Лабораторная работа №1. Фильтры

Оглавление

Создание проекта

Для лабораторных работ рекомендуется использовать Visual Studio 2010 или более поздние версии. Также необходимо использование библиотек Qt, для этого надо подключить Qt Visual Studio Tools.

Чтобы создать новый проект, выберете File → New → Project, или нажмите Ctrl + Shift + N. Выбираем вкладку Visual C++ →Qt5 Project → Qt Console Application. Затем выбираем путь, где будет находится наш проект, даём проекту имя и нажимаем OK.

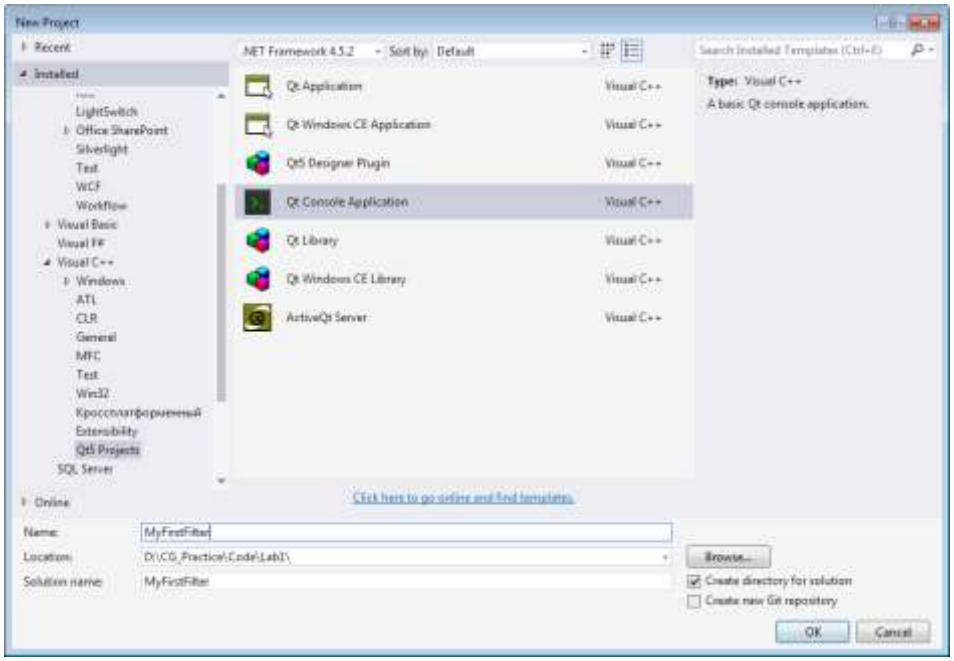


Рисунок 1. Настройка нового проекта.

После этого откроется окно Qt5 Console Project Wizard, нас устраивают стартовые настройки, поэтому ничего не меняя нажимаем кнопку Finish.

Загрузка изображения в программу

Изначально важно отметить, что загрузка изображения будет происходить с помощью аргументов функции main. Таким образом зададим их хотя бы в рамках дебага. Делаем клик правой кнопкой мыши по названию нашего проекта в окне Solution Explorer и выбираем Properties. Заходим во вкладку Configuration Properties, далее в Debugging. Пишем в строке Command Arguments "-p" и через пробел указываем адрес места на жёстком диске, где лежит изображение.

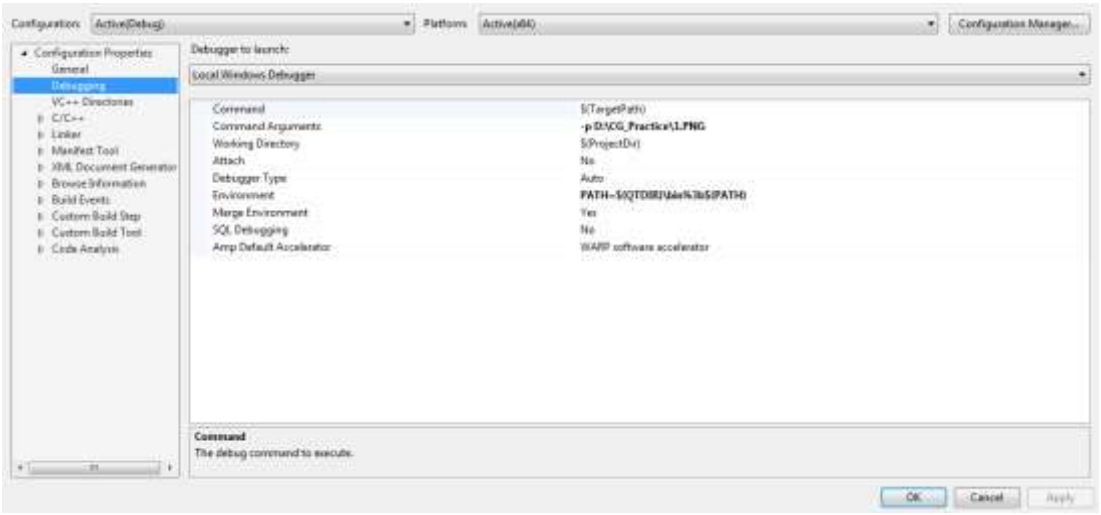


Рисунок 2. Окно настроек проекта

В данном случае "-p" можно заменить, это лишь необходимый флаг, после которого у нас однозначно определён адрес расположения изображения.

Теперь определяем переменные s строкового типа и переменную photo типа QImage. Для того, чтобы определить вторую, необходимо подключить библиотеку QImage.

```
#include <QImage>
```

Мы выбрали именно тип QImage потому, что в этом типе данных обращение к каждому отдельному пикселю и его RGB-значениям выполняется наиболее быстрым образом.

```
void main(int argc, char *argv[]) {
    string s;
    QImage photo;
```

Отметим, что параметр int argc хранит количество передаваемых значений в функцию main, а char *argv[] хранит сами значения.

Дальше организуем цикл, внутри которого будем запоминать адрес, где лежит изображение. Мы знаем, какое максимальное значение элементов передано в функцию, и если среди них есть адрес изображения, то он среди них. Поэтому цикл будет выглядеть так:

```
    for (int i = 0; i < argc; i++) {
        if ( !strcmp(argv[i], "-p") && (i + 1 < argc) ) {
            s = argv[i + 1];
        }
    }
```

Мы знаем, что если после "-p" есть какое-то значение, то это значение - адрес хранения нашего изображения, поэтому сохраняем в переменную s значения элемента стоящего следом за "-p".

Теперь, когда мы знаем адрес, мы можем загрузить изображение в программу. У переменных типа QImage есть метод bool load(const QString &fileName, const char *format = nullptr). Вызовем эту функцию для переменной photo следующим образом:

```
    photo.load( QString( s.c_str() ) );
    photo.save("Starting.PNG");
```

Конструкция QString(*char) позволяет нам перевести строку s в тип QString, который необходим функции load.

Следом мы сохраняем наше изображение с расширением .png и названием Starting в папку с проектом.

Таким образом мы загрузили изображение в нашу программу, и теперь оно готово, чтобы применять к нему фильтры.

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 Solution Explorer (Ctrl+Alt+L), нажмите правой кнопкой мыши по имени вашего проекта, выберите Add →Class. В открывшемся окне введите имя класса - Filters. В Solution Explorer появились файлы Filters.h и Filters.cpp. Открываем редактирование файла Filters.h и пишем пустому классу Filters, у которого есть только конструктор по умолчанию и деструктор виртуальную функцию вычисления новых цветов пикселей для изображения. Для этого перед типом вычисляемого значения пишем слово virtual, а после перечисления параметров функции пишем "= 0;"

```
#include <qimage.h>

class Filter {
public:
    Filter()  {};
    ~Filter() {};

    virtual QImage calculateNewImagePixMap( const QImage &photo, int radius ) = 0;
};
```

Для того, чтобы вычислять полную карту пикселей мы будем вычислять каждый пиксель отдельно, и для того, чтобы привести значения к допустимому значению создадим вспомогательную функцию Clamp. Откроем файл Filters.cpp и напишем шаблонную функцию clamp, у которой на входе будет три параметра, текущее значение на проверку, максимальное значение и минимальное значение. Идея этой функции в том, что если текущее значение больше максимального допустимого, то возвращаем максимальное, а если меньше минимального, то возвращаем минимум. Если же оба условия не верны, то возвращаем само число.

```
template <class T>
T clamp(T v, int max, int min) {
    if (v > max)
      return max;
    else  if (v < min)
      return min;

    return v;
}
```

Теперь все приготовления готовы и мы создадим наш первый фильтр. Для этого возвращаемся к редактированию файла Filters.h и создаём там класс InvertFilter, наследника класса Filter. Так как класс наследник Filter, то надо описать ему функцию calculateNewImagePixMap, важно написать то же самое, что и в родительском классе, только без слов virtual и приравнивания к нулю.

```
class Invert_filter : public Filter {
public:
    Invert_filter()  {};
    ~Invert_filter() {};

    QImage calculateNewImagePixMap( const QImage &photo, int radius );
};
```

Для того, чтобы определить вычисления методом calculateNewImagePixMap, вернёмся в файл Filters.cpp и определим функцию там. В самом начале необходимо создать переменную типа QImage, которую в конце мы будем возвращать результатом функции, в её конструктор вставляем передаваемый параметр photo.

Необходимо вычислить новый цвет для каждого пикселя, поэтому необходимо организовать цикл внутри цикла, первый, "внешний", цикл пойдёт от нуля до ширины изображения, второй же пойдёт от нуля до высоты изображения.

```
QImage
Invert_filter :: calculateNewImagePixMap( const QImage &photo, int radius ) {
    QImage result_Image(photo);

    for (int x = 0; x < photo.width(); x++)
        for (int y = 0; y < photo.height(); y++) {
```

В теле цикла определим объект типа QColor, который представляет из себя цвет конкретного пикселя и приравняем его к значениям пикселя в исходном изображении.

```
            QColor photo_color = photo.pixelColor(x, y);
```

Далее вычислим новый цвет. Наш фильтр подразумевает инвертирование цвета исходного изображения, для достижения такого эффекта необходимо вычитать из 255 текущее значение по каждому из компонентов.

```
            photo_color.setRgb( 255 - photo_color.red(),
                                255 - photo_color.green(),
                                255 - photo_color.blue() );
```

Теперь приравниваем значения пикселя result_Image в точке x, y к вычисленному цвету. Для этого воспользуемся методом setPixelColor(int x, int y, const QColor &color). После прохождения циклов возвращаем изображение result_Image.

```
            result_Image.setPixelColor( x, y, photo_color );
        }

    return result_Image;
}
```

Создайте в функции main объект класса InvertFilter. Создайте новый экземпляр класса QImage для изменённого фильтром изображения, и присвойте этому экземпляру результат функции calculateNewImagePixMap().

```
Invert_filter*        invert = new Invert_filter();

QImage invertImage = invert->calculateNewImagePixMap(photo, 0);
invertImage.save("Invert.png");
```

Запустите программу, проверьте работоспособность фильтра. Если всё выполнено верно, то после запуска в папке с проектом у вас появится инвертированное изображение.

Рисунок 3 - инвертированная фотография кота

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 Matrix_filter, содержащий в себе динамический массив чисел с плавающей запятой vector, и целочисленное значение mRadius.

```
class Matrix_filter : public Filter {
public:
    float* vector;
    int    mRadius;
};
```

Опишем конструктор по умолчанию для Matrix_filter внутри объявления класса, который будет устанавливать значение параметра mRadius.

```
Matrix_filter(int radius = 1) : mRadius(radius) {};
```

Также обозначим методы вычисления нового цвета для отдельного пикселя, QColor calculateNewPixelColor() и метод, наследуемый от родительского класса, QImage calculateNewImageixMap().

```
QImage calculateNewImagePixMap( const QImage &photo, int radius );
QColor calculateNewPixelColor( QImage photo, int x, int y, int radius );
```

Открываем файл Filters.cpp и определяем метод вычисления цвета отдельного пикселя. Обозначим три целочисленные переменные, которые будут хранить значению результирующего цвета по одному из трёх компонентов. И далее сразу определим и вычислим размер диметр матрицы фильтра, которая равна удвоенному радиусу плюс один.

```
int returnR = 0;
int returnG = 0;
int returnB = 0;
int size    = 2 * radius + 1;
```

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. По выполнению вернём значение полученного цвета пикселя.

```
    for (int i = -radius; i <= radius; i++)
        for (int j = -radius; j <= radius; j++) {
            int idx = (i + radius) * size + j + radius;

            QColor color = photo.pixelColor( clamp<int>(x + j, photo.width() - 1, 0),
                                             clamp<int>(y + i, photo.height() - 1, 0) );

            returnR += color.red()   * vector[ idx ];
            returnG += color.green() * vector[ idx ];
            returnB += color.blue()  * vector[ idx ];
        }
    return QColor( clamp(returnR, 255, 0),
                   clamp(returnG, 255, 0),
                   clamp(returnB, 255, 0) );
}
```

Тщательно разберите приведенный выше код. Переменные x и y – координаты текущего пикселя. Переменные i и j принимают значения от -radius до radius и означают положение точки в матрице ядра, если начало отсчета поместить в центр матрицы. В переменной idx хранятся координаты пикселя-соседа пикселя x,y, для которого происходит вычисления цвета.

В файле Filters.h создайте класс Blur_filter – наследник класса Matrix_filter. Переопределите конструктор по умолчанию, в котором создайте матрицу с размером зависящим от радиуса.

```
class Blur_filter : public Matrix_filter {
public:
    Blur_filter() {
        int size = 2 * mRadius + 1;
        vector   = new float[size * size];
        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                vector[i*size + j] = 1.0f / (size * size);
    }
};
```

Создаём объект класса Blur_filter в main, проводим те же действия, что и с Invert_filter, чтобы проверить работоспособность нашего кода.

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

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

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

```
void
Gaussian_blur_filter :: createGaussianVector(int radius, int sigma) {
    //Определяем размер ядра
    const unsigned int size = 2 * radius + 1;
    //коэффициент нормировки ядра
    float norm = 0;
    //создаём ядро фильтра
    vector = new float[size*size];

    //рассчитываем ядро линейного фильтра
    for (int i = -radius; i <= radius; i++)
        for (int j = -radius; j <= radius; j++) {
            int idx = (i + radius) * size + (j + radius);
            vector[ idx ] = exp(-(i * i + j * j) / (sigma * sigma));
            norm += vector[ idx ];
        }

    //нормируем ядро
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) {
            vector[i*size + j] /= norm;
        }

}
```

Создайте конструктор по умолчанию, который будет раздавать фильтр размером 3*7 и с коэффициентом сигма, равным 2.

```
class Gaussian_blur_filter : public Matrix_filter {
public:
    Gaussian_blur_filter()  {
        createGaussianVector(3, 2);
    }
    ~Gaussian_blur_filter() {};

    void createGaussianVector( int radius, int sigma );
};
```

Аналогично остальным фильтрам допишите код для запуска фильтра. Протестируйте.

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

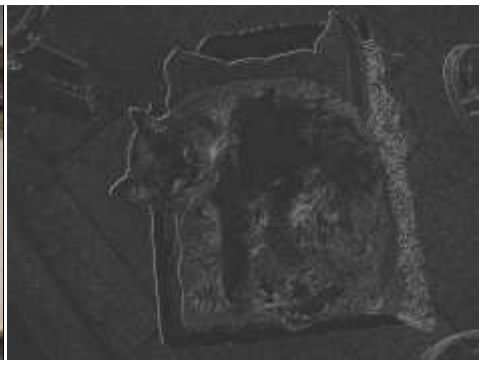
  По оси X:$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$

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

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

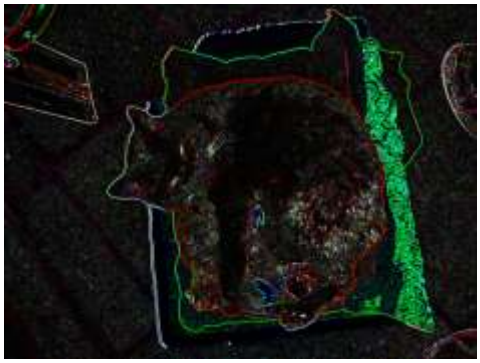
Список фильтров

Тиснение

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

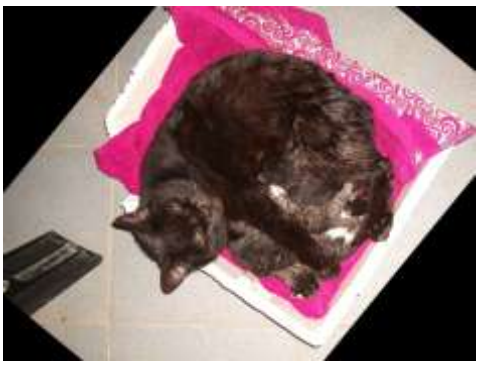
Светящиеся края

Медианный фильтр + выделение контуров + фильтр "максимума"

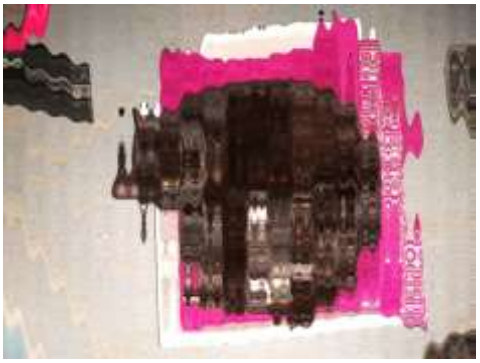
Перенес/Поворот

Перенос : $\begin{cases} x(k,l) = k + 50;\ k,l - \text{индексы для которых вычисляем цвет} \\ y(k,l) = l; x,y - \text{Новые индексы, по которым берётся цвет} \end{cases}$



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



Волны1: $\begin{cases} x(k,l) = k + 20\sin(2\pi l/60); \\ y(k,l) = l; \end{cases}$ Волны2: $\begin{cases} x(k,l) = k + 20\sin(2\pi k/30); \\ y(k,l) = l; \end{cases}$

Эффект "Стекла"

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

Motion Blur


Ядро фильтра: $\frac{1}{n}\begin{pmatrix} 1 & 0 & 0 & \cdots & & & 0 \\ 0 & 1 & 0 & & & & \\ 0 & 0 & 1 & & & & \\ \vdots & & & \ddots & & & \vdots \\ & & & & 1 & 0 & 0 \\ & & & & 0 & 1 & 0 \\ 0 & & & \cdots & 0 & 0 & 1 \end{pmatrix}$, n - количество единиц, т.е. количество столбцов или строк.

Резкость


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

Задачи для сдачи лабораторной работы

- Выберите из раздела «Список фильтров» 2 фильтра, где операции производятся над индексами пикселей и два матричных фильтра, реализуйте.
- Реализуйте операции мат. морфологии dilation, erosion, opening, closing. Выберите одну из top hat, black hat, grad, реализуйте.
- Реализуйте медианный фильтр
- Добавьте возможность задать структурный элемент для операций мат. морфологии.
- Реализуйте линейное растяжение
- Реализуйте один из фильтров: «Серый мир», «Идеальный отражатель», «Коррекция с опорным цветом»