МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по компьютерной графике

Учебно-методическое пособие
по дисциплине «Инженерная и компьютерная графика»

Часть 1. Графика в Windows

2016 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по компьютерной графике. Учебно-методическое пособие по дисциплине «Инженерная и компьютерная графика». Часть 1. Графика в Windows. Озерск: ОТИ НИЯУ МИФИ, 2016. — 39 с.

В пособии подробно излагается, как выполнять практические работы по дисциплине «Инженерная и компьютерная графика». Работы первой части изучения дисциплины включают в себя методы и алгоритмы рисования в Windows.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий 5
1. Растровые картинки 6
1.1. Пиксельный набор 6
1.2. Структура графического файла 7
1.3. Рабочее пространство 8
1.4. Открытие файла 9
1.5. Чтение заголовка файла 9
1.6. Чтение информации о картинке 10
1.7. Чтение набора пикселей 11
1.8. Совместимый контекст 11
1.9. Вывод картинки 12
1.10. Событие WM_PAINT 12
1.11. Дополнительные вопросы 12
2. Метафайлы 13
2.1. Структура метафайла 13
2.2. Создание метафайла 14
2.3. Рабочее пространство 14
2.4. Метафайл с автоматическим определением области вывода 14
2.5. Метафайл с заданной областью вывода 15
2.6. Проигрывание файла метафайла 15
2.7. Дополнительные вопросы 15
3. Работа со шрифтами 16
3.1. Шрифты 16
3.2. Надписи 16
3.3. Рабочее пространство 17
3.4. Использование стандартных шрифтов 17
3.5. Выбор шрифта при помощи стандартного диалога 18
3.6. Создание шрифта 18
3.7. Дополнительные вопросы 18
4. Рисование мышью 19
4.1. Сообщения о действиях с мышью 19
4.2. Рабочее пространство 20
4.3. Файл инициализации 21
4.4. Палитра цветов 23
4.5. Рисование мышью 28
4.6. Выравнивание линий 29
5. Основные графические примитивы 30
5.1. Рабочее пространство 30
5.2. Проектирование меню 30
5.3. Очистка поля 35
5.4. Рисование мышью 35
5.5. Отмена действия 37

5.6. Рисование графических примитивов.....37
5.7. Дополнительные вопросы.....37
6. Растровые алгоритмы.....38
6.1. Алгоритмы.....38
6.2. Рабочее пространство.....38
6.3. Дополнительные вопросы.....38
Литература.....39

Общие цели занятий

В ходе практических работ предлагается изучить основы программирования графических примитивов в среде Windows. В этой части работ рассматриваются следующие темы:

1) растровые картинки;
2) метафайлы;
3) шрифты;
4) рисование мышью;
5) графические примитивы;
6) растровые алгоритмы.

Основные сведения по графическим примитивам и функциям Windows приведены в учебно-методическом пособии автора «Компьютерная графика. 2006», а растровые алгоритмы — в пособии «Машинная графика. 2006». Наиболее актуальные версии этих документов доступны в Интернет по адресу http://revol.ponocom.ru.

На выполнение каждой работы предположительно отводится 2 академических часа, однако некоторые, наиболее сложные работы могут потребовать большего времени.

Поэтому, в зависимости от количества учебных часов, выделенных на проведение практических работ, сложности работ и навыков обучающегося, преподаватель может выбирать индивидуальные траектории работ.

Предполагается, что перед выполнением работы обучающийся самостоятельно изучает описание работы и изучаемые структуры и функции. Это позволит сэкономить время, отведенное на выполнение работы.

Каждая выполненная работа должна быть защищена.

1. Растровые картинки

Цели:
- изучение структур пиксельного набора.

Задачи:
- формирование структур пиксельного набора;
- чтение графического файла типа bmp;
- отображение пиксельного набора в окне;
- обработка события WM_PAINT.

1.1. Пиксельный набор

Пиксельным набором здесь называется пиксельное графическое изображение (*bitmap*), хранимое в памяти. Пиксельные наборы используются для работы с растровыми графическими изображениями. Фактически это массив цветов пикселей прямоугольной области размером BMPX×BMPY, полагая, что BMPX — ширина области набора, BMPY — высота. Далее вместо слов «пиксельный набор» будем пользоваться словом «картинка».

Различают DIB и DDB картинки.

DIB означает *Device-Independent Bitmap*, набор пикселей, независимый от устройства. В этом случае подразумевается, что цвета, заданные в пиксельном наборе, не зависят от устройства, а при выводе на устройство заданным в наборе цветам сопоставляются наиболее подходящие.

DDB означает *Device-Dependent Bitmap*, набор пикселей, зависимый от устройства. В этом случае подразумевается, что цвета картинки в точности совпадают с цветами устройства.

Например, графический файл формата bmp описывает *dib*-картинку, в то время как вывод ее осуществляется на конкретное устройство, для чего картинка должна быть преобразована в *ddb*-картинку.

Для описания *dib*-картинки используются информационные структуры, без которых невозможно отображение картинки на устройстве. Например, графический файл формата bmp описывает эти структуры в явном виде. Целью данной работы как раз является изучение этих структур, чтение их из файла, чтение из файла пиксельного набора, и отображение.

Важными характеристиками пиксельного набора являются:
- количество плоскостей;
- ширина BMPX и высота BMPY картинки;
- размер пиксельного набора.
- количество бит на пиксель;

Количество плоскостей в графических файлах типа bmp не используется и всегда равно единице.

Размеры картинки требуются для ее вывода на устройство.

Размер пиксельного набора нужен для того, чтобы знать, сколько байт занимает собственно набор при чтении графического файла.

Количество бит на пиксель (*глубина цвета*) является одной из самых важных характеристик, от этой характеристики зависит не только качество картинки, но и структура графического файла типа bmp.

Возможные значения глубины цвета равны 1, 4, 8, 24.

Значение 1 означает черно-белую (монохромную) картинку, палитра содержит 2 цвета, при этом для задания цвета используется один бит.

Значение 4 означает, что используется палитра из 16 цветов, а цвет пикселя в наборе задается половиной байта.

Значение 8 означает, что используется палитра из 256 цветов, а цвет задается в наборе одним байтом.

Значение 24 означает, что палитра не используется, а цвет задается в наборе непосредственно.

Файл типа bmp, описывающий картинку с глубиной цвета, не равной 24, имеет в своем составе *палитру цветов*, а пиксельный набор содержит *индексы* цветов палитры, а не сами цвета. Файл, описывающий картинку с глубиной цвета 24, *не содержит палитру*, а пиксельный набор состоит из структур, описывающих цвет одного пикселя в виде составляющих RGB.

Вам надлежит выучить наизусть цвета 16-цветной палитры.

1.2. Структура графического файла

Здесь рассматривается структура графического файла типа bmp.

Файл типа bmp состоит из 4-х разделов:

- заголовок файла в виде структуры BITMAPFILEHEADER;
- информация о наборе в виде структуры BITMAPINFOHEADER;
- палитра цветов (если есть);
- массив цветов пикселей (*индексы* в палитре или сами *цвета*).

Размер первой структуры равен 14 байт. Первые два байта имеют значение, отображающееся в буквы "BM", означающие BITMAP. Структура описана в файле wingdi.h и имеет следующий вид:

```
typedef struct tagBITMAPFILEHEADER {
    UINT  bfType;         // признак BM
    DWORD bfSize;         // размер файла
    UINT  bfReserved1;
    UINT  bfReserved2;
    DWORD bfOffBits;      // смещение к массиву пиксельного набора
} BITMAPFILEHEADER;
```

Буквы "BM" задаются полем bfType.

Поле bfSize задает размер файла в байтах.

Поле bfOffBits задает смещение к массиву пиксельного набора.

С помощью этой структуры определяется, что файл является действительным файлом типа bmp. Для этого сравниваются первые два байта файла, а размер файла сопоставляется с действительным.

Размер второй структуры равен 40 байт. Структура имеет вид:

```
typedef struct tagBITMAPINFOHEADER {
    DWORD  biSize;
    LONG   biWidth;          // ширина картинки
    LONG   biHeight;         // высота картинки
    WORD   biPlanes;         // для BMP всегда 1
    WORD   biBitCount;       // глубина цвета, 1, 4, 8, 24
    DWORD  biCompression;    // сжатие
    DWORD  biSizeImage;      // размер набора пикселей
    LONG   biXPelsPerMeter;
    LONG   biYPelsPerMeter;
    DWORD  biClrUsed;
    DWORD  biClrImportant;
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

В этой структуре описываются характеристики пиксельного набора.

Эта структура входит в состав структуры для считывания из файла не только информации о картинке, но и палитры целиком:

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD          bmiColors[1];
} BITMAPINFO, *PBITMAPINFO;
```

Здесь структура RGBQUAD описывает один элемент палитры:

```
typedef struct tagRGBQUAD {
   BYTE rgbBlue;
   BYTE rgbGreen;
   BYTE rgbRed;
   BYTE rgbReserved;
} RGBQUAD;
```

Если картинка имеет глубину цвета 1, 4 или 8, то с помощью структуры BITMAPINFO из файла считывается структура BITMAPINFOHEADER и палитра целиком. Поскольку в структуре определен всего один элемент RGBQUAD, то при считывании задается размер, соответствующий размеру структуры BITMAPINFO и всей палитры.

Например, если картинка имеет глубину цвета 4 (палитра 16 цветов), считываемый размер определяется следующим образом:

```
int cbmi = sizeof(BITMAPINFO) + sizeof(RGBQUAD) * 15;
```

Считывание производится следующим образом:

```
BITMAPINFO * pbmi = (BITMAPINFO*)LocalAlloc(LPTR, cbmi);
if (!fread((void*)pbmi, cbmi, 1, f)) {
    return 0;
}
```

Здесь подразумевается, что файл открыт с помощью структуры FILE, а переменная f — указатель на файловую структуру.

Если глубина цвета 24, то считывается структура BITMAPINFO без второго поля, считываемый размер при этом можно вычислить так:

```
int cbmi = sizeof(BITMAPINFO) - sizeof(RGBQUAD);
```

1.3. Рабочее пространство

Работа выполняется в приложении BMAP, подготовленном преподавателем. Архив проекта распаковывается на диск C:.

При этом в каталоге C:\BMAP\Debug должны находиться четыре файла: ch01.bmp, ch04.bmp, ch08.bmp, ch24.bmp. Название файла указывает на глубину цвета картинки.

Откройте проект и изучите его устройство. При защите работы будет контролироваться знание назначения всех функций и переменных. При необходимости используйте источник [2].

Основные действия производятся в функции read_bmp(), которая расположена в начале модуля BMAP.cpp:

```
#define PATH "ch01.bmp"
// чтение картинки
int read_bmp() {
    return 1;
}
```

Константа PATH задает путь к файлу.

В модуле BMAP.h располагаются глобальные константы и переменные.

Кроме того, действия будут также происходить в оконной процедуре в обработке события WM_PAINT.

Функция read_bmp вызывается в главной функции.

1.4. Открытие файла

Функция read_bmp().

Сначала нужно открыть файл для чтения в бинарном режиме.

Будем использовать для операций с файлом структуру FILE, а для открытия файла, путь к которому задается переменной file_name, будем использовать функцию fopen(). Пример, показывающий, как открыть файл:

```
FILE * f = 0;
f = fopen(PATH, "rb");
```

После открытия файла проверяем значение переменной f, и если оно равно нулю, завершаем функцию, возвращаем 0.

1.5. Чтение заголовка файла

Если файл открыт, считываем первую структуру, из которой состоит файл bmp, а именно структуру BITMAPFILEHEADER.

Для этого ее нужно объявить, переменная bmfh.

Затем с помощью функции fread() считываем заголовок файла:

```
int res = fread((void *) & bmfh, sizeof(bmfh), 1, f);
```

Здесь второй параметр — размер считываемой информации, третий параметр — количество считываемых единиц.

После считывания проверяем значение переменной res, и если оно равно нулю, завершаем функцию, возвращаем 0.

В этом месте нужно остановить выполнение программы и изучить считанную информацию. Эту информацию нужно сравнить с дампом исследуемого файла. Чтобы открыть дамп в файловом менеджере FAR, установите указатель на файл и нажмите F3, а затем, если необходимо, F4.

Исследования следует выполнить для всех графических файлов.

При защите этой части будет контролироваться знание расположения элементов структуры в дампе.

В завершении этой части нужно произвести проверку того, что считываемый файл является действительным файлом bmp. Для этого нужно убедиться, что первые два байта файла составляют буквы "BM", и сравнить действительную длину файла со значением поля bfSize. Для простоты можно проверить только значение поля bfType.

1.6. Чтение информации о картинке

В этой части сначала нужно считать из файла в динамическую структуру BITMAPINFOHEADER информационную часть графического файла, проанализировать ее и сопоставить с дампом. Сделать это нужно для каждого имеющегося файла.

Поскольку перед чтением информационной части неизвестно, какая глубина цвета используется в картинке, нужен какой-то прием, обходящий это недоразумение.

Например, можно сначала прочитать структуру BITMAPINFOHEADER, выяснить глубину цвета, рассчитать размер структуры BITMAPINFO и палитры вместе, и прочитать их заново, переустановив указатель позиции файла.

Другой вариант — вычислить необходимый размер по имеющимся после чтения заголовочной структуры файла данным, и сразу прочитать информационную часть вместе с палитрой.

BITMAPFILEHEADER
BITMAPINFOHEADER
Палитра
Массив пиксельного набора

В структуре BITMAPFILEHEADER поле bfOffBits — это смещение к началу пиксельного набора, или, иначе, размер первых трех структур файла. Если из этого значения вычесть размер структуры BITMAPFILEHEADER, то получим размер структуры BITMAPINFOHEADER вместе с палитрой.

Пусть переменная cbmi — это требуемый размер:

```
// размер BITMAPINFO
int cbmi = bmfh.bfOffBits - sizeof(BITMAPFILEHEADER);
```

Объявить переменную для считывания BITMAPINFO как статическую нельзя, потому что размер этой структуры учитывает только один элемент палитры, поэтому нужно выделить память динамически, используя функцию LocalAlloc():

```
// структура
BITMAPINFO * pbmi = (BITMAPINFO *)LocalAlloc(LPTR, cbmi);
```

После этого можно считывать структуру:

```
res = fread((void*)pbmi, cbmi, 1, f);
```

Значение переменной res опять нужно проверять.

Здесь также нужно остановить выполнение программы для того, чтобы исследовать считанные структуры, сравнить их с дампом файла.

Для анализа элементов палитры можно объявить указатель

```
RGBQUAD * ppal = &pbmi->bmiColors[0];
```

Если установить этот указатель на второй элемент BITMAPINFO, то в окне просмотра переменных Watch можно ввести, например, ppal[1], и проанализировать второй элемент палитры.

При защите этой части будет контролироваться значение всех элементов структуры BITMAPINFOHEADER, в том числе значения цветов палитры.

1.7. Чтение набора пикселей

После того, как информационная часть, вместе с палитрой, прочитаны из графического файла, можно считывать пиксельный массив:

```
// массив пиксельного набора
BYTE * dibs = (BYTE*)LocalAlloc(LPTR, pbmi->bmiHeader.biSizeImage);
// читаем пиксельный набор
res = fread((void*)dibs, pbmi->bmiHeader.biSizeImage, 1, f);
```

После этого все необходимые элементы для создания картинки в памяти есть, и картинку можно создать:

```
// картинка
HBMP = CreateDIBitmap(hDCBMP, &pbmi->bmiHeader, CBM_INIT,
    dibs, pbmi, DIB_RGB_COLORS);
```

1.8. Совместимый контекст

Теперь мы готовы к тому, чтобы вывести картинку.

Как известно, рисовать можно только в контексте устройства.

На самом деле рисовать картинку будем в совместимом контексте.

Совместимый контекст, — это контекст, совместимый с устройством. Он создается функцией CreateCompatibleDC(). Переменная для совместимого контекста CDC объявлена как глобальная в модуле BMAP.h.

Переходим в модуль BMAP.h, функция InitInstance().

Создаем совместимый контекст. Для создания контекста требуется дескриптор окна hWndBMP, который в этой функции как раз создается.

Возвращаемся в функцию read_bmp и выбираем картинку в совместимый контекст с помощью функции SelectObject().

1.9. Вывод картинки

Картинка выводится функций BitBlt(). Для вывода нужно иметь:
- контекст окна, в которое выводится картинка hWndBMP;
- размеры картинки BMPX и BMPY;
- совместимый контекст CDC.

Пример вывода есть в [1].

Масштабировать картинку при выводе можно, если вместо функции BitBlt() использовать функцию StretchBlt(). Используйте для этой цели коэффициент масштабирования ScaleBlt, определенный в модуле BMAP.h.

1.10. Событие WM_PAINT

После того, как картинка выводится в окно, следует убедиться в том, что она исчезает, как только она будет чем-то закрыта. Например, можно просто переместить окно так, чтобы картинка попала за пределы экрана и затем переместить окно обратно.

Нужно понимать, что тот факт, что мы нарисовали что-то в окне, вовсе не означает, что это теперь там находится. Графический вывод тем и отличается, что нарисованное изображение нужно выводить в окно каждый раз, когда часть окна закрывается и открывается вновь.

Для этой цели существует событие WM_PAINT.

Оно сообщает окну (*оконной процедуре*), что часть окна или окно целиком требует перерисовки (если что-то было нарисовано).

Перерисовать можно все окно целиком, если алгоритм несложный.

Если алгоритм рисования сложный, можно перерисовывать только ту часть окна, которая очистилась. Эту часть окна всегда можно узнать. На самом деле часто бывает проще перерисовать все окно, чем вычислять его часть. Мы так и поступим, хотя именно в нашем случае часть картинки, которую нужно обновить, вычислить несложно.

На самом деле нужно всего лишь заново нарисовать картинку при помощи функции BitBlt() в обработке события WM_PAINT.

Чтобы это сработало, все переменные, используемые функцией BitBlt(), должны быть глобальными.

1.11. Дополнительные вопросы

Измените один из цветов палитры перед ее выводом.

2. Метафайлы

Цели:
- изучение структуры метафайла Win32.
Задачи:
- формирование метафайла;
- вывод метафайла;
- запись метафайла в файл.

2.1. Структура метафайла

Метафайл — это запись команд графического ядра *Windows*, которые проигрываются с помощью функции PlayEnhMetaFile().

Различают метафайлы для платформы *Win16* (файлы типа wmf), и улучшенные метафайлы для платформы *Win32* (файлы типа emf).

Улучшенный метафайл (далее просто метафайл) состоит из массива записей, образующих следующие разделы:
- заголовок;
- необязательная таблица дескрипторов графических объектов;
- необязательная палитра;
- список команд для воспроизведения — мета-записи.

Структуры для метафайлов определены в файле wingdi.h.

Заголовок метафайла описывается структурой ENHMETAHEADER:

```
typedef struct tagENHMETAHEADER {
    DWORD   iType;          // тип EMR_HEADER
    DWORD   nSize;          // размер структуры
    RECTL   rclBounds;      // границы
    RECTL   rclFrame;       // рамка
    DWORD   dSignature;     // подпись ENHMETA_SIGNATURE
    DWORD   nVersion;       // версия
    DWORD   nBytes;         // размер метафайла в байтах
    DWORD   nRecords;       // число записей
    WORD    nHandles;       // число дескрипторов
    WORD    sReserved;      // 0
    DWORD   nDescription;   // символов Unicode в описании
    DWORD   offDescription; // смещение к описанию или 0
    DWORD   nPalEntries;    // число элементов палитры
    SIZEL   szlDevice;      // разрешение устройства в pels
    SIZEL   szlMillimeters; // разрешение устройства в мм
    DWORD   cbPixelFormat;  // размер PIXELFORMATDESCRIPTOR или 0
    DWORD   offPixelFormat; // смещение к PIXELFORMATDESCRIPTOR или 0
    DWORD   bOpenGL;        // есть команды OpenGL
} ENHMETAHEADER, *PENHMETAHEADER, *LPENHMETAHEADER;
```

Размер структуры в разных версиях операционной системы может быть различным (за счет дополнительных полей). Аналогичная структура для метафайла *Win16* раза в два меньше по размеру.

Метафайл может иметь описание, которое в этом случае располагается после заголовка.

Одиночный нулевой символ в описании завершает одну строку описания, два нулевых символа завершают описание в целом.

Мета-записи идентифицируют функции, используемые для рисования изображения, а также параметры этих функций, если они есть. Для этих целей определена структура METARECORD:

```
typedef struct tagMETARECORD {
    DWORD   rdSize;
    WORD    rdFunction;
    WORD    rdParm[1];
  } METARECORD;
```

Каждой функции сопоставлен номер в виде константы вида EMR_XXX, XXX — название функции. Всего определено 120 функций и констант.

Так как разные функции GDI имеют разное количество параметров, размеры мета-записей различны.

2.2. Создание метафайла

Создать метафайла не сложно, и использовать описанные структуры при этом нет необходимости. Сначала создается контекст метафайла с помощью функции CreateEnhMetaFile(). Затем с помощью функций GDI рисуется изображение или задаются режимы отображения. В завершение контекст метафайла закрывается функцией CloseEnhMetaFile().

При создании контекста метафайла указывается путь к файлу, в который будет записан метафайл. Если путь к файлу не задан, метафайл создается только в памяти.

Функция CloseEnhMetaFile(), закрывающая контекст метафайла, возвращает графический объект типа HENHMETAFILE, который можно использовать в функции PlayEnhMetaFile() для воспроизведения. Для воспроизведения нужно задать также прямоугольную область в виде структуры RECT.

2.3. Рабочее пространство

Работа выполняется в рамках предыдущего проекта BMAP.

Рабочим модулем является BMAP.cpp. Основные действия выполняются в функции metafile(), расположенной за функцией read_bmp(). Эти две функции вызываются друг за другом, но вывод осуществляется в разные окна.

Для получения заданного изображения используются графические функции, такие как MoveToEx, LineTo, Polygon, Rectangle, Ellipse и другие. Описание функций есть, например, в [1].

2.4. Метафайл с автоматическим определением области вывода

Первый метафайл, который нужно создать, не задает область вывода. Нарисуйте графическое изображение государственного флага РФ. Файл метафайла назовите flag.emf.

Изображение должно выводиться в контекст hDCEMF и наблюдаться в правом окне приложения. Созданный в каталоге C:\BMAP\Debug метафайл можно просмотреть стандартными средствами Windows.

2.5. Метафайл с заданной областью вывода

Второй метафайл должен задавать область вывода в HIMETRIC.

Код, формирующий первый метафайл, следует закомментировать.

С помощью линий толщиной в 3 мм нарисуйте в этом метафайле рамку, расчерченную на квадраты линиями толщиной 1 мм.

Размер рамки 10×10 см, размер квадратов 5×5 см.

Выведите этот метафайл в файл quad.emf и в окно приложения.

2.6. Проигрывание файла метафайла

В заключение выведите в окно приложения любой из полученных в предыдущих частях работы файлов.

Получить графический объект типа HENHMETAFILE можно с помощью функции GetEnhMetafile(). Дополнительно см. [1].

2.7. Дополнительные вопросы

1. Найдите файл типа wmf и выведите его в окно приложения.

2. Сравните пиксельную графику и метафайл. Укажите достоинства и недостатки растровых картинок и метафайлов.

3. Работа со шрифтами

Цели:
- изучение функций GDI для работы со шрифтами.

Задачи:
- формирование шрифтов;
- выбор шрифтов;
- вывод надписей.

3.1. Шрифты

Шрифты — важная часть графической подсистемы. С помощью шрифтов выполняются все надписи, видимые на экране компьютера.

Шрифт — это коллекция символов, имеющих одинаковое начертание. Три важнейших элемента шрифта — это гарнитура (*typeface*), начертание (*style*) и размер (*size*).

Гарнитура определяет внешний вид символов разной шириной толстых и тонких основных штрихов, а также наличием или отсутствием засечек. Засечка (*serif*) — небольшой поперечный штрих на конце основного штриха. Шрифт без засечек принято называть *sans-serif*.

Начертание определяет жирность штрихов и наклон символов.

Наклон определяется терминами *roman* (прямые символы), *oblique* (наклонные символы) и *italic* (истинно наклонные символы).

Жирность (*weight*, вес) определяется числовыми значениями от 100 до 900, но в обычных приложениях используется только два типа жирности — нормальная и полужирная, со значениями 400 и 700. Другие значения используются в полиграфии.

Размер шрифта — это высота от верхней части буквы типа À (буквы с надстрочным знаком) до нижней части буквы типа g. Для измерения высоты часто используются *пункты*, принимаемые равными 1/72 дюйма.

Шрифт создается функциями CreateFont() или CreateFontIndirect(), которые возвращают дескриптор шрифта типа HFONT. Во втором случае используется структура LOGFONT.

Перед использованием шрифта его нужно выбрать в контекст.

Характеристики шрифта описываются структурой TEXTMETRIC.

3.2. Надписи

Для вывода надписей используются функции TextOut() и DrawText(). Различия между ними заключаются в возможностях форматирования текста при выводе, которых у второй функции больше.

Надписи выводятся выбранным в контекст шрифтом, при этом используется текущие цвета фона и плана. Цвет надписи задается с помощью функции SetTextColor(), а цвет фона — функциями SetBkColor() и SetBkMode().

3.3. Рабочее пространство

Для выполнения работы используется проект FONTS, подготовленный преподавателем. Проект следует установить на диск C:.

Рабочим модулем является FONTS.cpp, программирование выполняется в функции fonts().

Надписи выводятся в окно вывода, но графический вывод следует направлять в совместимый контекст CDC. Для принудительного обновления окна вывода в конце функции fonts() должна быть инструкция:

```
// принудительный вывод в окно вывода
InvalidateRect(hWndOut, 0, 1);
```

3.4. Использование стандартных шрифтов

В этой части работы используем стандартные шрифты системы.

Для вывода текста используем функцию TextOut().

Общий порядок вывода текста:

- выбрать шрифт функцией GetStockObject() в переменную hf;
- выбрать шрифт hf в контекст CDC функцией SelectObject(), результат функции запомнить в переменной exhf;
- установить цвет, режимы вывода;
- вывести текст;
- выбрать шрифт exhf в контекст CDC функцией SelectObject();
- удалить шрифт hf функцией DeleteObject().

Пример см. [1, листинг 16].

Константы системных шрифтов см. [1, приложение «Стандартные объекты Windows»].

Следует вывести за один раз все имеющиеся системные шрифты. Для этого нужно каждый раз выбирать шрифт заново. В отличие от первого выбора, последующие выборы шрифтов не должны изменять значение переменной exhf, которая хранит шрифт по умолчанию.

Выводим надписи вида «Шрифт ANSI_VAR_FONT». Не забываем уточнять длину надписи и корректировать ее при вызове функции TextOut().

В этой части следует использовать разные цвета для разных надписей, для чего перед выводом надписи задайте цвет функцией SetTextColor().

Кроме того, следует использовать функции SetBkColor() и SetBkMode(). Первая функция задает цвет фона, вторая устанавливает прозрачность.

Еще одна функция, которую нужно применить — SetTextAlign().

С ее помощью задается выравнивание текста относительно указанной точки вывода. Константы, задающие выравнивание, приведены в файле wingdi.h, в разделе /* Text Alignment Options */.

При защите этой части способы выравнивания, указанные в wingdi.h, будут контролироваться.

3.5. Выбор шрифта при помощи стандартного диалога

В этой части работы нужно показать стандартный диалог для выбора шрифта, выбрать в нем шрифт и вывести этим шрифтом надпись.

Для выбора стандартного диалога используем функцию ChooseFont().

Пример см. [1, листинг 10].

Для использования функции ChooseFont() сначала нужно сформировать структуру CHOOSEFONT в переменной cf. После выбора шрифта характеристики шрифта формируются в виде структуры LOGFONT. Для разных выбранных шрифтов эти характеристики должны быть изучены.

Должны быть выбраны минимум следующие шрифты: System, Fixedsys, Courier New, Arial, Tahoma, Times New Roman, MS Sans Serif.

Шрифт в этом случае создается с помощью функции CreateFontIndirect().

Заметим, что функция стандартного диалога ChooseFont() возвращает *ложь* в случае, если пользователь отказывается от диалога.

3.6. Создание шрифта

В этой части работы изучаем функцию CreateFont().

Параметры этой функции схожи с полями структуры LOGFONT.

При выполнении этой части работы создаем несколько различных шрифтов. Различия должны быть принципиальными и затрагивать все характеристики шрифтов.

В частности, требуется вывести надписи:

- горизонтально,
- вертикально,
- с поворотом на угол 45°,
- с повернутыми буквами горизонтально и вертикально,
- с широкими и узкими символами.

Для выполнения некоторых частных случаев понадобится установить графический режим GM_ADVANCED при помощи функции SetGraphicsMode().

В качестве гарнитуры используем один шрифт с засечками и один шрифт без засечек.

Во время защиты этой части работы будет контролироваться знание всех характеристик шрифтов.

3.7. Дополнительные вопросы

В оставшееся время изучите структуру TEXTMETRIC.

Получите характеристики стандартного шрифта Tahoma, изучите их.

4. Рисование мышью

Цели:
- изучение сообщений о действиях мышью.

Задачи:
- запись и чтение установок программы;
- управление фоном окна рисования;
- создание пера для рисования;
- обработка событий мыши;
- рисование следа мыши.

4.1. Сообщения о действиях с мышью

Операционная система посылает следующие сообщения, когда пользователь выполняет действие с мышью:

WM_LBUTTONDOWN — левая кнопка мыши нажата (опущена);

WM_LBUTTONUP — левая кнопка мыши отпущена;

WM_MOUSEMOVE — мышь изменила положение.

Есть и другие сообщения, но эти наиболее важные, так как именно они соответствуют основным действиям, выполняемым мышью.

Рисование мышью производится в событии WM_MOUSEMOVE.

Поскольку это событие возникает при любом перемещении мыши, нужен признак, указывающий на то, что пользователь удерживает левую кнопку мыши нажатой.

Для этой цели используются два других события. При нажатии левой кнопки признак нажатия устанавливается, а при отпускании очищается.

Тогда, если при возникновении события перемещения мыши признак установлен, нужно нарисовать отрезок прямой линии от предыдущего положения мыши до текущего положения.

Предыдущее положение мыши фиксируется в момент нажатия левой кнопки. После рисования текущего отрезка текущие координаты запоминаются как предыдущие.

Кроме того, есть еще один важный момент. Предположим, пользователь начал рисовать мышью и в какой-то момент увел мышь за пределы окна рисования (в другое окно, поскольку на экране все есть окно). В этом случае события мыши начнут поступать в другое окно. Чтобы события поступали в окно рисования, нужно выполнить так называемый «*захват событий мыши*».

Захват производится с помощью функции SetCapture(), единственный параметр которой — дескриптор окна, выполняющего захват. После этого все события будут направляться в то окно, которое «*захватило*» мышь.

Захват мыши выполняется при нажатии кнопки мыши.

Выход из состояния захвата выполняет функция ReleaseCapture().

Эта функция вызывается в событии отпускания кнопки мыши.

4.2. Рабочее пространство

Для выполнения работы используется проект Paint, подготовленный преподавателем. Проект следует установить на диск C:.

Данный проект используется для этой и следующей работы. Цель проекта — создание приложения для рисования простых графических примитивов.

Для сохранения установок программы используется файл инициализации типа ini. Для работы с этим файлом в проекте предусмотрен специальный класс Settings.

В следующей работе будут применяться различные установки для рисования, такие, как стиль пера, используемый инструмент и другие. Поэтому в этой работе нужно подготовить проект к чтению установок.

Для рисования используется совместимый контекст hCDC. Для однократной отмены действия используется дополнительный совместимый контекст hCD2. В проекте создается окно для рисования и упомянутые совместимые контексты.

В модуле drawing.h объявлены следующие глобальные переменные:

MX0, MY0 — координаты начала действия;

MX1, MY1 — координаты начала отрезка;

MX2, MY2 — координаты конца отрезка.

Координаты мыши передаются вместе с сообщением и извлекаются из параметра lParam с помощью функции GetMouse(), определенной в модуле Paint.cpp.

В модуле drawing.h расположены функции для рисования:

```
// рисует линию
void DrawMouse(HDC hdc, int & X1, int & Y1, int X2, int Y2) {
}

// рисует прямую
void DrawLine(int X1, int Y1, int X2, int Y2) {
}

// рисует прямоугольник
void DrawRect(int X1, int Y1, int X2, int Y2) {
}

// рисует овал
void DrawEllipse(int X1, int Y1, int X2, int Y2) {
}

// выбирает инструмент
void SetTool(TOOLS tool) {
}

// создает новое перо плана
void CreatePenFore() {
}
```

Эти функции должны быть реализованы в рамках этой и следующей работ. Функция CreatePen() предназначена для создания пера для рисования, а функция SetTool() устанавливает текущий инструмент рисования.

Для пера определены переменные:

PenFore — само перо,

pen_style — тип пера,

line_style — тип линии,

pen_size — размер пера,

half_pen — половина ширины пера,

PenBrush — кисть геометрического пера,

end_cap — тип концевой точки,

g_style — тип кисти,

g_hatch — тип штриховки.

Кроме того, нужно создать палитру для выбора цветов плана и фона.

В модуле drawing.h для этой цели определены цвета 16-ти цветной палитры C16, переменные для цвета плана pen_color и фона back_color, переменные для дескрипторов окон палитры hWndPal.

4.3. Файл инициализации

Файл типа ini (файл инициализации) предназначен для локального хранения настроек программ. Запись и чтение этого файла производится специальными функциями. Функция WritePrivateProfileString() записывает в файл строку, функция GetPrivateProfileString() читает строку из файла.

Записи в файле имеют вид "ключ=значение". Поэтому при записи или чтении нужно указывать ключ. Для облегчения операций с файлом инициализации в проекте есть класс settings.

Запись и чтение строкового параметра выполняют функции класса

PutSetting()

GetSetting()

Запись и чтение целочисленного параметра выполняют функции

PutSettingLong()

GetSettingLong()

Из этого следует, что вещественное значение записывается как строковое с преобразованиями, выполняемыми вне класса.

Модуль Paint.cpp, функция PutSettings().

Записываем текущий инструмент рисования в файл инициализации:

```
// записывает параметры в файл инициализации
void PutSettings() {
    // создаем файл инициализации
    Settings s;
    // задаем путь к файлу инициализации
    s.PutPath(szIniPath);
    // текущий инструмент
    s.PutSettingLong((int)curr_tool, "curr_tool");
}
```

Заметим, что записываемый параметр нужно привести к типу int.

Запускаем проект и закрываем его.

В каталоге C:\Paint\Debug должен появиться файл Paint.ini, содержащий запись инструмента:

```
[PAINT]
curr_tool=0
```

Заметим также, что записи могут группироваться в разделы. В нашем примере раздел PAINT задан неизменным.

Переходим в функцию GetSettings(), и считываем инструмент:

```
// читает параметры из файла инициализации
void GetSettings() {
    // создаем файл инициализации
    Settings s;
    // задаем путь к файлу инициализации
    s.PutPath(szIniPath);
    // текущий инструмент
    curr_tool = (TOOLS)s.GetSettingLong("curr_tool", TOOL_MOUSE);
}
```

Здесь также нужно выполнять приведение к типу переменной.

Кроме того, нужно указывать значение по умолчанию, то, которое получит считываемый параметр, если в файле инициализации еще нет соответствующей записи. В данном случае это TOOL_MOUSE.

Теперь при смене инструмента он запоминается и при последующем старте программы восстанавливается.

Аналогичным образом нужно записать и прочитать следующие параметры программы:

pen_color — цвет плана;

back_color — цвет фона;

pen_size — размер пера;

pen_style — тип пера;

line_style — стиль линии пера;

end_cap — тип концевой точки;

g_style — тип кисти пера;

g_hatch — тип штриховки пера;

Значения по умолчанию — константы, определенные в файле wingdi.h.

Можно использовать значения, которыми переменные инициализируются при объявлении.

Естественно, при чтении параметров должны происходить какие-то изменения в программе. Например, после считывания параметров пера должно создаваться новое перо, при необходимости кисть пера. После считывания инструмента он должен выбираться. Записанные цвета плана и фона должны отображаться. В настоящий момент не все из этих действий можно выполнить.

В конце функции GetSettings() вызываем функцию SetTool() для установки текущего инструмента и функцию CreatePenFore() для создания пера.

Функция SetTool() присваивает переменной curr_tool значение tool.

Функция CreatePenFore() создает перо в соответствии с текущими установками, после чего корректирует флажки в меню.

Сейчас в этой функции можно создать только косметическое перо.

Прежде всего функция удаляет существующее перо с помощью функции DeleteObject(). Делать это нужно только если переменная пера PenFore не равна лжи.

Далее нужно проверить, чему равно значение pen_style.

Если значение равно PS_COSMETIC, создается косметическое перо, иначе создается геометрическое перо.

Создаем косметическое перо с помощью функции CreatePen().

Первый параметр формируется как сочетание константы PS_COSMETIC и переменной line_style, второй параметр равен pen_size, третий pen_color.

Возвращаемое значение присваивается переменной PenFore.

Пока в этой функции всё.

4.4. Палитра цветов

Создадим палитру цветов в виде шестнадцати цветных прямоугольников, закрашенных цветами стандартной 16-ти цветной палитры.

Для этого нужно выполнить следующие действия:

- определить оконную процедуру для окон палитры одну на все окна;
- описать функцию создания класса окна;
- создать окна палитры с помощью цикла;
- создать окна выбранных цветов;
- обработать сообщения окон палитры.

Кроме того, создадим также два окна, в которых будут отображаться текущие выбранные цвета плана и фона.

Необходимые для этих действий переменные определены в заголовочных файлах Paint.h и drawing.h.

Сначала описываем прототипы функций в модуле Paint.h:

```
// класс окна графического вывода
ATOM MyRegisterClassGDE(HINSTANCE);
// класс окна палитры
ATOM MyRegisterClassPal(HINSTANCE);
// создает главное окно
BOOL InitInstance(HINSTANCE, int);
// оконная процедура главного окна
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
// оконная процедура окна палитры
LRESULT CALLBACK PalProc(HWND, UINT, WPARAM, LPARAM);
```

Переходим в модуль Paint.cpp и описываем оконную процедуру окон палитры после оконной процедуры окна графического вывода:

```
// оконная процедура окна палитры
LRESULT CALLBACK PalProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
    // номер окна палитры
    int index = GetWindowLong(hWnd, GWL_ID);
    switch (message) {
    case WM_LBUTTONDOWN: // нажата левая кнопка мыши

        return 0;
    case WM_RBUTTONDOWN: // нажата правая кнопка мыши

        return 0;
    case WM_ERASEBKGND:  // очистка окна

        return 0;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Переходим в модуль Paint.h и описываем функцию создания класса окна палитры после функции MyRegisterClassGDE():

```
// класс окна палитры
ATOM MyRegisterClassPal(HINSTANCE hInstance) {
    WNDCLASSEX wcex;
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style          = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
    wcex.lpfnWndProc    = (WNDPROC)PalProc;
    wcex.cbClsExtra     = 0;
    wcex.cbWndExtra     = 0;
    wcex.hInstance      = hInstance;
    wcex.hIcon          = 0;
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = 0;
    wcex.lpszClassName  = szPaletteClass;
    wcex.hIconSm        = 0;
    return RegisterClassEx(&wcex);
}
```

Эту функцию можно легко получить копированием и редактированием функции MyRegisterClassGDE().

Функцию MyRegisterClassPal() необходимо вызвать в главной функции там, где вызываются функции создания классов главного окна и окна графического вывода.

Будем теперь создавать окна палитры в функции InitInstance(), которая также расположена в модуле Paint.h.

Палитру будем создавать правее окна графического вывода в виде двух вертикальных полос по 8 квадратов размером PAL_SIZE.

Для простоты сделаем это в двух циклах. Переменная ccx задает положение окна слева, переменная ccy — сверху.

Нужно сделать отступ вверху для двух окон текущих цветов.

Переходим в конец функции InitInstance():

```
// окна палитры
int ccx = UW_LEFT;
int ccy = UW_TOP + PAL_SIZE + PAL_SIZE;
for (int N = 0; N < 8; N++) {
    hWndPal[i] = CreateWindow(szPaletteClass, 0,
        WS_CHILD | WS_VISIBLE | WS_BORDER,
        ccx, ccy, PAL_SIZE, PAL_SIZE,
        hWndMain, (HMENU)N, hInst, 0);
    ccy += PAL_SIZE - 1;
}
ccx = UW_LEFT + PAL_SIZE - 1;
ccy = UW_TOP + PAL_SIZE + PAL_SIZE;
for (int N = 8; N < 16; N++) {
    hWndPal[i] = CreateWindow(szPaletteClass, 0,
        WS_CHILD | WS_VISIBLE | WS_BORDER,
        ccx, ccy, PAL_SIZE, PAL_SIZE,
        hWndMain, (HMENU)N, hInst, 0);
    ccy += PAL_SIZE - 1;
}
```

Первый цикл создает окна слева, второй — окна справа (рисунок 1). Каждому окну присваивается номер, равный значению переменной `N`.

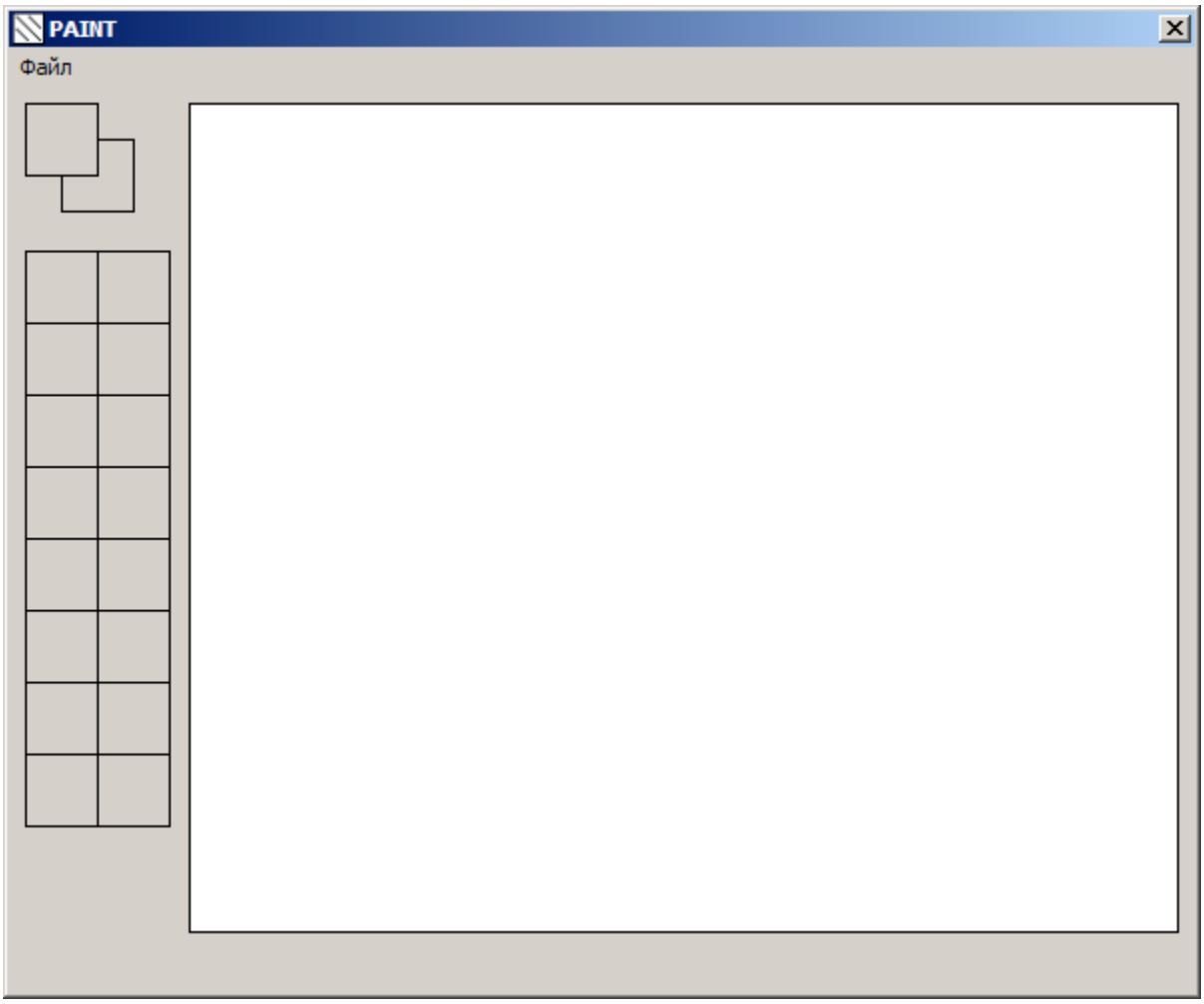

Рисунок 1 - Окна палитры

Окна для выбранных цветов поместим в специальное окно, поэтому далее создаем это окно-контейнер:

```
// окно для окон выбранных цветов
hWndColors = CreateWindow("STATIC", 0,
    WS_CHILD | WS_VISIBLE,
    UW_LEFT, UW_TOP, PAL_SIZE + PAL_SIZE, PAL_SIZE + PAL_SIZE,
    hWndMain, 0, hInst, 0);
```

Далее создаем два окна для цвета фона и плана:

```
// окно цвета фона
hWndBack = CreateWindow(szPaletteClass, 0,
    WS_CHILD | WS_VISIBLE | WS_BORDER,
    PAL_SIZE / 2, PAL_SIZE / 2, PAL_SIZE, PAL_SIZE,
    hWndColors, (HMENU)BACK_COLOR, hInst, 0);
// окно цвета плана
hWndPen = CreateWindow(szPaletteClass, 0,
    WS_CHILD | WS_VISIBLE | WS_BORDER,
    0, 0, PAL_SIZE, PAL_SIZE,
    hWndColors, (HMENU)PEN_COLOR, hInst, 0);
```

Порядок создания важен, так как одно окно должно перекрывать другое. Окна создаются как окна палитры для того, чтобы можно было одинаковым образом управлять перерисовкой окон. Цвета этих окон в массиве цветов C16 заданы константами PEN_COLOR и BACK_COLOR.

Сейчас окна палитры и цветов плана и фона не имеют цвета и наша следующая задача — закрасить их.

Для этой цели в модуле drawing.h определена функция PaintWindow():

```
// закрашивает окно
void PaintWindow(HWND hWnd, COLORREF color) {
}
```

Сейчас нужно наполнить эту функцию содержанием.

Сначала с помощью функции GetDC() получим контекст окна в переменную hdc типа HDC. Затем выберем в контекст окна кисть DC_BRUSH с помощью функции SelectObject(). Кисть создается с помощью функции выбора стандартного объекта:

```
SelectObject(hdc, GetStockObject(DC_BRUSH));
```

Далее нужно установить цвет этой кисти равным параметру color:

```
SetDCBrushColor(hdc, color);
```

Далее нужно определить прямоугольник rc клиентской части окна с помощью функции GetClientRect(), после чего с помощью функции PatBlt() закрашиваем вычисленный прямоугольник, указывая контекст hdc, поля прямоугольника rc и операцию PATCOPY:

```
PatBlt(hdc, rc.left, rc.top, rc.right, rc.bottom, PATCOPY);
```

В конце функции освобождаем контекст функцией ReleaseDC().

Теперь с помощью этой функции можно закрашивать окна палитры.

Найдем оконную процедуру окна палитры PalProc и в событии очистки окна вызываем функцию PaintWindow(), передавая дескриптор hWnd и цвет из массива C16[index], index — номер окна.

После этого окна палитры должны получить свой цвет.

Чтобы отобразить выбранные цвета плана и фона, нужно определить функции модуля drawing.h:

```
// устанавливает цвет плана
void SetColorFore(COLORREF color) {
}

// устанавливает цвет фона
void SetColorBack(COLORREF color) {
}
```

Функции похожи. Сначала нужно запомнить передаваемый параметр color в переменной pen_color или back_color. Затем установим цвет color в массиве C16, используя индексы PEN_COLOR и BACK_COLOR. После этого можно закрасить окна hWndPen и hWndBack цветом color функцией PaintWindow().

Поскольку при закраске этих окон мы использовали всю клиентскую часть окон, получим нежелательный эффект — при закраске окна цвета фона окно рисуется поверх окна цвета плана. Чтобы устранить этот эффект, в конце этих функций вызовем принудительную перерисовку окна контейнера с помощью функции InvalidateRect(), передавая этой функции дескриптор окна hWndColors и другие параметры, равные нулю.

Теперь мы готовы показать выбранные цвета.

Во-первых, эти две функции вызываются в GetSettings(), после считывания параметров pen_color и back_color из файла инициализации.

Во-вторых, эти функции вызываются в оконной процедуре палитры, при получении сообщений нажатия кнопок мыши. Поскольку только первые 16 окон палитры должны вызывать установку цветов, обработка сообщений начинается с отсечения по значению index, например:

```
switch (message) {
case WM_LBUTTONDOWN: // нажата правая кнопка мыши
    if (index > 15) return 0;
    // новый цвет плана
    SetColorFore(C16[index]);
    // создаем новое перо плана
    . . .
    return 0;
case WM_RBUTTONDOWN: // нажата левая кнопка мыши
```

При изменении цвета следует заново создать перо, поэтому при обработке сообщения нужно вызвать функцию создания нового пера и передать ей выбранный цвет index.

4.5. Рисование мышью

Выбор цвета реализован, перо плана создается, можно приступить к рисованию мышью.

Рисовать след мыши будет функция DrawMouse() модуля drawing.h, поэтому переходим к этой функции.

Рисование отдельного отрезка прямой линии выполняется так.

Сначала устанавливается текущая точка функцией MoveToEx(). Третий параметр этой функции возвращает бывшую текущую точку. Если эта точка не требуется, третий параметр можно принять равным нулю.

После этого вызывается функция LineTo(), которая рисует отрезок.

Чтобы нарисовать несколько непрерывных отрезков, нужно несколько раз вызвать функцию LineTo(), потому что она меняет текущую точку.

Нам как раз нужно нарисовать несколько непрерывных отрезков, поэтому функцию MoveToEx() следует вызвать в событии нажатия кнопки, а функцию LineTo() следует вызывать в событии перемещения мыши.

Полный перечень действий функции DrawMouse() следующий:

- нарисовать отрезок до точки (X2, Y2);
- создать прямоугольник RECT в переменной rc, вычислить область обновления, занимаемую отрезком, и задать поля rc;
- вызвать перерисовку изображения с помощью InvalidateRect(), указывая дескриптор окна hWndGDE и область обновления rc;
- запомнить координаты (X2, Y2) как (X1, Y1).

Можно не вычислять область обновления rc, но тогда придется перерисовывать все окно, что вызовет помехи при рисовании.

Самым сложным здесь является вычисление области обновления.

Следует учитывать, что мышь может двигаться в произвольном направлении, а функция вычисления координат мыши GetMouse() может также возвращать значения –1 при исключительных ситуациях. Кроме того, нужно учитывать также ширину пера с помощью переменной half_pen, хотя на самом деле не будет большой разницы, если учитывать полную ширину пера. В любом случае нужно прибавить пиксель к вычисленной области, чтобы случайно не отрезать часть вывода.

После этого в событии WM_LBUTTONDOWN окна графического вывода:

- захватываем события мыши,
- выбираем перо PenFore в контекст hCDC,
- устанавливаем признак mouse_down равным WM_LBUTTONDOWN,
- получаем координаты MX1, MY1 с помощью GetMouse(),
- устанавливаем текущую точку с помощью MoveToEx().

В событии WM_LBUTTONUP:

- очищаем mouse_down (в ноль),
- освобождаем захват событий мыши.

В событии WM_MOUSEMOVE:

- проверяем mouse_down, если не ноль, то возвращаем 0,

- получаем координаты MX2, MY2 с помощью GetMouse(),

- рисуем след мыши с помощью DrawMouse().

Чтобы рисовать правой кнопкой мыши (цветом фона), нужно определить функцию создания пера CreatePenBack(), и продублировать действия в событиях WM_RBUTTONDOWN и WM_RBUTTONUP окна графического вывода.

Общий вид оконной процедуры этого окна:

```
// оконная процедура окна графического вывода
LRESULT CALLBACK GDEProc(HWND hWnd, UINT message, WPARAM . . .
    PAINTSTRUCT ps;
    HDC hdc;
    int L, T, W, H;
    switch (message) {
    case WM_LBUTTONUP:    // отпущена левая кнопка мыши
    case WM_RBUTTONUP:    // отпущена правая кнопка мыши
        mouse_down = 0;
        ReleaseCapture();
        return 0;
    case WM_LBUTTONDOWN: // нажата левая кнопка мыши
        . . .
        return 0;
    case WM_RBUTTONDOWN: // нажата правая кнопка мыши
        . . .
        return 0;
    case WM_MOUSEMOVE:    // движение мыши
        if (!mouse_down) return 0;
        . . .
        return 0;
    case WM_PAINT:
        . . .
        return 0;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

4.6. Выравнивание линий

Чтобы рисовать прямые линии от следа мыши, нужно фиксировать координату X или Y, если во время движения клавиша Shift нажата. Чтобы понять, о чем идет речь, откройте стандартное приложение *Paint* и попробуйте рисовать мышью, удерживая Shift.

Для этой цели в проекте определяется состояние клавиши Shift с помощью переменной shift_on.

В обработке WM_MOUSEMOVE после GetMouse() нужно вызвать функцию ShiftMouse(), которая скорректирует координаты, если shift_on не ноль.

Реализация ShiftMouse() отдается на откуп обучающимся.

В ней нужно вычислить, в каком направлении идет движение, и сравнять либо координаты X, либо координаты Y.

5. Основные графические примитивы

Цели:
- изучение графических примитивов Windows GDI.
Задачи:
- проектирование меню;
- установка режимов рисования;
- создание перьев;
- рисование прямых, прямоугольников, овалов;
- рисование других примитивов.

5.1. Рабочее пространство

Работа является продолжением работы «Рисование мышью», которая должна быть полностью выполнена. Рабочими модулями являются Paint.cpp и drawing.h.

5.2. Проектирование меню

Для продолжения редактирования нам нужно обеспечить выбор инструмента и параметров рисования, влияющих на характеристики перьев.

Поэтому сейчас нужно открыть ресурсы приложения и создать меню.

Первый пункт меню «Файл» содержит только пункт «Закрыть».

Нужно создать еще три пункта.

Пункт	*Подпункты*	*Подпункты*	*ID*
Редактирование	Очистить		ID_CLEAR
	-		
	Отменить		ID_UNDO
Инструмент	Линия		ID_MOUSE
	Прямая		ID_LINE
	Прямоугольник		ID_RECT
	Овал		ID_ELLIPSE
Перо	Размер	1	ID_SIZE_1
		3	ID_SIZE_3
		5	ID_SIZE_5
	Косметическое		ID_COSMETIC
	Геометрическое		ID_GEOMETRIC
	Тип линии	Сплошная	ID_PS_SOLID
		Штриховая	ID_PS_DASH
		Пунктирная	ID_PS_DOT
	Концевая точка	Круглая	ID_CAP_ROUND
		Прямоугольная	ID_CAP_SQUARE
		Плоская	ID_CAP_FLAT
	Штриховка	Сплошная	ID_HATCH_SOLID
		Клетка	ID_HATCH_CROSS
		Диагональная клетка	ID_HATCH_DIAGCROSS

Результат показан на рисунке 2.

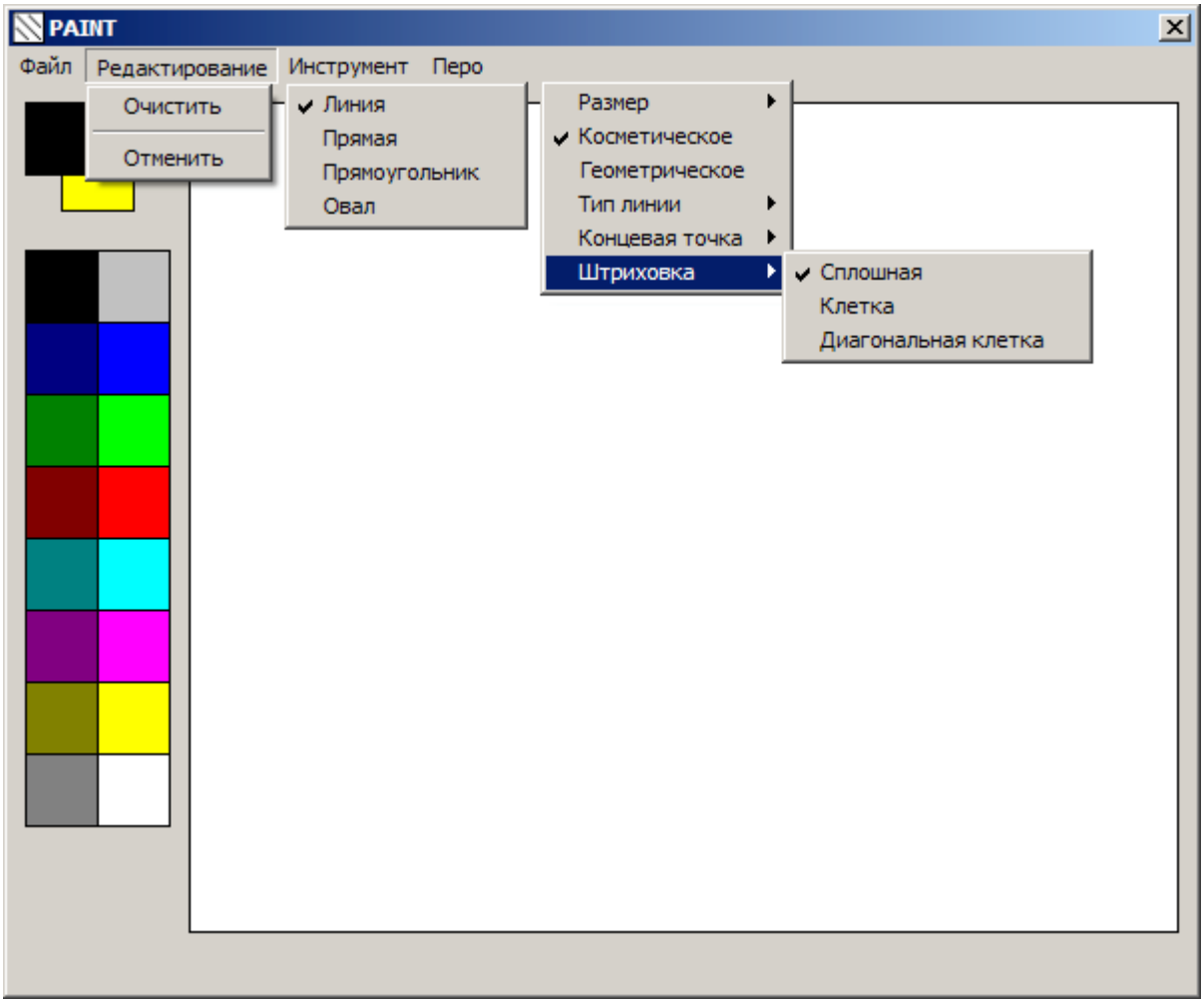

Рисунок 2 - Пункты меню

После создания пунктов меню проект нужно закрыть. Далее заходим в папку C:\Paint\Paint и вручную редактируем файл resource.h. Цель — установить нужные числовые значения идентификаторов пунктов меню.

ID	Значение
ID_SIZE_1	30001
ID_SIZE_3	30003
ID_SIZE_5	30005
ID_PS_SOLID	40000
ID_PS_DASH	40001
ID_PS_DOT	40002
ID_CAP_ROUND	50000
ID_CAP_SQUARE	50256
ID_CAP_FLAT	50512
ID_HATCH_SOLID	60000
ID_HATCH_CROSS	60204
ID_HATCH_DIAGCROSS	60205

Смысл задания таких значений следующий. Для примера рассмотрим идентификаторы концевых точек. В файле wingdi.h определены следующие константы для концевых точек:

```
#define PS_ENDCAP_ROUND         0x00000000
#define PS_ENDCAP_SQUARE        0x00000100
#define PS_ENDCAP_FLAT          0x00000200
```

То есть, десятичные значения равны 0, 256 и 512 соответственно.

Прибавляя к ним 60000, получим константы для файла resource.h.

Теперь переходим в модуль Paint.cpp, оконная процедура главного окна WndProc(), обрабатываем сообщения.

Сообщение ID_COSMETIC:

```
// оконная процедура главного окна
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, . . .
    int wmId, wmEvent;
    switch (message) {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // разбор команды (меню)
        switch (wmId) {
        case IDM_EXIT:
            DestroyWindow(hWndGDE);
            DestroyWindow(hWnd);
            return 0;
        case ID_COSMETIC:
            SetPenStyle(PS_COSMETIC);
            CreatePenFore();
            CreatePenBack();
            return 0;
    . . .
}
```

Здесь мы вызываем функцию, которая устанавливает тип пера и заново создаем перья для плана и фона. Далее все сообщения обрабатываем аналогичным образом. Сообщение ID_GEOMETRIC:

```
        case ID_GEOMETRIC:
            SetPenStyle(PS_GEOMETRIC);
            CreatePenFore();
            CreatePenBack();
            return 0;
```

Следующие сообщения — размер пера:

```
        case ID_SIZE_1:
        case ID_SIZE_3:
        case ID_SIZE_5:
            SetPenSize(wmId - 30000);
            CreatePenFore();
            CreatePenBack();
            return 0;
```

Здесь мы вычитаем 30000, чтобы получить размер пера.

Далее тип линии:

```
        case ID_PS_SOLID:
        case ID_PS_DASH:
        case ID_PS_DOT:
            SetLineStyle(wmId - 40000);
            CreatePenFore();
            CreatePenBack();
            return 0;
```

Здесь все то же самое, только вычитаем 40000.

Далее тип концевой точки:

```
        case ID_CAP_ROUND:
        case ID_CAP_SQUARE:
        case ID_CAP_FLAT:
            SetEndCap(wmId - 50000);
            CreatePenFore();
            CreatePenBack();
            return 0;
```

Далее штриховка пера. Она формируется по разному в зависимости от наличия или отсутствия штриховки. Если штриховки нет (сплошная):

```
        case ID_HATCH_SOLID:
            SetPenHatch(BS_SOLID, g_hatch);
            CreatePenFore();
            CreatePenBack();
            return 0;
```

Если штриховка есть:

```
        case ID_HATCH_CROSS:
        case ID_HATCH_DIAGCROSS:
            SetPenHatch(BS_HATCHED, wmId % 100);
            CreatePenFore();
            CreatePenBack();
            return 0;
```

Далее выбор инструмента:

```
        case ID_MOUSE:
        case ID_LINE:
        case ID_RECT:
        case ID_ELLIPSE:
            SetTool((TOOLS)wmId);
            return 0;
```

Здесь нужно сделать так, чтобы константы инструмента совпадали с константами файла resource.h. Поэтому переходим в модуль drawing.h:

```
enum TOOLS {
    TOOL_POINTER = 0,
    TOOL_MOUSE = ID_MOUSE,
    TOOL_LINE = ID_LINE,
    TOOL_RECT = ID_RECT,
    TOOL_ELLIPSE = ID_ELLIPSE
};
```

Возвращаемся в модуль Paint.cpp, функции редактирования:

```
        case ID_CLEAR:
            ImageClear(back_color);
            return 0;
        case ID_UNDO:
            ImageRestore();
            InvalidateRect(hWndGDE, 0, 0);
            return 0;
```

Заметим, что при очистке поле заполняется выбранным цветом фона.

Теперь нужно определить все вызванные функции. Смысл этих функций — установить флажки в меню. Переходим в модуль drawing.h.

Обрабатываем стиль пера:

```
// выбирает стиль пера
void SetPenStyle(int style) {
    CheckMenuItem(hMenu, ID_COSMETIC, MF_UNCHECKED | MF_BYCOMMAND);
    CheckMenuItem(hMenu, ID_GEOMETRIC, MF_UNCHECKED | MF_BYCOMMAND);
    if (style == PS_GEOMETRIC) {
        CheckMenuItem(hMenu, ID_GEOMETRIC, MF_CHECKED | MF_BYCOMMAND);
        pen_style = PS_GEOMETRIC;
    } else {
        CheckMenuItem(hMenu, ID_COSMETIC, MF_CHECKED | MF_BYCOMMAND);
        pen_style = PS_COSMETIC;
    }
}
```

Здесь функция CheckMenuItem() устанавливает или снимает флажок в меню. Если задан параметр MF_UNCHECKED, флажок снимается, а если задан параметр MF_CHECKED, флажок устанавливается. Сначала мы снимаем оба флажка, а затем один из них устанавливаем. Убедитесь, что флажки появляются в соответствующих пунктах меню.

Здесь сначала вычисляется текущее значение идентификатора путем прибавления 40000 и флажок снимается. После этого значение line_style устанавливается и используется для установки флажка.

Переходим к размеру пера:

```
// выбирает размер пера
void SetPenSize(int size) {
    CheckMenuItem(hMenu, pen_size + 30000, MF_UNCHECKED | MF_BYCOMMAND
    pen_size = size;
    CheckMenuItem(hMenu, pen_size + 30000, MF_CHECKED | MF_BYCOMMAND);
}
```

Переходим к типу линии:

```
// выбирает стиль линии
void SetLineStyle(int style) {
    CheckMenuItem(hMenu, line_style + 40000, MF_UNCHECKED | MF_BYCOMMAN
    line_style = style;
    CheckMenuItem(hMenu, line_style + 40000, MF_CHECKED | BYCOMMAND);
}
```

Концевые точки:

```
// выбирает концевую точку
```

```
void SetEndCap(int cap) {
    CheckMenuItem(hMenu, end_cap + 50000, MF_UNCHECKED | MF_BYCOMMAND);
    end_cap = cap;
    CheckMenuItem(hMenu, end_cap + 50000, MF_CHECKED | MF_BYCOMMAND);
}
```

Штриховка пера обрабатывается сложнее из-за того, что если штриховки нет, то задаются одни параметры, а если есть, другие:

```
// выбирает штриховку пера
void SetPenHatch(int style, int hatch) {
    int h = (g_style == BS_SOLID ? 0 : g_hatch) + g_style * 100 + 60000;
    CheckMenuItem(hMenu, h, MF_UNCHECKED | MF_BYCOMMAND);
    g_style = style;
    g_hatch = hatch;
    h = (g_style == BS_SOLID ? 0 : g_hatch) + g_style * 100 + 60000;
    CheckMenuItem(hMenu, h, MF_CHECKED | MF_BYCOMMAND);
}
```

Теперь все эти функции должны быть вызваны во время чтения параметров из файла инициализации. Функция GetSettings(), модуль Paint.cpp.

Функции вызываются в конце GetSettings(), после чего создаются перья, устанавливается инструмент. Кроме того, здесь же нужно вызвать функцию очистки ImageClear() и задать белый цвет.

5.3. Очистка поля

Модуль drawing.h, функция ImageClear().

Очистка поля в значительной степени напоминает закрашивание окна и в принципе, можно, наверное, использовать функцию PaintWindow(), однако есть одно отличие. Очищается не устройство вывода, а совместимый контекст, после чего вызывается функция InvalidateRect(), которая посылает сообщение WM_PAINT.

Сначала выбираем в контекст hCDC стандартную кисть DC_BRUSH.

Затем задается цвет кисти DC_BRUSH функцией SetDCBrushColor().

Затем с помощью PatBlt() закрашивается поле совместимого контекста, при этом используется размер поля GDE_SIZEX и GDE_SIZEY.

В конце функции вызывается InvalidateRect() для hWndGDE. Другие параметры этой функции равны нулю.

5.4. Рисование мышью

У нас осталось нереализованным геометрическое перо. Оно создается в функциях CreatePenFore() и CreatePenBack(). В них в зависимости от значения переменной pen_style создается либо косметическое перо, либо геометрическое. Создание косметического пера мы описали. Перейдем к созданию геометрического пера.

Сначала задаются параметры кисти пера PenBrush:

```
    } else {
        // геометрическое перо
```

```
    PenBrush.lbColor = fore_color;
    PenBrush.lbHatch = g_hatch;
    PenBrush.lbStyle = g_style;
}
```

Здесь показана функция CreatePenFore(). Для функции CreatePenBack() нужно использовать другой цвет, цвет фона.

Затем создается геометрическое перо функцией ExtCreatePen():

```
PenFore = ExtCreatePen(PS_GEOMETRIC | end_cap | line_style,
    pen_size, &PenBrush, 0, 0);
```

Для функции CreatePenBack() значение нужно присвоить PenBack.

Теперь можно попробовать рисовать геометрическим пером.

Интересно, заметите ли вы, что геометрическое перо рисует неправильно. Это потому, что геометрическое перо использует установленный цвет фона, равно как и многие графические примитивы, например, примитив, который рисует прямоугольник.

Поэтому сейчас нужно установить правильный цвет фона с помощью функции SetBkColor(). Вызывать эту функцию нужно в момент нажатия кнопки мыши. Если нажата левая кнопка, устанавливается цвет фона, а если нажата правая кнопка, устанавливается цвет плана. Модуль Paint.cpp, функция GDEProc(). Цвет устанавливается для совместимого контекста hCDC.

Примерный результат показан на рисунке 3.

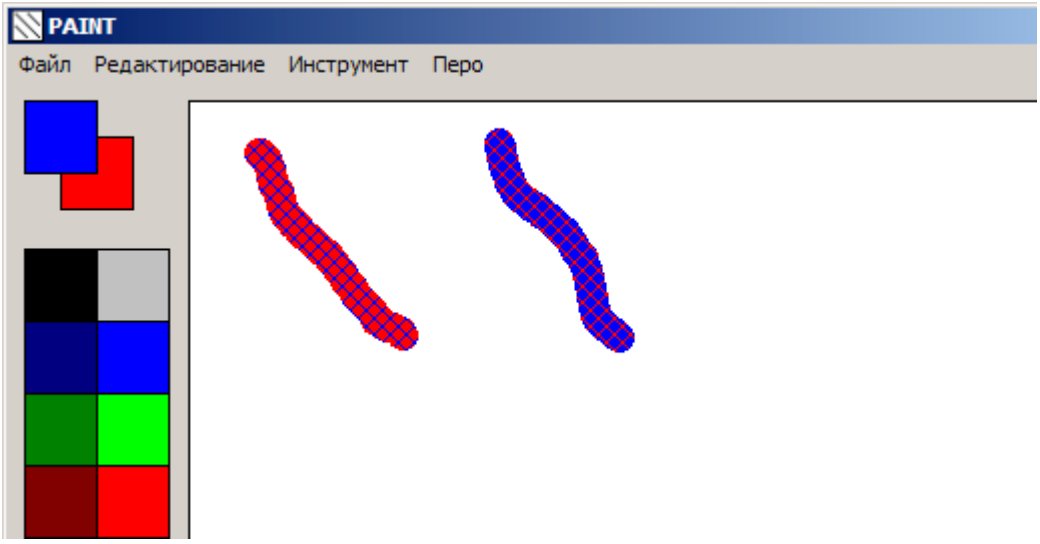

Рисунок 3 - Геометрическое перо

Левая линия получена левой кнопкой, а правая — правой.

Видно, что цвета этих линий противоположны друг другу.

Чтобы было лучше видно здесь размер пера равен 15.

5.5. Отмена действия

Отмена одного действия реализуется легко и она есть следствие построения приложения, то есть как бесплатное приложение.

Перед рисованием примитива нужно вызвать функцию ImageSave().

Она скопирует текущее состояние совместимого контекста в дополнительный совместимый контекст (одна картинка копируется в другую).

Когда вызывается отмена, функция ImageRestore() выполняет обратное действие, после чего вызывается InvalidateRect(), и совместимый контекст копируется в окно графического вывода.

Заметим, что изображение сохраняется для любой кнопки.

5.6. Рисование графических примитивов

Вам остается реализовать рисование линий, прямоугольников, овалов.

Графические функции, которые их рисуют, приведены в пособии.

Прежде нужно сделать разбор инструмента с помощью, например, оператора switch(curr_tool) {}. Делать это нужно в обработке сообщения движения мыши. Затем, в зависимости от инструмента, вызывается функция, которая собственно рисует примитив.

Для примера, как рисуется линия в DrawLine():

- восстановить изображение с помощью ImageRestore();
- установить текущую точку с помощью MoveToEx();
- нарисовать линию с помощью LineTo();
- вызывать WM_PAINT с помощью InvalidateRect().

Вообще говоря, для повышения производительности приложения при вызове события WM_PAINT нужно вычислить область обновления, как это сделано в DrawMouse(). Но для простоты это здесь можно пропустить.

5.7. Дополнительные вопросы

В описании не сказано, как установить флажки для выбранного инструмента. Предполагается, что вы сами сделаете это в SetTool().

Если есть время и желание, можно ввести примитив под названием «полилиния». Рисуется он так: сначала рисуется один отрезок, после чего сразу рисуется следующий, от точки конца предыдущего отрезка и так далее до того момента, как надоест. Получается серия отрезков. Здесь важно придумать, как обозначить конец действия. Для примера можно подсмотреть в других приложениях, как это делается. Кроме того, часто такой инструмент используется для рисования замкнутых фигур, и в этом случае полилиния завершается, когда последняя точка текущего отрезка примерно совпадает с начальной точкой первого отрезка и делается замыкание.

6. Растровые алгоритмы

Цели:
- изучение алгоритма Брезенхейма;
- изучение алгоритма Ву.

Задачи:
- реализация алгоритма Брезенхейма;
- модернизация алгоритма Брезенхейма алгоритмом Ву.

6.1. Алгоритмы

Алгоритм Брезенхейма был разработан для рисования прямых линий графопостроителем, но нашел широкое применение в компьютерной графике не только для рисования прямых, но и кривых линий. Особенностью алгоритма является использование только целочисленных вычислений.

Вам нужно изучить алгоритм, изложенный в [3], и реализовать его для рисования прямых линий.

Алгоритм Ву разработан для рисования сглаженных линий и является модернизацией алгоритма Брезенхейма. Вам нужно изучить его, используя сведения, приведенные в сети Интернет, и реализовать его для рисования прямых линий.

6.2. Рабочее пространство

Данная работа является продолжением предыдущей, и выполняется в проекте Paint. Алгоритм Брезенхейма реализуется в функции BresenLine(), а алгоритм Ву реализуется в функции SmoothLine().

Есть два варианта вызова этих функций.

В первом случае алгоритмы вызываются специальными командами меню, которые нужно добавить в меню. Идентификаторы пунктов меню в этом случае ID_BRESEN и ID_SMOOTH, идентификаторы перечисления инструментов TOOLS соответственно TOOL_BRESEN и TOOL_SMOOTH.

Во втором случае функции вызываются взамен DrawLine().

6.3. Дополнительные вопросы

В качестве дополнительного упражнения реализуется алгоритм закраски замкнутой области, изложенный в [3].

Функция закраски FillArea().

Для закраски создайте пункт меню.

Литература

1. Пономарев В.В. Компьютерная графика. Учебное пособие. Часть 1. Вводный курс. Редакция 2. Озерск: ОТИ МИФИ, 2006, 122 с.

2. Пономарев Вл. Microsoft Visual C++ 6.0. Приложение Windows с нуля. Учебное пособие. Озерск: ОТИ МИФИ, 2006. — 59 с., ил.

3. Пономарев В.В. Машинная графика. Методические пособие по дисциплине «Компьютерная графика». Часть 2. Озерск: ОТИ МИФИ, 2005. — 70 с. ил.: 84.