SHOPIFY

Metal Workshop

INTRODUCTION TO APPLE'S METAL API

Presenter:
Chris FEHER

Date:
August 24, 2017

What is Metal?

Metal is an iOS, macOS, and tvOS API for rendering 2D and 3D graphics. It provides near-direct access to the GPU to enhance the computational power of your apps. Metal is similar to OpenGL, but provides certain optimizations that a cross-platform API cannot provide.

What are we going to do with it?

We are going to draw some shapes and animate them. Similar to the lazerbeam map, we will go through the learning process necessary to draw a shockwave. A shockwave appears as a ring expanding out from the origin to some max radius while fading out at the same time.

The Basics

Vertex

For our workshop, a vertex in Metal is a point in 2D-space with a position and colour property.

```
struct Vertex {
    let position: vector_float4
    let color: vector_float4
}
```

Figure 1: Example Vertex

The position is made up of 4 values (x, y, z, w). Since we are drawing in 2D-space, the z value will be fixed at **0**. The w value is not used for this workshop but needs to be fixed to **1**. Similar to poition, colour is made up of 4 values (r, g, b, a) for red, green, blue, and alpha respectively. We will use green, **{0, 1, 0, 1}**, for the purposes of this workshop.

Coordinate Systems

Metal uses the **clip coordinate system** and iOS/MacOS uses the **window coordinate system**.

Window Coordinate System

On the Mac, the coordinate system's origin is in the bottom left of the screen. The x-axis goes from 0 to the width of your screen in pixels. The y-axis goes from 0 to the height of your screen in pixels.

Clip Coordinate System

In the Metal coordinate system the origin is in the centre of the screen. The the x-axis goes from -1 to 1 and the y-axis goes from -1 to 1.

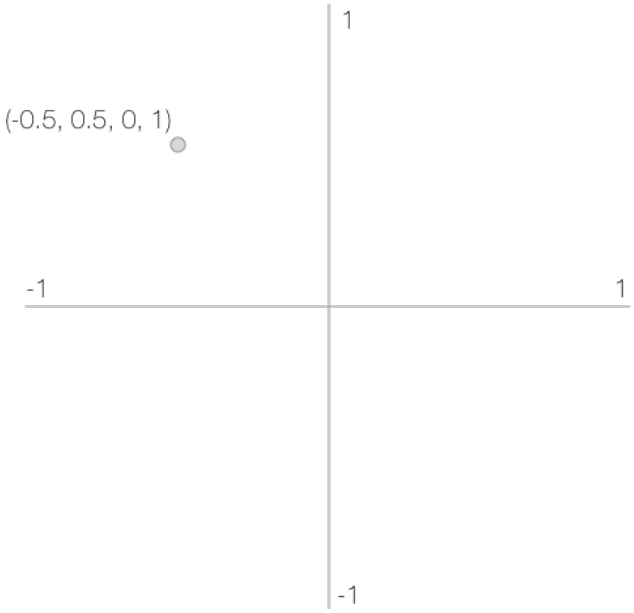


Figure 2: Clip Coordinate System

Shape

Shapes in Metal must be drawn using triangles. Each triangle is made up of 3 vertices. For example, a single square can be drawn with 2 triangles (Fig. 3).

Triangle Strip

A square consists of 2 triangles, therefore we could draw it with 6 vertices. Metal provides us with a primitive type to store a triangle. Such primitive is named **MTLPrimitiveType.triangle**. by specifying to Metal that we want to draw using the primitive type **MTLPrimitiveType.triangle**.

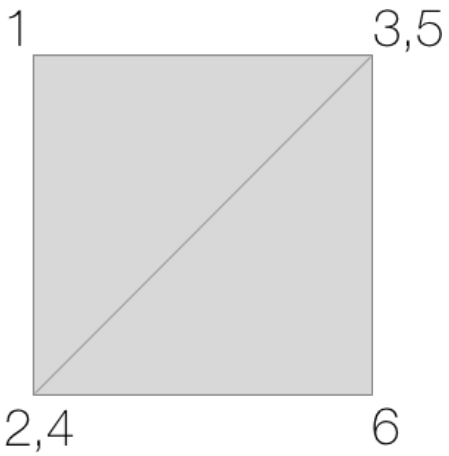


Figure 3: Inefficient Square

However, this would be an inefficient use of memory because we would be duplicating vertices. We can use the **MTLPrimitiveType.triangleStrip** primitive instead to draw the square with 4 vertices. Metal will reuse vertices to draw the second triangle.

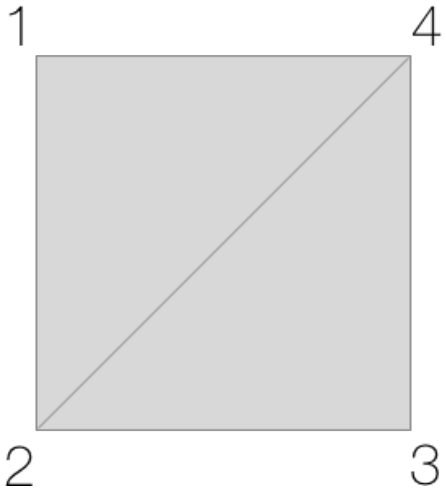


Figure 4: More Efficient Square

Instanced Rendering

Say you want to draw 6 copies of **Figure 4** on the screen. You would think that you pass in **6x4** vertices, but this would be incorrect if you are using instanced rendering. Instanced rendering allows us to provide one set of vertices for a shape and a number of copies. So, we will provide 4 vertices and tell the API we want 6 copies.

Vertex Buffer

A vertex buffer is an array that contains all the vertices for your shape. Since we are using instanced rendering to draw copies of shapes instead of duplicating vertices, we will have a vertex buffer containing 1 set of vertices that defines our shape.

For the purposes of this workshop, the center of our shape will be the origin **{0, 0, 0, 1}**. To draw the square in **Figure 6** we will have the following vertex buffer:

[{-0.5, 0.5, 0, 1}, {-0.5, -0.5, 0, 1}, {0.5, -0.5, 0, 1}, {0.5, 0.5, 0, 1}]

Figure 5: Vertex Buffer

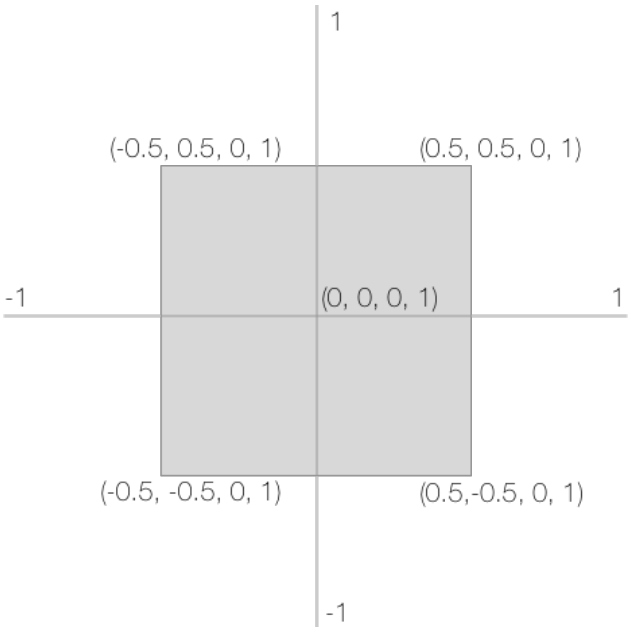


Figure 6: 4 vertices to draw square shape

Index Buffer

We now have a vertex buffer containing 4 vertices for our square, but we need to tell the system how we want them connected. To do this, we need to provide an index buffer.

An index buffer references the vertices from the vertex buffer in the order in which they should be drawn. For the vertex buffer in **Figure 5**, we would have the following index buffer:

Index Buffer = [0, 1, 3, 2]

The system will draw a triangle using the first 3 vertices in the index buffer and will then connect each additional vertex to the previous 2 forming a square!

Uniform Buffer

A uniform is a struct which contains information for transforming a vertex.

For example, if you recall, we draw all shapes around the origin, **{0, 0, 0, 1}**. If we want to position our shape somewhere else on the screen, we need to provide a position offset for each vertex in our shape. Since we are moving the positions of all vertices in our square, the uniforms for each vertex in the square will be equal.

The offset is specified in a struct as follows:

```
struct SimpleSquareUniform {
    let positionOffset: vector_float2
}
```

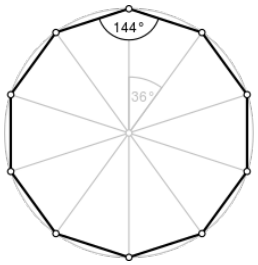
On every frame, we pass a vertex buffer containing 4 *Vertex* structs and a uniform buffer containing 4 *SimpleSquareUniform* structs into Metal. At this point, we move into a special file called a Shader where we can apply the uniform transformations to each corresponding vertex.

The Code

The code can be checked out using **git** from **`https://github.com/Shopify/metal-workshop`**.

1 Exercise - Draw a circle

A circle in Metal is drawn using triangles. A decagon looks kind of like a circle; if you make it small enough, it looks round.



For this part, move into the folder titled **part-1/start**. You will need to design a struct to draw the decagon above. Try to make the function generic enough that you could draw an **N** sided polygon. There is some boilerplate code provided for you. For this part of the workshop, you will only be working in **Shapes.swift** and modifying the struct called **InnefficientCircle**.

Note that there is an extension to **CGPoint** provided to you with the following signature. It takes a point in 2D-space and rotates it about (0,0).

```
extension CGPoint {
    func rotate(byRadians: CGFloat) -> CGPoint
}
```

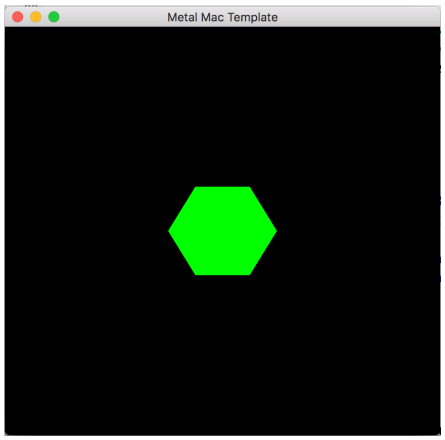


Figure 7: Result for exercise 1

More Advanced Topics

Metal Quirks

Even though you have created your structs in Swift, you will need to mirror those structs in Metal. It is important that they match in size and alignment (they do not have to match in name). In other words, **the bytes have to match up**. Creating a corresponding struct for **figure 1** would look like this:

```
struct InVertex
{
        float4  position [[position]];
        float4  color;
};
```

Figure 8: Mirrored Vertex Struct

Shaders

For our workshop, a Shader file contains 2 mandatory functions written in Apple's Metal Shader Language; a language akin to C. The shader file may also contain other helper functions that you create.

Vertex Shader

The vertex shader is the function where you apply transformations to vertices. The function is called once per (vertex, uniform) pair. The output of this function is a vertex struct with the applied transformations and may have additional properties on it. For the first part of this workshop, the Vertex shader will return a vertex of the same type as the input.

Fragment Shader

The vertex shader works in the clip coordinate system as expected, but the fragment shader is now back in the **window coordinate system**. The fragment shader interpolates points between 2 vertices and gets called for every pixel in your shape and returns a colour for said pixel. Say you want to draw **figure 6** and you have a window resolution of **1080x1080**. You have 4 vertices and thus your vertex shader gets called 4 times, one for each vertex, but your fragment shader would get called once for every pixel inside the square which is **1,166,400** times.

Shaders Recap

Vertex Shader: Takes in a Vertex and a Uniform and returns a vertex
Fragment Shader: Takes in a Vertex (translated to window coordinate system) and returns a colour

```
vertex Vertex circle_vertex_main(
           constant Vertex *vertices [[buffer(0)]],
           constant CircleUniform *uniforms [[buffer(1)]],
           ushort vid [[vertex_id]],
           ushort iid [[instance_id]])
```

Figure 9: Sample Vertex Shader Signature

```
fragment half4 circle_fragment_main(InVertex inFrag [[stage_in]])
```

Figure 10: Sample Fragment Shader Signature

Now that we can control the colours of pixels inside the bounds of our vertices, we can think about the way we draw shapes differently. If we want to draw a complex shape, we can simply set the alpha of the pixels we do not want to appear to clear. This will both reduce the memory and move complex mathematical calculations off of your CPU and on to your GPU but often requires that you pass more information into your shaders via your Uniform struct.

2 Exercise - Draw a Smarter Circle

For this part, move into the folder titled **part-2/start**. You are already provided with a shape to use which contains 4 vertices. This shape, if drawn using passthrough shaders, will produce a green square. You will need to modify the fragment shader in **SmartCircleShaders.metal** to draw a perfectly round circle using only 4 vertices.

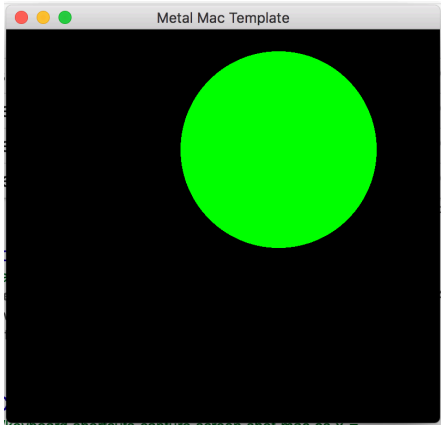


Figure 11: Result for exercise 2

Animations

This is where the uniform buffer becomes more important than ever. In order to animate something, you need to keep track of time for each shape on the screen. This timestep generally goes in the uniform for each vertex and gets updated every frame. iOS provides timers you can link into to get frame time which is roughly $\frac{1}{60}$ seconds. We will use $\frac{1}{30}$ seconds for this MacOS app.

The Bad Way

Say you want your shape to fade onto the screen to a classic **.easeInOut** animation curve. On every frame you would increment a timer on every vertex, then use one of Apple's built in timing functions to calculate the shapes alpha, then update your uniform buffer with the new alpha value. It doesn't seem like a lot, but when you have hundreds or even thousands of shapes on the screen at any given time, it is a lot. In this scenario, your GPU is doing very little work. We want the GPU to work harder and the CPU to work less

A Smarter Way

In the same scenario as above, we would increment **only** the time value on each vertex and not perform any calculations. We would then define a timing curve and perform the calculations in the vertex shader using the GPU.

3 Animating a Circle's Alpha

For this part, move into the folder titled **part-3/start**. We are going to make the circle pulse by animating it's alpha to a sin function. You'll need to modify the struct called **PulsingCircle** in **Shapes.swift** as well as the vertex shader in **PulsingCircleShaders.metal** to do this exercise.

The result should be a circle that perpetually animates its alpha from 0 to 1, then from 1 to 0.

4 Draw a Shockwave

For this part, you should use the code you just created in exercise 3 or move into **part-3/finish**. For this part, we are going to transform the circle in part 3 into a shockwave with a thickness of 2 pixels.

Remember: "A shockwave appears as a ring expanding out from the origin to some max radius while fading out at the same time"

Hint: Only 2 lines need to change//
The result should be a shockwave that grows and shrinks at the same rate that its alpha changes.

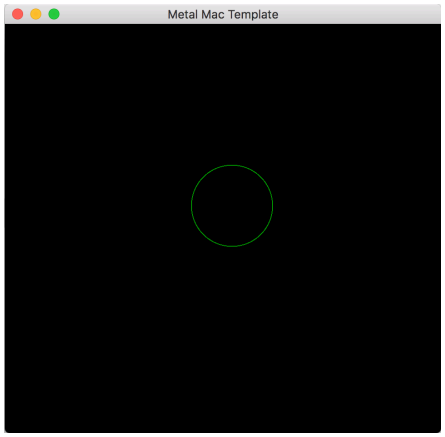


Figure 12: Result for exercise 4

5 Animate to a Curve

Animate the shockwave to a cubic function with control points **{0, 0.419999, 0.579999, 1}**