docs.godotengine.org

Shading language

23–29 minutes

Introduction¶

Godot uses a shading language similar to GLSL ES 3.0. Most datatypes and functions are supported, and the few remaining ones will likely be added over time.

If you are already familiar with GLSL, the Godot Shader Migration Guide is a resource that will help you transition from regular GLSL to Godot's shading language.

Data types¶

Most GLSL ES 3.0 datatypes are supported:

Type	Description
void	Void datatype, useful only for functions that return nothing.
bool	Boolean datatype, can only contain `true` or `false`.
bvec2	Two-component vector of booleans.
bvec3	Three-component vector of booleans.
bvec4	Four-component vector of booleans.
int	Signed scalar integer.
ivec2	Two-component vector of signed integers.
ivec3	Three-component vector of signed integers.
ivec4	Four-component vector of signed integers.
uint	Unsigned scalar integer; can't contain negative numbers.

Type	Description
uvec2	Two-component vector of unsigned integers.
uvec3	Three-component vector of unsigned integers.
uvec4	Four-component vector of unsigned integers.
float	Floating-point scalar.
vec2	Two-component vector of floating-point values.
vec3	Three-component vector of floating-point values.
vec4	Four-component vector of floating-point values.
mat2	2x2 matrix, in column major order.
mat3	3x3 matrix, in column major order.
mat4	4x4 matrix, in column major order.
sampler2D	Sampler type for binding 2D textures, which are read as float.
isampler2D	Sampler type for binding 2D textures, which are read as signed integer.
usampler2D	Sampler type for binding 2D textures, which are read as unsigned integer.
sampler2DArray	Sampler type for binding 2D texture arrays, which are read as float.
isampler2DArray	Sampler type for binding 2D texture arrays, which are read as signed integer.
usampler2DArray	Sampler type for binding 2D texture arrays, which are read as unsigned integer.
sampler3D	Sampler type for binding 3D textures, which are read as float.

Type	Description
isampler3D	Sampler type for binding 3D textures, which are read as signed integer.
usampler3D	Sampler type for binding 3D textures, which are read as unsigned integer.
samplerCube	Sampler type for binding Cubemaps, which are read as floats.

Casting¶

Just like GLSL ES 3.0, implicit casting between scalars and vectors of the same size but different type is not allowed. Casting of types of different size is also not allowed. Conversion must be done explicitly via constructors.

Example:

```
float a = 2; // invalid
float a = 2.0; // valid
float a = float(2); // valid
```

Default integer constants are signed, so casting is always needed to convert to unsigned:

```
int a = 2; // valid
uint a = 2; // invalid
uint a = uint(2); // valid
```

Members¶

Individual scalar members of vector types are accessed via the "x", "y", "z" and "w" members. Alternatively, using "r", "g", "b" and "a" also works and is equivalent. Use whatever fits best for your needs.

For matrices, use the `m[column][row]` indexing syntax to access each scalar, or `m[idx]` to access a vector by row index. For example, for accessing the y position of an object in a mat4 you use `m[3][1]`.

Constructing¶

Construction of vector types must always pass:

```
// The required amount of scalars
vec4 a = vec4(0.0, 1.0, 2.0, 3.0);
// Complementary vectors and/or scalars
vec4 a = vec4(vec2(0.0, 1.0), vec2(2.0, 3.0));
vec4 a = vec4(vec3(0.0, 1.0, 2.0), 3.0);
```

```
// A single scalar for the whole vector
vec4 a = vec4(0.0);
```

Construction of matrix types requires vectors of the same dimension as the matrix. You can also build a diagonal matrix using `matx(float)` syntax. Accordingly, `mat4(1.0)` is an identity matrix.

```
mat2 m2 = mat2(vec2(1.0, 0.0), vec2(0.0, 1.0));
mat3 m3 = mat3(vec3(1.0, 0.0, 0.0), vec3(0.0, 1.0, 0.0), vec3(0.0,
0.0, 1.0));
mat4 identity = mat4(1.0);
```

Matrices can also be built from a matrix of another dimension. There are two rules : If a larger matrix is constructed from a smaller matrix, the additional rows and columns are set to the values they would have in an identity matrix. If a smaller matrix is constructed from a larger matrix, the top, left submatrix of the larger matrix is used.

```
mat3 basis = mat3(WORLD_MATRIX);
mat4 m4 = mat4(basis);
mat2 m2 = mat2(m4);
```

Swizzling¶

It is possible to obtain any combination of components in any order, as long as the result is another vector type (or scalar). This is easier shown than explained:

```
vec4 a = vec4(0.0, 1.0, 2.0, 3.0);
vec3 b = a.rgb; // Creates a vec3 with vec4 components.
vec3 b = a.ggg; // Also valid; creates a vec3 and fills it with a
single vec4 component.
vec3 b = a.bgr; // "b" will be vec3(2.0, 1.0, 0.0).
vec3 b = a.xyz; // Also rgba, xyzw are equivalent.
vec3 b = a.stp; // And stpq (for texture coordinates).
float c = b.w; // Invalid, because "w" is not present in vec3 b.
vec3 c = b.xrt; // Invalid, mixing different styles is forbidden.
b.rrr = a.rgb; // Invalid, assignment with duplication.
b.bgr = a.rgb; // Valid assignment. "b"'s "blue" component will be
"a"'s "red" and vice versa.
```

Precision¶

It is possible to add precision modifiers to datatypes; use them for uniforms, variables, arguments and varyings:

```
lowp vec4 a = vec4(0.0, 1.0, 2.0, 3.0); // low precision, usually 8
bits per component mapped to 0-1
mediump vec4 a = vec4(0.0, 1.0, 2.0, 3.0); // medium precision,
```

```
usually 16 bits or half float
highp vec4 a = vec4(0.0, 1.0, 2.0, 3.0); // high precision, uses full
float or integer range (default)
```

Using lower precision for some operations can speed up the math involved (at the cost of less precision). This is rarely needed in the vertex processor function (where full precision is needed most of the time), but is often useful in the fragment processor.

Some architectures (mainly mobile) can benefit significantly from this, but there are downsides such as the additional overhead of conversion between precisions. Refer to the documentation of the target architecture for further information. In many cases, mobile drivers cause inconsistent or unexpected behavior and it is best to avoid specifying precision unless necessary.

Arrays¶

Arrays are containers for multiple variables of a similar type. Note: As of Godot 3.2, only local and varying arrays have been implemented.

Local arrays¶

Local arrays are declared in functions. They can use all of the allowed datatypes, except samplers. The array declaration follows a C-style syntax: `[const] + [precision] + typename + identifier + [array size]`.

```
void fragment() {
    float arr[3];
}
```

They can be initialized at the beginning like:

```
float float_arr[3] = float[3] (1.0, 0.5, 0.0); // first constructor

int int_arr[3] = int[] (2, 1, 0); // second constructor

vec2 vec2_arr[3] = { vec2(1.0, 1.0), vec2(0.5, 0.5), vec2(0.0, 0.0) };
// third constructor

bool bool_arr[] = { true, true, false }; // fourth constructor - size
is defined automatically from the element count
```

You can declare multiple arrays (even with different sizes) in one expression:

```
float a[3] = float[3] (1.0, 0.5, 0.0),
b[2] = { 1.0, 0.5 },
c[] = { 0.7 },
d = 0.0,
e[5];
```

To access an array element, use the indexing syntax:

```
float arr[3];

arr[0] = 1.0; // setter

COLOR.r = arr[0]; // getter
```

Arrays also have a built-in function `.length()` (not to be confused with the built-in `length()` function). It doesn't accept any parameters and will return the array's size.

```
float arr[] = { 0.0, 1.0, 0.5, -1.0 };
for (int i = 0; i < arr.length(); i++) {
    // ...
}
```

Note

If you use an index below 0 or greater than array size - the shader will crash and break rendering. To prevent this, use `length()`, `if`, or `clamp()` functions to ensure the index is between 0 and the array's length. Always carefully test and check your code. If you pass a constant expression or a simple number, the editor will check its bounds to prevent this crash.

Constants¶

Use the `const` keyword before the variable declaration to make that variable immutable, which means that it cannot be modified. All basic types, except samplers can be declared as constants. Accessing and using a constant value is slightly faster than using a uniform. Constants must be initialized at their declaration.

```
const vec2 a = vec2(0.0, 1.0);
vec2 b;

a = b; // invalid
b = a; // valid
```

Constants cannot be modified and additionally cannot have hints, but multiple of them (if they have the same type) can be declared in a single expression e.g

```
const vec2 V1 = vec2(1, 1), V2 = vec2(2, 2);
```

Similar to variables, arrays can also be declared with `const`.

```
const float arr[] = { 1.0, 0.5, 0.0 };

arr[0] = 1.0; // invalid
```

```
COLOR.r = arr[0]; // valid
```

Constants can be declared both globally (outside of any function) or locally (inside a function). Global constants are useful when you want to have access to a value throughout your shader that does not need to be modified. Like uniforms, global constants are shared between all shader stages, but they are not accessible outside of the shader.

```
shader_type spatial;

const float PI = 3.14159265358979323846;
```

Structs¶

Structs are compound types which can be used for better abstraction of shader code. You can declare them at the global scope like:

```
struct PointLight {
    vec3 position;
    vec3 color;
    float intensity;
};
```

After declaration, you can instantiate and initialize them like:

```
void fragment()
{
    PointLight light;
    light.position = vec3(0.0);
    light.color = vec3(1.0, 0.0, 0.0);
    light.intensity = 0.5;
}
```

Or use a struct constructor for the same purpose:

```
PointLight light = PointLight(vec3(0.0), vec3(1.0, 0.0, 0.0), 0.5);
```

Structs may contain other structs or arrays, you can also instance them as a global constant:

```
shader_type spatial;

...

struct Scene {
    PointLight lights[2];
};
```

```
const Scene scene = Scene(PointLight[2](PointLight(vec3(0.0, 0.0,
0.0), vec3(1.0, 0.0, 0.0), 1.0), PointLight(vec3(0.0, 0.0, 0.0),
vec3(1.0, 0.0, 0.0), 1.0)));

void fragment()
{
    ALBEDO = scene.lights[0].color;
}
```

You can also pass them to functions:

```
shader_type canvas_item;

...

Scene construct_scene(PointLight light1, PointLight light2) {
    return Scene({light1, light2});
}

void fragment()
{
    COLOR.rgb = construct_scene(PointLight(vec3(0.0, 0.0, 0.0),
vec3(1.0, 0.0, 0.0), 1.0), PointLight(vec3(0.0, 0.0, 0.0), vec3(1.0,
0.0, 1.0), 1.0)).lights[0].color;
}
```

Operators¶

Godot shading language supports the same set of operators as GLSL ES 3.0. Below is the list of them in precedence order:

Precedence	Class	Operator
1 (highest)	parenthetical grouping	**()**
2	unary	**+, -, !, ~**
3	multiplicative	**/, *, %**
4	additive	**+, -**
5	bit-wise shift	**<<, >>**
6	relational	**<, >, <=, >=**

7	equality	**==, !=**
8	bit-wise AND	**&**
9	bit-wise exclusive OR	**^**
10	bit-wise inclusive OR	**|**
11	logical AND	**&&**
12 (lowest)	logical inclusive OR	**||**

Flow control¶

Godot Shading language supports the most common types of flow control:

```
// if and else
if (cond) {

} else {

}

// switch
switch(i) { // signed integer expression
    case -1:
        break;
    case 0:
        return; // break or return
    case 1: // pass-through
    case 2:
        break;
    //...
    default: // optional
        break;
}

// for loops
for (int i = 0; i < 10; i++) {

}
```

```
// while
while (true) {

}

// do while
do {

} while(true);
```

Keep in mind that, in modern GPUs, an infinite loop can exist and can freeze your application (including editor). Godot can't protect you from this, so be careful not to make this mistake!

Warning

When exporting a GLES2 project to HTML5, WebGL 1.0 will be used. WebGL 1.0 doesn't support dynamic loops, so shaders using those won't work there.

Discarding¶

Fragment and light functions can use the `discard` keyword. If used, the fragment is discarded and nothing is written.

Beware that `discard` has a performance cost when used, as it will prevent the depth prepass from being effective on any surfaces using the shader. Also, a discarded pixel still needs to be rendered in the vertex shader, which means a shader that uses `discard` on all of its pixels is still more expensive to render compared to not rendering any object in the first place.

Functions¶

It is possible to define functions in a Godot shader. They use the following syntax:

```
ret_type func_name(args) {
    return ret_type; // if returning a value
}

// a more specific example:

int sum2(int a, int b) {
    return a + b;
}
```

You can only use functions that have been defined above (higher in the editor) the function from which you are calling them. Redefining a function that has already been defined above (or is a built-in function name) will cause an error.

Function arguments can have special qualifiers:

- **in**: Means the argument is only for reading (default).
- **out**: Means the argument is only for writing.
- **inout**: Means the argument is fully passed via reference.

Example below:

```
void sum2(int a, int b, inout int result) {
    result = a + b;
}
```

Note

Unlike GLSL, Godot's shader language does **not** support function overloading. This means that a function cannot be defined several times with different argument types or numbers of arguments. As a workaround, use different names for functions that accept a different number of arguments or arguments of different types.

Varyings¶

To send data from the vertex to the fragment (or light) processor function, *varyings* are used. They are set for every primitive vertex in the *vertex processor*, and the value is interpolated for every pixel in the *fragment processor*.

```
shader_type spatial;

varying vec3 some_color;

void vertex() {
    some_color = NORMAL; // Make the normal the color.
}

void fragment() {
    ALBEDO = some_color;
}

void light() {
    DIFFUSE_LIGHT = some_color * 100; // optionally
}
```

Varying can also be an array:

```
shader_type spatial;

varying float var_arr[3];
```

```
void vertex() {
    var_arr[0] = 1.0;
    var_arr[1] = 0.0;
}

void fragment() {
    ALBEDO = vec3(var_arr[0], var_arr[1], var_arr[2]); // red color
}
```

It's also possible to send data from *fragment* to *light* processors using *varying* keyword. To do so you can assign it in the *fragment* and later use it in the *light* function.

```
shader_type spatial;

varying vec3 some_light;

void fragment() {
    some_light = ALBEDO * 100.0; // Make a shining light.
}

void light() {
    DIFFUSE_LIGHT = some_light;
}
```

Note that varying may not be assigned in custom functions or a *light processor* function like:

```
shader_type spatial;

varying float test;

void foo() {
    test = 0.0; // Error.
}

void vertex() {
    test = 0.0;
}

void light() {
    test = 0.0; // Error too.
}
```

This limitation was introduced to prevent incorrect usage before initialization.

Interpolation qualifiers¶

Certain values are interpolated during the shading pipeline. You can modify how these interpolations are done by using *interpolation qualifiers*.

```
shader_type spatial;

varying flat vec3 our_color;

void vertex() {
    our_color = COLOR.rgb;
}

void fragment() {
    ALBEDO = our_color;
}
```

There are two possible interpolation qualifiers:

Qualifier	Description
flat	The value is not interpolated.
smooth	The value is interpolated in a perspective-correct fashion. This is the default.

Uniforms¶

Passing values to shaders is possible. These are global to the whole shader and are called *uniforms*. When a shader is later assigned to a material, the uniforms will appear as editable parameters in it. Uniforms can't be written from within the shader.

```
shader_type spatial;

uniform float some_value;
```

You can set uniforms in the editor in the material. Or you can set them through GDScript:

```
material.set_shader_param("some_value", some_value)
```

Note

The first argument to `set_shader_param` is the name of the uniform in the shader. It must match *exactly* to the name of the uniform in the shader or else it will not be recognized.

Any GLSL type except for *void* can be a uniform. Additionally, Godot provides optional shader hints to make the compiler understand for what the uniform is used, and how the editor should allow users to modify it.

```
shader_type spatial;

uniform vec4 color : hint_color;
uniform float amount : hint_range(0, 1);
uniform vec4 other_color : hint_color = vec4(1.0);
```

It's important to understand that textures that are supplied as color require hints for proper sRGB->linear conversion (i.e. `hint_albedo`), as Godot's 3D engine renders in linear color space.

Full list of hints below:

Type	Hint	Description
vec4	hint_color	Used as color.
int, float	hint_range(min, max[, step])	Restricted to values in a range (with min/max/step).
sampler2D	hint_albedo	Used as albedo color, default white.
sampler2D	hint_black_albedo	Used as albedo color, default black.
sampler2D	hint_normal	Used as normalmap.
sampler2D	hint_white	As value, default to white.
sampler2D	hint_black	As value, default to black
sampler2D	hint_aniso	As flowmap, default to right.

GDScript uses different variable types than GLSL does, so when passing variables from GDScript to shaders, Godot converts the type automatically. Below is a table of the corresponding types:

GDScript type	GLSL type
bool	**bool**
int	**int**

GDScript type	GLSL type
float	float
Vector2	vec2
Vector3	vec3
Color	vec4
Transform	mat4
Transform2D	mat4

Note

Be careful when setting shader uniforms from GDScript, no error will be thrown if the type does not match. Your shader will just exhibit undefined behavior.

Uniforms can also be assigned default values:

```
shader_type spatial;

uniform vec4 some_vector = vec4(0.0);
uniform vec4 some_color : hint_color = vec4(1.0);
```

Built-in variables¶

A large number of built-in variables are available, like `UV`, `COLOR` and `VERTEX`. What variables are available depends on the type of shader (`spatial`, `canvas_item` or `particle`) and the function used (`vertex`, `fragment` or `light`). For a list of the build-in variables that are available, please see the corresponding pages:

- Spatial shaders
- Canvas item shaders
- Particle shaders

Built-in functions¶

A large number of built-in functions are supported, conforming to GLSL ES 3.0. When vec_type (float), vec_int_type, vec_uint_type, vec_bool_type nomenclature is used, it can be scalar or vector.

Function	Description

Function	Description
vec_type **radians** (vec_type degrees)	Convert degrees to radians
vec_type **degrees** (vec_type radians)	Convert radians to degrees
vec_type **sin** (vec_type x)	Sine
vec_type **cos** (vec_type x)	Cosine
vec_type **tan** (vec_type x)	Tangent
vec_type **asin** (vec_type x)	Arcsine
vec_type **acos** (vec_type x)	Arccosine
vec_type **atan** (vec_type y_over_x)	Arctangent
vec_type **atan** (vec_type y, vec_type x)	Arctangent to convert vector to angle
vec_type **sinh** (vec_type x)	Hyperbolic sine
vec_type **cosh** (vec_type x)	Hyperbolic cosine
vec_type **tanh** (vec_type x)	Hyperbolic tangent
vec_type **asinh** (vec_type x)	Inverse hyperbolic sine
vec_type **acosh** (vec_type x)	Inverse hyperbolic cosine
vec_type **atanh** (vec_type x)	Inverse hyperbolic tangent
vec_type **pow** (vec_type x, vec_type y)	Power (undefined if x < 0 or if x = 0 and y <= 0)
vec_type **exp** (vec_type x)	Base-e exponential
vec_type **exp2** (vec_type x)	Base-2 exponential
vec_type **log** (vec_type x)	Natural logarithm

Function	Description
vec_type **log2** (vec_type x)	Base-2 logarithm
vec_type **sqrt** (vec_type x)	Square root
vec_type **inversesqrt** (vec_type x)	Inverse square root
vec_type **abs** (vec_type x)	Absolute
ivec_type **abs** (ivec_type x)	Absolute
vec_type **sign** (vec_type x)	Sign
ivec_type **sign** (ivec_type x)	Sign
vec_type **floor** (vec_type x)	Floor
vec_type **round** (vec_type x)	Round
vec_type **roundEven** (vec_type x)	Round to the nearest even number
vec_type **trunc** (vec_type x)	Truncation
vec_type **ceil** (vec_type x)	Ceil
vec_type **fract** (vec_type x)	Fractional
vec_type **mod** (vec_type x, vec_type y)	Remainder
vec_type **mod** (vec_type x , float y)	Remainder
vec_type **modf** (vec_type x, out vec_type i)	Fractional of x, with `i` as integer part
vec_type **min** (vec_type a, vec_type b)	Minimum
vec_type **max** (vec_type a, vec_type b)	Maximum
vec_type **clamp** (vec_type x, vec_type min, vec_type max)	Clamp to `min..max`

Function	Description
float **mix** (float a, float b, float c)	Linear interpolate
vec_type **mix** (vec_type a, vec_type b, float c)	Linear interpolate (scalar coefficient)
vec_type **mix** (vec_type a, vec_type b, vec_type c)	Linear interpolate (vector coefficient)
vec_type **mix** (vec_type a, vec_type b, bvec_type c)	Linear interpolate (boolean-vector selection)
vec_type **step** (vec_type a, vec_type b)	`b[i] < a[i] ? 0.0 : 1.0`
vec_type **step** (float a, vec_type b)	`b[i] < a ? 0.0 : 1.0`
vec_type **smoothstep** (vec_type a, vec_type b, vec_type c)	Hermite interpolate
vec_type **smoothstep** (float a, float b, vec_type c)	Hermite interpolate
bvec_type **isnan** (vec_type x)	Returns `true` if scalar or vector component is `NaN`
bvec_type **isinf** (vec_type x)	Returns `true` if scalar or vector component is `INF`
ivec_type **floatBitsToInt** (vec_type x)	Float->Int bit copying, no conversion
uvec_type **floatBitsToUint** (vec_type x)	Float->UInt bit copying, no conversion
vec_type **intBitsToFloat** (ivec_type x)	Int->Float bit copying, no conversion
vec_type **uintBitsToFloat** (uvec_type x)	UInt->Float bit copying, no conversion
float **length** (vec_type x)	Vector length

Function	Description
float **distance** (vec_type a, vec_type b)	Distance between vectors i.e `length(a - b)`
float **dot** (vec_type a, vec_type b)	Dot product
vec3 **cross** (vec3 a, vec3 b)	Cross product
vec_type **normalize** (vec_type x)	Normalize to unit length
vec3 **reflect** (vec3 I, vec3 N)	Reflect
vec3 **refract** (vec3 I, vec3 N, float eta)	Refract
vec_type **faceforward** (vec_type N, vec_type I, vec_type Nref)	If `dot(Nref, I)` < 0, return N, otherwise –N
mat_type **matrixCompMult** (mat_type x, mat_type y)	Matrix component multiplication
mat_type **outerProduct** (vec_type column, vec_type row)	Matrix outer product
mat_type **transpose** (mat_type m)	Transpose matrix
float **determinant** (mat_type m)	Matrix determinant
mat_type **inverse** (mat_type m)	Inverse matrix
bvec_type **lessThan** (vec_type x, vec_type y)	Bool vector comparison on < int/uint/float vectors
bvec_type **greaterThan** (vec_type x, vec_type y)	Bool vector comparison on > int/uint/float vectors
bvec_type **lessThanEqual** (vec_type x, vec_type y)	Bool vector comparison on <= int/uint/float vectors
bvec_type **greaterThanEqual** (vec_type x, vec_type y)	Bool vector comparison on >= int/uint/float vectors

Function	Description
bvec_type **equal** (vec_type x, vec_type y)	Bool vector comparison on == int/uint/float vectors
bvec_type **notEqual** (vec_type x, vec_type y)	Bool vector comparison on != int/uint/float vectors
bool **any** (bvec_type x)	Any component is `true`
bool **all** (bvec_type x)	All components are `true`
bvec_type **not** (bvec_type x)	Invert boolean vector
ivec2 **textureSize** (sampler2D_type s, int lod)	Get the size of a 2D texture
ivec3 **textureSize** (sampler2DArray_type s, int lod)	Get the size of a 2D texture array
ivec3 **textureSize** (sampler3D s, int lod)	Get the size of a 3D texture
ivec2 **textureSize** (samplerCube s, int lod)	Get the size of a cubemap texture
vec4_type **texture** (sampler2D_type s, vec2 uv [, float bias])	Perform a 2D texture read
vec4_type **texture** (sampler2DArray_type s, vec3 uv [, float bias])	Perform a 2D texture array read
vec4_type **texture** (sampler3D_type s, vec3 uv [, float bias])	Perform a 3D texture read
vec4 **texture** (samplerCube s, vec3 uv [, float bias])	Perform a cubemap texture read
vec4_type **textureProj** (sampler2D_type s, vec3 uv [, float bias])	Perform a 2D texture read with projection
vec4_type **textureProj** (sampler2D_type s, vec4 uv [, float bias])	Perform a 2D texture read with projection

Function	Description
vec4_type **textureProj** (sampler3D_type s, vec4 uv [, float bias])	Perform a 3D texture read with projection
vec4_type **textureLod** (sampler2D_type s, vec2 uv, float lod)	Perform a 2D texture read at custom mipmap
vec4_type **textureLod** (sampler2DArray_type s, vec3 uv, float lod)	Perform a 2D texture array read at custom mipmap
vec4_type **textureLod** (sampler3D_type s, vec3 uv, float lod)	Perform a 3D texture read at custom mipmap
vec4 **textureLod** (samplerCube s, vec3 uv, float lod)	Perform a 3D texture read at custom mipmap
vec4_type **textureProjLod** (sampler2D_type s, vec3 uv, float lod)	Perform a 2D texture read with projection/LOD
vec4_type **textureProjLod** (sampler2D_type s, vec4 uv, float lod)	Perform a 2D texture read with projection/LOD
vec4_type **textureProjLod** (sampler3D_type s, vec4 uv, float lod)	Perform a 3D texture read with projection/LOD
vec4_type **texelFetch** (sampler2D_type s, ivec2 uv, int lod)	Fetch a single texel using integer coordinates
vec4_type **texelFetch** (sampler2DArray_type s, ivec3 uv, int lod)	Fetch a single texel using integer coordinates
vec4_type **texelFetch** (sampler3D_type s, ivec3 uv, int lod)	Fetch a single texel using integer coordinates
vec_type **dFdx** (vec_type p)	Derivative in x using local differencing
vec_type **dFdy** (vec_type p)	Derivative in y using local differencing

Function	Description
vec_type **fwidth** (vec_type p)	Sum of absolute derivative in x and y