4.기본 도형

4-1.점

모든 디지털 그래픽의 기본은 점(Pixel)이다. 선이나 면도 모두 점의 집합으로 표현할 수 있다. 그러나 3차원 그래픽의 가장 원자적인 요소는 정점(Vertex)이다. 정점은 색상이나 크기에 대한 정보는 없고 오로지 위치만을 가진다는 면에서 점과는 다르다. 정점은 다음 두 함수 블록 사이에서 정의한다.

void glBegin(GLenum mode);
void glEnd(void);

glBegin ~ glEnd 블록 사이에 다양한 함수들이 호출되는데 주로 정점을 배치하거나 속성을 바꾸는 명령들이다. 정점은 다음 함수로 지정한다. 앞에서 설명한대로 좌표 지정 인수를 4개까지 취할 수 있고 인수의 타입도 다양하다. 여기서는 평면 도형을 그리는 실습을 먼저 하므로 좌표는 x, y 두 개만 지정한다.

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

블록내의 정점들로 무엇을 어떻게 그릴 것인가는 glBegin으로 전달되는 모드값에 의해 결정된다. 다음과 같은 모드가 제공되며 모드에 따라 정점을 연결하는 방식이 달라진다. 이 모드들을 완벽하게 이해하고 자유자재로 쓸 수 있어야 한다.

모드	설명
GL_POINTS	독립적인 점
GL_LINE_STRIP	연결된 선분
GL_LINE_LOOP	시작점과 끝점을 이은 선분
GL_LINES	두개의 정점들을 이은 선분
GL_TRIANGLES	세개씩 연결한 삼각형
GL_TRIANGLE_STRIP	연결된 삼각형
GL_TRIANGLE_FAN	중심을 공유하는 삼각형
GL_QUADS	정점 4개씩을 연결하여 사각형을 그린다.
GL_QUAD_STRIP	연결된 사각형
GL_POLYGON	연결된 볼록 다각형

기하학적으로 가장 간단한 점부터 찍어 보자. GL_POINTS 모드로 시작하고 블록내에서 정점의 좌표를 지정하면 정점 위치에 점들이 찍힌다.

Primitive

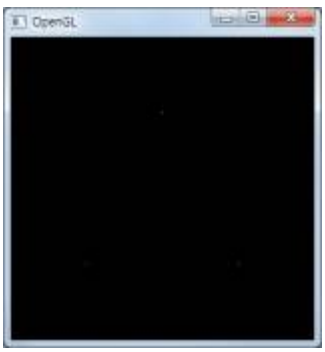
```
void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glBegin(GL_POINTS);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.5, -0.5);
	glVertex2f(0.5, -0.5);
```

```
	glEnd();
	glFlush();
}
```

glBegin ~ glEnd 사이는 하나의 도형을 정의하는 논리적인 블록이다. 그래서 가독성을 높이기 위해 두 블록 사이에 { } 괄호로 감싸고 들여쓰기를 하기도 한다.

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

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

glBegin과 glEnd 사이에 빈 { } 괄호를 두고 정점을 이 블록안에 배치하며 한칸 들여쓴다. 물론 이 괄호는 문법적으로는 아무 의미가 없으며 단지 블록을 읽기 쉽게 표시할 뿐이다. 보기에는 좋지만 다소 번거롭고 소스가 길어지므로 이 강좌에서는 그냥 평이하게 쓰기로 한다. 세 개의 정점을 정의했으므로 삼각형의 세 꼭지점에 점이 찍힐 것이다.



점의 디폴트 크기는 1픽셀이라 너무 작아서 잘 보이지 않는다. 좌표는 위치만 가지는데 비해 점은 화면상에 표시되므로 크기를 변경할 수 있고 색상도 지정할 수 있다. 점 크기의 변경 범위와 조정 단위는 구현에 따라 다른데 다음 코드로 조사한다.

```
GLfloat range[2], granu;
glGetFloatv(GL_POINT_SIZE_RANGE, range);
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &granu);
```

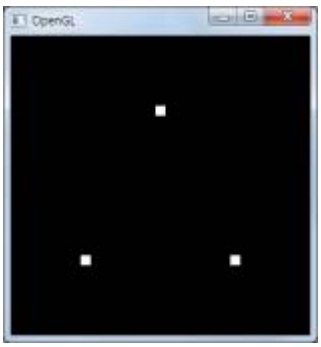
윈도우 환경에서는 1.0 ~ 63.375까지의 범위를 가지며 0.125 단위로 지정할 수 있다. 범위를 벗어나더라도 가장 가까운 값이 선택되며 에러는 나지 않는다. 점 크기는 다음 함수로 지정한다. 점 크기의 디폴트는 1이다.

void glPointSize(GLfloat size);

size 인수는 점을 감싸는 원의 직경을 지정한다. 안티 알리아싱을 하지 않으면 점은 시각형으로 출력되는데 size는 사각형의 한변 길이에 해당한다. 다음 코드는 점을 10 픽셀로 확대하여 출력한다.

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

	glPointSize(10.0);
	glBegin(GL_POINTS);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.5, -0.5);
	glVertex2f(0.5, -0.5);
	glEnd();
	glFlush();
}
```

glBegin 블록에 들어가기 전에 glPointSize 함수로 점 크기를 10으로 지정했다. glBegin 블록 안에서 지정해도 효과는 동일하며 중간에라도 언제든지 점의 크기를 변경하여 크기가 각각 다른 점을 찍을 수도 있다.



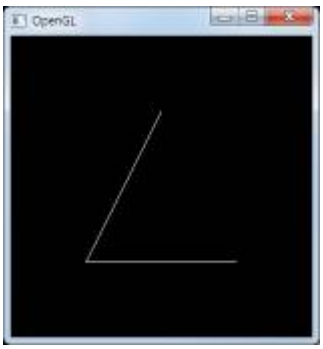
점을 확대하면 정사각형 형태로 출력되는데 안티알리아싱을 지정하면 둥그렇게 원 모양으로 표시된다. 점은 가장 기본적인 그래픽 요소이지만 점만으로 표현할 수 있는 것이 거의 없어 사용 빈도는 낮다.

4-2. 선

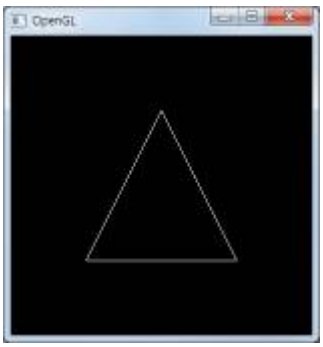
정점들을 연결하면 선분이 된다. 선을 그리는 모드는 3가지가 있다. 모드를 다음과 같이 변경해 보자.

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

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

GL_LINE_STRIP은 정점들을 연결하여 선분을 그린다. STRIP은 계속 이어서 그린다는 뜻이다. 첫번째 정점과 두번째 정점을 연결하고 이어서 두번째 정점과 세번째 정점을 연결한다. 그러나 처음과 끝을 연결하지 않으므로 열린 개곡선이 된다.



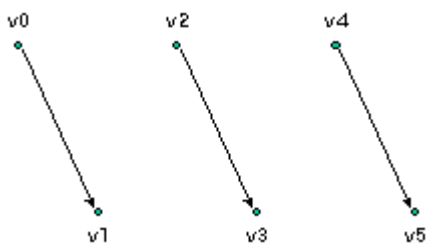
그려지는 선의 개수는 정점의 개수보다 하나 더 적다. 정점이 3개이므로 선분은 2개만 그려진다. GL_LINE_LOOP는 선분을 이어서 그리고 시작점과 끝점을 자동으로 연결하여 폐곡선을 만든다. 세번째 정점과 첫번째 정점이 연결되어 삼각형 모양이 그려진다.



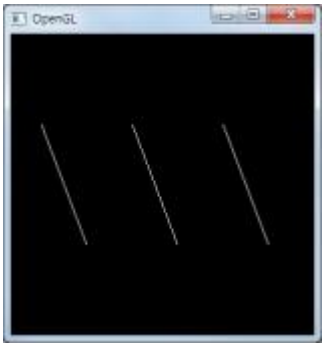
GL_LINES는 정점들을 두개씩 한 쌍으로 묶어 선을 그린다. 여러 개의 정점을 배치해 놓으면 각각 떨어진 선분이 그려진다. 다음 코드는 6개의 정점을 지그재그로 배치하고 6개의 선을 긋는다.

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

	glBegin(GL_LINES);
	GLfloat x = -0.8;
	GLfloat y = 0.4;
	for (int i = 0; i < 6; i++) {
		glVertex2f(x, y);
		x += 0.3;
		y *= -1;
	}
	glEnd();
	glFlush();
}
```

첫번째, 두번째 정점이 연결되고 세번째, 네번째 정점이 연결되는 식이다. 정점이 6개이므로 두 개씩 짝을 지어 총 3개의 선분이 그려진다.



만약 정점이 홀수개라면 마지막 정점은 대응되는 짝이 없으므로 무시된다.



선의 두께는 디폴트로 1이되 더 굵게 그릴 수도 있다. 점과 마찬가지로 구현에 따라 지정 가능한 범위가 다르다. 다음 코드로 범위를 구한다.

```
GLfloat range[2], granu;
glGetFloatv(GL_LINE_WIDTH_RANGE, range);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &granu);
```

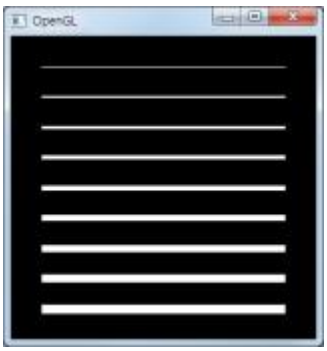
윈도우즈 환경에서 조사한 결과 0.5 ~ 10.0 까지 0.125 단위로 조정할 수 있다. 선의 두께를 변경할 때는 다음 함수를 호출한다.

void glLineWidth(GLfloat width);

다음 예제는 1부터 시작해서 선 두께를 점점 증가시켜가며 각각 다른 굵기로 수평선을 여러 개 긋는다.

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

	GLfloat y;
	GLfloat w = 1;
	for (y = 0.8; y > -0.8;y -= 0.2) {
		glLineWidth(w++);
		glBegin(GL_LINES);
		glVertex2f(-0.8, y);
		glVertex2f(0.8, y);
		glEnd();
	}
	glFlush();
}
```

y 좌표 0.8부터 시작해서 0.2만큼씩 아래로 이동하며 선을 그었다. 한단계 내려갈 때마다 선의 굵기는 1씩 증가한다. 윈도우 크기를 변경해도 선의 굵기는 유지된다.



실선이 아닌 다른 모양의 선을 그리려면 스티플(Stipple:점묘법) 기능을 켜야 한다. 특정 기능을 사용할 때는 glEnable 함수로 사용할 기능의 이름을 전달한다. 스티플 기능은 다음 호출문에 의해 활성화된다. 대부분의 기능은 디폴트로 꺼져 있으므로 사용하려면 반드시 켜야 한다.

```
glEnable(GL_LINE_STIPPLE);
```

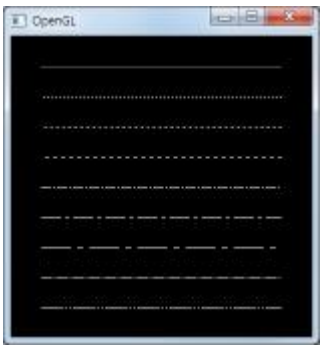
물론 이 기능이 더 이상 필요없으면 glDisable 함수로 언제든지 기능을 취소할 수 있다. 선의 모양은 다음 함수로 지정한다.

void glLineStipple(GLint factor, GLushort pattern);

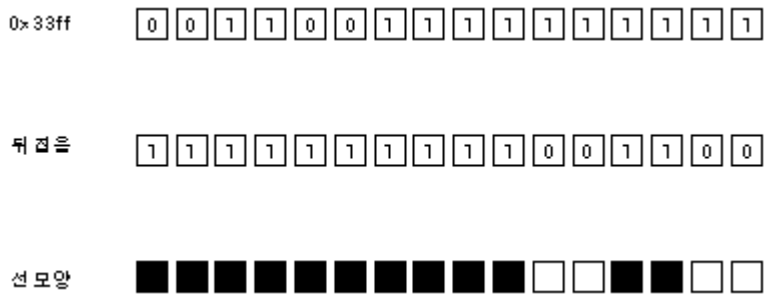
pattern은 이진수로 표현한 선의 모양이다. 하위 비트부터 선의 앞쪽 부분의 점 모양을 지정한다. 대응되는 비트가 1인 자리는 점이 찍히고 0인 부분은 찍히지 않는다. factor 인수는 비트 하나가 점 몇 개에 대응될 것인가를 지정한다. 이 값이 1이면 비트 하나가 점 하나에 대응되며 2이면 비트당 2개의 점이 그려져 좀 더 긴 모양을 만들 수 있다. 다음 예제는 점섬, 쇄선 등을 출력한다.

```
void DoDisplay()
{
	GLushort arPat[]={0xaaaa,0xaaaa,0xaaaa,0xaaaa,0x33ff,0x33ff,0x33ff,0x57ff,0x57ff };
	GLint arFac[] = { 1,	2,	3,	4,	1,	2,	3,	1,	2};

	glClear(GL_COLOR_BUFFER_BIT);
	glEnable(GL_LINE_STIPPLE);
	GLfloat y;
	GLint idx = 0;
	for (y = 0.8; y > -0.8;y -= 0.2) {
		glLineStipple(arFac[idx], arPat[idx]);
		glBegin(GL_LINES);
		glVertex2f(-0.8, y);
		glVertex2f(0.8, y);
		glEnd();
		idx++;
	}
	glFlush();
}
```

pattern와 factor 인수에 대해 9개의 인수쌍을 배열로 준비해 두고 각 모양대로 선을 그려 보았다. 배열의 앞쪽 요소부터 위쪽에서 순서대로 적용하여 그렸다.



0xaaaa는 이진수로 1010101010101010이므로 점과 공백이 계속 반복되는 점선이 된다. factor가 2면 점과 공백이 두 배로 확장되므로 더 성긴 점선이 그려진다. factor를 3, 4로 더 크게 지정하면 점 사이의 거리가 더욱 멀어진다. 0x33ff는 일점 쇄선의 패턴을 지정하는데 각 비트가 어떻게 대응되는지를 보자.



하위 비트부터 오른쪽에서 순서대로 점과 대응되므로 비트를 뒤집어야 한다. 왜 하위부터 앞쪽 점에 대응시키는가하면 기계적 연산이 간단하기 때문이다. 점 10개 그리고 2개 건너 뛰고 2개 그리고 다시 2개 건너 뛴다. 긴선, 짧은선이 계속 반복되므로 일점 쇄선이 된다. 끝부분에 여백이 있어야 반복될 때 긴선과 짧은선이 붙지 않는다.

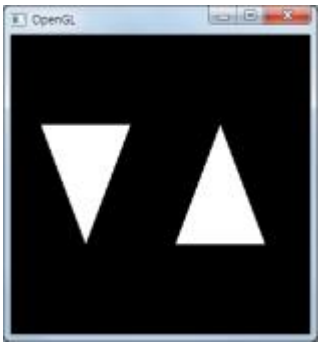
0x57ff는 이진수로 0101011111111111이 되며 긴선 하나에 짧은선 두 개가 계속 반복되므로 이점 쇄선이다. 어떤 모양이든간에 비트로 선 모양을 만들고 16진수로 바꿔서 패턴으로 사용하면 된다.

4-3. 삼각형

2차원 그래픽의 기본 요소는 픽셀이지만 3차원 그래픽의 기본 요소는 삼각형이다. 작은 삼각형들이 모여서 모든 물체들이 구성된다. 그래서 삼각형이 아주 중요하며 자주 사용된다. GL_TRIANGLES 모드는 정점 세개씩을 모아서 삼각형을 그린다. 제일 처음 만들었던 예제가 바로 이 모드로 삼각형을 그렸다.

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

	glBegin(GL_TRIANGLES);
	GLfloat x = -0.8;
	GLfloat y = 0.4;
	for (int i = 0; i < 6; i++) {
		glVertex2f(x, y);
		x += 0.3;
		y *= -1;
	}
	glEnd();
	glFlush();
}
```

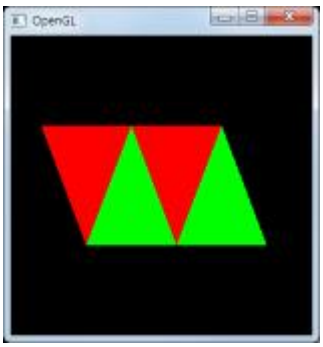
6개의 정점에 대해 삼각형을 그리면 2개의 삼각형이 만들어진다. 3개씩 짝을 지어 삼각형 하나를 그리므로 정점 개수를 3으로 나눈 개수만큼의 삼각형이 그려진다. 3의 배수에서 남는 정점은 무시된다.



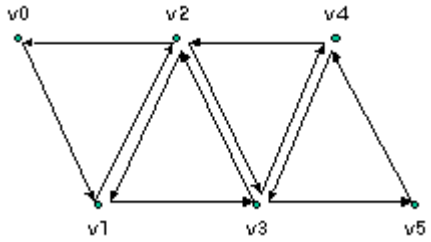
GL_TRIANGLE_STRIP은 삼각형을 계속 이어서 그린다. 첫 세 개의 정점으로 삼각형을 그리고 추가되는 정점을 새로운 꼭지점으로 하는 삼각형을 계속 이어서 그린다.

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

	glBegin(GL_TRIANGLE_STRIP);
	GLfloat x = -0.8;
	GLfloat y = 0.4;
	for (int i = 0; i < 6; i++) {
		if (i % 2 == 0) {
			glColor3f(1.0, 0.0, 0.0);
		} else {
			glColor3f(0.0, 1.0, 0.0);
		}
		glVertex2f(x, y);
		x += 0.3;
		y *= -1;
	}
	glEnd();
	glFlush();
}
```

정점이 6개이므로 삼각형은 모두 4개가 그려진다. 색상을 지정하지 않으면 전부 흰색으로 그려져 하나의 평행사변형으로 보이므로 각 정점마다 색상을 번갈아가며 지정하고 쉐이드 모델을 FLAT으로 지정하여 단색으로 채색했다. 쉐이드 모델에 대해서는 차후 설명하기로 한다.



각 삼각형을 분리해 보면 다음과 같다.



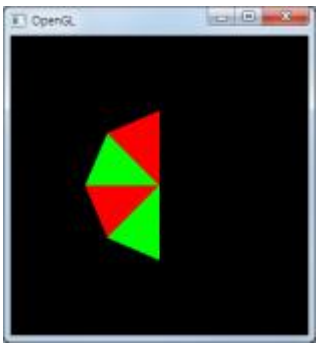
v0, v1, v2 세 개의 정점을 연결하여 첫번째 삼각형을 그린다. 그리고 v1, v2를 한 변으로 하고 새로 추가된 v3를 나머지 한 꼭지점으로 하는 삼각형을 그린다. 계속해서 v2, v3, v4를 꼭지점으로 하는 삼각형을 그리고 v3, v4, v5를 꼭지점으로 하는 삼각형이 그려진다. 이런식으로 이전의 정점들을 연결해서 그리면 적은 개수의 정점으로도 많은 삼각형을 그릴 수 있다는 이점이 있다.

n개의 삼각형을 그리는데 n+2개의 정점만 있으면 된다. 개별적으로 삼각형을 그리는 방법에 비해 이전 정점을 재활용하므로 메모리도 절약되고 속도도 훨씬 빠르다. 새로 그려지는 삼각형은 반시계 방향으로 그려진다. 삼각형을 그리는 방향(Winding)은 삼각형의 앞뒤를 구분하는 의미가 있는데 다음에 자세히 알아볼 것이다.

GL_TRIANGLE_FAN은 첫 삼각형의 꼭지점 하나를 고정해 두고 새로 추가되는 두 정점을 연결하여 계속 삼각형을 그린다. 마치 부채살을 추가하여 부채를 만드는 방법과 비슷해서 FAN이라는 이름이 붙었다.

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

	glBegin(GL_TRIANGLE_FAN);
	glColor3f(1.0, 0.0, 0.0);
	glVertex2f(0.0, 0.0);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.35, 0.35);

	glColor3f(0.0, 1.0, 0.0);
	glVertex2f(-0.5, 0.0);

	glColor3f(1.0, 0.0, 0.0);
	glVertex2f(-0.35, -0.35);

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

중심부에 첫 정점을 찍고 12시방향부터 반시계 방향으로 돌며 삼각형을 계속 이어 그렸다. 삼각형간의 구분을 위해 색상을 교대로 바꾸었다.



간단하게 작성하기 위해 정점들을 직접 찍었는데 루프를 돌며 원주상의 점들을 순회하면 원형이나 원뿔처럼 꼭지점을 공유하는 도형을 쉽게 만들 수 있다. 이 삼각형들을 아주 잘게 나누면 완전한 원모양이 될 것이다.

4-4. 사각형

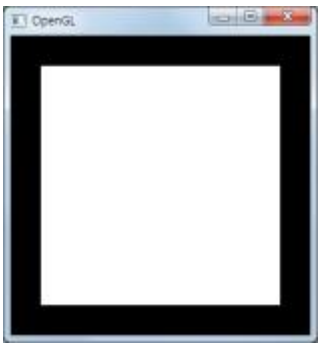
사각형도 입체 물체를 구성하는 기본 요소이다. 삼각형에 비해 약간의 제약이 있지만 그리기 속도가 빠르고 적은 개수로도 넓은 면적을 그릴 수 있어 종종 사용된다. 사각형은 자주 사용되므로 별도의 그리기 함수가 제공된다.

void glRect[i,s,f,d][v](x1, y1, x2, y2)

완전히 독립된 함수이므로 glBegin ~ glEnd 블록에 포함시키지 않고도 사각형을 그릴 수 있다. 인수로 좌상단 좌표와 우하단 좌표를 주거나 또는 각 좌표값의 배열을 전달한다. 다음은 사각형을 그린다.

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

	glRectf(-0.8, 0.8, 0.8, -0.8);
	glFlush();
}
```

왼쪽 위와 오른쪽 아래를 꼭지점으로 하는 큼직한 사각형이 그려진다.

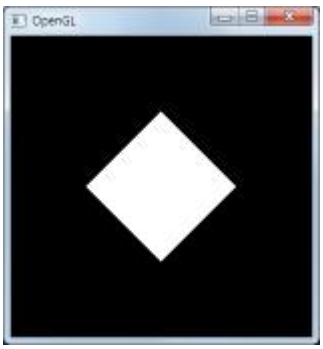


아주 쉽게 사각형을 그릴 수 있지만 대각선의 두 점을 지정하는 식이므로 각 변이 수직인 직사각형만 그릴 수 있다. 평행사변형이나 사다리꼴처럼 직각이 아닌 사각형은 그릴 수 없다. 불규칙한 사각형은 정점을 직접 지정하여 다각형으로 그려야 한다. 다음은 마름모를 그린다.

```
void DoDisplay()
{
```

```
    glClear(GL_COLOR_BUFFER_BIT);

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

GL_QUADS는 4개씩 정점을 연결하여 사각형을 그린다. 제일 위의 꼭지점을 시작으로 하여 반시계 방향으로 마름모를 구성하는 꼭지점을 전달했다. 임의 위치의 정점을 전달할 수 있으므로 직각이 아닌 사각형을 그릴 수 있다.



정점을 4개만 지정했으므로 사각형은 하나만 그려진다. 8개라면 2개, 12개라면 3개의 사각형이 그려질 것이다. GL_QUAD_STRIP 모드는 추가되는 2개의 정점으로 이전 사각형을 계속 이어서 그린다. 6개의 정점으로 2개의 사각형을 연속으로 그릴 수 있다.

4-5. 다각형

마지막 모드인 GL_POLYGON은 모든 정점을 하나로 연결하여 다각형을 그린다. 정점의 개수만큼의 다각형이 만들어지는데 5개면 5각형, 8개면 8각형이 그려진다. 삼각형이나 사각형도 이 모드로 그릴 수 있다. 정점이 몇개든간에 모조리 연결해서 하나의 도형을 정의할 수 있다.

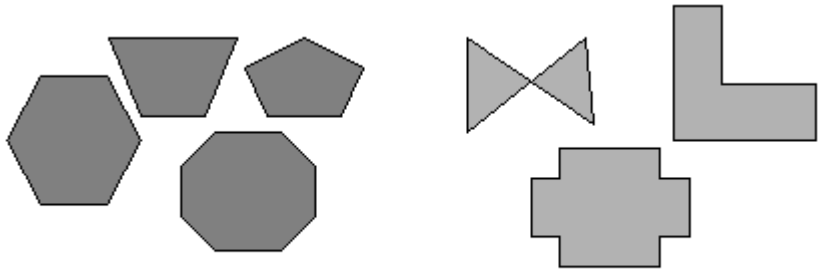
굉장히 자유도가 높은 모드인 것 같지만 아무 다각형이나 그릴 수 있는 것은 아니고 까다로운 조건이 적용된다. GL_POLYGON 모드로 그리는 다각형음 다음 세 가지 조건을 반드시 만족해야 한다.

① 정점의 선이 교차해서는 안된다.

② 다각형은 볼록해야 한다.

③ 모든 정점은 같은 평면내에 있어야 한다.

첫번째, 두번째 조건은 다소 이해하기 쉽다. 다각형은 하나의 닫힌 도형이어야 하며 어디가 안쪽이고 어디가 바깥쪽인지 명확하게 구분되어야 한다. 그러나 선이 교차하거나 오목하면 이런 판단이 아주 어려워진다. 다음 예를 보자.



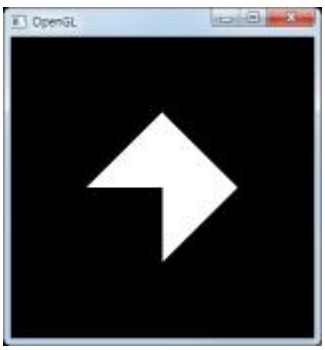
적법한 다각형 안적법한 다각형

왼쪽은 모두 GL_POLYGON 모드로 그릴 수 있는 다각형이다. 그러나 오른쪽은 제대로 그려지지 않는다. 리본 모양의 다각형은 선분이 교차되어 두 개의 다각형으로 분할된 것처럼 보인다. L자 모양의 다각형은 오목해서 적법하지 않다. 볼록하다는 것은 내부에서 임의의 선분을 그었을 때 선분이 다각형을 벗어나지 않아야 함을 뜻한다. 그러나 L자 모양은 선분이 바깥을 벗어날 수 있어 오목하다. 십자형의 다각형도 같은 이유로 적법하지 않다.

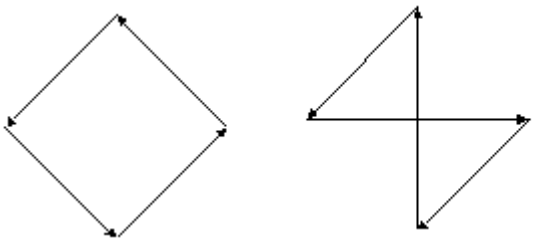
앞에서 마름모를 GL_QUADS로 그렸는데 4개의 정점을 연결하는 것이므로 GL_POLYGON으로 모드를 바꾸어도 정상적으로 잘 그려진다. 마름모를 구성하는 4 정점이 반시계 방향으로 제대로 나열되어 있기 때문이다. 이 코드를 다음과 같이 수정해 보자.

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

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

3번째 정점과 4번째 정점의 순서를 바꾸었다. 이렇게 되면 선분이 교차하며 오목한 다각형이 그려진다.



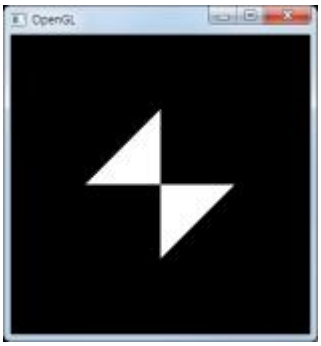
OpenGL이 인정하는 다각형이 아니다. 두 다각형의 차이점은 정점의 순서이다. 왼쪽 마름모는 정점을 잇는 선분들이 만나지 않지만 순서를 바꾸면 선분끼리 교차할 뿐만 아니라 오목해져 버린다.



에러는 발생하지 않고 어쨌든 그려지지만 이것이 과연 의도했던 다각형이 맞는지는 애매하다. 만약 꼭 저런 식으로 다각형을 그리고 싶다면 규칙에 맞는 두 개의 다각형으로 분할해야 한다.

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

	glBegin(GL_TRIANGLES);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.5, 0.0);
	glVertex2f(0.0, 0.0);

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

이 경우는 리본의 아래쪽과 위쪽을 두 개의 삼각형으로 나누고 GL_TRIANGLES 모드로 그리면 된다. 볼록한 두 개의 다각형이 그려지며 둘 다 다각형 조건에 적합하다.



3번째 조건은 다각형을 구성하는 모든 정점이 같은 평면에 소속되어야 한다는 것이다. 다음 두 개의 사각형을 보자.

2차원 화면상에서 이 두 도형의 상태를 정확하게 그리는 것은 사실 무척 어려운 일이어서 약간의 상상력이 필요하다. 왼쪽 사각형은 A4 용지를 책상위에 반듯하게 놓은 모양이며 용지의 꼭지점 4면이 책상이라는 평면에 모두 소속되어 있는 상태이다. 이런 도형은 적법한 다각형으로 인정된다.

오른쪽 사각형은 A4 용지를 책상에 놓은 후 오른쪽 아래 귀퉁이를 위쪽으로 약간 치켜든 상태이다. 한쪽이 하늘 위로 솟아 올랐으므로 사각형의 4점이 모두 책상면에 소속되지 않았다. OpenGL은 이런 다각형을 인정

하지 않으며 제대로 그려내지도 못한다. 좀 더 심하게 꼬아서 뫼비우스의 띠처럼 비틀어 놓으면 어디가 앞이고 어디가 뒤인지도 헷갈릴 것인다.

4각형, 6각형, 8각형 등 얼마든지 많은 꼭지점을 가지는 다각형을 정의할 수는 있다. 그러나 반드시 모든 정점은 한 면에 속해야 한다. 위 예에서 보다시피 사각형은 항상 이 조건을 만족시키지 못한다. 반면 삼각형은 어떤 경우라도 한 평면에 속한다는 특징이 있다. 왜 그런지 차원을 낮추어 생각해 보자. 두 개의 점은 무조건 1차원 직선에 속하지만 점이 3개로 늘어나면 세 점이 나란히 있는 특수한 경우를 제외하고는 한 선분에 속할 수 없다.

점 2개는 어떤 경우라도 한 선분으로 연결 가능하다.

점 3개는 한 선분으로 연결할 수 없다.

이제 이 이론을 한차원 높여 생각해 보면 삼각형과 사각형의 특징도 분명해진다. 점 3 개는 어떤 위치에 있더라도 이들을 모두 포괄하는 하나의 평면을 정의할 수 있다. 그러나 점이 하나 더 늘어나 4개가 되면 이 점들이 한 평면에 속하지 않는 경우가 더 많아지며 일반적으로 같은 평면에 속하기 어렵다. 손가락 3개를 편 상태에서 세 손가락을 모두 책상에 붙이는 것은 언제나 가능하지만 손가락 4개는 그렇지 않다는 것을 생각해 보자.

사각형도 조건만 맞추면 다각형이 될 수 있으며 삼각형보다 처리 속도가 빠르지만 반드시 조건을 지켜야 한다. 이런 이유로 3차원 그래픽을 구성하는 기본 단위는 대부분의 경우 삼각형이다. 아무리 복잡한 물체도 삼각형의 조합으로 표현할 수 있다. 사각형은 삼각형 2개를 붙이면 간단하게 정의된다. 참고로 모바일 환경의 OpenGL ES는 사각형을 아예 인정하지 않으며 무조건 삼각형만 가능하다.

OpenGL이 다각형에 대해 이런 까다로운 제한을 두는 이유는 그래야 속도가 빠르기 때문이다. 이런 저런 조건들 다 고려해서 다각형을 그리자면 너무 복잡해지고 계산양도 많아진다. 원자적인 다각형이 단순해야 더 효율적이고 빠른 알고리즘을 적용할 수 있으며 하드웨어의 도움도 받을 수 있다.

4-6. 블랜딩

화면에 그려진 그림은 색상 버퍼라는 메모리에 저장된다. 그림이 이미 그려져 있는 상태에서 같은 위치에 다른 그림을 그리면 새 그림을 메모리에 기록하므로 이전에 그려져 있던 그림은 덮여서 지워진다. 이 당연한 현상도 블랜딩 모드를 변경하면 달라질 수 있다. 블랜딩은 색상 버퍼에 이미 기록되어 있는 값과 새로 그려지는 값의 논리 연산 방법을 지정한다.

다른 프로그래밍 언어에서 ROP 모드(Raster Operation)라고 흔히 부르는 연산이되 그보다는 훨씬 더 상세하다. 디폴트 모드가 단순한 복사이므로 이전 그림이 지워지지만 다른 모드를 사용하면 두 값을 논리적으로 연산한 결과를 써 넣음으로써 특이한 효과를 낼 수 있다. 블랜딩 기능을 사용하라면 다음 명령으로 이 기능을 켜야 한다.

```
glEnable(GL_BLEND);
```

블랜딩은 색상 버퍼에 이미 기록되어 있는 값 D와 새로 기록되는 값 S와의 연산을 정의한다. 연산 방법은 다음 두 함수로 지정한다.

void glBlendFunc(GLenum sfactor, GLenum dfactor);

void glBlendEquation(GLenum mode);

sfactor와 dfactor는 S색상과 D색상에 각각 적용할 연산식을 정의하며 mode는 두 연산 결과를 합칠 방법을 정의한다. 모드에 따른 연산식은 다음과 같다. 디폴트는 두 연산식을 더하는 GL_FUNC_ADD이다.

모드	연산식
GL_FUNC_ADD	S*SF + D*DF. 이 모드가 디폴트이다.
GL_FUNC_SUBTRACT	S*SF - D*DF
GL_FUNC_REVERSE_SUBTRACT	D*DF -S*SF
GL_MIN	S*SF, D*DF 중 작은 값
GL_MAX	S*SF, D*DF 중 큰 값

S와 D에 적용되는 연산식의 종류는 다음과 같다. 이해를 돕기 위해 잘 사용되지 않는 일부 항을 생략했는데 더 정확한 식은 레퍼런스를 보기 바란다.

연산식	색상(FR, FG, FB, FA)
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_SRC_COLOR	(RS, GS, BS, AS)
GL_ONE_MINUS_SRC_COLOR	(1-RS, 1-GS, 1-BS, 1-AS)
GL_DST_COLOR	(RD, GD, BD, AD)
GL_ONE_MINUS_DST_COLOR	(1-RD, 1-GD, 1-BD, 1-AD)
GL_SRC_ALPHA	(AS, AS, AS, AS)
GL_ONE_MINUS_SRC_ALPHA	(1-AS, 1-AS, 1-AS, 1-AS)
GL_DST_ALPHA	(AD, AD, AD, AD)
GL_ONE_MINUS_DST_ALPHA	(1-AD, 1-AD, 1-AD, 1-AD)
GL_CONSTANT_COLOR	(RC, GC, BC, AC)
GL_ONE_MINUS_CONSTANT_COLOR	(1-RC, 1-GC, 1-BC, 1-AC)
GL_CONSTANT_ALPHA	(AC, AC, AC, AC)
GL_ONE_MINUS_CONSTANT_ALPHA	(1-AC, 1-AC, 1-AC, 1-AC)
GL_SRC_ALPHA_SATURATE	(i,i,i,1) i= min(AS, 1-AD)

연산식에 의해 R, G, B, A 색상 요소 각각에 곱해지는 FR, FG, FB, FA 함수가 정의되고 이 함수가 각 색상 요소에 적용됨으로써 중간식이 생성되며 두 중간식을 연산하여 최종 색상을 도출한다. 색상 요소가 아닌 상수와도 연산을 하는데 이때 사용할 상수는 다음 함수로 지정한다.

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

디폴트 상수는 (0,0,0,0)인 검정색이다. 다음 함수는 좀 더 상세한 연산 방법을 지정한다. GLBlendFunc는 RGB 색상 요소와 알파 요소를 같이 연산하는데 비해 이 함수는 두 요소에 대해 각각 다른 블렌딩 함수를 지정한다.

void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha);
void glBlendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);

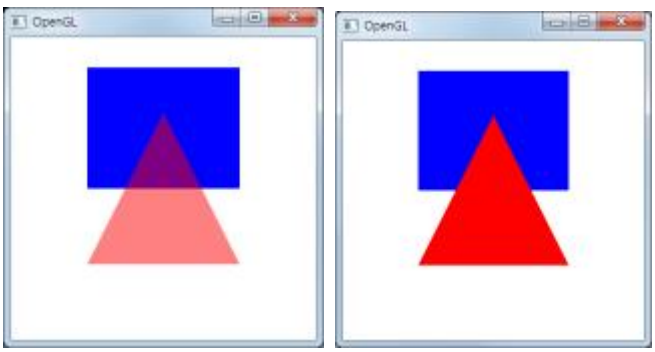
지정 가능한 연산의 종류는 동일하다. 계산식이 좀 더 복잡해지지만 대신 더 다양한 기교를 부릴 수 있다. 블랜딩은 설명만으로 이해하기는 어려우므로 예제를 분석해 보자.

Blend

```
#nclude <windows.h>
#nclude <gl/glut.h>

void DoDisplay();
void DoKeyboard(unsigned char key, int x, int y);
void DoMenu(int value);

GLfloat Alpha = 0.5f;
GLenum Src = GL_SRC_ALPHA;
GLenum Dest = GL_ONE_MINUS_SRC_ALPHA;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutKeyboardFunc(DoKeyboard);
      glutCreateMenu(DoMenu);
      glutAddMenuEntry("Opaque",1);
      glutAddMenuEntry("Traslucent",2);
      glutAttachMenu(GLUT_RIGHT_BUTTON);
      glutMainLoop();
      return 0;
}

void DoKeyboard(unsigned char key, int x, int y)
{
      switch(key) {
      case 'q':
            Alpha += 0.1;
            break;
      case 'a':
            Alpha -= 0.1;
            break;
      }
      glutPostRedisplay();
}

void DoMenu(int value)
{
      switch(value) {
      case 1:
            Src = GL_ONE;
            Dest = GL_ZERO;
            break;
      case 2:
            Src = GL_SRC_ALPHA;
            Dest = GL_ONE_MINUS_SRC_ALPHA;
            break;
      }
      glutPostRedisplay();
}
```

```
void DoDisplay()
{
	glClearColor(1,1,1,1);
	glClear(GL_COLOR_BUFFER_BIT);

	glEnable(GL_BLEND);
	glBlendFunc(Src, Dest);

	glColor3f(0, 0, 1);
	glRectf(-0.5, 0.8, 0.5, 0.0);

	glColor4f(1, 0, 0, Alpha);
	glBegin(GL_TRIANGLES);
	glVertex2f(0.0, 0.5);
	glVertex2f(-0.5, -0.5);
	glVertex2f(0.5, -0.5);
	glEnd();
	glFlush();
}
```

파란색 사각형을 그리고 그 위에 빨간색 삼각형을 그렸다. 투명한 블랜딩 모드를 사용했으므로 삼각형 뒤로 사각형이 흐릿하게 비친다. 키보드의 qa키로 알파값을 증가시켜 보면 비치는 정도가 달라진다. 팝업 메뉴에서 Opaque를 선택하면 디폴트 블랜딩 모드를 적용하여 불투명하게 출력된다.



이 예제에서 반투명 출력에 사용한 블랜드 연산식은 다음과 같다.

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

삼각형의 알파값이 0.4라고 했을 때 GL_SRC_ALPHA의 블랜딩 함수는 (AS, AS, AS, AS)이다. 그래서 S의 각 색상 요소에 0.4가 곱해진다. GL_ONE_MINUS_SRC_ALPHA의 블랜딩 함수는 모두 1-AS이므로 D의 각 색상 요소에 0.6이 곱해진다. 두 연산 결과를 연산하는 모드는 디폴트인 GL_FUNC_ADD이므로 두 값을 더해 최종 색상을 결정한다.

(0.4 * 1 + 0.6 * 0, 0.4 * 0 + 0.6 * 0, 0.4 * 0 + 0.6 * 1, 0.4 * 0.4 + 0.6 * 0.6)

(0.4 * (1,0,0,0.4) + 0.6 * (0,0,1,0.6))

0.4 * 빨간색 + 0.6 * 파란색

수식으로 표현하면 복잡해 보이는데 말로 설명하면 S색 40%, D색 60%를 섞어서 새로 기록한다는 뜻이다. 그래서 두 색상이 반쯤 섞여서 출력되며 뒤쪽 색상이 적당히 비쳐 보이는 것이다. 불투명 모드일 때는 다음 블

랜드 연산식을 사용한다.

```
glBlendFunc(GL_ONE, GL_ZERO);
```

이 식은 아주 쉽다. S의 모든 색상 요소에 1을 곱한다는 것은 곧 S의 색상을 그대로 유지한다는 뜻이고 D의 모든 색상 요소에 0을 곱한다는 것은 D의 색상을 완전히 무시한다는 뜻이다. 그러므로 D 색상은 S에 덮여서 안 보이게 되는 것이다.

블랜딩 모드는 응용의 묘미가 있는 기술이다. 연산식을 잘 조합하며 유리창에 흐릿하게 비치는 상태나 거울에 반사된 모양을 그릴 수 있다. 정확하게 원하는 효과를 내기 위해서는 많은 연습과 테스트가 필요하다.

4-7. 안티 알리아싱

컴퓨터 화면은 디지털이어서 해상도가 웬만큼 높아도 실세계의 장면들보다는 자연스럽지 못하다. 특히 색상 경계가 뚜렷할수록 경계면이 어색해서 이질감이 더해 보인다. 디지털 화면에서 나타나는 계단 현상 등을 알리아스라고 하며 이런 현상을 제거 또는 감소시키는 기술을 안티 알리아싱(Anti Aliasing)이라고 한다.

알리아스는 여러 가지 원인으로 인해 발생하는데 너무 뚜렷한 색상차가 주된 원인이다. 알리아스를 제거하려면 두 색상의 경계면에 중간색을 삽입하는 기법이 흔히 사용된다. 예를 들어 흰색과 검정색 사이에 회색을 단계적으로 삽입하는 식이다. 이 기능은 블랜딩 연산을 사용하므로 블랜딩 기능을 켜야 한다. 그리고 다음 함수로 점, 선, 다각형에 대해 안티 알리아싱을 적용한다.

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);
```

블랜딩을 켜고 알리아싱 기능을 켜 놓으면 OpenGL이 알아서 알리아스를 제거해 준다. 물론 추가 연산을 해야 하므로 시간은 좀 더 걸린다. 다각형에 대한 안티 알리아싱은 일부 플랫폼에서는 제대로 지원되지 않는다.

컴퓨터의 세계에서 속도와 품질은 항상 반비례 관계에 있다. 속도를 내려면 품질을 희생해야 하고 고품질을 얻으려면 시간이 오래 걸릴 수밖에 없다. 둘 다 좋을 수는 없으므로 개발자는 둘 중 어떤 것에 더 중점을 둘 것인지를 선택해야 한다. 다음 함수는 OpenGL 라이브러리에게 무엇을 더 우선시할 것인지 힌트를 제공한다.

```
void glHint(GLenum target, GLenum mode);
```

target은 옵션 조정의 대상이고 mode는 옵션을 어떻게 조정할 것인가를 지정한다. 조정 가능한 옵션 목록은 다음과 같다. 의미는 대부분 이름으로부터 쉽게 알 수 있도록 되어 있다.

```
GL_FOG_HINT
GL_GENERATE_MIPMAP_HINT
GL_LINE_SMOOTH_HINT
GL_PERSPECTIVE_CORRECTION_HINT
GL_POINT_SMOOTH_HINT
GL_POLYGON_SMOOTH_HINT
GL_TEXTURE_COMPRESSION_HINT
GL_FRAGMENT_SHADER_DERIVATIVE_HINT
```

각 타겟에 대해 mode로 힌트를 준다. 속도가 최우선일 때는 GL_FASTEST 모드를 지정하고 품질이 중요할 때는 GL_NICEST로 지정한다. 어느 것이나 상관없다면 GL_DONT_CARE로 지정하며 이 값이 디폴트이다. 그래서 별다른 지정이 없으면 OpenGL이 지 맘대로 속도와 품질 중 하나를 선택한다.

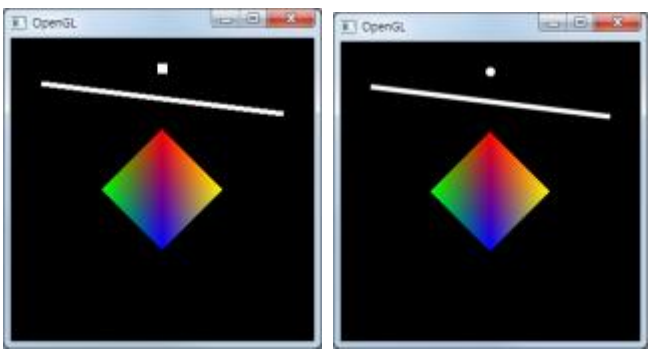
힌트는 강제적인 명령이 아니며 어디까지나 특정 기능이 어떤 식으로 구현되었으면 좋겠다는 희망 사항을 밝히는 것 뿐이어서 반드시 지정한대로 동작한다는 법은 없다. 힌트를 실제 그리기에 적용할 것인가 아닌가는 드라이버가 결정한다. 드라이버의 능력이 되고 상황이 허락한다면 요청을 받아줄 것이고 그렇지 않다면 무시해 버린다. 다음 예제는 안티알리아싱과 힌트 기능을 테스트한다.

AntiAlias

```
#nclude <windows.h>
#nclude <gl/glut.h>

void DoDisplay();
void DoMenu(int value);
BOOLEAN bAlias;
BOOLEAN bHint;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutCreateMenu(DoMenu);
      glutAddMenuEntry("Alias ON",1);
      glutAddMenuEntry("Alias OFF",2);
      glutAddMenuEntry("Hint ON",3);
      glutAddMenuEntry("Hint OFF",4);
      glutAttachMenu(GLUT_RIGHT_BUTTON);
      glutMainLoop();
      return 0;
}

void DoMenu(int value)
{
      switch(value) {
      case 1:
            bAlias = TRUE;
            break;
      case 2:
            bAlias = FALSE;
            break;
      case 3:
            bHint = TRUE;
            break;
      case 4:
            bHint = FALSE;
            break;
      }
      glutPostRedisplay();
}

void DoDisplay()
{
      glClear(GL_COLOR_BUFFER_BIT);

      // 블랜딩이 켜져 있어야 알리아싱이 제대로 된다.
      glEnable(GL_BLEND);
      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

      // 안티 알리아싱 on, off
      if (bAlias) {
```

```
            glEnable(GL_POINT_SMOOTH);
            glEnable(GL_LINE_SMOOTH);
            glEnable(GL_POLYGON_SMOOTH);
        } else {
            glDisable(GL_POINT_SMOOTH);
            glDisable(GL_LINE_SMOOTH);
            glDisable(GL_POLYGON_SMOOTH);
        }

        // 고품질 출력을 위한 힌트
        glHint(GL_POINT_SMOOTH_HINT, bHint ? GL_NICEST:GL_FASTEST);
        glHint(GL_LINE_SMOOTH_HINT, bHint ? GL_NICEST:GL_FASTEST);
        glHint(GL_POLYGON_SMOOTH_HINT, bHint ? GL_NICEST:GL_FASTEST);

        glPointSize(10.0);
        glColor3f(1,1,1);
        glBegin(GL_POINTS);
        glVertex2f(0.0, 0.8);
        glEnd();

        glLineWidth(5);
        glBegin(GL_LINE_STRIP);
        glVertex2f(-0.8, 0.7);
        glVertex2f(0.8, 0.5);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f(1,0,0);
        glVertex2f(0.0, 0.4);
        glColor3f(0,1,0);
        glVertex2f(-0.4, 0.0);
        glColor3f(0,0,1);
        glVertex2f(0.0, -0.4);
        glColor3f(1,1,0);
        glVertex2f(0.4, 0.0);
        glEnd();

        glFlush();
    }
```

검정색 바탕에 흰색으로 점과 선을 아주 굵게 그려 놓았으며 마름모는 4가지 색상으로 화려하게 그렸다. 팝업 메뉴로 안티 알리아싱과 힌트 기능을 토글해 보고 출력 결과가 어떻게 달라지는지 관찰해 보자.



안티 알리아싱을 적용하지 않은 상태에서는 점이 사각형으로 보이고 직선도 계단 현상이 심해 눈에 거슬린다. 검정 바탕에 흰색이다 보니 부자연스러움이 훨씬 더 심해 보인다. 다각형은 원색이라 그래도 덜 어색하다.

안티 알리아싱 기능을 켜면 점은 동그랗게 보이고 계단 현상도 훨씬 덜해 보인다. 돋보기로 화면을 확대해 보면 회색점이 흐릿하게 삽입되어 있는 것을 볼 수 있다. 다각형에 대해서는 안티 알리아싱이 제대로 지원되지 않는다. 힌트 기능도 토글할 수 있도록 해 두었는데 그림이 너무 단순해서 힌트의 효과를 실감하기 어렵다.