PARCIAL 2

Informe del desarrollo del proyecto de Informa2
S.A.S.
Implementación.

Juan Fernando Muñoz López.
Carlos Daniel Lora Larios.

Despartamento de Ingeniería Electrónica y
Telecomunicaciones
Universidad de Antioquia
Medellín
Marzo de 2021

Índice

1. DESARROLLO DE LA IMPLEMENTACIÓN.

1.1. Solución del problema:

Una vez desarrollados todas los análisis y propuestas a la solución del problema, nos podemos dar la enhorabuena, pues una vez concluida la implementación hemos determinado que el mapa que habíamos construido, nos llevó al punto al que queríamos llegar, dar una propuesta de solución acertada que cumpla con todos los parámetros requeridos por Informa2 S.A.S, donde buscamos simpleza y eficiencia en el proceso de ejecución del servicio de muestreo de imágenes. En el proceso de implementación le hemos estado dando una estructura sólida a aquel algoritmo propuesto en el informe de análisis, pues para cada uno de los puntos hemos creado una lógica particular para dar solución a cada uno de los subproblemas derivados del problema general: Transformar una imagen representada digitalmente, para ser representada en una matriz de leds RGB. Para ello partimos de lo esencial, seguir con las tareas que habíamos definido, la primera de ellas era la de interactuar con librería QImage, de donde empezamos a desarrollar nuestra implementación general en el entorno de desarrollo Qt, a través del lenguaje de programación de C++. Decidimos crear una clase madre, a la que denominamos FlagImage, la cual sería la encargada de dirigir la orquesta en el proceso de ejecución; para decidir que procesos hacer y como hacerlos, haciendo uso de los métodos y atributos que definimos, para esta solo será necesario entregarle la ruta de la imagen, y ella hará todo el proceso con ayuda de las clases auxiliares PixelImage, countColors, RWFiles para de esta manera exportar la sección de código que será llevada a la implementación de Tinkercad. Donde creamos la matriz leds RGB de 16*16 a partir de módulos de NeoPixels, que será la encargada de mostrar la representación de la imagen.

2. Clases implementadas.

2.1. FlagImage:

Clase que primordialmente analizará cuales son las dimensiones de la imagen con ayuda de la librería QImage, para posteriormente tomar la decisión de que método de transformación usar para que la imagen analizada sea representada como imagen de 16*16 pixeles. Los métodos de transformación que FlagImage usará son:

Submuestreo puro: Para el caso en el que el ancho y alto de la imagen superan los 16 pixeles. Para ello se divide el área total de la imagen en una matriz de 16*16 pixeles, una vez hecho esto hicimos el análisis de alejarnos de la pantalla para visualizar la imagen y concluimos que cada una de estas subáreas serían la representación del led de la matriz de Leds de 16*16, por lo que el color que más predominaba en el área sería el color de representación de esta subárea, y será el color que emitirá el led de nuestra matriz de Tinkercad; esta será la lógica base de nuestra implementación, llevar todo a una matriz de 16*16, a

partir de esto surgió la necesidad de que esta área fuera una clase, PixelIamge que se encargaría de tomar cada porción de área, analizarla y decidir cual es color de representación de su área.

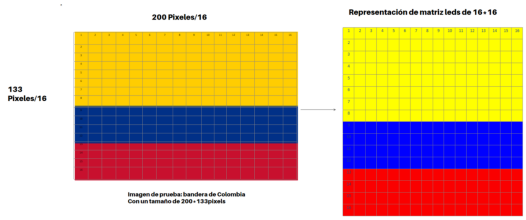

Figura 1: Ejemplo de submuestreo.

-Sobre muestreo:

Para el caso en el que el ancho y alto fueran inferiores o iguales a 16 pixeles. Para ello hicimos el mismo paralelo de división del área total en una matriz de 16*16 solo que no sería necesario implementar la clase PixelImage, puesto que las áreas son inferiores a un pixel, por eso decidimos que cada subárea en el interior del pixel sería la representación de la matriz de 16*16 manteniendo la lógica base, es por eso que de un solo pixel podrían salir varias subáreas y estas tener el mismo color, para de esta manera hacer el proceso de transformación. En el caso de que la imagen tenga el tamaño de 16*16 sería el caso ideal en el que no habría apenas redimensión.

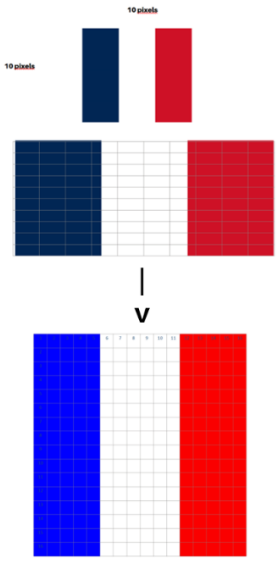

Figura 2: Ejemplo de sobremuestreo.

Casos Combinados:

-Cuando el ancho es superior pero el alto no a 16 pixeles:
Este es un caso particular en el que, si se hace uso de la clase pixelImage, en el que se le asignan áreas de n*1 pixeles, puesto que división equitativa de la imagen tendrá una forma particular, en donde las áreas tendrán un ancho muy grande y apenas una unidad de pixel de alto, para esto se crearon diferentes tipos de PixelImage, para que analicen las áreas según los tipos.

-Cuando el alto es superior pero el alto no a 16 pixeles:
Este es un caso particular en el que, si se hace uso de la clase pixelImage, en el que se le asignan áreas de n*1 pixeles, puesto que división equitativa de la imagen tendrá una forma particular, en donde las áreas tendrán un alto muy grande pero un ancho de una unidad de pixel, este es el último de los tipos de PixelImage. Para cada uno de los casos existen métodos que realizan según el caso la división de área respectiva al tamaño de la imagen con respecto a la dimensión base de 16*16 y decidir que tipo de PixeImage usar. Una vez FlagImage sepa que valor de representación RGB le corresponde a cada led de la matriz de 16*16 R, G, B, ejecutará un método que crea un formato en una cadena de caracteres, de la siguiente manera:

```
{{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 255, 205, 0}, { 255, 205, 0}, { 255, 205, 0}, { 255,
{ 0, 48, 135}, { 0, 48, 135}, { 0, 48, 135}, { 0, 48,
{ 0, 48, 135}, { 0, 48, 135}, { 0, 48, 135}, { 0, 48,
{ 0, 48, 135}, { 0, 48, 135}, { 0, 48, 135}, { 0, 48,
{ 0, 48, 135}, { 0, 48, 135}, { 0, 48, 135}, { 0, 48,
{ 200, 16, 46}, { 200, 16, 46}, { 200, 16, 46}, { 200,
{ 200, 16, 46}, { 200, 16, 46}, { 200, 16, 46}, { 200,
```

Figura 3: Representación del formato.

Información que posteriormente será escrita en archivo .txt que será usado para copiar esta información, y ser llevada a la plataforma Tinkercad.

2.2. PixelImage:

Se le determinarán los límites de su área de análisis, para que posteriormente esta acceda a los valores R, G, B de cada pixel del área y sepa cuál es el color que más predomina en el área. A partir de esta lógica surgió la necesidad de crear una clase, countColors, que, la cual sería útil en el momento de determinar cuál es color que predomina en la zona. A una instancia de la clase countColors , se le asigna la representación R, G, B de un color, y a través de un método PixelImage le asigne el número de veces que se repite, para así determinar el color de representación del área.

Tipos de PixelImage:

Como hay diferentes procesos de división del área total de la imagen por parte FlagImage, PixelImage, también realiza un proceso particular de análisis de su área para los siguientes casos:

-Tipo 1: Áreas superiores a 16*16 pixeles:

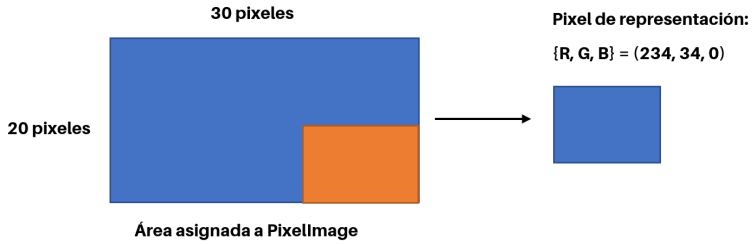

Figura 4: Áreas superiores a 16*16 pixeles.

Tipo 2: Mayor ancho menor alto a 16 pixeles:

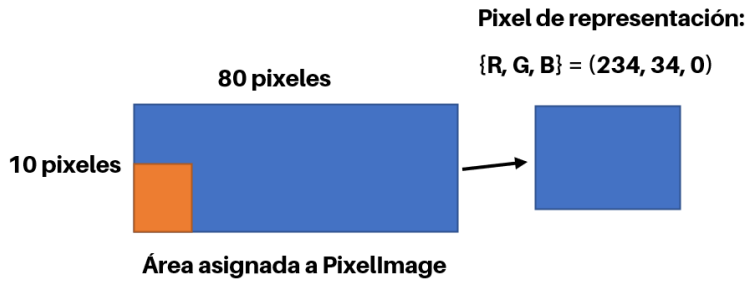

Figura 5: Mayor ancho menor alto a 16 pixeles.

Tipo 3: Mayor alto menor ancho a 16 pixeles:

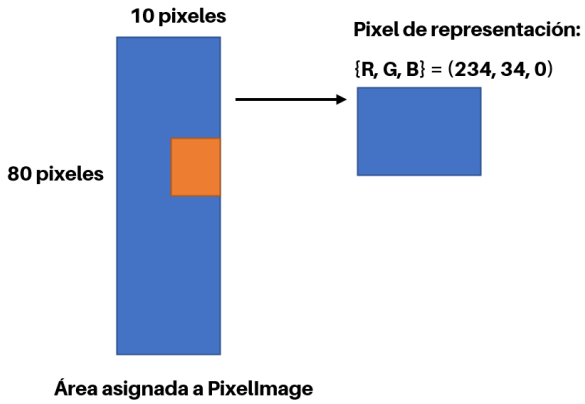

Figura 6: Mayor alto menor ancho a 16 pixeles.

2.3. countColors:

Clase auxiliar que cumple el objetivo ya descrito, determinar el número de veces que se repite un color, a la hora de hacer el análisis en la clase PixelImage de cuál es el color de representación.

2.4. RWfiles:

Es la clase que tiene el método necesario para escribir el archivo .txt, dándole la información a escribir y el nombre que deseamos poner al archivo .txt.

3. Esquema de la estructura de las clases implementadas.

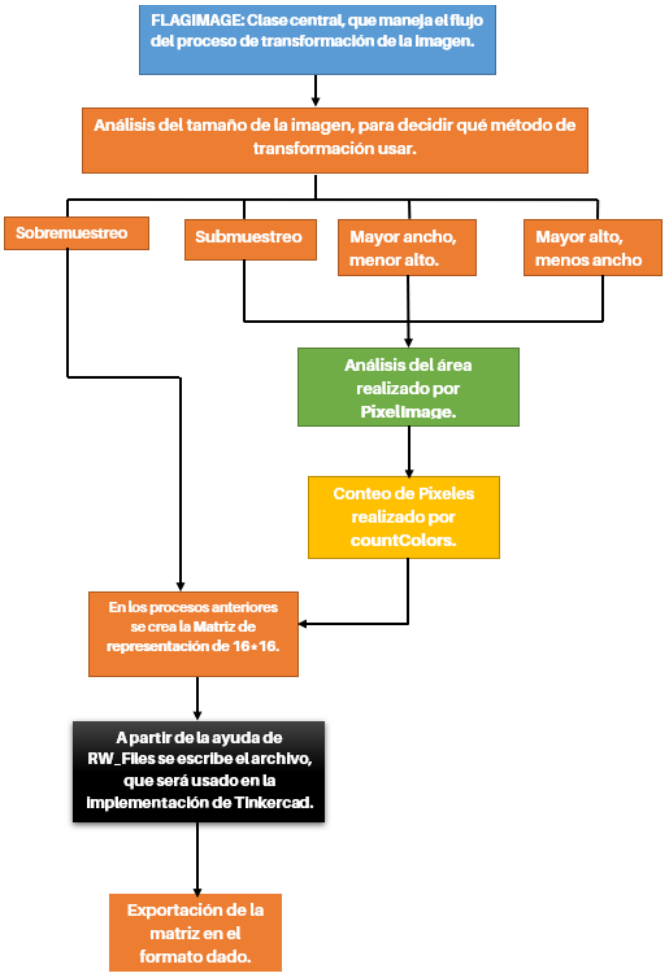

Figura 7: Esquema de clases.

4. Módulos de código de interacción entre clases.

Las clases que hemos implementado, representan cada uno de los pilares necesarios para que el servicio se brinde de manera eficiente; en el proceso de ejecución FlagImage es el encargado de determinar el flujo de trabajo, por ello es el encargado en una primera instancia de tomar la dirección necesario para cada y hacer de instancias tanto de la clase pixelImage y para la exportación de información por medio de RWFiles. PixelImage sabiendo del tipo que es, siempre realiza el mismo proceso en el que hace uso de la clase countColors, para determinar su color de representación.

4.1. Interacción entre la clase flagImage y PixelImage

La función que se presenta a continuación es uno de los métodos de FlagImage en el que le asigna las areas a analizar a las instancias de FlagImage, posteriormente estas realizan su función y le dan a FlagImage el valor de su color de representación y este las almacena en un vector.

```
void FlagImage::CreatePixeles()
{
    /*CreatePixeles: (submuestreo puro) Se divide el total del area de la image
     * en una matriz de 16*16, donde cada porsion de area se le asigna
     * a una instancia de la clase PixelImage, para que determine cual
     * es el pixel de representacion del area y sea almacenado en un vector
     * de la clase FlagImage
     *
     */
   int Width_=getWidth()/16;

   int Height_=getHeight()/16;

   int FaulPixelesWidth=getWidth()%16;

   int FaulPixelesHeight=getHeight()%16;

   if(FaulPixelesHeight!=0){

     FaulPixelesHeight+=-1;
   }
   if(FaulPixelesWidth!=0){

       FaulPixelesWidth+=-1;
   }
```

```
int LastValueW=0;

int LastValueH=0;

int contX=0;

int contY=0;

int co=0;
int co1=0;

for(int i=Height_; i<=getHeight() && co!=16; i+= Height_){

    for(int j=Width_; j<=getWidth() && co1!=16; j+=Width_){

        if(j!=getWidth()-FaulPixelesWidth-1 && i!=getHeight()-FaulPixelesHeigh

        PixelImage element(LastValueH, LastValueW, i, j,*Image, MatrizPixels,
        MatrizLeds[contX][contY]=element;

        contX++;

        }else if(j==getWidth()-FaulPixelesWidth-1 && i!=getHeight()-FaulPixele

            PixelImage element(LastValueH, LastValueW, i, j+FaulPixelesWidth,*
            MatrizLeds[contX][contY]=element;

            contX++;


        }
        else if(j!=getWidth()-FaulPixelesWidth-1 && i==getHeight()-FaulPixeles

                    PixelImage element(LastValueH, LastValueW,i+FaulPixelesH
                    MatrizLeds[contX][contY]=element;

                    contX++;


         }else if(j==getWidth()-FaulPixelesWidth-1 && i==getHeight()-FaulPixel

           PixelImage element(LastValueH, LastValueW, i+FaulPixelesHeight, j+F
           MatrizLeds[contX][contY]=element;

            contX++;
```

```
                }

                LastValueW=j ;
                co1++;

            }

            co++;
            co1=0;
            contX=0;
            contY++;
            LastValueH=i ;

            LastValueW=0;

        }



}
```

4.2. Interacción entre la clase PixelImage y countColors

La función que se presenta a continuación es uno de los métodos de PixelImage, en el que una vez sabe cuales son son todos los colores de su área asignada, se dispone a saber cuál es color que más predomina en el área y por lo tanto este sea su color de representación.

```
void PixelImage::getMyColor()
{
    /*getMyColor: Funcion que determina cual es color de represenatcion del area
     * utilizando instancias de la clase countColors, par saber cuantas veces se
     * en el area.
     *
     */

    vector <int> aux;

    bool flag=true;

    vector <countColors> *ColorsArea=new vector <countColors>;

    for(auto valor : *MyAreaOfPixeles){

        if(!ColorsArea->empty()){

        for(auto value: *ColorsArea){

        if(comparateVector(value.getMyColor(), valor)){

                value.setMyNumOfColors(value.getMyNumOfColors()+1);

                flag=false;
        }


        }

        if(flag){

            countColors element(valor);
            ColorsArea->push_back(element);


        }

        }
```

```
        else{

          countColors element(valor);
          ColorsArea->push_back(element);

        }


    }


    countColors MyUniqueColor;

    bool flag2=true;

    for(auto value : *ColorsArea){

        if(flag2){

            MyUniqueColor=value;

            flag=false;

        }
        else{

            if(value.getMyNumOfColors()>MyUniqueColor.getMyNumOfColors()){

                MyUniqueColor=value;

            }
        }

    }

    delete ColorsArea;

    MyColor=MyUniqueColor.getMyColor();

    MatrizPixelsCopy->push_back(MyColor);

}


}
```

4.3. Interacción entre la clase FlagImage y RWFiles

La función que se presenta a continuación es uno de los métodos de FlagImage, el cual crea el formato que será escrito por la clase RWfiles, en donde una vez se crea el formato, es escrito en un archivo .txt.

```
void FlagImage::genTheTxtPixels()
{
    /*genTheTxtPixels: Utiliza el vector MatrizPixels para crear un formato
     * de un matriz de enteros de [256][3], a cadenas de strings correspondintes
     * a la representacion de la imagen transformada a 16*16 pixeles, para poste
     * escribir esta informacion en el archivo que se guardara en la carpeta bui
     * proyecto con el nombre de MyFile.txt, informacion que sera llevada a Tink
     *
     */
    string MatrizLeds="";

    int contX=0;

    int contY=0;

    for(auto value: *MatrizPixels){

        if(contX==0){

            MatrizLeds+="{ "+to_string(value[0])+" ,"+to_string(value[1])+" ,"+t

              contY++;

        }else if(contX==255){

             MatrizLeds+="{ "+to_string(value[0])+" ,"+to_string(value[1])+" ,"+t

            contY++;



        }else{


             MatrizLeds+="{ "+to_string(value[0])+" ,"+to_string(value[1])+" ,"+t

             contY++;
```

```
            }
            contX++;
            if(contY==16){
            MatrizLeds+="\n";
            contY=0;
            }
        }
        delete MatrizPixels;
        cout<<endl<<"__________Exportacion_de_transformacion_de_la_imagen_completada
        cout<<"==========================================================================
        //////////ACA PUEDE CAMBIAR EL NOMBRE Y LA RUTA//////////
                    escribir("MyFile.txt", MatrizLeds);


}
```

5. Estructura del circuito.

La estructura general como ya hemos mencionado es una matriz de leds RGB de 16*16, creada a partir de módulos de tiras de led de Neopixels de 8 Leds. Distribuidos de la siguiente forma:

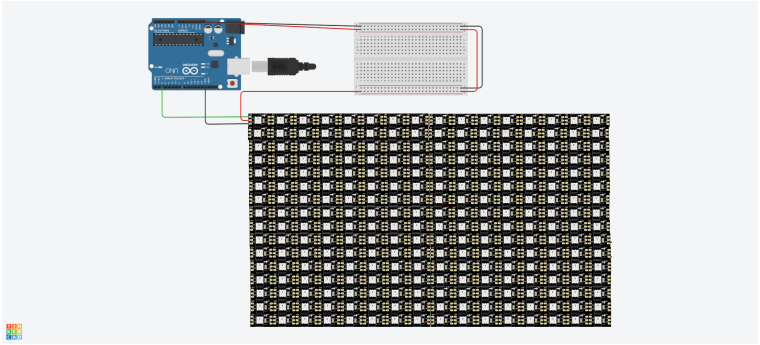

Figura 8: Matriz de 16*16.

En total se hacen uso de 32 módulos de 8 leds de tiras de NeoPixels, dispuestos de manera vertical, por lo tanto, en cada fila hay dos de estos módulos. El primero de estos módulos (el que se encuentra en la esquina superior izquierda) se le conectan en sus entradas, para uno de estas, se conecta con un puerto digital del módulo de Arduino que es por el cual se hará la comunicación y envío de datos, se conecta la corriente y el cable de tierra. Posteriormente estas conexiones son llevadas desde la salida del modulo hasta las entradas del otro, a través de la superposición, posteriormente de este segundo módulo de la primera fila su salida información es lleva a través de cables al primer módulo de la segunda, y así progresivamente. Estos cables son se encuentran por debajo de los módulos de NeoPixels. Una vez si tiene esta conexión lineal entre módulos, con ayuda de la librería AdafruitNeoPixel, es como lográbamos definir a todos estos módulos como uno solo, y de esta manera asignar cada valor de la matriz creada en Qt a su respectivo led en la matriz de Tinkercad.

6. Problemas presentados durante la implementación.

Ha sido un trabajo muy enriquecedor, pues nos ha hecho sacar lo mejor de nosotros para superar cada reto que se nos presentaba en el desarrollo. El problema más significativo que se nos presentó fue con el montaje de la matriz en Tinkercad, pues con las recomendaciones dadas por el profe, con el uso de la fuente poder estábamos teniendo ciertas dificultades, pues los datos directamente no mostraban nada o eran representados de maneras muy extrañas, para ello simplemente quitamos la potencias, reorganizamos los módulos y la matriz a la que se le dan los datos en el código implementado le dimos un valor extra, para que el almacenamiento de la información de exportada desde Qt, a la hora de ser leído no tuviera problema.