Metal Programming Guide
Metal 编程指南

About Metal and this Guide
关于Metal和本文

The Metal framework supports GPU-accelerated advanced 3D graphics rendering and data-parallel computation workloads. Metal provides a modern and streamlined API for fine-grain, low-level control of the organization, processing, and submission of graphics and computation commands and the management of the associated data and resources for these commands. A primary goal of Metal is to minimize the CPU overhead necessary for executing these GPU workloads.

Metal框架支持GPU加速的3D图形渲染和并行数据计算。Metal提供了一系列的API用于在细粒度和低层次上组织、处理、提交图形绘制指令、并行计算指令，并管理和这些指令相关联的数据和资源。Metal的一个主要目标是减少GPU执行这些计算的必要的开销。

At a Glance
概述

This document describes the fundamental concepts of Metal: the command submission model, the memory management model, and the use of independently compiled code (i.e., graphics shader functions or data-parallel computation functions). The document then details how to use the Metal API to write an app that:

sets hardware state necessary for graphics or data-parallel compute workloads,

commits commands for execution by the GPU,

manages memory allocations, including buffer and texture objects,

and manages compiled (or compilable) graphics shader or compute function code written in the Metal shading language.

本文描述Metal的基本概念：计算指令分发模型，内存管理模型，独立编译的代码的使用（比如图形着色器程序或者并行数据计算程序）。接下来本文详述如何使用Metal的API实现app：

为图形渲染或是并行计算设置必要的硬件状态
提交命令到GPU执行
管理内存分配，包括buffer对象和textrue对象
管理编译好的（可编译的）图形着色程序或是用Metal着色程序语言写的并行计算程序

How to Use This Document
如何使用本文

This document contains the following articles:

Fundamental Metal Concepts (page 9) briefly describes the main features of Metal.
Command Organization and Execution Model (page 10) explains creating, committing, and executing commands on the GPU.
Resource Objects: Buffers and Textures (page 18) discusses the management of device memory, including buffer and texture objects that represent GPU memory allocations.
Functions and Libraries (page 26) describes how Metal shading language code can be represented in an Metal app, and how Metal shading language code is loaded onto and executed by the GPU.

Graphics Rendering: Render Command Encoder (page 31) describes how to render 3D graphics, including how to distribute graphics operations across multiple threads.
Data-Parallel Compute Processing: Compute Command Encoder (page 57) explains how to perform data-parallel processing.
Buffer and Texture Operations: Blit Command Encoder (page 62) describes how to copy data between textures and buffers.
Metal Tips and Techniques (page 64) lists development tips, such as how to build libraries off-line with compiled code.

本文包括以下章节

1. Metal基础概念： 简要描述Metal的主要特性。
2. 命令编组和执行模型：解释GPU命令如何创建、提交、执行命令。
3. 资源对象（Buffer和Texture）：描述如何管理设备内存，包括对应着GPU内存分配的buffer和texture对象。
4. 着色程序和库：描述Metal着色程序如何在一个app中表示，以及如何将它载入GPU执行
5. 并行数据计算（Compute Command Encoder）：描述如何进行并行数据计算
6. Buffer和Texture操作（Blit Command Encoder）：描述如何在Buffer和Texture间拷贝数据
7. 诀窍：列举开发诀窍，比如如何使用编译好的代码制作库

Prerequisites
前设

This document assumes the reader is familiar with the Objective-C language and is experienced in programming with OpenGL, OpenCL, or similar APIs.

本文假定读者熟悉Objective-C语言并且有过编写OpenGL、OpengCL、或是其他类似图形绘制程序的经验。

See Also
其他相关文档

The *Metal Framework Reference* is a collection of documents that describes the interfaces in the Metal framework.

The *Metal Shading Language Guide* is a document that specifies the Metal shading language, which is used to write a graphics shader or a compute function that is used by a Metal app.

《Metal框架手册》，该文档描述了Metal框架各个接口。

《Metal着色语言指南》，该文档描述Metal着色语言，用于为一个Metal应用编写图形渲染着色器或是并行计算程序。

Fundamental Metal Concepts
Metal基础概念

Metal provides a single, unified programming interface and language for both graphics and data-parallel computation workloads. Metal enables you to integrate graphics and computation tasks much more efficiently without the need to use separate APIs and shader languages.

The Metal framework provides the following:

Low-overhead interface

Metal is a low-overhead framework for managing, committing, and executing both graphics and compute operations. Metal is designed to eliminate “hidden” performance bottlenecks such as implicit state validation. You get control over the asynchronous behavior of the GPU. You can use multithreading efficiently to create and commit command buffers in parallel.

For more detailed information on Metal command submission, see Command Organization and Execution Model (page 10).

Memory and resource management

The Metal interface describes buffer and texture objects that represent allocations of GPU memory. Texture objects have specified pixel formats and may be used for texture images or attachments.

For more detailed information on Metal memory objects, see Resource Objects: Buffers and Textures (page 18).

Integrated support for both graphics and compute state

The Metal framework is a single tightly integrated interface that performs both graphics and data-parallel compute operations on the GPU. Metal uses the same data structures and resources (such as command buffers, textures, etc.) for both graphics and compute operations.

Metal also has a corresponding shading language to describe both graphics shader and compute functions. The Metal framework and Metal shading language define interfaces for sharing data resources between the runtime interface and the graphics shaders and compute functions. Metal shading language code can be compiled during build time with the app code and then loaded at runtime. Metal also provides support for runtime compilation of Metal shading language code.

For more detailed information on writing apps that use Metal graphics shader and compute functions with Metal framework code, see Functions and Libraries (page 26).

A Metal app cannot execute Metal commands in the background, and a Metal app that attempts this is terminated.

Metal为图形渲染和并行数据计算提供了一个统一编程接口和编程语言。Metal使你可以更高效地将图形渲染和数据计算任务集成在一起，而不需要其他的独立API（比如OpenGl ES）和着色语言（比如GLShader）。

Metal框架提供了以下内容：

一组高效接口：Metal是一个高效的框架，可以用来管理、提交、执行图形渲染和并行计算操作。它被设计来消除“隐蔽”的性能瓶颈比如隐式状态校验。你将可以控制GPU的异步行为，你将可以高效地使用多线程特性来并行创建和提交command buffer。

内存和资源管理机制：Metal接口描述了buffer和texture对象，这两种对象对应这GPU内存分配，Texture对象具有特殊的像素格式可以被用作纹理图片或者附件。

对图形渲染和并行计算的统一支持：Metal框架用一个统一的接口来实现基于GPU的图形渲染和并行数据计算。它使用同样的数据结构和资源（比如command buffer、textrue等等）支持图形和计算。

Metal还定义了一个相应的着色语言来实现图形渲染着色程序和并行计算程序。Metal框架和Metal着色语言在运行时、图形着色器、并行计算程序之间，定义了共享数据资源接口。Meal着色程序代码和app代码一起在工程build时被编译，随后在运行时被加载。Metal还支持着色程序在运行时编译。

Command Organization and Execution Model
指令组织和执行模型

This chapter describes the Metal architecture and the framework for the organization, submission, and execution of commands. In the Metal architecture, the `MTLDevice` protocol defines the interface that represents a single GPU. The `MTLDevice` protocol supports methods for interrogating device properties and for creating other device-specific objects, such as buffers and textures.

A **command queue** consists of a queue of **command buffers**, and a command queue organizes the order of execution of those command buffers. A command buffer contains encoded commands that are intended for execution on a particular device. A **command encoder** appends rendering, computing, and blitting commands onto a command buffer, and those command buffers are eventually committed for execution on the device.

本章描述Metal的和指令组织、提交、执行相关的架构框架。在Metal框架中，`MTLDevice`协议定义的接口描述一个GPU，改协议提供了一系列方法，可以查询设备属性，创建设备相关的对象比如缓存和纹理。

一个`command queue`包含了一系列`command buffers`。`command queue`用于组织它拥有的各个`command buffer`按序执行。一个`command buffers`包含多个被编码的指令，这些指令将在一个特定的设备上执行。一个Encoder可以将绘制、计算、位图传输指令推入一个command buffer，最后这些command buffer将被提交到设备执行。

The `MTLCommandQueue` protocol defines an interface for command queues, primarily supporting methods for creating command buffer objects.
The `MTLCommandBuffer` protocol defines an interface for command buffers and supports methods for creating command encoders, enqueueing command buffers for execution, checking status, and other operations. The `MTLCommandBuffer` protocol supports the following command encoder types, which are interfaces for encoding different kinds of GPU workloads into a command buffer:

The `MTLRenderCommandEncoder` protocol encodes graphics (3D) rendering commands for a single rendering pass,
The `MTLComputeCommandEncoder` protocol encodes data-parallel computation workloads,
The `MTLBlitCommandEncoder` protocol encodes simple copy operations between buffers and textures, and utilities like mipmap generation.

`MTLCommandQueue`协议为指令队列定义了接口，它提供创建command buffer对象的方法。

`MTLCommandBuffer`协议为command buffers定义了接口，提供了创建Encoder、command buffers排队执行、状态检查等方法。`MTLCommandBuffer`协议支持下面的几中Encoder类型，它们被用于为command buffer编码不同的GPU计算任务：

`MTLRenderCommandEncoder`，该类型的Encoder编码在一个绘制pass中执行的图形绘制指令
`MTLComputeCommandEncoder`，该类型的Encoder编码数据并行计算任务
`MTLBlitCommandEncoder`，该类型的Encoder支持在缓存和纹理之间简单的拷贝操作，以及类似mipmap生成操作。

At any point in time, only a single command encoder can be active and append commands into a command buffer. Each command encoder must be ended before another command encoder may be created for use with the same command buffer. (The one exception to the “one active command encoder for each command buffer” rule is `MTLParallelRenderCommandEncoder`, which is discussed in Encoding a Single Rendering Pass Using Multiple Threads (page 55).)

Once all encoding has been completed, the `MTLCommandBuffer` itself must be committed, which marks the command buffer as ready for execution by the GPU. The `MTLCommandQueue` protocol controls when the commands in the committed `MTLCommandBuffer` object are executed, relative to other `MTLCommandBuffer` objects that are already in the command queue.

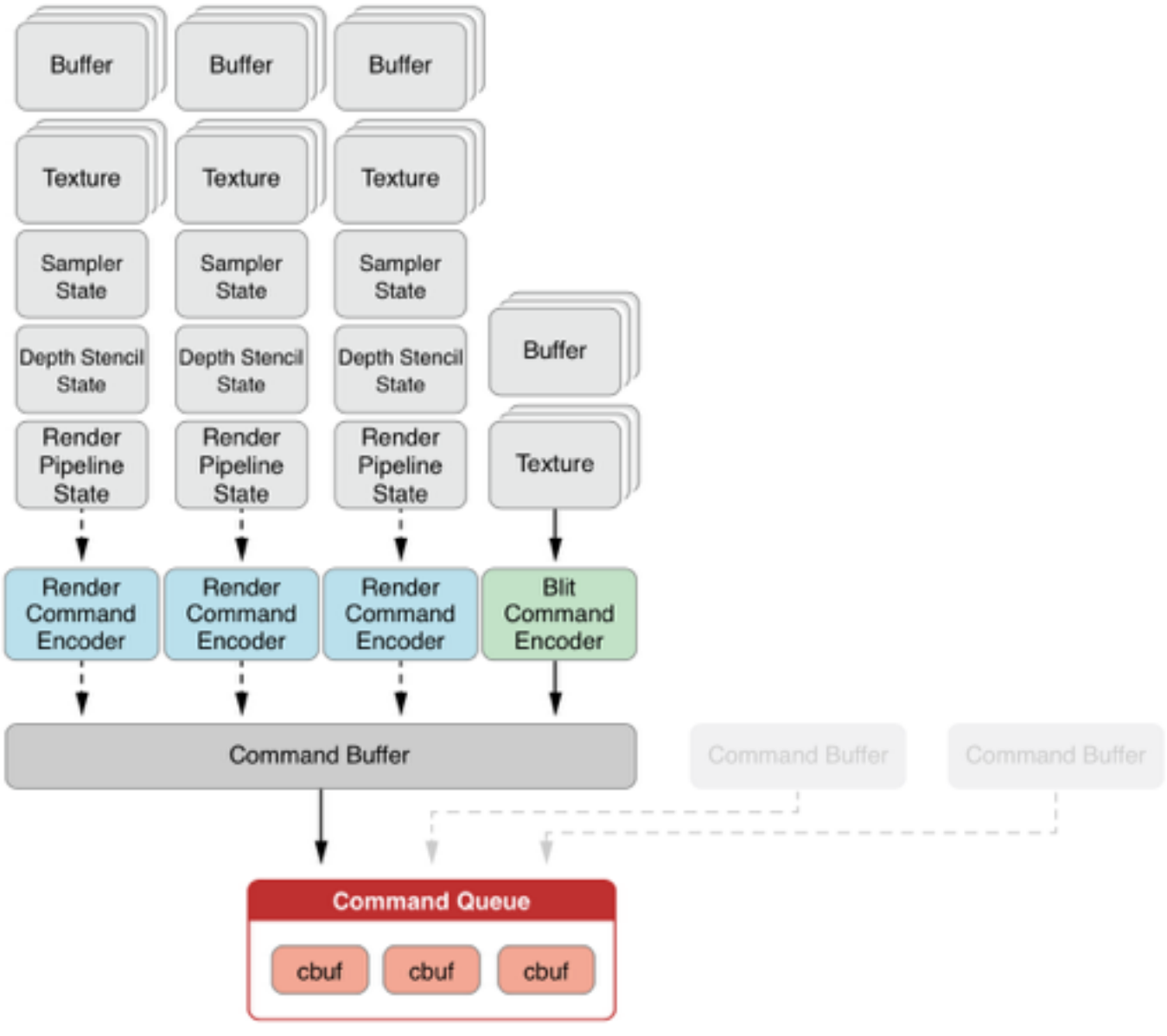
Figure 2-1 (page 11) shows how the command queue, command buffer, and command encoder objects are closely related. The components at the top of the diagram (buffer, texture, sampler, depth/stencil state, pipeline state) represent resources and state that are specific to that command encoder.

任一时刻，只会有一个Encoder是处于激活状态的，它可以向一个command buffer提交指令。前一个Encoder结束后，另一个Encoder才可以被创建并用于同一个command buffer（有一个例外，它不用遵守command buffer中只有一个处于激活状态的Encoder，参见讲述`MTLParallelRenderCommandEncoder`的章节）

一旦所有的指令编码结束，`MTLCommandBuffer`对象自己将被提交，如此它被标记为已经准备好被GPU设备执行。`MTLCommandQueue`协议会控制`MTLCommandBuffer`提交进来的指令的执行，协调已经在commmand queue中的其他`MTLCommandBuffer`提交的指令。

图2-1展示了command queue, command buffer, 和Encoder这几种对象的紧密联系。处于图的顶部的元素（缓存，纹理，采样器，深度/模板状态，绘制管线状态）表示和Encoder相关的资源和状态。

Figure 2-1 Metal Object Relationships



Device—The GPU
设备—GPU

A `MTLDevice` object represents a GPU that can execute commands. The `MTLDevice` protocol has methods to create new command queues, to allocate buffers from memory, to create textures, and to make queries about the device's capabilities. To obtain the preferred system device on the system, call the `MTLCreateSystemDefaultDevice` function.

一个`MTLDevice`对象代表一个可以执行指令的GPU，该协议包含创建新的command queue的方法，从内存申请缓存的方法，创建纹理对象的方法，查询设备功能的方法。调用`MTLCreateSystemDefaultDevice`方法获取系统首选的设备对象。

Transient and Non-Transient Objects in Metal
Metal中得暂态和非暂态对象

Some objects in Metal are designed to be transient and extremely lightweight, while others are more expensive and are intended to last for extended periods of time, perhaps the lifetime of the app.

The following objects are transient and designed for a single use. They are very inexpensive to allocate and deallocate, so their creation methods return autoreleased objects.

Command Buffers Command Encoders

In contrast, the following objects are not transient. You are encouraged to reuse these objects in performance sensitive code and discouraged from repeated object creation.

Command Queues
Buffers
Textures
Sampler States Libraries
Compute States
Render Pipeline States
Depth/Stencil States

在Metal中有些对象被设计成暂态，使用它们非常轻量。另外一些则要昂贵许多，因此它们应该拥有长的生命周期，比如可以是和整个应用相同的生命周期。

下面这些对象是暂态的，被设计来一次性使用，它们的创建和销毁的成本都非常廉价，它们的创建方法都返回autoreleased对象。

Command Buffers
Command Encoders

相反，下面这些对象是非暂态的，在性能相关的代码里应该尽量重用之，避免反复创建：

Command Queues
Buffers
Textures
Sampler States Libraries
Compute States
Render Pipeline States
Depth/Stencil States

Command Queue
指令队列

A command queue accepts an ordered list of command buffers that the GPU will execute. All command buffers sent to a single queue are guaranteed to execute in the order in which the command buffers were enqueued. In general, command queues are thread safe and allow multiple active command buffers to be encoded simultaneously.

To create a command queue, call either the `newCommandQueue` method or the `newCommandQueueWithMaxCommandBufferCount:` method of a `MTLDevice` object. In general, command queues are expected to be long-lived, so they should not be repeatedly created and destroyed.

一个command queue管理着一个有序的将要在GPU上执行的command buffer队列。所有被塞进同一个队列的command buffer都会被按照它进入队列的次序执行。通常command queue是线程安全的，允许多个command buffer同时编码。

`MTLDevice`的`newCommandQueue`方法、`newCommandQueueWithMaxCommandBufferCount`方法都可以创建一个command queue。通常情况下，command queue应该是具有长生命周期的，不要反复地创建和销毁这种对象。

Command Buffer
指令缓存

A command buffer stores encoded commands until it is committed for execution by the GPU. A single command buffer may contain many different kinds of encoded commands, depending on the number and type of encoders that are used to build it. In a typical app, an entire "frame" of rendering may be encoded into a single command buffer, even if multiple rendering passes, compute processing functions, or blit operations are all part of the same frame.

Command buffers are considered transient single-use objects and do not support reuse. Once a command buffer has been committed for execution, the only valid operations are to wait for the command buffer to be scheduled or completed (either via synchronous calls or handler blocks discussed in Registering Handler Blocks for Command Buffer Execution (page 14)) and to check the status of the command buffer execution.

Command buffers also represent the only independently trackable unit of work by the app, as well as define the coherency boundaries established by the Metal memory model, as detailed in Resource Objects: Buffers and Textures (page 18).

一个command buffer在被GPU执行之前会存储多个被编码的指令。一个command buffer可以包含多种类型的编码指令，这依赖于有几个有几种Encoder被用来创建command buffer。一个典型的app，一帧画面的绘制操作可以被编码到一个command buffer中，就算这帧画面是需要多个绘制pass，多个计算处理着色程序，或者多个位图操作才能完成。

command buffer被设计为暂态的一次性使用的对象，它不支持重用。一旦一个commad buffer被提交执行，接下来唯一有效的操作就是等待它被排入队列或是结束（同步阻塞式调用 和 基于block的调用在后面的章节会讲述到）然后检查它被执行的状态。

command buffer还代表了app中独立可被追踪的任务单元，它还定义了Metal的内存模型确立的一致性边界，在下一个章节中会具体讲述到。

Creating a Command Buffer
创建Command Buffer

To create a `MTLCommandBuffer` object, call the `commandBuffer` method of `MTLCommandQueue`. An `MTLCommandBuffer` can only be committed into the `MTLCommandQueue` object that created it.

Command buffers created by the `commandBuffer` method hold a retain on data that is needed for execution. For certain scenarios, where you hold a retain to these objects elsewhere for the duration of the execution of a `MTLCommandBuffer` object, you can instead create a command buffer by calling the `commandBufferWithUnretainedReferences` method of `MTLCommandQueue`. The `commandBufferWithUnretainedReferences` method should only be used for extremely performance-critical apps that can guarantee that crucial objects have references elsewhere in the app until command buffer execution is completed. Otherwise, an object that no longer has other references may be prematurely released, and the results of the command buffer execution are undefined.

调用`MTLCommandQueue`的`commandBuffer`方法创建一个`MTLCommandBuffer`对象。一个`MTLCommandBuffer`对象只能提交给创建它的那个command queue。由`commandBuffer`方法创建的command buffer对象将持有在它被执行时需要用到的数据。某些情况下，如果不需要强引用这些相关数据，可以调用`MTLCommandQueue`的`commandBufferWithUnretainedReferences`方法。在可以保证和command buffer相关数据在其被执行时都有引用计数的情况下，又极端需要提升性能，才使用该方法。否则，一个command buffer相关的资源对象可能会因为没有引用计数而被释放，command buffer的执行结果就不可预料了。

Executing Commands
执行指令

The `MTLCommandBuffer` protocol has the following methods to establish the execution order of command buffers in the command queue. A command buffer does not begin execution until it is committed. Once committed, command buffers are executed in the order in which they were enqueued.

The `enqueue` method reserves a place for the command buffer on the command queue, but does not commit the command buffer for execution. When this command buffer is eventually committed, it is executed after any previously enqueued command buffers within the associated command queue.

The `commit` method causes the command buffer to be executed as soon as possible, but after any previously enqueued command buffers in the same command queue have been committed. If the command buffer has not previously been enqueued, `commit` makes an implied `enqueue` call.

(For an example of using `enqueue` with multiple threads, see Multiple Threads, Command Buffers, and Command Encoders (page 16).)

MTLCommandBuffer协议有如下的方法来设定其在command queue中的执行顺序。一个command buffer一定要先提交然后再执行，一旦提交，它按照被排入队列的顺序执行。

enqueue方法为一个command buffer在command queue中预定一个位置，但是不会提交这个command buffer。当这个command buffer最终被提交，command queue把它安排在之前做enqueue操作的command buffer之后执行。

commit方法使得command buffer尽可能快地被执行，但是还是得等到所有在command queue中的早前排入队列的command buffer被执行完成后才能执行。如果commad queue中没有排在前面的，该方法隐式执行插入队列操作。

Registering Handler Blocks for Command Buffer Execution
为command buffer的执行注册处理程序块

The `MTLCommandBuffer` methods listed below monitor command execution. When used, scheduled and completed handlers are invoked in execution order on an undefined thread. These handlers should be performed quickly; if expensive or blocking work needs to be done, then you should defer that work to another thread.

The `addScheduledHandler:` method registers a block of code to be called when the command buffer is scheduled. A command buffer is considered *scheduled* when any dependencies between work submitted by other `MTLCommandBuffer` objects or other APIs in the system has been satisfied. Multiple scheduled handlers may be registered for a command buffer.

The `waitUntilScheduled` method synchronously waits and returns after the command buffer is scheduled and all handlers registered by the `addScheduledHandler:` method are completed.

The `addCompletedHandler:` method registers a block of code to be called immediately after the device has completed the execution of the command buffer. Multiple completed handlers may be registered for a command buffer.

The `waitUntilCompleted` method synchronously waits and returns after the device has completed the execution of the command buffer and all handlers registered by the `addCompletedHandler:` method have returned.

The `presentDrawable:` method is a convenience method that presents the contents of a displayable resource (`CAMetalDrawable` object) when the command buffer is scheduled. For details about the `presentDrawable:` method, see Integration with Core Animation: CAMetalLayer (page 36).

The read-only `status` property contains a `MTLCommandBufferStatus` enum value listed in `Command Buffer Status Codes` that reflects the current scheduling stage in the lifetime of this command buffer.

If execution finishes successfully, the value of the read-only `error` property is `nil`. If execution fails, then `status` is set to `MTLCommandBufferStatusError`, and the `error` property may contain a value listed in `Command Buffer Error Codes` that indicates the cause of the failure.

下列的MTLCommandBuffer方法可以监视指令的执行。使用了这些方法注册处理程序块，那么在某个线程中，这些处理程序块会按照执行顺序被调用。这些处理程序块应该是迅速可被执行完成的，如果有开销大的造成阻塞的任务需要执行，那么应该将它们安排到其他线程执行。

addScheduledHandler:，该方法注册的处理程序块将在command buffer被排定好时调用。所谓排定好，是指当所有MTLCommandBuffer对象或是系统API提交的任务之间的依赖被满足，一个command buffer才被认为是排定好。一个command buffer对象可以为“排定好”注册多个处理程序块。
waitUntilScheduled，该方法调用后开始等待并在command buffer排定好且所有通过addScheduledHandler方法注册的处理程序块都执行完毕后返回。

addCompletedHandler:，该方法注册的处理程序块将在command buffer被执行完毕后立即调用,一个command buffer对象可以为“执行完”注册多个处理程序块。
waitUntilCompleted，该方法调用后开始等待，一直到command buffer被执行完毕并且所有的通过上面这个addCompletedHandler方法注册的处理程序块都执行完毕后才返回。

presentDrawable:，该方法是一个便捷方法，它用于当command buffer处于排定好时呈现一个可显示资源（CAMetalDrawable类型对象）的内容。

status，是command buffer的一个只读属性，包含了一个MTLCommandBufferStatus类型的枚举值，它反映了command buffer在其生命周期中处于那个阶段。

error，如果command buffer成功执行，这个属性值为nil。如果有异常发生，那么上面一行描述的status属性被设置为MTLCommandBufferStatusError，同时error属性将包含一个列举在“Command Buffer Error Codes”中的值，这个值指示了错误原因。

Command Encoder

A command encoder is a transient object that is used once to write commands and state into a single command buffer in a format that the GPU can execute. Many command encoder object methods append commands for the command buffer. While a command encoder is active, it has the exclusive right to append commands for its command buffer. Once you have finished encoding commands, call the `endEncoding` method. To write further commands, a new command encoder can be created.

Encoder是一个一次性使用的暂态对象，它用来编码计算指令和绘制状态，然后它被推入到一个command buffer并最终在GPU上执行。有很多的Encoder方法可以把指令推入command buffer。当一个Encoder是激活状态时，它就可以调用endEncoding方法向它所属的command buffer推送指令。如果要写入更多的指令，就创建一个新的Encoder。

Creating a Command Encoder Object
创建一个Encoder对象

The `MTLCommandBuffer` protocol has the following methods to create one of those types of command encoder objects that will append commands into the originating command buffer:

The `renderCommandEncoderWithDescriptor:` method creates a `MTLRenderCommandEncoder` for graphics rendering to an attachment in a `MTLRenderPassDescriptor`.

The `computeCommandEncoder` method creates a `MTLComputeCommandEncoder` for data-parallel computations.

The `blitCommandEncoder` method creates a `MTLBlitCommandEncoder` for memory operations.

The `parallelRenderCommandEncoderWithDescriptor:` method creates a `MTLParallelRenderCommandEncoder` that enables several `MTLRenderCommandEncoder` objects to run on different threads while still rendering to an attachment that is specified in a shared `MTLRenderPassDescriptor`.

MTLCommandBuffer协议有如下方法来创建各种Encoder对象，这些对象可以向创建它们出来这个的command buffer推送计算指令：

renderCommandEncoderWithDescriptor:，该方法创建一个MTLRenderCommandEncoder类型的Encoder来实现图形绘制，图形绘制用到的attachment由那个MTLRenderPassDescriptor类型的入参对象指定。
computeCommandEncoder，该方法创建一个MTLComputeCommandEncoder类型的Encoder来实现并行数据计算。
blitCommandEncoder，该方法创建一个MTLBlitCommandEncoder类型的Encoder来实现内存操作。
parallelRenderCommandEncoderWithDescriptor:,该方法创建一个MTLParallelRenderCommandEncoder类型的Encoder，它用于支持多个MTLRenderCommandEncoder类型的子Encoder同时在不同的线程中运行，但是依然把绘制结果写入共享的由入参MTLRenderPassDescriptor指定的attachment中。

Render Command Encoder
绘制用的Encoder

Graphics rendering can be described in terms of a *rendering pass* . A `MTLRenderCommandEncoder` represents the rendering state and drawing commands associated with a single rendering pass. A `MTLRenderCommandEncoder` requires an associated `MTLRenderPassDescriptor` (described in Creating a Render Pass Descriptor (page 32)) that includes the color, depth, and stencil attachments that serve as destinations for rendering commands. The `MTLRenderCommandEncoder` has methods to:

specify graphics resources, such as buffer and texture objects, that contain vertex, fragment, or texture image data,

specify a `MTLRenderPipelineState` object that contains compiled rendering state, including vertex and fragment shaders,

specify fixed-function state, including viewport, triangle fill mode, scissor rectangle, depth and stencil tests, and other values, and draw 3D primitives.

For detailed information about the `MTLRenderCommandEncoder` protocol, see Graphics Rendering: Render Command Encoder (page 31).

图形绘制可以被描述为一系列的绘制pass，一个`MTLRenderCommandEncoder`对象表示和一个绘制pass相关联的绘制状态和绘制命令。这个Encoder对象需要一个`MTLRenderPassDescriptor`对象，在这个descriptor对象中包含了颜色、深度、模板attachment，这些attachment将被当做绘制命令的目标结果，Encoder拥有下面这些方法：
设定图形资源，比如缓存和纹理对象，这些对象包含着顶点片元和纹理数据。
设定固定图形绘制管线状态，包括视口，三角形填充模式，裁剪矩形，深度和模板测试等等。
绘制图元。

Compute Command Encoder
并行计算用的Encoder

For data-parallel computing, the `MTLComputeCommandEncoder` protocol provides methods to encode commands in the command buffer that can specify the compute function and its arguments (e.g., texture, buffer, or sampler state) and dispatch the compute function for execution. To create a compute command encoder object, use the `computeCommandEncoder` method of `MTLCommandBuffer`. For detailed information about the `MTLComputeCommandEncoder` methods and properties, see Data-Parallel Compute Processing: Compute Command Encoder (page 57).

对于并行数据计算，协议提供了方法来编码计算指令到command buffer，设定计算程序和参数，调度计算程序执行。使用`MTLCommandBuffer`的`computeCommandEncoder`方法可以创建一个并行计算用的Encoder。

Blit Command Encoder
位图操作用的Encoder

The `MTLBlitCommandEncoder` protocol has methods that append commands for memory copy operations between buffers (`MTLBuffer`) and textures (`MTLTexture`). The `MTLBlitCommandEncoder` protocol also provides methods to fill textures with a solid color and to generate mipmaps. To create a blit command encoder object, use the `blitCommandEncoder` method of `MTLCommandBuffer`. For detailed information about the `MTLBlitCommandEncoder` methods and properties, see Buffer and Texture Operations: Blit Command Encoder (page 62).

`MTLBlitCommandEncoder`协议提供了方法用来在缓存(`MTLBuffer`) 和 纹理(`MTLTexture`)之间进行拷贝。该协议还提供了用一种颜色填充纹理的方法，以及创建mipmap的方法。使用的方法创建一个位图操作用的Encoder。
将在“缓存和纹理操作：位图指令Encoder”章节详细描述其方法和属性。

Multiple Threads, Command Buffers, and Command Encoders
多线程，command buffer 以及 Encoder

Most apps use a single thread to encode the rendering commands for a single frame in a single command buffer. At the end of each frame, you commit the command buffer, which both schedules and begins command execution.

If you want to parallelize command buffer encoding, then you can create multiple command buffers at the same time, and encode to each one with a separate thread. If you know ahead of time in what order a command buffer should execute, then the `enqueue` method of `MTLCommandBuffer` can declare the execution order within the command queue without needing to wait for the commands to be encoded and committed. Otherwise, when a command buffer is committed, it is assigned a place in the command queue after any previously enqueued command buffers.

很多的应用程序只是用一个线程来编码绘制指令到一个command buffer来绘制一帧画面。在每帧绘制的末尾，提交commad buffer，如此可以安排和开始指令的执行。

如果你希望并行为command buffer执行指令编码，那么可以同时创建多个command buffer，使用多个线程，每个线程单独为一个command buffer编码指令。如果你事先知道command buffer应该以如何的顺序执行，那么`MTLCommandBuffer`的`enqueue`方法可以在command queue中声明执行顺序，而不必等待执行编码和提交操作。要不然就只有等到command buffer被提交，这时在command queue中它就被指定了一个位置，位于所有先提交的command buffer后面。

Only one CPU thread can access a command buffer at time. Multithreaded apps can use one thread per command buffer to create multiple command buffers in parallel.

Figure 2-2 (page 17) shows an example with three threads. Each thread has its own command buffer. For each thread, one command encoder at a time has access to its associated command buffer. Figure 2-2 (page 17) also shows each command buffer receiving commands from different command encoders. When you are done encoding, call the `endEncoding` method of the command encoder, and a new command encoder object can then begin encoding commands to the command buffer.

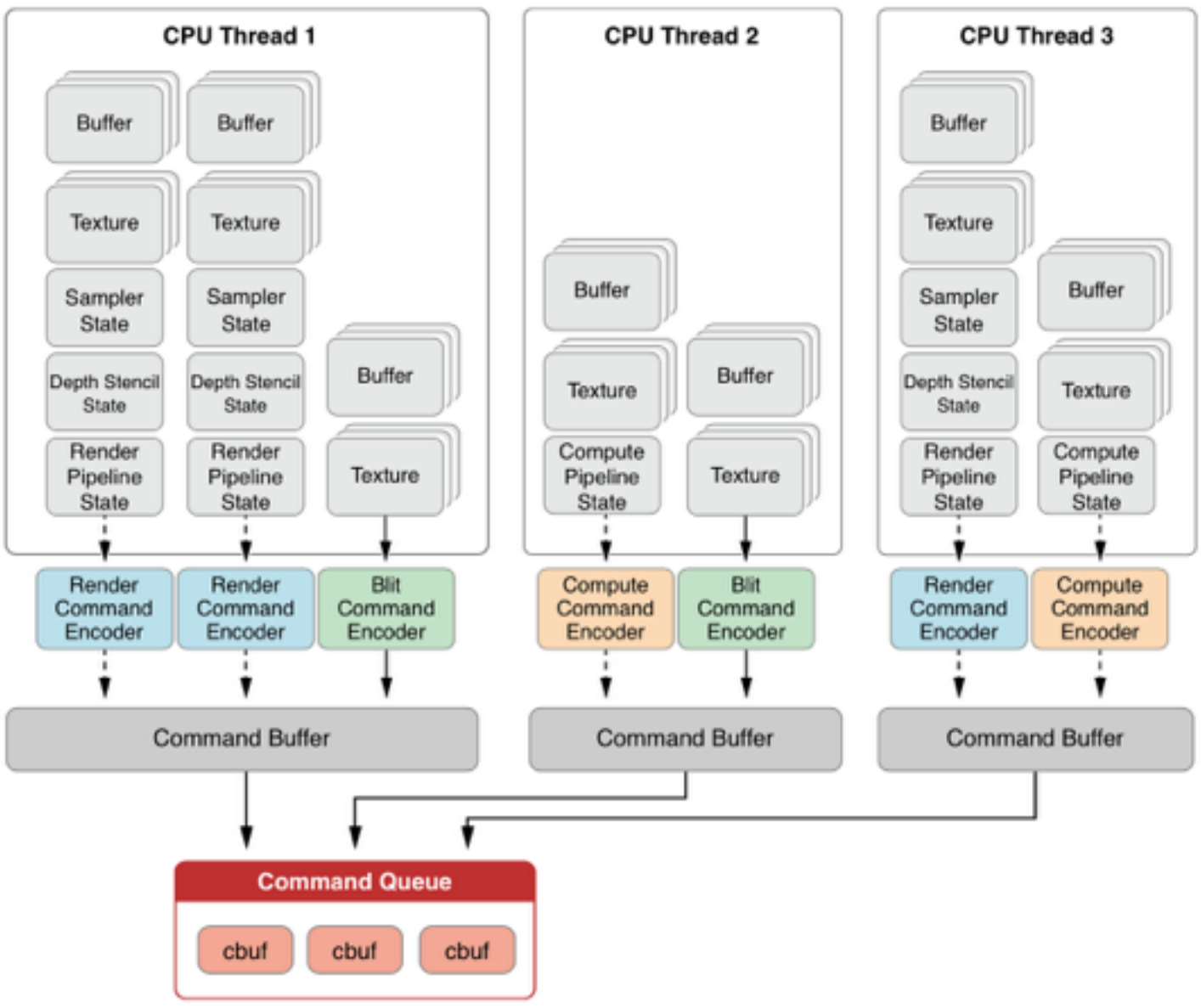
A `MTLParallelRenderCommandEncoder` object allows a single rendering pass to be broken up across multiple command encoders and assigned to separate threads. For more information about `MTLParallelRenderCommandEncoder`, see Encoding a Single Rendering Pass Using Multiple Threads (page 55).

任一时刻只有一个GPU线程在访问command buffer，多线程的应用可以为每个command buffer准备一个线程，如此实现并行创建多个command buffer。

图2-2示例了一个3线程应用，每个线程都操作一个command buffer，每个线程中，任一时刻只有一个Encoder在访问它对应的command buffer。当你完成了编码，调用Encoder的`endEncoding`方法，然后一个新的Encoder才可以开始为command buffer编码指令。

一种`MTLParallelRenderCommandEncoder`对象可以允许一个绘制pass被打散成多个子Encoder并分配到多个线程进行操作。

Figure 2-2 Metal Command Buffers with Multiple Threads



Resource Objects: Buffers and Textures
资源对象：缓存和纹理

This chapter describes Metal resource objects (`MTLResource`) for storing unformatted memory and formatted image data. There are two types of `MTLResource` objects:

`MTLBuffer` represents an allocation of unformatted memory that can contain any type of data. Buffers are often used for vertex, shader, and compute state data.

`MTLTexture` represents an allocation of formatted image data with a specified texture type and pixel format. Texture objects are used as source textures for vertex, fragment, or compute functions, as well as to store graphics rendering output (i.e., an attachment).

`MTLSamplerState` objects are also discussed in this chapter. Although samplers are not resources themselves, they are used when performing lookup calculations with a texture object.

本章节描述Metal的资源对象(`MTLResource`)，这些对象可以存储非格式化的内存数据 或是 格式化的图像数据，有下面两种`MTLResource`类型的对象。

`MTLBuffer`，它表示一块非格式化的内存可以存放任何类型的数据。它通常用于存放顶点数据，着色和计算状态数据。
`MTLTexture`，它表示一块有格式的图像数据，有特定的纹理类型和像素格式。它通常被用作顶点着色程序，片元着色程序，或是并行计算着色程序的纹理源。它也可以存放图形绘制的结果输出。

`MTLSamplerState`类型的对象也将在这个章节讲述，虽然采样器不是资源，但它总是用于纹理对象的查找计算。

Buffers are Typeless Allocations of Memory
缓存是无类型的内存片段

A `MTLBuffer` object represents an allocation of memory that can contain any type of data.

一个`MTLBuffer`对象表示了一个可以装载任一类型数据的内存片段。

Creating a Buffer Object
创建一个缓存对象

The following `MTLDevice` methods create and return a `MTLBuffer` object:

The `newBufferWithLength:options:` method creates a `MTLBuffer` object with a new storage allocation.

The `newBufferWithBytes:length:options:` method creates a `MTLBuffer` object by copying data from existing storage (located at the CPU address `pointer`) into a new storage allocation.

The `newBufferWithBytesNoCopy:length:options:deallocator:` method creates a `MTLBuffer` object with an existing storage allocation and does not allocate any new storage for this object.

All buffer creation methods have the input value `length` to indicate the size of the storage allocation, in bytes. All the methods also accept a `MTLResourceOptions` object for `options` that can modify the behavior of the created buffer. If the value for `options` is 0, the default values are used for resource options.

下面的`MTLDevice`方法用户创建并返回类型`MTLBuffer`对象：

`newBufferWithLength:options:`，该方法新分配内存创建一个`MTLBuffer`对象。
`newBufferWithBytes:length:options:`，该方法新分配内存创建一个`MTLBuffer`对象，并且从已有的存储（由CPU内存地址指针指定）拷贝数据到新分配的内存中。
`newBufferWithBytesNoCopy:length:options:deallocator:`，该方法创建一个`MTLBuffer`对象，但是不会为这个缓存对象分配新内存，而是使用一个已经存在的内存。

以上所有的创建方法都有叫做length的入参，它用来指定分配存储的多少（按byte）；他们还有叫做的options入参，接受一个MTLResourceOptions类型的值，用来设定缓存创建的行为。options的默认值为0。

Buffer Methods
缓存对象方法

The `MTLBuffer` protocol has the following methods:

The `contents` method returns the CPU address of the buffer's storage allocation.
The `newTextureWithDescriptor:offset:bytesPerRow:` method creates a special kind of texture

object that references the buffer's data. This method is detailed in Creating a Texture Object (page 19).

缓存对象有如下的方法：

contents，该方法返回缓存对象对应的内存的CPU地址。
newTextureWithDescriptor:offset:bytesPerRow:，该方法创建某种特定类型的纹理。

Textures are Formatted Image Data
纹理是格式化的图像数据

A `MTLTexture` object represents an allocation of formatted image data that can be used as a resource for a vertex shader, fragment shader, or compute function, or as an attachment to be used as a rendering destination. A `MTLTexture` object can have one of the following structures:

a 1D, 2D, or 3D image, arrays of 1D or 2D images, or a *cube* of six 2D images.

`MTLPixelFormat` specifies the organization of individual pixels in a `MTLTexture` object. Pixel formats are discussed further in Pixel Formats for Textures (page 23).

一个MTLTexture对象代表了一个格式化的图像数据内存片段，它在顶点着色程序、片元着色程序、并行计算着色程序中被用作输入源，或是作为绘制操作的目标输出。一个对象有如下结构：

一个1维，2维，或是3维图像
一个含有1维或是2维的图像的数组
拥有6个2维图像的立方体

MTLPixelFormat，该属性描述了一个MTLTexture对象中的每个单独像素的组织排列。

Creating a Texture Object
创建纹理对象

The following methods create and return a `MTLTexture` object:

The `newTextureWithDescriptor:` method of `MTLDevice` creates a `MTLTexture` object with a new storage allocation for the texture image data, using a `MTLTextureDescriptor` object to describe the texture's properties.

The `newTextureViewWithPixelFormat:` method of `MTLTexture` creates a `MTLTexture` object that shares the same storage allocation as the calling `MTLTexture` object. Since they share the same storage, any changes to the pixels of the new texture object are reflected in the calling texture object, and vice versa. For the newly created texture, the `newTextureViewWithPixelFormat:` method reinterprets the existing texture image data of the storage allocation of the calling `MTLTexture` object as if the data was stored in the specified pixel format. The `MTLPixelFormat` of the new texture object must be *compatible* with the `MTLPixelFormat` of the original texture object. (See Pixel Formats for Textures (page 23) for details about the ordinary, packed, and compressed pixel formats.)

The `newTextureWithDescriptor:offset:bytesPerRow:` method of `MTLBuffer` creates a `MTLTexture` object that shares the storage allocation of the calling `MTLBuffer` object as its texture image data. As they share the same storage, any changes to the pixels of the new texture object are reflected in the calling texture object, and vice versa. Sharing storage between a texture and a buffer may prevent applying certain texturing optimizations (e.g., pixel swizzling or tiling).

下列的方法用于创建并返回一个MTLTexture对象

newTextureWithDescriptor:，这是一个MTLDevice方法，该方法新分配内存创建一个MTLTexture对象，创建时需要传入的MTLTextureDescriptor类型的参数描述了纹理的属性。

newTextureViewWithPixelFormat:，不同于上一个方法，这是一个MTLTexture方法，该方法创建出来的MTLTexture对象和调用源对象共享存储。因为共享存储，所以新创建出来的对象的像素发生了任何改变，调用源对象的像素也就随之变化，反之亦然。对于新建出来的纹理对象，该方法从新解释了调用源纹理对象对应的内存中的图像数据是一种什么格式。MTLPixelFormat类型的入参，必须和调用源纹理对象的像素格式属性相适配。

newTextureWithDescriptor:offset:bytesPerRow:，这是一个MTLBuffer方法，它创建一个MTLTexture对象，共享调用源对象的内存，作为它自己的图像数据。因为他们共享存储，所以任何发生在新建纹理对象上的像素改变都将随之反映到调用源对象，反之亦然。纹理对象和缓存对象共享内存将阻碍实施某些纹理优化操作（比如像素混合或分片）

Texture Descriptors
纹理descriptor

`MTLTextureDescriptor` defines the properties that are used to create a `MTLTexture` object, including its image size (width, height, and depth), pixel format, arrangement (array or cube type) and number of mipmaps. The `MTLTextureDescriptor` properties are only used during the creation of a `MTLTexture` object. After a `MTLTexture` object has been created, property changes in its `MTLTextureDescriptor` object no longer have any effect on that texture.

MTLTextureDescriptor描述用于创建一个MTLTexture对象的各属性。包括图形尺寸（宽，高，深），像素格式，组合模式（数组或是立方体）还有mipmaps的数量。这些属性都只用在MTLTexture对象的创建过程中。当纹理对象创建完毕，descritor中的值再改变不会影响纹理之前由它创建的对象。

First you create a custom `MTLTextureDescriptor` object that contains texture properties that include type, size (width, height, and depth), pixel format, and whether mipmaps are used:

The `textureType` property specifies a texture's dimensionality and arrangement (i.e., array or cube).
The `width`, `height`, and `depth` properties specify the pixel size in each dimension of the base level texture mipmap.
The `pixelFormat` property specifies how a pixel is stored in a texture.
The `arrayLength` property specifies the number of array elements for a `MTLTextureType1DArray` or `MTLTextureType2DArray` type texture object.
The `mipmapLevelCount` property specifies the number of mipmap levels.
The `sampleCount` property specifies the number of samples in each pixel.
The `resourceOptions` property specifies the behavior of its memory allocation.

首先创建一个MTLTextureDescriptor对象，它包含各种纹理属性，类型，尺寸（宽，高，深），像素格式，以及是否使用mipmap：

textureType，该属性用来指定纹理的维度 和 组合模式（数组 或是 立方）。
width,height,depth，这些属性指定了基层mipmap纹理在各个维度的尺寸。
pixelFormat，指定了一个纹理中的每个像素如何存储。
arrayLength，对于MTLTextureType1DArray类型或是MTLTextureType2DArray类型的纹理对象，这个属性指定数组中的元素的数量。
mipmapLevelCount，指定mipmap类型纹理的层数。
sampleCount，指定每个像素的样本数。
resourceOptions，指定内存分配的行为方式。

When creating a new texture with the `newTextureWithDescriptor:` method of `MTLDevice`, you provide a `MTLTextureDescriptor` object. After texture creation, call replaceRegion:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage: to load the texture image data, as detailed in Copying Image Data to and from a Texture (page 22).

You can reuse the descriptor object `MTLTextureDescriptor` to create more `MTLTexture` objects, modifying the descriptor's property values as needed.

Listing 3-1 (page 21) shows code to create a texture descriptor `txDesc` and set its properties for a 3D, 64x64x64 texture.

使用MTLDevice的newTextureWithDescriptor方法创建一个新的纹理对象，首先要提供一个MTLTextureDescriptor对象。纹理创建完毕后，调用replaceRegion:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage: 载入纹理图像数据。

之前创建的MTLTextureDescriptor对象，在做必要的修改后可以继续用来创建新的纹理对象。

代码 3-1 展示了一段代码，创建纹理descriptor变量txDesc，然后设置它为3维纹理，长宽高都是64.

Listing 3-1 Creating a Texture Object with a Custom Texture Descriptor

```
MTLTextureDescriptor* txDesc = [MTLTextureDescriptor new];
txDesc.textureType = MTLTextureType3D;
txDesc.height = 64;
txDesc.width = 64;
txDesc.depth = 64;
txDesc.pixelFormat = MTLPixelFormatBGRA8Unorm;
txDesc.arrayLength = 1;
txDesc.mipmapLevelCount = 1;
id <MTLTexture> aTexture = [device newTextureWithDescriptor:txDesc];
```

Slices
纹理切片

A *slice* is a single 1D, 2D, or 3D texture image and all its associated mipmaps. For each slice:

The size of the base level mipmap is specified by the `width`, `height`, and `depth` properties of the `MTLTextureDescriptor` object.

The scaled size of mipmap level *i* is specified by max(1, floor(`width` / 2^i)) x max(1, floor(`height` / 2^i)) x max(1, floor(`depth` / 2^i)). The maximum mipmap level is the first mipmap level where the size 1 x 1 x 1 is achieved.

The number of mipmap levels in one slice can be determined by floor($\log_2$(max(`width`, `height`, `depth`)))+1.

All texture objects have at least one slice; cube and array texture types may have several slices. In the methods that write and read texture image data that are discussed in Copying Image Data to and from a Texture (page 22), `slice` is a zero-based input value. For a 1D, 2D, or 3D texture, there is only one slice, so the value of `slice` must be 0. A cube texture has six total 2D slices, addressed from 0 to 5. For the 1DArray and 2DArray texture types, each array element represents one slice. For example, for a 2DArray texture type with `arrayLength` = 10, there are 10 total slices, addressed from 0 to 9. To choose a single 1D, 2D, or 3D image out of an overall texture structure, first select a slice, and then select a mipmap level within that slice.

一个纹理分片是一个单独的1维、2维或是3维纹理数据以及和它关联的mipmap，对于每一个切片：

其基层的mipmap的尺寸由MTLTextureDescriptor对象的width ，height，depth属性设定。

mipmap的第 i 层的尺寸计算公式是 width = max(1, floor(`width` / 2^i))；height=max(1, floor(`height` / 2^i))；depth=max(1, floor(`depth` / 2^i))。mipmap的层数的最大值，是当width = height= depth=1时，i的值。

在一个切片中的mipmap的层数的计算公式为 floor($\log_2$(max(`width`, `height`, `depth`)))+1

所有的纹理对象都至少有一个切片；立方和数组类型的纹理可以有多个分片，在读写纹理的图像数据的方法里，slice是一个0base的值，对于1维、2维或是3维纹理，它们只有一个切片，所以其slice值为0.一个立方纹理有6个2维的切片，地址从0到5.对于1DArray和2DArray类型的纹理，每个数组元素代表一个切片。比如一个arrayLength值为10的2DArray类型的纹理，它拥有10个切片，地址从0到9。从一个纹理的数据结构中选择一个单独的图像（1维、2维或是3维）首先要指定一个切片，然后在选择切片上的mipmap层。

Texture Descriptor Convenience Creation Methods
纹理descriptor的快捷创建方法

For common 2D and cube textures, the following convenience methods create a `MTLTextureDescriptor` object that automatically set several of the properties as follows:

The `texture2DDescriptorWithPixelFormat:width:height:mipmapped:` method creates a `MTLTextureDescriptor` object for a 2D texture. `width` and `height` define the dimensions of the 2D texture. The `type` property is automatically set to `MTLTextureType2D`, and `depth` and `arrayLength` are set to 1.

The `textureCubeDescriptorWithPixelFormat:size:mipmapped:` method creates a `MTLTextureDescriptor` object for a cube texture, where the `type` property is set to `MTLTextureTypeCube`, `width` and `height` are set to size, and `depth` and `arrayLength` are set to 1.

Both `MTLTextureDescriptor` convenience methods accept an input value, `pixelFormat`, which defines the pixel format of the texture. Both methods also accept the input value `mipmapped`, which determines whether or not the texture image is mipmapped. (If `mipmapped` is YES, the texture is mipmapped.)

Listing 3-2 (page 22) uses the `texture2DDescriptorWithPixelFormat:width:height:mipmapped:` method to create a descriptor object for a `64x64`, non-mipmapped 2D texture.

对一个2维纹理或是立方纹理来说，下面的快捷方法创建MTLTextureDescriptor对象并且自动设置多个值：

texture2DDescriptorWithPixelFormat:width:height:mipmapped:，该方法创建一个描述2维纹理的MTLTextureDescriptor对象，入参width和height定义2维纹理的尺寸，descriptor的type属性自动设置为MTLTextureType2D，属性depth和属性arrayLength自动设置为1。

textureCubeDescriptorWithPixelFormat:size:mipmapped:,该方法创建一个描述立方纹理的MTLTextureDescriptor对象，入参size被设置给width和height，type属性自动设置为MTLTextureTypeCube，属性depth和属性arrayLength自动设置为1

上面两个快捷方法都接受一个入参pixelFormat，它定义了纹理的像素格式。这两个方法都接受一个入参mipmapped，它指定纹理是否支持mipmap。

代码3-2 使用texture2DDescriptorWithPixelFormat:width:height:mipmapped:方法创建了一个64x64的不支持mipmap的2维纹理对象

Listing 3-2 Creating a Texture Object with a Convenience Texture Descriptor

```
MTLTextureDescriptor *texDesc = [MTLTextureDescriptor
        texture2DDescriptorWithPixelFormat:MTLPixelFormatBGRA8Unorm
        width:64 height:64 mipmapped:NO];
id <MTLTexture> myTexture = [device newTextureWithDescriptor:texDesc];
```

Copying Image Data to and from a Texture
拷贝图像数据进出纹理对象

To synchronously copy image data into or copy data from the storage allocation of a `MTLTexture` object, you can use the following methods:

`replaceRegion:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage:` copies a region of pixel data from the caller's pointer into a portion of the storage allocation of a specified texture slice.
`replaceRegion:mipmapLevel:withBytes:bytesPerRow:` is a similar convenience method that copies a region of pixel data into the default slice, assuming default values for slice-related arguments (i.e., `slice` = 0 and `bytesPerImage` = 0).

`getBytes:bytesPerRow:bytesPerImage:fromRegion:mipmapLevel:slice:` retrieves a region of pixel data from a specified texture slice.
`getBytes:bytesPerRow:fromRegion:mipmapLevel:` is a similar convenience method that retrieves a region of pixel data from the default slice, assuming default values for slice-related arguments (`slice` = 0 and `bytesPerImage` = 0).

Listing 3-3 (page 22) shows how to call `replaceRegion:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage:` to specify a texture image from source data in system memory, `textureData`, at slice `0` and mipmap level `0`.

以同步阻塞式拷贝图像数据进出纹理对象的内存，可以使用如下方法：

replaceRegion:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage:，从withBytes参数表示的资源对象中拷贝指定区域的像素数据 到 一个指定的纹理切片的指定内存区域。
replaceRegion:mipmapLevel:withBytes:bytesPerRow:，和前一个方法类似，只是slice和bytesPerImage设置为0
getBytes:bytesPerRow:bytesPerImage:fromRegion:mipmapLevel:slice:，获取指定切片的指定区域的像素数据
getBytes:bytesPerRow:fromRegion:mipmapLevel:，和前一个方法类似，，只是slice和bytesPerImage设置为0
代码3-3展示了如何调用replaceRegion:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage:方法将资源对象textureData对应的内容设定到纹理对象tex的切片0的mipmap第0层图像中。

Listing 3-3 Copying Image Data into the Texture

```
//  pixelSize is the size of one pixel, in bytes
//  width, height – number of pixels in each dimension

NSUInteger myRowBytes = width * pixelSize;
NSUInteger myImageBytes = rowBytes * height;
[tex replaceRegion:MTLRegionMake2D(0,0,width,height)
      mipmapLevel:0 slice:0 withBytes:textureData
      bytesPerRow:myRowBytes bytesPerImage:myImageBytes];
```

Pixel Formats for Textures
纹理的像素格式

`MTLPixelFormat` specifies the organization of color, depth, or stencil data storage in individual pixels of a `MTLTexture` object. There are three varieties of pixel formats: ordinary, packed, and compressed.

Ordinary formats have only regular 8, 16, or 32-bit color components. Each component is arranged in increasing memory addresses with the first listed component at the lowest address. For example, `MTLPixelFormatRGBA8Unorm` is a 32-bit format with eight bits for each color component; the lowest addresses contains red, the next addresses contain green, etc. In contrast, for `MTLPixelFormatBGRA8Unorm`, the lowest addresses contains blue, the next addresses contain green, etc.

Packed formats combine multiple components into one 16-bit or 32-bit value, where the components are stored from the least to most significant bit (LSB to MSB). For example, `MTLPixelFormatRGB10A2Uint` is a 32-bit packed format that consists of three 10-bit channels (for R, G, and B) and two bits for alpha.

Compressed formats are arranged in blocks of pixels, and the layout of each block is specific to that pixel format. Compressed pixel formats can only be used for 2D, 2D Array, or cube texture types. Compressed formats cannot be used to create 1D, 2DMultisample or 3D textures.

MTLTexture对象的MTLPixelFormat属性指定颜色、深度或是模板缓存数据中每一个像素如何组织。有3种像素格式：原生，紧密填充和压缩:

原生格式只有8、16或是32位颜色值，每个分量以升序内存地址排列，第一个分量处于最低的内存地址处。举例来说，MTLPixelFormatRGBA8Unorm是一个32位格式颜色值，每个8个比特位表示一个颜色分量，那么最低位内存地址保存红色分量，接下来的地址保存绿色分量。而对于MTLPixelFormatBGRA8Unorm类型的颜色值，最低位内存地址保存的时蓝色分量，接下来的地址是绿色分量。

紧密填充格式把多个颜色分量结合起来存放在一个16位或是32为的值中，这些颜色分量从最低位（LSB）向最高位（MSB）填充。例如，MTLPixelFormatRGB10A2Uint是一个32位的填充格式色值，它含有3个10比特长的颜色通道（分别存放RGB分量）以及一个2比特的alpha分量。

压缩格式用于排列像素块的，每个块的布局被设定为这种像素格式。压缩格式只能被用于2维、2维数组或是立方类型的纹理。它不能被用于创建1维，2维多重采样或是3维类型纹理。

The `MTLPixelFormatGBGR422` and `MTLPixelFormatBGRG422` are special pixel formats that are intended to store pixels in the YUV color space. These formats are only supported for 2D textures (but neither 2D Array, nor cube type), without mipmaps, and an even `width`.

Several pixel formats store color components with sRGB color space values (e.g., `MTLPixelFormatRGBA8Unorm_sRGB` or `MTLPixelFormatETC2_RGB8_sRGB`). When a sampling operation references a texture with an sRGB pixel format, the Metal implementation converts the sRGB color space components to a linear color space before the sampling operation takes place. The conversion from an sRGB component, S, to a linear component, L, is as follows:

`MTLPixelFormatGBGR422`和`MTLPixelFormatBGRG422`是两种特殊的像素格式，它们用于存储YUV颜色空间的像素。这种格式只支持不含mipmap的且width为偶数的2维纹理和2维纹理数组（不含立方纹理）

还有几种支持sRGB颜色空间的格式，比如`MTLPixelFormatRGBA8Unorm_sRGB`和`MTLPixelFormatETC2_RGB8_sRGB`。当采样操作用到具有sRGB格式的纹理时，metal将在采样操作前把sRGB颜色空间的颜色分量转换成线性颜色空间分量。转换公式如下，S表示sRGB颜色，L表示线性颜色空间颜色。

if $S <= 0.04045$, $L = S/12.92$
if $S > 0.04045$, $L = ((S+0.055)/1.055)^{2.4}$

Conversely, when rendering to a color-renderable attachment that uses a texture with an sRGB pixel format (see Creating a Render Pass Descriptor (page 32)), the implementation converts the linear color values to sRGB, as follows:

相反的，当使用sRGB像素格式的纹理绘制一个attachment的时候，会按下面的公式将线性颜色值转换为sRGB颜色值。

if $L <= 0.0031308$, $S = L * 12.92$
if $L > 0.0031308$, $S = (1.055 * L^{0.41667}) - 0.055$

Sampler States Contain Properties for Texture Lookup
采样器状态包含的用于纹理查找的属性

`MTLSamplerState` defines the addressing, filtering, and other properties that are used when a graphics or compute function performs texture sampling operations on a `MTLTexture`. To create a sampler state object:

1. First create a **`MTLSamplerDescriptor`** to define the sampler state properties.
2. Set the desired values in the **`MTLSamplerDescriptor`**, including filtering options, addressing modes, maximum anisotropy, and level-of-detail parameters.
3. To create a **`MTLSamplerState`**, call the **`newSamplerStateWithDescriptor:`** method of **`MTLDevice`**.

You can reuse the sampler descriptor object to create more `MTLSamplerState` objects, modifying the descriptor's property values as needed. The descriptor's properties are only used during object creation. After a sampler state has been created, changing the properties in its descriptor no longer has an effect on that sampler state.

Listing 3-4 (page 24) is a code example that creates a `MTLSamplerDescriptor` and configures it in order to create a `MTLSamplerState`. Non-default values are set for filter and address mode properties of the descriptor object. Then the `newSamplerStateWithDescriptor:` method uses the sampler descriptor to create a sampler state object.

`MTLSamplerState`定义了寻址、过滤还有其他属性，用于一个图形着色程序或是并行计算着色程序对一个`MTLTexture`对象实施采样操作。创建一个采样器state对象的步骤如下：
1. 创建一个MTLSamplerDescriptor类型对象定义采样器状态属性。
2. 为这个descriptor设置相应的值，比如过滤操作，寻址模式，最大的异向性，还有LOD参数
3 调用MTLDevice的newSamplerStateWithDescriptor:方法，使用先前创建的descriptor，就可以创建一个MTLSamplerState类型对象。
可以仅仅修改descritor的必要的属性值后，重用它创建更多的`MTLSamplerState`对象。descriptor的属性值仅仅在创建`MTLSamplerState`对象时生效，state对象创建完成后，改变descriptor的属性值不会影响已经创建的state对象。

代码3-4是示例了如何创建并设置一个MTLSamplerDescriptor对象，然后用它来创建一个MTLSamplerState对象。descriptor的过滤和寻址属性被设置为非默认值。最后newSamplerStateWithDescriptor:方法使用descriptor为参数创建了一个采样state对象。

Listing 3-4 Creating a Sampler Object

```
// create MTLSamplerDescriptor
MTLSamplerDescriptor *desc = [[MTLSamplerDescriptor alloc] init];
desc.minFilter = MTLSamplerMinMagFilterLinear;
desc.magFilter = MTLSamplerMinMagFilterLinear;
desc.sAddressMode = MTLSamplerAddressModeRepeat;
desc.tAddressMode = MTLSamplerAddressModeRepeat;
// all properties below have default values
desc.mipFilter        = MTLSamplerMipFilterNotMipmapped;
desc.maxAnisotropy    = 1U;
desc.normalizedCoords = YES;
desc.lodMinClamp      = 0.0f;
desc.lodMaxClamp      = FLT_MAX;
// create MTLSamplerState
id <MTLSamplerState> sampler = [device newSamplerStateWithDescriptor:desc];
```

Maintaining Coherency between CPU and GPU Memory

在CPU内存和GPU内存间保持一致性

Both the CPU and GPU can access the underlying storage for a `MTLResource` object. However, the GPU operates asynchronously from the host CPU, so you should keep the following in mind when using the host CPU to access the storage for these resources.

When executing a `MTLCommandBuffer` object, the `MTLDevice` object is only guaranteed to observe any changes made by the host CPU to the storage allocation of any `MTLResource` object referenced by that `MTLCommandBuffer` object if (and only if) those changes were made by the host CPU before the `MTLCommandBuffer` object was committed. That is, the `MTLDevice` object might not observe changes to the resource that the host CPU makes after the corresponding `MTLCommandBuffer` object was committed (i.e., the `status` property of the `MTLCommandBuffer` object is `MTLCommandBufferStatusCommitted`).

Similarly, after the `MTLDevice` object executes a `MTLCommandBuffer` object, the host CPU is only guaranteed to observe any changes the `MTLDevice` object makes to the storage allocation of any resource referenced by that command buffer if the command buffer has completed execution (i.e., the `status` property of the `MTLCommandBuffer` object is `MTLCommandBufferStatusCompleted`).

CPU和GPU都可以访问一个MTLResource类型的对象管理的存储数据。但是GPU和CPU的操作是异步进行的，所以当使用CPU访问这些资源对应的存储时，有如下注意事项：

当执行一个MTLCommandBuffer对象。MTLDevice对象只能保证观察到由CPU带来的和command buffer相关的那些MTLResource对象的存储上产生的变化，并且这些变化是在command buffer被提交之前产生的。也就是说，在command buffer被提交之后（这时command buffer的status是MTLCommandBufferStatusCommitted），MTLDevice对象就观察不到这些资源的变化情况了。
类似的，当MTLDevice对象执行完一个command buffer对象后。这时command buffer的status值为MTLCommandBufferStatusCompleted，CPU只保证能观察到由MTLDevice对象带来的command buffer相关的那些资源文件的存储上的变化。

Functions and Libraries
着色程序和库

This chapter describes how to create a `MTLFunction` object as a reference to a Metal shader or compute function and how to organize and access `MTLFunction` objects in a `MTLLibrary`.

本章描述如何创建一个`MTLFunction`对象用于表示一个Metal绘制着色程序或是并行计算着色程序，以及如何在`MTLLibrary`中组织和访问`MTLFunction`对象。

MTLFunction Represents a Shader or Compute Function
MTLFunction表示一个绘制着色程序或是并行计算着色程序

`MTLFunction` represents a single function that is written in the Metal shading language and executed by the `MTLDevice` as part of a graphics or compute function. For details on the Metal shading language, see the *Metal Shading Language Guide* .

To pass data or state between the Metal runtime and a graphics or compute function written in the Metal shading language, you assign an argument index for textures, buffers, and samplers. The argument index identifies which texture, buffer or sampler is being referenced by both the Metal runtime and Metal shading code.

For a rendering pass, you can specify a `MTLFunction` for use in a vertex or fragment shader in a `MTLRenderPipelineDescriptor` object, as detailed in Creating a Render Pipeline State (page 37). Alternately, you can specify a `MTLFunction` in a `MTLComputePipelineState` object, as described in Specify a Compute State and Resources for a Compute Command Encoder (page 58).

`MTLFunction`表示一个着色程序，一个着色程序是用Metal着色语言编写的，它作为图形渲染或是数据并行计算全过程的一部分，在`MTLDevice`中执行。

为了在CPU端的Metal运行时 和 以Metal着色语言编写的运行在GPU端的图形着色程序、并行数据计算程序 之间传递数据和状态，纹理、缓存和采样器被赋予参数索引，这个索引指定了哪个纹理、缓存或是采样器被Metal运行时和Metal着色代码引用。

对于一个绘制pass，可以在`MTLRenderPipelineDescriptor`中用`MTLFunction`类型的值设定一个顶点或是片元着色器。对于并行数据计算，可以在`MTLComputePipelineState`中设定`MTLFunction`对象。

A Library is a Repository of Functions
Library是着色程序资源库

`MTLLibrary` represents a repository of one or more `MTLFunction` objects. A single `MTLFunction` represents one Metal function that has been written with the shading language. In the Metal shading language source code, any function that uses a Metal function qualifier (e.g., `vertex`, `fragment`, or `kernel`) can be represented by a `MTLFunction` in a `MTLLibrary`. A Metal function without one of these function qualifiers cannot be directly represented by a `MTLFunction`, although it can called by another function within the shader.

The `MTLFunction` objects in a `MTLLibrary` can be created from:
Metal shading language code that was compiled into a binary *library* format during the app build process, or a text string containing Metal shading language source code that is compiled by the app at runtime.

`MTLLibrary`代表了一个或是多个`MTLFunction`对象的资源库。一个单独的`MTLFunction`对象代表一个使用Metal着色语言编写的Metal着色程序。在着色程序源代码中，所有的使用Metal函数修饰符号的函数（比如，`vertex`、`fragment`、`kernel`）可以在`MTLLibrary`中被表示为一个`MTLFunction`对象。而一个没有函数修饰符号的函数，尽管在着色程序中可以被其他函数调用，但是不能被直接表示为一个`MTLFunction`对象。

在`MTLLibrary`中`MTLFunction`对象的创建方法如下两种：
Metal着色语言源代码在app的编译阶段就被编译到二进制库；或者一个包含了Metal着色语言源代码的字符串在app运行时被编译。

Creating a Library from Compiled Code
从已编译的代码中创建Library

To improve performance, compile your Metal shading language source code into a library file during your app's build process in Xcode, which avoids the costs of compiling function source during the runtime of your app. To create a `MTLLibrary` object from a library binary, call one of the following methods of `MTLDevice`:

`newDefaultLibrary` retrieves a library built for the main bundle that contains all shader and compute functions in an app's Xcode project.

`newLibraryWithFile:error:` takes the path to a library file and returns a `MTLLibrary` that contains all the functions stored in that library file.

`newLibraryWithData:error:` takes a binary blob containing code for the functions in a library and returns a `MTLLibrary` object.

For more information about compiling Metal shading language source code during the build process, see Creating Libraries During the App Build Process (page 64).

为了提升性能，可以利用xcode工具在app的创建过程中就把Metal着色语言源代码编译打包到一个库文件中，如此避免了在app的运行时编译着色程序代码的开销。从已编译的二进制库中创建一个`MTLLibrary`对象，可以使用如下的`MTLDevice`方法。

`newDefaultLibrary`，该方法返回一个library对象，它包含了一个app工程中所有的着色程序、并行计算程序。
`newLibraryWithFile:error:`，该方法指定了一个库文件路径，它将返回一个`MTLLibrary`对象，包含这个库文件中的所有着色程序。
`newLibraryWithData:error:`，该方法指定了一个二进制数据块对象，它将返回一个`MTLLibrary`对象，包含这个数据块中的所有着色程序。

Creating a Library from Source Code
从源代码中创建Library

To create a `MTLLibrary` from a string of Metal shading language source code that may contain several functions, call one of the following methods of `MTLDevice`. These methods compile the source code when the `MTLLibrary` is created. To specify the compiler options to use, set the properties in a `MTLCompileOptions` object.

`newLibraryWithSource:options:error:` synchronously compiles source code from the input string to create `MTLFunction` objects and then returns a `MTLLibrary` object that contains them.

`newLibraryWithSource:options:completionHandler:` asynchronously compiles source code from the input string to create `MTLFunction` objects and then returns a `MTLLibrary` object that contains them. `completionHandler` is a block of code that is invoked when object creation is completed.

从包含有Metal着色语言源代码编写的多个着色程序的字符串中创建`MTLLibrary`对象，可以调用如下的`MTLDevice`方法。这些方法将在`MTLLibrary`对象创建时编译源代码。可以设置`MTLCompileOptions`对象的属性值来指定编译器选项：

`newLibraryWithSource:options:error:`，该方法是同步阻塞调用的，它编译输入字符串里的源代码创建多个`MTLFunction`对象，最后返回一个包含这些`MTLFunction`对象的`MTLLibrary`对象。

`newLibraryWithSource:options:completionHandler:`，该方法是异步调用的，它也是编译输入字符串里的源代码创建多个`MTLFunction`对象最后返回一个包含这些`MTLFunction`对象的`MTLLibrary`对象。参数`completionHandler`是一个block，它将在`MTLLibrary`对象创建完成后被调用。

Getting a Function from a Library
从Library中获取Function

The `newFunctionWithName:` method of `MTLLibrary` returns a `MTLFunction` object with the requested name. If the name of a function that uses a Metal shading language function qualifier is not found in the library, then `newFunctionWithName:` returns `nil`.

Listing 4-1 (page 28) uses the `newLibraryWithFile:error:` method of `MTLDevice` to locate a library file by its full path name and uses its contents to create a `MTLLibrary` object with one or more `MTLFunction` objects. Any errors from loading the file are returned in `error`. Then `newFunctionWithName:` method of `MTLLibrary` creates a `MTLFunction` object that represents the function called `my_func` in the source code. The returned function object `myFunc` can now be used in an app.

`MTLLibrary`的`newFunctionWithName:`方法返回一个名字为输入参数的`MTLFunction`对象。如果在library中没有找到一个方法其修饰名字匹配输入参数，那么该函数返回nil。

代码4-1 使用一个库文件的全路径调用`MTLDevice`的`newLibraryWithFile:error:`方法载入一个库文件，使用其内容创建了一个包含多个`MTLFunction`对象的`MTLLibrary`对象。载入过程中如果有任何出错，将被记录在error中。然后`MTLLibrary`的`newFunctionWithName:`方法创建了一个`MTLFunction`对象。它代表了源代码中被命名为`my_func`的着色程序。最后`MTLFunction`对象`myFunc`可以被应用在app中。

Listing 4-1 Accessing a Function from a Library

```
NSError *errors;
id <MTLLibrary> library = [device newLibraryWithFile:@"myarchive.metallib"
                           error:&errors];
id <MTLFunction> myFunc = [library newFunctionWithName:@“my_func"];
```

Determining Function Details at Runtime
在运行时决定Function的细节

Since the actual contents of a `MTLFunction` are defined by a graphics shader or compute function that may be compiled before the `MTLFunction` was created, its source code might not be directly available to the app. You can query the following `MTLFunction` properties at run time:

`name`, a string with the name of the function.
`functionType`, which indicates whether the function is declared as a vertex, fragment, or compute function.

`vertexAttributes`, an array of `MTLVertexAttribute` objects that describe how vertex attribute data is organized in memory and how it is mapped to vertex function arguments. For more details, see Vertex Descriptor for Data Organization (page 44).

一个`MTLFunction`对象被定义为图形绘制着色程序或是并行计算着色程序，它的实质内容在这个对象被创建前就编译好了，着色程序的源代码不能直接被app使用。但是你可以在运行时查询下面的`MTLFunction`属性：

`name`，是一个表示着色程序名字的字符串。
`functionType`，这个属性说明这个着色程序的类型是顶点着色器，还是片元着色器，还是并行计算着色器。
`vertexAttributes`，这是一个`MTLVertexAttribute`对象组成的数组，它描述顶点属性数据在内存中如何组织以及如何映射到顶点着色程序参数中。

`MTLFunction` does not provide access to function arguments. A reflection object (either `MTLRenderPipelineReflection` or `MTLComputePipelineReflection`, depending upon the type of command encoder) reveals details of shader or compute function arguments can be obtained during the creation of a pipeline state. For details on creating pipeline state and reflection objects, see Creating a Render Pipeline State (page 37) or Creating a Compute Pipeline State (page 57). Avoid obtaining reflection data if it will not be used.

A reflection object contains an array of `MTLArgument` objects for each type of function supported by the command encoder. For `MTLComputeCommandEncoder`, `MTLComputePipelineReflection` has one array of `MTLArgument` objects in the `arguments` property that correspond to the arguments of its compute function. For `MTLRenderCommandEncoder`, `MTLRenderPipelineReflection` has two properties, `vertexArguments` and `fragmentArguments`, that are arrays that correspond to the vertex function arguments and fragment function arguments, respectively.

Not all arguments of a function are present in a reflection object. A reflection object only contains arguments that have an associated resource, but not arguments declared with the `[[stage_in]]` qualifier or built-in `[[vertex_id]]` or `[[attribute_id]]` qualifier.

Listing 4-2 (page 29) shows how you can obtain a reflection object (in this example, `MTLComputePipelineReflection`) and then iterate through the `MTLArgument` objects in its `arguments` property.

`MTLFunction`不提供访问着色程序参数的方法。在pipeline state对象创建过程中，可以获得reflection对象，它用于展现着色程序的参数细节。
根据command encoder的类型不同，reflection对象是`MTLRenderPipelineReflection`类型或是`MTLComputePipelineReflection`类型。如果不使用reflection数据，请不要创建这种对象。

一个reflection对象包含了一个`MTLArgument`数组，视其关联的Encoder不同而不同。对于`MTLComputeCommandEncoder`，`MTLComputePipelineReflection`类型的reflection对象的`arguments`属性含有一个`MTLArgument`数组，该数组和并行计算着色程序的参数相关联。对于`MTLRenderCommandEncoder`，`MTLRenderPipelineReflection`类型的reflection对象有两个属性，`vertexArguments`和`fragmentArguments`，分别对应顶点着色程序的输入参数和片元着色程序的输入参数。

不是所有的着色程序的参数都在reflection对象中表示，一个reflection对象仅包含那些引用了相应资源的参数，通过修饰符`[[stage_in]]`、`[[vertex_id]]`、`[[attribute_id]]`修饰的参数不会被包含。

代码4-2展示了如何获取reflection对象（代码中是`MTLComputePipelineReflection`类型的）然后遍历它的`arguments`属性中的`MTLArgument`对象。

Listing 4-2 Iteration through Function Arguments

```
MTLComputePipelineReflection* reflection;
id <MTLComputePipelineState> computePS = [device
              newComputePipelineStateWithFunction:func
              options:MTLPipelineOptionArgumentInfo
              reflection:reflection
              error:&error];
for (MTLArgument *arg in reflection.arguments) {
    //  process each MTLArgument
}
```

The `MTLArgument` properties reveal the details of an argument to a shading language function.

The `name` property is simply the name of the argument.
`active` is a Boolean that indicates whether the argument can be ignored.
`index` is a zero-based position in its corresponding argument table (e.g., for `[[buffer(2)]]`, `index` is 2).
`access` describes any access restrictions (e.g., the read or write access qualifier).
`type` is indicated by the shading language qualifier (e.g., `[[buffer(n)]]`, `[[texture(n)]]`, [[sampler(n)]], or [[local(n)]]).

`MTLArgument`类型对象表示一个着色语言的方法入参细节：

`name`，参数名字。
`active`，一个布尔值，指示是个参数是否可被忽略。
`index`，一个0based的表示该参数在其对应的参数索引表的下标位置（比如，对应着色程序中的`[[buffer(2)]]`参数，index为2）。
`access`，描述了访问限制（比如读或写访问权限）
`type`，表示着色语言的修饰符（比如`[[buffer(n)]]`, `[[texture(n)]]`, [[sampler(n)]], or [[local(n)]]）

`type` determines which other `MTLArgument` properties are relevant.

If `type` is `MTLArgumentTypeTexture`, then the `textureType` property indicates the overall texture type (e.g., `texture1d_array`, `texture2d_ms`, and `texturecube` types in the shading language), and the `textureDataType` property indicates the component data type (e.g., `half`, `float`, `int`, or `uint`).

If `type` is `MTLArgumentTypeThreadgroupMemory`, then the `threadgroupMemoryAlignment` and `threadgroupMemoryDataSize` properties are relevant.

If `type` is `MTLArgumentTypeBuffer`, then the `bufferAlignment`, `bufferDataSize`, `bufferDataType`, and `bufferStructType` properties are relevant.

type还决定了哪些MTLArgument属性是相关的。

如果type是MTLArgumentTypeTexture，那么textureType属性指示纹理类型（可能的值有texture1d_array，texture2d_ms，texturecube），textureDataType属性指示其分量数据类型（可能的值有half，float，int，uint）。
如果type是MTLArgumentTypeThreadgroupMemory，那么threadgroupMemoryAlignment和threadgroupMemoryDataSize这两个属性相关。
如果type是MTLArgumentTypeBuffer，那么bufferAlignment, bufferDataSize, bufferDataType, bufferStructType这几个属相相关。

If the buffer argument is a struct (i.e., `bufferDataType` is `MTLDataTypeStruct`), then the `bufferStructType` property contains a `MTLStructType` that represents the struct, and `bufferDataSize` contains the size of the struct, in bytes.

If the buffer argument is an array (or pointer to an array), then `bufferDataType` indicates the data type of an element, and `bufferDataSize` contains the size of one array element, in bytes.

Listing 4-3 (page 30) describes pseudocode for how to drill down the returned `MTLStructType` and to examine the details of struct members, each represented by a `MTLStructMember`. A struct member may be a simple type or may be an array or a nested struct.

If the member is a nested sub-struct, then call the `structType` method of `MTLStructMember` to obtain a `MTLStructType` that represents the sub-struct and then recursively drill down to analyze it.

If the member is an array, use the `arrayType` method of `MTLStructMember` to obtain a `MTLArrayType` that represents it. Then examine its `elementType` property of `MTLArrayType`. If `elementType` is `MTLDataTypeStruct`, call the `elementStructType` method to obtain the struct and continue to drill down into its members. If `elementType` is `MTLDataTypeArray`, call the `elementArrayType` method to obtain the sub-array and analyze it further.

如果缓存参数是一个结构体（比如，bufferDataType的值是MTLDataTypeStruct），那么bufferStructType属性含有一个MTLStructType类型的值，它表示这个结构体，同时bufferDataSize属性表示这个结构体的长度（以byte计算）。

如果缓存参数是一个数组（或者一个指向数组的指针），那么bufferDataType属性表示数组中元素的数据类型，bufferDataSize属性表示数组中一个元素的长度（以byte计算）。

代码4-3所示的伪代码描述了如何一个展开MTLStructType类型的返回值来检查结构体各个成员变量（由一个MTLStructMember类型值表示）的细节。一个结构体的成员可能是一个简单类型值，也可能是一个数组，还可能也是一个嵌套结构体。

如果成员是一个嵌套结构体，调用它的structType方法获得一个MTLStructType对象，该对象表示一个子结构体，可以接下来递归展开分析之。

如果成员是一个数组，调用它的arrayType方法可以获取一个MTLArrayType类型对象用来表示它。然后检测这个对象的elementType属性，如果该属性是一个MTLDataTypeStruct类型值，调用其elementStructType方法可以获得结构体可以向下递归之；如果该属性是一个MTLDataTypeArray类型值，调用其elementArrayType方法可以获得一个子数组可以继续往下分析。

Listing 4-3 Pseudocode to Drill Down a Struct Argument

```
MTLStructType *structObj = [arg.bufferStructType];
for (MTLStructMember *member in structObj.members) {
    //  process each MTLStructMember
    if (member.dataType == MTLDataTypeStruct) {
       // obtain MTLStructType* with [member structType]
       // recursively drill down into the sub-struct
    }
    else if (member.dataType == MTLDataTypeArray) {
       // obtain MTLArrayType* with [member arrayType]
```

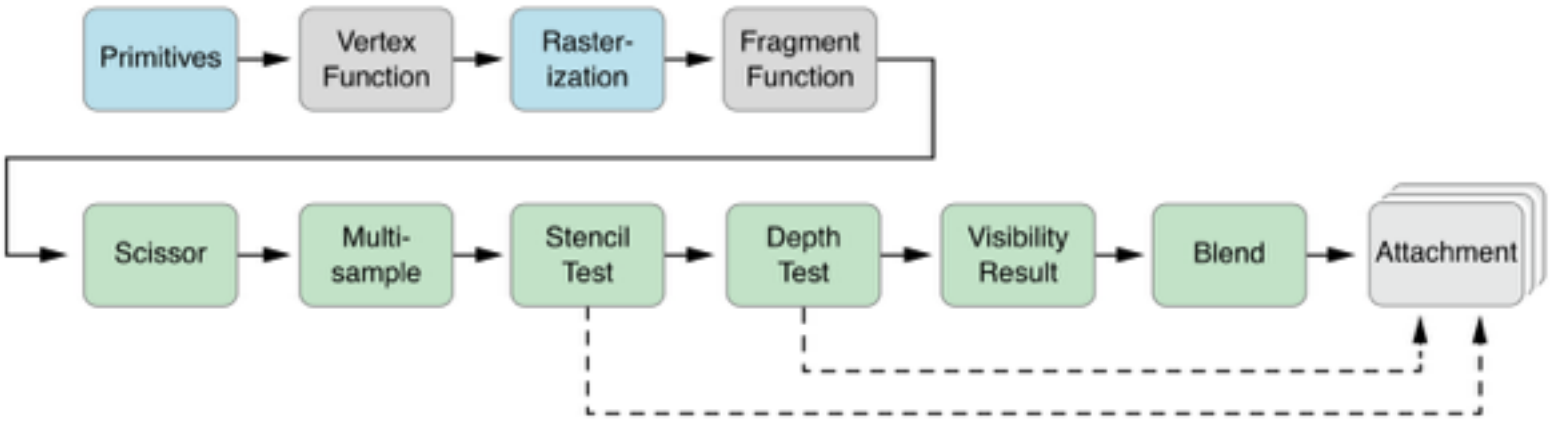
```
        // examine the elementType and drill down, if necessary
    } else {
        // member is neither struct nor array
        // analyze it; no need to drill down further
    }
}
```

Graphics Rendering: Render Command Encoder
图形渲染：渲染指令编码器

This chapter describes how to create and use `MTLRenderCommandEncoder` and `MTLParallelRenderCommandEncoder`, which are used to encode graphics rendering commands into a command buffer. `MTLRenderCommandEncoder` commands describe a graphics rendering pipeline, as seen in Figure 5-1 (page 31).

本章节描述如何创建并使用`MTLRenderCommandEncoder`和`MTLParallelRenderCommandEncoder`，它们被用来编码图形渲染指令，然后插入到command buffer。`MTLRenderCommandEncoder`的指令所运行的图形渲染管线模型如图5-1所示：

Figure 5-1 Metal Graphics Rendering Pipeline



`MTLRenderCommandEncoder` represents a single rendering command encoder. `MTLParallelRenderCommandEncoder` enables a single rendering pass to be broken into a number of separate `MTLRenderCommandEncoder` objects, each of which may be assigned to a different thread. The commands from the different render command encoders are then chained together and executed in a consistent, predictable order, as described in Multiple Threads for a Rendering Pass (page 55).

`MTLRenderCommandEncoder`表示一个单独的图形渲染指令编码器，`MTLParallelRenderCommandEncoder`使得一个单独的渲染pass被分成若干个独立的对象，每一个都可以被分配到不同的线程，这些编码器中的指令随后将被串行起来以一致的可预测的顺序被执行。

Creating and Using a Render Command Encoder
创建并使用一个渲染指令编码器

To create, initialize, and use a single `MTLRenderCommandEncoder`:

First create a `MTLRenderPassDescriptor` to define a collection of attachments that serve as the rendering destination for the graphics commands in its command buffer. A single `MTLRenderPassDescriptor` object is usually created once and then used to create many `MTLRenderCommandEncoder` objects.

Create a `MTLRenderCommandEncoder` by calling the `renderCommandEncoderWithDescriptor:` method of `MTLCommandBuffer` with the specified render pass descriptor.

Create a `MTLRenderPipelineState` to define the state of the graphics rendering pipeline (including shaders, blending, multisampling, and visibility testing) for one or more draw calls. To use this render pipeline state for drawing primitives, call the `setRenderPipelineState:` method of `MTLRenderCommandEncoder`. For details, see Creating a Render Pipeline State (page 37).

Set textures, buffers, and samplers to be used by the `MTLRenderCommandEncoder`, as described in Specifying Resources for a Render Command Encoder (page 42).

Call `MTLRenderCommandEncoder` methods to specify additional fixed-function state, including the depth and stencil state, as explained in Fixed-Function State Operations (page 48).

Finally, call `MTLRenderCommandEncoder` methods to draw graphics primitives, as described in Drawing Geometric Primitives (page 51).

使用以下步骤（6步）来创建、初始化、使用一个单独的MTLRenderCommandEncoder：

1. 创建一个MTLRenderPassDescriptor对象来定义一系列attachment，这些attachment将成为在command buffer中的绘制指令操作的目标产出物。一个MTLRenderPassDescriptor对象通常被创建一次然后被多次用来创建MTLRenderCommandEncoder对象。

2. 使用之前准备的MTLRenderPassDescriptor对象来调用MTLCommandBuffer的renderCommandEncoderWithDescriptor:方法，创建MTLRenderCommandEncoder对象

3. 创建一个MTLRenderPipelineState对象来定义图形渲染管线的状态（包括着色器，混合、多重采样 和 可见测试）以便进行1次或者多次绘制命令调用。调用MTLRenderCommandEncoder对象的setRenderPipelineState方法，设置state对象给一个Encoder，表示使用这个state对象来进行图元绘制。

4. 设置MTLRenderCommandEncoder对象需要用到的纹理、缓存和采样器。

5. 使用MTLRenderCommandEncoder的特定方法设置一些额外的固定绘制管线状态，包括深度和模板缓存状态。

6. 调用MTLRenderCommandEncoder对象的方法绘制图元。

Creating a Render Pass Descriptor
创建一个MTLRenderPassDescriptor

`MTLRenderPassDescriptor` represents the destination for the encoded rendering commands, which is a collection of attachments. The properties of `MTLRenderPassDescriptor` may include an array of up to four attachments for color pixel data, one attachment for depth pixel data, and one attachment for stencil pixel data. `MTLRenderPassDescriptor` has the `renderPassDescriptor` convenience method that creates an `MTLRenderPassDescriptor` object with color, depth, and stencil attachment properties with default attachment state. (`visibilityResultBuffer` is another `MTLRenderPassDescriptor` property that specifies a buffer where the device can update to indicate whether any samples pass the depth and stencil tests. `visibilityResultBuffer` is discussed with the `setVisibilityResultMode:offset:` method of `MTLRenderCommandEncoder` in Fixed-Function State Operations (page 48).)

Each individual attachment, including the texture that will be written to, is represented by an attachment descriptor. For an attachment descriptor, the pixel format of the associated texture must be chosen appropriately to store color, depth, or stencil data. For a color attachment descriptor, `MTLRenderPassColorAttachmentDescriptor`, use a color-renderable pixel format. For a depth attachment descriptor, `MTLRenderPassDepthAttachmentDescriptor`, use a depth-renderable pixel format, such as `MTLPixelFormatDepth32Float`. For a stencil attachment descriptor, `MTLRenderPassStencilAttachmentDescriptor`, use a stencil-renderable pixel format, such as `MTLPixelFormatStencil8`.

The amount of memory the texture actually uses per pixel on the device does not always match the size of the texture's pixel format in the Metal framework code, because the device adds padding for alignment or other purposes. See *Metal Constants Reference* for how much memory is actually used for each pixel format.

The limitations on the size and number of attachments are described in Metal Capabilities and Limitations (page 65).

MTLRenderPassDescriptor对象代表被编码的绘制指令操作的目标产出物，它是一些列的attachment对象。

它的属性包括一个最多含有4个元素的表示颜色的attachment对象数组、一个表示深度缓存的attachment对象、一个表示模板缓存的attachment对象。

MTLRenderPassDescriptor类有一个便捷方法renderPassDescriptor，用来产生一个descriptor对象，这个对象的color（颜色数组）、depth（深度缓存）、stencil（模板缓存）attachment具有默认的状态。

visibilityResultBuffer是MTLRenderPassDescriptor对象的另一个属性，它用来指明一块缓存，在这里设备可以更新其数据，用于标识采样器是否通过深度和模板测试，这个属性将在“固定绘制管线状态操作”章节，和MTLRenderCommandEncoder的setVisibilityResultMode:offset:方法一起讨论。

每一个单独的attachment对象，包括将被写入数据的纹理，都是由一个attachment descriptor来描述的。对于一个attachment descriptor来说，和其关联的纹理的像素格式必须正确选择，以便用来存放颜色、深度或是模板数据。
对于一个颜色attachment descriptor来说，MTLRenderPassColorAttachmentDescriptor，要使用颜色缓存适配的像素格式；
对一个深度attachment descriptor来说，MTLRenderPassDepthAttachmentDescriptor，要使用深度缓存适配的像素格式，比如MTLPixelFormatDepth32Float；
对一个模板attachment descriptor来说，MTLRenderPassStencilAttachmentDescriptor，要使用模板缓存适配的像素格式，比如MTLPixelFormatStencil8；

对于一个纹理，其每像素实际使用的设备上的内存数并不总是和代码中的像素格式相同，因为设备会因为对齐或是其他原因添加留白字节，查阅《*Metal Constants Reference*》可知每种像素格式实际使用多少内存。每个附件对象的尺寸限制在章节“Metal 性能和限制”中描述

`loadAction` and `storeAction` are attachment descriptor properties that specify an action that is performed at either the start or end of a rendering pass, respectively, for the specified attachment descriptor. (For `MTLParallelRenderCommandEncoder`, the `loadAction` and `storeAction` occur at the boundaries of the overall command, not for each of its `MTLRenderCommandEncoder` objects. For details, see Multiple Threads for a Rendering Pass (page 55).)

loadAction和storeAction是attachment descriptor的两个属性，它们用来设定在每个绘制pass的开头和结尾做什么特定操作（对于一个MTLParallelRenderCommandEncoder对象来说，loadAction和storeAction发生在overall指令栅栏而不是它拥有的每个MTLRenderCommandEncoder对象上）

`loadAction` values include:

`MTLLoadActionClear`, which writes the same value to every pixel in the specified attachment descriptor. (For more detail about `MTLLoadActionClear`, see Specifying the Clear Load Action (page 35).)

`MTLLoadActionLoad`, which preserves the existing contents of the texture.

`MTLLoadActionDontCare`, which allows each pixel in the attachment to take on any value at the start of the rendering pass.

If your application will render all pixels of the attachment for a given frame, then for best performance, you should use the default value `MTLLoadActionDontCare` for `loadAction`. Otherwise, you can use `MTLLoadActionClear` to clear the previous contents of the attachment, and `MTLLoadActionLoad` to preserve them.

For best performance, use `MTLLoadActionDontCare` for `loadAction`, which allows the GPU to avoid loading the existing contents of the texture. `MTLLoadActionClear` also avoids loading the existing texture contents, but it incurs the cost of filling the destination with a solid color.

loadAction有这么几种取值：
MTLLoadActionClear，它将为每个像素（由某个attachment descriptor设定的）写入相同的值。
MTLLoadActionLoad，它将保留纹理中已经存在的内容。
MTLLoadActionDontCare，它将允许在绘制pass开始时，纹理中得每个像素取任意值。
如果你的app将为一帧画面绘制所有的像素，那么为了得到最好的性能，你应该使用默认值MTLLoadActionDontCare设定loadAction。
如果不是这样，你可以使用MTLLoadActionClear来清除之前的内容，或是MTLLoadActionLoad来保留之前的内容。
设置loadAction为MTLLoadActionDontCare将获得最好的性能，因为GPU在这个阶段避免了为缓存加载数据的动作。MTLLoadActionClear也可以避免了加载数据的操作，但是它将花费时间用颜色填充缓存。

`storeAction` values include:

`MTLStoreActionStore`, which saves the final results of the rendering pass into the attachment.

`MTLStoreActionMultisampleResolve`, which resolves the multisample data from the render target into single sample values, stores them in the texture specified by the attachment property `resolveTexture`, and leaves the contents of the attachment undefined.

`MTLStoreActionDontCare`, which leaves the attachment in an undefined state after the rendering pass is complete. This may improve performance as it enables the implementation to avoid any work necessary to preserve the rendering results.

`MTLStoreActionStore` is the default value for the store action for color attachments, since applications almost always preserve the final color values in the attachment at the end of rendering pass. `MTLStoreActionDontCare` is the default value for the store action for depth and stencil data, since those attachments typically do not need to be preserved after the rendering pass is complete.

storeAction有这么几种取值：
MTLStoreActionStore，它将绘制pass的最终结果保存到缓存。
MTLStoreActionMultisampleResolve，它将绘制pass得到的数据作为多重采样数据，计算得到一个采样值，将采样值存储到由resolveTexture指定的一块纹理中，而不操作缓存。
MTLStoreActionDontCare，它将在绘制pass结束后不操作缓存，这将提升性能因为它避免了保存绘制结果到缓存的操作。
MTLStoreActionStore，是颜色缓存的storeAction的默认值，app几乎总是在绘制pass结束的时候保存最终的颜色值在缓存中。MTLStoreActionDontCare是深度缓存和模板缓存的默认值，这两种缓存通常不需要在绘制pass结束时保存绘制结果。

Listing 5-1 (page 34) creates a simple render pass descriptor with color and depth attachments. First, two texture objects are created, one with a color-renderable pixel format and the other with a depth pixel format. Next the `renderPassDescriptor` convenience method of `MTLRenderPassDescriptor` creates a default

render pass descriptor. Then the color and depth attachments are accessed through the properties of `MTLRenderPassDescriptor`. The textures and actions are set in `colorAttachments[0]`, which represents the first color attachment (at index 0 in the array), and the depth attachment.

列表5-1 用颜色和深度缓存创建了一个简单的render pass descriptor。
首先，两个纹理对象被创建出来，一个被赋予颜色缓存适配的像素格式，另一个被赋予深度缓存适配像素格式。
接着MTLRenderPassDescriptor类的快捷方法renderPassDescriptor被调用，创建了一个默认的MTLRenderPassDescriptor对象 renderPassDesc。
然后renderPassDesc的颜色缓存属性（数组）和深度缓存属性被访问，位于colorAttachments[0]的纹理对象和action被设定，colorAttachments[0]表示renderPassDesc的第一个颜色attachment对象。
最后renderPassDesc的深度缓存属性也被设置。

Listing 5-1 Creating a Render Pass Descriptor with Color and Depth Attachments

```
MTLTextureDescriptor *colorTexDesc = [MTLTextureDescriptor
            texture2DDescriptorWithPixelFormat:MTLPixelFormatRGBA8Unorm
            width:IMAGE_WIDTH height:IMAGE_HEIGHT mipmapped:NO];
id <MTLTexture> colorTex = [gDevice newTextureWithDescriptor:colorTexDesc];

MTLTextureDescriptor *depthTexDesc = [MTLTextureDescriptor
            texture2DDescriptorWithPixelFormat:MTLPixelFormatDepth32Float
            width:IMAGE_WIDTH height:IMAGE_HEIGHT mipmapped:NO];
id <MTLTexture> depthTex = [gDevice newTextureWithDescriptor:depthTexDesc];

MTLRenderPassDescriptor *renderPassDesc = [MTLRenderPassDescriptor renderPassDescriptor];
renderPassDesc.colorAttachments[0].texture = colorTex;
renderPassDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
renderPassDesc.colorAttachments[0].storeAction = MTLStoreActionStore;
renderPassDesc.colorAttachments[0].clearColor = MTLClearColorMake(0.0,1.0,0.0,1.0);

renderPassDesc.depthAttachment.texture = depthTex;
renderPassDesc.depthAttachment.loadAction = MTLLoadActionClear;
renderPassDesc.depthAttachment.storeAction = MTLStoreActionStore;
renderPassDesc.depthAttachment.clearDepth = 1.0;
```

When performing the `MTLStoreActionMultisampleResolve` action, the `texture` property must be set to a multisample-type texture, and the `resolveTexture` property will contain the result of the multisample downsample operation. (If `texture` does not support multisampling, then `MTLStoreActionMultisampleResolve` is undefined.) The `resolveLevel`, `resolveSlice`, and `resolveDepthPlane` properties may also be used for the multisample resolve operation to specify the mipmap level, cube slice, and depth plane of the multisample texture, respectively. In most cases, the default values for `resolveLevel`, `resolveSlice`, and `resolveDepthPlane` are usable. In Listing 5-2

(page 34), an attachment is initially created and then its `loadAction`, `storeAction`, `texture`, and `resolveTexture` properties are set to support multisample resolve.

当一个attachment对象应用MTLStoreActionMultisampleResolve的时候，的texture属性必须被设置为一个可重采样的texture，同时resolveTexture属性要指定一个texture对象存放重采样的结果（如果被设置的texture不支持可重采样，那么MTLStoreActionMultisampleResolve的行为未知）。
attachment对象的下面这三个属性也用于重采样分解操作：resolveLevel指定mipmap的层级，resolveSlice指定立方切片，resolveDepthPlane指定重采样纹理的深度平面。几乎所有的情况下，这三个属性的默认值是可用的。
在列表5-2中，一个attachment被创建，然后它的loadAction, storeAction, texture, resolveTexture这几个属性被赋值用于支持重采样。

Listing 5-2 Setting Properties for an Attachment with Multisample Resolve

```
MTLTextureDescriptor *colorTexDesc = [MTLTextureDescriptor
            texture2DDescriptorWithPixelFormat:MTLPixelFormatRGBA8Unorm
            width:IMAGE_WIDTH height:IMAGE_HEIGHT mipmapped:NO];
id <MTLTexture> colorTex = [gDevice newTextureWithDescriptor:colorTexDesc];

MTLTextureDescriptor *msaaTexDesc = [MTLTextureDescriptor
            texture2DDescriptorWithPixelFormat:MTLPixelFormatRGBA8Unorm
            width:IMAGE_WIDTH height:IMAGE_HEIGHT mipmapped:NO];
msaaTexDesc.textureType = MTLTextureType2DMultisample;
msaaTexDesc.sampleCount = sampleCount;  //  must be > 1
id <MTLTexture> msaaTex = [gDevice newTextureWithDescriptor:msaaTexDesc];

MTLRenderPassDescriptor *renderPassDesc = [MTLRenderPassDescriptor renderPassDescriptor];
renderPassDesc.colorAttachments[0].texture = msaaTex;
renderPassDesc.colorAttachments[0].resolveTexture = colorTex;
renderPassDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
renderPassDesc.colorAttachments[0].storeAction = MTLStoreActionMultisampleResolve;
renderPassDesc.colorAttachments[0].clearColor = MTLClearColorMake(0.0,1.0,0.0,1.0);
```

Specifying the Clear Load Action
设定清除操作属性

If the `loadAction` property of an attachment descriptor is set to `MTLLoadActionClear`, then a clearing value is written to every pixel in the specified attachment descriptor at the start of a rendering pass. The clearing value property depends upon the type of attachment.

For `MTLRenderPassColorAttachmentDescriptor`, `clearColor` contains a `MTLClearColor` value that consists of four double-precision floating-point RGBA components and is used to clear the color attachment. `MTLClearColorMake(red, green, blue, alpha)` creates the corresponding `MTLClearColor` value. The default value is (0.0, 0.0, 0.0, 1.0), which is opaque black.

For `MTLRenderPassDepthAttachmentDescriptor`, `clearDepth` contains one double-precision floating-point clearing value in the range [0.0, 1.0] that is used to clear the depth attachment. The default value is 1.0.

For `MTLRenderPassStencilAttachmentDescriptor`, `clearStencil` contains one 32-bit unsigned integer that is used to clear the stencil attachment. The default value is 0.

如果一个attachment descripter对象的loadAction属性被设置为MTLLoadActionClear，那么每个绘制pass开始的时候这个descripter对应每个像素都将被写入一个“清除”值，这个值视attachement对象的类型而定。
对于一个MTLRenderPassColorAttachmentDescriptor，MTLClearColor类型的“清除”值由4个双精度浮点型数组成，代表RGBA颜色分量，用来填充清理颜色attachment。
MTLClearColorMake(red, green, blue, alpha)用来创建一个相应的MTLClearColor类型的颜色值，默认值是(0.0, 0.0, 0.0, 1.0)，这代表不透明的黑色。
对于一个MTLRenderPassDepthAttachmentDescriptor，clearDepth类型的“清除”值由1个双精度浮点型数组成，取值范围是[0.0, 1.0]，默认值是1.0，用来填充清理深度attachment。
对于一个MTLRenderPassStencilAttachmentDescriptor，clearStencil类型的“清除”值由1个32位无符号整数组成，默认值是0，用来填充清理模板attachment。

Using the Render Pass Descriptor to Create a Render Command Encoder

使用RenderPassDescriptor来创建一个RenderCommandEncoder

After the creation and specification of `MTLRenderPassDescriptor`, the `renderCommandEncoderWithDescriptor:` method of `MTLCommandBuffer` uses the render pass descriptor `renderPassDesc` to create the `MTLRenderCommandEncoder`, as seen in Listing 5-3 (page 35).

在创建并设定好MTLRenderPassDescriptor对象后，MTLCommandBuffer的renderCommandEncoderWithDescriptor方法使用这个descriptor对象创建MTLRenderCommandEncoder对象，如代码5-3所示

Listing 5-3 Creating a Render Command Encoder with the Render Pass Descriptor

```
id <MTLRenderCommandEncoder> renderCE = [commandBuffer
                    renderCommandEncoderWithDescriptor:renderPassDesc];
```

Integrating with Core Animation
和Core Animation集成

Core Animation defines the `CAMetalLayer` class, which is designed for the specialized behavior of a layer-backed view with Metal rendered content. A `CAMetalLayer` object represents information about the geometry of the content (position and size), its visual attributes (background color, border, and shadow), and the resources used by Metal to present the content in a color attachment. It also encapsulates the timing of content presentation so that the content can be displayed as soon as it is available or at a specified time. (For more information about Core Animation, see the *Core Animation Programming Guide* and the *Core Animation Cookbook* .)

Core Animation also defines the `CAMetalDrawable` protocol for objects that are displayable resources. The `CAMetalDrawable` protocol extends `MTLDrawable` and vends an object that conforms to the `MTLTexture` protocol, so it can be used as a destination for rendering commands. To render into a `CAMetalLayer` object, you should get a new `CAMetalDrawable` object for each rendering pass, get the `MTLTexture` object that it vends, and use that texture to create the color attachment. Unlike color attachments, creation and destruction of a depth or stencil attachment are costly. If you need either depth or stencil attachments, create them once and then reuse them each time a frame is rendered.

Core Animation定义了CAMetalLayer类，它被设计用来控制承载了metal绘制内容的基于层的视图的行为。一个CAMetalLayer对象表示了绘制内容的几何区域（位置和尺寸）、视觉属性（背景色，边框，阴影）、还有metal用来呈现内容的颜色缓存资源。它还封装了内容呈现的定时器，如此当内容准备好或是某个特定时刻，内容就可以被显示出来。
CoreAnimation还为可视化资源定义了CAMetalDrawable协议，CAMetalDrawable协议扩展了MTLDrawable协议，并且产出适配MTLTexture协议的对象，所以声明为符合CAMetalDrawable协议的可视化对象可以用作渲染命令的目标产出物。在CAMetalLayer对象中实现绘制渲染，你应该为每个绘制pass创建新的CAMetalDrawable对象，获取它提供的MTLTexture对象，使用MTLTexture对象创建颜色attachment对象。
和attachment颜色不同，深度attachment对象和模板attachment对象的创建和销毁成本很高。如果你需要使用这两种attachment对象，一次创建，在随后的每帧绘制中重用之。

To create a `CAMetalLayer` object, call its `init` method.

To create a displayable resource (a `CAMetalDrawable` object), first set the appropriate properties of the `CAMetalLayer` object and then call its `nextDrawable` method. If the `CAMetalLayer` properties are not set, the `nextDrawable` method call fails. The following `CAMetalLayer` properties describe the drawable object:

The `device` property declares the `MTLDevice` object that the resource is created from.
The `pixelFormat` property declares the pixel format of the texture. The supported values are

`MTLPixelFormatBGRA8Unorm` (the default) and `MTLPixelFormatBGRA8Unorm_sRGB`.

The `drawableSize` property declares the pixel dimensions of the texture.

The `framebufferOnly` property declares whether or not the texture can only be used as an attachment (`YES`) or whether it can also be used for texture sampling and pixel read/write operations (`NO`). If `YES`, the layer object can optimize the texture for display. For most apps, the recommended value for is `YES`.

The `presentsWithTransaction` property declares whether or not changes to the layer's rendered resource are updated with standard Core Animation transaction mechanisms (`YES`) or are updated asynchronously to normal layer updates (`NO`, the default value).

调用CAMetalLayer类的init方法创建一个CAMetalLayer对象
创建一个可视化资源（一个CAMetalDrawable对象），首先设置好CAMetalLayer对象的各个属性，然后调用它的nextDrawable方法。如果CAMetalLayer对象的属性没有被设置，nextDrawable调用将失败。下面这些CAMetalLayer对象的属性其实是描述了CAMetalDrawable对象：
device，资源对象由它创建。
pixelFormat，描述纹理对象的像素格式。可用的值有：MTLPixelFormatBGRA8Unorm（默认值）和MTLPixelFormatBGRA8Unorm_sRGB。
drawableSize，描述了纹理的像素尺寸。
framebufferOnly，指明纹理对象只能用做attachment（属性值为YES），或是它还能被用作纹理采样和像素读写操作（属性值为NO）。如果设置为Yes，layer对象可以为显示优化纹理对象。对于大多数的app，建议值是YES。
presentsWithTransaction，layer对象的绘制资源的变化，使用标准的Core Animation变换机制跟新（属性值为YES），或是相对于普通的layer的变化异步地更新（属性值为NO）

If the `nextDrawable` method succeeds, it returns a `CAMetalDrawable` object with the following read-only properties:

The `texture` property holds the texture object. You use this as an attachment when creating your rendering pipeline (`MTLRenderPipelineColorAttachmentDescriptor` object).

The `layer` property points to the `CAMetalLayer` object that responsible for displaying the drawable.

Calling the `nextDrawable` method of `CAMetalLayer` blocks its CPU thread until the method is completed. There are only a small set of drawable resources, so a long frame rendering time could temporarily exhaust those resources, which causes a `nextDrawable` call to block. Because of this possibility, it is always a good idea to delay the call to `nextDrawable` as long as possible with regard to other work being performed on the CPU, which helps hide any latency incurred by long GPU frame times.

To call the `present` method for a `CAMetalDrawable` object when the command buffer has been scheduled on the device, you can call either the `presentDrawable:` or `presentDrawable:atTime:` convenience method on a`MTLCommandBuffer` object. The `presentDrawable:` and `presentDrawable:atTime:` methods use the scheduled handler to present one drawable, which covers most scenarios. The `presentDrawable:atTime:` method provides further control over when the drawable is presented.

如果CAMetalLayer的nextDrawable方法成功，它返回一个CAMetalDrawable对象，这个对象具有以下的只读的属性：
texture，它持有一个纹理对象，当设定渲染管线（MTLRenderPipelineColorAttachmentDescriptor对象）时，你可以用它作为一个attachment对象。
layer，指向CAMetalLayer对象，它和显示相关。

调用nextDrawable方法将阻断CPU线程直到方法完成。系统只有为数不多的drawable资源，所以一个长时间的帧绘制将短暂耗尽这些资源，如此导致nextDrawable方法阻断线程。因为存在这种可能，尽可能推延nextDrawable的调用，而让其他的任务在CPU上优先完成为好，这将有助于利用较长的GPU帧绘制时间掩盖nextDrawable造成的耗时。

当command buffer已经在设备上排进队列，就可以为CAMetalDrawable对象调用present方法。可以使用aMTLCommandBuffer对象的presentDrawable方法或presentDrawable:atTime:方法。这两个方法使用预备好的handler（包含了所有的绘制逻辑）来呈现一个drawable对象，，后者在drawable对象被绘制时提供更多控制。

Creating a Render Pipeline State
创建一个绘制管线state对象

To use a `MTLRenderCommandEncoder` object to encode rendering commands, you must first specify a `MTLRenderPipelineState` object to define the graphics state for any draw calls. A render pipeline state object is a

long-lived persistent object that can be created outside of a render command encoder, cached in advance, and reused across several render command encoders. When describing the same set of graphics state, reusing a previously created render pipeline state object may avoid expensive operations that re-evaluate and translate the specified state to GPU commands.

To create a `MTLRenderPipelineState`, first create a `MTLRenderPipelineDescriptor`, which has properties that describe the graphics rendering pipeline state you want to use during the rendering pass, as depicted in Figure 5-2 (page 38).

The `colorAttachments` property of the new `MTLRenderPipelineDescriptor` object contains an array of `MTLRenderPipelineColorAttachmentDescriptor` objects, and each descriptor represents a color attachment state that specifies the blend operations and factors for that attachment, as

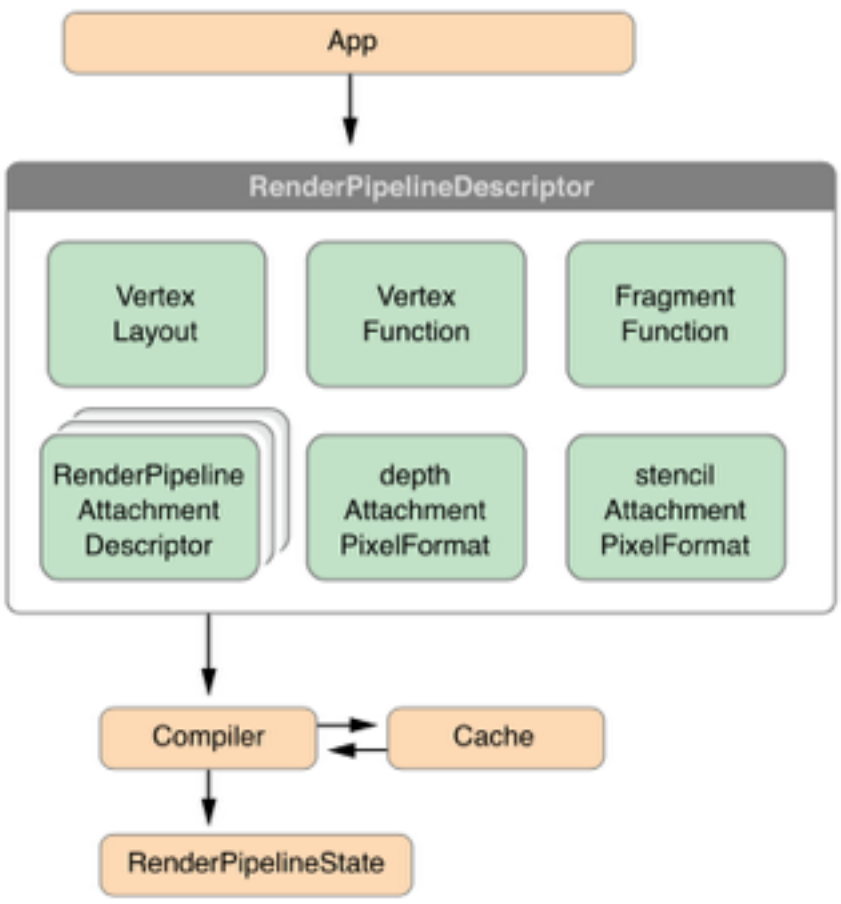
detailed in Configuring Blending in a Render Pipeline Attachment Descriptor (page 40). The attachment descriptor also specifies the pixel format of the attachment, which must match the pixel format for the texture of the `MTLRenderPipelineDescriptor` with the corresponding attachment index, or an error occurs.

使用MTLRenderCommandEncoder对象来编码绘制指令，你必须先制定一个MTLRenderPipelineState对象，它是用来为每个绘制命令调用定义图形状态，一个state对象是一个拥有长生命周期的对象，它可以在encoder对象生效范围之外被创建，最好被缓存起来，然后被重用于多个encoder对象。描述相同的图形绘制状态，重用先前创建的state对象可以避免高成本的转换操作（将特定状态转换成GPU指令的操作）

要创建一个state对象，首先创建一个MTLRenderPipelineDescriptor对象，它含有的属性可以描述在一个绘制pass中使用到的图形绘制管线状态，如图5-2所示。

MTLRenderPipelineDescriptor对象的colorAttachments属性包含了一个MTLRenderPipelineColorAttachmentDescriptor对象数组，其中的每个descriptor对象表示一个颜色attachment状态，这个状态为attachment指定混合操作类型和因子。descriptor对象还设定了像素格式，这个像素格式必须和MTLRenderPipelineDescriptor的相应attachment下标的texture相匹配，否则会产生错误。

Figure 5-2 Creating a Render Pipeline State from a Descriptor



Set these properties in `MTLRenderPipelineDescriptor`:

Set the `depthAttachmentPixelFormat` property to match the pixel format for the texture of `depthAttachment` in `MTLRenderPassDescriptor`.

Set the `stencilAttachmentPixelFormat` property to match the pixel format for the texture of `stencilAttachment` in `MTLRenderPassDescriptor`.

To specify the vertex or fragment shader in the render pipeline state, set the `vertexFunction` or `fragmentFunction` property, respectively. Setting `fragmentFunction` to `nil` disables the rasterization of pixels into the specified color

attachment, which is typically used for depth-only rendering or for outputting data into a buffer object from the vertex shader.

If the vertex shader has an argument with per-vertex input attributes, set the `vertexDescriptor` property to describe the organization of the vertex data in that argument, as described in Vertex Descriptor for Data Organization (page 44).

In the unusual situation when you may need to disable rasterization (e.g., to gather data from vertex-only transformations), set the `rasterizationEnabled` property to `NO`. In most cases, use the default value of `YES`.

If the attachment supports multisampling (i.e., is a `MTLTextureType2DMultisample` type texture), then multiple samples can be created per pixel, and the following `MTLRenderPipelineDescriptor` properties are set to determine coverage.

`sampleCount` is the number of samples for each pixel. When `MTLRenderCommandEncoder` is created, the `sampleCount` for the textures for all attachments must match this `sampleCount` property. If the attachment cannot support multisampling, then `sampleCount` is 1, which is also the default value.

If `alphaToCoverageEnabled` is set to `YES`, then the alpha channel fragment output for `colorAttachments[0]` is read and used to determine a coverage mask.

If `alphaToOneEnabled` is set to `YES`, then alpha channel fragment values for `colorAttachments[0]` are forced to 1.0, which is the largest representable value. (Other attachments are unaffected.)

为MTLRenderPipelineDescriptor对象设置属性：

depthAttachmentPixelFormat，设置为和 MTLRenderPassDescriptor 对象的depthAttachment的texture像素格式一致。
stencilAttachmentPixelFormat，设置为和 MTLRenderPassDescriptor 对象的stencilAttachment的texture像素格式一致。
通过设置vertexFunction来设定绘制管线中的顶点着色程序，设置fragmentFunction来设定绘制管线中的片元着色程序，如果fragmentFunction设置为nil，将使得像素光栅化不可用（无法将渲染结果写入特定的颜色attachment），通常只有这两种情况会做如此的设定，只和深度相关的绘制 或是 从顶点着色器输出数据到特定缓存。
如果顶点着色程序具有描述每像素输入属性的参数，则可以设置vertexDescriptor属性来描述顶点数据的组织方式。
在某些不常见的情况下，当你需要关闭光栅化（比如仅收集由顶点着色程序计算出来的结果），设置rasterizationEnabled为NO，在其他的情况下，使用默认值YES。
如果attachment支持多重采样（比如一个MTLTextureType2DMultisample类型的纹理），多重采样器会被创建，下面几个属性被设置来确定覆盖范围。
sampleCount表示每像素的采样数，当MTLRenderCommandEncoder对象被创建，它的sampleCount属性值必须和这个一致。如果attachment不支持多重采样，sampleCount为1，这也是该属性的默认值。
如alphaToCoverageEnabled果设置为YES，位于colorAttachments[0]的attachement对象，其alpha通道将被读取并用来做成一个覆盖遮罩。
如alphaToOneEnabled果设置为YES，位于colorAttachments[0]的attachement对象，其alpha通道将被设置1.0，这是最大的有效值（其他的attachment对象不受影响）

After creating `MTLRenderPipelineDescriptor` and specifying its properties, use it to create the `MTLRenderPipelineState`. Reflection data that reveals details of the shader function and its arguments may be created along with the `MTLRenderPipelineState`. Avoid obtaining reflection data if it will not be used. For more information on how to analyze reflection data, see Determining Function Details at Runtime (page 28).

Creating a `MTLRenderPipelineState` can require an expensive evaluation of graphics state and a possible compilation of the specified graphics shaders, so you can use either a blocking or asynchronous method.

To synchronously compile the graphics state and create the render pipeline state object, call either the `newRenderPipelineStateWithDescriptor:error:` or `newRenderPipelineStateWithDescriptor:options:reflection:error:` method of `MTLDevice`. The latter method also returns reflection data of the shader function.

To asynchronously compile the graphics state and register a handler to be called when the render pipeline state object is created, call either the `newRenderPipelineStateWithDescriptor:completionHandler:` or `newRenderPipelineStateWithDescriptor:options:completionHandler:` method of `MTLDevice`. The latter method also returns reflection data of the shader function.

Then call the `setRenderPipelineState:` method of `MTLRenderCommandEncoder` to assign the `MTLRenderPipelineState`.

The `reset` method of `MTLRenderPipelineDescriptor` can be used to specify the default pipeline state values for the descriptor.

当创建MTLRenderPipelineDescriptor对象并设置好它的各属性值后，就可以用它来创建MTLRenderPipelineState对象了。描述着色程序和它的参数关系的反射数据也可以和MTLRenderPipelineState对象一同创建。如果不使用反射数据，就不要创建它。在“运行时决定函数细节”章节讲述如何分析反射数据

创建一个对象需要一个耗时的图形状态诊断并且可能伴随特定的着色程序编译，所以你可以使用阻塞式调用或是异步调用。

同步（阻塞式）编译图形绘制状态并创建state对象可以调用MTLDevice的newRenderPipelineStateWithDescriptor:error:方法或是newRenderPipelineStateWithDescriptor:options:reflection:error:方法，后者将返回着色程序的反射数据。

异步编译图形绘制状态并且为state对象创建成功注册回调函数，可以调用MTLDevice的newRenderPipelineStateWithDescriptor:completionHandler:方法或是newRenderPipelineStateWithDescriptor:options:completionHandler:方法，后者将返回着色程序的反射数据。

MTLRenderCommandEncoder 的 setRenderPipelineState方法用来设置encoder对象的state属性

MTLRenderPipelineDescriptor的reset方法用于设置绘制管线为默认值。

Listing 5-4 (page 40) demonstrates the creation of a render pipeline state object called `pipeline`. `vertFunc` and `fragFunc` are shader functions that are specified as properties of the render pipeline state descriptor called `renderPipelineDesc`. Calling the `newRenderPipelineStateWithDescriptor:error:` method of `MTLDevice` synchronously uses the pipeline state descriptor to create the render pipeline state object. Then

calling the `setRenderPipelineState:` method of `MTLRenderCommandEncoder` specifies the `MTLRenderPipelineState` to use with the render command encoder. Remember that the `MTLRenderPipelineState` object is expensive to create, so you are encouraged to reuse it whenever you want to use the same graphics state.

代码5-4示例了如何创建一个叫做pipeline的state对象，vertFunc和fragFunc是着色程序，它们被赋值给一个叫renderPipelineDesc的state descriptor对象，使用这个state descriptor对象为参数，调用MTLDevice的newRenderPipelineStateWithDescriptor:error:方法（阻塞式）就创建了state对象。然后调用MTLRenderCommandEncoder的setRenderPipelineState方法，指定刚刚创建的state被encoder对象用于绘制。记住，state对象的创建开销很大，如果你使用的图形状态相同，就尽量重用它。

Listing 5-4 Creating a Simple Pipeline State

```
MTLRenderPipelineDescriptor *renderPipelineDesc =
                              [[MTLRenderPipelineDescriptor alloc] init];
renderPipelineDesc.vertexFunction = vertFunc;
renderPipelineDesc.fragmentFunction = fragFunc;
renderPipelineDesc.colorAttachments[0].pixelFormat = MTLPixelFormatRGBA8Unorm;
// Create MTLRenderPipelineState from MTLRenderPipelineDescriptor
NSError *errors = nil;
id <MTLRenderPipelineState> pipeline = [device
         newRenderPipelineStateWithDescriptor:renderPipelineDesc error:&errors];
assert(pipeline && !errors);
// Set the pipeline state for MTLRenderCommandEncoder
[renderCE setRenderPipelineState:pipeline];
```

Configuring Blending in a Render Pipeline Attachment Descriptor
在attachment descriptor中配置混合

Blending uses a highly configurable blend operation to mix the output returned by the fragment function (source) with pixel values in the attachment (destination). Blend operations determine how the source and destination values are combined with blend factors. To configure blending for a color attachment, set the following `MTLRenderPipelineColorAttachmentDescriptor` properties:

To enable blending, set `blendingEnabled` to `YES`. Blending is disabled, by default. `writeMask` identifies which color channels are blended. The default value `MTLColorWriteMaskAll` allows all color channels to be blended.

`rgbBlendOperation` and `alphaBlendOperation` separately assign the blend operations for the RGB and Alpha fragment data with a `MTLBlendOperation` value. The default value for both properties is `MTLBlendOperationAdd`.

`sourceRGBBlendFactor`, `sourceAlphaBlendFactor`, `destinationRGBBlendFactor`, and `destinationAlphaBlendFactor` assign the source and destination blend factors.

Four blend factors refer to a constant blend color value: `MTLBlendFactorBlendColor`, `MTLBlendFactorOneMinusBlendColor`, `MTLBlendFactorBlendAlpha`, and `MTLBlendFactorOneMinusBlendAlpha`. Call the `setBlendColorRed:green:blue:alpha:` method of `MTLRenderCommandEncoder` to specify the constant color and alpha values used with these blend factors, as described in Fixed-Function State Operations (page 48).

混合是使用高度可配置的混色操作，将片元着色器的输出（作为源）和 attachment中的像素值（作为目标）进行计算。混色操作决定如何将源和目标值按混合因子进行组合。配置颜色attachment的混合操作，需要设置如下的`MTLRenderPipelineColorAttachmentDescriptor`对象的属性值：

`blendingEnabled`，如果要使混合操作生效，设置该属性为YES，默认是NO，不生效。

`writeMask`，设置哪个颜色通道参与混合，默认值是`MTLColorWriteMaskAll`，表示所有的颜色通道都参与混合。

`rgbBlendOperation`，设置片元数据的RGB如何混合（用`MTLBlendOperation`类型的值），默认值是`MTLBlendOperationAdd`。

`alphaBlendOperation`，设置片元数据的alpha如何混合，默认值也是`MTLBlendOperationAdd`。

`sourceRGBBlendFactor`

`sourceAlphaBlendFactor`

`destinationRGBBlendFactor`

`destinationAlphaBlendFactor`

以上4个属性用于设置源和目标的混合因子。这4个混合因子和如下的混合颜色常量值相关：

`MTLBlendFactorBlendColor`

`MTLBlendFactorOneMinusBlendColor`

`MTLBlendFactorBlendAlpha`

`MTLBlendFactorOneMinusBlendAlpha`

MTLRenderCommandEncoder的setBlendColorRed:green:blue:alpha:方法来设定颜色常量和alpha常量，将和这些混合因子起作用。

Some blend operations combine the fragment values by multiplying the source values by a source `MTLBlendFactor` (abbreviated SBF), multiplying the destination values by a destination `MTLBlendFactor` (DBF), and combining the results using the arithmetic indicated by the `MTLBlendOperation` value. (If the `MTLBlendOperation` value is either `MTLBlendOperationMin` or `MTLBlendOperationMax`, the SBF and DBF blend factors are ignored.) For example,

`MTLBlendOperationAdd` for both `rgbBlendOperation` and `alphaBlendOperation` properties defines the following additive blend operation for RGB and Alpha values:

RGB = (Source.rgb * `sourceRGBBlendFactor`) + (Dest.rgb * `destinationRGBBlendFactor`)

Alpha = (Source.a * `sourceAlphaBlendFactor`) + (Dest.a * `destinationAlphaBlendFactor`)

To get the default blend behavior, where the source completely overwrites the destination, set both the `sourceRGBBlendFactor` and `sourceAlphaBlendFactor` to `MTLBlendFactorOne`, and the `destinationRGBBlendFactor` and `destinationAlphaBlendFactor` to `MTLBlendFactorZero`. This behavior is expressed mathematically as:

RGB = (Source.rgb * 1.0) + (Dest.rgb * 0.0)

Alpha = (Source.a * 1.0) + (Dest.a * 0.0)

Another commonly used blend operation, where the source alpha defines how much of the destination color remains, can be expressed mathematically as:

RGB = (Source.rgb * 1.0) + (Dest.rgb * (1 - Source.a))

Alpha = (Source.a * 1.0) + (Dest.a * (1 - Source.a))

有一些混合操作是这样实施的，使用一个源混合因子（简称SBF）和源像素值相乘，使用一个目标混合因子（简称DBF）和目标像素值相乘，最后根据由`MTLBlendOperation`值指定的混合算法去计算最后的色值。（如果`MTLBlendOperation`的值是`MTLBlendOperationMin`或是`MTLBlendOperationMax`，SBF和DBF混合因子被忽略）。举例来说，如果`rgbBlendOperation`和`alphaBlendOperation`的值都是`MTLBlendOperationAdd`，那么RGB和Alpha的色值计算公式如下：

RGB = (Source.rgb * `SBF`) + (Dest.rgb * `DBF`)
Alpha = (Source.a * `SBF`) + (Dest.a * `DBF`)

默认的混合行为是将源完全覆盖目标，这时会将`sourceRGBBlendFactor`和`sourceAlphaBlendFactor`设置为`MTLBlendFactorOne`，并且将`destinationRGBBlendFactor`和`destinationAlphaBlendFactor`设置为`MTLBlendFactorZero`。色值计算公式如下：

RGB = (Source.rgb * 1.0) + (Dest.rgb * 0.0)
Alpha = (Source.a * 1.0) + (Dest.a * 0.0)

另外一个常用的混合操作设置，源的alpha值来决定目标颜色被保留多少，色值计算公式如下：

RGB = (Source.rgb * 1.0) + (Dest.rgb * (1 - Source.a))
Alpha = (Source.a * 1.0) + (Dest.a * (1 - Source.a))

Listing 5-5 (page 41) shows code for a custom blending configuration, using the blend operation `MTLBlendOperationAdd`, the source blend factor `MTLBlendFactorOne`, and the destination blend factor `MTLBlendFactorOneMinusSourceAlpha`. `colorAttachments[0]` is a `MTLRenderPipelineColorAttachmentDescriptor` object with properties that specify the blending configuration.

代码5-5展示如何设定一个混合，使用`MTLBlendFactorOne`混合操作，SBF设置为`MTLBlendFactorOne`，DBF设置为`MTLBlendFactorOneMinusSourceAlpha`，`colorAttachments[0]`是一个`MTLRenderPipelineColorAttachmentDescriptor`对象，混合的设置设定过程就是为它设定属性。代码示例的混合操作用公式表示出来是：

RGB = (Source.rgb * 1.0) + (Dest.rgb * (1 - Source.a))
Alpha = (Source.a * 1.0) + (Dest.a * (1 - Source.a))

Listing 5-5 Specifying A Custom Blend Configuration

```
MTLRenderPipelineDescriptor *renderPipelineDesc =
                                [[MTLRenderPipelineDescriptor alloc] init];
renderPipelineDesc.colorAttachments[0].blendingEnabled = YES;
renderPipelineDesc.colorAttachments[0].rgbBlendOperation = MTLBlendOperationAdd;
renderPipelineDesc.colorAttachments[0].alphaBlendOperation = MTLBlendOperationAdd;
renderPipelineDesc.colorAttachments[0].sourceRGBBlendFactor = MTLBlendFactorOne;
```

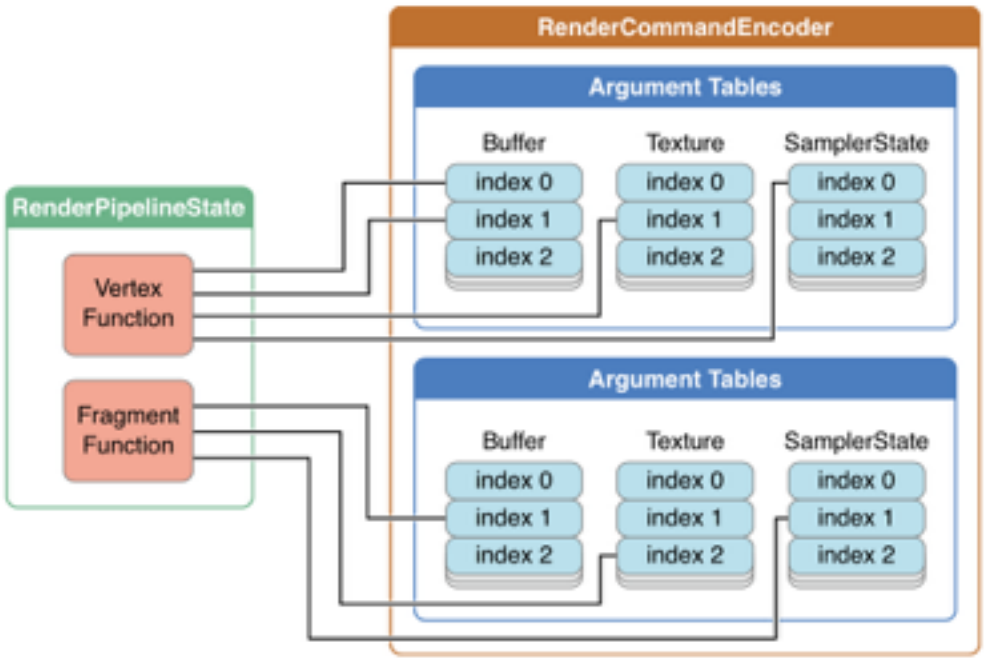
```
renderPipelineDesc.colorAttachments[0].sourceAlphaBlendFactor = MTLBlendFactorOne;
renderPipelineDesc.colorAttachments[0].destinationRGBBlendFactor =
MTLBlendFactorOneMinusSourceAlpha;

renderPipelineDesc.colorAttachments[0].destinationAlphaBlendFactor =
MTLBlendFactorOneMinusSourceAlpha;

NSError *errors = nil;

id <MTLRenderPipelineState> pipeline = [device
newRenderPipelineStateWithDescriptor:renderPipelineDesc error:&errors];
```

Specifying Resources for a Render Command Encoder
为encoder设定绘制资源

The following `MTLRenderCommandEncoder` methods specify resources that are used as arguments for the vertex and fragment shader functions, which are specified by the `vertexFunction` and `fragmentFunction` properties in a `MTLRenderPipelineState` object. These methods assign a shader resource (buffers, textures, and samplers) to the corresponding argument table index (`atIndex`) in the render command encoder, as shown in Figure 5-3 (page 42).

下面这些MTLRenderCommandEncoder的方法为顶点和片元着色程序（着色程序由state对象的vertexFunction和fragmentFunction属性指定）的参数设定绘制资源。这些方法设定一个着色器资源（缓存，纹理，采样器）到encoder的对应的参数表下标位置（atIndex），如图5-3所示。

Figure 5-3 Argument Tables for the Render Command Encoder



The following `setVertex*` methods assign one or more resources to corresponding arguments of a vertex shader function.

下面这些setVertex开头的方法为相应的顶点着色程序参数设置1个或是多个资源。

```
setVertexBuffer:offset:atIndex:
setVertexBuffers:offsets:withRange:
setVertexTexture:atIndex:
setVertexTextures:withRange:
setVertexSamplerState:atIndex:
setVertexSamplerState:lodMinClamp:lodMaxClamp:atIndex:
setVertexSamplerStates:withRange:
setVertexSamplerStates:lodMinClamps:lodMaxClamps:withRange:
```

These `setFragment*` methods similarly assign one or more resources to corresponding arguments of a fragment shader function.

下面这些setFragment开头的方法为相应的片元着色程序参数设置1个或是多个资源。

```
setFragmentBuffer:offset:atIndex:
setFragmentBuffers:offsets:withRange:
```

```
setFragmentTexture:atIndex:
setFragmentTextures:withRange:
setFragmentSamplerState:atIndex:
setFragmentSamplerState:lodMinClamp:lodMaxClamp:atIndex:
setFragmentSamplerStates:withRange:
setFragmentSamplerStates:lodMinClamps:lodMaxClamps:withRange:
```

There are a maximum of 31 entries in the buffer argument table, 31 entries in the texture argument table, and 16 entries in the sampler state argument table.

The attribute qualifiers that specify resource locations in the Metal shading language source code must match the argument table indices in the Metal framework methods.

In Listing 5-6 (page 43), two buffers (`posBuf` and `texCoordBuf`) with indices 0 and 1, respectively, are defined for the vertex shader.

In Listing 5-7 (page 44), the function signature has corresponding arguments with the attribute qualifiers `buffer(0)` and `buffer(1)`.

Similarly in Listing 5-8 (page 44), three resources, a buffer, a texture, and a sampler (`fragmentColorBuf`, `shadeTex`, and `sampler`, respectively), all with index 0, are defined for the fragment shader.

In Listing 5-9 (page 44), the function signature has corresponding arguments with the attribute qualifiers `buffer(0)`, `texture(0)`, and `sampler(0)`, respectively.

在缓存参数表中最多有31个入口，在纹理参数表最多有31个入口，在采样器状态表里最多有16个入口。
在metal着色语言源代码中的用于定位资源的属性修饰符 必须和metal框架代码中相应方法设定的参数表下标 相吻合。

如代码5-6所示，两个缓存（posBuf 和 texCoordBuf）分别对应下标0和1，那么在顶点着色程序也要如此设定。
如代码5-7所示的顶点着色器程序，方法签名有对应的用“[[index]]”修饰的参数声明，表示使用`buffer(0)`和`buffer(1)`。
类似的，如代码5-8所示，3个资源，一个缓存（`fragmentColorBuf`），一个纹理（`shadeTex`），一个采样器（`sampler`），都使用下标0，它们是为一个片元着色程序准备的。
如代码5-9所示的片元着色程序，方法签名有对应的用“[[index]]”修饰的参数声明，表示使用`buffer(0)`和`texture(0)`和`sampler(0)`。

Listing 5-6 Metal Framework: Specifying Resources for a Vertex Function

```
[renderEnc setVertexBuffer:posBuf offset:0 atIndex:0];
[renderEnc setVertexBuffer:texCoordBuf offset:0 atIndex:1];
```

Listing 5-7 Metal Shading Language: Vertex Function Arguments Match the Framework Argument Table Indices

```
vertex VertexOutput metal_vert(float4 *posData [[ buffer(0) ]],
                               float2 *texCoordData [[ buffer(1) ]])
```

Listing 5-8 Metal Framework: Specifying Resources for a Fragment Function

```
[renderEnc setFragmentBuffer:fragmentColorBuf offset:0 atIndex:0];
[renderEnc setFragmentTexture:shadeTex atIndex:0];
[renderEnc setFragmentSamplerState:sampler atIndex:0];
```

Listing 5-9 Metal Shading Language: Fragment Function Arguments Match the Framework Argument Table Indices

```
fragment float4 metal_frag(VertexOutput in [[stage_in]],
                           float4 *fragColorData [[ buffer(0) ]],
                           texture2d<float> shadeTexValues [[ texture(0) ]],
                           sampler samplerValues [[ sampler(0) ]] )
```

Vertex Descriptor for Data Organization
用于数据组织的顶点descriptor

In Metal framework code, there can be one `MTLVertexDescriptor` for every pipeline state that describes the organization of data input to the vertex shader function and shares resource location information between the shading language and framework code.

In Metal shading language code, per-vertex inputs (i.e., scalars or vectors of integer or floating-point values) can be organized in one struct, which can be passed in one argument that is declared with the `[[stage_in]]` attribute qualifier, as seen in the `VertexInput` struct for the example vertex function `vertexMath` in Listing 5-10 (page 44). Each field of the per-vertex input struct has the `[[attribute(index)]]` qualifier, which specifies the index in the vertex attribute argument table.

对于metal框架代码来说（使用oc编写，运行于CPU上），可以为每个绘制管线state准备一个`MTLVertexDescriptor`对象，它用来描述顶点着色程序使用的输入数据如何组织，以及资源在框架代码和着色程序代码（使用metal着色语言编写，运行于GPU上）之间的映射信息

To refer to the shader function input from Metal framework code, describe a `MTLVertexDescriptor` object and then set it as the `vertexDescriptor` property of `MTLRenderPipelineState`. `MTLVertexDescriptor` has two properties: `attributes` and `layouts`.

The `attributes` property of `MTLVertexDescriptor` is a `MTLVertexAttributeDescriptorArray` that defines how each vertex attribute is organized in a buffer that is mapped to a vertex function argument. The `attributes` property can support access to multiple attributes (such as vertex coordinates, surface normals, and texture coordinates) that are interleaved within the same buffer. The order of the members in the shading language code does not have to be preserved in the buffer in the framework code. Each `MTLVertexAttributeDescriptor` in the array has the following properties that provide a vertex shader function information to locate and load the argument data:

`bufferIndex`, which is an index to the buffer argument table that specifies which `MTLBuffer` is accessed. (The buffer argument table was discussed in Specifying Resources for a Render Command Encoder (page 42).)

`format`, which specifies how the data should be interpreted in the framework code. If the data type is not an exact type match, it may be converted or expanded. For example, if the shading language type is `half4` and the framework `format` is `MTLVertexFormatFloat2`, then when the data is used as an argument to the vertex function, it may be converted from float to half and expanded from two to four elements (with 0.0, 1.0 in the last two elements).

`offset`, which specifies where the data can be found from the start of a vertex.

Figure 5-4 (page 46) illustrates a `MTLVertexAttributeDescriptorArray` in Metal framework code that implements an interleaved buffer that corresponds to the input to the vertex function `vertexMath` in the shading language code in Listing 5-10 (page 44).

Listing 5-11 (page 46) shows the Metal framework code that corresponds to the interleaved buffer shown in Figure 5-4 (page 46). Each `MTLVertexAttributeDescriptor` object in the `attributes` array of the `MTLVertexDescriptor` object corresponds to the indexed struct member in `VertexInput` in the shader function. `attributes[1].bufferIndex = 0` specifies the use of the buffer at index 0 in the argument table. (In this example, each `MTLVertexAttributeDescriptor` has the same `bufferIndex`, so each refers to the same vertex buffer at index 0 in the argument table.) The `offset` values specify the location of data within the vertex, so `attributes[1].offset = 2 * sizeof(float)` locates the start of the corresponding data 8 bytes from the start of the buffer. The `format` values are chosen to match the data type in the shader function, so `attributes[1].format = MTLVertexFormatFloat4` specifies the use of four floating-point values.

从metal框架代码中为着色程序指定输入，制作并设置好一个`MTLVertexDescriptor`对象，然后设置state对象的`vertexDescriptor`属性。`MTLVertexDescriptor`对象有两个属性：attributes和layouts。

`attributes`，是一个`MTLVertexAttributeDescriptor`类型数组，它定义在一个缓存中每个顶点的各属性如何组织并且映射到一个顶点着色程序的入参中。该`attributes`属性可以支持访问在同一块缓存的交叉存放的多重顶点数据信息（比如顶点坐标，表面法向量，纹理坐标）。在着色程序和框架程序中，这些交替存放的信息的顺序可以不相同。在`attributes`数组中得每个descriptor对象有如下3个属性，他们用于为顶点着色程序提供定位和加载入参数据：

`bufferIndex`，是一个缓存参数表（在上一个章节中讲述）的下标，它标识哪个`MTLBuffer`对象将被访问。

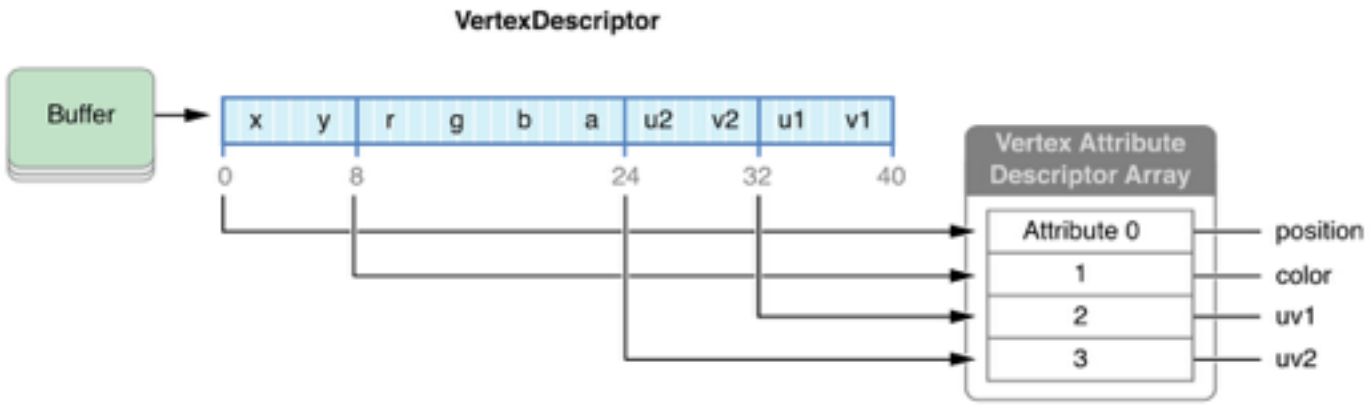
`format`，设定在metal框架代码中数据是如何被解释的。如果数据类型不是完全符合，它将被转换或是延展。比如，如果着色语言中类型为`half4`（4维向量且每个分量为half类型），框架代码中`format`是`MTLVertexFormatFloat2`，那当数据被顶点着色程序用作入参的时候，它将从float转换为half并且从2维向量变成4维向量（后面多出来的两个分量将被赋予0.0 和 1.0）。

offset，指定从一个顶点缓存哪个位置获取数据。

代码5-10 是一段叫做 vertexMath的顶点着色程序，图5-4示例了在框架代码中的一个MTLVertexAttributeDescriptor数组，它描述了缓存中顶点数据如何交替，如何为顶点着色程序vertexMath提供入参。

代码5-11展示了框架代码如何实现图5-4描述的缓存。中的每一个对象

Figure 5-4 Buffer Organization with Vertex Attribute Descriptors



Listing 5-10 Metal Shading Language: Vertex Function Inputs with Attribute Indices

```
struct VertexInput {
      float2    position [[attribute(0)]];
      float4    color [[attribute(1)]];
      float2    uv1 [[attribute(2)]];
      float2    uv2 [[attribute(3)]];
};
struct VertexOutput {
      float4 pos [[position]];
      float4 color;
};

vertex VertexOutput vertexMath(VertexInput in [[stage_in]])
{
    VertexOutput out;
    out.pos = float4(in.position.x, in.position.y, 0.0, 1.0);
    float sum1 = in.uv1.x + in.uv2.x;
    float sum2 = in.uv1.y + in.uv2.y;
    out.color = in.color + float4(sum1, sum2, 0.0f, 0.0f);
    return out;
}
```

Listing 5-11 Metal Framework: Using a Vertex Descriptor to Access Interleaved Data

```
id <MTLFunction> vertexFunc = [library newFunctionWithName:@"vertexMath"];
MTLRenderPipelineDescriptor* pipelineDesc = [[MTLRenderPipelineDescriptor alloc] init];
MTLVertexDescriptor* vertexDesc = [[MTLVertexDescriptor alloc] init];
// 设置attributes
// x y
vertexDesc.attributes[0].format = MTLVertexFormatFloat2;
vertexDesc.attributes[0].bufferIndex = 0;
vertexDesc.attributes[0].offset = 0;
// rgba
vertexDesc.attributes[1].format = MTLVertexFormatFloat4;
vertexDesc.attributes[1].bufferIndex = 0;
vertexDesc.attributes[1].offset = 2 * sizeof(float);  // 8 bytes
// u1 v1
vertexDesc.attributes[2].format = MTLVertexFormatFloat2;
vertexDesc.attributes[2].bufferIndex = 0;
vertexDesc.attributes[2].offset = 8 * sizeof(float);  // 32 bytes
// u2 v2
vertexDesc.attributes[3].format = MTLVertexFormatFloat2;
vertexDesc.attributes[3].bufferIndex = 0;
vertexDesc.attributes[3].offset = 6 * sizeof(float);  // 24 bytes
// 设置layouts
vertexDesc.layouts[0].stride = 10 * sizeof(float);    // 40 bytes
vertexDesc.layouts[0].stepFunction = MTLVertexStepFunctionPerVertex;
//
pipelineDesc.vertexDescriptor = vertexDesc;
```

```
pipelineDesc.vertexFunction = vertFunc;
```

The `layouts` property of `MTLVertexDescriptor` is a `MTLVertexBufferLayoutDescriptor Array`. For each `MTLVertexBufferLayoutDescriptor` in `layouts`, the properties specify how vertex and attribute data are fetched from the corresponding `MTLBuffer` in the argument table when drawing primitives. (For more on drawing primitives, see Drawing Geometric Primitives (page 51).) The `stepFunction` property of `MTLVertexBufferLayoutDescriptor` determines whether to fetch attribute data for every vertex, for some number of instances, or just once. If `stepFunction` is set to fetch attribute data for some number of instances, then the `stepRate` property of `MTLVertexBufferLayoutDescriptor` determines how many instances. The `stride` property specifies the distance between the data of two vertices, in bytes.

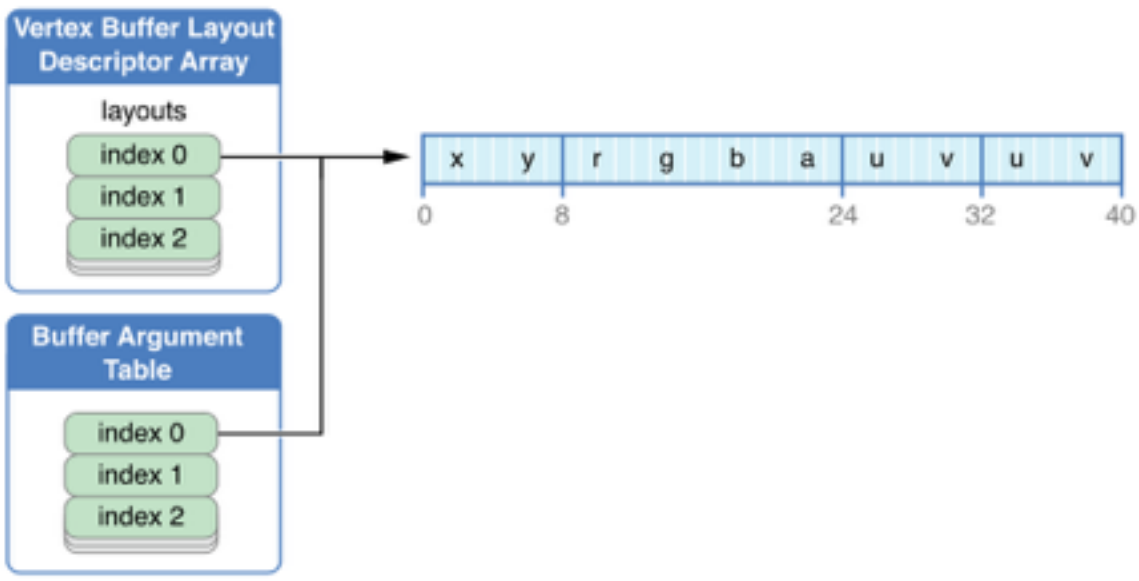
`MTLVertexDescriptor`对象的`layouts`属性是一个`MTLVertexBufferLayoutDescriptor`数组，在layouts数组中的每一个descriptor对象的属性值描述了当绘制图元的时候如何从相应的`MTLBuffer`对象中获取顶点和其属性数据。descriptor的`stepFunction`属性决定了是为每个顶点获取属性数据、还是一些图元、还是只获取1次；如果`stepFunction`被设置为为图元，那么descriptor的`stepRate`属性决定有多少图元；descriptor的`stride`属性决定相邻两个顶点数据间的间隔（多少字节）。

Figure 5-5 (page 47) depicts the `MTLVertexBufferLayoutDescriptor` that corresponds to the code in Listing 5-11 (page 46). `layouts[0]` specifies how vertex data are fetched from corresponding index 0 in the buffer argument table. `layouts[0].stride` specifies a distance of 40 bytes between the data of two vertices. The value of `layouts[0].stepFunction`, `MTLVertexStepFunctionPerVertex`, specifies that attribute data is fetched for every vertex when drawing. (If the value of `stepFunction` is `MTLVertexStepFunctionPerInstance`, then the `stepRate` property determines how often attribute data is fetched. For example, if `stepRate` is 1, then data is fetched for every instance; if `stepRate` is 2, for every two instances, etc.)

图5-5示例了代码5-11的设置效果。`layouts[0]`设定了如何从对应的下标为0的缓存参数表中获取顶点数据。`layouts[0].stride`表示相邻两个顶点数据相隔40字节。`layouts[0].stepFunction`被设置为`MTLVertexStepFunctionPerVertex`表示当每个顶点顶点绘制的时候都要获取顶点属性数据。

如果`stepFunction`被设置为`MTLVertexStepFunctionPerInstance`，那么`stepRate`属性决定获取顶点属性数据的频率，举例来说，如果`stepRate`是1，每个图元都获取数据；如果`stepRate`是2，每两个图元获取一次数据。

Figure 5-5 Buffer Organization with Vertex Buffer Layout Descriptors



Fixed-Function Render Command Encoder Operations
Encoder对象的固定绘制管线操作控制

The `MTLRenderCommandEncoder` methods listed below set fixed-function graphics state values:

`setViewport:` specifies the region, in screen coordinates, which is the destination for the projection of the virtual 3D world. The viewport is 3D, so it includes depth values.

`setTriangleFillMode:` determines whether to rasterize triangle and triangle strip primitives with lines (`MTLTriangleFillModeLines`) or as filled triangles (`MTLTriangleFillModeFill`). The default value is `MTLTriangleFillModeFill`.

`setCullMode:` and `setFrontFacingWinding:` are used together to determine if and how culling is applied. Culling can be used for hidden surface removal on some geometric models, such as an *orientable* sphere rendered with filled triangles. (A surface is orientable if its primitives are consistently drawn in either clockwise or counter-clockwise order.)

The value of `setFrontFacingWinding:` indicates whether a front-facing primitive has its vertices drawn in clockwise (`MTLWindingClockwise`) or counter-clockwise (`MTLWindingCounterClockwise`) order. The default value is `MTLWindingClockwise`.

The value of `setCullMode:` determines whether to perform culling (`MTLCullModeNone`, if culling disabled) or which type of primitive to cull (`MTLCullModeFront` or `MTLCullModeBack`).

The following `MTLRenderCommandEncoder` methods encode fixed-function state change commands.

`setScissorRect:` specifies a 2D scissor rectangle. Fragments that lie outside the specified scissor rectangle are discarded.

`setDepthStencilState:` sets the depth and stencil test state as described in Depth and Stencil States (page 49).

`setStencilReferenceValue:` specifies the stencil reference value. `setDepthBias:slopeScale:clamp:` specifies an adjustment for comparing shadow maps to the depth values output from fragment shaders.

`setVisibilityResultMode:offset:` determines whether to monitor if any samples pass the depth and stencil tests. If set to `MTLVisibilityResultModeBoolean`, then if any samples pass the depth and stencil tests, a non-zero value is written to a buffer specified by the `visibilityResultBuffer` property of `MTLRenderPassDescriptor` (described in Creating a Render Pass Descriptor (page 32)). If a bounding box is drawn and no samples pass, then the app may conclude that objects within that bounding box are occluded and those objects do not have to be drawn.

`setBlendColorRed:green:blue:alpha:` specifies the constant blend color and alpha values, as detailed in Configuring Blending in a Render Pipeline Attachment Descriptor (page 40).

Encoder对象有如下用于设置固定图形绘制管线状态的方法：

`setViewport:`，在屏幕坐标系下设定了一个区域，改区域是虚拟3维场景的投影目标，用于设置的视口值是3维的，它含有深度值。
`setTriangleFillMode:`，用于决定如何填充三角形或是三角条带图元，`MTLTriangleFillModeLines`表示仅描绘线条，`MTLTriangleFillModeFill`表示填充描绘，默认值是`MTLTriangleFillModeFill`。
`setFrontFacingWinding:`，指定图元的正面绘制是顺时针方向处理顶点（`MTLWindingClockwise`）还是逆时针方向处理顶点（`MTLWindingCounterClockwise`），默认值是顺时针方向。
`setCullMode:`，设置剔除模式，`MTLCullModeNone`不剔除，`MTLCullModeFront`正面剔除，`MTLCullModeBack`背面剔除。

下面几个的方法用于编码图形绘制状态变更指令：

`setScissorRect:`，设定一个2维的裁剪矩形，位于该矩形外侧的图元将被丢弃。
`setDepthStencilState:`，设置深度和模板缓存状态。
`setStencilReferenceValue:`，设置模板缓存参考值。
`setDepthBias:slopeScale:clamp:`，设置一个偏移量，它用于比较阴影位图 和 深度值（由片元着色器的输出）
`setVisibilityResultMode:offset:`，设置如果一个采样器通过深度和模板测试，是否要被监视。当设置为`MTLVisibilityResultModeBoolean`时，采样通过深度和模板测试，一个非零的值被写入由`MTLRenderPassDescriptor`对象的`visibilityResultBuffer`属性所指定的缓存中。如果绘制的是一个包围盒，而它没有采样通过测试，那么app可以推断在包围盒内的物体不可见，这些物体是无需绘制的。
`setBlendColorRed:green:blue:alpha:`，设置常量混合色值和alpha值。

Coordinate Systems
坐标系

Metal defines its Normalized Device Coordinate (NDC) system as a 2x2x1 cube with its center at (0, 0, 0.5). The left and bottom for x and y, respectively, of the NDC system are specified as -1. The right and top for x and y, respectively, of the NDC system are specified as +1.

The viewport specifies the transformation from NDC to the window coordinates. The Metal viewport is a 3D transformation specified by the `setViewport:` method of `MTLRenderCommandEncoder`. The origin of the window coordinates is in the upper-left corner.

In Metal, pixel centers are offset by (0.5, 0.5). For example, the pixel at the origin has its center at (0.5, 0.5); the center of the adjacent pixel to its right is (1.5, 0.5). This is also true for textures.

metal定义了一个归一化的设备坐标系（NDC），它是一个2x2x1的立方体，它的中心坐标为(0, 0, 0.5)。其左边界为x的-1，底边界为y的-1，右边界为x的+1，上边界为y的+1.
视口设定了NDC到窗口坐标系的转换，Metal的视口是一个由Encoder对象的`setViewport:`方法指定的3维向量。窗坐标系的原点在窗口的左上角。
在Metal中，像素的中心被偏移了(0.5, 0.5),比如，在原点的像素其中心位于（0.5, 0.5），这个像素点右边毗邻的那个像素，其中心位于（1.5, 0.5）。对于纹理也是这样的。

Depth and Stencil States
深度和模板状态

The depth and stencil operations are fragment operations that are specified as follows:

1. Specify a custom **`MTLDepthStencilDescriptor`** object that contains settings for the depth/stencil state. Creating a custom **`MTLDepthStencilDescriptor`** object may require creating one or two **`MTLStencilDescriptor`** objects that are applicable to front-facing primitives and back-facing primitives.

2. Create a **`MTLDepthStencilState`** object by calling the **`newDepthStencilStateWithDescriptor:`** method of **`MTLDevice`** with a depth/stencil state descriptor.

3. To set the depth/stencil state, call the **`setDepthStencilState:`** method of **`MTLRenderCommandEncoder`** with the **`MTLDepthStencilState`**.

4. If the stencil test is in use, call **`setStencilReferenceValue:`** to specify the stencil reference value.

If the depth test is enabled, there must be a depth attachment to support writing the depth value. To perform the stencil test, there must be a stencil attachment.

If you will be changing the depth/stencil state regularly, then you may want to reuse the state descriptor object, modifying its property values as needed to create more state objects.

Note: To sample from a depth-format texture within a shader function, implement the sampling operation within the shader without using `MTLSamplerState`.

深度和模板测试都是片元操作，它们的设定方法如下：
1. 设定一个**`MTLDepthStencilDescriptor`**对象，它包含着深度/模板状态信息。这个对象的创建需要一个或是两个**`MTLStencilDescriptor`**对象来对应正面图元和背面图元。
2. 调用**`MTLDevice`**的**`newDepthStencilStateWithDescriptor:`**方法创建一个**`MTLDepthStencilState`**对象，使用步骤1设置号的descriptor对象。
3. 调用Encoder的**`setDepthStencilState:`**方法，将步骤2创建的state对象设置给encoder。
4. 如果模板测试可用，调用Encoder的**`setStencilReferenceValue:`**方法设定模板测试参考值。

如果深度测试可用，必须有一个深度attachment对象用于写入深度值。如果应用模板测试，也必须有一个模板attachment。
如果需要周期性地改变深度/模板状态，可以重用state descriptor对象。重用descritor来创建新的state对象时，如果需要，可以改变descritor对象的属性值。
注意：如果在一个着色程序中从深度格式的纹理做采样，那么在着色器中实现采样操作不要使用`MTLSamplerState`。

Using a Depth/Stencil Descriptor
使用一个深度/模板Descriptor

The following properties of the `MTLDepthStencilDescriptor` object are used to set the depth and stencil state.

To enable writing the depth value to the depth attachment, set `depthWriteEnabled` to `YES`.

`depthCompareFunction` specifies how the depth test is performed. If a fragment's depth value fails the depth test, the fragment is discarded. `MTLCompareFunctionLess` is a commonly used for `depthCompareFunction`, because fragment values that are further away from the viewer than the pixel depth value (a previously written fragment) will fail the depth test and be considered occluded by the earlier depth value.

The `frontFaceStencil` and `backFaceStencil` properties each specify a separate `MTLStencilDescriptor` for front and back-facing primitives. To use the same stencil state for both front and back-facing primitives, you can assign the same `MTLStencilDescriptor` to both `frontFaceStencil` and `backFaceStencil` properties. To explicitly disable the stencil test for one or both faces, set the corresponding property to `nil`, the default value.

Explicit disabling of a stencil state is not necessary. Metal determines whether to enable a stencil test based on whether the stencil test is effectively a no-op.

Listing 5-12 (page 50) shows an example of creation and use of a `MTLDepthStencilDescriptor` object for the creation of a `MTLDepthStencilState` object, which is used by the render command encoder object `renderEnc`. In this example, the stencil state for the front-facing primitives is accessed from the `frontFaceStencil` property of the depth/stencil state descriptor. The stencil test is explicitly disabled for the back-facing primitives.

The following properties define a stencil test in the `MTLStencilDescriptor`:

`stencilCompareFunction` specifies how the stencil test is performed for fragments.
`readMask` is a bitmask that is ANDed to both the stencil reference value and the stored stencil value. The stencil test is a comparison between the resulting masked reference value and the masked stored value.

`writeMask` is a bitmask that restricts which stencil values are written to the stencil attachment by the stencil operations.

`stencilFailureOperation`, `depthFailureOperation`, and `depthStencilPassOperation` specify what to do to a stencil value stored in the stencil attachment for three different test outcomes: if the stencil test fails, if the stencil test passes and the depth test fails, or if both stencil and depth tests succeed, respectively.

如下的MTLDepthStencilDescriptor对象的属性值被用于设置深度和模板状态：

depthCompareFunction，指定深度测试如何进行。如果一个片元的深度值没有通过深度测试，该片元被丢弃。MTLCompareFunctionLess是通常用来设置该属性值的。因为当前进行测试的片元的深度值相比之前写入深度缓存片元的深度值大，那么这个片元是远离观察者的，它将不能通过深度测试而被认为被遮挡不可见。
frontFaceStencil，设定图元正面模板测试用的MTLStencilDescriptor。
backFaceStencil，设定图元背面模板测试用的MTLStencilDescriptor。如果图元的正面和背面的模板测试状态一样，那么可以设置同一个对象给frontFaceStencil和backFaceStencil。如果要显示地设置正面或是背面模板测试不可用，设置相应的属性为nil，nil也是默认值。

显示地禁用模板状态不是必要的。Metal基于模板测试是否是一个“无操作”来打开或是关闭模板测试。

代码5-12示例了如何创建并使用MTLDepthStencilDescriptor对象来配置MTLDepthStencilState对象，最终用于encoder对象renderEnc。在这个例子中，图元的正面模板测试状态由dsDesc的frontFaceStencil属性指定，图元的背面模板测试被显示关闭。

MTLStencilDescriptor的如下属性定义模板测试：

stencilCompareFunction，定义片元的模板测试如何实施。
readMask，是一个“与”操作位掩码，它即用于模板参考值，又用于模板存储值。模板测试一个比较操作，比较参考值的与运算结果 和 存储值的与运算结果。
writeMask，是一个位掩码用来计算那些模板值在模板操作中写入模板attachment。
stencilFailureOperation，指定如果模板测试失败，存储于模板attachment中的模板值如何处理。
depthFailureOperation，指定如果模板测试通过，但是深度测试失败，存储于模板attachment中的模板值如何处理。
depthStencilPassOperation，指定如果模板测试和深度测试都通过，存储于模板attachment中的模板值如何处理。

Listing 5-12 Creating and Using a Depth/Stencil Descriptor

```
MTLDepthStencilDescriptor *dsDesc = [[MTLDepthStencilDescriptor alloc] init];
if (dsDesc == nil)
     exit(1);   //  if the descriptor could not be allocated
dsDesc.depthCompareFunction = MTLCompareFunctionLess;
dsDesc.depthWriteEnabled = YES;
dsDesc.frontFaceStencil.stencilCompareFunction = MTLCompareFunctionEqual;
dsDesc.frontFaceStencil.stencilFailureOperation = MTLStencilOperationKeep;
dsDesc.frontFaceStencil.depthFailureOperation = MTLStencilOperationIncrementClamp;
dsDesc.frontFaceStencil.depthStencilPassOperation =
                           MTLStencilOperationIncrementClamp;
dsDesc.frontFaceStencil.readMask = 0x1;
dsDesc.frontFaceStencil.writeMask = 0x1;
dsDesc.backFaceStencil = nil;
id <MTLDepthStencilState> dsState = [device
                           newDepthStencilStateWithDescriptor:dsDesc];
[renderEnc setDepthStencilState:dsState];
[renderEnc setStencilReferenceValue:0xFF];
```

In Listing 5-12 (page 50), the stencil comparison function is `MTLCompareFunctionEqual`, so the stencil test passes if the masked reference value is equal to masked stencil value already stored at the location of a fragment. (A value is *masked* if it is bitwise ANDed with the `readMask`, which is 0x1 in this example.) The `stencilFailureOperation`, `depthFailureOperation`, and `depthStencilPassOperation` properties specify what to do to the stored stencil value for different test outcomes. In this example, the stencil value is unchanged (`MTLStencilOperationKeep`) if the stencil test fails, but it is incremented if the stencil test passes, unless the stencil value is already the maximum possible (`MTLStencilOperationIncrementClamp`).

在代码5-12中，模板测试函数是MTLCompareFunctionEqual，那么当一个片元位运算过后的参考值和位运算过后存储值相同时，模板测试通过（在本例中，模板值将和readMask设定的数字0x1做按位与操作）。在本例中，如果模板测试失败，存储的模板值不变，因为stencilFailureOperation设置为MTLStencilOperationKeep。但是如果模板测试成功，存储的模板值增大，一直到最大可能值，因为depthFailureOperation和depthStencilPassOperation设置为MTLStencilOperationIncrementClamp。

Drawing Geometric Primitives
绘制几何图元

After you have established the pipeline state and fixed-function state, you can call the following `MTLRenderCommandEncoder` methods to draw the geometric primitives. These draw methods reference resources (such as buffers that contain vertex coordinates, texture coordinates, surface normals, and other data) to execute the pipeline with the shader functions and other state you have previously established with `MTLRenderCommandEncoder`:

`drawPrimitives:vertexStart:vertexCount:instanceCount:` renders a number of instances (`instanceCount`) of primitives using vertex data in contiguous array elements, starting with the first vertex at the array element at the index `vertexStart` and ending at the array element at the index `vertexStart + vertexCount - 1`.

`drawPrimitives:vertexStart:vertexCount:` is the same as the previous method with an `instanceCount` of 1.

`drawIndexedPrimitives:indexCount:indexType:indexBuffer:indexBufferOffset:instanceCount:` renders a number of instances (`instanceCount`) of primitives using an index list specified in the `MTLBuffer` object `indexBuffer`. `indexCount` determines the number of indices. The index list starts at the index that is `indexBufferOffset` byte offset within the data in `indexBuffer`. `indexBufferOffset` must be a multiple of the size of an index, which is determined by `indexType`.

`drawIndexedPrimitives:indexCount:indexType:indexBuffer:indexBufferOffset:` is similar to the previous method with an `instanceCount` of 1.

当你已经创建了绘制管线state和固定管线state，你可以调用如下MTLRenderCommandEncoder的方法绘制几何图元：

drawPrimitives:vertexStart:vertexCount:instanceCount:，将绘制由instanceCount指定数量的图元实例，将使用连续存放的数组元素中的顶点数据，vertexStart指示数组的一个下标位置，这里是第一顶点数据开始的地方，vertexStart + vertexCount - 1 指示的下标是顶点数据结束的位置。
drawPrimitives:vertexStart:vertexCount:，和前一个方法相同，只是instanceCount被指定为1。

drawIndexedPrimitives:indexCount:indexType:indexBuffer:indexBufferOffset:instanceCount:，将绘制由instanceCount指定数量的图元实例。将使用indexBuffer参数指定的MTLBuffer对象中的索引列表，参数indexCount决定了索引的数量。indexBufferOffset参数指定了在indexBuffer中偏移的字节数，偏移后就是索引列表开始的地方。indexBufferOffset参数是索引长度的乘积，索引的类型由indexType决定。
drawIndexedPrimitives:indexCount:indexType:indexBuffer:indexBufferOffset:和前一个方法相同，只是instanceCount被指定为1。

For every primitive rendering method just listed, the first input value determines the primitive type with one of the `MTLPrimitiveType` values. The other input values determine which vertices are used to assemble the primitives. For all these methods, the `instanceStart` input value determines the first instance to draw, and `instanceCount` input value determines how many instances to draw.

As previously discussed, `setTriangleFillMode:` determines if the triangles are rendered as filled or wireframe. Triangles may also be culled, depending upon the `setCullMode:` and `setFrontFacingWinding:` settings. For more information, see Fixed-Function State Operations (page 48)).

When rendering a `MTLPrimitiveTypePoint` primitive, the shader language code for the vertex function must provide the `[[point_size]]` attribute, or the point size is undefined. For details on all Metal shading language point attributes, see the *Metal Shading Language Guide* document.

上面列出的每一个图元绘制方法，第一个输入参数决定了图元的类型（一个`MTLPrimitiveType`类型的值），其他的参数决定那些顶点数据用于组装图元。这几个方法中，start参数决定绘制第一个图元实例，`instanceCount`参数决定绘制多少个图元实例。

之前讨论的，encoder的`setTriangleFillMode:`方法决定了三角形绘制的时候填充的还是线框的。绘制三角形时的剔除算法由`setCullMode:`和 `setFrontFacingWinding:` 设置决定。

当绘制`MTLPrimitiveTypePoint`类型的图元，顶点着色程序必须支持属性，否则点的大小是不确定的。更多的metal着色语言关于点属性的内容参见《metal着色语言指南》

Finish Encoding for the Render Command Encoder
Encoder对象结束编码

To terminate a rendering pass, call `endEncoding`. After ending the previous command encoder, you can create a new command encoder of any type to encode additional commands into the command buffer.

要结束一个绘制pass，调用encoder的`endEncoding`方法。当这个方法被调用后，你可以创建一个新的任意类型的encoder，来编码新的指令，塞到commad buffer中。

Code Example: Drawing a Triangle
代码示例：如何绘制一个三角形

Listing 5-13 (page 53) shows how you can write Metal code that describes a rendering target in a `MTLRenderPassDescriptor`, stores geometry data in `MTLBuffer` objects, configures rendering state in a `MTLRenderPipelineDescriptor`, and then calls `MTLRenderCommandEncoder` methods to draw a single triangle. (In Listing 5-13 (page 53), presume that the `device` variable contains a `MTLDevice`, and `currentTexture` contains a `MTLTexture` that is used for a color attachment.)

代码5-13展示如何编写一段Metal代码，如何在`MTLRenderPassDescriptor`中描述绘制目标，如何在`MTLBuffer`对象中存储几何数据，在`MTLRenderPipelineDescriptor`中配置绘制状态，然后调用`MTLRenderCommandEncoder`对象的方法绘制一个三角形。（前提如下，变量`device`是`MTLDevice`类型的，变量是`MTLTexture`类型的，用作颜色attachment）

First create a `MTLCommandQueue` and use it to create a `MTLCommandBuffer`.

Create a `MTLRenderPassDescriptor` that specifies a collection of attachments that serve as the destination for encoded rendering commands in the command buffer. In this example, only the first color attachment is setup and used. Then the `MTLRenderPassDescriptor` is used to create a new `MTLRenderCommandEncoder`.

Create two `MTLBuffer` objects, `posBuf` and `colBuf`, and call `newBufferWithBytes:length:options:` to copy vertex coordinate and vertex color data, `posData` and `colData`, respectively, into the buffer storage. Call the `setVertexBuffer:offset:atIndex:` method of `MTLRenderCommandEncoder` twice

to specify the coordinates and colors. The `atIndex` input value of the `setVertexBuffer:offset:atIndex:` method corresponds to the attribute `buffer(atIndex)` in the source code of the vertex function.

Create a `MTLRenderPipelineDescriptor` and establish the vertex and fragment functions in the pipeline descriptor. In this example, create a `MTLLibrary` with source code from `progSrc`, which is assumed to be a string that contains Metal shading language source code. Next call the `newFunctionWithName:` method of `MTLLibrary` to create the `MTLFunction vertFunc` that represents the function called `hello_vertex` and to create the `MTLFunction fragFunc` that represents the function called `hello_fragment`. Then set the `vertexFunction` and `fragmentFunction` properties of the `MTLRenderPipelineDescriptor` with these `MTLFunction` objects.

Create a `MTLRenderPipelineState` from the `MTLRenderPipelineDescriptor` by calling `newRenderPipelineStateWithDescriptor:error:` or a similar method of `MTLDevice`. Then call the `setRenderPipelineState:` method of `MTLRenderCommandEncoder` to use the `MTLRenderPipelineState` values for rendering.

Call the `drawPrimitives:vertexStart:vertexCount:` method of `MTLRenderCommandEncoder` to append commands to perform the rendering of a filled triangle (type `MTLPrimitiveTypeTriangle`). Call the `endEncoding` method to end encoding for this rendering pass. And call the `commit` method of `MTLCommandBuffer` to execute the commands on the device.

首先创建一个MTLCommandQueue对象，然后用它创建一个MTLCommandBuffer对象。

接下来创建一个MTLRenderPassDescriptor对象，它代表了一系列的attachment，attachment将作为command buffer中的被编码的绘制指令的最终产出目标。在这个例子中，只有第一个颜色attachment被设置和使用。然后MTLRenderPassDescriptor被用来创建一个新的MTLRenderCommandEncoder对象。

接着创建两个MTLBuffer对象，posBuf和colBuf，然后使用newBufferWithBytes:length:options:方法来拷贝顶点坐标posBuf和顶点颜色colBuf数据。两次调用MTLRenderCommandEncoder的setVertexBuffer:offset:atIndex方法设定坐标和颜色。该方法的输入参数atIndex对应顶点着色程序中的属性缓存索引(atIndex)

接着创建一个MTLRenderPipelineDescriptor对象，并且为它设定顶点着色程序和片元着色程序。在这个例子中我们使用progSrc中的源代码创建一个MTLLibrary对象，假设progSrc是一个字符串，包含有Metal着色语言源代码。然后调用MTLLibrary的newFunctionWithName:方法创建一个MTLFunction类型的变量vertFunc，它代表了着色程序hello_vertex，创建一个MTLFunction类型的变量fragFunc，代表着色程序hello_fragment。然后vertFunc和fragFunc被分别设置给MTLRenderPipelineDescriptor对象的vertexFunction属性和fragmentFunction属性。

接着以MTLRenderPipelineDescriptor为参数，调用MTLDevice的newRenderPipelineStateWithDescriptor:error方法创建一个MTLRenderPipelineState对象。调用Encoder的setRenderPipelineState方法使Encoder得到这个state对象，以便在绘制时使用。

接着调用encoder的drawPrimitives:vertexStart:vertexCount方法，把绘制一个填充的三角形的指令推入command buffer。调用endEncoding方法来结束这个绘制pass的编码。最后调用MTLCommandBuffer的commit方法开始在设备上执行绘制指令。

Listing 5-13 Metal Code for Drawing a Triangle

```
id <MTLCommandQueue> commandQueue = [device newCommandQueue];
id <MTLCommandBuffer> commandBuffer = [commandQueue commandBuffer];
MTLRenderPassDescriptor *renderPassDesc
                                  = [MTLRenderPassDescriptor renderPassDescriptor];
renderPassDesc.colorAttachments[0].texture = currentTexture;
renderPassDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
renderPassDesc.colorAttachments[0].clearColor = MTLClearColorMake(0.0,1.0,1.0,1.0); id
<MTLRenderCommandEncoder> renderEncoder =
             [commandBuffer renderCommandEncoderWithDescriptor:renderPassDesc];
static const float posData[] = {
```

```
            0.0f, 0.33f, 0.0f, 1.f,
            -0.33f, -0.33f, 0.0f, 1.f,
            0.33f, -0.33f, 0.0f, 1.f,
};
static const float colData[] = {
            1.f, 0.f, 0.f, 1.f,
            0.f, 1.f, 0.f, 1.f,
            0.f, 0.f, 1.f, 1.f,
};
id <MTLBuffer> posBuf = [device newBufferWithBytes:posData
            length:sizeof(posData) options:nil];
id <MTLBuffer> colBuf = [device newBufferWithBytes:colorData length:sizeof(colData) options:nil];
[renderEncoder setVertexBuffer:posBuf offset:0 atIndex:0];
[renderEncoder setVertexBuffer:colBuf offset:0 atIndex:1];
NSError *errors;
id <MTLLibrary> library = [device newLibraryWithSource:progSrc options:nil
                                error:&errors];
id <MTLFunction> vertFunc = [library newFunctionWithName:@"hello_vertex"
                                 options:nil error:&errors];
id <MTLFunction> fragFunc = [library newFunctionWithName:@"hello_fragment"
                                 options:nil error:&errors];
MTLRenderPipelineDescriptor *renderPipelineDesc
                                        = [[MTLRenderPipelineDescriptor alloc] init];
renderPipelineDesc.vertexFunction = vertFunc;
renderPipelineDesc.fragmentFunction = fragFunc;
renderPipelineDesc.colorAttachments[0].pixelFormat = currentTexture.pixelFormat;
id <MTLRenderPipelineState> pipeline = [device
newRenderPipelineStateWithDescriptor:renderPipelineDesc error:&errors];

[renderEncoder setRenderPipelineState:pipeline];
[renderEncoder drawPrimitives:MTLPrimitiveTypeTriangle
                   vertexStart:0 vertexCount:3];
[renderEncoder endEncoding];
[commandBuffer commit];
```

In Listing 5-13 (page 53), a `MTLFunction` object represents the shader function called `hello_vertex`. The `setVertexBuffer:offset:atIndex:` method of `MTLRenderCommandEncoder` is used to specify the vertex resources (in this case, two buffer objects) that are passed as arguments into `hello_vertex`. The `atIndex` input value of the `setVertexBuffer:offset:atIndex:` method corresponds to the attribute `buffer(atIndex)` in the source code of the vertex function, as shown in Listing 5-14 (page 54).

代码5-13中，一个`MTLFunction`对象表示一个了叫做`hello_vertex`的着色程序。encoder的`setVertexBuffer:offset:atIndex`方法被用来设定顶点数据资源（在此例中是两个缓存对象），这些资源作为输入参数传递到着色程序`hello_vertex`中。`setVertexBuffer:offset:atIndex`方法中的输入参数`atIndex`，对应着顶点着色程序源代码中的缓存属性（`atIndex`），着色程序如代码5-14所示

Listing 5-14 Corresponding Shader Language Function Declaration

```
vertex VertexOutput hello_vertex(
                    const global float4 *pos_data [[ buffer(0) ]],
{
...
}

const global float4 *color_data [[ buffer(1) ]])
```

Encoding a Single Rendering Pass Using Multiple Threads
使用多线程编码一个绘制pass

In some cases, you need to encode so many commands for a single rendering pass that using multiple threads to encode rendering commands can improve performance substantially. If the standard `MTLRenderCommandEncoder` is used, each rendering pass requires its own intermediate attachment store and load actions to preserve the render target contents, which can adversely impact performance.

A better way to accomplish this is to create a `MTLParallelRenderCommandEncoder` object by calling the `parallelRenderCommandEncoderWithDescriptor:` method of `MTLCommandBuffer`. You can call the `renderCommandEncoder` method of `MTLParallelRenderCommandEncoder` several times to create multiple

subordinate `MTLRenderCommandEncoder` objects, all sharing the same `MTLCommandBuffer` and `MTLRenderPassDescriptor`. The `MTLParallelRenderCommandEncoder` ensures the attachment load and store actions only occur at the start and end of the entire rendering pass, not at the start and end of each `MTLRenderCommandEncoder`. Each `MTLRenderCommandEncoder` can be assigned to its own thread in parallel in a safe and highly performant manner.

在某些情况下，你需要为一个绘制pass编码非常多的绘制命令，使用多线程来编码可以大幅提高效率。如果使用标准的`MTLRenderCommandEncoder`对象，每一个绘制pass都需要执行对中间attachment的存储以及加载操作，才能保存绘制结果内容，这都对提高性能不利。

一个更好的方法是通过`MTLCommandBuffer`的`parallelRenderCommandEncoderWithDescriptor`方法创建一个`MTLParallelRenderCommandEncoder`对象。再通过多次调用这个`Parallel Encoder`的`renderCommandEncoder`方法创建多个子`MTLRenderCommandEncoder`对象，这些子Encoder对象都共享同一个`MTLCommandBuffer`对象和`MTLRenderPassDescriptor`对象。`Parallel Encoder`使得attachment的加载和存储操作只发生整个绘制pass的开头和结尾。每个子`Encoder`对象可以被赋予它自己的线程，如此encoder可以安全高效的并行起来。

All subordinate `MTLRenderCommandEncoder` objects created from the same `MTLParallelRenderCommandEncoder` encoded commands to the same command buffer. Commands are encoded to a command buffer in the order in which the render command encoders are created. To end encoding for a specific render command encoder, call the `endEncoding` method of `MTLRenderCommandEncoder`. After you have ended all render command encoders created by `MTLParallelRenderCommandEncoder`, call the `endEncoding` method of `MTLParallelRenderCommandEncoder` to end the rendering pass.

Listing 5-15 (page 55) shows the `MTLParallelRenderCommandEncoder` creating three `MTLRenderCommandEncoder` objects `rCE1`, `rCE2`, and `rCE3`.

所有的由`Parallel Encoder`创建的子`Encoder`对象都为同一个command buffer编码绘制指令。绘制指令被编码进入command buffer的顺序和encoder被创建的顺序相同。要结束一个特定的子encoder，调用`Encoder`的`endEncoding`方法。如果由`Parallel Encoder`创建的所有的子encoder都结束了，调用`Parallel Encoder`的`endEncoding`方法结束一个绘制pass。

代码5-15示例了一个Parallel Encoder创建3个子Encoder对象，`rCE1`, `rCE2`, 和 `rCE3`

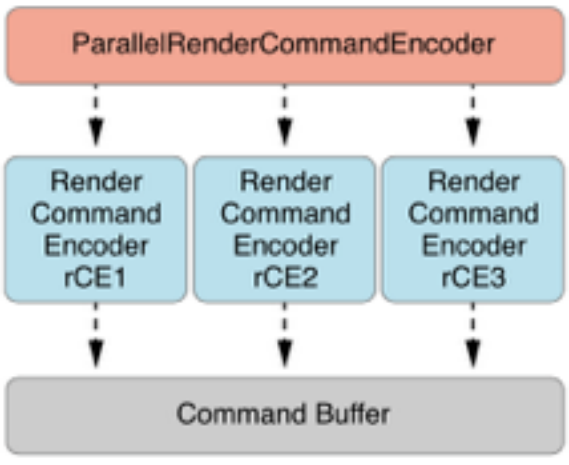
Listing 5-15 A Parallel Rendering Encoder with Three Render Command Encoders

```
MTLRenderPassDescriptor *renderPassDesc
                        = [MTLRenderPassDescriptor renderPassDescriptor];
renderPassDesc.colorAttachments[0].texture = currentTexture;
renderPassDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
renderPassDesc.colorAttachments[0].clearColor = MTLClearColorMake(0.0,0.0,0.0,1.0);
id <MTLParallelRenderCommandEncoder> parallelRCE = [commandBuffer
                        parallelRenderCommandEncoderWithDescriptor:renderPassDesc];
id <MTLRenderCommandEncoder> rCE1 = [parallelRCE renderCommandEncoder];
id <MTLRenderCommandEncoder> rCE2 = [parallelRCE renderCommandEncoder];
id <MTLRenderCommandEncoder> rCE3 = [parallelRCE renderCommandEncoder];
//  not shown: rCE1, rCE2, and rCE3 call methods to encode graphics commands
//
//  rCE1 commands are processed first, because it was created first
//  even though rCE2 and rCE3 end earlier than rCE1
[rCE2 endEncoding];
[rCE3 endEncoding];
[rCE1 endEncoding];
//  all MTLRenderCommandEncoders must end before MTLParallelRenderCommandEncoder
[parallelRCE endEncoding];
```

The order in which the command encoders call `endEncoding` is not relevant to the order in which commands are encoded and appended to the `MTLCommandBuffer`. For `MTLParallelRenderCommandEncoder`, the `MTLCommandBuffer` always contains commands in the order the subordinate render command encoders were created, as seen in Figure 5-6 (page 56).

各子encoder调用`endEncoding`方法的顺序和encoder中被编码的绘制指令被推入`MTLCommandBuffer`中执行的顺序不相关。一个Parallel encoder对象包含着很多绘制指令（在其子Encoder中），这些指令的执行顺序和子encoder被创建顺序相同，如同图5-6所示。

Figure 5-6 Ordering of Render Command Encoders in a Parallel Rendering Pass



Data-Parallel Compute Processing: Compute Command Encoder
并行数据计算处理过程：并行计算指令Encoder

This chapter explains how to create and use `MTLComputeCommandEncoder`, which encodes data-parallel compute processing state and commands that can be executed on the device. To create a `MTLComputeCommandEncoder`, call the `computeCommandEncoder` method of `MTLCommandBuffer`. To perform a data-parallel computation, you:

use a `MTLDevice` method to create a compute state, `MTLComputePipelineState`, that contains compiled code from a `MTLFunction` object, as discussed in Creating a Compute State (page 57). The `MTLFunction` object represents a compute function written with the Metal shading language, as described in Functions and Libraries (page 26).

specify the `MTLComputePipelineState` for the `MTLComputeCommandEncoder`. At any given moment, a `MTLComputeCommandEncoder` can be associated to only one compute function.

specify resources and related objects (`MTLBuffer`, `MTLTexture`, and possibly `MTLSamplerState`) that may contain the data to be processed and returned by the compute state, as discussed in Specify a Compute State and Resources for a Compute Command Encoder (page 58). Also set their argument table indices, so that Metal framework code can locate a corresponding resource in the shader code. At any given moment, the `MTLComputeCommandEncoder` can be associated to a number of resource objects.

dispatch the compute function a specified number of times, as explained in Compute Command Execution (page 59).

本章展示如何创建并使用`MTLComputeCommandEncoder`，它把并行数据计算状态和指令打包编码，最后在设备上执行。 调用`MTLCommandBuffer`的`computeCommandEncoder`方法可以创建一个这种类型的Encoder。进行并行数据计算的步骤如下：

使用`MTLDevice`的方法创建一个`MTLComputePipelineState`类型对象，state对象包含编译好的着色程序代码，这是由`MTLFunction`对象装载的。`MTLFunction`表示一个用Metal着色语言编写的着色程序。

为`MTLComputeCommandEncoder`设定`MTLComputePipelineState`对象，任一时刻，一个并行计算Encoder只能被设定使用一个计算着色程序。

为state对象设定资源（`MTLBuffer`,`MTLTexture`, `MTLSamplerState`），这些资源中包括了待处理数据或是被用于返回数据。同时还要设定这些资源的参数索引表，如此Metal的框架代码才能为着色器代码定位相关资源。在任一时刻，并行计算Encoder可以关联多个资源对象。

按指定次数分发计算程序（构建线程组）。

Creating a Compute Pipeline State
创建一个并行计算state

A `MTLFunction` object represents data-parallel code that can be executed by a `MTLComputePipelineState` object. The `MTLComputeCommandEncoder` encodes commands that set arguments and execute the compute function. Reflection data that reveals details of the compute function and its arguments is optionally created in `MTLComputePipelineReflection` along with the `MTLComputePipelineState`. Avoid obtaining reflection data if it will not be used. For more information on how to analyze reflection data, see Determining Function Details at Runtime (page 28). Creating a `MTLComputePipelineState` may be an expensive operation that involves the possible compilation of the compute function, so you can use either a blocking or asynchronous method. To create a `MTLComputePipelineState`, call one of the following `MTLDevice` methods with a compute function object:

To synchronously compile the compute function and create the compute pipeline state object, call either the `newComputePipelineStateWithFunction:error:` or `newComputePipelineStateWithFunction:options:reflection:error:` method of `MTLDevice`. The latter method also returns reflection data of the compute function.

To asynchronously compile the compute function and register a handler to be called when the compute pipeline state object is created, call either the `newComputePipelineStateWithFunction:completionHandler:` or `newComputePipelineStateWithFunction:options:completionHandler:` method of `MTLDevice`. The latter method also returns reflection data of the compute function.

一个MTLFunction对象可以代表一段并行计算代码，它可以被一个MTLComputePipelineState对象执行。MTLComputeCommandEncoder用于编码计算指令、设置入参、执行计算程序。reflection数据（MTLComputePipelineReflection）用于描述并行计算程序和它的入参的细节，它的创建伴随MTLComputePipelineState对象的创建，是可选的。如果不适用reflection数据，则不要创建。前面已有章节对如何分析reflection数据有详细描述。
创建MTLComputePipelineState对象会是一种高开销的操作，因为它可能包含对并行计算着色程序的编译过程，所以还可以选择使用异步加代码块方式创建。调用下列MTLDevice方法中以MTLFunction对象为参数创建MTLComputePipelineState对象。

同步阻塞编译计算着色程序来创建并行计算state对象可以调用下面两个方法中的一个：
newComputePipelineStateWithFunction:error:
newComputePipelineStateWithFunction:options:reflection:error:
这两个都是MTLDevice方法，后面这个方法返回计算着色程序参数的reflection数据

异步编译计算着色程序，以注册程序块方式完成并行计算state对象的创建，可以调用下面两个方法中的一个：
newComputePipelineStateWithFunction:completionHandler:
newComputePipelineStateWithFunction:options:completionHandler:
和同步方法一样，这两个都是MTLDevice方法，后面这个方法返回计算着色程序参数的reflection数据

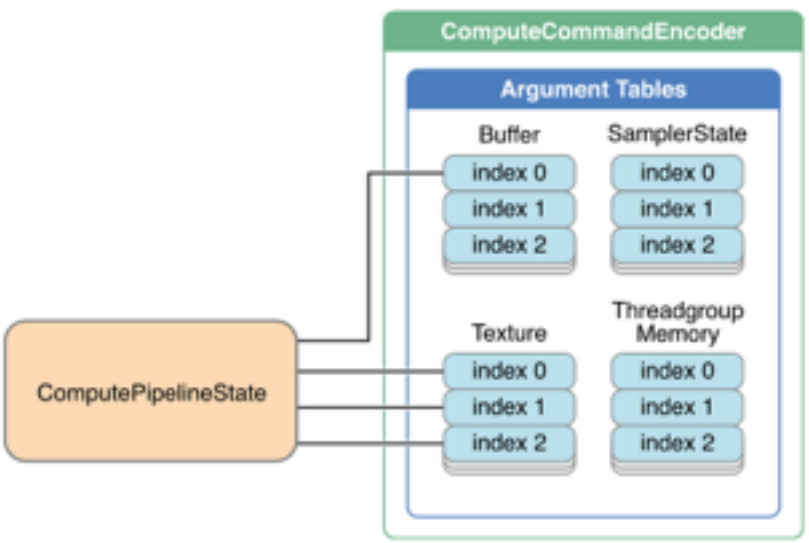
Specify a Compute State and Resources for a Compute Command Encoder
为并行计算Encoder指定state和资源对象

The `setComputePipelineState:` method specifies the compute state to use. The following `MTLComputeCommandEncoder` methods specify a resource (i.e., a buffer, texture, sampler state, or threadgroup memory) that is used as an argument to the compute function represented by the `MTLComputePipelineState`. Each method assigns one or more resources to the corresponding argument(s), as illustrated in Figure 6-1 (page 59).

setComputePipelineState:，该MTLComputeCommandEncoder方法用于设定Encoder使用的state对象。
下列的MTLComputeCommandEncoder方法用于设定资源对象，这些资源对象（缓存、纹理、采样器state、线程组内存）是包含于state中的并行计算着色程序的入参使用的。
每个方法都设置一个或多个资源给对应的并行计算着色程序入参，如图6-1所示：

```
setBuffer:offset:atIndex:
setBuffers:offsets:withRange:
setTexture:atIndex:
setTextures:withRange:
setSamplerState:atIndex:
setSamplerState:lodMinClamp:lodMaxClamp:atIndex:
setSamplerStates:withRange:
setSamplerStates:lodMinClamps:lodMaxClamps:withRange:
setThreadgroupMemoryLength:atIndex:
```

Figure 6-1 Argument Tables for the Compute Command Encoder



There are a maximum of 31 entries in the buffer argument table, 31 entries in the texture argument table, and 16 entries in the sampler state argument table.
The total of all threadgroup memory allocations must not exceed 16K bytes; otherwise, an error occurs.

如上图所示，Encoder对象中，缓存参数索引表最多存放31个元素，纹理参数索引表最多存放31个元素，采样器state参数索引表最多存放16个元素。线程组内存总量不能超过16K字节，否则将导致错误。

Compute Command Execution
并行计算指令的执行

To encode a command to execute a compute function, call the `dispatchThreadgroups:threadsPerThreadgroup:` method of `MTLComputeCommandEncoder` and specify the threadgroup dimensions and the number of threadgroups. You can query the `threadExecutionWidth` and `maxTotalThreadsPerThreadgroup` properties of `MTLComputePipelineState` to optimize the execution of the compute function on this device. For most efficient execution of the compute function, the total number of threads specified by the `threadsPerThreadgroup` argument to the `dispatchThreadgroups:threadsPerThreadgroup:` method should be a multiple of `threadExecutionWidth`. The total number of threads in a threadgroup is the product of the components of `threadsPerThreadgroup`: `threadsPerThreadgroup.width * threadsPerThreadgroup.height * threadsPerThreadgroup.depth`. The `maxTotalThreadsPerThreadgroup` property specifies the maximum number of threads that can be in a single threadgroup to execute this compute function on the device.

编码指令，执行并行计算着色程序，可以调用MTLComputeCommandEncoder的dispatchThreadgroups:threadsPerThreadgroup:方法，这时需要设定线程组规模（threadsPerThreadgroup有width、height、depth三个分量）和有几个线程组。
可以通过查询MTLComputePipelineState对象的threadExecutionWidth和maxTotalThreadsPerThreadgroup属性来优化在设备上执行的并行计算着色程序。
要使并行计算着色程序的执行最为高效，通过dispatchThreadgroups:threadsPerThreadgroup:方法指定的threadsPerThreadgroup参数应该是threadExecutionWidth的整数倍。
一个线程组内的总线程数是入参threadsPerThreadgroup各分量的乘积，也就是threadsPerThreadgroup.width*threadsPerThreadgroup.height*threadsPerThreadgroup.depth。
maxTotalThreadsPerThreadgroup属性表示并行计算着色程序在设备上执行时，在一个线程组中允许的最大得线程数量。
Compute commands are executed in the order in which they were encoded into the command buffer. A compute command has finished execution when all of the threadgroups associated with the command have finished execution and all of the results are written to memory. This means that the results of a compute command are available to commands encoded after it in the command buffer.

To end encoding commands for a compute command encoder, call the `endEncoding` method of `MTLComputeCommandEncoder`. After ending the previous command encoder, you can create a new command encoder of any type to encode additional commands into the command buffer.

并行计算程序按照Encoder被推入command buffer的次序执行。一个并行计算着色程序所拥有的所有线程组都执行完毕且计算结果都写入内存后，才表示这个并行计算着色程序执行完毕。这意味着前一个Encoder生产的数据可以被下一个Encoder使用

要结束Encoder编码，调用MTLComputeCommandEncoder的endEncoding方法。对于同一个command buffer来说，在前一个Encoder结束编码后，才可以创建一个新的任意类型的Encoder来编码新的指令并推送之。

Code Template for Data-Parallel Functions
数据并行计算着色程序的代码模板

Listing 6-1 (page 60) shows an example that creates and uses a `MTLComputeCommandEncoder` object to perform the parallel computations of an image transformation on specified data. (This example does not show how the device, library, command queue, and resource objects are created and initialized.) The example creates a command buffer and then uses it to create the `MTLComputeCommandEncoder`. Next a `MTLFunction` is created that represents the entry point

`filter_main` from the `MTLLibrary`, shown in Listing 6-2 (page 61). Then the function object is used to create a `MTLComputePipelineState` called `filterState`.

代码6-1展示了一个例子,创建并使用一个MTLComputeCommandEncoder对象对一个图像变换并行计算（这个例子没有展示device, library, command queue, 资源对象如何创建并初始化）。在例子中，先创建了一个command buffer，用它创建一个MTLComputeCommandEncoder，然后一个MTLFunction对象被创建出来表示来自于MTLLibrary中的着色程序filter_main的入口（着色程序在代码6-2展示），接着MTLFunction对象被用来创建MTLComputePipelineState对象，名叫 filterState。

The compute function performs an image transformation and filtering operation on the image `inputImage` with the results returned in `outputImage`. First the `setTexture:atIndex:` and `setBuffer:offset:atIndex:` methods assign texture and buffer objects to indices in the specified argument tables. `paramsBuffer` specifies values used to perform the image transformation, and `inputTableData` specifies filter weights. The compute function is executed as a 2D threadgroup of size 16 x 16 pixels in each dimension. The `dispatchThreadgroups:threadsPerThreadgroup:` method enqueues the command to dispatch the threads executing the compute function, and the `endEncoding` method terminates the `MTLComputeCommandEncoder`. Finally the `commit` method of `MTLCommandBuffer` causes the commands to be executed as soon as possible.

代码6-1 6-2示例的并行计算着色程序是要对一个图像进行变换和过滤操作，输入是inputImage，输出是outputImage。首先调用setTexture:atIndex:和setBuffer:offset:atIndex:，为参数索引表中的索引项设置了纹理和缓存对象。变量paramsBuffer指定了用于实施图像变换的值，inputTableData指定了过滤权重。并行计算着色程序的线程组被设定为2维，16*16。dispatchThreadgroups:threadsPerThreadgroup:方法将并行计算程序指令排入队列分派给线程待执行，然后endEncoding方法结束Encoder编码过程。最后MTLCommandBuffer的commit方法被调用，使得计算指令被尽快执行。

Listing 6-1 Specifying and Running a Function in a Compute State

```
  id <MTLDevice> device;
  id <MTLLibrary> library;
  id <MTLCommandQueue> commandQueue;
  id <MTLTexture> inputImage;
  id <MTLTexture> outputImage;
  id <MTLTexture> inputTableData;
  id <MTLBuffer> paramsBuffer;
  // ... Create and initialize device, library, queue, resources
  // Obtain a new command buffer
  id <MTLCommandBuffer> commandBuffer = [commandQueue commandBuffer];

  // Create a compute command encoder
  id <MTLComputeCommandEncoder> computeCE = [commandBuffer computeCommandEncoder];
  NSError *errors;
  id <MTLFunction> func = [library newFunctionWithName:@"filter_main" error:&errors];
  id <MTLComputePipelineState> filterState = [device newComputePipelineStateWithFunction:func
error:&errors];
  [computeCE setComputePipelineState:filterState];
  [computeCE setTexture:inputImage atIndex:0];
  [computeCE setTexture:outputImage atIndex:1];
  [computeCE setTexture:inputTableData atIndex:2];
  [computeCE setBuffer:paramsBuffer offset:0 atIndex:0];
  MTLSize threadsPerGroup = {16, 16, 1};
  MTLSize numThreadgroups = {inputImage.width/threadsPerGroup.width,
                             inputImage.height/threadsPerGroup.height, 1};
  [computeCE dispatchThreadgroups:numThreadgroups
                            threadsPerThreadgroup:threadsPerGroup];
  [computeCE endEncoding];
  // Commit the command buffer
  [commandBuffer commit];
```

In Listing 6-2 (page 61), the code for the user-defined functions `read_and_transform` and `filter_table` is not shown.

代码6-2示例了用户定义的着色程序，它使用到得两个函数read_and_transform和filter_table没有展示。

Listing 6-2 Shading Language Compute Function Declaration

```
kernel void filter_main(
  texture2d<float,access::read>   inputImage   [[ texture(0) ]],
```

```
  texture2d<float,access::write>  outputImage  [[ texture(1) ]],
  uint2 gid
  texture2d<float,access::sample> table
  constant Parameters* params
)
[[ thread_position_in_grid ]],
[[ texture(2) ]],
[[ buffer(0) ]]
{
  float2 p0          = static_cast<float2>(gid);
  float3x3 transform = params->transform;
  float4   dims      = params->dims;
  float4 v0 = read_and_transform(inputImage, p0, transform);
  float4 v1 = filter_table(v0,table, dims);
  outputImage.write(v1,gid);
}
```

Buffer and Texture Operations: Blit Command Encoder
缓存和纹理操作：位图指令Encoder

`MTLBlitCommandEncoder` provides methods for copying data between resources (buffers and textures). Data copying operations may be necessary for image processing and texture effects, such as blurring or reflections. They may be used to access image data that is rendered off-screen.

To perform these operations, create a `MTLBlitCommandEncoder` object by calling the `blitCommandEncoder` method of `MTLCommandBuffer`. Then call the `MTLBlitCommandEncoder` methods described below to encode commands onto the command buffer.

`MTLBlitCommandEncoder`提供方法用于在资源对象间（缓存和纹理）拷贝数据。数据拷贝操作对于图像处理或是纹理特效（比如模糊和反射）来说是必要。它可以访问离屏渲染的图像数据。

为了实施这些操作，调用`MTLCommandBuffer`的`blitCommandEncoder`方法创建一个`MTLBlitCommandEncoder`对象。然后调用这个Encoder的方法来编码指令推送给command buffer。

Copying Data in GPU Memory Between Resource Objects
在GPU内存中资源对象间拷贝数据

The following `MTLBlitCommandEncoder` methods copy image data between resource objects: between two buffer objects, between two texture objects, or between a buffer and a texture.

下面的`MTLBlitCommandEncoder`方法在资源对象间拷贝图像数据：在两个缓存对象间，在两个纹理对象间，在一个缓存和一个纹理间。

Copying Data Between Two Buffers
在两个缓存对象间拷贝数据

`copyFromBuffer:sourceOffset:toBuffer:destinationOffset:size:` copies data between two buffers: from the source buffer into the destination buffer `toBuffer`. If the source and destination are the same buffer, and the range being copied overlaps, the results are undefined.

`copyFromBuffer:sourceOffset:toBuffer:destinationOffset:size:`，该方法在两个缓存对象间拷贝数据，从源缓存对象到目标缓存对象`toBuffer`。如果源和对象是同一个缓存对象，并且拷贝区域的制定重合，结果是未定义的。

Copying Data From a Buffer to a Texture
从缓存对象拷贝数据到纹理对象

`copyFromBuffer:sourceOffset:sourceBytesPerRow:sourceBytesPerImage:sourceSize:toTexture:destinationSlice:destinationLevel:destinationOrigin:` copies image data from a source buffer into the destination texture `toTexture`.

`copyFromBuffer:sourceOffset:sourceBytesPerRow:sourceBytesPerImage:sourceSize:toTexture:destinationSlice:destinationLevel:destinationOrigin:`，该方法从源缓存对象拷贝图像数据到目标纹理对象`toTexture`。

Copying Data Between Two Textures
在两个纹理对象间拷贝数据

`copyFromTexture:sourceSlice:sourceLevel:sourceOrigin:sourceSize:toTexture:destinationSlice:destinationLevel:destinationOrigin:`

copies a region of image data between two textures: from a single cube slice and mipmap level of the source texture to the destination texture `toTexture`.

copyFromTexture:sourceSlice:sourceLevel:sourceOrigin:sourceSize:toTexture:destinationSlice:destinationLevel:destinationOrigin:，该方法在两个纹理将拷贝某个区域的图像数据：从作为源的纹理对象的一个单独的立方切片和mipmap层次向目标纹理对象toTexture做拷贝。

Copying Data From a Texture to a Buffer
从一个纹理对象向一个缓存对象拷贝数据

`copyFromTexture:sourceSlice:sourceLevel:sourceOrigin:sourceSize:toBuffer:destinationOffset:destinationBytesPerRow:destinationBytesPerImage:`

copies a region of image data from a single cube slice and mipmap level of a source texture into the destination buffer `toBuffer`.

copyFromTexture:sourceSlice:sourceLevel:sourceOrigin:sourceSize:toBuffer:destinationOffset:destinationBytesPerRow:destinationBytesPerImage:，该方法从作为源的纹理对象的一个单独的立方切片和mipmap层次的某个区域向目标缓存对象toBuffer拷贝数据。

Generating Mipmaps
生成mipmaps

The `generateMipmapsForTexture:` method of `MTLBlitCommandEncoder` automatically generate mipmaps for the given texture, starting from the base level texture image. `generateMipmapsForTexture:` creates scaled images for all mipmap levels up to the maximum level.

For details on how the number of mipmaps and the size of each mipmap are determined, see Slices (page 21).

MTLBlitCommandEncoder的generateMipmapsForTexture:方法将为给定纹理自动生成mipmaps（以纹理图像作为基层数据）。该方法创建mipmap各个层级的缩放过的图像直到最高层（1个像素）。在讲述纹理切片的章节有mipmap的层数和每层图像大小如何确定的详细内容。

Filling the Contents of a Buffer
为一个缓存填充内容

The `fillBuffer:range:value:` method of `MTLBlitCommandEncoder` stores the 8-bit constant `value` in every byte over the specified `range` of the given buffer.

MTLBlitCommandEncoder的fillBuffer:range:value:方法将为一个给定缓存对象的制定区域的每个字节填充一个8bit的常量值。

Finish Encoding for the Blit Command Encoder
位图指令Encoder结束编码

To end encoding commands for a blit command encoder, call `endEncoding`. After ending the previous command encoder, you can create a new command encoder of any type to encode additional commands into the command buffer.

一个位图指令Encoder完成指令编码，调用endEncoding方法。对于一个command buffer，当前一个Encoder完成后，才可以创建任意类型的新Encoder编码更多的指令到command buffer。

Metal Tips and Techniques
Metal诀窍和相关技术

This chapter discusses tips and techniques that may improve app performance or developer productivity.

本章描述Metal诀窍和相关技术用于提高app性能和开发者的效率。

Creating Libraries During the App Build Process
在app工程的编译打包阶段创建着色程序Liberies

Compiling shader language source files and building a library (`.metallib` file) during the app build process achieves better app performance than compiling shader source code at runtime. You can build a library within Xcode or by using command line utilities.

在app工程的编译打包阶段编译着色语言源文件并生成一个library（`.metallib`文件）相比 在运行时编译着色语言源代码，前者有更好的性能。生成着色程序 library可以使用Xcode集成环境或是命令行工具。

Using Xcode to Build a Library
使用Xcode生成着色程序library

Any shader source files that are in your project are automatically used to generate the default library, which you can access from Metal framework code with the `newDefaultLibrary` method of `MTLDevice`.

任何包含在Xcode工程中的着色语言源代码文件都自动被编译打包成默认的library，这个library是可以在Metal框架代码中用`MTLDevice`的`newDefaultLibrary`方法访问到。

Using Command Line Utilities to Build a Library
使用命令行生成着色程序library

Figure 8-1 (page 65) shows the command line utilities that form the compiler toolchain for shader language source code. To build a library file that you can access from the app, you can use:

1. **`metal`** to compile a **`.metal`** file into a **`.air`** file, which stores an intermediate representation of shader language code.
2. **`metal-ar`** to archive several **`.air`** files together into a single **`.metalar`** file. **`metal-ar`** is similar to the Unix utility **`ar`**.
3. **`metallib`** to build a Metal **`.metallib`** library file from the archive **`.metalar`** file.

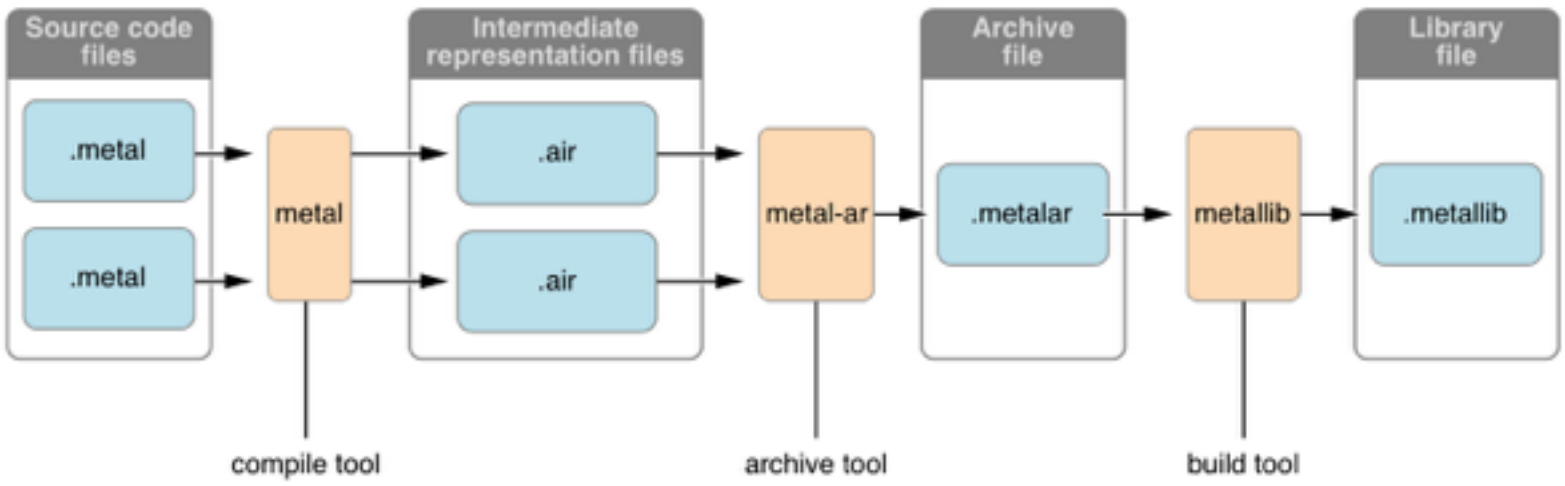
To access the resulting library in framework code, you can call the `newLibraryWithFile:error:` method of `MTLDevice`.

如图8-1所示，命令行工具集为着色语言源代码准备了一个编译工具链，可以用来生成一个library文件，如此就可以在app中访问到它。

1. **`metal`**，这是一个将**`.metal`**文件编译生成**`.air`**文件的命令行工具。**`.air`**文件保存了表示着色语言源代码的中间编译结果。
2. **`metal-ar`**，该命令行工具将多个**`.air`**文件打包成一个单独的**`.metalar`**文件。改命令行工具类似于Unix系统的工具**`ar`**。
3. **`metallib`**，该命令行工具从**`.metalar`**文件生成一个.metallib文件

在metal框架代码中访问命令行工具生成的.metallib文件，使用`MTLDevice`的`newLibraryWithFile:error:`方法。

Figure 8-1 Building a Library File with Command Line Utilities



Metal Feature Sets
Metal 属性集

A Metal feature set describes the capabilities and limitations of a Metal implementation. The `supportsFeatureSet:` method of `MTLDevice` returns a `BOOL` that indicates whether the capabilities and limitations of a particular feature set apply to this implementation.

Metal属性集描述Metal设备的能力和限制，`MTLDevice`的`supportsFeatureSet:`方法返回一个`BOOL`值表示某一项特定的Metal能力或是限制是否应用到当前Metal实现

There are two feature sets: `MTLFeatureSet_iOS_GPUFamily1_v1` and `MTLFeatureSet_iOS_GPUFamily2_v1`. Within an `iOS_GPUFamilyN`, a later version is indicated by the suffix `_vN` and is guaranteed to be a superset of an earlier version in the same feature family. Note that `MTLFeatureSet_iOS_GPUFamily1_v1` and `MTLFeatureSet_iOS_GPUFamily2_v1` are in different families, so they are neither supersets nor subsets of one another.

有两个属性集：`MTLFeatureSet_iOS_GPUFamily1_v1`和`MTLFeatureSet_iOS_GPUFamily2_v1`。对于`iOS_GPUFamilyN`来说，一个由后缀`_vN`指定的靠后的版本号保证是靠前版本号的超集（在同一个属性族内）。注意`MTLFeatureSet_iOS_GPUFamily1_v1`和`MTLFeatureSet_iOS_GPUFamily2_v1`处于两个不同的属性族，他们之间没有子集或是超集的关系。

The `MTLFeatureSet_iOS_GPUFamily1_v1` feature set has the following Metal capabilities and limitations:

- A `MTLRenderPassDescriptor` can have up to `4` color attachments, in addition to a depth attachment and a stencil attachment.
- Each render pass can store a maximum of `16` bytes of color data per sample across all of its color attachments. (Depth and stencil attachments do not count against this limit.) If you create a `MTLRenderPassDescriptor` and the sum of the storage requirements for all color attachments is greater than the maximum allowed, a fatal error occurs.
- ASTC pixel formats are **not** supported.
- All pixel formats consume a minimum of `4` bytes per sample in an attachment, even if the pixel formats use fewer than `4` bytes per pixel in memory. For example, the `MTLPixelFormatR8Unorm`, `MTLPixelFormatR8Uint`, and `MTLPixelFormatR8Sint` pixel formats use 1 byte per pixel in memory, but consume `4` bytes per sample in an attachment. The `MTLPixelFormatRGB10A2Unorm`, `MTLPixelFormatRG11B10Float`, and `MTLPixelFormatRGB9E5Float` pixel formats use `4` bytes per pixel in memory, but consume `8` bytes per sample in an attachment. All other pixel formats take the same amount of space in memory as in attachment storage.
- An attachment should not be smaller than `32` pixels in either width or height. An attachment (especially depth or stencil attachment) that is smaller than `32` pixels will have a performance cost.
- Every threadgroup memory allocation is rounded up to `16` bytes.
  The total of all threadgroup memory allocations must not exceed `16384` bytes.
- For a `MTLTextureDescriptor`, the `height` and `width` properties must not exceed `4096`. The `depth` property must not exceed `2048`.
- In the argument tables for the render and compute command encoders, there are a maximum of `31` entries in the buffer argument table, `31` entries in the texture argument table, and `16` entries in the sampler state argument table.

`MTLFeatureSet_iOS_GPUFamily1_v1`属性集有如下的Metal能力和限制：

- 一个MTLRenderPassDescriptor可以最多有4个颜色attachment，一个深度缓存，一个模板缓存。
- 每个绘制pass，其所有的颜色attachment的单像素存储空间之和最多为16个字节长（深度和模板attachment不算在内）。如果创建了一个MTLRenderPassDescriptor并且所有颜色attachment的存储需求的总和大于16字节，将导致错误发生。
- ASTC像素格式是不被支持的。
- 在一个attachment中，所有的像素格式，其单个像素都消耗最少4个字节空间，就算这种像素格式内存中每像素尺寸少于4个字节。举例来说，MTLPixelFormatR8Unorm、MTLPixelFormatR8Uint、MTLPixelFormatR8Sint这三种像素格式在内存中每个像素占用1字节，但是在attachment中，每个像素消耗4字节。MTLPixelFormatRGB10A2Unorm、MTLPixelFormatRG11B10Float、MTLPixelFormatRGB9E5Float这三种像素格式在内存中每个像素占用4字节，但是在attachment中，每个像素消耗8字节。其他的像素格式在内存中的占用的空间和在attachment中占用的空间相同。
- 一个attachment的宽和高都不应该小于32像素，一个attachment（特别是深度和模板attachment）如果小于32像素，将导致性能受损。
- 每次线程组内存分配应该到取整到16字节，所有的线程组内存分配总量不能超过16384字节。
- 对于一个MTLTextureDescriptor对象，height和width不能超过4096，depth不能超过2048。
- 对于绘制Encoder和并行计算Encoder，其参数索引表，缓存参数表最多有31项，纹理参数表最多有31项，采样器state参数表最多有16项。

The MTLFeatureSet_iOS_GPUFamily2_v1 feature set has the following Metal capabilities and limitations:

- A MTLRenderPassDescriptor can have up to 8 color attachments, in addition to a depth attachment and a stencil attachment.
- Each render pass can store a maximum of 32 bytes of color data per sample across all of its color attachments. (Depth and stencil attachments do not count against this limit.) If you create a MTLRenderPassDescriptor and the sum of the storage requirements for all color attachments is greater than the maximum allowed, a fatal error occurs.
- ASTC pixel formats are supported.
- All pixel formats consume a minimum of 4 bytes per sample in an attachment, even if the pixel formats use fewer than 4 bytes per pixel in memory. For example, the MTLPixelFormatR8Unorm, MTLPixelFormatR8Uint, and MTLPixelFormatR8Sint pixel formats use 1 byte per pixel in memory, but consume 4 bytes per sample in an attachment. The MTLPixelFormatRGB10A2Unorm, MTLPixelFormatRG11B10Float, and MTLPixelFormatRGB9E5Float pixel formats use 4 bytes per pixel in memory, but consume 8 bytes per sample in an attachment. All other pixel formats take the same amount of space in memory as in attachment storage.
- An attachment should not be smaller than 32 pixels in either width or height. An attachment (especially depth or stencil attachment) that is smaller than 32 pixels will have a performance cost.
- Every threadgroup memory allocation is rounded up to 16 bytes.
- The total of all threadgroup memory allocations must not exceed 16384 bytes.
  For a MTLTextureDescriptor, the height and width properties must not exceed 4096. The depth
- property must not exceed 2048.
- In the argument tables for the render and compute command encoders, there are a maximum of 31 entries in the buffer argument table, 31 entries in the texture argument table, and 16 entries in the sampler state argument table.

MTLFeatureSet_iOS_GPUFamily2_v1属性集有如下的Metal能力和限制：

- 一个MTLRenderPassDescriptor可以最多有8个颜色attachment，一个深度缓存，一个模板缓存。
- 每个绘制pass，其所有的颜色attachment的单像素存储空间之和最多为16个字节长（深度和模板attachment不算在内）。如果创建了一个MTLRenderPassDescriptor并且所有颜色attachment的存储需求的总和大于16字节，将导致错误发生。
- 支持ASTC像素格式。
- 在一个attachment中，所有的像素格式，其单个像素都消耗最少4个字节空间，就算这种像素格式内存中每像素尺寸少于4个字节。举例来说，MTLPixelFormatR8Unorm、MTLPixelFormatR8Uint、MTLPixelFormatR8Sint这三种像素格式在内存中每个像素占用1字节，但是在attachment中，每个像素消耗4字节。MTLPixelFormatRGB10A2Unorm、MTLPixelFormatRG11B10Float、MTLPixelFormatRGB9E5Float这三种像素格式在内存中每个像素占用4字节，但是在attachment中，每个像素消耗8字节。其他的像素格式在内存中的占用的空间和在attachment中占用的空间相同。
- 一个attachment的宽和高都不应该小于32像素，一个attachment（特别是深度和模板attachment）如果小于32像素，将导致性能受损。

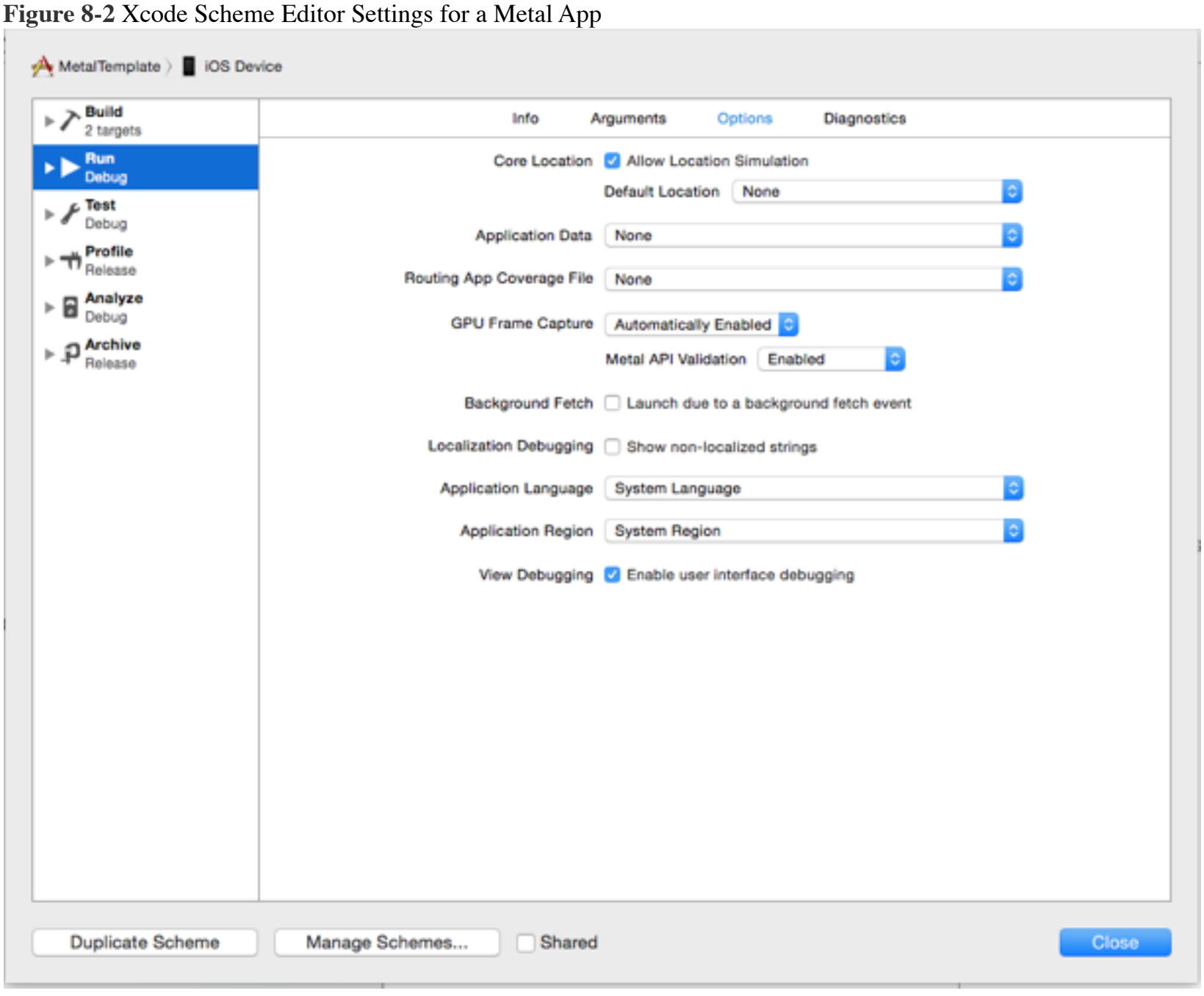
- 每次线程组内存分配应该到取整到16字节，所有的线程组内存分配总量不能超过16384字节。
- 对于一个`MTLTextureDescriptor`对象，`height`和`width`不能超过4096，`depth`不能超过2048。
- 对于绘制Encoder和并行计算Encoder，其参数索引表，缓存参数表最多有31项，纹理参数表最多有31项，采样器state参数表最多有16项。

Xcode Scheme Settings and Performance
Xcode Scheme 设置和性能

When running a Metal app from Xcode, the default scheme settings reduce performance. Xcode detects whether the Metal API is used in the source code and automatically enables the GPU Frame Capture and Metal API Validation settings, as seen in Figure 8-2 (page 68). When GPU Frame Capture is enabled, the debug layer is activated. When Metal API Validation is enabled, each call is validated, which affects performance further. For both settings, CPU performance is more affected than GPU performance. Unless you disable these settings, app performance may noticeably improve when the app is run outside of Xcode.

从Xcode中运行一个Metal应用，默认的sheme设置不利于提高性能。Xcode检查Metal API调用是否存在于源代码中，如果是将自动启用 GPU Frame Capture 和 Metal API Validation 设置。如图8-2所示。当GPU Frame Capture被启用，debug 层被激活。当Metal API Validation被启用，每次metal调用都被校验，这将更影响性能。这两个设置，对CPU的性能影响多过GPU。禁用这两个选项，app在Xcode之外运行，app的性能将显著提升，

Figure 8-2 Xcode Scheme Editor Settings for a Metal App



Debugging
调试

File Extension for Metal Shading Language Source Files
Metal着色语言源文件的文件后缀

For Metal shading language source code file names, you must use the `.metal` file name extension to ensure that the development tools (Xcode and the GPU frame debugger) recognize the source files when debugging or profiling.

Metal着色语言源代码文件的文件名使用`.metal`作为后缀，如此调试分析时可以让开发工具（Xcode 和 GPU frame debugger）认出。

Performing Frame Capture with Xcode
在Xcode中实现帧捕捉

If you want to perform frame capture in Xcode, enable debug and call the `insertDebugCaptureBoundary` method of `MTLCommandQueue` to inform Xcode. The `presentDrawable:` and `presentDrawable:atTime:` methods of `MTLCommandBuffer` similarly inform Xcode about frame capture, so call `insertDebugCaptureBoundary` only if those methods are not present.

如果希望在Xcode中实施帧绘制捕捉，启用debug并且调用`MTLCommandQueue`的`insertDebugCaptureBoundary`方法来通知Xcode。`MTLCommandBuffer`的`presentDrawable:`方法和`presentDrawable:atTime:`方法也通知Xcode进行帧捕捉。所以，仅在present方法失效时调用`insertDebugCaptureBoundary`方法。

Common Metal Properties

Table 8-1 (page 69) lists properties are defined for many objects in the Metal framework. During debugging, `label` can be used to identify an object.

下表列出了很多Metal框架中的对象都定义的属性，在调试中，label可以用来标识一个对象。

Table 8-1 Common Properties

Properties	Type	Description
`device`	`MTLDevice`	device used to execute the commands in this queue
`label`	`NSString *`	string used to identify the object

Document Revision History

This table describes the changes to *Metal Programming Guide* .

2014-09-17 New document that describes how to use the Metal framework to implement low-overhead graphics rendering or parallel computational

Date	Notes
2014-09-17	New document that describes how to use the Metal framework to implement low-overhead graphics rendering or parallel computational tasks.

Apple Inc.
1 Infinite Loop Cupertino, CA 95014 408-996-1010