Metal Programming Guide

Developer

Contents

Figures, Tables, and Listings

About Metal and this Guide

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

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.

At a Glance

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 shader programming language.

How to Use This Document

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 30) describes how to render 3D graphics, including how to distribute graphics operations across multiple threads.
- Data-Parallel Compute Processing: Compute Command Encoder (page 55) explains how to perform data-parallel processing.
- Buffer and Texture Operations: Blit Command Encoder (page 60) describes how to copy data between textures and buffers.
- Metal Tips and Techniques (page 62) lists development tips, such as how to build libraries off-line with compiled code.

Prerequisites

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

See Also

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.

Fundamental Metal Concepts

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 framebuffers.

  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.

Command Organization and Execution Model

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.

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:

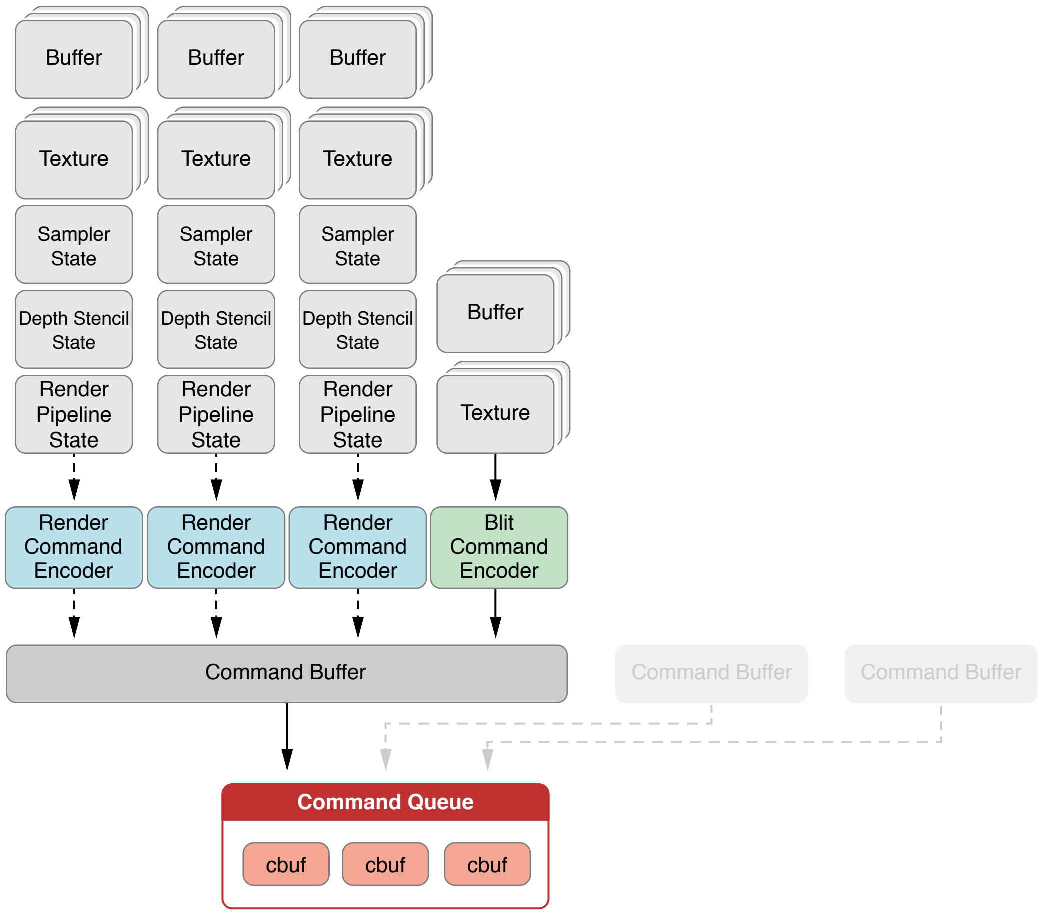
- The `MTLRenderCommandEncoder` protocol encodes graphics (3D) rendering commands for a single rendering pass,
- The `MTLComputeCommandEncoder` protocol encodes data-parallel computation workloads,
- The `MTLBlitCommandEncoder` protocol encodes simple copy operations between buffers and textures, and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

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 52).)

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.

Figure 2-1 Metal Object Relationships



Device—The 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.

Transient and Non-Transient Objects in Metal

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

Command Queue

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.

Command Buffer

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).

Creating a 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.

Executing Commands

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).)

Registering Handler Blocks for Command Buffer Execution

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 35).

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.

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.

Creating a Command Encoder Object

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, specified in a shared `MTLRenderPassDescriptor`.

Render Command 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 31)) 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 30).

Compute Command Encoder

For data-parallel computing, the `MTLComputeCommandEncoder` protocol provides methods to append commands in the command buffer that can specify the compute program and its arguments (e.g., texture, buffer, or sampler state), enqueue the compute program for execution, and enforce synchronization (execution ordering). 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 55).

Blit Command 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 60).

Multiple Threads, Command Buffers, and Command Encoders

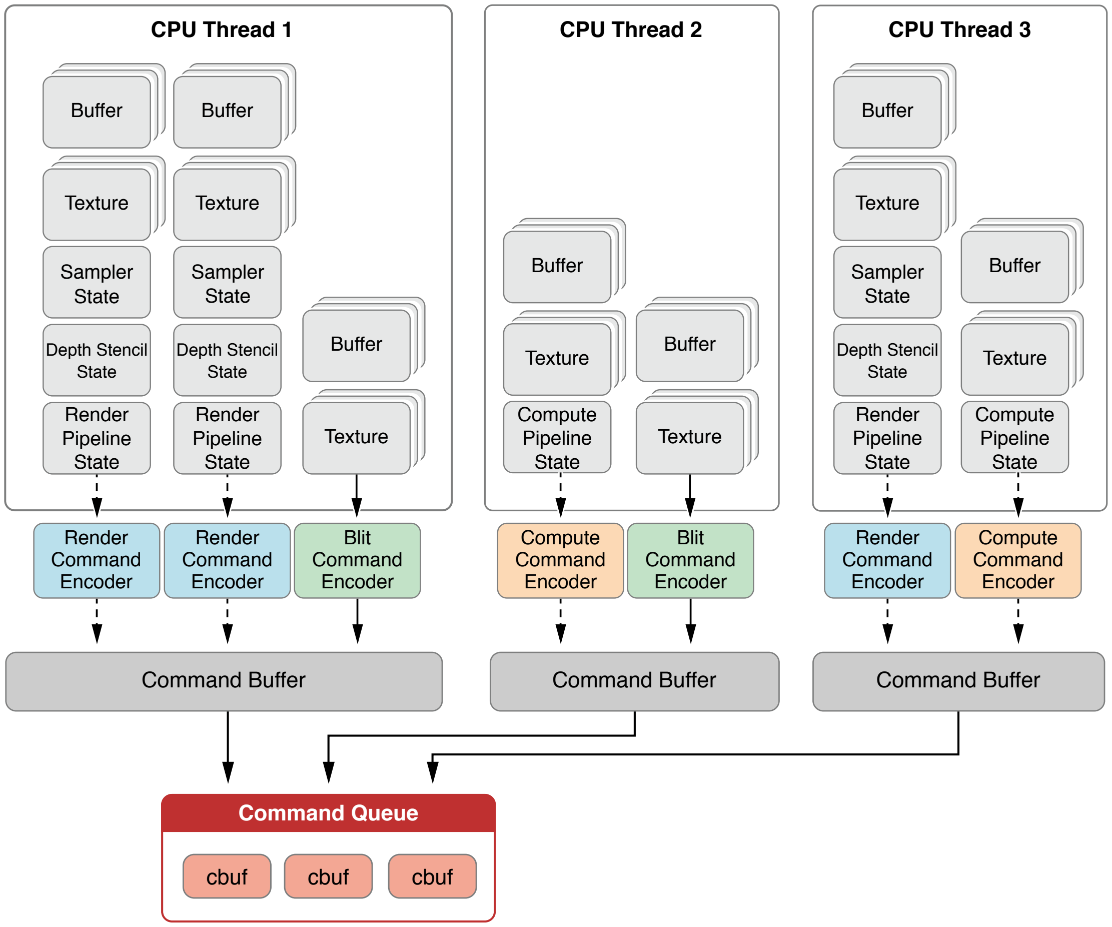
Most apps use a single thread to encode the rendering commands for a single frame in a single command buffer. At the end of each frame, you commit the command buffer, which both schedules and begins command execution.

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.

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.

Figure 2-2 Metal Command Buffers with Multiple Threads



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 52).

Resource Objects: Buffers and Textures

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

- A `MTLBuffer` object 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., a framebuffer 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.

Buffers are Typeless Allocations of Memory

Creating a Buffer Object

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.

Buffer Methods

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).

Textures are Formatted Image Data

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 a framebuffer 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.

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

Creating a Texture Object

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 they were stored in 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).

Texture Descriptors

A `MTLTextureDescriptor` object 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.

- 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.
- 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 20) shows code to create a texture descriptor `txDesc` and set its properties for a 3D, `64x64x64` texture.

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

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.

Texture Descriptor Convenience Creation Methods

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.

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

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`.

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:MTLTextureRegionMake2D(0,0,width,height)
    mipmapLevel:0 slice:0 withBytes:textureData
    bytesPerRow:myRowBytes bytesPerImage:myImageBytes];
```

Pixel Formats for Textures

A `MTLPixelFormat` object 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 most to least significant bit (MSB to LSB). For example, `MTLPixelFormatA2BGR10Uint` is a 32-bit packed format that consists of two bits for alpha and three 10-bit channels (for B, G, and R). Therefore, the first two high-order bits (MSB) contain alpha, the next 10 high-order bits contain blue, etc.
- 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.

The `MTLPixelFormat422_GBGR` and `MTLPixelFormat422_BGRG` 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:

- 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 framebuffer attachment that uses a texture with an sRGB pixel format (see Listing 5-1 (page 33)), the implementation converts the linear color values to sRGB, as follows:

- 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

A `MTLSamplerState` object 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` object and configures it in order to create a `MTLSamplerState` object. 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.

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

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 `MTLCommandStatusCommitted`).

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 `MTLCommandStatusCompleted`).

Functions and Libraries

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` object.

MTLFunction Represents a Shader or Compute Function

`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 Object (page 36). 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 56).

A Library is a Repository of Functions

`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.

Creating a Library from Compiled Code

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 62).

Creating a Library from Source Code

To create a `MTLLibrary` object 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.

Getting a Function from a Library

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.

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

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.
- `functionArguments`, an array of `MTLFunctionArgument` objects, each representing one argument.

Function Arguments

For each `MTLFunctionArgument` in the `functionArguments` array, you can query the following properties at run time:

- `name`, a string with the name of the argument.
- `argumentType`, the type (texture, buffer, sampler, or local memory) of the `MTLResource` identified by the function argument.
- `bindingIndex`, the index of the corresponding argument table.

The `MTLFunction` object that corresponds to the Metal shading language code in Listing 4-2 (page 28) has a `functionArguments` property that is an array with one `MTLFunctionArgument` element. The value of the `name` property for that function argument is `"img"`. Since the function argument is a `MTLTexture` object, the value of the `argumentType` property is `MTLArgumentTypeTexture`. The function also uses the `texture` attribute qualifier with the index set to 0, which identifies the location on the argument table for this texture, so the value of the `bindingIndex` property is 0.

Listing 4-2 Metal Shading Language Function Declaration Example

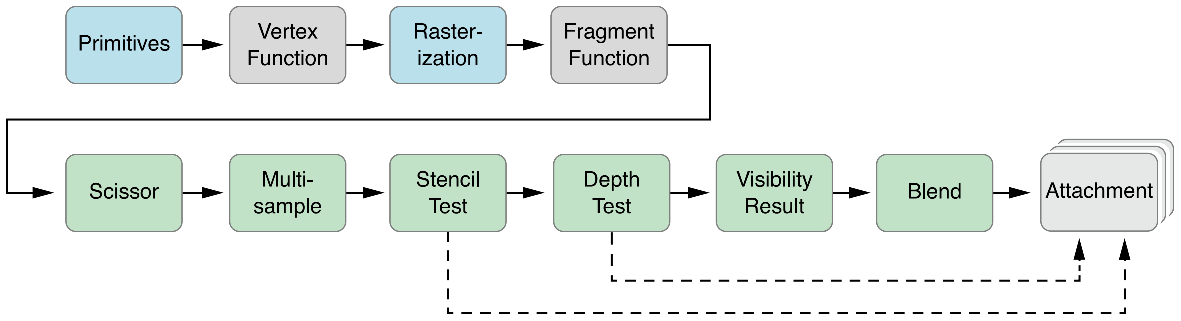
```
void metal_func (texture2d<float, access::write> img [[ texture(0) ]])
```

```
{
...
}
```

Graphics Rendering: Render Command Encoder

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 30).

Figure 5-1 Metal Graphics Rendering Pipeline



A `MTLRenderCommandEncoder` object 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 52).

Creating and Using a Render Command Encoder

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

- 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` object by calling the `renderCommandEncoderWithDescriptor:` method of `MTLCommandBuffer` with the specified render pass descriptor.

- Create a `MTLRenderPipelineState` object 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 Object (page 36).
- Set textures, buffers, and samplers to be used by the `MTLRenderCommandEncoder`, as described in Specifying Resources for a Render Command Encoder (page 41).
- Call `MTLRenderCommandEncoder` methods to specify additional fixed-function state, including the depth and stencil state, as explained in Fixed-Function State Operations (page 46).
- Finally, call `MTLRenderCommandEncoder` methods to draw graphics primitives, as described in Drawing Geometric Primitives (page 49).

Creating a Render Pass Descriptor

`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 46).)

Each individual attachment, including the texture that will be written to, is represented by an `MTLRenderPassAttachmentDescriptor`. 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, use a color-renderable pixel format. For a depth attachment, use a depth-renderable pixel format, such as `MTLPixelFormatDepth32Float`. For a stencil attachment, use a stencil-renderable pixel format, such as `MTLPixelFormatStencil8`.

Up to 16 bytes of color data can be stored per sample. 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.

`loadAction` and `storeAction` are `MTLRenderPassAttachmentDescriptor` 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 52).)

`loadAction` values include:

- `MTLLoadActionClear`, which writes the same value to every pixel in the specified attachment descriptor. (For more detail about `MTLLoadActionClear`, see MTLClearValue Specifies the Clear Load Action (page 34).)
- `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.

`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.

Listing 5-1 (page 33) 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 in the `MTLRenderPassAttachmentDescriptor` objects that represent the first color attachment in the array (at index 0) and the depth attachment are set.

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 *rpDesc = [MTLRenderPassDescriptor renderPassDescriptor];
rpDesc.colorAttachments[0].texture = colorTex;
rpDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
rpDesc.colorAttachments[0].storeAction = MTLStoreActionStore;
rpDesc.colorAttachments[0].clearValue = MTLClearValueMakeColor(0.0,1.0,0.0,1.0);

rpDesc.depthAttachment.texture = depthTex;
rpDesc.depthAttachment.loadAction = MTLLoadActionClear;
rpDesc.depthAttachment.storeAction = MTLStoreActionStore;
rpDesc.depthAttachment.clearValue = MTLClearValueMakeDepth(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 33), an attachment is initially created and then its `loadAction`, `storeAction`, `texture`, and `resolveTexture` properties are set to support multisample resolve.

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 *rpDesc = [MTLRenderPassDescriptor renderPassDescriptor];
```

```
rpDesc.colorAttachments[0].texture = msaaTex;
rpDesc.colorAttachments[0].resolveTexture = colorTex;
rpDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
rpDesc.colorAttachments[0].storeAction = MTLStoreActionMultisampleResolve;
rpDesc.colorAttachments[0].clearValue = MTLClearValueMakeColor(0.0,1.0,0.0,1.0);
```

Specifying the Clear Load Action

If the `loadAction` property of an attachment descriptor is set to `MTLLoadActionClear`, then the `clearValue` property contains a `MTLClearValue` type value that is written to every pixel in the specified attachment descriptor at the start of a rendering pass. The `clearValue` property depends upon the pixel format of the texture, which determines how the attachment is used.

- For an attachment descriptor that uses a color texture, `clearValue` contains a `MTLClearColor` value that consists of four double-precision floating-point RGBA components. `MTLClearValueMakeColor(red, green, blue, alpha)` creates the corresponding `MTLClearColor` value. The default value is `nil`, which is the same as (0.0, 0.0, 0.0, 1.0), which is opaque black.
- For an attachment descriptor that uses a depth texture, `clearValue` contains a `MTLClearColor` value that is one double-precision floating-point value in the range [0.0, 1.0]. `MTLClearValueMakeDepth(depth)` creates the corresponding `MTLClearColor` value. The default value is 1.0.
- For an attachment descriptor that uses a stencil texture, `clearValue` contains a `MTLClearColor` value that is one 64-bit unsigned integer. `MTLClearValueMakeStencil(stencil)` creates the corresponding `MTLClearColor` value. The default value is 0.

Using the Render Pass Descriptor to Create a Render Command Encoder

After the creation and specification of `MTLRenderPassDescriptor`, the `renderCommandEncoderWithDescriptor:` method of `MTLCommandBuffer` uses the render pass descriptor `rpDesc` to create the `MTLRenderCommandEncoder`, as seen in Listing 5-3 (page 34).

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

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

Integration with Core Animation: CAMetalLayer

Core Animation defines the `CAMetalLayer` class, which is designed for the specialized behavior of a layer-backed view with Metal rendered content. `CAMetalLayer` represents information about the geometry of the content (e.g., position and size), its visual attributes (e.g., background color, border, and shadow), and the resources used by Metal to present the content in a color attachment. `CAMetalLayer` encapsulates the timing of content presentation, which enables the content to be displayed as soon as it is available ("do it now") or presented 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 `CAMetalDrawable`, which is a protocol that represents a displayable resource. `CAMetalDrawable` extends `MTLDrawable` and vends an object that conforms to `MTLTexture`, which may be used as a destination for rendering commands. To render into a `CAMetalLayer`, you should get a new `CAMetalDrawable` for each rendering pass, get the `MTLTexture` 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, you should reuse them among rendering passes.)

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

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

- `device`, the `MTLDevice` that the resource is created from.
- `pixelFormat`, the pixel format of the texture of the `CAMetalDrawable` object to be created. The supported values are `MTLPixelFormatBGRA8Unorm` (the default) and `MTLPixelFormatBGRA8Unorm_sRGB`.
- `drawableSize`, the pixel dimensions of the texture of the `CAMetalDrawable` object to be created.
- `framebufferOnly`, whether or not the texture of the `CAMetalDrawable` can only be used for attachments (`YES`) or whether it can also be used for texture sampling and pixel read/write operations (`NO`). If `YES`, `CAMetalLayer` can allocate the `MTLTexture` objects in ways that are optimized for display purposes that makes them unsuitable for sampling. For most apps, the recommended value for is `YES`.
- `presentsWithTransaction`, 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).

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

- `texture`, a `MTLTexture` object that is typically used to create a `MTLRenderPipelineAttachmentDescriptor`.
- `layer`, the `CAMetalLayer` object that responsible for displaying the drawable.

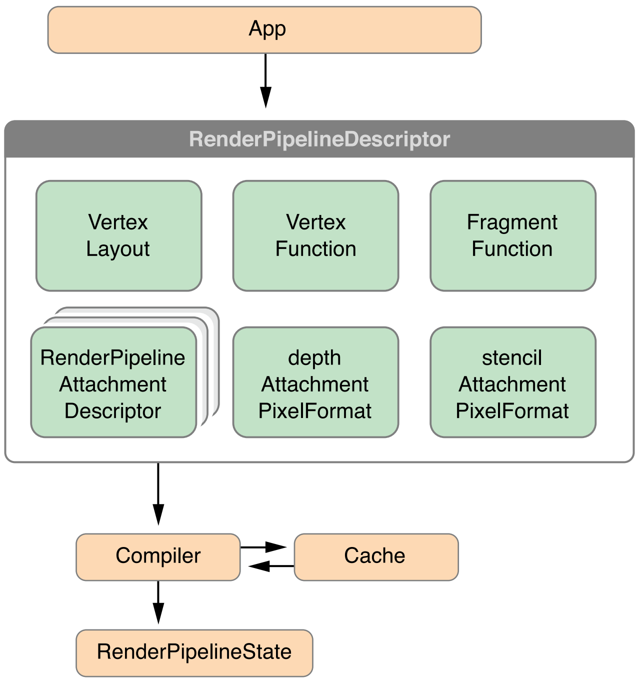
To call the `present` method for one `CAMetalDrawable` when the command buffer has been scheduled on the device, you can call the `presentDrawable:` convenience method of `MTLCommandBuffer`. `presentDrawable:` uses the scheduled handler to present one drawable, which covers most scenarios.

Creating a Render Pipeline State Object

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 36).

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



Set these properties in `MTLRenderPipelineDescriptor`:

- For each active color attachment, set a `MTLRenderPipelineAttachmentDescriptor` at an attachment index in the `colorAttachments` property of `MTLRenderPipelineDescriptor`. The attachment descriptor specifies the blend operations and factors for the attachment, as detailed in Configuring Blending in a Render Pipeline Attachment Descriptor (page 39). The attachment descriptor also specifies the pixel format of the texture of the attachment. The pixel format for the attachment descriptor must match the pixel format for the texture of the `MTLRenderPipelineDescriptor` with the corresponding attachment index, or an error occurs.
- To enable writing to the depth attachment, set the `depthWriteEnabled` property of `MTLRenderPipelineDescriptor` to `YES`. Also set the `depthAttachmentPixelFormat` property to match the pixel format for the texture of `depthAttachment` in `MTLRenderPassDescriptor`.
- Similarly, to enable writing to the stencil attachment, set the `stencilWriteEnabled` property to `YES`. Also 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 Fetching Data (page 43).
- 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.
    - `sampleMask` property specifies a bitmask that is initially bitwise ANDed with the coverage mask produced by the rasterizer. By default, the `sampleMask` bitmask is all ones, so a bitwise AND with that bitmask does not change any values.
    - 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.)

- sampleCoverage specifies a value (between 0.0 and 1.0, inclusive) that is used to determine a coverage mask, which is then bitwise ANDed with the coverage value produced by the rasterizer. (For more details on the coverage mask, see MTLRenderPipelineDescriptor.)

After creating MTLRenderPipelineDescriptor and specifying its properties, use it to create the MTLRenderPipelineState. Since creating a MTLRenderPipelineState can require an expensive evaluation of graphics state and a possible compilation of the specified graphics shaders, you can use either a blocking or asynchronous method.

- To synchronously compile the graphics state and create the render pipeline state object, call the newRenderPipelineStateWithDescriptor:error: method of MTLDevice.
- To asynchronously compile the graphics state and register a handler to be called when the render pipeline state object is created, call the newRenderPipelineStateWithDescriptor:completionHandler: method of MTLDevice.

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.

Listing 5-4 (page 38) demonstrates the creation of a render pipeline state object pipeline. vertFunc and fragFunc are shader functions that are specified as properties of the render pipeline state descriptor psDesc. 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.

Listing 5-4 Creating a Simple Pipeline State

```
MTLRenderPipelineDescriptor *psDesc = [[MTLRenderPipelineDescriptor alloc] init];
psDesc.vertexFunction = vertFunc;
psDesc.fragmentFunction = fragFunc;
psDesc.colorAttachments[0].pixelFormat = MTLPixelFormatRGBA8Unorm;

// Create MTLRenderPipelineState from MTLRenderPipelineDescriptor
NSError *errors = nil;
id <MTLRenderPipelineState> pipeline = [device
newRenderPipelineStateWithDescriptor:psDesc
                                        error:&errors];
assert(pipeline && !errors);
```

```
// Set the pipeline state for MTLRenderCommandEncoder
[renderCE setRenderPipelineState:pipeline];
```

Configuring Blending in a Render Pipeline Attachment Descriptor

Blending uses a highly configurable blend operation to mix the output returned by the fragment function (source) with pixel values in the framebuffer (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 `MTLRenderPipelineAttachmentDescriptor` 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 46).

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

Listing 5-5 (page 40) shows code for a custom blending configuration, using the blend operation `MTLBlendOperationAdd`, the source blend factor `MTLBlendFactorOne`, and the destination blend factor `MTLBlendFactorOneMinusSourceAlpha`. Properties of `colorAttachments[0]`, which is a `MTLRenderPipelineAttachmentDescriptor` object that describes a color attachment, specify the blending configuration.

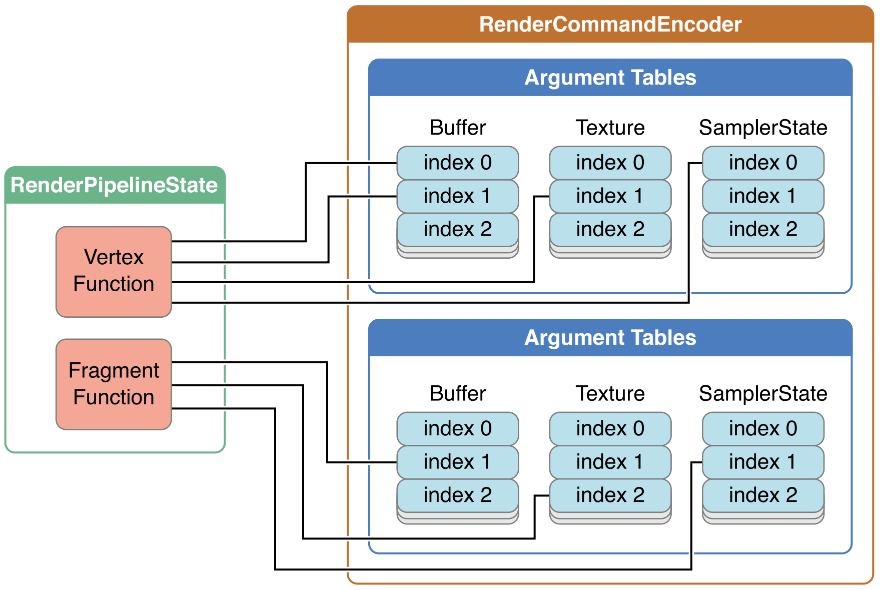
Listing 5-5 Specifying A Custom Blend Configuration

```
MTLRenderPipelineDescriptor *psDesc = [[MTLRenderPipelineDescriptor alloc] init];
psDesc.colorAttachments[0].blendingEnabled = YES;
psDesc.colorAttachments[0].rgbBlendOperation = MTLBlendOperationAdd;
psDesc.colorAttachments[0].alphaBlendOperation = MTLBlendOperationAdd;
psDesc.colorAttachments[0].sourceRGBBlendFactor = MTLBlendFactorOne;
psDesc.colorAttachments[0].sourceAlphaBlendFactor = MTLBlendFactorOne;
psDesc.colorAttachments[0].destinationRGBBlendFactor =
       MTLBlendFactorOneMinusSourceAlpha;
psDesc.colorAttachments[0].destinationAlphaBlendFactor =
       MTLBlendFactorOneMinusSourceAlpha;

NSError *errors = nil;
id <MTLRenderPipelineState> pipeline =
       [device newRenderPipelineStateWithDescriptor:psDesc error:&errors];
```

Specifying Resources for a Render Command Encoder

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 41).

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.

- `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.

- `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 42), two buffers (`posBuf` and `texCoordBuf`) with indices 0 and 1, respectively, are defined for the vertex shader.

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];
```

In Listing 5-7 (page 42), the function signature has corresponding arguments with the attribute qualifiers `buffer(0)` and `buffer(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) ]])
```

Similarly in Listing 5-8 (page 42), three resources, a buffer, a texture, and a sampler (`fragmentColorBuf`, `shadeTex`, and `sampler`, respectively), all with index 0, are defined for the fragment shader.

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];
```

In Listing 5-9 (page 43), the function signature has corresponding arguments with the attribute qualifiers `buffer(0)`, `texture(0)`, and `sampler(0)`, respectively.

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 Fetching Data

`MTLVertexDescriptor` describes the organization of vertex data in the attribute's argument table. `MTLVertexDescriptor` enables a vertex shader to load data through an argument table without knowing how the data is organized in memory. `MTLVertexDescriptor` supports access to multiple attributes (such as vertex coordinates, surface normals, and texture coordinates) that are interleaved within the same buffer.

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. Each field of the per-vertex input struct specifies a location in the vertex attribute argument table with the `[[attribute(index)]]` qualifier.

In Metal framework code that corresponds with the vertex shader, create a `MTLVertexDescriptor` object that corresponds to the `[[stage_in]]` input argument. To specify the format of each attribute in the vertex buffer (i.e., each field in the argument passed to the vertex function), call `setVertexFormat:offset:vertexBufferIndex:atAttributeIndex:`.

If `vertexFormat` is not an exact match for the corresponding shading language type, the Metal framework tries to convert or expand the vertex attribute to agree. For example, if the shading language type is `half4`, and in the framework `vertexFormat` is `MTLVertexFormatFloat2`, then when the data is used as an argument to the vertex function, it is converted from float to half and expanded from two to four elements (with 0.0, 1.0 in the last two elements).

To specify how to reference the attribute array when rendering, call `setStride:stepFunction:stepRate:atVertexBufferIndex:`. `stepFunction` and `stepRate` determine how often new attribute data are fetched from the buffer.

- If `stepFunction` is `MTLVertexStepFunctionPerVertex`, `MTLVertexDescriptor` enables the shader to fetch attribute data based on the `[[ vertex_id ]]` attribute qualifier, and the shader fetches new attribute data for every vertex. `stepRate` should be set to 1.

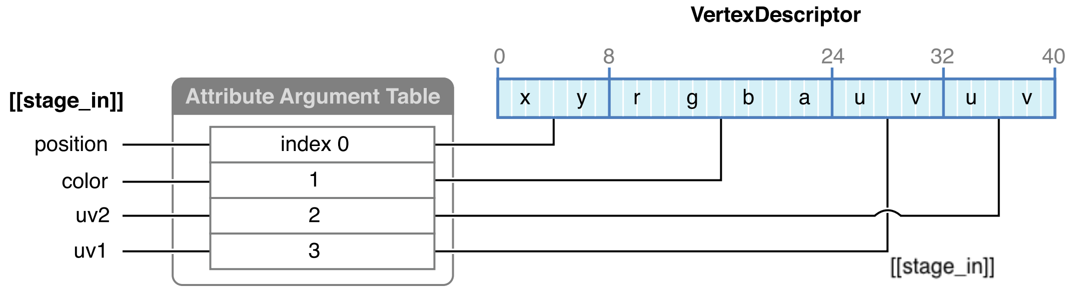
- If `stepFunction` is `MTLVertexStepFunctionPerInstance`, `MTLVertexDescriptor` enables the shader to fetch attribute data based on the `[[ instance_id ]]` attribute qualifier. `stepRate` determines how often the shader fetches new attribute data. If `stepRate = 1`, the shader fetches attribute data for every instance, if `stepRate = 2`, for every two instances, and so on. `stepRate` must be a positive integer.
- If `stepFunction` is `MTLVertexStepFunctionConstant`, then the shader fetches attribute data just once. `stepRate` should be set to 0.

The `setStride:atVertexBufferIndex:` method enables the shader to fetch new attribute data for every vertex. It is the same as calling `setStride:stepFunction:stepRate:atVertexBufferIndex:` with `stepFunction = MTLVertexStepFunctionPerVertex` and `stepRate = 1`.

For every pipeline state, there can be only one `MTLVertexDescriptor`. To establish the vertex descriptor of the argument for the vertex function associated with the pipeline, set the `vertexDescriptor` property of `MTLRenderPipelineState`.

Figure 5-4 (page 44) illustrates an example how a `MTLVertexDescriptor` object shares resource location information between the shading language and framework code. The left side of Figure 5-4 (page 44) depicts how the fields in the Metal shading language code in Listing 5-10 (page 44) occupy the indices in the attribute argument table. The right side of Figure 5-4 (page 44) shows how the `MTLVertexDescriptor` in the Metal framework code in Listing 5-11 (page 45) references the attribute argument table to define the organization of the vertex data in the buffer object. The order of the fields in the shading language code does not have to be preserved in the buffer object in the framework code.

Figure 5-4 Vertex Attribute Organization: Correspondence Between Shading Language and Framework



In Listing 5-10 (page 44), the code for a vertex shader defines a struct called `VertexInput3` with `attribute` qualifiers for each of its fields.

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

```
struct VertexInput3 {
    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 vertex3(VertexInput3 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;\n"
}
```

In Listing 5-11 (page 45), a function is first created from the `vertex3` entry point in a library. Then the `MTLVertexDescriptor` `vertexDesc` is created for the `MTLRenderPipelineDescriptor`. `setStride:stepFunction:stepRate:atVertexBufferIndex:` specifies the stride between attribute data for every vertex. Several `setVertexFormat:offset:vertexBufferIndex:atAttributeIndex:` calls then specify the format of each attribute.

Listing 5-11 Metal Framework: Specifying a Vertex Descriptor

```
id <MTLFunction> vertexFunc = [library newFunctionWithName:@"vertex3"];

MTLRenderPipelineDescriptor* pipelineDesc = [[MTLRenderPipelineDescriptor alloc]
init];

NSUInteger interleavedBufferStride = 10 * sizeof(float);
MTLVertexDescriptor* vertexDesc = [[MTLVertexDescriptor alloc] init];

[vertexDesc setStride:interleavedBufferStride
            stepFunction:MTLVertexStepFunctionPerVertex
            stepRate:1 atVertexBufferIndex:0];
[vertexDesc setVertexFormat:MTLVertexFormatFloat2
            offset:0 vertexBufferIndex:0 atAttributeIndex:0];
[vertexDesc setVertexFormat:MTLVertexFormatFloat4
            offset:(2 * sizeof(float)) vertexBufferIndex:0 atAttributeIndex:1];
[vertexDesc setVertexFormat:MTLVertexFormatFloat2
            offset:(6 * sizeof(float)) vertexBufferIndex:0 atAttributeIndex:3];
[vertexDesc setVertexFormat:MTLVertexFormatFloat2
            offset:(8 * sizeof(float)) vertexBufferIndex:0 atAttributeIndex:2];
```

```
pipelineDesc.vertexDescriptor = vertexDesc;
pipelineDesc.vertexFunction = vertFunc;
```

Fixed-Function Render Command Encoder Operations

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 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 47).
- `setStencilReferenceValue:` specifies the stencil reference value.
- `setDepthBias:slopeScale:clamp:` specifies an adjustment for comparing shadow maps to the depth values output from fragment shaders.
- `setDepthClipMode:` establishes whether to clamp fragments that are outside the depth clipping planes to the near or far value (`MTLDepthClipModeClamp`) or to discard (clip) fragments outside those planes (`MTLDepthClipModeClip`, the default).

- `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 31)). 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 39).

Depth and Stencil States

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`. If `MTLDepthStencilState` is `nil`, the resulting render command encoder has the default depth/stencil state.
4. If the stencil test is enabled, 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. If the stencil test is enabled, 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`.

Using a Depth/Stencil 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`. (Also set the `depthWriteEnabled` property in `MTLRenderPipelineDescriptor`.)

- `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 48) 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.

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];
```

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.

In Listing 5-12 (page 48), 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`).

Drawing Geometric Primitives

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 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.

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 46)).

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.

Finish Encoding for the Render Command Encoder

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.

Code Example: Drawing a Triangle

Listing 5-13 (page 51) shows how you can write Metal code that uses several `MTLRenderCommandEncoder` methods to draw a single triangle. (Presume that the `device` variable contains a `MTLDevice` object, and `currentTexture` contains a `MTLTexture` object that will be used as a color attachment.) First create a `MTLCommandQueue` object, which is then used to create a `MTLCommandBuffer` object. Next create two `MTLBuffer` objects, `posBuf` and `colBuf`, and copy vertex coordinate and vertex color data, respectively, into their storage. After that, create a new `MTLRenderCommandEncoder` object and use it to perform the following actions:

- Establish vertex and fragment shader functions. In this example, create a `MTLLibrary` object with source code from the single `NSString` variable `progSrc`. Next call the `newFunctionWithName:` method of `MTLLibrary` to create a `MTLFunction` object, `vertFunc`, that represents the shader function called `hello_vertex`. Then call the `setVertexShader:` method of `MTLRenderCommandEncoder` with a `MTLFunction` object to set the vertex function. Perform similar calls to create a `MTLFunction` for the fragment shader `fragFunc` and then set the fragment function with `setFragmentShader:`.

- Next call the `setVertexBuffer:offset:atIndex:` method of `MTLRenderCommandEncoder` twice to append commands that specify the coordinates and colors. The `atIndex` input value for the `setVertexBuffer:offset:atIndex:` method corresponds to the attribute `buffer(atIndex)` in the source code of the vertex shader.
- Then call the `drawPrimitives:vertexStart:vertexCount:` method of `MTLRenderCommandEncoder` to append commands to perform the rendering of a filled triangle (type `MTLPrimitiveTypeTriangle`).
- Finally call the `endEncoding` method of `MTLRenderCommandEncoder` to end encoding for this rendering pass.

After that, call the `commit` method of `MTLCommandBuffer` to execute the commands on the device.

Listing 5-13 Metal Code for Drawing a Triangle

```
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 colorData[] = {
        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 <MTLCommandQueue> commandQueue = [device newCommandQueue];
id <MTLCommandBuffer> commandBuffer = [commandQueue commandBuffer];

id <MTLBuffer> posBuf = [device newBufferWithBytes:posData
        length:sizeof(posData)
        options:nil];
id <MTLBuffer> colBuf = [device newBufferWithBytes:colorData
        length:sizeof(colorData)
        options:nil];

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];

MTLRenderPassDescriptor *rpDesc = [MTLRenderPassDescriptor renderPassDescriptor];
rpDesc.colorAttachments[0].texture = currentTexture;
rpDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
rpDesc.colorAttachments[0].clearValue = MTLClearValueMakeColor(0.0,1.0,1.0,1.0);
```

```
id <MTLRenderCommandEncoder> renderEncoder =
           [commandBuffer renderCommandEncoderWithDescriptor:rpDesc];

MTLRenderPipelineDescriptor* rpDesc = [[MTLRenderPipelineDescriptor alloc] init];
rpDesc.vertexFunction = vertFunc;
rpDesc.fragmentFunction = fragFunc;
rpDesc.colorAttachments[0].pixelFormat = currentTexture.pixelFormat;
pipeline = [device newRenderPipelineStateWithDescriptor:rpDesc error:&errors];


[renderEncoder setRenderPipelineState:pipeline];
[renderEncoder setVertexBuffer:posBuf offset:0 atIndex:0];
[renderEncoder setVertexBuffer:colBuf offset:0 atIndex:1];
[renderEncoder drawPrimitives:MTLPrimitiveTypeTriangle
               vertexStart:0 vertexCount:3];
[renderEncoder endEncoding];
[commandBuffer commit];
```

In Listing 5-13 (page 51), the `MTLFunction` object represents the shader function called `hello_vertex`. The `setVertexBuffer:` method of `MTLRenderCommandEncoder` is used to specify the vertex resources (in this case, two buffer objects) that are passed as arguments into `hello_vertex` and the indices on the buffer argument table, specified by `atIndex`. Listing 5-14 (page 52) shows the declaration of the corresponding shader code for `hello_vertex` with indices specified by the `buffer(x)` attributes that match `atIndex` in the `setVertexBuffer:` calls.

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

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.

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.

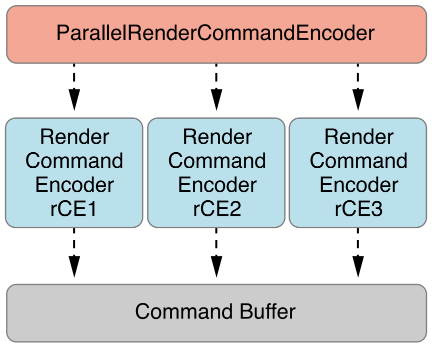
Listing 5-15 (page 53) shows a model for how `MTLParallelRenderCommandEncoder` works. In this example, the `MTLParallelRenderCommandEncoder` creates three `MTLRenderCommandEncoder` objects `rCE1`, `rCE2`, and `rCE3`.

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].clearValue =
MTLClearValueMakeColor(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 5-5 (page 54).

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


ParallelRenderCommandEncoder
Render Command Encoder rCE1
Render Command Encoder rCE2
Render Command Encoder rCE3
Command Buffer

Data-Parallel Compute Processing: Compute Command Encoder

This chapter explains how to create and use a `MTLComputeCommandEncoder` object, 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 55). 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 56). 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` object can be associated to a number of resource objects.
- execute the `MTLComputePipelineState` a specified number of times, as explained in Compute State Execution and Synchronization (page 57).

Creating a Compute 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. To create a `MTLComputePipelineState` object, call one of the following two `MTLDevice` methods with a compute function object:

- `newComputePipelineStateWithDescriptor:error:` synchronously compiles the compute function and creates a `MTLComputePipelineState` object.
- `newComputePipelineStateWithDescriptor:completionHandler:` asynchronously compiles the compute function and registers a handler to be called when the `MTLComputePipelineState` object has been created.

Creating a `MTLComputePipelineState` object may be an expensive operation that involves the possible compilation of the compute function.

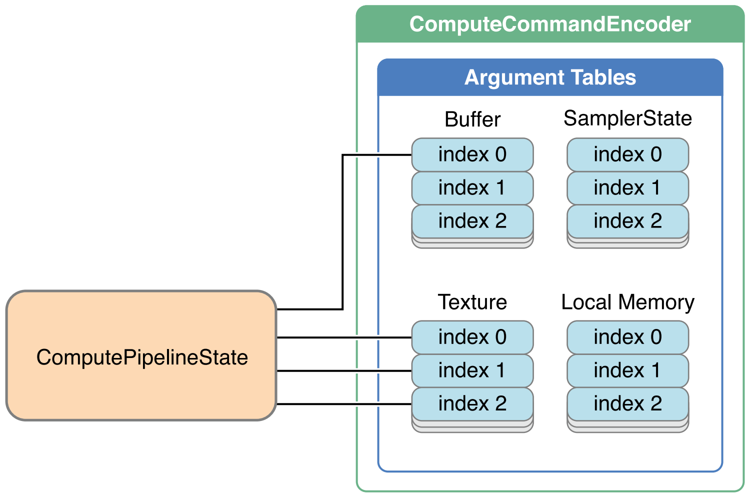
Specify a Compute State and Resources for a Compute Command Encoder

The `setComputePipelineState:` method specifies the compute state to use. The following `MTLComputeCommandEncoder` methods specify a resource (i.e., a buffer, texture, sampler state, or local memory) that is used as an argument to the compute function represented by the `MTLComputePipelineState` object.

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

Each method assigns one or more resources to the corresponding argument(s), as illustrated in Figure 6-1 (page 57).

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 local memory allocations must not exceed 16K bytes; otherwise, an error occurs.

Compute State Execution and Synchronization

To encode a command to execute a compute function, call the `executeKernelWithWorkGroupSize:workGroupCount:` method of `MTLComputeCommandEncoder` with specified work group dimensions in `workGroupSize` and the number of work groups in `workGroupCount`. You can query the `preferredWorkGroupFactor` and `maxWorkGroupSize` properties of `MTLComputePipelineState` to improve compute state execution on this device. For most efficient execution, the `workGroupSize` should be a multiple of `preferredWorkGroupFactor`. `maxWorkGroupSize` specifies the largest number of work items that can be in a single compute work group.

By default, within a given `MTLComputeCommandEncoder`, compute functions can complete in arbitrary order. If you use the output of one compute function as input for another compute function, you need to ensure that the pre-requisite functions have written their outputs before the dependent functions begin execution.

To block the execution of any subsequently encoded commands in this command buffer until the device executes the barrier command, you can call the `executeBarrier` method of `MTLComputeCommandEncoder`. You can use barrier commands to enforce ordering for the execution of a series of dependent compute functions. The barrier command also serves as a memory barrier, so all writes issued by compute functions executed before the barrier will be completed and visible to all loads that occur in compute functions after the barrier. Without the barrier, there are no guarantees of memory consistency between simultaneously executing compute functions.

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.

Code Template for Data-Parallel Programs

Listing 6-1 (page 58) 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` object is created that represents the entry point `filter_main` from the `MTLLibrary`, shown in Listing 6-2 (page 59). Then the function object is used to create a `MTLComputePipelineState` object called `filterState`.

The compute function performs an image transformation and filtering operation on the image `inputImage` with the results returned in `outputImage`. First the `setTexture:` and `setBuffer:` 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 work-group of size 16 x 16 pixels in each dimension. The `executeKernelWithWorkGroupSize:workGroupCount:` method encodes the command to execute 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.

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
newComputePipelineStateWithDescriptor:func error:&errors];
[computeDesc release];
[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 workgroupSize = {16, 16, 1};
MTLSize numWorkgroups = {inputImage.width/workgroupSize.width,
                        inputImage.height/workgroupSize.height, 1};

[computeCE executeKernelWithWorkGroupSize:workgroupSize
                                          workGroupCount:numWorkgroups];
[computeCE endEncoding];

// Commit the command buffer
[commandBuffer commit];
```

In Listing 6-2 (page 59), the code for the user-defined functions `read_and_transform` and `filter_table` is not shown.

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                                    [[ global_id ]],
  texture2d<float,access::sample> table        [[ texture(2) ]],
  constant Parameters* params                  [[ 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

`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.

Copying Data in GPU Memory Between Resource Objects

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.

Copying Data Between Two Buffers

`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.

Copying Data From a Buffer to a Texture

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

Copying Data Between Two Textures

`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`.

Copying Data From a Texture to a Buffer

`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`.

Generating 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).

Filling the Contents of a Buffer

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

Finish Encoding for the Blit Command Encoder

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.

Metal Tips and Techniques

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

Creating Libraries During the App Build Process

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.

Using Xcode to Build a 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`.

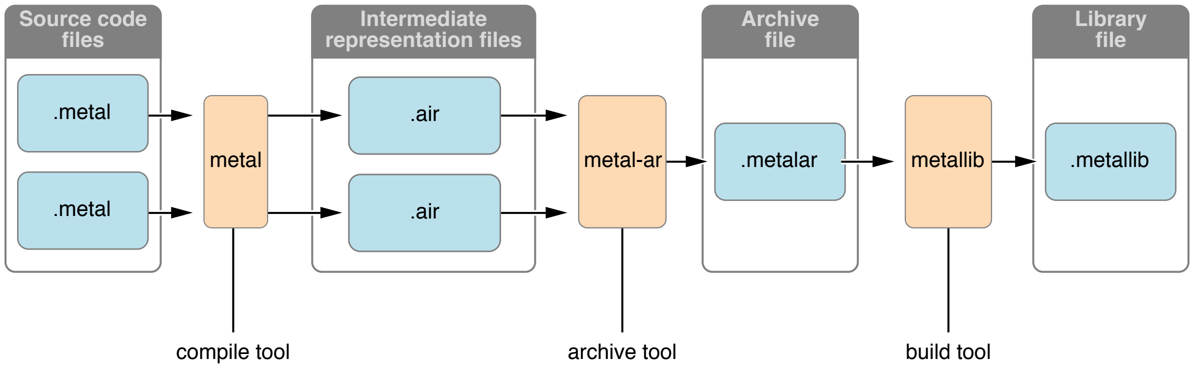
Using Command Line Utilities to Build a Library

Figure 8-1 (page 63) 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.

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



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

Metal Capabilities and Limitations

Metal has the following capabilities and limitations:

- A framebuffer can have up to 4 color framebuffer attachments, in addition to a depth attachment and a stencil attachment.
- A framebuffer can store 16 bytes of color data per sample. (Depth and stencil framebuffer attachments do not count against this limit.) If you create a `MTLFramebufferDescriptor` and the sum of the storage requirements for all color framebuffer attachments is greater than the maximum allowed, a fatal error occurs.
- All pixel formats consume a minimum of 4 bytes per sample in the framebuffer, 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 the framebuffer. The `MTLPixelFormatA2BGR10Unorm`, `MTLPixelFormatB10GR11Float`, and `MTLPixelFormatSE5BGR9Float` pixel formats use 4 bytes per pixel in memory, but consume 8 bytes per sample in the framebuffer. All other pixel formats take the same amount of space in memory as in framebuffer storage.
- Every local memory allocation is rounded up to 16 bytes.
- The total of all local memory allocations must not exceed 16K bytes; otherwise, an error occurs.
- For a `MTLTextureDescriptor`, the `height` and `width` properties must not exceed 4096. The `depth` property must not exceed 2048. Otherwise, an error results.

- 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.

Debugging

Table 8-1 (page 64) lists properties are defined for many objects in the Metal framework. In particular, `label` can be used to identify an object.

Table 8-1 Common Properties

Common Metal 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* .

Date	Notes
2014-06-12	New document that describes how to use the Metal framework to implement low-overhead graphics rendering or parallel computational tasks.