Лабораторная работа №1. Введение в обработку цифровых изображений

Начало работы

В Visual Studio создайте проект **Приложение WPF (Microsoft)**, данный курс использует платформу .NET и технологии WPF (примеры выполнены с использованием платформы .NET 7.0):

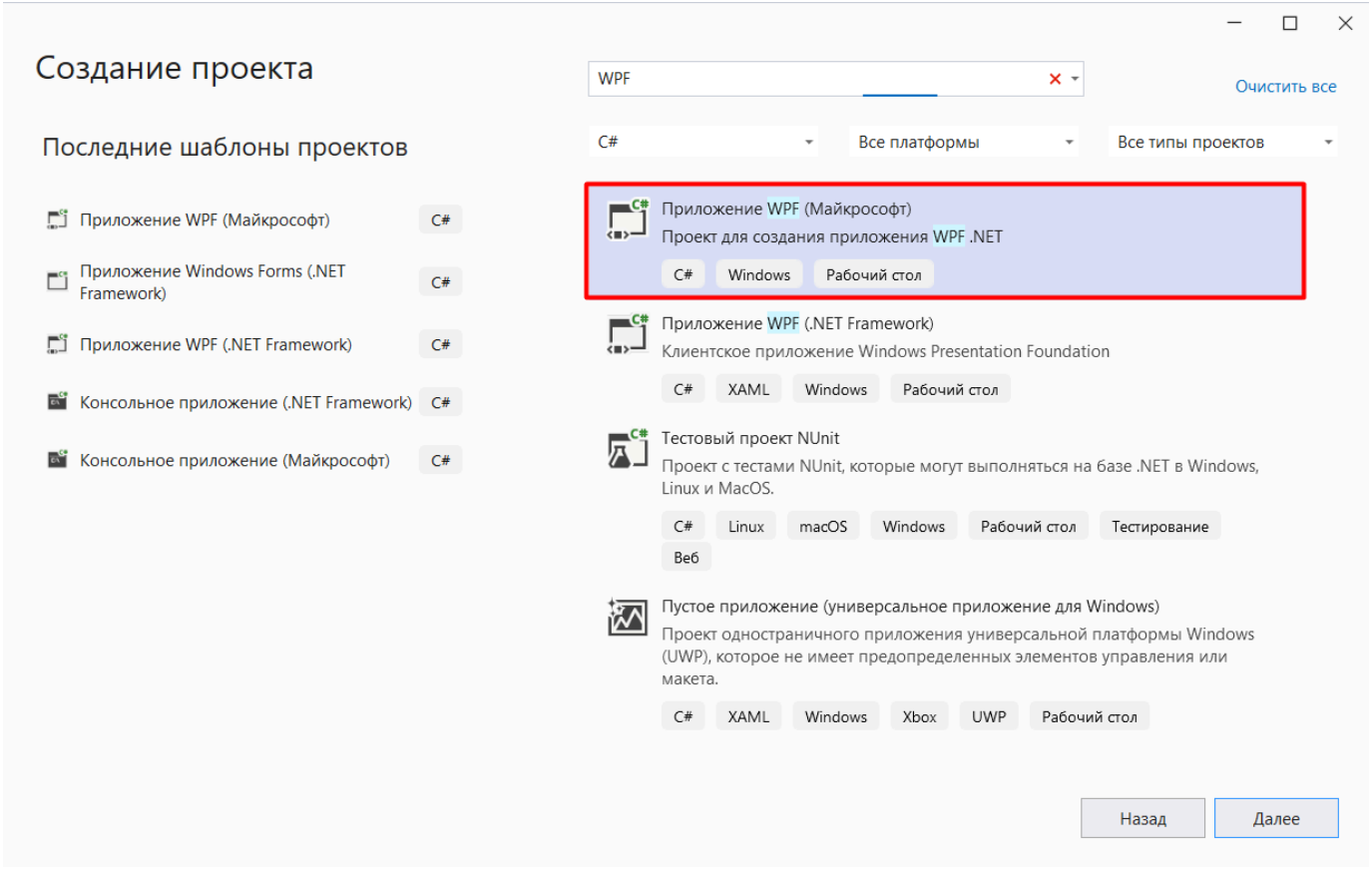

Рисунок 1 - Создание проекта WPF.

После создания проекта перейдите во вкладку **Проект** в верхней панели приложения и выберите пункт **Управление проектами Nuget**:

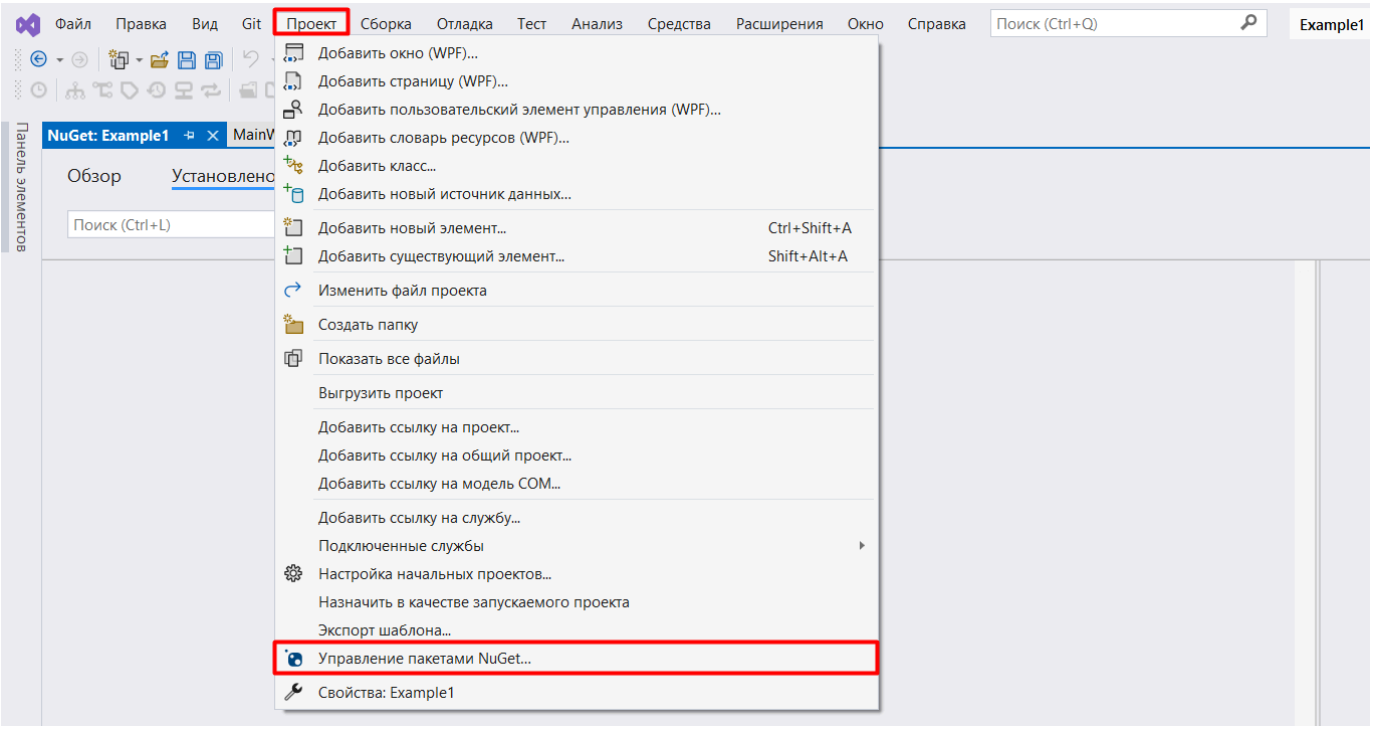

Рисунок 2 - Открытие вкладки управления пакетами.

Затем перейдите в панель **Обзор** и установите следующие пакеты в проект:

- **Emgu.CV** - обертка для библиотеки компьютерного зрения и обработки изображений **OpenCV**. Позволяет использовать функции библиотеки в .NET приложениях.
- **Emgu.CV.runtime.windows** - компонент библиотеки требующийся для запуска приложений на платформе windows.

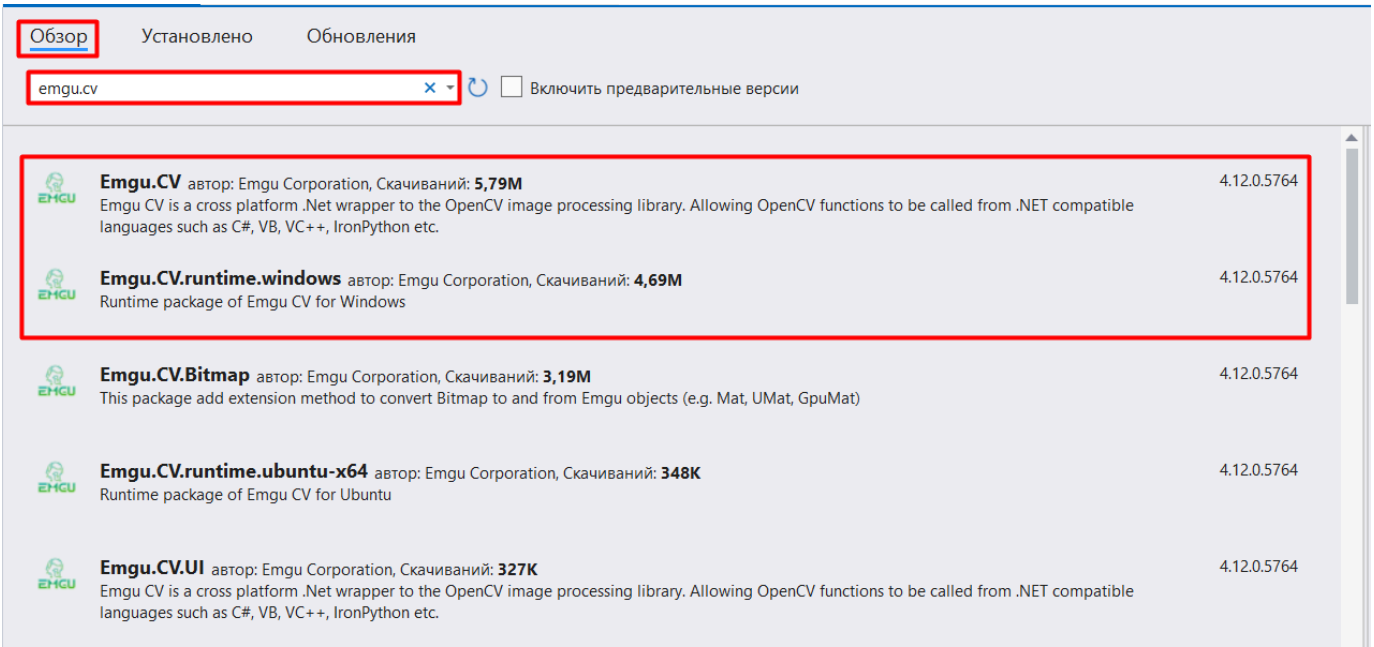

Рисунок 3 - Установка пакетов Emgu.CV.

В проект необходимо добавить пространства имен `Emgu.CV.Structure` и `Emgu.CV`. В примере используется следующие пространства имен:

Блок кода №1. Пространства имен проекта.

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using Microsoft.Win32;

//Пространства имен которые необходимо добавить
using Emgu.CV.Structure;
using Emgu.CV;
```

OpenCV является библиотекой с открытым исходным кодом, которая на данный момент является индустриальным стандартом в области компьютерного зрения и обработки изображений. Использование OpenCV в данном курсе обусловлено её универсальностью, а обертка в виде Emgu.CV обладает возможностями построения удобного пользовательского интерфейса в WPF.

Обработка растровых изображений

Существует две основных графических технологии - **растровая** и **векторная** графика. Первая представляет изображение в виде сетки пикселей , а вторая – с помощью линий и геометрических фигур. В данном курсе анализ и обработка изображений будет осуществляться с растровыми изображениями.

Как было сказано ранее в растровой графике изображение представлено в виде сетки пикселей. **Пиксель** - это наименьшая единица растрового изображения которая хранит информацию о цвете одной конкретной точки изображения. Если переводить сетку пикселей на язык программирования то изображения можно представить как двумерный массив или матрицу структур хранящих некоторое количество целочисленных и/или вещественных значений о характеристике цвета.

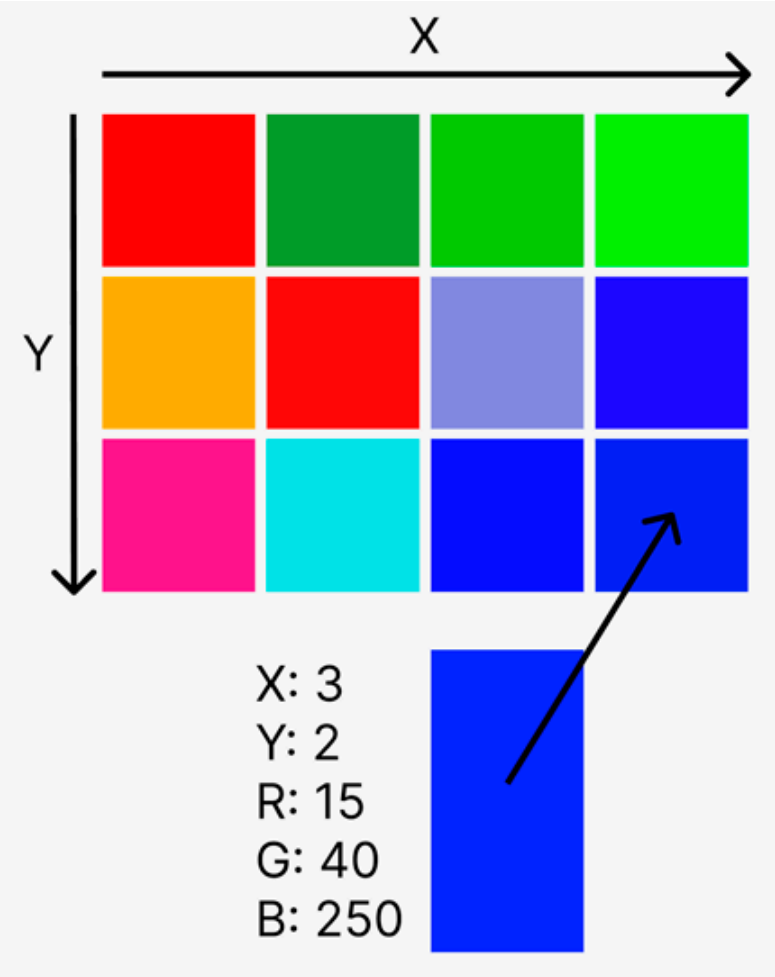

Рисунок 4 - Представление изображения в виде матрицы пикселей и выделение конкретного пикселя как структуры хранящей данные о цвете.

В библиотеке Emgu.CV основным классом для хранения и обработки растровых изображений является обобщенный класс `Image<TColor, TDepth>`. В отличие от стандартного `Bitmap` в .NET, этот класс обеспечивает строгую типизацию цветового пространства и глубины цвета.

TColor — это цветовое пространство изображения, оно определяет сколько каналов имеет изображения и что они означают. Все типы цветов являются структурами:

Тип	Кол-во каналов	Краткое описание
Bgr	3	Стандартный формат для OpenCV. **Важно:** каналы идут именно в порядке Синий-Зеленый-Красный, а не в привычном варианте RGB.
Bgra	4	Тот же Bgr, только с добавлением канала прозрачности Alpha.
Gray	1	Изображение в оттенках серого (черно-белое)
Hsv	3	Цветовая модель, использующаяся для специфических алгоритмов цветокоррекции и сегментации.

Почему BGR? OpenCV использует формат **BGR** (Blue-Green-Red) вместо привычного **RGB** по историческим причинам. Когда библиотека создавалась (в 1999 году), архитектура процессоров и камер была оптимизирована под формат хранения данных, при котором байты в памяти располагались в обратном порядке. Использование BGR позволяло процессору считывать цвет пикселя как единое целое число максимально быстро, не тратя драгоценные ресурсы "железа" того времени на лишнюю перестановку байтов. Сейчас это осталось индустриальным стандартом для обратной совместимости.

TDepth — определяет тип данных использующийся для хранения одного пикселя в одном канале:

Тип	Кол-во бит на канал	Значение
byte	8	от 0 до 255
sbyte	8	от -128 до 127
float	32	от 0.0 до 1.0
double	64	от 0.0 до 1.0

Блок кода №2. Примеры создания и загрузки изображений с использованием класса Image<TColor, TDepth> Emgu.CV.

```
//Загрузка файла "image.jpg" в формате BGR
Image<Bgr, byte> image = new Image<Bgr, byte>("image.jpg");

//Создание черно-белого изображения размером 640x480
Image<Gray, byte> grayImage = new Image<Gray, byte>(640, 480);
```

Класс `Image` предоставляет свойство `Data` — это трехмерный массив `[height, width, channel]`, который позволяет читать и изменять пиксели напрямую.

Блок кода №3. Примеры получения и записи данных из пикселей изображения.

```
int x = 0;
int y = 0;

//Получение данных о синем цвете из пикселя в строке y столбце x
byte pixel1 = image.Data[y, x, 0];

//Запись данных о красном цвете в пиксель в строке y столбце x
image.Data[y, x, 0] = pixel1;

//Получение данных о единственном канале из черно-белого изображения
byte pixel2 = grayImage.Data[y, x, 0];
```

Внимание! Обращение через свойство `Data[,,]` в Emgu.CV работает медленно, так как на каждый вызов происходят несколько проверок границ массива. В реальных высокопроизводительных приложениях на OpenCV используются указатели и работа с памятью, но для обучения базовым алгоритмам этот способ подходит лучше из-за своей наглядности.

Так как библиотека Emgu.CV использует свой собственный формат для хранения изображений `Image<TColor, TDepth>` требуется использовать конвертеры из данного формата в формат стандартный для приложения WPF. В данном случае будет использоваться конвертер в формат BitmapSource который можно удобно передавать в компонент Image для отображения загруженного изображения.

Блок кода №4. Код функции-конвертера изображения из формата Image<Bgr, byte> в формат BitmapSource.

```
/*/
Функция конвертирует изображение из формата Emgu.CV (Image<Bgr, byte>) в формат, понятный для WPF (BitmapSource).
Это необходимо, чтобы мы могли отобразить результат обработки в элементе Image на нашем окне.
/*/

public BitmapSource ToBitmapSource(Image<Bgr, byte> image)
{
    //У каждого объекта Image<,> есть свойство .Mat, которое предоставляет
    //доступ к матрице пикселей.
    var mat = image.Mat;

    return BitmapSource.Create(
        mat.Width,
        mat.Height,
        96d, //Горизонтальное разрешение (DPI), 96 - стандарт для экранов
        96d, //Вертикальное разрешение (DPI)
```

```
            PixelFormats.Bgr24, //Мы указываем WPF, что данные идут в формате Bgr, по 24 бита на пиксель (8 бит на синий, 8 на зеленый, 8 на красный).
            null, //Палитра не используется для 24-битных изображений
            mat.DataPointer, //Указатель на начало данных изображения в памяти.
            mat.Step * mat.Height, //Общий размер буфера данных в байтах.
            mat.Step); // Шаг - это длина одной строки изображения в байтах.
    }
```

Добавьте в интерфейсе компонент Image и дайте ему название (в данном примере он имеет название MainImage), данный компонент будет использоваться для отображения загруженного изображения. Для загрузки изображения можно использовать следующий код:

Блок кода №5. Загрузка и отображение изображения в компоненте Image WPF

```
//Главный объект для работы с изображением в Emgu.CV. Этот обьект должен быть обьявлен глобально в проекте
//Это поле будет хранить наше оригинальное изображение для всех операций.
private Image<Bgr, byte> sourceImage;

...

public MainWindow()
{
    InitializeComponent();
    //Создаем объект Image<Bgr, byte> напрямую из пути к файлу, Emgu.CV сам его загрузит и декодирует.
    sourceImage = new Image<Bgr, byte>("image.jpg");
    //Конвертируем наше Emgu.CV изображение в понятный для WPF формат с помощью нашего конвертера.
    MainImage.Source = ToBitmapSource(sourceImage);
}

```

Для сохранения изображения можно использовать встроенную в `Image<TColor, TDepth>` функцию `Save`, которая в качестве аргумента принимает путь до файла куда требуется записать изображение:

Блок кода №6. Сохранение изображения используя встроенную в Image<TColor, TDepth> функцию

```
//У Emgu.CV есть удобный встроенный метод .Save(), в качестве аргумента передается путь до файла
imageToSave.Save("image.jpg");
```

Задача №1.

Реализуйте функционал загрузки и сохранения изображения из проводника используя классы `OpenFileDialog` и `SaveFileDialog`.

Попиксельная обработка изображений

Попиксельная обработка — это базовый уровень алгоритмов, где выходное значение пикселя зависит только от его входного значения (или значений в той же координате).

Так как в классе `Image<TColor, TDepth>` изображение представляет собой матрицу, чтобы изменить изображение, необходимо организовать вложенные циклы для прохода по всем координатам.

Блок кода №7. Цикл прохода по каждому пикселю изображения

```
//Проходим по каждому пикселю изображения.
for (int y = 0; y < sourceImage.Rows; y++)
{
    for (int x = 0; x < sourceImage.Cols; x++)
    {

        //Получаем доступ к пикселю по его координатам (y, x).
        //Emgu.CV возвращает структуру Bgr, у которой есть поля .Blue, .Green, .Red.
        Bgr pixel = sourceImage[y, x];

        //Устанавливаем значение каждого канала на 255.
        //ВАЖНО! Мы используем изображение в формате <Bgr, BYTE> - поэтому значения которые мы записываем в каналы пикселя не должны быть больше 255 и меньше 0
        pixel.Blue = 255;
        pixel.Green = 255;
        pixel.Red = 255;

        //Записываем измененный пиксель обратно в изображение.
        sourceImage[y, x] = pixel;
    }
}

//Отображаем результат в окне.
//В результате все изображение должно стать белым
MainImage.Source = ToBitmapSource(invertedImage);
```

Одним из самых простых алгоритмов изменения изображения является алгоритм инверсии цвета и реализуется по формуле:

Формула №1. Инверсия цвета.

$$pixel.Channel = 255 - pixel.Channel,$$

где pixel - текущий обрабатываемый пиксель, а Channel - значение одного из каналов заметьте что формула используется для изображений с глубиной цвета записывающихся в формате byte

Модифицируем предыдущий код для реализации данной формулы и оформим его как отдельную функцию:

Блок кода №8. Функция инверсии цветов изображения.

```
//ВАЖНО: Мы не хотим изменять оригинальное изображение (sourceImage).
//Вместо этого мы создаем его точную копию (клон) и работаем с ней.
Image<Bgr, byte> invertedImage = sourceImage.Clone();

//Проходим по каждому пикселю изображения.
for (int y = 0; y < invertedImage.Rows; y++)
{
    for (int x = 0; x < invertedImage.Cols; x++)
    {

        //Получаем доступ к пикселю по его координатам (y, x).
        //Emgu.CV возвращает структуру Bgr, у которой есть поля .Blue, .Green, .Red.
        Bgr pixel = invertedImage[y, x];

        // Инвертируем каждый цветовой канал. Максимальное значение - 255.
        pixel.Blue = 255 - pixel.Blue;
        pixel.Green = 255 - pixel.Green;
        pixel.Red = 255 - pixel.Red;

        //Записываем измененный пиксель обратно в изображение.
        invertedImage[y, x] = pixel;
    }
}

//Отображаем результат в окне.
MainImage.Source = ToBitmapSource(invertedImage);
```

Другим стандартным алгоритмом попиксельной обработки является алгоритм перевода цветного трехканального изображения в черно-белый формат. Для этого используется следующая формула:

Формула №2. Конвертация цветного изображения в черно-белое методом усреднения.

$$grayPixel = \frac{originalPixel.R + originalPixel.G + originalPixel.B}{3}$$

где grayPixel - результат в формате byte,
originalPixel - оригинальный пиксель изображения;

После чего требуется записать результат вычисления в обрабатываемый пиксель:

$$originalPixel.Red = grayPixel$$
$$originalPixel.Green = grayPixel$$
$$originalPixel.Blue = grayPixel$$

Важно! Тут мы записываем полученное значение в каждый из каналов цветного изображения чтобы привести его в черно-белый формат. Данный вариант является излишним так как дублирует ненужную информацию, более корректным вариантом является запись значений в изображение в формате `Image<gray, byte>`, но вопрос конвертации в разные типы будет разобран подробнее в следующих лабораторных.

Реализовав и проверив данную функцию в коде можно заметить что изображения получаются "плоскими" и неестественными. В действительности корректным решением задачи конвертации цветного изображения в черно-белое является использование формулы светимости Luma (Rec. 601):

Формула №3. Конвертация цветного изображения в черно-белое по формуле Luma Rec. 601.

$$grayPixel = originalPixel.Red * 0.299 + originalPixel.Green * 0.587 + originalPixel.Blue * 0.114$$

После получения значения его так же как и в предыдущем варианте требуется записать обратно в обрабатываемый пиксель.

Эти числа основаны на психофизиологии восприятия света человеческим глазом. Наши глаза содержат разные типы колбочек, и мы воспринимаем цвета с разной интенсивностью:

- Зеленый (0.587): Глаз наиболее чувствителен к зеленому спектру. Он кажется нам самым ярким. Поэтому он вносит самый большой вклад (почти 60%) в итоговую яркость пикселя.
- Красный (0.299): Воспринимается менее ярким, чем зеленый.
- Синий (0.114): Кажется нам самым темным цветом. Его вклад в общую яркость минимален (всего около 11%).

Задача №2.

Реализуйте следующие фильтры обработки изображения:

- Инверсия цвета;
- Конвертация цветного изображения в черно-белое методом усреднения;
- Конвертации цветного изображения в черно-белое по формуле Luma Rec. 601;

Базовые операции обработки изображения

Такие операции как изменение яркости, контрастности или усиление одного из цветовых каналов так же являются операциями попиксельной обработки.

Яркость — это смещение значений интенсивности пикселей. С математической точки зрения это простое прибавление значения ко всем каналам изображения.

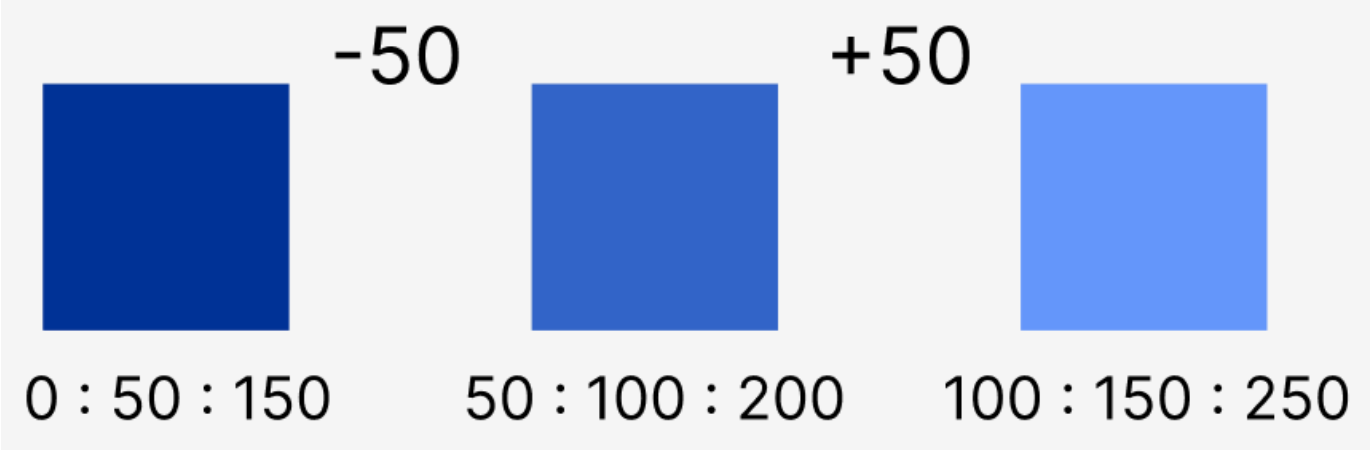

Рисунок 5. Изменение яркости пикселя.

Если прибавляем положительное число — изображение светлеет. Если отрицательное — изображение темнеет. Изменение яркости можно представить следующей формулой:

Формула №4. Изменение яркости.

$$pixel = pixel + brightness$$

где pixel - значение цветовых каналов пикселя,
brightness - добавляемое значение яркости;

Перенесем данную формулу в код:

Блок кода №9. Функция изменения яркости изображения.

```
Image<Bgr, byte> lightImage = sourceImage.Clone();

// Устанавливаем значение яркости
int brightness = 50;

for (int y = 0; y < lightImage.Rows; y++)
{
    for (int x = 0; x < lightImage.Cols; x++)
    {
        Bgr pixel = lightImage[y, x];

        //Прибавляем значение яркости к каждому каналу.
        int b = (int)pixel.Blue + brightness;
        int g = (int)pixel.Green + brightness;
        int r = (int)pixel.Red + brightness;

```

```
            //Результат сложения может выйти за пределы 0-255.
            //Поэтому результат зажимается допустимом диапазоне.
            pixel.Blue = (byte)Math.Max(0, Math.Min(255, b));
            pixel.Green = (byte)Math.Max(0, Math.Min(255, g));
            pixel.Red = (byte)Math.Max(0, Math.Min(255, r));

            lightImage[y, x] = pixel;
        }
    }

    MainImage.Source = ToBitmapSource(lightImage);
```

Контрастность — это разница между самыми светлыми и самыми темными участками изображения. Повышение контраста делает тени темнее, а света ярче. Если яркость можно представить как сложение, то контрастность - это операция умножения.

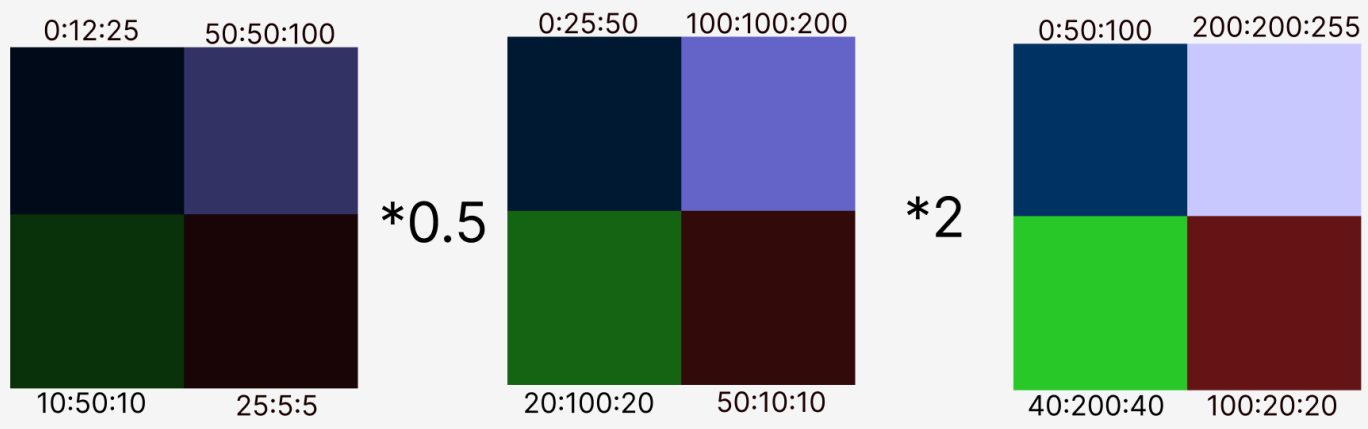

Рисунок 6. Изменение контрастности изображения.

Формула №5. Изменение контраста.

$$pixel = pixel * contrast$$

где pixel - значение цветовых каналов пикселя,
contrast - коэффициент контрастности;

Обратите внимание: данная формула является упрощенной. В следующей лабораторной работе мы разберем, почему она дает результат отличный от привычного контраста, и реализуем более правильный алгоритм.

В `Emgu.CV` изображение `Image<Bgr, byte>` состоит из трех независимых каналов. Мы можем обрабатывать их раздельно для создания цветовых фильтров. Изменяя каждый канал по отдельности позволяет является основой множества художественных фильтров, оно схоже с изменением яркости, только в данном случае значение прибавляется/отнимается только от одного канала:

Формула №6. Изменение каналов.

$$pixel.Red = pixel.Red + red$$

$$pixel.Green = pixel.Green + green$$

$$pixel.Blue = pixel.Blue + blue$$

где red, green и blue - значения переменных для изменения каждого из каналов

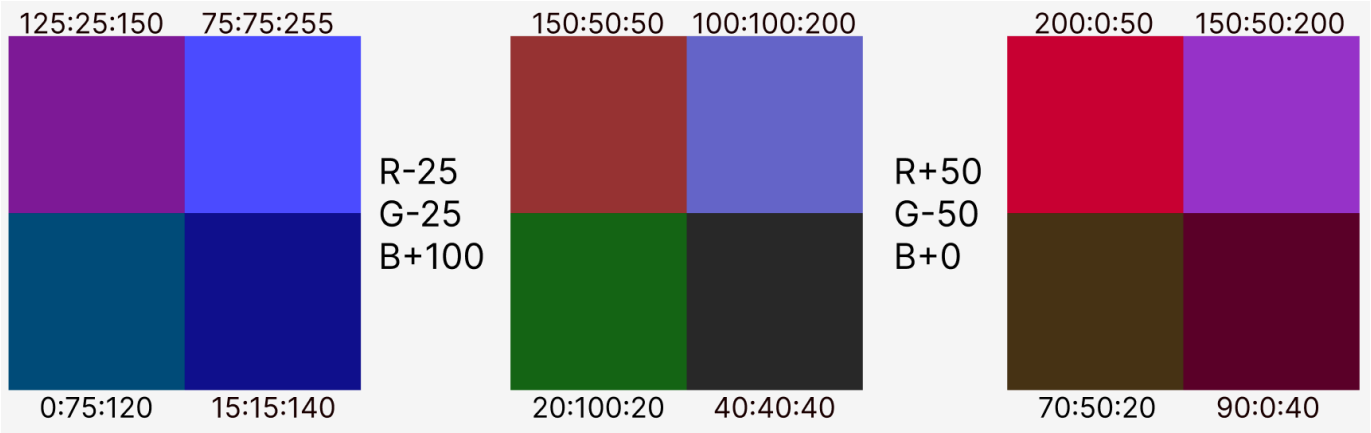

Рисунок 7. Изменение каналов изображения.

Задача №3.

Реализуйте следующие фильтры:

- Изменение яркости и контрастности изображения;
- Изменение цветовых каналов изображения.

Задание на лабораторную работу №1. Простой редактор изображения.

В данной лабораторной работе требуется разработать простой редактор изображений который включает в себя следующие функции:

1. Загрузка и сохранение изображения. Операции должны выполнятся используя стандартный проводник;
2. Если к изображению применен фильтр последующие изменения должны применятся к измененному изображению;
3. Если к изображению применен фильтр должен быть сохранен вариант изображения с примененным эффектом;
4. Функция возвращения на предыдущий шаг/применения изменений к изображению позволяющая вернуть результат до применения фильтра;
5. Фильтры инверсии цветов и два варианта конвертации изображения в черно-белый формат (используя метод усреднения и формулу Luma);
6. Фильтры изменения яркости и контрастности изображения. Для визуального интерфейса рекомендуется использовать ползунки.
7. Фильтры изменения цветовых каналов изображения. Для визуального интерфейса рекомендуется использовать ползунки.

Полезные ссылки

1. GitHub репозиторий с учебным проектом: https://github.com/TheSkyEye1/AOCI-Lab1-Simple-Image-Redactior
2. GitHub релиз проекта демонстрирующий весь функционал реализуемый в работе: https://github.com/TheSkyEye1/AOCI-Lab1-Simple-Image-Redactior/releases/tag/v1.0.0
3. Документация Emgu.CV: https://www.emgu.com/wiki/index.php?title=Main_Page
4. Работа с изображениями в Emgu.CV https://www.emgu.com/wiki/index.php/Working_with_Images

Дополнительная информация

Функция выполняющая обратную операцию по переводу изображения из формата `BitmapSource` в формат `Emgu.CV Image<Bgr, byte>` :

Блок кода №10. Функция конвертации изображения из формата BitmapSource в изображение Image<Bgr, byte>.

```
1   //Функция конвертирует изображение из формата WPF (BitmapSource) обратно в формат
    Emgu.CV (Image<Bgr, byte>).
2
3   public Image<Bgr, byte> ToEmguImage(BitmapSource source)
4   {
5       if (source == null) return null;
6
```

```
    //Чтобы гарантировать, что у нас есть данные в формате Bgr24, мы создаем "конвертер" FormatConvertedBitmap.
    FormatConvertedBitmap safeSource = new FormatConvertedBitmap();
    safeSource.BeginInit();
    safeSource.Source = source;
    safeSource.DestinationFormat = PixelFormats.Bgr24; //Явно указываем нужный нам формат
    safeSource.EndInit();

    //Создаем пустое изображение Emgu.CV нужного размера.
    Image<Bgr, byte> resultImage = new Image<Bgr, byte>(safeSource.PixelWidth, safeSource.PixelHeight);
    var mat = resultImage.Mat;

    //Копируем пиксели из WPF-изображения (safeSource) напрямую в память нашего Emgu.CV изображения (resultImage).
    safeSource.CopyPixels(
        new System.Windows.Int32Rect(0, 0, safeSource.PixelWidth, safeSource.PixelHeight), //Какую область копировать
        mat.DataPointer, //Куда копировать (в начало данных матрицы Emgu.CV)
        mat.Step * mat.Height, //Размер буфера назначения
        mat.Step); //Шаг в буфере назначения

    return resultImage;
}
```