Metal Shading Language Guide

Metal着色语言指南

Introduction

The Metal shading language is a unified programming language for writing both graphics and compute kernel functions that are used by apps written with the Metal framework.

The Metal shading language is designed to work together with the Metal framework, which manages the execution, and optionally the compilation, of the Metal shading language code. The Metal shading language uses clang and LLVM so developers get a compiler that delivers close to the metal performance for code executing on the GPU.

Metal着色语言是一个用来编写3d图形渲染逻辑 和 并行计算核心逻辑的编程语言，编写Metal框架的app需要使用Metal着色语言程序。

Metal着色语言和Metal框架配合使用，Metal框架管理Metal着色语言的运行和可选编译选项。Metal着色语言使用clang和 LLVM，编译器对于在GPU上的代码执行效率有更好的控制。

At a Glance

This document describes the Metal unified graphics and compute shading language. The Metal shading language is a C++ based programming language that developers can use to write code that is executed on the GPU for graphics and general-purpose data-parallel computations. Since the Metal shading language is based on C++, developers will find it familiar and easy to use. With the Metal shading language, both graphics and compute programs can be written with a single, unified language, which allows tighter integration between the two.

本文描述了Metal着色语言的使用（图形渲染和并行计算）。Metal着色语言基于C++设计，开发者可以用它来编写在GPU上执行的图形渲染逻辑代码和通用并行计算逻辑代码。因为是基于C++设计，开发者会发现它熟悉便于使用。使用Metal着色语言，图形渲染和并行计算可以使用同一种语言，这就允许把两种任务结合在一起完成。

How to Use This Document

Developers who are writing code with the Metal framework will want to read this document, because they will need to use the Metal shading language to write graphics and compute programs to be executed on the GPU. This document is organized into the following chapters:

- Metal and C++11 (page 8) covers the similarities and differences between the Metal shading language and C++11.
- Metal Data Types (page 10) lists the Metal shading language data types, including types that represent vectors, matrices, buffers, textures, and samplers. It also discusses type alignment and type conversion.
- Operators (page 30) lists the Metal shading language operators.
- Functions, Variables, and Qualifiers (page 36) details how functions and variables are declared, sometimes with qualifiers that restrict how they are used.
- Metal Standard Library (page 62) defines a collection of built-in Metal shading language functions.
- Compiler Options (page 93) details the options for the Metal shading language compiler, including pre-processor directives, options for math intrinsics, and options that control optimization.
- Numerical Compliance (page 95) describes requirements for representing floating-point numbers, including accuracy in mathematical operations.

使用Metal框架的开发者应该阅读本文，因为他们需要使用Metal着色语言编写运行在GPU上的图形渲染和并行计算逻辑。本文包含如下的章节：

- Metal and C++11， 描述Metal着色语言和它的基础C++11的异同。

- Metal数据类型，　　列举Metal着色语言的数据类型，包括表示向量，矩阵，缓存，纹理，采样器等等，还描述了类型对齐和类型转换。
- 运算符，　　列举Metal着色语言操作运算符。
- 方法、变量、修饰符，　　详细描述了方法变量如何声明，已经如何使用修饰符修饰，表示使用时的限制。
- Metal标准库，　　描述了一些Metal着色语言内建的方法如何使用。
- 编译选项　　描述了Metal着色语言编译器的可选项，包括预编译指令，数学内联计算函数选项和其他优化控制选项。
- 数值规则，　　描述了表示浮点数的要求，包括数学运算的精度。

See Also

C++11

Stroustrup, Bjarne. The C++ Programming Language. Harlow: Addison-Wesley, 2013.

Metal Framework

The 《*Metal Programming Guide*》 provides a detailed introduction to writing apps with the Metal framework. 详细介绍如何使用Metal框架编写app

The 《*Metal Framework Reference*》 details individual classes in the Metal framework. 详细描述每一个Metal框架中的类。

Metal and C++11

The Metal shading language is based on the *C++11 Specification* (a.k.a., the ISO/IEC JTC1/SC22/WG21 N3290 Language Specification) with specific extensions and restrictions. Please refer to the C++11 Specification for a detailed description of the language grammar. This section and its subsections describe modifications and restrictions to the C++11 language supported in the Metal shading language. For more information about Metal shading language pre-processing directives and compiler options, see Compiler Options (page 93) of this document.

Metal着色语言是基于C++ 11标准设计的，它在C++基础上多了一些扩展和限制。下面的内容就是在描述Metal着色语言相比C++11的修改和限制。Metal着色语言的预编指令和编译器选项后面的章节有描述。

Overloading 重载

The Metal shading language supports overloading as defined by section 13 of the *C++11 Specification* . The function overloading rules are extended to include the address space qualifier of an argument. The Metal shading language graphics and kernel functions cannot be overloaded. (For definition of graphics and kernel functions, see Function Qualifiers (page 36).)

Metal着色语言支持的重载如同C++11标准的section 13所述，方法重载规则有扩展，可以包括参数的地址空间描述符。Metal着色语言中的标识为图形渲染入口函数或是并行计算入口函数的不可以被重载。（如何标识函数为图形渲染入口函数 或 并行计算入口函数，参见后面的章节）。

Templates 模板

The Metal shading language supports templates as defined by section 14 of the *C++11 Specification* .

Metal着色语言支持的模板如同C++11标准的section 14所述

Preprocessing Directives

预编译指令

The Metal shading language supports the pre-processing directives defined by section 16 of the *C++11 Specification* .

Metal着色语言支持的预编译指令如同C++11标准的section 16所述

Restrictions 限制

The following C++11 features are not available in the Metal shading language (section numbers in this list refer to the *C++11 Specification*):

如下的C++11特性在Metal着色语言中不支持：

- lambda expressions (section 5.1.2) lambda表达式
- recursive function calls (section 5.2.2, item 9) 递归函数调用
- dynamic_cast operator (section 5.2.7) 动态转换操作符
- type identification (section 5.2.8) 类型识别
- new and delete operators (sections 5.3.4 and 5.3.5) 对象创建(new)和销毁（delete）操作符
- noexcept operator (section 5.3.7) 操作符 noexcept
- goto statement (section 6.6) goto跳转
- register, thread_local storage qualifiers (section 7.1.1) 变量存储修饰符register 和 thread_local
- virtual function qualifier (section 7.1.2) 虚函数修饰符
- derived classes (sections 10 and 11) 派生类
- exception handling (section 15) 异常处理

The C++ standard library must not be used in the Metal shading language code. Instead of the C++ standard library, Metal has its own standard library that is described in Metal Standard Library (page 62).

C++标准库不可以在Metal着色语言中使用，Metal着色语言使用自己的标准库，后面专门有一个章节描述。

The Metal shading language restricts the use of pointers: Metal着色语言中对于指针的使用的限制

- Arguments to Metal graphics and kernel functions that are pointers declared in a program must be declared with the Metal `device`, `threadgroup`, or `constant` address space qualifier. (See Address Space Qualifiers for Variables and Arguments (page 37) for more about address space qualifiers.)
  Metal图形和并行计算函数用到的入参如果是指针必须使用地址空间修饰符（`device`, `threadgroup`, `constant`）。
- Function pointers are not supported.
  不支持函数指针。

A Metal function cannot be called `main`.

Metal函数名不能叫main。

Metal Pixel Coordinate System

Metal像素坐标系统

In Metal, the origin of the pixel coordinate system of a texture or a framebuffer attachment is defined at the top left corner.

在Metal中，纹理 或是 帧缓存attachment 中的像素 使用的坐标系统的原点定义在左上角。

Metal Data Types

Metal数据类型

This chapter details the Metal shading language data types, including types that represent vectors and matrices. Atomic data types, buffers, textures, samplers, arrays, and user-defined structs are also discussed. Type alignment and type conversion are also described.

本章详述Metal着色语言的数据类型，包括表示向量和矩阵的类型，原子数据类型，缓存，纹理，采样器，数组，已经用户自定义结构体。还会描述类型对齐和类型转换。

Scalar Data Types

标量数据类型

Metal supports the scalar types listed in Table 2-1 (page 10). Metal does **not** support the `double`, `long`, `unsigned long`, `long long`, `unsigned long long`, and `long double` data types.

Metal支持如表2-1列举的标量数据类型，Metal不支持这些数据类型：`double`, `long`, `unsigned long`, `long long`, `unsigned long long`, `long double`。

Table 2-1 Metal scalar data types

Type	Description
`bool`	A conditional data type that has the value of either `true` or `false`. The value `true` expands to the integer constant 1, and the value `false` expands to the integer constant 0. 布尔数据类型，取值有true或false，true可以扩展为整数常量1，false可以扩展为整数常量0
`char`	A signed two's complement 8-bit integer. 有符号8-bit整数
`unsigned char` `uchar`	An unsigned 8-bit integer. 无符号8-bit整数
`short`	A signed two's complement 16-bit integer. 有符号16-bit整数
`unsigned short` `ushort`	An unsigned 16-bit integer. 无符号16-bit整数
`int`	A signed two's complement 32-bit integer. 有符号32-bit整数
`unsigned int` `uint`	An unsigned 32-bit integer. 无符号32-bit整数
`half`	A 16-bit floating-point. The half data type must conform to the IEEE 754 binary16 storage format. 一个16-bit浮点数，符合IEEE754的16位二进制浮点数存储格式规范
`float`	A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format. 一个32-bit浮点数，符合IEEE754的单精度浮点数存储格式规范
`size_t`	An unsigned integer type of the result of the `sizeof` operator. This is a 64-bit unsigned integer. 64-bit无符号整数，表示sizeof操作符的结果，

`ptrdiff_t`	A signed integer type that is the result of subtracting two pointers. This is a 64-bit signed integer. 64-bit有符号整数，表示两个指针的差。
`void`	The `void` type comprises an empty set of values; it is an incomplete type that cannot be completed. 该类型表示一个空的值集合，

Note: Metal supports the following suffixes that specify the type for a literal:

the standard `f` or `F` suffix to specify a single precision floating-point literal value (e.g., `0.5f` or `0.5F`).

the `h` or `H` suffix to specify a half precision floating-point literal value (e.g., `0.5h` or `0.5H`).

the `u` or `U` suffix for unsigned integer literals.

注意：Metal支持使用下面的后缀用来表示字面量类型

f或是F用来表示单精度浮点型字面量（比如 0.5f 或是 0.5F）

h或是H用来表示半单精度浮点型字面量（比如0.5h 或是0.5H）

u或是U用来表示无符号整形字面量。

Vector and Matrix Data Types

向量和矩阵数据类型

The Metal shading language supports a subset of the vector and matrix data types implemented by the system vector math library.

The vector type names supported are:

```
booln charn,
shortn, intn, ucharn, ushortn, uintn
halfn and floatn
```

`n` is 2, 3, or 4 representing a 2-, 3- or 4- component vector type.

Metal着色语言通过系统向量数学库支持一系列的向量和矩阵数据类型。

Metal支持的向量类型名如下：booln, charn, shortn, intn, ucharn, ushortn, uintn, halfn, floatn

n的取值为2,3或是4，分别表示一个2维，3维或是4维向量类型。

The matrix type names supported are:

`halfnxm` and `floatnxm`

where `n` and `m` are number of columns and rows. `n` and `m` can be 2, 3, or 4. A matrix is composed of several vectors. For example, a `floatnx3` matrix is composed of `n float3` vectors. Similarly, a `halfnx4` matrix is composed of `n half4` vectors.

Metal支持的矩阵类型名如下：halfnxm, floatnxm

n和m分别表示矩阵列数和行数，n和m可以是2,3或是4.一个矩阵被看做是由几个向量构成，比如一个floatnx3的矩阵由n个float3类型向量构成。类似的一个halfnx4矩阵由n个half4类型向量构成。

Accessing Vector Components

访问向量的分量

Vector components can be accessed using an array index. Array index 0 refers to the first component of the vector, index 1 to the second component, and so on. The following examples show various ways to access array components:

向量的分量可以使用数组下标进行访问存取。数组下标0引用了向量的第一个分量，数组下标1引用了向量的第二个分量，以此类推。下面的例子展示了多种访问向量分量：

```
pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
float x = pos[0];   // x = 1.0
float z = pos[2];   // z = 3.0
float4 vA = float4(1.0f, 2.0f, 3.0f, 4.0f);
float4 vB;
for (int i=0; i<4; i++)
   vB[i] = vA[i] * 2.0f // vB = (2.0, 4.0, 6.0, 8.0);
```

Metal supports using the period (.) as a selection operator to access vector components, using letters that may indicate coordinate or color data: `<vector_data_type>.xyzw` or `<vector_data_type>.rgba`.

In the following code, the vector test is initialized, and then components are accessed using the `.xyzw` or `.rgba` selection syntax:

Metal支持使用（.）作为选择向量分量进行访问的操作符，可以使用表示坐标分量或是颜色分量的字母来存取向量：向量名.xyzw 或是 向量名.rgba。

下面的代码，向量test被初始化，然后使用.xyzw 和 .rgba 选择语法来访问它的各个分量：

```
int4 test = int4(0, 1, 2, 3);
int a = test.x; // a=0
int b = test.y; // b=1
int c = test.z; // c=2
int d = test.w; // d=3
int e = test.r; // e=0
int f = test.g; // f=1
int g = test.b; // g=2
int h = test.a; // h=3
```

The component selection syntax allows multiple components to be selected.

分量选择语法允许多个分量同时被选择访问。如下面的例子所示：

```
float4 c;
c.xyzw = float4(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = float2(3.0f, 4.0f);
c.xyz = float3(3.0f, 4.0f, 5.0f);
```

The component selection syntax also allows components to be permuted or replicated.

分量选择语法允许多个分量乱序或是重复出现。如下面的例子所示：

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
float4 swiz = pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)
float4 dup = pos.xxyy;  // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

The component group notation can occur on the left hand side of an expression. To form the lvalue, swizzling may be applied. The resulting lvalue may be either the scalar or vector type, depending on number of components specified. Each component must be a supported scalar or vector type. The resulting lvalue of vector type must not contain duplicate components.

向量的分量组标记可以出现在表达式的左边，表示左值，乱序的分量也是支持的。左值可以是标量或是向量，这取决于被指定的分量的个数。每一个分量必须是一个被支持的标量或是向量类型。作为左值，不可以包含重复分量。以上描述的规则如下面的例子所示：

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);

// pos = (5.0, 2.0, 3.0, 6.0)
pos.xw = float2(5.0f, 6.0f);

// pos = (8.0, 2.0, 3.0, 7.0)
pos.wx = float2(7.0f, 8.0f);

// pos = (3.0, 5.0, 9.0, 7.0)
pos.xyz = float3(3.0f, 5.0f, 9.0f);
```

The following methods of vector component access are not permitted and result in a compile-time error:

如下的向量分量访问方法是不被允许会导致编译错误：

- Accessing components beyond those declared for the vector type is an error. 2-component vector data types can only access `.xy` or `.rg` elements. 3-component vector data types can only access `.xyz` or `.rgb` elements. For instance:

访问向量分量时，如果超过了向量声明的维度数会产生错误。一个2维向量可以通过 .xy 或 .rg 访问其分量。一个3维的向量可以通过 .xyz 或 .rgb 访问其分量，如下面的例子所示：

```
float2 pos;
pos.x = 1.0f; // is legal; so is y
pos.z = 1.0f; // is illegal; so is w
float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

- Accessing the same component twice on the left-hand side is ambiguous; for instance,

  如果作为左值，同一个分量出现多于一次是错误的，如下例所示：

```
// illegal - 'x' used twice
pos.xx = float2(3.0f, 4.0f);
// illegal - mismatch between float2 and float4
pos.xy = float4(1.0f, 2.0f, 3.0f, 4.0f);
```

- The .rgba and .xyzw qualifiers cannot be intermixed in a single access; for instance,

  .rbga 和 .xyzw 不能在同一次访问中混用，如下例所示：

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
pos.x = 1.0f;    // OK
pos.g = 2.0f;    // OK
pos.xg = float2(3.0f, 4.0f); // illegal - mixed qualifiers used
float3 coord = pos.ryz;  // illegal - mixed qualifiers used
```

- A pointer or reference to a vector with swizzles; for instance

  指向向量分量的指针或是引用也是不合法的。如下例所示：

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
my_func(&pos.xy);    // illegal
```

The `sizeof` operator on a vector type returns the size of the vector, which is given as the number of components * size of each component. For example, `sizeof(float4)` returns 16 and `sizeof(half4)` returns 8.

sizeof 操作符号返回向量的尺寸，这个尺寸等于向量分量的数量 * 每个分量的尺寸。比如sizeof(float4)返回16，而sizeof(half4) 返回8。

Accessing Matrix Components

访问矩阵分量

The `floatnxm` and `halfnxm` matrices can be accessed as an array of n `floatm` or n `halfm` entries.

The components of a matrix can be accessed using the array subscripting syntax. Applying a single subscript to a matrix treats the matrix as an array of column vectors. The first column is column 0. A second subscript would then operate on the resulting vector, as defined earlier for vectors. Hence, two subscripts select a column and then a row.

floatnxm可以被当做一个有n个floatm向量构成的数组来访问，halfnxm可以被当做一个有n个halfm向量构成的数组来访问。

矩阵的各分量可以使用数组下标语法来访问。如果只使用一个数组下标访问矩阵，那么数组就是被当做列向量数组被访问，第一列的下标是0。如果有第二个下标，那么表示访问列向量内的某个分量，也就是说两个下标先选择了列后选择行。下面就是使用数组下标访问矩阵分量的示例：

```
float4x4 m;
// sets the 2nd column to all 2.0
m[1] = float4(2.0f);

// sets the 1st element of the 1st column to 1.0
m[0][0] = 1.0f;

// sets the 4th element of the 3rd column to 3.0
m[2][3] = 3.0f;
```

Accessing a component outside the bounds of a matrix with a non-constant expression results in undefined behavior. Accessing a matrix component that is outside the bounds of the matrix with a constant expression generates a compile-time error.

如果使用一个非常量的表达式作为下标访问矩阵分量越界，那么这个访问行为未知。如果使用一个常量表达式作为下标访问矩阵分量越界，那么将产生一个编译时错误。

Vector Constructors

向量的构造

Constructors can be used to create vectors from a set of scalars or vectors. When a vector is initialized, its parameter signature determines how it is constructed. For instance, if the vector is initialized with only a single scalar parameter, all components of the constructed vector are set to that scalar value.

If a vector is constructed from multiple scalars, one or more vectors, or a mixture of these, the vector's components are constructed in order from the components of the arguments. The arguments are consumed from left to right. Each argument has all its components consumed, in order, before any components from the next argument are consumed.

This is a complete list of constructors that are available for `float4`:

构造器可以用一组标量或是向量来构造向量。当一个向量被初始化，参数签名决定了它如何被构造。比如，如果向量的构造只使用一个标量作为参数，那么这个向量所有的分量都被设置为这个标量的值。

如过构造向量时，使用多个标量，一个或是多个向量，或是标量和向量的混合，那么向量的构造就按照参数出现的顺序。构造时，参数从左到右被使用。每个参数所有的分量都被按顺序读取。

如下是float4类型向量的所有可能的构造方式：

```
float4(float x);
float4(float x, float y, float z, float w);
float4(float2 a, float2 b);
float4(float2 a, float b, float c);
float4(float a, float b, float2 c);
float4(float a, float2 b, float c);
float4(float3 a, float b);
float4(float a, float3 b);
float4(float4 x);
```

This is a complete list of constructors that are available for `float3`:

如下是float3类型向量的所有可能的构造方式：

```
float3(float x);
float3(float x, float y, float z);
float3(float a, float2 b);
float3(float2 a, float b);
float3(float3 x);
```

This is a complete list of constructors that are available for `float2`:

如下是float2类型向量的所有可能的构造方式：

```
float2(float x);
float2(float x, float y);
float2(float2 x);
```

The following examples illustrate uses of the constructors:

下面的例子展示了多个向量构造器的实际使用：

```
float x = 1.0f, y = 2.0f, z = 3.0f, w = 4.0f;
float4 a = float4(0.0f);
float4 b = float4(x, y, z, w);
float2 c = float2(5.0f, 6.0f);
float2 a = float2(x, y);
float2 b = float2(z, w);
float4 x = float4(a.xy, b.xy);
```

Under-initializing a vector constructor is a compile-time error.

如果构造器参数不足会产生编译时错误。

Matrix Constructors

矩阵的构造

Constructors can be used to create matrices from a set of scalars, vectors or matrices. When a matrix is initialized, its parameter signature determines how it is constructed. For example, if a matrix is initialized with only a single scalar parameter, the result is a matrix that contains that scalar for all components of the matrix's diagonal, with the remaining components initialized to 0.0. For example, a call to

矩阵构造器可以使用一组标量、向量、或是矩阵来创建新矩阵。当矩阵被初始化，构造器的参数签名决定矩阵如何被构造。比如，如果一个矩阵只使用一个标量初始化，那么构造出来的矩阵的对角线上的分量被赋值成这个标量的值，其他的矩阵分量被设置为0.0，如下例所示：

```
float4x4(fval);
```

where `fval` is a scalar floating-point value, constructs a matrix with these initial contents:

fval是一个标量浮点数，那么构造出来的矩阵的各分量如下所示：

```
fval   0.0   0.0   0.0
0.0   fval   0.0   0.0
0.0   0.0   fval   0.0
0.0   0.0   0.0   fval
```

A matrix can also be constructed from another matrix that is of the same size, i.e., has the same number of rows and columns. For example,

一个矩阵也可以由另一个有同样尺寸（也就是有相同的行数和列数）的矩阵构造，比如：

```
float3x4(float3x4);
float3x4(half3x4);
```

Matrix components are constructed and consumed in column-major order. The matrix constructor must have just enough values specified in its arguments to initialize every component in the constructed matrix object. Providing more arguments than are needed results in an error. Under-initializing a matrix constructor also results in a compile-time error.

如上从另一个矩阵构造新矩阵的情况下，矩阵的各分量将以列主序来构造。作为参数矩阵必须要有足够的分量来构造新矩阵的每一个分量，提供了过多的分量（参数矩阵的尺寸比构造矩阵大）会导致编译错误。如果参数矩阵的分量个数不足用来初始化（参数矩阵的尺寸比构造矩阵小）也会导致编译错误。

A matrix of type `T` with `n` columns and `m` rows can also be constructed from `n` vectors of type `T` with `m` components. The following examples are legal constructors:

一个T类型的n列m行的矩阵可以被n个T类型的m维向量初始化。下面的例子都是合法的初始化：

```
float2x2(float2, float2);
float3x3(float3, float3, float3);
float3x2(float2, float2, float2);
```

The following are examples of matrix constructors that are not supported. A matrix cannot be constructed from multiple scalar values, nor from combinations of vectors and scalars.

如下面的例子中的矩阵的构造是不支持的。一个矩阵不支持从多个标量构造，也不支持标量和向量混合构造。

```
// both cases below are not supported
float2x2(float a00, float a01, float a10, float a11);
float2x3(float2 a, float b, float2 c, float d);
```

Atomic Data Types

原子数据类型

The Metal atomic data type is restricted for use by atomic functions implemented by the Metal shading language, as described in Atomic Functions (page 89). These atomic functions are a subset of the C++11 atomic and synchronization functions. Metal atomic functions must operate on Metal atomic data.

The Metal atomic types are defined as: `atomic_int` and `atomic_uint`.

Metal原子数据类型被限制于只能在Metal着色语言提供的原子函数中使用。原子函数是一组C++ 11标准的原子同步函数。Metal原子函数必须操作Metal原子数据类型数据。

Metal原子数据类型被定义为：atomic_int 和 atomic_uint

Buffers

缓存

Metal implements buffers as a pointer to a built-in or user defined data type described in the `device` or `constant` address space. (Refer to Address Space Qualifiers for Variables and Arguments (page 37) for a full description of these address qualifiers.) These buffers can be declared in program scope or passed as arguments to a function.

Metal中实现的缓存是一个指针，它指向一个在device 或是 constant地址空间中的内建或是用户自定义的数据块。缓存可以被定义在程序域中，或是当做函数的参数传递。

Example:

```
device float4   *device_buffer;
struct my_user_data {
   float4 a;
   float  b;
   int2   c;
};
constant my_user_data *user_data;
```

Textures

纹理

The texture data type is a handle to one-, two-, or three-dimensional texture data that corresponds to all or a portion of a single mipmap level of a texture. The following templates define specific texture data types:

纹理数据类型是一个句柄，它指向一个1维、2维或是3维纹理数据，而纹理数据对应着一个纹理的某个level的mipmap的全部或是一部分。下面的模板定义了特定的纹理数据类型。

```
enum class access { sample, read, write };

texture1d<T, access a = access::sample>
texture1d_array<T, access a = access::sample>
texture2d<T, access a = access::sample>
texture2d_array<T, access a = access::sample>
texture3d<T, access a = access::sample>
texturecube<T, access a = access::sample>
texture2d_ms<T, access a = access::read>
```

Textures with depth formats must be declared as one of the following texture data types:

带有深度格式的纹理必须被声明为下面纹理数据类型中的一个：

```
enum class depth_format { depth_float };
depth2d<T, access a = access::sample,
         depth_format d = depth_format::depth_float>
depth2d_array<T, access a = access::sample,
           depth_format d = depth_format::depth_float>
depthcube<T, access a = access::sample,
           depth_format d = depth_format::depth_float>
depth2d_ms<T, access a = access::read,
           depth_format d = depth_format::depth_float>
```

`T` specifies the color type returned when reading from a texture or the color type specified when writing to the texture. For texture types (except depth texture types), `T` can be `half`, `float`, `short`, `ushort`, `int`, or `uint`. For depth texture types, `T` must be `float`.

T设定了从纹理中读取或是向纹理中写入时的颜色类型，纹理数据类型（除了深度纹理类型），T可以是half，float，short，ushort, int 或是uint。对于深度纹理，T必须是float。

Note: If `T` is `int` or `short`, the data associated with the texture must use assigned integer format. If `T` is `uint` or `ushort`, the data associated with the texture must use an unsigned integer format. If `T` is `half`, the data associated with the texture must

either be a normalized (signed or unsigned integer) or half precision format. If `T` is `float`, the data associated with the texture must either be a normalized (signed or unsigned integer), half or single precision format.

注意：如果T是int或是short，纹理相关的数据必须使用有符号整形。如果T是uint或是ushort，纹理相关数据必须使用无符号整形。如果T是half，和纹理相关数据必须要么是归一化的（整形），要么是half，要么是单精度浮点数。

The `access` qualifier describes how the texture can be accessed. The supported access qualifiers are:

- `sample` - The texture object can be sampled. `sample` implies the ability to read from a texture with and without a sampler.
- `read` - Without a sampler, a graphics, or kernel function can only read the texture object.
- `write` – A graphics or kernel function can write to the texture object.

Access修饰符描述了纹理如何被访问。被支持的access修饰符如下：

- sample – 纹理对象可以被采样，采样意味着使用或是不使用采样器从纹理中读取数据。
- read – 不使用采样器，一个图形渲染函数或是一个并行计算函数可以读取纹理对象。
- write - 一个图形渲染函数或是一个并行计算函数可以向纹理对象写入数据。

The `depth_format` qualifier describes the depth texture format. The only supported value is `depth_format`.

`depth_format`修饰符描述了深度纹理格式。唯一支持的值是`depth_format`.

Note: For multisampled textures, only the `read` qualifier is supported. For cube textures, only the `sample` and `read` qualifiers are supported. For depth textures, only the `sample` and `read` qualifiers are supported.A pointer or a reference to a texture type is not supported and will result in a compilation error.

注意：对于多重采样纹理，只有`read`修饰符是支持的。对于立方纹理，只有`sample`和`read`修饰符是支持的。对于深度纹理，只有`sample`和`read`修饰符是支持的。纹理指针或是引用是不支持的，将会导致编译错误。

The following example uses these access qualifiers with texture object arguments.

下面的例子，纹理对象参数使用了access修饰符

```
void foo (texture2d<float> imgA [[ texture(0) ]],
   texture2d<float, access::read> imgB [[ texture(1) ]],
   texture2d<float, access::write> imgC [[ texture(2) ]])
{
   ...
}
```

(See Attribute Qualifiers to Locate Resources (page 40) for description of the `texture` attribute qualifier.)

Samplers

采样器

In the Metal shading language, the `sampler` type identifies how to sample a texture. The Metal framework allows you to create a corresponding `MTLSamplerState` object and pass it in an argument to a graphics or kernel function. A `sampler` object can also be described in Metal shading language program source instead of in the Metal framework. For these cases we only allow a subset of the sampler state to be specified: the addressing mode, filter mode, normalized coordinates, and comparison function.

Table 2-2 (page 20) describes the list of supported sampler state enums and their associated values (and defaults). These states can be specified when a sampler is initialized in Metal shading language program source.

在Metal着色语言中，采样器类型决定了如何对一个纹理进行采样操作。Metal框架代码中可以创建一个对应着色语言中的采样器的对象`MTLSamplerState`，这个对象作为图形渲染着色函数参数或是并行计算着色函数的参数传递。除了在Metal框架代码中定义描述，一个采样器对象也可以在Metal着色语言程序中定义描述。如果是这种情况，那么只允许设定部分的采样器状态：寻址模式，过滤模式，归一化坐标，比较函数。

表2-2描述了Metal支持的采样器状态和关联数值（默认值）。在Metal着色语言程序中，这些状态可以在一个采样器初始化时设定。

Table 2-2 Sampler State Enumeration Values

Enum Name	Valid Values	Description
`coord`	`normalized` (default) `pixel`	Specifies whether the texture coordinates when sampling from a texture are normalized or unnormalized values. 从纹理中采样时，纹理坐标是否是归一化的

`address`	`clamp_to_edge` (default)
	`clamp_to_zero`
	`mirrored_repeat`
	`repeat`
		Sets the addressing mode for all texture coordinates.
		设置所有的纹理坐标的寻址模式

`s_address`
`t_address`
`r_address`
	`clamp_to_edge` (default)
	`clamp_to_zero`
	`mirrored_repeat` `repeat`
		Sets the addressing mode for an individual texture coordinate.
		设置某个纹理坐标的寻址模式

`filter`	`nearest` (default) `linear`	Sets the magnification and minification filtering modes for texture sampling. 设置纹理采样的放大和缩小过滤模式
`mag_filter`	`nearest` (default) `linear`	Sets the magnification filtering mode for texture sampling. 设置纹理采样的放大过滤模式
`min_filter`	`nearest` (default) `linear`	Sets the minification filtering modes for texture sampling. 纹理采样的缩小过滤模式

`mip_filter`	`none` (default)
	`nearest`
	`linear`
		Sets the mipmap filtering mode for texture sampling. If `none`, then only one level-of-detail is active
		纹理采样的mipmap滤模式，如果是none，那么只有一个层的纹理生效。

Enum Name	Valid Values	Description

`compare_func`	`none` (default)
	`les`
	`less_equal`
	`greater`
	`greater_equal`
	`equal`
	`not_equal`
		Sets comparison test to use with r texture coordinate for shadow maps.
		The `compare_func` can only be specified for samplers declared in Metal shading language source.
		为使用r纹理坐标做shadow map设置比较测试逻辑。
		这个状态值的设置只可以在Metal着色语言程序中完成。

For the addressing mode, `clamp_to_zero` is similar to the OpenGL clamp to border addressing mode except that the border color value is always (0.0, 0.0, 0.0, 1.0) when sampling outside a texture that does not have an alpha component or is (0.0, 0.0, 0.0, 0.0) when sampling outside the texture that has an alpha component.

The enumeration types used by the sampler data type as described in Table 2-2 (page 20) are specified as follows. (If `coord` is set to `pixel`, the `min_filter` and `mag_filter` values must be the same, the `mip_filter` and `compare_func` values must be `none`, and the address modes must be either `clamp_to_zero` or `clamp_to_edge`.)

对于寻址模式，`clamp_to_zero`和OpenGL的clamp to border行为类似，只是当采样超出了纹理（无alpha通道）边缘的颜色值为(0.0, 0.0, 0.0, 1.0)，如果是有alpha通道的纹理，超出部分的颜色值为(0.0, 0.0, 0.0, 0.0)。

表2-2描述的采样器使用的枚举类型的定义如下所示（如果coord被设置为pixel，那么`min_filter`和`mag_filter`的值必须一致，`mip_filter`和`compare_func`的值必须为none，且寻址模式必须为`clamp_to_zero`或`clamp_to_edge`）。

```
enum class coord { normalized, pixel };
enum class filter { nearest, linear };
enum class min_filter { nearest, linear };
enum class mag_filter { nearest, linear };
enum class s_address { clamp_to_zero, clamp_to_edge, repeat, mirrored_repeat };
enum class t_address { clamp_to_zero, clamp_to_edge, repeat, mirrored_repeat };
enum class r_address { clamp_to_zero, clamp_to_edge, repeat, mirrored_repeat };
enum class address { clamp_to_zero, clamp_to_edge, repeat, mirrored_repeat };
enum class mip_filter { none, nearest, linear };
// can only be used with depth_sampler
enum class compare_func { none, less, less_equal, greater, greater_equal, equal, not_equal };
```

The Metal shading language implements a sampler object as follows:

Metal着色语言按照如下所示的代码实现一个采样器对象：

```
struct sampler {
   public:
      // full version of sampler constructor
      template<typename... Ts>
      constexpr sampler(Ts... sampler_params){};
   private:
};
```

`Ts` must be the enumeration types listed above that can be used by the sampler data type. If the same enumeration type is declared multiple times in a given sampler constructor, the last listed value will take effect.

如上的代码中 Ts 必须是一个如上列出的可以用于采样器的枚举类型。如果在一个采样器构造器中，一个同样的枚举类型被声明多次，最后一个值生效。

The following Metal program source illustrates several ways to declare samplers. (The attribute qualifiers (`sampler(n)`, `buffer(n)`, and `texture(n)`) that appear in the code below are explained in Attribute Qualifiers to Locate Resources (page 40).). Note that samplers or constant buffers declared in program source do not need these attribute qualifers.

下面这段Metal程序展示了多种声明采样器的方法。（其中的属性修饰符sampler(n), buffer(n), 和texture(n)将在后面的章节讲述）。注意，声明在程序域的采样器或常量缓存不需要这些属性修饰符。

```
constexpr sampler s(coord::pixel,
                    address::clamp_to_zero,
                    filter::linear);
constexpr sampler a(coord::normalized);
constexpr sampler b(address::repeat);
constexpr sampler s(address::clamp_to_zero,
                    filter::linear,
                    compare_func::less);
kernel void my_kernel(device float4 *p [[ buffer(0) ]],
                    texture2d<float4> img [[ texture(0) ]],
                    sampler smp [[ sampler(3) ]],
         ...)
{
   ...
}
```

Note: Samplers that are initialized in the Metal program source must be declared with the `constexpr` qualifier.

A pointer or a reference to a sampler type is not supported and will result in a compilation error.

注意：在Metal程序里初始化的采样器必须使用constexpr修饰符声明。

采样器指针和引用是不支持的，将会导致编译错误。

Arrays and Structs

数组和结构体

Arrays and structs are supported with the following restrictions:

- Arrays of texture and sampler types are not supported.
- The `texture` and `sampler` types cannot be declared in a struct.

- Arguments to graphics and kernel functions cannot be declared to be of type `size_t`, `ptrdiff_t`, or a struct and/or union that contain members declared to be one of these built-in scalar types.
- Members of a struct must belong to the same address space.

数组和结构体要遵守如下的限制：

- 纹理和采样器类型的数组不支持。
- 纹理和采样器类型不能在一个结构体内声明
- 图形渲染函数 和 并行计算函数 的参数不能被声明为size_t, ptrdiff_t。如果参数是结构体（或是联合），且包含了size_t或是,ptrdiff_t类型的成员，也不被支持。
- 结构体中得成员必须属于同一个地址空间。

Alignment and Size of Types

对齐和类型尺寸

Table 2-3 (page 23) lists the alignment and size of the scalar and vector data types.

表2-3 列举了标量和向量数据类型的对齐和尺寸。（貌似都一样）

Table 2-3 Alignment and Size of Scalar and Vector Data Types

Type	Alignment (in bytes)	Size (in bytes)
`bool`	1	1
`char uchar`	1	1
`char2` `uchar2`	2	2
`char3` `uchar3`	4	4 不是3哦
`char4` `uchar4`	4	4
`short` `ushort`	2	2
`short2` `ushort2`	4	4
`short3` `ushort3`	8	8 不是6哦
`short4` `ushort4`	8	8

Type	Alignment (in bytes)	Size (in bytes)
`int uint`	4	4
`int2` `uint2`	8	8
`int3` `uint3`	16	16 不是12哦
`int4` `uint4`	16	16
`half`	2	2
`half2`	4	4
`half3`	8	8 不是6哦

`half4`	8	8
`float`	4	4
`float2`	8	8
`float3`	16	16 不是12哦
`float4`	16	16

Table 2-4 (page 24) lists the alignment and size of the matrix data types.

表2-4列举了矩阵类型的对齐和尺寸。

Table 2-4 Alignment and Size of Matrix Data Types

Type	Alignment (in bytes)	Size (in bytes)
`half2x2`	4	8
`half2x3`	8	16 不是12哦
`half2x4`	8	16
`half3x2`	4	12
`half3x3`	8	24 不是18哦

Type	Alignment (in bytes)	Size (in bytes)
`half3x4`	8	24
`half4x2`	4	16
`half4x3`	8	32 不是24哦
`half4x4`	8	32
`float2x2`	8	16
`float2x3`	16	32 不是24哦
`float2x4`	16	32
`float3x2`	8	24
`float3x3`	16	48 不是36哦
`float3x4`	16	48
`float4x2`	8	32
`float4x3`	16	64 不是48哦
`float4x4`	16	64

Since a matrix is composed of vectors, each column of a matrix has the alignment of its vector component. For example, each column of a `floatnx3` matrix is a `float3` vector that is aligned on a 16-byte boundary, as shown in Table 2-3 (page 23). Similarly, each column of a `halfnx2` matrix is a `half2` vector that is aligned on a 4-byte boundary.

The `alignas` alignment specifier can be used to specify the alignment requirement of a type or an object. The `alignas` specifier may be applied to the declaration of a variable or a data member of a struct or class. It may also be applied to the declaration of a struct, class or enumeration type.

The Metal shading language compiler is responsible for aligning data items to the appropriate alignment as required by the data type. For arguments to a graphics or kernel function declared to be a pointer to a data type, the Metal shading language compiler can assume that the pointer is always appropriately aligned as required by the data type.

一个矩阵是由一组向量构成，一个矩阵的每一列就是一个向量，按照向量的维度对齐。比如，floatnx3的每一列是一个float3向量，那么应用16byte对齐，如表2-3所示。类似的，一个halfnx2矩阵的每一列是一个half2向量，应用4byte对齐。

对齐声明符`alignas`可以被用来设定一种数据类型或是一个对象的对齐要求。对齐声明符`alignas`可以应用在变量声明或是结构体（类）成员声明，Metal着色语言编译器假设指针总是被申明为使用对齐的。

Packed Vector Data Types

紧密填充向量类型

The vector data types described in Vector and Matrix Data Types(page11) are aligned to the size of the vector. There are a number of use cases where developers require their vector data to be tightly packed. For example – a vertex struct that may contain position, normal, tangent vectors and texture coordinates tightly packed and passed as a buffer to a vertex function.

The packed vector type names supported are:

`packed_charn`, `packed_shortn`, `packed_intn`,

`packed_ucharn`, `packed_ushortn`, `packed_uintn`,

`packed_halfn`, and `packed_floatn`

where `n` is 2, 3, or 4 representing a 2-, 3- or 4- component vector type. (The `packed_booln` vector type names are reserved.)

Table 2-5 (page 26) lists the alignment and size of the packed vector data types.

向量数据类型都是按照其尺寸对齐的，但是会出现一些情况，开发者需要将向量数据紧密填充。比如，一个顶点数据结构可能由位置、法向量、正切向量和纹理坐标紧密填充并且以缓存方式传递给一个顶点计算着色函数。

紧密填充向量类型支持如下这些：

packed_charn, packed_shortn, packed_intn,

packed_ucharn, packed_ushortn, packed_uintn,

packed_halfn, packed_floatn。

其中n可以是2，3，或是4,代表2维、3维、4维类型向量。（packed_booln被保留不可用）。

表2-5列举了紧密填充类型向量的对齐和尺寸。

Table 2-5 Alignment and Size of Packed Vector Data Types

Packed Vector Type	Alignment (in bytes)	sizeof (in bytes)
`packed_char2` `packed_uchar2`	1	2
`packed_char3` `packed_uchar3`	1	3 不是4哦
`packed_char4` `packed_uchar4`	1	4
`packed_short2` `packed_ushort2`	2	4
`packed_short3` `packed_ushort3`	2	6 不是8哦
`packed_short4` `packed_ushort4`	2	8
`packed_int2` `packed_uint2`	4	8

Packed Vector Type	Alignment (in bytes)	sizeof (in bytes)
`packed_int3` `packed_uint3`	4	12
`packed_int4` `packed_uint4`	4	16
`packed_half2`	2	4

`packed_half3`	2	6 不是8哦
`packed_half4`	2	8
`packed_float2`	4	8
`packed_float3`	4	12 不是16哦
`packed_float4`	4	16

Packed vector data types are typically used as a data storage format. Loads and stores from a packed vector data type to an aligned vector data type and vice-versa, copy constructor and assignment operator are supported. The arithmetic, logical, and relational operators are also supported for packed vector data types.

紧密填充向量通常用于数据存储格式。载入和存储紧密填充类型向量到一个对齐向量或是反之，都有相应拷贝构造和赋值操作符支持。各算术、逻辑和关系运算符也都支持紧密填充类型向量。

Example:

```
device float4 *buffer;
device packed_float4 *packed_buffer;
int i;
packed_float4 f ( buffer[i] );
pack_buffer[i] = buffer[i];
// operator to convert from packed_float4 to float4.
buffer[i] = float4( packed_buffer[i] );
```

Components of a packed vector data type can be accessed with an array index. However, components of a packed vector data type cannot be accessed with the `.xyzw` or `.rgba` selection syntax.

可以使用数组下标语法来访问紧密填充类型向量的分量，但是不支持使用.xyzw 和 .rgba语法来访问紧密填充类型向量的分量。

Example:

```
packed_float4 f;
f[0] = 1.0f;  // OK
f.x = 1.0f;   // Illegal - compilation error
```

Implicit Type Conversions

隐式类型转换

Implicit conversions between scalar built-in types (except void) are supported. When an implicit conversion is done, it is not just a re-interpretation of the expression's value but a conversion of that value to an equivalent value in the new type. For example, the integer value 5 is converted to the floating-point value 5.0.

在两个内建数据类型标量间的隐式类型转换是支持的。当隐式转换完成，不只是对表达式的值重新解释，还包含了向新类型数值的等效转换。比如整形值5被转换成浮点数5.0.

All vector types are considered to have a higher conversion rank than scalar types. Implicit conversions from a vector type to another vector or scalar type are not permitted and a compilation error results. For example, the following attempt to convert from a 4-component integer vector to a 4-component floating-point vector fails.

所有的向量类型相比标量类型有更高的转换条件。一个向量向另一个向量或是标量的隐式转换是不允许的，会导致编译错误。比如，如下代码试图将一个4维的整形向量转换为4维的浮点向量，将导致编译错误。

```
int4 i; float4 f = i;    // compile error
```

Implicit conversions from scalar-to-vector types are supported. The scalar value is replicated in each element of the vector. The scalar may also be subject to the usual arithmetic conversion to the element type used by the vector or matrix.

从标量到向量的隐式转换是支持的，标量被赋值给向量的每一个分量，而且赋值时也会视分量的数据类型进行数据类型转换。如下列所示。

For example:

```
float4 f = 2.0f;    // f = (2.0f, 2.0f, 2.0f, 2.0f)
```

Implicit conversions from scalar-to-matrix types and vector-to-matrix types are not supported and a compilation error results. Implicit conversions from a matrix type to another matrix, vector or scalar type are not permitted and a compilation error results.

Implicit conversions for pointer types follow the rules described in the *C++11 Specification* .

标量到矩阵，向量到矩阵的隐式转换是不支持的，会导致编译错误。矩阵到矩阵，到向量，到标量的转换都是不允许的，会导致编译错误。

指针类型的隐式转换遵从C++11标准。

Type Conversions and Re-interpreting Data

类型转换和数据重解析

The `static_cast` operator is used to convert from a scalar or vector type to another scalar or vector type with no saturation and with a default rounding mode (i.e., when converting to floating-point, round to zero or round to the nearest even number depending on the rounding mode supported by the GPU; when converting to integer, round toward zero). If the source type is a scalar or vector boolean, the value `false` is converted to zero and the value `true` is converted to one.

static_cast操作符被用于从一个标量或是向量 转换为 另一个标量或是向量，转换将使用不饱和方式 和 默认的舍入模式，（比如转换为浮点型时，，使用向0舍入，还是向最近偶数舍入，是看GPU支持的舍入模式；转换为整数时，向0舍入）。如果源类型是一个布尔标量或是向量，false的值被转换为0，true的值被转换为1。

The Metal shading language adds an `as_type<type-id>` operator to allow any scalar or vector data type (that is not a pointer) to be reinterpreted as another scalar or vector data type of the same size. The bits in the operand are returned directly without modification as the new type. The usual type promotion for function arguments is not performed.

For example, `as_type<float>(0x3f800000)` returns 1.0f, which is the value of the bit pattern `0x3f800000` if viewed as an IEEE-754 single precision value.

It is an error to use the `as_type<type-id>` operator to reinterpret data to a type of a different number of bytes.

Metal着色语言添加了一个as_type<type-id>操作符，它允许任意标量或向量类型（非指针）用来重新解析成另一个同尺寸的标量或是向量。这种转换操作中，源数据的各bit位直接无修改地被返回作为新的数据类型。函数参数的类型升级是不允许的。

举例来说，表达式 as_type<float>(0x3f800000) 返回1.0f。因为 0x3f800000 表示的比特位按照IEEE-754规范解析出来的单精度浮点数就是1.0f。

使用as_type<type-id>操作符来重解析bit位长度不同的数据类型将产生错误。如下例所示：

Examples:

```
float f = 1.0f;
// Legal. Contains: 0x3f800000
uint u = as_type<uint>(f);

// Legal. Contains:
// (int4)(0x3f800000, 0x40000000,
//        0x40400000, 0x40800000)
float4 f = float4(1.0f, 2.0f, 3.0f, 4.0f);
int4 i = as_type<int4>(f);

int i;
// Legal.
short2 j = as_type<short2>(i);

half4 f;
// Error. Result and operand have different sizes
float4 g = as_type<float4>(f);

float4 f;
// Legal. g.xyz will have same values as f.xyz.
// g.w is undefined
float3 g = as_type<float3>(f);
```

Operators

运算符

This chapter lists and describes the Metal shading language operators.

本章节列举并描述Metal着色语言的运算符。

Scalar and Vector Operators

标量和向量运算符

1. The arithmetic operators, add (+), subtract (-), multiply (*) and divide (/), operate on scalar and vector, integer and floating-point data types. All arithmetic operators return a result of the same built-in type (integer or floating-point) as the type of the operands, after operand type conversion. After conversion, the following cases are valid:

    - The two operands are scalars. In this case, the operation is applied, and the result is a scalar.
    - One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a same size vector.
    - The two operands are vectors of the same size. In this case, the operation is performed component-wise, which results in a same size vector.

    Division on integer types that results in a value that lies outside of the range bounded by the maximum and minimum representable values of the integer type, such as TYPE_MIN/-1 for signed integer types, or division by zero does not cause an exception but results in an unspecified value. Division by zero for floating-point types results in ±infinity or NaN, as prescribed by the IEEE-754 standard. (For details about numerical accuracy of floating-point operations, see Numerical Compliance (page 95).)

    双目算术运算符有 加(+)减(-)乘(*)除(/)，可以应用于标量或向量，整形或浮点数。所有的算符运算符返回值类型和参与运算的变量运算变换后的数据类型相同。操作数类型变换后，下面这些例子是合法的：

    - 两个参与算术运算的都是标量，如此，返回值也是标量。
    - 一个运算数是标量，另一个是向量，如此，标量被转换为向量分量的数据类型，接下来标量被延展成一个向量（维度和参与运算的向量相同），然后运算在两个向量的各个分量之间进行，最后返回一个向量。
    - 两个运行数都是相同维度的向量，如此运算在两个向量的各个分量之间进行，最后返回一个向量。

    整形数的除法运算结果如果超出了整形值的范围，比如有符号整形运算 TYPE_MIN/-1 ，或者进行除0运算，都不会导致异常发生，只是结果未定义。浮点数除0运算会导致得到±infinity或是NaN，这种情况将遵从IEEE-745规范描述。

2. The operator modulus (%) operates on scalar and vector integer data types. All arithmetic operators return a result of the same built-in type as the type of the operands, after operand type conversion. The following cases are valid:

    - The two operands are scalars. In this case, the operation is applied, and the result is a scalar.
    - One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a same size vector.
    - The two operands are vectors of the same size. In this case, the operation is performed component-wise, which results in a same size vector.

    The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero operands remain defined. If both operands are non-negative, the remainder is non-negative. If one or both operands are negative, results are undefined.

    取模运算(%)可以操作整形标量和向量。返回值类型和参与运算的变量运算变换后的数据类型相同。操作数类型变换后，下面这些例子是合法的：

- 两个参与取模运算的都是标量，如此，返回值也是标量。
- 一个运算数是标量，另一个是向量，如此，标量被转换为向量分量的数据类型，接下来标量被延展成一个向量（维度和参与运算的向量相同），然后运算在两个向量的各个分量之间进行，最后返回一个向量。
- 两个运行数都是相同维度的向量，如此运算在两个向量的各个分量之间进行，最后返回一个向量。

如果取模运算符后面的数为0，计算结果未定义，对于向量取模运算在每个分量间进行，那些取模运算非0的分量的计算结果还是可以保证的。如果两个运算书都是非负的，运算结果非负，如果1个或是两个数为负，结果未定义。

3. The arithmetic unary operators (+ and -) operate on scalar and vector, integer and floating-point types.

单目算术运算符（+和-）可以操作标量或是向量，整型值或是浮点数值。

4. The arithmetic post- and pre-increment and decrement operators (-- and ++) operate on scalar and vector integer types. All unary operators work component-wise on their operands. The result is the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract 1 to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.

单目前置或是后置的累加递减运算符（-- 和++）可以操作整形的标量和向量。对于向量，单目操作符按分量逐个累加或是递减。返回值和被操作值类型相同。这些单目运算符表达式可以赋值给左值。前置累加递减操作符，累加或是递减被操作数（如果是向量，各分量都被操作），而整个表达式的返回值是操作后的值。后置累加递减操作符，累加或是递减被操作数（如果是向量，各分量都被操作），而整个表达式的返回值是操作前的值。

5. The relational operators greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate on scalar and vector, integer and floating-point types. The result is a Boolean (bool type) scalar or vector. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a bool.
- One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a Boolean vector.
- The two operands are vectors of the same type. In this case, the operation is performed component-wise, which results in a Boolean vector.

The relational operators always return false if either argument is a NaN.

For vector operands, to test whether any or all elements in the result of a vector relational operator test true, use the any or all built-in functions defined in Relational Functions (page 65).

双目关系运算符有，大于（>），小于（<），大等于（>=），小等于（<=）。它们可以操作标量和向量，整形和浮点。运算结果是一个布尔标量，或是布尔向量。操作数类型变换后，下面这些是合法的：

- 两个操作数都是标量，如此，实施比较，返回一个布尔值。
- 一个操作数是标量，另一个操作数是向量，如此标量被转换为向量分量的数据类型，接下来标量被延展成一个向量（维度和参与运算的向量相同），然后比较运算在两个向量的各个分量之间进行，最后返回一个布尔向量。
- 两个运行数都是相同维度的向量，如此比较运算在两个向量的各个分量之间进行，最后返回一个布尔向量。

如果参与运算的操作数有NaN，比较总是返回false。

对于向量操作数，要测试关系运算后得到的布尔向量中，是否有分量为true或是所有分量都为true，可以使用后面“关系函数”章节描述的内建函数。

6. The equality operators, equal (==) and not equal (!=), operate on scalar and vector, integer and floating-point types. All equality operators result in a Boolean (bool type) scalar or vector. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a bool.
- One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, resulting in a Boolean vector.
- The two operands are vectors of the same type. In this case, the operation is performed component-wise resulting in a same size Boolean vector.

All other cases of implicit conversions are illegal. If one or both arguments is “Not a Number” (NaN), the equality operator equal (==) returns false. If one or both arguments is “Not a Number” (NaN), the equality operator not equal (!=) returns true.

双目相等关系运算符有，相等比较（==）,不等比较(!=)，可以操作标量和向量，整形和浮点型。运算符返回一个布尔标量或是向量，操作数类型变换后，下面这些是合法的：

- 两个操作数都是标量，如此，实施相等关系比较，返回一个布尔值。
- 一个操作数是标量，另一个操作数是向量，如此标量被转换为向量分量的数据类型，接下来标量被延展成一个向量（维度和参与运算的向量相同），然后相等关系比较运算在两个向量的各个分量之间进行，最后返回一个布尔向量。
- 两个运行数都是相同维度的向量，如此相等比较运算在两个向量的各个分量之间进行，最后返回一个布尔向量。

所有不在上述的相等比较操作都是不合法的。如果相等比较的两个操作数种有NaN，那么相等比较（==）返回false，不等比较（！=）返回true。

7. The bitwise operators and (&), or (|), exclusive or (^), not (~) operate on all scalar and vector built-in types except the built-in scalar and vector floating-point types. For built-in vector types, the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise resulting in a same size vector.

   位操作运算符有，与运算(&)，或运算(|)，异或运算（^），非运算（~）。可以操作除浮点数外的所有的内建类型标量和向量。对于向量，位运算操作分别操作向量的每一个分量。如果一个操作数是标量，另一个是向量，那么标量被转换类型和向量的分量匹配，接着延展成一个维度和被操作向量相同的向量，然后两个向量的每个分量非别进行位操作。

8. The logical operators and (&&), or (||) operate on two Boolean expressions. The result is a scalar or vector Boolean.

   逻辑运算符，且运算（&&），或运算（||），操作两个布尔表达式，其结果是一个布尔标量或是布尔向量。

9. The logical unary operator not (!) operates on a Boolean expression. The result is a scalar or vector Boolean.

   单目逻辑运算符，非运算（！），操作一个布尔表达式，其结果是一个布尔标量或是布尔向量。

10. The ternary selection operator (?:) operates on three expressions (exp1 ? exp2 : exp3). This operator evaluates the first expression exp1 , which must result in a scalar Boolean. If the result is true, it selects to evaluate the second expression; otherwise it evaluates the third expression. Only one of the second and third expressions is evaluated. The second and third expressions can be any type, as long their types match, or there is a conversion in Implicit Type Conversions (page 28) that can be applied to one of the expressions to make their types match, or one is a vector and the other is a scalar in which case the scalar is widened to the same type as the vector type. This resulting matching type is the type of the entire expression.

    三目运算符(?:)，操作三个表达式（exp1 ? exp2 : exp3）。它求解第一个表达式的值（必须是一个布尔标量），如果为true，它求解并返回第二个表达式的值，如果为false，求解并返回第三个表达式的值。第二个和第三个表达式只会有一个被求解。第二个和第三个表达式可以是任意类型，可以是相同类型；或者被隐式转换使得类型相同；或者一个是向量，一个标量，如此标量被延展成向量；这个类型就是整个表达式的返回值类型。

11. The ones’ complement operator (~). The operand must be of a scalar or vector integer type, and the result is the ones’ complement of its operand.

The operators right-shift (>>), left-shift (<<) operate on all scalar and vector integer types. For built-in vector types, the operators are applied component-wise. For the right-shift (>>), left-shift (<<) operators, if the first operand is a scalar, the rightmost operand must be a scalar. If the first operand is a vector, the rightmost operand can be a vector or scalar.

The result of E1 << E2 is E1 left-shifted by log2(N) least significant bits in E2 viewed as an unsigned integer value, where N is the number of bits used to represent the data type of E1, if E1 is a scalar, or the number of bits used to represent the type of E1 elements, if E1 is a vector. The vacated bits are filled with zeros.

The result of E1 >> E2 is E1 right-shifted by log2(N) least significant bits in E2 viewed as an unsigned integer value, where N is the number of bits used to represent the data type of E1, if E1 is a scalar, or the number of bits used to represent the type of E1 elements, if E1 is a vector. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the vacated bits are filled with zeros. If E1 has a signed type and a negative value, the vacated bits are filled with ones.

单目取反运算（~）,操作数必须是一个整形的标量或是向量，其结果是操作数按二进制取反。

按位右移（>>）按位左移（<<）操作所有的整形标量或是向量，对于向量来说，移位操作在每个分量上进行。如果第一个操作数是标量，那么第二个操作数必须是标量；如果第一个操作数是向量，那第二个操作数可以标量也可以是向量。

E1 << E2的结果是E1被左移，左移的位数由E2的最低有效位（把E2当做一个无符号整形）的log2(N)个比特位确定。N是用于表示E1的数据类型的二进制位的数量，如果E1是向量，数据类型指其分量的类型。左移时多出来的bit用0填充。（比如E1是int，那么N为16，log2(16)=4，那么E2的最低有效位的4个bits确定了左移的位数。4个二进制位的取值是0到15，表示最少左移0位，最多左移15位，这表示是对一个int类型值的有效的左移位操作）。

E1 >> E2的结果是E1被右移，右移的位数由E2的最低有效位（把E2当做一个无符号整形）的log2(N)个比特位确定。N是用于表示E1的数据类型的二进制位的数量，如果E1是向量，数据类型指其分量的类型。如果E1是无符号类型，或是有符号类型且为正数，那么右移时多出来的bit用0填充；如果E1是有符号类型且为负数，那么右移时多出来的bit用1填充。（比如E1是int，那么N为16，log2(16)=4，那么E2的最低有效位的4个bits确定了右移的位数。4个二进制位的取值是0到15，表示最少右移0位，最多右移15位，这表示是对一个int类型值的有效的右移位操作）。

12. The assignment operator behaves as described by the C++11 Specification. For the lvalue = expression assignment operation, if expression is a scalar type and lvalue is a vector type, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a same size vector.

    赋值操作符的行为如C++ 11标准，对于lvalue = expression赋值操作，如果右边的表达式expression结果是一个标量，而左值lvalue是一个向量，那么标量被转换类型和左值向量的分量类型相匹配，接着标量被延展成一个维度和左值向量相同的向量，然后各个分量分别进行赋值操作。

Note: Operators not described above that are supported by C++11(such as `sizeof(T)`, unary(&) operator, and comma (`,`) operator) behave as described in the *C++11 Specification* .

Unsigned integers shall obey the laws of arithmetic modulo 2^n, where `n` is the number of bits in the value representation of that particular size of integer. The result of signed integer overflow is undefined.

For integral operands the divide (`/`) operator yields the algebraic quotient with any fractional part discarded. (This is often called truncation towards zero.) If the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`.

注意：没在在上面提到的又被C++ 11标准支持的操作符的行为如同C++ 11标准描述（比如sizeof(T)，单目取地址符号(&)， 逗号(,)）。

无符号整形应该遵循2^n算术取模法则，n是用于表示整数的二进制位的数量。有符号整形溢出的结果是未定义的。

对于整型被操作数，除操作（/）得到算术商，而舍弃了余数（通常称为0截尾）。如果a/b的商是可被表示的（如果b为0，会得到NaN，这就是不可被表示），那么(a/b)*b = a。

Matrix Operators

矩阵运算符

The arithmetic operators add (+), subtract (-) operate on matrices. Both matrices must have the same numbers of rows and columns. The operation is done component-wise resulting in the same size matrix. The arithmetic operator multiply (*), operates on:

算术运算符，加(+)，减(-)可以操作矩阵，参与运算的两个矩阵要有同样数量的行和列。算术操作在矩阵的每一个元素上进行。乘操作（*）可以在如下的情况进行：

- a scalar and a matrix, 一个标量 * 一个矩阵
- a matrix and a scalar, 一个矩阵 * 一个标量
- a vector and a matrix, 一个向量 * 一个矩阵
- a matrix and a vector, 一个矩阵 * 一个向量
- a matrix and a matrix. 一个矩阵 * 一个矩阵

If one operand is a scalar, the scalar value is multiplied to each component of the matrix resulting in the same size matrix. A right vector operand is treated as a column vector and a left vector operand as a row vector. For vector – matrix, matrix – vector and matrix – matrix multiplication, the number of columns of the left operand is required to be equal to the number of rows of the right operand. The multiply operation does a linear algebraic multiply, yielding a vector or a matrix that has the same number of rows as the left operand and the same number of columns as the right operand.

如果矩阵相乘有一个操作数是标量，那么这个标量和矩阵中的每一个元素相乘，得到一个和矩阵有相同行列的新矩阵。

如果右操作数是一个向量，那么它被看做一个列向量，如果左操作数是一个向量，那么他被看做一个行向量。对于 向量*矩阵，矩阵*向量，还有矩阵*矩阵，左操作数的列数需要和右操作数的行数相等。乘法操作实施的实际上是一个线性代数乘法，结果产生一个向量或是矩阵，它的行数和左操作数的行数相同（如果左操作数是向量，被看做行向量，行数为1，列数为n），列数和右操作数的列数相同（如果，右操作数是向量，被看做列向量，行数为n，列数为1）。比如左操作数矩阵是A列B行的，右操作数矩阵是C列A行的，那么乘操作产生的结果是一个C行B列的矩阵：mLeft(AxB) * mRight(CxA) 将得到 mReturn(CxB)。

The examples below presume these vector, matrix, and scalar variables are initialized:

下面的例子中假设向量，矩阵和标量都是初始化过的:

```
float3 v;
float3x3 m;
float a = 3.0f;
```

The following matrix-to-scalar multiplication如下的 矩阵*标量

```
float3x3 m1 = m * a;
```

is equivalent to: 其计算过程是：

```
m1[0][0] = m[0][0] * a;
m1[0][1] = m[0][1] * a;
m1[0][2] = m[0][2] * a;
m1[1][0] = m[1][0] * a;
m1[1][1] = m[1][1] * a;
m1[1][2] = m[1][2] * a;
m1[2][0] = m[2][0] * a;
m1[2][1] = m[2][1] * a;
m1[2][2] = m[2][2] * a;
```

The following vector-to-matrix multiplication， 如下的 向量*矩阵

```
float3 u = v * m;
```

is equivalent to: 其计算过程是：

```
u.x = dot(v, m[0]);// m[0]表示第0列向量
u.y = dot(v, m[1]);// dot, 向量点乘函数，两个向量各分量分别相乘后，乘积再求和。
u.z = dot(v, m[2]);
```

The following matrix-to-vector multiplication 如下的 矩阵 * 向量

```
float3 u = m * v;
```

is equivalent to: 其计算过程是：

```
u = v.x * m[0];  // u.x = v.x * m[0][0]
u += v.y * m[1]; // u.x += v.y * m[1][0]
u += v.z * m[2]; // u.x += v.z * m[2][0], so u.x = dot(m第一行向量, v)
```

和下面的这个计算过程等价：

```
u.x = dot(float3(m[0][0] m[1][0] m[2][0]), v); // m的第1行点乘v
u.y = dot(float3(m[0][1] m[1][1] m[2][1]), v); // m的第2行点乘v
u.z = dot(float3(m[0][2] m[1][2] m[2][2]), v); // m的第3行点乘v
```

The following matrix-to-matrix multiplication 如下的矩阵 * 矩阵

```
float3x3   m, n, r;
r = m * n;
```

is equivalent to: 其计算过程是：

```
r[0] = m[0] * n[0].x;
r[0] += m[1] * n[0].y;
r[0] += m[2] * n[0].z;
r[1] = m[0] * n[1].x;
r[1] += m[1] * n[1].y;
r[1] += m[2] * n[1].z;
r[2] = m[0] * n[2].x;
r[2] += m[1] * n[2].y;
r[2] += m[2] * n[2].z;
```

Note: The order of partial sums for the vector-to-matrix, matrix-to-vector and matrix-to-matrix multiplication operations described above is undefined.

注意：如上描述的乘法操作中，向量*矩阵，矩阵*向量，矩阵*矩阵，分量求和的顺序是未定义的。

Functions, Variables, and Qualifiers

方法，变量，修饰符

This chapter describes how functions, arguments, and variables are declared. It also details how qualifiers are often used with functions, arguments, and variables to specify restrictions.

本章描述了函数、参数、变量如何定义，还详述了经常和函数、参数、变量一起出现的修饰符如何对它们进行约束。

Function Qualifiers

函数修饰符

The Metal shading language supports the following qualifiers that restrict how a function may be used:

- `kernel` - A data-parallel function that is executed over a 1-, 2- or 3-dimensional grid.
- `vertex` - A vertex function that is executed for each vertex in the vertex stream and generates per-vertex output.
- `fragment` – A fragment function that is executed for each fragment in the fragment stream and their associated data and generates per-fragment output.

A function qualifier is used at the start of a function, before its return type. The following example shows the syntax for a compute function.

Metal着色语言支持下列的函数修饰符：

- `kernel`，表示该函数是一个数据并行计算着色函数，它将被分配在一个1维、2维或是3维的线程组网格中执行。
- `vertex`，表示该函数是一个顶点着色函数，它将为顶点数据流中的每个顶点数据执行一次然后为每个顶点生成数据输出到绘制管线。
- `fragment`，表示该函数是一个片元着色函数，它将为片元数据流中得每个片元和其关联数据执行一次然后为每个片元生成数据输出到绘制管线。

一个函数修饰符出现在函数签名的开始，在函数返回值声明之前，下面的例子展示了并行计算函数修饰符语法：

```
kernel void foo(...)
{
   ...
}
```

For functions declared with the `kernel` qualifier, the return type must be `void`.

Only a graphics function can be declared with one of the `vertex` or `fragment` qualifiers. For graphics functions, the return type identifies whether the output generated by the function is either per-vertex or per-fragment. The return type for a graphics function may be `void` indicating that the function does not generate output.

Functions that use a `kernel`, `vertex` or `fragment` function qualifier cannot call functions that also use these qualifiers, or a compilation error results.

用`kernel`修饰的函数，其返回值类型必须为`void`。

只有图形着色函数才可以被`vertex`或`fragment`修饰。对于图形着色函数，从其返回值类型可以辨认出它是为每顶点做计算的还是为每像素做计算的。图形着色函数的返回值可以为`void`，如此表示该函数不产生数据输出到绘制管线。

一个被函数修饰符修饰的函数中不能再调用其它也被函数修饰符修饰的函数，这样会导致编译错误。

Address Space Qualifiers for Variables and Arguments

用于变量和参数的地址空间修饰符

The Metal shading language implements address space qualifiers to specify the region of memory where a function variable or argument is allocated. These qualifiers describe disjoint address spaces for variables:

- `device` (for more details, see device Address Space (page 37))
- `threadgroup` (see threadgroup Address Space (page 38))
- `constant` (see constant Address Space (page 39))
- `thread` (see thread Address Space (page 39))

All arguments to a graphics (vertex or fragment) or compute function that are a pointer or reference to a type must be declared with an address space qualifier. For graphics functions, an argument that is a pointer or reference to a type must be declared in the `device` or `constant` address space. For kernel functions, an argument that is a pointer or reference to a type must be declared in the `device`, `threadgroup`, or `constant` address space. The following example introduces the use of several address space qualifiers. (The`threadgroup` qualifier is supported here for the pointer `l_data` only if `foo` is called by a kernel function, as detailed in threadgroup Address Space (page 38))

Metal着色语言使用“地址空间修饰符号”来表示一个函数变量或是参数被分配于哪片内存区域。下面这些修饰符描述了不相交叠地址空间：

- `device`
- `threadgroup`
- `constant`
- `thread`

所有的着色函数（vertex、fragment、`kernel`）的参数，如果是指针或是引用都必须带有地址空间修饰符号。对于图形着色函数，其指针或是引用类型的参数必须定义在`device`或是`constant`地址空间。对于并行计算着色函数，其指针或是引用类型的参数必须定义在`device`或是`threadgroup`或是`constant`地址空间，下面的例子展示了如何使用不同的地址空间修饰符（例子中因为The`threadgroup`修饰符修饰了参数指针`l_data`，那么该函数就只能被并行计算着色函数调用）

```
void foo(device int *g_data,
         threadgroup int *l_data,
         constant float *c_data)
{...
}
```

The address space for a variable at program scope must be `constant`.

Any variable that is a pointer or reference must be declared with one of the address space qualifiers discussed in this section. If an address space qualifier is missing on a pointer or reference type declaration, a compilation error occurs.

在程序域（在程序中各函数声明定义区域之外的地方）的变量的地址空间必须是`constant`。

本段中任何指针或是引用类型的变量都需要被一个地址空间修饰符修饰，如果没有将产生编译错误。

device Address Space

设备地址空间

The `device` address space name refers to buffer memory objects allocated from the device memory pool that are both readable and writeable.

A buffer memory object can be declared as a pointer or reference to a scalar, vector or user-defined struct. The actual size of the buffer memory object is determined when the memory object is allocated via appropriate Metal framework API calls in the host code.

Some examples are:

设备地址空间指向从设备内存池分配出来的缓存对象，它是可读也可写的。

一个缓存对象可以被声明成一个标量、向量或是用户自定义结构体的指针或是引用。缓存对象使用的内存实际大小在CPU端的特定Metal框架API被调用时就确定了。

使用device修饰符的例子如下

```
// an array of a float vector with 4 components
device float4 *color;
struct Foo {
   float a[3];
   int b[2];
};
// an array of Foo elements
device Foo *my_info;
```

Since texture objects are always allocated from the device address space, the `device` address qualifier is not needed for texture types. The elements of a texture object cannot be directly accessed. Functions to read from and write to a texture object are provided.

纹理对象总是在设备地址空间分配内存，device地址空间修饰符号不必出现在纹理类型定义中。一个纹理对象的内容无法直接访问，Metal提供了读写纹理对象的函数。

threadgroup Address Space

线程组地址空间

The `threadgroup` address space name is used to allocate variables used by a kernel function that are shared by all theads of a threadgroup. Variables declared in the `threadgroup` address space *cannot* be used in graphics functions.

Variables allocated in the `threadgroup` address space in a kernel function are allocated for each threadgroup executing the kernel and exist only for the lifetime of the threadgroup that is executing the kernel.

The example below shows how variables allocated in the `threadgroup` address space can be passed either as arguments or be declared inside a kernel function. (The qualifier `[[ threadgroup(0) ]]` in the code below is explained in Attribute Qualifiers to Locate Resources (page 40).)

线程组地址空间用于为并行计算着色函数分配内存变量，这些变量被一个线程组的所有的线程共享，在线程组地址空间分配的变量不能被用于图形绘制着色函数。

在并行计算着色函数中，在线程组地址空间分配的变量为一个线程组使用，其声明周期和线程组相同。

下面的例子展示了线程组变量可以作为参数传递或是定义在并行计算着色函数内部。

```
kernel void my_func(threadgroup float *a [[ threadgroup(0) ]], ...)
{
    // A float allocated in threadgroup address space
    threadgroup float x;
    // An array of 10 floats allocated in
    // threadgroup address space
    threadgroup float b[10];
    ...
}
```

constant Address Space

常量地址空间

The `constant` address space name refers to buffer memory objects allocated from the device memory pool but are read-only. Variables in program scope must be declared in the `constant` address space and initialized during the declaration statement. The values used to initialize them must be a compile-time constant. Variables in program scope have the same lifetime as the program, and their values persist between calls to any of the compute or graphics functions in the program.

Pointers or references to the `constant` address space are allowed as arguments to functions.Writing to variables declared in the `constant` address space is a compile-time error. Declaring such a variable without initialization is also a compile-time error.

Note: To decide which address space(`device` or `constant`), a read-only buffer passed to a graphics or kernel function should use, look at how the buffer is accessed inside the graphics or kernel function. The `constant` address space is optimized for multiple instances executing a graphics or kernel function accessing the same location in the buffer. Some examples of this access pattern are accessing light or material properties for lighting / shading, matrix of a matrix array used for skinning, filter weight accessed from a filter weight array for convolution. If multiple executing instances of a graphics or kernel function are accessing the buffer using an index such as the vertex ID, fragment coordinate, or the thread position in grid, then the buffer should be allocated in the `device` address space.

常量地址空间指向的缓存对象也是从设备内存池分配存储，但是它是只读的。在程序域的变量必须定义在常量地址空间并且在声明的时候就初始化。用来初始化的值必须是编译时常量。程序域的变量的生命周期和程序一样，在程序中并行计算着色函数或是图形绘制着色函数会被调用，但是constan修饰的变量的值保持不变。

常量地址空间的指针或是引用可以做函数的参数，向声明为常量的变量赋值会产生编译错误，声明常量但是没有赋予初始值也会产生编译错误。

注意：传递只读的缓存到图形绘制或是并行计算着色函数应该使用哪种地址空间（device 或是 constant）呢？这要看看在着色函数逻辑中缓存是如何被访问的。constant地址空间为多个实例执行图形绘制或是并行计算访问缓存中相同的地址做了优化。它适用的例子有，在光照或是阴影计算中访问灯光或是材质属性，访问蒙皮矩阵数组中的某一个，访问卷积滤波权重数组中的某一个。如果多个实例执行图形绘制或是并行计算访问缓存的时候，需使用索引，比如vertex ID，片元坐标，或是线程组定位参数，那么缓存应该定义在device地址空间中。

thread Address Space

The `thread` address space refers to the per-thread memory address space. Variables allocated in this address space are not visible to other threads. Variables declared inside a graphics or kernel function are allocated in the `thread` address space.

thread地址空间指向每个线程准备的地址空间，在这个线程的地址空间定义的变量在其他线程不可见，在图形绘制或是并行计算着色函数中声明的变量在thread地址空间分配存储。

```
constant float samples[] = { 1.0f, 2.0f, 3.0f, 4.0f };
kernel void my_func(...)
{
   // A float allocated in the per-thread address space
   float x;
   // A pointer to variable x in per-thread address space
   thread float p = &x;
...
}
```

Function Arguments and Variables

函数参数和变量

All inputs and outputs to graphics or kernel functions are passed as arguments (except for initialized variables in the `constant` address space and samplers declared in program scope). Arguments to graphics and kernel functions can be one of the following:

- device buffer – a pointer or reference to any data type in the `device` address space (see Buffers (page 17))
- constant buffer – a pointer or reference to any data type in the `constant` address space (see Buffers (page 17))
- `texture` object (see Textures (page 18))
- `sampler` object (see Samplers (page 20))
- a buffer shared between threads in a threadgroup – a pointer to a type in the `threadgroup` address space. (This buffer can only be used as an argument with kernel functions.)

Buffers (device and constant) specified as argument values to a graphics or kernel function cannot alias; i.e., a buffer passed as an argument value cannot overlap another buffer passed to a separate argument of the same graphics or kernel function.

The arguments to these functions are often specified with attribute qualifiers to provide further guidance on their use. Attribute qualifiers are used to specify:

- the resource location for the argument (see Attribute Qualifiers to Locate Resources (page 40)),
- built-in variables that support communicating data between fixed-function and programmable pipeline stages (see Attribute Qualifiers to Locate Per-Vertex Inputs (page 42)),
- which data is sent down the pipeline from vertex function to fragment function (see stage_in Qualifier (page 50)).

图形绘制或是并行计算着色函数的输入输出都要通过参数传递（除了常量地址空间变量和程序域中定义的采样器以外）。参数可以是下列之一：

- 设备缓存，一个指向设备地址空间的任意数据类型的指针或引用。
- 常量缓存，一个指向常量地址空间的任意数据类型的指针或引用。
- 纹理对象。
- 采样器对象。
- 在线程组中供各线程共享的缓存，一个指向线程组地址空间的任意数据类型的指针或引用。

被设定为着色函数参数的缓存（device和constant）不能重名，比如，一个作为参数的缓存不能和同属于一个着色函数的其他参数重名。着色函数的参数通常还有属性修饰符，它们声明了这参数更多的如何被使用的信息，属性修饰被用来指定：

- 参数表示的资源如何定位。
- 用于在固定管线和可编程着色器之间传递信息的内建的变量。

- 哪些数据沿着绘制管线从顶点着色函数传递到片元着色函数。

Attribute Qualifiers to Locate Buffers, Textures, and Samplers

用于寻址缓存、纹理、采样器的属性修饰符

For each argument, an attribute qualifier must be specified to identify the location of a buffer, texture, or sampler to use for this argument type. The Metal framework API uses this attribute to identify the location for these argument types.

对每个着色函数参数来说，一个修饰符是必须指定的，它用来设定一个缓存、纹理、采样器的位置（在函数参数索引表中的位置），Metal框架API也要使用这个属性和参数类型来定义如何寻址。各种类型的参数的属性修饰符号如下所示。

- device and constant buffers – `[[ buffer(index) ]]`
- texture – `[[ texture (index) ]]`
- sampler – `[[ sampler (index) ]]`
- threadgroup buffer – `[[ threadgroup (index) ]]`

The `index` value is an unsigned integer that identifies the location of a buffer, texture, or sampler argument that is being assigned. The proper syntax is for the attribute qualifier to follow the argument/variable name.

The example below is a simple kernel function, `add_vectors`, that adds an array of two buffers in the device address space, `inA` and `inB`, and returns the result in the buffer `out`. The attribute qualifiers (`buffer(index)`) specify the buffer locations for the function arguments.

index是一个unsigned integer类型的值，它表示了一个缓存、纹理、采样器参数的位置（在函数参数索引表中的位置）。从语法上讲，属性修饰符的声明位置应该位于参数变量名之后。下面的例子中展示了一个简单的并行计算着色函数add_vectors，它把两个设备地址空间中的缓存inA和inB相加，然后把结果写入到缓存out。属性修饰符“(buffer(index))”为着色函数参数设定了缓存的位置。

```
kernel void add_vectors(const device float4 *inA [[ buffer(0) ]],
          const device float4 *inB [[ buffer(1) ]],
          device float4 *out [[ buffer(2) ]],
          uint id [[ thread_position_in_grid ]])
{
    out[id] = inA[id] + inB[id];
}
```

The example below shows attribute qualifiers used for function arguments of several different types (a buffer, a texture, and a sampler):

下面这个例子展示了一个着色函数的多个参数使用不同类型的属性修饰符的情况。

```
kernel void my_kernel(device float4 *p [[ buffer(0) ]],
   texture2d<float> img [[ texture(0) ]],
   sampler sam [[ sampler(1) ]])
{
   ...
}
```

Vertex function example that specifies resources and outputs to device memory

顶点着色函数示例，在设备内存中设置资源和输出

The following example is a vertex function, `render_vertex`, which outputs to device memory in the array `xform_pos_output`, which is a function argument specified with the `device` qualifier (introduced in Function Arguments and Variables (page 40)). All the `render_vertex` function arguments are specified with qualifiers (`buffer(0)`, `buffer(1)`, `buffer(2)`, and `buffer(3)`), as introduced in Attribute Qualifiers to Locate Resources (page 40). (The `position` qualifier shown in this example is discussed in Attribute Qualifiers to Locate Per-Vertex Inputs (page 42).)

下面的例子是一个顶点着色函数render_vertex，它的输出是位于设备内存中数组xform_pos_output，该数组是着色函数的一个参数，使用了device地址修饰符，着色函数的所有参数都带有诸如这样buffer(0), buffer(1), buffer(2), buffer(3)的属性修饰符。

```
#include <metal_stdlib>

using namespace metal;

struct VertexOutput {
```

```
    float4 position [[position]];
    float4 color;
    float2 texcoord;
 };

struct VertexInput {
    float4 position;
    float3 normal;
    float2 texcoord;
};

constexpr constant uint MAX_LIGHTS = 4;
struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};

vertex VertexOutput render_vertex(
        const device VertexInput* v_in [[ buffer(0) ]],
        constant float4x4& mvp_matrix [[ buffer(1) ]],
        constant LightDesc& light_desc [[ buffer(2) ]],
        device float4* xform_pos_output [[ buffer(3) ]],
        uint v_id [[ vertex_id ]]
{
    VertexOutput v_out;
    v_out.position = v_in[v_id].position * mvp_matrix;
    v_out.color = do_lighting(v_in[v_id].position,
                  v_in[v_id].normal,
                  light_desc);
    v_out.texcoord = v_in[v_id].texcoord;

    // output position to a buffer
    xform_pos_output[v_id] = v_out.position;
    return v_out;
 }
```

Attribute Qualifiers to Locate Per-Vertex Inputs

单个顶点输入数据用的属性修饰符

A vertex function can read per-vertex inputs by indexing into a buffer(s) passed as arguments to the vertex function using the vertex and instance IDs. In addition, per-vertex inputs can also be passed as an argument to a vertex function by declaring them with the `[[stage_in]]` attribute qualifier. For per-vertex inputs passed as an argument declared with the `stage_in` qualifier, each element of the per-vertex input must specify the vertex attribute location as

`[[ attribute(index) ]]`

The index value is an unsigned integer that identifies the vertex input location that is being assigned. The proper syntax is for the attribute qualifier to follow the argument/variable name. The Metal framework API uses this attribute to identify the location of the vertex buffer and describe the vertex data such as the buffer to fetch the per-vertex data from, its data format, and stride.

The example below shows how vertex attributes can be assigned to elements of a vertex input struct passed to a vertex function using the `stage_in` qualifier.

一个顶点着色函数可以读取单个顶点的输入数据，这些输入数据存放于参数传递的缓存中，使用顶点和实例ID在这些缓存中寻址，读取到单个顶点的数据。另外，单个顶点输入数据还可通过使用了“`[[stage_in]]`”属性修饰符的参数传递给顶点着色函数，在这种情况下，单顶点输入数据中的每个元素必须用“`[[ attribute(index) ]]`”指定顶点数据元素位置，其中index是一个unsigned integer类型的值，它指出了顶点输入数据的位置，语法上，它出现在参数变量名的后面。Metal框架API也要使用这个属性定义顶点缓存的位置以及顶点数据（比如从何处获取单个顶点数据，数据格式，单顶点数据跨度）。

下面的例子展示了使用“`stage_in`”修饰符修饰了传入顶点着色函数入参时，顶点数据输入结构该如何定义。

```
#include <metal_stdlib>
using namespace metal;
struct VertexInput {
   float4 position [[ attribute(0) ]];
   float3 normal   [[ attribute(1) ]];
   half4 color     [[ attribute(2) ]];
```

```
    half2 texcoord  [[ attribute(3) ]];
};
constexpr constant uint MAX_LIGHTS = 4;
struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};
constexpr sampler s = sampler(coord::normalized,
                          address::clamp_to_zero,
                          filter::linear);
vertex VertexOutput
render_vertex(VertexInput v_in [[ stage_in ]],
            constant float4x4& mvp_matrix [[ buffer(1) ]],
            constant LightDesc& lights [[ buffer(2) ]],
            uint v_id [[ vertex_id ]])
{
    VertexOutput v_out;
    ...
    return v_out;
}
```

The example below shows how both buffers and the `stage_in` qualifier can be used to fetch per-vertex inputs in a vertex function.

下面这个例子展示buffer(index)和`stage_in` 两种属性修饰符在同一个顶点着色函数中一起用于定位获取单个顶点输入数据的情况。

```
#include <metal_stdlib>
using namespace metal;
  struct VertexInput {
      float4 position [[ attribute(0) ]];
      float3 normal   [[ attribute(1) ]];
};
  struct VertexInput2 {
      half4 color;
      half2 texcoord[4];
  };
  constexpr constant uint MAX_LIGHTS = 4;
  struct LightDesc {
      uint   num_lights;
      float4 light_position[MAX_LIGHTS];
      float4 light_color[MAX_LIGHTS];
      float4 light_attenuation_factors[MAX_LIGHTS];
};
  constexpr sampler s = sampler(coord::normalized,
                            address::clamp_to_zero,
                            filter::linear);
  vertex VertexOutput
  render_vertex(VertexInput v_in [[ stage_in ]],
              VertexInput2 v_in2 [[ buffer(0) ]],
              constant float4x4& mvp_matrix [[ buffer(1) ]],
              constant LightDesc& lights [[ buffer(2) ]],
              uint v_id [[ vertex_id ]])
  {
      VertexOutput vOut;
      ...
      return vOut;
}
```

Attribute Qualifiers for Built-in Variables

内建变量用的属性修饰符

Some graphics operations occur in the fixed-function pipeline stages and need to provide values to or receive values from graphics functions. Built-in input and output variables are used to communicate values between the graphics (vertex and fragment) functions and the fixed-function graphics pipeline stages. Attribute qualifiers are used with arguments and the return type of graphics functions to identify these built-in variables.

有一些图形绘制操作发生在图形绘制管线的固定功能阶段，这些操作需要为图形着色函数提供数据或是从图形着色函数获取数据。内建输入输出变量就是用来在整个图形绘制管线中连通图形着色函数和固定功能部分用的。有一些属性修饰符用于着色函数参数和返回值类型，它们可以标记这些内建变量。

Attribute Qualifiers for Vertex Function Input

用于顶点着色函数输入的内建变量的属性修饰符

Table 4-1 (page 45) lists the built-in attribute qualifiers that can be specified for arguments to a vertex function and the corresponding data types with which they can be used.

表4-1 列出了一个顶点着色函数参数用的标示内建变量的属性修饰符，以及这些修饰符对应的可用的数据类型。

Table 4-1 Attribute Qualifiers for Vertex Function Input Arguments

Attribute Qualifier	Corresponding Data Types
`[[vertex_id]]`	`ushort` or `uint`
`[[instance_id]]`	`ushort` or `uint`

Attribute Qualifiers for Vertex Function Output

用于顶点着色函数输出的内建变量的属性修饰符

Table 4-2 (page 45) lists the built-in attribute qualifiers that can be specified for a return type of a vertex function or the members of a struct that are returned by a vertex function (and the corresponding data types with which they can be used).

表4-2列出了一个顶点着色函数可以用来标示返回值（或是返回数据结构中的某些成员）为输出用的内建变量的属性修饰符，以及这些修饰符对应的可用的数据类型。

Table 4-2 Attribute Qualifiers for Vertex Function Return Types

Attribute Qualifier	Corresponding Data Types
`[[clip_distance]]`	`float` or `float[n]n` must be known at compile time
`[[point_size]]`	`float`
`[[position]]`	`float4`

The example below describes a vertex function called `process_vertex`. The function returns a user-defined struct called `VertexOutput`, which contains a built-in variable that represents the vertex position, so it requires the `[[position]]` qualifier.

下面这个例子展示了一个顶点着色函数process_vertex，它返回一个名叫VertexOutput的自定义的结构体，这个结构体中包含了内建变量用于标示一个顶点的位置，所以它需要使用“[[position]]”属性修饰符。

```
struct VertexOutput {
    float4 position [[position]];
    float4 color;
    float2 texcoord;
};
vertex VertexOutput process_vertex(...)
{
   VertexOutput v_out;
   // compute per-vertex output
   ...
   return v_out;
}
```

Attribute Qualifiers for Fragment Function Input

片元着色函数输入用的内建属性修饰符

Table 4-3 (page 46) lists the built-in attribute qualifiers that can be specified for arguments of a fragment function (and their corresponding data types).

Note: If the return type of a vertex function is not `void`, it must include the vertex position. If the vertex return type is `float4` this always refers to the vertex position (and the `[[ position ]]` qualifier need not be specified). If the vertex return type is a struct, it must include an element declared with the `[[ position ]]` qualifier.

表4-3列出了用于片元着色函数入参的内建属性修饰符。

注意：如果一个顶点着色函数被声明成非voide的，它必须包含顶点位置，如果它的返回值是float4，这通常表示位置，如此“[[position]]”修饰符不是必要的。如果一个顶点着色函数的返回值类型是一个结构体，那么结构体必须包含一个被“[[position]]”修饰的成员。

Table 4-3 Attribute Qualifiers for Fragment Function Input Arguments

Attribute Qualifier	Corresponding Data Types	Description
`[[color(m)]]`	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn` m必须是编译时常量	The input value read from a color attachment. The index `m` indicates which color attachment to read from. 表示输入值从一个颜色attachment中读取，m用于指定从哪个颜色attachment中读取。
`[[front_facing]]`	`bool`	This value is `true` if the fragment belongs to a front-facing primitive. 如果片元属于是一个正面片元这个值为true
`[[point_coord]]`	`float2`	Two-dimensional coordinates indicating where within a point primitive the current fragment is located. They range from 0.0 to 1.0 across the point. 这是一个二维坐标，表示在一个点图元中当前片元位于何处，取值范围是0.0到1.0
`[[position]]`	`float4`	Describes the window relative coordinate (x, y, z, 1/w) values for the fragment. 描述了片元的窗口相对坐标（x, y, z, 1/w）
`[[sample_id]]`	`uint`	The sample number of the sample currently being processed. 当前正在被处理的样本的采样数。（这TM是个啥）
`[[sample_mask]]`	`uint`	The set of samples covered by the primitive generating the fragment during multisample rasterization. 图元在多重采样光栅化中产生的片元所覆盖的样本的集合。（这TM是个啥）

A variable declared with the `[[ position ]]` attribute as input to a fragment function can only be declared with the `center_no_perspective` sampling and interpolation qualifier.

For `[[color(m)]]`, `m` is used to specify the color attachment index when accessing (reading or writing) multiple color attachments in a fragment function.

一个被“[[position]]”属性修饰的变量作为片元着色函数的输入，只能被声明为 `center_no_perspective` 采样插值属性符。

对于“[[color(m)]]”，m用于在片元着色函数中访问（读写）多个颜色attachment的时候，指定索引下标。

Attribute Qualifiers for Fragment Function Output

片元着色函数输出用的内建属性修饰符

The return type of a fragment function describes the per-fragment output. A fragment function can output one or more render-target color values, a depth value, and a coverage mask, which must be identified by using the attribute qualifiers listed in Table 4-4 (page 47). If the depth value is not output by the fragment function, the depth value generated by the rasterizer is output to the depth attachment.

一个片元着色函数的返回值类型描述了单个片元的输出，一个片元着色函数可以输出一个或是多个渲染结果颜色值，一个深度值，还有一个覆盖遮罩，如表4-4所列出的属性修饰符。如果片元着色函数不输出深度值，那么在光栅化阶段产生的深度值将输出到深度attachment。

Table 4-4 Attribute Qualifiers for Fragment Function Return Types

Attribute Qualifier	Corresponding Data Types
`[[color(m)]]`	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn` m必须在编译时就已确定
`[[depth(depth_qualifier)]]`	`float`
`[[sample_mask]]`	`uint`

The color attachment index `m` for fragment output is specified in the same way as it is for `[[color(m)]]` for fragment input (see discussion for Table 4-3 (page 46)).

If there is only a single color attachment in a fragment function, then `[[color(m)]]` is optional. If `[[color(m)]]` is not specified, the attachment index will be 0. If multiple color attachments are specified, `[[color(m)]]` must be specified for all color values. See examples of specifying the color attachment in Per-Fragment Function vs. Per-Sample Function (page 54) and Programmable Blending (page 55).

If a fragment function writes a depth value, the `depth_qualifier` must be specified with one of the following values: `any`, `greater`, or `less`.

The following example shows how color attachment indices can be specified. Color values written in `clr_f` write to color attachment index 0, `clr_i` to color attachment index 1, and `clr_ui` to color attachment index 2.

表4-4列出的片元着色函数输出用的属性修饰符，其中颜色attachment索引下标m和表4-3中的含义一样。

如果在一个片元着色函数中仅有一个颜色attachment，那么“[[color(m)]]”不是必须出现的。如果“[[color(m)]]”没有设定，那么attachment的下标将是0。如果多个attachment被指定，“[[color(m)]]”必须为所有的颜色值设置。后面的章节有这样的例子。

如果一个片元着色函数要输出一个深度值，depth_qualifier必须被设定为这些值中的一个：any, greater, less

下面的例子展示了一个颜色attachment如何设定，被写入`clr_f`的颜色值输出到下标为0的颜色attachment，`clr_i`中的颜色值输出到下标为1的颜色attachment，`clr_ui`中的颜色值输出到下标为2的颜色attachment。

```
struct MyFragmentOutput {
    // color attachment 0
    float4 clr_f [[color(0)]];
    // color attachment 1
    int4 clr_i [[color(1)]];
    // color attachment 2
   uint4 clr_ui [[color(2)]];
};

fragment MyFragmentOutput my_frag_shader( ... )
{
   MyFragmentOutput f;
   ....
   f.clr_f = ...;
   ...
    return f;
}
```

Note: If a color attachment index is used both as an input to and output of a fragment function, the data types associated with the input argument and output declared with this color attachment index must match.

注意：如果一个颜色attachment下标被用于一个片元着色函数的输入和输出，那么这个attachment对应的输入输出的数据类型要匹配。

Attribute Qualifiers for Kernel Function Input

并行计算着色函数输入用的属性修饰符

When a kernel is submitted for execution, it executes over an N-dimensional grid of threads, where N is one, two or three. An instance of the kernel executes for each point in this grid. A thread is an instance of the kernel that executes for each point in this grid, and `thread_position_in_grid` identifies its position in the grid.

Threads are organized into **threadgroups**. Threads in a threadgroup cooperate by sharing data through `threadgroup` memory and by synchronizing their execution to coordinate memory accesses to both `device` and `threadgroup` memory. The threads in a given threadgroup execute concurrently on a single compute unit on the GPU. (A GPU may have multiple compute units. Multiple threadgroups can execute concurrently across multiple compute units.) Within a compute unit, a threadgroup is partitioned into multiple smaller groups for execution. The execution width of the compute unit, referred to as the `thread_execution_width`, determines the recommended size of this smaller group. For best performance, the total number of threads in the threadgroup should be a multiple of the `thread_execution_width`.

并行计算着色函数被提交执行的时候，函数实际运行在N维线程网格中，N可以是1、2、3。着色函数的实例将运行在线程网格中的每个点上，一个线程就是一个运行在网格上某个节点的着色函数实例，thread_position_in_grid用于标识当前节点在网格中的位置。

线程被组织在线程组中，在线程组中的各线程通过共享线程组内存数据以及同步执行协调对设备和线程组存储的访问。一个线程组的线程同时在一个GPU的计算单元执行（一个GPU可以有多个计算单元，多个线程组可以同时在多个计算单元执行）。在一个计算单元，一个线程组被分成多个更小的组执行。计算单元的执行宽度由`thread_execution_width`表示，它表示一个线程小组的推荐尺寸大小。为了获得最好性能，线程组中的线程数量应该是`thread_execution_width`的整数倍。

Threadgroups are assigned a unique position within the grid (referred to as `threadgroup_position_in_grid`) with the same dimensionality as the index space used for the threads. Threads are assigned a unique position within a threadgroup (referred to as `thread_position_in_threadgroup`). The unique scalar index of a thread within a threadgroup is given by `thread_index_in_threadgroup`.

Each thread's position in the grid and position in the threadgroup are N-dimensional tuples. Threadgroups are assigned a position using a similar approach to that used for threads. Threads are assigned to a threadgroup and given a position in the threadgroup with components in the range from zero to the size of the threadgroup in that dimension minus one.

When a kernel is submitted for execution, the number of threadgroups and the threadgroup size are specified. For example, consider a kernel submitted for execution that uses a 2-dimensional grid where the number of threadgroups specified are (W_x, W_y) and the threadgroup size is (S_x, S_y). Let (w_x, w_y) be the position of each threadgroup in the grid (i.e., `threadgroup_position_in_grid`), and (l_x, l_y) be the position of each thread in the threadgroup (i.e., `thread_position_in_threadgroup`).

The thread position in the grid (i.e., `thread_position_in_grid`) is:

$(g_x,g_y)=(w_x *S_x +l_x,w_y *S_y +l_y)$

The grid size (i.e., `threads_per_grid`) is:

$(G_x, G_y) = (W_x * S_x , W_y * S_y)$

The thread index in the threadgroup (i.e., `thread_index_in_threadgroup`) is:

$l_y *S_x +l_x$

在网格中，线程组赋予每个线程一个唯一的位置（由threadgroup_position_in_grid指定，它是一个维度和线程组一致向量，各分量表示网格下标）。线程组中的线程还有唯一的下标，由thread_index_in_threadgroup表示。

每个线程在线程网格中的位置一个N维向量，相似的，线程组也被赋予一个位置。线程组中的线程的位置向量，其每个维度的取值范围是0到线程组在该维度上的长度减1.

当一个并行计算着色函数被提交执行，线程组的数量和尺寸被设定，比如，有一个着色函数执行时，使用一个2维线程组网格，其线程组的数量设定为(W_x, W_y)，其中每个线程组的尺寸是(S_x, S_y)。用(w_x, w_y)表示每个线程组在网格中的位置（由threadgroup_position_in_grid表示），用(l_x, l_y)表示某线程在某个线程组中的位置（由thread_position_in_threadgroup表示）。那么这个线程在大网格中的位置（由thread_position_in_grid表示）的计算公式如下：

$(g_x,g_y)=(w_x *S_x +l_x,w_y *S_y +l_y)$

大网格的尺寸计算公式如下（由threads_per_grid表示）

$(G_x, G_y) = (W_x * S_x , W_y * S_y)$

线程在线程组中的下标（由thread_index_in_threadgroup表示）为

$l_y * S_x + l_x$

Table 4-5 (page 49) lists the built-in attribute qualifiers that can be specified for arguments to a compute function and the corresponding data types with which they can be used.

表4-5列出了用于并行计算着色函数参数的内建的属性修饰符以及对应的可用的数据类型。

Table 4-5 Attribute Qualifiers for Kernel Function Input Arguments

Attribute Qualifier	Corresponding Data Types
`[[thread_position_in_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[thread_position_in_threadgroup]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[thread_index_in_threadgroup]]`	`uint`, `ushort`
`[[threadgroup_position_in_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[threads_per_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[threads_per_threadgroup]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[threadgroups_per_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[thread_execution_width]]`	`ushort` or `uint`

Notes on kernel function attribute qualifiers:

- The type used to declare `[[thread_position_in_grid]]`, `[[threads_per_grid]]`, `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`, `[[threadgroup_position_in_grid]]` and `[[threadgroups_per_grid]]` must be a scalar type or a vector type. If it is a vector type, the number of components for the vector types used to declare these arguments must match.
- The data types used to declare `[[thread_position_in_grid]` and `[[threads_per_grid]]` must match.
- The data types used to declare `[[thread_position_in_threadgroup]]` and `[[threads_per_threadgroup]]` must match.
- If `thread_position_in_threadgroup` is declared to be of type `uint`, `uint2`, or `uint3`, then `[[thread_index_in_threadgroup]]` must be declared to be of type `uint`.

并行计算着色函数的属性修饰符的注意事项：

- [[thread_position_in_grid]], [[threads_per_grid]], [[thread_position_in_threadgroup]], [[threads_per_threadgroup]], [[threadgroup_position_in_grid]]，[[threadgroups_per_grid]]的声明类型必须是一个标量或是一个向量。如果是一个向量，向量的维度必须和被声明的参数相符合。
- [[thread_position_in_grid]和[[threads_per_grid]]在声明时数据类型必须相符合。
- [[thread_position_in_threadgroup]和[[threads_per_threadgroup]]在声明时数据类型必须相符合。
- 如果[[thread_position_in_threadgroup]]被声明为uint, uint2, uint3中的一个，那么[[thread_index_in_threadgroup]]必须声明为uint类型

stage_in Qualifier

stage_in修饰符号

The per-fragment inputs to a fragment function are generated using the output from a vertex function and the fragments generated by the rasterizer. The per-fragment inputs are identified using the `[[stage_in]]` attribute qualifier.

A vertex function can read per-vertex inputs by indexing into a buffer(s) passed as arguments to the vertex function using the vertex and instance IDs. In addition, per-vertex inputs can also be passed as arguments to a vertex function by declaring them with the `[[stage_in]]` attribute qualifier.

Only one argument of the fragment or vertex function can be declared with the stage_in qualifier. For a user-defined struct declared with the `stage_in` qualifier, the members of the struct can be:

a scalar integer or floating-point value or a vector of integer or floating-point values.

Note: Packed vectors, matrices, structs, references or pointers to a type, and arrays of scalars, vectors, and matrices are not supported as members of the struct declared with the `stage_in` qualifier.

片元着色函数使用的单个片元输入数据是由顶点着色函数输出然后经过光栅化生成的。单个片元输入数据可以使用“`[[stage_in]]`”属性修饰符标识。

一个顶点着色函数可以读取单个顶点的输入数据，这些输入数据存放于参数传递的缓存中，使用顶点和实例ID在这些缓存中寻址，读取到单个顶点的数据。另外，单个顶点输入数据还可通过使用了“`[[stage_in]]`”属性修饰符的参数传递给顶点着色函数。

顶点和片元着色函数都是只能有一个参数被声明为使用“stage_in”修饰符，对于一个使用了“stage_in”修饰符的自定义的结构体，其成员可以为一个整形或浮点标量，或是整形或浮点向量。

注意：被stage_in修饰的结构体的成员不能是如下这些：Packed vectors紧密填充类型向量, matrices矩阵, structs结构体, references or pointers to a type某类型的引用或是指针, and arrays of scalars, vectors, and matrices标量、向量、矩阵数组。

Vertex function example that uses the stage_in qualifier

使用stage_in修饰符的顶点着色函数示例

The following example shows how to pass per-vertex inputs using the `stage_in` qualifier.

下面这个例子展示了如何使用stage_in修饰符传递单个顶点输入数据。

```
#include <metal_stdlib>

using namespace metal;

struct VertexOutput {
   float4 position [[position]];
   float4 color;
   float2 texcoord[4];
};

struct VertexInput {
   float4 position [[ attribute(0) ]];
   float3 normal   [[ attribute(1) ]];
   half4 color     [[ attribute(2) ]];
   half2 texcoord  [[ attribute(3) ]];
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
     uint   num_lights;
     float4 light_position[MAX_LIGHTS];
     float4 light-colored[MAX_LIGHTS];
     float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
                              address::clamp_to_zero,
                              filter::linear);

vertex VertexOutput render_vertex(
      VertexInput v_in [[ stage_in ]],
      constant float4x4& mvp_matrix [[ buffer(1) ]],
      constant LightDesc& lights [[ buffer(2) ]],
      uint v_id [[ vertex_id ]])
{
   VertexOutput v_out;
   v_out.position = v_in.position * mvp_matrix;
   v_out.color = do_lighting(v_in.position, v_in.normal, lights);

   ...
    return v_out;
}
```

Fragment function example that uses the stage_in qualifier

使用stage_in修饰符的片元着色函数示例

An example in Attribute Qualifiers to Locate Per-Vertex Inputs (page 42) previously introduces the `process_vertex` vertex function, which returns a `VertexOutput` struct per vertex. In the following example, the output from `process_vertex` is pipelined to become input for a fragment function called `render_pixel`, so the first argument of the fragment function uses the `[[stage_in]]` qualifier and the incoming `VertexOutput` type. (In `render_pixel`, the `imgA` and `imgB` 2D textures call the built-in function `sample`, which is introduced in 2D Texture (page 76).)

上一小节已经展示了顶点着色程序`process_vertex`，它为每个顶点返回一个结构体`VertexOutput`，下面的例子中，`process_vertex`的输出经过绘制管线变成了片元着色函数`render_pixel`的输入。所以片元着色函数的第一个参数使用“`[[stage_in]]`”属性修饰符以及`VertexOutput`数据类型。（片元着色函数`render_pixel`中，二维纹理`imgA`和`imgB`调用内建函数sample，将在后面的章节讲述）

```
#include <metal_stdlib>

using namespace metal;

struct VertexOutput {
     float4 position [[position]];
     float4 color;
     float2 texcoord;
};

struct VertexInput {
     float4 position;
     float3 normal;
     float2 texcoord;
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
     uint   num_lights;
     float4 light_position[MAX_LIGHTS];
     float4 light_color[MAX_LIGHTS];
     float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
                              address::clamp_to_border,
                              filter::linear);

vertex VertexOutput render_vertex(
      const device VertexInput* v_in [[ buffer(0) ]],
      constant float4x4& mvp_matrix [[ buffer(1) ]],
      constant LightDesc& lights [[ buffer(2) ]],
      uint v_id [[ vertex_id ]])
{
   VertexOutput v_out;
   v_out.position = v_in[v_id].position * mvp_matrix;
   v_out.color = do_lighting(v_in[v_id].position,
                             v_in[v_id].normal,
                             lights);
    v_out.texcoord = v_in[v_id].texcoord;
    return v_out;
}

fragment float4 render_pixel(
      VertexOutput input [[stage_in]],
      texture2d<float> imgA [[ texture(0) ]],
      texture2d<float> imgB [[ texture(1) ]])
{
   float4 tex_clr0 = imgA.sample(s, input.texcoord);
   float4 tex_clr1 = imgB.sample(s, input.texcoord);

// compute color
   float4 clr = compute_color(tex_clr0, tex_clr1, ...);
   return clr;
}
```

Storage Class Specifiers

存储限定符

The Metal shading language supports the `static` and `extern` storage class specifiers. Metal does not support the `thread_local` storage class specifiers.

The `extern` storage-class specifier can only be used for functions and variables declared in program scope or variables declared inside a function. The `static` storage-class specifier is only for variables declared in program scope (see constant Address Space (page 39)) and is not for variables declared inside a graphics or kernel function. In the following example, the `static` specifier is incorrectly used by the variables `b` and `c` declared inside a kernel function.

Metal着色语言支持`static`和`extern`存储限定符。Metal不支持`thread_local`存储限定符。

`extern`存储限定符用于修饰在程序域中的函数和变量或是在函数内声明的变量。

`static`存储限定符仅用于修饰在程序域中的变量，不能用于函数内部的变量，在下面的例子中，`static`存储限定符被用于一个并行计算着色函数内部声明的变量b和c，这是不正确的。

```
#include <metal_stdlib>

using namespace metal;

extern constant float4 noise_table[256];
static constant float4 color_table[256] = { ... }; // static is okay
extern void my_foo(texture2d<float> img);
extern void my_bar(device float *a);

kernel void my_func(texture2d<float> img [[ texture(0) ]],
                    device float *ptr [[ buffer(0) ]])
{
    extern constant float4 a;
    static constant float4 b; // static is an error.
    static float c;           // static is an error.
    ...
    my_foo(img);
    ...
    my_bar(ptr);
    ...
}
```

Sampling and Interpolation Qualifiers

采样和插值修饰符

Sampling and interpolation qualifiers are used with inputs to fragment functions declared with the `stage_in` qualifier. The qualifier determines what sampling method the fragment function uses and how the interpolation is performed, including whether to use perspective-correct interpolation, linear interpolation, or no interpolation.

The sampling and interpolation qualifier can be specified on any structure member declared with the `stage_in` qualifier. The sampling and interpolation qualifiers supported are:

采样和插值修饰符用于被stage_in修饰符片元着色函数的输入数据，这个修饰符确定了片元着色函数使用哪种采样方法以及插值如何实施，包括是否使用透视校正插值，线性插值还是不插值。

被stage_in修饰符声明的结构体的成员可以使用采样和插值修饰符，可用的修饰符如下：

- center_perspective
- center_no_perspective
- centroid_perspective
- centroid_no_perspective
- sample_perspective
- sample_no_perspective
- flat

`center_perspective` is the default sampling and interpolation qualifier for all attribute qualfiiers for return types of vertex functions and arguments to fragment functions, except for `[[position]]`, which can only be declared with `center_no_perspective`.

The following example is user-defined struct that specifies how data in certain members are interpolated:

对于顶点着色程序的输出用的各属性修饰符 以及 片元程序的输入用的各属性修饰符来说，center_perspective是默认的采样和插值修饰符，但是“`[[position]]`”例外，它使用center_no_perspective。

下面的例子是一个自定义的结构，指定了某些成员如何插值：

```
struct FragmentInput {
       float4 pos [[center_no_perspective]];
       float4 color [[center_perspective]];
       float2 texcoord;
       int index [[flat]];
       float f [[sample_perspective]];
};
```

For integer types, the only valid interpolation qualifier is `flat`.

The sampling qualifier variants (`sample_perspective` and `sample_no_perspective`) interpolate at a sample location rather than at the pixel center. With one of these qualifiers, the fragment function or code blocks in the fragment function that use these variables execute per-sample rather than per-fragment.

对于整形值，唯一有效的插值修饰符是 flat

采样修饰符变量（由`sample_perspective` 和 `sample_no_perspective`修饰的）在采样位置做插值而不在像素中心做插值。使用了某个采样修饰符号的情况下，片元着色函数或者其中着色代码块按照采样计算模式而不是片元计算模式执行。

Per-Fragment Function vs. Per-Sample Function

片元计算模式 对比 采样计算模式

The fragment function is typically executed per-fragment. The sampling qualifier identifies if any fragment input is to be interpolated at per-sample vs. per-fragment. Similarly, the `[[sample_id]]` attribute is used to identify the current sample index and the `[[color(m)]]` attribute is used to identify the destination fragment color or sample color (for a multisampled color attachment) value. If any of these qualifiers are used with arguments to a fragment function, the fragment function may execute per-sample instead of per-pixel. (The implementation may decide to only execute the code per-sample that depends on the per-sample values, and the rest of the fragment function may execute per-fragment.)

Only the inputs with sample specified (or declared with the `[[sample_id]]` or `[[color(m)]]` qualifier) differ between invocations per-fragment or per-sample, whereas other inputs still interpolate at the pixel center.

The following example uses the `[[color(m)]]` attribute to specify that this fragment function should be executed on a per-sample basis.

片元着色函数通常是按照片元计算模式执行的。采样修饰符可以用来决定片元着色函数的输入插值是按采样（per-sample）还是按片元（per-fragment）。类似的，“[[sample_id]]”修饰符用来表示当前采样索引，“[[color(m)]]”修饰符用来表示目标片元颜色值或是采样颜色值（对于一个多重采样颜色attachment）。如果以上任何一个修饰符被用于片元着色函数参数，片元着色函数按采样计算模式（per-sample）执行而不是片元计算模式（per-pixel）（根据采样值，Metal可以决定以采样模式执行代码，其他的片元着色函数按片元计算模式执行代码）。

只有带特定采样的输入（或是由`[[sample_id]]`、`[[color(m)]]`修饰符的），片元计算模式，采样计算模式会出现差别。而其他的输入还是在像素中心插值计算。

下面的例子中，使用了“`[[color(m)]]`”属性修饰符，表示这个片元着色函数按照采样计算模式执行。

```
#include <metal_stdlib>

using namespace metal;

fragment float4 my_frag_shader(
               float2 tex_coord [[ stage_in ]],
                texture2d<float> img [[ texture(0) ]],
                sampler s [[ sampler(0) ]],
                float4 framebuffer [[color(0)]])
```

```
{
   return c = mix(img.sample(s, tex_coord), framebuffer, mix_factor);
}
```

Programmable Blending

可编程的混合操作

The fragment function can be used to perform per-fragment or per-sample programmable blending. The color attachment index identified by the `[[color(m)]]` attribute qualifier can be specified as an argument to a fragment function.

Below is the programmable blending example from Allan Schaffer's *Advances in OpenGL and OpenGL ES* talk at WWDC 2012 that describes how to paint grayscale onto what is below. (Look for the talk and slide set among the WWDC 2012 talks at https://developer.apple.com/videos/wwdc/2012/)

片元着色函数可以用来实现按像素或是按采样的可编程混合。通过“`[[color(m)]]`”属性修饰符指定的颜色attachments索引可以被设定为片元着色函数的参数。

下面的可编程混合例子来自于WWDC2012，Allan Schaffer宣讲的《*Advances in OpenGL and OpenGL ES*》，它描述了如何绘制灰阶。

The GLSL version is:

GLSL语言实现的例子如下：

```
#extension GL_APPLE_shader_framebuffer_fetch : require


void main() {

    // RGB to grayscale
    mediump float lum = dot(gl_LastFragData[0].rgb,
                        vec3(0.30,0.59,0.11));
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

The equivalent Metal function is:

对等的Metal着色函数代码如下：

```
#include <metal_stdlib>

using namespace metal;

fragment half4 paint_grayscale(half4 dst_color [[color(0)]])
{
    // RGB to grayscale
    half lum = dot(dst_color.rgb,
                   half3(0.30h, 0.59h, 0.11h));

    return half4(lum, lum, lum, 1.0h);
}
```

Graphics Function – Signature Matching

图形绘制着色函数——签名匹配

A graphics function signature is a list of parameters that are either input to or output from a graphics function.

一个图形着色函数的签名是一个由函数的输入和输出组成的参数列表，

Vertex – Fragment Signature Matching

顶点—片元 着色函数签名匹配

The per-instance input to a fragment function is declared with the `[[stage_in]]` qualifier. These are output by an associated vertex function.

Built-in variables are declared with one of the attribute qualifiers defined in Attribute Qualifiers to Locate Per-Vertex Inputs (page 42). These are either generated by a vertex function (such as `[[position]]`, `[[point_size]]`, `[[clip_distance]]`), are generated by the rasterizer (such as `[[point_coord]]`, `[[front_facing]]`, `[[sample_id]]`, `[[sample_mask]]`) or refer to a framebuffer color value (such as `[[color]]`) passed as an input to the fragment function.

片元着色函数的单个实例输入使用“[[stage_in]]”修饰符定义，这些输入都是由相应的顶点着色函数输出的。

如章节“单个顶点输入数据用的属性修饰符”描述的那样，内建变量通过属性修饰符声明。这些内建变量通过顶点着色函数生成（比如`[[position]]`, `[[point_size]]`, `[[clip_distance]]`），或是通过光栅化生成（比如`[[point_coord]]`, `[[front_facing]]`, `[[sample_id]]`, `[[sample_mask]]`），或是引用一个帧缓存颜色值（比如`[[color]]`），他们可以作为输入传递到片元着色函数。

The built-in variable `[[position]]` must always be returned. The other built-in variables (`[[point_size]]`, `[[clip_distance]]`) generated by a vertex function, if needed, must be declared in the return type of the vertex function but cannot be accessed by the fragment function.

Built-in variables generated by the rasterizer or refer to a framebuffer color value may also declared as arguments of the fragment function with the appropriate attribute qualifier.

The attribute `[[user(name)]]` syntax can also be used to specify an attribute name for any user-defined variables.

内建变量“`[[position]]`”是顶点着色函数必须返回的，其它的内建变量（`[[point_size]]`, `[[clip_distance]]`）如果需要，在函数返回数据类型中声明，但是它们不能被片元着色函数访问到。

由光栅化生成的内建变量引用一个帧缓存颜色值，也可以通过加以特定的属性修饰符被声明为片元着色函数的参数。

如同“`[[user(name)]]`”的语法用于指定任一自定义变量的属性名。

A vertex and fragment function are considered to have matching signatures if:

- There is no input argument with the `[[stage_in]]` qualifier declared in the fragment function.
- For a fragment function argument declared with `[[stage_in]]`, each element in the type associated with this argument can be one of the following: a built-in variable generated by the rasterizer, a framebuffer color value passed as input to the fragment function, or a user-generated output from a vertex function. For built-in variables generated by the rasterizer or framebuffer color values, there is no requirement for a matching type to be associated with elements of the vertex return type. For elements that are user-generated outputs, the following rules apply:
  - If the attribute name given by `[[user(name)]]` is specified for an element, then this attribute name must match with an element in the return type of the vertex function, and their corresponding data types must also match.
  - If the `[[user(name)]]` attribute name is not specified, then the argument name and types must match.

一个顶点和片元着色函数如果满足一下条件之一，被认为是签名匹配的：

- 在片元着色函数的输入参数中没有使用“[[stage_in]]”修饰符。
- 有一个片元着色函数的输入参数使用“[[stage_in]]”修饰符，该参数的数据类型中的每个元素可以是下列之一：一个由光栅化生成的内建变量，一个帧缓存颜色值，一个来自顶点着色函数的用户自定义输出。对于来自光栅化的内建变量 或是 帧缓存颜色值，不要求与之对应的顶点着色函数返回值类型相匹配。对于用户自定义输入，要满足如下的规则：
  - 如果“[[user(name)]]”给出的属性名指定了一个元素，那么这个属性名必须和顶点着色函数返回值类型中的一个元素匹配，而且它们相关的数据类型也需要匹配。
  - 如果“[[user(name)]]”给出的属性名没有指明，那么参数名和类型必须匹配。

Below is an example of compatible signatures:

下面是一个签名匹配的例子：

```
struct VertexOutput
{
   float4 position [[position]];
   float3 normal;
   float2 texcoord;
};

vertex VertexOutput my_vertex_shader(...)
```

```
{
   VertexOutput v;
   ...
   return v;
}

fragment float4 my_fragment_shader(VertexOutput f [[stage_in]], ...)
{
   float4 clr;
   ...
   return clr;
}

fragment float4 my_fragment_shader2(
   VertexOutput f [[stage_in]],
   bool is_front_face [[front_facing]], ...)

{
   float4 clr;
   ...
   return clr;
}
```

`my_vertex_shader` and `my_fragment_shader`, or `my_vertex_shader` and `my_fragment_shader2` can be used together to render a primitive.

上面例子中，`my_vertex_shader` 加 `my_fragment_shader`，或是 `my_vertex_shader` 加 `my_fragment_shader2`，都是签名匹配的，可以用于绘制一个图元。

Below is another example of compatible signatures:

下面是另一个签名匹配的例子：[[user(normal)]]出现在顶点着色函数的输出中和片元着色函数的输入中，`user(texturecoord)`仅出现在顶点着色函数的输出中。

```
struct VertexOutput
{
   float4 position [[position]];
   float3 vertex_normal [[user(normal)]];
   float2 texcoord [[user(texturecoord)]];
};

struct FragInput
{
   float3 frag_normal [[user(normal)]];
   float4 position [[position]];
   float4 framebuffer_color [[color(0)]];
   bool is_front_face [[front_facing]];
};

vertex VertexOutput my_vertex_shader(...)
{
   VertexOutput v;
   ...
   return v;
}

fragment float4 my_fragment_shader(FragInput f [[stage_in]], ...)
{
   float4 clr;
   ...
   return clr;
}
```

Below is another example of compatible signatures:

下面是一个签名匹配的例子：

```
struct VertexOutput
{
   float4 position [[position]];
   float3 normal;
   float2 texcoord;
};

struct FragInput
```

```
{
   float4 position [[position]];
   float2 texcoord;
};

vertex VertexOutput my_vertex_shader(...)
{
   VertexOutput v;
   ...
   return v;
}

fragment float4 my_fragment_shader(FragInput f [[stage_in]], ...)
{
   float4 clr;
   ...
   return clr;
}
```

Below is another example of compatible signatures:

下面是另一个签名匹配的例子：

```
struct VertexOutput
{
     float4 position [[position]];
     float3 normal;
     float2 texcoord;
};

vertex VertexOutput my_vertex_shader(...)
{
   VertexOutput v;
   ...
   return v;
}

fragment float4 my_fragment_shader(float4 p [[position]], ...)
{
   float4 clr;
   ...
   return clr;
}
```

Below is an example of **in**compatible signatures:

下面是一个签名不匹配的例子：

```
struct VertexOutput
{
   float4 position [[position]];
   float3 normal;
   float2 texcoord;
};

struct FragInput
{
   float4 position [[position]];
   half3 normal; // 和顶点着色函数的输出不匹配？
};

vertex VertexOutput my_vertex_shader(...)
{
   VertexOutput v;
   ...
   return v;
}

fragment float4 my_fragment_shader(FragInput f [[stage_in]], ...)
{
   float4 clr;
   ...
   return clr;
}
```

Below is another example of incompatible signatures:

下面是另一个签名不匹配的例子

```
struct VertexOutput
{
   float4 position [[position]];
   float3 normal [[user(normal)]];
   float2 texcoord [[user(texturecoord)]];
};

struct FragInput
{
   float3 normal [[user(foo)]]; // 和顶点着色函数的输出不匹配?
   float4 position [[position]];
};

vertex VertexOutput my_vertex_shader(...)
{
   VertexOutput v;
   ...
   return v;
}

fragment float4 my_fragment_shader(FragInput f [[stage_in]], ...)
{
   float4 clr;
   ...
   return clr;
}
```

Additional Restrictions

其他约束

Writes to a buffer or a texture are disallowed from a fragment function. Writes to a texture are disallowed from a vertex shader.

Writes to a buffer from a vertex function are not guaranteed to be visible to reads from the associated fragment function of a given primitive.

The return type of a vertex or fragment function cannot include an element that is a packed vector type, matrix type, array type, or a reference or a pointer to a type.

A maximum of 128 scalars can be used as inputs to a fragment function declared with the `stage_in` qualifier. (The restriction does not include the built-in variables declared with one of the following attributes: `[[ color(m) ]]`, `[[ front_facing ]]`, `[[ sample_id ]]`, and `[[ sample_mask ]]`.) If an input to a fragment function is a vector, then the input vector counts as *n* scalars where *n* is the number of components in the vector.

在片元着色函数中写缓存和纹理是不允许的，在顶点着色函数中写数据到一个纹理也是不允许的。

在顶点着色函数中可以写数据到一个缓存，但是不保证，绘制特定图元，其对应的片元着色函数能够读取到新写入的值。

顶点着色函数和片元着色函数的返回值类型不可以包含这些类型的元素：packed vector, matrix, array（紧密填充的向量、矩阵、数组），指针引用。

片元着色函数使用了`stage_in`修饰符的情况下，最多可以有128个标量作为输入。（这个限制不包含如下这些内建变量：`[[ color(m) ]]`, `[[ front_facing ]]`, `[[ sample_id ]]`, `[[ sample_mask ]]`）如果片元着色函数的输入是一个向量，这个向量算作n个标量，n为向量的分量数。

Metal Standard Library

Metal 标准库

This chapter describes the functions supported by the Metal shading language standard library.

本章节描述Metal着色语言标准库提供的方法。

Namespace and Header Files

命名空间和头文件

The Metal standard library functions and enums are declared in the `metal` namespace. In addition to the header files described in the Metal standard library functions, the `<metal_stdlib>` header is available and can access all the functions supported by the Metal standard library.

Metal标准库中的方法和枚举值在metal命名空间中声明。除了标准库方法头文件以外，可以通过`<metal_stdlib>`访问所有的Metal标准库支持的方法（意思是有不少的标准库头文件，但是只要引用`< metal_stdlib >`就完事大吉了）。

Common Functions

通用方法

The functions in this section are in the Metal standard library and are defined in the header `<metal_common>`. `T` is one of the scalar or vector floating-point types.

这部分Metal标准库方法被声明在头文件`<metal_common>`中。T表示一个标量或是浮点类型向量。

`T clamp(T x, T minval, T maxval)`Returns `fmin(fmax(x, minval), maxval)`. Results are undefined if `minval > maxval`.

`T clamp(T x, T minval, T maxval)`，在保证minval比maxval小的基础上：如果x比minval小，返回minval。如果x比maxval大，返回maxval。如果x在minval 和 maxval之间，返回x。其逻辑等于fmin(fmax(x, minval), maxval)。如果minval不比maxval小，返回结果未定义。

`T mix(T x, T y, T a)`Returns the linear blend of `x` and `y` implemented as: `x + (y − x ) ∗ a`. `a` must be a value in the range 0.0 ... 1.0. If `a` is not in the range 0.0 ... 1.0, the return values are undefined.

`T mix(T x, T y, T a)`，返回x和y的线性混合结果，a为0，返回x。a为1返回y。a为0.5，返回x和y的中值。a越小返回值越偏向x，a越大返回值越偏向y。其逻辑等于`x + (y − x ) ∗ a`。a的取值方位是0.0到1.0。如果a超出取值范围，返回结果未定义。

`T saturate(T x)`Clamp `x` within the range of 0.0 to 1.0.

`T saturate(T x)`，该函数等于clamp(x, 0.0, 1.0)

`T sign(T x)`Returns 1.0 if x> 0, -0.0 if x= -0.0, +0.0 if x= +0.0, or -1.0 if x< 0. Returns 0.0 if `x` is a NaN.

`T sign(T x)`，返回x的正负情况，正返回1.0，负返回-1.0，如果x为NaN，返回0。

`T smoothstep(T edge0, T edge1, T x)`
Returns 0.0 if `x <= edge0` and 1.0 if `x >= edge1` and performs a smooth Hermite interpolation between 0 and 1 when `edge0 < x < edge1`. This is useful in cases where you want a threshold function with a smooth transition. Results are undefined if `edge0 >= edge1` or if `x`, `edge0` or `edge1` is a NaN. This is equivalent to:

T smoothstep(T edge0, T edge1, T x)，如果x<=edege0返回0.0，如果x>=edge1返回1.0。如果x在edege0和edge1之间，则返回一个埃尔米特插值。当你希望实施一个在临界值之间的平滑变换，可以使用该函数。如果edge0 >= edge1 时返回值未定义。该函数的逻辑等于如下

```
t = clamp((x − edge0)/(edge1 − edge0), 0, 1);
return t ∗ t ∗ (3 − 2 ∗ t);
```

`T step(T edge, T x)`Returns 0.0 if `x < edge`, otherwise it returns 1.0.

T step(T edge, T x)，当x<edge返回0.0，否则返回1.0。

For single-precision floating-point, Metal also supports a precise and fast variant of the following common functions: `clamp` and `saturate`. The difference between the fast and precise variants is how NaNs are handled. In the fast variant, the behavior of NaNs is undefined, whereas the precise variants follow the IEEE 754 rules for NaN handling. The `-ffast-math` compiler option (see Math Intrinsics Options (page 93)) is used to select the appropriate variant when compiling the Metal source. In addition, the `metal::precise` and `metal::fast` nested namespaces are available to provide developers a way to explicitly select the fast or precise variant of these common functions, respectively.

对于单精度浮点数，`clamp`和`saturate`方法，Metal支持精确（precise）和快速（fast）两种实现。其区别在于如何处理NaN数值。在fast模式下，NaN数值的行为未定义。在precise模式下，NaN遵循IEEE745规则。`-ffast-math` 编译选项可以在编译Metal源代码时用来指定选用哪种实现模式。另外，`metal::precise` 和 `metal::fast`可以用来显示指定在这些方法里面对一个变量使用precise或fast模式。

Integer Functions

整数函数

The functions in this section are in the Metal standard library and are defined in the header `<metal_integer>`. `T` is one of the scalar or vector integer types. `Tu` is the corresponding unsigned scalar or vector integer type.

这个章节中的函数被定义在头文件<metal_integer>中。T表示整形标量或是整形向量，Tu是无符号整形标量或是无符号整形向量。

`T abs(T x))` Returns `| x |`. 返回X的绝对值。

`Tu absdiff(T x, T y)` Returns `| x − y |` without modulo overflow. 返回x和y的差值的绝对值。

`T clamp(T x, T minval, T maxval)`Returns `min(max(x, minval), maxval)`. Results are undefined if `minval > maxval`. x如果在minval和maxval之间返回x，如果x比minval小返回minval，如果x比maxval大返回maxval。

`T clz(T x)`Returns the number of leading 0-bits in `x`, starting at the most significant bit position. If `x` is 0, returns the size in bits of the type of `x` or component type of `x`, if `x` is a vector. 表示x的二进制，从符号为开始数有多少个0，就返回多少。如果x为0，返回表示x数据类型（或者是x的分量的数据类型）的比特数。

`T ctz(T x)`Returns the count of trailing 0-bits in `x`. If `x` is 0, returns the size in bits of the type of `x` or component type of `x`, if `x` is a vector. 表示x的二进制，从末端开始数有多少个0，就返回多少。如果x为0，返回表示x数据类型（或者是x的分量的数据类型）的比特数。

`T hadd(T x, T y)`Returns `(x + y) >> 1`. The intermediate sum does not modulo overflow. 返回x和y的中值（如果x+y没有溢出）。

`T mulhi(T x, T y)`Computes `x * y` and returns the high half of the product of `x` and `y`. 计算x*y，乘积截半，返回高位。

`T madhi(T a, T b, T c)` Returns `mulhi(a, b) + c`.

`T madsat(T a, T b, T c)`Returns `a * b + c` and saturates the result. 计算a*b+c的值，然后用1填充高位返回。

`T max(T x, T y)`Returns `y` if `x < y`, otherwise it returns `x`. 返回x和y中较大的一个。

`T min(T x, T x)`Returns `y` if `y < x`, otherwise it returns `x`. 返回x和y中较小的一个。

`T popcount(T x)`Returns the number of non-zero bits in `x`. 返回表示x的二进制数中有多少非零的比特位。

`T rhadd(T x, T y)`Returns `(x + y + 1) >> 1`. The intermediate sum does not modulo overflow. 返回x和y的右中值（如果x+y+1没有溢出）。

`T rotate(T v, T i)`For each element in `v`, the bits are shifted left by the number of bits given by the corresponding element in `i`. Bits shifted off the left side of the element are shifted back in from the right. 对于v的每个分量进行操作，每个分量都左移i个比特位，被左移出去的i个比特位填充在分量左移后右边留下的i位中。

`T subsat(T x, T y)`Returns `x - y` and saturates the result. 计算x和y的差值，然后用1填充高位返回。

Relational Functions

关系比较函数

The functions in this section are in the Metal standard library and are defined in the header `<metal_relational>`. `T` is one of the scalar or vector floating-point types. `Ti` is one of the scalar or vector integer or boolean types. `Tb` only refers to the scalar or vector boolean types.

这个章节中的函数被定义在头文件<metal_relational>中。T表示浮点形标量或是浮点形向量，Ti是整形标量或是整形向量，Tb是布尔型标量或布尔型向量。

`bool all(Tb x)`Returns true only if all components of `x` are true. 如果x的所有分量都是true，返回true。

`bool any(Tb x)`Returns true only if any component of `x` is true. 如果x的分量中有一个维true，返回true。

Tb isfinite(T x) Tests for finite value. 测试x是否非无穷大值。

Tb isinf(T x) Tests for infinity value (positive or negative). 测试x是否无穷大值。

Tb isnan(T x) Tests for a NaN. 测试x是否 NaN。

Tb isnormal(T x)Tests for a normal value. 测试x是否一个归一化值。

Tb isordered(T x, T y)Tests if arguments x and y are ordered. Returns the result (x == x) && (y == y). 测试参数x和y是否规则，返回 (x == x) && (y == y)

Tb `isunordered(T x, T y)`Tests if arguments `x` and `y` are unordered. Returns true if `x` or `y` is NaN and false otherwise. 测试参数x和鱼是否非规则。如果x或y是NaN返回true，否则返回false。

`Tb not(Tb x)`Returns the component-wise logical complement of `x`. 返回x的非运算结果。

```
T select(T a, T b, Tb c)
Ti select(Ti a, Ti b, Tb c)
```
For each component of a vector type, `result[i] = c[i] ? b[i] : a[i]` For a scalar type,`result = c ? b : a`.

对于向量，每个元素如此计算，result[i] = c[i] ? b[i] : a[i]。对于标量，返回c ? b : a。

`Tb signbit(T x)`Tests for sign bit. Returns true if the sign bit is set for the floating-point value in `x` and false otherwise. 测试符号位，对于x表示的浮点数，如果符号位为1返回true，否则返回false。

Math Functions

数学函数

The functions in this section are in the Metal standard library and are defined in the header `<metal_math>`. `T` is one of the scalar or vector floating-point types. `Ti` refers only to the scalar or vector integer types.

这个章节中的函数被定义在头文件<metal_math>中。T表示浮点形标量或是浮点形向量，Ti是整形标量或是整形向量。

```
T abs(T x)
```

T fabs(T x) Computes absolute value of x.计算x的绝对值。

T acos(T x) Computes arc cosine function of x. 计算x的反余弦值。

T acosh(T x) Computes inverse hyperbolic cosine of x. 计算x的反双曲余弦值。

T asin(T x) Computes arc sine function of x. 计算x的反正弦值。

T asinh(T x) Computes inverse hyperbolic sine of x. 计算x的反双曲正弦值。

T atan(T y_over_x) Computes arc tangent function of y_over_x. 计算y_over_x的反正切值。

T atan2(T y, T x) Computes arc tangent of y over x. 计算y对于x的反正切值。

T atanh(T x) Computes hyperbolic arc tangent of x. 计算x的反双曲正切值。

T ceil(T x) Rounds x to integral value using the round to positive infinity rounding mode. 将x取整，使用向正无穷取整模式。

T copysign(T x, T y)Returns x with its sign changed to match the sign of y. 改变x的符号位和y的符号位相同，返回之。

T cos(T x)Computes cosine of x. 计算x的余弦值。

T cosh(T x)Computes hyperbolic cosine of x. 计算x的双曲余弦值。

T exp(T x)Computes the base-e exponential of x. 计算e为底x的指数。

T exp2(T x)Computes the base-2 exponential of x. 计算2为底x的指数。

T exp10(T x)Computes the base-10 exponential of x. 计算10为底x的指数。

T fdim(T x, T y)Returns x − y if x > y, +0 if x is less than or equal toy. 如果x>y,返回x-y，否则返回+0

T floor(T x)Rounds x to integral value using the round to negative infinity rounding mode. X取整，使用向负无穷取整模式。

T fmod(T x, T y)Returns x − y ∗ trunc(x/y). 计算x的对y取模后的余数。比如 fmod(8.1 , 3) = 2.1 = 8.1 – 3 * trunk(2.7) = 8.1 – 3 * 2 = 2.1

T fract(T x)Returns the fractional part of x which is greater than or equal to 0 or less than 1. 返回x的小数部分（是一个大等于0小于1的数）。

T frexp(T x, Ti &exp)

Extracts mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1] or 0. Each component of x equals mantissa returned ∗ 2^{exp} 。

从 x 提取尾数和幂指数。其中尾数作为返回值，它是一个在[1/2, 1]区间的浮点数或是 0，幂指数在 exp 变量中被带回。返回值和 exp 满足这个公式：x = 返回值 * 2^{exp}。比如 x=16.4; int n; 那么 y=frexp(x, &n); 则 y=0.5125，n=5。x= 16.4 = 0.5124 * 2^5 = 0.5125 * 32

Ti ilogb(T x) Returns the exponent as an integer value.

T ldexp(T x, Ti k) Multiply x by 2 to the power k. 返回x乘以2的k次方。

T log(T x)Computes natural logarithm of x. 返回x的自然对数。

T log2(T x)Computes a base-2 logarithm of x. 返回以2为底x的对数。

T log10(T x)Computes a base-10 logarithm of x. 返回以10为底x的对数。

T max(T x, T y) 和 T fmax(T x, T y)

Returns y if x < y, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN. (The NaN behavior is only valid for precise::fmax and precise::max.)

如果x<y,返回y，否则返回x，如果其中一个参数是NaN，函数返回另一个参数，如果两个参数都是NaN，函数返回NaN（NaN仅对precise::fmax和precise::max有效）

T min(T x, T y) 和 T fmin(T x, T y)

Returns y if y < x, otherwise it returns x. If one argument is a NaN, fmin() returns the other argument. If both arguments are NaNs, fmin() returns a NaN. (The NaN behavior is only valid for precise::fmin and precise::min.)

如果y<x,返回y，否则返回x，如果其中一个参数是NaN，函数返回另一个参数，如果两个参数都是NaN，函数返回NaN（NaN仅对precise::fmin和precise::min有效）

T modf(T x, T &intval)

Decompose a floating-point number. The modf function breaks the argument x into integral and fractional parts each of which has the same sign as the argument. Returns the fractional value. The integral value is returned in intval.

该函数分解一个浮点数值，把参数分成整数部分和剩余小数部分，整数部分和小数部分都具有和x一样的正负符号。函数返回小数部分，整数部分以赋值给intval的方式被带回。

`T pow(T x, T y)` Computes `x` to the power `y`. 计算x的y次方。

`T powr(T x, T y)` Computes `x` to the power `y`, where `x` is >= 0. 计算x的y次方，其中x>=0

`T rint(T x)` Rounds `x` to integral value using round to nearest even rounding mode in floating-point format.

`T round(T x)` Returns the integral value nearest to `x` rounding halfway cases away from zero.

`T rsqrt(T x)` Computes inverse square root of `x`. 计算x的平方根倒数。

`T sin(T x)` Computes sine of `x`. 计算x的正弦值。

`T sincos(T x, T &cosval)`Computes sine and cosine of `x`. The computed sine is the return value and the computed cosine is returned in `cosval`. 计算x的正弦值和余弦值，计算得到的正弦值作为返回值，计算得到的余弦值以赋值给cosval的方式被带回。

`T sinh(T x)`Computes hyperbolic sine of `x`. 计算x的双曲正弦值。

`T sqrt(T x)`Computes square root of `x`. 计算x的平方根。

`T tan(T x)`Computes tangent of `x`. 计算x的正切值。

`T tanh(T x)`Computes hyperbolic tangent of `x`. 计算x的双曲正切值。

`T trunc(T x)`Rounds `x` to integral value using the round to zero rounding mode. 对x进行取整计算，使用向0取整模式。

For single precision floating-point, the Metal shading language supports two variants of the math functions listed above: the precise and the fast variants. The `-ffast-math` compiler option (refer to Math Intrinsics Options (page 93)) is used to select the appropriate variant when compiling Metal shading language source. In addition, the `metal::precise` and `metal::fast` nested namespaces are also available and provide developers a way to explicitly select the fast or precise variant of these math functions for single precision floating-point.

对于单精度浮点数，Metal着色语言支持两种版本的上面列出的数学函数：精确版和快速版。编译选项-ffast-math用来指定选取哪种版本用于编译Metal着色语言源代码。在以上函数运算单精度浮点数的时候，metal::precise和metal::fast命名空间可以用于显示选择使用精确版或是快速版。

Examples:

```
#include <metal_stdlib>

using namespace metal;

float x;
float a = sin(x); // use fast or precise version of sin based on whether -ffast-math is specified as compile
option.
float b = fast::sin(x);    // use fast version of sin()
float c = precise::cos(x); // use precise version of cos()
```

Matrix Functions

矩阵函数

The functions in this section are in the Metal standard library and are defined in the header `<metal_matrix>`.

下面这些函数被定义在头文件`<metal_matrix>`中。

```
float determinant(floatnxn)
half determinant(halfnxn)
```

Computes the determinant of `matrix`, which must be a square matrix. 计算矩阵的行列式值，作为参数的矩阵是方阵。

```
floatmxn transpose(floatnxm)
halfmxn transpose(halfnxm)
```

Transpose `matrix`. 计算转置矩阵。

Example:

```
float4x4 mA;
float det = determinant(mA);
```

Geometric Functions

几何函数

The functions in this section are in the Metal standard library and are defined in the header `<metal_geometric>`. `T` is a vector floating-point type (`floatn` or `halfn`). `Ts` refers to the corresponding scalar type (i.e. `float` if `T` is `floatn` and `half` if `T` is `halfn`).

本小节列出的函数被定义在头文件中`<metal_geometric>`。T是一个浮点向量（floatn或halfn）。Ts表示T对应的标量类型。

`T cross(T x, T y)` Returns the cross product of `x` and `y`. `T` must be a 3-component vector type. 返回向量x和y的叉积（外积），T必须是3维向量。

`Ts distance(T x, T y)` Returns the distance between `x` and `y`; i.e., `length(x - y)` 返回向量x和y的距离。

`Ts distance_squared(T x, T y)` Returns the square of the distance between `x` and `y`. 返回向量x和y的距离的平方。

`Ts dot(T x, T y)` Returns the dot product of `x` and `y`; i.e., `x[0] * y[0] + x[1] * y[1] + ...` 返回向量x和y的点积。

`T faceforward(T N, T I, T Nref)` If `dot(Nref, I) < 0.0` returns `N`, otherwise returns `–N`. 计算参数I和参数Nref的点积，如果点积小于0.0，返回N，否则返回 –N。

`Ts length(T x)` Returns the length of vector `x`; i.e., `sqrt(x[0]^2 + x[1]^2 + ...)` 返回向量x的长度，计算公式为sqrt(x[0]^2 + x[1]^2 + ...)。

`Ts length_squared(T x)` Returns the square of the length of vector `x`; i.e., `x[0]^2 + x[1]^2 + ...` 返回向量x的长度的平方，计算公式为x[0]^2 + x[1]^2 + ...。

`T normalize(T x)` Returns a vector in the same direction as `x` but with a length of 1. 返回和x方向一致但是长度为1的向量。

`T reflect(T I, T N)`

For the incident vector `I` and surface normal `N`, returns the reflection direction: `I – 2 * dot(N, I) * N` In order to achieve the desired result, `N` must be normalized.

对于入射向量I，和平面法向量N，返回反射向量。计算公式为I – 2 * dot(N, I) * N。为了得到期望的值，N必须是单位向量。

`T refract(T I, T N, Ts eta)`

For the incident vector `I`, the surface normal `N`, and the ratio of indices of refraction `eta`, returns the refraction vector. The input parameters for the incident vector `I` and the surface normal `N` must already be normalized to get the desired results.

对于入射向量I，和平面法向量N，入射向量的折射率eta，返回折射向量。入射向量I，平面法向量N必须是单位向量，才能得到期望的结果。

For single precision floating-point, Metal also supports a precise and fast variant of the following geometric functions: `distance`, `length`, and `normalize`. The `-ffast-math` compiler option (refer to Math Intrinsics Options (page 93)) is used to select the appropriate variant when compiling the Metal shading language source. In addition, the `metal::precise` and `metal::fast` namespaces are also available and provide developers a way to explicitly select the fast or precise variant of these geometric functions.

对于单精度浮点数，这些函数：distance, length, normalize，Metal支持精确版和快速版。编译选项-ffast-math被用来在编译时选取使用哪个版本。metal::precise和metal::fast命名空间可以用于显示选择使用精确版或是快速版。

Compute Functions

并行计算用函数

The functions in this section and its subsections can only be called from a `kernel` function and are defined in the header `<metal_compute>`.

这个小节列出的函数只能在并行计算程序中使用，这些函数定义在头文件<metal_compute>中。

threadgroup Synchronization Functions

线程组同步函数

The threadgroup function described below is supported.

`void threadgroup_barrier(mem_flags flags)`
All threads in a threadgroup executing the kernel must execute this function before any thread is allowed to continue execution beyond the `threadgroup_barrier`.

对于线程组中的任一个线程（执行kernel程序），在其他线程被允许越过线程组栅栏（threadgroup_barrier）继续执行之前，必须执行这个函数。

The `threadgroup_barrier` function acts as an execution and memory barrier. The `threadgroup_barrier` function must be encountered by all threads in a threadgroup executing the kernel.

threadgroup_barrier作为执行和内存栅栏。在一个线程组中的所有线程都必须调用到该函数（如果kernal程序中出现了它）。

If `threadgroup_barrier` is inside a conditional statement and if any thread enters the conditional statement and executes the barrier, then all threads in the threadgroup must enter the conditional and execute the barrier.

如果threadgroup_barrier在一个条件语句中并且任一个线程进入了这个语句调用到了它，那么线程组中的所有线程必须都进入到这个条件语句并执行这个栅栏。

If `threadgroup_barrier` is inside a loop, for each iteration of the loop, all threads in the threadgroup must execute the `threadgroup_barrier` before any threads are allowed to continue execution beyond the `threadgroup_barrier`.

如果threadgroup_barrier在一个循环体内，每次循环迭代，线程组中得所有的线程都必须在执行到threadgroup_barrier停下来，直到线程组中所有的线程都执行了threadgroup_barrier，再继续执行。

The `threadgroup_barrier` function can also queue a memory fence (reads and writes) to ensure correct ordering of memory operations to threadgroup or device memory.

threadgroup_barrier还可以串行化一个内存栅栏操作（读和写），如此保证线程组内存操作或是设备内存操作按正确的时序进行。

The `mem_flags` argument in `threadgroup_barrier` can be one of the following flags, as described in Table 5-1 (page 72).

threadgroup_barrier函数中的mem_flags参数可以是下表中的值。

Table 5-1 `mem_flags` Enum Values for `threadgroup_barrier`

mem_flags	Description
`mem_none`	In this case, no memory fence is applied, and `threadgroup_barrier` acts only as an execution barrier. 表示不实施内存栅栏操作，函数只做执行栅栏。
`mem_device`	Ensure correct ordering of memory operations to device memory. 表示确保设备内存操作按照正确的时序进行。
`mem_threadgroup`	Ensure correct ordering of memory operations to threadgroup memory for threads in a threadgroup. 表示确保线程组内存操作按照正确的时序进行。

mem_flags	Description
`mem_device_and_threadgroup`	Ensure correct ordering of memory operations to device and threadgroup memory for threads in a threadgroup.

	表示确保设备内存操作 和 设备内存操作 都按照正确的时序进行。

The enumeration types used by `mem_flags` are specified as follows:

mem_flags使用的枚举类型如下所示：

```
enum class mem_flags {mem_none,
                      mem_device,
                      mem_threadgroup,
                      mem_device_and_threadgroup };
```

Graphics Functions

图形绘制用函数

This section and its subsections list the set of graphics functions that can be called by fragment functions. These are defined in the header `<metal_graphics>`.

该小节列出的图形绘制用的函数可以在片元着色程序中使用，它们被定义在头文件< metal_graphics >中

Fragment Functions

The functions in these subsections can only be called inside a fragment function (a function declared with the `fragment` qualifier) or inside a function called from a fragment function. Otherwise the behavior is undefined and may result in a compile-time error.

下面列出的函数只能在片元着色程序中被调用（使用fragment修饰符号声明的）或者出现在片元着色程序调用的函数中。如果不是这样，这些函数的行为未定义，或许会造成编译错误。

Fragment Functions – Derivatives

Metal includes the following functions to compute derivatives. `T` is one of `float`, `float2`, `float3`, `float4`, `half`, `half2`, `half3` or `half4`.

Metal包含下面这些函数用于计算导数，T的类型可以是 float, float2, float3, float4, half, half2, half3, half4

Note: Derivatives are undefined within non-uniform control flow.

```
T dfdx(T p)
```

Returns a high precision partial derivative of the specified value with respect to the screen space x coordinate. 返回相对于屏幕空间的x坐标的高精度的导数值。

```
T dfdy(T p)
```

Returns a high precision partial derivative of the specified value with respect to the screen space y coordinate. 返回相对于屏幕空间的y坐标的高精度的导数值。

`T fwidth(T p)`Returns the sum of the absolute derivatives in x and y using local differencing for `p`; i.e. `fabs(dfdx(p)) + fabs(dfdy(p))`. 计算参数p的dfdx的绝对值和dfdy的绝对值之和。公式为：fabs(dfdx(p)) + fabs(dfdy(p)

Fragment Functions – Samples

Metal includes the following per-sample functions.

Metal包含下面这些函数用于单个采样

```
uint get_num_samples()
```

Returns the number of samples for the multisampled color attachment. 对于多重采样颜色attachment，返回采样数。

```
float2 get_sample_position(uint indx)
```

Returns the normalized sample offset (x, y) for a given sample index `indx`. Values of x and y are in [-1.0, 1.0]. 对于一个给定的采样索引值indx，返回归一化的偏移向量(x, y)，其中x和y的取值范围是[-1.0, 1.0]。

`get_num_samples` and `get_sample_position`

return the number of samples for the color attachment and the sample offsets for a given sample index. For example, for transparency super-sampling, this can be used to shade per-fragment, but do the alpha test per-sample. 对于颜色attachment返回样本数量，对于一个采样索引返回采样的偏移。例如，对于“透明超采样”，这可以用来屏蔽每片元的alpha测试，启用每采样的alpha测试。

Fragment Functions – Flow Control

The following Metal function is used to terminate a fragment.

下面这个函数用来终结一个片元程序。

```
void discard_fragment(void)
```

Marks the current fragment as being terminated, and the output of the fragment function for this fragment is discarded. 该函数将当前的片元着色程序标记为结束，当前片元的输出被放弃。

Texture Functions

纹理函数

The texture functions are categorized into: sample from a texture, sample compare from a texture, read (sampler-less read) from a texture, gather from a texture, gather compare from a texture, write to a texture, and texture query functions.

纹理函数可以分类如下几类：从一个纹理采样，从一个纹理采样比较，从一个纹理中读取(sampler-less read)，从一个纹理中聚合，从一个纹理聚合比较，写入一个纹理，纹理查询函数。

These are defined in the header <metal_texture>.

这些函数定义在头文件<metal_texture>中。

The texture `sample`, `sample_compare`, `gather`, and `gather_compare` functions take an `offset` argument for a 2D texture, 2D texture array, 3D texture, and cube texture. The `offset` is an integer value that is applied to the texture coordinate before looking up each pixel. This integer value can be in the range -8 to +7. The default value is 0.

对于2D纹理，2D纹理数组，3D纹理和立方纹理，采样，采样比较，聚合，聚合比较函数都有一个offset参数。该参数是一个整型值，在查询每个像素之前用做纹理坐标，这个整数的值在-8到7之间，默认值是0。

Overloaded variants of texture `sample` and `sample_compare` functions for a 2D texture, 2D texture array, 3D texture, and cube texture are available and allow the texture to be sampled using a bias that is applied to a mip-level before sampling or with user-provided gradients in the x and y direction.

2D纹理，2D纹理数组，3D纹理和立方纹理的采样和采样比较函数存在Overloaded变量，它允许纹理采样时使用一个偏差（应用于一个mipmap层） 或 是一个用户提供的x方向和y方向的梯度。

Note: The texture `sample`, `sample_compare`, `gather`, and `gather_compare` functions require that the texture is declared with the `sample` access qualifier.

The texture `sample_compare` and `gather_compare` functions are only available for depth texture types.

The texture read functions require that the texture is declared with the `sample` or `read` access qualifier.

The texture write functions require that the texture is declared with the `write` access qualifier.

注意：纹理采样，采样比较，聚合，聚合比较函数需要纹理通过sample访问修饰符号定义。

纹理采样比较和聚合比较方法只适用于深度纹理类型。

纹理读取函数需要纹理定义时使用sample或read访问修饰符。

纹理写入函数需要纹理定义时使用write访问修饰符。

In this section, `Tv` is a 4-component vector type based on the templated type <T> used to declare the texture type. If `T` is `float`, `Tv` is `float4`. If `T` is `half`, `Tv` is `half4`. If `T` is `int`, `Tv` is `int4`. If `T` is `uint`, `Tv` is `uint4`. If `T` is `short`, `Tv` is `short4`. If `T` is `ushort`, `Tv` is `ushort4`.

在这个小节中，Tv是一个基于模板type<T>的4维向量类型，它用来定义纹理类型。如果T是float那么Tv是float4。如果T是half那么Tv是half4。如果T是int那么Tv是int4。如果T是uint那么Tv是unit4。如果T是short那么Tv是short4。如果T是ushort那么Tv是ushort4。

1D Texture

1维纹理方法

The following built-in function samples from a 1D texture. 下面的内建函数从一个1维纹理中采样。

```
Tv sample(sampler s, float coord) const
```

The following built-in function performs sampler-less reads from a 1D texture. 下面的内建函数从一个1维纹理做sampler-less读取。

```
Tv read(uint coord, uint lod = 0) const
```

The following built-in function writes to a specific mip-level of a 1D texture. 下面的内建函数向一个1维纹理的特定mipmap层写入数据。

```
void write(Tv color, uint coord, uint lod = 0)
```

The following built-in 1D texture query function is provided. 下面列举了系统支持的1维纹理查询方法。

```
uint get_width(uint lod = 0) const
uint get_num_mip_levels() const
```

1D Texture Array

1维纹理数组方法

The following built-in function samples from a 1D texture array. 下面的内建函数从一个1维纹理数组中采样。

```
Tv  sample(sampler s, float coord, uint array) const
```

The following built-in function performs sampler-less reads from a 1D texture array. 下面的内建函数从一个1维纹理数组做sampler-less读取。

```
Tv  read(uint coord, uint array, uint lod = 0) const
```

The following built-in function writes to a specific mip-level of a 1D texture array. 下面的内建函数向一个1维纹理的特定mipmap层写入数据。

```
void write(Tv  color, uint coord, uint array, uint lod = 0)
```

The following built-in 1D texture array query functions are provided. 下面列举了系统支持的1维纹理数组查询方法。

```
uint get_width(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

2D Texture

2维纹理方法

The following data types and corresponding constructor functions are available to specify various sampling options.

下面列出的数据类型和对应的构造方法用于设定多个采样参数。

```
bias(float value)
level(float lod)
gradient2d(float2 dPdx, float2 dPdy)
```

The following built-in functions sample from a 2D texture.

下面列出的内建方法从一个2维纹理采样

```
Tv sample(sampler s,
```

```
        float2 coord,
        int2 offset = int2(0)) const

Tv sample(sampler s,
        float2 coord,
        lod_options options,
        int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

上面列出的方法中的参数`lod_options`必须取以下值中的一个：`bias`, `level`, `gradient2d`。

The following built-in function performs sampler-less reads from a 2D texture.

下面列出的内建方法从一个2维纹理实施sampler-less读。

```
Tv read(uint2 coord, uint lod = 0) const
```

The following built-in function writes to a 2D texture.

下面列出的内建方法向一个2维纹理写入数据。

```
void write(Tv color, uint2 coord, uint lod = 0)
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D texture.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个2维纹理采样时做双线性插值。

```
enum class component { x, y, z, w };
Tv gather(sampler s,
        float2 coord,
        int2 offset = int2(0),
        component c = component::x) const
```

The following built-in 2D texture query functions are provided.

下面列出了可用的内建的2维纹理查询方法

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Texture Sampling Example

2维纹理采样示例

The following code shows several uses of the 2D texture sample function, depending upon its arguments.

下面的代码展示2维纹理采样方法的几种用法（使用不同的参数）

```
#include <metal_stdlib>
using namespace metal;

texture2d<float> tex;
sampler s;

float2 coord;
int2 offset;
float lod;

// no optional arguments
float4 clr = tex.sample(s, coord);

// sample using a mip-level
clr = tex.sample(s, coord, level(lod));

// sample with an offset
clr = tex.sample(s, coord, offset);

// sample using a mip-level and an offset
clr = tex.sample(s, coord, level(lod), offset);
```

2D Texture Array

2维纹理数组方法

The following built-in functions sample from a 2D texture array.

下面列出的内建方法从一个2维纹理数组中采样。

```
Tv  sample(sampler s,
           float2 coord,
           uint array,
           int2 offset = int2(0)) const

Tv  sample(sampler s,
           float2 coord,
           uint array,
           lod_options options,
           int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

以上方法中的参数`lod_options`必须取以下值中的一个：`bias`, `level`, `gradient2d`。

The following built-in function performs sampler-less reads from a 2D texture array.

下面列出的内建方法从一个2维纹理数组实施sampler-less读。

```
Tv read(uint2 coord, uint array, uint lod = 0) const
```

The following built-in function writes to a 2D texture array.

下面列出的内建方法向一个2维纹理数组写入数据。

```
void write(Tv color,
           uint2 coord,
           uint array,
           uint lod = 0)
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D texture array.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个2维纹理数组采样时做双线性插值。

```
Tv gather(sampler s,
          float2 coord,
          uint array,
          int2 offset = int2(0),
          component c = component::x) const
```

The following built-in 2D texture array query functions are provided.

下面列出了可用的内建的2维纹理数组查询方法

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

3D Texture

3维纹理方法

The following data types and corresponding constructor functions are available to specify various sampling options.

下面列出的数据类型和对应的构造方法用于设定多个采样参数。

```
bias(float value)
level(float lod)
gradient3d(float3 dPdx, float3 dPdy)
```

The following built-in functions sample from a 3D texture.

下面列出的内建方法从一个3维纹理数组中采样。

```
Tv  sample(sampler s,
           float3 coord,
           int3 offset = int3(0)) const

Tv  sample(sampler s,
          float3 coord,
          lod_options options,
          int3 offset = int3(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient3d`.

以上方法中的参数`lod_options`必须取以下值中的一个：`bias`, `level`, `gradient2d`。

The following built-in function performs sampler-less reads from a 3D texture.

下面列出的内建方法从一个3维纹理数组进行sampler-less读操作。

```
Tv read(uint3 coord, uint lod = 0) const
```

The following built-in function writes to a 3D texture.

下面列出的内建方法向一个3维纹理数组写入数据。

```
void write(Tv color, uint3 coord, uint lod = 0)
```

The following built-in 3D texture query functions are provided.

下面列出了可用的内建的3维纹理数组查询方法

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

Cube Texture

立方纹理方法

The following data types and corresponding constructor functions are available to specify various sampling options.

下面列出的数据类型和对应的构造方法用于设定多个采样参数。

```
bias(float value)
level(float lod)
gradientcube(float3 dPdx, float3 dPdy)
```

The following built-in functions sample from a cube texture.

下面列出的内建方法从一个立方纹理数组中采样。

```
Tv sample(sampler s,
          float3 coord) const

Tv sample(sampler s,
          float3 coord,
          lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradientcube`.

以上方法中的参数`lod_options`必须取以下值中的一个：`bias`, `level`, `gradient2d`。

The following built-in function writes to a cube texture.

下面列出的内建方法向一个3维纹理数组写入数据。

```
void write(Tv color, float3 coord, uint lod = 0)
```

Note: Table5-2(page81)describes the cube texture face and the number used to identify the face.

下表描述了立方纹理的面和用于定义面的数字对应关系

Table 5-2 Cube Texture Face Number 立方纹理面编号表

Face number	Cube texture face
0	Positive X
1	Negative X
2	Positive Y
3	Negative Y
4	Positive Z
5	Negative Z

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a cube texture.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个立方纹理采样时做双线性插值。

```
Tv gather(sampler s,
        float3 coord,
        component c = component::x) const
```

The following built-in cube texture query functions are provided.

下面列出了内建的可用的立方纹理查询方法。

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_depth(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Multisampled Texture

2维多重采样纹理方法

The following built-in function performs a sampler-less read from a 2D multisampled texture.

下面列出的内建方法从一个2维多重采样纹理实施sampler-less读。

```
Tv read(uint2 coord, uint sample) const
```

The following built-in 2D multisampled texture query functions are provided.

下面列出了内建的可用的2维多重采样纹理查询方法。

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Depth Texture

2维深度纹理方法

The following data types and corresponding constructor functions are available to specify various sampling options.

下面列出的数据类型和对应的构造方法用于设定多个采样参数。

```
bias(float value)
level(float lod)
gradient2d(float2 dPdx, float2 dPdy)
```

The following built-in functions sample from a 2D depth texture.

下面列出的内建方法从一个2维深度纹理中采样。

```
T sample(sampler s,
         float2 coord,
         int2 offset = int2(0)) const

T sample(sampler s,
         float2 coord,
         lod_options options,
         int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

以上方法中的参数`lod_options`必须取以下值中的一个：`bias`, `level`, `gradient2d`。

The following built-in functions sample from a 2D depth texture and compare a single component against the specified comparison value.

下面列出的内建方法从一个2维深度纹理中采样并且和一个特定的变量进行比较。

```
T sample_compare(sampler s,
                 float2 coord,
                 float compare_value,
                 int2 offset = int2(0)) const

T sample_compare(sampler s,
                 float2 coord,
                 float compare_value,
                lod_options options,
                int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`. `T` must be a `float` type.

以上方法中的参数`lod_options`必须取以下值中的一个：`bias`, `level`, `gradient2d`。`T`必须是一个浮点类型。

Note: `sample_compare` performs a comparison of `compare_value` against the pixel value (1.0 if the comparison passes and 0.0 if it fails). These comparison result values per-pixel are then blended together as in normal texture filtering and the resulting value between 0.0 and 1.0 is returned.

注意：函数 sample_compare 实施 compare_value 和 像素值（如果比较通过返回1.0，比较不通过返回0.0）之间的比较。这些比较结果数值之后将在每像素被混合（如同向量纹理过滤操作）并且返回在0.0和1.0之间的返回值，

The following built-in function performs a sampler-less read from a 2D depth texture.

下面列出的内建方法从一个2维深度纹理进行sampler-less读操作。

```
T read(uint2 coord, uint lod = 0) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个2维深度纹理采样时做双线性插值。

```
Tv gather(sampler s,
          float2 coord,
          int2 offset = int2(0)) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture and comparing these samples with a specified comparison value (1.0 if the comparison passes and 0.0 if it fails).

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个2维深度纹理采样时做双线性插值，同时，还将这些采样点和一个特定值比较（如果比较通过返回1.0，比较不通过返回0.0）

```
Tv gather_compare(sampler s,
                  float2 coord,
                  float compare_value,
                  int2 offset = int2(0)) const
```

`T` must be a `float` type.

以上的方法中得T必须是一个浮点类型。

The following built-in 2D depth texture query functions are provided.

下面列出了内建的可用的2维深度纹理查询方法。

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Depth Texture Array

2维深度纹理数组方法

The following built-in functions sample from a 2D depth texture array.

下面列出的内建方法从一个2维深度纹理数组中采样。

```
T sample(sampler s,
         float2 coord,
         uint array,
         int2 offset = int2(0)) const

T sample(sampler s,
         float2 coord,
         uint array,
         lod_options options,
         int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

以上方法中的参数`lod_options`必须取以下值中的一个：`bias`, `level`, `gradient2d`。

The following built-in functions sample from a 2D depth texture array and compare a single component against the specified comparison value.

下面列出的内建方法从一个2维深度纹理数组中采样并且和一个特定的变量进行比较。

```
T sample_compare(sampler s,
                 float2 coord,
                 uint array,
                 float compare_value,
                 int2 offset = int2(0)) const
T sample_compare(sampler s,
                 float2 coord,
                 uint array,
                 float compare_value,
                 lod_options options,
                 int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`. `T` must be a `float` type.

以上方法中的参数`lod_options`必须取以下值中的一个：bias, level, gradient2d。T必须是一个浮点类型。

The following built-in function performs a sampler-less read from a 2D depth texture array.

下面列出的内建方法从一个2维深度纹理数组进行sampler-less读操作。

```
T read(uint2 coord, uint array, uint lod = 0) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture array.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个2维深度纹理数组采样时做双线性插值。

```
Tv gather(sampler s,
          float2 coord,
            uint array,
```

```
        int2 offset = int2(0)) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture array and comparing these samples with a specified comparison value.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个2维深度纹理数组采样时做双线性插值，同时，还将这些采样点和一个特定值比较（如果比较通过返回1.0，比较不通过返回0.0）

```
Tv gather_compare(sampler s,
                  float2 coord,
                  uint array,
                  float compare_value,
                  int2 offset = int2(0)) const
```

`T` must be a `float` type.

以上方法T必须是一个浮点类型。

The following built-in 2D depth texture array query functions are provided.

下面列出了内建的可用的2维深度纹理数组查询方法。

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

Cube Depth Texture

立方深度纹理

The following data types and corresponding constructor functions are available to specify various sampling options.

```
bias(float value)
level(float lod)
gradientcube(float3 dPdx, float3 dPdy)
```

The following built-in functions sample from a cube depth texture.

下面列出的内建方法从一个立方深度纹理数组中采样。

```
T sample(sampler s,
         float3 coord) const

T sample(sampler s,
         float3 coord,
         lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradientcube`.

以上方法中的参数`lod_options`必须取以下值中的一个：bias, level, gradient2d。

The following built-in functions sample from a cube depth texture and compare a single component against the specified comparison value.

下面列出的内建方法从一个立方深度纹理数组中采样并且和一个特定的变量进行比较。

```
T sample_compare(sampler s,
                 float3 coord,
                 float compare_value) const

T sample_compare(sampler s,
                 float3 coord,
                 float compare_value,
                 lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradientcube`. `T` must be a `float` type.

以上方法中的参数`lod_options`必须取以下值中的一个：bias, level, gradient2d。T必须是一个浮点类型。

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a cube depth texture.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个立方深度纹理数组采样时做双线性插值。

```
Tv gather(sampler s, float3 coord) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a cube depth texture and comparing these samples with a specified comparison value.

下面列出的内建方法可以实现一个4采样点聚合，这可以用于在对一个立方深度纹理数组采样时做双线性插值，同时，还将这些采样点和一个特定值比较（如果比较通过返回1.0，比较不通过返回0.0）

```
Tv gather_compare(sampler s,
                  float3 coord,
                  float compare_value) const
```

`T` must be a `float` type.

T必须是一个浮点类型。

The following built-in cube depth texture query functions are provided.

下面列出了内建的可用的立方深度纹理数组查询方法。

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Multisampled Depth Texture

2维多重采样深度纹理方法

The following built-in function performs a sampler-less read from a 2D multisampled depth texture.

下面列出的内建方法从一个2维多重采样深度纹理进行sampler-less读操作。

```
T read(uint2 coord, uint sample) const
```

The following built-in 2D multisampled depth texture query functions are provided.

下面列出了内建的可用的2维多重采样深度纹理查询方法。

```
uint get_width() const
uint get_height() const
uint get_num_samples() const
```

Pack and Unpack Functions

装包拆包函数

This section lists the Metal functions for converting a vector floating-point data to and from a packed integer value. The functions are defined in the header `<metal_pack>`. Refer to Texture Addressing and Conversion Rules (page 100) for details on how to convert from a 8-bit, 10-bit or 16-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value and vice-versa.

这个小节列出了一个浮点向量 和 一个紧密填充整数 之间装换的Metal函数。这些函数被定义在头文件<metal_pack>中。小节“纹理寻址和转换规则”将详细描述了如何从一个8比特，10比特或是16比特（有符号或是无符号）整形值转换成一个归一化的单精度或是半精度浮点数，以及如何反向转换。

Unpack Integer(s); Convert to a Floating-Point Vector

拆包紧密填充整数转换为一个浮点向量

The following functions unpack multiple values from a single unsigned integer and then convert them into floating-point values that are stored in a vector.

下列的方法从一个无符号整数中拆包多个值，然后将这多个值转换成浮点数然后存储在一个向量中返回。

```
float4 unpack_unorm4x8_to_float(uint x) // 单精度无符号
float4 unpack_snorm4x8_to_float(uint x) // 单精度有符号
half4 unpack_unorm4x8_to_half(uint x)   // 半精度无符号
half4 unpack_snorm4x8_to_half(uint x)   // 半精度有符号
```

Unpack a 32-bit unsigned integer into four 8-bit signed or unsigned integers and then convert each 8-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 4-component vector.

上面列出的函数拆包一个32比特的无符号整数为4个8比特的有符号或是无符号整数，然后将每个整数值转换为一个归一化的单精度或是半精度浮点数，然后生成一个4维的向量。

```
float4 unpack_unorm4x8_srgb_to_float(uint x)
half4 unpack_unorm4x8_srgb_to_half(uint x)
```

Unpack a 32-bit unsigned integer into four 8-bit signed or unsigned integers and then convert each 8-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 4-component vector. The r, g, and b color values are converted from sRGB to linear RGB.

上面列出的函数拆包一个32比特的无符号整数为4个8比特的有符号或是无符号整数，然后将每个整数值转换为一个归一化的单精度或是半精度浮点数，然后生成一个4维的向量。把待返回的向量看做一个颜色向量，其r，g，b分量颜色值再做一个从sRGB颜色空间到RGB颜色空间的转换。

```
float2 unpack_unorm2x16_to_float(uint x)
float2 unpack_snorm2x16_to_float(uint x)
half2 unpack_unorm2x16_to_half(uint x)
half2 unpack_snorm2x16_to_half(uint x)
```

Unpack a 32-bit unsigned integer into two 16-bit signed or unsigned integers and then convert each 16-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 2-component vector. (The `unpack_unorm2x16_to_half` and `unpack_snorm2x16_to_half` functions may result in precision loss when converting from a 16-bit unorm or snorm value to a half-precision floating point.)

上面列出的函数拆包一个32比特的无符号整数为2个16比特的有符号或是无符号整数，然后将每个整数值转换为一个归一化的单精度或是半精度浮点数，然后生成一个2维的向量。（函数 unpack_unorm2x16_to_half 和 unpack_snorm2x16_to_half 在转换一个16比特的有符号或是无符号数值为一个半精度浮点数时，将导致精度损失，）

```
float4 unpack_unorm10a2_to_float(uint x)
float3 unpack_unorm565_to_float(ushort x)
half4 unpack_unorm10a2_to_half(uint x)
half3 unpack_unorm565_to_half(ushort x)
```

Convert a 1010102 (10a2), or 565 color value to the corresponding normalized single- or half-precision floating-point vector.

上面列出的函数转换一个“1010102 (10a2)类型数值” 或 565颜色数值为对应的归一化的单精度或半精度浮点向量。

Convert Floating-Point Vector to Integers, then Pack the Integers

转换浮点向量为多个整数，然后装包这些整数

The following functions start with a floating-point vector, convert the components into integer values, and then pack the multiple values into a single unsigned integer.

下列的函数接受一个浮点向量为参数，转换其每一个分量为整数值，然后把这多个数值装包到一个无符号整数中。

```
uint pack_float_to_unorm4x8(float4 x)
uint pack_float_to_snorm4x8(float4 x)
uint pack_half_to_unorm4x8(half4 x)
uint pack_half_to_snorm4x8(half4 x)
```

Convert a 4-component vector normalized single- or half-precision floating-point value to four 8-bit integer values and pack these 8-bit integer values into a 32-bit unsigned integer.

上面列出的函数转换一个4维向量的归一化的单精度或是半精度浮点值为4个8比特的整数值，然后把这4个整数值装包到一个32比特的无符号整数中。

```
uint pack_float_to_srgb_unorm4x8(float4 x)
```

```
uint pack_half_to_srgb_unorm4x8(half4 x)
```

Convert a 4-component vector normalized single- or half-precision floating-point value to four 8-bit integer values and pack these 8-bit integer values into a 32-bit unsigned integer. The color values are converted from linear RGB to sRGB.

上面列出的函数转换一个4维向量的归一化的单精度或是半精度浮点值为4个8比特的整数值，然后把这4个整数值装包到一个32比特的无符号整数中。颜色值要从RGB颜色空间转换维sRGB颜色空间。

```
uint pack_float_to_unorm2x16(float2 x)
uint pack_float_to_snorm2x16(float2 x)
uint pack_half_to_unorm2x16(half2 x)
uint pack_half_to_snorm2x16(half2 x)
```

Convert a 2-component vector of normalized single- or half-precision floating-point values to two 16-bit integer values and pack these 16-bit integer values into a 32-bit unsigned integer.

上面列出的函数转换一个2维向量的归一化的单精度或是半精度浮点值为2个16比特的整数值，然后把这2个整数值装包到一个32比特的无符号整数中。

```
uint pack_float_to_unorm10a2(float4)
ushort pack_float_to_unorm565(float3)
uint pack_half_to_unorm10a2(half4)
ushort pack_half_to_unorm565(half3)
```

Convert a 4- or 3-component vector of normalized single- or half-precision floating-point values to a packed, 1010102 or 565 color integer value.

上面列出的函数，转换一个4维或是3维向量的归一化的单精度或是半精度浮点值为一个紧密填充的“1010102类型”或是565颜色类型整数值。

Atomic Functions

原子操作函数

The Metal shading language implements a subset of the C++11 atomics and synchronization operations. For atomic operations, only a `memory_order` of `memory_order_relaxed` is supported. If the atomic operation is to threadgroup memory, the memory scope of these atomic operations is a threadgroup. If the atomic operation is to device memory, then the memory scope is the GPU device.

Metal着色语言实现了C++ 11的原子和同步操作规范的一个子集。对于原子操作，只有memory_order_relaxed 的memory_order是被支持的。如果原子操作是对线程组内存而言，那么这些原子操作的内存范围就是一个线程组。如果原子操作是对设备内存而言，那么内存范围是GPU设备。

There are only a few kinds of operations on atomic types, although there are many instances of those kinds. This section specifies each general kind.

These are defined in the header `<metal_atomic>`.

只有不多几种原子类型操作，但是有很多的这些类型的实例，这个小节将讲述每一个类别。原子操作函数被定义在头文件< metal_atomic >中。

Note: Atomic operations to device and threadgroup memory can only be performed inside a kernel function (a function declared with the `kernel` qualifier) or inside a function called from a kernel function.

注意：对于设备内存 和 线程组内存原子操作只能在一个并行计算kernel程序中出现（函数签名使用了kernel修饰符的）或者 被kernel程序调用的函数。

The `memory_order` enum is defined as follows.

memory_order枚举定义如下：

```
enum memory_order {
       memory_order_relaxed
};
```

Atomic Store Functions

原子存储函数

These functions atomically replace the value pointed to by `obj` (`*obj`) with `desired`.

下列函数将替换由*obj指向的值为参数 desired 的值。

```
void atomic_store_explicit(volatile device atomic_int* obj,
                           int desired,
                           memory_order order)

void atomic_store_explicit(volatile device atomic_uint* obj,
                           uint desired,
                           memory_order order)

void atomic_store_explicit(volatile threadgroup atomic_int* obj,
                           int desired,
                           memory_order order)

void atomic_store_explicit(volatile threadgroup atomic_uint* obj,
                           uint desired,
                           memory_order order)
```

Atomic Load Functions

原子载入函数

These functions atomically obtain the value pointed to by `obj`.

下列函数将获取并返回由*obj指向的值。

```
int atomic_load_explicit(volatile device atomic_int* obj,
                         memory_order order)

uint atomic_load_explicit(volatile device atomic_uint* obj,
                          memory_order order)

int atomic_load_explicit(volatile threadgroup atomic_int* obj,
                         memory_order order)

uint atomic_load_explicit(volatile threadgroup atomic_uint* obj,
                          memory_order order)
```

Atomic Exchange Functions

原子交换函数

These functions atomically replace the value pointed to by `obj` with `desired` and return the value `obj` previously held.

下列函数将使用参数desired的值替换由*obj指向的值，并且把obj的原值作为返回值返回。

```
int atomic_exchange_explicit(volatile device atomic_int *obj,
                             int desired,
                             memory_order order)

uint atomic_exchange_explicit(volatile device atomic_uint *obj,
                              uint desired,
                              memory_order order)
int atomic_exchange_explicit(volatile threadgroup atomic_int *obj,
                             int desired,
                             memory_order order)

uint atomic_exchange_explicit(volatile threadgroup atomic_uint *obj,
                              uint desired,
                              memory_order order)
```

Atomic Compare and Exchange Functions

原子比较交换函数

These functions atomically compare the value pointed to by `obj` with the value in expected. If those values are equal, the function replaces `*obj` with `desired` (by performing a read-modify-write operation). The function returns the value `obj` previously held.

下列的函数比较由*obj指向的值和参数expected的值，如果两个值相等，那么替换obj为参数desired的值（实施 读-修改-写入 操作）。函数返回obj的原值。

```
bool atomic_compare_exchange_weak_explicit(
                        volatile device atomic_int *obj,
                        int *expected,
                        int desired,
                        memory_order succ,
                        memory_order fail)

bool atomic_compare_exchange_weak_explicit(
                        volatile device atomic_uint *obj,
                        uint *expected,
                        uint desired,
                        memory_order succ,
                        memory_order fail)

bool atomic_compare_exchange_weak_explicit(
                        volatile threadgroup atomic_int *obj,
                        int *expected,
                        int desired,
                        memory_order succ,
                        memory_order fail)

bool atomic_compare_exchange_weak_explicit(
                        volatile threadgroup atomic_uint *obj,
                        uint *expected,
                        uint desired,
                        memory_order succ,
                        memory_order fail)
```

Atomic Fetch and Modify Functions

原子获取和修改函数

The following operations perform arithmetic and bitwise computations. All these operations are applicable to an object of any atomic type. The key, operator, and computation correspondence is given in Table 5-3 (page 91).

下面列出的操作符实现算术和位移计算。所有这些操作都可以应用到一个对象的任意原子类型，下表给出了键名，操作符，对应计算方法。

Table 5-3 Atomic Operation Function

key	operator	computation
`add`	`+`	addition
`and`	`&`	bitwise and 按位与
`max`	`max`	compute max
`min`	`min`	compute min

key	operator	computation
`or`	`|`	bitwise inclusive or 按位或操作
`sub`	`-`	subtraction
`xor`	`^`	bitwise exclusive or 异或操作

Atomically replaces the value pointed to by `obj` with the result of the computation of the value specified by `key` and `arg`. These operations are atomic read-modify-write operations. For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow. There are no undefined results. It returns the value `obj` held previously.

替换由*obj指向的变量的值为 由参数key 和 arg计算得出的结果。这些操作是原子的 读取-修改-写入 操作。对于有符号的整形，算术计算被定义为使用两个操作数的补码，溢出时做silent wrap-around。不存在未定义的结果。函数返回obj的原值。

```
int atomic_fetch_key_explicit(volatile device atomic_int *obj,
                          int arg,
                          memory_order order)

uint atomic_fetch_key_explicit(volatile device atomic_uint *obj,
                          uint arg,
                          memory_order order)

int atomic_fetch_key_explicit(volatile threadgroup atomic_int *obj,
                          int arg,
                          memory_order order)

uint atomic_fetch_key_explicit(volatile threadgroup atomic_uint *obj,
                          uint arg,
                          memory_order order)
```

Compiler Options

编译选项

The Metal compiler can be used online (i.e. using the appropriate APIs to compile Metal shading language sources) or offline. Metal shading language source code that is compiled offline can be loaded as binaries, using the appropriate Metal framework API.

This chapter explains the compiler options supported by the Metal shading language compiler, which are categorized as pre-processor options, options for math intrinsics, options that control optimization and miscellaneous options. The online and offline Metal compiler support these options.

Metal编译器可以在运行时使用（比如使用特定的API编译Metal着色语言源代码）或是非运行时使用。Metal着色语言源代码在非运行时被编译后可以使用特定的Metal框架API，将编译结果作为二进制库载入。

这个章节解释Metal着色语言编译器所支持的编译选项，分为预处理选项，数学计算选项，优化选项和其他杂项。运行时和非运行时的情况Metal编译器都支持这些选项。

Pre-Processor Options

预处理选项

These options control the Metal preprocessor that is run on each program source before actual compilation.

一下这些选项控制Metal预处理器，预处理器在每个程序源码被实际编译前运行处理程序源代码。

`-D` *name*

Predefine *name* as a macro, with definition 1.

定义name为一个宏名，其值为1.

`-D` *name=definition*

The contents of *definition* are tokenized and processed as if they appeared in a #define directive. This option may receive multiple options, which are processed in the order in which they appear. This option allows developers to compile Metal code to change which features are enabled or disabled.

*definition*的内容被标记和处理，如同源代码中#define指令一般，这个选项可以接收多个值，它们将按照出现的先后顺序来处理。这个选项允许开发者编译Metal源代码时决定哪些特性开哪些特性关。

`-I` *dir*

Add the directory *dir* to the list of directories to be searched for header files. This option is only available for the offline compiler.

添加目录，用于寻找头文件，这个选项仅仅支持非运行时编译。

Math Intrinsics Options

数学计算选项

These options control compiler behavior regarding floating-point arithmetic. These options trade off between speed and correctness.

下列选项控制编译器在处理浮点算术运算时的行为，这两个选项一个有利于运算速度和另一个有利于运算精度。

`-ffast-math (default)`

This option enables the optimizations for floating-point arithmetic that may violate the IEEE 754 standard. It also enables the high-precision variant of math functions for single-precision floating-point scalar and vector types. `-ffast-math` is the default.

该选项为浮点算术运算做优化，但是可能不按IEEE754规范，它还使得数学函数中的高精度变量（单精度浮点标量和向量类型）生效。这是默认的数学计算选项。

`-fno-fast-math`

This option is the opposite of `-ffast-math`. It disables the optimizations for floating-point arithmetic that may violate the IEEE 754 standard. It also disables the high-precision variant of math functions for single-precision floating-point scalar and vector types.

该选项和“`-ffast-math`”相反，它关闭了对于浮点数学运算的优化（此优化有可能违反IEEE754规范）。它还关闭了数学函数中的高精度变量（单精度浮点标量和向量类型）支持。

Options to Request or Suppress Warnings

警告开关选项

The following options are available. 支持下面的选项

`-Werror`

Make all warnings into errors. 使得所有的警告都按错误输出。

`-W`

Inhibit all warning messages. 抑制不输出警告信息。

Numerical Compliance

数值规则

This chapter covers how the Metal shading language represents floating-point numbers with regard to accuracy in mathematical operations. Metal is compliant to a subset of the IEEE 754 standard.

这个章节讲述了Metal着色语言如何表现浮点数并且在数学计算操作中保持精度。Metal遵从IEEE754规范的一个子集。

INF, NaN and Denormalized Numbers

INF must be supported for single precision floating-point numbers and are optional for half precision floating-point numbers.

INF 对于单精度浮点数是一定支持的，对于半精度浮点数是可选支持的。

NaNs must be supported for single precision floating-point numbers (with fast math disabled) and are optional for half precision floating-point numbers. If fast math is enabled the behavior of handling NaN (as inputs or outputs) is undefined.

NaNs对于单精度浮点数是一定支持的（fast数学编译选项关闭的情况下），对于半精度浮点数是可选支持的。如果fast数学编译选项开启，NaN的行为未定义。

Denormalized single or half precision floating-point numbers passed as input or produced as the output of single or half precision floating-point operations may be flushed to zero.

Denormalized 单精度或是半精度浮点数，作为 浮点数操作的 输入参数或是输出结果 可以被冲刷为0。

Rounding Mode

取整模式

Either round to zero or round to nearest rounding mode may be supported for single precision and half precision floating-point operations.

向0取整 或是 向最近的整数取整，对于单精度和半精度浮点数操作都是支持的。

Floating-Point Exceptions

浮点异常

Floating-point exceptions are disabled in Metal.

浮点异常在Metal中不支持。

Relative Error as ULPs

以ULP值给出的相对误差

Table 7-1 (page 96) describes the minimum accuracy of single-precision floating-point basic arithmetic operations and math functions given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

表7-1描述了单精度浮点数基本算术操作和数学函数的最小精度（使用ULP值给出）

Table 7-1 Minimum Accuracy of Floating-Point Operations and Functions

Math Function	Min Accuracy - ULP values
x+y	Correctly rounded
x-y	Correctly rounded

x*y	Correctly rounded
1.0 / x	<= 2.5 ulp for single precision
x/y	<= 2.5 ulp for single precision
acos(x)	<= 4 ulp
acosh(x)	<= 4 ulp
asin(x)	<= 4 ulp
asinh(x)	<= 4 ulp
atan(x)	<= 5 ulp
atan2(y, x)	<= 6 ulp
atanh(x)	<= 5 ulp
ceil(x)	Correctly rounded
copysign(x)	0 ulp
cos(x)	<= 4 ulp
cosh(x)	<= 4 ulp
exp(x)	<= 4 ulp
exp2(x)	<= 4 ulp
exp10(x)	<= 4 ulp
fabs(x)	0 ulp
fdim(x, y)	Correctly rounded
floor(x)	Correctly rounded
fma(x, y, z)	Correctly rounded — see note below the table

Math Function	**Min Accuracy - ULP values**
fmax(x, y)	0 ulp
fmin(x, y)	0 ulp
fmod(x, y)	0 ulp
fract(x)	Correctly rounded
frexp(x, y)	0 ulp
ilogb(x)	0 ulp
ldexp(x, y)	Correctly rounded
log(x)	<= 4 ulp
log2(x)	<= 4 ulp
log10(x)	<= 4 ulp
modf(x, i)	0 ulp
pow(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
rint(x)	Correctly rounded

round(x)	Correctly rounded
rsqrt(x)	<= 2 ulp
sin(x)	<= 4 ulp
sincos(x, c)	<= 4 ulp
sinh(x)	<= 4 ulp
sqrt(x)	<= 3 ulp for single precision
tan(x)	<= 6 ulp
tanh(x)	<= 5 ulp
trunc(x)	Correctly rounded

Note: The `metal_math` header does not declare a `fma()` function that developers can directly call. However, the Metal compiler contracts floating-point expressions (i.e., a * b + c) to a fused multiply-add instruction.

注意：头文件“metal_math”没有声明一个叫fma()的函数供开发者直接使用，但是，Metal编译器将浮点表达式（a*b+c）看做一个相乘累加指令。

Table 7-2 (page 98) describes the minimum accuracy of single-precision floating-point arithmetic operations given as ULP values with fast math enabled (which is the default unless -ffast-math-disable is specified as a compiler option).

表7-2描述了在fast数学编译选项打开的情况下（没有显示使用编译选项-ffast-math-disable的情况下），使用ULP值标记的单精度浮点数算术操作的最小精度。

Table 7-2 Minimum Accuracy of Operations and Functions with Fast Math Enabled

Math Function	Min Accuracy - ULP values
x+y	Correctly rounded
x-y	Correctly rounded
x*y	Correctly rounded
1.0 / x	<= 2.5 ulp for x in the domain of 2^{-126} to 2^{126}
x/y	<= 2.5 ulp for x in the domain of 2^{-126} to 2^{126}
acos(x)	<= 5 ulp
acosh(x)	Implemented as log(x + sqrt(x * x – 1.0))
asin(x)	<= 5 ulp for |x| >= 2^{-125}
asinh(x)	Implemented as log(x + sqrt(x * x + 1.0))
atan(x)	<= 5 ulp
atanh(x)	Implemented as 0.5 * (log(1.0 + x) / log(1.0 – x))
atan2(y, x)	Implemented as atan(y / x)
cos(x)	For x in the domain [-pi, pi], the maximum absolute error is <= 2^{-13}. Outside that domain, the error is larger. 当x在区间[-pi, pi]中，最大的误差小于等于2^{-13}，在区间外，误差变大。
cosh(x)	Implemented as 0.5 * (exp(x) + exp(-x))
exp(x)	<= 3 + floor(fabs(2 * x)) ulp
exp2(x)	<= 3 + floor(fabs(2 * x)) ulp

Math Function	Min Accuracy - ULP values
exp10(x)	Implemented as exp2(x * log2(10))
fma(x, y, z)	Implemented either as a correctly rounded fma or as a multiply and an add, both of which are correctly rounded.
log(x)	For x in the domain [0.5, 2], the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <= 3 ulp 当x在区间[0.5, 2],最大的误差小于等于2^{-21}，x在区间外，最大误差小于等于3ulp。
log2(x)	For x in the domain [0.5, 2], the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <= 3 ulp 当x在区间[0.5, 2],最大的误差小于等于2^{-21}，x在区间外，最大误差小于等于3ulp。
log10(x)	Implemented as log2(x) * log10(2)
pow(x, y)	Implemented as exp2(y * log2(x))
powr(x, y)	Implemented as exp2(y * log2(x))
round(x)	Returns a value equal to the nearest integer to x. The fraction 0.5 is rounded in a direction chosen by the implementation. 返回最接近x的整型值，分数0.5的舍入方向由实现决定。
sin(x)	For x in the domain [-pi, pi], the maximum absolute error is <= 2^{-13}. Outside that domain, the error is larger. 当x在区间[-pi, pi]中，最大的误差小于等于2^{-13}，在区间外，误差变大。
sinh(x)	Implemented as 0.5 * (exp(x) – exp(-x))
sincos(x)	ulp values as defined for sin(x) and cos(x)
sqrt(x)	Implemented as 1.0 / rsqrt(x)
tan(x)	Implemented as sin(x) * (1.0 / cos(x))
tanh(x)	Implemented as (t – 1.0)/(t + 1.0) where t = exp(2.0 * x)

Edge Case Behavior in Flush To Zero Mode

边界行为

If denormals are flushed to zero, then a function may return one of these two results:

Any conforming result for non-flush-to-zero mode. If the result is a subnormal before rounding, it may be flushed to zero.

Any non-flushed conforming result for the function if one or more of its subnormal operands are flushed to zero. If the result is a subnormal before rounding, it may be flushed to zero

In each of the cases, if an operand or result is flushed to zero, the sign of the zero is undefined.

如果，那么一个函数可以返回下面两个结果中的一个：

任意符合non-flush-to-zero模式的值，如果结果在取整前是一个subnormal,它可以被冲刷为0。

任意符合non-flushed模式的值，如果一个或是多个操作数被冲刷为0。如果结果在取整前是一个subnormal,它可以被冲刷为0。

以上两种情况，如果一个操作数或是结果被冲刷为0，那么0的符号是未定义的。

Texture Addressing and Conversion Rules

纹理寻址和变换规则

The texture coordinates specified to the `sample`, `sample_compare`, `gather`, `gather_compare`, `read`, and `write` functions cannot be INF or NaN. In addition, the texture coordinate must refer to a region inside the texture for the texture `read` and `write` functions.

The following sections discuss conversion rules that are applied when reading and writing textures in a graphics or kernel function.

纹理坐标对于下面这些函数不能为INF 或是 NaN：sample, sample_compare, gather, gather_compare, read, write。还有，对于read和write函数，纹理坐标必须对应的是一个区域内部。

下面将讨论在图形着色程序和并行计算kernel程序中，纹理的读取和写入中应用的变换规则。

Conversion rules for normalized integer pixel data types

归一化整形像素数据类型

This section discusses converting normalized integer pixel data types to floating-point values and vice-versa.

这个小节讨论转换归一化的像素数据类型 到 浮点型数值 以及 反向转换。

Converting normalized integer pixel data types to floating-point values

转换归一化的整形像素数据类型到浮点数值

For textures that have 8-bit, 10-bit or 16-bit normalized unsigned integer pixel values, the texture sample and read functions convert the pixel values from an 8-bit or 16-bit unsigned integer to a normalized single or half-precision floating-point value in the range [0.0, 1.0].

对于含有8比特，10比特，16比特的无符号整形像素值的纹理，纹理采样和读取函数将转换像素数值，从一个8比特或是16比特的无符号整数转换为归一化的单精度或是半精度浮点数，其值域为[0.0, 1.0]。

For textures that have 8-bit or 16-bit normalized signed integer pixel values, the texture sample and read functions convert the pixel values from an 8-bit or 16-bit signed integer to a normalized single or half-precision floating-point value in the range [-1.0, 1.0].

对于含有8比特，10比特，16比特的有符号整形像素值的纹理，纹理采样和读取函数将转换像素数值，从一个8比特或是16比特的有符号整数转换为归一化的单精度或是半精度浮点数，其值域为[-1.0, 1.0]。

These conversions are performed as listed in the second column of Table 7-3 (page 100). The precision of the conversion rules are guaranteed to be <= 1.5 ulp except for the cases described in the third column.

转换的算法如下表7-3的第二列所示。转换精度保证小于1.5ulp，除去下表第三列中得特殊情况。

Table 7-3 Rules for Conversion from a Normalized Integer to a Normalized Floating-Point Value

Convert from	Conversion Rule to Normalized Float	Corner Cases
8-bit normalized unsigned integer	`float(c) / 255.0`	0 must convert to 0.0 255 must convert to 1.0
10-bit normalized unsigned integer	`float(c) / 1023.0`	0 must convert to 0.0 1023 must convert to 1.0
16-bit normalized unsigned integer	`float(c) / 65535.0`	0 must convert to 0.0 65535 must convert to 1.0

Convert from	Conversion Rule to Normalized Float	Corner Cases
8-bit normalized signed integer	`max(-1.0, float(c)/127.0)`	-128 and -127 must convert to -1.0 0 must convert must convert to 1.0
16-bit normalized signed integer	`max(-1.0, float(c)/32767.0)`	-32768 and -32767 must convert to -1.0 0 must co 0.032767 must convert to 1.0

Converting floating-point values to normalized integer pixel data types

转换浮点数到归一化整形像素数据类型

For textures that have 8-bit, 10-bit or 16-bit normalized unsigned integer pixel values, the texture write functions convert the single or half-precision floating-point pixel value to an 8-bit or 16-bit unsigned integer.

对于含有8比特，10比特，16比特的无符号整形像素值的纹理，纹理写函数转换单精度或是半精度浮点像素值为一个8比特或是16比特无符号整数。

For textures that have 8-bit or 16-bit normalized signed integer pixel values, the texture write functions convert the single or half-precision floating-point pixel value to an 8-bit or 16-bit signed integer.

对于含有8比特，10比特，16比特的有符号整形像素值的纹理，纹理写函数转换单精度或是半精度浮点像素值为一个8比特或是16比特有符号整数。

The preferred methods to perform conversions from floating-point values to normalized integer values are listed in Table 7-4 (page 101).

表7-4 列出了从浮点数值转换为归一化整型值的方法。

Table 7-4 Rules for Conversion from a Floating-Point to a Normalized Integer Value

Convert to	Conversion Rule to Normalized Integer
8-bit normalized unsigned integer	`float(c) / 255.0`
10-bit normalized unsigned integer	`float(c) / 1023.0`
16-bit normalized unsigned integer	`float(c) / 65535.0`
8-bit normalized signed integer	`max(-1.0, float(c)/127.0)`
16-bit normalized signed integer	`max(-1.0, float(c)/32767.0)`

The GPU may choose to approximate the rounding mode used in the conversions described in Table 7-4 (page 101). If a rounding mode other than round to nearest even is used, the absolute error of the implementation dependent rounding mode vs. the result produced by the round to nearest even rounding mode must be <= 0.6.

GPU将选择合适的取整模式应用于表7-4的转换方法。如果使用一个不同于“取整到最近偶数”的取整模式，那么绝对误差相比使用“取整到最近偶数”必定是小于等于0.6。

Conversion rules for half precision floating-point pixel data type

半精度浮点数像素数据类型转换规则

For textures that have half-precision floating-point pixel color values, the conversions from `half` to `float` are lossless. Conversions from `float` to `half` round the mantissa using the round to nearest even or round to zero rounding mode. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` may be flushed to zero. A float `NaN` may be converted to an appropriate NaN or be flushed to zero in the `half` type. A `float` INF may be converted to an appropriate INF or be flushed to zero in the `half` type.

对于半精度浮点数像素颜色值的纹理，从半精度到单精度浮点数的转换是没有损失的，从单精度到半精度浮点数的转换时，使用“向最近偶数”或是“向零取整”取整舍入尾数。Denormalized 对于半精度数据类型（转换一个单精度数为半精度数的过程产生）会被冲刷为0。一个浮点数NaN可以被转换维一个恰当的NaN或是被冲刷为0（半精度类型）。一个浮点数INF可以被转换维一个恰当的INF或是被冲刷为0（半精度）。

Conversion rules for floating-point channel data type

浮点通道数据类型的转换规则

The following rules apply for reading and writing textures that have single-precision floating-point pixel color values.

读取和写入单精度浮点像素颜色类型纹理对象时，下列规则将被应用

- NaNs may be converted to a NaN value(s) or be flushed to zero. NaNs将被转换为NaN值或是冲刷为0
- INFs may be converted to a INF value(s) or be flushed to zero. INFs将被转换为INF值或者冲刷为0
- Denorms may be flushed to zero. Denorms将被冲刷为0
- All other values must be preserved. 其他的值被保留

Conversion rules for signed and unsigned integer pixel data types

有符号和无符号整形像素数据类型的转换规则

For textures that have 8-bit or 16-bit signed or unsigned integer pixel values, the texture sample and read functions return a signed or unsigned 32-bit integer pixel value. The conversions described in this section must be correctly saturated. Writes to these integer textures perform one of the conversions listed in Table 7-5 (page 102).

对于8比特或是16比特，有符号或是无符号整形像素值的纹理，纹理的采样和读取操作返回一个有符号或是无符号的32比特整数像素值。这个小节描述的转换必须正确实施延展。向这类整形像素纹理写入时，将应用下标列出的转换方法中的一个。

Table 7-5 Rules for Conversion between Integer Pixel Data Types

Convert from	Convert to	Conversion Rule
32-bit signed integer	8-bit signed integer	`result = convert_char_saturate(val)`
32-bit signed integer	16-bit signed integer	`result = convert_short_saturate(val)`
32-bit unsigned integer	8-bit unsigned integer	`result = convert_uchar_saturate(val)`
32-bit unsigned integer	16-bit unsigned integer	`result = convert_ushort_saturate(val)`

Conversion rules for sRGBA and sBGRA Textures

sRGBA和SBGRA纹理的转换规则

Conversion from sRGB space to linear space is automatically done when sampling from an sRGB texture. The conversion from sRGB to linear RGB is performed before the filter specified in the sampler specified when sampling the texture is applied. If the texture has an alpha channel, the alpha data is stored in linear color space.

当从一个sRGB纹理中采样的时候，从sRGB颜色空间转换到线性颜色空间是自动完成的。当纹理采样实施的时候，从sRGB颜色空间到线性RGB颜色空间的转换 先于 特定采样器的滤波操作。如果纹理有alpha通道，alpha数据被存储在线性颜色空间中。

Conversion from linear to sRGB space is automatically done when writing to an sRGB texture. If the texture has an alpha channel, the alpha data is stored in linear color space.

当向一个sRGB纹理写入时，从sRGB颜色空间转换到线性颜色空间是自动完成的。如果纹理有alpha通道，alpha数据被存储在线性颜色空间中。

The following is the conversion rule for converting a normalized 8-bit unsigned integer sRGB color value to a floating-point linear RGB color value (call it `c`) as per rules described in Converting normalized integer pixel data types to floating-point values (page 100).

下面列出的是转换一个归一化的8比特无符号整形sRGB颜色值 为 一个浮点线性RGB颜色值（命名为c）的逻辑。

```
if (c <= 0.04045),
    result = c / 12.92;
else
    result = powr((c + 0.055) / 1.055, 2.4);
```

The resulting floating point value, if converted back to an sRGB value without rounding to a 8-bit unsigned integer value, must be within 0.5 ulp of the original sRGB value.

上面逻辑的计算结果浮点数值，如果转换回去到一个sRGB值（不经过舍入，8bit无符号整形值），那么和原来的sRGB值相比，误差将在0.5ulp。

The following are the conversion rules for converting a linear RGB floating-point color value (call it `c`) to a normalized 8-bit unsigned integer sRGB value.

下面列出的是转换一个线性RGB浮点颜色值（命名为c）为 一个归一化的8比特无符号整形sRGB值 的逻辑。

```
if (isnan(c))
    c = 0.0;
```

```
if (c > 1.0)
    c = 1.0;
else if (c < 0.0)
    c = 0.0;
else if (c < 0.0031308)
    c = 12.92 * c;
else
    c = 1.055 * powr(c, 1.0/2.4) - 0.055;
convert to integer scale i.e. c = c * 255.0
convert to integer:
    c = c + 0.5
     drop the decimal fraction, and the remaining
     floating-point(integral) value is converted
       directly to an integer.
```

The precision of the above conversion should be such that:

上面的转换逻辑的精度计算如下：

```
fabs(reference result – integer result) <= 0.6
```

Document Revision History

文档版本历史

This table describes the changes to *Metal Shading Language Guide* .

2014-09-17 New document that describes the programming language used to create graphics and compute functions to use with the Metal framework.

2014-09-17 新建文档，描述了用于创建图形着色程序和并行计算程序（配合Metal框架使用的）编程语言。