

Chapter 10 - 동명대학교 게임공학과 3D 그래픽스 프로그래밍

정점별 법선



“소프트웨어 재사용 전에, 우선은 사용부터 가능해야 한다.”

“Before software should be reusable, it should be usable”

– 랄프 존슨 Ralph Johnson

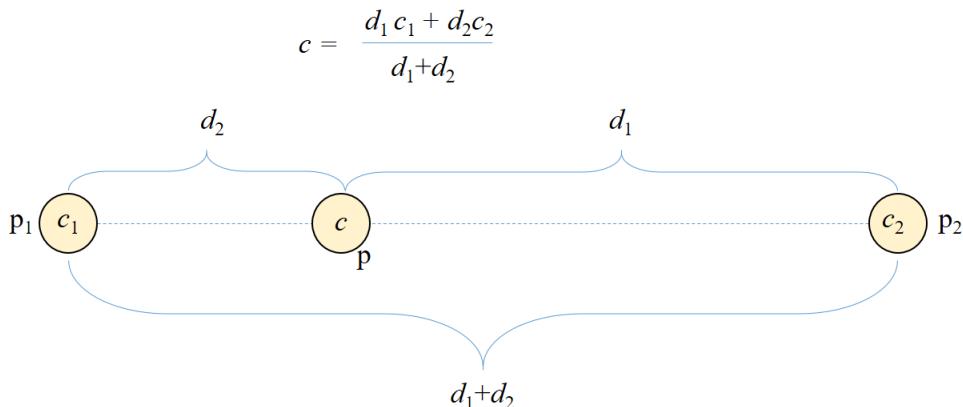
이 장에서 생각할 문제

- ❖ 파일에 담길 기하 객체 정보 읽기
- ❖ 기하 객체를 효율적으로 다룰 수 있는 기술
- ❖ 메시 테이터의 가시화를 빠르게 하는 방법

10.1 삼각형 세 정점의 법선이 다르다면?

퐁^{Phong} 셰이딩은 눈에 관찰되는 색을 결정할 때 법선과 광원을 이용하였다. 삼각형 한 면의 법선 벡터는 삼각형 내의 어디든지 동일하다. 따라서 우리는 이전 장에서 하나의 삼각형을 그릴 때, 해당 면의 법선을 계산하고 삼각형을 구성하는 정점에 이 법선을 적용함으로써 하나의 면 내부가 (거의) 같은 색상과 밝기로 칠해지는 것을 확인하였다. 그런데, 각 정점의 법선을 다르게 적용하면 어떻게 될까? 퐁 셰이딩 방식에 의해 각 정점의 색상과 밝기가 다르게 결정될 것이다. 그러면 삼각형 내부는 서로 다른 세 가지 색상을 부드럽게 보간^{interpolation}하여 채워질 것이다. 이 보간은 어떻게 계산될까?

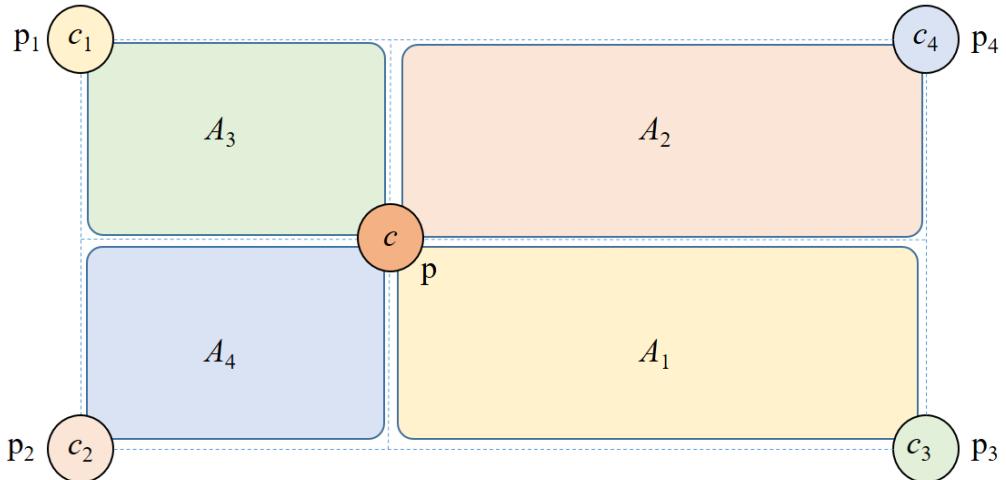
우선 가장 간단한 두 지점 사이의 선형 보간^{linear interpolation}을 생각해 보자. 그림 1과 같이 두 지점 p_1 과 p_2 가 각각 c_1 과 c_2 의 값을 가지고 있을 때, 그 사이에 있는 p 의 값 c 는 무엇이 될까? 그림에서 p 가 p_1 의 위치에 가까워진다고 하면, d_2 는 작아지고 d_1 은 커질 것이다. 이때 값 c 는 c_1 에 가까워진다. 이것은 d_1 이 커질수록 c 는 c_1 에 가까워지고, d_1 이 바로 p_1 이 c 값을 결정하는 데에 미치는 영향력의 크기라는 것을 알 수 있다. 반대로 p 를 p_2 의 방향으로 움직여보자. d_2 가 증가할 것이다. 이것은 p_2 의 영향력이 커지는 것을 의미한다. 따라서 두 점 사이에 있는 p 의 값은 d_1 와 d_2 를 가중치로 c_1 과 c_2 의 값을 혼합하면 된다. 이것이 두 개의 값 사이를 선형으로 보간하는 방법에 대해 이해 방법이다.



[그림 1] 두 지점 사이를 연결하는 1차원 공간상의 값을 선형보간하는 방법

그렇다면 데이터가 2차원 공간에 존재할 경우에는 어떻게 보간할 수 있을까? 이때는 쌍선형 보간^{bi-linear interpolation}을 사용한다. 그림 2와 같이 네 개의 점 p_1, p_2, p_3, p_4 이 축에

정렬된 형태로 직사각형을 이루고 있을 때를 생각해 보자. 보간의 대상이 되는 점 \mathbf{P} 는 한 축으로 일정한 길이, 다른 한 축으로 일정한 길이 만큼 이동할 수 있다. 그러면 이 점을 기준으로 직사각형 영역은 네 개의 영역 A_1, A_2, A_3, A_4 가 형성된다. 만약 점 \mathbf{P} 를 \mathbf{p}_1 쪽으로 옮겨보자. 값은 c_1 에 가까워지고, A_1 의 넓이가 커질 것이다. 즉 A_1 은 \mathbf{p}_1 의 영향력이다.

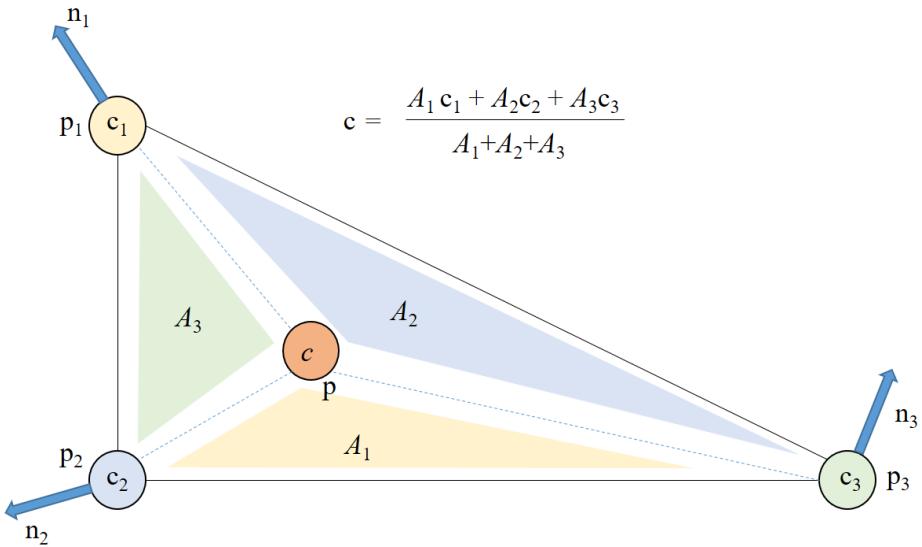


[그림 2] 쌍선형 보간의 개념

이것은 결국 다음과 같이 \mathbf{P} 의 보간값 c 는 각 지점에서의 값에 대해 A_i 를 가중치로 곱해서 얻는 값이 된다는 것이다.

$$c = \frac{A_1c_1 + A_2c_2 + A_3c_3 + A_4c_4}{A_1 + A_2 + A_3 + A_4}$$

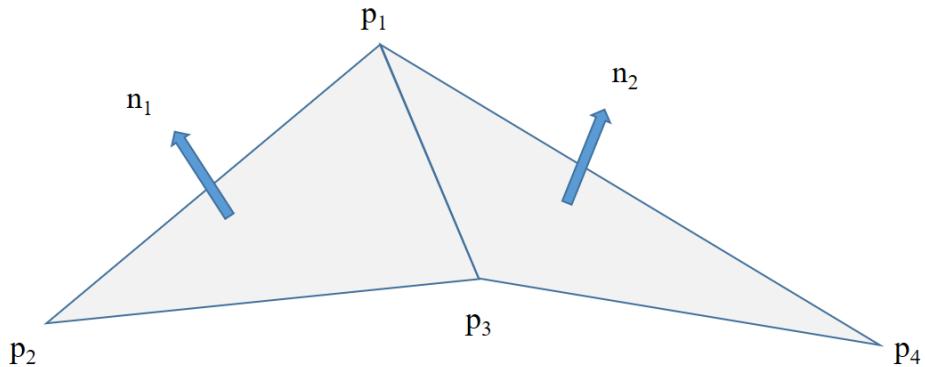
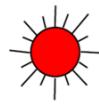
이러한 원리를 이용하여 삼각형의 세 정점 $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ 이 서로 다른 법선 벡터 $\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$ 를 가진 경우에 이 면의 음영 계산이 어떻게 되어 칠해지는지 생각해 보자. 각각의 정점은 풍 셰이딩을 이용하여 색상이 결정될 것이다. 이 값들을 $\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$ 이라고 할 수 있다. 이를 그림으로 표현하면 그림 3과 같을 것이다. 여기에서도 보간의 대상이 되는 지점 \mathbf{P} 를 세 정점으로 옮겨보면 커지는 영역이 있을 것이다. 이를 가중치로 이용하여 보간할 수 있다.



[그림 3] 삼각형 세 정점의 색을 기준으로 내부의 한 지점에 대한 색을 보간으로 구하기

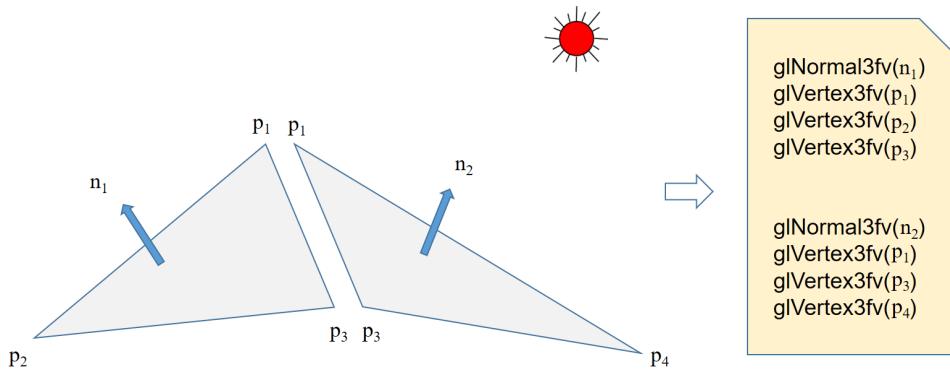
이렇게 삼각형 세 개의 색이 정해졌을 때, 삼각형 내부의 색을 보간하는 방식을 이용하여 면을 칠하는 기법을 구로 Gouraud 세이딩이라고 한다. 우리가 앞 장에서 삼각형의 꼭지점에 임의의 색을 지정했을 때, 삼각형 내부의 색상이 부드럽게 변화하는 것을 보았다. 그 결과가 바로 이 구로 세이딩을 통해 생성된 것이다. 다만 앞 장에서는 명시적으로 색을 지정했지만, 세 정점마다 법선 벡터만 지정할 수도 있다. 법선 벡터가 지정되면 조명 모델에 반응하게 되며, 조명 계산을 통해 얻어진 색을 보간하는 것이다. 면을 구성하는 모든 지점에 대해 조명 계산을 하는 것은 계산량이 크기 때문에 OpenGL이나 DirectX에서는 기본적으로 이러한 방식으로 세이딩을 한다.

그러면 이렇게 정점마다 다른 법선 벡터를 주는 것은 어떤 의미가 있는지 살펴보자. 우선 그림 4와 같이 두 개의 삼각형이 서로 다른 법선 n_1 과 n_2 를 가지고 있다고 가정해 보자.



[그림 4] 두 개의 삼각형이 서로 다른 방향을 향하고 있는 예

이를 그리는 방법은 그림 5와 같이 두 개의 삼각형을 완전히 별개로 간주하고 그리는 방법이다. 그러면 각각의 면을 그릴 때마다 하나씩의 법선이 적용되고, 하나의 면 내의 모든 점들은 같은 법선을 가진 것으로 칠해질 것이다.

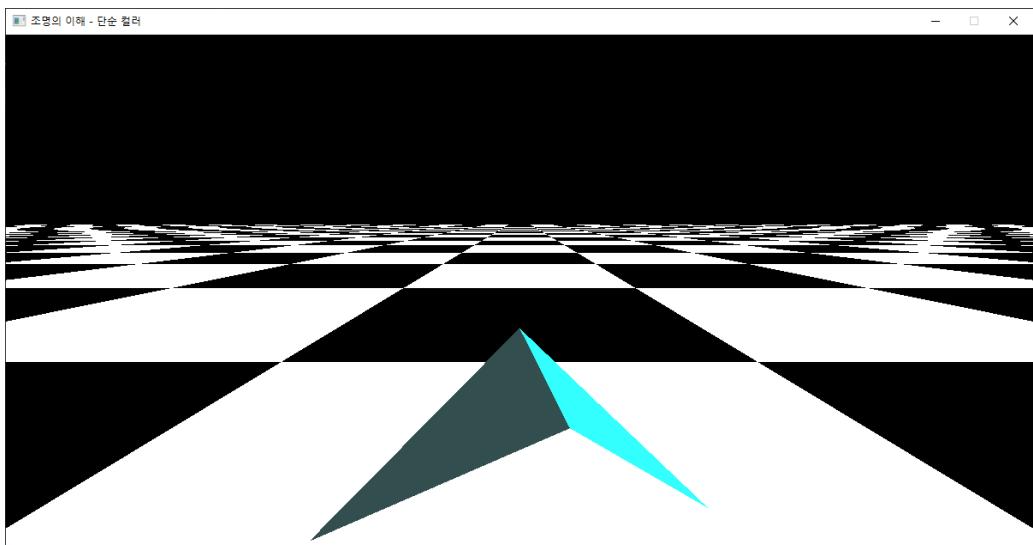


[그림 5] 삼각형을 그릴 때마다 하나의 법선을 부여하는 방식

면마다 하나의 법선이 제공되었기 때문에 그림 6과 같이 하나의 면은 평평한 면으로 칠해지게 될 것이다.

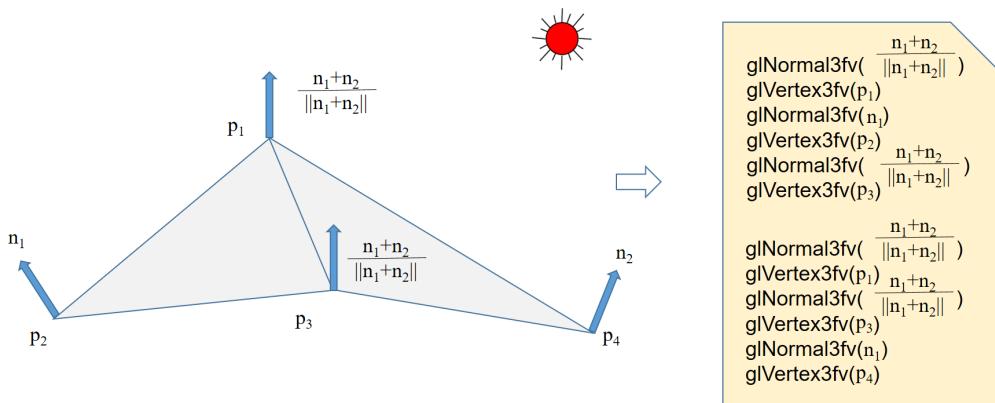
```
glEnable(GL_LIGHTING)
v = 1/math.sqrt(2)

glBegin(GL_TRIANGLES)
glNormal3f (-v, v, 0)
glVertex3f(0,1,0)
glVertex3f(-1,0,0)
glVertex3f(0,1,1)
glNormal3f (v, v, 0)
glVertex3f(0,1,0)
glVertex3f(0,1,1)
glVertex3f(1,0,0)
glEnd()
```



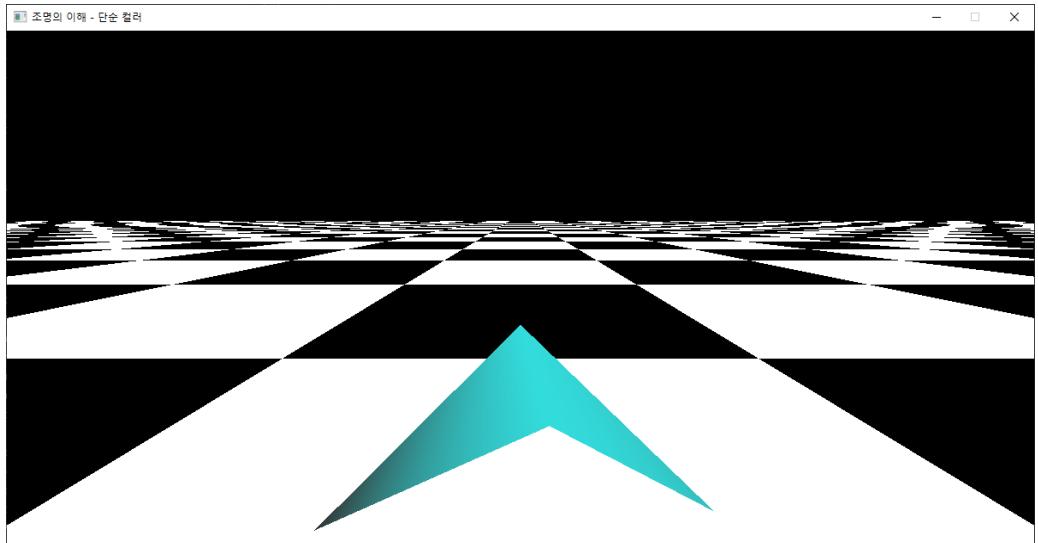
[그림 6] 면마다 법선 벡터를 하나씩 지정한 결과

그러면 정점마다 법선을 지정한다는 것은 어떤 의미를 가질까? 앞의 예에서 두 개의 삼각형에 동시에 적용되는 정점은 p_1 과 p_3 이다. 이들의 법선은 n_1 으로 지정될 때도 있고, n_2 로 지정될 때도 있다. 그러면 이 정점들의 법선은 이 두 법선 벡터를 모두 고려하여 결정하면 어떨까? 즉, $n_1 + n_2$ 를 정규화하여 $n_1 + n_2 / \|n_1 + n_2\|$ 를 적용하는 것이다. 이것은 그림 7과 같은 의미를 갖는다.



[그림 7] 하나의 정점에 대해 자신이 사용된 모든 면의 법선을 고려해 법선 정하기

이런 방식으로 코딩을 하게 되면 그림 8과 같은 결과를 얻게 될 것이다. 그림에서 보는 바와 같이 하나의 정점이 하나의 법선을 갖게되면 정점 공유하는 서로 다른 면의 음영이 경계면을 가지고 확연히 달라지는 현상이 사라지게 된다. 이것은 실제로는 두 개의 면으로 이루어졌지만 부드러운 곡면처럼 음영이 바뀌도록 하는 효과를 갖는다. 따라서 메시를 렌더링할 때 부드러운 곡면 표현을 하려면 이러한 방식으로 법선을 설정해야 한다.



[그림 8] 하나의 정점에 대해 하나의 법선을 적용한 결과

9.2 메시 데이터에 조명 적용하기

메시 데이터에 조명이 적용될 수 있도록 하는 방법은 메시의 각 면을 그릴 때마다 적절한 법선 벡터가 적용되도록 하는 것이다. 우선은 각 면에 대해 하나의 법선 벡터를 계산해 저장했다가, 면을 그릴 때마다 해당 면의 법선을 적용하는 방식을 구현해 보자.

우리가 이미 만들어 보았던 메시 클래스를 수정할 것이다. 메시 클래스에는 loadData 메소드가 있었다. 면마다 계산될 법선 벡터를 저장할 공간은 self.normalBuffer라고 하자. 이것은 면의 개수 self.nF가 결정되면 필요한 공간이 얼마인지 결정할 수 있다. 하나의 면에 3차원 벡터로 표현되는 법선 벡터가 하나씩 있으므로 self.nF * 3 개의 원소를 가진 배열을 준비하면 된다. 이는 다음과 같이 구현할 수 있다.

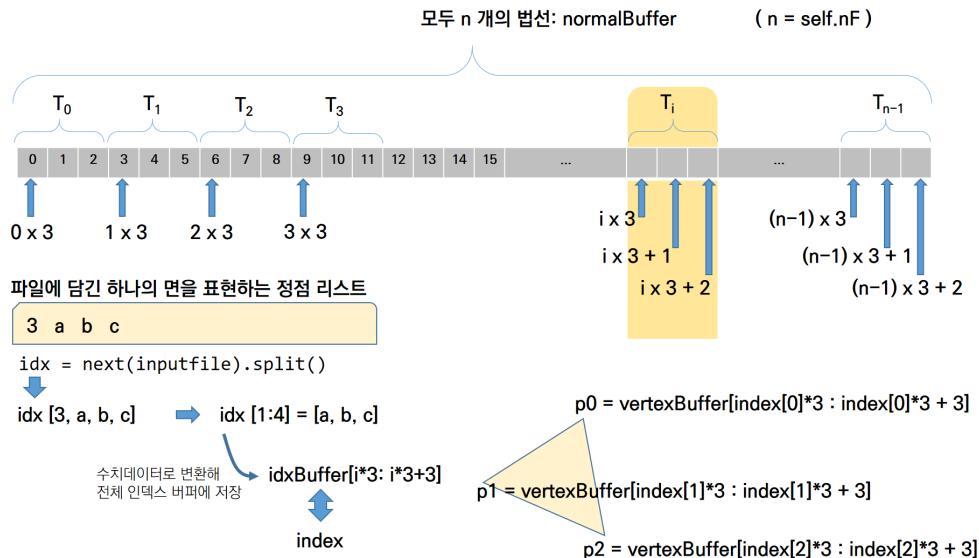
```
def loadData(self, filename):
    self.nF = int(next(inputfile))
    self.idxBuffer = np.zeros(shape=(self.nF*3, ), dtype=int)

    ### 법선벡터 저장을 위한 공간 준비
    self.normalBuffer = np.zeros(shape=(self.nF*3, ), dtype=float )
```

다음으로 각 면을 구성하는 정점 인덱스를 읽어 들이는 곳에서 이 인덱스를 이용하여 정점의 좌표를 확인하고, 이 좌표들을 이용하여 법선을 계산하는 작업을 수행한 뒤 해당 면에 해당하는 법선을 법선 벡터 버퍼에 저장하는 일을 하면 된다.

```
def loadData(self, filename):
    ...
    for i in range(self.nF):
        idx = next(inputfile).split()
        self.idxBuffer[i*3: i*3+3] = idx[1:4]
        index = self.idxBuffer[i*3: i*3+3]
        p0 = self.vertexBuffer[index[0]*3: index[0]*3 + 3]
        p1 = self.vertexBuffer[index[1]*3: index[1]*3 + 3]
        p2 = self.vertexBuffer[index[2]*3: index[2]*3 + 3]
        u = p1-p0
        v = p2-p0
        N = np.cross(u, v)
        norm = np.linalg.norm(N)
        N = N/norm
        self.normalBuffer[i*3: i*3+3] = N
```

그림 9는 이 코드 내에서 하나의 삼각형을 처리할 때 이 삼각형의 세 정점 좌표를 계산하는 방법을 코드를 설명하고 있다. 세 정점의 좌표를 알면 이 평면 위의 두 벡터를 쉽게 구할 수 있고, 이 두 벡터에 가우고을 적용하여 법선 벡터를 쉽게 구할 수 있다. 이렇게 해서 얻어진 법선 벡터는 normalBuffer의 i 번째 면을 위한 공간에 저장된다.

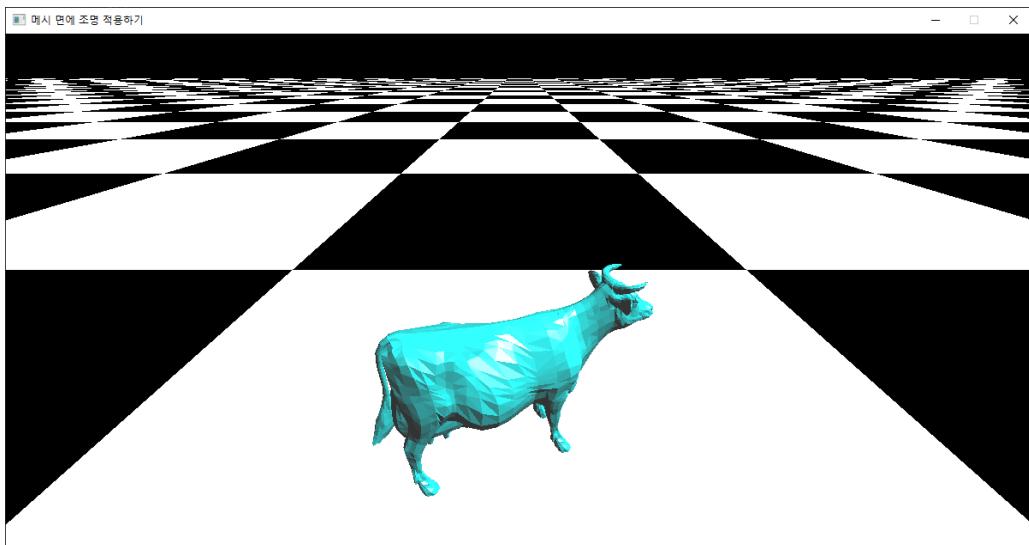


[그림 9] 한 삼각형 면을 구성하는 세 정점의 좌표의 값을 계산하는 방법

이를 그리는 방법을 생각해 보자. draw 메소드에서는 self.nF 회의 반복을 수행하면서 각 면을 그린다. 이때 저장해 놓은 normalBuffer에서 차례로 법선을 꺼내어 각 면에 한 번씩 적용하면 될 것이다. 이는 다음과 같이 구현할 수 있다.

```
def draw(self):
    glBegin(GL_TRIANGLES)
    for i in range(self.nF):
        verts = self.idxBuffer[i*3: i*3+3]
        glNormal3fv( self.normalBuffer[i*3: i*3+3] )
        glVertex3fv( self.vertexBuffer[verts[0]*3 : verts[0]*3 + 3] )
        glVertex3fv( self.vertexBuffer[verts[1]*3 : verts[1]*3 + 3] )
        glVertex3fv( self.vertexBuffer[verts[2]*3 : verts[2]*3 + 3] )
    glEnd()
```

이제 메시를 읽고 그림을 그리면 아래 그림 10과 같이 각 면이 조명에 따라 다른 음영을 가진 결과가 나타나게 될 것이다.

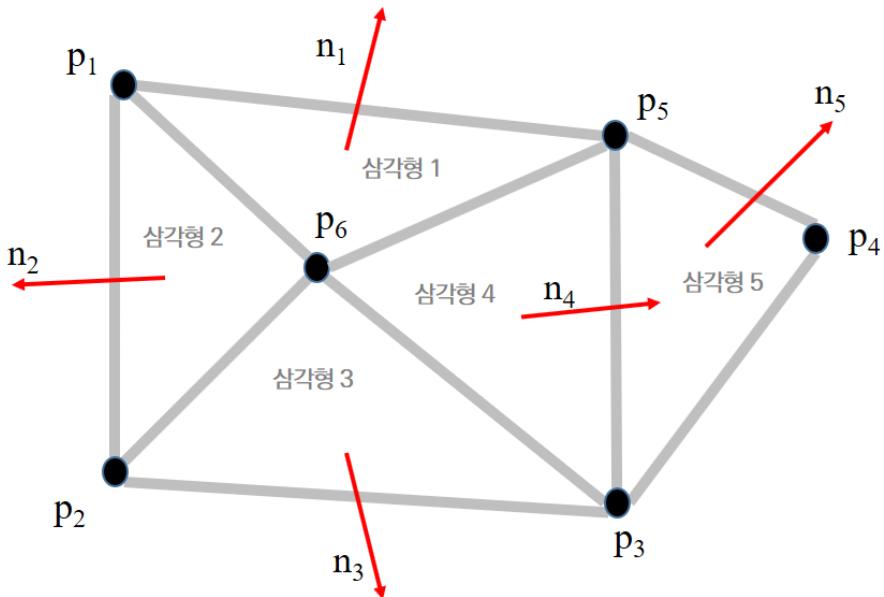


[그림 10] 면마다 법선을 계산하여 적용한 결과

면을 부드럽게 만들고 싶다면 면마다 적용되는 법선을 정점마다 적용되도록 변경하면 된다. 그런데 하나의 정점은 어떤 면에 사용되던지 하나의 법선을 유지해야 한다. 이를 위해서는 면을 읽어 들일 때 계산된 법선을 이 계산에 참여한 정점들에게 계속 누적해 가는 방식을 사용하면 된다. 그리고 최종적으로 하나의 정점에 대해 자신이 가진 법선 벡터를 정규화하는 과정을 수행하여 길이 1인 법선 벡터가 되도록 하는 것이다.

아래 그림 11을 통해 이를 이해해 보자. 그림 11에는 모두 6 개의 정점 $p_1, p_2, p_3, p_4, p_5, p_6$ 가 있다. 그리고 이 6 개의 정점이 5 개의 삼각형을 이루도록 메시가 구성되어 있다. 따라서 각각의 면에서 5 개의 법선 벡터 n_1, n_2, n_3, n_4, n_5 를 계산할 수 있다.

이 그림을 잘 보면 정점 p_4 는 삼각형 5에만 참여하고 있다 따라서 이 정점을 사용할 때는 법선 벡터로 n_5 를 사용하면 될 것이다. 그러면 p_1 은 어떤지 살펴보자. 이 정점은 삼각형 1과 삼각형 2에 참여한다. 두 삼각형의 법선 벡터는 n_1 과 n_2 이다. p_1 의 법선 벡터는 이 두 법선벡터를 고르게 반영하여 $(n_1 + n_2)/\|n_1 + n_2\|$ 를 사용하면 될 것이다. 정점 p_6 은 어떨까? 삼각형 1, 삼각형 2, 삼각형 3, 삼각형 4의 법선을 누적하면 될 것이다.



[그림 11] 정점별 법선을 계산하기 위해 고려할 상황의 예시

이를 구현하는 코드를 고려해 보자. 우선 normalBuffer는 이제 정점별 법선 벡터를 저장해야 하므로 그 공간의 크기는 면의 수가 아니라 정점의 수에 비례한다. 하나의 법선이 3 개의 원소를 가지므로 다음과 같이 준비할 수 있다. 이것은 정점 버퍼의 크기와 동일하다.

```
def loadData(self, filename):
    with open(filename, 'rt') as inputfile:
        self.nV = int(next(inputfile))

        self.vertexBuffer = np.zeros(shape = (self.nV*3, ),
                                     dtype=float)
    ### 정점 별 법선벡터 저장을 위한 공간 준비
    self.normalBuffer = np.zeros(shape = (self.nV*3, ),
                                 dtype=float)
```

각각의 면을 읽고 법선을 계산하는 것은 앞의 방식과 동일하다. 그런데 법선 벡터 버퍼에 저장할 때는 면 단위로 저장하는 것이 아니라 이 면의 구성에 참여한 모든 정점의 위치에 계산된 법선을 누적한다.

이러한 결과는 glShadeMode(GL_SMOOTH)를 이용하여 면의 내부를 칠하는 방식을 GL_SMOOTH로 지정하여 정점의 색상이 부드럽게 보간되도록 해야만 한다. 이것이 디폴트default 상태이다. 만약 glShadeMode(GL_FLAT)을 하게 되면 한 면의 색은 한 색으로 고정된다. 다음과 같이 코드를 바꾸어 보자. 그림 11과 같은 결과를 얻을 것이다. 이때 결정된 색상은 한 면을 그리기 직전에 마지막으로 설정된 색상이 된다. 이러한 누적이 완료되고 나면 정점의 개수만큼 반복하며 각 정점에 대응되는 법선 벡터를 정규화한다.

```
def loadData(self, filename):
    with open(filename, 'rt') as inputfile:
        self.nV = int(next(inputfile))

    ...

    for i in range(self.nF):
        idx = next(inputfile).split()
        self.idxBuffer[i*3: i*3+3] = idx[1:4]
        index = self.idxBuffer[i*3: i*3+3]
        p0 = self.vertexBuffer[index[0]*3: index[0]*3 + 3]
        p1 = self.vertexBuffer[index[1]*3: index[1]*3 + 3]
        p2 = self.vertexBuffer[index[2]*3: index[2]*3 + 3]
        u = p1-p0
        v = p2-p0
        N = np.cross(u, v)
        self.normalBuffer[index[0]*3: index[0]*3 + 3] += N
        self.normalBuffer[index[1]*3: index[1]*3 + 3] += N
        self.normalBuffer[index[2]*3: index[2]*3 + 3] += N

    for i in range(self.nV):
        N = self.normalBuffer[i*3: i*3 + 3]
        norm = np.linalg.norm(N)
        N = N/norm
        self.normalBuffer[i*3: i*3 + 3] = N
```

이제 이렇게 계산된 정점별 법선을 이용하여 그림을 그린다. 그리는 정점과 같은 위치에서 법선 벡터를 구해오면 된다.

```
def draw(self):
    glBegin(GL_TRIANGLES)
    for i in range(self.nF):
        verts = self.idxBuffer[i*3: i*3+3]
        glNormal3fv( self.normalBuffer[verts[0]*3 : verts[0]*3 + 3] )
```

```

glVertex3fv( self.vertexBuffer[verts[0]*3 : verts[0]*3 +3] )
glNormal3fv( self.normalBuffer[verts[1]*3 : verts[1]*3 +3] )
glVertex3fv( self.vertexBuffer[verts[1]*3 : verts[1]*3 +3] )
glNormal3fv( self.normalBuffer[verts[2]*3 : verts[2]*3 +3] )
glVertex3fv( self.vertexBuffer[verts[2]*3 : verts[2]*3 +3] )

glEnd()

```

지금까지의 코드를 정리하여 하나의 프로그램을 만들어 보자.

정점 별 법선 적용으로 부드러운 메시 그리기

```

from OpenGL.GL import *
from OpenGL.GLU import *
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtOpenGLWidgets import QOpenGLWidget
from PyQt6.QtCore import *

import math
import numpy as np

mat_spec = [1, 1, 1, 1]
mat_diff = [1.0, 0.7, 0.3, 1]
mat_ambi = [1, 1, 1, 1]
mat_shin = [16]

lit_spec = [1, 1, 1, 1]
lit_diff = [1, 1, 1, 1]
lit_ambi = [0, 0, 0, 1]

light_pos = [1, 1, 1, 0]

def LightSet():
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_spec)
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diff)
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambi)
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shin)

    glLightfv(GL_LIGHT0, GL_SPECULAR, lit_spec)
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lit_diff)
    glLightfv(GL_LIGHT0, GL_AMBIENT, lit_ambi)

    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)

```

```

def LightPositioning() :
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos)

def drawPlane():
    n, w = 100, 500
    # n: 체스판 한면의 정점수, w: 체스판 한면의 길이

    d = w / (n-1) # 인접한 두 정점 사이의 간격

    # 체스판 그리기
    glNormal3f(0, 1, 0)
    glBegin(GL_QUADS)
    for i in range(n):
        for j in range(n):
            if (i+j)%2 == 0:
                startX = -w/2 + i*d
                startZ = -w/2 + j*d
                glVertex3f(startX, 0, startZ)
                glVertex3f(startX, 0, startZ+d)
                glVertex3f(startX+d, 0, startZ+d)
                glVertex3f(startX+d, 0, startZ)
    glEnd()

class MeshLoader:
    def __init__(self):
        self.nV = 0 # 정점의 개수
        self.nF = 0 # 면의 개수
        self.vertexBuffer = None # 정점 버퍼
        self.idxBuffer = None # 면을 구성하는 정점 인덱스 버퍼
        self.list = None

    def loadData(self, filename):
        with open(filename, 'rt') as inputfile:
            self.nV = int(next(inputfile))

            self.vertexBuffer = np.zeros(shape = (self.nV*3, ), 
                                         dtype=float)
            ### 정점 별 법선벡터 저장을 위한 공간 준비
            self.normalBuffer = np.zeros(shape = (self.nV*3, ),
                                         dtype=float)

            for i in range(self.nV):
                verts = next(inputfile).split()
                self.vertexBuffer[i*3:i*3+3] = verts[0:3]

```

```

        coordMin = self.vertexBuffer.min()
        coordMax = self.vertexBuffer.max()
        scale = max([coordMin, coordMax], key=abs)
        self.vertexBuffer /= scale

        self.nF = int(next(inputfile))
        self.idxBuffer = np.zeros(shape=(self.nF*3, ), dtype=int)

        for i in range(self.nF):
            idx = next(inputfile).split()
            self.idxBuffer[i*3: i*3+3] = idx[1:4]
            index = self.idxBuffer[i*3: i*3+3]
            p0 = self.vertexBuffer[index[0]*3: index[0]*3 + 3]
            p1 = self.vertexBuffer[index[1]*3: index[1]*3 + 3]
            p2 = self.vertexBuffer[index[2]*3: index[2]*3 + 3]
            u = p1-p0
            v = p2-p0
            N = np.cross(u, v)
            self.normalBuffer[index[0]*3: index[0]*3 + 3] += N
            self.normalBuffer[index[1]*3: index[1]*3 + 3] += N
            self.normalBuffer[index[2]*3: index[2]*3 + 3] += N

        for i in range(self.nV):
            N = self.normalBuffer[i*3: i*3 + 3]
            norm = np.linalg.norm(N)
            N = N/norm
            self.normalBuffer[i*3: i*3 + 3] = N

    def draw(self):
        glBegin(GL_TRIANGLES)
        for i in range(self.nF):
            verts = self.idxBuffer[i*3: i*3+3]
            glNormal3fv( self.normalBuffer[verts[0]*3 : verts[0]*3 +3] )
            glVertex3fv( self.vertexBuffer[verts[0]*3 : verts[0]*3 +3] )
            glNormal3fv( self.normalBuffer[verts[1]*3 : verts[1]*3 +3] )
            glVertex3fv( self.vertexBuffer[verts[1]*3 : verts[1]*3 +3] )
            glNormal3fv( self.normalBuffer[verts[2]*3 : verts[2]*3 +3] )
            glVertex3fv( self.vertexBuffer[verts[2]*3 : verts[2]*3 +3] )
        glEnd()

    def make_displayList(self):
        self.list = glGenLists(1)
        glNewList(self.list, GL_COMPILE)

```

```

        self.draw()
        glEndList()

    def draw_list(self):
        glCallList(self.list)

class MyGLWidget(QOpenGLWidget):
    def __init__(self, parent=None):
        super().__init__(parent)

        self.angle = 0

    def initializeGL(self):
        self.meshLoader = MeshLoader() # 메시 로더 생성
        self.meshLoader.loadData('./cow.txt')
        self.meshLoader.make_displayList()

        glClearColor(0.0, 0.0, 0.0, 1.0)
        self.planeList = glGenLists(1)
        glNewList(self.planeList, GL_COMPILE)
        # 그리기 코드
        drawPlane()
        glEndList()

        glEnable(GL_DEPTH_TEST)
        LightSet()
        MeshLoader

    def resizeGL(self, width, height):
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.01, 100)

    def paintGL(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        gluLookAt(0,3,3, 0,2,0, 0,1,0)
        LightPositioning()

        glCallList(self.planeList)

        glTranslatef(0, 2, 0)
        glRotatef(self.angle, 0, 1, 0)
        self.angle += 5

```

```

glEnable(GL_LIGHTING)
self.meshLoader.draw_list()
glDisable(GL_LIGHTING)

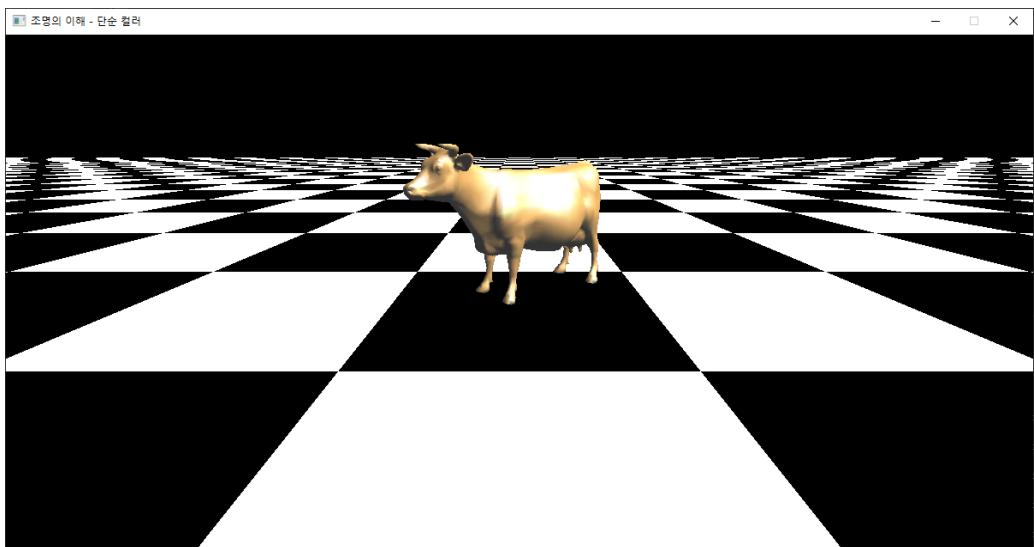
class MyWindow(QMainWindow):
    def __init__(self, title=''):
        QMainWindow.__init__(self)
        self.setWindowTitle(title)
        self.glWidget = MyGLWidget()
        self.setCentralWidget(self.glWidget)

    def keyPressEvent(self, e):
        self.glWidget.update()

def main(argv = []):
    app = QApplication(argv)
    window = MyWindow('조명의 이해 - 단순 컬러')
    window.setFixedSize(1200, 600)
    window.show()
    app.exec()

if __name__ == '__main__':
    main(sys.argv)

```



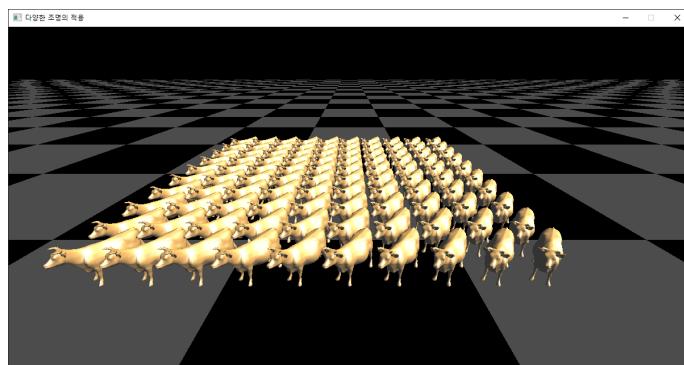
[그림 12] 부드러운 메시 그리기 완성

9.3 다양한 조명 적용

이제 다양한 조명 모델을 적용하여 보자. 지금까지 살펴본 광원은 “방향광원(directional light)”였다. 여기에는 광원의 위치가 동차좌표계로 표현되어 있고, 그 w 좌표가 0으로 설정되어 있다. 이것은 앞서 동차좌표계에서 다루었던 것처럼 3차원 공간의 어떤 점이 아니라 벡터를 의미하는 것이다. 따라서 확인할 수 있는 것처럼 모든 객체가 동일한 방향에서 오는 빛을 받아 음영이 그려지게 된다. 방향광원은 햇볕과 같은 빛을 표현하는데에 적합하다. 메시를 여러 개 그려서 이들이 동일한 방향에서 빛을 받는 것을 확인해 보자.

```
def paintGL(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    gluLookAt(0,4,9, 0,0,0, 0,1,0)
    LightPositioning()

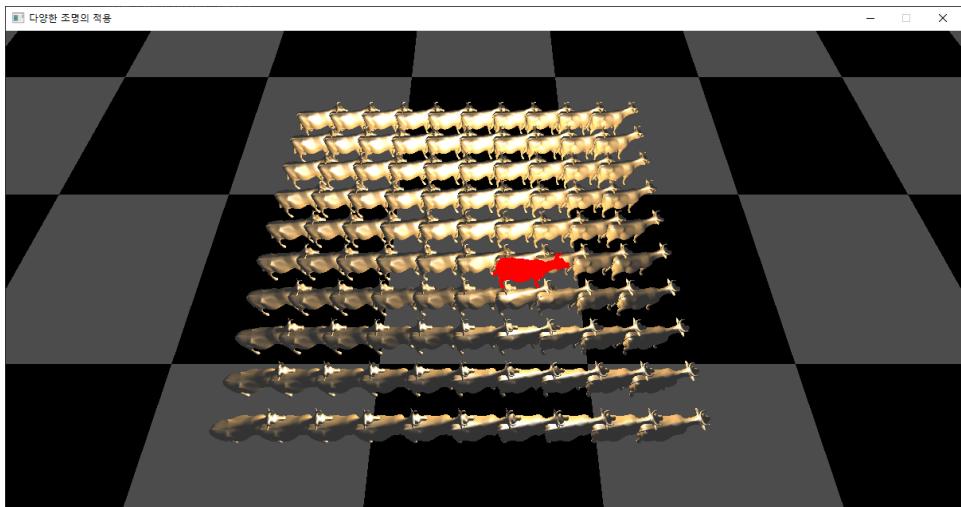
    glCallList(self.planeList)
    self.angle += 5
    glEnable(GL_LIGHTING)
    for i in range(10):
        for j in range(10):
            glPushMatrix()
            glTranslatef(i-5, 0, j-5)
            glRotatef(self.angle, 0, 1, 0)
            self.meshLoader.draw_list()
            glPopMatrix()
    glDisable(GL_LIGHTING)
```



[그림 13] 방향 광원이 적용된 예

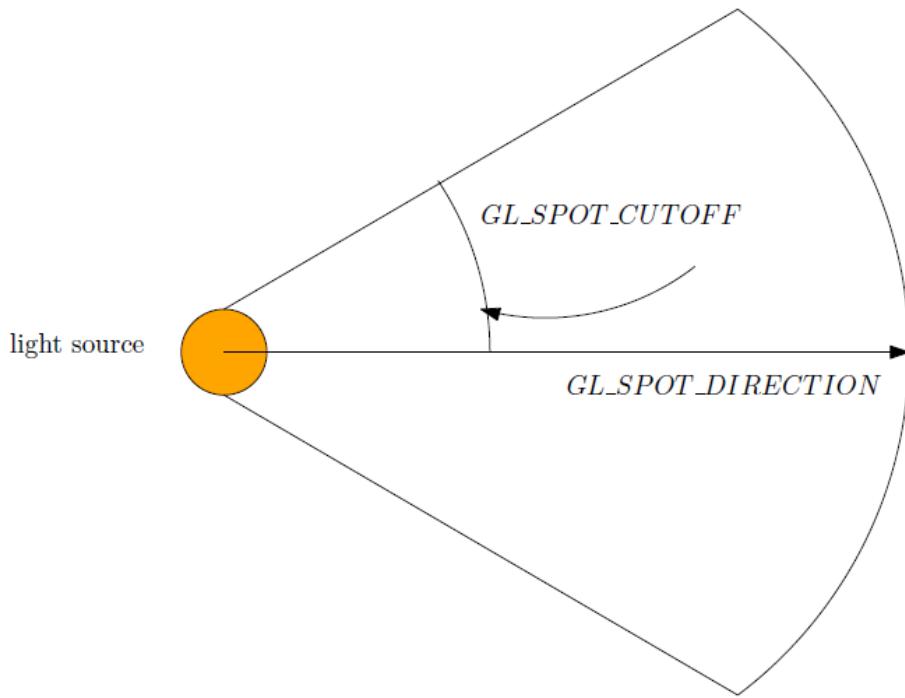
그런데 많은 경우 전구와 같이 하나의 점에서 빛이 모든 방향으로 퍼져 나가는 경우도 표현할 필요가 있다. 이렇게 한 점에서 모든 방향으로 빛이 퍼지는 광원을 점광원(point light)라고 한다. 점광원을 구현하는 방법은 광원의 좌표에서 w 성분을 0이 아닌 값, 일반적으로는 1을 설정하면 된다. 결과는 그림 14와 같이 특정한 지점에서 빛이 나는 것을 확인할 수 있다.

```
light_pos = [1, 1.5, 1, 1]
```



[그림 14] 점광원이 적용된 예

집중광원은 탐조등(探照燈)처럼 특정한 방향으로 빛을 집중하여 보내는 것이다. 이러한 집중광원은 점광원의 특징에 몇 가지 제약을 주면 된다. 그림 15에 나타난 있는 것처럼, 집중광원은 광원이 진행하는 방향을 나타내는 GL SPOT DIRECTION과, 빛이 퍼지는 범위의 한계를 설정하는 GL SPOT CUTOFF가 필요하다.



[그림 15] 집중 광원의 추가적인 속성

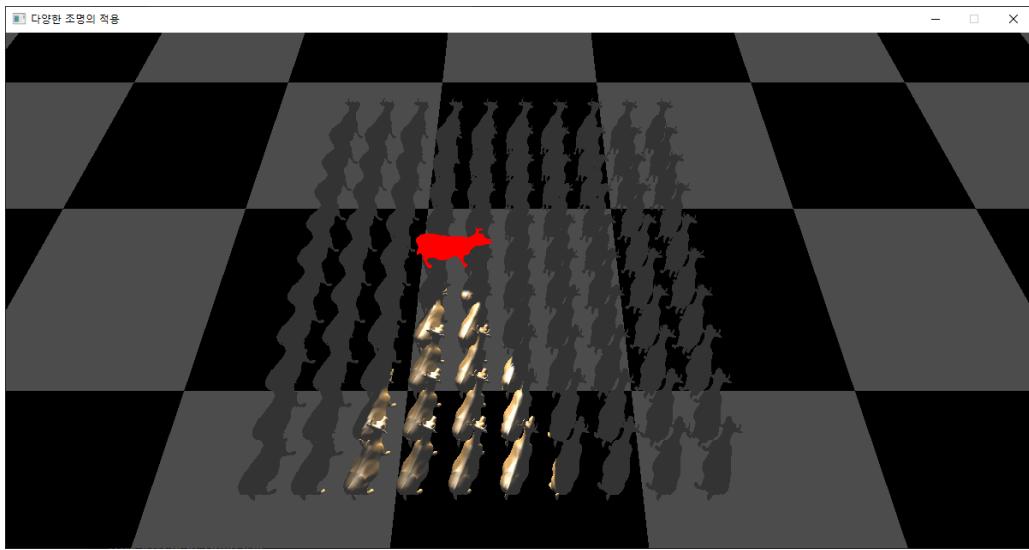
이러한 추가적인 속성은 glLightfv를 이용하여 적용한다.

```
def LightSet():
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_spec)
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diff)
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambi)
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shin)

    glLightfv(GL_LIGHT0, GL_SPECULAR, lit_spec)
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lit_diff)
    glLightfv(GL_LIGHT0, GL_AMBIENT, lit_ambi)
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 30.0 )
    glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, [0, -1, 0])

    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)
```

이러한 제약 조건을 추가한 뒤에 실행을 하면 그림 16과 같이 집중 광원이 적용된 결과를 확인할 수 있을 것이다.



[그림 15] 집중 광원이 적용된 결과