

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
**Национальный Исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского**

Институт информационных технологий, математики и механики
Кафедра математического обеспечения и суперкомпьютерных технологий

В.Е. Турлапов
А.А. Гетманская
Е.П. Васильев
М.В. Дубровская

**Методические указания для проведения
лабораторных работ по курсу
«КОМПЬЮТЕРНАЯ ГРАФИКА»**

Учебное пособие

Рекомендовано методической комиссией ИИТММ для студентов ННГУ,
обучающихся по направлениям подготовки
02.03.02 «Фундаментальная информатика и информационные технологии»,
01.03.02 «Прикладная математика и информатика»
00.00.00 «Программная инженерия»

Нижний Новгород
2018

УДК
ББК

Турлапов В.Е., Гетманская А.А., Васильев Е.П., Дубровская М.В.
Методические указания для проведения лабораторных работ по курсу
"КОМПЬЮТЕРНАЯ ГРАФИКА": Учебное пособие. – Нижний Новгород,
Национальный Исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского, 2018 – 96 с.

Рецензент: профессор **Н.Ю. Золотых**

Учебное пособие посвящено вопросам освоения основ компьютерной графики. Приведены точечные, матричные и морфологические фильтры для фильтрации изображений. Рассмотрен метод визуализации данных компьютерной томографии при помощи технологии OpenGL. Описан алгоритм и структуры данных, позволяющие реализовать трассировку лучей с применением графических процессоров.

Для выполнения лабораторных работ рекомендуется использовать программное обеспечение Visual Studio 2010 или более поздние версии.

Учебное пособие предназначено для студентов младших курсов Института ИТММ в качестве пособия при подготовке и проведении лабораторных работ по курсу «Компьютерная графика».

УДК
ББК

© Национальный
Исследовательский Нижегородский
государственный университет им. Н.И.
Лобачевского, 2018

Оглавление

Введение.....	4
Лабораторная работа №1. Обработка изображений	5
Лабораторная работа №2. "Визуализация томограмм".....	22
Лабораторная работа №3. «Трассировка лучей»	34
Лабораторная работа №4. "Стеганография".....	54
Список литературы	95

Введение

Значительную долю полученной информации об окружающем мире люди получают при помощи зрительного восприятия, поэтому предмет "Компьютерная графика" является необходимой частью образовательной программы.

Компьютерная графика - область науки и технологий, изучающая создание, способы хранения, обработки и распознавания изображений, 3D научной визуализации и визуализации 3D-сцен виртуальной реальности на компьютере. Можно выделить 2D графику и 3D графику. Под 2D графикой понимается работа с двумерными изображениями или последовательностями изображений, а под 3D графикой подразумевается построение двумерного или стерео- изображения 3D-сцены по ее математическому описанию.

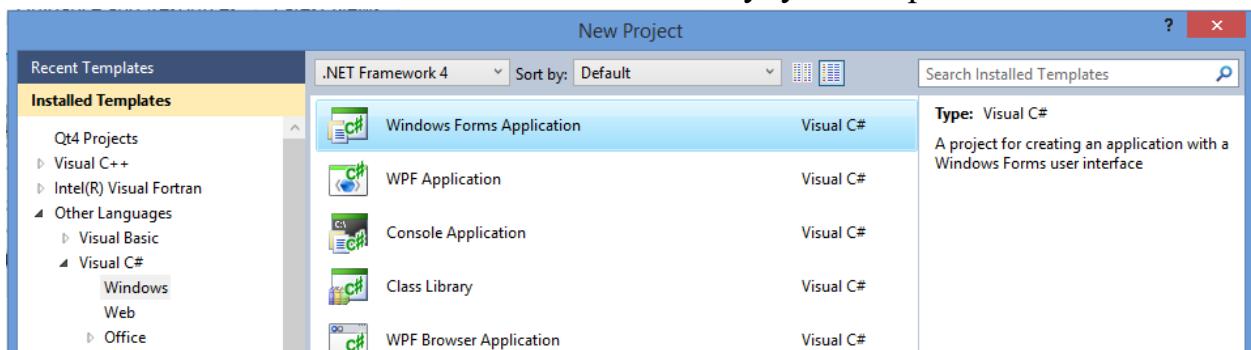
В данном пособии предложены три лабораторные работы, связанные с 2D и 3D графикой и с использованием графических ускорителей. Лабораторные работы представлены в формате пошаговых описаний, позволяющих самостоятельно с нуля создать законченные приложения на языке C#. Для сдачи лабораторной работы необходимо изучить материал лекций по теме лабораторной работы, выполнить основные задания, показать преподавателю, и выполнить дополнительные задания, которые даст преподаватель.

Лабораторная работа №1. Обработка изображений

Лабораторные работы выполнены в среде разработки VisualStudio. Рекомендуется использовать VisualStudio 2010 или более поздние версии. Первая лабораторная работа посвящена базовым принципам обработки изображений, рассматриваются алгоритмы подавление и устранение шума в черно-белых и цветных изображениях, выделение границ, краев, однородных зон, улучшение качества изображения, спецэффекты.

1. Создание проекта

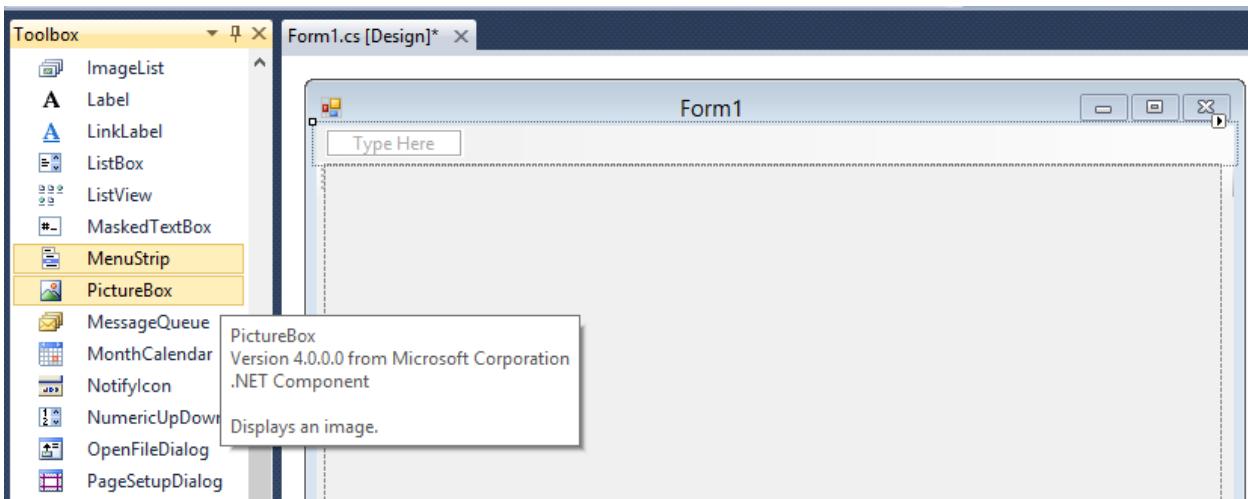
Чтобы создать новый проект, выберите File ->New ->Project, или нажмите Ctrl+Shift+N. В открывшемся окне выберите шаблон WindowsFormsApplication. В нижней части окна введите имя для вашего будущего проекта.



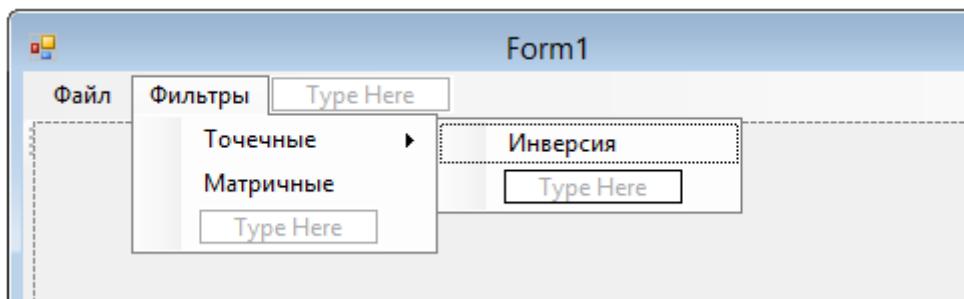
Нажмите F5. На экране должно открыться пустое окно вашей программы.

2. Добавление графических элементов на форму

Чтобы добавить графические элементы на форму, откройте SolutionExplorer (Ctrl+Alt+L). В открывшемся окне выберите файл, содержащий код вашей формы (по умолчанию Form1.cs), по щелчку ПКМ (правой клавиши мыши) выберите пункт ViewDesigner (или нажмите Shift+F7), откроется окно с формой. Потяните за правый нижний маркер, чтобы увеличить размеры формы. Нажмите Ctrl+Alt+X, чтобы открыть панель Toolbox. С Toolbox на форму перетащите элементы PictureBox и ToolStrip, они появятся на форме. ToolStrip автоматически займет место под заголовком формы, а PictureBox появится на том месте, куда вы его перетащили, растяните его до размеров самой формы. Окно примет следующий вид:



Щелкните ЛКМ (левой клавишей мыши) по панели **MenuStrip**, в появившемся текстовом поле введите строку «Файл». После этого появится возможность создать вложенное текстовое поле, впишите строку «Открыть». По аналогии сделайте главный пункт меню «Фильтры», а вложенными элементами «Точечные» и «Матричные». В точечные фильтры аналогично добавьте пункт «Инверсия». В результате получится такая иерархия.



3. Загрузка изображения в программу

Откройте исходный код формы (На форме ПКМ ->ViewCode или Ctrl+Alt+0). Найдите место, где начинается код нашей формы Form1, и создайте объект Bitmap.

```
public partial class Form1 : Form
{
    Bitmap image;
```

Возвращаемся к графическому представлению формы, и делаем двойной щелчок по элементу меню «Открыть», у нас автоматически создастся функция *открытьToolStripMenuItem_Click*, в которую мы будем добавлять код.

```
private void открытьToolStripMenuItem_Click(object sender, EventArgs e)
{
}
```

Создайте объект типа OpenFileDialog и инициализируйте его конструктором по умолчанию (конструктором без параметров):

```
private void открытьToolStripMenuItem_Click(object sender, EventArgs e)
{
    // создаем диалог для открытия файла
    OpenFileDialog dialog = new OpenFileDialog();
```

Для удобства открытия только изображений, чтобы в окне проводника не было видно других файлов, добавьте фильтр:

```
 OpenFileDialog dialog = new OpenFileDialog();
dialog.Filter = "Image files (*.png; *.jpg; *.bmp)|All files (*.*)|*.*";
```

Проверить, выбрал ли пользователь файл, можно с помощью следующего условия:

```
if (dialog.ShowDialog() == DialogResult.OK)
{
}
```

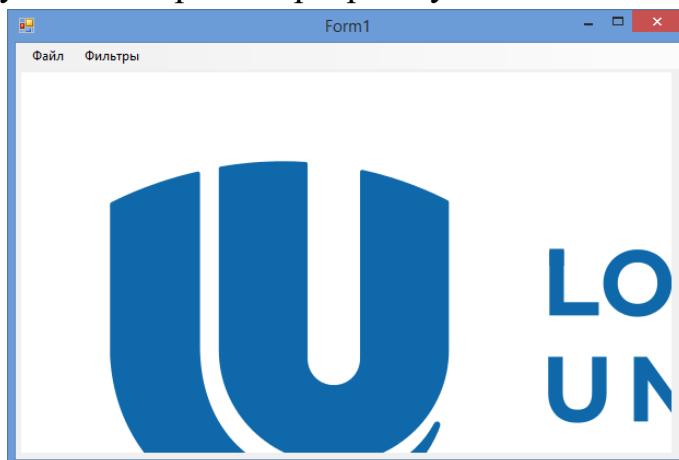
В случае выполнения данного условия инициализируйте Вашу переменную image выбранным изображением. Для этого воспользуйтесь конструкцией ниже.

```
image = new Bitmap(dialog.FileName);
```

После того, как вы загрузили картинку в программу, необходимо ее визуализировать на форме, для этого свойству pictureBox.Image присвойте внутри тех же фигурных скобок значение image и обновите ваш pictureBox:

```
pictureBox1.Image = image;
pictureBox1.Refresh();
```

Проверьте, что ни одна из строчек кода выше не пропущена. Нажмите F5, чтобы запустить программу. Выберите картинку и посмотрите на получившийся результат. Закройте программу.



В данном случае размеры изображения больше размера окна, и оно полностью не входит. Чтобы изменить вариант отображения, откройте свойства PictureBox. Для этого вернитесь в дизайнер (ПКМ нажмите на pictureBox1, выберите View Designer) и ПКМ откройте меню, в котором выберите Properties. В появившемся окне с аналогичным названием параметруSizeMode установите

значение Zoom. Снова запустите программу. Поэкспериментируйте с другими значениями этого параметра.



4. Создание класса для фильтров и фильтра «Инверсия»

Каждый фильтр будем представлять в коде отдельным классом. С другой стороны, все фильтры будут иметь абсолютно одинаковую функцию, запускающую процесс обработки и перебирающую в цикле все пиксели результирующего изображения. Исходя из этих соображений, создадим родительский абстрактный класс Filters, который и будет содержать эту функцию. Для этого создайте новый файл. Откройте окно SolutionExplorer (Ctrl+Alt+L), нажмите ПКМ по имени вашего проекта, выберите Add ->Class. В открывшемся окне введите имя класса – Filters. В SolutionExplorer появится файл Filters.cs, двойным щелчком ПКМ откройте файл для редактирования. Чтобы использовать классы для работы с изображениями, входящие в состав библиотеки базовых классов (BCL), в раздел объявлений зависимостей добавьте строку using System.Drawing. Сделайте класс Filters абстрактным, добавив модификатор abstract. Код пустого абстрактного класса должен выглядеть так:

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

namespace Filters_Ivanov
{
    abstract class Filters
    {
    }
}
```

В классе Filters создайте функцию processImage, принимающую на вход объект типа Bitmap и возвращающую объект типа Bitmap. В этой функции будет находиться общая для всех фильтров часть.

```

public Bitmap processImage(Bitmap sourceImage)
{
    Bitmap resultImage = new Bitmap(sourceImage.Width, sourceImage.Height);

    return resultImage;
}

```

На данный момент функция создает пустое изображение такого же размера, как и полученное ею на входе. Чтобы обойти все пиксели изображения, создайте два вложенных цикла: от 0 до ширины и от 0 до высоты изображения. Внутри цикла с помощью метода SetPixel установите пикселью с текущими координатами значение функции calculateNewPixelColor.

```

Bitmap resultImage = new Bitmap(sourceImage.Width, sourceImage.Height);
for (int i = 0; i < sourceImage.Width; i++)
{
    for (int j = 0; j < sourceImage.Height; j++)
    {
        resultImage.SetPixel(i, j, calculateNewPixelColor(sourceImage, i, j));
    }
}
return resultImage;

```

VisualStudio подчеркивает имя calculateNewPixelColor, потому что она еще не создана. Создайте эту функцию в классе Filters и сделайте абстрактной. Данная функция будет вычислять значение пикселя отфильтрованного изображения, и для каждого из фильтров будет уникальной.

```

abstract class Filters
{
    protected abstract Color calculateNewPixelColor(Bitmap sourceImage, int x, int y);

    public Bitmap processImage(Bitmap sourceImage)

```

Цвет пикселя в модуле System.Drawing библиотеки базовых классов (BCL) представляется тремя (если не считать прозрачность) компонентами, каждая из которых может принимать значение от 0 до 255. В некоторых фильтрах результат может выходить за эти рамки, что приведет к падению программы. В классе Filters напишите функцию Clamp, чтобы привести значения к допустимому диапазону.

```

public int Clamp(int value, int min, int max)
{
    if (value < min)
        return min;
    if (value > max)
        return max;
    return value;
}

```

Создайте класс InvertFilter, наследник класса Filters, с переопределенной функцией calculateNewPixelColor.

```

protected override Color calculateNewPixelColor(Bitmap sourceImage, int x, int y)
{
}

```

В теле функции получите цвет исходного пикселя, а затем вычислите инверсию этого цвета, и верните как результат работы функции.

```
protected override Color calculateNewPixelColor(Bitmap sourceImage, int x, int y)
{
    Color sourceColor = sourceImage.GetPixel(x, y);
    Color resultColor = Color.FromArgb(255 - sourceColor.R,
                                      255 - sourceColor.G,
                                      255 - sourceColor.B);
    return resultColor;
}
```

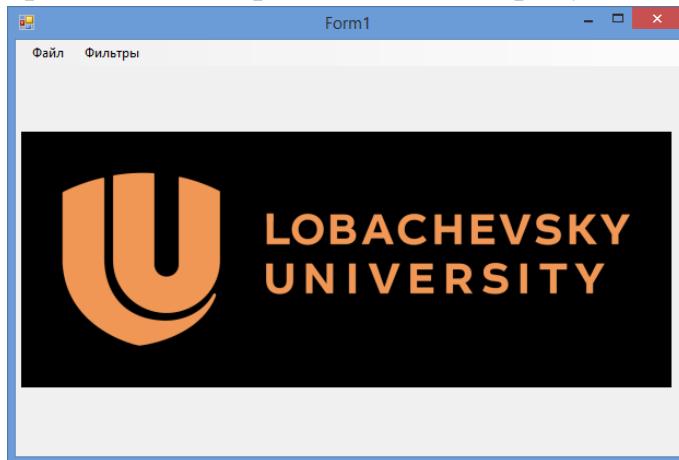
Откройте графический редактор формы. Сделайте двойной щелчок ЛКМ по элементу меню «Инверсия». Создастся новая функция, которая будет вызываться при выборе элемента «Инверсия».

```
private void инверсияToolStripMenuItem_Click(object sender, EventArgs e)
{
}
```

Создайте новый объект класса InvertFilter и инициализируйте его значением по умолчанию. Создайте новый экземпляр класса Bitmap для измененного фильтром изображения, и присвойте этому экземпляру результат функции processImage().

```
InvertFilter filter = new InvertFilter();
Bitmap resultImage = filter.processImage(image);
pictureBox1.Image = resultImage;
pictureBox1.Refresh();
```

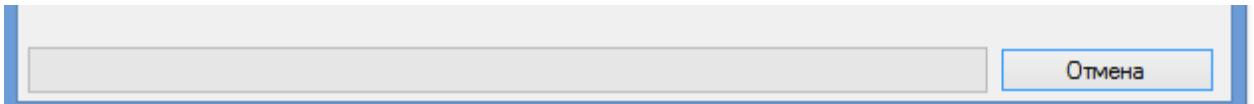
Запустите программу, проверьте работоспособность фильтра. Должно получиться инвертированное изображение, как на рисунке ниже:



5. Создание индикатора прогресса

Некоторые фильтры работают сравнительно долгое время, и желательно знать прогресс выполнения и иметь возможность безболезненного прекращения операции. Воспользуемся компонентами ProgressBar и BackgroundWorker для реализации этой функциональности. Для этого откройте ToolBox (открывать панель следует при открытом окне дизайнера, для того чтобы были видны его элементы) и перетащите на форму элементы

BackgroundWorker, ProgressBar и Button, измените надпись на кнопке на «Отмена». Значок BackgroundWorker появится под окном формы, т.к. компонент не относится к пользовательскому интерфейсу и не является видимым.

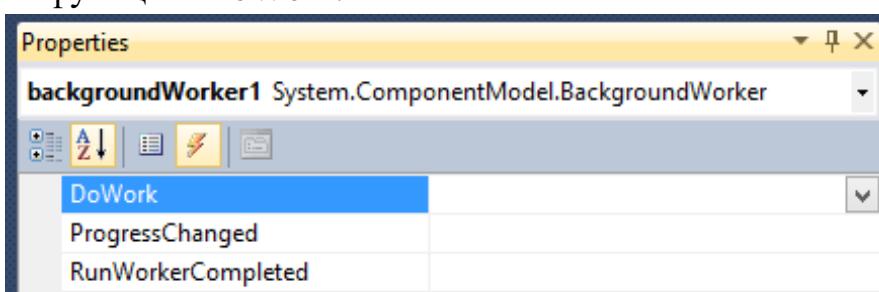


BackgroundWorker – (модуль System.ComponentModel.BackgroundWorker) – это класс, предназначенный для создания и управления работой потоков. Он предоставляет следующие возможности:

- Стандартизованный протокол сигнализации о ходе создания, выполнения и завершения потока,
- Возможность прерывания потока,
- Возможность обработки исключений в фоновом потоке,
- Возможность связи с основным потоком через сигнализацию о ходе выполнения и окончания.

Таким образом, при использовании BackgroundWorker нет необходимости включения try/catch в рабочий поток и есть возможность выдачи информации в основной поток без явного вызова Control.Invoke.

Откройте свойства BackgroundWorker, установите параметру WorkerReportProgress значение True, параметру WorkerSupportsCancellation тоже значение True, переключитесь на вкладку Events (в этом же окне свойств), на которой расположены доступные события для элемента. Двойным щелчком ЛКМ создайте функцию DoWork.



Добавьте в функцию код, который будет выполнять код одного из фильтров.

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    Bitmap newImage = ((Filters)e.Argument).processImage(image, backgroundWorker1);
    if (backgroundWorker1.CancellationPending != true)
        image = newImage;
}
```

Подключите зависимость System.ComponentModel и измените объявление функции processImage() в классе Filters.

```
public Bitmap processImage(Bitmap sourceImage, BackgroundWorker worker)
```

В функции processImage во внешний цикл добавьте строку, которая будет сигнализировать элементу BackgroundWorker о текущем прогрессе. Используйте приведения типов для корректных расчетов.

```
for (int i = 0; i < sourceImage.Width; i++)
{
    worker.ReportProgress((int)((float)i / resultImage.Width * 100));
    if (worker.CancellationPending)
        return null;
```

Аналогично созданию функции DoWork создайте функцию ProgressChanged. Добавьте строку кода, которая будет изменять цвет полосы.

```
private void backgroundWorker1_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}
```

Создайте функцию, которая будет визуализировать обработанное изображение на форме и обнулять полосу прогресса.

```
private void backgroundWorker1_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        pictureBox1.Image = image;
        pictureBox1.Refresh();
    }
    progressBar1.Value = 0;
}
```

Измените функцию вызова фильтра инверсии, чтобы фильтр запускался в отдельном потоке.

```
private void инверсияToolStripMenuItem_Click(object sender, EventArgs e)
{
    Filters filter = new InvertFilter();
    backgroundWorker1.RunWorkerAsync(filter);
}
```

Сделайте двойной клик ЛКМ по кнопке «Отмена» для создания функции, выполняемой по нажатию кнопки. Используйте функцию CancelAsync класса BackgroundWorker, чтобы остановить выполнение фильтра.

```
private void button1_Click(object sender, EventArgs e)
{
    backgroundWorker1.CancelAsync();
}
```

6. Матричные фильтры

Главная часть матричного фильтра – ядро. Ядро – это матрица коэффициентов, которая покомпонентно умножается на значение пикселей изображения для получения требуемого результата (это НЕ то же самое, что матричное умножение, коэффициенты матрицы являются весовыми коэффициентами для выбранного подмассива изображения).

Создайте класс MatrixFilter, содержащий в себе двумерный массив kernel.

```

class MatrixFilter : Filters
{
    protected float[,] kernel = null;
    protected MatrixFilter() { }
    public MatrixFilter(float[,] kernel)
    {
        this.kernel = kernel;
    }
}

```

Создайте функцию calculateNewPixelColor, которая будет вычислять цвет пикселя на основании своих соседей. Первым делом найдите радиусы фильтра по ширине и по высоте на основании матрицы. Переменные x и y – это поступающие на вход координаты текущего пикселя изображения, цвет которого мы собираемся считать.

```

protected override Color calculateNewPixelColor(Bitmap sourceImage, int x, int y)
{
    int radiusX = kernel.GetLength(0) / 2;
    int radiusY = kernel.GetLength(1) / 2;

```

Создайте переменные типа float, в которых будут храниться цветовые компоненты результирующего цвета. Создайте два вложенных цикла, которые будут перебирать окрестность пикселя. В каждой из точек окрестности вычислите цвет, умножьте на значение из ядра и прибавьте к результирующим компонентам цвета. Чтобы на граничных пикселях не выйти за границы изображения, используйте функцию Clamp.

```

float resultR = 0;
float resultG = 0;
float resultB = 0;
for (int l = -radiusY; l <= radiusY; l++)
    for (int k = -radiusX; k <= radiusX; k++)
    {
        int idX = Clamp(x + k, 0, sourceImage.Width - 1);
        int idY = Clamp(y + l, 0, sourceImage.Height - 1);
        Color neighborColor = sourceImage.GetPixel(idX, idY);
        resultR += neighborColor.R * kernel[k + radiusX, l + radiusY];
        resultG += neighborColor.G * kernel[k + radiusX, l + radiusY];
        resultB += neighborColor.B * kernel[k + radiusX, l + radiusY];
    }
}

```

Тщательно разберите приведенный выше код. Переменные x и y, как было сказано выше, – это координаты текущего пикселя. Переменные l и k принимают значения от -radius до radius и означают положение элемента в матрице фильтра (ядре), если начало отсчета поместить в центр матрицы. В переменных idX и idY хранятся координаты пикселей-соседей пикселя (x,y), с которым совмещается центр матрицы, и для которого происходит вычисления цвета (т.е. это координаты окрестности пикселя (x,y), в которую входят и его собственные координаты). Предполагаем, что ядро фильтра, поступающее на вход, уже является нормированным (т.е. сумма его коэффициентов не выходит за пределы [0, 1]), что нужно, чтобы не выйти за допустимые границы интенсивности изображения и при этом не потерять часть информации после обрезки результата функцией Clamp.

В качестве результата работы функции создайте экземпляр класса Color, состоящий из вычисленных вами компонент цвета. Используйте функцию Clamp, чтобы все значения компонент были в допустимом диапазоне.

```
return Color.FromArgb(  
    Clamp((int)resultR, 0, 255),  
    Clamp((int)resultG, 0, 255),  
    Clamp((int)resultB, 0, 255)  
>;
```

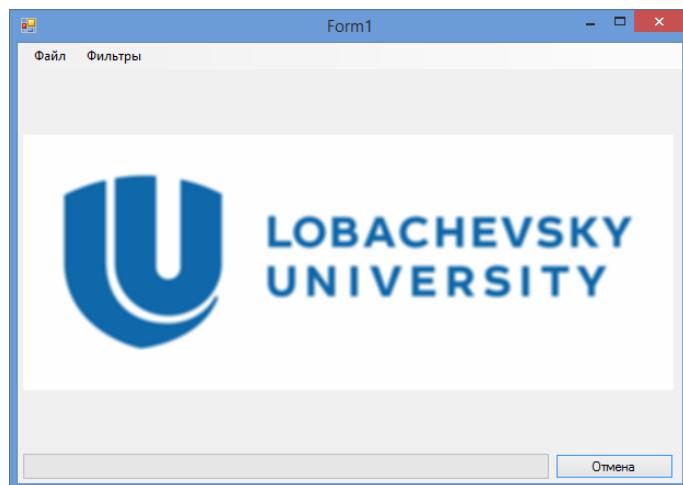
Создайте класс BlurFilter – наследник класса MatrixFilter. Переопределите конструктор по умолчанию, в котором создайте матрицу 3*3 со значением 1/9 в каждой ячейке.

```
class BlurFilter : MatrixFilter  
{  
    public BlurFilter()  
    {  
        int sizeX = 3;  
        int sizeY = 3;  
        kernel = new float[sizeX, sizeY];  
        for (int i = 0; i < sizeX; i++)  
            for (int j = 0; j < sizeY; j++)  
                kernel[i, j] = 1.0f / (float)(sizeX * sizeY);  
  
    }  
}
```

На форме в панели матричных фильтров добавьте элемент «Размытие», создайте двойным щелчком функцию для ее применения.

```
private void размытиеToolStripMenuItem_Click(object sender, EventArgs e)  
{  
    Filters filter = new BlurFilter();  
    backgroundWorker1.RunWorkerAsync(filter);  
}
```

Проверьте результат работы. Для проверки результата не берите изображение большого разрешения, потому что матричные фильтры работают намного дольше точечных. Кроме того, при использовании большого изображения оно будет уменьшаться до размеров pictureBox из-за чего эффект размытия будет менее заметен.



Более совершенным фильтром для размытия изображений является фильтр Гаусса, коэффициенты для которого в одномерном случае рассчитываются по формуле:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Здесь σ – среднеквадратическое отклонение, μ – математическое ожидание.

Фильтр Гаусса является сепарабельным, поэтому для обработки двумерного сигнала (изображения) можно записать: $f(i,j)=f(i)\cdot f(j)$. Интеграл от плотности вероятности для распределения Гаусса на интервале $[-3\sigma, 3\sigma]$, д.б. равен единице. → Нормируем коэффициенты матрицы на их сумму.

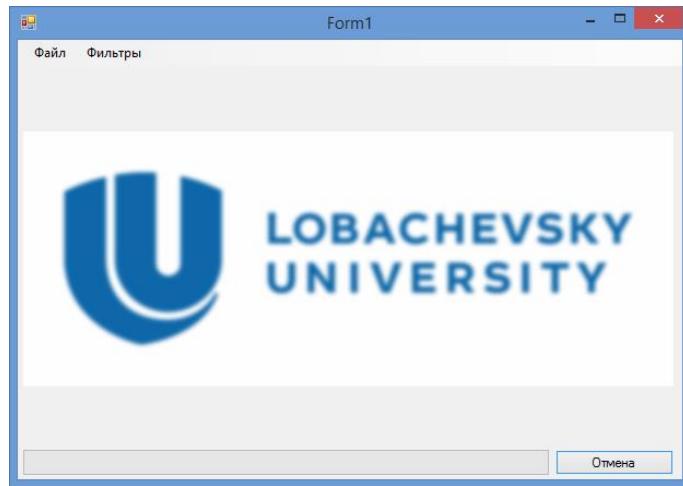
Создайте новый класс GaussianFilter – наследник класса MatrixFilter. Создайте функцию, которая будет рассчитывать ядро преобразования по формуле Гаусса:

```
public void createGaussianKernel(int radius, float sigma)
{
    // определяем размер ядра
    int size = 2 * radius + 1;
    // создаем ядро фильтра
    kernel = new float[size, size];
    // коэффициент нормировки ядра
    float norm = 0;
    // рассчитываем ядро линейного фильтра
    for (int i = -radius; i <= radius; i++)
        for (int j = -radius; j <= radius; j++)
    {
        kernel[i + radius, j + radius] = (float)(Math.Exp(-(i * i + j * j) / (2 * sigma * sigma)));
        norm += kernel[i + radius, j + radius];
    }
    // нормируем ядро
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            kernel[i, j] /= norm;
}
```

Создайте конструктор по умолчанию, который будет раздавать фильтр размером 7*7 и с коэффициентом сигма, равным 2.

```
public GaussianFilter()
{
    createGaussianKernel(3, 2);
}
```

Аналогично остальным фильтрам допишите код для запуска фильтра. Протестируйте.



7. Задания для самостоятельного выполнения

Применив полученные знания, добавьте в программу следующие фильтры:

- Создайте точечный фильтр, переводящий изображение из цветного в полуточновое (GrayScale, в оттенках серого). Для этого создайте фильтр GrayScaleFilter – наследник класса Filters, и создайте функцию calculateNewPixelColor, которая переводит цветное изображение в полуточновое по следующей формуле:

$$Intensity = 0.299 * R + 0.587 * G + 0.114 * B$$

Полученное значение записывается во все три канала выходного пикселя.

- Создайте точечный фильтр «Сепия», переводящий цветное изображение в изображение песочно-коричневых оттенков. Для этого вначале найдите интенсивность, как для полуточнового изображения. Цвет выходного пикселя задайте по формуле:

$$R = Intensity + 2 * k; G = Intensity + 0.5 * k; B = Intensity - 1 * k$$

Подберите коэффициент k для наиболее оптимального на Ваш взгляд оттенка сепии, не забудьте при написании фильтра использовать функцию Clamp для приведения всех значений к допустимому интервалу.

- Создайте точечный фильтр, увеличивающий яркость изображения. Для этого в каждый канал пикселя прибавьте константу, позаботьтесь о допустимости значений.
- Создайте матричный фильтр Собеля. Оператор Собеля представляет собой матрицу 3×3 . Оператор Собеля вычисляет приближенное значение градиента яркости изображения. Ниже представлены операторы Собеля, ориентированные по разным осям:

$$\text{По оси Y: } \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \quad \text{По оси X: } \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

- Создайте матричный фильтр, повышающий резкость изображения. Матрица для данного фильтра задается следующим образом:

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

8. Список фильтров

Тиснение

Ядро фильтра: $\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ + сдвиг по яркости + нормировка.



Вначале изображение переводится в полуточновое, затем к нему применяется указанная маска. После применения фильтра максимальный диапазон плотностей составляет [-255, 255], но все значения должны удовлетворять допустимому пределу интенсивностей [0, 255], поэтому прибавляем к полученному результату 255, чтобы все значения оказались неотрицательными. Возможный диапазон составит [0, 510]. Теперь нормируем его к [0, 255]: делим все значения на 2.

Перенос/Поворот



Перенос: $\begin{cases} x(k, l) = k + 50; \\ y(k, l) = l; \end{cases}$

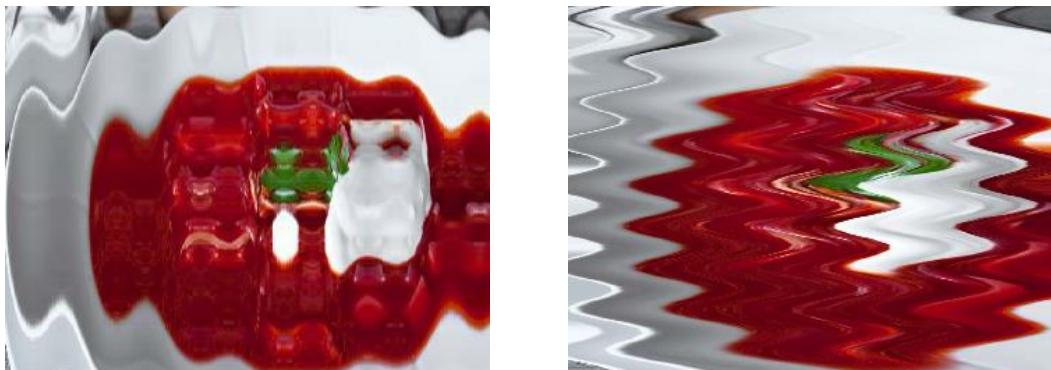
k, l – индексы результирующего изображения, для которых вычисляется цвет
 x, y – индексы по которым берется цвет из исходного изображения



Поворот:

$$\begin{cases} x(k, l) = (k - x_0)\cos(\mu) - (l - y_0)\sin(\mu) + x_0; \\ y(k, l) = (k - x_0)\sin(\mu) + (l - y_0)\cos(\mu) + y_0; \end{cases}, \text{ где } (x_0, y_0) - \text{центр поворота, } \mu - \text{угол поворота}$$

«Волны»



Волны 1: $\begin{cases} x(k, l) = k + 20\sin(2\pi l / 60); \\ y(k, l) = l; \end{cases}$

Волны 2: $\begin{cases} x(k, l) = k + 20\sin(2\pi k / 30); \\ y(k, l) = l; \end{cases}$

Эффект «стекла»



$$\begin{cases} x(k, l) = k + (\text{rand}(1) - 0.5) * 10; \\ y(k, l) = l + (\text{rand}(1) - 0.5) * 10; \end{cases}$$

Motion blur



Ядро фильтра: $\frac{1}{n} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & \\ \vdots & & & \ddots & \vdots \\ & & & & 1 & 0 & 0 \\ 0 & & \cdots & 0 & 1 & 0 \\ 0 & 0 & & & & 1 \end{bmatrix}, n -$

количество единиц, т. е. количество столбцов или строк

Резкость



Ядро фильтра: $\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$

Выделение границ

Оператор Щарра:

По оси Y: $\begin{pmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{pmatrix}$ По оси X: $\begin{pmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{pmatrix}$

Оператор Прюитта:

По оси Y: $\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ По оси X: $\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$

Светящиеся края

Медианный фильтр + выделение контуров + фильтр «максимума»



Вначале нужно применить ко всему изображению медианный фильтр. Это позволит убрать излишне мелкие детали. Затем к результату применяется какой-либо фильтр выделения краёв, чтобы получить наиболее значимые границы на изображении. К полученной карте краёв применяем фильтр максимумов по каждому из каналов. Принцип фильтра схож с принципом медианного фильтра, поскольку основан на сортировке. В отличие от медианного, в нём из окрестности пикселя выбирается не средняя в списке интенсивность из усреднённых по всем каналам, а максимальная, по каждому каналу отдельно. После применения последнего фильтра полученные ранее границы будут выглядеть ярче и станут более утолщёнными.

9. Дополнительные задания

Для развития навыков программирования на языке C# в среде VisualStudio, предлагается расширить программу дополнительной функциональностью.

- Реализовать возможность сохранять изображения. Для сохранения файлов существует класс SaveFileDialog, который работает аналогично рассмотренному в работе OpenFileDialog. Дополнительную информацию по его использованию можно найти на сайте Microsoft [3].
- Реализовать возможность изменения размера окна, при которой элементы будут перемещаться или растягиваться пропорционально изменению размера окна.
- Реализовать отмену последнего или нескольких последних действий.

10. Задания для сдачи лабораторной работы

Замечание: для фильтров, требующих последовательные преобразования нескольких промежуточных изображений в процессе получения итогового, можно переопределить функцию processImage, в которой будет организована передача промежуточных изображений от одних обработок фильтрами к другим.

- Реализуйте один из фильтров: «Серый мир», «Идеальный отражатель», «Коррекция с опорным цветом»;
- Реализуйте линейное растяжение гистограммы;
- Выберите и реализуйте по 2 точечных фильтра: из раздела «Список фильтров», где операции производятся над индексами пикселей, и из раздела матричных фильтров;

- Реализуйте операции математической морфологии dilation, erosion, opening, closing. Выберите и реализуйте одну из top hat, black hat, grad;
- Реализуйте медианный фильтр;
- Добавьте возможность задать и изменить структурный элемент для операций матморфологии.

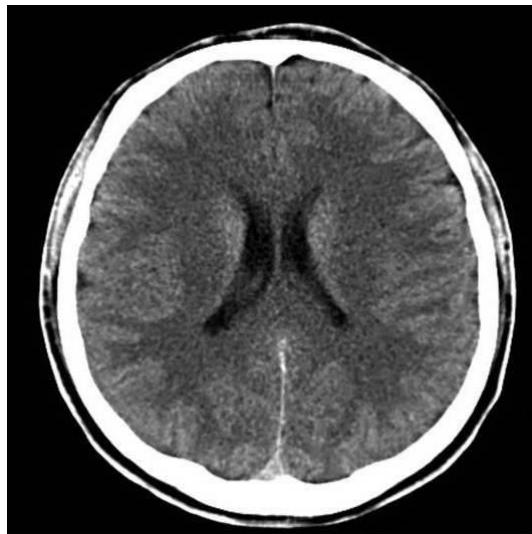
11. Ссылки

1. http://www.graph.unn.ru/rus/materials/CG/CG03_ImageProcessing.pdf - Курс компьютерной графики. Обработка изображений, часть 1.
2. http://www.graph.unn.ru/rus/materials/CG/CG04_ImageProcessing2.pdf - Курс компьютерной графики. Обработка изображений, часть 2.
3. <https://msdn.microsoft.com/ru-ru/library/sfezx97z%28v=vs.110%29.aspx> - Практическое руководство. Сохранение файлов с помощью компонента SaveFileDialog

Лабораторная работа №2. "Визуализация томограмм"

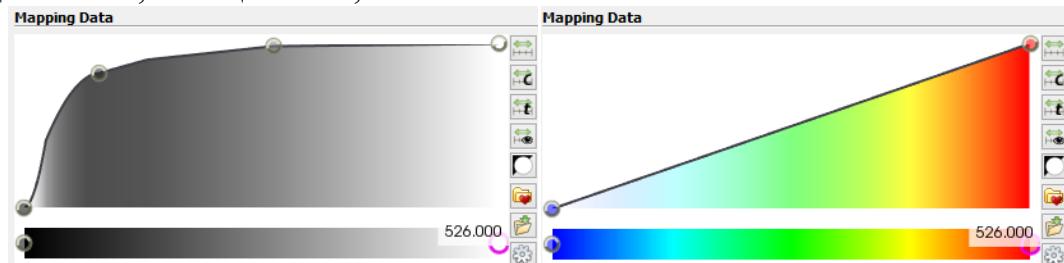
Томография - это получение послойного изображения внутренней структуры объекта.

Чаще всего, но далеко не всегда, объектом томографического исследования являются живые ткани. На рисунке ниже представлен слой томограммы головы.



Данные томографии представляют собой трехмерный массив вокселов - элементов трехмерной регулярной сетки. Каждый воксель содержит одно значение плотности, как правило, типа short или ushort.

Для перевода значения плотности в цвет используется передаточная функция = Transfer Function (TF). Transfer Function может быть серой, от черного до белого, или цветной, линейной или нелинейной.



В данной лабораторной работе будет использоваться линейная TF от черного к белому, так как ее очень просто создать, все значения рассчитываются по формуле:

$$intensity = \frac{x - min}{max - min} * 255$$

1. О графической библиотеке OpenTK

Библиотека Open Toolkit является быстрой низкоуровневой оберткой для языка C# технологий OpenGL, OpenGL ES и OpenAL. Она работает на всех основных платформах и используется в приложениях, играх, научных исследованиях.

Используйте OpenTK, чтобы добавить кросс-платформенные 3D-графику, аудио, вычисления на GPU и тактильные взаимодействия к приложению C#. Интегрируйте в существующий пользовательский интерфейс без лишних зависимостей.

Где скачать

Стабильную версию 1.1 можно скачать с сайта SourceForge:

<https://sourceforge.net/projects/opentk/>

Для новых версий Visual Studio, поддерживающих использование NuGet пакетов, можно скачать соответствующий NuGet пакет.

Установка

Для установки с помощью инсталлятора не требуется права администратора. По умолчанию библиотека устанавливается в папку "Мои документы", в инсталляторе кроме самой библиотеки также содержатся исходный код большого количества примеров.

Для установки с помощью NuGet: откройте вкладку меню Tools -> NuGet Package Manager -> Package Manager Console, и введите команды

Install-Package OpenTK и Install-Package OpenTK.GLControl. Файлы dll скопируются в папку с проектом.

Состав OpenTK

Подключение библиотеки к проекту происходит с помощью пространств имен OpenTK

```
using namespace OpenTK;
using namespace OpenTK.Graphics.OpenGL;
```

В пространстве имен OpenTK описаны:

- Вектора (*Vector2*, *Vector3*, *Vector4*) и функции для работы с ними;
- Матрицы (*Matrix2*, *Matrix2x3*, *Matrix3* и т.д.) и функции для работы с ними;
- Структуры для кривых Безье (*BezierCurve*, *BezierCurveCubic*, *BezierCurveQuadratic*);
- Прикладные функции и константы (*Factorial*, *Swap*, *Pi*, *DegreesToRadians* и т.д.);
- Классы для работы с устройствами вывода и т.д.

В пространстве имен OpenTK.Graphics.OpenGL описаны:

- Функции OpenGL (Функции вида *<glFunction>* в OpenTK выглядят *<GL.Function>*, например *glColor3f -> GL.Color3*);
- Перечисления *EnableCap* для функции *Enable(GL_CULL_FACE -> EnableCap.CullFace)*;
- Перечисления *PrimitiveType* для функции *Begin (GL_TRIANGLES -> PrimitiveType.Triangles)* и т.д.

Подробнее посмотреть можно в окне Object Browser (Ctrl+Alt+J).

Добавление виджета в проект Windows Forms.

Если вы хотите использовать виджет - окно для визуализации OpenGL аналогично PictureBox, необходимо сделать следующее:

- 1) Откройте конструктор формы Designer (Shift+F7);
- 2) Откройте панель инструментов ToolBox (Ctrl+Alt+X);
- 3) Правой кнопкой мыши вызовите контекстное меню, выберите Choose Items, в открывшемся окне выберите Browse..., выберите OpenTK.GLControl.dll, и когда GLControl появится в списке доступных элементов, установите галочку рядом с ним и нажмите "Ok". На панели ToolBox в разделе General появится новый элемент GLControl, который можно добавить на форму.

2. Создание проекта

Для решения задачи послойной визуализации томограммы мы будем использовать язык C# и стандарт и технологию OpenGL.

Создайте новый проект "Приложение Windows Forms" на языке C#, дайте ему название <Фамилия>_tomogram_visualizer.

В качестве библиотеки, реализующей стандарт OpenGL в проекте будет использоваться библиотека OpenTK. Подключение библиотеки к проекту подробно описано в документе "Подключение OpenTK в Visual Studio". Подключите библиотеку OpenTK к своему проекту согласно инструкции.

3. Чтение файла томограммы

Обычно файлы томограмм хранятся в файлах формата DICOM, но в связи с нетривиальностью данного формата в данной работе будет использоваться томограмма, сохраненная в бинарном формате. Для загрузки томограммы потребуется прочитать из бинарного файла размеры томограммы (3 числа в формате int) и массив данных типа short. Создайте класс для чтения данных файлов:

```

class Bin
{
    public static int X, Y, Z;
    public static short[] array;
    public Bin() { }

    public void readBIN(string path)
    {
        if (File.Exists(path))
        {
            BinaryReader reader =
                new BinaryReader(File.Open(path, FileMode.Open));

            X = reader.ReadInt32();
            Y = reader.ReadInt32();
            Z = reader.ReadInt32();

            int arraySize = X * Y * Z;
            array = new short[arraySize];
            for (int i = 0; i < arraySize; ++i)
            {
                array[i] = reader.ReadInt16();
            }
        }
    }
}

```

Создайте и инициализируйте объект класса Bin в классе Form1.

4. Классы для визуализации

Создайте класс View, который будет содержать функции для визуализации томограммы.

5. Настройка камеры

В классе View создайте функцию SetupView, которая будет настраивать окно вывода.

Включите интерполирование цветов, установив тип Smooth функцией GL.ShadeModel.

Матрицу проекции сначала инициализируйте, установив ее равной матрице тождественного преобразования (GL.LoadIdentity()). А затем задайте обращением к GL.Ortho() ортогональное проецирование массива данных томограммы в окно вывода, которое попутно преобразует размеры массива в канонический видимый объем (CVV).

Настройте вывод в окно OpenTK таким образом, чтобы разрешение синтезируемого изображения было равно размеру окна OpenTK.

Все действия по настройке камеры записаны в коде ниже:

```

public void SetupView(int width, int height)
{
    GL.ShadeModel(ShadingModel.Smooth);
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    GL.Ortho(0, Bin.X, 0, Bin.Y, -1, 1);
    GL.Viewport(0, 0, width, height);
}

```

6. Визуализация томограммы

В данной лабораторной работе будет проведено сравнение двух вариантов визуализации томограммы.

Вариант 1 - отрисовка четырехугольниками, вершинами которых являются центры вокселов текущего слоя регулярной воксельной сетки. Цвет формируется на центральном процессоре и отрисовывается с помощью функции `GL.Begin(BeginMode.Quads)`.

Вариант 2 - отрисовка текстурой. Текущий слой томограммы визуализируется как один большой четырехугольник, на который изображение слоя накладывается как текстура аппаратной билинейной интерполяцией.

7. Создание Transfer Function (TF)

TF - функция перевода значения плотностей томограммы в цвет. Диапазон визуализируемых плотностей называется окном визуализации. В нашем случае мы хотим, чтобы TF переводила плотности окна визуализации от 0 до 2000 линейно в цвет от черного до белого (от 0 до 255).

```

Color TransferFunction(short value)
{
    int min = 0;
    int max = 2000;
    int newVal = clamp((value - min) * 255 / (max - min), 0, 255);
    return Color.FromArgb(255, newVal, newVal, newVal);
}

```

Меняя параметры `min` и `max`, мы будем получать различные изображения для нашей томограммы. Часто TF имеет более сложную структуру, чем линейная зависимость от максимума и минимума, но в данной лабораторной работе нам достаточно такой.

8. Отрисовка четырехугольника

В классе `View` создайте функцию `DrawQuads` с параметром `int layerNumber` (номер визуализируемого слоя).

```

public void DrawQuads(int layerNumber)
{
    GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
    GL.Begin(BeginMode.Quads);
    for (int x_coord = 0; x_coord < Bin.X - 1; x_coord++)
        for (int y_coord = 0; y_coord < Bin.Y - 1; y_coord++)
    {
        short value;
        //1 вершина
        value = Bin.array[x_coord + y_coord * Bin.X
                          + layerNumber * Bin.X * Bin.Y];
        GL.Color3(TransferFunction(value));
        GL.Vertex2(x_coord, y_coord);
        //2 вершина
        value = Bin.array[x_coord + (y_coord + 1) * Bin.X
                          + layerNumber * Bin.X * Bin.Y];
        GL.Color3(TransferFunction(value));
        GL.Vertex2(x_coord, y_coord + 1);
        //3 вершина
        value = Bin.array[x_coord + 1 + (y_coord + 1) * Bin.X
                          + layerNumber * Bin.X * Bin.Y];
        GL.Color3(TransferFunction(value));
        GL.Vertex2(x_coord + 1, y_coord + 1);
        //4 вершина
        value = Bin.array[x_coord + 1 + y_coord * Bin.X
                          + layerNumber * Bin.X * Bin.Y];
        GL.Color3(TransferFunction(value));
        GL.Vertex2(x_coord + 1, y_coord);
    }
    GL.End();
}

```

Из томограммы извлекаются значения томограммы в 4 ячейках: (x,y), (x+1,y), (x,y+1), (x+1,y+1). Эти значения заносятся в цвет вершин четырехугольника, и данный четырехугольник визуализируется. Данная операция происходит в цикле по ширине и высоте томограммы. Перечисление вершин четырехугольника происходит против часовой стрелки.

9. Загрузка файла с данными и его визуализация

Создайте кнопку, либо элемент меню, по нажатию на который будет вызываться функция, которая будет открывать файл с томограммой и настраивать OpenGL окно. Код функции приведен ниже. В классе Form1 необходимо объявить переменную bool loaded = false, чтобы не запускать отрисовку, пока не загружены данные.

```

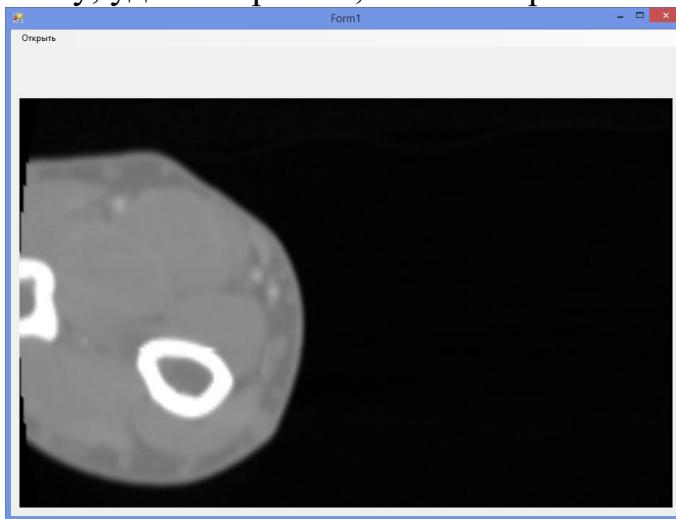
private void открытьToolStripMenuItem_Click(object sender, EventArgs e)
{
    OpenFileDialog dialog = new OpenFileDialog();
    if (dialog.ShowDialog() == DialogResult.OK)
    {
        string str = dialog.FileName;
        bin.readBIN(str);
        view.SetupView(glControl1.Width, glControl1.Height);
        loaded = true;
        glControl1.Invalidate();
    }
}

```

Откройте конструктор формы, откройте свойства OpenGL окна и в событиях (Events) выберите событие Paint, двойным щелчком создайте новую функцию, в ней вызовите функцию DrawQuads и функцию SwapBuffers. В OpenGL используется двойная буферизация (буфер, выводящий изображение на экран и буфер, используемый для создания изображения), функция SwapBuffers загружает наш буфер в буфер экрана. currentLayer - переменная типа int, которая хранит номер слоя для визуализации.

```
private void glControl1_Paint(object sender, PaintEventArgs e)
{
    if (loaded)
    {
        view.DrawQuads(currentLayer);
        glControl1.SwapBuffers();
    }
}
```

Запустите программу, удостоверьтесь, что томограмма визуализируется.



10. Перемотка слоев

Добавьте на форму Trackbar, по умолчанию значения значение Maxiumx равно 10. В функции загрузки томограммы измените значение максимума на количество слоев (переменная Z в классе Bin).

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    currentLayer = trackBar1.Value;
}
```

Проверьте, что теперь слои томограммы перелистываются.

11. Создание бесконечного цикла рендеринга и счетчика кадров

Производительность визуализации, измеряется в кадрах в секунду (Frames per second, FPS), чтобы её измерить нужно после рендера одного кадра и вывода его на экран автоматически начинать рендерить следующий кадр. Функция Application_Idle проверяет, занято ли OpenGL окно работой, если нет, то вызывается функция Invalidate, которая заставляет кадр рендериться заново.

```

void Application_Idle(object sender, EventArgs e)
{
    while (glControl1.IsIdle)
    {
        glControl1.Invalidate();
    }
}

```

Чтобы функция Application_Idle работала автоматически, вам нужно подключить ее в программе. В конструкторе формы создайте функцию Form1_load, в которой вы подключите Application_Idle на автоматическое выполнение.

```

private void Form1_Load(object sender, EventArgs e)
{
    Application.Idle += Application_Idle;
}

```

В классе Form1 создайте переменные int FrameCount, DateTime NextFPSUpdate и функцию displayFPS.

```

int FrameCount;
DateTime NextFPSUpdate = DateTime.Now.AddSeconds(1);
void displayFPS()
{
    if (DateTime.Now >= NextFPSUpdate)
    {
        this.Text = String.Format("CT Visualizer (fps={0})", FrameCount);
        NextFPSUpdate = DateTime.Now.AddSeconds(1);
        FrameCount = 0;
    }
    FrameCount++;
}

```

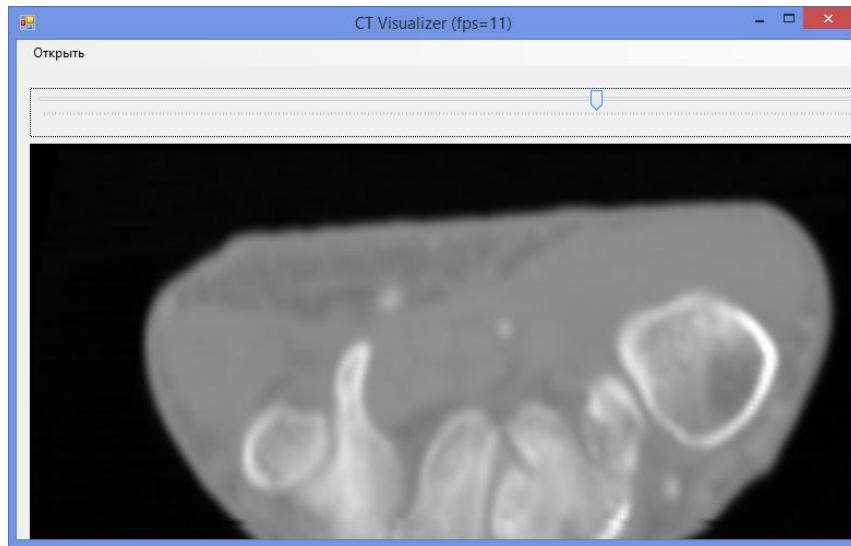
Вызовите данную функцию обновления FPS в функции Application_Idle.

```

void Application_Idle(object sender, EventArgs e)
{
    while (glControl1.IsIdle)
    {
        displayFPS();
        glControl1.Invalidate();
    }
}

```

Запустите программу, посмотрите, какой fps будет выдавать ваша программа.



12. Визуализация томограммы как текстуры

Создайте функцию загрузки и функцию визуализации.

13. Загрузка текстуры в память видеокарты

В класса View создайте переменную int VBOtexture и функцию Load2dTexture. Переменная VBOtexture будет хранить номер текстуры в памяти видеокарты. Функция GenTextures генерирует уникальный номер текстуры, функция BindTexture связывает текстуру, делает ее активной, а также указывает ее тип, функция TexImage2D загружает текстуру в память видеокарты.

```
Bitmap textureImage;
int VBOtexture;
public void Load2DTexture()
{
    GL.BindTexture(TextureTarget.Texture2D, VBOtexture);
    BitmapData data = textureImage.LockBits(
        new System.Drawing.Rectangle(0, 0, textureImage.Width, textureImage.Height),
        ImageLockMode.ReadOnly,
        System.Drawing.Imaging.PixelFormat.Format32bppArgb);

    GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba,
        data.Width, data.Height, 0, OpenGL.PixelFormat.Bgra,
        PixelType.UnsignedByte, data.Scan0);

    textureImage.UnlockBits(data);

    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter,
        (int)TextureMinFilter.Linear);
    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter,
        (int)TextureMagFilter.Linear);

    ErrorCode Er = GL.GetError();
    string str = Er.ToString();
}
```

14. Визуализация томограммы одним прямоугольником

Суть визуализации томограммы одним прямоугольником будет заключаться в следующем: мы сделаем картинку из текстуры один раз на процессоре, передадим ее в видеопамять, и будем производить текстурирование одного прямоугольника.

В классе View создайте переменную Bitmap textureImage и функцию generateTextureImage, которая будет генерировать изображение из томограммы при помощи созданной Transfer Function.

```
public void generateTextureImage(int layerNumber)
{
    textureImage = new Bitmap(Bin.X, Bin.Y);
    for (int i = 0; i < Bin.X; ++i)
        for (int j = 0; j < Bin.Y; ++j)
    {
        int pixelNumber = i + j * Bin.X + layerNumber * Bin.X * Bin.Y;
        textureImage.SetPixel(i, j, TransferFunction(Bin.array[pixelNumber]));
    }
}
```

Создайте функцию drawTexture(), которая будет включать 2d-текстурирование, выбирать текстуру и рисовать один прямоугольник с наложенной текстурой, потом выключать 2d-текстурирование.

```
public void DrawTexture()
{
    GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
    GL.Enable(EnableCap.Texture2D);
    GL.BindTexture(TextureTarget.Texture2D, VBOtexture);

    GL.Begin(BeginMode.Quads);
    GL.Color3(Color.White);
    GL.TexCoord2(0f, 0f);
    GL.Vertex2(0, 0);
    GL.TexCoord2(0f, 1f);
    GL.Vertex2(0, Bin.Y);
    GL.TexCoord2(1f, 1f);
    GL.Vertex2(Bin.X, Bin.Y);
    GL.TexCoord2(1f, 0f);
    GL.Vertex2(Bin.X, 0);
    GL.End();

    GL.Disable(EnableCap.Texture2D);
}
```

Визуализация с помощью четырехугольников разбивается на 2 подзадачи:

1. Генерация текстуры и загрузка в видеопамять. Выполняется один раз для слоя.

2. Визуализация текстуры. Происходит постоянно.

В классе Form1 измените функцию glControl1_Paint так, чтобы она рисовала томограмму с помощью текстуры, а загружала текстуру только когда переменная needReload будет установлена в true.

```

bool needReload = false;
private void glControl1_Paint(object sender, PaintEventArgs e)
{
    if (loaded)
    {
        //view.DrawQuads(currentLayer);
        if (needReload)
        {
            view.generateTextureImage(currentLayer);
            view.Load2DTexture();
            needReload = false;
        }
        view.DrawTexture();
        glControl1.SwapBuffers();
    }
}

```

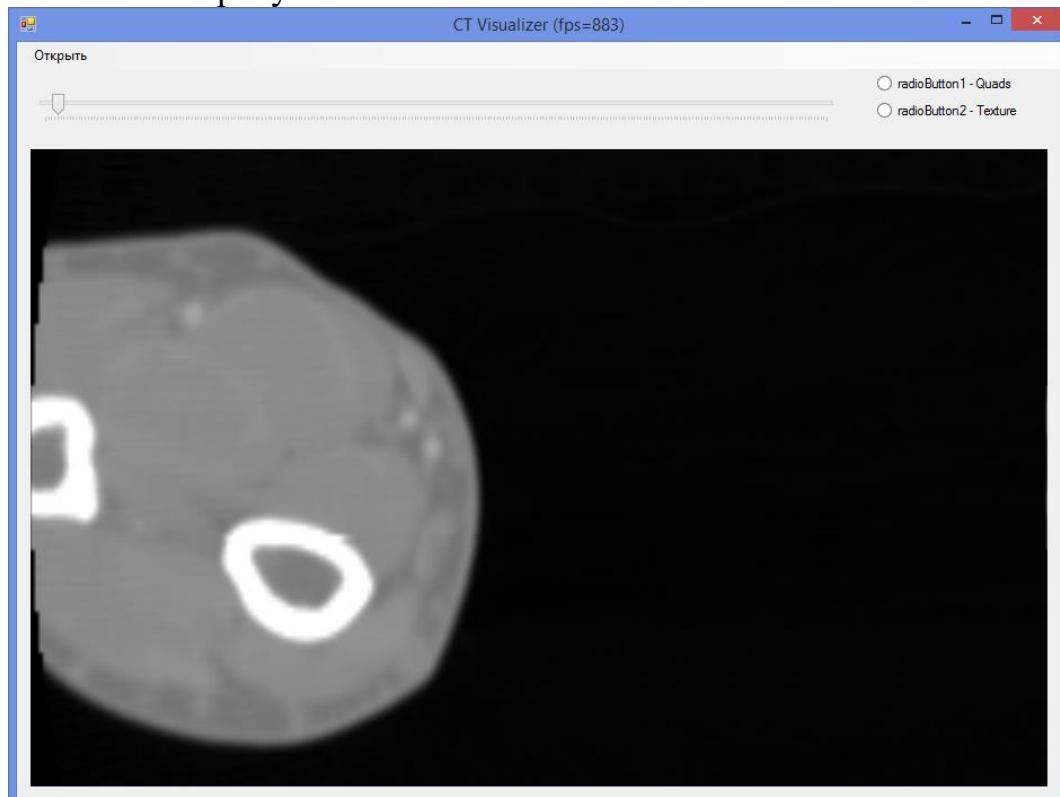
Переменную needReload необходимо устанавливать в значение true тогда, когда мы изменяем trackbar.

```

private void trackBar1_Scroll(object sender, EventArgs e)
{
    currentLayer = trackBar1.Value;
    needReload = true;
}

```

Запустите программу. Сравните FPS версии рисования текстурой с FPS версии рисования четырехугольниками.



Сделайте возможность переключаться между режимами визуализации четырехугольниками и текстурой (например, при помощи CheckBox или RadioButton).

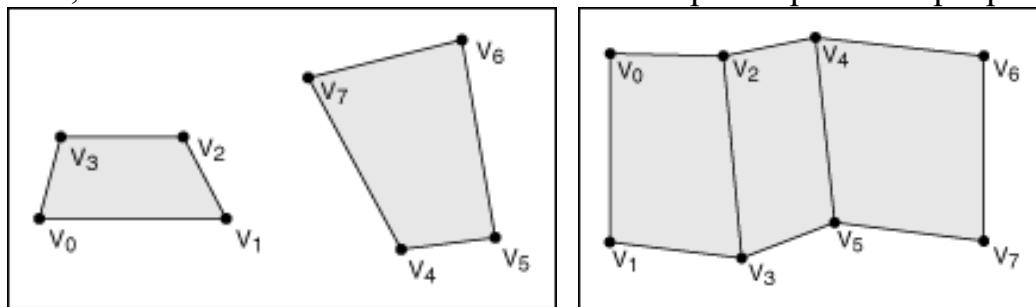
15. Задания для самостоятельной работы

1. Изменение Transfer Function

Создайте два TrackBar, которые будут использоваться для задания Transfer Function. Первый TrackBar будет указывать на значение минимума, второй - на ширину TF, тогда значение максимума можно вычислить как сумму данных двух значений. Не забудьте, что при изменении TF в режиме рисования текстурой необходимо загружать новую текстуру в видеопамять.

2. Отрисовка при помощи QuadStrip

В OpenGL есть тип визуализации QuadStrip, когда первый четырехугольник рисуется 4 вершинами, а последующие - 2 вершинами, присоединенными к предыдущему четырехугольнику (рис. ниже). Таким образом для отрисовки N четырехугольников требуется не $4*N$ вершин, а $2*N+2$ вершин, что положительно сказывается на скорости работы программы.



16. Ссылки

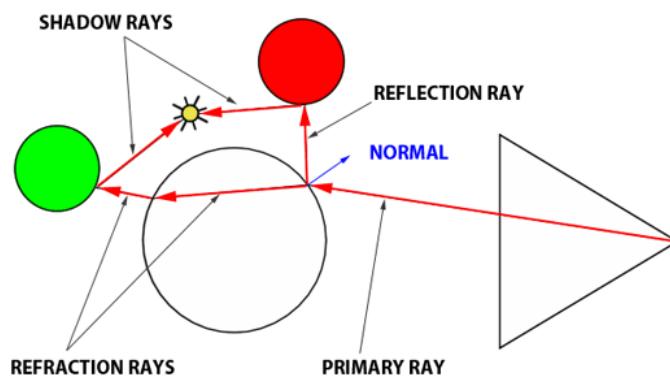
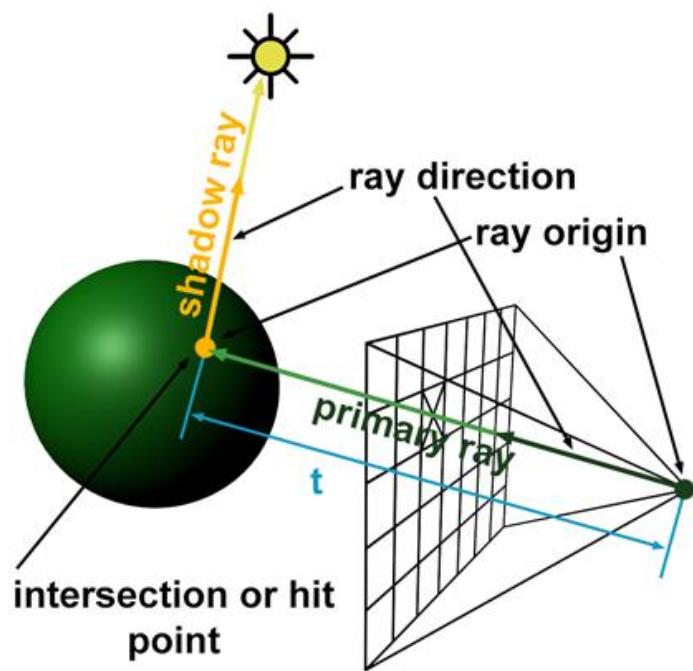
1. <https://habr.com/post/310790/> - большой цикл статей по OpenGL.
2. <https://github.com/opentk/opentk> - основной репозиторий библиотеки OpenTK.
3. <https://www.intuit.ru/studies/courses/2313/613/lecture/13296> - ИНТУИТ. Создание графических моделей с помощью Open Graphics Library.
4. <https://habr.com/post/173131/> - эмулятор кубика-рубика на OpenTK.

Лабораторная работа №3. «Трассировка лучей»

1. Что такое Трассировка лучей (Ray Tracing)

В контексте данной лабораторной Ray Tracing - это алгоритм построения изображения трёхмерных моделей в компьютерных программах, при которых отслеживается обратная траектория распространения луча (от экрана к источнику).

На рисунке ниже показано прохождение луча из камеры через экран и полупрозрачную сферу. При достижении лучом сферы луч раздваивается, и один из новых лучей отражается, а другой преломляется и проходит сквозь сферу. При достижении лучами диффузных объектов вычисляется цвет, цвет от обоих лучей суммируется и окрашивает пикселя.



В алгоритме присутствует несколько важных вычислительных этапов:

- Вычисление цвета для диффузной поверхности;

- b. Вычисление направления отражения;
- c. Вычисление направления преломления;

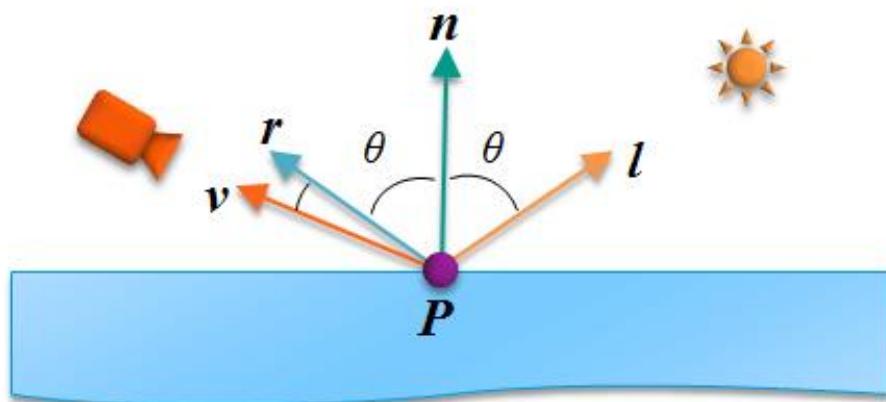
Вычисление цвета для диффузной поверхности: освещение по формуле Фонга:

$$C_{out} = \text{C} \cdot k_a + \text{C} \cdot k_d \cdot \max((\vec{n}, \vec{l}), 0) + L \cdot k_s \cdot \max((\vec{v}, \vec{r}), 0)^p, \text{ где}$$

$$r = \text{reflect}(-v, n)$$

C – цвет материала

L – цвет блика



Вычисление цвета точки Р по формуле Фонга.

Вычисление направления отражения. Если поверхность обладает отражающими свойствами, то строится второй луч отражения. Направление луча определяется по закону отражения (геометрическая оптика):

$$r = i - 2 \cdot n \cdot (n \cdot i)$$

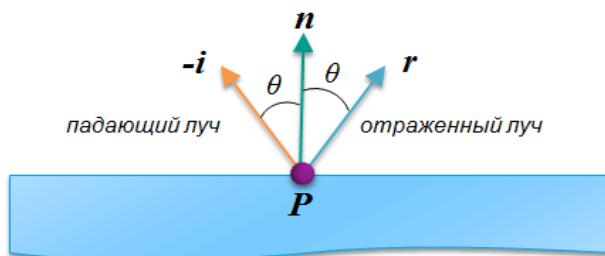


Рисунок 3. Вычисление отраженного луча.

Вычисление направления преломления. Если поверхность прозрачна, то строится еще и второй луч прозрачности (transparency ray). Для определения направления луча используется закон преломления (геометрическая оптика):

$$\sin(\alpha) / \sin(\beta) = \eta_2 / \eta_1$$

$$\begin{aligned} \mathbf{t} &= (\eta_1 / \eta_2) \cdot \mathbf{i} - [\cos(\beta) + (\eta_1 / \eta_2) \cdot (\mathbf{n} \cdot \mathbf{i})] \cdot \mathbf{n}, \\ \cos(\beta) &= \sqrt{1 - (\eta_1 / \eta_2)^2 \cdot (1 - (\mathbf{n} \cdot \mathbf{i})^2)} \end{aligned}$$

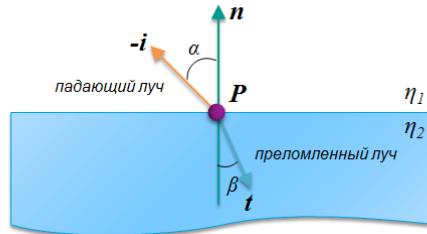


Рисунок 4. Вычисление преломленного луча.

2. О шейдерных программах

Шейдеры – это части шейдерной программы, которые обеспечивают исполнение определенных шагов трехмерного графического конвейера, реализуемого через посредство межплатформенного стандарта OpenGL или через MS DirectX. Они выполняются непосредственно на GPU и в параллельных потоках. Шейдерные программы предназначены для замены определенной части шагов Стандартного трехмерного графического конвейера (с фиксированной функциональностью). С версии GPU, соответствующей OpenGL 3.1, фиксированная функциональность программируемой части графического конвейера была удалена и шейдеры стали обязательными. Шейдеры для OpenGL пишутся на специализированном C-подобном языке — для стандарта OpenGL этот язык называется GLSL (для MS DirectX – HLSL). В OpenGL программа в GLSL компилируется перед использованием в команды эквивалентные операции и загружается в GPU для исполнения.

Шейдерная программа объединяет набор шейдеров. В простейшем случае шейдерная программа состоит из двух шейдеров: вершинного и фрагментного.

Вершинный шейдер вызывается для каждой вершины. Его выходные данные интерполируются и поступают на вход фрагментного шейдера. Обычно, работа вершинного шейдера состоит в том, чтобы перевести координаты вершин из пространства сцены в пространство экрана и выполнить вспомогательные расчёты для фрагментного шейдера.

Фрагментный шейдер вызывается для каждого графического фрагмента (пикселя растеризованной геометрии, попадающего на экран). Выходом фрагментного шейдера, как правило, является цвет фрагмента, идущий в буфер цвета. На фрагментный шейдер обычно ложится основная часть расчёта освещения по Фонгу.

Виды шейдеров: вершинный, тесселяции, геометрический, фрагментный (рисунок 5).

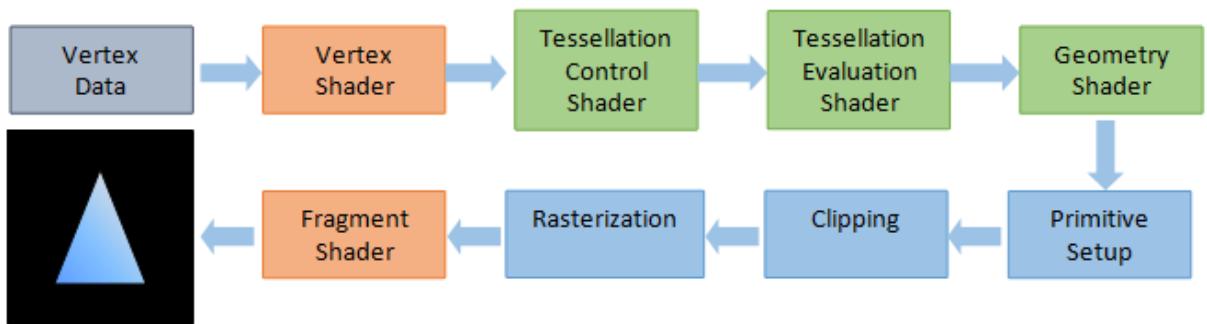


Рисунок 5. Графический конвейер OpenGL 4.3

3. Создание шейдерной программы.

В данной лабораторной работе мы будем программировать вершинный и фрагментный шейдеры. Шейдеры – это два текстовых файла. Их нужно загрузить с диска, скомпилировать и слинковать в шейдерную программу. Создайте два пустых текстовых файла «raytracing.vert» для вершинного шейдера и «raytracing.frag» - для фрагментного. По стандарту OpenGL шейдеры после загрузки компилируются основной CPU программой. Первым этапом лабораторной работы является создание, компиляция и подключение простейшей шейдерной программы.

Для более упорядоченного вида программы функции, связанные с OpenGL и шейдерами, можно вынести в отдельный класс View, как это было сделано в предыдущей лабораторной работе.

Загрузка шейдеров

```

void loadShader(String filename, ShaderType type, uint program, out uint address)
{
    address = GL.CreateShader(type);
    using (System.IO.StreamReader sr = new StreamReader(filename))
    {
        GL.ShaderSource(address, sr.ReadToEnd());
    }
    GL.CompileShader(address);
    GL.AttachShader(program, address);
    Console.WriteLine(GL.GetShaderInfoLog(address));
}

```

glCreateShader создаёт объект шейдера, её аргумент определяет тип шейдера, возвращает значение, которое можно использовать для ссылки на объект шейдера (дескриптор, то есть в нашем случае это идентификаторы uint VertexShader и uint FragmentShader, которые в данную функцию передаются через аргумент address).

glShaderSource загружает исходный код в созданный шейдерный объект.

Далее надо скомпилировать исходный шейдер, делается это вызовом функции `glCompileShader()` и передачей ей дескриптора шейдера, который требуется скомпилировать.

Перед тем, как шейдеры будут добавлены в конвейер OpenGL их нужно скомпоновать в шейдерную программу с помощью функции `glAttachShader()`. На этапе компоновки производитсястыковка входных переменных одного шейдера с выходными переменными другого, а также стыковка входных/выходных переменных шейдеров с соответствующими областями памяти в окружении OpenGL.

Инициализация шейдерной программы

В функции `InitShaders()` теперь необходимо создать объект шейдерной программы и вызвать ранее реализованную функцию `loadShader()`, чтобы создать объекты шейдеров, скомпилировать их и скомпоновать в объекте шейдерной программы:

```
BasicProgramID = GL.CreateProgram(); // создание объекта программы
loadShader("../..\..\raytracing.vert", ShaderType.VertexShader, BasicProgramID,
           out BasicVertexShader);
loadShader("../..\..\raytracing.frag", ShaderType.FragmentShader, BasicProgramID,
           out BasicFragmentShader);
GL.LinkProgram(BasicProgramID);
// Проверяем успех компоновки
int status = 0;
GL.GetProgram(BasicProgramID, GetProgramParameterName.LinkStatus, out status);
Console.WriteLine(GL.GetProgramInfoLog(BasicProgramID));
```

Настройка буферных объектов

Для того, чтобы начали работать (синтезировать изображение) нужно нарисовать квадрат (`GL_QUAD`), заполняющий окно визуализации. Можно рисовать его классическим методом, можно используя буферный объект. Код для буферного объекта ниже.

Сначала создайте член класса `int vbo_position` для хранения дескриптора объекта массива вершин и массив вершин.

```
vertdata = new Vector3[] {
    new Vector3(-1f, -1f, 0f),
    new Vector3( 1f, -1f, 0f),
    new Vector3( 1f,  1f, 0f),
    new Vector3(-1f,  1f, 0f)};

GL.GenBuffers(1, out vbo_position);

GL.BindBuffer(BufferTarget.ArrayBuffer, vbo_position);

GL.BufferData<Vector3>(BufferTarget.ArrayBuffer, (IntPtr)(vertdata.Length *
    Vector3.SizeInBytes), vertdata, BufferUsageHint.StaticDraw);
GL.VertexAttribPointer(attribute_vpos, 3, VertexAttribPointerType.Float, false, 0, 0);

GL.Uniform3(uniform_pos, campos);
GL.Uniform1(uniform_aspect, aspect);
```

```
GL.UseProgram(BasicProgramID);
GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
```

Вершинный шейдер

Вершинный шейдер может быть самым простым – перекладывать интерполированные координаты вершин в выходную переменную, и тогда генерировать луч надо во фрагментном шейдере, а может быть более сложным - с генерацией направления луча.

```
in vec3 vPosition; //Входные переменные vPosition - позиция вершины
out vec3 glPosition;

void main (void)
{
    gl_Position = vec4(vPosition, 1.0);
    glPosition = vPosition;
}
```

Переменные, отдаваемые вершинным шейдером дальше по конвейеру объявлены со спецификатором `out`.

Фрагментный шейдер

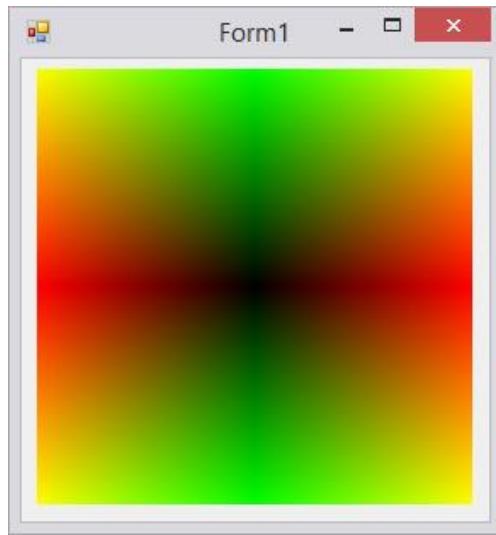
```
#version 430

out vec4 FragColor;
in vec3 glPosition;

void main ( void )
{
    FragColor = vec4 ( abs(glPosition.xy), 0, 1.0);
}
```

У этого шейдера единственная выходная переменная: `FragColor`. Система сама, что если выходная переменная одна, значит она соответствует пикселу в буфере экрана. Единственная входная переменная соответствует выходной переменной вершинного шейдера. Функция `main` записывает интерполированные координаты в выходной буфер цвета. Функция `abs` (модуль) применяется потому, что компонента цвета не может быть отрицательной, а наши интерполированные значения лежат в диапазоне от -1 до 1.

После запуска у вас должна появиться картинка:

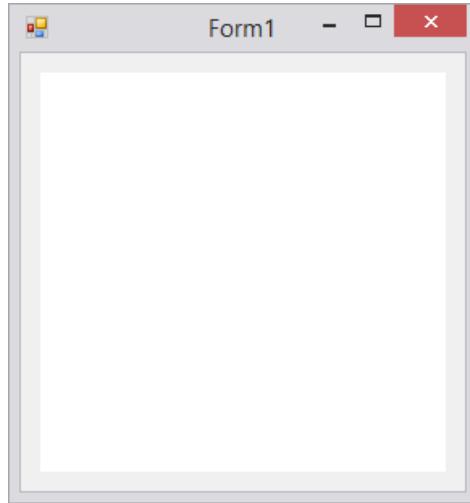


Левый нижний угол соответствует координатам $(-1, -1)$, правый верхний – $(1, 1)$.

Если в коде шейдеров содержится ошибка и компилятор не смог его скомпилировать, то после запуска программы у вас будет пустой экран, а в окно вывода напечатается лог с указанием ошибки компиляции. Например,

```
0(8) : error C1068: too much data in type constructor
```

Это значит, что в восьмой строке был передан лишний параметр в конструктор.



4. Генерация первичного луча

<http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays>

Для моделирования наблюдателя предназначена камера. В обратной трассировке лучей (Ray Tracing) через каждый пиксель окна изображения должен быть выпущен луч в сцену. Обозначим новый раздел “DATA STRUCTURES” и создадим две структуры для камеры и для луча:

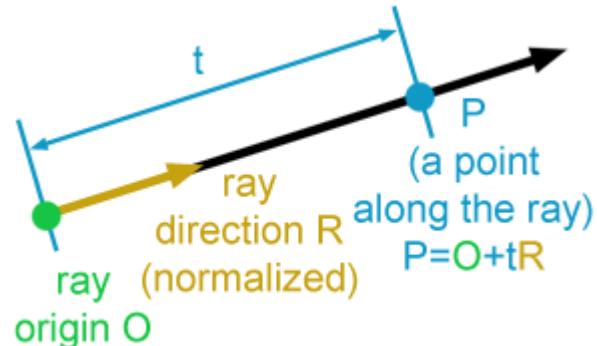
```

/*** DATA STRUCTURES ***/
struct SCamera
{
    vec3 Position;
    vec3 View;
    vec3 Up;
    vec3 Side;
    // отношение сторон выходного изображения
    vec2 Scale;
};

struct SRay
{
    vec3 Origin;
    vec3 Direction;
};

```

Первичный луч - луч, который исходит из камеры. Чтобы правильно вычислить луч для каждого пикселя, нужно вычислить его начало и его направление. Координаты всех точек на луче $r = o + dt, t \in [0, \infty)$.



Добавляем функцию генерации луча:

```

SRay GenerateRay ( SCamera uCamera )
{
    vec2 coords = glPosition.xy * uCamera.Scale;
    vec3 direction = uCamera.View + uCamera.Side * coords.x + uCamera.Up * coords.y;
    return SRay ( uCamera.Position, normalize(direction) );
}

SCamera initializeDefaultCamera()
{
    /** CAMERA ***/
    camera.Position = vec3(0.0, 0.0, -8.0);
    camera.View = vec3(0.0, 0.0, 1.0);
    camera.Up = vec3(0.0, 1.0, 0.0);
    camera.Side = vec3(1.0, 0.0, 0.0);
    camera.Scale = vec2(1.0);
}

```

Изменяем main()

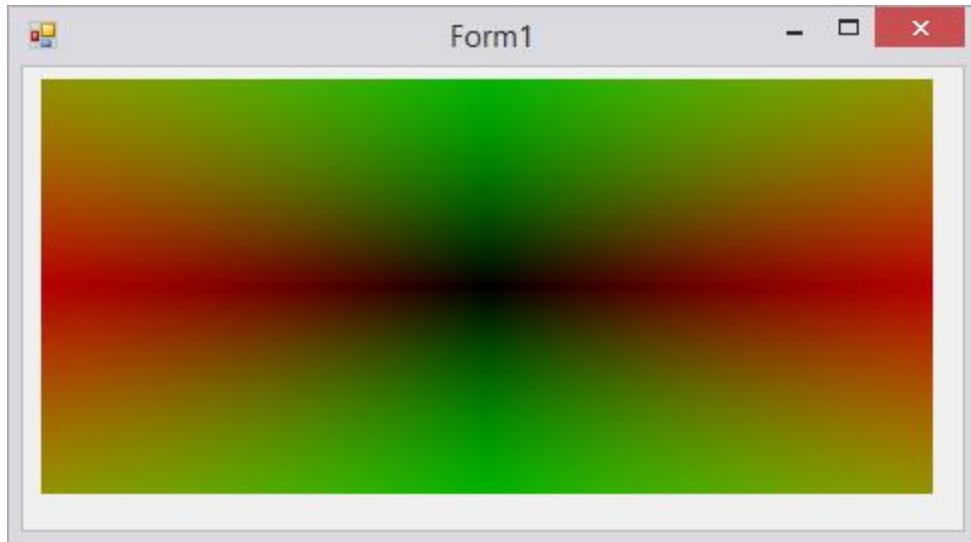
```

void main ( void )
{

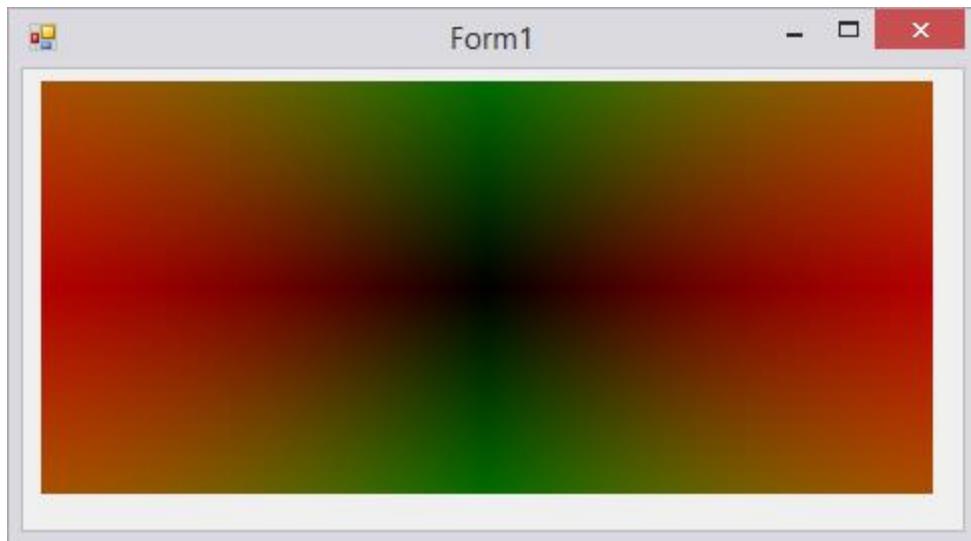
```

```
SCamera uCamera = initializeDefaultCamera();
SRay ray = GenerateRay( uCamera);
FragColor = vec4 ( abs(ray.Direction.xy), 0, 1.0);
}
```

Умножение на `uCamera.Scale` необходимо, чтобы изображение не деформировалось при изменении размеров окна:



Без умножения на `uCamera.Scale`.



После умножения на `uCamera.Scale`.

5. Добавление структур данных сцены и источников света.

Объявление типов данных

Большая часть кода будет написана в фрагментном шейдере. Для нашей программы понадобятся структуры для следующих объектов: камера, источник света, луч, пересечение, сфера, треугольник, материал.

```
#version 430
#define EPSILON = 0.001
#define BIG = 1000000.0
const int DIFFUSE = 1;
const int REFLECTION = 2;
const int REFRACTION = 3;

struct SSphere
{
    vec3 Center;
    float Radius;
    int MaterialIdx;
};

struct STriangle
{
    vec3 v1;
    vec3 v2;
    vec3 v3;
    int MaterialIdx;
};
```

Объявление и инициализация данных

В фрагментном шейдере объявите глобальные массивы сфер и треугольников. В структурах есть переменные MaterialIdx, которые будут содержать индекс в массиве материалов. Структура материала будет рассмотрена в следующем разделе. Пока можно задать все индексы нулевыми.

```
STriangle triangles[10];
SSphere spheres[2];
```

Создайте функцию *initializeDefaultScene()*, которая будет инициализировать переменные данными по умолчанию.

```
void initializeDefaultScene(out STriangle triangles[], out SSphere spheres[])
{
    /** TRIANGLES */
    /* left wall */
    triangles[0].v1 = vec3(-5.0, -5.0, -5.0);
    triangles[0].v2 = vec3(-5.0, 5.0, 5.0);
    triangles[0].v3 = vec3(-5.0, 5.0, -5.0);
    triangles[0].MaterialIdx = 0;

    triangles[1].v1 = vec3(-5.0, -5.0, -5.0);
    triangles[1].v2 = vec3(-5.0, -5.0, 5.0);
    triangles[1].v3 = vec3(-5.0, 5.0, 5.0);
    triangles[1].MaterialIdx = 0;

    /* back wall */
    triangles[2].v1 = vec3(-5.0, -5.0, 5.0);
```

```

triangles[2].v2 = vec3( 5.0,-5.0, 5.0);
triangles[2].v3 = vec3(-5.0, 5.0, 5.0);
triangles[2].MaterialIdx = 0;

triangles[3].v1 = vec3( 5.0, 5.0, 5.0);
triangles[3].v2 = vec3(-5.0, 5.0, 5.0);
triangles[3].v3 = vec3( 5.0,-5.0, 5.0);
triangles[3].MaterialIdx = 0;

/* Самостоятельно добавьте треугольники так, чтобы получился куб */

/** SPHERES */
spheres[0].Center = vec3(-1.0,-1.0,-2.0);
spheres[0].Radius = 2.0;
spheres[0].MaterialIdx = 0;

spheres[1].Center = vec3(2.0,1.0,2.0);
spheres[1].Radius = 1.0;
spheres[1].MaterialIdx = 0;
}

```

Вызовите Вашу функцию после объявленных переменных.

```
initializeDefaultScene ( triangles, spheres );
```

Проверьте, что Ваш шейдер компилируется и при запуске Вашей программы Output не содержит ошибок.

6. Пересечение луча с объектами

Для того, чтобы отрисовать сцену, необходимо реализовать пересечение луча с объектами сцены - треугольниками и сферами.

Пересечение луча со сферой

Есть несколько алгоритмов для пересечения луча со сферой, - можно воспользоваться аналитическим решением:

Уравнение луча: $r = o + d \cdot t$

Уравнение сферы: $x^2 + y^2 + z^2 = R^2$ или $P^2 - R^2 = 0$

Подставляем $(o + d \cdot t)^2 - R^2 = 0$

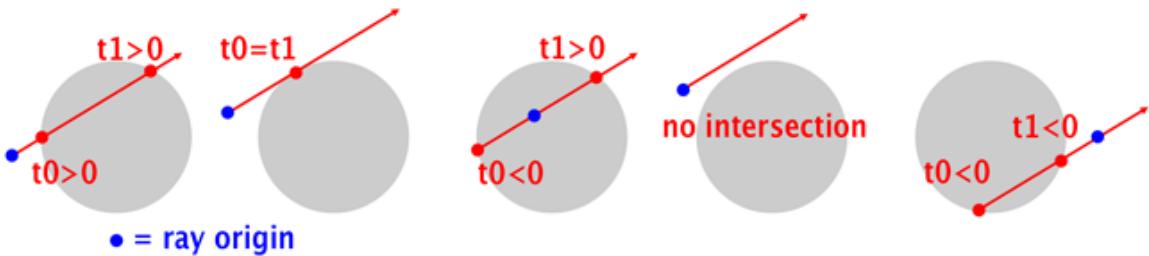
Раскрываем скобки и получаем квадратное уравнение относительно t :

$$d^2t^2 + 2o \cdot d \cdot t + o^2 - R^2 = 0$$

Если дискриминант > 0 , то луч пересекает сферу в двух местах, нам нужно ближайшее пересечение, при $t > 0$. Т.к. если $t < 0$, значит луч пересекает сферу до его начала.

Если дискриминант = 0, то точка пересечения (касания) одна.

Если дискриминант < 0 , то точек пересечения нет.



Реализация функции IntersectSphere представлена ниже:

```
bool IntersectSphere ( SSphere sphere, SRay ray, float start, float final, out float time )
{
    ray.Origin -= sphere.Center;
    float A = dot ( ray.Direction, ray.Direction );
    float B = dot ( ray.Direction, ray.Origin );
    float C = dot ( ray.Origin, ray.Origin ) - sphere.Radius * sphere.Radius;
    float D = B * B - A * C;
    if ( D > 0.0 )
    {
        D = sqrt ( D );
        //time = min ( max ( 0.0, ( -B - D ) / A ), ( -B + D ) / A );
        float t1 = ( -B - D ) / A;
        float t2 = ( -B + D ) / A;
        if(t1 < 0 && t2 < 0)
            return false;

        if(min(t1, t2) < 0)
        {
            time = max(t1,t2);
            return true;
        }
        time = min(t1, t2);
        return true;
    }
    return false;
}
```

Способы пересечения треугольника с лучом можно найти в презентации http://www2.sssc.ru/Seminars/Docum/NVIDIA%20CUDA-2012/Lec_6_Novsb.pdf

Реализация ниже:

```
bool IntersectTriangle (SRay ray, vec3 v1, vec3 v2, vec3 v3, out float time )
{
    // // Compute the intersection of ray with a triangle using geometric solution
    // Input: // points v0, v1, v2 are the triangle's vertices
    // rayOrig and rayDir are the ray's origin (point) and the ray's direction
    // Return: // return true is the ray intersects the triangle, false otherwise
    // bool intersectTriangle(point v0, point v1, point v2, point rayOrig, vector rayDir) {
    // compute plane's normal vector
    time = -1;
    vec3 A = v2 - v1;
    vec3 B = v3 - v1;
    // no need to normalize vector
    vec3 N = cross(A, B);
    // N
    // // Step 1: finding P
```

```

// // check if ray and plane are parallel ?
float NdotRayDirection = dot(N, ray.Direction);
if (abs(NdotRayDirection) < 0.001)
    return false;
// they are parallel so they don't intersect !
// compute d parameter using equation 2
float d = dot(N, v1);
// compute t (equation 3)
float t = -(dot(N, ray.Origin) - d) / NdotRayDirection;
// check if the triangle is in behind the ray
if (t < 0)
    return false;
// the triangle is behind
// compute the intersection point using equation 1
vec3 P = ray.Origin + t * ray.Direction;
// // Step 2: inside-outside test //
vec3 C;
// vector perpendicular to triangle's plane
// edge 0
vec3 edge1 = v2 - v1;
vec3 VP1 = P - v1;
C = cross(edge1, VP1);
if (dot(N, C) < 0)
    return false;
// P is on the right side
// edge 1
vec3 edge2 = v3 - v2;
vec3 VP2 = P - v2;
C = cross(edge2, VP2);
if (dot(N, C) < 0)
    return false;
// P is on the right side
// edge 2
vec3 edge3 = v1 - v3;
vec3 VP3 = P - v3;
C = cross(edge3, VP3);
if (dot(N, C) < 0)
    return false;
// P is on the right side;
time = t;
return true;
// this ray hits the triangle
}

```

Функция Raytrace пересекает луч со всеми примитивами сцены и возвращает ближайшее пересечение.

Создаём структуру для хранения пересечения. В структуре предусмотрены поля для хранения цвета и материала текущей точки пересечения, материалы будут рассмотрены в следующем разделе, пока заполним их нулями.

```

struct SIntersection
{
    float Time;
    vec3 Point;
    vec3 Normal;
    vec3 Color;
    // ambient, diffuse and specular coeffs

```

```

    vec4 LightCoeffs;
    // 0 - non-reflection, 1 - mirror
    float ReflectionCoef;
    float RefractionCoef;
    int MaterialType;
};

}

```

Создаём функцию трассирующую луч. Пока у вас нет материалов, просто присвойте любой цвет.

```

bool Raytrace ( SRay ray, SSphere spheres[], STriangle triangles[], SMaterial
materials[], float start, float final, inout SIntersection intersect )
{
bool result = false;
float test = start;
intersect.Time = final;
//calculate intersect with spheres
for(int i = 0; i < 2; i++)
{
    SSphere sphere = spheres[i];
    if( IntersectSphere (sphere, ray, start, final, test) && test < intersect.Time )
    {
        intersect.Time = test;
        intersect.Point = ray.Origin + ray.Direction * test;
        intersect.Normal = normalize ( intersect.Point - spheres[i].Center );
        intersect.Color = vec3(1,0,0);
        intersect.LightCoeffs =  vec4(0,0,0,0);
        intersect.ReflectionCoef = 0;
        intersect.RefractionCoef = 0;
        intersect.MaterialType = 0;
        result = true;
    }
}
//calculate intersect with triangles
for(int i = 0; i < 10; i++)
{
    STriangle triangle = triangles[i];

    if(IntersectTriangle(ray, triangle.v1, triangle.v2, triangle.v3, test)
      && test < intersect.Time)
    {
        intersect.Time = test;
        intersect.Point = ray.Origin + ray.Direction * test;
        intersect.Normal =
            normalize(cross(triangle.v1 - triangle.v2, triangle.v3 - triangle.v2));
        intersect.Color = vec3(1,0,0);
        intersect.LightCoeffs =  vec4(0,0,0,0);
        intersect.ReflectionCoef = 0;
        intersect.RefractionCoef = 0;
        intersect.MaterialType = 0;
        result = true;
    }
}
return result;
}

```

Модифицируйте функцию main следующим образом:

```

#define BIG 1000000.0

void main ( void )
{
    float start = 0;
    float final = BIG;

    SCamera uCamera = initializeDefaultCamera();
    SRay ray = GenerateRay( uCamera );
    SIntersection intersect;
    intersect.Time = BIG;
    vec3 resultColor = vec3(0,0,0);
    initializeDefaultScene(triangles, spheres);
    if (Raytrace(ray, spheres, triangles, materials, start, final, intersect))
    {
        resultColor = vec3(1,0,0);
    }
    FragColor = vec4 (resultColor, 1.0);
}

```

Теперь на экране должны появиться красные силуэты геометрии сцены.

7. Настройка освещения

Чтобы объекты имели собственную тень и отбрасывали падающую в нашу модель необходимо добавить источник света.

```

struct SLight
{
    vec3 Position;
};

```

Можно выбрать любую из моделей освещения: Ламберт, Фонг, Блинн, Кук-Торренс. Мы будем реализовывать шейдинг (закрашивание) по Фонгу (http://compgraphics.info/3D/lighting/phong_reflection_model.php). Это локальная модель освещения, т.е. она учитывает только свойства заданной точки и источников освещения, игнорируя эффекты рассеивания, линзирования, отражения от соседних тел. Расчёт освещения по Фонгу требует вычисления цветовой интенсивности трёх компонент освещения: фоновой (ambient), рассеянной (diffuse) и глянцевых бликов (specular). Фоновая компонента — грубое приближение лучей света, рассеянных соседними объектами и затем достигших заданной точки; остальные две компоненты имитируют рассеивание и зеркальное отражение прямого излучения.

$$I = k_a I_a + k_d (\vec{n}, \vec{l}) + k_s \cdot \max((\vec{v}, \vec{r}), 0)^p, \text{ где}$$

\vec{n} – вектор нормали к поверхности в точке

\vec{l} – направление на источник света

\vec{v} – направление на наблюдателя

k_a – коэффициент фонового освещения

k_d – коэффициент диффузного освещения

k_s – коэффициент зеркального отражения, r – коэффициент резкости бликов

$$r = \text{reflect}(-\vec{v}, \vec{n})$$

Для расчета кроме цвета предмета нам потребуются коэффициенты k_a , k_d , k_s , и r .

Создаем структуру для хранения коэффициентов материала. Для расчета освещения по Фонгу нам потребуются первые два поля, остальные поля потребуются при реализации зеркального отражения и преломления.

```
struct SMaterial
{
    //diffuse color
    vec3 Color;
    // ambient, diffuse and specular coeffs
    vec4 LightCoeffs;
    // 0 - non-reflection, 1 - mirror
    float ReflectionCoef;
    float RefractionCoef;
    int MaterialType;
};
```

Добавляем как глобальные переменные источник освещения и массив материалов, а также функцию, задающую им значения по умолчанию. Вызовите эту функцию в main.

```
SLight light;
SMaterial materials[6];

void initializeDefaultLightMaterials(out SLight light, out SMaterial materials[])
{
    /** LIGHT ***/
    light.Position = vec3(0.0, 2.0, -4.0f);

    /** MATERIALS ***/
    vec4 lightCoefs = vec4(0.4, 0.9, 0.0, 512.0);
    materials[0].Color = vec3(0.0, 1.0, 0.0);
    materials[0].LightCoeffs = vec4(lightCoefs);
    materials[0].ReflectionCoef = 0.5;
    materials[0].RefractionCoef = 1.0;
    materials[0].MaterialType = DIFFUSE;

    materials[1].Color = vec3(0.0, 0.0, 1.0);
    materials[1].LightCoeffs = vec4(lightCoefs);
    materials[1].ReflectionCoef = 0.5;
    materials[1].RefractionCoef = 1.0;
    materials[1].MaterialType = DIFFUSE;
}
```

Теперь можно назначить для поверхностей различные материалы, и копировать значения материалов в функции Raytrace при пересечении в переменную ближайшего пересечения SIntersect intersect (там, где раньше мы поставили нули).

Теперь напишем функцию Phong.

```
vec3 Phong ( SIntersection intersect, SLight currLight)
{
    vec3 light = normalize ( currLight.Position - intersect.Point );
    float diffuse = max(dot(light, intersect.Normal), 0.0);
    vec3 view = normalize(uCamera.Position - intersect.Point);
    vec3 reflected= reflect( -view, intersect.Normal );
    float specular = pow(max(dot(reflected, light), 0.0), intersect.LightCoeffs.w);
    return intersect.LightCoeffs.x * intersect.Color +
        intersect.LightCoeffs.y * diffuse * intersect.Color +
        intersect.LightCoeffs.z * specular * Unit;
}
```

Чтобы «нарисовать» падающие тени необходимо выпустить, так называемые теневые лучи. Из каждой точки, для которой рассчитываем освещение выпускается луч на источник света, если этот луч пересекает какой-нибудь объект сцены, значит точка в тени и она освещена только ambient компонентой.

```
float Shadow(SLight currLight, SIntersection intersect)
{
    // Point is lighted
    float shadowing = 1.0;
    // Vector to the light source
    vec3 direction = normalize(currLight.Position - intersect.Point);
    // Distance to the light source
    float distanceLight = distance(currLight.Position, intersect.Point);
    // Generation shadow ray for this light source
    SRay shadowRay = SRay(intersect.Point + direction * EPSILON, direction);
    // ...test intersection this ray with each scene object
    SIntersection shadowIntersect;
    shadowIntersect.Time = BIG;
    // trace ray from shadow ray begining to light source position
    if(Raytrace(shadowRay, spheres, triangles, materials, 0, distanceLight,
                shadowIntersect))
    {
        // this light source is invisible in the intercection point
        shadowing = 0.0;
    }
    return shadowing;
}
```

Обратите внимание, что для вычисления пересечения используется та же функция Raytrace.

Этот вычисленный коэффициент необходимо передать параметром в функцию Phong и изменить вычисление цвета следующим образом:

```
return intersect.LightCoeffs.x * intersect.Color +
```

```
intersect.LightCoeffs.y * diffuse * intersect.Color * shadow +  
intersect.LightCoeffs.z * specular * Unit;
```

8. Зеркальное отражение

До этого моделировались исключительно объекты из материала диффузно рассеивающего свет. С помощью рейтрейсинга можно моделировать также зеркально отражающие объекты и прозрачные объекты.

Если в сцене есть зеркальный объект, это значит, что он не имеет собственного цвета, а отражает окружающие его объекты. Чтобы узнать какой итоговый цвет мы получим, необходимо выпустить из точки пересечения луч в сцену, согласно закону отражения (угол падения равен углу отражения).

Введем типы материалов: диффузное отражение и зеркальное отражение.

```
const int DIFFUSE_REFLECTION = 1;  
const int MIRROR_REFLECTION = 2;
```

Если луч пересекается с диффузным объектом, то вычисляется цвет объекта, а если с зеркальным, то создается новый зеркальный луч, который снова трассируется в сцену. Если зеркальных объектов в сцене много, то луч может переотразиться не один раз прежде чем пересечется с диффузным объектом. Т.к. в шейдерах запрещена рекурсия, введем стек для хранения лучей. На тот случай, если в сцене очень много зеркальных объектов и мало диффузных, настолько, что алгоритм имеет шанс зациклиться, введем ограничение на количество переотражений. Это ограничение называется глубиной трассировки.

Итак, создадим структуру TracingRay, содержащую луч, число depth, означающую номер переотражения, после которого этот луч был создан и contribution, для хранения вклада луча в результирующий цвет.

```
struct STracingRay  
{  
    SRay ray;  
    float contribution;  
    int depth;  
};
```

Создайте стек на основе массива, который умеет класть луч на стек (pushRay), брать луч со стека (popRay) и проверять есть ли ещё элементы в стеке (isEmpty).

Модифицируем функцию main. Первичный луч превращаем в луч, пригодный для стека и оборачиваем функцию Raytrace в цикла while.

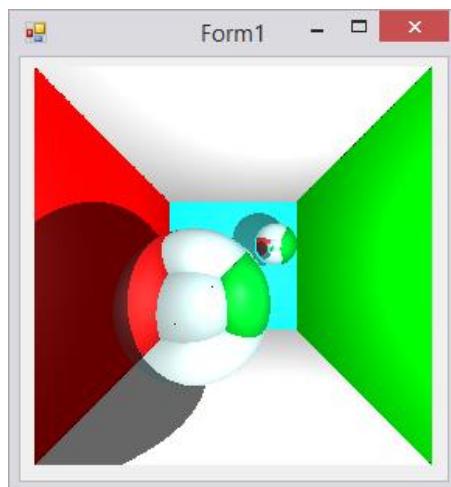
```
STracingRay trRay = STracingRay(ray, 1, 0);  
pushRay(trRay);  
while(!isEmpty())  
{  
    STracingRay trRay = popRay();  
    ray = trRay.ray;
```

```

SIntersection intersect;
intersect.Time = BIG;
start = 0;
final = BIG;
if (Raytrace(ray, start, final, intersect))
{
    switch(intersect.MaterialType)
    {
        case DIFFUSE_REFLECTION:
        {
            float shadowing = Shadow(uLight, intersect);
            resultColor += trRay.contribution * Phong ( intersect, uLight, shadowing );
            break;
        }
        case MIRROR_REFLECTION:
        {
            if(intersect.ReflectionCoef < 1)
            {
                float contribution = trRay.contribution * (1 -
intersect.ReflectionCoef);
                float shadowing = Shadow(uLight, intersect);
                resultColor += contribution * Phong(intersect, uLight, shadowing);
            }
            vec3 reflectDirection = reflect(ray.Direction, intersect.Normal);
            // creare reflection ray
            float contribution = trRay.contribution * intersect.ReflectionCoef;
            STracingRay reflectRay = STracingRay(
                SRay(intersect.Point + reflectDirection * EPSILON,
reflectDirection),
                contribution, trRay.depth + 1);
            pushRay(reflectRay);
            break;
        }
    } // switch
} // if (Raytrace(ray, start, final, intersect))
} // while(!isEmpty())

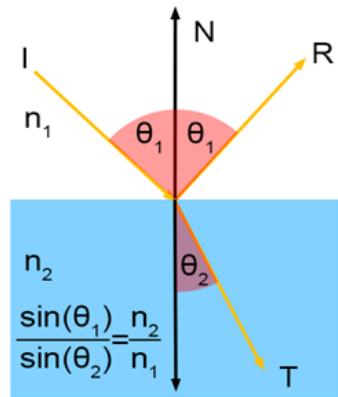
```

Запускаем, - при должной расстановке материалов должно получиться что-то подобное:



9. Добавление преломления

Для отрисовки прозрачных объектов сначала добавьте новый тип объекта REFRACTION и case в функции main, где будут обрабатываться объекты этого типа.



На границе двух прозрачных сред свет меняет направление своего распространения. Новое направление света зависит от угла падения луча и от коэффициентов преломления сред (index of refraction, ior, η).

Когда свет распространяется в вакууме, его скорость равна c . Когда свет распространяется в другой среде его скорость (v) уменьшается. Показатель преломления это отношения скорости света в среде к скорости света в вакууме $\eta = \frac{v}{c}$. Скорость света в воде выше, чем скорость света в стекле. $\eta_{water} = 1.3, \eta_{glass} = 1.5$.

Для вычисления вектора преломления можно воспользоваться функцией *refract*.

10. Дополнительные задания

Версия программы, представленная в методичке, может быть подвергнута расширению функционала, можно выполнить следующие задания:

1. Добавление геометрической фигуры куб;
2. Добавление геометрической фигуры тетраэдр;
3. Передача параметров объектов (цвет, прозрачность, зеркальная составляющая) в шейдер при помощи глобальных переменных и изменение их без перекомпиляции программы.
4. Изменение глубины рейтрейсинга;

Лабораторная работа №4. "Стеганография"

1. Введение в теорию

В данной лабораторной работе будет проводиться кодирование текстовой информации в изображения.

Одним из стеганографических методов является НЗБ, модификация наименьшего значимого бита.

Перед скрытием информация должна, как правило, подлежать кодированию. В рамках данной работы мы этот шаг опустим.

Для усложнения обнаружения факта скрытия информации исходное изображение зашумляют. Для зашумления младшим битам присваиваем значения 0 или 1, например, по равномерному закону распределения. Изображение может быть изначально зашумлено в связи с невысоким качеством.

Кроме того, для усложнения обнаружения факта скрытия информации проводят предварительный анализ распределения уже модифицированных битов, чтобы следуя этому же закону добавлять шум в оставшуюся часть изображения. Или же рекомендуется хотя бы продублировать часть скрытого сообщения так, чтобы заполнить всё изображение до конца.

Для исключения возможности сравнения оригинальной копии используемого изображения с закодированной копией и обнаружения таким образом факта кодирования, желательно, чтобы изображение было уникальным.

При обратном кодировании заранее неизвестно, какую часть пикселей занимает полезная информация. Декодируемое сообщение называется «квазисообщением». Установлено, что визуально не заметно внесение изменений и во второй из младших бит, особенно если в изображении отсутствуют большие однотонные участки. Таким образом, жертвуя ёмкостью изображения, можно увеличить его скрытность, записывая информацию поочерёдно в первый и второй младшие биты.

Текст полезной информации обычно ограничивают известными заранее метками, состоящими каждая из некоторого набора символов. Метки должны иметь достаточную длину, чтобы не путать их со случайными комбинациями символов, а также желательно формировать их с добавлением символов, достаточно хорошо разнесённых по ASCII-оси (например, вставка служебных символов или символов из кириллицы), для сложности их обнаружения.

Замечание. Иногда используют не только два младших бита, но и третий младший бит.

Глубина изображения

Изображения могут иметь различную битность. К примеру, **36/42/48-битная** цветность изображения означает, что на каждый цвет отводится: 12 бит / 14 бит / 16 бит (2 байта).

Кодировки

Наиболее универсальными и базовыми кодировками текстовых сообщений считаются такие, как ASCII, UTF-8. Мы будем работать с кодировкой ASCII, которая поддерживает символы латинского алфавита и знаки пунктуации.

Существует и множество других кодировок, в частности таких, которые позволяют работу с кириллицей, например koi8-r и windows-1251:

Info.CodePage 20866	Info.Name koi8-r	Info.DisplayName Cyrillic (KOI8-R)
1251	windows-1251	Cyrillic (Windows)

Одной из наиболее популярных кодировок для кириллицы является KOI8-R.

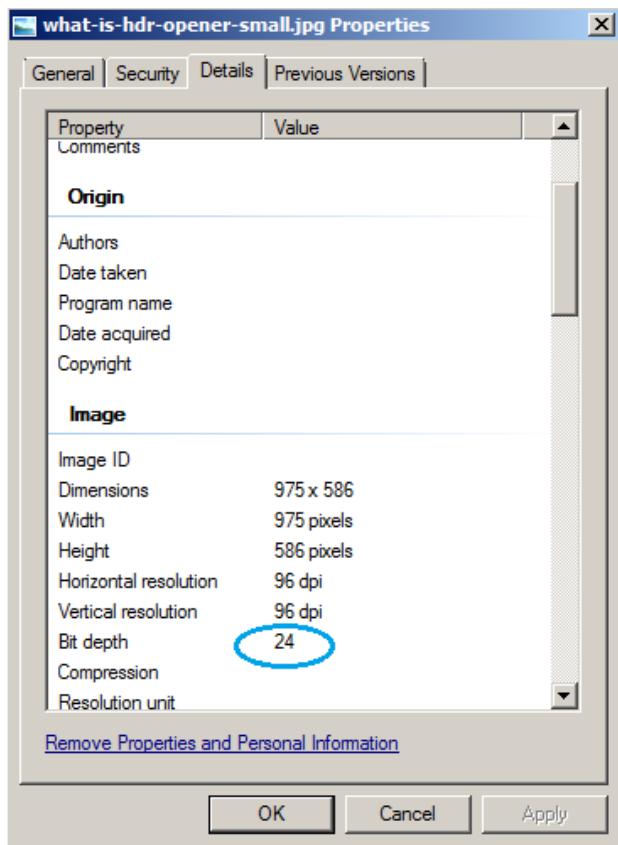
КОИ-8 (код обмена информацией, **8** бит), *KOI8* – восьмибитовая кодовая страница, совместимая с ASCII.

IETF (Internet Engineering Task Force, инженерный совет Интернета) утвердил RFC 1489 (Registration of a Cyrillic Character Set) по вариантам кодировки KOI-8.

2. Введение в лабораторную работу

Кодировать можно не только НЗБ изображения, но и НЗБ палитры изображения и т.д. Возможные способы преобразований НЗБ описаны в главе 5.3.2 книги [1], и других главах. Мы возьмём первый метод из главы 5.3.2 – раздел 5.3.2.1.

Первоначально нужно найти изображение нужного формата, с глубиной цвета 24 бит. Проверить этот параметр можно в свойствах изображения (ПКМ (правой клавишей мыши) по значку изображения, Свойства):

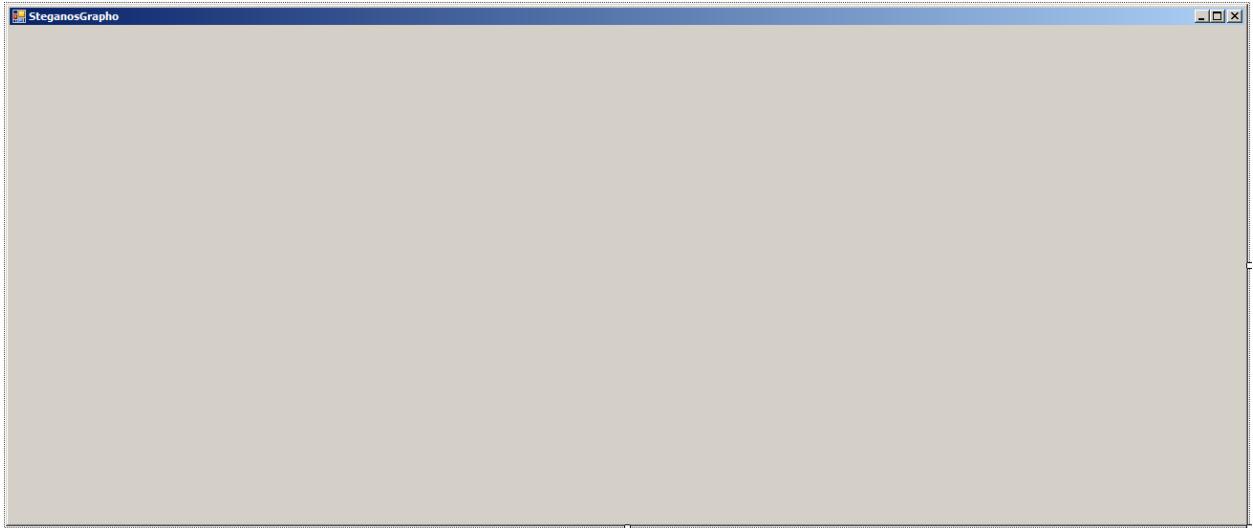


Текст полезной информации ограничим метками «St@rT» и «4IniSh».

Замечание: для работы со своими текстовыми файлами следует учесть, что они должны быть представлены в той кодировке, с которой вы работаете в программе. В нашем случае это ASCII.

3. Реализация

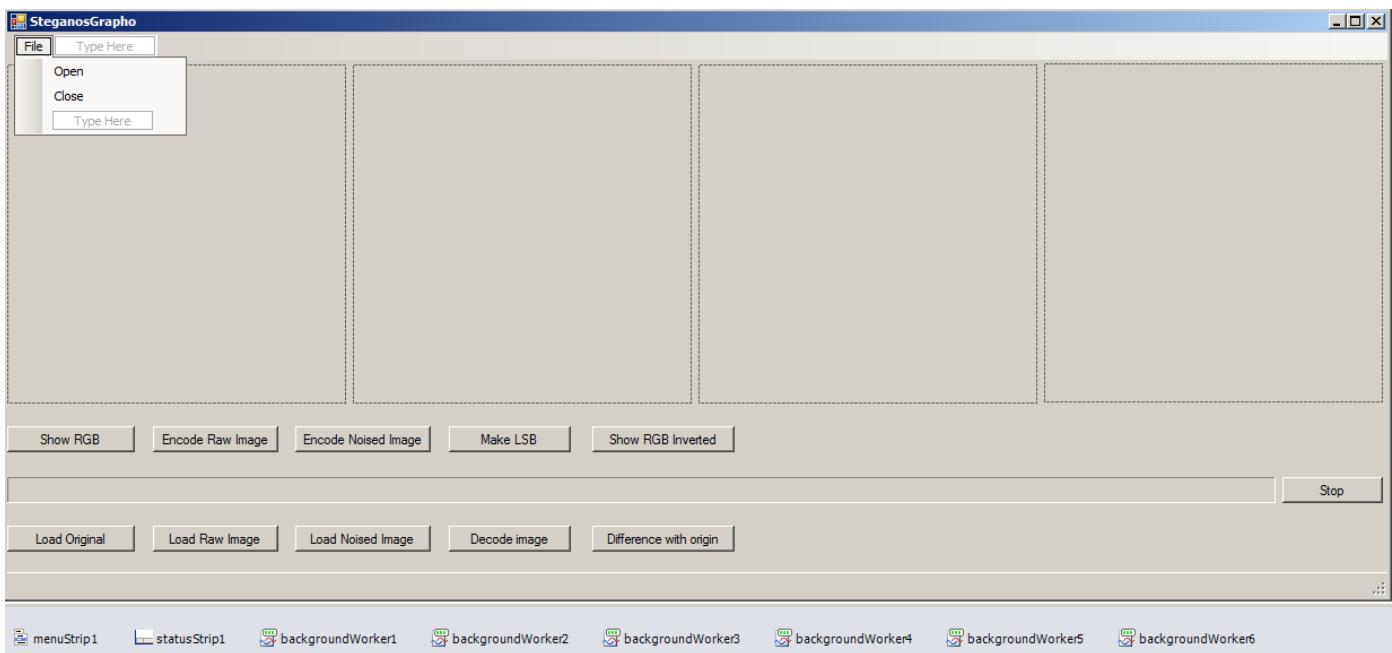
В среде разработки Visual Studio создадим проект, именованный SteganosGrapho (Visual C#, Windows Forms Application). Чтобы добавить графические элементы на форму, откройте Solution Explorer (Ctrl+Alt+L). В открывшемся окне выберите файл, содержащий код вашей формы (по умолчанию Form1.cs), по щелчку ПКМ выберите пункт View Designer (или нажмите Shift+F7), откроется окно с формой. Потяните за правый нижний маркер, чтобы увеличить размеры формы. Увеличим размеры окна интерфейса по вертикали и по горизонтали так, чтобы по вертикали оно имело достаточную высоту, а по горизонтали – ширину, близкую к ширине монитора. Далее, после добавления в интерфейс необходимых элементов, размеры окна можно будет модифицировать. Дадим сразу нашему приложению имя. Для этого следует нажать ЛКМ (левой клавишей мыши) на заголовке окна интерфейса «Form1». Откроется панель свойств. У параметра Text значение Form1 изменим на SteganosGrapho.



Нажмите Ctrl+Alt+X, чтобы открыть панель Toolbox. Из Toolbox будем последовательно добавлять перечисленные ниже элементы, перетаскивая их на окно формы:

1. `MenuStrip` (будет автоматически назван `menuStrip1`), для возможности открывать и сохранять изображения. Именуем меню как «File».
 - a. Создаём в нём элемент с именем `Open`,
 - b. Элемент с именем `Close`.
2. Четыре элемента `PictureBox`, в которых будут размещаться изображения (слева направо `pictureBox1`, `pictureBox2`, `pictureBox3`, `pictureBox4`). В свойствах каждого из них зададим им одинаковые размеры (параметр `size`, высота должна быть равна ширине), например 350, 350. Окно интерфейса отредактируем по ширине для соответствия правому краю крайнего справа `PictureBox`а`. Чтобы изображения подстраивались по размерам `PictureBox`ов` и умещались в них полностью, в свойстве `SizeMode` каждого `PictureBox`а` установите значение `Zoom`.
3. Одиннадцать клавиш `Button`. Расположение клавиш см. на рисунке ниже. Между горизонтальными рядами клавиш расположен элемент `ProgressBar` (будет добавлен позже).
 - a. Клавише `button1` дадим отображаемое в интерфейсе имя (параметр `Text`) «Show RGB»,
 - b. `button2` – «Stop»,
 - c. `button3` – «Encode Raw Image» (закодировать чистое изображение, т.е. не зашумлённое дополнительно),
 - d. `button4` – «Encode Noised Image» (закодировать зашумлённое изображение),
 - e. `button5` – «Load Original»,

- f. button6 – «Make LSB»,
 - g. button7 – «Decode image»,
 - h. button8 – «Load Raw Image»,
 - i. button9 – «Load Noised Image»,
 - j. button10 – «Show RGB Inverted» (каналы изображения в инвертированных цветах),
 - k. button11 – «Difference with origin»
4. ProgressBar (progressBar1) располагаем между рядами клавиш. Справа от него располагаем оставшуюся клавишу «Stop».
 5. Создаём элемент «статусная строка» statusStrip (statusStrip1). Он займёт положение в окне снизу. Используется для вывода пользователю на экран информации о состоянии приложения.
 - a. Нажатием на значок слева в статусной строке создаём элемент toolStripStatusLabel1, который необходим для отображения текста посредством StatusStrip. В свойствах инструмента toolStripStatusLabel1 очищаем значение параметра Text, убрав таким образом текст по умолчанию.
 6. Создаём шесть элементов BackgroundWorker для работы с потоками (backgroundWorker1-6).



Обеспечим наше приложение функциональностью, последовательно добавляя её для каждого элемента интерфейса.

В дизайнере интерфейса двойным щелчком ЛКМ по элементу меню Open создадим его обработчик:

```

private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
}

```

Добавим в него функциональность открытия файла изображения и его сохранения в памяти приложения (в переменной `image`):

```

private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    OpenFileDialog dialog = new OpenFileDialog();
    dialog.Filter = "Image files (*.png; *.jpg; *.bmp) | All files (*.*)";
    dialog.Title = "Open an Image File";

    if (dialog.ShowDialog() == DialogResult.OK)
    {
        image = new Bitmap(dialog.FileName);
        image_original = new Bitmap(dialog.FileName);
        pictureBox1.Image = image;
        pictureBox1.Refresh();
    }
}

```

Диалоговое окно позволит выбрать изображение, в которое мы будем кодировать информацию. При успешном выборе файла сохраним это изображение сразу в две переменные: `image` и `image_original`. Добавим их в класс интерфейса `Form1`:

```

public partial class Form1 : Form
{
    static Bitmap image;

    Bitmap image_original;
}

```

Переменная `image` в нашей программе будет хранить текущую рабочую копию обрабатываемого изображения. Это означает, что все изменения, которым будет подвергаться изображение, автоматически будут помещаться в эту переменную. В переменной `image_original` будет храниться каждое новое открытое изображение с помощью элемента меню `Open`.

Пробуем запустить приложение (Ctrl+F5) и убедиться, что при этом нет ошибок компиляции. Затем так же нажатием ЛКМ по элементу меню `Close` создаём его обработчик. Сохранение изображения из `image` на диск реализуйте самостоятельно (это дополнительное задание):

```

private void closeToolStripMenuItem_Click(object sender, EventArgs e)
{
}

```

Обеспечим возможность просмотра цветовых каналов изображения. Для этого в класс `Form1` добавим переменные, в которых будут храниться интенсивности каналов:

```

Bitmap image_r;
Bitmap image_g;
Bitmap image_b;

```

Вернёмся в дизайнер. ЛКМ откроем свойства `backgroundWorker1`. Параметрам `WorkerReportProgress` и `WorkerSupportsCancellation` установим значение `True`. Аналогичные значения параметров необходимо выставить и для всех остальных `backgroundWorker`ов`. переключаемся на вкладку `Events` (в этом же окне свойств), на которой расположены доступные события для элемента. Двойным щелчком ЛКМ создаём последовательно функции `DoWork`, `ProgressChanged` и `RunWorkerCompleted`. Для того чтобы заполнить их содержимое, создадим класс обработки изображений `ProcessImg`. Для этого создадим новый файл. Откроем окно `Solution Explorer` (`Ctrl+Alt+L`) и после нажатия ПКМ по имени проекта выберем `Add -> Class`. В открывшемся окне нужно ввести имя класса - `ProcessImg`. Класс `ProcessImg` будет представлять собой интерфейс, то есть класс, сам не несущий в себе смысловой нагрузки по обработке изображений, но предоставляющий возможность универсального обращения к его будущим классам-наследникам. Это означает, что его классы-наследники в одноимённых функциях будут выполнять какие-то реальные операции, а их базовый класс `ProcessImg` нужен только для того, чтобы позволить создавать все его классы-наследники единым образом (через указатель на `ProcessImg`, увидим это далее) и обращаться к его функциям (через указатель на него) тоже единым образом.

Зададим базовую часть классов обработки изображений, то есть сам класс `ProcessImg`. Для различных его классов-наследников нам понадобятся функции с такими наборами аргументов (их назначение будем рассматривать параллельно реализуемому функционалу):

```

abstract class ProcessImg
{
    public virtual Bitmap get_image_r() { return null; }
    public virtual Bitmap get_image_g() { return null; }
    public virtual Bitmap get_image_b() { return null; }
    public virtual Bitmap get_image_raw() { return null; }
    public virtual Bitmap get_image_noised() { return null; }
    public virtual Bitmap get_image_LSB() { return null; }
    public virtual Bitmap get_image_diff() { return null; }

    protected virtual Color calculateNewPixelColor(Bitmap sourceImage,
        int x, int y)
    {
        return sourceImage.GetPixel(x, y);
    }

    protected virtual Color calculateNewPixelColor(Bitmap currImage,
        Bitmap originImage, int x, int y)
    {
        Color currColor = currImage.GetPixel(x, y);
        return Color.FromArgb(currColor.R, currColor.G, currColor.B);
    }

    public virtual Bitmap processImage(Bitmap currImage, Bitmap originImage,
        BackgroundWorker worker)
    {

```

```

        return currImage;
    }

    public virtual Bitmap processImage(Bitmap sourceImage,
        ref BackgroundWorker worker)
    {
        return sourceImage;
    }

    public virtual void processImage(Bitmap sourceImage,
        ref BackgroundWorker worker, bool no_return_value)
    {
        return;
    }

    public int Clamp(int value, int min, int max)
    {
        if (value < min)
            return min;
        if (value > max)
            return max;
        return value;
    }
}

```

Сейчас в данном классе не различаются слова, предназначенные для работы с изображениями и потоками, что приводит к ошибкам компиляции. Для того чтобы сделать их видимыми, добавим в начало файла следующие строки, подключающие нужные модули платформы .Net:

```

using System.Drawing;
using System.ComponentModel;

```

Как видно, все методы класса `ProcessImg`, кроме метода `Clamp` представляют собой заглушки. Они будут переопределены в соответствующих классах-наследниках. Метод `Clamp` позволит отсекать значения какого-либо числового диапазона, делая его допустимым, без выхода за допустимые границы значений.

Модификатор `protected` полей (переменных) и методов (функций) класса говорит о том, что поле (или метод) класса может использоваться только методами самого класса, а также методами его наследников; модификатор `public` – о том, что поле (или метод) класса может использоваться объектами любых классов; и `private` – о том, что поле (или метод) класса может использоваться только методами самого класса.

Добавим к классу `ProcessImg` слово `abstract`, которое не запретит создание объекта этого класса, но будет сигнализировать о том, что данный класс был создан не для создания его собственных объектов. Абстрактных функций в классе `ProcessImg` не будет, а будут только виртуальные, что позволит переопределять эти функции не в каждом наследнике, а только в тех, которые используют определённую подгруппу из всего набора функций класса `ProcessImg`. Данный подход используется в шаблонах объектно-ориентированного проектирования.

Отображение цветовых каналов

Объявим первые классы-наследники, которые и будут преобразовывать исходное изображение в три отдельных изображения, представляющих каждый свой канал исходного. Основной из них — `FormChannels`. Он будет хранить в своих полях `image_r`, `image_g` и `image_b` интенсивности каналов изображения. Даём им модификатор `protected` потому, что у класса `FormChannels` будет наследник, который также будет использовать переменные для хранения трёх цветовых каналов. Методы `get_image_r`, `get_image_g` и `get_image_b` позволяют получить доступ к названным выше полям извне. Эти методы, а также метод `processImage` переопределены необходимые здесь одноимённые методы базового абстрактного класса `ProcessImg`.

```
class FormChannels : ProcessImg
{
    protected Bitmap image_r;
    protected Bitmap image_g;
    protected Bitmap image_b;

    public override Bitmap get_image_r() { return image_r; }
    public override Bitmap get_image_g() { return image_g; }
    public override Bitmap get_image_b() { return image_b; }

    public override void processImage(Bitmap sourceImage,
        ref BackgroundWorker worker, bool no_return_value)
    {
        ProcessImg prc_im_r = new FormChannelR();
        image_r = prc_im_r.processImage(sourceImage, ref worker);
        ProcessImg prc_im_g = new FormChannelG();
        image_g = prc_im_g.processImage(sourceImage, ref worker);
        ProcessImg prc_im_b = new FormChannelB();
        image_b = prc_im_b.processImage(sourceImage, ref worker);
    }
}
```

В методе `ProcessImg`, который обеспечивает обработку изображения, для получения каждого из каналов вызывается отдельный класс (описаны ниже). Именно здесь представлен один из примеров создания объектов классов-наследников (`FormChannelR`, `FormChannelG` и `FormChannelB`) на указателях на базовый класс (`ProcessImg`) и обращения к созданным объектам через эти указатели.

```
class FormChannelR : FormChannelX
{
    public override Bitmap processImage(Bitmap sourceImage,
        ref BackgroundWorker worker)
    {
        return processImage(sourceImage, ref worker, 33, 0);
    }

    protected override Color calculateNewPixelColor(Bitmap sourceImage,
        int x, int y)
    {
        Color sourceColor = sourceImage.GetPixel(x, y);
        return Color.FromArgb(sourceColor.R, sourceColor.G, sourceColor.B);
    }
}
```

```

class FormChannelG : FormChannelX
{
    public override Bitmap processImage(Bitmap sourceImage,
        ref BackgroundWorker worker)
    {
        return processImage(sourceImage, ref worker, 33, 33);
    }

    protected override Color calculateNewPixelColor(Bitmap sourceImage,
        int x, int y)
    {
        Color sourceColor = sourceImage.GetPixel(x, y);
        return Color.FromArgb(sourceColor.G, sourceColor.G, sourceColor.G);
    }
}

class FormChannelB : FormChannelX
{
    public override Bitmap processImage(Bitmap sourceImage,
        ref BackgroundWorker worker)
    {
        return processImage(sourceImage, ref worker, 33, 66);
    }

    protected override Color calculateNewPixelColor(Bitmap sourceImage,
        int x, int y)
    {
        Color sourceColor = sourceImage.GetPixel(x, y);
        return Color.FromArgb(sourceColor.B, sourceColor.B, sourceColor.B);
    }
}

```

Все они в свою очередь наследуются от обобщающего класса FormChannelX. Это делается для уменьшения объёма повторяющегося кода, который всегда может повлечь множество ошибок при его написании и модификации.

```

class FormChannelX : ProcessImg
{
    public Bitmap processImage(Bitmap sourceImage, ref BackgroundWorker worker,
        int mult, int summand)
    {
        Bitmap resultImage = new Bitmap(sourceImage.Width, sourceImage.Height);
        for (int i = 0; i < sourceImage.Width; i++)
        {
            worker.ReportProgress(Clamp((int)((float)i / resultImage.Width
                * mult + summand), 0, 100));
            if (worker.CancellationPending)
                return null;
            for (int j = 0; j < sourceImage.Height; j++)
            {
                resultImage.SetPixel(i, j, calculateNewPixelColor(sourceImage,
                    i, j));
            }
        }
        return resultImage;
    }
}

```

В методах processImage классов FormChannelR, FormChannelG И FormChannelB вызывается метод processImage как раз того типа (с определённым набором

аргументов), который переопределён в классе `FormChannelX`. Мы ему передаём данные для инструмента `BackgroundWorker`, которые нужны для верного отображения завершённости работы. Так как мы последовательно вычисляем три канала, то суммарную готовность условно делим на три части, и эти данные в процентах передаём в `ReportProgress`.

Теперь в соответствующий поток добавим создание объекта класса `FormChannels` и вызов его реализации функции `processImage`. Для этого в созданный ранее обработчик `backgroundWorker1_DoWork` добавим:

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    ProcessImg prc_im = (ProcessImg)e.Argument;
    prc_im.processImage(image, ref backgroundWorker1, true);
    if (backgroundWorker1.CancellationPending != true)
    {
        image_r = prc_im.get_image_r();
        image_g = prc_im.get_image_g();
        image_b = prc_im.get_image_b();
    }
}
```

Мы видим, что здесь фигурирует переменная `image`. Именно её содержимое здесь пойдёт на разделение по каналам. Если выполнение потока не будет прервано, то результат разделения изображения по каналам будет занесён в переменные `image_r`, `image_g` и `image_b`. В обработчик изменения прогресса выполнения операции потока добавим присвоение прогресса потока инструменту визуального отображения прогресса в виде бегущей линии:

```
private void backgroundWorker1_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}
```

В обработчик окончания работы потока добавим присвоение новых значений переменных `image_r`, `image_g` и `image_b` соответствующим `pictureBox`ам` и сразу обновим их для отображения. В случае успеха или прерывания работы потока выдадим соответствующие сообщения. Обнулим значение бегунка прогресса:

```
private void backgroundWorker1_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        pictureBox2.Image = image_r;
        pictureBox2.Refresh();
        pictureBox3.Image = image_g;
        pictureBox3.Refresh();
        pictureBox4.Image = image_b;
        pictureBox4.Refresh();
        toolStripStatusLabel1.Text =
            "Channels` retrieving has just completed";
    }
}
```

```

        }
    else
    {
        toolStripStatusLabel1.Text = "Channels` retrieving was cancelled";
    }
    progressBar1.Value = 0;
}

```

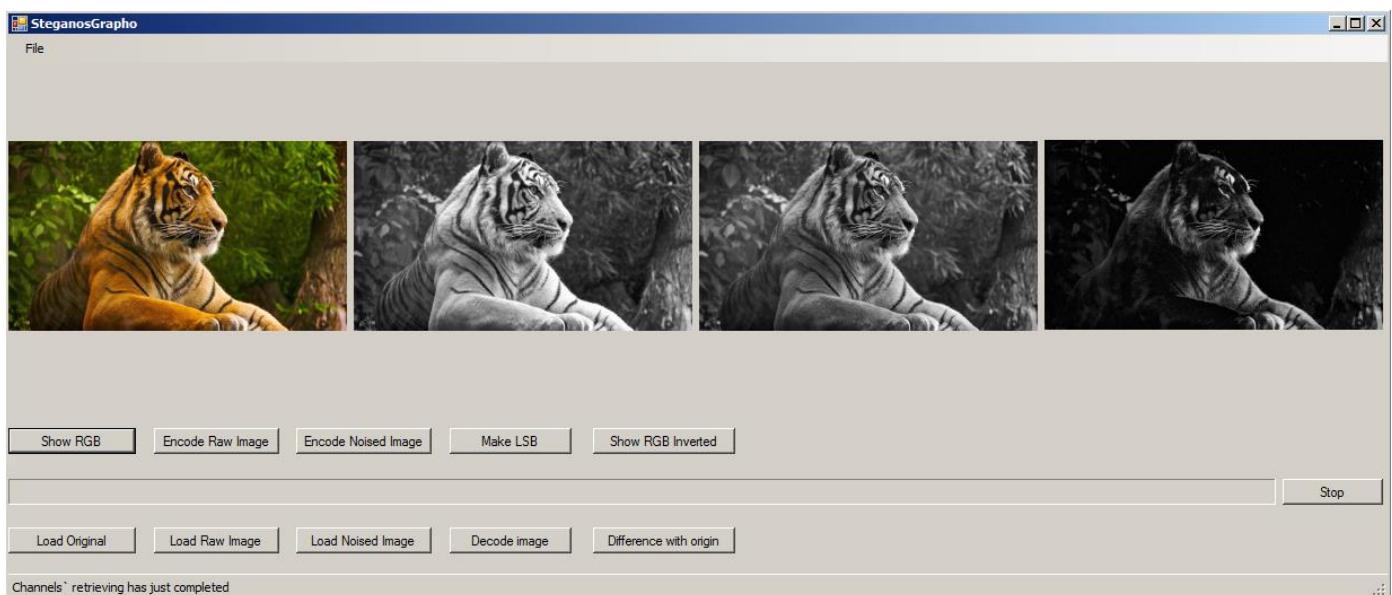
Осталось прикрепить вызов данного функционала к соответствующей клавише. В дизайнере двойным щелчком ЛКМ по клавише «Show RGB» создадим её обработчик, в котором по нажатию клавиши будет создаваться объект класса FormChannels, работу с которым продолжит поток backgroundWorker1:

```

// Show RGB
private void button1_Click(object sender, EventArgs e)
{
    ProcessImg prc_im = new FormChannels();
    backgroundWorker1.RunWorkerAsync(prc_im);
}

```

Запустим приложение. Откроем изображение tiger.bmp и нажмём «Show RGB». Должен получиться такой результат:



Добавим возможность прерывания работы потоков. В дизайнере нажатием ЛКМ на клавишу «Stop» получим её обработчик и добавим в него код:

```

// Stop
private void button2_Click(object sender, EventArgs e)
{
    backgroundWorker1.CancelAsync();
    backgroundWorker2.CancelAsync();
    backgroundWorker3.CancelAsync();
    backgroundWorker4.CancelAsync();
    backgroundWorker5.CancelAsync();
    backgroundWorker6.CancelAsync();
}

```

```
}
```

Теперь в процессе выполнения любого из этих потоков нажатием данной клавиши его можно прервать. Прерван будет тот поток, который работает в данный момент.

Адаптированное отображение цветовых каналов

Так как глаз человека практически не чувствителен к незначительным изменениям интенсивностей, нам понадобится ещё один режим просмотра интенсивностей по каналам.

В ProcessImg.cs создадим класс-наследник от FormChannels. Назовём его FormChannelsInverted. Класс FormChannelsInverted наследуется от класса FormChannels, так что первый имеет свою собственную надстройку над вторым в виде собственной реализации метода processImage, но остальное его содержимое полностью повторяет содержимое класса FormChannels. Это означает, что экземпляры (объекты) класса FormChannelsInverted также будут содержать в себе переменные переменные image_r, image_g и image_b и возможность доступа к ним:

```
class FormChannelsInverted : FormChannels
{
    public override void processImage(Bitmap sourceImage,
        ref BackgroundWorker worker, bool no_return_value)
    {
        ProcessImg prc_im_r = new FormChannelR_Inverted();
        image_r = prc_im_r.processImage(sourceImage, ref worker);
        ProcessImg prc_im_g = new FormChannelG_Inverted();
        image_g = prc_im_g.processImage(sourceImage, ref worker);
        ProcessImg prc_im_b = new FormChannelB_Inverted();
        image_b = prc_im_b.processImage(sourceImage, ref worker);
    }
}
```

Изменения заключаются в том, что будут вызываться другие классы-обработчики каналов. Добавим сюда и их:

```
class FormChannelR_Inverted : FormChannelR
{
    protected override Color calculateNewPixelColor(Bitmap sourceImage,
        int x, int y)
    {
        Color sourceColor = sourceImage.GetPixel(x, y);
        int color = (sourceColor.R > 0) ? 0 : 255;
        return Color.FromArgb(color, color, color);
    }
}

class FormChannelG_Inverted : FormChannelG
{
    protected override Color calculateNewPixelColor(Bitmap sourceImage,
        int x, int y)
    {
        Color sourceColor = sourceImage.GetPixel(x, y);
        int color = (sourceColor.G > 0) ? 0 : 255;
        return Color.FromArgb(color, color, color);
    }
}
```

```

}

class FormChannelB_Inverted : FormChannelB
{
    protected override Color calculateNewPixelColor(Bitmap sourceImage,
        int x, int y)
    {
        Color sourceColor = sourceImage.GetPixel(x, y);
        int color = (sourceColor.B > 0) ? 0 : 255;
        return Color.FromArgb(color, color, color);
    }
}

```

Все эти классы наследуются от соответствующих классов обработки каналов, поэтому отличаться они от последних будут только переопределёнными методами `calculateNewPixelColor`. Их логика заключается в том, что чёрным цветом (присваиваем значение 0) будут отображаться интенсивности каналов, чьё оригинальное значение больше нуля, и белым (значение 255) – только те, значение интенсивности которых – ноль.

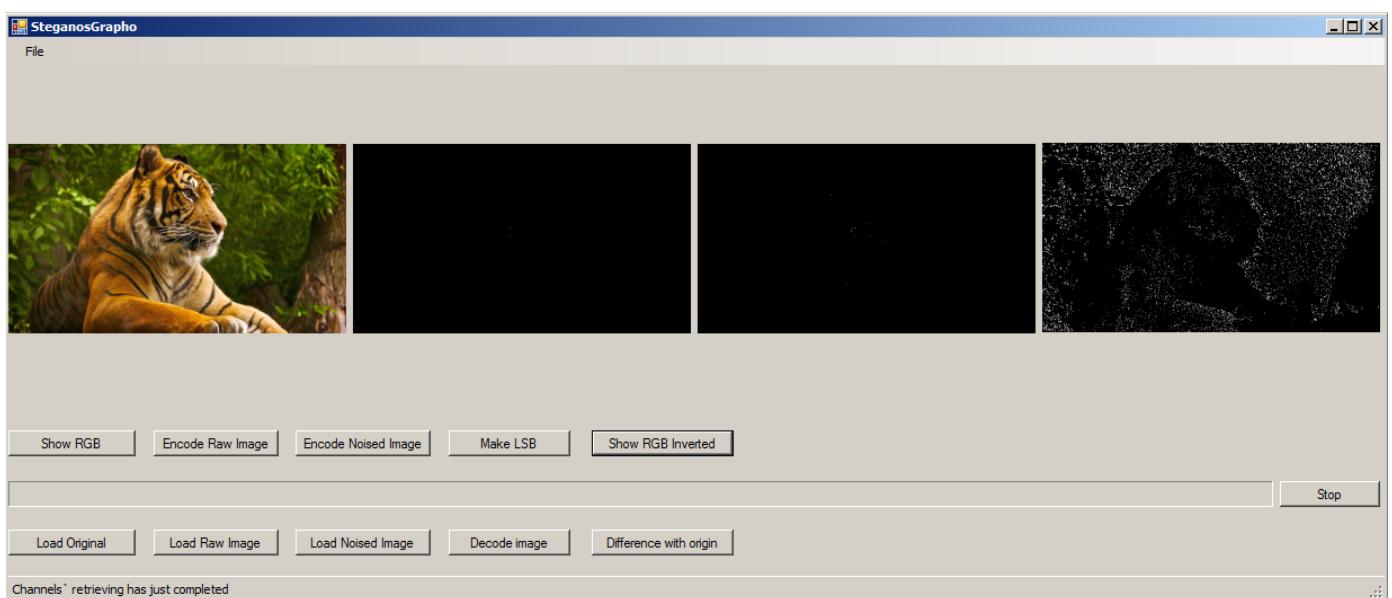
Поскольку возвращаемые значения классов `FormChannels` и `FormChannelsInverted` одинаковые, для вычисления инвертированных интенсивностей каналов будем использовать тот же `backgroundWorker1`. В дизайнере двойным нажатием ЛКМ по клавише «Show RGB Inverted» создадим её обработчик и заполним его для работы с `FormChannelsInverted`:

```

// Show RGB Inverted
private void button10_Click(object sender, EventArgs e)
{
    ProcessImg prc_im = new FormChannelsInverted();
    backgroundWorker1.RunWorkerAsync(prc_im);
}

```

Запустим приложение и, нажав «Show RGB Inverted» после открытия того же файла, увидим результат:



Особенно полезной эта функциональность будет при работе с наименьшими значащими битами.

Выделение НЗБ

Научимся их просматривать. Для этого тем же способом, что описан выше, создадим в отдельном файле ещё один класс ProcessBytes, который будет содержать в себе функционал побайтовой обработки интенсивностей каналов. Прежде всего, добавим в него метод выделения НЗБ:

```
public Bitmap MakeLSB(Bitmap src, ref BackgroundWorker worker)
{
    Bitmap res = new Bitmap(src.Width, src.Height);
    for (int i = 0; i < src.Width; i++)
    {
        worker.ReportProgress((int)((float)i / src.Width * 100));
        if (worker.CancellationPending)
            return null;
        for (int j = 0; j < src.Height; j++)
        {
            Color pixelColor = src.GetPixel(i, j);

            BitArray colorArray = Byte2Bit(pixelColor.R);
            colorArray[3] = false; // т.е. 0
            colorArray[4] = false;
            colorArray[5] = false;
            colorArray[6] = false;
            colorArray[7] = false;
            byte newR = Bit2Byte(colorArray);

            colorArray = Byte2Bit(pixelColor.G);
            colorArray[3] = false;
            colorArray[4] = false;
            colorArray[5] = false;
            colorArray[6] = false;
            colorArray[7] = false;
            byte newG = Bit2Byte(colorArray);

            colorArray = Byte2Bit(pixelColor.B);
            colorArray[3] = false;
            colorArray[4] = false;
            colorArray[5] = false;
            colorArray[6] = false;
            colorArray[7] = false;
            byte newB = Bit2Byte(colorArray);

            Color newColor = Color.FromArgb(newR, newG, newB);
            res.SetPixel(i, j, newColor);
        }
    }
    return res;
}
```

Здесь для всякого пикселя изображения значение каждого его канала записывается в массив бит (bitArray) и в нём зануляются все старшие биты (в данном случае пять старших бит). Младшие биты (здесь – с нулевого по второй) остаются в первоначальном виде. После внесения изменений в биты

для каждого пикселя собираем их снова в байты для присвоения на прежнюю позицию в изображении.

Чтобы необходимые модули .Net стали видимыми, добавим строки:

```
using System.Drawing;
using System.ComponentModel;
using System.Collections;
```

Методы `Byte2Bit` и `Bit2Byte` – это наши собственные методы перевода 1) байт (в которых представлены интенсивности в пикселях изображения) в биты для их побитовой обработки и 2) бит пикселов обратно в байты. Сразу добавим эти методы в класс, а также добавим метод `clamp`, который пригодится и в этом классе:

```
private BitArray Byte2Bit(byte src)
{
    BitArray bitArray = new BitArray(8);
    bool bit = false;
    for (int i = 0; i < 8; i++)
    {
        if ((src >> i & 1) == 1)
        {
            bit = true;
        }
        else bit = false;
        bitArray[i] = bit;
    }
    return bitArray;
}

private byte Bit2Byte(BitArray src)
{
    byte value = 0;
    for (int i = 0; i < src.Count; i++)
        if (src[i] == true)
            value += (byte)Math.Pow(2, i);
    return value;
}

public int Clamp(int value, int min, int max)
{
    if (value < min)
        return min;
    if (value > max)
        return max;
    return value;
}
```

Привяжем новую функциональность к соответствующей клавише. В дизайнере двойным нажатием ЛКМ на «**Make LSB**» получим обработчик этой клавиши и добавим в него:

```
// LSB
private void button6_Click(object sender, EventArgs e)
{
    ProcessImg prc_im = new LSB4Img();
    backgroundWorker5.RunWorkerAsync(prc_im);
```

```
}
```

Класс `LSB4Img`, участвующий в обработке НЗБ, необходимо добавить в `ProcessImg.cs`:

```
class LSB4Img : ProcessImg
{
    Bitmap image_LSB;

    public override Bitmap get_image_LSB() { return image_LSB; }

    public override Bitmap processImage(Bitmap sourceImage,
        ref BackgroundWorker worker)
    {
        process_bytes = new ProcessBytes();
        Bitmap resultImage = process_bytes.MakeLSB(sourceImage, ref worker);
        image_LSB = resultImage;
        return resultImage;
    }
}
```

Для того чтобы объекты класса `ProcessImg` могли использовать функциональность класса `ProcessBytes`, объект последнего создадим в виде поля класса `ProcessImg`:

```
protected ProcessBytes process_bytes;
```

Как вы уже заметили, обработкой НЗБ будет заниматься поток `backgroundWorker5`. Через его свойства в дизайнере во вкладке `Events` двойным щелчком ЛКМ создаём обработчики `DoWork`, `ProgressChanged` и `RunWorkerCompleted` и заполняем их:

```
// LSB
private void backgroundWorker5_DoWork(object sender, DoWorkEventArgs e)
{
    ProcessImg prc_im = (ProcessImg)e.Argument;
    prc_im.processImage(image, ref backgroundWorker5);
    if (backgroundWorker5.CancellationPending != true)
    {
        image = prc_im.get_image_LSB();
    }
}

private void backgroundWorker5_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}

private void backgroundWorker5_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        pictureBox1.Image = image;
        pictureBox1.Refresh();
        toolStripStatusLabel1.Text = "LSB retrieving has just completed";
    }
}
```

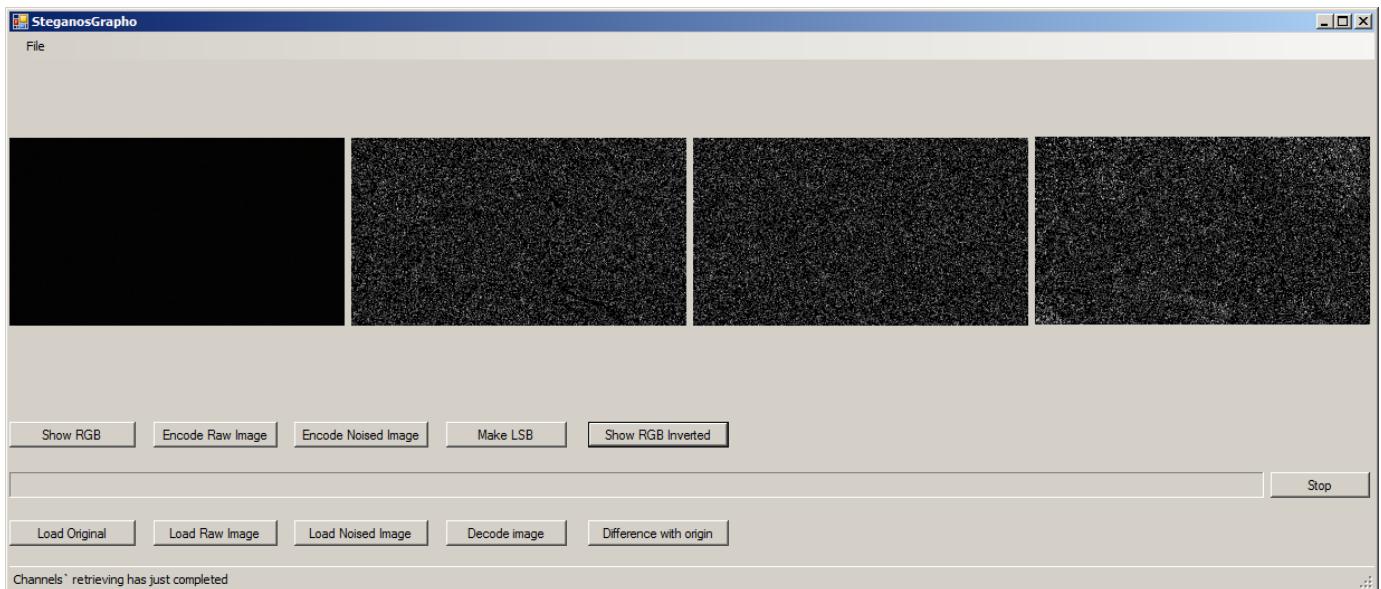
```

    }
else
{
    toolStripStatusLabel1.Text = "LSB retrieving was cancelled";
}
progressBar1.Value = 0;
}

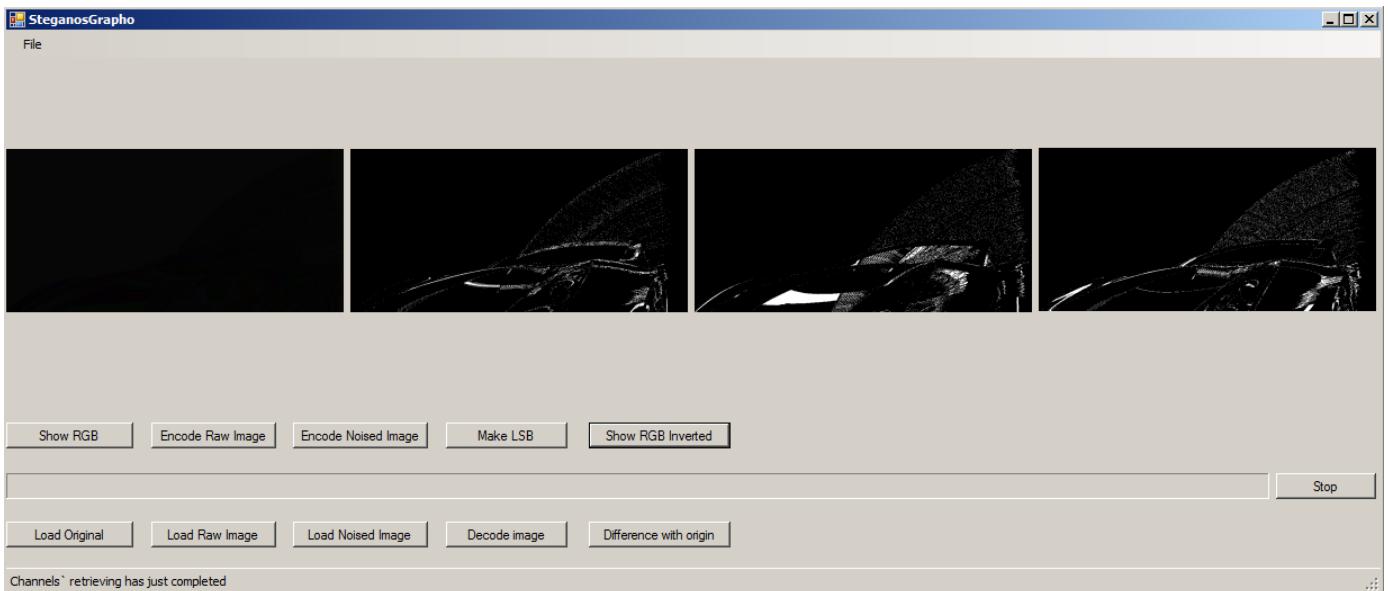
```

В `Dowork` по окончании работы, если поток не был прерван, результат запишем в рабочую копию изображения, `image`. `ProgressChanged` для всех потоков имеет одинаковую функциональность. `RunWorkerCompleted` здесь, также если поток не был прерван, присвоит результат в первый из четырёх `pictureBox`ов` и обновит его, так что мы сразу увидим в нём изменённое изображение. В случае прерывания работы потока, а также его успешной работы выведем в статусную строку соответствующее сообщение. Не забывайте, что для запуска очередного потока необходимо дожидаться таких сигналов о завершении предыдущего.

Запускаем приложение. Снова открываем картинку и жмём «Make LSB». В основном `pictureBox`е` увидим чёрное изображение, так как наименьшие биты слишком малы для их фиксации глазом человека и практически не отличны глазом от нулевой интенсивности, то есть от чёрного цвета. После нажатия «Show RGB» то же самое будем наблюдать и в остальных `pictureBox`ах`. «Show RGB Inverted», в свою очередь, покажет для данного изображения его зашумлённость, поскольку младшие биты как раз имеют значение на уровне шума, если он есть в изображении:



Теперь воспроизведём последовательность «Make LSB», «Show RGB Inverted» для картинки `aboutus-rightbottomv3.png`. Так как она имеет искусственную природу, шум в ней отсутствует, и даже на уровне НЗБ мы видим оригинальные контуры:



Использовать такие изображения для кодирования информации в общем случае не выгодно, поскольку изменения могут быть очень легко обнаружены. Для таких и не только изображений применяют зашумление изображения на уровне НЗБ для скрытия факта записи информации.

Возвращение оригинального изображения

Если мы снова захотим работать с неизменённым последним открытым изображением, это можно будет осуществить через его загрузку из `image_original` в рабочую копию `image`. Для этого двойным нажатием ЛКМ по клавише «Load Original» создадим её обработчик и в нём разместим:

```
private void button5_Click(object sender, EventArgs e)
{
    if (image_original == null)
    {
        MessageBox.Show("Original image is empty!");
        return;
    }
    image = image_original;
    pictureBox1.Image = image;
    pictureBox1.Refresh();
}
```

Именно с этой целью ранее была предусмотрена переменная `image_original`, в которую при каждом открытии нового изображения, оно также сохраняется. Вначале осуществим проверку безопасности на пустоту переменной. В случае её непустоты копируем её значение в `image` и обновляем соответствующий `pictureBox`.

Проверим функционал. Откроем изображение, изменим `image` запуском «Make LSB» и кликнем «Load Original».

Просмотр разницы изменённого и оригинального изображений

Сразу добавим возможность попиксельного сравнения двух изображений: хранящихся в переменных `image` и `image_original`. В дизайнере двойным нажатием ЛКМ по клавише «Difference with origin» создадим её обработчик и в него добавим:

```
private void button11_Click(object sender, EventArgs e)
{
    ProcessImg prc_im = new DiffWithOrigin();
    backgroundWorker6.RunWorkerAsync(prc_im);
}
```

В `ProcessImg.cs` добавим класс `DiffWithOrigin`:

```
class DiffWithOrigin : ProcessImg
{
    Bitmap diff_img;

    public override Bitmap get_image_diff() { return diff_img; }

    protected override Color calculateNewPixelColor(Bitmap currImage,
        Bitmap originImage, int x, int y)
    {
        Color currColor = currImage.GetPixel(x, y);
        Color originColor = originImage.GetPixel(x, y);
        return Color.FromArgb(Math.Abs(currColor.R - originColor.R),
            Math.Abs(currColor.G - originColor.G),
            Math.Abs(currColor.B - originColor.B));
    }

    public override Bitmap processImage(Bitmap currImage, Bitmap originImage,
        BackgroundWorker worker)
    {
        Bitmap resultImage = new Bitmap(currImage.Width, currImage.Height);
        for (int i = 0; i < currImage.Width; i++)
        {
            worker.ReportProgress((int)((float)i / resultImage.Width * 100));
            if (worker.CancellationPending)
                return null;
            for (int j = 0; j < currImage.Height; j++)
            {
                resultImage.SetPixel(i, j, calculateNewPixelColor(currImage,
                    originImage, i, j));
            }
        }
        diff_img = resultImage;
        return resultImage;
    }
}
```

Аналогично другим классам, `DiffWithOrigin` сохраняет в себе результат своей работы (`diff_img`) для того чтобы впоследствии передать его в `Form1`. Связка переопределённых здесь методов `processImage` и `calculateNewPixelColor` обеспечивает попиксельное вычисление разницы между двумя заданными изображениями.

Добавим функциональность потоку `backgroundWorker6`, в рамках которого будет осуществляться эта операция. В его свойствах в дизайнере во вкладке

Events двойным щелчком ЛКМ создаём обработчики `DoWork`, `ProgressChanged` и `RunWorkerCompleted` и заполняем их:

```
// Diff
private void backgroundWorker6_DoWork(object sender, DoWorkEventArgs e)
{
    ProcessImg prc_im = (ProcessImg)e.Argument;
    prc_im.processImage(image, image_original, backgroundWorker6);
    if (backgroundWorker6.CancellationPending != true)
    {
        diff_img = prc_im.get_image_diff();
        image = diff_img;
    }
}

private void backgroundWorker6_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}

private void backgroundWorker6_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        pictureBox1.Image = image;
        pictureBox1.Refresh();
        toolStripStatusLabel1.Text = "Computing of the difference against original
            image has just completed";
    }
    else
    {
        toolStripStatusLabel1.Text = "Computing of the difference against original
            image was cancelled";
    }
    progressBar1.Value = 0;
}
```

В поля класса `Form1` необходимо добавить переменную:

```
Bitmap diff_img;
```

В неё и в рабочую копию `image` будет записан результат работы в `DoWork`. В `RunWorkerCompleted` результат присвоим главному `pictureBox`'у и обновим его, чтобы сразу увидеть изменения.

«Чистое» кодирование информации

Приступим к кодированию информации в изображение. Вначале закодируем её «как есть», без дополнительной обработки. В дизайнере двойным нажатием ЛКМ по клавише «Encode Raw Image» создадим её обработчик и в него добавим логику для кодирования «сырого» изображения:

```
// Encode raw
private void button3_Click(object sender, EventArgs e)
{
    OpenFileDialog dialog = new OpenFileDialog();
```

```

dialog.Filter = "Text files (*.txt)|*.txt|All files(*.*)|*.*";
dialog.Title = "Open a Text File";

if (dialog.ShowDialog() == DialogResult.OK)
{
    src_text_file = dialog.FileName;
}
ProcessImg prc_im = new EncodeRawImg();
backgroundWorker2.RunWorkerAsync(prc_im);
}

```

В поля класса Form1 добавим переменную:

```
static string src_text_file;
```

Переменная создана статической для того чтобы её содержимое не терялось на протяжении всего запуска приложения.

В ProcessImg.cs создадим класс EncodeRawImg:

```

class EncodeRawImg : ProcessImg
{
    Bitmap image_raw;

    public override Bitmap get_image_raw() { return image_raw; }

    public override Bitmap processImage(Bitmap sourceImage,
        ref BackgroundWorker worker)
    {
        process_bytes = new ProcessBytes();
        image_raw = process_bytes.EncodeRawImage(sourceImage, ref worker);
        return image_raw;
    }
}

```

В EncodeRawImg вызывается одна из функций обработки бит – EncodeRawImage (из класса ProcessBytes), которая осуществляет «чистое» кодирование заданного текста в изображение.

В класс ProcessBytes добавим метод EncodeRawImage:

```

public Bitmap EncodeRawImage(Bitmap src, ref BackgroundWorker worker)
{
    ReadText(ref worker);
    if (input_text_byte_list == null)
    {
        Form1.ShowMessageBox("List is empty!");
        return src;
    }

    Bitmap res = new Bitmap(src);
    int i = 0, j = 0;

    // write text into image
    bool stop = false;
    int text_ind = 0;
    for (; i < src.Width; i++)

```

```

    {
        worker.ReportProgress(Clamp((int)((float)i / src.Width * 50 + 50),
            0, 100));
        if (worker.CancellationPending)
            return null;
        for (j = 0; j < src.Height; j++)
        {
            if (text_ind == text_size)
            {
                stop = true;
                break;
            }
            EncodeSymbol(ref src, ref res, ref i, ref j, ref text_ind,
                input_text_byte_list[text_ind]);
        }
        if (stop) break;
    }
    return res;
}

```

В начале метода `EncodeRawImage` вызывается функция `ReadText` (представлена ниже). Она читает содержимое текстового файла, который мы будем кодировать, и записывает его в массив байт `input_text_byte_list`. После чтения текста проверяем массив на пустоту. Если он пуст, выводим на экран сообщение об этом. В противном случае начинаем кодирование. На процесс кодирования функции `ReportProgress` оставляем только 50%, которые будут заполняться по мере готовности операции, начиная с половины прогресса. 50% - это условная величина. Первую половину прогресса заполняет функция `ReadText`, что мы увидим далее. В двойном цикле для каждого пикселя изображения вызываем функцию записи информации в него (`EncodeSymbol`). Ей передаём координаты текущего пикселя и индекс очередного записываемого символа текста (теперь он лежит в виде байт в `input_text_byte_list`). Выход из цикла осуществляется по факту конца набора кодируемых байт (за счёт переменной `stop`).

В этот же класс добавим методы `ReadText` и `EncodeSymbol`:

```

private void ReadText(ref BackgroundWorker worker)
{
    string text_file = Form1.input_file_name();
    if (!File.Exists(text_file))
    {
        Form1.ShowMessageBox("File is empty!");
        return;
    }
    int input_text_byte_list_ind = 0;
    FileStream fs = File.Open(text_file, FileMode.Open, FileAccess.ReadWrite);
    FileInfo finfo = new FileInfo(text_file);
    text_size = finfo.Length;
    if (text_size > Form1.img_size())
    {
        Form1.ShowMessageBox("Picture is too small for encoding!");
        return;
    }
    fs.SetLength(text_size);
    input_text_byte_list = new byte[text_size];
}

```

```

        for (int w = 0; w < fs.Length; w++)
    {
        worker.ReportProgress((int)((float)w / fs.Length * 50));
        if (worker.CancellationPending)
            return;
        input_text_byte_list[input_text_byte_list_ind] =
            Convert.ToByte(fs.ReadByte());
        input_text_byte_list_ind++;
    }
    fs.Close();
}

```

В функции `ReadText` сначала локально сохраняем в переменную `text_file` полное имя текстового файла для кодирования, который мы выбрали в диалоговом окне. Дополнительно проверяем, существует ли этот файл. Если да, начинаем считывать из него текст. Открываем этот файл в режиме чтения и записи. Получим информацию о файле (`finfo`), которая укажет размер содержимого файла в байтах. Сохраним её локально в `text_size`.

Замечание. Перед декодированием информации, занесённой в изображение (описано далее), приложение после кодирования закрывать нельзя, поскольку в данной тестовой программе длина текста запоминается (в `text_size`) во время кодирования и используется функциями декодирования для упрощения кода и сокращения времени обработки. В реальных «боевых» условиях, конечно, будет необходимо добавлять обработку изображения, не зависящую от данных о размере текстового файла.

Далее проверяем, хватит ли размеров выбранного изображения для кодирования текста заданной длины. Если да, считываем из файла текст в массив байт `input_text_byte_list`.

```

private void EncodeSymbol(ref Bitmap src, ref Bitmap res,
    ref int i, ref int j, ref int index, byte byte_)
{
    Color pixelColor = src.GetPixel(i, j);
    BitArray colorArray = Byte2Bit(pixelColor.R);
    BitArray messageArray = Byte2Bit(byte_);
    colorArray[0] = messageArray[0];
    colorArray[1] = messageArray[1];
    byte newR = Bit2Byte(colorArray);

    colorArray = Byte2Bit(pixelColor.G);
    colorArray[0] = messageArray[2];
    colorArray[1] = messageArray[3];
    colorArray[2] = messageArray[4];
    byte newG = Bit2Byte(colorArray);

    colorArray = Byte2Bit(pixelColor.B);
    colorArray[0] = messageArray[5];
    colorArray[1] = messageArray[6];
    colorArray[2] = messageArray[7];
    byte newB = Bit2Byte(colorArray);

    Color newColor = Color.FromArgb(newR, newG, newB);
    res.SetPixel(i, j, newColor);
    index++;
}

```

Функция `EncodeSymbol` записывает переданный ей байт (`byte_`, в данном случае соответствует одному символу текста) частями во все три канала текущего пикселя изображения. Байт представим в виде массива бит (`messageArray`) для возможности побитовой работы с ним. Для каждого канала в его массив бит (`colorArray`) на позиции НЗБ последовательно записываем информацию из массива `messageArray`. Так в два НЗБ красного канала записываем первые два бита сообщения, в три НЗБ зелёного – с третьего по пятый биты сообщения и в три НЗБ синего – с шестого по восьмой биты сообщения.

Также в этот класс добавим поля:

```
public static long text_size = 0;
byte[] input_text_byte_list;
```

Именно в эти переменные представленные выше функции записывают длину кодируемого текста и массив байт текста.

В сам файл добавим строку:

```
using System.IO;
```

В класс `Form1` добавим функцию, позволяющую выводить на экран сообщение с заданным текстом:

```
public static void ShowMsgBox(string input)
{
    MessageBox.Show(input);
}
```

В этот же класс добавим функции, позволяющие получить доступ к информации о размерах изображения, расположенного в рабочей копии `image`, а также об имени текстового файла, содержимое которого мы будем кодировать:

```
public static int img_size()
{
    if (image == null)
        return 0;
    return image.Width * image.Height;
}

public static string input_file_name()
{
    return src_text_file;
}
```

Все эти три функции созданы статическими из-за специфики класса `Form1`: его экземпляр создаётся единственный на весь запуск программы.

Добавим функциональность потоку `backgroundWorker2`, в котором будет выполняться «чистое» кодирование. В свойствах потока в дизайнере во вкладке

Events двойным щелчком ЛКМ создаём обработчики `DoWork`, `ProgressChanged` и `RunWorkerCompleted` и заполняем их:

```
// Encode raw
private void backgroundWorker2_DoWork(object sender, DoWorkEventArgs e)
{
    ProcessImg prc_im = (ProcessImg)e.Argument;
    prc_im.processImage(image, ref backgroundWorker2);
    if (backgroundWorker2.CancellationPending != true)
    {
        image_encoded_raw = prc_im.get_image_raw();
        image = image_encoded_raw;
    }
}

private void backgroundWorker2_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}

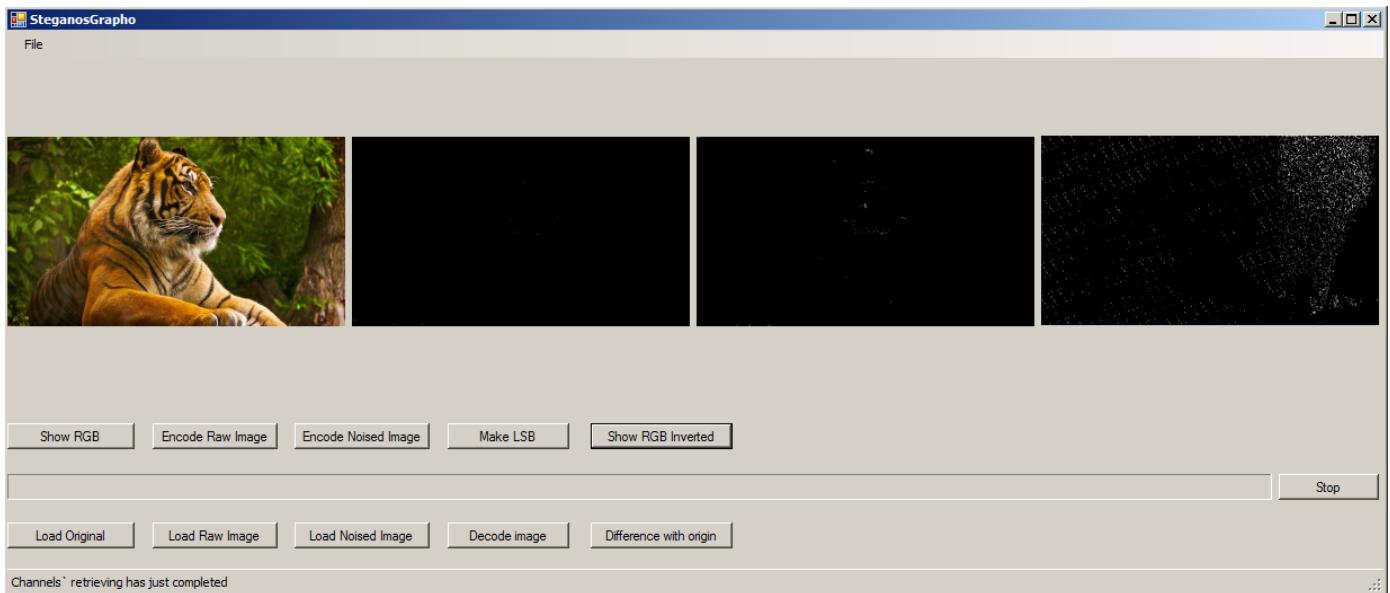
private void backgroundWorker2_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        pictureBox1.Image = image;
        pictureBox1.Refresh();
        toolStripStatusLabel1.Text = "Raw image encoding has just completed";
    }
    else
    {
        toolStripStatusLabel1.Text = "Raw image encoding was cancelled";
    }
    progressBar1.Value = 0;
}
```

В класс `Form1` добавим поле:

```
Bitmap image_encoded_raw;
```

Здесь в `DoWork` по окончании работы результат присваивается переменной класса интерфейса `image_encoded_raw`. В `RunWorkerCompleted` также обновляется основной `pictureBox`.

Запустим приложение и проверим новый функционал. Откроем `tiger.bmp` и после нажатия «Encode Raw Image» выберем текстовый файл `_TEXT_.txt`. По окончании процесса кодирования нажмём «Show RGB Inverted». По каналу синего (последний `pictureBox`) видно, что примерно в изображение внесены изменения, занявшие примерно 2/3 его объёма:



В разделе Эксперименты эти различия рассмотрены подробнее.

Возвращение закодированного «чистого» изображения

Добавим также возможность подгрузки последнего закодированного таким способом изображения в рабочую копию `image`. Двойным кликом ЛКМ по клавише «Load Raw Image» в дизайнере создадим её обработчик и в него поместим:

```
private void button8_Click(object sender, EventArgs e)
{
    if (image_encoded_raw == null)
    {
        MessageBox.Show("Raw image is empty!");
        return;
    }
    image = image_encoded_raw;
    pictureBox1.Image = image;
    pictureBox1.Refresh();
}
```

В случае, если `image_encoded_raw` не пустое, его значение копируем в `image`. Проверим работу функционала. Как было сделано выше, закодируем `_TEXT_.txt` в `tiger.bmp` и нажмём «Show RGB Inverted». Затем «Load Original» и «Show RGB Inverted» (убеждаемся, что сейчас в рабоче копии `image` снова исходное изображение). И затем «Load Raw Image» и «Show RGB Inverted». Снова видим распределение интенсивностей, полученное кодированием «чистого» изображения.

Декодирование информации

Научимся декодировать информацию обратно из изображения. В дизайнере двойным нажатием ЛКМ по клавише «Decode image» получим её обработчик и заполним его:

```

// Decode
private void button7_Click(object sender, EventArgs e)
{
    SaveFileDialog dialog = new SaveFileDialog();
    dialog.Filter = "Text files (*.txt)|*.txt";
    dialog.Title = "Save a Text File";
    dialog.ShowDialog();

    if (dialog.FileName != "")
    {
        System.IO.FileStream fs =
            (System.IO.FileStream)dialog.OpenFile();
        fs.Close();
    }
    ejected_text_file = dialog.FileName;
    ProcessImg prc_im = new DecodeImg();
    backgroundWorker4.RunWorkerAsync(prc_im);
}

```

В класс Form1 добавим поле:

```
static string ejected_text_file;
```

В ProcessImg.cs создадим класс DecodeImg:

```

class DecodeImg : ProcessImg
{
    public override void processImage(Bitmap sourceImage,
        ref BackgroundWorker worker, bool no_return_value)
    {
        process_bytes = new ProcessBytes();
        process_bytes.EjectTextFromImage(sourceImage, ref worker);
    }
}

```

В класс ProcessBytes добавим метод EjectTextFromImage:

```

public void EjectTextFromImage(Bitmap src, ref BackgroundWorker worker)
{
    // Считываем из картинки текст длиной, равной размеру текста из исходного
    // файла, которую мы запомнили в переменной text_size во время записи его
    // содержимого в изображение. В данный размер включены также и длины меток
    // начала и конца сообщения, поскольку они уже присутствовали в исходном
    // тексте.
    // В двойном цикле считываем в подготовленный массив байт отведённого
    // размера (temp_message) символы, закодированные в пикселях изображения.
    // После этого полученный массив байт переводим в строку
    // ((temp_str_message)), применяя кодировку ASCII, поскольку её достаточно
    // для работы с текстом, состоящим из латинских букв и знаков препинания.
    byte[] temp_message = new byte[text_size];
    int i = 0, j = 0;
    int msg_ind = 0;
    bool stop = false;
    for (; i < src.Width; i++)
    {
        worker.ReportProgress((int)((float)i / src.Width * 33));
        if (worker.CancellationPending)
            return;
        for (j = 0; j < src.Height; j++)
        {

```

```

        if (msg_ind == text_size)
        {
            stop = true;
            break;
        }
        temp_message[msg_ind] = Bit2Byte(DecodeSymbol(ref src,
            ref i, ref j));
        msg_ind++;
    }
    if (stop) break;
}
string temp_str_message = Encoding.ASCII.GetString(temp_message);

// Ищем в тексте метку начала. Только если она будет найдена,
// декодирование будем считать успешным. Поиск будет проводиться на строках
//(string) для наглядности. Пока не дойдём до конца строки
//temp_str_message, движемся от её начала и берём из неё подстроку длины
//метки начала. Сравниваем эту подстроку с меткой начала. Если строки
//совпадают, отмечаем, что метка начала найдена успешно.
msg_ind = 0;
bool start_reading = false;
while (msg_ind <= temp_str_message.Length)
{
    worker.ReportProgress((int)((float)msg_ind / temp_str_message.Length
        * 33 + 33));
    if (worker.CancellationPending)
        return;
    if (label_start == temp_str_message.Substring(msg_ind,
        label_start.Length))
    {
        msg_ind += label_start.Length;
        start_reading = true;
        break;
    }
    else msg_ind++;
}

// Если метка начала найдена успешно, ищем метку конца, считывая
// информацию из изображения. Поиск метки конца сообщения проводится по
// тому же принципу. Если в итоге метка не была найдена, выводим на экран
// сообщение об этом, но весь считанный текст всё равно, как и при её
// успешном поиске, записываем в файл.
// Прогресс выполнения в данной функции разделён на три части: по
// количеству отдельных операций. Третья из долей расположена в методе
// /WriteText.
bool stop_found = false;
int begin_ind = msg_ind;
if (start_reading)
{
    while (msg_ind <= temp_str_message.Length)
    {
        worker.ReportProgress(Clamp((int)((float)msg_ind
            / temp_str_message.Length * 33 + 66), 0, 100));
        if (worker.CancellationPending)
            return;
        if (label_end == temp_str_message.Substring(msg_ind,
            label_end.Length))
        {
            stop_found = true;
            strMessage = temp_str_message.Substring(begin_ind, msg_ind
                - begin_ind);
            break;
        }
        else msg_ind++;
    }
}

```

```

        }
        if (!stop_found)
        {
            strMessage = temp_str_message.Substring(begin_ind,
                temp_str_message.Length - begin_ind);
            Form1.ShowMsgBox("End label NOT found, text EJECTED!");
        }
    }
    else
    {
        Form1.ShowMsgBox("Text not found!");
        return;
    }
    WriteText();
}

```

При извлечении информации (`EjectTextFromImage`) байты можно сравнивать напрямую, но для компактности и понятности кода сравнение организовано на строках (что возможно, если работаем с текстовой информацией), хотя такой код будет выполнять дольше.

Замечание. Кодировать информацию (причём не только текстовую, но и любую другую, поскольку файлы различных типов могут быть прочитаны в бинарном формате) в изображения можно любым способом: в любые цветовые каналы и по любому закону [1]. Цель выбора правил кодирования только одна – минимизировать вероятность обнаружения факта скрытия информации. В данной лабораторной мы работаем именно с текстовой информацией.

В этот же класс добавим методы `WriteText` и `DecodeSymbol`:

```

private void WriteText()
{
    byte[] temp_message = Encoding.ASCII.GetBytes(strMessage);
    FileStream SourceStream = File.Open(Form1.ejected_file_name(),
        FileMode.Create);
    SourceStream.Seek(0, SeekOrigin.Begin);
    SourceStream.Write(temp_message, 0, temp_message.Length);
    SourceStream.Close();
}

```

Запись текста из строки (`WriteText`), как и её получение из массива байт (`EjectTextFromImage`), проводим в кодировке ASCII. В массив байт `temp_message` конвертируем данные уже полученной нами строки `strMessage`. В `WriteText` создадим и откроем для записи текстовый файл с именем, заданным нами в диалоговом окне. Убедимся, что находимся в начальной позиции файла, поместив его указатель на начало, и запишем в него последовательность байт `temp_message`.

```

private BitArray DecodeSymbol(ref Bitmap src, ref int i, ref int j)
{
    Color pixelColor = src.GetPixel(i, j);

    BitArray colorArray = Byte2Bit(pixelColor.R);
    BitArray messageArray = Byte2Bit(pixelColor.R);

```

```

        messageArray[0] = colorArray[0];
        messageArray[1] = colorArray[1];

        colorArray = Byte2Bit(pixelColor.G);
        messageArray[2] = colorArray[0];
        messageArray[3] = colorArray[1];
        messageArray[4] = colorArray[2];

        colorArray = Byte2Bit(pixelColor.B);
        messageArray[5] = colorArray[0];
        messageArray[6] = colorArray[1];
        messageArray[7] = colorArray[2];
        return messageArray;
    }
}

```

Обратите внимание, что обращение к младшим битам массива `colorArray` происходит не по последним индексам, которыми принято обозначать младшие биты в байте, а по первым, начиная от нуля. Это обусловлено спецификой заполнения массива бит функцией `Byte2Bit`, где он начинает заполняться с младших бит.

Попиксельное декодирование (`DecodeSymbol`) информации осуществляем симметрично функции `EncodeSymbol`. Каждый канал текущего пикселя, представленный в виде байта, копируем в массив бит (`colorArray`) и извлекаем из них информацию (в массив бит символа сообщения `messageArray`) теми же частями, какими она была закодирована. Первые два бита массива `messageArray` получаем из двух НЗБ красного канала, с третьего по пятый биты – из трёх НЗБ зелёного, с шестого по восьмой биты – из трёх НЗБ синего.

Также в класс `ProcessBytes` добавим поля:

```

string label_start = "St@rT";
string label_end = "4InIsh";
string strMessage;

```

`strMessage` – это та самая строка, в которую мы декодируем текстовую информацию из изображения.

В класс `Form1` добавим функцию доступа к полю `ejected_text_file`:

```

public static string ejected_file_name()
{
    return ejected_text_file;
}

```

Добавим функциональность потоку `backgroundWorker4`, в котором будет выполняться декодирование. В свойствах потока в дизайнере во вкладке `Events` двойным нажатием ЛКМ создаём обработчики `DoWork`, `ProgressChanged` и `RunWorkerCompleted` и заполняем их:

```

// Decode
private void backgroundWorker4_DoWork(object sender, DoWorkEventArgs e)

```

```

{
    ProcessImg prc_im = (ProcessImg)e.Argument;
    prc_im.processImage(image, ref backgroundWorker4, true);
}

private void backgroundWorker4_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}

private void backgroundWorker4_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        toolStripStatusLabel1.Text = "Image decoding has just completed";
    }
    else
    {
        toolStripStatusLabel1.Text = "Image decoding was cancelled";
    }
    progressBar1.Value = 0;
}

```

Проверим декодирование. Запустим приложение и, открыв tiger.bmp, проведём «чистое» кодирование текста _TEXT_.txt. Затем, нажав «Decode image», в открывшемся диалоговом окне введём название текстового файла, в который будет записан декодированный из изображения текст. По завершении декодирования в файле с заданным названием должен появиться тот же текст, что был и в исходном, кроме меток начала и конца кодируемой информации, расположенных в исходном файле перед и после всего объёма текста.

Зашумлённое кодирование информации

Теперь самостоятельно (обязательное задание) реализуйте любой способ «зашумлённого» кодирования из главы 5.3.2.1 книги [1]. Подготовим для этого инструментарий. В дизайнере двойным нажатием ЛКМ по клавише «Encode Noised Image» получаем её обработчик и заполняем его:

```

// Encode noised
private void button4_Click(object sender, EventArgs e)
{
    OpenFileDialog dialog = new OpenFileDialog();
    dialog.Filter = "Text files (*.txt)|*.txt|All files(*.*)|*.*";
    dialog.Title = "Open a Text File";

    if (dialog.ShowDialog() == DialogResult.OK)
    {
        src_text_file = dialog.FileName;
    }
    ProcessImg prc_im = new EncodeNoisedImg();
    backgroundWorker3.RunWorkerAsync(prc_im);
}

```

В ProcessImg.cs создадим класс EncodeNoisedImg:

```

class EncodeNoisedImg : ProcessImg
{
    Bitmap image_noised;

    public override Bitmap get_image_noised() { return image_noised; }

    public override Bitmap processImage(Bitmap sourceImage,
        ref BackgroundWorker worker)
    {
        process_bytes = new ProcessBytes();
        image_noised = process_bytes.EncodeNoisedImage(sourceImage, ref worker);
        return image_noised;
    }
}

```

В класс ProcessBytes добавим метод EncodeNoisedImage:

```

public Bitmap EncodeNoisedImage(Bitmap src, ref BackgroundWorker worker)
{
    return src;
}

```

Добавим функциональность потоку backgroundWorker3, в котором будет выполняться зашумлённое кодирование. В свойствах потока в дизайнере во вкладке Events двойным щелчком ЛКМ создаём обработчики DoWork, ProgressChanged и RunWorkerCompleted и заполняем их:

```

// Encode noised
private void backgroundWorker3_DoWork(object sender, DoWorkEventArgs e)
{
    ProcessImg prc_im = (ProcessImg)e.Argument;
    prc_im.processImage(image, ref backgroundWorker3);
    if (backgroundWorker3.CancellationPending != true)
    {
        image_encoded_noised = prc_im.get_image_noised();
        image = image_encoded_noised;
    }
}

private void backgroundWorker3_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
}

private void backgroundWorker3_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        pictureBox1.Image = image;
        pictureBox1.Refresh();
        toolStripStatusLabel1.Text =
            "Noised image encoding has just completed";
    }
    else
    {
        toolStripStatusLabel1.Text = "Noised image encoding was cancelled";
    }
    progressBar1.Value = 0;
}

```

```
}
```

В класс Form1 добавим поле:

```
Bitmap image_encoded_noised;
```

Здесь в DоРаВ по окончании работы результат присваивается переменной класса интерфейса image_encoded_noised. В RunWorkerCompleted также обновляется основной pictureBox.

Возвращение закодированного зашумлённого изображения

Сразу добавим функциональность загрузки в рабочую копию image последнего закодированного таким способом изображения. Двойным кликом ЛКМ по клавише «Load Noised Image» в дизайнере создадим её обработчик и в него поместим:

```
private void button9_Click(object sender, EventArgs e)
{
    if (image_encoded_noised == null)
    {
        MessageBox.Show("Noised image is empty!");
        return;
    }
    image = image_encoded_noised;
    pictureBox1.Image = image;
    pictureBox1.Refresh();
}
```

Зашумление при записи информации в изображения всё равно требуется, даже на изображениях естественной природы или изображениях, полученных сканированием невысокого качества, то есть на таких, на которых уже присутствует шум в младших битах, так как распределение НЗБ в пикселях с закодированной информацией, вероятнее всего, будет отличаться от распределения этого шума, и факт кодирования будет легко обнаружить.

Задание. Аналогично функции EncodeRawImage, реализуйте функцию EncodeNoisedImage, не только кодирующую информацию в изображение, но и зашумляющую всё изображение.

4. Эксперименты

Эксперименты можно проводить на файлах из каталога ...\\Практика 3\\тексты. Файл _TEXT_.txt следует использовать для наблюдения границы зашифрованного текста в изображении, а также для прослеживания этой границы при переходе от чистого кодирования к зашумлённому.

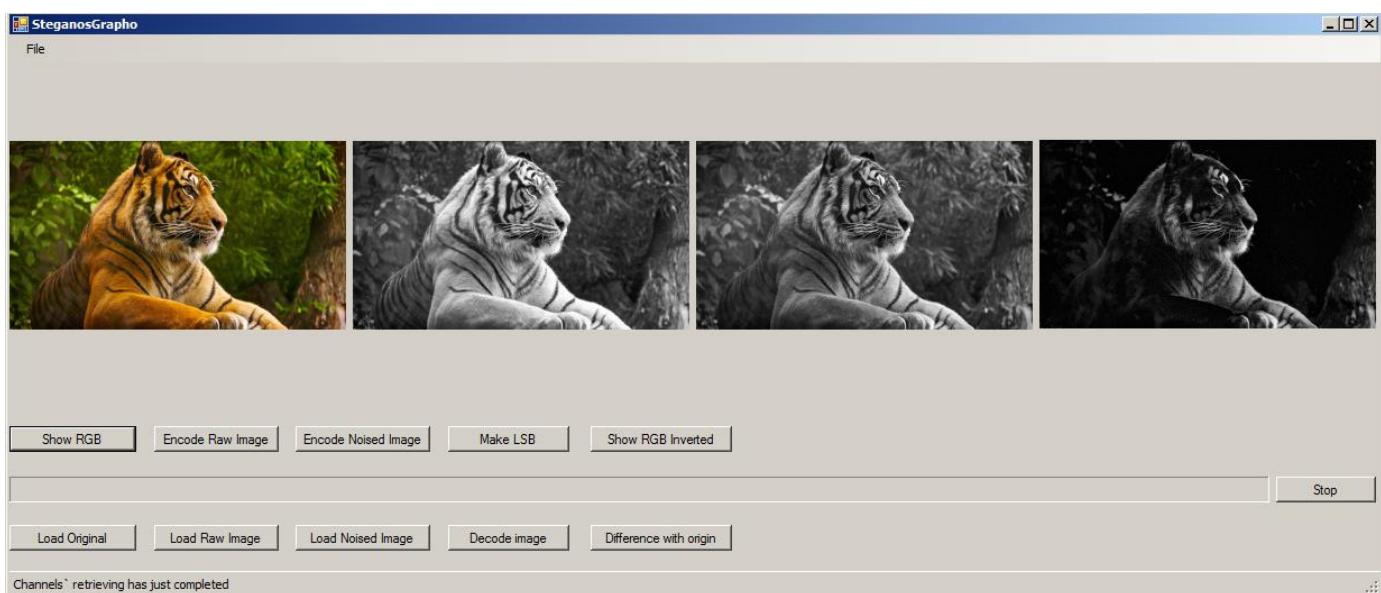
В файле _TEXT_.txt, помимо содержательного текста, записаны метки его начала и конца: St@rT и 4IniSh соответственно. При извлечении данного текста, ранее уже закодированного в изображение, его наличие мы поймём именно по этим меткам.

Просмотр каналов изображения

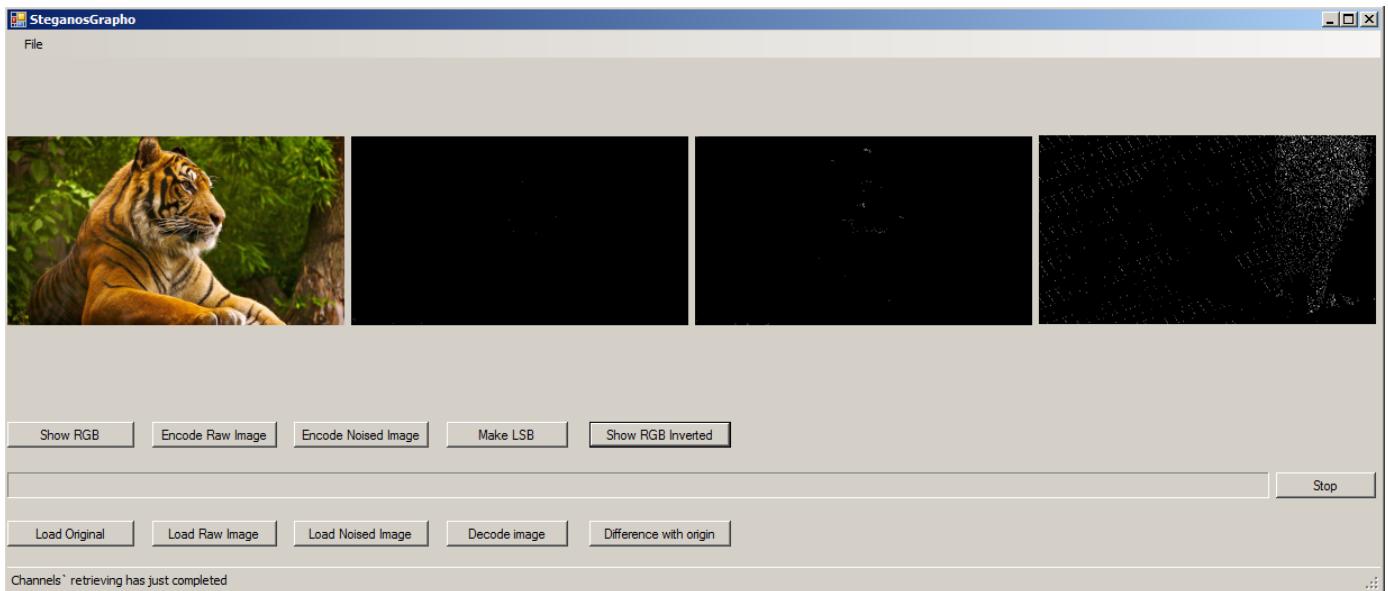
Средствами приложения откройте изображение tiger.bmp. Проверьте функциональность извлечения и отображения каналов изображения в традиционном RGB, а также в его инверсии («Show RGB Inverted»). Аналогичное проделайте с изображениями water_PNG3290.png, aboutus-rightbottomv3.png и what-is-hdr-opener-small.jpg.

Цветовые каналы после «чистого» кодирования

Откройте изображение tiger.bmp. Нажмите клавишу кодирования чистого изображения «Encode Raw Image». Выберите файл _TEXT_.txt. Дождитесь завершения операции. О нём просигнализируют бегунок progressBar`а и соответствующее сообщение в статусной строке. После нажатия «Show RGB» в цветовых каналах появятся изображения, визуально не отличные от каналов неизменённого изображения, поскольку, как уже говорилось, человек практически не воспринимает изменения на уровне трёх НЗБ:



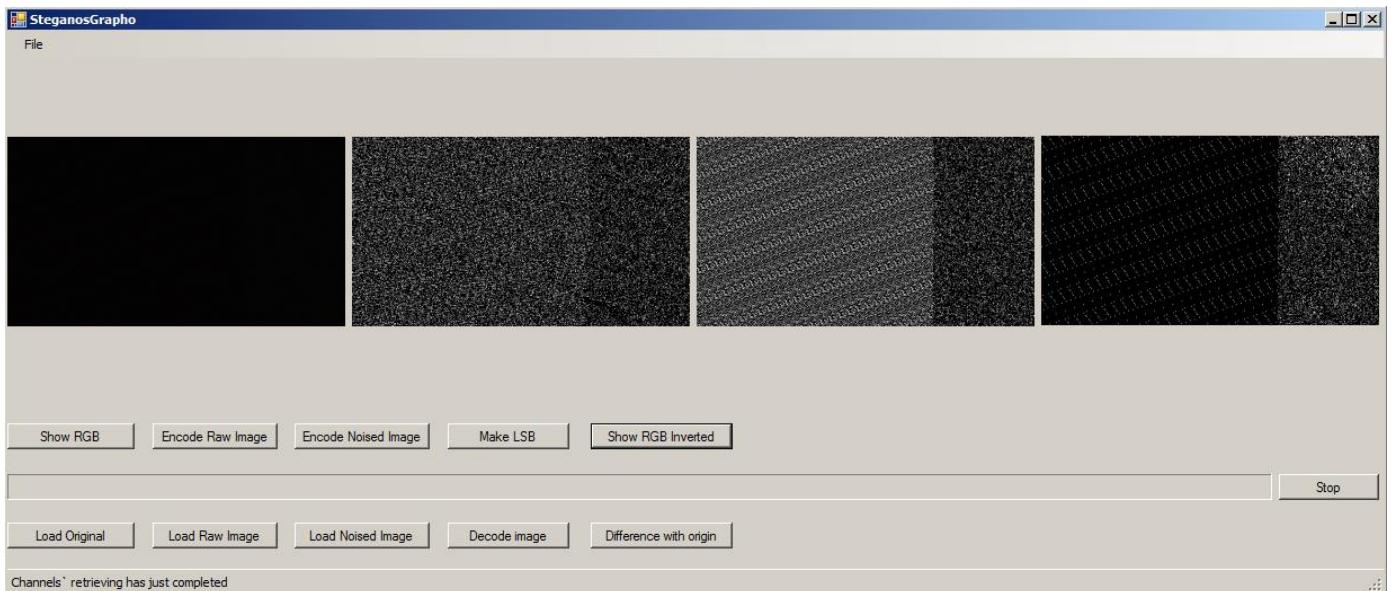
Напротив, функция «Show RGB Inverted» устроена так, что все различия между изображениями отображаются строго чёрным цветом (интенсивность по всем каналам равна 0), а отсутствие каких-либо различий между соответствующими пикселями – белым. Поэтому все изменения при таком способе отображения становятся заметными. Не забывайте дожидаться завершения выполнения каждой предыдущей операции:



Как видно, изменения в цветовых каналах изображения стали визуально заметными.

Отображение НЗБ

Не закрывая приложение после «чистого» кодирования выполните выделение наименьших значащих битов (клавиша «Make LSB»). PictureBox основного изображения автоматически примет новый вид и его каналы будут состоять из байт, представленных только указанным вами количеством младших бит. Для того чтобы просмотреть изменения в каждом канале отдельно, снова нажмите последовательно клавиши «Show RGB» и «Show RGB Inverted». Классическое отображение RGB не обнаружит никаких изменений. На позициях каналов будут видны практически чёрные изображения, так как значения НЗБ мало отличны от минимальных интенсивностей, то есть нулевых. Инвертированный режим RGB покажет как распределение символов записанного текста, так и границу между изменённой частью изображения и нетронутой:



С очевидными проявлениями факта скрытия информации необходимо бороться! Границу между использованной и нетронутой частями изображения следует ликвидировать зашумлением изображения (способы рассмотрены в главе 5.3.2.1 книги [1]).

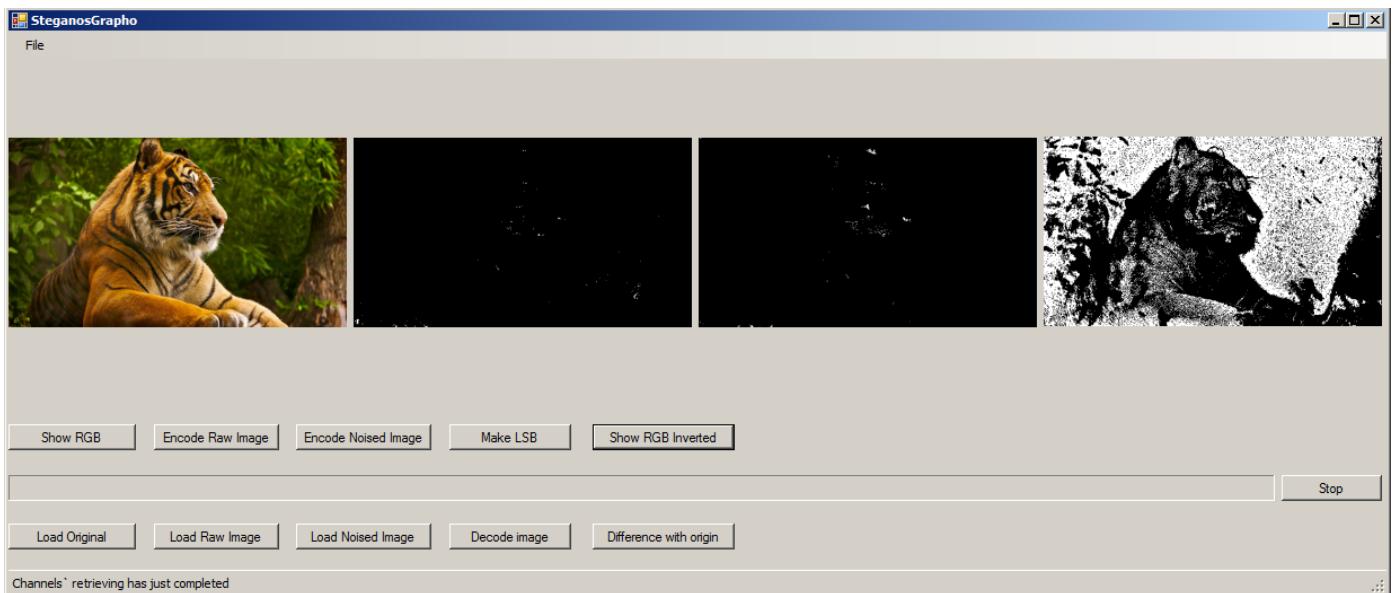
В первоначальной реализации, представленной в главе Реализация, меняются только три младшие бита.

Задание. (Обязательное) Для отображения классического RGB мы убедились, что когда изменяются только младшие три бита, это не даёт увидеть разницу в их значениях, поскольку действует закон о том, что биты, не старше третьего, практически не привносят визуальных изменений в изображение. Если не вносить никаких изменений в изображение, но выделять различное количество НЗБ и отображать их классическим способом («Show RGB»), до третьего бита включительно НЗБ обычно несут в себе только шум. Убедитесь в этом или опровергните на различных изображениях: tiger.bmp, aboutus-rightbottomv3.png и других. В функции `MakeLSB` класса `ProcessBytes` изменяйте количество НЗБ по одному и тем же способом наблюдайте изменения в каналах. Начиная с какого бита в каналах, становится виден не только шум, но и очертания содержимого изображения?

Разница между оригинальным и изменённым изображениями

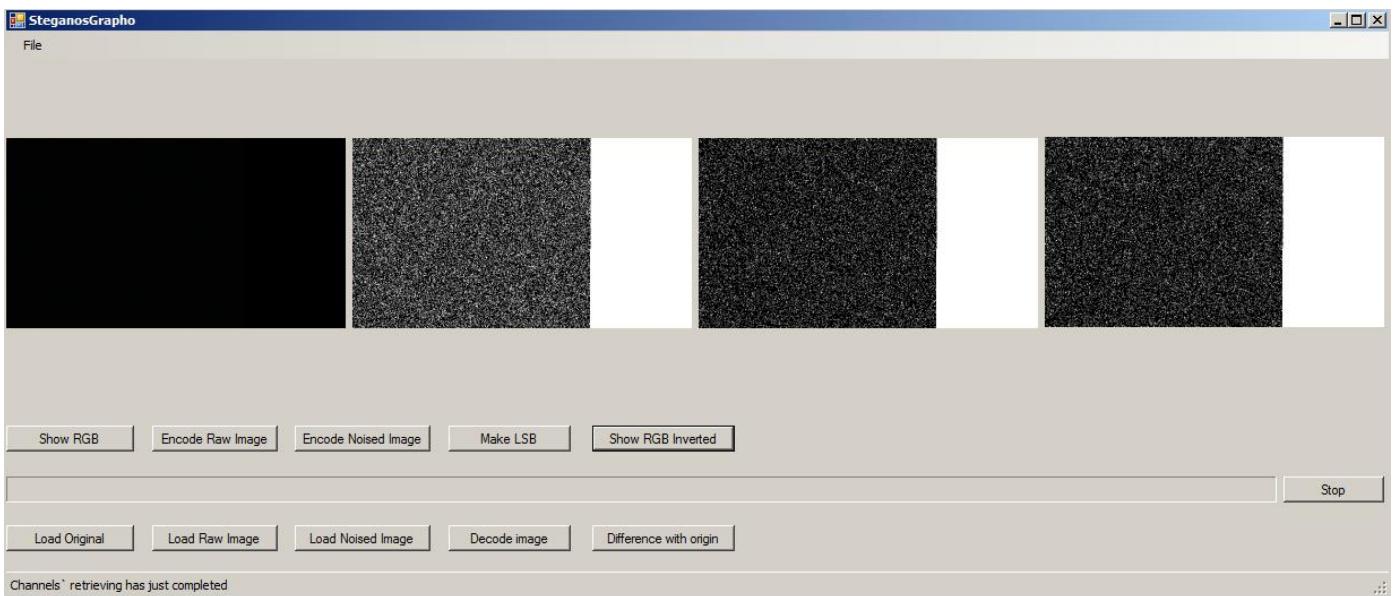
Можно просмотреть разницу между оригинальным изображением и изображением, только что изменённым посредством обнуления всех бит, кроме младших бит заданного количества и позиций. Таким образом, мы увидим разницу в младших битах этих двух изображений. Для этого нажмите «Difference with origin».

После открытия tiger.bmp последовательность «Make LSB», «Difference with origin», «Show RGB Inverted» даст результат:



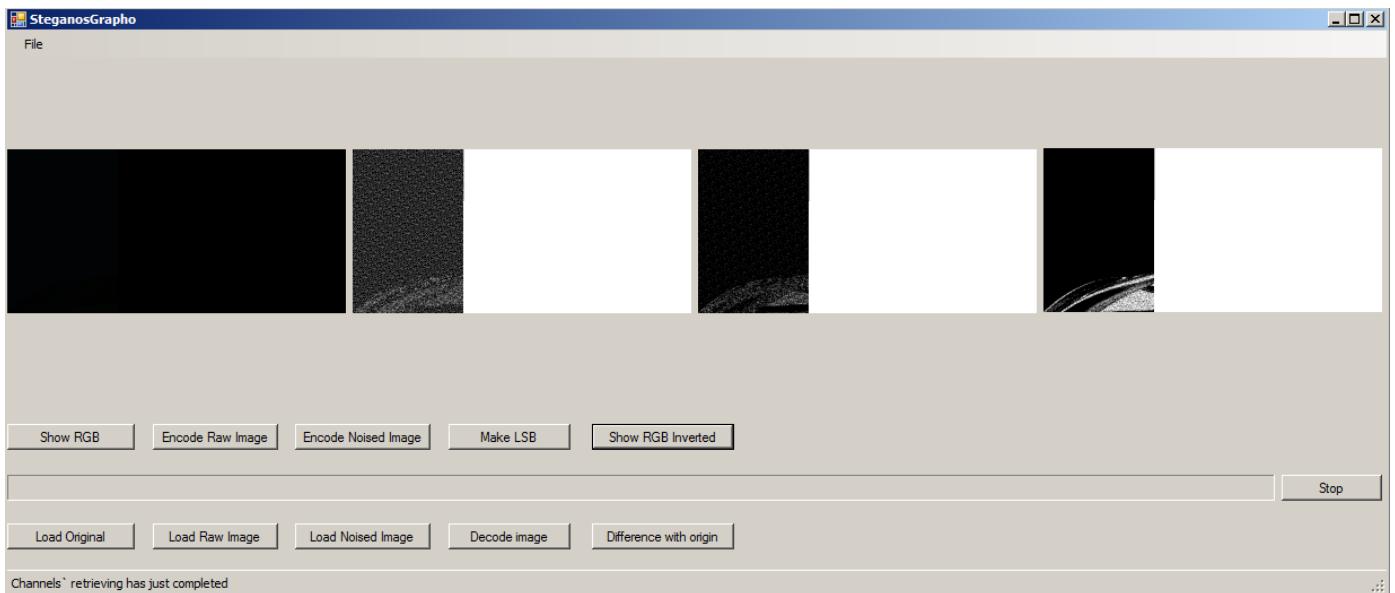
Только что мы фактически увидели всю полезную, «чистую» информацию данной картинки, так как шум, который заключён здесь в трёх НЗБ, теперь отсутствует. «Очищеное» изображение лежит в рабочей копии image.

Последовательность (также для tiger.bmp) «Encode Raw Image», «Difference with origin», «Show RGB Inverted» даст:

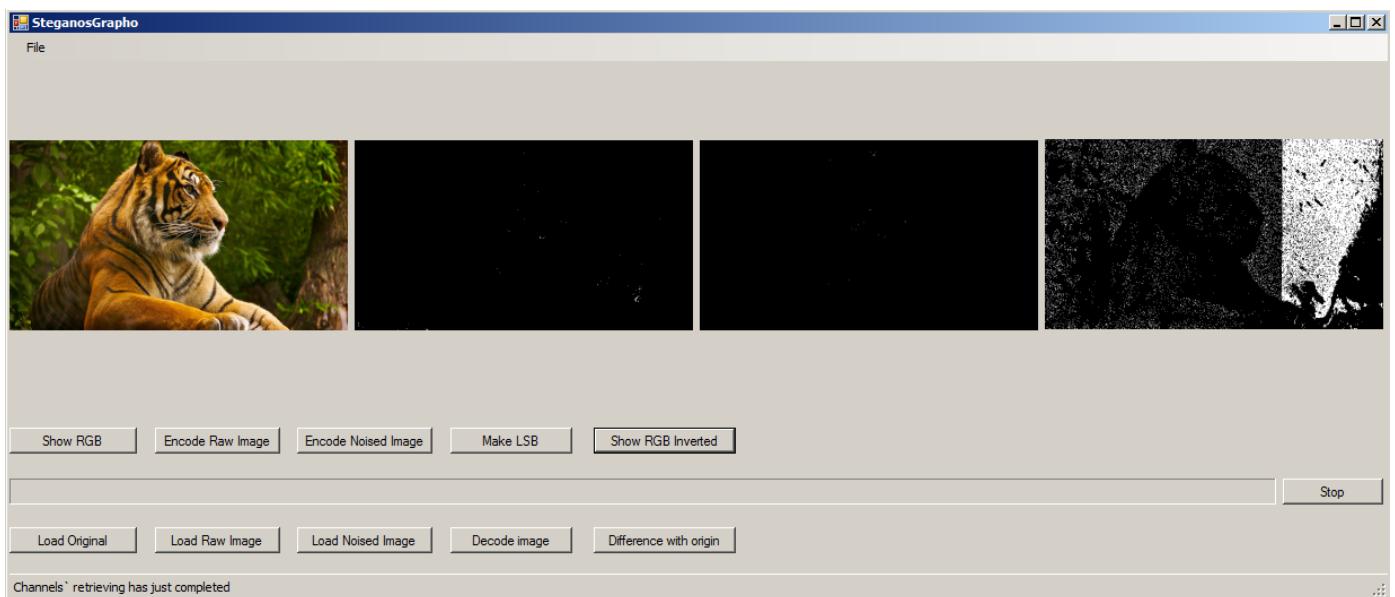


Видно, что кодирование самих значений прошло на уровне шума. Резко видна лишь граница между изменённым и оригинальным изображениями. Поэтому для кодирования информации берут изображения, которых нет в широком доступе, чтобы нельзя было сравнить таким образом изображение, подозреваемое на наличие информации в нём, и оригинальное.

Аналогичная последовательность действий для aboutus-rightbottomv3.png:

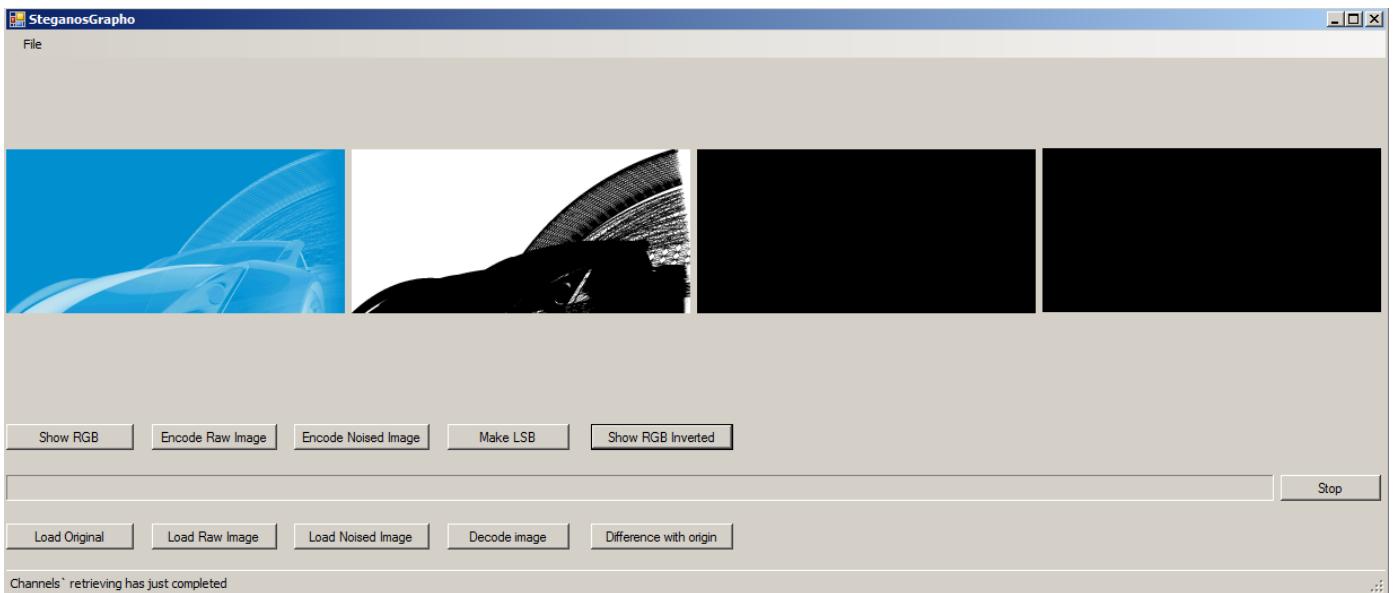


Не закрывая приложение, можно просмотреть разницу между оригинальным изображением и закодированным «чистым» способом с последующим выделением НЗБ: «Load Original», «Encode Raw Image», «Make LSB», «Difference with origin», «Show RGB Inverted»:



Оставшаяся справа часть изображения показывает разницу между НЗБ и всеми битами, то есть «чистые» составляющие байт без шума. Часть слева – сразу отображает (чёрным) также «чистые» составляющие байт, и разницу в НЗБ между оригинальным и полученным изображениями (т.е., можно сказать, инверсию оригинальных значений НЗБ).

Аналогичная последовательность действий для aboutus-rightbottomv3.png:



Наибольшие видимые изменения для tiger.bmp в синем канале, а для aboutus-rightbottomv3.png – в красном обусловлены тем, что на первом почти не задействован в составлении цветов синий канал, а на втором – красный, поэтому все проводимые в этих каналах изменения вносят большее различие с их оригинальной версией, чем в других каналах.

Декодирование изображения

Декодируйте изображение нажатием «Decode image». Полученный файл должен совпадать с исходным, за исключением того, что в полученном таким способом файле будут отсутствовать метки начала и конца, которые были нами записаны в него первоначально.

Не забывайте, что перед декодированием не следует закрывать приложение, поскольку данные о длине сообщения, которое было туда ранее закодировано, сохраняется только на время единого запуска.

Если в исходном тексте метки начала и конца сместить (т.е. поставить не непосредственно в начало и конец, а метку начала – сделать чуть смещённой от начала файла, и метку конца – смещённой от конца файла), то на выходе получим текст, заключённый ка раз между этими метками.

5. Для сдачи лабораторной

Необходимо:

1. Ориентироваться в главе 5.3.2.1 книги [1]
2. Реализовать кодирование информации в зашумлённое изображение, любым из способов (см. главу 5.3.2.1 книги [1]): например, зашумлением НЗБ случайными битами или дублированием участка закодированного изображения
3. Ориентироваться в реализации работы с байтами, представленной в данной лабораторной (класс ProcessBytes)

4. Средствами приложения, построенного на основе данной лабораторной, уметь демонстрировать кодирование изображения, отображение наименьших значащих битов каналов изображения, границу между закодированной частью изображения и его оставшейся свободной частью, разницу между оригинальным изображением и изменённым
5. Уметь (любым способом) оставлять в каналах изображения то или иное количество бит (более подробно задание описано в разделе «Эксперименты», «отображение НЗБ»)
6. Средствами приложения, построенного на основе данной лабораторной, уметь просматривать разницу между исходным и изменённым изображениями

6. Ссылки на литературу

[1] «Коханович Г.Ф., Пузыренко А.Ю. Компьютерная стеганография. Теория и практика (2006)»

Список литературы

1. Никулин Е.А., Компьютерная графика. Модели и алгоритмы: Учебное пособие. – СПб.: Издательство "Лань", 2017. – 368 с.: ил.
2. Дёмин А.Ю., Основы компьютерной графики: учебное пособие / Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2011. – 191 с.
3. Дейтел, Х. C#: Пер. с англ. / Дейтел Х., Дейтел П., Листфилд Дж., Нието Т., Йегер Ш., Златкина М. – СПб.: БХВ - Петербург, 2006. – 1056 с.: ил.
4. Вольф Д., OpenGL 4. Язык шейдеров. Книга рецептов / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2015. – 368 с.: ил.
5. Learn Computer Graphics From Scratch! [Электронный ресурс]. – Режим доступа: <http://www.scratchapixel.com/>

Вадим Евгеньевич **Турлапов**
Александра Александровна **Гетманская**
Евгений Павлович **Васильев**
Мария Владимировна **Дубровская**

**Методические указания для проведения
лабораторных работ по курсу
"КОМПЬЮТЕРНАЯ ГРАФИКА"**

Учебное пособие

Федеральное государственное автономное
образовательное учреждение высшего образования
Национальный Исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского
603950, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная. Гарнитура Таймс.
Усл. печ. л.. Уч.-изд. л. 3,3.
Заказ № 325. Тираж экз.

Отпечатано в типографии Нижегородского госуниверситета
им. Н.И. Лобачевского
603600, г. Нижний Новгород, ул. Большая Покровская, 37
Лицензия ПД № 18-0099 от 14.05.01