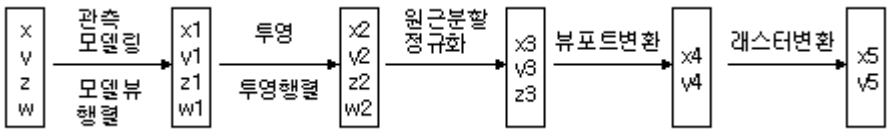
8. 변환

8-1. 변환 과정

지금까지의 실습에서는 디폴트로 주어진 좌표 공간을 그냥 사용했다. -1 ~ 1 사이의 좁은 공간에 물체를 배치했으면 위치도 항상 중앙의 원점이었다. 이제 좌표 공간과 물체에 변화를 주어 보자. 지금까지는 입체 확인을 위해 회전 변환만 사용해 보았는데 이 절에서 여러 가지 변환을 실습해 볼 것이다.

3차원 공간에 배치된 물체는 이동, 확대, 회전 등 여러 가지로 변화를 줄 수 있으며 보는 각도에 따라 모양이 달라지기도 한다. 좌표 공간 자체도 범위와 증감 방향에 대해 원하는대로 지정할 수 있다. 또 3차원 공간의 장면도 결국은 모니터나 지면에 출력해야 하므로 2차원으로 바꿔야 한다. 이 각각의 과정에서 복잡한 변환이 발생한다. 전체적인 변환 과정은 다음과 같다.



각 변환 단계에서 정점들의 실제 위치가 수시로 바뀐다. OpenGL은 변환 단계에서 수행되는 복잡한 연산들을 행렬로 처리한다. 모든 변환 함수의 연산 결과는 현재 행렬에 반영된다. 좀 더 정확하게는 현재 선택된 행렬 스택의 최상단 행렬인데 일단은 현재 행렬이라고 생각하면 된다.

행렬을 조작하는 함수를 호출할 때마다 대상 행렬을 일일이 밝히는 것은 너무 번거롭기 때문에 항상 현재 행렬에 연산을 하되 현재 행렬을 바꿀 수 있도록 되어 있다. 그래서 변환을 수행하기 전에 어떤 행렬에 대해 연산을 할 것인지를 지정해야 한다. 다음 함수는 행렬 연산의 목적지를 지정한다.

void glMatrixMode(GLenum mode);

인수로 대상 행렬을 지정하며 이를 행렬 모드라고 한다. 이 함수로 지정한 행렬은 이후의 모든 행렬 연산의 대상이 된다. 물론 다른 행렬을 조작하고 싶을 때는 언제든지 행렬 모드를 바꿀 수 있다. 다음과 같은 행렬 모드가 있는데 주로 모델뷰 행렬이나 투영 행렬이 변환 대상이다. 디폴트는 모델뷰 행렬이다.

열거형	설명
GL_MODELVIEW	모델 뷰 변환 행렬. 이 값이 디폴트이다.
GL_PROJECTION	투영 행렬
GL_TEXTURE	텍스처 행렬
GL_COLOR	색상 행렬. 단 이 기능은 ARB_imaging 확장 기능이 지원되어야 한다.

현재 행렬이 무엇인지를 알아내려면 glGet 함수로 GL_MATRIX_MODE를 전달한다. 현재 행렬을 지정한 후 다양한 행렬 조작을 할 수 있다. 앞으로 여러 가지 변환 함수들을 배우겠지만 일단 대표적으로 다음 함수 하나에 대해 알아 보자.

void glLoadIdentity(void);

이 함수는 현재 행렬을 단위 행렬로 만든다. 단위 행렬은 우하향 대각선 방향만 1이고 나머지 요소는 모두 0인 행렬로서 임의의 행렬을 곱해도 원래 행렬이 계산되는 특수한 행렬이다. 곰셉의 1, 덧셈의 0과 같은 항등원으로서 연산을 해도 처음값이 유지된다. 현재 행렬을 단위 행렬로 만든다는 것은 행렬을 리셋한다는 뜻이며 이는 곧 어떠한 변환도 하지 않는다는 뜻이다. 지금까지 입체 확인을 위해 회전 기능을 사용했던 모든 예제를 보면 다음 두 행의 코드가 있다.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

이 코드는 모델뷰 행렬을 리셋한다. DoDisplay를 이전에 실행했을 때 적용했던 회전값을 무시하고 다시 설정하기 위해 리셋을 해야 한다. 이 리셋 코드가 없으면 회전이 계속 누적 적용되어 원하는 대로 회전되지 않는다. 그림을 그리기 전에 화면을 지우는 것과 마찬가지로 행렬을 사용하기 전에 리셋을 먼저 해야 한다.

8-2.관측 변환

관측(Viewing)이란 3차원 공간의 장면을 바라본다는 뜻이다. 관측 변환은 장면을 바라보는 사용자의 위치나 시점, 각도를 변경하는 변환이다. 같은 장면이라도 어디서 어떤 각도로 바라 보는가에 따라 모양이 완전히 달라진다. 마치 장면을 촬영하는 카메라를 이리 저리 옮기는 기법과 유사해서 카메라 변환이라고도 한다. 관측 지점은 다음 함수로 지정한다.

```
void gluLookAt(
   GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,
   GLdouble centerX, GLdouble centerY, GLdouble centerZ,
   GLdouble upX, GLdouble upY, GLdouble upZ
);
```

보다시피 이 함수는 gl의 함수가 아니라 glu의 유틸리티 함수이다. 왜냐하면 이 함수가 아니어도 동일한 효과를 낼 수 있는 방법이 있기 때문이다. 인수가 굉장히 많은 것처럼 보이는데 세 가지 관점에 대해 x, y, z 각 세쌍씩이므로 총 9개의 인수가 필요하다.

eye 좌표는 시선의 좌표 즉, 관찰자의 위치를 나타내는 좌표이다. center는 관찰자가 바라보고 있는 좌표이다. up은 위쪽을 가리키는 업 벡터를 나타낸다. 카메라로 장면을 촬영하고 있다면 카메라가 있는 곳이 eye 좌표이고 카메라가 초점으로 정한 부분이 center 좌표이며 카메라의 각도가 up 벡터이다. 규칙상 up 벡터는 시선과 평행해서는 안된다.

Viewing

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

void DoDisplay();
void DoKeyboard(unsigned char key, int x, int y);
GLfloat xAngle, yAngle, zAngle;
GLfloat ex, ey, ez;

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

```
	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;

	case 'j':ex += 0.1;break;
	case 'l':ex -= 0.1;break;
	case 'i':ey -= 0.1;break;
	case 'k':ey += 0.1;break;
	case 'u':ez += 0.1;break;
	case 'o':ez -= 0.1;break;
	case 'm':ex = ey = ez = 0.0;break;
	}
	char info[128];
	sprintf(info, "ex=%.1f, ey=%.1f, ez=%.1f", ex, ey, ez);
	glutSetWindowTitle(info);
	glutPostRedisplay();
}

void DrawPyramid()
{
	=== 소스 생략 ===
}

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

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
	glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
	glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

	gluLookAt(
		ex, ey, ez,
		0.0, 0.0, -1.0,
		0.0, 1.0, 0.0
		);

	DrawPyramid();

	glFlush();
}
```

화면 정 중앙에 피라미드를 놓았다. 관측 지점은 ex, ey, ez로 하되 이 값은 키보드의 ujikol 키로 조정할 수 있도록 해 두었다. m키는 관측 지점을 (0,0,0)으로 리셋한다. 바라보는 곳은 (0,0,-1)로 하여 피라미드의 꼭대

기보다 약간 더 위쪽으로 설정했고 업 벡터는 y 방향을 위쪽으로 하였다.

j키를 눌러 x축을 따라 오른쪽으로 관측 지점을 옮기면 물체는 왼쪽으로 이동한다. 시점이 고정된 상태에서 카메라를 이동시키면 각도가 틀어지므로 평행하게 이동하지 않고 약간 비스듬하게 이동할 것이다. 마찬가지로 k키를 눌러 y 축을 따라 위쪽으로 관측 지점을 옮기면 물체는 아래쪽으로 이동한다.

u, o 키로 z 좌표를 앞뒤로 이동해 보면 피라미드 꼭대기가 사라진다. 피라미드 안쪽으로 시점이 이동해 버리기 때문이다. 뒤쪽으로 너무 멀리 가도 피라미드가 사라지는데 이는 직교 투영의 가시 영역이 1 ~ -1사이로 지정되어 있기 때문이다. 이 범위를 넘어서면 가시 역역에서 사라져 버린다.

8-3.모델링 변환

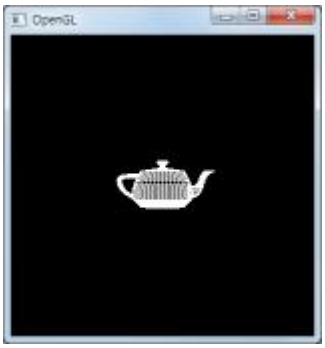
모델링(Modeling) 변환은 3차원 공간에 배치된 물체를 변형한다. 이동, 확대/축소, 회전 등 여러 가지 변환이 있으며 두 가지 이상의 변환을 동시에 적용하기도 한다. 양이 좀 많지만 특별히 어렵지는 않으므로 순서대로 구경만 해 보면 된다.

Modeling

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

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

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutCreateMenu(DoMenu);
      glutAddMenuEntry("변환 없음",0);
      glutAddMenuEntry("이동",1);
      glutAddMenuEntry("엉뚱한 위치에 나타나는 이동",2);
      glutAddMenuEntry("단위 행렬로 리셋",3);
      glutAddMenuEntry("스택에 저장 및 복구",4);
      glutAddMenuEntry("확대",5);
      glutAddMenuEntry("회전",6);
      glutAddMenuEntry("확대 후 이동",7);
      glutAddMenuEntry("이동 후 확대",8);
      glutAddMenuEntry("원점 기준 회전",9);
      glutAddMenuEntry("제자리 회전",10);
      glutAttachMenu(GLUT_RIGHT_BUTTON);
      glutMainLoop();
      return 0;
}
```

```
void DoMenu(int value)
{
	if (value < 100) {
		Action = value;
		glMatrixMode(GL_MODELVIEW);
		glLoadIdentity();
		glColor3f(1,1,1);
		glutPostRedisplay();
		return;
	}
}

void DoDisplay()
{
	// 변환 없음
	glClear(GL_COLOR_BUFFER_BIT);

	glutWireTeapot(0.2);

	glFlush();
	break;
}
```

중앙에 주전자를 그려 놓고 정면에서 바라보았다. 회전 변환을 쉽게 확인하기 위해 모양이 특이한 주전자를 사용했다. 중앙에 주전자가 나타날 것이다.



이 주전자를 여러 가지 형태로 변환해 보자. 팝업 메뉴에 각 변환 동작이 정의되어 있으므로 메뉴만 선택하면 효과를 즉시 확인할 수 있다. 위치는 다음 함수로 이동시킨다.

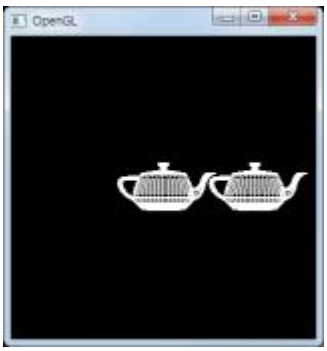
void glTranslate[f,d](GLfloat x, GLfloat y, GLfloat z);

각 축으로 이동할 거리를 지정하며 이 거리가 물체를 구성하는 모든 정점에 더해진다. 결국 물체가 이 거리만큼 이동하는 것이다. 다음과 같이 수정해 보자.

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

	glutWireTeapot(0.2);

	glTranslatef(0.6, 0.0, 0.0);
	glutWireTeapot(0.2);
```

```
	glFlush();
}
```

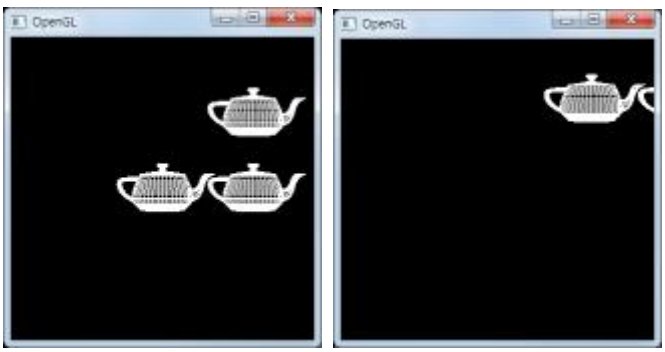
오른쪽으로 0.6만큼 이동시킨 후 똑같은 주전자를 하나 더 그렸다. 조금 이동한 위치에 그려질 것이다. 필요한만큼 이동 거리만 밝히면 된다.



이번에는 오른쪽뿐만 아니라 가운데 주전자의 위쪽에도 하나를 더 그려 보자. y 축으로 0.5만큼 이동한 곳에 하나 더 그리면 될 것 같다.

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

	glutWireTeapot(0.2);

	glTranslatef(0.6, 0.0, 0.0);
	glutWireTeapot(0.2);

	glTranslatef(0.0, 0.6, 0.0);
	glutWireTeapot(0.2);

	glFlush();
}
```

이렇게 하면 될 것 같지만 엉뚱한 결과가 나타난다. y축으로만 0.5 이동시켰는데 x축으로도 0.5만큼 이동하여 오른쪽 위에 그려졌다.



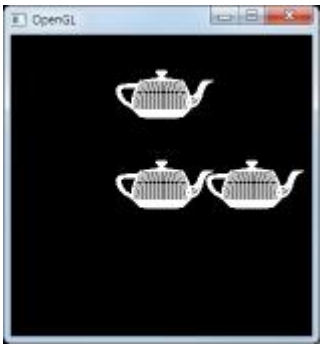
이렇게 되는 이유는 변환이 누적적으로 적용되기 때문이다. 변환 함수는 현재 행렬에 변환 공식을 써 넣으며 현재 행렬은 이후의 모든 출력에 영향을 미친다. 행렬을 리셋하거나 특별히 다른 값으로 바꾸지 않는 한 현재 값이 계속 유지되고 매 호출시마다 행렬에 변환이 가해진다. 먼저 오른쪽으로 0.5 이동했고 다시 위로 0.5 이동했으므로 세번째 주전자는 오른쪽 위로 이동하는 것이다.

문제는 그 뿐만이 아니다. 행렬에 한번 누적된 것은 계속 유효하므로 다음번 그릴 때는 추가로 이동하게 된다. 위 예제를 실행해 놓은 상태에서 창의 폭을 조금씩 키워 보면 그때마다 그리기 함수가 다시 호출되고 매번 0.5만큼 누적적으로 이동한다. 그래서 주전자가 점점 오른쪽으로 이동하다가 결국은 뷰포트 밖으로 사라져 버린다. 이 예제뿐만 아니라 앞 두 예제도 동일한 부작용이 있다.

그리기를 할때마다 앞서 수행한 변환을 취소해야 한다. 예를 들어 앞에서 오른쪽으로 이동했으면 왼쪽으로 이동하여 원점으로 다시 옮기면 된다. 그러나 앞서 어떤 변환을 했는데 일일이 알아 내기 어렵다. 그래서 아예 그리기를 할 때마다 현재 행렬 자체를 초기화하는 것이 편리하다. 단위행렬로 만들면 이전의 변환은 모두 리셋된다. 다음과 같이 수정하면 문제가 해결된다.

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

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	glutWireTeapot(0.2);

	glTranslatef(0.6, 0.0, 0.0);
	glutWireTeapot(0.2);

	glLoadIdentity();
	glTranslatef(0.0, 0.6, 0.0);
	glutWireTeapot(0.2);

	glFlush();
}
```

이전의 변환을 취소하기 위해 그리기를 처음 시작할 때 모델뷰 행렬을 단위 행렬로 리셋한다. 이 상태에서 중앙에 하나 그리고 오른쪽으로 이동한 후 하나 더 그린다. 다시 단위 행렬로 리셋한 후 위로 이동하여 하나 더 그렸다. 원하는대로 출력되며 매 그리기를 수행할 때마다 행렬을 리셋한 후 다시 이동하므로 창의 크기가 변해도 문제가 없다.



단위 행렬로 리셋하는 대신에 이전의 행렬을 스택에 저장하는 방법도 사용할 수 있다. 행렬을 스택에 저장하거나 복구할 때는 다음 함수를 호출한다.

void glPushMatrix(void);
void glPopMatrix(void);

행렬은 단순한 하나의 값이 아니라 스택에 여러 개가 저장되며 그 중 스택의 제일 위에 있는 행렬이 현재 행렬이다. 스택의 깊이는 행렬의 종류에 따라 다른데 모델뷰 행렬은 32개까지 저장할 수 있으며 투영 행렬은 2개만 저장할 수 있다. 다음과 같이 해도 결과는 같다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	glutWireTeapot(0.2);

	glPushMatrix();
	glTranslatef(0.6, 0.0, 0.0);
	glutWireTeapot(0.2);
	glPopMatrix();

	glTranslatef(0.0, 0.6, 0.0);
	glutWireTeapot(0.2);

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

그리기 전의 행렬을 일단 저장한다. 원점에 주전자를 그린 후 오른쪽으로 이동하기 전에 한번 더 저장한다. 이 상태에서 오른쪽으로 이동한 곳에 주전자를 그리고 다시 복구한다. 원점으로 제깍 돌아올 것이다. 이 상태에서 위로 이동한 후 그리면 원점 바로 위가 된다. 모든 그리기가 끝난 후 다시 복구하면 DoDisplay를 호출하기 전의 상태로 돌아간다.

행렬을 리셋하는 방법은 무조건 단위 행렬로 돌아가는데 비해 스택을 사용하는 방법은 이전 상태를 그대로 저장했다가 복구한다는 면에서 더 범용적이다. DoDisplay 안에서 행렬을 어떻게 조작하든간에 리턴할 때 원래대로 복구하므로 외부에서 가한 변환이 유지되며 그리기를 여러 번 하더라도 변환이 누적되지 않는다.

여러 곳에서 공유되는 전역 변수나 상태는 원칙적으로 바꾼 놈이 원래대로 돌려 놓는 것이 옳다. 행렬도 상태 머신에 저장되는 전역 값이므로 행렬을 바꾸는 측에서 원래값을 복원하는 것이 좋다. 이후 변환을 수행하는 코드에서는 원칙대로 행렬을 저장한 후 리턴하기 전에 복구할 것이다. 다음 함수는 물체를 확대한다.

void glScale[f, d](GLfloat x, GLfloat y, GLfloat z);

스케일값이 1보다 더 크면 확대되고 더 작으면 축소된다. 일정한 크기로 확대, 축소하고 싶다면 세 방향 모두 같은 비율을 주어야 한다. 각 방향별로 다른 배율을 주면 찌그러진 모양을 만들 수도 있다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	glScalef(2.0, 3.0, 1.0);

	glutWireTeapot(0.2);

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

수평으로 2배, 수직으로 3배 확대하였다. 원래 모양보다 아래위로 훨씬 더 길쭉한 모양의 주전자가 된다.

1보다 더 작은 값을 주면 축소되며 배율이 0이면 화면에서 사라진다. 음수를 주면 반대로 뒤집힌다. 다음은 glScalef(-2.0, 3.0, 1.0);로 확대하여 수평 방향으로 주전자를 뒤집은 것이다.

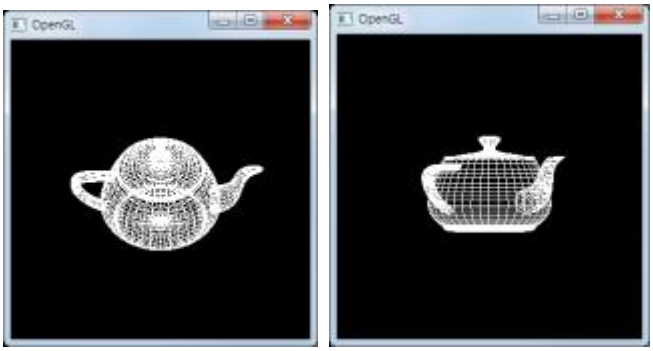
주전자의 손잡이가 원래 오른쪽에 있었는데 왼쪽으로 이동했다. y 축 배율을 음수로 주면 거꾸로 뒤집힐 것이다. 다음 함수는 회전시킨다.

void glRotate[f, d](GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

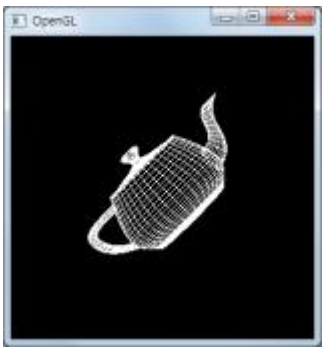
angle은 회전시킬 각도이며 반시계 방향의 360분법 각도이다. x, y, z는 회전의 기준이 되는 벡터이다. 다음은 X 축을 기준으로 45도 회전시킨 것이다. 회전 효과를 좀 더 분명히 관찰하기 위해 주전자 크기를 0.4로 확대했다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	glRotatef(45.0, 1.0, 0.0, 0.0);

	glutWireTeapot(0.4);

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

x축을 기준으로 한다는 것은 주전자의 수평 방향으로 막대기를 꽂아 놓고 이 막대기를 회전시키는 것과 같다. x축 기준이라고 해서 좌우로 회전하는 것이 아님을 주의하자. 오른쪽은 y축을 기준으로 회전한 것이다.



다음은 Z축을 중심으로 45도 회전시켜 본 것이다. 반시계 방향으로 회전되므로 주전자의 코가 위쪽으로 올라간다. 물론 여러 축에 대해 동시에 회전을 적용하는 것도 가능하며 요리 조리 돌려 보면 물체의 모든 면을 골고루 관찰할 수 있다.

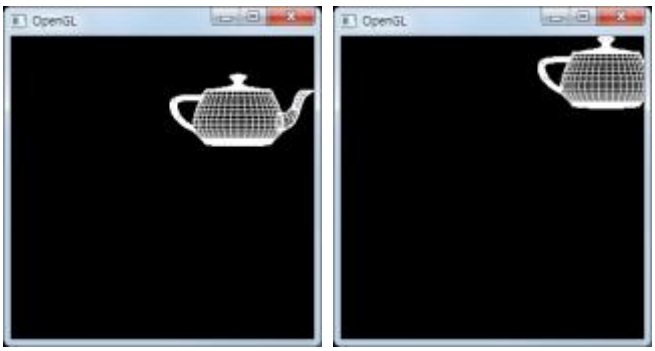


회전은 이미 앞에서 입체적인 모양을 확인하기 위해 여러번 사용했었다. 이제 회전 코드를 보면 완벽하게 이해할 수 있을 것이다. 수평으로 회전하는 ad키가 y축 회전값을 증감시키고 수직으로 회전하는 ws키가 x축 회전값을 증감시키는 이유를 알 수 있을 것이다. 이런 것을 잘 이해하려면 공간적 상상력이 풍부해야 한다.

한번에 두가지 이상의 변환을 동시에 적용할 수도 있는데 이런 변환을 복합 변환이라고 한다. 이때 변환 순서에 따라 결과가 달라진다. 다음은 이동 함수와 확대 함수를 연거푸 호출한다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	glTranslatef(0.5, 0.5, 0.0);
	glScalef(1.5, 1.5, 1.0);

	glutWireTeapot(0.2);

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

여러 개의 변환을 연이어 가하면 나중에 호출한 변환이 먼저 적용된다. 이 경우는 이동, 확대 함수를 이어서 호출했으므로 확대 후 이동된다. 순서를 바꾸어 이동 후 확대하면 이동 거리가 더 멀어진다.

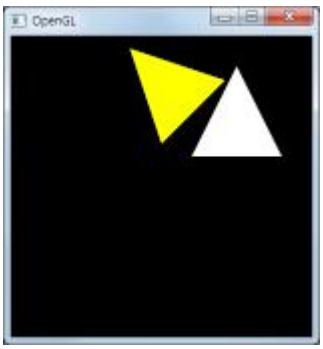


이 현상은 연산의 우선 순위로 간단하게 설명된다. 이동은 덧셈이고 확대는 곱셈인데 곱셈의 우선 순위가 더 높기 때문에 순서에 따라 최종 결과가 달라지는 것이다. 더한 후에 곱할 것인가 곱한 후에 더할 것인가의 차이이다.

복합 변환은 어쩌다 한번씩 사용되는 것이 아니라 일반적으로 자주 사용되며 그것도 두 번 정도가 아니라 여러 개의 변환이 한꺼번에 적용되는 경우가 많다. 대표적인 예가 물체를 제자리에서 회전시키는 것이다. 다음 코드는 오른쪽 위에 삼각형을 하나 그려 놓고 이 삼각형을 45도 회전시킨다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	glBegin(GL_TRIANGLES);
	glVertex2f(0.5, 0.8);
	glVertex2f(0.2, 0.2);
	glVertex2f(0.8, 0.2);
	glEnd();

	glRotatef(45.0, 0.0, 0.0, 1.0);

	glColor3f(1,1,0);
	glBegin(GL_TRIANGLES);
	glVertex2f(0.5, 0.8);
	glVertex2f(0.2, 0.2);
	glVertex2f(0.8, 0.2);
	glEnd();

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

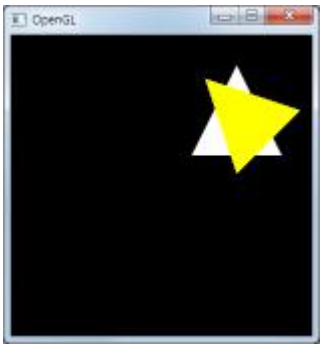
코드는 아주 쉽다. 흰색 삼각형 하나 그리고 z축을 기준으로 45도 회전시킨 후 노란색으로 다시 그렸다. 원본과 회전 후의 물체를 구분하기 위해 색상을 일부러 다르게 주었다. 그러나 결과는 기대한 것과는 다르게 나타난다.



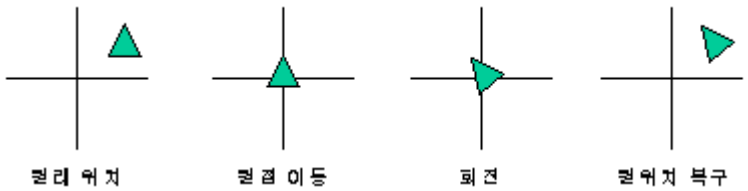
회전 함수들은 항상 원점을 기준으로 물체를 회전시킨다. 물체의 중심이 어디인지는 알지도 못하며 각 물체마다 중심이 다르므로 그렇게 할 수도 없다. 삼각형의 중심을 기준으로 회전시키려면 다음 세 단계를 거쳐야 한다.

```
glTranslatef(0.5, 0.5, 0.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
glTranslatef(-0.5, -0.5, 0.0);
```

삼각형의 중심이 (0.5, 0.5)이므로 이 중심점을 평행이동시켜 원점으로 가져온다. 이 상태에서 축을 기준으로 회전시키고 다시 원래 위치로 이동시켜야 한다. 변환은 반대 순서로 적용되므로 음수 방향으로의 이동이 제일 나중이고 회전 후 반대 방향으로 재이동한다. 이 코드를 실행해 보면 노란색 삼각형이 흰 삼각형 위에서 45도 회전되어 있을 것이다.



각 물체의 회전 중심이 달라도 이 방법대로 변환하면 원하는 중심점을 기준으로 회전할 수 있다. 회전 뿐만 확대도 확대의 중심점을 지정할 수 있는데 이때도 동일한 절차대로 중심점을 지정한다. 이 경우 변환 과정은 다음과 같이 수행된다.



뭔가 복잡해 보이고 여러 단계를 거쳐 느릴 것 같지만 이 단계들이 모두 행렬식 하나로 합쳐서 실행되므로 실행 속도상의 불이익은 없다. 이 예제에서 지금까지 사용하던 주전자를 사용하지 않고 삼각형을 사용한 이유는 GLUT의 주전자는 무조건 원점에만 출력되어 복합 변환을 설명하기에는 부적합하기 때문이다.

이상으로 모델링 변환에 대해 순서대로 실습해 보았다. 함수의 인수가 직관적이서 변환의 정도를 쉽게 결정할 수 있고 결과를 눈으로 바로 바로 확인할 수 있다. 그래서 모델링 변환은 다른 변환에 비해 쉽고 재미있는 부분이다.

관측 변환과 모델링 변환은 효과가 비슷해서 서로 대체 가능하다. 물체를 약간 왼쪽으로 옮기고 싶다면 물체 자체를 왼쪽으로 옮겨도 되지만 관찰자가 오른쪽으로 이동해도 마찬가지다. 확대하고 싶다면 관찰자가 물체에 더 가까이 다가서면 되고 회전시키고 싶다면 고개를 빼딱하게 옆으로 돌리면 된다. 그래서 이 둘을 합쳐 모델뷰(ModelView) 변환이라고 하며 변환 결과를 기록하는 행렬도 동일하다.

8-4.투영

모델뷰 변환은 3차원 공간상에서 물체의 위치나 크기를 조정한다. 이 물체들을 화면에 출력하려면 결국은 2차원의 평면으로 옮겨야 한다. 아직까지 3차원 모니터라는 것은 없으며 인쇄를 해도 평평한 종이에 출력해야 하기 때문이다. 3차원 좌표를 2차원으로 바꾸는 것을 투영(Projection)이라고 한다. 투영은 3차원 좌표를 대응되는 2차원 좌표로 전환한다.

또한 투영은 보이는 범위를 제한하기도 한다. 3차원 공간에 그려진 물체라고 해서 모두 다 보여야 하는 것은 아니며 그 중 필요한 범위를 설정하여 일부만 표시한다. 보이는 영역을 잘라내는 것을 클리핑이라고 하며 클리핑 영역 안쪽의 보이는 범위를 가시 영역(View volume)이라고 한다. 상하좌우의 물체중 시야에 들어오는 것만 포함하며 나머지는 잘라낸다. 또한 관찰자의 뒤쪽에 있는 것도 잘라내며 앞쪽에 있더라도 너무 멀리 있는 것은 제외해야 한다.

개념적으로 투영은 유리창에 비친 모습을 그대로 그려내는 것이다. 어떤 3차원 장면 앞에 유리창을 대고 그 유리창에 보이는 모습을 그대로 그리는 것이 바로 투영이다. 유리창은 평면이고 이 평면의 면적은 제한적이므로 장면의 일부만 담을 수 있다. 카메라가 장면을 촬영하는 것도 일종의 투영인데 3차원 공간을 찍어도 필름은 2차원의 평면이기 때문이다. 사람의 감각기관도 망막은 2차원이므로 3차원을 투영하여 인식하는 것이다. 다만 두 눈의 시야가 겹치는 것을 이용하여 거리를 판별해내는 것이다.

투영과 클리핑은 화면의 크기에 영향을 받는다. 그래서 투영은 보통 화면의 크기가 바뀔 때 현재 화면 크기에 맞게 지정한다. 화면 크기가 바뀌는 시점에 특정한 처리를 하려면 다음 함수로 콜백을 등록한다.

void glutReshapeFunc(void (*func)(int width, int height));

콜백 함수의 인수로 작업영역의 폭과 높이가 전달된다. 윈도우의 크기가 아니라 작업영역 즉, 그림이 그려지는 영역의 크기임을 유의하자. 이 영역의 비율로 종횡비를 계산하고 화면에 들어올만큼만 클리핑해야 한다.

투영은 굉장히 변수가 많은 기법이라 이것 저것 바꿔 가며 결과를 비교해 봐야 이해하기 쉽다. 다음 예제는 투영 기능을 테스트한다. 여러 가지 투영 방식을 동일하게 비교하기 위해 직교 투영의 Near, Far를 디폴트와 달리 -1, 1로 지정했다. 직교 투영의 Near, Far 디폴트는 원래 1, -1이며 음수로 적용하므로 사용자측의 z좌표는 음수이다. 디폴트를 -1, 1로 조정하면 z축은 사용자쪽이 양수가 되므로 피라미드의 꼭대기를 0.8로 부호를 바꾸었다.

Projection

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

void DoDisplay();
void DoReshape(GLsizei width, GLsizei height);
void DoKeyboard(unsigned char key, int x, int y);
void DoMenu(int value);
GLfloat xAngle, yAngle, zAngle;
GLfloat left=-1, right=1, bottom=-1, top=1, Near=-1, Far=1;
GLfloat fov = 45;
int Projection;
int Object;
```

```
GLsizei lastWidth, lastHeight;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutReshapeFunc(DoReshape);
      glutKeyboardFunc(DoKeyboard);
      glutCreateMenu(DoMenu);
      glutAddMenuEntry("Orthographic",1);
      glutAddMenuEntry("Frustrum",2);
      glutAddMenuEntry("Perspective",3);
      glutAddMenuEntry("Pyramid",4);
      glutAddMenuEntry("Cylinder",5);
      glutAttachMenu(GLUT_RIGHT_BUTTON);
      glEnable(GL_DEPTH_TEST);
      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;

      case 'r':left += 0.1;break;
      case 'f':left -= 0.1;break;
      case 't':right += 0.1;break;
      case 'g':right -= 0.1;break;
      case 'y':bottom -= 0.1;break;
      case 'h':bottom += 0.1;break;
      case 'u':top -= 0.1;break;
      case 'j':top += 0.1;break;
      case 'i':Near -= 0.1;break;
      case 'k':Near += 0.1;break;
      case 'o':Far -= 0.1;break;
      case 'l':Far += 0.1;break;
      case 'p':fov -= 1;break;
      case ';':fov += 1;break;
      case 'v':left=-1, right=1, bottom=-1, top=1;
            if (Projection == 0) {
                  Near=-1, Far=1;
            } else {
                  Near=1, Far=10;
            }
            break;
      }
      char info[128];
      sprintf(info, "(%.0f,%.0f,%.0f)"
            "(%.1f,%.1f,%.1f,%.1f,%.1f,%.1f)",
            xAngle, yAngle, zAngle,
            left, right, bottom, top, Near, Far);
```

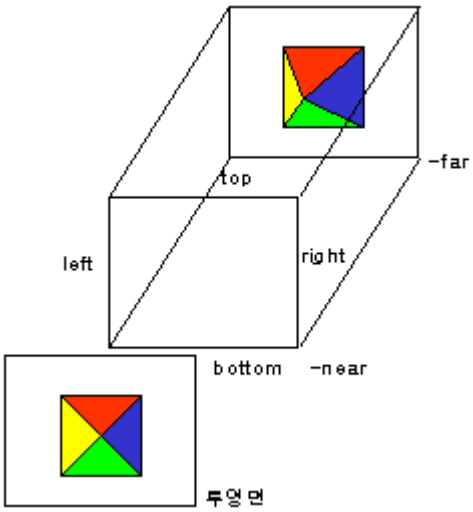
```
	glutSetWindowTitle(info);
	glutPostRedisplay();
}

void DoMenu(int value)
{
	switch(value) {
	case 1:
		Projection = 0;
		Near = -1;
		Far = 1;
		break;
	case 2:
		Projection = 1;
		Near = 1;
		Far = 10;
		break;
	case 3:
		Projection = 2;
		Near = 1;
		Far = 10;
		break;
	case 4:
		Object = 0;
		break;
	case 5:
		Object = 1;
		break;
	}
	glutPostRedisplay();
}

void DoReshape(GLsizei width, GLsizei height)
{
	glViewport(0,0,width,height);

	lastWidth = width;
	lastHeight = height;
}

void DrawPyramid()
{
	// 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();
}

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

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	switch (Projection) {
	case 0:
		glOrtho(left, right, bottom, top, Near, Far);
		break;
	case 1:
		glFrustum(left, right, bottom, top,Near,Far);
		break;
	case 2:
		GLfloat aspect = (GLfloat)lastWidth / (GLfloat)lastHeight;
		gluPerspective(fov, aspect, Near, Far);
		break;
	}

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	// 원근투영일 때는 약간 더 뒤쪽에서 보아야 한다.
	if (Projection != 0) {
		glTranslatef(0,0,-2);
	}
	glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
	glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
	glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

	if (Object == 0) {
		DrawPyramid();
	} else {
		GLUquadricObj *pQuad;
		pQuad = gluNewQuadric();
		gluQuadricDrawStyle(pQuad, GLU_LINE);
		glTranslatef(0.0, 0.0, -0.6);
		glColor3f(1,1,1);
		gluCylinder(pQuad, 0.3, 0.3, 1.2, 20, 20);
	}

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

투영 방법은 직교 투영(Orhographic)과 원근 투영(Perspective) 두 가지가 있다. 어떤 방법을 사용하는가에 따라 3차원 장면이 2차원 평면에 사상되는 방법이 달라진다. 디폴트는 직교 투영이며 위 예제도 직교 투영을 디폴트로 선택한다. 직교 투영은 거리에 상관없이 물체의 크기를 계산한다. 멀리 있는 물체라도 크기가 같다면 투영된 결과도 동일한 크기를 가진다. 직교 투영은 다음 함수로 지정한다.

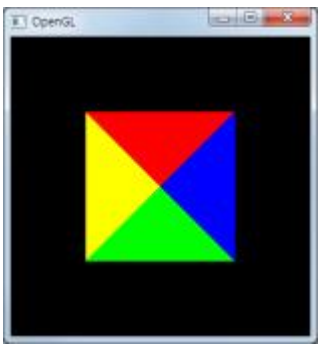
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble nearVal, GLdouble farVal);

좌우하상근원(LRBTNF) 순서대로 클리핑 영역을 설정한다. 이 영역이 평행하게 투영면에 비춰진다. 물체가 투영면에 평행하게 맺히므로 평행 투영이라고도 한다.

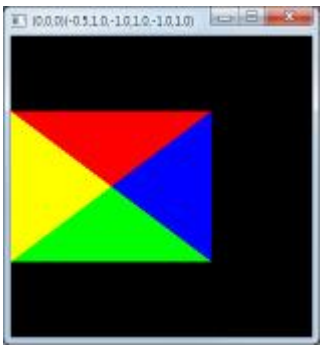


left, right는 수평으로 3차원 공간의 어느 부분을 자를 것인가를 지정한다. bottom, top은 수직으로 어느 부분을 자를 것인가를 지정한다. near, far는 가시 영역의 전방 끝과 후방 끝을 나타낸다. 시점 좌표계가 오른손 법칙을 따르기 때문에 절단면의 z 좌표는 이 값의 부호를 바꾸어야 한다. near, far가 -1, 1로 되어 있으면 앞쪽이 1이고 뒤쪽이 -1이 된다.

이 예제는 디폴트와 동일한 클리핑 영역을 사용하되 near와 far의 부호만 바꾸었다. 대신 피라미드의 꼭대기를 양수로 줌으로써 사용자쪽을 바라보도록 했다. 그래서 지금까지의 예제와 마찬가지로 피라미드를 정면에서 바라본 모양으로 그려진다.

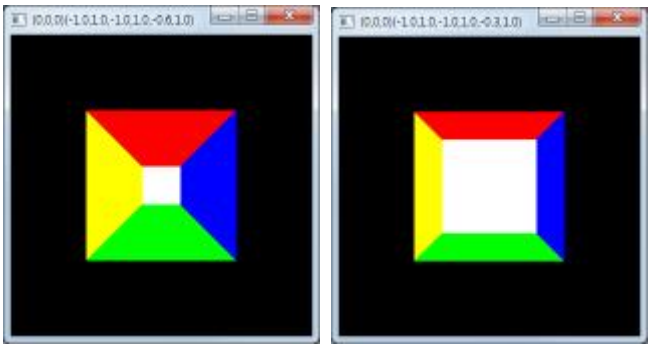


이제 클리핑 영역을 조정해 보자. 키보드의 rf, tg, yh, uj, ik, ol 키로 좌우하상근원값을 실행중에 조정할 수 있도록 해 두었으며 v키는 클리핑 영역을 리셋한다. 클리핑 영역의 왼쪽이 -1이며 피라미드의 왼쪽 좌표가 -0.5이므로 중앙과 왼쪽변의 절반쯤에 피라미드의 왼쪽면이 놓인다. r키를 눌러 왼쪽면을 -0.5까지 점점 줄여보자.

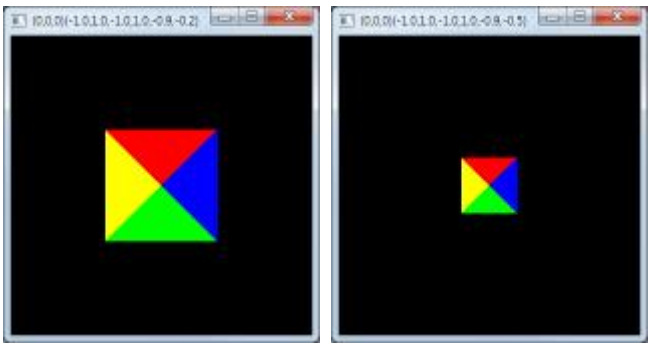


클리핑 영역이 더 좁게 설정되므로 피라미드의 왼쪽면이 클리핑 영역의 왼쪽면으로 점점 이동한다. 화면 크기는 그대로인데 투영할 영역이 줄어들었으므로 물체가 커지는 것이다. 만약 left가 -0.5보다 더 큰 값이 되면 예를 들어 -0.3이 된다면 피라미드의 왼쪽변은 클리핑 영역을 벗어나므로 화면에서 잘려 사라진다. 오른쪽 변과 위, 아래변의 클리핑 영역을 조정해도 피라미드의 위치가 같은 방식으로 조정될 것이다.

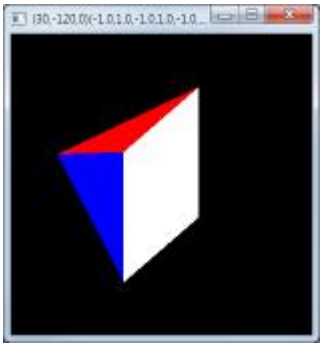
이번에는 near와 far를 조정해 보자. -near가 0.8이면 가시 영역의 전면이 피라미드의 꼭지점과 일치하므로 피라미드 전체가 보인다. 그러나 0.8보다 더 작아지면 꼭지점이 가시 영역을 벗어나므로 보이지 않는다. 피라미드의 꼭지점 보다 더 아래쪽에서 자르기 때문이다. 시점이 피라미드 안으로 들어가 버렸으므로 바닥의 흰 면이 보인다. 다음은 각각 -near를 0.6과 0.3으로 조정한 것이다.



near와 far가 같아지면 가시영역의 부피가 0이 되므로 아무것도 보이지 않는다. 이번에는 far를 조정해 보자. -far를 0보다 더 큰 값으로 조정하면 피라미드의 뒤쪽이 잘리며 0.8보다 더 커지면 아무것도 보이지 않는다. 피라미드를 회전한 상태에서 near, far를 조정해 보면 비스듬하게 잘리기도 한다.



직교 투영은 평행하게 투영하므로 공간상의 크기가 같으면 거리에 상관없이 동일한 크기로 보인다. 피라미드를 (30, -120) 각도로 회전하여 밑면이 보이도록 해 보자.

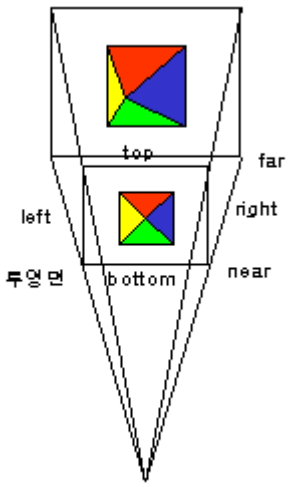


밑면의 앞쪽변과 뒷쪽변, 윗변과 아랫변이 똑같은 길이로 보인다. 실세계에서는 앞쪽과 뒤쪽의 거리가 다르기 때문에 앞쪽변이 더 길게 보이고 뒷쪽변이 짧게 보여야 한다. 책을 비스듬하게 들고 옆에서 바라보면 멀리 있는 쪽과 가까이 있는 쪽의 길이가 다른 것이 정상이다. 하지만 직교 투영은 평행하게 투영하므로 공간상의 크기가 투영면에 그대로 반영되는 특징을 보인다.

반면 원근 투영은 거리에 따라 물체의 크기가 달라진다. 똑같은 크기라도 가까이 있는 물체는 조금 크게 그리고 멀리 있는 물체는 작게 그린다. 실제로 우리가 풍경을 바라보는대로 사실적으로 그려진다. 사실감이 중요한 3D 게임에서는 주로 원근 투영을 사용한다. 원근 투영을 할 때는 다음 함수를 호출한다.

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble nearVal, GLdouble farVal);

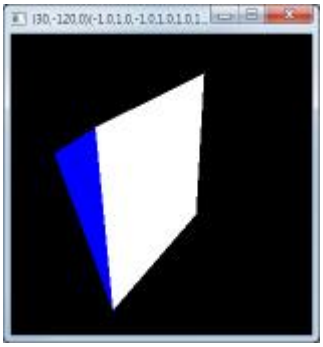
인수의 순서는 직교 투영과 동일하되 near, far가 좌표가 아니라 시점에서의 거리를 지정하므로 둘 다 반드시 양수여야 한다. 이 함수는 사각뿔의 윗부분을 잘라낸 절두체(Frustum)로 가시 영역을 설정한다.



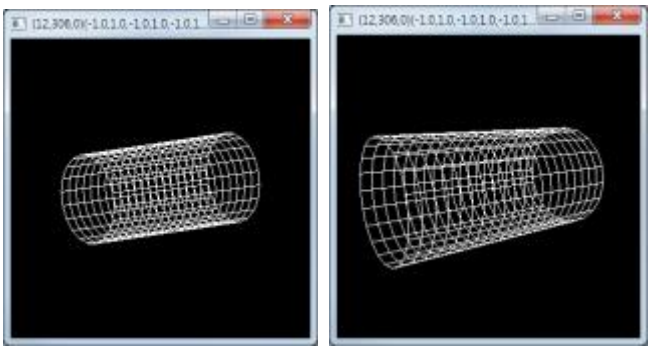
절두체는 직관성이 떨어진다. 그래서 다음 유틸리티 함수로 원근 투영을 설정하기도 한다. 시야각과 화면의 종횡비, near, far를 지정함으로써 절두체를 정의한다. 절두체와는 달리 시선이 정확하게 가시영역의 중심을 통과한다.

void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);

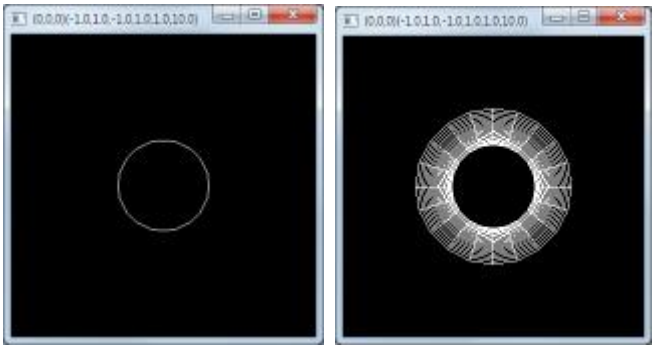
팝업 메뉴에서 Frustum이나 Perspective를 선택하면 원근 투영으로 바뀌며 Near는 1, far는 10으로 정의된다. 원근 투영을 하면 원점에 있는 물체가 너무 가까와 가시 영역에 들어오지 못하므로 z축으로 -2만큼 더 뒤쪽으로 이동시켰다.



직교 투영에 비해 앞쪽에 보이는 면이 더 길어 보인다. 회전시켜 보면 두 투영 모드의 차이점을 더 직관적으로 이해할 수 있을 것이다. 피라미드로 관찰하기 어렵다면 원통형으로 바꿔 보자. 팝업 메뉴에서 Cylinder를 선택하면 도형이 바뀔 것이다.



직교 투영을 하면 회전 각도에 상관없이 양 끝의 원 크기가 동일하다. 회전시켜 보면 어느 방향으로 돌고 있는지 잘 분간되지도 않는다. 원근 투영을 하면 가까이 있는 원이 더 크게 보여 훨씬 더 사실적이다. z키를 누르면 회전이 리셋되는데 직교 투영을 하면 양끝의 지름이 똑같아 그냥 원으로만 보이지만 원근 투영을 하면 앞쪽이 훨씬 더 크게 보여 원통형을 위에서 바라본 모양이 된다.



이 실험에서 보다시피 직교 투영보다는 원근 투영이 훨씬 더 사실적이고 우리가 실세계를 보는 것과 비슷하다. 그러나 직교 투영이 적합한 경우도 많은데 CAD 설계도면이나 지도, 아파트 평면도 등이 직교 투영으로 그려진 대표적인 예이다.


지도는 하늘에서 바라본 모양을 그린 것인데 실제로 하늘에서 땅쪽을 내려다 보면 저렇게 보이지 않는다. 시점 바로 아래의 도로는 폭이 넓어 보이고 멀리 있는 도로는 좁아 보이다가 결국은 소실점으로 사라지는 것이 정상이다. 하지만 지도는 직교 투영으로 그리는 것이 보통이므로 도로의 폭이 똑같아 보이는 것이다.

8-5.뷰포트 변환

투영 변환 후에는 뷰포트 변환이 수행된다. 투영 변환은 클리핑 영역으로 장면의 어디를 출력할 것인가를 결정하는 것이고 뷰포트 변환은 클리핑 및 투영된 평면 이미지를 윈도우의 어디쯤에다 출력할 것인지를 지정한다. 뷰포트 변환은 다음 함수로 수행한다.

void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);

(x,y)는 뷰포트의 왼쪽 아래 좌표이며 width, height는 폭과 높이이다. OpenGL에서는 윈도우의 좌표계도 좌상단이 아닌 좌하단이 원점이다. 디폴트 뷰포트 변환은 다음과 같다.

```
glViewport(0,0,width,height);
```

이 함수는 보통 윈도우의 크기가 변경되는 ReShape 콜백에서 호출한다. 좌하단은 원점이고 폭과 높이는 창의 크기와 일치하므로 디폴트대로 출력하면 윈도우 전체를 가득 채운다. 이 값을 조정하면 창의 일부만 채울 수도 있다. 다음 예제로 테스트해 보자.

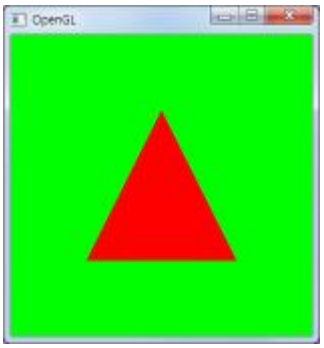
Viewport

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

void DoDisplay();
void DoReshape(GLsizei width, GLsizei height);
void DoMenu(int value);
int Action = 1;
GLsizei lastWidth, lastHeight;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
        glutCreateWindow("OpenGL");
        glutDisplayFunc(DoDisplay);
        glutReshapeFunc(DoReshape);
        glutCreateMenu(DoMenu);
        glutAddMenuEntry("전체 창 사용",1);
        glutAddMenuEntry("좌하단 사용",2);
        glutAddMenuEntry("우하단 사용",3);
```

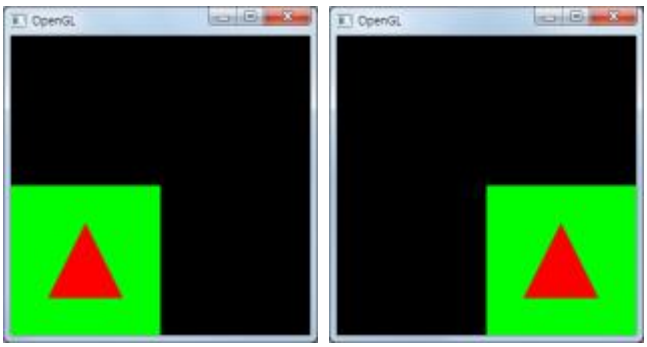
```
	glutAddMenuEntry("절대 크기 사용",4);
	glutAttachMenu(GLUT_RIGHT_BUTTON);
	glutMainLoop();
	return 0;
}

void DoMenu(int value)
{
	Action = value;
	DoReshape(lastWidth, lastHeight);
	glutPostRedisplay();
}

void DoReshape(GLsizei width, GLsizei height)
{
	lastWidth = width;
	lastHeight = height;

	switch(Action) {
	case 1:
		// 전체 창 사용
		glViewport(0,0,width, height);
		break;
	case 2:
		// 좌하단 사용
		glViewport(0,0,width/2, height/2);
		break;
	case 3:
		// 우하단 사용
		glViewport(width/2,0,width/2, height/2);
		break;
	case 4:
		// 절대 크기 사용
		glViewport(30,30,200,200);
		break;
	}
}

void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glColor3f(0.0, 1.0, 0.0);
	glRectf(-1.0, 1.0, 1.0, -1.0);

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

투영에 대해서는 별다른 지정을 하지 않았으므로 클리핑 영역은 모든 축으로 -1 ~ 1사이이다. DoDisplay에서 -1 ~ 1 영역을 가득 채우는 초록색 사각형을 그렸다. 이 사각형은 클리핑 영역을 표시한다. 그리고 안쪽에 빨간색 삼각형을 그렸다.

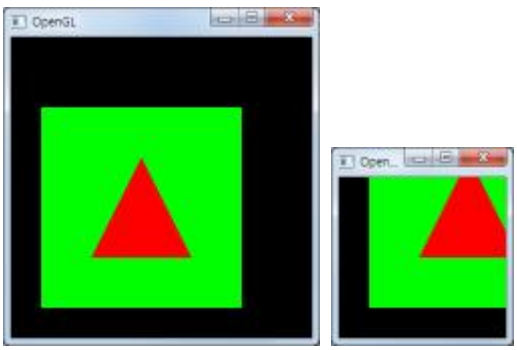
DoReshape에서는 Action에 따라 뷰포트 영역을 지정하며 Action은 팝업 메뉴로 선택한다. 디폴트대로 그리면 파란색 배경에 빨간색 삼각형이 그려질 것이다. 클리핑 영역이 뷰포트를 가득 채운다.



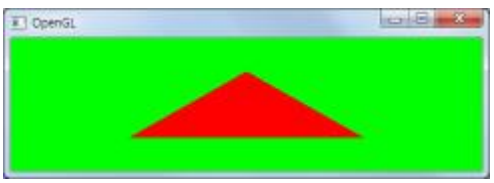
폭이나 높이를 절반으로 줄이면 뷰포트의 일부만 사용할 수도 있다. 좌하단만 사용할 때는 시작점을 원점에 고정하고 폭과 높이를 절반으로 줄이면 된다. 윈도우의 크기를 조정해도 뷰포트는 항상 동일한 위치에 있을 것이다. 시작 지점을 폭의 절반으로 지정하면 오른쪽 아래에 출력된다.



윈도우의 폭과 높이를 참조하지 않고 상수로 절대 크기를 지정할 수도 있다. 물론 원하는 위치에 원하는 크기대로 배치되지만 윈도우 크기에 무관하게 항상 일정한 크기를 가진다는 점에서 바람직하지 않다. 윈도우가 작아지면 장면의 일부가 보이지 않는다.



뷰포트 변환을 윈도우 크기에 종속적으로 지정하면 윈도우 크기 변화에 따라 장면이 찌그러지는 부작용이 있다. 다음은 윈도우를 옆으로 길게 늘인 모양이다.



윈도우의 크기에 따라 그림이 작아지거나 커질 수는 있지만 모양이 찌그러지는 것은 일반적으로 바람직하지 않다. 크기는 어쩔 수 없다 하더라도 종횡비는 가급적 맞추는 것이 바람직하다. 다음 예제는 두 가지 방법으로 종횡비를 유지한다.

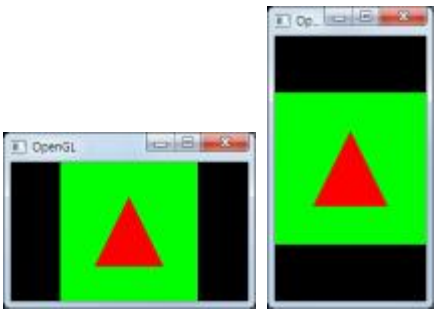
Aspect

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

void DoDisplay();
void DoReshape(GLsizei width, GLsizei height);

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

void DoReshape(GLsizei width, GLsizei height)
{
	/* 뷰포트 변환으로 종횡비 유지
	if (width > height) {
		glViewport((width - height)/2, 0, height, height);
	} else {
		glViewport(0, (height - width)/2, width, width);
	}
	//*/

	//* 직교 투영 영역을 조정하여 종횡비 유지
	glViewport(0,0,width,height);

	if (height == 0) return;

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();

	GLfloat aspect = (GLfloat)width / (GLfloat)height;
	if (width > height) {
		glOrtho(-1.0 * aspect, 1.0 * aspect, -1.0, 1.0, 1.0, -1.0);
	} else {
		glOrtho(-1.0, 1.0, -1.0/aspect, 1.0/aspect, 1.0, -1.0);
	}

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	//*/
}

void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glColor3f(0.0, 1.0, 0.0);
	glRectf(-1.0, 1.0, 1.0, -1.0);
```

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

첫 번째 방법은 뷰포트의 위치와 크기를 조정하는 것이다. 윈도우가 가로로 길쭉한 경우와 세로로 길쭉한 경우의 처리가 달라진다. 폭이 높이보다 더 큰 경우 양쪽 크기를 모두 높이로 맞추고 폭과 높이차의 절반만큼을 왼쪽에 더해 중앙으로 오게 만든다. 반대인 경우는 양쪽을 폭에 맞추고 아래쪽에 두 값차의 절반을 더한다. 길이가 짧은 쪽에 맞추되 긴쪽의 남는 여백만큼 중앙으로 이동시키는 것이다.

예를 들어 폭이 600이고 높이가 300이면 뷰포트 크기를 더 작은쪽인 300으로 맞추고 두 값의 차인 300의 절반만을 x에 더해 수평 중앙에 뷰포트를 배치하는 것이다. 수평 양쪽으로 150만큼 여백이 생기므로 그림은 정확하게 중앙에 배치되며 윈도우 크기에 상관없이 항상 일정한 종횡비를 유지한다.



두 번째 방법은 직교 투영시 클리핑 영역의 크기를 조정하는 것이다. 윈도우의 폭이 더 길다면 그 비율만큼 클리핑 영역의 x축 범위를 넓혀주면 된다. 예를 들어 폭이 높이보다 2배 더 길다면 수평 클리핑 영역을 -2 ~ 2 사이로 넓혀준다. 이렇게 되면 -1라는 좌표가 왼쪽변 끝이었다가 중간 지점이 되므로 왼쪽에 절반만큼의 여백이 생긴다.

높이가 더 길다면 이때는 종횡비가 1보다 작은 값이 되므로 수직 클리핑 영역에 이 값을 나누어야 수직 영역이 더 커진다. 어쨌든 폭, 높이의 비율 중 더 큰쪽의 클리핑 영역을 비율만큼 늘려줌으로써 그림 자체를 윈도우 종횡비에 맞추는 것이다. 굉장히 복잡한 것 같지만 알고 보면 간단한 산수일 뿐이다.

뷰포트 변환 다음 단계는 실제 화면에 출력하기 위한 래스터 변환이다. 투영 변환 후에 3차원 좌표가 2차원이 되고 뷰포트 변환에 의해 출력 윈도우 크기에 맞추어지지만 아직까지도 좌표값은 실수 타입을 유지하고 있다. 이 좌표를 화면의 각 화소에 출력하려면 결국은 정수로 바꾸어야 하는데 이 변환이 바로 래스터 변환이다. 래스터 변환 단계에도 깊이 테스트, 텍스처 맵핑같은 상당히 많은 처리가 수행되지만 우리가 개입할 여지는 별로 없다.

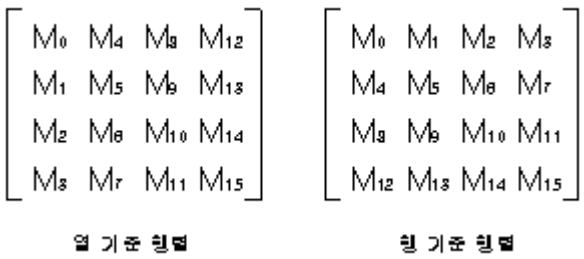
8-6. 행렬

앞에서 봤다시피 여러 가지 변환을 순차적으로 적용하려면 엄청난 연산 과정이 필요하다. 연산이 복잡하므로 굉장히 오랜 시간이 걸릴 것 같지만 행렬의 수학적 특성을 잘 이용하면 빠른 속도로 연산을 수행할 수 있다. 그래서 OpenGL은 각종 변환에 행렬을 많이 사용한다. 여기서는 행렬 연산이 수행되는 과정을 연구해 보고 왜 빠른지에 대해 알아 보자.

OpenGL이 변환에 사용하는 행렬은 4*4 크기의 행렬이다. 3차원 공간은 3개의 좌표로 구성되지만 연산의 편의를 위해 한차원 더 높은 4*4 행렬을 사용한다. 왜 그런지는 잠시 후에 연구해 보기로 하자. 메모리에서 4*4 행렬을 표현하는 방법은 여러 가지가 있는데 일단 다음 두 가지를 생각할 수 있다.

```
GLfloat matrix[4][4];
GLfloat matrix[16];
```

4*4 이차원 배열로 표현할 수도 있고 16개의 요소를 가지는 일차원 배열로 표현할 수도 있다. 이차원 배열이 더 직관적이지만 효율은 일차원 배열이 더 좋다. 2차원 배열이라고 해도 어차피 요소 개수가 고정되어 있으므로 1차원으로 표현할 수 있다. 1차원 배열로 고정 크기의 2차원 배열을 표현하는 방법은 원소를 나열하는 순서에 따라 다음 2가지로 나누어진다.



열 기준 행렬 행 기준 행렬

OpenGL은 주로 열 기준 행렬을 사용하는데 열 기준 행렬이 몇 가지 이점이 있기 때문이다. 다음 함수는 배열로부터 열 기준 행열을 읽어들인다. 배열 m은 16개의 요소를 열 기준으로 가지고 있어야 한다.

void glLoadMatrix[f,d](const GLfloat * m);

수학에서는 흔히 행 기준 행렬을 많이 사용한다. OpenGL은 주로 열 기준 행렬을 사용하지만 원한다면 행 기준 행렬도 사용할 수는 있다. 행 기준 행렬로 읽어들일 때는 다음 함수를 사용한다.

void glLoadTransposeMatrix[f, d](const GLfloat * m);

이 함수의 이름에 포함된 Transpose는 전치라는 뜻인데 열 기준 행렬의 전치 행렬이 행 기준 행렬이기 때문이다. 전치라는 것은 대각선을 기준으로 원소를 맞바꾸는 연산이다. 행렬끼리 곱할 때는 다음 함수를 호출한다.

void glMultMatrixf(const GLfloat * m);
void glMultTransposeMatrixf(const GLfloat * m);

앞에서 함수를 호출하여 수행했던 변환 예제를 이번에는 행렬 연산으로 직접 구현해 보자. 다음 예제는 주전자를 0.5, 0.5만큼 이동시킨다.

Matrix

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	GLfloat trans[16] = {
		1, 0, 0, 0,
```

```
            0, 1, 0, 0,
            0, 0, 1, 0,
            0.5, 0.5, 0, 1
        };
        glMultMatrixf(trans);

        glutWireTeapot(0.2);

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

이 행렬은 다음 수식을 정의한다. 열기준 행렬이므로 수학식으로 쓸 때는 전치됨을 주의하자.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

어떤 점 V에 행렬 M을 곱해 V' 가 생성된다. 수식으로 표현하면 V' = MV이다. 행렬끼리 곱할 때 앞 행렬의 열 수와 뒤 행렬의 행 수가 같아야 한다. 그래서 M과 V를 곱할 때는 M이 V의 앞에 와 MV가 되어야 한다. VM은 행렬의 규칙상 곱할 수 없는 수식이다.

벡터를 행렬로 표현하는 방법은 행 벡터와 열 벡터 두 가지가 있다. OpenGL은 주로 원소를 세로로 나열하는 열 벡터를 사용한다. 그 이유는 행렬이 열 기준이기 때문이다. 만약 행 벡터를 사용하고 행렬도 행 기준을 사용한다면 위 수식은 V' = VM이 될 것이다.

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix}$$

위 두 수식은 똑같은 식의 다른 표현일 뿐이다. OpenGL은 주로 전자의 수식을 사용한다. 이 행렬곱에 의해 다음 수식이 생성된다. 행렬식은 단 하나의 수식일 뿐이지만 원소끼리 연산되므로 여러 개의 다항식을 생성해낸다.

```
x' = x + 0.5
y' = y + 0.5
z' = z
1 = 1
```

x에 0.5를 더해 x' 좌표를 정의하고 y에 0.5를 더해 y' 좌표를 정의하므로 가로, 세로로 0.5만큼 이동하는 것이다. z'는 변화가 없고 마지막 수식은 1 = 1 이라는 더미 식을 만들 뿐이다. 이 행렬식을 대신 생성해 주는 함수가 바로 glTranslatef(0.5, 0.5, 0.0);이다. 이 함수를 호출하면 위 예제의 행렬을 만들어 현재 행렬에 곱함으로써 모든 정점을 이동시키는 효과가 나타난다.

다음은 확대를 해 보자. 앞 예제와 형식은 동일하되 행렬 내부의 원소들이 다를 뿐이다. 원소가 달라지면 다항식의 계수와 더해지는 값이 달라짐으로써 변환 연산도 달라진다.

```
void DoDisplay()
{
```

```
	glClear(GL_COLOR_BUFFER_BIT);

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	GLfloat scale[16] = {
		1.5, 0, 0, 0,
		0, 1.5, 0, 0,
		0, 0, 1, 0,
		0, 0, 0, 1
	};
	glMultMatrixf(scale);

	glutWireTeapot(0.2);

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

이 수식은 glScalef(1.5, 1.5, 1.0)와 동일하다. 어째서 그렇게 되는지 행렬을 곱해 보자.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

이 행렬에 의해 다음 수식이 도출된다. 원래의 x값에 1.5를 곱해 새로운 x를 정의하고 y도 마찬가지로 1.5를 곱한다. 그러므로 가로, 세로로 1.5배 확대되는 것이다.

```
x' = 1.5 * x
y' = 1.5 * y
z' = z
1 = 1
```

다음은 조금 복잡한 회전을 해 보자. z축을 중심으로 45도 회전하는 glRotatef(45.0, 0.0, 0.0, 1.0) 호출문을 행렬로 구현하는 것이다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	GLfloat rotate[16] = {
		cos(45.0*3.14/180), sin(45.0*3.14/180), 0, 0,
		-sin(45.0*3.14/180), cos(45.0*3.14/180), 0, 0,
		0, 0, 1, 0,
		0, 0, 0, 1
	};
	glMultMatrixf(rotate);

	glutWireTeapot(0.2);

	glPopMatrix();
	glFlush();
```

```
}
```

회전은 각도의 개념이 들어가므로 삼각함수를 사용하는데 수식의 증명은 생략한다. 수학 함수를 사용하므로 math.h를 인클루드해야 한다. OpenGL은 360분법의 각도를 사용한데 비해 수학 함수는 라디안을 사용하므로 360분법의 각도로 바꿔서 사용해야 한다. 이 수식은 z축을 기준으로 45도만큼 회전시킨다. 축이 바뀌면 위 행렬의 모양도 달라진다.

다음은 확대와 이동을 동시에 수행해 보자. 두 가지 이상의 변환을 복합 변환이라고 하며 둘 이상의 행렬이 순서대로 곱해진다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	GLfloat trans[16] = {
		1, 0, 0, 0,
		0, 1, 0, 0,
		0, 0, 1, 0,
		0.5, 0.5, 0, 1
	};
	glMultMatrixf(trans);

	GLfloat scale[16] = {
		1.5, 0, 0, 0,
		0, 1.5, 0, 0,
		0, 0, 1, 0,
		0, 0, 0, 1
	};
	glMultMatrixf(scale);

	glutWireTeapot(0.2);

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

곱해지는 행렬이 앞쪽에 붙으며 가해지는 변환의 역순으로 곱해진다. 확대행렬을 S, 이동 행렬을 T라고 할 때 새로운 정점은 다음과 같이 계산된다.

$$V' = TSV$$

수식으로 풀어 보면 다음과 같다.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 0 & 0 & 0.5 \\ 0 & 1.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$x' = 1.5x + 0.5$$

$$y' = 1.5y + 0.5$$

$$z' = z$$

$$1 = 1$$

이때 TS는 미리 계산해 둘 수 있다. 행렬은 교환 법칙은 성립하지 않지만 결합 법칙은 성립한다. TS를 미리 곱해 M을 정의하고 V'=MV 연산을 해도 결과는 동일하다. 소스를 다음과 같이 바꾸어도 효과는 동일하다. 앞 예제의 TS를 미리 곱해 하나의 행렬을 정의하고 행렬 곱셈을 한번만 수행한다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	GLfloat transscale[16] = {
		1.5, 0, 0, 0,
		0, 1.5, 0, 0,
		0, 0, 1, 0,
		0.5, 0.5, 0, 1
	};
	glMultMatrixf(transscale);

	glutWireTeapot(0.2);

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

두 행렬을 미리 곱한 후 벡터에 곱해도 결과는 각각 곱한 것과 동일하다. 이것이 바로 행렬 연산의 우수함이다. 3차원 공간의 물체들을 제 위치에 적당한 크기로 배치해 놓고 투영하여 뷰포트에 배치하기까지는 굉장히 많은 연산들이 필요하다. 각 연산들이 모두 행렬의 곱셈으로 표현되므로 이 변환은 다음과 같이 수식으로 쓸 수 있다.

V'=POTSRLV

시점 좌표 바꾸고 회전, 확대, 이동한 후 직교 투영, 뷰포트 변환을 거쳐 최종적으로 화면에 보이게 된다. 실제로는 이보다 훨씬 더 복잡한 변환이 수행되기도 한다. 이 변환을 위해 각 정점마다 이 행렬들을 일일이 곱

할 필요가 없다. POTSRL 행렬들을 미리 계산하여 M에 대입해 두면 이후의 변환은 다음 하나의 수식으로 처리된다.

V'=MV

모든 변환 과정을 하나의 행렬 M에 모을 수 있으며 그래서 OpenGL은 모든 변환을 현재 행렬에 누적시킨다. 이것이 가능한 이유는 모든 변환이 행렬의 곱셈으로만 처리되기 때문이다. 행렬의 곱셈은 결합 법칙이 성립됨을 이용하여 미리 계산해 놓고 일관되게 적용한다. 사실 평행 이동 변환의 경우는 곱셈보다는 덧셈으로 더 간단하게 연산할 수 있다. x, y, z 세 개에 대해 dx, dy, dz 행렬을 더하면 된다.

x' = x + dx

y' = y + dy

z' = z + dz

이 세 개의 다항식을 수행하면 각 축으로 원하는 거리만큼 평행 이동할 수 있다. 이 수식은 다음과 같은 행렬 덧셈으로 표현할 수 있다.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix}$$

그러나 평행 이동만 덧셈으로 계산하자면 미리 곱해 놓는 전략을 쓸 수가 없다. 이 덧셈식을 어떻게 하든간에 곱셈 식으로 바꿔야 한다. 하지만 곱해지는 행렬 원소에 어떤 값을 사용하더라도 행렬 곱으로는 상수를 더할 방법이 없다.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

? 자리에 어떤 값을 넣어도 x'에 기존의 y, z와 상관없는 상수를 더하지 못한다. 평행 이동하려면 x'=x+dx 수식을 만들 수 있어야 한다. 그래서 3 * 3 행렬의 곱으로는 일반화를 할 수가 없으며 더미식을 위해 4*4행렬이 필요하다.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

이렇게 하면 제일 오른쪽열에 dx, dy, dz를 둠으로써 변환 후의 좌표에 상수를 더할 수 있다. OpenGL은 3차원 그래픽이면서도 4차원의 좌표를 사용함으로써 모든 행렬 연산을 곱셈 하나로 통일하며 그럼으로써 행렬 연산을 미리 해 둘 수 있는 것이다.

모든 연산을 곱셈으로 처리하고 미리 계산해 두는 것이 왜 빠른지 실생활에서 이런 연산의 예를 들어 보자. 정기 예금의 지급 금액을 계산하는데는 여러 가지 요소들이 개입된다. 간단하게 다음과 같다고 하자.

지급액 = 원금 * 이자율 * 세율

계산을 일관되게 하기 위해 모든 연산은 곱셈으로 변환해야 한다. 예를 들어 이자율이 10%라면 지급액 = 원금 + 원금 * 0.1 식으로 계산할 수 있지만 덧셈이 들어간다. 이 식을 지급액 = 원금 * 1.1로 곱셈만으로도 표기할 수 있다. 이자율이 10%이고 세율이 2%라 하면 이 식은 다음과 같아진다.

지급액 = 원금 * 1.1 * 0.98

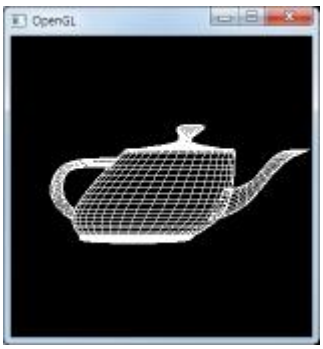
이 수식대로 고객 1000명의 지급액을 계산한다고 할 때 매 고객의 원금마다 1.1 곱하고 0.98을 곱할 필요없이 1.1과 0.98을 미리 곱한 1.078을 곱하면 된다. 수수료니 부가세니 하는 것들이 더 많이 개입되더라도 단 하나의 값을 미리 계산해 놓고 일괄 적용할 수 있으므로 효율적이다. 이것이 가능한 이유는 곱셈은 결합 법칙이 성립하기 때문이다.

OpenGL의 행렬도 이런 식이기 때문에 아무리 중간 변환이 많더라도 성능상의 불이익이 거의 없다. 앞에서 세가지 변환으로 중심점 회전을 해 보았는데 세 변환을 한꺼번에 하더라도 결국 하나의 행렬만 곱해지므로 속도는 원점 회전과 차이가 없다. 뿐만 아니라 행렬 계산은 하드웨어 가속기가 직접 처리한다. 행렬 곱셈은 사람이 하기에는 복잡한 계산이지만 연산 절차가 단순해서 기계가 하기에는 전혀 어렵지 않다. 따라서 변환이 상상 이상의 초고속으로 처리된다.

사실 OpenGL을 사용하기만 한다면 굳이 복잡한 내부적인 행렬 연산까지 다 알아야 할 필요는 없다. 이동, 확대, 투영 등의 연산에 필요한 행렬을 만들어서 곱해주는 함수들이 있기 때문이다. 그러나 내부를 알면 이 함수들의 동작을 좀 더 잘 이해할 수 있고 이상 동작시 대처도 더 잘하게 된다. 또한 함수가 지원하지 않는 변환도 행렬로 직접 수행할 수 있어 자유도가 높아진다. 가령 다음 예제는 물체를 기울인다.

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

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();

	GLfloat sheer[16] = {
		1, 0, 0, 0,
		0.5, 1, 0, 0,
		0, 0, 1, 0,
		0, 0, 0, 1
	};
	glMultMatrixf(sheer);

	glutWireTeapot(0.5);

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

행렬을 수식으로 바꾸어 계산해 보면 x' = x + 0.5y가 된다. x 좌표에 y의 절반이 더해지므로 위로 올라갈수록 x 좌표가 더 오른쪽으로 치우치며 원점 아래에서는 오히려 x가 감소한다. 그래서 주전자가 비스듬하게 기울어지는 것이다.



OpenGL이 기울어지는 변환을 함수로 지원하지 않더라도 행렬을 직접 만들어 곱하면 이런 것도 가능하다. 행렬 원소의 값에는 별다른 제약이 없으므로 삼각함수나 지수함수 등을 동원하면 끝을 뾰족하게 깍는다거나 휘게 만드는 훨씬 더 복잡한 변환도 가능하다.

8-7.출력 영역의 제한

별다른 제한이 없는 한 모든 출력문은 좌표 공간으로 출력된다. 물론 클리핑이나 뷰포트 변환 단계에서 잘려 나가는 부분이 있지만 그래도 일단은 출력된 후 잘린다. 특정 영역을 아예 처음부터 출력되지 않도록 제한해야 하는 경우가 있는데 이럴 때는 두 가지 방법을 사용할 수 있다. 첫번째 방법은 가위를 사용하는 것이다. 가위 기능을 켜 주고 표시할 영역을 알려 주기만 하면 이 영역으로 출력이 제한된다.

```
glEnable(GL_SCISSOR_TEST);
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

좌표는 윈도우 좌하단을 원점으로 하며 폭과 높이를 지정하므로 사각형 형태로만 제한할 수 있다. 이 기법은 출력 영역을 최소화하는 효과가 있다. 장면의 대부분은 그대로 유지되고 일부만 변한다면 전체 장면을 모두 그릴 필요없이 변하는 부분만 그리면 된다. 이때 다시 그릴 필요가 있는 부분에 대해 가위 영역으로 지정하면 훨씬 더 빠르게 그려진다. 무효 영역을 최소화함으로써 출력 속도를 높이는 기법이다.

좀 더 복잡한 형태의 출력 제한은 스텐실 버퍼로 수행한다. 스텐실 버퍼에 임의의 모양을 그려 두고 이 버퍼와 화면 버퍼를 연산하여 조건에 맞는 부분만 출력할 수 있다. 이 기법을 사용하려면 별도의 스텐실 버퍼를 준비해야 한다. 디스플레이 모드를 초기화할 때 GLUT_STENCIL 플래그를 지정해야 하며 화면을 삭제할 때 스텐실 버퍼도 같이 삭제한다. 깊이 버퍼를 사용하는 방법과 동일하다.

```
glutInitDisplayMode(GLUT_RGB | GLUT_STENCIL);
glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

삭제시 스텐실 버퍼에 쓸 값은 glClearStencil 함수로 지정하는데 디폴트가 0이므로 보통은 디폴트를 받아들이면 된다. 스텐실 기능이 켜지면 출력할 때마다 스텐실 테스트를 수행하여 허가된 영역에만 출력을 내 보낸다. 다음 함수로 스텐실 테스트 방법을 지정한다.

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

func는 테스트 함수이다. ref는 비교 대상값이며 mask는 비교전에 대상값과 스텐실값에 & 연산을 취해 특정 비트를 마스크 오프시킨다. 결국 비교 대상은 ref & mask와 스텐실버퍼값 & mask이이다. mask의 디폴트는 모든 비트가 1이므로 이 경우 ref와 스텐실 버퍼에 저장된 값을 비교하게 된다. 테스트 함수는 다음과 같다.

함수	설명

GL_NEVER	항상 실패한다.
GL_ALWAYS	항상 성공한다.
GL_LESS	비교값이 더 작을 때 성공
GL_LEQUAL	비교값이 더 작거나 같을 때 성공
GL_GREATER	비교값이 더 클 때 성공
GL_GEQUAL	비교값이 더 크거나 같을 때 성공
GL_EQUAL	비교값과 스텐실 버퍼값이 같을 때 성공
GL_NOTEQUAL	비교값과 스텐실 버퍼값이 다를 때 성공

스텐실 테스트를 수행한 후 스텐실 버퍼의 값도 변경되는데 다음 함수로 변경 방식을 지정한다. 세가지 경우에 대해 각각 어떤식으로 변경할 것인가를 지정한다.

void glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass);

sfail은 스텐실 테스트 실패시의 동작을 지정한다. dpfail은 스텐실 테스트는 성공했지만 깊이 테스트는 실패했을 때의 동작을 지정한다. dppass는 스텐실 테스트와 깊이 테스트를 모두 성공했을 때의 동작을 지정한다. 각각 다음과 같은 동작을 지정할 수 있다.

동작	설명
GL_KEEP	현재값을 유지한다.
GL_ZERO	0으로 기록한다.
GL_REPLACE	ref 비교값을 기록한다.
GL_INCR	값을 1증가시킨다.
GL_INCR_WRAP	값을 1증가시킨다. 최대값에 도달하면 0으로 돌아간다.
GL_DECR	값을 1감소시킨다.
GL_DECR_WRAP	값을 1감소시킨다. 0보다 작아지면 최대값으로 돌아간다.
GL_INVERT	비트 반전시킨다.

설명만 읽어서는 스텐실의 동작 방식을 이해하기 쉽지 않다. 다음 예제는 가위와 스텐실 기능을 테스트한다. 팝업 메뉴로 옵션을 바꿔 가며 결과를 비교해 보자.

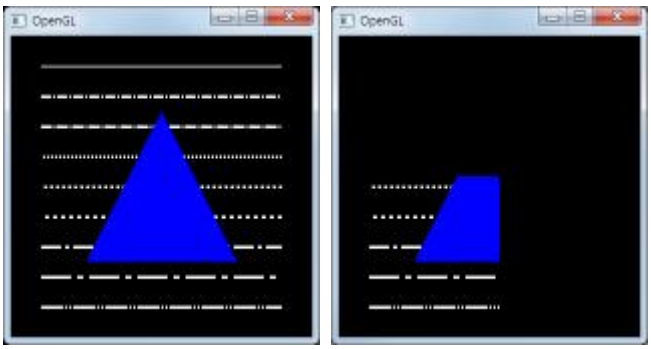
Stencil

```
#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 nx, ny;
BOOLEAN bScissor;
BOOLEAN bStencil;
BOOLEAN bEqual;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutInitDisplayMode(GLUT_RGB | GLUT_STENCIL);
```

```
	glutCreateWindow("OpenGL");
	glutDisplayFunc(DoDisplay);
	glutKeyboardFunc(DoKeyboard);
	glutCreateMenu(DoMenu);
	glutAddMenuEntry("Scissor ON",1);
	glutAddMenuEntry("Scissor OFF",2);
	glutAddMenuEntry("Stencil ON",3);
	glutAddMenuEntry("Stencil OFF",4);
	glutAddMenuEntry("Equal",5);
	glutAddMenuEntry("Not Equal",6);
	glutAttachMenu(GLUT_RIGHT_BUTTON);
	glutMainLoop();
	return 0;
}

void DoKeyboard(unsigned char key, int x, int y)
{
	switch(key) {
	case 'a':nx -= 0.1;break;
	case 'd':nx += 0.1;break;
	case 'w':ny += 0.1;break;
	case 's':ny -= 0.1;break;
	}
	glutPostRedisplay();
}

void DoMenu(int value)
{
	switch(value) {
	case 1:
		bScissor=TRUE;
		break;
	case 2:
		bScissor=FALSE;
		break;
	case 3:
		bStencil=TRUE;
		break;
	case 4:
		bStencil=FALSE;
		break;
	case 5:
		bEqual=TRUE;
		break;
	case 6:
		bEqual=FALSE;
		break;
	}
	glutPostRedisplay();
}

void DoDisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

	// 가위 테스트
	if (bScissor) {
		glEnable(GL_SCISSOR_TEST);
	} else {
		glDisable(GL_SCISSOR_TEST);
```

```
        }
        glScissor(10,10,150,150);

        if (bStencil) {
            glEnable(GL_STENCIL_TEST);
        } else {
            glDisable(GL_STENCIL_TEST);
        }

        // 스탠실 버퍼에 마킹만 한다.
        glStencilFunc(GL_NEVER, 0x0, 0x0);
        glStencilOp(GL_INCR, GL_INCR, GL_INCR);

        // 수평 선 그음
        glColor3f(1,1,1);
        GLint arFac[] = { 1,    1,    1,    2,    3,    4,    2,    3,    2};
        GLushort arPat[]={0xaaaa,0x33ff,0x57ff,0xaaaa,0xaaaa,0xaaaa,0x33ff,0x33ff,0x57ff };
        glEnable(GL_LINE_STIPPLE);
        glLineWidth(3);
        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++;
        }

        // 스텐실 값과 비교하여 특정 영역에만 출력한다.
        glStencilFunc(bEqual ? GL_EQUAL:GL_NOTEQUAL, 0x1, 0xff);

        // nx, ny 위치에 삼각형 그림
        glColor3f(0,0,1);
        glBegin(GL_TRIANGLES);
        glVertex2f(nx + 0.0, ny + 0.5);
        glVertex2f(nx -0.5, ny - 0.5);
        glVertex2f(nx + 0.5, ny - 0.5);
        glEnd();

        glFlush();
    }
```

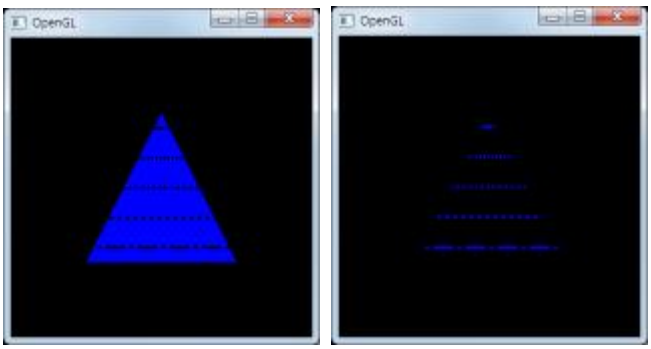
선 모양을 바꿔가며 수평선 여러 개를 그었다. 그리고 그 위에 파란색 삼각형을 그렸다. 둘 다 앞에서 이미 실습해 봤던 코드들이다. 별다른 제약 조건이 없다면 두 그림은 겹쳐서 출력된다. 가위 기능을 켜면 좌하단 (10,10)좌표에서 150, 150만큼의 영역에만 출력되고 그 바깥은 잘린다.



가위가 지정하는 좌표는 윈도우 좌표이므로 창의 크기를 줄여도 제한되는 영역은 동일하다. 그래서 제한 영역보다 더 작게 윈도우를 만들면 모두 보이기도 한다.

스텐실 기능을 켜면 점선이 직접적으로 보이지 않는다. 점선을 그리기 전에 스텐실 테스트를 GL_NEVER로 지정했으므로 선은 결국 그려지지 않는 셈이다. 대신 스텐실 버퍼에 점선이 그려지는 영역이 1씩 증가하여 1의 값을 갖게 된다.

삼각형을 그릴 때는 스텐실 함수를 1과 같거나 다른 값으로 지정했으므로 삼각형의 모든 영역이 그려지지 않고 스템실 버퍼의 값과 비교하여 점선이 지나갔던 영역이나 또는 그 반대 영역만 그려진다. 직전에 그렸던 그림은 색상 버퍼에는 기록되지 않지만 스텐실 버퍼에 기록되어 다음 출력에 영향을 미친다.



복잡한 모양으로 스텐실을 만들어 둘 수 있어 임의의 모양으로 출력을 제한할 수 있다. 예를 들어 복잡한 무늬로 글자를 쓰고 싶다면 글자 모양을 스텐실 버퍼에 먼저 쓰고 이 버퍼에 글자가 지나간 부분에 대해서만 무늬를 칠하면 된다.