그래픽, 멀티미디어 어플리케이션 제작

(Creating Graphic and Multimedia Application)

델파이를 이용해서 어플리케이션에 그래픽 기능을 추가하거나, 멀티 미디어를 지원하게 하는 것은 그다지 어려운 일이 아니다. 간단하게 미리 그려진 그림을 디자인 시에 추가할 수도 있고, 여러가지 그래픽 컨트롤을 추가하거나 런타임에서 동적으로 그리는 등의 방법이 모두 가능하다.
또한, 델파이 4 에서 멀티미디어 컴포넌트를 사용하면 어플리케이션에서 동영상이나 각종 사운드 등을 지원하게 할 수 있다.
이번 장에서는 각종 어플리케이션을 제작할 때 화려함을 더해줄 수 있는 그래픽과 멀티 미디어를 지원하게 하는 방법에 대해서 알아보도록 한다.

그래픽 프로그래밍 개괄

VCL 그래픽 컴포넌트는 윈도우의 GDI(Graphics Device Interface)를 캡슐화한다. 델파이 어플리케이션에서 그림을 그리려면, 객체의 캔버스를 이용한다. 캔버스는 객체의 프로퍼티로 제공되지만, 그 자체가 객체이다. 캔버스 객체의 가장 큰 장점은 리소스를 효과적으로 사용하며, 디바이스 컨텍스트를 관리하기 때문에 스크린, 프린터나 비트맵, 메타 파일 등의 종류에 관계 없이 같은 방법으로 사용할 수 있다는 것이다.
캔버스는 런타임에만 사용할 수 있다. 또한, TCanvas 객체가 윈도우의 디바이스 컨텍스트의 wrapper 이므로, 캔버스에 대해 윈도우 GDI 함수를 사용할 때에는 캔버스 객체의 Handle 프로퍼티를 이용해서 디바이스 컨텍스트 핸들을 얻어야 한다.

- 디바이스 컨텍스트의 이해

윈도우의 그래픽에 대해 배울 때 가장 먼저 알아야 할 것이 디바이스 컨텍스트이다. 모든 윈도우 응용 프로그램들이 실제 장치 대신에 가상 화면과 가상 프린터를 사용한다. 윈도우는 본질적으로 실제 디스플레이 하기 전에, 내부적으로 그릴 것을 그리게 되는데 이때 사용하는 것이 디바이스 컨텍스트이다.
따라서 화면이나 프린터에 그리기를 원하는 어떤 것과 디바이스 컨텍스트를 분리할 필요가 있다. 디바이스 컨텍스트는 단순한 메모리라기 보다는 객체이다. 이 등록정보를 변경하면 다양한 특수 효과를 줄 수도 있다.
윈도우는 디바이스 컨텍스트를 가지고 직접 작업하는 것을 허용하지 않는다. 이를 처리하기 위해서는 API 함수를 호출해야 한다. 이때 컨텍스트 핸들에 대한 변수로 hDC 를 사용

한다.
델파이는 미리 정의된 디바이스 컨텍스트의 wrapper 를 제공한다. 이것이 바로 캔버스이다.
윈도우의 디바이스 컨텍스트의 유형에는 다음과 같은 4 가지 종류가 있다.

1. 디스플레이 (Display)

보통 윈도우나 TMemo 컨트롤과 같이 그릴 수 있는 다른 영역에 부착되어 있다. 이 컨텍스트에 그리면 항상 정보가 화면에 전달된다. 디스플레이 디바이스 컨텍스트에는 Class, Common, Private 형의 3 가지 종류가 있다. 32 비트 응용 프로그램에서는 항상 private 형을 사용하면 된다. 만약 빨리 그래프나 차트를 출력하고자 할 때에는 common 형을 사용할 수 있다. Common 디바이스 컨텍스트를 얻을 때에는 GetDC 나 GetDCEx, BeginPaint 함수 등을 사용하면 된다. 물론 이때 받은 핸들은 빨리 사용하고 반납해야 한다. Private 디바이스 컨텍스트는 실제로 응용 프로그램이 소유하고 있는 컨텍스트이다. 이것의 장점은 윈도우에 돌려주기 전에 마음대로 그릴 수 있다는 점이다. 캐드나 그래픽 응용 프로그램의 경우에는 이 유형의 디스플레이 디바이스 컨텍스트를 사용한다.

2. 프린터 (Printer)

디스플레이 디바이스 컨텍스트와 같이 프린터 디바이스 컨텍스트에 보내는 것도 당장 인쇄된다.

3. 메모리 (Memory)

디스플레이나 프린터 디바이스 컨텍스트에 보낼 수 있는 것은 메모리에도 보낼 수 있다. 메모리 디바이스 컨텍스트는 CreateCompatibleDC 함수를 사용하여 만든다. 인쇄나 지금 디스플레이에 나타낼 때, 또는 나중에 출력할 때 메모리 디바이스 컨텍스트가 사용되고 있으며, 비트맵과 같은 것에도 이 디바이스 컨텍스트 유형을 사용할 수 있다.

4. 정보 (Information)

이 디바이스 컨텍스트는 디스플레이의 경우에는 그다지 문제가 되지 않지만, 프린터의 경우에는 중요한 역할을 한다. 예를 들어 프린터에 인쇄할 컬러 문서가 있는 경우, 프린터가 컬러 출력을 지원하는지 알면 도움이 될 것이다. 만약 흑백 프린터의 경우에는 특별한 디더링 루틴을 추가하는 것이 필요하다. 정보 디바이스 컨텍스트는 CreateIC 함수를 이용하여 만든다. 디바이스 컨텍스트를 만들었으면, GetCurrentObject 나 GetObject 함수를 이용하여 특정 객체에 대한 정보를 얻을 수 있다.

● 윈도우의 그래픽 객체

윈도우는 7 개의 서로 다른 그래픽 객체를 가지고 있다. 이들은 윈도우의 다른 객체 들을 만들 때 사용되는 기본 객체이다. 이들 객체에 대해서 알아보자.

1. 비트맵 (Bitmap)

비트맵은 특정 유형의 래스터 그림이다. 예를 들어 아이콘과 PCX 파일은 물론 BMP 파일도 이 범주에 속한다. 비트맵 객체는 바이트 단위의 크기, 픽셀 단위의 차원, 컬러 포맷, 압축 스키마 등의 정보를 가진다. 비트맵 객체는 자신의 특정 유형에 기초한 다른 속성들도 다양하게 가지고 있다.

2. 브러쉬 (Brush)

브러쉬 색상을 적용할 때 사용된다. 벽에 칠을 할 때 붓을 사용하는 것과 같이 윈도우는 다각형이나 패스와 같은 다른 그래픽 객체의 내부를 칠할 때 브러쉬를 사용하고 있다. 특정 패턴으로 내부를 채우고자 할 때에는 비트맵을 브러쉬로 사용할 수도 있다.

3. 팔레트 (Palette)

윈도우는 풍부한 색상을 정의하여 사용하는 색상의 집합인 팔레트를 사용한다. 그런데, 일부 비디오 카드는 한번에 256 색상만 나타낼 수 있다. 이것이 하나 이상의 팔레트를 사용하는 이유로 서로 다른 256 색상이 필요할 때마다 서로 다른 팔레트를 사용하면 깨끗하고 견고한 색상을 화면에 나타낼 수 있다.

4. 폰트 (Font)

글자를 나타내는 폰트 역시 마찬가지로 그래픽 객체이다. 적절한 크기와 서체를 사용하여야 좋은 어플리케이션을 만들 수 있다.

5. 패스 (Path)

다각형이나 호, 선, 그리고 타원과 같은 그려진 객체의 특정 유형을 의미한다. 윈도우는 수많은 서로 다른 패스에 대한 함수를 가지고 있으며, 이들은 모두 서로 다른 속성들을 가지고 있다.

6. 펜 (Pen)

선을 그리거나, 도형을 그릴 때 사용하는 객체로 라인의 두께와 스타일과 같은 것들을 설정하게 된다.

7. 지역 (Region)

지역은 디바이스 컨텍스트에 있는 한 영역의 위치를 나타낸다. 때때로 캔버스 전체가 아니라 일정 영역에 대해서만 윈도우가 작업하도록 요청해야 할 때가 있다. 지역 객체는 이와 같은 일을 할 수 있게 해 주는 역할을 한다.

- 윈도우의 그래픽 모드

디바이스 컨텍스트를 사용할 때에, 디바이스 컨텍스트의 유형을 명시하면 그래픽 객체를 담을 컨테이너를 정의하는 셈이다. 그러면, 이 객체 들이 어떻게 작용할 것인지를 결정해야 하는데, 이를 결정하는 것이 그래픽 모드이다. 그래픽 모드는 윈도우에게 디바이스 컨텍스트의 컨테이너에 있는 객체들이 서로 어떻게 동작할 것인지를 지정한다.
이들이 동작하는 방법은 대개 특정 유형의 함수를 사용할 때 어떤 종류의 화면 효과를 얻을 것인가를 결정한다. 이는 브러쉬를 사용하여 특정 영역을 채울 때 영향이 크다.
윈도우의 그래픽 모드에는 다음의 5 가지가 있으며, 이들을 동시에 사용한다.

1. 배경 (Background)

윈도우가 배경색을 섞는 방법을 정의한다. 윈도우가 한 색상을 다른 색상을 대치해야 하는지, 또는 두 색상을 합해서 새로운 색상을 만들어야 하는지를 나타낸다. 텍스트와 비트맵 동작에 있어서 보통 이 모드를 사용하게 된다.

2. 드로잉 (Drawing)

이 모드는 전경색(foreground color)을 같이 섞는 방법을 윈도우에게 알려주는 역할을 한다. 이 모드는 보통 펜이나 브러쉬, 텍스트, 그리고 비트맵 동작을 사용할 때 사용된다.

3. 매핑 (Mapping)

매핑은 윈도우에게 크기 조정을 어떻게 해결할 것인지를 알려주는 역할을 한다. 예를 들어,

이론적으로 가능한 커다란 공간을 그래픽으로 할 때, 시스템에 있는 메모리 공간에 의해 이를 모두 나타내게 할 수 없을 것이다. 프린터가 디스플레이에 이를 어떻게 보여줄 것인가를 결정하는 역할을 하는 것이 매핑이다.

4. 다각형 채우기 (Polygon-Fill)

이 모드는 윈도우가 다각형과 다른 도형을 채우는 방법을 정의한다. 본질적으로 윈도우가 그릴 때 사용할 도형의 모양과 브러쉬의 크기를 결정하는 역할을 한다.

5. 스트레칭 (Stretching)

이미지를 압축하면, 이미지의 세세한 내용의 일부를 잃어버리게 된다. 이미지의 질을 가능한 많이 유지하려면 윈도우가 색상들을 섞는 방법을 알아야 한다.

- 캔버스의 공통적인 프로퍼티와 메소드

캔버스 객체에서 공통적으로 사용하는 프로퍼티를 나열하면 다음과 같다.

프로퍼티	설 명
Font	이미지에 텍스트를 기록할 때 사용할 폰트를 지정한다. TFont 객체를 설정한다.
Brush	그래픽 모양과 배경을 채울 색깔과 패턴을 결정한다. TBrush 객체를 설정한다.
Pen	캔버스가 그림을 그릴 때 사용할 라인의 펜 종류를 결정한다. TPen 객체를 설정한다.
PenPos	펜의 현재의 위치를 지정한다.
Pixels	현재의 ClipRect 내의 픽셀 영역의 색깔을 지정한다.

캔버스 객체에서 사용하는 메소드에는 다음과 같은 것들이 있다. 이들에 대해서 모두 자세하게 설명할 수는 없으므로, 간단히 설명한다. 이중에서 CopyRect, Ellipase, RectAngle, MoveTo, TextOut, TextWidth, TextHeight 는 이미 9 장에서 사용한 바 있다.

메소드	설 명
Arc	지정된 사각영역의 타원을 따라서 호를 그린다.
Chord	타원이 라인으로 절단된 닫힌 그림을 그린다.
CopyRect	다른 캔버스의 영역에서 이미지를 복사한다.
Draw	캔버스의 Graphic 파라미터로 지정된 그래픽 객체를 지정한 위치에 그린다.
Ellipse	지정된 사각영역을 따라서 타원을 그린다.
FillRect	지정된 사각영역을 현재의 브러쉬로 채운다.

FloodFill	캔버스 전체 영역을 현재의 브러쉬로 채운다.
FrameRect	캔버스의 브러쉬를 이용하여 사각형을 그린다.
LineTo	PenPos 의 위치에서 지정된 위치까지 라인을 그린다.
MoveTo	현재의 위치를 지정된 위치로 옮긴다.
Pie	타원에서 지정된 사각영역으로 경계된 부분을 파이 형태로 그린다.
Polygon	파라미터로 넘긴 점들을 연결하여 다각형을 만든다. 처음 점과 마지막 점이 연결된다.
PolyLine	현재의 펜으로 Points 에 넘어온 점들을 서로 연결한다.
Rectangle	좌측 상단점과 우측 하단점을 대각선으로 하는 사각형을 펜을 이용해서 그리고, 내부를 브러쉬로 채운다.
RoundRect	코너가 둥근 사각형을 그린다.
StretchDraw	캔버스에 그린 그래픽을 지정된 사각영역에 맞춘다. 경우에 따라서 확대되거나 상하좌우 비율이 변경된다.
TextHeight, TextWidth	각각 현재 폰트로 문자열을 쓸 때의 높이와 폭을 반환한다. 높이에는 줄 사이의 여백이 포함된다.
TextOut	문자열을 지정된 위치에 첫자부터 출력한다. 그리고, PenPos 프로퍼티는 문자열의 끝부분의 위치로 업데이트 된다.
TextRect	영역안에 문자열을 기록한다. 영역 바깥으로 나가는 문자열은 보이지 않게 된다.

그래픽 작업을 할 때에는 드로잉(drawing)과 페인팅(painting)이란 용어를 구별해서 사용해야 한다.
드로잉은 특정 그래픽 요소를 생성하는 것이다. 예를 들어, 라인이나 특정 형태를 코드를 이용해서 그리는 것이다. 보통 앞에서 설명한 캔버스의 드로잉 메소드를 호출해서 사용한다. 이미지가 저장되지 않기 때문에, 그 내용의 전부 또는 일부를 잃을 수 있으며, 출력도 이미지가 저장되지 않고, 어플리케이션은 어떻게 다시 그리는 지를 알지 못하므로 변할 수 있다.
그에 비해 페인팅은 객체 전체의 형태를 생성하는 것으로, 드로잉을 포함한다. 즉, 어떤 상황에서든 어플리케이션이 그 전체 화면을 다시 칠할 수 있도록 하는 것이다. 사실 사용자가 마우스 버튼을 누르거나 다른 어떤 동작을 취하면 그 위치와 다른 요소들을 저장해야 하고, 페인팅 메소드에서 이 정보를 실제로 해당 이미지를 페인팅하기 위해 사용한다. .

- 스크린의 리프레쉬

윈도우는 스크린 위의 객체의 형태가 변경되거나, 리프레쉬할 필요가 있을 때에는 WM_PAINT 메시지를 생성해낸다. 이 메시지는 VCL 에서 OnPaint 이벤트로 표현된다. 그러므로, VCL 객체가 Refersh 메소드를 호출하면 언제나 OnPaint 이벤트가 발생한다.

폼에서 Refresh 메소드를 사용하면 지정된 형태로 그래픽을 다시 그리게 된다. 그러므로, 예를 들어 폼의 OnResize 이벤트 핸들러에서 폼의 형태를 변경하는 코드를 집어 넣은 경우에는 Refresh 메소드를 호출해서 변경된 사항을 반영하도록 해야 한다.
일부 운영체제에서는 윈도우의 클라이언트 영역의 일부가 무효화(invalidated)된 경우 자동으로 그 부분을 그려주지만, 윈도우는 그렇지 않다. 윈도우 운영체제에서는 일단 스크린에 그려진 것은 영구적이다. 그러므로, 폼이나 컨트롤이 드래그 등의 작업에 의해 잠시 가려지는 경우 폼이나 컨트롤은 반드시 불명확해진 영역을 다시 페인트 해야 한다.
TImage 컨트롤을 사용한다면, TImage 내부에 있는 그래픽의 페인팅과 refreshing 은 VCL에 의해 자동으로 관리된다. 또한 TImage 에 드로잉을 한 경우에는 이것이 지속적인(persistent) 이미지이기 때문에, 포함된 이미지를 다시 그려줄 필요도 없다. 이와는 반대로 TPaintBox 의 캔버스는 스크린 디바이스에 직접 연결되어 있기 때문에, PaintBox 에 그려진 모든 것은 임시로 저장된다. 대부분의 경우 TPaintBox 와 거의 동일하다.
그러므로, TPaintBox 에 그리거나 페인팅을 한 경우에는 OnPaint 이벤트 핸들러에 클라이언트 영역이 무효화될 때마다 이를 다시 그려주는 코드를 추가할 필요가 있다.
그래픽 이미지가 어플리케이션에 나타나는 형태는 그리는 방법에 따라 틀리다. 만약 TBitmap 캔버스처럼 오프 스크린 이미지를 그리는 경우에는 컨트롤이 컨트롤의 캔버스에 비트맵을 복사하기 전에는 나타나지 않는다. 그러므로, 비트맵을 그리고 이것을 이미지 컨트롤에 대입하려면, 이미지는 컨트롤이 OnPaint 메시지를 처리할 기회가 있을 때에 보여줄 수 있다.
화면의 Refresh 와 연관된 메소드에는 Invalidate, Update, Refresh(Repaint)의 3 가지가 있다. 이들의 특징은 각각 다음과 같다.

1. Invalidate 메소드

이 메소드는 윈도우에게 폼의 전체 표면이 페인팅되어야 한다는 것을 알려준다. 그런데, Invalidate 는 페인팅 동작을 즉각 강제하지 않고 윈도우가 이 요구사항을 저장하고나서, 현재 프로시저가 완전히 수행되고 시스템에 다른 이벤트가 남아 있지 않을 경우에 여기에 반응하게 된다. 때때로 이 지연 시간 때문에, 페인팅 작업에 여러 변경이 일어난 후에야 비로소 폼이 페인팅되는 수가 있다. 그렇기 때문에, 느린 페인트 메소드가 여러 번 호출되서 수행속도를 더욱 느리게 하는 것을 막을 수 있다.

2. Update 메소드

윈도우에게 폼의 내용을 갱신할 것인지를 물어 보아서 그것을 곧장 페인팅하는 메소드이다. 이 메소드는 무효 영역(invalidated area)이 있을 경우에만 동작한다. 그러므로, 이 동작은 Invalidate 메소드가 막 호출되었을 때 일어나거나 아니면 사용자에 의한 동작의 결과로서

일어날 수 있다. 만약 무효 영역이 없다면 아무런 동작을 하지 않으므로, 보통 Invalidate 를 호출한 뒤에 바로 Update 메소드를 호출한다.

3. Refresh(Repaint) 메소드

Invalidate 와 Update 메소드를 차례로 호출한다. 그 결과로 이 메소드는 OnPaint 이벤트를 즉각 동작시킨다.

이 메소드 들은 잘 사용해야 한다. 컨트롤을 반복 페이팅을 요청해야 할 경우에는 Invalidate 를 호출하는 것이 좋다. 이는 윈도우가 화면을 갱신하는데 너무 많은 시간을 소비하면 이 호출들을 한데 모아서 한 번에 처리할 수도 있기 때문이다. 윈도우의 WM_PAINT 메시지는 낮은 우선 순위 메시지이기 때문에, 이것이 가능하고 더 효과적이다. 반면에, Refresh 를 여러 번 호출했을 경우에는 윈도우가 매번 다른 메시지만 처리할 수는 없으니 화면이 다시 그려져야 하고, 페인팅 작업이 느리기 때문에 어플리케이션 전체의 성능을 떨어뜨릴 수 있다. 그렇지만, 가능한 빠르게 화면을 다시 그려야 하는 때가 있는데 이런 경우에는 Refresh 를 호출하는 것이 좋다.

- 그래픽 객체의 종류

VCL 은 다음과 같은 그래픽 객체를 제공한다.

객 체	설 명
Picture	그래픽 이미지를 담을 수 있다. 만약에 추가적인 그래픽 파일 포맷을 추가하려면 RegisterFileFormat 메소드를 사용한다. 그래픽 파일을 보여준다.
Bitmap	이미지를 생성, 관리, 저장하는데 사용되는 강력한 그래픽 객체
Clipboard	어플리케이션에 cut, copy, paste 를 지원하게 될 텍스트나 그래픽에 대한 컨테이너를 나타낸다. 적절한 포맷 관리, 참조 계수 관리 등을 한다.
Icon	윈도우 아이콘 파일을 다룬다.
Metafile	이미지를 다룰 때 실제 픽셀로서 구성되는 것이 아니라, 이미지를 구성하기 위한 여러 작업들을 기록하는 것이다. 메타 파일은 비트맵에 비해 메모리도 적게 차지하고, 이미지 손상이 적은 장점이 있으나, 처리 속도가 느리다.

캔버스 객체의 프로퍼티 활용

캔버스 객체에서 제공하는 여러가지 프로퍼티를 설정하여 사용하는 방법에 대해서 알아보자. 펜을 이용하여 라인을 그리고, 브러쉬를 이용하여 내부를 채우며, 적절한 폰트를 골라서 텍

스트를 기록하는 등의 작업이 캔버스 객체의 프로퍼티를 활용하여 이루어진다.

- 펜의 활용

캔버스의 Pen 프로퍼티는 라인의 형태를 결정한다. 선을 긋는 작업을 궁극적으로 생각해 보면 두 개의 점 사이에 있는 픽셀의 그룹을 바꾸는 것이다. 펜에 대해서 연상할 때 가장 쉽게 생각할 수 있는 것은 포토샵이나 페이트샵 프로와 같은 그래픽 프로그램의 도구 상자에서 여러가지 모양의 붓을 선택해서 선이나 도형을 그릴 수 있다는 것이다. 펜을 선택한다는 것은 이런 그래픽 프로그램에서 붓을 선택하는 것과 같은 것이다.
펜 자체에는 변경해서 사용할 수 있는 프로퍼티가 4 가지가 있다. Color, Width, Style, Mode 가 그것인데 이들 프로퍼티의 값들이 펜의 형태를 결정하게 된다. 디폴트로 모든 펜은 검정색, 두께는 1 픽셀이고 solid 형을 가지며, 모드는 캔버스에 있는 어떤 것이든 덧 씌워그리는 copy 모드로 설정되어 있다.

1. 펜 색상의 변경

펜의 색상을 변경하는 것은 다른 여러가지 컴포넌트의 Color 프로퍼티를 런타임에서 변경하는 것과 똑같다. 펜의 색상은 그려지는 라인의 색상을 결정하게 된다.
사용자에게 펜에 대한 새로운 색상을 결정하게 하려면, 보통 컬러 그리드를 띄워서 선택하게 한다. 컬러 그리드는 foreground 색상과 background 색상을 결정할 수 있게 구성되어 있다. 이때 foreground 색상은 보통 펜의 색상을 나타내며, background 색상은 브러쉬의 색상을 나타낸다고 생각하면 된다.

2. 펜의 두께 변경

펜의 두께는 그려지는 라인의 두께를 픽셀 단위로 결정하는 것이다. 펜의 두께가 1 보다 크면, 펜의 Style 프로퍼티 값에 관계 없이 윈도우 95 는 언제나 solid 라인을 그리게 된다. 펜의 두께를 변경하려면 펜의 Width 프로퍼티에 정수값을 대입하면 된다.

3. 펜의 스타일 변경

펜의 Style 프로퍼티는 선의 종류를 일반적인 선(solid), 점선(dotted line), 대쉬(dashed line) 등이 있다. 모두 6 개의 스타일이 존재하는데 이들은 각각 psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear 이다. 앞에서도 설명 했지만, 펜의 두께가 1 픽셀을 넘을 경우 이 프로퍼티의 값은 무시된다.

4. 펜의 모드 변경

펜의 모드 프로퍼티는 펜의 색상과 캔버스의 색상을 결합하는 방법을 결정하는 것이다. 예를 들어, 펜은 언제나 검정색이게 할 수도 있고, 캔버스의 배경색의 보색으로 보이게 할 수도 있고, 펜 색상의 보색으로 보이게 할 수도 있다.

5. 펜의 위치 얻기

펜이 다음 라인을 그릴 때 시작점이 되는 위치를 펜의 위치(position)라고 한다. 캔버스는 펜의 위치를 PenPos 프로퍼티에 저장한다. 펜의 위치는 라인을 그릴 때에만 영향을 미친다. 그러므로, 다른 도형이나 텍스트를 그리는 경우에는 그리려는 위치를 좌표로 전달해야 한다. 펜의 위치를 설정할 때에는 MoveTo 메소드를 사용한다. 예를 들어, 다음의 코드는 펜의 위치를 캔버스의 좌측 상단으로 이동한다.

```
Canvas.MoveTo(0, 0);
```

LineTo 메소드로 라인을 그릴 때, 라인의 끝점으로 현재의 위치를 이동시킨다.

- 브러쉬의 활용

캔버스 컨트롤의 Brush 프로퍼티는 도형의 내부를 채우는 방법을 결정한다. 브러쉬 객체에는 Color, Style, Bitmap 의 3 개의 프로퍼티를 이용해서 여러가지 작업을 하게 된다.
이들 프로퍼티의 디폴트 값은 백색, solid 스타일, 패턴은 없는 것이다.

1. 브러쉬 색상의 변경

브러쉬의 Color 프로퍼티는 영역의 내부를 채울 색상을 결정한다. 보통 브러쉬는 배경색을 결정하는 것으로 이해하면 된다.

2. 브러쉬 스타일의 변경

브러쉬 스타일은 캔버스에 채우는 방법을 결정한다. 스타일에 따라 브러쉬 색상과 캔버스의 색상을 결합하는 방법을 결정한다. 미리 지정된 스타일에는 solid, clear 와 여러가지 라인과 해치 패턴 등을 결정할 수 있다.
Style 프로퍼티에는 bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross 등의 값들이 있다.

3. 브러쉬 비트맵 프로퍼티의 설정

브러쉬의 Bitmap 프로퍼티는 브러쉬가 여기에 지정된 비트맵을 패턴으로 사용하여 영역을 채우게 된다. 다음의 코드는 비트맵을 파일에서 로드하여, 이를 브러쉬의 패턴으로 사용하는 예이다.

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0, 0, 100, 100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;
```

참고: 브러쉬 비트맵

브러쉬의 비트맵은 8x8 픽셀의 비트맵 패턴이어야 한다. 브러쉬에 비트맵을 사용하기 위해서는 우선 비트맵을 생성하고, 이것을 지정하여 사용한 후, 모든 작업이 완료되었으면 이를 해제시켜 주어야 한다.

그래픽 객체 그리기

- 라인 그리기

캔버스에서 그릴 수 있는 라인은 일반적인 라인과 폴리 라인(polyline)이 있다. 라인은 두 개의 점을 이어주는 픽셀 들이다. 폴리 라인은 라인이 서로 연결된 것으로, 이들을 그릴 때에 캔버스는 펜을 이용하게 된다.

1. 라인 그리기

캔버스에서 라인을 그리려면, LineTo 메소드를 사용한다. LineTo 메소드는 현재의 위치에서 지정한 좌표까지 현재의 펜으로 라인을 그린다. 그리고, 현재 위치가 끝점으로 이동하게 된다. 예를 들어, 다음의 코드는 폼이 그려질 때 폼에 라인으로 'X'자를 그리게 된다.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    MoveTo(0, 0);
    LineTo(ClientWidth, ClientHeight);
    MoveTo(0, ClientHeight);
    LineTo(ClientWidth, 0);
  end;
end;
```

2. 폴리 라인 그리기

폴리 라인을 캔버스에 드리려면 PolyLine 메소드를 사용한다. PolyLine 메소드에서 사용하는 파라미터는 Point 의 배열이다. PolyLine 은 기능적으로는 MoveTo 와 LineTo 메소드를 계속해서 호출하는 것과 같지만 호출에 따른 오버헤드가 없기 때문에 수행 속도가 빠르다.

- 도형 그리기

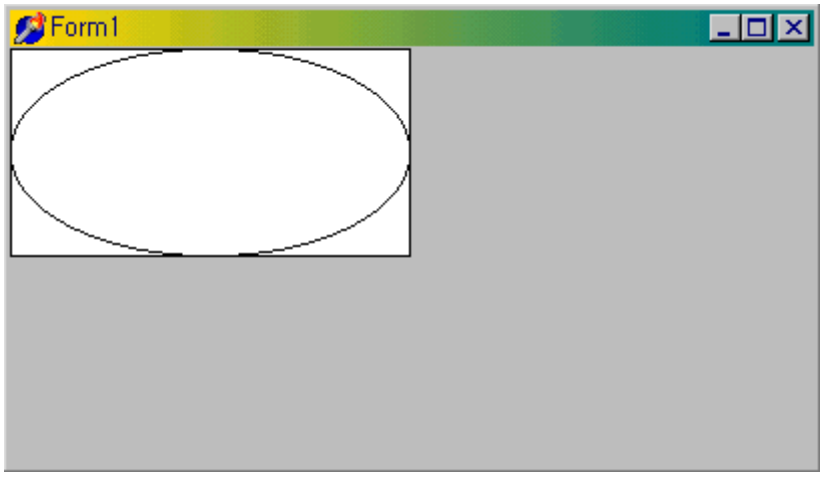
캔버스에는 여러가지 종류의 도형을 그리는 메소드가 있다. 캔버스는 도형의 외곽선은 펜으로 그리고, 내부는 브러쉬로 채운다.

1. 사각형과 타원 그리기

캔버스에 사각형과 타원을 그리기 위해서는 Rectangle, Ellipse 메소드를 사용하면 된다. 이 두 메소드 모두 좌표로는 경계가 되는 4 개의 좌표를 사용한다.
Rectangle 메소드는 넘어온 4 개의 좌표를 연결한 사각형을 그리며, Ellipse 메소드는 사각형내를 채우는 타원을 그린다.
다음의 코드는 좌측 상단의 1/4 영역에 사각형을 그리고, 내부에 타원을 그린다.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
  Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

이 코드를 실행하면 다음과 같은 실행 화면을 볼 수 있다.



2. 끝이 둥근 사각형 그리기

끝이 둥근 사각형을 그릴 때에는 RoundRect 메소드를 사용한다. RoundRect 메소드의 4 개의 파라미터는 다른 메소드와 마찬가지로 사각형의 좌표를 나타내며, 이 밖에도 둥근 코너를 어떻게 그릴지를 나타내는 2 개의 파라미터를 추가로 가진다.
다음의 메소드는 폼의 좌측 상단 1/4 영역에 10 픽셀의 지름의 원형 코너를 가진 끝이 둥근 사각형을 그리게 된다.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

3. 다각형 그리기

다각형을 그리기 위해서는 Polygon 메소드를 사용하면 된다. 다각형은 파라미터로 Point 의 배열을 사용하며, 이들의 점이 꼭지점이 된다.

그래픽 컨트롤 사용하기

어플리케이션에 그래픽 객체를 다루기 위해서 특별히 다른 컴포넌트를 사용할 필요는 없다. 그렇지만, 사실상 폼에 직접 드로잉을 하는 경우는 거의 없다. 그 보다는 보통 VCL 이미지 컨트롤을 이용해서 그래픽을 보여주는 경우가 많다.
이미지 컨트롤은 비트맵 객체를 디스플레이할 수 있는 컨테이너 컴포넌트이다. 이렇게 이미지 컨트롤을 사용하면 인쇄, 클립보드, 그래픽 객체 읽기와 저장 등의 편리한 기능을 이용할 수 있다. 그래픽 객체는 비트맵 파일, 메타 파일, 아이콘 등의 여러가지 그래픽 클래스일 수 있다.

- 스크롤 가능한 그래픽 만들기

그래픽의 크기는 폼과 같은 크기일 필요는 없다. 그래픽의 종류에 따라서는 폼보다 더 크거나 작을 수 있다. 이럴 때 스크롤 박스 컨트롤(TScrollBox)을 폼에 추가하고, 그래픽 이미지를 그 안에 위치시키면 그래픽이 폼보다 훨씬 커도(스크린 전체 보다 커도) 스크롤 박스의 기능을 통해 전체를 볼 수 있다.

- 초기 비트맵 크기 설정

이미지 컨트롤을 폼위에 올려 놓으면, 처음에는 단순한 컨테이너이다. 실제로 그래픽을 사용하려면 Picture 프로퍼티를 설정해 주어야 한다. 그런데, 처음에 비트맵을 보여주지 않더라도, 자리를 차지한 컨트롤이 보이지 않으면 보기가 무척 싫을 것이다. 이를 피하기 위해서는 폼의 OnCreate 이벤트 핸들러에 비트맵 객체를 생성해서 Picture.Graphic 프로퍼티에 대입하면 된다.
다음은 어플리케이션의 폼에 초기 비트맵의 크기를 설정해서 대입하는 코드이다.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  Bitmap.Width := Image1.Width;
  Bitmap.Height := Image1.Height;
  Image1.Picture.Graphic := Bitmap;
end;
```

비트맵을 Picture.Graphic 프로퍼티에 대입하면, Picture 객체가 비트맵의 Owner 가 된다. 그러므로, Picture 객체가 파괴되면, 비트맵 객체도 자동을 파괴된다. 어플리케이션을 실행하면 폼의 클라이언트 영역에 비트맵이 하얀 색으로 자리를 차지하고 있을 것이다.

그래픽 파일의 읽기와 쓰기

그래픽 이미지는 어플리케이션이 실행되는 동안에만 존재한다. 그렇기 때문에, 이를 파일로 저장하고, 불러오는 것은 필수적인 부분이다. 그래픽 이미지 컨트롤도 일반적인 다른 VCL 컴포넌트와 마찬가지로 이미지를 파일로 저장하고, 읽어올 수 있다. 또한, 이미지 컨트롤은 설치 가능한 그래픽 클래스를 지원한다.

- 파일에서 그림 읽어 오기

이미지 컨트롤에 그래픽을 로드하려면 Picture 객체의 LoadFromFile 메소드를 이용하면 된다. 9 장의 그래픽 인쇄 루틴에서 이미 한번 소개한 바 있으므로 이해가 어렵지는 않을 것이다. 다음의 코드는 OpenDialog 컴포넌트를 이용해서 그림 파일을 이미지 컨트롤에 불러오는 예이다.

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    CurrentFile := OpenDialog1.FileName;
   Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

- 그림을 파일로 저장하기

VCL 그림 객체는 몇 가지 포맷의 그래픽을 읽고, 쓸 수 있다. 그리고, 자신 만의 그래픽 파일 포맷을 생성하고, 등록시킬 수 있다. 이미지 컨트롤의 내용을 파일에 저장하려면 Picture 객체의 SaveToFile 메소드를 이용하면 된다.
다음의 이벤트 핸들러는 File|Save, File|Save As 메뉴를 선택했을 때 사용할 만한 코드이다. 참고로 하면, 쓸모가 있을 것이다. 이때 CurrentFile 변수는 문자열 형의 전역 변수로 선언해서 사용하면 된다.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile)
  else SaveAs1Click(Sender);
end;

procedure TForm1.SaveAs1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    CurrentFile := SaveDialog1.FileName;
    Save1Click(Sender);
  end;
end;
```

그래픽에 클립보드 이용하기

이미지 컨트롤을 이용해서 윈도우 클립보드에 cut, copy, paste 기능을 이용하려면 VCL 의 클립보드 객체를 사용하면 된다. VCL 클립보드 객체를 사용하려면 사용하려는 유닛의 uses 절에 Clipbrd.pas 유닛을 추가하여야 한다.

- 클립보드에 그래픽 복사, 잘라넣기 (copy and cut)

그림을 클립보드에 복사하려면, Picture 객체를 클립보드 객체로 Assign 메소드를 사용해서 대입하면 된다. 그래픽을 클립보드에 잘라 넣는 것도 복사하는 것과 비슷한 방법을 사용하지만, 소스에서 그래픽을 삭제하게 된다.
그래픽을 클립보드에 잘라 넣으려면, 먼저 클립보드로 복사를 하고 원래의 내용을 삭제하면 된다. 삭제한 이미지를 어떻게 보여줄 것인가 하는 것도 비교적 중요한 점이다. 흔히 사용하는 방법으로는 지워진 영역을 하얗게 표시하는 방법이다. 다음의 코드는 클립보드로 복사하기와 잘라내기를 메뉴에서 선택했을 때 쓸 수 있는 코드이다.

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image1.Picture);
```

```
end;

procedure TForm1.Cut1Click(Sender: TObject);
var
  ARect: TRect;
begin
  Copy1Click(Sender);                                  {클립보드로 일단 복사한다.}
  with Image1.Canvas do
  begin
    CopyMode := cmWhiteness;                           {복사 모드를 하얗게 설정}
    ARect := Rect(0, 0, Image1.Width, Image1.Height);  {비트맵 크기를 설정}
    CopyRect(ARect, Image1.Canvas, ARect);             {비트맵을 복사해 넣는다.}
    CopyMode := cmSrcCopy;                             {원래의 복사 모드로 복구}
  end;
end;
```

- 클립보드에서 그래픽 붙여 넣기 (Paste)

윈도우 클립보드에 비트맵 그래픽이 담겨 있으면, 이를 이미지 객체에 붙여넣을 수 있다. 이를 위해서는 먼저 클립보드의 HasFormat 메소드를 호출하여 클립보드가 그래픽을 가지고 있는지를 확인한다. HasFormat 메소드는 Boolean 값을 반환하는데, 지정한 데이터 형이 클립보드에 담겨 있으면 True 를 반환한다. 비트맵이 있는지 확인하려면 파라미터로 CF_BITMAP 를 넘겨주면 된다. 비트맵이 있으면 클립보드에서 대입하면 된다.
다음의 코드는 붙여 넣기를 구현한 예제 코드이다.

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
  if Clipboard.HasFormat(CF_BITMAP) then
  begin
    Image.Picture.Bitmap.Assign(Clipboard);
  end;
end;
```

어플리케이션에 드로잉 객체 이용하기

앞에서 설명한 다양한 드로잉 메소드는 툴바나 버튼 패널에서도 사용할 수 있다. 어플리케이션에 그래픽을 잘 사용하면 프로그램이 화려하며 고급스럽게 보이게 할 수 있지만, 너무 많은 색이나 너무 많은 그래픽을 사용할 경우에는 오히려 비생산적이 될 수도 있다.
그러므로, 적당한 수준에서 드로잉 객체를 잘 사용하는 것이 좋은 어플리케이션을 만드는데 상당히 중요한 역할을 하는 것이다.

DIBs(Device Independent Bitmaps)의 지원

델파이는 델파이 3 버전부터 TBitmap 객체에 비트맵 비트에 대한 포인터를 프로퍼티로제공하기 시작했다. 델파이 2 까지는 DDBs(Device Dependent Bitmaps)를 사용했으나, 델파이 3 부터 DIBs 를 지원하게 된 것이다. DIB 비트에 대한 포인터를 얻으려면 Bitmap.DibMemory 프로퍼티에 접근하기만 하면 된다.
다음의 코드는 첫번째 스캔 라인의 픽셀 들을 팔레트의 첫번째 색상을 변경한다. 이 코드는 256 색상의 비트맵 팔레트를 기준으로 작성되었기 때문에, 다른 형태의 비트맵에는 동작하지 않을 수 있다.

```
type
  TByteArray = array[0..0] of Byte;

procedure TForm1.Button1Click(Sender: TObject);
var
  p: ^TByteArray;
  i: Integer;
begin
  Image1.Picture.LoadFromFile(c:₩Windows₩구름.bmp');
  p := Image1.Picture.Bitmap.DibMemory;
  for i := 0 to (Image1.Picture.Bitmap.Width – 1) do
    p^[i] := 0;
end;
```

이 코드를 살펴 보면, 바이트 형의 배열을 선언하고 여기에 대한 포인터 변수 p 를 선언한다. 그리고, 비트맵을 디스크에서 읽어온 후, DibMemoey 에 대한 포인터를 변수 p 에 대입하고, DIB 메모리의 첫번째 스캔 라인의 색상을 DIB 팔레트의 첫번째 색상으로 바꾼다.

라인을 마음대로 그리자 !

지금까지 설명한 내용을 바탕으로 간단하게 라인을 그어주는 어플리케이션을 하나 만들어 보자. 제작할 어플리케이션은 사용자가 마우스를 움직이면 이를 따라 런타임에서 라인을 그리는 간단한 드로잉 프로그램이다. 이 어플리케이션은 마우스로 클릭하고 드래그를 하면 윈도우의 캔버스에 라인을 그린다. 마우스 버튼을 누르면 그리기가 시작되고, 버튼을 떼면 그리기가 완료된다.

- 마우스에 반응하기

어플리케이션에서 사용하는 마우스의 동작에는 버튼을 누르고(MouseDown), 마우스를 움직이고(MouseMove), 버튼을 떼는(MouseUp) 동작과 클릭(Click)이 있다.

1. 마우스 이벤트의 종류

VCL 에는 OnMouseDown, OnMouseMove, OnMouseUp 의 3 가지 마우스 이벤트가 있다. VCL 어플리케이션이 마우스의 동작을 감지하면 해당되는 이벤트 핸들러를 호출하게 되는데, 이들 이벤트는 모두 5 개의 파라미터를 사용한다. 이들 파라미터의 내용은 다음과 같다.

파라미터	의 미
Sender	마우스의 동작을 감지한 객체
Button	어떤 버튼의 동작인가 ? mbLeft, mbMiddle, mbRight 중의 하나이다.
Shift	Alt, Ctrl, Shift 키가 눌렸는가 ?
X, Y	이벤트가 발생한 좌표

2. MouseDown 이벤트의 처리

사용자가 마우스의 버튼을 누를 때마다 OnMouseDown 이벤트가 발생한다. 참고로 라인을 긋는 경우를 생각해보자. 이때 마우스 버튼을 누르면, 펜의 위치가 이동하도록 해야 한다. 그러므로 다음과 유사한 이벤트 핸들러를 작성하여야 할 것이다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(X, Y);
end;
```

3. MouseUp 이벤트의 처리

OnMouseUp 이벤트는 사용자가 마우스 버튼을 뗄 때 발생하게 된다. 이때 이 이벤트는 마우스를 누른 컨트롤에 발생하기 때문에, 커서의 위치가 컨트롤의 범위를 벗어나더라도 이를 처리할 수 있다. 그러므로, 라인을 폼의 범위를 벗어난 경우에도 처리를 할 수 있다.
어쨌든 앞에서의 MouseDown 이벤트에 이어서 라인을 긋도록 하려면, OnMouseUp 이벤트 핸들러를 다음과 같이 작성하면 된다.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
end;
```

이렇게 함으로써 사용자는 마우스 버튼을 클릭하고 드래그한 후, 이를 놓으면 라인을 그릴 수 있게 된다.

4. MouseMove 이벤트의 처리

OnMouseMove 이벤트는 사용자가 마우스를 움직일 때마다 주기적으로 발생한다. 다음의 예제 코드는 사용자가 마우스 버튼을 누른 채로 마우스를 움직일 때 라인을 그려주게 된다.

```
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
end;
```

- 마우스 동작의 기록

드로잉 어플리케이션을 작성하기 위해서는 마우스의 동작을 감시하는 것이 필요하다. 이를 위해서는 폼 객체에 여기에 대한 객체 필드를 추가하는 것이 좋다.
다음의 코드는 마우스 버튼의 동작을 기록하기 위해, 마우스 버튼이 눌려있는지 여부를 기록하는 Drawing 이라는 Boolean 필드를 추가하였다. 또한, 마우스의 위치를 기록하기 위해서 TPoint 형의 Orgin, MovePT 필드도 사용한다.

참고: TPoint 데이터 형

TPoint 데이터 형은 X 와 Y 의 값을 포인트(Point)라고 불리는 하나의 레코드에 저장한 것이다. 포인트를 생성하는 가장 쉬운 방법은 Point 함수를 사용하는 것이다. Point 함수는 X 와 Y 의 값을 받아들여 TPoint 라는 레코드를 반환하게 된다.

```
type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
 public
    Drawing: Boolean;                    //마우스 버튼이 눌렸는지 여부를 기록한다.
    Origin, MovePt: TPoint;              //포인트를 기록하는 필드
 end;
```

사용자가 마우스 버튼을 누르면 Drawing 필드를 True 로, 버튼을 떼면 False 로 설정한다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
  Drawing := False;
end;
```

OnMouseMove 이벤트 핸들러는 Drawing 프로퍼티가 True 일 때에만 동작하도록 수정한다.

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then                          //Drawing 플래그가 설정되었을 때만 그린다.
    Canvas.LineTo(X, Y);
end;
```

이렇게 하면, 마우스 버튼을 누르고 뗄 때까지 계속해서 라인을 이어서 그리게 된다.

● 마우스 포인트의 추적

라인을 그릴 때, Origin 필드에는 MouseDown 이벤트가 발생한 위치를 기록하고 이를 이용해서 MouseUp 이벤트에서 라인을 그려주면, 마우스를 눌렀다가 뗄 때 하나의 라인이 그려질 것이다. 다음과 같이 구현하면 된다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);
  Canvas.LineTo(X, Y);
  Drawing := False;
end;
```

이렇게 하면 라인을 제대로 그릴 수 있게 되지만, 마우스를 움직일 때의 라인을 볼 수가 없다. 이를 위해서는 마우스의 움직임을 점검해서 적절한 조치를 취할 필요가 있다.

- 마우스 움직임의 추적

현재의 OnMouseMove 이벤트 핸들러는 마지막 마우스의 위치를 따라서 라인을 그리기 때문에 문제가 있다. 이를 해결하기 위해서는 라인을 그리는 위치를 origin 포인트로 옮길 필요가 있다.

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y);            //펜을 시작 포인트로 이동
    Canvas.LineTo(X, Y);
  end;
end;
```

이렇게 하면, 현재의 마우스 위치를 따라서 라인을 그리게 되지만, 계속 라인을 겹쳐서 그리기 때문에 제대로 볼 수가 없다. 그러므로, 이를 시정하기 위해서는 다음 라인을 그리기 전에 원래의 라인을 지워야 한다. 이를 위해서는 이전에 사용한 점의 위치를 알아야 하는데, 이를 저장하기 위해 MovePt 필드를 사용한다.
MovePt 필드는 각각의 중간 라인의 끝점으로 설정되며, Origin 과 MovePt 를 연결하는 라인을 다음과 같이 그리면 이전의 라인을 지우게 된다.

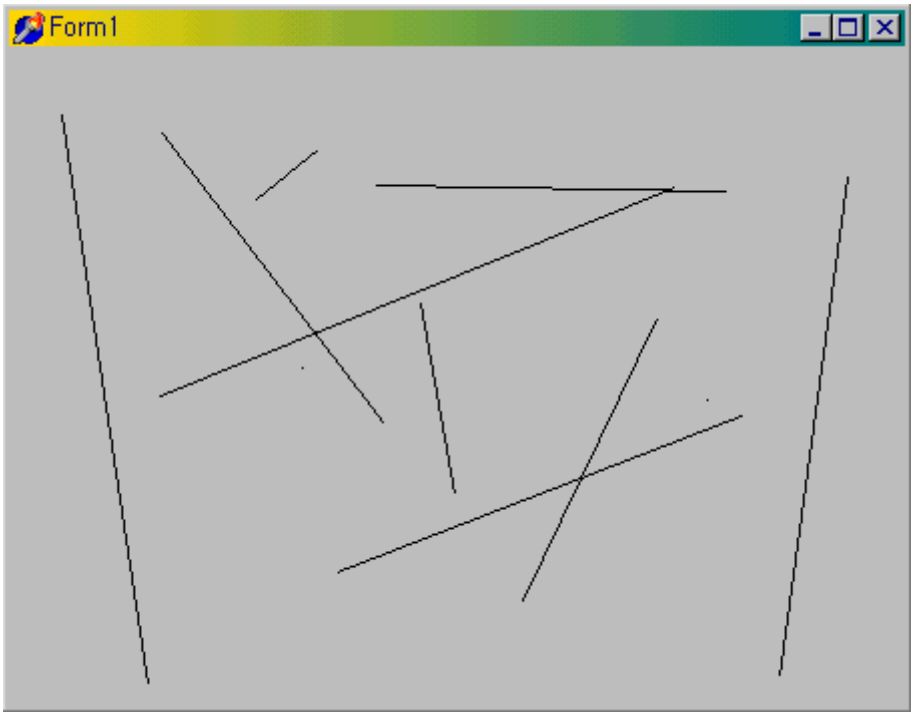
```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);
  MovePt := Point(X, Y);
end;

procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
```

```
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;              //그리고, 지우기 위해서 XOR 모드를 사용
    Canvas.MoveTo(Origin.X, Origin.Y);        //펜의 위치를 origin 으로
    Canvas.LineTo(MovePt.X, MovePt.Y);        //이전 라인을 지운다.
    Canvas.MoveTo(Origin.X, Origin.Y);        //펜의 위치를 origin 으로
    Canvas.LineTo(X, Y);                      //새로운 라인을 그린다.
  end;
  MovePt := Point(X, Y);                      //현재 위치를 기록한다.
  Canvas.Pen.Mode := pmCopy;
end;
```

펜의 모드를 pmNotXor 로 선택하면, 라인이 배경색과 결합되어 나타나게 된다. 라인을 지우기 위해서는 현재 라인이 그려진 위치에 라인을 그리면 된다. 그리고 나면, 펜의 모드를 pmCopy(디폴트 값)로 바꾼다.

이제 완성이 되었다. 특별히 펜의 스타일이나 색상 등을 정하지 않고 디폴트 값을 사용하였으므로, 멋도 없는 매우 단순한 어플리케이션이지만 기본적으로 마우스를 사용한 드로잉 프로그램을 제작하는 방법에 대해서는 익혔을 것으로 믿는다.

이 어플리케이션을 실행하고 라인을 마음대로 그려보자. 그러면 다음과 같은 형태의 실행 화면을 얻을 수 있을 것이다.



스크린 캡쳐 어플리케이션의 제작

이번에는 간단한 스크린 캡쳐 어플리케이션을 제작해보자. 이번 어플리케이션에서 배울 점은 DC 에 대한 개념과 비트맵을 처리할 때 많이 사용되는 API 함수를 익히는 것이다.
먼저 폼을 다음과 같이 디자인 한다. 캡쳐 프로그램의 폼은 그다지 화려하지 않아도 되고, 어차피 캡쳐를 할 때 폼이 숨겨져야 하므로 단순하게 디자인한다.

캡쳐한 파일을 저장해야 하므로, TSaveDialog 컴포넌트를 하나 추가한다. 그리고, DefaultExt 프로퍼티를 '.bmp', Filter 프로퍼티를 '*.bmp|*.bmp'로 설정하자.
캡쳐할 이미지를 저장할 TImage 변수를 다음과 같이 전역 변수로 선언한다.

```
var
  Form1: TForm1;
  CaptureImage: TImage;
```

그러면, 실제로 캡쳐를 수행하는 프로시저를 작성해보자. 다음과 같이 Capture 라는 프로시저를 private 섹션에 선언한다.

```
private
  procedure Capture;
```

그러면, Capture 프로시저를 실제로 구현해보자. 스크린 전체를 캡쳐하는 프로시저는 다음과 같이 구현할 수 있다.

```
procedure TForm1.Capture;
var
  DC, DCBuffer, Buffer: hDC;
  x, y: integer;
begin
  Hide;
  Sleep(200);
  DC := CreateDC('DISPLAY', nil, nil, nil);
  x:= Screen.Width;
  Y:= Screen.Height;
  DCBuffer := CreateCompatibleDC(DC);
  Buffer := CreateCompatibleBitmap(DC, x, y);
  SelectObject(DCBuffer, Buffer);
  BitBlt(DCBuffer, 0, 0, x, y, DC, 0, 0, SRCCOPY);
  BitBlt(CaptureImage.Canvas.Handle, 0, 0, CaptureImage.Width,
    CaptureImage.Height, DCBuffer, 0, 0, SRCCOPY);
  DeleteDC(DCBuffer);
  DeleteDC(DC);
  CaptureImage.Refresh;
  Show;
end;
```

지역 변수로 선언한 DC 는 디스플레이 객체의 디바이스 컨텍스트를 저장한다. 그리고, DCBuffer 에는 DC 에 대한 메모리 디바이스 컨텍스트를 생성해서 대입하며, 실제 비트맵에 대한 디바이스 컨텍스트는 Buffer 변수에 저장된다. x, y 는 캡쳐할 화면의 크기를 담는 변수로 쓰인다.
먼저 Hide 를 호출하여 폼을 숨긴다. 그 이후에 Sleep(200)을 호출한 이유는 폼을 숨기는데 약간의 시간을 벌어주기 위한 것이다. 필자가 테스트한 바로는 바로 캡쳐를 시도할 경우 폼이 숨겨지는 모양이 그대로 캡쳐되어 버린다. 그 다음 라인에서는 사용할 디바이스

컨텍스트를 생성한다. CreateDC 함수는 지정된 이름의 디바이스 컨텍스트를 생성한다. 여기서는 화면에 보여주는 비트맵이므로 'DISPLAY'로 설정하여 디스플레이 디바이스 컨텍스트를 선택한다. 디바이스 컨텍스트에 대한 자세한 내용은 이장의 앞 부분에 잘 정리해 놓았으므로, 이를 참고하기 바란다.

그 다음에는 캡쳐할 이미지의 크기를 스크린 전체 크기로 설정하고, DCBuffer 와 Buffer 변수의 값을 설정한다. CreateCompatibleDC 함수는 지정된 디바이스에 대한 메모리 디바이스 컨텍스트를 생성하는 함수이다. 비트맵을 선택하면, 디바이스 컨텍스트는 스크린으로 복사되거나 인쇄될 이미지를 준비하는데 사용된다. CreateCompatibleDC 함수는 래스터 작업을 지원하는 장치에서만 사용될 수 있다.

디스플레이 DC 는 보존성이 없기 때문에, 이를 통해 출력한 것은 다른 윈도우가 그 위치를 덮어 씌우기만 해도 문제가 생길 수 밖에 없다. 이렇게 하면 윈도우의 특성 상 좋지 않은 것은 당연하다. 이를 막기 위해서는 디스플레이 DC 의 내용을 계속 간직하고 있을 보존성이 있는 DC 가 필요하게 되는데, 이런 것이 있다면 디스플레이 DC 의 내용을 여기에 보관했다가 필요할 때 다시 그려주면 될 것이다.

이 때 사용하는 것이 CreateCompatibleDC 함수를 이용해서 생성하는 메모리 DC 이다. 메모리 DC 의 용도로는 이미지를 반복적으로 사용할 때와 비트 연산을 하고자 할 때를 들 수 있다. 메모리 DC 는 경우에 따라서 빠르게 생성해서 사용할 수 있고, 저장도 어느 정도 가능하기 때문에 장점이 많지만, 메모리를 소모한다는 단점을 가지고 있다. 별 것 아닌 것 같지만 디스플레이에 필요한 메모리가 워낙 크기 때문에, 사용을 남발하는 것은 절대로 삼가해야 하며, 사용한 뒤에는 반드시 DeleteDC 를 호출하여 꼭 해제를 해 주어야 한다.

Buffer 변수에 값을 대입하기 위해서 사용하는 CreateCompatibleBitmap 함수는 지정된 디바이스 컨텍스트의 장치와 호환되는 비트맵을 생성한다. 이 함수에 의해 생성되는 비트맵의 컬러 포맷은 파라미터에 지정된 디바이스의 컬러 포맷과 일치한다. 이 비트맵은 디바이스와 호환되는 메모리 디바이스 컨텍스트에 선택되어 사용되는 경우가 많다. 메모리 디바이스 컨텍스트는 컬러와 모노 비트맵을 모두 허용하기 때문에, 디바이스 컨텍스트로 메모리 디바이스 컨텍스트가 지정될 경우, CreateCompatibleBitmap 함수에 의해 반환되는 비트맵의 포맷은 다를 수 있다. CreateCompatibleBitmap 함수의 두번째와 세번째 파라미터는 비트맵의 크기를 지정한다. 여기서는 스크린의 크기를 지정한다.

그 다음에는 SelectObject 함수를 이용해서 지정된 디바이스 컨텍스트에 사용할 그래픽 객체를 선택한다. 여기서는 비트맵 객체인 Buffer 를 선택해야 한다.

이제 가장 중요한 BitBlt 함수의 사용법을 익힐 차례이다. BitBlt 함수에 의해서 실제로 비트맵에 그림을 대입하게 된다. BitBlt 함수는 지정된 소스 디바이스 컨텍스트에서 목적 디바이스 컨텍스트로 해당되는 픽셀들의 컬러 데이터를 전송하게 된다. 이때 스트레치(stretch), 압축(compress), 회전(ratate) 등의 변형을 할 수가 있다. 소스와 목적 디바이스 컨텍스트의 컬러 포맷이 맞지 않으면, 소스 컬러 포맷을 목적 포맷에 맞추어 변환을 한다. 우리가 사용한 두가지 디바이스 컨텍스트는 서로 호환되도록 설정하였으므로 컬러 포맷의

문제는 없다.

BitBlt 함수는 파라미터가 모두 9 개로 무척 많은데, 사실 그 내용을 보면 그다지 어렵지 않다. 첫번째 파라미터에는 목적 디바이스 컨텍스트의 핸들을 지정하며, 2~5 번째 파라미터는 목적 디바이스 컨텍스트에 복사할 영역의 4 군데 좌표를 설정한다. 6 번째 파라미터는 소스 디바이스 컨텍스트의 핸들을 지정하면 되고, 7~8 번째 파라미터는 소스 디바이스 컨텍스트의 좌측상단 꼭지점의 좌표를 설정한다. 마지막으로 9 번째 파라미터에는 레스터 작업의 코드를 지정한다. 여기서 가장 중요한 것은 마지막 파라미터인데 지우거나, 복사, 반전 등의 여러가지 효과를 지정할 수 있다. Capture 프로시저에 사용한 BitBlt 함수의 역할은 디바이스 컨텍스트인 디스플레이 장치의 전체 스크린 영역을 일단 메모리 디바이스 컨텍스트로 복사한 후, 이를 다시 CaptureImage 이미지 컴포넌트의 캔버스로 복사하는 것이다. 이때에는 SRCCOPY 를 사용하여 소스 영역의 내용을 직접 목적 영역으로 복사한다. 이것으로 캡쳐 작업이 완료된다.

레스터 작업 코드는 여러가지로 사용되기 때문에, 익혀두면 많은 도움이 된다. 이를 다음에 정리하였으므로, 참고하기 바란다.

값	설 명
BLACKNESS	목적 영역을 물리적 팔레트의 인덱스 0 인 색상(보통 검은 색)으로 채운다.
DSTINVERT	목적 영역을 반전한다.
MERGECOPY	소스 영역의 색상을 AND 연산을 통해 지정된 패턴으로 색상을 합친다.
MERGEPAINT	반전된 소스 영역의 색상을 목적 영역의 색상과 OR 연산을 통해 합친다.
NOTSRCCOPY	소스 영역을 목적 영역으로 반전하여 복사한다.
NOTSRCERASE	소스와 목적 영역의 색상을 OR 연산을 해서 합친 후, 이를 반전한다.
PATCOPY	지정된 패턴으로 목적 비트맵에 복사한다.
PATINVERT	지정된 패턴의 색상과 목적 영역의 색상을 XOR 연산으로 합친다.
PATPAINT	소스 영역의 반전된 색상과 패턴의 색상을 OR 연산을 하고, 이를 다시 목적 영역의 색상과 OR 연산을 한다.
SRCAND	소스와 목적 영역의 색상을 AND 연산한다.
SRCCOPY	소스 영역의 내용을 목적 영역으로 직접 복사한다.
SRCERASE	목적 영역의 반전된 색상과 소스 영역의 색상을 AND 연산한다.
SRCINVERT	소스와 목적 영역의 색상을 XOR 연산한다.
SRCPAINT	소스와 목적 영역의 색상을 OR 연산한다.
WHITENESS	목적 영역을 물리적 팔레트의 인덱스 1 인 색상(보통 흰색)으로 채운다.

그러면, 이제 폼의 OnCreate 이벤트 핸들러에서 이미지 컴포넌트를 생성하고, 프로퍼티를 지정한다. 그리고, Capture 함수를 호출하여 일단 스크린을 CaptureImage 변수에 캡쳐한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  CaptureImage := TImage.Create(Self);
  CaptureImage.Left := 0;
  CaptureImage.Top := 0;
  CaptureImage.Width:=Screen.Width;
  CaptureImage.Height:=Screen.Height;
  Capture;
end;
```

Button1 과 Button2 의 이벤트 핸들러를 다음과 같이 작성하여 캡쳐를 하고, 캡쳐한 이미지를 저장할 수 있도록 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Capture;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
    CaptureImage.Picture.Bitmap.SaveToFile(SaveDialog1.FileName);
end;
```

지역(Region) 이란 ?

지역은 상상할 수 있는 모든 형태의 모양을 클립핑 영역으로 설정할 수 있도록 허용한다. 이를 이용하면 그리게 되는 모든 것들이 그 영역에 클리핑되거나 마스크되게 할 수 있다. 여기에 대해서는 델파이 자체가 지원하는 메소드는 없다. 그러므로, 이를 사용하기 위해서는 GDI 를 직접 사용해야 한다.
지역을 사용하는 예제를 간단하게 하나 만들어 보자. 폼에 버튼을 하나 얹고, Caption 을 '실 행'으로 설정한 뒤 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
```

```
  NewRgn, OldRgn: hRgn;
begin
  NewRgn := CreateEllipticRgn(0, 0, 100, 100);
  OldRgn := SelectObject(Canvas.Handle, NewRgn);
  PatBlt(Canvas.Handle, 0, 0, Width, Height, BLACKNESS);
  SelectObject(Canvas.Handle, OldRgn);
  DeleteObject(NewRgn);
end;
```

지역을 사용하기 위해서는 일단 2 개의 지역을 지정할 수 있는 핸들을 저장할 변수를 선언한 후 CreateEllipticRgn 함수를 실행하여 동그란 형태의 지역을 NewRgn 변수에 저장한다. 그리고, SelectObejct 함수로 NewRgn 지역 객체를 선택한다. 이때 SelectObject 객체는 이전의 지역 객체를 반환하므로, 이를 저장해 두었다가 나중에 지역을 복구할 때 사용한다. PatBlt 함수는 주어진 사각 영역에 현재 선택된 브러쉬를 이용해서 칠하는 함수로, 칠하는 방법을 마지막 파라미터에 지정하도록 되어 있다. 여기서는 BLACKNESS 를 지정하여 까맣게 칠하도록 하였다. 그 다음에는 원래의 지역을 복구하고, 지역 객체를 해제하면 끝난다. 이 어플리케이션을 실행하면 분명히 PatBlt 로 사각형 영역을 칠하도록 했지만, NewRgn 이 동그란 영역으로 지정되었으므로 다음과 같이 동그랗게 칠해진다.

매핑 모드 (Mapping Modes)

일반적으로 윈도우의 기본적인 매핑 모드는 MM_TEXT 를 사용한다. 이 모드에서는 모든 드로잉 좌표가 좌상측에서 우하로 내려올수록 X, Y 가 1 픽셀씩 증가한다. 즉, 좌상 꼭지점을 (0, 0)으로 하여 우하로 내려오면 양의 값으로 증가하는 것이다.
윈도우에는 이 모드 말고도 여러가지 매핑에 대한 모드가 존재하는데, 여기에는 다음과 같은 것들이 있다.

모 드	설 명

MM_TEXT	1 픽셀 단위로, y 가 증가할수록 아래로 내려온다.
MM_LOMETRIC	0.1mm 단위로, y 가 증가할수록 위로 올라간다.
MM_HIMETRIC	0.01mm 단위로, y 가 증가할수록 위로 올라간다.
MM_LOENGLISH	0.01 인치 단위로, y 가 증가할수록 위로 올라간다.
MM_HIENGLISH	0.001 인치 단위로, y 가 증가할수록 위로 올라간다.
MM_TWIPS	1/20 포인트(1/1440 인치) 단위로, y 가 증가할수록 위로 올라간다.
MM_ISOTROPIC	SetWindowExtEx 와 SetViewportExtEx 함수를 이용하여 단위와 축의 방향을 결정한다. 그런데, x 와 y 의 단위는 같은 크기이다.
MM_ANISOTROPIC	MM_ISOTROPIC 과 거의 동일하지만, x 와 y 의 단위 크기가 다를 수 있다.

매핑 모드를 사용할 때에는 이들의 변형을 위해서 몇 가지 함수를 사용할 수 있다. 이들 함수는 과거의 설정을 반환하므로 이를 임시 변수에 저장했다가, 사용이 끝나면 다시 복구시켜 주는 것이 좋다. 가장 흔히 사용되는 함수로는 다음과 같은 것들이 있다.

1. SetMapMode: 매핑 모드를 결정한다.
2. SetWindowOrgEx: (0, 0)이 위치할 좌표를 설정한다.
3. SetWindowExtEx: 논리적 단위를 결정한다.
4. SetViewportExtEx: 논리적 단위가 매핑할 유닛의 디바이스 크기를 결정한다.

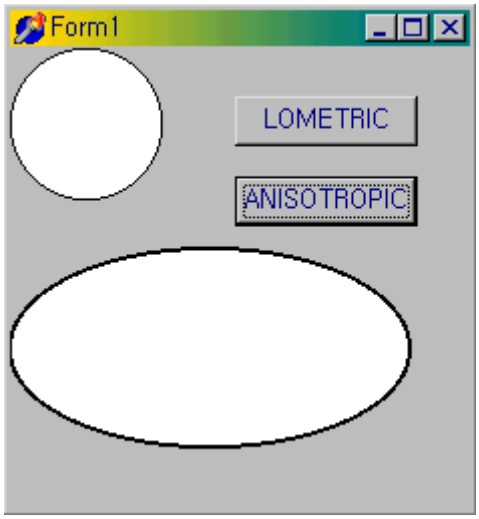
간단한 예제를 하나 만들어 보자. 폼에 버튼을 2 개 얹고 각각의 캡션을 'LOMETRIC', 'ANISOTROPIC'으로 설정한다. 버튼의 캡션에서도 추측할 수 있겠지만, Button1 은 MM_LOMETRIC 모드를, Button2 는 MM_ANISOTROPIC 모드를 사용한다.
먼저 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldMapMode: Integer;
  OldOrigin: TPoint;
begin
  OldMapMode := SetMapMode(Canvas.Handle, MM_LOMETRIC);
  SetWindowOrgEx(Canvas.Handle, 0, 200, @OldOrigin);
  Canvas.Ellipse(0, 0, 200, 200);
  SetWindowOrgEx(Canvas.Handle, OldOrigin.X, OldOrigin.Y, nil);
  SetMapMode(Canvas.Handle, OldMapMode);
end;
```

코드는 단순하다. SetMapMode 함수를 이용하여 MM_LOMETRIC 모드로 설정하고, 원점을 (0, 200)으로 설정한다. MM_LOMETRIC 모드는 0.1mm 단위이므로 원점은 폼의 좌상에서 아래로 2cm 아래에 위치하게 된다. 여기에서 지름이 2cm 인 원을 그리게 된다. 그리고 나서, 원래의 원점과 매핑 모드를 각각 OldMapMode 와 OldOrigin 변수에 저장했던 것을 다시 복구하면 된다.
Button2 의 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  OldMapMode: Integer;
  OldOrigin: TSize;
  OldWindowExtent: TSize;
  OldViewportExtent: TSize;
begin
  OldMapMode := SetMapMode(Canvas.Handle, MM_ANISOTROPIC);
  SetWindowExtEx(Canvas.Handle, 100, 100, @OldWindowExtent);
  SetViewportExtEx(Canvas.Handle, 200, 100, @OldViewportExtent);
  SetWindowOrgEx(Canvas.Handle, 0, -100, @OldOrigin);
  Canvas.Ellipse(0, 0, 100, 100);
  SetWindowOrgEx(Canvas.Handle, OldOrigin.cx, OldOrigin.cy, nil);
  SetViewportExtEx(Canvas.Handle, OldViewportExtent.cx, OldViewportExtent.cy, nil);
  SetWindowExtEx(Canvas.Handle, OldWindowExtent.cx, OldWindowExtent.cy, nil);
  SetMapMode(Canvas.Handle, OldMapMode);
end;
```

별로 다를 내용은 없지만, MM_ANISOTROPIC 모드에서 SetViewportExtEx 함수에 의해 x 축의 크기 요소를 y 축의 2 배로 설정하였으므로 Ellipas(0, 0, 100, 100) 메소드에 의해 가로의 길이가 세로의 2 배인 타원이 그려지게 된다.
이 어플리케이션을 실행하고, 두 개의 버튼을 차례로 클릭하면 다음과 같은 결과를 볼 수 있다.



어플리케이션에 비디오 클립 추가

델파이 4 의 애니메이션 컨트롤을 이용하면 조용한 비디오 클립을 간단하게 어플리케이션에 추가할 수 있다. Win32 페이지의 TAnimate 컴포넌트를 사용하려면 CommonAVI, FileName, ResName, ResID 프로퍼티 중의 하나를 설정해서 보여줄 비디오 클립을 선택한다. 일단 AVI 파일을 메모리에 적재한 뒤에, Active 프로퍼티를 설정하거나 Play 메소드에 의해 스크린에 AVI 클립을 보여줄 때 첫번째 프레임부터 보여주려 할 경우에는 Open 프로퍼티를 True 로 설정한다.
그리고, AVI 클립을 반복할 횟수를 Repetitions 프로퍼티에 설정하는데, 이 값에 0 을 주면 Stop 메소드가 호출될 때까지 재생이 반복된다. 그 밖에 여러가지 설정을 변경할 수 있는데 예를 들어, 첫번째 프레임부터 보여주지 않고 특정 프레임부터 시작하게 하려면 StartFrame 프로퍼티를 프레임의 번호로 설정하면 된다.

오디오/비디오를 모두 재생할 수 있는 어플리케이션의 제작

오디오와 비디오를 모두 재생할 수 있는 어플리케이션을 제작하려면, System 페이지에 있는 TMediaPlayer 컴포넌트를 사용한다.
이 컴포넌트를 사용하려면 먼저 DeviceType 프로퍼티를 사용하려는 적절한 디바이스 유형으로 설정해야 한다. 이 프로퍼티가 dtAutoSelect 로 설정되면 디바이스 유형은 미디어 파일의 확장자에 의해 결정된다. 디바이스가 미디어를 파일로 저장하면, 미디어 파일의 이름을 FileName 프로퍼티에 지정하여 사용하게 된다
AutoOpen 프로퍼티를 True 로 설정하면, 미디어 플레이어는 자동으로 지정된 디바이스를 열게 된다. 이 값이 False 이면 Open 메소드가 호출되어야만 디바이스가 열린다.
AutoEnable 프로퍼티는 미디어 플레이어 버튼을 자동을 enable, disable 할 것인지를 결정

하는 것이다. 이를 설정하거나 아니면, EnableButtons 프로퍼티를 이용하여 각각의 버튼을 enable, disable 시킬 수 있다.
미디어 플레이어의 버튼으로는 다음과 같이 Play, Pause, Stop, Next, Previous 등이 있다.

경우에 따라서는 미디어 플레이어를 런타임에 보이지 않게 하고 싶을 때에는 Visible 프로퍼티를 False 로 설정하면 되며, 이럴 때에는 Play, Pause, Stop, Next, Previous 등의 적절한 메소드를 호출하여 작동시키면 된다.
그 밖에도 미디어가 디스플레이 윈도우를 요구할 경우에는 Display 프로퍼티를 미디어를 디스플레이할 컨트롤로 설정하면 되며, 디바이스가 다중 트랙을 사용할 경우에는 Tracks 프로퍼티를 해당되는 트랙으로 설정하면 된다.
디바이스 유형과 재생되는 종류와 트랙과 디스플레이 윈도우의 사용 여부에 대한 사항을 다음에 나타내었다.

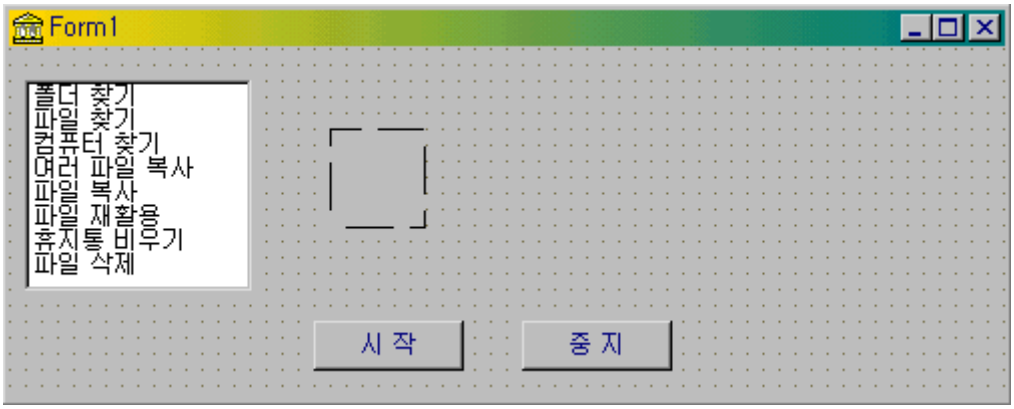
디바이스	사용되는 하드웨어/소프트웨어	트랙 사용여부	디스플레이 컨트롤 사용여부
dtAVIVideo	AVI Video Player for Windows (AVI Video files)	No	Yes
dtCDAudio	CD Audio Player for Windows CD Audio Player (CD Audio Disks)	Yes	No
dtDAT	Digital Audio Tape Player (Digital Audio Tapes)	Yes	No
dtDigitalVideo	Digital Video Player for Windows (AVI, MPG, MOV files)	No	Yes
dtMMMovie	MM Movie Player (MM film)	No	Yes
dtOverlay	Overlay device (Analog Video)	No	Yes
dtScanner	Image Scanner N/a for Play (scans images on Record)	No	No
dtSequencer	MIDI Sequencer for Windows (MIDI files)	Yes	No
dtVCR	Video Cassette Recorder (Video Cassettes)	No	Yes
dtWaveAudio	Wave Audio Player for Windows (WAV files)	No	No

애니메이션 컨트롤의 활용

Animate 컨트롤은 단일 비디오 스트림을 가진 AVI 파일 만을 재생할 수 있으며, RLE8 압축 기법으로 압축되거나 압축을 해제하며, 팔레트 변경은 할 수 없다. 그리고, 사운드가 있을 경우 압축이 무시된다.

Animate 컨트롤의 재미있는 특징 중에 하나는 폼 위에 올려 놓고, FileName 이나 CommonAVI 프로퍼티를 설정하고 Active 프로퍼티를 True 로 하면, 디자인 시에서도 애니메이션을 볼 수 있다. 그러면, 표준으로 제공되는 애니메이션인 CommonAVI 프로퍼티의 내용을 사용자가 선택하면 이를 보여주는 간단한 예제를 하나 만들어 보자

먼저 폼에 리스트 박스 하나와 버튼 2 개, 애니메이트 컨트롤 1 개를 얹어서 다음과 같이 디자인 한다.



리스트 박스의 Items 프로퍼티를 앞의 화면에 보이는 대로 설정하고, 버튼 2 개의 Caption 프로퍼티를 '시 작'과 '중 지'로 설정한다.

ListBox1 의 OnClick 이벤트 핸들러를 다음과 같이 설정한다.

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  case ListBox1.ItemIndex of
    0: Animate1.CommonAVI := aviFindFolder;
    1: Animate1.CommonAVI := aviFindFile;
    2: Animate1.CommonAVI := aviFindComputer;
    3: Animate1.CommonAVI := aviCopyFiles;
    4: Animate1.CommonAVI := aviCopyFile;
    5: Animate1.CommonAVI := aviRecycleFile;
    6: Animate1.CommonAVI := aviEmptyRecycle;
    7: Animate1.CommonAVI := aviDeleteFile;
  end;
end;
```

즉, 리스트 박스에 보여주는 내용을 그대로 적용하는 것이다. 그리고, 두 개의 버튼의 OnClick 이벤트 핸들러는 각각 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Animate1.Active := True;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Animate1.Stop;
end;
```

이제 완성이 되었으므로, 이를 실행해 보자. '파일 복사'를 선택하면 다음과 같은 화면을 볼 수 있을 것이다.

정 리 (Summary)

이번 장에서는 델파이를 이용하여 그림을 그리고, 비트맵을 다루는 방법과 비디오와 오디오 클립을 사용하는 방법에 대해서 알아보았다. 사실 그래픽에 대한 부분은 따로 책을 한 권 써야할 정도로 설명할 내용도 많고, 이해해야 할 내용도 많다. 그리고, 시각적 컨트롤을 나름대로 제작해서 쓰려고 하는 사람들은 여기에 대한 필수적인 이해가 있어야 한다.
팔레트에 대해서도 잘 알고 있어야 하며, 그래픽이라는 객체들이 크기가 상당히 큰 경우들이 많기 때문에 메모리를 효과적으로 사용하는 방법이나 파일 포맷과 파일을 다루는 방법 등에 대해서도 잘 알고 있어야 한다.
이 책에서는 지면 관계상 그래픽에 대해서 가장 기본적이고 기초적인 부분만 소개하였다.

하지만, 훌륭한 델파이 프로그래머가 되기 위해서는 꼭 넘어야 할 산이므로 단순히 델파이가 제공하는 캔버스의 메소드에 의존하지 말고, 필수적인 API 함수들의 사용법은 반드시 익혀놓도록 권하는 바이다.

다음 장에서는 윈도우 95 가 등장하면서부터 제공되기 시작한 Win32 의 향상된 공통 컨트롤(Common Control) 들의 사용법과 활용 방법에 대해서 알아볼 것이다. 델파이의 수많은 컴포넌트 들 중에서 유용하게 사용될 수 있음에도 불구하고, 사용법에 대한 소개와 활용이 비교적 미흡한 편인 컨트롤 들을 중심으로 설명할 것이다.