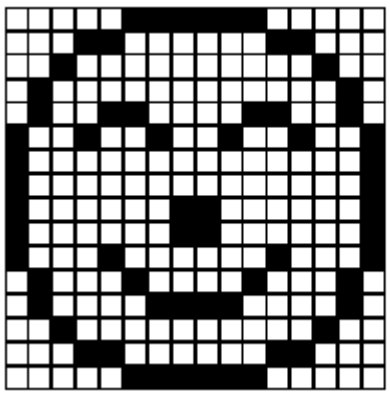
10.텍스처

10-1.비트맵

OpenGL은 3차원 그래픽 라이브러리이지만 2차원의 그래픽을 관리하는 기능도 제공한다. 텍스처 맵핑을 위해 2차원 비트맵을 처리하는 기능이 필요하며 또한 3차원 그래픽이라도 투영 후에는 2차원이 되므로 이 또한 다룰 수 있어야 하기 때문이다. 물론 전문적인 2차원 그래픽 라이브러리에 비할 바는 아니다.

비트맵은 미리 만들어진 이미지를 의미하며 다양한 크기와 색상을 가질 수 있다. 그러나 OpenGL에서는 흑백 이미지만을 비트맵이라고 칭하며 컬러 이미지는 픽셀맵(Pixelmap)이라는 별도의 용어를 사용한다. bit가 용어가 원래 0과 1만을 의미하는 흑백적인 뜻이기 때문이라고 한다. 사실 디지털 이미지라는 정의가 가장 정확한데 어쨌든 OpenGL은 컬러 이미지와 흑백 이미지를 구분한다고 알아 두자.

비트맵은 보통 대용량의 래스터 데이터로 구성되므로 파일 형태로 저장되는 것이 일반적이다. 그러나 안타깝게도 OpenGL은 파일 입출력 기능을 제공하지 않으므로 파일이나 리소스로부터 비트맵을 생성하기 어렵다. 간단한 이미지라면 메모리에서 래스터 데이터를 직접 정의하여 만들어 써야 한다. 모눈 종이에 다음과 같이 이미지를 디자인해 보자.



비트맵을 만드는 아주 원시적인 방법인데 각 칸의 흰색은 이진수 0에 대응시키고 검정색은 1에 대응시킨다. 이 수를 나열하면 2진수가 나오는데 자리 수가 너무 많으므로 보통 16진수로 압축한다. 이렇게 만든 16진수를 배열에 순서대로 기록하면 비트맵이 된다. OpenGL의 비트맵은 아래에서 위로 정의되므로 제일 아래줄부터 배열에 기록한다.

비트맵의 래스터 데이터는 처리의 효율성을 위해 4바이트 단위로 정렬된다. 왜냐하면 대부분의 컴퓨터는 32비트이고 한번에 4바이트씩 읽고 쓰는 속도가 가장 빠르기 때문이다. 한 행이 4의 배수가 안되더라도 뒷 부분에 더미를 추가해서라도 4의 배수로 맞추는 것이 유리하다. 그러나 위 이미지는 한 행이 2바이트 밖에 안되어 더미를 넣으면 무려 2배나 커진다. 정렬 단위를 바꾸고 싶다면 다음 함수로 비트맵의 저장 방식을 변경해야 한다.

void glPixelStore[f, i](GLenum pname, GLfloat param);

이 함수로 바이트 순서(GL_PACK_SWAP_BYTES)나 바이트 정렬(GL_PACK_ALIGNMENT) 등의 옵션을 지정한다. 메모리에서 비트맵을 읽을 때의 정렬은 GL_UNPACK_ALIGNMENT 옵션으로 지정하며 이 값의 디폴트가 4이므로 4바이트 정렬된다. 2나 1로 바꾸면 더 작은 단위로 정렬할 수 있되 단, 값을 일일이 잘라서 읽어야 하므로 속도에는 불리하다. 비트맵의 출력 위치는 다음 함수로 지정한다.

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

좌표를 지정하는 함수이므로 glVertex 함수와 형식이 거의 동일하다. 이 함수가 지정한 좌표가 현재 위치가 되며 이 위치에서부터 비트맵이 출력되고 비트맵 출력 후 현재 위치는 자동으로 다음 위치로 갱신된다. 마치 텍스트 모드에서 다음 문자가 출력될 위치를 지정하는 커서의 개념과 유사하다.

이 함수가 지정한 좌표는 변환의 영향을 받는다. 만약 변환에 상관없이 윈도우의 특정 좌표에 출력하려면 glWindowPos 함수를 대신 사용한다. 그러나 이 함수는 OpenGL 1.4 이후부터만 지원되므로 윈도우즈의 기본 구현에서는 당장 사용할 수 없다. 비트맵을 출력할 때는 다음 함수를 호출한다.

void glBitmap(GLsizei width, GLsizei height, GLfloat xorig, GLfloat yorig, GLfloat xmove, GLfloat ymove, const GLubyte * bitmap);

제일 마지막 인수인 bitmap이 비트맵의 모양을 정의하는 래스터 데이터 배열이다. width, height는 비트맵의 폭과 높이를 픽셀 단위로 알려 주는데 일차원 배열의 래스터만으로는 비트맵의 모양을 알 수 없기 때문에 크기 정보를 별도로 제공해야 한다. 위 비트맵은 16 * 16 크기이므로 폭과 높이가 모두 16이다.

xorig, yorig 인수는 비트맵의 어느 부분이 현재 위치에 출력될 것인지 원점을 지정한다. 보통은 좌하단인 (0,0)을 지정하여 비트맵의 아래쪽이 현재 위치에 출력되지만 모양이 특이한 비트맵은 원점을 변경할 수도 있다. xmove, ymove는 비트맵을 그린 후에 이동할 양을 거리를 지정한다. 마치 문자를 출력한 후 커서를 옮기듯이 현재 위치를 적당량 띄운다.

비트맵은 단 두 가지 색으로만 구성된 그림이지만 그렇다고 해서 꼭 흑백으로만 출력해야 하는 것은 아니다. glColor 함수로 색상을 지정하면 비트맵의 1에 대응되는 부분에 이 색상이 출력된다. 그래서 단색 비트맵이지만 노란색이나 빨간색으로도 출력할 수 있다. 다음 예제는 앞에서 디자인한 16 * 16 크기의 비트맵을 출력한다.

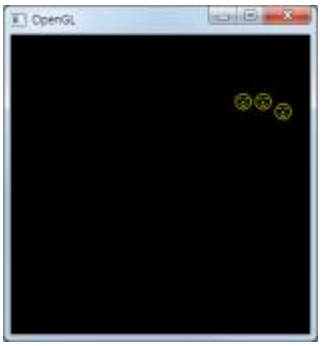
Bitmap

```
void DoDisplay()
{
	static GLubyte bitmap[] = {
		0x07, 0xe0, 0x18, 0x18, 0x20, 0x04, 0x43, 0xc2,
		0x44, 0x22, 0x88, 0x11, 0x81, 0x81, 0x81, 0x81,
		0x80, 0x01, 0x80, 0x01, 0x92, 0x49, 0x4c, 0x32,
		0x40, 0x02, 0x20, 0x04, 0x18, 0x18, 0x07, 0xe0,
	};
	glClear(GL_COLOR_BUFFER_BIT);

	glPixelStorei(GL_UNPACK_ALIGNMENT, 2);
	glColor3f(1,1,0);
	glRasterPos2f(0.5,0.5);
	glBitmap(16,16,0,0,20,0,bitmap);
	glBitmap(16,16,0,0,20,0,bitmap);
	glBitmap(16,16,0,10,20,0,bitmap);

	glFlush();
}
```

bitmap 배열에 래스터 데이터를 16진수로 저장했다. 2바이트 정렬로 지정했으므로 별도의 더미 데이터를 삽입할 필요는 없으며 아래행부터 순서대로 죽 나열하면 된다. 만약 4바이트 정렬이라면 한 행당 최소 4바이

트를 써야 하므로 두 바이트마다 0x00, 0x00을 삽입해야 한다.

색상은 노란색으로 지정했다. 비트맵의 색상은 glBitmap을 호출할 때 적용되는 것이 아니라 glRasterPos를 호출할 때 결정되므로 glRasterPos 함수를 호출하기 전에 먼저 지정해야 한다. 두 함수의 순서를 바꾸면 디폴트인 흰색으로 출력된다. 출력 위치는 0.5, 0.5로 지정했으므로 오른쪽 위 중간쯤에에서부터 비트맵이 나타날 것이다.



세 개의 비트맵을 출력해 보았다. xmove를 모두 20으로 지정했으므로 비트맵 하나를 출력한 후 오른쪽으로 20픽셀만큼 이동한다. 비트맵 폭인 16보다 4만큼 더 주어 약간씩 여백을 띄웠다. 만약 ymove를 20으로 지정하면 비트맵은 세로로 연이어 출력될 것이다. xmove, ymove는 비트맵을 연속 출력할 때 방향과 거리를 지정하는 인수이다.

앞쪽 두 비트맵은 원점을 (0,0)으로 지정하여 비트맵의 좌하단이 현재 위치에 대응된다. 세번째 비트맵은 y 원점을 10으로 주어 비트맵의 위쪽 10픽셀 지점이 현재 위치에 출력되며 그래서 비트맵이 조금 아래쪽으로 내려간다. 원점을 조정하는 기능은 폰트처럼 베이스가 각각 다른 이미지를 출력할 때 기준선에 맞추어 나란히 출력하기 위해서이다.

3차원 그래픽에서는 글꼴도 비트맵으로 제작하여 직접 출력하는 경우가 많은데 영문자의 경우 알파벳마다 베이스가 제각각이다. 이런 글꼴 비트맵은 모양에 따라 원점을 달리해야 한다. 한글은 정사각형 문자이므로 원점 조정을 따로 할 필요가 없고 글자의 폭도 대부분 균일하므로 띄어쓰는 간격도 일정하다. 세종대왕 만세.

10-2. 픽셀맵

컬러를 표현할 수 있는 픽셀맵은 비트맵에 비해서는 이미지를 만들거나 출력하기 훨씬 더 복잡하다. 출력 함수는 다음과 같다.

void glDrawPixels(GLsizei width, GLsizei height, GLenum format, GLenum type, const GLvoid * data);

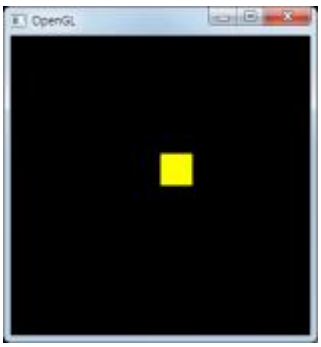
width, height는 이미지의 폭과 높이이다. format은 픽셀의 색상 구성이며 GL_RGB, GL_BGA 등을 지정한다. 이미지 파일에 따라 색상 정보가 RGB 순으로 되어 있는 것이 있고 BGA 순으로 되어 있는 것이 있다. 포맷의 종류는 레퍼런스를 참고하기 바란다. type은 data 배열의 요소 타입을 지정하며 data가 실제 이미지 정보인 래스터 데이터이다. 다음 예제는 간단한 픽셀맵을 배열로 정의하고 출력한다.

```
void DoDisplay()
{
	GLubyte data[32*32*3];

	for (int y=0;y<32;y++) {
		for (int x=0;x<32;x++) {
			data[y*32*3+x*3+0] = 0xff;
			data[y*32*3+x*3+1] = 0xff;
```

```
                data[y*32*3+x*3+2] = 0x00;
            }
        }

        glClear(GL_COLOR_BUFFER_BIT);

        glRasterPos2f(0.0,0.0);
        glDrawPixels(32, 32, GL_RGB, GL_UNSIGNED_BYTE, data);

        glFlush();
}
```

32*32크기의 픽셀맵을 정의하되 각 픽셀당 색상 정보 3개씩이 필요하므로 총 필요한 바이트수는 32*32*3이다. 이렇게 준비된 버퍼안에 각 점에 대해 RGB 요소를 일일이 정의하면 작으나마 아이콘 크기 정도의 이미지를 만들 수 있다. 그러나 이 작업은 16진수 3072개를 일일이 쳐 넣어야 하므로 제정신을 가진 사람이 손으로 할만한 작업이 못된다. 이미지로부터 색상 정보를 16진수로 추출해 내는 것만 해도 보통 일이 아니다.

그래서 어쩔 수 없이 루프를 돌며 모든 픽셀에 대해 R, G 요소는 255로 대입하고 B 요소는 0으로 대입하여 노란색으로 채워 넣었다. 좀 비겁하지만 귀차니즘으로 인해 모든 픽셀이 노란색인 작은 사각형 이미지를 하나 만든 것이다. 이렇게 만든 이미지를 화면 중앙에 출력했다. 픽셀맵은 비트맵과는 달리 원점을 지정하는 기능이 없으므로 무조건 이미지의 좌하단이 현재 위치에 출력된다. 그래서 중앙에서 약간 오른쪽 위에 이미지가 출력되었다.



이 예제는 픽셀맵을 출력하는 가장 원론적인 방법을 보이기 위해 작성한 것 뿐이다. 제대로 모양을 갖춘 이미지를 사용하려면 파일에서 읽어오는 것이 원칙적이다. OpenGL은 이미지 파일을 읽는 기능을 제공하지 않으므로 운영체제별로 적당한 이미지 읽기 함수를 만들어 사용해야 한다. 이 강좌는 윈도우즈 환경에서 실습을 진행하고 있으므로 윈도우즈 API로 읽기 용이한 BMP 파일을 출력했다.

```
GLubyte *LoadBmp(const char *Path, int *Width, int *Height)
{
        HANDLE hFile;
        DWORD FileSize, dwRead;
        BITMAPFILEHEADER *fh=NULL;
        BITMAPINFOHEADER *ih;
        BYTE *pRaster;

        hFile=CreateFileA(Path,GENERIC_READ,0,NULL,
                OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
        if (hFile==INVALID_HANDLE_VALUE) {
                return NULL;
        }

        FileSize=GetFileSize(hFile,NULL);
```

```
	fh=(BITMAPFILEHEADER *)malloc(FileSize);
	ReadFile(hFile,fh,FileSize,&dwRead,NULL);
	CloseHandle(hFile);

	int len = FileSize - fh->bfOffBits;
	pRaster=(GLubyte *)malloc(len);
	memcpy(pRaster, (BYTE *)fh+fh->bfOffBits, len);

	// RGB로 순서를 바꾼다.
	for (BYTE *p=pRaster;p < pRaster + len - 3;p+=3) {
		BYTE b = *p;
		*p = *(p+2);
		*(p+2) = b;
	}

	ih=(BITMAPINFOHEADER *)((PBYTE)fh+sizeof(BITMAPFILEHEADER));
	*Width=ih->biWidth;
	*Height=ih->biHeight;

	free(fh);
	return pRaster;
}

void DoDisplay()
{
	GLubyte *data;
	int Width, Height;

	glClear(GL_COLOR_BUFFER_BIT);

	data = LoadBmp("chestnut.bmp", &Width, &Height);
	if (data != NULL) {
		glRasterPos2f(-0.5,-0.5);
		glDrawPixels(Width, Height, GL_RGB, GL_UNSIGNED_BYTE, data);
		free(data);
	}

	glFlush();
}
```

이 예제의 LoadBmp 함수는 인수로 전달받은 이미지 파일로부터 래스터 데이터를 배열로 읽어들인다. BMP 파일의 구조와 파일 입출력 함수 정도만 쓸 수 있으면 이 정도 함수는 어렵지 않게 만들 수 있다. 파일에서 래스터 데이터에 해당하는 부분을 추출하여 버퍼에 복사하면 된다. 단, BMP는 색상 정보가 BGR 순으로 저장되어 있으므로 RGB 순서로 바꾸어 사용했다. OpenGL 도 BGR 순의 이미지 포맷인 GL_BGR을 지원하지만 안타깝게도 윈도우즈의 OpenGL은 버전이 낮은 관계로 GL_BGR을 지원하지 않는다.

LoadBmp 함수에 대한 자세한 분석이나 설명은 생략한다. 관심있는 사람은 Win32 관련 서적을 참고하기 바란다. 어떤 책이라고 굳이 꼭 집어서 소개하지는 않겠지만 시중에 좋은 Win32 서적들이 참 많이 있다. 이미지를 파일로 저장하는 기능이 필요하다면 SaveBmp 함수를 만들어 사용하면 된다. 만약 윈도우즈가 아닌 다른 운영체제에서 실습을 진행하거나 프로젝트를 해야 한다면 해당 운영체제의 이미지 조작 방법을 참고하거나 아니면 남이 만들어 놓은 유틸리티 함수를 구해 보도록 하자. 실행하면 컬러 이미지가 출력될 것이다.

이 예제에서 사용한 chestnut.bmp 파일은 잘 익은 밤 이미지이며 글쓴이가 손수 찍은 사진이다. 어떤 이미지든지 BMP로 변환해서 사용하면 위 함수를 활용할 수 있다. 안타깝게도 BMP는 윈도우즈 전용 포맷이라 다른 운영체제에서 사용하기는 어렵다. 좀 더 범용적인 포맷을 사용하되 압축 포맷보다는 TGA같은 비압축 포맷이 유리하다. JPG나 PNG 같은 압축 포맷도 별도의 그래픽 라이브러리를 활용하면 물론 사용할 수 있다.

파일로부터 이미지를 구하는 방법 외에 이미 출력된 화면의 이미지를 재사용할 수도 있다. 색상 버퍼의 픽셀값을 버퍼로 읽어들이거나 색상 버퍼의 일정 영역을 다른 곳으로 복사하면 된다. 다음 함수들은 화면의 일부를 버퍼로 읽어 들이고 화면끼리 복사한다. glRasterPos로 복사할 위치를 지정한 후 이 함수로 복사한다.

void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid * data);
void glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum type);

복사할 영역과 복사할 정보의 종류를 밝힌다. type에 GL_COLOR를 지정하면 색상 데이터를 복사하는 것이고 GL_DEPTH나 GL_STENCIL을 지정하면 깊이 정보와 스텐실 정보까지 복사한다. 다음 예제는 밤 이미지의 일부를 왼쪽 아래로 복사한다.

```
void DoDisplay()
{
	GLubyte *data;
	int Width, Height;

	glClear(GL_COLOR_BUFFER_BIT);

	data = LoadBmp("chestnut.bmp", &Width, &Height);
	if (data != NULL) {
		glRasterPos2f(-0.5,-0.5);
		glDrawPixels(Width, Height, GL_RGB, GL_UNSIGNED_BYTE, data);
		free(data);
	}

	glRasterPos2f(-1.0,-1.0);
	glCopyPixels(100,100,80,50,GL_COLOR);

	glFlush();
}
```

실행 결과는 다음과 같다.

이미지끼리 복사해서 별 실용성이 없어 보이지만 3차원 그래픽으로 그린 이미지의 일부를 잘라 텍스처로 사용할 때는 이런 기법이 편리하다. 화면에 그려진 래스터 데이터를 활용하는 것이므로 대용량의 배열을 선언할 필요도 없고 파일 입출력을 할 필요도 없다.

비트맵 작업은 싱글 버퍼 환경일 때는 프론트 버퍼를 대상으로 하지만 더블 버퍼링 환경에서는 백 버퍼를 대상으로 수행된다. 대상 버퍼를 강제로 바꾸고 싶다면 다음 두 함수를 사용한다.

void glDrawBuffer(GLenum mode);
void glReadBuffer(GLenum mode);

백 버퍼에 그려 놓고 스왑하면 깜박거림이 제거된다.

10-3.폴리곤 스티플

폴리곤 스티플링은 다각형의 내부를 채우는 아주 간단한 방법이다. 32*32 크기의 흑백 비트맵을 정의하고 다음 두 함수를 호출하면 된다.

glEnable(GL_POLYGON_STIPPLE);
void glPolygonStipple(const GLubyte * pattern);

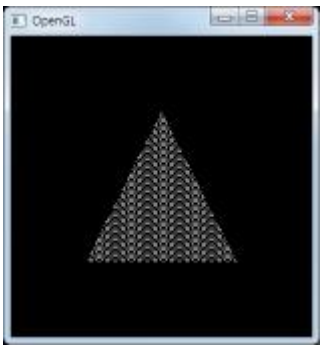
pattern은 32 * 32의 비트맵 패턴이므로 총 128 바이트로 구성된 배열이어야 한다. 비트맵을 만드는 것이 귀찮아서 그렇지 사용하는 방법은 아주 원론적이다.

PolygonStipple

```
void DoDisplay()
{
     static GLubyte bitmap[] = {
          0x07, 0xe0, 0x18, 0x18, 0x20, 0x04, 0x43, 0xc2,
          0x44, 0x22, 0x88, 0x11, 0x81, 0x81, 0x81, 0x81,
          0x80, 0x01, 0x80, 0x01, 0x92, 0x49, 0x4c, 0x32,
          0x40, 0x02, 0x20, 0x04, 0x18, 0x18, 0x07, 0xe0,
          0x07, 0xe0, 0x18, 0x18, 0x20, 0x04, 0x43, 0xc2,
          0x44, 0x22, 0x88, 0x11, 0x81, 0x81, 0x81, 0x81,
          0x80, 0x01, 0x80, 0x01, 0x92, 0x49, 0x4c, 0x32,
          0x40, 0x02, 0x20, 0x04, 0x18, 0x18, 0x07, 0xe0,
          0x07, 0xe0, 0x18, 0x18, 0x20, 0x04, 0x43, 0xc2,
          0x44, 0x22, 0x88, 0x11, 0x81, 0x81, 0x81, 0x81,
          0x80, 0x01, 0x80, 0x01, 0x92, 0x49, 0x4c, 0x32,
          0x40, 0x02, 0x20, 0x04, 0x18, 0x18, 0x07, 0xe0,
          0x07, 0xe0, 0x18, 0x18, 0x20, 0x04, 0x43, 0xc2,
          0x44, 0x22, 0x88, 0x11, 0x81, 0x81, 0x81, 0x81,
          0x80, 0x01, 0x80, 0x01, 0x92, 0x49, 0x4c, 0x32,
          0x40, 0x02, 0x20, 0x04, 0x18, 0x18, 0x07, 0xe0,
```

```
        };

        glClear(GL_COLOR_BUFFER_BIT);

        glEnable(GL_POLYGON_STIPPLE);
        glPolygonStipple(bitmap);

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

32*32 정도의 흑백 비트맵이면 금방 만들 수 있지만 시간에 쫓기다 보니 앞에서 만들었던 작은 비트맵을 그냥 4번 반복했다. 차후 시간이 허락되면 예쁘장한 이미지를 디자인해 보기로 한다. 어쨌든 데이터의 개수는 채워 넣었으므로 출력은 되지만 행 오프셋이 맞지 않아 괴상 망칙한 무늬가 나타날 것이다.



스티플링은 래스터 데이터만으로 도형의 내부를 채울 수 있다는 면에서 아주 간편한 방법이기다. 그러나 간편한만큼 제약이 많아 실용성은 떨어진다. 비트맵은 무조건 32 * 32 크기여야 하며 그나마도 단색 무늬만 가능해서 표현력의 제약이 심하다. 게다가 회전이나 확대, 축소도 적용되지 않아 현실적으로 거의 쓸 모가 없다. 도형의 내부를 원하는 무늬로 채우는 정석은 텍스처 맵핑이다.

10-4.텍스처 맵핑

텍스처 맵핑(Texture Mapping)은 3차원 물체의 표면에 이미지를 입히는 기법이다. 3차원 표면에 울퉁불퉁한 모양이나 나무 무늬 등을 그려 넣으려면 조명이나 물체의 재질을 지정하는 방법으로는 한계가 많다. 대량의 정점이 필요하고 정점이 많아지면 속도도 느려진다. 텍스처 맵핑은 미리 만들어져 있는 이미지를 다각형의 표면에 그려 넣음으로써 임의의 무늬를 빠른 속도로 그려내는 획기적인 방법이다.

텍스처 맵핑은 이미지를 읽어야 하고 각 면에 이미지의 어디쯤이 대응될 것인지를 일일이 지정해야 하므로 구현 단계는 다소 복잡하고 어렵다. 이미지의 데이터양도 꽤 많은 편이지만 다행히 현대의 그래픽 카드는 하드웨어 차원에서 텍스처를 지원하므로 속도는 만족할만큼 빠른 편이다. 텍스처 맵핑 기능을 사용하려면 먼저 해당 기능을 켜야 한다.

```
glEnable(GL_TEXTURE_2D);
```

주로 2차원 평면 이미지로 텍스처를 입히지만 1차원이나 3차원의 이미지를 사용할 수도 있다. 사용할 텍스처의 차원에 따라 개별적으로 활성화시켜야 하며 한번에 하나만 활성화할 수 있다. 통상은 2차원 텍스처를 가장 많이 사용하며 ES 버전은 2차원밖에 지원하지 않는다.

다음은 텍스처로 사용할 이미지를 메모리 버퍼로 로드한다. 여러 가지 방법으로 이미지를 준비할 수 있지만 가장 일반적인 방법은 이미지 파일에서 읽어오는 것이다. 앞에서 준비해둔 함수로 래스터 데이터를 배열에 읽어 놓는다. 이미지를 바로 사용할 수는 없으며 텍스처 맵핑에 적당한 형태로 가공해야 하는데 이때는 다음 함수를 호출한다.

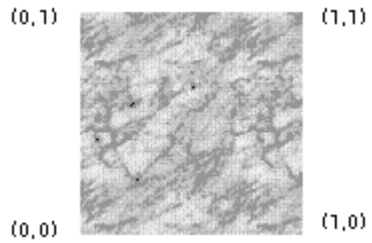
void glTexImage2D(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid * data);

인수가 굉장히 많을 뿐만 아니라 각 인수의 의미도 어렵고 선택할 수 있는 값의 종류도 많다. target은 이미지로부터 어떤 텍스처를 만들 것인가를 지정한다. 평범한 텍스처를 만들 수도 있고 프록시나 큐브맵을 만들 수도 있다. level은 밉멥 레벨이되 밉맵을 사용하지 않을 때는 0으로 지정한다.

내부 포맷은 텍셀의 포맷을 지정하는데 GL_RGB, GL_BGR 등등 여러 가지 포맷으로 만들 수 있다. width, height는 텍스처의 높이이되 반드시 2의 거듭승 크기여야 하며 여기에 경계선의 두께를 더해야 한다. 경계선이 없을 경우 32, 64, 128, 256 정도의 크기면 된다. border는 경계선의 두께를 지정하는데 없으면 0을 준다. format, type, data는 래스터 데이터의 포맷과 픽셀의 타입, 그리고 래스터 데이터 배열이다.

텍스처를 적용하기 전에 맵핑에 관련된 여러 가지 옵션들을 변경할 수 있는데 대부분은 디폴트가 무난하므로 디폴트를 일단 적용하면 된다. 자세한 것은 다음 항에서 따로 연구해 보자. 텍스처가 준비된 다음은 도형의 각 정점과 텍스처의 좌표를 대응시킨다. 도형의 어느 부분에 텍스처의 어디쯤을 출력할 것인가를 지정하는 것이다.

텍스처 이미지는 크기가 제각각인데다 크기를 바꾸는 경우도 종종 있으므로 픽셀 단위로 좌표를 지정하면 일관성이 없다. 그래서 0 ~ 1 사이의 실수 좌표로 비율을 지정한다. 정점의 좌표 표현시 보통 xyzw 명칭을 축으로 사용하지만 축 명칭이 같으면 헷갈리므로 텍스처는 strq라는 좌표축 명칭을 대신 사용한다.



이미지의 크기에 상관없이 텍스처의 좌하단은 (0,0)이고 우상단은 (1,1)이다. 정중앙은 물론 (0.5, 0.5)가 될 것이다. 각 정점에 텍스처의 좌표를 지정할 때는 다음 함수를 사용한다.

void glTexCoord[1,2,3,4][s,i,f,d][v](s, t, r, q);

정점에 텍스처 좌표를 대응시켜 놓으면 OpenGL이 정점의 위치에 텍스처의 대응되는 픽셀을 읽어 출력할 것이다. 정점들의 중간에는 정점과 텍스처의 거리 비율에 맞게 적당히 늘리거나 줄여서 픽셀들을 대응시켜 다각형 전체에 텍스처를 입힌다. 텍스처와 다각형의 크기가 보통 일치하지 않으므로 스트레칭은 거의 항상 발생한다. 다음 예제는 피라미드에 대리석 모양의 텍스처를 입힌다.

Mapping

```
#include <windows.h>
#include <gl/glut.h>
#include <stdio.h>
```

```
void DoDisplay();
void DoKeyboard(unsigned char key, int x, int y);
void DoMenu(int value);
GLfloat xAngle, yAngle, zAngle;
GLint EnvMode = GL_REPLACE;
GLint TexFilter = GL_LINEAR;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutKeyboardFunc(DoKeyboard);
      glutCreateMenu(DoMenu);
      glutAddMenuEntry("Replace",1);
      glutAddMenuEntry("Modulate",2);
      glutAddMenuEntry("Add",3);
      glutAddMenuEntry("Nearest Filter",4);
      glutAddMenuEntry("Linear Filter",5);
      glutAttachMenu(GLUT_RIGHT_BUTTON);
      glutMainLoop();
      return 0;
}

void DoKeyboard(unsigned char key, int x, int y)
{
      switch(key) {
      case 'a':yAngle += 2;break;
      case 'd':yAngle -= 2;break;
      case 'w':xAngle += 2;break;
      case 's':xAngle -= 2;break;
      case 'q':zAngle += 2;break;
      case 'e':zAngle -= 2;break;
      case 'z':xAngle = yAngle = zAngle = 0.0;break;
      }
      char info[128];
      sprintf(info, "x=%.1f, y=%.1f, z=%.1f", xAngle, yAngle, zAngle);
      glutSetWindowTitle(info);
      glutPostRedisplay();
}

void DoMenu(int value)
{
      switch(value) {
      case 1:
            EnvMode = GL_REPLACE;
            break;
      case 2:
            EnvMode = GL_MODULATE;
            break;
      case 3:
            EnvMode = GL_ADD;
            break;
      case 4:
            TexFilter = GL_NEAREST;
            break;
      case 5:
            TexFilter = GL_LINEAR;
            break;
      }
```

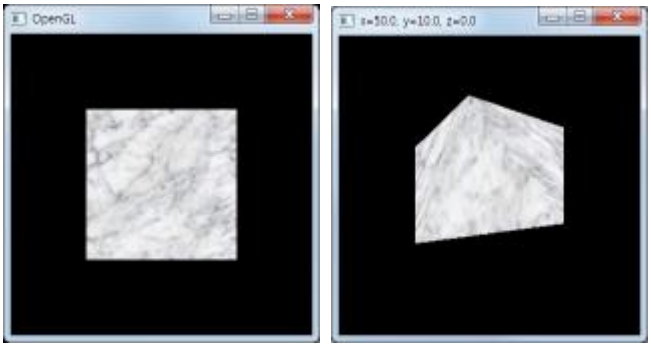
```
	glutPostRedisplay();
}

GLubyte *LoadBmp(const char *Path, int *Width, int *Height)
{
	HANDLE hFile;
	DWORD FileSize, dwRead;
	BITMAPFILEHEADER *fh=NULL;
	BITMAPINFOHEADER *ih;
	BYTE *pRaster;

	hFile=CreateFileA(Path,GENERIC_READ,0,NULL,
		OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
	if (hFile==INVALID_HANDLE_VALUE) {
		return NULL;
	}

	FileSize=GetFileSize(hFile,NULL);
	fh=(BITMAPFILEHEADER *)malloc(FileSize);
	ReadFile(hFile,fh,FileSize,&dwRead,NULL);
	CloseHandle(hFile);

	int len = FileSize - fh->bfOffBits;
	pRaster=(GLubyte *)malloc(len);
	memcpy(pRaster, (BYTE *)fh+fh->bfOffBits, len);

	// RGB로 순서를 바꾼다.
	for (BYTE *p=pRaster;p < pRaster + len - 3;p+=3) {
		BYTE b = *p;
		*p = *(p+2);
		*(p+2) = b;
	}

	ih=(BITMAPINFOHEADER *)((PBYTE)fh+sizeof(BITMAPFILEHEADER));
	*Width=ih->biWidth;
	*Height=ih->biHeight;

	free(fh);
	return pRaster;
}

void DoDisplay()
{
	GLubyte *data;
	int Width, Height;

	// 텍스처 이미지 준비
	glEnable(GL_TEXTURE_2D);
	data = LoadBmp("marble64.bmp", &Width, &Height);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, Width, Height,
		0, GL_RGB,  GL_UNSIGNED_BYTE, data);
	free(data);

	// 텍스처 환경 설정
	glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, EnvMode);

	// 텍스처 필터 설정
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, TexFilter);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, TexFilter);
```

```
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glShadeModel(GL_FLAT);
        glEnable(GL_DEPTH_TEST);

        glPushMatrix();
        glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
        glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
        glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

        // 아랫면 흰 바닥
        glBegin(GL_QUADS);
        glColor3f(1,1,1);
        glTexCoord2f(0.0, 1.0);
        glVertex2f(-0.5, 0.5);
        glTexCoord2f(1.0, 1.0);
        glVertex2f(0.5, 0.5);
        glTexCoord2f(1.0, 0.0);
        glVertex2f(0.5, -0.5);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(-0.5, -0.5);
        glEnd();

        // 위쪽 빨간 변
        glBegin(GL_TRIANGLE_FAN);
        glColor3f(1,0,0);
        glTexCoord2f(0.5, 0.5);
        glVertex3f(0.0, 0.0, -0.8);
        glTexCoord2f(1.0, 1.0);
        glVertex2f(0.5, 0.5);
        glTexCoord2f(0.0, 1.0);
        glVertex2f(-0.5, 0.5);

        // 왼쪽 노란 변
        glColor3f(1,1,0);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(-0.5, -0.5);

        // 아래쪽 초록 변
        glColor3f(0,1,0);
        glTexCoord2f(1.0, 0.0);
        glVertex2f(0.5, -0.5);

        // 오른쪽 파란 변
        glColor3f(0,0,1);
        glTexCoord2f(1.0, 1.0);
        glVertex2f(0.5, 0.5);
        glEnd();

        glPopMatrix();
        glFlush();
    }
```

64 * 64 크기의 대리석 표면 모양 이미지를 marble64.bmp 파일로 준비했다. 이미지의 각 부분을 피라미드의 각 정점에 적당히 대응시킨다. 피라미드의 꼭대기 정점에 이미지의 중앙을 대응시키고 나머지 모서리 정점에는 이미지의 같은 방향 모서리를 대응시켰다.

피라미드를 마루 바닥에 올려 놓고 대리석 모양이 그려진 보자기를 잘라서 위에서 그대로 씌운 것과 같다. 중심을 기준으로 피라미드의 각 변에 맞는 삼각형 조각을 입혔으므로 정면에서 보면 그냥 대리석 평면처럼 보

이지만 회전해 보면 각 면의 경계가 드러난다.



텍스처 방식을 여러 가지로 변형하는 기능들이 미리 작성되어 있는데 다음 항에서 따로 상세하게 연구해 보자.

10-5.텍스처 맵핑 옵션

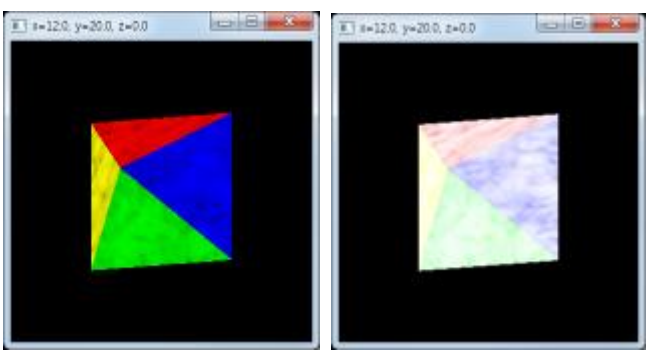
텍스처 맵핑은 굉장히 복잡한 연산이어서 선택할 수 있는 옵션들이 아주 많다. 다음 함수는 텍스처 맵핑 환경을 지정한다.

void glTexEnv[f, i](GLenum target, GLenum pname, GLfloat param);

target 인수는 어떤 환경을 조정할 것인가를 지정한다. GL_TEXTURE_ENV, GL_TEXTURE_FILTER_CONTROL, GL_POINT_SPRITE 중 하나이다. pname은 조정하고자 하는 옵션의 이름이고 param은 옵션의 값이다. GL_TEXTURE_ENV_MODE는 텍셀의 색상과 지오메트리의 색상을 결합하는 방식 지정하는데 다음 여섯 가지 방식이 있다.

방식	설명
GL_MODULATE	두 색상을 곱한다.
GL_REPLACE	색상을 무시하고 텍스처로 덮어쓴다.
GL_ADD	두 색상을 더한다.
GL_DECAL	
GL_BLEND	블랜딩 색상과 텍스처를 혼합한다.
GL_COMBINE	

피라미드의 각 면에 고유의 색상이 지정되어 있는데 결합 모드에 따라 이 색상과 텍스처 색상의 연산이 결정된다. REPLACE로 덮어 쓰면 텍스처의 무늬만 나타나지만 MODULATE나 ADD로 혼합하면 면의 원래색과 논리적으로 섞인다. 팝업 메뉴로 모드를 바꿔 보자.



텍스처 이미지가 다각형을 덮어 버리는 것이 아니라 원래색과 섞이는 효과가 나타난다. 이 모드에서는 똑같은 텍스처라도 다각형의 원래 색에 따라 무늬의 색상이 결정되므로 하나의 텍스처로 노란색, 초록색 대리석을 표현할 수 있다.

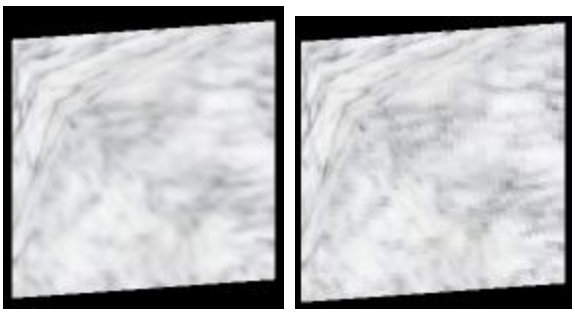
다음 함수는 텍스처 파라미터를 지정하는데 이 파라미터에 따라 텍스처를 그리는 랜더링 방식이 달라진다.

void glTexParameter[f, i][v](GLenum target, GLenum pname, GLfloat param);

target은 텍스처의 차원을 지정한다. pname은 파라미터의 이름이고 param은 값이다. 텍스처와 물체의 면적이 정확하게 일치하는 경우가 드물므로 텍스처를 늘리거나 줄여서 입혀야 한다. 텍스처 맵으로부터 입힐 색상을 계산하는 과정을 텍스처 필터링이라고 한다. GL_TEXTURE_MIN_FILTER는 축소시의 필터링을 지정하고 GL_TEXTURE_MAG_FILTER는 확대시의 필터링을 지정한다. 주로 다음 두 가지 필터가 사용된다.

- 최단거리(GL_NEAREST) : 비율상 대응되는 위치의 텍셀값을 그대로 사용한다. 알고리즘은 단순하지만 품질은 떨어진다.
- 선형(GL_LINEAR) : 대응되는 위치 주변의 텍셀값에 대한 평균을 계산한다. 오버헤더가 많지만 품질은 훨씬 더 좋다.

축소시의 디폴트는 GL_NEAREST_MIPMAP_LINEAR이며 확대시의 디폴트는 GL_LINEAR이다. 예제에서는 의도적으로 64 크기의 작은 텍스처를 입혀 놓고 필터링을 토글해 보았다. 차이를 분명히 보고 싶으면 창을 크게 해서 도형을 확대한 상태로 필터링을 바꿔 보면 된다.



LINEAR 필터링은 부드럽게 확대되지만 NEAREST 필터링은 품질이 훨씬 더 거칠며 계단 현상도 나타난다. 물론 속도는 반대겠지만 이 정도 도형에서 속도차를 실감하기는 어렵다. 텍스처를 더 큰 것으로 사용하면 확대에 의한 부작용이 감소하므로 품질은 더 좋아진다.

GL_TEXTURE_WRAP_S, T, R 파라미터는 각 축에 대한 텍스처 랩핑 방식을 지정하며 다음값중 하나를 지정한다.

랩핑	설명
GL_CLAMP	경계 부근을 반복한다.
GL_CLAMP_TO_BORDER	경계 텍셀의 값을 반복한다.
GL_CLAMP_TO_EDGE	
GL_MIRRORED_REPEAT	반사된 모양으로 반복한다.
GL_REPEAT	반복한다.

텍스처 범위와 텍셀 범위가 일치하지 않을 경우 반복 및 경계 부근의 처리 방식을 지정한다. 이 예제는 텍스처가 다각형보다 작으므로 이 파라미터를 지정할 필요가 없다.