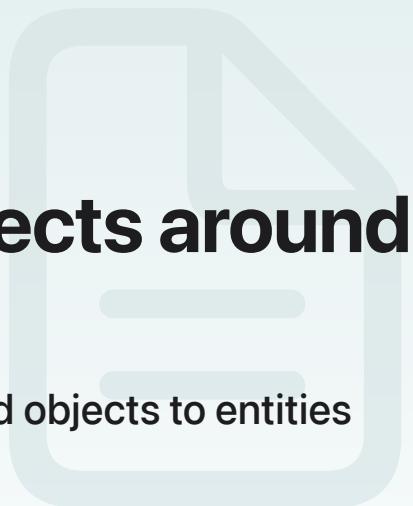


[RealityKit](#) / Passing Metal command objects around your application

## Article

# Passing Metal command objects around your application

Build a system that creates and passes Metal command objects to entities dispatching Metal compute shaders.



## Overview

To dispatch a Metal compute shader function, you need a command queue, a command buffer, and a compute command encoder. Creating these command objects comes at a cost, so avoid making them on demand whenever you need to dispatch a compute shader function (see [Setting Up a Command Structure](#)). Instead, for single-threaded apps, employ a single command queue for the entirety of the application, as well as a single command buffer and compute command encoder for all compute commands that you dispatch in every frame.

The [Generating interactive geometry with RealityKit](#) sample demonstrates one possible approach to managing the life cycle and injection of these command objects throughout an application. Leverage RealityKit's [Entity Component System](#) (ECS) to pass an [MTLCommandBuffer](#) and an [MTLComputeCommandEncoder](#) to each entity dispatching compute shaders in every frame, while maintaining a single global [MTLCommandQueue](#) for the entire application.

## Define a compute system protocol

Start by creating a structure that contains the context necessary to dispatch compute shader functions in every frame:

```
/// A structure containing the context a `ComputeSystem` needs to dispatch compute commands.
struct ComputeUpdateContext {
    /// The number of seconds elapsed since the last frame.
    let deltaTime: TimeInterval
```

```
    /// The command buffer for the current frame.  
    let commandBuffer: MTLCommandBuffer  
    /// The compute command encoder for the current frame.  
    let computeEncoder: MTLComputeCommandEncoder  
}
```

You can choose not to include the `deltaTime` property in your structure, or you can add additional properties, such as [SceneUpdateContext](#).

Next, define a protocol with an update method that takes `ComputeUpdateContext` as a parameter:

```
/// A protocol that enables its adoptees to dispatch their own compute commands in each frame.  
protocol ComputeSystem {  
    @MainActor  
    func update(computeContext: ComputeUpdateContext)  
}
```

## Dispatch compute systems with RealityKit's ECS

Create a component that holds a `ComputeSystem`:

```
/// A component that contains a `ComputeSystem`.  
struct ComputeSystemComponent: Component {  
    let computeSystem: ComputeSystem  
}
```

Then, create a custom [System](#) that finds all entities with a `ComputeSystemComponent` in every frame and passes that frame's `ComputeUpdateContext` to their `ComputeSystem` instances:

```
/// A class that updates the `ComputeSystem` of each `ComputeSystemComponent` with each frame.  
class ComputeDispatchSystem: System {  
    /// The application's command queue.  
    ///  
    /// A single, global command queue to use throughout the entire application.  
    static let commandQueue: MTLCommandQueue? = makeCommandQueue(labeled: "Compute Dispatch")  
  
    /// The query this system uses to get all entities with a `ComputeSystemComponent`.  
    let query = EntityQuery(where: .has(ComputeSystemComponent.self))
```

```

required init(scene: Scene) { }

/// Updates all compute systems with the current frame's `ComputeUpdateContext`.
func update(context: SceneUpdateContext) {
    // Get all entities with a `ComputeSystemComponent` in every frame.
    let computeSystemEntities = context.entities(matching: query, updatingSystems: true)

    // Create the command buffer and compute encoder responsible for dispatching.
    guard let commandBuffer = Self.commandQueue?.makeCommandBuffer(),
          let computeEncoder = commandBuffer.makeComputeCommandEncoder() else {
        return
    }

    // Enqueue the command buffer.
    commandBuffer.enqueue()

    // Dispatch all compute systems to encode their compute commands.
    let computeContext = ComputeUpdateContext(deltaTime: context.deltaTime,
                                                commandBuffer: commandBuffer,
                                                computeEncoder: computeEncoder)
    for computeSystemEntity in computeSystemEntities {
        if let computeSystemComponent = computeSystemEntity.components[ComputeSystemComponent.self] {
            computeSystemComponent.computeSystem.update(computeContext: computeContext)
        }
    }

    // Stop encoding compute commands and commit them to run on the GPU.
    computeEncoder.endEncoding()
    commandBuffer.commit()
}
}

```

In this example, a helper method assists in the creation of the Metal command queue:

```

/// The device Metal selects as the default.
let metalDevice: MTLDevice? = MTLCreateSystemDefaultDevice()

/// Makes a command queue with the given label.
func makeCommandQueue(labeled label: String) -> MTLCommandQueue? {
    guard let metalDevice, let queue = metalDevice.makeCommandQueue() else {
        return nil
    }
    queue.label = label
}

```

```
    return queue
```

```
}
```

# Create a custom compute system

You can dispatch your compute shader functions in every frame by creating a custom Compute System and implementing its update method:

```
struct MyComputeSystem: ComputeSystem {  
    func update(computeContext: ComputeUpdateContext) {  
        // Dispatch compute shader functions here.  
    }  
}
```

Be sure to register the `ComputeDispatchSystem` so that the update method fires every frame:

```
ComputeDispatchSystem.registerSystem()
```

Finally, attach your custom `ComputeSystem` to an entity with a `ComputeSystemComponent`:

```
let myComputeSystem = MyComputeSystem()  
let myComputeEntity = Entity()  
myComputeEntity.components.set(ComputeSystemComponent(computeSystem: myComputeSystem))
```

## See Also

### Performance improvements

#### Improving the Performance of a RealityKit App

Measure CPU and GPU utilization to find ways to improve your app's performance.

#### Reducing GPU Utilization in Your RealityKit App

Prevent the GPU from limiting your app's frame rate by reducing the complexity of your render.

#### Reducing CPU Utilization in Your RealityKit App

Target specific CPU metrics with adjustments to your app and its content.

{ } Construct an immersive environment for visionOS

Build efficient custom worlds for your app.

## protocol Resource

A shared resource you use to configure a component, like a material, mesh, or texture.