컴퓨터 게임 제작을 위한 그래픽 처리

최학현
건국대학교 컴퓨터학과

Graphic Processing for Computer Game Production

Hak-Hyun Choi
Dept. of Computer, Konkuk University

요 약

본 논문은 컴퓨터 게임 제작을 위한 그래픽 처리 실무기술에 대해서 실제 방법을 고찰해 보고저 한다. 이러한 연구는 국제적인 경쟁력을 가지는 국산 게임개발이라는 과제의 실무기술로 사용될 수 있게 하기 위함이다.

1. 서론

현재 컴퓨터 게임 그래픽 처리 기술의 경우 국내에서 자체 기술을 보유하고 있기는 하지만 폐쇄적인 개발 구조로 인하여 국내 기술의 전반적인 발전에 가장 큰 저해요소가 되고 있으며 몇몇 개발회사를 제외한 중소개발업체들은 대부분 많은 시행착오와 함께 게임에 필요한 그래픽 처리 기술력 보유를 위해 상당한 비중을 두고 기술개발이 진행되고 있는 상태이다. 때문에 직접적으로 사용자들에게 전달되는 게임성 내지는 게임요소 구현보다는 게임 기반 기술에 더 많은 노력을 쏟고 있으며, 필연적으로 개발되는 게임의 질적 수준을 향상시키는데 상당한 걸림돌이 되고 있는 현실이다.

2. 그래픽 처리 및 구현 방법

1. 스프라이트 처리

가. Sprite Image 기본구조

스프라이트 이미지를 출력하기 위해 우선적으로 필요한 것이 출력에 필요한 데이터를 만드는 것이다. 구조체는 아주 기본적인 부분만 기술되어있다. 필요에 따라 원하는 정보를 추가, 제거할 수 있다.

나. Sprite 압축

압축원리는 스프라이트 이미지의 투명색(여기에서는 검정색으로 설정하자.)으로 지정한 RGB(0,0,0)이 반복되는 횟수만큼 수치로 저장하고 이미지는 그대로 저장하는 것이다. 이것을 흔히 0번 컬러 압축이라고 한다. 이 압축 알고리즘은 이미지 라인(Line) 단위로 압축이 이루어진다. 투명색과 이미지를 하나로 묶어 패턴으로 정의하고 한 라인에 몇 개의 패턴이 존재하는지를 나타내는 정보가 선두에 들어간다. 만일 이 값이 3이라면 투명색과 이미지가 3번 반복되는 것을 의미한다. 하나의 패턴에는 투명색이 반복되는 횟수가 저장되며 그 바로 뒤에는 이미지의 개수와 실제 이미지 데이터가 차례대로 저장된다.

* 압축 스프라이트 데이터 구조

0	23	56	98	0	0	12	78	53	0

위의 데이터에서 0이 들어간 부분이 RGB(0,0,0)이고 0 이외의 값이 들어간 회색을 이미지라고 가정하고, 이것을 실제로 압축하면 다음과 같다.

3	1	3	23 56 98	2	3	12 78 53	1	0
패턴의 총개수	0번 패턴의 투명색 개수	0번 패턴의 이미지 개수	0번 패턴의 이미지 데이터	1번 패턴의 투명색 개수	1번 패턴의 이미지 개수	1번 패턴의 이미지 데이터	2번 패턴의 투명색 개수	2번 패턴의 이미지 개수

다. Sprite Tool

게임의 모든 툴들은 프로그래머가 계획한 데이터로 이미지를 변환시키고, 편집하는 역할을 한다. 게임의 장르나 기획에 따라 그에 알맞은 툴이 만들어지며, 프로그래머와 그래픽 디자이너 등 게임을 개발하는 모든 사람들이 사용하게되므로 처음부터 신중하게 계획되어야 한다. 스프라이트 툴을 코딩하기 전에 해야 할 일은 어떠한 기능이 필요한가이다. 반드시 들어가야 할 기능들과 추가적으로 들어갔을 때 작업시간을 줄여주는 편리한 기능들 그리고 나머지 부수적인 것들, 이와 같이 툴을 제작하기 전에 필요한 것들을 정리하여 작업 우선순위를 결정한다.

라. 이미지 출력

이미지를 출력하기 위해 먼저 선행되어야 할 점은 Clipping이다. Clipping이란 그려져야 할 이미지 중 화면에 보여지지 않는 부분을 잘라내는 것을 말한다.

2. GDI를 이용한 처리 방법

가. 점, 선, 면

(1) 점

SetPixel 함수는 지정된 x와 y좌표의 픽셀을 특정 색상으로 설정한다. SetPixel(hdc, x, y, Color);

첫 번째 인자는 장치 컨텍스트에 대한 핸들이다. 두 번째와 세 번째 인자는 클라이언트 영역의 좌상단 코너에 대한 상대 좌표가 된다. 그리고 마지막 인자는 COLORREF 형식으로 색상을 지정한다.

(2) 선

직선을 그리려면 반드시 2개의 함수를 호출해야 한다. 첫 번째 함수는 선이 시작하는 위치를 지정하고 두 번째 함수는 선이 끝나는 위치를 지정한다.

MoveToEx(hdc, xStart, yStart, NULL);

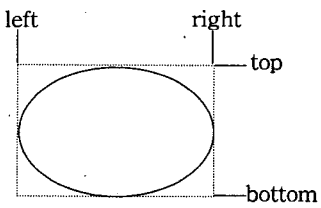
LineTo(hdc, xEnd, yEnd);

(3) 면

이 함수들이 선을 그리기는 하지만 둘러싸인 현재의 영역을 채우기 브러쉬를 사용하여 채우는 것이다. 이 함수들 중 가장 간단한 것은 사각형을 그리는 것이다.

Rectangle(hdc, left, top, right, bottom);

(left, top)포인트는 사각형의 왼쪽 상단 모서리이며 (right, bottom)은 오른쪽 하단 모서리이다. 일단 사각형을 어떻게 그리는지 알고 나면 타원을 어떻게 그리는지도 안 것과 다름없다. 왜냐하면 동일한 인자를 사용하기 때문이다. Ellipse(hdc, left, top, right, bottom);



둥근 모서리를 가지는 사각형을 그리는 함수는 Rectangle과 Ellipse 함수에서처럼 동일한 bounding 상자를 사용하지만 두 개의 인자를 추가로 포함한다.

RoundRect(hdc, left, top, right, bottom, widthEllipse, heightEllipse);

여기서 widthEllipse와 heightEllipse는 타원의 폭과 넓이로, Windows는 작은 타원을 사용하여 둥근 모서리를 그린다. widthEllipse값이 left와 right의 차이와 같고, heightEllipse값이 top과 bottom과 같으면 RoundRect 함수는 타원을 그리게 된다.

나. 그리기 모드

Windows가 펜을 사용하여 선을 그릴 때 실제로는 펜의 픽셀과 목표가 되는 디스플레이 픽셀의 bitwise 2진 연산을 수행한다. 픽셀을 가지고 Bitwise 2진 연산을 수행하는 것을 '래스터 연산' 또는 'ROP'라고 불린다. 선을 그리는 것은 오직 두 개의 픽셀 패턴만(펜과 목표)을 사용하기 때문에 2진 연산은 '2진 래스터 연산' 또는 'ROP2'라고 부린다.

Windows는 펜 픽셀과 목표 픽셀을 조합하는 방법을 나타내는 16개의 ROP2 코드를 정의하였다. 디폴트 장치 컨텍스트에서 그리기 모드는 R2_COPYPEN으로 정의되어 있는데 이것은 Windows가 단순히 펜의 픽셀을 목표에 복사하는 것을 나타내며 이것은 우리가 일반적으로 펜에 대해 생각하는 방식이다. 여기서 P는 새로이 그려지는 그림을 의미하고, D는 기존에 화면에 그려져 있는 그림을 의미한다.

논리연산자	의미
~	NOT
&	AND
|	OR
^	XOR

3. 텍스트 처리 방법

가. 문자 출력

가장 일반적인 텍스트 출력 함수는 TextOut이다. TextOut 함수에서처럼 DrawText도 문자열에 대한 표인터와 문자열의 길이를 필요로 한다. 그러나 만일 NULL로 종료된 문자열을 DrawText에 사용할 경우에는 iCount를 1로 설정하면 Window가 문자열의 길이를 계산하여 준다. iFormat이 0으로 설정되어 있으면 Window는 텍스트를 캐리지리턴 문자('\r' 나 0x0D)나 라이피드 문자('\n'나 0x0A)로 구분된 일련의 행으로 해석된다. 캐리지리턴이나 라인피드는 '개행(newline)'문자로 해석되어 Window는 현재의 행을 종료하고 새로운 행을 시작한다. 새로운 해은 이전 행 아래문자 하나의 높이만큼 떨어진 사각형의 왼쪽부터 시작한다. 문자를 포함한 임의의 텍스트가 사각형의 오른쪽이나 제일 하단에 출력될 경우에는 잘리워진다.

나. 텍스트에 대한 장치 컨텍스트 속성

디폴트 장치 컨텍스트에서 텍스트 색은 검은색이지만 다음 함수로 변경할 수 있다.

SetTextColor(hdc, rgbColor);

예) 문자를 빨간색으로 출력 : SetTextColor(hdc, RGB(255, 0, 0));

Window는 텍스트를 사각형 배경 영역에 출력하는데 이 영역은 배경 모드의 설정에 따른 색상에 기반할 수도 그

렇지 않을 수도 있다. 배경 모드는 다음 함수를 사용하여 변경할 수 있다. SetBkMode(hdc, iMode);

여기서 iMode는 OPAQUE나 TRANSPARENT이다. 디폴트 배경 모드는 OPAQUE로 Window가 배경 색상을 사용하여 사각형의 배경을 채운다는 것을 의미한다. 반대로 TRANSPARENT는 배경색을 채우지 않고 투명처리 한다. 예) 배경투명처리 : SetBkMode(hdc, TRANSPARENT);

배경 색상은 다음 함수를 사용하여 변경할 수 있다.

SetBkColor(hdc, rgbColor);

예) 배경을 파란색으로 출력 : SetBkColor(hdc, RGB(0, 0, 255));

다. 논리적 글꼴

논리적 글꼴은 GDI 개체로 그 핸들은 HFONT 형식의 변수에 저장된다. 논리적 글꼴은 글꼴에 대한 기술이다. 논리적 글꼴은 추상적인 개체로 응용 프로그램에서 SelectObject를 호출하여 장치 컨텍스트로 선택할 때에만 실제적이 된다.

(1) 논리적 글꼴 생성

CreateFont나 CreateFontIndirect를 호출하여 논리적 글꼴을 생성할 수 있다. CreateFontIndirect 함수는 14개의 필드를 가지는 LOGFONT 구조체에 대한 포인터를 취한다. CreateFont 함수는 14개의 인자를 취하는데 이것은 LOGFONT 구조체의 14개 필도와 동일하다.

(2) 글꼴 정보 얻기

자세한 글꼴정보는 GetTextMetrics로부터 얻을 수 있다. GetTextMetircs(hdc, &textmetrics);

여기서 textmetrics는 TEXTMETRIC 형식의 변수이다.

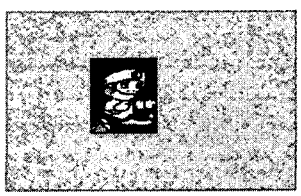
4. 래스터 연산 처리 및 구현 방법

BitBlt 함수는 한 장치 컨텍스트(원본)의 직사각형 영역에서 같은 크기의 다른 장치 컨텍스트의 직사각형 영역(대상)으로 픽셀을 전송한다.

BitBlt(hdcDest, xDest, yDext, cx, cy, hdcSrc, xSrc, ySrc, dwROP);

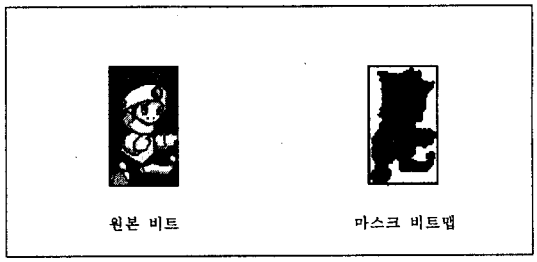
BitBlt 함수는 폭과 높이를 지정하는 두 인수만 가지고 있기 때문에 대상 이미지의 크기는 원본 이미지의 크기와 같다. 복사하면서 이미지의 크기를 늘이거나 줄이려는 경우, StretchBlt 함수를 사용하면 된다.

StretchBlt(hdcDst, xDst, yDst, cxDst, cyDst, hdcSrc, xSrc, ySrc, cxSrc, cySrc, dwROP);



예) 마스크와 레스터 연산을 이용한 합성

위의 그림과 같이 배경위에 이미지를 바로 찍으면 투명색이 바로 찍힌다. 래스터 연산을 이용하여 배경그림과 캐릭터 그림이 자연스럽게 합성되도록 할 수 있다. 먼저, 배경과 합성되는 캐릭터 그림에서 실제 배경을 가리고 그림이 그려질 부분은 검정색으로 되어 있고, 나머지 부분은 흰색으로 되어 있는 비트맵을 만든다. 이와 같은 비트맵을 마스크라고 한다.



원본 비트 마스크 비트맵

마스크의 까만 부분은 모든 비트가 0이므로, 어떤 색깔과 AND연산을 취해도 결과는 0이 되어 검정색이 출력된다. 그리고, 마스크의 하얀 부분은 모든 비트가 1이므로, 어떤 색과 AND연산을 취해도 색이 변하지 않는다. 따라서 실제 아이콘 그림이 그려질 부분만 까맣게 변하고, 나머지 부분은 배경 그림이 그대로 남게 된다. 여기에 다음과 같이 캐릭터 그림을 OR연산을 이용하여 겹쳐 그린다.

BitBlt(hdc, 50, 50, 40, 90, SrcMemDC, 0, 0, SRCPAINT);

이때, 배경 그림이 그대로 표시되어야 하는 부분은 반드시 검정색으로 되어 있어야 한다. 검정색은 모든 비트가 0이므로, 어떤 색과 OR연산을 취해도 색이 변하지 않는다. 또한 마스크에 의해 검정색으로 변한 부분은 모든 비트가 0이므로, 이것과 캐릭터 그림을 OR연산으로 합성하면 캐릭터 그림이 그대로 출력된다.

3. 결론

본 연구에서는 국내에서 접근 가능한 몇가지 게임제작을 위한 그래픽 처리 실무기술에 대해서 실제 구현 방법들을 고찰하였다. 본 논문이 게임을 개발하고 있는 국내 업체와 게임기술을 연구하고 있는 학계 등에 도움이 되기를 기대한다.

[참고문헌]

[1] Mickey Kawick, Real-Time Strategy Game Programming, Wordware Publishing Inc., 1999.

[2] Kevin Hawkins, Dave Astle, Andre Lamothe, OpenGL Game Programming, Prima Tech, 2001.

[3] http://unreal.epicgames.com/computergraphic/

[4] http://www.gamedev.net/gameanimation/

[5] http://unreal.epicgames.com/gamegraphic/