

Tentu, berikut adalah dokumentasi lengkap dari kode BlackWhite.cs, yang menjelaskan fungsionalitas dan alasan desainnya blok per blok.

Dokumentasi: BlackWhite.cs

Ringkasan File

- **Namespace:** MiniPhotoshop.Logic.ImageProcessing
 - **Kelas:** BlackWhite
 - **Tujuan:** Kelas ini adalah *utility class* (kelas pembantu) statis yang menyediakan fungsionalitas untuk mengubah gambar berwarna atau grayscale menjadi gambar **hitam putih murni** (biner) berdasarkan nilai ambang batas (*threshold*) tertentu.
 - **Fitur Utama:** Kode ini menggunakan metode LockBits untuk pemrosesan gambar, yang **jauh lebih cepat** daripada metode GetPixel/SetPixel standar.
-

Alasan Utama: Mengapa Menggunakan LockBits?

Sebelum masuk ke detail baris per baris, penting untuk memahami *mengapa* kode ini ditulis dengan cara yang terlihat kompleks.

1. **Metode Lambat (GetPixel/SetPixel):** Metode seperti img.GetPixel(x, y) dan img.SetPixel(x, y) (seperti yang digunakan di ArithmeticOperations.cs) sangat mudah digunakan, tetapi **sangat lambat**. Setiap panggilan adalah *function call* terpisah yang memiliki banyak *overhead* (pengecekan keamanan, *marshalling* data, dll.). Untuk gambar berukuran 1920x1080 (sekitar 2 juta piksel), Anda akan memanggil fungsi ini 2 juta kali untuk membaca dan 2 juta kali untuk menulis, yang bisa memakan waktu beberapa detik.
2. **Metode Cepat (LockBits):** Metode LockBits bekerja secara fundamental berbeda:
 - Ia "mengunci" seluruh data gambar di memori.
 - Ia menyalin *seluruh blok* data piksel dari memori GDI+ (yang tidak aman/unmanaged) ke dalam byte[] array C# (yang aman/managed) dalam *satu operasi cepat*.
 - Kita memanipulasi byte[] array secara langsung, yang secepat kilat (karena ini hanya akses memori dasar).
 - Kita menyalin byte[] array yang sudah diubah kembali ke memori gambar dalam *satu operasi cepat*.

- Ia "membuka kunci" memori.

Kesimpulan: Penggunaan LockBits mengubah operasi yang memakan waktu beberapa detik menjadi operasi yang hampir instan (seringkali hanya beberapa milidetik), yang sangat penting untuk aplikasi pengeditan gambar yang responsif.

Analisis Kode per Blok

Blok 1: using Statements

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

using System.Drawing;
using System.Drawing.Imaging; // <- TAMBAHKAN INI
using System.Runtime.InteropServices; // <- TAMBAHKAN INI
```

- **using System.Drawing;**
 - **Penjelasan:** Mengimpor namespace dasar GDI+ .NET.
 - **Fungsi:** Diperlukan untuk kelas inti seperti Bitmap, Rectangle, dan Color.
- **using System.Drawing.Imaging;**
 - **Penjelasan:** Mengimpor namespace pencitraan GDI+ yang lebih canggih.
 - **Fungsi:** Diperlukan untuk BitmapData, ImageLockMode, dan PixelFormat. Ini adalah inti dari fungsionalitas LockBits.
- **using System.Runtime.InteropServices;**
 - **Penjelasan:** Mengimpor layanan "Interop".
 - **Fungsi:** Diperlukan untuk kelas Marshal, yang menyediakan metode untuk

berinteraksi dengan memori *unmanaged* (memori di luar kendali .NET). Kita menggunakan untuk Marshal.Copy (menyalin data dari pointer memori ke array).

Blok 2: Metode Publik ApplyBinarization

C#

```
public static class BlackWhite
{
    // Fungsi ini tetap sama, hanya memanggil fungsi ApplyThreshold
    public static Bitmap ApplyBinarization(Bitmap originalImage, int sliderStep)
    {
        int threshold;
        switch (sliderStep)
        {
            case 0: threshold = 50; break;
            case 1: threshold = 100; break;
            case 2: threshold = 127; break; // Nilai tengah standar
            case 3: threshold = 150; break;
            case 4: threshold = 200; break;
            default: threshold = 127; break;
        }
        // Panggil fungsi baru yang cepat
        return ApplyThreshold(originalImage, threshold);
    }
}
```

- **public static Bitmap ApplyBinarization(...)**
 - **Penjelasan:** Ini adalah metode public yang menjadi *interface* bagi pengguna kelas ini (misalnya, dari GUI).
 - **Fungsi:** Menerima gambar asli dan sliderStep (langkah slider, misal 0-4). Tujuannya adalah untuk **menerjemahkan** input GUI yang sederhana (sliderStep) menjadi nilai threshold (ambang batas) teknis yang sebenarnya.
 - **switch (sliderStep):** Ini adalah logika bisnis sederhana. Ia memetakan nilai 0 ke 50, 1 ke 100, dst. Ini memisahkan logika antarmuka (GUI) dari logika pemrosesan inti.
 - **return ApplyThreshold(originalImage, threshold);**: Setelah nilai threshold ditentukan, ia memanggil metode ApplyThreshold internal (yang cepat) untuk melakukan pekerjaan sebenarnya.

- **Alasan:** Desain ini sangat baik. Kode GUI tidak perlu tahu tentang angka 127 atau 150; ia hanya perlu tahu "slider ada di langkah ke-2". Kelas ini yang bertanggung jawab atas penerjemahan itu.
-

Blok 3: Metode Inti ApplyThreshold (Bagian 1: Penguncian)

C#

```
// ---INI FUNGSI BARU YANG MENGGUNAKAN LOCKBITS ---
private static Bitmap ApplyThreshold(Bitmap originalImage, int threshold)
{
    // Buat gambar hasil dari gambar asli
    Bitmap resultImage = new Bitmap(originalImage);
    Rectangle rect = new Rectangle(0, 0, resultImage.Width, resultImage.Height);

    // Kunci memori gambar
    BitmapData bmpData = resultImage.LockBits(rect, ImageLockMode.ReadWrite,
        resultImage.PixelFormat);
```

- **private static Bitmap ApplyThreshold(...)**
 - **Penjelasan:** Metode private di mana semua logika pemrosesan cepat terjadi.
- **Bitmap resultImage = new Bitmap(originalImage);**
 - **Penjelasan:** Membuat **salinan** dari gambar asli.
 - **Alasan:** Ini adalah praktik standar dalam *non-destructive editing*. Kita tidak ingin mengubah gambar aslinya. Kita memodifikasi salinannya dan mengembalikan salinan yang sudah dimodifikasi.
- **Rectangle rect = new Rectangle(...)**
 - **Penjelasan:** Membuat objek Rectangle yang mencakup *seluruh* area gambar (dari (0,0) ke (lebar, tinggi)).
 - **Alasan:** LockBits perlu tahu area memori mana yang ingin kita kunci. Dalam hal ini, kita ingin mengunci semuanya.
- **BitmapData bmpData = resultImage.LockBits(...)**
 - **Penjelasan:** Ini adalah perintah inti untuk "mengunci" memori.
 - **Fungsi:**
 - rect: Memberitahu area mana yang akan dikunci.
 - ImageLockMode.ReadWrite: Memberitahu GDI+ bahwa kita berniat untuk **membaca** data piksel dan **menulis** data baru ke dalamnya.

- `resultImage.PixelFormat`: Memberitahu GDI+ untuk menjaga format piksel tetap sama.
 - **Hasil:** `bmpData` adalah objek yang berisi informasi penting tentang memori yang dikunci, terutama alamat awal (`Scan0`) dan ukuran baris (`Stride`).
-

Blok 4: ApplyThreshold (Bagian 2: Persiapan Salin Memori)

C#

```
IntPtr ptr = bmpData.Scan0;
int bytes = Math.Abs(bmpData.Stride) * resultImage.Height;
byte[] rgbValues = new byte[bytes];

// Salin SEMUA data piksel ke array
Marshal.Copy(ptr, rgbValues, 0, bytes);
```

- **IntPtr ptr = bmpData.Scan0;**
 - **Penjelasan:** `Scan0` adalah *pointer* (penunjuk memori) ke **byte pertama** dari data piksel gambar yang dikunci. `IntPtr` adalah tipe data C# untuk menyimpan *pointer*.
- **int bytes = Math.Abs(bmpData.Stride) * resultImage.Height;**
 - **Penjelasan:** Menghitung jumlah total byte dalam data gambar.
 - **bmpData.Stride:** Ini adalah konsep penting. `Stride` adalah panjang satu baris gambar dalam byte. Nilainya **tidak** selalu lebar * 3. Kadang-kadang GDI+ menambahkan "bantalan" (padding) di akhir setiap baris untuk menyelaraskan data di memori agar lebih efisien.
 - **Math.Abs(...):** `Stride` bisa jadi negatif jika gambar disimpan terbalik (bottom-up), jadi `Math.Abs` digunakan untuk memastikan kita mendapatkan nilai absolut untuk perhitungan ukuran.
- **byte[] rgbValues = new byte[bytes];**
 - **Penjelasan:** Membuat array byte C# yang "terkelola" (managed) yang ukurannya sama persis dengan total data gambar.
- **Marshal.Copy(ptr, rgbValues, 0, bytes);**
 - **Penjelasan:** Ini adalah operasi penyalinan berkecepatan tinggi.
 - **Fungsi:** Menyalin bytes data, dimulai dari alamat memori `ptr` (gambar), ke dalam array `rgbValues`, dimulai dari indeks 0.
 - **Alasan:** Sekarang semua data piksel ada di dalam `rgbValues`, kita bisa memprosesnya dengan aman dan cepat menggunakan C# murni.

Blok 5: ApplyThreshold (Bagian 3: Inti Algoritma)

C#

```
int bytesPerPixel = Image.GetPixelFormatSize(resultImage.PixelFormat) / 8;
int stride = bmpData.Stride;

// Proses array (super cepat)
for (int y = 0; y < resultImage.Height; y++)
{
    int rowOffset = y * stride;
    for (int x = 0; x < resultImage.Width; x++)
    {
        int i = rowOffset + (x * bytesPerPixel);

        // 1. Grayscale (urutan BGR di memori)
        int gray = (int)(rgbValues[i + 2] * 0.299 + // R
                        rgbValues[i + 1] * 0.587 + // G
                        rgbValues[i] * 0.114); // B

        // 2. Threshold
        byte bwValue = (gray < threshold) ? (byte)0 : (byte)255;

        // Setel kembali nilainya
        rgbValues[i] = bwValue;
        rgbValues[i + 1] = bwValue;
        rgbValues[i + 2] = bwValue;
    }
}
```

- **int bytesPerPixel = ...:** Menghitung berapa byte yang digunakan per piksel (misalnya, 3 untuk 24bpp, 4 untuk 32bpp). / 8 karena GetPixelFormatSize mengembalikan *bits*.
- **int stride = bmpData.Stride;:** Mendapatkan nilai *stride* (panjang baris dalam byte) untuk navigasi.
- **for (int y = 0; ...):** Perulangan untuk setiap baris (tinggi).
- **int rowOffset = y * stride;:** Menghitung indeks *byte* awal untuk baris *y* saat ini.

- **for (int x = 0; ...):** Perulangan untuk setiap piksel dalam baris (lebar).
 - **int i = rowOffset + (x * bytesPerPixel);**: Menghitung indeks byte awal untuk piksel (x, y) saat ini di dalam array `rgbValues`.
 - **int gray = (int)(...);**
 - **Penjelasan:** Ini adalah dua langkah dalam satu:
 1. Membaca nilai R, G, B dari piksel.
 2. Mengonversinya menjadi nilai *grayscale* (skala keabuan) menggunakan formula luminans standar.
 - **Penting:** Perhatikan urutannya: `[i+2]` adalah **Red**, `[i+1]` adalah **Green**, dan `[i]` adalah **Blue**. Ini adalah urutan memori **BGR** standar yang digunakan oleh GDI+ Bitmap, bukan RGB.
 - **byte bwValue = (gray < threshold) ? (byte)0 : (byte)255;**
 - **Penjelasan:** Ini adalah **algoritma binarisasi**.
 - **Fungsi:** Menggunakan operator ternary. Jika nilai gray (keabuan) piksel **kurang dari** `threshold`, maka nilai hitam-putihnya (`bwValue`) adalah 0 (hitam pekat). Jika tidak (jika gray lebih besar atau sama dengan `threshold`), nilainya adalah 255 (putih pekat).
 - **rgbValues[i] = bwValue; (dan i+1, i+2)**
 - **Penjelasan:** Menulis kembali nilai `bwValue` (yang sekarang 0 atau 255) ke dalam array.
 - **Alasan:** Kita mengatur *ketiga* channel (Biru, Hijau, Merah) ke nilai yang sama. Ini untuk memastikan piksel hasilnya adalah benar-benar hitam (0,0,0) atau putih (255,255,255).
-

Blok 6: ApplyThreshold (Bagian 4: Penyalinan Kembali & Buka Kunci)

C#

```
// Salin SEMUA data dari array kembali ke gambar
Marshal.Copy(rgbValues, 0, ptr, bytes);

// Buka kunci memori
resultImage.UnlockBits(bmpData);
return resultImage;
}
```

- **Marshal.Copy(rgbValues, 0, ptr, bytes);**
 - **Penjelasan:** Ini adalah kebalikan dari Marshal.Copy sebelumnya.
 - **Fungsi:** Menyalin bytes data, dimulai dari array rgbValues (indeks 0), kembali ke alamat memori ptr (gambar).
 - **Alasan:** Ini adalah operasi yang "menerapkan" semua perubahan yang kita buat di rgbValues kembali ke Bitmap yang sebenarnya.
- **resultImage.UnlockBits(bmpData);**
 - **Penjelasan:** Perintah untuk "membuka kunci" memori gambar.
 - **Alasan:** Ini **WAJIB** dilakukan. Jika Anda lupa memanggil UnlockBits, memori gambar akan tetap terkunci selamanya, yang akan menyebabkan kebocoran memori (*memory leak*) besar dan aplikasi Anda akan crash.
- **return resultImage;**
 - **Penjelasan:** Mengembalikan Bitmap yang sekarang berisi data piksel hitam-putih yang sudah dimodifikasi.