Advanced Physics

Assignment 2: Shader programming.

Alberto Martínez de Murga Ramírez, Stud. nr. 500693826

1. Make the shader display your webcam picture1 by calculating fragment uv coordinates (from pixel range to 0-1 range) and setting `gl_FragColor` to sample the channel texture.

I followed the examples in the tutorial and I realised that the exercise was very similar to the flatten shader example. It was a direct translation pixel by pixel from the original image once scaled to the `gl_FragColor`.

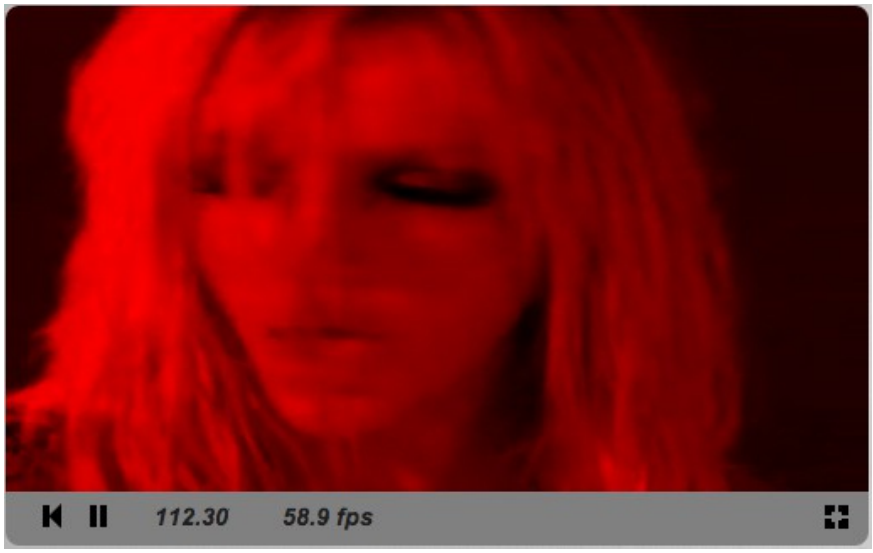
```
void main(void)
{
      vec2 uv = gl_FragCoord.xy / iResolution.xy;
      vec4 color = texture2D(iChannel0,uv);
    gl_FragColor = color;
}
```

2. Make the shader output only the red component of the video.

This was easy. As each colour is created by the mix of three base colours in RGB (Red, Green, Blue) mode, if I only want the red component, I just have to make the other two 0.

```
void main(void)
{
      vec2 uv = gl_FragCoord.xy / iResolution.xy;
      vec4 color = texture2D(iChannel0,uv);
      color.y = 0.0;
```

```
        color.z = 0.0;
    gl_FragColor = color;
}
```



3. Mirror the image horizontally, so that you can use your webcam like a mirror (as most webcam programs do).

I had the idea in mind that I need to look for the complementary value of each pixel, and after a little research, I found the answer about how to calculate the complementary value here: http://irrlicht.sourceforge.net/forum/viewtopic.php?f=4&t=23265

```
void main(void)
{
      vec2 uv = gl_FragCoord.xy / iResolution.xy;
      uv.x = 1.0 - uv.x;
      vec4 color = texture2D(iChannel0,uv);
    gl_FragColor = color;
}
```

4. Invert the colours.

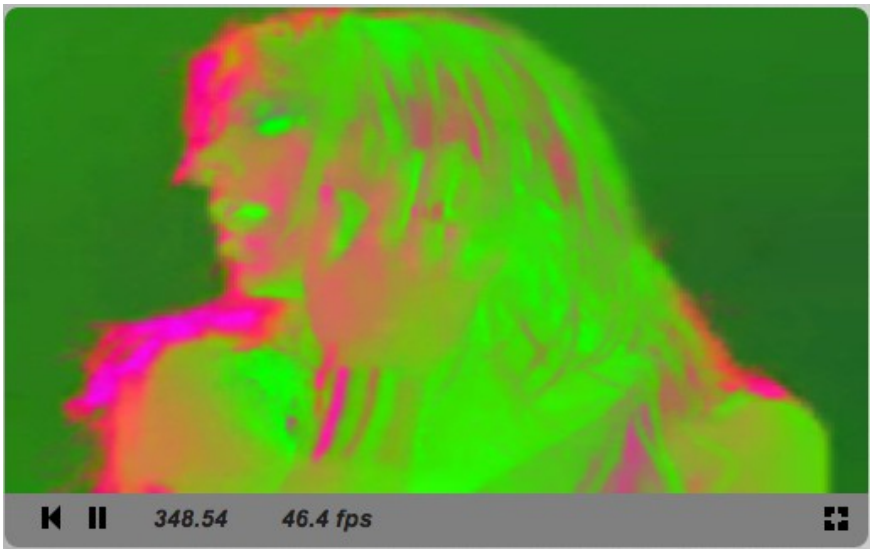
I tried the same idea (1 - value) as in the previous example and it was right.

```
void main(void)
{
	vec2 uv = gl_FragCoord.xy / iResolution.xy;
	vec4 color = texture2D(iChannel0,uv);
	color.x = 1.0 - color.x;
	color.y = 1.0 - color.y;
	color.z = 1.0 - color.z;
	gl_FragColor = color;
}
```

5. Now invert only the green channel.

This one was a mix of the solutions of exercises 2 and 4.

```
void main(void)
{
	vec2 uv = gl_FragCoord.xy / iResolution.xy;
	vec4 color = texture2D(iChannel0,uv);
	color.y = 1.0 - color.y;
	gl_FragColor = color;
}
```



6. Revert colour changes; scale the picture by 0.5 horizontally, centre the scaled picture, and let the area outside the picture be black.

The scaling and moving part was more or less intuitive because at this point I realised that what a shader does is to apply a function to every pixel, in this case a linear function with an offset. After that, I checked if the uv where out of range and I painted them in black.

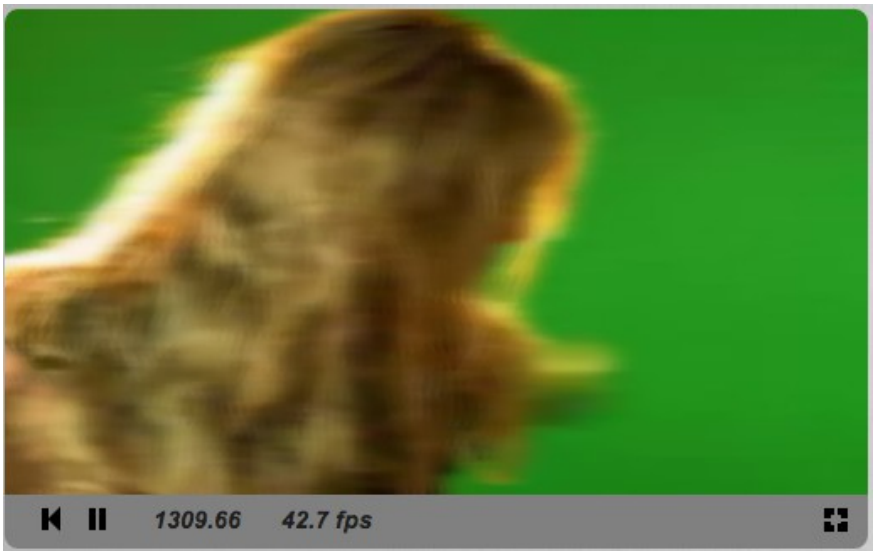
```
void main(void)
{
	vec2 uv = gl_FragCoord.xy / iResolution.xy;
	uv.x = uv.x*2.0 - 0.5;
	vec4 color = texture2D(iChannel0,uv);
	if(uv.x < 0.0 || uv.x > 1.0 || uv.y < 0.0 || uv.y > 1.0){
		color *= 0.0;
	}
	gl_FragColor = color;
}
```

7. Create a horizontal blur effect by sampling not only the current pixel from the texel, but the neighbouring pixels as well, weighing 0.4 for the pixel itself, 0.2 for the pixels next to it, and 0.1 for the pixels at a distance of 2.

I had to make some research on the internet and after finding this example, http://www.gamerendering.com/2008/10/11/gaussian-blur-filter-shader/, I adapted it to my necessities following the same philosophy. Essentially, I am calculating each pixel texture and multiplying its value by 0.4 and at the same time, to the adjacent pixels. At first I tried with referenced values, but the result with fixed values for the adjacent pixels were better.

```
void main(void)
{
	vec2 uv = gl_FragCoord.xy / iResolution.xy;
	vec4 sum = vec4(0.0);
	sum += texture2D(iChannel0, vec2(uv.x - 0.02, uv.y)) * 0.1;
	sum += texture2D(iChannel0, vec2(uv.x - 0.01, uv.y)) * 0.2;
	sum += texture2D(iChannel0, vec2(uv.x, uv.y)) * 0.4;
	sum += texture2D(iChannel0, vec2(uv.x + 0.01, uv.y)) * 0.2;
	sum += texture2D(iChannel0, vec2(uv.x + 0.02, uv.y)) * 0.1;
	gl_FragColor = sum;

}
```



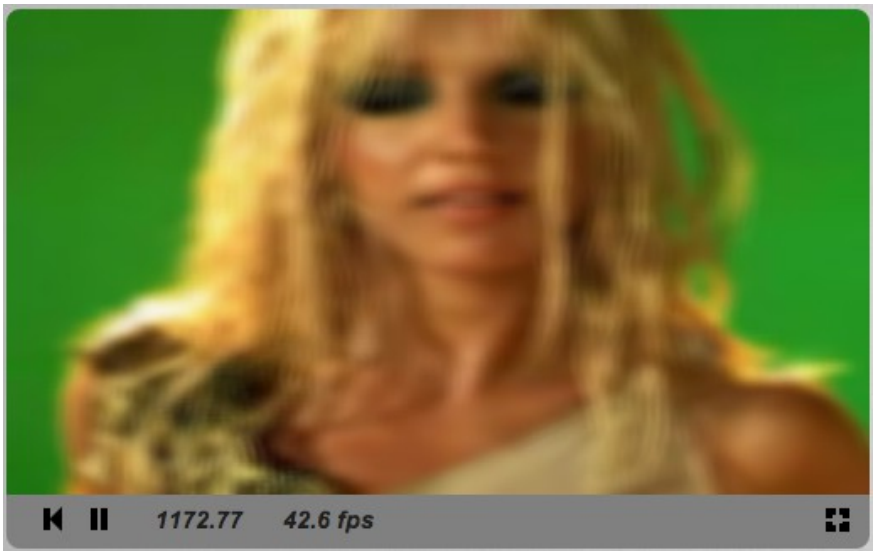
8. Expand the blur filter to also weigh vertically adjacent pixels.
This is a variation of the previous exercise.

```
void main(void)
{
	vec2 uv = gl_FragCoord.xy / iResolution.xy;

	vec4 sum = vec4(0.0);
	// First row
	sum += texture2D(iChannel0, vec2(uv.x - 0.01, uv.y - 0.01)) * 0.1;
```

```
        sum += texture2D(iChannel0, vec2(uv.x       , uv.y - 0.01))  * 0.1;
        sum += texture2D(iChannel0, vec2(uv.x + 0.01, uv.y - 0.01)) * 0.1;
        // Second row
        sum += texture2D(iChannel0, vec2(uv.x - 0.01, uv.y)) * 0.1;
        sum += texture2D(iChannel0, vec2(uv.x       , uv.y)) * 0.2;
        sum += texture2D(iChannel0, vec2(uv.x + 0.01, uv.y)) * 0.1;
        // Third row
        sum += texture2D(iChannel0, vec2(uv.x - 0.01, uv.y + 0.01)) * 0.1;
        sum += texture2D(iChannel0, vec2(uv.x       , uv.y + 0.01))  * 0.1;
        sum += texture2D(iChannel0, vec2(uv.x + 0.01, uv.y + 0.01)) * 0.1;
        gl_FragColor = sum;

    }
```



9. Distort the image using a sine wave, then use iGlobalTime to make the sine wave move over time.

I remembered something about the transverse wave formula (f(x, t) = A sin(wt – kx); u(x, t) = F(x - vt) + G(x + vt))from my high-school physics class and after some research and try and error, I found the solution.

```
    void main(void)
    {
          vec2 uv = gl_FragCoord.xy / iResolution.xy;
        // u(x,t) = f(x - vt) + g(x + vt)
          uv.y += (sin((uv.x + (iGlobalTime * 0.25)) * 10.0) * 0.05)
            + (sin((uv.x + (iGlobalTime * 0.25)) * 10.0) * 0.05);
          vec4 color = texture2D(iChannel0,uv);

          gl_FragColor = color;

    }
```

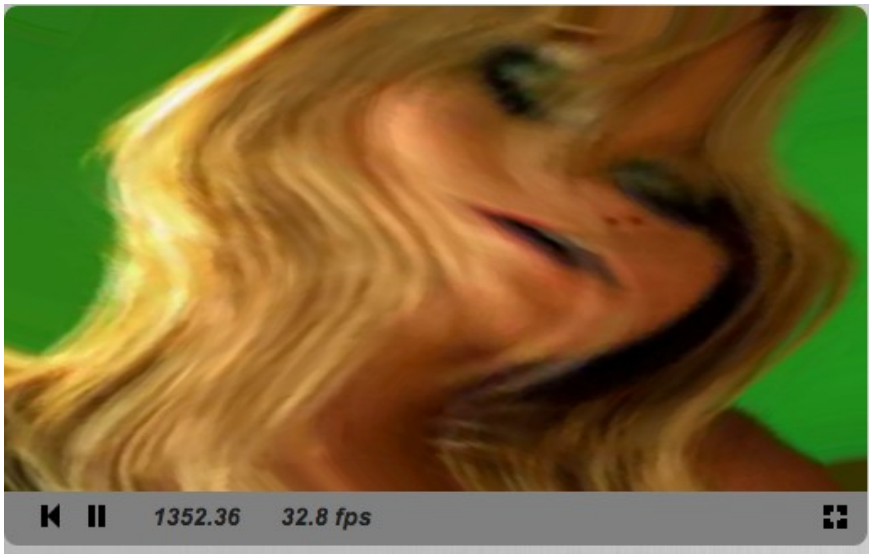
10. Combine multiple horizontal and vertical sine waves with different wavelengths and speeds in order to achieve something that looks like water ripples.

Variation of the previous exercise. Sadly, I was not able to make it look a like to real water, this was my better approach.

```
void main(void)
{
      vec2 uv = gl_FragCoord.xy / iResolution.xy;
    // u(x,t) = f(x - vt) + g(x + vt)
      uv.y += (sin((uv.x + (iGlobalTime * 0.25)) * 10.0) * 0.015)
        + (sin((uv.x + (iGlobalTime * 0.25)) * 10.0) * 0.015);
      uv.x += (sin((uv.y + (iGlobalTime * 0.25)) * 10.0) * 0.015)
        + (sin((uv.y + (iGlobalTime * 0.25)) * 10.0) * 0.015);
      uv.x += (sin((uv.y + (iGlobalTime * 0.25)) * 10.0) * 0.015)
        + (sin((uv.y + (iGlobalTime * 0.25)) * 10.0) * 0.015);
      uv.y += (sin((uv.x + (iGlobalTime * 0.25)) * 10.0) * 0.015)
        + (sin((uv.x + (iGlobalTime * 0.25)) * 10.0) * 0.015);
      vec4 color = texture2D(iChannel0,uv);

      gl_FragColor = color;

}
```



11.Analyse the way 3D vertices are projected in the shader at:

The vertex of the cube and the floor are modelled in triangles, which are transformed to the clip space. After, the area of the sub triangle is calculated to check if a point it is inside or outside the shape. If it is inside, that means that must be rendered. The barycentric coords are calculated to get the interpolation and the 1/z. These are the values needed to apply the textures and lights. For each triangle the textures are applied knowing the normal, the uv, the interpolation and the z value.