MFC 4 - 1

www.gamecodi.com / 이주행 (master@gamecodi.com)

1. 스프라이트란 ?

스프라이트란 화면에 그려진 그림 위에 그림을 합성, 출력 하고 이동할 수 있는 기능을 의미한다. 이런 기능은 MSX (마이크로 소프트와 아스키가 만든 8Bit 컴퓨터) 에서 처음 사용되었으며 게임 개발에 있어서 아주 화려하고 띄어난 매력적인 기능이었다.

스프라이트는 그래픽 화면과 상관 없이 출력과 이동이 가능 하였으며, 충돌 처리, 클리핑 처리까지 모두 하드웨어 적으로 지원을 해주었다.

스프라이트 기능은 그래픽 화면 외에 투명색으로 된 스프라이트 화면이 제공되었다. (포토샵의 레이어와 비슷한 개념) 여기에 비트단위의 그림을 그려 넣으면 0 과 1 로 투명색과 출력색상을 구분하여 그래픽 화면과 합성 하였으며, MSX 기종에서는 8x8 크기의 32장의 스프라이트가 가능하였다.

이와 같이 스프라이트의 정의는 그래픽 화면(배경 화면) 위에 배경과 상관 없이 패턴 데이터를 출력하고 이를 이동시킬 수 있는 기능을 뜻한다.

MSX 기종에서는 스프라이트의 처리 자체가 하드웨어 적으로 이루어 졌기 때문에 스프라이트 간의 충돌 처리 역시 하드웨어적으로 처리 되었다. 스프라이트간에 충돌이 일어나면 인터럽트가 발생되어 해당 스프라이트의 번호를 알려주었으므로 간편하게 처리할 수 있었다.

이처럼 스프라이트의 기능은 하드웨어적으로 지원이 되었었지만 우리의 IBM-PC 에서는 전혀 이런 기능이 없으므로 손수 직접 구현을 해줘야만 한다. 스프라이트를 구현함에 있어서 필요한 기능들은 투명색 처리, 충돌 처리, 클리핑 처리가 되겠다.

DirectX 에서는 투명색 처리와 클리핑 처리는 자체적으로 지원되므로 DirectX 의 기능을 사용한다면 더 편리하게 사용할 수 있다.

또한, 모든 게임 처리 자체가 3D 기반으로 처리되는 추세로 바뀌어가고 있으므로 스프라이트 구현 방법은 계속해서 변화되어 가고 있다.

실습) DDB로 BMP를 로드 하여 비트맵DC 에서 화면DC로 픽셀 하나하나를 검사하여 투명색을 제외한 실제 이미지만 화면에 출력하도록 만들어보자. (GetPixel, SetPixel 함수 사용)

2. 스프라이트 툴 이란 ?

이미지 파일(BMP, PCX 등...)을 게임에서 바로 사용 한다면 스프라이트 툴이 필요 없게 된다. 하지만 이런 경우는 그래픽 데이터의 노출 및 하드코딩이 많아지므로 개발에 많은 어려움이 따르게 된다. 그러므로 게임 개발에서 스프라이트 툴을 만드는 것은 필수적이라 할 수 있다.

스프라이트 툴이란 말 그대로 스프라이트를 위한 도구로써 스프라이트 에디터 라고도 한다. 스프라이트로 사용될 이미지를 사용하여 게임에서 사용하기 편리하게 편집하고, 부가적인 정보들을 입력하여 게임에서 빠르고 편리하게 사용할 수 있도록 도와주는 프로그램 이다.

예전 DOS 초창기에는 그래픽 유저 인터페이스 환경이 익숙하지 않았으므로 이미지 편집의 기능 보다는 이미지 파일을 스프라이트 파일로 변환하는 변환기의 역할이 대부분 이었다. 그러므로 당연히 소스 코드에 게임의 데이터를 직접 입력하는 하드코딩이 많이 이루어 졌다.

그래픽 기술이 발전하고 사용자 인터페이스 수준이 올라가며 점점 스프라이트 편집과 추가적인 기능들을 툴을 통해서 입력, 수정 하는 추세가 되어가고 있다.

스프라이트 툴 뿐만 아니라 게임에 사용되는 모든 툴은 게임의 장르, 내용, 기능에 따라서 툴의 기능도 추가, 삭제 되므로 게임개발 초기에 툴을 만들게 된다. 그러므로 처음부터 만능 툴을 만들어 모든 게임에 사용하고자 한다면 이는 쓸모 없는 욕심이며 덩치만 커져서 역효과가 나타나게 된다.

많은 기능이 들어있는 통합 툴과 작은 기능들의 툴을 여러 개 만들어 사용하는 것, 모두 장단점이 있으므로 자신과 개발 팀, 시간 여건에 맞는 툴을 만들어서 사용하면 된다.

3. 스프라이트 툴의 기능 (2D 스프라이트 툴)

스프라이트 툴에 꼭 필요한 필수적 기능으로 정해진 것은 없으므로 여기서는 일반적인 스프라이트 툴을 예로 들어 소개 한다.

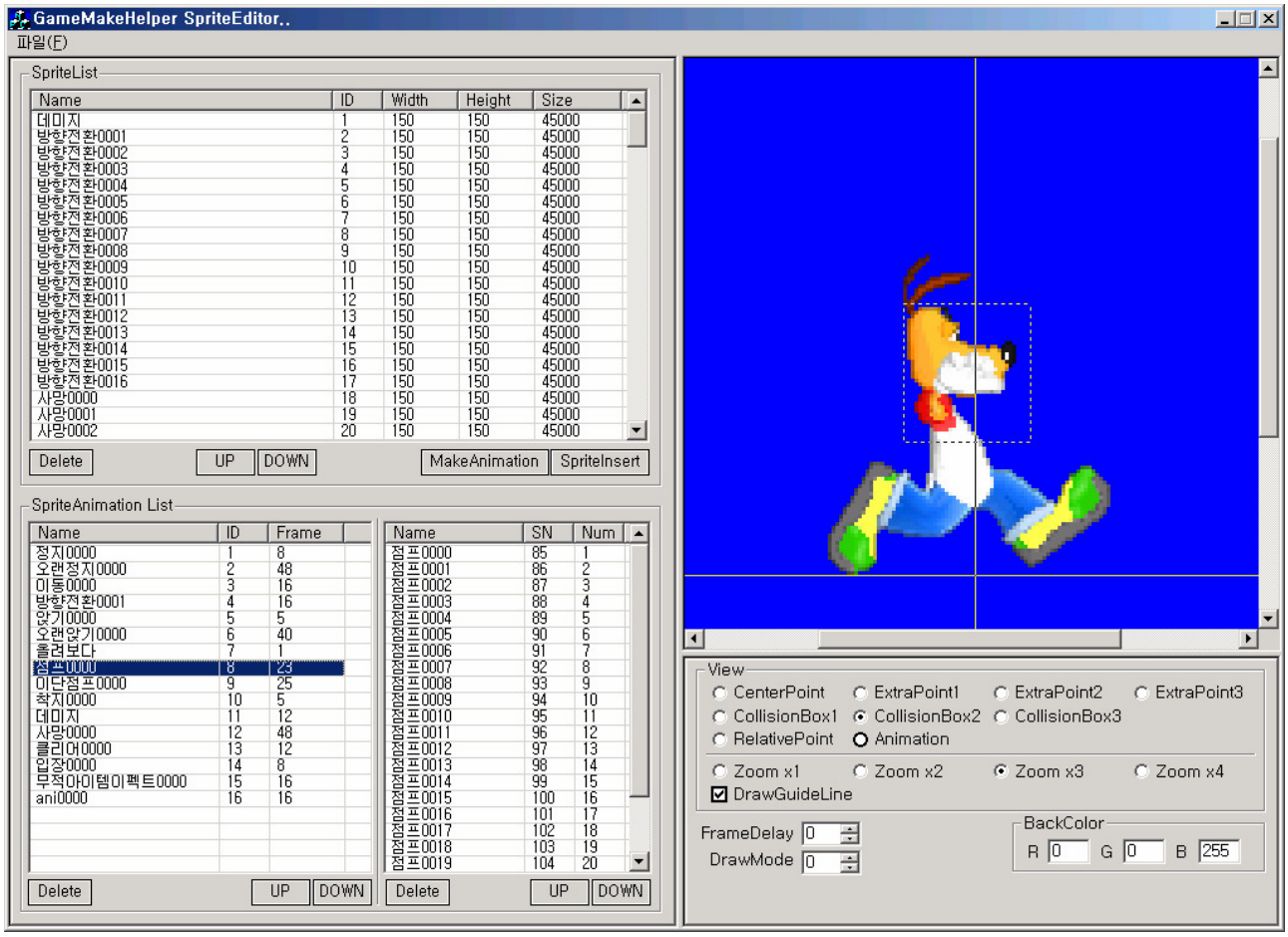
1. 이미지 파일을 읽어 자신만의 스프라이트 파일로 변환.
2. 에니메이션을 위한 중점 좌표 설정.
3. 충돌처리를 위한 충돌 박스 입력.

이외에 추가적으로는 프레임 별 속성, 공격 박스, 데미지 박스, 기타 포인트 등... 이 올 수 있다.

< 스프라이트 툴 샘플 1 >

왼쪽 상단 : 트리 컨트롤을 사용하여 에니메이션 목록을 보여준다.
오른쪽 상단 : 스크롤 뷰를 사용하여 에니메이션 프레임을 보여주며, 선택 가능하다.
왼쪽 하단 : 폼뷰를 사용하여 중앙 하단의 뷰에 대한 기능을 선택한다.
중앙 하단 : 폼뷰에서 선택된 4가지 뷰 중 한가지를 보여주며 해당 기능을 처리한다.
왼쪽 하단 : 현재 선택된 에니메이션을 보여준다.

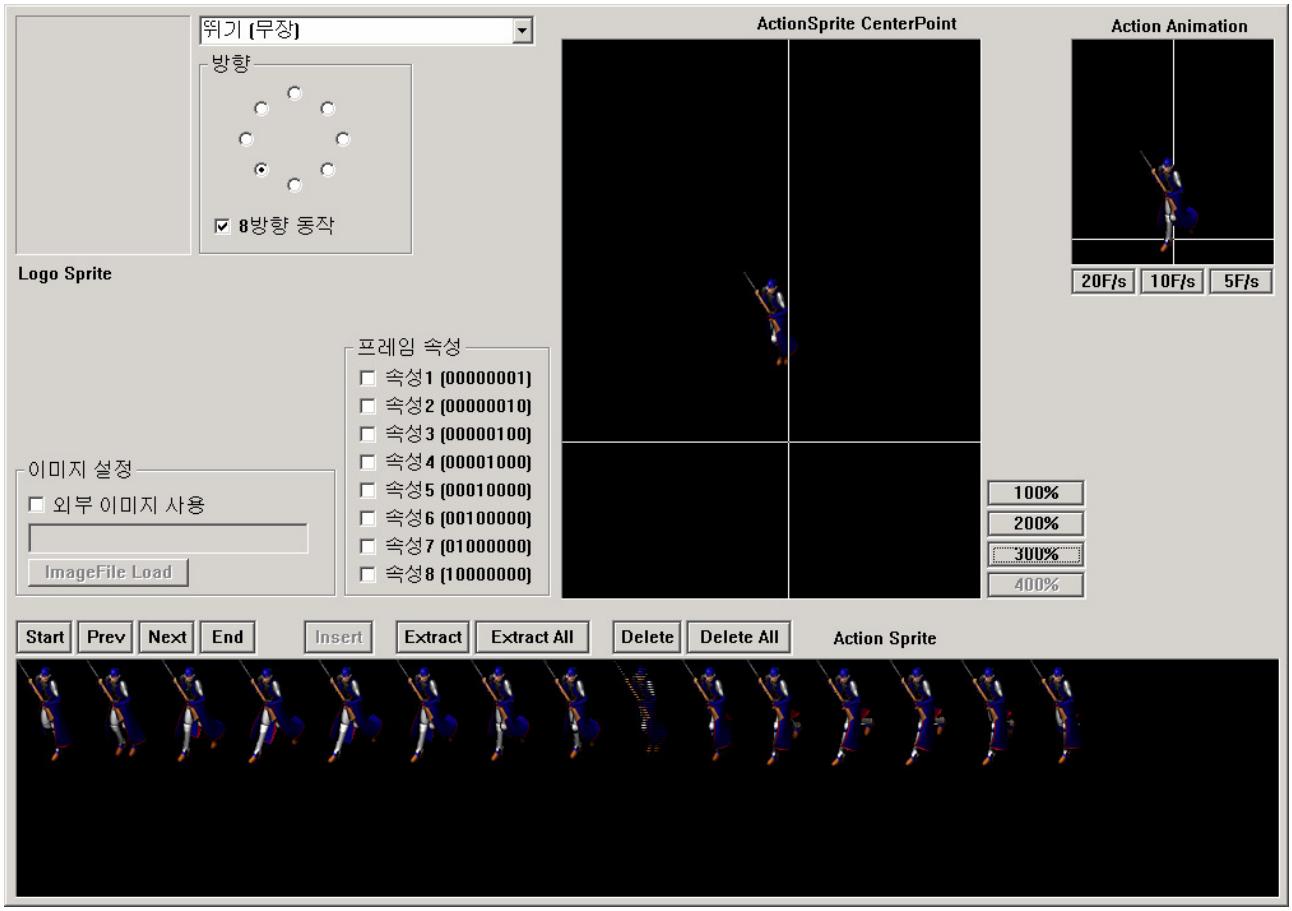
이와 같은 툴은 범용적인 툴로써 에니메이션 개수에 제한이 없으며, 중점/공격박스/피해박스 등... 을 지정할 수 있게 되어있다.



< 스프라이트 툴 샘플 2 >

위의 툴은 파일 하나에 이미지가 하나씩 있는 (3D Max 에서 프레임 별로 렌더링 된 이미지) 파일을 사용할 목적으로 만들어진 툴이다. 리스트 컨트롤을 사용하여 파일 목록을 보여주고 있다.

최대 3가지 충돌 박스를 입력할 수 있으며, 엑스트라 포인트로 부가 포인트 위치를 지정할 수 있도록 되어있다. 프레임 개수, 에니메이션 개수에 상관 없이 추가가 가능하므로 범용적으로 사용할 수 있다.



< 스프라이트 툴 샘플 3 >

위의 툴은 DirectX 를 사용하여 전체 화면에서 만들어 졌으며, RPG 게임 개발 시 게임특성에 맞도록 제작된 툴이다. 캐릭터의 방향 (8방향)과 각 캐릭터별 동작이 모두 정해진 상태에서 개발 되었기 때문에 툴 자체에 방향과 동작에 대한 선택기능이 있다.

RPG 게임이므로 충돌박스에 대한 정보를 입력 받지 않고, 단지 프레임의 에니메이션 중점과 프레임별 속성 값 만을 입력 받고 있다.

툴에는 기본적인 틀이 없으므로 만들고자 하는 게임에 적용 가능하고 편리한 툴을 만들면 된다.

4. 스프라이트 파일의 구조.

스프라이트 툴의 파일 구조는 자신만의 파일이므로 취향대로 저장 하면 된다. 단, 읽을 수 있도록 저장해야 한다. 그리고 메인 프로그래머나 다른 프로그래머의 요청에 의해 파일 구조가 달라져야 하는 경우도 있다.

일반적으로 대부분의 모든 바이너리 파일의 경우는 헤더와 데이터 부분으로 나누어 저장하게 되며, 헤더 부분에는 파일에 대한 정보가 들어가며 데이터 부분은 말 그대로 실제 데이터가 저장된다.

다음은 실제 게임 개발에 사용되었던 스프라이트 파일의 구조 이다.

- SpriteEditor FileInfo -

: 24Bit BMP 파일을 소스로 사용하여 16Bit 칼라키 압축 저장방식을 사용한다.
: 16Bit 압축 저장 시 한 라인당 4바이트 정렬된 메모리 블럭으로 저장이 된다.
- ((CompressLine Size + 3) & ~3) - CompressLine Size 만큼 바이트 추가저장.

ValueType	: Data	: Info.
- Header ----------		
char[32]	: "SpriteFile 0105"	: SpriteFileID.
int	: SpriteImageNum	: 스프라이트 개수.
int	: SpriteAnimationNum	: 에니메이션 개수.
DWORD	: BackColor	: 투명 컬러
BYTE[64]	: ExtraData	: 임시 할당영역.
- SpriteData[SpriteImageNum] ----------		
char[32]	: SpriteName	: 스프라이트 이름.
int	: CompressSpriteSize	: 압축된 스프라이트 사이즈.
int	: NormalSpriteSize	: 압축되지않은 스프라이트 사이즈.
int	: iWidth	: 스프라이트 가로.
int	: iHeight	: 스프라이트 세로.
BYTE[SpriteSize]	: bypImage	: 압축된 용량만큼의 압축스프라이트 데이타.
int	: iCenterX	: 중점 X.
int	: iCenterY	: 중점 Y.

int[3] : iExtraX : 예비 포인트 X.
int[3] : iExtraY : 예비 포인트 Y.

int : iRelativeX : 상대좌표 X.
int : iRelativeY : 상대좌표 Y.

int : iFrameDelay : 프레임 딜레이 값.
int : iDrawMode : 스프라이트 찍는 방법.

DWORD : dwAttribute : Bit 단위의 32개의 복합속성.

int : iCollisionBox1Num : 충돌박스1 개수.
RECT[Box1Num] : CollisionBox1 : 충돌박스1 배열.
int : iCollisionBox2Num : 충돌박스2 개수.
RECT[Box2Num] : CollisionBox2 : 충돌박스2 배열.
int : iCollisionBox3Num : 충돌박스3 개수.
RECT[Box3Num] : CollisionBox3 : 충돌박스3 배열.

BYTE[64] : byExtra : 임시공간.

- AnimationData[SpriteAnimationNum] ----------

char[32] : szName : 에니메이션 이름.
int : iFrameNum : 에니메이션 프레임 개수.

- AnimationFrameData[iFrameNum] ----------

int : iSpriteNum : 스프라이트 번호.
int : iCenterX : 중점 X.
int : iCenterY : 중점 Y.

int[3] : iExtraX : 임시포인트 X.
int[3] : iExtraY : 임시포인트 Y.

int : iRelativeX : 상대좌표 X.

int	: iRelativeY	: 상대좌표 Y.
int	: iFrameDelay	: 프레임 지연값.
int	: iDrawMode	: 스프라이트 찍기 방법.
DWORD	: dwAttribute	: Bit단위의 32개 속성.
int	: iCollisionBox1Num	: 충돌박스1 개수.
RECT[Box1Num]	: CollisionBox1	: 충돌박스1 배열.
int	: iCollisionBox2Num	: 충돌박스2 개수.
RECT[Box2Num]	: CollisionBox2	: 충돌박스2 배열.
int	: iCollisionBox3Num	: 충돌박스3 개수.
RECT[Box3Num]	: CollisionBox3	: 충돌박스3 배열.
BYTE[64]	: byExtra	: 임시할당 영역.

위의 파일 구조는 상당히 복잡한 구조에 속하며 복잡하고 많은 데이터가 있는 구조가 꼭 좋은 구조는 아니므로 각자가 필요한 기능들을 넣어서 사용하면 된다.

5. 스프라이트 0번 압축 방법.

게임에서 스프라이트 이미지를 직접 손수 찍어줄 경우(DirectX 의 Surface 기능을 쓰지 않을 때)에는 if 문을 사용하여 픽셀들의 정보를 하나하나 비교하여 투명색과 그려질 일반 색상을 구분하는 작업이 필요하다.

스프라이트의 개수가 많아지거나 크기가 큰 스프라이트를 찍을 경우에는 속도가 느려지기 때문에 이를 위한 개선 방법으로 0번 압축 방법이 많이 사용된다.
(8Bit 시절 0번 팔레트 색상을 투명색으로 많이 사용했기 때문에 일반적으로 0번 압축이라 부른다.)

0번 압축은 RLE 압축방법에서 조금 변형된 방법으로 이미지에 대한 부분은 전혀 압축하지 않고, 투명색 부분만을 압축하는 방법이다. 2D 게임의 스프라이트에는 화면에 찍혀지지 않아야 할 투명색이 많은 자리를 차지하며, 대부분 바깥 영역에만 일관성 있게 투명색이 뭉쳐져 있기 때문에 좋은 효과를 줄 수 있게 된다.

단, 투명색이 그물처럼 작은 점으로 이루어진 스프라이트의 경우라면 오히려 더 느려지고 용량이 커질 우려가 있다.

위의 스프라이트는 파란색(0,0,255) 부분이 투명색으로 파란색 부분만 압축되게 된다.

기본적인 0번 압축 방법.

(0번 압축 스프라이트는 구현상 약간씩 방법이 다를 수 있지만 기본적인 원리는 모두 같다.)
아래는 가로 10Pixel 의 스프라이트 이미지 중 한 줄을 색상으로 표현 한 것이다.

(0.0.255) (0.0.255) (0.0.255) (0.10.10) (0.60.60) (90.10.0) (0.10.10) (0.0.255) (0.0.255) (0.0.255)

위의 이미지를 보면 검정색으로 칠해진 부분이 이미지 부분이며 나머지 부분은 모두 투명색으로 그림이 그려지지 않을 부분이다. 이 이미지 한 줄의 용량은 40Byte 이며(32Bit 일 경우), 점 하나하나 검사하여 찍어 줄 경우 If 문을 10번 해줘야 하고, 픽셀 역시 4Byte 씩 4번을 찍어줘야만 우리가 원하는 스프라이트 처리가 완료되게 된다.

하지만 이를 0번 압축으로 압축 하게 되면 다음과 같이 줄어들 게 된다.

(0.0.255) (3) (0.10.10) (0.60.60) (90.10.0) (0.10.10) (0.0.255) (3)

이처럼 투명색상이 나온 후 그 뒤에는 투명색이 몇 번 나오게 되는지 반복 횟수가 입력 된다. 이렇게 압축된 스프라이트 이미지는 일단 용량이 줄었다. 투명색 반복 값을 1Byte로 했을 경우 26Byte로 압축되었다. 또한 출력 시에도 투명색에 대한 반복적인 If 문이 필요하지 않게 되었다.

(투명색 발견 시 투명색 반복 횟수를 얻어서 이 횟수 만큼 뛰어 넘어 뒤로 이동 하면 된다.)

< 0번 압축 스프라이트 한 줄 출력 예제>

COLORKEY : 투명색 / pdwDest : 화면 또는 출력 버퍼 / pdwSprite : 스프라이트 이미지

```
DWORD *pdwDest, *pdwSprite;
BYTE byColorKeyNum;

While ( 스프라이트 크기 )
{
	if ( COLORKEY == *pdwSprite )
	{
		pdwSprite++;
		byColorKeyNum = *((BYTE *)pdwSprite);
		pdwDest += byColorKeyNum;

		pdwSprite = (DWORD *)(((BYTE *)pdwSprite) + 1);
	}
	else
	{
		*pdwDest = *pdwSprite;
		pdwDest++;
		pdwSprite++;
	}
}
```

위의 코드에 클리핑 처리 및 스프라이트 크기에 대한 처리가 없으므로 이런 부분을 추가해야 실제 게임에서 사용할 수 있게 된다.

조금 더 발전된 0번 압축 방법.

위의 방법에서는 투명색의 경우는 횟수를 저장하여 한번에 투명색 처리를 하였지만, 실제 이미지 부분에서는 if 문으로 픽셀 하나하나를 비교하여 출력해야만 했다.

이 부분을 조금 더 개선하여 이미지 부분 역시 한번에 메모리 복사를 할 수 있도록 바꿔보도록 하자.

원본 : (0.0.255)(0.0.255)(0.0.255)(0.10.10)(0.60.60)(90.10.0)(0.10.10)(0.0.255)(0.0.255)(0.0.255)

압축 : (3)(4) (0.10.10) (0.60.60) (90.10.0) (0.10.10) (3)(0)

위의 압축 내용을 보면 투명색의 픽셀 값 자체가 없고 단지 일정한 패턴으로 압축되었음을 알 수 있다.

– 투명색길이 – 이미지길이 이미지 | 투명색길이 – 이미지길이 – 이미지

위와 같은 패턴으로 모든 스프라이트 이미지는 [투명색 – 이미지] 패턴으로 이루어 졌다는 기본에서 만들어진 압축 방법이다.

< 0번 압축 스프라이트 패턴방식 한 줄 출력 예제>

COLORKEY : 투명색 / pdwDest : 화면 또는 출력 버퍼 / pdwSprite : 스프라이트 이미지

```
BYTE *pbyDest, *pbySprite;
BYTE byCount;

while ( 스프라이트 크기 )
{
        byCount = *pbySprite;
        pbySprite += 1;
        pbyDest += byCount;

        byCount = *pbySprite;
        pbySprite += 1;
        memcpy(pbyDest, pbySprite, byCount * 3);
        pbySprite += byCount;
        pdwDest += byCount;
}
```

위의 코드 역시 클리핑 처리 및 스프라이트의 크기에 대한 추가적인 부분들이 처리 되어야 한다.