

Palette-based Photo Recoloring

Alejandra Cristina Callo Aguilar

Maestría en Ciencia de la computación

1 Introducción

Se presenta una GUI que crea una paleta de colores a partir de una imagen, usa una clasificación de colores en base al algoritmo k-means y la transferencia de colores entre píxeles; se recolorea la imagen creando texturas según el nuevo color seleccionado por el usuario.



2 Implementación

Se reproduce el presente artículo en WebGL basado en OpenGL 2.0 soportado en los diferentes navegadores en un espacio HTML en código JavaScript.

2.1 Interfaz

Se genera un espacio dentro de HTML el cual denominaremos ***canvas*** en el que se recibe la imagen de entrada y un espacio ***glcanvas*** en el que se guardará y mostrará la salida en un contexto WebGL que sera renderizado.

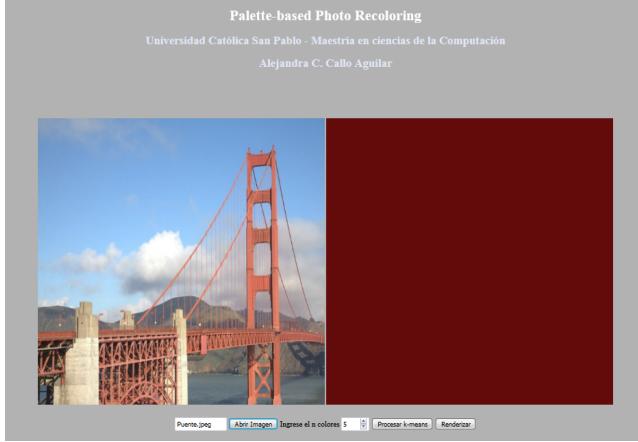


Figure 1: Interfaz

La interfaz es simple e intuitiva ya que recibe un archivo de entrada y presenta en botones los colores en predominancia, los que se han de clasificar según el número de k-means seleccionado. El botón renderizar procesará los cambios para el Recoloreando de la imagen.

```

image_test.html  ●  style.css  ✘
<html>
<head>
    <title>Proyecto webGL</title>
    <link rel="stylesheet" href="style.css" type="text/css" />
</head>
<body>
    <h1>Palette-based Photo Recoloring </h1>
    <h2>Universidad Católica San Pablo - Maestría en ciencias de la Computación </h2>
    <h2>Alejandra C. Callo Aguilar</h2>
    <div id="main">
        <canvas id="canvas" width="600" height="450"></canvas>
        <table border="1">
            <tr>
                <td><input type="color" id="color1" hidden="hidden" oninput="app.Ale.setColor(1)"/></td>
                <td><input type="color" id="color2" hidden="hidden" oninput="app.Ale.setColor(2)"/></td>
            </tr>
            <tr>
                <td><input type="color" id="color3" hidden="hidden" oninput="app.Ale.setColor(3)"/></td>
                <td><input type="color" id="color4" hidden="hidden" oninput="app.Ale.setColor(4)"/></td>
            </tr>
            <tr>
                <td><input type="color" id="color5" hidden="hidden" oninput="app.Ale.setColor(5)"/></td>
                <td><input type="color" id="color6" hidden="hidden" oninput="app.Ale.setColor(6)"/></td>
            </tr>
            <tr>
                <td><input type="color" id="color7" hidden="hidden" oninput="app.Ale.setColor(7)"/></td>
                <td><input type="color" id="color8" hidden="hidden" oninput="app.Ale.setColor(8)"/></td>
            </tr>
            <tr>
                <td><input type="color" id="color9" hidden="hidden" oninput="app.Ale.setColor(9)"/></td>
                <td><input type="color" id="color10" hidden="hidden" oninput="app.Ale.setColor(10)"/></td>
            </tr>
        </table>
        <br />
        <input style="width:100px; height:20px" type="text" id="texto" value="entropy.jpg" />
        <script src="gl-matrix.js"></script>
        <script src="Color_utils.js"></script>
        <script src="kmeans.js"></script>
        <script src="web_gl.js"></script>
        <button onclick="app.loadPicture()"> Abrir Imagen</button>
        <label for="in_num_k"> Ingrese el n colores</label>
        <input style="width:50px; height:20px" id="n_" type="number" min="3" max="10" value="5">
        <button onclick="app.Ale.procesar( document.getElementById('n_').value )"> Procesar k-means </button>
        <button onclick="main()> Renderizar</button>
    </div>
</body>

```

Figure 2: index.html

3 Implementación

Se muestra la implementación del recoloreado de imágenes mediante el uso de kmeans. Y con la ayuda de webgl para la renderización de estas.

3.1 De la paleta de Colores

La paleta de colores muestra los colores en predominancia de la imagen de entrada, la cantidad de colores de esta paleta depende de el número de k-means en los que se clasificará; el mínimo tamaño de la paleta es 3 y el máximo es 7, adicionalmente el color negro se mantiene en un vector de correspondencia y no se modifica.

3.2 Iniciar centroides

Se usa un método deterministico para la creación inicial de los k-means. Para lo cual empecemos a usar un contenedor representativo del espacio RGB y hallamos sus respectivos valores LAB. Se llena este contenedor aumentando en 1 por cada píxel que corresponda al contenedor.

```
public.Ale.centros = (k) => {
    k = Number(k);
    var means = [];
    //Creamos bins(contenedores)
    public.Ale.cargarBins();
    means[0] = { l: 0, a: 0, b: 0 };
    //Iterar por cada k
    for (var j = 1; j < k+1 ; j++) {
        var max = 0, indice = 0;
        //Hallamos el mayor
        for (var i = 1; i < 4096; i++) {
            if (max < bins[i]) {
                indice = i;
                max = bins[i];
            }
        }
        color = bins_color[indice];
        bins[indice] = 0;
        //Atenuamos los valores
        for (var i = 0; i < 4096; i++) {
            if (bins[i] == 0) continue;
            var delta = deltani(color, bins_color[i]);
            bins[i] *= (1 - Math.exp(-Math.pow(delta, 2) / Math.pow(80, 2)));
        }
        means[j] = bins_color[indice];
    }
    return means;
}
```

Figure 3: Inicialización de Centros(means)

Después de esto se selecciona el color con el mayor peso, luego se atenúa los demás pesos de acuerdo a la distancia que se tiene del color seleccionado; dentro

de este nuevo grupo se selecciona el siguiente con el mayor peso para repetir el proceso hasta que se tengan los K centroides.

3.3 Del procesamiento de los Kmeans

Para la obtención de los k-means se empieza con sus valores inicializados como se menciono en el punto anterior. Usando el contenedor de colores hallamos el promedio de todos los colores que estén mas cerca a un determinado k-mean actualizamos este con el nuevo valor obtenido Y así por cada Kmeans. Una vez que el valor obtenido es el mismo del que ya se tiene para cada kmeans se detiene la iteración.

```

do {
    var old_means = public.Ale.copiar(means);
    //vaciamos contadores
    for (var i = 0; i < K + 1; i++) {
        C_L[i] = 0;
        C_A[i] = 0;
        C_B[i] = 0;
        C[i] = 0;
    }
    //calculamos a que grupo pertenecen
    for (var i = 0; i < 4096; i++) {
        correspondencias[i] = public.Ale.mas_cercano(bins_color[i], means);
    }
    //sumamos contadores
    for (var i = 0; i < 4096; i++) {
        C_L[correspondencias[i]] += bins_color[i].l * bins[i];
        C_A[correspondencias[i]] += bins_color[i].a * bins[i];
        C_B[correspondencias[i]] += bins_color[i].b * bins[i];
        C[correspondencias[i]] += bins[i];
    }
    for (var i = 1; i < K+1 ; i++) {
        if (C[i] != 0) {
            means[i].l = (C_L[i] / C[i]);
            means[i].a = (C_A[i] / C[i]);
            means[i].b = (C_B[i] / C[i]);
        }
    }
} while (public.Ale.diferencial(means, old_means) > 0)

```

Figure 4: Iteración del K-means

3.4 Del proceso de la clasificación

Una vez hallado los K-means. Se empieza un proceso de clasificación de los pixeles de la imagen por su correspondiente. Para esto cada Pixel es transformado es su equivalente LAB. Se halla su k-mean mas cercano a este. y guardado en un vector de correspondencias.

Basado en el vector de correspondencias se llenan subimagenes. Estos posteriormente serán guardados como texturas tal que la suma de todas de a la imagen original.

```

public.Ale.Dividir = () => {
    var images = [];
    var imagen_base = app.getImgData();
    for (var i = 0; i < 10; i++) {
        imagen_temp = app.getImgData();
        images.push(imagen_temp);
    }
    numPixels = images[0].width * images[0].height;

    for (var j = 0; j < 10; j++) {
        for (var i = 0; i < numPixels; i++) {
            if (correspondencias[i] != j) {
                images[j].data[i * 4] = 0;
                images[j].data[i * 4 + 1] = 0;
                images[j].data[i * 4 + 2] = 0;
            }
        }
    }
    imagenes = images;
}

```

Figure 5: usando correspondencias para dividir la imagen

3.5 De la transferencia de colores

Debido a que se trabaja en el espacio de colores CIELAB. Se tomaran en cuenta los valores 'a' y 'b' para lo cual se utiliza una función de transferencia tomando en cuenta el desplazamiento relativo del centroide hacia el nuevo color escogido.

```

for (var i = 0; i < numpixels; i++) {
    if (pixels[i * 4] == 0 && pixels[i * 4 + 1] == 0 && pixels[i * 4 + 2] == 0) {
        continue;
    } else {
        x = rgb2lab(pixels[i * 4], pixels[i * 4 + 1], pixels[i * 4 + 2]);
        x0_a = x.a + C_.a - c_.a;
        x0_b = x.b + C_.b - c_.b;
        if (estaDentro(x0_a, x0_b)) {
            //far case
            xb_ab = puntoInterseccion(x.a, x.b, x0_a, x0_b);
        } else {
            //near case
            xb_ab = puntoInterseccion(c_.a, c_.b, x0_a, x0_b);
        }
        x_ab = f1(C, cb_ab, C_, x, xb_ab);
        ncolor = lab2rgb(x.l, x_ab.a, x_ab.b);
        pixels[i * 4] = ncolor.r;
        pixels[i * 4 + 1] = ncolor.g;
        pixels[i * 4 + 2] = ncolor.b;
    }
}
imagenes[indice].data = pixels;
public.Ale.actualizarPaletas();
}

```

Figure 6: f1(x)

3.6 De la Luminosidad

Se mantiene un orden de acuerdo a la luminosidad de los centroides y este cada vez que se actualiza también se actualizan la luminosidad de los otros centroides para que se mantenga el orden. Mientras que en la función de transferencia. El nuevo valor de la luminosidad del píxel se mantiene.

```
public.Ale.modificarL = (indice) => {
    var l = means_salida.length;
    for (var i = indice + 1; i < l; i++) {
        means_salida[i].l = Math.max(means_salida[i].l, means_salida[i - 1].l);
    }

    for (var i = indice - 1; i > 0; i--) {
        means_salida[i].l = Math.min(means_salida[i].l, means_salida[i + 1].l);
    }
}
```

Figure 7: función de Luminosidad

3.7 De la Creación de Texturas

Se crea "nTexturas" según el número de k-means almacenándolos en una array de texturas y enviándolos después de hallar los k-means correspondientes a un array de imágenes que servirán para nuestra salida. Luego se obtiene la referencia de los campos en los shaders para crear las coordenadas UV y dibujarlas en la imagen de salida.

```
varying highp vec2 vTextureCoord;
//usamos varias texturas
uniform sampler2D uSampler0;
uniform sampler2D uSampler1;
uniform sampler2D uSampler2;
uniform sampler2D uSampler3;
uniform sampler2D uSampler4;
uniform sampler2D uSampler5;
uniform sampler2D uSampler6;
uniform sampler2D uSampler7;
uniform sampler2D uSampler8;
void main(void) {
    gl_FragColor = texture2D(uSampler0, vTextureCoord) +
                    texture2D(uSampler1, vTextureCoord) +
                    texture2D(uSampler2, vTextureCoord) +
                    texture2D(uSampler3, vTextureCoord) +
                    texture2D(uSampler4, vTextureCoord) +
                    texture2D(uSampler5, vTextureCoord) +
                    texture2D(uSampler6, vTextureCoord) +
                    texture2D(uSampler7, vTextureCoord) +
                    texture2D(uSampler8, vTextureCoord);
}
```

Figure 8: Fragment Shader usado por el WebGL

4 Limitaciones

4.1 Inicialización

La función para cargar deterministicamente los valores iniciales del k-means no genera los valores esperados a pesar de seguir el algoritmo mostrado. Esto se debe a que el primer k-mean es escogido adecuadamente pero al momento de atenuar los valores restantes. Este hace que el color contrario adquiera prioridad y al ser este escogido la atenuación regresara de nuevo al color principal o uno de sus valores continuos.

La solución a este problema reside en replantear la atenuación de los colores para asegurarnos que no regrese la prioridad al color inicial.

4.2 Función de transferencia completa

El desplazamiento de un color inicial a otro nuevo es correcto. Pero la interacción de varios colores a un solo grupo del k-means esta ausente. Esto se debe a que la imagen esta divida en varias texturas que son mandadas al webgl.

La solución es formular la función de transferencia $f(x)$ el cual debe trabajar con varias imágenes a la vez sin hacerles perder su fraccionamiento, que todos sumen la imagen completa).

5 Resultados

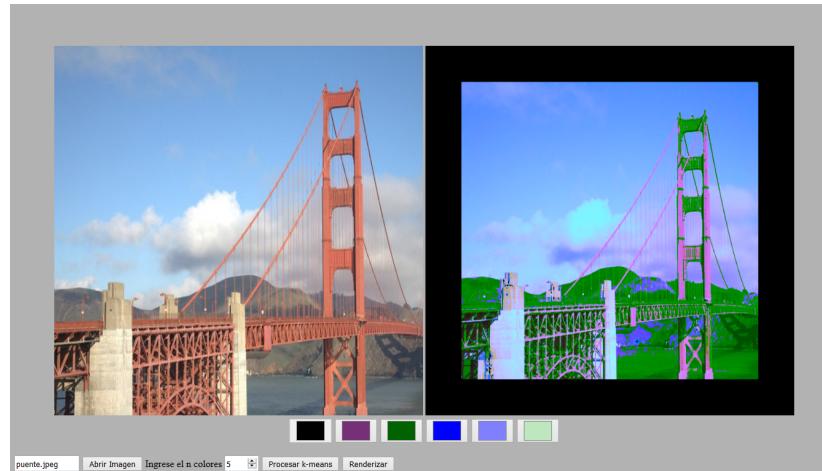


Figure 9: Mantiene la luminosidad



Figure 10: Transferencia de Color 1



Figure 11: Transfrenicia de Color 2

5.1 Comparación con los resultados del Artículo



Figure 12: Transferencia de Color 1

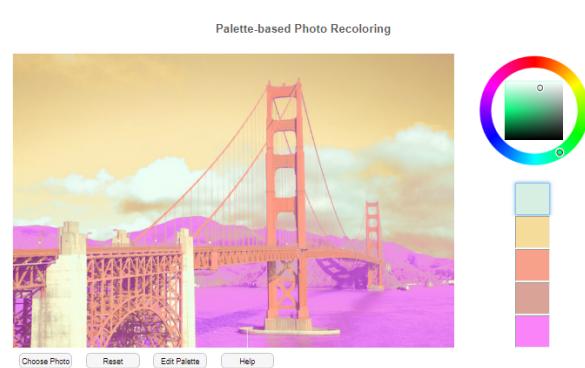


Figure 13: Transfrenicia de Color 2



Figure 14: Transferencia de Color 1

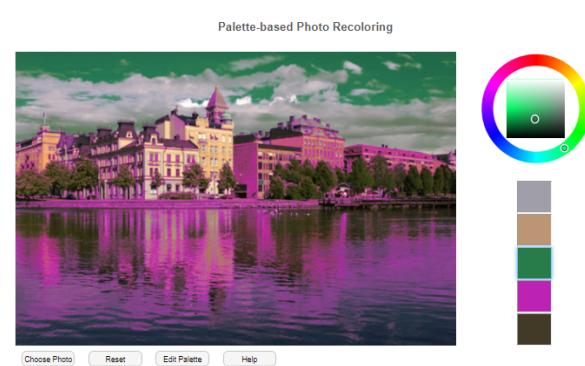


Figure 15: Transfrenicia de Color 2

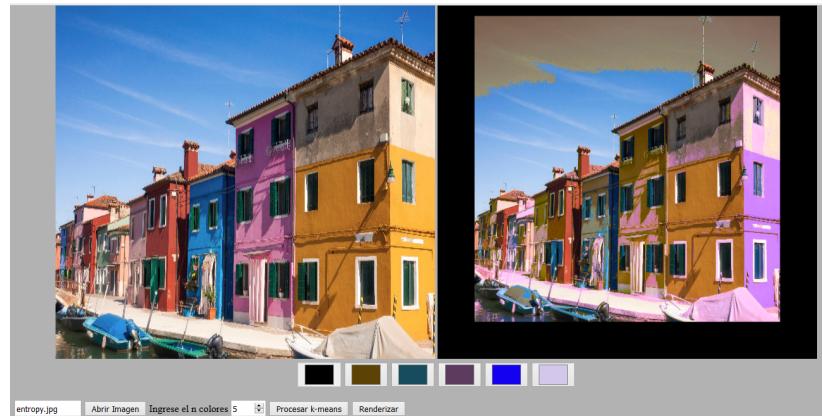


Figure 16: Mantiene la luminosidad



Figure 17: Mantiene la luminosidad

6 Conclusiones

- Un aspecto importante es el trabajo en el espacio LAB. el cual es mas amigable a la vista humana. Pero a la vez implica ciertas reglas. Ya que un color de RGB a LAB y luego a RGB no implica que sea el mismo numero. Si no uno similar y diferente en décimas. Esto puede acarrear problemas si se hace muchas conversiones.
- El espacio en el que trabaja WebGL es en RGBA. Por el cual no se puede

descartar completamente estos valores. Si nos hacer las conversiones necesarias mínimas.

- Se debe tener especial cuidado en las implicaciones matemáticas que conlleva hacer varias operaciones especialmente en las de transferencia.
- A pesar de las limitaciones, se obtienen resultados razonables para los objetivos propuestos.