Introduction to Custom Filters (aka CSS Shaders)

By Paul Lewis

Paul is a Design and Perf Advocate

Custom Filters, or CSS Shaders as they used to be called, allow you to use the power of WebGL's shaders with your DOM content. Since in the current implementation the shaders used are virtually the same as those in WebGL, you need to take a step back and understand some 3D terminology and a little bit of the graphics pipeline.

I've included a recorded version of a presentation I recently delivered to LondonJS. In the video I step through an overview of the 3D terminology you need to understand, what the different variable types are that you'll encounter, and how you can start playing with Custom Filters today. You should also grab the slides so you can play with the demos yourself.

Introduction to Shaders

I've previously written an introduction to shaders which will give you a good breakdown of what shaders are and how you can use them from a WebGL point of view. If you've never dealt with shaders it's something of a required read before you go much further, because many of the Custom Filters concepts and language hinges on the existing WebGL shader terminology.

So with that said, let's enable Custom Filters and plough on!

Enabling Custom Filters

Custom Filters are available in both Chrome and Canary as well as Chrome for Android. Simply head over to `about:flags` and search for "CSS Shaders", enable them and restart the browser. Now you're good to go!

The syntax

Custom Filters expands on the set of filters that you can already apply, like `blur` or `sepia`, to your DOM elements. Eric Bidelman wrote a great playground tool for those, which you should check out.

To apply a Custom Filter to a DOM element you use the following syntax:

```
.customShader {
  -webkit-filter:

    custom(
      url(vertexshader.vert)
      mix(url(fragment.frag) normal source-atop),

    /* Row, columns - the vertices are made automatically */
    4 5,

    /* We set uniforms; we can't set attributes */
    time 0)
}
```

You'll see from this that we declare our vertex and fragment shaders, the number of rows and columns we want our DOM element to get broken down into, and then any uniforms we want to pass through.

A final thing to point out here is that we use the `mix()` function around the fragment shader with a blend mode (`normal`), and a composite mode (`source-atop`). Let's take a look at the fragment shader itself to see why we even need a `mix()` function.

Pixel Pushing

If you're familiar with WebGL's shaders you'll notice that in Custom Filters things are a little different. For one we don't create the texture(s) that our fragment shader uses to fill in the pixels. Rather the DOM content that has the filter applied gets mapped to a texture automatically, and that means two things:

1. For security reasons we can't query individual pixel color values of the DOM's texture
2. We don't (at least in current implementations) set the final pixel color ourselves, i.e. **gl_FragColor** is off limits. Rather, it's assumed that you will want to render the DOM content, and what you get to do is manipulate its pixels indirectly through **css_ColorMatrix** and **css_MixColor**.

That means our Hello World of fragment shaders looks more like this:

```
void main() {
  css_ColorMatrix = mat4(1.0, 0.0, 0.0, 0.0,
                         0.0, 1.0, 0.0, 0.0,
                         0.0, 0.0, 1.0, 0.0,
                         0.0, 0.0, 0.0, 1.0);

  css_MixColor = vec4(0.0, 0.0, 0.0, 0.0);

  // umm, where did gl_FragColor go?
}
```

Each pixel of the DOM content is multiplied by the **css_ColorMatrix**, which in the above case does nothing as its the identity matrix and changes none of the RGBA values. If we did want to, say, just keep the red values we would use a **css_ColorMatrix** like this:

```
// keep only red and alpha
css_ColorMatrix = mat4(1.0, 0.0, 0.0, 0.0,
                       0.0, 0.0, 0.0, 0.0,
                       0.0, 0.0, 0.0, 0.0,
                       0.0, 0.0, 0.0, 1.0);
```

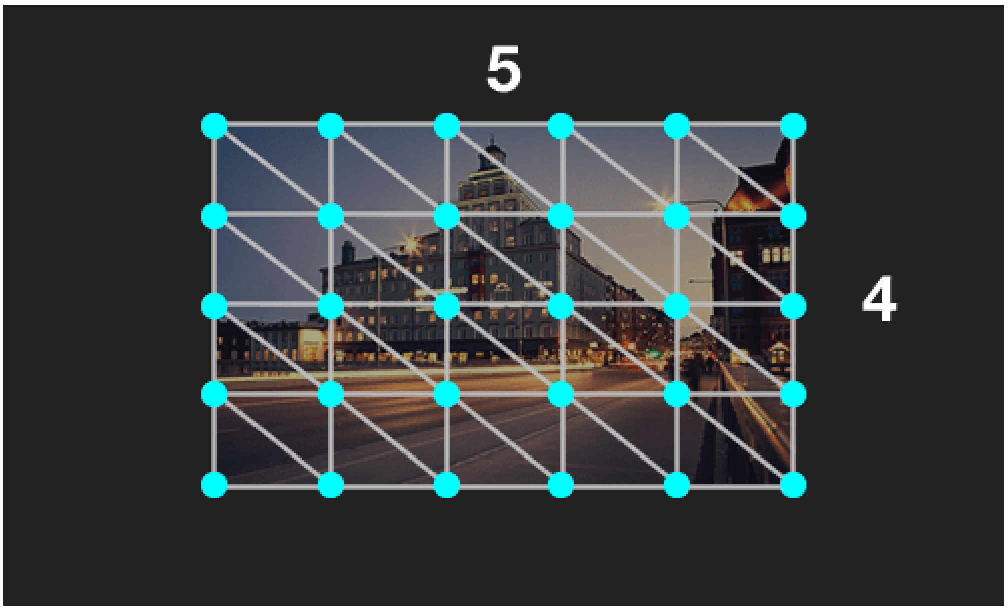
You can hopefully see that as you multiply the 4D (RGBA) pixel values by the matrix that you get a manipulated pixel value out of the other side, and in this case one that zeros out the green and blue components.

The **css_MixColor** is mainly used as a base color that you want to, well, mix in with your DOM content. The mixing is done through the blend modes that you'll be familiar with from art packages: overlay, screen, color dodge, hard light and so on.

There are plenty of ways that these two variables can manipulate the pixels, and included in my presentation is a demo that you can play around with. You should check out the Custom Filters specification to get a better handle on how the blend and composite modes interact.

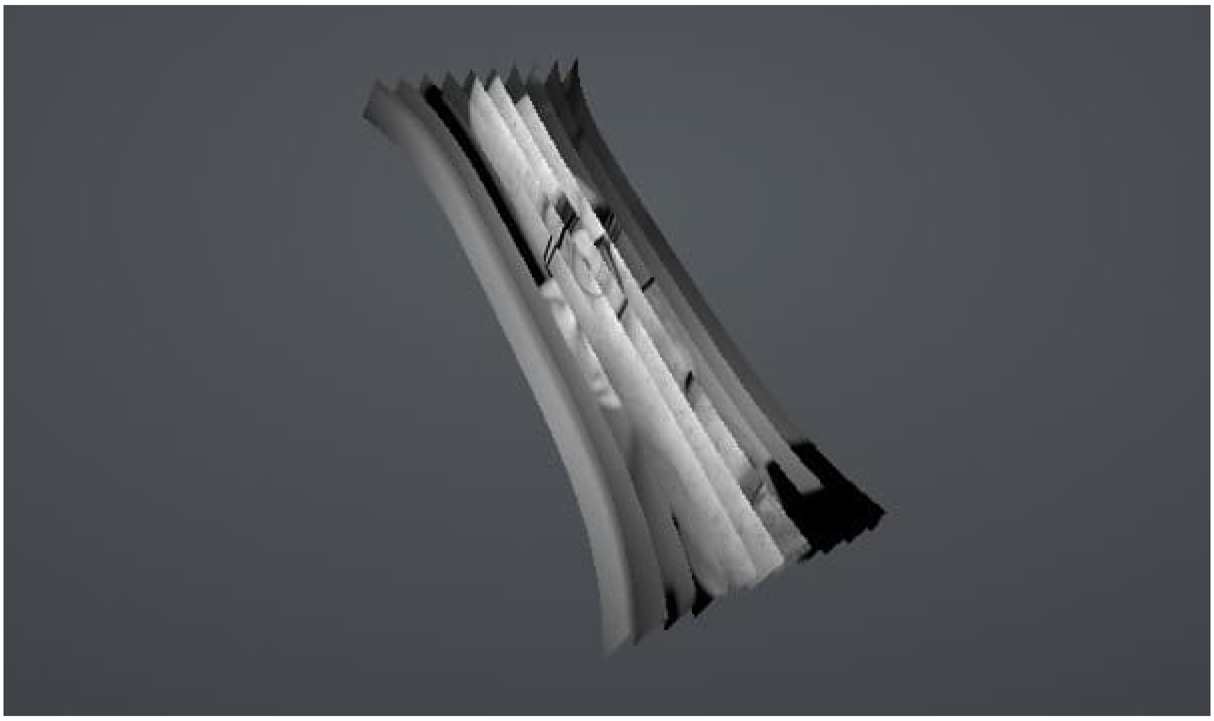
Vertex Creation

In WebGL we take full responsibility for the creation of our mesh's 3D points, but in Custom Filters all you have to do is specify the number of rows and columns that you want and the browser will automatically break down your DOM content into a bunch of triangles:



An image being broken down into rows and columns

Each one of those vertices then gets passed through to our vertex shader for manipulation, and that means we can start moving them around in 3D space as we need. It's not long before you can make some pretty crazy effects!



An image being warped by an accordion effect

Animating with Shaders

Bringing in animations to your shaders is what makes them fun and engaging. To do that you simply use a transition (or animation) in your CSS to update uniform values:

```
.shader {
  /* transition on the filter property */
  -webkit-transition: -webkit-filter 2500ms ease-out;

  -webkit-filter: custom(
    url(vshader.vert)
    mix(url(fshader.frag) normal source-atop),
    1 1,
    time 0);
}

 .shader:hover {
  -webkit-filter: custom(
    url(vshader.vert)
    mix(url(fshader.frag) normal source-atop),
    1 1,
    time 1);
}
```

So the thing to notice in the code above is that time is going to ease from `0` to `1` during the transition. Inside the shader we can declare the uniform `time` and use whatever its current value is:

```
uniform float time;

uniform mat4 u_projectionMatrix;
attribute vec4 a_position;

void main() {
  // copy a_position to position - attributes are read only!
  vec4 position = a_position;

  // use our time uniform from the CSS declaration
  position.x += time;

  gl_Position = u_projectionMatrix * position;
}
```

Get Playing!

Custom Filters are great fun to play with, and the amazing effects you can create are difficult (and in some cases impossible) without them. It is still early days, and things are changing quite a bit, but adding them will add a little bit of showbiz to your projects, so why not give them a go?

Additional Resources

- GL Shader Validator
- Intro to Shaders
- Getting Started with CSS Custom Filters
- Learning WebGL
- Adobe CSS Filter Lab