Sample Code

Reading pixel data from a drawable texture

Access texture data from the CPU by copying it to a buffer.

Download

iOS 12.0+ | iPadOS 12.0+ | macOS 10.13+ | Xcode 12.5+

Overview

Metal optimizes textures for fast access by the GPU, but it doesn't allow you to directly access a texture's contents from the CPU. When your app code needs to change or read a texture's contents, you use Metal to copy data between textures and CPU-accessible memory — either system memory or a Metal buffer allocated using shared storage. This sample configures drawable textures for read access and copies rendered pixel data from those textures to a Metal buffer.

Run the sample, then tap or click on a single point to read the pixel data stored at that point. Alternatively, drag out a rectangle to capture pixel data for a region on the screen. The sample converts your selection to a rectangle in the drawable texture's coordinate system. Next, it renders an image to the texture. Finally, it copies the pixel data from the selected rectangle into a buffer for the sample to process further.

Configure the drawable texture for read access

By default, MetalKit views create drawable textures for rendering only, so other Metal commands can't access the texture. The code below creates a view whose textures include read access. Because the sample needs to get a texture whenever the user selects part of the view, the code configures the view's Metal layer to wait indefinitely for a new drawable.

```
_view.framebufferOnly = NO;
((CAMetalLayer*)_view.layer).allowsNextDrawableTimeout = NO;
_view.colorPixelFormat = MTLPixelFormatBGRA8Unorm;
```

Because configuring the drawable textures for read access means that Metal may not apply some optimizations, only change the drawable configuration when necessary. For similar reasons, don't configure the view to wait indefinitely in performance-sensitive apps.

Determine which pixels to copy

The `AAPLViewController` class manages user interaction. When a user interacts with a view, AppKit and UIKit send events with positions specified in the view's coordinate system. To determine which pixels to copy from the Metal drawable texture, the app transforms these view coordinates into the Metal texture coordinate system.

Because of differences in graphics coordinate systems and APIs, the code to convert between view coordinates and texture coordinates varies by platform.

In macOS, the code calls the `pointToBacking:` method on the view to convert a position into a pixel location in the backing store, and then applies a coordinate transformation to adjust the origin and the y-axis.

```
CGPoint bottomUpPixelPosition = [_view convertPointToBacking:event.locationInWindow
CGPoint topDownPixelPosition = CGPointMake(bottomUpPixelPosition.x,
                                           _view.drawableSize.height - bottomUpPixel
```

In iOS, the app reads the view's `contentScaleFactor` and applies a scaling transform to the view coordinate. iOS views and Metal textures use the same coordinate conventions, so the code doesn't move the origin or change the y-axis orientation.

```
- (CGPoint)pointToBacking:(CGPoint)point
{
    CGFloat scale = _view.contentScaleFactor;

    CGPoint pixel;

    pixel.x = point.x * scale;
    pixel.y = point.y * scale;

    // Round the pixel values down to put them on a well-defined grid.
    pixel.x = (int64_t)pixel.x;
    pixel.y = (int64_t)pixel.y;

    // Add .5 to move to the center of the pixel.
    pixel.x += 0.5f;
    pixel.y += 0.5f;

    return pixel;
}
```

Render the pixel data

When the user selects a rectangle in the view, the view controller calls the `renderAndReadPixelsFromView:withRegion` method to render the drawable's contents and copy them to a Metal buffer.

It creates a new command buffer and calls a utility method to encode a render pass. The specific rendered image isn't important to this sample.

```
id<MTLCommandBuffer> commandBuffer = [_commandQueue commandBuffer];

// Encode a render pass to render the image to the drawable texture.
[self drawScene:view withCommandBuffer:commandBuffer];
```

After encoding the render pass, it calls another method to encode commands to copy a section of the rendered texture. The sample encodes the commands to copy the pixel data before presenting the drawable texture because the system discards the texture's contents after presenting it.

```
id<MTLTexture> readTexture = view.currentDrawable.texture;

MTLOrigin readOrigin = MTLOriginMake(region.origin.x, region.origin.y, 0);
MTLSize readSize = MTLSizeMake(region.size.width, region.size.height, 1);

const id<MTLBuffer> pixelBuffer = [self readPixelsWithCommandBuffer:commandBuffer
                                                        fromTexture:readTexture
                                                           atOrigin:readOrigin
                                                           withSize:readSize];
```

Copy pixel data to a buffer

The renderer's `readPixelsWithCommandBuffer:fromTexture:atOrigin:withSize:` method encodes the commands to copy the texture. Because the sample passes the same command buffer into this method, Metal encodes these new commands after the render pass. Metal automatically manages the dependencies on the destination texture, and ensures that rendering completes before copying the texture data.

First, the method allocates a Metal buffer to hold the pixel data. It calculates the size of the buffer by multiplying the size of one pixel in bytes by the region's width and height. Similarly, the code calculates the number of bytes per row, which the code needs later when copying the data. The sample doesn't add any padding at the end of rows. Then, it calls the Metal device object to create the new Metal buffer, specifying a shared storage mode so that the app can read the buffer's contents afterwards.

```
NSUInteger bytesPerPixel = sizeofPixelFormat(texture.pixelFormat);
NSUInteger bytesPerRow   = size.width * bytesPerPixel;
NSUInteger bytesPerImage = size.height * bytesPerRow;

_readBuffer = [texture.device newBufferWithLength:bytesPerImage options:MTLResourceS
```

Next, the method creates an `MTLBlitCommandEncoder`, which provides commands that copy data between Metal resources, fill resources with data, and perform other similar resource-related tasks that don't directly involve computation or rendering. The sample encodes a blit command to copy the texture data to the beginning of the new buffer. It then ends the blit pass.

```
id <MTLBlitCommandEncoder> blitEncoder = [commandBuffer blitCommandEncoder];

[blitEncoder copyFromTexture:texture
                 sourceSlice:0
                 sourceLevel:0
                sourceOrigin:origin
                  sourceSize:size
                    toBuffer:_readBuffer
           destinationOffset:0
      destinationBytesPerRow:bytesPerRow
    destinationBytesPerImage:bytesPerImage];

[blitEncoder endEncoding];
```

Finally, it commits the command buffer and calls `waitUntilCompleted()` to immediately wait for the GPU to finish executing the rendering and blit commands. After this call returns control to the method, the buffer contains the requested pixel data. In a real-time app, synchronizing commands unnecessarily reduces parallelism between the CPU and GPU; this sample synchronizes in this way to simplify the code.

```
[commandBuffer commit];

// The app must wait for the GPU to complete the blit pass before it can
// read data from _readBuffer.
[commandBuffer waitUntilCompleted];
```

Read the pixels from the buffer

The app calls the buffer's `contents()` method to get a pointer to the pixel data.

```
AAPLPixelBGRA8Unorm *pixels = (AAPLPixelBGRA8Unorm *)pixelBuffer.contents;
```

The sample copies the buffer's data into an `NSData` object and passes it to another method to initialize an `AAPLImage` object. For more information on `AAPLImage`, see Creating and sampling textures.

```
// Create an `NSData` object and initialize it with the pixel data.
// Use the CPU to copy the pixel data from the `pixelBuffer.contents`
// pointer to `data`.
NSData *data = [[NSData alloc] initWithBytes:pixels length:pixelBuffer.length];

// Create a new image from the pixel data.
AAPLImage *image = [[AAPLImage alloc] initWithBGRA8UnormData:data
                                                        width:readSize.width
                                                       height:readSize.height];
```

The renderer returns this image object to the view controller for further processing. The view controller's behavior varies depending on the operating system. In MacOS, the sample writes the image to the file `~/Desktop/ReadPixelsImage.tga`, while in iOS, the sample adds the image to the Photos library. The view controller performs this processing without using Metal, so the steps it takes aren't important to this sample.

See Also

Textures

{} Processing a texture in a compute function

Create textures by running copy and dispatch commands in a compute pass on a GPU.

{} Creating and sampling textures

Load image data into a texture and apply it to a quadrangle.

{} Streaming large images with Metal sparse textures

Limit texture memory usage for large textures by loading or unloading image detail on the basis of MIP and tile region.