Pocket PC에서 사용되는 GAPI에 대한 설명서

김대희

1. GAPI 의 개요

- GAPI 란?

  Windows CE 는 세가사의 드림캐스에서 볼 수 있듯이 Direct X 라이브러리를 사용할 수 있으며, 몇 개의 게임은 Direct X 로 만들어졌다. 그러나 PPC(Pocket PC) 개발자들은 Direct X 가 PPC 에 대해서 언젠가는 사용할 것이지만 너무 커서 사용하는데 적합하지 않다고 생각한다. 그리고 현재는 3-D 가속기를 사용할 수 없는 실정이나 가속기 없이도 게임이 실행될 수 있다고 여기고 있다. PPC 팀은 이 중간단계를 위해 PPC Game API(GAPI)를 제공한 것이다. GAPI 는 gx.dll 에 들어있는 함수 라이브러리이다. GAPI 를 사용하지않으면, 직접 비디오 메모리를 조작할 수는 없으므로 GDI 만으로는 빠른 게임을 만들 수는 없다.

- Direct X 와의 유사성관계

  GAPI 는 Windows CE 를 전혀 사용하지 않으며, PPC 의 비디오 메모리를 직접 조작할 수 있다. 이것이 Windows CE 용 고성능 게임을 만드는 핵심이다. 또한 비디오 재생과 다른 그래픽 어플리케이션에 유용하다. Direct X 의 가장 빠른 버전을 Direct X 1.0 이라 하지 않았다. 실제로 다선 번째를 Game SDK 라 했으며, WinG 의 다음 버전이라 할 수 있다. WinG 라이브러리는 마이크로소프트사가 윈도우 프로그래머에게 MS-DOS 게임의 장점인 비디오 메모리를 조작할 수 있게 해주는 과정에 시금석을 제공한 것이다.

  WinG 는 BitBlt 과 같이 빠른 속도를 내는 함수로 일반적인 GDI 함수를 대체할 수 있다. 그러나 WinG 는 윈도우 내에서 비디오 메모리를 조작할 수 있는 어떠한 방법도 제공하지 않았다. 그래서 Windows 95 가 나올 때 까지도 또는 그 이후까지도 MS-DOS 게임이 명맥을 유지해왔다. Civilization II 같이 많은 게임이 WinG 를 사용해 제작되었다. 그러나 대체로 많은 비용을 들여 만든 MS-DOS 게임 라이브러리를 계속 사용했다.

  Windows 95 가 출시될 즈음에 마이크로소프트사는 게임 개발자의 반발에 부딪혔다. 새로운 32 비트 운영시스템이 게임 프로그래머가 Windows 의 GDI 를 사용하지 않을 방법을 제시하지 못해 실망을 안겨주었다. Windows 95 출시 후 바로 Game SDK 를 출시하였으며, 이것은 향후에 대한 약속이었다. Game SDK 는 일년 내에 기능이 보강되어 Direct X 2.0 으로 출시되었다. 이 버전을 이용해 큰 인기를 모은 많은 게임들이 만들어졌다. 그 후의 버전은 몇 십가지의 특

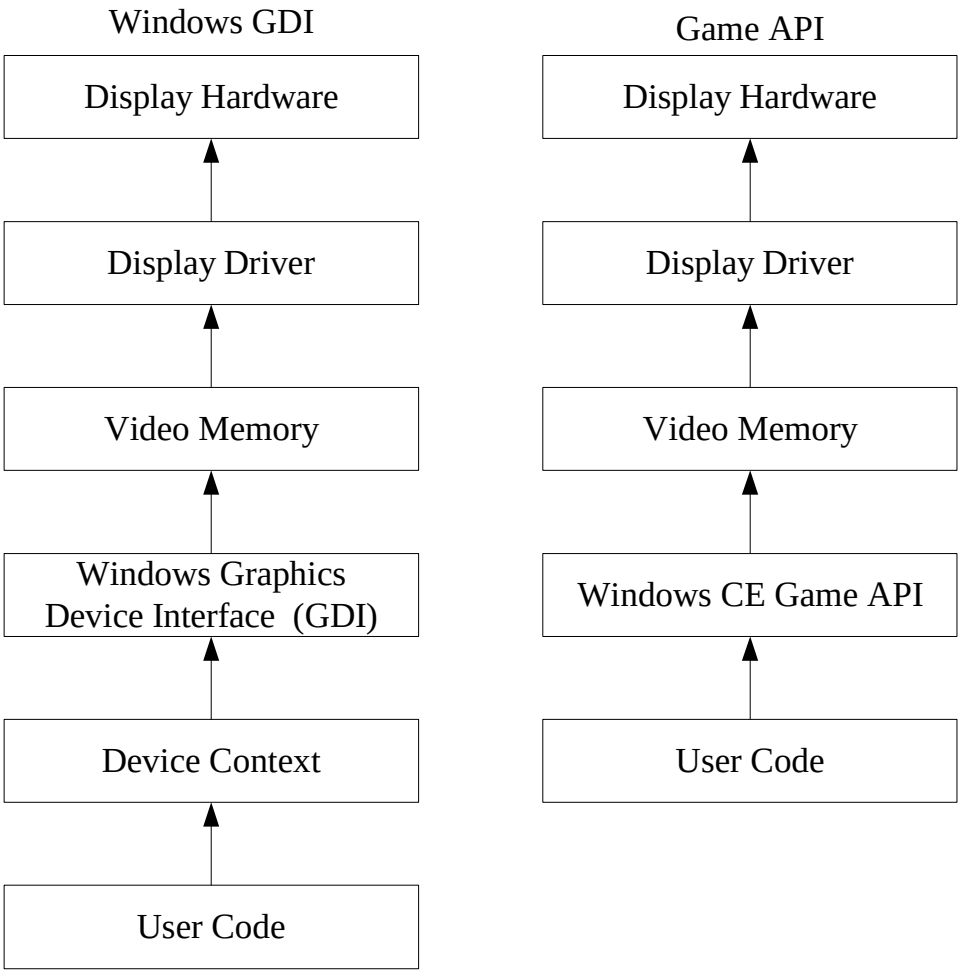
징이 추가되어 이제는 주된 PC 용 게임개발 라이브러리가 되었다.

PPC API 는 GAPI 가 비디오 메모리를 조작할 수 있다는 점에서 첫번째 Direct X 버전과 유사하다. 현재 사용되고 있는 버전은 1.2 이다. 조만간 GAPI 는 마이크로소프트의 모바일 시장 전략인 Windows CE .NET 을 기반으로 하는 PPC Phone Edition 과 Smart Phone 의 .NET Compact Framework 구성에 의해 Direct X 의 형태로 Visual Studio .NET 의 확장 라이브러리의 일부가 되어 제공될 것이다.

GAPI 는 그리 많지 않은 한계를 가지고있다. GAPI 에서 BitBlt 또는 TransparentImage 에 대한 지원을 지원하지 않는 다는 것이다. 그러나 미래에는 해결되리라 본다. GAPI 는 외부 블리팅 루틴을 제공한다.

■ Window GDI vs Game API

아래 그림은 GDI 와 Game API 가 그림을 화면에 디스플레이하는 구조를 보여주고 있다. 그림에서 보듯이 GAPI 는 Device Context 를 통하지 않고 직접 비디오 메모리를 접근하기 때문에 속도의 개선을 이룰 수 있다.



2. GAPI 의 구조체

GAPI 는 게임 라이브러리가 사용하는 3 개의 구조체를 가진다.

■ GXDisplayProperties

GXDisplayProperties 구조체는 PPC 상의 비디오 프레임 버퍼에 관한 정보를 제공한다. 이 구조체는 실제 디스플레이 속성을 알아내기 위해 GXDisplayProperties 로 넘겨진다.

```
struct GXDisplayProperties {
    DWORD cxWidth;
    DWORD cyHeight;          // notice lack of 'th' in the word height.
    long cbxPitch;    // number of bytes to move right one x pixel
                      //- can be negative.
    long cbyPitch;    // number of bytes to move down one y pixel
                      //- can be negative.
    long cBPP;        // # of bits in each pixel
```

```
    DWORD ffFormat;          // format flags.
};
```

cxWidth 와 cyHeight 는 스크린의 폭과 높이이며, 또한 SystemParametersInfo 호출로 알아낼 수 있다. CbxPitchdhk cbyPitch 는 칼라 피치 변수인 cBpp 에다라 픽셀이 비디오 메모리에 어떻게 배치될 것인지를 결정하는 아주 중요한 값이다.

ffFormat 은 특정한 비트 깊이를 리턴한다. 이 값은 비디오 메모리에서 연결된 픽셀이 어떻게 설정되어야 하는지를 결정한다. 비트깊이에 적당한 ffFormat 값은 gx.h 파일에 아래처럼 정의되어 있다.

```
#define kfLandscape   0x8     // Screen is rotated 270 degrees
#define kfPalette     0x10    // Pixel values are indexes into a palette
#define kfDirect      0x20    // Pixel values contain
                              // actual level information
#define kfDirect555   0x40    // 5 bits each for red, green
                              // and blue values in a pixel.
#define kfDirect565   0x80    // 5 red bits, 6 green bits
                              // and 5 blue bits per pixel
#define kfDirect888   0x100   // 8 bits each for red, green
                              // and blue values in a pixel.
#define kfDirect444   0x200   // 4 red, 4 green, 4 blue
#define kfDirectInverted 0x400
```

■ GXKeyList

GXKeyList 구조체는 디폴트로 할당된 하드웨어 버튼에 관한 정보를 제공한다. PPC 의 일반 적인 버튼은 UP, DOWN, LEFT, RIGHT, 버튼으로 구성되어 있다. GXKeyList 변수는 버튼 값을 구하기 위해 GXGetDefaultKeys 함수로 전달한다. GAPI 라이브러리에 있는 모든 함수와 같이 이 구조체는 gx.h 헤더화일에 포함되어 있다.

```
struct GXKeyList {
    short vkUp;       // key for up
    POINT ptUp;       // x,y position of key/button.
                      // Not on screen but in screen coordinates.
    short vkDown;
    POINT ptDown;
```

```
    short vkLeft;
    POINT ptLeft;
    short vkRight;
    POINT ptRight;
    short vkA;
    POINT ptA;
    short vkB;
    POINT ptB;
    short vkC;
    POINT ptC;
    short vkStart;
    POINT ptStart;
};
```

위에서 살펴보면, 각각의 버튼은 그와 관련한 POINT 변수를 가진다. 이 POINT 변수는 버튼의 물리적인 X, Y 좌표값을 저장한다. 버튼은 WM_KEYDOWN 이벤트처럼 WndProc 로 보내진다. WParam 을 사용하는 WndProc 에서 각 버튼을 검사할 수 있다.

```
case WM_KEYDOWN:
    Virtual_Key = (short) wParam;
    if(Virtural_Key == gx.KeyList.vkUp) {
       // "UP" 버튼이 눌렸다.
          break;
    }
```

■ GXScreenRect

GXScreenRect 구조체는 스크린 rect 에 관한 정보를 가지고 있으며, 새로운 윈도우의 clip 영역을 지정하는데 사용된다. GXScreenRect 은 어떠한 GAPI 도 사용하지 않으며, 보조함수 또는 GAPI 1.2 에 포함된 함수가 사용하는 것 같다. 보조함수(실제 프로그래머가 호출할 수 없는 함수) 그리 걱정하지 않아도 될 것이다. 참고로 이 구조체는 아래와 같다.

```
struct GXScreenRect {
    DWORD dwTop;
    DWORD dwLeft;
    DWORD dwWidth;
    DWORD dwHeight;
```

};

3. GAPI 함수

GAPI 는 12 개의 함수로 구성되어 있으며, 함수명이 GX 로 시작해서 위도우 CE API 에서 쉽게 구별할 수 있다. GAPI 가 할 수 있는 범위는 비디오 직접제어, 하드웨어 버튼제어, 일시 중지/재실행 기능 등이 있다. 헤더화일에서 각각의 GAPI 함수는 GXDLL_API 보다 밑에 있으며, GXDLL_API 는 DLL 이 사용하는 단순한 export symbol 이다. GAPI 함수는 다음과 같다.

- GXGetDisplayProperties

  GXGetDisplayProperties 는 GXDisplayProperties 형으로 선언된 변수를 리턴하며, PPC 의 디스플레이 하드웨어에 관한 세세한 정보를 가지고 있다.

  GXDisplayProperties GXGetDisplayProperties();

  GXDisplayProperties 의 구조체는 이미 앞에 나와 있다. 일반적으로 다음과 같이 호출된다.

  GXDisplayProperties gxdp;
  gxdp = GXGetDisplayProperties();
  int iScreenWidth = gxdp.cxWidth;
  int iScreenHeight = gxdp.cxHeight;

- GXGetDefaultKeys

  GXGetDefaultKeyssms 특정한 PPC 장치에서 게임용으로 가장 적합한 하드웨어 버튼용 가상 키코드의 리스트를 리턴한다. 이 값은 GXKeyList 구조체에 정의된 변수로 리턴된다.

  GXKeyList GXGetDefaultKeys(int iOptions);

  iOption 은 버튼이 portrait(GX_NORMALKEYS) 또는 landscape(GX_LANDSCAPEKEY)로 리턴 될 것인지를 결정하는 값이다. 일반적으로 다음과 같이 호출된다.

  GXKeylist gxkl;
  gxkl = GXGetDefaultKeys(GX_NORMALKEYS);
  gxkl 안의 키코드는 GXKeyList 의 샘플코드에 나와 있듯이 WndProc 의 WM_KEYDOWN 이벤트에서 테스트 된다.

- GXOpenDisplay

  GXOpenDisplay 는 디스플레이를 개방사용하며, 비디오 메모리에 절대모드로 제어할 수 있다. GXOpenDisplay 는 일반적으로 디스플레이를 초기화하고 게임

이 시작될 때 호출이 된다.

```
int GXOpenDisplay(HWND hWnd, DWORD dwFlags);
```

GXOpenDisplaysms 프로그램 윈도우가 생성되고, 디스플레이되자 마자 바로 호출되어야 한다. 일반적으로 InitInstance 함수에 둔다.

```
if(GXOpenDisplay(hWnd, GX_FULLSCREEN) == 0)
{
        // GAPI Initialization failed
        return;
}
```

■ GXCloseDisplay

GXCloseDisplay 함수는 GAPI 가 사용한 디스플레이 리소스를 해제하며, 일반적으로 프로그램이 종료하기 전에 호출된다.

```
int GXCloseDisplay();
```

GXCloseDisplay 는 다음과 같이 GAPI 를 종료하는 WM_DESTROY 에서 호출되어야 한다.

```
case WM_DESTROY:
    GXCloseInput();
    GXCloseDisplay();
    PostQuitMessage(0);
break;
```

■ GXBeginDraw

GXBeginDraw 함수는 디스플레이를 풀스크린으로 준비한다. GXBeginDraw 는 GDI 함수인 BeginPaint 와 유사하나, HDC 를 리턴하는 대신 GXBeginDraw 는 비디오 메모리의 포인터를 리턴한다.

```
void * GXBeginDraw();
```

GXBeginDraw 는 스크린 업데이트가 시작되는 첫 부분에 호출되고, 드로잉 함수가 종료하자 마자 GXEndDraw 로 종료한다. 이것은 일반적으로 이중 버퍼가 신속히 스크린에 블리팅하는 게임루프에 둔다. 그러나 게임에서 직접 출력하는 데도 사용이 된다.

```
void *ptrVideoMemory;
ptrVideoMemory=GXBeginDraw();
if(ptrVideoMemory == NULL) {
      return;
}
```

■ GXEndDraw

GXEndDraw 는 비디오 메모리에 드로잉 작업이 끝날 때 호출이 된다. GXEndDraw 는 GXBeginDraw 함수가 생성한 리소스를 해제하는 데만 호출되어야 한다. GXBeginDraw 와 GXEndDraw 는 항상 한 쌍으로 호출되어야 한다. GXEndDraw 는 호출인자가 없으며, 에러가 발생하지 않는다면 리턴값은 없다.

```
int GXEndDraw();
```

■ GXOpenInput

GXOpenInput 함수는 버튼의 버튼 메시지 핸들링이 실행되게 하고, 버튼입력이 WndProc 의 WM_KEYDOWN 으로 바로 전송이 된다. 이 함수는 프로그램이 종료하기 전에 GXCloseInput 과 한 쌍을 이루어야 한다.

```
int GXOpenInput();
```

■ GXCloseInput

GXCloseInput 함수는 버튼의 입력용으로 GAPI 가 사용한 리소스를 해제하는데 사용이 되고, 다른 디폴트 filtered mode 에 버튼 메시지를 리턴한다. GXCloseInput 은 프로그램이 종료되기 전에 호출되어야 한다.

```
int GXCloseInput();
```

■ GXSuspend

GXSuspend 함수는 GAPI 연산을 일시중지 시키며, WM_KILLFOCUS 와 WM_HIBERNATE 메시지 내에서만 호출해야 한다. GXResum 함수로 다시 실행된다.

```
int GXSuspend();
```

아래와 같이 WndProc 에서 호출한다.

```
case WM_KILLFOCUS:
    GXSuspend();
    break;
```

■ GXResume

GXResume 함수는 이전의 GXSuspend 가 먼저 호출되고, GAPI 연산을 다시 실행하기 위해서 사용되며, WM_SETFOCUS 에서만 호출되어야 하고, 또한 WM_ACTIVE 에서도 호출 가능하다.

```
int GXResume();
```

아래와 같이 WndProc 에서 호출된다.

```
case WM_SETFOCUS:
```

```
GXResume();
break;
```

- GXSetViewport

  GXSetViewport 함수는 GAPI 1.2 에 새로 추가된 함수로 비표준 PPC 디스플레이장치에 대한 GAPI 뷰포트를 설정하는데 사용된다. GXSetViewport 는 대부분의 경우에 사용되지 않는다.

```
int GXSetViewport( DWORD dwTop,
                   DWORD dwHeight,
                   DWORD dwReserved1,
                   DWORD dwReserved2 );
```

  dwTop 은 스크린에서 뷰포트의 맨 위쪽 위치이고, dwHeight 는 픽셀로 뷰포트의 높이이다. dwReserved1 과 dwReserved2 는 GAPI 1.2 에서 사용되지 않으며, 필히 0 으로 설정한다. GXSetViewport 는 어떠한 클리핑을 하지 않는다. 향유에 뷰포트의 left 와 width 가 될 거라는 가정을 할 수도 있다. 대부분의 경우 이 함수는 PPC 에서 아무런 효과를 미치지 않는다.

- GXIsDisplayDRAMBuffer

  GXIsDisplayDRAMBuffer 함수는 GAPI 1.2 에 새로 추가된 함수로 PPC 가 비표준 디스플레이를 가졌는지를 알아보는데 사용한다. GXIsDisplayDRAMBuffer 는 비표준 디스플레이를 다루기 위해 일반적으로 GXSetViewport 와 함께 사용된다.

```
BOOL GXIsDisplayDRAMBuffer();
```

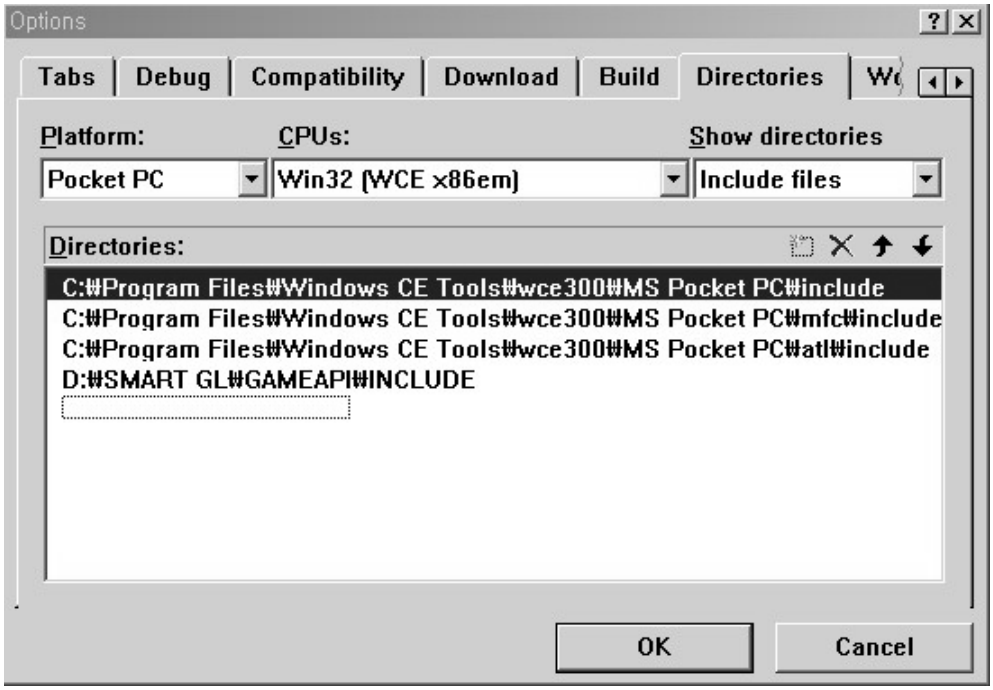
4. GAPI 사용법

- 최신 API 사용하기

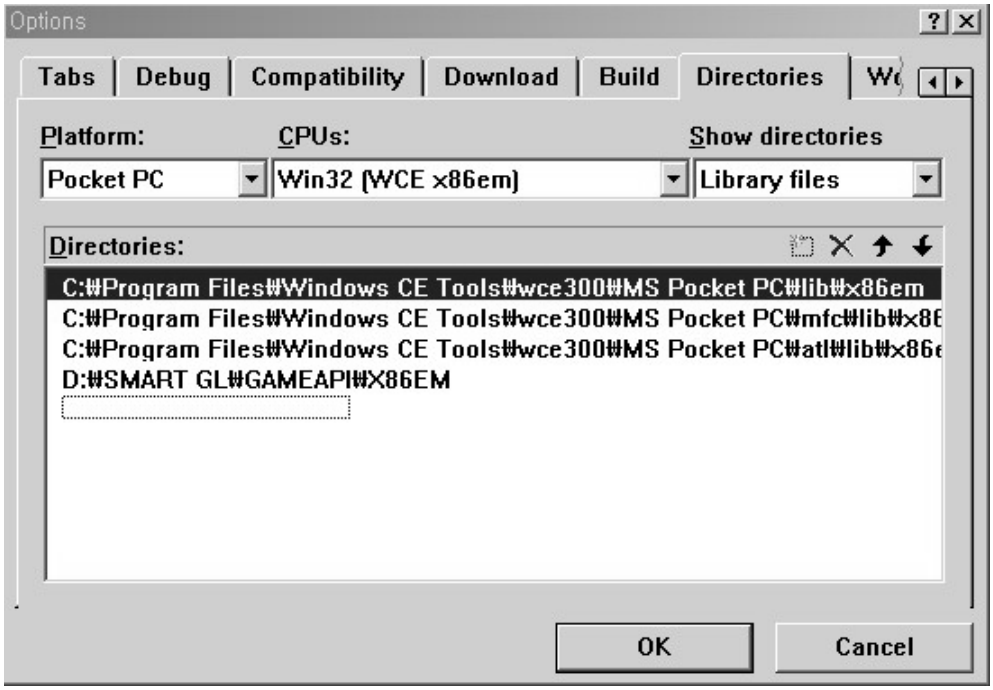
  Game API 는 현재 버전 1.2 까지 나와 있으며, 다운로드 받기 위해서는 Microsoft 의 개발자 홈페이지에서 (http://www.microsoft.com/downloads/details.aspx?FamilyID=d9879b0e-4ef1-4049-9c61-e758933d84c4&DisplayLang=en)에서 다운로드 받을 수 있다. 다운받은 파일의 압축을 풀어 gx.dll 은 PPC 의 window 폴더로 복사해야한다.

  그리고 gx.lib 는 eMbedded Visual Tool 에 위치를 알려주어야 하고 include 파일인 gx.h 도 위치를 알려주어야 한다. 각각의 위치를 등록하는 방법은 아래 그림과 같다. 그림은 에뮬레이터를 사용하는 경우로 예를 들었지만 PPC 의 해당 CPU 에 대해서도 동일한 방법을 사용할 수 있다. 아래와 같이 설정하면

#include <gx.h>로 헤더파일을 프로그램에 포함시키면 GAPI 함수를 프로그램에서 사용할 수 있다.

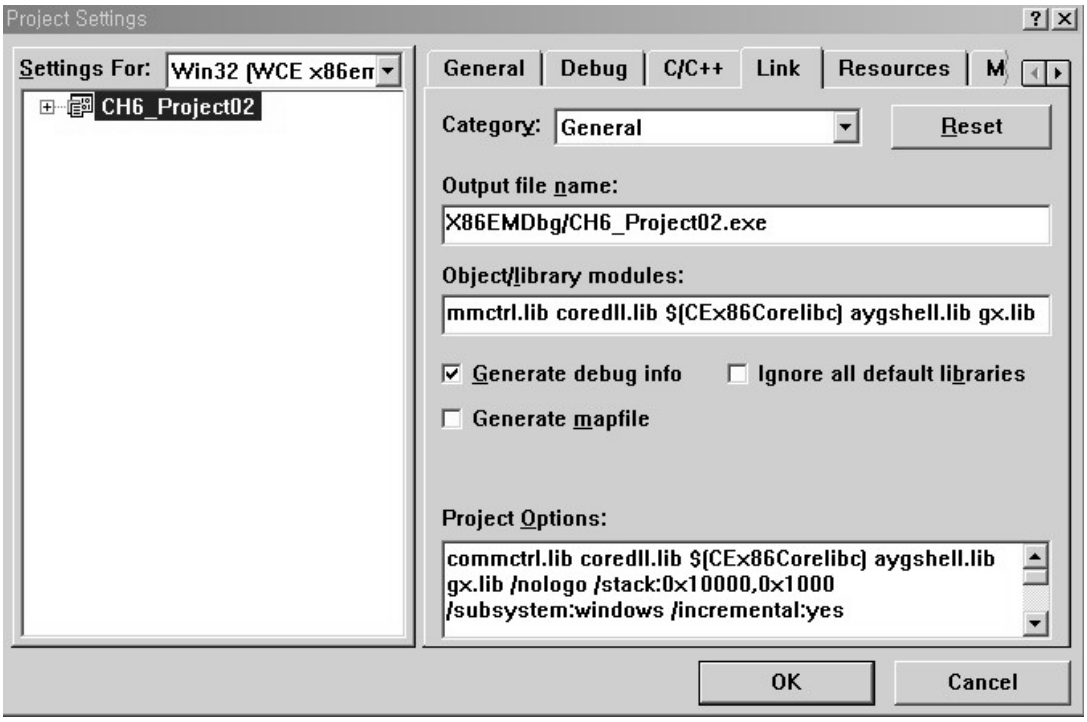


<Include 파일의 위치를 등록시키는 방법>



<라이브러리 파일의 위치를 등록시키는 방법>

아래 그림은 eMbedded Visual Tool 에 GAPI 라이브러리 함수를 사용하여 실행프로그램을 생성하도록 옵션을 설정하는 화면이다. 그림을 살펴보면 Object/library modules 의 마지막 항목에 gx.lib 가 있는 것을 발견할 수 있다.



■ Game API 게임 루프

GAPI 게임은 일반적인 WinMain 함수와 WndProc 함수와 같은 일반적인 윈도우 함수를 가지고 있으나, 직접 비디오메모리를 제어하기 위해 스크린을 구성하거나 직접 하드웨어 버튼을 처리하기위해 몇 가지의 GAPI 함수를 호출할 필요가 있다. 일반적인 GAPI 게임 루프는 다음과 같다.

1. GXOpenDisplay
2. GXOpenInput
3. GXGetDisplayProperties
4. GXGetDefaultKeys

일반적으로 타이머가 실행되든가 또는 메일 게임루프가 WinMain 에서 메시지로 처리될 수 있다. 비디오 메모리에 작업할 시간이 되면 일반적으로 시간이 정해진 에니메이션 루프에서는 처리과정이 아래와 비슷할 것이다.

5. GXBeginDraw
6. 스크린 출력작업
7. GXEndDraw
8. 게임내의 변수 업데이트
9. #5 로 루프를 되돌린다.

프로그램이 종료해야할 시점이면 아래의 순서를 따른다.

10. GXCloseInput

11 GXCloseDisplay

■ 스크린에서 픽셀 그리기

그래픽 디스플레이의 기본적인 단위는 픽셀이다. GAPI 는 직접 비디오메모리를 사용하며, 하나의 큰 바이트배열을 제공한다. 비디오 메모리의 각 바이트는 디스플레이의 bit depth 에 따라 픽셀 일부분, 전체 픽셀, 몇 개의 픽셀을 가질 수 있다.

■ GAPI 비디오 모드

PPC 에서 다음의 세가지 bit depth 가 일반적이다.

1) 픽셀당 4 비트
2) 픽셀당 8 비트
3) 픽셀당 16 비트

4 BPP 디스플레이는 현재 드물다. 게임 프로그램에서 가능한 많은 사용자를 지원하게 될 것이다. 그러나 4BPP 는 사용하지 않는 게 좋다. 많은 사용자층을 확보하고 있는 16 비트, 또는 그레이스케일을 지원하려면 특별한 노력을 기울여야 한다.

8 비트도 마찬가지로 PPC 게임이 거의 없다. 8 비트는 이전의 256 칼라, MS-DOS 의 인터럽트 모드 13H 디스플레이와 비슷하다. 이들은 정확히 1 바이트의 비디오 메모리를 사용한다. 또한 아주 빠른 블리팅이 되며, 코드를 작성하기 쉽다. 불행히도 GAPI 는 비디오 모드를 변경할 수 있는 방법을 제공하지 못한다. PPC 디스플레이는 오직 하나의 디스플레이 모드만을 지원하지만 모든 PPC 가 동일한 디스플레이 하드웨어를 가지지 않는다.

16 비트 디스플레이는 65536 칼라를 처리할 수 있다. 비디오 메모리에 하나의 픽셀을 표현하는데 2 바이트를 사용한다. 대체로 PPC 에서 가장 일반적인 비디오 시스템은 16 비트이다.

■ GAPI 비디오 프레임버퍼

게임 프로그램에 표준 Windows CE 의 GDI 대신 Game API 를 사용하는데 따른 실질적인 이점은 비디오 프레임 버퍼(비디오 메모리)를 직접 제어할 수 있다는 것이다. 비디오 메모리는 선형 바이트 배열로 구성되며, 스크린에 픽셀을 채우기 위해 비디오 하드웨어 시스템이 직접 사용한다. PPC 상에서 스크린은

수평으로 240 개의 픽셀과 수직으로 320 개의 픽셀로 구성되어 있다.

16 비트 디스플레이에서 하나의 픽셀에 대한 공식은 아래와 같다.

Pixel_Add = (X_Pos * X_Pitch) + (Y_Pos * Y_Pitch);

X_Pitch 와 Y_Pitch 는 GXGetDisplayProperties 를 사용해 알아낼 수 있다. 실제 변수는 cbxPitch 와 cbyPitch 이다. 픽셀의 비디오 메모리에 인텍스가 있는 경우는 각 픽셀에 사용되는 비트 포맷을 따라야 한다. (Windows GDI 는 이 과정을 자동으로 처리하며, 이것이 GDI 를 사용하는 이유이다) 이 공식을 상용해 다음의 코드로 픽셀을 그리는데 사용할 수 있다.

```
int DrawPixel(unsigned char *VidMem, int X, int Y, int color) {
      int address = (X * X_Pitch) + (Y * Y_Pitch);
      *(VidMem + address) = color;
      return 0;
}
```

불행히도 칼라인자에 문제가 있다. 그래서 수정을 하지 않으면 이 함수는 작동하지 않는다. 8 비트 디스플레이는 팔레트가 실제 칼라값을 가지고 있기 때문에 단순하다. 그래서 한의 픽셀을 설정하기 위해 비디오 메모리에 한 바이트만 저장하면 된다. 그러나 대부분의 PPC 는 16 비트 디스플레이를 사용한다. 3 개의 RGB 값을 저장하기 위해 어떻게 2 바이트 안에 집어 넣을 수 있는가? GXGetDisplayProperties 함수를 사용할 수 있다. ffFormat 이라는 GXGetDisplayProperties 구조체의 멤버가 있다. 16 비트 포맷은 다음과 같이 2 개이다.

kfDirect555

kfDirect565

555 디스플레이는 거의 사용하지 않는다. 대부분의 PPC 가 사용하는 565 디스플레이에 대해 알아본다. 각 칼라에 대한 red, green, blue 값이 디스플레이를 나타내는 포맷으로 압축되었다. 565 포맷은 5 비트 red, 6 비트 green, 5 비트 blue 를 나타낸다. 하나의 565 픽셀에 대한 칼라는 다음으로 나타낸다.

```
unsigned short color;
Color = (unsigned short)
  (((color.Red & 0xf8) << 8) | ((color.Green & 0xfc) << 3) |
  ((color.Blue & 0xf8) >> 8));
```

■ 시프트 연산

칼라값 지정에서 두개의 left shift 비트 연산과 하나의 right shift 비트연산을 한다. 비트를 left shift 하는 것은 곱하기 연산에 가장 빠른 방법이고 right

shift 하는 것은 나누기 연산에 가장 빠른 방법이다.

left shift 연산은 CPU 에서 처리가 되며, 종종 하나의 어셈블리 명령으로도 처리가 되고, 비디오 메모리에 기록 또는 루프가 많은 연산에서 속도를 아주 많이 증가시킨다. 예를 들어 비디오 메모리에서 픽셀을 구하기 위해 사용될 수 있는 다음의 연산을 고려해보자

Result = Y * 240 + X;

left shift 연산을 사용하면 다음과 같이 된다.

Result = (Y << 7) + (Y << 6) + (Y << 5) + (Y << 4) + X;

위의 코드는 다음과 동일한 내용이다.

Result = (Y << 128) + (Y << 64) + (Y << 32) + (Y << 16) + X;

이러한 shift 연산을 사용함으로서 어떠한 부분에서는 덧셈이 곱셈 또는 나눗셈보다 따른다. shift 연산과 문장을 추가하는 것이 7 개 또는 8 개의 명령어보다 많다면 원래 하던 연산을 그대로 하는 게 좋을 것이다. 이는 프로세서가 그 점에 있어서는 최적화를 하기 때문이다. 240 은 4 가지의 인자를 필요로 하기 때문에 처리하기 어렵다. 많은 수치를 이보다 작제 만들 수 있다. 특히 320 같이 8 로 나눌 수 있는 숫자의 경우에 적합하다.

Result = Y * 320 + X;

다음과 같이 달리 표현할 수 있다.

Result = (Y << 8) + (Y << 6) + X;

보듯이 일부의 경우에 수치는 아주 잘 처리된다. 이 특별한 시프트 연산은 VGA 모드 13h 디스플레이에서 일반적으로 사용되었다. 이는 수평해상도가 320 이었기 때문이다.

대부분의 PPC 프로세서가 RISC 구조를 가지고 있다는 것을 고려할 때 보다더 큰 수준의 shift 연산의 이점을 보여준다. 이것은 프로세서의 핵심이 어셈블리 명령을 적은 단위로 나누며, 종종 수평적으로 처리된다. 많은 연산이 필요한 픽셀 출력루프에서 하나의 복잡한 수학문장을 세분화 시키는 것은 함수의 속도를 몇 배 증가 시킨다.

복잡한 루프 안에 if 문과 같은 조건문을 사용하지 않도록 한다. 시프트 연산으로 증가시킨 속도를 무용지물로 만든다. 비록 디스플레이 루프가 여러 가지 조건을 처리하기 위한 몇 가지 방법을 가지고 있더라도 디스플레이 루프는 아주 복잡하다.

■ 픽셀 출력

적어도 PPC 에서 비디오 메모리가 어떻게 작동되는지 대략은 알게 되었으므로 몇 개의 픽셀을 그려보도록 한다. 다음은 완전한 16 비트 DrawPixel 루틴이다.

```
int DrawPixel16(unsigned char *VidMem, int x, int y, COLOR color)
{
    int address = (X * g_gxdp.cbxPitch) + (Y * g_gxdp.cbyPitch)
    unsigned short usColor;

    usColor = (unsigned short)
              (((color.Red & 0xf8)<<8)|
               ((color.Green & 0xfc)<<3) |
               ((color.Blue &  0xf8)>>3) );
    *(unsigned short *)(VidMem +  address) = usColor;
    return 0;
}
```

RGB 칼라의 구성요소는 바이트이다. 그래서 칼라를 손쉽게 565 포맷으로 전환할 수 있다. DrawPixel16 함수의 첫번째 부분은 인자로 넘겨받은 X, Y 값을 기본으로 픽셀의 주소를 계산한다. 이것이 최적화된 샘플이다. 그러나 보다 최적화하는 방법은 픽셀을 그리는 함수를 루프 속으로 코드를 옮기고 함수를 호출하지 않는 것이다. g_gxdp 는 GXDisplayProperties 구조체에 선언된 전연 변수이다. 이 함수가 메인 게임 루프에서 몇 천번 호출될 것이기 때문에 부하가 걸릴 수 있다. 이 구조체는 이 함수를 보다 최적화 하는 데 필요한 첫번째의 것이다.

두번째 최적화는 구조체에서 참조되는 변수이다. (cbxPitch 와 cbyPitch) 이 값은 한 PPC 비디오 시스템에서 다른 비디오 시스템에서도 동일할 것이다. 비록 각각의 비디오 디스플레이 시스템용으로 여러 가지 DrawPixel16 버전이 필요하더라도 결과는 가치 있는 노력이 될 것이다.

DrawPixel16 의 다른 최적화는 해당하는 것은 칼라가 비디오 메모리 조직에 맞게 설정된 것이다. 두 바이트 값이 직접 비디오 메모리에 기록된다. COLOR 구조체는 다음과 같이 함수의 인자로 넘겨진다.

COLOR 구조체의 각 칼라 값은 BYTE 또는 unsigned char 형이므로 이 코드는 비디오 메모리 포인터 VidMem 과 함께 잘 실행된다.