Core Image Kernel Language Reference

Developer

Contents

Overview 3

CIKernel Function Requirements 4

Data Types 5
Scalar Types . . . 5
Vector Types . . . 5
Matrix Types . . . 6
Sampler Types . . . 6

Operators 7

Functions 8
Relational Functions . . . 8
Math Functions . . . 8
Trigonometry Functions . . . 9
Matrix Functions . . . 9
Geometry Functions . . . 9
Color Functions . . . 10
Sampling Functions . . . 10

Additional Language Features 12
Globals . . . 12
Control Flow . . . 12
Type Qualifiers . . . 12
Attributes . . . 12

Overview

The Core Image Kernel Language is a simple shading language that you use to add custom image processing routines to a Core Image pipeline. This document defines the features supported in Core Image Kernel Language.

Source code written in Core Image Kernel Language should contain one or more image processing routines; optionally, it can contain other functions called by those routines. The source code is parsed and validated when it's passed to the APIs that create `CIKernel` objects. The source code is compiled when `CIImage` objects that are created using `CIKernel` objects are rendered to a `CIContext`. When rendering, Core Image concatenates kernel functions to optimize shader programs.

CIKernel Function Requirements

Denote a `CIKernel` function with the `kernel` keyword qualifier. The `[CIKernel kernelWithString...]` API returns the first `kernel`-qualified function in the source. Similarly, the `[CIKernel kernelsWithString...]` function returns an array of all `kernel`-qualified functions in the kernel source.

Write `CIKernel` functions to return a four-element vector type, except for `CIWarpKernel` functions, which return a two-element vector type.

The name of your `CIKernel` must not start with an underscore character (_), which is reserved for Apple's use.

Data Types

Core Image Kernel Language supports a relatively minimal set of primitive scalar data types, along with their respective vector and matrix types. Historically, Core Image Kernel Language keywords are derived from the OpenGL Shading Language (GLSL). Additional Metal-styled type aliases have been added to iOS 12 and macOS 10.14.

Scalar Types

Core Image Kernel Language supports the scalar types listed below:

Type	Description
`int`	A signed two's complement 32-bit integer. Cannot be used as the parameter type of a `kernel`-qualified function.
`bool`	A conditional data type with value either `true` or `false`. `true` has integer value 1, and `false` has integer value 0. Cannot be used as the parameter type of a `kernel`-qualified function.
`float`	A 32-bit floating point. Must conform to the IEEE 754 single-precision storage format.
`half`[1]	A 16-bit floating point. Must conform to the IEEE 754 binary16 storage format. Supported only when rendering using a Metal-backed `CIContext`; treated as 32-bit floats otherwise.

[1] Available in iOS 12 and later and macOS 10.14 and later.

Vector Types

Core Image Kernel Language supports n-dimensional vector types derived from the scalar types, where n is 2, 3, or 4.

Type	Alias	Description
`bvec<n>`	`bool<n>`[1]	An n-element vector of Boolean values. Cannot be used as the parameter type of a `kernel`-qualified function.
`ivec<n>`	`int<n>`[1]	An n-element vector of signed two's complement 32-bit integers. Cannot be used as the parameter type of a `kernel`-qualified function.
`vec<n>`	`float<n>`[1]	An n-element vector of 32-bit floating point values.
`hvec<n>`[1]	`half<n>`[1]	An n-element vector of 16-bit floating point values.

[1] Available in iOS 12 and later and macOS 10.14 and later.

The type `__color` is an alias to the type `vec4` and represents a premultiplied RGBA color.

Matrix Types

Core Image Kernel Language supports n-by-n square matrix types derived from the floating-point scalar types, where n is 2, 3, or 4.

Type	Alias	Description
`mat<n>`	`float<n>x<n>`[1]	An n-by-n square matrix of 32-bit floating-point values.
`hmat<n>`[1]	`half<n>x<n>`[1]	An n-by-n square matrix of 16-bit floating-point values.

[1] Available in iOS 12 and later and macOS 10.14 and later.

Sampler Types

Type	Alias	Description
`sample_f`[1]	`__sample`[2]	A sample value from a `CIImage` represented by a 4D 32-bit floating-point vector. Use as a parameter type only for representing a sample from an image. Otherwise behaves as a `vec4`.
`sample_h`[1]		A sample value from a `CIImage` represented by a 4D 16-bit floating-point vector. Use as a parameter type only for representing a sample from an image. Otherwise behaves as an `hvec4`.
`sampler_f`[1]	`sampler`	A sampler for a `CIImage` that returns 4D 32-bit floating-point precision samples.
`sampler_h`[1]		A sampler for a `CIImage` that returns 4D 16-bit floating-point precision samples.

[1] Available in iOS 12 and later and macOS 10.14 and later.
[2] Available in iOS 8 and later and macOS 10.11 and later.

Operators

Core Image Kernel Language supports the following operators on scalar, vector, and matrix types:

Category	Operators	Supported Types
Basic Math	+ − ∗ /	Vectors, matrices, and mixed types
Assignment	= += −= ∗= /=	Vectors, matrices, and mixed types
Comparison	== != < ≤ > ≥	Vectors, matrices, and mixed types
Bitwise Logic	~ ! ^ & |	Vector types
Boolean Logic	! && ||	Vector types

In addition, the ternary conditional operator `a ? b : c` is supported, where `a` is a scalar Boolean expression, and `b` and `c` are values of the same type.

Functions

You can write custom functions in Core Image Kernel Language with standard C syntax. Functions are overloadable. Most functions operate elementwise on vectors and matrices. Tables of built-in Core Image Kernel Language functions are written in the following type shorthand:

- `bvec` represents any of `bvec2, bvec3, bvec4`
- `ivec` represents any of `ivec2, ivec3, ivec4`
- `vec` represents any of `vec2`, `vec3`, `vec4`, `hvec2`, `hvec3`, `hvec4`
- `anyvec` represents any vector type
- `type` represents any of `float`, `vec2`, `vec3`, `vec4`, `half`, `hvec2`, `hvec3`, `hvec4`
- `mat` represents any of `mat2`, `mat3`, `mat4`, `hmat2`, `hmat3`, `hmat4`

Relational Functions

Function	Returns
`bool any(bvec)`	`true` if any argument is `true`
`bool all(bvec)`	`true` if all arguments are `true`
`bool not(bvec)`	`true` if no argument is `true`
`bvec equal(anyvec, anyvec)`	Elementwise equality of two vectors
`bvec notEqual(anyvec, anyvec)`	Elementwise inequality of two vectors
`bvec lessThan(anyvec, anyvec)`	Elementwise < of two vectors
`bvec lessThanEqual(anyvec, anyvec)`	Elementwise ≤ of two vectors
`bvec greaterThan(anyvec, anyvec)`	Elementwise > of two vectors
`bvec greaterThanEqual(anyvec, anyvec)`	Elementwise ≥ of two vectors
`type compare(type c, type a, type b)`	Elementwise `(c < 0) ? a : b`

Math Functions

Function	Returns
`type pow(type a, type b)`	a to the power of b
`type exp(type)`	e to the power of value
`type log(type)`	log base e of value
`type exp2(type)`	2 to the power of value
`type log2(type)`	log base 2 of value
`type sqrt(type)`	Square root of value
`type inversesqrt(type)`	Inverse square root of value
`type abs(type)`	Absolute value
`type floor(type)`	Closest integer less than or equal to value
`type ceil(type)`	Closest integer greater than or equal to value
`type fract(type x)`	`x - floor(x)`
`type mod(type x, type y)`	`x - y * floor(x/y)`
`type min(type)`	Minimum of two values
`type max(type)`	Maximum of two values
`type clamp(type x, type a, type b)`	`min(max(x,a),b)`

Function	Returns
`type clamp(type x, float a, float b)`	`min(max(x,a),b)`
`type mix(type x, type y, type a)`	`x*(1-a) + y*a`
`type mix(type x, type y, float a)`	`x*(1-a) + y*a`
`type step(type edge, type x)`	`x < edge ? 0 : 1`
`type step(float edge, type x)`	`x < edge ? 0 : 1`
`type sign(type x)`	`x < 0 ? -1 ? x > 0 : 1 : 0`
`type smoothstep(type edge0, type edge1, type x)`	`0` if x ≤ `edge0`, `1` if x ≥ `edge1`, Hermite in between
`type smoothstep(float edge0, float edge1, type x)`	`0` if x ≤ `edge0`, `1` if x ≥ `edge1`, Hermite in between

Trigonometry Functions

Function	Returns
`type radians(type)`	Degrees converted to radians
`type degrees(type)`	Radians converted to degrees
`type sin(type)`	Sine of an angle in radians
`type cos(type)`	Cosine of an angle in radians
`type tan(type)`	Tangent of an angle in radians
`type asin(type)`	Inverse sine as an angle between 0 and π
`type acos(type)`	Inverse cosine as an angle between -π/2 and +π/2
`float atan(float)`	Inverse tangent as an angle between -π/2 and +π/2
`float atan(vec2)`	Inverse tangent of `.y/.x` as an angle between -π and π
`vec2 sincos(float)`	Vector containing sine and cosine of an angle
`vec2 cossin(float)`	Vector containing cosine and sine of an angle

Matrix Functions

Function	Returns
`mat matrixCompMult(mat x, mat y)`	Elementwise product of two matrices
`float determinant(mat)`	Scalar determinant of a matrix
`mat determinant(mat)`	Transpose of a matrix
`mat inverse(mat)`	Inverse of a matrix

Geometry Functions

Function	Returns
`float dot(type x, type y)`	Scalar dot product of two vectors
`float length(type)`	Scalar length of a vector
`float distance(type x, type y)`	Length of the `x - y` vector
`float normalize(type)`	Length-1 vector in the same direction

Function	Returns
`vec3 cross(vec3 x, vec3 y)`	Vector cross product of `x` and `y`

Color Functions

Function	Returns
`vec4|hvec4 premultiply(vec4|hvec4)`	Multiplies red, green, and blue components of the parameter by its alpha component.
`vec4|hvec4 unpremultiply(vec4|hvec4)`	If the alpha component of the parameter is greater than 0, divides the red, green, and blue components by alpha. If alpha is 0, this function returns the parameter.
`vec3|hvec3 srgb_to_linear(vec3|hvec3)`	`abs(s) < 0.04045 ? (s / 12.92) : sign(s) * pow(abs(s)*0.947867298578199 + 0.052132701421801, vec3(2.4))`
`vec3|hvec3 linear_to_srgb(vec3|hvec3)`	`abs(s) < 0.0031308 ? (s * 12.92) : sign(s) * pow(abs(s), vec3(1.0/2.4)) * 1.055 - 0.055)`
`vec4|hvec4 srgb_to_linear(vec4|hvec4)`	`unpremultiply(s);srgb_to_linear(s.rgb);premultiply(s);`
`vec4|hvec4 linear_to_srgb(vec4|hvec4)`	`unpremultiply(s);linear_to_srgb(s.rgb);premultiply(s);`

Sampling Functions

destCoord

```
float2 destCoord()
```

Returns the position, in working space coordinates, of the pixel currently being computed. The destination space refers to the coordinate space of the image you're rendering.

sample

```
float4 sample(sampler_f src, float2 coord)
half4 sample(sampler_h src, float2 coord)
```

Returns the pixel value produced from sampler `src` at the position `coord`, where `coord` is specified in the sampler's coordinate system.

samplerTransform

```
float2 samplerTransform(sampler_f src, float2 coord)
float2 samplerTransform(sampler_h src, float2 coord)
```

Returns the position in the coordinate space of the `src` argument that's associated with the position defined in working space coordinates `coord`. Working space coordinates reflect any transformations that you've applied to the working space.

For example, if you're producing a pixel in the working space, and you need to retrieve the pixels that surround this pixel in the original image, you'd make calls similar to the following, where d is the location of the pixel you're producing in the working space, and `image` is the image source for the pixels.

```
samplerTransform(src, d + vec2(-1.0,-1.0));
samplerTransform(src, d + vec2(+1.0,-1.0));
samplerTransform(src, d + vec2(-1.0,+1.0));
samplerTransform(src, d + vec2(+1.0,+1.0));
```

samplerCoord

```
float2 samplerCoord(sampler_f src)
float2 samplerCoord(sampler_h src)
```

Returns the position, in sampler space, of the sampler `src` that's associated with the current output pixel after applying any transformation matrix associated with `src`. The sample space refers to the coordinate space you're texturing from. If your source data is tiled, the sample coordinate will have an offset `(dx/dy)`. You can convert a destination location to the sampler location using the `samplerTransform` function, which is equivalent to `samplerTransform(src, destCoord())`.

samplerExtent

```
float4 samplerExtent(sampler_f src)
float4 samplerExtent(sampler_h src)
```

Returns the extent `(x, y, width, height)` of the sampler in world coordinates as a four-element vector. If the extent is infinite, the vector `(-INF,-INF,INF,INF)` is returned.

samplerOrigin

```
float2 samplerOrigin(sampler_f src)
float2 samplerOrigin(sampler_h src)
```

Returns the origin of the sampler extent; equivalent to `samplerExtent(src).xy`.

samplerSize

```
float2 samplerSize(sampler_f src)
float2 samplerSize(sampler_h src)
```

Returns the size of the sampler extent; equivalent to `samplerExtent(src).zw`.

Additional Language Features

Globals

Core Image Kernel Language supports declaring global scope constants of scalar, vector, or matrix type.

```
const float rotation = 1.6180339887498;
const vec2 d65illuminant = vec2(0.31271, 0.32902);
```

Control Flow

Core Image Kernel Language supports the following standard control statements from C:

`if, then, else, for, while, break, continue`

Type Qualifiers

On macOS 10.10 and earlier, you could qualify sampler parameters as `__table` to alter the behavior of ROI callbacks for that sampler, but now this qualifier has no effect.

Attributes

For kernel functions, you can use optional attributes to indicate additional kernel properties.

```
kernel vec4 myColorKernel(sample_f s) __attribute__((<attribute_list>))
{
    return s;
}
```

`<attribute_list>` is a comma-separated list of attribute keywords or `attribute(value)` pairs.

Core Image Kernel Language supports the following kernel function attribute:

Attribute	Description
`outputFormat`	Specifies the desirable output format of a kernel function. The format argument should be the name of a `CIFormat`, like `__attribute__(outputFormat(kCIFormatA8))`.

Copyright and Notices

Apple Inc.
One Apple Park Way
Cupertino, CA 95014
USA
408-996-1010