3.OpenGL의 기초

3-1.기본 타입

OpenGL 자체는 C 언어로 제작되었지만 함수 수준의 라이브러리이므로 특정 언어에 종속되지는 않는다. 그래서 타입도 독립적으로 정의하여 사용한다. 비록 C 언어의 타입과 거의 유사하지만 차후 C 언어의 타입이 달라질 수도 있고 플랫폼에 따라 약간씩 차이도 있으므로 영향을 최소화하기 위해 고유의 타입을 정의해 두었다.

OpenGL 타입	C 타입	접두어
GLbyte	signed char	b
GLshort	short	s
GLint, GLsizei	int, long	i
GLfloat, GLclampf	float	f
GLdouble, GLclampd	double	d
GLubyte, GLboolean	unsigned char	ub
GLushort	unsigned short	us
GLuint, GLenum, GLbitfield	unsigned long	ui

대부분은 C타입에 대한 별명으로 정의되어 있어 C 언어에 익숙한 사람은 이 타입들에 대해서도 쉽게 익숙해질 것이다. C 타입명앞에 GL 접두어를 붙인 정도에 불과하다. 그러나 상식과 다른 부분이 있으므로 몇 가지 주의 사항을 잘 알아 두어야 한다.

정수형인 GLint는 32비트의 정수로 정의되어 있으며 길이가 고정적이다. C 언어에서 int는 운영체제에 따라 달라지는 가변타입이지만 OpenGL의 GLint는 항상 32비트이다. GLsizei는 GLint와 같은 타입이지만 크기값을 나타내는 값에 주로 사용된다. 이런 식으로 같은 타입에 대해서도 용도에 따라 별칭이 정의되어 있다.

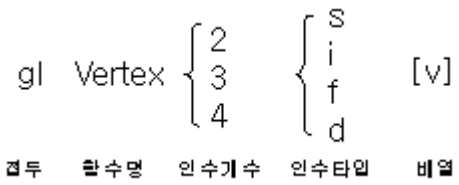
진위형을 나타내는 GLboolean은 부호없는 1바이트 정수로 정의되어 있다. C 언어는 4바이트의 BOOLEAN 열거형을 정의하고 진위형 상수로 TRUE, FALSE를 사용하는데 크기가 다르지만 같은 정수형이므로 이 타입과 호환된다. TRUE, FALSE를 사용할 수도 있고 GL_TRUE, GL_FALSE를 사용할 수도 있되 둘 다 정의는 동일하다. 그러나 C++의 bool 타입과는 호환성이 좋지 않으므로 섞어서 사용하지 않는 것이 좋다. 에러는 나지 않지만 경고가 발생하며 잠재적인 문제가 될 수도 있다.

실수 타입인 GLclampf, GLclampd의 clamp라는 단어의 의미는 값의 범위가 0.0 ~ 1.0 사이로 제한된다는 뜻이다. 색상의 강도나 텍스처의 좌표처럼 범위가 명확한 값을 지정할 때 이 타입을 흔히 사용한다. 타입명에 clamp가 들어 있으면 유효한 값은 0 ~ 1 사이임을 바로 알 수 있다.

3-2.OpenGL의 함수 형식

OpenGL 함수들은 일반 함수와 구분하기 위해 예외없이 접두어가 붙는다. gl 라이브러리의 함수는 gl 접두어로 시작하고 glu의 라이브러리는 glu로 시작한다. 마찬가지로 glut 소속의 함수는 glut 접두가 붙는다. 접두만으로도 OpenGL 함수임을 쉽게 알 수 있다.

OpenGL은 C 수준의 함수들로 구성되어 있으므로 객체 지향과는 거리가 멀다. 그래서 함수 오버로딩(Overloading)을 지원하지 않으며 인수의 개수와 타입이 다른 동명 함수를 작성할 수 없다. 동일한 동작을 하되 인수 목록이 다르면 함수의 이름도 매번 달라져야 한다. 대표적인 예로 정점의 좌표를 정의하는 glVertex 함수를 보자.



이 함수는 GL 라이브러리 소속이므로 접두로 gl이 붙으며 정점을 정의하므로 함수의 이름은 Vertex이다. 함수명 뒤에는 인수의 개수와 타입을 명시하는 접미어가 각각 3종류, 4종류가 붙는다. 3차원상의 좌표는 x, y, z 세가지 값으로 표현하는 것이 원칙적이되 분수 표현을 위해 w로 분모를 지정할 수 있다. 또 평면상의 정점인 경우는 z 좌표를 생략하고 x, y만 밝힐 수도 있다. w가 생략되면 1로 간주되며 z를 생략하면 0으로 간주한다. 인수의 개수는 2개, 3개, 4개 세 가지 종류가 있다.

각 인수의 타입은 GLshort, GLint, GLfloat, GLdouble 4가지 타입으로 전달 가능하다. 보통은 실수값인 GLfloat를 많이 사용하지만 소수부가 필요없다면 GLint 타입을 사용할 수도 있고 더 높은 정밀도가 요구되면 GLdouble 타입을 쓸 수도 있다. 평면상에 삼각형을 그리는 첫 예제에서는 2개의 실수로 좌표를 지정했으므로 glVertex2f 함수를 호출했다. 이 함수명은 GL 라이브러리에 속한 Vertex 함수이며 인수는 2개이고 인수 타입은 GLfloat를 취한다는 뜻이다.

또 좌표를 구성하는 값들을 개별 인수로 전달하는 대신 배열로 전달할 수도 있다. 이때는 함수명 제일 뒤에 배열(Vector)을 의미하는 v 접두어가 추가로 붙는다. 다음 두 호출문은 동일하다. 세 개의 정수를 glVertex3i 함수의 인수 목록에 나열할 수도 있고 세 정수를 가지는 배열을 정의한 후 glVertex3iv 함수로 배열을 전달할 수도 있다. 전달 방식만 다를 뿐 결국 지정하는 좌표는 동일하다.

```
glVertex3i(1,2,3);
int arv[]={1,2,3}; glVertex3iv(arv);
```

개수, 타입, 배열 여부의 모든 조합을 취하면 glVertex 함수만 해도 3 * 4 * 2 = 24개의 함수 그룹이 정의된다. 24개나 되는 함수의 원형을 일일이 밝힐 수는 없으며 또한 그럴 필요도 없다. 그래서 다음과 같이 간략하게 표기한다.

glVertex[2,3,4][s,i,f,d][v](x,y,z,w)

[ ] 괄호안의 문자들은 선택적이라는 뜻이다. 인수의 타입도 함수명에 따라 가변적으로 달라지므로 인수 목록에는 이름만 밝히고 타입은 적지 않는다. 또 인수의 개수도 함수에 따라 달라지므로 최대 개수의 인수 목록을 적어 놓았다. 함수를 칭할 때는 접미를 제외하고 통칭하여 함수의 이름만 밝히는 것이 보통이다. glVertex 뿐만 아니라 다른 함수들도 마찬가지이다. 특정 함수명을 인터넷으로 검색할 때도 접미는 빼고 검색하는 것이 바람직하다.

OpenGL이 오버로딩을 지원하지 않는 이유는 객체 지향 기법이 일반화되기 전에 작성했기 때문이다. 그러나 속도 최적화를 위해 객체 지향을 선택하지 않은 면도 있다. 원론적으로 객체지향의 C++은 절차적인 C언어보다 작고 빠를 수 없기 때문이다. 최적의 속도를 요하는 그래픽 라이브러리로서 합당한 선택이기는 하지만 함수명에 대한 오버로딩 미지원은 살짝쿵 아쉬운 부분이다.

3-3.색상 변경

이제 예제를 조금씩 변경해 보면서 OpenGL 함수를 하나 둘씩 익혀 보자. 윈도우의 배경색상이 칙칙한 검정색으로 되어 있는데 디폴트가 검정색이기 때문이다. 다음 함수로 배경색을 바꿔 보자.

void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);

4개의 인수는 RGBA 각 색상 요소의 강도를 지정한다. 각 요소의 강도는 0 ~ 1 사이의 실수값이며 0이면 해당 요소가 하나도 없는 것이고 1이면 최대 밝기이다. 타입명에 clamp가 들어간 인수는 모두 0 ~ 1 사이의 범위를 가진다. (1,1,1,1)은 모든 요소가 최대 밝기이므로 흰색이고 (0,0,0,0)은 검정색이다.

투명도를 지정하는 알파값은 블랜딩 등에 활용되는데 당장은 사용되지 않는다. 이 함수는 이후부터 배경으로 사용할 색상을 지정하기만 할 뿐 실제 배경을 지우지는 않는다. 배경을 실제로 지울 때는 다음 함수를 호출한다.

void glClear(GLbitfield mask);

인수로 어떤 버퍼를 지울 것인가를 지정한다. OR 연산자로 연결하여 두 개 이상의 버퍼를 한꺼번에 지울 수도 있다. 버퍼를 지우는 것은 대단히 대용량의 쓰기 동작이라 시간이 오래 걸린다. 두 개 이상의 버퍼를 지울 때는 가급적이면 OR 연산자로 한꺼번에 지우는 것이 속도상 유리하다.

버퍼	설명
GL_COLOR_BUFFER_BIT	색상 버퍼를 지운다.
GL_DEPTH_BUFFER_BIT	깊이 버퍼를 지운다.
GL_STENCIL_BUFFER_BIT	스텐실 버퍼를 비운다.
GL_ACCUM_BUFFER_BIT	누적 버퍼를 비운다.

그래픽 정보들은 여러 버퍼에 나누어 저장된다. 색상 버퍼는 이미지의 색상값이 저장되는 버퍼이며 여기에 있는 정보들이 모니터에 출력된다. 우리가 흔히 비디오 램이라고 부르는 영역이며 픽셀 버퍼, 래스터 버퍼라고도 부른다. 픽셀 버퍼라는 말은 화소의 정보가 저장되었다는 뜻인데 각 화소는 색상의 조합으로 구성되므로 색상 버퍼라고도 부르는 것이다.

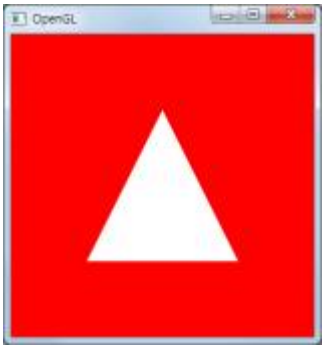
glClearColor로 지정해 놓은 색으로 색상 버퍼를 가득 채우면 이것이 곧 화면을 지우는 것이 된다. 나머지 버퍼들은 좀 더 복잡한 정보를 저장하는데 현재 단계에서는 설명하기 어려우므로 차후 따로 연구해 보기로 하자. DoDisplay에서 뭔가를 출력할 때는 이전 영상을 제거하기 위해 glClear로 화면을 먼저 지워야 하며 그래서 항상 glClear문이 선두에 온다. 다음 예제는 배경색을 빨간색으로 변경한다.

ChangeState

```
void DoDisplay()
{
	glClearColor(1.0, 0.0, 0.0, 1.0);
	glClear(GL_COLOR_BUFFER_BIT);

	glBegin(GL_TRIANGLES);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.5, -0.5);
	glVertex2f(0.5, -0.5);
	glEnd();
	glFlush();
}
```

glClear를 호출하기 전에 glClearColor를 호출하여 배경색상을 빨간색으로 바꾸어 두었다. R 요소만 최대치이고 나머지는 0이므로 빨간색이며 삼각형 뒤쪽의 배경이 빨간색으로 바뀔 것이다.



배경 색상은 한번 지정해 놓으면 다른 값으로 바꾸지 않는 한은 계속 유지된다. 그래서 매번 그리기를 할 때마다 지정할 필요없이 프로그램 초기화시에 별도의 함수에서 따로 설정하는 것이 원칙적이다. 구조상으로는 다음 코드가 더 바람직하다.

```
int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
	glutCreateWindow("OpenGL");
	glutDisplayFunc(DoDisplay);
	DoInit();
	glutMainLoop();
	return 0;
}

void DoInit()
{
	glClearColor(1.0, 0.0, 0.0, 1.0);
}

void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glBegin(GL_TRIANGLES);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.5, -0.5);
	glVertex2f(0.5, -0.5);
	glEnd();
	glFlush();
}
```

초기화만 하는 DoInit 함수를 따로 만들고 메인에서 메시지 루프에 들어가기 전에 한번만 초기화한다. 결과는 같지만 DoDisplay는 윈도우 크기가 바뀌거나 출력 내용이 바뀔 때마다 자주 호출되므로 초기화 시점으로는 적당하지 않다. 불필요하게 같은 배경색을 매번 지정할 필요는 없는 것이다.

이 예제처럼 하는 것이 원칙이다. 그러나 함수를 따로 만들어야 된다는 면에서 너무 번거롭고 코드를 볼 때도 두 함수를 같이 봐야 하므로 정신 사납다. 그래서 그냥 DoDisplay 선두에서 초기화하는 형식으로 예제를 작성했으며 앞으로도 그럴 것이다. 가급적 한 함수내에 코드를 집중시켜 위에서 아래로 코드를 읽음으로써 효과를 살펴볼 수 있도록 했다.

이번에는 배경색이 아닌 삼각형의 색상을 바꾸어 보자. 도형의 색상은 다음 함수로 지정한다. 정확하게는 도형의 색상이 아닌 정점의 색상을 변경하는데 정점의 색상이 결과적으로 도형의 색상이 된다.

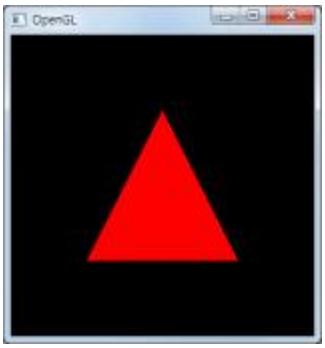
glColor[3,4][b,s,i,f,d,ub,us,ui][v](red, green, blue, alpha)

세가지 색상 요소의 강도를 실수로 지정하는 glColor3f가 가장 일반적이다. 0 ~ 255까지의 정수로 강도를 지정할 때는 glColor3ub 함수도 종종 사용되는데 윈도우즈나 웹 환경의 색상 포맷과 동일해서 친숙하다. 다음은 빨간색 삼각형을 그린다.

```
void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glColor3f(1.0, 0.0, 0.0);
	glBegin(GL_TRIANGLES);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.5, -0.5);
	glVertex2f(0.5, -0.5);
	glEnd();
	glFlush();
}
```

삼각형을 그리기 전에 정점의 색을 빨간색으로 지정했으므로 모든 정점은 빨간색으로 찍히며 이 정점들로 구성된 삼각형 내부도 빨간색으로 채색된다. glColor3ub(255, 0, 0);로 지정해도 결과는 동일하다.

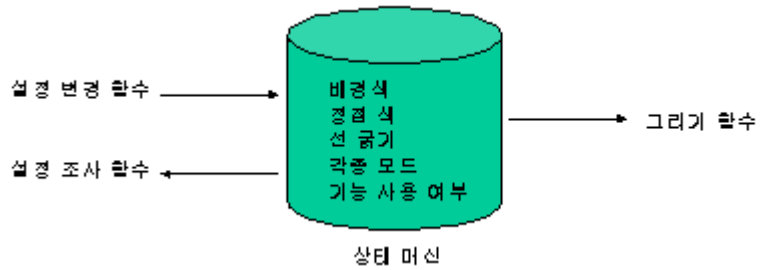


참고로 앞에서도 설명했지만 이 예제도 ChangeState 예제에 같이 작성되어 있다. 팝업 메뉴를 열어서 예제를 바꾸면 출력 내용도 바뀔 것이다. 이후의 예제도 유사한 방식으로 출력 내용이나 주요 출력값들을 변경해 본다.

3-4.상태 머신

그래픽 출력에는 좌표뿐만 아니라 아주 많은 요소들이 개입된다. 색상, 굵기, 모양, 조명, 각종 모드 등등 여러 가지 정보들이 필요하다. 그리기 함수들은 도형 출력을 위해 이 모든 정보들을 참조하지만 그렇다고 해서 이 정보들을 모두 함수의 인수로 전달할 수는 없다. 그렇게 하다가는 인수 목록이 한없이 길어질 것이며 잘 바뀌지도 않는 값을 매번 전달하는 것도 낭비이다.

그래서 OpenGL은 그리기에 필요한 여러 가지 정보들을 상태 머신(State Machine)에 저장한다. 상태 머신이란 상태를 저장하는 장소이며 그리기에 영향을 미치는 여러 변수값들이 집합이다. 앞 항에서 실습해 본 배경 색상은 GL_COLOR_CLEAR_VALUE 상태 변수에 저장되며 현재 정점의 색상은 GL_CURRENT_COLOR 상태 변수에 저장된다.



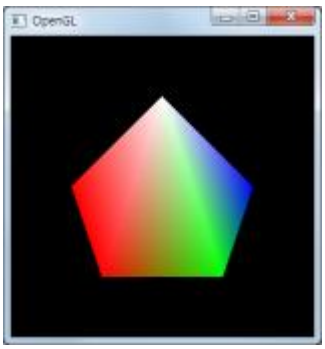
모든 그리기 함수들은 인수로 전달받은 것 외의 정보들에 대해서는 상태 변수값을 읽어 사용한다. 상태 변수들은 적당한 디폴트로 초기화되어 있는데 예를 들어 배경색은 검정색이고 정점의 색은 흰색이다. 디폴트를 바꾸고 싶으면 변경 함수로 언제든지 다른 값으로 바꿀 수 있다. 배경색을 바꾸고 싶으면 glClearColor 함수를 호출하고 정점의 색상은 glColor 함수로 바꾼다.

상태 머신은 전역적이며 영속적인 저장소이므로 한번 지정해 놓은 상태 변수는 다른 값으로 바꾸기 전에는 계속 유효하다. 그래서 같은 값이라면 이전 값을 계속 사용할 수 있으며 매번 새로 지정할 필요가 없다. 다음 예제로 테스트해 보자.

```
void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glBegin(GL_POLYGON);
	//glColor3f(1.0, 1.0, 1.0);
	glVertex2f(0.0, 0.6);
	glColor3f(1.0, 0.0, 0.0);
	glVertex2f(-0.6, 0.0);
	//glColor3f(1.0, 0.0, 0.0);
	glVertex2f(-0.4, -0.6);
	glColor3f(0.0, 1.0, 0.0);
	glVertex2f(0.4, -0.6);
	glColor3f(0.0, 0.0, 1.0);
	glVertex2f(0.6, 0.0);
	glEnd();
	glFlush();
}
```

윗쪽 꼭지점부터 반시계 방향으로 돌며 다섯개의 정점을 지정하여 오각형을 그렸다. 각 정점마다 다양한 색을 지정했는데 정점의 색상이 제각각이면 다각형 내부는 각 정점의 거리에 따라 색상을 부드럽게 섞어서 그린다.



첫번째 정점은 흰색인데 디폴트가 흰색이므로 굳이 흰색으로 지정할 필요가 없다. 코드에서 주석 처리한 glColor 호출문은 있으나 마나한 호출문이다. 두번째 정점은 빨간색으로 바꾼 후 그렸으므로 빨간색이다. 이후부터 정점의 색은 계속 빨간색이 유지되며 별 지정없이 출력한 세번째 정점은 역시 빨간색이다. 이 경우도 정점을 그리기 전에 빨간색으로 바꿀 필요가 없다. 마지막 두 정점은 각각 초록색, 파란색으로 출력했으며 이전 색상과 다르므로 이때는 glColor 호출문이 필요하다.

색상 뿐만 아니라 상태 머신에 저장되어 있는 모든 값이 이런 식으로 관리된다. 상태값은 전역적이므로 함수 내부에서 뿐만 아니라 외부에서 바꾼 값에도 영향을 받으며 어디서 바꾸었건 한번 설정한 값은 다른 값으로 덮어 쓰기 전에는 계속 유효하다.

상태 머신에는 색상처럼 복잡한 정보도 있지만 특정 기능을 사용할 것인가 아닌가를 지정하는 단순한 진위형 옵션들도 많이 있다. on/off 두 가지 값을 가지는 상태는 다음 함수로 설정, 해제, 조사한다.

```
void glEnable(GLenum cap);
void glDisable(GLenum cap);
GLboolean glIsEnabled(GLenum cap);
```

설정값은 열거형으로 지정하는데 수십 가지의 값이 있다. 예를 들어 GL_FOG는 안개 효과이며 GL_LIGHTING은 조명 효과이다. 해당 기능을 쓰고 싶으면 glEnable로 활성화하고 그만 쓰고 싶으면 glDisable로 취소하면 된다. 각 기능에 대해서는 해당 주제를 다룰 때 다시 소개할 것이다. 진위형이 아닌 상태값은 다음 함수들로 조사한다.

```
void glGetIntegerv(GLenum pname, GLint * params);
void glGetFloatv(GLenum pname, GLfloat * params);
void glGetDoublev(GLenum pname, GLdouble * params);
void glGetBooleanv(GLenum pname, GLboolean * params);
```

첫번째 인수는 알고 싶은 상태 변수 값이며 두 번째 인수는 상태값을 리턴받을 배열이다. 상태 변수에 따라 필요한 배열의 크기는 달라진다. 예를 들어 현재 설정된 색상값을 알고 싶다면 다음과 같이 호출한다. 색상은 4개의 실수로 구성되므로 배열 크기도 4로 준비해야 한다.

```
GLfloat arColor[4];
glGetFloatv(GL_CURRENT_COLOR, arColor);
```

원하는 상태대로 출력하려면 상태 변수부터 먼저 설정해야 한다. 이것 저것 많은 상태를 바꾸다 보면 이전 상태로 돌아가기 어려워지는데 이때는 현재 상태의 일부를 스택에 저장해 놓았다가 복구한다. 다음 함수는 상태값중 원하는 부분을 스택에 넣고 뺀다.

```
void glPushAttrib(GLbitfield mask);
void glPopAttrib(void);
```

저장 대상이 되는 상태값은 그룹별로 비트 마스크로 정의되어 있다. 그룹별로 다음과 같은 정보들이 저장되는데 일부만 보이므로 정확한 목록은 레퍼런스를 참조하기 바란다.

마스크	설명
GL_COLOR_BUFFER_BIT	GL_BLEND, GL_DITHER
GL_CURRENT_BIT	현재 색상, 현재 법선, GL_EDGE_FLAG
GL_DEPTH_BUFFER_BIT	GL_DEPTH_TEST

GL_ENABLE_BIT	GL_COLOR_MATERIAL, GL_DEPTH_TEST, GL_CULL_FACE
GL_FOG_BIT	GL_FOG, 안개 색상, 밀도, 시작점, 끝점
GL_HINT_BIT	GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT
GL_LIGHTING_BIT	GL_LIGHTING

복잡한 출력을 하려면 상태를 자주 바꿔야 하는데 이전 상태값으로 그대로 돌아 오려면 스택에 저장해 놓는 것이 편리하다.

3-5.버전 조사

다음 두 함수는 GL과 GLU 라이브러리 자체에 대한 정보를 조사한다. 그리기와는 직접적인 상관이 없지만 특정 기능의 지원 여부를 조사할 때 이 정보가 필요하다.

const GLubyte* glGetString(GLenum name);
const GLubyte * gluGetString(GLenum name);

name 인수로 어떤 정보를 조사할 것인지를 지정하며 조사 결과는 문자열로 리턴된다. 정보의 종류는 다음과 같다.

정보	설명
GL_VENDOR	GL을 구현한 회사명이 리턴된다.
GL_RENDERER	랜더러의 이름을 리턴한다.
GL_VERSION	버전 또는 릴리즈 번호이다.
GL_SHADING_LANGUAGE_VERSION	쉐이더 언어의 버전을 조사한다.
GL_EXTENSIONS	확장 기능 목록을 조사한다.

다음 예제는 이 두 함수로 조사한 정보들을 에디트 컨트롤에 출력한다. OpenGL은 문자열 출력을 직접 지원하지 않는데다 정보의 양이 대단히 많아 메시지 박스 정도로는 결과를 확인하기 어려워 전체 윈도우에 출력했다. 그래서 지금까지의 예제와는 달리 Win32 윈도우를 생성하고 메시지 루프도 돌린다.

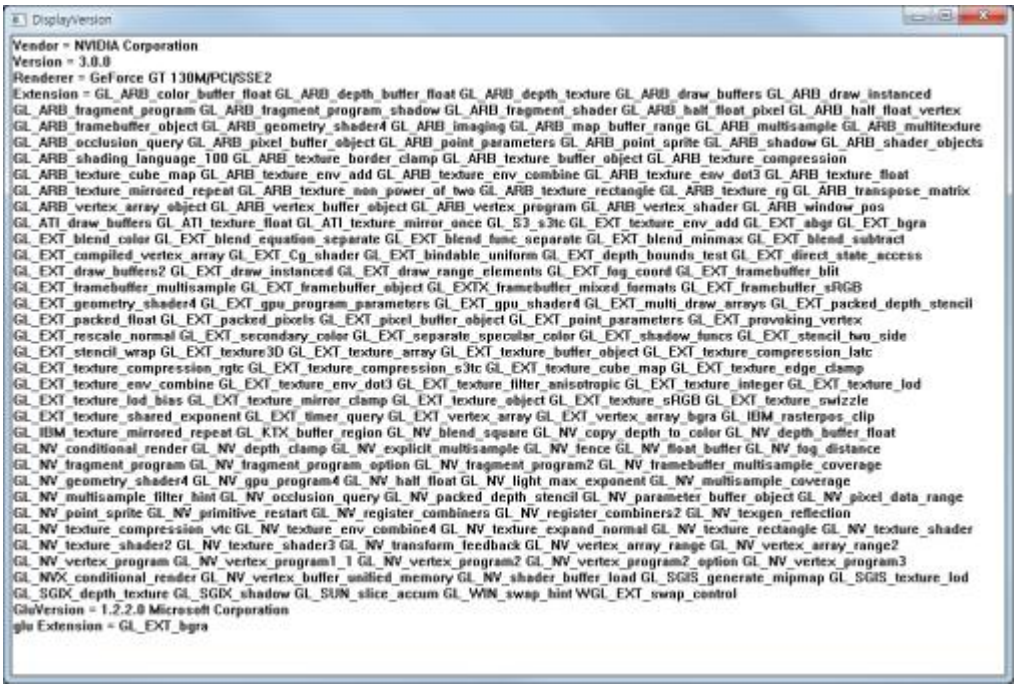
리턴하는 문자열이 안시 인코딩으로 되어 있으므로 이 프로젝트는 유니코드 문자 집합을 사용하지 않도록 설정했다. 또 윈도우즈의 문자열 조립 함수인 wsprintf가 1024 길이밖에 지원하지 못해 C 런타임의 sprintf 함수를 사용했다.

DisplayVersion

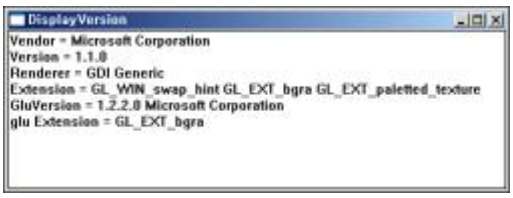
```
#define ID_EDIT 100
HWND hEdit;
#include <stdio.h>
LRESULT CALLBACK WndProc(HWND hWnd,UINT iMessage,WPARAM wParam,LPARAM lParam)
{
	TCHAR str[128];
	switch (iMessage) {
	case WM_CREATE:
		hEdit=CreateWindow(TEXT("edit"),NULL,WS_CHILD | WS_VISIBLE | WS_BORDER |
			ES_MULTILINE | ES_AUTOVSCROLL,
			10,10,200,25,hWnd,(HMENU)ID_EDIT,g_hInst,NULL);
		glutCreateWindow("OpenGL");
		char info[10240];
		sprintf(info ,"Vendor = %s₩r₩nVersion = %s₩r₩nRenderer = %s"
			"₩r₩nExtension = %s₩r₩nGluVersion = %s₩r₩nglu Extension = %s",
```

```
                (LPCSTR)glGetString(GL_VENDOR), (LPCSTR)glGetString(GL_VERSION),
                (LPCSTR)glGetString(GL_RENDERER), (LPCSTR)glGetString(GL_EXTENSIONS),
                (LPCSTR)gluGetString(GLU_VERSION), (LPCSTR)gluGetString(GLU_EXTENSIONS));
            SetWindowText(hEdit,info);
            return 0;
        case WM_SIZE:
            MoveWindow(hEdit,0,0,LOWORD(lParam), HIWORD(lParam), TRUE);
            return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        }
        return(DefWindowProc(hWnd,iMessage,wParam,lParam));
}
```

코드 자체는 무척 간단하다. 메인 윈도우가 생성되는 WM_CREATE에 모든 코드가 작성되어 있다. 일단 초기화는 해야 정보가 조사되므로 OpenGL 윈도우는 만들어야 한다. GetString 함수가 리턴하는 정보들을 하나의 문자열로 조립하여 에디트 컨트롤에 출력했다. 실행 환경에 따라 결과는 다르게 나타날 것이다. 다음은 윈도우즈 7에서의 실행 결과이다.



GL 버전과 GLU의 버전이 다르게 조사된다. 엄청나게 많은 확장 기능이 제공된다. 다음은 윈도우즈 XP에서의 실행 결과이다. GL 버전은 1.1이며 확장 기능은 거의 없다.



이 정보들 중에 확장 기능에 대해 자세히 알아 보자. OpenGL은 크로노스 그룹이 표준을 관리하며 모든 OpenGL 구현이 이 표준을 따른다. 그러나 그래픽 환경이 급진적으로 발전하기 때문에 표준 이상의 기능이 계속적으로 필요해진다. 표준이 이를 막아서는 안되므로 확장 기능의 추가를 공식적으로 허용한다. 단, 중구난방으로 아무렇게나 확장해서는 표준으로서의 의미가 없어지므로 제조사별로 접두어를 붙여 관리한다.

접두어	제조사
GL_ATI	ATI
GL_NV	NVidia
GL_IBM	IBM
GL_SUN	썬 마이크로 시스템즈
GL_WGL	마이크로소프트
GL_SGI	실리콘 그래픽스
GL_EXT	공용 확장 기능
GL_ARB	승인 확장 기능

확장된 기능은 최초 제작사의 하드웨어에서만 동작할 것이다. 이 기능의 유용성이 입증되어 일반화되면 차츰 다른 회사들도 이 기술을 공유할 것이며 이 상태가 되면 접두가 EXT로 바뀐다. 표준 위원회에서 해당 기능을 승인하면 ARB로 접두가 바뀌며 최종적으로는 접두가 없는 채로 다음 표준에 추가될 것이다.

확장 기능을 사용하려면 지원 여부를 먼저 조사해야 한다. 조사하는 방법은 아주 원시적인데 glGetString(GL_VERSION) 함수가 리턴하는 확장 기능 목록에 해당 기능이 있는지 strstr 등의 함수로 문자열 검색한다. 이때 부분 문자열을 검색하지 않도록 좌우에 공백을 잘 넣어 주어야 한다.

확장 기능의 존재를 확인했으면 호출하여 사용할 수 있되 확장 함수는 아직 공식적인 라이브러리에 포함되지 않았으므로 해당 함수의 포인터를 찾아 호출해야 한다. 함수 포인터를 검색하는 방법은 운영체제마다 다르다. 윈도우즈의 경우 wglGetProcAddress 함수로 찾는다. 함수의 번지를 찾았으면 함수 포인터 문법으로 호출한다.

좀 까다롭기는 하지만 꼭 필요하다면 이런 식으로 확장 기능을 검색하여 사용할 수 있다. 확장 기능을 사용하면 해당 기능이 있는 하드웨어에서만 동작할 것이다. 기능이 제공되지 않는 환경에서는 대체되는 기술을 사용하거나 아니면 최소한 에러 처리라도 우아하게 해야 한다.