2017.12.15

Photo Factory Design Paper

An Image Processing Software

Zhou Jianming, 1409853D-I011-0036

Contents

Abstract

As the need of post-production for photography, which are high-amount of pixels, high quality insurance, real-time responding, and can do enhance the image into a new stage of taste, resulting designing and programming an image editor in a challengeable topic. Despite its difficulty, I succeed to implement an image editor to meet these conflicted requirements together, powering by the Core Image and Metal Framework and running on macOS 10.13. In this report, the technology the software used would be discussed, including my own work and other works which helps execute some calculating, conversion and I/0 functions; I would also talk about design, icon, and the experiment.

Keywords: image-processing, swift, macOS, coreimage, metal

Introduction

The image editor is named Photo Factory, written in Swift 4.0, and built by Xcode 9.x. The user interface is configured by StoryBoard, a UI tool from Xcode. The framework I select to deal with the image processing is called Core Image, which is an image processing and analysis technology designed to provide near real-time processing for still and video images. It operates on image data types from the Core Graphics, Core Video, and Image I/O frameworks, using either a GPU or CPU rendering path. In another hand, it hides the low-level details, such as rendering configuration, thread management, pixel and file format, providing the highest level of convenience to do image processing. (Apple Developer, 2016)

While it does offer these convenience, I persist in controlling some low-level behavior for better performance. Such as the rendering on Metal platform, a brand-new graphic API running on Apple devices, it shares similar features with Vulkan API which are low cost, high performance comparing to OpenGL and hardware control. I managed to set the rendering pipeline on Metal, and chained all light adjustment filter together, to get the real-time (24 fps) responding ability.

Leave platform and technology along, the custom filters are also my contribution. These algorithms are part from another work of me, the Github repository page is here.

Design

The icon is as followed, 8 petals represent 8 colors in HSL Adjustment, from red to magenta. What's more is that, some other gallery or photo related app also draw the flower as symbol, so this icon can also suggest user there is an image editor application.

For the interface, I tried my best to reduce its color, and let it not disturb the user to determine image color. It is known to all that, color is so important for photographers and their precious work, thus I set the light grey as background, and dismiss the title bar to keep it simplicity. The HSL tab is so sad for I am not able to implement it, there are some limitation in macOS tabview. All sliders value are limited from -100 to 100 for user convenience, the technical discussion of these input unit will cover later. The other is closed to my paper design in group project.

Special Technical Details

This part is written by markdown editor for better codes and algorithm representation, please switch to next page.

SPECIAL TECHNICAL DETAILS

Core Image and Metal

At the moment loading the view of Photo Factory, a plenty of environmental initialization is on the road.

First I declared a CIContext and Metal object to config the graphic API and link them together.

```
init(MTLDevice device: MTLDevice)
```

```
// Declare the Metal Object, and set to use system default GPU
// If device have 1 GPU, then that is the default
// If own 2, the default is the more powerful one.
device = MTLCreateSystemDefaultDevice()

// Link the cicontext
context = CIContext(mtlDevice: device)
```

Next step is preparation of loading tools, command queue for thread management and display view

```
var commandQueue: MTLCommandQueue!
commandQueue = device.makeCommandQueue()
var textureLoader = MTKTextureLoader(device: device)

// Set up MTKView ->>> MetalKit View
metalview = MTKView(frame: CGRect(x: 60, y: 60, width: 600, height:
400), device: self.device)
// Re-calibrate the MTKView Size
metalview.setFrameSize(sourceTexture.aspectRadio.FrameSize)
metalview.delegate = self
metalview.framebufferOnly = false

// Save the depth drawable to lower memory increasing
metalview.sampleCount = 1
metalview.depthStencilPixelFormat = .invalid
// FPS rate Control
metalview.preferredFramesPerSecond = 24
metalview.clearColor = MTLClearColor(red: 1, green: 1, blue: 1,
alpha: 1)

// Tell the view about texture(image) size
metalview.drawableSize = CGSize(width: sourceTexture.width, height:
sourceTexture.height)

//Put it into the window
view.addSubview(metalview)
```

As the rendering codes, I set up a value to tell the render function if there any complex filter(the filter requires more time to execute, such as blur and sharpen), if not, just rendering the light adjustment filter with the factors of sliders. In these codes, you can also figure out and feel about the idea of Filter Chain, by setting the output image of filter A as input image of filter B.

Besides, to make the slider easier to use, and the adjust factor simpler to understand, I scale all the slider from -100 to 100, and adjust them to the suitable input units by following codes.

```
let contrast = ContrastSlider.floatValue < 0 ? 1 -
ContrastSlider.floatValue / 133.4 : 1 + ContrastSlider.floatValue /
34
// These are one single line codes, for the contrast factor scale
from 0.25 to 4, 1 is the default

// From -2 to 2
let highlight = HighlightSlider.floatValue / 50.0
// From 0 to 2
let saturation = (SatSlider.floatValue + 100) / 100.0


exposureFilter?.setValue(inputImage, forKey: "inputImage")
// From -2 to 2
exposureFilter?.setValue(ExpoSlider.floatValue / 50.0, forKey:
"inputEV")
```

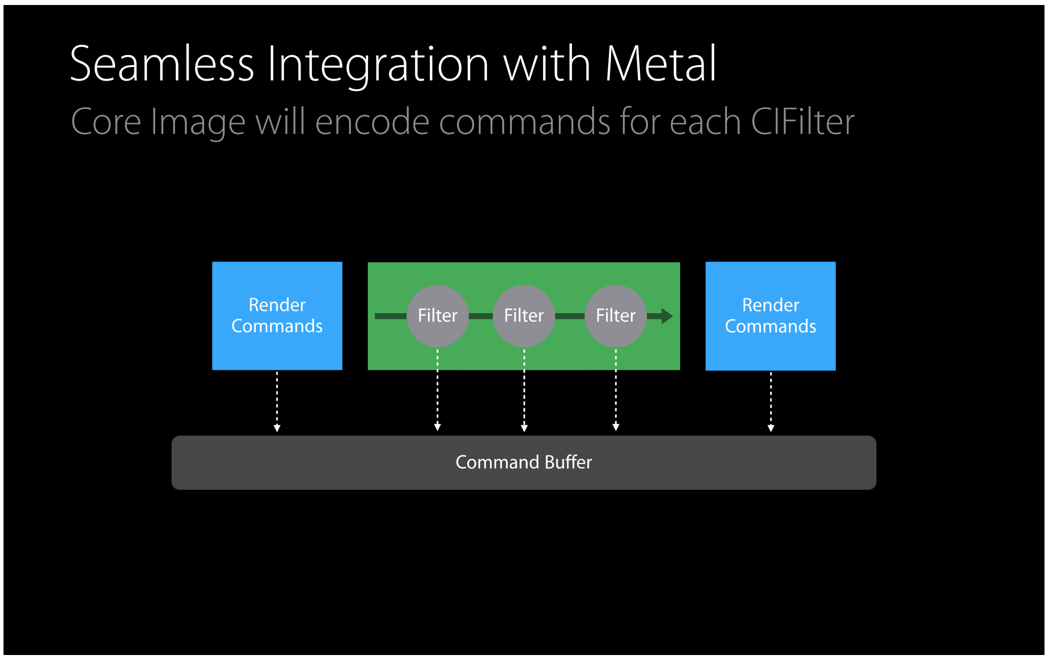
```
// Set for shadow, from -1 to 1
highshadowFilter?.setValue(exposureFilter?.outputImage, forKey:
"inputImage")
highshadowFilter?.setValue(ShadowSlider.floatValue / 100.0, forKey:
"inputShadowAmount")

// Custom Highlight Filter
let highlightFilter = HighlightFilter()
highlightFilter.inputImage = highshadowFilter?.outputImage
highlightFilter.inputUnit = CGFloat(highlight)

// Set contrast
conFilter?.setValue(highlightFilter.outputImage, forKey:
"inputImage")
conFilter?.setValue(contrast, forKey: "inputContrast")
conFilter?.setValue(saturation, forKey: "inputSaturation")

// Set saturation
satFilter.inputImage = conFilter?.outputImage
satFilter.inputUnit = CGFloat(saturation)



// Set HSL Adjustment filter, and shift for each color
// Each shift vector consist of three element, represent hue,
saturation and luminance shift, from 0 to 2

hslFilter.inputImage = satFilter.outputImage
let redshift = CIVector(x: CGFloat((RedHueS.floatValue + 100) /
100), y: CGFloat((RedSatS.floatValue + 100) / 100), z:
CGFloat((RedLumS.floatValue + 100) / 100))
let orashift = ...
let yellshift = ...
let greshift = ...
let aqushift = ...
let blueshift = ...
let purshift = ...
let magshift = CIVector(x: CGFloat((MagHueS.floatValue + 100) /
100), y: CGFloat((MagSatS.floatValue + 100) / 100), z:
CGFloat((MagLumS.floatValue + 100) / 100))

hslFilter.inputRedShift = redshift
hslFilter.inputOrangeShift = orashift
hslFilter.inputYellowShift = yellshift
hslFilter.inputGreenShift = greshift
hslFilter.inputAquaShift = aqushift
hslFilter.inputBlueShift = blueshift
hslFilter.inputPurpleShift = purshift
hslFilter.inputMagentaShift = magshift
```

Last Part of rendering, is setting the last output image as metalview input, and then rendering(display) on it.

```
context.render((hslFilter.outputImage)!, to:
currentDrawable.texture, commandBuffer: commandBuffer, bounds:
inputImage.extent, colorSpace: colorSpace!)
commandBuffer?.present(currentDrawable)
commandBuffer?.commit()
```

The whole rendering codes is written in func draw(in view: MTKView). The function would be called at each rendering, in this case, call for 24 times per second.

```
func draw(in view: MTKView) {
    if complexOperation != .None{
        // Do complex filter operation
    }

    /* Filter Chain codes */

    /* Rendering to metalview */

}
```


Seamless Integration with Metal
Core Image will encode commands for each CIFilter
Render Commands
Filter
Filter
Filter
Render Commands
Command Buffer

(Apple Developer, 2015)

In this way software can get the newest adjustment factor for each filter 24 times per second which beyond the human eyes re-fresh rate, put the operations to high performance approach, deal with the problem that only one thread can assign to user interface (by submit the rendering work directly to system), and minimize the memory transition between CPU and GPU to low the execute costs.

Object Brief

CIImage is not a real image object, but a recipe about how an image is filtered, processed and changed.

NSImage is an image object which holds image pixel information, and can write it into a file in JPG or PNG.

MTLTexture is a kind of texture that Metal shader can read and write, just like the texture in OpenGL.

Custom Filter

There are three customized filters in the Photo Factory, and share the same benchmark with build-in filter. I will go through them one by one.

Pre-processing for Blur Filters

All blur filters are partly customized for preventing the black edge issues, as a result of border processing, the build-in blur filters have no idea about it.

At the beginning, I created a "white filter" that is only return vector of all 1 values, and modify its extent control to let it output the larger white image with an amount, 100 pixel each direction of image in my case and merge with the original one, then put it into blur filter.

```
whiteFilter.inputImage = inputImage

// Merge
let combine = CIFilter(name: "CISourceOverCompositing")
// Set the coordinate
let trans = inputImage?.transformed(by:
CGAffineTransform(translationX: 100, y: 100))
combine?.setValue(trans, forKey: "inputImage")
combine?.setValue(whiteFilter.outputImage, forKey:
"inputBackgroundImage")

// Declare the blur filer
var blurFilter = CIFilter()

if complexOperation == .Guassian {
    blurFilter = CIFilter(name: "CIGaussianBlur")!
    // Continues
}else if complexOperation == .Box{
    blurFilter = CIFilter(name: "CIBoxBlur")!
    // Continues
}else if complexOperation == .Motion {
    blurFilter = CIFilter(name: "CIMotionBlur")!
    // Continues
}
```

When I got the blur result, the only thing I need to do is crop it by the 100 pixels of four edges.

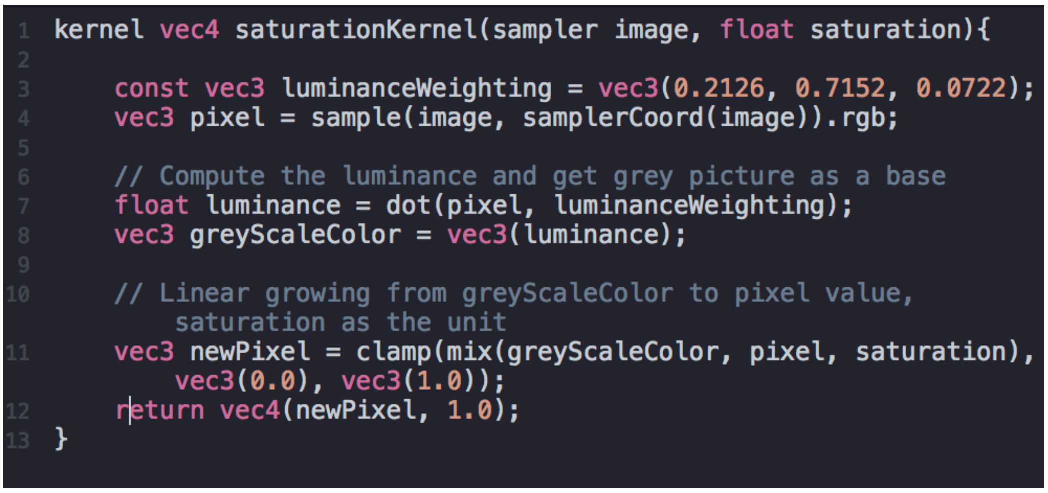
```
filter = CIFilter(name: "CICrop")!
filter.setValue(blurFilter.outputImage, forKey: "inputImage")
```

```
let size = CIVector(x: 100, y: 100, z: width!, w: height!)
filter.setValue(size, forKey: "inputRectangle")
```

Saturation Filter

This filter is written in Core Image Kernel Language, it defines functions, data types, and keywords that I can use to specify image processing operations for custom Core Image filters that I write. I can also use a subset of the OpenGL Shading Language (glslang).(Apple Developer, 2015)

The code is as followed, you can also find it in class SaturationFilter under CustomFilter.swift: (stylise for readability)



```
kernel vec4 saturationKernel(sampler image, float saturation){

    const vec3 luminanceWeighting = vec3(0.2126, 0.7152, 0.0722);
    vec3 pixel = sample(image, samplerCoord(image)).rgb;

    // Compute the luminance and get grey picture as a base
    float luminance = dot(pixel, luminanceWeighting);
    vec3 greyScaleColor = vec3(luminance);

    // Linear growing from greyScaleColor to pixel value,
        saturation as the unit
    vec3 newPixel = clamp(mix(greyScaleColor, pixel, saturation),
        vec3(0.0), vec3(1.0));
    return vec4(newPixel, 1.0);
}
```

The kernel read the pixel and calculate its luminance, then mix it between, even beyond the luminance and original pixel value.

As I mentioned before, the kernel is from another work of myself, its Github page is mentioned before.

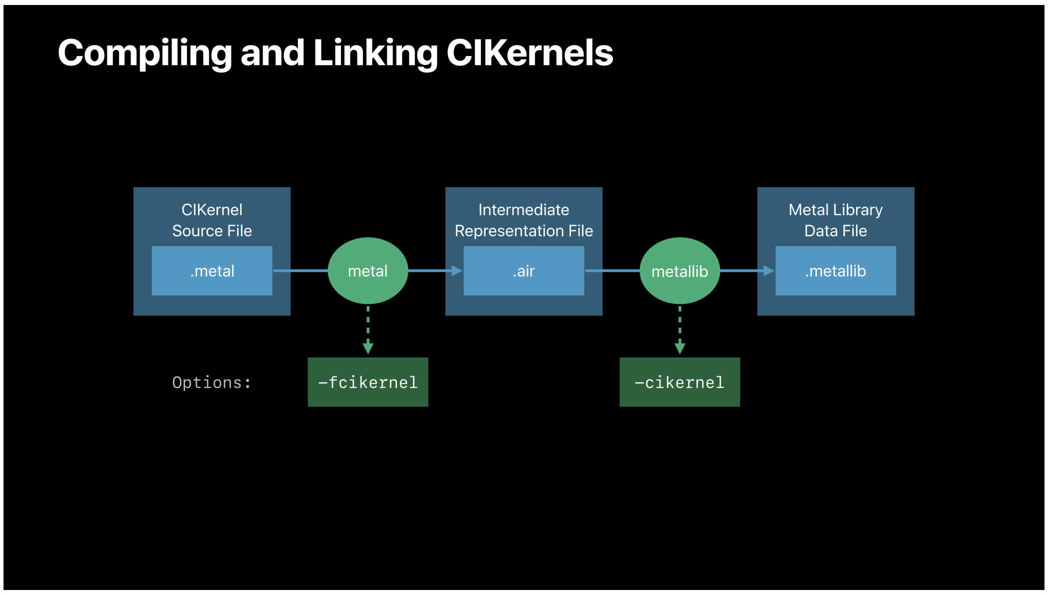
Highlight Filter

This filter is written in Metal Shading Language, which is a C++ based programming language that developers can use to write code that is executed on the GPU for graphics and general-purpose data-parallel computations.(Apple Developer, 2017)

```
float4 highlight(sampler img, float unit){
    float2 coordinate = img.coord();
    float3 pixel = img.sample(coordinate).rgb;
    const float3 luminanceWeighting = float3(0.2126, 0.7152, 0.0722);

    float luminance = dot(pixel, luminanceWeighting);
    float shadowGreyScale = clamp(luminance - 0.55, 0.0, 0.2);

    // Compute new pixel value with 20 * x^2 + x transiting function
    // Add the function to adjust exposure instruction
    float3 newPixel = pixel * pow(2.0, unit * (pow(shadowGreyScale, 2.0) * 20.0 + 1.0 * shadowGreyScale));
    newPixel = clamp(newPixel, float3(0.0), float3(1.0));

    // Alpha for opacity of image
    return float4(newPixel, 1.0);
}
```

During compiling the metal shader will translate to .air file, then coding into metallic file as Metal Library Data File, just like a dll file in windows, then Core Image will load the library to execute the filter. You can find the detail in class HighlightFilter.

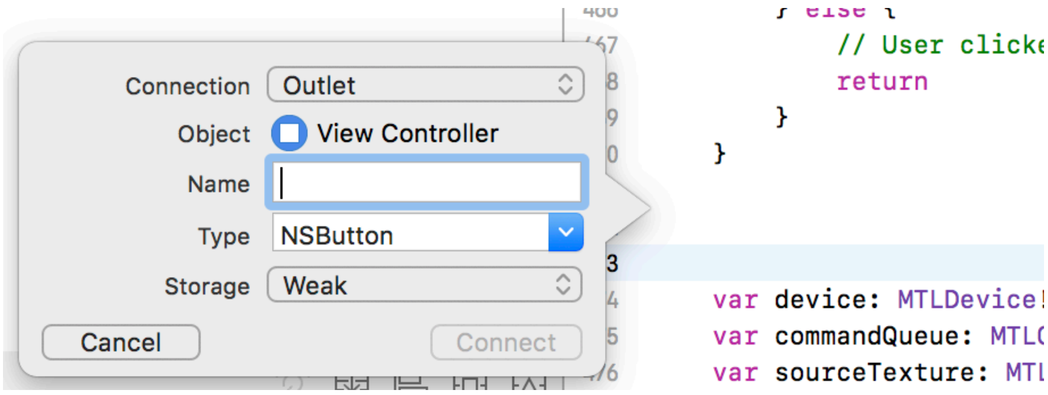
```
let url = Bundle.main.url(forResource: "default", withExtension:
"metallib")

let data = try! Data(contentsOf: url!)
let kernel = try! CIKernel(functionName: "highlight",
fromMetalLibraryData: data)
```



(Apple Developer, 2017)

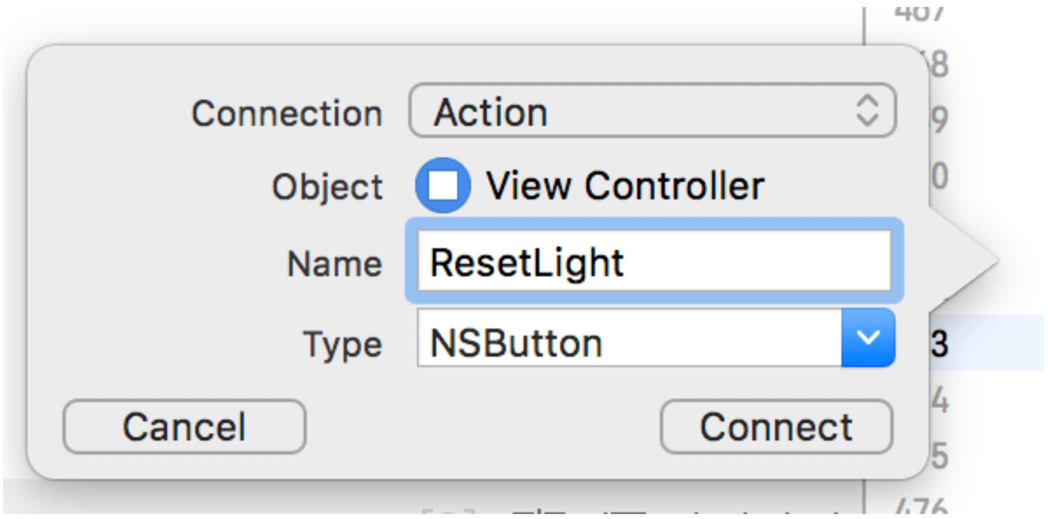
UI Control

The whole UI element except metalview is draw on Main.storyboard, this tool also supplies the possibility of linking UI element and event with codes. If you see any function which is beginning with @IBAction, it is the event control to handle slider value changing, button clicking or other events; if you see @IBOutlet, it represents one single UI element, I can access its title or value it holds.



Connection: Outlet
Object: View Controller
Name:
Type: NSButton
Storage: Weak

```
        } else {
            // User clicke
            return
        }
    }

    var device: MTLDevice!
    var commandQueue: MTLC
    var sourceTexture: MTL
```



Connection: Action
Object: View Controller
Name: ResetLight
Type: NSButton

I/O

The I/O implementation consists some work belonging to others, and I am going to discuss it in two parts

Load Image

Thanks to Core Image, not only JPG, PNG, tiff can Photo Factory load into processing, but most of the RAW file type is also supported.

If users want to open non-RAW file, just click the Open Button and find the file. This function is powered by @IBAction func OpenFile(sender: AnyObject), which load the file directly into a MTLTexture object, and wait for processing.

If users want to enable all potential of their camera, to open a RAW file, he/she can click Open RAW and find the file. CIImage initialization of Core Image responses for this part, it can open almost each type of camera raw file of DSLR and other type devices you could imagine, including Nikon, Sony, Canon, Fujifilm, or Panasonic. The support list is here

Save Image

The saving function depends on extensions of MTLTexture named toNSImage and toCGImage, you can find it in extension MTLTexture {} under Extension.swift. And the final file output is from NSImage to file, this is executed by func writeJPG(toURL url: URL) in extension NSImage.

Complex Filter

There is a global value named complexOperation with default value Filter.None to tell rendering function is there any time-consuming filter need to execute. The enum type Filter is made up of all supported complex filter with the build-in filter name. Each menu item has a identifier matching the build-in filter name, it is also helpful to initial the right enum value via init(rawValue: identity).

When the value is not equal to .None, the complex filter begin to execute. First it pauses the metalview rendering in case of unexpected queue full of time out. Then set the input of corresponding filter, direct the output to the light filter input and replace the based CIImage (the one which loads into light filter each rendering), continue the metalview and reset the variable to .None.

The software checkes this value in each rendering, so if user click the menu item, it will execute the complex filter immediately and continue the rendering.

Reference:

Apple. (2017, December 4). Digital camera RAW formats supported by iOS 11 and macOS High Sierra. Retrieved from https://support.apple.com/en-us/HT207972

Apple Developer. (2014, June). Developing Core Image Filters for iOS. Presented at the WWDC 2014, San Francisco. Retrieved from https://developer.apple.com/videos/play/wwdc2014/515/

Apple Developer. (2015a, January 12). Core Image Kernel Language Reference. Retrieved from https://developer.apple.com/library/content/documentation/GraphicsImaging/Reference/CIKernelLangRef/Introduction/Introduction.html

Apple Developer. (2015b, June). What’s New in Core Image. Presented at the WWDC 2015, San Francisco. Retrieved from https://developer.apple.com/videos/play/wwdc2015/510/

Apple Developer. (2016a, June). Live Photo Editing and RAW Processing with Core Image. Presented at the WWDC 2016, San Francisco. Retrieved from https://developer.apple.com/videos/play/wwdc2016/505/

Apple Developer. (2016b, September 13). About Core Image. Retrieved from https://developer.apple.com/library/content/documentation/GraphicsImaging/Conceptual/CoreImaging/ci_intro/ci_intro.html

Apple Developer. (2017a, June). Advances in Core Image: Filters, Metal, Vision, and More. Presented at the WWDC 2017, San Francisco. Retrieved from https://developer.apple.com/videos/play/wwdc2017/510/

Apple Developer. (2017b, September 12). Metal Shading Language Specification - Apple Developer. Retrieved from https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf

Colin. (2016). Core Image 你需要了解的那些事~. Retrieved from https://colin1994.github.io/2016/10/21/Core-Image-OverView/