Informe Desafío I - Análisis y Diseño de la Solución

1. Introducción

Este documento presenta el desarrollo de un programa en C++ con Qt para reconstruir una imagen original (IO) a partir de una imagen transformada (ID) mediante operaciones a nivel de bits y enmascaramiento. El informe detalla el análisis, diseño, implementación y resultados del proyecto, siguiendo los requisitos establecidos en el desafío.

2. Análisis del Problema

* 1. Definición General

Se requiere reconstruir una imagen BMP de 24 bits que fue modificada mediante:

Operaciones a nivel de bits: XOR y rotaciones (izquierda/derecha).

Enmascaramiento: Suma de una máscara (M) a regiones específicas de la imagen transformada, almacenando los resultados en archivos .txt.

* 1. Datos de Entrada
* I\_D.bmp: Imagen transformada (resultado final).
* IM.bmp: Imagen aleatoria usada en operaciones XOR.
* M.bmp: Máscara para enmascaramiento.
* M1.txt y M2.txt: Archivos con resultados de enmascaramiento (semilla s y sumas RGB).

2.3. Objetivo

Revertir las transformaciones aplicadas a I\_D.bmp para recuperar IO.bmp, identificando el orden correcto de las operaciones mediante los archivos de enmascaramiento.

# 3. Diseño de la Solución

# 3.1 Estructura del Programa

# El programa se divide en módulos funcionales:

| Módulo | Funciones |
| --- | --- |
| Carga/Exportación | loadPixels, exportImage |
| Operaciones de Bits | xorImages, rotateImageRight, rotateImageLeft |
| Enmascaramiento | loadSeedMasking, verificarEnmascaramiento |
| Reconstrucción | revertTransformation (prueba combinaciones de transformaciones) |

# 3.2 Diagrama de Flujo

# 

# 4. Implementación

# 4.1 Operaciones Clave

# 4.1.1 XOR entre Imágenes

# void xorImages(unsigned char\* img1, unsigned char\* img2, unsigned char\* result, int size) {

# for (int i = 0; i < size; ++i) {

# result[i] = img1[i] ^ img2[i];

# }

# }

# 4.1.2 Rotación de Bits

# unsigned char rotateRight(unsigned char value, int bits) {

# return (value >> bits) | (value << (8 - bits));

# }

# 4.1.3 Verificación de Enmascaramiento

# bool verificarEnmascaramiento(unsigned char\* imagen, unsigned char\* mascara, MaskingResult\* resultado) {

# for (int i = 0; i < resultado->pixelCount \* 3; ++i) {

# int idxImagen = resultado->seed \* 3 + i;

# if ((int(imagen[idxImagen]) + int(mascara[i])) % 256 != resultado->RGB[i]) {

# return false;

# }

# }

# return true;

# }

# 4.2 Algoritmo de Reconstrucción

# bool revertTransformation(unsigned char\* currentImage, unsigned char\* IM, MaskingResult\* masking,

# int width, int height, int& transformationType) {

# unsigned char\* temp = new unsigned char[width \* height \* 3];

# 

# *// Prueba XOR*

# xorImages(currentImage, IM, temp, width \* height \* 3);

# if (masking && verificarEnmascaramiento(temp, IM, masking)) {

# transformationType = 1; *// XOR*

# memcpy(currentImage, temp, width \* height \* 3);

# delete[] temp;

# return true;

# }

# *// Prueba rotación derecha*

# rotateImageLeft(currentImage, temp, width \* height \* 3, 3);

# if (masking && verificarEnmascaramiento(temp, IM, masking)) {

# transformationType = 2; *// Rotación derecha*

# memcpy(currentImage, temp, width \* height \* 3);

# delete[] temp;

# return true;

# }

# delete[] temp;

# return false;

# }

# 5. Resultados y Validación

# 5.1 Caso de Prueba

# Entrada:

# I\_D.bmp (imagen transformada).

# IM.bmp y M.bmp.

# M1.txt y M2.txt (enmascaramientos intermedios).

# Proceso:

# Revertir XOR usando IM.bmp y verificar con M2.txt.

# Revertir rotación de 3 bits y verificar con M1.txt.

# Salida:

# reconstructed.bmp (imagen original recuperada).

# 6. Conclusiones y Lecciones Aprendidas

# 6.1 Logros

# Eficiencia: El algoritmo prueba solo combinaciones válidas gracias a la verificación con archivos .txt.

# Modularidad: Funciones bien definidas para cada operación.

# Robustez: Manejo de errores en carga de archivos y memoria.

# 6.2 Desafíos

# Orden de Transformaciones:

# Solución: Prueba sistemática con revertTransformation.

# Precisión en Enmascaramiento:

# Solución: Uso de módulo 256 para evitar desbordamientos.

# 6.3 Recomendaciones

# Extender el programa para soportar más transformaciones (ej: desplazamientos).

# Implementar paralelismo para acelerar la búsqueda de combinaciones.