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

Dokumentasi: SelectionColor.cs

Ringkasan File

- **Namespace:** MiniPhotoshop.Logic.ImageProcessing
- **Kelas:** SelectionColor
- **Tujuan:** Kelas ini adalah *utility class* (kelas pembantu) statis yang menyediakan satu fungsi: ApplySelection. Fungsi ini melakukan operasi "color keying" atau "magic wand" sederhana. Ia mengambil gambar asli dan satu warna target (selectedColor). Hasilnya adalah gambar baru di mana **hanya** piksel yang warnanya *sama persis* dengan selectedColor yang dipertahankan. Semua piksel lain diubah menjadi putih.
- **Fitur Utama:** Kode ini menggunakan metode LockBits untuk performa tinggi, dan secara efisien mengunci *dua* gambar sekaligus (satu untuk dibaca, satu untuk ditulis) untuk pemrosesan yang cepat.

Alasan Utama: Mengapa Menggunakan LockBits?

Seperti file Brightness.cs dan ImageNegation.cs, kode ini menggunakan LockBits untuk **kecepatan maksimum**.

1. **Metode Lambat (GetPixel/SetPixel):** Melakukan perbandingan warna piksel per piksel menggunakan GetPixel akan sangat lambat pada gambar besar.
2. **Metode Cepat (LockBits):** Kode ini mengunci data gambar di memori dan memanipulasinya sebagai array byte[] besar. Ini adalah cara tercepat untuk memproses gambar di C#.
3. **Penguncian Ganda:** Tidak seperti file lain yang mengunci satu gambar, kode ini mengunci *dua* gambar:
    - originalImage dikunci sebagai ImageLockMode.ReadOnly (hanya dibaca).
    - resultImage dikunci sebagai ImageLockMode.WriteOnly (hanya ditulis).
      Ini adalah pola yang sangat efisien untuk operasi yang membaca dari satu sumber dan menulis ke tujuan yang terpisah.

Analisis Kode per Blok

Blok 1: using Statements

C#

```
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
```

- **using System.Drawing.Imaging;**: Diperlukan untuk kelas-kelas inti LockBits seperti BitmapData, ImageLockMode, dan PixelFormat.
- **using System.Runtime.InteropServices;**: Diperlukan untuk kelas Marshal, yang menyediakan metode Marshal.Copy untuk menyalin data antara memori *unmanaged* GDI+ dan array byte[] C# *managed*.

Blok 2: Metode ApplySelection (Bagian 1: Setup)

C#

```
  public static class SelectionColor
  {
      public static Bitmap ApplySelection(Bitmap originalImage, Color selectedColor)
      {
          Bitmap resultImage = new Bitmap(originalImage.Width,
originalImage.Height,originalImage.PixelFormat);
          int targetArgb = selectedColor.ToArgb();
```

```
        Rectangle rect = new Rectangle(0, 0, originalImage.Width, originalImage.Height);
```

- **public static Bitmap ApplySelection(...)**: Metode publik yang mengambil gambar sumber dan Color yang ingin dipilih.
- **Bitmap resultImage = new Bitmap(...)**: Membuat Bitmap baru untuk hasil. Sangat penting bahwa ia dibuat dengan lebar, tinggi, dan PixelFormat yang sama dengan aslinya untuk memastikan LockBits bekerja dengan benar dan data *stride* cocok.
- **int targetArgb = selectedColor.ToArgb();**: Ini adalah **optimasi kunci**. Daripada membandingkan 4 nilai (R, G, B, A) di dalam *loop* untuk setiap piksel, kita mengubah warna target menjadi satu int (Integer 32-bit) di awal. Membandingkan dua int jauh lebih cepat daripada membandingkan empat byte.
- **Rectangle rect = ...**: Membuat Rectangle yang mendefinisikan area kerja kita (seluruh gambar).

Blok 3: ApplySelection (Bagian 2: Penguncian Ganda)

C#

```
        BitmapData originalData = originalImage.LockBits(rect,
ImageLockMode.ReadOnly,originalImage.PixelFormat);
        BitmapData resultData = resultImage.LockBits(rect,
ImageLockMode.WriteOnly,resultImage.PixelFormat);

        IntPtr ptrAsli = originalData.Scan0;
        IntPtr ptrHasil = resultData.Scan0;
```

- **BitmapData originalData = ...**: Mengunci gambar *asli*. Mode ImageLockMode.ReadOnly digunakan karena kita hanya akan membacanya, yang lebih aman dan bisa sedikit lebih cepat.
- **BitmapData resultData = ...**: Mengunci gambar *hasil* yang baru dan kosong. Mode ImageLockMode.WriteOnly digunakan karena kita hanya akan menulis ke dalamnya.
- **IntPtr ptrAsli = originalData.Scan0;**: Mendapatkan *pointer* (penunjuk memori) ke data piksel pertama dari gambar *asli*.
- **IntPtr ptrHasil = resultData.Scan0;**: Mendapatkan *pointer* ke data piksel pertama dari gambar *hasil*.

Blok 4: ApplySelection (Bagian 3: Persiapan Array)

C#

```
        int bytes = Math.Abs(originalData.Stride) * originalImage.Height;
        byte[] rgbAsli = new byte[bytes];
        byte[] rgbHasil = new byte[bytes];

        Marshal.Copy(ptrAsli, rgbAsli, 0, bytes);
```

- **int bytes = ...**: Menghitung total jumlah *byte* dalam data gambar (panjang baris/stride dikali tinggi).
- **byte[] rgbAsli = new byte[bytes];**: Membuat array byte[] C# untuk menampung data dari gambar *asli*.
- **byte[] rgbHasil = new byte[bytes];**: Membuat array byte[] C# untuk *membangun* data gambar *hasil*.
- **Marshal.Copy(ptrAsli, rgbAsli, 0, bytes);**: Operasi penyalinan super cepat. Menyalin semua data piksel dari memori *unmanaged* (ptrAsli) ke dalam array managed C# (rgbAsli) agar kita bisa bekerja dengannya.

Blok 5: ApplySelection (Bagian 4: Inti Algoritma)

C#

```
        int bytesPerPixel = Image.GetPixelFormatSize(originalImage.PixelFormat) / 8;
        int stride = originalData.Stride;

        for (int y = 0; y < originalImage.Height; y++)
        {
            int rowOffset = y * stride;
            for (int x = 0; x < originalImage.Width; x++)
            {
```

```
            int i = rowOffset + (x * bytesPerPixel);

            byte b = rgbAsli[i];
            byte g = rgbAsli[i + 1];
            byte r = rgbAsli[i + 2];
            byte a = (bytesPerPixel == 4) ? rgbAsli[i + 3] : (byte)255;

            int currentArgb = (a << 24) | (r << 16) | (g << 8) | b;

            if (currentArgb == targetArgb)
            {
                rgbHasil[i] = b;
                rgbHasil[i + 1] = g;
                rgbHasil[i + 2] = r;
                if (bytesPerPixel == 4) rgbHasil[i + 3] = a;
            }
            else
            {
                rgbHasil[i] = 255;
                rgbHasil[i + 1] = 255;
                rgbHasil[i + 2] = 255;
                if (bytesPerPixel == 4) rgbHasil[i + 3] = a;
            }
        }
    }
```

- **int bytesPerPixel = ... / int stride = ...**: Variabel pembantu untuk navigasi array.
- **for (int y...) / for (int x...)**: Perulangan standar untuk mengunjungi setiap piksel.
- **int i = rowOffset + ...**: Menghitung indeks *byte* awal untuk piksel saat ini (menunjuk ke channel Biru dalam urutan BGR).
- **byte b = rgbAsli[i]; ...**: Membaca nilai B, G, R, dan A dari array *asli*. Perhatikan (bytesPerPixel == 4) ? ... : (byte)255; adalah cara aman untuk menangani gambar yang mungkin tidak memiliki channel alpha (seperti 24bpp), di mana alpha diasumsikan 255 (opaque).
- **int currentArgb = (a << 24) | (r << 16) | (g << 8) | b;**: Ini adalah **optimasi kunci kedua**. Ini menggunakan operasi *bitwise* (pergeseran bit << dan OR |) untuk menggabungkan empat nilai byte (A, R, G, B) kembali menjadi satu int yang formatnya sama persis dengan targetArgb yang kita buat sebelumnya.
- **if (currentArgb == targetArgb)**: Perbandingan inti yang super cepat.
- **BLOK IF (MATCH)**: { rgbHasil[i] = b; ... }
  - **Fungsi:** Jika warna piksel *sama persis* dengan warna target.
  - **Alasan:** Salin nilai B, G, R, A asli ke array rgbHasil. Piksel ini "lolos" seleksi.
- **BLOK ELSE (NO MATCH)**: { rgbHasil[i] = 255; ... }

- **Fungsi:** Jika warna piksel *tidak sama*.
- **Alasan:** Tulis nilai B=255, G=255, R=255 (Putih) ke array rgbHasil. Piksel ini "dihapus" atau diganti dengan warna putih. Channel alpha-nya tetap disalin (rgbHasil[i+3] = a) untuk menjaga transparansi asli jika ada.

Blok 6: ApplySelection (Bagian 5: Salin Kembali & Selesai)

C#

```
        Marshal.Copy(rgbHasil, 0, ptrHasil, bytes);

        originalImage.UnlockBits(originalData);
        resultImage.UnlockBits(resultData);

        return resultImage;
    }
  }
}
```

- **Marshal.Copy(rgbHasil, 0, ptrHasil, bytes);**: Ini adalah kebalikan dari Copy sebelumnya. Ini mengambil array rgbHasil yang sudah kita proses dan menyalin semua datanya kembali ke memori *unmanaged* dari gambar *hasil* (ptrHasil).
- **originalImage.UnlockBits(originalData);**: Melepaskan kunci pada gambar *asli*.
- **resultImage.UnlockBits(resultData);**: Melepaskan kunci pada gambar *hasil*.
- **return resultImage;**: Mengembalikan gambar hasil yang baru, yang sekarang berisi piksel terpilih dan latar belakang putih.