Analisis Desafío 1

Jorim de Jesus Saltarin Villamizar
Duvian Alexander Flores Munera

Informática II
Augusto Enrique Salazar Jiménez

Universidad de Antioquia

12 de abril de 2025

Análisis Desafío

El objetivo de este desafío es la reconstrucción de una imagen BMP, la cual se encuentra distorsionada, todo esto a través de una lógica inversa para recuperar su forma original. Una correcta reconstrucción implica analizar las transformaciones que se le aplicaron en un inicio, con el fin de revertir esos cambios. Lo que significa que se requiere entender la lógica detrás de la distorsión para luego aplicar el proceso contrario.

Cómo podríamos reconocer cada color

Existen varias paletas de colores, por las cuales hacen un poco complicado en aplicar una forma de generar imágenes, pero entre esas descubrimos una forma rentable de evitar extender tanto la forma en la que se ubica cada color en el pixel, es decir que se necesita un byte para cada píxel (donde en color verdadero necesita 3 veces más).

También se puede considerar que los colores en las imágenes generalmente están representados por una tabla de colores o por colores rgb con valores que van desde 0–255 o 0–65535, esto significa que cada color está representado por una combinación de números específica. Además también podría juzgar la forma en que se generan los colores de una manera tan interesante como la mezcla de colores primarios de la luz(rojo, verde y azul). Los colores secundarios de la luz (cian, amarillo y magenta). Haciendo uso de estas podría acortarse la sintaxis a utilizar en el código y una posible mejor facilidad de reconocer los colores en cada pixel.

Posibles soluciones

- Para llevar a cabo este desafío debemos saber cómo funcionan los archivos BMP y la memoria dinámica.
- Revisando el código proporcionado, descubrimos que el almacenamiento de cada píxel después del enmascaramiento sobrepasa el intervalo de 0 a 255. Lo que nos hace pensar que una posible solución, se resuelve en este apartado del código.
- Investigando encontramos información que puede ser útil, como aplicar el uso de números hexadecimales.
- Además también consideramos la investigación acerca de tarjetas gráficas y circuitos integrados, lo que convierte los 1 y 0 de una imagen que se envía a la pantalla para producir colores predefinidos.
- Aplicar el uso de una paleta de colores que facilite y disminuya la dificultad de este proceso en la aplicación con los colores primarios y secundarios de la luz que es comúnmente utilizado para imágenes digitales.

Informe de Desarrollo del Desafío de Procesamiento de Imágenes

Análisis del problema y consideraciones para la alternativa de solución propuesta

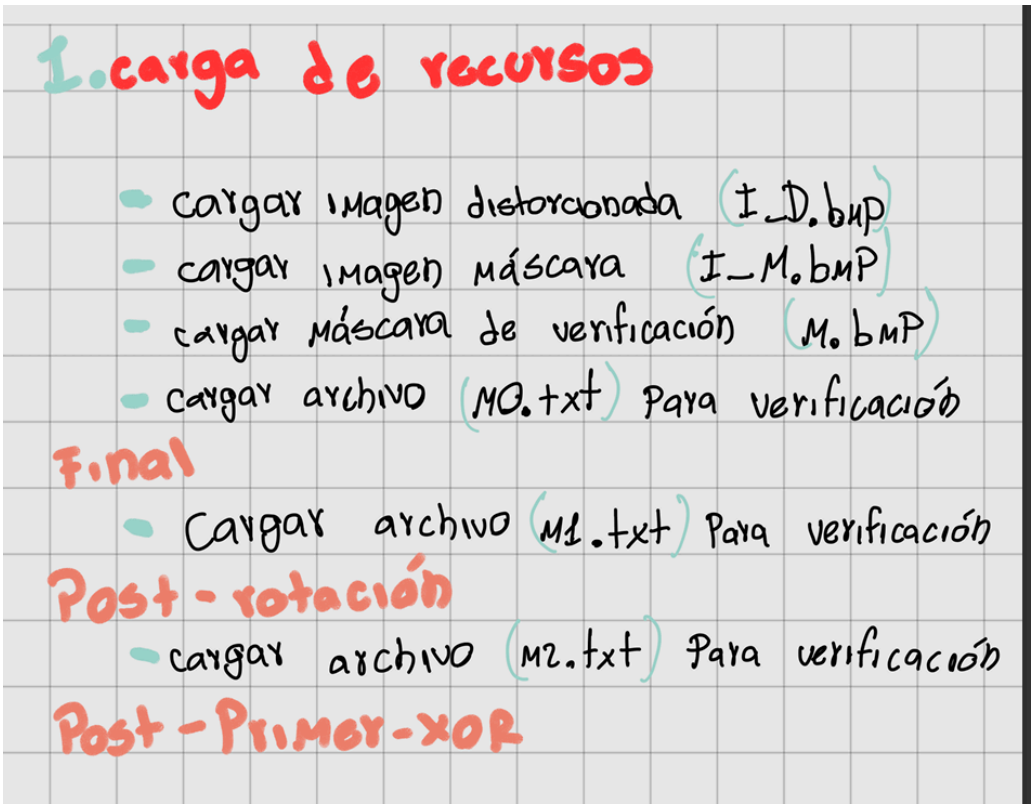
El desafío consiste en el desarrollo de un sistema de procesamiento de imágenes que permite reconstruir una imagen distorsionada mediante operaciones bit a bit. El sistema trabaja con 3 imágenes principales.

I_D.bmp: Imagen distorsionada que debe ser procesada
I_M.bmp: Imagen de máscara que se utiliza para las operaciones XOR
M.bmp: Máscara adicional utilizada para verificaciones entre operaciones.

Además, se utilizaron 3 archivos de texto (M0.txt, M1.txt, M2.txt) los cuales contienen información de control sobre píxeles específicos, incluyendo una semilla (posición inicial) y valores RGB esperados en distintas etapas del procesamiento.

El objetivo principal es aplicar una serie de transformaciones (operaciones XOR y rotación de bits) para reconstruir la imagen original, guardandola como I_DO.bmp

Esquema de tareas definidas en el desarrollo de los algoritmos



2. Verificación inicial imágenes coinciden

- comprobar que las dimensiones de las imagenes coincidan

3. Primera transformación

- Aplicar operación XOR entre (I_D) e (I_M)
- verificar resultados contra datos en (M2.txt)

4. segunda transformación

- Aplicar rotación de bits (3 bits a la izquierda) al resultado anterior
- verificar resultado contra datos en (M1.txt)

5. tercera transformación

- Aplicar operación XOR nuevamente entre resultado anterior e (I_M)
- verificar resultado contra datos en (M0.txt)

6. Generación de salidad

- Guardar imagen reconstruida como (I_D0.bmp)

Algoritmos implementados

1. **Carga de imagenes:** se utilizo la funcion que venia en |el código fuente

```
//Función para cargar los pixeles de una imagen*****************
unsigned char* cargarImagen(QString input, int &ancho, int &alto)
    QImage imagen(input);
    if (imagen.isNull()) {
        cout << "Error: No se pudo cargar la imagen." << endl;
        return nullptr;
    }
    imagen = imagen.convertToFormat(QImage::Format_RGB888);
    ancho = imagen.width();
    alto = imagen.height();
    int tamañoDatos = ancho * alto * 3;
    unsigned char* datosPixel = new unsigned char[tamañoDatos];
    for (int y = 0; y < alto; ++y) {
        const uchar* lineaOrigen = imagen.scanLine(y);
        unsigned char* lineaDestino = datosPixel + y * ancho * 3;
        memcpy(lineaDestino, lineaOrigen, ancho * 3);
    }
    return datosPixel;
}
```

2. **Carga de máscaras de verificación:** fue utilizado de parte del código fuente proporcionado.

```
//Función para cargar una máscara desde archivo*****************
unsigned int* cargarMascara(const char* nombreArchivo, int &semilla, int &num_pixels){
    // Abrir el archivo que contiene la semilla y los valores RGB
    ifstream archivo(nombreArchivo);
    if (!archivo.is_open()) {
        // Verificar si el archivo pudo abrirse correctamente
        cout << "No se pudo abrir el archivo." << endl;
        return nullptr;
    }
    archivo >> semilla;
    int r, g, b;
    while (archivo >> r >> g >> b) {
        num_pixels++;
    }
    archivo.close();
    archivo.open(nombreArchivo);
    if (!archivo.is_open()) {
        cout << "Error al reabrir el archivo." << endl;
        return nullptr;
    }
    unsigned int* RGB = new unsigned int[num_pixels * 3];
    archivo >> semilla;
    for (int i = 0; i < num_pixels * 3; i += 3) {
        archivo >> r >> g >> b;
        RGB[i] = r;
        RGB[i + 1] = g;
        RGB[i + 2] = b;
    }
    archivo.close();
    cout << "Semilla: " << semilla << endl;
    cout << "Cantidad de píxeles leídos: " << num_pixels << endl;
    return RGB;
}
```

3. **Operación XOR:** Implementa la operación XOR bit a bit entre dos imágenes .bmp

```
//Aplica operación XOR entre imagen y máscara*************************
void aplicarXOR(unsigned char* imagenfinal, unsigned char* idistorcionada, int tamaño) {
    for (int i = 0; i < tamaño; i++) {
        imagenfinal[i] ^= idistorcionada[i];
    }
}
```

4. **Rotación de bits:** Implementa la rotación de bits hacia la izquierda para cada byte de la imagen

```
//Funcion para la rotacion de los bits hacia la izquierda*********
void rotacionBits(unsigned char* datos, int tamaño, int totalBits){
    for(int i = 0; i < tamaño; i++){
        datos[i] = (datos[i] << totalBits) | (datos[i] >> (8 - totalBits));
    }
}
```

5. **Guardado de imagen:** Este algoritmo crea una nueva imagen a partir de los datos procesados y la guarda en formato BMP, utilizada por parte del codigo fuente.

```
//Función para guardar la imagen procesada*************************
bool guardarImagen(unsigned char* datosPixel, int ancho, int alto, QString rutaSalida) {
    QImage imagenSalida(ancho, alto, QImage::Format_RGB888);
    for (int y = 0; y < alto; ++y) {
        memcpy(imagenSalida.scanLine(y), datosPixel + y * ancho * 3, ancho * 3);
    }
    if (!imagenSalida.save(rutaSalida, "BMP")) {
        cout << "Error: No se pudo guardar la imagen." << endl;
        return false;
    } else {
        cout << "Imagen guardada como " << rutaSalida.toStdString() << endl;
        return true;
    }
}
```

Problemas de desarrollo que se afrontó

Durante el desarrollo del proyecto se enfrentaron los siguientes desafíos:

1. Verificación de operaciones bit a bit: La comprobación de que las operaciones XOR y rotación de bits produjeran exactamente los resultados esperados según los archivos de verificación fue un reto de precisión.
2. Algunos errores a la hora de abrir archivos correctamente.
3. Durante las pruebas iniciales, los resultados de las operaciones XOR y rotación de bits no coincidían exactamente con los valores esperados en los archivos de verificación
4. Problemas que la correcta liberación de la memoria, ya que se estaba trabajando con arreglos dinámicos para almacenar los datos de las imágenes

Evolución de la solución

El proyecto partió de un análisis fundamental del problema: reconstruir una imagen BMP distorsionada mediante ingeniería inversa, identificando las transformaciones aplicadas originalmente para poder revertirlas. Este análisis inicial fue crucial para establecer las bases conceptuales del proyecto.

Al principio del análisis, se investigaron varios aspectos que se consideraron importantes:

1. **Representación de colores:** Se estudió cómo diferentes formatos representan el color desde paletas indexadas hasta representaciones RGB de 8, 16 o 24 bits. Se identificó que trabajar con el formato RGB888 (donde cada color ocupa 3 bytes) proporcionaba la precisión necesaria para las operaciones a realizar.
2. **Fundamentos de imagen digital:** Se profundizó en la forma en que las imágenes digitales representan colores mediante la combinación de los colores primarios de luz (rojo, verde y azul) y cómo estos forman los colores secundarios (cian, amarillo y magenta).
3. **Estructura de archivos BMP:** Se analizó la estructura de los archivos BMP para entender cómo acceder y manipular directamente los datos de píxeles.

Desarrollo

El desarrollo progresa desde estas bases conceptuales hacia implementaciones concretas:

1. **Primeras exploraciones:** Se identificó que el almacenamiento después del enmascaramiento excede el rango estándar de 0-255, lo que indicaba la necesidad de operaciones matemáticas específicas.
2. **Investigación de soluciones:** Se investigaron enfoques basados en:
    - Uso de números hexadecimales para representación
    - Técnicas de procesamiento de imágenes inspiradas en tarjetas gráficas
    - Manipulación directa de bits
3. **Implementación de funciones básicas:** Se utilizaron responsablemente las funciones para cargar y guardar imágenes, proporcionados en el código fuente.
4. **Incorporación de operaciones bit a bit:** Se implementaron las operaciones XOR y rotación de bits como funciones independientes.
5. **Sistema de verificación:** Se agregó la capacidad de verificar resultados contra valores esperados en cada etapa del procesamiento utilizando la máscara y los archivos .txt.
6. **Refinamiento:** Se optimizaron los algoritmos.