is bound, as in the initial state, attempts to modify or query state on the target to which it is bound generates an `GL_INVALID_OPERATION` error.

When a non-zero buffer object is bound to the `GL_ARRAY_BUFFER` target, the vertex array pointer parameter is interpreted as an offset within the buffer object measured in basic machine units.

While a non-zero buffer object is bound to the `GL_ELEMENT_ARRAY_BUFFER` target, the indices parameter of glDrawElements, glDrawElementsInstanced, glDrawRangeElements, offset within the buffer object measured in basic machine units.

While a non-zero buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target, the following commands are affected: glReadPixels. The pointer parameter is interpreted as an offset within the buffer object measured in basic machine units.

While a non-zero buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target, the following commands are affected: glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glTexImage2D, glTexImage3D, glTexSubImage2D, and glTexSubImage3D. The pointer parameter is interpreted as an offset within the buffer object measured in basic machine units.

The buffer targets `GL_COPY_READ_BUFFER` and `GL_COPY_WRITE_BUFFER` are provided to allow glCopyBufferSubData to be used without disturbing the state of other bindings. However, glCopyBufferSubData may be used with any pair of buffer binding points.

The `GL_TRANSFORM_FEEDBACK_BUFFER` buffer binding point may be passed to `glBindBuffer`, but will not directly affect transform feedback state. Instead, the indexed `GL_TRANSFORM_FEEDBACK_BUFFER` bindings must be used through a call to glBindBufferBase or glBindBufferRange. This will affect the generic `GL_TRANSFORM_FEEDBACK_BUFFER` binding.

Likewise, the `GL_UNIFORM_BUFFER` buffer binding point may be used, but does not directly affect uniform buffer state. glBindBufferBase or glBindBufferRange must be used to bind a buffer to an indexed uniform buffer binding point.

A buffer object binding created with `glBindBuffer` remains active until a different buffer object name is bound to the same target, or until the bound buffer object is deleted with glDeleteBuffers.

Once created, a named buffer object may be re-bound to any target as often as needed. However, the GL implementation may make choices about how to optimize the storage of a buffer object based on its initial binding target.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not one of the allowable values.

Associated Gets

glGet with argument `GL_ARRAY_BUFFER_BINDING`

glGet with argument `GL_COPY_READ_BUFFER_BINDING`

glGet with argument `GL_COPY_WRITE_BUFFER_BINDING`

glGet with argument `GL_ELEMENT_ARRAY_BUFFER_BINDING`

glGet with argument `GL_PIXEL_PACK_BUFFER_BINDING`

glGet with argument `GL_PIXEL_UNPACK_BUFFER_BINDING`

glGet with argument `GL_TRANSFORM_FEEDBACK_BUFFER_BINDING`

glGet with argument `GL_UNIFORM_BUFFER_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBindBuffer	✔	✔

See Also

glGenBuffers, glBindBufferBase, glBindBufferRange, glMapBufferRange, glUnmapBuffer, glDeleteBuffers, glGet, glIsBuffer

Copyright

Name

glBindBufferBase — bind a buffer object to an indexed buffer target

C Specification

```
void glBindBufferBase (target, index, buffer);

GLenum target;
GLuint index;
GLuint buffer;
```

Parameters

target	Specify the target of the bind operation. *target* must be either `GL_TRANSFORM_FEEDBACK_BUFFER` or `GL_UNIFORM_BUFFER`.
index	Specify the index of the binding point within the array specified by *target*.
buffer	The name of a buffer object to bind to the specified binding point.

Description

`glBindBufferBase` binds the buffer object *buffer* to the binding point at index *index* of the array of targets specified by *target*. Each *target* represents an indexed array of buffer binding points, as well as a single general binding point that can be used by other buffer manipulation functions such as glBindBuffer or glMapBufferRange. In addition to binding *buffer* to the indexed buffer binding target, `glBindBufferBase` also binds *buffer* to the generic buffer binding point specified by *target*.

Notes

Calling `glBindBufferBase` binds the entire buffer, even when the size of the buffer is changed after the binding is established. The starting offset is zero, and the amount of data that can be read from or written to the buffer is determined by the size of the bound buffer at the time the binding is used.

Errors

`GL_INVALID_ENUM` is generated if *target* is not `GL_TRANSFORM_FEEDBACK_BUFFER` or `GL_UNIFORM_BUFFER`.

`GL_INVALID_VALUE` is generated if *target* is `GL_TRANSFORM_FEEDBACK_BUFFER` and *index* is greater than or equal to `GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

`GL_INVALID_VALUE` is generated if *target* is `GL_UNIFORM_BUFFER` and *index* is greater than or equal to `GL_MAX_UNIFORM_BUFFER_BINDINGS`.

Associated Gets

glGet with argument `GL_MAX_UNIFORM_BUFFER_BINDINGS`, or `GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBindBufferBase	-	✔

See Also

glGenBuffers, glDeleteBuffers, glBindBuffer, glBindBufferRange, glMapBufferRange, glUnmapBuffer,

Copyright

Name

glBindBufferRange — bind a range within a buffer object to an indexed buffer target

C Specification

```
void glBindBufferRange (target, index, buffer, offset, size);

GLenumtarget;
GLuintindex;
GLuintbuffer;
GLintptroffset;
GLsizeiptrsize;
```

Parameters

target	Specify the target of the bind operation. *target* must be either `GL_TRANSFORM_FEEDBACK_BUFFER` or `GL_UNIFORM_BUFFER`.
index	Specify the index of the binding point within the array specified by *target*.
buffer	The name of a buffer object to bind to the specified binding point.
offset	The starting offset in basic machine units into the buffer object *buffer*.
size	The amount of data in machine units that can be read from the buffet object while used as an indexed target.

Description

`glBindBufferRange` binds a range of the buffer object *buffer* represented by *offset* and *size* to the binding point at index *index* of the array of targets specified by *target*. Each *target* represents an indexed array of buffer binding points, as well as a single general binding point that can be used by other buffer manipulation functions such as glBindBuffer or glMapBufferRange. In addition to binding a range of *buffer* to the indexed buffer binding target, `glBindBufferBase` also binds the range to the generic buffer binding point specified by *target*.

offset specifies the offset in basic machine units into the buffer object *buffer* and *size* specifies the amount of data that can be read from the buffer object while used as an indexed target.

Errors

`GL_INVALID_ENUM` is generated if *target* is not `GL_TRANSFORM_FEEDBACK_BUFFER` or `GL_UNIFORM_BUFFER`.

`GL_INVALID_VALUE` is generated if *target* is `GL_TRANSFORM_FEEDBACK_BUFFER` and*index* is greater than or equal to `GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

`GL_INVALID_VALUE` is generated if *target* is `GL_UNIFORM_BUFFER` and*index* is greater than or equal to `GL_MAX_UNIFORM_BUFFER_BINDINGS`.

`GL_INVALID_VALUE` is generated if *buffer* is not zero and *size* is less than or equal to zero.

`GL_INVALID_VALUE` is generated if *target* is `GL_TRANSFORM_FEEDBACK_BUFFER` and *size* or *offset* are not multiples of 4.

`GL_INVALID_VALUE` is generated if *`target`* is `GL_UNIFORM_BUFFER` and *`offset`* is not a multiple of `GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT`.

Associated Gets

glGet with argument `GL_MAX_UNIFORM_BUFFER_BINDINGS`, `GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`, or `GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBindBufferRange	-	✔

See Also

glGenBuffers, glDeleteBuffers, glBindBuffer, glBindBufferBase, glMapBufferRange, glUnmapBuffer,

Copyright

Name

glBindFramebuffer — bind a framebuffer to a framebuffer target

C Specification

```
void glBindFramebuffer (target, framebuffer);

GLenum target;
GLuint framebuffer;
```

Parameters

`target`	Specifies the framebuffer target of the binding operation.
`framebuffer`	Specifies the name of the framebuffer object to bind.

Description

`glBindFramebuffer` binds the framebuffer object with name *`framebuffer`* to the framebuffer target specified by *`target`*. *`target`* must be either `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER` or `GL_FRAMEBUFFER`. If a framebuffer object is bound to `GL_DRAW_FRAMEBUFFER` or `GL_READ_FRAMEBUFFER`, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer is bound to the corresponding bind point. Calling `glBindFramebuffer` with *`target`* set to `GL_FRAMEBUFFER` binds *`framebuffer`* to both the read and draw framebuffer targets.

glGenFramebuffers may be used to generate a set of unused framebuffer object names.

The storage, dimensions, allocation, and format of the images attached to the default framebuffer are managed entirely by the window system. In order that access to the default framebuffer is not lost, it is treated as a framebuffer object with the name of zero. The default framebuffer is therefore rendered to and read from while zero is bound to the corresponding targets.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER` or `GL_FRAMEBUFFER`.

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glBindFramebuffer	✔	✔

See Also

glGenFramebuffers, glDeleteFramebuffers, glFramebufferRenderbuffer, glFramebufferTexture2D, glFramebufferTextureLayer, glIsFramebuffer

Copyright

Name

glBindRenderbuffer — bind a renderbuffer to a renderbuffer target

C Specification

```
void glBindRenderbuffer (target, renderbuffer);

GLenum target;
GLuint renderbuffer;
```

Parameters

`target`	Specifies the renderbuffer target of the binding operation. *`target`* must be `GL_RENDERBUFFER`.
`renderbuffer`	Specifies the name of the renderbuffer object to bind.

Description

`glBindRenderbuffer` binds the renderbuffer object with name *`renderbuffer`* to the renderbuffer target specified by *`target`*. *`target`* must be `GL_RENDERBUFFER`. Calling `glBindRenderbuffer` with *`renderbuffer`* set to a value of zero breaks the existing binding of a renderbuffer object to *`target`*.

glGenRenderbuffers may be used to generate a set of unused renderbuffer object names.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_RENDERBUFFER`.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glBindRenderbuffer	✔	✔

See Also

glGenRenderbuffers, glDeleteRenderbuffers, glRenderbufferStorage, glRenderbufferStorageMultisample, glIsRenderbuffer

Copyright

Name

glBindSampler — bind a named sampler to a texturing target

C Specification

```
void glBindSampler (unit, sampler);

GLuint unit;
GLuint sampler;
```

Parameters

unit	Specifies the index of the texture unit to which the sampler is bound.
sampler	Specifies the name of a sampler.

Description

`glBindSampler` binds *sampler* to the texture unit at index *unit*. *sampler* must be zero or the name of a sampler object previously returned from a call to glGenSamplers. *unit* must be less than the value of `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

When a sampler object is bound to a texture unit, its state supersedes that of the texture object bound to that texture unit. If the sampler name zero is bound to a texture unit, the currently bound texture's sampler state becomes active. A single sampler object may be bound to multiple texture units simultaneously.

Errors

`GL_INVALID_VALUE` is generated if *unit* is greater than or equal to the value of `GL_MAX_COMBIED_TEXTURE_IMAGE_UNITS`.

`GL_INVALID_OPERATION` is generated if *sampler* is not zero or a name previously returned from a call to glGenSamplers, or if such a name has been deleted by a call to glDeleteSamplers.

Associated Gets

glGet with argument `GL_SAMPLER_BINDING`

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glBindSampler	-	✔

See Also

glGenSamplers, glDeleteSamplers, glGet, glSamplerParameter, glGetSamplerParameter, glGenTextures, glBindTexture, glDeleteTextures

Copyright

Name

glBindTexture — bind a named texture to a texturing target

C Specification

```
void glBindTexture (target, texture);

GLenum target;
GLuint texture;
```

Parameters

target	Specifies the target to which the texture is bound. Must be either `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_2D_ARRAY`, or `GL_TEXTURE_CUBE_MAP`,
texture	Specifies the name of a texture.

Description

`glBindTexture` binds the texture object with name *texture* to the texture target specified by *target*. Calling `glBindTexture` with *target* set to `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_2D_ARRAY`, or `GL_TEXTURE_CUBE_MAP` and *texture* set to the name of the new texture binds the texture name to that target. When a texture is bound to a target, the previous binding for that target is automatically broken.

Texture names are unsigned integers. The value zero is reserved to represent the default texture for each texture target. Texture names and the corresponding texture contents are local to the shared object space of the current GL rendering context; two rendering contexts share texture names only if they explicitly enable sharing between contexts through the appropriate GL windows interfaces functions.

You must use glGenTextures to generate a set of new texture names.

When a texture is first bound, it assumes the specified target: A texture first bound to `GL_TEXTURE_2D` becomes two-dimensional texture, a texture first bound to `GL_TEXTURE_3D` becomes three-dimensional texture, a texture first bound to `GL_TEXTURE_2D_ARRAY` becomes two-dimensional arary texture, and a texture first bound to `GL_TEXTURE_CUBE_MAP` becomes a cube-mapped texture. The state of a two-dimensional texture immediately after it is first bound is equivalent to the state of the default `GL_TEXTURE_2D` at GL initialization, and similarly for the other texture types.

While a texture is bound, GL operations on the target to which it is bound affect the bound texture, and queries of the target to which it is bound return state from the bound texture. In effect, the texture targets become aliases for the textures currently bound to them, and the texture name zero refers to the default textures that were bound to them at initialization.

A texture binding created with `glBindTexture` remains active until a different texture is bound to the same target, or until the bound texture is deleted with glDeleteTextures.

Once created, a named texture may be re-bound to its same original target as often as needed. It is usually much faster to use `glBindTexture` to bind an existing named texture to one of the texture targets than it is to reload the texture image using glTexImage2D, glTexImage3D or another similar function.

Texture binding is affected by the setting of the state `GL_ACTIVE_TEXTURE` (see glActiveTexture). A texture object may be bound to more than one texture unit simultaneously.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not one of the allowable values.

`GL_INVALID_OPERATION` is generated if *`texture`* was previously created with a target that doesn't match that of *`target`*.

Associated Gets

glGet with argument `GL_TEXTURE_BINDING_2D`, `GL_TEXTURE_BINDING_3D`, `GL_TEXTURE_BINDING_2D_ARRAY`, or `GL_TEXTURE_BINDING_CUBE_MAP`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBindTexture	✔	✔

See Also

glDeleteTextures, glGenTextures, glGet, glGetTexParameter, glIsTexture, glTexImage2D, glTexImage3D, glTexParameter

Copyright

Name

glBindTransformFeedback — bind a transform feedback object

C Specification

```
void glBindTransformFeedback (target, id);

GLenum target;
GLuint id;
```

Parameters

`target`	Specifies the target to which to bind the transform feedback object *`id`*. *`target`* must be `GL_TRANSFORM_FEEDBACK`.
`id`	Specifies the name of a transform feedback object reserved by glGenTransformFeedbacks.

Description

`glBindTransformFeedback` binds the transform feedback object with name *`id`* to the current GL state. *`id`* must be a name previously returned from a call to glGenTransformFeedbacks. If *`id`* has not previously been bound, a new transform feedback object with name *`id`* and initialized with the default transform state vector is created.

In the initial state, a default transform feedback object is bound and treated as a transform feedback object with a name of zero. If the name zero is subsequently bound, the default transform feedback object is again bound to the GL state.

While a transform feedback buffer object is bound, GL operations on the target to which it is bound affect the bound transform feedback object, and queries of the target to which a transform feedback object is bound return state from the bound object. When buffer objects are bound for transform feedback, they are attached to the currently bound transform feedback object. Buffer objects are used for transform feedback only if they are attached to the currently bound transform feedback object.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_TRANSFORM_FEEDBACK`.

`GL_INVALID_OPERATION` is generated if the transform feedback operation is active on the currently bound transform feedback object, and that operation is not paused.

`GL_INVALID_OPERATION` is generated if *`id`* is not zero or the name of a transform feedback object returned from a previous call to glGenTransformFeedbacks, or if such a name has been deleted by glDeleteTransformFeedbacks.

Associated Gets

glGet with argument `GL_TRANSFORM_FEEDBACK_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBindTransformFeedback	-	✔

See Also

glGenTransformFeedbacks, glDeleteTransformFeedbacks, glIsTransformFeedback, glBeginTransformFeedback, glPauseTransformFeedback, glResumeTransformFeedback, glEndTransformFeedback

Copyright

Name

glBindVertexArray — bind a vertex array object

C Specification

```
void glBindVertexArray (array);
```

```
GLuint array;
```

Parameters

array Specifies the name of the vertex array to bind.

Description

`glBindVertexArray` binds the vertex array object with name *array*. *array* is the name of a vertex array object previously returned from a call to glGenVertexArrays, or zero to bind the default vertex array object binding.

If no vertex array object with name *array* exists, one is created when *array* is first bound. If the bind is successful no change is made to the state of the vertex array object, and any previous vertex array object binding is broken.

Errors

`GL_INVALID_OPERATION` is generated if *array* is not zero or the name of a vertex array object previously returned from a call to glGenVertexArrays.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBindVertexArray	-	✔

See Also

glGenVertexArrays, glDeleteVertexArrays glVertexAttribPointer glEnableVertexAttribArray

Copyright

Name

glBlendColor — set the blend color

C Specification

```
void glBlendColor (red, green, blue, alpha);

GLfloat red;
GLfloat green;
GLfloat blue;
GLfloat alpha;
```

Parameters

red, *green*, *blue*, *alpha*	specify the components of `GL_BLEND_COLOR`

Description

The `GL_BLEND_COLOR` may be used to calculate the source and destination blending factors. If destination framebuffer components use an unsigned normalized fixed-point representation, the constant color components are clamped to the range when computing the blend factors. See glBlendFunc for a complete description of the blending operations. Initially the `GL_BLEND_COLOR` is set to (0, 0, 0, 0).

Associated Gets

glGet with an argument of `GL_BLEND_COLOR`

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glBlendColor	✔	✔

See Also

glBlendEquation, glBlendFunc, glGet

Copyright

Name

glBlendEquation — specify the equation used for both the RGB blend equation and the Alpha blend equation

C Specification

```
void glBlendEquation (mode);

GLenum mode;
```

Parameters

mode specifies how source and destination colors are combined. It must be `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN`, `GL_MAX`.

Description

The blend equations determine how a new pixel (the "source" color) is combined with a pixel already in the framebuffer (the "destination" color). This function sets both the RGB blend equation and the alpha blend equation to a single equation.

Calling this function is equivalent to calling glBlendEquationSeparate with *modeRGB* and *modeAlpha* both set to the value of *mode*.

These equations use the source and destination blend factors specified by either glBlendFunc or glBlendFuncSeparate. See glBlendFunc or glBlendFuncSeparate for a description of the various blend factors.

In the equations that follow, source and destination color components are referred to as and , respectively. The result color is referred to as . The source and destination blend factors are denoted and , respectively. For these equations all color components are understood to have values in the range .

Mode	**RGB Components**	**Alpha Component**
`GL_FUNC_ADD`		
`GL_FUNC_SUBTRACT`		
`GL_FUNC_REVERSE_SUBTRACT`		
`GL_MIN`		
`GL_MAX`		

The results of these equations are clamped to the range .

The `GL_MIN` and `GL_MAX` equations are useful for applications that analyze image data (image thresholding against a constant color, for example). The `GL_FUNC_ADD` equation is useful for antialiasing and transparency, among other things.

Initially, both the RGB blend equation and the alpha blend equation are set to `GL_FUNC_ADD`.

Notes

The `GL_MIN`, and `GL_MAX` equations do not use the source or destination factors, only the source and destination colors.

Errors

`GL_INVALID_ENUM` is generated if *mode* is not one of `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MAX`, or `GL_MIN`.

Associated Gets

glGet with an argument of `GL_BLEND_EQUATION_RGB`

glGet with an argument of `GL_BLEND_EQUATION_ALPHA`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBlendEquation	✔	✔

See Also

glBlendColor, glBlendEquationSeparate glBlendFunc glBlendFuncSeparate

Copyright

Name

glBlendEquationSeparate — set the RGB blend equation and the alpha blend equation separately

C Specification

```
void glBlendEquationSeparate (modeRGB, modeAlpha);

GLenum modeRGB;
GLenum modeAlpha;
```

Parameters

`modeRGB`	specifies the RGB blend equation, how the red, green, and blue components of the source and destination colors are combined. It must be `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN`, `GL_MAX`.
`modeAlpha`	specifies the alpha blend equation, how the alpha component of the source and destination colors are combined. It must be `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN`, `GL_MAX`.

Description

The blend equations determines how a new pixel (the "source" color) is combined with a pixel already in the framebuffer (the "destination" color). This function specifies one blend equation for the RGB-color components and one blend equation for the alpha component.

The blend equations use the source and destination blend factors specified by either glBlendFunc or glBlendFuncSeparate. See glBlendFunc or glBlendFuncSeparate for a description of the various blend factors.

In the equations that follow, source and destination color components are referred to as and , respectively. The result color is referred to as . The source and destination blend factors are denoted and , respectively. For these equations all color components are understood to have values in the range .

Mode	**RGB Components**	**Alpha Component**
`GL_FUNC_ADD`		
`GL_FUNC_SUBTRACT`		
`GL_FUNC_REVERSE_SUBTRACT`		
`GL_MIN`		
`GL_MAX`		

The results of these equations are clamped to the range .

The `GL_MIN` and `GL_MAX` equations are useful for applications that analyze image data (image thresholding against a constant color, for example). The `GL_FUNC_ADD` equation is useful for antialiasing and transparency, among other things.

Initially, both the RGB blend equation and the alpha blend equation are set to `GL_FUNC_ADD`.

Notes

The `GL_MIN`, and `GL_MAX` equations do not use the source or destination factors, only the source and destination colors.

Errors

`GL_INVALID_ENUM` is generated if either *`modeRGB`* or *`modeAlpha`* is not one of `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MAX`, or `GL_MIN`.

Associated Gets

glGet with an argument of `GL_BLEND_EQUATION_RGB`

glGet with an argument of `GL_BLEND_EQUATION_ALPHA`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBlendEquationSeparate	✔	✔

See Also

glGetString, glBlendColor, glBlendEquation, glBlendFunc, glBlendFuncSeparate

Copyright

Name

glBlendFunc — specify pixel arithmetic

C Specification

```
void glBlendFunc (sfactor, dfactor);

GLenum sfactor;
GLenum dfactor;
```

Parameters

sfactor	Specifies how the red, green, blue, and alpha source blending factors are computed. The initial value is `GL_ONE`.
dfactor	Specifies how the red, green, blue, and alpha destination blending factors are computed. The following symbolic constants are accepted: `GL_ZERO`, `GL_ONE`, `GL_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_DST_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`. `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, and `GL_ONE_MINUS_CONSTANT_ALPHA`. The initial value is `GL_ZERO`.

Description

Pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). Blending is initially disabled. Use glEnable and glDisable with argument `GL_BLEND` to enable and disable blending.

`glBlendFunc` defines the operation of blending when it is enabled. *sfactor* specifies which method is used to scale the source color components. *dfactor* specifies which method is used to scale the destination color components. Both parameters must be one of the following symbolic constants: `GL_ZERO`, `GL_ONE`, `GL_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_DST_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`, `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, `GL_ONE_MINUS_CONSTANT_ALPHA`, `GL_SRC_ALPHA_SATURATE`, The possible methods are described in the following table. Each method defines four scale factors, one each for red, green, blue, and alpha. In the table and in subsequent equations, source and destination color components are referred to as , and , respectively. The color specified by glBlendColor is referred to as .

Source and destination scale factors are referred to as and . The scale factors described in the table, denoted , represent either source or destination factors. All scale factors have range .

Prior to blending, unsigned normalized fixed-point color components undergo an implied conversion to floating-point using equation 2.1. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating-point and will be performed with a precision and dynamic range no lower than that used to represent destination components. If the value of `GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `GL_SRGB`, the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components.

Parameter	
GL_ZERO	
GL_ONE	
GL_SRC_COLOR	
GL_ONE_MINUS_SRC_COLOR	
GL_DST_COLOR	
GL_ONE_MINUS_DST_COLOR	
GL_SRC_ALPHA	
GL_ONE_MINUS_SRC_ALPHA	
GL_DST_ALPHA	
GL_ONE_MINUS_DST_ALPHA	
GL_CONSTANT_COLOR	
GL_ONE_MINUS_CONSTANT_COLOR	
GL_CONSTANT_ALPHA	
GL_ONE_MINUS_CONSTANT_ALPHA	
GL_SRC_ALPHA_SATURATE	

In the table,

To determine the blended RGBA values of a pixel, the system uses the following equations:

If the value of `GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `GL_SRGB`, the R, G, and B values after blending are converted into the non-linear sRGB color space by computing where cl is the R, G, or B element and cs is the result (effectively converted into an sRGB color space). If `GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` is not `GL_SRGB`, then cs = cl: The resulting cs values for R, G, and B, and the unmodified A form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range [0; 1] and then converted to a fixed-point value using equation

Examples

Transparency is best implemented using blend function (`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) with primitives sorted from farthest to nearest. Note that this transparency calculation does not require the presence of alpha bitplanes in the frame buffer.

Blend function (`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) is also useful for rendering antialiased points and lines in arbitrary order.

Notes

Incoming (source) alpha is correctly thought of as a material opacity, ranging from 1.0 (), representing complete opacity, to 0.0 (0), representing complete transparency.

When more than one color buffer is enabled for drawing, the GL performs blending separately for each enabled buffer, using the contents of that buffer for destination color. (See glDrawBuffers.)

Errors

`GL_INVALID_ENUM` is generated if either *`sfactor`* or *`dfactor`* is not an accepted value.

Associated Gets

glGet with argument `GL_BLEND_SRC`

glGet with argument `GL_BLEND_DST`

glIsEnabled with argument `GL_BLEND`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBlendFunc	✔	✔

See Also

glBlendColor, glBlendEquation, glBlendFuncSeparate, glClear, glDrawBuffers, glEnable,

Copyright

Name

glBlendFuncSeparate — specify pixel arithmetic for RGB and alpha components separately

C Specification

```
void glBlendFuncSeparate (srcRGB, dstRGB, srcAlpha, dstAlpha);

GLenum srcRGB;
GLenum dstRGB;
GLenum srcAlpha;
GLenum dstAlpha;
```

Parameters

srcRGB	Specifies how the red, green, and blue blending factors are computed. The initial value is `GL_ONE`.
dstRGB	Specifies how the red, green, and blue destination blending factors are computed. The initial value is `GL_ZERO`.
srcAlpha	Specified how the alpha source blending factor is computed. The initial value is `GL_ONE`.
dstAlpha	Specified how the alpha destination blending factor is computed. The initial value is `GL_ZERO`.

Description

Pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). Blending is initially disabled. Use glEnable and glDisable with argument `GL_BLEND` to enable and disable blending.

`glBlendFuncSeparate` defines the operation of blending when it is enabled. *srcRGB* specifies which method is used to scale the source RGB-color components. *dstRGB* specifies which method is used to scale the destination RGB-color components. Likewise, *srcAlpha* specifies which method is used to scale the source alpha color component, and *dstAlpha* specifies which method is used to scale the destination alpha component. The possible methods are described in the following table. Each method defines four scale factors, one each for red, green, blue, and alpha.

In the table and in subsequent equations, source and destination color components are referred to as , and , respectively. The color specified by glBlendColor is referred to as .

Source and destination scale factors are referred to as and . All scale factors have range .

Prior to blending, unsigned normalized fixed-point color components undergo an implied conversion to floating-point using equation 2.1. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating-point and will be performed with a precision and dynamic range no lower than that used to represent destination components. If the value of `GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `GL_SRGB`, the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components.

Parameter	RGB Factor	Alpha Factor
GL_ZERO		
GL_ONE		
GL_SRC_COLOR		
GL_ONE_MINUS_SRC_COLOR		
GL_DST_COLOR		
GL_ONE_MINUS_DST_COLOR		
GL_SRC_ALPHA		
GL_ONE_MINUS_SRC_ALPHA		
GL_DST_ALPHA		
GL_ONE_MINUS_DST_ALPHA		
GL_CONSTANT_COLOR		
GL_ONE_MINUS_CONSTANT_COLOR		
GL_CONSTANT_ALPHA		
GL_ONE_MINUS_CONSTANT_ALPHA		
GL_SRC_ALPHA_SATURATE		

In the table,

To determine the blended RGBA values of a pixel, the system uses the following equations:

If the value of `GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `GL_SRGB`, the R, G, and B values after blending are converted into the non-linear sRGB color space by computing where cl is the R, G, or B element and cs is the result (effectively converted into an sRGB color space). If `GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` is not `GL_SRGB`, then cs = cl: The resulting cs values for R, G, and B, and the unmodified A form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range [0; 1] and then converted to a fixed-point value using equation

Notes

Incoming (source) alpha is correctly thought of as a material opacity, ranging from 1.0 (), representing complete opacity, to 0.0 (0), representing complete transparency.

When more than one color buffer is enabled for drawing, the GL performs blending separately for each enabled buffer, using the contents of that buffer for destination color. (See glDrawBuffers.)

Errors

`GL_INVALID_ENUM` is generated if either *`srcRGB`* or *`dstRGB`* is not an accepted value.

Associated Gets

glGet with argument `GL_BLEND_SRC_RGB`

glGet with argument `GL_BLEND_SRC_ALPHA`

glGet with argument `GL_BLEND_DST_RGB`

glGet with argument `GL_BLEND_DST_ALPHA`

glIsEnabled with argument `GL_BLEND`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBlendFuncSeparate	✔	✔

See Also

glBlendColor, glBlendFunc, glBlendEquation, glBlendEquationSeparate, glClear, glDrawBuffers, glEnable,

Copyright

Name

glBlitFramebuffer — copy a block of pixels from the read framebuffer to the draw framebuffer

C Specification

```
void glBlitFramebuffer (srcX0, srcY0, srcX1, srcY1, dstX0, dstY0, dstX1,
dstY1, mask, filter);

GLint srcX0;
GLint srcY0;
GLint srcX1;
GLint srcY1;
GLint dstX0;
GLint dstY0;
GLint dstX1;
GLint dstY1;
GLbitfield mask;
GLenum filter;
```

Parameters

srcX0, *srcY0*, *srcX1*, *srcY1*	Specify the bounds of the source rectangle within the read buffer of the read framebuffer.
dstX0, *dstY0*, *dstX1*, *dstY1*	Specify the bounds of the destination rectangle within the write buffer of the write framebuffer.
mask	The bitwise OR of the flags indicating which buffers are to be copied. The allowed flags are `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT` and `GL_STENCIL_BUFFER_BIT`.
filter	Specifies the interpolation to be applied if the image is stretched. Must be `GL_NEAREST` or `GL_LINEAR`.

Description

`glBlitFramebuffer` transfers a rectangle of pixel values from one region of the read framebuffer to another region in the draw framebuffer. *mask* is the bitwise OR of a number of values indicating which buffers are to be copied. The values are `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT`. The pixels corresponding to these buffers are copied from the source rectangle bounded by the locations (*srcX0*; *srcY0*) and (*srcX1*; *srcY1*) to the destination rectangle bounded by the locations (*dstX0*; *dstY0*) and (*dstX1*; *dstY1*). The lower bounds of the rectangle are inclusive, while the upper bounds are exclusive.

The actual region taken from the read framebuffer is limited to the intersection of the source buffers being transferred, which may include the color buffer selected by the read buffer, the depth buffer, and/or the stencil buffer depending on mask. The actual region written to the draw framebuffer is limited to the intersection of the destination buffers being written, which may include multiple draw buffers, the depth buffer, and/or the stencil buffer depending on mask. Whether or not the source or destination regions are altered due to these limits, the scaling and offset applied to pixels being transferred is performed as though no such limits were present.

If the sizes of the source and destination rectangles are not equal, *`filter`* specifies the interpolation method that will be applied to resize the source image , and must be `GL_NEAREST` or `GL_LINEAR`. `GL_LINEAR` is only a valid interpolation method for the color buffer. If *`filter`* is not `GL_NEAREST` and *`mask`* includes `GL_DEPTH_BUFFER_BIT` or `GL_STENCIL_BUFFER_BIT`, no data is transferred and a `GL_INVALID_OPERATION` error is generated.

If *`filter`* is `GL_LINEAR` and the source rectangle would require sampling outside the bounds of the source framebuffer, values are read as if the `GL_CLAMP_TO_EDGE` texture wrapping mode were applied.

When the color buffer is transferred, values are taken from the read buffer of the read framebuffer and written to each of the draw buffers of the draw framebuffer.

If the source and destination rectangles overlap or are the same, and the read and draw buffers are the same, the result of the operation is undefined.

If *`SAMPLE_BUFFERS`* for the read framebuffer is greater than zero and *`SAMPLE_BUFFERS`* for the draw framebuffer is zero, the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination.

Errors

`GL_INVALID_OPERATION` is generated if *`mask`* contains any of the `GL_DEPTH_BUFFER_BIT` or `GL_STENCIL_BUFFER_BIT` and *`filter`* is not `GL_NEAREST`.

`GL_INVALID_OPERATION` is generated if *`mask`* contains `GL_COLOR_BUFFER_BIT` and any of the following conditions hold:

- The read buffer contains fixed-point or floating-point values and any draw buffer contains neither fixed-point nor floating-point values.
- The read buffer contains unsigned integer values and any draw buffer does not contain unsigned integer values.
- The read buffer contains signed integer values and any draw buffer does not contain signed integer values.

`GL_INVALID_OPERATION` is generated if *`mask`* contains `GL_DEPTH_BUFFER_BIT` or `GL_STENCIL_BUFFER_BIT` and the source and destination depth and stencil formats do not match.

`GL_INVALID_OPERATION` is generated if *`filter`* is `GL_LINEAR` and the read buffer contains integer data.

`GL_INVALID_OPERATION` is generated if the value of *`GL_SAMPLE_BUFFERS`* for the draw buffer is greater than zero.

`GL_INVALID_OPERATION` is generated if *`GL_SAMPLE_BUFFERS`* for the read buffer is greater than zero and the formats of draw and read buffers are not identical, or the source and destination rectangles are not defined with the same (X0, Y0) and (X1, Y1) bounds.

`GL_INVALID_FRAMEBUFFER_OPERATION` is generated if the objects bound to `GL_DRAW_FRAMEBUFFER_BINDING` or `GL_READ_FRAMEBUFFER_BINDING` are not framebuffer complete.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBlitFramebuffer	-	✔

See Also

glReadPixels glCheckFramebufferStatus, glGenFramebuffers glBindFramebuffer glDeleteFramebuffers

Copyright

Name

glBufferData — creates and initializes a buffer object's data store

C Specification

```
void glBufferData (target, size, data, usage);

GLenum target;
GLsizeiptr size;
const void * data;
GLenum usage;
```

Parameters

target	Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.
size	Specifies the size in bytes of the buffer object's new data store.
data	Specifies a pointer to data that will be copied into the data store for initialization, or `NULL` if no data is to be copied.
usage	Specifies the expected usage pattern of the data store. The symbolic constant must be `GL_STREAM_DRAW`, `GL_STREAM_READ`, `GL_STREAM_COPY`, `GL_STATIC_DRAW`, `GL_STATIC_READ`, `GL_STATIC_COPY`, `GL_DYNAMIC_DRAW`, `GL_DYNAMIC_READ`, or `GL_DYNAMIC_COPY`.

Description

`glBufferData` creates a new data store for the buffer object currently bound to *target*. Any pre-existing data store is deleted. The new data store is created with the specified *size* in bytes and *usage*. If *data* is not `NULL`, the data store is initialized with data from this pointer. In its initial state, the new data store is not mapped, it has a `NULL` mapped pointer, and its mapped access is `GL_READ_WRITE`.

usage is a hint to the GL implementation as to how a buffer object's data store will be accessed. This enables the GL implementation to make more intelligent decisions that may significantly impact buffer object performance. It does not, however, constrain the actual usage of the data store. *usage* can be broken down into two parts: first, the frequency of access (modification and usage), and second, the nature of that access. The frequency of access may be one of these:

STREAM	The data store contents will be modified once and used at most a few times.
STATIC	The data store contents will be modified once and used many times.
DYNAMIC	The data store contents will be modified repeatedly and used many times.

The nature of access may be one of these:

DRAW	The data store contents are modified by the application, and used as the source for GL drawing and image specification commands.

READ The data store contents are modified by reading data from the GL, and used to return that data when queried by the application.

COPY The data store contents are modified by reading data from the GL, and used as the source for GL drawing and image specification commands.

Notes

If *data* is NULL, a data store of the specified size is still created, but its contents remain uninitialized and thus undefined.

Clients must align data elements consistently with the requirements of the client platform, with an additional base-level requirement that an offset within a buffer to a datum comprising bytes be a multiple of .

Errors

GL_INVALID_ENUM is generated if *target* is not one of the accepted buffer targets.

GL_INVALID_ENUM is generated if *usage* is not GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, or GL_DYNAMIC_COPY.

GL_INVALID_VALUE is generated if *size* is negative.

GL_INVALID_OPERATION is generated if the reserved buffer object name 0 is bound to *target*.

GL_OUT_OF_MEMORY is generated if the GL is unable to create a data store with the specified *size*.

Associated Gets

glGetBufferParameter with argument GL_BUFFER_SIZE or GL_BUFFER_USAGE

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBufferData	✔	✔

See Also

glBindBuffer, glBufferSubData, glMapBufferRange, glUnmapBuffer

Copyright

Name

glBufferSubData — updates a subset of a buffer object's data store

C Specification

```
void glBufferSubData (target, offset, size, data);

GLenum target;
GLintptr offset;
GLsizeiptr size;
const void * data;
```

Parameters

target	Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.
offset	Specifies the offset into the buffer object's data store where data replacement will begin, measured in bytes.
size	Specifies the size in bytes of the data store region being replaced.
data	Specifies a pointer to the new data that will be copied into the data store.

Description

`glBufferSubData` redefines some or all of the data store for the buffer object currently bound to *target*. Data starting at byte offset *offset* and extending for *size* bytes is copied to the data store from the memory pointed to by *data*. An error is thrown if *offset* and *size* together define a range beyond the bounds of the buffer object's data store.

Notes

When replacing the entire data store, consider using `glBufferSubData` rather than completely recreating the data store with `glBufferData`. This avoids the cost of reallocating the data store.

Consider using multiple buffer objects to avoid stalling the rendering pipeline during data store updates. If any rendering in the pipeline makes reference to data in the buffer object being updated by `glBufferSubData`, especially from the specific region being updated, that rendering must drain from the pipeline before the data store can be updated.

Clients must align data elements consistently with the requirements of the client platform, with an additional base-level requirement that an offset within a buffer to a datum comprising bytes be a multiple of .

Errors

`GL_INVALID_ENUM` is generated if *target* is not one of the accepted buffer targets.

`GL_INVALID_VALUE` is generated if *offset* or *size* is negative, or if together they define a region of memory that extends beyond the buffer object's allocated data store.

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *`target`*.

`GL_INVALID_OPERATION` is generated if the buffer object being updated is mapped.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glBufferSubData	✔	✔

See Also

glBindBuffer, glBufferData, glMapBufferRange, glUnmapBuffer

Copyright

Name

glCheckFramebufferStatus — check the completeness status of a framebuffer

C Specification

```
GLenum glCheckFramebufferStatus (target);
```

```
GLenum target;
```

Parameters

target Specify the target of the framebuffer completeness check.

Description

`glCheckFramebufferStatus` queries the completeness status of the framebuffer object currently bound to *target*. *target* must be `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER` or `GL_FRAMEBUFFER`. `GL_FRAMEBUFFER` is equivalent to `GL_DRAW_FRAMEBUFFER`.

The return value is `GL_FRAMEBUFFER_COMPLETE` if the framebuffer bound to *target* is complete. Otherwise, the return value is determined as follows:

- `GL_FRAMEBUFFER_UNDEFINED` is returned if *target* is the default framebuffer, but the default framebuffer does not exist.
- `GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT` is returned if any of the framebuffer attachment points are framebuffer incomplete.
- `GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT` is returned if the framebuffer does not have at least one image attached to it.
- `GL_FRAMEBUFFER_UNSUPPORTED` is returned if depth and stencil attachments, if present, are not the same renderbuffer, or if the combination of internal formats of the attached images violates an implementation-dependent set of restrictions.
- `GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE` is returned if the value of `GL_RENDERBUFFER_SAMPLES` is not the same for all attached renderbuffers or, if the attached images are a mix of renderbuffers and textures, the value of `GL_RENDERBUFFER_SAMPLES` is not zero.

Additionally, if an error occurs, zero is returned.

Errors

`GL_INVALID_ENUM` is generated if *target* is not `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER` or `GL_FRAMEBUFFER`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glCheckFramebufferStatus	✔	✔

See Also

glGenFramebuffers, glDeleteFramebuffers glBindFramebuffer

Copyright

Name

glClear — clear buffers to preset values

C Specification

`void glClear (mask);`

`GLbitfield mask;`

Parameters

mask	Bitwise OR of masks that indicate the buffers to be cleared. The three masks are `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT`.

Description

`glClear` sets the bitplane area of the window to values previously selected by `glClearColor`, `glClearDepthf`, and `glClearStencil`. Multiple color buffers can be cleared simultaneously by selecting more than one buffer at a time using glDrawBuffers.

The pixel ownership test, the scissor test, sRGB conversion, dithering, and the buffer writemasks affect the operation of `glClear`. The scissor box bounds the cleared region. Alpha function, blend function, stenciling, texture mapping, and depth-buffering are ignored by `glClear`.

`glClear` takes a single argument that is the bitwise OR of several values indicating which buffer is to be cleared.

The values are as follows:

`GL_COLOR_BUFFER_BIT`	Indicates the buffers currently enabled for color writing.
`GL_DEPTH_BUFFER_BIT`	Indicates the depth buffer.
`GL_STENCIL_BUFFER_BIT`	Indicates the stencil buffer.

The value to which each buffer is cleared depends on the setting of the clear value for that buffer.

Notes

If a buffer is not present, then a `glClear` directed at that buffer has no effect.

Errors

`GL_INVALID_VALUE` is generated if any bit other than the three defined bits is set in *`mask`*.

Associated Gets

glGet with argument `GL_DEPTH_CLEAR_VALUE`

glGet with argument `GL_COLOR_CLEAR_VALUE`

glGet with argument `GL_STENCIL_CLEAR_VALUE`

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glClear	✔	✔

See Also

glClearBuffer, glClearColor, glClearDepthf, glClearStencil, glColorMask, glDepthMask, glDrawBuffers, glScissor, glStencilMask

Copyright

Name

glClearBuffer — clear individual buffers of the currently bound draw framebuffer

C Specification

```
void glClearBufferiv (buffer, drawBuffer, value);

GLenum buffer;
GLint drawBuffer;
const GLint * value;

void glClearBufferuiv (buffer, drawBuffer, value);

GLenum buffer;
GLint drawBuffer;
const GLuint * value;

void glClearBufferfv (buffer, drawBuffer, value);

GLenum buffer;
GLint drawBuffer;
const GLfloat * value;

void glClearBufferfi (buffer, drawBuffer, depth, stencil);

GLenum buffer;
GLint drawBuffer;
GLfloat depth;
GLint stencil;
```

Parameters

buffer	Specify the buffer to clear.
drawBuffer	Specify a particular draw buffer to clear.
value	For color buffers, a pointer to a four-element vector specifying R, G, B and A values to clear the buffer to. For depth buffers, a pointer to a single depth value to clear the buffer to. For stencil buffers, a pointer to a single stencil value to clear the buffer to.
depth	The value to clear a depth render buffer to.
stencil	The value to clear a stencil render buffer to.

Description

`glClearBuffer*` clears the specified buffer to the specified value(s). If *buffer* is `GL_COLOR`, a particular draw buffer `GL_DRAWBUFFER`*i* is specified by passing *i* as *drawBuffer*. In this case, *value* points to a four-element vector specifying the R, G, B and A color to clear that draw buffer to. The `glClearBufferfv`, `glClearBufferiv`, and `glClearBufferuiv` commands should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively. Clamping and conversion for fixed-point color buffers are performed in the same fashion as glClearColor.

If *buffer* is `GL_DEPTH`, *drawBuffer* must be zero, and *value* points to a single value to clear the depth buffer to. Only `glClearBufferfv` should be used to clear depth buffers. Clamping and conversion for fixed-point depth buffers are performed in the same fashion as glClearDepthf.

If *buffer* is GL_STENCIL, *drawBuffer* must be zero, and *value* points to a single value to clear the stencil buffer to. Only glClearBufferiv should be used to clear stencil buffers. Masking and type conversion are performed in the same fashion as glClearStencil.

glClearBufferfi may be used to clear the depth and stencil buffers. *buffer* must be GL_DEPTH_STENCIL and *drawBuffer* must be zero. *depth* and *stencil* are the depth and stencil values, respectively.

The result of glClearBuffer is undefined if no conversion between the type of *value* and the buffer being cleared is defined. However, this is not an error.

Errors

GL_INVALID_ENUM is generated by glClearBufferiv if *buffer* is not GL_COLOR or GL_STENCIL.

GL_INVALID_ENUM is generated by glClearBufferfv if *buffer* is not GL_COLOR or GL_DEPTH.

GL_INVALID_ENUM is generated by glClearBufferuiv if *buffer* is not GL_COLOR.

GL_INVALID_ENUM is generated by glClearBufferfi if *buffer* is not GL_DEPTH_STENCIL.

GL_INVALID_VALUE is generated if *buffer* is GL_COLOR and *drawBuffer* is greater than or equal to GL_MAX_DRAW_BUFFERS.

GL_INVALID_VALUE is generated if *buffer* is GL_DEPTH, GL_STENCIL or GL_DEPTH_STENCIL and *drawBuffer* is not zero.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glClearBufferiv	-	✔
glClearBufferuiv	-	✔
glClearBufferfv	-	✔
glClearBufferfi	-	✔

See Also

glClearColor, glClearDepthf, glClearStencil, glClear

Copyright

Name

glClearColor — specify clear values for the color buffers

C Specification

```
void glClearColor (red, green, blue, alpha);

GLfloat red;
GLfloat green;
GLfloat blue;
GLfloat alpha;
```

Parameters

red, *green*, *blue*, *alpha*	Specify the red, green, blue, and alpha values used when the color buffers are cleared. The initial values are all 0.

Description

`glClearColor` specifies the red, green, blue, and alpha values used by glClear to clear fixed- and floating-point color buffers. Unsigned normalized fixed point RGBA color buffers are cleared to color values derived by clamping each component of the clear color to the range , then converting the (possibly sRGB converted and/or dithered) color to fixed-point.

Associated Gets

glGet with argument `GL_COLOR_CLEAR_VALUE`

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glClearColor	✔	✔

See Also

glClear

Copyright

Name

glClearDepthf — specify the clear value for the depth buffer

C Specification

```
void glClearDepthf (depth);
```

```
GLfloat depth;
```

Parameters

depth Specifies the depth value used when the depth buffer is cleared. The initial value is 1.

Description

`glClearDepthf` specifies the depth value used by glClear to clear the depth buffer. When clearing a fixed-point depth buffer, values specified by `glClearDepthf` are clamped to the range , and converted to fixed-point. No clamping or conversion is applied when clearing a floating-point depth buffer.

Associated Gets

glGet with argument `GL_DEPTH_CLEAR_VALUE`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glClearDepthf	✔	✔

See Also

glClear

Copyright

Name

glClearStencil — specify the clear value for the stencil buffer

C Specification

```
void glClearStencil (s);
```

```
GLint s;
```

Parameters

s Specifies the index used when the stencil buffer is cleared. The initial value is 0.

Description

`glClearStencil` specifies the index used by glClear to clear the stencil buffer. When clearing a stencil buffer, *s* is masked with , where is the number of bits in the stencil buffer.

Associated Gets

glGet with argument `GL_STENCIL_CLEAR_VALUE`

glGet with argument `GL_STENCIL_BITS`

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glClearStencil	✔	✔

See Also

glClear, glStencilFunc, glStencilFuncSeparate, glStencilMask, glStencilMaskSeparate, glStencilOp, glStencilOpSeparate

Copyright

Name

glClientWaitSync — block and wait for a sync object to become signaled

C Specification

```
GLenum glClientWaitSync (sync, flags, timeout);

GLsync sync;
GLbitfield flags;
GLuint64 timeout;
```

Parameters

sync	The sync object whose status to wait on.
flags	A bitfield controlling the command flushing behavior. *flags* may be `GL_SYNC_FLUSH_COMMANDS_BIT`.
timeout	The timeout, specified in nanoseconds, for which the implementation should wait for *sync* to become signaled.

Description

`glClientWaitSync` causes the client to block and wait for the sync object specified by *sync* to become signaled. If *sync* is signaled when `glClientWaitSync` is called, `glClientWaitSync` returns immediately, otherwise it will block and wait for up to *timeout* nanoseconds for *sync* to become signaled.

The return value is one of four status values:

- `GL_ALREADY_SIGNALED` indicates that *sync* was signaled at the time that `glClientWaitSync` was called.
- `GL_TIMEOUT_EXPIRED` indicates that at least *timeout* nanoseconds passed and *sync* did not become signaled.
- `GL_CONDITION_SATISFIED` indicates that *sync* was signaled before the timeout expired.
- `GL_WAIT_FAILED` indicates that an error occurred. Additionally, an OpenGL error will be generated.

Errors

`GL_INVALID_VALUE` is generated if *sync* is not the name of an existing sync object.

`GL_INVALID_VALUE` is generated if *flags* contains any unsupported flag.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glClientWaitSync	-	✔

See Also

glFenceSync, glIsSync glWaitSync

Copyright

Name

glColorMask — enable and disable writing of frame buffer color components

C Specification

```
void glColorMask (red, green, blue, alpha);

GLboolean red;
GLboolean green;
GLboolean blue;
GLboolean alpha;
```

Parameters

red, *green*, *blue*, *alpha*	Specify whether red, green, blue, and alpha are to be written into the frame buffer. The initial values are all `GL_TRUE`, indicating that the color components are written.

Description

`glColorMask` specifies whether the individual color components in the frame buffer can or cannot be written. `glColorMask` sets the mask for all active draw buffers. If *red* is `GL_FALSE`, for example, no change is made to the red component of any pixel in any of the color buffers, regardless of the drawing operation attempted.

Changes to individual bits of components cannot be controlled. Rather, changes are either enabled or disabled for entire color components.

Associated Gets

glGet with argument `GL_COLOR_WRITEMASK`

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glColorMask	✔	✔

See Also

glClear, glDepthMask, glStencilMask

Copyright

Name

glCompileShader — Compiles a shader object

C Specification

```
void glCompileShader (shader);
```

```
GLuint shader;
```

Parameters

`shader` Specifies the shader object to be compiled.

Description

`glCompileShader` compiles the source code strings that have been stored in the shader object specified by *`shader`*.

The compilation status will be stored as part of the shader object's state. This value will be set to `GL_TRUE` if the shader was compiled without errors and is ready for use, and `GL_FALSE` otherwise. It can be queried by calling glGetShaderiv with arguments *`shader`* and `GL_COMPILE_STATUS`.

Compilation of a shader can fail for a number of reasons as specified by the OpenGL ES Shading Language Specification. Whether or not the compilation was successful, information about the compilation can be obtained from the shader object's information log by calling glGetShaderInfoLog.

Errors

`GL_INVALID_VALUE` is generated if *`shader`* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`shader`* is not a shader object.

Associated Gets

glGetShaderInfoLog with argument *`shader`*

glGetShaderiv with arguments *`shader`* and `GL_COMPILE_STATUS`

glIsShader

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glCompileShader	✔	✔

See Also

glCreateShader, glLinkProgram, glShaderSource

Copyright

Name

glCompressedTexImage2D — specify a two-dimensional texture image in a compressed format

C Specification

```
void glCompressedTexImage2D (target, level, internalformat, width,
height, border, imageSize, data);

GLenum target;
GLint level;
GLenum internalformat;
GLsizei width;
GLsizei height;
GLint border;
GLsizei imageSize;
const void * data;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
internalformat	Specifies the format of the compressed image data stored at address *data*.
width	Specifies the width of the texture image. All implementations support 2D and cube-mapped texture images that are at least 2048 texels wide.
height	Specifies the height of the texture image. All implementations support 2D and cube-mapped texture images that are at least 2048 texels high.
border	This value must be 0.
imageSize	Specifies the number of unsigned bytes of image data starting at the address specified by *data*.
data	Specifies a pointer to the compressed image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

`glCompressedTexImage2D` loads a previously defined, and retrieved, compressed two-dimensional texture image if *target* is `GL_TEXTURE_2D`, or one of the cube map faces such as `GL_TEXTURE_CUBE_MAP_POSITIVE_X`. (see glTexImage2D).

internalformat must be a compressed image format from Table 1 below, or an extension-specified compressed-texture format.

imageSize must be appropriate for the *width*, *height* and *depth* of the *internalformat* specified. The size for an ETC/EAC image is given in Table 1 below.

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

Table 1. Compressed Internal Formats

Compressed Internal Format	Base Internal Format	Image Size
GL_COMPRESSED_R11_EAC	GL_RED	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SIGNED_R11_EAC	GL_RED	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RG11_EAC	GL_RG	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_SIGNED_RG11_EAC	GL_RG	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_RGB8_ETC2	GL_RGB	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SRGB8_ETC2	GL_RGB	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RGBA8_ETC2_EAC	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 16

Errors

GL_INVALID_ENUM is generated if *internalformat* is not one of the specific compressed internal formats: GL_COMPRESSED_R11_EAC, GL_COMPRESSED_SIGNED_R11_EAC, GL_COMPRESSED_RG11_EAC, GL_COMPRESSED_SIGNED_RG11_EAC, GL_COMPRESSED_RGB8_ETC2, GL_COMPRESSED_SRGB8_ETC2, GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2, GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2, GL_COMPRESSED_RGBA8_ETC2_EAC, or GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC.

GL_INVALID_VALUE is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

GL_INVALID_VALUE is generated if *border* is not 0.

GL_INVALID_OPERATION is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

Undefined results, including abnormal program termination, are generated if *`data`* is not encoded in a manner consistent with the extension specification defining the internal compression format.

Associated Gets

glGet with argument `GL_PIXEL_UNPACK_BUFFER_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glCompressedTexImage2D`	✔	✔

See Also

glActiveTexture, glCompressedTexImage3D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage2D, glTexImage3D, glTexSubImage2D, glTexSubImage3D, glTexParameter

Copyright

Name

glCompressedTexImage3D — specify a three-dimensional texture image in a compressed format

C Specification

```
void glCompressedTexImage3D (target, level, internalformat, width,
height, depth, border, imageSize, data);

GLenum target;
GLint level;
GLenum internalformat;
GLsizei width;
GLsizei height;
GLsizei depth;
GLint border;
GLsizei imageSize;
const void * data;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_3D`, or `GL_TEXTURE_2D_ARRAY`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
internalformat	Specifies the format of the compressed image data stored at address *data*.
width	Specifies the width of the texture image.
height	Specifies the height of the texture image.
depth	Specifies the depth of the texture image.
border	This value must be 0.
imageSize	Specifies the number of unsigned bytes of image data starting at the address specified by *data*.
data	Specifies a pointer to the compressed image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

`glCompressedTexImage3D` loads a previously defined, and retrieved, compressed three-dimensional texture image if *target* is `GL_TEXTURE_3D` (see glTexImage3D).

If *target* is `GL_TEXTURE_2D_ARRAY`, *data* is treated as an array of compressed 2D textures.

internalformat must be a compressed image format from Table 1 below, or an extension-specified compressed-texture format.

imageSize must be appropriate for the *width*, *height* and *depth* of the *internalformat* specified. The size for a single slice of ETC/EAC is given in Table 1 below.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see glBindBuffer) while a texture image is specified, *`data`* is treated as a byte offset into the buffer object's data store.

Table 1. Compressed Internal Formats

Compressed Internal Format	Base Internal Format	Image Size
`GL_COMPRESSED_R11_EAC`	`GL_RED`	ceil(*width*/4) * ceil(*height*/4) * 8
`GL_COMPRESSED_SIGNED_R11_EAC`	`GL_RED`	ceil(*width*/4) * ceil(*height*/4) * 8
`GL_COMPRESSED_RG11_EAC`	`GL_RG`	ceil(*width*/4) * ceil(*height*/4) * 16
`GL_COMPRESSED_SIGNED_RG11_EAC`	`GL_RG`	ceil(*width*/4) * ceil(*height*/4) * 16
`GL_COMPRESSED_RGB8_ETC2`	`GL_RGB`	ceil(*width*/4) * ceil(*height*/4) * 8
`GL_COMPRESSED_SRGB8_ETC2`	`GL_RGB`	ceil(*width*/4) * ceil(*height*/4) * 8
`GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2`	`GL_RGBA`	ceil(*width*/4) * ceil(*height*/4) * 8
`GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2`	`GL_RGBA`	ceil(*width*/4) * ceil(*height*/4) * 8
`GL_COMPRESSED_RGBA8_ETC2_EAC`	`GL_RGBA`	ceil(*width*/4) * ceil(*height*/4) * 16
`GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC`	`GL_RGBA`	ceil(*width*/4) * ceil(*height*/4) * 16

Errors

`GL_INVALID_ENUM` is generated if *`internalformat`* is not one of the specific compressed internal formats: `GL_COMPRESSED_R11_EAC`, `GL_COMPRESSED_SIGNED_R11_EAC`, `GL_COMPRESSED_RG11_EAC`, `GL_COMPRESSED_SIGNED_RG11_EAC`, `GL_COMPRESSED_RGB8_ETC2`, `GL_COMPRESSED_SRGB8_ETC2`, `GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2`, `GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2`, `GL_COMPRESSED_RGBA8_ETC2_EAC`, or `GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC`.

`GL_INVALID_VALUE` is generated if *`imageSize`* is not consistent with the format, dimensions, and contents of the specified compressed image data.

`GL_INVALID_VALUE` is generated if *`border`* is not 0.

`GL_INVALID_OPERATION` is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension. The ETC2/EAC texture compression algorithm supports only two-dimensional images. If internalformat is an ETC2/EAC format, `glCompressedTexImage3D` will generate an INVALID_OPERATION error if target is not TEXTURE_2D_ARRAY.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

Undefined results, including abnormal program termination, are generated if *`data`* is not encoded in a manner consistent with the extension specification defining the internal compression format.

Associated Gets

glGet with argument `GL_PIXEL_UNPACK_BUFFER_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glCompressedTexImage3D	-	✔

See Also

glActiveTexture, glCompressedTexImage2D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage2D, glTexSubImage2D, glTexSubImage3D, glTexParameter

Copyright

Name

glCompressedTexSubImage2D — specify a two-dimensional texture subimage in a compressed format

C Specification

```
void glCompressedTexSubImage2D (target, level, xoffset, yoffset, width,
height, format, imageSize, data);

GLenum target;
GLint level;
GLint xoffset;
GLint yoffset;
GLsizei width;
GLsizei height;
GLenum format;
GLsizei imageSize;
const void * data;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
format	Specifies the format of the compressed image data stored at address *data*.
imageSize	Specifies the number of unsigned bytes of image data starting at the address specified by *data*.
data	Specifies a pointer to the compressed image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

`glCompressedTexSubImage2D` redefines a contiguous subregion of an existing two-dimensional texture image. The texels referenced by *data* replace the portion of the existing texture array with x indices *xoffset* and , and the y indices *yoffset* and , inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

format must be a known compressed image format (such as GL_COMPRESSED_R11_EAC) or an extension-specified compressed-texture format.

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

Errors

GL_INVALID_ENUM is generated if *format* is not one of the specific compressed internal formats: GL_COMPRESSED_R11_EAC, GL_COMPRESSED_SIGNED_R11_EAC, GL_COMPRESSED_RG11_EAC, GL_COMPRESSED_SIGNED_RG11_EAC, GL_COMPRESSED_RGB8_ETC2, GL_COMPRESSED_SRGB8_ETC2, GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2, GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2, GL_COMPRESSED_RGBA8_ETC2_EAC, or GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC.

GL_INVALID_VALUE is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

GL_INVALID_OPERATION is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

For ETC2/EAC images GL_INVALID_OPERATION is generated if *width* is not a multiple of four, and *width* + *xoffset* is not equal to the width of the texture level; if *height* is not a multiple of four, and *height* + *yoffset* is not equal to the height of the texture level; or if *xoffset* or *yoffset* is not a multiple of four.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

Associated Gets

glGet with argument GL_PIXEL_UNPACK_BUFFER_BINDING

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glCompressedTexSubImage2D	✔	✔

See Also

glActiveTexture, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage3D, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage2D, glTexImage3D, glTexSubImage2D, glTexSubImage3D, glTexParameter

Copyright

Name

glCompressedTexSubImage3D — specify a three-dimensional texture subimage in a compressed format

C Specification

```
void glCompressedTexSubImage3D (target, level, xoffset, yoffset, zoff-
set, width, height, depth, format, imageSize, data);
```

```
GLenum target;
GLint level;
GLint xoffset;
GLint yoffset;
GLint zoffset;
GLsizei width;
GLsizei height;
GLsizei depth;
GLenum format;
GLsizei imageSize;
const void * data;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_3D` or `GL_TEXTURE_2D_ARRAY`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
zoffset	Specifies a texel offset in the z direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
depth	Specifies the depth of the texture subimage.
format	Specifies the format of the compressed image data stored at address *data*.
imageSize	Specifies the number of unsigned bytes of image data starting at the address specified by *data*.
data	Specifies a pointer to the compressed image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

`glCompressedTexSubImage3D` redefines a contiguous subregion of an existing three-dimensional or two-dimensional array texture image. The texels referenced by *data* replace the portion of the existing texture array with x indices *xoffset* and , and the y indices *yoffset* and , and the z indices *zoffset* and , inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

format must be a known compressed image format (such as GL_COMPRESSED_R11_EAC) or an extension-specified compressed-texture format.

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

Errors

GL_INVALID_ENUM is generated if *format* is not one of the specific compressed internal formats: GL_COMPRESSED_R11_EAC, GL_COMPRESSED_SIGNED_R11_EAC, GL_COMPRESSED_RG11_EAC, GL_COMPRESSED_SIGNED_RG11_EAC, GL_COMPRESSED_RGB8_ETC2, GL_COMPRESSED_SRGB8_ETC2, GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2, GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2, GL_COMPRESSED_RGBA8_ETC2_EAC, or GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC.

GL_INVALID_VALUE is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

GL_INVALID_OPERATION is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension. For ETC2/EAC images GL_INVALID_OPERATION is generated if *width* is not a multiple of four, and *width* + *xoffset* is not equal to the width of the texture level; if *height* is not a multiple of four, and *height* + *yoffset* is not equal to the height of the texture level; or if *xoffset* or *yoffset* is not a multiple of four. The ETC2/EAC texture compression algorithm supports only two-dimensional images. If format is an ETC2/EAC format, glCompressedTexSubImage3D will generate an GL_INVALID_OPERATION error if target is not GL_TEXTURE_2D_ARRAY.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

Associated Gets

glGet with argument GL_PIXEL_UNPACK_BUFFER_BINDING

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glCompressedTexSubImage3D	-	✔

See Also

glActiveTexture, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage2D, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage2D, glTexImage3D, glTexSubImage2D, glTexSubImage3D, glTexParameter

Copyright

Name

glCopyBufferSubData — copy part of the data store of a buffer object to the data store of another buffer object

C Specification

```
void glCopyBufferSubData (readtarget, writetarget, readoffset, write-
offset, size);
```

```
GLenum readtarget;
GLenum writetarget;
GLintptr readoffset;
GLintptr writeoffset;
GLsizeiptr size;
```

Parameters

readtarget	Specifies the target from whose data store data should be read.
writetarget	Specifies the target to whose data store data should be written.
readoffset	Specifies the offset, in basic machine units, within the data store of *readtarget* from which data should be read.
writeoffset	Specifies the offset, in basic machine units, within the data store of *writetarget* to which data should be written.
size	Specifies the size, in basic machine units, of the data to be copied from *readtarget* to *writetarget*.

Description

glCopyBufferSubData copies part of the data store attached to *readtarget* to the data store attached to *writetarget*. The number of basic machine units indicated by *size* is copied from the source, at offset *readoffset* to the destination at *writeoffset*, also in basic machine units.

readtarget and *writetarget* must be GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER or GL_UNIFOR-M_BUFFER. Any of these targets may be used, although the targets GL_COPY_READ_BUFFER and GL_COPY_WRITE_BUFFER are provided specifically to allow copies between buffers without disturbing other GL state.

readoffset, *writeoffset* and *size* must all be greater than or equal to zero. Furthermore, *readoffset* + *size* must not exceeed the size of the buffer object bound to *readtarget*, and *writeoffset* + *size* must not exceeed the size of the buffer bound to *writetarget*. If the same buffer object is bound to both *readtarget* and *writetarget*, then the ranges specified by *readoffset*, *writeoffset* and *size* must not overlap.

Errors

GL_INVALID_VALUE is generated if any of *readoffset*, *writeoffset* or *size* is negative, if *readoffset* + *size* exceeds the size of the buffer object bound to *readtarget* or if *writeoffset* + *size* exceeds the size of the buffer object bound to *writetarget*.

`GL_INVALID_VALUE` is generated if the same buffer object is bound to both *`readtarget`* and *`writetarget`* and the ranges [*`readoffset`*, *`readoffset`* + *`size`*) and [*`writeoffset`*, *`writeoffset`* + *`size`*) overlap.

`GL_INVALID_OPERATION` is generated if zero is bound to *`readtarget`* or *`writetarget`*.

`GL_INVALID_OPERATION` is generated if the buffer object bound to either *`readtarget`* or *`writetarget`* is mapped.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glCopyBufferSubData	-	✔

See Also

glGenBuffers, glBindBuffer, glBufferData, glBufferSubData,

Copyright

Name

glCopyTexImage2D — copy pixels into a 2D texture image

C Specification

```
void glCopyTexImage2D (target, level, internalformat, x, y, width,
height, border);

GLenum target;
GLint level;
GLenum internalformat;
GLint x;
GLint y;
GLsizei width;
GLsizei height;
GLint border;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
internalformat	Specifies the internal format of the texture. Must be one of the following symbolic constants: `GL_ALPHA`, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_R8`, `GL_RG8`, `GL_RGB565`, `GL_RGB8`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_SRGB8`, `GL_SRGB8_ALPHA8`, `GL_R8I`, `GL_R8UI`, `GL_R16I`, `GL_R16UI`, `GL_R32I`, `GL_R32UI`, `GL_RG8I`, `GL_RG8UI`, `GL_RG16I`, `GL_RG16UI`, `GL_RG32I`, `GL_RG32UI`, `GL_RGBA8I`, `GL_RGBA8UI`, `GL_RGB10_A2UI`, `GL_RGBA16I`, `GL_RGBA16UI`, `GL_RGBA32I`, `GL_RGBA32UI`.
x, *y*	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width	Specifies the width of the texture image.
height	Specifies the height of the texture image.
border	Specifies the width of the border. Must be 0.

Description

`glCopyTexImage2D` defines a two-dimensional texture image, or cube-map texture image with pixels from the current `GL_READ_BUFFER`.

The screen-aligned pixel rectangle with lower left corner at (`x`, `y`) and with a width of *`width`* and a height of *`height`* defines the texture array at the mipmap level specified by *`level`*. *`internalformat`* specifies the internal format of the texture array.

The pixels in the rectangle are processed exactly as if glReadPixels had been called, but the process stops after conversion to RGBA values. The error `GL_INVALID_OPERATION` is generated if integer RGBA data is required and the format of the current color buffer is not integer; or if floating- or fixed-point RGBA data is required and the format of the current color buffer is integer.

Pixel ordering is such that lower and screen coordinates correspond to lower and texture coordinates.

If any of the pixels within the specified rectangle of the current `GL_READ_BUFFER` are outside the window associated with the current rendering context, then the values obtained for those pixels are undefined.

When *`internalformat`* is one of the sRGB types, the GL does not automatically convert the source pixels to the sRGB color space.

Notes

An image with height or width of 0 indicates a NULL texture.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

`GL_INVALID_VALUE` is generated if *`level`* is less than 0.

`GL_INVALID_VALUE` may be generated if *`level`* is greater than , where is the returned value of `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if *`width`* or *`height`* is less than 0 or greater than `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if *`border`* is not 0.

`GL_INVALID_ENUM` is generated if *`internalformat`* is not an accepted format.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glCopyTexImage2D	✔	✔

See Also

glCopyTexSubImage2D, glPixelStorei, glTexImage2D, glTexSubImage2D, glTexParameter

Copyright

Name

glCopyTexSubImage2D — copy a two-dimensional texture subimage

C Specification

```
void glCopyTexSubImage2D (target, level, xoffset, yoffset, x, y, width,
height);

GLenum target;
GLint level;
GLint xoffset;
GLint yoffset;
GLint x;
GLint y;
GLsizei width;
GLsizei height;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
x, *y*	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.

Description

`glCopyTexSubImage2D` replaces a rectangular portion of a two-dimensional texture image or cube-map texture image with pixels from the current `GL_READ_BUFFER` (rather than from main memory, as is the case for glTexSubImage2D).

The screen-aligned pixel rectangle with lower left corner at and with width *width* and height *height* replaces the portion of the texture array with x indices *xoffset* through , inclusive, and y indices *yoffset* through , inclusive, at the mipmap level specified by *level*.

The pixels in the rectangle are processed exactly as if glReadPixels had been called, but the process stops after conversion to RGBA values. The error `GL_INVALID_OPERATION` is generated if integer RGBA data is required and the format of the current color buffer is not integer; or if floating- or fixed-point RGBA data is required and the format of the current color buffer is integer.

The destination rectangle in the texture array may not include any texels outside the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

If any of the pixels within the specified rectangle of the current `GL_READ_BUFFER` are outside the read window associated with the current rendering context, then the values obtained for those pixels are undefined.

No change is made to the *internalformat*, *width*, *height*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

Notes

glPixelStorei modes affect texture images.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

`GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous glTexImage2D, glCopyTexImage2D, or glTexStorage2D operation.

`GL_INVALID_VALUE` is generated if *`level`* is less than 0.

`GL_INVALID_VALUE` may be generated if , where is the returned value of `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if , or , where is the `GL_TEXTURE_WIDTH`, is the `GL_TEXTURE_HEIGHT`, of the texture image being modified.

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glCopyTexSubImage2D	✔	✔

See Also

glCopyTexImage2D, glCopyTexSubImage3D, glPixelStorei, glReadBuffer, glTexImage2D, glTexImage3D, glTexParameter, glTexSubImage2D, glTexSubImage3D

Copyright

Name

glCopyTexSubImage3D — copy a three-dimensional texture subimage

C Specification

```
void glCopyTexSubImage3D (target, level, xoffset, yoffset, zoffset, x,
y, width, height);

GLenum target;
GLint level;
GLint xoffset;
GLint yoffset;
GLint zoffset;
GLint x;
GLint y;
GLsizei width;
GLsizei height;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_3D` or `GL_TEXTURE_2D_ARRAY`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
zoffset	Specifies a texel offset in the z direction within the texture array.
x, *y*	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.

Description

`glCopyTexSubImage3D` replaces a rectangular portion of a three-dimensional or two-dimensional array texture image with pixels from the current `GL_READ_BUFFER` (rather than from main memory, as is the case for glTexSubImage3D).

The screen-aligned pixel rectangle with lower left corner at (*x*, *y*) and with width *width* and height *height* replaces the portion of the texture array with x indices *xoffset* through , inclusive, and y indices *yoffset* through , inclusive, at z index *zoffset* and at the mipmap level specified by *level*.

The pixels in the rectangle are processed exactly as if glReadPixels had been called, but the process stops after conversion to RGBA values.

The destination rectangle in the texture array may not include any texels outside the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

If any of the pixels within the specified rectangle of the current `GL_READ_BUFFER` are outside the read window associated with the current rendering context, then the values obtained for those pixels are undefined.

No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

Notes

glPixelStorei modes affect texture images.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_TEXTURE_3D` or `GL_TEXTURE_2D_ARRAY`.

`GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous glTexImage3D or glTexStorage3D operation.

`GL_INVALID_VALUE` is generated if *`level`* is less than 0.

`GL_INVALID_VALUE` may be generated if , where is the returned value of `GL_MAX_3D_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if , , or , where is the `GL_TEXTURE_WIDTH`, is the `GL_TEXTURE_HEIGHT`, is the `GL_TEXTURE_DEPTH` of the texture image being modified.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glCopyTexSubImage3D	-	✔

See Also

glCopyTexImage2D, glCopyTexSubImage2D, glPixelStorei, glReadBuffer, glTexImage2D, glTexImage3D, glTexParameter, glTexSubImage2D, glTexSubImage3D

Copyright

Name

glCreateProgram — Creates a program object

C Specification

```
GLuint **glCreateProgram** (*void*);
```

void;

Description

`glCreateProgram` creates an empty program object and returns a non-zero value by which it can be referenced. A program object is an object to which shader objects can be attached. This provides a mechanism to specify the shader objects that will be linked to create a program. It also provides a means for checking the compatibility of the shaders that will be used to create a program (for instance, checking the compatibility between a vertex shader and a fragment shader). When no longer needed as part of a program object, shader objects can be detached.

One or more executables are created in a program object by successfully attaching shader objects to it with glAttachShader, successfully compiling the shader objects with glCompileShader, and successfully linking the program object with glLinkProgram. These executables are made part of current state when glUseProgram is called. Program objects can be deleted by calling glDeleteProgram. The memory associated with the program object will be deleted when it is no longer part of current rendering state for any context.

Notes

Like buffer and texture objects, the name space for program objects may be shared across a set of contexts, as long as the server sides of the contexts share the same address space. If the name space is shared across contexts, any attached objects and the data associated with those attached objects are shared as well.

Applications are responsible for providing the synchronization across API calls when objects are accessed from different execution threads.

Errors

This function returns 0 if an error occurs creating the program object.

Associated Gets

glGet with the argument `GL_CURRENT_PROGRAM`

glGetActiveAttrib with a valid program object and the index of an active attribute variable

glGetActiveUniform with a valid program object and the index of an active uniform variable

glGetAttachedShaders with a valid program object

glGetAttribLocation with a valid program object and the name of an attribute variable

glGetProgramiv with a valid program object and the parameter to be queried

glGetProgramInfoLog with a valid program object glGetUniform with a valid program object and the location of a uniform variable

glGetUniformLocation with a valid program object and the name of a uniform variable

glIsProgram

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glCreateProgram	✔	✔

See Also

glAttachShader, glBindAttribLocation, glCreateShader, glDeleteProgram, glDetachShader, glLinkProgram, glUniform, glUseProgram, glValidateProgram

Copyright

Name

glCreateShader — Creates a shader object

C Specification

```
GLuint glCreateShader (shaderType);

GLenum shaderType;
```

Parameters

shaderType	Specifies the type of shader to be created. Must be one of `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`.

Description

`glCreateShader` creates an empty shader object and returns a non-zero value by which it can be referenced. A shader object is used to maintain the source code strings that define a shader. *shaderType* indicates the type of shader to be created. Three types of shaders are supported. A shader of type `GL_VERTEX_SHADER` is a shader that is intended to run on the programmable vertex processor. A shader of type `GL_FRAGMENT_SHADER` is a shader that is intended to run on the programmable fragment processor.

When created, a shader object's `GL_SHADER_TYPE` parameter is set to either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`, depending on the value of *shaderType*.

Notes

Like buffer and texture objects, the name space for shader objects may be shared across a set of contexts, as long as the server sides of the contexts share the same address space. If the name space is shared across contexts, any attached objects and the data associated with those attached objects are shared as well.

Applications are responsible for providing the synchronization across API calls when objects are accessed from different execution threads.

Errors

This function returns 0 if an error occurs creating the shader object.

`GL_INVALID_ENUM` is generated if *shaderType* is not an accepted value.

Associated Gets

glGetShaderiv with a valid shader object and the parameter to be queried

glGetShaderInfoLog with a valid shader object

glGetShaderSource with a valid shader object

glIsShader

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
`glCreateShader`	✔	✔

See Also

glAttachShader, glCompileShader, glDeleteShader, glDetachShader, glShaderSource

Copyright

Name

glCullFace — specify whether front- or back-facing polygons can be culled

C Specification

```
void glCullFace (mode);

GLenum mode;
```

Parameters

mode Specifies whether front- or back-facing polygons are candidates for culling. Symbolic constants `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK` are accepted. The initial value is `GL_BACK`.

Description

`glCullFace` specifies whether front- or back-facing polygons are culled (as specified by *mode*) when polygon culling is enabled. Polygon culling is initially disabled. To enable and disable polygon culling, call the glEnable and glDisable commands with the argument `GL_CULL_FACE`.

glFrontFace specifies which of the clockwise and counterclockwise polygons are front-facing and back-facing. See glFrontFace.

Notes

If *mode* is `GL_FRONT_AND_BACK`, no polygons are drawn, but other primitives such as points and lines are drawn.

Errors

`GL_INVALID_ENUM` is generated if *mode* is not an accepted value.

Associated Gets

glIsEnabled with argument `GL_CULL_FACE`

glGet with argument `GL_CULL_FACE_MODE`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glCullFace	✔	✔

See Also

glEnable, glFrontFace

Copyright

Name

glDeleteBuffers — delete named buffer objects

C Specification

```
void glDeleteBuffers (n, buffers);

GLsizei n;
const GLuint * buffers;
```

Parameters

n	Specifies the number of buffer objects to be deleted.
buffers	Specifies an array of buffer objects to be deleted.

Description

`glDeleteBuffers` deletes *n* buffer objects named by the elements of the array *buffers*. After a buffer object is deleted it has no contents, and its name is again unused. Unused names in *buffers* that have been marked as used for the purposes of glGenBuffers are marked as unused again. Unused names in buffers are silently ignored, as is the value zero. If a buffer object is deleted while it is bound, all bindings to that object in the current context are reset to zero. Bindings to that buffer in other contexts are not affected.

`glDeleteBuffers` silently ignores 0's and names that do not correspond to existing buffer objects.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsBuffer

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteBuffers	✔	✔

See Also

glBindBuffer, glGenBuffers, glGet

Copyright

Name

glDeleteFramebuffers — delete framebuffer objects

C Specification

```
void glDeleteFramebuffers (n, framebuffers);

GLsizei n;
GLuint *framebuffers;
```

Parameters

n	Specifies the number of framebuffer objects to be deleted.
framebuffers	A pointer to an array containing *n* framebuffer objects to be deleted.

Description

`glDeleteFramebuffers` deletes the *n* framebuffer objects whose names are stored in the array addressed by *framebuffers*. Unused names in *framebuffers* that have been marked as used for the purposes of glGenFramebuffers are marked as unused again. The name zero is reserved by the GL and is silently ignored, should it occur in *framebuffers*, as are other unused names. Once a framebuffer object is deleted, its name is again unused and it has no attachments. If a framebuffer that is currently bound to one or more of the targets `GL_DRAW_FRAMEBUFFER` or `GL_READ_FRAMEBUFFER` is deleted, it is as though glBindFramebuffer had been executed with the corresponding *target* and *framebuffer* zero.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteFramebuffers	✔	✔

See Also

glGenFramebuffers, glBindFramebuffer, glCheckFramebufferStatus

Copyright

Name

glDeleteProgram — Deletes a program object

C Specification

```
void glDeleteProgram (program);

GLuint program;
```

Parameters

`program`	Specifies the program object to be deleted.

Description

`glDeleteProgram` frees the memory and invalidates the name associated with the program object specified by *`program.`* This command effectively undoes the effects of a call to glCreateProgram.

If a program object is in use as part of current rendering state, it will be flagged for deletion, but it will not be deleted until it is no longer part of current state for any rendering context. If a program object to be deleted has shader objects attached to it, those shader objects will be automatically detached but not deleted unless they have already been flagged for deletion by a previous call to glDeleteShader. A value of 0 for *`program`* will be silently ignored.

To determine whether a program object has been flagged for deletion, call glGetProgramiv with arguments *`program`* and `GL_DELETE_STATUS`.

Errors

`GL_INVALID_VALUE` is generated if *`program`* is not a value generated by OpenGL.

Associated Gets

glGet with argument `GL_CURRENT_PROGRAM`

glGetProgramiv with arguments *`program`* and `GL_DELETE_STATUS`

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteProgram	✔	✔

See Also

glCreateShader, glDetachShader, glUseProgram

Copyright

Name

glDeleteQueries — delete named query objects

C Specification

```
void glDeleteQueries (n, ids);

GLsizei n;
const GLuint * ids;
```

Parameters

n	Specifies the number of query objects to be deleted.
ids	Specifies an array of query objects to be deleted.

Description

`glDeleteQueries` deletes *n* query objects named by the elements of the array *ids*. After a query object is deleted, its name is again unused. Unused names in *ids* that have been marked as used for the purposes of glGenQueries are marked as unused again. If an active query object is deleted its name immediately becomes unused, but the underlying object is not deleted until it is no longer active.

`glDeleteQueries` silently ignores 0's and names that do not correspond to existing query objects.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsQuery

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glDeleteQueries	-	✔

See Also

glBeginQuery, glEndQuery, glGenQueries, glGetQueryiv, glGetQueryObjectuiv

Copyright

Name

glDeleteRenderbuffers — delete renderbuffer objects

C Specification

```
void glDeleteRenderbuffers (n, renderbuffers);

GLsizei n;
GLuint *renderbuffers;
```

Parameters

n	Specifies the number of renderbuffer objects to be deleted.
renderbuffers	A pointer to an array containing *n* renderbuffer objects to be deleted.

Description

`glDeleteRenderbuffers` deletes the *n* renderbuffer objects whose names are stored in the array addressed by *renderbuffers*. Unused names in *renderbuffers* that have been marked as used for the purposes of glGenRenderbuffers are marked as unused again. The name zero is reserved by the GL and is silently ignored, should it occur in *renderbuffers*, as are other unused names. Once a renderbuffer object is deleted, its name is again unused and it has no contents. If a renderbuffer that is currently bound to the target `GL_RENDERBUFFER` is deleted, it is as though glBindRenderbuffer had been executed with a *target* of `GL_RENDERBUFFER` and a *name* of zero.

If a renderbuffer object is attached to one or more attachment points in the currently bound framebuffer, then it as if glFramebufferRenderbuffer had been called, with a *renderbuffer* of zero for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this renderbuffer object is first detached from all attachment ponits in the currently bound framebuffer. Note that the renderbuffer image is specifically *not* detached from any non-bound framebuffers.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glDeleteRenderbuffers	✔	✔

See Also

glGenRenderbuffers, glFramebufferRenderbuffer, glRenderbufferStorage, glRenderbufferStorageMultisample

Copyright

Name

glDeleteSamplers — delete named sampler objects

C Specification

```
void glDeleteSamplers (n, samplers);

GLsizei n;
const GLuint * samplers;
```

Parameters

n	Specifies the number of sampler objects to be deleted.
samplers	Specifies an array of sampler objects to be deleted.

Description

`glDeleteSamplers` deletes *n* sampler objects named by the elements of the array *samplers*. After a sampler object is deleted, its name is again unused. If a sampler object that is currently bound to one or more texture units is deleted, it is as though glBindSampler is called once for each texture unit to which the sampler is bound, with *unit* set to the texture unit and *sampler* set to zero. Unused names in *samplers* that have been marked as used for the purposes of glGenSamplers are marked as unused again. Unused names in *samplers* are silently ignored, as is the reserved name zero.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsSampler

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteSamplers	-	✔

See Also

glGenSamplers, glBindSampler, glDeleteSamplers, glIsSampler

Copyright

Name

glDeleteShader — Deletes a shader object

C Specification

```
void glDeleteShader (shader);

GLuint shader;
```

Parameters

shader Specifies the shader object to be deleted.

Description

`glDeleteShader` frees the memory and invalidates the name associated with the shader object specified by *shader*. This command effectively undoes the effects of a call to glCreateShader.

If a shader object to be deleted is attached to a program object, it will be flagged for deletion, but it will not be deleted until it is no longer attached to any program object, for any rendering context (i.e., it must be detached from wherever it was attached before it will be deleted). A value of 0 for *shader* will be silently ignored.

To determine whether an object has been flagged for deletion, call glGetShaderiv with arguments *shader* and `GL_DELETE_STATUS`.

Errors

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

Associated Gets

glGetAttachedShaders with the program object to be queried

glGetShaderiv with arguments *shader* and `GL_DELETE_STATUS`

glIsShader

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteShader	✔	✔

See Also

glCreateProgram, glCreateShader, glDetachShader, glUseProgram

Copyright

Name

glDeleteSync — delete a sync object

C Specification

```
void glDeleteSync (sync);
GLsync sync;
```

Parameters

sync The sync object to be deleted.

Description

`glDeleteSync` deletes the sync object specified by *sync*. If the fence command corresponding to the specified sync object has completed, or if no glWaitSync or glClientWaitSync commands are blocking on *sync*, the object is deleted immediately. Otherwise, *sync* is flagged for deletion and will be deleted when it is no longer associated with any fence command and is no longer blocking any glWaitSync or glClientWaitSync command. In either case, after `glDeleteSync` returns, the name *sync* is invalid and can no longer be used to refer to the sync object.

`glDeleteSync` will silently ignore a *sync* value of zero.

Errors

`GL_INVALID_VALUE` is generated if *sync* is neither zero or the name of a sync object.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteSync	-	✔

See Also

glFenceSync, glWaitSync, glClientWaitSync

Copyright

Name

glDeleteTextures — delete named textures

C Specification

```
void glDeleteTextures (n, textures);

GLsizei n;
const GLuint * textures;
```

Parameters

n	Specifies the number of textures to be deleted.
textures	Specifies an array of textures to be deleted.

Description

`glDeleteTextures` deletes *n* textures named by the elements of the array *textures*. After a texture is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound is deleted, the binding reverts to 0 (the default texture).

Unused names in *textures* that have been marked as used for the purposes of glGenTextures are marked as unused again. `glDeleteTextures` silently ignores 0's and names that do not correspond to existing textures.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsTexture

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteTextures	✔	✔

See Also

glBindTexture, glCopyTexImage2D, glGenTextures, glGet, glGetTexParameter, glTexImage2D, glTexStorage2D, glTexImage3D, glTexStorage3D, glTexParameter

Copyright

Name

glDeleteTransformFeedbacks — delete transform feedback objects

C Specification

```
void glDeleteTransformFeedbacks (n, ids);

GLsizei n;
const GLuint *ids;
```

Parameters

n	Specifies the number of transform feedback objects to delete.
ids	Specifies an array of names of transform feedback objects to delete.

Description

`glDeleteTransformFeedbacks` deletes the *n* transform feedback objects whose names are stored in the array *ids*. Unused names in *ids* that have been marked as used for the purposes of glGenTransformFeedbacks, are marked as unused again. Unused names in *ids* are ignored, as is the name zero. After a transform feedback object is deleted, its name is again unused and it has no contents. If an active transform feedback object is deleted, its name immediately becomes unused, but the underlying object is not deleted until it is no longer active.

Associated Gets

glGet with argument `GL_TRANSFORM_FEEDBACK_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteTransformFeedbacks	-	✔

See Also

glGenTransformFeedbacks, glBindTransformFeedback, glIsTransformFeedback, glBeginTransformFeedback, glPauseTransformFeedback, glResumeTransformFeedback, glEndTransformFeedback

Copyright

Name

glDeleteVertexArrays — delete vertex array objects

C Specification

```
void glDeleteVertexArrays (n, arrays);

GLsizei n;
const GLuint *arrays;
```

Parameters

n	Specifies the number of vertex array objects to be deleted.
arrays	Specifies the address of an array containing the *n* names of the objects to be deleted.

Description

`glDeleteVertexArrays` deletes *n* vertex array objects whose names are stored in the array addressed by *arrays*. Once a vertex array object is deleted it has no contents and its name is again unused. If a vertex array object that is currently bound is deleted, the binding for that object reverts to zero and the default vertex array becomes current.

Unused names in *arrays* that have been marked as used for the purposes of glGenVertexArrays, are marked as unused again. Unused names in *arrays* are silently ignored, as is the value zero.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDeleteVertexArrays	-	✔

See Also

glGenVertexArrays, glIsVertexArray, glBindVertexArray

Copyright

Name

glDepthFunc — specify the value used for depth buffer comparisons

C Specification

```
void glDepthFunc (func);
```

```
GLenum func;
```

Parameters

func	Specifies the depth comparison function. Symbolic constants `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_LEQUAL`, `GL_GREATER`, `GL_NOTEQUAL`, `GL_GEQUAL`, and `GL_ALWAYS` are accepted. The initial value is `GL_LESS`.

Description

`glDepthFunc` specifies the function used to compare each incoming pixel depth value with the depth value present in the depth buffer. The comparison is performed only if depth testing is enabled. (See glEnable and glDisable of `GL_DEPTH_TEST`.)

func specifies the conditions under which the pixel will be drawn. The comparison functions are as follows:

`GL_NEVER`	Never passes.
`GL_LESS`	Passes if the incoming depth value is less than the stored depth value.
`GL_EQUAL`	Passes if the incoming depth value is equal to the stored depth value.
`GL_LEQUAL`	Passes if the incoming depth value is less than or equal to the stored depth value.
`GL_GREATER`	Passes if the incoming depth value is greater than the stored depth value.
`GL_NOTEQUAL`	Passes if the incoming depth value is not equal to the stored depth value.
`GL_GEQUAL`	Passes if the incoming depth value is greater than or equal to the stored depth value.
`GL_ALWAYS`	Always passes.

The initial value of *func* is `GL_LESS`. Initially, depth testing is disabled. If depth testing is disabled or if no depth buffer exists, it is as if the depth test always passes.

Notes

Even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled. In order to unconditionally write to the depth buffer, the depth test should be enabled and set to `GL_ALWAYS`.

Errors

`GL_INVALID_ENUM` is generated if *func* is not an accepted value.

Associated Gets

glGet with argument `GL_DEPTH_FUNC`

glIsEnabled with argument `GL_DEPTH_TEST`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDepthFunc	✔	✔

See Also

glDepthRangef, glEnable, glPolygonOffset

Copyright

Name

glDepthMask — enable or disable writing into the depth buffer

C Specification

```
void glDepthMask (flag);

GLboolean flag;
```

Parameters

flag	Specifies whether the depth buffer is enabled for writing. If *flag* is `GL_FALSE`, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Description

`glDepthMask` specifies whether the depth buffer is enabled for writing. If *flag* is `GL_FALSE`, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Associated Gets

glGet with argument `GL_DEPTH_WRITEMASK`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDepthMask	✔	✔

See Also

glColorMask, glDepthFunc, glDepthRangef, glStencilMask

Copyright

Name

glDepthRangef — specify mapping of depth values from normalized device coordinates to window coordinates

C Specification

```
void glDepthRangef (n, f);

GLfloat n;
GLfloat f;
```

Parameters

n	Specifies the mapping of the near clipping plane to window coordinates. The initial value is 0.
f	Specifies the mapping of the far clipping plane to window coordinates. The initial value is 1.

Description

After clipping and division by *w*, depth coordinates range from to 1, corresponding to the near and far clipping planes. `glDepthRangef` specifies a linear mapping of the normalized depth coordinates in this range to window depth coordinates. If a fixed-point depth representation is used, the parameters *n* and *f* are clamped to the range [0, 1] when specified.

The setting of (0,1) maps the near plane to 0 and the far plane to 1. With this mapping, the depth buffer range is fully utilized.

Notes

It is not necessary that *n* be less than *f*. Reverse mappings such as , and are acceptable.

Associated Gets

glGet with argument `GL_DEPTH_RANGE`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDepthRangef	✔	✔

See Also

glDepthFunc, glPolygonOffset, glViewport

Copyright

Name

glDetachShader — Detaches a shader object from a program object to which it is attached

C Specification

```
void glDetachShader (program, shader);

GLuint program;
GLuint shader;
```

Parameters

program	Specifies the program object from which to detach the shader object.
shader	Specifies the shader object to be detached.

Description

`glDetachShader` detaches the shader object specified by *shader* from the program object specified by *program*. This command can be used to undo the effect of the command glAttachShader.

If *shader* has already been flagged for deletion by a call to glDeleteShader and it is not attached to any other program object, it will be deleted after it has been detached.

Errors

`GL_INVALID_VALUE` is generated if either *program* or *shader* is a value that was not generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_OPERATION` is generated if *shader* is not attached to *program*.

Associated Gets

glGetAttachedShaders with the handle of a valid program object

glGetShaderiv with arguments *shader* and `GL_DELETE_STATUS`

glIsProgram

glIsShader

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDetachShader	✔	✔

See Also

glAttachShader

Copyright

Name

glDrawArrays — render primitives from array data

C Specification

```
void glDrawArrays (mode, first, count);

GLenum mode;
GLint first;
GLsizei count;
```

Parameters

mode Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` and `GL_TRIANGLES` are accepted.

first Specifies the starting index in the enabled arrays.

count Specifies the number of indices to be rendered.

Description

`glDrawArrays` specifies multiple geometric primitives with very few subroutine calls. It is possible to prespecify separate arrays of attributes and use them to construct a sequence of primitives with a single call to `glDrawArrays`.

When `glDrawArrays` is called, it uses *count* sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element *first*. *mode* specifies what kind of primitives are constructed and how the array elements construct those primitives.

To enable and disable a generic vertex attribute array, call glEnableVertexAttribArray and glDisableVertexAttribArray.

If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state.

Errors

`GL_INVALID_ENUM` is generated if *mode* is not an accepted value.

`GL_INVALID_VALUE` is generated if *count* is negative.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

`GL_INVALID_FRAMEBUFFER_OPERATION` is generated if the currently bound framebuffer is not framebuffer complete (i.e. the return value from glCheckFramebufferStatus is not `GL_FRAMEBUFFER_COMPLETE`).

`GL_INVALID_OPERATION` is generated if recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object’s size, or in exceeding the end position *offset* + *size* - 1, as set by glBindBufferRange.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glDrawArrays	✔	✔

See Also

glCheckFramebufferStatus, glDisableVertexAttribArray, glDrawArraysInstanced, glDrawElements, gl-DrawElementsInstanced, glDrawRangeElements, glEnableVertexAttribArray

Copyright

Name

glDrawArraysInstanced — draw multiple instances of a range of elements

C Specification

```
void glDrawArraysInstanced (mode, first, count, primcount);

GLenum mode;
GLint first;
GLsizei count;
GLsizei primcount;
```

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` and `GL_TRIANGLES` are accepted.
first	Specifies the starting index in the enabled arrays.
count	Specifies the number of indices to be rendered.
primcount	Specifies the number of instances of the specified range of indices to be rendered.

Description

`glDrawArraysInstanced` behaves identically to glDrawArrays except that *primcount* instances of the range of elements are executed. Those attributes that have divisor N where N is other than zero (as specified by glVertexAttribDivisor) advance once every N instances. Thus, the element transferred from instanced vertex attributes is given by:

The value of *instance* may be read by a vertex shader as `gl_InstanceID`.

To enable and disable a generic vertex attribute array, call glEnableVertexAttribArray and glDisableVertexAttribArray.

If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state.

Errors

`GL_INVALID_ENUM` is generated if *mode* is not one of the accepted values.

`GL_INVALID_VALUE` is generated if *count* or *primcount* are negative.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

`GL_INVALID_FRAMEBUFFER_OPERATION` is generated if the currently bound framebuffer is not framebuffer complete (i.e. the return value from glCheckFramebufferStatus is not `GL_FRAMEBUFFER_COMPLETE`).

`GL_INVALID_OPERATION` is generated if recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position *`offset`* + *`size`* - 1, as set by glBindBufferRange.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDrawArraysInstanced	-	✔

See Also

glCheckFramebufferStatus, glDisableVertexAttribArray, glDrawArrays, glDrawElements, glDrawElementsInstanced, glEnableVertexAttribArray , glVertexAttribDivisor

Copyright

Name

glDrawBuffers — Specifies a list of color buffers to be drawn into

C Specification

```
void glDrawBuffers (n, bufs);

GLsizei n;
const GLenum *bufs;
```

Parameters

`n`	Specifies the number of buffers in *`bufs`*.
`bufs`	Points to an array of symbolic constants specifying the buffers into which fragment colors or data values will be written.

Description

`glDrawBuffers` defines an array of buffers into which outputs from the fragment shader data will be written. If a fragment shader writes a value to one or more user defined output variables, then the value of each variable will be written into the buffer specified at a location within *`bufs`* corresponding to the location assigned to that user defined output. The draw buffer used for user defined outputs assigned to locations greater than or equal to *`n`* is implicitly set to `GL_NONE` and any data written to such an output is discarded.

The symbolic constants contained in *`bufs`* must be one of the following, depending on whether GL is bound to the default framebuffer or not:

`GL_NONE`	The fragment shader output value is not written into any color buffer.
`GL_BACK`	The fragment shader output value is written into the back color buffer.
`GL_COLOR_ATTACHMENT`*n*	The fragment shader output value is written into the *n*th color attachment of the current framebuffer. *n* may range from zero to the value of `GL_MAX_COLOR_ATTACHMENTS`.

Except for `GL_NONE`, the preceding symbolic constants may not appear more than once in *`bufs`*. The maximum number of draw buffers supported is implementation dependent and can be queried by calling glGet with the argument `GL_MAX_DRAW_BUFFERS`.

Notes

If a fragment shader does not write to a user defined output variable, the values of the fragment colors following shader execution are undefined. For each fragment generated in this situation, a different value may be written into each of the buffers specified by *`bufs`*.

Errors

`GL_INVALID_ENUM` is generated if one of the values in *`bufs`* is not an accepted value.

`GL_INVALID_OPERATION` is generated if the GL is bound to the default framebuffer and *`n`* is not 1, or if the value in *`bufs`* is one of the `GL_COLOR_ATTACHMENT`*n* tokens.

`GL_INVALID_OPERATION` is generated if the GL is bound to a framebuffer object and the ith buffer listed in *`bufs`* is anything other than `GL_NONE` or `GL_COLOR_ATTACHMENTS`*i*.

`GL_INVALID_VALUE` is generated if *`n`* is less than 0 or greater than `GL_MAX_DRAW_BUFFERS`.

Associated Gets

glGet with argument `GL_MAX_DRAW_BUFFERS`

glGet with argument `GL_DRAW_BUFFER`*i* where *i* indicates the number of the draw buffer whose value is to be queried.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDrawBuffers	-	✔

See Also

glReadBuffer

Copyright

Name

glDrawElements — render primitives from array data

C Specification

```
void glDrawElements (mode, count, type, indices);

GLenum mode;
GLsizei count;
GLenum type;
const void * indices;
```

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` and `GL_TRIANGLES` are accepted.
count	Specifies the number of elements to be rendered.
type	Specifies the type of the values in *indices*. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.
indices	Specifies a byte offset (cast to a pointer type) into the buffer bound to `GL_ELEMENT_ARRAY_BUFFER` to start reading indices from. If no buffer is bound, specifies a pointer to the location where the indices are stored.

Description

`glDrawElements` specifies multiple geometric primitives with very few subroutine calls. It is possible to prespecify separate arrays of attributes and use them to construct a sequence of primitives with a single call to `glDrawElements`.

When `glDrawElements` is called, it uses *count* sequential elements from an enabled array, starting at *indices* to construct a sequence of geometric primitives. *mode* specifies what kind of primitives are constructed and how the array elements construct these primitives. If more than one array is enabled, each is used.

To enable and disable a generic vertex attribute array, call glEnableVertexAttribArray and glDisableVertexAttribArray.

If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state.

Errors

`GL_INVALID_ENUM` is generated if *mode* is not an accepted value.

`GL_INVALID_VALUE` is generated if *count* is negative.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array or the element array and the buffer object's data store is currently mapped.

`GL_INVALID_FRAMEBUFFER_OPERATION` is generated if the currently bound framebuffer is not framebuffer complete (i.e. the return value from glCheckFramebufferStatus is not `GL_FRAMEBUFFER_COMPLETE`).

`GL_INVALID_OPERATION` is generated if transform feedback is active and not paused.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDrawElements	✔	✔

See Also

glCheckFramebufferStatus, glDisableVertexAttribArray, glDrawArrays, glDrawArraysInstanced, glDrawElementsInstanced, glDrawRangeElements, glEnableVertexAttribArray

Copyright

Name

glDrawElementsInstanced — draw multiple instances of a set of elements

C Specification

```
void glDrawElementsInstanced (mode, count, type, indices, primcount);

GLenum mode;
GLsizei count;
GLenum type;
const void * indices;
GLsizei primcount;
```

Parameters

`mode`	Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` and `GL_TRIANGLES` are accepted.
`count`	Specifies the number of elements to be rendered.
`type`	Specifies the type of the values in *`indices`*. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.
`indices`	Specifies a byte offset (cast to a pointer type) into the buffer bound to `GL_ELEMENT_ARRAY_BUFFER` to start reading indices from. If no buffer is bound, specifies a pointer to the location where the indices are stored.
`primcount`	Specifies the number of instances of the specified range of indices to be rendered.

Description

`glDrawElementsInstanced` behaves identically to glDrawElements except that *`primcount`* instances of the set of elements are executed. Those attributes that have divisor N where N is other than zero (as specified by glVertexAttribDivisor) advance once every N instances. Thus, the element transferred from instanced vertex attributes is given by:

The value of *`instance`* may be read by a vertex shader as `gl_InstanceID`.

To enable and disable a generic vertex attribute array, call glEnableVertexAttribArray and glDisableVertexAttribArray.

If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state.

Errors

`GL_INVALID_ENUM` is generated if *`mode`* is not one of `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, or `GL_TRIANGLES`.

`GL_INVALID_VALUE` is generated if *`count`* or *`primcount`* are negative.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

`GL_INVALID_FRAMEBUFFER_OPERATION` is generated if the currently bound framebuffer is not framebuffer complete (i.e. the return value from glCheckFramebufferStatus is not `GL_FRAMEBUFFER_COMPLETE`).

`GL_INVALID_OPERATION` is generated if transform feedback is active and not paused.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDrawElementsInstanced	-	✔

See Also

glCheckFramebufferStatus, glDisableVertexAttribArray, glDrawElements, glDrawArrays, glDrawArraysInstanced, glDrawRangeElements, glEnableVertexAttribArray, glVertexAttribDivisor

Copyright

Name

glDrawRangeElements — render primitives from array data

C Specification

```
void glDrawRangeElements (mode, start, end, count, type, indices);

GLenum mode;
GLuint start;
GLuint end;
GLsizei count;
GLenum type;
const void * indices;
```

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` and `GL_TRIANGLES` are accepted.
start	Specifies the minimum array index contained in *indices*.
end	Specifies the maximum array index contained in *indices*.
count	Specifies the number of elements to be rendered.
type	Specifies the type of the values in *indices*. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.
indices	Specifies a byte offset (cast to a pointer type) into the buffer bound to `GL_ELEMENT_ARRAY_BUFFER` to start reading indices from. If no buffer is bound, specifies a pointer to the location where the indices are stored.

Description

`glDrawRangeElements` is a restricted form of glDrawElements. *mode*, *count*, and *type* match the corresponding arguments to glDrawElements, with the additional constraint that all values in the arrays *count* must lie between *start* and *end*, inclusive.

Implementations denote recommended maximum amounts of vertex and index data, which may be queried by calling glGet with argument `GL_MAX_ELEMENTS_VERTICES` and `GL_MAX_ELEMENTS_INDICES`. If is greater than the value of `GL_MAX_ELEMENTS_VERTICES`, or if *count* is greater than the value of `GL_MAX_ELEMENTS_INDICES`, then the call may operate at reduced performance. There is no requirement that all vertices in the range be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.

When `glDrawRangeElements` is called, it uses *count* sequential elements from an enabled array, starting at *start* to construct a sequence of geometric primitives. *mode* specifies what kind of primitives are constructed, and how the array elements construct these primitives. If more than one array is enabled, each is used.

To enable and disable a generic vertex attribute array, call glEnableVertexAttribArray and glDisableVertexAttribArray.

If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state.

Errors

It is an error for indices to lie outside the range , but implementations may not check for this situation. Such indices cause implementation-dependent behavior.

`GL_INVALID_ENUM` is generated if *`mode`* is not an accepted value.

`GL_INVALID_VALUE` is generated if *`count`* is negative.

`GL_INVALID_VALUE` is generated if .

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array or the element array and the buffer object's data store is currently mapped.

`GL_INVALID_FRAMEBUFFER_OPERATION` is generated if the currently bound framebuffer is not framebuffer complete (i.e. the return value from glCheckFramebufferStatus is not `GL_FRAMEBUFFER_COMPLETE`).

`GL_INVALID_OPERATION` is generated if transform feedback is active and not paused.

Associated Gets

glGet with argument `GL_MAX_ELEMENTS_VERTICES`

glGet with argument `GL_MAX_ELEMENTS_INDICES`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glDrawRangeElements	-	✔

See Also

glCheckFramebufferStatus, glDisableVertexAttribArray, glDrawArrays, glDrawElements, glDrawArraysInstanced, glDrawElementsInstanced, glEnableVertexAttribArray,

Copyright

Name

glEnable — enable or disable server-side GL capabilities

C Specification

```
void glEnable (cap);
GLenum cap;
void glDisable (cap);
GLenum cap;
```

Parameters

cap Specifies a symbolic constant indicating a GL capability.

Description

`glEnable` and glDisable enable and disable various capabilities. Use glIsEnabled or glGet to determine the current setting of any capability. The initial value for each capability with the exception of `GL_DITHER` is `GL_FALSE`. The initial value for `GL_DITHER` is `GL_TRUE`.

Both `glEnable` and glDisable take a single argument, *cap*, which can assume one of the following values:

`GL_BLEND`	If enabled, blend the computed fragment color values with the values in the color buffers. See glBlendFunc.
`GL_CULL_FACE`	If enabled, cull polygons based on their winding in window coordinates. See glCullFace.
`GL_DEPTH_TEST`	If enabled, do depth comparisons and update the depth buffer. Note that even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled. See glDepthFunc and glDepthRangef.
`GL_DITHER`	If enabled, dither color components or indices before they are written to the color buffer.
`GL_POLYGON_OFFSET_FILL`	If enabled, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See glPolygonOffset.
`GL_PRIMITIVE_RESTART_FIXED_INDEX`	Enables primitive restarting. If enabled, any one of the draw commands which transfers a set of generic attribute array elements to the GL will restart the primitive when the index of the vertex is equal to where *n* is 8, 16 or 32 if the type is `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`, respectively.
`GL_RASTERIZER_DISCARD`	If enabled, primitives are discarded immediately before the rasterization stage, but after the optional transform feedback stage. glClear and glClearBuffer* commands are also ignored.

`GL_SAMPLE_ALPHA_TO_COVERAGE`	If enabled, compute a temporary coverage value where each bit is determined by the alpha value at the corresponding sample location. The temporary coverage value is then ANDed with the fragment coverage value.
`GL_SAMPLE_COVERAGE`	If enabled, the fragment's coverage is ANDed with the temporary coverage value. If `GL_SAMPLE_COVERAGE_INVERT` is set to `GL_TRUE`, invert the coverage value. See glSampleCoverage.
`GL_SCISSOR_TEST`	If enabled, discard fragments that are outside the scissor rectangle. See glScissor.
`GL_STENCIL_TEST`	If enabled, do stencil testing and update the stencil buffer. See glStencilFunc and glStencilOp.

Errors

`GL_INVALID_ENUM` is generated if *`cap`* is not one of the values listed previously.

Associated Gets

glIsEnabled

glGet

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glEnable`	✔	✔
`glDisable`	✔	✔

See Also

glBlendFunc, glCullFace, glDepthFunc, glDepthRangef, glGet, glIsEnabled, glPolygonOffset, glSampleCoverage, glScissor, glStencilFunc, glStencilOp,

Copyright

Name

glEnableVertexAttribArray — Enable or disable a generic vertex attribute array

C Specification

`void` **`glEnableVertexAttribArray`** `(`*`index`*`);`

`GLuint` *`index`*`;`

`void` **`glDisableVertexAttribArray`** `(`*`index`*`);`

`GLuint` *`index`*`;`

Parameters

`index` Specifies the index of the generic vertex attribute to be enabled or disabled.

Description

`glEnableVertexAttribArray` enables the generic vertex attribute array specified by *`index`*. `glDisableVertexAttribArray` disables the generic vertex attribute array specified by *`index`*. By default, all generic vertex attribute arrays are disabled. If enabled, the values in the generic vertex attribute array will be accessed and used for rendering when calls are made to vertex array commands such as glDrawArrays, glDrawArraysInstanced, glDrawElements, glDrawElementsInstanced, or glDrawRangeElements.

Errors

`GL_INVALID_VALUE` is generated if *`index`* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

Associated Gets

glGet with argument `GL_MAX_VERTEX_ATTRIBS`

glGetVertexAttrib with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_ENABLED`

glGetVertexAttribPointerv with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_POINTER`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glEnableVertexAttribArray`	✔	✔
`glDisableVertexAttribArray`	✔	✔

See Also

glBindAttribLocation, glDrawArrays, glDrawArraysInstanced, glDrawElements, glDrawElementsInstanced, glDrawRangeElements, glVertexAttrib, glVertexAttribPointer

Copyright

Name

glFenceSync — create a new sync object and insert it into the GL command stream

C Specification

```
GLsync glFenceSync (condition, flags);
```

```
GLenum condition;
GLbitfield flags;
```

Parameters

condition	Specifies the condition that must be met to set the sync object's state to signaled. *condition* must be `GL_SYNC_GPU_COMMANDS_COMPLETE`.
flags	Specifies a bitwise combination of flags controlling the behavior of the sync object. No flags are presently defined for this operation and *flags* must be zero.[1]

Description

`glFenceSync` creates a new fence sync object, inserts a fence command into the GL command stream and associates it with that sync object, and returns a non-zero name corresponding to the sync object.

When the specified *condition* of the sync object is satisfied by the fence command, the sync object is signaled by the GL, causing any glWaitSync, glClientWaitSync commands blocking in *sync* to *unblock*. No other state is affected by `glFenceSync` or by the execution of the associated fence command.

condition must be `GL_SYNC_GPU_COMMANDS_COMPLETE`. This condition is satisfied by completion of the fence command corresponding to the sync object and all preceding commands in the same command stream. The sync object will not be signaled until all effects from these commands on GL client and server state and the framebuffer are fully realized. Note that completion of the fence command occurs once the state of the corresponding sync object has been changed, but commands waiting on that sync object may not be unblocked until after the fence command completes.

Errors

`GL_INVALID_ENUM` is generated if *condition* is not `GL_SYNC_GPU_COMMANDS_COMPLETE`.

`GL_INVALID_VALUE` is generated if *flags* is not zero.

Additionally, if `glFenceSync` fails, it will return zero.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glFenceSync	-	✔

See Also

glDeleteSync, glGetSynciv, glWaitSync, glClientWaitSync

[1] *flags* is a placeholder for anticipated future extensions of fence sync object capabilities.

Copyright

Name

glFinish — block until all GL execution is complete

C Specification

```
void glFinish (void);
```

void;

Description

`glFinish` does not return until the effects of all previously called GL commands are complete. Such effects include all changes to GL state, all changes to connection state, and all changes to the frame buffer contents.

Notes

`glFinish` requires a round trip to the server.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glFinish	✔	✔

See Also

glFlush

Copyright

Name

glFlush — force execution of GL commands in finite time

C Specification

```
void glFlush (void);
```

void;

Description

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. `glFlush` empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

Because any GL program might be executed over a network, or on an accelerator that buffers commands, all programs should call `glFlush` whenever they count on having all of their previously issued commands completed. For example, call `glFlush` before waiting for user input that depends on the generated image.

Notes

`glFlush` can return at any time. It does not wait until the execution of all previously issued GL commands is complete.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glFlush	✔	✔

See Also

glFinish

Copyright

Name

glFlushMappedBufferRange — indicate modifications to a range of a mapped buffer

C Specification

```
void glFlushMappedBufferRange (target, offset, length);

GLenum target;
GLintptr offset;
GLsizeiptr length;
```

Parameters

target	Specifies the target of the flush operation. *target* must be `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.
offset	Specifies the start of the buffer subrange, in basic machine units.
length	Specifies the length of the buffer subrange, in basic machine units.

Description

`glFlushMappedBufferRange` indicates that modifications have been made to a range of a mapped buffer. The buffer must previously have been mapped with the `GL_MAP_FLUSH_EXPLICIT` flag. *offset* and *length* indicate the modified subrange of the mapping, in basic units. The specified subrange to flush is relative to the start of the currently mapped range of the buffer. `glFlushMappedBufferRange` may be called multiple times to indicate distinct subranges of the mapping which require flushing.

Errors

`GL_INVALID_VALUE` is generated if *offset* or *length* is negative, or if *offset* + *length* exceeds the size of the mapping.

`GL_INVALID_OPERATION` is generated if zero is bound to *target*.

`GL_INVALID_OPERATION` is generated if the buffer bound to *target* is not mapped, or is mapped without the `GL_MAP_FLUSH_EXPLICIT` flag.

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glFlushMappedBufferRange	-	✔

See Also

glMapBufferRange, glUnmapBuffer

Copyright

Name

glFramebufferRenderbuffer — attach a renderbuffer as a logical buffer to the currently bound framebuffer object

C Specification

```
void glFramebufferRenderbuffer (target, attachment, renderbuffertarget,
renderbuffer);
```

```
GLenum target;
GLenum attachment;
GLenum renderbuffertarget;
GLuint renderbuffer;
```

Parameters

target	Specifies the framebuffer target. *target* must be `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, or `GL_FRAMEBUFFER`. `GL_FRAMEBUFFER` is equivalent to `GL_DRAW_FRAMEBUFFER`.
attachment	Specifies the attachment point of the framebuffer.
renderbuffertarget	Specifies the renderbuffer target and must be `GL_RENDERBUFFER`.
renderbuffer	Specifies the name of an existing renderbuffer object of type *renderbuffertarget* to attach.

Description

`glFramebufferRenderbuffer` attaches a renderbuffer as one of the logical buffers of the currently bound framebuffer object. *renderbuffer* is the name of the renderbuffer object to attach and must be either zero, or the name of an existing renderbuffer object of type *renderbuffertarget*. If *renderbuffer* is not zero and if `glFramebufferRenderbuffer` is successful, then the renderbuffer name *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*.

The value of `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `GL_RENDERBUFFER` and the value of `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values. No change is made to the state of the renderbuuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer *target* is broken.

Calling `glFramebufferRenderbuffer` with the renderbuffer name zero will detach the image, if any, identified by *attachment*, in the framebuffer currently bound to *target*. All state values of the attachment point specified by attachment in the object bound to target are set to their default values.

Setting *attachment* to the value `GL_DEPTH_STENCIL_ATTACHMENT` is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *renderbuffer*, which should have the base internal format `GL_DEPTH_STENCIL`.

Errors

`GL_INVALID_ENUM` is generated if *target* is not one of the accepted tokens.

`GL_INVALID_ENUM` is generated if *`renderbuffertarget`* is not `GL_RENDERBUFFER`.

`GL_INVALID_OPERATION` is generated if zero is bound to *`target`*.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glFramebufferRenderbuffer	✔	✔

See Also

glGenFramebuffers, glBindFramebuffer, glGenRenderbuffers, glFramebufferTexture, glFramebufferTexture2D, glFramebufferTextureLayer

Copyright

Name

glFramebufferTexture2D — attach a level of a texture object as a logical buffer to the currently bound framebuffer object

C Specification

```
void glFramebufferTexture2D (target, attachment, textarget, texture,
level);

GLenum target;
GLenum attachment;
GLenum textarget;
GLuint texture;
GLint level;
```

Parameters

target	Specifies the framebuffer target. *target* must be `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, or `GL_FRAMEBUFFER`. `GL_FRAMEBUFFER` is equivalent to `GL_DRAW_FRAMEBUFFER`.
attachment	Specifies the attachment point of the framebuffer. *attachment* must be `GL_COLOR_ATTACHMENT`*i*, `GL_DEPTH_ATTACHMENT`, `GL_STENCIL_ATTACHMENT` or `GL_DEPTH_STENCIL_ATTACHMENT`.
textarget	Specifies a 2D texture target, or for cube map textures, which face is to be attached.
texture	Specifies the texture object to attach to the framebuffer attachment point named by *attachment*.
level	Specifies the mipmap level of *texture* to attach.

Description

`glFramebufferTexture2D` attaches a selected mipmap level or image of a texture object as one of the logical buffers of the framebuffer object currently bound to *target*. *target* must be `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, or `GL_FRAMEBUFFER`. `GL_FRAMEBUFFER` is equivalent to `GL_DRAW_FRAMEBUFFER`.

attachment specifies the logical attachment of the framebuffer and must be `GL_COLOR_ATTACHMENT`*i*, `GL_DEPTH_ATTACHMENT`, `GL_STENCIL_ATTACHMENT` or `GL_DEPTH_STENCIL_ATTACHMENT`. *i* in `GL_COLOR_ATTACHMENT`*i* may range from zero to the value of `GL_MAX_COLOR_ATTACHMENTS` - 1. Attaching a level of a texture to `GL_DEPTH_STENCIL_ATTACHMENT` is equivalent to attaching that level to both the `GL_DEPTH_ATTACHMENT` *and* the `GL_STENCIL_ATTACHMENT` attachment points simultaneously.

textarget specifies what type of texture is named by *texture*, and for cube map textures, specifies the face that is to be attached. If *texture* is not zero, it must be the name of an existing two dimensional texture with *textarget* set to `GL_TEXTURE_2D`, unless it is a cube map texture, in which case *textarget* must be `GL_TEXTURE_CUBE_MAP_POSITIVE_X` `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

If *texture* is non-zero, the specified *level* of the texture object named *texture* is attached to the framebuffer attachment point named by *attachment*.

If *textarget* is one of GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, then *level* must be greater than or equal to zero and less than or equal to $\log_2$ of the value of GL_MAX_CUBE_MAP_TEXTURE_SIZE. If *textarget* is GL_TEXTURE_2D, *level* must be greater than or equal to zero and no larger than $\log_2$ of the value of GL_MAX_TEXTURE_SIZE.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the accepted tokens.

GL_INVALID_ENUM is generated if *attachment* is not one of the attachment points listed above.

GL_INVALID_OPERATION is generated if zero is bound to *target*.

GL_INVALID_OPERATION is generated if *textarget* and *texture* are not compatible.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glFramebufferTexture2D	✔	✔

See Also

glGenFramebuffers, glBindFramebuffer, glGenRenderbuffers, glFramebufferRenderbuffer, glFramebufferTextureLayer,

Copyright

Name

glFramebufferTextureLayer — attach a single layer of a texture to a framebuffer

C Specification

void **glFramebufferTextureLayer** (*target*, *attachment*, *texture*, *level*, *layer*);

```
GLenum target;
GLenum attachment;
GLuint texture;
GLint level;
GLint layer;
```

Parameters

target	Specifies the framebuffer target. *target* must be GL_DRAW_FRAMEBUFFER, GL_READ_FRAMEBUFFER, or GL_FRAMEBUFFER. GL_FRAMEBUFFER is equivalent to GL_DRAW_FRAMEBUFFER.
attachment	Specifies the attachment point of the framebuffer. *attachment* must be GL_COLOR_ATTACHMENT*i*, GL_DEPTH_ATTACHMENT, GL_STENCIL_ATTACHMENT or GL_DEPTH_STENCIL_ATTACHMMENT.
texture	Specifies the texture object to attach to the framebuffer attachment point named by *attachment*.
level	Specifies the mipmap level of *texture* to attach.
layer	Specifies the layer of *texture* to attach.

Description

glFramebufferTextureLayer operates like glFramebufferTexture2D, except that only a single layer of the texture level, given by *layer*, is attached to the attachment point. If *texture* is not zero, *layer* must be greater than or equal to zero. *texture* must either be zero or the name of an existing three-dimensional texture, or a two-dimensional array texture.

If *texture* is a 3D texture, then *level* must be greater than or equal to zero and less than or equal to $\log_2$ of the value of GL_MAX_3D_TEXTURE_SIZE. If *texture* is a 2D array texture, *level* must be greater than or equal to zero and no larger than $\log_2$ of the value of GL_MAX_TEXTURE_SIZE.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the accepted tokens.

GL_INVALID_ENUM is generated if *attachment* is not one of the accepted tokens.

GL_INVALID_VALUE is generated if *texture* is not zero or the name of an existing texture object.

GL_INVALID_VALUE is generated if *texture* is not zero and *layer* is negative.

GL_INVALID_VALUE is generated if *texture* is not zero and *layer* is greater than the value of GL_MAX_3D_TEXTURE_SIZE minus one for a 3D texture or greater than the value of GL_MAX_ARRAY_TEXTURE_LAYERS minus one for a 2D array texture.

`GL_INVALID_OPERATION` is generated if zero is bound to *`target`*.

`GL_INVALID_OPERATION` is generated if *`texture`* is not zero or the name of an existing three-dimensional texture, or a two-dimensional array texture.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glFramebufferTextureLayer	-	✔

See Also

glGenFramebuffers, glBindFramebuffer, glGenRenderbuffers, glFramebufferRenderbuffer, glFramebufferTexture2D,

Copyright

Name

glFrontFace — define front- and back-facing polygons

C Specification

```
void glFrontFace (mode);

GLenum mode;
```

Parameters

mode Specifies the orientation of front-facing polygons. `GL_CW` and `GL_CCW` are accepted. The initial value is `GL_CCW`.

Description

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call glEnable and glDisable with argument `GL_CULL_FACE`.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon's winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. `glFrontFace` specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing `GL_CCW` to *mode* selects counterclockwise polygons as front-facing; `GL_CW` selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.

Errors

`GL_INVALID_ENUM` is generated if *mode* is not an accepted value.

Associated Gets

glGet with argument `GL_FRONT_FACE`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glFrontFace	✔	✔

See Also

glCullFace,

Copyright

Name

glGenBuffers — generate buffer object names

C Specification

```
void glGenBuffers (n, buffers);

GLsizei n;
GLuint * buffers;
```

Parameters

n	Specifies the number of buffer object names to be generated.
buffers	Specifies an array in which the generated buffer object names are stored.

Description

`glGenBuffers` returns *n* buffer object names in *buffers*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenBuffers`.

Buffer object names returned by a call to `glGenBuffers` are not returned by subsequent calls, unless they are first deleted with glDeleteBuffers.

The names returned in *buffers* are marked as used, for the purposes of `glGenBuffers` only, but they acquire state and type only when they are first bound by calling glBindBuffer.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsBuffer

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenBuffers	✔	✔

See Also

glBindBuffer, glDeleteBuffers, glGet

Copyright

Name

glGenFramebuffers — generate framebuffer object names

C Specification

```
void glGenFramebuffers (n, framebuffers);

GLsizei n;
GLuint *framebuffers;
```

Parameters

n	Specifies the number of framebuffer object names to generate.
framebuffers	Specifies an array in which the generated framebuffer object names are stored.

Description

`glGenFramebuffers` returns *n* framebuffer object names in *framebuffers*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenFramebuffers`.

Framebuffer object names returned by a call to `glGenFramebuffers` are not returned by subsequent calls, unless they are first deleted with glDeleteFramebuffers.

The names returned in *framebuffers* are marked as used, for the purposes of `glGenFramebuffers` only, but they acquire state and type only when they are first bound.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsFramebuffer, glGetFramebufferAttachmentParameteriv

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenFramebuffers	✔	✔

See Also

glBindFramebuffer, glDeleteFramebuffers, glGet

Copyright

Name

glGenQueries — generate query object names

C Specification

```
void glGenQueries (n, ids);

GLsizei n;
GLuint * ids;
```

Parameters

n	Specifies the number of query object names to be generated.
ids	Specifies an array in which the generated query object names are stored.

Description

`glGenQueries` returns *n* query object names in *ids*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenQueries`.

Query object names returned by a call to `glGenQueries` are not returned by subsequent calls, unless they are first deleted with glDeleteQueries.

The names returned in *ids* are marked as used, for the purposes of `glGenQueries` only, but no query objects are associated with the returned query object names until they are first used by calling glBegin-Query.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsQuery

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenQueries	-	✔

See Also

glBeginQuery, glDeleteQueries, glEndQuery, glGet

Copyright

Name

glGenRenderbuffers — generate renderbuffer object names

C Specification

```
void glGenRenderbuffers (n, renderbuffers);

GLsizei n;
GLuint *renderbuffers;
```

Parameters

n	Specifies the number of renderbuffer object names to generate.
renderbuffers	Specifies an array in which the generated renderbuffer object names are stored.

Description

`glGenRenderbuffers` returns *n* renderbuffer object names in *renderbuffers*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenRenderbuffers`.

Renderbuffer object names returned by a call to `glGenRenderbuffers` are not returned by subsequent calls, unless they are first deleted with glDeleteRenderbuffers.

The names returned in *renderbuffers* are marked as used, for the purposes of `glGenRenderbuffers` only, but they acquire state and type only when they are first bound.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsRenderbuffer, glGetRenderbufferParameteriv

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenRenderbuffers	✔	✔

See Also

glBindRenderbuffer, glFramebufferRenderbuffer, glDeleteRenderbuffers

Copyright

Name

glGenSamplers — generate sampler object names

C Specification

```
void glGenSamplers (n, samplers);

GLsizei n;
GLuint *samplers;
```

Parameters

n	Specifies the number of sampler object names to generate.
samplers	Specifies an array in which the generated sampler object names are stored.

Description

`glGenSamplers` returns *n* sampler object names in *samplers*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenSamplers`.

Sampler object names returned by a call to `glGenSamplers` are not returned by subsequent calls, unless they are first deleted with glDeleteSamplers.

The names returned in *samplers* are marked as used, for the purposes of `glGenSamplers` only, but they acquire state and type only when they are first used as a parameter to glBindSampler, glSamplerParameter*, glGetSamplerParameter* or glIsSampler.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsSampler, glGetSamplerParameter

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenSamplers	-	✔

See Also

glBindSampler, glDeleteSamplers, glIsSampler, glGetSamplerParameter, glSamplerParameter

Copyright

Name

glGenTextures — generate texture names

C Specification

```
void glGenTextures (n, textures);

GLsizei n;
GLuint * textures;
```

Parameters

n	Specifies the number of texture names to be generated.
textures	Specifies an array in which the generated texture names are stored.

Description

`glGenTextures` returns *n* texture names in *textures*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenTextures`.

Texture names returned by a call to `glGenTextures` are not returned by subsequent calls, unless they are first deleted with glDeleteTextures.

The names returned in *textures* are marked as used, for the purposes of `glGenTextures` only, but they acquire state and dimensionality only when they are first bound using glBindTexture.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsTexture

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenTextures	✔	✔

See Also

glBindTexture, glCopyTexImage2D, glDeleteTextures, glGet, glGetTexParameter, glTexImage2D, glTexImage3D, glTexParameter

Copyright

Name

glGenTransformFeedbacks — reserve transform feedback object names

C Specification

```
void glGenTransformFeedbacks (n, ids);

GLsizei n;
GLuint *ids;
```

Parameters

n	Specifies the number of transform feedback object names to reserve.
ids	Specifies an array of into which the reserved names will be written.

Description

`glGenTransformFeedbacks` returns *n* transform feedback object names in *ids*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenTransformFeedbacks`.

Transform feedback object names returned by a call to `glGenTransformFeedbacks` are not returned by subsequent calls, unless they are first deleted with glDeleteTransformFeedbacks.

The names returned in *ids* are marked as used, for the purposes of `glGenTransformFeedbacks` only, but they acquire state and type only when they are first bound.

Associated Gets

glGet with argument `GL_TRANSFORM_FEEDBACK_BINDING`

glIsTransformFeedback

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenTransformFeedbacks	-	✔

See Also

glDeleteTransformFeedbacks, glBindTransformFeedback, glIsTransformFeedback, glBeginTransformFeedback, glPauseTransformFeedback, glResumeTransformFeedback, glEndTransformFeedback

Copyright

Name

glGenVertexArrays — generate vertex array object names

C Specification

```
void glGenVertexArrays (n, arrays);

GLsizei n;
GLuint *arrays;
```

Parameters

n	Specifies the number of vertex array object names to generate.
arrays	Specifies an array in which the generated vertex array object names are stored.

Description

`glGenVertexArrays` returns *n* vertex array object names in *arrays*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `glGenVertexArrays`.

Vertex array object names returned by a call to `glGenVertexArrays` are not returned by subsequent calls, unless they are first deleted with glDeleteVertexArrays.

The names returned in *arrays* are marked as used, for the purposes of `glGenVertexArrays` only, but they acquire state and type only when they are first bound.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

Associated Gets

glIsVertexArray

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glGenVertexArrays	-	✔

See Also

glBindVertexArray, glDeleteVertexArrays

Copyright

Name

glGenerateMipmap — generate mipmaps for a specified texture target

C Specification

```
void glGenerateMipmap (target);

GLenum target;
```

Parameters

`target`	Specifies the target to which the texture whose mimaps to generate is bound. *`target`* must be `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_2D_ARRAY` or `GL_TEXTURE_CUBE_MAP`.

Description

`glGenerateMipmap` generates mipmaps for the texture attached to *`target`* of the active texture unit. For cube map textures, a `GL_INVALID_OPERATION` error is generated if the texture attached to *`target`* is not cube complete.

Mipmap generation replaces texel array levels through with arrays derived from the array, regardless of their previous contents. All other mimap arrays, including the array, are left unchanged by this computation.

The internal formats of the derived mipmap arrays all match those of the array. The contents of the derived arrays are computed by repeated, filtered reduction of the array. For two-dimensional texture arrays, each layer is filtered independently.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not one of the accepted texture targets.

`GL_INVALID_OPERATION` is generated if *`target`* is `GL_TEXTURE_CUBE_MAP` and the texture bound to the `GL_TEXTURE_CUBE_MAP` target of the active texture unit is not cube complete.

`GL_INVALID_OPERATION` is generated if the array is stored in a compressed internal format.

`GL_INVALID_OPERATION` is generated if the array was not specified with an unsized internal format or a sized internal format that is both color-renderable and texture-filterable.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGenerateMipmap	✔	✔

See Also

glTexImage2D, glBindTexture, glGenTextures

Copyright

Name

glGet — return the value or values of a selected parameter

C Specification

```
void glGetBooleanv (pname, data);

GLenum pname;
GLboolean * data;

void glGetFloatv (pname, data);

GLenum pname;
GLfloat * data;

void glGetIntegerv (pname, data);

GLenum pname;
GLint * data;

void glGetInteger64v (pname, data);

GLenum pname;
GLint64 * data;

void glGetIntegeri_v (target, index, data);

GLenum target;
GLuint index;
GLint * data;

void glGetInteger64i_v (target, index, data);

GLenum target;
GLuint index;
GLint64 * data;
```

Parameters

pname	Specifies the parameter value to be returned. The symbolic constants in the list below are accepted.
target	Specifies the parameter value to be returned for indexed versions of `glGet`. The symbolic constants in the list below are accepted.
index	Specifies the index of the particular element being queried.
data	Returns the value or values of the specified parameter.

Description

These commands return values for simple state variables in GL. *pname* is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type than the state variable value being requested. If `glGetBooleanv` is called, a floating-point (or integer) value is converted to `GL_FALSE` if and only if it is 0.0 (or 0). Otherwise, it is converted to `GL_TRUE`. If `glGetIntegerv` is called, boolean values are returned as `GL_TRUE` or `GL_FALSE`, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value and to the most negative representable integer value. If `glGetFloatv` is called, boolean values are returned as `GL_TRUE` or `GL_FALSE`, and integer values are converted to floating-point values.

The following symbolic constants are accepted by *pname*:

`GL_ACTIVE_TEXTURE`	*params* returns a single value indicating the active multitexture unit. The initial value is `GL_TEXTURE0`. See glActiveTexture.
`GL_ALIASED_LINE_WIDTH_RANGE`	*params* returns a pair of values indicating the range of widths supported for aliased lines. See glLineWidth.
`GL_ALIASED_POIN-T_SIZE_RANGE`	*params* returns two values: the smallest and largest supported sizes for points. The smallest size must be at most 1, and the largest size must be at least 1.
`GL_ALPHA_BITS`	*params* returns one value, the number of alpha bitplanes in the color buffer of the currently bound draw framebuffer. This is de#ned only if all color attachments of the draw framebuffer have identical formats, in which case the number of alpha bits of color attachment zero are returned.
`GL_ARRAY_BUFFER_BINDING`	*params* returns a single value, the name of the buffer object currently bound to the target `GL_ARRAY_BUFFER`. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.
`GL_BLEND`	*params* returns a single boolean value indicating whether blending is enabled. The initial value is `GL_FALSE`. See glBlendFunc.
`GL_BLEND_COLOR`	*params* returns four values, the red, green, blue, and alpha values which are the components of the blend color. See glBlendColor.
`GL_BLEND_DST_ALPHA`	*params* returns one value, the symbolic constant identifying the alpha destination blend function. The initial value is `GL_ZERO`. See glBlendFunc and glBlendFuncSeparate.
`GL_BLEND_DST_RGB`	*params* returns one value, the symbolic constant identifying the RGB destination blend function. The initial value is `GL_ZERO`. See glBlendFunc and glBlendFuncSeparate.
`GL_BLEND_EQUATION_ALPHA`	*params* returns one value, a symbolic constant indicating whether the Alpha blend equation is `GL_FUNC_ADD`, `GL_FUNC_SUB-TRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN` or `GL_MAX`. See glBlendEquationSeparate.
`GL_BLEND_EQUATION_RGB`	*params* returns one value, a symbolic constant indicating whether the RGB blend equation is `GL_FUNC_ADD`, `GL_FUNC_SUB-TRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN` or `GL_MAX`. See glBlendEquationSeparate.

`GL_BLEND_SRC_ALPHA`	*params* returns one value, the symbolic constant identifying the alpha source blend function. The initial value is `GL_ONE`. See glBlendFunc and glBlendFuncSeparate.
`GL_BLEND_SRC_RGB`	*params* returns one value, the symbolic constant identifying the RGB source blend function. The initial value is `GL_ONE`. See glBlendFunc and glBlendFuncSeparate.
`GL_BLUE_BITS`	*params* returns one value, the number of blue bitplanes in the color buffer of the currently bound draw framebuffer. This is de#ned only if all color attachments of the draw framebuffer have identical formats, in which case the number of blue bits of color attachment zero are returned.
`GL_COLOR_CLEAR_VALUE`	*params* returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and returns the most negative representable integer value. The initial value is (0, 0, 0, 0). See glClearColor.
`GL_COLOR_WRITEMASK`	*params* returns four boolean values: the red, green, blue, and alpha write enables for the color buffers. The initial value is (`GL_TRUE`, `GL_TRUE`, `GL_TRUE`, `GL_TRUE`). See glColorMask.
`GL_COMPRESSED_TEXTURE_FORMATS`	*params* returns a list of symbolic constants of length `GL_NUM_COMPRESSED_TEXTURE_FORMATS` indicating which compressed texture formats are available. See glCompressedTexImage2D.
`GL_COPY_READ_BUFFER_BINDING`	*params* returns a single value, the name of the buffer object currently bound to the target `GL_COPY_READ_BUFFER`. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.
`GL_COPY_WRITE_BUFFER_BINDING`	*params* returns a single value, the name of the buffer object currently bound to the target `GL_COPY_WRITE_BUFFER`. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.
`GL_CULL_FACE`	*params* returns a single boolean value indicating whether polygon culling is enabled. The initial value is `GL_FALSE`. See glCullFace.
`GL_CULL_FACE_MODE`	*params* returns a single value indicating the mode of polygon culling. The initial value is `GL_BACK`. See glCullFace.
`GL_CURRENT_PROGRAM`	*params* returns one value, the name of the program object that is currently active, or 0 if no program object is active. See glUseProgram.
`GL_DEPTH_BITS`	*params* returns one value, the number of bitplanes in the depth buffer of the currently bound framebuffer.
`GL_DEPTH_CLEAR_VALUE`	*params* returns one value, the value that is used to clear the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the

	most positive representable integer value, and returns the most negative representable integer value. The initial value is 1. See glClearDepthf.
`GL_DEPTH_FUNC`	*params* returns one value, the symbolic constant that indicates the depth comparison function. The initial value is `GL_LESS`. See glDepthFunc.
`GL_DEPTH_RANGE`	*params* returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and returns the most negative representable integer value. The initial value is (0, 1). See glDepthRangef.
`GL_DEPTH_TEST`	*params* returns a single boolean value indicating whether depth testing of fragments is enabled. The initial value is `GL_FALSE`. See glDepthFunc and glDepthRangef.
`GL_DEPTH_WRITEMASK`	*params* returns a single boolean value indicating if the depth buffer is enabled for writing. The initial value is `GL_TRUE`. See glDepthMask.
`GL_DITHER`	*params* returns a single boolean value indicating whether dithering of fragment colors and indices is enabled. The initial value is `GL_TRUE`.
`GL_DRAW_BUFFER`*i*	*params* returns one value, a symbolic constant indicating which buffers are being drawn to by the corresponding output color. See glDrawBuffers. The initial value of `GL_DRAW_BUFFER0` is `GL_BACK`. The initial values of draw buffers for all other output colors is `GL_NONE`.
`GL_DRAW_FRAME-BUFFER_BINDING`	*params* returns one value, the name of the framebuffer object currently bound to the `GL_DRAW_FRAMEBUFFER` target. If the default framebuffer is bound, this value will be zero. The initial value is zero. See glBindFramebuffer.
`GL_ELEMENT_AR-RAY_BUFFER_BINDING`	*params* returns a single value, the name of the buffer object currently bound to the target `GL_ELEMENT_ARRAY_BUFFER`. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.
`GL_FRAGMENT_SHADER_DERI-VATIVE_HINT`	*params* returns one value, a symbolic constant indicating the mode of the derivative accuracy hint for fragment shaders. The initial value is `GL_DONT_CARE`. See glHint.
`GL_FRONT_FACE`	*params* returns a single value indicating the winding order of polygon front faces. The initial value is `GL_CCW`. See glFrontFace.
`GL_GENERATE_MIPMAP_HINT`	*params* returns one value, a symbolic constant indicating the mode of the generate mipmap quality hint. The initial value is `GL_DONT_CARE`. See glHint.
`GL_GREEN_BITS`	*params* returns one value, the number of green bitplanes in the color buffer of the currently bound draw framebuffer. This is de#ned

	only if all color attachments of the draw framebuffer have identical formats, in which case the number of green bits of color attachment zero are returned.
`GL_IMPLEMENTATION_COLOR_READ_FORMAT`	*`params`* returns one value, the format chosen by the implementation in which pixels may be read from the color buffer of the currently bound framebuffer in conjunction with `GL_IMPLEMENTATION_COLOR_READ_TYPE`. See glReadPixels.
`GL_IMPLEMENTATION_COLOR_READ_TYPE`	*`params`* returns one value, the type chosen by the implementation with which pixels may be read from the color buffer of the currently bound framebuffer in conjunction with `GL_IMPLEMENTATION_COLOR_READ_FORMAT`. See glReadPixels.
`GL_LINE_WIDTH`	*`params`* returns one value, the line width as specified with glLineWidth. The initial value is 1.
`GL_MAJOR_VERSION`	*`params`* returns one value, the major version number of the OpenGL ES API supported by the current context. This must be 3.
`GL_MAX_3D_TEXTURE_SIZE`	*`params`* returns one value, a rough estimate of the largest 3D texture that the GL can handle. The value must be at least 256. See glTexImage3D.
`GL_MAX_ARRAY_TEXTURE_LAYERS`	*`params`* returns one value. The value indicates the maximum number of layers allowed in an array texture, and must be at least 256. See glTexImage2D.
`GL_MAX_COLOR_ATTACHMENTS`	*`params`* returns one value, the maximum number of color attachment points in a framebuffer object. The value must be at least 4. See glFramebufferRenderbuffer and glFramebufferTexture2D.
`GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS`	*`params`* returns one value, the number of words for fragment shader uniform variables in all uniform blocks (including default). The value must be at least `GL_MAX_FRAGMENT_UNIFORM_COMPONENTS + GL_MAX_UNIFORM_BLOCK_SIZE * GL_MAX_FRAGMENT_UNIFORM_BLOCKS` / 4. See glUniform.
`GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`	*`params`* returns one value, the maximum supported texture image units that can be used to access texture maps from the vertex shader and the fragment processor combined. If both the vertex shader and the fragment processing stage access the same texture image unit, then that counts as using two texture image units against this limit. The value must be at least 32. See glActiveTexture.
`GL_MAX_COMBINED_UNIFORM_BLOCKS`	*`params`* returns one value, the maximum number of uniform blocks per program. The value must be at least 24. See glUniformBlockBinding.
`GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS`	*`params`* returns one value, the number of words for vertex shader uniform variables in all uniform blocks (including default). The value must be at least . `GL_MAX_VERTEX_UNIFORM_COMPONENTS` + `GL_MAX_UNIFORM_BLOCK_SIZE` * `GL_MAX_VERTEX_UNIFORM_BLOCKS` / 4. See glUniform.

`GL_MAX_CUBE_MAP_TEXTURE_SIZE`	*params* returns one value. The value gives a rough estimate of the largest cube-map texture that the GL can handle. The value must be at least 2048. See glTexImage2D.
`GL_MAX_DRAW_BUFFERS`	*params* returns one value, the maximum number of simultaneous outputs that may be written in a fragment shader. The value must be at least 4. See glDrawBuffers.
`GL_MAX_ELEMENT_INDEX`	*params* returns one value, the maximum index supported by the implementation. The value must be at least .
`GL_MAX_ELEMENTS_INDICES`	*params* returns one value, the recommended maximum number of vertex array indices. See glDrawRangeElements.
`GL_MAX_ELEMENTS_VERTICES`	*params* returns one value, the recommended maximum number of vertex array vertices. See glDrawRangeElements.
`GL_MAX_FRAGMENT_INPUT_COMPONENTS`	*params* returns one value, the maximum number of components of the inputs read by the fragment shader, which must be at least 60.
`GL_MAX_FRAGMENT_UNIFORM_BLOCKS`	*params* returns one value, the maximum number of uniform blocks per fragment shader. The value must be at least 12. See glUniformBlockBinding.
`GL_MAX_FRAGMENT_UNIFORM_COMPONENTS`	*params* returns one value, the maximum number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader. The value must be at least 896. See glUniform.
`GL_MAX_FRAGMENT_UNIFORM_VECTORS`	*params* returns one value, the maximum number of vector floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader. The value must be at least 224. See glUniform.
`GL_MAX_PROGRAM_TEXEL_OFFSET`	*params* returns one value, the maximum texel offset allowed in a texture lookup, which must be at least 7.
`GL_MAX_RENDERBUFFER_SIZE`	*params* returns one value. The value indicates the maximum supported size for renderbuffers and must be at least 2048. See glFramebufferRenderbuffer.
`GL_MAX_SAMPLES`	*params* returns one value. The value indicates the maximum supported number of samples for multisampling. The value must be at least 4. See glGetInternalformativ.
`GL_MAX_SERVER_WAIT_TIMEOUT`	*params* returns one value, the maximum glWaitSync timeout interval.
`GL_MAX_TEXTURE_IMAGE_UNITS`	*params* returns one value, the maximum supported texture image units that can be used to access texture maps from the fragment shader. The value must be at least 16. See glActiveTexture.
`GL_MAX_TEXTURE_LOD_BIAS`	*params* returns one value, the maximum, absolute value of the texture level-of-detail bias. The value must be at least 2.0.
`GL_MAX_TEXTURE_SIZE`	*params* returns one value. The value gives a rough estimate of the largest texture that the GL can handle. The value must be at least 2048. See glTexImage2D.

`GL_MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS`	*params* returns one value, the maximum number of components which can be written to a single transform feedback buffer in interleaved mode. The value must be at least 64. See glTransformFeedbackVaryings.
`GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`	*params* returns one value, the maximum separate attributes or outputs which can be captured in separate transform feedback mode. The value must be at least 4. See glTransformFeedbackVaryings.
`GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS`	*params* returns one value, the maximum number of components which can be written per attribute or output in separate transform feedback mode. The value must be at least 4. See glTransformFeedbackVaryings.
`GL_MAX_UNIFORM_BLOCK_SIZE`	*params* returns one value, the maximum size in basic machine units of a uniform block. The value must be at least 16384. See glUniformBlockBinding.
`GL_MAX_UNIFORM_BUFFER_BINDINGS`	*params* returns one value, the maximum number of uniform buffer binding points on the context, which must be at least 24.
`GL_MAX_VARYING_COMPONENTS`	*params* returns one value, the number components for varying variables, which must be at least 60.
`GL_MAX_VARYING_VECTORS`	*params* returns one value, the maximum number of interpolators available for processing varying variables used by vertex and fragment shaders. This value represents the number of vector values that can be interpolated; varying variables declared as matrices and arrays will consume multiple interpolators. The value must be at least 15.
`GL_MAX_VERTEX_ATTRIBS`	*params* returns one value, the maximum number of 4-component generic vertex attributes accessible to a vertex shader. The value must be at least 16. See glVertexAttrib.
`GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS`	*params* returns one value, the maximum supported texture image units that can be used to access texture maps from the vertex shader. The value may be at least 16. See glActiveTexture.
`GL_MAX_VERTEX_OUTPUT_COMPONENTS`	*params* returns one value, the maximum number of components of output written by a vertex shader, which must be at least 64.
`GL_MAX_VERTEX_UNIFORM_BLOCKS`	*params* returns one value, the maximum number of uniform blocks per vertex shader. The value must be at least 12. See glUniformBlockBinding.
`GL_MAX_VERTEX_UNIFORM_COMPONENTS`	*params* returns one value, the maximum number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a vertex shader. The value must be at least 1024. See glUniform.
`GL_MAX_VERTEX_UNIFORM_VECTORS`	

	params returns one value, the maximum number of vector floating-point, integer, or boolean values that can be held in uniform variable storage for a vertex shader. The value must be at least 256. See glUniform.
`GL_MAX_VIEWPORT_DIMS`	*params* returns two values: the maximum supported width and height of the viewport. These must be at least as large as the visible dimensions of the display being rendered to. See glViewport.
`GL_MIN_PROGRAM_TEXEL_OFFSET`	*params* returns one value, the minimum texel offset allowed in a texture lookup, which must be at most -8.
`GL_MINOR_VERSION`	*params* returns one value, the minor version number of the OpenGL ES API supported by the current context.
`GL_NUM_COMPRESSED_TEXTURE_FORMATS`	*params* returns a single integer value indicating the number of available compressed texture formats. The minimum value is 10. See glCompressedTexImage2D.
`GL_NUM_EXTENSIONS`	*params* returns one value, the number of extensions supported by the GL implementation for the current context. See glGetString.
`GL_NUM_PROGRAM_BINARY_FORMATS`	*params* returns a single integer value indicating the number of available program binary formats. The minimum value is 0. See glProgramBinary.
`GL_NUM_SHADER_BINARY_FORMATS`	*params* returns a single integer value indicating the number of available shader binary formats. The minimum value is 0. See glShaderBinary.
`GL_PACK_ALIGNMENT`	*params* returns one value, the byte alignment used for writing pixel data to memory. The initial value is 4. See glPixelStorei.
`GL_PACK_ROW_LENGTH`	*params* returns one value, the row length used for writing pixel data to memory. The initial value is 0. See glPixelStorei.
`GL_PACK_SKIP_PIXELS`	*params* returns one value, the number of pixel locations skipped before the first pixel is written into memory. The initial value is 0. See glPixelStorei.
`GL_PACK_SKIP_ROWS`	*params* returns one value, the number of rows of pixel locations skipped before the first pixel is written into memory. The initial value is 0. See glPixelStorei.
`GL_PIXEL_PACK_BUFFER_BINDING`	*params* returns a single value, the name of the buffer object currently bound to the target `GL_PIXEL_PACK_BUFFER`. If no

	buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.
`GL_PIXEL_UN-PACK_BUFFER_BINDING`	*`params`* returns a single value, the name of the buffer object currently bound to the target `GL_PIXEL_UNPACK_BUFFER`. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.
`GL_POLYGON_OFFSET_FACTOR`	*`params`* returns one value, the scaling factor used to determine the variable offset that is added to the depth value of each fragment generated when a polygon is rasterized. The initial value is 0. See glPolygonOffset.
`GL_POLYGON_OFFSET_FILL`	*`params`* returns a single boolean value indicating whether polygon offset is enabled for polygons. The initial value is `GL_FALSE`. See glPolygonOffset.
`GL_POLYGON_OFFSET_UNITS`	*`params`* returns one value. This value is multiplied by an implementation-specific value and then added to the depth value of each fragment generated when a polygon is rasterized. The initial value is 0. See glPolygonOffset.
`GL_PRIMITIVE_RES-TART_FIXED_INDEX`	*`params`* returns a single boolean value indicating whether primitive restart with a fixed index is enabled. The initial value is `GL_FALSE`.
`GL_PROGRAM_BINARY_FOR-MATS`	*`params`* returns a list of symbolic constants of length `GL_NUM_PROGRAM_BINARY_FORMATS` indicating which program binary formats are available. See glProgramBinary.
`GL_RASTERIZER_DISCARD`	*`params`* returns one value, a single boolean value indicating whether primitives are discarded immediately before the rasterization stage, but after the optional transform feedback stage. See glEnable.
`GL_READ_BUFFER`	*`params`* returns one value, a symbolic constant indicating which color buffer is selected for reading. The initial value is `GL_BACK`. See glReadPixels.
`GL_READ_FRAME-BUFFER_BINDING`	*`params`* returns one value, the name of the framebuffer object currently bound to the `GL_READ_FRAMEBUFFER` target. If the default framebuffer is bound, this value will be zero. The initial value is zero. See glBindFramebuffer.
`GL_RED_BITS`	*`params`* returns one value, the number of red bitplanes in the color buffer of the currently bound draw framebuffer. This is de#ned only if all color attachments of the draw framebuffer have identical formats, in which case the number of red bits of color attachment zero are returned.
`GL_RENDERBUFFER_BINDING`	*`params`* returns a single value, the name of the renderbuffer object currently bound to the target `GL_RENDERBUFFER`. If no renderbuffer object is bound to this target, 0 is returned. The initial value is 0. See glBindRenderbuffer.

`GL_SAMPLE_ALPHA_TO_COVERAGE`	*`params`* returns a single boolean value indicating whether modification of sample coverage based on alpha is enabled. The initial value is `GL_FALSE`. See glSampleCoverage.
`GL_SAMPLE_BUFFERS`	*`params`* returns a single integer value indicating the number of sample buffers associated with the framebuffer. See glSampleCoverage.
`GL_SAMPLE_COVERAGE`	*`params`* returns a single boolean value indicating whether modification of sample coverage based on the value specified by glSampleCoverage is enabled. The initial value is `GL_FALSE`.
`GL_SAMPLE_COVERAGE_INVERT`	*`params`* returns a single boolean value indicating if the temporary coverage value should be inverted. See glSampleCoverage.
`GL_SAMPLE_COVERAGE_VALUE`	*`params`* returns a single positive floating-point value indicating the current sample coverage value. See glSampleCoverage.
`GL_SAMPLER_BINDING`	*`params`* returns a single value, the name of the sampler object currently bound to the active texture unit. The initial value is 0. See glBindSampler.
`GL_SAMPLES`	*`params`* returns a single integer value indicating the coverage mask size. See glSampleCoverage.
`GL_SCISSOR_BOX`	*`params`* returns four values: the and window coordinates of the scissor box, followed by its width and height. Initially the and window coordinates are both 0 and the width and height are set to the size of the window. See glScissor.
`GL_SCISSOR_TEST`	*`params`* returns a single boolean value indicating whether scissoring is enabled. The initial value is `GL_FALSE`. See glScissor.
`GL_SHADER_BINARY_FORMATS`	*`params`* returns a list of symbolic constants of length `GL_NUM_SHADER_BINARY_FORMATS` indicating which shader binary formats are available. See glShaderBinary.
`GL_SHADER_COMPILER`	*`params`* returns a single boolean value indicating whether a shader compiler is supported. This value is always `GL_TRUE`. See glCompileShader.
`GL_STENCIL_BACK_FAIL`	*`params`* returns one value, a symbolic constant indicating what action is taken for back-facing polygons when the stencil test fails. The initial value is `GL_KEEP`. See glStencilOpSeparate.
`GL_STENCIL_BACK_FUNC`	

	params returns one value, a symbolic constant indicating what function is used for back-facing polygons to compare the stencil reference value with the stencil buffer value. The initial value is `GL_ALWAYS`. See glStencilFuncSeparate.
`GL_STENCIL_BACK_PASS_DEPTH_FAIL`	*params* returns one value, a symbolic constant indicating what action is taken for back-facing polygons when the stencil test passes, but the depth test fails. The initial value is `GL_KEEP`. See glStencilOpSeparate.
`GL_STENCIL_BACK_PASS_DEPTH_PASS`	*params* returns one value, a symbolic constant indicating what action is taken for back-facing polygons when the stencil test passes and the depth test passes. The initial value is `GL_KEEP`. See glStencilOpSeparate.
`GL_STENCIL_BACK_REF`	*params* returns one value, the reference value that is compared with the contents of the stencil buffer for back-facing polygons. The initial value is 0. See glStencilFuncSeparate.
`GL_STENCIL_BACK_VALUE_MASK`	*params* returns one value, the mask that is used for back-facing polygons to mask both the stencil reference value and the stencil buffer value before they are compared. The initial value is all 1's. See glStencilFuncSeparate.
`GL_STENCIL_BACK_WRITEMASK`	*params* returns one value, the mask that controls writing of the stencil bitplanes for back-facing polygons. The initial value is all 1's. See glStencilMaskSeparate.
`GL_STENCIL_BITS`	*params* returns one value, the number of bitplanes in the stencil buffer of the currently bound framebuffer.
`GL_STENCIL_CLEAR_VALUE`	*params* returns one value, the index to which the stencil bitplanes are cleared. The initial value is 0. See glClearStencil.
`GL_STENCIL_FAIL`	*params* returns one value, a symbolic constant indicating what action is taken when the stencil test fails. The initial value is `GL_KEEP`. See glStencilOp. This stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilOpSeparate.
`GL_STENCIL_FUNC`	*params* returns one value, a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. The initial value is `GL_ALWAYS`. See glStencilFunc. This stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilFuncSeparate.
`GL_STENCIL_PASS_DEPTH_FAIL`	

params returns one value, a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. The initial value is `GL_KEEP`. See glStencilOp. This stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilOpSeparate.

`GL_S-TENCIL_PASS_DEPTH_PASS`
: *params* returns one value, a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. The initial value is `GL_KEEP`. See glStencilOp. This stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilOpSeparate.

`GL_STENCIL_REF`
: *params* returns one value, the reference value that is compared with the contents of the stencil buffer. The initial value is 0. See glS-tencilFunc. This stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilFuncSeparate.

`GL_STENCIL_TEST`
: *params* returns a single boolean value indicating whether stencil testing of fragments is enabled. The initial value is `GL_FALSE`. See glStencilFunc and glStencilOp.

`GL_STENCIL_VALUE_MASK`
: *params* returns one value, the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. The initial value is all 1's. See glStencilFunc. This stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencil-FuncSeparate.

`GL_STENCIL_WRITEMASK`
: *params* returns one value, the mask that controls writing of the stencil bitplanes. The initial value is all 1's. See glStencilMask. This stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilMaskSeparate.

`GL_SUBPIXEL_BITS`
: *params* returns one value, an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates. The value must be at least 4.

`GL_TEXTURE_BINDING_2D`
: *params* returns a single value, the name of the texture currently bound to the target `GL_TEXTURE_2D`. The initial value is 0. See glBindTexture.

`GL_TEX-TURE_BINDING_2D_ARRAY`
: *params* returns a single value, the name of the texture currently bound to the target `GL_TEXTURE_2D_ARRAY`. The initial value is 0. See glBindTexture.

`GL_TEXTURE_BINDING_3D`

	params returns a single value, the name of the texture currently bound to the target `GL_TEXTURE_3D`. The initial value is 0. See glBindTexture.
`GL_TEXTURE_BINDING_CUBE_MAP`	*params* returns a single value, the name of the texture currently bound to the target `GL_TEXTURE_CUBE_MAP`. The initial value is 0. See glBindTexture.
`GL_TRANSFORM_FEEDBACK_BINDING`	*params* returns a single value, the name of the transform feedback object currently bound to the `GL_TRANSFORM_FEEDBACK` target. If no transform feedback object is bound to this target, 0 is returned. The initial value is 0. See glBindTransformFeedback.
`GL_TRANSFORM_FEEDBACK_ACTIVE`	*params* returns a single boolean value indicating if the currently bound transform feedback object is active. See glBeginTransformFeedback and glResumeTransformFeedback.
`GL_TRANSFORM_FEEDBACK_BUFFER_BINDING`	When used with non-indexed variants of `glGet` (such as `glGetIntegerv`), *params* returns a single value, the name of the buffer object currently bound to the target `GL_TRANSFORM_FEEDBACK_BUFFER`. If no buffer object is bound to this target, 0 is returned. When used with indexed variants of `glGet` (such as `glGetIntegeri_v`), *params* returns a single value, the name of the buffer object bound to the indexed transform feedback attribute stream. The initial value is 0 for all targets. See glBindBuffer, glBindBufferBase, and glBindBufferRange.
`GL_TRANSFORM_FEEDBACK_PAUSED`	*params* returns a single boolean value indicating if the currently bound transform feedback object is paused. See glPauseTransformFeedback.
`GL_TRANSFORM_FEEDBACK_BUFFER_SIZE`	When used with indexed variants of `glGet` (such as `glGetInteger64i_v`), *params* returns a single value, the size of the binding range for each transform feedback attribute stream. The initial value is 0 for all streams. See glBindBufferRange.
`GL_TRANSFORM_FEEDBACK_BUFFER_START`	When used with indexed variants of `glGet` (such as `glGetInteger64i_v`), *params* returns a single value, the start offset of the binding range for each transform feedback attribute stream. The initial value is 0 for all streams. See glBindBufferRange.
`GL_UNIFORM_BUFFER_BINDING`	When used with non-indexed variants of `glGet` (such as `glGetIntegerv`), *params* returns a single value, the name of the buffer object currently bound to the target `GL_UNIFORM_BUFFER`. If no buffer object is bound to this target, 0 is returned. When used with indexed variants of `glGet` (such as `glGetIntegeri_v`), *params* returns a single value, the name of the buffer object bound to the indexed uniform buffer binding point. The ini-

	tial value is 0 for all targets. See glBindBuffer, glBindBufferBase, and glBindBufferRange.
`GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT`	*`params`* returns a single value, the minimum required alignment for uniform buffer sizes and offset. The initial value is 1. See glUniformBlockBinding.
`GL_UNIFORM_BUFFER_SIZE`	When used with indexed variants of `glGet` (such as `glGetInteger64i_v`), *`params`* returns a single value, the size of the binding range for each indexed uniform buffer binding. The initial value is 0 for all bindings. See glBindBufferRange.
`GL_UNIFORM_BUFFER_START`	When used with indexed variants of `glGet` (such as `glGetInteger64i_v`), *`params`* returns a single value, the start offset of the binding range for each indexed uniform buffer binding. The initial value is 0 for all bindings. See glBindBufferRange.
`GL_UNPACK_ALIGNMENT`	*`params`* returns one value, the byte alignment used for reading pixel data from memory. The initial value is 4. See glPixelStorei.
`GL_UNPACK_IMAGE_HEIGHT`	*`params`* returns one value, the image height used for reading pixel data from memory. The initial is 0. See glPixelStorei.
`GL_UNPACK_ROW_LENGTH`	*`params`* returns one value, the row length used for reading pixel data from memory. The initial value is 0. See glPixelStorei.
`GL_UNPACK_SKIP_IMAGES`	*`params`* returns one value, the number of pixel images skipped before the first pixel is read from memory. The initial value is 0. See glPixelStorei.
`GL_UNPACK_SKIP_PIXELS`	*`params`* returns one value, the number of pixel locations skipped before the first pixel is read from memory. The initial value is 0. See glPixelStorei.
`GL_UNPACK_SKIP_ROWS`	*`params`* returns one value, the number of rows of pixel locations skipped before the first pixel is read from memory. The initial value is 0. See glPixelStorei.
`GL_VERTEX_ARRAY_BINDING`	*`params`* returns a single value, the name of the vertex array object currently bound. If no vertex array object is bound, 0 is returned. The initial value is 0. See glBindVertexArray.
`GL_VIEWPORT`	*`params`* returns four values: the and window coordinates of the viewport, followed by its width and height. Initially the and window coordinates are both set to 0, and the width and height are set to the width and height of the window into which the GL will do its rendering. See glViewport.

Many of the boolean parameters can also be queried more easily using glIsEnabled.

Notes

The following parameters return the associated value for the active texture unit: `GL_TEXTURE_2D`, `GL_TEXTURE_BINDING_2D`, `GL_TEXTURE_3D` and `GL_TEXTURE_BINDING_3D`.

Errors

`GL_INVALID_ENUM` is generated if *`pname`* is not an accepted value.

`GL_INVALID_VALUE` is generated on either `glGetIntegeri_v`, or `glGetInteger64i_v` if *`index`* is outside of the valid range for the indexed state *`target`*.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glGetBooleanv`	✔	✔
`glGetFloatv`	✔	✔
`glGetIntegerv`	✔	✔
`glGetInteger64v`	-	✔
`glGetIntegeri_v`	-	✔
`glGetInteger64i_v`	-	✔

See Also

glGetActiveUniform, glGetAttachedShaders, glGetAttribLocation, glGetBufferParameter, glGetBufferPointerv, glGetError, glGetProgramiv, glGetProgramInfoLog, glGetQueryiv, glGetQueryObjectuiv, glGetShaderiv, glGetShaderInfoLog, glGetShaderSource, glGetString, glGetTexParameter, glGetUniform, glGetUniformLocation, glGetVertexAttrib, glGetVertexAttribPointerv, glIsEnabled

Copyright

Name

glGetActiveAttrib — Returns information about an active attribute variable for the specified program object

C Specification

```
void glGetActiveAttrib (program, index, bufSize, length, size, type,
name);

GLuint program;
GLuint index;
GLsizei bufSize;
GLsizei *length;
GLint *size;
GLenum *type;
GLchar *name;
```

Parameters

program	Specifies the program object to be queried.
index	Specifies the index of the attribute variable to be queried.
bufSize	Specifies the maximum number of characters OpenGL is allowed to write in the character buffer indicated by *name*.
length	Returns the number of characters actually written by OpenGL in the string indicated by *name* (excluding the null terminator) if a value other than `NULL` is passed.
size	Returns the size of the attribute variable.
type	Returns the data type of the attribute variable.
name	Returns a null terminated string containing the name of the attribute variable.

Description

`glGetActiveAttrib` returns information about an active attribute variable in the program object specified by *program*. The number of active attributes can be obtained by calling glGetProgramiv with the value `GL_ACTIVE_ATTRIBUTES`. A value of zero for *index* selects the first active attribute variable. Permissible values for *index* range from zero to the number of active attribute variables minus one.

Attribute variables have arbitrary names and obtain their values through numbered generic vertex attributes. An attribute variable is considered active if it is determined during the link operation that it may be accessed during program execution. Therefore, *program* should have previously been the target of a call to glLinkProgram, but it is not necessary for it to have been linked successfully.

The size of the character buffer required to store the longest attribute variable name in *program* can be obtained by calling glGetProgramiv with the value `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`. This value should be used to allocate a buffer of sufficient size to store the returned attribute name. The size of this character buffer is passed in *bufSize*, and a pointer to this character buffer is passed in *name*.

`glGetActiveAttrib` returns the name of the attribute variable indicated by *index*, storing it in the character buffer specified by *name*. The string returned will be null terminated. The actual number of characters written into this buffer is returned in *length*, and this count does not include the null termination character. If the length of the returned string is not required, a value of NULL can be passed in the *length* argument.

The *type* argument will return a pointer to the attribute variable's data type. The symbolic constants GL_FLOAT, GL_FLOAT_VEC2, GL_FLOAT_VEC3, GL_FLOAT_VEC4, GL_FLOAT_MAT2, GL_FLOAT_MAT3, GL_FLOAT_MAT4, GL_FLOAT_MAT2x3, GL_FLOAT_MAT2x4, GL_FLOAT_MAT3x2, GL_FLOAT_MAT3x4, GL_FLOAT_MAT4x2, GL_FLOAT_MAT4x3, GL_INT, GL_INT_VEC2, GL_INT_VEC3, GL_INT_VEC4, GL_UNSIGNED_INT, GL_UNSIGNED_INT_VEC2, GL_UNSIGNED_INT_VEC3, or GL_UNSIGNED_INT_VEC4 may be returned. The *size* argument will return the size of the attribute, in units of the type returned in *type*.

This function will return as much information as it can about the specified active attribute variable. If no information is available, *length* will be 0, and *name* will be an empty string. This situation could occur if this function is called after a link operation that failed. If an error occurs, the return values *length*, *size*, *type*, and *name* will be unmodified.

Errors

GL_INVALID_VALUE is generated if *program* is not a value generated by OpenGL.

GL_INVALID_OPERATION is generated if *program* is not a program object.

GL_INVALID_VALUE is generated if *index* is greater than or equal to the number of active attribute variables in *program*.

GL_INVALID_VALUE is generated if *bufSize* is less than 0.

Associated Gets

glGet with argument GL_MAX_VERTEX_ATTRIBS.

glGetProgramiv with argument GL_ACTIVE_ATTRIBUTES or GL_ACTIVE_ATTRIBUTE_MAX_LENGTH.

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetActiveAttrib	✔	✔

See Also

glBindAttribLocation, glLinkProgram, glVertexAttrib, glVertexAttribPointer

Copyright

Name

glGetActiveUniform — Returns information about an active uniform variable for the specified program object

C Specification

```
void glGetActiveUniform (program, index, bufSize, length, size, type,
name);
```

```
GLuint program;
GLuint index;
GLsizei bufSize;
GLsizei *length;
GLint *size;
GLenum *type;
GLchar *name;
```

Parameters

program	Specifies the program object to be queried.
index	Specifies the index of the uniform variable to be queried.
bufSize	Specifies the maximum number of characters OpenGL is allowed to write in the character buffer indicated by *name*.
length	Returns the number of characters actually written by OpenGL in the string indicated by *name* (excluding the null terminator) if a value other than `NULL` is passed.
size	Returns the size of the uniform variable.
type	Returns the data type of the uniform variable.
name	Returns a null terminated string containing the name of the uniform variable.

Description

`glGetActiveUniform` returns information about an active uniform variable in the program object specified by *program*. The number of active uniform variables can be obtained by calling glGetProgramiv with the value `GL_ACTIVE_UNIFORMS`. A value of zero for *index* selects the first active uniform variable. Permissible values for *index* range from zero to the number of active uniform variables minus one.

Shaders may use either built-in uniform variables, user-defined uniform variables, or both. Built-in uniform variables have a prefix of "gl_" and reference existing OpenGL state or values derived from such state (e.g., *gl_DepthRange*, see the OpenGL Shading Language specification for a complete list.) User-defined uniform variables have arbitrary names and obtain their values from the application through calls to glUniform. A uniform variable (either built-in or user-defined) is considered active if it is determined during the link operation that it may be accessed during program execution. Therefore, *program* should have previously been the target of a call to glLinkProgram, but it is not necessary for it to have been linked successfully.

The size of the character buffer required to store the longest uniform variable name in *program* can be obtained by calling glGetProgramiv with the value `GL_ACTIVE_UNIFORM_MAX_LENGTH`. This value should be used to allocate a buffer of sufficient size to store the returned uniform variable name. The size of this character buffer is passed in *`bufSize`*, and a pointer to this character buffer is passed in *`name.`*

`glGetActiveUniform` returns the name of the uniform variable indicated by *`index`*, storing it in the character buffer specified by *`name`*. The string returned will be null terminated. The actual number of characters written into this buffer is returned in *`length`*, and this count does not include the null termination character. If the length of the returned string is not required, a value of `NULL` can be passed in the *`length`* argument.

The *`type`* argument will return a pointer to the uniform variable's data type. The symbolic constants returned for uniform types are shown in the table below.

Returned Symbolic Contant	**Shader Uniform Type**
GL_FLOAT	float
GL_FLOAT_VEC2	vec2
GL_FLOAT_VEC3	vec3
GL_FLOAT_VEC4	vec4
GL_INT	int
GL_INT_VEC2	ivec2
GL_INT_VEC3	ivec3
GL_INT_VEC4	ivec4
GL_UNSIGNED_INT	unsigned int
GL_UNSIGNED_INT_VEC2	uvec2
GL_UNSIGNED_INT_VEC3	uvec3
GL_UNSIGNED_INT_VEC4	uvec4
GL_BOOL	bool
GL_BOOL_VEC2	bvec2
GL_BOOL_VEC3	bvec3
GL_BOOL_VEC4	bvec4
GL_FLOAT_MAT2	mat2
GL_FLOAT_MAT3	mat3
GL_FLOAT_MAT4	mat4
GL_FLOAT_MAT2x3	mat2x3
GL_FLOAT_MAT2x4	mat2x4
GL_FLOAT_MAT3x2	mat3x2
GL_FLOAT_MAT3x4	mat3x4
GL_FLOAT_MAT4x2	mat4x2
GL_FLOAT_MAT4x3	mat4x3
GL_SAMPLER_2D	sampler2D
GL_SAMPLER_3D	sampler3D
GL_SAMPLER_CUBE	samplerCube
GL_SAMPLER_2D_SHADOW	sampler2DShadow
GL_SAMPLER_2D_ARRAY	sampler2DArray

Returned Symbolic Contant	**Shader Uniform Type**
`GL_SAMPLER_2D_ARRAY_SHADOW`	`sampler2DArrayShadow`
`GL_SAMPLER_CUBE_SHADOW`	`samplerCubeShadow`
`GL_INT_SAMPLER_2D`	`isampler2D`
`GL_INT_SAMPLER_3D`	`isampler3D`
`GL_INT_SAMPLER_CUBE`	`isamplerCube`
`GL_INT_SAMPLER_2D_ARRAY`	`isampler2DArray`
`GL_UNSIGNED_INT_SAMPLER_2D`	`usampler2D`
`GL_UNSIGNED_INT_SAMPLER_3D`	`usampler3D`
`GL_UNSIGNED_INT_SAMPLER_CUBE`	`usamplerCube`
`GL_UNSIGNED_INT_SAMPLER_2D_ARRAY`	`usampler2DArray`

If one or more elements of an array are active, the name of the array is returned in *`name`*, the type is returned in *`type`*, and the *`size`* parameter returns the highest array element index used, plus one, as determined by the compiler and/or linker. Only one active uniform variable will be reported for a uniform array. If the active uniform is an array, the uniform name returned in *`name`* will always be the name of the uniform array appended with "[0]".

Uniform variables that are declared as structures or arrays of structures will not be returned directly by this function. Instead, each of these uniform variables will be reduced to its fundamental components containing the "." and "[]" operators such that each of the names is valid as an argument to glGetUniformLocation. Each of these reduced uniform variables is counted as one active uniform variable and is assigned an index. A valid name cannot be a structure, an array of structures, or a subcomponent of a vector or matrix.

The size of the uniform variable will be returned in *`size`*. Uniform variables other than arrays will have a size of 1. Structures and arrays of structures will be reduced as described earlier, such that each of the names returned will be a data type in the earlier list. If this reduction results in an array, the size returned will be as described for uniform arrays; otherwise, the size returned will be 1.

The list of active uniform variables may include both built-in uniform variables (which begin with the prefix "gl_") as well as user-defined uniform variable names.

This function will return as much information as it can about the specified active uniform variable. If no information is available, *`length`* will be 0, and *`name`* will be an empty string. This situation could occur if this function is called after a link operation that failed. If an error occurs, the return values *`length`*, *`size`*, *`type`*, and *`name`* will be unmodified.

Errors

`GL_INVALID_VALUE` is generated if *`program`* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`program`* is not a program object.

`GL_INVALID_VALUE` is generated if *`index`* is greater than or equal to the number of active uniform variables in *`program`*.

`GL_INVALID_VALUE` is generated if *`bufSize`* is less than 0.

Associated Gets

glGet with argument GL_MAX_VERTEX_UNIFORM_COMPONENTS, GL_MAX_FRAGMENT_UNIFORM_COMPONENTS, GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS, or GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS.

glGetProgramiv with argument GL_ACTIVE_UNIFORMS or GL_ACTIVE_UNIFORM_MAX_LENGTH.

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetActiveUniform	✔	✔

See Also

glGetUniform, glGetUniformLocation, glLinkProgram, glUniform, glUseProgram

Copyright

Name

glGetActiveUniformBlockName — retrieve the name of an active uniform block

C Specification

```
void glGetActiveUniformBlockName (program, uniformBlockIndex, bufSize,
length, uniformBlockName);

GLuint program;
GLuint uniformBlockIndex;
GLsizei bufSize;
GLsizei *length;
GLchar *uniformBlockName;
```

Parameters

program	Specifies the name of a program containing the uniform block.
uniformBlockIndex	Specifies the index of the uniform block within *program*.
bufSize	Specifies the size of the buffer addressed by *uniformBlockName*.
length	Specifies the address of a variable to receive the number of characters that were written to *uniformBlockName*.
uniformBlockName	Specifies the address an array of characters to receive the name of the uniform block at *uniformBlockIndex*.

Description

`glGetActiveUniformBlockName` retrieves the name of the active uniform block at *uniform-BlockIndex* within *program*.

program must be the name of a program object for which the command glLinkProgram must have been called in the past, although it is not required that glLinkProgram must have succeeded. The link could have failed because the number of active uniforms exceeded the limit.

uniformBlockIndex is an active uniform block index of *program*, and must be less than the value of `GL_ACTIVE_UNIFORM_BLOCKS`.

Upon success, the name of the uniform block identified by *unifomBlockIndex* is returned into *uniformBlockName*. The name is nul-terminated. The actual number of characters written into *uniform-BlockName*, excluding the nul terminator, is returned in *length*. If *length* is `NULL`, no length is returned.

bufSize contains the maximum number of characters (including the nul terminator) that will be written into *uniformBlockName*.

If an error occurs, nothing will be written to *uniformBlockName* or *length*.

Errors

`GL_INVALID_OPERATION` is generated if *program* is not the name of a program object for which glLinkProgram has been called in the past.

`GL_INVALID_VALUE` is generated if *`uniformBlockIndex`* is greater than or equal to the value of `GL_ACTIVE_UNIFORM_BLOCKS` or is not the index of an active uniform block in *`program`*.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetActiveUniformBlockName	-	✔

See Also

glGetActiveUniformBlockiv, glGetUniformBlockIndex

Copyright

Name

glGetActiveUniformBlockiv — query information about an active uniform block

C Specification

```
void glGetActiveUniformBlockiv (program, uniformBlockIndex, pname,
params);

GLuint program;
GLuint uniformBlockIndex;
GLenum pname;
GLint *params;
```

Parameters

program	Specifies the name of a program containing the uniform block.
uniformBlockIndex	Specifies the index of the uniform block within `program`.
pname	Specifies the name of the parameter to query.
params	Specifies the address of a variable to receive the result of the query.

Description

`glGetActiveUniformBlockiv` retrieves information about an active uniform block within *program*.

program must be the name of a program object for which the command glLinkProgram must have been called in the past, although it is not required that glLinkProgram must have succeeded. The link could have failed because the number of active uniforms exceeded the limit.

uniformBlockIndex is an active uniform block index of *program*, and must be less than the value of `GL_ACTIVE_UNIFORM_BLOCKS`.

Upon success, the uniform block parameter(s) specified by *pname* are returned in *params*. If an error occurs, nothing will be written to *params*.

If *pname* is `GL_UNIFORM_BLOCK_BINDING`, then the index of the uniform buffer binding point last selected by the uniform block specified by *uniformBlockIndex* for *program* is returned. If no uniform block has been previously specified, zero is returned.

If *pname* is `GL_UNIFORM_BLOCK_DATA_SIZE`, then the implementation-dependent minimum total buffer object size, in basic machine units, required to hold all active uniforms in the uniform block identified by *uniformBlockIndex* is returned. It is neither guaranteed nor expected that a given implementation will arrange uniform values as tightly packed in a buffer object. The exception to this is the *std140 uniform block layout*, which guarantees specific packing behavior and does not require the application to query for offsets and strides. In this case the minimum size may still be queried, even though it is determined in advance based only on the uniform block declaration.

If *pname* is `GL_UNIFORM_BLOCK_NAME_LENGTH`, then the total length (including the nul terminator) of the name of the uniform block identified by *uniformBlockIndex* is returned.

If *pname* is `GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS`, then the number of active uniforms in the uniform block identified by *uniformBlockIndex* is returned.

If *`pname`* is `GL_UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES`, then a list of the active uniform indices for the uniform block identified by *`uniformBlockIndex`* is returned. The number of elements that will be written to *`params`* is the value of `GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS` for *`uniformBlockIndex`*.

If *`pname`* is `GL_UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER`, or `GL_UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER`, then a boolean value indicating whether the uniform block identified by *`uniformBlockIndex`* is referenced by the vertex or fragment programming stages of program, respectively, is returned.

Errors

`GL_INVALID_VALUE` is generated if *`uniformBlockIndex`* is greater than or equal to the value of `GL_ACTIVE_UNIFORM_BLOCKS` or is not the index of an active uniform block in *`program`*.

`GL_INVALID_ENUM` is generated if *`pname`* is not one of the accepted tokens.

`GL_INVALID_OPERATION` is generated if *`program`* is not the name of a program object for which glLinkProgram has been called in the past.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetActiveUniformBlockiv	-	✔

See Also

glGetActiveUniformBlockName, glGetUniformBlockIndex, glLinkProgram

Copyright

Name

glGetActiveUniformsiv — Returns information about several active uniform variables for the specified program object

C Specification

```
void glGetActiveUniformsiv (program, uniformCount, uniformIndices,
pname, params);
```

```
GLuint program;
GLsizei uniformCount;
const GLuint *uniformIndices;
GLenum pname;
GLint *params;
```

Parameters

program	Specifies the program object to be queried.
uniformCount	Specifies both the number of elements in the array of indices *uniformIndices* and the number of parameters written to *params* upon successful return.
uniformIndices	Specifies the address of an array of *uniformCount* integers containing the indices of uniforms within *program* whose parameter *pname* should be queried.
pname	Specifies the property of each uniform in *uniformIndices* that should be written into the corresponding element of *params*.
params	Specifies the address of an array of *uniformCount* integers which are to receive the value of *pname* for each uniform in *uniformIndices*.

Description

`glGetActiveUniformsiv` queries the value of the parameter named *pname* for each of the uniforms within *program* whose indices are specified in the array of *uniformCount* unsigned integers *uniformIndices*. Upon success, the value of the parameter for each uniform is written into the corresponding entry in the array whose address is given in *params*. If an error is generated, nothing is written into *params*.

If *pname* is `GL_UNIFORM_TYPE`, then an array identifying the types of uniforms specified by the corresponding array of *uniformIndices* is returned. The returned types can be any of the values from the following table:

Returned Symbolic Contant	**Shader Uniform Type**
`GL_FLOAT`	`float`
`GL_FLOAT_VEC2`	`vec2`
`GL_FLOAT_VEC3`	`vec3`
`GL_FLOAT_VEC4`	`vec4`
`GL_INT`	`int`
`GL_INT_VEC2`	`ivec2`
`GL_INT_VEC3`	`ivec3`
`GL_INT_VEC4`	`ivec4`

Returned Symbolic Contant	**Shader Uniform Type**
`GL_UNSIGNED_INT`	`unsigned int`
`GL_UNSIGNED_INT_VEC2`	`uvec2`
`GL_UNSIGNED_INT_VEC3`	`uvec3`
`GL_UNSIGNED_INT_VEC4`	`uvec4`
`GL_BOOL`	`bool`
`GL_BOOL_VEC2`	`bvec2`
`GL_BOOL_VEC3`	`bvec3`
`GL_BOOL_VEC4`	`bvec4`
`GL_FLOAT_MAT2`	`mat2`
`GL_FLOAT_MAT3`	`mat3`
`GL_FLOAT_MAT4`	`mat4`
`GL_FLOAT_MAT2x3`	`mat2x3`
`GL_FLOAT_MAT2x4`	`mat2x4`
`GL_FLOAT_MAT3x2`	`mat3x2`
`GL_FLOAT_MAT3x4`	`mat3x4`
`GL_FLOAT_MAT4x2`	`mat4x2`
`GL_FLOAT_MAT4x3`	`mat4x3`
`GL_SAMPLER_2D`	`sampler2D`
`GL_SAMPLER_3D`	`sampler3D`
`GL_SAMPLER_CUBE`	`samplerCube`
`GL_SAMPLER_2D_SHADOW`	`sampler2DShadow`
`GL_SAMPLER_2D_ARRAY`	`sampler2DArray`
`GL_SAMPLER_2D_ARRAY_SHADOW`	`sampler2DArrayShadow`
`GL_SAMPLER_CUBE_SHADOW`	`samplerCubeShadow`
`GL_INT_SAMPLER_2D`	`isampler2D`
`GL_INT_SAMPLER_3D`	`isampler3D`
`GL_INT_SAMPLER_CUBE`	`isamplerCube`
`GL_INT_SAMPLER_2D_ARRAY`	`isampler2DArray`
`GL_UNSIGNED_INT_SAMPLER_2D`	`usampler2D`
`GL_UNSIGNED_INT_SAMPLER_3D`	`usampler3D`
`GL_UNSIGNED_INT_SAMPLER_CUBE`	`usamplerCube`
`GL_UNSIGNED_INT_SAMPLER_2D_ARRAY`	`usampler2DArray`

If *`pname`* is `GL_UNIFORM_SIZE`, then an array identifying the size of the uniforms specified by the corresponding array of *`uniformIndices`* is returned. The sizes returned are in units of the type returned by a query of `GL_UNIFORM_TYPE`. For active uniforms that are arrays, the size is the number of active elements in the array; for all other uniforms, the size is one.

If *`pname`* is `GL_UNIFORM_NAME_LENGTH`, then an array identifying the length, including the terminating null character, of the uniform name strings specified by the corresponding array of *`uniformIndices`* is returned.

If *pname* is `GL_UNIFORM_BLOCK_INDEX`, then an array identifying the uniform block index of each of the uniforms specified by the corresponding array of *uniformIndices* is returned. The uniform block index of a uniform associated with the default uniform block is -1.

If *pname* is `GL_UNIFORM_OFFSET`, then an array of uniform buffer offsets is returned. For uniforms in a named uniform block, the returned value will be its offset, in basic machine units, relative to the beginning of the uniform block in the buffer object data store. For uniforms in the default uniform block, -1 will be returned.

If *pname* is `GL_UNIFORM_ARRAY_STRIDE`, then an array identifying the stride between elements, in basic machine units, of each of the uniforms specified by the corresponding array of *uniformIndices* is returned. The stride of a uniform associated with the default uniform block is -1. Note that this information only makes sense for uniforms that are arrays. For uniforms that are not arrays, but are declared in a named uniform block, an array stride of zero is returned.

If *pname* is `GL_UNIFORM_MATRIX_STRIDE`, then an array identifying the stride between columns of a column-major matrix or rows of a row-major matrix, in basic machine units, of each of the uniforms specified by the corresponding array of *uniformIndices* is returned. The matrix stride of a uniform associated with the default uniform block is -1. Note that this information only makes sense for uniforms that are matrices. For uniforms that are not matrices, but are declared in a named uniform block, a matrix stride of zero is returned.

If *pname* is `GL_UNIFORM_IS_ROW_MAJOR`, then an array identifying whether each of the uniforms specified by the corresponding array of *uniformIndices* is a row-major matrix or not is returned. A value of one indicates a row-major matrix, and a value of zero indicates a column-major matrix, a matrix in the default uniform block, or a non-matrix.

Errors

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_VALUE` is generated if *uniformCount* is greater than or equal to the value of `GL_ACTIVE_UNIFORMS` for *program*.

`GL_INVALID_ENUM` is generated if *pname* is not an accepted token.

Associated Gets

glGet with argument `GL_MAX_VERTEX_UNIFORM_COMPONENTS`, `GL_MAX_FRAGMENT_UNIFORM_COMPONENTS`, `GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS`, or `GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS`.

glGetProgramiv with argument `GL_ACTIVE_UNIFORMS` or `GL_ACTIVE_UNIFORM_MAX_LENGTH`.

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetActiveUniformsiv	-	✔

See Also

glGetUniform, glGetActiveUniform, glGetUniformLocation, glLinkProgram, glUniform, glUseProgram

Copyright

Name

glGetAttachedShaders — Returns the handles of the shader objects attached to a program object

C Specification

```
void glGetAttachedShaders (program, maxCount, count, shaders);
```

```
GLuint program;
GLsizei maxCount;
GLsizei *count;
GLuint *shaders;
```

Parameters

program	Specifies the program object to be queried.
maxCount	Specifies the size of the array for storing the returned object names.
count	Returns the number of names actually returned in *shaders*.
shaders	Specifies an array that is used to return the names of attached shader objects.

Description

`glGetAttachedShaders` returns the names of the shader objects attached to *program*. The names of shader objects that are attached to *program* will be returned in *shaders.* The actual number of shader names written into *shaders* is returned in *count.* If no shader objects are attached to *program*, *count* is set to 0. The maximum number of shader names that may be returned in *shaders* is specified by *maxCount*.

If the number of names actually returned is not required (for instance, if it has just been obtained by calling glGetProgramiv), a value of `NULL` may be passed for count. If no shader objects are attached to *program*, a value of 0 will be returned in *count*. The actual number of attached shaders can be obtained by calling glGetProgramiv with the value `GL_ATTACHED_SHADERS`.

Errors

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_VALUE` is generated if *maxCount* is less than 0.

Associated Gets

glGetProgramiv with argument `GL_ATTACHED_SHADERS`

glIsProgram

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glGetAttachedShaders	✔	✔

See Also

glAttachShader, glDetachShader.

Copyright

Name

glGetAttribLocation — Returns the location of an attribute variable

C Specification

```
GLint glGetAttribLocation (program, name);

GLuint program;
const GLchar *name;
```

Parameters

program	Specifies the program object to be queried.
name	Points to a null terminated string containing the name of the attribute variable whose location is to be queried.

Description

`glGetAttribLocation` queries the previously linked program object specified by *program* for the attribute variable specified by *name* and returns the index of the generic vertex attribute that is bound to that attribute variable. If *name* is a matrix attribute variable, the index of the first column of the matrix is returned. If the named attribute variable is not an active attribute in the specified program object or if *name* starts with the reserved prefix "gl_", a value of -1 is returned.

The association between an attribute variable name and a generic attribute index can be specified at any time by calling glBindAttribLocation. Attribute bindings do not go into effect until glLinkProgram is called. After a program object has been linked successfully, the index values for attribute variables remain fixed until the next link command occurs. The attribute values can only be queried after a link if the link was successful. `glGetAttribLocation` returns the binding that actually went into effect the last time glLinkProgram was called for the specified program object. Attribute bindings that have been specified since the last link operation are not returned by `glGetAttribLocation`.

Errors

`GL_INVALID_OPERATION` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *program* has not been successfully linked.

Associated Gets

glGetActiveAttrib with argument *program* and the index of an active attribute

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glGetAttribLocation`	✔	✔

See Also

glBindAttribLocation, glIsProgram, glLinkProgram, glVertexAttrib, glVertexAttribPointer

Copyright

Name

glGetBufferParameter — return parameters of a buffer object

C Specification

```
void glGetBufferParameteriv (target, value, data);

GLenum target;
GLenum value;
GLint * data;

void glGetBufferParameteri64v (target, value, data);

GLenum target;
GLenum value;
GLint64 * data;
```

Parameters

target	Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.
value	Specifies the symbolic name of a buffer object parameter. Accepted values are `GL_BUFFER_ACCESS_FLAGS`, `GL_BUFFER_MAPPED`, `GL_BUFFER_MAP_LENGTH`, `GL_BUFFER_MAP_OFFSET`, `GL_BUFFER_SIZE`, or `GL_BUFFER_USAGE`.
data	Returns the requested parameter.

Description

`glGetBufferParameteriv` and `glGetBufferParameteri64v` return in *data* a selected parameter of the buffer object specified by *target*.

value names a specific buffer object parameter, as follows:

`GL_BUFFER_ACCESS_FLAGS`	*params* returns the access policy set while mapping the buffer object.
`GL_BUFFER_MAPPED`	*params* returns a flag indicating whether the buffer object is currently mapped. The initial value is `GL_FALSE`.
`GL_BUFFER_MAP_LENGTH`	*params* returns the length of the buffer object mapping, measured in bytes. The initial value is 0.
`GL_BUFFER_MAP_OFFSET`	*params* returns the offset (start) of the buffer object mapping, measured in bytes. The initial value is 0.
`GL_BUFFER_SIZE`	*params* returns the size of the buffer object, measured in bytes. The initial value is 0.
`GL_BUFFER_USAGE`	*params* returns the buffer object's usage pattern.

Notes

If an error is generated, no change is made to the contents of *data*.

If `glGetBufferParameteriv` is used to query a *`value`* of `GL_BUFFER_SIZE`, values greater than or equal to will be clamped to .

Errors

`GL_INVALID_ENUM` is generated if *`target`* or *`value`* is not an accepted value.

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *`target`*.

`GL_INVALID_ENUM` is generated if `glGetBufferParameteri64v` is used to query a *`value`* of `GL_BUFFER_ACCESS_FLAGS`, `GL_BUFFER_MAPPED` or `GL_BUFFER_USAGE`.

`GL_INVALID_ENUM` is generated if `glGetBufferParameteriv` is used to query a *`value`* of `GL_BUFFER_MAP_LENGTH` or `GL_BUFFER_MAP_OFFSET`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glGetBufferParameteriv`	✔	✔
`glGetBufferParameteri64v`	-	✔

See Also

glBindBuffer, glBufferData, glMapBufferRange, glUnmapBuffer

Copyright

Name

glGetBufferPointerv — return the pointer to a mapped buffer object's data store

C Specification

```
void glGetBufferPointerv (target, pname, params);

GLenum target;
GLenum pname;
void ** params;
```

Parameters

target	Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.
pname	Specifies the pointer to be returned. The symbolic constant must be `GL_BUFFER_MAP_POINTER`.
params	Returns the pointer value specified by *pname*.

Description

`glGetBufferPointerv` returns pointer information. *pname* is a symbolic constant indicating the pointer to be returned, which must be `GL_BUFFER_MAP_POINTER`, the pointer to which the buffer object's data store is mapped. If the data store is not currently mapped, `NULL` is returned. *params* is a pointer to a location in which to place the returned pointer value.

Notes

If an error is generated, no change is made to the contents of *params*.

The initial value for the pointer is `NULL`.

Errors

`GL_INVALID_ENUM` is generated if *target* or *pname* is not an accepted value.

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *target*.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetBufferPointerv	-	✔

See Also

glBindBuffer, glMapBufferRange

Copyright

Name

glGetError — return error information

C Specification

```
GLenum glGetError (void);
```

```
void;
```

Description

`glGetError` returns the value of the error flag. Each detectable error is assigned a numeric code and symbolic name. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until `glGetError` is called, the error code is returned, and the flag is reset to `GL_NO_ERROR`. If a call to `glGetError` returns `GL_NO_ERROR`, there has been no detectable error since the last call to `glGetError`, or since the GL was initialized.

To allow for distributed implementations, there may be several error flags. If any single error flag has recorded an error, the value of that flag is returned and that flag is reset to `GL_NO_ERROR` when `glGetError` is called. If more than one flag has recorded an error, `glGetError` returns and clears an arbitrary error flag value. Thus, `glGetError` should always be called in a loop, until it returns `GL_NO_ERROR`, if all error flags are to be reset.

Initially, all error flags are set to `GL_NO_ERROR`.

The following errors are currently defined:

`GL_NO_ERROR`	No error has been recorded. The value of this symbolic constant is guaranteed to be 0.
`GL_INVALID_ENUM`	An unacceptable value is specified for an enumerated argument. The offending command is ignored and has no other side effect than to set the error flag.
`GL_INVALID_VALUE`	A numeric argument is out of range. The offending command is ignored and has no other side effect than to set the error flag.
`GL_INVALID_OPERATION`	The specified operation is not allowed in the current state. The offending command is ignored and has no other side effect than to set the error flag.
`GL_INVALID_FRAMEBUFFER_OPERATION`	The framebuffer object is not complete. The offending command is ignored and has no other side effect than to set the error flag.
`GL_OUT_OF_MEMORY`	There is not enough memory left to execute the command. The state of the GL is undefined, except for the state of the error flags, after this error is recorded.

When an error flag is set, results of a GL operation are undefined only if `GL_OUT_OF_MEMORY` has occurred. In all other cases, the command generating the error is ignored and has no effect on the GL state or frame buffer contents. If the generating command returns a value, it returns 0. If `glGetError` itself generates an error, it returns 0.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glGetError	✔	✔

Copyright

Name

glGetFragDataLocation — query the bindings of color numbers to user-defined varying out variables

C Specification

```
GLint glGetFragDataLocation (program, name);

GLuint program;
const char * name;
```

Parameters

program	The name of the program containing varying out variable whose binding to query
name	The name of the user-defined varying out variable whose binding to query

Description

`glGetFragDataLocation` retrieves the assigned color number binding for the user-defined varying out variable *name* for program *program*. *program* must have previously been linked. *name* must be a null-terminated string. If *name* is not the name of an active user-defined varying out fragment shader variable within *program*, -1 will be returned.

Notes

In OpenGL ES Shading Language version 3.00, output variables must be explicitly bound to fragment colors within the shader text. This query simply returns that binding information.

Errors

`GL_INVALID_OPERATION` is generated if *program* is not the name of a program object.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetFragDataLocation	-	✔

See Also

glCreateProgram,

Copyright

Name

glGetFramebufferAttachmentParameteriv — retrieve information about attachments of a bound framebuffer object

C Specification

```
void glGetFramebufferAttachmentParameteriv (target, attachment, pname,
params);

GLenum target;
GLenum attachment;
GLenum pname;
GLint *params;
```

Parameters

target	Specifies the target of the query operation.
attachment	Specifies the attachment within *target*
pname	Specifies the parameter of *attachment* to query.
params	Specifies the address of a variable receive the value of *pname* for *attachment*.

Description

`glGetFramebufferAttachmentParameteriv` returns information about attachments of a bound framebuffer object. *target* specifies the framebuffer binding point and must be `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER` or `GL_FRAMEBUFFER`. `GL_FRAMEBUFFER` is equivalent to `GL_DRAW_FRAMEBUFFER`.

If the default framebuffer is bound to *target* then *attachment* must be one of `GL_BACK`, identifying a color buffer, `GL_DEPTH`, identifying the depth buffer, or `GL_STENCIL`, identifying the stencil buffer.

If a framebuffer object is bound, then *attachment* must be one of `GL_COLOR_ATTACHMENT`*i*, `GL_DEPTH_ATTACHMENT`, `GL_STENCIL_ATTACHMENT`, or `GL_DEPTH_STENCIL_ATTACHMENT`. *i* in `GL_COLOR_ATTACHMENT`*i* must be in the range zero to the value of `GL_MAX_COLOR_ATTACHMENTS` minus one.

If *attachment* is `GL_DEPTH_STENCIL_ATTACHMENT` and different objects are bound to the depth and stencil attachment points of *target* the query will fail. If the same object is bound to both attachment points, information about that object will be returned.

Upon successful return from `glGetFramebufferAttachmentParameteriv`, if *pname* is `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`, then *params* will contain one of `GL_NONE`, `GL_FRAMEBUFFER_DEFAULT`, `GL_TEXTURE`, or `GL_RENDERBUFFER`, identifying the type of object which contains the attached image. Other values accepted for *pname* depend on the type of object, as described below.

If the value of `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `GL_NONE`, no framebuffer is bound to *target*. In this case querying *pname* `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` will return zero, and all other queries will generate an error.

If the value of `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is not `GL_NONE`, these queries apply to all other framebuffer types:

- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_RED_SIZE, GL_FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, GL_FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, GL_FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, GL_FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, or GL_FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, then *params* will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified attachment. Zero is returned if the requested component is not present in *attachment*.
- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, *params* will contain the format of components of the specified attachment, one of GL_FLOAT, *GL_INT*, *GL_UNSIGNED_INT*, *GL_SIGNED_NORMALIZED*, or *GL_UNSIGNED_NORMALIZED* for floating-point, signed integer, unsigned integer, signed normalized fixed-point, or unsigned normalized fixed-point components respectively. Only color buffers may have integer components.
- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, *param* will contain the encoding of components of the specified attachment, one of GL_LINEAR or GL_SRGB for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded. For the default framebuffer, color encoding is determined by the implementation. For framebuffer objects, components are sRGB-encoded if the internal format of a color attachment is one of the color-renderable SRGB formats.

If the value of GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is GL_RENDERBUFFER, then:

- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, *params* will contain the name of the renderbuffer object which contains the attached image.

If the value of GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is GL_TEXTURE, then:

- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, then *params* will contain the name of the texture object which contains the attached image.
- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, then *params* will contain the mipmap level of the texture object which contains the attached image.
- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE and the texture object named GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is a cube map texture, then *params* will contain the cube map face of the cubemap texture object which contains the attached image. Otherwise *params* will contain the value zero.
- If *pname* is GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER and the texture object named GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is a three-dimensional texture or a two-dimensional array texture, then *params* will contain the number of the texture layer which contains the attached image. Otherwise *params* will contain the value zero.

Any combinations of framebuffer type and *pname* not described above will generate an error.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the accepted tokens.

GL_INVALID_ENUM is generated if *pname* is not valid for the value of GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE.

GL_INVALID_OPERATION is generated if *attachment* is not the accepted values for *target*.

GL_INVALID_OPERATION is generated if *attachment* is GL_DEPTH_STENCIL_ATTACHMENT and different objects are bound to the depth and stencil attachment points of *target*.

`GL_INVALID_OPERATION` is generated if the value of `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `GL_NONE` and *`pname`* is not `GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glGetFramebufferAttachmentParameteriv	✔	✔

See Also

glGenFramebuffers, glBindFramebuffer

Copyright

Name

glGetInternalformativ — retrieve information about implementation-dependent support for internal formats

C Specification

```
void glGetInternalformativ (target, internalformat, pname, bufSize,
params);

GLenum target;
GLenum internalformat;
GLenum pname;
GLsizei bufSize;
GLint *params;
```

Parameters

target	Indicates the usage of the internal format. *target* must be `GL_RENDERBUFFER`.
internalformat	Specifies the internal format about which to retrieve information.
pname	Specifies the type of information to query.
bufSize	Specifies the maximum number of integers that may be written to *params* by the function.
params	Specifies the address of a variable into which to write the retrieved information.

Description

`glGetInternalformativ` retrieves information about implementation-dependent support for internal formats. *target* indicates the target with which the internal format will be used and must be `GL_RENDERBUFFER`, corresponding to usage as a renderbuffer.

internalformat specifies the internal format about which to retrieve information and must be a color-renderable, depth-renderable or stencil-renderable format.

The information retrieved will be written to memory addressed by the pointer specified in *params*. No more than *bufSize* integers will be written to this memory.

If *pname* is `GL_NUM_SAMPLE_COUNTS`, the number of sample counts that would be returned by querying `GL_SAMPLES` will be returned in *params*.

If *pname* is `GL_SAMPLES`, the sample counts supported for *internalformat* and *target* are written into *params* in descending numeric order. Only positive values are returned. Querying `GL_SAMPLES` with *bufSize* of one will return just the maximum supported number of samples for this format.

Notes

Since multisampling is not supported for signed and unsigned integer internal formats, the value of `GL_NUM_SAMPLE_COUNTS` will be zero for such formats. If *internalformat* is `GL_RGBA16F`, `GL_R32F`, `GL_RG32F`, or `GL_RGBA32F`, the value of `GL_NUM_SAMPLE_COUNTS` may be zero, or else the maximum value in `GL_SAMPLES` may be less than the value of `GL_MAX_SAMPLES`. For every other accepted *`internalformat`*, the maximum value in `GL_SAMPLES` is guaranteed to be at least `GL_MAX_SAMPLES`.

Errors

`GL_INVALID_VALUE` is generated if *`bufSize`* is negative.

`GL_INVALID_ENUM` is generated if *`pname`* is not `GL_SAMPLES` or `GL_NUM_SAMPLE_COUNTS`.

`GL_INVALID_ENUM` is generated if *`internalformat`* is not color-, depth-, or stencil-renderable.

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_RENDERBUFFER`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetInternalformativ	-	✔

See Also

glGet

Copyright

Name

glGetProgramBinary — return a binary representation of a program object's compiled and linked executable source

C Specification

```
void glGetProgramBinary (program, bufsize, length, binaryFormat, binary);
```

```
GLuint program;
GLsizei bufsize;
GLsizei *length;
GLenum *binaryFormat;
void *binary;
```

Parameters

program	Specifies the name of a program object whose binary representation to retrieve.
bufSize	Specifies the size of the buffer whose address is given by *binary*.
length	Specifies the address of a variable to receive the number of bytes written into *binary*.
binaryFormat	Specifies the address of a variable to receive a token indicating the format of the binary data returned by the GL.
binary	Specifies the address an array into which the GL will return *program*'s binary representation.

Description

`glGetProgramBinary` returns a binary representation of the compiled and linked executable for *program* into the array of bytes whose address is specified in *binary*. The maximum number of bytes that may be written into *binary* is specified by *bufSize*. If the program binary is greater in size than *bufSize* bytes, then an error is generated, otherwise the actual number of bytes written into *binary* is returned in the variable whose address is given by *length*. If *length* is `NULL`, then no length is returned.

The format of the program binary written into *binary* is returned in the variable whose address is given by *binaryFormat*, and may be implementation dependent. The binary produced by the GL may subsequently be returned to the GL by calling glProgramBinary, with *binaryFormat* and *length* set to the values returned by `glGetProgramBinary`, and passing the returned binary data in the *binary* parameter.

Errors

`GL_INVALID_OPERATION` is generated if *bufSize* is less than the size of `GL_PROGRAM_BINARY_LENGTH` for *program*.

`GL_INVALID_OPERATION` is generated if `GL_LINK_STATUS` for the program object is false.

Associated Gets

glGetProgramiv with argument `GL_PROGRAM_BINARY_LENGTH`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetProgramBinary	-	✔

See Also

glGetProgramiv, glProgramBinary

Copyright

Name

glGetProgramInfoLog — Returns the information log for a program object

C Specification

```
void glGetProgramInfoLog (program, maxLength, length, infoLog);

GLuint program;
GLsizei maxLength;
GLsizei *length;
GLchar *infoLog;
```

Parameters

program	Specifies the program object whose information log is to be queried.
maxLength	Specifies the size of the character buffer for storing the returned information log.
length	Returns the length of the string returned in *infoLog* (excluding the null terminator).
infoLog	Specifies an array of characters that is used to return the information log.

Description

`glGetProgramInfoLog` returns the information log for the specified program object. The information log for a program object is modified when the program object is linked or validated. The string that is returned will be null terminated.

`glGetProgramInfoLog` returns in *infoLog* as much of the information log as it can, up to a maximum of *maxLength* characters. The number of characters actually returned, excluding the null termination character, is specified by *length*. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument. The size of the buffer required to store the returned information log can be obtained by calling glGetProgramiv with the value `GL_INFO_LOG_LENGTH`.

The information log for a program object is either an empty string, or a string containing information about the last link operation, or a string containing information about the last validation operation. It may contain diagnostic messages, warning messages, and other information. When a program object is created, its information log will be a string of length 0.

Notes

The information log for a program object is the OpenGL implementer's primary mechanism for conveying information about linking and validating. Therefore, the information log can be helpful to application developers during the development process, even when these operations are successful. Application developers should not expect different OpenGL implementations to produce identical information logs.

Errors

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_VALUE` is generated if *maxLength* is less than 0.

Associated Gets

glGetProgramiv with argument `GL_INFO_LOG_LENGTH`

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetProgramInfoLog	✔	✔

See Also

glCompileShader, glGetShaderInfoLog, glLinkProgram, glValidateProgram

Copyright

Name

glGetProgramiv — Returns a parameter from a program object

C Specification

```
void glGetProgramiv (program, pname, params);

GLuint program;
GLenum pname;
GLint *params;
```

Parameters

program	Specifies the program object to be queried.
pname	Specifies the object parameter. Accepted symbolic names are GL_ACTIVE_ATTRIBUTES, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, GL_ACTIVE_UNIFORMS, GL_ACTIVE_UNIFORM_BLOCKS, GL_ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, GL_ACTIVE_UNIFORM_MAX_LENGTH, GL_ATTACHED_SHADERS, GL_DELETE_STATUS, GL_INFO_LOG_LENGTH, GL_LINK_STATUS, GL_PROGRAM_BINARY_RETRIEVABLE_HINT, GL_TRANSFORM_FEEDBACK_BUFFER_MODE, GL_TRANSFORM_FEEDBACK_VARYINGS, GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH and GL_VALIDATE_STATUS.
params	Returns the requested object parameter.

Description

glGetProgramiv returns in *params* the value of a parameter for a specific program object. The following parameters are defined:

GL_ACTIVE_ATTRIBUTES	*params* returns the number of active attribute variables for *program*.
GL_ACTIVE_ATTRIBUTE_MAX_LENGTH	*params* returns the length of the longest active attribute name for *program*, including the null termination character (i.e., the size of the character buffer required to store the longest attribute name). If no active attributes exist, 0 is returned.
GL_ACTIVE_UNIFORM_BLOCKS	*params* returns the number of uniform blocks for *program* containing active uniforms.
GL_ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH	*params* returns the length of the longest active uniform block name for *program*, including the null termination character (i.e., the size of the character buffer required to store the longest uniform block name). If no active uniform blocks exist, 0 is returned.
GL_ACTIVE_UNIFORMS	*params* returns the number of active uniform variables for *program*.

`GL_ACTIVE_UNIFORM_MAX_LENGTH`	*`params`* returns the length of the longest active uniform variable name for *`program`*, including the null termination character (i.e., the size of the character buffer required to store the longest uniform variable name). If no active uniform variables exist, 0 is returned.
`GL_ATTACHED_SHADERS`	*`params`* returns the number of shader objects attached to *`program`*.
`GL_DELETE_STATUS`	*`params`* returns `GL_TRUE` if *`program`* is currently flagged for deletion, and `GL_FALSE` otherwise.
`GL_INFO_LOG_LENGTH`	*`params`* returns the number of characters in the information log for *`program`* including the null termination character (i.e., the size of the character buffer required to store the information log). If *`program`* has no information log, a value of 0 is returned.
`GL_LINK_STATUS`	*`params`* returns `GL_TRUE` if the last link operation on *`program`* was successful, and `GL_FALSE` otherwise.
`GL_PROGRAM_BINARY_RETRIEVABLE_HINT`	*`params`* returns the current value of whether the binary retrieval hint is enabled for *`program`*.
`GL_TRANSFORM_FEEDBACK_BUFFER_MODE`	*`params`* returns a symbolic constant indicating the buffer mode used when transform feedback is active. This may be `GL_SEPARATE_ATTRIBS` or `GL_INTERLEAVED_ATTRIBS`.
`GL_TRANSFORM_FEEDBACK_VARYINGS`	*`params`* returns the number of varying variables to capture in transform feedback mode for the program.
`GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`	*`params`* returns the length of the longest variable name to be used for transform feedback, including the null-terminator.
`GL_VALIDATE_STATUS`	*`params`* returns `GL_TRUE` or if the last validation operation on *`program`* was successful, and `GL_FALSE` otherwise.

Notes

If an error is generated, no change is made to the contents of *`params`*.

Errors

`GL_INVALID_VALUE` is generated if *`program`* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`program`* does not refer to a program object.

`GL_INVALID_ENUM` is generated if *`pname`* is not an accepted value.

Associated Gets

glGetActiveAttrib with argument *`program`*

glGetActiveUniform with argument *`program`*

glGetAttachedShaders with argument *`program`*

glGetProgramInfoLog with argument *`program`*

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetProgramiv	✔	✔

See Also

glAttachShader, glCreateProgram, glDeleteProgram, glGetShaderiv, glLinkProgram, glValidateProgram

Copyright

Name

glGetQueryObjectuiv — return parameters of a query object

C Specification

```
void glGetQueryObjectuiv (id, pname, params);

GLuint id;
GLenum pname;
GLuint * params;
```

Parameters

id	Specifies the name of a query object.
pname	Specifies the symbolic name of a query object parameter. Accepted values are `GL_QUERY_RESULT` or `GL_QUERY_RESULT_AVAILABLE`.
params	Returns the requested data.

Description

`glGetQueryObjectuiv` returns in *params* a selected parameter of the query object specified by *id*.

pname names a specific query object parameter. *pname* can be as follows:

`GL_QUERY_RESULT`	*params* returns the value of the query object's passed samples counter. The initial value is 0.
`GL_QUERY_RESULT_AVAILABLE`	*params* returns whether the passed samples counter is immediately available. If a delay would occur waiting for the query result, `GL_FALSE` is returned. Otherwise, `GL_TRUE` is returned, which also indicates that the results of all previous queries of the same type are available as well.

Notes

If an error is generated, no change is made to the contents of *params*.

`glGetQueryObjectuiv` implicitly flushes the GL pipeline so that any incomplete rendering delimited by the occlusion query completes in finite time.

Repeatedly querying the `GL_QUERY_RESULT_AVAILABLE` state for any given query object is guaranteed to return true eventually. Note that multiple queries to the same occlusion object may result in a significant performance loss. For better performance it is recommended to wait N frames before querying this state. N is implementation-dependent but is generally between one and three.

If multiple queries are issued using the same query object *id* before calling `glGetQueryObjectuiv`, the results of the most recent query will be returned. In this case, when issuing a new query, the results of the previous query are discarded.

Errors

`GL_INVALID_ENUM` is generated if *pname* is not an accepted value.

`GL_INVALID_OPERATION` is generated if *`id`* is not the name of a query object.

`GL_INVALID_OPERATION` is generated if *`id`* is the name of a currently active query object.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetQueryObjectuiv	-	✔

See Also

glBeginQuery, glEndQuery, glGetQueryiv, glIsQuery,

Copyright

Name

glGetQueryiv — return parameters of a query object target

C Specification

```
void glGetQueryiv (target, pname, params);

GLenum target;
GLenum pname;
GLint * params;
```

Parameters

target	Specifies a query object target. Must be `GL_ANY_SAMPLES_PASSED`, `GL_ANY_SAMPLES_PASSED_CONSERVATIVE`, or `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`.
pname	Specifies the symbolic name of a query object target parameter. Must be `GL_CURRENT_QUERY`.
params	Returns the requested data.

Description

`glGetQueryiv` returns in *params* a selected parameter of the query object target specified by *target*.

pname names a specific query object target parameter. When *pname* is `GL_CURRENT_QUERY`, the name of the currently active query for *target*, or zero if no query is active, will be placed in *params*.

Notes

If an error is generated, no change is made to the contents of *params*.

Errors

`GL_INVALID_ENUM` is generated if *target* or *pname* is not an accepted value.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetQueryiv	-	✔

See Also

glGetQueryObjectuiv, glIsQuery

Copyright

Name

glGetRenderbufferParameteriv — retrieve information about a bound renderbuffer object

C Specification

```
void glGetRenderbufferParameteriv (target, pname, params);

GLenum target;
GLenum pname;
GLint *params;
```

Parameters

target	Specifies the target of the query operation. *target* must be GL_RENDERBUFFER.
pname	Specifies the parameter whose value to retrieve from the renderbuffer bound to *target*.
params	Specifies the address of an array to receive the value of the queried parameter.

Description

glGetRenderbufferParameteriv retrieves information about a bound renderbuffer object. *target* specifies the target of the query operation and must be GL_RENDERBUFFER. *pname* specifies the parameter whose value to query and must be one of GL_RENDERBUFFER_WIDTH, GL_RENDERBUFFER_HEIGHT, GL_RENDERBUFFER_INTERNAL_FORMAT, GL_RENDERBUFFER_RED_SIZE, GL_RENDERBUFFER_GREEN_SIZE, GL_RENDERBUFFER_BLUE_SIZE, GL_RENDERBUFFER_ALPHA_SIZE, GL_RENDERBUFFER_DEPTH_SIZE, GL_RENDERBUFFER_STENCIL_SIZE, or GL_RENDERBUFFER_SAMPLES.

Upon a successful return from glGetRenderbufferParameteriv, if *pname* is GL_RENDERBUFFER_WIDTH, GL_RENDERBUFFER_HEIGHT, GL_RENDERBUFFER_INTERNAL_FORMAT, or GL_RENDERBUFFER_SAMPLES, then *params* will contain the width in pixels, the height in pixels, the internal format, or the number of samples, respectively, of the image of the renderbuffer currently bound to *target*.

If *pname* is GL_RENDERBUFFER_RED_SIZE, GL_RENDERBUFFER_GREEN_SIZE, GL_RENDERBUFFER_BLUE_SIZE, GL_RENDERBUFFER_ALPHA_SIZE, GL_RENDERBUFFER_DEPTH_SIZE, or GL_RENDERBUFFER_STENCIL_SIZE, then *params* will contain the actual resolutions (not the resolutions specified when the image array was defined) for the red, green, blue, alpha depth, or stencil components, respectively, of the image of the renderbuffer currently bound to *target*.

Errors

GL_INVALID_ENUM is generated if *pname* is not one of the accepted tokens.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetRenderbufferParameteriv	✔	✔

See Also

glGenRenderbuffers, glFramebufferRenderbuffer, glBindRenderbuffer, glRenderbufferStorage, glRenderbufferStorageMultisample

Copyright

Name

glGetSamplerParameter — return sampler parameter values

C Specification

```
void glGetSamplerParameterfv (sampler, pname, params);

GLuint sampler;
GLenum pname;
GLfloat * params;

void glGetSamplerParameteriv (sampler, pname, params);

GLuint sampler;
GLenum pname;
GLint * params;
```

Parameters

sampler	Specifies name of the sampler object from which to retrieve parameters.
pname	Specifies the symbolic name of a sampler parameter. GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MIN_LOD, GL_TEXTURE_MAX_LOD, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_WRAP_R, GL_TEXTURE_COMPARE_MODE, and GL_TEXTURE_COMPARE_FUNC are accepted.
params	Returns the sampler parameters.

Description

glGetSamplerParameter* returns in *params* the value or values of the sampler parameter specified as *pname*. *sampler* defines the target sampler, and must be the name of an existing sampler object, returned from a previous call to glGenSamplers. *pname* accepts the same symbols as glSamplerParameter*, with the same interpretations:

GL_TEXTURE_MAG_FILTER	Returns the single-valued texture magnification filter, a symbolic constant. The initial value is GL_LINEAR.
GL_TEXTURE_MIN_FILTER	Returns the single-valued texture minification filter, a symbolic constant. The initial value is GL_NEAREST_MIPMAP_LINEAR.
GL_TEXTURE_MIN_LOD	Returns the single-valued texture minimum level-of-detail value. The initial value is .
GL_TEXTURE_MAX_LOD	Returns the single-valued texture maximum level-of-detail value. The initial value is 1000.
GL_TEXTURE_WRAP_S	Returns the single-valued wrapping function for texture coordinate , a symbolic constant. The initial value is GL_REPEAT.
GL_TEXTURE_WRAP_T	Returns the single-valued wrapping function for texture coordinate , a symbolic constant. The initial value is GL_REPEAT.
GL_TEXTURE_WRAP_R	Returns the single-valued wrapping function for texture coordinate , a symbolic constant. The initial value is GL_REPEAT.

`GL_TEXTURE_COMPARE_MODE`	Returns a single-valued texture comparison mode, a symbolic constant. The initial value is `GL_NONE`. See glSamplerParameter.
`GL_TEXTURE_COMPARE_FUNC`	Returns a single-valued texture comparison function, a symbolic constant. The initial value is `GL_LEQUAL`. See glSamplerParameter.

Notes

If an error is generated, no change is made to the contents of *`params`*.

Errors

`GL_INVALID_OPERATION` is generated if *`sampler`* is not the name of a sampler object returned from a previous call to glGenSamplers.

`GL_INVALID_ENUM` is generated if *`pname`* is not an accepted value.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glGetSamplerParameterfv`	-	✔
`glGetSamplerParameteriv`	-	✔

See Also

glSamplerParameter, glGenSamplers, glDeleteSamplers, glSamplerParameter

Copyright

Name

glGetShaderInfoLog — Returns the information log for a shader object

C Specification

```
void glGetShaderInfoLog (shader, maxLength, length, infoLog);

GLuint shader;
GLsizei maxLength;
GLsizei *length;
GLchar *infoLog;
```

Parameters

shader	Specifies the shader object whose information log is to be queried.
maxLength	Specifies the size of the character buffer for storing the returned information log.
length	Returns the length of the string returned in *infoLog* (excluding the null terminator).
infoLog	Specifies an array of characters that is used to return the information log.

Description

`glGetShaderInfoLog` returns the information log for the specified shader object. The information log for a shader object is modified when the shader is compiled. The string that is returned will be null terminated.

`glGetShaderInfoLog` returns in *infoLog* as much of the information log as it can, up to a maximum of *maxLength* characters. The number of characters actually returned, excluding the null termination character, is specified by *length*. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument. The size of the buffer required to store the returned information log can be obtained by calling glGetShaderiv with the value `GL_INFO_LOG_LENGTH`.

The information log for a shader object is a string that may contain diagnostic messages, warning messages, and other information about the last compile operation. When a shader object is created, its information log will be a string of length 0.

Notes

The information log for a shader object is the OpenGL implementer's primary mechanism for conveying information about the compilation process. Therefore, the information log can be helpful to application developers during the development process, even when compilation is successful. Application developers should not expect different OpenGL implementations to produce identical information logs.

Errors

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_VALUE` is generated if *maxLength* is less than 0.

Associated Gets

glGetShaderiv with argument `GL_INFO_LOG_LENGTH`

glIsShader

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glGetShaderInfoLog	✔	✔

See Also

glCompileShader, glGetProgramInfoLog, glLinkProgram, glValidateProgram

Copyright

Name

glGetShaderPrecisionFormat — retrieve the range and precision for numeric formats supported by the shader compiler

C Specification

```
void glGetShaderPrecisionFormat (shaderType, precisionType, range, precision);

GLenum shaderType;
GLenum precisionType;
GLint *range;
GLint *precision;
```

Parameters

shaderType	Specifies the type of shader whose precision to query. *shaderType* must be `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`.
precisionType	Specifies the numeric format whose precision and range to query.
range	Specifies the address of array of two integers into which encodings of the implementation's numeric range are returned.
precision	Specifies the address of an integer into which the numeric precision of the implementation is written.

Description

`glGetShaderPrecisionFormat` retrieves the numeric range and precision for the implementation's representation of quantities in different numeric formats in specified shader type. *shaderType* specifies the type of shader for which the numeric precision and range is to be retrieved and must be one of `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`. *precisionType* specifies the numeric format to query and must be one of `GL_LOW_FLOAT`, `GL_MEDIUM_FLOAT` `GL_HIGH_FLOAT`, `GL_LOW_INT`, `GL_MEDIUM_INT`, or `GL_HIGH_INT`.

range points to an array of two integers into which the format's numeric range will be returned. If min and max are the smallest values representable in the format, then the values returned are defined to be: *range*[0] = floor(log2(|min|)) and *range*[1] = floor(log2(|max|)).

precision specifies the address of an integer into which will be written the log2 value of the number of bits of precision of the format. If the smallest representable value greater than 1 is 1 + *eps*, then the integer addressed by *precision* will contain floor(-log2(eps)).

Errors

`GL_INVALID_ENUM` is generated if *shaderType* or *precisionType* is not an accepted value.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetShaderPrecisionFormat	✔	✔

Copyright

Name

glGetShaderSource — Returns the source code string from a shader object

C Specification

```
void glGetShaderSource (shader, bufSize, length, source);

GLuint shader;
GLsizei bufSize;
GLsizei *length;
GLchar *source;
```

Parameters

shader	Specifies the shader object to be queried.
bufSize	Specifies the size of the character buffer for storing the returned source code string.
length	Returns the length of the string returned in *source* (excluding the null terminator).
source	Specifies an array of characters that is used to return the source code string.

Description

`glGetShaderSource` returns the concatenation of the source code strings from the shader object specified by *shader*. The source code strings for a shader object are the result of a previous call to glShaderSource. The string returned by the function will be null terminated.

`glGetShaderSource` returns in *source* as much of the source code string as it can, up to a maximum of *bufSize* characters. The number of characters actually returned, excluding the null termination character, is specified by *length*. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument. The size of the buffer required to store the returned source code string can be obtained by calling glGetShaderiv with the value `GL_SHADER_SOURCE_LENGTH`.

Errors

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_VALUE` is generated if *bufSize* is less than 0.

Associated Gets

glGetShaderiv with argument `GL_SHADER_SOURCE_LENGTH`

glIsShader

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetShaderSource	✔	✔

See Also

glCreateShader, glShaderSource

Copyright

Name

glGetShaderiv — Returns a parameter from a shader object

C Specification

```
void glGetShaderiv (shader, pname, params);

GLuint shader;
GLenum pname;
GLint *params;
```

Parameters

shader	Specifies the shader object to be queried.
pname	Specifies the object parameter. Accepted symbolic names are `GL_SHADER_TYPE`, `GL_DELETE_STATUS`, `GL_COMPILE_STATUS`, `GL_INFO_LOG_LENGTH`, `GL_SHADER_SOURCE_LENGTH`.
params	Returns the requested object parameter.

Description

`glGetShaderiv` returns in *params* the value of a parameter for a specific shader object. The following parameters are defined:

`GL_SHADER_TYPE`	*params* returns `GL_VERTEX_SHADER` if *shader* is a vertex shader object, and `GL_FRAGMENT_SHADER` if *shader* is a fragment shader object.
`GL_DELETE_STATUS`	*params* returns `GL_TRUE` if *shader* is currently flagged for deletion, and `GL_FALSE` otherwise.
`GL_COMPILE_STATUS`	*params* returns `GL_TRUE` if the last compile operation on *shader* was successful, and `GL_FALSE` otherwise.
`GL_INFO_LOG_LENGTH`	*params* returns the number of characters in the information log for *shader* including the null termination character (i.e., the size of the character buffer required to store the information log). If *shader* has no information log, a value of 0 is returned.
`GL_SHADER_SOURCE_LENGTH`	*params* returns the length of the concatenation of the source strings that make up the shader source for the *shader*, including the null termination character. (i.e., the size of the character buffer required to store the shader source). If no source code exists, 0 is returned.

Notes

If an error is generated, no change is made to the contents of *params*.

Errors

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`shader`* does not refer to a shader object.

`GL_INVALID_ENUM` is generated if *`pname`* is not an accepted value.

Associated Gets

glGetShaderInfoLog with argument *`shader`*

glGetShaderSource with argument *`shader`*

glIsShader

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetShaderiv	✔	✔

See Also

glCompileShader, glCreateShader, glDeleteShader, glGetProgramiv, glShaderSource

Copyright

Name

glGetString — return a string describing the current GL connection

C Specification

```
const GLubyte* glGetString (name);

GLenum name;

const GLubyte* glGetStringi (name, index);

GLenum name;
GLuint index;
```

Parameters

name	Specifies a symbolic constant, one of `GL_EXTENSIONS`, `GL_RENDERER`, `GL_SHADING_LANGUAGE_VERSION`, `GL_VENDOR`, or `GL_VERSION`. `glGetStringi` accepts only the `GL_EXTENSIONS` token.
index	For `glGetStringi`, specifies the index of the string to return.

Description

`glGetString` returns a pointer to a static string describing some aspect of the current GL connection. *name* can be one of the following:

`GL_EXTENSIONS`	Returns the extension string supported by the implementation.
`GL_VENDOR`	Returns the company responsible for this GL implementation. This name does not change from release to release.
`GL_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.
`GL_VERSION`	Returns a version or release number.
`GL_SHADING_LANGUAGE_VERSION`	Returns a version or release number for the shading language.

`glGetStringi` returns a pointer to a static string indexed by *index*. *name* can be one of the following:

`GL_EXTENSIONS`	Returns the extension string supported by the implementation at *index*.

Strings `GL_VENDOR` and `GL_RENDERER` together uniquely specify a platform. They do not change from release to release and should be used by platform-recognition algorithms.

The `GL_VERSION` and `GL_SHADING_LANGUAGE_VERSION` strings begin with a version number. The version number uses one of these forms:

major_number.minor_number major_number.minor_number.release_number

Vendor-specific information may follow the version number. Its format depends on the implementation, but a space always separates the version number and the vendor-specific information.

All strings are null-terminated.

Notes

If an error is generated, `glGetString` returns 0.

The client and server may support different versions. `glGetString` always returns a compatible version number. The release number always describes the server.

There is no defined relationship between the order in which extension names appear in the non-indexed string and the order in which they appear in the indexed query.

There is no defined relationship between any particular extension name and the index values; an extension name may correspond to a different index in different GL contexts and/or implementations.

Errors

`GL_INVALID_ENUM` is generated if *`name`* is not an accepted value.

`GL_INVALID_VALUE` is generated by `glGetStringi` if *`index`* is outside the valid range for indexed state *`name`*.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`GetString`	✔	✔
`GetStringi`	-	✔

Copyright

Name

glGetSynciv — query the properties of a sync object

C Specification

```
void glGetSynciv (sync, pname, bufSize, length, values);

GLsync sync;
GLenum pname;
GLsizei bufSize;
GLsizei *length;
GLint *values;
```

Parameters

sync	Specifies the sync object whose properties to query.
pname	Specifies the parameter whose value to retrieve from the sync object specified in *sync*.
bufSize	Specifies the size of the buffer whose address is given in *values*.
length	Specifies the address of an variable to receive the number of integers placed in *values*.
values	Specifies the address of an array to receive the values of the queried parameter.

Description

`glGetSynciv` retrieves properties of a sync object. *sync* specifies the name of the sync object whose properties to retrieve.

On success, `glGetSynciv` replaces up to *bufSize* integers in *values* with the corresponding property values of the object being queried. The actual number of integers replaced is returned in the variable whose address is specified in *length*. If *length* is `NULL`, no length is returned.

If *pname* is `GL_OBJECT_TYPE`, a single value representing the specific type of the sync object is placed in *values*. The only type supported is `GL_SYNC_FENCE`.

If *pname* is `GL_SYNC_STATUS`, a single value representing the status of the sync object (`GL_SIGNALED` or `GL_UNSIGNALED`) is placed in *values*.

If *pname* is `GL_SYNC_CONDITION`, a single value representing the condition of the sync object is placed in *values*. The only condition supported is `GL_SYNC_GPU_COMMANDS_COMPLETE`.

If *pname* is `GL_SYNC_FLAGS`, a single value representing the flags with which the sync object was created is placed in *values*. No flags are currently supported[1].

If an error occurs, nothing will be written to *values* or *length*.

Errors

`GL_INVALID_VALUE` is generated if *sync* is not the name of a sync object.

[1]*flags* is expected to be used in future extensions to the sync objects.

`GL_INVALID_ENUM` is generated if *`pname`* is not one of the accepted tokens.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetSynciv	-	✔

See Also

glFenceSync, glWaitSync, glClientWaitSync

Copyright

Name

glGetTexParameter — return texture parameter values

C Specification

```
void glGetTexParameterfv (target, pname, params);

GLenum target;
GLenum pname;
GLfloat * params;

void glGetTexParameteriv (target, pname, params);

GLenum target;
GLenum pname;
GLint * params;
```

Parameters

target	Specifies the symbolic name of the target texture. GL_TEXTURE_2D, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_3D, and GL_TEXTURE_CUBE_MAP are accepted.
pname	Specifies the symbolic name of a texture parameter. GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_COMPARE_FUNC, GL_TEXTURE_COMPARE_MODE, GL_TEXTURE_IMMUTABLE_FORMAT, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_MAX_LOD, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MIN_LOD, GL_TEXTURE_SWIZZLE_R, GL_TEXTURE_SWIZZLE_G, GL_TEXTURE_SWIZZLE_B, GL_TEXTURE_SWIZZLE_A, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, and GL_TEXTURE_WRAP_R are accepted.
params	Returns the texture parameters.

Description

glGetTexParameter returns in *params* the value or values of the texture parameter specified as *pname*. *target* defines the target texture. GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, and GL_TEXTURE_CUBE_MAP specify two- or three-dimensional, two-dimensional array, or cube-mapped texturing, respectively. *pname* accepts the same symbols as glTexParameter, with the same interpretations:

GL_TEXTURE_BASE_LEVEL	Returns the single-valued base texture mipmap level. The initial value is 0.
GL_TEXTURE_COMPARE_FUNC	Returns a single-valued texture comparison function, a symbolic constant. The initial value is GL_LEQUAL. See glTexParameter.
GL_TEXTURE_COMPARE_MODE	Returns a single-valued texture comparison mode, a symbolic constant. The initial value is GL_NONE. See glTexParameter.
GL_TEXTURE_IMMUTABLE_FORMAT	Returns a single-value boolean representing the immutability of the texture format and size. initial value is GL_FALSE. See glTexStorage2D.

`GL_TEXTURE_MAG_FILTER`	Returns the single-valued texture magnification filter, a symbolic constant. The initial value is `GL_LINEAR`.
`GL_TEXTURE_MAX_LEVEL`	Returns the single-valued maximum texture mipmap array level. The initial value is 1000.
`GL_TEXTURE_MAX_LOD`	Returns the single-valued texture maximum level-of-detail value. The initial value is 1000.
`GL_TEXTURE_MIN_FILTER`	Returns the single-valued texture minification filter, a symbolic constant. The initial value is `GL_NEAREST_MIPMAP_LINEAR`.
`GL_TEXTURE_MIN_LOD`	Returns the single-valued texture minimum level-of-detail value. The initial value is .
`GL_TEXTURE_SWIZZLE_R`	Returns the red component swizzle. The initial value is `GL_RED`.
`GL_TEXTURE_SWIZZLE_G`	Returns the green component swizzle. The initial value is `GL_GREEN`.
`GL_TEXTURE_SWIZZLE_B`	Returns the blue component swizzle. The initial value is `GL_BLUE`.
`GL_TEXTURE_SWIZZLE_A`	Returns the alpha component swizzle. The initial value is `GL_ALPHA`.
`GL_TEXTURE_WRAP_S`	Returns the single-valued wrapping function for texture coordinate , a symbolic constant. The initial value is `GL_REPEAT`.
`GL_TEXTURE_WRAP_T`	Returns the single-valued wrapping function for texture coordinate , a symbolic constant. The initial value is `GL_REPEAT`.
`GL_TEXTURE_WRAP_R`	Returns the single-valued wrapping function for texture coordinate , a symbolic constant. The initial value is `GL_REPEAT`.

Notes

If an error is generated, no change is made to the contents of *`params`*.

Errors

`GL_INVALID_ENUM` is generated if *`target`* or *`pname`* is not an accepted value.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glGetTexParameterfv`	✔	✔
`glGetTexParameteriv`	✔	✔

See Also

glTexParameter, glTexStorage2D

Copyright

Name

glGetTransformFeedbackVarying — retrieve information about varying variables selected for transform feedback

C Specification

```
void glGetTransformFeedbackVarying (program, index, bufSize, length,
size, type, name);
```

```
GLuint program;
GLuint index;
GLsizei bufSize;
GLsizei * length;
GLsizei * size;
GLenum * type;
char * name;
```

Parameters

program	The name of the target program object.
index	The index of the varying variable whose information to retrieve.
bufSize	The maximum number of characters, including the null terminator, that may be written into *name*.
length	The address of a variable which will receive the number of characters written into *name*, excluding the null-terminator. If *length* is `NULL` no length is returned.
size	The address of a variable that will receive the size of the varying.
type	The address of a variable that will receive the type of the varying.
name	The address of a buffer into which will be written the name of the varying.

Description

Information about the set of varying variables in a linked program that will be captured during transform feedback may be retrieved by calling `glGetTransformFeedbackVarying`. `glGetTransformFeedbackVarying` provides information about the varying variable selected by *index*. An *index* of 0 selects the first varying variable specified in the *varyings* array passed to glTransformFeedbackVaryings, and an *index* of `GL_TRANSFORM_FEEDBACK_VARYINGS-1` selects the last such variable.

The name of the selected varying is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is NULL, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *bufSize*.

The length of the longest varying name in program is given by `GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`, which can be queried with glGetProgramiv.

For the selected varying variable, its type is returned into *type*. The size of the varying is returned into *size*. The value in *size* is in units of the type returned in *type*. The type returned can be any of the scalar, vector, or matrix attribute types returned by glGetActiveAttrib. If an error occurred, the return parameters *`length`*, *`size`*, *`type`* and *`name`* will be unmodified. This command will return as much information about the varying variables as possible. If no information is available, *`length`* will be set to zero and *`name`* will be an empty string. This situation could arise if `glGetTransformFeedbackVarying` is called after a failed link.

Errors

`GL_INVALID_VALUE` is generated if *`program`* is not the name of a program object.

`GL_INVALID_VALUE` is generated if *`index`* is greater or equal to the value of `GL_TRANSFORM_FEEDBACK_VARYINGS`.

`GL_INVALID_OPERATION` is generated *`program`* has not been linked.

Associated Gets

glGetProgramiv with argument `GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH` or `GL_TRANSFORM_FEEDBACK_VARYINGS`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetTransformFeedbackVarying	-	✔

See Also

glBeginTransformFeedback, glEndTransformFeedback, glTransformFeedbackVaryings, glGetProgramiv

Copyright

Name

glGetUniform — Returns the value of a uniform variable

C Specification

```
void glGetUniformfv (program, location, params);

GLuint program;
GLint location;
GLfloat *params;

void glGetUniformiv (program, location, params);

GLuint program;
GLint location;
GLint *params;

void glGetUniformuiv (program, location, params);

GLuint program;
GLint location;
GLuint *params;
```

Parameters

program	Specifies the program object to be queried.
location	Specifies the location of the uniform variable to be queried.
params	Returns the value of the specified uniform variable.

Description

`glGetUniform` returns in *params* the value(s) of the specified uniform variable. The type of the uniform variable specified by *location* determines the number of values returned. If the uniform variable is defined in the shader as a boolean, int, unsigned int, or float, a single value will be returned. If it is defined as a vec2, ivec2, uvec2, or bvec2, two values will be returned. If it is defined as a vec3, ivec3, uvec3, or bvec3, three values will be returned, and so on. To query values stored in uniform variables declared as arrays, call `glGetUniform` for each element of the array. To query values stored in uniform variables declared as structures, call `glGetUniform` for each field in the structure. The values for uniform variables declared as a matrix will be returned in column major order.

The locations assigned to uniform variables are not known until the program object is linked. After linking has occurred, the command glGetUniformLocation can be used to obtain the location of a uniform variable. This location value can then be passed to `glGetUniform` in order to query the current value of the uniform variable. After a program object has been linked successfully, the index values for uniform variables remain fixed until the next link command occurs. The uniform variable values can only be queried after a link if the link was successful.

Notes

If an error is generated, no change is made to the contents of *params*.

Errors

`GL_INVALID_VALUE` is generated if *`program`* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`program`* is not a program object.

`GL_INVALID_OPERATION` is generated if *`program`* has not been successfully linked.

`GL_INVALID_OPERATION` is generated if *`location`* does not correspond to a valid uniform variable location for the specified program object.

Associated Gets

glGetActiveUniform with arguments *`program`* and the index of an active uniform variable

glGetProgramiv with arguments *`program`* and `GL_ACTIVE_UNIFORMS` or `GL_ACTIVE_UNIFORM_MAX_LENGTH`

glGetUniformLocation with arguments *`program`* and the name of a uniform variable

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glGetUniformfv`	✔	✔
`glGetUniformiv`	✔	✔
`glGetUniformuiv`	-	✔

See Also

glCreateProgram, glLinkProgram, glUniform

Copyright

Name

glGetUniformBlockIndex — retrieve the index of a named uniform block

C Specification

```
GLuint glGetUniformBlockIndex (program, uniformBlockName);

GLuint program;
const GLchar *uniformBlockName;
```

Parameters

program	Specifies the name of a program containing the uniform block.
uniformBlockName	Specifies the address an array of characters containing the name of the uniform block whose index to retrieve.

Description

`glGetUniformBlockIndex` retrieves the index of a uniform block within *program*.

program must be the name of a program object for which the command glLinkProgram must have been called in the past, although it is not required that glLinkProgram must have succeeded. The link could have failed because the number of active uniforms exceeded the limit.

uniformBlockName must contain a nul-terminated string specifying the name of the uniform block.

`glGetUniformBlockIndex` returns the uniform block index for the uniform block named *uniformBlockName* of *program*. If *uniformBlockName* does not identify an active uniform block of *program*, `glGetUniformBlockIndex` returns the special identifier, `GL_INVALID_INDEX`. Indices of the active uniform blocks of a program are assigned in consecutive order, beginning with zero.

Errors

`GL_INVALID_OPERATION` is generated if *program* is not the name of a program object for which glLinkProgram has been called in the past.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetUniformBlockIndex	-	✔

See Also

glGetActiveUniformBlockName, glGetActiveUniformBlockiv, glLinkProgram

Copyright

Name

glGetUniformIndices — retrieve the index of a named uniform block

C Specification

```
void glGetUniformIndices (program, uniformCount, uniformNames, uniformIndices);

GLuint program;
GLsizei uniformCount;
const GLchar **uniformNames;
GLuint *uniformIndices;
```

Parameters

program	Specifies the name of a program containing uniforms whose indices to query.
uniformCount	Specifies the number of uniforms whose indices to query.
uniformNames	Specifies the address of an array of pointers to buffers containing the names of the queried uniforms.
uniformIndices	Specifies the address of an array that will receive the indices of the uniforms.

Description

glGetUniformIndices retrieves the indices of a number of uniforms within *program*.

program must be the name of a program object for which the command glLinkProgram must have been called in the past, although it is not required that glLinkProgram must have succeeded. The link could have failed because the number of active uniforms exceeded the limit.

uniformCount indicates both the number of elements in the array of names *uniformNames* and the number of indices that may be written to *uniformIndices*.

uniformNames contains a list of *uniformCount* nul-terminated name strings identifying the uniform names to be queried for indices. For each name string in *uniformNames*, the index assigned to the active uniform of that name will be written to the corresponding element of *uniformIndices*. If a string in *uniformNames* is not the name of an active uniform, the special value GL_INVALID_INDEX will be written to the corresponding element of *uniformIndices*.

If an error occurs, nothing is written to *uniformIndices*.

Errors

GL_INVALID_OPERATION is generated if *program* is not the name of a program object for which glLinkProgram has been called in the past.

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glGetUniformIndices	-	✔

See Also

glGetActiveUniform, glGetActiveUniformsiv, glLinkProgram

Copyright

Name

glGetUniformLocation — Returns the location of a uniform variable

C Specification

```
GLint glGetUniformLocation (program, name);

GLuint program;
const GLchar *name;
```

Parameters

program	Specifies the program object to be queried.
name	Points to a null terminated string containing the name of the uniform variable whose location is to be queried.

Description

`glGetUniformLocation` returns an integer that represents the location of a specific uniform variable within a the default uniform block of a program object. *name* must be a null terminated string that contains no white space. *name* must be an active uniform variable name in *program* that is not a structure, an array of structures, or a subcomponent of a vector or a matrix. This function returns -1 if *name* does not correspond to an active uniform variable in *program* or if *name* is associated with a named uniform block.

Uniform variables that are structures or arrays of structures may be queried by calling `glGetUniform-Location` for each field within the structure. The array element operator "[]" and the structure field operator "." may be used in *name* in order to select elements within an array or fields within a structure. The result of using these operators is not allowed to be another structure, an array of structures, or a sub-component of a vector or a matrix. The first element of a uniform array is identified using the name of the uniform array appended with "[0]". If the last part of the string name indicates a uniform array, then the location of the first element of that array can be retrieved by either using the name of the array, or by using the name appended by "[0]".

Locations for sequential array indices are not required to be sequential. The location for "a[1]" may or may not be equal to the location for "a[0]" + 1. Furthermore, since unused elements at the end of uniform arrays may be trimmed the location of the i + 1 array element may not be valid even if the location of the i element is valid. As a direct consequence, the value of the location of "a[0]" + 1 may refer to a different uniform entirely. Applications that wish to set individual array elements should query the locations of each element separately.

The actual locations assigned to uniform variables are not known until the program object is linked successfully. After linking has occurred, the command `glGetUniformLocation` can be used to obtain the location of a uniform variable. This location value can then be passed to glUniform to set the value of the uniform variable or to glGetUniform in order to query the current value of the uniform variable. After a program object has been linked successfully, the index values for uniform variables remain fixed until the next link command occurs. Uniform variable locations and values can only be queried after a link if the link was successful.

Errors

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`program`* is not a program object.

`GL_INVALID_OPERATION` is generated if *`program`* has not been successfully linked.

Associated Gets

glGetActiveUniform with arguments *`program`* and the index of an active uniform variable

glGetProgramiv with arguments *`program`* and `GL_ACTIVE_UNIFORMS` or `GL_ACTIVE_UNIFOR-M_MAX_LENGTH`

glGetUniform with arguments *`program`* and the name of a uniform variable

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetUniformLocation	✔	✔

See Also

glLinkProgram, glUniform

Copyright

Name

glGetVertexAttrib — Return a generic vertex attribute parameter

C Specification

```
void glGetVertexAttribfv (index, pname, params);

GLuint index;
GLenum pname;
GLfloat *params;

void glGetVertexAttribiv (index, pname, params);

GLuint index;
GLenum pname;
GLint *params;

void glGetVertexAttribIiv (index, pname, params);

GLuint index;
GLenum pname;
GLint *params;

void glGetVertexAttribIuiv (index, pname, params);

GLuint index;
GLenum pname;
GLuint *params;
```

Parameters

index	Specifies the generic vertex attribute parameter to be queried.
pname	Specifies the symbolic name of the vertex attribute parameter to be queried. Accepted values are `GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`, `GL_VERTEX_ATTRIB_ARRAY_ENABLED`, `GL_VERTEX_ATTRIB_ARRAY_SIZE`, `GL_VERTEX_ATTRIB_ARRAY_STRIDE`, `GL_VERTEX_ATTRIB_ARRAY_TYPE`, `GL_VERTEX_ATTRIB_ARRAY_NORMALIZED`, `GL_VERTEX_ATTRIB_ARRAY_INTEGER`, `GL_VERTEX_ATTRIB_ARRAY_DIVISOR`, or `GL_CURRENT_VERTEX_ATTRIB`.
params	Returns the requested data.

Description

`glGetVertexAttrib` returns in *params* the value of a generic vertex attribute parameter. The generic vertex attribute to be queried is specified by *index*, and the parameter to be queried is specified by *pname*.

The accepted parameter names are as follows:

`GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`
: *params* returns a single value, the name of the buffer object currently bound to the binding point corresponding to generic vertex

attribute array *index*. If no buffer object is bound, 0 is returned. The initial value is 0.

GL_VERTEX_ATTRIB_ARRAY_ENABLED	*params* returns a single value that is non-zero (true) if the vertex attribute array for *index* is enabled and 0 (false) if it is disabled. The initial value is GL_FALSE.
GL_VERTEX_ATTRIB_ARRAY_SIZE	*params* returns a single value, the size of the vertex attribute array for *index*. The size is the number of values for each element of the vertex attribute array, and it will be 1, 2, 3, or 4. The initial value is 4.
GL_VERTEX_ATTRIB_ARRAY_STRIDE	*params* returns a single value, the array stride for (number of bytes between successive elements in) the vertex attribute array for *index*. A value of 0 indicates that the array elements are stored sequentially in memory. The initial value is 0.
GL_VERTEX_ATTRIB_ARRAY_TYPE	*params* returns a single value, a symbolic constant indicating the array type for the vertex attribute array for *index*. Possible values are GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_INT_2_10_10_10_REV, GL_UNSIGNED_INT, GL_FIXED, GL_HALF_FLOAT, and GL_FLOAT. The initial value is GL_FLOAT.
GL_VERTEX_ATTRIB_ARRAY_NORMALIZED	*params* returns a single value that is non-zero (true) if fixed-point data types for the vertex attribute array indicated by *index* are normalized when they are converted to floating point, and 0 (false) otherwise. The initial value is GL_FALSE.
GL_VERTEX_ATTRIB_ARRAY_INTEGER	*params* returns a single value that is non-zero (true) if fixed-point data types for the vertex attribute array indicated by *index* have integer data types, and 0 (false) otherwise. The initial value is 0 (GL_FALSE).
GL_VERTEX_ATTRIB_ARRAY_DIVISOR	*params* returns a single value that is the frequency divisor used for instanced rendering. See glVertexAttribDivisor. The initial value is 0.
GL_CURRENT_VERTEX_ATTRIB	*params* returns four values that represent the current value for the generic vertex attribute specified by index. The initial value for all generic vertex attributes is (0,0,0,1).

All of the parameters except GL_CURRENT_VERTEX_ATTRIB represent state stored in the currently bound vertex array object. If the zero object is bound, these values are client state.

Notes

If an error is generated, no change is made to the contents of *params*.

Errors

`GL_INVALID_VALUE` is generated if *`index`* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`GL_INVALID_ENUM` is generated if *`pname`* is not an accepted value.

Associated Gets

glGet with argument `GL_MAX_VERTEX_ATTRIBS`

glGetVertexAttribPointerv with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_POINTER`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glGetVertexAttribfv`	✔	✔
`glGetVertexAttribiv`	✔	✔
`glGetVertexAttribIiv`	-	✔
`glGetVertexAttribIuiv`	-	✔

See Also

glBindAttribLocation, glBindBuffer, glDisableVertexAttribArray, glEnableVertexAttribArray, glVertex-Attrib, glVertexAttribDivisor, glVertexAttribPointer

Copyright

Name

glGetVertexAttribPointerv — return the address of the specified generic vertex attribute pointer

C Specification

```
void glGetVertexAttribPointerv (index, pname, pointer);

GLuint index;
GLenum pname;
void **pointer;
```

Parameters

index	Specifies the generic vertex attribute parameter to be returned.
pname	Specifies the symbolic name of the generic vertex attribute parameter to be returned. Must be `GL_VERTEX_ATTRIB_ARRAY_POINTER`.
pointer	Returns the pointer value.

Description

`glGetVertexAttribPointerv` returns pointer information. *index* is the generic vertex attribute to be queried, *pname* is a symbolic constant indicating the pointer to be returned, and *params* is a pointer to a location in which to place the returned data.

The *pointer* returned is a byte offset into the data store of the buffer object that was bound to the `GL_ARRAY_BUFFER` target (see glBindBuffer) when the desired pointer was previously specified.

Notes

The state returned is retrieved from the currently bound vertex array object. If the zero object is bound, the value is queried from client state.

The initial value for each pointer is 0.

Errors

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`GL_INVALID_ENUM` is generated if *pname* is not an accepted value.

Associated Gets

glGet with argument `GL_MAX_VERTEX_ATTRIBS`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glGetVertexAttribPointerv	✔	✔

See Also

glGetVertexAttrib, glVertexAttribPointer

Copyright

Name

glHint — specify implementation-specific hints

C Specification

```
void glHint (target, mode);

GLenum target;
GLenum mode;
```

Parameters

target	Specifies a symbolic constant indicating the behavior to be controlled. `GL_FRAGMENT_SHADER_DERIVATIVE_HINT`, and `GL_GENERATE_MIPMAP_HINT` are accepted.
mode	Specifies a symbolic constant indicating the desired behavior. `GL_FASTEST`, `GL_NICEST`, and `GL_DONT_CARE` are accepted.

Description

Certain aspects of GL behavior, when there is room for interpretation, can be controlled with hints. A hint is specified with two arguments. *target* is a symbolic constant indicating the behavior to be controlled, and *mode* is another symbolic constant indicating the desired behavior. The initial value for each *target* is `GL_DONT_CARE`. *mode* can be one of the following:

`GL_FASTEST`	The most efficient option should be chosen.
`GL_NICEST`	The most correct, or highest quality, option should be chosen.
`GL_DONT_CARE`	No preference.

Though the implementation aspects that can be hinted are well defined, the interpretation of the hints depends on the implementation. The hint aspects that can be specified with *target*, along with suggested semantics, are as follows:

`GL_FRAGMENT_SHADER_DERIVATIVE_HINT`	Indicates the accuracy of the derivative calculation for the GL shading language fragment processing built-in functions: `dFdx`, `dFdy`, and `fwidth`.
`GL_GENERATE_MIPMAP_HINT`	Indicates the quality of filtering when generating mipmap images with glGenerateMipmap.

Notes

The interpretation of hints depends on the implementation. Some implementations ignore `glHint` settings.

Errors

`GL_INVALID_ENUM` is generated if either *target* or *mode* is not an accepted value.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glHint	✔	✔

Copyright

Name

glInvalidateFramebuffer — Invalidate the contents of attachments within a framebuffer

C Specification

```
void glInvalidateFramebuffer (target, numAttachments, attachments);

GLenum target;
GLsizei numAttachments;
const GLenum *attachments;
```

Parameters

target	Specifies the target of the invalidate operation.
numAttachments	Specifies how many attachments are supplied in the *attachments* list.
attachments	A list of *numAttachments* attachments to invalidate.

Description

glInvalidateFramebuffer signals to the GL that it need not preserve all pixels of the framebuffer bound to *target*. *target* must be GL_READ_FRAMEBFUFFER, GL_DRAW_FRAMEBUFFER or GL_FRAMEBUFFER. The token GL_FRAMEBUFFER is treated as GL_DRAW_FRAMEBUFFER. *attachments* contains a list of *numAttachments* to be invalidated. If an attachment is specified that does not exist in the bound framebuffer, it is ignored.

If a framebuffer object is bound, then *attachments* may contain GL_COLOR_ATTACHMENTi, GL_DEPTH_ATTACHMENT, GL_STENCIL_ATTACHMENT, and/or GL_DEPTH_STENCIL_ATTACHMENT. If the framebuffer object is not complete, glInvalidateFramebuffer may be ignored.

If the default framebuffer is bound, then *attachments* may contain GL_COLOR, identifying the color buffer; GL_DEPTH, identifying the depth buffer; and/or GL_STENCIL, identifying the stencil buffer.

Notes

The intention of this function is to provide a hint to the GL implementation that there is no longer a need to preserve the contents of particular attachments of a framebuffer object, or the default framebuffer. It is possible, for example, to signal the intention that depth and or stencil data is no longer needed at the end of a scene, or that multisample color buffer data is no longer needed after a resolve through glBlitFramebuffer.

Errors

GL_INVALID_ENUM is generated if *target* is not GL_DRAW_FRAMEBUFFER, GL_READ_FRAMEBUFFER, or GL_FRAMEBUFFER.

GL_INVALID_OPERATION is generated if *attachments* contains GL_COLOR_ATTACHMENTm and m is greater than or equal to the value of GL_MAX_COLOR_ATTACHMENTS.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glInvalidateFramebuffer	-	✔

See Also

glBindFramebuffer, glBlitFramebuffer glFramebufferRenderbuffer, glFramebufferTexture2D, glFramebufferTextureLayer, glInvalidateSubFramebuffer

Copyright

Name

glInvalidateSubFramebuffer — Invalidate portions of the contents of attachments within a framebuffer

C Specification

```
void glInvalidateSubFramebuffer (target, numAttachments, attachments,
x, y, width, height);

GLenum target;
GLsizei numAttachments;
const GLenum *attachments;
GLintx;
GLinty;
GLsizei width;
GLsizei height;
```

Parameters

target	Specifies the target of the invalidate operation.
numAttachments	Specifies how many attachments are supplied in the *attachments* list.
attachments	A list of *numAttachments* attachments to invalidate.
x	Specifies the left origin of the pixel rectangle to invalidate, with lower left hand corner at (0,0).
y	Specifies the bottom origin of the pixel rectangle to invalidate, with lower left hand corner at (0,0).
width	Specifies the width of the pixel rectangle to invalidate.
height	Specifies the height of the pixel rectangle to invalidate.

Description

`glInvalidateSubFramebuffer` signals to the GL that it need not preserve all pixels of a specified region of the framebuffer bound to *target*. *target* must be `GL_READ_FRAMEBFUFFER`, `GL_DRAW_FRAMEBUFFER` or `GL_FRAMEBUFFER`. The token `GL_FRAMEBUFFER` is treated as `GL_DRAW_FRAMEBUFFER`. *attachments* contains a list of *numAttachments* to be invalidated. If an attachment is specified that does not exist in the bound framebuffer, it is ignored. *x*, *y*, *width* and *height* specify the bounds of the pixel rectangle to invalidate. Any of these pixels lying outside of the window allocated to the current GL context, or outside of the image attached to the currently bound framebuffer object, are ignored.

If a framebuffer object is bound, then *attachments* may contain `GL_COLOR_ATTACHMENTi`, `GL_DEPTH_ATTACHMENT`, `GL_STENCIL_ATTACHMENT`, and/or `GL_DEPTH_STENCIL_ATTACHMENT`. If the framebuffer object is not complete, `glInvalidateSubFramebuffer` may be ignored.

If the default framebuffer is bound, then *attachments* may contain `GL_COLOR`, identifying the color buffer; `GL_DEPTH`, identifying the depth buffer; and/or `GL_STENCIL`, identifying the stencil buffer.

Notes

The intention of this function is to provide a hint to the GL implementation that there is no longer a need to preserve the contents of particular attachments of a framebuffer object, or the default framebuffer. It is possible, for example, to signal the intention that depth and or stencil data is no longer needed at the end of a scene, or that multisample color buffer data is no longer needed after a resolve through glBlitFramebuffer.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, or `GL_FRAMEBUFFER`.

`GL_INVALID_OPERATION` is generated if *`attachments`* contains `GL_COLOR_ATTACHMENTm` and m is greater than or equal to the value of `GL_MAX_COLOR_ATTACHMENTS`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glInvalidateSubFramebuffer	-	✔

See Also

glBindFramebuffer, glBlitFramebuffer glFramebufferRenderbuffer, glFramebufferTexture2D, glFramebufferTextureLayer, glInvalidateFramebuffer

Copyright

Name

glIsBuffer — determine if a name corresponds to a buffer object

C Specification

```
GLboolean glIsBuffer (buffer);

GLuint buffer;
```

Parameters

buffer Specifies a value that may be the name of a buffer object.

Description

`glIsBuffer` returns `GL_TRUE` if *buffer* is currently the name of a buffer object. If *buffer* is zero, or is a non-zero value that is not currently the name of a buffer object, or if an error occurs, `glIsBuffer` returns `GL_FALSE`.

A name returned by glGenBuffers, but not yet associated with a buffer object by calling glBindBuffer, is not the name of a buffer object.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsBuffer	✔	✔

See Also

glBindBuffer, glDeleteBuffers, glGenBuffers, glGet

Copyright

Name

glIsEnabled — test whether a capability is enabled

C Specification

```
GLboolean glIsEnabled (cap);

GLenum cap;
```

Parameters

cap	Specifies a symbolic constant indicating a GL capability.
index	Specifies the index of the capability.

Description

`glIsEnabled` returns `GL_TRUE` if *cap* is an enabled capability and returns `GL_FALSE` otherwise. Initially all capabilities except `GL_DITHER` are disabled; `GL_DITHER` is initially enabled.

The following capabilities are accepted for *cap*:

Constant	**See**
`GL_BLEND`	glBlendFunc
`GL_CULL_FACE`	glCullFace
`GL_DEPTH_TEST`	glDepthFunc, glDepthRangef
`GL_DITHER`	glEnable
`GL_POLYGON_OFFSET_FILL`	glPolygonOffset
`GL_PRIMITIVE_RESTART_FIXED_INDEX`	glEnable
`GL_RASTERIZER_DISCARD`	glEnable
`GL_SAMPLE_ALPHA_TO_COVERAGE`	glSampleCoverage
`GL_SAMPLE_COVERAGE`	glSampleCoverage
`GL_SCISSOR_TEST`	glScissor
`GL_STENCIL_TEST`	glStencilFunc, glStencilOp

Notes

If an error is generated, `glIsEnabled` returns `GL_FALSE`.

Errors

`GL_INVALID_ENUM` is generated if *cap* is not an accepted value.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glIsEnabled	✔	✔

See Also

glEnable, glDisable, glGet

Copyright

Name

glIsFramebuffer — determine if a name corresponds to a framebuffer object

C Specification

```
GLboolean glIsFramebuffer (framebuffer);
```

```
GLuint framebuffer;
```

Parameters

framebuffer Specifies a value that may be the name of a framebuffer object.

Description

`glIsFramebuffer` returns `GL_TRUE` if *framebuffer* is currently the name of a framebuffer object. If *framebuffer* is zero, or if `framebuffer` is not the name of a framebuffer object, or if an error occurs, `glIsFramebuffer` returns `GL_FALSE`. If *framebuffer* is a name returned by glGenFramebuffers, by that has not yet been bound through a call to glBindFramebuffer, then the name is not a framebuffer object and `glIsFramebuffer` returns `GL_FALSE`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsFramebuffer	✔	✔

See Also

glGenFramebuffers, glBindFramebuffer, glDeleteFramebuffers

Copyright

Name

glIsProgram — Determines if a name corresponds to a program object

C Specification

`GLboolean` **`glIsProgram`** `(`*`program`*`);`

`GLuint` *`program`*`;`

Parameters

`program` Specifies a potential program object.

Description

`glIsProgram` returns `GL_TRUE` if *`program`* is the name of a program object previously created with glCreateProgram and not yet deleted with glDeleteProgram. If *`program`* is zero or a non-zero value that is not the name of a program object, or if an error occurs, `glIsProgram` returns `GL_FALSE`.

Notes

No error is generated if *`program`* is not a valid program object name.

A program object marked for deletion with glDeleteProgram but still in use as part of current rendering state is still considered a program object and `glIsProgram` will return `GL_TRUE`.

Associated Gets

glGet with the argument `GL_CURRENT_PROGRAM`

glGetActiveAttrib with arguments *`program`* and the index of an active attribute variable

glGetActiveUniform with arguments *`program`* and the index of an active uniform variable

glGetAttachedShaders with argument *`program`*

glGetAttribLocation with arguments *`program`* and the name of an attribute variable

glGetProgramiv with arguments *`program`* and the parameter to be queried

glGetProgramInfoLog with argument *`program`*

glGetUniform with arguments *`program`* and the location of a uniform variable

glGetUniformLocation with arguments *`program`* and the name of a uniform variable

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsProgram	✔	✔

See Also

glAttachShader, glBindAttribLocation, glCreateProgram, glDeleteProgram, glDetachShader, glLinkProgram, glUniform, glUseProgram, glValidateProgram

Copyright

Name

glIsQuery — determine if a name corresponds to a query object

C Specification

```
GLboolean glIsQuery (id);
```

```
GLuint id;
```

Parameters

id Specifies a value that may be the name of a query object.

Description

`glIsQuery` returns `GL_TRUE` if *id* is currently the name of a query object. If *id* is zero, or is a non-zero value that is not currently the name of a query object, or if an error occurs, `glIsQuery` returns `GL_FALSE`.

A name returned by glGenQueries, but not yet associated with a query object by calling glBeginQuery, is not the name of a query object.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsQuery	-	✔

See Also

glBeginQuery, glDeleteQueries, glEndQuery, glGenQueries

Copyright

Name

glIsRenderbuffer — determine if a name corresponds to a renderbuffer object

C Specification

```
GLboolean glIsRenderbuffer (renderbuffer);

GLuint renderbuffer;
```

Parameters

renderbuffer Specifies a value that may be the name of a renderbuffer object.

Description

`glIsRenderbuffer` returns `GL_TRUE` if *renderbuffer* is currently the name of a renderbuffer object. If *renderbuffer* is zero, or if *renderbuffer* is not the name of a renderbuffer object, or if an error occurs, `glIsRenderbuffer` returns `GL_FALSE`. If *renderbuffer* is a name returned by glGenRenderbuffers, by that has not yet been bound through a call to glBindRenderbuffer or glFramebufferRenderbuffer, then the name is not a renderbuffer object and `glIsRenderbuffer` returns `GL_FALSE`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsRenderbuffer	✔	✔

See Also

glGenRenderbuffers, glBindRenderbuffer, glFramebufferRenderbuffer, glDeleteRenderbuffers

Copyright

Name

glIsSampler — determine if a name corresponds to a sampler object

C Specification

```
GLboolean glIsSampler (id);

GLuint id;
```

Parameters

id Specifies a value that may be the name of a sampler object.

Description

`glIsSampler` returns `GL_TRUE` if *id* is currently the name of a sampler object. If *id* is zero, or is a non-zero value that is not currently the name of a sampler object, or if an error occurs, `glIsSampler` returns `GL_FALSE`.

A name returned by glGenSamplers, is the name of a sampler object.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsSampler	-	✔

See Also

glGenSamplers, glBindSampler, glDeleteSamplers

Copyright

Name

glIsShader — Determines if a name corresponds to a shader object

C Specification

```
GLboolean glIsShader (shader);
GLuint shader;
```

Parameters

`shader` Specifies a potential shader object.

Description

`glIsShader` returns `GL_TRUE` if *`shader`* is the name of a shader object previously created with glCreateShader and not yet deleted with glDeleteShader. If *`shader`* is zero or a non-zero value that is not the name of a shader object, or if an error occurs, `glIsShader` returns `GL_FALSE`.

Notes

No error is generated if *`shader`* is not a valid shader object name.

A shader object marked for deletion with glDeleteShader but still attached to a program object is still considered a shader object and `glIsShader` will return `GL_TRUE`.

Associated Gets

glGetAttachedShaders with a valid program object

glGetShaderiv with arguments *`shader`* and a parameter to be queried

glGetShaderInfoLog with argument *`object`*

glGetShaderSource with argument *`object`*

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glIsShader	✔	✔

See Also

glAttachShader, glCompileShader, glCreateShader, glDeleteShader, glDetachShader, glLinkProgram, glShaderSource

Copyright

Name

glIsSync — determine if a name corresponds to a sync object

C Specification

```
GLboolean glIsSync (sync);
```

```
GLsync sync;
```

Parameters

sync Specifies a value that may be the name of a sync object.

Description

`glIsSync` returns `GL_TRUE` if *sync* is currently the name of a sync object. If *sync* is not the name of a sync object, or if an error occurs, `glIsSync` returns `GL_FALSE`. Note that zero is not the name of a sync object.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsSync	-	✔

See Also

glFenceSync, glWaitSync, glClientWaitSync, glDeleteSync

Copyright

Name

glIsTexture — determine if a name corresponds to a texture

C Specification

```
GLboolean glIsTexture (texture);
```

```
GLuint texture;
```

Parameters

texture	Specifies a value that may be the name of a texture.

Description

`glIsTexture` returns `GL_TRUE` if *`texture`* is currently the name of a texture. If *`texture`* is zero, or is a non-zero value that is not currently the name of a texture, or if an error occurs, `glIsTexture` returns `GL_FALSE`.

A name returned by glGenTextures, but not yet associated with a texture by calling glBindTexture, is not the name of a texture.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsTexture	✔	✔

See Also

glBindTexture, glCopyTexImage2D, glDeleteTextures, glGenTextures, glGet, glGetTexParameter, glTexImage2D, glTexImage3D, glTexParameter

Copyright

Name

glIsTransformFeedback — determine if a name corresponds to a transform feedback object

C Specification

```
GLboolean glIsTransformFeedback (id);
```

```
GLuint id;
```

Parameters

id Specifies a value that may be the name of a transform feedback object.

Description

`glIsTransformFeedback` returns `GL_TRUE` if *id* is currently the name of a transform feedback object. If *id* is zero, or if `id` is not the name of a transform feedback object, or if an error occurs, `glIsTransformFeedback` returns `GL_FALSE`. If *id* is a name returned by glGenTransformFeedbacks, but that has not yet been bound through a call to glBindTransformFeedback, then the name is not a transform feedback object and `glIsTransformFeedback` returns `GL_FALSE`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsTransformFeedback	-	✔

See Also

glGenTransformFeedbacks, glBindTransformFeedback, glDeleteTransformFeedbacks

Copyright

Name

glIsVertexArray — determine if a name corresponds to a vertex array object

C Specification

```
GLboolean glIsVertexArray (array);

GLuint array;
```

Parameters

array Specifies a value that may be the name of a vertex array object.

Description

`glIsVertexArray` returns `GL_TRUE` if *array* is currently the name of a vertex array object. If *array* is zero, or if *array* is not the name of a vertex array object, or if an error occurs, `glIsVertexArray` returns `GL_FALSE`. If *array* is a name returned by glGenVertexArrays, by that has not yet been bound through a call to glBindVertexArray, then the name is not a vertex array object and `glIsVertexArray` returns `GL_FALSE`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glIsVertexArray	-	✔

See Also

glGenVertexArrays, glBindVertexArray, glDeleteVertexArrays

Copyright

Name

glLineWidth — specify the width of rasterized lines

C Specification

```
void glLineWidth (width);

GLfloat width;
```

Parameters

width Specifies the width of rasterized lines. The initial value is 1.

Description

`glLineWidth` specifies the rasterized width of lines.

The actual width is determined by rounding the supplied width to the nearest integer. (If the rounding results in the value 0, it is as if the line width were 1.) If , *i* pixels are filled in each column that is rasterized, where *i* is the rounded value of *width*. Otherwise, *i* pixels are filled in each row that is rasterized.

There is a range of supported line widths. Only width 1 is guaranteed to be supported; others depend on the implementation. To query the range of supported widths, call glGet with argument `GL_ALIASED_LINE_WIDTH_RANGE`.

Notes

The line width specified by `glLineWidth` is always returned when `GL_LINE_WIDTH` is queried. Clamping and rounding have no effect on the specified value.

Line width may be clamped to an implementation-dependent maximum. Call glGet with `GL_ALIASED_LINE_WIDTH_RANGE` to determine the maximum width.

Errors

`GL_INVALID_VALUE` is generated if *width* is less than or equal to 0.

Associated Gets

glGet with argument `GL_LINE_WIDTH`

glGet with argument `GL_ALIASED_LINE_WIDTH_RANGE`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glLineWidth	✔	✔

See Also

glEnable

Copyright

Name

glLinkProgram — Links a program object

C Specification

```
void glLinkProgram (program);
```

```
GLuint program;
```

Parameters

program	Specifies the handle of the program object to be linked.

Description

`glLinkProgram` links the program object specified by *program*. Shader objects of type `GL_VERTEX_SHADER` attached to *program* are used to create an executable that will run on the programmable vertex processor. Shader objects of type `GL_FRAGMENT_SHADER` attached to *program* are used to create an executable that will run on the programmable fragment processor.

The status of the link operation will be stored as part of the program object's state. This value will be set to `GL_TRUE` if the program object was linked without errors and is ready for use, and `GL_FALSE` otherwise. It can be queried by calling glGetProgramiv with arguments *program* and `GL_LINK_STATUS`.

As a result of a successful link operation, all active user-defined uniform variables belonging to *program* will be initialized to 0, and each of the program object's active uniform variables will be assigned a location that can be queried by calling glGetUniformLocation. All active uniforms belonging to the program's named uniform blocks are assigned offsets (and strides for array and matrix type uniforms) within the uniform block. Also, any active user-defined attribute variables that have not been bound to a generic vertex attribute index will be bound to one at this time.

Linking of a program object can fail for a number of reasons as specified in the *OpenGL ES Shading Language Specification*. The following lists some of the conditions that will cause a link error.

- A vertex shader and a fragment shader are not both present in the program object.
- The vertex and fragment shader do not use the same shader language version.
- The number of active attribute variables supported by the implementation has been exceeded.
- The storage limit for uniform variables has been exceeded.
- The number of active uniform variables supported by the implementation has been exceeded.
- The `main` function is missing for the vertex or fragment shader.
- A varying variable actually used in the fragment shader is not declared in the same way (or is not declared at all) in the vertex shader.
- A reference to a function or variable name is unresolved.
- A shared global is declared with two different types or two different initial values.
- One or more of the attached shader objects has not been successfully compiled (via glCompileShader) or loaded with a pre-compiled shader binary (via glShaderBinary).

- Binding a generic attribute matrix caused some rows of the matrix to fall outside the allowed maximum of `GL_MAX_VERTEX_ATTRIBS`.
- Not enough contiguous vertex attribute slots could be found to bind attribute matrices.
- Any variable name specified to glTransformFeedbackVaryings in the *`varyings`* array is not declared as an output in the vertex shader.
- Any two entries in the *`varyings`* array given glTransformFeedbackVaryings specify the same varying variable.
- The total number of components to capture in any transform feedback varying variable is greater than the constant `GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS` and the buffer mode is `GL_SEPARATE_ATTRIBS`.
- The total number of components to capture in any transform feedback varying variable is greater than the constant `GL_MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS` and the buffer mode is `GL_INTERLEAVED_ATTRIBS`.

When a program object has been successfully linked, the program object can be made part of current state by calling glUseProgram. Whether or not the link operation was successful, the program object's information log will be overwritten. The information log can be retrieved by calling glGetProgramInfoLog.

`glLinkProgram` will also install the generated executables as part of the current rendering state if the link operation was successful and the specified program object is already currently in use as a result of a previous call to glUseProgram. If the program object currently in use is relinked unsuccessfully, its link status will be set to `GL_FALSE` , but the executables and associated state will remain part of the current state until a subsequent call to `glUseProgram` removes it from use. After it is removed from use, it cannot be made part of current state until it has been successfully relinked.

The program object's information log is updated and the program is generated at the time of the link operation. After the link operation, applications are free to modify attached shader objects, compile attached shader objects, detach shader objects, delete shader objects, and attach additional shader objects. None of these operations affects the information log or the program that is part of the program object.

Notes

If the link operation is unsuccessful, any information about a previous link operation on *`program`* is lost (i.e., a failed link does not restore the old state of *`program`*). Certain information can still be retrieved from *`program`* even after an unsuccessful link operation. See for instance glGetActiveAttrib and glGetActiveUniform.

Errors

`GL_INVALID_VALUE` is generated if *`program`* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`program`* is not a program object.

`GL_INVALID_OPERATION` is generated if *`program`* is the currently active program object and transform feedback mode is active.

Associated Gets

glGet with the argument `GL_CURRENT_PROGRAM` glGetActiveAttrib with argument *`program`* and the index of an active attribute variable

glGetActiveUniform with argument *`program`* and the index of an active uniform variable

glGetActiveUniformBlockiv with argument *`program`* and the index of an active uniform block

glGetAttachedShaders with argument *`program`*

glGetAttribLocation with argument *`program`* and an attribute variable name

glGetProgramiv with arguments *`program`* and `GL_LINK_STATUS`

glGetProgramInfoLog with argument *`program`*

glGetUniform with argument *`program`* and a uniform variable location

glGetUniformLocation with argument *`program`* and a uniform variable name

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glLinkProgram	✔	✔

See Also

glAttachShader, glBindAttribLocation, glCompileShader, glCreateProgram, glDeleteProgram, glDetachShader, glUniform, glUseProgram, glValidateProgram

Copyright

Name

glMapBufferRange — map a section of a buffer object's data store

C Specification

```
void *glMapBufferRange (target, offset, length, access);

GLenum target;
GLintptr offset;
GLsizeiptr length;
GLbitfield access;

GLboolean glUnmapBuffer (target);

GLenum target;
```

Parameters for `glMapBufferRange`

`target`	Specifies a binding to which the target buffer is bound.
`offset`	Specifies the starting offset within the buffer of the range to be mapped.
`length`	Specifies the length of the range to be mapped.
`access`	Specifies a combination of access flags indicating the desired access to the range.

Parameters for `glUnmapBuffer`

`target`	Specifies a binding to which the target buffer is bound.

Description

`glMapBufferRange` maps all or part of the data store of a buffer object into the client's address space. *`target`* specifies the target to which the buffer is bound and must be one of `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`. *`offset`* and *`length`* indicate the range of data in the buffer object that is to be mapped, in terms of basic machine units. *`access`* is a bitfield containing flags which describe the requested mapping. These flags are described below.

If no error occurs, a pointer to the beginning of the mapped range is returned once all pending operations on that buffer have completed, and may be used to modify and/or query the corresponding range of the buffer, according to the following flag bits set in *`access`*:

- `GL_MAP_READ_BIT` indicates that the returned pointer may be used to read buffer object data. No GL error is generated if the pointer is used to query a mapping which excludes this flag, but the result is undefined and system errors (possibly including program termination) may occur.

- `GL_MAP_WRITE_BIT` indicates that the returned pointer may be used to modify buffer object data. No GL error is generated if the pointer is used to modify a mapping which excludes this flag, but the result is undefined and system errors (possibly including program termination) may occur.

Furthermore, the following *optional* flag bits in *`access`* may be used to modify the mapping:

- `GL_MAP_INVALIDATE_RANGE_BIT` indicates that the previous contents of the specified range may be discarded. Data within this range are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with `GL_MAP_READ_BIT`.

- `GL_MAP_INVALIDATE_BUFFER_BIT` indicates that the previous contents of the entire buffer may be discarded. Data within the entire buffer are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with `GL_MAP_READ_BIT`.

- `GL_MAP_FLUSH_EXPLICIT_BIT` indicates that one or more discrete subranges of the mapping may be modified. When this flag is set, modifications to each subrange must be explicitly flushed by calling glFlushMappedBufferRange. No GL error is set if a subrange of the mapping is modified and not flushed, but data within the corresponding subrange of the buffer are undefined. This flag may only be used in conjunction with `GL_MAP_WRITE_BIT`. When this option is selected, flushing is strictly limited to regions that are explicitly indicated with calls to glFlushMappedBufferRange prior to unmap; if this option is not selected glUnmapBuffer will automatically flush the entire mapped range when called.

- `GL_MAP_UNSYNCHRONIZED_BIT` indicates that the GL should not attempt to synchronize pending operations on the buffer prior to returning from `glMapBufferRange`. No GL error is generated if pending operations which source or modify the buffer overlap the mapped region, but the result of such previous and any subsequent operations is undefined.

If an error occurs, `glMapBufferRange` returns a `NULL` pointer.

A mapped data store must be unmapped with `glUnmapBuffer` before its buffer object is used. Otherwise an error will be generated by any GL command that attempts to dereference the buffer object's data store. When a data store is unmapped, the pointer to its data store becomes invalid. `glUnmapBuffer` returns `GL_TRUE` unless the data store contents have become corrupt during the time the data store was mapped. This can occur for system-specific reasons that affect the availability of graphics memory, such as screen mode changes. In such situations, `GL_FALSE` is returned and the data store contents are undefined. An application must detect this rare condition and reinitialize the data store.

A buffer object's mapped data store is automatically unmapped when the buffer object is deleted or its data store is recreated with `glBufferData`.

Notes

Mappings to the data stores of buffer objects may have nonstandard performance characteristics. For example, such mappings may be marked as uncacheable regions of memory, and in such cases reading from them may be very slow. To ensure optimal performance, the client should use the mapping in a fashion consistent with the values of `GL_BUFFER_USAGE` and *`access`*. Using a mapping in a fashion inconsistent with these values is liable to be multiple orders of magnitude slower than using normal memory.

Errors

`GL_INVALID_VALUE` is generated if either of *`offset`* or *`length`* is negative, or if *`offset`* + *`length`* is greater than the value of `GL_BUFFER_SIZE`.

`GL_INVALID_VALUE` is generated if *`access`* has any bits set other than those defined above.

`GL_INVALID_OPERATION` is generated for any of the following conditions:

- The buffer is already in a mapped state.
- Neither `GL_MAP_READ_BIT` or `GL_MAP_WRITE_BIT` is set.
- `GL_MAP_READ_BIT` is set and any of `GL_MAP_INVALIDATE_RANGE_BIT`, `GL_MAP_INVALIDATE_BUFFER_BIT`, or `GL_MAP_UNSYNCHRONIZED_BIT` is set.
- `GL_MAP_FLUSH_EXPLICIT_BIT` is set and `GL_MAP_WRITE_BIT` is not set.

`GL_OUT_OF_MEMORY` is generated if `glMapBufferRange` fails because memory for the mapping could not be obtained.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glMapBufferRange`	-	✔
`glUnmapBuffer`	-	✔

See Also

glBindBuffer glFlushMappedBufferRange, glUnmapBuffer,

Copyright

Name

glPauseTransformFeedback — pause transform feedback operations

C Specification

```
void glPauseTransformFeedback ();

void;
```

Description

`glPauseTransformFeedback` pauses transform feedback operations on the currently active transform feedback object. When transform feedback operations are paused, transform feedback is still considered active and changing most transform feedback state related to the object results in an error. However, a new transform feedback object may be bound while transform feedback is paused.

Errors

`GL_INVALID_OPERATION` is generated if the currently bound transform feedback object is not active or is paused.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glPauseTransformFeedback	-	✔

See Also

glGenTransformFeedbacks, glBindTransformFeedback, glBeginTransformFeedback, glResumeTransformFeedback, glEndTransformFeedback, glDeleteTransformFeedbacks

Copyright

Name

glPixelStorei — set pixel storage modes

C Specification

```
void glPixelStorei (pname, param);

GLenum pname;
GLint param;
```

Parameters

pname Specifies the symbolic name of the parameter to be set. Four values affect the packing of pixel data into memory: GL_PACK_ROW_LENGTH, GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS, and GL_PACK_ALIGNMENT. Six more affect the unpacking of pixel data *from* memory: GL_UNPACK_ROW_LENGTH, GL_UNPACK_IMAGE_HEIGHT, GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS, GL_UNPACK_SKIP_IMAGES, and GL_UNPACK_ALIGNMENT.

param Specifies the value that *pname* is set to.

Description

glPixelStorei sets pixel storage modes that affect the operation of subsequent glReadPixels as well as the unpacking of texture patterns (see glTexImage2D, glTexImage3D, glTexSubImage2D, glTexSubImage3D).

pname is a symbolic constant indicating the parameter to be set, and *param* is the new value. Four of the ten storage parameters affect how pixel data is returned to client memory. They are as follows:

GL_PACK_ROW_LENGTH

If greater than 0, GL_PACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location in memory, then the location of the first pixel of the next row is obtained by skipping

components or indices, where is the number of components or indices in a pixel, is the number of pixels in a row (GL_PACK_ROW_LENGTH if it is greater than 0, the argument to the pixel routine otherwise), is the value of GL_PACK_ALIGNMENT, and is the size, in bytes, of a single component (if , then it is as if). In the case of 1-bit values, the location of the next row is obtained by skipping

components or indices.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

`GL_PACK_SKIP_PIXELS` and `GL_PACK_SKIP_ROWS`

These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to glReadPixels. Setting `GL_PACK_SKIP_PIXELS` to is equivalent to incrementing the pointer by components or indices, where is the number of components or indices in each pixel. Setting `GL_PACK_SKIP_ROWS` to is equivalent to incrementing the pointer by components or indices, where is the number of components or indices per row, as just computed in the `GL_PACK_ROW_LENGTH` section.

`GL_PACK_ALIGNMENT`

Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The other six of the ten storage parameters affect how pixel data is read from client memory. These values are significant for glTexImage2D, glTexImage3D, glTexSubImage2D, and glTexSubImage3D

They are as follows:

`GL_UNPACK_ROW_LENGTH`

If greater than 0, `GL_UNPACK_ROW_LENGTH` defines the number of pixels in a row. If the first pixel of a row is placed at location in memory, then the location of the first pixel of the next row is obtained by skipping

components or indices, where is the number of components or indices in a pixel, is the number of pixels in a row (`GL_UNPACK_ROW_LENGTH` if it is greater than 0, the argument to the pixel routine otherwise), is the value of `GL_UNPACK_ALIGNMENT`, and is the size, in bytes, of a single component (if , then it is as if). In the case of 1-bit values, the location of the next row is obtained by skipping

components or indices.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format `GL_RGB`, for example, has three components per pixel: first red, then green, and finally blue.

`GL_UNPACK_IMAGE_HEIGHT`

If greater than 0, `GL_UNPACK_IMAGE_HEIGHT` defines the number of pixels in an image of a three-dimensional texture volume. Where ``image'' is defined by all pixel sharing the same third dimension index. If the first pixel of a row is placed at location in memory, then the location of the first pixel of the next row is obtained by skipping

components or indices, where is the number of components or indices in a pixel, is the number of pixels in a row (`GL_UNPACK_ROW_LENGTH` if it is greater than 0, the argument to glTexImage3D otherwise), is the number of rows in an image (GL_UNPACK_IMAGE_HEIGHT if it is greater than 0, the argument to glTexImage3D otherwise), is the value of GL_UNPACK_ALIGNMENT, and is the size, in bytes, of a single component (if , then it is as if).

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS and GL_UNPACK_SKIP_IMAGES	These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated by incrementing the pointer passed to glTexImage2D or glTexSubImage2D. Setting GL_UNPACK_SKIP_PIXELS to is equivalent to incrementing the pointer by components or indices, where is the number of components or indices in each pixel. Setting GL_UNPACK_SKIP_ROWS to is equivalent to incrementing the pointer by components or indices, where is the number of components or indices per row, as just computed in the GL_UNPACK_ROW_LENGTH section. Setting GL_UNPACK_SKIP_IMAGES to is equivalent to incrementing the pointer by , where is the number of components or indices per image, as computed in the GL_UNPACK_IMAGE_HEIGHT section.
GL_UNPACK_ALIGNMENT	Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The following table gives the type, initial value, and range of valid values for each storage parameter that can be set with glPixelStorei.

pname	**Type**	**Initial Value**	**Valid Range**
GL_PACK_ROW_LENGTH	integer	0	
GL_PACK_SKIP_ROWS	integer	0	
GL_PACK_SKIP_PIXELS	integer	0	
GL_PACK_ALIGNMENT	integer	4	1, 2, 4, or 8
GL_UNPACK_ROW_LENGTH	integer	0	
GL_UNPACK_IMAGE_HEIGHT	integer	0	
GL_UNPACK_SKIP_ROWS	integer	0	
GL_UNPACK_SKIP_PIXELS	integer	0	
GL_UNPACK_SKIP_IMAGES	integer	0	
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4, or 8

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

`GL_INVALID_VALUE` is generated if a negative row length, pixel skip, or row skip value is specified, or if alignment is specified as other than 1, 2, 4, or 8.

Associated Gets

glGet with argument `GL_PACK_ROW_LENGTH`

glGet with argument `GL_PACK_SKIP_ROWS`

glGet with argument `GL_PACK_SKIP_PIXELS`

glGet with argument `GL_PACK_ALIGNMENT`

glGet with argument `GL_UNPACK_ROW_LENGTH`

glGet with argument `GL_UNPACK_IMAGE_HEIGHT`

glGet with argument `GL_UNPACK_SKIP_ROWS`

glGet with argument `GL_UNPACK_SKIP_PIXELS`

glGet with argument `GL_UNPACK_SKIP_IMAGES`

glGet with argument `GL_UNPACK_ALIGNMENT`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glPixelStorei	✔	✔

See Also

glReadPixels, glTexImage2D, glTexImage3D, glTexSubImage2D, glTexSubImage3D

Copyright

Name

glPolygonOffset — set the scale and units used to calculate depth values

C Specification

```
void glPolygonOffset (factor, units);

GLfloat factor;
GLfloat units;
```

Parameters

factor	Specifies a scale factor that is used to create a variable depth offset for each polygon. The initial value is 0.
units	Is multiplied by an implementation-specific value to create a constant depth offset. The initial value is 0.

Description

When `GL_POLYGON_OFFSET_FILL` is enabled, each fragment's *depth* value will be offset after it is interpolated from the *depth* values of the appropriate vertices. The value of the offset is , where is a measurement of the change in depth relative to the screen area of the polygon, and is the smallest value that is guaranteed to produce a resolvable offset for a given implementation. The offset is added before the depth test is performed and before the value is written into the depth buffer.

`glPolygonOffset` is useful for applying decals.

Associated Gets

glIsEnabled with argument `GL_POLYGON_OFFSET_FILL`.

glGet with argument `GL_POLYGON_OFFSET_FACTOR` or `GL_POLYGON_OFFSET_UNITS`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glPolygonOffset	✔	✔

See Also

glDepthFunc, glEnable, glGet, glIsEnabled

Copyright

Name

glProgramBinary — load a program object with a program binary

C Specification

```
void glProgramBinary (program, binaryFormat, binary, length);

GLuint program;
GLenum binaryFormat;
const void *binary;
GLsizei length;
```

Parameters

program	Specifies the name of a program object into which to load a program binary.
binaryFormat	Specifies the format of the binary data in binary.
binary	Specifies the address of an array containing the binary to be loaded into *program*.
length	Specifies the number of bytes contained in *binary*.

Description

`glProgramBinary` loads a program object with a program binary previously returned from glGetProgramBinary. *binaryFormat* and *binary* must be those returned by a previous call to glGetProgramBinary, and *length* must be the length returned by glGetProgramBinary, or by glGetProgramiv when called with *pname* set to `GL_PROGRAM_BINARY_LENGTH`. If these conditions are not met, loading the program binary will fail and *program*'s `GL_LINK_STATUS` will be set to `GL_FALSE`.

A program object's program binary is replaced by calls to glLinkProgram or `glProgramBinary`. When linking success or failure is concerned, `glProgramBinary` can be considered to perform an implicit linking operation. glLinkProgram and `glProgramBinary` both set the program object's `GL_LINK_STATUS` to `GL_TRUE` or `GL_FALSE`.

A successful call to `glProgramBinary` will reset all uniform variables to their initial values, `GL_FALSE` for booleans and zero for all others. Additionally, all vertex shader input and fragment shader output assignments that were in effect when the program was linked before saving are restored with `glProgramBinary` is called.

Errors

`GL_INVALID_OPERATION` is generated if *program* is not the name of an existing program object.

`GL_INVALID_ENUM` is generated if *binaryFormat* is not a value recognized by the implementation.

Notes

A program binary may fail to load if the implementation determines that there has been a change in hardware or software configuration from when the program binary was produced such as having been compiled with an incompatible or outdated version of the compiler.

Associated Gets

glGetProgramiv with argument `GL_PROGRAM_BINARY_LENGTH`

glGet with argument `GL_NUM_PROGRAM_BINARY_FORMATS`

glGet with argument `GL_PROGRAM_BINARY_FORMATS`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glProgramBinary	-	✔

See Also

glGetProgramiv, glGetProgramBinary

Copyright

Name

glProgramParameteri — specify a parameter for a program object

C Specification

```
void glProgramParameteri (program, pname, value);

GLuint program;
GLenum pname;
GLint value;
```

Parameters

program	Specifies the name of a program object whose parameter to modify.
pname	Specifies the name of the parameter to modify.
value	Specifies the new value of the parameter specified by *pname* for *program*.

Description

`glProgramParameteri` specifies a new value for the parameter nameed by *pname* for the program object *program*.

If *pname* is `GL_PROGRAM_BINARY_RETRIEVABLE_HINT`, *value* should be `GL_FALSE` or `GL_TRUE` to indicate to the implementation the intention of the application to retrieve the program's binary representation with glGetProgramBinary. The implementation may use this information to store information that may be useful for a future query of the program's binary. It is recommended to set `GL_PROGRAM_BINARY_RETRIEVABLE_HINT` for the program to `GL_TRUE` before calling glLinkProgram, and using the program at run-time if the binary is to be retrieved later.

Errors

`GL_INVALID_OPERATION` is generated if *program* is not the name of an existing program object.

`GL_INVALID_ENUM` is generated if *pname* is not `GL_PROGRAM_BINARY_RETRIEVABLE_HINT`.

`GL_INVALID_VALUE` is generated if *value* is not `GL_FALSE` or `GL_TRUE`.

Associated Gets

glGetProgramiv.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glProgramParameteri	-	✔

See Also

glGetProgramiv, glGetProgramBinary, glProgramBinary

Copyright

Name

glReadBuffer — select a color buffer source for pixels

C Specification

`void glReadBuffer (src);`

`GLenum src;`

Parameters

src Specifies a color buffer. Accepted values are `GL_BACK`, `GL_NONE`, and `GL_COLOR_ATTACHMENTi`.

Description

`glReadBuffer` specifies a color buffer as the source for subsequent glReadPixels, , glCopyTexImage2D, glCopyTexSubImage2D, and glCopyTexSubImage3D commands. *src* accepts one of the following values: `GL_NONE`, `GL_BACK` names the back buffer of the default framebuffer, and `GL_COLOR_ATTACHMENTi` names a color attachment of the current framebuffer,

Errors

`GL_INVALID_ENUM` is generated if *src* is not `GL_BACK`, `GL_NONE`, or `GL_COLOR_ATTACHMENTi`, where i is less than `GL_MAX_COLOR_ATTACHMENTS`.

`GL_INVALID_OPERATION` is generated if the current framebuffer is the default framebufer and *src* is not `GL_NONE` or `GL_BACK`.

`GL_INVALID_OPERATION` is generated if the current framebuffer is a named framebufer and *src* is not `GL_NONE` or `GL_COLOR_ATTACHMENTi`.

Associated Gets

glGet with argument `GL_READ_BUFFER`

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glReadBuffer	-	✔

See Also

glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glDrawBuffers, glReadPixels

Copyright

Name

glReadPixels — read a block of pixels from the frame buffer

C Specification

```
void glReadPixels (x, y, width, height, format, type, data);

GLint x;
GLint y;
GLsizei width;
GLsizei height;
GLenum format;
GLenum type;
void * data;
```

Parameters

x, *y*	Specify the window coordinates of the first pixel that is read from the frame buffer. This location is the lower left corner of a rectangular block of pixels.
width, *height*	Specify the dimensions of the pixel rectangle. *width* and *height* of one correspond to a single pixel.
format	Specifies the format of the pixel data. The following symbolic values are accepted: `GL_RGBA`, and `GL_RGBA_INTEGER`. An implementation-chosen format will also be accepted. This can be queried with glGet and `GL_IMPLEMENTATION_COLOR_READ_FORMAT`.
type	Specifies the data type of the pixel data. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_INT`, `GL_INT`, or `GL_FLOAT`. An implementation-chosen type will also be accepted. This can be queried with glGet and `GL_IMPLEMENTATION_COLOR_READ_TYPE`.
data	Returns the pixel data.

Description

`glReadPixels` returns pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (*x*, *y*), into client memory starting at location *data*. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with glPixelStorei. This reference page describes the effects on `glReadPixels` of most, but not all of the parameters specified by these three commands.

If a non-zero named buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target (see glBindBuffer) while a block of pixels is requested, *data* is treated as a byte offset into the buffer object's data store rather than a pointer to client memory.

`glReadPixels` returns values from each pixel with lower left corner at for and . This pixel is said to be the th pixel in the th row. Pixels are returned in row order from the lowest to the highest row, left to right in each row.

format specifies the format for the returned pixel values; accepted values are `GL_RED`, `GL_RED_INTEGER`, `GL_RG`, `GL_RG_INTEGER`, `GL_RGB`, `GL_RGB_INTEGER`, `GL_RGBA`, `GL_RGBA_INTEGER`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE`, and `GL_ALPHA`

Finally, the indices or components are converted to the proper format, as specified by *`type`*. If *`type`* is `GL_FLOAT`, then each integer index is converted to single-precision floating-point format.

If *`format`* is `GL_RED`, `GL_RG`, `GL_RGB`, or `GL_RGBA`, and *`type`* is not `GL_FLOAT`, each component is multiplied by the multiplier shown in the following table. If type is `GL_FLOAT`, then each component is passed as is (or converted to the client's single-precision floating-point format if it is different from the one used by the GL).

`type`	**Index Mask**	**Component Conversion**
`GL_UNSIGNED_BYTE`		
`GL_BYTE`		
`GL_HALF_FLOAT`	none	
`GL_FLOAT`	none	
`GL_UNSIGNED_SHORT_5_6_5`		
`GL_UNSIGNED_SHORT_4_4_4_4`		
`GL_UNSIGNED_SHORT_5_5_5_1`		
`GL_UNSIGNED_INT_2_10_10_10_REV`		
`GL_UNSIGNED_INT_10F_11F_11F_REV`	--	Special
`GL_UNSIGNED_INT_5_9_9_9_REV`	--	Special

Return values are placed in memory as follows. If *`format`* is `GL_RED`, or `GL_RED_INTEGER`, a single value is returned and the data for the th pixel in the th row is placed in location . `GL_RG` and `GL_RG_INTEGER` return two values, `GL_RGB` and `GL_RGB_INTEGER` return three values, `GL_RGBA` and `GL_RGBA_INTEGER` return four values for each pixel, with all values corresponding to a single pixel occupying contiguous space in *`data`*. See glPixelStorei for a description of parameters which affect the packing of data into memory.

Notes

Values for pixels that lie outside the window connected to the current GL context are undefined.

If an error is generated, no change is made to the contents of *`data`*.

Only two *`format`*/*`type`* parameter pairs are accepted. For normalized fixed point rendering surfaces, `GL_RGBA`/`GL_UNSIGNED_BYTE` is accepted. For signed integer rendering surfaces, `GL_RGBA_INTEGER`/`GL_INT` is accepted. For unsigned integer rendering surfaces, `GL_RGBA_INTEGER`/`GL_UNSIGNED_INT` is accepted. The other acceptable pair can be discovered by querying `GL_IMPLEMENTATION_COLOR_READ_FORMAT` and `GL_IMPLEMENTATION_COLOR_READ_TYPE`. The implementation chosen format may also vary depending on the format of the currently bound rendering surface.

Errors

`GL_INVALID_ENUM` is generated if *`format`* or *`type`* is not an accepted value.

`GL_INVALID_VALUE` is generated if either *`width`* or *`height`* is negative.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

GL_INVALID_OPERATION is generated if GL_READ_BUFFER is GL_NONE or if GL_READ_FRAMEBUFFER_BINDING is non-zero and the read buffer selects an attachment that has no image attached.

GL_INVALID_OPERATION is generated if GL_READ_FRAMEBUFFER_BINDING is non-zero, the read framebuffer is complete, and the value of GL_SAMPLE_BUFFERS for the read framebuffer is greater than zero.

GL_INVALID_OPERATION is generated if the readbuffer of the currently bound framebuffer is a fixed point normalized surface and *format* and *type* are neither GL_RGBA and GL_UNSIGNED_BYTE, respectively, nor the format/type pair returned by querying GL_IMPLEMENTATION_COLOR_READ_FORMAT and GL_IMPLEMENTATION_COLOR_READ_TYPE.

GL_INVALID_OPERATION is generated if the readbuffer of the currently bound framebuffer is a floating point surface and *format* and *type* are neither GL_RGBA and GL_FLOAT, respectively, nor the format/type pair returned by querying GL_IMPLEMENTATION_COLOR_READ_FORMAT and GL_IMPLEMENTATION_COLOR_READ_TYPE.

GL_INVALID_OPERATION is generated if the readbuffer of the currently bound framebuffer is a signed integer surface and *format* and *type* are neither GL_RGBA_INTEGER and GL_INT, respectively, nor the format/type pair returned by querying GL_IMPLEMENTATION_COLOR_READ_FORMAT and GL_IMPLEMENTATION_COLOR_READ_TYPE.

GL_INVALID_OPERATION is generated if the readbuffer of the currently bound framebuffer is an unsigned integer surface and *format* and *type* are neither GL_RGBA_INTEGER and GL_UNSIGNED_INT, respectively, nor the format/type pair returned by querying GL_IMPLEMENTATION_COLOR_READ_FORMAT and GL_IMPLEMENTATION_COLOR_READ_TYPE.

GL_INVALID_FRAMEBUFFER_OPERATION is generated if the currently bound framebuffer is not framebuffer complete (i.e. the return value from glCheckFramebufferStatus is not GL_FRAMEBUFFER_COMPLETE).

Associated Gets

glGet with argument GL_PIXEL_PACK_BUFFER_BINDING

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glReadPixels	✔	✔

See Also

glPixelStorei, glReadBuffer

Copyright

Name

glReleaseShaderCompiler — release resources consumed by the implementation's shader compiler

C Specification

```
void glReleaseShaderCompiler ();

void;
```

Description

`glReleaseShaderCompiler` provides a hint to the implementation that it may free internal resources associated with its shader compiler. glCompileShader may subsequently be called and the implementation may at that time reallocate resources previously freed by the call to `glReleaseShaderCompiler`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glReleaseShaderCompiler	✔	✔

See Also

glCompileShader, glLinkProgram

Copyright

Name

glRenderbufferStorage — establish data storage, format and dimensions of a renderbuffer object's image

C Specification

```
void glRenderbufferStorage (target, internalformat, width, height);

GLenum target;
GLenum internalformat;
GLsizei width;
GLsizei height;
```

Parameters

target	Specifies a binding to which the target of the allocation and must be `GL_RENDERBUFFER`.
internalformat	Specifies the internal format to use for the renderbuffer object's image.
width	Specifies the width of the renderbuffer, in pixels.
height	Specifies the height of the renderbuffer, in pixels.

Description

`glRenderbufferStorage` is equivalent to calling glRenderbufferStorageMultisample with the *samples* set to zero.

The target of the operation, specified by *target* must be `GL_RENDERBUFFER`. *internalformat* specifies the internal format to be used for the renderbuffer object's storage and must be a color-renderable, depth-renderable, or stencil-renderable format, as shown in Table 1 below. *width* and *height* are the dimensions, in pixels, of the renderbuffer. Both *width* and *height* must be less than or equal to the value of `GL_MAX_RENDERBUFFER_SIZE`.

Upon success, `glRenderbufferStorage` deletes any existing data store for the renderbuffer image and the contents of the data store after calling `glRenderbufferStorage` are undefined.

Table 1. Sized Internal Formats

Sized Internal Format	Base Format	Red Bits	Green Bits	Blue Bits	Alpha Bits
GL_R8	GL_RED	8			
GL_R8UI	GL_RED_INTEGER	ui8			
GL_R8I	GL_RED_INTEGER	i8			
GL_R16UI	GL_RED_INTEGER	ui16			
GL_R16I	GL_RED_INTEGER	i16			
GL_R32UI	GL_RED_INTEGER	ui32			

Sized Internal Format	**Base Format**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**
`GL_R32I`	`GL_RED_INTEGER`	i32			
`GL_RG8`	`GL_RG`	8	8		
`GL_RG8UI`	`GL_RG_INTEGER`	ui8	ui8		
`GL_RG8I`	`GL_RG_INTEGER`	i8	i8		
`GL_RG16UI`	`GL_RG_INTEGER`	ui16	ui16		
`GL_RG16I`	`GL_RG_INTEGER`	i16	i16		
`GL_RG32UI`	`GL_RG_INTEGER`	ui32	ui32		
`GL_RG32I`	`GL_RG_INTEGER`	i32	i32		
`GL_RGB8`	`GL_RGB`	8	8	8	
`GL_RGB565`	`GL_RGB`	5	6	5	
`GL_RGBA8`	`GL_RGBA`	8	8	8	8
`GL_SRGB8_ALPHA8`	`GL_RGBA`	8	8	8	8
`GL_RGB5_A1`	`GL_RGBA`	5	5	5	1
`GL_RGBA4`	`GL_RGBA`	4	4	4	4
`GL_RGB10_A2`	`GL_RGBA`	10	10	10	2
`GL_RGBA8UI`	`GL_RGBA_INTEGER`	ui8	ui8	ui8	ui8
`GL_RGBA8I`	`GL_RGBA_INTEGER`	i8	i8	i8	i8
`GL_RGB10_A2UI`	`GL_RGBA_INTEGER`	ui10	ui10	ui10	ui2
`GL_RGBA16UI`	`GL_RGBA_INTEGER`	ui16	ui16	ui16	ui16
`GL_RGBA16I`	`GL_RGBA_INTEGER`	i16	i16	i16	i16
`GL_RGBA32I`	`GL_RGBA_INTEGER`	i32	i32	i32	i32
`GL_RGBA32UI`	`GL_RGBA_INTEGER`	ui32	ui32	ui32	ui32

Sized Internal Format	**Base Format**	**Depth Bits**	**Stencil Bits**
`GL_DEPTH_COMPONENT16`	`GL_DEPTH_COMPONENT`	16	

Sized Internal Format	Base Format	Depth Bits	Stencil Bits
GL_DEPTH_COMPONENT24	GL_DEPTH_COMPONENT	24	
GL_DEPTH_COMPONENT32F	GL_DEPTH_COMPONENT	f32	
GL_DEPTH24_STENCIL8	GL_DEPTH_STENCIL	24	8
GL_DEPTH32F_STENCIL8	GL_DEPTH_STENCIL	f32	8
GL_STENCIL_INDEX8	GL_STENCIL		8

Errors

GL_INVALID_ENUM is generated if *target* is not GL_RENDERBUFFER.

GL_INVALID_VALUE is generated if either of *width* or *height* is negative, or greater than the value of GL_MAX_RENDERBUFFER_SIZE.

GL_INVALID_ENUM is generated if *internalformat* is not a color-renderable, depth-renderable, or stencil-renderable format.

GL_OUT_OF_MEMORY is generated if the GL is unable to create a data store of the requested size.

API Version Support

	OpenGL ES API Version	
Function Name	2.0	3.0
glRenderbufferStorage	✔	✔

See Also

glGenRenderbuffers, glBindRenderbuffer, glRenderbufferStorageMultisample, glFramebufferRenderbuffer, glDeleteRenderbuffers

Copyright

Name

glRenderbufferStorageMultisample — establish data storage, format, dimensions and sample count of a renderbuffer object's image

C Specification

```
void glRenderbufferStorageMultisample (target, samples, internalformat,
width, height);

GLenum target;
GLsizei samples;
GLenum internalformat;
GLsizei width;
GLsizei height;
```

Parameters

target	Specifies a binding to which the target of the allocation and must be `GL_RENDERBUFFER`.
samples	Specifies the number of samples to be used for the renderbuffer object's storage.
internalformat	Specifies the internal format to use for the renderbuffer object's image.
width	Specifies the width of the renderbuffer, in pixels.
height	Specifies the height of the renderbuffer, in pixels.

Description

`glRenderbufferStorageMultisample` establishes the data storage, format, dimensions and number of samples of a renderbuffer object's image.

The target of the operation, specified by *target* must be `GL_RENDERBUFFER`. *internalformat* specifies the internal format to be used for the renderbuffer object's storage and must be a color-renderable, depth-renderable, or stencil-renderable format, as shown in Table 1 below. *width* and *height* are the dimensions, in pixels, of the renderbuffer. Both *width* and *height* must be less than or equal to the value of `GL_MAX_RENDERBUFFER_SIZE`. *samples* specifies the number of samples to be used for the renderbuffer object's image. If *internalformat* is a signed or unsigned integer format then *samples* must be 0. Otherwise, *samples* must be less than or equal to the maximum number of samples supported for *internalformat*. (see glGetInternalformativ).

Upon success, `glRenderbufferStorageMultisample` deletes any existing data store for the renderbuffer image and the contents of the data store after calling `glRenderbufferStorageMultisample` are undefined.

Table 1. Sized Internal Formats

Sized Internal Format	Base Format	Red Bits	Green Bits	Blue Bits	Alpha Bits
`GL_R8`	`GL_RED`	8			
`GL_R8UI`	`GL_RED_INTEGER`	ui8			

Sized Internal Format	Base Format	Red Bits	Green Bits	Blue Bits	Alpha Bits
GL_R8I	GL_RED_INTEGER	i8			
GL_R16UI	GL_RED_INTEGER	ui16			
GL_R16I	GL_RED_INTEGER	i16			
GL_R32UI	GL_RED_INTEGER	ui32			
GL_R32I	GL_RED_INTEGER	i32			
GL_RG8	GL_RG	8	8		
GL_RG8UI	GL_RG_INTEGER	ui8	ui8		
GL_RG8I	GL_RG_INTEGER	i8	i8		
GL_RG16UI	GL_RG_INTEGER	ui16	ui16		
GL_RG16I	GL_RG_INTEGER	i16	i16		
GL_RG32UI	GL_RG_INTEGER	ui32	ui32		
GL_RG32I	GL_RG_INTEGER	i32	i32		
GL_RGB8	GL_RGB	8	8	8	
GL_RGB565	GL_RGB	5	6	5	
GL_RGBA8	GL_RGBA	8	8	8	8
GL_SRGB8_ALPHA8	GL_RGBA	8	8	8	8
GL_RGB5_A1	GL_RGBA	5	5	5	1
GL_RGBA4	GL_RGBA	4	4	4	4
GL_RGB10_A2	GL_RGBA	10	10	10	2
GL_RGBA8UI	GL_RGBA_INTEGER	ui8	ui8	ui8	ui8
GL_RGBA8I	GL_RGBA_INTEGER	i8	i8	i8	i8
GL_RGB10_A2UI	GL_RGBA_INTEGER	ui10	ui10	ui10	ui2
GL_RGBA16UI	GL_RGBA_INTEGER	ui16	ui16	ui16	ui16
GL_RGBA16I	GL_RGBA_INTEGER	i16	i16	i16	i16

Sized Internal Format	Base Format	Red Bits	Green Bits	Blue Bits	Alpha Bits
GL_RGBA32I	GL_RG-BA_INTEGER	i32	i32	i32	i32
GL_RG-BA32UI	GL_RG-BA_INTEGER	ui32	ui32	ui32	ui32

Sized Internal Format	Base Format	Depth Bits	Stencil Bits
GL_DEPTH_COMPO-NENT16	GL_DEPTH_COMPO-NENT	16	
GL_DEPTH_COMPO-NENT24	GL_DEPTH_COMPO-NENT	24	
GL_DEPTH_COMPO-NENT32F	GL_DEPTH_COMPO-NENT	f32	
GL_DEPTH24_S-TENCIL8	GL_DEPTH_STENCIL	24	8
GL_DEPTH32F_S-TENCIL8	GL_DEPTH_STENCIL	f32	8
GL_STENCIL_IN-DEX8	GL_STENCIL		8

Notes

Since different implementations may support different sample counts for multisample rendering, the actual number of samples allocated for the renderbuffer image is implementation-dependent. However, the resulting value for `GL_RENDERBUFFER_SAMPLES` is guaranteed to be greater than or equal to *`samples`* and no more than the next larger sample count supported by the implementation.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_RENDERBUFFER`.

`GL_INVALID_VALUE` is generated if *`samples`* is greater than the maximum number of samples supported for *`internalformat`*.

`GL_INVALID_ENUM` is generated if *`internalformat`* is not a color-renderable, depth-renderable, or stencil-renderable format.

`GL_INVALID_OPERATION` is generated if *`internalformat`* is a signed or unsigned integer format and *`samples`* is greater than 0.

`GL_INVALID_VALUE` is generated if either of *`width`* or *`height`* is negative, or greater than the value of `GL_MAX_RENDERBUFFER_SIZE`.

`GL_OUT_OF_MEMORY` is generated if the GL is unable to create a data store of the requested size.

API Version Support

	OpenGL ES API Version	
Function Name	2.0	3.0
glRenderbufferStorageMultisample	-	✔

See Also

glGenRenderbuffers, glGetInternalformativ, glBindRenderbuffer, glRenderbufferStorage, glFramebuffer-Renderbuffer, glDeleteRenderbuffers

Copyright

Name

glResumeTransformFeedback — resume transform feedback operations

C Specification

```
void glResumeTransformFeedback ();
```

```
void;
```

Description

`glResumeTransformFeedback` resumes transform feedback operations on the currently active transform feedback object. When transform feedback operations are paused, transform feedback is still considered active and changing most transform feedback state related to the object results in an error. However, a new transform feedback object may be bound while transform feedback is paused.

Errors

`GL_INVALID_OPERATION` is generated if the currently bound transform feedback object is not active or is not paused.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glResumeTransformFeedback	-	✔

See Also

glGenTransformFeedbacks, glBindTransformFeedback, glBeginTransformFeedback, glPauseTransformFeedback, glEndTransformFeedback, glDeleteTransformFeedbacks

Copyright

Name

glSampleCoverage — specify multisample coverage parameters

C Specification

```
void glSampleCoverage (value, invert);

GLfloat value;
GLboolean invert;
```

Parameters

value Specify a single floating-point sample coverage value. The value is clamped to the range . The initial value is 1.0.

invert Specify a single boolean value representing if the coverage masks should be inverted. `GL_TRUE` and `GL_FALSE` are accepted. The initial value is `GL_FALSE`.

Description

Multisampling samples a pixel multiple times at various implementation-dependent subpixel locations to generate antialiasing effects. Multisampling transparently antialiases points, lines, polygons, and images if it is enabled.

value is used in constructing a temporary mask used in determining which samples will be used in resolving the final fragment color. This mask is bitwise-anded with the coverage mask generated from the multisampling computation. If the *invert* flag is set, the temporary mask is inverted (all bits flipped) and then the bitwise-and is computed.

If an implementation does not have any multisample buffers available, or multisampling is disabled, rasterization occurs with only a single sample computing a pixel's final RGB color.

Provided an implementation supports multisample buffers, and multisampling is enabled, then a pixel's final color is generated by combining several samples per pixel. Each sample contains color, depth, and stencil information, allowing those operations to be performed on each sample.

Associated Gets

glGet with argument `GL_SAMPLE_COVERAGE_VALUE`

glGet with argument `GL_SAMPLE_COVERAGE_INVERT`

glIsEnabled with argument `GL_SAMPLE_ALPHA_TO_COVERAGE`

glIsEnabled with argument `GL_SAMPLE_COVERAGE`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glSampleCoverage	✔	✔

See Also

glEnable

Copyright

Name

glSamplerParameter — set sampler parameters

C Specification

```
void glSamplerParameterf (sampler, pname, param);

GLuint sampler;
GLenum pname;
GLfloat param;

void glSamplerParameteri (sampler, pname, param);

GLuint sampler;
GLenum pname;
GLint param;

void glSamplerParameterfv (sampler, pname, params);

GLuint sampler;
GLenum pname;
const GLfloat * params;

void glSamplerParameteriv (sampler, pname, params);

GLuint sampler;
GLenum pname;
const GLint * params;
```

Parameters

sampler	Specifies the sampler object whose parameter to modify.
pname	Specifies the symbolic name of a single-valued sampler parameter. *pname* can be one of the following: `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, `GL_TEXTURE_WRAP_R`, `GL_TEXTURE_MIN_FILTER`, `GL_TEXTURE_MAG_FILTER`, `GL_TEXTURE_MIN_LOD`, `GL_TEXTURE_MAX_LOD`, `GL_TEXTURE_COMPARE_MODE`, or `GL_TEXTURE_COMPARE_FUNC`.
param	For the scalar commands, specifies the value of *pname*.
params	For the vector commands (`glSamplerParameter*v`), specifies a pointer to an array where the value or values of *pname* are stored.

Description

`glSamplerParameter` assigns the value or values in *params* to the sampler parameter specified as *pname*. *sampler* specifies the sampler object to be modified, and must be the name of a sampler object previously returned from a call to glGenSamplers. The following symbols are accepted in *pname*:

`GL_TEXTURE_MIN_FILTER` The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps.

A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions , there are mipmaps. The first mipmap is the original texture, with dimensions . Each subsequent mipmap has dimensions , where are the dimensions of the previous mipmap, until either or . At that point, subsequent mipmaps have dimension or until the final mipmap, which has dimension . To define the mipmaps, call glTexStorage2D, glTexImage2D, glTexStorage2D, glTexImage3D, or glCopyTexImage2D with the *level* argument indicating the order of the mipmaps. Level 0 is the original texture; level is the final mipmap.

`params` supplies a function for minifying the texture as one of the following:

`GL_NEAREST`	Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.
`GL_LINEAR`	Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured.
`GL_NEAREST_MIPMAP_NEAREST`	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `GL_NEAREST` criterion (the texture element nearest to the center of the pixel) to produce a texture value.
`GL_LINEAR_MIPMAP_NEAREST`	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `GL_LINEAR` criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value.
`GL_NEAREST_MIPMAP_LINEAR`	Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the `GL_NEAREST` criterion (the texture element nearest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.
`GL_LINEAR_MIPMAP_LINEAR`	Chooses the two mipmaps that most closely match the size of

the pixel being textured and uses the `GL_LINEAR` criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the `GL_NEAREST` and `GL_LINEAR` minification functions can be faster than the other four, they sample only one or four texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The initial value of `GL_TEXTURE_MIN_FILTER` is `GL_NEAREST_MIPMAP_LINEAR`.

`GL_TEXTURE_MAG_FILTER` The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either `GL_NEAREST` or `GL_LINEAR` (see below). `GL_NEAREST` is generally faster than `GL_LINEAR`, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The initial value of `GL_TEXTURE_MAG_FILTER` is `GL_LINEAR`.

`GL_NEAREST` Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.

`GL_LINEAR` Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured.

`GL_TEXTURE_MIN_LOD` Sets the minimum level-of-detail parameter. This floating-point value limits the selection of highest resolution mipmap (lowest mipmap level). The initial value is -1000.

`GL_TEXTURE_MAX_LOD` Sets the maximum level-of-detail parameter. This floating-point value limits the selection of the lowest resolution mipmap (highest mipmap level). The initial value is 1000.

`GL_TEXTURE_WRAP_S` Sets the wrap parameter for texture coordinate to either `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. `GL_CLAMP_TO_EDGE` causes coordinates to be clamped to the range , where is the size of the texture in the direction of clamping. `GL_REPEAT` causes the integer part of the coordinate to be ignored; the GL uses only the fractional part, thereby creating a repeating pattern. `GL_MIRRORED_REPEAT` causes the coordinate to be set to the fractional part of the texture coordinate if the integer part of is even; if the integer part of is odd, then the texture coordinate is set to , where represents the fractional part of . Initially, `GL_TEXTURE_WRAP_S` is set to `GL_REPEAT`.

`GL_TEXTURE_WRAP_T`

Sets the wrap parameter for texture coordinate to either `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. See the discussion under `GL_TEXTURE_WRAP_S`. Initially, `GL_TEXTURE_WRAP_T` is set to `GL_REPEAT`.

`GL_TEXTURE_WRAP_R`

Sets the wrap parameter for texture coordinate to either `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. See the discussion under `GL_TEXTURE_WRAP_S`. Initially, `GL_TEXTURE_WRAP_R` is set to `GL_REPEAT`.

`GL_TEXTURE_COMPARE_MODE`

Specifies the texture comparison mode for currently bound textures. That is, a texture whose base internal format is `GL_DEPTH_COMPONENT` or `GL_DEPTH_STENCIL`; see glTexImage2D) Permissible values are:

`GL_COMPARE_REF_TO_TEXTURE`

Specifies that the interpolated and clamped texture coordinate should be compared to the value in the currently bound texture. See the discussion of `GL_TEXTURE_COMPARE_FUNC` for details of how the comparison is evaluated. The result of the comparison is assigned to the red channel.

`GL_NONE`

Specifies that the red channel should be assigned the appropriate value from the currently bound texture.

`GL_TEXTURE_COMPARE_FUNC`

Specifies the comparison operator used when `GL_TEXTURE_COMPARE_MODE` is set to `GL_COMPARE_REF_TO_TEXTURE`. Permissible values are:

Texture Comparison Function	**Computed result**
`GL_LEQUAL`	
`GL_GEQUAL`	
`GL_LESS`	
`GL_GREATER`	
`GL_EQUAL`	
`GL_NOTEQUAL`	
`GL_ALWAYS`	
`GL_NEVER`	

where is the current interpolated texture coordinate, and is the texture value sampled from the currently bound texture. is assigned to .

Notes

If a sampler object is bound to a texture unit and that unit is used to sample from a texture, the parameters in the sampler are used to sample from the texture, rather than the equivalent parameters in the texture object bound to that unit. This introduces the possibility of sampling from the same texture object with different sets of sampler state, which may lead to a condition where a texture is *incomplete* with respect to one sampler object and not with respect to another. Thus, completeness can be considered a function of a sampler object and a texture object bound to a single texture unit, rather than a property of the texture object itself.

The results of a texture lookup are undefined if:

- The sampler used in a texture lookup function is not one of the shadow sampler types, the texture object's base internal format is `GL_DEPTH_COMPONENT` or `GL_DEPTH_STENCIL`, and the `GL_TEXTURE_COMPARE_MODE` is not `GL_NONE`.
- The sampler used in a texture lookup function is one of the shadow sampler types, the texture object's base internal format is `GL_DEPTH_COMPONENT` or `GL_DEPTH_STENCIL`, and the `GL_TEXTURE_COMPARE_MODE` is `GL_NONE`.
- The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's base internal format is not `GL_DEPTH_COMPONENT` or `GL_DEPTH_STENCIL`.

Errors

`GL_INVALID_OPERATION` is generated if *`sampler`* is not the name of a sampler object previously returned from a call to glGenSamplers.

`GL_INVALID_ENUM` is generated if *`params`* should have a defined constant value (based on the value of *`pname`*) and does not.

Associated Gets

glGetSamplerParameter

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glSamplerParameterf`	-	✔
`glSamplerParameteri`	-	✔

See Also

glGenSamplers, glBindSampler, glDeleteSamplers, glIsSampler, glBindTexture, glTexParameter

Copyright

Name

glScissor — define the scissor box

C Specification

```
void glScissor (x, y, width, height);

GLint x;
GLint y;
GLsizei width;
GLsizei height;
```

Parameters

x, *y*	Specify the lower left corner of the scissor box. Initially (0, 0).
width, *height*	Specify the width and height of the scissor box. When a GL context is first attached to a window, *width* and *height* are set to the dimensions of that window.

Description

`glScissor` defines a rectangle, called the scissor box, in window coordinates. The first two arguments, *x* and *y*, specify the lower left corner of the box. *width* and *height* specify the width and height of the box.

To enable and disable the scissor test, call glEnable and glDisable with argument `GL_SCISSOR_TEST`. The test is initially disabled. While the test is enabled, only pixels that lie within the scissor box can be modified by drawing commands. Window coordinates have integer values at the shared corners of frame buffer pixels. `glScissor(0,0,1,1)` allows modification of only the lower left pixel in the window, and `glScissor(0,0,0,0)` doesn't allow modification of any pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

Errors

`GL_INVALID_VALUE` is generated if either *width* or *height* is negative.

Associated Gets

glGet with argument `GL_SCISSOR_BOX`

glIsEnabled with argument `GL_SCISSOR_TEST`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glScissor	✔	✔

See Also

glEnable, glViewport

Copyright

Name

glShaderBinary — load pre-compiled shader binaries

C Specification

```
void glShaderBinary (count, shaders, binaryFormat, binary, length);

GLsizei count;
const GLuint *shaders;
GLenum binaryFormat;
const void *binary;
GLsizei length;
```

Parameters

count	Specifies the number of shader object handles contained in *shaders*.
shaders	Specifies the address of an array of shader handles into which to load pre-compiled shader binaries.
binaryFormat	Specifies the format of the shader binaries contained in *binary*.
binary	Specifies the address of an array of bytes containing pre-compiled binary shader code.
length	Specifies the length of the array whose address is given in *binary*.

Description

`glShaderBinary` loads pre-compiled shader binary code into the *count* shader objects whose handles are given in *shaders*. *binary* points to *length* bytes of binary shader code stored in client memory. *binaryFormat* specifies the format of the pre-compiled code.

The binary image contained in *binary* will be decoded according to the extension specification defining the specified *binaryFormat* token. OpenGL ES does not define any specific binary formats, but it does provide a mechanism to obtain token vaues for such formats provided by such extensions.

Depending on the types of the shader objects in *shaders*, `glShaderBinary` will individually load binary vertex or fragment shaders, or load an executable binary that contains an optimized pair of vertex and fragment shaders stored in the same binary.

Errors

`GL_INVALID_OPERATION` is generated if more than one of the handles in *shaders* refers to the same shader object.

`GL_INVALID_ENUM` is generated if *binaryFormat* is not an accepted value.

`GL_INVALID_VALUE` is generated if the data pointed to by *binary* does not match the format specified by *binaryFormat*.

Associated Gets

glGet with parameter `GL_NUM_SHADER_BINARY_FORMATS`.

glGet with parameter `GL_SHADER_BINARY_FORMATS`.

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glShaderBinary	✔	✔

See Also

glGetProgramiv, glGetProgramBinary, glProgramBinary

Copyright

Name

glShaderSource — Replaces the source code in a shader object

C Specification

```
void glShaderSource (shader, count, string, length);

GLuint shader;
GLsizei count;
const GLchar **string;
const GLint *length;
```

Parameters

shader	Specifies the handle of the shader object whose source code is to be replaced.
count	Specifies the number of elements in the *string* and *length* arrays.
string	Specifies an array of pointers to strings containing the source code to be loaded into the shader.
length	Specifies an array of string lengths.

Description

`glShaderSource` sets the source code in *shader* to the source code in the array of strings specified by *string*. Any source code previously stored in the shader object is completely replaced. The number of strings in the array is specified by *count*. If *length* is `NULL`, each string is assumed to be null terminated. If *length* is a value other than `NULL`, it points to an array containing a string length for each of the corresponding elements of *string*. Each element in the *length* array may contain the length of the corresponding string (the null character is not counted as part of the string length) or a value less than 0 to indicate that the string is null terminated. The source code strings are not scanned or parsed at this time; they are simply copied into the specified shader object.

Notes

The GL copies the shader source code strings when `glShaderSource` is called, so an application may free its copy of the source code strings immediately after the function returns.

Errors

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_VALUE` is generated if *count* is less than 0.

Associated Gets

glGetShaderiv with arguments *shader* and `GL_SHADER_SOURCE_LENGTH`

glGetShaderSource with argument *shader*

glIsShader

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
glShaderSource	✔	✔

See Also

glCompileShader, glCreateShader, glDeleteShader

Copyright

Name

glStencilFunc — set front and back function and reference value for stencil testing

C Specification

```
void glStencilFunc (func, ref, mask);

GLenum func;
GLint ref;
GLuint mask;
```

Parameters

func Specifies the test function. Eight symbolic constants are valid: `GL_NEVER`, `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL`, and `GL_ALWAYS`. The initial value is `GL_ALWAYS`.

ref Specifies the reference value for the stencil test. Stencil comparison operations and queries of *ref* clamp its value to the range , where is the number of bitplanes in the stencil buffer. The initial value is 0.

mask Specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done. The initial value is all 1's.

Description

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. Stencil planes are first drawn into using GL drawing primitives, then geometry and images are rendered using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. To enable and disable the test, call glEnable and glDisable with argument `GL_STENCIL_TEST`. To specify actions based on the outcome of the stencil test, call glStencilOp or glStencilOpSeparate.

There can be two separate sets of *func*, *ref*, and *mask* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilFunc` sets both front and back stencil state to the same values. Use glStencilFuncSeparate to set front and back stencil state to different values.

func is a symbolic constant that determines the stencil comparison function. It accepts one of eight values, shown in the following list. *ref* is an integer reference value that is used in the stencil comparison. Stencil comparison operations and queries clamp the value to the range , where is the number of bitplanes in the stencil buffer. *mask* is bitwise ANDed with both the reference value and the stored stencil value, with the ANDed values participating in the comparison.

If *stencil* represents the value stored in the corresponding stencil buffer location, the following list shows the effect of each comparison function that can be specified by *func*. Only if the comparison succeeds is the pixel passed through to the next stage in the rasterization process (see glStencilOp). All tests treat *stencil* values as unsigned integers in the range , where is the number of bitplanes in the stencil buffer.

The following values are accepted by *func*:

`GL_NEVER`	Always fails.
`GL_LESS`	Passes if (*`ref`* & *`mask`*) < (*stencil* & *`mask`*).
`GL_LEQUAL`	Passes if (*`ref`* & *`mask`*) <= (*stencil* & *`mask`*).
`GL_GREATER`	Passes if (*`ref`* & *`mask`*) > (*stencil* & *`mask`*).
`GL_GEQUAL`	Passes if (*`ref`* & *`mask`*) >= (*stencil* & *`mask`*).
`GL_EQUAL`	Passes if (*`ref`* & *`mask`*) = (*stencil* & *`mask`*).
`GL_NOTEQUAL`	Passes if (*`ref`* & *`mask`*) != (*stencil* & *`mask`*).
`GL_ALWAYS`	Always passes.

Notes

Initially, the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil test always passes.

`glStencilFunc` is the same as calling glStencilFuncSeparate with *`face`* set to `GL_FRONT_AND_BACK`.

Errors

`GL_INVALID_ENUM` is generated if *`func`* is not one of the eight accepted values.

Associated Gets

glGet with argument `GL_STENCIL_FUNC`, `GL_STENCIL_VALUE_MASK`, `GL_STENCIL_REF`, `GL_STENCIL_BACK_FUNC`, `GL_STENCIL_BACK_VALUE_MASK`, `GL_STENCIL_BACK_REF`, or `GL_STENCIL_BITS`

glIsEnabled with argument `GL_STENCIL_TEST`

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glStencilFunc	✔	✔

See Also

glDepthFunc, glEnable, glStencilFuncSeparate, glStencilMask, glStencilMaskSeparate, glStencilOp, glStencilOpSeparate

Copyright

Name

glStencilFuncSeparate — set front and/or back function and reference value for stencil testing

C Specification

```
void glStencilFuncSeparate (face, func, ref, mask);

GLenum face;
GLenum func;
GLint ref;
GLuint mask;
```

Parameters

face Specifies whether front and/or back stencil state is updated. Three symbolic constants are valid: `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`.

func Specifies the test function. Eight symbolic constants are valid: `GL_NEVER`, `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL`, and `GL_ALWAYS`. The initial value is `GL_ALWAYS`.

ref Specifies the reference value for the stencil test. Stencil comparison operations and queries of *ref* clamp its value to the range , where is the number of bitplanes in the stencil buffer. The initial value is 0.

mask Specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done. The initial value is all 1's.

Description

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. To enable and disable the test, call glEnable and glDisable with argument `GL_STENCIL_TEST`. To specify actions based on the outcome of the stencil test, call glStencilOp or glStencilOpSeparate.

There can be two separate sets of *func*, *ref*, and *mask* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. glStencilFunc sets both front and back stencil state to the same values, as if `glStencilFuncSeparate` were called with *face* set to `GL_FRONT_AND_BACK`.

func is a symbolic constant that determines the stencil comparison function. It accepts one of eight values, shown in the following list. *ref* is an integer reference value that is used in the stencil comparison. Stencil comparison operations and queries clamp the value to the range , where is the number of bitplanes in the stencil buffer. *mask* is bitwise ANDed with both the reference value and the stored stencil value, with the ANDed values participating in the comparison.

If *stencil* represents the value stored in the corresponding stencil buffer location, the following list shows the effect of each comparison function that can be specified by *func*. Only if the comparison succeeds is the pixel passed through to the next stage in the rasterization process (see glStencilOp). All tests treat *stencil* values as unsigned integers in the range , where is the number of bitplanes in the stencil buffer.

The following values are accepted by `func`:

`GL_NEVER`	Always fails.
`GL_LESS`	Passes if (`ref` & `mask`) < (*stencil* & `mask`).
`GL_LEQUAL`	Passes if (`ref` & `mask`) <= (*stencil* & `mask`).
`GL_GREATER`	Passes if (`ref` & `mask`) > (*stencil* & `mask`).
`GL_GEQUAL`	Passes if (`ref` & `mask`) >= (*stencil* & `mask`).
`GL_EQUAL`	Passes if (`ref` & `mask`) = (*stencil* & `mask`).
`GL_NOTEQUAL`	Passes if (`ref` & `mask`) != (*stencil* & `mask`).
`GL_ALWAYS`	Always passes.

Notes

Initially, the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil test always passes.

Errors

`GL_INVALID_ENUM` is generated if `func` is not one of the eight accepted values.

Associated Gets

glGet with argument `GL_STENCIL_FUNC`, `GL_STENCIL_VALUE_MASK`, `GL_STENCIL_REF`, `GL_STENCIL_BACK_FUNC`, `GL_STENCIL_BACK_VALUE_MASK`, `GL_STENCIL_BACK_REF`, or `GL_STENCIL_BITS`

glIsEnabled with argument `GL_STENCIL_TEST`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glStencilFuncSeparate	✔	✔

See Also

glDepthFunc, glEnable, glStencilFunc, glStencilMask, glStencilMaskSeparate, glStencilOp, glStencilOpSeparate

Copyright

Name

glStencilMask — control the front and back writing of individual bits in the stencil planes

C Specification

```
void glStencilMask (mask);
```

```
GLuint mask;
```

Parameters

mask	Specifies a bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all 1's.

Description

`glStencilMask` controls the writing of individual bits in the stencil planes. The least significant bits of *mask*, where is the number of bits in the stencil buffer, specify a mask. Where a 1 appears in the mask, it's possible to write to the corresponding bit in the stencil buffer. Where a 0 appears, the corresponding bit is write-protected. Initially, all bits are enabled for writing.

There can be two separate *mask* writemasks; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilMask` sets both front and back stencil writemasks to the same values. Use glStencilMaskSeparate to set front and back stencil writemasks to different values.

Notes

`glStencilMask` is the same as calling glStencilMaskSeparate with *face* set to `GL_FRONT_AND_BACK`.

Associated Gets

glGet with argument `GL_STENCIL_WRITEMASK`, `GL_STENCIL_BACK_WRITEMASK`, or `GL_STENCIL_BITS`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glStencilMask	✔	✔

See Also

glColorMask, glDepthMask, glStencilFunc, glStencilFuncSeparate, glStencilMaskSeparate, glStencilOp, glStencilOpSeparate

Copyright

Name

glStencilMaskSeparate — control the front and/or back writing of individual bits in the stencil planes

C Specification

```
void glStencilMaskSeparate (face, mask);

GLenum face;
GLuint mask;
```

Parameters

face Specifies whether the front and/or back stencil writemask is updated. Three symbolic constants are valid: `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`.

mask Specifies a bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all 1's.

Description

`glStencilMaskSeparate` controls the writing of individual bits in the stencil planes. The least significant bits of *mask*, where is the number of bits in the stencil buffer, specify a mask. Where a 1 appears in the mask, it's possible to write to the corresponding bit in the stencil buffer. Where a 0 appears, the corresponding bit is write-protected. Initially, all bits are enabled for writing.

There can be two separate *mask* writemasks; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. glStencilMask sets both front and back stencil writemasks to the same values, as if glStencilMaskSeparate were called with *face* set to `GL_FRONT_AND_BACK`.

Errors

`GL_INVALID_ENUM` is generated if *face* is not one of the accepted tokens.

Associated Gets

glGet with argument `GL_STENCIL_WRITEMASK`, `GL_STENCIL_BACK_WRITEMASK`, or `GL_STENCIL_BITS`

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glStencilMaskSeparate	✔	✔

See Also

glColorMask, glDepthMask, glStencilFunc, glStencilFuncSeparate, glStencilMask, glStencilOp, glStencilOpSeparate

Copyright

Name

glStencilOp — set front and back stencil test actions

C Specification

```
void glStencilOp (sfail, dpfail, dppass);

GLenum sfail;
GLenum dpfail;
GLenum dppass;
```

Parameters

sfail	Specifies the action to take when the stencil test fails. Eight symbolic constants are accepted: `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_INCR_WRAP`, `GL_DECR`, `GL_DECR_WRAP`, and `GL_INVERT`. The initial value is `GL_KEEP`.
dpfail	Specifies the stencil action when the stencil test passes, but the depth test fails. *dpfail* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.
dppass	Specifies the stencil action when both the stencil test and the depth test pass, or when the stencil test passes and either there is no depth buffer or depth testing is not enabled. *dppass* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.

Description

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. To enable and disable the test, call glEnable and glDisable with argument `GL_STENCIL_TEST`; to control it, call glStencilFunc or glStencilFuncSeparate.

There can be two separate sets of *sfail*, *dpfail*, and *dppass* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilOp` sets both front and back stencil state to the same values. Use glStencilOpSeparate to set front and back stencil state to different values.

`glStencilOp` takes three arguments that indicate what happens to the stored stencil value while stenciling is enabled. If the stencil test fails, no change is made to the pixel's color or depth buffers, and *sfail* specifies what happens to the stencil buffer contents. The following eight actions are possible.

`GL_KEEP`	Keeps the current value.
`GL_ZERO`	Sets the stencil buffer value to 0.
`GL_REPLACE`	Sets the stencil buffer value to *ref*, as specified by glStencilFunc.
`GL_INCR`	Increments the current stencil buffer value. Clamps to the maximum representable unsigned value.
`GL_INCR_WRAP`	Increments the current stencil buffer value. Wraps stencil buffer value to zero when incrementing the maximum representable unsigned value.

`GL_DECR`	Decrements the current stencil buffer value. Clamps to 0.
`GL_DECR_WRAP`	Decrements the current stencil buffer value. Wraps stencil buffer value to the maximum representable unsigned value when decrementing a stencil buffer value of zero.
`GL_INVERT`	Bitwise inverts the current stencil buffer value.

Stencil buffer values are treated as unsigned integers. When incremented and decremented, values are clamped to 0 and , where is the value returned by querying `GL_STENCIL_BITS`.

The other two arguments to `glStencilOp` specify stencil buffer actions that depend on whether subsequent depth buffer tests succeed (*`dppass`*) or fail (*`dpfail`*) (see glDepthFunc). The actions are specified using the same eight symbolic constants as *`sfail`*. Note that *`dpfail`* is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, *`sfail`* and *`dppass`* specify stencil action when the stencil test fails and passes, respectively.

Notes

Initially the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil tests always pass, regardless of any call to `glStencilOp`.

`glStencilOp` is the same as calling glStencilOpSeparate with *`face`* set to `GL_FRONT_AND_BACK`.

Errors

`GL_INVALID_ENUM` is generated if *`sfail`*, *`dpfail`*, or *`dppass`* is any value other than the defined constant values.

Associated Gets

glGet with argument `GL_STENCIL_FAIL`, `GL_STENCIL_PASS_DEPTH_PASS`, `GL_STENCIL_PASS_DEPTH_FAIL`, `GL_STENCIL_BACK_FAIL`, `GL_STENCIL_BACK_PASS_DEPTH_PASS`, `GL_STENCIL_BACK_PASS_DEPTH_FAIL`, or `GL_STENCIL_BITS`

glIsEnabled with argument `GL_STENCIL_TEST`

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glStencilOp	✔	✔

See Also

glDepthFunc, glEnable, glStencilFunc, glStencilFuncSeparate, glStencilMask, glStencilMaskSeparate, glStencilOpSeparate

Copyright

Name

glStencilOpSeparate — set front and/or back stencil test actions

C Specification

```
void glStencilOpSeparate (face, sfail, dpfail, dppass);

GLenum face;
GLenum sfail;
GLenum dpfail;
GLenum dppass;
```

Parameters

face	Specifies whether front and/or back stencil state is updated. Three symbolic constants are valid: `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`.
sfail	Specifies the action to take when the stencil test fails. Eight symbolic constants are accepted: `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_INCR_WRAP`, `GL_DECR`, `GL_DECR_WRAP`, and `GL_INVERT`. The initial value is `GL_KEEP`.
dpfail	Specifies the stencil action when the stencil test passes, but the depth test fails. *dpfail* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.
dppass	Specifies the stencil action when both the stencil test and the depth test pass, or when the stencil test passes and either there is no depth buffer or depth testing is not enabled. *dppass* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.

Description

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. To enable and disable the test, call glEnable and glDisable with argument `GL_STENCIL_TEST`; to control it, call glStencilFunc or glStencilFuncSeparate.

There can be two separate sets of *sfail*, *dpfail*, and *dppass* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. glStencilOp sets both front and back stencil state to the same values, as if `glStencilOpSeparate` were called with *face* set to `GL_FRONT_AND_BACK`.

`glStencilOpSeparate` takes three arguments that indicate what happens to the stored stencil value while stenciling is enabled. If the stencil test fails, no change is made to the pixel's color or depth buffers, and *sfail* specifies what happens to the stencil buffer contents. The following eight actions are possible.

`GL_KEEP`	Keeps the current value.
`GL_ZERO`	Sets the stencil buffer value to 0.
`GL_REPLACE`	Sets the stencil buffer value to *ref*, as specified by glStencilFunc.

`GL_INCR`	Increments the current stencil buffer value. Clamps to the maximum representable unsigned value.
`GL_INCR_WRAP`	Increments the current stencil buffer value. Wraps stencil buffer value to zero when incrementing the maximum representable unsigned value.
`GL_DECR`	Decrements the current stencil buffer value. Clamps to 0.
`GL_DECR_WRAP`	Decrements the current stencil buffer value. Wraps stencil buffer value to the maximum representable unsigned value when decrementing a stencil buffer value of zero.
`GL_INVERT`	Bitwise inverts the current stencil buffer value.

Stencil buffer values are treated as unsigned integers. When incremented and decremented, values are clamped to 0 and , where is the value returned by querying `GL_STENCIL_BITS`.

The other two arguments to `glStencilOpSeparate` specify stencil buffer actions that depend on whether subsequent depth buffer tests succeed (*`dppass`*) or fail (*`dpfail`*) (see glDepthFunc). The actions are specified using the same eight symbolic constants as *`sfail`*. Note that *`dpfail`* is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, *`sfail`* and *`dppass`* specify stencil action when the stencil test fails and passes, respectively.

Notes

Initially the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil test always passes.

Errors

`GL_INVALID_ENUM` is generated if *`face`* is any value other than `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

`GL_INVALID_ENUM` is generated if *`sfail`*, *`dpfail`*, or *`dppass`* is any value other than the eight defined constant values.

Associated Gets

glGet with argument `GL_STENCIL_FAIL`, `GL_STENCIL_PASS_DEPTH_PASS`, `GL_STENCIL_PASS_DEPTH_FAIL`, `GL_STENCIL_BACK_FAIL`, `GL_STENCIL_BACK_PASS_DEPTH_PASS`, `GL_STENCIL_BACK_PASS_DEPTH_FAIL`, or `GL_STENCIL_BITS`

glIsEnabled with argument `GL_STENCIL_TEST`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glStencilOpSeparate	✔	✔

See Also

glDepthFunc, glEnable, glStencilFunc, glStencilFuncSeparate, glStencilMask, glStencilMaskSeparate, glStencilOp

Copyright

Name

glTexImage2D — specify a two-dimensional texture image

C Specification

```
void glTexImage2D (target, level, internalFormat, width, height, border,
format, type, data);
```

```
GLenum target;
GLint level;
GLint internalFormat;
GLsizei width;
GLsizei height;
GLint border;
GLenum format;
GLenum type;
const void * data;
```

Parameters

target	Specifies the target texture. Must be GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
internalFormat	Specifies the number of color components in the texture. Must be one of base internal formats given in Table 1, or one of the sized internal formats given in Table 2, below.
width	Specifies the width of the texture image. All implementations support texture images that are at least 2048 texels wide.
height	Specifies the height of the texture image. All implementations support texture images that are at least 2048 texels high.
border	This value must be 0.
format	Specifies the format of the pixel data. The following symbolic values are accepted: GL_RED, GL_RED_INTEGER, GL_RG, GL_RG_INTEGER, GL_RGB, GL_RGB_INTEGER, GL_RGBA, GL_RGBA_INTEGER, GL_DEPTH_COMPONENT, GL_DEPTH_STENCIL, GL_LUMINANCE_ALPHA, GL_LUMINANCE, and GL_ALPHA.
type	Specifies the data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_HALF_FLOAT, GL_FLOAT, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_INT_2_10_10_10_REV, GL_UNSIGNED_IN-

	`T_10F_11F_11F_REV`, `GL_UNSIGNED_INT_5_9_9_9_REV`, `GL_UNSIGNED_INT_24_8`, and `GL_FLOAT_32_UNSIGNED_IN-T_24_8_REV`.
`data`	Specifies a pointer to the image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

To define texture images, call `glTexImage2D`. The arguments describe the parameters of the texture image, such as height, width, width of the border, level-of-detail number (see glTexParameter), and number of color components provided. The last three arguments describe how the image is represented in memory.

If *`target`* is `GL_TEXTURE_2D` or one of the `GL_TEXTURE_CUBE_MAP` targets, data is read from *`data`* as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on *`type`*. These values are grouped into sets of one, two, three, or four values, depending on *`format`*, to form elements.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see glBind-Buffer) while a texture image is specified, *`data`* is treated as a byte offset into the buffer object's data store.

The first element corresponds to the lower left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper right corner of the texture image.

`format` determines the composition of each element in *`data`*. It can assume one of these symbolic values:

`GL_RED`	Each element is a single red component. For fixed point normalized components, the GL converts it to floating point, clamps to the range [0,1], and assembles it into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha.
`GL_RED_INTEGER`	Each element is a single red component. The GL performs assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha.
`GL_RG`	Each element is a red/green double. For fixed point normalized components, the GL converts each component to floating point, clamps to the range [0,1], and assembles them into an RGBA element by attaching 0.0 for blue, and 1.0 for alpha.
`GL_RG_INTEGER`	Each element is a red/green double. The GL assembles them into an RGBA element by attaching 0 for blue, and 1 for alpha.
`GL_RGB`	Each element is an RGB triple. For fixed point normalized components, the GL converts each component to floating point, clamps to the range [0,1], and assembles them into an RGBA element by attaching 1.0 for alpha.
`GL_RGB_INTEGER`	Each element is an RGB triple. The GL assembles them into an RGBA element by attaching 1 for alpha.
`GL_RGBA`	Each element contains all four components. For fixed point normalized components, the GL converts each component to floating point and clamps them to the range [0,1].
`GL_RGBA_INTEGER`	Each element contains all four components.

`GL_DEPTH_COMPONENT`	Each element is a single depth value. The GL converts it to floating point, and clamps to the range [0,1].
`GL_DEPTH_STENCIL`	Each element is a pair of depth and stencil values. The depth component of the pair is interpreted as in `GL_DEPTH_COMPONENT`. The stencil component is interpreted based on specified the depth + stencil internal format.
`GL_LUMINANCE_ALPHA`	Each element is an luminance/alpha double. The GL converts each component to floating point, clamps to the range [0,1], and assembles them into an RGBA element by placing the luminance value in the red, green and blue channels.
`GL_LUMINANCE`	Each element is a single luminance component. The GL converts it to floating point, clamps to the range [0,1], and assembles it into an RGBA element by placing the luminance value in the red, green and blue channels, and attaching 1.0 to the alpha channel.
`GL_ALPHA`	Each element is a single alpha component. The GL converts it to floating point, clamps to the range [0,1], and assembles it into an RGBA element by placing attaching 0.0 to the red, green and blue channels.

If an application wants to store the texture at a certain resolution or in a certain format, it can request the resolution and format with *`internalFormat`*. The GL will choose an internal representation with least the internal component sizes, and exactly the component types shown for that format, although it may not match exactly.

`internalFormat` may be one of the unsized (base) internal formats shown, together with valid *`format`* and *`type`* combinations, in Table 1, below

Table 1. Unsized Internal Formats

Unsized Internal Format	**Format**	**Type**	**RGBA and Luminance Values**	**Internal Components**
`GL_RGB`	`GL_RGB`	`GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT_5_6_5`	Red, Green, Blue	R, G, B
`GL_RGBA`	`GL_RGBA`	`GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_5_5_5_1`	Red, Green, Blue, Alpha	R, G, B, A
`GL_LUMINANCE_ALPHA`	`GL_LUMINANCE_ALPHA`	`GL_UNSIGNED_BYTE`	Luminance, Alpha	L, A
`GL_LUMINANCE`	`GL_LUMINANCE`	`GL_UNSIGNED_BYTE`	Luminance	L
`GL_ALPHA`	`GL_ALPHA`	`GL_UNSIGNED_BYTE`	Alpha	A

`internalFormat` may also be one of the sized internal formats shown, together with valid *`format`* and *`type`* combinations, in Table 2, below

Table 2. Sized Internal Formats

Sized Internal Format	**Format**	**Type**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**	**Shared Bits**	**Color renderable**	**Texture filterable**
`GL_R8`	`GL_RED`	`GL_UNSIGNED_BYTE`						Y	Y
`GL_R8_SNORM`	`GL_RED`	`GL_BYTE`	s8						Y

Sized Internal Format	Format	Type	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits	Color render-able	Texture filter-able
GL_R16F	GL_RED	GL_HALF_FLOAT, GL_FLOAT	f16						Y
GL_R32F	GL_RED	GL_FLOAT	f32						
GL_R8UI	GL_RED_INTEGER	GL_UNSIGNED_BYTE	ui8					Y	
GL_R8I	GL_RED_INTEGER	GL_BYTE	i8					Y	
GL_R16UI	GL_RED_INTEGER	GL_UNSIGNED_SHORT	ui16					Y	
GL_R16I	GL_RED_INTEGER	GL_SHORT	i16					Y	
GL_R32UI	GL_RED_INTEGER	GL_UNSIGNED_INT	ui32					Y	
GL_R32I	GL_RED_INTEGER	GL_INT	i32					Y	
GL_RG8	GL_RG	GL_UNSIGNED_BYTE	8	8				Y	Y
GL_RG8_SNORM	GL_RG	GL_BYTE	s8	s8					Y
GL_RG16F	GL_RG	GL_HALF_FLOAT, GL_FLOAT	f16	f16					Y
GL_RG32F	GL_RG	GL_FLOAT	f32	f32					
GL_RG8UI	GL_RG_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8				Y	
GL_RG8I	GL_RG_INTEGER	GL_BYTE	i8	i8				Y	
GL_RG16UI	GL_RG_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16				Y	
GL_RG16I	GL_RG_INTEGER	GL_SHORT	i16	i16				Y	
GL_RG32UI	GL_RG_INTEGER	GL_UNSIGNED_INT	ui32	ui32				Y	
GL_RG32I	GL_RG_INTEGER	GL_INT	i32	i32				Y	
GL_RGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8			Y	Y
GL_SRGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8				Y
GL_RGB565	GL_RGB	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_6_5	5	6	5			Y	Y
GL_RGB8_SNORM	GL_RGB	GL_BYTE	s8	s8	s8				Y
GL_R11F_G11F_B10F	GL_RGB	GL_UNSIGNED_INT_10F_11F_11F_REV, GL_HALF_FLOAT, GL_FLOAT	f11	f11	f10				Y

Sized Internal Format	**Format**	**Type**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**	**Shared Bits**	**Color render-able**	**Texture filter-able**
GL_RGB9_E5	GL_RGB	GL_UNSIGNED_INT_5_9_9_9_REV, GL_HALF_FLOAT, GL_FLOAT	9	9	9		5		Y
GL_RGB16F	GL_RGB	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16				Y
GL_RGB32F	GL_RGB	GL_FLOAT	f32	f32	f32				
GL_RGB8UI	GL_RGB_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8	ui8				
GL_RGB8I	GL_RGB_INTEGER	GL_BYTE	i8	i8	i8				
GL_RGB16UI	GL_RGB_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16	ui16				
GL_RGB16I	GL_RGB_INTEGER	GL_SHORT	i16	i16	i16				
GL_RGB32UI	GL_RGB_INTEGER	GL_UNSIGNED_INT	ui32	ui32	ui32				
GL_RGB32I	GL_RGB_INTEGER	GL_INT	i32	i32	i32				
GL_RGBA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y
GL_SRGB8_ALPHA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y
GL_RGBA8_SNORM	GL_RGBA	GL_BYTE	s8	s8	s8	s8			Y
GL_RGB5_A1	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_INT_2_10_10_10_REV	5	5	5	1		Y	Y
GL_RGBA4	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_4_4_4_4	4	4	4	4		Y	Y
GL_RGB10_A2	GL_RGBA	GL_UNSIGNED_INT_2_10_10_10_REV	10	10	10	2		Y	Y
GL_RGBA16F	GL_RGBA	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16	f16			Y

Sized Internal Format	**Format**	**Type**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**	**Shared Bits**	**Color render-able**	**Texture filter-able**
GL_RGBA32F	GL_RGBA	GL_FLOAT	f32	f32	f32	f32			
GL_RGBA8UI	GL_RGBA_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8	ui8	ui8		Y	
GL_RGBA8I	GL_RGBA_INTEGER	GL_BYTE	i8	i8	i8	i8		Y	
GL_RGB10_A2UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT_2_10_10_10_REV	ui10	ui10	ui10	ui2		Y	
GL_RGBA16UI	GL_RGBA_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16	ui16	ui16		Y	
GL_RGBA16I	GL_RGBA_INTEGER	GL_SHORT	i16	i16	i16	i16		Y	
GL_RGBA32I	GL_RGBA_INTEGER	GL_INT	i32	i32	i32	i32		Y	
GL_RGBA32UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT	ui32	ui32	ui32	ui32		Y	

Sized Internal Format	**Format**	**Type**	**Depth Bits**	**Stencil Bits**
GL_DEPTH_COMPONENT16	GL_DEPTH_COMPONENT	GL_UNSIGNED_SHORT, GL_UNSIGNED_INT	16	
GL_DEPTH_COMPONENT24	GL_DEPTH_COMPONENT	GL_UNSIGNED_INT	24	
GL_DEPTH_COMPONENT32F	GL_DEPTH_COMPONENT	GL_FLOAT	f32	
GL_DEPTH24_STENCIL8	GL_DEPTH_STENCIL	GL_UNSIGNED_INT_24_8	24	8
GL_DEPTH32F_STENCIL8	GL_DEPTH_STENCIL	GL_FLOAT_32_UNSIGNED_INT_24_8_REV	f32	8

If the *`internalFormat`* parameter is `GL_SRGB8`, or `GL_SRGB8_ALPHA8`, the texture is treated as if the red, green, or blue components are encoded in the sRGB color space. Any alpha component is left unchanged. The conversion from the sRGB encoded component to a linear component is:

Assume is the sRGB component in the range [0,1].

A one-component texture image uses only the red component of the RGBA color extracted from `data`. A two-component image uses the R and G values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Image-based shadowing can be enabled by comparing texture r coordinates to depth texture values to generate a boolean result. See glTexParameter for details on texture comparison.

Notes

The glPixelStorei mode affects texture images.

`data` may be a null pointer. In this case, texture memory is allocated to accommodate a texture of width *`width`* and height *`height`*. You can then download subtextures to initialize this texture memory. The image is undefined if the user tries to apply an uninitialized portion of the texture image to a primitive.

`glTexImage2D` specifies the two-dimensional texture for the texture object bound to the current texture unit, specified with glActiveTexture.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

`GL_INVALID_VALUE` is generated if *`target`* is one of the six cube map 2D image targets and the width and height parameters are not equal.

`GL_INVALID_ENUM` is generated if *`type`* is not a type constant.

`GL_INVALID_VALUE` is generated if *`width`* is less than 0 or greater than `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if *`level`* is less than 0.

`GL_INVALID_VALUE` may be generated if *`level`* is greater than , where *max* is the returned value of `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_ENUM` is generated if *`internalFormat`* is not one of the accepted resolution and format symbolic constants.

`GL_INVALID_VALUE` is generated if *`width`* or *`height`* is less than 0 or greater than `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if *`border`* is not 0.

`GL_INVALID_OPERATION` is generated if the combination of *`internalFormat`*, *`format`* and *`type`* is not one of those in the tables above.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *`data`* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *`type`*.

Associated Gets

glGet with argument `GL_PIXEL_UNPACK_BUFFER_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glTexImage2D	✔	✔

See Also

glActiveTexture, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage3D, glTexStorage2D, glTexStorage3D, glTexSubImage2D, glTexSubImage3D, glTexParameter

Copyright

Name

glTexImage3D — specify a three-dimensional texture image

C Specification

```
void glTexImage3D (target, level, internalFormat, width, height, depth,
border, format, type, data);
```

```
GLenum target;
GLint level;
GLint internalFormat;
GLsizei width;
GLsizei height;
GLsizei depth;
GLint border;
GLenum format;
GLenum type;
const void * data;
```

Parameters

target	Specifies the target texture. Must be one of `GL_TEXTURE_3D` or `GL_TEXTURE_2D_ARRAY`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level is the mipmap reduction image.
internalFormat	Specifies the number of color components in the texture. Must be one of base internal formats given in Table 1, or one of the sized internal formats given in Table 2, below.
width	Specifies the width of the texture image. All implementations support 3D texture images that are at least 256 texels wide.
height	Specifies the height of the texture image. All implementations support 3D texture images that are at least 256 texels high.
depth	Specifies the depth of the texture image, or the number of layers in a texture array. All implementations support 3D texture images that are at least 256 texels deep, and texture arrays that are at least 256 layers deep.
border	This value must be 0.
format	Specifies the format of the pixel data. The following symbolic values are accepted: `GL_RED`, `GL_RED_INTEGER`, `GL_RG`, `GL_RG_INTEGER`, `GL_RGB`, `GL_RGB_INTEGER`, `GL_RGBA`, `GL_RGBA_INTEGER`, `GL_DEPTH_COMPONENT`, `GL_DEPTH_STENCIL`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE`, and `GL_ALPHA`,
type	Specifies the data type of the pixel data. The following symbolic values are accepted: `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_INT_2_10_10_10_REV`, `GL_UNSIGNED_IN-`

	T_10F_11F_11F_REV, GL_UNSIGNED_INT_5_9_9_9_REV, GL_UNSIGNED_INT_24_8, and GL_FLOAT_32_UNSIGNED_INT_24_8_REV.
data	Specifies a pointer to the image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

To define texture images, call glTexImage3D. The arguments describe the parameters of the texture image, such as height, width, depth, width of the border, level-of-detail number (see glTexParameter), and number of color components provided. The last three arguments describe how the image is represented in memory.

If *target* is GL_TEXTURE_3D, data is read from *data* as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements.

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

The first element corresponds to the lower left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper right corner of the texture image.

format determines the composition of each element in *data*. It can assume one of these symbolic values:

GL_RED	Each element is a single red component. For fixed point normalized components, the GL converts it to floating point, clamps to the range [0,1], and assembles it into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha.
GL_RED_INTEGER	Each element is a single red component. The GL performs assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha.
GL_RG	Each element is a red/green double. For fixed point normalized components, the GL converts each component to floating point, clamps to the range [0,1], and assembles them into an RGBA element by attaching 0.0 for blue, and 1.0 for alpha.
GL_RG_INTEGER	Each element is a red/green double. The GL assembles them into an RGBA element by attaching 0 for blue, and 1 for alpha.
GL_RGB	Each element is an RGB triple. For fixed point normalized components, the GL converts each component to floating point, clamps to the range [0,1], and assembles them into an RGBA element by attaching 1.0 for alpha.
GL_RGB_INTEGER	Each element is an RGB triple. The GL assembles them into an RGBA element by attaching 1 for alpha.
GL_RGBA	Each element contains all four components. For fixed point normalized components, the GL converts each component to floating point and clamps them to the range [0,1].
GL_RGBA_INTEGER	Each element contains all four components.

`GL_DEPTH_COMPONENT`	Each element is a single depth value. The GL converts it to floating point, and clamps to the range [0,1].
`GL_DEPTH_STENCIL`	Each element is a pair of depth and stencil values. The depth component of the pair is interpreted as in `GL_DEPTH_COMPONENT`. The stencil component is interpreted based on specified the depth + stencil internal format.
`GL_LUMINANCE_ALPHA`	Each element is an luminance/alpha double. The GL converts each component to floating point, clamps to the range [0,1], and assembles them into an RGBA element by placing the luminance value in the red, green and blue channels.
`GL_LUMINANCE`	Each element is a single luminance component. The GL converts it to floating point, clamps to the range [0,1], and assembles it into an RGBA element by placing the luminance value in the red, green and blue channels, and attaching 1.0 to the alpha channel.
`GL_ALPHA`	Each element is a single alpha component. The GL converts it to floating point, clamps to the range [0,1], and assembles it into an RGBA element by placing attaching 0.0 to the red, green and blue channels.

If an application wants to store the texture at a certain resolution or in a certain format, it can request the resolution and format with *`internalFormat`*. The GL will choose an internal representation with least the internal component sizes, and exactly the component types shown for that format, although it may not match exactly.

`internalFormat` may be one of the unsized (base) internal formats shown, together with valid *`format`* and *`type`* combinations, in Table 1, below

Table 1. Unsized Internal Formats

Unsized Internal Format	**Format**	**Type**	**RGBA and Luminance Values**	**Internal Components**
`GL_RGB`	`GL_RGB`	`GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT_5_6_5`	Red, Green, Blue	R, G, B
`GL_RGBA`	`GL_RGBA`	`GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_5_5_5_1`	Red, Green, Blue, Alpha	R, G, B, A
`GL_LUMINANCE_ALPHA`	`GL_LUMINANCE_ALPHA`	`GL_UNSIGNED_BYTE`	Luminance, Alpha	L, A
`GL_LUMINANCE`	`GL_LUMINANCE`	`GL_UNSIGNED_BYTE`	Luminance	L
`GL_ALPHA`	`GL_ALPHA`	`GL_UNSIGNED_BYTE`	Alpha	A

`internalFormat` may also be one of the sized internal formats shown, together with valid *`format`* and *`type`* combinations, in Table 2, below

Table 2. Sized Internal Formats

Sized Internal Format	**Format**	**Type**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**	**Shared Bits**	**Color renderable**	**Texture filterable**
`GL_R8`	`GL_RED`	`GL_UNSIGNED_BYTE`	8					Y	Y
`GL_R8_SNORM`	`GL_RED`	`GL_BYTE`	s8						Y

Sized Internal Format	Format	Type	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits	Color render-able	Texture filter-able
GL_R16F	GL_RED	GL_HALF_FLOAT, GL_FLOAT	f16						Y
GL_R32F	GL_RED	GL_FLOAT	f32						
GL_R8UI	GL_RED_INTEGER	GL_UNSIGNED_BYTE	ui8					Y	
GL_R8I	GL_RED_INTEGER	GL_BYTE	i8					Y	
GL_R16UI	GL_RED_INTEGER	GL_UNSIGNED_SHORT	ui16					Y	
GL_R16I	GL_RED_INTEGER	GL_SHORT	i16					Y	
GL_R32UI	GL_RED_INTEGER	GL_UNSIGNED_INT	ui32					Y	
GL_R32I	GL_RED_INTEGER	GL_INT	i32					Y	
GL_RG8	GL_RG	GL_UNSIGNED_BYTE	8	8				Y	Y
GL_RG8_SNORM	GL_RG	GL_BYTE	s8	s8					Y
GL_RG16F	GL_RG	GL_HALF_FLOAT, GL_FLOAT	f16	f16					Y
GL_RG32F	GL_RG	GL_FLOAT	f32	f32					
GL_RG8UI	GL_RG_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8				Y	
GL_RG8I	GL_RG_INTEGER	GL_BYTE	i8	i8				Y	
GL_RG16UI	GL_RG_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16				Y	
GL_RG16I	GL_RG_INTEGER	GL_SHORT	i16	i16				Y	
GL_RG32UI	GL_RG_INTEGER	GL_UNSIGNED_INT	ui32	ui32				Y	
GL_RG32I	GL_RG_INTEGER	GL_INT	i32	i32				Y	
GL_RGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8			Y	Y
GL_SRGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8				Y
GL_RGB565	GL_RGB	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_6_5	5	6	5			Y	Y
GL_RGB8_SNORM	GL_RGB	GL_BYTE	s8	s8	s8				Y
GL_R11F_G11F_B10F	GL_RGB	GL_UNSIGNED_INT_10F_11F_11F_REV, GL_HALF_FLOAT, GL_FLOAT	f11	f11	f10				Y

Sized Internal Format	Format	Type	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits	Color render-able	Texture filter-able
GL_RGB9_E5	GL_RGB	GL_UNSIGNED_INT_5_9_9_9_REV, GL_HALF_FLOAT, GL_FLOAT	9	9	9		5		Y
GL_RGB16F	GL_RGB	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16				Y
GL_RGB32F	GL_RGB	GL_FLOAT	f32	f32	f32				
GL_RGB8UI	GL_RGB_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8	ui8				
GL_RGB8I	GL_RGB_INTEGER	GL_BYTE	i8	i8	i8				
GL_RGB16UI	GL_RGB_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16	ui16				
GL_RGB16I	GL_RGB_INTEGER	GL_SHORT	i16	i16	i16				
GL_RGB32UI	GL_RGB_INTEGER	GL_UNSIGNED_INT	ui32	ui32	ui32				
GL_RGB32I	GL_RGB_INTEGER	GL_INT	i32	i32	i32				
GL_RGBA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y
GL_SRGB8_ALPHA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y
GL_RGBA8_SNORM	GL_RGBA	GL_BYTE	s8	s8	s8	s8			Y
GL_RGB5_A1	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_INT_2_10_10_10_REV	5	5	5	1		Y	Y
GL_RGBA4	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_4_4_4_4	4	4	4	4		Y	Y
GL_RGB10_A2	GL_RGBA	GL_UNSIGNED_INT_2_10_10_10_REV	10	10	10	2		Y	Y
GL_RGBA16F	GL_RGBA	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16	f16			Y

Sized Internal Format	Format	Type	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits	Color render-able	Texture filter-able
`GL_RGBA32F`	`GL_RGBA`	`GL_FLOAT`	f32	f32	f32	f32			
`GL_RGBA8UI`	`GL_RGBA_INTEGER`	`GL_UNSIGNED_BYTE`	ui8	ui8	ui8	ui8		Y	
`GL_RGBA8I`	`GL_RGBA_INTEGER`	`GL_BYTE`	i8	i8	i8	i8		Y	
`GL_RGB10_A2UI`	`GL_RGBA_INTEGER`	`GL_UNSIGNED_INT_2_10_10_10_REV`	ui10	ui10	ui10	ui2		Y	
`GL_RGBA16UI`	`GL_RGBA_INTEGER`	`GL_UNSIGNED_SHORT`	ui16	ui16	ui16	ui16		Y	
`GL_RGBA16I`	`GL_RGBA_INTEGER`	`GL_SHORT`	i16	i16	i16	i16		Y	
`GL_RGBA32I`	`GL_RGBA_INTEGER`	`GL_INT`	i32	i32	i32	i32		Y	
`GL_RGBA32UI`	`GL_RGBA_INTEGER`	`GL_UNSIGNED_INT`	ui32	ui32	ui32	ui32		Y	

Sized Internal Format	Format	Type	Depth Bits	Stencil Bits
`GL_DEPTH_COMPONENT16`	`GL_DEPTH_COMPONENT`	`GL_UNSIGNED_SHORT`, `GL_UNSIGNED_INT`	16	
`GL_DEPTH_COMPONENT24`	`GL_DEPTH_COMPONENT`	`GL_UNSIGNED_INT`	24	
`GL_DEPTH_COMPONENT32F`	`GL_DEPTH_COMPONENT`	`GL_FLOAT`	f32	
`GL_DEPTH24_STENCIL8`	`GL_DEPTH_STENCIL`	`GL_UNSIGNED_INT_24_8`	24	8
`GL_DEPTH32F_STENCIL8`	`GL_DEPTH_STENCIL`	`GL_FLOAT_32_UNSIGNED_INT_24_8_REV`	f32	8

If the *`internalFormat`* parameter is `GL_SRGB`, `GL_SRGB8`, or `GL_SRGB8_ALPHA8`, the texture is treated as if the red, green, blue, or luminance components are encoded in the sRGB color space. Any alpha component is left unchanged. The conversion from the sRGB encoded component to a linear component is:

Assume is the sRGB component in the range [0,1].

A one-component texture image uses only the red component of the RGBA color extracted from *data*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Notes

The glPixelStorei mode affects texture images.

data may be a null pointer. In this case texture memory is allocated to accommodate a texture of width *width*, height *height*, and depth *depth*. You can then download subtextures to initialize this texture memory. The image is undefined if the user tries to apply an uninitialized portion of the texture image to a primitive.

`glTexImage3D` specifies the two-dimensional array or three-dimensional texture for the texture object bound to the current texture unit, specified with glActiveTexture.

Errors

`GL_INVALID_ENUM` is generated if *target* is not `GL_TEXTURE_3D` or `GL_TEXTURE_2D_ARRAY`.

`GL_INVALID_ENUM` is generated if *format* is not an accepted format constant. Format constants other than `GL_STENCIL_INDEX` and `GL_DEPTH_COMPONENT` are accepted.

`GL_INVALID_ENUM` is generated if *type* is not a type constant.

`GL_INVALID_VALUE` is generated if *level* is less than 0.

`GL_INVALID_VALUE` may be generated if *level* is greater than , where *max* is the returned value of `GL_MAX_3D_TEXTURE_SIZE`.

`GL_INVALID_ENUM` is generated if *internalFormat* is not one of the accepted resolution and format symbolic constants.

`GL_INVALID_VALUE` is generated if *width*, *height*, or *depth* is less than 0 or greater than `GL_MAX_3D_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if *border* is not 0 or 1.

`GL_INVALID_OPERATION` is generated if the combination of *internalFormat*, *format* and *type* is not one of those in the tables above.

`GL_INVALID_OPERATION` is generated if *target* is `GL_TEXTURE_3D` and *format* is `GL_DEPTH_COMPONENT`, or `GL_DEPTH_STENCIL`.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

Associated Gets

glGet with argument `GL_PIXEL_UNPACK_BUFFER_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glTexImage3D	-	✔

See Also

glActiveTexture, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage2D, glTexSubImage2D, glTexSubImage3D, glTexParameter

Copyright

Name

glTexParameter — set texture parameters

C Specification

```
void glTexParameterf (target, pname, param);

GLenum target;
GLenum pname;
GLfloat param;

void glTexParameteri (target, pname, param);

GLenum target;
GLenum pname;
GLint param;

void glTexParameterfv (target, pname, params);

GLenum target;
GLenum pname;
const GLfloat * params;

void glTexParameteriv (target, pname, params);

GLenum target;
GLenum pname;
const GLint * params;
```

Parameters

target Specifies the target texture, which must be either GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, or GL_TEXTURE_CUBE_MAP.

pname Specifies the symbolic name of a single-valued texture parameter. *pname* can be one of the following: GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_COMPARE_FUNC, GL_TEXTURE_COMPARE_MODE, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_LOD, GL_TEXTURE_MAX_LOD, GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_SWIZZLE_R, GL_TEXTURE_SWIZZLE_G, GL_TEXTURE_SWIZZLE_B, GL_TEXTURE_SWIZZLE_A, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, or GL_TEXTURE_WRAP_R.

param Specifies the value of *pname*.

params For the vector commands, specifies a pointer to an array where the value or values of *pname* are stored.

Description

glTexParameter assigns the value or values in *params* to the texture parameter specified as *pname*. *target* defines the target texture, either GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_2D_ARRAY, or GL_TEXTURE_3D. The following symbols are accepted in *pname*:

GL_TEXTURE_BASE_LEVEL Specifies the index of the lowest defined mipmap level. This is an integer value. The initial value is 0.

`GL_TEXTURE_COMPARE_FUNC` Specifies the comparison operator used when `GL_TEXTURE_COMPARE_MODE` is set to `GL_COMPARE_REF_TO_TEXTURE`. Permissible values are:

Texture Comparison Function	**Computed result**
`GL_LEQUAL`	
`GL_GEQUAL`	
`GL_LESS`	
`GL_GREATER`	
`GL_EQUAL`	
`GL_NOTEQUAL`	
`GL_ALWAYS`	
`GL_NEVER`	

where is the current interpolated texture coordinate, and is the depth texture value sampled from the currently bound depth texture. is assigned to the red channel.

`GL_TEXTURE_COMPARE_MODE` Specifies the texture comparison mode for currently bound depth textures. That is, a texture whose internal format is `GL_DEPTH_COMPONENT_*`; see glTexImage2D) Permissible values are:

`GL_COMPARE_REF_TO_TEXTURE`	Specifies that the interpolated and clamped texture coordinate should be compared to the value in the currently bound depth texture. See the discussion of `GL_TEXTURE_COMPARE_FUNC` for details of how the comparison is evaluated. The result of the comparison is assigned to the red channel.
`GL_NONE`	Specifies that the red channel should be assigned the appropriate value from the currently bound depth texture.

`GL_TEXTURE_MIN_FILTER` The texture minifying function is used whenever the level-of-detail function used when sampling from the texture determines that the texture should be minified. There are six defined minifying functions. Two of them use either the nearest texture elements or a weighted average of multiple texture elements to compute the texture value. The other four use mipmaps.

A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions , there are mipmaps. The first mipmap is the original texture, with dimensions . Each subsequent mipmap has dimensions , where are the dimensions of the previous mipmap, until either or . At that point, subsequent mipmaps have dimension or until the final mipmap, which has dimension . To define the mipmaps, call glTexImage2D, glTexImage3D, or glCopyTexImage2D with the *level* argument indicating the order of the mipmaps. Level 0 is the original texture; level is the final mipmap.

`params` supplies a function for minifying the texture as one of the following:

`GL_NEAREST`	Returns the value of the texture element that is nearest (in Manhattan distance) to the specified texture coordinates.
`GL_LINEAR`	Returns the weighted average of the four texture elements that are closest to the specified texture coordinates. These can include items wrapped or repeated from other parts of a texture, depending on the values of `GL_TEXTURE_WRAP_S` and `GL_TEXTURE_WRAP_T`, and on the exact mapping.
`GL_NEAREST_MIPMAP_NEAREST`	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `GL_NEAREST` criterion (the texture element closest to the specified texture coordinates) to produce a texture value.
`GL_LINEAR_MIPMAP_NEAREST`	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `GL_LINEAR` criterion (a weighted average of the four texture elements that are closest to the specified texture coordinates) to produce a texture value.
`GL_NEAREST_MIPMAP_LINEAR`	Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the `GL_NEAREST` criterion (the texture element closest to the specified texture coordinates) to produce a texture value from each mipmap. The

final texture value is a weighted average of those two values.

`GL_LINEAR_MIPMAP_LINEAR` Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the `GL_LINEAR` criterion (a weighted average of the texture elements that are closest to the specified texture coordinates) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the `GL_NEAREST` and `GL_LINEAR` minification functions can be faster than the other four, they sample only one or multiple texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The initial value of `GL_TEXTURE_MIN_FILTER` is `GL_NEAREST_MIPMAP_LINEAR`.

`GL_TEXTURE_MAG_FILTER` The texture magnification function is used whenever the level-of-detail function used when sampling from the texture determines that the texture should be magified. It sets the texture magnification function to either `GL_NEAREST` or `GL_LINEAR` (see below). `GL_NEAREST` is generally faster than `GL_LINEAR`, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The initial value of `GL_TEXTURE_MAG_FILTER` is `GL_LINEAR`.

`GL_NEAREST` Returns the value of the texture element that is nearest (in Manhattan distance) to the specified texture coordinates.

`GL_LINEAR` Returns the weighted average of the texture elements that are closest to the specified texture coordinates. These can include items wrapped or repeated from other parts of a texture, depending on the values of `GL_TEXTURE_WRAP_S` and `GL_TEXTURE_WRAP_T`, and on the exact mapping.

`GL_TEXTURE_MIN_LOD` Sets the minimum level-of-detail parameter. This floating-point value limits the selection of highest resolution mipmap (lowest mipmap level). The initial value is -1000.

`GL_TEXTURE_MAX_LOD` Sets the maximum level-of-detail parameter. This floating-point value limits the selection of the lowest resolution mipmap (highest mipmap level). The initial value is 1000.

`GL_TEXTURE_MAX_LEVEL`	Sets the index of the highest defined mipmap level. This is an integer value. The initial value is 1000.
`GL_TEXTURE_SWIZZLE_R`	Sets the swizzle that will be applied to the component of a texel before it is returned to the shader. Valid values for *`param`* are `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_ZERO` and `GL_ONE`. If `GL_TEXTURE_SWIZZLE_R` is `GL_RED`, the value for will be taken from the first channel of the fetched texel. If `GL_TEXTURE_SWIZZLE_R` is `GL_GREEN`, the value for will be taken from the second channel of the fetched texel. If `GL_TEXTURE_SWIZZLE_R` is `GL_BLUE`, the value for will be taken from the third channel of the fetched texel. If `GL_TEXTURE_SWIZZLE_R` is `GL_ALPHA`, the value for will be taken from the fourth channel of the fetched texel. If `GL_TEXTURE_SWIZZLE_R` is `GL_ZERO`, the value for will be subtituted with . If `GL_TEXTURE_SWIZZLE_R` is `GL_ONE`, the value for will be subtituted with for integer texture components, otherwise . The initial value is `GL_RED`.
`GL_TEXTURE_SWIZZLE_G`	Sets the swizzle that will be applied to the component of a texel before it is returned to the shader. Valid values for *`param`* and their effects are similar to those of `GL_TEXTURE_SWIZZLE_R`. The initial value is `GL_GREEN`.
`GL_TEXTURE_SWIZZLE_B`	Sets the swizzle that will be applied to the component of a texel before it is returned to the shader. Valid values for *`param`* and their effects are similar to those of `GL_TEXTURE_SWIZZLE_R`. The initial value is `GL_BLUE`.
`GL_TEXTURE_SWIZZLE_A`	Sets the swizzle that will be applied to the component of a texel before it is returned to the shader. Valid values for *`param`* and their effects are similar to those of `GL_TEXTURE_SWIZZLE_R`. The initial value is `GL_ALPHA`.
`GL_TEXTURE_WRAP_S`	Sets the wrap parameter for texture coordinate to either `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. `GL_CLAMP_TO_EDGE` causes coordinates to be clamped to the range , where is the size of the texture in the direction of clamping. `GL_REPEAT` causes the integer part of the coordinate to be ignored; the GL uses only the fractional part, thereby creating a repeating pattern. `GL_MIRRORED_REPEAT` causes the coordinate to be set to the fractional part of the texture coordinate if the integer part of is even; if the integer part of is odd, then the texture coordinate is set to , where represents the fractional part of . Initially, `GL_TEXTURE_WRAP_S` is set to `GL_REPEAT`.
`GL_TEXTURE_WRAP_T`	Sets the wrap parameter for texture coordinate to either `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. See the discussion under `GL_TEXTURE_WRAP_S`. Initially, `GL_TEXTURE_WRAP_T` is set to `GL_REPEAT`.
`GL_TEXTURE_WRAP_R`	Sets the wrap parameter for texture coordinate to either `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. See the discus-

sion under `GL_TEXTURE_WRAP_S`. Initially, `GL_TEXTURE_WRAP_R` is set to `GL_REPEAT`.

Notes

Suppose that a program attempts to sample from a texture and has set `GL_TEXTURE_MIN_FILTER` to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to glTexStorage2D, glTexImage2D, glTexStorage3D, glTexImage3D, or glCopyTexImage2D) do not follow the proper sequence for mipmaps (described above), or there are fewer texture images defined than are needed, or the set of texture images have differing numbers of texture components, then the texture is considered *incomplete*.

Linear filtering accesses the four nearest texture elements only in 2D textures. In 1D textures, linear filtering accesses the two nearest texture elements. In 3D textures, linear filtering accesses the eight nearest texture elements.

`glTexParameter` specifies the texture parameters for the texture object bound to the active texture unit, specified by calling glActiveTexture.

Errors

`GL_INVALID_ENUM` is generated if *`target`* or *`pname`* is not one of the accepted defined values.

`GL_INVALID_ENUM` is generated if *`params`* should have a defined constant value (based on the value of *`pname`*) and does not.

Associated Gets

glGetTexParameter

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glTexParameterf`	✔	✔
`glTexParameterfv`	✔	✔
`glTexParameteri`	✔	✔
`glTexParameteriv`	✔	✔

See Also

glActiveTexture, glBindTexture, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glSamplerParameter, glTexStorage2D, glTexImage2D, glTexStorage3D, glTexImage3D, glTexSubImage2D, glTexSubImage3D

Copyright

Name

glTexStorage2D — simultaneously specify storage for all levels of a two-dimensional texture

C Specification

```
void glTexStorage2D (target, levels, internalformat, width, height);

GLenum target;
GLsizei levels;
GLenum internalformat;
GLsizei width;
GLsizei height;
```

Parameters

target	Specify the target of the operation. *target* must be one of `GL_TEXTURE_2D`, or `GL_TEXTURE_CUBE_MAP`.
levels	Specify the number of texture levels.
internalformat	Specifies the sized internal format to be used to store texture image data.
width	Specifies the width of the texture, in texels.
height	Specifies the height of the texture, in texels.

Description

`glTexStorage2D` specifies the storage requirements for all levels of a two-dimensional texture simultaneously. Once a texture is specified with this command, the format and dimensions of all levels become immutable. The contents of the image may still be modified, however, its storage requirements may not change. Such a texture is referred to as an *immutable-format* texture.

The behavior of `glTexStorage2D` depends on the *target* parameter. When *target* is `GL_TEXTURE_2D`, calling `glTexStorage2D` is equivalent, assuming no errors are generated, to executing the following pseudo-code:

```
for (i = 0; i < levels; i++)
{
    glTexImage2D(target, i, internalformat, width, height, 0, format, type, NU
    width = max(1, (width / 2));
    height = max(1, (height / 2));
}
```

When *target* is `GL_TEXTURE_CUBE_MAP`, `glTexStorage2D` is equivalent to:

```
for (i = 0; i < levels; i++)
{
    for (face in (+X, -X, +Y, -Y, +Z, -Z))
    {
        glTexImage2D(face, i, internalformat, width, height, 0, format, type,
    }
    width = max(1, (width / 2));
```

```
        height = max(1, (height / 2));
    }
```

Since no texture data is actually provided, the values used in the pseudo-code for *`format`* and *`type`* are irrelevant and may be considered to be any values that are legal for the chosen *`internalformat`* enumerant. *`internalformat`* must be one of the sized internal formats given in Table 1, or one of the compressed internal formats given in Table 2 below. Upon success, the value of `GL_TEXTURE_IMMUTABLE_FORMAT` becomes `GL_TRUE`. The value of `GL_TEXTURE_IMMUTABLE_FORMAT` may be discovered by calling glGetTexParameter with *`pname`* set to `GL_TEXTURE_IMMUTABLE_FORMAT`. No further changes to the dimensions or format of the texture object may be made. Using any command that might alter the dimensions or format of the texture object (such as glTexImage2D or another call to `glTexStorage2D`) will result in the generation of a `GL_INVALID_OPERATION` error, even if it would not, in fact, alter the dimensions or format of the object.

Table 1. Sized Internal Formats

Sized Internal Format	Format	Type	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits	Color render-able	Texture filter-able
`GL_R8`	`GL_RED`	`GL_UNSIGNED_BYTE`	8					Y	Y
`GL_R8_SNORM`	`GL_RED`	`GL_BYTE`	s8						Y
`GL_R16F`	`GL_RED`	`GL_HALF_FLOAT`,`GL_FLOAT`	f16						Y
`GL_R32F`	`GL_RED`	`GL_FLOAT`	f32						
`GL_R8UI`	`GL_RED_INTEGER`	`GL_UNSIGNED_BYTE`	ui8					Y	
`GL_R8I`	`GL_RED_INTEGER`	`GL_BYTE`	i8					Y	
`GL_R16UI`	`GL_RED_INTEGER`	`GL_UNSIGNED_SHORT`	ui16					Y	
`GL_R16I`	`GL_RED_INTEGER`	`GL_SHORT`	i16					Y	
`GL_R32UI`	`GL_RED_INTEGER`	`GL_UNSIGNED_INT`	ui32					Y	
`GL_R32I`	`GL_RED_INTEGER`	`GL_INT`	i32					Y	
`GL_RG8`	`GL_RG`	`GL_UNSIGNED_BYTE`	8	8				Y	Y
`GL_RG8_SNORM`	`GL_RG`	`GL_BYTE`	s8	s8					Y
`GL_RG16F`	`GL_RG`	`GL_HALF_FLOAT`,`GL_FLOAT`	f16	f16					Y
`GL_RG32F`	`GL_RG`	`GL_FLOAT`	f32	f32					
`GL_RG8UI`	`GL_RG_INTEGER`	`GL_UNSIGNED_BYTE`	ui8	ui8				Y	
`GL_RG8I`	`GL_RG_INTEGER`	`GL_BYTE`	i8	i8				Y	
`GL_RG16UI`	`GL_RG_INTEGER`	`GL_UNSIGNED_SHORT`	ui16	ui16				Y	
`GL_RG16I`	`GL_RG_INTEGER`	`GL_SHORT`	i16	i16				Y	

Sized Internal Format	**Format**	**Type**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**	**Shared Bits**	**Color renderable**	**Texture filterable**
GL_RG32UI	GL_RG_INTEGER	GL_UNSIGNED_INT	ui32	ui32				Y	
GL_RG32I	GL_RG_INTEGER	GL_INT	i32	i32				Y	
GL_RGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8			Y	Y
GL_SRGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8				Y
GL_RGB565	GL_RGB	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_6_5	5	6	5			Y	Y
GL_RGB8_SNORM	GL_RGB	GL_BYTE	s8	s8	s8				Y
GL_R11F_G11F_B10F	GL_RGB	GL_UNSIGNED_INT_10F_11F_11F_REV, GL_HALF_FLOAT, GL_FLOAT	f11	f11	f10				Y
GL_RGB9_E5	GL_RGB	GL_UNSIGNED_INT_5_9_9_9_REV, GL_HALF_FLOAT, GL_FLOAT	9	9	9		5		Y
GL_RGB16F	GL_RGB	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16				Y
GL_RGB32F	GL_RGB	GL_FLOAT	f32	f32	f32				
GL_RGB8UI	GL_RGB_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8	ui8				
GL_RGB8I	GL_RGB_INTEGER	GL_BYTE	i8	i8	i8				
GL_RGB16UI	GL_RGB_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16	ui16				
GL_RGB16I	GL_RGB_INTEGER	GL_SHORT	i16	i16	i16				
GL_RGB32UI	GL_RGB_INTEGER	GL_UNSIGNED_INT	ui32	ui32	ui32				
GL_RGB32I	GL_RGB_INTEGER	GL_INT	i32	i32	i32				
GL_RGBA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y

Sized Internal Format	**Format**	**Type**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**	**Shared Bits**	**Color render-able**	**Texture filter-able**
GL_SRGB8_ALPHA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y
GL_RGBA8_SNORM	GL_RGBA	GL_BYTE	s8	s8	s8	s8			Y
GL_RGB5_A1	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_INT_2_10_10_10_REV	5	5	5	1		Y	Y
GL_RGBA4	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_4_4_4_4	4	4	4	4		Y	Y
GL_RGB10_A2	GL_RGBA	GL_UNSIGNED_INT_2_10_10_10_REV	10	10	10	2		Y	Y
GL_RGBA16F	GL_RGBA	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16	f16			Y
GL_RGBA32F	GL_RGBA	GL_FLOAT	f32	f32	f32	f32			
GL_RGBA8UI	GL_RGBA_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8	ui8	ui8		Y	
GL_RGBA8I	GL_RGBA_INTEGER	GL_BYTE	i8	i8	i8	i8		Y	
GL_RGB10_A2UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT_2_10_10_10_REV	ui10	ui10	ui10	ui2		Y	
GL_RGBA16UI	GL_RGBA_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16	ui16	ui16		Y	
GL_RGBA16I	GL_RGBA_INTEGER	GL_SHORT	i16	i16	i16	i16		Y	
GL_RGBA32I	GL_RGBA_INTEGER	GL_INT	i32	i32	i32	i32		Y	
GL_RGBA32UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT	ui32	ui32	ui32	ui32		Y	

Sized Internal Format	**Format**	**Type**	**Depth Bits**	**Stencil Bits**
GL_DEPTH_COMPONENT16	GL_DEPTH_COMPONENT	GL_UNSIGNED_SHORT, GL_UNSIGNED_INT	16	
GL_DEPTH_COMPONENT24	GL_DEPTH_COMPONENT	GL_UNSIGNED_INT	24	

Sized Internal Format	Format	Type	Depth Bits	Stencil Bits
GL_DEPTH_COMPONENT32F	GL_DEPTH_COMPONENT	GL_FLOAT	f32	
GL_DEPTH24_STENCIL8	GL_DEPTH_STENCIL	GL_UNSIGNED_INT_24_8	24	8
GL_DEPTH32F_STENCIL8	GL_DEPTH_STENCIL	GL_FLOAT_32_UNSIGNED_INT_24_8_REV	f32	8

Table 2. Compressed Internal Formats

Compressed Internal Format	Base Internal Format	Image Size
GL_COMPRESSED_R11_EAC	GL_RED	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SIGNED_R11_EAC	GL_RED	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RG11_EAC	GL_RG	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_SIGNED_RG11_EAC	GL_RG	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_RGB8_ETC2	GL_RGB	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SRGB8_ETC2	GL_RGB	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RGBA8_ETC2_EAC	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 16

Errors

GL_INVALID_OPERATION is generated if the default texture object is curently bound to *target*.

GL_INVALID_OPERATION is generated if the texture object curently bound to *target* already has GL_TEXTURE_IMMUTABLE_FORMAT set to GL_TRUE.

GL_INVALID_ENUM is generated if *internalformat* is not a valid sized internal format.

GL_INVALID_ENUM is generated if *target* is not one of the accepted target enumerants.

GL_INVALID_VALUE is generated if *width*, *height* or *levels* are less than 1.

GL_INVALID_OPERATION is generated if *levels* is greater than .

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glTexStorage2D	-	✔

See Also

glTexImage2D, glCompressedTexImage2D, glTexStorage3D.

Copyright

Name

glTexStorage3D — simultaneously specify storage for all levels of a three-dimensional or two-dimensional array texture

C Specification

`void glTexStorage3D (target, levels, internalformat, width, height, depth);`

```
GLenum target;
GLsizei levels;
GLenum internalformat;
GLsizei width;
GLsizei height;
GLsizei depth;
```

Parameters

target	Specify the target of the operation. *target* must be one of `GL_TEXTURE_3D`, or `GL_TEXTURE_2D_ARRAY`.
levels	Specify the number of texture levels.
internalformat	Specifies the sized internal format to be used to store texture image data.
width	Specifies the width of the texture, in texels.
height	Specifies the height of the texture, in texels.
depth	Specifies the depth of the texture, in texels.

Description

`glTexStorage3D` specifies the storage requirements for all levels of a three-dimensional or two-dimensional array texture simultaneously. Once a texture is specified with this command, the format and dimensions of all levels become immutable. The contents of the image may still be modified, however, its storage requirements may not change. Such a texture is referred to as an *immutable-format* texture.

The behavior of `glTexStorage3D` depends on the *target* parameter. When *target* is `GL_TEXTURE_3D`, calling `glTexStorage3D` is equivalent, assuming no errors are generated, to executing the following pseudo-code:

```
for (i = 0; i < levels; i++)
{
    glTexImage3D(target, i, internalformat, width, height, depth, 0, format, t
    width = max(1, (width / 2));
    height = max(1, (height / 2));
    depth = max(1, (depth / 2));
}
```

When *target* is `GL_TEXTURE_2D_ARRAY`, `glTexStorage3D` is equivalent to:

```
for (i = 0; i < levels; i++)
{
    glTexImage3D(target, i, internalformat, width, height, depth, 0, format, t
    width = max(1, (width / 2));
```

```
        height = max(1, (height / 2));
    }
```

Since no texture data is actually provided, the values used in the pseudo-code for *`format`* and *`type`* are irrelevant and may be considered to be any values that are legal for the chosen *`internalformat`* enumerant. *`internalformat`* must be one of the sized internal formats given in Table 1, or one of the compressed internal formats given in Table 2 below. Upon success, the value of `GL_TEXTURE_IMMUTABLE_FORMAT` becomes `GL_TRUE`. The value of `GL_TEXTURE_IMMUTABLE_FORMAT` may be discovered by calling glGetTexParameter with *`pname`* set to `GL_TEXTURE_IMMUTABLE_FORMAT`. No further changes to the dimensions or format of the texture object may be made. Using any command that might alter the dimensions or format of the texture object (such as glTexImage3D or another call to `glTexStorage3D`) will result in the generation of a `GL_INVALID_OPERATION` error, even if it would not, in fact, alter the dimensions or format of the object.

Table 1. Sized Internal Formats

Sized Internal Format	Format	Type	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits	Color render-able	Texture filter-able
`GL_R8`	`GL_RED`	`GL_UNSIGNED_BYTE`	8					Y	Y
`GL_R8_SNORM`	`GL_RED`	`GL_BYTE`	s8						Y
`GL_R16F`	`GL_RED`	`GL_HALF_FLOAT`,`GL_FLOAT`	f16						Y
`GL_R32F`	`GL_RED`	`GL_FLOAT`	f32						
`GL_R8UI`	`GL_RED_INTEGER`	`GL_UNSIGNED_BYTE`	ui8					Y	
`GL_R8I`	`GL_RED_INTEGER`	`GL_BYTE`	i8					Y	
`GL_R16UI`	`GL_RED_INTEGER`	`GL_UNSIGNED_SHORT`	ui16					Y	
`GL_R16I`	`GL_RED_INTEGER`	`GL_SHORT`	i16					Y	
`GL_R32UI`	`GL_RED_INTEGER`	`GL_UNSIGNED_INT`	ui32					Y	
`GL_R32I`	`GL_RED_INTEGER`	`GL_INT`	i32					Y	
`GL_RG8`	`GL_RG`	`GL_UNSIGNED_BYTE`	8	8				Y	Y
`GL_RG8_SNORM`	`GL_RG`	`GL_BYTE`	s8	s8					Y
`GL_RG16F`	`GL_RG`	`GL_HALF_FLOAT`,`GL_FLOAT`	f16	f16					Y
`GL_RG32F`	`GL_RG`	`GL_FLOAT`	f32	f32					
`GL_RG8UI`	`GL_RG_INTEGER`	`GL_UNSIGNED_BYTE`	ui8	ui8				Y	
`GL_RG8I`	`GL_RG_INTEGER`	`GL_BYTE`	i8	i8				Y	
`GL_RG16UI`	`GL_RG_INTEGER`	`GL_UNSIGNED_SHORT`	ui16	ui16				Y	
`GL_RG16I`	`GL_RG_INTEGER`	`GL_SHORT`	i16	i16				Y	

Sized Internal Format	Format	Type	Red Bits	Green Bits	Blue Bits	Alpha Bits	Shared Bits	Color renderable	Texture filterable
GL_RG32UI	GL_RG_INTEGER	GL_UNSIGNED_INT	ui32	ui32				Y	
GL_RG32I	GL_RG_INTEGER	GL_INT	i32	i32				Y	
GL_RGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8			Y	Y
GL_SRGB8	GL_RGB	GL_UNSIGNED_BYTE	8	8	8				Y
GL_RGB565	GL_RGB	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_6_5	5	6	5			Y	Y
GL_RGB8_SNORM	GL_RGB	GL_BYTE	s8	s8	s8				Y
GL_R11F_G11F_B10F	GL_RGB	GL_UNSIGNED_INT_10F_11F_11F_REV, GL_HALF_FLOAT, GL_FLOAT	f11	f11	f10				Y
GL_RGB9_E5	GL_RGB	GL_UNSIGNED_INT_5_9_9_9_REV, GL_HALF_FLOAT, GL_FLOAT	9	9	9		5		Y
GL_RGB16F	GL_RGB	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16				Y
GL_RGB32F	GL_RGB	GL_FLOAT	f32	f32	f32				
GL_RGB8UI	GL_RGB_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8	ui8				
GL_RGB8I	GL_RGB_INTEGER	GL_BYTE	i8	i8	i8				
GL_RGB16UI	GL_RGB_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16	ui16				
GL_RGB16I	GL_RGB_INTEGER	GL_SHORT	i16	i16	i16				
GL_RGB32UI	GL_RGB_INTEGER	GL_UNSIGNED_INT	ui32	ui32	ui32				
GL_RGB32I	GL_RGB_INTEGER	GL_INT	i32	i32	i32				
GL_RGBA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y

Sized Internal Format	**Format**	**Type**	**Red Bits**	**Green Bits**	**Blue Bits**	**Alpha Bits**	**Shared Bits**	**Color render-able**	**Texture filter-able**
GL_SRGB8_ALPHA8	GL_RGBA	GL_UNSIGNED_BYTE	8	8	8	8		Y	Y
GL_RGBA8_SNORM	GL_RGBA	GL_BYTE	s8	s8	s8	s8			Y
GL_RGB5_A1	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_INT_2_10_10_10_REV	5	5	5	1		Y	Y
GL_RGBA4	GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_4_4_4_4	4	4	4	4		Y	Y
GL_RGB10_A2	GL_RGBA	GL_UNSIGNED_INT_2_10_10_10_REV	10	10	10	2		Y	Y
GL_RGBA16F	GL_RGBA	GL_HALF_FLOAT, GL_FLOAT	f16	f16	f16	f16			Y
GL_RGBA32F	GL_RGBA	GL_FLOAT	f32	f32	f32	f32			
GL_RGBA8UI	GL_RGBA_INTEGER	GL_UNSIGNED_BYTE	ui8	ui8	ui8	ui8		Y	
GL_RGBA8I	GL_RGBA_INTEGER	GL_BYTE	i8	i8	i8	i8		Y	
GL_RGB10_A2UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT_2_10_10_10_REV	ui10	ui10	ui10	ui2		Y	
GL_RGBA16UI	GL_RGBA_INTEGER	GL_UNSIGNED_SHORT	ui16	ui16	ui16	ui16		Y	
GL_RGBA16I	GL_RGBA_INTEGER	GL_SHORT	i16	i16	i16	i16		Y	
GL_RGBA32I	GL_RGBA_INTEGER	GL_INT	i32	i32	i32	i32		Y	
GL_RGBA32UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT	ui32	ui32	ui32	ui32		Y	

Sized Internal Format	**Format**	**Type**	**Depth Bits**	**Stencil Bits**
GL_DEPTH_COMPONENT16	GL_DEPTH_COMPONENT	GL_UNSIGNED_SHORT, GL_UNSIGNED_INT	16	
GL_DEPTH_COMPONENT24	GL_DEPTH_COMPONENT	GL_UNSIGNED_INT	24	

Sized Internal Format	Format	Type	Depth Bits	Stencil Bits
GL_DEPTH_COMPONENT32F	GL_DEPTH_COMPONENT	GL_FLOAT	f32	
GL_DEPTH24_STENCIL8	GL_DEPTH_STENCIL	GL_UNSIGNED_INT_24_8	24	8
GL_DEPTH32F_STENCIL8	GL_DEPTH_STENCIL	GL_FLOAT_32_UNSIGNED_INT_24_8_REV	f32	8

Table 2. Compressed Internal Formats

Compressed Internal Format	Base Internal Format	Image Size
GL_COMPRESSED_R11_EAC	GL_RED	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SIGNED_R11_EAC	GL_RED	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RG11_EAC	GL_RG	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_SIGNED_RG11_EAC	GL_RG	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_RGB8_ETC2	GL_RGB	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SRGB8_ETC2	GL_RGB	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 8
GL_COMPRESSED_RGBA8_ETC2_EAC	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 16
GL_COMPRESSED_SRGB8_ALPHA8_ETC2_EAC	GL_RGBA	ceil(*width*/4) * ceil(*height*/4) * 16

Errors

GL_INVALID_OPERATION is generated if the default texture object is curently bound to *target*.

GL_INVALID_OPERATION is generated if the texture object curently bound to *target* already has GL_TEXTURE_IMMUTABLE_FORMAT set to GL_TRUE.

GL_INVALID_ENUM is generated if *internalformat* is not a valid sized internal format.

GL_INVALID_ENUM is generated if *target* is not one of the accepted target enumerants.

GL_INVALID_VALUE is generated if *width*, *height*, *depth* or *levels* are less than 1.

GL_INVALID_OPERATION is generated if *target* is GL_TEXTURE_3D and *levels* is greater than .

`GL_INVALID_OPERATION` is generated if *`target`* is `GL_TEXTURE_2D_ARRAY` and *`levels`* is greater than .

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glTexStorage3D	-	✔

See Also

glTexImage3D, glCompressedTexImage3D, glTexStorage2D.

Copyright

Name

glTexSubImage2D — specify a two-dimensional texture subimage

C Specification

```
void glTexSubImage2D (target, level, xoffset, yoffset, width, height,
format, type, data);

GLenum target;
GLint level;
GLint xoffset;
GLint yoffset;
GLsizei width;
GLsizei height;
GLenum format;
GLenum type;
const void * data;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
format	Specifies the format of the pixel data. The following symbolic values are accepted: `GL_RED`, `GL_RED_INTEGER`, `GL_RG`, `GL_RG_INTEGER`, `GL_RGB`, `GL_RGB_INTEGER`, `GL_RGBA`, `GL_RGBA_INTEGER`, `GL_DEPTH_COMPONENT`, `GL_DEPTH_STENCIL`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE`, and `GL_ALPHA`.
type	Specifies the data type of the pixel data. The following symbolic values are accepted: `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_10F_11F_11F_REV`, `GL_UNSIGNED_INT_5_9_9_9_REV`, `GL_UNSIGNED_INT_24_8`, and `GL_FLOAT_32_UNSIGNED_INT_24_8_REV`.
data	Specifies a pointer to the image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

`glTexSubImage2D` redefines a contiguous subregion of an existing two-dimensional texture image. The texels referenced by *`data`* replace the portion of the existing texture array with x indices *`xoffset`* and , inclusive, and y indices *`yoffset`* and , inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see glBindBuffer) while a texture image is specified, *`data`* is treated as a byte offset into the buffer object's data store.

Notes

glPixelStorei modes affect texture images.

`glTexSubImage2D` specifies a two-dimensional subtexture for the texture object bound to the current texture unit, specified with glActiveTexture.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

`GL_INVALID_ENUM` is generated if *`format`* is not an accepted format constant.

`GL_INVALID_ENUM` is generated if *`type`* is not a type constant.

`GL_INVALID_VALUE` is generated if *`level`* is less than 0.

`GL_INVALID_VALUE` may be generated if *`level`* is greater than *max*, where *max* is the returned value of `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if , , , or , where is the `GL_TEXTURE_WIDTH`, and is the `GL_TEXTURE_HEIGHT` of the texture image being modified.

`GL_INVALID_VALUE` is generated if *`width`* or *`height`* is less than 0.

`GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous glTexImage2D or glTexStorage2D operation.

`GL_INVALID_OPERATION` is generated if the combination of *`internalFormat`* of the previously specified texture array, *`format`* and *`type`* is not valid. See glTexImage2D.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *`data`* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *`type`*.

Associated Gets

glGet with argument `GL_PIXEL_UNPACK_BUFFER_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glTexSubImage2D	✔	✔

See Also

glActiveTexture, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage2D, glTexImage3D, glTexStorage2D, glTexStorage3D, glTexSubImage3D, glTexParameter

Copyright

Name

glTexSubImage3D — specify a three-dimensional texture subimage

C Specification

```
void glTexSubImage3D (target, level, xoffset, yoffset, zoffset, width,
height, depth, format, type, data);

GLenum target;
GLint level;
GLint xoffset;
GLint yoffset;
GLint zoffset;
GLsizei width;
GLsizei height;
GLsizei depth;
GLenum format;
GLenum type;
const void * data;
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_3D` or `GL_TEXTURE_2D_ARRAY`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
zoffset	Specifies a texel offset in the z direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
depth	Specifies the depth of the texture subimage.
format	Specifies the format of the pixel data. The following symbolic values are accepted: `GL_RED`, `GL_RED_INTEGER`, `GL_RG`, `GL_RG_INTEGER`, `GL_RGB`, `GL_RGB_INTEGER`, `GL_RGBA`, `GL_RGBA_INTEGER`, `GL_DEPTH_COMPONENT`, `GL_DEPTH_STENCIL`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE`, and `GL_ALPHA`.
type	Specifies the data type of the pixel data. The following symbolic values are accepted: `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_10F_11F_11F_REV`, `GL_UNSIGNED_INT_5_9_9_9_REV`, `GL_UNSIGNED_INT_24_8`, and `GL_FLOAT_32_UNSIGNED_INT_24_8_REV`.
data	Specifies a pointer to the image data in memory.

Description

Texturing allows elements of an image array to be read by shaders.

`glTexSubImage3D` redefines a contiguous subregion of an existing three-dimensional or two-dimensional array texture image. The texels referenced by *`data`* replace the portion of the existing texture array with x indices *`xoffset`* and , inclusive, y indices *`yoffset`* and , inclusive, and z indices *`zoffset`* and , inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width, height, or depth but such a specification has no effect.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see glBindBuffer) while a texture image is specified, *`data`* is treated as a byte offset into the buffer object's data store.

Notes

The glPixelStorei modes affect texture images.

`glTexSubImage3D` specifies a three-dimensional subtexture for the texture object bound to the current texture unit, specified with glActiveTexture.

Errors

`GL_INVALID_ENUM` is generated if *`target`* is not `GL_TEXTURE_3D` or `GL_TEXTURE_2D_ARRAY`.

`GL_INVALID_ENUM` is generated if *`format`* is not an accepted format constant.

`GL_INVALID_ENUM` is generated if *`type`* is not a type constant.

`GL_INVALID_VALUE` is generated if *`level`* is less than 0.

`GL_INVALID_VALUE` may be generated if *`level`* is greater than *max*, where *max* is the returned value of `GL_MAX_3D_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if , , , or , or , or , where is the `GL_TEXTURE_WIDTH`, is the `GL_TEXTURE_HEIGHT`, is the `GL_TEXTURE_DEPTH` of the texture image being modified.

`GL_INVALID_VALUE` is generated if *`width`*, *`height`*, or *`depth`* is less than 0.

`GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous glTexImage3D or glTexStorage3D operation.

`GL_INVALID_OPERATION` is generated if the combination of *`internalFormat`* of the previously specified texture array, *`format`* and *`type`* is not valid. See glTexImage3D.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *`data`* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *`type`*.

Associated Gets

glGet with argument `GL_PIXEL_UNPACK_BUFFER_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glTexSubImage3D	-	✔

See Also

glActiveTexture, glCopyTexImage2D, glCopyTexSubImage2D, glCopyTexSubImage3D, glPixelStorei, glTexImage2D, glTexImage3D, glTexStorage2D, glTexStorage3D, glTexSubImage2D, glTexParameter

Copyright

Name

glTransformFeedbackVaryings — specify values to record in transform feedback buffers

C Specification

```
void glTransformFeedbackVaryings (program, count, varyings, buffer-
Mode);
```

```
GLuint program;
GLsizei count;
const char ** varyings;
GLenum bufferMode;
```

Parameters

program	The name of the target program object.
count	The number of varying variables used for transform feedback.
varyings	An array of *count* zero-terminated strings specifying the names of the varying variables to use for transform feedback.
bufferMode	Identifies the mode used to capture the varying variables when transform feedback is active. *bufferMode* must be `GL_INTERLEAVED_ATTRIBS` or `GL_SEPARATE_ATTRIBS`.

Description

The names of the vertex shader outputs to be recorded in transform feedback mode are specified using `glTransformFeedbackVaryings` .Transform feedback records the values of the selected vertex shader outputs.

The state set by `glTranformFeedbackVaryings` is stored and takes effect next time glLinkProgram is called on *program*. When glLinkProgram is called, *program* is linked so that the values of the specified varying variables for the vertices of each primitive generated by the GL are written to a single buffer object if *bufferMode* is `GL_INTERLEAVED_ATTRIBS` or multiple buffer objects if *bufferMode* is `GL_SEPARATE_ATTRIBS`.

In addition to the errors generated by `glTransformFeedbackVaryings`, the program *program* will fail to link if:

- Any variable name specified in the *varyings* array is not declared as an output in the vertex shader.
- Any two entries in the *varyings* array specify the same varying variable.
- The total number of components to capture in any varying variable in *varyings* is greater than the constant `GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS` and the buffer mode is `GL_SEPARATE_ATTRIBS`.
- The total number of components to capture is greater than the constant `GL_MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS` and the buffer mode is `GL_INTERLEAVED_ATTRIBS`.

Errors

`GL_INVALID_VALUE` is generated if *program* is not the name of a program object.

`GL_INVALID_VALUE` is generated if *`bufferMode`* is `GL_SEPARATE_ATTRIBS` and *`count`* is greater than the implementation-dependent limit `GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

Associated Gets

glGetTransformFeedbackVarying

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glTransformFeedbackVaryings	-	✔

See Also

glBeginTransformFeedback, glEndTransformFeedback, glGetTransformFeedbackVarying

Copyright

Name

glUniform — Specify the value of a uniform variable for the current program object

C Specification

void **glUniform1f** (*location*, *v0*);

GLint *location*;
GLfloat *v0*;

void **glUniform2f** (*location*, *v0*, *v1*);

GLint *location*;
GLfloat *v0*;
GLfloat *v1*;

void **glUniform3f** (*location*, *v0*, *v1*, *v2*);

GLint *location*;
GLfloat *v0*;
GLfloat *v1*;
GLfloat *v2*;

void **glUniform4f** (*location*, *v0*, *v1*, *v2*, *v3*);

GLint *location*;
GLfloat *v0*;
GLfloat *v1*;
GLfloat *v2*;
GLfloat *v3*;

void **glUniform1i** (*location*, *v0*);

GLint *location*;
GLint *v0*;

void **glUniform2i** (*location*, *v0*, *v1*);

GLint *location*;
GLint *v0*;
GLint *v1*;

void **glUniform3i** (*location*, *v0*, *v1*, *v2*);

GLint *location*;
GLint *v0*;
GLint *v1*;
GLint *v2*;

void **glUniform4i** (*location*, *v0*, *v1*, *v2*, *v3*);

GLint *location*;
GLint *v0*;
GLint *v1*;

GLint *v2*;
GLint *v3*;

void **glUniform1ui** (*location*, *v0*);

GLint *location*;
GLuint *v0*;

void **glUniform2ui** (*location*, *v0*, *v1*);

GLint *location*;
GLuint *v0*;
GLuint *v1*;

void **glUniform3ui** (*location*, *v0*, *v1*, *v2*);

GLint *location*;
GLuint *v0*;
GLuint *v1*;
GLuint *v2*;

void **glUniform4ui** (*location*, *v0*, *v1*, *v2*, *v3*);

GLint *location*;
GLint *v0*;
GLuint *v1*;
GLuint *v2*;
GLuint *v3*;

void **glUniform1fv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLfloat **value*;

void **glUniform2fv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLfloat **value*;

void **glUniform3fv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLfloat **value*;

void **glUniform4fv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLfloat **value*;

void **glUniform1iv** (*location*, *count*, *value*);

GLint *location*;

GLsizei *count*;
const GLint **value*;

void **glUniform2iv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLint **value*;

void **glUniform3iv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLint **value*;

void **glUniform4iv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLint **value*;

void **glUniform1uiv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLuint **value*;

void **glUniform2uiv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLuint **value*;

void **glUniform3uiv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLuint **value*;

void **glUniform4uiv** (*location*, *count*, *value*);

GLint *location*;
GLsizei *count*;
const GLuint **value*;

void **glUniformMatrix2fv** (*location*, *count*, *transpose*, *value*);

GLint *location*;
GLsizei *count*;
GLboolean *transpose*;
const GLfloat **value*;

void **glUniformMatrix3fv** (*location*, *count*, *transpose*, *value*);

GLint *location*;
GLsizei *count*;

```
GLboolean transpose;
const GLfloat *value;

void glUniformMatrix4fv (location, count, transpose, value);

GLint location;
GLsizei count;
GLboolean transpose;
const GLfloat *value;

void glUniformMatrix2x3fv (location, count, transpose, value);

GLint location;
GLsizei count;
GLboolean transpose;
const GLfloat *value;

void glUniformMatrix3x2fv (location, count, transpose, value);

GLint location;
GLsizei count;
GLboolean transpose;
const GLfloat *value;

void glUniformMatrix2x4fv (location, count, transpose, value);

GLint location;
GLsizei count;
GLboolean transpose;
const GLfloat *value;

void glUniformMatrix4x2fv (location, count, transpose, value);

GLint location;
GLsizei count;
GLboolean transpose;
const GLfloat *value;

void glUniformMatrix3x4fv (location, count, transpose, value);

GLint location;
GLsizei count;
GLboolean transpose;
const GLfloat *value;

void glUniformMatrix4x3fv (location, count, transpose, value);

GLint location;
GLsizei count;
GLboolean transpose;
const GLfloat *value;
```

Parameters

location	Specifies the location of the uniform variable to be modified.

`count`	For the vector (`glUniform*v`) commands, specifies the number of elements that are to be modified. This should be 1 if the targeted uniform variable is not an array, and 1 or more if it is an array. For the matrix (`glUniformMatrix*`) commands, specifies the number of matrices that are to be modified. This should be 1 if the targeted uniform variable is not an array of matrices, and 1 or more if it is an array of matrices.
`transpose`	For the matrix commands, specifies whether to transpose the matrix as the values are loaded into the uniform variable.
`v0`, *`v1`*, *`v2`*, *`v3`*	For the scalar commands, specifies the new values to be used for the specified uniform variable.
`value`	For the vector and matrix commands, specifies a pointer to an array of *`count`* values that will be used to update the specified uniform variable.

Description

`glUniform*` modifies the value of a uniform variable or a uniform variable array in the default uniform block. The location of the uniform variable to be modified is specified by *`location`*, which should be a value returned by glGetUniformLocation. `glUniform` operates on the program object that was made part of current state by calling glUseProgram.

The commands `glUniform{1|2|3|4}{f|i|ui}` are used to change the value of the uniform variable specified by *`location`* using the values passed as arguments. The number specified in the command should match the number of components in the data type of the specified uniform variable (e.g., `1` for `float`, `int`, `unsigned int`, `bool`; `2` for `vec2`, `ivec2`, `uvec2`, `bvec2`, etc.). The suffix `f` indicates that floating-point values are being passed; the suffix `i` indicates that integer values are being passed; the suffix `ui` indicates that unsigned integer values are being passed, and this type should also match the data type of the specified uniform variable. The `i` variants of this function should be used to provide values for uniform variables defined as `int`, `ivec2`, `ivec3`, `ivec4`, or arrays of these. The `ui` variants of this function should be used to provide values for uniform variables defined as `unsigned int`, `uvec2`, `uvec3`, `uvec4`, or arrays of these. The `f` variants should be used to provide values for uniform variables of type `float`, `vec2`, `vec3`, `vec4`, or arrays of these. Either the `i`, `ui` or `f` variants may be used to provide values for uniform variables of type `bool`, `bvec2`, `bvec3`, `bvec4`, or arrays of these. The uniform variable will be set to `false` if the input value is 0 or 0.0f, and it will be set to `true` otherwise.

All active uniform variables defined in a program object are initialized to 0 when the program object is linked successfully. They retain the values assigned to them by a call to `glUniform` until the next successful link operation occurs on the program object, when they are once again initialized to 0.

The commands `glUniform{1|2|3|4}{f|i|ui}v` can be used to modify a single uniform variable or a uniform variable array. These commands pass a count and a pointer to the values to be loaded into a uniform variable or a uniform variable array. A count of 1 should be used if modifying the value of a single uniform variable, and a count of 1 or greater can be used to modify an entire array or part of an array. When loading *n* elements starting at an arbitrary position *m* in a uniform variable array, elements *m* + *n* - 1 in the array will be replaced with the new values. If *m* + *n* - 1 is larger than the size of the uniform variable array, values for all array elements beyond the end of the array will be ignored. The number specified in the name of the command indicates the number of components for each element in *`value`*, and it should match the number of components in the data type of the specified uniform variable (e.g., `1` for float, int, bool; `2` for vec2, ivec2, bvec2, etc.). The data type specified in the name of the command must match the data type for the specified uniform variable as described previously for `glUniform{1|2|3|4}{f|i|ui}`.

For uniform variable arrays, each element of the array is considered to be of the type indicated in the name of the command (e.g., `glUniform3f` or `glUniform3fv` can be used to load a uniform variable array of type vec3). The number of elements of the uniform variable array to be modified is specified by *`count`*

The commands `glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv` are used to modify a matrix or an array of matrices. The numbers in the command name are interpreted as the dimensionality of the matrix. The number `2` indicates a 2 × 2 matrix (i.e., 4 values), the number `3` indicates a 3 × 3 matrix (i.e., 9 values), and the number `4` indicates a 4 × 4 matrix (i.e., 16 values). Non-square matrix dimensionality is explicit, with the first number representing the number of columns and the second number representing the number of rows. For example, `2x4` indicates a 2 × 4 matrix with 2 columns and 4 rows (i.e., 8 values). If *`transpose`* is `GL_FALSE`, each matrix is assumed to be supplied in column major order. If *`transpose`* is `GL_TRUE`, each matrix is assumed to be supplied in row major order. The *`count`* argument indicates the number of matrices to be passed. A count of 1 should be used if modifying the value of a single matrix, and a count greater than 1 can be used to modify an array of matrices.

Notes

`glUniform1i` and `glUniform1iv` are the only two functions that may be used to load uniform variables defined as sampler types. Loading samplers with any other function will result in a `GL_INVALID_OPERATION` error.

If *`count`* is greater than 1 and the indicated uniform variable is not an array, a `GL_INVALID_OPERATION` error is generated and the specified uniform variable will remain unchanged.

Other than the preceding exceptions, if the type and size of the uniform variable as defined in the shader do not match the type and size specified in the name of the command used to load its value, a `GL_INVALID_OPERATION` error will be generated and the specified uniform variable will remain unchanged.

If *`location`* is a value other than -1 and it does not represent a valid uniform variable location in the current program object, an error will be generated, and no changes will be made to the uniform variable storage of the current program object. If *`location`* is equal to -1, the data passed in will be silently ignored and the specified uniform variable will not be changed.

Errors

`GL_INVALID_OPERATION` is generated if there is no current program object.

`GL_INVALID_OPERATION` is generated if the size of the uniform variable declared in the shader does not match the size indicated by the `glUniform` command.

`GL_INVALID_OPERATION` is generated if one of the signed or unsigned integer variants of this function is used to load a uniform variable of type `float`, `vec2`, `vec3`, `vec4`, or an array of these, or if one of the floating-point variants of this function is used to load a uniform variable of type `int`, `ivec2`, `ivec3`, `ivec4`, `unsigned int`, `uvec2`, `uvec3`, `uvec4`, or an array of these.

`GL_INVALID_OPERATION` is generated if one of the signed integer variants of this function is used to load a uniform variable of type `unsigned int`, `uvec2`, `uvec3`, `uvec4`, or an array of these.

`GL_INVALID_OPERATION` is generated if one of the unsigned integer variants of this function is used to load a uniform variable of type `int`, `ivec2`, `ivec3`, `ivec4`, or an array of these.

`GL_INVALID_OPERATION` is generated if *`location`* is an invalid uniform location for the current program object and *`location`* is not equal to -1.

`GL_INVALID_VALUE` is generated if *`count`* is less than 0.

`GL_INVALID_OPERATION` is generated if *`count`* is greater than 1 and the indicated uniform variable is not an array variable.

`GL_INVALID_OPERATION` is generated if a sampler is loaded using a command other than `glUniform1i` and `glUniform1iv`.

Associated Gets

glGet with the argument `GL_CURRENT_PROGRAM`

glGetActiveUniform with the handle of a program object and the index of an active uniform variable

glGetUniform with the handle of a program object and the location of a uniform variable

glGetUniformLocation with the handle of a program object and the name of a uniform variable

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glUniform1f`	✔	✔
`glUniform2f`	✔	✔
`glUniform3f`	✔	✔
`glUniform4f`	✔	✔
`glUniform1i`	✔	✔
`glUniform2i`	✔	✔
`glUniform3i`	✔	✔
`glUniform4i`	✔	✔
`glUniform1ui`	-	✔
`glUniform2ui`	-	✔
`glUniform3ui`	-	✔
`glUniform4ui`	-	✔
`glUniform1fv`	✔	✔
`glUniform2fv`	✔	✔
`glUniform3fv`	✔	✔
`glUniform4fv`	✔	✔
`glUniform1iv`	✔	✔
`glUniform2iv`	✔	✔
`glUniform3iv`	✔	✔
`glUniform4iv`	✔	✔
`glUniform1uiv`	-	✔
`glUniform2uiv`	-	✔
`glUniform3uiv`	-	✔
`glUniform4uiv`	-	✔

Function Name	OpenGL ES API Version	
	2.0	3.0
`glUniformMatrix2fv`	✔	✔
`glUniformMatrix3fv`	✔	✔
`glUniformMatrix4fv`	✔	✔
`glUniformMatrix2x3fv`	-	✔
`glUniformMatrix3x2fv`	-	✔
`glUniformMatrix2x4fv`	-	✔
`glUniformMatrix4x2fv`	-	✔
`glUniformMatrix3x4fv`	-	✔
`glUniformMatrix4x3fv`	-	✔

See Also

glLinkProgram, glUseProgram

Copyright

Name

glUniformBlockBinding — assign a binding point to an active uniform block

C Specification

```
void glUniformBlockBinding (program, uniformBlockIndex, uniformBlock-
Binding);

GLuint program;
GLuint uniformBlockIndex;
GLuint uniformBlockBinding;
```

Parameters

program	The name of a program object containing the active uniform block whose binding to assign.
uniformBlockIndex	The index of the active uniform block within *program* whose binding to assign.
uniformBlockBinding	Specifies the binding point to which to bind the uniform block with index *uniformBlockIndex* within *program*.

Description

Binding points for active uniform blocks are assigned using `glUniformBlockBinding`. Each of a program's active uniform blocks has a corresponding uniform buffer binding point. *program* is the name of a program object for which the command glLinkProgram has been issued in the past.

If successful, `glUniformBlockBinding` specifies that *program* will use the data store of the buffer object bound to the binding point *uniformBlockBinding* to extract the values of the uniforms in the uniform block identified by *uniformBlockIndex*.

When a program object is linked or re-linked, the uniform buffer object binding point assigned to each of its active uniform blocks is reset to zero.

Errors

`GL_INVALID_VALUE` is generated if *uniformBlockIndex* is not an active uniform block index of *program*.

`GL_INVALID_VALUE` is generated if *uniformBlockBinding* is greater than or equal to the value of `GL_MAX_UNIFORM_BUFFER_BINDINGS`.

`GL_INVALID_VALUE` is generated if *program* is not the name of a program object generated by the GL.

Associated Gets

glGet with argument `GL_MAX_UNIFORM_BUFFER_BINDINGS`

glGetActiveUniformBlockiv with argument `GL_UNIFORM_BLOCK_BINDING`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glUniformBlockBinding	-	✔

See Also

glLinkProgram, glBindBufferBase, glBindBufferRange, glGetActiveUniformBlockiv

Copyright

Name

glUseProgram — Installs a program object as part of current rendering state

C Specification

```
void glUseProgram (program);
```

```
GLuint program;
```

Parameters

program	Specifies the handle of the program object whose executables are to be used as part of current rendering state.

Description

`glUseProgram` installs the program object specified by *program* as part of current rendering state. One or more executables are created in a program object by successfully attaching shader objects to it with glAttachShader, successfully compiling the shader objects with glCompileShader, and successfully linking the program object with glLinkProgram.

A program object will contain an executable that will run on the vertex processor if it contains a shader object of type `GL_VERTEX_SHADER` that has been successfully compiled and linked. Similarly, a program object will contain an executable that will run on the fragment processor if it contains a shader object of type `GL_FRAGMENT_SHADER` that has been successfully compiled and linked.

While a program object is in use, applications are free to modify attached shader objects, compile attached shader objects, attach additional shader objects, and detach or delete shader objects. None of these operations will affect the executables that are part of the current state. However, relinking the program object that is currently in use will install the program object as part of the current rendering state if the link operation was successful (see glLinkProgram). If the program object currently in use is relinked unsuccessfully, its link status will be set to `GL_FALSE`, but the executables and associated state will remain part of the current state until a subsequent call to `glUseProgram` removes it from use. After it is removed from use, it cannot be made part of current state until it has been successfully relinked.

If *program* is zero, then the current rendering state refers to an *invalid* program object and the results of shader execution are undefined. However, this is not an error.

Notes

Like buffer and texture objects, the name space for program objects may be shared across a set of contexts, as long as the server sides of the contexts share the same address space. If the name space is shared across contexts, any attached objects and the data associated with those attached objects are shared as well.

Applications are responsible for providing the synchronization across API calls when objects are accessed from different execution threads.

Errors

`GL_INVALID_VALUE` is generated if *program* is neither 0 nor a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *`program`* could not be made part of current state.

`GL_INVALID_OPERATION` is generated if transform feedback mode is active and not paused.

Associated Gets

glGet with the argument `GL_CURRENT_PROGRAM`

glGetActiveAttrib with a valid program object and the index of an active attribute variable

glGetActiveUniform with a valid program object and the index of an active uniform variable

glGetAttachedShaders with a valid program object

glGetAttribLocation with a valid program object and the name of an attribute variable

glGetProgramiv with a valid program object and the parameter to be queried

glGetProgramInfoLog with a valid program object

glGetUniform with a valid program object and the location of a uniform variable

glGetUniformLocation with a valid program object and the name of a uniform variable

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glUseProgram	✔	✔

See Also

glAttachShader, glBindAttribLocation, glCompileShader, glCreateProgram, glDeleteProgram, glDetachShader, glLinkProgram, glUniform, glValidateProgram, glVertexAttrib

Copyright

Name

glValidateProgram — Validates a program object

C Specification

`void **glValidateProgram** (*program*);`

`GLuint *program*;`

Parameters

`program` Specifies the handle of the program object to be validated.

Description

`glValidateProgram` checks to see whether the executables contained in *`program`* can execute given the current OpenGL state. The information generated by the validation process will be stored in *`program`*'s information log. The validation information may consist of an empty string, or it may be a string containing information about how the current program object interacts with the rest of current OpenGL state. This provides a way for OpenGL implementers to convey more information about why the current program is inefficient, suboptimal, failing to execute, and so on.

The status of the validation operation will be stored as part of the program object's state. This value will be set to `GL_TRUE` if the validation succeeded, and `GL_FALSE` otherwise. It can be queried by calling glGetProgramiv with arguments *`program`* and `GL_VALIDATE_STATUS`. If validation is successful, *`program`* is guaranteed to execute given the current state. Otherwise, *`program`* is guaranteed to not execute.

This function is typically useful only during application development. The informational string stored in the information log is completely implementation dependent; therefore, an application should not expect different OpenGL implementations to produce identical information strings.

Notes

This function mimics the validation operation that OpenGL implementations must perform when rendering commands are issued while programmable shaders are part of current state. The error `GL_INVALID_OPERATION` will be generated by any command that triggers the rendering of geometry if:

- any two active samplers in the current program object are of different types, but refer to the same texture image unit,
- the number of active samplers in the program exceeds the maximum number of texture image units allowed.

It may be difficult or cause a performance degradation for applications to catch these errors when rendering commands are issued. Therefore, applications are advised to make calls to `glValidateProgram` to detect these issues during application development.

Errors

`GL_INVALID_VALUE` is generated if *`program`* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *`program`* is not a program object.

Associated Gets

glGetProgramiv with arguments *`program`* and `GL_VALIDATE_STATUS`

glGetProgramInfoLog with argument *`program`*

glIsProgram

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glValidateProgram	✔	✔

See Also

glLinkProgram, glUseProgram

Copyright

Name

glVertexAttrib — Specifies the value of a generic vertex attribute

C Specification

```
void glVertexAttrib1f (index, v0);

GLuint index;
GLfloat v0;

void glVertexAttrib2f (index, v0, v1);

GLuint index;
GLfloat v0;
GLfloat v1;

void glVertexAttrib3f (index, v0, v1, v2);

GLuint index;
GLfloat v0;
GLfloat v1;
GLfloat v2;

void glVertexAttrib4f (index, v0, v1, v2, v3);

GLuint index;
GLfloat v0;
GLfloat v1;
GLfloat v2;
GLfloat v3;

void glVertexAttribI4i (index, v0, v1, v2, v3);

GLuint index;
GLint v0;
GLint v1;
GLint v2;
GLint v3;

void glVertexAttribI4ui (index, v0, v1, v2, v3);

GLuint index;
GLuint v0;
GLuint v1;
GLuint v2;
GLuint v3;

void glVertexAttrib1fv (index, v);

GLuint index;
const GLfloat *v;

void glVertexAttrib2fv (index, v);

GLuint index;
```

```
const GLfloat *v;

void glVertexAttrib3fv (index, v);

GLuint index;
const GLfloat *v;

void glVertexAttrib4fv (index, v);

GLuint index;
const GLfloat *v;

void glVertexAttribI4iv (index, v);

GLuint index;
const GLint *v;

void glVertexAttribI4uiv (index, v);

GLuint index;
const GLuint *v;
```

Parameters

index	Specifies the index of the generic vertex attribute to be modified.
v0, *v1*, *v2*, *v3*	For the scalar commands, specifies the new values to be used for the specified vertex attribute.
v	For the vector commands (`glVertexAttrib*v`), specifies a pointer to an array of values to be used for the generic vertex attribute.

Description

The `glVertexAttrib` family of entry points allows an application to pass generic vertex attributes in numbered locations.

Generic attributes are defined as four-component values that are organized into an array. The first entry of this array is numbered 0, and the size of the array is specified by the implementation-dependent constant `GL_MAX_VERTEX_ATTRIBS`. Individual elements of this array can be modified with a `glVertexAttrib` call that specifies the index of the element to be modified and a value for that element.

These commands can be used to specify one, two, three, or all four components of the generic vertex attribute specified by *index*. A `1` in the name of the command indicates that only one value is passed, and it will be used to modify the first component of the generic vertex attribute. The second and third components will be set to 0, and the fourth component will be set to 1. Similarly, a `2` in the name of the command indicates that values are provided for the first two components, the third component will be set to 0, and the fourth component will be set to 1. A `3` in the name of the command indicates that values are provided for the first three components and the fourth component will be set to 1, whereas a `4` in the name indicates that values are provided for all four components.

The letters `f`, `i`, and `ui` indicate whether the arguments are of type float, int, or unsigned int. When `v` is appended to the name, the commands can take a pointer to an array of such values.

Additional capitalized letters can indicate further alterations to the default behavior of the glVertexAttrib function:

The commands containing `I` indicate that the arguments are extended to full signed or unsigned integers.

OpenGL ES Shading Language attribute variables are allowed to be of type mat2, mat3, or mat4. Attributes of these types may be loaded using the `glVertexAttrib` entry points. Matrices must be loaded into successive generic attribute slots in column major order, with one column of the matrix in each generic attribute slot.

A user-defined attribute variable declared in a vertex shader can be bound to a generic attribute index by calling glBindAttribLocation. This allows an application to use more descriptive variable names in a vertex shader. A subsequent change to the specified generic vertex attribute will be immediately reflected as a change to the corresponding attribute variable in the vertex shader.

The binding between a generic vertex attribute index and a user-defined attribute variable in a vertex shader is part of the state of a program object, but the current value of the generic vertex attribute is not. The value of each generic vertex attribute is part of current state, and it is maintained even if a different program object is used.

An application may freely modify generic vertex attributes that are not bound to a named vertex shader attribute variable. These values are simply maintained as part of current state and will not be accessed by the vertex shader. If a generic vertex attribute bound to an attribute variable in a vertex shader is not updated while the vertex shader is executing, the vertex shader will repeatedly use the current value for the generic vertex attribute.

Notes

Generic vertex attributes can be updated at any time.

It is possible for an application to bind more than one attribute name to the same generic vertex attribute index. This is referred to as aliasing, and it is allowed only if just one of the aliased attribute variables is active in the vertex shader, or if no path through the vertex shader consumes more than one of the attributes aliased to the same location. OpenGL implementations are not required to do error checking to detect aliasing, they are allowed to assume that aliasing will not occur, and they are allowed to employ optimizations that work only in the absence of aliasing.

The resulting attribute values are undefined if the base type of the shader attribute at slot *`index`* does not match the type of glUniform command used. If the attribute is floating point, the `glUniform*f[v]` commands should be used. If the attribute is unsigned integer, the `glUniformI4ui*` commands should be used. If the attribute is a signed integer, the `glUniformI4i*` commands should be used.

Errors

`GL_INVALID_VALUE` is generated if *`index`* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

Associated Gets

glGet with the argument `GL_CURRENT_PROGRAM`

glGetActiveAttrib with argument *`program`* and the index of an active attribute variable

glGetAttribLocation with argument *`program`* and an attribute variable name

glGetVertexAttrib with arguments `GL_CURRENT_VERTEX_ATTRIB` and *`index`*

API Version Support

Function Name	OpenGL ES API Version	
	2.0	3.0
`glVertexAttrib1f`	✔	✔
`glVertexAttrib2f`	✔	✔
`glVertexAttrib3f`	✔	✔
`glVertexAttrib4f`	✔	✔
`glVertexAttribI4i`	-	✔
`glVertexAttribI4ui`	-	✔
`glVertexAttrib1fv`	✔	✔
`glVertexAttrib2fv`	✔	✔
`glVertexAttrib3fv`	✔	✔
`glVertexAttrib4fv`	✔	✔
`glVertexAttribI4iv`	-	✔
`glVertexAttribI4uiv`	-	✔

See Also

glBindAttribLocation, glVertexAttribPointer

Copyright

Name

glVertexAttribDivisor — modify the rate at which generic vertex attributes advance during instanced rendering

C Specification

```
void glVertexAttribDivisor (index, divisor);

GLuint index;
GLuint divisor;
```

Parameters

index	Specify the index of the generic vertex attribute.
divisor	Specify the number of instances that will pass between updates of the generic attribute at slot *index*.

Description

`glVertexAttribDivisor` modifies the rate at which generic vertex attributes advance when rendering multiple instances of primitives in a single draw call (see glDrawArraysInstanced and glDrawElementsInstanced). If *divisor* is zero, the attribute at slot *index* advances once per vertex. If *divisor* is non-zero, the attribute advances once per *divisor* instances of the set(s) of vertices being rendered. An attribute is referred to as instanced if its `GL_VERTEX_ATTRIB_ARRAY_DIVISOR` value is non-zero.

index must be less than the value of `GL_MAX_VERTEX_ATTRIBUTES`.

Errors

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to the value of `GL_MAX_VERTEX_ATTRIBUTES`.

API Version Support

Function Name	OpenGL ES API Version	
	2.0	**3.0**
glVertexAttribDivisor	-	✔

See Also

glDrawArraysInstanced, glDrawElementsInstanced, glVertexAttribPointer, glEnableVertexAttribArray, glDisableVertexAttribArray

Copyright

Name

glVertexAttribPointer — define an array of generic vertex attribute data

C Specification

```
void glVertexAttribPointer (index, size, type, normalized, stride,
pointer);

GLuint index;
GLint size;
GLenum type;
GLboolean normalized;
GLsizei stride;
const void * pointer;

void glVertexAttribIPointer (index, size, type, stride, pointer);

GLuint index;
GLint size;
GLenum type;
GLsizei stride;
const void * pointer;
```

Parameters

index	Specifies the index of the generic vertex attribute to be modified.
size	Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, 4. The initial value is 4.
type	Specifies the data type of each component in the array. The symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, and `GL_UNSIGNED_INT` are accepted by both functions. Additionally `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_FIXED`, `GL_INT_2_10_10_10_REV`, and `GL_UNSIGNED_INT_2_10_10_10_REV` are accepted by `glVertexAttribPointer`. The initial value is `GL_FLOAT`.
normalized	For `glVertexAttribPointer`, specifies whether fixed-point data values should be normalized (`GL_TRUE`) or converted directly as fixed-point values (`GL_FALSE`) when they are accessed. This parameter is ignored if *type* is `GL_FIXED`.
stride	Specifies the byte offset between consecutive generic vertex attributes. If *stride* is 0, the generic vertex attributes are understood to be tightly packed in the array. The initial value is 0.
pointer	Specifies a pointer to the first generic vertex attribute in the array. If a non-zero buffer is currently bound to the `GL_ARRAY_BUFFER` target, *pointer* specifies an offset of into the array in the data store of that buffer. The initial value is 0.

Description

`glVertexAttribPointer` and `glVertexAttribIPointer` specify the location and data format of the array of generic vertex attributes at index *index* to use when rendering. *size* specifies the number of components per attribute and must be 1, 2, 3 or 4. *`type`* specifies the data type of each component, and *`stride`* specifies the byte stride from one attribute to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

For `glVertexAttribPointer`, if *`normalized`* is set to `GL_TRUE`, it indicates that values stored in an integer format are to be mapped to the range [-1,1] (for signed values) or [0,1] (for unsigned values) when they are accessed and converted to floating point. Otherwise, values will be converted to floats directly without normalization.

For `glVertexAttribIPointer`, only the integer types `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT` are accepted. Values are always left as integer values.

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see glBindBuffer), *`pointer`* is treated as a byte offset into the buffer object's data store and the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as generic vertex attribute array state (`GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`) for index *`index`*.

Client vertex arrays (a binding of zero to the `GL_ARRAY_BUFFER` target) are only valid in conjunction with the zero named vertex array object. This is provided for backwards compatibility with OpenGL ES 2.0.

When a generic vertex attribute array is specified, *`size`*, *`type`*, *`normalized`*, *`stride`*, and *`pointer`* are saved as vertex array state, in addition to the current vertex array buffer object binding.

To enable and disable a generic vertex attribute array, call glEnableVertexAttribArray and glDisableVertexAttribArray with *`index`*. If enabled, the generic vertex attribute array is used when glDrawArrays, glDrawArraysInstanced, glDrawElements, glDrawElementsIntanced, or glDrawRangeElements is called.

Notes

Each generic vertex attribute array is initially disabled and isn't accessed when glDrawElements, glDrawRangeElements, glDrawArrays, glDrawArraysInstanced, or glDrawElementsInstanced is called.

Errors

`GL_INVALID_VALUE` is generated if *`index`* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`GL_INVALID_VALUE` is generated if *`size`* is not 1, 2, 3 or 4.

`GL_INVALID_ENUM` is generated if *`type`* is not an accepted value.

`GL_INVALID_VALUE` is generated if *`stride`* is negative.

`GL_INVALID_OPERATION` is generated if *`type`* is `GL_INT_2_10_10_10_REV` or `GL_UNSIGNED_INT_2_10_10_10_REV` and *`size`* is not 4.

`GL_INVALID_OPERATION` is generated a non-zero vertex array object is bound, zero is bound to the `GL_ARRAY_BUFFER` buffer object binding point and the *`pointer`* argument is not `NULL`.

Associated Gets

glGet with argument `GL_MAX_VERTEX_ATTRIBS`

glGetVertexAttrib with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_ENABLED` glGetVertexAttrib with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_SIZE`

glGetVertexAttrib with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_TYPE`

glGetVertexAttrib with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_NORMALIZED`

glGetVertexAttrib with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_STRIDE`

glGetVertexAttrib with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`

glGet with argument `GL_ARRAY_BUFFER_BINDING`

glGetVertexAttribPointerv with arguments *`index`* and `GL_VERTEX_ATTRIB_ARRAY_POINTER`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
`glVertexAttribPointer`	✔	✔
`glVertexAttribIPointer`	-	✔

See Also

glBindAttribLocation, glBindBuffer, glDisableVertexAttribArray, glDrawArrays, glDrawElements, glDrawRangeElements, glEnableVertexAttribArray, glDrawArraysInstanced, glDrawElementsIntanced, glVertexAttrib

Copyright

Name

glViewport — set the viewport

C Specification

```
void glViewport (x, y, width, height);

GLint x;
GLint y;
GLsizei width;
GLsizei height;
```

Parameters

x, *y*	Specify the lower left corner of the viewport rectangle, in pixels. The initial value is (0,0).
width, *height*	Specify the width and height of the viewport. When a GL context is first attached to a window, *width* and *height* are set to the dimensions of that window.

Description

`glViewport` specifies the affine transformation of and from normalized device coordinates to window coordinates. Let be normalized device coordinates. Then the window coordinates are computed as follows:

Viewport width and height are silently clamped to a range that depends on the implementation. To query this range, call glGet with argument `GL_MAX_VIEWPORT_DIMS`.

Errors

`GL_INVALID_VALUE` is generated if either *width* or *height* is negative.

Associated Gets

glGet with argument `GL_VIEWPORT`

glGet with argument `GL_MAX_VIEWPORT_DIMS`

API Version Support

	OpenGL ES API Version	
Function Name	**2.0**	**3.0**
glViewport	✔	✔

See Also

glDepthRangef

Copyright

Name

glWaitSync — instruct the GL server to block until the specified sync object becomes signaled

C Specification

```
void glWaitSync (sync, flags, timeout);

GLsync sync;
GLbitfield flags;
GLuint64 timeout;
```

Parameters

sync	Specifies the sync object whose status to wait on.
flags	A bitfield controlling the command flushing behavior. *flags* must be zero.
timeout	Specifies the timeout that the server should wait before continuing. *timeout* must be `GL_TIMEOUT_IGNORED`.

Description

`glWaitSync` causes the GL server to block and wait until *sync* becomes signaled. *sync* is the name of an existing sync object upon which to wait. *flags* and *timeout* are currently not used and must be set to zero and the special value `GL_TIMEOUT_IGNORED`, respectively[1]. `glWaitSync` will always wait no longer than an implementation-dependent timeout. The duration of this timeout in nanoseconds may be queried by calling glGet with the parameter `GL_MAX_SERVER_WAIT_TIMEOUT`. There is currently no way to determine whether `glWaitSync` unblocked because the timeout expired or because the sync object being waited on was signaled.

If an error occurs, `glWaitSync` does not cause the GL server to block.

Errors

`GL_INVALID_OPERATION` is generated if *sync* is not the name of a sync object.

`GL_INVALID_VALUE` is generated if *flags* is not zero.

`GL_INVALID_VALUE` is generated if *timeout* is not `GL_TIMEOUT_IGNORED`.

API Version Support

Function Name	**OpenGL ES API Version**	
	2.0	**3.0**
glWaitSync	-	✔

See Also

glFenceSync, glClientWaitSync

[1]*flags* and *timeout* are placeholders for anticipated future extensions of sync object capabilities. They must have these reserved values in order that existing code calling `glWaitSync` operate properly in the presence of such extensions.

Copyright

Name

gl_FragCoord — contains the window-relative coordinates of the current fragment

Declaration

```
                    in highp
                    vec4
                    gl_FragCoord
                ;
```

Description

Available only in the fragment language, `gl_FragCoord` is an input variable that contains the window relative coordinate (x, y, z, 1/w) values for the fragment. If multi-sampling, this value can be for any location within the pixel, or one of the fragment samples. This value is the result of fixed functionality that interpolates primitives after vertex processing to generate fragments. The z component is the depth value that would be used for the fragment's depth if no shader contained any writes to gl_FragDepth.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_FragCoord	✔	✔

See Also

gl_FragDepth

Copyright

Name

gl_FragDepth — establishes a depth value for the current fragment

Declaration

```
                    out highp
                    float
                    gl_FragDepth
               ;
```

Description

Available only in the fragment language, `gl_FragDepth` is an output variable that is used to establish the depth value for the current fragment. If depth buffering is enabled and no shader writes to `gl_FragDepth`, then the fixed varname value for depth will be used (this value is contained in the z component of gl_FragCoord) otherwise, the value written to `gl_FragDepth` is used. If a shader statically assigns to `gl_FragDepth`, then the value of the fragment's depth may be undefined for executions of the shader that don't take that path. That is, if the set of linked fragment shaders statically contain a write to `gl_FragDepth`, then it is responsible for always writing it.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_FragDepth	-	✔

See Also

gl_FragCoord

Copyright

Name

gl_FrontFacing — indicates whether a primitive is front or back facing

Declaration

```
                 in
                 bool
                 gl_FrontFacing
            ;
```

Description

Available only in the fragment language, `gl_FrontFacing` is an input variable whose value is `true` if the fragment belongs to a front-facing primitive and false otherwise. The determination of whether a triangle primitive is front-facing is made by examining the sign of the area of the triangle, including a possible reversal of this sign as controlled by `glFrontFace`. One way to compute this area is:

where and are the x and y window coordinates of the *i*th vertex of the n-vertex polygon.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_FrontFacing	✔	✔

See Also

gl_FragCoord

Copyright

Name

gl_InstanceID — contains the instance number of the current primitive in an instanced draw command

Declaration

```
                in highp
                int
                gl_InstanceID
        ;
```

Description

`gl_InstanceID` is a vertex language input variable that holds the integer instance number of the current primitive in an instanced draw command such as `glDrawArraysInstanced`. If the current primitive does not originate from an instanced draw command, the value of `gl_InstanceID` is zero.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_InstanceID	-	✔

See Also

gl_VertexID

Copyright

Name

gl_PointCoord — contains the coordinate of a fragment within a point

Declaration

```
                    in mediump
                    vec2
                    gl_PointCoord
                ;
```

Description

`gl_PointCoord` is a fragment language input variable that contains the two-dimensional coordinates indicating where within a point primitive the current fragment is located. If the current primitive is not a point, then values read from `gl_PointCoord` are undefined.

`gl_PointCoord.s` ranges from 0.0 to 1.0 across the point horizontally from left to right. `gl_PointCoord.t` varies from 0.0 to 1.0 vertically from top to bottom.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_PointCoord	✔	✔

See Also

gl_FragCoord, gl_FragDepth

Copyright

Name

gl_PointSize — contains size of rasterized points, in pixels

Declaration

```
          out highp
          float
          gl_PointSize
      ;
```

Description

The variable `gl_PointSize` is intended for a vertex shader to write the size of the point to be rasterized. It is measured in pixels. If `gl_PointSize` is not written to, its value is undefined in subsequent pipeline stages.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_PointSize	✔	✔

See Also

gl_Position

Copyright

Name

gl_Position — contains the position of the current vertex

Declaration

```
          out highp
          vec4
          gl_Position
      ;
```

Description

The variable `gl_Position` is intended for writing the homogeneous vertex position. It can be written at any time during vertexshader execution. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations, if present, that operate on primitives after vertex processing has occurred. Its value is undefined after the vertex processing stage if the vertex shader executable does not write `gl_Position`.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_Position	✔	✔

See Also

gl_PointSize

Copyright

Name

gl_VertexID — contains the index of the current vertex

Declaration

```
                    in highp
                    int
                    gl_VertexID
               ;
```

Description

`gl_VertexID` is a vertex language input variable that holds an integer index for the vertex. The index is implicitly generated by `glDrawArrays` and other commands that do not reference the content of the `GL_ELEMENT_ARRAY_BUFFER`, or explicitly generated from the content of the `GL_ELEMENT_ARRAY_BUFFER` by commands such as `glDrawElements`.

Version Support

	OpenGL ES Shading Language Version	
Variable Name	**1.00**	**3.00**
gl_VertexID	-	✔

See Also

gl_InstanceID

Copyright

Name

greaterThan — perform a component-wise greater-than comparison of two vectors

Declaration

```
bvec greaterThan (x, y);

vec x;
vec y;

bvec greaterThan (x, y);

ivec x;
ivec y;

bvec greaterThan (x, y);

uvec x;
uvec y;
```

Parameters

x Specifies the first vector to be used in the comparison operation.

y Specifies the second vector to be used in the comparison operation.

Description

`greaterThan` returns a boolean vector in which each element *i* is computed as *x*[*i*] > *y*[*i*].

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
greaterThan (vec)	✔	✔
greaterThan (ivec)	✔	✔
greaterThan (uvec)	-	✔

See Also

lessThan, lessThanEqual, greaterThanEqual, equal, notEqual, any, all, not

Copyright

Name

greaterThanEqual — perform a component-wise greater-than-or-equal comparison of two vectors

Declaration

```
bvec greaterThanEqual (x, y);

vec x;
vec y;

bvec greaterThanEqual (x, y);

ivec x;
ivec y;

bvec greaterThanEqual (x, y);

uvec x;
uvec y;
```

Parameters

x Specifies the first vector to be used in the comparison operation.

y Specifies the second vector to be used in the comparison operation.

Description

`greaterThanEqual` returns a boolean vector in which each element *i* is computed as $x[i] \geq y[i]$.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
greaterThanEqual (vec)	✔	✔
greaterThanEqual (ivec)	✔	✔
greaterThanEqual (uvec)	-	✔

See Also

lessThan, lessThanEqual, greaterThan, equal, notEqual, any, all, not

Copyright

Name

intBitsToFloat, uintBitsToFloat — produce a floating point using an encoding supplied as an integer

Declaration

```
genType intBitsToFloat (x);

genIType x;

genType uintBitsToFloat (x);

genUType x;
```

Parameters

x Specifies the bit encoding to return as a floating point value.

Description

`intBitsToFloat` and `uintBitsToFloat` return the encoding passed in parameter *x* as a highp floating-point value. If the encoding of a NaN is passed in *x*, it will not signal and the resulting value will be undefined. If the encoding of a floating point infinity is passed in parameter *x*, the resulting floating-point value is the corresponding (positive or negative) floating point infinity. For lowp and mediump, the value is first converted to the corresponding signed or unsigned highp integer and then reinterpreted as a highp floating point value as before.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
intBitsToFloat	-	✔
uintBitsToFloat	-	✔

See Also

floatBitsToInt, floatBitsToUint, isnan, isinf

Copyright

Name

inverse — calculate the inverse of a matrix

Declaration

```
mat2 inverse (m);

mat2 m;

mat3 inverse (m);

mat3 m;

mat4 inverse (m);

mat4 m;
```

Parameters

m Specifies the matrix of which to take the inverse.

Description

`inverse` returns the inverse of the matrix *m*. The values in the returned matrix are undefined if *m* is singular or poorly-conditioned (nearly singular).

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
inverse (genType)	-	✔

See Also

transpose, determinant

Copyright

Name

inversesqrt — return the inverse of the square root of the parameter

Declaration

```
genType inversesqrt (x);

genType x;
```

Parameters

x Specify the value of which to take the inverse of the square root.

Description

`inversesqrt` returns the inverse of the square root of x; i.e. the value $1 \over { \sqrt x }$. The result is undefined if $x \leq 0$.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
inversesqrt (genType)	✔	✔

See Also

sqrt

Copyright

Name

isinf — determine whether the parameter is positive or negative infinity

Declaration

```
genBType isinf (x);
genType x;
```

Parameters

x Specifies the value to test for infinity.

Description

For each element *i* of the result, `isinf` returns `true` if *x*[*i*] is positive or negative floating point infinity and false otherwise.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
isinf (genType)	-	✔

See Also

isnan

Copyright

Name

isnan — determine whether the parameter is a number

Declaration

```
genBType isnan (x);

genType x;
```

Parameters

x Specifies the value to test for NaN.

Description

For each element *i* of the result, `isnan` returns `true` if *x*[*i*] is positive or negative floating point NaN (Not a Number) and false otherwise. NaNs may not be supported by the implementation, in which case `isnan` returns false.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
isnan (genType)	-	✔

See Also

isinf

Copyright

Name

length — calculate the length of a vector

Declaration

```
float length (x);
```

```
genType x;
```

Parameters

x Specifies a vector of which to calculate the length.

Description

`length` returns the length of the vector, i.e. $\sqrt { { x[0] }^2 + { x[1] }^2 + \dots }$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
length (genType)	✔	✔

See Also

distance, normalize

Copyright

Name

lessThan — perform a component-wise less-than comparison of two vectors

Declaration

```
bvec lessThan (x, y);

vec x;
vec y;

bvec lessThan (x, y);

ivec x;
ivec y;

bvec lessThan (x, y);

uvec x;
uvec y;
```

Parameters

x Specifies the first vector to be used in the comparison operation.

y Specifies the second vector to be used in the comparison operation.

Description

`lessThan` returns a boolean vector in which each element *i* is computed as $x[i] < y[i]$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
lessThan (vec)	✔	✔
lessThan (ivec)	✔	✔
lessThan (uvec)	-	✔

See Also

lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, any, all, not

Copyright

Name

lessThanEqual — perform a component-wise less-than-or-equal comparison of two vectors

Declaration

```
bvec lessThanEqual (x, y);

vec x;
vec y;

bvec lessThanEqual (x, y);

ivec x;
ivec y;

bvec lessThanEqual (x, y);

uvec x;
uvec y;
```

Parameters

x Specifies the first vector to be used in the comparison operation.

y Specifies the second vector to be used in the comparison operation.

Description

`lessThanEqual` returns a boolean vector in which each element *i* is computed as $x[i] \leq y[i]$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
lessThanEqual (vec)	✔	✔
lessThanEqual (ivec)	✔	✔
lessThanEqual (uvec)	-	✔

See Also

lessThan, greaterThan, greaterThanEqual, equal, notEqual, any, all, not

Copyright

Name

log — return the natural logarithm of the parameter

Declaration

```
genType log (x);
genType x;
```

Parameters

x Specify the value of which to take the natural logarithm.

Description

`log` returns the natural logarithm of x, i.e. the value y which satisfies $x = e^y$. The result is undefined if $x \leq 0$.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	3.00
log	✔	✔

See Also

exp, exp2, log2

Copyright

Name

log2 — return the base 2 logarithm of the parameter

Declaration

```
genType log2 (x);
```

```
genType x;
```

Parameters

x Specify the value of which to take the base 2 logarithm.

Description

`log2` returns the base 2 logarithm of x, i.e. the value y which satisfies $x = 2^y$. The result is undefined if $x \leq 0$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
log2	✔	✔

See Also

exp, log, exp2

Copyright

Name

matrixCompMult — perform a component-wise multiplication of two matrices

Declaration

```
mat matrixCompMult (x, y);

mat x;
mat y;
```

Parameters

x Specifies the first matrix multiplicand.

y Specifies the second matrix multiplicand.

Description

`matrixCompMult` performs a component-wise multiplication of two matrices, yielding a result matrix where each component, `result[i][j]` is computed as the scalar product of *x*`[i][j]` and *y*`[i][j]`.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
matrixCompMult (genType)	✔	✔

See Also

dot, reflect

Copyright

Name

max — return the greater of two values

Declaration

```
genType max (x, y);

genType x;
genType y;

genType max (x, y);

genType x;
float y;

genIType max (x, y);

genIType x;
genIType y;

genIType max (x, y);

genIType x;
int y;

genUType max (x, y);

genUType x;
genUType y;

genUType max (x, y);

genUType x;
uint y;
```

Parameters

x	Specify the first value to compare.
y	Specify the second value to compare.

Description

`max` returns the maximum of the two parameters. It returns *y* if *y* is greater than *x*, otherwise it returns *x*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
max (genType)	✔	✔
max (genIType)	-	✔
max (genUType)	-	✔

See Also

min, abs

Copyright

Name

min — return the lesser of two values

Declaration

```
genType min (x, y);

genType x;
genType y;

genType min (x, y);

genType x;
float y;

genIType min (x, y);

genIType x;
genIType y;

genIType min (x, y);

genIType x;
int y;

genUType min (x, y);

genUType x;
genUType y;

genUType min (x, y);

genUType x;
uint y;
```

Parameters

x	Specify the first value to compare.
y	Specify the second value to compare.

Description

`min` returns the minimum of the two parameters. It returns *y* if *y* is less than *x*, otherwise it returns *x*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
min (genType)	✔	✔
min (genIType)	-	✔
min (genUType)	-	✔

See Also

max, abs

Copyright

Name

mix — linearly interpolate between two values

Declaration

```
genType mix (x, y, a);

genType x;
genType y;
genType a;

genType mix (x, y, a);

genType x;
genType y;
float a;

genType mix (x, y, a);

genType x;
genType y;
genBType a;
```

Parameters

x	Specify the start of the range in which to interpolate.
y	Specify the end of the range in which to interpolate.
a	Specify the value to use to interpolate between *x* and *y*.

Description

`mix` performs a linear interpolation between *x* and *y* using *a* to weight between them. The return value is computed as follows: .

For the variants of `mix` where *a* is `genBType`, elements for which *a*[*i*] is `false`, the result for that element is taken from *x*, and where *a*[*i*] is `true`, it will be taken from *y*. Components of *x* and *y* that are not selected are allowed to be invalid floating point values and will have no effect on the results.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
mix (genType)	✔	✔
mix (genBType)	-	✔

See Also

min, max

Copyright

Name

mod — compute value of one parameter modulo another

Declaration

```
genType mod (x, y);

genType x;
float y;

genType mod (x, y);

genType x;
genType y;
```

Parameters

x Specify the value to evaluate.

y Specify the value by which to perform the modulo.

Description

`mod` returns the value of *x* modulo *y*. This is computed as *x* - *y* * floor(*x*/*y*).

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
mod (genType)	✔	✔

See Also

modf, floor

Copyright

Name

modf — separate a value into its integer and fractional components

Declaration

```
genType modf (x, i);

genType x;
out genType i;
```

Parameters

x	Specify the value to separate.
out i	A variable that receives the integer part of the argument.

Description

`modf` separates a floating point value *x* into its integer and fractional parts. The fractional part of the number is returned from the function and the integer part (as a floating point quantity) is returned in the output parameter *i*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
modf (genType)	-	✔

See Also

fract, floor

Copyright

Name

normalize — calculates the unit vector in the same direction as the original vector

Declaration

```
genType normalize (v);

genType v;
```

Parameters

v Specifies the vector to normalize.

Description

`normalize` returns a vector with the same direction as its parameter, *v*, but with length 1.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
normalize (genType)	✔	✔

See Also

length

Copyright

Name

not — logically invert a boolean vector

Declaration

```
bvec not (x);

bvec x;
```

Parameters

x Specifies the vector to be inverted.

Description

`not` logically inverts the boolean vector *x*. It returns a new boolean vector for which each element *i* is computed as `!x[i]`.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
not	✔	✔

See Also

any, all

Copyright

Name

notEqual — perform a component-wise not-equal-to comparison of two vectors

Declaration

```
bvec **notEqual** (*x*, *y*);

vec *x*;
vec *y*;

bvec **notEqual** (*x*, *y*);

ivec *x*;
ivec *y*;

bvec **notEqual** (*x*, *y*);

bvec *x*;
bvec *y*;

bvec **notEqual** (*x*, *y*);

uvec *x*;
uvec *y*;
```

Parameters

x	Specifies the first vector to be used in the comparison operation.
y	Specifies the second vector to be used in the comparison operation.

Description

`notEqual` returns a boolean vector in which each element *i* is computed as *x*[*i*] != *y*[*i*].

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
notEqual (vec)	✔	✔
notEqual (ivec)	✔	✔
notEqual (bvec)	✔	✔
notEqual (uvec)	-	✔

See Also

lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, any, all, not

Copyright

Name

outerProduct — calculate the outer product of a pair of vectors

Declaration

```
mat2 outerProduct (c, r);

vec2 c;
vec2 r;

mat3 outerProduct (c, r);

vec3 c;
vec3 r;

mat4 outerProduct (c, r);

vec4 c;
vec4 r;

mat2x3 outerProduct (c, r);

vec3 c;
vec2 r;

mat3x2 outerProduct (c, r);

vec2 c;
vec3 r;

mat2x4 outerProduct (c, r);

vec4 c;
vec2 r;

mat4x2 outerProduct (c, r);

vec2 c;
vec4 r;

mat3x4 outerProduct (c, r);

vec4 c;
vec3 r;

mat4x3 outerProduct (c, r);

vec3 c;
vec4 r;
```

Parameters

c Specifies the parameter to be treated as a column vector.

r Specifies the parameter to be treated as a row vector.

Description

`outerProduct` treats the first parameter *`c`* as a column vector (matrix with one column) and the second parameter *`r`* as a row vector (matrix with one row) and does a linear algebraic matrix multiply *`c`* `*` *`r`*, yielding a matrix whose number of rows is the number of components in *`c`* and whose number of columns is the number of components in *`r`*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
outerProduct (float)	-	✔

See Also

dot

Copyright

Name

packHalf2x16 — convert two 32-bit floating-point quantities to 16-bit quantities and pack them into a single 32-bit integer

Declaration

```
uint packHalf2x16 (v);

vec2 v;
```

Parameters

v Specify a vector of two 32-bit floating point values that are to be converted to 16-bit representation and packed into the result.

Description

`packHalf2x16` returns an unsigned integer obtained by converting the components of a two-component floating-point vector to the 16-bit floating-point representation found in the OpenGL ES Specification, and then packing these two 16-bit integers into a 32-bit unsigned integer. The first vector component specifies the 16 least-significant bits of the result; the second component specifies the 16 most-significant bits.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
packHalf2x16	-	✔

See Also

unpackHalf2x16

Copyright

Name

packUnorm2x16, packSnorm2x16 — pack floating-point values into an unsigned integer

Declaration

```
uint packUnorm2x16 (v);
vec2 v;
uint packSnorm2x16 (v);
vec2 v;
```

Parameters

v Specifies a vector of values to be packed into an unsigned integer.

Description

`packUnorm2x16` and `packSnorm2x16` converts each component of the normalized floating-point value *v* into 16-bit integer values and then packs the results into a 32-bit unsigned intgeter.

The conversion for component *c* of *v* to fixed-point is performed as follows:

- packUnorm2x16: `round(clamp(c, 0.0, 1.0) * 65535.0)`
- packSnorm2x16: `round(clamp(c, -1.0, 1.0) * 32767.0)`

The first component of the vector will be written to the least significant bits of the output; the last component will be written to the most significant bits.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
packUnorm2x16	-	✔
packSnorm2x16	-	✔

See Also

clamp, round, unpackUnorm2x16, unpackSnorm2x16,

Copyright

Name

pow — return the value of the first parameter raised to the power of the second

Declaration

```
genType pow (x, y);

genType x;
genType y;
```

Parameters

x Specify the value to raise to the power *y*.

y Specify the power to which to raise *x*.

Description

`pow` returns the value of *x* raised to the *y* power. i.e., . Results are undefined if $x < 0$ or if $x == 0$ and $y \leq 0$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
pow	✔	✔

See Also

exp

Copyright

Name

radians — convert a quantity in degrees to radians

Declaration

```
genType radians (degrees);
```

```
genType degrees;
```

Parameters

degrees	Specify the quantity, in degrees, to be converted to radians.

Description

`radians` converts a quantity, specified in degrees into radians. That is, the return value is .

Version Support

Function Name	**OpenGL ES Shading Language Version**	
	1.00	**3.00**
radians	✔	✔

See Also

degrees,

Copyright

Name

reflect — calculate the reflection direction for an incident vector

Declaration

```
genType reflect (I, N);

genType I;
genType N;
```

Parameters

`I` Specifies the incident vector.

`N` Specifies the normal vector.

Description

For a given incident vector `I` and surface normal `N` `reflect` returns the reflection direction calculated as `I - 2.0 * dot(N, I) * N`.

`N` should be normalized in order to achieve the desired result.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
reflect (genType)	✔	✔

See Also

dot, refract

Copyright

Name

refract — calculate the refraction direction for an incident vector

Declaration

```
genType refract (I, N, eta);

genType I;
genType N;
float eta;
```

Parameters

I	Specifies the incident vector.
N	Specifies the normal vector.
eta	Specifies the ratio of indices of refraction.

Description

For a given incident vector *I*, surface normal *N* and ratio of indices of refraction, *eta*, `refract` returns the refraction vector, *R*.

R is calculated as:

```
    k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I));
    if (k < 0.0)
        R = genType(0.0);       // or genDType(0.0)
    else
        R = eta * I - (eta * dot(N, I) + sqrt(k)) * N;
```

The input parameters *I* and *N* should be normalized in order to achieve the desired result.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
refract (genType)	✔	✔

See Also

dot, reflect

Copyright

Name

round — find the nearest integer to the parameter

Declaration

```
genType round (x);
```

```
genType x;
```

Parameters

x Specify the value to evaluate.

Description

`round` returns a value equal to the nearest integer to *x*. The fraction 0.5 will round in a direction chosen by the implementation, usually in the direction that is fastest. This includes the possibility that `round`(*x*) returns the same value as roundEven(*x*) for all values of *x*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
round (genType)	-	✔

See Also

floor, roundEven

Copyright

Name

roundEven — find the nearest even integer to the parameter

Declaration

```
genType roundEven (x);

genType x;
```

Parameters

x Specify the value to evaluate.

Description

`roundEven` returns a value equal to the nearest integer to *x*. The fractional part of 0.5 will round toward the nearest even integer. For example, both 3.5 and 4.5 will round to 4.0.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
roundEven (genType)	-	✔

See Also

floor, round

Copyright

Name

sign — extract the sign of the parameter

Declaration

```
genType sign (x);
genType x;
genIType sign (x);
genIType x;
```

Parameters

x Specify the value from which to extract the sign.

Description

`sign` returns -1.0 if *x* is less than 0.0, 0.0 if *x* is equal to 0.0, and +1.0 if *x* is greater than 0.0.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
sign (genType)	✔	✔
sign (genIType)	-	✔

See Also

abs

Copyright

Name

sin — return the sine of the parameter

Declaration

```
genType sin (angle);
```

```
genType angle;
```

Parameters

`angle` Specify the quantity, in radians, of which to return the sine.

Description

`sin` returns the trigonometric sine of *`angle`*.

Version Support

Function Name	**OpenGL ES Shading Language Version**	
	1.00	**3.00**
sin	✔	✔

See Also

cos, tan, asin, acos, atan

Copyright

Name

sinh — return the hyperbolic sine of the parameter

Declaration

```
genType sinh (x);

genType x;
```

Parameters

x Specify the value whose hyperbolic sine to return.

Description

`sinh` returns the hyperbolic sine of . The hyperbolic sine of is computed as .

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
sinh	-	✓

See Also

sin, cos, cosh

Copyright

Name

smoothstep — perform Hermite interpolation between two values

Declaration

```
genType smoothstep (edge0, edge1, x);

genType edge0;
genType edge1;
genType x;

genType smoothstep (edge0, edge1, x);

float edge0;
float edge1;
genType x;
```

Parameters

edge0	Specifies the value of the lower edge of the Hermite function.
edge1	Specifies the value of the upper edge of the Hermite function.
x	Specifies the source value for interpolation.

Description

`smoothstep` performs smooth Hermite interpolation between 0 and 1 when *edge0* < *x* < *edge1*. This is useful in cases where a threshold function with a smooth transition is desired. `smoothstep` is equivalent to:

```
    genType t;  /* Or genDType t; */
    t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
    return t * t * (3.0 - 2.0 * t);
```

`smoothstep` returns 0.0 if $x \leq edge0$ and 1.0 if $x \geq edge1$.

Results are undefined if $edge0 \geq edge1$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
smoothstep (genType)	✔	✔

See Also

mix, step

Copyright

Name

sqrt — return the square root of the parameter

Declaration

```
genType sqrt (x);
```

```
genType x;
```

Parameters

x Specify the value of which to take the square root.

Description

`sqrt` returns the square root of x, i.e. the value $\sqrt { x }$. The result is undefined if $x < 0$.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
sqrt (genType)	✔	✔

See Also

inversesqrt

Copyright

Name

step — generate a step function by comparing two values

Declaration

```
genType step (edge, x);

genType edge;
genType x;

genType step (edge, x);

float edge;
genType x;
```

Parameters

edge	Specifies the location of the edge of the step function.
x	Specify the value to be used to generate the step function.

Description

`step` generates a step function by comparing *x* to *edge*.

For element *i* of the return value, 0.0 is returned if *x*[*i*] < *edge*[*i*], and 1.0 is returned otherwise.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
step (genType)	✔	✔

See Also

mix, smoothstep

Copyright

Name

tan — return the tangent of the parameter

Declaration

```
genType tan (angle);

genType angle;
```

Parameters

`angle`	Specify the quantity, in radians, of which to return the tangent.

Description

`tan` returns the trigonometric tangent of *`angle`*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
tan	✔	✔

See Also

sin, cos, asin, acos, atan

Copyright

Name

tanh — return the hyperbolic tangent of the parameter

Declaration

```
genType tanh (x);
genType x;
```

Parameters

x Specify the value whose hyperbolic tangent to return.

Description

`tanh` returns the hyperbolic tangent of . The hyperbolic tangent of is computed as .

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
tanh	-	✔

See Also

sin, cos, sinh, cosh

Copyright

Name

texelFetch — perform a lookup of a single texel within a texture

Declaration

```
gvec4 texelFetch (sampler, P, lod);

gsampler2D sampler;
ivec2 P;
int lod;

gvec4 texelFetch (sampler, P, lod);

gsampler3D sampler;
ivec3 P;
int lod;

gvec4 texelFetch (sampler, P, lod);

gsampler2DArray sampler;
ivec3 P;
int lod;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
lod	If present, specifies the level-of-detail within the texture from which the texel will be fetched.

Description

`texelFetch` performs a lookup of a single texel from texture coordinate *P* in the texture bound to *sampler*. The array layer is specified in the last component of *P* for array forms. The *lod* parameter (if present) specifies the level-of-detail from which the texel will be fetched.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
texelFetch	-	✔

See Also

texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

texelFetchOffset — perform a lookup of a single texel within a texture with an offset

Declaration

```
gvec4 texelFetchOffset (sampler, P, lod, offset);

gsampler2D sampler;
ivec2 P;
int lod;
ivec2 offset;

gvec4 texelFetchOffset (sampler, P, lod, offset);

gsampler3D sampler;
ivec3 P;
int lod;
ivec3 offset;

gvec4 texelFetchOffset (sampler, P, lod, offset);

gsampler2DArray sampler;
ivec3 P;
int lod;
ivec2 offset;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
lod	If present, specifies the level-of-detail within the texture from which the texel will be fetched.
offset	Specifies offset, in texels that will be applied to *P* before looking up the texel.

Description

`texelFetchOffset` performs a lookup of a single texel from texture coordinate *P* in the texture bound to *sampler*. Before fetching the texel, the offset specified in *offset* is added to *P*. *offset* must be a constant expression. The array layer is specified in the last component of *P* for array forms. The *lod* parameter (if present) specifies the level-of-detail from which the texel will be fetched. The *sample* parameter specifies which sample within the texel will be returned when reading from a multi-sample texure.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
texelFetchOffset	-	✔

See Also

texelFetch, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

texture — retrieves texels from a texture

Declaration

```
gvec4 texture (sampler, P, bias);

gsampler2D sampler;
vec2 P;
[float bias];

gvec4 texture (sampler, P, bias);

gsampler3D sampler;
vec3 P;
[float bias];

gvec4 texture (sampler, P, bias);

gsamplerCube sampler;
vec3 P;
[float bias];

float texture (sampler, P, bias);

sampler2DShadow sampler;
vec3 P;
[float bias];

float texture (sampler, P, bias);

samplerCubeShadow sampler;
vec4 P;
[float bias];

gvec4 texture (sampler, P, bias);

gsampler2DArray sampler;
vec3 P;
[float bias];

float texture (sampler, P);

sampler2DArrayShadow sampler;
vec4 P;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
bias	Specifies an optional bias to be applied during level-of-detail computation.

Description

`texture` samples texels from the texture bound to *`sampler`* at texture coordinate *`P`*. An optional bias, specified in *`bias`* is included in the level-of-detail computation that is used to choose mipmap(s) from which to sample.

For *shadow* forms, when *`compare`* is present, it is used as and the array layer is specified in *`P.w`*. When *`compare`* is not present, the last component of *`P`* is used as and the array layer is specified in the second to last component of *`P`*.

For non-shadow variants, the array layer comes from the last component of *`P`*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
texture	-	✔

See Also

texelFetch, texelFetchOffset, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureGrad — perform a texture lookup with explicit gradients

Declaration

```
gvec4 textureGrad (sampler, P, dPdx, dPdy);

gsampler2D sampler;
vec2 P;
vec2 dPdx;
vec2 dPdy;

gvec4 textureGrad (sampler, P, dPdx, dPdy);

gsampler3D sampler;
vec3 P;
vec3 dPdx;
vec3 dPdy;

gvec4 textureGrad (sampler, P, dPdx, dPdy);

gsamplerCube sampler;
vec3 P;
vec3 dPdx;
vec3 dPdy;

float textureGrad (sampler, P, dPdx, dPdy);

sampler2DShadow sampler;
vec3 P;
vec2 dPdx;
vec2 dPdy;

float textureGrad (sampler, P, dPdx, dPdy);

samplerCubeShadow sampler;
vec4 P;
vec3 dPdx;
vec3 dPdy;

gvec4 textureGrad (sampler, P, dPdx, dPdy);

gsampler2DArray sampler;
vec3 P;
vec2 dPdx;
vec2 dPdy;

float textureGrad (sampler, P, dPdx, dPdy);

sampler2DArrayShadow sampler;
vec4 P;
vec2 dPdx;
vec2 dPdy;
```

Parameters

`sampler`	Specifies the sampler to which the texture from which texels will be retrieved is bound.
`P`	Specifies the texture coordinates at which texture will be sampled.
`dPdx`	Specifies the partial derivative of *`P`* with respect to window x.
`dPdy`	Specifies the partial derivative of *`P`* with respect to window y.

Description

`textureGrad` performs a texture lookup at coordinate *`P`* from the texture bound to *`sampler`* with explicit texture coordinate gradiends as specified in *`dPdx`* and *`dPdy`*. Set:

For the cube version, the partial derivatives of *`P`* are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureGrad	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureGradOffset — perform a texture lookup with explicit gradients and offset

Declaration

```
gvec4 textureGradOffset (sampler, P, dPdx, dPdy, offset);

gsampler2D sampler;
vec2 P;
vec2 dPdx;
vec2 dPdy;
ivec2 offset;

gvec4 textureGradOffset (sampler, P, dPdx, dPdy, offset);

gsampler3D sampler;
vec3 P;
vec3 dPdx;
vec3 dPdy;
ivec3 offset;

float textureGradOffset (sampler, P, dPdx, dPdy, offset);

sampler2DShadow sampler;
vec3 P;
vec2 dPdx;
vec2 dPdy;
ivec2 offset;

gvec4 textureGradOffset (sampler, P, dPdx, dPdy, offset);

gsampler2DArray sampler;
vec3 P;
vec2 dPdx;
vec2 dPdy;
ivec2 offset;

float textureGradOffset (sampler, P, dPdx, dPdy, offset);

sampler2DArrayShadow sampler;
vec4 P;
vec2 dPdx;
vec2 dPdy;
ivec2 offset;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
dPdx	Specifies the partial derivative of *P* with respect to window x.
dPdy	Specifies the partial derivative of *P* with respect to window y.

offset Specifies the offset to be applied to the texture coordinates before sampling.

Description

`textureGradOffset` performs a texture lookup at coordinate *P* from the texture bound to *sampler* with explicit texture coordinate gradiends as specified in *dPdx* and *dPdy*. An explicit offset is also supplied in *offset*. `textureGradOffset` consumes *dPdx* and *dPdy* as textureGrad and *offset* as textureOffset.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureGradOffset	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureLod — perform a texture lookup with explicit level-of-detail

Declaration

```
gvec4 textureLod (sampler, P, lod);

gsampler2D sampler;
vec2 P;
float lod;

gvec4 textureLod (sampler, P, lod);

gsampler3D sampler;
vec3 P;
float lod;

gvec4 textureLod (sampler, P, lod);

gsamplerCube sampler;
vec3 P;
float lod;

float textureLod (sampler, P, lod);

sampler2DShadow sampler;
vec3 P;
float lod;

gvec4 textureLod (sampler, P, lod);

gsampler2DArray sampler;
vec3 P;
float lod;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
lod	Specifies the explicit level-of-detail

Description

`textureLod` performs a texture lookup at coordinate *`P`* from the texture bound to *`sampler`* with an explicit level-of-detail as specified in *`lod`*. *`lod`* specifies and sets the partial derivatives as follows:

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureLod	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureLodOffset — perform a texture lookup with explicit level-of-detail and offset

Declaration

```
gvec4 textureLodOffset (sampler, P, lod, offset);

gsampler2D sampler;
vec2 P;
float lod;
ivec2 offset;

gvec4 textureLodOffset (sampler, P, lod, offset);

gsampler3D sampler;
vec3 P;
float lod;
ivec3 offset;

float textureLodOffset (sampler, P, lod, offset);

sampler2DShadow sampler;
vec3 P;
float lod;
ivec2 offset;

gvec4 textureLodOffset (sampler, P, lod, offset);

gsampler2DArray sampler;
vec3 P;
float lod;
ivec2 offset;
```

Parameters

`sampler`	Specifies the sampler to which the texture from which texels will be retrieved is bound.
`P`	Specifies the texture coordinates at which the texture will be sampled.
`lod`	Specifies the explicit level-of-detail from which texels will be fetched.
`offset`	Specifies the offset that will be applied to *P* before texels are fetched.

Description

`textureLodOffset` performs a texture lookup at coordinate *P* from the texture bound to *sampler* with an explicit level-of-detail as specified in *lod*. Behavior is the same as in textureLod except that before sampling, *offset* is added to *P*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureLodOffset	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureOffset — perform a texture lookup with offset

Declaration

```
gvec4 textureOffset (sampler, P, offset, bias);

gsampler2D sampler;
vec2 P;
ivec2 offset;
[float bias];

gvec4 textureOffset (sampler, P, offset, bias);

gsampler3D sampler;
vec3 P;
ivec3 offset;
[float bias];

float textureOffset (sampler, P, offset, bias);

sampler2DShadow sampler;
vec3 P;
ivec2 offset;
[float bias];

gvec4 textureOffset (sampler, P, offset, bias);

gsampler2DArray sampler;
vec3 P;
ivec2 offset;
[float bias];
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
offset	Specifies offset, in texels that will be applied to *P* before looking up the texel.
bias	Specifies an optional bias to be applied during level-of-detail computation.

Description

`textureOffset` performs a texture lookup at coordinate *P* from the texture bound to *sampler* with an additional offset, specified in texels in *offset* that will be applied to the (u, v, w) texture coordinates before looking up each texel. The offset value must be a constant expression. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and may be determined by querying `GL_MIN_PROGRAM_TEXEL_OFFSET` and `GL_MAX_PROGRAM_TEXEL_OFFSET`, respectively.

Note that *offset* does not apply to the layer coordinate for texture arrays. Also note that offsets are not supported for cube maps.

Version Support

Function Name	**OpenGL ES Shading Language Version**	
	1.00	**3.00**
textureOffset	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureProj — perform a texture lookup with projection

Declaration

```
gvec4 textureProj (sampler, P, bias);

gsampler2D sampler;
vec3 P;
[float bias];

gvec4 textureProj (sampler, P, bias);

gsampler2D sampler;
vec4 P;
[float bias];

gvec4 textureProj (sampler, P, bias);

gsampler3D sampler;
vec4 P;
[float bias];

float textureProj (sampler, P, bias);

sampler2DShadow sampler;
vec4 P;
[float bias];
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
bias	Specifies an optional bias to be applied during level-of-detail computation.

Description

`textureProj` performs a texture lookup with projection. The texture coordinates consumed from *P*, not including the last component of *P*, are divided by the last component of *P*. The resulting component of *P* in the shadow forms is used as . After these values are computed, the texture lookup proceeds as in texture.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureProjOffset	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureProjGrad — perform a texture lookup with projection and explicit gradients

Declaration

```
gvec4 textureProjGrad (sampler, P, dPdx, dPdy);

gsampler2D sampler;
vec3 P;
vec2 dPdx;
vec2 dPdy;

gvec4 textureProjGrad (sampler, P, dPdx, dPdy);

gsampler2D sampler;
vec4 P;
vec2 dPdx;
vec2 dPdy;

gvec4 textureProjGrad (sampler, P, dPdx, dPdy);

gsampler3D sampler;
vec4 P;
vec3 dPdx;
vec3 dPdy;

float textureProjGrad (sampler, P, dPdx, dPdy);

sampler2DShadow sampler;
vec4 P;
vec2 dPdx;
vec2 dPdy;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
dPdx	Specifies the partial derivative of *P* with respect to window x.
dPdy	Specifies the partial derivative of *P* with respect to window y.

Description

`textureProjGrad` performs a texture lookup with projection and explicit gradients. The texture coordinates consumed from *P*, not including the last component of *P*, are divided by the last component of *P*. The resulting component of *P* in the shadow forms is used as . After these values are computed, the texture lookup proceeds as in textureGrad, passing *dPdx* and *dPdy* as gradients.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureProjGrad	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureProjGradOffset — perform a texture lookup with projection, explicit gradients and offset

Declaration

```
gvec4 textureProjGradOffset (sampler, P, dPdx, dPdy, offset);

gsampler2D sampler;
vec3 P;
vec2 dPdx;
vec2 dPdy;
ivec2 offset;

gvec4 textureProjGradOffset (sampler, P, dPdx, dPdy, offset);

gsampler2D sampler;
vec4 P;
vec2 dPdx;
vec2 dPdy;
ivec2 offset;

gvec4 textureProjGradOffset (sampler, P, dPdx, dPdy, offset);

gsampler3D sampler;
vec4 P;
vec3 dPdx;
vec3 dPdy;
ivec3 offset;

float textureProjGradOffset (sampler, P, dPdx, dPdy, offset);

sampler2DShadow sampler;
vec4 P;
vec2 dPdx;
vec2 dPdy;
ivec2 offset;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
dPdx	Specifies the partial derivative of *P* with respect to window x.
dPdy	Specifies the partial derivative of *P* with respect to window y.
offset	Specifies the offsets, in texels at which the texture will be sampled relative to the projection of *P*.

Description

`textureProjGradOffset` performs a texture lookup with projection and explicit gradients and offsets. The texture coordinates consumed from *P*, not including the last component of *P*, are divided by the last component of *P*. The resulting component of *P* in the shadow forms is used as . After these values are computed, the texture lookup proceeds as in textureGradOffset, passing *dPdx* and *dPdy* as gradients, and *offset* as the offset.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureProjGradOffset	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjLod, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureProjLod — perform a texture lookup with projection and explicit level-of-detail

Declaration

```
gvec4 textureProjLod (sampler, P, lod);

gsampler2D sampler;
vec3 P;
float lod;

gvec4 textureProjLod (sampler, P, lod);

gsampler2D sampler;
vec4 P;
float lod;

gvec4 textureProjLod (sampler, P, lod);

gsampler3D sampler;
vec4 P;
float lod;

float textureProjLod (sampler, P, lod);

sampler2DShadow sampler;
vec4 P;
float lod;
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which texture will be sampled.
lod	Specifies the explicit level-of-detail from which to fetch texels.

Description

`textureProjLod` performs a texture lookup with projection from an explicitly specified level-of-detail. The texture coordinates consumed from *P*, not including the last component of *P*, are divided by the last component of *P*. The resulting component of *P* in the shadow forms is used as . After these values are computed, the texture lookup proceeds as in textureLod, with *lod* used to specify the level-of-detail from which the texture will be sampled.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureProjLod	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLodOffset, textureProjOffset, textureSize

Copyright

Name

textureProjLodOffset — perform a texture lookup with projection and explicit level-of-detail and offset

Declaration

```
gvec4 textureProjLodOffset (sampler, P, lod, offset);

gsampler2D sampler;
vec3 P;
float lod;
ivec2 offset;

gvec4 textureProjLodOffset (sampler, P, lod, offset);

gsampler2D sampler;
vec4 P;
float lod;
ivec2 offset;

gvec4 textureProjLodOffset (sampler, P, lod, offset);

gsampler3D sampler;
vec4 P;
float lod;
ivec3 offset;

float textureProjLodOffset (sampler, P, lod, offset);

sampler2DShadow sampler;
vec4 P;
float lod;
ivec2 offset;
```

Parameters

`sampler`	Specifies the sampler to which the texture from which texels will be retrieved is bound.
`P`	Specifies the texture coordinates at which texture will be sampled.
`lod`	Specifies the explicit level-of-detail from which to fetch texels.
`offset`	Specifies the offset, in texels, to be applied to *`P`* before fetching texels.

Description

`textureProjLodOffset` performs a texture lookup with projection from an explicitly specified level-of-detail with an offset applied to the texture coordinates before sampling. The texture coordinates consumed from *`P`*, not including the last component of *`P`*, are divided by the last component of *`P`*. The resulting component of *`P`* in the shadow forms is used as . After these values are computed, the texture lookup proceeds as in textureLodOffset, with *`lod`* used to specify the level-of-detail from which the texture will be sampled and *`offset`* used to specifiy the offset, in texels, to be applied to the texture coordinates before sampling.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
textureProjLodOffset	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjOffset, textureSize

Copyright

Name

textureProjOffset — perform a texture lookup with projection and offset

Declaration

```
gvec4 textureProjOffset (sampler, P, offset, bias);

gsampler2D sampler;
vec3 P;
ivec2 offset;
[float bias];

gvec4 textureProjOffset (sampler, P, offset, bias);

gsampler2D sampler;
vec4 P;
ivec2 offset;
[float bias];

gvec4 textureProjOffset (sampler, P, offset, bias);

gsampler3D sampler;
vec4 P;
ivec3 offset;
[float bias];

float textureProjOffset (sampler, P, offset, bias);

sampler2DShadow sampler;
vec4 P;
ivec2 offset;
[float bias];
```

Parameters

sampler	Specifies the sampler to which the texture from which texels will be retrieved is bound.
P	Specifies the texture coordinates at which the texture will be sampled.
offset	Specifies the offset that is applied to *P* before sampling occurs.
bias	Specifies an optional bias to be applied during level-of-detail computation.

Description

`textureProjOffset` performs a texture lookup with projection. The texture coordinates consumed from *P*, not including the last component of *P*, are divided by the last component of *P*. The resulting component of *P* in the shadow forms is used as . After these values are computed, the texture lookup proceeds as in textureOffset, with the *offset* used to offset the computed texture coordinates.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
textureProjOffset	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureSize

Copyright

Name

textureSize — retrieve the dimensions of a level of a texture

Declaration

```
ivec2 textureSize (sampler, lod);

gsampler2D sampler;
int lod;

ivec3 textureSize (sampler, lod);

gsampler3D sampler;
int lod;

ivec2 textureSize (sampler, lod);

gsamplerCube sampler;
int lod;

ivec2 textureSize (sampler, lod);

sampler2DShadow sampler;
int lod;

ivec2 textureSize (sampler, lod);

samplerCubeShadow sampler;
int lod;

ivec3 textureSize (sampler, lod);

gsampler2DArray sampler;
int lod;

ivec3 textureSize (sampler, lod);

sampler2DArrayShadow sampler;
int lod;
```

Parameters

sampler	Specifies the sampler to which the texture whose dimensions to retrieve is bound.
lod	Specifies the level of the texture for which to retrieve the dimensions.

Description

`textureSize` returns the dimensions of level *lod* (if present) of the texture bound to *sampler*. The components in the return value are filled in, in order, with the width, height and depth of the texture. For the array forms, the last component of the return value is the number of layers in the texture array. The return values are returned as highp ints.

Version Support

Function Name	OpenGL ES Shading Language Version	
	1.00	**3.00**
textureSize	-	✔

See Also

texelFetch, texelFetchOffset, texture, textureGrad, textureGradOffset, textureLod, textureLodOffset, textureOffset, textureProj, textureProjGrad, textureProjGradOffset, textureProjLod, textureProjLodOffset, textureProjOffset,

Copyright

Name

transpose — calculate the transpose of a matrix

Declaration

```
mat2 transpose (m);
mat2 m;
mat3 transpose (m);
mat3 m;
mat4 transpose (m);
mat4 m;
mat2x3 transpose (m);
mat3x2 m;
mat2x4 transpose (m);
mat4x2 m;
mat3x2 transpose (m);
mat2x3 m;
mat3x4 transpose (m);
mat4x3 m;
mat4x2 transpose (m);
mat2x4 m;
mat4x3 transpose (m);
mat3x4 m;
```

Parameters

m Specifies the matrix of which to take the transpose.

Description

`transpose` returns the transpose of the matrix *m*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
transpose (float)	-	✔

See Also

determinant, inverse

Copyright

Name

trunc — find the truncated value of the parameter

Declaration

```
genType trunc (x);

genType x;
```

Parameters

x Specify the value to evaluate.

Description

`trunc` returns a value equal to the nearest integer to *x* whose absolute value is not larger than the absolute value of *x*.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
trunc (genType)	-	✔

See Also

floor, round

Copyright

Name

unpackHalf2x16 — convert two 16-bit floating-point values packed into a single 32-bit integer into a vector of two 32-bit floating-point quantities

Declaration

```
vec2 unpackHalf2x16 (v);

uint v;
```

Parameters

v Specify a single 32-bit unsigned integer values that contains two 16-bit floating point values to be unpacked.

Description

`unpackHalf2x16` returns a two-component floating-point vector with components obtained by unpacking a 32-bit unsigned integer into a pair of 16-bit values, interpreting those values as 16-bit floating-point numbers according to the OpenGL ES Specification, and converting them to 32-bit floating-point values. The first component of the vector is obtained from the 16 least-significant bits of v; the second component is obtained from the 16 most-significant bits of v.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
unpackHalf2x16	-	✔

See Also

packHalf2x16

Copyright

Name

unpackUnorm2x16, unpackSnorm2x16 — unpack floating-point values from an unsigned integer

Declaration

```
vec2 unpackUnorm2x16 (p);

uint p;

vec2 unpackSnorm2x16 (p);

uint p;
```

Parameters

p Specifies an unsigned integer containing packed floating-point values.

Description

`unpackUnorm2x16`, `unpackSnorm2x16` unpack single 32-bit unsigned integers, specified in the parameter *p* into a pair of 16-bit unsigned integers. Then, each component is converted to a normalized floating-point value to generate the returned two- or four-component vector.

The conversion for unpacked fixed point value *f* to floating-point is performed as follows:

- `unpackUnorm2x16`: `f / 65535.0`
- `unpackSnorm2x16`: `clamp(f / 32727.0, -1.0, 1.0)`

The first component of the returned vector will be extracted from the least significant bits of the input; the last component will be extracted from the most significant bits.

Version Support

	OpenGL ES Shading Language Version	
Function Name	**1.00**	**3.00**
unpackUnorm2x16	-	✔
unpackSnorm2x16	-	✔

See Also

clamp, packUnorm2x16, packSnorm2x16

Copyright