Graphics and Games

#WWDC16

Adopting Metal, Part 2

Session 603

Matt Collins GPU Software Engineer
Jared Marsau GPU Software Engineer

© 2016 Apple Inc. All rights reserved. Redistribution or public display not permitted without written permission from Apple.

Metal at WWDC This Year

A look at the sessions

Adopting Metal

Part One

- Fundamental Concepts
- Basic Drawing
- Lighting and Texturing

Part Two

- Dynamic Data Management
- CPU-GPU Synchronization
- Multithreaded Encoding

Metal at WWDC This Year

A look at the sessions

What's New in Metal

Part One

- Tessellation
- Resource Heaps and Memoryless Render Targets
- Improved Tools

Part Two

- Function Specialization and Function Resource Read-Writes
- Wide Color and Texture Assets
- Additions to Metal Performance Shaders

Metal at WWDC This Year

A look at the sessions

Advanced Shader Optimization

- Shader Performance Fundamentals
- Turning Shader Code

Agenda

Agenda

Conceptual Overview

Creating a Metal Device

Loading Data

Metal Shading Language

Building Pipeline States

Issuing GPU Commands

Animation and Texturing

Managing Dynamic Data

CPU/GPU Synchronization

Multithreaded Encoding

Demo

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

```
// Managing Data

func draw()
{
    updateObjects(renderables)

    drawObjects(renderables)

    submitGPUCommands()
}
```

Avoid This at All Costs!

Push Data

Bind shader, buffers, textures

Draw

Object 0 Data

Push Data

Bind shader, buffers, textures

Draw

Object 1 Data

A New Paradigm

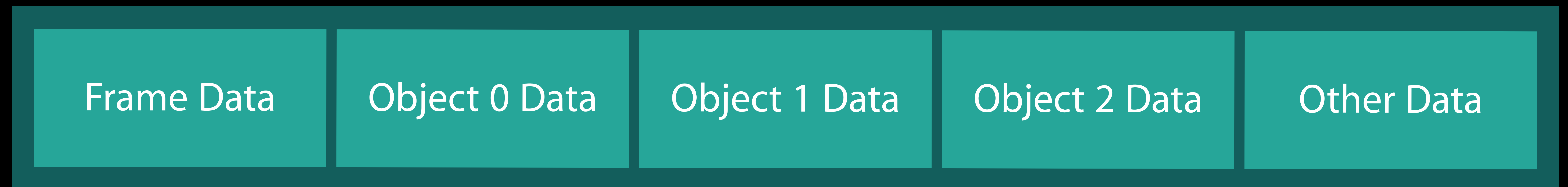
All data should be in place before rendering starts

Constant Buffer

A New Paradigm

All data should be in place before rendering starts

Constant Buffer

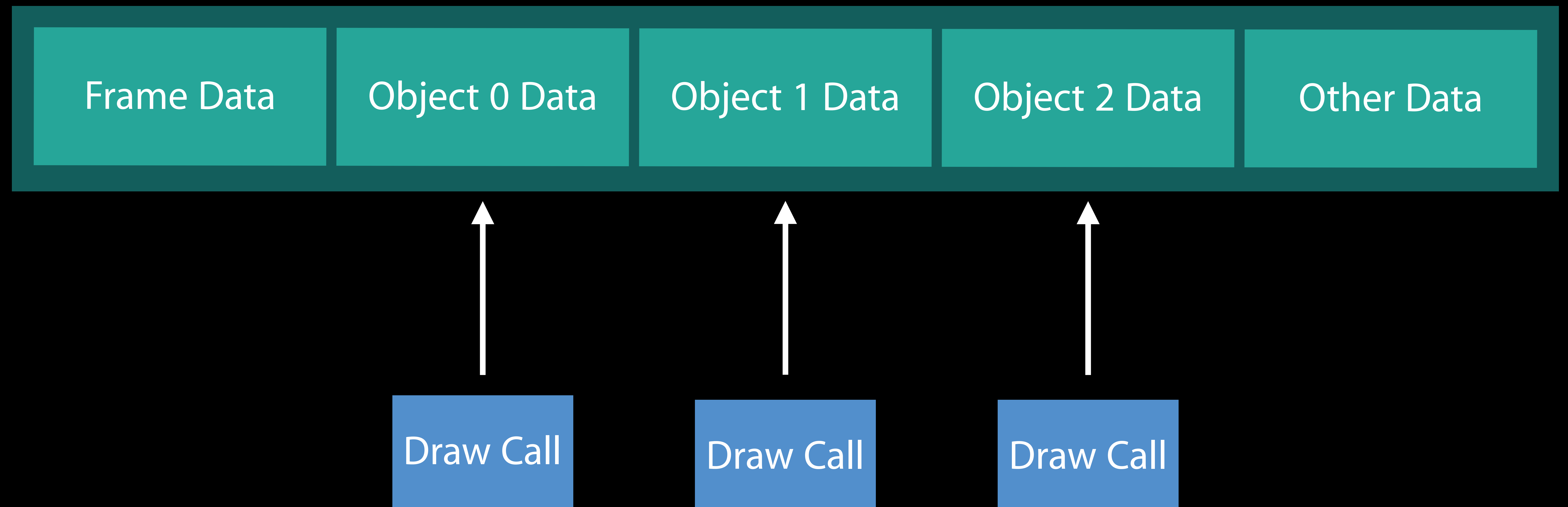


A New Paradigm

Prep all data

Draw entire scene

Constant Buffer



Managing Data

Create one Constant Buffer for the entire frame

Do not duplicate data

Reference offsets with draw calls

```
// Updating Dynamic Data

let constantBuf = constantBuffer

// Frame data
var mainPtr = constantBuf.contents()
mainPassFrameData = GetFrameData()
copyDataToBuffer(mainPtr, from: mainPassFrameData, offset: 0)

// Per-object data
var offset = sizeof(MainPass)
for o in renderables {
    o.updateData(ptr, deltaTime: deltaTime, offset: offset)
    offset += strideof(ObjectData)
}
```

```
// Updating Dynamic Data

let constantBuf = constantBuffer

// Frame data
var mainPtr = constantBuf.contents()
mainPassFrameData = GetFrameData()
copyDataToBuffer(mainPtr, from: mainPassFrameData, offset: 0)

// Per-object data
var offset = sizeof(MainPass)
for o in renderables {
    o.updateData(ptr, deltaTime: deltaTime, offset: offset)
    offset += strideof(ObjectData)
}
```

```
// Updating Dynamic Data

let constantBuf = constantBuffer

// Frame data
var mainPtr = constantBuf.contents()
mainPassFrameData = GetFrameData()
copyDataToBuffer(mainPtr, from: mainPassFrameData, offset: 0)

// Per-object data
var offset = sizeof(MainPass)
for o in renderables {
    o.updateData(ptr, deltaTime: deltaTime, offset: offset)
    offset += strideof(ObjectData)
}
```

Per-Frame Data

Data that is constant across ALL draw calls

Only need one copy

```
struct MainPass
{
   float4x4 ViewProjection;
};
```

```
// Writing Per-Frame data

// Get buffer pointer
var mainPtr = constantBuf.contents()

// Get our ViewProjection matrix
mainPassFrameData = GetFrameData()

// Copy MainPass struct into buffer
copyDataToBuffer(mainPtr, from: mainPassFrameData, offset: 0)
```

```
// Writing Per-Frame data

// Get buffer pointer
var mainPtr = constantBuf.contents()

// Get our ViewProjection matrix
mainPassFrameData = GetFrameData()

// Copy MainPass struct into buffer
copyDataToBuffer(mainPtr, from: mainPassFrameData, offset: 0)
```

```
// Writing Per-Frame data

// Get buffer pointer
var mainPtr = constantBuf.contents()

// Get our ViewProjection matrix
mainPassFrameData = GetFrameData()

// Copy MainPass struct into buffer
copyDataToBuffer(mainPtr, from: mainPassFrameData, offset: 0)
```

```
// Writing Per-Frame data

// Get buffer pointer
var mainPtr = constantBuf.contents()

// Get our ViewProjection matrix
mainPassFrameData = GetFrameData()

// Copy MainPass struct into buffer
copyDataToBuffer(mainPtr, from: mainPassFrameData, offset: 0)
```

Prepare Data

```
struct MainPass
{
   float4x4 ViewProjection;
};
```

Prepare Data

```
struct MainPass
{
   float4x4 ViewProjection;
};
```

Constant Buffer

Per-Object Data

Unique data needed to draw a single object

```
struct ObjectData
{
   float4x4 LocalToWorld;
   float4 color;
};
```

```swift
// Updating Per-object data

// Track the offset into the constant buffer
var offset = strideof(MainPass)
for o in renderables {
    o.updateData(ptr, deltaTime: deltaTime, offset: offset)
    offset += strideof(ObjectData)
}
```

```swift
// Updating Per-object data

// Track the offset into the constant buffer
var offset = strideof(MainPass)
for o in renderables {
    o.updateData(ptr, deltaTime: deltaTime, offset: offset)
    offset += strideof(ObjectData)
}
```

```swift
// Updating Per-object data

func UpdateData(dest : Pointer<ObjectData>, deltaTime : Float, offset : Int)
{
    // Fetch data from animation system
    var objectData = updateAnimation(deltaTime)

    // Copy data to buffer
    copyDataToBuffer(dest, from: objectData, offset: offset)
}
```

```
// Updating Per-object data

func UpdateData(dest : Pointer<ObjectData>, deltaTime : Float, offset : Int)
{
    // Fetch data from animation system
    var objectData = updateAnimation(deltaTime)

    // Copy data to buffer
    copyDataToBuffer(dest, from: objectData, offset: offset)
}
```

```
// Updating Per-object data

func UpdateData(dest : Pointer<ObjectData>, deltaTime : Float, offset : Int)
{
    // Fetch data from animation system
    var objectData = updateAnimation(deltaTime)

    // Copy data to buffer
    copyDataToBuffer(dest, from: objectData, offset: offset)
}
```

Prepare Data

```
struct MainPass
{
   float4x4 ViewProjection;
};
```

Constant Buffer

Prepare Data

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

Constant Buffer

Frame Data

Prepare Data

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

Prepare Data

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

Constant Buffer

Frame Data	Object 0 Data	Object 1 Data

Prepare Data

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

```
struct MainPass
{
    float4x4 ViewProjection;
};
```

Constant Buffer

Frame Data	Object 0 Data	Object 1 Data	Object 2 Data

```
// How Do You Protect Your Constant Buffer?

let constantBuf = constantBuffer
```

Resource Contention

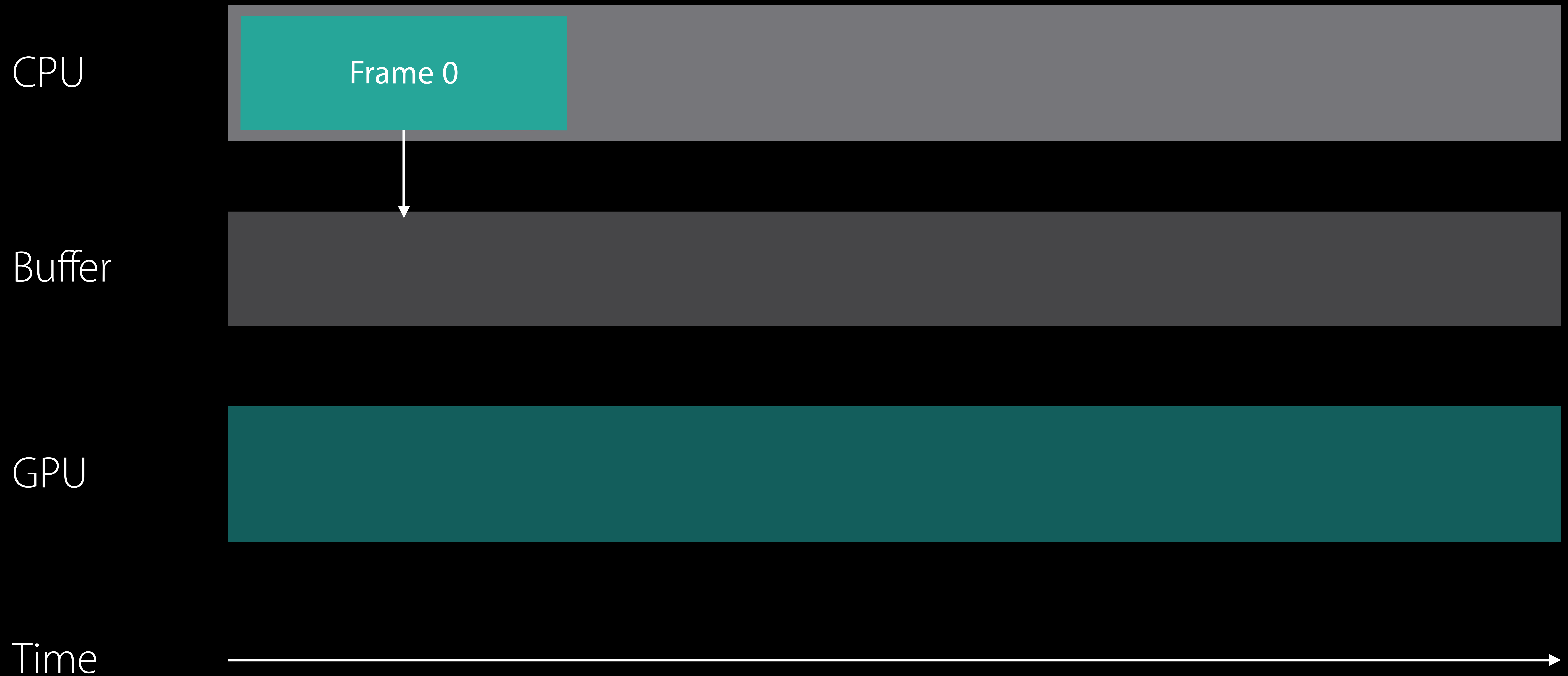
CPU

Buffer

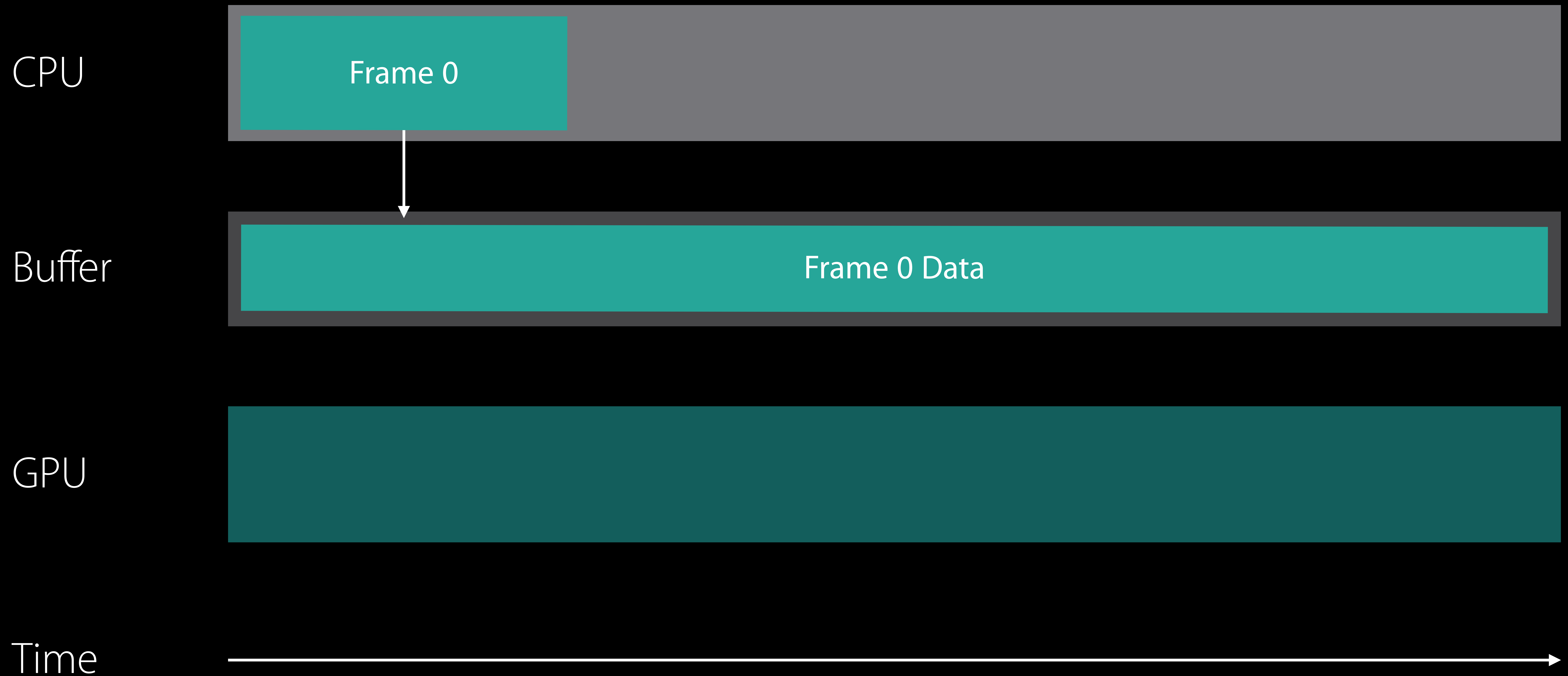
GPU

Time

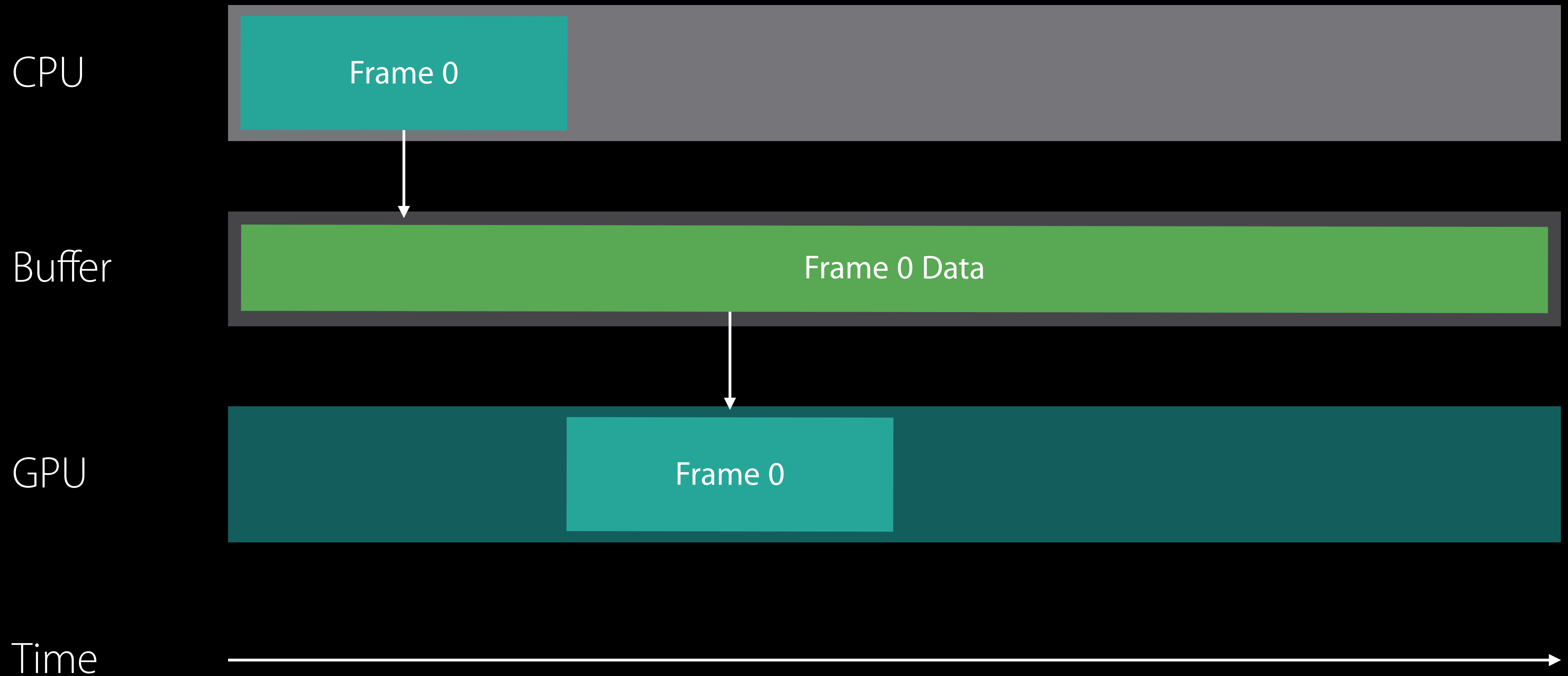
Resource Contention



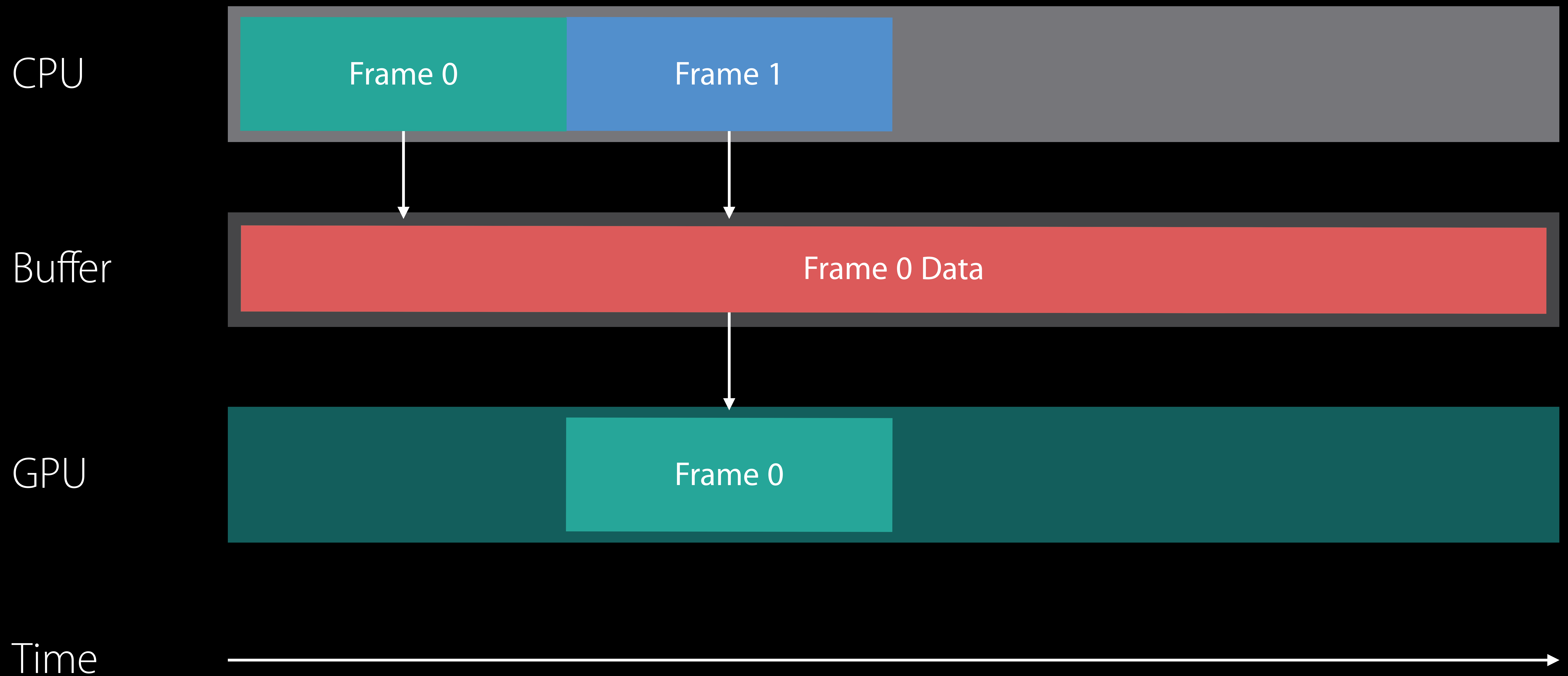
Resource Contention



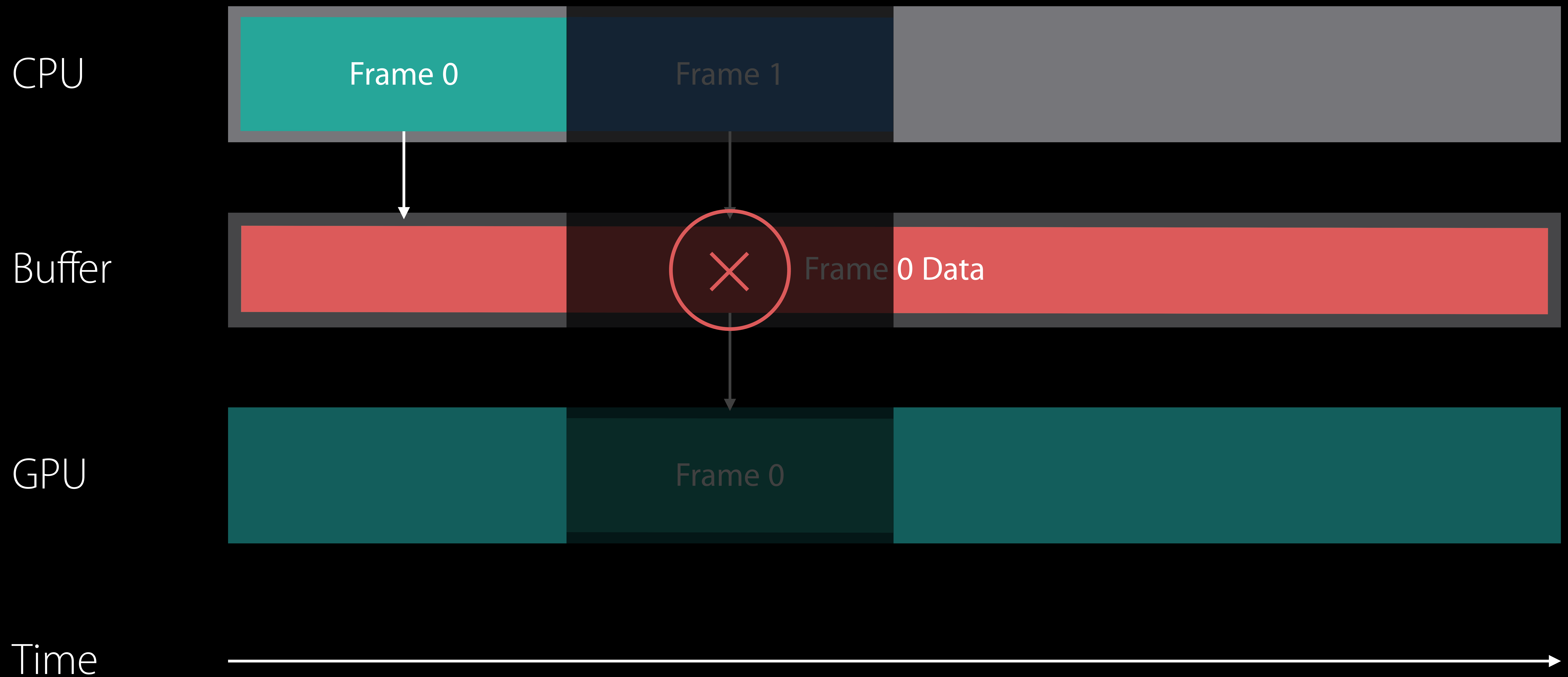
Resource Contention



Resource Contention



Resource Contention



Resource Contention

This is not handled implicitly!

CPU and GPU can read and write the same data simultaneously

You must synchronize access yourself

Agenda

Conceptual Overview

Creating a Metal Device

Loading Data

Metal Shading Language

Building Pipeline States

Issuing GPU Commands

Animation and Texturing

Managing Dynamic Data

CPU/GPU Synchronization

Multithreaded Encoding

Agenda

Conceptual Overview

Creating a Metal Device

Loading Data

Metal Shading Language

Building Pipeline States

Issuing GPU Commands

Animation and Texturing

Managing Dynamic Data

CPU/GPU Synchronization

Multithreaded Encoding

```
// Naive Synchronization

func draw()
{
    updateData(renderables)

    drawObjects(renderables)

    submitGPUCommands()

    waitUntilCompleted()
}
```

```swift
// Naive Synchronization

func draw()
{
    updateData(renderables)

    drawObjects(renderables)

    submitGPUCommands()

    waitUntilCompleted()
}
```

```
// Command Buffer Handler Callbacks

cmdBuf.addScheduledHandler
{
    // This code will execute when the command buffer is sent to the GPU
}

cmdBuf.addCompletedHandler
{
    // This code will execute when the command buffer is retired
    // It is now safe to modify/destroy data
    signalGPUCommandsComplete()
}
```

```
// Command Buffer Handler Callbacks

cmdBuf.addScheduledHandler
{
    // This code will execute when the command buffer is sent to the GPU
}

cmdBuf.addCompletedHandler
{
    // This code will execute when the command buffer is retired
    // It is now safe to modify/destroy data
    signalGPUCommandsComplete()
}
```

```swift
// Restricting access to a single resource
var semaphore = DispatchSemaphore(value: 1)

func draw
{
    // Block until resource is available
    _ = semaphore.wait(timeout: DispatchTime.distantFuture)

    // Frame continues, creates command buffers
    cmdBuf.addCompletedHandler
    {
        // Signal resource is available
        semaphore.signal()
    }
    cmdBuf.commit()
}
```

```swift
// Restricting access to a single resource
var semaphore = DispatchSemaphore(value: 1)

func draw
{
    // Block until resource is available
    _ = semaphore.wait(timeout: DispatchTime.distantFuture)

    // Frame continues, creates command buffers
    cmdBuf.addCompletedHandler
    {
        // Signal resource is available
        semaphore.signal()
    }
    cmdBuf.commit()
}
```

```swift
// Restricting access to a single resource
var semaphore = DispatchSemaphore(value: 1)

func draw
{
    // Block until resource is available
    _ = semaphore.wait(timeout: DispatchTime.distantFuture)

    // Frame continues, creates command buffers
    cmdBuf.addCompletedHandler
    {
        // Signal resource is available
        semaphore.signal()
    }
    cmdBuf.commit()
}
```

```swift
// Restricting access to a single resource
var semaphore = DispatchSemaphore(value: 1)

func draw
{
    // Block until resource is available
    _ = semaphore.wait(timeout: DispatchTime.distantFuture)

    // Frame continues, creates command buffers
    cmdBuf.addCompletedHandler
    {
        // Signal resource is available
        semaphore.signal()
    }
    cmdBuf.commit()
}
```

Naive Synchronization

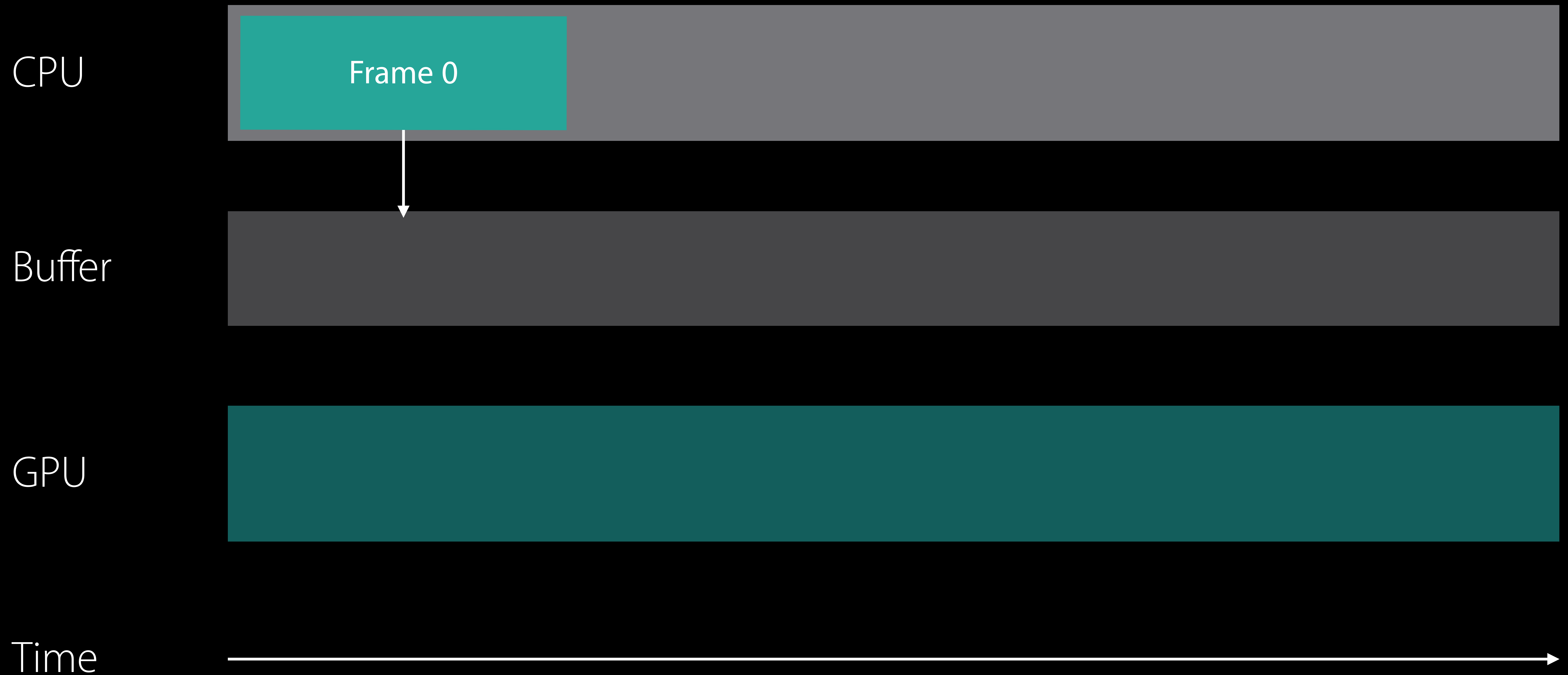
CPU

Buffer

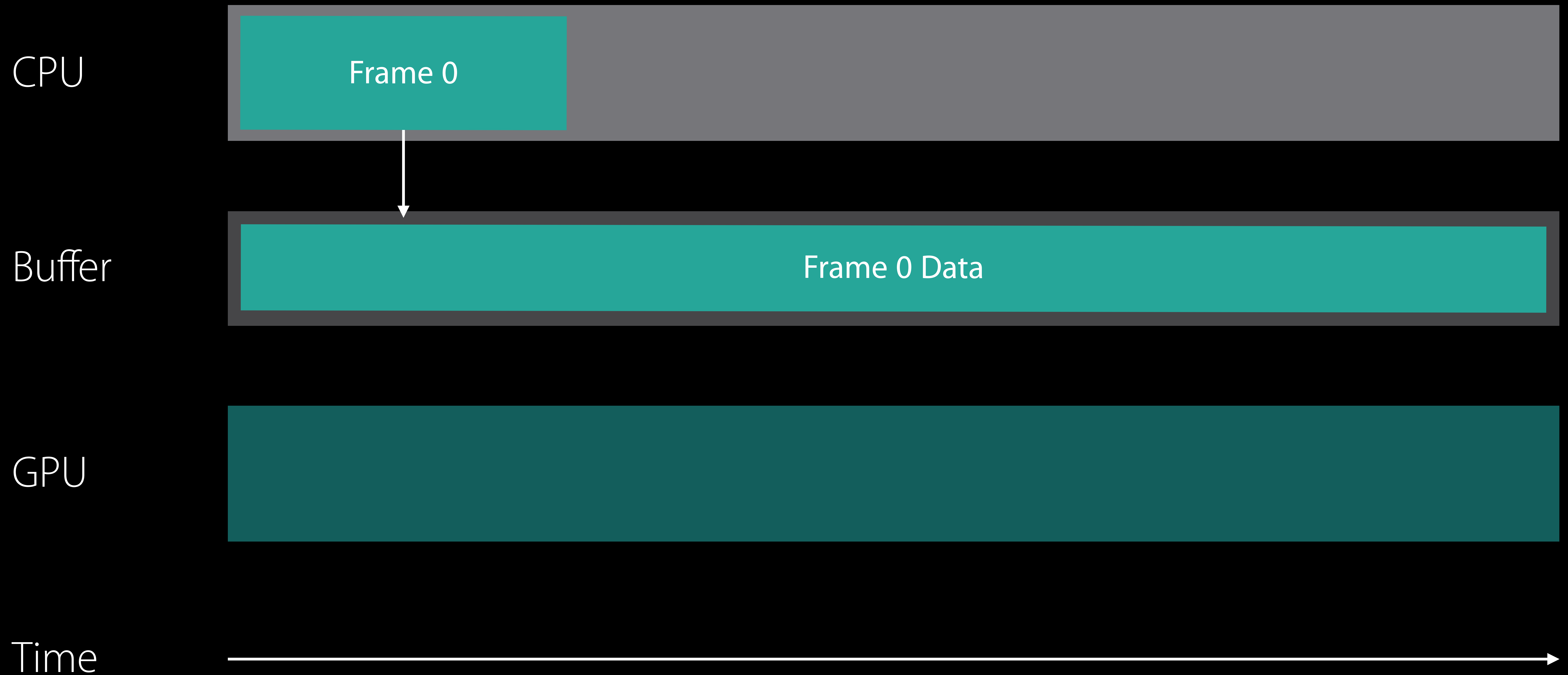
GPU

Time

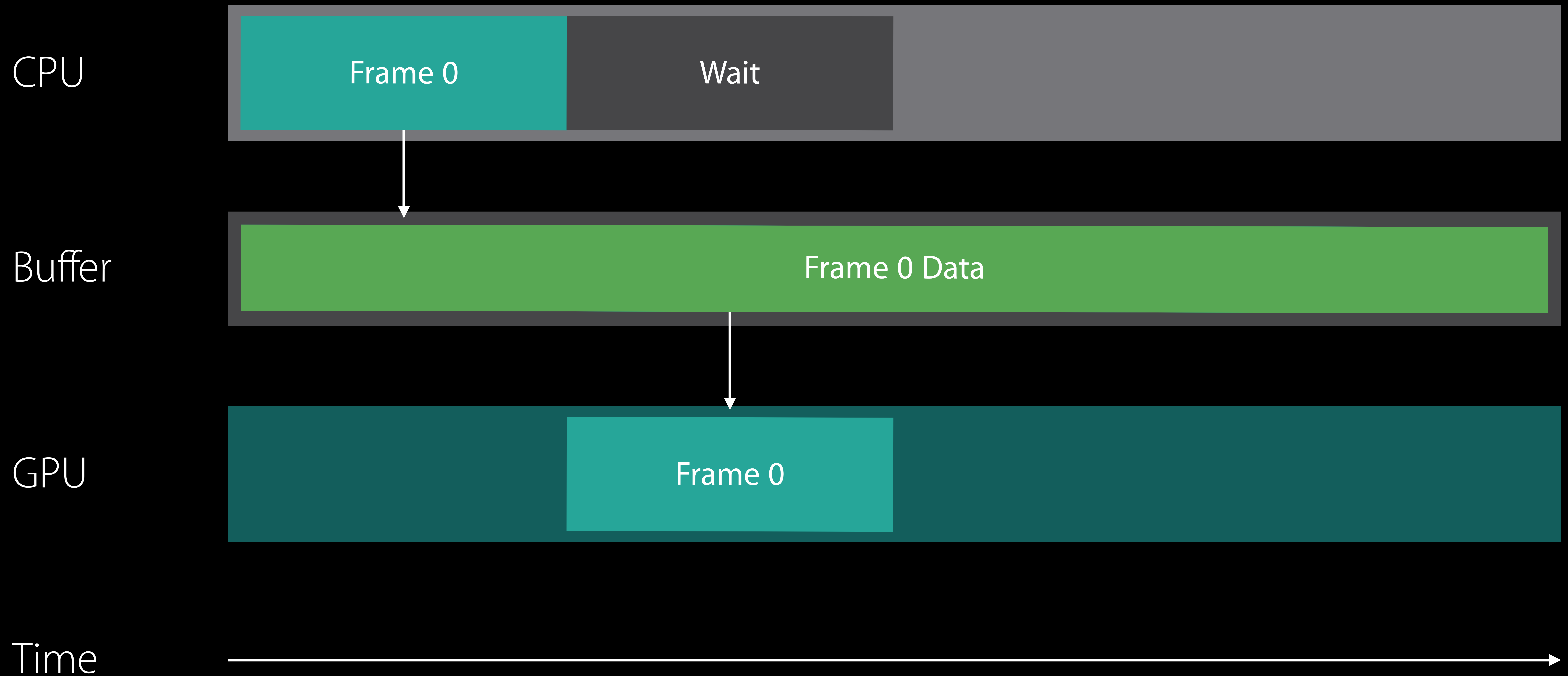
Naive Synchronization



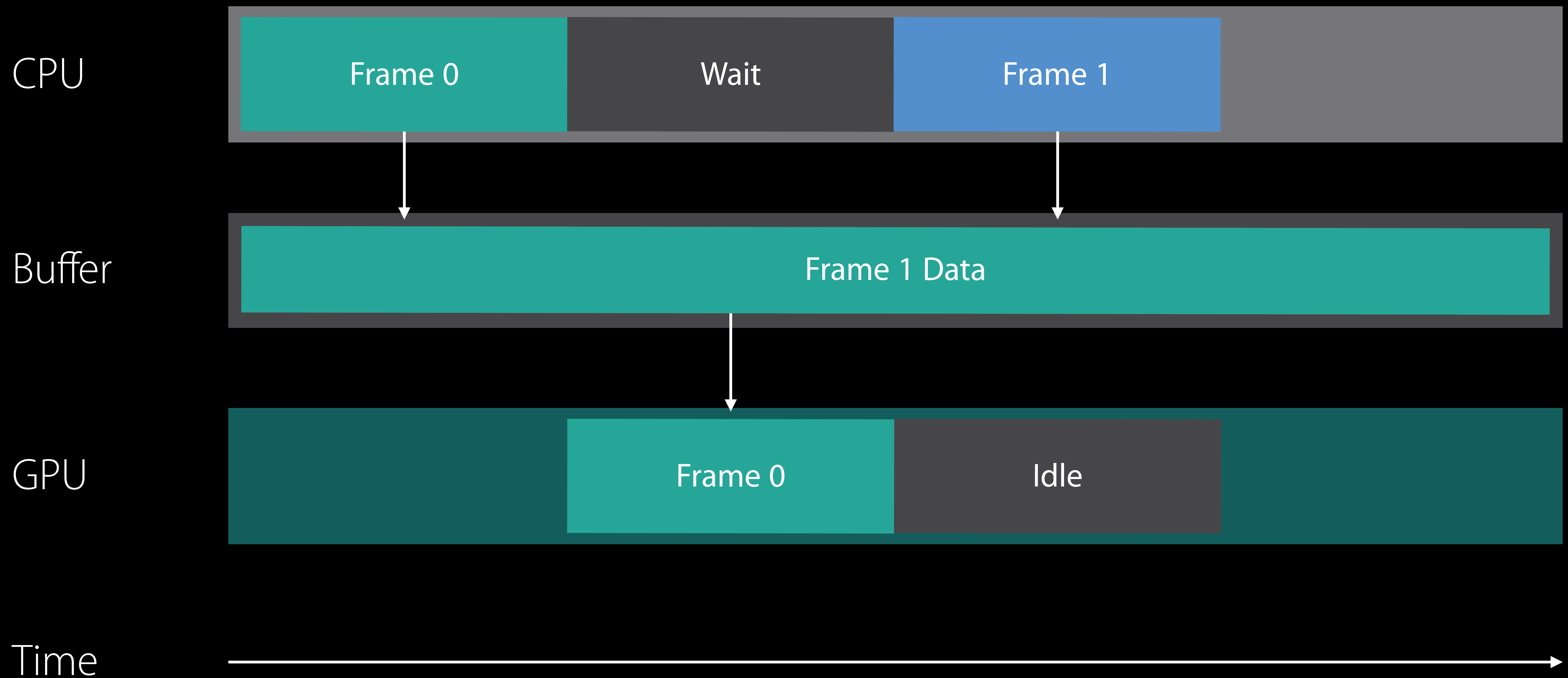
Naive Synchronization



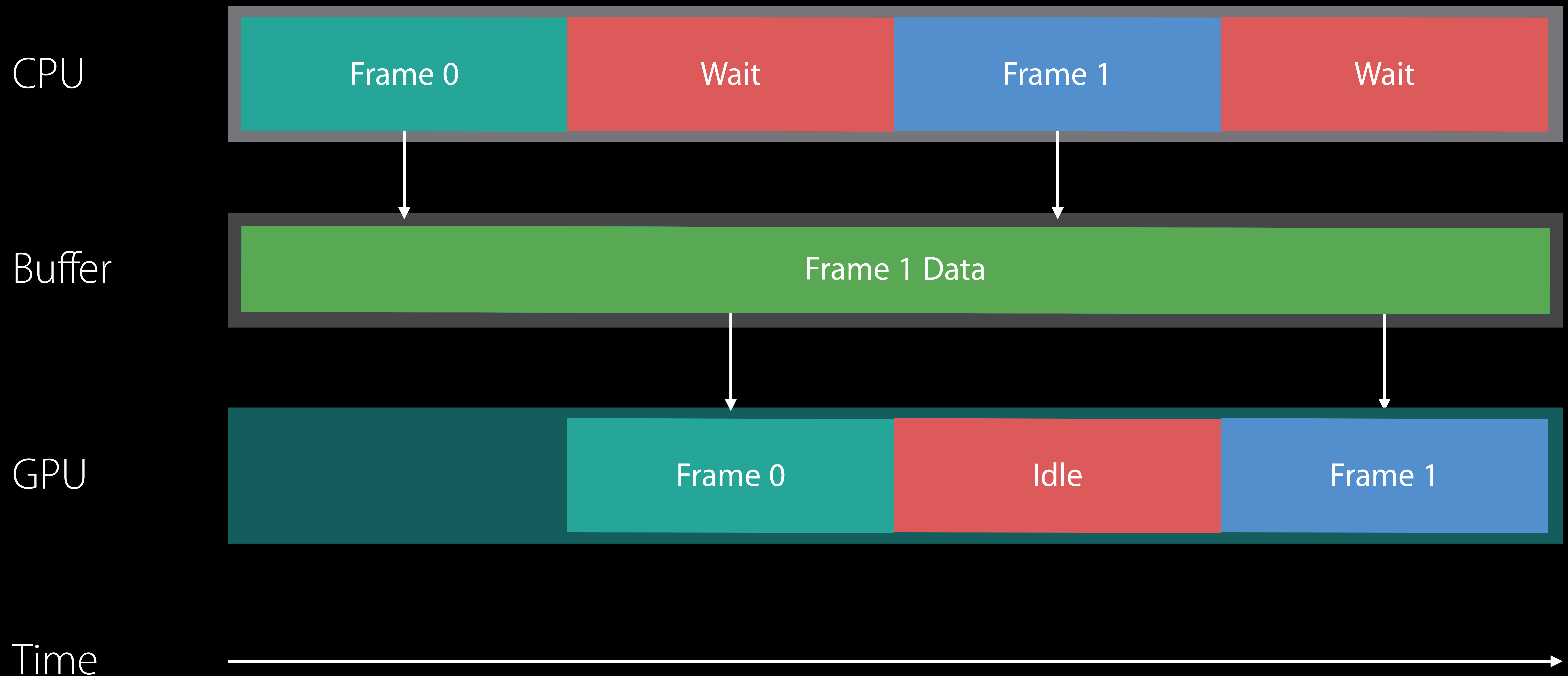
Naive Synchronization



Naive Synchronization



Naive Synchronization



Overlap CPU and GPU Work

- Improve parallelism between CPU and GPU
- Need to avoid stomping data

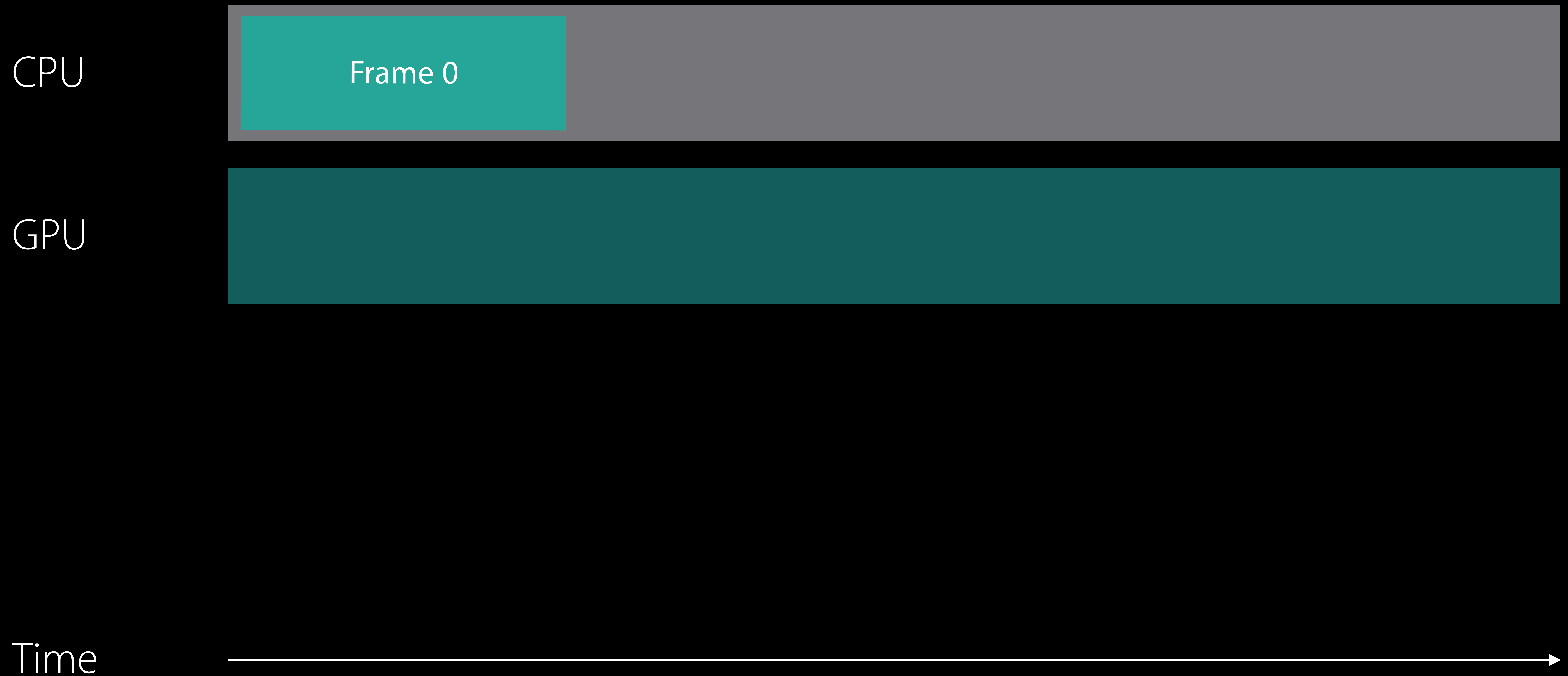
Ideal Workload

CPU

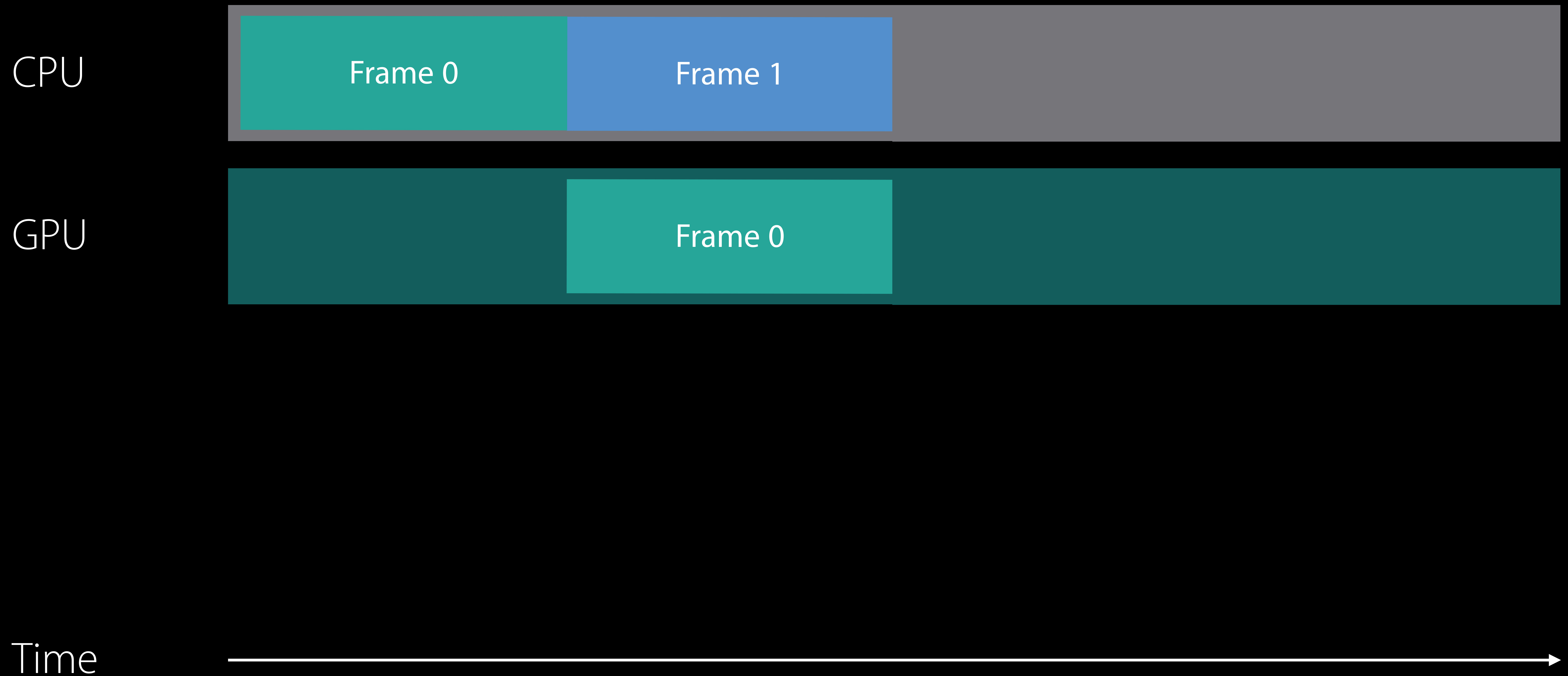
GPU

Time

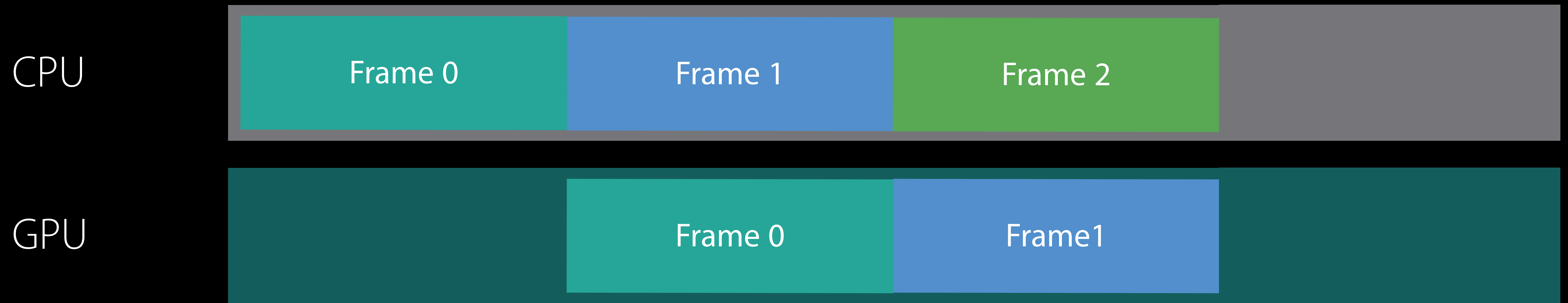
Ideal Workload



Ideal Workload

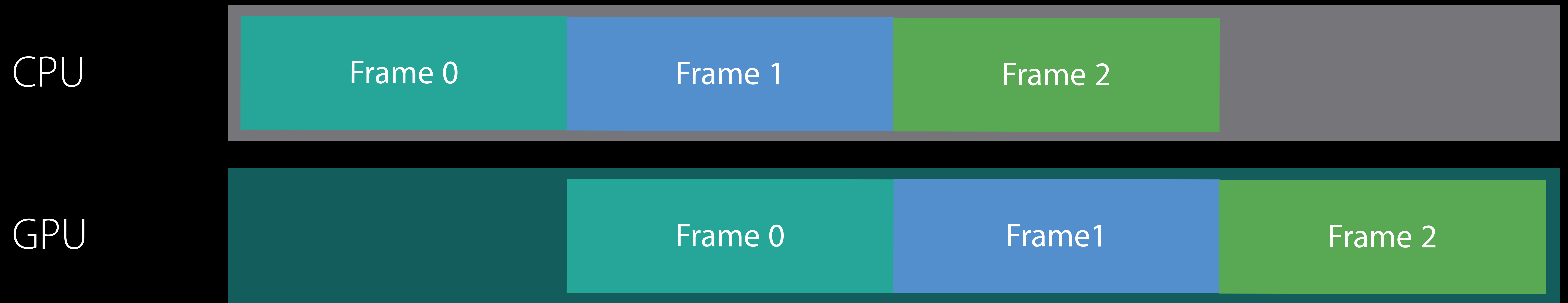


Ideal Workload



Time

Ideal Workload



Time

Demo Synchronization

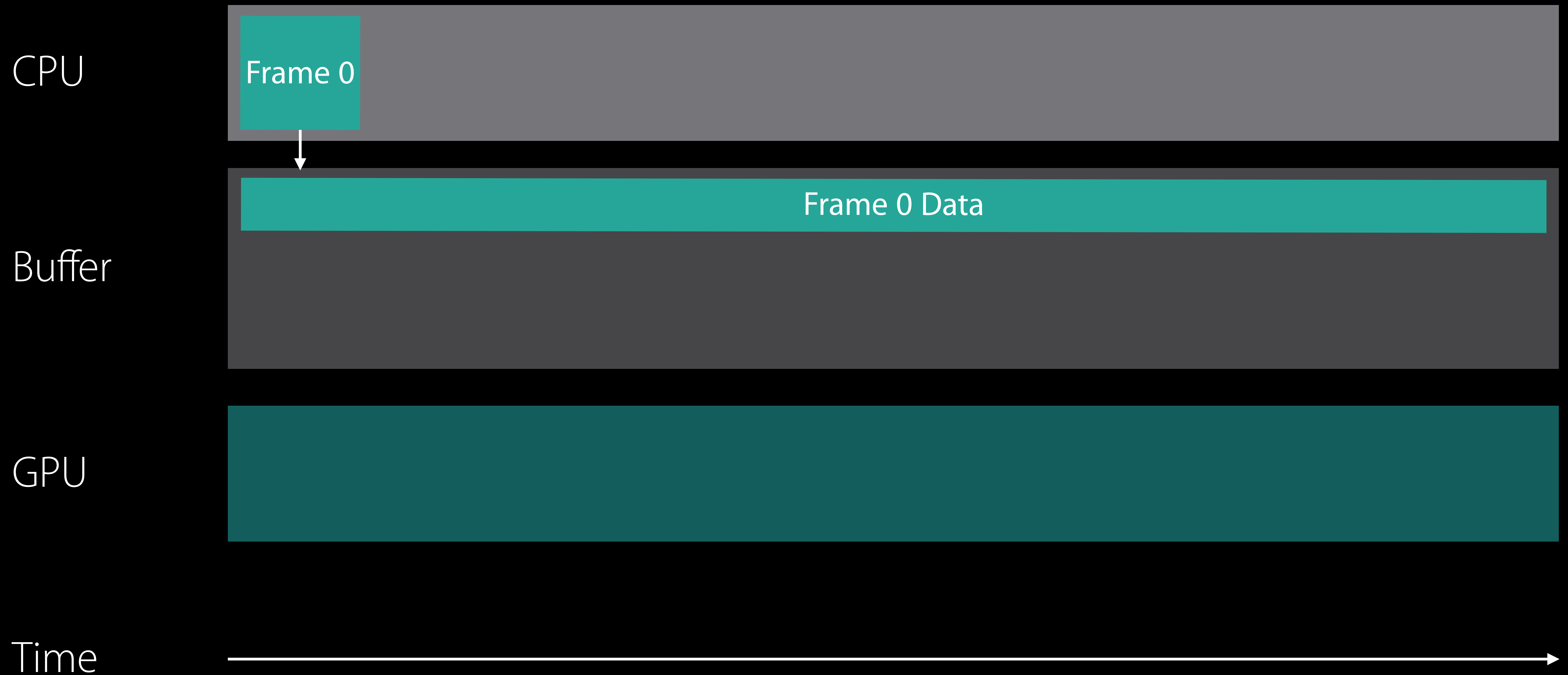
CPU

Buffer

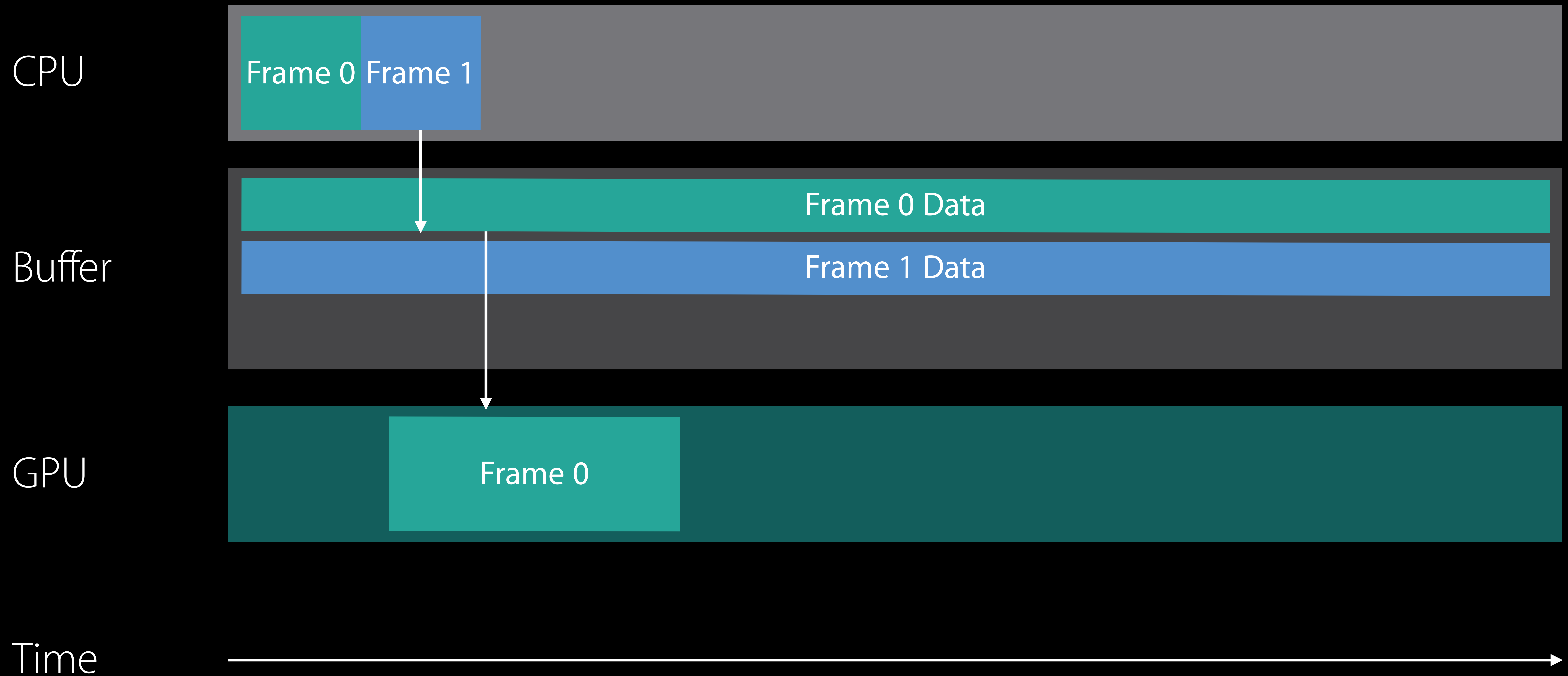
GPU

Time

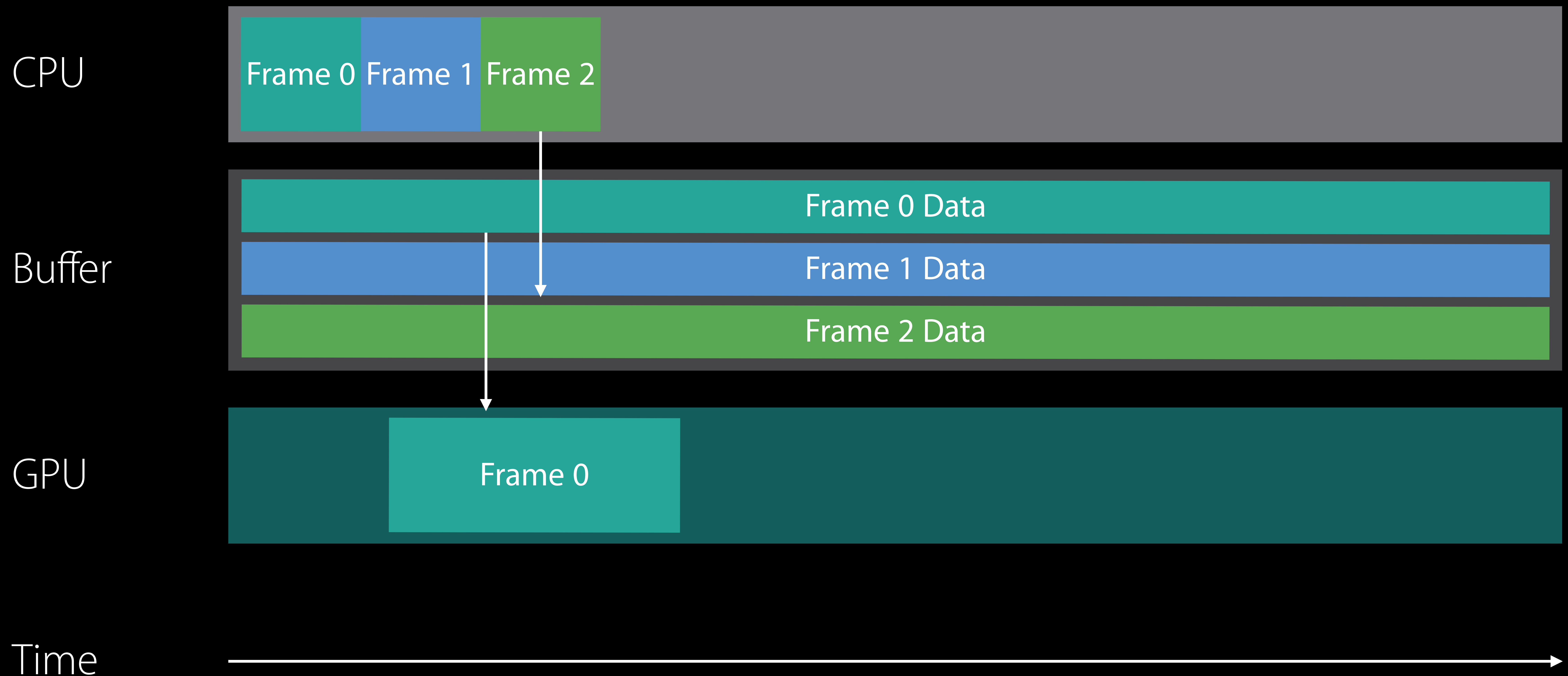
Demo Synchronization



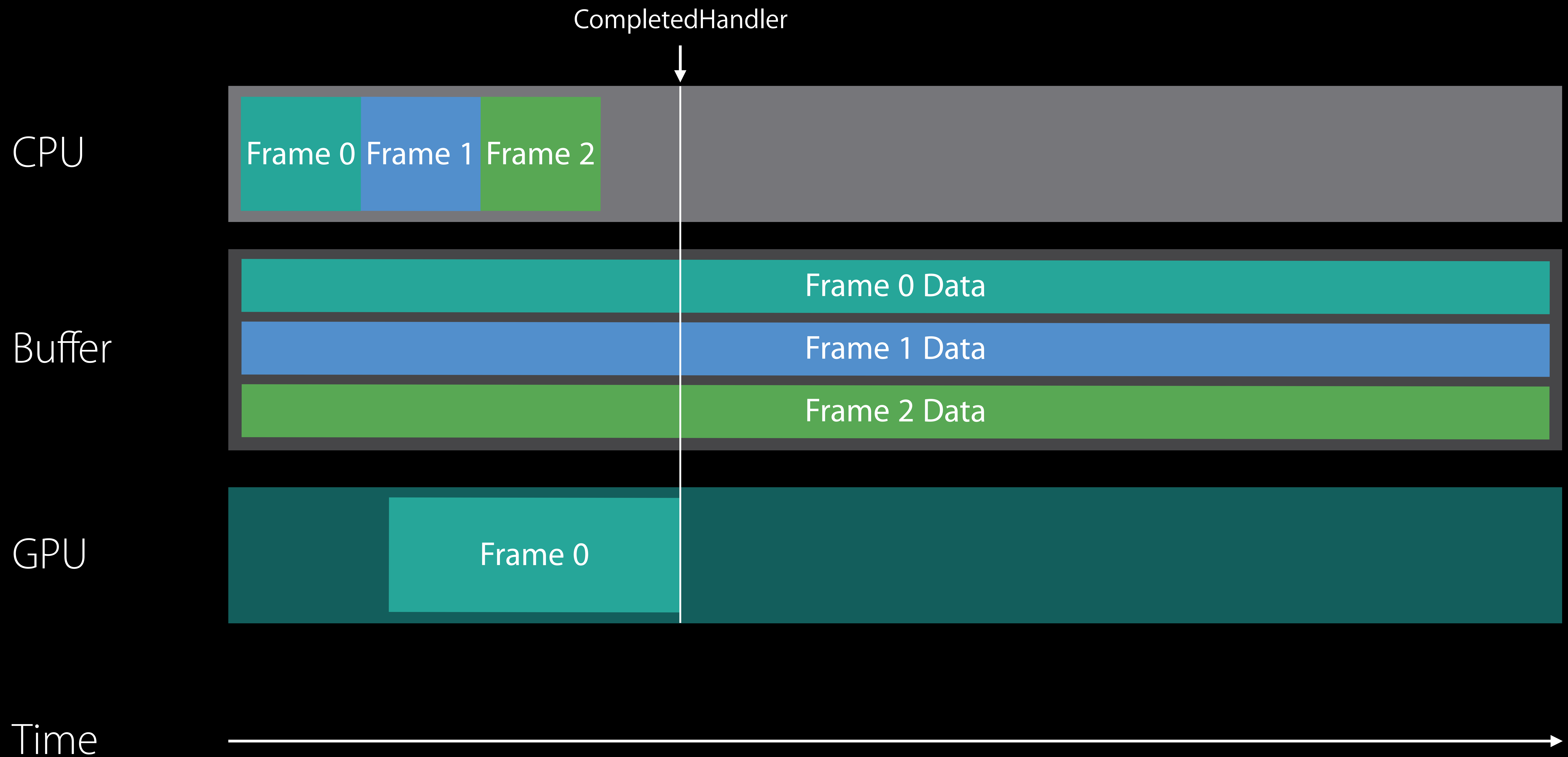
Demo Synchronization



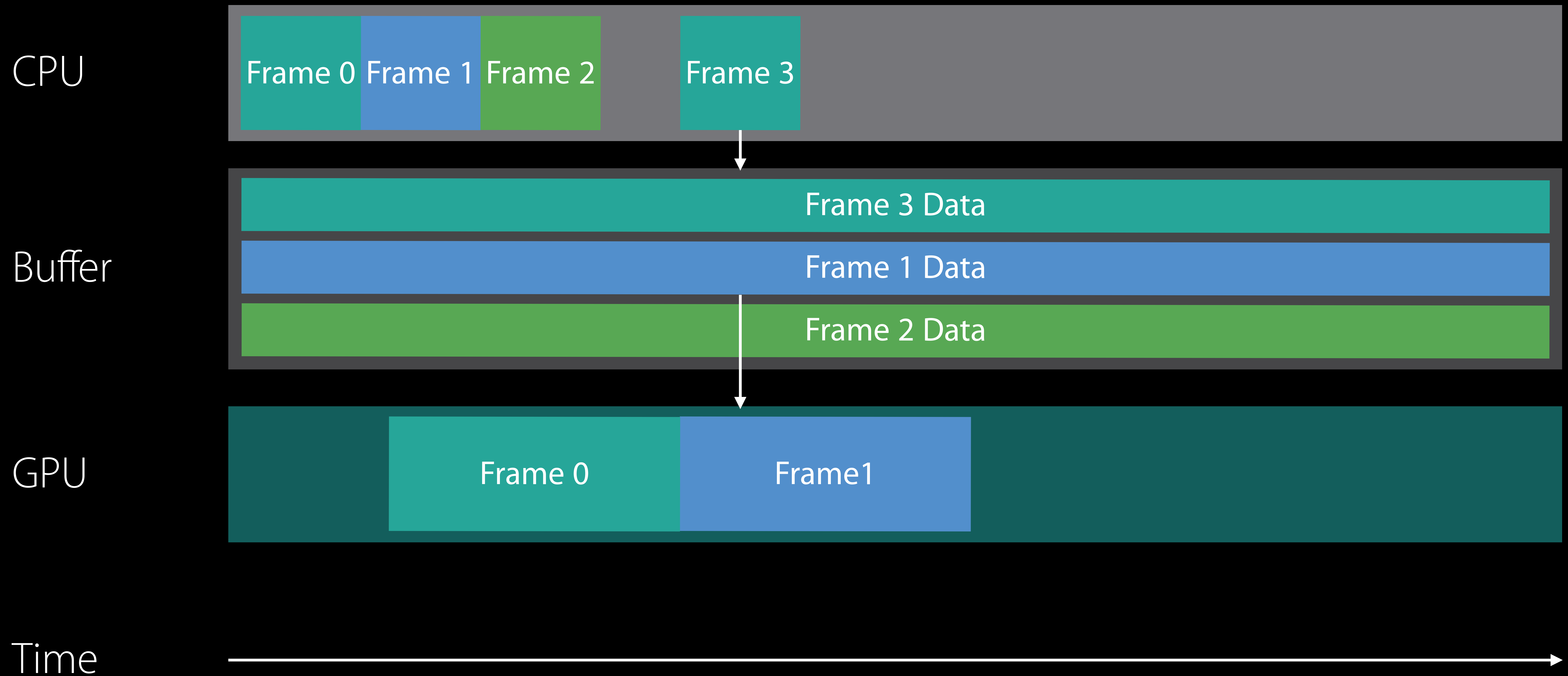
Demo Synchronization



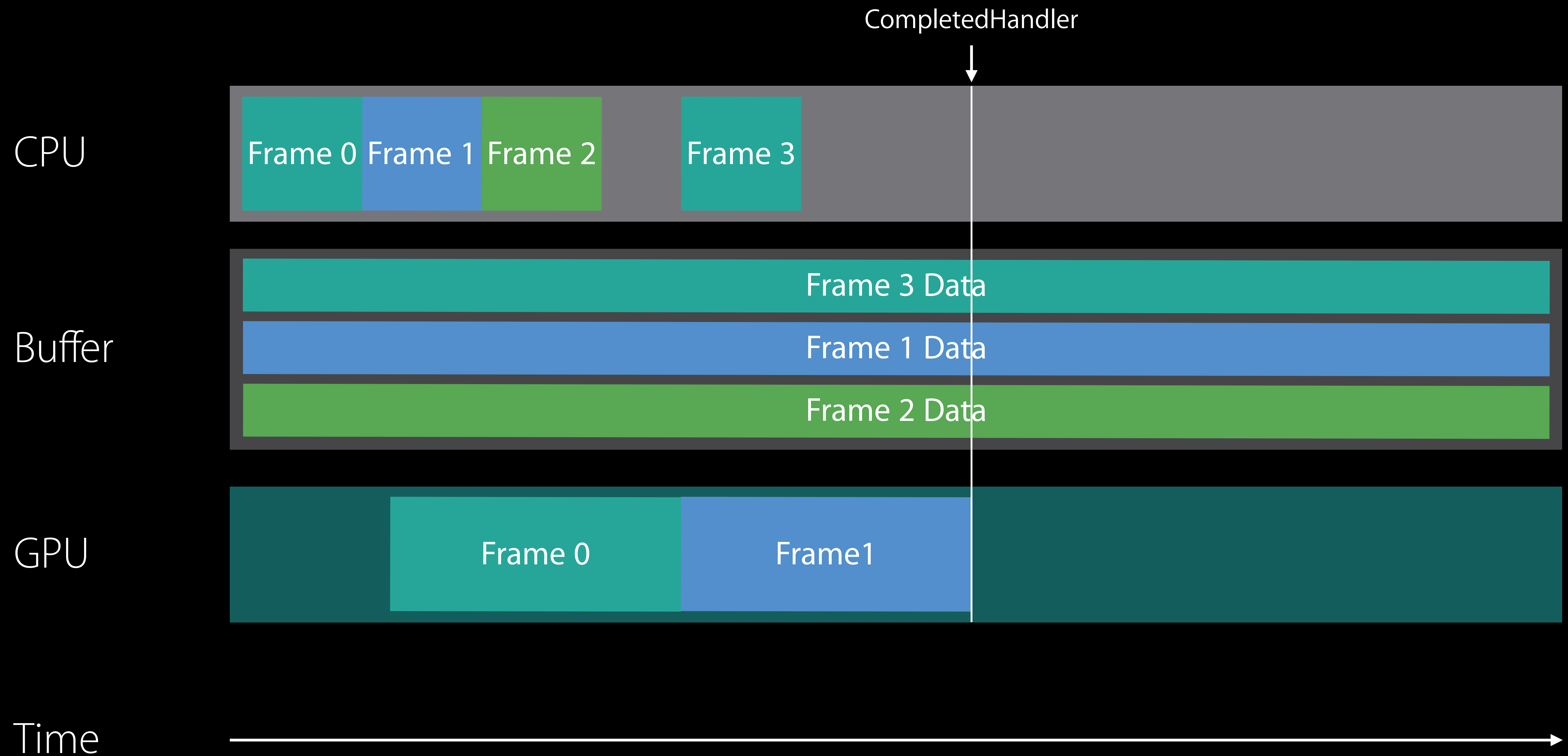
Demo Synchronization



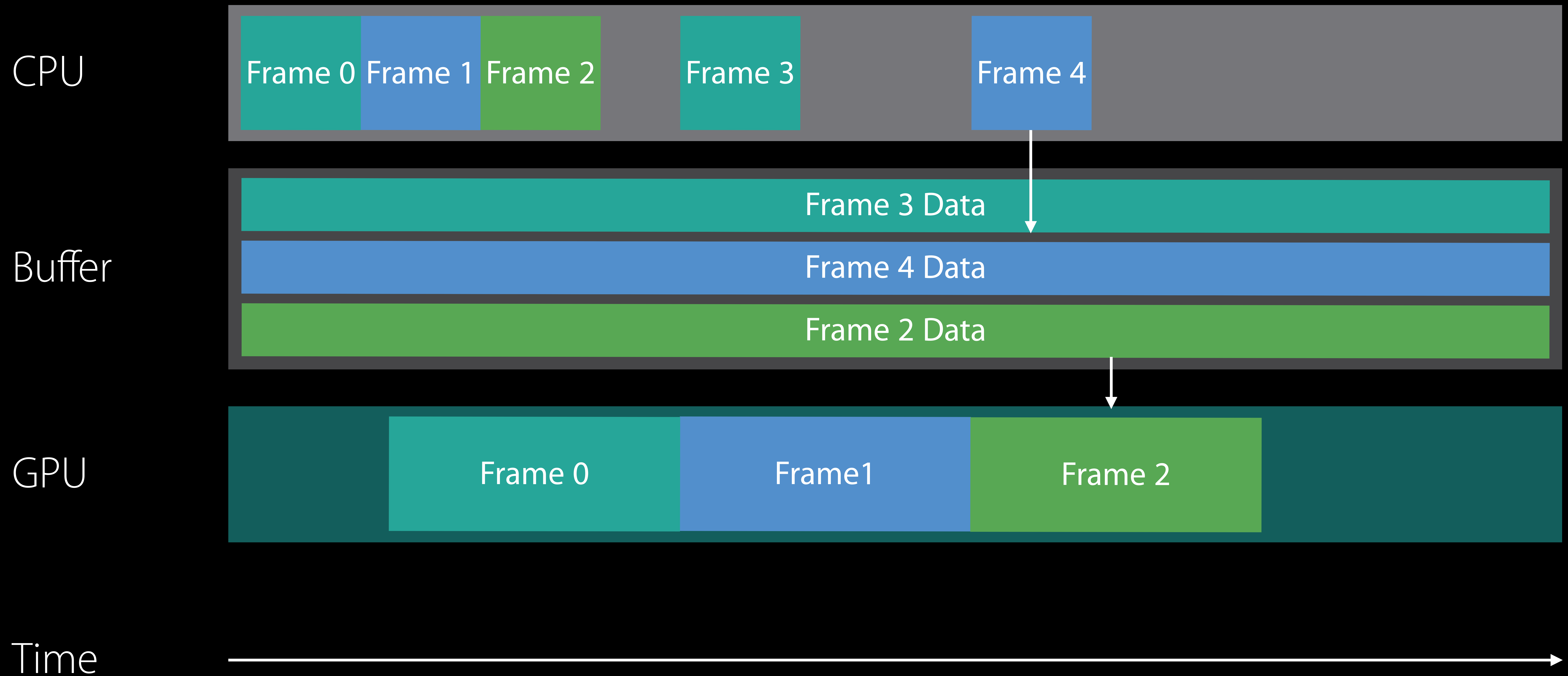
Demo Synchronization



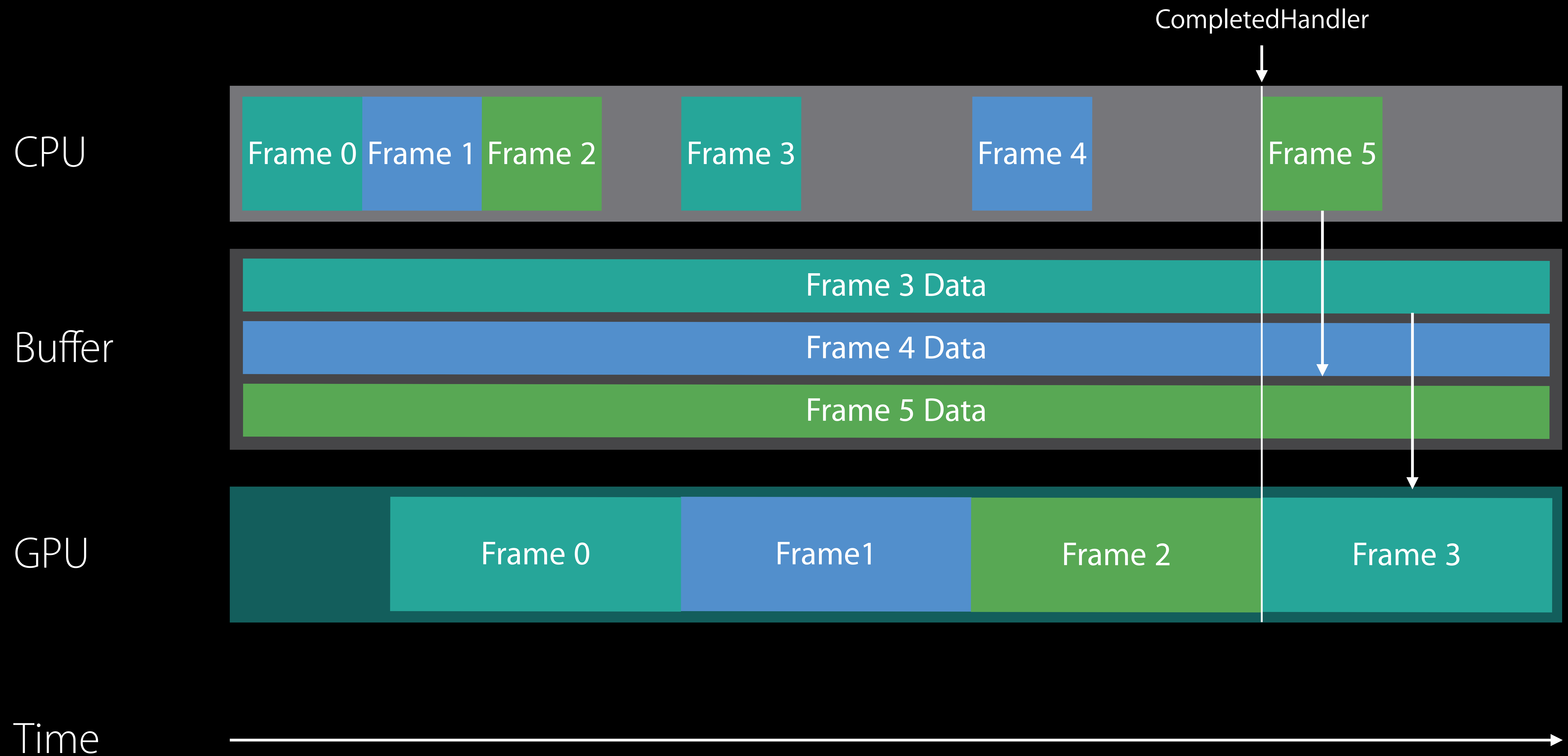
Demo Synchronization



Demo Synchronization



Demo Synchronization



```swift
// Synchronizing access to constant buffers

var semaphore = DispatchSemaphore(value: MAX_FRAMES_IN_FLIGHT)
var constantBuffers = CreateConstantBuffers(MAX_FRAMES_IN_FLIGHT)
var currentConstantBuffer = 0
```

```
// Synchronizing access to constant buffers

var semaphore = DispatchSemaphore(value: MAX_FRAMES_IN_FLIGHT)
var constantBuffers = CreateConstantBuffers(MAX_FRAMES_IN_FLIGHT)
var currentConstantBuffer = 0
```

```
// Synchronizing access to constant buffers

var semaphore = DispatchSemaphore(value: MAX_FRAMES_IN_FLIGHT)
var constantBuffers = CreateConstantBuffers(MAX_FRAMES_IN_FLIGHT)
var currentConstantBuffer = 0
```

```
// Synchronizing access to constant buffers

var semaphore = DispatchSemaphore(value: MAX_FRAMES_IN_FLIGHT)
var constantBuffers = CreateConstantBuffers(MAX_FRAMES_IN_FLIGHT)
var currentConstantBuffer = 0
```

```swift
// Synchronizing access to constant buffers

// Block until resource is available
_ = semaphore.wait(timeout: DispatchTime.distantFuture)
// Grab the current constant buffer
var constantBuf = constantBuffers[currentConstantBuffer]

// Frame continues, fills out command buffers

cmdBuf.addCompletedHandler
{
    // Signal our resource is free
    semaphore.signal()
}
cmdBuf.commit()
// Update the constant buffer index for the next frame
currentConstantBuffer = (currentConstantBuffer+1) % MAX_FRAMES_IN_FLIGHT
```

```
// Synchronizing access to constant buffers

// Block until resource is available
_ = semaphore.wait(timeout: DispatchTime.distantFuture)
// Grab the current constant buffer
var constantBuf = constantBuffers[currentConstantBuffer]

// Frame continues, fills out command buffers

cmdBuf.addCompletedHandler
{
    // Signal our resource is free
    semaphore.signal()
}
cmdBuf.commit()
// Update the constant buffer index for the next frame
currentConstantBuffer = (currentConstantBuffer+1) % MAX_FRAMES_IN_FLIGHT
```

```swift
// Synchronizing access to constant buffers

// Block until resource is available
_ = semaphore.wait(timeout: DispatchTime.distantFuture)
// Grab the current constant buffer
var constantBuf = constantBuffers[currentConstantBuffer]

// Frame continues, fills out command buffers

cmdBuf.addCompletedHandler
{
    // Signal our resource is free
    semaphore.signal()
}
cmdBuf.commit()
// Update the constant buffer index for the next frame
currentConstantBuffer = (currentConstantBuffer+1) % MAX_FRAMES_IN_FLIGHT
```

```swift
// Synchronizing access to constant buffers

// Block until resource is available
_ = semaphore.wait(timeout: DispatchTime.distantFuture)
// Grab the current constant buffer
var constantBuf = constantBuffers[currentConstantBuffer]

// Frame continues, fills out command buffers

cmdBuf.addCompletedHandler
{
    // Signal our resource is free
    semaphore.signal()
}
cmdBuf.commit()
// Update the constant buffer index for the next frame
currentConstantBuffer = (currentConstantBuffer+1) % MAX_FRAMES_IN_FLIGHT
```

```swift
// Synchronizing access to constant buffers

// Block until resource is available
_ = semaphore.wait(timeout: DispatchTime.distantFuture)
// Grab the current constant buffer
var constantBuf = constantBuffers[currentConstantBuffer]

// Frame continues, fills out command buffers

cmdBuf.addCompletedHandler
{
    // Signal our resource is free
    semaphore.signal()
}
cmdBuf.commit()
// Update the constant buffer index for the next frame
currentConstantBuffer = (currentConstantBuffer+1) % MAX_FRAMES_IN_FLIGHT
```

Constant Buffers in the Demo

Array of three buffers

Don't worry about marking them

- With a semaphore, you guarantee frame 0 will be done when frame 3 wants to use that slot again

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

```swift
// Basic Rendering Steps

let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()

encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
encodeMainPass(mainCommandBuffer, constantBuf: constantBuf, ...)

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

Encoding Commands

What do we need to render a cube?

- Cube Geometry
- Pipeline State Object
- Per-frame Data
- Per-object Data

Referencing Data

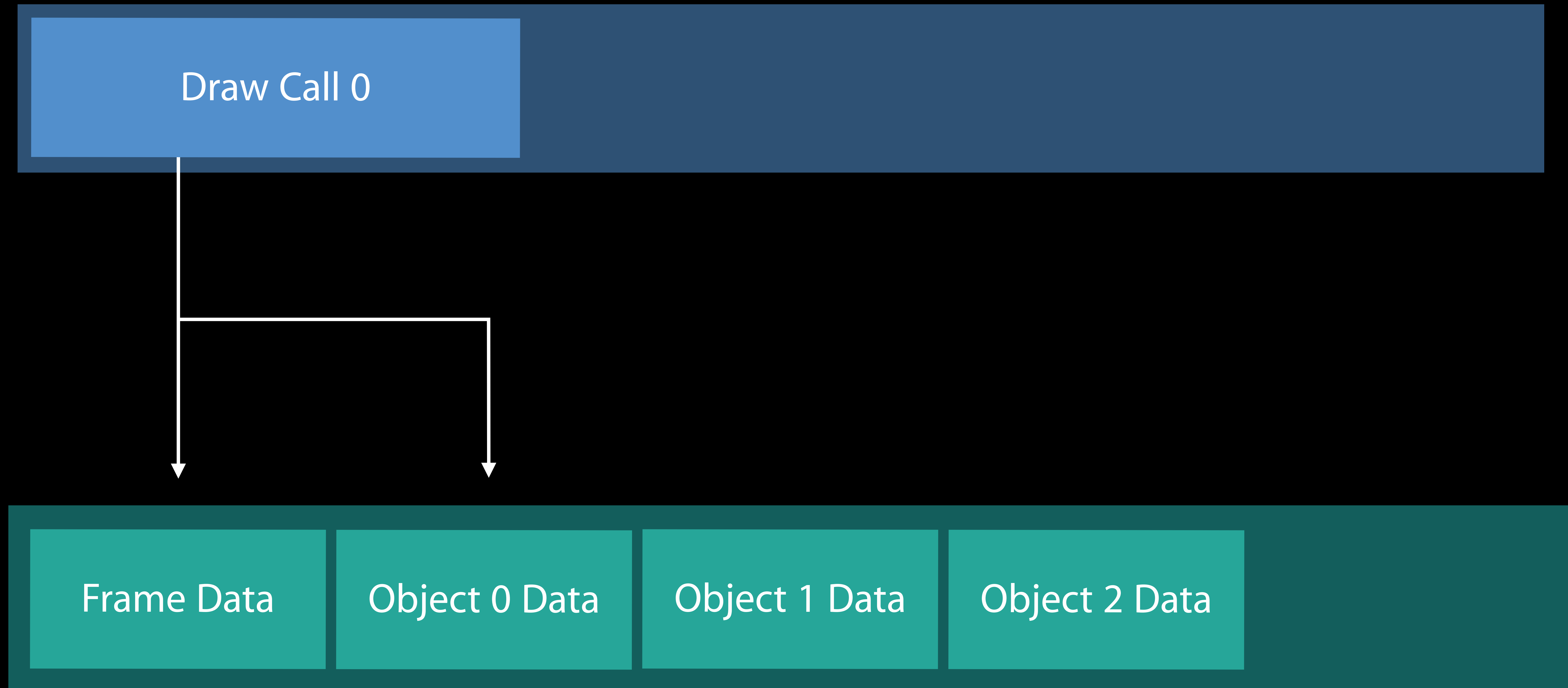
Encoder

Command Buffer

Constant Buffer

Frame Data	Object 0 Data	Object 1 Data	Object 2 Data

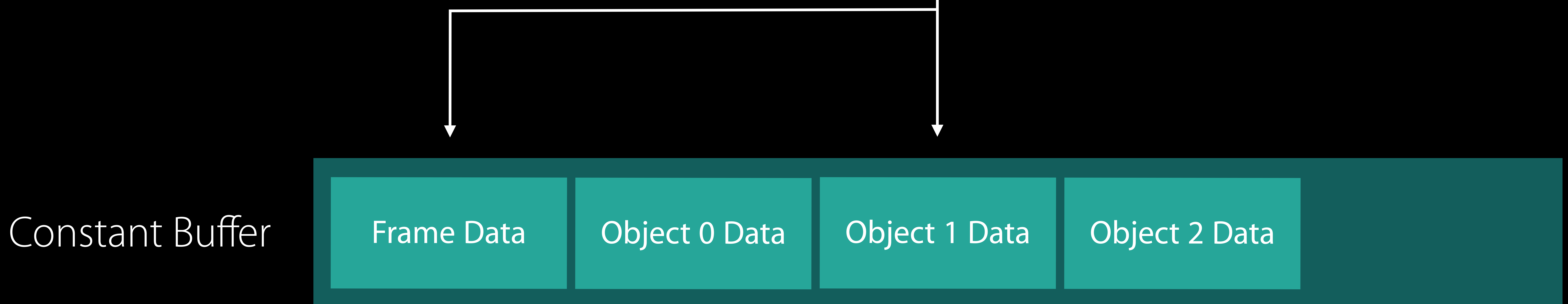
Referencing Data



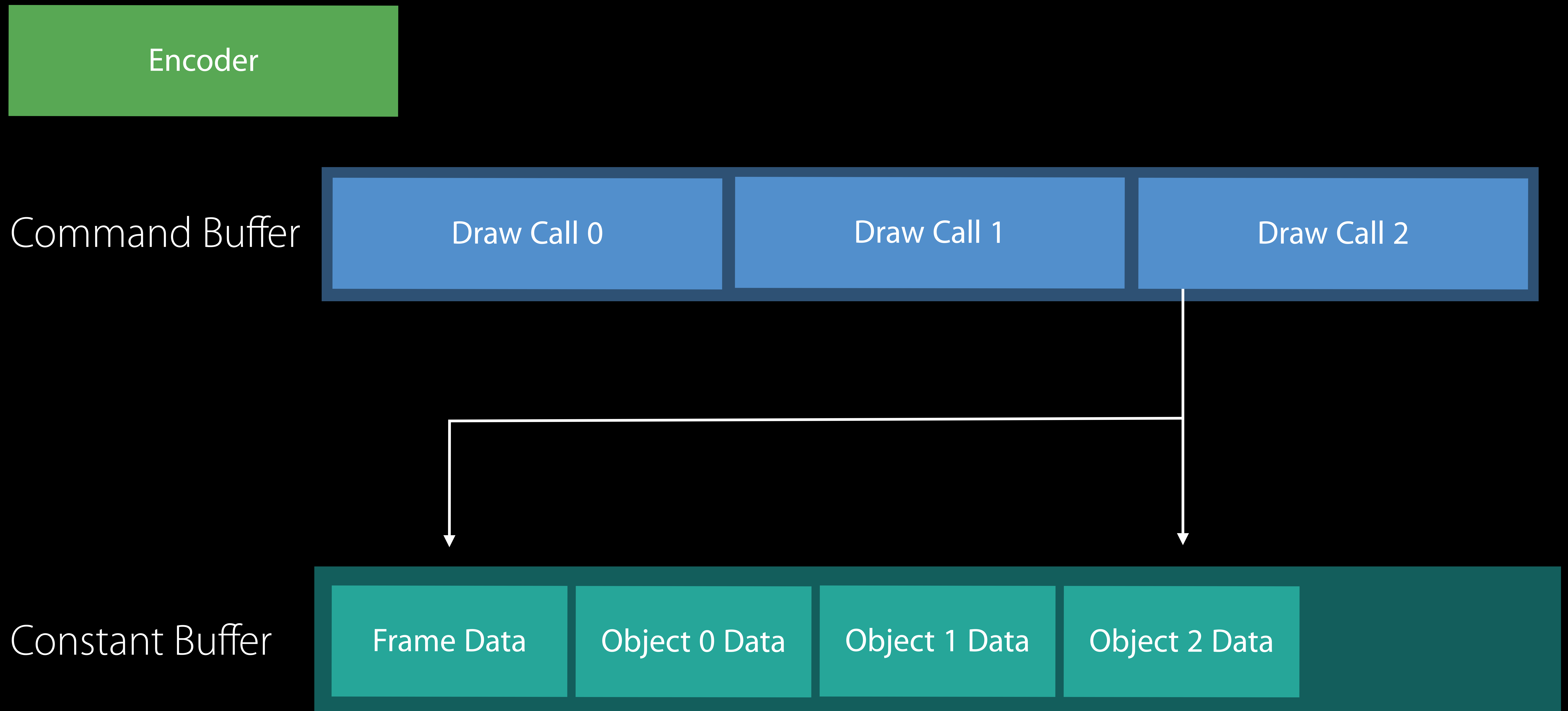
Referencing Data

Command Buffer

Constant Buffer



Referencing Data



Encoding Guidelines

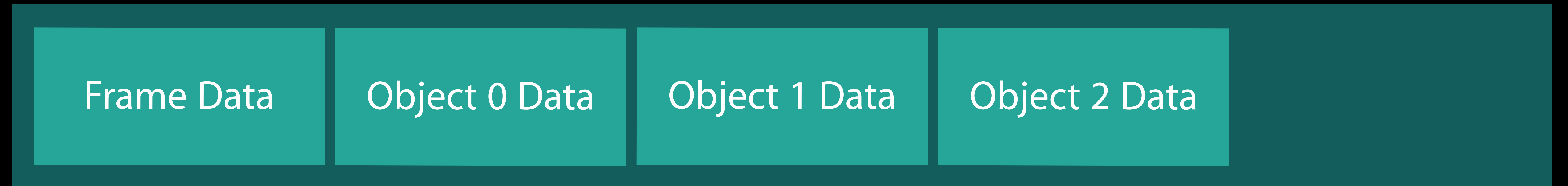
Avoid redundant work

- Reuse as much data as possible
- Avoid redundant state updates
- Avoid redundant argument table updates
- Use `setVertexBufferOffset/setFragmentBufferOffset`

```swift
// Changing Buffer Offsets

setVertexBufferOffset(_ offset: Int, at: Int)
setFragmentBufferOffset(_ offset: Int, at: Int)
```

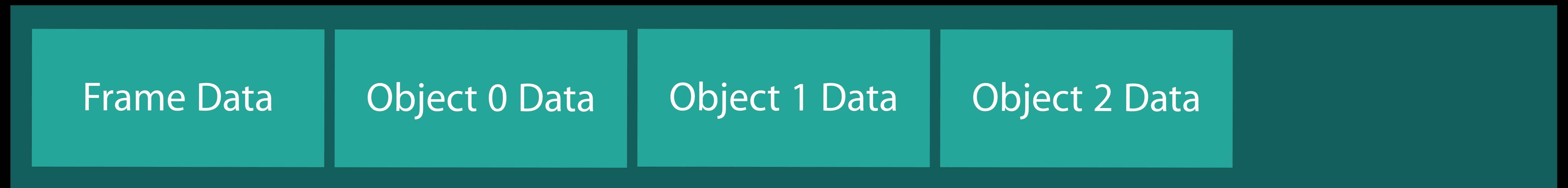
Referencing Data

Constant Buffer



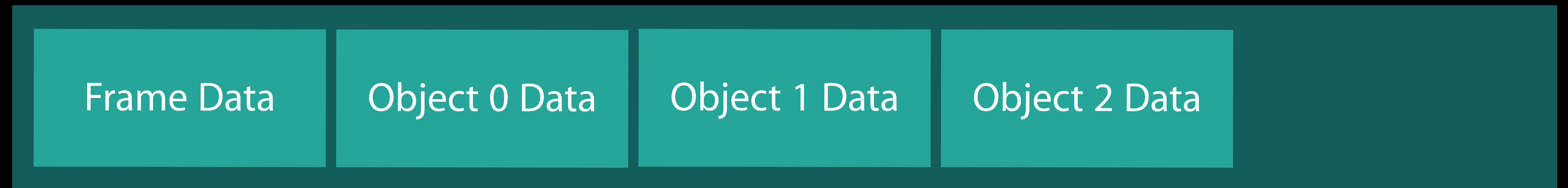
Referencing Data

Constant Buffer



`setVertexBufferOffset`

Referencing Data

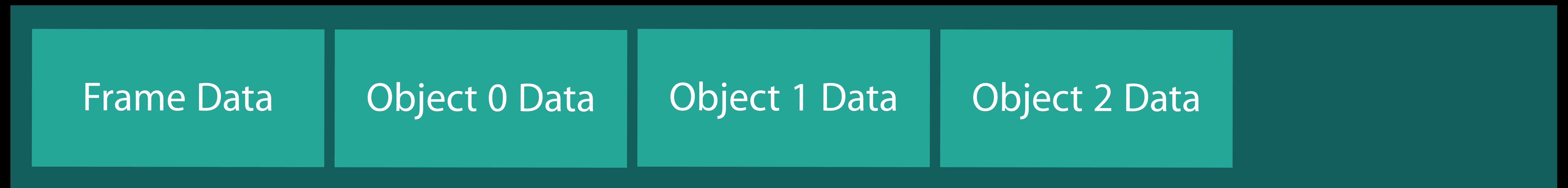


Constant Buffer

`setVertexBufferOffset`

Referencing Data

Constant Buffer



`setVertexBufferOffset`

Referencing Data

Referencing Data

Set per-frame constants

Referencing Data

Set per-frame constants

Set constant buffer

Referencing Data

Set per-frame constants

Set constant buffer

Set geometry buffer

Set pipeline state object

Referencing Data

Set per-frame constants

Set constant buffer

Set geometry buffer

Set pipeline state object

For each object

Referencing Data

Set per-frame constants

Set constant buffer

Set geometry buffer

Set pipeline state object

For each object

- Set offset of constant buffer

Referencing Data

Set per-frame constants

Set constant buffer

Set geometry buffer

Set pipeline state object

For each object

- Set offset of constant buffer
- Issue draw call

```swift
// Avoiding redundant work

// Set geometry and pipeline state object
enc.setVertexBuffer(geometry, offset: 0, at: 0)
enc.setRenderPipelineState(litShadowedPipeline)

// Set constant buffer
enc.setVertexBuffer(constantBuf, offset: 0, at: 1)
enc.setFragmentBuffer(constantBuf, offset: 0, at: 1)

// Set per-frame data
enc.setVertexBuffer(constantBuf, offset: passDataOffset, at: 2)
```

```
// Avoiding redundant work

// Set geometry and pipeline state object
enc.setVertexBuffer(geometry, offset: 0, at: 0)
enc.setRenderPipelineState(litShadowedPipeline)

// Set constant buffer
enc.setVertexBuffer(constantBuf, offset: 0, at: 1)
enc.setFragmentBuffer(constantBuf, offset: 0, at: 1)

// Set per-frame data
enc.setVertexBuffer(constantBuf, offset: passDataOffset, at: 2)
```

```
// Avoiding redundant work

// Set geometry and pipeline state object
enc.setVertexBuffer(geometry, offset: 0, at: 0)
enc.setRenderPipelineState(litShadowedPipeline)

// Set constant buffer
enc.setVertexBuffer(constantBuf, offset: 0, at: 1)
enc.setFragmentBuffer(constantBuf, offset: 0, at: 1)

// Set per-frame data
enc.setVertexBuffer(constantBuf, offset: passDataOffset, at: 2)
```

```
// Avoiding redundant work

// Set geometry and pipeline state object
enc.setVertexBuffer(geometry, offset: 0, at: 0)
enc.setRenderPipelineState(litShadowedPipeline)

// Set constant buffer
enc.setVertexBuffer(constantBuf, offset: 0, at: 1)
enc.setFragmentBuffer(constantBuf, offset: 0, at: 1)

// Set per-frame data
enc.setVertexBuffer(constantBuf, offset: passDataOffset, at: 2)
```

```
var offset = passDataOffset
for index in 0..<objectsToRender
{
    // Set offset into constant buffer
    enc.setVertexBufferOffset(offset, at: 1)
    enc.setFragmentBufferOffset(offset, at: 1)

    // Issue draw call, update offset
    enc.drawIndexedPrimitives(MTLPrimitiveType.triangle, ...)
    offset += strideof(ObjectData)
}
```

```swift
var offset = passDataOffset
for index in 0..<objectsToRender
{
    // Set offset into constant buffer
    enc.setVertexBufferOffset(offset, at: 1)
    enc.setFragmentBufferOffset(offset, at: 1)

    // Issue draw call, update offset
    enc.drawIndexedPrimitives(MTLPrimitiveType.triangle, ...)
    offset += strideof(ObjectData)
}
```

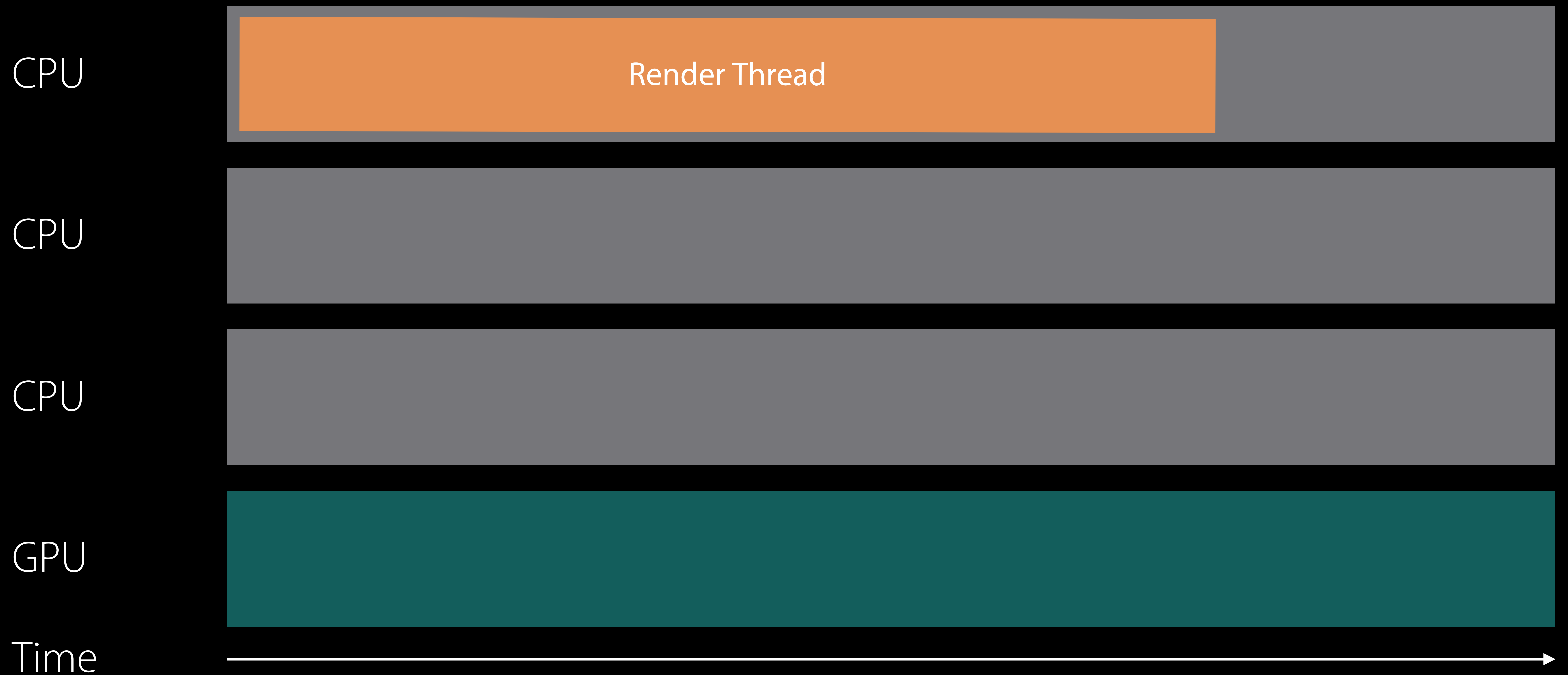
```swift
var offset = passDataOffset
for index in 0..<objectsToRender
{
    // Set offset into constant buffer
    enc.setVertexBufferOffset(offset, at: 1)
    enc.setFragmentBufferOffset(offset, at: 1)

    // Issue draw call, update offset
    enc.drawIndexedPrimitives(MTLPrimitiveType.triangle, ...)
    offset += strideof(ObjectData)
}
```

Command Encoding and Submission

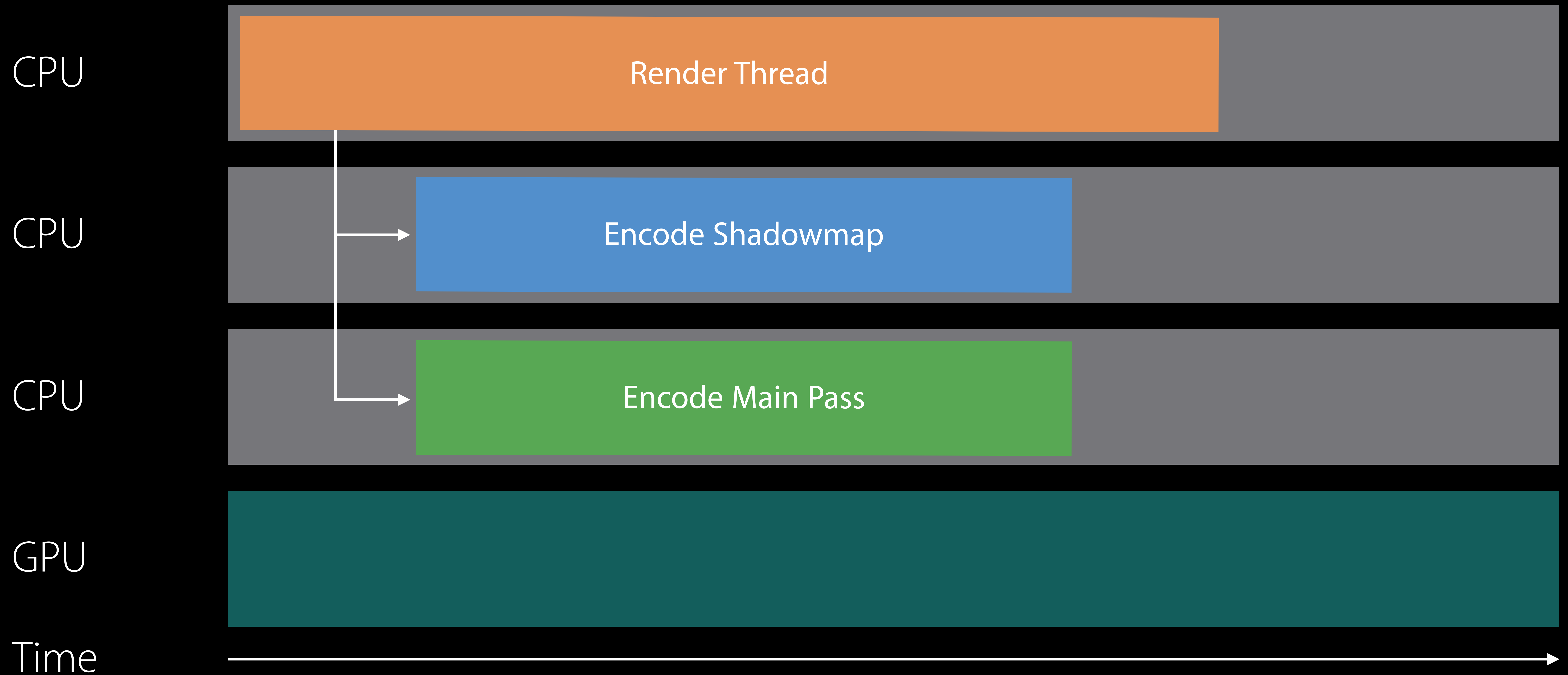
Still very linear…

Command buffers can be constructed in parallel!

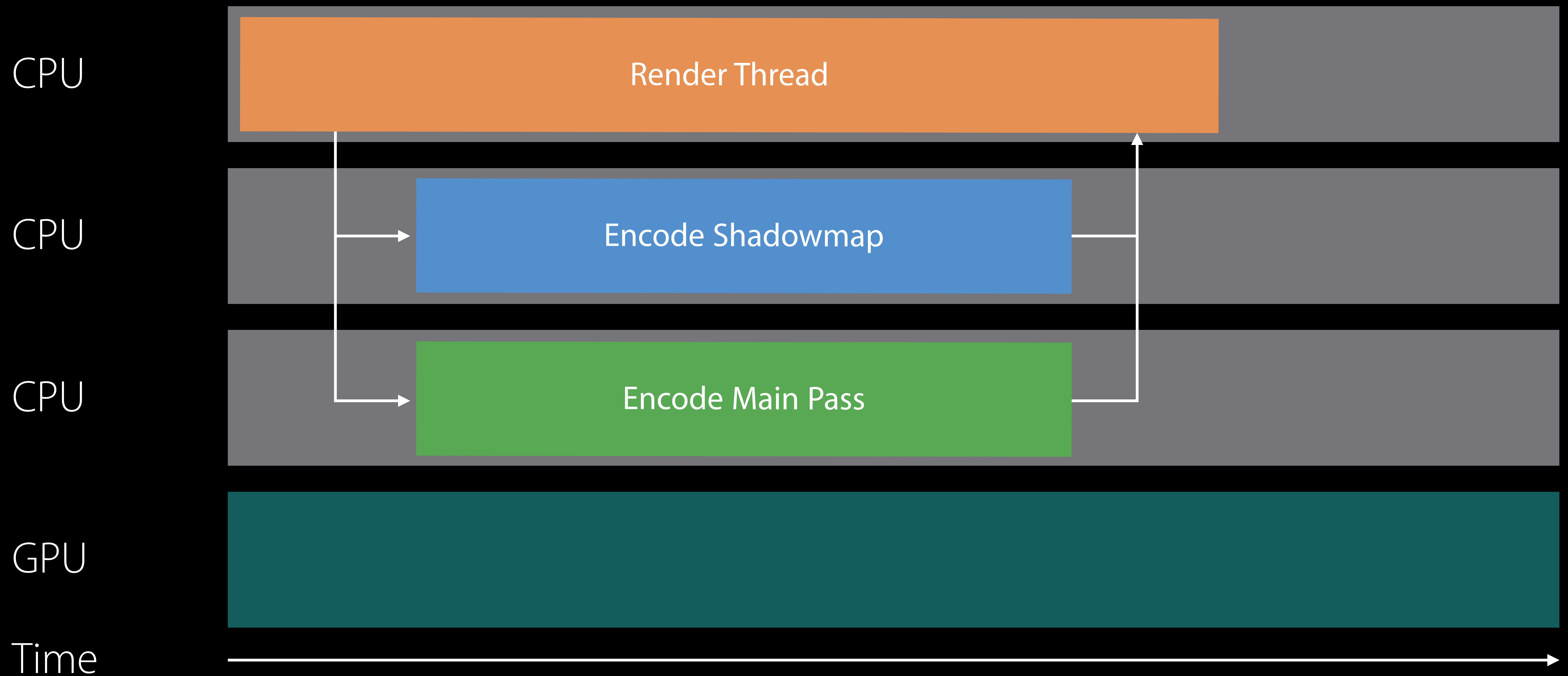
Ideal Scene Encoding



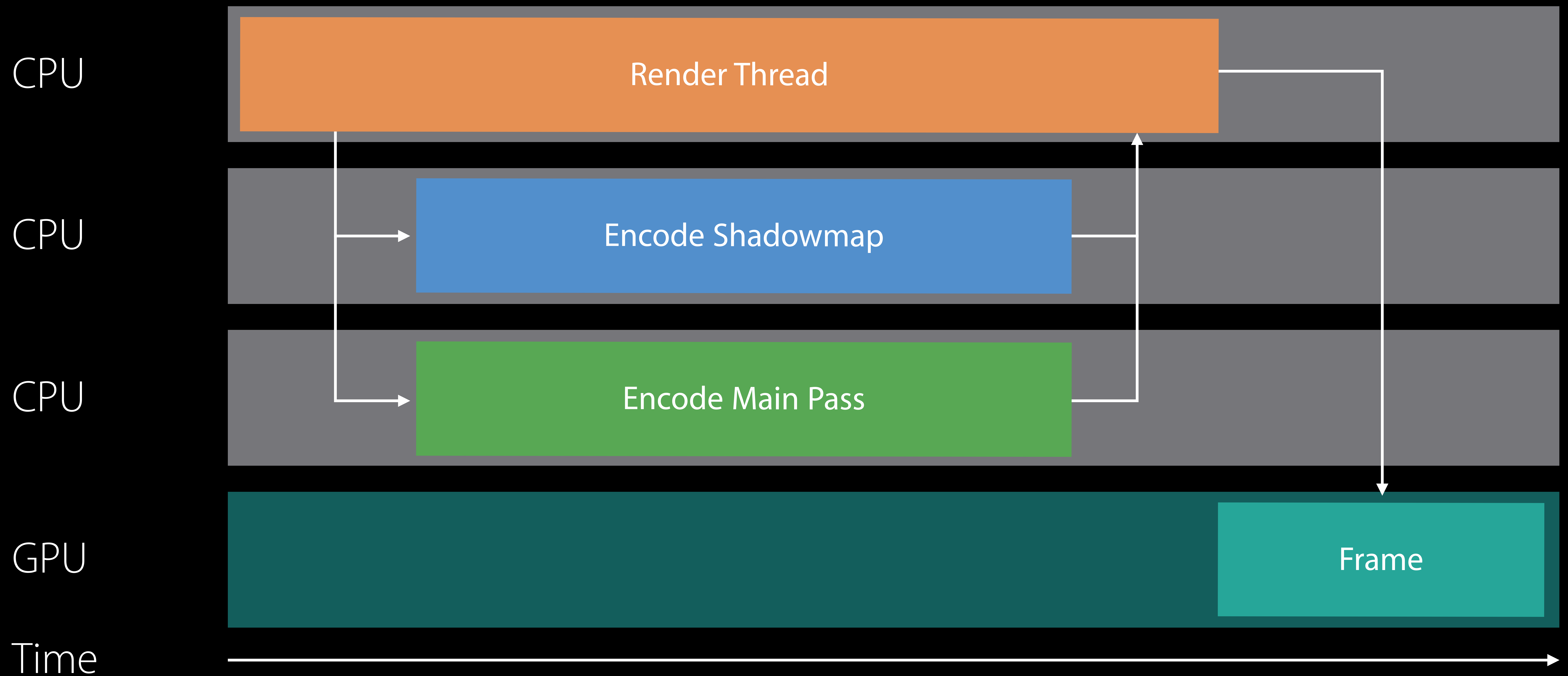
Ideal Scene Encoding



Ideal Scene Encoding



Ideal Scene Encoding



```
// Creating a parallel dispatch queue

var dispatchQueue = DispatchQueue(label: “queue", attributes:.concurrent)
```

```swift
let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()
// Dispatch encoding in parallel
dispatchQueue.async
{
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}

dispatchQueue.async
{
    self.encodeMainPass(mainCommandBuffer, constantBuf: constantBuf, ...)
}
// Block until encoding is complete
__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

```swift
let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()
// Dispatch encoding in parallel
dispatchQueue.async
{
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}

dispatchQueue.async
{
    self.encodeMainPass(mainCommandBuffer, constantBuf: constantBuf, ...)
}
// Block until encoding is complete
__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

```swift
let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()
// Dispatch encoding in parallel
dispatchQueue.async
{
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}

dispatchQueue.async
{
    self.encodeMainPass(mainCommandBuffer, constantBuf: constantBuf, ...)
}
// Block until encoding is complete
__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

```
let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()
// Dispatch encoding in parallel
dispatchQueue.async
{
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}

dispatchQueue.async
{
    self.encodeMainPass(mainCommandBuffer, constantBuf: constantBuf, ...)
}
// Block until encoding is complete
__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

```
let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()
// Dispatch encoding in parallel
dispatchQueue.async
{
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}

dispatchQueue.async
{
    self.encodeMainPass(mainCommandBuffer, constantBuf: constantBuf, ...)
}
// Block until encoding is complete
__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

Closures Capture References

Closures capture `self`

Explicitly capture members to ensure correctness

```swift
// Captures a reference to self

dispatchQueue.async
{
    self.encodeShadowPass(cmdBuf, constantBuf:constantBuffers[self.constantBufferSlot],...)
}

// Capture a reference to constantBuf instead!

let constantBuf = constantBuffers[self.constantBufferSlot]
dispatchQueue.async {
    self.encodeMainPass(cmdBuf, constantBuf:constantBuf, ...)
}
```

```swift
// Captures a reference to self

dispatchQueue.async
{
    self.encodeShadowPass(cmdBuf, constantBuf:constantBuffers[self.constantBufferSlot],...)
}

// Capture a reference to constantBuf instead!

let constantBuf = constantBuffers[self.constantBufferSlot]
dispatchQueue.async {
    self.encodeMainPass(cmdBuf, constantBuf:constantBuf, ...)
}
```

```
// Captures a reference to self

dispatchQueue.async
{
    self.encodeShadowPass(cmdBuf, constantBuf:constantBuffers[self.constantBufferSlot],...)
}
```

```
// Capture a reference to constantBuf instead!

let constantBuf = constantBuffers[self.constantBufferSlot]
dispatchQueue.async {
    self.encodeMainPass(cmdBuf, constantBuf:constantBuf, ...)
}
```

```
// Captures a reference to self

dispatchQueue.async
{
    self.encodeShadowPass(cmdBuf, constantBuf:constantBuffers[self.constantBufferSlot],...)
}
```

```
// Capture a reference to constantBuf instead!

let constantBuf = constantBuffers[self.constantBufferSlot]
dispatchQueue.async {
    self.encodeMainPass(cmdBuf, constantBuf:constantBuf, ...)
}
```

```swift
let constantBuf = constantBuffers[self.constantBufferSlot]

let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
}

__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

```swift
let constantBuf = constantBuffers[self.constantBufferSlot]

let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
}

__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

```swift
let constantBuf = constantBuffers[self.constantBufferSlot]

let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
}

__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

```swift
let constantBuf = constantBuffers[self.constantBufferSlot]

let shadowCommandBuffer = metalQueue.commandBuffer()
let mainCommandBuffer = metalQueue.commandBuffer()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
}

__dispatch_barrier_sync(dispatchQueue) { }

shadowCommandBuffer.commit()
mainCommandBuffer.commit()
```

Explicit Command Buffer Ordering

`enqueue()` enforces ordering

Allows you to `commit()` command buffers from multiple threads

```
// Enqueue command buffers in order

shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
    // Commit command buffer within dispatch
    shadowCommandBuffer.commit()
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Commit command buffer within dispatch
    mainCommandBuffer.commit()
}
```

```swift
// Enqueue command buffers in order

shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
    // Commit command buffer within dispatch
    shadowCommandBuffer.commit()
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Commit command buffer within dispatch
    mainCommandBuffer.commit()
}
```

```swift
// Enqueue command buffers in order

shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
    // Commit command buffer within dispatch
    shadowCommandBuffer.commit()
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Commit command buffer within dispatch
    mainCommandBuffer.commit()
}
```

```
// Enqueue command buffers in order

shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    self.encodeShadowPass(shadowCommandBuffer, constantBuf: constantBuf, ...)
    // Commit command buffer within dispatch
    shadowCommandBuffer.commit()
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Commit command buffer within dispatch
    mainCommandBuffer.commit()
}
```

Synchronization

How do we synchronize this?

```swift
// Block until a resource is available

_ = self.semaphore.wait(timeout: DispatchTime.distantFuture)

let constantBuf = constantBuffers[constantBufferSlot]
shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    // Dispatch shadow pass
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Add completion handler to the last command buffer
    mainCommandBuffer.addCompletedHandler {
        self.semaphore.signal()
    }
    mainCommandBuffer.commit()
}
```

```
// Block until a resource is available

_ = self.semaphore.wait(timeout: DispatchTime.distantFuture)

let constantBuf = constantBuffers[constantBufferSlot]
shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    // Dispatch shadow pass
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Add completion handler to the last command buffer
    mainCommandBuffer.addCompletedHandler {
        self.semaphore.signal()
    }
    mainCommandBuffer.commit()
}
```

```swift
// Block until a resource is available

_ = self.semaphore.wait(timeout: DispatchTime.distantFuture)

let constantBuf = constantBuffers[constantBufferSlot]
shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    // Dispatch shadow pass
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Add completion handler to the last command buffer
    mainCommandBuffer.addCompletedHandler {
        self.semaphore.signal()
    }
    mainCommandBuffer.commit()
}
```

```swift
// Block until a resource is available

_ = self.semaphore.wait(timeout: DispatchTime.distantFuture)

let constantBuf = constantBuffers[constantBufferSlot]
shadowCommandBuffer.enqueue()
mainCommandBuffer.enqueue()

dispatchQueue.async {
    // Dispatch shadow pass
}
dispatchQueue.async {
    self.encodeMainPass(mainCommandBuffer, constantBuf:constantBuf, ...)
    // Add completion handler to the last command buffer
    mainCommandBuffer.addCompletedHandler {
        self.semaphore.signal()
    }
    mainCommandBuffer.commit()
}
```

The Recipe

Wait on semaphore

Select current constant buffer

Write data into constant buffer

Encode commands into command buffers

Add a completion handler to signal semaphore

Commit command buffers

Demo

Summary

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

More Information

https://developer.apple.com/wwdc16/603

Related Sessions

What's New in Metal, Part 1	Pacific Heights	Wednesday 11:00AM
What's New in Metal, Part 2	Pacific Heights	Wednesday 1:40PM
Advanced Metal Shader Optimization	Pacific Heights	Wednesday 3:00PM

Labs

Metal Lab	Graphics, Games, and Media Lab A	Tuesday 4:00PM
Metal Lab	Graphics, Games, and Media Lab A	Thursday 12:00PM

WWDC16