API Collection

Buffers

Create and manage untyped data your app uses to exchange information with its shader functions.

Overview

Each `MTLBuffer` instance represents a general purpose, typeless memory allocation that your app uses to send and retrieve data from a shader. Your app decides how to use and interpret the buffer's underlying bytes.

You create buffers from either an `MTLDevice` or `MTLHeap` instance.

Swift | Objective-C | C++

```
let deviceBuffer = device.makeBuffer(length: bufferSize,
                                     options: .storageModeShared)

let heapBuffer = heap.makeBuffer(length: bufferSize,
                                 options: .storageModePrivate)
```

Buffers inherently support the `MTLResource` protocol's properties and methods, including `storageMode`, which controls how the GPU handles its memory (see Resource fundamentals).

Topics

General purpose buffers

Store arbitrary data in a buffer, such as vertex locations or your own custom data structure.

`protocol` `MTLBuffer`

A resource that stores data in a format defined by your app.

Argument buffers

Group resources together into an argument buffer.

Improving CPU performance by using argument buffers

Optimize your app's performance by grouping your resources into argument buffers.

{} Managing groups of resources with argument buffers

Create argument buffers to organize related resources.

Tracking the resource residency of argument buffers

Optimize resource performance within an argument buffer.

Indexing argument buffers

Assign resource indices within an argument buffer.

{} Rendering terrain dynamically with argument buffers

Use argument buffers to render terrain in real time with a GPU-driven pipeline.

{} Encoding argument buffers on the GPU

Use a compute pass to encode an argument buffer and access its arguments in a subsequent render pass.

{} Using argument buffers with resource heaps

Reduce CPU overhead by using arrays inside argument buffers and combining them with resource heaps.

`class` `MTLArgumentDescriptor`

A representation of an argument within an argument buffer.

`protocol` `MTLArgumentEncoder`

An interface you can use to encode argument data into an argument buffer.

`let` `MTLAttributeStrideStatic: Int`

Model I/O interoperability

Load complex 3D meshes and textures from Model I/O assets, and prepare to draw them in your Metal render pipelines.

`class MTKMesh`

A container for the vertex data of a Model I/O mesh, suitable for use in a Metal app.

`class MTKMeshBuffer`

A buffer that backs the vertex data of a Model I/O mesh, suitable for use in a Metal app.

`class MTKMeshBufferAllocator`

An interface for allocating a MetalKit buffer that backs the vertex data of a Model I/O mesh, suitable for use in a Metal app.

`class MTKSubmesh`

A container for the index data of a Model I/O submesh, suitable for use in a Metal app.

`struct MTKModelError`

Constants used to declare Model Errors.

```
func MTKMetalVertexFormatFromModelIO(_ vertexFormat: MDLVertexFormat) -
> MTLVertexFormat
```

Returns a converted Metal vertex format.

```
func MTKModelIOVertexFormatFromMetal(_ vertexFormat: MTLVertexFormat) -
> MDLVertexFormat
```

Returns a converted Model I/O vertex format.

```
func MTKMetalVertexDescriptorFromModelIO(_ modelIODescriptor: MDLVertex
Descriptor) -> MTLVertexDescriptor?
```

Returns a partially converted Metal vertex descriptor.

```
func MTKModelIOVertexDescriptorFromMetal(_ metalDescriptor: MTLVertex
Descriptor) -> MDLVertexDescriptor
```

Returns a partially converted Model I/O vertex descriptor.

See Also

Resources

Resource fundamentals

Control the common attributes of all Metal memory resources, including buffers and textures, and how to configure their underlying memory.

Textures

Create and manage typed data your app uses to exchange information with its shader functions.

Memory heaps

Take control of your app's GPU memory management by creating a large memory allocation for various buffers, textures, and other resources.

Resource loading

Load assets in your games and apps quickly by running a dedicated input/output queue alongside your GPU tasks.

Resource synchronization

Prevent multiple commands that can access the same resources simultaneously by coordinating those accesses with barriers, fences, or events.