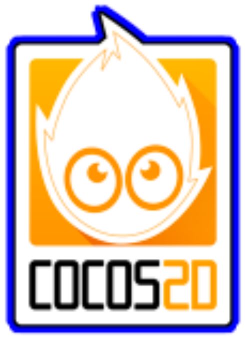
# E08: Outline



The idea for this shader is similar to the shadow shader. Instead of just taking one texture sample where the shadow will be, take several in a circular pattern. Once they are blended together, you'll have a mask of the pixels near the sprite. Blend that mask with a color and composite it under the sprite. Tada! Let's look at the fragment shader.

```
const int SAMPLES = 6;
varying vec2 v_OutlineSamples[SAMPLES];

vec4 composite(vec4 over, vec4 under){
  return over + (1.0 - over.a)*under;
}

void main(){
  // Use the coordinates from the v_OutlineSamples[] array.
  float outlineAlpha = 0.0;
  for(int i=0; i<SAMPLES; i++){
    float alpha = texture2D(cc_MainTexture, v_OutlineSamples[i]).a;

    // Blend the alpha with our other samples.
    outlineAlpha = outlineAlpha + (1.0 - outlineAlpha)*alpha;
  }

  // Let's repurpose cc_FragColor for the outline color.
    vec4 outlineColor = cc_FragColor*outlineAlpha;
  vec4 textureColor = texture2D(cc_MainTexture, cc_FragTexCoord1);
  gl_FragColor = composite(textureColor, outlineColor);
}
```

Let's break it down:

- `v_OutlineSamples` is a array of `vec2`. It also uses the `varying` keyword. Must mean something important!
- A for loop, just like you'd see in Objective-C. It blends a bunch of texture samples together for the outline's alpha.
- `cc_FragColor` is used for the outline color instead of using like a tint color. Multiply it against the outline alpha to make the correct premultiplied color.
- Finally the texture and outline are composited together.

The only real question is, what is that magical `v_OutlineSamples` that had all the right texture coordinates? It has to do with that `varying` keyword. That means that it's calculated in a *vertex shader*. More code!

## Vertex Shaders and Varying Variables:

If you've never done OpenGL programming before, you are probably wondering what vertexes have to do with rendering sprites? It turns out that modern GPUs only know about drawing triangles. To draw a rectangle (a sprite), you need to draw two triangles. In order to draw the triangles, you need the four corners of the rectangle. The corners are the vertexes. The vertex shader tells the GPU where the vertexes are on the screen and sets variables used by the fragment shader like the color and texture coordinates. I've never met anybody that thought vertex shaders were intuitive. Don't be discouraged if they don't make sense at first.

If you remember from the first example, I mentioned that every CCShader needs both a vertex shader and a fragment shader. Since we've only been making fragment shaders, Cocos2D has been pairing them with it's default vertex shader. Let's look at that first. It's actually pretty simple.

```
void main(){
  gl_Position = cc_Position;
  cc_FragColor = clamp(cc_Color, 0.0, 1.0);
  cc_FragTexCoord1 = cc_TexCoord1;
  cc_FragTexCoord2 = cc_TexCoord2;
}
```

A lot of this should look familiar already. It starts out with a `main()` function that writes to a GL variable and some builtin Cocos2D variables. Like how fragment shaders must write to the `gl_FragColor` variable, a vertex shader must write to `gl_Position`. This is the position of the vertex on the screen. Cocos2D calculates this when batching sprites, so the vertex shader just needs to copy it.

What about the `cc_Frag*` variables though? Each pixel gets a different value, but the vertex shader only sets the value once. That's what `varying` variables are. When a pixel is halfway between two vertexes, when the fragment shader runs the varying variable will be halfway between the value set in the two vertexes.

Ok! So what does the outline vertex shader look like?

```
uniform vec2 u_MainTextureSize;
uniform float u_OutlineWidth;

const int SAMPLES = 6;
varying vec2 v_OutlineSamples[SAMPLES];

void main(){
  gl_Position = cc_Position;
  cc_FragColor = clamp(cc_Color, 0.0, 1.0);
  cc_FragTexCoord1 = cc_TexCoord1;

  vec2 outlineSize = u_OutlineWidth/u_MainTextureSize;
  for(int i=0; i<SAMPLES; i++){
    float angle = 2.0*3.14159*float(i)/float(SAMPLES);
    v_OutlineSamples[i] = cc_TexCoord1 + outlineSize*vec2(cos(angle
), sin(angle));
  }
}
```

Let's break it down:

- The `main()` function starts out the same way as the default. It skips `cc_FragTexCoord2` since it's not used.
- Another for loop that iterates over the array. This one calculates offsets from the regular texture coordinate in a circle.
- Since the offsets are in texture coordinates, not Cocos2D points, we need to pass the texture size in using a uniform to rescale the offsets.

## Why Vertex Shaders?

### Avoiding Redundant Calculations:

Vertex shaders seem like an awful lot of work. So why use them? Think about it this way. Given a 100x100 pixel sprite, the fragment shader has to run 10,000 times while the vertex shader only needs to run 4. You can save a *lot* of work for the GPU if you can calculate something in a vertex shader instead of a fragment shader.

### Dependent Texture Samples:

Remember back to the basic Cocos2D shader. It looked up the texture like this:

```
texture2D(cc_MainTexture, cc_FragTexCoord1)
```

In this case, the texture coordinate is constant value. Because of this, the GPU can actually look up the texture sample before the fragment shader program even starts. If your shader makes many texture samples, the GPU can even do them at the same time!

On the other hand, some of the shaders we've made so far have looked like this:

```
texture2D(cc_MainTexture, somethingWeCalculated);
```

The sample coordinate is no longer constant. The GPU has to start the program running to calculate the texture coordinate then stop the program to make the sample. This is called a dependent texture sample. If you make many samples, it will have to start and stop your program many times. The same thing happens when you access RAM on a regular CPU, but the effect is usually much worse on a GPU.[1] It's one of the more expensive things you can do in a fragment shader.

## Excercises:

- Try rewriting the effect to only use the fragment shader.

1. Apple’s documentation for the A7 (iPad Air, iPhone 5s) states that dependent texture reads aren’t penalized. ↩