МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Белгородский государственный технологический университет
им. В.Г. Шухова

А. В. Глухоедов

КОМПЬЮТЕРНАЯ ГЕОМЕТРИЯ И ГРАФИКА

Лабораторный практикум

Утверждено ученым советом университета в качестве учебного пособия для студентов направления бакалавриата 09.03.02 – Информационные системы и технологии

Белгород
2015

УДК004.92 (07)
ББК 32.973.26-018.2я7
Г55

Рецензенты

кандидат технических наук, доцент Белгородского государственного технологического университета им. В.Г. Шухова *Д.Н. Старченко*

Глухоедов, А.В.
Г55 Компьютерная геометрия и графика: лабораторный практикум: учебное пособие / А.В. Глухоедов. – Белгород: Изд-во БГТУ, 2015. – 184 с.

В лабораторный практикум включены теоретические сведения и рекомендации к выполнению лабораторных работ по дисциплине «Компьютерной геометрия и графика», а также задания для выполнения данных работ.

Лабораторный практикум предназначен для студентов направления бакалавриата 09.03.02 – Информационные системы и технологии.

Данное издание публикуется в авторской редакции.

УДК 004.92 (07)
ББК 32.973.26-018.2я7

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1. ВВЕДЕНИЕ В GDI+

Цель работы

Получение практических навыков построения и визуализации графических примитивов с помощью GDI+.

Основные понятия

В операционной системе Windows приложения не имеют непосредственного доступа к устройствам отображения, таким как монитор или принтер. Вместо этого они обращаются к интерфейсу графического устройства Windows, а он, в свою очередь, транслирует эти обращения к драйверам устройств отображения, обеспечивая аппаратную независимость приложений.

Интерфейс графического устройства (Graphics Device Interface, GDI) – это часть Windows API, которая представляет собой библиотеку функций, обеспечивающих графический вывод на различные устройства отображения.

С выходом Windows XP появилась библиотека GDI+, призванная заменить уже устаревшую библиотеку GDI. Однако, несмотря на это, библиотека GDI+ содержит в себе все возможности своего предшественника и предоставляет множество новых. Следует также отметить, что в новых версиях Windows приложения, использующие GDI, будут выполняться, но все же при создании приложений рекомендуется использовать GDI+.

В библиотеке GDI+ (в отличие от GDI) используется объектно-ориентированный подход к работе с графическими объектами. Библиотека GDI+ содержит около 40 классов. Документация по всем классам GDI+ имеется в Platform Software Development Kit (Platform SDK), которая доступна по адресу: http://msdn.microsoft.com/.

Обработка ошибок в GDI+

Большинство функций и методов классов GDI+ после выполнения возвращают одно из значений перечисляемого типа `Status`, определенного в заголовочном файле gdiplustypes.h. В случае успешного выполнения возвращается значение `Ok`. Остальные же значения типа `Status` указывают на причину возникновения ошибки. Подробное описание всех значений перечисляемого типа `Status` можно найти в документации Platform SDK.

Следует также отметить, что большинство классов GDI+ содержат метод `GetLastStatus`, который возвращает значение, указывающее причину возникновения ошибки (или ее отсутствие) в последнем вызове одного из методов этих классов. Метод `GetLastStatus` имеет следующий прототип:

```
Status GetLastStatus() const;
```

Кроме того, при создании объектов одного из классов GDI+ можно вызвать метод `GetLastStatus` этого объекта, для того чтобы определить успешно или нет был создан объект. В случае успешного создания метод `GetLastStatus` возвращает значение `Ok`. Нужно также отметить, что при создании объектов некоторых классов GDI+ в случае возникновения ошибки метод `GetLastStatus` независимо от ее причины может вернуть значение `OutOfMemory`, указывающее на нехватку памяти для создания объекта.

Цвет в GDI+

Для работы с цветом в GDI+ есть класс `Color`, который определен в заголовочном файле gdipluscolor.h. Класс `Color` имеет следующие конструкторы:

```
Color();
Color(ARGB argb);
Color(BYTE r, BYTE g, BYTE b);
Color(BYTE a, BYTE r, BYTE g, BYTE b);
```

Первый конструктор создает объект класса `Color`, в котором значение цвета соответствует черному цвету. Второй конструктор создает объект класса `Color`, в котором значение цвета задается параметром *argb* в виде 32-разрядного значения в формате ARGB. На рис. 1.1 показано, как кодируется цвет в формате ARGB.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
альфа-фактор (alpha-value)								красный (red)								зеленый (green)								синий (blue)							

Рис. 1.1. Формат ARGB

Третий и четвертый конструкторы создают объект класса `Color` на основе значений компонентов цветовой модели RGB, каждый из которых задается параметрами *r*, *g* и *b* соответственно. Эти параметры могут принимать значения в диапазоне от `0` до `255`. В четвертом конструкторе в качестве параметра *a* задается альфа-фактор, который определяет степень непрозрачности. Этот параметр может принимать

значения в диапазоне от 0 до 255, где значение 0 означает полную прозрачность, а 255 – полную непрозрачность.

В следующем примере проиллюстрировано создание трех объектов класса Color для желтого цвета с использованием разных конструкторов:

```
// создаем объект класса Color для желтого цвета
Color clrYellow1(0xFFFFFF00);

// создаем объект класса Color для желтого цвета
Color clrYellow2(255, 255, 0);

// создаем объект класса Color для желтого цвета
Color clrYellow3(255, 255, 255, 0);
```

В классе Color имеется несколько методов (табл. 1.1), позволяющих получить или задать значение цвета, а также получить значение отдельных компонентов цвета.

Таблица 1.1. Методы класса Color

Метод	Описание
`BYTE GetAlpha() const;` `BYTE GetA() const;`	Возвращает значение альфа-фактора
`BYTE GetBlue() const;` `BYTE GetB() const;`	Возвращает значение синего компонента цвета
`BYTE GetGreen() const;` `BYTE GetG() const;`	Возвращает значение зеленого компонента цвета
`BYTE GetRed() const;` `BYTE GetR() const;`	Возвращает значение красного компонента цвета
`ARGB GetValue() const;`	Возвращает значение цвета в формате ARGB
`static ARGB MakeARGB(BYTE a, BYTE r, BYTE g, BYTE b);`	Собирает значения альфа-фактора и компонентов цвета в одно значение цвета в формате ARGB
`void SetFromCOLORREF(` `  COLORREF rgb);`	Задает значение цвета в формате типа данных COLORREF
`void SetValue(ARGB argb);`	Задает значение цвета в формате ARGB
`COLORREF ToCOLORREF() const;`	Возвращает значение цвета в формате типа данных COLORREF

Тип данных COLORREF

В GDI работа с цветом осуществляется с помощью типа данных COLORREF, который представляет собой целое 32-разрядное значение

без знака. Если старший байт в таком значение равняется `0`, цвет кодируется в цветовом пространстве RGB. Как видно из рис. 1.2, формат кодирования цвета в типе данных `COLORREF` отличается от формата ARGB, который используется в классе `Color` (см. рис. 1.1).

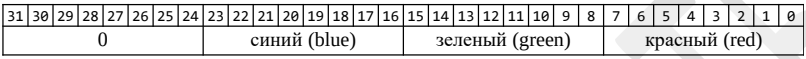


Рис. 1.2. Тип данных COLORREF

При разработке приложений иногда приходится работать с типом данных `COLORREF`. Например, чтобы получить значение системного цвета Windows с помощью функции `GetSysColor`:

```
DWORD GetSysColor(int nIndex);
```

Эта функция возвращает значение (как значение `COLORREF`) системного цвета, на который указывает параметр *`nIndex`*. Аргумент *`nIndex`* должен содержать константу, идентификатор которой начинается с префикса `COLOR_` (например, `COLOR_WINDOW`, которая соответствует цвету фона окна).

В следующем примере продемонстрировано использование метода `SetFromCOLORREF` класса `Color` для получения значения системного цвета Windows в формате ARGB:

Листинг 1.1. Функция для получения системного цвета в формате ARGB

```
1   ARGB GetSysColorAsARGB(int nIndex)
2   {
3       Color result;
4
5       result.SetFromCOLORREF( GetSysColor(nIndex) );
6       return result.GetValue();
7   } // GetSysColorAsARGB
```

Функцию из этого примера можно использовать для создания объектов класса `Color` следующим образом:

```
1   // создаем объект класса Color для цвета фона окна
2   Color clr = GetSysColorAsARGB(COLOR_WINDOW);
```

Предопределенные цветовые константы

В классе `Color` имеется большой набор цветовых констант, которые тоже можно использовать при создании объектов этого класса:

```
1   // создаем объект класса Color для желтого цвета
2   Color clrYellow = Color::Yellow;
```

В таблице 1.2 перечислены все цветовые константы, предопределенные в классе Color.

Таблица 1.2. Набор цветовых констант в классе Color

Константа	Значение (шестн.)	a, r, g, b	Цвет
AliceBlue	FFF0F8FF	255, 240, 248, 255	
AntiqueWhite	FFFAEBD7	255, 250, 235, 215	
Aqua	FF00FFFF	255, 0, 255, 255	
Aquamarine	FF7FFFD4	255, 127, 255, 212	
Azure	FFF0FFFF	255, 240, 255, 255	
Beige	FFF5F5DC	255, 245, 245, 220	
Bisque	FFFFE4C4	255, 255, 228, 196	
Black	FF000000	255, 0, 0, 0	
BlanchedAlmond	FFFFEBCD	255, 255, 235, 205	
Blue	FF0000FF	255, 0, 0, 255	
BlueViolet	FF8A2BE2	255, 138, 43, 226	
Brown	FFA52A2A	255, 165, 42, 42	
BurlyWood	FFDEB887	255, 222, 184, 135	
CadetBlue	FF5F9EA0	255, 95, 158, 160	
Chartreuse	FF7FFF00	255, 127, 255, 0	
Chocolate	FFD2691E	255, 210, 105, 30	
Coral	FFFF7F50	255, 255, 127, 80	
CornflowerBlue	FF6495ED	255, 100, 149, 237	
Cornsilk	FFFFF8DC	255, 255, 248, 220	
Crimson	FFDC143C	255, 220, 20, 60	
Cyan	FF00FFFF	255, 0, 255, 255	
DarkBlue	FF00008B	255, 0, 0, 139	
DarkCyan	FF008B8B	255, 0, 139, 139	
DarkGoldenrod	FFB8860B	255, 184, 134, 11	
DarkGray	FFA9A9A9	255, 169, 169, 169	
DarkGreen	FF006400	255, 0, 100, 0	
DarkKhaki	FFBDB76B	255, 189, 183, 107	
DarkMagenta	FF8B008B	255, 139, 0, 139	

Продолжение табл. 1.2

Константа	Значение (шестн.)	a, r, g, b	Цвет
DarkOliveGreen	FF556B2F	255, 85, 107, 47	
DarkOrange	FFFF8C00	255, 255, 140, 0	
DarkOrchid	FF9932CC	255, 153, 50, 204	
DarkRed	FF8B0000	255, 139, 0, 0	
DarkSalmon	FFE9967A	255, 233, 150, 122	
DarkSeaGreen	FF8FBC8B	255, 143, 188, 139	
DarkSlateBlue	FF483D8B	255, 72, 61, 139	
DarkSlateGray	FF2F4F4F	255, 47, 79, 79	
DarkTurquoise	FF00CED1	255, 0, 206, 209	
DarkViolet	FF9400D3	255, 148, 0, 211	
DeepPink	FFFF1493	255, 255, 20, 147	
DeepSkyBlue	FF00BFFF	255, 0, 191, 255	
DimGray	FF696969	255, 105, 105, 105	
DodgerBlue	FF1E90FF	255, 30, 144, 255	
Firebrick	FFB22222	255, 178, 34, 34	
FloralWhite	FFFFFAF0	255, 255, 250, 240	
ForestGreen	FF228B22	255, 34, 139, 34	
Fuchsia	FFFF00FF	255, 255, 0, 255	
Gainsboro	FFDCDCDC	255, 220, 220, 220	
GhostWhite	FFF8F8FF	255, 248, 248, 255	
Gold	FFFFD700	255, 255, 215, 0	
Goldenrod	FFDAA520	255, 218, 165, 32	
Gray	FF808080	255, 128, 128, 128	
Green	FF008000	255, 0, 128, 0	
GreenYellow	FFADFF2F	255, 173, 255, 47	
Honeydew	FFF0FFF0	255, 240, 255, 240	
HotPink	FFFF69B4	255, 255, 105, 180	
IndianRed	FFCD5C5C	255, 205, 92, 92	
Indigo	FF4B0082	255, 75, 0, 130	
Ivory	FFFFFFF0	255, 255, 255, 240	

Продолжение табл. 1.2

Константа	Значение (шестн.)	a, r, g, b	Цвет
Khaki	FFF0E68C	255, 240, 230, 140	
Lavender	FFE6E6FA	255, 230, 230, 250	
LavenderBlush	FFFFF0F5	255, 255, 240, 245	
LawnGreen	FF7CFC00	255, 124, 252, 0	
LemonChiffon	FFFFFACD	255, 255, 250, 205	
LightBlue	FFADD8E6	255, 173, 216, 230	
LightCoral	FFF08080	255, 240, 128, 128	
LightCyan	FFE0FFFF	255, 224, 255, 255	
LightGoldenrodYellow	FFFAFAD2	255, 250, 250, 210	
LightGray	FFD3D3D3	255, 211, 211, 211	
LightGreen	FF90EE90	255, 144, 238, 144	
LightPink	FFFFB6C1	255, 255, 182, 193	
LightSalmon	FFFFA07A	255, 255, 160, 122	
LightSeaGreen	FF20B2AA	255, 32, 178, 170	
LightSkyBlue	FF87CEFA	255, 135, 206, 250	
LightSlateGray	FF778899	255, 119, 136, 153	
LightSteelBlue	FFB0C4DE	255, 176, 196, 222	
LightYellow	FFFFFFE0	255, 255, 255, 224	
Lime	FF00FF00	255, 0, 255, 0	
LimeGreen	FF32CD32	255, 50, 205, 50	
Linen	FFFAF0E6	255, 250, 240, 230	
Magenta	FFFF00FF	255, 255, 0, 255	
Maroon	FF800000	255, 128, 0, 0	
MediumAquamarine	FF66CDAA	255, 102, 205, 170	
MediumBlue	FF0000CD	255, 0, 0, 205	
MediumOrchid	FFBA55D3	255, 186, 85, 211	
MediumPurple	FF9370DB	255, 147, 112, 219	
MediumSeaGreen	FF3CB371	255, 60, 179, 113	
MediumSlateBlue	FF7B68EE	255, 123, 104, 238	
MediumSpringGreen	FF00FA9A	255, 0, 250, 154	

Продолжение табл. 1.2

Константа	Значение (шестн.)	a, r, g, b	Цвет
MediumTurquoise	FF48D1CC	255, 72, 209, 204	
MediumVioletRed	FFC71585	255, 199, 21, 133	
MidnightBlue	FF191970	255, 25, 25, 112	
MintCream	FFF5FFFA	255, 245, 255, 250	
MistyRose	FFFFE4E1	255, 255, 228, 225	
Moccasin	FFFFE4B5	255, 255, 228, 181	
NavajoWhite	FFFFDEAD	255, 255, 222, 173	
Navy	FF000080	255, 0, 0, 128	
OldLace	FFFDF5E6	255, 253, 245, 230	
Olive	FF808000	255, 128, 128, 0	
OliveDrab	FF6B8E23	255, 107, 142, 35	
Orange	FFFFA500	255, 255, 165, 0	
OrangeRed	FFFF4500	255, 255, 69, 0	
Orchid	FFDA70D6	255, 218, 112, 214	
PaleGoldenrod	FFEEE8AA	255, 238, 232, 170	
PaleGreen	FF98FB98	255, 152, 251, 152	
PaleTurquoise	FFAFEEEE	255, 175, 238, 238	
PaleVioletRed	FFDB7093	255, 219, 112, 147	
PapayaWhip	FFFFEFD5	255, 255, 239, 213	
PeachPuff	FFFFDAB9	255, 255, 218, 185	
Peru	FFCD853F	255, 205, 133, 63	
Pink	FFFFC0CB	255, 255, 192, 203	
Plum	FFDDA0DD	255, 221, 160, 221	
PowderBlue	FFB0E0E6	255, 176, 224, 230	
Purple	FF800080	255, 128, 0, 128	
Red	FFFF0000	255, 255, 0, 0	
RosyBrown	FFBC8F8F	255, 188, 143, 143	
RoyalBlue	FF4169E1	255, 65, 105, 225	
SaddleBrown	FF8B4513	255, 139, 69, 19	
Salmon	FFFA8072	255, 250, 128, 114	

Окончание табл. 1.2

Константа	Значение (шестн.)	a, r, g, b	Цвет
SandyBrown	FFF4A460	255, 250, 164, 96	
SeaGreen	FF2E8B57	255, 46, 139, 87	
SeaShell	FFFFF5EE	255, 255, 245, 238	
Sienna	FFA0522D	255, 160, 82, 45	
Silver	FFC0C0C0	255, 192, 192, 192	
SkyBlue	FF87CEEB	255, 135, 206, 235	
SlateBlue	FF6A5ACD	255, 106, 90, 205	
SlateGray	FF708090	255, 112, 128, 144	
Snow	FFFFFAFA	255, 255, 250, 250	
SpringGreen	FF00FF7F	255, 0, 255, 127	
SteelBlue	FF4682B4	255, 70, 130, 180	
Tan	FFD2B48C	255, 210, 180, 140	
Teal	FF008080	255, 0, 128, 128	
Thistle	FFD8BFD8	255, 216, 191, 216	
Tomato	FFFF6347	255, 255, 99, 71	
Transparent	00FFFFFF	0, 255, 255, 255	
Turquoise	FF40E0D0	255, 64, 224, 208	
Violet	FFEE82EE	255, 238, 130, 238	
Wheat	FFF5DEB3	255, 245, 222, 179	
White	FFFFFFFF	255, 255, 255, 255	
WhiteSmoke	FFF5F5F5	255, 245, 245, 245	
Yellow	FFFFFF00	255, 255, 255, 0	
YellowGreen	FF9ACD32	255, 154, 205, 50	

Расположение и размеры в GDI+

В табл. 1.3 представлены классы GDI+, используемые для задания или определения расположения и/или размеров различных фигур или графических объектов GDI+. Использовать эти классы бывает зачастую гораздо удобнее, нежели отдельные величины.

Все классы, представленные в табл. 1.3, определены в заголовочном файле gdiplustypes.h.

Таблица 1.3. Классы GDI+ для работы с координатами и размерами

Класс	Поля	Описание
`Point`	*X*, *Y*	Содержит пару целых чисел, представляющих собой координаты точки
`PointF`	*X*, *Y*	Содержит пару чисел с плавающей запятой, представляющих собой координаты точки
`Size`	*Width*, *Height*	Содержит пару целых чисел, представляющих собой ширину и высоту
`SizeF`	*Width*, *Height*	Содержит пару чисел с плавающей запятой, представляющих собой ширину и высоту
`Rect`	*X*, *Y*, *Width*, *Height*	Содержит набор из четырех целых чисел, определяющих расположение и размер прямоугольника
`RectF`	*X*, *Y*, *Width*, *Height*	Содержит набор из четырех чисел с плавающей запятой, определяющих расположение и размер прямоугольника

Point и PointF

Классы `Point` и `PointF` используются для представления координат точек в двумерной прямоугольной системе координат. При этом в классе `Point` поля *X* и *Y* имеют тип `int`, а в классе `PointF` – `float`. В остальном классы `Point` и `PointF` идентичны. Поэтому достаточно рассмотреть только класс `Point`.

Класс `Point` имеет следующие конструкторы:

```
Point();
Point(INT x, INT y);
Point(const Point &point);
Point(const Size &size);
```

Первый конструктор создает объект класса `Point`, в котором значения полей *X* и *Y* равны нулю. Второй конструктор создает объект класса `Point`, в котором значения полей *X* и *Y* задаются параметрами *x* и *y*. Третий конструктор создает копию объекта *point*. Четвертый конструктор создает объект класса `Point` на основе объекта класса `Size`.

В классе `Point` определены операторы «+» и «–», которые выполняют, соответственно, сложение и вычитание координат точек:

```
1   Point ptA, ptB(5,3); // точки A(0,0) и B(5,3)
2
3   Point pt1 = ptA + ptB; // сложение двух точек
4   Point pt2 = ptB - ptA; // вычитание двух точек
```

В классе Point также определен метод Equals, который выясняет равенство двух объектов класса Point, сравнивая их соответствующие поля. Метод Equals имеет следующий прототип:

```
BOOL Equals(const Point &point);
```

Если объекты класса Point равны метод Equals, возвращает TRUE, иначе – FALSE.

В следующем примере продемонстрировано использование метода Equals класса Point для сравнения двух точек:

```
1   Point pt1(5,3); // точка (5,3)
2   Point pt2(5,3); // точка (5,3)
3
4   if (pt1.Equals(pt2)) // сравнение двух точек
5   {
6       /* pt1 и pt2 равны */
7   } // if
```

Size и SizeF

Классы Size и SizeF используются для представления пары чисел, которые определяют ширину и высоту. При этом в классе Size поля *Width* и *Height* имеют тип int, а в классе SizeF – float. В остальном классы Size и SizeF идентичны. Поэтому так же, как в случае с классами Point и PointF, будет достаточно рассмотреть только класс Size.

Класс Size имеет следующие конструкторы:

```
Size();
Size(INT width, INT height);
Size(const Size &size);
```

Первый конструктор создает объект класса Size, в котором значения полей *Width* и *Height* равны нулю. Второй конструктор создает объект класса Size, в котором значения полей *Width* и *Height* задаются параметрами *width* и *height*. Третий конструктор создает копию объекта *size*.

В классе Size определен оператор «+», который выполняет сложение значений соответствующих полей, и оператор «–», который выполняет их вычитание.

```
1   // сложение двух величин (4,1) и (3,5)
2   Size size1 = Size(4,1) + Size(3,5); // результат (7,6)
3
4   // вычитание двух величин (7,6) и (3,5)
5   Size size2 = size1 - Size(3,5); // результат (4,1)
```

В классе Size определены два метода – Equals и Empty, которые имеют следующие прототипы:

```
BOOL Equals(const Size &sz) const;
BOOL Empty() const;
```

Метод Equals определяет равенство двух объектов класса Size, сравнивая их соответствующие поля. Если объекты класса Size равны метод Equals, возвращает TRUE, и FALSE – в противном случае.

Метод Empty возвращает TRUE, если поля *Width* и *Height* равны нулю; иначе – FALSE.

Rect и RectF

Классы Rect и RectF используются для представления прямоугольников. Классы Rect и RectF в основном идентичны, с тем отличием, что в классе Rect поля *X*, *Y*, *Width* и *Height* имеют тип int, а в классе RectF – float. Поэтому здесь будет рассмотрен только класс Rect.

Класс Rect имеет следующие конструкторы:

```
Rect();
Rect(INT x, INT y, INT width, INT height);
Rect(const Point &location, const Size &size);
```

Первый конструктор создает объект класса Rect, в котором значения полей *X*, *Y*, *Width* и *Height* равны нулю. Второй конструктор создает объект класса Rect, в котором значения полей *X*, *Y*, *Width* и *Height* задаются параметрами *x*, *y*, *width* и *height*. Третий конструктор создает объект класса Rect на основе объектов классов Point и Size, которые, соответственно, задают расположение и размер прямоугольника.

В классе Rect определено несколько методов, упрощающую работу с прямоугольником. Методы класса Rect перечислены в табл. 1.4.

Таблица 1.4. Методы класса Rect

Метод	Описание
`Rect* Clone() const;`	Клонирует текущий объект класса Rect. Возвращает указатель на созданный объект класса Rect, который должен быть удален вызовом оператора delete после того как станет не нужен

`BOOL Contains(`
`  const Point &pt) const;`
`BOOL Contains(`
`  INT x, INT y) const;`
	Возвращает TRUE, если прямоугольник, представленный текущим объектом класса Rect, включает указанную точку, и FALSE – в противном случае

Продолжение табл. 1.4

Метод	Описание
`BOOL Contains(` `  Rect &rect) const;`	Возвращает `TRUE`, если прямоугольник, представленный текущим объектом класса `Rect`, включает прямоугольник, указанный параметром *`rect`*, и `FALSE` – в противном случае
`BOOL Equals(` `  const Rect &rect) const;`	Возвращает `TRUE`, если прямоугольник, представленный текущим объектом класса `Rect`, равен прямоугольнику, указанному параметром *`rect`*, и `FALSE` – в противном случае
`INT GetBottom() const;`	Возвращает координату по оси Y нижней стороны прямоугольника
`void GetBounds(` `  Rect *rect) const;`	Возвращает прямоугольник, представленный текущим объектом класса `Rect`. Результат сохраняется в объекте класса `Rect`, на который указывает параметр *`rect`*
`INT GetLeft() const;`	Возвращает координату по оси X левой стороны прямоугольника
`void GetLocation(` `  Point *point) const;`	Возвращает расположение прямоугольника, представленного текущим объектом класса `Rect`. Результат сохраняется в объекте класса `Point`, на который указывает параметр *`point`*
`INT GetRight() const;`	Возвращает координату по оси X правой стороны прямоугольника
`void GetSize(Size *size) const;`	Возвращает размер прямоугольника, представленного текущим объектом класса `Rect`. Результат сохраняется в объекте класса `Size`, на который указывает параметр *`size`*
`void Inflate(INT dx, INT dy);`	Расширяет прямоугольник, представленный текущим объектом класса `Rect`, на значение *`dx`* с левой и правой стороны и *`dy`* с верхней и нижней стороны
`void Inflate(` `  const Point &point);`	Расширяет прямоугольник, представленный текущим объектом класса `Rect`, на значение *`point.X`* с левой и правой стороны и *`point.Y`* с верхней и нижней стороны

Окончание табл. 1.4

Метод	Описание
`static BOOL Intersect(Rect &c, const Rect &a, const Rect &b);`	Находит пересечение прямоугольников, указанных параметрами *a* и *b*. Результат сохраняется в объекте класса `Rect`, на который указывает параметр *c*. Возвращаемое значение равно `TRUE`, если пересечение найдено, и `FALSE` – в противном случае
`BOOL Intersect(const Rect &rect);`	Находит пересечение прямоугольника, представленного текущим объектом класса `Rect`, и прямоугольника, указанного параметром *rect*. Результат сохраняется в текущем объекте класса `Rect`. Возвращаемое значение равно `TRUE`, если пересечение найдено, и `FALSE` – в противном случае
`BOOL IntersectsWith(const Rect &rect) const;`	Возвращает `TRUE`, если прямоугольник, представленный текущим объектом класса `Rect`, пересекается с прямоугольником, указанным параметром *rect*, и `FALSE` – в противном случае
`BOOL IsEmptyArea() const;`	Возвращает `TRUE`, если поля *Width* и *Height* текущего объекта класса `Rect` меньше или равны нулю, и `FALSE` – в противном случае
`void Offset(INT dx, INT dy);`	Перемещает прямоугольник, представленный текущим объектом класса `Rect`, на значение *dx* по горизонтали и *dy* по вертикали
`void Offset(const Point &point);`	Перемещает прямоугольник, представленный текущим объектом класса `Rect`, на значение *point.X* по горизонтали и *point.Y* по вертикали
`static BOOL Union(Rect &c, const Rect &a, const Rect &b);`	Находит объединение прямоугольников, указанных параметрами *a* и *b*. Результат сохраняется в объекте класса `Rect`, на который указывает параметр *c*. Возвращаемое значение равно `TRUE`, если объединение найдено, и `FALSE` – в противном случае

Подробное описание методов класса `Rect` можно найти в документации Platform SDK.

Графические объекты GDI+

Библиотека GDI+ содержит набор графических объектов, обеспечивающих выполнение различных графических операций. К таким объектам относятся кисти, перья, шрифты и изображения.

Кисти

Замкнутая фигура (например, такая как окружность или многоугольник) закрашивается с помощью *кисти* (*brush*). В GDI+ имеется несколько классов для работы кистями, каждый из которых является производным от класса `Brush`.

Большинство классов GDI+ для работы кистями (включая класс `Brush`) определено в заголовочном файле gdiplusbrush.h.

Сплошные кисти

Сплошная кисть (класс `SolidBrush`) используется для заполнения замкнутой фигуры однородным цветом.

Класс `SolidBrush` имеет следующий конструктор:

```
SolidBrush(const Color &color);
```

Этот конструктор создает объект класса `SolidBrush` на основе объекта класса `Color`, который задает цвет сплошной кисти.

В классе `SolidBrush` определены два метода – `GetColor` и `SetColor`, которые работают с цветом кисти. Эти методы имеют следующие прототипы:

```
Status GetColor(Color *color) const;
Status SetColor(const Color &color);
```

Метод `GetColor` сохраняет текущее значение цвета кисти в объект класса `Color`, на который указывает параметр *color*, а метод `SetColor` задает новое значение цвета кисти.

В следующем примере показано, что цвет кисти можно задать как в конструкторе класса `SolidBrush`, так и позднее, для уже созданного объекта класса `SolidBrush`:

```
1  // создаем сплошную кисть
2  SolidBrush solidBrush(Color(0, 255, 0)); // зеленый цвет
3
4  // изменяем цвет кисти
5  solidBrush.SetColor(Color::Yellow); // желтый цвет
```

Штриховые кисти

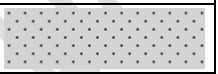
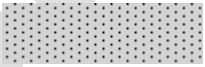
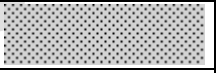
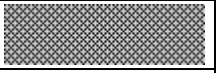
Штриховая кисть (класс `HatchBrush`) позволяет заполнять замкнутую фигуру с использованием определенного узора.

Класс `HatchBrush` имеет следующий конструктор:

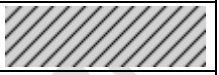
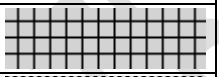
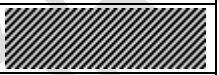
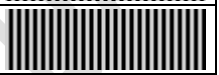
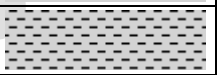
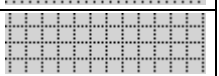
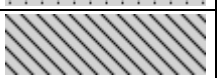
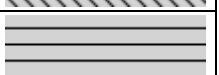
```
HatchBrush(HatchStyle hatchStyle, const Color &foreColor,
    const Color &backColor = Color());
```

Этот конструктор создает объект класса `HatchBrush` на основе предопределенного стиля штриховой кисти, который задает его первый параметр – *`hatchStyle`*. Этот параметр должен принимать одно из значений перечисляемого типа `HatchStyle`, приведенных в табл. 1.5. Второй параметр, *`foreColor`*, задает цвет, который используется при отображении линий узора штриховой кисти, а третий параметр, *`backColor`*, задает цвет фона штриховой кисти.

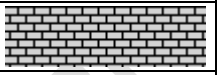
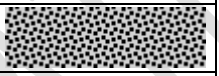
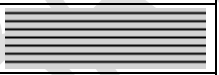
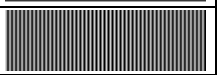
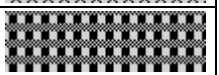
Таблица 1.5. Значения параметра *`hatchStyle`*

Значение	**Узор**
`HatchStyle05Percent`	
`HatchStyle10Percent`	
`HatchStyle20Percent`	
`HatchStyle25Percent`	
`HatchStyle30Percent`	
`HatchStyle40Percent`	
`HatchStyle50Percent`	
`HatchStyle60Percent`	
`HatchStyle70Percent`	
`HatchStyle75Percent`	
`HatchStyle80Percent`	
`HatchStyle90Percent`	

Продолжение табл. 1.5

Значение	Узор
HatchStyleBackwardDiagonal	
HatchStyleCross	
HatchStyleDarkDownwardDiagonal	
HatchStyleDarkHorizontal	
HatchStyleDarkUpwardDiagonal	
HatchStyleDarkVertical	
HatchStyleDashedDownwardDiagonal	
HatchStyleDashedHorizontal	
HatchStyleDashedUpwardDiagonal	
HatchStyleDashedVertical	
HatchStyleDiagonalBrick	
HatchStyleDiagonalCross	
HatchStyleDivot	
HatchStyleDottedDiamond	
HatchStyleDottedGrid	
HatchStyleForwardDiagonal	
HatchStyleHorizontal	

Продолжение табл. 1.5

Значение	Узор
HatchStyleHorizontalBrick	
HatchStyleLargeCheckerBoard	
HatchStyleLargeConfetti	
HatchStyleLightDownwardDiagonal	
HatchStyleLightHorizontal	
HatchStyleLightUpwardDiagonal	
HatchStyleLightVertical	
HatchStyleNarrowHorizontal	
HatchStyleNarrowVertical	
HatchStyleOutlinedDiamond	
HatchStylePlaid	
HatchStyleShingle	
HatchStyleSmallCheckerBoard	
HatchStyleSmallConfetti	
HatchStyleSmallGrid	
HatchStyleSolidDiamond	
HatchStyleSphere	

Окончание табл. 1.5

Значение	Узор
HatchStyleTrellis	
HatchStyleVertical	
HatchStyleWave	
HatchStyleWeave	
HatchStyleWideDownwardDiagonal	
HatchStyleWideUpwardDiagonal	
HatchStyleZigZag	

В табл. 1.6 перечислены методы класса `HatchBrush`, которые могут быть полезны при работе с объектами этого класса.

Таблица 1.6. Методы класса `HatchBrush`

Метод	Описание
`Status GetBackgroundColor( Color *color) const;`	Возвращает цвет, который используется при отображении узора штриховой кисти. Результат сохраняется в объект класса `Color`, на который указывает параметр *color*
`Status GetForegroundColor( Color *color) const;`	Возвращает цвет фона штриховой кисти. Результат сохраняется в объект класса `Color`, на который указывает параметр *color*
`HatchStyle GetHatchStyle() const;`	Возвращает стиль штриховой кисти

Текстурные кисти

Текстурная кисть (класс `TextureBrush`) позволяет заполнять замкнутую фигуру с использованием изображения.

Класс `TextureBrush` имеет следующие конструкторы:

```
TextureBrush(Image *image, WrapMode wrapMode = WrapModeTile);
```

```
TextureBrush(Image *image, WrapMode wrapMode,
    const RectF &dstRect);

TextureBrush(Image *image, WrapMode wrapMode,
    const RectF &dstRect,
    const ImageAttributes *imageAttributes = NULL);

TextureBrush(Image *image, WrapMode wrapMode,
    const Rect &dstRect,
    const ImageAttributes *imageAttributes = NULL);

TextureBrush(Image *image, WrapMode wrapMode,
    const Rect &dstRect);

TextureBrush(Image *image, WrapMode wrapMode, REAL dstX,
    REAL dstY, REAL dstWidth, REAL dstHeight);

TextureBrush(Image *image, WrapMode wrapMode, INT dstX,
    INT dstY, INT dstWidth, INT dstHeight);
```

Все эти конструкторы создают объект класса `TextureBrush` на основе объекта класса `Image`, на который указывает параметр *`image`*. Объект класса `Image`, который в GDI+ используется для работы с изображениями (подробнее см. в разделе «Изображения»), содержит изображение (например, как на рис. 1.3), используемое для создания текстурной кисти.

Рис. 1.3. Пример изображения для создания кисти

Параметр *`wrapMode`* задает режим заполнения, когда изображение меньше заполняемой области. Этот параметр должен принимать одно из значений перечисляемого типа `WrapMode`:

- `WrapModeTile` – мозаичное заполнение;
- `WrapModeTileFlipX` – мозаичное заполнение с зеркальным отображением изображения по горизонтали;
- `WrapModeTileFlipY` – мозаичное заполнение с зеркальным отображением изображения по вертикали;
- `WrapModeTileFlipXY` – мозаичное заполнение с зеркальным отображением изображения по горизонтали и вертикали;
- `WrapModeClamp` – заполнение не производится.

На рис. 1.4 показаны возможные варианты мозаичного заполнения прямоугольника изображением, представленным на рис. 1.3.

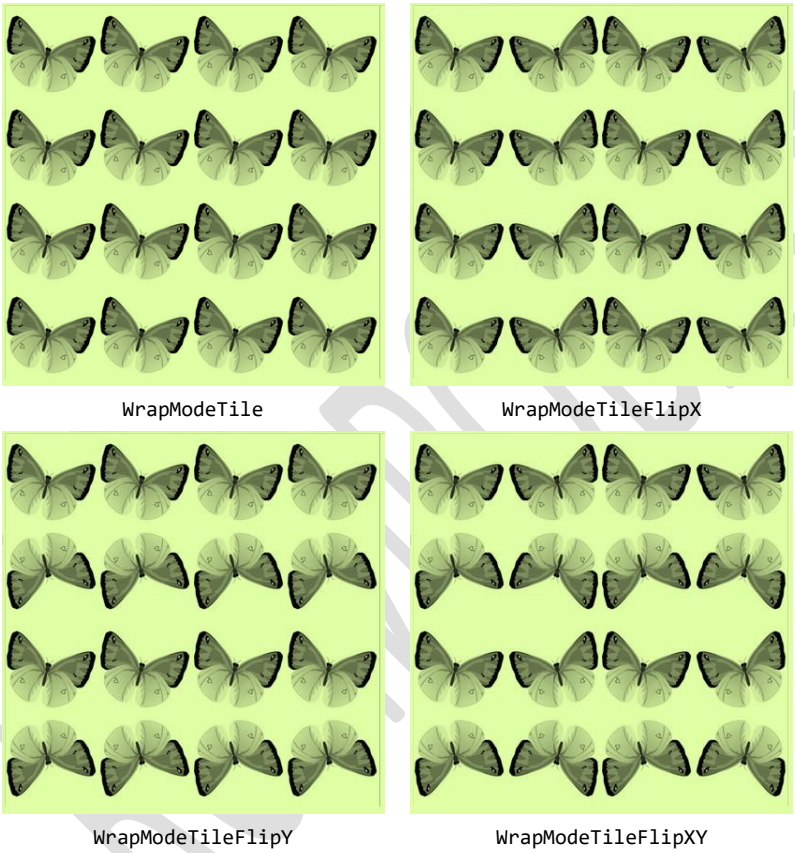


Рис. 1.4. Мозаичное заполнение

Параметр *dstRect*, а также параметры *dstX*, *dstY*, *dstWidth* и *dstHeight*, определяют часть изображения, которая будет использоваться для создания кисти.

Параметр *imageAttributes*, указывает на объект класса ImageAttributes, задающий свойства изображения (подробнее см. в документации Platform SDK). Этот параметр может быть установлен в NULL.

В классе TextureBrush определено несколько методов, которые могут быть полезны при работе с объектами этого класса. Основные методы этого класса перечислены в табл. 1.7.

Таблица 1.7. Методы класса `TextureBrush`

Метод	Описание
`Image* GetImage() const;`	Возвращает указатель на объект класса `Image`, который использовался при создании текстурной кисти
`WrapMode GetWrapMode() const;`	Возвращает режим заполнения
`Status SetWrapMode(` `  WrapMode wrapMode);`	Задает режим заполнения

Полный перечень методов класса `TextureBrush`, а также их описание, можно найти в документации Platform SDK.

Кисти линейного градиента

Кисть линейного градиента (класс `LinearGradientBrush`) используется для заполнения замкнутой фигуры цветовым градиентом, в котором цвет меняется параллельно некоторой линии.

Класс `LinearGradientBrush` имеет следующие конструкторы:

```
LinearGradientBrush(const PointF &point1,
    const PointF &point2, const Color &color1,
    const Color &color2);

LinearGradientBrush(const Point &point1, const Point &point2,
    const Color &color1, const Color &color2);

LinearGradientBrush(const RectF &rect, const Color &color1,
    const Color &color2, LinearGradientMode mode);

LinearGradientBrush(const Rect &rect, const Color &color1,
    const Color &color2, LinearGradientMode mode);

LinearGradientBrush(const RectF &rect, const Color &color1,
    const Color &color2, REAL angle,
    BOOL isAngleScalable = FALSE);

LinearGradientBrush(const Rect &rect, const Color &color1,
    const Color &color2, REAL angle,
    BOOL isAngleScalable = FALSE);
```

Все эти конструкторы создают объект класса `LinearGradientBrush` на основе двух объектов класса `Color`, которые задают начальный и конечный цвет в линейном градиенте (соответственно параметры *`color1`* и *`color2`*).

В первом и втором конструкторах класса `LinearGradientBrush` параметры *`point1`* и *`point2`* задают, соответственно, начальную и конечную точку линейного градиента, которые образуют вектор. Как видно

из рис. 1.5, длина этого вектора определяет плавность градиента, а его направление – направление градиента.

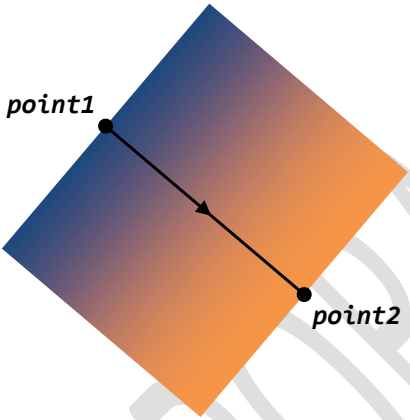


Рис. 1.5. Кисть линейного градиента, определяемая двумя точками

В третьем и четвертом конструкторах класса `LinearGradientBrush` параметр *`rect`* задает прямоугольник, в котором определены начальная и конечная точки линейного градиента. При этом на определение этих точек влияет параметр *`mode`*, который должен принимать одно из значений перечисляемого типа `LinearGradientMode`:

- `LinearGradientModeHorizontal` – начальная точка линейного градиента расположена в левом верхнем углу заданного прямоугольника, а конечная – в правом верхнем;
- `LinearGradientModeVertical` – начальная точка линейного градиента расположена в левом верхнем углу заданного прямоугольника, а конечная – в левом нижнем;
- `LinearGradientModeForwardDiagonal` – начальная точка линейного градиента расположена в левом верхнем углу заданного прямоугольника, а конечная – в правом нижнем;
- `LinearGradientModeBackwardDiagonal` – начальная точка линейного градиента расположена в правом верхнем углу заданного прямоугольника, а конечная – в левом нижнем.

На рис. 1.6 проиллюстрированы возможные варианты расположения начальной и конечной точки линейного градиента, определяемого прямоугольником.

В пятом и шестом конструкторах класса `LinearGradientBrush` параметр *`rect`* задает прямоугольник, в верхнем левом углу которого

находится начальная точка линейного градиента, а в правом нижнем углу – конечная точка. Параметр *angle* задает угол (в градусах) по часовой стрелке от верхней границы заданного прямоугольника. При этом если параметр *isAngleScalable* принимает значение равное FALSE, параметр *angle* задает угол (рис. 1.7, *а*), определяющий направление градиента. Если же параметр *isAngleScalable* принимает значение равное TRUE, параметр *angle* задает базовый угол (рис. 1.7, *б*), из которого рассчитывается угол, определяющий направление градиента, по следующей формуле:

$$\beta = \arctan\left(\frac{width}{height}\tan\alpha\right), \tag{1.1}$$

где β – угол, определяющий направление градиента; $width$ и $height$ – ширина и высота прямоугольника; а α – базовый угол. Следует отметить, что соотношение (1.1) справедливо только, если α меньше 90°.

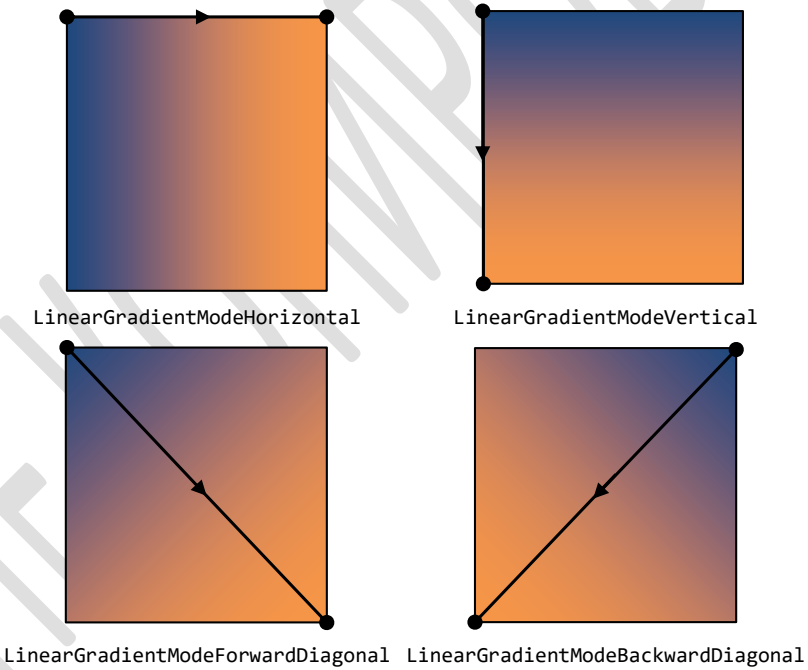


Рис. 1.6. Кисть линейного градиента, определяемая прямоугольником

Рассмотрим некоторые методы класса LinearGradientBrush, которые могут пригодиться при работе кистями линейного градиента. Пол-

ный перечень и подробное описание методов класса `LinearGradient-Brush` можно найти в документации Platform SDK.

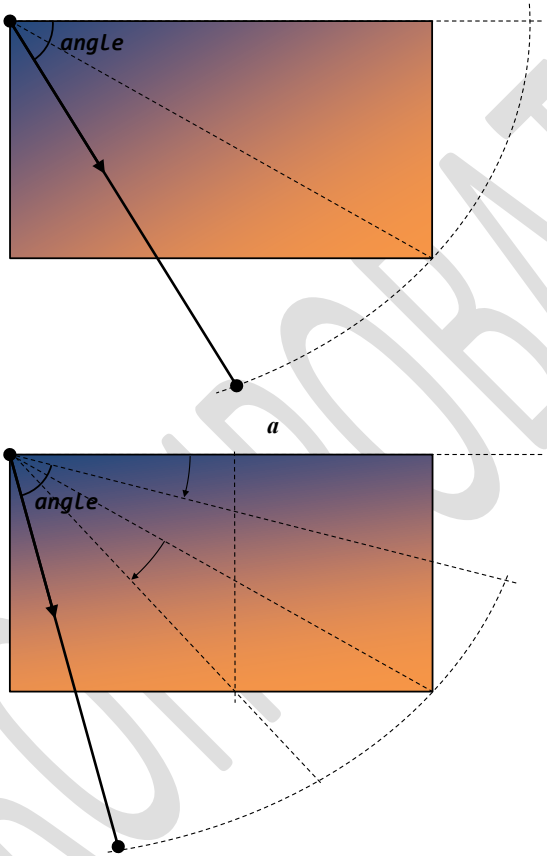


Рис. 1.7. Кисть линейного градиента, определяемая прямоугольником и углом, который задается (*а*) или вычисляется из базового угла (*б*)

Для изменения начального и конечного цвета линейного градиента в классе `LinearGradientBrush` определен метод `SetLinearColors`:

```
Status SetLinearColors(const Color &color1,
    const Color &color2);
```

Параметры *color1* и *color2* задают, соответственно, начальный и конечный цвет линейного градиента.

В классе `LinearGradientBrush` также имеется метод `GetLinearColors`, который возвращает начальный и конечный цвет градиента:

```
Status GetLinearColors(Color *colors) const;
```

Параметр *colors* указывает на массив из двух объектов класса `Color`, в которые будут сохранены значения начального и конечного цвета линейного градиента.

В следующем примере показано, как можно получить начальный и конечный цвет с помощью метода `GetLinearColors`.

Листинг 1.2. Пример получения крайних цветов кисти линейного градиента

```
// создаем кисть линейного градиента
LinearGradientBrush linGrBrush(
    Rect(0, 0, 100, 50),
    Color(255, 0, 0, 0), // черный цвет
    Color(255, 0, 0, 255), // синий цвет
    LinearGradientModeHorizontal);

// получаем информацию о кисти линейного градиента
Color colors[2];
linGrBrush.GetLinearColors(colors);
```

Следует отметить, что цвет в линейном градиенте может меняться не только между двумя цветами. В классе `LinearGradientBrush` есть метод `SetInterpolationColors`, позволяющий задать несколько цветов интерполяции, между которыми будет изменяться цвет в линейном градиенте.

```
Status SetInterpolationColors(const Color *presetColors,
    const REAL *blendPositions, INT count);
```

Первый параметр, *presetColors*, указывает на массив объектов класса `Color`, каждый из которых хранит значение цветов интерполяции в линейном градиенте.

Второй параметр, *blendPositions*, указывает на массив значений типа `float`, которые задают положение цвета в градиенте. Значение каждого элемента этого массива должно находиться в диапазоне от `0.0` до `1.0`, где `0.0` указывает начало градиента, а `1.0` – окончание градиента. Поэтому в массиве, на который указывает параметр *blendPositions*, должно быть, по крайней мере, два элемента: начало градиента и его окончание.

Третий параметр, *count*, задает количество элементов в массиве *presetColors*. При этом количество элементов в массиве *presetColors* должно быть таким же, как и в массиве *blendPositions*.

В листинге 1.3 показан пример создания кисти линейного градиента, изображенного на рис. 1.8.

Листинг 1.3. Создание градиентной кисти, определяемой тремя цветами

```
// создаем кисть линейного градиента
LinearGradientBrush linGrBrush(
    Rect(0, 0, 200, 200), Color::White, Color::Black, 45.f);

// изменяем параметры кисти линейного градиента...

Color colors[3] = {
    Color(255, 255, 0, 0),  // красный цвет
    Color(255, 0, 0, 255),  // синий цвет
    Color(255, 0, 255, 0)}; // зеленый цвет

float pos[3] = {
    0.0f,  // начало градиента
    0.3f,  // 30% от начала градиента до его конца
    1.0f}; // конец градиента

linGrBrush.SetInterpolationColors(colors, pos, 3);
```

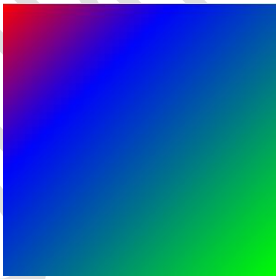


Рис. 1.8. Линейный градиент, определяемый тремя цветами

Определить количество установленных цветов интерполяции в линейном градиенте можно с помощью метода `GetInterpolationColorCount`, а получить значения этих цветов и их позиции в градиенте можно с помощью метода `GetInterpolationColors`.

```
INT GetInterpolationColorCount() const;

Status GetInterpolationColors(Color *presetColors,
    REAL *blendPositions, INT count) const;
```

Первый параметр, *presetColors*, указывает на массив объектов класса `Color`, в который будут сохранены значения цветов интерполяции в линейном градиенте.

Второй параметр, *blendPositions*, указывает на массив типа float, в который будут сохранены позиции цветов в градиенте.

Третий параметр, *count*, задает количество элементов в массиве *presetColors*. При этом количество элементов в массиве *presetColors* должно быть таким же, как и в массиве *blendPositions*.

По умолчанию цвет в линейном градиенте меняется однородным образом. Однако можно настроить линейный градиент так, чтобы изменение цвета осуществлялось неоднородным образом. В классе LinearGradientBrush есть метод SetBlend, который позволяет настраивать способ градиентного изменения цвета при движении от начала до конца градиента.

```
Status SetBlend(const REAL *blendFactors,
    const REAL *blendPositions, INT count);
```

Первый параметр, *blendFactors*, указывает на массив значений типа float, которые задают факторы наложения. Значение каждого элемента в этом массиве определяет процент от конечного цвета линейного градиента и должно задаваться в пределах от 0.0 до 1.0.

Второй параметр, *blendPositions*, указывает на массив значений типа float, которые задают положение фактора наложения в градиенте. Значение каждого элемента этого массива должно находиться в диапазоне от 0.0 до 1.0, где 0.0 указывает начало градиента, а 1.0 – окончание градиента.

Третий параметр, *count*, задает количество элементов в массиве *blendFactors*. При этом количество элементов в массиве *blendFactors* должно быть таким же, как и в массиве *blendPositions*.

Следующий пример демонстрирует создание кисти линейного градиента, изображенного на рис. 1.9, *б*.

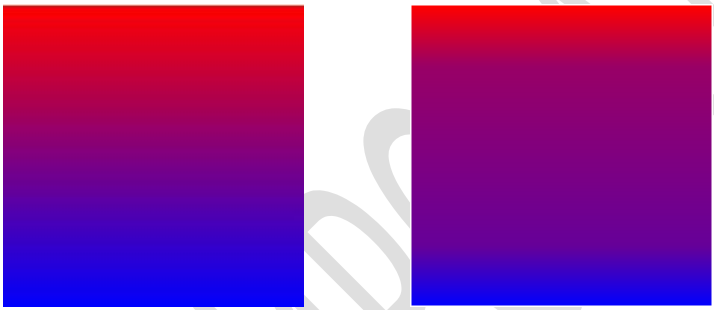
Листинг 1.4. Создание градиентной кисти с неоднородным изменением цвета

```
1   // создаем кисть линейного градиента
2   LinearGradientBrush linGrBrush(
3       Point(0, 0),
4       Point(0, 200),
5       Color::Red, // красный цвет
6       Color::Blue); // синний цвет
7
8   // изменяем параметры кисти линейного градиента...
9
10  float factors[4] = {
11      0.0f,  // 0% синиего цвета, соответственно 100% красного
12      0.4f,  // 40% синиего цвета, соответственно 60% красного
13      0.6f,  // 60% синиего цвета, соответственно 40% красного
```

```
    1.0f}; // 100% синиего цвета, соответственно 0% красного

float pos[4] = {
    0.0f,  // начало градиента
    0.2f,  // 20% от начала градиента до его конца
    0.8f,  // 80% от начала градиента до его конца
    1.0f}; // конец градиента

linGrBrush.SetBlend(factors, pos, 4);
```



а *б*

Рис. 1.9. Цветовые градиенты
с однородным (*а*) и неоднородным (*б*) изменением цвета

Определить количество установленных факторов наложения в линейном градиенте можно с помощью метода `GetBlendCount`, а получить значения этих факторов и их позиции в градиенте можно с помощью метода `GetBlend`. Прототипы этих методов имеют следующий вид:

```
INT GetBlendCount() const;

Status GetBlend(REAL *blendFactors, REAL *blendPositions,
    INT count) const;
```

Первый параметр, *`blendFactors`*, указывает на массив типа `float`, в который будут сохранены значения факторов наложения в линейном градиенте.

Второй параметр, *`blendPositions`*, указывает на массив значений типа `float`, в который будут сохранены позиции факторов наложения в градиенте.

Третий параметр, *`count`*, задает количество элементов в массиве *`blendFactors`*. При этом количество элементов в массиве *`blendFactors`* должно быть таким же, как в массиве *`blendPositions`*.

Чтобы интенсивность цветов в линейном градиенте была распределена более равномерно, можно включить гамма-коррекцию. Для

этого в классе LinearGradientBrush определен метод SetGammaCorrection:

```
Status SetGammaCorrection(BOOL useGammaCorrection);
```

Параметр *useGammaCorrection* определяет, будет ли включена гамма-коррекция или нет. Если значение этого параметра равно TRUE, гамма-коррекция будет включена. По умолчанию гамма-коррекция отключена.

На рис. 1.10 представлены два линейных градиента: в одном гамма-коррекция отключена (*а*), а в другом – включена (*б*).

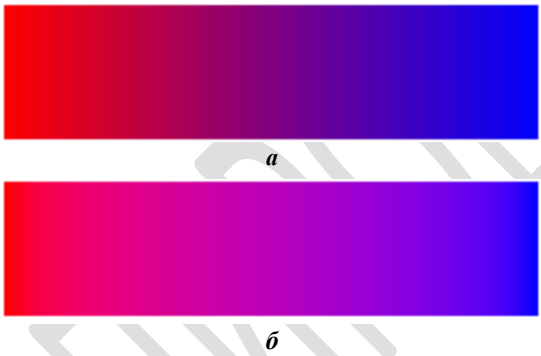


Рис. 1.10. Линейный градиент:
а – без гамма-коррекции; *б* – с гамма-коррекцией

Определить включена или нет гамма-коррекция можно с помощью метода GetGammaCorrection:

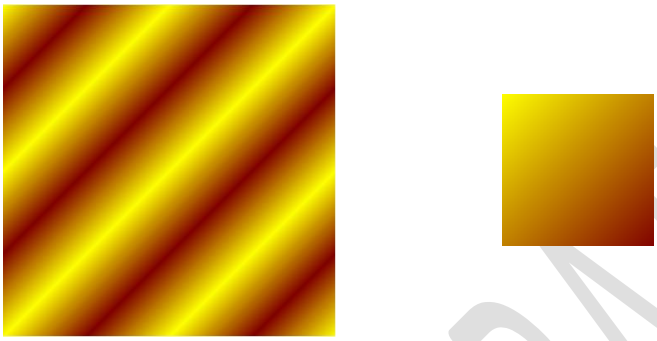
```
BOOL GetGammaCorrection() const;
```

Если гамма-коррекция включена, метод GetGammaCorrection возвращает значение TRUE, и FALSE – в противном случае.

Класс LinearGradientBrush (подобно классу TextureBrush) позволяет определить режим заполнения, когда размер линейного градиента меньше заполняемой области. Для этого у него есть два метода – GetWrapMode и SetWrapMode, которые, соответственно, возвращает и задает режим заполнения. Эти методы имеют следующие прототипы:

```
WrapMode GetWrapMode() const;
Status SetWrapMode(WrapMode wrapMode);
```

На рис. 1.11, *а* показан пример мозаичного заполнения прямоугольника линейным градиентом, представленным на рис. 1.11, *б*.



а *б*

Рис. 1.11. Мозаичное заполнение прямоугольника (*а*) линейным градиентом (*б*)

Также класс `LinearGradientBrush` позволяет определить границы линейного градиента с помощью метода `GetRectangle`:

```
Status GetRectangle(Rect *rect) const;
Status GetRectangle(RectF *rect) const;
```

Параметр *`rect`* указывает на объект класса `Rect` (или `RectF`), в который будет сохранен прямоугольник, определяющий границы линейного градиента.

Кисти градиента контура

Кисть градиента контура (класс `PathGradientBrush`) используется для заполнения замкнутой фигуры цветовым градиентом, в котором цвет меняется от центральной точки наружу до границы, определяемой замкнутым контуром.

Класс `PathGradientBrush` определен в заголовочном файле gdipluspath.h и имеет следующие конструкторы:

```
PathGradientBrush(const GraphicsPath *path);
PathGradientBrush(const Point *points, INT count,
    WrapMode wrapMode = WrapModeClamp);
PathGradientBrush(const Point *points, INT count,
    WrapMode wrapMode = WrapModeClamp);
```

Первый конструктор создает объект класса `PathGradientBrush` на основе объекта класса `GraphicsPath`, который в GDI+ используется для работы с контурами (подробнее см. в разделе «Контуры»). Второй и третий конструкторы создают объект класса `PathGradientBrush` на основе массива точек, на который указывает параметр *`points`*, а параметр

count задает количество элементов в этом массиве. Параметр *wrapMode* задает режим заполнения мозаичного заполнения. По умолчанию заполнение не производится.

В следующем примере продемонстрировано создание кисти градиента контура на основе контура треугольной формы.

Листинг 1.5. Пример создания кисти градиента контура

```
1   // вершины треугольника
2   Point points[3] = {
3       Point(100, 0),
4       Point(200, 200),
5       Point(0, 200)};
6
7   // создаем кисть градиента контура
8   PathGradientBrush pthGrBrush(points, 3);
```

Рассмотрим на основе этого примера некоторые методы класса PathGradientBrush, которые могут пригодиться при работе кистями градиента контура. Полный перечень и подробное описание методов класса PathGradientBrush см. в документации Platform SDK.

Класс PathGradientBrush позволяет определять отдельные цвета для центра, границы и даже для каждой точки на границе контура. Для работы с цветом центра контура в классе PathGradientBrush определены два метода – GetCenterColor и SetCenterColor:

```
Status GetCenterColor(Color *color) const;

Status SetCenterColor(const Color &color);
```

Метод GetCenterColor сохраняет текущее значение цвета центра контура в объект класса Color, на который указывает параметр *color*, а метод SetCenterColor задает новое значение цвета для центра контура.

Для задания цвета границы контура в классе PathGradientBrush определен метод SetSurroundColors:

```
Status SetSurroundColors(const Color *colors, INT *count);
```

Первый параметр, *colors*, указывает на массив объектов класса Color, каждый из которых хранит значение цветов.

Второй параметр, *count*, указывает на переменную типа int, которая должна содержать количество задаваемых цветов. Если метод SetSurroundColors будет выполнен успешно, в переменную, на которую указывает этот параметр, будет сохранено количество заданных цветов.

В листинге 1.6 продемонстрирован пример задания отдельных цветов для центра и точек на границе контура для кисти, созданной в

предыдущем примере (см. листинг 1.5). Полученный результат показан на рис. 1.12, *а*.

Листинг 1.6. Пример задания цветов для кисти градиента контура

```
// задаем цвет центра контура
pthGrBrush.SetCenterColor(Color::Red);

// задаем цвета для каждой точки на границе контура...

Color colors[3] = {
    Color::DarkGreen, Color::Aqua, Color::Blue};

int n = 3;
pthGrBrush.SetSurroundColors(colors, &n);
```

Следующий пример демонстрирует задание цветов для центра и для всей границы контура для кисти из примера в листинге 1.5. Результат представлен на рис. 1.12, *б*.

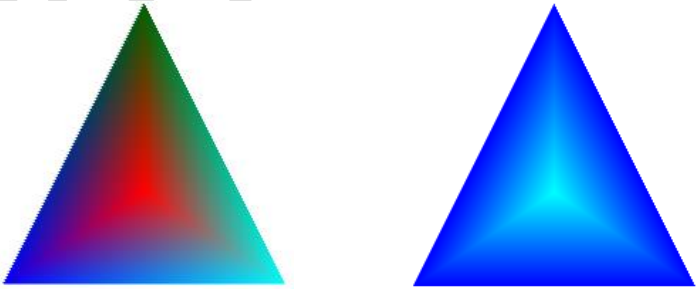
Листинг 1.7. Еще один пример задания цветов для кисти градиента контура

```
// задаем цвет центра контура
pthGrBrush.SetCenterColor(Color::Aqua);

// задаем цвет границы контура...

Color color = Color::Blue;

int n = 1;
pthGrBrush.SetSurroundColors(&color, &n);
```



а ***б***

Рис. 1.12. Различные варианты градиента контура:
а – каждая точка на границе имеет свой цвет; *б* – общий цвет для всей границы

Определить количество установленных цветов в градиенте контура можно с помощью метода GetSurroundColorCount, а получить значения этих цветов можно с помощью метода GetSurroundColors. Прототипы этих методов записываются следующим образом:

```
INT GetSurroundColorCount() const;

Status GetSurroundColors(Color *colors, INT *count) const;
```

Первый параметр, *colors*, указывает на массив объектов класса Color, в который будут сохранены значения цветов.

Второй параметр, *count*, указывает на переменную типа int, которая должна содержать количество запрашиваемых цветов. Если метод GetSurroundColors будет выполнен успешно, в переменную, на которую указывает этот параметр, будет сохранено количество полученных цветов.

Цвет центра используется не только в центральной точке, но и везде в пределах внутреннего контура. Класс PathGradientBrush позволяет настраивать внутренний контур, увеличивая размер фокуса с помощью метода SetFocusScales:

```
Status SetFocusScales(REAL xScale, REAL yScale);
```

Параметры *xScale* и *yScale* задают коэффициенты масштабирования, соответственно по шкале x и y.

На рис. 1.13 показан эффект применения коэффициентов масштабирования к фокусу кисти градиента контура, который можно получить, если добавить в предыдущий пример (см. листинг 1.7) следующий программный код:

```
1   // применим коэффициенты масштабирования
2   pthGrBrush.SetFocusScales(0.3f, 0.7f);
```

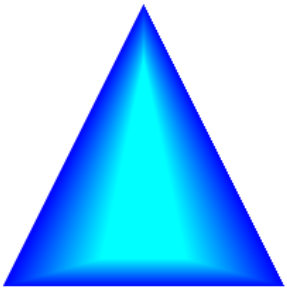


Рис. 1.13. Масштабирование фокуса кисти градиента контура

Чтобы получить установленные коэффициенты масштабирования фокуса нужно использовать метод `GetFocusScales`:

```
Status GetFocusScales(REAL *xScale, REAL *yScale) const;
```

Параметры *xScale* и *yScale* указывают на переменные типа `float`, в которые будут записаны коэффициенты масштабирования, соответственно по шкале x и y.

По умолчанию центральной точкой кисти градиента контура является центр контура, используемого при создании этой кисти. Положение центральной точки можно изменить с помощью метода `SetCenterPoint` класса `PathGradientBrush`:

```
Status SetCenterPoint(const Point &point);
Status SetCenterPoint(const PointF &point);
```

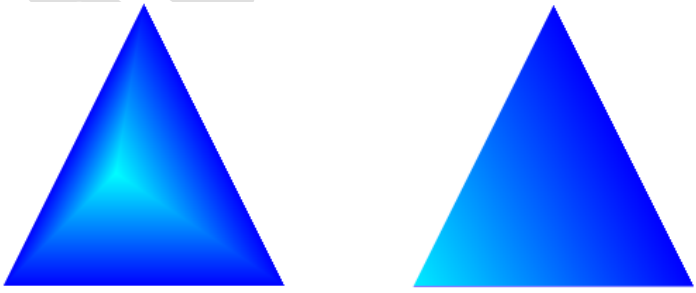
Параметр *point* ссылается на объект класса `Point` (или `PointF`), в котором представлены координаты новой центральной точки.

На рис. 1.14, *а* показан результат изменения положения центральной точки, который можно получить, если добавить в пример из листинга 1.7 следующий программный код:

```
1   // изменяем положение центральной точки
2   pthGrBrush.SetCenterPoint( Point(80,120) );
```

В качестве центральной точки кисти градиента контура можно установить точку, которая лежит вне контура. На рис. 1.14, *б* показан результат задания центральной точки вне контура, представленного в следующем фрагменте программного кода:

```
1   // изменяем положение центральной точки
2   pthGrBrush.SetCenterPoint( Point(-10,210) );
```



а ***б***

Рис. 1.14. Изменение положения центральной точки кисти градиента контура: *а* – точка внутри контура; *б* – точка вне контура

Определить текущее положение центральной точки можно с помощью метода GetCenterPoint:

```
Status GetCenterPoint(Point *point) const;
Status GetCenterPoint(PointF *point) const;
```

Параметр *point* указывает на объект класса Point (или PointF), в который будут сохранены координаты центральной точки.

Нужно отметить, что в классе PathGradientBrush также имеются методы аналогичные тем, что были уже рассмотрены в классе LinearGradientBrush. Вот эти методы:

- GetBlend;
- GetBlendCount;
- GetGammaCorrection;
- GetInterpolationColorCount;
- GetInterpolationColors;
- GetRectangle;
- GetWrapMode;
- SetBlend;
- SetGammaCorrection;
- SetInterpolationColors;
- SetWrapMode.

Конечно же, перечисленные методы класса PathGradientBrush дают несколько другой эффект чем аналогичные методы класса LinearGradientBrush. В листинге 1.8 показан пример использования метода SetInterpolationColors для того, чтобы изменить созданную кисть градиента контура (см. листинг 1.5), как показано на рис. 1.15.

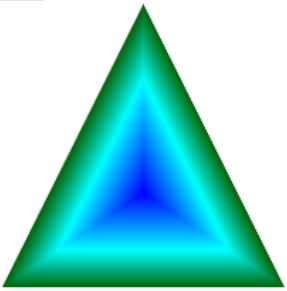


Рис. 1.15. Градиент контура, определяемый тремя цветами

Листинг 1.8. Создание кисти с неоднородным изменением цвета

```
// изменяем параметры кисти градиента контура...

Color colors[3] = {
    Color::DarkGreen, Color::Aqua, Color::Blue};

float pos[3] = {
    0.0f,  // начало градиента
    0.4f,  // 40% от начала градиента до его конца
    1.0f}; // конец градиента

pthGrBrush.SetInterpolationColors(colors, pos, 3);
```

Класс Brush

Как уже было сказано ранее класс `Brush` является родительским для классов `SolidBrush`, `HatchBrush`, `TextureBrush`, `LinearGradientBrush` и `PathGradientBrush`. Однако в отличие от своих потомков сам по себе класс `Brush` практически бесполезен, так как его нельзя использовать непосредственно.

Несмотря на это, различные классы GDI+ в своих методах, выполняющих заполнение замкнутой фигуры, требуют указывать в качестве аргумента именно объект класса `Brush`. Это сделано только для того, чтобы такие методы могли работать идентично вне зависимости от кисти, используемой для заполнения. Дело в том, что все классы для работы кистями, являясь производными от класса `Brush`, могут использоваться в качестве аргумента в методах, которые требуют объект класса `Brush`.

Следует также отметить, что класс `Brush` имеет несколько методов, которые наследуют все потомки этого класса. Среди этих методов есть уже рассмотренный ранее метод `GetLastStatus`.

В классе также есть метод `Clone`, который клонирует текущий объект класса производного от класса `Brush`. Метод `Clone` имеет следующий прототип:

```
Brush* Clone() const;
```

В случае успеха этот метод возвращает указатель на объект класса `Brush`, который должен быть удален вызовом оператора `delete` после того как станет не нужен. В случае ошибки – `NULL`.

В листинге 1.9 продемонстрирован пример использования метода `Clone` для клонирования объекта класса `SolidBrush`. При этом если аргумент *`brush`* указывает не на объект класса `SolidBrush`, клонирование не выполняется.

Листинг 1.9. Функция клонирования объекта класса `SolidBrush`

```
SolidBrush* CloneSolidBrush(Brush *brush)
{
    SolidBrush *result = dynamic_cast<SolidBrush *>(brush);

    if (NULL != result)
        result = (SolidBrush *)result->Clone();

    return result;
} // CloneSolidBrush
```

Функцию из предыдущего примера можно использовать следующим образом:

```
SolidBrush *cloneBrush = NULL;

// создаем сплошную кисть
SolidBrush solidBrush(Color::Red);
// создаем штриховую кисть
HatchBrush hatchBrush(HatchStyleCross, Color::Blue);

// клонируем объект solidBrush
// результат - указатель новый экземпляр класса SolidBrush
cloneBrush = CloneSolidBrush(&solidBrush);

// удаляем клон
delete cloneBrush, cloneBrush = NULL;

// клонируем объект hatchBrush
// результат - NULL
cloneBrush = CloneSolidBrush(&hatchBrush);
```

Определить тип кисти можно и без использования оператора `dynamic_cast`. В классе `Brush` есть метод `GetType`, который возвращает тип кисти. Прототип метода `GetType` имеет следующий вид:

```
BrushType GetType() const;
```

Этот метод возвращает одно из следующих значений перечисляемого типа `BrushType`:

- `BrushTypeSolidColor` – сплошная кисть;
- `BrushTypeHatchFill` – штриховая кисть;
- `BrushTypeTextureFill` – текстурная кисть;
- `BrushTypePathGradient` – кисть градиента контура;
- `BrushTypeLinearGradient` – кисть линейного градиента.

В следующем примере продемонстрирована слегка видоизмененная функция из листинга 1.9, которая клонирует объект класса SolidBrush:

Листинг 1.10. Функция клонирования объекта класса SolidBrush

```
SolidBrush* CloneSolidBrush(Brush *brush)
{
    BrushType type = brush->GetType();

    if (type == BrushTypeSolidColor)
        return (SolidBrush *)brush->Clone();

    return NULL;
} // CloneSolidBrush
```

Перья

Для рисования линий, кривых и границ (контуров) различных фигур используется *перо* (*pen*), которое определяет графические атрибуты такие, как цвет, толщина, стиль штриха и т.п. Работа с перьями в GDI+ осуществляется с помощью класса Pen, который определен в заголовочном файле gdipluspen.h.

Класс Pen имеет следующие конструкторы:

```
Pen(const Color &color, REAL width = 1.0f);
Pen(const Brush *brush, REAL width = 1.0f);
```

Первый конструктор создает объект класса Pen на основе объекта класса Color. Второй конструктор создает объект класса Pen на основе объекта класса Brush. Параметр *width* задает толщину пера в логических единицах.

В следующем примере показаны два способа создания пера красного цвета и толщиной 4.

```
// способ первый:

// просто создаем объект класса Pen
Pen pen1(Color(255, 0, 0), 4.f);

// способ второй:

// сперва создаем сплошную кисть
SolidBrush sBrush(Color::Red); // красный цвет
// а затем создаем объект класса Pen на основе объекта sBrush
Pen pen2(&sBrush, 4.f);
```

Прежде чем перейти к изучению основных методов класса Pen, следует отметить, что в этом классе (как и в классе Brush) есть методы GetLastStatus и Clone. Метод Clone имеет следующий прототип:

```
Pen* Clone() const;
```

В случае успеха метод Clone возвращает указатель на объект класса Pen, который должен быть удален вызовом оператора delete после того как станет не нужен. В случае ошибки – NULL.

Цвет пера

Цвет пера определяет цвет рисуемых линий. В классе Pen определены два метода – GetColor и SetColor, которые работают с цветом пера. Эти методы имеют следующие прототипы:

```
Status GetColor(Color *color) const;
Status SetColor(const Color &color);
```

Метод GetColor сохраняет текущее значение цвета пера в объект класса Color, на который указывает параметр *color*, а метод SetColor задает новое значение цвета пера.

Толщина и выравнивание пера

Толщину пера можно указать в качестве одного из параметров конструктора класса Pen при создании объекта это класса. Можно также изменять толщину пера с помощью метода SetWidth:

```
Status SetWidth(REAL width);
```

Параметр *width* задает новую толщину пера. Определить текущую толщину пера можно с помощью метода GetWidth класса Pen:

```
REAL GetWidth() const;
```

Если толщина пера больше 1.0, при рисовании замкнутого контура можно указать положение такого пера относительно рисуемого контура. Для этого используется метод SetAlignment класса Pen:

```
Status SetAlignment(PenAlignment penAlignment);
```

Параметр *penAlignment* задает положение пера. Этот параметр может принимать одно из значений перечисляемого типа PenAlignment:

- PenAlignmentCenter – перо располагается по центру контура рисуемой фигуры;
- PenAlignmentInset – перо располагается с внутренней стороны контура рисуемой фигуры.

На рис. 1.16 показаны различные варианты положения пера при рисовании окружности.

Рис. 1.16. Положение пера при рисовании окружности

По умолчанию перо имеет положение по центру контура. Определить текущее положение пера относительно контура можно с помощью метода `GetAlignment` класса `Pen`:

```
PenAlignment GetAlignment() const;
```

Стиль штриха

Стиль штриха определяет, каким образом будут рисоваться различные линии и контуры. Например, они могут рисоваться непрерывно или пунктиром, или еще как-нибудь. В классе `Pen` есть метод `SetDashStyle`, который задает стиль штриха. Этот метод имеет следующий прототип:

```
Status SetDashStyle(DashStyle dashStyle);
```

Параметр *dashStyle* определяет новый стиль штриха. Этот параметр должен принимать одно из значений перечисляемого типа `DashStyle`:

- `DashStyleSolid` – сплошная линия;
- `DashStyleDash` – обычный штрих;
- `DashStyleDot` – точки;
- `DashStyleDashDot` – штрих-пунктир;
- `DashStyleDashDotDot` – штрих-две точки.

На рис. 1.17 представлены варианты линий, нарисованных с различными стилями штриха.

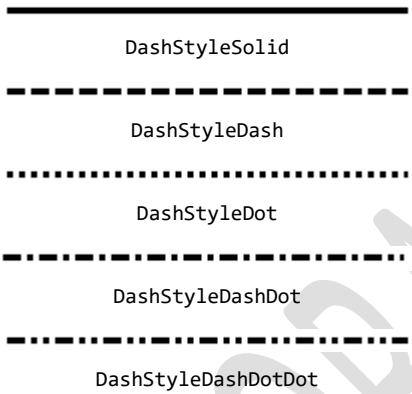


Рис. 1.17. Линии с различными стилями штриха

По умолчанию перо рисует сплошную линию. Определить текущий стиль штриха можно с помощью метода `GetDashStyle` класса `Pen`:

```
DashStyle GetDashStyle() const;
```

Если стандартные стили штрихов не подходят, можно использовать собственный шаблон штриха. В классе `Pen` есть метод `SetDashPattern`, который задает шаблон штриха. Этот метод имеет следующий прототип:

```
Status SetDashPattern(const REAL *dashArray, INT count);
```

Первый параметр, *dashArray*, указывает на массив значений типа *float*, каждое из которых определяет длину штрихов и пробелов между ними. Все значения в этом массиве должны быть больше нуля.

Второй параметр, *count*, определяет количество элементов в массиве, на который указывает параметр *dashArray*.

В следующем примере показано, как можно задавать шаблон штриха с помощью метода `SetDashPattern` класса `Pen`.

Листинг 1.11. Пример задания шаблона штриха

```
1  // массив штрихов и пробелов
2  float dash[8] = {
3        6.f,  // штрих длиной 6
4        3.f,  // пробел длиной 3
5        1.f,  // штрих длиной 1
6        2.f,  // пробел длиной 2
7        1.f,  // штрих длиной 1
8        2.f,  // пробел длиной 2
```

```
9       1.f,  // штрих длиной 1
10      3.f}; // пробел длиной 3
11
12  // создаем перо черного цвета и толщиной 2
13  Pen pen(Color::Black, 2.f);
14
15  // задаем шаблон штриха
16  pen.SetDashPattern(dash, 8);
```

В приведенном примере шаблон штриха создается на основе массива {6, 3, 1, 2, 1, 2, 1, 3}. Умножив элементы этого массива на толщину пера, равную 2, получим массив {12, 6, 2, 4, 2, 4, 2, 6}. Получившиеся штрихи будут иметь длину 12 и 2, а длина промежутков между ними будет равна 6 или 4. Нарисованная с помощью такого пера линия показана на рис. 1.18.

Рис. 1.18. Пример штриховой линии

Если для пера установлен шаблон штриха, определить количество элементов в массиве, задающем длину штрихов и пробелов, можно с помощью метода GetDashPatternCount класса Pen, а получить значения этого массива можно с помощью метода GetDashPattern.

```
INT GetDashPatternCount() const;

Status GetDashPattern(REAL *dashArray, INT count) const;
```

Первый параметр, *dashArray*, указывает на массив типа float, в который будут сохранены длины штрихов и пробелов в установленном шаблоне штриха.

Второй параметр, *count*, задает количество элементов в массиве, на который указывает параметр *dashArray*.

Стиль штриха определяет не только длину штрихов, но и форму их концов. В классе Pen есть метод SetDashCap, который указывает форму обоих концов каждого штриха. Метод SetDashCap имеет следующий прототип:

```
Status SetDashCap(DashCap dashCap);
```

Параметр *dashCap* задает форму концов штриха. Этот параметр должен принимать одно из значений перечисляемого типа DashCap:

- DashCapFlat – прямоугольная форма;
- DashCapRound – круглая форма;
- DashCapTriangle – треугольная форма.

На рис. 1.19 представлены линии, нарисованные с различными концами штриха.

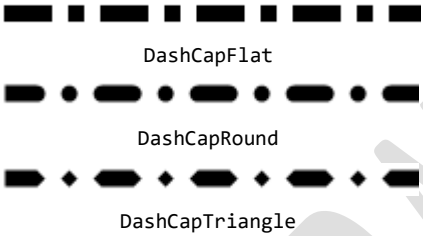


Рис. 1.19. Штрихованные линии с различными формами концов штриха

По умолчанию перо использует прямоугольную форму для обоих концов штриха. Определить текущую форму конца штриха можно с помощью метода `GetDashCap` класса `Pen`:

```
DashCap GetDashCap() const;
```

При рисовании линий также можно указать расстояние от начала линии до начала штриха. На рис. 1.20, *а* показана линия, для которой расстояние от начала линии до начала штриха равно `0`, а на рис. 1.20, *б* – линия, для которой такое расстояние равно `3`. При этом в обоих случаях используется шаблон штриха из примера в листинге 1.11.

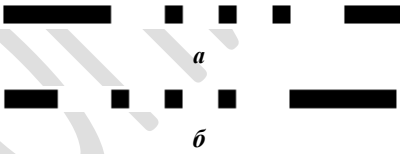


Рис. 1.20. Пример штриховой линии, в которой штрих начинается: *а* – с начала; *б* – с небольшим смещением

В классе Pen есть два метода – `GetDashOffset` и `SetDashOffset`. Метод `GetDashOffset` возвращает текущее расстояние от начала линии до начала штриха, а метод `SetDashOffset` изменяет это расстояние.

```
REAL GetDashOffset() const;
Status SetDashOffset(REAL dashOffset);
```

Соединение и концы линий

Соединение линий – это область, образуемая двумя линиями с соприкасающимися концами. Стиль соединения линий является атрибутом. После задания стиля соединения линий для объекта класса `Pen`

этот стиль будет применяться ко всем соединенным линиям, для рисования которых используется данный объект.

Для задания стиля соединения линий в классе `Pen` служит метод `SetLineJoin`:

```
Status SetLineJoin(LineJoin lineJoin);
```

Параметр *`lineJoin`* задает новый стиль соединения линий. Этот параметр должен принимать одно из значений перечисляемого типа `LineJoin`:

- `LineJoinMiter` – угловое соединение со скосом в 45°. Получается острый (рис. 1.21, *а*) или обрезанный угол (рис. 1.21, *б*) в зависимости от того, превышает ли длина среза ограничение по срезу;
- `LineJoinBevel` – скошенное соединение. Получается угол при диагонали;
- `LineJoinRound` – круговое соединение. Получается ровная круговая дуга между двумя линиями;
- `LineJoinMiterClipped` – угловое соединение со скосом в 45°. Получается острый (рис. 1.21, *а*) или срезанный угол (рис. 1.21, *б*) в зависимости от того, превышает ли длина среза ограничение по срезу.

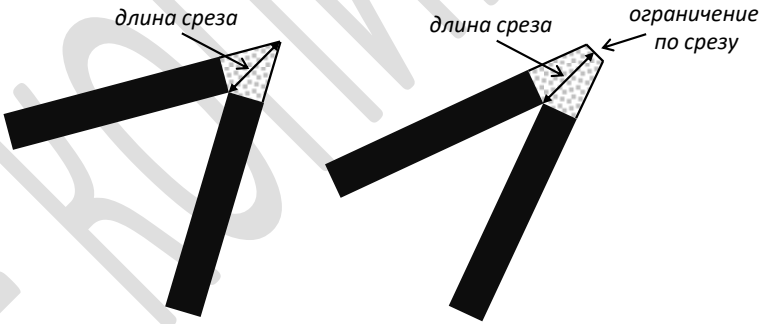


Рис. 1.21. Угловое соединение, в котором длина среза:
а – не превышает ограничение; *б* – превышает ограничение

По умолчанию перо использует угловое соединение. Определить текущий стиль соединения линий можно с помощью метода `GetLineJoin` класса `Pen`:

```
LineJoin GetLineJoin() const;
```

Для того чтобы установить ограничение по срезу следует использовать метод SetMiterLimit класса Pen:

```
Status SetMiterLimit(REAL miterLimit);
```

Параметр *miterLimit* задает новое ограничение по срезу. Если значение параметра *miterLimit* меньше 1, оно будет заменено на 1.

По умолчанию ограничение по срезу имеет значение 10. Определить текущее значение ограничения по срезу можно с помощью метода GetMiterLimit класса Pen:

```
REAL GetMiterLimit() const;
```

Еще одним атрибутом является форма концов линии. Для указания формы концов линий в классе Pen есть два метода – SetStartCap и SetEndCap. Метод SetStartCap задает форму начала линии, а метод SetEndCap – форму окончания линии. Прототипы этих методов имеют следующий вид:

```
Status SetStartCap(LineCap startCap);
Status SetEndCap(LineCap endCap);
```

Параметр *startCap* (*endCap*) определяет форму начала (окончания) линии. Этот параметр должен принимать одно из значений перечисляемого типа LineCap:

- LineCapFlat – прямоугольная форма;
- LineCapSquare – квадратная форма;
- LineCapRound – круглая форма;
- LineCapTriangle – треугольная форма;
- LineCapNoAnchor – без маркера;
- LineCapSquareAnchor – квадратный маркер;
- LineCapRoundAnchor – круглый маркер;
- LineCapDiamondAnchor – маркер в форме ромба;
- LineCapArrowAnchor – маркер в форме стрелки.

По умолчанию на обоих концах перо использует плоское завершение. Определить текущую форму можно с помощью двух методов класса Pen – GetStartCap и GetEndCap. Метод GetStartCap определяет форму начала линии, а метод GetEndCap – форму окончания линии.

```
LineCap GetStartCap() const;
LineCap GetEndCap() const;
```

На рис. 1.22 показаны различные стили соединений в сочетании с разными формами начала и окончания ломаной линии.

Рис. 1.22. Ломаные линии с различными стилями соединений в сочетании с разными формами начала и окончания

Так же стоит обратить внимание на рис. 1.23, на котором представлены варианты линий, на обоих концах которых нарисованы специальные фигуры, называемые маркерами.

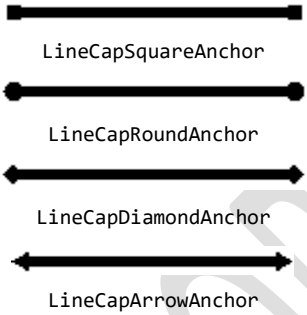


Рис. 1.23. Линии с различными типами маркеров

Нужно отметить, что в классе `Pen` имеется метод `SetLineCap`, который позволяет сразу задать форму для обоих концов линии и форму для концов штриха. Метод имеет `SetLineCap` следующий прототип:

```
Status SetLineCap(LineCap startCap, LineCap endCap,
    DashCap dashCap);
```

Параметры *`startCap`* и *`endCap`* определяют форму, соответственно, начала и окончания линии. Эти параметры должны принимать одно из значений перечисляемого типа `LineCap`.

Последний параметр, *`dashCap`*, определяет форму обоих концов каждого штриха. Этот параметр должен принимать одно из значений перечисляемого типа `DashCap`.

Составные перья

Составное перо рисует составную (сложную) линию, состоящую из параллельных линий и разделяющих их промежутков.

Класс `Pen` позволяет делать из обычного пера составное перо. Для этого в классе `Pen` есть метод `SetCompoundArray`:

```
Status SetCompoundArray(const REAL *compoundArray, INT count);
```

Первый параметр, *`compoundArray`*, указывает на массив значений типа `float`, определяющих составное перо. Значения в массиве должны быть в порядке возрастания и находиться в диапазоне от `0.0` до `1.0`.

Второй параметр, *`count`*, определяет количество элементов в массиве, на который указывает параметр *`compoundArray`*.

В листинге 1.12 приведен пример создания составного пера, которым нарисованы контуры для фигур, изображенных на рис. 1.24.

Листинг 1.12. Пример создания составного пера

```
// создаем перо черного цвета и толщиной 10
Gdiplus::Pen pen(Color::Black, 10.f);

// массив, определяющий составное перо
float comp[6] = {
    0.0f, 0.2f,  // 1-я линия: от 0% до 20% от толщины пера
    0.3f, 0.7f,  // 2-я линия: от 30% до 70% от толщины пера
    0.8f, 1.0f}; // 3-я линия: от 80% до 100% от толщины пера

pen.SetCompoundArray(comp, 6); // делаем из пера составное перо
```

Рис. 1.24. Рисование контуров составным пером

Если перо является составным, определить количество элементов в массиве, определяющем это составное перо, можно с помощью метода `GetCompoundArrayCount` класса `Pen`, а получить значения этого массива можно с помощью метода `GetCompoundArray`.

```
INT GetCompoundArrayCount() const;

Status GetCompoundArray(REAL *compoundArray, INT count) const;
```

Первый параметр, *`compoundArray`*, указывает на массив типа `float`, в который будут сохранены значения массив, определяющего это составного перо.

Второй параметр, *`count`*, задает количество элементов в массиве, на который указывает параметр *`compoundArray`*.

Перья с текстурным заполнением

Вместо рисования линий сплошным цветом можно нарисовать их с текстурной заливкой, например, как на рис. 1.25. Для рисования линий с текстурной заливкой необходимо создать объект класса `Texture-`

Brush, определяющего текстурную кисть, и передать этот объект конструктору класса Pen при создании пера.

Рис. 1.25. Рисование пером с текстурным наполнением

Шрифты

Для отображения текста используется *шрифт* (*font*), который определяет форму букв отображаемого текста. Работа с шрифтами в GDI+ осуществляется с помощью класса Font, который определен в заголовочном файле gdiplusheaders.h.

Класс Font имеет следующие конструкторы:

```
Font(HDC hdc);

Font(HDC hdc, const LOGFONTA *logfont);

Font(HDC hdc, const LOGFONTW *logfont);

Font(HDC hdc, const HFONT hfont);

Font(const FontFamily *family, REAL emSize,
    INT style = FontStyleRegular, Unit unit = UnitPoint);

Font(const WCHAR *familyName, REAL emSize,
    INT style = FontStyleRegular, Unit unit = UnitPoint,
    const FontCollection *fontCollection = NULL);
```

Первый конструктор создает объект класса Font на основе дескриптора контекста устройства (подробнее см. в разделе «Контекст устройства»), на который указывает параметр *hdc*.

Второй и третий конструкторы создают объект класса Font на основе дескриптора контекста устройства и структуры LOGFONT, на которую указывает параметр *logfont*. Структура LOGFONT используется в библиотеке GDI для создания шрифтов. Подробную информацию о структуре LOGFONT см. в документации Platform SDK.

Четвертый конструктор создает объект класса Font на основе дескриптора контекста устройства и дескриптора шрифта, на который указывает параметр *hfont*. Дескриптор шрифта возвращают функции библиотеки GDI, которые создают шрифт, – CreateFont и CreateFontIndirect. Подробное описание этих функций можно найти в документации Platform SDK.

Пятый и шестой конструкторы создают объект класса `Font` на основе имени гарнитуры шрифта, например, одними из известных гарнитур являются Arial, Times New Roman и Courier New. В пятом конструкторе имя гарнитуры определяется объектом класса `FontFamily` (подробнее см. в документации Platform SDK), на который указывает параметр *`family`*, а в шестом – строкой в формате Unicode, на которую указывает *`familyName`*.

Параметр *`emSize`* задает размер шрифта в единицах измерения, задаваемых параметром *`unit`*. Параметр *`unit`* должен принимать одно из значений перечисляемого типа `Unit` (подробнее см. в разделе «Логические единицы»).

Параметр *`style`* задет стиль шрифта. Этот параметр должен принимать одно (или комбинацию) из значений перечисляемого типа `FontStyle`:

- `FontStyleRegular` – обычный текст;
- `FontStyleBold` – полужирный текст;
- `FontStyleItalic` – курсивный текст;
- `FontStyleBoldItalic` – полужирный и курсивный текст;
- `FontStyleUnderline` – подчеркнутый текст;
- `FontStyleStrikeout` – перечеркнутый текст.

Параметр *`fontCollection`* указывает на объект класса `FontCollection`. Класс `FontCollection` используется в GDI+ для представления набора шрифтов, из которого будет выбираться шрифт, подходящий под задаваемые параметры, при создании объекта класса `Font` (подробнее см. в документации Platform SDK). Если этот параметр установлен в `NULL`, шрифт будет выбираться из набора шрифтов, установленных в операционной системе.

В следующем примере показаны два способа создания шрифта Times New Roman размером 14 пт., который соответствует курсивному и перечеркнутому тексту.

Листинг 1.13. Пример создания шрифта

```
1  // способ первый: просто создаем объект класса Font
2  Font font1(L"Times New Roman", 14.f,
3      FontStyleItalic|FontStyleUnderline);
4
5  // способ второй: сперва объект класса FontFamily,
6  FontFamily family(L"Times New Roman");
7  // а затем объект класса Font
8  Font font2(&family, 14.f, FontStyleItalic|FontStyleUnderline);
```

На рис. 1.26 показаны различные варианты гарнитур Arial, Times New Roman и Courier New в сочетании с различными стилями.

Рис. 1.26. Варианты шрифтов с различными стилями

В классе `Font` определено несколько методов, которые могут быть полезны при работе с объектами этого класса. Основные методы этого класса перечислены в табл. 1.8. Подробное описание методов класса `Font` можно найти в документации Platform SDK.

Таблица 1.8. Методы класса `Font`

Метод	Описание
`Font* Clone() const;`	Клонирует текущий объект класса `Font`. Возвращает указатель на созданный объект класса `Font`, который должен быть удален вызовом оператора `delete` после того как станет не нужен
`Status GetFamily(` `  FontFamily *family) const;`	Возвращает имя гарнитуры шрифта. Результат сохраняется в объекте класса `FontFamily`, на который указывает параметр *`family`*

`REAL GetHeight(REAL dpi) const;`
`REAL GetHeight(`
`  const Graphics *graphics) const;`
	Возвращает значение междустрочного интервала шрифта

`Status GetLastStatus() const;`	Возвращает значение, указывающее на причину возникновения ошибки (или ее отсутствие) в последнем вызове одного из методов класса `Font`

`Status GetLogFontA(`
`  const Graphics *g,`
`  LOGFONTA *logfontA) const;`
`Status GetLogFontW(`
`  const Graphics *g,`
`  LOGFONTW *logfontW) const;`
	Возвращает информацию о шрифте. Результат сохраняется в структуру `LOGFONTA` (`LOGFONTW`), на которую указывает параметр *`logfontA`* (*`logfontW`*)

`REAL GetSize() const;`	Возвращает размер шрифта
`INT GetStyle() const;`	Возвращает стиль шрифта
`Unit GetUnit() const;`	Возвращает единицы изменения, в которых задается размер шрифта
`BOOL IsAvailable() const;`	Возвращает `TRUE`, если объект класса `Font` был успешно создан, и `FALSE` – в противном случае

Изображения

Для работы с изображениями в GDI+ имеется класс `Image` и два его потомка `Metafile` и `Bitmap`. Все эти классы определены в заголовочном файле gdiplusheaders.h.

Векторные изображения

Для работы с векторным изображением в GDI+ используется класс `Metafile`, который является потомком класса `Image`.

Класс `Metafile` имеет следующий конструктор:

```
Metafile(const WCHAR *filename);
```

Этот конструктор создает объект класса `Metafile` на основе метафайла (metafile). Параметр *`filename`* указывает на Unicode-строку, содержащую имя метафайла.

В следующем небольшом примере показано, как можно создать объект класса `Metafile` из метафайла Sample.emf.

```
1   // загрузка из файла Sample.emf
2   Metafile mf(L"Sample.emf");
```

Помимо загрузки векторных изображений из файла, объект класса `Metafile` можно создавать и с помощью других конструкторов:

```
Metafile(HMETAFILE hWmf,
    const WmfPlaceableFileHeader *wmfPlaceableFileHeader,
    BOOL deleteWmf = FALSE);

Metafile(HENHMETAFILE hEmf, BOOL deleteEmf = FALSE);

Metafile(const WCHAR *filename,
    const WmfPlaceableFileHeader *wmfPlaceableFileHeader);

Metafile(IStream *stream);

Metafile(HDC referenceHdc, EmfType type = EmfTypeEmfPlusDual,
    const WCHAR *description = NULL);

Metafile(HDC referenceHdc, const RectF &frameRect,
    MetafileFrameUnit frameUnit = MetafileFrameUnitGdi,
    EmfType type = EmfTypeEmfPlusDual,
    const WCHAR *description = NULL);

Metafile(HDC referenceHdc, const Rect &frameRect,
    MetafileFrameUnit frameUnit = MetafileFrameUnitGdi,
    EmfType type = EmfTypeEmfPlusDual,
    const WCHAR *description = NULL);

Metafile(const WCHAR *fileName, HDC referenceHdc,
    EmfType type = EmfTypeEmfPlusDual,
    const WCHAR *description = NULL);

Metafile(const WCHAR *fileName, HDC referenceHdc,
    const RectF &frameRect,
    MetafileFrameUnit frameUnit = MetafileFrameUnitGdi,
```

```
        EmfType type = EmfTypeEmfPlusDual,
        const WCHAR *description = NULL);

    Metafile(const WCHAR *fileName, HDC referenceHdc,
        const Rect &frameRect,
        MetafileFrameUnit frameUnit = MetafileFrameUnitGdi,
        EmfType type = EmfTypeEmfPlusDual,
        const WCHAR *description = NULL);

    Metafile(IStream *stream, HDC referenceHdc,
        EmfType type = EmfTypeEmfPlusDual,
        const WCHAR *description = NULL);

    Metafile(IStream *stream, HDC referenceHdc,
        const RectF &frameRect,
        MetafileFrameUnit frameUnit = MetafileFrameUnitGdi,
        EmfType type = EmfTypeEmfPlusDual,
        const WCHAR *description = NULL);

    Metafile(IStream *stream, HDC referenceHdc,
        const Rect &frameRect,
        MetafileFrameUnit frameUnit = MetafileFrameUnitGdi,
        EmfType type = EmfTypeEmfPlusDual,
        const WCHAR *description = NULL);
```

Подробное описание этих конструкторов, а также всех методов класса `Metafile` можно найти в документации Platform SDK.

Растровые изображения

Для работы с растровым изображением в GDI+ используется класс `Bitmap`, который является потомком класса `Image`.

Класс `Bitmap` имеет следующие конструкторы:

```
Bitmap(const WCHAR *filename, BOOL useIcm = FALSE);

Bitmap(INT width, INT height,
    PixelFormat format = PixelFormat32bppARGB);
```

Первый конструктор создает объект класса `Bitmap` на основе растрового файла. Параметр *`filename`* указывает на Unicode-строку, содержащую имя файла, а параметр *`useIcm`* указывает, применяется ли цветокоррекция согласно информации об управлении цветом в файле с изображением. Если параметр *`useIcm`* имеет значение `FALSE`, цветокоррекция не применяется.

Второй конструктор создает объект класса `Bitmap` на основе параметров *`width`* и *`height`*, которые определяют, соответственно ширину и высоту растрового изображений. Параметр *`format`* задает формат ко-

дирования цвета в новом растровом изображении. Этот параметр должен принимать одно следующих значений:

- `PixelFormat1bppIndexed` – индексированные цвета, которые кодируются 1 битом (таблица цветов содержит два цвета);
- `PixelFormat4bppIndexed` – индексированные цвета, которые кодируются 4 битами (таблица цветов содержит 16 цветов);
- `PixelFormat8bppIndexed` – индексированные цвета, которые кодируются 8 битами (таблица цветов содержит 256 цветов);
- `PixelFormat16bppGrayScale` – оттенки серого, которые кодируются 16 битами (65 536 оттенков серого);
- `PixelFormat16bppRGB555` – цвет кодируется 16 битами: по 5 бит на красный, зеленый и синий компоненты модели RGB. Оставшийся бит не используется;
- `PixelFormat16bppRGB565` – цвет кодируется 16 битами: по 5 бит на красный и синий, и по 6 бит на зеленый компоненты модели RGB;
- `PixelFormat16bppARGB1555` – цвет кодируется 16 битами: по 5 бит на красный, зеленый и синий компоненты модели RGB. Оставшийся бит используется на альфа-фактор;
- `PixelFormat24bppRGB` – цвет кодируется 24 битами: по 8 бит на красный, зеленый и синий компоненты модели RGB;
- `PixelFormat32bppARGB` – цвет кодируется 32 битами: по 8 бит на красный, зеленый и синий компоненты модели RGB. Оставшиеся биты используются на альфа-фактор;
- `PixelFormat32bppRGB` – цвет кодируется 32 битами: по 8 бит на красный, зеленый и синий компоненты модели RGB. Оставшиеся биты не используются;
- `PixelFormat48bppRGB` – цвет кодируется 64 битами: по 16 бит на красный, зеленый и синий компоненты модели RGB. Оставшиеся биты не используются;
- `PixelFormat64bppARGB` – цвет кодируется 64 битами: по 16 бит на красный, зеленый и синий компоненты модели RGB. Оставшиеся биты используются на альфа-фактор.

В следующем примере показано, как можно создать объект класса `Bitmap` из файла Sample.png.

```
1   // загрузка из файла Sample.png
2   Bitmap bmp(L"Sample.png");
```

Кроме уже рассмотренных конструкторов класс Bitmap имеет еще и следующие конструкторы:

```
Bitmap(IStream *stream,
    BOOL useEmbeddedColorManagement = FALSE);

Bitmap(INT width, INT height, INT stride, PixelFormat format,
    BYTE *scan0);

Bitmap(INT width, INT height, Graphics *target);

Bitmap(IDirectDrawSurface7 *surface);

Bitmap(const BITMAPINFO *gdiBitmapInfo, VOID *gdiBitmapData);

Bitmap(HBITMAP hbm, HPALETTE hpal);

Bitmap(HICON hicon);

Bitmap(HINSTANCE hInstance, const WCHAR *bitmapName);
```

Подробное описание этих конструкторов и методов класса Bitmap можно найти в документации Platform SDK.

Для получения и установки цвета определенного пиксела в растровом изображении класс предоставляет методы GetPixel и SetPixel:

```
Status GetPixel(INT x, INT y, Color *color);

Status SetPixel(INT x, INT y, const Color &color);
```

Метод GetPixel сохраняет текущее значение цвета пикселя в объект класса Color, на который указывает параметр *color*, а метод SetPixel задает новое значение цвета пикселя. Параметры *x* и *y* задают координаты указанного пикселя.

Класс Image

Класс Image позволяет работать, как с векторным изображением, так и растровым.

Класс Image имеет следующие конструкторы:

```
Image(const WCHAR *filename,
    BOOL useEmbeddedColorManagement = FALSE);

Image(IStream *stream,
    BOOL useEmbeddedColorManagement = FALSE);
```

Первый конструктор создает объект класса Image на основе файла (векторного или растрового). Параметр *filename* указывает на Unicode-строку, содержащую имя файла. Второй конструктор создает объект класса Image на основе интерфейса IStream COM-объекта (см. в документации Platform SDK), на который указывает параметр *stream*. Параметр *useEmbeddedColorManagement* указывает, применяется ли цвето-

коррекция согласно информации об управлении цветом в файле с изображением. Если параметр *useEmbeddedColorManagement* имеет значение FALSE, цветокоррекция не применяется.

Рассмотрим некоторые методы класса Image, которые наследуют все потомки этого класса. Полный перечень и подробное описание методов класса Image см. в документации Platform SDK.

Среди методов класса Image есть методы GetLastStatus и Clone. Метод Clone имеет следующий прототип:

```
Image* Clone() const;
```

В случае успеха метод Clone возвращает указатель на объект класса Image, который должен быть удален вызовом оператора delete после того как станет не нужен. В случае ошибки – NULL.

Определить тип изображения можно с помощью метода GetType класса Image. Этот метод имеет следующий прототип:

```
ImageType GetType() const;
```

Этот метод возвращает одно из следующих значений перечисляемого типа ImageType:

- ImageTypeUnknown – неизвестный тип изображения;
- ImageTypeBitmap – растровое изображение;
- ImageTypeMetafile – векторное изображение.

Для определения формата кодирования цвета в изображении в классе Image есть метод GetPixelFormat:

```
PixelFormat GetPixelFormat() const;
```

Этот метод возвращает одно из значений перечисляемого типа PixelFormat (см. в разделе «Растровые изображения»).

Ширину и высоту изображения можно определить, соответственно, с помощью методов GetWidth и GetHeight класса Image:

```
UINT GetWidth();
UINT GetHeight();
```

В классе Image также есть два статических метода FromFile и FromStream, которые создают объект класса Image аналогично его конструкторам:

```
static Image* FromFile(const WCHAR *filename,
    BOOL useEmbeddedColorManagement = FALSE);
static Image* FromStream(IStream *stream,
    BOOL useEmbeddedColorManagement = FALSE);
```

В случае успеха методы `FromFile` и `FromStream` возвращают указатель на объект класса `Image`, который должен быть удален вызовом оператора `delete` после того как станет не нужен. В случае ошибки – `NULL`.

Следующий небольшой пример демонстрирует загрузку растрового изображения из файла с помощью метода `FromFile` класса `Image`:

```
// загрузка из файла Sample.png
Image *image = Image::FromFile(L"Sample.png");

if (NULL != image)
{
    /* файл успешно загружен */
} // if
```

Метод `FromStream` класса `Image` можно, например, использовать для загрузки изображений из ресурсов приложения Windows, как показано в примере из листинга 1.14.

Листинг 1.14. Функция загрузки изображения из ресурсов приложения

```
Image* ImageFromResource(HINSTANCE hInstance, LPCTSTR szResName,
LPCTSTR szResType)
{
    // поиск ресурса указанного типа
    HRSRC hRsrc = FindResource(hInstance, szResName, szResType);
    if (NULL == hRsrc) return NULL;

    // загрузка ресурса
    HGLOBAL hResource = LoadResource(hInstance, hRsrc);
    if (NULL == hResource) return NULL;

    // определяем размера ресурса
    DWORD cbImage = SizeofResource(hInstance, hRsrc);
    // выделение памяти для буфера
    HGLOBAL hData = GlobalAlloc(GMEM_FIXED, cbImage);

    if (NULL != hData)
    {
        // копируем изображение в буфер
        CopyMemory(GlobalLock(hData), LockResource(hResource),
cbImage);

        GlobalUnlock(hData);
        UnlockResource(hResource);
    } // if

```

```
25      FreeResource(hResource); // освобождение ресурса
26
27      //////////////////
28
29      if (NULL != hData)
30      {
31          IStream *pStream = NULL;
32
33          // создание COM-объекта
34          HRESULT hr = CreateStreamOnHGlobal(hData, TRUE,
    &pStream);
35
36          if (SUCCEEDED(hr) && NULL != pStream)
37          {
38              Image *image = Image::FromStream(pStream); //
    загрузка изображения
39              pStream->Release();
40
41              return image;
42          } // if
43
44          GlobalFree(hData); // освобожение памяти
45      } // if
46
47      return NULL;
48  } // ImageFromResource
```

Класс `Image` предоставляет очень удобный механизм создания эскизов изображений, которые содержат уменьшенную копию изображения. Поскольку эскизы имеют очень малые размеры, они очень быстро отображаются. Для создания эскизов в классе `Image` определен метод `GetThumbnailImage`:

```
Image* GetThumbnailImage(UINT thumbWidth, UINT thumbHeight,
    GetThumbnailImageAbort callback = NULL,
    VOID *callbackData = NULL);
```

Первые два параметра, *`thumbWidth`* и *`thumbHeight`*, указывают требуемые размеры эскиза: соответственно, ширину и высоту. Параметры *`callback`* и *`callbackData`* могут использоваться, если нужна возможность прерывания процесса создания эскиза (см. документацию Platform SDK). Эти параметры могут быть установлены в `NULL`.

В случае успеха метод `GetThumbnailImage` возвращает указатель на объект класса `Image`, который должен быть удален вызовом оператора `delete` после того как станет не нужен. В случае ошибки – `NULL`.

Следующий пример демонстрирует создание эскиза размером 320×240 из файла растрового изображения:

```
// загрузка изображения 320x240 из файла Sample.png
Image *image = Image(L"Sample.png").GetThumbnailImage(320, 240);

if (NULL != image)
{
    /* Изображение успешно загружено */
} // if
```

Вывод графики с помощью GDI+

Все операции графического вывода в GDI+ выполняются с помощью методов класса `Graphics`, который определен в заголовочном файле gdiplusgraphics.h.

Класс `Graphics` имеет следующие конструкторы:

```
Graphics(HDC hdc);

Graphics(HDC hdc, HANDLE hdevice);

Graphics(HWND hwnd, BOOL icm = FALSE);

Graphics(Image *image);
```

Первый и второй конструкторы создают объект класса `Graphics` на основе дескриптора контекста устройства (подробнее см. в разделе «Контекст устройства»), на который указывает параметр *hdc*. Второй конструктор для создания объекта класса `Graphics` также использует дескриптор устройства отображения (параметр *hdevice*). Третий конструктор создает объект класса `Graphics` на основе дескриптора окна *hwnd*. Параметр *icm* определяет, будет ли применяться настройка цвета. Если параметр *icm* имеет значение `FALSE`, настройка цвета не применяется. Четвертый конструктор создает объект класса `Graphics` на основе объекта класса `Image`, на который указывает параметр *image*.

В приведенном далее примере показана функция отображения, в которой на основе контекста устройства отображения создается объект класса `Graphics`.

Листинг 1.15. Пример функции отображения

```
void Display(HDC hdc)
{
    // создаем объект класса Graphics для рисования
    Graphics g(hdc);
} // Display
```

Очистка

Рисование на экране компьютера отличается от рисования на бумаге, так как бумага изначально является белой и все, что нужно сделать – это нарисовать на ней изображение. Однако при рисовании на экране компьютера прежде необходимо удалить старое уже нарисованное изображение, хранящееся в памяти компьютера.

В классе `Graphics` есть метод `Clear`, который очищает область отображения некоторым цветом фона. Метод `Clear` имеет следующий прототип:

```
Status Clear(const Color &color);
```

Параметр *color* – объект класса `Color`, в котором представлен цвет фона, используемый для очистки области отображения.

Рисование фигур и линий

В классе `Graphics` все методы рисования имеют две версии: методы с префиксом `Draw` для рисования линий, контуров и т.п. и с префиксом `Fill` для заливки замкнутых фигур. Методам с префиксом `Draw` в качестве параметра требуется указатель на объект класса `Pen`, а методам с префиксом `Fill` передается указатель на объект класса, порожденного от класса `Brush`.

Рисование линий

Для рисования различных линий используется метод `DrawLine` класса `Graphics`. Этот метод имеет следующие прототипы:

```
Status DrawLine(const Pen *pen,
    INT x1, INT y1, INT x2, INT y2);

Status DrawLine(const Pen *pen,
    REAL x1, REAL y1, REAL x2, REAL y2);

Status DrawLine(const Pen *pen,
    const Point &pt1, const Point &pt2);

Status DrawLine(const Pen *pen,
    const PointF &pt1, const PointF &pt2);
```

Параметр *pt1* (*pt2*), а также параметры *x1* (*x2*) и *y1* (*y2*) определяют координаты первой (второй) точки рисуемой линии.

В листинге 1.16 представлен пример, в котором демонстрируется рисование двух линий пером черного цвета.

Листинг 1.16. Пример рисования линий

```
1   void Display(HDC hdc)
2   {
```

```
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем перо черного цвета
9       Pen blackPen(Color::Black);
10
11
12      // рисуем линию с координатами (100,100) и (200,80)
13      g.DrawLine(&blackPen, 100, 100, 200, 80);
14
15
16      Point pt1(120, 20); // точка (120,20)
17      Point pt2(40, 60); // точка (40,60)
18
19      // рисуем линию с координатами (120,20) и (40,60)
20      g.DrawLine(&blackPen, pt1, pt2);
21  } // Display
```

Для рисования ломаных линий используется метод `DrawLines` класса `Graphics`. Этот метод имеет следующие прототипы:

```
Status DrawLines(const Pen *pen,
    const Point *points, INT count);

Status DrawLines(const Pen *pen,
    const PointF *points, INT count);
```

Параметр *points* указывает на массив объектов класса `Point` (`PointF`), которые содержат координаты вершин рисуемой ломаной линии, параметр *count* задает количество элементов в массиве, на который указывает параметр *points*.

В следующем примере демонстрируется рисование ломаной линии из четырех точек с помощью пера черного цвета.

Листинг 1.17. Пример рисования ломаной линии

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем перо черного цвета
9       Pen blackPen(Color::Black);
10
```

```
11      Point points[4] = {
12          Point(100, 100), // точка (100,100)
13          Point(200, 80),  // точка (200,80)
14          Point(120, 20),  // точка (120,20)
15          Point(40, 60)    // точка (40,60)
16      };
17
18      // рисуем ломаную линию
19      g.DrawLines(&blackPen, points, 4);
20  } // Display
```

Рисование прямоугольников

Для рисования прямоугольников используется метод `DrawRectangle` класса `Graphics`, а для закрашивания прямоугольников используется метод `FillRectangle` этого класса. Методы `DrawRectangle` и `FillRectangle` имеет следующие прототипы:

```
Status DrawRectangle(const Pen *pen,
    INT x, INT y, INT width, INT height);

Status DrawRectangle(const Pen *pen,
    REAL x, REAL y, REAL width, REAL height);

Status DrawRectangle(const Pen *pen, const Rect &rect);

Status DrawRectangle(const Pen *pen, const RectF &rect);


Status FillRectangle(const Brush *brush,
    INT x, INT y, INT width, INT height);

Status FillRectangle(const Brush *brush,
    REAL x, REAL y, REAL width, REAL height);

Status FillRectangle(const Brush *brush, const Rect &rect);

Status FillRectangle(const Brush *brush, const RectF &rect);
```

Параметр *rect*, а также параметры *x*, *y*, *width* и *height*, определяют расположение и размер прямоугольника.

Как уже было сказано ранее, в классе `Graphics` методы закрашивания замкнутых фигур отделены от методов рисования контуров таких фигур. Поэтому если необходимо нарисовать закрашенный прямоугольник с контуром, то первым делом необходимо нарисовать закрашенный прямоугольник с помощью метода `FillRectangle`, а затем контур этого прямоугольника с помощью метода `DrawRectangle`.

В листинге 1.18 представлен пример демонстрирующий рисование двух прямоугольников, изображенных на рис. 1.27.

Листинг 1.18. Пример рисования прямоугольников

```
void Display(HDC hdc)
{
    // создаем объект класса Graphics для рисования
    Graphics g(hdc);
    // выполняем очистку перед рисованием
    g.Clear(Color::White); // белый цвет фона

    // создаем перо зеленого цвета
    Pen pen(Color(0, 128, 0));

    // создаем сплошную кисть
    SolidBrush solidBrush(Color::Yellow); // желтый цвет


    // рисуем закрашенный прямоугольник
    g.FillRectangle(&solidBrush, 100, 70, 120, 50);
    // рисуем контур прямоугольника
    g.DrawRectangle(&pen, 100, 70, 120, 50);


    Rect rect(10, 10, 100, 50);

    // рисуем закрашенный прямоугольник
    g.FillRectangle(&solidBrush, rect);
    // рисуем контур прямоугольника
    g.DrawRectangle(&pen, rect);
} // Display
```

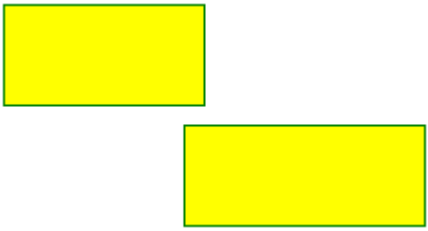


Рис. 1.27. Пример нарисованных прямоугольников

Для рисования и закрашивания сразу нескольких прямоугольников используются, соответственно, методы `DrawRectangles` и `FillRectangles` класса `Graphics`. Эти методы имеет следующие прототипы:

```
Status DrawRectangles(const Pen *pen,
    const Rect *rects, INT count);
```

```
Status DrawRectangles(const Pen *pen,
    const RectF *rects, INT count);

Status FillRectangles(const Brush *brush,
    const Rect *rects, INT count);
Status FillRectangles(const Brush *brush,
    const RectF *rects, INT count);
```

Параметр *rects* указывает на массив объектов класса Rect (RectF), которые определяют расположение и размеры прямоугольников, а параметр *count* задает количество элементов в массиве, на который указывает параметр *rects*.

В листинге 1.19 представлен немного видоизменный пример из листинга 1.18. В этом примере для рисования сразу двух прямоугольников (см. рис. 1.27) используются методы FillRectangles и DrawRectangles.

Листинг 1.19. Пример рисования сразу нескольких прямоугольников

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем перо зеленого цвета
9       Pen pen(Color(0, 128, 0));
10
11      // создаем сплошную кисть
12      SolidBrush solidBrush(Color::Yellow); // желтый цвет
13
14
15      // массив прямоугольников
16      Rect rects[2] = {
17          Rect(100, 70, 120, 50),
18          Rect(10, 10, 100, 50)
19      };
20
21      // рисуем закрашенные прямоугольники
22      g.FillRectangles(&solidBrush, rects, 2);
23      // рисуем контуры прямоугольников
24      g.DrawRectangles(&pen, rects, 2);
25  } // Display
```

Рисование многоугольников

Для рисования и закрашивания полигонов (многоугольников) используются, соответственно, методы `DrawPolygon` и `FillPolygon` класса `Graphics`. Эти методы имеют следующие прототипы:

```
Status DrawPolygon(const Pen *pen,
    const Point *points, INT count);

Status DrawPolygon(const Pen *pen,
    const PointF *points, INT count);

Status FillPolygon(const Brush *brush,
    const Point *points, INT count);

Status FillPolygon(const Brush *brush,
    const PointF *points, INT count);

Status FillPolygon(const Brush *brush,
    const Point *points, INT count, FillMode fillMode);

Status FillPolygon(const Brush *brush,
    const PointF *points, INT count, FillMode fillMode);
```

Параметр *points* указывает на массив объектов класса `Point` (`PointF`), которые содержат координаты вершин рисуемого полигона, а параметр *count* задает количество элементов в массиве, на который указывает параметр *points*. Параметр *fillMode* определяет, как будет производиться заполнение для внутренней части полигона. Этот параметр должен принимать одно из следующих значений перечисляемого типа `FillMode`:

- `FillModeAlternate` – режим заполнения с чередованием;
- `FillModeWinding` – режим заполнения с поворотом.

При заполнении полигонов необходимо определить область, которая должна быть заполнена. По умолчанию используется режим заполнения с чередованием. Для того чтобы определить область заполнения в режиме заполнения с чередованием, строится луч из произвольной точки внутри полигона к некоторой точке, которая явно расположена вне полигона. Если луч пересекает нечетное число ребер полигона, выбранная точка расположена в заполняемой области. Четное число пересечений означает, что точка не находится в области заполнения.

В режиме заполнения с поворотом, так же строится луч, а затем находятся пересечения этого луча с ребрами полигона. Каждому такому пересечению присваивается +1 или –1, в зависимости от того, как ребро пересекает луч – по часовой (слева направо) или против часовой

стрелки (справа налево). Если сумма этих значений будет отлична от нуля, точка считается расположенной внутри области заполнения. Если же сумма равна нулю, точка находится вне области заполнения.

На рис. 1.28 показана пятиконечная звезда, нарисованная с различными режимами заполнения.

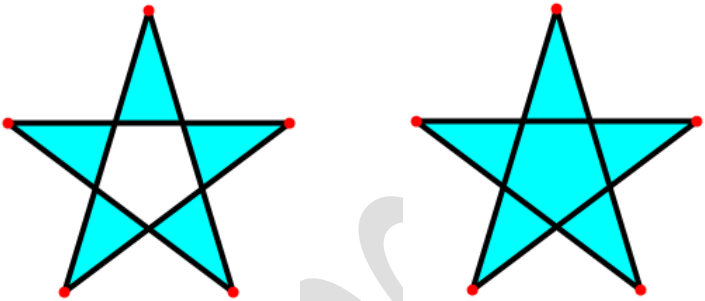


Рис. 1.28. Примеры заполнения полигонов (с указанием вершин)

В следующем примере демонстрируется рисование закрашенного треугольника с контуром черного цвета.

Листинг 1.20. Пример рисования треугольника

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем перо черного цвета
9       Pen blackPen(Color::Black);
10
11      // создаем штриховую кисть
12      HatchBrush hatchBrush(
13          HatchStyleDiagonalBrick, Color::Black, Color::LightGray);
14
15      // вершины треугольника
16      Point points[3] = {
17          Point(100, 100), // точка (100,100)
18          Point(200, 130), // точка (200,130)
19          Point(110, 200)  // точка (110,200)
20      };
21
```

```
22      // рисуем закрашенный треугольник
23      g.FillPolygon(&hatchBrush, points, 3);
24      // рисуем контур треугольника
25      g.DrawPolygon(&blackPen, points, 3);
26  } // Display
```

Рисование эллипсов, окружностей, дуг и секторов

Для рисования и закрашивания эллипсов используются, соответственно, методы `DrawEllipse` и `FillEllipse` класса `Graphics`. Эти методы имеют следующие прототипы:

```
Status DrawEllipse(const Pen *pen,
    INT x, INT y, INT width, INT height);

Status DrawEllipse(const Pen *pen,
    REAL x, REAL y, REAL width, REAL height);

Status DrawEllipse(const Pen *pen, const Rect &rect);

Status DrawEllipse(const Pen *pen, const RectF &rect);

Status FillEllipse(const Brush *brush,
    INT x, INT y, INT width, INT height);

Status FillEllipse(const Brush *brush,
    REAL x, REAL y, REAL width, REAL height);

Status FillEllipse(const Brush *brush, const Rect &rect);

Status FillEllipse(const Brush *brush, const RectF &rect);
```

Параметр *rect*, а также параметры *x*, *y*, *width* и *height*, определяют расположение и размер эллипса, как показано на рис. 1.29.

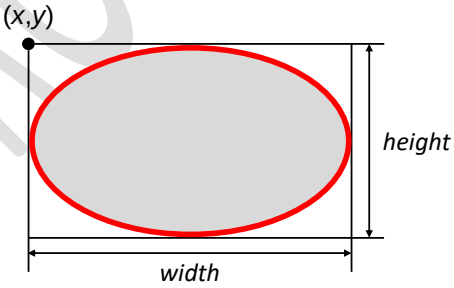


Рис. 1.29. Расположение и размер эллипса

Следующий пример демонстрирует рисование закрашенного эллипса, круга и окружности, изображенных на рис. 1.30.

Листинг 1.21. Пример рисования эллипса, круга и окружности

```
void Display(HDC hdc)
{
    // создаем объект класса Graphics для рисования
    Graphics g(hdc);
    // выполняем очистку перед рисованием
    g.Clear(Color::White); // белый цвет фона

    // создаем перо черного цвета
    Pen blackPen(Color::Black);

    // создаем две сплошные кисти
    SolidBrush solidBrush1(Color::Yellow);
    SolidBrush solidBrush2(Color::Aqua);

    // рисуем закрашенный эллипс
    g.FillEllipse(&solidBrush1, 10, 10, 160, 80);
    // рисуем контур эллипса
    g.DrawEllipse(&blackPen, 10, 10, 160, 80);


    Rect rect(180, 50, 70, 70);

    // рисуем круг
    g.FillEllipse(&solidBrush2, rect);
    // рисуем окружность
    g.DrawEllipse(&blackPen, rect);
} // Display
```

Как видно из приведенного примера, перечисленные методы класса `Graphics` для рисования и закрашивания эллипсов также можно использовать, соответственно, для рисования окружности и круга.

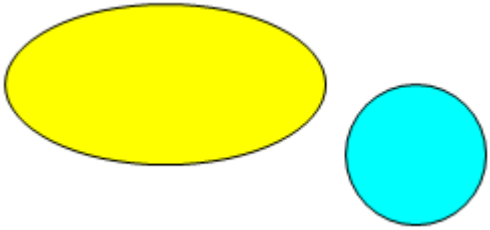


Рис. 1.30. Пример эллипса, круга и окружности

Рисование дуги похоже на рисование эллипса. Для рисования дуг использует метод DrawArc класса Graphics. Этот метод имеет следующие прототипы:

```
Status DrawArc(const Pen *pen,
    INT x, INT y, INT width, INT height,
    REAL startAngle, REAL sweepAngle);

Status DrawArc(const Pen *pen,
    REAL x, REAL y, REAL width, REAL height,
    REAL startAngle, REAL sweepAngle);

Status DrawArc(const Pen *pen,
    const Rect &rect, REAL startAngle, REAL sweepAngle);

Status DrawArc(const Pen *pen,
    const RectF &rect, REAL startAngle, REAL sweepAngle);
```

Параметр *rect*, а также параметры *x*, *y*, *width* и *height*, определяют расположение и размер прямоугольника, в который вписывается рисуемая дуга. Параметр *startAngle* задает угол (в градусах) между осью *x* и начальной точкой дуги. Параметр *sweepAngle* задает угол (в градусах) между начальной и конечной точкой дуги.

На рис. 1.31 показано, как рисуется дуга по заданным параметрам.

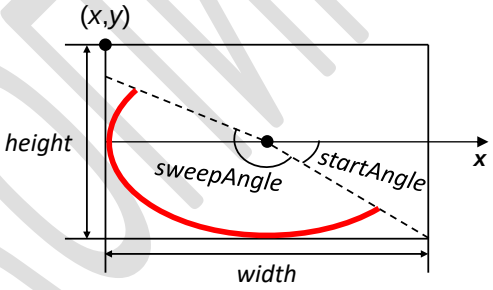


Рис. 1.31. Рисование дуги

В листинге 1.22 представлен пример демонстрирующий рисование двух дуг пером красного цвета и толщиной 2.

Листинг 1.22. Пример рисования дуг

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
```

```
7
8       // создаем перо красного цвета и толщиной 2
9       Pen redPen(Color::Red, 2.f);
10
11      // рисуем дугу от -10 градусов до 130 градусов
12      g.DrawArc(&redPen, 100, 70, 120, 50, -10.f, 140.f);
13
14
15      Rect rect(10, 10, 100, 100);
16
17      // рисуем дугу от 90 градусов до 300 градусов
18      g.DrawArc(&redPen, rect, 90.f, 210.f);
19  } // Display
```

Для того чтобы нарисовать фигуру «сектор» (например, как на рис. 1.32) в классе `Graphics` есть метод `DrawPie` и есть метод `FillPie`, который закрашивает эту фигуру.

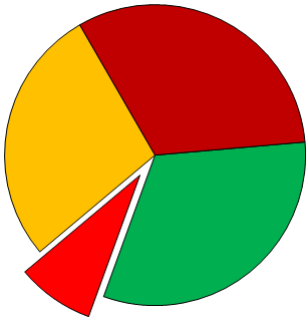


Рис. 1.32. Пример рисования секторов

Методы `DrawPie` и `FillPie` класса `Graphics` имеют следующие прототипы:

```
Status DrawPie(const Pen *pen,
    INT x, INT y, INT width, INT height,
    REAL startAngle, REAL sweepAngle);

Status DrawPie(const Pen *pen,
    REAL x, REAL y, REAL width, REAL height,
    REAL startAngle, REAL sweepAngle);

Status DrawPie(const Pen *pen,
    const Rect &rect, REAL startAngle, REAL sweepAngle);

Status DrawPie(const Pen *pen,
    const RectF &rect, REAL startAngle, REAL sweepAngle);
```

```
Status FillPie(const Brush *brush,
    INT x, INT y, INT width, INT height,
    REAL startAngle, REAL sweepAngle);

Status FillPie(const Brush *brush,
    REAL x, REAL y, REAL width, REAL height,
    REAL startAngle, REAL sweepAngle);

Status FillPie(const Brush *brush,
    const Rect &rect, REAL startAngle, REAL sweepAngle);

Status FillPie(const Brush *brush,
    const RectF &rect, REAL startAngle, REAL sweepAngle);
```

Параметр *rect*, а также параметры *x*, *y*, *width* и *height*, определяют расположение и размер прямоугольника, в который вписывается сектор. Параметр *startAngle* задает угол (в градусах) между осью *x* и начальной точкой сектора. Параметр *sweepAngle* задает угол (в градусах) между начальной и конечной точкой сектора.

В следующем примере продемонстрировано рисование диаграммы (см. рис. 1.31), состоящей из четырех секторов.

Листинг 1.23. Пример рисования секторной диаграммы

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем перо черного цвета
9       Pen blackPen(Color::Black);
10
11      // создаем сплошную кисть
12      SolidBrush solidBrush(Color::Red); // красный цвет
13
14      // рисуем первый сектор
15      g.FillPie(&solidBrush, -5, 30, 300, 300, 110.f, 30.f);
16      // рисуем контур первого сектора
17      g.DrawPie(&blackPen, -5, 30, 300, 300, 110.f, 30.f);
18
19      // рисуем еще три сектора...
20
21      Color colors[3] = {
22          Color(255, 192, 0),
23          Color(192, 0, 0),
24          Color(0, 176, 80)};
```

```
25
26      float sweepAngle[3] = {
27          100.f, 115.f, 115.f};
28
29      float angle = 140.f; // начальный угол
30      Rect rect(10, 10, 300, 300);
31
32      for (int i = 0; i < 3; ++i)
33      {
34          solidBrush.SetColor(colors[i]); // изменяем цвет кисти
35
36          // рисуем сектор
37          g.FillPie(&solidBrush, rect, angle, sweepAngle[i]);
38          // рисуем контур сектора
39          g.DrawPie(&blackPen, rect, angle, sweepAngle[i]);
40
41          angle += sweepAngle[i]; // увеличиваем угол
42      } // for
43  } // Display
```

Рисование сплайнов

В дополнение к рисованию линий и дуг GDI+ поддерживает рисование *сплайнов* (splines), которые являются очень удобным инструментом для рисования кривых в компьютерной графике.

В классе `Graphics` есть несколько методов, которые позволяют рисовать *кубические сплайны* (cubic splines) и *сплайны Безье* (splines Bezier). Для рисования кубических сплайнов необходимо использовать методы `DrawCurve` и `DrawClosedCurve`. Метод `DrawCurve` рисует обычный кубический сплайн, а метод `DrawClosedCurve` – замкнутый кубический сплайн. Эти методы имеют следующие прототипы:

```
Status DrawCurve(const Pen *pen,
    const Point *points, INT count);

Status DrawCurve(const Pen *pen,
    const PointF *points, INT count);

Status DrawCurve(const Pen *pen,
    const Point *points, INT count, REAL tension);

Status DrawCurve(const Pen *pen,
    const PointF *points, INT count, REAL tension);

Status DrawCurve(const Pen *pen,
    const Point *points, INT count, INT offset,
    INT numberOfSegments, REAL tension = 0.5f);
```

```
Status DrawCurve(const Pen *pen,
    const PointF *points, INT count, INT offset,
    INT numberOfSegments, REAL tension = 0.5f);

Status DrawClosedCurve(const Pen *pen,
    const Point *points, INT count);
Status DrawClosedCurve(const Pen *pen,
    const PointF *points, INT count);
Status DrawClosedCurve(const Pen *pen,
    const Point *points, INT count, REAL tension);
Status DrawClosedCurve(const Pen *pen,
    const PointF *points, INT count, REAL tension);
```

Параметр *points* указывает на массив объектов класса `Point` (`PointF`), определяющий набор точек, через которые проходит сплайн (рис. 1.33), а параметр *count* задает количество элементов в массиве, на который указывает параметр *points*.

Параметр *tension* задает напряжение сплайна. По умолчанию напряжение сплайна равно `0.5`.

Параметр *offset* указывает индекс (начиная с 0) точки в массиве, с которой начинается рисование сплайна, а параметр *numberOfSegments* указывает их количество. Важно отметить, что все «невидимые» точки из массива, на который указывает параметр *points*, все равно будут использованы для расчетов.

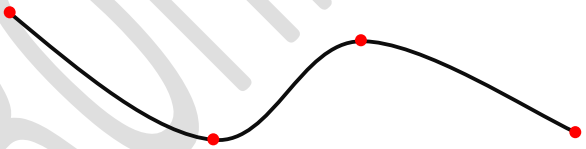


Рис. 1.33. Кубический сплайн, заданный четырьмя точками

Кроме того в классе `Graphics` есть метод `FillClosedCurve` для закрашивания замкнутых кубических сплайнов.

```
Status FillClosedCurve(const Brush *brush,
    const Point *points, INT count);
Status FillClosedCurve(const Brush *brush,
    const PointF *points, INT count);
Status FillClosedCurve(const Brush *brush,
    const Point *points, INT count, FillMode fillMode,
    REAL tension = 0.5f);
```

```
Status FillClosedCurve(const Brush *brush,
    const PointF *points, INT count, FillMode fillMode,
    REAL tension = 0.5f);
```

Параметры *points*, *count* и *tension* аналогичны одноименным параметрам метода DrawClosedCurve. Параметр *fillMode* определяет, как будет производиться заполнение области, образуемой замкнутым сплайном. Этот параметр должен принимать одно из значений перечисляемого типа FillMode.

В листинге 1.24 показан пример рисования замкнутого кубического сплайна, проходящего через семь точек.

Листинг 1.24. Пример рисования замкнутого кубического сплайна

```
void Display(HDC hdc)
{
    // создаем объект класса Graphics для рисования
    Graphics g(hdc);
    // выполняем очистку перед рисованием
    g.Clear(Color::White); // белый цвет фона

    // создаем перо зеленого цвета
    Pen pen(Color::Green);

    // создаем сплошную кисть
    SolidBrush solidBrush(Color::Yellow); // желтый цвет

    // массив точек, через кот. Проходит сплайн
    Point points[7] = {
        Point(50, 50),   // точка (50,50)
        Point(100, 25),  // точка (100,25)
        Point(200, 5),   // точка (200,5)
        Point(250, 50),  // точка (250,50)
        Point(300, 100), // точка (300,100)
        Point(350, 200), // точка (350,200)
        Point(250, 250)  // точка (250,250)
    };

    // закрашиваем замкнутый сплайн
    g.FillClosedCurve(&solidBrush, points, 7);
    // рисуем замкнутый сплайн
    g.DrawClosedCurve(&pen, points, 7);
} // Display
```

Если нужно нарисовать сплайн Безье следует использовать метод DrawBezier или DrawBeziers. Метод DrawBezier рисует обычный сплайн

Безье, а метод DrawBeziers – последовательность соединенных сплайнов Безье.

Метод DrawBezier имеет следующие прототипы:

```
Status DrawBezier(const Pen *pen,
    INT x1, INT y1, INT x2, INT y2, INT x3, INT y3,
    INT x4, INT y4);

Status DrawBezier(const Pen *pen,
    REAL x1, REAL y1, REAL x2, REAL y2, REAL x3, REAL y3,
    REAL x4, REAL y4);

Status DrawBezier(const Pen *pen,
    const Point &pt1, const Point &pt2, const Point &pt3,
    const Point &pt4);

Status DrawBezier(const Pen *pen,
    const PointF &pt1, const PointF &pt2, const PointF &pt3,
    const PointF &pt4);
```

Параметры *pt1*, *pt2*, *pt3* и *pt4*, а также параметры *x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4* и *y4* задают координаты точек, определяющих положение и кривизну сплайна Безье.

В следующем примере демонстрируется рисование двух различных сплайнов Безье с помощью метода DrawBezier класса Graphics.

Листинг 1.25. Пример рисования сплайнов Безье

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем два пера
9       Pen pen1(Color::Green); // зеленый цвет
10      Pen pen2(Color::Red); // красный цвет
11
12      // рисуем сплайн Безье с начальной точкой (100,100)
13      // и конечной точкой (500,100)
14      g.DrawBezier(&pen1,
15          100, 100, 200, 10, 350, 50, 500, 100);
16
17      Point pt1(100, 200); // точка (100,200)
18      Point pt2(200, 150); // точка (200,150)
19      Point pt3(400, 110); // точка (400,110)
20      Point pt4(500, 200); // точка (500,200)
```

```
21
22      // рисуем сплайн Безье с начальной точкой (100,200)
23      // и конечной точкой (500,200)
24      g.DrawBezier(&pen2, pt1, pt2, pt3, pt4);
25  } // Display
```

Метод `DrawBeziers` имеет следующие прототипы:

```
Status DrawBeziers(const Pen *pen,
    const Point *points, INT count);

Status DrawBeziers(const Pen *pen,
    const PointF *points, INT count);
```

Параметр *points* указывает на массив объектов класса `Point` (`PointF`), определяющий набор контрольных точек сплайнов Безье, а параметр *count* задает количество элементов в массиве, на который указывает параметр *points*. При этом точки в массиве должны располагаться так, чтобы последняя точка одного сплайна Безье, являлась начальной точкой следующего сплайна Безье.

Следующий пример демонстрирует рисование кривой, состоящей сразу из двух сплайнов Безье.

Листинг 1.26. Пример рисования кривой, состоящей из сплайнов Безье

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем перо синего цвета
9       Pen pen(Color::Blue);
10
11      Point points[7] = {
12          Point(100, 100),  // начальная точка 1-го сплайна
13          Point(200, 50),
14          Point(400, 10),
15          Point(500, 100),  // конечная точка 1-го сплайна
16          Point(600, 200),
17          Point(500, 400),
18          Point(800, 350)}; // конечная точка 2-го сплайна
19
20      // рисуем кривую из сплайнов Безье
21      g.DrawBeziers(&pen, points, 7);
22  } // Display
```

Отображение текста

Для отображения текста в классе Graphics определен метод DrawString, который имеет следующие прототипы:

```
Status DrawString(const WCHAR *string, INT length,
    const Font *font, const PointF &origin,
    const Brush *brush);

Status DrawString(const WCHAR *string, INT length,
    const Font *font, const PointF &origin,
    const StringFormat *stringFormat, const Brush *brush);

Status DrawString(const WCHAR *string, INT length,
    const Font *font, const RectF &layoutRect,
    const StringFormat *stringFormat, const Brush *brush);
```

Первый параметр, *string*, указывает на Unicode-строку, содержащую отображаемый текст, а второй параметр, *length*, задает длину этой строки. Параметр *length* может принимать значение -1, если строка завершается нуль-символом.

Третий параметр, *font*, указывает на объект класса Font, который представляет шрифт для отображения текста.

Параметр *origin* – объект класса PointF, который содержит координаты точки расположения текста, а параметр *layoutRect* определяет прямоугольник, в котором будет отображаться текст.

Параметр *stringFormat* указывает на объект класса StringFormat, который управляет форматированием текста.

Последний параметр, *brush*, указывает на объект класса Brush, который представляет кисть для отображения текста.

В следующем примере показано, как с помощью метода DrawString класса Graphics вывести текст «Привет мир!» в виде одной горизонтальной строки.

Листинг 1.27. Пример простого вывода текста

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем шрифт Times New Roman
9       Font font(L"Times New Roman", 20.f, FontStyleBold);
10      // создаем сплошную кисть
11      SolidBrush blackBrush(Color::Black); // черный цвет
12      // выводим текст, который начинается в точке (10,10)
```

```
13      g.DrawString(L"Привет мир!", -1, &font, PointF(10.f, 10.f),
    &blackBrush);
14  } // Display
```

При необходимости форматированного вывода текста необходимо использовать объект класса `StringFormat`, который определен в заголовочном файле gdiplusstringformat.h.

Класс `StringFormat` имеет следующие конструкторы:

```
StringFormat(INT formatFlags = 0,
    LANGID language = LANG_NEUTRAL);

StringFormat(const StringFormat *format);
```

Первый конструктор создает объект класса `StringFormat` на основе флагов форматирования, которые задает параметр *`formatFlags`*, и идентификатора язык, который задается параметром *`language`*.

Параметр *`formatFlags`* может принимать одно (или комбинацию) из перечисляемого типа `StringFormatFlags` (подробнее см. в разделе «Флаги форматирования»), а значение параметра *`language`* можно задавать с помощью макроса `MAKELANGID`, следующим образом:

```
1   LANGID language = MAKELANGID(
2       LANG_RUSSIAN, SUBLANG_RUSSIAN_RUSSIA);
```

Второй конструктор класса `StringFormat` создает копию объекта этого же класса, на который указывает параметр *`format`*.

В листинге 1.28 представлен пример форматированного вывода текста, результат которого показан на рис. 1.34.

Листинг 1.28. Пример форматированного вывода текста

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем шрифт Times New Roman
9       Font font(L"Times New Roman", 20.f, FontStyleBold);
10      // создаем сплошную кисть
11      SolidBrush blackBrush(Color::Black); // черный цвет
12      // создаем перо красного цвета
13      Pen pen(Color::Red);
14
15      // создаем объект класса StringFormat
16      // для форматированного вывода текста
```

```
17      StringFormat sf;
18
19      // ограничивающий прямоугольник
20      RectF rect(10.f, 10.f, 300.f, 80.f);
21
22      // выводим текст
23      g.DrawString(L"Съешь ещё этих мягких французских булок, да
    выпей чаю.", -1, &font, rect, &sf, &blackBrush);
24      // рисуем прямоугольник (только для наглядности)
25      g.DrawRectangle(&pen, rect);
26  } // Display
```

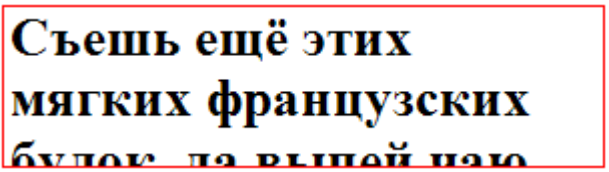


Рис. 1.34. Форматированный вывод текста

Прежде чем перейти к изучению методов класса `StringFormat`, следует отметить, что в этом классе (как и во многих других классах GDI+) есть методы `GetLastStatus` и `Clone`. Метод `Clone` имеет следующий прототип:

```
StringFormat* Clone() const;
```

В случае успеха метод `Clone` возвращает указатель на объект класса `StringFormat`, который должен быть удален вызовом оператора `delete` после того как станет не нужен. В случае ошибки – `NULL`.

Флаги форматирования

Флаги форматирования можно задавать в качестве одного из параметров конструктора класса `StringFormat` при создании объекта это класса. Можно также задать флаги форматирования с помощью метода `SetFormatFlags`:

```
Status SetFormatFlags(INT flags);
```

Параметр *`flags`* может принимать одно (или комбинацию) из следующих значений перечисляемого типа `StringFormatFlags`:

- `StringFormatFlagsDirectionRightToLeft` – текст будет отображаться справа на лево;
- `StringFormatFlagsDirectionVertical` – текст будет отображаться вертикально;

- StringFormatFlagsNoFitBlackBox – текст будет отображаться без дополнительного пустого пространства между внутренним контуром прямоугольной зоны форматирования и прямоугольником, ограничивающего отображаемый текст;
- StringFormatFlagsDisplayFormatControl – разрешает отображение управляющих символов, например, таких как метка слева направо (Left-to-Right Embedding, LRE);
- StringFormatFlagsNoFontFallback – запрещается обращение к альтернативным шрифтам при отображении символов, не поддерживаемых в указанном шрифте. Такие символы, как правило, будут отображаться с помощью пустого квадрата;
- StringFormatFlagsMeasureTrailingSpaces – запрещается исключать пробелы в конце каждой строки при определении размеров ограничивающего прямоугольника;
- StringFormatFlagsNoWrap – запрещается автоматический перенос текста на новую строку при форматировании в прямоугольнике;
- StringFormatFlagsLineLimit – при форматировании в прямоугольнике будут отображаться только целые строки;
- StringFormatFlagsNoClip – разрешается отображать текст, выходящий за пределы прямоугольной зоны форматирования.

Определить установленные флаги форматирования можно с помощью метода GetFormatFlags класса StringFormat:

```
INT GetFormatFlags() const;
```

На рис. 1.35 показаны результаты использования некоторых флагов форматирования текста.

Выравнивание текста

При отображении текста может потребоваться выровнять этот текст. Чтобы выровнять текст необходимо в метод DrawString класса Graphics передать объект класса StringFormat, в котором задать параметры выравнивания текста с помощью методов SetAlignment и SetLineAlignment. Метод SetAlignment задает выравнивание текста по горизонтали, а метод SetLineAlignment – по вертикали. Эти методы имеют следующие прототипы:

```
Status SetAlignment(StringAlignment align);
Status SetLineAlignment(StringAlignment align);
```

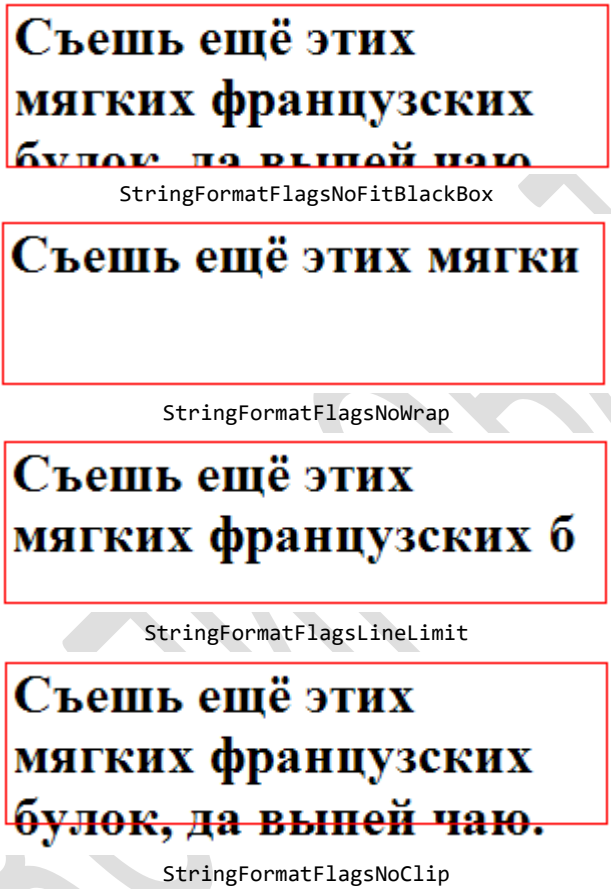


Рис. 1.35. Применение флагов форматирования текста

Параметр *`align`* задает выравнивание текста относительно точки расположения текста или относительно границ прямоугольника, в котором будет отображаться текст. Этот параметр должен принимать одно из следующих значений перечисляемого типа `StringAlignment`:

- `StringAlignmentNear` – выравнивание по ближнему краю;
- `StringAlignmentCenter` – выравнивание по центру;
- `StringAlignmentFar` – выравнивание по дальнему краю.

На рис. 1.36-37 показаны различные варианты выравнивания текста относительно точки и границ прямоугольника.

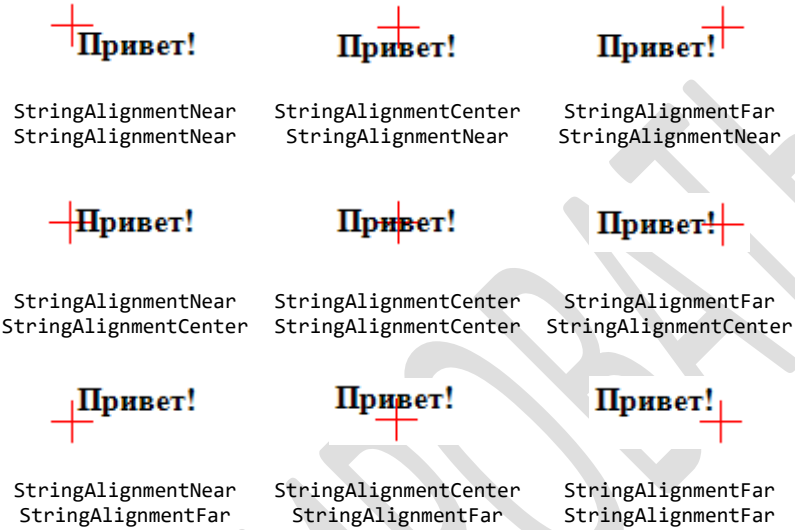


Рис. 1.36. Варианты выравнивания текста относительно точки

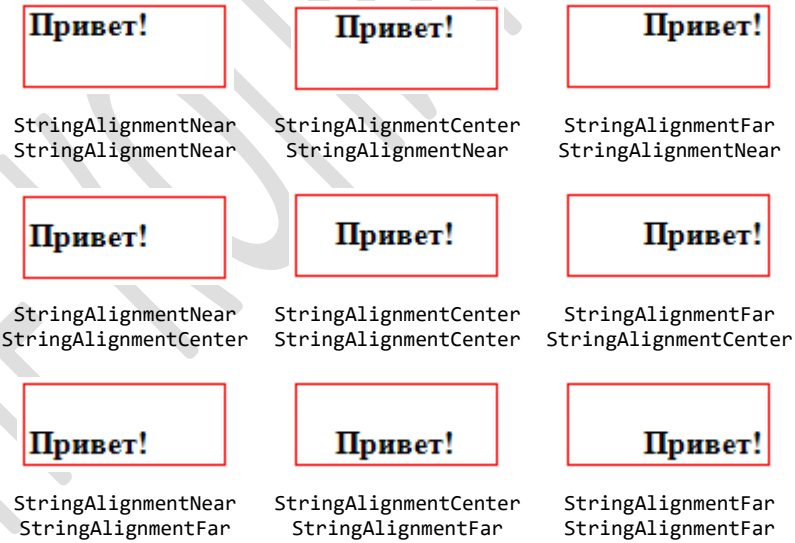


Рис. 1.37. Варианты выравнивания текста относительно границ прямоугольника

Для горизонтального выравнивания ближним краем будет левый, а дальним краем – правый. Однако, при отображений текста справа на лево ближним краем будет правый, а дальним краем – левый. Для вертикального выравнивания ближним краем будет верхний, а дальним краем – нижний.

Определить текущее горизонтальное и вертикальное выравнивание можно, соответственно, с помощью методов `GetAlignment` и `GetLineAlignment` класса `StringFormat`. Эти методы имеют следующие прототипы:

```
StringAlignment GetAlignment() const;
StringAlignment GetLineAlignment() const;
```

Вывод текста с табуляцией

Для отображаемого текста можно устанавливать позиции табуляции. Для этого в классе `StringFormat` есть метод `SetTabStops`. Этот метод имеет следующий прототип:

```
Status SetTabStops(REAL firstTabOffset, INT count,
    const REAL *tabStops);
```

Первый параметр, *`firstTabOffset`*, задает первое смещение позиции табуляции. Второй параметр, *`count`*, определяет количество элементов в массиве табуляции, на который указывает третий параметр – *`tabStops`*. Параметр *`tabStops`* указывает на массив значений типа `float`, которые задают смешение позиции табуляции.

Каждое смещение позиции табуляции, кроме первого, выполняется относительно предыдущей позиции. Первое смещение позиции табуляции выполняется относительно первоначальной позиции смещения. Например, если начальная позиция смещения равна `5` и первое смещение позиции табуляции равно `40`, первое смещение позиции табуляции расположено в позиции `45`. Если начальная позиция смещения равна нулю, первое смещение позиции табуляции определяется относительно позиции `0`, начала строки.

Следующий пример демонстрирует форматированный вывода текста с табуляцией. Результат показан на рис. 1.38.

Листинг 1.29. Пример задания позиций табуляции для выводимого текста

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
```

```
7
8       // создаем шрифт Arial
9       Font font(L"Arial", 11.f, FontStyleRegular);
10      // создаем сплошную кисть
11      SolidBrush blackBrush(Color::Black); // черный цвет
12      // создаем перо серого цвета
13      Pen pen(Color::Gray);
14
15      // создаем объект класса StringFormat
16      // для форматированного вывода текста
17      StringFormat sf;
18
19      // задаем выравнивание по левому краю
20      sf.SetAlignment(StringAlignmentNear);
21      // задаем выравнивание по верхнему краю
22      sf.SetLineAlignment(StringAlignmentNear);
23
24      float tabs[4] = {50.f, 100.f, 100.f, 100.f};
25      // устанавливаем позиции табуляции
26      sf.SetTabStops(0.f, 4, tabs);
27
28      // ограничивающий прямоугольник
29      RectF rect(10.f, 10.f, 350.f, 90.f);
30
31      const wchar_t text[] = L"№\tФамилия\tИмя\tОтчество\n\
32  1.\tАндреев\tАлександр\tДмитриевич\n\
33  2.\tВасильев\tЮрий\tГеннадиевич\n\
34  3.\tГромова\tИнна\tВикторовна\n\
35  4.\tМартынов\tПавел\tБорисович";
36
37      // выводим текст
38      g.DrawString(text, -1, &font, rect, &sf, &blackBrush);
39      // рисуем прямоугольник
40      g.DrawRectangle(&pen, rect);
41  } // Display
```

№	Фамилия	Имя	Отчество
1.	Андреев	Александр	Дмитриевич
2.	Васильев	Юрий	Геннадиевич
3.	Громова	Инна	Викторовна
4.	Мартынов	Павел	Борисович

Рис. 1.38. Форматированный вывод текста с табуляцией

Определить количество установленных позиций табуляции можно с помощью метода GetTabStopCount, а получить значения этих позиций можно с помощью метода GetTabStops. Прототипы этих методов записываются следующим образом:

```
INT GetTabStopCount() const;

Status GetTabStops(INT count, REAL *firstTabOffset,
     REAL *tabStops) const;
```

Первый параметр, *count*, указывает количество элементов в массиве, на который указывает параметр *tabStops*.

Второй параметр, *firstTabOffset*, указывает на переменную типа float, в которую будет записано значение первого смещения позиции табуляции.

Третий параметр, *tabStops*, указывает на массив типа float, в который будут сохранены значения установленных позиций табуляции.

Вывод «горячих» клавиш

Символ «&» (амперсанд) имеет специальное назначение в тексте, используемом в меню, кнопках и других элементах управления – он означает, что следующий за ним знак должен быть подчеркнут и будет использоваться для быстрого доступа к элементу управления. В классе StringFormat есть метод SetHotkeyPrefix, который определяет, как при отображении интерпретируются амперсанды.

Метод SetHotkeyPrefix имеет следующий прототип:

```
Status SetHotkeyPrefix(HotkeyPrefix hotkeyPrefix);
```

Параметр *hotkeyPrefix* определяет, как будут интерпретироваться амперсанды. Этот параметр должен принимать одно из следующих значений перечисляемого типа HotkeyPrefix:

- HotkeyPrefixNone – префикс «горячих» клавиш отсутствует;
- HotkeyPrefixShow – отображать префикс «горячих» клавиш;
- HotkeyPrefixHide – не отображать префикс «горячих» клавиш.

По умолчанию префикс «горячих» клавиш отсутствует. Определить, как интерпретируются амперсанды можно с помощью метода GetHotkeyPrefix класса StringFormat. Этот метод имеет следующий прототип:

```
HotkeyPrefix GetHotkeyPrefix() const;
```

На рис. 1.39 показаны варианты отображения текста «&Файл» с различными режимами интерпретации амперсанда.

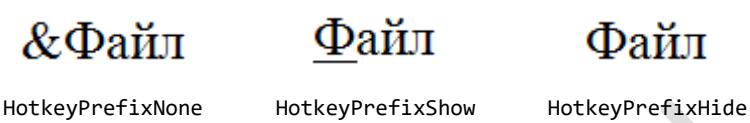


Рис. 1.39. Варианты интерпретации амперсанда

Обрезание текста

В классе `StringFormat` есть метод `SetTrimming`, который определяет, как должны отображаться символы, если текст не полностью помещается в ограничивающий прямоугольник. Метод `SetTrimming` имеет следующий прототип:

```
Status SetTrimming(StringTrimming trimming);
```

Параметр *`trimming`* определяет формат обрезания текста. Этот параметр должен принимать одно из следующих значений перечисляемого типа `StringTrimming`:

- `StringTrimmingNone` – текст не обрезается;
- `StringTrimmingCharacter` – текст обрезается по ближайшему символу;
- `StringTrimmingWord` – текст обрезается по ближайшему слову;
- `StringTrimmingEllipsisCharacter` – текст обрезается по ближайшему символу и в конце обрезанной строки вставляется троеточие;
- `StringTrimmingEllipsisWord` – текст обрезается по ближайшему слову и в конце обрезанной строки вставляется троеточие;
- `StringTrimmingEllipsisPath` – если текст не помещается в ограничивающий прямоугольник из него удаляется центральная часть и заменяется троеточием.

По умолчанию текст обрезается по ближайшему символу. Определить текущий формат обрезания текста можно с помощью метода `GetTrimming` класса `StringFormat`. Этот метод имеет следующий формат:

```
StringTrimming GetTrimming() const;
```

На рис. 1.40 показаны различные варианты обрезания символов, если текст не помещается в ограничивающий прямоугольник.

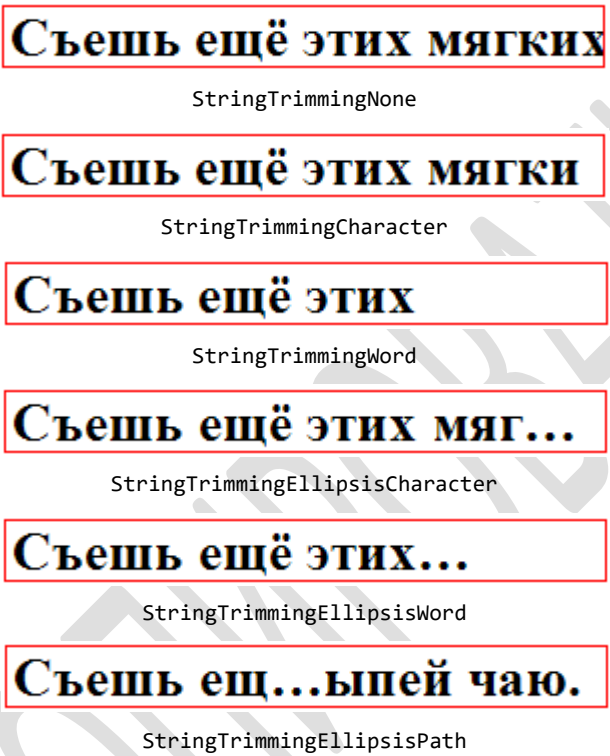


Рис. 1.40. Варианты обрезания текста

Определить ограничивающий прямоугольник для отображения текста с заданным шрифтом можно с помощью метода `MeasureString` класса `Graphics`. Этот метод имеет следующие прототипы:

```
Status MeasureString(const WCHAR *string, INT length,
    const Font *font, const PointF &origin,
    RectF *boundingBox) const;

Status MeasureString(const WCHAR *string, INT length,
    const Font *font, const RectF &layoutRect,
    RectF *boundingBox) const;

Status MeasureString(const WCHAR *string, INT length,
    const Font *font, const PointF &origin,
    const StringFormat *stringFormat,
    RectF *boundingBox) const;
```

```
Status MeasureString(const WCHAR *string, INT length,
    const Font *font, const RectF &layoutRect,
    const StringFormat *stringFormat, RectF *boundingBox,
    INT *codepointsFitted = 0, INT *linesFilled = 0) const;

Status MeasureString(const WCHAR *string, INT length,
    const Font *font, const RectF &layoutRect,
    const StringFormat *stringFormat, SizeF *size,
    INT *codepointsFitted = 0, INT *linesFilled = 0) const;
```

Параметры *string*, *length*, *font*, *origin*, *layoutRect* и *stringFormat* аналогичны одноименным параметрам метода DrawString класса Graphics.

Параметр *boundingBox* указывает на объект класса RectF, в который будут сохранены расположение и размер ограничивающего прямоугольника.

Параметр *size* указывает на объект класса SizeF, в который будет сохранен размер ограничивающего прямоугольника.

Параметр *codepointsFitted* указывает на переменную типа INT, в которую записывается количество символов в тексте, размещенном в ограничивающем прямоугольнике.

Параметр *linesFilled* указывает на переменную типа INT, в которую записывается количество строк в тексте, размещенном в ограничивающем прямоугольнике.

Предопределенные форматы

В классе StringFormat есть два статических метода GenericDefault и GenericTypographic, которые возвращают указатель на предопределенный объект класса StringFormat:

```
static const StringFormat* GenericDefault();

static const StringFormat* GenericTypographic();
```

Метод GenericDefault возвращает указатель на объект класса StringFormat, который представляет следующими параметрами форматирования:

- флагов форматирования нет;
- горизонтальное выравнивание по ближнему краю;
- вертикальное выравнивание по ближнему краю;
- позиции табуляции не установлены;
- префикс «горячих» клавиш отсутствует;
- текст обрезается по ближайшему символу.

Метод GenericTypographic возвращает указатель на объект класса StringFormat, который обладает следующими параметрами форматирования:

- заданы флаги форматирования StringFormatFlagsLineLimit, StringFormatFlagsNoClip и StringFormatFlagsNoFitBlackBox;
- горизонтальное выравнивание по ближнему краю;
- вертикальное выравнивание по ближнему краю;
- позиции табуляции не установлены;
- префикс «горячих» клавиш отсутствует;
- текст не обрезается.

Важно отметить, что не нужно удалять объект, указатель на который возвращают методы метода GenericDefault и GenericTypographic, после того, как необходимость в нем отпадет.

В следующем примере демонстрируется использование предопределенных объектов класса StringFormat для вывода текста.

Листинг 1.30. Пример форматированного вывода текста

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем шрифт Arial
9       Font font(L"Arial", 14.f, FontStyleRegular);
10      // создаем сплошную кисть
11      SolidBrush blackBrush(Color::Black); // черный цвет
12
13      // выводим текст, который начинается в точке (10,50)
14      g.DrawString(L"Привет мир!", -1, &font, PointF(10.f, 50.f),
    StringFormat::GenericDefault(), &blackBrush);
15
16      // создаем объект класса StringFormat
17      // на основе предопределенного объекта
18      StringFormat sf(StringFormat::GenericTypographic());
19
20      // выводим текст, который начинается в точке (10,10)
21      g.DrawString(L"Привет мир!", -1, &font, PointF(10.f, 10.f),
    &sf, &blackBrush);
22  } // Display
```

Вывод графических изображений

Для отображения содержимого растрового или векторного изображения в классе `Graphics` есть метод `DrawImage`, который имеет огромное количество перегруженных вариантов. Наиболее употребительные варианты имеют следующие прототипы:

```
Status DrawImage(Image *image, INT x, INT y);
Status DrawImage(Image *image, REAL x, REAL y);
Status DrawImage(Image *image, INT x, INT y, INT width,
    INT height);
Status DrawImage(Image *image, REAL x, REAL y, REAL width,
    REAL height);
Status DrawImage(Image *image, const Point &point);
Status DrawImage(Image *image, const PointF &point);
Status DrawImage(Image *image, const Rect &rect);
Status DrawImage(Image *image, const RectF &rect);
```

Первый параметр, *image*, указывает на объект класса `Image` или класса, производного от класса `Image`. Этот объект представляет изображение, содержимое которого необходимо отобразить.

Параметр *point*, а также параметры *x* и *y*, определяют расположение выводимого изображения.

Параметр *rect*, а также параметры *x*, *y*, *width* и *height*, определяют расположение и размер выводимого изображения.

В следующем примере демонстрируется использование метода `DrawImage` класса `Graphics` для вывода изображения.

Листинг 1.31. Пример вывода изображения

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // загружаем изображение Sample.png
9       Image image(L"Sample.png");
10
11      // вывод загруженного изображения
12      g.DrawImage(&image, 10, 10);
13  } // Display
```

Важно отметить, что чтение изображения из файла может происходить достаточно медленно. Поэтому разумнее всего было бы выне-

сти загрузку выводимых изображений из функции отображения, а не так как это сделано в приведенном примере (см. листинг 1.31).

Пропорциональное масштабирование изображения

Как показано на рис. 1.41 в процессе вывода изображения может возникнуть искажение из-за того, что при масштабировании форматное соотношение прямоугольника, который определяет расположение и размер выводимого изображения, не совпадает с форматным соотношением этого изображения.

Рис. 1.41. Искажение изображения

Форматное соотношение – это отношение ширины и высоты, которое находится по следующей формуле:

$$Ratio = \frac{width}{height}, \tag{1.2}$$

где $Ratio$ – форматное соотношение, а $width$ и $height$ – ширина и высота.

Для того чтобы при масштабировании изображения не происходило его искажения необходимо найти для него такие ширину и высоту, для которых бы сохранялось их изначальное отношение и которые бы полностью влезали бы в заданный прямоугольник.

Рассмотрим случай, когда форматное соотношение изображения больше чем форматное соотношение заданного прямоугольника. В этом случае ширина и высота изображения находятся по формуле:

$$Image.width = Rectangle.width\ ,$$
$$Image.height = \frac{1}{Ratio} \cdot Rectangle.width, \tag{1.3}$$

где $Image.width$ и $Image.height$ – новые ширина и высота изображения; $Rectangle.width$ – ширина прямоугольника; а $Ratio$ – форматное соотношение изображения.

В другом случае, когда изображение имеет меньшее форматное соотношение, чем у заданного прямоугольника, новые ширина и высота изображения находятся по другой формуле:

$$Image.width = Ratio \cdot Rectangle.height,$$
$$Image.height = Rectangle.height, \quad (1.4)$$

где $Image.width$ и $Image.height$ – новые ширина и высота изображения; $Rectangle.height$–высота прямоугольника; а $Ratio$ – форматное соотношение изображения.

В листинге 1.32 представлена функция, определяющей ширину и высоту изображения, для которых сохраняется их изначальное отношение, и которые будут полностью влезать в прямоугольник.

Листинг 1.32. Функция определяющая положение и размер изображения

```
1   Status MeasureImage(Image *image, const RectF &layout,
    StringAlignment align, RectF *result)
2   {
3       if (NULL == image || layout.IsEmptyArea() || NULL == result)
4       {
5           return InvalidParameter;
6       } // if
7
8       RectF rect;
9
10      // форматное соотношение изображения
11      float fRatio = (float)image->GetWidth() / (float)image-
    >GetHeight();
12
13      if (fRatio > (layout.Width / layout.Height))
14      {
15          rect.Width  = layout.Width;
16          rect.Height = layout.Width / fRatio;
17
18          rect.X = layout.GetLeft();
19
20          // выравнивание по вертикали
21          switch (align)
22          {
23          case StringAlignmentNear: // по ближнему краю
24              rect.Y = layout.GetTop();
25              break;
26          case StringAlignmentCenter: // по центру
27              rect.Y = layout.Y + (layout.Height-rect.Height)/2.f;
28              break;
```

```
29          case StringAlignmentFar: // по дальнему краю
30              rect.Y = layout.GetBottom() - rect.Height;
31              break;
32          } // switch
33      } // if
34      else
35      {
36          rect.Width  = fRatio * layout.Height;
37          rect.Height = layout.Height;
38
39          rect.Y = layout.GetTop();
40
41          // выравнивание по горизонтали
42          switch (align)
43          {
44          case StringAlignmentNear: // по ближнему краю
45              rect.X = layout.GetLeft();
46              break;
47          case StringAlignmentCenter: // по центру
48              rect.X = layout.X + (layout.Width-rect.Width)/2.f;
49              break;
50          case StringAlignmentFar: // по дальнему краю
51              rect.X = layout.GetRight() - rect.Width;
52              break;
53          } // switch
54      } // else
55
56      rect.GetBounds(result); // возвращаем результат
57      return Ok;
58  } // MeasureImage
```

Функция, приведенная в этом примере, определяет также положение изображения в задаваемом прямоугольнике в зависимости от параметров выравнивания, т.к. в результате пропорционального масштабирования может оставаться неиспользованное пространство.

На рис. 1.42 представлены различные варианты выравнивания изображения для двух рассмотренных случаев соотношения изображения и прямоугольника.

Контуры

Контур (path) представляет собой последовательность графических примитивов (линий, прямоугольников, кривых, текста и т. п.), которые обрабатываются и отображаются как один объект. Контур можно разделить на отдельные фигуры, которые являются либо незамкнутыми, либо замкнутыми.

В GDI+ создание контуров выполняется с помощью методов класса `GraphicsPath`, определенного в заголовочном файле gdipluspath.h.

`StringAlignmentNear`

`StringAlignmentCenter`

`StringAlignmentFar`

Рис. 1.42. Варианты выравнивания изображения относительно границ прямоугольника

Класс GraphicsPath имеет следующие конструкторы:

```
GraphicsPath(FillMode fillMode = FillModeAlternate);

GraphicsPath(const Point *points, const BYTE *types,
    INT count, FillMode fillMode = FillModeAlternate);

GraphicsPath(const PointF *points, const BYTE *types,
    INT count, FillMode fillMode = FillModeAlternate);
```

Первый конструктор создает объект класса GraphicsPath на основе режима заполнения, который задает параметр *fillMode*.

Второй и третий конструкторы создают объект класса GraphicsPath на основе массива точек, на который указывает параметр *points*, а параметр *count* задает количество элементов в этом массиве. Параметр *types* указывает на массив типа INT, элементы которого определяют тип точек, указанных в массиве. Параметр *fillMode* задает режим заполнения.

Тип точек определяется одним (или комбинацией) из значений перечисляемого типа PathPointType (подробную информацию см. в документации Platform SDK).

Для построения контура в классе GraphicsPath есть множество методов схожих по названию с методами класса Graphics, предназначенных для рисования таких же фигур. В таблице 1.9 перечислены схожие методы класса GraphicsPath и класса Graphics.

Таблица 1.9. Сопоставление методов классов Graphics и GraphicsPath

Метод класса GraphicsPath	Аналог в классе Graphics	Описание
AddArc	DrawArc	Добавляет в контур дугу
AddBezier	DrawBezier	Добавляет в контур сплайн Безье
AddBeziers	DrawBeziers	Добавляет в контур кривую, состоящую из сплайнов Безье
AddClosedCurve	DrawClosedCurve	Добавляет в контур замкнутый кубический сплайн
AddCurve	DrawCurve	Добавляет в контур кубический сплайн
AddEllipse	DrawEllipse	Добавляет в контур эллипс
AddLine	DrawLine	Добавляет в контур линию
AddLines	DrawLines	Добавляет в контур ломаную линию
AddPath	DrawPath	Добавляет в контур другой контур
AddPie	DrawPie	Добавляет в контур сектор

Окончание табл. 1.9

Метод класса GraphicsPath	Аналог в классе Graphics	Описание
AddPolygon	DrawPolygon	Добавляет в контур полигон
AddRectangle	DrawRectangle	Добавляет в контур прямоугольник
AddRectangles	DrawRectangles	Добавляет в контур несколько прямоугольников
AddString	DrawString	Добавляет в контур строку

Каждый метод класса GraphicsPath, имеет несколько прототипов схожих с аналогичными методами класса Graphics (подробнее см. в документации Platform SDK).

Следует отметить, что в классе GraphicsPath, как и в других классах GDI+, есть методы GetLastStatus и Clone. Метод Clone имеет следующий прототип:

```
GraphicsPath* Clone() const;
```

В случае успеха метод Clone возвращает указатель на объект класса GraphicsPath, который должен быть удален вызовом оператора delete после того как станет не нужен. В случае ошибки – NULL.

Рисование контуров

Для рисования контуров используется метод DrawPath класса Graphics, а для закрашивания контуров используется метод FillPath этого класса. Эти методы имеют следующие прототипы:

```
Status DrawPath(const Pen *pen,
    const GraphicsPath *path);
Status FillPath(const Brush *brush,
    const GraphicsPath *path);
```

Параметр *path* указывает на объект класса GraphicsPath, который представляет отображаемый контур.

В листинге 1.33 представлен пример демонстрирующий рисование контура, изображенного на рис. 1.43.

Листинг 1.33. Пример рисования контура

```
1  void Display(HDC hdc)
2  {
3      // создаем объект класса Graphics для рисования
4      Graphics g(hdc);
5      // выполняем очистку перед рисованием
6      g.Clear(Color::White); // белый цвет фона
```

```
7
8       // создаем перо черного цвета
9       Pen blackPen(Color::Black);
10      // создаем сплошную кисть
11      SolidBrush solidBrush(Color::Yellow); // желтый цвет
12
13      Rect rect(70, 70, 100, 100);
14
15      // создаем объект класса GraphicsPath
16      // для построения контура
17      GraphicsPath path;
18
19      // добавляем в контур прямоугольник
20      path.AddRectangle(rect);
21
22      rect.Inflate(50, 50);
23
24      // добавляем в контур круг
25      path.AddEllipse(rect);
26
27      // рисуем закрашенный контур
28      g.FillPath(&solidBrush, &path);
29      // рисуем контур
30      g.DrawPath(&blackPen, &path);
31  } // Display
```

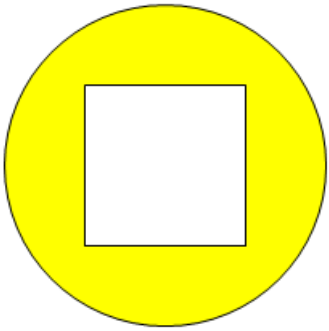


Рис. 1.43. Пример контура

Замкнутые контуры

Для создания замкнутого контура в классе `GraphicsPath` есть методы `StartFigure`, `CloseFigure` и `CloseAllFigures`.

Методы `StartFigure`, `CloseFigure` и `CloseAllFigures` имеют следующие прототипы:

```
Status StartFigure();
Status CloseFigure();
Status CloseAllFigures();
```

Метод `StartFigure` открывает новую фигуру в контуре, не замыкая при этом текущую фигуру. Все последующие примитивы, добавляемые к контуру, добавляются к этой новой фигуре.

Метод `CloseFigure` замыкает текущую фигуру и открывает новую фигуру. Если текущая фигура содержит последовательность соединенных линий, метод `CloseFigure` замыкает ее путем соединения начальной и конечной точек этих линий.

Метод `CloseAllFigures` замыкает все незамкнутые фигуры в контуре и открывает новую фигуру.

В листинге 1.34 приведен пример создания замкнутого контура прямоугольника с закругленными углами (например, как на рис. 1.43).

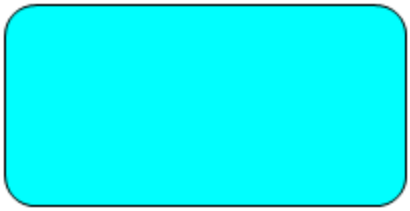


Рис. 1.44. Прямоугольник с закругленными краями

Листинг 1.34. Функция, создающая замкнутый контур

```
1   GraphicsPath* CreateRoundRectangle(const Rect &rect, int Radius)
2   {
3       // создаем обьект класса GraphicsPath
4       // для построения контура
5       GraphicsPath path;
6
7       // добавляем в контур линию
8       path.AddLine(rect.GetLeft()+Radius, rect.GetTop(),
    rect.GetRight()-Radius, rect.GetTop());
9       // добавляем в контур дугу
10      path.AddArc(rect.GetRight()-Radius, rect.GetTop(), Radius,
    Radius, 270.f, 90.f);
11      // добавляем в контур линию
12      path.AddLine(rect.GetRight(), rect.GetTop()+Radius,
    rect.GetRight(), rect.GetBottom()-Radius);
```

```
13      // добавляем в контур дугу
14      path.AddArc(rect.GetRight()-Radius, rect.GetBottom()-Radius,
    Radius, Radius, 0.f, 90.f);
15      // добавляем в контур линию
16      path.AddLine(rect.GetRight()-Radius, rect.GetBottom(),
    rect.GetLeft()+Radius, rect.GetBottom());
17      // добавляем в контур дугу
18      path.AddArc(rect.GetLeft(), rect.GetBottom() - Radius,
    Radius, Radius, 90.f, 90.f);
19      // добавляем в контур линию
20      path.AddLine(rect.GetLeft(), rect.GetBottom()-Radius,
    rect.GetLeft(), rect.GetTop()+Radius);
21      // добавляем в контур дугу
22      path.AddArc(rect.GetLeft(), rect.GetTop(), Radius, Radius,
    180.f, 90.f);
23
24      // добавляем закрываем фигуру в контуре
25      path.CloseFigure();
26
27      return path.Clone();  // клонируем объект класса GraphicsPath
28  } // CreateRoundRectangle
```

Функцию из приведенного примера можно использовать следующим образом:

```
1   Rect rect(30, 30, 200, 100);
2
3   // создаем контур
4   GraphicsPath *path = CreateRoundRectangle(rect, 20);
5
6   if (NULL != path)
7   {
8       /* контур успешно создан */
9   } // if
```

Когда объект класса `GraphicsPath`, на который указывает переменная *`path`* из приведенного примера, более станет не нужен, он должен быть удален с помощью оператора `delete`.

Поддержка полупрозрачности

В GDI+ можно создавать, как непрозрачные перья и кисти, так и полупрозрачные перья и кисти. При рисовании полупрозрачным пером или кистью выполняется альфа-смешивание, которое представляет собой смешение фонового цвета и цвета пера или кисти. Каждый из трех компонентов (красный, зеленый, синий) фонового цвета смеши-

вается с соответствующим компонентом цвета пера или кисти, согласно следующей формуле:

$$Color = SrcColor * \frac{alpha}{255} + BkndColor * \frac{255 - alpha}{255}, \quad (1.5)$$

где $Color$ – отображаем цвет; $BkndColor$ – фоновый цвет; $SrcColor$ – цвет пера или кисти, а $alpha$ – альфа-фактор цвета пера или кисти.

Для создания непрозрачного пера или кисти следует установить для цвета альфа-фактор равным 255. Чтобы создать полупрозрачное перо или кисть, нужно задать альфа-фактору любое значение из диапазона от 1 до 254.

В листинге 1.35 приведен пример альфа-смешивания при рисовании изображения, показанного на рис. 1.45.

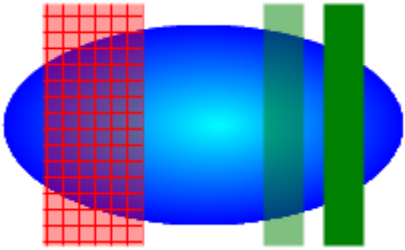


Рис. 1.45. Пример альфа-смешивания

Листинг 1.35. Пример рисования полупрозрачными графическими объектами

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       g.SetSmoothingMode(SmoothingModeHighQuality);
6       // выполняем очистку перед рисованием
7       g.Clear(Color::White); // белый цвет фона
8
9       Rect rect(10, 20, 200, 100);
10
11      // создаем эллиптический контур
12      GraphicsPath path;
13      path.AddEllipse(rect);
14
15      // создаем полупрозрачное перо
16      Pen pen1(Color(128, 0, 128, 0), 20.f);
17      // создаем непрозрачное перо
18      Pen pen2(Color(255, 0, 128, 0), 20.f);
```

```
19      // создаем штриховую кисть с полупрозрачным фоном
20      HatchBrush hatchBrush(HatchStyleCross, Color::Red, Color(100,
    255, 0, 0));
21
22      // создаем непрозрачную кисть градиента контура
23      PathGradientBrush pthGrBrush(&path);
24
25      // задаем цвет центра контура
26      pthGrBrush.SetCenterColor(Color::Aqua);
27      // задаем цвет границы контура...
28      Color color = Color::Blue;
29      int n = 1;
30      pthGrBrush.SetSurroundColors(&color, &n);
31
32
33      // рисуем непрозрачный контур
34      g.FillPath(&pthGrBrush, &path);
35      // рисуем полупрозрачный прямоугольник
36      g.FillRectangle(&hatchBrush, 30, 10, 50, 120);
37      // рисуем полупрозрачную линию
38      g.DrawLine(&pen1, 150, 10, 150, 130);
39      // рисуем непрозрачную линию
40      g.DrawLine(&pen2, 180, 10, 180, 130);
41  } // Display
```

Настройка графического вывода

Класс `Graphics` содержит также методы, которые способны влиять на качество отображаемых объектов.

Сглаживание линий и текста

При отображении различных линий, кривых и текста могут появляться контурные неровности. Для устранения контурных неровностей применяется *сглаживание* (smoothing). Обычно отображение более качественных сглаженных линии, кривых и текста требует больших затрат вычислительных ресурсов, чем менее качественных.

В классе `Graphics` есть метод `SetSmoothingMode`, который задает режим сглаживания линий. Этот метод имеет следующий прототип:

```
Status SetSmoothingMode(SmoothingMode smoothingMode);
```

Параметр *smoothingMode* задает режим сглаживания линий. Этот параметр может принимать одно из следующих значений перечисляемого типа `SmoothingMode`:

- `SmoothingModeDefault` – отображение без сглаживания линий;
- `SmoothingModeHighSpeed` – отображение без сглаживания линий;

- `SmoothingModeHighQuality` – отображение со сглаживанием линий;
- `SmoothingModeNone` – отображение без сглаживания линий;
- `SmoothingModeAntiAlias` – отображение со сглаживанием линий.

На рис. 1.46 изображен сплайн в двух вариантах без применения и с применением сглаживания линий.

`SmoothingModeNone` `SmoothingModeAntiAlias`

Рис. 1.46. Изображение сплайна и сглаживание

По умолчанию сглаживание линий не применяется. Определить текущий режим сглаживания линий можно с помощью метода `GetSmoothingMode` класса `Graphics`:

```
SmoothingMode GetSmoothingMode() const;
```

Режим сглаживания линий никак не влияет на качество отображения текста. Чтобы задать режим сглаживания текста, используется метод `SetTextRenderingHint` класса `Graphics`. Этот метод имеет следующий прототип:

```
Status SetTextRenderingHint(TextRenderingHint newMode);
```

Параметр *newMode* задает режим сглаживания текста. Этот параметр может принимать одно из следующих значений перечисляемого типа `TextRenderingHint`:

- `TextRenderingHintSystemDefault` – отображение текста с параметрами сглаживания, указанными для операционной системы;
- `TextRenderingHintSingleBitPerPixelGridFit` – отображение текста без сглаживания и с хинтованием;
- `TextRenderingHintSingleBitPerPixel` – отображение текста без сглаживания и без хинтования;
- `TextRenderingHintAntiAliasGridFit` – отображение текста со сглаживанием и хинтованием;
- `TextRenderingHintAntiAlias` – отображение текста со сглаживанием и без хинтования;

- `TextRenderingHintClearTypeGridFit` – отображение текста со сглаживанием текста по технологии ClearType, применяемой для ЖК-мониторов.

На рис. 1.47 приведены различные варианты сглаживания текста.

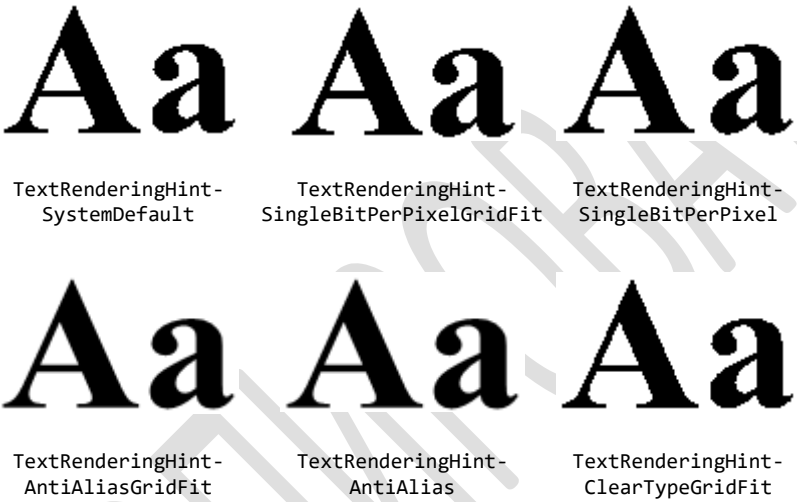


Рис. 1.47. Варианты сглаживания текста

Определить текущий режим сглаживания текста можно с помощью метода `GetTextRenderingHint` класса `Graphics`. Этот метод имеет следующий прототип:

```
TextRenderingHint GetTextRenderingHint() const;
```

Качество растровых изображений

При масштабировании растрового изображения происходит потеря качества этого изображения. В GDI+ реализованы различные алгоритмы интерполяции, позволяющие добиться улучшения качества масштабируемого изображения.

В классе `Graphics` определен метод `SetInterpolationMode`, который позволяет указать, какой алгоритм интерполяции будет использоваться при выводе растровых изображений с изменением его размеров.

Метод `SetInterpolationMode` имеет следующий прототип:

```
Status SetInterpolationMode(
    InterpolationMode interpolationMode);
```

Параметр *interpolationMode* задает режим интерполяции изображения. Этот параметр должен принимать одно из следующих значений перечисляемого типа InterpolationMode:

- InterpolationModeDefault – интерполяция по умолчанию;
- InterpolationModeLowQuality – низкокачественная интерполяция;
- InterpolationModeHighQuality – высококачественная интерполяция;
- InterpolationModeBilinear – билинейная интерполяция (не используется для сжатия изображения до размера менее 50 % от его исходного размера);
- InterpolationModeBicubic – бикубическая интерполяция (не используется для сжатия изображения до размера менее 25 % от его исходного размера);
- InterpolationModeNearestNeighbor – интерполяция по ближайшим соседним значениям;
- InterpolationModeHighQualityBilinear – высококачественная билинейная интерполяция;
- InterpolationModeHighQualityBicubic – высококачественная бикубическая интерполяция.

На рис. 1.49 представлены различные варианты интерполяции увеличенного в три раза изображения, показанного на рис.1.48.

Рис. 1.48. Изображение для масштабирования

Определить текущий режим интерполяции изображения можно с помощью метода GetInterpolationMode класса Graphics. Этот метод имеет следующий формат:

```
InterpolationMode GetInterpolationMode() const;
```

Логические единицы

Класс Graphics позволяет абстрагироваться от физических характеристик устройства, позволяя указать, в какой логической системе координат будет происходить графический вывод.

InterpolationModeLowQuality

InterpolationModeHighQuality

InterpolationModeBilinear

InterpolationModeHighQualityBilinear

InterpolationModeBicubic

InterpolationModeHighQualityBicubic

Рис. 1.49. Различные варианты интерполяции изображения

Для установки логической системы координат в классе Graphics имеется метод SetPageUnit. Этот метод имеет следующий прототип:

```
Status SetPageUnit(Unit unit);
```

Параметр *unit* указывает логические единицы устройства отображения. Этот параметр должен принимать одно из следующих перечисляемого типа Unit:

- UnitWorld – мировые (не физические) единицы;
- UnitDisplay – дисплейные единицы (пиксели);
- UnitPixel – пиксели;
- UnitPoint – 1 единица равна $^1/_{72}$ дюйма;
- UnitInch – дюймы;
- UnitDocument – 1 единица равна $^1/_{300}$ дюйма;
- UnitMillimeter – миллиметры.

Определить, какие используются логические единицы можно с помощью метода GetPageUnit класса Graphics:

```
Unit GetPageUnit() const;
```

Кроме того, можно задать коэффициент масштабирования логических единиц. Для этого в классе Graphics существует метод SetPageScale. Этот метод имеет следующий прототип:

```
Status SetPageScale(REAL scale);
```

Параметр *scale* определяет коэффициент масштабирования.

В следующем небольшом примере продемонстрированно, как задать в качестве логической единицы 1 см.

```
1  Graphics g(hdc);
2  // выбираем миллиметры в качестве логических единиц
3  g.SetPageUnit(UnitMillimeter);
4  // устанавливаем масштабирование логических единиц
5  g.SetPageScale(10.f);
```

Определить текущий коэффициент масштабирования логических единиц можно с помощью метода GetPageScale класса Graphics:

```
REAL GetPageScale() const;
```

Использование GDI+ в приложениях Win32

Библиотека GDI+ полностью реализована в DLL-библиотеке GdiPlus.dll, для получения доступа к которой следует подключить к программе библиотеку импорта GdiPlus.lib и заголовочный файл gdiplus.h.

Кроме того, библиотека GDI+ прибывает в собственном пространстве имен `Gdiplus`. По этой причине, чтобы открыть прямой доступ к функциям и классам, определенным внутри GDI+, необходимо использовать следующую директиву:

```
using namespace Gdiplus;
```

Инициализация и завершение работы GDI+

Перед тем как использовать классы и функции GDI+ следует инициализировать эту библиотеку с помощью функции `GdiplusStartup`, которая должна быть первой из функций GDI+, вызываемых в программе. Функция `GdiplusStartup` имеет следующий прототип:

```
Status GdiplusStartup(ULONG_PTR *token,
    const GdiplusStartupInput *input,
    GdiplusStartupOutput *output);
```

Первый параметр, *`token`*, указывает на переменную типа `ULONG_PTR`, в которую сохраняется маркер, необходимый для завершения работы с GDI+.

Второй параметр, *`input`*, указывает на структуру `GdiplusStartupInput`, поля которой управляют различными аспектами использования библиотеки GDI+.

Третий параметр, *`output`*, указывает на структуру `GdiplusStartupOutput`. Это поле может быть установлено в `NULL`.

После выполнения функция `GdiplusStartup` возвращает одно из значений перечисляемого типа `Status`, определенного в заголовочном файле GdiPlusTypes.h. В случае успешного выполнения эта функция возвращается значение `Ok`.

Структура `GdiplusStartupInput` имеет следующее описание:

```
struct GdiplusStartupInput
{
   UINT32 GdiplusVersion; // версия
   DebugEventProc DebugEventCallback; // функция отладки
   BOOL SuppressBackgroundThread; // флаг, разрешающий
                                  // отложенную инициализацию
   BOOL SuppressExternalCodecs; // флаг, разрешающий
                                // использование внешних
                                // кодеков изображений
};
```

Первое поле, *`GdiplusVersion`*, задает версию GDI+. Это поле должно принимать значение `1`, которое соответствует версии 1.0.

Второе поле, *DebugEventCallback*, указывает на функцию, которая будет вызываться при возникновении ошибки в различных функциях GDI+. Это поле может быть установлено в NULL.

Третье поле, *SuppressBackgroundThread*, определяет, необходимо ли создавать фоновый поток (например, когда начинаются манипуляции с изображениями) для окончательной инициализации GDI+. При этом инициализируется только часть библиотеки GDI+ (такой механизм отложенной инициализации довольно распространен в библиотеках Microsoft). Если поле *SuppressBackgroundThread* принимает значение TRUE, то функция GdiplusStartup возвращает (в параметре *output*) указатель на специальные функции, которые должны быть вызваны в том же потоке, что и функция GdiplusStartup.

Четвертое поле, *SuppressExternalCodecs*, определяет, необходимо ли использовать внешние кодеки изображений. Так как GDI+ версии 1.0 не поддерживает внешние кодеки изображений, поэтому значения этого поля игнорируются.

Следует отметить, что конструктор структуры GdiplusStartupInput по умолчанию выполняет инициализацию ее полей, достаточную для большинства программ. По умолчанию поле *GdiplusVersion* принимает значение равное 1, поле *DebugEventCallback* установлено в NULL, а поля *SuppressBackgroundThread* и *SuppressExternalCodecs* принимают значение равное FALSE.

Функция, на которую указывает поле *DebugEventCallback*, может называться как угодно, но должна имеет следующую сигнатуру:

```
void WINAPI DebugEventProc(DebugEventLevel level,
    CHAR *message);
```

Первый параметр, *level*, определяет уровень сообщения отладки и может принимать одно из следующих значений:

- DebugEventLevelFatal – критическая ошибка;
- DebugEventLevelWarning – предупреждение.

Второй параметр, *message*, указывает на строку, в которой содержится текст сообщения отладки.

Структура GdiplusStartupOutput имеет следующее описание:

```
struct GdiplusStartupOutput
{
   NotificationHookProc NotificationHook;
   NotificationUnhookProc NotificationUnhook;
};
```

Поля этой структуры, *NotificationHook* и *NotificationUnhook*, указывают на функции, которые нужно вызвать, соответственно, до и после цикла обработки оконных сообщений в приложении Win32.

Функция, на которую указывает поле *NotificationHook*, имеет следующую сигнатуру:

```
Status WINAPI NotificationHookProc(ULONG_PTR *token);
```

Параметр *token* указывает на переменную типа ULONG_PTR, в которую будет сохранен маркер, необходимый для вызова функции, на которую указывает поле *NotificationUnhook*.

Функция, на которую указывает поле *NotificationUnhook*, имеет следующую сигнатуру:

```
void WINAPI NotificationUnhookProc(ULONG_PTR token);
```

Когда больше нет необходимости в использовании функций и классов GDI+, следует вызвать функцию GdiplusShutdown:

```
void GdiplusShutdown(ULONG_PTR token);
```

Здесь в качестве параметра необходимо передать маркер, который вернула функция *GdiplusStartup* через параметр *token*.

Следующий пример демонстрирует фрагмент программного кода инициализации и завершения работы с GDI+.

Листинг 1.36. Пример инициализации и завершения работы с GDI+

```
1   ULONG_PTR           gdiplusToken;
2   GdiplusStartupInput  gdiplusStartupInput;
3   GdiplusStartupOutput gdiplusStartupOutput;
4
5   // указываем функцию отладки
6   gdiplusStartupInput.DebugEventCallback = GdiplusDebugProc;
7   // разрешаем отложенную инициализацию
8   gdiplusStartupInput.SuppressBackgroundThread = TRUE;
9
10  // инициализация GDI+
11  Status stRet = GdiplusStartup(&gdiplusToken,
        &gdiplusStartupInput, &gdiplusStartupOutput);
12
13  if (Ok == stRet)
14  {
15      ULONG_PTR token;
16      gdiplusStartupOutput.NotificationHook(&token);
17
18      /* цикл обработки оконных сообщений */
19
```

```
20      gdiplusStartupOutput.NotificationUnhook(token);
21
22      GdiplusShutdown(gdiplusToken); // завершение работы GDI+
23  } // if
```

В этом примере используется отложенная инициализация GDI+ и кроме того при дальнейшей отладке программы используется функция обратного вызова GdiplusDebugProc, которая например может быть такой:

```
1   void WINAPI GdiplusDebugProc(DebugEventLevel level, CHAR
    *message)
2   {
3       switch (level)
4       {
5       case DebugEventLevelFatal:
6           MessageBoxA(NULL, message, "Критическая ошибка",
    MB_OK|MB_ICONERROR|MB_TASKMODAL);
7           break;
8       case DebugEventLevelWarning:
9           MessageBoxA(NULL, message, "Предупреждение",
    MB_OK|MB_ICONWARNING|MB_TASKMODAL);
10          break;
11      } // switch
12  } // GdiplusDebugProc
```

Однако в большинстве программ инициализация GDI+ выполняется проще. Следующий пример демонстрирует фрагмент программного кода более простой инициализации и завершения работы GDI+:

Листинг 1.37. Пример простой инициализации и завершения работы с GDI+

```
1   ULONG_PTR          gdiplusToken;
2   GdiplusStartupInput gdiplusStartupInput;
3
4   // инициализация GDI+
5   Status stRet = GdiplusStartup(&gdiplusToken,
        &gdiplusStartupInput, NULL);
6
7   if (Ok == stRet)
8   {
9       /* цикл обработки оконных сообщений */
10
11      GdiplusShutdown(gdiplusToken); // завершение работы GDI+
12  } // if
```

Контекст устройства

Контекст устройства (device context) представляет собой внутреннюю структуру данных, содержащую информацию о параметрах и атрибутах графического вывода на устройство отображения.

Прежде чем рисовать на устройстве отображения необходимо получить *дескриптор контекста устройства* (handle device context), чтобы идентифицировать устройство, на котором должно выполняться рисование.

В GDI поддерживаются следующие типы контекста устройства:

- контекст экрана монитора (display context);
- контекст принтера (printer context);
- контекст виртуального устройства в памяти (memory context);
- контекст метафайла (metafile context);
- информационный контекст (information context).

Конечно, чаще всего используется контекст экрана монитора. Для получения дескриптора этого контекста Win32 API предоставляет два метода. Первый метод заключается в обработке оконного сообщения `WM_PAINT`, а второй – в использовании функций `GetDC` и `GetWindowDC`.

Обработка оконного сообщения WM_PAINT

Обычно приложение рисует в клиентской области окна, реагируя на сообщение `WM_PAINT`, которое посылается окну всякий раз, когда требуется перерисовать клиентскую область окна. Например, типичными причинами отправки сообщения `WM_PAINT` окну могут быть следующие события:

- изменились размеры или местоположение окна;
- клиентская область окна была частично или полностью закрыта другим окном, но теперь эта область открыта.

При обработке оконного сообщения `WM_PAINT` дескриптор контекста устройства получают вызовом функции `BeginPaint`:

```
HDC BeginPaint(HWND hwnd, LPPAINTSTRUCT lpPaint);
```

При успешном завершении эта функция возвращает дескриптор контекста устройства для рисования в клиентской области окна, на которое указывает параметр *`hwnd`*. Параметр *`lpPaint`* указывает на структуру `PAINTSTRUCT`, в которую сохраняется вся необходимая для рисования информация.

Функция `BeginPaint` не должна вызваться нигде кроме, как при обработке оконного сообщения `WM_PAINT`. При этом у каждого вызова

функции BeginPaint должен быть соответствующий вызов функции EndPaint, которая завершает процесс рисования и освобождает контекст устройства.

Функция EndPaint имеет следующий прототип:

```
BOOL EndPaint(HWND hWnd, const PAINTSTRUCT *lpPaint);
```

Первый параметр, *hWnd*, – дескриптор окна, в клиентской области которого выполнялось рисование.

Второй параметр, *lpPaint*, указывает на структуру PAINTSTRUCT, которая содержит информацию для рисования, извлекаемую при помощи функции BeginPaint.

Если функция EndPaint завершается успешно, возвращаемое значение отлично от FALSE.

Структура PAINTSTRUCT имеет следующее описание:

```
typedef struct tagPAINTSTRUCT {
  HDC  hdc; // дескриптор контекста устройства
  BOOL fErase; // признак удаления фона клиентской области
  RECT rcPaint; // границы прямоугольной области, которая
                // должна быть перерисована
  BOOL fRestore; // зарезервировано
  BOOL fIncUpdate; // зарезервировано
  BYTE rgbReserved[32]; // зарезервировано
} PAINTSTRUCT, *PPAINTSTRUCT;
```

Первое поле, *hdc*, содержит дескриптор контекста устройства, который используется для рисования.

Второе поле, *fErase*, определяет, будет ли обновляться фон в области рисования. Если это поле принимает значение TRUE, которое означает обновление фона, функция BeginPaint посылает соответствующему окну сообщение WM_ERASEBKGND. Приложение может обрабатывать сообщение WM_ERASEBKGND, но обычно оно обрабатывается по умолчанию функцией DefWindowProc, которая обновляет фон с использованием кисти, указанной в поле *hbrBackground* при регистрации оконного класса.

Третье поле, *rcPaint*, определяет структуру RECT, поля которой задают границы прямоугольной области, которая должна быть перерисована. Границы этой области задаются в координатах клиентской области, в которой выполняется рисование.

Последние три поля (*fRestore*, *fIncUpdate* и *rgbReserved*) зарезервированы и используются операционной системой.

В листинге 1.38 представлен пример типичной обработка оконного сообщения WM_PAINT. Заметим, что в этом примере используется функция Display, в которой должно выполняться рисование.

Листинг 1.38. Пример обработки сообщения WM_PAINT

```
case WM_PAINT:
    {
        PAINTSTRUCT ps;
        // начинаем процесс рисования
        HDC hdc = BeginPaint(hWnd, &ps);

        Display(hdc); // вызываем фукнцию отображения

        // завершаем процесс рисования
        EndPaint(hWnd, &ps);
    }
    return 0;
```

Отправка сообщения WM_PAINT

В некоторых случаях причиной отправки сообщения WM_PAINT являются не внешние события, а например вызов функции InvalidateRect, которая указывает часть клиентской области окна, требующую обновления. Эта часть клиентской области окна называется *недействительный регион* (invalid region).

Функция InvalidateRect имеет следующий прототип:

```
BOOL InvalidateRect(HWND hWnd, const RECT *lpRect,
  BOOL bErase);
```

Первый параметр, *hWnd*, – дескриптор окна, клиентскую область которого необходимо перерисовать. Если этот параметр установлен в NULL, будут перерисованы все окна приложения.

Второй параметр, *lpRect*, указывает на структуру RECT, содержащую границы прямоугольной области, которая задается как недействительный регион. Если этот параметр установлен в NULL, то обновляется вся клиентская область окна.

Третий параметр, *bErase*, определяет, будет ли обновляться фон недействительного региона. Если этот параметр принимает значение TRUE, фон обновляется.

Если функция InvalidateRect завершается успешно, она возвращает значение отличное от FALSE. При этом указанное окно получает сообщение WM_PAINT.

Использование функций GetDC и GetWindowDC

Иногда рисование должно выполняться не при обработке сообщения `WM_PAINT`, а при обработке других оконных сообщений. В таких случаях дескриптор контекста устройства получают с помощью вызова одной из двух функций – `GetDC` и `GetWindowDC`:

```
HDC GetDC(HWND hWnd);
HDC GetWindowDC(HWND hWnd);
```

При успешном выполнении функция `GetDC` возвращает дескриптор контекста устройства для клиентской области окна, на которое указывает параметр *hWnd*, а функция `GetWindowDC` – для всего окна. Если параметр *hWnd* установлен в `NULL`, то обе функции вернут дескриптор контекста устройства для всего экрана. В случае ошибки обе функции возвращают `NULL`.

По окончании работы с контекстом устройства, его необходимо освободить с помощью вызова функции `ReleaseDC`:

```
int ReleaseDC(HWND hWnd, HDC hDC);
```

Первый параметр, *hWnd*, – дескриптор окна, контекст устройства которого необходимо освободить.

Второй параметр, *hDC*, – дескриптор освобождаемого контекста устройства, который должен быть получен с помощью одной из двух функций – `GetDC` и `GetWindowDC`.

При успешном завершении функция `ReleaseDC` возвращает значение равное `1`, в случае ошибки – `0`.

Задание к работе

1. Изучить возможности графического вывода с помощью GDI+.
2. Разработать в Visual C++ оконное приложение Win32, на главном окне которого:
   - должен отображаться рисунок, состоящий из линий, кривых, многоугольников и закрашенных областей, в соответствии с вариантом задания;
   - должно выводиться растровое изображение рисунка, указанного в варианте задания;
   - должно выводиться название рисунка, указанного в варианте задания.

Обязательно использовать графические объекты, указанные в варианте задания. При необходимости можно также использовать и другие графические объекты.

3. Протестировать работу приложения, разработанного в п. 2. Результаты тестирования отразить в отчете.
4. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Рисунок	Графические объекты
1,16	Автомобиль	Кисти линейного градиента Составные перья Сплошные кисти
2,17	Паровоз	Кисти градиента контура Перья с текстурным заполнением Штриховые кисти
3,18	Пароход	Составные перья Сплошные кисти Текстурные кисти
4,19	Самолет	Кисти линейного градиента Перья с текстурным заполнением Штриховые кисти
5,20	Ракета	Кисти градиента контура Составные перья Сплошные кисти
6,21	Автобус	Перья с текстурным заполнением Текстурные кисти Штриховые кисти
7,22	Луноход	Кисти линейного градиента Составные перья Сплошные кисти
8,23	Трактор	Кисти градиента контура Составные перья Штриховые кисти
9,24	Грузовик	Перья с текстурным заполнением Сплошные кисти Текстурные кисти

№	Рисунок	Графические объекты
10,25	Катер	Кисти линейного градиента Составные перья Штриховые кисти
11,26	Вертолет	Кисти градиента контура Перья с текстурным заполнением Сплошные кисти
12,27	Велосипед	Составные перья Текстурные кисти Штриховые кисти
13,28	Мотоцикл	Кисти линейного градиента Перья с текстурным заполнением Сплошные кисти
14,29	Парусник	Штриховые кисти Кисти градиента контура Составные перья
15,30	Звездолет	Перья с текстурным заполнением Сплошные кисти Текстурные кисти

Контрольные вопросы

1. Что в Windows называют интерфейсом графического устройства?
2. Для чего предназначены библиотеки GDI и GDI+?
3. Как работать с цветом в GDI+?
4. Для чего используются графические объекты GDI+?
5. Как в GDI+ создать сплошную кисть?
6. Как в GDI+ создать штриховую кисть?
7. Как в GDI+ создать текстурную кисть?
8. Как в GDI+ создать кисть линейного градиента?
9. Как в GDI+ создать кисть линейного градиента, определяемого несколькими цветами?
10. Как в GDI+ создать кисть градиента контура?
11. Как в GDI+ создать перо?
12. Как в GDI+ создать составное перо?
13. Как в GDI+ создать перо с текстурным наполнением?
14. Как в GDI+ создать шрифт?
15. Как в GDI+ загрузить изображение?
16. Как в GDI+ выполняется графический вывод?
17. Как в GDI+ выполняется очистка области отображения?
18. Как в GDI+ выполняется рисование линий?

19. Как в GDI+ выполняется рисование прямоугольников?
20. Как в GDI+ выполняется рисование многоугольников?
21. Как в GDI+ выполняется рисование эллипсов?
22. Как в GDI+ выполняется рисование дуг?
23. Как в GDI+ выполняется рисование секторов?
24. Как в GDI+ выполняется рисование сплайнов? Какие сплайны можно рисовать в GDI+?
25. Как в GDI+ выполняется обычный вывод текста?
26. Как в GDI+ выполняется вывод графических изображений?
27. Как в GDI+ создать контур?
28. Как в GDI+ выполняется рисование контуров?
29. Как в GDI+ создать полупрозрачные перья и кисти?
30. Как в GDI+ улучшить качество отображаемых объектов?
31. Как в GDI+ изменить логические единицы?
32. Как выполняется инициализация и завершение работы с библиотекой GDI+ в приложении Win32?
33. Что в Windows называют контекстом устройства? Какие типы контекстов устройства поддерживаются?
34. Что такое дескриптор контекста устройства? Как получить дескриптор контекста устройства?

ЛАБОРАТОРНАЯ РАБОТА № 2. ОСНОВНЫЕ ЗАДАЧИ КОМПЬЮТЕРНОЙ ГЕОМЕТРИИ

Цель работы

Получение практических навыков применения различных математических инструментов для решения основных задач компьютерной геометрии.

Основные понятия

Компьютерная геометрия представляет собой математический аппарат, составляющий основу компьютерной графики. В свою очередь основу компьютерной геометрии составляют аналитическая геометрия, вычислительная математика, а также линейная и векторная алгебра. К основным задачам компьютерной геометрии можно отнести следующие задачи:

- преобразование из мировых координат в координаты порта просмотра (window-to-viewport mapping);
- определение принадлежности точки полигону (point in polygon);
- отсечение (clipping).

Математические инструменты

В данной главе рассматриваются примеры различных математических функции и классов, которые образуют математическую библиотеку, используемую при программировании компьютерной графики.

Нужно отметить, что при реализации функций и методов классов математической библиотеки используется директива `inline`, которая позволяет сделать программу быстрее. Директива `inline` указывает компилятору на необходимость размещения кода функции (или метода класса) непосредственно в месте ее вызова.

Сравнение значений с плавающей точкой

Достаточно распространенной операцией является проверка на равенство значений с плавающей точкой. Однако непосредственное сравнение значений с плавающей точкой некорректно по причине потери точности после выполнения различных вычислений. Поэтому при сравнении значений с плавающей точкой необходимо определить, как близко эти значения расположены друг к другу. При определении бли-

зости сравниваемых значений используется *допустимое отклонение* (*tolerance*), которое определяет точность производимых вычислений.

В следующем небольшом примере продемонстрировано сравнение нулевого значения с некоторым значением x (значение с плавающей точкой) при допустимом отклонении 10^{-6}.

```
if (abs(x) < 1e-6f) // -0.000001 < x < +0.000001
{
    /* x достаточно близко к нулю, то есть x ≈ 0 */
} // if
```

В листинге 2.1 приведены функции, которые выполняют основные операции сравнения значений с плавающей точкой.

Листинг 2.1. Функции сравнения значений с плавающей точкой

```
// Функция IsZero возвращает true, если x ≈ 0; в противном случае – false
inline bool IsZero(float x, float tolerance = 1e-6f)
{
    return (abs(x) < tolerance);
} // IsZero

// Функция Equals возвращает true, если a ≈ b; в противном случае – false
inline bool Equals(float a, float b, float tolerance = 1e-6f)
{
    return (abs(a - b) < tolerance);
} // Equals

// Функция Less возвращает true, если a < b; в противном случае – false
inline bool Less(float a, float b, float tolerance = 1e-6f)
{
    return (b - a) > tolerance;
} // Less

// Функция Greater возвращает true, если a > b; в противном случае – false
inline bool Greater(float a, float b, float tolerance = 1e-6f)
{
    return (a - b) > tolerance;
} // Greater

// Функция LessOrEquals возвращает true, если a ≤ b; в противном случае – false
inline bool LessOrEquals(float a, float b, float tolerance = 1e-6f)
{
```

```
28      return !((a - b) > tolerance);
29  } // LessOrEquals
30
31  // Функция GreaterOrEquals возвращает true, если a ≥ b; в
    противном случае – false
32  inline bool GreaterOrEquals(float a, float b, float tolerance =
    1e-6f)
33  {
34      return !((b - a) > tolerance);
35  } // GreaterOrEquals
36
37  // Функция Max возвращает наибольшее из a и b
38  inline float Max(float a, float b, float tolerance = 1e-6f)
39  {
40      return ((a - b) > tolerance) ? a : b;
41  } // Max
42
43  // Функция Min возвращает наименьшее из a и b
44  inline float Min(float a, float b, float tolerance = 1e-6f)
45  {
46      return ((b - a) > tolerance) ? a : b;
47  } // Min
```

В представленных функциях по умолчанию используется допустимое отклонение 10^{-6}. Однако, в зависимости от требуемой точности, можно использовать любые значения допустимого отклонения. В следующем примере показано, как значение допустимого отклонения влияет на результат сравнения двух значений с плавающей точкой:

```
1   float a = 1.04f, b = 1.05f;
2
3   // сравнение при допустимом отклонении по умолчанию
4   Equals(a, b); // результат a ≠ b
5   // сравнение при допустимом отклонении 0.001
6   Equals(a, b, 1e-3f); // результат a ≠ b
7   // сравнение при допустимом отклонении 0.1
8   Equals(a, b, 1e-1f); // результат a ≈ b
```

Арифметический сопроцессор и расширение SSE

В некоторых функциях математической библиотеки используются различные команды *арифметического сопроцессора* (Floating Point Unit, FPU), которые значительно ускоряют вычисления, связанные с вещественными числами. Кроме того используется расширение SSE (Streaming SIMD Extensions), которое дает возможность одновременно обрабатывать сразу четыре значения с плавающей точкой, что также может значительно ускорить вычисления.

Вычисление тригонометрических функций

Хорошим примером использования команд арифметического сопроцессора является вычисление значений тригонометрических функций. В компьютерной геометрии часто применяются тригонометрические вычисления.

В листинге 2.2 приводится программный код функций, которые вычисляют значения основных тригонометрических функций с помощью соответствующих команд трансцендентных функций арифметического сопроцессора: `FSIN`, `FCOS`, `FSINCOS` и `FPTAN`. Эти команды вычисляют значения синуса, косинуса и тангенса немного быстрее, чем функции `sin`, `cos` и `tan` из стандартной библиотеки C.

Листинг 2.2. Функции вычисления синуса, косинуса и тангенса

```
// Функция Sin вычисляет значение синуса для аргумента angle
inline float Sin(float angle)
{
    __asm
    {
        fld   angle
        fsin
    }
} // Sin

// Функция Cos вычисляет значение косинуса для аргумента angle
inline float Cos(float angle)
{
    __asm
    {
        fld   angle
        fcos
    }
} // Cos

// Функция SinCos вычисляет значения синуса и косинуса для
аргумента angle
inline void __fastcall SinCos(float angle, float *sin, float
*cos)
{
    __asm
    {
        fld     angle
        fsincos
        fstp    dword ptr [edx] // cos
        fstp    dword ptr [ecx] // sin
    }
} // SinCos

// Функция Tan вычисляет значение тангенса для аргумента angle
```

```
34  inline float Tan(float angle)
35  {
36      __asm
37      {
38          fld   angle
39          fptan
40          fdiv
41      }
42  } // Tan
```

Следует отметить что, функция `SinCos` выполняется быстрее, чем последовательный вызов функций `Sin` и `Cos`. Поэтому для вычисления синуса и косинуса одного угла (что бывает довольно часто), лучше использовать функцию `SinCos`.

Вычисление обратных тригонометрических функций

Нередко приходится сталкиваться с задачей поиска угла, если известно значение тригонометрической функции, примененной к этому углу. Для этого используют обратные тригонометрические функции.

В листинге 2.3 приводится программный код функций, которые вычисляют значения арксинуса, арккосинуса и арктангенса с помощью команды арифметического сопроцессора `FPATAN`. Тот факт, что команда `FPATAN` принимает два аргумента и возвращает арктангенс их отношения, упрощает вычисление других обратных тригонометрических функций с помощью следующих формул:

$$\arcsin x = \operatorname{arctg}\frac{x}{\sqrt{1-x^2}}, \qquad \arccos x = 2\operatorname{arctg}\sqrt{\frac{1-x}{1+x}}. \qquad (2.1)$$

Листинг 2.3. Функции вычисления арксинуса, арккосинуса и арктангенса

```
1   // Функция ArcSin вычисляет значение арксинуса
2   inline float ArcSin(float x)
3   {
4       __asm
5       {
6           fld    x  // a = x
7
8           fld    x
9           fmul   x
10          fld1
11          fsub   ST,ST(1)
12          fsqrt  // b = √(1 - x*x)
13
14          fpatan // вычисляем arctg(a/b)
15      }
```

```
} // ArcSin

// Функция ArcCos вычисляет значение арккосинуса
inline float ArcCos(float x)
{
    const float two = 2.f;

    __asm
    {
        fld1
        fsub   x
        fsqrt  // a = √(1 - x)

        fld1
        fadd   x
        fsqrt  // b = √(1 + x)

        fpatan // вычисляем arctg(a/b)
        fmul   two
    }
} // ArcCos

// Функция ArcTan вычисляет значение арктангенса
inline float ArcTan(float x)
{
    __asm
    {
        fld   x // a = x
        fld1    // b = 1

        fpatan  // вычисляем arctg(a/b)
    }
} // ArcTan
```

Функции `ArcSin`, `ArcCos` и `ArcTan` вычисляют значения обратных тригонометрических функций достаточно быстро и даже быстрее, чем функции `asin`, `acos` и `atan` из стандартной библиотеки C. Однако, несмотря на это, перед тем как проводить вычисления обратных тригонометрических функций, следует убедиться в том, что действительно необходимо знать значение угла. Очень часто оказывается, что достаточно знать только значения x (например, заранее известно, что при $x = 0$ арккосинус имеет значение $\pi/2$).

Радианы и градусы

Все представленные ранее тригонометрические и обратные тригонометрические функции используют *радианы* (*radians*). Если же необходимо использовать *градусы* (*degree*), придется выполнять преобра-

зования между градусами и радианами. Для того чтобы перевести градусы в радианы, можно воспользоваться следующей формулой:

$$\alpha_{\text{рад}} = \alpha^\circ \cdot \frac{\pi}{180^\circ}, \tag{2.2}$$

а для обратного преобразования:

$$\alpha^\circ = \alpha_{\text{рад}} \cdot \frac{180^\circ}{\pi}. \tag{2.3}$$

В листинге 6.9 показан программный код, в котором продемонстрированно применение формул (6.17-6.18) в реализации функций преобразования между радианами и градусами.

Листинг 2.4. Функции преобразования между радианами и градусами

```
1   // Функция RadToDeg возвращает значение, выраженное в градусах,
    для заданного в радианах аргумента arg
1   inline float RadToDeg(float arg)
2   {
3       const float mult = 180.f;
4
5       __asm
6       {
7           fld   arg
8           fmul  mult
9           fldpi
10          fdivp ST(1), ST
11      }
12  } // RadToDeg
13
14  // Функция DegToRad возвращает значение, выраженное в радианах,
    для заданного в градусах аргумента arg
15  inline float DegToRad(float arg)
16  {
17      const float denom = 180.f;
18
19      __asm
20      {
21          fldpi
22          fmul  arg
23          fdiv  denom
24      }
25  } // DegToRad
```

Извлечение квадратного корня

В качестве примера использования команд SSE можно рассмотреть пример извлечения квадратного корня. Извлечение квадратного

корня довольно частая операция в компьютерной геометрии. Поэтому необходимо, чтобы эта операция выполнялась как можно быстрее.

В листинге 2.5 продемонстрированы функции извлечения квадратных корней с помощью соответствующих команд SSE.

Листинг 2.5. Функция извлечения квадратного корня

```
1   // Функция SquareRoot возвращает кв. корень значения arg
2   inline float SquareRoot(float arg)
3   {
4       float result;
5
6       __asm
7       {
8           movss   xmm0, arg
9           sqrtss  xmm1, xmm0
10          movss   result, xmm1
11      }
12
13      return result;
14  } // SquareRoot
15
16  // Функция SquareRoot2 возвращает кв. корни значений из массива
    arg
17  inline void __fastcall SquareRoot2(const float arg[2], float
    result[2])
18  {
19      __asm
20      {
21          xorps   xmm0, xmm0 // обнуление значения регистра XMM0
22          movlps  xmm0, xmmword ptr [ecx]
23          sqrtps  xmm1, xmm0
24          movlps  xmmword ptr [edx], xmm1
25      }
26  } // SquareRoot2
27
28  // Функция SquareRoot4 возвращает кв. корни значений из массива
    arg
29  inline void __fastcall SquareRoot4(const float arg[4], float
    result[4])
30  {
31      __asm
32      {
33          movups  xmm0, xmmword ptr [ecx]
34          sqrtps  xmm1, xmm0
35          movups  xmmword ptr [edx], xmm1
36      }
37  } // SquareRoot4
```

Функция SquareRoot работает намного быстрее, чем аналогичная функция sqrt из стандартной библиотеки C. Кроме того, функции SquareRoot2 и SquareRoot4 выполняются быстрее, чем последовательный вызов функции SquareRoot. Поэтому если возникнет необходимость в извлечении сразу нескольких квадратных корней, лучше воспользоваться функцией SquareRoot2 или SquareRoot4.

Математические константы

В стандартной библиотеке C имеется несколько математических констант (табл. 2.1), которые могут быть весьма полезны при реализации различных алгоритмов компьютерной геометрии. По умолчанию эти математические константы не доступны, так как они не определены в стандарте C/C++. Чтобы их использовать, необходимо определить _USE_MATH_DEFINES до подключения заголовочного файла math.h:

```
1   #define _USE_MATH_DEFINES
2   #include <math.h>
```

Таблица 2.1. Математические константы стандартной библиотеки C

Константа	Выражение	Значение
M_E	e	2,71828182845904523536
M_LOG2E	$\log_2(e)$	1,44269504088896340736
M_LOG10E	$\log_{10}(e)$	0,434294481903251827651
M_LN2	$\ln(2)$	0,693147180559945309417
M_LN10	$\ln(10)$	2,30258509299404568402
M_PI	π	3,14159265358979323846
M_PI_2	$\pi/2$	1,57079632679489661923
M_PI_4	$\pi/4$	0,785398163397448309616
M_1_PI	$1/\pi$	0,318309886183790671538
M_2_PI	$2/\pi$	0,636619772367581343076
M_2_SQRTPI	$2/\sqrt{\pi}$	1,12837916709551257390
M_SQRT2	$\sqrt{2}$	1,41421356237309504880
M_SQRT1_2	$\frac{1}{\sqrt{2}}$	0,707106781186547524401

Также полезно будет иметь константу для величины 2π, которой нет в стандартной библиотеке C, но которая может пригодиться:

```
#define M_2PI  6.283185307179586476925
```

Следующий набор констант может быть использован в функциях, сравнивающих значения с плавающий точкой (см. листинг 2.1), для задания величины допустимого отклонения:

```
// ε = 0.001
#define M_EPSILON_E3   1e-3f
// ε = 0.0001
#define M_EPSILON_E4   1e-4f
// ε = 0.00001
#define M_EPSILON_E5   1e-5f
// ε = 0.000001
#define M_EPSILON_E6   1e-6f
```

Точки

Для работы с точками можно использовать класс `PointF` из библиотеки GDI+ (см. лабораторную работу №1 раздел «Расположение и размеры в GDI+»). Класс `PointF` позволяет работать не только с отдельными координатами точки, но и непосредственно с точкой. Однако метод `Equals` класса `PointF` при сравнении двух точек не учитывает потери точности при различных вычислениях.

В листинге 2.6 показан программный код функция `Equals`, которая выполняет сравнение соответствующих координат двух точек при заданном допустимом отклонении. В приведенной функции по умолчанию используется допустимое отклонение 10^{-6}, но в зависимости от требуемой точности, можно использовать любые значения допустимого отклонения.

Листинг 2.6. Функция сравнения точек

```
inline bool Equals(const Gdiplus::PointF &a,
const Gdiplus::PointF &b, float tolerance = 1e-6f)
{
    return (&a == &b) || (Equals(a.X, b.X, tolerance) &&
Equals(a.Y, b.Y, tolerance));
} // Equals
```

В следующем примере продемонстрировано использование функции `Equals` для сравнения двух точек с различными значениями допустимого отклонения:

```
PointF pt1(5.04f, 3.f); // точка (5.04, 3.00)
PointF pt2(5.f, 3.05f); // точка (5.00, 3.05)

// сравнение при допустимом отклонении по умолчанию
Equals(pt1, pt2); // результат pt1 ≠ pt2
// сравнение при допустимом отклонении 0.001
```

```
Equals(pt1, pt2, 1e-3f); // результат pt1 ≠ pt2
// сравнение при допустимом отклонении 0.1
Equals(pt1, pt2, 1e-1f); // результат pt1 ≈ pt2
```

В листинге 2.7 приведен программный код операторов сравнения «==» и «!=» применительно к классу `PointF`. Такие операторы позволят значительно повысить гибкость при работе с объектами класса `PointF`.

Листинг 2.7. Перегруженные операторы сравнения для класса `PointF`

```
inline bool operator==(const Gdiplus::PointF &a, const
Gdiplus::PointF &b)
{
    return Equals(a,b);
} // operator==

inline bool operator!=(const Gdiplus::PointF &a, const
Gdiplus::PointF &b)
{
    return !Equals(a,b);
} // operator!=
```

В следующем примере показан пример использования оператора «==» для сравнения двух объектов класса `PointF`:

```
PointF pt1(2.f, 1.5f); // точка (2.0, 1.5)
PointF pt2(2.f, 1.5f); // точка (2.0, 1.5)

if (pt1 == pt2) // сравнение двух точек
{
    /* точки pt1 и pt2 достаточно близко, то есть pt1 ≈ pt2 */
} // if
```

Векторы

Большинство алгоритмов компьютерной геометрии сводится к выполнению определенных операций с векторами. Для эффективной работы с векторами, как и в случае с точками, можно использовать классы, реализующие векторы. В листинге 2.8 приведено определение простого класса для представления двумерных векторов.

Листинг 2.8. Класс `Vector2d`

```
class Vector2d
{
public:

    float X, Y; // компоненты
```

```
6
7       // Конструкторы
8
9       Vector2d();
10      Vector2d(float x, float y);
11      Vector2d(const Vector2d &v);
12  #ifdef _GDIPLUSTYPES_H
13      Vector2d(const Gdiplus::PointF &pt);
14      Vector2d(const Gdiplus::PointF &a, const Gdiplus::PointF &b);
15  #endif
16
17      // Операторы
18
19      // сложение векторов
20      Vector2d operator+(const Vector2d &v) const;
21
22      // вычитание векторов
23      Vector2d operator-(const Vector2d &v) const;
24
25      // умножение на число
26      Vector2d operator*(float scalar) const;
27      friend Vector2d operator*(float scalar, const Vector2d &v);
28      Vector2d& operator*=(float scalar);
29
30      // деление на число
31      Vector2d operator/(float scalar) const;
32      Vector2d& operator/=(float scalar);
33
34      // противоположный вектор
35      Vector2d operator-() const;
36
37      // равенство с вектором
38      bool operator==(const Vector2d &v) const;
39      // не равенство с вектором
40      bool operator!=(const Vector2d &v) const;
41
42      // Методы
43
44      // возвращает true, если векторы равны, и false - в противном
    случае
45      bool Equals(const Vector2d &v, float tolerance = 1e-6f)
    const;
46
47      // возвращает норму (длину) вектора
48      float Norm() const;
49      // возвращает квадррат нормы (длины) вектора
```

```
50      float NormSquared() const;
51
52      // возвращает результат скалярного произведения с вектором v
53      float DotProduct(const Vector2d &v) const;
54
55      // возвращает угол с вектором v
56      float Angle(const Vector2d &v) const;
57
58      // возвращает нормированный вектор
59      Vector2d Normalize() const;
60  }; // class Vector2d
```

В приведенном классе `Vector2d` компоненты двумерного вектора хранятся соответственно в полях `X` и `Y`, которые инициализируются в конструкторах класса `Vector2d`, приведенных в листинге 2.9. Первый конструктор создает объект класса `Vector2d`, в котором значения полей `X` и `Y` равны нулю. Второй конструктор создает объект класса `Vector2d`, в котором значения полей `X` и `Y` задаются соответственно параметрами `x` и `y`. Третий конструктор создает копию объекта `v` класса `Vector2d`. Четвертый конструктор создает объект класса `Vector2d` на основе объекта `pt` класса `PointF`. Пятый конструктор создает объект класса `Vector2d` с помощью начальной и конечной точки, которые задаются объектами `a` и `b` класса `PointF`.

Листинг 2.9. Конструкторы класса `Vector2d`

```
1   inline Vector2d::Vector2d()
2       : X(0.f), Y(0.f)
3   {
4   }
5
6   inline Vector2d::Vector2d(float x, float y)
7       : X(x), Y(y)
8   {
9   }
10
11  inline Vector2d::Vector2d(const Vector2d &v)
12      : X(v.X), Y(v.Y)
13  {
14  }
15
16  #ifdef _GDIPLUSTYPES_H
17
18  inline Vector2d::Vector2d(const Gdiplus::PointF &pt)
19      : X(pt.X), Y(pt.Y)
20  {
```

```
}

inline Vector2d::Vector2d(const Gdiplus::PointF &a, const
Gdiplus::PointF &b)
    : X(b.X - a.X), Y(b.Y - a.Y)
{
}

#endif
```

В листинге 2.10 приводится код определения класса для представления трехмерных векторов. В классе `Vector3d` для хранения компонентов трехмерного вектора используются поля `X`, `Y` и `Z`.

Листинг 2.10. Класс `Vector3d`

```
class Vector3d
{
public:

    float X, Y, Z; // компоненты

    // Конструкторы

    Vector3d();
    Vector3d(float x, float y, float z);
    Vector3d(const Vector2d &v, float z = 0.f);
    Vector3d(const Vector3d &v);

    // Операторы

    // сложение векторов
    Vector3d operator+(const Vector3d &v) const;

    // вычитание векторов
    Vector3d operator-(const Vector3d &v) const;

    // умножение на число
    Vector3d operator*(float scalar) const;
    friend Vector3d operator*(float scalar, const Vector3d &v);
    Vector3d& operator*=(float scalar);

    // деление на число
    Vector3d operator/(float scalar) const;
    Vector3d& operator/=(float scalar);

    // противоположный вектор
```

```
32      Vector3d operator-() const;
33
34      // равенство с вектором
35      bool operator==(const Vector3d &v) const;
36      // не равенство с вектором
37      bool operator!=(const Vector3d &v) const;
38
39      // Методы
40
41      // возвращает true, если векторы равны, и false - в противном
    случае
42      bool Equals(const Vector3d &v, float tolerance = 1e-6f)
    const;
43
44      // возвращает норму (длину) вектора
45      float Norm() const;
46      // возвращает квадррат нормы (длины) вектора
47      float NormSquared() const;
48
49      // возвращает результат скалярного произведения с вектором v
50      float DotProduct(const Vector3d &v) const;
51
52      // возвращает угол с вектором v
53      float Angle(const Vector3d &v) const;
54
55      // возвращает результат векторного произведения с вектором v
56      Vector3d CrossProduct(const Vector3d &v) const;
57
58      // возвращает нормированный вектор
59      Vector3d Normalize() const;
60  }; // class Vector3d
```

Начальные значения полей `X`, `Y` и `Z` класса `Vector3d` задаются в его конструкторах, представленных в листинге 2.11. Первый конструктор создает объект класса `Vector3d`, в котором значения полей `X`, `Y` и `Z` равны нулю. Второй конструктор создает объект класса `Vector3d`, в котором значения полей `X`, `Y` и `Z` задаются соответственно параметрами `x`, `y` и `z`. Третий конструктор создает копию объекта `v` класса `Vector2d`, при этом значение поля `Z` задается параметром `z` (по умолчанию параметр `z` принимает значение равное нулю). Четвертый конструктор создает копию объекта `v` класса `Vector3d`.

Листинг 2.11. Конструкторы класса `Vector3d`

```
1   inline Vector3d::Vector3d()
2       : X(0.f), Y(0.f), Z(0.f)
```

```
3   {
4   }
5
6   inline Vector3d::Vector3d(float x, float y, float z)
7       : X(x), Y(y), Z(z)
8   {
9   }
10
11  inline Vector3d::Vector3d(const Vector2d &v, float z)
12      : X(v.X), Y(v.Y), Z(z)
13  {
14  }
15
16  inline Vector3d::Vector3d(const Vector3d &v)
17      : X(v.X), Y(v.Y), Z(v.Z)
18  {
19  }
```

Сложение и вычитание векторов

При сложении векторов $\vec{a}$ и $\vec{b}$ компоненты нового вектора получаются сложением соответствующих компонентов этих векторов:

$$\vec{a} + \vec{b} = \langle a_x + b_x,\ a_y + b_y \rangle, \tag{2.4}$$

$$\vec{a} + \vec{b} = \langle a_x + b_x,\ a_y + b_y,\ a_z + b_z \rangle. \tag{2.5}$$

Вычитание векторов $\vec{a}$ и $\vec{b}$ может быть представлено, как сложение вектора $\vec{a}$ с вектором, который коллинеарен вектору $\vec{b}$, равен ему по длине, но является с ним противоположно направленным, то есть:

$$\vec{a} - \vec{b} = \vec{a} + (-\vec{b}). \tag{2.6}$$

В листинге 2.12 приводится программный код операторов «+» и «–» классов `Vector2d` и `Vector3d` для сложения и вычитания векторов.

Листинг 2.12. Операторы сложения и вычитания векторов

```
1   inline Vector2d Vector2d::operator+(const Vector2d &v) const
2   {
3       return Vector2d(X + v.X, Y + v.Y);
4   } // operator+
5
6   inline Vector2d Vector2d::operator-(const Vector2d &v) const
7   {
8       return Vector2d(X - v.X, Y - v.Y);
```

```
} // operator-

inline Vector3d Vector3d::operator+(const Vector3d &v) const
{
    return Vector3d(X + v.X, Y + v.Y, Z + v.Z);
} // operator+

inline Vector3d Vector3d::operator-(const Vector3d &v) const
{
    return Vector3d(X - v.X, Y - v.Y, Z - v.Z);
} // operator-
```

Умножение и деление на число

Для умножения вектора $\vec{v}$ на число λ следует умножить каждый компонент вектора на это число. Кроме того, вектор можно разделить на число λ, что соответствует умножению вектора на величину $^1/_\lambda$.

$$\lambda\vec{v} = \langle \lambda v_x,\ \lambda v_y \rangle, \tag{2.7}$$

$$\lambda\vec{v} = \langle \lambda v_x,\ \lambda v_y,\ \lambda v_z \rangle. \tag{2.8}$$

В листингах 2.13,2.14 показан программный код операторов классов `Vector2d` и `Vector3d` для умножения и деления вектора на число.

Листинг 2.13. Операторы умножение вектора на число

```
inline Vector2d Vector2d::operator*(float scalar) const
{
    return Vector2d(X*scalar, Y*scalar);
} // operator*

inline Vector2d operator*(float scalar, const Vector2d &v)
{
    return Vector2d(v.X*scalar, v.Y*scalar);
} // operator*

inline Vector2d& Vector2d::operator*=(float scalar)
{
    X *= scalar;
    Y *= scalar;
    return (*this);
} // operator*=

inline Vector3d Vector3d::operator*(float scalar) const
{
```

```
20      return Vector3d(X*scalar, Y*scalar, Z*scalar);
21  } // operator*
22
23  inline Vector3d operator*(float scalar, const Vector3d &v)
24  {
25      return Vector3d(v.X*scalar, v.Y*scalar, v.Z*scalar);
26  } // operator*
27
28  inline Vector3d& Vector3d::operator*=(float scalar)
29  {
30      X *= scalar;
31      Y *= scalar;
32      Z *= scalar;
33      return (*this);
34  } // operator*=
```

Листинг 2.14. Операторы деления вектора на число

```
1   inline Vector2d Vector2d::operator/(float scalar) const
2   {
3       return Vector2d(X/scalar, Y/scalar);
4   } // operator/
5
6   inline Vector2d& Vector2d::operator/=(float scalar)
7   {
8       X /= scalar;
9       Y /= scalar;
10      return (*this);
11  } // operator/=
12
13  inline Vector3d Vector3d::operator/(float scalar) const
14  {
15      return Vector3d(X/scalar, Y/scalar, Z/scalar);
16  } // operator/
17
18  inline Vector3d& Vector3d::operator/=(float scalar)
19  {
20      X /= scalar;
21      Y /= scalar;
22      Z /= scalar;
23      return (*this);
24  } // operator/=
```

Если умножить вектор на отрицательное число (например, на -1), можно изменить его направление на противоположное. В листинге 2.15 представлен программный код оператора классов `Vector2d` и

Vector3d для инвертирования вектора, который повышает гибкость при работе с векторами.

Листинг 2.15. Операторы инвертирования вектора

```
inline Vector2d Vector2d::operator-() const
{
    return Vector2d(-X, -Y);
} // operator-

inline Vector3d Vector3d::operator-() const
{
    return Vector3d(-X, -Y, -Z);
} // operator-
```

В следующем примере показы различные варианты изменения направления вектора на противоположное:

```
Vector2d v1(2.f, 3.f); // вектор <2.0, 3.0>

// умножение вектора на -1
Vector2d v2 = -1.f * v1; // результат вектор <-2.0,-3.0>
// инвертирование вектора
Vector2d v3 = -v1; // результат вектор <-2.0,-3.0>
```

Сравнение векторов

В общем случае сравнение двух векторов выполняется путем сравнения соответствующих компонентов этих векторов. В листингах 2.16,2.17 показан программный код метода Equals и операторов «==» и «!=» классов Vector2d и Vector3d, которые выполняют сравнение векторов при заданном допустимом отклонении (по умолчанию 10^{-6}).

Листинг 2.16. Методы сравнения векторов

```
inline bool Vector2d::Equals(const Vector2d &v, float tolerance)
const
{
    return (this == &v) || (::Equals(X, v.X, tolerance) &&
::Equals(Y, v.Y, tolerance));
} // Equals

inline bool Vector3d::Equals(const Vector3d &v, float tolerance)
const
{
    return (this == &v) || (::Equals(X, v.X, tolerance) &&
::Equals(Y, v.Y, tolerance) && ::Equals(Z, v.Z, tolerance));
} // Equals
```

Листинг 2.17. Операторы сравнения векторов

```
inline bool Vector2d::operator==(const Vector2d &v) const
{
    return Equals(v);
} // operator==

inline bool Vector2d::operator!=(const Vector2d &v) const
{
    return !Equals(v);
} // operator!=

inline bool Vector3d::operator==(const Vector3d &v) const
{
    return Equals(v);
} // operator==

inline bool Vector3d::operator!=(const Vector3d &v) const
{
    return !Equals(v);
} // operator!=
```

Длина вектора

Длиной (*модулем, нормой*) *вектора* называется скалярная величина, равная расстоянию между его началом и концом. Длина двумерного и трехмерного вектора вычисляется через его компоненты по следующим формулам:

$$|\vec{a}| = \sqrt{a_x^2 + a_y^2}, \tag{2.9}$$

$$|\vec{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}. \tag{2.10}$$

Длина нулевого вектора равна нулю. Вектор, длина которого равна единице, называется *единичным вектором* (*unit vector*). Изменить длину вектора можно умножением/делением этого вектора на число.

Важно отметить, что если для решения задачи можно обойтись только *квадратом* длины вектора, не следует извлекать квадратный корень. Например, при проверке является ли вектор единичным или нулевым, достаточно вычислить только квадрат длины этого вектора.

В листингах 2.18,2.19 приводится программный код двух методов (`Norm` и `NormSquared`) классов `Vector2d` и `Vector3d`, которые вычисляют длину вектора и квадрат его длины.

Листинг 2.18. Методы вычисления длины вектора

```
1   inline float Vector2d::Norm() const
2   {
3       return SquareRoot(X*X + Y*Y);
4   } // Norm
5
6   inline float Vector3d::Norm() const
7   {
8       return SquareRoot(X*X + Y*Y + Z*Z);
9   } // Norm
10
```

Листинг 2.19. Методы вычисления квадрата длины вектора

```
1   inline float Vector2d::NormSquared() const
2   {
3       return (X*X + Y*Y);
4   } // NormSquared
5
6   inline float Vector3d::NormSquared() const
7   {
8       return (X*X + Y*Y + Z*Z);
9   } // NormSquared
```

Часто некоторый вектор $\vec{v}$ требуется преобразовать в единичный вектор $\hat{v}$, который будет сонаправлен с вектором $\vec{v}$. Операция приведения вектора $\vec{v}$ к единичному вектору $\hat{v}$ называется *нормализацией* (*normalize*) вектора и выполняется по следующей формуле:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}. \tag{2.11}$$

В классах `Vector2d` и `Vector3d` имеется метод `Normalize`, который возвращает нормализованный вектор. В листинге 2.20 приведен программный код метода `Normalize` класса `Vector2d`. Аналогичный метод класса `Vector3d` имеет схожий программный код, поэтому нет нужды его отдельно демонстрировать.

Листинг 2.20. Метод нормализации вектора

```
1   inline Vector2d Vector2d::Normalize() const
2   {
3       float norm = NormSquared();
4       if ( ::Equals(norm, 1.f) ) return (*this); // norm ≈ 1
5
6       return (*this) / SquareRoot(norm);
7   } // Normalize
```

Скалярное произведение векторов

Скалярное произведение (*dot product*) – операция перемножения двух векторов, в результате которого получается скалярная величина. Скалярное произведение $\vec{a} \cdot \vec{b}$ векторов $\vec{a}$ и $\vec{b}$, заданных своими компонентами, вычисляется как сумма попарных произведений компонентов векторов:

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y, \tag{2.12}$$

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z. \tag{2.13}$$

В листинге 2.21 приведен программный код метода `DotProduct` классов `Vector2d` и `Vector3d`, который вычисляет скалярное произведение двух векторов.

Листинг 2.21. Методы вычисления скалярного произведения векторов

```
1  inline float Vector2d::DotProduct(const Vector2d &v) const
2  {
3      return (X*v.X + Y*v.Y);
4  } // DotProduct
5
6  inline float Vector3d::DotProduct(const Vector3d &v) const
7  {
8      return (X*v.X + Y*v.Y + Z*v.Z);
9  } // DotProduct
```

Скалярное произведение $\vec{a} \cdot \vec{b}$ векторов $\vec{a}$ и $\vec{b}$ можно также вычислить через произведение длин этих векторов на косинус угла α между ними:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos \alpha. \tag{2.14}$$

Можно отметить, что если угол между ненулевыми векторами $\vec{a}$ и $\vec{b}$ острый, их скалярное произведение больше нуля ($\vec{a} \cdot \vec{b} > 0$), а если угол между ними тупой, скалярное произведение меньше нуля ($\vec{a} \cdot \vec{b} < 0$). С помощью формулы (2.14) можно найти угол α между векторами $\vec{a}$ и $\vec{b}$ через их скалярное произведение:

$$\alpha = \arccos \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}. \tag{2.15}$$

Важно отметить, что формула (2.15) не подходит для нахождения угла, если хотя бы один из векторов является нулевым вектором. По

определению угол между любым вектором и нулевым вектором считаем равным нулю. Окончательная формула нахождения угла α между векторами $\vec{a}$ и $\vec{b}$ имеет следующий вид:

$$\alpha = \begin{cases} 0, & |\vec{a}||\vec{b}| = 0 \\ \arccos\dfrac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|}, & |\vec{a}||\vec{b}| \neq 0 \end{cases} \quad (2.16)$$

Угол между сонаправленными векторами равен нулю, а между противоположно направленными векторами равен 180° (в радианах π). Если векторы ортогональны, угол равен 90° (в радианах $\pi/2$).

В классах `Vector2d` и `Vector3d` имеется метод `Angle` для нахождения угла (в радианах) между двумя векторами. В листинге 2.22 приведен программный код метода `Angle` класса `Vector2d`. Одноименный метод класса `Vector3d` имеет аналогичный программный код.

Листинг 2.22. Метод вычисления угла между векторами

```
1   inline float Vector2d::Angle(const Vector2d &v) const
2   {
3       float norm = NormSquared() * v.NormSquared();
4       if (IsZero(norm)) return 0.f; // угол = 0°, если хотя бы один
    из векторов нулевой
5
6       float dot = DotProduct(v); // находим скалярное произведение
7       if (IsZero(dot)) return (float)M_PI_2; // угол = 90°, если
    векторы ортогональны
8
9       float cos = dot / SquareRoot(norm); // находим косинус
10
11      if (::Equals(cos, 1.f)) return 0.f; // угол = 0°
12      if (::Equals(cos,-1.f)) return (float)M_PI; // угол = 180°
13
14      return ArcCos(cos); // находим угол (в радианах)
15  } // Angle
```

Векторное произведение

Векторное произведение (*cross product*) – операция перемножения двух векторов, в результате которого получается вектор перпендикулярный к плоскости, в которой лежат перемножаемые векторы. Векторное произведение существует только для трехмерных векторов. Для вычисления векторного произведения $\vec{a} \times \vec{b}$ векторов $\vec{a}$ и $\vec{b}$ используется следующая формула.

$$\vec{a} \times \vec{b} = \langle a_y b_z - a_z b_y,\ a_z b_x - a_x b_z,\ a_x b_y - a_y b_x \rangle. \quad (2.17)$$

В листинге 2.23 представлен программный код метода `CrossProduct` класса `Vector3d`, который вычисляет векторное произведение.

Листинг 2.23. Метод вычисления векторного произведения

```
1   inline Vector3d Vector3d::CrossProduct(const Vector3d &v) const
2   {
3       return Vector3d((Y*v.Z - Z*v.Y), (Z*v.X - X*v.Z), (X*v.Y -
    Y*v.X));
4   } // CrossProduct
```

Мировые окна и порты просмотра

При работе с мировыми окнами и портами просмотра удобно использовать определенные структуры данных (или классы), которые буду хранить границы мирового окна или порта просмотра.

В листинге 2.24 приводится код определения простого класса для представления мировых окон.

Листинг 2.24. Класс `WorldWindow`

```
1   class WorldWindow
2   {
3   public:
4
5       float Left; // левая граница
6       float Top; // верхняя граница
7       float Right; // правая граница
8       float Bottom; // нижняя граница
9
10      inline WorldWindow(float left, float top, float right, float
    bottom)
11          : Left(left), Top(top), Right(right), Bottom(bottom)
12      {
13      }
14
15      // возвращает ширину мирового окна
16      inline float Width() const
17      {
18          return (Right - Left);
19      } // Width
20
21      // возвращает высоту мирового окна
22      inline float Height() const
23      {
24          return (Top - Bottom);
```

```
25      } // Height
26  }; // class WorldWindow
```

Для представления порта просмотра можно использовать класс `Rect` из библиотеки GDI+ (см. лабораторную работу №1 раздел «Расположение и размеры в GDI+») или структуру `RECT` из Win32 API, которая имеет следующее описание:

```
typedef struct _RECT {
    LONG left; // левая граница
    LONG top; // верхняя граница
    LONG right; // правая граница
    LONG bottom; // нижняя граница
} RECT, *PRECT;
```

В листинге 2.25 приведено определение класса для представления порта просмотра. Этот класс сочетает в себе достоинства класса `Rect` и поддерживает структуру `RECT`.

Листинг 2.25. Класс `Viewport`

```
1   class Viewport : public Gdiplus::Rect
2   {
3   public:
4
5       inline Viewport(int left, int top, int right, int bottom)
6           : Gdiplus::Rect(left, top, right-left, bottom-top)
7       {
8       }
9
10      inline Viewport(const RECT &rect)
11          : Gdiplus::Rect(rect.left, rect.top, rect.right-
    rect.left, rect.bottom-rect.top)
12      {
13      }
14
15      inline operator RECT() const
16      {
17          RECT rect = {X, Y, X+Width, Y+Height};
18          return rect;
19      }
20  }; // class Viewport
```

Преобразование из мирового окна в порт просмотра

Имея описание мирового окна и порта просмотра, можно производить *преобразование мировое окно – порт просмотра* (*window-to-*

viewport mapping). Это преобразование основывается на формуле, которая вычисляет точку (sx, sy) в координатах поверхности отображения для любой заданной точки (x, y) в мировых координатах.

$$\begin{aligned} sx &= A \cdot x + C, \\ sy &= B \cdot y + D, \end{aligned} \tag{2.18}$$

где A, B, C и D константы, которые вычисляются следующим образом:

$$\begin{aligned} A &= \frac{V_{right} - V_{left}}{W_{right} - W_{left}}, C = V_{left} - A \cdot W_{left}, \\ B &= \frac{V_{bottom} - V_{top}}{W_{bottom} - W_{top}}, D = V_{top} - B \cdot W_{top}. \end{aligned} \tag{2.19}$$

Константы A и B изменяют масштаб координат x и y, а C и D переносят их.

Обратное преобразование *порт просмотра – мировое окно* выполняется по следующей формуле:

$$\begin{aligned} x &= A \cdot sx + C, \\ y &= B \cdot sy + D, \end{aligned} \tag{2.20}$$

где константы A, B, C и D вычисляются следующим образом:

$$\begin{aligned} A &= \frac{W_{right} - W_{left}}{V_{right} - V_{left}}, C = W_{left} - A \cdot V_{left}, \\ B &= \frac{W_{bottom} - W_{top}}{V_{bottom} - V_{top}}, D = W_{top} - B \cdot V_{top}. \end{aligned} \tag{2.21}$$

В листинге 2.26 представлен программный код функций, где выполняется преобразование между мировым окном и портом просмотра.

Листинг 2.26. Функции преобразования для мирового окна и порта просмотра

```
1   // выполняет преобразование из мирового окна в порт просмотра
2   inline void WorldToViewport(const WorldWindow &w, const Viewport
    &vp, Gdiplus::PointF *points, unsigned int count)
3   {
4       float A = vp.Width / (w.Right - w.Left);
5       float B = vp.Height / (w.Bottom - w.Top);
6       float C = vp.X - A*w.Left;
7       float D = vp.Y - B*w.Top;
8
9       for (unsigned int i = 0; i < count; ++i)
10      {
```

```
11          points[i].X = A*points[i].X + C;
12          points[i].Y = B*points[i].Y + D;
13      } // for
14  } // WorldToViewport
15
16  // выполняет преобразование из порта просмотра в мировое окно
17  inline void ViewportToWorld(const WorldWindow &w, const Viewport
    &vp, Gdiplus::PointF *points, unsigned int count)
18  {
19      float A = (w.Right - w.Left) / (float)vp.Width;
20      float B = (w.Bottom - w.Top) / (float)vp.Height;
21      float C = w.Left - A*vp.X;
22      float D = w.Top - B*vp.Y;
23
24      for (unsigned int i = 0; i < count; ++i)
25      {
26          points[i].X = A*points[i].X + C;
27          points[i].Y = B*points[i].Y + D;
28      } // for
29  } // ViewportToWorld
```

Функция отсечение отрезков

Как правило, отсечение производят в мировых координатах относительно границ мирового окна. Поэтому перед отсечением нет необходимости выполнять преобразование из мирового окна в порт просмотра. Функция отсечения может иметь следующую сигнатуру:

```
bool LineClip(const WorldWindow &w, PointF &pt1, PointF &pt2);
```

Первый параметр, *w*, – объект класса `WorldWindow`, который задает мировое окно. Второй и третий параметры (*pt1* и *pt2*) указывают на объекты класса `PointF`, которые представляют соответственно начальную и конечную точку отрезка. Кроме того через эти объекты функция `LineClip` возвращает «новые» вершины отсекаемого отрезка.

Функция `LineClip` возвращает `true`, если отрезок «выжил» в процессе отсечения, и `false` – в противном случае.

Создание кривых с помощью ломаных линий

Простой способ создания кривой – аппроксимировать ее с помощью ломаной линии. Нужно всего лишь определить набор точек, расположенных на этой кривой, а затем соединить эти точки отрезками.

Важно отметить, если вершины ломаной линии будут располагаться достаточно близко друг от друга, человеческий глаз естественным образом соединит нарисованные ребра, и получится гладкая кри-

вая (рис. 2.1, *а*). Если же вершины расположены не очень близко друг к другу, рисуемая кривая гладкой может и не получиться (рис. 2.1, *б*).

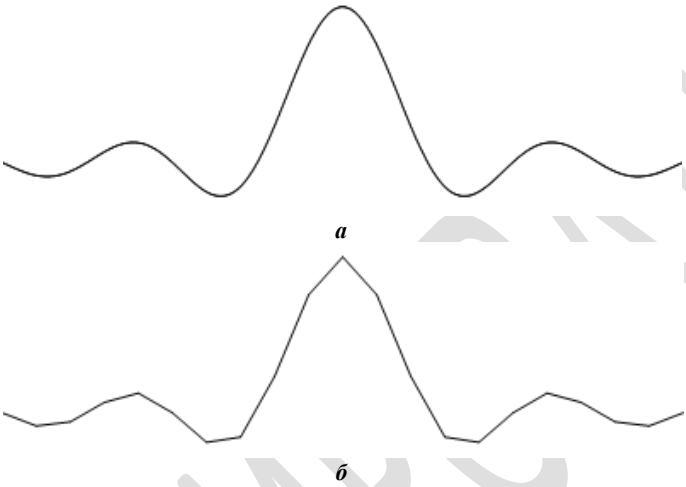


Рис. 2.1. Варианты аппроксимации кривой с помощью ломаной линии при маленьком (*а*) и большом (*б*) шаге аппроксимации

Создание ломаной линии

Так как ломаная линия представляет собой последовательность соединенных между собой отрезков, то для задания ломаных линий, как нельзя лучше подойдут массивы. Вот как в C++ выглядит объявление массива точек (вершин) состоящего из десяти элементов:

```
Gdiplus::PointF points[10];
```

Этот способ определения массива точек наиболее эффективен. Однако он связан с серьезными ограничениями, поскольку размер массива в ходе программы изменяться не может. Конечно, можно определить массив достаточно большого размера, но это может привести к перерасходу памяти. В языке C++ предусмотрены два оператора динамического распределения памяти: `new` и `delete`. Эти операторы выделяют и освобождают память в ходе выполнение программы. С помощью оператора `new` можно выделить память для массива точек:

```
Gdiplus::PointF *points = new Gdiplus::PointF[10];
```

Оператор `new` выделяет память и создает массив. Однако такой массив сам по себе из памяти не удалится; поэтому, когда необходимо

его удалить, следует явно вызвать оператор `delete` в следующей форме:

```
delete [] points;
```

Программисты часто создают вспомогательные классы, которые являются контейнерами для динамических массивов. Одним из таких контейнеров является класс `vector` из библиотеки STL (Standard Template Library – стандартная библиотека шаблонов). Для использования класса `vector` необходимо включить соответствующий заголовочный файл:

```
#include <vector>
```

Объявление массива точек с помощью класса `vector` выполняется следующим образом:

```
1  // создается пустой массив точек
2  std::vector<Gdiplus::PointF> points1;
3
4  // создается массив из 100 точек
5  std::vector<Gdiplus::PointF> points2(100);
```

Предположим, что необходимо создать ломаную линию, которая представляет собой график некоторой математической функции $y = f(x)$ при изменении аргумента x в интервале от x_{min} до x_{max}. Для этого необходимо найти значения данной функции при равноудаленных значениях аргумента x и получить последовательность точек с координатами $\left(x_i, f(x_i)\right)$. Значение x_i можно определить как

$$x_i = x_{min} + i \cdot \Delta x, \text{ при } 0 \le i < n, \tag{2.22}$$

где Δx – шаг изменения аргумента x; n – количество точек на графике (чем выше значение n, тем более точным получится график).

Шаг изменения аргумента x определяется следующим образом:

$$\Delta x = \frac{x_{max} - x_{min}}{n - 1}. \tag{2.23}$$

Рис. 2.2 представляет собой пример построения графика функции $y = \sin x$ при изменении аргумента x от 0 до 2π.

В листинге 2.27 показан программный код функции `Plot`, которая создает ломаную линию, состоящую из `n` точек и представляющую собой график функции `f(x)` при изменении аргумента от `xMin` до `xMax`. Отметим, что функция `Plot` определена в пространстве имен `Shapes2d`.

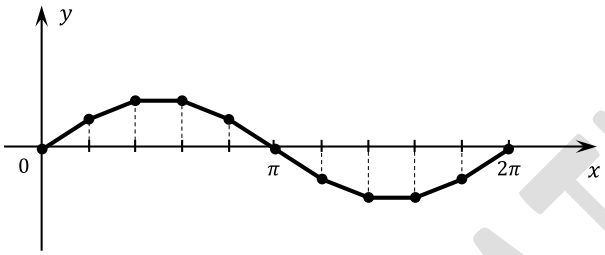


Рис. 2.2. *Построение графика функции* $y = \sin x$

Листинг 2.27. *Функция* `Plot` *для создания ломаной линии*

```
1   template <class Func>
2   inline std::vector<Gdiplus::PointF> Plot(float xMin, float xMax,
    size_t n, Func f)
3   {
4       std::vector<Gdiplus::PointF> result;
5
6       if (n > 1)
7       {
8           result.reserve(n); // зарезервируем место для n точек
9
10          // определяем шаг изменения x
11          float dx = (xMax - xMin) / (float)(n - 1);
12
13          for (size_t i = 0; i < n; ++i)
14          {
15              float x = xMin + i*dx;
16
17              // добавим точку в массив
18              result.push_back( Gdiplus::PointF(x, f(x)) );
19          } // for
20      } // if
21
22      return result;
23  } // Plot
```

В следующем примере показано, как использовать функцию `Plot` для создания ломаной линии, представляющей собой график функции $y = \sin x$ при изменении аргумента x от 0 до 2π (см. рис. 2.2).

```
1   std::vector<Gdiplus::PointF> sinx = Plot(0.f, (float)M_2PI, 10,
    Sin);
2
3   if (!sinx.empty())
4   {
5       /* ломаная линия успешно создана */
6   } // if
```

Как видно из этого примера, для создания ломаной линии достаточно задать количество точек и интервал изменения аргумента, а так же указать функцию, которая имеет следующую сигнатуру:

```
float f(float x);
```

Также для создания ломаной линии с помощью функции `Plot` можно использовать лямбда-выражение C++. В следующем примере показано, как использовать функцию `Plot` и лямбда-выражение для создания ломаной линии, представляющей собой график функции $y = 1.9\,x^2 - 2.2\,x - 3.1$ при изменении аргумента x от -1.5 до 2.5.

```
1   std::vector<Gdiplus::PointF> parabola = Plot(-1.5f, 2.5f, 100,
    [](float x) -> Point2d { return 1.9*x*x - 2.2f*x - 3.1f; });
2
3   if (!parabola.empty())
4   {
5       /* ломаная линия успешно создана */
6   } // if
```

При отыскании точек кривой не всегда бывает удобно или возможно использовать уравнение, связывающее координаты x и y. В таком случае бывает полезно использовать параметрическое представление кривой. В параметрическом виде каждая координата точки кривой представлена как функция параметра t.

В листинге 2.28 показан программный код функции `Curve`, которая создает ломаную линию, состоящую из `n` точек и представляющую собой кривую, заданную параметрическим уравнением при изменении параметра от `tMin` до `tMax`.

Листинг 2.28. *Функция `Curve` для создания ломаной линии*

```
1   template <class Func>
2   inline std::vector<Gdiplus::PointF> Curve(float tMin, float tMax,
    size_t n, Func f)
3   {
4       std::vector<Gdiplus::PointF> result;
5
6       if (n > 1)
7       {
8           result.reserve(n); // зарезервируем место для n точек
9
10          // определяем шаг изменения t
11          float dt = (tMax - tMin) / (float)(n - 1);
12
13          for (size_t i = 0; i < n; ++i)
14          {
15              // добавим точку в массив
```

```
            result.push_back( f(tMin + i*dt) );
        } // for
    } // if

    return result;
} // Curve
```

Для создания ломаной линии с помощью функции `Curve` достаточно задать количество точек и интервал изменения параметра t, а так же указать функцию (или лямбда-выражение), которая имеет следующую сигнатуру:

```
Gdiplus::PointF f(float t);
```

После чего полученные ребра ломаной линии подвергаем отсечению. Далее координаты тех ребер, которые останутся (в какой-либо форме) внутри мирового окна, будут преобразованы в координаты порта просмотра. Наконец полученные координаты будут использованы для рисования «выжившего» ребра ломаной линии.

Рисование ломаной линии

В листинге 2.29 показан программный код функции, которая рисует ломаная линия. При рисовании ломаной линии каждое ее ребро подвергается отсечению. Далее координаты тех ребер, которые останутся (в каком-либо виде) внутри мирового окна, нужно преобразовать в координаты порта просмотра. Наконец полученные координаты должны быть использованы для рисования «выживших» ребер ломаной линии.

Листинг 2.29. Функция рисования ломаной линии

```
1   void DrawPolyline(Gdiplus::Graphics &g, const Gdiplus::Pen *pen,
    const std::vector<Gdiplus::PointF> &polyline,
    const WorldWindow &w, const Viewport &vp)
2   {
3       for (size_t i = 1; i < polyline.size(); ++i)
4       {
5           PointF points[2] = { polyline[i-1], polyline[i] };
6
7           // производим отсечение
8           bool fIsDraw = LineClip(w, points[0], points[1]);
9
10          if (fIsDraw) // если ребро «выжило»
11          {
12              // выполняем преобразование координат точек
13              WorldToViewport(w, vp, points, 2);
14              // рисуем ребро
```

```
15              g.DrawLines(pen, points, 2);
16          } // if
17      } // for
18  } // DrawPolyline
```

Регионы GDI+

Для проверки принадлежности точки полигону или выполнения операции отсечения в библиотеке GDI+ используется *регион* (*region*). Регионы так же, как кисти, перья, шрифты и изображения, являются графическими объектами GDI+.

Работа с регионами в GDI+ осуществляется с помощью класса `Region`, который определен в заголовочном файле gdiplusheaders.h.

Класс `Region` имеет следующие конструкторы:

```
Region();
Region(const Rect &rect);
Region(const RectF &rect);
Region(const GraphicsPath *path);
Region(const BYTE *regionData, INT size);
Region(HRGN hRgn);
```

Первый конструктор создает объект класса `Region`, представляющий бесконечный регион. Второй и третий конструкторы создают объекты класса `Region` (представляющий прямоугольный регион) на основе объекта, соответственно, класса `Rect` и класса `RectF`. Четвертый конструктор создает объект класса `Region` на основе объекта класса `GraphicsPath`. Пятый конструктор создает объект класса `Region` на основе массива данных, описывающего регион (см. в документации Platform SDK). Шестой конструктор создает объект класса `Region` на основе дескриптора региона, который используются в библиотеке GDI (более подробную информацию см. в документации Platform SDK).

В следующем примере проиллюстрировано создание прямоугольного и эллиптического регионов.

```
1   // создаем прямоугольный регион
2
3   Rect rect(50, 50, 200, 100); // прямоугольник 200x100
4   Region rgn1(rect);
5
6   // создаем элиптический регион
7
8   // сперва создаем контур
9   GraphicsPath path;
```

```
10  path.AddEllipse(Rect(150, 60, 200, 100));
11
12  // а затем создаем регион
13  Region rgn2(&path);
```

Прежде чем перейти к изучению основных методов класса Region, следует отметить, что в этом классе, как и в других классах GDI+, есть методы GetLastStatus и Clone, который имеет следующий прототип:

```
Region* Clone() const;
```

В случае успеха метод Clone возвращает указатель на объект класса Region, который должен быть удален вызовом оператора delete после того как станет не нужен. В случае ошибки – NULL.

Изменение регионов

В классе Region есть ряд методов, которые позволяют изменять регион уже после его создания. Прототипы и краткое описание этих методов приведены в табл. 2.2.

Таблица 2.2. Методы класса Region для изменения регионов

Метод	Описание
`Status Complement(` `  const GraphicsPath *path);`	Добавляет в регион, представленный текущим объектом класса Region, контур, представленный указанным объектом класса GraphicsPath

`Status Complement(`
`  const Rect &rect);`
`Status Complement(`
`  const RectF &rect);`
	Добавляет в регион, представленный текущим объектом класса Region, прямоугольник, представленный указанным объектом класса Rect (или RectF)

`Status Complement(` `  const Region *region);`	Добавляет в регион, представленный текущим объектом класса Region, другой регион, представленный указанным объектом класса Region
`Status Exclude(` `  const GraphicsPath *path);`	Исключает из региона, представленный текущим объектом класса Region, контур, представленный указанным объектом класса GraphicsPath

Продолжение табл. 2.2

Метод	Описание

`Status Exclude(`
`  const Rect &rect);`
`Status Exclude(`
`  const RectF &rect);`
	Исключает из региона, представленный текущим объектом класса `Region`, прямоугольник, представленный указанным объектом класса `Rect` (или `RectF`)

`Status Exclude(` `  const Region *region);`	Исключает из региона, представленный текущим объектом класса `Region`, другой регион, представленный указанным объектом класса `Region`
`Status Intersect(` `  const GraphicsPath *path);`	Заменяет регион, представленный текущим объектом класса `Region`, на его пересечение с контуром, представленным указанным объектом класса `GraphicsPath`

`Status Intersect(`
`  const Rect &rect);`
`Status Intersect(`
`  const RectF &rect);`
	Заменяет регион, представленный текущим объектом класса `Region`, на его пересечение с прямоугольником, представленным указанным объектом класса `Rect` (или `RectF`)

`Status Intersect(` `  const Region *region);`	Заменяет регион, представленный текущим объектом класса `Region`, на его пересечение с другим регионом, представленным указанным объектом класса `Region`
`Status MakeEmpty();`	Заменяет регион, представленный текущим объектом класса `Region`, на пустой регион
`Status MakeInfinite();`	Заменяет регион, представленный текущим объектом класса `Region`, на бесконечный регион
`Status Union(` `  const GraphicsPath *path);`	Заменяет регион, представленный текущим объектом класса `Region`, на его объединение с контуром, представленным указанным объектом класса `GraphicsPath`

`Status Union(`
`  const Rect &rect);`
`Status Union(`
`  const RectF &rect);`
	Заменяет регион, представленный текущим объектом класса `Region`, на его объединение с прямоугольником, представленным указанным объектом класса `Rect` (или `RectF`)

Окончание табл. 2.2

Метод	Описание
`Status Union(` `  const Region *region);`	Заменяет регион, представленный текущим объектом класса Region, на его объединение с другим регионом, представленным указанным объектом класса Region
`Status Xor(` `  const GraphicsPath *path);`	Заменяет регион, представленный текущим объектом класса Region, разностью его объединения и пересечения с контуром, представленным указанным объектом класса GraphicsPath
`Status Xor(` `  const Region *region);`	Заменяет регион, представленный текущим объектом класса Region, разностью его объединения и пересечения с другим регионом, представленным указанным объектом класса Region
`Status Xor(const Rect &rect);` `Status Xor(const RectF &rect);`	Заменяет регион, представленный текущим объектом класса Region, разностью его объединения и пересечения с прямоугольником, представленным указанным объектом класса Rect (или RectF)

В классе Region есть также два метода, которые могут определить является ли регион пустым или бесконечным – IsEmpty и IsInfinite:

```
BOOL IsEmpty(const Graphics *g) const;

BOOL IsInfinite(const Graphics *g) const;
```

Метод IsEmpty возвращает TRUE если регион пустой, и FALSE – в противном случае. Метод IsInfinite возвращает TRUE если регион бесконечный, и FALSE – в противном случае.

Определение принадлежности региону

Для определения принадлежности точки или прямоугольника региону используется метод IsVisible класса Region. Этот метод имеет следующие прототипы:

```
BOOL IsVisible(INT x, INT y,
    const Graphics *g = NULL) const;

BOOL IsVisible(REAL x, REAL y,
    const Graphics *g = NULL) const;
```

```
BOOL IsVisible(const Point &point,
    const Graphics *g = NULL) const;
BOOL IsVisible(const PointF &point,
    const Graphics *g = NULL) const;
BOOL IsVisible(INT x, INT y, INT width, INT height,
    const Graphics *g) const;
BOOL IsVisible(REAL x, REAL y, REAL width, REAL height,
    const Graphics *g = NULL) const;
BOOL IsVisible(const Rect &rect,
    const Graphics *g = NULL) const;
BOOL IsVisible(const RectF &rect,
    const Graphics *g = NULL) const;
```

Если указанная точка или прямоугольник лежат внутри региона метод `IsVisible` возвращает `TRUE`, и `FALSE` – в противном случае.

Отсечение

Отсечение в GDI+ заключается в запрете рисования за пределами определенного региона. Для задания региона отсечения в классе `Graphics` есть метод `SetClip`, который имеет следующие прототипы:

```
Status SetClip(const Graphics *g,
    CombineMode combineMode = CombineModeReplace);
Status SetClip(const Rect &rect,
    CombineMode combineMode = CombineModeReplace);
Status SetClip(const RectF &rect,
    CombineMode combineMode = CombineModeReplace);
Status SetClip(const GraphicsPath *path,
    CombineMode combineMode = CombineModeReplace);
Status SetClip(const Region *region,
    CombineMode combineMode = CombineModeReplace);
Status SetClip(HRGN hRgn,
    CombineMode combineMode = CombineModeReplace);
```

Первый прототип позволяет задать регион отсечения на основе региона отсечения, заданного для объекта класса `Graphics`. Второй и третий прототипы позволяют задать регион отсечения на основе прямоугольника, представленный соответственно объектами класса `Rect` и класса `RectF`. Четвертый прототип позволяет задать регион отсечения на основе контура, представленного объектом класса `GraphicsPath`. Пятый прототип позволяет задать регион отсечения на основе региона,

представленного объектом класса `Region`. Шестой прототип позволяет задать регион отсечения на основе дескриптора региона (см. в документации Platform SDK).

Параметр *`combineMode`* определяет, как задаваемый регион сочетается с существующим регионом отсечения. Этот параметр должен принимать одно из значений перечисляемого типа `CombineMode`:

- `CombineModeReplace` – существующий регион отсечения заменяется задаваемым регионом;
- `CombineModeIntersect` – существующий регион отсечения заменяется его пересечением с задаваемым регионом;
- `CombineModeUnion` – существующий регион отсечения заменяется его объединением с задаваемым регионом;
- `CombineModeXor` – существующий регион отсечения заменяется разностью его объединения и пересечения с задаваемым регионом;
- `CombineModeExclude` – задаваемый регион исключается из существующего региона отсечения;
- `CombineModeComplement` – существующий регион отсечения заменяется частью задаваемого региона, которая находится вне существующего региона отсечения.

По умолчанию в качестве региона отсечения используется бесконечный регион. Получить существующий регион отсечения можно с помощью метода `GetClip` класса `Graphics`:

```
Status GetClip(Region *region) const;
```

Параметр указывает на объект класса `Region`, в который будет сохранен существующий регион отсечения.

В следующем примере демонстрируется рисование с отсечением для получения изображения, представленного на рис 2.3.

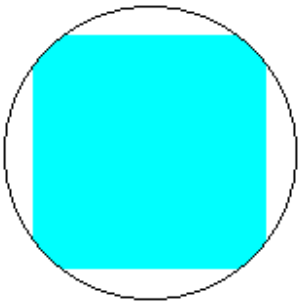


Рис. 2.3. Пример отсечения

Листинг 2.30. Пример рисования с отсечением

```
void Display(HDC hdc)
{
    // создаем объект класса Graphics для рисования
    Graphics g(hdc);
    // выполняем очистку перед рисованием
    g.Clear(Color::White); // белый цвет фона

    // создаем перо черного цвета
    Pen pen(Color::Black);
    // создаем сплошную кисть
    SolidBrush brush(Color::Aqua);

    // создаем прямоугольный регион
    Region rgn1(Rect(50, 50, 120, 120));
    // создаем бесконечный регион
    Region rgn2;

    // задаем регион отсечения
    g.SetClip(&rgn1);

    // рисуем круг
    g.FillEllipse(&brush, 35, 35, 150, 150);

    // задаем регион отсечения
    g.SetClip(&rgn2);

    // рисуем окружность
    g.DrawEllipse(&pen, 35, 35, 150, 150);
} // Display
```

Задание к работе

1. Разработать в Visual C++ приложение, которое решает задачу принадлежности точки многоугольнику с помощью метода, указанного в варианте задания.
2. Разработать в Visual C++ оконное приложение, которое рисует кривую согласно варианту задания.

   При рисовании кривой необходимо использовать алгоритм отсечения, указанный в варианте задания
3. Изменить оконное приложение Win32, разработанное в лабораторной работе №1, так чтобы в нем использовалось отсечение с помощью регионов GDI+.

4. Протестировать работу приложений, разработанных в п. 1, 2 и 3. Результаты тестирования отразить в отчете.
5. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Задание
1,16	**Метод определения принадлежности точки многоугольнику:** С помощью треугольников и барицентрических координат **Уравнение кривой**: $\rho = R \cdot \sin\frac{4}{7}t, 0 \le t \le 14\pi$ **Алгоритм отсечения**: Коэна-Сазерленда
2,17	**Метод определения принадлежности точки многоугольнику:** Суммирование углов **Уравнение кривой**: $\rho = R \cdot \sin\frac{7}{4}t, 0 \le t \le 8\pi$ **Алгоритм отсечения**: Лианга-Барски
3,18	**Метод определения принадлежности точки многоугольнику:** Определение стороны **Уравнение кривой**: $\rho = R \cdot \sin\frac{5}{2}t, 0 \le t \le 4\pi$ **Алгоритм отсечения**: Лианга-Барски
4,19	**Метод определения принадлежности точки многоугольнику:** Учет числа оборотов **Уравнение кривой**: $\rho = R \cdot \sin\frac{7}{2}t, 0 \le t \le 4\pi$ **Алгоритм отсечения**: Лианга-Барски
5,20	**Метод определения принадлежности точки многоугольнику:** Учет числа пересечений **Уравнение кривой**: $\rho = R \cdot \sin\frac{4}{3}t, 0 \le t \le 6\pi$ **Алгоритм отсечения**: Коэна-Сазерленда

№	Задание
6,21	**Метод определения принадлежности точки многоугольнику:** Суммирование углов **Уравнение кривой**: $\begin{cases} x = 14\cos t + 5\cos\left(\frac{14}{5}t\right) \\ y = 14\sin t - 5\sin\left(\frac{14}{5}t\right) \end{cases}$ $0 \le t \le 10\pi$ **Алгоритм отсечения**: Лианга-Барски
7,22	**Метод определения принадлежности точки многоугольнику:** Учет числа оборотов **Уравнение кривой**: $\begin{cases} x = 31\cos t + 5\cos\left(\frac{31}{5}t\right) \\ y = 31\sin t - 5\sin\left(\frac{31}{5}t\right) \end{cases}$ $0 \le t \le 10\pi$ **Алгоритм отсечения**: Коэна-Сазерленда
8,23	**Метод определения принадлежности точки многоугольнику:** Определение стороны **Уравнение кривой**: $\begin{cases} x = 11\cos t + 10\cos\left(\frac{11}{10}t\right) \\ y = 11\sin t - 10\sin\left(\frac{11}{10}t\right) \end{cases}$ $0 \le t \le 20\pi$ **Алгоритм отсечения**: Лианга-Барски
9,24	**Метод определения принадлежности точки многоугольнику:** Учет числа пересечений **Уравнение кривой**: $\begin{cases} x = 9\cos t + 2\cos\left(\frac{9}{2}t\right) \\ y = 9\sin t - 2\sin\left(\frac{9}{2}t\right) \end{cases}$ $0 \le t \le 4\pi$ **Алгоритм отсечения**: Коэна-Сазерленда

№	Задание
10,25	**Метод определения принадлежности точки многоугольнику:** С помощью треугольников и барицентрических координат **Уравнение кривой**: $\begin{cases} x = -2\cos t + 3\cos\left(\frac{-2}{3}t\right) \\ y = -2\sin t - 3\sin\left(\frac{-2}{3}t\right) \end{cases}$ $0 \le t \le 6\pi$ **Алгоритм отсечения**: Лианга-Барски
11,26	**Метод определения принадлежности точки многоугольнику:** Учет числа оборотов **Уравнение кривой**: $x = 24\cos t - 5\cos\left(\frac{24}{5}t\right)$ $y = 24\sin t - 5\sin\left(\frac{24}{5}t\right)$ $0 \le t \le 10\pi$ **Алгоритм отсечения**: Коэна-Сазерленда
12,27	**Метод определения принадлежности точки многоугольнику:** Суммирование углов **Уравнение кривой**: $x = 41\cos t - 5\cos\left(\frac{41}{5}t\right)$ $y = 41\sin t - 5\sin\left(\frac{41}{5}t\right)$ $0 \le t \le 10\pi$ **Алгоритм отсечения**: Лианга-Барски
13,28	**Метод определения принадлежности точки многоугольнику:** Определение стороны **Уравнение кривой**: $x = 41\cos t - 10\cos\left(\frac{41}{10}t\right)$ $y = 41\sin t - 10\sin\left(\frac{41}{10}t\right)$ $0 \le t \le 20\pi$ **Алгоритм отсечения**: Коэна-Сазерленда

№	Задание
14,29	**Метод определения принадлежности точки многоугольнику:** С помощью треугольников и барицентрических координат **Уравнение кривой**: $x = 13\cos t - 2\cos\left(\frac{13}{2}t\right)$ $y = 13\sin t - 2\sin\left(\frac{13}{2}t\right)$ $0 \le t \le 4\pi$ **Алгоритм отсечения**: Лианга-Барски
15,30	**Метод определения принадлежности точки многоугольнику:** Учет числа пересечений **Уравнение кривой**: $x = 11\cos t - 3\cos\left(\frac{11}{3}t\right)$ $y = 11\sin t - 3\sin\left(\frac{11}{3}t\right)$ $0 \le t \le 6\pi$ **Алгоритм отсечения**: Коэна-Сазерленда

Контрольные вопросы

1. Что называется компьютерной геометрией?
2. Как выполняется сравнение значений с плавающей точкой?
3. Что называется мировым окном?
4. Что называется портом просмотра?
5. Как производится преобразование из мирового окна в порт просмотра?
6. Как производится преобразование из порта просмотра в мировое окно?
7. Что называется задачей принадлежности точки полигону?
8. В чем заключается метод суммирования углов?
9. В чем заключается метод определения стороны?
10. В чем заключается метод барицентрических координат?
11. В чем заключается метод трассировки луча?
12. В чем заключается метод учета числа оборотов?
13. Что называется задачей отсечения?
14. В чем заключается алгоритм Коэна-Сазерленда?
15. В чем заключается алгоритм Сайреса-Бека?
16. В чем заключается алгоритм Лианга-Барски?

17. Как создать кривую с помощью ломаной линии?
18. Как рисуются ломаные линии?
19. Как в GDI+ создать и изменять регионы?
20. Как в GDI+ определить принадлежность точки и прямоугольника региону?
21. Как в GDI+ использовать отсечение?

ЛАБОРАТОРНАЯ РАБОТА № 3. ОСНОВЫ КОМПЬЮТЕРНОЙ АНИМАЦИИ

Цель работы

Изучение основных методов и получение практических навыков создания компьютерной анимации.

Основные понятия

Анимация представляет собой создание иллюзии движущегося изображения с помощью последовательности неподвижных изображений (кадров), сменяющих друг друга с некоторой частотой. Эффект движения создается благодаря инерции зрительного восприятия.

Количество сменяемых кадров за единицу времени называется *кадровой частотой* (Frames per Second, FPS). После проведенных исследований было установлено, что для нормального восприятия анимационного изображения FPS должна составлять от 24 до 30 кадров в секунду. Если менять кадры медленнее, то изображение начинает раздражающе «скакать» и иллюзия движения не получится. Если же наоборот ускориться, например, до 50 кадров в секунду, то особого реализма изображению это не добавит. Необходимо отметить, что в компьютерных играх под кадровой частотой понимают частоту создаваемых кадров.

Покадровая анимация

Схематически покадровую анимацию можно представить в виде последовательности отдельных готовых изображений (кадров). Такая схема называется *раскадровкой*. Раскадровка позволяет получить представление о том, что происходит в анимации.

На рис. 3.1 представлена раскадровка бегущей лошади, содержащая пять кадров. Этих кадров достаточно для того, чтобы создать примитивную анимацию. Однако потребуется большее количество кадров для того, чтобы анимация выглядела плавной и реалистичной.

Покадровая анимация с использованием GDI+

Некоторые графические форматы поддерживают хранение сразу нескольких изображений (кадров) в одном файле. К таким графическим форматам относится формат GIF, который активно используется в компьютерной графике для создания анимации.

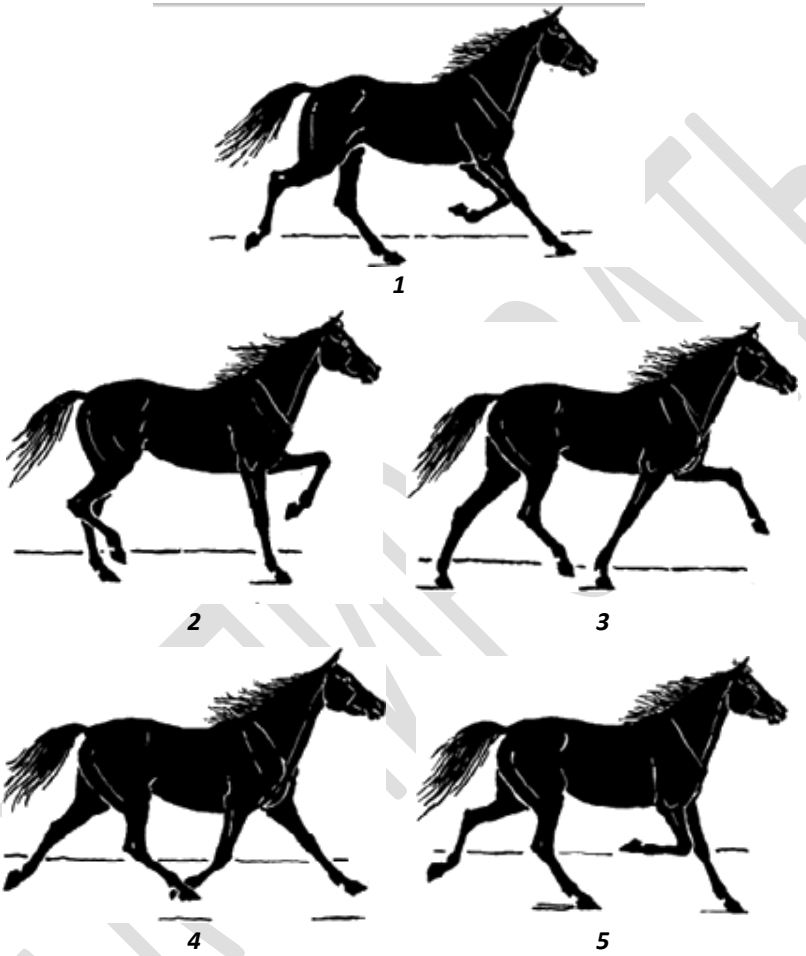


Рис. 3.1. Раскадровка бегущей лошади

В GDI+ загрузка GIF-изображения (или любого другого многокадрового изображения) ничем не отличается от обычной загрузки графического файла (см. Лабораторная работа №1 раздел «Изображения»).

После того, как изображение будет загружено. Узнать число кадров в загруженном изображении можно с помощью метода `GetFrameCount` класса `Image`. Этот метод имеет следующий прототип:

```
UINT GetFrameCount(const GUID *dimensionID);
```

Параметр *dimensionID* указывает на GUID-константу, определяющую тип хранимых кадров. В заголовочном файле gdiplusimaging.h определены следующие возможные значения:

- `FrameDimensionTime` – кадры разделяемые по порядку следования в анимации;
- `FrameDimensionResolution` – кадры разделяемые по разрешению изображения;
- `FrameDimensionPage` – кадры разделяемые по порядковому номеру страницы.

Для GIF-изображений необходимо использовать константу `FrameDimensionTime`. Следующий небольшой пример демонстрирует загрузку GIF-изображения и определение количества кадров в нем:

```
1   UINT frameCount = 0;
2
3   // загрузка из файла Sample.gif
4   Image *image = Image::FromFile(L"Sample.gif");
5
6   if (NULL != image) // файл успешно загружен
7   {
8       // определение количества кадров в изображении
9       nFrameCount = image->GetFrameCount(&FrameDimensionTime);
10  } // if
```

Перед отображением кадра его необходимо сделать активным с помощью метода `SelectActiveFrame` класса `Image`. Этот метод имеет следующий прототип:

```
Status SelectActiveFrame(const GUID* dimensionID,
    UINT frameIndex);
```

Первый параметр, *dimensionID*, указывает на GUID-константу, определяющую тип хранимых кадров.

Второй параметр, *frameIndex*, задает номер кадра, который необходимо сделать активным.

В листинге 3.1 показан фрагмент программного кода оконной процедуры окна, в котором будет отображаться анимация.

Листинг 3.1. Функция, возвращающая количество кадров (по времени)

```
1   case WM_CREATE:
2       image = Image::FromFile(L"Sample.gif"); // загрузка из файла
    Sample.gif
3       if (NULL == image) return -1; // загрузка не удалась
4
```

```
5       frameIndex = 0;
6
7       // определение количества кадров в изображении
8       frameCount = image->GetFrameCount(&FrameDimensionTime);
9
10      SetTimer(hWnd, 1, 40, NULL); // устанавливаем таймер ~ 25 FPS
11      return 0;
12
13  case WM_TIMER:
14      // увеличиваем индекс кадра (циклически)
15      frameIndex = (frameIndex + 1) % frameCount;
16      // делаем кадр активным
17      image->SelectActiveFrame(&FrameDimensionTime, frameIndex);
18
19      // требуем обновления клиентской области окна (перерисовки)
20      InvalidateRect(hWnd, NULL, FALSE);
21      return 0;
22
23  case WM_PAINT:
24      {
25          PAINTSTRUCT ps;
26
27          // начинаем процесс рисования
28          HDC hdc = BeginPaint(hWnd, &ps);
29
30          Display(hdc); // вызываем фукнцию рисования
31
32          // завершаем процесс рисования
33          EndPaint(hWnd, &ps);
34      }
35      return 0;
36
37  case WM_DESTROY:
38      if (NULL != image) delete image; // удаляем объект
39      return 0;
```

В указанном фрагменте программного кода используется несколько глобальных переменных, которые определены следующим образом:

```
1   Image *image; // GIF-изображение
2
3   UINT frameIndex; // индекс активного кадра
4   UINT frameCount; // количество кадров
```

В следующем примере показана функция отображения, которая используется в программном коде из приведенного ранее примера (см. листинг 3.1) для отображения активного кадра GIF-изображения.

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       g.DrawImage(image, 0, 0); // отображаем активный кадр
9   } // Display
```

Программная анимация

При создании анимации можно обойтись без длинного списка готовых кадров. Вместо этого можно готовить кадр «на лету». Такая разновидность анимации называется программной.

Программная анимация имеет несколько преимуществ по отношению к покадровой. Во-первых, программная анимация не нуждается в создании и хранении кадров. Во-вторых, программная анимация позволяет создавать более динамичную анимацию.

Анимация по ключевым кадрам

Ключевой кадр – состояние объекта (или группы объектов) в определенный момент времени. В качестве примера на рис. 3.2 показаны несколько ключевых кадров с шагающей фигуркой.

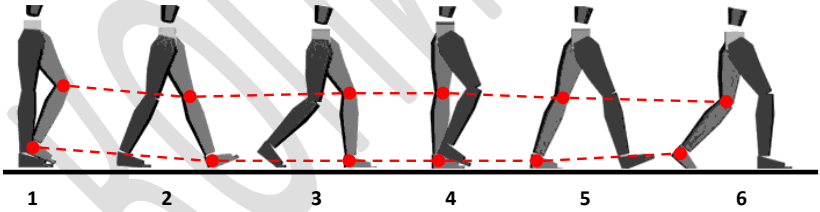


Рис. 3.2. Анимация по ключевым кадрам

Для подготовки промежуточных кадров определяется положение объектов между ключевыми кадрами. Процесс подготовки промежуточных кадров называется *твининг* (tweening). Твининг заключается в интерполяции между соответствующими вершинами объектов ключевых кадров (см. рис. 3.2). Таким образом, каждая вершина P объекта промежуточного кадра вычисляется по следующей формуле:

$$P = A \cdot (1 - t) + B \cdot t, \tag{3.1}$$

где A и B – соответствующие вершины объектов ключевых кадров; параметр t должен изменяться от `0` до `1`.

В следующем примере показаны функции, которые вычисляют по формуле (3.1) промежуточные координаты точек между двумя соответствующими точками.

Листинг 3.2. Функции нахождения промежуточных точек

```
PointF Tween(const PointF &A, const PointF &B, float t)
{
    return PointF(
        A.X*(1.f - t) + B.X*t, A.Y*(1.f - t) + B.Y*t);
} // Tween

void Tween(const PointF *A, const PointF *B, PointF *P, int
count, float t)
{
    for (int i = 0; i < count; ++i) P[i] = Tween(A[i], B[i], t);
} // Tween
```

При создании анимации по ключевым кадрам объект не обязательно должен менять свое положение (см. рис. 3.2), он может изменять свою форму или внешний вид. На рис. 3.3 проиллюстрирован пример изменения формы объекта. При $t = 0$ и $t = 1$ объект имеет форму, соответствующую ключевым кадрам, а при $t = 0.5$ – форму, соответствующую промежуточному кадру.

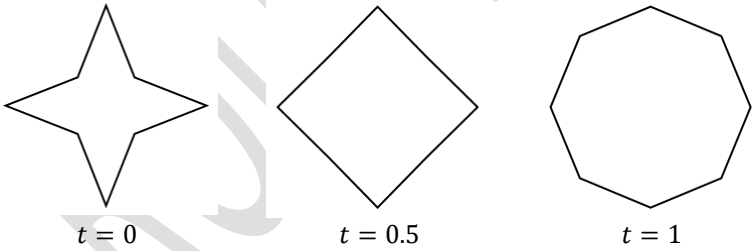


Рис. 3.3. Изменение формы объекта

Аффинные преобразования

Для создания анимации можно также использовать аффинные преобразования (affine transformations), которые применяются к объектам изображения для изменения их положения или формы.

Для работы с аффинными преобразованиями в GDI+ есть класс `Matrix`, который определен в заголовочном файле gdiplusmatrix.h. Класс `Matrix` представляет матрицы размером 3×3 и имеет следующие конструкторы:

```
Matrix();
```

```
Matrix(REAL m11, REAL m12, REAL m21, REAL m22,
    REAL dx, REAL dy);

Matrix(const Rect &rect, const Point *dstplg);

Matrix(const RectF &rect, const PointF *dstplg);
```

Первый конструктор создает объект класса `Matrix`, представляющего единичную матрицу. Второй конструктор создает объект класса `Matrix` на основе констант m_{11}, m_{12}, m_{21}, m_{22}, dx и dy, определяющих аффинное преобразование. Из следующей формулы видно, как будут располагаться задаваемые константы в полученной матрице:

$$\begin{pmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ dx & dy & 1 \end{pmatrix} \quad (3.2)$$

Третий и четвертый конструкторы создают объект класса `Matrix` на основе объекта класса `Rect` (`RectF`), который задается параметром *`rect`*, и объекта класса `Point` (`PointF`), который задается параметром *`dstplg`*. Согласно формуле (3.2) поля *`X`*, *`Y`*, *`Width`* и *`Height`* объекта *`rect`* задают значения элементов m_{11}, m_{12}, m_{21} и m_{22}, а поля *`X`* и *`Y`* объекта *`dstplg`* задают значения элементов dx и dy.

Прежде чем перейти к изучению основных методов класса `Matrix`, следует отметить, что в этом классе, как и в других классах GDI+, есть методы `GetLastStatus` и `Clone`, который имеет следующий прототип:

```
Matrix* Clone() const;
```

В случае успеха метод `Clone` возвращает указатель на объект класса `Matrix`, который должен быть удален вызовом оператора `delete` после того как станет не нужен. В случае ошибки – `NULL`.

В классе `Matrix` также определен метод `Equals`, который выясняет равенство двух объектов класса `Matrix`. Метод `Equals` имеет следующий прототип:

```
BOOL Equals(const Matrix *matrix) const;
```

Если объекты класса `Matrix` равны метод `Equals`, возвращает `TRUE`, иначе – `FALSE`.

Элементы матрицы

Значения элементов матрицы можно задавать при создании объекта класса `Matrix`. Можно также изменять значения элементов матрицы с помощью метода `SetElements`:

```
Status SetElements(REAL m11, REAL m12, REAL m21, REAL m22,
    REAL dx, REAL dy);
```

Параметры *m11*, *m12*, *m21*, *m22*, *dx* и *dy* задают новые значения констант m_{11}, m_{12}, m_{21}, m_{22}, dx и dy из формулы (3.2), соответственно.

Получить значения элементов матрицы можно с помощью метода `GetElements`:

```
Status GetElements(REAL *m) const;
```

Параметр *m* – массив из шести элементов, в которые будут записаны значения элементов матрицы в следующем порядке m_{11}, m_{12}, m_{21}, m_{22}, dx и dy.

Также в классе `Matrix` есть два метода, которые возвращают соответственно значения элементов dx и dy – `OffsetX` и `OffsetY`:

```
REAL OffsetX() const;
REAL OffsetY() const;
```

Умножение матриц

Для умножения матрицы на другую матрицу в классе `Matrix` есть метод `Multiply`:

```
Status Multiply(const Matrix *matrix,
    MatrixOrder order = MatrixOrderPrepend);
```

Первый параметр, *matrix*, указывает на матрицу, с которой производится перемножение.

Второй параметр, *order*, определяет, как именно должны перемножаться матрицы. Этот параметр должен принимать одно из следующих значений перечисляемого типа `MatrixOrder`:

- `MatrixOrderPrepend` – матрица, задаваемая параметром *matrix*, умножается слева на текущую матрицу;
- `MatrixOrderAppend` – матрица, задаваемая параметром *matrix*, умножается справа на текущую матрицу.

В классе `Matrix` определено еще несколько методов выполняющих перемножение матриц, но в отличие от метода `Multiply` эти методы умножают текущую матрицу на одну из матриц элементарного преобразования. В табл. 3.1 перечислены эти методы класса `Matrix`.

Таблица 3.1. Методы умножения матриц класса `Matrix`

Метод	Описание
`Status Rotate(REAL angle, MatrixOrder order = MatrixOrderPrepend);`	Умножает текущую матрицу на матрицу поворота вокруг начала координат. Параметр *angle* задает угол поворота

Окончание табл. 3.1

Метод	Описание
`Status RotateAt(REAL angle, const PointF &center, MatrixOrder order = MatrixOrderPrepend);`	Умножает текущую матрицу на матрицу поворота вокруг заданной точки. Параметр *angle* задает угол (в градусах) поворота, а параметр *center* – точку, вокруг которой выполняется поворот
`Status Scale(REAL scaleX, REAL scaleY, MatrixOrder order = MatrixOrderPrepend);`	Умножает текущую матрицу на матрицу масштабирования. Параметры *scaleX* и *scaleY* задают соответствующие масштабные множители
`Status Shear(REAL shearX, REAL shearY, MatrixOrder order = MatrixOrderPrepend);`	Умножает текущую матрицу на матрицу сдвига. Параметры *shearX* и *shearY* задают соответствующие коэффициенты, определяющие сдвиг
`Status Translate(REAL offsetX, REAL offsetY, MatrixOrder order = MatrixOrderPrepend);`	Умножает текущую матрицу на матрицу переноса. Параметры *offsetX* и *offsetY* задают компоненты вектора переноса

При успешном выполнении методов класса `Matrix`, предназначенных для перемножения матриц, текущая матрица будет заменена результатом умножения.

Единичная матрица

В процессе построения анимация может понадобиться сделать из матрицы единичную матрицу. Для этого можно воспользоваться методом `Reset` класса `Matrix`:

```
Status Reset();
```

Проверить является ли матрица единичной можно с помощью метода `IsIdentity` класса `Matrix`:

```
BOOL IsIdentity() const;
```

Для единичной матрицы метод `IsIdentity`, вернет `TRUE`, а для всех остальных матриц – `FALSE`.

Обратная матрица

Как известно для отмены аффинных преобразований используются обратные матрицы. Для того чтобы сделать из матрицы обратную матрицу нужно воспользоваться методом `Invert` класса `Matrix`:

```
Status Invert();
```

Проверить является ли матрица обратной можно с помощью метода `IsInvertible` класса `Matrix`:

```
BOOL IsInvertible() const;
```

Для обратной матрицы метод `IsInvertible`, вернет `TRUE`, а для всех остальных матриц – `FALSE`.

Применение аффинных преобразований

В GDI+ аффинные преобразования можно применять, как отдельным точкам и векторам, так и ко всему изображению. Кроме того GDI+ позволяет применять аффинные преобразования для различных графических объектов.

Для того чтобы применить аффинные преобразования к отдельным точкам следует воспользоваться методом `TransformPoints` класса `Matrix`. Этот метод имеет следующие прототипы:

```
Status TransformPoints(Point *pts, INT count = 1) const;
Status TransformPoints(PointF* pts, INT count = 1) const;
```

Первый параметр, *pts*, указывает на массив объектов класса `Point` (`PointF`), каждый из которых содержит точку, подлежащую преобразованию. Если метод `TransformPoints` будет выполнен успешно, в массив, на который указывает параметр *pts*, будут сохранены координаты новых точек.

Второй параметр, *count*, определяет размер массива, на который указывает *pts* параметр. По умолчанию параметр *count* принимает значение равное `1`.

Важно отметить, что в GDI+ перемножение точки и матрицы выполняется в однородных координатах по следующей формуле:

$$(x' \quad y' \quad 1) = (x \quad y \quad 1)\begin{pmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ dx & dy & 1 \end{pmatrix}, \tag{3.3}$$

где x и y – координаты точки, которая умножается на матрицу.

В следующем примере демонстрируется рисование двух треугольников, изображенных на рис. 3.4, с использованием аффинного преобразования.

Листинг 3.3. Пример использования аффинного преобразования

```
1  void Display(HDC hdc)
2  {
3      // создаем объект класса Graphics для рисования
4      Graphics g(hdc);
```

```
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // создаем перо черного цвета
9       Pen blackPen(Color::Black, 2.f);
10      // создаем перо красного цвета
11      Pen redPen(Color::Red, 2.f);
12
13      // вершины треугольника
14      Point points[3] = {
15          Point(100, 100), // точка (100,100)
16          Point(200, 130), // точка (200,130)
17          Point(110, 200)  // точка (110,200)
18      };
19
20      // рисуем контур треугольника
21      g.DrawPolygon(&blackPen, points, 3);
22
23      // создаем единичную матрицу
24      Matrix mx;
25      // умножаем матрицу на матрицу переноса
26      mx.Translate(50.f, 0.f);
27
28      // выполняем преобразование - переносим треугольник вправо
29      mx.TransformPoints(points, 3);
30
31      // рисуем контур треугольника
32      g.DrawPolygon(&redPen, points, 3);
33  } // Display
```

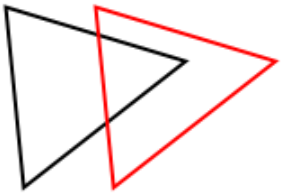


Рис. 3.4. Результат использования преобразования «перенос»

Для того чтобы применить аффинные преобразования к векторам нужно воспользоваться методом `TransformVectors` класса `Matrix`. Этот метод имеет следующие прототипы:

```
Status TransformVectors(Point *pts, INT count = 1) const;
Status TransformVectors(PointF* pts, INT count = 1) const;
```

Первый параметр, *pts*, указывает на массив объектов класса Point (PointF), каждый из которых содержит компоненты вектора, подлежащие преобразованию. Если метод TransformVectors будет выполнен успешно, в массив, на который указывает параметр *pts*, будут сохранены новые компоненты вектора.

Второй параметр, *count*, определяет размер массива, на который указывает *pts* параметр. По умолчанию параметр *count* принимает значение равное 1.

В GDI+ перемножение вектора и матрицы выполняется в однородных координатах по следующей формуле:

$$(x' \quad y' \quad 0) = (x \quad y \quad 0)\begin{pmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ dx & dy & 1 \end{pmatrix}, \tag{3.4}$$

где x и y – компоненты вектора, который умножается на матрицу.

Для применения аффинных преобразований ко всему рисуемому изображению необходимо использовать один из методов класса Graphics, перечисленных в табл. 3.2.

Таблица 3.2. Методы аффинных преобразований класса Graphics

Метод	Описание
`Status SetTransform(` `  const Matrix *matrix);`	Задает новую матрицу аффинных преобразований

`Status MultiplyTransform(`
`  const Matrix *matrix,`
`  MatrixOrder order =`
`  MatrixOrderPrepend);`
	Умножает текущую матрицу аффинных преобразований на другую матрицу

`Status ResetTransform();`	Устанавливает текущую матрицу аффинных преобразований в единичную матрицу (фактически отменяет преобразование)

`Status RotateTransform(`
`  REAL angle,`
`  MatrixOrder order =`
`  MatrixOrderPrepend);`
	Умножает текущую матрицу аффинных преобразований на матрицу поворота вокруг начала координат.
	Параметр *angle* задает угол (в градусах) поворота

`Status ScaleTransform(`
`  REAL scaleX,`
`  REAL scaleY,`
`  MatrixOrder order =`
`  MatrixOrderPrepend);`
	Умножает текущую матрицу аффинных преобразований на матрицу масштабирования.
	Параметры *scaleX* и *scaleY* задают соответствующие масштабные множители

Окончание табл. 3.2

Метод	Описание

`Status TranslateTransform(`
`  REAL offsetX,`
`  REAL offsetY,`
`  MatrixOrder order =`
`  MatrixOrderPrepend);`
	Умножает текущую матрицу аффинных преобразований на матрицу переноса.
	Параметры *offsetX* и *offsetY* задают компоненты вектора переноса

В следующем примере показано, как применять аффинные преобразования ко всему рисуемому изображению:

Листинг 3.4. Пример применения преобразования для всего изображения

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5       // выполняем очистку перед рисованием
6       g.Clear(Color::White); // белый цвет фона
7
8       // применим масштабирование
9       g.ScaleTransform(2.f, 1.f);
10
11      // создаем перо черного цвета
12      Pen blackPen(Color::Black);
13      // создаем сплошную кисть
14      SolidBrush solidBrush(Color::Yellow);
15
16      // рисуем закрашенный эллипс
17      g.FillEllipse(&solidBrush, 10, 10, 160, 80);
18      // рисуем контур эллипса
19      g.DrawEllipse(&blackPen, 10, 10, 160, 80);
20  } // Display
```

Следует отметить, что некоторые классы GDI+, представляющие различные графические объекты, могут иметь методы для применения к этим объектам аффинных преобразований. Более подробную информацию можно найти в документации Platform SDK.

Двойная буферизация

Распространенной проблемой при создании компьютерной анимации является мерцание. Мерцание возникает, когда графические операции, выполняющие несколько сложных операций рисования, неоднократно повторяются в течение короткого отрезка времени.

Для устранения мерцания традиционно используется *двойная буферизация* (*double buffering*). При использовании двойной буферизации результаты всех операций рисования сначала помещаются в специальный буфер, а лишь затем готовое изображение будет выводиться на экран из буфера, в котором оно хранится.

Двойная буферизация с использованием GDI+

В приведенном далее примере показана функция отображения, в которой создаются буфер, в котором будет храниться рисуемое изображение, и два объект класса `Graphics`. Первый объект класса `Graphics` используется для рисования в буфер, а второй – для вывода содержимого буфера на экран, когда изображение будет готово.

Листинг 3.5. Пример функции отображения

```
1   void Display(HDC hdc)
2   {
3       // создаем объект класса Graphics для рисования
4       Graphics g(hdc);
5
6       // определяем размер области рисования
7       Rect rect;
8       g.GetVisibleClipBounds(&rect);
9
10      // создаем буфер, кот. будет хранить рисуемое изображение
11      Bitmap backBuffer(rect.Width, rect.Height, &g);
12      // создаем объект класса Graphics для рисования в буфер
13      Graphics temp(&backBuffer);
14
15      // рисуем в буфер...
16
17      // выполняем очистку перед рисованием
18      temp.Clear(Color::White); // белый цвет фона
19      // рисуем окружность пером черного цвета
20      temp.DrawEllipse(&Pen(Color::Black), 35, 35, 150, 150);
21
22      // выводим на экран готовое изображение из буфера
23      g.DrawImage(&backBuffer, rect);
24  } // Display
```

Нужно отметить, что мерцание может присутствовать, даже при использовании двойной буферизации. Дело в том, что при перерисовке окна перед сообщением `WM_PAINT` окну сначала посылается сообщение `WM_ERASEBKGND` и при этом функция `DefWindowProc`, обрабатывая сообщение `WM_ERASEBKGND`, будет закрашивать клиентскую область окна

кистью, указанной в свойствах оконного класса. Для устранения такого рода мерцания, следует при регистрации оконного класса установить поле hbrBackground структуры WNDCLASSEX в NULL или подавить обработку сообщения WM_ERASEBKGND следующим образом:

```
1   case WM_ERASEBKGND:
2       return 1;
```

Задание к работе

1. Изучить возможности создания компьютерной анимации с помощью GDI+.
2. Разработать в Visual C++ оконное приложение, которое создает анимированное изображение согласно варианту задания.

   При создании изображения обязательно использовать все рассмотренные виды компьютерной анимации. Кроме того, следует применить не менее одного сложного аффинного преобразования
3. Протестировать работу приложения, разработанного в п. 2. Результаты тестирования отразить в отчете.
4. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Рисунок	№	Рисунок
1,16	Автомобиль	9,24	Грузовик
2,17	Паровоз	10,25	Катер
3,18	Пароход	11,26	Вертолет
4,19	Самолет	12,27	Велосипед
5,20	Ракета	13,28	Мотоцикл
6,21	Автобус	14,29	Парусник
7,22	Луноход	15,30	Звездолет
8,23	Трактор		

Контрольные вопросы

1. Что называется анимацией?
2. Что называет кадровой частотой?

3. Что называется покадровой анимацией?
4. Как создать покадровую анимацию используя GDI+?
5. Что называется программной анимацией?
6. Какими преимуществами обладает программная анимация по отношению к покадровой?
7. В чем заключается анимация по ключам?
8. Что называется аффинным преобразованием?
9. Что такое однородные координаты?
10. Какими свойствами обладают аффинные преобразования?
11. Что называется тождественным преобразование?
12. Что называется элементарным аффинным преобразованием?
13. Что называется сложным аффинным преобразованием?
14. Как осуществляется отмена аффинных преобразований?
15. Как применять аффинные преобразования в GDI+?
16. Для чего используется двойная буферизация?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Аммерал, Л.* Принципы программирования в машинной графике / Л. Аммерал; пер. с англ. В. А. Львова. – М.: Сол Систем, 1992. – 224 с.
2. *Божко, А.Н.* Компьютерная графика: Учеб. пособие для вузов / А.Н. Божко, Д.М. Жук, В.Б. Маничев. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2007. – 392 с.: ил.
3. Полный справочник по C++, 4-е издание / Г. Шилдт; пер. с англ. Д.А. Клюшина. – М.: Вильямс, 2007. – 800 с.
4. *Пореев, В.Н.* Компьютерная графика / В.Н. Пореев. – СПб.: БХВ-Петербург, 2002. – 432 с.: ил.
5. *Роджерс, Д.* Алгоритмические основы машинной графики / Д. Роджерс; пер с англ. С.А. Вичеса [и др.]. – М. Мир, 1989. – 512 с.
6. *Роджерс, Д.* Математические основы машинной графики / Д. Роджерс, А. Адамс; пер. с англ. П.А. Монахова [и др.]. - 2-е изд., доп. и перераб. – М. Мир, 2001. – 604 с.
7. *Шикин, А.В.* Компьютерная графика. Полигональные модели / А.В. Шикин., А.В. Боресков; – М.: ДИАЛОГ-МИФИ, 2001. – 463 с.
8. *Шикин, Е.В.* Кривые и поверхности на экране компьютера. Руководство к сплайнам для пользователей / Е.В. Шикин, А.И. Плис. – М.: ДИАЛОГ-МИФИ, 1996. – 240 с.

Учебное издание

Глухоедов Андрей Владимирович

КОМПЬЮТЕРНАЯ ГЕОМЕТРИЯ И ГРАФИКА

Лабораторный практикум

Подписано в печать 25.11.15. Формат 60x84/16. Усл. печ. л. 10,7. Уч.-изд. л. 11,5.
Тираж 57 экз. Заказ Цена
Отпечатано в Белгородском государственном технологическом университете
им. В.Г. Шухова
308012, г. Белгород, Костюкова, 46