

WEEK 2

DAY 8

그래픽, 그림 그리기, 비트맵

지금까지 보아왔던 것들이 모두 윈도우 애플리케이션에서 혼한 것이기 때문에 또 이런말을 하고 싶지는 않지만, 대다수의 애플리케이션이 그래픽과 이미지 표시 기능을 가지고 있으며, 어떤 애플리케이션의 경우에는 그래픽이 자신의 전문 기능인 것도 있다. 어쨌거나, 여러분의 애플리케이션에 이러한 기능을 포함시키는데 필요한 것들을 이해하는 것 자체가 윈도우 플랫폼을 위한 프로그래밍에 있어서 열쇠가 된다. 이미 점을 그리고, 여러 개의 선을 이어서 기괴한 형상의 그림도 그릴 수 있는 방법은 공부했었다. 이번 장에서는 이런 단순한 기능을 넘어서 애플리케이션에 좀더 고급스런 그래픽 기능을 추가하는 방법에 대해 배워보도록 하자.

- ❖ 윈도우 운영체제가 디바이스 컨텍스트를 사용해서 그리기 명령을 그래픽 출력으로 변환하는 과정
- ❖ 여러 가지 매핑 모드를 통해 그래픽 출력의 조절 수준을 결정하는 방법
- ❖ 윈도우 운영체제가 펜과 브러쉬를 사용해서 그래픽 이미지의 여러 가지 영역을 그려내는 과정
- ❖ 비트맵을 로드하고 표시하는 방법

GDI에 대하여

윈도우 운영체제는 애플리케이션에서의 그래픽 사용에 대해 두 가지 레벨의 추상개념을 제공한다. DOS 프로그래밍 시절에는 애플리케이션에 이미지 딱지를 그리기 위해 그래픽 하드웨어를 거의 일일이 조절하다시피 했었다. 따라서, 엔드유저의 컴퓨터가 사용하고 있을만한 모든 그래픽 카드에 대한 광범위한 지식과 이해가 필요했고, 심지어는 모니터와 해상도까지 고려해야 했었다. 물론 애플리케이션 제작에 사용할 수 있는 그래픽 라이브러리도 몇 개 나왔지만, 대체적으로 그래픽 기능을 애플리케이션에 탑재시킨다는 것은 보통 코딩 가지고는 될 일이 아니었다.

윈도우 운영체제와 함께 마이크로소프트는 이 작업을 매우 단순하게 만들었다. 우선, 마이크로소프트는 모든 윈도우 애플리케이션에서 사용할 수 있는 가상 그래픽 디바이스(이것을 GDI라고 부른다)를 제공하였는데, 이 가상 디바이스는 하드웨어에 구애 받지 않고 모든 그래픽 하드웨어에 대해 동일하게 작동한다. 이러한 일관성으로 인해 프로그래머들은 원하는 이미지를 마음껏 그릴 수 있게 되었다. 왜냐하면, 그래픽 명령을 하드웨어가 이해하는 출력으로 바꾸어 주는 것이 여러분의 문제가 아니기 때문이다.

■ 디바이스 컨텍스트

그래픽 기능을 구사하기 전에 필요한 것이 하나 있는데, 바로 그래픽이 표시될 디바이스 컨텍스트(Device Context : DC)를 얻는 일이다. 디바이스 컨텍스트는 시스템, 애플리케이션, 그리고 그래픽을 그리고자 하는 윈도우에 관한 정보를 가지고 있다. 운영체제는 이 디바이스 컨텍스트를 사용해서 어떤 부분에서 그래픽이 그려지는지, 얼마나 많은 영역이 보이는지, 그려지는 영역이 화면 내의 어느 위치인지를 알아낼 수 있다.

여러분이 그래픽을 그릴 때는 애플리케이션 윈도우의 디바이스 컨텍스트에 그리는 것이라고 생각하면 틀리지 않는다. 어느 때이건 간에 이 윈도우는 전체화면이 될 수 있고, 최소화될 수도 있으며, 부분적으로 가려질 수도 있고 완전히 가려질 수 있다. 이러한 상태는 사실 윈도우 운영체제가 신경 써줄 문제이지 여러분 관심사는 아니다. 단지 여러분은 디바이스 컨텍스트를 사용해서 윈도우에 그림만 그리면 된다. 윈도우 운영체제는 각각의 디바이스 컨텍스트의 행보를 추적하고 있다가, 이것을 사용해서 여러분이 그런 것 중에 어떤 부분이 얼마큼 실제로 사용자에게 보여질지 결정한다. 정리하면, 그래픽 표시에 사용되는 디바이스 컨텍스트는 여러분이 윈도우 안에서 그런 그림이 화면에 보여지는 부분이라고 생각하면 된다.

디바이스 컨텍스트가 대부분의 그래픽 기능을 위해 가장 많이 사용하는 두 가지 자원이 있는데, 펜과 브러쉬이다. 실세계의 펜과 브러쉬(붓)를 생각하면 아주 간단할 것이다. 펜으로는 선과 도형의 외곽선을 그리고, 브러쉬는 영역을 칠한다.

디바이스 컨텍스트 클래스

비주얼 C++에서는 MFC 디바이스 컨텍스트 클래스(CDC)를 통해 원, 사각형, 직선, 곡선 등을 그리는 기능을 제공하고 있다. 사실 그림을 그리는데 필요한 함수들은 애플리케이션에 그림을 그리기 위해 모두 디바이스 컨텍스트의 정보를 사용하기 때문에 CDC의 멤버로 넣어두는 것이 합당하다고 생각된 모양이다.

그림이 그려지는 윈도우의 MFC 클래스의 포인터를 생성자 인수로 넘겨서 디바이스 컨텍스트 클래스의 인스턴스를 선언하면 디바이스 컨텍스트가 만들어진다. 디바이스 컨텍스트를 MFC 클래스로 관리하면 클래스 생성자와 소멸자에서 자동으로 할당과 해제를 맡아주기 때문에 간편하다.

Note

디바이스 컨텍스트 개체는 다른 그리기 개체와 마찬가지로 윈도우 운영체제의 리소스로 분류되어 있다. 윈도우 운영체제가 가질 수 있는 리소스에는 제한이 있으며, 최신 버전의 윈도우 운영체제의 경우 사용할 수 있는 리소스의 수가 많기는 하지만 아직도 애플리케이션이 리소스를 할당한 다음 제 때에 삭제해 주지 않으면 전체 리소스가 고갈되어 버릴 가능성은 남아있다. 이러한 손실을 리소스 누수(leak)라고 부르며, 메모리 누수 현상과 마찬가지로 시스템 전체를 다운시키는데 일익을 담당한다. 따라서, 어떤 함수에서 리소스를 생성했다면 이 리소스의 사용을 마칠 무렵에 반드시 삭제해 주는 것이 좋다.

리소스의 생성과 삭제를 제 때에 해주기 위해서 대부분의 프로그래머들이 취하는 방법은 리소스를 한 함수 안에서 지역 변수로만 사용하는 것이다. 예외가 한 가지 있다면 원도우 운영체제에 의해서 디바이스 컨텍스트 개체가 생성된 다음, 이벤트 처리 함수로 건네어질 때이다.

펜 클래스

이미 펜 클래스, CPen을 사용한 예는 본 적이 있을 것이다. CPen은 화면에 선을 그리는데 사용되는 1차적인 GDI 자원이다. CPen 클래스의 인스턴스를 선언할 때 선의 타입, 색깔, 두께를 설정해줄 수 있으며, 펜을 생성하고 나면 이 펜을 디바이스 컨텍스트가 현재 사용할 펜으로 선택해 주어서 이후의 그리기 명령이 이 펜을 통해 수행될 수 있도록 한다. 새 펜을 생성하고 디바이스 컨텍스트에 선택해 넣으려면 다음과 같이 하자.

```
// 디바이스 컨텍스트를 얻어낸다
CDC dc(this);
// 펜을 생성한다
CPen lPen(PS_SOLID, 1, RGB(0, 0, 0));
// 이 펜을 현재의 그리기용 펜으로 선택한다
dc.SelectObject(&lPen);
```

펜에는 여러 가지의 선 스타일, 즉 선을 그리는 패턴을 같이 설정할 수 있다. 애플리케이션에서 사용될 수 있는 기본적인 펜 스타일을 [그림 8.1]에 보였다.

그림 8.1 →

펜 스타일	PS_SOLID	---
	PS_DOT
	PS_DASH	- - -
	PS_DASDOT	- - - -
	PS_DASHDOTDOT	- - - - - - - -
	PS_NULL	
	PS_INSIDEFRAME	-----

Note

어떤 선 스타일이든지 두께를 1보다 두껍게 하면 실선 스타일로 바뀐다. 만일 PS_SOLID 이외의 다른 선 스타일을 사용하고 싶으면 반드시 두께를 1로 해야 한다.

펜이 그리는 선의 스타일뿐만 아니라 펜의 두께와 색깔도 같이 정해줄 수 있으며, 스타일, 두께, 색깔이 펜으로 그려내는 선의 외양을 결정하는 세 가지 요소가 된다.

색을 설정해 주는 단위는 RGB 값인데, 말 그대로 빨강, 녹색, 파랑의 각 밝기를 조합해서 설정하는 값이다. 각각의 값은 0부터 255의 범위를 가지며, 이 세 가지 값을 조합해서 윈도우 운영체제에 필요한 포맷으로 만들어 주려면 RGB 함수를 사용한다. [표 8.1]에 몇 가지 색깔 조합을 나타내어 보았다.

[표 8.1] 윈도우 운영체제에서 사용되는 색깔

색상	빨강	녹색	파랑
검정	0	0	0
파랑	0	0	255
어두운 파랑	0	0	128
녹색	0	255	0
어두운 녹색	0	128	0
청록색	0	255	255
어두운 청록색	0	128	128
빨강	255	0	0
어두운 빨강	128	0	0
자홍색	255	0	255
어두운 자홍색	128	0	128
노랑	255	255	0
어두운 노랑	128	128	0
회색	128	128	128
밝은 회색	192	192	192
흰색	255	255	255

브러쉬 클래스

브러쉬 클래스인 CBrush는 일정 영역을 채우는 브러쉬를 관리한다. 닫힌 도형을 하나 그리고 이 도형의 내부를 채우면, 외곽선은 디바이스 컨텍스트에서 현재 사용되는 펜으로 그려지고, 내부는 디바이스 컨텍스트에서 현재 사용되는 브러쉬로 채워진다.

다. 브러쉬는 내부를 빈틈없이 칠하는 색깔(solid color)이 될 수 있고, 빗금 패턴 또는 작은 비트맵으로 구성된 반복 패턴이 될 수 있다. 내부를 칠하는 색깔의 브러쉬를 생성하고자 한다면 다음의 함수에 색깔을 인수로 넘겨준다.

```
CBrush lSolidBrush(RGB(255, 0, 0));
```

패턴 브러쉬를 생성하려면 색깔과 패턴의 종류를 같이 설정해 주어야 한다.

```
CBrush lPatternBrush(HS_BDIAGONAL, RGB(0, 0, 255));
```

브러쉬를 생성한 다음에는 펜과 마찬가지로 디바이스 컨텍스트 개체에 선택해 넣어야 하며, 이후 브러쉬를 사용해서 그려야 하는 어떤 명령이 주어질 때마다 새로 선택된 브러쉬가 사용되게 된다.

브러쉬를 생성할 때 역시 표준 패턴을 사용할 수 있다([표 8.2] 참조). 이들 패턴뿐 아니라 HS_BITMAP이라는 비트맵 패턴을 특정한 영역에 채울 수 있는 브러쉬를 만들 수 있는데, 이 비트맵의 크기는 8×8픽셀 크기로 제한되어 있다(도구바의 비트맵이나 탐색기에 쓰이는 작은 아이콘보다도 작다). 심술궂게 큰 비트맵을 사용한다고 해도 왼쪽 상단의 8×8픽셀 부분만이 나타날 뿐이다. 어쨌거나, 비트맵 브러쉬를 만들려면 일단 비트맵 리소스를 만들고 ID를 붙여주어야 하며, 이 작업이 끝나면 다음과 같이 코딩해 주자.

```
CBitmap m_bmpBitmap;
// 이미지를 로드한다
m_bmpBitmap.LoadBitmap(IDB_MYBITMAP);
// 브러쉬를 생성한다
CBrush lBitmapBrush(&m_bmpBitmap);
```

Tip

브러쉬 패턴으로 사용될 여러분만의 패턴을 만들고 싶으면 8×8픽셀의 비트맵을 그려서 비트맵 브러쉬를 사용하면 된다. 제한된 표준 패턴 이외의 브러쉬를 사용할 수 있는 방법이다.

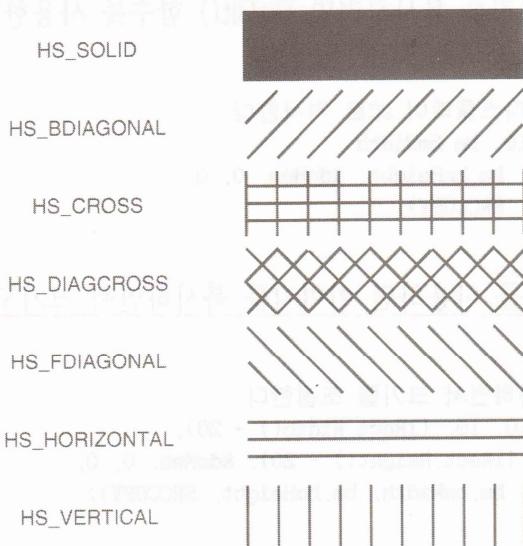
비트맵 클래스

여러분의 애플리케이션에서 이미지를 표시할 때 생각해 둘 선택 사항은 두 가지이다. 애플리케이션 안에 리소스 ID를 가진 비트맵을 만들던지, 이미지를 표시할 수 있는

그림(static picture) 컨트롤이나 ActiveX 컨트롤을 사용하는 방법이 있을 수 있으며, 이미지 표시에 관한 한 완벽한 지원을 해주는 MFC 비트맵 클래스인 CBitmap을 사용하는 방법이 있을 수 있다. 비트맵 클래스를 사용하면 시스템의 디스크에 있는 파일에서 이미지를 로드할 수 있으며, 설정한 공간에 딱 맞도록 이미지를 줄였다 하는 일도 얼마든지 가능하다.

그림 8.2 ➔

표준 브러쉬 패턴



비트맵을 리소스로 추가했다면, CBitmap 클래스의 인스턴스를 만들 때 로드할 비트맵을 설정하기 위해서 해당 비트맵의 리소스 ID를 넣어주면 된다. 파일에서 비트맵을 로드하려 한다면 LoadImage() API를 사용할 수 있다. 비트맵을 로드한 후에는, 이 비트맵의 핸들을 사용해서 CBitmap 클래스의 인스턴스에 부착하도록 한다. 바로 다음과 같이 말이다.

```
// 비트맵 파일을 로드한다
HBITMAP hBitmap = (HBITMAP)::LoadImage(AfxGetInstanceHandle(),
    m_sFileName, IMAGE_BITMAP, 0, 0,
    LR_LOADFROMFILE | LR_CREATEDIBSECTION);
// 로드된 이미지를 CBitmap 개체에 부착한다.
m_bmpBitmap.Attach(hBitmap);
```

비트맵을 CBitmap 개체로 로드한 다음에는, 디바이스 컨텍스트를 하나 더 만들어서 비트맵을 여기에 선택해 넣는다. 두번째 디바이스 컨텍스트를 만들 때는 비트맵을 선택하기 전에 반드시 원래 디바이스 컨텍스트와 호환되도록 만들어 두어야 한다. 디바이스 컨텍스트는 특정한 출력 디바이스(화면, 프린터 등)에 대해 운영체제가 만 들어내는 것이기 때문에, 두번째 디바이스 컨텍스트 역시 원래 디바이스 컨텍스트와 똑같은 출력 디바이스에 연결되도록 하는 것이다.

```
// 디바이스 컨텍스트를 생성한다
CDC dcMem;
// DC 새 디바이스 컨텍스가 진짜(원래) 디바이스 컨텍스트와 호환되도록 한다
dcMem.CreateCompatibleDC(dc);
// 새 DC에 비트맵을 선택해 넣는다
dcMem.SelectObject(&m_bmpBitmap);
```

비트맵을 호환 디바이스 컨텍스트에 선택해 넣은 다음에 원래 디바이스 컨텍스트로 해당 비트맵 이미지를 복사하려면 BitBlt() 함수를 사용한다.

```
// DC 비트맵을 디스플레이 DC로 복사한다
dc->BitBlt(10, 10, bm.bmWidth,
bm.bmHeight, &dcMem, 0, 0,
SRCCOPY);
```

StretchBlt() 함수를 사용하면 이미지를 복사하면서 크기도 조절할 수 있다.

```
// 비트맵을 복사하면서 크기를 조절한다
dc->StretchBlt(10, 10, (lRect.Width() - 20),
(lRect.Height() - 20), &dcMem, 0, 0,
bm.bmWidth, bm.bmHeight, SRCCOPY);
```

StretchBlt()를 사용하면 복사될 화면의 크기에 비트맵을 딱 맞출 수도 있게 된다.

■ 매핑 모드와 좌표 시스템

윈도우에 그림을 그릴 준비가 다 된 상태에서, 여러분이 그릴 영역과 여러분이 사용할 스케일에 대해 조정을 해줄 수 있는데, 매핑 모드와 그릴 영역을 설정해 주면 된다.

매핑 모드(Mapping Mode)는 여러분이 명시한 좌표가 화면상의 위치로 어떻게 변환될 것인지를 정해주는 시스템이다. 여러 가지 모드가 준비되어 있으며, 매핑 모드를 설정할 때 호출하는 함수는 SetMapMode()란 디바이스 컨텍스트의 멤버 함수이다.

```
dc->SetMapMode(MM_ANISOTROPIC);
```

사용 가능한 매핑 모드는 [표 8.2]에 나열해 두었다.

(표 8.2) 매핑 모드

모드	설명
MM_ANISOTROPIC	논리 단위가 임의의 축에서 임의의 단위로 변환된다.
MM_HIENGLISH	논리 단위가 <u>0.001인치 단위</u> 를 가진다. x는 오른쪽으로 갈수록 증가하며, y는 위로 갈수록 증가한다.
MM HIMETRIC	논리 단위가 <u>0.01밀리미터 단위</u> 이다. x는 오른쪽으로 갈수록 증가하며, y는 위로 갈수록 증가한다.
MM_ISOTROPIC	논리 단위가 동일한 스케일의 축에서 임의 단위로 변환된다
MM LOENGLISH	논리 단위가 0.01인치 단위이다. x는 오른쪽으로 갈수록 증가하며, y는 위로 갈수록 증가한다.
MM LOMETRIC	논리 단위가 0.1밀리미터 단위이다. x는 오른쪽으로 갈수록 증가하며, y는 위로 갈수록 증가한다.
MM_TEXT	논리 단위가 <u>1픽셀 단위</u> 이다. 논리 단위가 0.01인치 단위이다. x는 오른쪽으로 갈수록 증가하며, y는 아래로 갈수록 증가한다.
MM_TWIPS	논리 단위가 1/20포인트(약 1/1440인치) 단위이다. x는 오른쪽으로 갈수록 증가하며, y는 위로 갈수록 증가한다.

MM_ANISOTROPIC이나 MM_ISOTROPIC 매핑 모드 중 하나를 사용하고 있다면, 여러분이 그림이 표시될 영역을 설정해 주기 위해 SetWindowExt()나 SetViewportExt() 함수를 사용할 수 있게 된다.

그래픽 애플리케이션을 만들자

배운 것은 바로 퀘어봐야 보배인 법이다. 이번 장에서 다른 부분을 포괄하는 애플리케이션을 만들어 보도록 하자. 이 애플리케이션은 두 개의 독립적인 윈도우를 가지고 있는데, 한쪽은 도형, 도구, 표시 색깔을 선택할 수 있는 윈도우이며, 또 한쪽은 캔버스로 사용하기 위한 것으로 선택된 옵션이 사용되어 그림이 그려진다. 사용자는 선, 사각형, 원, 또는 비트맵을 이 윈도우에 그릴 수 있다. 또한 표시될 색깔과 브러시/펜 여부도 동시에 선택하여 그릴 수 있다.



■ 애플리케이션 골격을 만들자

지금까지 배운 것을 토대로 생각해 볼 때, 애플리케이션을 만드는 첫번째는 초기 골격을 만드는 일이란 것쯤은 눈감고 떠올릴 수 있을 것이다. 이 골격에는 기본적인 애플리케이션 기능과 다이얼로그, 그리고 프로그램 시작과 종료 코드가 이미 구현되어 있다.

이번 장에서 만들 애플리케이션을 위한 골격 역시 표준 다이얼로그 스타일이다. Graphics이란 이름의 AppWizard 프로젝트를 새로 만든 다음, 1단계에서 dialog-based를 선택하자. 이후의 설정은 모두 디폴트로 해주고, 윈도우 타이틀은 여러분이 원하는 대로 바꿔준다.

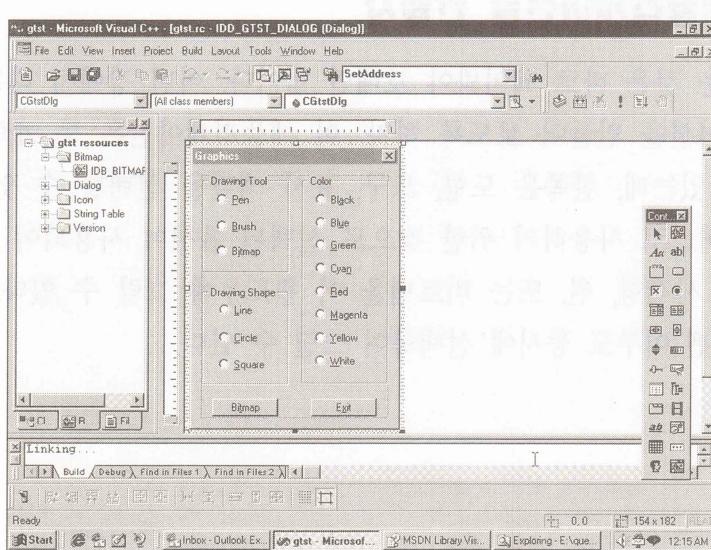
메인 다이얼로그를 꾸미자

애플리케이션 위저드 설정을 모두 마쳤으면 이제 다이얼로그 꾸미기에 들어가자. 메인 다이얼로그에는 세 개의 라디오 버튼 그룹이 들어있는데, 첫번째 그룹은 그리기 도구를 선택하는 기능을, 두번째 그룹은 그리고자 하는 도형을, 세번째 그룹은 색깔을 설정하기 위한 것이다. 이들과 함께 두 개의 버튼이 달려 있는데, 하나는 파일 열기 다이얼로그를 띄워서 표시할 비트맵을 선택하기 위한 것이고, 또 하나는 애플리케이션을 종료하기 위한 것이다.

[그림 8.3]을 참조하여 여러분의 다이얼로그에 이들 컨트롤을 추가하고, [표 8.3]을 참조해서 각 컨트롤의 프로퍼티를 설정해 주자.

그림 8.3 →

메인 다이얼로그
레이아웃



(표 8.3) 컨트롤 프로퍼티 설정

컨트롤	프로퍼티	설정
그룹박스	ID	IDC_STATIC
	Caption	Drawing Tool
라디오 버튼	ID	IDC_RTPEN
	Caption	&Pen
	Group	체크
라디오 버튼	ID	IDC_RTBRUSH
	Caption	&Brush
라디오 버튼	ID	IDC_RTBITMAP
	Caption	B&itmap
그룹 박스	ID	IDC_STATIC
	Caption	Drawing Shape
라디오 버튼	ID	IDC_RSLINE
	Caption	&Line
	Group	체크
라디오 버튼	ID	IDC_RSCIRCLE
	Caption	&Circle
라디오 버튼	ID	IDC_RSSQUARE
	Caption	&Square
그룹 박스	ID	IDC_STATIC
	Caption	Color
라디오 버튼	ID	IDC_RCBLACK
	Caption	Bl&ack
	Group	체크
라디오 버튼	ID	IDC_RCBLUE
	Caption	Bl&ue
라디오 버튼	ID	IDC_RCGREEN
	Caption	&Green

컨트롤	프로퍼티	설정
라디오 버튼	ID	IDC_RCCYAN
	Caption	Cya&n
라디오 버튼	ID	IDC_RCRED
	Caption	&Red
라디오 버튼	ID	IDC_RCMAGENTA
	Caption	&Magenta
라디오 버튼	ID	IDC_RCYELLOW
	Caption	&Yellow
라디오 버튼	ID	IDC_RCWHITE
	Caption	&White
푸시 버튼	ID	IDC_BBITMAP
	Caption	Bi&tmap
푸시 버튼	ID	IDC_BEXIT
	Caption	E&xit

메인 다이얼로그 꾸미기에 있어서 라디오 버튼의 각 그룹에 변수를 물려두는 일에 주의하자. 클래스 위저드를 열고, Group 옵션이 체크된 라디오 버튼에만 변수를 물리도록 해야 한다. Group 옵션이 체크된 라디오 버튼 이후의 컨트롤의 ID를 순서대로 가지는 나머지 라디오 버튼들이 같은 변수에 연관되기 때문에, 되도록 한 그룹에 넣고 싶은 라디오 버튼은 한번에 순서대로 모두 만드는 것이 덜 피곤하다. 자, 그러면 [표 8.4]를 참고해서 애플리케이션 내의 라디오 버튼 그룹에 필요한 변수를 물려 두도록 하자.

(표 8.4) 컨트롤 변수

컨트롤	이름	카테고리	타입
IDC RTPEN	m_iTool	Value	int
IDC_RSLINE	m_iShape	Value	int
IDC_RCBLACK	m_iColor	Value	int

클래스위저드를 닫기 전에 첫번째 탭으로 가서, Exit 버튼의 이벤트 핸들러를 추가한 다음 이 함수에 OnOK() 함수를 호출하는 문장을 추가하도록 하자. 애플리케이션을 컴파일하고 실행한 다음, 여러분이 확인할 것은 라디오 버튼이 제대로 작동되는지이다. 한 그룹 내에서는 단 한 개의 버튼만 선택되어야 한다.

다이얼로그를 하나 더 만들자

P114 Note

그럼이 그려지는 캔버스로 사용할 두번째 다이얼로그 윈도우는 모듈리스 다이얼로그로 띄워진다. 즉, 애플리케이션이 실행될 동안 계속 열려 있는 채로 사용할 수 있다. 캔버스로 사용할 것이기 때문에 아무 컨트롤도 놓아서는 안되겠다.

일단 프로젝트 워크스페이스에서 리소스뷰를 선택하고, Dialogs 폴더에서 오른쪽 클릭한 다음 팝업 메뉴에서 Insert Dialog를 선택한다. 새 다이얼로그 템플릿이 에디터 영역에 나타나면 컨트롤을 모두 지워버린다. 컨트롤을 모두 지워버린 다음, 이 윈도우에 대한 프로퍼티 다이얼로그 박스를 띄워서(오른쪽 클릭한 다음, Properties를 선택한다), 두번째 탭에서 System Menu 옵션의 체크를 해제한다. 아울러서 이 다이얼로그 윈도우의 ID를 새로 지어주어도 된다. IDD_PAINT_DLG 식으로 말이다.

이젠 클래스위저드를 통해 이 다이얼로그 윈도우를 담당하는 새 클래스를 만들어야 한다. 클래스위저드를 열려고 하면 새 클래스를 만들겠냐는 메시지가 담긴 다이얼로그 박스가 나타나는데, 디폴트 설정으로 해놓고 OK 버튼을 클릭하자. 이 다음에 나타나는 다이얼로그 박스에서는 클래스 이름과 기본 클래스를 설정해 주어야 하는데, 클래스 이름으로는 CPaintDlg를 입력하고, 기본 클래스로는 CDialog를 선택하자. OK를 클릭하여 새 클래스 만들기를 마친 후에는 바로 클래스위저드를 닫는다.

Note

클래스위저드를 열 때, 새 다이얼로그가 선택되었는지 꼭 확인하기 바란다. 만일 다이얼로그가 선택되지 않았으면 다른 개체가 클래스위저드에서 선택되기 때문이다. 클래스위저드는 여러분이 두번째 다이얼로그에 대한 클래스가 필요한지, 어떤지 알지 못한다.

두번째 다이얼로그가 정의된 상태이기 때문에, 이제는 첫번째 다이얼로그에서 두번째 다이얼로그를 모듈리스로 띄우는 코드를 추가해야 한다. 첫번째 다이얼로그 윈도우 클래스의 OnInitDialog() 함수에 두 줄의 코드만 넣으면 끝이니까 큰 걱정은 말자. 우선, CDialog 클래스의 Create() 함수를 사용해서 다이얼로그를 생성한다. 이 함수

OnInitDialog()는 두 개의 인수, 즉 다이얼로그 템플릿 리소스의 ID와 부모 윈도우(메인 다이얼로그 윈도우)의 포인터이다. 다음, 윈도우를 보여야 하기 때문에 ShowWindow() 함수를 SW_SHOW를 인수로 주어 호출한다. 방금 필자가 말한 내용을 [리스트 8.1]에 정리하였다.

리스트 8.1 OnInitDialog() 함수

```

1:  BOOL CGraphicsDlg::OnInitDialog()
2:  {
3:      CDIALOG::OnInitDialog();
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
27:
28: // TODO: Add extra initialization here
29:
30: /////////////////
31: // 새로 넣을 코드가 여기서부터 시작된다
32: /////////////////
33:
34: // 변수를 초기화하고 다이얼로그 윈도우를 생성한다
35: m_iColor = 0;
36: m_iShape = 0;
37: m_iTool = 0;
38: UpdateData(FALSE);
39:
40: // 두번째 다이얼로그 윈도우를 생성한다
41: m_dlgPaint.Create(IDD_PAINT_DLG, this);
42: // Show the second dialog window
43: m_dlgPaint.ShowWindow(SW_SHOW);
44:
45: /////////////////
46: // 새로 넣을 코드는 여기서 끝난다
47: /////////////////
48:
49: return TRUE; // return TRUE unless you set the focus to a control
50: }
```

✓ 여러분의 애플리케이션을 컴파일하고 실행하기 전에, 두번째 다이얼로그 윈도우 클래스의 헤더를 첫번째 다이얼로그에 관한 소스 코드 파일에서 인클루드하는 일과 두 번째 다이얼로그 클래스 변수를 첫번째 다이얼로그 클래스의 멤버로 넣어 두는 일을 빼먹어선 안된다. 후자는 CPaintDlg 타입의 m_dlgPaint란 이름을 가진 변수를 CGraphicsDlg 안에다가 선언하면 되는 것이고, 전자는 첫번째 다이얼로그의 소스 코드의 윗부분으로 가서 [리스트 8.2]와 같이 인클루드 문장을 추가하면 되는 것이다.

리스트 8.2 메인 다이얼로그 소스 파일의 인클루드 문장

```

1: // GraphicsDlg.cpp : implementation file
2: //
3:
4: #include "stdafx.h"
5: #include "Graphics.h"
6: #include "PaintDlg.h"
7: #include "GraphicsDlg.h"
8:

```

반대로, 두번째 다이얼로그의 소스 코드에 메인 다이얼로그의 헤더 파일도 인클루드 해야 한다. PaintDlg.cpp를 열고 [리스트 8.2]와 일치하도록 인클루드 문장을 추가하도록 하자.

이제 애플리케이션을 컴파일하고 실행시키면, 메인 다이얼로그 윈도우와 함께 열리는 두번째 다이얼로그 윈도우를 보게 될 것이다. 또한 주목해서 보아야 할 것은 첫 번째 다이얼로그를 닫았을 때 두번째 다이얼로그도 따라서 닫힌다는 사실이다. 절대로 여기에 관련된 코딩을 해준적이 없는데 말이다. 두번째 다이얼로그는 메인 다이얼로그의 자식 윈도우이다. 리스트의 41째 줄에서 이 다이얼로그를 생성할 때, 메인 다이얼로그의 포인터를 인수로 넘겼기 때문에 이 두 윈도우 사이에 부모 - 자식 관계가 성립된 것이다. 부모 - 자식 관계가 있는 두 윈도우 사이에서는 부모 윈도우가 사라지면 자식 윈도우도 따라서 사라져야 한다. 메인 다이얼로그에 있는 컨트롤과 메인 다이얼로그 사이에도 동일한 부모 - 자식 관계가 성립한다. 각 컨트롤은 다이얼로그의 자식 윈도우이다. 어떤 의미에서는 두번째 다이얼로그 박스 역시 첫번째 다이얼로그에 놓인 컨트롤이 되는 셈이다.

■ 그래픽 기능을 추가하자

모든 라디오 버튼 변수가 public으로 선언되어 있기 때문에, 두번째 다이얼로그는 필요할 때마다 자유롭게 이 변수를 액세스할 수 있다. 그림을 그리는 기능을 모두 두 번째 다이얼로그 클래스에 넣어둘 수도 있겠지만, 이 중에서 몇 가지는 첫번째 다이얼로그 클래스에 넣어 변수의 값이 일치되도록 유지하고 두번째 다이얼로그에 그림을 그리라는 명령을 내리는 것이 합당하다. 말은 좀 어렵지만 같이 하다보면 무슨 뜻인지 이해할 수 있을 것이다.

A 윈도우 자체가 다시 그려져야 할 때마다(다른 윈도우에 의해 가려져 있다가 나타났다든지, 최소화되어 있거나 화면 바깥에 있다가 보이게 되었다든지) 윈도우 운영체제는 해당 윈도우(여기서는 다이얼로그)의 OnPaint() 함수를 호출한다. 따라서, 이 함수 안에 그래픽에 관련된 모든 처리 코드를 해두면 여러분이 표시하는 그림을 항상 일관되게 나타낼 수 있을 것이다.

어디에 그리기 코드를 두어야 하는지도 감을 잡았는데, 만약 사용자가 첫번째 다이얼로그에서 선택 사항을 바꿀 때마다 두번째 다이얼로그의 OnPaint() 함수를 호출하려면 어떻게 해야 할까? 뭐, 두번째 다이얼로그를 다른 윈도우로 가렸다가 보이게 하면 되겠지만 이게 어디 정상적인 프로그램이 사용자에게 할 짓인가? 전체 윈도우가 다시 그려지도록 하는 함수가 엄연히 존재한다. Invalidate()란 함수로서, 아무런 인수도 받지 않는다, 또한 CWnd 클래스의 멤버이기 때문에 어떤 윈도우나 컨트롤에서도 사용할 수 있다. Invalidate() 함수는 윈도우와 운영체제에게 “윈도우의 디스플레이 영역이 더 이상 유효하지 않아서 다시 그려야 하오”라고 알려주는 역할을 한다. 별 이상한 방법이 필요없이 두번째 다이얼로그의 OnPaint() 함수를 호출해도 된다.

모든 라디오 버튼은 클릭된 이벤트에서 똑같은 동작을 수행한다는 사실도 주목할만하다. 모든 라디오 버튼 컨트롤의 클릭 이벤트에 대한 이벤트 핸들러를 딱 하나 만들고, 이 함수 안에서는 UpdateData() 함수를 호출해서 다이얼로그 컨트롤의 현재 값을 컨트롤에 물려둔 변수로 옮긴 다음, 두번째 다이얼로그의 Invalidate() 함수를 호출해서 다시 그리라는 명령을 내리면 된다. 그 하나의 이벤트 핸들러인 [리스트 8.3]은 방금 말한 두 가지를 축실히 수행한다.

리스트 8.3 OnRSelection() 함수

```

1: void CGraphicsDlg::OnRSelection()
2: {
3:     // TODO: Add your control notification handler code here
4:
5:     // 컨트롤에서 데이터를 읽어온다
6:     UpdateData(TRUE);
7:     // 두번째 다이얼로그를 다시 그린다
8:     m_dlgPaint.Invalidate();
9: }
```

선을 그리자

방금 컴파일하고 실행시킨 프로그램을 보면, 메인 디아얼로그 윈도우의 라디오 버튼을 선택할 때마다 두번째 디아얼로그 윈도우가 다시 그려(진다고했으니까)지지만, 실제로 여러분의 총명한 눈에는 아무 변화도 일어나지 않는다. 윈도우를 다시 그리게는 하지만, 무엇을 그릴 것인지 정해주지 않았기 때문이란 것이 자연스럽게 머리에 떠오르지 않는가? 즉, 바로 이 과정이 여기서 거쳐갈 과정이다.

두번째 윈도우에 그림을 그리는 가장 간단한 것이 여러 가지 선을 그는 것일 것이다 (이미 한번 해봤으니까). 각기 다른 펜 스타일에 대해 현재 선택된 하나의 펜을 생성하는 것이 급선무이고, 루프를 통해 생성된 펜을 모두 순환하면서 차례로 디아얼로그를 가로지르는 직선을 그으면 될 것 같다. 루프에 들어가기 전에, 디아얼로그 내의 어느 위치부터 어느 위치까지 선을 그릴 것인지를 계산해 주는 과정이 필요하다.

여러분은 일단 메인 디아얼로그에서 선택할 수 있는 색깔 그룹에 속해 있는 각각의 색깔을 엔트리로 가지고 있는 색깔 테이블을 추가해야 한다. 색깔 테이블을 만들려면 두번째 디아얼로그 클래스인 CPaintDlg에 새 멤버 변수를 추가하는데, 이 변수의 타입은 static const COLORREF이고, 이름은 m_crColors[8]이며, 액세스 지정자는 public이다. 다음, 두번째 디아얼로그 클래스의 소스 파일을 열고 클래스 생성자와 소멸자 앞에 [리스트 8.4]와 같은 색깔 테이블을 추가하도록 하자.

리스트 8.4 색깔 테이블

```

1: const COLORREF CPaintDlg::m_crColors[8] = {
2:     RGB( 0, 0, 0), // Black
3:     RGB( 0, 0, 255), // Blue
4:     RGB( 0, 255, 0), // Green
5:     RGB( 0, 255, 255), // Cyan
6:     RGB( 255, 0, 0), // Red
7:     RGB( 255, 0, 255), // Magenta
8:     RGB( 255, 255, 0), // Yellow
9:     RGB( 255, 255, 255) // White
10: };
11: ///////////////////////////////////////////////////////////////////
12: // CPaintDlg dialog

```

① CGraphs Dlg ② CPaint Dlg

* 순서

1. 색깔 테이블 추가
 - i) 색 지정 엘리먼트를 ①에 추가. static const COLORREF m_crColor[8]
 - ii) 테이블을 ②에 변수로 추가.
2. 선그리는 함수 추가(③에) void DrawLine(CPaintDC& pdc, int iColor, private

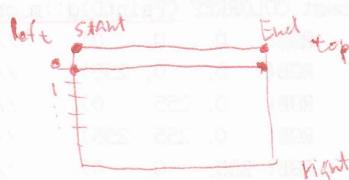
색깔 테이블도 장착해 넣었으니, 선을 긋는 새 함수를 넣을 차례이다. OnPaint() 함수에 그림을 그리는 코드를 이것저것 끼워 넣다 보면 굉장히 난잡해지고 이해가 어려워지는데, 이것을 막기 위해 제한된 양의 코드만을 넣어서, 두번째 다이얼로그에 그려질 것을 결정하고 그것을 가리는 별도의 함수를 호출하도록 하는 것이 좋다. 그래서 CPaintDlg 클래스에 새 멤버 함수를 넣도록 하는데, 반환 타입은 void, 선언 형식은 DrawLine(CPaintDC *pdc, int iColor)로 정해주며, 액세스 지정자는 private로 해주자. [리스트 8.5]는 바로 이 함수의 본체이다.

리스트 8.5 DrawLine() 함수

```

1: void CPaintDlg::DrawLine(CPaintDC *pdc, int iColor)
2: {
3:     // 펜을 선언하고 생성한다
4:     CPen lSolidPen (PS_SOLID, 1, m_crColors[iColor]);
5:     CPen lDotPen (PS_DOT, 1, m_crColors[iColor]);
6:     CPen lDashPen (PS_DASH, 1, m_crColors[iColor]);
7:     CPen lDashDotPen (PS_DASHDOT, 1, m_crColors[iColor]);
8:     CPen lDashDotDotPen (PS_DASHDOTDOT, 1, m_crColors[iColor]);
9:     CPen lNullPen (PS_NULL, 1, m_crColors[iColor]);
10:    CPen lInsidePen (PS_INSIDEFRAME, 1, m_crColors[iColor]);
11:
12:    // 그릴 영역을 얻어낸다
13:    CRect lRect;
14:    GetClientRect(lRect);
15:    lRect.NormalizeRect();
16:
17:    // 선들 사이의 거리를 계산한다
18:    CPoint pStart;
19:    CPoint pEnd;
20:    int liDist = lRect.Height() / 8;
21:    CPen *lOldPen;
22:    // 시작점을 설정한다
23:    pStart.y = lRect.top;
24:    pStart.x = lRect.left;
25:    pEnd.y = pStart.y;
26:    pEnd.x = lRect.right;
27:    int i;
28:    // 펜들을 모두 순환한다
29:    for (i = 0; i < 7; i++)
30:    {
31:        // 어떤 펜인가?
32:        switch (i)
33:        {
34:            case 0: // 실선
35:                lOldPen = pdc->SelectObject(&lSolidPen);
36:                break;
37:            case 1: // 점선

```



```

38:     pdc->SelectObject(&lDotPen);
39:     break;
40: case 2: // 쇄선
41:     pdc->SelectObject(&lDashPen);
42:     break;
43: case 3: // 점쇄선
44:     pdc->SelectObject(&lDashDotPen);
45:     break;
46: case 4: // 이점 쇄선
47:     pdc->SelectObject(&lDashDotDotPen);
48:     break;
49: case 5: // 투명선
50:     pdc->SelectObject(&lNullPen);
51:     break;
52: case 6: // 내부
53:     pdc->SelectObject(&lInsidePen);
54:     break;
55: }
56: // 다음 아래 위치로 이동한다
57: pStart.y = pStart.y + liDist;
58: pEnd.y = pStart.y;
59: // 선을 그린다
60: pdc->MoveTo(pStart);
61: pdc->LineTo(pEnd);
62: }
63: // 원래 펜으로 되돌려 선택한다
64: pdc->SelectObject(10ldPen);
65: }

```

이제 OnPaint() 함수에는 필요할 때마다 DrawLine() 함수가 호출되도록 하는 코드를 넣어주면 된다. 클래스위저드를 사용해서 WM_PAINT 메시지에 대한 이벤트 핸들러로 OnPaint()를 추가하면, 자동으로 생성된 코드에 CPaintDC 변수가 미리 만들어져 있음을 알 수 있다. CPaintDC 클래스는 CDC 디바이스 컨텍스트 클래스의 파생 클래스이다. 이 클래스는 WM_PAINT 메시지를 처리할 때 거의 관습적으로 해주어야 하는 BeginPaint()와 EndPaint() 함수 호출을 자동으로 도맡는다. 따라서 프로그래머 입장에서는 보통의 디바이스 컨텍스트 개체처럼 사용하면 그만이다. 어차피 그리기 함수도 CDC 것을 모두 쓰니까.

OnPaint() 함수로 잘 찾아 왔으면, 우선 부모 윈도우(메인 디아일로그 윈도우)의 포인터를 얻어내어 라디오 버튼에 물려진 변수의 값을 점검한 다음 색깔, 도구, 도형을 결정할 수 있도록 해야 한다. 이렇게 결정된 정보는 DrawLine() 함수를 호출할 것인가, 아니면 아직 작성하지 않은 다른 함수를 호출할 것이냐를 구분하는데 사용된다.

두번째 다이얼로그 클래스의 WM_PAINT 메시지에 대한 이벤트 핸들러를 추가한 상태에서 [리스트 8.6]에 나온 코드를 입력하도록 하자.

리스트 8.6 OnPaint() 함수

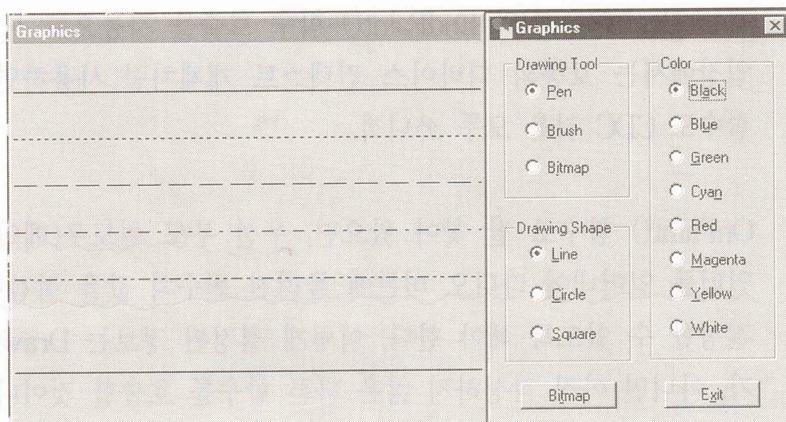
```

1: void CPaintDlg::OnPaint()
2: {
3:     CPaintDC dc(this); // device context for painting
4:
5:     // TODO: Add your message handler code here
6:
7:     // 부모 윈도우의 포인터를 얻어낸다
8:     CGraphicsDlg *pWnd = (CGraphicsDlg*)GetParent();
9:     // 포인터가 제대로 되어 있나요?
10:    if (pWnd)
11:    {
12:        // 선택된 도구가 비트맵입니까?
13:        if (pWnd->m_iTool == 2)
14:        {
15:        }
16:        else // 아닙니다. 도형을 그리는겁니다.
17:        {
18:            // 선을 그리는 건가요?
19:            if (pWnd->m_iShape == 0)
20:                DrawLine(&dc, pWnd->m_iColor);
21:        }
22:    }
23:    // CDialog::OnPaint()를 호출하지 않도록 하자
24:}
```

이제 애플리케이션을 컴파일하고 실행시키면, [그림 8.4]와 같은 여러 가지 선이 그려지는 다이얼로그 윈도우를 볼 수 있을 것이다.

그림 8.4 →

두번째 다이얼로그에서 선이 그려진다.



원과 사각형을 그리자

도구를 선택하고 그림을 그리는 기본적인 구조는 이제 다 갖추었고, 두번째 다이얼로그에 그려지는 내용을 마음먹은 대로 변경할 수 있는 방법도 공부하였기 때문에, 이젠 원과 사각형도 그려보도록 하자. 원을 그리는데 사용하는 함수는 `Ellipse()`이고 사각형을 그리는데 사용하는 함수는 `Rectangle()`이란 함수이며, 둘다 디바이스 컨텍스트 클래스의 멤버이다. 두 함수는 공통적으로 해당 도형에 딱 맞는 사각형 정보를 가진 `CRect` 객체를 인수로 받는다. `Rectangle()` 함수는 전체 사각 영역을 채우며, `Ellipse()` 함수는 사각 영역에 접하는 타원 혹은 원을 그린다. 또한 펜과 브러쉬를 동시에 사용하기 때문에, 사용자가 펜이나 브러쉬 중 하나를 선택할 수 있도록 하기 위해서 보이지 않는 펜과 보이지 않는 브러쉬를 생성해서 선택해 둘 필요가 있다. 펜의 경우 널 펜(null pen)을 사용하면 되지만, 브러쉬의 경우에는 솔리드 브러쉬를 윈도우 배경색으로 설정해서 생성해야 한다.

각 도형의 위치를 계산할 때는 조금 전에 선을 그릴 때 취해주었던 것과 다른 접근 방식을 취해야 한다. 선을 그릴 때는 다이얼로그 윈도우의 높이를 얻어낸 다음 8로 나누고, 그 높이에서 다이얼로그 윈도우의 왼쪽 경계에서 오른쪽 경계까지 선을 그었지만, 원과 사각형의 경우에는 다이얼로그 윈도우의 공간을 여덟 개의 균등한 사각 영역으로 나누어야 한다. 이렇게 하는 가장 쉬운 방법은 한 줄에 네 개의 사각 영역이 차지하고 있는 두 줄을 만드는 것이다. 또한, 각 사각 영역 사이에는 약간의 간격을 두어서 외곽선을 그릴 때 사용되는 여러 가지 펜을 확인할 수 있도록 하자.

자, 이제 시작해 보자. 두번째 다이얼로그 클래스에 새 함수를 추가하는데, 반환 타입은 `void`로, 선언 형식은 `DrawRegion(CPaintDC *pdc, int iColor, int iTool, int iShape)`로, 액세스 지정자는 `private`로 해준다. [리스트 8.7]은 이 함수의 전부이다.

리스트 8.7 `DrawRegion()` 함수

```

1: void CPaintDlg::DrawRegion(CPaintDC *pdc, int iColor, int iTool, int Shape)
2: {
3:     // 펜을 선언하고 생성한다
4:     CPen lSolidPen (PS_SOLID, 1, m_crColors[iColor]);
5:     CPen lDotPen (PS_DOT, 1, m_crColors[iColor]);
6:     CPen lDashPen (PS_DASH, 1, m_crColors[iColor]);
7:     CPen lDashDotPen (PS_DASHDOT, 1, m_crColors[iColor]);
8:     CPen lDashDotDotPen (PS_DASHDOTDOT, 1, m_crColors[iColor]);
9:     CPen lNullPen (PS_NULL, 1, m_crColors[iColor]);

```

```

10: CPen lInsidePen (PS_INSIDEFRAME, 1, m_crColors[iColor]);
11:
12: // 브러쉬를 선언하고 생성한다
13: CBrush lSolidBrush(m_crColors[iColor]);
14: CBrush lBDiagBrush(HS_BDIAGONAL, m_crColors[iColor]);
15: CBrush lCrossBrush(HS_CROSS, m_crColors[iColor]);
16: CBrush lDiagCrossBrush(HS_DIAGCROSS, m_crColors[iColor]);
17: CBrush lFDiagBrush(HS_FDIAGONAL, m_crColors[iColor]);
18: CBrush lHorizBrush(HS_HORIZONTAL, m_crColors[iColor]);
19: CBrush lVertBrush(HS_VERTICAL, m_crColors[iColor]);
20: CBrush lNullBrush(RGB(192, 192, 192));
21:
22: // 그릴 영역의 크기를 계산한다
23: CRect lRect;
24: GetClientRect(lRect);
25: lRect.NormalizeRect();
26: int liVert = lRect.Height() / 2;
27: int liHeight = liVert - 10;
28: int liHorz = lRect.Width() / 4;
29: int liWidth = liHorz - 10;
30: CRect lDrawRect;
31: CPen *lOldPen;
32: CBrush *lOldBrush;
33: int i;
34: // 모든 펜과 브러쉬에 대해 루프를 돈다
35: for (i = 0; i < 7; i++)
36: {
37:     switch (i)
38:     {
39:         case 0: // 실선
40:             // 이 도형의 위치를 결정한다
41:             // 첫번째 줄을 시작한다
42:             lDrawRect.top = lRect.top + 5;
43:             lDrawRect.left = lRect.left + 5;
44:             lDrawRect.bottom = lDrawRect.top + liHeight;
45:             lDrawRect.right = lDrawRect.left + liWidth;
46:             // 적당한 펜과 브러쉬를 선택한다
47:             lOldPen = pdc->SelectObject(&lSolidPen);
48:             lOldBrush = pdc->SelectObject(&lSolidBrush);
49:             break;
50:         case 1: // 점선, 역대각 빛금 브러쉬
51:             // 이 도형의 위치를 결정한다
52:             lDrawRect.left = lDrawRect.left + liHorz;
53:             lDrawRect.right = lDrawRect.left + liWidth;
54:             // 적당한 펜과 브러쉬를 선택한다
55:             pdc->SelectObject(&lDotPen);
56:             pdc->SelectObject(&lBDiagBrush);
57:             break;
58:         case 2: // 쇄선, 바둑판 무늬 브러쉬
59:             // 이 도형의 위치를 결정한다
60:             lDrawRect.left = lDrawRect.left + liHorz;

```

```

61:         lDrawRect.right = lDrawRect.left + liWidth;
62:         // 적당한 펜과 브러쉬를 선택한다
63:         pdc->SelectObject(&lDashPen);
64:         pdc->SelectObject(&lCrossBrush);
65:         break;
66:     case 3:   // 점 쇄선 -X자 무늬 브러쉬
67:         // 이 도형의 위치를 결정한다
68:         lDrawRect.left = lDrawRect.left + liHorz;
69:         lDrawRect.right = lDrawRect.left + liWidth;
70:         // 적당한 펜과 브러쉬를 선택한다
71:         pdc->SelectObject(&lDashDotPen);
72:         pdc->SelectObject(&lDiagCrossBrush);
73:         break;
74:     case 4:   // 이점 쇄선, 대각 빗금 브러쉬
75:         // 이 도형의 위치를 결정한다
76:         // 두번째 줄을 시작한다
77:         lDrawRect.top = lDrawRect.top + liVert;
78:         lDrawRect.left = lRect.left + 5;
79:         lDrawRect.bottom = lDrawRect.top + liHeight;
80:         lDrawRect.right = lDrawRect.left + liWidth;
81:         // 적당한 펜과 브러쉬를 선택한다
82:         pdc->SelectObject(&lDashDotDotPen);
83:         pdc->SelectObject(&lFDiagBrush);
84:         break;
85:     case 5:   // 널 펜, 수평무늬 브러쉬
86:         // 이 도형의 위치를 결정한다
87:         lDrawRect.left = lDrawRect.left + liHorz;
88:         lDrawRect.right = lDrawRect.left + liWidth;
89:         // 적당한 펜과 브러쉬를 선택한다
90:         pdc->SelectObject(&lNullPen);
91:         pdc->SelectObject(&lHorizBrush);
92:         break;
93:     case 6:   // 내부, 수직 무늬 브러쉬
94:         // 이 도형의 위치를 결정한다
95:         lDrawRect.left = lDrawRect.left + liHorz;
96:         lDrawRect.right = lDrawRect.left + liWidth;
97:         // 적당한 펜과 브러쉬를 선택한다
98:         pdc->SelectObject(&lInsidePen);
99:         pdc->SelectObject(&lVertBrush);
100:        break;
101:    }
102:    // 어떤 도구를 사용하고 있나요?
103:    if (iTool == 0)
104:        pdc->SelectObject(lNullBrush); Pen Brush, Bitmap
105:    else
106:        pdc->SelectObject(lNullPen);
107:    // 어떤 도형을 그릴건가요?
108:    if (iShape == 1)
109:        pdc->Ellipse(lDrawRect);
110:    else
111:        pdc->Rectangle(lDrawRect);

```

```

112: }
113: // 원래 브러쉬와 펜으로 되돌린다는 블록을 뒤집어
114: pdc->SelectObject(101dBrush);
115: pdc->SelectObject(101dPen);
116: }
```

두번째 다이얼로그에 원과 사각형을 그릴 수 있게 되었으니까, 이젠 사용자가 펜 또는 브러쉬가 선택된 상태에서 원(Circle)과 사각형(Square) 중 하나를 선택했을 때 우리가 만든 이 함수를 호출하면 된다. [리스트 8.8]과 같이 OnPaint() 함수의 21째 줄부터 두 줄을 추가하도록 하자.

리스트 8.8 고쳐진 OnPaint() 함수

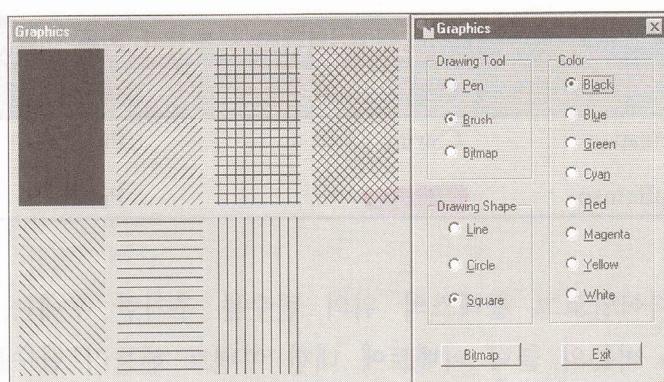
```

1: void CPaintDlg::OnPaint()
2: {
3:     CPaintDC dc(this); // device context for painting
4:
5:     // TODO: Add your message handler code here
6:
7:     // 부모 윈도우의 포인터를 얻어낸다
8:     CGraphicsDlg *pWnd = (CGraphicsDlg*)GetParent();
9:     // 포인터가 제대로 되어 있나요?
10:    if (pWnd)
11:    {
12:        // 선택된 도구가 비트맵입니까?
13:        if (pWnd->m_iTool == 2)
14:        {
15:        }
16:        else // 아니오. 도형을 그릴겁니다
17:        {
18:            // 선을 그릴건가요?
19:            if (m_iShape == 0)
20:                DrawLine(&dc, pWnd->m_iColor);
21:            else // 원과 사각형을 그릴겁니다
22:                DrawRegion(&dc, pWnd->m_iColor, pWnd->m_iTool, pWnd->m_iShape);
23:        }
24:    }
25:    // CDialog::OnPaint()는 호출하지 않는다
26:}
```

애플리케이션을 컴파일하고 실행시키면, 이젠 직선뿐만 아니라 원과 사각형까지 두 번째 다이얼로그 윈도우에서 볼 수 있을 것이며, 외곽선이 그려진 도형과 외곽선 없이 내부가 칠해진 도형도 만들어 볼 수 있을 것이다([그림 8.5] 참조).

그림 8.5 →

두번째 다이얼로그에서
사각형이 그려진다.

**비트맵을 로드하자**

이제 비트맵을 로드해서 표시하는 일만 남았다. 쉽게 하려면 애플리케이션을 만드는 도중에 비트맵을 리소스에 추가하고, ID를 붙인 다음, LoadBitmap()과 MAKEINTRESOURCE() 함수를 사용해서 CBitmap 클래스 개체에 비트맵을 로드하면 되겠지만, 그리 유용한 방법이 아니다. 진정한 유용성은 파일에서 비트맵 파일을 읽어서 표시할 수 있는 기능에서 나온다고 본다. 여기에 사용되는 함수는 LoadImage()란 Win32 API이며, 이 함수를 사용해서 비트맵 이미지를 메모리에 로드하고, CBitmap 개체에 이미지 핸들을 부착하는 순서를 밟을 것이다.

우선, 메인 다이얼로그의 Bitmap 푸시 버튼에 함수를 추가하여 파일 열기 다이얼로그 박스를 사용자에게 보여 사용자가 비트맵을 선택할 수 있도록 해야 하는데, 비트맵 파일만이 파일 열기 다이얼로그 박스에 표시될 수 있도록 필터링 문자열을 만들어 두어야 한다. 사용자가 제대로 된 비트맵을 선택하면, 여러분은 파일 열기 다이얼로그 박스로부터 파일 이름과 경로 이름을 얻어내고, 이어서 LoadImage() 함수를 써서 이미지를 로드한다. 메모리에 로드된 비트맵에 대한 유효한 핸들을 얻어냈으면, CBitmap 개체에서 현재의 비트맵을 삭제해야 한다. 만일 CBitmap 개체에 로드된 비트맵이 있으면, 방금 삭제한 비트맵과 CBitmap 개체를 떼어(detach) 놓도록 하자. 아무튼 중요한 것은 CBitmap 개체에 로드된 이미지가 없도록 해두어야 한다는 것이다. 이 상태가 되면 Attach() 함수를 사용해서 새 비트맵 이미지를 부착한다. 이 시점에서 두번째 다이얼로그 윈도우를 무효화하여(비트맵이 표시되도록 설정된 상태라면) 새로 로드된 비트맵이 나타나도록 하는 것이다.

이 기능을 넣기 위해서 일단 비트맵 이름을 담기 위한 문자열 변수와 비트맵 이미지를 담기 위한 CBitmap 변수를 메인 다이얼로그 클래스의 멤버로 추가해야 한다. [표 8.5]를 보도록 하자.

CGraphicsDlg.h
변수 추가

(표 8.5) 비트맵을 처리하기 위한 변수

이름	타입	액세스 지정자
m_sBitmap	CString	Public
m_bmpBitmap	CBitmap	Public

메인 다이얼로그 클래스의 위의 변수를 추가한 후에는, 클래스워저드를 사용해서 Bitmap 버튼의 클릭 이벤트에 대한 이벤트 핸들러 함수를 추가하고 [리스트 8.9]에 나온 코드를 입력한다.

리스트 8.9 OnBbitmap() 함수

```

1: void CGraphicsDlg::OnBbitmap()
2: {
3:     // TODO: Add your control notification handler code here
4:
5:     // 파일 열기 다이얼로그에 대한 필터링 문자열을 만든다
6:     static char BASED_CODE szFilter[] = "Bitmap Files (*.bmp)|*.bmp|";
7:     // 파일 열기 다이얼로그를 생성한다
8:     CFileDialog m_ldFile(TRUE, ".bmp", m_sBitmap,
9:                         OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, szFilter);
10:
11:    // 파일 열기 다이얼로그를 보이고 결과값을 얻는다
12:    if (m_ldFile.DoModal() == IDOK)
13:    {
14:        // 선택된 파일 이름을 얻는다
15:        m_sBitmap = m_ldFile.GetPathName();
16:        // 선택된 비트맵 파일을 로드한다
17:        HBITMAP hBitmap = (HBITMAP)::LoadImage(AfxGetInstanceHandle(),
18:                                              m_sBitmap, IMAGE_BITMAP, 0, 0,
19:                                              LR_LOADFROMFILE | LR_CREATEDIBSECTION);
20:
21:        // 로드된 이미지에 대한 핸들이 유효한가요?
22:        if (hBitmap)
23:        {
24:            // 현재 비트맵을 삭제한다
25:            if (m_bmpBitmap.DeleteObject())
26:                // 비트맵이 있으면 떼어낸다
27:                m_bmpBitmap.Detach();
28:            // 현재 로드된 비트맵을 CBitmap 개체에 부착한다
29:            m_bmpBitmap.Attach(hBitmap);
30:        }
31:        // 두번째 다이얼로그 윈도우를 무효화한다
32:        m_dlgPaint.Invalidate();
33:    }
34: }
```

비트맵을 표시하자

이제 비트맵은 메모리에 로드된 상태이고, 사용자에게 보여줄 순서만 남았다. GetBitmap() 함수를 사용해서 CBitmap 개체로부터 BITMAP 개체로 비트맵 정보를 복사할 필요가 있는데, 이후에 BITMAP에서 비트맵 이미지의 높이과 폭을 알아내기 위해서이다. 다음, 화면 디바이스 컨텍스트와 호환되는 새 디바이스 컨텍스트를 생성하고, 이 디바이스 컨텍스트에 비트맵을 선택해 넣은 다음, StretchBlt() 함수를 사용해서 호환 디바이스 컨텍스트에서 원래의 디바이스 컨텍스트로 비트맵을 복사한다 (필요하면 복사할 때 크기도 조정한다).

두번째 다이얼로그 클래스에 새 멤버 함수를 추가하자. 함수 타입은 void로, 선언 형식은 ShowBitmap(CPaintDC *pdc, CWnd *pWnd)로 입력하며, 액세스 지정자는 private로 지정한다. 끝났으면 [리스트 8.10]을 입력하자.

Note

메인 다이얼로그의 타입이 아닌 CWnd 개체의 포인터로 인수가 넘겨진 사실에 주목하자. 첫번째 다이얼로그의 클래스 타입 포인터로 선언하려면 첫번째 다이얼로그에 대한 클래스를 두번째 다이얼로그의 클래스 선언문 앞에 선언해 두어야 한다. 동시에, 첫번째 다이얼로그는 두번째 다이얼로그 클래스가 먼저 선언되어 있어야 함을 요구하고 있기 때문에 각각의 소스 코드 파일의 앞에 헤더를 인클루드해야 한다는 결론이 나온다. 하지만, 어느 클래스이든지 동시에 바로 앞에 선언된 클래스를 가진 채로 있을 수는 없는 일이다. 이 문제를 해결하기 위한 방법, 즉 두번째 클래스에 대한 자리 채움자를 첫번째 클래스의 선언문 앞에 두는 방법이 있긴 하지만, 그냥 포인터를 첫번째 다이얼로그 클래스에 대한 포인터로 캐스팅하는 것이 여기서는 훨씬 편하다. 두번째 클래스에 대한 자리 채움자를 두는 방법은 부록 A, “C++도 다시 보자”에서 찾을 수 있을 것이다.

리스트 8.10 ShowBitmap() 함수

```

1: void CPaintDlg::ShowBitmap(CPaintDC *pdc, CWnd *pWnd)
2: {
3:     // 포인터를 메인 다이얼로그 클래스의 포인터로 캐스팅한다
4:     CGraphicsDlg *lpWnd = (CGraphicsDlg*)pWnd;
5:     BITMAP bm;
6:     // 로드된 비트맵의 정보를 얻어낸다
7:     lpWnd->m_bmpBitmap.GetBitmap(&bm);
8:     CDC dcMem;
9:     // 비트맵을 로드할 호환 디바이스 컨텍스트를 생성한다
10:    dcMem.CreateCompatibleDC(pdc);
11:    // 호환 디바이스 컨텍스트에 비트맵을 선택해 넣는다
12:    CBitmap* pOldBitmap = (CBitmap*)dcMem.SelectObject (lpWnd->m_bmpBitmap);
13:    CRect lRect;
```

```
14: // 디스플레이 영역을 얻어낸다
15: GetClientRect(lRect);
16: lRect.NormalizeRect();
17: // 다이얼로그 윈도우로 비트맵을 복사하면서 크기를 조정한다
18: pdc->StretchBlt(10, 10, (lRect.Width() - 20),
19:                 (lRect.Height() - 20), &dcMem, 0, 0,
20:                 bm.bmWidth, bm.bmHeight, SRCCOPY);
21: }
```

현재 선택된 비트맵을 다이얼로그 윈도우로 보내어 표시할 수 있는 준비는 다 끝났으니까 이 함수를 두번째 다이얼로그 클래스의 OnPaint() 함수에서 호출하면 된다. 첫번째 다이얼로그의 m_sBitmap 변수의 값을 체크해서 비트맵이 설정되었는가를 알아본다. 만일 이 문자열이 비어 있으면 표시될 비트맵이 없는 것이고, 비어 있지 않을 때 ShowBitmap() 함수를 호출하면 된다. OnPaint() 함수를 고쳐서, [리스트 8.11]의 15째 줄부터 18째 줄 사이를 입력하도록 하자.

리스트 8.11 고쳐진 OnPaint() 함수

```
1: void CPaintDlg::OnPaint()
2: {
3:     CPaintDC dc(this); // device context for painting
4:
5:     // TODO: Add your message handler code here
6:
7:     // 부모 윈도우의 포인터를 얻어낸다
8:     CGraphicsDlg *pWnd = (CGraphicsDlg*)GetParent();
9:     // 포인터가 제대로 되어 있나요?
10:    if (pWnd)
11:    {
12:        // 선택된 도구가 비트맵입니까?
13:        if (pWnd->m_iTool == 2)
14:        {
15:            // 비트맵이 선택되어 있고 로드되어 있나요?
16:            if (pWnd->m_sBitmap != "")
17:                // 표시하세요.
18:                ShowBitmap(&dc, pWnd);
19:        }
20:        else // 아니오. 도형을 그릴겁니다
21:        {
22:            // 선을 그릴건가요?
23:            if (m_iShape == 0)
24:                DrawLine(&dc, pWnd->m_iColor);
25:            else // 원이나 사각형을 그립니다.
26:                DrawRegion(&dc, pWnd->m_iColor, pWnd->m_iTool,
27:                           pWnd->m_iShape);
28:        }
29:    }
30: }
```

```

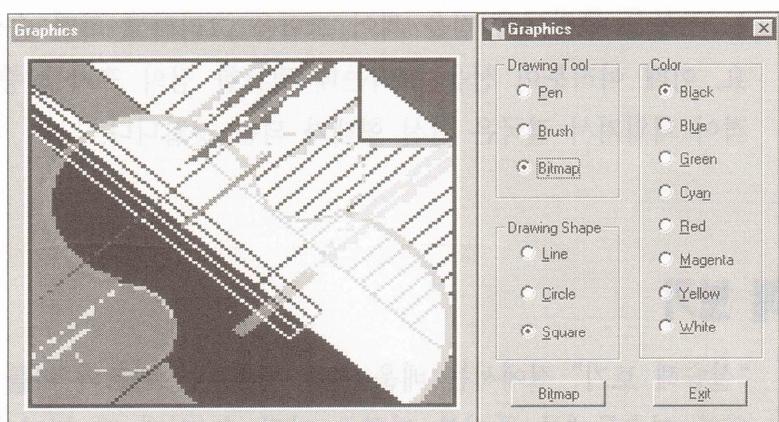
29: }
30: // CDlg::OnPaint()는 호출하지 않는다
31: }

```

애플리케이션을 컴파일하고 실행하면, 시스템에서 비트맵을 로드하여 다이얼로그 윈도우에 출력할 수 있을 것이다. [그림 8.6]은 한 예이다.

그림 8.6

두 번째 다이얼로그에 비트맵을 표시하고 있다.



요약

한 주의 첫 날을 장식하는 장이 이렇게 길 줄이야! 이번 장에서는 정말 많은 것을 배웠다. 우선 윈도우 운영체제가 컴퓨터에 장착된 하드웨어에 관계없이 일관성 있는 그래픽 기능을 구사하기 위해 사용하는 디바이스 컨텍스트(Device Context)라는 것에 대해 알아보았고, 기본적인 GDI 개체인 펜과 브러쉬, 그리고 이것을 사용해서 그림을 그리는 방법도 같이 실습하면서 확인하였다. 디스크에서 비트맵을 로드하여 화면에 표시하는 방법은 꽤 복잡하지만 매우 유용한 학습이었다. 아울러서 여러 가지의 펜과 브러쉬 스타일을 사용하는 방법과 여기에 색깔까지 설정하는 방법도 공부하여, 그래픽에 꼭 필요한 것은 거의 다 익힌 셈이다.

Q&A

- ① 왜 반드시 펜과 브러쉬를 동시에 설정해야 하는 것입니까?
- ② 항상 어떤 도형을 그릴 때 내부가 채워지도록 하기 때문입니다. 펜은 외곽선을 그리고, 브러쉬는 안을 채우죠. 둘 중에 하나만 선택해서 사용할 수는 없습니다. 둘 다 사

용해야 하지요. 둘 중에 하나만 사용해서 그런 효과를 내고 싶으면 특수한 단계를 거쳐야 합니다.

② 펜 두께를 1보다 두껍게 하기만 하면, 왜 모든 펜 스타일이 실선으로 되는거죠?

① 펜의 두께를 증가시키면, 선을 그리는데 사용되는 도트(dot)의 크기를 증가시키는 셈이 됩니다. 3장, “마우스와 키보드 입력의 처리”에서 보셨겠지만 마우스가 돌아다니는 부분을 추적해서 점을 찍어 그림을 그리려고 하면 순식간에 점박이 그림이 되지요. 이때 여러분이 선을 그리는데 필요한 점의 크기를 증가시키면 그 점 사이의 간격이 채워져서 결국은 실선 형태가 되는 것입니다.

실습해 보기

“실습해 보기” 절에서는 배운 것을 확인하는 퀴즈와 이를 활용해서 응용력을 높이기 위한 연습문제를 풀어볼 기회를 가질 수 있게 될 것이다. 퀴즈와 연습문제의 답은 부록 B, “퀴즈 및 연습문제 해답”에 있고, 정답 안보기 실천하여 정의 사회 구현하자.

퀴즈

- 색을 설정하는데 필요한 세 가지 값은 각각 무엇인가?
- 어떤 그래픽 카드를 사용하는지 신경 쓰지 않고도 편하게 그림을 그릴 수 있는 것은 무엇을 사용하기 때문인가?
- 어떤 크기의 비트맵을 브러쉬로 사용할 수 있는가?
- 윈도우가 다시 그려져야 한다는 것을 알리기 위해 전송되는 이벤트 메시지는?
- 윈도우를 일부러 다시 그리게 하려면 어떻게 할까?

연습문제

- 크기를 조정할 수 있는 다이얼로그 윈도우를 하나 더 만들고, 윈도우의 크기에 그림의 크기도 따라서 맞추어지도록 해보자.
- 사각형과 타원을 그리기 위한 브러쉬 집합에 비트맵 브러쉬를 추가하자.