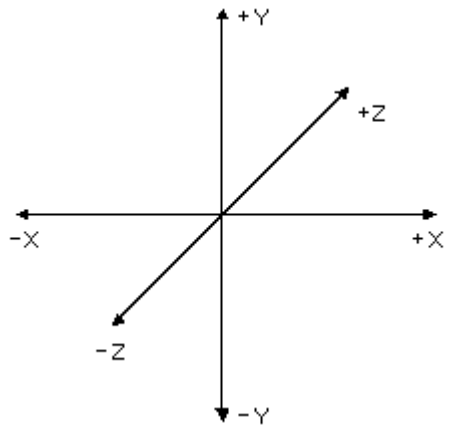
6.입체 도형

6-1.3차원 좌표

지금까지 우리는 z축 좌표를 한번도 사용한 적이 없으며 2차원 평면에서 납작한 도형만 그려 보았다. 정점의 x, y 좌표만 지정하면 z는 항상 디폴트인 0으로 평가되며 그래서 평면 도형으로 출력된 것이다. 이제 드디어 z 좌표를 지정하여 3차원 공간에서 입체적인 도형을 그려 보자. OpenGL의 3차원 공간은 다음과 같은 좌표계로 정의되어 있다.



3차원 좌표 공간이므로 축이 3개 있다. x 축은 오른쪽으로 증가하고 y축은 위쪽으로 증가하며 z축은 사용자 반대쪽으로 증가한다. 클리핑 영역, 즉 좌표의 범위는 x, y, z 모두 -1 ~ 1 사이이다. 원점은 (0, 0, 0)이고 사용자 가까운 쪽의 왼쪽 아래는 (-1, -1, -1)이고 사용자와 먼쪽의 오른쪽 위는 (1, 1, 1)이다.

이 좌표 공간에서 주의할 점은 이것은 어디까지나 디폴트일 뿐이라는 것이지 항상 그런 것은 아니라는 점이다. 클리핑 영역을 지정하면 좌표의 범위와 증가 방향 등을 원하는대로 설정할 수 있다. -1 ~ 1 사이의 실수값으로 좌표를 지정하는 것이 불편하면 왼쪽을 -1000, 오른쪽을 1000으로 정의해 두고 정수값으로 좌표를 지정할 수도 있으며 상화좌우의 증가 방향을 반대로 뒤집을 수도 있다.

다음 예제는 3차원 공간안에 피라미드를 그린다. 피라미드는 밑면이 사각형이고 각 변에서 하늘쪽으로 솟은 4개의 삼각형으로 구성된다. 입체로 만들 수 있는 모양중에 거의 제일 단순한 모양인 셈이다. 더 단순하게는 4개의 삼각형으로 구성된 4면체가 있지만 밑몉과 옆면이 모두 삼각형이라 잘 구분되지 않는다. 그래서 5면체인 피라미드 모양을 사용하기로 한다.

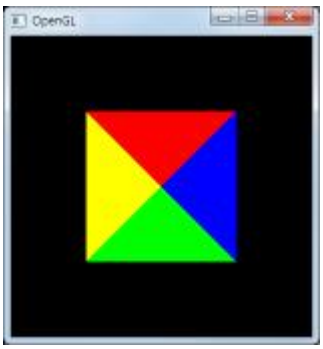
Pyramid

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

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

GLfloat xAngle, yAngle, zAngle;
GLboolean bDepthTest = GL_TRUE;
GLboolean bCullFace = GL_FALSE;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
```

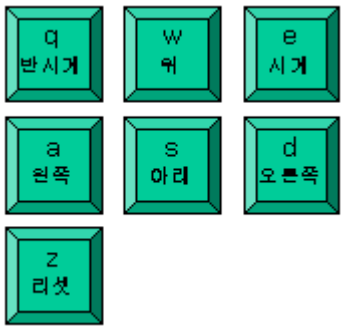
```
{
	glutCreateWindow("OpenGL");
	glutDisplayFunc(DoDisplay);
	glutKeyboardFunc(DoKeyboard);
	glutCreateMenu(DoMenu);
	glutAddMenuEntry("Depth Test ON",1);
	glutAddMenuEntry("Depth Test OFF",2);
	glutAddMenuEntry("Cull Face ON",3);
	glutAddMenuEntry("Cull Face OFF",4);
	glutAttachMenu(GLUT_RIGHT_BUTTON);
	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:
		bDepthTest = GL_TRUE;
		break;
	case 2:
		bDepthTest = GL_FALSE;
		break;
	case 3:
		bCullFace = GL_TRUE;
		break;
	case 4:
		bCullFace = GL_FALSE;
		break;
	}
	glutPostRedisplay();
}

void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glShadeModel(GL_FLAT);

	if (bDepthTest) {
		glEnable(GL_DEPTH_TEST);
	} else {
		glDisable(GL_DEPTH_TEST);
	}
```

```
    if (bCullFace) {
        glEnable(GL_CULL_FACE);
    } else {
        glDisable(GL_CULL_FACE);
    }

    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
    glColor3f(1,1,1);
    glBegin(GL_QUADS);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
    glVertex2f(-0.5, -0.5);
    glEnd();

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

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

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

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

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

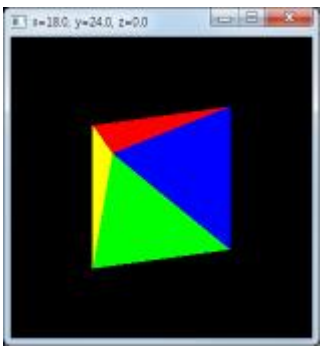
피라미드의 각면을 쉽게 알아 볼 수 있도록 좀 유치찬란하지만 일부러 각각 다른 원색을 칠해 두었다. 입체 도형을 그리는데 필요한 몇 가지 옵션들이 팝업 메뉴에 정의되어 있는데 다음항에서 천천히 연구해 보자. 디폴트 옵션대로 그리면 피라미드가 제대로 그려진다.



분명히 피라미드인 것 같은데 z축의 정면에서 아래쪽으로 바라 보고 있는 상황이라 무슨 딱지 모양처럼 보인다. 아무리 3차원 물체를 그렸더라도 이 물체를 보여줄 모니터가 2차원 평면이므로 어쩔 수가 없다. 모니터 상에서 이 물체가 과연 입체적인 피라미드가 맞는지 확인하려면 요리 조리 돌려 봐야 한다. 회전을 시킨다는 것은 x, y 축에 시간축을 더하는 것이므로 구성이 다르기는 하지만 3차원으로 보는 셈이다. 그래서 키보드로 회전시키는 기능을 작성해 두었다.



왼손으로 회전하기 편하도록 키를 배치했다. 타이틀 바에는 현재 회전 각도를 표시해 두었으며 z키를 누르면 언제든지 리셋한다. DoKeyboard 함수의 회전과 관련된 코드는 어디까지나 입체임을 확인하기 위해 작성한 코드이므로 당분간은 무시하자. 이 키들은 나머지 예제들에도 일관되게 적용해 두었으므로 언제든지 확인을 위해 회전시켜 보도록 하자. a키와 w키를 눌러 왼쪽 위로 살짝 회전시켜 보면 과연 피라미드임을 확인할 수 있다.



그리기 코드를 보자. 아래쪽에 깔리는 흰색 사각형은 z 좌표를 지정하지 않았으므로 z가 0인 평면에 그려진다. 빨감색 삼각형의 첫 꼭지점 좌표는 (0, 0, -0.8)로 되어 있다. 밑면의 정중앙에서 사용자쪽으로 0.8만큼 위쪽으로 솟은 정점을 지정했으며 나머지 두 변은 밑면의 우상, 좌상점을 지정했다. 그래서 삼각형이 밑면에서 비스듬히 솟은 모양으로 배치된다.

GL_TRIANLGE_FAN 모드로 그리고 있으므로 나머지 삼각형들은 모두 중앙의 꼭지점을 공유하며 밑면의 각 두 변과 연결하는 삼각형이 된다. 밑면에 흰 바닥 사각형 놓고 위쪽부터 반시계 방향으로 빨, 노, 초, 파 삼

각형을 비스듬히 얹어 피라미드 모양을 완성했다. 빨간색 삼각형의 첫 꼭지점인 피라미드 꼭대기 외에는 모두 평면상의 좌표이므로 x, y만 지정하면 된다.

이 예제에서 피라미드의 꼭대기 z 축 좌표가 -0.8인 것은 상당히 헷갈리는 부분이다. 디폴트 클리핑은 사용자쪽 좌표가 1, 멀어지는 좌표가 -1로 정의되어 있어 0.8로 지정해야 할 듯 하지만 이 두 값은 음수로 평가되기 때문에 사용자쪽이 음수라고 한다. 어디까지나 디폴트가 그럴 뿐이어서 클리핑을 바꾸면 사용자쪽을 양수로 할 수도 있다. 솔직히 이 부분은 왜 그런지 아직 정확하게 설명할 능력이 안된다. 열심히 스펙 문서를 읽었는데도 시원스런 설명을 찾지 못했으며 공부가 더 필요한 부분이다.

6-2.깊이 테스트

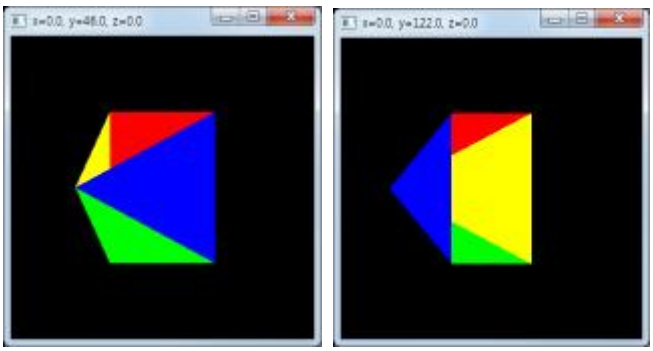
피라미드 예제는 입체를 그리기 위해 깊이 테스트 기능을 사용한다. 깊이 테스트 기능을 사용하려면 glutInitDisplayMode 함수로 디스플레이 모드를 설정할 때 GLUT_DEPTH 플래그를 지정해야 한다. 이 플래그가 있어야 각 픽셀의 깊이값을 저장할 수 있는 깊이 버퍼가 생성된다.

다행히 이 플래그는 디폴트로 선택되어 있으므로 일부러 빼지 않는한 따로 설정하지 않아도 깊이 버퍼는 자동으로 생성된다. 디스플레이 모드에 깊이 버퍼는 디폴트로 선택되어 있지만 깊이 테스트 기능은 디폴트로 꺼져 있다. 그래서 다음 호출문으로 깊이 테스트 기능을 켜야 기능이 동작한다.

```
glEnable(GL_DEPTH_TEST);
```

예제에서는 bDepthTest 변수값에 따라 이 기능을 켜고 끌 수 있도록 해 두었으며 팝업 메뉴를 통해 이 변수값을 토글할 수 있다. bDepthTest의 초기값은 GL_TRUE여서 실행 직후부터 깊이 테스트를 수행한다. 그렇다면 깊이 테스트는 과연 어떤 기능이며 이 기능이 없으면 무슨 문제가 발생하는지 테스트해 보자.

팝업 메뉴에서 Depth Test OFF 항목을 선택하여 깊이 테스트 기능을 끄고 a키를 눌러 왼쪽으로 살짝쿵 돌려 보자. 노란색 면이 가려질 때까지 돌리고 그 상태에서 조금 더 돌리면 사라졌던 노란면이 빨간면 위에 그려진다. 아예 180도 회전시켜 보면 흰색 밑면은 보이지도 않는다. 마치 피라미드 아래면이 뻥 뚫린 것처럼 보인다.



이렇게 보이는 이유는 OpenGL이 순서대로 도형을 그리기 때문이다. 코드를 보면 흰색 밑면을 먼저 그리고 빨노초파순으로 면을 그리도록 되어 있다. 그러다 보니 먼저 그려진 도형이 나중에 그려지는 도형에 의해 지워지는 것이다. 빨간색 면을 먼저 그리고 노란색 면을 그리다 보니 나중에 그려지는 노란면이 빨간면을 가리는 것이다. 노란면의 일부는 더 나중에 그려지는 초록면과 파란면에 의해 일부가 가려진다.

제일 먼저 그려지는 흰면은 나머지 빗변에 전부 가려지므로 아예 보이지도 않는다. 밑면이 없으므로 회전시킬 때 피라미드 꼭대기가 지금 앞쪽에 있는건지 뒤쪽에 있는건지도 잘 분간되지 않는다. 한쪽으로 계속 회전시켜 보면 왼쪽으로 돌고 있는 것 같기도 하고 오른쪽으로 돌고 있는 것 같기도 하고 무진장 헷갈릴 것이다.

이런 현상을 방지하려면 그리는 순서보다 도형의 아래 위를 따져야 한다. 어떤 면이 더 위쪽에 있는지 즉, 사용자의 시선과 가까운지를 판별하여 순서에 상관없이 더 위쪽에 있는 면이 보이도록 해야 한다. 나중에 그

려지는 도형이라도 사용자 시선보다 더 먼쪽이면 그리지 말아야 한다. 이런 판별을 하려면 화면상의 모든 점에 대해 깊이갚을 버퍼에 따로 저장하고 그릴 때 각 점의 깊이를 비교해야 한다.

이 테스트를 깊이 테스트라고 하며 2차원 그래픽과 3차원 그래픽의 주요한 차이점이기도 하다. 2차원 그래픽은 그리는 순서만으로 앞뒤를 분간할 수 있다. 뒤쪽에 있는 물체를 먼저 그리고 앞쪽에 있는 물체를 나중에 그리면 아무 문제가 없다. 예를 들어 참새가 들판을 날아가는 장면이라면 들판을 먼저 그리고 참새를 나중에 그리면 아무 문제가 없다.

그러나 3차원 그래픽은 모든 물체가 입체적이므로 순서만으로는 앞뒤를 정확히 분간하기 어렵다. 손가락에 끼워진 반지의 경우 반지가 손가락을 에워싸고 있으므로 어느 물체가 더 앞쪽인지 명확하지 않다. 두 물체의 순서가 순환적이라 앞뒤 분간을 할 수 없으며 어떤 것을 먼저 그리더라도 일부가 가려질 수밖에 없다.

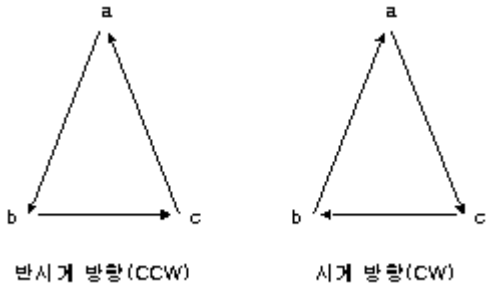
그래서 모든 화소의 깊이값을 별도의 버퍼에 따로 저장해 두고 매 화소를 그릴 때마다 이 화소가 출력 대상인지 판단해야 한다. 도형의 모든 점에 대해 깊이 정보가 필요하므로 정점의 깊이값이 아니라 화소의 깊이값을 저장해야 한다. 그래서 OpenGL은 별도의 깊이 버퍼를 관리하고 깊이값을 참조하여 물체를 그린다. 깊이 버퍼는 색상 버퍼와는 다른 완전히 분리된 메모리 영역이다. 그래서 화면을 삭제할 때 색상 버퍼외에 깊이 버퍼도 같이 삭제해야 한다. DoDisplay 함수 선두의 glClear문을 보면 다음과 같이 되어 있다.

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

색상 버퍼뿐만 아니라 깊이 버퍼도 같이 지운다. 깊이 버퍼를 초기화하지 않으면 쓰레기값으로 가득차므로 그려져야 할 물체가 화면에 나타나지 않을 것이다. 색상 버퍼를 지우지 않으면 쓰레기 그림이 출력되는 것과 마찬가지이다.

6-3.와인딩 모드

도형은 여러 개의 정점으로 구성된다. 이때 도형을 구성하는 정점의 순서를 와인딩(Winding)이라고 한다. 똑같은 삼각형을 그리더라도 정점을 시계 방향으로 나열할 수도 있고 반시계 방향으로 나열할 수도 있다.



둘 다 3개의 꼭지점으로 구성되어 있는 삼각형인데 abc 순으로 돌 수도 있고 acb순으로 돌 수도 있다. 어쨌거나 세 정점을 연결하여 삼각형을 그리므로 결과는 같다. 그러나 방향에 따라 도형의 앞 뒤가 달라진다. 디폴트 와인딩은 반시계 방향으로 정의되어 있으며 관찰자가 보기에 반시계 방향으로 그려진 면이 도형의 앞으로 간주된다.

위 그림에서 반시계 방향으로 그려진 왼쪽 삼각형은 앞면이 보이는 상태이고 시계 방향으로 그려진 오른쪽 삼각형은 뒷면이 보이는 상태이다. 반대쪽에서 보면 둘 다 방향이 바뀌고 앞면과 뒷면도 자연스럽게 뒤집어진다. 3D 그래픽은 회전이나 시점 변경이 가능해서 어디서 보더라도 앞뒷면이 일관된 방법이 필요한데 그것이 바로 와인딩이다. 와인딩 모드는 다음 함수로 지정한다.

void glFrontFace(GLenum mode);

인수로 GL_CW를 주면 시계 방향이 앞면이고 GL_CCW를 주면 반시계 방향이 앞면이다. 항상 한 방향으로만 그리면 일관되지만 다른 프로그램에서 가져온 데이터를 조합할 때는 수시로 방향을 바꿀 필요도 있다. 그래서 필요에 따라 GL_CW, GL_CCW를 선택할 수 있다. 디폴트는 반시계 방향이 앞면이다.

그렇다면 앞면, 뒷면이 왜 중요할까? 어차피 정점을 연결해서 채우면 똑같이 그려진다. 하지만 앞 뒤에 각각 다른 색상을 칠한다거나 조명을 비출 때는 앞뒤구분이 중요해진다. 또 뒷면을 생략하는 최적화 기법을 적용할 수 있는데 이럴 때는 어디가 생략 가능한 면인지를 분명히 해야 한다.

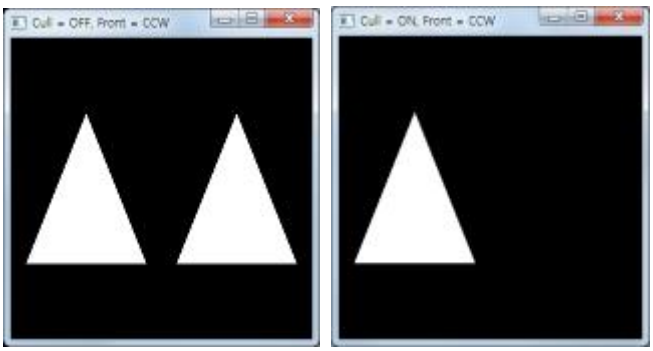
뒷면이 다른 도형에 완전히 둘러싸여 보이지 않는다면 그리기를 생략함으로써 속도를 대폭적으로 향상시킬 수 있다. 불필요한 그리기를 하지 않는 이런 기법을 컬링(Culling)이라고 한다. 그리되 가려져서 안 보이는 것과 아예 그리지 않는 것은 완전히 다르며 속도 차이가 엄청나게 벌어진다. 다음 예제는 와인딩 모드와 컬링을 테스트한다.

Culling

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

void DoDisplay();
void DoMenu(int value);

GLboolean bCullFace = GL_FALSE;
GLboolean bCcwFront = GL_TRUE;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutCreateMenu(DoMenu);
      glutAddMenuEntry("CullFace ON",1);
      glutAddMenuEntry("CullFace OFF",2);
      glutAddMenuEntry("CCW",3);
      glutAddMenuEntry("CW",4);
      glutAttachMenu(GLUT_RIGHT_BUTTON);

      glutMainLoop();
      return 0;
}

void DoMenu(int value)
{
      switch(value) {
      case 1:
            bCullFace = GL_TRUE;
            break;
      case 2:
            bCullFace = GL_FALSE;
            break;
      case 3:
            bCcwFront = GL_TRUE;
            break;
      case 4:
            bCcwFront = GL_FALSE;
            break;
      }
      char info[128];
```

```
	sprintf(info, "Cull = %s, Front = %s", bCullFace ? "ON":"OFF", bCcwFront ? "CCW":"CW");
	glutSetWindowTitle(info);
	glutPostRedisplay();
}

void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	if (bCullFace) {
		glEnable(GL_CULL_FACE);
	} else {
		glDisable(GL_CULL_FACE);
	}
	glFrontFace(bCcwFront ? GL_CCW:GL_CW);

	// CCW
	glBegin(GL_POLYGON);
	glVertex2f(-0.5, 0.5);
	glVertex2f(-0.9, -0.5);
	glVertex2f(-0.1, -0.5);
	glEnd();

	// CW
	glBegin(GL_POLYGON);
	glVertex2f(0.5, 0.5);
	glVertex2f(0.9, -0.5);
	glVertex2f(0.1, -0.5);
	glEnd();

	glFlush();
}
```

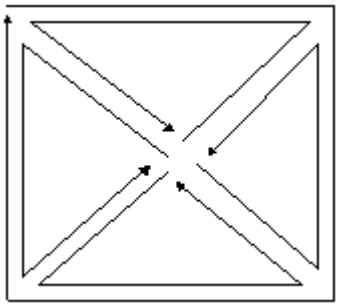
두 개의 삼각형을 나란히 그리되 왼쪽은 반시계, 오른쪽은 시계 방향으로 그렸다. 컬링은 적용하지 않았으므로 일단은 두 삼각형이 똑같은 모양으로 보인다.



팝업 메뉴에서 컬링과 와인딩 모드를 토글해 보자. 후면에 대해 컬링을 적용하려면 GL_CULL_FACE 기능을 켠다. 이 기능을 켜면 앞면이 보이는 도형만 그리므로 왼쪽 삼각형만 남고 오른쪽 삼각형은 보이지 않는다. 와인딩 모드를 시계 방향으로 바꾸면 이번에는 오른쪽 삼각형이 앞면을 보고 있는 것이므로 왼쪽 삼각형이 사라진다. 컬링 기능을 끄면 방향에 상관없이 둘 다 그려진다.

Pyramid 예제에도 컬링 기능이 들어 있다. 이 예제는 컬링이 어떤 효과가 있는지를 좀 더 분명하게 보여준다. 일단 컬링 기능을 사용하지 않도록 해 두었는데 이 기능을 사용하나 마나 출력 결과는 동일하다. 그러나 삼각뿔 안쪽의 안 보이는 면을 불필요하게 그릴 필요는 없으므로 컬링 기능을 사용하면 그리기 속도가 훨씬

빨라진다. 너무 단순한 도형이라 차이를 체감하기는 어렵지만 최소한 2배 이상 빨라질 것이다. 이 예제의 도형들은 다음과 같은 와인딩으로 그려져 있다.



GL_TRIANGLE_FAN 모드는 항상 반시계 방향을 유지하며 윗변 삼각형 4개는 모두 반시계 방향으로 그려져 있다. 그래서 관찰자가 바라보면 쪽이 앞면이다. 반면 아래쪽의 흰색 밑면은 시계 방향으로 되어 있다. 왜냐하면 관찰자가 바라보는 쪽이 뒷면이고 피라미드를 뒤집었을 때 보이는 면이 앞면이기 때문이다. 다섯면 모두 피라미드 속이 아닌 바깥쪽이 앞면으로 지정되어 있다.

모든 면의 와인딩이 합리적으로 설정되어 있으므로 이 물체에는 컬링을 적용할 수 있다. 컬링 기능을 켜면 가려진 안쪽 면을 그리지 않으므로 그리는 속도가 훨씬 더 빨라진다. 이 정도 도형으로 속도차를 체감하는 것은 불가능하지만 수만개의 삼각형으로 구성된 복잡한 도형에서는 확실히 속도차가 날 것이다.

컬링 기능을 켜면 깊이 테스트는 생략해도 상관없다. 왜냐하면 피라미드는 완전히 볼록한 물체여서 특정 면이 시야에서 사라지는 경우는 항상 뒷면이 관찰자쪽이기 때문이다. 노란면이 가려져서 빨간면을 덮을 각도가 되면 앞면이 관찰자로부터 멀어지고 뒷면이 관찰자쪽으로 보이므로 이면을 그릴 필요가 없다. 그래서 깊이 테스트를 하지 않더라도 빨간면이 노란면에 가리지 않는다.

180도 돌렸을 때도 마찬가지이다. 관찰자쪽으로 밑면의 앞면이 보이고 나머지 빗면들은 모두 뒷면쪽으로 돌아서 있기 때문에 흰색 밑면이 가려지지 않는다. 만약 와인딩이 하나라도 잘못되어 있다면 이런 기법은 사용할 수 없으며 엉뚱한 결과가 나타난다. 아래쪽 흰 변의 정점 순서를 반시계 방향으로 잠시 바꿔 보아라.

```
// 아랫면 흰 바닥
glColor3f(1,1,1);
glBegin(GL_POLYGON);
glVertex2f(-0.5, 0.5);
glVertex2f(-0.5, -0.5);
glVertex2f(0.5, -0.5);
glVertex2f(0.5, 0.5);
glEnd();
```

순서를 바꾸면 와인딩이 바뀌고 앞뒷면이 뒤집어진다. 피라미드의 안쪽을 보고 있는 면이 앞면이 된다. 이 상태에서 컬링을 적용하면 바깥쪽이 그려지지 않으므로 밑면이 아예 보이지 않을 뿐만 아니라 빗변도 윗면에 가려 사라져 버린다. 비단 컬링 뿐만 아니라 조명이나 폴리곤 모드 등에도 앞뒷면 구분은 중요하다. 관찰자가 바라보는 면이 앞면이 되도록 정점의 순서에 항상 신경을 써야 한다.

6-4.폴리곤 모드

Pyramid 예제는 면간의 구분을 확실히 하기 위해 좀 유치하지만 5가지 원색을 사용했다. 이때 glColor 함수가 지정하는 색상은 정점에 대해 적용되는 것이지 면에 대해 적용되는 것이 아니다. 다각형 하나는 여러 개의 정점으로 구성된다. 각 정점의 색상이 제각각이면 과연 어떤 색상으로 어떻게 채색해야 할까. 다각형 채색 방식을 결정하는 것을 쉐이드 모델이라고 하며 다음 함수로 지정한다.

void glShadeModel(GLenum mode);

인수로 GL_FLAT과 GL_SMOOTH 둘 중 하나의 값을 전달한다. GL_FLAT은 대표 정점의 색상으로 다각형 전체를 균일하게 채운다. 어떤 정점의 색을 사용할 것인가는 다각형에 따라 다른데 GL_POLYGON인 경우는 첫 정점의 색상을 사용하고 나머지는 마지막 정점의 색상을 사용한다. 피라미드의 각면은 GL_QUADS와 GL_TRAINGLE_FAN 모드로 그렸으므로 마지막 정점의 색이 적용되었다.

GL_SMOOTH 모드인 경우는 모든 정점의 색상을 섞어서 사용한다. 각 정점은 지정한 색으로 채색되고 중간 지점은 정점에서 떨어진 거리만큼 부드럽게 섞인다. 노란색과 빨간색 정점이 있다면 두 정점의 중간은 주황색이 될 것이다. 쉽게 말해서 그라데이션 처리된다. 디폴트 쉐이드 모드는 GL_SMOOTH이되 Pyramid 예제는 GL_FALT 쉐이드 모드로 지정하여 단색으로 채색하였다. 다음 함수는 다각형의 폴리곤 모드를 지정한다.

void glPolygonMode(GLenum face, GLenum mode);

face 인수는 GL_FRONT, GL_BACK, GL_FRONT_AND_BACK 중 하나이며 앞뒷면 중 어느면에 대해 폴리곤 모드를 지정할 것인지를 지정한다. 앞뒤 각각 다른 폴리곤 모드를 적용할 수도 있고 두 면 모두 동일하게 지정할 수도 있다. 폴리곤 모드는 다음 세가지가 있다. 디폴트 폴리곤 모드는 앞뒷면 모두 GL_FILL이이서 내부를 가득 채운다.

폴리곤 모드	설명
GL_POINT	정점만 점으로 그린다. 점 크기는 GL_POINT_SIZE 설정을 따른다.
GL_LINE	정점끼리 선으로만 연결한다. 선 굵기와 스티플 설정을 따른다.
GL_FILL	면을 가득 채운다. 쉐이드 모델을 따른다.

쉐이드 모델과 폴리곤 모드(둘 다 설정이라는 면에서는 같은데 왜 하나는 모델이고 하나는 모드로 이름을 붙여 놨는지 모르겠다)는 둘 다 어렵지 않게 이해할 수 있다. 예제 만들어서 눈으로 차이점을 보기만 하면 바로 머리에 쏙쏙 들어올 것이다. 다음 예제는 피라미드를 그리되 팝업 메뉴로 두 기능을 토글해 가며 출력 결과가 어떻게 달라지는지 관찰한다.

PolygonMode

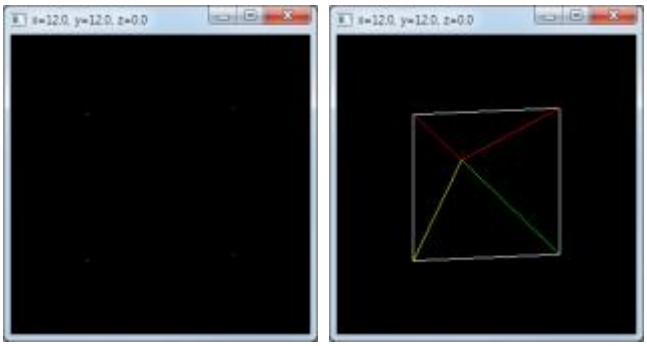
```
#include <windows.h>
#include <gl/glut.h>
#include <stdio.h>

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

GLfloat xAngle, yAngle, zAngle;
GLenum PolygonMode = GL_FILL;
GLenum ShadeMode = GL_FLAT;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutKeyboardFunc(DoKeyboard);
```

```
	glutCreateMenu(DoMenu);
	glutAddMenuEntry("Polygon POINT",1);
	glutAddMenuEntry("Polygon LINE",2);
	glutAddMenuEntry("Polygon FILL",3);
	glutAddMenuEntry("Smooth Shade",4);
	glutAddMenuEntry("Flat Shade",5);
	glutAttachMenu(GLUT_RIGHT_BUTTON);
	glutMainLoop();
	return 0;
}

void DoKeyboard(unsigned char key, int x, int y)
{
	=== 코드 생략 ===
}

void DoMenu(int value)
{
	switch(value) {
	case 1:
		PolygonMode = GL_POINT;
		break;
	case 2:
		PolygonMode = GL_LINE;
		break;
	case 3:
		PolygonMode = GL_FILL;
		break;
	case 4:
		ShadeMode = GL_SMOOTH;
		break;
	case 5:
		ShadeMode = GL_FLAT;
		break;
	}
	glutPostRedisplay();
}

void DrawPyramid()
{
	=== 코드 생략 ===
}

void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glEnable(GL_DEPTH_TEST);

	glPolygonMode(GL_FRONT_AND_BACK, PolygonMode);
	glShadeModel(ShadeMode);

	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);

	DrawPyramid();

	glPopMatrix();
	glFlush();
```

```
}
```

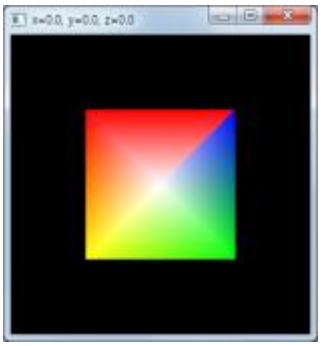
피라미드를 그리는 코드는 앞 예제에서 이미 살펴 보았으므로 DrawPyramid 함수로 따로 분리했다. 계속 반복되므로 지면을 낭비할 필요가 없을 뿐만 아니라 코드가 길어지면 핵심 코드가 잘 안 보이므로 정신 사나와진다. 같은 이유로 입체 도형을 회전시키는 키보드 입력 처리 루틴의 코드도 생략했다.

최초 실행 결과는 Pyramid 예제와 동일하도록 변수의 초기값을 설정했다. 각 면이 단색으로 채색될 것이다. 팝업 메뉴에서 폴리곤 모드를 바꾸면 점이나 선으로 도형을 그린다.



폴리곤 모드를 점으로 바꿀 경우는 거의 없지만 GL_LINE으로 바꾸는 경우는 종종 있다. 선만으로 그려진 와이어 프레임은 그리는 속도가 환상적이면서도 대충의 모양을 확인할 수 있으므로 디자인중에는 면을 다 채우는 것보다 오히려 더 실용적이다. 선으로만 구성된 물체라도 회전시켜 보면 피라미드라는 것을 쉽게 알 수 있다.

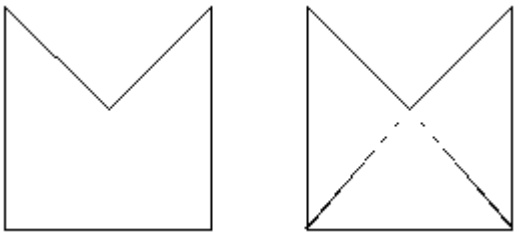
폴리곤 모드를 GL_FILL로 설정하고 쉐이드 모델을 GL_SMOOTH로 설정하면 면의 색이 그라데이션으로 부드럽게 채색된다. 정점에서 얼마나 가까이 있는가에 따라 중간 지점의 색이 결정된다.



피라미드를 그리는 코드를 다시 보면 꼭대기의 정점색을 흰색으로 지정하고 있다. 꼭대기가 흰색이어야 옆면이 각각 다른 색으로 부드럽게 섞이기 때문이다. 이 색상은 쉐이드 모델이 GL_SMOOTH일 때만 사용되며 GL_FLAT일 때는 무시된다. 아래쪽 밑면은 4개의 정점이 모두 흰색이므로 쉐이드 모델에 상관없이 항상 흰색 단일색이다.

6-5.에지 플래그

앞에서 설명했다시피 OpenGL의 규칙상 다각형은 반드시 볼록해야 하며 오목해서는 안된다. 그러나 현실적으로는 오목한 다각형을 그려야 할 경우도 있는데 이럴 때는 여러 개의 작은 다각형으로 분할해야 한다. 예를 들어 다음과 같이 주방장 모자 같은 모양을 그리고 싶다고 해보자.



모양이 오목해서 다각형 구성 규칙에 어긋나며 한번에 그릴 수가 없다. 이럴 때는 오른쪽처럼 삼각형 세개로 분할해서 그리면 된다. 폴리곤 모드가 GL_FILL이면 어차피 다 채워지므로 분할된 것인지 원래 하나인지 구분되지도 않는다. 그러나 GL_LINE인 경우는 분할된 안쪽의 선도 그려진다. 안쪽 선을 숨기려면 매 선분마다 외곽선인지 아니면 다른 도형을 구성하는 내부 선분인지 지정해야 한다. 다음 함수로 지정한다.

void glEdgeFlag(GLboolean flag);

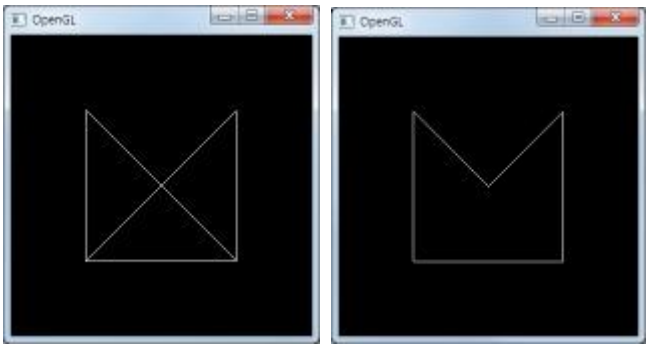
GL_TRUE이면 이후의 정점으로 이동하면서 그려지는 선은 외곽선으로 인식된다. GL_FALSE로 지정하면 큰 다각형을 구성하는 내부의 선으로 인식된다. 각 정점마다 외부의 선인지 외부의 선인지를 잘 구분해야 한다.

EdgeFlag

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

void DoDisplay();
void DoMenu(int value);
GLboolean bEdge = GL_FALSE;

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

void DoMenu(int value)
{
	switch(value) {
	case 1:
		bEdge = GL_TRUE;
		break;
	case 2:
		bEdge = GL_FALSE;
		break;
	}
	glutPostRedisplay();
}

void DoDisplay()
```

```
{
    glClear(GL_COLOR_BUFFER_BIT);

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glEdgeFlag(TRUE);

    glBegin(GL_TRIANGLES);
    if (bEdge) glEdgeFlag(TRUE);
    glVertex2f(0.0, 0.0);
    glVertex2f(-0.5, 0.5);
    if (bEdge) glEdgeFlag(FALSE);
    glVertex2f(-0.5, -0.5);

    glVertex2f(0.0, 0.0);
    if (bEdge) glEdgeFlag(TRUE);
    glVertex2f(0.5, -0.5);
    glVertex2f(0.5, 0.5);

    if (bEdge) glEdgeFlag(FALSE);
    glVertex2f(0.0, 0.0);
    if (bEdge) glEdgeFlag(TRUE);
    glVertex2f(-0.5, -0.5);
    if (bEdge) glEdgeFlag(FALSE);
    glVertex2f(0.5, -0.5);

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

그리기 코드 중간 중간에 조건문이 삽입되어 다소 복잡해 보이는데 이 조건문들을 다 빼면 단순히 삼각형 세 개를 그리는 코드이다. 중간 중간에 각 정점에 대해 에지 플래그를 지정하는 문장들이 배치되어 있되 실행 중에 토글하기 위해 bEdge 변수로 조건 처리했다. bEdge 변수는 팝업 메뉴에서 토글한다.



bEdge 기능을 사용하지 않으면 디폴트로 모든 선분이 외곽선으로 인식되므로 안쪽 삼각형도 경계선이 그려진다. bEdge 변수를 토글하여 내부의 선임을 알려 주면 내부의 선이 사라져 오각형만 남는다.

이 기능은 사실 별 실용성이 없다. 큰 다각형을 삼각형으로 억지로 쪼개다 보니 외곽선이 아닌 부분이 생기는데 애초부터 아예 볼록한 삼각형이나 사각형으로 구성하면 이럴 필요가 없다. 설사 그럴 필요가 있더라도 와이어 프레임 랜더링을 할 때만 문제가 될 뿐이고 솔리드 랜더링을 할 때는 경계선이 보이지도 않는다.

6-6.3차원 물체

이번 절에서는 처음으로 입체 도형을 3차원 공간에 그려 보았다. 그런데 기껏 그린 것이 정점 5개로 구성된 피라미드여서 별로 3차원 같아 보이지도 않고 시시껄렁해 본다. 피라미드 말고 이왕이면 로보트 태권 V나 뽀

로로 같은 멋진 캐릭터를 그려 보면 진짜 3D 그래픽을 하는 것 같은 느낌도 들고 실습하는 재미도 있고 있을 것이다.

그러나 태권 V를 그리려면 도대체 정점을 몇개나 찍어야 할까? 아무리 대충 그려도 수천개는 필요하고 섬세하게 그리려면 수십만개가 필요하니 대략 아찔하다. 사실 복잡한 입체 도형을 일일이 정점 좌표를 지정하여 그리는 것은 결코 쉬운 일이 아니다. IT 업계에서는 이런 일을 전문용어로 노가다라고 하며 정도가 좀 심하면 개노가다라고 한다.

그래도 최소한 피라미드보다는 좀 그럴 듯한 물체를 그려 봐야 실습하는 재미가 있을 것 같은데 다행히 손쉽게 3차원 물체를 그릴 수 있는 방법이 있다. GLUT운 실습용 물체를 그리는 기능을 제공하므로 테스트용으로 사용할 물체 정도는 함수 호출 하나로 그릴 수 있다. 다음 함수는 큐브, 즉 입체적인 육면체를 그린다.

```
void glutWireCube(GLdouble size);
void glutSolidCube(GLdouble size);
```

육면체를 그리려면 정점 8개를 배치하고 사각형 6개를 그려야 하지만 이 함수를 호출하면 한면의 길이만 전달함으로써 간단하게 육면체를 그릴 수 있다. 다음 함수는 구를 그린다.

```
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
```

반지름과 가로, 세로 분할 수를 지정하면 구가 그려진다. 분할 수는 구를 몇등분하여 그릴 것인가를 지정하는데 지구본의 위도, 경도선과 비슷하다. 분할수가 높을수록 다각형 수가 많아지므로 구에 더 가까워진다. 물론 시간은 더 오래 걸린다. 다음 함수는 원뿔을 그린다.

```
void glutWireCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
void glutSolidCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
```

아랫변 원의 반지름과 높이, 분할수를 전달한다. 다음 함수는 토러스를 그린다. 토러스는 환원체라고도 하는데 쉽게 말해서 던킨 도너츠를 생각하면 된다.

```
void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);
void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);
```

안쪽 반지름, 바깥쪽 반지름, 분할수를 지정한다. 다음 함수들은 정다각형을 그린다. 4면체, 8면체, 12면체, 20면체이 등이다. 앞에서 먼저 소개한 6면체도 물론 정다각형이다. 수학에 조금이라도 관심있는 사람들은 상식적으로 알고 있겠지만 정다면체는 딱 이 다섯 가지밖에 없다.

```
glutWireTetrahedron
glutWireOctahedron
glutWireDodecahedron
glutWireIcosahedron
```

다음 함수는 다소 복잡한 물체인 주전자를 그린다. 이 주전자는 유타 대학에서 처음 디자인했다고 해서 유타 주전자로도 불리며 3D 그래픽 프로그램의 예제로 종종 사용된다. 인수로 크기만 전달하면 주전자가 그려진다.

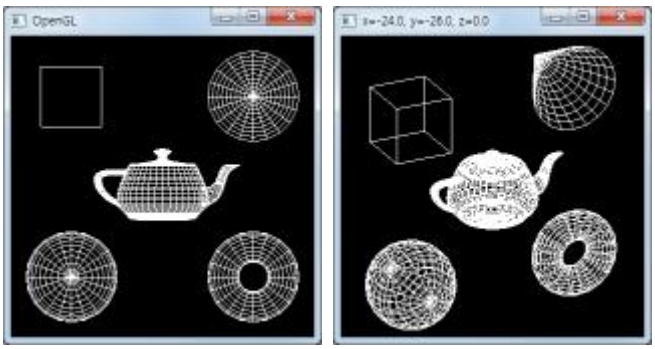
void glutWireTeapot(GLdouble size);
void glutSolidTeapot(GLdouble size);

이 함수들을 사용하면 테스트용 도형을 아주 쉽게 생성할 수 있다. 각 도형이 어떻게 생겼는지 구경해 보기 위해 다섯 개의 입체 도형을 한 화면에 그려 보았다. GLUT은 모든 도형을 원점에 그리므로 적당히 평행시켜야 한 화면에 여러 개의 물체를 그릴 수 있다. 이동 변환에 대해서는 잠시 후 연구해 볼 것이다.

glutObject

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

	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);

	glutWireTeapot(0.3);

	glPushMatrix();
	glTranslatef(-0.6, 0.6, 0.0);
	glutWireCube(0.4);
	glPopMatrix();

	glPushMatrix();
	glTranslatef(-0.6, -0.6, 0.0);
	glutWireSphere(0.3, 20, 20);
	glPopMatrix();

	glPushMatrix();
	glTranslatef(0.6, 0.6, 0.0);
	glutWireCone(0.3, 0.6, 20, 10);
	glPopMatrix();

	glPushMatrix();
	glTranslatef(0.6, -0.6, 0.0);
	glutWireTorus(0.1,0.2,20, 20);
	glPopMatrix();

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

왼쪽 위에는 한 면의 길이가 0.4인 육면체의 와이어 프레임을 그렸다. 좌표 공간의 딱 중앙에 놓이므로 최초 실행시는 그냥 사각형으로 보이며 약간 회전시켜 봐야 육면체로 보인다. 입체 모양 확인을 위해 회전 기능이 필요하며 다른 도형들도 마찬가지이다.



창을 충분히 크게 해 놓고 각 방향으로 회전시켜 보아라. glutSolidCube 함수는 면이 채워진 모양을 그리되 아직 조명이나 채색을 배우지 않았으므로 지금은 그려 봐야 그냥 흰색으로만 보인다.

glu 라이브러리도 테스트를 위한 입체 도형을 제공한다. glu는 별도의 설정 객체로 그리는 방법을 다양하게 지정할 수 있다. 설정 객체는 다음 함수로 생성 및 파괴한다.

```
GLUquadric* gluNewQuadric();
void gluDeleteQuadric(GLUquadric*  quad);
```

객체를 생성한 후 다음 함수들로 그리기 속성을 설정한다.

```
void gluQuadricDrawStyle(GLUquadric* quad, GLenum draw);
void gluQuadricNormals(GLUquadric* quad, GLenum normal);
void gluQuadricOrientation(GLUquadric* quad, GLenum orientation);
void gluQuadricTexture(GLUquadric* quad, GLboolean texture);
```

법선이나 텍스처 등에 대한 복잡한 설정은 다음에 연구해 보기로 하고 그리기 스타일에 대해서만 알아 보자. 다음 인수로 도형을 어떻게 그릴지를 지정한다.

인수	설명
GLU_FILL	가득 채운다.
GLU_LINE	와이어 프레임으로 그린다.
GLU_SILHOUETTE	폴리곤의 인접면을 제외하고 와이어 프레임으로 그린다.
GLU_POINT	점의 집합으로 그린다.

도형을 그리는 모든 함수들은 설정 객체를 첫번째 인수로 받는다. 나머지 인수는 그리는 도형에 대한 인수들이되 GLUT의 그리기 함수들과 거의 비슷하다.

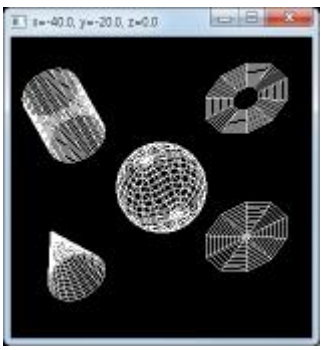
```
void gluSphere(GLUquadric* quad, GLdouble radius, GLint slices, GLint stacks);
void gluCylinder(GLUquadric* quad, GLdouble base, GLdouble top, GLdouble height, GLint
   slices, GLint stacks);
void gluDisk(GLUquadric* quad, GLdouble inner, GLdouble outer, GLint slices, GLint  loops);
```

구는 반지름과 분할 수를 지정한다. 실린더는 아랫면과 윗면의 반지를 지정하되 이 둘이 같으면 원기둥이 되고 다르면 원뿔이 된다. 디스크는 안쪽 반지름과 바깥쪽 반지름을 지정하되 안쪽이 0이 아니면 구멍이 뚫인 엽전 모양이 된다. 간단하므로 예제 만들어서 확인만 해 보자.

gluObject

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

	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);

	GLUquadricObj *pQuad;
	pQuad = gluNewQuadric();
	gluQuadricDrawStyle(pQuad, GLU_LINE);

	gluSphere(pQuad, 0.3, 20, 20);

	glPushMatrix();
	glTranslatef(-0.6, 0.6, 0.0);
	gluCylinder(pQuad, 0.2, 0.2, 0.5, 20, 20);
	glPopMatrix();

	glPushMatrix();
	glTranslatef(-0.6, -0.6, 0.0);
	gluCylinder(pQuad, 0.2, 0.0, 0.5, 20, 20);
	glPopMatrix();

	glPushMatrix();
	glTranslatef(0.6, 0.6, 0.0);
	gluDisk(pQuad, 0.1, 0.3, 10, 10);
	glPopMatrix();

	glPushMatrix();
	glTranslatef(0.6, -0.6, 0.0);
	gluDisk(pQuad, 0.0, 0.3, 10, 10);
	glPopMatrix();

	gluDeleteQuadric(pQuad);
	glPopMatrix();
	glFlush();
}
```

다섯 개의 물체를 그리되 와이어 프레임으로 그려 모양만 확인해 보았다. 회전해 보면 과연 입체임을 확인할 수 있다.



두 라이브러리가 제공하는 이 물체들은 어디까지나 테스트용이며 OpenGL을 처음 배우는 사람이 입체 물체를 쉽게 만들 수 있도록 도와준다. 앞으로의 실습에서 이 물체들을 종종 사용할 것인다.