Оглавление

[Лабораторная работа № 3. Цифровая обработка изображений 1](#_Toc336643192)

[Простейшая реализация фильтра на основе свертки на примере фильтра Blur 1](#_Toc336643193)

[Принцип действия фильтров на основе свертки. 1](#_Toc336643194)

[Практические задания 15](#_Toc336643195)

[Обязательные задания 15](#_Toc336643196)

# Лабораторная работа № 3. Цифровая обработка изображений

## Простейшая реализация фильтра на основе свертки на примере фильтра Blur

### Принцип действия фильтров на основе свертки.

Фильтр размытия выполняет отображение исходного изображения в конечное при помощи свертки к значениям цвета в окрестностях каждого пикселя исходного изображения.

Ядром свертки фильтра размытия будет являться матрица, элементами которой являются удельные веса каждого пикселя окрестности данного пикселя.

Одной из простейших матриц фильтра размытия будет следующая матрица 3\*3:

Для простоты рассмотрим применение данной матрицы к grayscale-изображению, у которого яркость каждого пикселя определяется значением от 0 до 255.

Допустим, что значения яркости пикселей в окрестностях текущего обрабатываемого пикселя изображения заданы следующим образом:

Для определения цвета данного пикселя в результате применения фильтра выберем блок пикселей размером 3\*3 вокруг него. Размер блока совпадает с размером матрицы фильтра. На рисунке данных блок выделен желтой рамкой.

Далее нужно просто найти сумму произведений соответствующих элементов матрицы фильтра и яркостей пикселей:

Полученное значение 110,1 округляется до 110 и используется в качестве нового цвета данного пикселя. Аналогичным образом обрабатываются все остальные пиксели.

Внимание: Каждый пиксель должен обрабатываться независимо от остальных и его обновление не должно оказывать влияния на расчет цвета соседних пикселей. На практике это означает, что результат применения фильтра должен сохраняться в новое изображение, не модифицируя при этом старое в процессе применения фильтра.

#### Обработка граничных значений

Каким образом рассчитывать яркость пикселей вблизи границ изображения. Ведь при их обработке требуется также обработать и соседние к ним пиксели, выходящие за пределы изображения? Какой цвет выбрать для этих «заграничных» пикселей?

Вспомним, что растровое изображение представляет собой цифровой двухмерный сигнал – зависимость цвета (яркости) от координаты точки. Значения цифрового сигнала определены на заданном интервале координат, за пределами которого нам не известны. Необходимо выбрать один из возможных способов определения значения сигнала за пределами области определения.

На рисунке ниже представлено графическое изображение данной проблемы на примере одномерного сигнала. Как определить значение функции в точках, отмеченных красным цветом?

Возможны несколько вариантов:

* Считать их яркость равной нулю (или другому числу).
* Считать их яркость равной цвету ближайшего пикселя на границе изображения
* [Экстраполяция](http://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D1%81%D1%82%D1%80%D0%B0%D0%BF%D0%BE%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) – вычисление значений вне заданного интервала определения функции с использованием интерполяционных формул.

Мы выберем второй способ, т. к. он значительно проще экстраполяции и дает лучшие результаты по сравнению с первым.

Для фильтрации цветных изображений формата RGB (RGBA) вышеупомянутому преобразованию подвергается независимо каждая цветовая компонента пикселя.

#### Разработка класса «Матричный фильтр»

Поскольку фильтр размытия – не единственный фильтр на основе свертки, логично будет вынести реализацию матричного фильтра в базовый класс CMatrixFilter, который будет применять произвольную матрицу фильтра к двухмерному растровому изображению. От данного класса будет порожден класс CBlurFilter, выполняющий настройку матрицы перед применением фильтра.

class CMatrixFilter

{

public:

// Применяем фильтр к исходному изображению и возвращаем результат

std::auto\_ptr<Gdiplus::Bitmap> ApplyFilter(

Gdiplus::Bitmap & srcBitmap);

protected:

enum

{

// максимальный размер матрицы фильтра

// (больше делать смысла нет - производительность будет никакая)

MAX\_MATRIX\_SIZE = 50,

};

// Защищаем конструктор и деструктор матрицы,

// делая их доступными только для классов-наследников

CMatrixFilter(void);

virtual ~CMatrixFilter(void);

// Изменяет размер матрицы,

// а также опционально выполняет инициализацию ее элементов

void InitializeMatrix (

size\_t width,

size\_t height,

const float \* initialValues = NULL,

float scale = 1.0f);

// Методы доступа к коэффициентам матрицы свертки

void SetMatrixValue(size\_t row, size\_t column, float value);

float GetMatrixValue(size\_t row, size\_t column)const;

// Даем возможность классам-наследникам осуществить

// выбор предпочтительного формата пикселей для выходного изображения

virtual Gdiplus::PixelFormat ChoosePreferableOutputPixelFormat(

Gdiplus::Bitmap & bitmapData)const;

// Даем возможность классам-наследникам осуществить

// обработку цвета пикселя после применения к нему матричного фильтра

virtual void AdjustColor(

float & r, float & g, float & b, float & a,

Gdiplus::Color const& srcColor)const;

private:

// приводим цвет к диапазону от 0 до 255

inline static BYTE Clamp(float color)

{

return static\_cast<BYTE>(

(color < 0) ? 0 :

(color > 255) ? 255 :

color

);

}

private:

// коэффициенты матрицы фильтра

std::vector<float> m\_matrix;

// размеры матрицы фильтра

size\_t m\_matrixWidth;

size\_t m\_matrixHeight;

};

Пару слов о классе. Из публичных методов здесь только метод ApplyFilter, выполняющий применение матричного фильтра к некоторому растровому изображению. Остальные методы класса сделаны защищенными, дабы оставить доступ к ним только из классов-наследников, которые «знают» каким образом настроить матричный фильтр для реализации соответствующего эффекта.

Метод **AdjustColor** сделан виртуальным, чтобы классы-наследники имели возможность перегрузить его, добавив пост-обработку цвета отфильтрованного пикселя перед его записью в выходное изображение. Впрочем, об этом позже.

В конструкторе задаем начальный размер матрицы фильтра равный 1. При применении такой матрицы получится изображение, идентичное исходному.

CMatrixFilter::CMatrixFilter(void)

:m\_matrix(1, 1.0f)

,m\_matrixWidth(1)

,m\_matrixHeight(1)

{

}

CMatrixFilter::~CMatrixFilter(void)

{

}

Метод **InitializeMatrix** выполняет инициализацию матрицы – задает ее размер, а также опционально задает начальные значения ее элементов, умножая их на параметр scale. Метод иллюстрирует использование алгоритма [transform](http://msdn.microsoft.com/ru-ru/library/391xya49.aspx) в связке с функцией [bind1st](http://msdn.microsoft.com/ru-ru/library/sh813d0y.aspx) библиотеки STL для умножения каждого элемента массива на заданный параметр scale.

void CMatrixFilter::InitializeMatrix(

size\_t width, size\_t height,

const float \* initialValues, float scale)

{

if ((width > 0) && (width <= MAX\_MATRIX\_SIZE) &&

(height > 0) && (height <= MAX\_MATRIX\_SIZE))

{

m\_matrixWidth = width;

m\_matrixHeight = height;

if (initialValues != NULL)

{

m\_matrix.assign(&initialValues[0], &initialValues[width \* height]);

}

else

{

// задаем матрице начальный размер и обнуляем ее элементы

m\_matrix.assign(width \* height, 0.0f);

// Устанавливаем центральный элемент в 1

SetMatrixValue(width / 2, height / 2, 1);

}

if (scale != 1.0f)

{

// умножаем все элементы матрицы на масштабный множитель

std::transform(

m\_matrix.begin(), // начиная с первого элемента матрицы

m\_matrix.end(), // до последнего

m\_matrix.begin(), // результат помещаем в саму матрицу

std::bind1st(

std::multiplies<float>(),

scale) // bind1st создает унарную функцию из

// бинарной функции std::multiplies

// созданная функция умножает значение

// элемента матрицы на scale

);

}

}

else

{

throw std::out\_of\_range("Matrix dimensions is out of range");

}

}

Масштабирование элементов m\_matrix заслуживает некоторого пояснения. Как известно, оператор умножения является бинарным, а алгоритм **transform** требует передачи унарной функции (или функтора) в качестве одного из своих аргументов. Т.к. масштабный множитель одинаковый для всех элементов матрицы, то операция умножения из бинарной функции превращается в унарную операцию умножения на **scale**. Для конструирования такой функции используется функция **bind1st**, при помощи которой бинарная функция std::multiplies<float>(float a, float b) превращается, нужную нам операцию (scale \* x). Т.к. операция умножения является коммутативной, с таким же успехом можно было бы воспользоваться функцией [bind2nd](http://msdn.microsoft.com/ru-ru/library/3f0defz2.aspx), которая превратила бы ее в (x \* scale).

Для использования функций **transform** и **bind1st** необходимо подключить заголовочные файлы **<algorithm>** и **<functional>** соответственно.

Для масштабирования матрицы также можно было бы воспользоваться циклом **for**:

for (std::vector<float>::iterator it = m\_matrix.begin();

it != m\_matrix.end();

++it)

{

\*it \*= scale;

}

При помощи методов SetMatrixValue и GetMatrixValue классы-наследники осуществляют доступ к элементам матрицы:

void CMatrixFilter::SetMatrixValue(size\_t row, size\_t column, float value)

{

if ((row < m\_matrixHeight) && (column < m\_matrixWidth))

{

m\_matrix[row \* m\_matrixWidth + column] = value;

}

else

{

throw std::invalid\_argument("Row or Column index is out of range");

}

}

float CMatrixFilter::GetMatrixValue(size\_t row, size\_t column)const

{

if ((row < m\_matrixHeight) && (column < m\_matrixWidth))

{

return m\_matrix[row \* m\_matrixWidth + column];

}

else

{

throw std::invalid\_argument("Row or Column index is out of range");

}

}

Рассмотрим реализацию метода **ApplyFilter** по частям. В самом начале нам необходимо создать новое растровое изображение для хранения результата фильтра. Размеры выходного изображения мы будут равным размерам исходного, а с выбором формата пикселей результирующего изображения ситуация интереснее. В ряде случаев достаточно выбрать тот же формат, что и у исходного, а вот для некоторых фильтров лучше выбрать вполне определенный формат пикселей изображения (например, без альфа-канала). Поэтому данный код будет вынесен в виртуальный метод ChoosePreferableOutputPixelFormat, чтобы дать классам-наследникам возможность переопределить поведение по умолчанию:

Gdiplus::PixelFormat CMatrixFilter::ChoosePreferableOutputPixelFormat(

Gdiplus::Bitmap & bitmapData)const

{

return bitmapData.GetPixelFormat();

}

std::auto\_ptr<Gdiplus::Bitmap> CMatrixFilter::ApplyFilter(

Gdiplus::Bitmap & srcBitmap)

{

// Инициализируем переменные для хранения часто используемых значений

int w = srcBitmap.GetWidth();

int h = srcBitmap.GetHeight();

int matrixHalfWidth = m\_matrixWidth / 2;

int matrixHalfHeight = m\_matrixHeight / 2;

// создаем растровое изображение для хранения результата,

// давая возможность классам наследникам переопределить выбор

// формата пикселей изображения

std::auto\_ptr<Bitmap> pResult(

new Bitmap(w, h, ChoosePreferableOutputPixelFormat(srcBitmap)));

// Ссылка на результирующее изображение

Bitmap & dstBitmap = \*pResult;

Далее идет вложенный цикл по строкам и столбцам исходного изображения. Перед обработкой каждого пикселя инициализируем начальные значения цветовых составляющих нулями:

// цикл по строкам изображения

for (int y0 = 0; y0 < h; ++y0)

{

// цикл по столбцам изображения

for (int x0 = 0; x0 < w; ++x0)

{

int y = y0 - matrixHalfHeight;

// задаем начальное значение цвета пикселя

float r = 0, g = 0, b = 0, a = 0;

В окрестностях каждого пикселя исходного изображения применяется фильтр на основе свертки – еще два вложенных цикла по строкам и столбцам матрицы для вычисления взвешенной суммы цветов пикселей окрестности. Здесь также реализовано решение проблемы с определением цвета пикселей за границами изображения (они принимаются равными граничным пикселям).

// цикл по строкам матрицы

for (size\_t row = 0; row < m\_matrixHeight; ++row, ++y)

{

int x = x0 - matrixHalfWidth;

// ограничиваем координату y областью изображения

int srcY = min(h - 1, max(y, 0));

// цикл по столбцам матрицы

for (size\_t column = 0; column < m\_matrixWidth; ++column, ++x)

{

// ограничиваем координату x областью изображения

int srcX = min(w - 1, max(x, 0));

// получаем текущий коэффициент матрицы

float coeff = GetMatrixValue(row, column);

// используем его для обновления взвешенной суммы

// цвета пикселя

Color c;

srcBitmap.GetPixel(srcX, srcY, &c);

r += c.GetRed() \* coeff;

g += c.GetGreen() \* coeff;

b += c.GetBlue() \* coeff;

a += c.GetAlpha() \* coeff;

}

}

После вычисления составляющих результирующего цвета изображения необходимо дать возможность классу-наследнику скорректировать их значение. Для этого мы передаем в метод **AdjustColor** информацию об исходном и обработанном цвете пикселя, а затем записываем скорректированные цвета в выходное растровое изображение, не забыв предварительно ограничить значение результирующего цвета диапазоном от 0 до 255. В конце возвращаем выходной растр (управление временем жизни данного объекта берет на себя умный указатель std::auto\_ptr):

// корректируем цвет пикселя (данная операция реализуется

// в зависимости от типа фильтра)

Color srcColor;

srcBitmap.GetPixel(x0, y0, &srcColor);

AdjustColor(r, g, b, a, srcColor);

// Формируем цвет выходного пикселя и записываем его

// в выходное растровое изображение

Color dstColor(Clamp(a), Clamp(r), Clamp(g), Clamp(b));

dstBitmap.SetPixel(x0, y0, dstColor);

}

}

return pResult;

}

void CMatrixFilter::AdjustColor(

float & /\*r\*/, float & /\*g\*/, float & /\*b\*/, float & /\*a\*/,

Gdiplus::Color const& /\*srcColor\*/)const

{

// оставляем без изменения r, g, b и a составляющие цвета

// этот метод может быть перегружен в производном классе

}

#### Класс CBlurFilter – фильтр размытия

Теперь, когда большая часть функционала реализована, осталось реализовать класс **CBlurFilter**, являющийся наследником класса CMatrixFilter, в котором выполнить необходимую настройку матричного фильтра:

class CBlurFilter : public CMatrixFilter

{

public:

CBlurFilter(void);

};

CBlurFilter::CBlurFilter(void)

{

/\*

Инициализируем ядро свертки следующей матрицей 3\*3

|1 3 1|

1/20 \* |3 4 3|

|1 3 1|

\*/

static const float blurMatrix[3][3] =

{

{1, 3, 1},

{3, 4, 3},

{1, 3, 1}

};

InitializeMatrix(3, 3, &blurMatrix[0][0], 1.0f / 20);

}

Использование класса CBlurFilter предельно простое: нужно создать экземпляр данного класса и вызывать у него метод **ApplyFilter**, передав ссылку на обрабатываемое растровое изображение в качестве параметра:

CMainView::CMainView()

{

Image img(L"car.jpg");

if (img.GetLastStatus() == Ok)

{

m\_pSrcPicture.reset(

new Bitmap(img.GetWidth(), img.GetHeight(), img.GetPixelFormat()));

Graphics g(m\_pSrcPicture.get());

g.DrawImage(&img, 0, 0, img.GetWidth(), img.GetHeight());

CBlurFilter blur;

m\_pDstPicture = blur.ApplyFilter(\*m\_pSrcPicture.get());

}

}

В обработчике сообщения WM\_PAINT выведем исходное и фильтрованное изображения:

LRESULT CMainView::OnPaint(

UINT /\*uMsg\*/, WPARAM /\*wParam\*/, LPARAM /\*lParam\*/, BOOL& /\*bHandled\*/)

{

CPaintDC dc(m\_hWnd);

// Если исходное и обработанное изображения заданы,

// то нарисуем их рядом друг с другом

if ((m\_pSrcPicture.get() != NULL) && (m\_pDstPicture.get() != NULL))

{

Graphics g(dc);

g.DrawImage(m\_pSrcPicture.get(),

0, 0, m\_pSrcPicture->GetWidth(), m\_pSrcPicture->GetHeight());

g.DrawImage(m\_pDstPicture.get(),

m\_pSrcPicture->GetWidth(), 0,

m\_pDstPicture->GetWidth(), m\_pDstPicture->GetHeight());

}

return 0;

}

Результат работы программы представлен на следующем рисунке:



Изображение справа получилось слегка размытым. Степень размытия можно увеличить путем увеличения размеров матрицы фильтра и инициализации ее соответствующими значениями, например, [двухмерным распределением Гаусса](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%93%D0%B0%D1%83%D1%81%D1%81%D0%B0) (нормальное распределение). В таком случае фильтр будет называться [фильтром Гаусса](http://ru.wikipedia.org/wiki/%D0%A4%D0%B8%D0%BB%D1%8C%D1%82%D1%80_%D0%93%D0%B0%D1%83%D1%81%D1%81%D0%B0) (Гауссово размытие).

#### Повышение скорости работы фильтра

Скорость разработанного нами фильтра оставляет желать лучшего. Время обработки изображения разрешением 14,9 мегапикселей составляет порядка 125 секунд на процессоре AMD Athlon 64 X2 Dual 2.11 GHz, что соответствует скорости обработки немногим более чем 100 тысяч пикселей в секунду.

Причина столь низкой производительности заключается в том, что доступ к отдельным пикселям растрового изображения в GDI+ чрезвычайно низкая. Гораздо быстрее было бы работать напрямую с массивом RGBA-значений цветов пикселей изображения. Для этого класс Bitmap предоставляет методы [LockBits](http://msdn.microsoft.com/en-us/library/ms536298%28v=vs.85%29.aspx) и [UnlockBits](http://msdn.microsoft.com/en-us/library/ms536301%28v=vs.85%29.aspx), при помощи которых можно получить прямой доступ к прямоугольной области растрового изображения в заданном формате для записи либо чтения.

Обновленный код метода **CMatrixFilter::ApplyFilter**, использующий упомянутую выше оптимизацию представлен ниже:

std::auto\_ptr<Gdiplus::Bitmap> CMatrixFilter::ApplyFilter(

Gdiplus::Bitmap & srcBitmap)

{

// Инициализируем переменные для хранения часто используемых значений

int w = srcBitmap.GetWidth();

int h = srcBitmap.GetHeight();

int matrixHalfWidth = m\_matrixWidth / 2;

int matrixHalfHeight = m\_matrixHeight / 2;

// создаем растровое изображение для хранения результата,

// давая возможность классам наследникам переопределить выбор

// формата пикселей изображения

std::auto\_ptr<Bitmap> pResult(

new Bitmap(w, h, ChoosePreferableOutputPixelFormat(srcBitmap)));

// Ссылка на результирующее изображение

Bitmap & dstBitmap = \*pResult;

// Размеры и положение прямоугольной области для осуществления

// прямого доступа

Rect lockRect(0, 0, srcBitmap.GetWidth(), srcBitmap.GetHeight());

// Получаем прямой доступ ко всем пикселям исходного

// изображения для чтения

BitmapData srcData;

srcBitmap.LockBits(

&lockRect, Gdiplus::ImageLockModeRead, PixelFormat32bppARGB, &srcData);

BYTE const \* pSrcData = reinterpret\_cast<BYTE const\*>(srcData.Scan0);

// Получаем прямой доступ ко всем пикселям результирующего

// изображения для записи

BitmapData dstData;

dstBitmap.LockBits(

&lockRect, Gdiplus::ImageLockModeWrite, PixelFormat32bppARGB, &dstData);

BYTE \* pDstData = reinterpret\_cast<BYTE\*>(dstData.Scan0);

// цикл по строкам изображения

for (int y0 = 0; y0 < h; ++y0)

{

// Вычисляем адрес начала строки с индексом y0 в исходном изображении

const DWORD \* pSrcLine =

reinterpret\_cast<const DWORD\*>(pSrcData + (srcData.Stride \* y0));

// Вычисляем адрес начала строки с индексом y0 в

// результирующем изображении

DWORD \* pDstLine =

reinterpret\_cast<DWORD\*>(pDstData + (dstData.Stride \* y0));

// цикл по столбцам изображения

for (int x0 = 0; x0 < w; ++x0)

{

int y = y0 - matrixHalfHeight;

// задаем начальное значение цвета пикселя

float r = 0, g = 0, b = 0, a = 0;

// цикл по строкам матрицы

for (size\_t row = 0; row < m\_matrixHeight; ++row, ++y)

{

int x = x0 - matrixHalfWidth;

// ограничиваем координату y областью изображения

int srcY = min(h - 1, max(y, 0));

// Вычисляем адрес начала строки исходного изображения,

// участвующей в свертке

const DWORD \* pSrcFilterLine =

reinterpret\_cast<const DWORD\*>(pSrcData +

(srcData.Stride \* srcY));

// цикл по столбцам матрицы

for (size\_t column = 0; column < m\_matrixWidth; ++column, ++x)

{

// ограничиваем координату x областью изображения

int srcX = min(w - 1, max(x, 0));

// Получаем цвет пикселя, участвующего в фильтре свертки

// в виде 32-битного целого числа в формате 0xAARRGGBB

DWORD c = pSrcFilterLine[srcX];

// получаем текущий коэффициент из матрицы и

// используем его для обновления взвешенной суммы

// цвета пикселя

float coeff = GetMatrixValue(row, column);

a += ((c >> 24) & 0xff) \* coeff;

r += ((c >> 16) & 0xff) \* coeff;

g += ((c >> 8) & 0xff) \* coeff;

b += (c & 0xff) \* coeff;

}

}

// Получаем цвет исходного пикселя в виде 32-битного

// целого числа в формате 0xAARRGGBB

DWORD srcColor = pSrcLine[x0];

// корректируем цвет пикселя (данная операция реализуется

// в зависимости от типа фильтра)

AdjustColor(r, g, b, a, srcColor);

// Формируем цвет выходного пикселя и записываем его

// в выходное растровое изображение

pDstLine[x0] = MakeARGB(Clamp(a), Clamp(r), Clamp(g), Clamp(b));

}

}

// Вызываем UnlockBits для того, чтобы просигнализировать о завершении

// операций прямого доступа к исходному и результирующему изображениям.

srcBitmap.UnlockBits(&srcData);

// Поскольку к результирующему изображению был получен доступ для записи

// то по при вызове UnlockBits данные из массива результирующих пикселей

// будут записаны в результирующее растровое изображение

dstBitmap.UnlockBits(&dstData);

return pResult;

}

Небольшое обновление было сделано и в виртуальном методе CMatrixFilter::AdjustColor, связанное с тем, что для работы с цветом используется тип DWORD, а не класс Color.

class CMatrixFilter

{

...

virtual void AdjustColor(

float & r, float & g, float & b, float & a, DWORD srcColor)const;

...

};

Реализация данного метода в классе CMatrixFilter останется прежней:

void CMatrixFilter::AdjustColor(

float & /\*r\*/, float & /\*g\*/,

float & /\*b\*/, float & /\*a\*/, DWORD /\*srcColor\*/)const

{

// оставляем без изменения r, g, b и a составляющие цвета

// этот метод может быть перегружен в производном классе

}

В результате проделанных оптимизаций быстродействие фильтра увеличится более чем в 12 раз. Тестовое изображение разрешением 14,9 мегапикселя будет обработано за 9,5 секунд.

##### Повышение быстродействия за счет многопоточности

Процессоры современных компьютеров, как правило, содержат несколько ядер, работающих параллельно, что позволяет добиться еще большей производительности. Повысить быстродействие нашего приложения можно за счет разбиения задачи на несколько независящих друг от друга подзадач, которые можно было бы выполнить в отдельных потоках. Операционная система сама позаботится о том, чтобы распределить ресурсы процессора между потоками приложения.

Узким местом нашего фильтра, потребляющим больше всего процессорного времени, является, собственно, работа самого фильтра. При этом каждый пиксель изображения обрабатывается независимо от остальных[[1]](#footnote-1), что позволяет распределить обработку изображения между несколькими потоками.

Одним из самых простых способов распределить работу между потоками – это создать количество потоков, равное количеству ядер, и каждому ядру поручить задачу обработки своей доли изображения. На следующем рисунке показан процесс распределения задачи обработки изображения между двумя потоками.



Главный поток должен сначала создать необходимое количество рабочих потоков, поручив каждому из них обработку соответствующего фрагмента изображения, а затем дождаться завершения работы всех рабочих потоков. Важно, чтобы всем потокам было поручено примерно одинаковое количество работы, чтобы сократить время ожидания[[2]](#footnote-2).

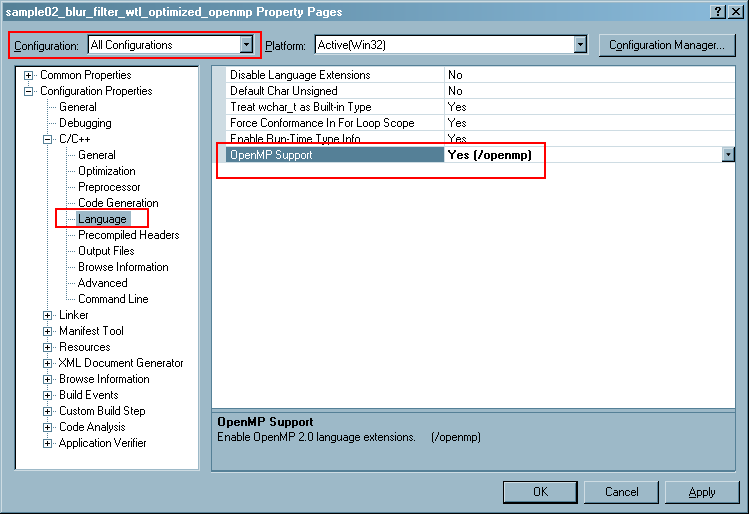


Создать потоки и дождаться окончания их выполнения можно при помощи соответствующих функций Windows API[[3]](#footnote-3).

Более простым решением будет использование специальных библиотек, упрощающих распараллеливание вычислений.

Одной из таких библиотек, имеющей встроенную поддержку в Visual Studio 2008, является библиотека [OpenMP](http://openmp.org). С ее помощью очень просто[[4]](#footnote-4) будет добавить поддержку многопоточности в наше приложение.

Для включения поддержки OpenMP в нашем проекте необходимо воспользоваться опцией OpenMP Support в свойствах проекта.



Реализация функций библиотеки OpenMP, поддерживаемой компилятором Visual Studio 2008, располагается в .dll файлах, которые необходимо распространять вместе с нашим приложением, поскольку в состав операционной системы они по умолчанию не входят.

Release-версия .dll файла библиотеки OpenMP для 32-битных приложений, предназначенная для распространения вместе с использующим ее приложением, может быть найдена в каталоге

**{Каталог установки Visual C++ 2008}\redist\x86\Microsoft.VC90.OPENMP**

Отладочная версия .dll файла библиотеки, предназначенная для целей отладки на компьютере разработчика (но не для распространения вместе с приложением) располагается в каталоге:

**{Каталог установки Visual C++ 2008}\redist\Debug\_NonRedist\x86\Microsoft.VC90.DebugOpenMP**

Файлы с расширениями .dll и .manifest необходимо скопировать в папку с исполняемым файлом нашего приложения:

* отладочную версию поместить в папку, в которую помещается отладочная версия приложения при сборке проекта
* Release-ную версию библиотеки следует поместить в каталог, в который помещается Release-версия приложения

Если все сделать правильно, то при запуске нашего приложения, собранного с поддержкой OpenMP, никаких сообщений об отсутствии .dll файла выдаваться не должно.

Теперь осталось превратить внешний цикл for (по строкам исходного изображения) в методе CMatrixFilter::ApplyFilter в пареллельный при помощи соответствующей директивы OpenMP.

std::auto\_ptr<Gdiplus::Bitmap> CMatrixFilter::ApplyFilter(

Gdiplus::Bitmap & srcBitmap)

{

// цикл по строкам изображения

// Директива #pragma omp parallel for дает сигнал компилятору о том, что

// итерации данного цикла for должны выполняться параллельно

// в нескольких потоках.

// Компилятор сам сгенерирует код, обеспечивающий создание потоков

// и передачу им соответствующих параметров

// Более подробно можно ознакомиться в спецификации библиотеки OpenMP

// на сайте openmp.org

#pragma omp parallel for

for (int y0 = 0; y0 < h; ++y0)

{

...

}

...

}

Теперь во время работы фильтра приложение будет использовать ресурсы нескольких доступных процессоров или ядер, обеспечивая заметный[[5]](#footnote-5) прирост в быстродействии.

## Практические задания

Для получения оценки «**удовлетворительно**» необходимо набрать **не менее 60 баллов**.

Для получения оценки «**хорошо**» необходимо набрать **не менее 250 баллов**.

Для получения оценки «**отлично**» необходимо набрать **не менее 400 баллов**.

### Обязательные задания

#### Задание 1 – Фильтрация изображений – 20 баллов (базовая часть)

Разработайте приложение, позволяющее пользователю открывать графические файлы форматов JPG, PNG, BMP, TIFF и GIF и применять к ним различные фильтры. Применение фильтра осуществляется путем выбора команды в меню или на панели инструментов.

В приложении должен быть реализован, **как минимум**, один из ниже перечисленных фильтров. В случае реализации нескольких фильтров каждый последующий фильтр должен применяться к результату ранее примененного фильтра.

##### Фильтр «Гауссово размытие» – 25 баллов

Фильтр гаусса. Размер ядра свертки – минимум 7\*7. Коэффициенты фильтра должны быть рассчитаны с использованием нормального распределения. Параметр **σ** – дисперсия, определяет степень размытости, - должен быть выбран так, чтобы работа фильтра была заметна. При расчете коэффициентов нормального распределения следить за тем, чтобы сумма коэффициентов матрицы была равна 1 (при необходимости – выполнить нормализацию).

###### Бонус – 40 баллов за интерактивное изменение радиуса фильтра и дисперсии

Бонус начисляется, если пользователь может задавать значение радиуса фильтра и дисперсии в диалоге настройки фильтра, которые отображается при выборе команды применения фильтра. Для выбора данных параметров должен использоваться элемент управления Slider (ползунок).



Рисунок 1 Диалог настройки фильтра Гаусса

При нажатии на Ok в диалоге настройки параметров фильтра изменения применяются, на Cancel – не применяются.

###### Бонус –20 баллов за предварительный просмотр

Начисляется дополнительно к предыдущему бонусу, если в диалоге настройки будет возможность включить предварительный просмотр при помощи check box-а «Preview». В данном режиме изображение должно откликаться на движения ползунка обновляя свой внешний вид. При нажатии на Ok фильтр окончательно применяется. При нажатии на Cancel – изображение принимает вид, в котором оно было до применения данного фильтра.



Рисунок 2 Предварительный просмотр

##### Фильтр «Выделение краев» (Edge detection) – 15 баллов

##### Фильтр «Чеканка» (Emboss) – 15 баллов

##### Фильтр «Смазывание движения» (Motion Blur) – по 5 баллов за Horizontal и Vertical

Необходимо реализовать вертикальный, либо (и) горизонтальный Motion Blur.

##### Фильтр повышения резкости (Sharpen) – 15 баллов.

##### Фильтр шумоподавления «Медианный фильтр» (Median Blur) – 40 баллов

Фильтр имеет радиус 5\*5

###### Бонус - 30 баллов за возможность выбора радиуса фильтра

Бонус начисляется, если у пользователя есть возможность выбора размера радиуса фильтра.

###### Бонус – до 80 баллов за возможность просмотра предварительного результата фильтра

Начисляется дополнительно к предыдущему бонусу, если в диалоге настройки будет возможность включить предварительный просмотр небольшой[[6]](#footnote-6) части изображения при помощи check box-а «Предварительный просмотр».



Рисунок 3 Предварительный просмотр выключен

В режиме предварительного просмотра пользователь может видеть фрагмент исходного и обработанного изображения (в масштабе 1:1), при этом при изменении радиуса фильтра фрагмент обработанного изображения обновляется в реальном времени.

Пользователь может «схватить» при помощи мыши за точку внутри мини-окошке исходного или обработанного изображения (при этом курсор меняет свой внешний вид на ) и потянуть за нее для выбора области предварительного просмотра. При этом в другом окошке область просмотра меняется соответствующим образом. После отпускания мыши происходит перерасчет обработанного изображения.

median.wmf

Рисунок 4 Предварительный просмотр включен

##### Бонус - 10 баллов

Данный бонус начисляется, если будет реализована возможность сохранения фильтрованного изображения в файл одного из форматов: TIFF, JPG, PNG, BMP.

##### Бонус – 50 баллов за выполнение фильтра в фоновом потоке

Бонус начисляется за запуск фильтра в отдельном потоке, что позволит избежать подвисания пользовательского интерфейса приложения при обработке больших изображений. В приложении следует обеспечить необходимые механизмы синхронизации для доступа к объектам, общим для нескольких потоков[[7]](#footnote-7).

##### Бонус – 50 баллов за постепенное обновление изображения при обработке в фоновом потоке

Бонус начисляется дополнительно к предыдущему, если во время обработки изображения в фоновом потоке в окне будет периодически (раза 3-4 в секунду) отображаться обработанная часть изображения. В приложении следует обеспечить необходимые механизмы синхронизации для доступа к объектам, общим для нескольких потоков.

#### Задание 2 – Цветокоррекция изображения – 30 баллов (базовая часть)

Разработать приложение, позволяющее загружать изображение из файла в формате BMP, JPG, PNG, GIF и TIFF и выполнять над ним HSL-коррекцию, а также коррекцию яркости/контрастности.

**Как минимум должна быть реализована возможность корректирования хотя бы одной из вышеперечисленных характеристик изображения.**

Допускается объединение данной программы с программой из задания 1 в единое целое.

##### HSL-коррекция – 40 баллов

Пользователь при помощи команды меню или кнопки на панели инструментов вызывает диалог настройки Цвета, Насыщености и Яркости и при помощи слайдеров устанавливает необходимые значения соответствующих параметров. При нажатии на Ok изменения применяются к изображению. Окно настройки схематически представлено на следующем рисунке:



Рисунок 5 Макет диалога настройки цвета

Применение HSL коррекции должно быть реализовано своими силами без использования сторонних библиотек.

###### Бонус – 50 баллов за режим предварительного просмотра

Бонус присуждается за наличие режима Preview в диалоге настройки HSL-трансформации, позволяющего пользователю в реальном времени (на уменьшенной копии исходного изображения) увидеть результат применения трансформации еще до нажатия клавиши OK.



Рисунок 6 Предварительный просмотр выключен



Рисунок 7 Предварительный просмотр включен

###### Бонус – 10 баллов за ускорение обработки за счет многопоточности

Бонус присуждается за реализацию многопоточной обработки изображения с использованием библиотеки OpenMP. На защите необходимо продемонстрировать заметный прирост в производительности при использовании многопоточности по сравнению с обработкой в одном потоке.

##### Настройка яркости/контрастности – 40 баллов

Пользователь при помощи команды меню или кнопки на панели инструментов вызывает диалог настройки яркости и контрастности. При помощи слайдеров «Яркость» и «Констрастность» пользователь имеет возможность задать желаемые параметры яркости (от -100% до +100%) и контрастности (от -100% до +100%).

Изменение яркости/контрастности изображения требуется реализовать самостоятельно без использования сторонних библиотек

###### Бонус – 40 баллов за возможность предварительного просмотра

Бонус присуждается за наличие режима Preview в диалоге настройки яркости-контраста, позволяющего оценить результат применения фильтра (на уменьшенной копии изображения) еще до нажатия клавиши OK.



Рисунок 8 Предварительный просмотр выключен



Рисунок 9 Предварительный просмотр включен

###### Бонус – 10 баллов

Бонус присуждается за реализацию многопоточной обработки изображения с использованием библиотеки OpenMP. На защите необходимо продемонстрировать прирост в производительности при использовании многопоточности по сравнению с обработкой в одном потоке.

##### Бонус – 10 баллов

Бонус присуждается за возможность сохранения изображения в формате BMP, JPG, PNG и TIFF.

#### Задание 3 – Фильтр «Выборочный цвет» - 70 баллов

Разработать приложение, позволяющее пользователю загрузить изображение и применить к нему фильтр «Выборочный цвет»[[8]](#footnote-8), суть которого заключается в том, что все изображение становится черно-белым (в градациях серого), за исключением тех его областей, имеющих цвет, близкий к цвету, заданному пользователем.

Перед применением фильтра пользователь должен иметь возможность выбрать цвет, оттенки которого следует сохранить. Пример применения данного фильтра показан на следующем рисунке. Как можно заметить, изображение сохранило только цвета красного оттенка.

Пользователь должен иметь возможность сохранить обработанное изображение в файл формата PNG, BMP или JPEG.



Рисунок 10 Результат применения фильтра "Выборочный цвет"

1. Несмотря на то, что при работе фильтра на основе свертки используются данные соседних пикселей исходного изображения, исходное изображение не модифицируется, а в результирующем изображении на каждой итерации модифицируется только цвет текущего пикселя. [↑](#footnote-ref-1)
2. Согласно народной мудрости, телега едет со скоростью самой медленной лошади. В нашем случае время работы параллельного участка приложения будет равно времени работы самого тяжело нагруженного потока [↑](#footnote-ref-2)
3. Пытливому студенту не составит труда ознакомиться с процессом создания многопоточных приложений в соответствующем разделе MSDN:

   <http://msdn.microsoft.com/en-us/library/ms684254%28v=VS.85%29.aspx> [↑](#footnote-ref-3)
4. Несмотря на простоту приведенного здесь решения, разработка многопоточных приложений требует от программиста соответствующих знаний и навыков [↑](#footnote-ref-4)
5. Прирост производительности становится заметнее на больших изображениях, поскольку создание потоков и переключение между ними, требуют определенных вычислительных ресурсов. В системе Windows нет большого смысла в распараллеливании задачи, выполняющейся быстрее сотни-другой миллисекунд.

   На двухъядерном процессоре автора время обработки изображения сократилось почти в 2 раза (до 4,9 секунд). [↑](#footnote-ref-5)
6. Применение медианного фильтра большого радиуса может занимать продолжительное время, поэтому для предварительного просмотра в реальном времени следует применять фильтр лишь к небольшой части изображения. [↑](#footnote-ref-6)
7. Автор настоятельно рекомендует ознакомиться с библиотекой Boost.thread, значительно упрощающей создание многопоточных кроссплатформенных C++ приложений:

   <http://www.boost.org/doc/libs/1_51_0/doc/html/thread.html> [↑](#footnote-ref-7)
8. Данное название фильтра используется в цифровых фотоаппаратах Nikon Coolpix. Аналогичная функция в фотоаппаратах фирмы Canon называется «Акцентирование цветом». [↑](#footnote-ref-8)