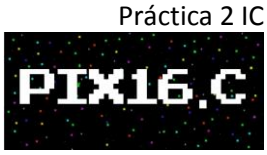
PIX16.C

ALGORITMO PIXELIZADOR UTILIZANDO MAGICK++

INGENIERIA DE LOS COMPUTADORES
VICENTE MARTIN RUEDA
PABLO REQUENA GONZALEZ
MARCOS GONZALEZ VERDU

ÍNDICE



INTRODUCCIÓN AL PROBLEMA

El objetivo de esta práctica era implementar un problema con grandes necesidades computacionales, pensamos que sería adecuado hacer un filtro de imagen que transforme la imagen en píxeles más grandes (como si perdiera resolución) y convierta los colores, según unos rangos, en los colores de la paleta de 16 colores.

El primer problema a resolver era leer y escribir en imágenes. Para ello, trabajamos con una librería de C++ llamada Magick++. La librería permite muchas más cosas para el procesamiento de imágenes, pero para implementar nosotros el algoritmo, únicamente utilizamos la librería para leer y escribir imágenes y para consultar y editar los píxeles.

Para conseguir el efecto pixelizador, calculamos el color medio de un sector (inicializado por parámetro, es decir, el tamaño de los sectores se puede elegir) y lo asignamos a todos los píxeles implicados, de manera que queda todo un sector del mismo color, es decir, un “píxel grande”.

Al realizar la media del color, según el valor que se obtenga, se calcula su equivalente en la paleta de 16 colores, y este es el color que finalmente se asigna.

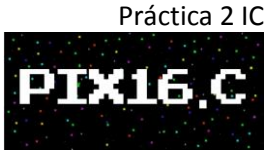
Esta es la manera en que el algoritmo PIX16.C transforma cualquier imagen como si la viéramos en un videojuego antiguo.

INTRODUCCIÓN A MAGICK++

Buscando como trabajar con imagenes en C++, encontramos la libería Magick++, que trabaja y procesa imagenes, además de tener un montón de funciones con ellas. Magick++ es una librería implementada en C++ que utilizan varios proyectos como Gimp, Octave, varios sistemas operativos de la familia GNU Linux, en general es muy habitual.

Encontramos un procesador de imagen orientado a objetos de manejo muy sencillo y una API muy completa. La parte que nos interesa es la clase Image, uno de sus atributos es Pixels, que funciona como un mapa de píxels a los que se accede por fila y columna. Cada uno de estos píxels contiene un PixelPacket que tiene 3 números en porcentaje (de 0 a 1) que representa la cantidad de rojo, verde y azul que se necesitan para obtener ese color RGB. La clase Image permite hacer read directamente de un archivo, y write para guardar la imagen con la que hemos trabajado.

El alcance de la librería va mucho más allá, sobre todo en los STL Algorithms, que ofrece un montón de funciones útiles para el procesamiento de imagen como aplicar un filtro de color, redimensionar, calcular la media del color de un bloque de píxeles, etc. pero estas no nos interesan para este proyecto.



MANEJO DEL PROGRAMA

Como la API y la Documentación de Magick++ (ImageMagick) no está lo suficientemente clara como para empezar a utilizarla desde el inicio sin conocimientos, encontramos un foro donde nos explicaban como instalarlo en Ubuntu (Linux), así como qué parametros hay que utilizar para compilar los códigos fuentes.

$ apt-cache search dev | grep magick // comprueba que el paquete exista

$ sudo apt-get install libmagick++-dev // te instala el paquete

$ sudo apt-get install libgraphicsmagick1-dev // instalas una librería necesaria (dependencias)

Instalé Magick++ con los comandos anteriormente escritos y encontramos el comando para compilar el código.

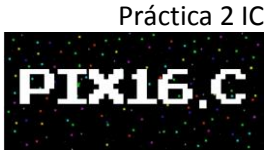
$ g++ `Magick++-config --cxxflags --cppflags` -o EJECUTABLE CODIGO.cpp `Magick++-config --ldflags –libs`

Tras muchas horas de investigación, por fin funcionaba.

Empezamos a leer el API de Magick++ y con los pocos y confusos ejemplos que ponen conseguimos hacer nuestro primer “programa”, el cual consistía en abrir una imagen y guardarla con otro nombre.

Aprendimos que Magick++ cuenta con una clase “Image” que hace el trabajo relacionado con la Imagen, a nosotros de esta clase, solo nos importaba abrir la imagen. El cual te abre la imagen y ya te deja tabajar con ella.

Como nuestro problema a tratar estaba claro desde el principio, y solo nos importaba el tratamiento de colores, lo cual se realizaba trabajando conjuntamente con dos clases “Image” y “ColorRGB” que con una podríamos trabajar con los pixeles (La de Image) y con la otra trabajar con colores.



IMPLEMENTACIÓN Y EXPLICACIÓN DEL CÓDIGO

```
int main(int argc, char ** argv)
{
    long double tiempoTotal_1, tiempoTotal_2, tiempoTotal, tiempo1, tiempo1fin;
    InitializeMagick(*argv);

    if(argc<4)
    {
        cerr << "-------------------------------------------------------\n";
        cerr << "Error: prueba con " << argv[0] << " <Num_Pixels> <Nombre_Entrada> <Nombre_Salida> ... \n";
        cerr << "-------------------------------------------------------\n";
    }
    else if(argc>=4)
    {
        int tamBigPix = atoi(argv[1]);
        tiempoTotal_1=tiempo();
        for(int i=2; i<argc; i+=2)
        {
            tiempo1=tiempo();
            if(argc%2!=0 && i==argc-1)
                cerr << "ERROR: Falta archivo de destino para " << argv[argc-1] << endl;
            else
            {
                try
                {
                    Image imagen(argv[i]);

                    int ancho = imagen.columns(), alto = imagen.rows();
                    for(int j=0; j<ancho; j+=tamBigPix)
                    {
                        for(int k=0; k<alto; k+=tamBigPix)
                            subSector(imagen, j, k, tamBigPix, ancho, alto);
                    }

                    imagen.write(argv[i+1]);
                }
                catch(Exception &error_)
                {
                    cout << "Ha habido algún problema: " << error_.what() << endl;
                }
            }
            tiempo1fin=tiempo();
        }
            tiempoTotal_2=tiempo();
            tiempoTotal = tiempoTotal_2 - tiempoTotal_1;

            return 0;
```

Main:

- Hace la comprobación de número de argumentos, hace la inicialización del “Magick++” para trabajar con las imágenes, guarda el número de pixeles que queremos utilizar para nuestro pixel grande (este valor se le pasa por parámetro).

- Tras esto, tenemos un primer bucle “for”, que trabajará con los pares FicheroEntrada/FicheroSalida comprobando que los argumentos sean pares, y que si hay un par suelto, no procesará. Se abre la imagen, con “Image imagen(nombre_archivo_entrada), llamando al constructor de Image que nos proporciona la librería Magick++, con la que vamos a trabajar, guardamos la anchura y altura.

- Ahora tenemos un par de bucles “for” anidados, que sirven para moverse en la totalidad de la imagen, llamando a subsector, con la imagen, los márgenes y el tamaño de pixeles por pixel grande (tamBigPix, a partir de ahora.)

- Una vez acabados los bucles, escribiremos el archivo con el nombre que nos han indicado, esto se realiza con la línea “imagen.write(nombre_archivo_salida)”.

- El bloque try/catch hace las comprobaciones de la apertura de las imágenes, si alguna imagen no la encuentra, no terminará la ejecución del programa.

```
void subSector(Image &ima, int x, int y, int rango, int ancho, int alto)
{
    double r = 0, g = 0, b = 0;
    ColorRGB auxColor;

    for (int i = x; (i < rango+x) && (i < ancho); i++)
    {
        for (int j = y; (j < rango+y) && (j < alto); j++)
        {
            auxColor = ima.pixelColor(i, j);
            r = r + (auxColor.red()*255);
            g = g + (auxColor.green()*255);
            b = b + (auxColor.blue()*255);
        }
    }
    r = r/(rango*rango);
    g = g/(rango*rango);
    b = b/(rango*rango);

    ColorRGB color(to16Palette(r, g, b));

    for (int i = x; (i < rango+x) && (i < ancho); i++) {
        for (int j = y; (j < rango+y) && (j < alto); j++) {
            ima.pixelColor(i, j, color);
        }
    }
}
```

Subsector:

- Es el modulo encargado de calcular la media de los pixeles grandes, así como dibujar el pixel grande con el mismo color, transformado de una paleta e 256 colores a 16.

- Primero crea las variables “r” (rojo), “g” (verde), “b” (azul), así como una variable ColorRGB auxiliar que nos facilitará las cosas a la hora de calcular la media.

- Tenemos un par de bucles anidados que recorren el sub sector, marcado por el rango, el cual recibimos del main, y este como parámetro.

- Iremos sumando los valores “r”, “g”, y “b” para dividirlos entre el total de pixeles que hay en el sector, para sacar la media.

- Una vez tenemos la media, llamaremos al módulo “to16Pallete” guardaremos lo que nos devuelve este módulo en un ColorRGB.

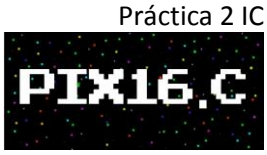
- Una vez tenemos el color con el que queremos rellenar el pixel, con:

“Imagen.pixelColor(coordenadaX, coordY, color)” dibujaremos el sector.

```
ColorRGB to16Palette(double r, double g, double b)
{
  if(0 <= r && r <= 255 && 0 <= g && g <= 255 && 0 <= b && b <= 255)
  {
        if(r <= 66)
        {
              if(g <= 66)
              {
                if(b <= 66)
                {
                    ColorRGB color16b(0, 0, 0); // 0 - Black
                    return color16b;
                }
                else if(b <= 198)
                {
                    ColorRGB color16b(0, 0, 132);   // 1 - RedBrown
                    return color16b;
                }
                else // 199 - 255
                {
                    ColorRGB color16b(0, 0, 255);   // 9 - OrangeRed
                    return color16b;
                }
              }
              else if(g <= 198)
              {
                if(b <= 66)
                {
                    ColorRGB color16b(0, 132, 0);   // 2 - Green
                    return color16b;
                }
                else // 67 - 255
                {
                    ColorRGB color16b(0, 132, 132); // 3 - BrownGreen
                    return color16b;
                }
              }
```

To16Pallette: - Como no encontrábamos ninguna forma de pasar una paleta de 256 colores a 16, hicimos una aproximación. Asignamos unos rangos a cada color, de manera que si la media queda determinada devolvamos el color correspondiente de la paleta de 16 colores. Los rangos van en función de los valores del rojo, el verde y el azul (RGB).

El tamaño de los colores en 16 colores, es 8 bits por cada color, es decir, 24 bits (originalmente eran 8 bits para codificar todo el color), así que cada valor va del 0 al 255, como se puede observar.



PRUEBAS DE EJECUCIÓN Y RENDIMIENTO

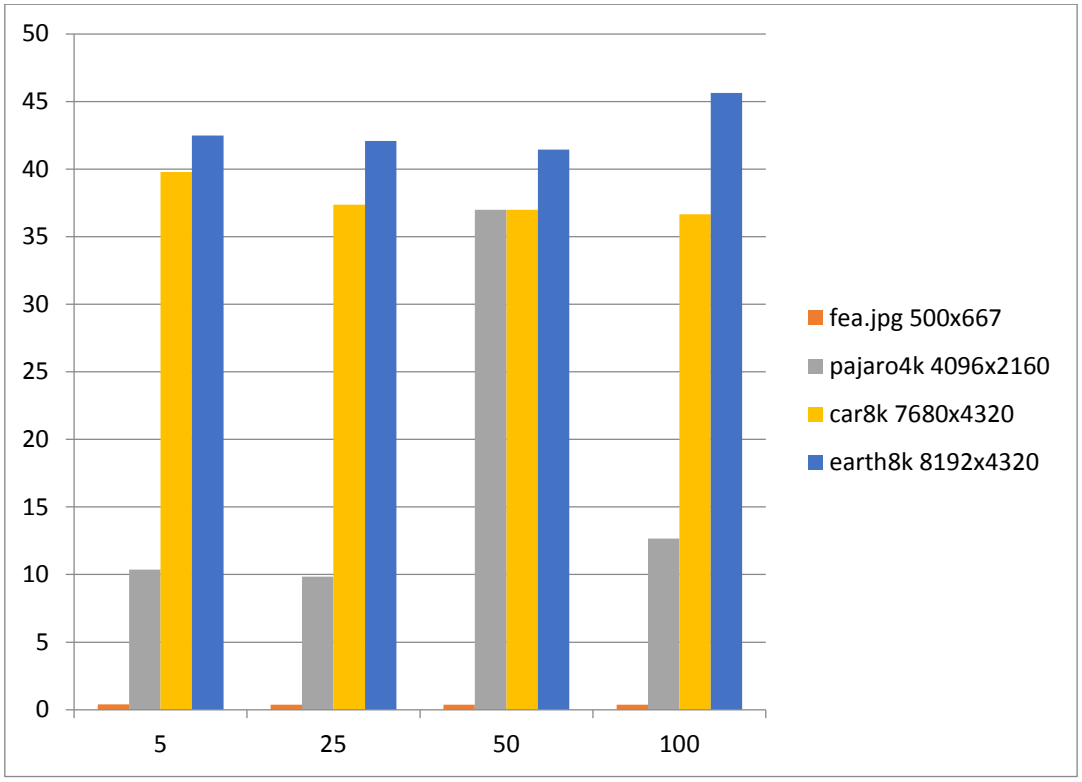
Para las pruebas hemos escogido cuatro imágenes de nuestro repertorio, las hemos ejecutado todas a la vez, con el mismo tamaño de Pixeles pequeños por Pixel Grande.

Las imágenes son de tamaño: “500*667”, “4096*2160”, “7680*4320” y “8192*4320” (De menor a mayor).

Los pixeles por sector (Píxel grande) son: “5”, “25”, “50” y “100”.

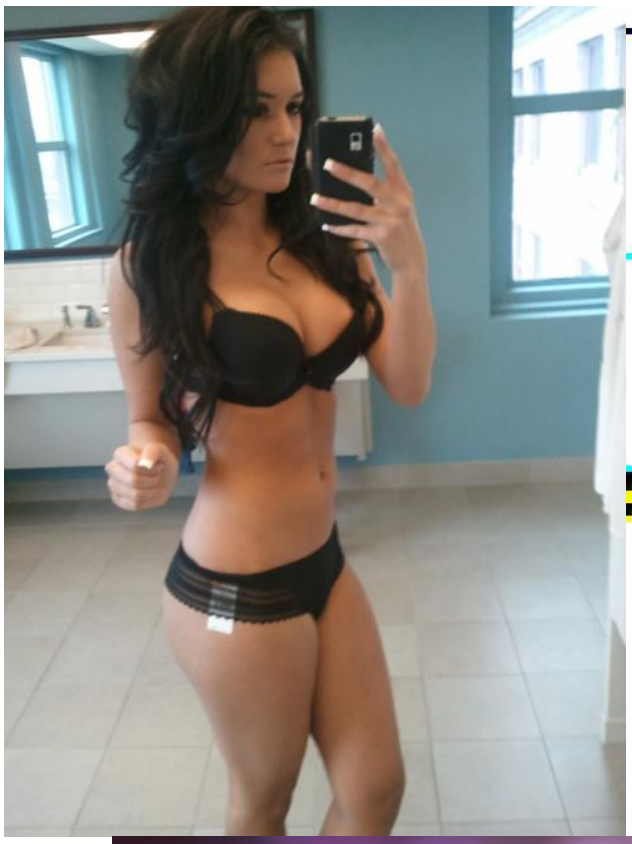
Gracias a la forma que ha sido diseñado el algoritmo, nos permite una gran facilidad para realizar estas pruebas, pudiendo modificar el número de imágenes que se quieren modificar a la vez, y el número de Pixeles que se utilizarán para hacer los subsectores.

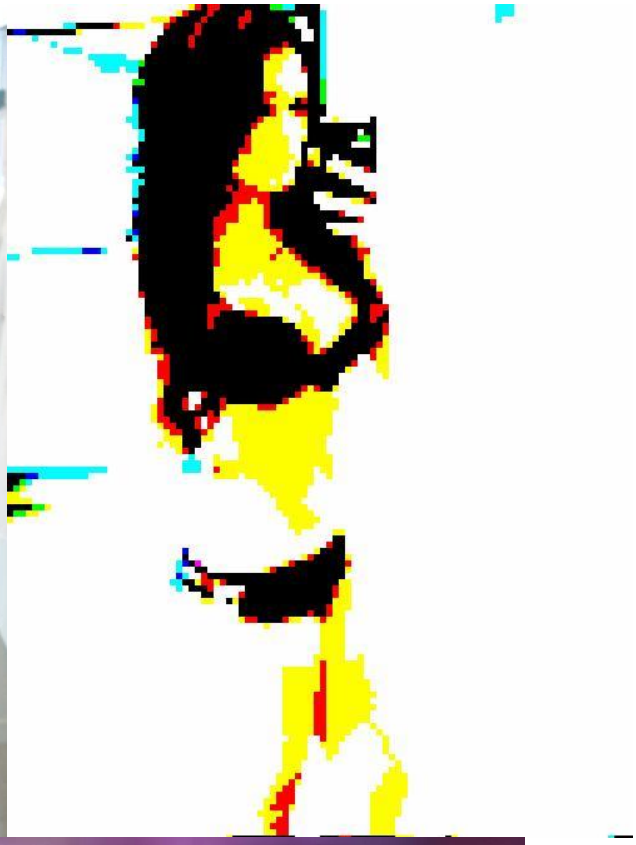
Se han realizado en distintos ordenadores. El resultado son los segundos que el algoritmo ha tardado en transformar las imágenes.

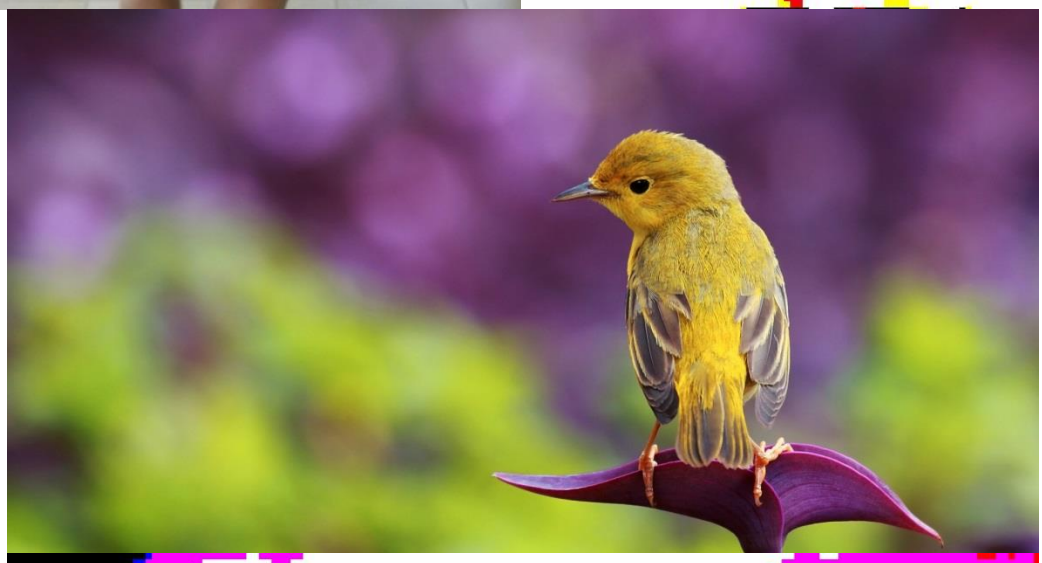


	Procesador	RAM	Ciclos
Máquina 1	i5-M460	4 GB SDRAM	2,53Ghz
Máquina 2	i7-3630QM	8 GB DDR3	2,40GHz
Máquina 3	I7	12 GB RAM	2.40GHz x8

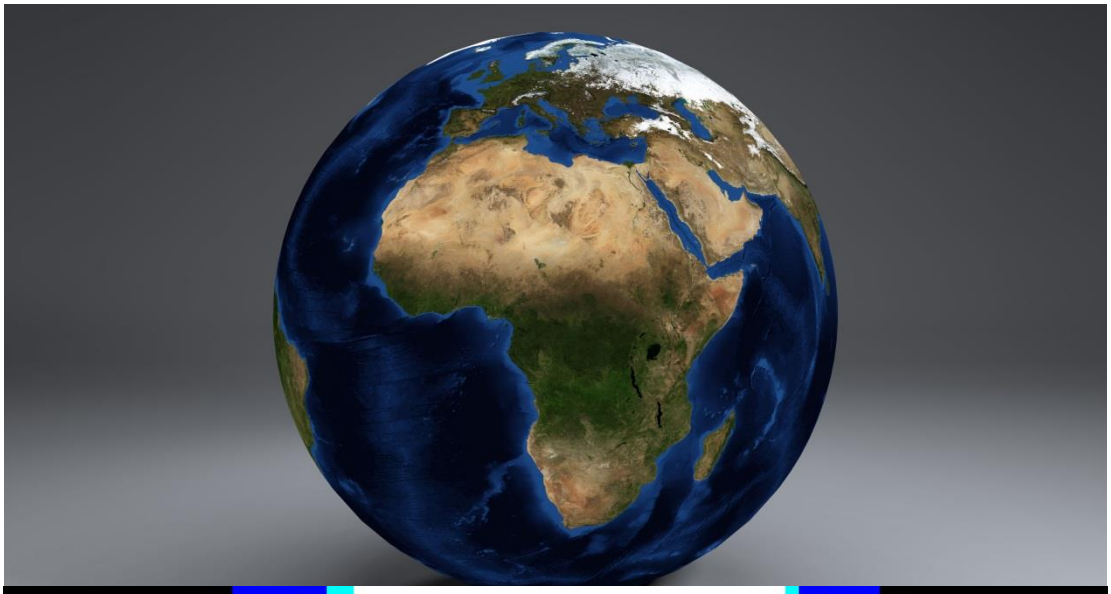
Ahora exponemos las imágenes originales y el resultado observado.

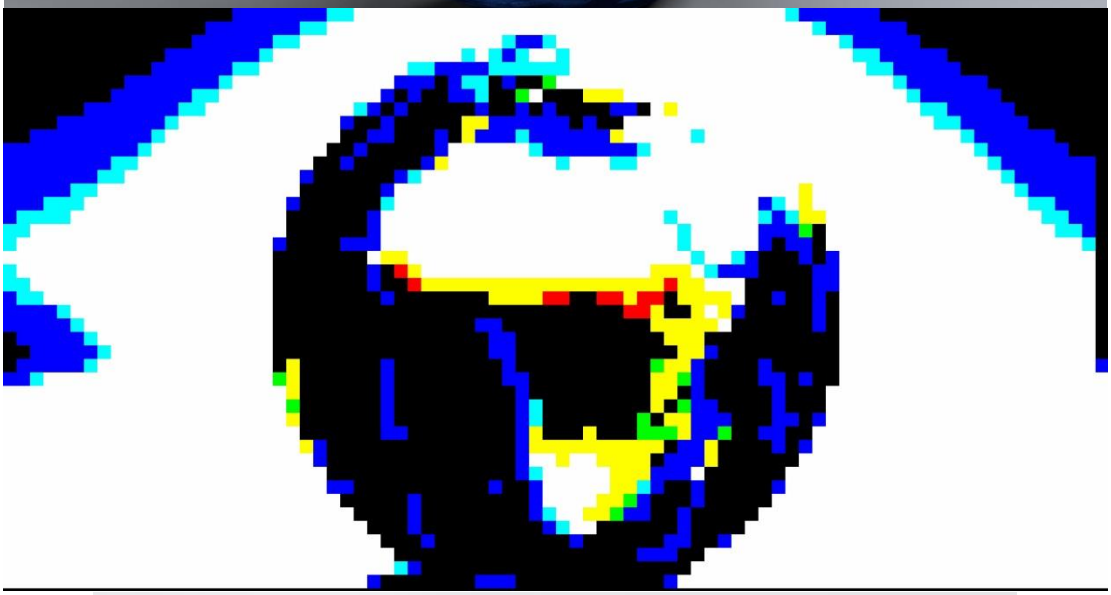






PIX16.C





PIX16.C

BIBLIOGRAFÍA

(http://www.awitness.org, s.f.)

(December, 2013)

(steeldriver, 2015)

(ImageMagic.org, s.f.)