Informática II

Solución del Parcial 2

Reinaldo Marín Nieto

Despartamento de Ingeniería Electrónica y
Telecomunicaciones
Universidad de Antioquia
Septiembre de 2021

Índice

1. Presentación del problema **2**

2. Solución implementada **2**
2.1. Clases implementadas 2
2.2. Módulos de código implementados 3
2.3. Estructura final del circuito 7
2.4. Problemas presentados durante la implementación. 8

1. Presentación del problema

La tarea consiste en recibir una imagen de una bandera en dos dimensiones y en formato jpg y realizar el procesamiento de la información contenida en esta, de tal forma que se haga un ajuste de sus dimensiones, para que pueda ser presentada en una matriz de LEDs RGB.

2. Solución implementada

El método que fue usado para la solución consta de recibir la imagen ingresada por el usuario, y según su tamaño dividirla en fragmentos, simulando una cuadrícula de 7x6. Recorre cada fragmento y toma el color que más se repite entre los pixeles que habitan cada pedazo de imagen. Luego toma ese color y lo imprime dentro del archivo de texto, a manera de órden para la plataforma Arduino; texto que al final será copiado en la consola del simulador donde se encuentra el circuito.

2.1. Clases implementadas

Cuadrito: La primera clase que usé fue llamada como çuadrito". Su función es abarcar un pixel que se encuentre dentro de un fragmento (Con sus correspondientes valores de RGB) de manera cíclica. Dando un total de 12000 çuadritos"dentro de cada fragmento.

```
#ifndef CUADRITO_H
#define CUADRITO_H
#include <string>
#include <vector>
#include <string>
using namespace  std;
class Cuadrito
{
   //atributos
   int pos;
   int red;
   int green;
   int blue;

    //Metodos
public:
    Cuadrito();
    Cuadrito(int pos);
    Cuadrito(int pos,int red, int green, int blue);
    int getpos() const;
    int getred() const;
    int getgreen() const;
    int getblue () const;


};
#endif // CuadritoES_H
```

Figura 1: Fragmento de código de la estructura de la clase Çuadrito"

Inventario: La segunda y última clase que se usa dentro del programa se llama ïnventario", y se encuentra en el header "listacuadritos.h". Su función principal es contener una lista de la clase çuadrito", almacenando todos los pixeles de un fragmento y facilitando su indexación, pero reiniciándose en cada nuevo fragmento a recorrer, para ahorrar memoria y facilitar la indexación cíclica.

```
#ifndef Inventario_H
#define Inventario_H
#include <cuadrito.h>
#include <vector>
#include <fstream>

class Inventario
{
    vector<Cuadrito*>Cuadritos;

public:
    Inventario();
    bool AgregarCuadrito(Cuadrito *Cuadrito);
    Cuadrito *BuscarCuadrito(int pos);
    int ContarCuadritoes();
    vector<Cuadrito*>getCuadritos();

};

#endif // Inventario_H
```

Figura 2: Fragmento de código de la estructura de la clase Ïnventario"

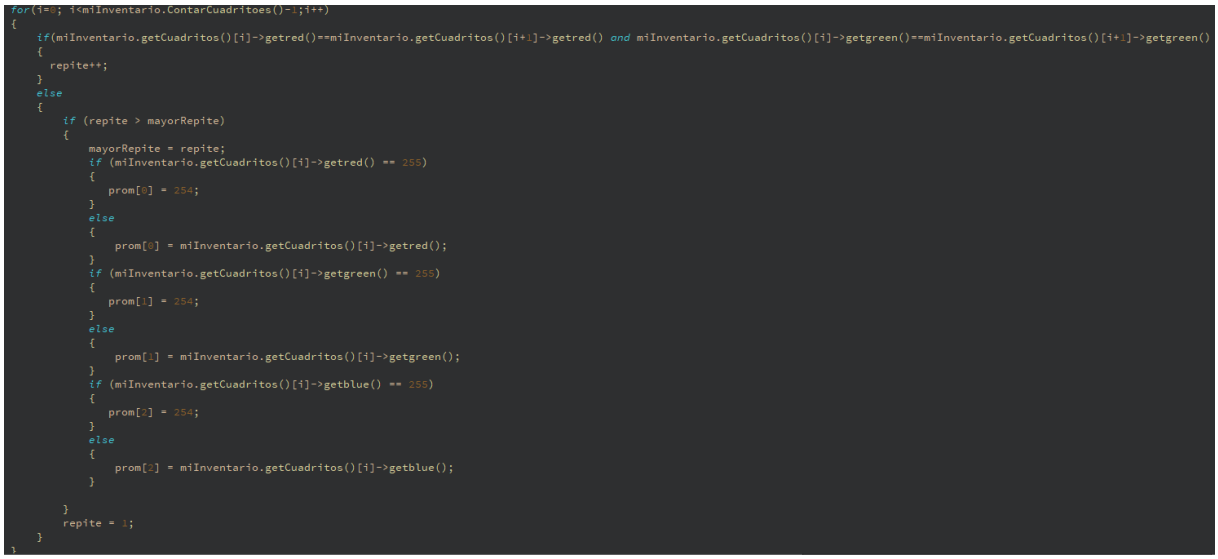
2.2. Módulos de código implementados

El programa se diseñó con una función bastante robusta y completa, llamada "Llenar". Ésta se encarga de convertir los píxeles muestreados, según sus coordenadas (las cuáles son sus valores de entrada o parámetros) en clases Çuadrito", y de agregarlas a la clase Ïnventario".

```
vector<int> llenar (int inix, int iniy, int endx, int endy)
{
    string filename = "imagen.jpg";
    QImage im(filename.c_str() );
    Cuadrito *cuadrito;
    Inventario miInventario;
    vector<int> prom = {0,0,0};
    int contador = 0;
    int aumento = iniy;
    for(inix;inix<endx;inix++)
    {
        for(iniy;iniy<endy;iniy++)
        {
            int red = im.pixelColor(inix,iniy).red();
            int green = im.pixelColor(inix,iniy).green();
            int blue = im.pixelColor(inix,iniy).blue();
            if(im.pixelColor(inix,iniy).red() == 255)
            {
                red = 254;
            }
            if(im.pixelColor(inix,iniy).green() == 255)
            {
                green = 254;
            }
            if(im.pixelColor(inix,iniy).blue() == 255)
            {
                blue = 254;
            }
            cuadrito = new Cuadrito(contador,red,green,blue);
            if(miInventario.AgregarCuadrito(cuadrito))
            {
                cout<<"";
            }
            else
            {
                cout<<"Error"<<endl;
            }
            contador++;
        }
```

Figura 3: Fragmento de código de la estructura de la primera parte de la función

La segunda parte de la función se encarga de revisar todos los çuadritos.ªlmacenados en ïnventario", comparar sus valores RGB y encontrar el valor RGB que más se repita, para así obtener el valor de retorno, un vector de enteros que contiene los tres valores (rojo, verde y azul) y entregarlo a la función main.



```
for(i=0; i<miInventario.ContarCuadritoes()-1;i++)
{
    if(miInventario.getCuadritos()[i]->getred()==miInventario.getCuadritos()[i+1]->getred() and miInventario.getCuadritos()[i]->getgreen()==miInventario.getCuadritos()[i+1]->getgreen()
    {
      repite++;
    }
    else
    {
        if (repite > mayorRepite)
        {
            mayorRepite = repite;
            if (miInventario.getCuadritos()[i]->getred() == 255)
            {
               prom[0] = 254;
            }
            else
            {
                prom[0] = miInventario.getCuadritos()[i]->getred();
            }
            if (miInventario.getCuadritos()[i]->getgreen() == 255)
            {
               prom[1] = 254;
            }
            else
            {
                prom[1] = miInventario.getCuadritos()[i]->getgreen();
            }
            if (miInventario.getCuadritos()[i]->getblue() == 255)
            {
               prom[2] = 254;
            }
            else
            {
                prom[2] = miInventario.getCuadritos()[i]->getblue();
            }

        }
        repite = 1;
    }
}
```

Figura 4: Fragmento de código de la estructura de la segunda parte parte de la función

Finalmente, la función "main"se encarga de recibir la imagen, obtener el tamaño de la misma y dividir éstos valores según su eje. La altura se divide por 6, y el ancho se divide por 7, para obtener la distancia entre los fragmentos en los que se dividirá la imagen y crear un ciclo donde ese valor se sumará a sí mismo para recorrer la imagen.
Las condiciones del ciclo principal se basan en la cantidad de grupos en los que se dividirá la imagen de manera vertical, es decir, en 6 grupos. Además de ésto, contiene un ciclo interno que se encarga de la sección horizontal de la imagen, pero esta vez, sus condiciones de rompimiento dependen del número de LEDs que tendrá cada tira NeoPixel dentro del circuito. Es decir, 15.
Los ciclos se encargaran de invocar a la función llenar con cada fragmento de la imagen que vaya obteniendo según la operación de aumento de distancia. Luego se encargarán de escribir en el fichero las órdenes correspondientes a ingresar en la consola de TinkerCAD. Cabe aclarar que los índices horizontales correspondientes a los LEDs que deben encender según cada tira NeoPixel comienza desde el índice 1 debido a la estructura del circuito, explicada en la próxima sección
Además de ésto, la función "main"también incluye el ingreso de datos al fichero .ºrdenes.txt"de las órdenes básicas de inicialización, que no tienen que ver con la función de muestreo directamente.

```
archivo<<"void setup()"<<endl;
archivo<<"{"<<endl;
for(int i = 0; i<=5;i++)
{
   archivo<<"  leds"<<i<<".begin();"<<endl;
   for (int z = 1; z<=13;z+=2)
   {
     archivo<<"  for (int i ="<<z<<"; i < "<<z+2<<"; i++)"<<endl;
     archivo<<"  {"<<endl;
     prom = llenar (inix,iniy,endx,endy);
     archivo<<"   leds"<<i<<".setPixelColor(i,"<<prom[0]<<","<<prom[1]<<","<<prom[2]<<");"<<endl;
     archivo<<"  }"<<endl;
     archivo<<"  leds"<<i<<".show();"<<endl;
     if(endx<w)
     {
       inix += w/7;
       endx += w/7;
     }
     else
     {
         break;
     }

   }
   inix = 0;
   endx = w/7;
   if(endy<=h and iniy <=h-(h/7))
   {
     iniy += h/6;
     endy += h/6;
   }
   else
   {
       break;
   }


}
archivo<<"}"<<endl;
archivo<<"void loop(){}"<<endl;
```

Figura 5: Fragmento de código de la estructura de la función "main"

```
// C++ code
//
#include <Adafruit_NeoPixel.h>
Adafruit_NeoPixel leds0 (15,2,NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel leds1 (15,3,NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel leds2 (15,4,NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel leds3 (15,5,NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel leds4 (15,6,NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel leds5 (15,7,NEO_GRB + NEO_KHZ800);
void setup()
{
  leds0.begin();
  for (int i =1; i < 3; i++)
  {
   leds0.setPixelColor(i,254,205,0);
  }
  leds0.show();
  for (int i =3; i < 5; i++)
  {
   leds0.setPixelColor(i,254,205,0);
  }
  leds0.show();
  for (int i =5; i < 7; i++)
  {
   leds0.setPixelColor(i,254,205,0);
  }
  leds0.show();
```

Figura 6: Fragmento del fichero de salida a ingresar en el simulador

2.3. Estructura final del circuito

El circuito elaborado para el pleno funcionamiento del programa consta de un Arduino, que se encarga de procesar los comandos y llevarlos a las salidas digitales. Cada salida se conecta a la entrada de cada uno de los 6 NeoPixel que sirven como controladores de las dos tiras NeoPixel a las que está conectado correspondientemente, por esa razón se dan órdenes de encender los LEDs desde el índice 1, y no del 0.
Al tener 6 NeoPixel de manera vertical, y de manera horizontal le corresponden dos tiras de 14 NeoPixels a cada uno, forman como resultado una matriz de 14x12 NeoPixels.
Además de todo ésto, el circuito cuenta con una fuente externa que se usa para el correcto funcionamiento del circuito.

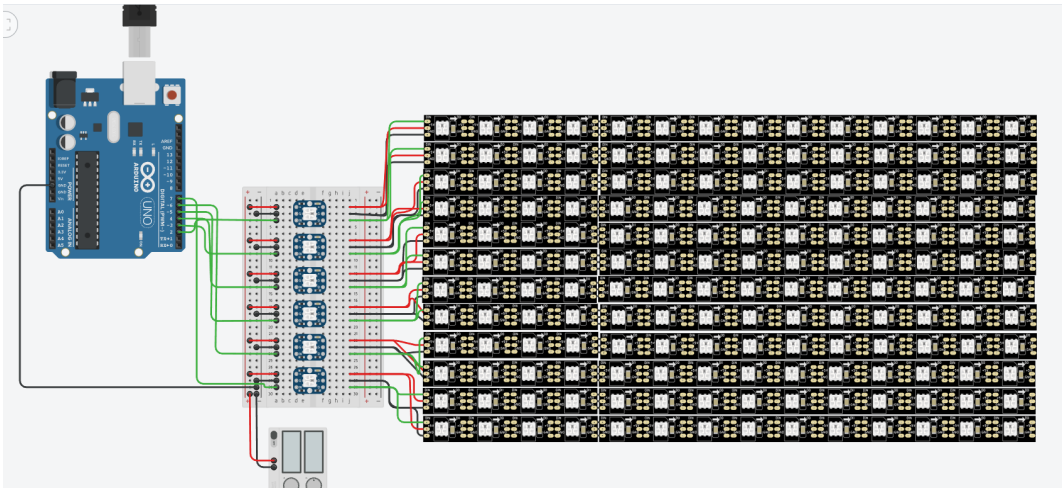


Figura 7: Captura directa del circuito completo desde el simulador TinkerCAD

2.4. Problemas presentados durante la implementación.

Los inconvenientes con la implementación fueron bastantes. Desde errores básicos de sintaxis hasta errores lógicos a la hora de implementar los algoritmos. Sin embargo, se presentaron problemas que pusieron en jaque el destino del proyecto.
El principal problema se basaba en que el programa no cumplía con uno de los requisitos mínimos exigidos. El código estaba diseñado única y exclusivamente para imágenes de 700x600 píxeles. Al final lo pude arreglar mediante operaciones matemáticas para dividir cualquier imagen en los grupos 7x6 que necesitaba para el correcto cumplimiento.
Además de ésto, surgieron errores de indexación, donde los valores que se buscaban dentro de la imagen, superaban el tamaño de la misma y las coordenadas se perdían del rango completamente. Pude arreglarlo por medio de condicionales.
En la etapa final del proyecto, tuve un error grave con el repositorio de GitHub, ya que por muchos commits que hiciera, tanto de manera manual como desde Git GUI, los archivos no se reemplazaban. Tuve que hacer un borrado manual y posteriormente una resubida de la carpeta de proyecto.
Otro error que se presentó, pero que fue mínimo, fue el de la indexación del color blanco en los valores RGB, debido al error del simulador donde los 3 valores RGB no pueden estar en 255, tuve que crear el condicional que los rebajara a 254 para así poder mostrar el color blanco de manera adecuada.