Metal Programming Guide

Developer

Contents

Figures, Tables, and Listings

Objective-CSwift

About Metal and This Guide

The Metal framework supports GPU-accelerated advanced 3D graphics rendering and data-parallel computation workloads. Metal provides a modern and streamlined API for fine-grained, low-level control of the organization, processing, and submission of graphics and computation commands, as well as the management of the associated data and resources for these commands. A primary goal of Metal is to minimize the CPU overhead incurred by executing 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 for graphics shader and data-parallel computation functions. The document then details how to use the Metal API to write an app.

You can find more details in the following chapters:

- Fundamental Metal Concepts (page 10) briefly describes the main features of Metal.
- Command Organization and Execution Model (page 11) explains how to create, commit, and submit commands to the GPU for execution.
- Resource Objects: Buffers and Textures (page 19) discusses the management of device memory, including buffer and texture objects that represent GPU memory allocations.
- Functions and Libraries (page 27) describes how Metal shading language code can be represented in a Metal app, and how Metal shading language code is loaded onto and executed by the GPU.
- Graphics Rendering: Render Command Encoder (page 32) describes how to render 3D graphics, including how to distribute graphics operations across multiple threads.
- Data-Parallel Compute Processing: Compute Command Encoder (page 60) explains how to perform data-parallel processing.
- Buffer and Texture Operations: Blit Command Encoder (page 66) describes how to copy data between textures and buffers.
- Metal Tips and Techniques (page 68) lists development tips, such as how to build libraries offline with compiled code.

Prerequisites

You should be familiar with the Objective-C language and 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.

In addition, several sample code projects using Metal are available in the Apple Developer Library.

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 needing to use separate APIs and shader languages.

The Metal framework provides the following:

- **Low-overhead interface.** 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 details on Metal command submission, see Command Organization and Execution Model (page 11).

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

  For details on Metal memory objects, see Resource Objects: Buffers and Textures (page 19).

- **Integrated support for both graphics and compute operations.** Metal uses the same data structures and resources (such as buffers, textures, and command queues) for both graphics and compute operations. In addition, the Metal shading language supports both graphics and compute functions. The Metal framework enables resourcs to be shared between the runtime interface, graphics shaders, and compute functions.

  For details on writing apps that use Metal for graphics rendering or data-parallel compute operations, see Graphics Rendering: Render Command Encoder (page 32) or Data-Parallel Compute Processing: Compute Command Encoder (page 60).

- **Precompiled shaders.** Metal shaders can be compiled during build time along with your app code and then loaded at runtime. This workflow provides better code generation as well as easier debugging of shader code. (Metal also supports runtime compilation of shader code.)

  For details on working with Metal shaders from your Metal framework code, see Functions and Libraries (page 27). For details on the Metal shading language itself, see *Metal Shading Language Guide* .

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

Command Organization and Execution Model

In the Metal architecture, the `MTLDevice` protocol defines the interface that represents a single GPU. The `MTLDevice` protocol supports methods for interrogating device properties, for creating other device-specific objects such as buffers and textures, and for encoding and queueing render and compute commands to be submitted to the GPU for execution.

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:

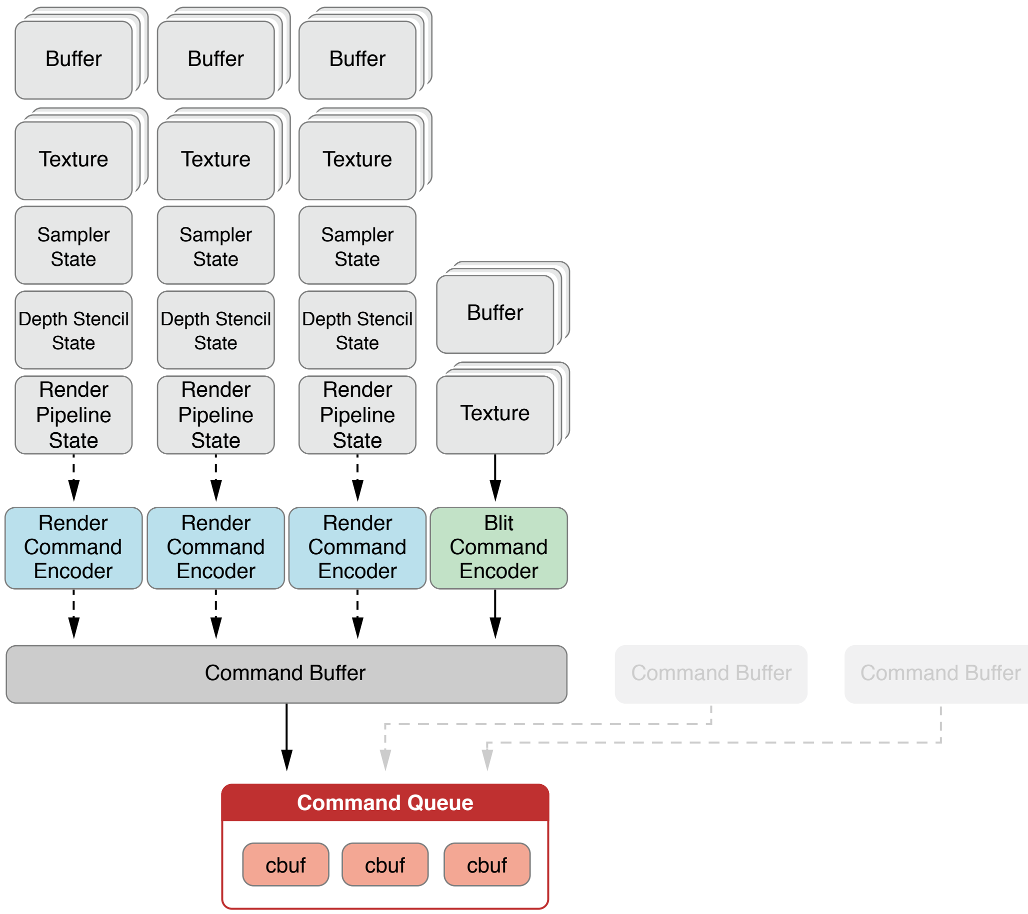
- The `MTLRenderCommandEncoder` protocol encodes graphics (3D) rendering commands for a single rendering pass.
- The `MTLComputeCommandEncoder` protocol encodes data-parallel computation workloads.
- The `MTLBlitCommandEncoder` protocol encodes simple copy operations between buffers and textures, as well as utility operations 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 can be created for use with the same command buffer. The one exception to the "one active command encoder for each command buffer" rule is the `MTLParallelRenderCommandEncoder` protocol, discussed in Encoding a Single Rendering Pass Using Multiple Threads (page 58).

Once all encoding is completed, you commit the `MTLCommandBuffer` object itself, 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 shows how the command queue, command buffer, and command encoder objects are closely related. Each column of components at the top of the diagram (buffer, texture, sampler, depth and stencil state, pipeline state) represent resources and state that are specific to a particular command encoder.

Figure 2-1 Metal Object Relationships



The Device Object Represents a 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 Nontransient Objects in Metal

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

Command buffer and command encoder objects are transient and designed for a single use. They are very inexpensive to allocate and deallocate, so their creation methods return autoreleased objects.

The following objects are not transient. Reuse these objects in performance sensitive code, and avoid creating them repeatedly.

- Command queues
- Data 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 the buffer is committed for execution by the GPU. A single command buffer can 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 is encoded into a single command buffer, even if rendering that frame involves multiple rendering passes, compute processing functions, or blit operations.

Command buffers are 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—through synchronous calls or handler blocks discussed in Registering Handler Blocks for Command Buffer Execution (page 15)—and to check the status of the command buffer execution.

Command buffers also represent the only independently trackable unit of work by the app, and they define the coherency boundaries established by the Metal memory model, as detailed in Resource Objects: Buffers and Textures (page 19).

Creating a Command Buffer

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

Command buffers created by the `commandBuffer` method retain 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`. Use the `commandBufferWithUnretainedReferences` method only 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 uses 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 are 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 17).

Registering Handler Blocks for Command Buffer Execution

The `MTLCommandBuffer` methods listed below monitor command execution. Scheduled and completed handlers are invoked in execution order on an undefined thread. Any code you execute in these handlers should complete quickly; if expensive or blocking work needs to be done, 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 is satisfied. You can register multiple scheduled handlers 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 completes the execution of the command buffer. You can register multiple completed handlers 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 special case of completed handler. This convenience method presents the contents of a displayable resource (a `CAMetalDrawable` object) when the command buffer is scheduled. For details about the `presentDrawable:` method, see Integration with Core Animation: CAMetalLayer (page 37).

Monitoring Command Buffer Execution Status

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 you use 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 finish encoding commands, call the `endEncoding` method. To write further commands, create a new command encoder.

Creating a Command Encoder Object

Becasuse a command encoder appends commands into a specific command buffer, you create a command encoder by requesting one from the `MTLCommandBuffer` object you want to use it with. Use the following `MTLCommandBuffer` methods to create command encoders of each type:

- The `renderCommandEncoderWithDescriptor:` method creates a `MTLRenderCommandEncoder` object for graphics rendering to an attachment in a `MTLRenderPassDescriptor`.
- The `computeCommandEncoder` method creates a `MTLComputeCommandEncoder` object for data-parallel computations.
- The `blitCommandEncoder` method creates a `MTLBlitCommandEncoder` object for memory operations.
- The `parallelRenderCommandEncoderWithDescriptor:` method creates a `MTLParallelRenderCommandEncoder` object that enables several `MTLRenderCommandEncoder` objects to run on different threads while still rendering to an attachment that is specified in a shared `MTLRenderPassDescriptor`.

Render Command Encoder

Graphics rendering can be described in terms of a *rendering pass*. A `MTLRenderCommandEncoder` object 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 33)) 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
- Draw 3D primitives

For detailed information about the `MTLRenderCommandEncoder` protocol, see Graphics Rendering: Render Command Encoder (page 32).

Compute Command Encoder

For data-parallel computing, the `MTLComputeCommandEncoder` protocol provides methods to encode commands in the command buffer that can specify the compute function and its arguments (for example, texture, buffer, and sampler state) and dispatch the compute function for execution. To create a compute command encoder object, use the `computeCommandEncoder` method of `MTLCommandBuffer`. For detailed information about the `MTLComputeCommandEncoder` methods and properties, see Data-Parallel Compute Processing: Compute Command Encoder (page 60).

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 66).

Multiple Threads, Command Buffers, and Command Encoders

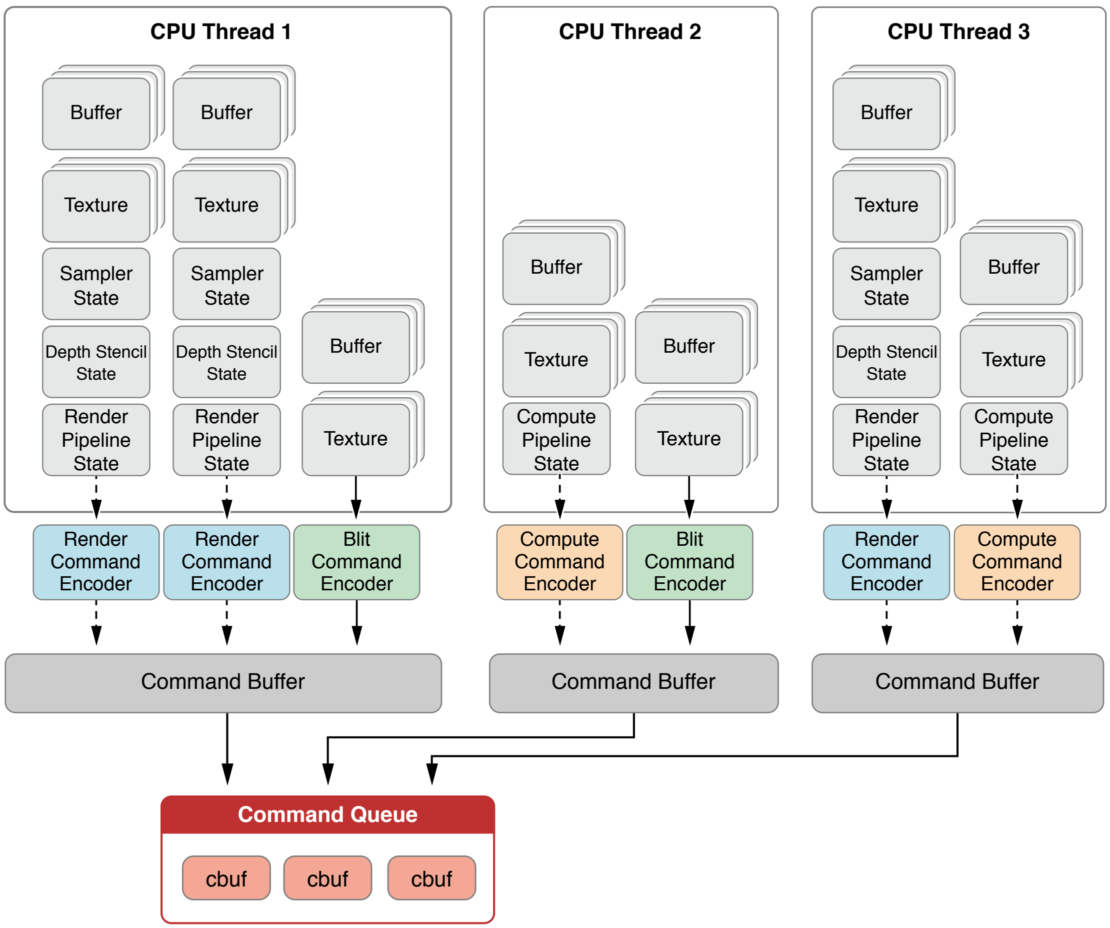
Most apps use a single thread to encode the rendering commands for a single frame in a single command buffer. At the end of each frame, you commit the command buffer, which both schedules and begins command execution.

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 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 also shows each command buffer receiving commands from different command encoders. When you finish 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 58).

Resource Objects: Buffers and Textures

SwiftObjective-C

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

- `MTLBuffer` represents an allocation of unformatted memory that can contain any type of data. Buffers are often used for vertex, shader, and compute state data.
- `MTLTexture` represents an allocation of formatted image data with a specified texture type and pixel format. Texture objects are used as source textures for vertex, fragment, or compute functions, as well as to store graphics rendering output (that is, as an attachment).

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

Buffers Are Typeless Allocations of Memory

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

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 20).

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 an attachment to be used as a rendering destination. A `MTLTexture` object can have *one* of the following structures:

- A 1D, 2D, or 3D image
- An array of 1D or 2D images
- A *cube* of six 2D images

`MTLPixelFormat` specifies the organization of individual pixels in a `MTLTexture` object. Pixel formats are discussed further in Pixel Formats for Textures (page 24).

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 the data was stored in the specified pixel format. The `MTLPixelFormat` of the new texture object must be *compatible* with the `MTLPixelFormat` of the original texture object. (See Pixel Formats for Textures (page 24) 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 can prevent the use of certain texturing optimizations, such as pixel swizzling or tiling.

Creating a Texture Object with a Texture Descriptor

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

To create one or more textures from a descriptor:

1. Create a custom `MTLTextureDescriptor` object that contains texture properties that describe the texture data:
    - The `textureType` property specifies a texture's dimensionality and arrangement (for example, 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.
2. Create a texture from the `MTLTextureDescriptor` object by calling the `newTextureWithDescriptor:` method of a `MTLDevice` object. After texture creation, call the `replaceRegion:mipmapLevel:slice:withBytes:bytesPerRow:bytesPerImage:` method to load the texture image data, as detailed in Copying Image Data to and from a Texture (page 23).
3. To create more `MTLTexture` objects, you can reuse the same `MTLTextureDescriptor` object, modifying the descriptor's property values as needed.

Listing 3-1 shows code for creating a texture descriptor `txDesc` and setting 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];
```

Working with Texture 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 23), `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.

Creating a Texture Descriptor with Convenience Methods

For common 2D and cube textures, use the following convenience methods to create a `MTLTextureDescriptor` object with several of its property values automatically set:

- The `texture2DDescriptorWithPixelFormat:width:height:mipmapped:` method creates a `MTLTextureDescriptor` object for a 2D texture. The `width` and `height` values 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 uses the `texture2DDescriptorWithPixelFormat:width:height:mipmapped:` method to create a descriptor object for a `64x64` 2D texture that is not mipmapped.

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, 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 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:MTLRegionMake2D(0,0,width,height)
    mipmapLevel:0 slice:0 withBytes:textureData
    bytesPerRow:myRowBytes bytesPerImage:myImageBytes];
```

Pixel Formats for Textures

`MTLPixelFormat` specifies the organization of color, depth, and 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, and so on. In contrast, for `MTLPixelFormatBGRA8Unorm`, the lowest addresses contains blue, the next addresses contain green, and so on.
- Packed formats combine multiple components into one 16-bit or 32-bit value, where the components are stored from the least to most significant bit (LSB to MSB). For example, `MTLPixelFormatRGB10A2Uint` is a 32-bit packed format that consists of three 10-bit channels (for R, G, and B) and two bits for alpha.
- Compressed formats are arranged in blocks of pixels, and the layout of each block is specific to that pixel format. Compressed pixel formats can only be used for 2D, 2D Array, or cube texture types. Compressed formats cannot be used to create 1D, 2DMultisample or 3D textures.

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

Several pixel formats store color components with sRGB color space values (for example, `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 attachment that uses a texture with an sRGB pixel format, the implementation converts the linear color values to sRGB, as follows:

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

For more information about pixel format for rendering, see Creating a Render Pass Descriptor (page 33).

Creating a Sampler States Object 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` object. A sampler descriptor defines the properties of a sampler state object. To create a sampler state object:

1. Call the `newSamplerStateWithDescriptor:` method of a `MTLDevice` object to create a `MTLSamplerDescriptor` object.
2. Set the desired values in the `MTLSamplerDescriptor` object, including filtering options, addressing modes, maximum anisotropy, and level-of-detail parameters.
3. Create a `MTLSamplerState` object from the sampler descriptor by calling the `newSamplerStateWithDescriptor:` method of the `MTLDevice` object that created the descriptor.

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

Listing 3-4 Creating a Sampler State 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 keep the following in mind when using the host CPU to access the storage for these resources.

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

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

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 functions with a `MTLLibrary` object.

MTLFunction Represents a Shader or Compute Function

A `MTLFunction` object represents a single function that is written in the Metal shading language and executed on the GPU as part of a graphics or compute pipeline. 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 specify a `MTLFunction` object for use as a vertex or fragment shader in a `MTLRenderPipelineDescriptor` object, as detailed in Creating a Render Pipeline State (page 39). For a compute pass, you specify a `MTLFunction` object when creating a `MTLComputePipelineState` object for a target device, as described in Specify a Compute State and Resources for a Compute Command Encoder (page 61).

A Library Is a Repository of Functions

A `MTLLibrary` object represents a repository of one or more `MTLFunction` objects. A single `MTLFunction` object 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 (`vertex`, `fragment`, or `kernel`) can be represented by a `MTLFunction` object in a library. A Metal function without one of these function qualifiers cannot be directly represented by a `MTLFunction` object, although it can called by another function within the shader.

The `MTLFunction` objects in a library can be created from either of these sources:

- Metal shading language code that was compiled into a binary *library* format during the app build process.
- A text string containing Metal shading language source code that is compiled by the app at runtime.

Creating a Library from Compiled Code

For the best 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` object 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 68).

Creating a Library from Source Code

To create a `MTLLibrary` from a string of Metal shading language source code that may contain several functions, call one of the following methods of `MTLDevice`. These methods compile the source code when the library 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 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 the `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

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

- `name`, a string with the name of the function.
- `functionType`, which indicates whether the function is declared as a vertex, fragment, or compute function.
- `vertexAttributes`, an array of `MTLVertexAttribute` objects that describe how vertex attribute data is organized in memory and how it is mapped to vertex function arguments. For more details, see Vertex Descriptor for Data Organization (page 47).

`MTLFunction` does not provide access to function arguments. A reflection object (either `MTLRenderPipelineReflection` or `MTLComputePipelineReflection`, depending upon the type of command encoder) that reveals details of shader or compute function arguments can be obtained during the creation of a pipeline state. For details on creating pipeline state and reflection objects, see Creating a Render Pipeline State (page 39) or Creating a Compute Pipeline State (page 60). Avoid obtaining reflection data if it will not be used.

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

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

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

Listing 4-2 Iteration Through Function Arguments

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

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

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

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

- If `type` is `MTLArgumentTypeTexture`, then the `textureType` property indicates the overall texture type (such as `texture1d_array`, `texture2d_ms`, and `texturecube` types in the shading language), and the `textureDataType` property indicates the component data type (such as `half`, `float`, `int`, or `uint`).
- If `type` is `MTLArgumentTypeThreadgroupMemory`, the `threadgroupMemoryAlignment` and `threadgroupMemoryDataSize` properties are relevant.
- If `type` is `MTLArgumentTypeBuffer`, the `bufferAlignment`, `bufferDataSize`, `bufferDataType`, and `bufferStructType` properties are relevant.

If the buffer argument is a struct (that is, `bufferDataType` is `MTLDataTypeStruct`), the `bufferStructType` property contains a `MTLStructType` that represents the struct, and `bufferDataSize` contains the size of the struct, in bytes. If the buffer argument is an array (or pointer to an array), then `bufferDataType` indicates the data type of an element, and `bufferDataSize` contains the size of one array element, in bytes.

Listing 4-3 drills down in a `MTLStructType`object to examine the details of struct members, each represented by a `MTLStructMember` object. A struct member may be a simple type or may be an array or a nested struct. If the member is a nested struct, then call the `structType` method of `MTLStructMember` to obtain a `MTLStructType` object that represents the struct and then recursively drill down to analyze it. If the member is an array, use the `arrayType` method of `MTLStructMember` to obtain a `MTLArrayType` that represents it. Then examine its `elementType` property of `MTLArrayType`. If `elementType` is `MTLDataTypeStruct`, call the `elementStructType` method to obtain the struct and continue to drill down into its members. If `elementType` is `MTLDataTypeArray`, call the `elementArrayType` method to obtain the subarray and analyze it further.

Listing 4-3 Processing a Struct Argument

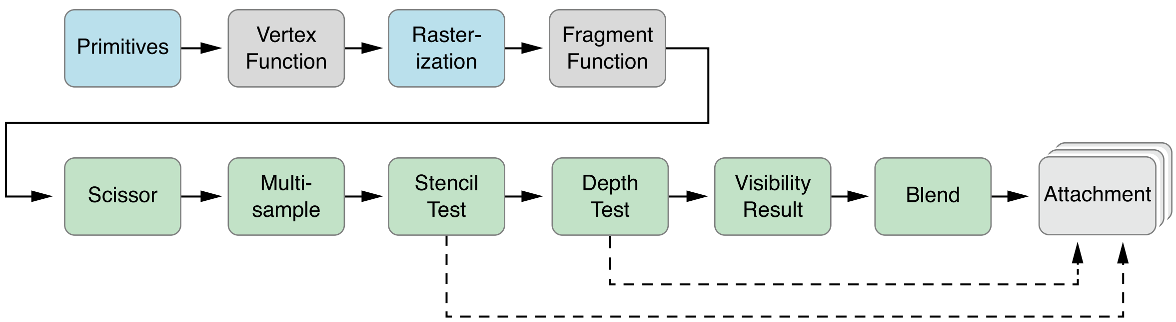
```
MTLStructType *structObj = [arg.bufferStructType];
for (MTLStructMember *member in structObj.members) {
    //  process each MTLStructMember
    if (member.dataType == MTLDataTypeStruct) {
       // obtain MTLStructType* with [member structType]
       // recursively drill down into the nested struct
    }
    else if (member.dataType == MTLDataTypeArray) {
       // obtain MTLArrayType* with [member arrayType]
       // examine the elementType and drill down, if necessary
    }
    else {
       // member is neither struct nor array
       // analyze it; no need to drill down further
    }
}
```

Graphics Rendering: Render Command Encoder

Objective-CSwift

This chapter describes how to create and work with `MTLRenderCommandEncoder` and `MTLParallelRenderCommandEncoder` objects, 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.

Figure 5-1 Metal Graphics Rendering Pipeline



A `MTLRenderCommandEncoder` object represents a single rendering command encoder. A `MTLParallelRenderCommandEncoder` object 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 58).

Creating and Using a Render Command Encoder

To create, initialize, and use a single render command encoder:

1. Create a `MTLRenderPassDescriptor` object to define a collection of attachments that serve as the rendering destination for the graphics commands in the command buffer for that rendering pass. Typically, you create a `MTLRenderPassDescriptor` object once and reuse it each time your app renders a frame. See Creating a Render Pass Descriptor (page 33).
2. Create a `MTLRenderCommandEncoder` object by calling the `renderCommandEncoderWithDescriptor:` method of `MTLCommandBuffer` with the specified render pass descriptor. See Using the Render Pass Descriptor to Create a Render Command Encoder (page 37).

3. 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 (page 39).
4. Set textures, buffers, and samplers to be used by the render command encoder, as described in Specifying Resources for a Render Command Encoder (page 45).
5. Call `MTLRenderCommandEncoder` methods to specify additional fixed-function state, including the depth and stencil state, as explained in Fixed-Function State Operations (page 50).
6. Finally, call `MTLRenderCommandEncoder` methods to draw graphics primitives, as described in Drawing Geometric Primitives (page 54).

Creating a Render Pass Descriptor

A `MTLRenderPassDescriptor` object represents the destination for the encoded rendering commands, which is a collection of attachments. The properties of a render pass descriptor 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. The `renderPassDescriptor` convenience method creates a `MTLRenderPassDescriptor` object with color, depth, and stencil attachment properties with default attachment state. The `visibilityResultBuffer` property specifies a buffer where the device can update to indicate whether any samples pass the depth and stencil tests—for details, see Fixed-Function State Operations (page 50).

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

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

The limitations on the size and number of attachments are described in Metal Capabilities and Limitations (page 69).

Load and Store Actions

The `loadAction` and `storeAction` properties of an attachment descriptor specify an action that is performed at either the start or end of a rendering pass. (For `MTLParallelRenderCommandEncoder`, the load and store actions 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 58).)

Possible `loadAction` values include:

- `MTLLoadActionClear`, which writes the same value to every pixel in the specified attachment descriptor. For more detail about this action, see Specifying the Clear Load Action (page 35).
- `MTLLoadActionLoad`, which preserves the existing contents of the texture.
- `MTLLoadActionDontCare`, which allows each pixel in the attachment to take on any value at the start of the rendering pass.

If your application will render all pixels of the attachment for a given frame, use the default load action `MTLLoadActionDontCare`. The `MTLLoadActionDontCare` action allows the GPU to avoid loading the existing contents of the texture, ensuring the best performance. Otherwise, you can use the `MTLLoadActionClear` action to clear the previous contents of the attachment, or the `MTLLoadActionLoad` action to preserve them. The `MTLLoadActionClear` action also avoids loading the existing texture contents, but it incurs the cost of filling the destination with a solid color.

Possible `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. For details, see Example: Creating a Render Pass Descriptor for Multisampled Rendering (page 36).
- `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.

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

Specifying the Clear Load Action

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

- For `MTLRenderPassColorAttachmentDescriptor`, `clearColor` contains a `MTLClearColor` value that consists of four double-precision floating-point RGBA components and is used to clear the color attachment. The `MTLClearColorMake` function creates a clear color value from red, green, blue, and alpha components. The default clear color is (0.0, 0.0, 0.0, 1.0), or opaque black.
- For `MTLRenderPassDepthAttachmentDescriptor`, `clearDepth` contains one double-precision floating-point clearing value in the range [0.0, 1.0] that is used to clear the depth attachment. The default value is 1.0.
- For `MTLRenderPassStencilAttachmentDescriptor`, `clearStencil` contains one 32-bit unsigned integer that is used to clear the stencil attachment. The default value is 0.

Example: Creating a Render Pass Descriptor with Load and Store Actions

Listing 5-1 creates a simple render pass descriptor with color and depth attachments. First, two texture objects are created, one with a color-renderable pixel format and the other with a depth pixel format. Next the `renderPassDescriptor` convenience method of `MTLRenderPassDescriptor` creates a default render pass descriptor. Then the color and depth attachments are accessed through the properties of `MTLRenderPassDescriptor`. The textures and actions are set in `colorAttachments[0]`, which represents the first color attachment (at index 0 in the array), and the depth attachment.

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

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

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

MTLRenderPassDescriptor *renderPassDesc = [MTLRenderPassDescriptor
renderPassDescriptor];
renderPassDesc.colorAttachments[0].texture = colorTex;
```

```
renderPassDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
renderPassDesc.colorAttachments[0].storeAction = MTLStoreActionStore;
renderPassDesc.colorAttachments[0].clearColor = MTLClearColorMake(0.0,1.0,0.0,1.0);

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

Example: Creating a Render Pass Descriptor for Multisampled Rendering

To use the `MTLStoreActionMultisampleResolve` action, you must set the `texture` property to a multisample-type texture, and the `resolveTexture` property will contain the result of the multisample resolve operation. (If `texture` does not support multisampling, then the result of a multisample resolve action 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, 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 *renderPassDesc = [MTLRenderPassDescriptor
renderPassDescriptor];
renderPassDesc.colorAttachments[0].texture = msaaTex;
```

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

Using the Render Pass Descriptor to Create a Render Command Encoder

After you create a render pass descriptor and specify its properties, use the the `renderCommandEncoderWithDescriptor:` method of a `MTLCommandBuffer` object to create a render command encoder, as shown in Listing 5-3.

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

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

Displaying Rendered Content with Core Animation

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

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

Typically, you use the `layerClass` method to designate `CAMetalLayer` as the backing layer type for your own custom UIView subclass, as shown in Listing 5-4. Otherwise, you can create a `CAMetalLayer` with its `init` method and include the layer in an existing view.

Listing 5-4 Using CAMetalLayer as the backing layer for a UIView subclass

```
+ (id) layerClass {
    return [CAMetalLayer class];
}
```

To display content rendered by Metal in the layer, you must obtain a displayable resource (a `CAMetalDrawable` object) from the the `CAMetalLayer` object and then use that resource as a color attachment in your rendering pipeline. To do this, you first set properties of the `CAMetalLayer` object that describe the drawable resources it vends, then call its `nextDrawable` method each time you begin rendering a new frame. If the `CAMetalLayer` properties are not set, the `nextDrawable` method call fails. The following `CAMetalLayer` properties describe the drawable object:

- The `device` property declares the `MTLDevice` object that the resource is created from.
- The `pixelFormat` property declares the pixel format of the texture. The supported values are `MTLPixelFormatBGRA8Unorm` (the default) and `MTLPixelFormatBGRA8Unorm_sRGB`.
- The `drawableSize` property declares the dimensions of the texture in device pixels. To ensure that your app renders content at the precise dimensions of the display (without requiring an additional sampling stage on some devices), take the target screen's `nativeScale` or `nativeBounds` property into account when calculating the desired size for your layer.
- The `framebufferOnly` property declares whether the texture can be used only as an attachment (`YES`) or whether it can also be used for texture sampling and pixel read/write operations (`NO`). If `YES`, the layer object can optimize the texture for display. For most apps, the recommended value is `YES`.
- The `presentsWithTransaction` property declares whether 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 `nextDrawable` method succeeds, it returns a `CAMetalDrawable` object with the following read-only properties:

- The `texture` property holds the texture object. You use this as an attachment when creating your rendering pipeline (`MTLRenderPipelineColorAttachmentDescriptor` object).
- The `layer` property points to the `CAMetalLayer` object that responsible for displaying the drawable.

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

To display the contents of a drawable object after rendering is complete, you must submit it to Core Animation by calling the drawable object's `present` method. To synchronize presentation of a drawable with completion of the command buffer responsible for its rendering, you can call either the `presentDrawable:` or `presentDrawable:atTime:` convenience method on a`MTLCommandBuffer` object. These methods use the scheduled handler (see Registering Handler Blocks for Command Buffer Execution (page 15)) to call the drawable's `present` method, which covers most scenarios. The `presentDrawable:atTime:` method provides further control over when the drawable is presented.

Creating a Render Pipeline State

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.

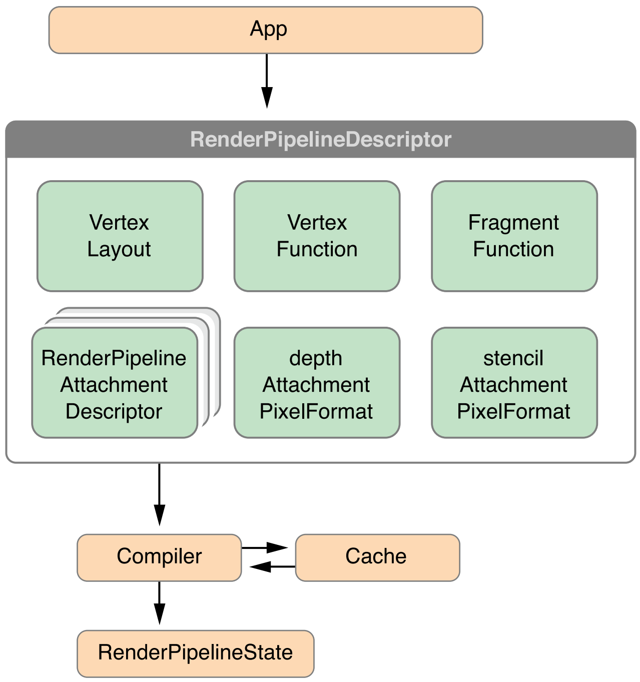
A render pipeline state is an immutable object. To create a render pipeline state, you first create and configure a mutable `MTLRenderPipelineDescriptor` object that describes the attributes of a render pipeline state. Then, you use the descriptor to create a `MTLRenderPipelineState` object.

Creating and Configuring a Render Pipeline Descriptor

To create a render pipeline state, first create a `MTLRenderPipelineDescriptor` object, which has properties that describe the graphics rendering pipeline state you want to use during the rendering pass, as depicted in Figure 5-2. The `colorAttachments` property of the new `MTLRenderPipelineDescriptor` object contains an array of `MTLRenderPipelineColorAttachmentDescriptor` objects, and each descriptor represents a color attachment state that specifies the blend operations and factors for that attachment, as detailed in

Configuring Blending in a Render Pipeline Attachment Descriptor (page 43). The attachment descriptor also specifies the pixel format of the attachment, which must match the pixel format for the texture of the render pipeline descriptor with the corresponding attachment index, or an error occurs.

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



Set these properties for the `MTLRenderPipelineDescriptor` object:

- Set the `depthAttachmentPixelFormat` property to match the pixel format for the texture of `depthAttachment` in `MTLRenderPassDescriptor`.
- Set the `stencilAttachmentPixelFormat` property to match the pixel format for the texture of `stencilAttachment` in `MTLRenderPassDescriptor`.
- To specify the vertex or fragment shader in the render pipeline state, set the `vertexFunction` or `fragmentFunction` property, respectively. Setting `fragmentFunction` to `nil` disables the rasterization of pixels into the specified color attachment, which is typically used for depth-only rendering or for outputting data into a buffer object from the vertex shader.
- If the vertex shader has an argument with per-vertex input attributes, set the `vertexDescriptor` property to describe the organization of the vertex data in that argument, as described in Vertex Descriptor for Data Organization (page 47).
- The default value of `YES` for the `rasterizationEnabled` property is sufficient for most typical rendering tasks. To use only the vertex stage of the graphics pipeline (for example, to gather data transformed in a vertex shader), set this property to `NO`.

- If the attachment supports multisampling (that is, the attachment is a `MTLTextureType2DMultisample` type texture), then multiple samples can be created per pixel. To determine how fragments combine to provide pixel coverage, use the following `MTLRenderPipelineDescriptor` properties.
  - The `sampleCount` property determines the number of samples for each pixel. When `MTLRenderCommandEncoder` is created, the `sampleCount` for the textures for all attachments must match this `sampleCount` property. If the attachment cannot support multisampling, then `sampleCount` is 1, which is also the default value.
  - If `alphaToCoverageEnabled` is set to `YES`, then the alpha channel fragment output for `colorAttachments[0]` is read and used to determine a coverage mask.
  - If `alphaToOneEnabled` is set to `YES`, then alpha channel fragment values for `colorAttachments[0]` are forced to 1.0, which is the largest representable value. (Other attachments are unaffected.)

Creating a Render Pipeline State from a Descriptor

After creating a render pipeline descriptor and specifying its properties, use it to create the `MTLRenderPipelineState` object. Because creating a render pipeline state can require an expensive evaluation of graphics state and a possible compilation of the specified graphics shaders, you can use either a blocking or an asynchronous method to schedule such work in a way that best fits the design of your app.

- To synchronously create the render pipeline state object, call either the `newRenderPipelineStateWithDescriptor:error:` or `newRenderPipelineStateWithDescriptor:options:reflection:error:` method of a `MTLDevice` object. These methods block the current thread while Metal evaluates the descriptor's graphics state information and compiles shader code to create the pipeline state object.
- To asynchronously create the render pipeline state object, call either the `newRenderPipelineStateWithDescriptor:completionHandler:` or `newRenderPipelineStateWithDescriptor:options:completionHandler:` method of a `MTLDevice` object. These methods return immediately—Metal asynchronously evaluates the descriptor's graphics state information and compiles shader code to create the pipeline state object, then calls your completion handler to provide the new `MTLRenderPipelineState` object.

When you create a `MTLRenderPipelineState` object you can also choose to create reflection data that reveals details of the pipeline's shader function and its arguments. The `newRenderPipelineStateWithDescriptor:options:reflection:error:` and `newRenderPipelineStateWithDescriptor:options:completionHandler:` methods provide this data. Avoid obtaining reflection data if it will not be used. For more information on how to analyze reflection data, see Determining Function Details at Runtime (page 29).

After you create a `MTLRenderPipelineState` object, call the `setRenderPipelineState:` method of `MTLRenderCommandEncoder` to associate the render pipeline state with the command encoder for use in rendering.

Listing 5-5 demonstrates the creation of a render pipeline state object called `pipeline`.

Listing 5-5 Creating a Simple Pipeline State

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

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

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

The variables `vertFunc` and `fragFunc` are shader functions that are specified as properties of the render pipeline state descriptor called `renderPipelineDesc`. Calling the `newRenderPipelineStateWithDescriptor:error:` method of the `MTLDevice` object synchronously uses the pipeline state descriptor to create the render pipeline state object. Calling the `setRenderPipelineState:` method of `MTLRenderCommandEncoder` specifies the `MTLRenderPipelineState` object to use with the render command encoder.

Note: Because a `MTLRenderPipelineState` object is expensive to create, you should reuse it whenever you want to use the same graphics state.

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 attachment (destination). Blend operations determine how the source and destination values are combined with blend factors.

To configure blending for a color attachment, set the following `MTLRenderPipelineColorAttachmentDescriptor` properties:

- To enable blending, set `blendingEnabled` to `YES`. Blending is disabled, by default.
- `writeMask` identifies which color channels are blended. The default value `MTLColorWriteMaskAll` allows all color channels to be blended.
- `rgbBlendOperation` and `alphaBlendOperation` separately assign the blend operations for the RGB and Alpha fragment data with a `MTLBlendOperation` value. The default value for both properties is `MTLBlendOperationAdd`.
- `sourceRGBBlendFactor`, `sourceAlphaBlendFactor`, `destinationRGBBlendFactor`, and `destinationAlphaBlendFactor` assign the source and destination blend factors.

Understanding Blending Factors and Operations

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 50).

Some blend operations combine the fragment values by multiplying the source values by a source `MTLBlendFactor` value (abbreviated SBF), multiplying the destination values by a destination blend factor (DBF), and combining the results using the arithmetic indicated by the `MTLBlendOperation` value. (If the blend operation 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`)

In the default blend behavior, the source completely overwrites the destination. This behavior is equivalent to setting 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))

Using a Custom Blending Configuration

Listing 5-6 shows code for a custom blending configuration, using the blend operation `MTLBlendOperationAdd`, the source blend factor `MTLBlendFactorOne`, and the destination blend factor `MTLBlendFactorOneMinusSourceAlpha`. `colorAttachments[0]` is a `MTLRenderPipelineColorAttachmentDescriptor` object with properties that specify the blending configuration.

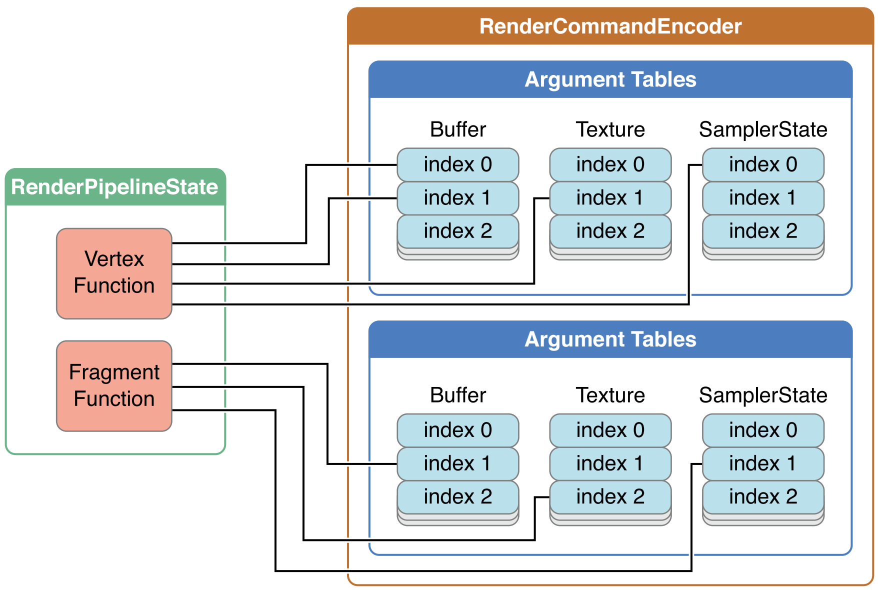
Listing 5-6 Specifying a Custom Blending Configuration

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

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

Specifying Resources for a Render Command Encoder

The `MTLRenderCommandEncoder` methods discussed in this section 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.

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 5-7, two buffers (`posBuf` and `texCoordBuf`) with indices 0 and 1, respectively, are defined for the vertex shader.

Listing 5-7 Metal Framework: Specifying Resources for a Vertex Function

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

In Listing 5-8, the function signature has corresponding arguments with the attribute qualifiers `buffer(0)` and `buffer(1)`.

Listing 5-8 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 5-9, a buffer, a texture, and a sampler (`fragmentColorBuf`, `shadeTex`, and `sampler`, respectively), all with index 0, are defined for the fragment shader.

Listing 5-9 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 5-10, the function signature has corresponding arguments with the attribute qualifiers `buffer(0)`, `texture(0)`, and `sampler(0)`, respectively.

Listing 5-10 Metal Shading Language: Fragment Function Arguments Match the Framework Argument Table Indices

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

Vertex Descriptor for Data Organization

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

In Metal shading language code, per-vertex inputs (such as scalars or vectors of integer or floating-point values) can be organized in one struct, which can be passed in one argument that is declared with the `[[ stage_in ]]` attribute qualifier, as seen in the `VertexInput` struct for the example vertex function `vertexMath` in Listing 5-11. Each field of the per-vertex input struct has the `[[ attribute(index) ]]` qualifier, which specifies the index in the vertex attribute argument table.

Listing 5-11 Metal Shading Language: Vertex Function Inputs with Attribute Indices

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

struct VertexOutput {
    float4 pos [[ position ]];
    float4 color;
};

vertex VertexOutput vertexMath(VertexInput in [[ stage_in ]])
{
  VertexOutput out;
  out.pos = float4(in.position.x, in.position.y, 0.0, 1.0);

  float sum1 = in.uv1.x + in.uv2.x;
  float sum2 = in.uv1.y + in.uv2.y;
  out.color = in.color + float4(sum1, sum2, 0.0f, 0.0f);
  return out;
}
```

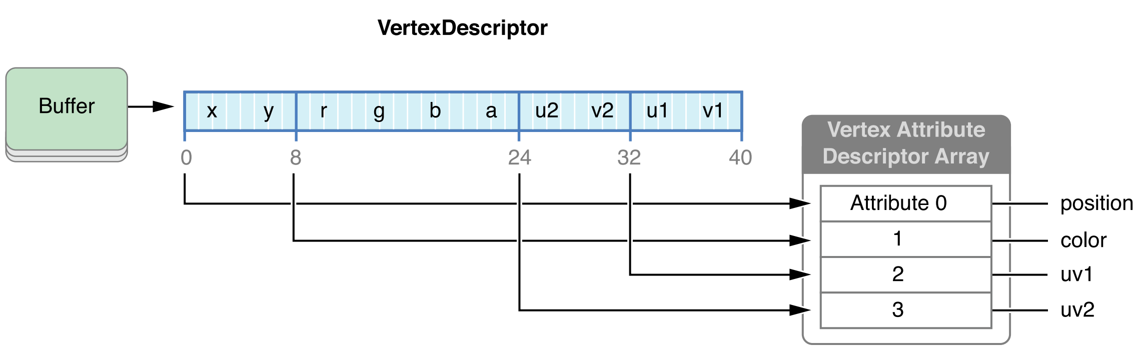
To refer to the shader function input from Metal framework code, describe a `MTLVertexDescriptor` object and then set it as the `vertexDescriptor` property of `MTLRenderPipelineState`. `MTLVertexDescriptor` has two properties: `attributes` and `layouts`.

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

- `bufferIndex`, which is an index to the buffer argument table that specifies which `MTLBuffer` is accessed. The buffer argument table is discussed in Specifying Resources for a Render Command Encoder (page 45).
- `format`, which specifies how the data should be interpreted in the framework code. If the data type is not an exact type match, it may be converted or expanded. For example, if the shading language type is `half4` and the framework `format` is `MTLVertexFormatFloat2`, then when the data is used as an argument to the vertex function, it may be converted from float to half and expanded from two to four elements (with 0.0, 1.0 in the last two elements).
- `offset`, which specifies where the data can be found from the start of a vertex.

Figure 5-4 illustrates a `MTLVertexAttributeDescriptorArray` in Metal framework code that implements an interleaved buffer that corresponds to the input to the vertex function `vertexMath` in the shading language code in Listing 5-11 (page 47).

Figure 5-4 Buffer Organization with Vertex Attribute Descriptors



Listing 5-12 shows the Metal framework code that corresponds to the interleaved buffer shown in Figure 5-4.

Listing 5-12 Metal Framework: Using a Vertex Descriptor to Access Interleaved Data

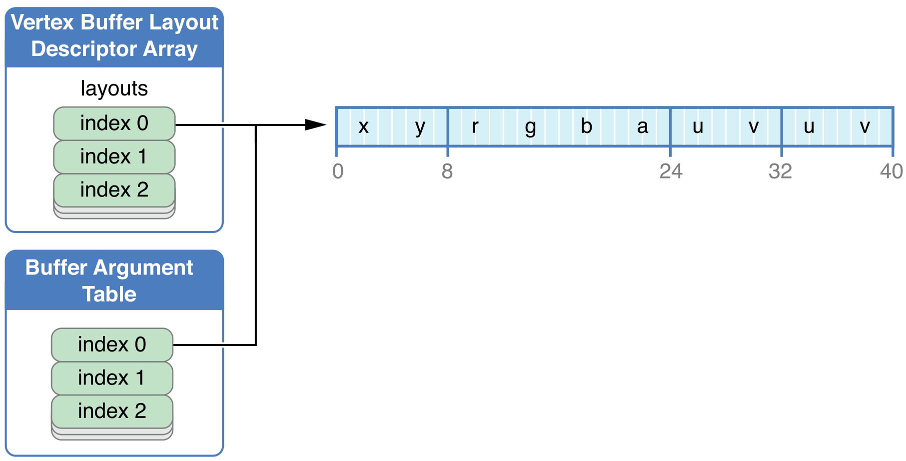
```
id <MTLFunction> vertexFunc = [library newFunctionWithName:@"vertexMath"];

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

vertexDesc.attributes[0].format = MTLVertexFormatFloat2;
vertexDesc.attributes[0].bufferIndex = 0;
vertexDesc.attributes[0].offset = 0;
vertexDesc.attributes[1].format = MTLVertexFormatFloat4;
vertexDesc.attributes[1].bufferIndex = 0;
vertexDesc.attributes[1].offset = 2 * sizeof(float);  // 8 bytes
vertexDesc.attributes[2].format = MTLVertexFormatFloat2;
vertexDesc.attributes[2].bufferIndex = 0;
vertexDesc.attributes[2].offset = 8 * sizeof(float);  // 32 bytes
vertexDesc.attributes[3].format = MTLVertexFormatFloat2;
vertexDesc.attributes[3].bufferIndex = 0;
vertexDesc.attributes[3].offset = 6 * sizeof(float);  // 24 bytes
vertexDesc.layouts[0].stride = 10 * sizeof(float);    // 40 bytes
vertexDesc.layouts[0].stepFunction = MTLVertexStepFunctionPerVertex;

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

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

The `layouts` property of `MTLVertexDescriptor` is a `MTLVertexBufferLayoutDescriptorArray`. For each `MTLVertexBufferLayoutDescriptor` in `layouts`, the properties specify how vertex and attribute data are fetched from the corresponding `MTLBuffer` in the argument table when Metal draws primitives. (For more on drawing primitives, see Drawing Geometric Primitives (page 54).) The `stepFunction` property of `MTLVertexBufferLayoutDescriptor` determines whether to fetch attribute data for every vertex, for some number of instances, or just once. If `stepFunction` is set to fetch attribute data for some number of instances, then the `stepRate` property of `MTLVertexBufferLayoutDescriptor` determines how many instances. The `stride` property specifies the distance between the data of two vertices, in bytes.

Figure 5-5 depicts the `MTLVertexBufferLayoutDescriptor` that corresponds to the code in Listing 5-12 (page 49). `layouts[0]` specifies how vertex data is fetched from corresponding index 0 in the buffer argument table. `layouts[0].stride` specifies a distance of 40 bytes between the data of two vertices. The value of `layouts[0].stepFunction`, `MTLVertexStepFunctionPerVertex`, specifies that attribute data is fetched for every vertex when drawing. If the value of `stepFunction` is `MTLVertexStepFunctionPerInstance`, the `stepRate` property determines how often attribute data is fetched. For example, if `stepRate` is 1, data is fetched for every instance; if `stepRate` is 2, for every two instances, and so on.

Figure 5-5 Buffer Organization with Vertex Buffer Layout Descriptors



Performing Fixed-Function Render Command Encoder Operations

Use these `MTLRenderCommandEncoder` methods to 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; for details, see Working with Viewport and Pixel Coordinate Systems (page 51).
- `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. You can use culling 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 counterclockwise order.)

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

Use the following `MTLRenderCommandEncoder` methods to encode fixed-function state change commands:

- `setScissorRect:` specifies a 2D scissor rectangle. Fragments that lie outside the specified scissor rectangle are discarded.
- `setDepthStencilState:` sets the depth and stencil test state as described in Depth and Stencil States (page 52).
- `setStencilReferenceValue:` specifies the stencil reference value.
- `setDepthBias:slopeScale:clamp:` specifies an adjustment for comparing shadow maps to the depth values output from fragment shaders.
- `setVisibilityResultMode:offset:` determines whether to monitor if any samples pass the depth and stencil tests. If set to `MTLVisibilityResultModeBoolean`, then if any samples pass the depth and stencil tests, a non-zero value is written to a buffer specified by the `visibilityResultBuffer` property of `MTLRenderPassDescriptor`, as described in Creating a Render Pass Descriptor (page 33).

  You can use this mode to perform occlusion testing. If you draw a bounding box and no samples pass, then you may conclude that any objects within that bounding box are occluded and thus do not require rendering.
- `setBlendColorRed:green:blue:alpha:` specifies the constant blend color and alpha values, as detailed in Configuring Blending in a Render Pipeline Attachment Descriptor (page 43).

Working with Viewport and Pixel Coordinate Systems

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

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

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

Performing Depth and Stencil Operations

The depth and stencil operations are fragment operations that you specify as follows:

1. Specify a custom `MTLDepthStencilDescriptor` object that contains settings for the depth/stencil state. Creating a custom `MTLDepthStencilDescriptor` object may require creating one or two `MTLStencilDescriptor` objects that are applicable to front-facing primitives and back-facing primitives.
2. Create a `MTLDepthStencilState` object by calling the `newDepthStencilStateWithDescriptor:` method of `MTLDevice` with a depth/stencil state descriptor.
3. To set the depth/stencil state, call the `setDepthStencilState:` method of `MTLRenderCommandEncoder` with the `MTLDepthStencilState`.
4. If the stencil test is in use, call `setStencilReferenceValue:` to specify the stencil reference value.

If the depth test is enabled, the render pipeline state must include a depth attachment to support writing the depth value. To perform the stencil test, the render pipeline state must include a stencil attachment. To configure attachments, see Creating and Configuring a Render Pipeline Descriptor (page 39).

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`.

Use the properties of a `MTLDepthStencilDescriptor` object as follows to set the depth and stencil state:

- To enable writing the depth value to the depth attachment, set `depthWriteEnabled` to `YES`.
- `depthCompareFunction` specifies how the depth test is performed. If a fragment's depth value fails the depth test, the fragment is discarded. For example, the commonly used `MTLCompareFunctionLess` function causes fragment values that are further away from the viewer than the (previously written) pixel depth value to fail the depth test; that is, the fragment is considered occluded by the earlier depth value.
- The `frontFaceStencil` and `backFaceStencil` properties each specify a separate `MTLStencilDescriptor` object 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 descriptor is configured for a valid stencil operation.

Listing 5-13 shows an example of creation and use of a `MTLDepthStencilDescriptor` object for the creation of a `MTLDepthStencilState` object, which is then used with a render command encoder. 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 5-13 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`:

- `readMask` is a bitmask; the GPU computes the bitwise AND of this mask with 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.
- `stencilCompareFunction` specifies how the stencil test is performed for fragments. In Listing 5-13, 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.

- `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 the preceding 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 a number of instances (`instanceCount`) of primitives using an index list specified in the `MTLBuffer` object `indexBuffer`. `indexCount` determines the number of indices. The index list starts at the index that is `indexBufferOffset` byte offset within the data in `indexBuffer`. `indexBufferOffset` must be a multiple of the size of an index, which is determined by `indexType`.
- `drawIndexedPrimitives:indexCount:indexType:indexBuffer:indexBufferOffset:` is similar to the previous method with an `instanceCount` of 1.

For every primitive rendering method listed above, 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 whether the triangles are rendered as filled or wireframe, and the `setCullMode:` and `setFrontFacingWinding:` settings determine whether the GPU culls triangles during rendering. For more information, see Fixed-Function State Operations (page 50)).

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 *Metal Shading Language Guide*.

Ending a Rendering Pass

To terminate a rendering pass, call `endEncoding` on the render command encoder. 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

The following steps, illustrated in Listing 5-14 (page 56), describe a basic procedure for rendering a triangle.

1. Create a `MTLCommandQueue` and use it to create a `MTLCommandBuffer`.
2. Create a `MTLRenderPassDescriptor` that specifies a collection of attachments that serve as the destination for encoded rendering commands in the command buffer.

   In this example, only the first color attachment is set up and used. (The variable `currentTexture` is assumed to contain a `MTLTexture` that is used for a color attachment.) Then the `MTLRenderPassDescriptor` is used to create a new `MTLRenderCommandEncoder`.
3. Create two `MTLBuffer` objects, `posBuf` and `colBuf`, and call `newBufferWithBytes:length:options:` to copy vertex coordinate and vertex color data, `posData` and `colData`, respectively, into the buffer storage.
4. Call the `setVertexBuffer:offset:atIndex:` method of `MTLRenderCommandEncoder` twice to specify the coordinates and colors.

   The `atIndex` input value of the `setVertexBuffer:offset:atIndex:` method corresponds to the attribute `buffer(atIndex)` in the source code of the vertex function.
5. Create a `MTLRenderPipelineDescriptor` and establish the vertex and fragment functions in the pipeline descriptor:
   - Create a `MTLLibrary` with source code from `progSrc`, which is assumed to be a string that contains Metal shader source code.
   - Then call the `newFunctionWithName:` method of `MTLLibrary` to create the `MTLFunction vertFunc` that represents the function called `hello_vertex` and to create the `MTLFunction fragFunc` that represents the function called `hello_fragment`.
   - Finally, set the `vertexFunction` and `fragmentFunction` properties of the `MTLRenderPipelineDescriptor` with these `MTLFunction` objects.

 |

6. Create a `MTLRenderPipelineState` from the `MTLRenderPipelineDescriptor` by calling `newRenderPipelineStateWithDescriptor:error:` or a similar method of `MTLDevice`. Then the `setRenderPipelineState:` method of `MTLRenderCommandEncoder` uses the created pipeline state for rendering.
7. Call the `drawPrimitives:vertexStart:vertexCount:` method of `MTLRenderCommandEncoder` to append commands to perform the rendering of a filled triangle (type `MTLPrimitiveTypeTriangle`).
8. Call the `endEncoding` method to end encoding for this rendering pass. And call the `commit` method of `MTLCommandBuffer` to execute the commands on the device.

Listing 5-14 Metal Code for Drawing a Triangle

```
id <MTLDevice> device = MTLCreateSystemDefaultDevice();

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

MTLRenderPassDescriptor *renderPassDesc
                              = [MTLRenderPassDescriptor renderPassDescriptor];
renderPassDesc.colorAttachments[0].texture = currentTexture;
renderPassDesc.colorAttachments[0].loadAction = MTLLoadActionClear;
renderPassDesc.colorAttachments[0].clearColor = MTLClearColorMake(0.0,1.0,1.0,1.0);
id <MTLRenderCommandEncoder> renderEncoder =
           [commandBuffer renderCommandEncoderWithDescriptor:renderPassDesc];

static const float posData[] = {
        0.0f, 0.33f, 0.0f, 1.f,
        -0.33f, -0.33f, 0.0f, 1.f,
        0.33f, -0.33f, 0.0f, 1.f,
};
static const float colData[] = {
        1.f, 0.f, 0.f, 1.f,
        0.f, 1.f, 0.f, 1.f,
        0.f, 0.f, 1.f, 1.f,
};
id <MTLBuffer> posBuf = [device newBufferWithBytes:posData
        length:sizeof(posData) options:nil];
```

```
id <MTLBuffer> colBuf = [device newBufferWithBytes:colorData
        length:sizeof(colData) options:nil];
[renderEncoder setVertexBuffer:posBuf offset:0 atIndex:0];
[renderEncoder setVertexBuffer:colBuf offset:0 atIndex:1];

NSError *errors;
id <MTLLibrary> library = [device newLibraryWithSource:progSrc options:nil
                           error:&errors];
id <MTLFunction> vertFunc = [library newFunctionWithName:@"hello_vertex"];
id <MTLFunction> fragFunc = [library newFunctionWithName:@"hello_fragment"];
MTLRenderPipelineDescriptor *renderPipelineDesc
                                   = [[MTLRenderPipelineDescriptor alloc] init];
renderPipelineDesc.vertexFunction = vertFunc;
renderPipelineDesc.fragmentFunction = fragFunc;
renderPipelineDesc.colorAttachments[0].pixelFormat = currentTexture.pixelFormat;
id <MTLRenderPipelineState> pipeline = [device
           newRenderPipelineStateWithDescriptor:renderPipelineDesc error:&errors];
[renderEncoder setRenderPipelineState:pipeline];
[renderEncoder drawPrimitives:MTLPrimitiveTypeTriangle
               vertexStart:0 vertexCount:3];
[renderEncoder endEncoding];
[commandBuffer commit];
```

In Listing 5-14, a `MTLFunction` object represents the shader function called `hello_vertex`. The `setVertexBuffer:offset:atIndex:` method of `MTLRenderCommandEncoder` is used to specify the vertex resources (in this case, two buffer objects) that are passed as arguments into `hello_vertex`. The `atIndex` input value of the `setVertexBuffer:offset:atIndex:` method corresponds to the attribute `buffer(atIndex)` in the source code of the vertex function, as shown in Listing 5-15.

Listing 5-15 Corresponding Shader 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, your app's performance can be limited by the single-CPU workload of encoding commands for a single rendering pass. However, attempting to circumvent this bottleneck by separating the workload into multiple rendering passes encoded on multiple CPU threads can also adversely impact performance, because each rendering pass requires its own intermediate attachment store and load actions to preserve the render target contents.

Instead, use a `MTLParallelRenderCommandEncoder` object, which manages multiple subordinate `MTLRenderCommandEncoder` objects that share the same command buffer and render pass descriptor. The parallel render command encoder ensures that the attachment load and store actions occur only at the start and end of the entire rendering pass, not at the start and end of each subordinate render command encoder's set of commands. With this architecture, you can assign each `MTLRenderCommandEncoder` object to its own thread in parallel in a safe and highly performant manner.

To create a parallel render command encoder, use the `parallelRenderCommandEncoderWithDescriptor:` method of a `MTLCommandBuffer` object. To create subordinate render command encoders, call the `renderCommandEncoder` method of the `MTLParallelRenderCommandEncoder` object once for each CPU thread from which you want to perform command encoding. All subordinate command encoders created from the same parallel render command encoder encode 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 encoding on all render command encoders created by the parallel render command encoder, call the `endEncoding` method of `MTLParallelRenderCommandEncoder` to end the rendering pass.

Listing 5-16 shows the `MTLParallelRenderCommandEncoder` creating three `MTLRenderCommandEncoder` objects: `rCE1`, `rCE2`, and `rCE3`.

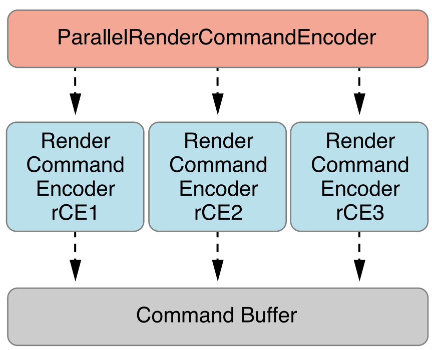
Listing 5-16 A Parallel Rendering Encoder with Three Render Command Encoders

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

id <MTLParallelRenderCommandEncoder> parallelRCE = [commandBuffer
```

```
                     parallelRenderCommandEncoderWithDescriptor:renderPassDesc];
id <MTLRenderCommandEncoder> rCE1 = [parallelRCE renderCommandEncoder];
id <MTLRenderCommandEncoder> rCE2 = [parallelRCE renderCommandEncoder];
id <MTLRenderCommandEncoder> rCE3 = [parallelRCE renderCommandEncoder];

//  not shown: rCE1, rCE2, and rCE3 call methods to encode graphics commands
//
//  rCE1 commands are processed first, because it was created first
//  even though rCE2 and rCE3 end earlier than rCE1
[rCE2 endEncoding];
[rCE3 endEncoding];
[rCE1 endEncoding];

//  all MTLRenderCommandEncoders must end before MTLParallelRenderCommandEncoder
[parallelRCE endEncoding];
```

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

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



Data-Parallel Compute Processing: Compute Command Encoder

This chapter explains how to create and use a `MTLComputeCommandEncoder` object to encode data-parallel compute processing state and commands and submit them for execution on a device.

To perform a data-parallel computation, follow these main steps:

1. 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 60). The `MTLFunction` object represents a compute function written with the Metal shading language, as described in Functions and Libraries (page 27).
2. Specify the `MTLComputePipelineState` object to be used by the compute command encoder, as discussed in Specifying a Compute State and Resources for a Compute Command Encoder (page 61).
3. 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 Specifying a Compute State and Resources for a Compute Command Encoder (page 61). Also set their argument table indices, so that Metal framework code can locate a corresponding resource in the shader code. At any given moment, the `MTLComputeCommandEncoder` can be associated to a number of resource objects.
4. Dispatch the compute function a specified number of times, as explained in Executing a Compute Command (page 62).

Creating a Compute Pipeline State

A `MTLFunction` object represents data-parallel code that can be executed by a `MTLComputePipelineState` object. The `MTLComputeCommandEncoder` object encodes commands that set arguments and execute the compute function. Because creating a compute pipeline state can require an expensive compilation of Metal shading language code, you can use either a blocking or an asynchronous method to schedule such work in a way that best fits the design of your app.

- To synchronously create the compute pipeline state object, call either the `newComputePipelineStateWithFunction:error:` or `newComputePipelineStateWithFunction:options:reflection:error:` method of `MTLDevice`. These methods block the current thread while Metal compiles shader code to create the pipeline state object.

- To asynchronously create the compute pipeline state object, call either the `newComputePipelineStateWithFunction:completionHandler:` or `newComputePipelineStateWithFunction:options:completionHandler:` method of `MTLDevice`. These methods return immediately—Metal asynchronously compiles shader code to create the pipeline state object, then calls your completion handler to provide the new `MTLComputePipelineState` object.

When you create a `MTLComputePipelineState` object you can also choose to create reflection data that reveals details of the compute function and its arguments. The `newComputePipelineStateWithFunction:options:reflection:error:` and `newComputePipelineStateWithFunction:options:completionHandler:` methods provide this data. Avoid obtaining reflection data if it will not be used. For more information on how to analyze reflection data, see Determining Function Details at Runtime (page 29).

Specifying a Compute State and Resources for a Compute Command Encoder

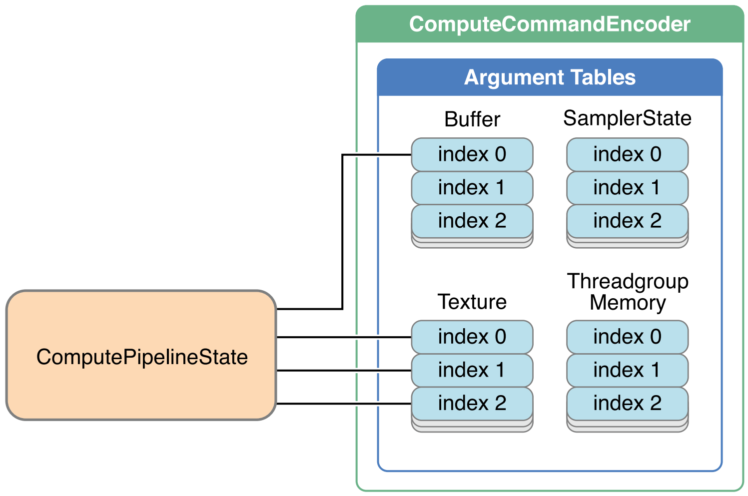
The `setComputePipelineState:` method of a `MTLComputeCommandEncoder` object specifies the state, including a compiled compute shader function, to use for a data-parallel compute pass. At any given moment, a compute command encoder can be associated to only one compute function.

The following `MTLComputeCommandEncoder` methods specify a resource (that is, a buffer, texture, sampler state, or threadgroup 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:withRange:`
- `setThreadgroupMemoryLength:atIndex:`

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

Figure 6-1 Argument Tables for the Compute Command Encoder



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

The total of all threadgroup memory allocations must not exceed 16 KB; otherwise, an error occurs.

Executing a Compute Command

To encode a command to execute a compute function, call the `dispatchThreadgroups:threadsPerThreadgroup:` method of `MTLComputeCommandEncoder` and specify the threadgroup dimensions and the number of threadgroups. You can query the `threadExecutionWidth` and `maxTotalThreadsPerThreadgroup` properties of `MTLComputePipelineState` to optimize the execution of the compute function on this device.

For most efficient execution of the compute function, set the total number of threads specified by the `threadsPerThreadgroup` argument to the `dispatchThreadgroups:threadsPerThreadgroup:` method to a multiple of `threadExecutionWidth`. The total number of threads in a threadgroup is the product of the components of `threadsPerThreadgroup`: `threadsPerThreadgroup.width * threadsPerThreadgroup.height * threadsPerThreadgroup.depth`. The `maxTotalThreadsPerThreadgroup` property specifies the maximum number of threads that can be in a single threadgroup to execute this compute function on the device.

Compute commands are executed in the order in which they are encoded into the command buffer. A compute command finishes execution when all threadgroups associated with the command finish execution and all results are written to memory. Because of this sequencing, the results of a compute command are available to any commands encoded after it in the command buffer.

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

Code Example: Executing Data-Parallel Functions

Listing 6-1 (page 63) 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` object. Next a `MTLFunction` object is created that represents the entry point `filter_main` loaded from the `MTLLibrary` object, shown in Listing 6-2 (page 65). 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:atIndex:` and `setBuffer:offset:atIndex:` methods assign texture and buffer objects to indices in the specified argument tables. `paramsBuffer` specifies values used to perform the image transformation, and `inputTableData` specifies filter weights. The compute function is executed as a 2D threadgroup of size 16 x 16 pixels in each dimension. The `dispatchThreadgroups:threadsPerThreadgroup:` method enqueues the command to dispatch the threads executing the compute function, and the `endEncoding` method terminates the `MTLComputeCommandEncoder`. Finally, the `commit` method of `MTLCommandBuffer` causes the commands to be executed as soon as possible.

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"];
id <MTLComputePipelineState> filterState
              = [device newComputePipelineStateWithFunction:func error:&errors];
[computeCE setComputePipelineState:filterState];
[computeCE setTexture:inputImage atIndex:0];
[computeCE setTexture:outputImage atIndex:1];
[computeCE setTexture:inputTableData atIndex:2];
[computeCE setBuffer:paramsBuffer offset:0 atIndex:0];

MTLSize threadsPerGroup = {16, 16, 1};
MTLSize numThreadgroups = {inputImage.width/threadsPerGroup.width,
                           inputImage.height/threadsPerGroup.height, 1};

[computeCE dispatchThreadgroups:numThreadgroups
                                threadsPerThreadgroup:threadsPerGroup];
[computeCE endEncoding];

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

Listing 6-2 shows the corresponding shader code for the preceding example. (The functions `read_and_transform` and `filter_table` are placeholders for user-defined code).

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                                    [[ thread_position_in_grid ]],
  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 data copying operations, first 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, and between a buffer and a texture.

Copying Data Between Two Buffers

The method `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

The method
`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

The method
`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

The method `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 22).

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.

Ending 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 can 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

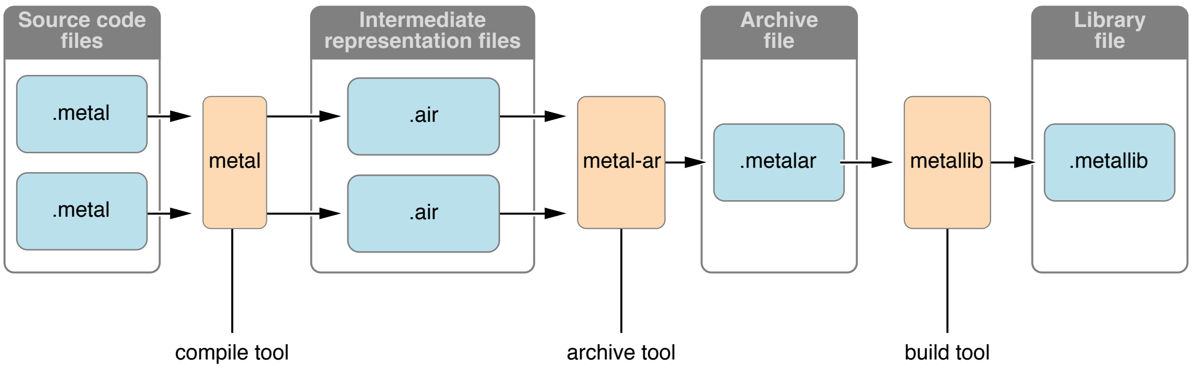
Figure 8-1 (page 69) shows the command line utilities that form the compiler toolchain for Metal shader source code. When you include `.metal` files in your project, Xcode invokes these tools to build a library file that you can access in your app at run time.

To compile shader source into a library without using Xcode:

1. Use `metal` to compile each `.metal` file into a single `.air` file, which stores an intermediate representation of shader language code.
2. Use `metal-ar` to archive several `.air` files together into a single `.metalar` file. `metal-ar` is similar to the Unix utility `ar`.

3. Use `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, call the `newLibraryWithFile:error:` method of `MTLDevice`.

Metal Feature Sets

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

There are two feature sets: `MTLFeatureSet_iOS_GPUFamily1_v1` and `MTLFeatureSet_iOS_GPUFamily2_v1`. Within an `iOS_GPUFamilyN`, the suffix _vN indicates a different version in the same feature family. Note that `MTLFeatureSet_iOS_GPUFamily1_v1` and `MTLFeatureSet_iOS_GPUFamily2_v1` are in different feature families.

Table 8-1 summarizes notable capabilities and limitations of each feature set.

Table 8-1 Metal Feature Sets and Capabilities

Feature	GPU Family 1 v1 Value	GPU Family 2 v1 Value
Maximum number of color attachments per `MTLRenderPassDescriptor`	4	8
Maximum color data output per sample (across all color attachments) per rendering pass	16 bytes	32 bytes
Supports ASTC pixel formats	No	Yes

Feature	GPU Family 1 v1 Value	GPU Family 2 v1 Value
Minimum attachment size (color or depth/stencil), width or height	32 pixels	32 pixels
Threadgroup memory allocation size increment	16 bytes	16 bytes
Maximum total threadgroup memory allocation	16384 bytes	16384 bytes
Maximum `MTLTextureDescriptor` `height` and `width`	4096	4096
Maximum `MTLTextureDescriptordepth`	2048	2048
Maximum number of entries in argument tables for render and compute command encoders	31 (buffer) 31 (texture) 16 (sampler state)	31 (buffer) 31 (texture) 16 (sampler state)

The following notes apply to both feature sets:

- The *Maximum color data output* listed in Table 8-1 limits the total color data each render pass can store for each output pixel, across all of its color attachments. (Depth and stencil attachments do not count against this limit.) If you create a `MTLRenderPassDescriptor` and the sum of the storage requirements for all color attachments is greater than the maximum allowed, a fatal error occurs.

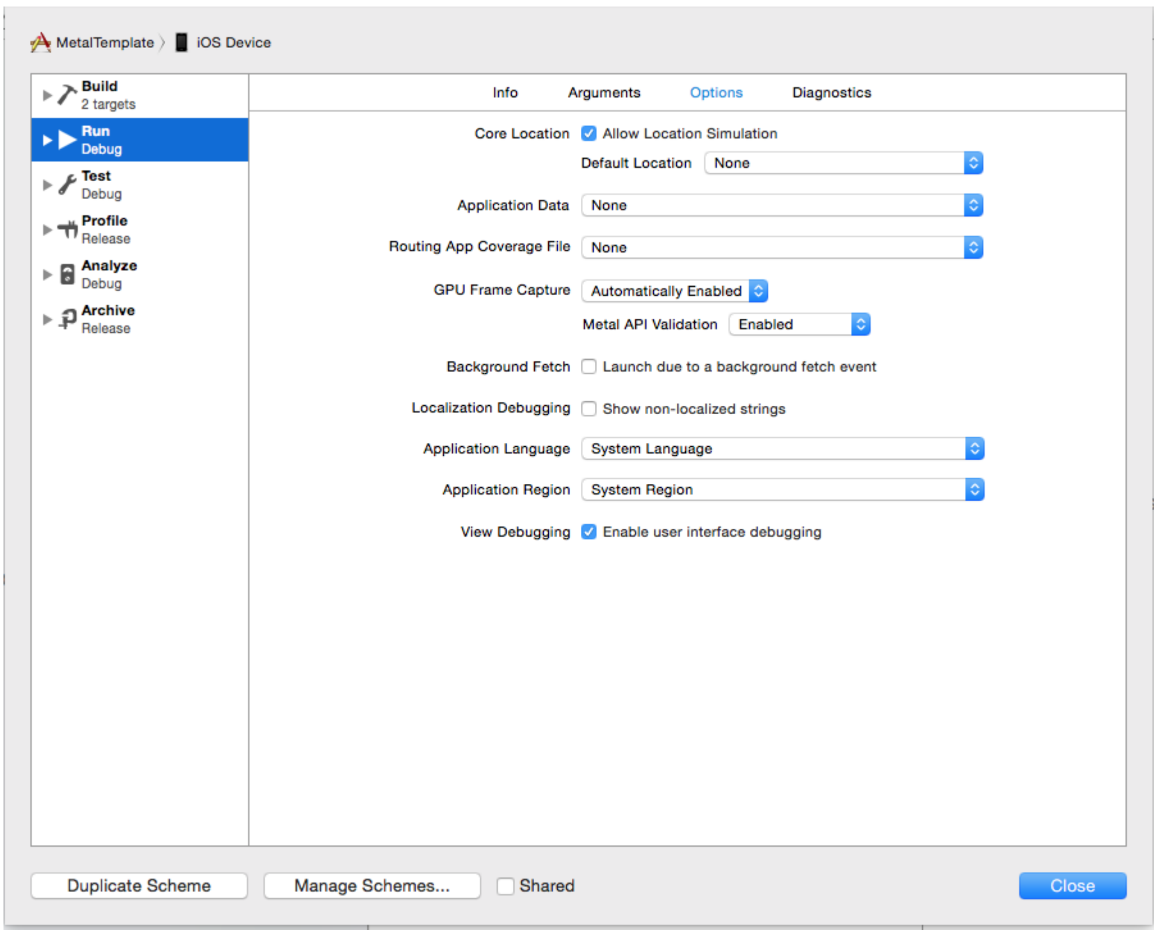
  All pixel formats consume a minimum of `4` bytes per sample in an attachment, even if the pixel formats use fewer than `4` bytes per pixel in memory. For example, the `MTLPixelFormatR8Unorm`, `MTLPixelFormatR8Uint`, and `MTLPixelFormatR8Sint` pixel formats use 1 byte per pixel in memory, but consume `4` bytes per sample in an attachment. The `MTLPixelFormatRGB10A2Unorm`, `MTLPixelFormatRG11B10Float`, and `MTLPixelFormatRGB9E5Float` pixel formats use `4` bytes per pixel in memory, but consume `8` bytes per sample in an attachment. All other pixel formats take the same amount of space in memory as in attachment storage.

- The *Minimum attachment size* listed in Table 8-1 is not a limit; however, attachments smaller in either width or depth than this value have an increased performance cost. This is especially true of depth/stencil attachments.

Xcode Scheme Settings and Performance

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

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



Debugging

Use the tips in the following sections to gain more useful diagnostic information when debugging and profiling your Metal app.

File Extension for Metal Shading Language Source Files

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

Performing Frame Capture with Xcode

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

The Label Property

Many Metal framework objects—such as command buffers, pipeline states, and resources—support a `label` property. You can use this property to assign a name for each object that is meaningful in the context of your application's design. These labels appear in the Xcode Frame Capture debugging interface, alowing you to more easily identify objects.

Document Revision History

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

Date	Notes
2015-03-09	Fixed bugs in code snippets and included more information about display scale.
2015-01-12	Edited for clarity.
2014-09-17	New document that describes how to use the Metal framework to implement low-overhead graphics rendering or parallel computational tasks.

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