МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ 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

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. Методические указания для проведения лабораторных работ по курсу "КОМПЬЮТЕРНАЯ ГРАФИКА": Учебное пособие. – Нижний Новгород, Национальный Исследовательский Нижегородский государственный университет им. Н.И. Лобачевского, 2018 – X 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 рейтрейсинг 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 ИТММ в качестве пособия при подготовке и проведении лабораторных работ по курсу «Компьютерная графика».

УДК
ББК

Оглавление

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 деятельности, изучающая создание, способы хранения и обработки изображений в компьютере. В настоящий момент 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 изображения сцены по ее математическому описанию.

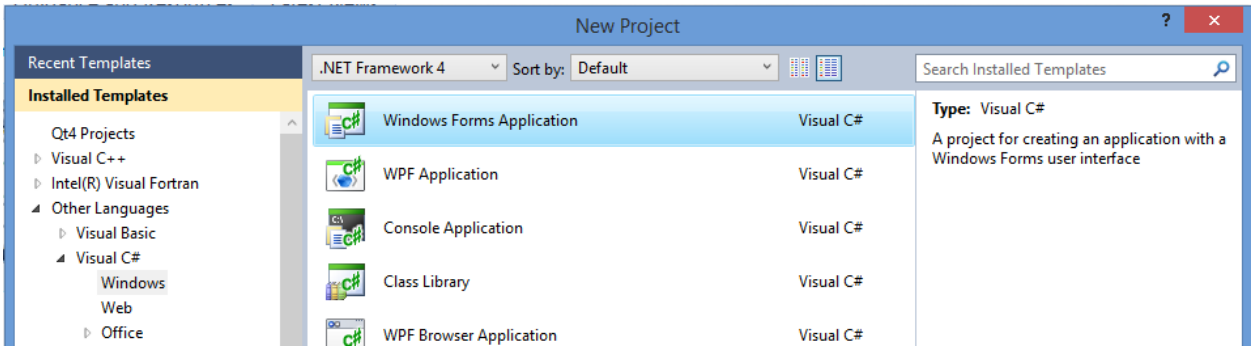
В данном пособии предложено три лабораторные работы, связанные с 2D и 3D графикой и с использованием графических ускорителей. Лабораторные работы представлены в формате пошаговых туториалов, позволяющих самостоятельно с нуля создать законченные приложения на языке C#. Для сдачи лабораторной работы необходимо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

Лабораторная работа №1. Обработка изображений

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

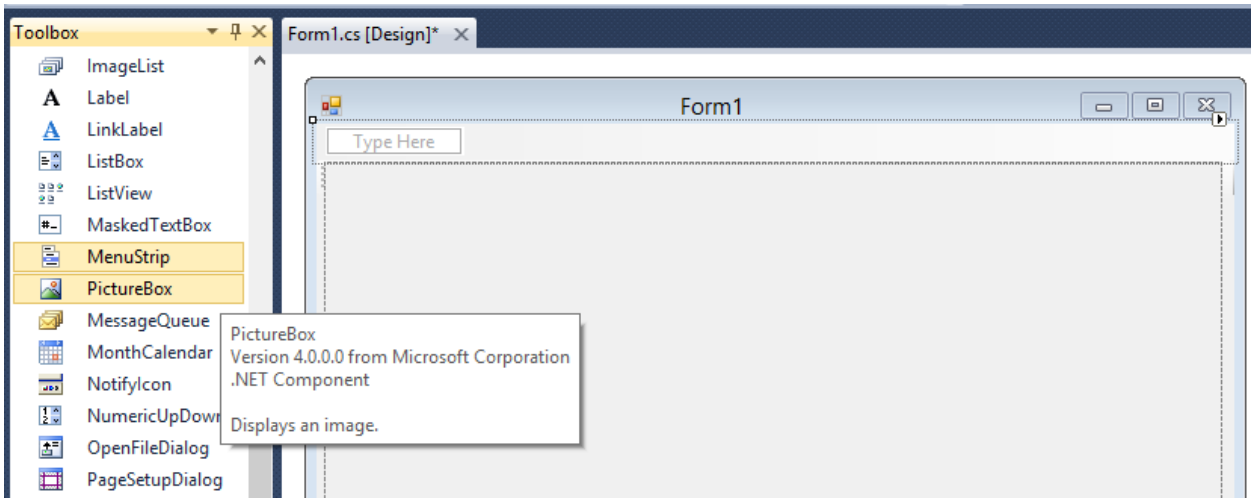
Чтобы создать новый проект, выберитеFile ->New ->Project, или нажмите Ctrl+Shift+N. В открывшемся окне выберите шаблон WindowsFormsApplication. В нижней части окна введите имя для вашего будущего проекта.



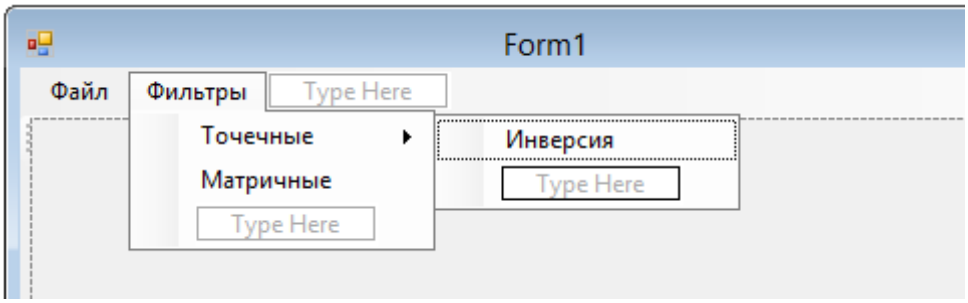
Нажмите F5. На экране должно открыться пустое окно вашей программы.

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 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 MenuStrip, они появятся на форме. MenuStrip автоматически займет место под заголовком формы, а PictureBox появится на том месте, куда вы его перетащили, растяните его до размеров самой формы. Окно пример следующий вид:



Щелкните ЛКМ 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.



2. 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 вы 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)
{
}
```

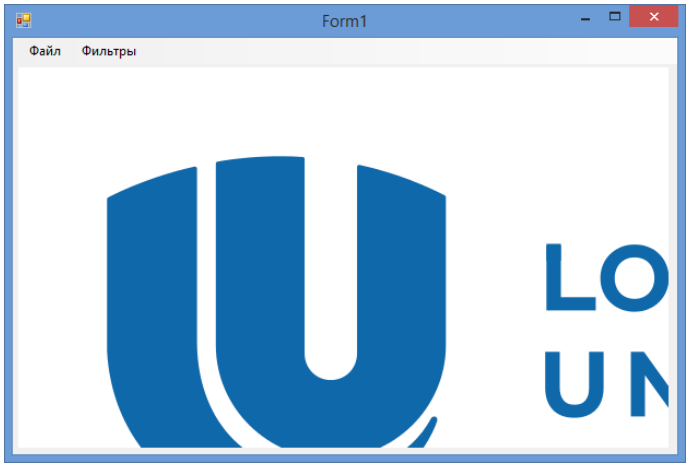
В случае выполнения данного условия инициализируйте вашу переменную image выбранным изображением. Для этого воспользуйтесь конструкцией ниже.

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

После того, как вы загрузили картинку в программу, необходимо ее визуализировать на форме, для этого image присвойте свойству pictureBox.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. Параметру SizeMode установите значение Zoom. Снова запустите программу. Поэкспериментируйте с другими значениями этого параметра.

3. 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 объявления 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 подающееся ей на вход. 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 пикселю 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 VisualStudio 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 Filter, 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;
}
```

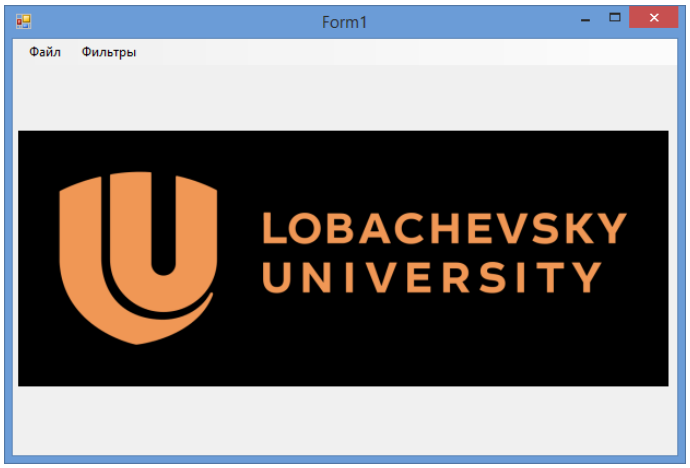
Откройте графический редактор формы. Сделайте двойной щелчок ЛКМ по элементу меню «Инверсия». Создастся новая функция, которая будет вызываться при выборе элемента «Инверсия».

```
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().

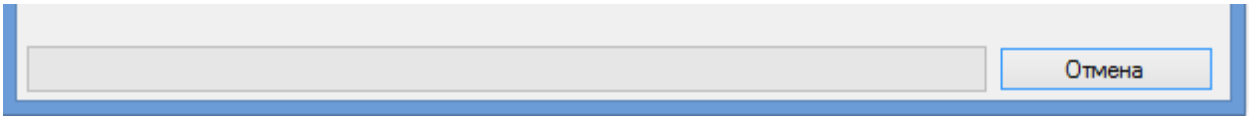
```
InvertFilter filter = new InvertFilter();
Bitmap resultImage = filter.processImage(image);
pictureBox1.Image = resultImage;
pictureBox1.Refresh();
```

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



4. Создание индикатора прогресса

Некоторые фильтры работают сравнительное 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 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.

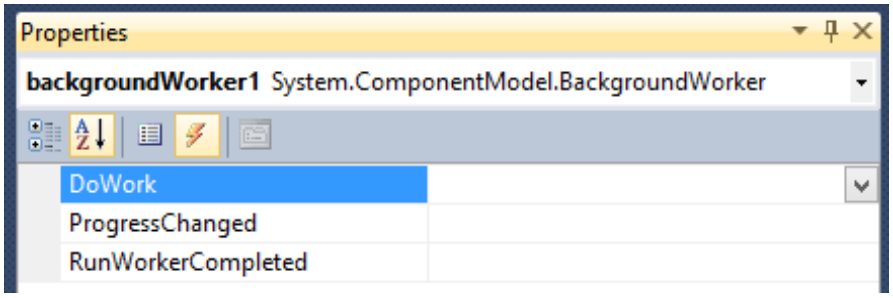


BackgroundWorker - (System.ComponentModel.BackgroundWorker) класс, предназначенный для создания и управления работой потоков. Он предоставляет следующие возможности:

- Стандартизированный протокол создания, сигнализации о ходе 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, на которой расположены доступные события для элемента. Двойным щелчком 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 сроку, 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, выполняющей по нажатию кнопки. Используйте функцию CancelAsync с класса BackgroundWorker, чтобы остановить выполнение фильтра.

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

5. Матричные фильтры

Главная часть матричного фильтра – ядро. Ядро – это матрица коэффициентов, которая покомпонентно умножается на значение пикселей изображения для получения требуемого результата (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.

```
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 – координаты текущего пикселя. Переменные l и k принимают значения от -radius до radius и означают положение точки в матрице ядра, если начало отсчета поместить в центр матрицы. В переменных idX и idY хранятся координаты пикселей-соседей пикселя x,y, для которого происходит вычисления цвета.

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)
    );
```

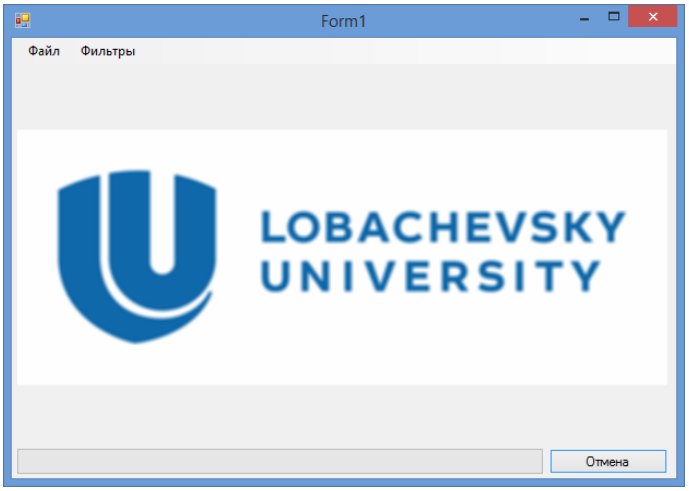
Создайте класс BlurFilter – наследник класса MatrixFilter. Переопределите конструктор по умолчанию, в котором создайте матрицу 3*3 со значением 1/9 в каждой ячейке.

```
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. Также 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, коэфициенты для которого рассчитываются по формуле Гаусса:

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

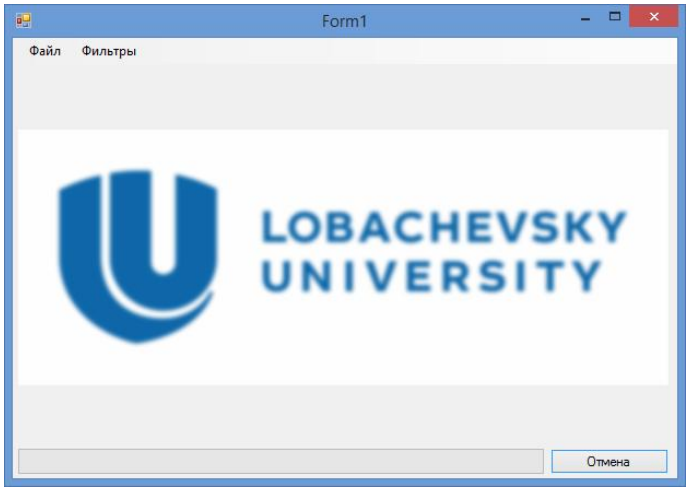
Создайте новый класс GaussianFilter – наследник класса MatrixFilter. Создайте функцию, которая будет рассчитывать ядро преобразования по формуле Гаусса

```
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) / (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.



6. Задания для самостоятельного выполнения

Применив полученные знания, добавьте в программу следующие фильтры:

- Создайте точечный фильтр, переводящий изображение из цветного в черно-белое. Для этого создайте фильтр GrayScaleFilter – наследник класса Filters, и создайте функцию calculateNewPixelColor, которая переводит цветное изображение в черно-белое по следующей формуле:

$$Intensity = 0.36 * R + 0.53 * G + 0.11 * B$$

Полученное значение записывается во все три канала выходного пикселя.

- Создайте точечный фильтр «Сепия», переводящий цветное изображение в изображение песочно-коричневых оттенков. Для этого найдите интенсивность, как у черно-белого 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:

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

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

7. Список фильтров

Тиснение

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

Перенос/Поворот

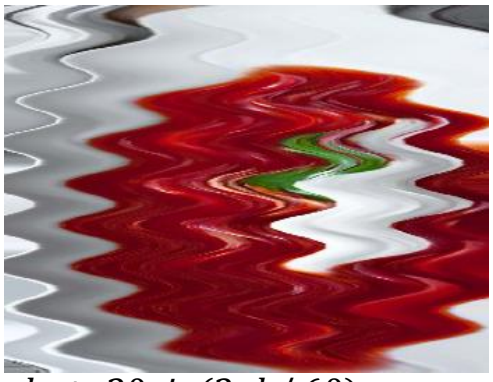
Перенос: $\begin{bmatrix} x(k,l) = k + 50; \\ y(k,l) = l; \end{bmatrix}$

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


Поворот:

$$\begin{cases} x(k,l) = (k - x0)cos(\mu) - (l - y0)sin(\mu) + x0; \\ y(k,l) = (k - x0)sin(\mu) + (l - y0)cos(\mu) + y0; \end{cases}$$, где $(x0, y0)$ – центр поворота, μ – угол поворота

«Волны»

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

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

Эффект «стекла»


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

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

8. 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 [1].
- Реализовать возможность изменения размера окна, при которой элементы будут перемещаться или растягиваться пропорционально изменению размера окна.
- Реализовать отмену последнего или нескольких последних действий.

9. Задания для сдачи лабораторной работы

- 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 фильтра.
- 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.

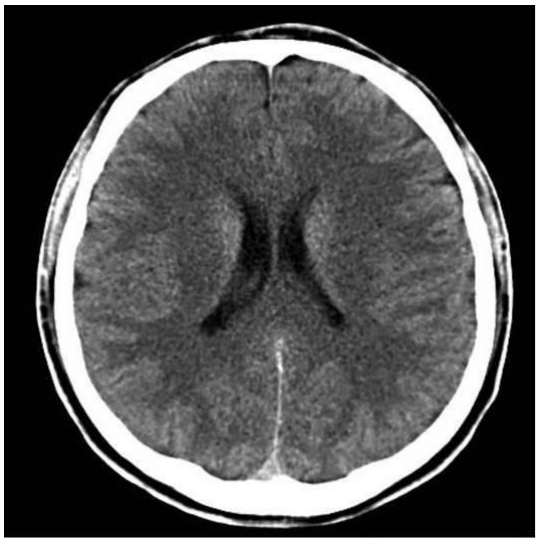
10. Ссылки:

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"

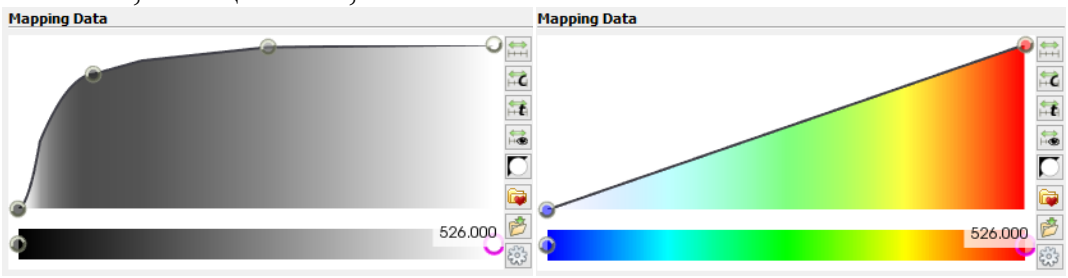
Томография - это получение послойного изображения внутренней структуры объекта.

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 воксел 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 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 требуются 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*, *BezieCurveQuadric*);
- 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 выглядит <*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 Toobox (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 нажните "Ok". На панели ToolBox в разделе General появится новый элемент GLContol, 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.Orto() 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 будет 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

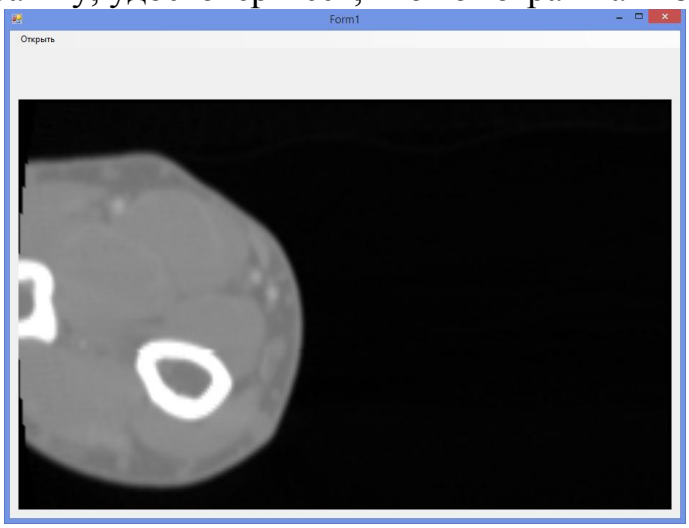
Создайте кнопку, либо элемент меню, по нажатию на который будет вызываться функция, которая будет открывать файл с томограммой и настраивать OpenGL окно. Код функции приведен ниже. В классе Form1 необходимо объявить переменную bool loaded = false, чтобы не запускать отрисовку, пока не загружены данные.

```
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 Maximux 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;
}
```

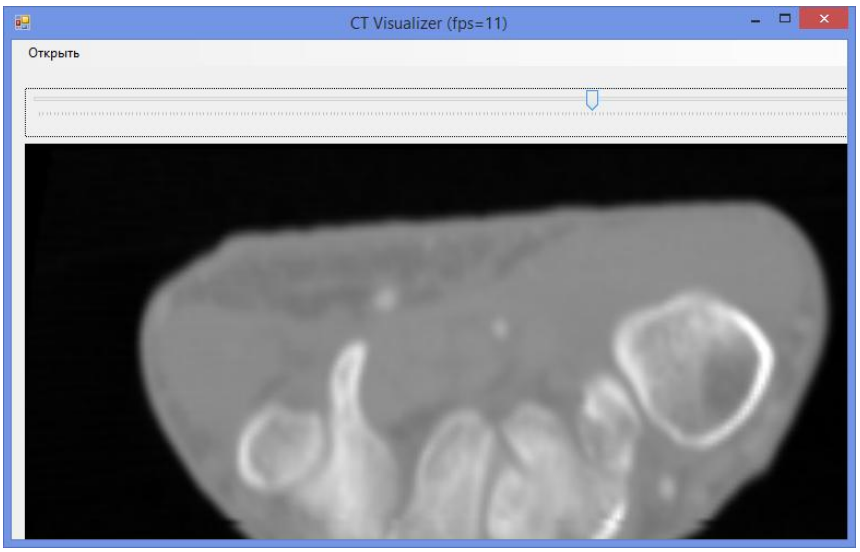
В классе Form1 создайте переменные int FrameCount, DateTime NextFPSUpdate и функцию displayFPS.

```
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, OpenTK.Graphics.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.

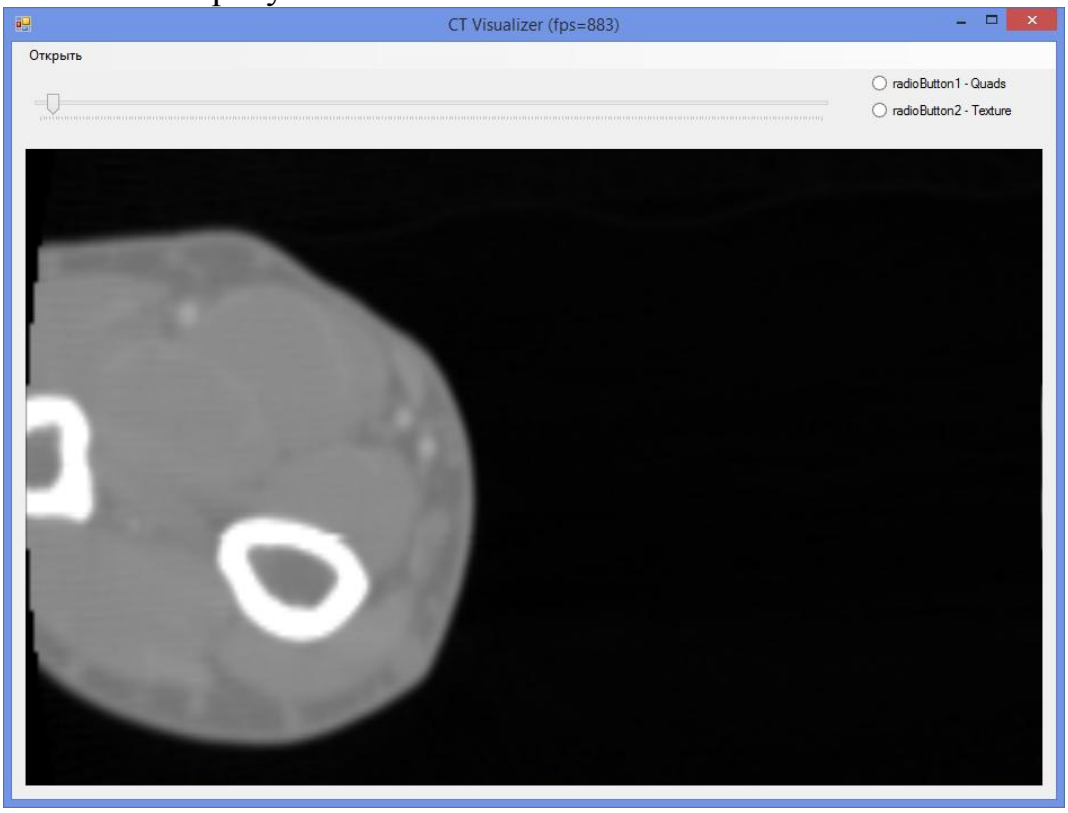
В классе Form1 измените функцию glControl1_Paint так, чтобы она рисовала томограмму с помощью текстуры, а загружала текстуру только когда переменная needReload будет установлена в true.

```
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).

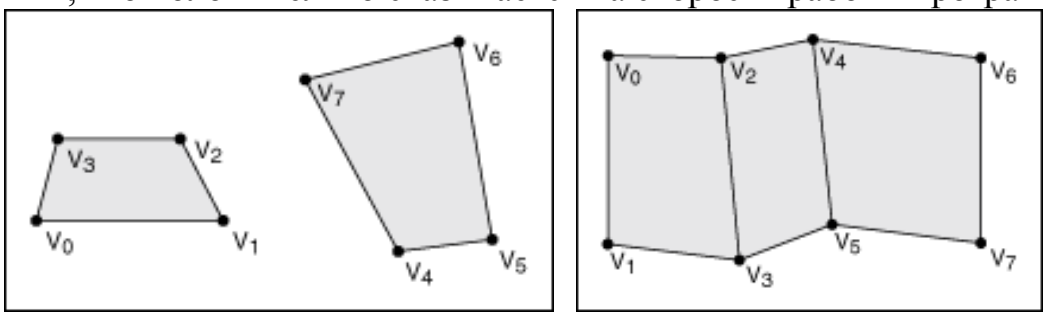
15. Задания для самостоятельной работы

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

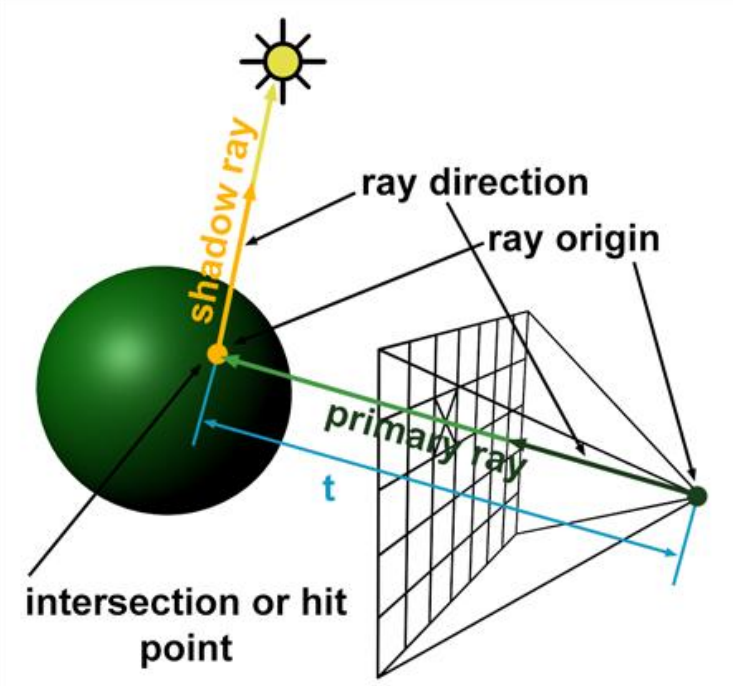
1. https://habr.com/post/310790/- большой цикл статей по OpenGL.
2. https://github.com/opentk/opentk - основной репозитори й библиотеки OpenTK.
3. https://www.intuit.ru/studies/courses/2313/613/lecture/13296 - ИНТУИТ. Создание графических моделей с помощью Open Graphics Library.
4. https://habr.com/post/173131/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

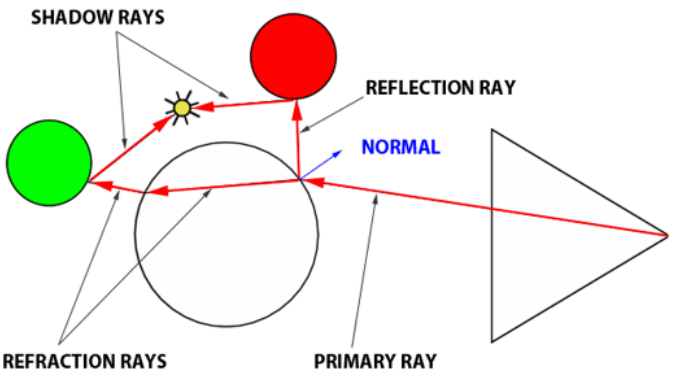
Лабораторная работа №3. Рейтрейсинг

1. Что такое рейтрейсинг

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 цвет пикселя.





В алгоритме присутствует несколько важных вычислительных этапов:

a. Вычисление цвета для диффузной поверхности;

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

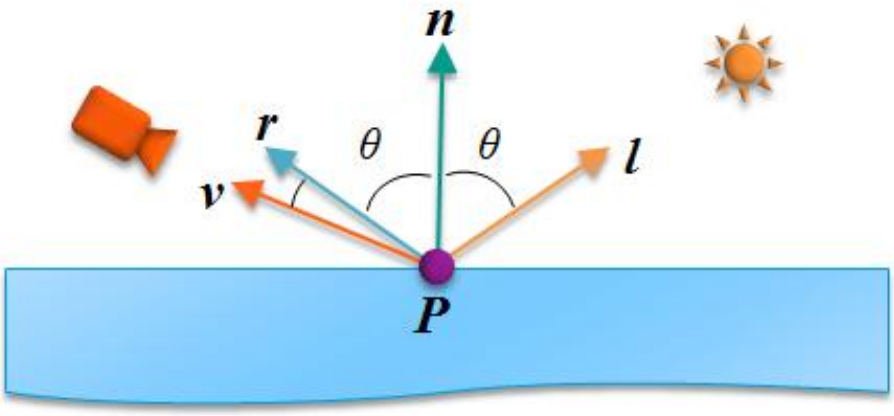
Вычисление цвета для диффузной поверхности: освещение по формуле Фонга:

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

$r = reflect(-v, n)$

C – цвет материала

L – цвет блика



Вычисление цвета точки P по формуле Фонга.

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

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

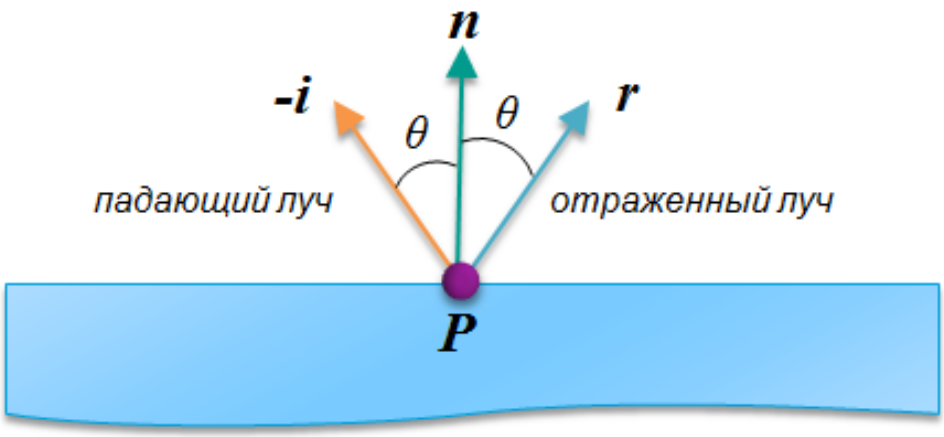


Рисунок 3. Вычисление отраженного луча.

Вычисление направления преломления. Если поверхность прозрачна, то строится еще и вторичный луч прозрачности (transparency ray). Для

определения направления луча используется закон преломления (геометрическая оптика):

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

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

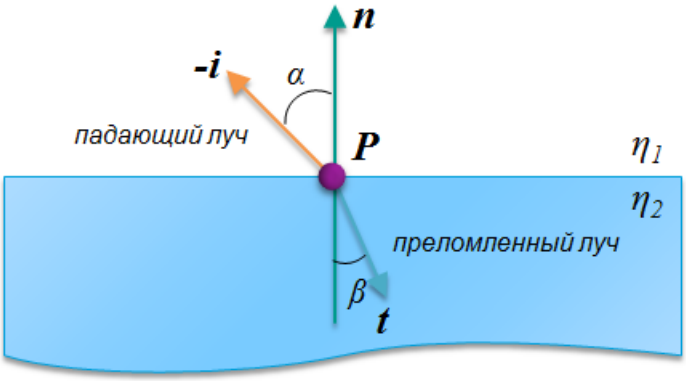


Рисунок 4. Вычисление преломленного луча.

2. О шейдерных программах

Шейдеры – это компонент шейдерной программы, мини-программа, которая выполняется в рамках отдельного этапа в общем конвейере OpenGL, выполняются непосредственно на GPU и действуют параллельно. Шейдерные программы предназначены для замены части архитектуры OpenGL, которую называют конвейером с фиксированной функциональностью. С версии OpenGL 3.1 фиксированная функциональность была удалена и шейдеры стали обязательными. Шейдер — специальная подпрограмма, выполняемая на GPU. Шейдеры для OpenGL пишутся на специализированном C-подобном языке — GLSL. Они компилируются самим OpenGL перед использованием.

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.

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

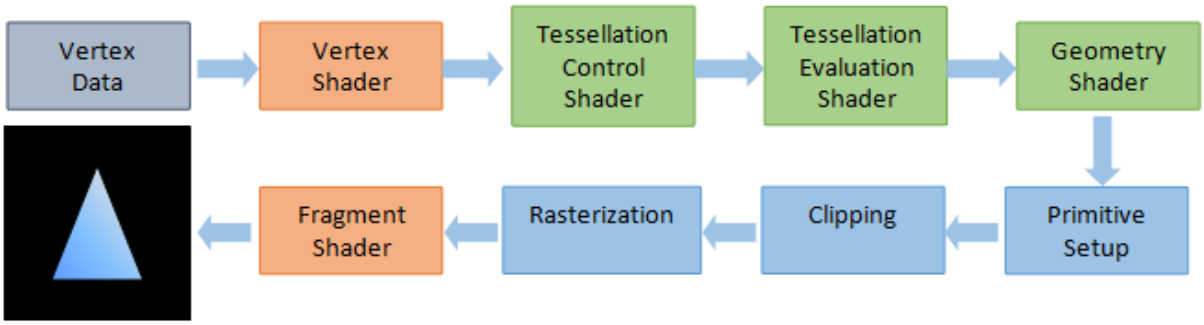


Рисунок 5. Графический конвейер OpenGL 4.3

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 и скомпилировать в шейдерную программу. Создайте два пустых текстовых файла «raytracing.vert» для вершинного шейдера и «raytracing.frag» - для фрагментного. По стандарту OpenGL шейдеры компилируются основной CPU программой после запуска. 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 производится стыковка 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 что-то нарисовать нужно нарисовать квадрат (GL_QUAD), заполняющий весь экран. 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;
}
```

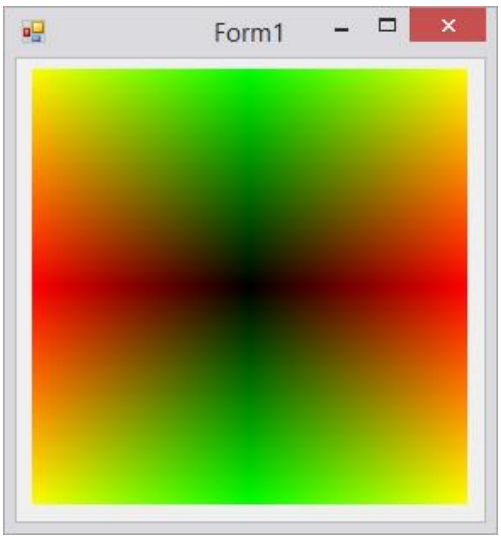
Переменные, отдаваемые вершинным шейдером дальше по конвейеру объявлены со спецификатором out.

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:

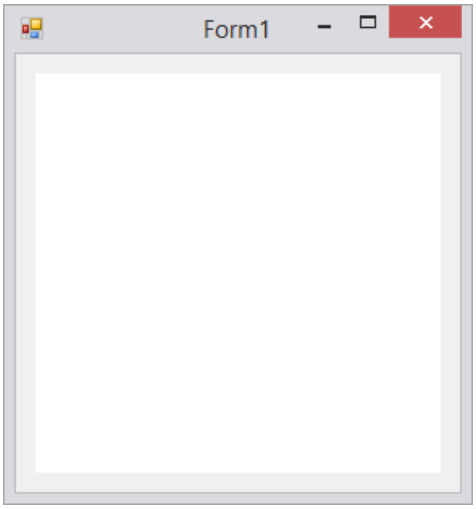


Левый нижний угол соответствует координатам (-1,-1), правый верхний – (1,1).

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.



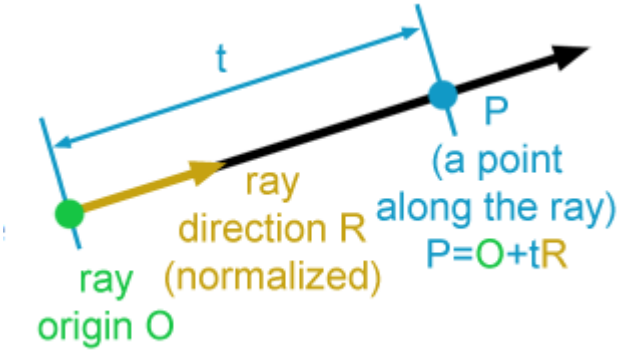
4. Генерация первичного луча

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

Для моделирования наблюдателя у нас есть камера. В обратном рейтрейсинге через каждый пиксель выходного 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 $r = o + dt, t \epsilon [0, \infty)$.



Добавляем функцию генерации луча:

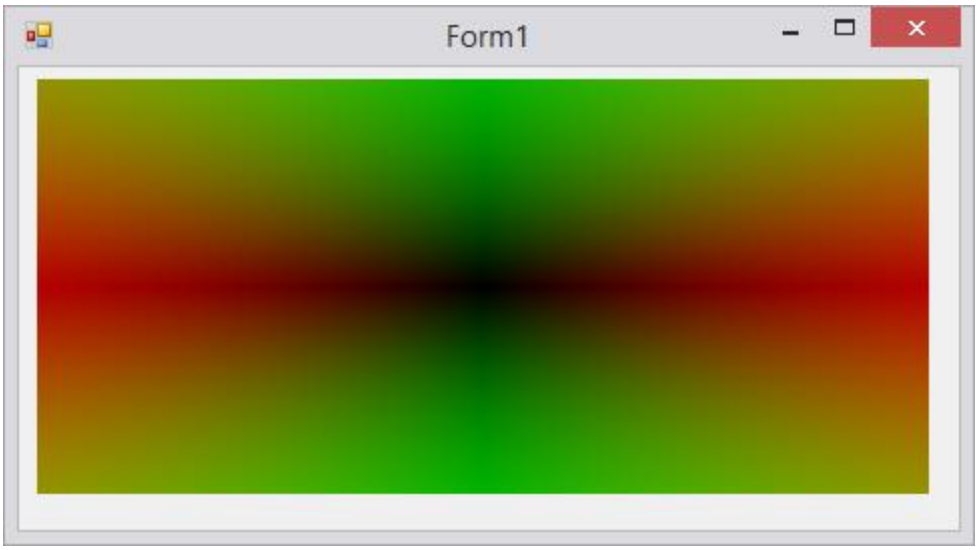
```
SRay GenerateRay ( SCamera uCamera )
{
      vec2 coords = glPosition.xy * uCamera.Scale;
      vec3 direction = uCamera.View + uCamera.Side * coords.x + uCamera.Up * coords.y;
      return SRay ( uCamera.Position, normalize(direction) );
}

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()

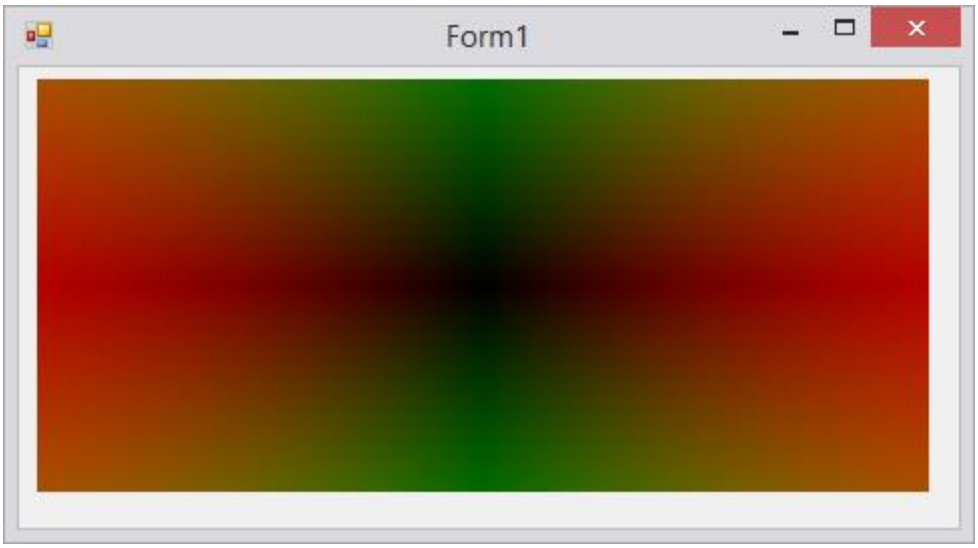
```
void main ( void )
{
```

```
    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 леча со сферой, можно воспользоваться аналичтическим решением:

Уравнение луча: $r = o + dt$

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

Подставляем $(o + dt)^2 - R^2 = 0$

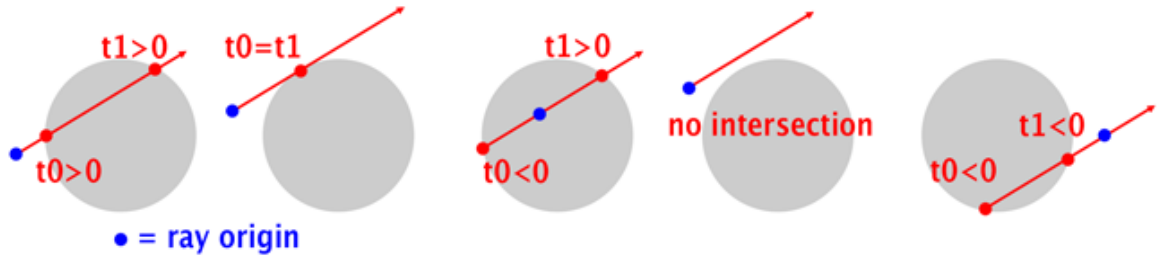
Раскрываем скобки и получаем квадратное уравнение относительно t:

$$d^2t^2 + 2odt + o^2 - R^2 = 0$$

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.sscc.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, Блинна, Кук-Торренс. Мы будем реализовывать затенение 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 отражение прямого излучения.

$$I = k_a I_a + k_d(\vec{n}, \vec{l}) + k_s \cdot \max((\vec{v}, \vec{r}), 0)^p$$, 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 освещения, $p-$ коэффициент резкости бликов

$$r = reflect(-v, n)$$

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

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 геометрии 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 какую-нибудь ещё геометрию 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 зациклится 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;
};
```

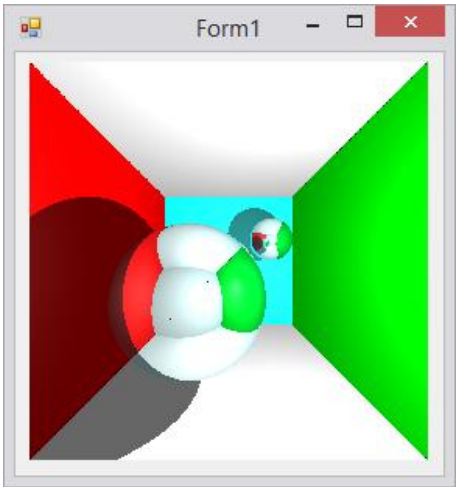
Создайте стек на основе массива, который умеет класть луч на стек (pushRay), брать луч со стека (popRay) и проверять есть ли ещё элементы в стеке (isEmpty).

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();
```

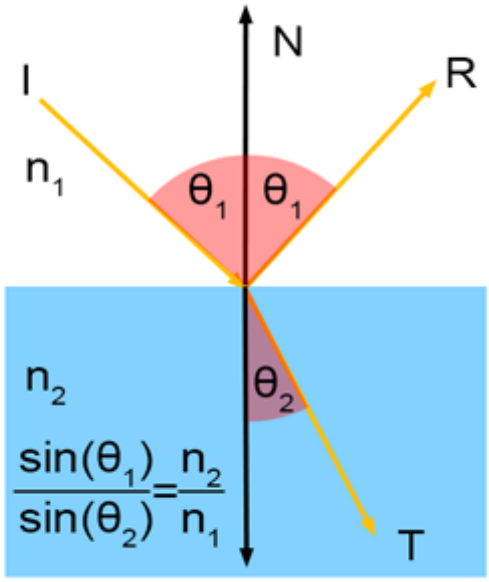
```
    ray = trRay.ray;
    SIntersection intersect;
    intersect.Time = BIG;
    start = 0;
    final = BIG;
    if (Raytrace(ray, start, final, intersect))
    {
        switch(intersect.MaterialType)
        {
        case DIFFUSE_REFLECTION:
        {
            float shadowing = Shadow(uLight, 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 свободна подвергнута расширенью 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;

Список литературы

1. Никулин Е.А., Компьютерная графика. Модели и алгоритмы: Учебное пособие. – СПб.: Издтельство "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**

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