WebGL Fundamentals

WebGL makes it possible to display amazing realtime 3D graphics in your browser but what many people don't know is that WebGL is actually a 2D API, not a 3D API. Let me explain. WebGL only cares about 2 things. Clipspace coordinates in 2D and colors. Your job as a programmer using WebGL is to provide WebGL with those 2 things. You provide 2 "shaders" to do this. A Vertex shader which provides the clipspace coordinates and a fragment shader that provides the color.

Clipspace coordinates always go from -1 to +1 no matter what size your canvas is. Here is a simple WebGL example that shows WebGL in its simplest form.

```
// Get A WebGL context

var canvas = document.getElementById("canvas");

var gl = canvas.getContext("experimental-webgl");




// setup a GLSL program

var vertexShader = createShaderFromScriptElement(gl, "2d-vertex-
shader");

var fragmentShader = createShaderFromScriptElement(gl, "2d-
fragment-shader");

var program = createProgram(gl, [vertexShader, fragmentShader]);

gl.useProgram(program);




// look up where the vertex data needs to go.
```

```
var positionLocation = gl.getAttribLocation(program,
"a_position");



// Create a buffer and put a single clipspace rectangle in

// it (2 triangles)

var buffer = gl.createBuffer();

gl.bindBuffer(gl.ARRAY_BUFFER, buffer);

gl.bufferData(

    gl.ARRAY_BUFFER,

    new Float32Array([

        -1.0, -1.0,

         1.0, -1.0,

        -1.0,  1.0,

        -1.0,  1.0,

         1.0, -1.0,

         1.0,  1.0]),

    gl.STATIC_DRAW);

gl.enableVertexAttribArray(positionLocation);

gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0,
0);
```

```
// draw

gl.drawArrays(gl.TRIANGLES, 0, 6);
```

Here's the 2 shaders

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">

attribute vec2 a_position;



void main() {

  gl_Position = vec4(a_position, 0, 1);

}

</script>



<script id="2d-fragment-shader" type="x-shader/x-fragment">

void main() {

  gl_FragColor = vec4(0,1,0,1);  // green

}

</script>
```

This will draw a green rectangle the entire size of the canvas. Here it is

click here to open in a separate window

Not very exciting :-p

Again, clipspace coordinates always go from -1 to +1 regardless of the size of the canvas. In the case above you can see we are doing nothing but passing on our position data directly. Since the position data is already in clipspace there is no work to do. **If you want 3D it's up to you to supply shaders that convert from 3D to 2D because WebGL IS A 2D API!!!**

For 2D stuff you would probably rather work in pixels than clipspace so let's change the shader so we can supply rectangles in pixels and have it convert to clipspace for us. Here's the new vertex shader

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">

attribute vec2 a_position;



uniform vec2 u_resolution;



void main() {

   // convert the rectangle from pixels to 0.0 to 1.0

   vec2 zeroToOne = a_position / u_resolution;



   // convert from 0->1 to 0->2

   vec2 zeroToTwo = zeroToOne * 2.0;



   // convert from 0->2 to -1->+1 (clipspace)
```

```
   vec2 clipSpace = zeroToTwo - 1.0;



   gl_Position = vec4(clipSpace, 0, 1);

}

</script>
```

Now we can change our data from clipspace to pixels

```
// set the resolution

var resolutionLocation = gl.getUniformLocation(program,
"u_resolution");

gl.uniform2f(resolutionLocation, canvas.width, canvas.height);



// setup a rectangle from 10,20 to 80,30 in pixels

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([

    10, 20,

    80, 20,

    10, 30,

    10, 30,

    80, 20,

    80, 30]), gl.STATIC_DRAW);
```

And here it is

click here to open in a separate window

You might notice the rectangle is near the bottom of that area. WebGL considers the bottom left corner to be 0,0. To get it to be the more traditional top left corner used for 2d graphics APIs we just flip the y coordinate.

```
gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
```

And now our rectangle is where we expect it.

click here to open in a separate window

Let's make the code that defines a rectangle into a function so we can call it for different sized rectangles. While we're at it we'll make the color settable.

First we make the fragment shader take a color uniform input.

```
<script id="2d-fragment-shader" type="x-shader/x-fragment">

precision mediump float;



uniform vec4 u_color;



void main() {

   gl_FragColor = u_color;

}

</script>
```

And here's the new code that draws 50 rectangles in random places and random colors.

```
...



  var colorLocation = gl.getUniformLocation(program, "u_color");

  ...

  // Create a buffer

  var buffer = gl.createBuffer();

  gl.bindBuffer(gl.ARRAY_BUFFER, buffer);

  gl.enableVertexAttribArray(positionLocation);

  gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0,
0);



  // draw 50 random rectangles in random colors

  for (var ii = 0; ii < 50; ++ii) {

    // Setup a random rectangle

    setRectangle(

        gl, randomInt(300), randomInt(300), randomInt(300),
randomInt(300));



    // Set a random color.

    gl.uniform4f(colorLocation, Math.random(), Math.random(),
Math.random(), 1);
```

```
    // Draw the rectangle.
    gl.drawArrays(gl.TRIANGLES, 0, 6);
  }
}

// Returns a random integer from 0 to range - 1.
function randomInt(range) {
  return Math.floor(Math.random() * range);
}

// Fills the buffer with the values that define a rectangle.
function setRectangle(gl, x, y, width, height) {
  var x1 = x;
  var x2 = x + width;
  var y1 = y;
  var y2 = y + height;
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
     x1, y1,
```

```
      x2, y1,

      x1, y2,

      x1, y2,

      x2, y1,

      x2, y2]), gl.STATIC_DRAW);

}
```

And here's the rectangles.

click here to open in a separate window

I hope you can see that WebGL is actually a pretty simple API. While it can get more complicated to do 3D that complication is added by you, the programmer, in the form of more complex shaders. The WebGL API itself is 2D and fairly simple.

What do type="x-shader/x-vertex" and type="x-shader/x-fragment" mean?

`<script>` tags default to having JavaScript in them. You can put no type or you can put `type="javascript"` or `type="text/javascript"` and the browser will interpret the contents as JavaScript. If you put anything else the browser ignores the contents of the script tag.

We can use this feature to store shaders in script tags. Even better, we can make up our own type and in our javascript look for that to decide whether to compile the shader as a vertex shader or a fragment shader.

In this case the function `createShaderFromScriptElement` looks for a script with specified `id` and then looks at the `type` to decide what type of shader to create.

WebGL Image Processing

Image processing is easy in WebGL. How easy? Read below.

To draw images in WebGL we need to use textures. Similarly to the way WebGL expects clipspace coordinates when rendering instead of pixels, WebGL expects texture coordinates when reading a texture. Texture coordinates go from 0.0 to 1.0 no matter the dimensions of the texture.

Since we are only drawing a single rectangle (well, 2 triangles) we need to tell WebGL which place in the texture each point in the rectangle corresponds to. We'll pass this information from the vertex shader to the fragment shader using a special kind of variable called a 'varying'. It's called a varying because it varies. WebGL will interpolate the values we provide in the vertex shader as it draws each pixel using the fragment shader.

Using the vertex shader from the end of previous section we need to add an attribute to pass in texture coordinates and then pass those on to the fragment shader.

```
attribute vec2 a_texCoord;

...

varying vec2 v_texCoord;



void main() {

   ...

   // pass the texCoord to the fragment shader

   // The GPU will interpolate this value between points

   v_texCoord = a_texCoord;

}
```

Then we supply a fragment shader to look up colors from the texture.

```
<script id="2d-fragment-shader" type="x-shader/x-fragment">

precision mediump float;



// our texture

uniform sampler2D u_image;



// the texCoords passed in from the vertex shader.

varying vec2 v_texCoord;



void main() {

   // Look up a color from the texture.

   gl_FragColor = texture2D(u_image, v_texCoord);

}

</script>
```

Finally we need to load an image, create a texture and copy the image into the texture. Because we are in a browser images load asynchronously so we have to re-arrange our code a little to wait for the texture to load. Once it loads we'll draw it.

```
function main() {

  var image = new Image();
```

```
  image.src = "http://someimage/on/our/server";  // MUST BE SAME
DOMAIN!!!

  image.onload = function() {

    render(image);

  }

}



function render(image) {

  ...

  // all the code we had before.

  ...

  // look up where the texture coordinates need to go.

  var texCoordLocation = gl.getAttribLocation(program,
"a_texCoord");



  // provide texture coordinates for the rectangle.

  var texCoordBuffer = gl.createBuffer();

  gl.bindBuffer(gl.ARRAY_BUFFER, texCoordBuffer);

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([

      0.0,  0.0,
```

```
      1.0,  0.0,

      0.0,  1.0,

      0.0,  1.0,

      1.0,  0.0,

      1.0,  1.0]), gl.STATIC_DRAW);

  gl.enableVertexAttribArray(texCoordLocation);

  gl.vertexAttribPointer(texCoordLocation, 2, gl.FLOAT, false, 0,
0);



  // Create a texture.

  var texture = gl.createTexture();

  gl.bindTexture(gl.TEXTURE_2D, texture);



  // Set the parameters so we can render any size image.

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
```

```
  // Upload the image into the texture.

  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, image);

  ...

}
```

And here's the image rendered in WebGL.

click here to open in a separate window

Not too exciting so let's manipulate that image. How about just swapping red and blue?

```
...

gl_FragColor = texture2D(u_image, v_texCoord).bgra;

...
```

And now red and blue are swapped.

click here to open in a separate window

What if we want to do image processing that actually looks at other pixels? Since WebGL references textures in texture coordinates which go from 0.0 to 1.0 then we can calculate how much to move for 1 pixel with the simple math `onePixel = 1.0 / textureSize`.

Here's a fragment shader that averages the left and right pixels of each pixel in the texture.

```
<script id="2d-fragment-shader" type="x-shader/x-fragment">

precision mediump float;
```

```
// our texture
uniform sampler2D u_image;
uniform vec2 u_textureSize;

// the texCoords passed in from the vertex shader.
varying vec2 v_texCoord;

void main() {
   // compute 1 pixel in texture coordinates.
   vec2 onePixel = vec2(1.0, 1.0) / u_textureSize;

   // average the left, middle, and right pixels.
   gl_FragColor = (
       texture2D(u_image, v_texCoord) +
       texture2D(u_image, v_texCoord + vec2(onePixel.x, 0.0)) +
       texture2D(u_image, v_texCoord + vec2(-onePixel.x, 0.0))) /
3.0;
}
</script>
```

We then need to pass in the size of the texture from JavaScript.

```
...

var textureSizeLocation = gl.getUniformLocation(program,
"u_textureSize");

...

// set the size of the image

gl.uniform2f(textureSizeLocation, image.width, image.height);

...
```

Compare to the un-blurred image above.

click here to open in a separate window

Now that we know how to reference other pixels let's use a convolution kernel to do a bunch of common image processing. In this case we'll use a 3x3 kernel. A convolution kernel is just a 3x3 matrix where each entry in the matrix represents how much to multiply the 8 pixels around the pixel we are rendering. We then divide the result by the weight of the kernel (the sum of all values in the kernel) or 1.0, which ever is greater. Here's a pretty good article on it. And here's another article showing some actual code if you were to write this by hand in C++.

In our case we're going to do that work in the shader so here's the new fragment shader.

```
<script id="2d-fragment-shader" type="x-shader/x-fragment">

precision mediump float;



// our texture
```

```
uniform sampler2D u_image;

uniform vec2 u_textureSize;

uniform float u_kernel[9];



// the texCoords passed in from the vertex shader.

varying vec2 v_texCoord;



void main() {

   vec2 onePixel = vec2(1.0, 1.0) / u_textureSize;

   vec4 colorSum =

     texture2D(u_image, v_texCoord + onePixel * vec2(-1, -1)) *
u_kernel[0] +

     texture2D(u_image, v_texCoord + onePixel * vec2( 0, -1)) *
u_kernel[1] +

     texture2D(u_image, v_texCoord + onePixel * vec2( 1, -1)) *
u_kernel[2] +

     texture2D(u_image, v_texCoord + onePixel * vec2(-1,  0)) *
u_kernel[3] +

     texture2D(u_image, v_texCoord + onePixel * vec2( 0,  0)) *
u_kernel[4] +

     texture2D(u_image, v_texCoord + onePixel * vec2( 1,  0)) *
u_kernel[5] +

     texture2D(u_image, v_texCoord + onePixel * vec2(-1,  1)) *
u_kernel[6] +
```

```
    texture2D(u_image, v_texCoord + onePixel * vec2( 0,  1)) *
u_kernel[7] +

    texture2D(u_image, v_texCoord + onePixel * vec2( 1,  1)) *
u_kernel[8] ;

  float kernelWeight =

    u_kernel[0] +

    u_kernel[1] +

    u_kernel[2] +

    u_kernel[3] +

    u_kernel[4] +

    u_kernel[5] +

    u_kernel[6] +

    u_kernel[7] +

    u_kernel[8] ;



  if (kernelWeight <= 0.0) {

    kernelWeight = 1.0;

  }



  // Divide the sum by the weight but just use rgb
```

```
   // we'll set alpha to 1.0

   gl_FragColor = vec4((colorSum / kernelWeight).rgb, 1.0);

}

</script>
```

In JavaScript we need to supply a convolution kernel.

```
...

var kernelLocation = gl.getUniformLocation(program,
"u_kernel[0]");

...

var edgeDetectKernel = [

    -1, -1, -1,

    -1,  8, -1,

    -1, -1, -1

];

gl.uniform1fv(kernelLocation, edgeDetectKernel);

...
```

And voila... Use the drop down list to select different kernels.

click here to open in a separate window

I hope this has convinced you image processing in WebGL is pretty simple. Next up I'll go over how to apply more than one effect to the image.

What's with the a_, u_, and v_ prefixes in from of variables in GLSL?

That's just a naming convention. a_ for attributes which is the data provided by buffers. u_ for uniforms which are inputs to the shaders, v_ for varyings which are values passed from a vertex shader to a fragment shader and interpolated (or varied) between the vertices for each pixel drawn.

Applying multiple effects

The next most obvious question for image processing is how do apply multiple effects? Well, you could try to generate shaders on the fly. Provide a UI that lets the user select the effects he wants to use then generate a shader that does all of the effects. That might not always be possible though that technique is often used to create effects for real time graphics.

A more flexible way is to use 2 more textures and render to each texture in turn, ping ponging back and forth and applying the next effect each time.

```
Original Image -> [Blur]         -> Texture 1
Texture 1      -> [Sharpen]      -> Texture 2
Texture 2      -> [Edge Detect]  -> Texture 1
Texture 1      -> [Blur]         -> Texture 2
Texture 2      -> [Normal]       -> Canvas
```

To do this we need to create framebuffers. In WebGL and OpenGL, a Framebuffer is actually a poor name. A WebGL/OpenGL Framebuffer is really just a collection of state and not actually a buffer of any kind. But, by attaching a texture to a framebuffer we can render into that texture.

First let's turn the old texture creation code into a function

```
function createAndSetupTexture(gl) {

  var texture = gl.createTexture();

  gl.bindTexture(gl.TEXTURE_2D, texture);

```

```
  // Set up texture so we can render any size image and so we are

  // working with pixels.

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST);

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);




  return texture;

}




// Create a texture and put the image in it.

var originalImageTexture = createAndSetupTexture(gl);

gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, image);
```

And now let's use that function to make 2 more textures and attach them to 2 framebuffers.

```
// create 2 textures and attach them to framebuffers.

var textures = [];

var framebuffers = [];
```

```
for (var ii = 0; ii < 2; ++ii) {

  var texture = createAndSetupTexture(gl);

  textures.push(texture);



  // make the texture the same size as the image

  gl.texImage2D(

      gl.TEXTURE_2D, 0, gl.RGBA, image.width, image.height, 0,

      gl.RGBA, gl.UNSIGNED_BYTE, null);



  // Create a framebuffer

  var fbo = gl.createFramebuffer();

  framebuffers.push(fbo);

  gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);



  // Attach a texture to it.

  gl.framebufferTexture2D(

      gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D,
texture, 0);

}
```

Now let's make a set of kernels and then a list of them to apply.

```
// Define several convolution kernels

var kernels = {

  normal: [

    0, 0, 0,

    0, 1, 0,

    0, 0, 0

  ],

  gaussianBlur: [

    0.045, 0.122, 0.045,

    0.122, 0.332, 0.122,

    0.045, 0.122, 0.045

  ],

  unsharpen: [

    -1, -1, -1,

    -1,  9, -1,

    -1, -1, -1

  ],

  emboss: [
```

```
    -2, -1,  0,
    -1,  1,  1,
     0,  1,  2
  ]
};

// List of effects to apply.
var effectsToApply = [
  "gaussianBlur",
  "emboss",
  "gaussianBlur",
  "unsharpen"
];
```

And finally let's apply each one, ping ponging which texture we are rendering too

```
// start with the original image
gl.bindTexture(gl.TEXTURE_2D, originalImageTexture);

// don't y flip images while drawing to the textures
gl.uniform1f(flipYLocation, 1);
```

```
// loop through each effect we want to apply.
for (var ii = 0; ii < effectsToApply.length; ++ii) {
  // Setup to draw into one of the framebuffers.
  setFramebuffer(framebuffers[ii % 2], image.width,
image.height);

  drawWithKernel(effectsToApply[ii]);

  // for the next draw, use the texture we just rendered to.
  gl.bindTexture(gl.TEXTURE_2D, textures[ii % 2]);
}

// finally draw the result to the canvas.
gl.uniform1f(flipYLocation, -1);  // need to y flip for canvas
setFramebuffer(null, canvas.width, canvas.height);
drawWithKernel("normal");

function setFramebuffer(fbo, width, height) {
  // make this the framebuffer we are rendering to.
```

```
  gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);



  // Tell the shader the resolution of the framebuffer.

  gl.uniform2f(resolutionLocation, width, height);



  // Tell webgl the viewport setting needed for framebuffer.

  gl.viewport(0, 0, width, height);

}



function drawWithKernel(name) {

  // set the kernel

  gl.uniform1fv(kernelLocation, kernels[name]);



  // Draw the rectangle.

  gl.drawArrays(gl.TRIANGLES, 0, 6);

}
```

Here's a working version with a slightly more flexible UI. Check the effects to turn them on. Drag the effects to reorder how they are applied.

click here to open in a separate window

Some things I should go over.

Calling `gl.bindFramebuffer` with `null` tells WebGL you want to render to the canvas instead of to one of your framebuffers.

WebGL has to convert from clipspace back into pixels. It does this based on the settings of `gl.viewport`. The settings of `gl.viewport` default to the size of the canvas when we initialize WebGL. Since the framebuffers we are rendering into are a different size then the canvas we need to set the viewport appropriately.

Finally in the WebGL fundamentals examples we flipped the Y coordinate when rendering because WebGL displays the canvas with 0,0 being the bottom left corner instead of the more traditional for 2D top left. That's not needed when rendering to a framebuffer. Because the framebuffer is never displayed, which part is top and bottom is irrelevant. All that matters is that pixel 0,0 in the framebuffer corresponds to 0,0 in our calculations. To deal with this I made it possible to set whether to flip or not by adding one more input into the shader.

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">

...

uniform float u_flipY;

...



void main() {

   ...

   gl_Position = vec4(clipSpace * vec2(1, u_flipY), 0, 1);

   ...

}
```

```
</script>
```

And then we can set it when we render with

```
...

var flipYLocation = gl.getUniformLocation(program, "u_flipY");

...

// don't flip

gl.uniform1f(flipYLocation, 1);

...

// flip

gl.uniform1f(flipYLocation, -1);
```

I kept this example simple by using single GLSL program that can achieve multiple effects. If you wanted to do full on image processing you'd probably need many GLSL programs. A program for hue, saturation and luminance adjustment. Another for brightness and contrast. One for inverting, another for adjusting levels, etc. You'd need to change the code to switch GLSL programs and update the parameters for that particular program. I'd considered writing that example but it's an exercise best left to the reader because multiple GLSL programs each with their own parameter needs probably means some major refactoring to keep it all from becoming a big mess of spaghetti.

I hope this and the preceding examples have made WebGL seem a little more approachable and I hope starting with 2D helps make WebGL a little easier to understand. If I find the time I'll try to write a few more articles about how to do 3D as well as more details on what WebGL is really doing under the hood.

WebGL and Alpha

I've noticed some OpenGL developers having issues with how WebGL treats alpha in the backbuffer (ie, the canvas), so I thought it might be good to go over some of the differences between WebGL and OpenGL related to alpha.

The biggest difference between OpenGL and WebGL is that OpenGL renders to a backbuffer that is not composited with anything so, or effectively not composited with anything by the OS's window manager, so it doesn't matter what your alpha is.

WebGL is composited by the browser with the web page and the default is to use pre-multiplied alpha the same as .png <img> tags with transparency and 2d canvas tags.

WebGL has several ways to make this more like OpenGL.

#1) Tell WebGL you want it composited with non-premultiplied alpha

```
gl = canvas.getContext("experimental-webgl", {premultipliedAlpha: false});
```

The default is true.

Of course the result will still be composited over page with whatever background color ends up being under the canvas (the canvas's background color, the canvas's container background color, the page's background color, the stuff behind the canvas if the canvas has a z-index > 0, etc....) in other words, the color CSS defines for that area of the webpage.

I really good way to find if you have any alpha problems is to set the canvas's background to a bright color like red. You'll immediately see what is happening.

```
<canvas style="background: red;"></canvas>
```

You could also set it to black which will hide any alpha issues you have.

#2) Tell WebGL you don't want alpha in the backbuffer

```
gl = canvas.getContext("experimental-webgl", {alpha: false});
```

This will make it act more like OpenGL since the backbuffer will only have RGB. This is probably the best option because a good browser could see that you have no alpha and actually optimize the way WebGL is composited. Of course that also means it actually won't have alpha in the backbuffer so if you are using alpha in the backbuffer for some purpose that might not work for you. Few apps that I know of use alpha in the backbuffer. I think arguably this should have been the default.

#3) Clear alpha at the end of your rendering

```
..

renderScene();

..

// Set the backbuffer's alpha to 1.0

gl.clearColor(1, 1, 1, 1);

gl.colorMask(false, false, false, true);

gl.clear(gl.COLOR_BUFFER_BIT);
```

Clearing is generally very fast as there is a special case for it in most hardware. I did this in most of my demos. If I was smart I'd switch to method #2 above. Maybe I'll do that right after I post this. It seems like most WebGL libraries should default to this method. Those few developers that are actually using alpha for compositing effects can ask for it. The rest will just get the best perf and the least surprises.

#4) Clear the alpha once then don't render to it anymore

```
// At init time. Clear the back buffer.

gl.clearColor(1,1,1,1);

gl.clear(gl.COLOR_BUFFER_BIT);



// Turn off rendering to alpha

gl.colorMask(true, true, true, false);
```

Of course if you are rendering to your own framebuffers you may need to turn rendering to alpha back on and then turn it off again when you switch to rendering to the canvas.

#5) Handling Images

Also, if you are loading PNG files with alpha into textures, the default is that their alpha is pre-multiplied which is generally NOT the way most games do things. If you want to prevent that behavior you need to tell WebGL with

```
gl.pixelStorei(gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL, false);
```

#6) Using a blending equation that works with pre-multiplied alpha.

Almost all OpenGL apps I've writing or worked on use

```
gl.blendFunc(gl.SRC_ALPHA, gl_ONE_MINUS_SRC_ALPHA);
```

That works for non pre-multiplied alpha textures.

If you actually want to work with pre-multiplied alpha textures then you probably want

```
gl.blendFunc(gl.ONE, gl_ONE_MINUS_SRC_ALPHA);
```

Those are the methods I'm aware of. If you know of more please post them below.