Introducing Image Processing in Metal

Harnessing the power of Metal for image processing.

Simon Gladman / Sep 2015

Introduction

Metal is Apple's framework to provide the lowest overhead, highest performance access to the GPU. Although GPU programming is often associated with rendering and texturing 3D scenes, Metal supports *compute shaders* which allow massively parallel computation over datasets such as image or bitmap data. This allows developers to manipulate still and moving images with existing filters such as blur and convolution or even create their own filters from scratch.

Metal is now available under both iOS and OS X and the same Metal code can be used across both classes of device. Metal is coded in a language based on C++ and its shaders, which are the small programs that are executed on the GPU, are precompiled to offer the highest performance.

MetalKit

The easiest way to integrate Metal into a Swift project is with Apple's *MetalKit* framework. Its *MTKView* class is an extension of *UIView* and offers an simple API to display textures to the user.

Furthermore, MetalKit includes *MTKTextureLoader* which allows developers to generate Metal textures from image assets.

There are a few steps required to get up and running with image processing in Metal which, briefly, are:

Creating a device

A Metal device (*MTLDevice*) is the interface to the GPU. It supports methods for creating objects such as function libraries and textures.

Creating a library

The library (*MTLLibrary*) is a repository of kernel functions (in our case, compute shaders) that are typically compiled into your application by Xcode.

Creating a Command Queue

The command queue (*MTLCommandQueue*) is the object that queues and submits commands to the device for execution.

Creating a Metal Function

Once the library is instantiated, we can create a function object (*MTLFunction*) from it. This represents a single Metal shader.

Creating a Pipeline State

The pipeline state takes the Metal function and compiles it for the device to execute.

Creating a Command Buffer

The command buffer (*MTLCommandBuffer*) stores the encoded commands that will be committed to the device for execution.

Creating a Command Encoder

Creating the command encoder (*MTLCommandEncoder*) is the final step and it is responsible for encoding the commands and resources into byte code that can be written into the command buffer.

Although that may seem like a lot of steps, many of these objects are persistent and long lived and, therefore, only need to be instantiated once. Take a look at some of the projects referred to in the links at the end of this article to see examples of MetalKit based projects.

Compute Shaders

GPU programming has typically been based on two different types of shader or program.

A *vertex shader* is responsible for taking each vertex of each object in a 3D scene and translating its position from its 3D universe to a screen location.

The second type of shader, a *fragment shader*, takes the data from the vertex shader and calculates the colour for every on-screen pixel. It's responsible for shading, texturing, reflections and refractions and so on.

However, for image processing, we need to use a third type of shader, a *compute shader*. Compute shaders aren't concerned with 3D space at all, rather they accept a dataset and, in our case, that dataset it a two dimensional array describing an image.

The simplest compute shader for image processing would simply pass through the colour of each pixel from a source image and write it to a target image and look something like this:

```
kernel void customFunction(

       texture2d<float, access::read> inTexture [[texture(0)]],

       texture2d<float, access::write> outTexture [[texture(1)]],

       uint2 gid [[thread_position_in_grid]])

{

       const float4 colorAtPixel = inTexture.read(gid);

       outTexture.write(colorAtPixel, gid);

}
```

Here, our kernel function, *customFunction*, is run once for every pixel. It has three arguments, the first is the readable input texture, *inTexture*, and the second is the writable output texture, *outTexture*. The final argument, *gid*, is supplied by the framework and refers to the coordinates of the current pixel.

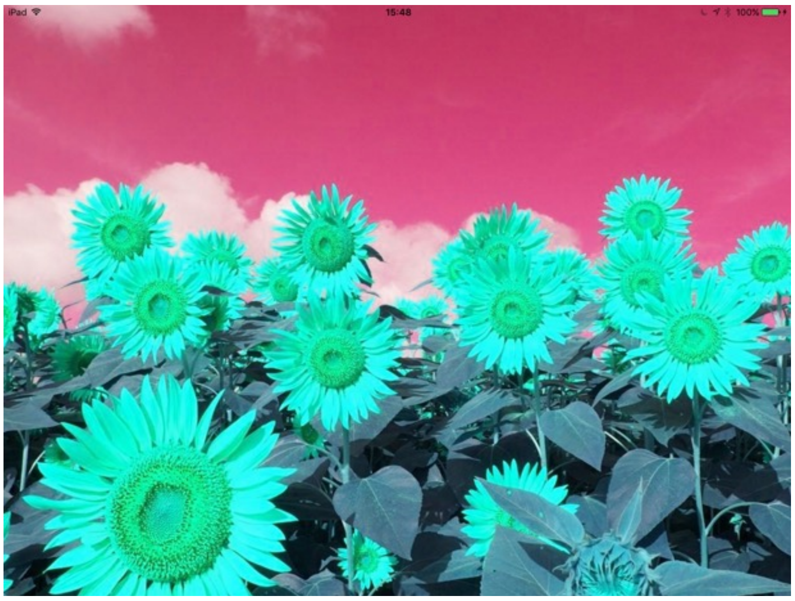
The first line of the function copies the RGBA colour value from the current pixel of the input texture into a constant, *colorAtPixel* and finally, that colour value is written out to the output texture.

Unsurprisingly, the output image looks exactly the same as the input image.

We can start playing with the image by mixing up the colour channels. By taking the red, green an blue input channels and writing them to the blue, red and green output channels respectively:

```
kernel void customFunction(

        texture2d<float, access::read> inTexture [[texture(0)]],

        texture2d<float, access::write> outTexture [[texture(1)]],

        uint2 gid [[thread_position_in_grid]])

{

        const float4 colorAtPixel = inTexture.read(gid);

        const float4 outputColor = float4(colorAtPixel.b, colorAtPixel.r, 
colorAtPixel.g, 1);

        outTexture.write(outputColor, gid);

}
```

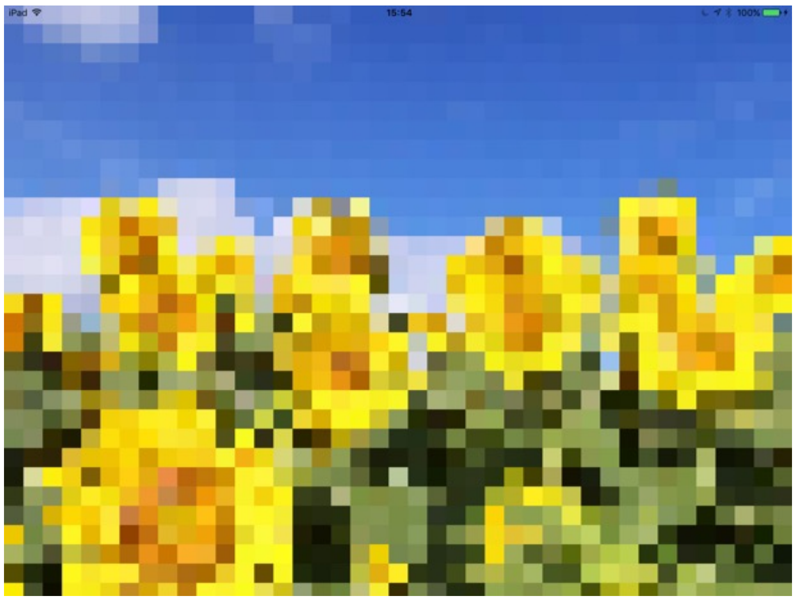
...we get this strangely coloured scene:



We're not limited to reading at the *gid* provided. By reading at different coordinates, we can change the sampling of the image. In this example, the *x* and *y* coordinates are rounded to the nearest 50:

```
kernel void customFunction(

       texture2d<float, access::read> inTexture [[texture(0)]],

       texture2d<float, access::write> outTexture [[texture(1)]],

       uint2 gid [[thread_position_in_grid]])

{

    const uint2 pixellatedGid = uint2((gid.x / 50) * 50, (gid.y / 50) * 50);

    const float4 colorAtPixel = inTexture.read(pixellatedGid);

    outTexture.write(colorAtPixel, gid);

}
```

To give a pixellated version of the image:



As you may be able to imagine, these two examples really only scratch the surface of what's possible with Metal. The Metal Shading Language is a fully fledged language in its on right, so the possibilities are limitless.

Metal Performance Shaders

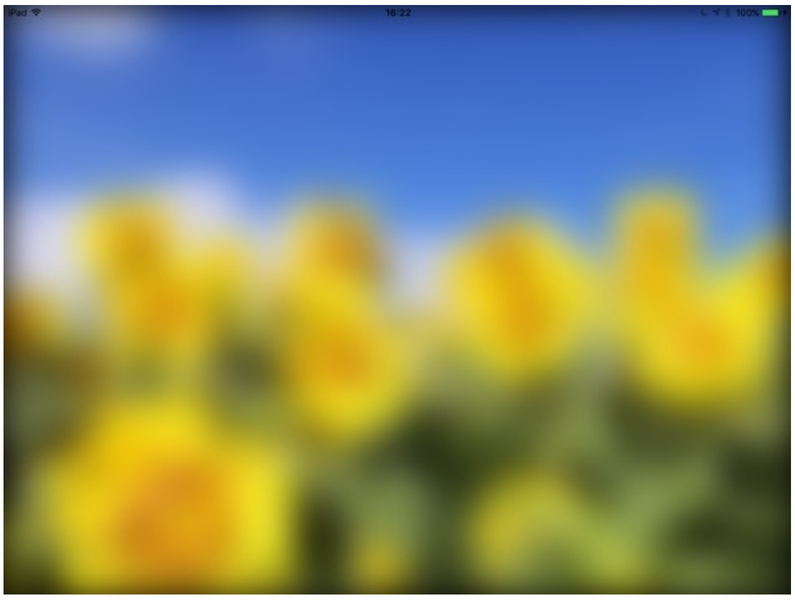
There are some very common image processing functions, such as Gaussian blur and Sobel edge detection, that Apple have written as Metal Performance Shaders.

These have been optimised to extraordinary lengths. For example their version of Gaussian blur actually consists of 821 different implementations that Apple tested for different conditions (e.g. kernel radius, pixel format, memory hierarchy, etc.) and the shader dynamically selects to give the best performance.

These shaders are easy to incorporate into existing Swift code that runs Metal. After instantiating a Metal Performance Shader, here we'll use a Gaussian blur, simply invoking *encodeToCommandBuffer* creates a new texture based on a filtered version of the source texture.

```
let blur = MPSImageGaussianBlur(device: device!, sigma: 50)


blur.encodeToCommandBuffer(commandBuffer,

        sourceTexture: sourceTexture,

        destinationTexture: destinationTexture)
```

Which gives a result like this:



]

Processing Live Video

Metal is fast enough to process a live video feed - even at the 2048 × 1536 frame size on an iPad. However, the *CVMetalTextureCacheCreateTextureFromImage* method in CoreVideo returns two image planes that constitute a YCbCr colour space - one plane holds the luma of the image and the other holds the blue and red deltas.

The Core Video framework includes the methods *CVMetalTextureCacheCreateTextureFromImage* and *CVMetalTextureGetTexture* that are able to generate Metal textures from these two planes. The *cameraOutput* protocol method of *AVCaptureVideoDataOutputSampleBufferDelegate* is an ideal place to do this and the code would look something like this:

```
let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer)

var cbcrTextureRef : Unmanaged<CVMetalTextureRef>?
let cbcrWidth = CVPixelBufferGetWidthOfPlane(pixelBuffer!, 1);
let cbcrHeight = CVPixelBufferGetHeightOfPlane(pixelBuffer!, 1);

CVMetalTextureCacheCreateTextureFromImage(kCFAllocatorDefault,
       videoTextureCache!.takeUnretainedValue(),
       pixelBuffer!,
       nil,
       MTLPixelFormat.RG8Unorm,
       cbcrWidth, cbcrHeight, 1,
       &cbcrTextureRef)

let cbcrTexture =
       CVMetalTextureGetTexture((cbcrTextureRef?.takeUnretainedValue())!)
 cbcrTextureRef?.release()
```

With similar code for the luma plane but with a plane index (the number one in bold) of zero.

This means that to display a video image to the user, those two planes need to be passed into a compute shader that converts YCbCr to RBG. Here's the shader to do that:

```
kernel void YCbCrColorConversion(
        texture2d<float, access::read> yTexture [[texture(0)]],
        texture2d<float, access::read> cbcrTexture [[texture(1)]],
        texture2d<float, access::write> outTexture [[texture(2)]],
        uint2 gid [[thread_position_in_grid]])
{
        float3 colorOffset = float3(-(16.0/255.0), -0.5, -0.5);
        float3x3 colorMatrix = float3x3(
                float3(1.164,  1.164, 1.164),
                float3(0.000, -0.392, 2.017),
                float3(1.596, -0.813, 0.000));
        uint2 cbcrCoordinates = uint2(gid.x / 2, gid.y / 2);
        float y = yTexture.read(gid).r;
        float2 cbcr = cbcrTexture.read(cbcrCoordinates).rg;
        float3 ycbcr = float3(y, cbcr);
        float3 rgb = colorMatrix * (ycbcr + colorOffset);

        outTexture.write(float4(float3(rgb), 1.0), gid);
}
```

Once that conversion is done, the resulting texture can be manipulated with a custom compute shader or a Metal Performance Shader.

In Conclusion

This is been a very quick whistle stop tour introduction to some of what Metal is able to do.

Over the last year, I've been blogging about Metal and the following posts expand on what's covered in this article:

iOS Live Camera Controlled Particles in Swift & Metal

Metal Performance Shaders & Fallback Copy Allocators

Generating & Filtering Metal Textures From Live Video

Using MetalPerformanceShaders on Images with MTKTextureLoader

A First Look at Metal Performance Shaders in iOS 9

Metal By Example also discusses using compute shaders for image processing in The Fundamentals of Image Processing in Metal.