

Chapter 09 - 동명대학교 게임공학과 3D 그래픽스 프로그래밍

색과 조명



“소프트웨어 재사용 전에, 우선은 사용부터 가능해야 한다.”

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

– 랄프 존슨 Ralph Johnson

이 장에서 생각할 문제

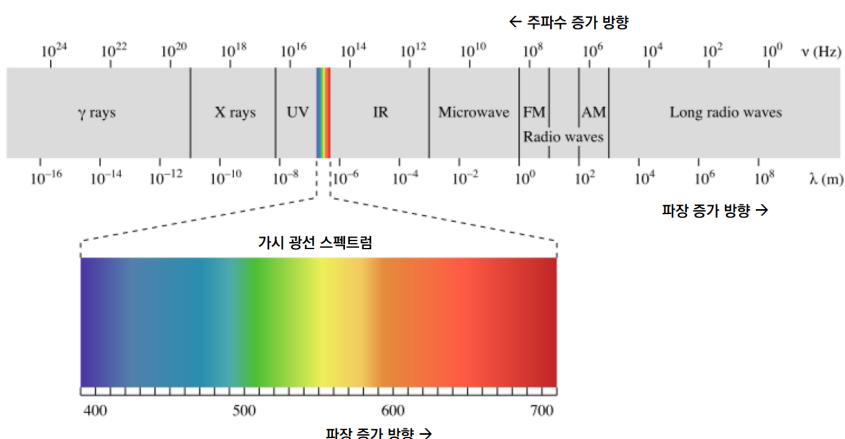
- ❖ 파일에 담길 기하 객체 정보 읽기
- ❖ 기하 객체를 효율적으로 다룰 수 있는 기술
- ❖ 메시 데이터의 가시화를 빠르게 하는 방법

9.1 색에 대한 이해

컴퓨터 그래픽스의 결과는 영상^{image}로 표현된다. 영상은 하나 이상의 색으로 표현된다. 색이라는 것은 빛이라는 전자기파를 우리 뇌가 인지한 결과이다. 이러한 인지 기능을 시각^{vision}이라고 한다. 우리가 주변 환경을 이해하고 해석할 때 일반적으로 가장 많이 의존하는 정보가 시각 정보이다. 이러한 이유로 우리는 시각 정보를 구성하는 색을 구분하는 데에 매우 익숙하다. 따라서 제대로 된 색상 표현을 하지 못하면 사실적인 컴퓨터 그래픽스 결과를 만들 수 없다.

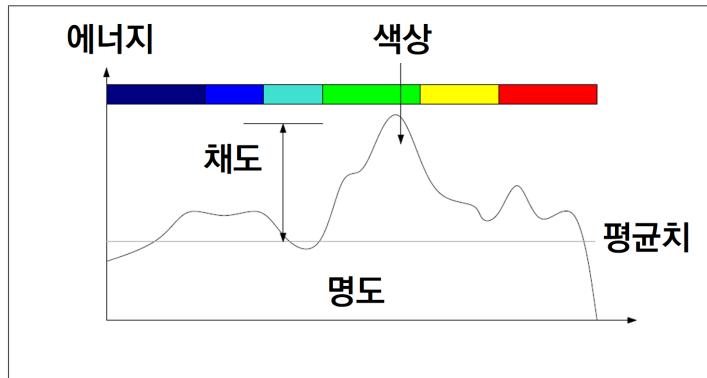
색이라는 것은 눈으로 감지할 수 있는 전자기파의 각 주파수 영역에 붙은 이름과 같은 것이다. 사람이 감지할 수 있는 전자기파를 가시광선이라고 한다. 사람에 따라 이 감지할 수 있는 빛의 범위가 다르지만, 일반적으로 390 nm에서 720 nm의 범위를 가시광성이라고 한다. 눈으로 볼수 있는 가시 광선이 모두 제공될 때 우리는 이 신호를 흰색으로 인식한다. 이 백색광을 프리즘prism을 이용하여 분리한 사람이 바로 아이작 뉴턴^{Isaac Newton}이다. 그는 이 프리즘 실험을 통해 빛이라는 것이 하나의 동일한 성질을 갖는 것이 아니라 서로 다른 성질의 여러 빛이 섞여 있는 것임을 보여 주었다.

그림 1은 가시광선의 범위를 보이고 있다. 그리고 이 가시광선을 구성하는 다양한 빛들의 색상을 함께 표시하고 있다. 프리즘을 통과한 빛은 파장에 따라 진행방향이 나뉘게 되고, 각각의 파장이 가진 고유한 색을 관찰할 수 있다. 우리가 보게되는 색상은 이렇게 나뉘어진 각 파장의 빛 가운데 일부를 선택하여 가중치를 달리 주어 합성한 것이라고 생각할 수 있다.



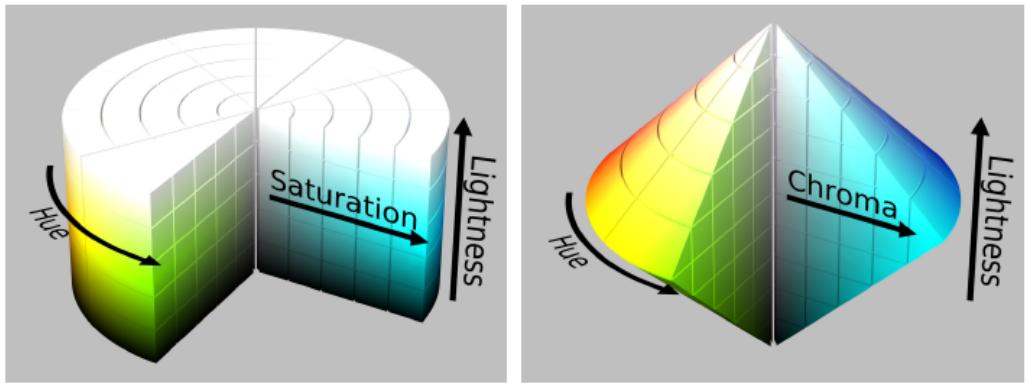
[그림 1] 여러 전자기파의 범위와 가시광선의 스펙트럼

색을 이야기할 때, 우리는 색상^{hue}, 채도^{saturation}, 그리고 명도^{lightness}라는 용어를 종종 쓴다. 색상은 우세 주파수의 색이며, 명도는 색의 밝기를 의미하며, 가시광선 스펙트럼^{spectrum}에서 각 주파수 별 에너지의 총합, 즉 주파수 별 강도 곡선을 적분한 값이 된다. 또한 채도는 색상의 선명한 정도를 의미하는데, 우세 주파수 영역의 에너지 값에서 평균 에너지를 뺀 값으로 결정된다. 이를 그림으로 표현하면 그림 2와 같다.

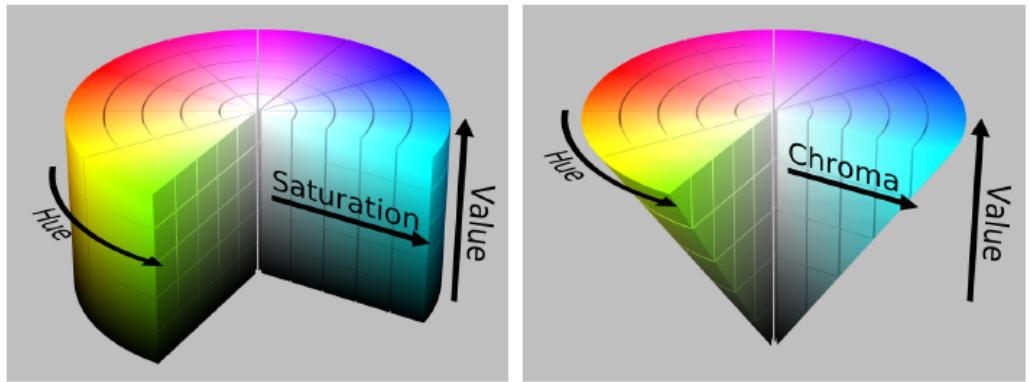


[그림 2] 색상, 채도, 명도의 이해

HSL과 HSV 색 공간 모형은 모두 원기둥 공간에서 색을 표현하는 모델들이다. HSL은 색상^{hue}, 채도^{saturation}, 그리고 명도^{lightness}로 색을 표현하는 것이고, HSV 모델은 색상, 채도와 함께 그리고 명도^{Value}로 표현하는 모형으로 원기둥으로 표현할 수 있는 공간이다. HSL의 ‘L’과 HSV의 ‘V’가 모두 밝은 정도를 나타내는데, 그 의미는 약간의 차이가 있다. HSL의 ‘L’과 HSV의 ‘V’ 모두 가장 낮은 값일 때는 검정색을 표현한다는 점에서는 같다. 하지만 ‘L’이 가장 큰 값에 이르면 색이 흰색이 되는데 반해, ‘V’가 최대치에 이르면 가장 채도가 높은 선명한 색이 된다. 이 두 색공간에서 색상은 각도로 표현되며, 채도는 반지름, 명도는 높이로 표현된다. 예를 들어 남색은 HSV 모델에서 240도의 색상, 100%의 채도, 50%의 명도로 표현할 수 있어 (240, 100, 50)으로 표현된다. 명도가 0에 가까워지면 사실상 어떤 색상과 채도를 가져도 검은색에 수렴하므로, 원기둥 모델을 수정하여 원뿔로 표현하는 모델도 있다[Wik13c]. HSL 모델에서는 ‘L’이 최대치에 이르면 역시 모든 색이 흰색에 수렴하기 때문에 원뿔의 꼭지점처럼 모이게 된다. 그림 7.3은 HSL 모델이 표현하는 색 공간을 보이고 있다. ‘L’ 값의 크기에 따라 한점에서 커졌다가 다시 한점으로 모이는 모습을 보이고 있다. 또한 그림 7.4는 HSV 모델이 표현하는 색 공간을 보이고 있다. 이 모델에서는 ‘V’ 값이 0일 때 하나의 색(검정)으로 모이기 때문에 원뿔 모양으로 표현할 수도 있다.



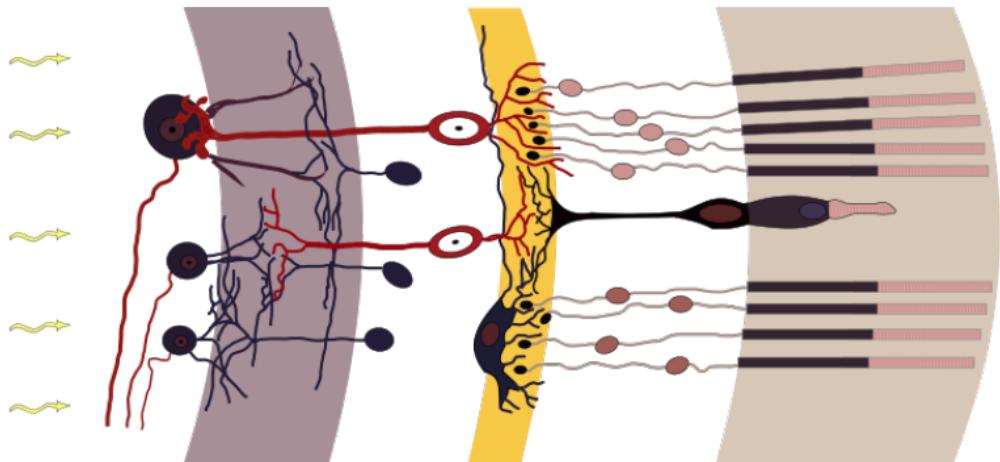
(a) HSL 모델의 색 기둥



(b) HSV 모델의 색기둥

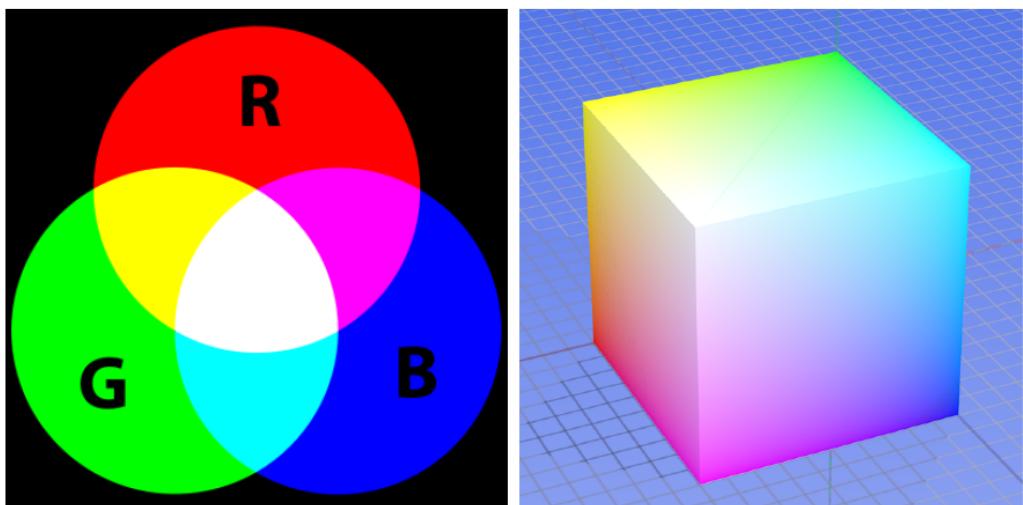
[그림 3] HSL 모델과 HSV 모델의 비교

RGB 색 모델은 삼중자극(三重刺戟, tristimulus) 이론에 근거하여 하나의 색을 세 종류의 색의 합성으로 보는 것이다. 요소가 되는 세 색은 빨강, 녹색, 파랑이며 이 색들의 이름에서 각각 R, G, B를 따온 것이다. 이것은 색을 인식하는 원추 세포가 이 세 가지 색상에 가장 민감하게 반응한다는 사실에 근거를 두고 있다. 눈은 각막(角膜, cornea)을 통해 들어온 빛이 렌즈의 역할을 하는 수정체^{lens}를 거친 뒤에 망막^{retina}에 상을 맺게 한다. 망막에는 명암을 인식하는 막대^{rod} 세포와 색상을 인지하는 원추^{cone} 세포가 그림 4처럼 존재한다. 이 원추 세포가 빨강, 녹색, 파랑에 민감하게 반응하며, 우리가 인지하는 색은 이 세 가지 색이 얼마나 많이 섞여 있나에 의해 결정된다는 것이다.



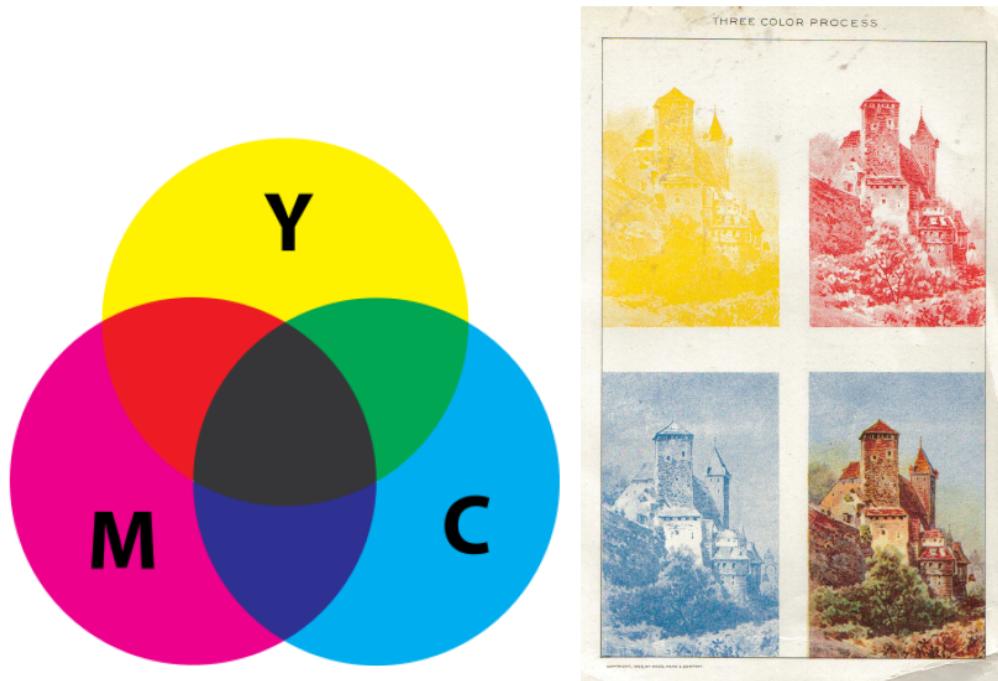
[그림 4] 망막 뒤에 존재하는 원추 세포와 막대 세포

그림 5는 이러한 RGB 색 공간 모델의 기본 요소가 되는 색상과 이 색상들의 합성을 보이고 있으며, 이 모델로 표현할 수 있는 색상 공간은 오른쪽에 나타나 있다. RGB 모델은 색이 섞일 때에 명도가 높아지는 가산혼합의 특성을 갖는다. 따라서 각 요소 색이 가장 높은 명도로 모두 다 섞일 경우 하얗을 얻게된다. 이러한 모델로 표현할 수 있는 색 공간은 세 가지 요소를 축으로 하는 3차원 공간의 단위 입방체^{cube}로 표현할 수 있다. 다양한 분야에서 서로 다른 색 모델을 사용하는데, OpenGL이나 DirectX와 같은 실시간 그래픽스 라이브러리에서는 이 RGB 모델을 사용한다.



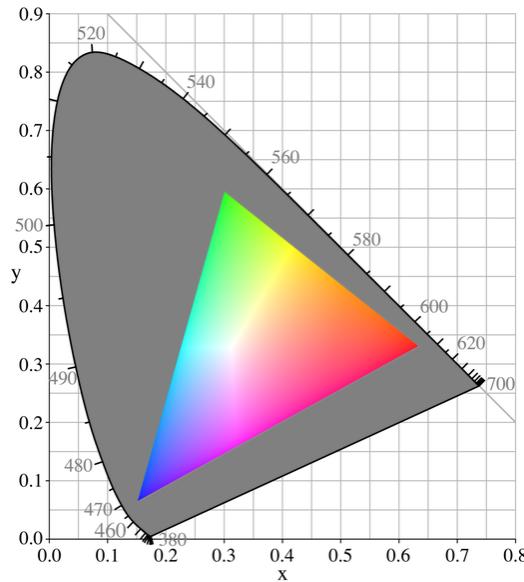
[그림 5] RGB 색상 공간

RGB 모델은 두 가지 요소 색이 합해졌을 때에 명도가 높아지는 가산혼합이다. 그런데 CMYK 모델은 그림 6과 같이 요소 색을 합쳤을 때 명도가 낮아지는 감산혼합 모델이다. 기본 색은 시안(cyan), 마젠타(magenta), 노랑(yellow), 검정(black)이며, 각각의 머리글자를 따면서 검정은 K로 표현해 CMYK 모델로 불린다. 이 모델로는 하양(white)을 표현할 수 없다. 오프셋(offset) 인쇄에 사용되는 모델이며, 출판용 소프트웨어에서 이 색 공간 모델을 지원한다. RGB나 HSV 모델에 비해 표현할 수 있는 색의 범위가 좁다. 그림 7.8의 오른쪽은 시안, 마젠타, 노랑의 세 가지 색으로 분할한 예를 보이고 있다. 이 경우에는 검정 잉크를 사용하지 않았는데, 검정 잉크를 사용할 경우 전체적으로 잉크의 양을 줄일 수 있다.



[그림 6] CMY 색 혼합의 특성과 색상 분할 예시

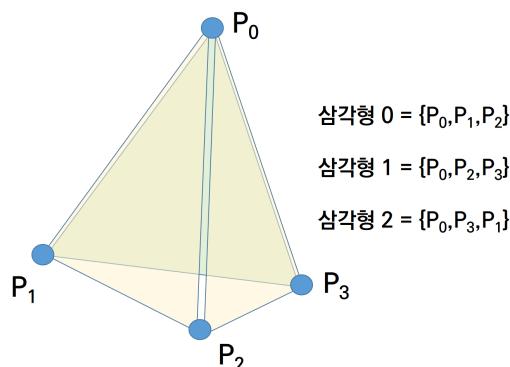
색 모델에 따라서 표현할 수 있는 색의 범위가 제한된다. 이를 색의 범위, 혹은 색역^{color gamut}이라고 부른다. 색역의 표현은 국제조명위원회^{CIE}에서 정한 CIE 1931 색공간으로 표현하는 것이 일반적이다. 그림 7은 색공간 내에서 CRT 모니터가 갖는 색역을 보이고 있다.



[그림 7] CRT 모니터의 색역

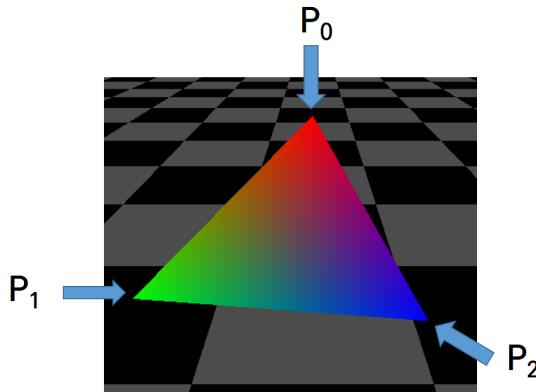
9.2 단순한 색상 지정

OpenGL에서 그리려는 기하 객체의 색상을 지정하는 가장 단순한 방법은 각각의 정점에 대해 색상을 지정하는 것이다. 다음 그림 8과 같이 네 개의 정점 P_0, P_1, P_2, P_3 을 이용하여 세 개의 삼각형을 그릴 수 있을 것이다. 이때 이 삼각형의 색깔을 어떻게 설정하는지 생각해 보다.



[그림 8] 삼각형 3 개로 이루어진 객체의 예시

여기에서 P_0 , P_1 , P_2 로 그려지는 삼각형에 색상을 지정해 보자. 이때 각 정점의 색을 설정하는 것은 `glColor3f`를 이용하여 지정하면 될 것이다. P_0 는 빨강, P_1 은 녹색, P_2 에는 파랑을 지정했을 때, 그림 9와 같이 이들 정점으로 만들어지는 삼각형의 내부는 색상이 부드럽게 보간^{interpolation}되어 나타난다.



[그림 9] 각 정점에 다른 색상을 부여했을 때에 그려지는 모습

이를 그리는 방법을 생각해 보자. 아래 코드처럼 4 개의 정점 좌표를 준비했다. 그리고 삼각형을 그리면서 이 중 세 개의 정점을 사용한다. 각 정점을 그릴 때마다 원하는 색상을 지정할 수 있다.

```

P0 = [0,5,0]
P1 = [-3,1,3]
P2 = [ 3,1,3]
P3 = [ 0,1,-3]

glBegin(GL_TRIANGLES)
glColor3f(1, 0, 0)
glVertex3fv(P0)
glColor3f(0, 1, 0)
glVertex3fv(P1)
glColor3f(0, 0, 1)
glVertex3fv(P2)
glEnd()

```

정점이 하나의 색상만 가질 수 있는 것은 아니다. 다음 코드와 같이 정점 네 개를 이용하여 세 개의 삼각형을 그린다고 하자. P_2 정점은 첫 삼각형을 그릴 때는 파랑이 지정되었지만,

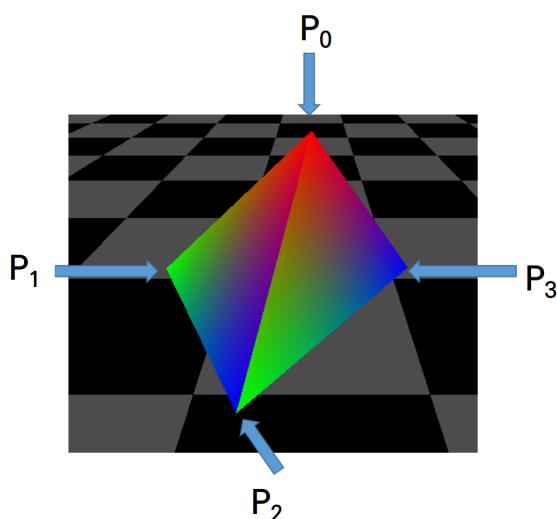
두 번째 삼각형을 그리기 위해 사용될 때에는 녹색으로 지정되었다. 따라서 이 정점을 기준으로 첫번째 삼각형 쪽은 파랑색이, 두번째 삼각형 쪽으로는 녹색의 색상이 퍼져나가는 모습이 된다.

```
P0 = [0,5,0]
P1 = [-3,1,3]
P2 = [ 3,1,3]
P3 = [ 0,1,-3]

glRotatef(self.angle, 0, 1, 0)
self.angle += 1
glBegin(GL_TRIANGLES)
glColor3f(1, 0, 0)
glVertex3fv(P0)
glColor3f(0, 1, 0)
glVertex3fv(P1)
glColor3f(0, 0, 1)
glVertex3fv(P2)

glColor3f(1, 0, 0)
glVertex3fv(P0)
glColor3f(0, 1, 0)
glVertex3fv(P2)
glColor3f(0, 0, 1)
glVertex3fv(P3)

...
glEnd()
```

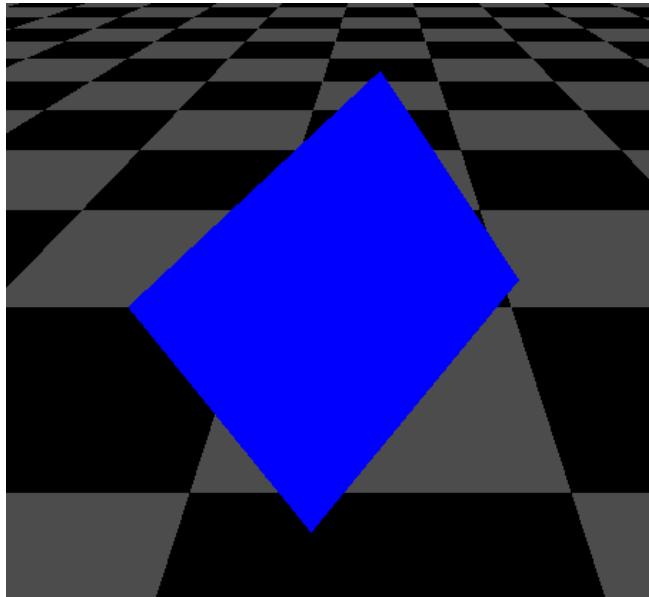


[그림 10] 하나의 정점에 대해 사용될 때마다 다른 색을 지정한 결과

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.

```
glShadeModel(GL_FLAT)
```

```
glBegin(GL_TRIANGLES)
	glColor3f(1, 0, 0)
	glVertex3fv(P0)
	glColor3f(0, 1, 0)
	glVertex3fv(P1)
	glColor3f(0, 0, 1)
	glVertex3fv(P2)
```



[그림 11] glShadeMode(GL_FLAT)을 이용하여 지정된 마지막 색상으로 면 그리기

이제 지금까지의 코드를 정리하여 하나의 프로그램을 만들어 보자. 평면 위에 세 개의 삼각형을 그리고, 각 면마다 세 정점에 각각 다른 색을 지정한다. 그리고 키보드를 이용하여 이를 회전시킬 수 있도록 하자. 어떤 키 값이 입력되든지 이 객체에 적용되는 회전 각도를 증가시켜 키보드를 누르고 있으면 계속해서 회전할 것이다.

객체의 색상을 지정하고 회전시켜 보기

```
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

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
    glColor3f(0.3,0.3,0.3)
    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 MyGLWidget(QOpenGLWidget):
    def __init__(self, parent=None):
        super().__init__(parent)

        self.angle = 0

    def initializeGL(self):
        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)

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(0,7,10, 0,2,0, 0,1,0)

    glCallList(self.planeList)

    P0 = [0,5,0]
    P1 = [-3,1,3]
    P2 = [ 3,1,3]
    P3 = [ 0,1,-3]

    glRotatef(self.angle, 0, 1, 0)
    self.angle += 1

    glBegin(GL_TRIANGLES)
    glColor3f(1, 0, 0)
    glVertex3fv(P0)
    glColor3f(0, 1, 0)
    glVertex3fv(P1)
    glColor3f(0, 0, 1)
    glVertex3fv(P2)

    glColor3f(1, 0, 0)
    glVertex3fv(P0)
    glColor3f(0, 1, 0)
    glVertex3fv(P2)
    glColor3f(0, 0, 1)
    glVertex3fv(P3)

    glColor3f(1, 0, 0)
    glVertex3fv(P0)
    glColor3f(0, 1, 0)
    glVertex3fv(P3)
    glColor3f(0, 0, 1)
    glVertex3fv(P1)
    glEnd()
```

```

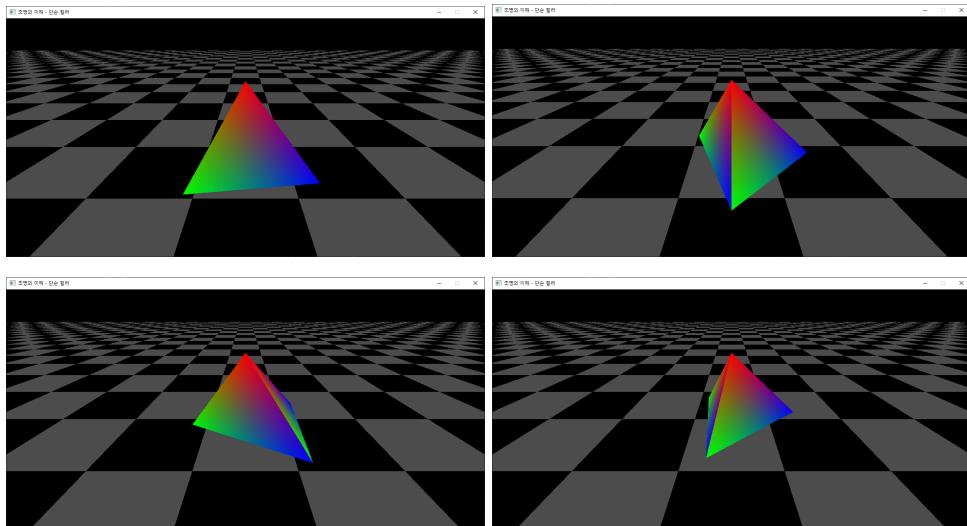
class MyWindow(QMainWindow):
    def __init__(self, title=''):
        QMainWindow.__init__(self)
        self.setWindowTitle(title)
        self.glWidget = MyGLWidget()
        self.setCentralWidget(self.glWidget)

    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] 입체 물체를 그리고 색상을 지정하여 회전시키기

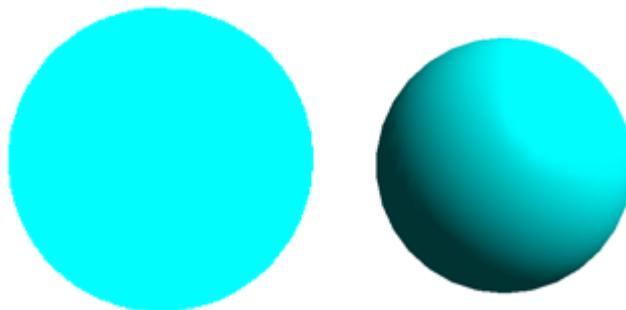
이 모델의 단점은 무엇일까? 실제 세계에 존재하는 입체 물체는 빛을 받아 반사시켜서 자신의 색을 드러낸다. 그리고 위의 예와 같이 회전 등을 통해 조명을 받는 정도가 바뀌면 그 색이 밝아지거나 어두워지는 방식으로 색상이 바뀌어야 한다. 그런데 우리가 사용한

방식은 객체의 색이 전혀 바뀌지 않는 방식이다. 이제 빛을 내는 광원^{light source}가 있고, 객체는 이 광원에서 오는 빛을 받아 반사하는 모습을 갖도록 만들어 보자.

9.3 조명 모델

컴퓨터 그래픽스의 최종 결과는 시각 정보이다. 시각 정보는 우리의 눈을 통해 입력된다. 눈이 인지하는 신호는 전자기파이며, 우리가 인지할 수 있는 영역의 전자기파를 일상적으로 ‘빛’이라 부른다. 빛은 시각정보를 생성하는 데에 있어 필수적인 요소이다. 같은 빛이라고 모든 물체가 같은 색으로 보이지는 않는다. 물체는 저마다의 재질^{material} 특성에 따라 서로 다른 방식으로 빛을 반사하여 다른 색상을 갖게 된다.

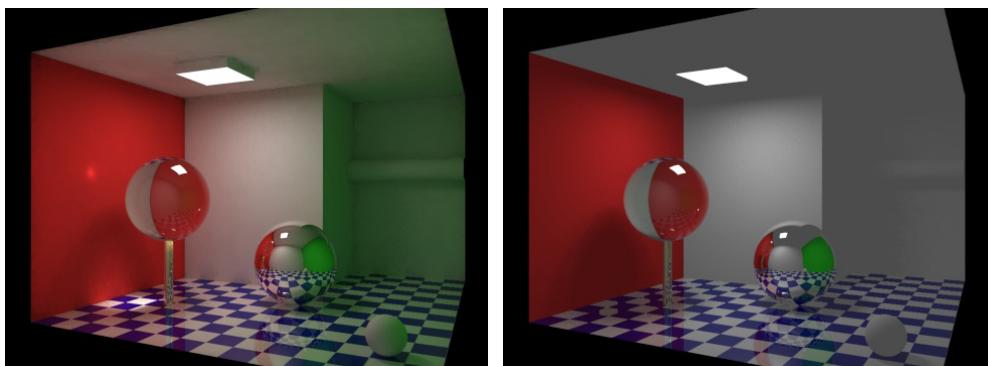
음영(陰影) 계산, 혹은 셰이딩(shading)은 어떤 물체의 표면에서 어두운 부분과 밝은 부분을 서로 다른 밝기로 그려내는 것이다. 어떤 구를 그렸는데 그림 13의 왼쪽과 같이 모든 면이 동일한 색으로 그리면 입체감이 없다. 구는 그림 13의 오른쪽처럼 그려져야 한다.



[그림 13] 입체감이 없는 구와 입체감이 있는 구의 예

그림 13의 왼쪽 그림은 표면의 모든 지점이 동일한 색을 가지고 있는 것이고, 오른쪽 그림은 서로 다른 색을 가지고 있다. 색은 앞서 살펴본 것과 같이 눈에 의해 감지되는 전자기파, 즉 빛이다. 따라서 각각의 지점은 서로 다른 빛을 우리 눈에 보내고 있는 것이다. 구는 동일한 재질로 이루어졌지만, 조명과 물체의 상호작용이 동일한 재질의 구면 각각에서 서로 다른 빛을 눈에 보내도록 하는 것이다. 이러한 상호작용에서 고려해야 하는 것들은 광원, 재질의 속성, 관찰자의 위치, 표면의 방향 등이 있다.

색은 곧 감지된 빛이다. 빛은 광원에서 나온다. 광원을 출발한 빛이 물체에 닿으면 일부는 반사되고 일부는 흡수된다. 반사된 빛은 또 다시 다른 물체에 닿아 반사, 흡수를 반복한다. 이러한 과정은 무한히 반복되며, 눈에 도달한 빛은 그쪽 방향으로의 색을 결정하게 된다. 이러한 빛의 반사를 흉내내어 장면(scene) 안에 존재하는 모든 물체를 고려하여 빛의 반사를 계산하여 각 지점의 밝기와 색상을 계산하는 방식을 전역조명(global illumination)이라고 한다. 이 방법은 매우 사실적인 렌더링 결과를 얻을 수 있지만 계산량이 너무 커서 실시간 렌더링 환경에는 적합하지 않다. 어떤 빛이 물체에 닿아 반사된 빛까지 새로이 광원의 역할을 수행하는 것이 전역조명이라고 하면, 지역조명(local illumination) 모델은 렌더링하려는 물체 외에는 어떤 다른 객체도 고려하지 않고 빛과 렌더링하는 물체 둘만의 관계를 이용하여 음영을 결정하는 방식이다. 지역조명 모델은 물체 상호간의 빛 가림 등을 고려하지 않기 때문에 물체의 그림자 영역에 다른 물체가 들어가도 그림자가 지지 않고 밝게 렌더링 된다. 이러한 이유로 원래의 지역조명은 그림자 등이 생길 수가 없다. 하지만 전역조명과 같이 사실적인 느낌을 주기 위해 빛의 굴절, 반사, 그림자 등을 텍스처 등을 활용하여 그려 넣을 수는 있다. 지역조명 모델에서 이러한 효과를 흉내낸다 하더라도 다른 객체에 반사된 빛에 의한 효과는 무시하기 전역조명에 비해서는 사실감이 떨어진다. 사실성을 포기한 대신에 지역조명은 매우 빠른 계산이 가능하다. 따라서 게임이나 가상현실과 같은 실시간 응용에서는 각각의 표면이 조명과 어떤 관계에 놓여있는지만 고려하는 지역조명(local illumination)을 일반적으로 사용한다. 그림 14의 왼쪽은 전역조명 기법을 이용하여 장면을 렌더링 한 결과이다. 이와 달리 그림 14의 오른쪽은 지역조명을 이용하여 음영을 결정하고, 텍스처 기술등을 활용하여 최대한 이 장면을 비슷하게 만들어 본 결과이다. 붉은 색 벽이 다른 벽에 붉은 빛을 보내는 컬러 블리딩color bleeding 등이 나타나지 않고, 전등의 빛이 유리 구를 통하여 굴절되어 바닥에 비치는 효과 등이 나타나지 않는다.



[그림 14] 전역 조명 모델과 지역 조명 모델에 의한 렌더링 결과의 비교

광원 모델의 이해

일반적인 광원^{light source}은 다루기가 쉽지 않다. 하나의 광원은 일정한 면적이나 체적을 가지기 때문에 이 광원에서 나온 빛들을 적분해야 한다. 실제 실시간 렌더링에서는 광원을 하나의 점이나 방향으로 보는 단순화된 모델을 사용한다. 사용되는 광원은 다음과 같은 것들이 있다.

종류	특징
점 광원	광원의 위치와 색으로 결정. 전방향(omni-directional)으로 빛 진행
집중 광원	점광원과 동일하나 빛의 진행을 일부 입체각으로 빛을 제한
방향 광원	특정한 위치가 아니라 방향에 광원 존재
주변 광원	모든 곳에 동일하게 가해지는 빛

9.4 풍^{Phong} 지역조명 모델 이해하기

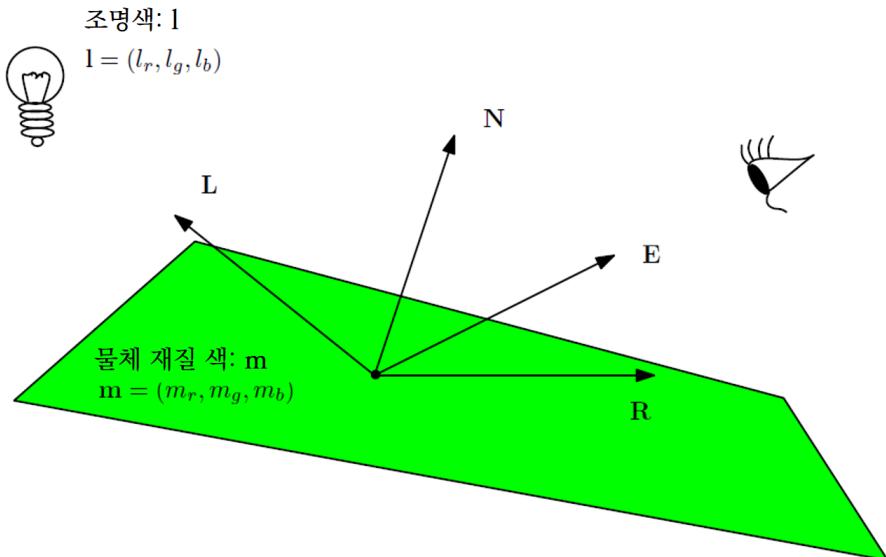
부드러운 면은 거울면과 같이 입사된 빛을 특정 반사 방향으로 집중적으로 보낸다. 이를 정반사^{specular reflection}라 한다. 반면 거친 면은 미세면을 관찰하면 다양한 방향의 조각면들로 구성되어 있어 한쪽 방향이 아니라 반구형으로 여러 방향에 골고루 빛을 보낼 수 있다. 이러한 반사를 난반사(diffuse reflection)이라고 한다. 실제 반사는 이러한 표면의 특성에 따라 매우 다양한 반사모델이 가능한데, 단순화된 모델은 물체 표면의 반사를 이 두 반사의 조합으로 보고 각각 모델링하는 것이다. 대표적인 모델이 바로 풍(Phong) 모델이다. 이 반사 모델은 OpenGL이나 DirectX에서 기본적으로 사용되는 모델이다. 풍 모델은 물체의 정반사와 난반사 특성을 모두 표현할 수 있는 간단하고 빠른 모델이다. 이 모델에서는 다음과 같은 세 종류의 반사 요소가 있다.

- 난반사 (diffuse reflection)
- 정반사 (specular reflection)
- 주변광 반사 (ambient reflection)

이 반사의 정도를 계산하기 위해서는 다음과 같은 정보가 필요하다.

- 광원으로 향하는 벡터 **L**
- 카메라(시점)을 향하는 벡터 **E**
- 법선 벡터 **N**
- 이상적인 반사 방향 **R**

이러한 정보가 가지는 의미가 그림 15에 나타나 있다.



[그림 15] 풍 모델에서 조명에 따른 관찰 색상을 결정하기 위해 필요한 정보들

풍 모델은 난반사, 정반사, 주변광을 각각 독립적으로 계산하여 합성한다. 각각의 반사 요소에 따라 결정해야 하는 것은 눈을 향해 오는 빛의 강도 ^{intensity}와 색상이다. 어떤 광원에서 나오는 빛의 색상이 $\mathbf{l} = (l_r, l_g, l_b)$ 라고 하고, 물체의 재질 색상이 $\mathbf{m} = (m_r, m_g, m_b)$ 라고 하자.

빛의 색상은 이 빛이 눈에 감지되었을 때 우리가 느끼는 색이라고 할 수 있다. 그러면 물체의 재질 색상은 무엇일까? 예를 들어 빨간 색을 가진 물체의 재질 색상 $(1.0, 0.0, 0.0)$ 은 무슨 의미일까? 이것은 도착하는 빛의 RGB 3 개 채널에서 R 채널의 빛을 100% 반사한다는 것을 의미한다. 따라서 반사되는 빛의 색상은 다음과 같다.

$$\mathbf{c} = (l_r \cdot m_r, l_g \cdot m_g, l_b \cdot m_b)$$

빛의 색을 표현하는 3차원 벡터를 **l**이라고 하고 물체의 재질 색을 3차원 벡터 **m**이라고 하면 이 색상 값 c 는 벡터 각 차원별 성분의 곱을 의미하는 연산자 \otimes 를 사용하여 간단히 다음과 같이 표현한다.

$$\mathbf{c} = \mathbf{l} \otimes \mathbf{m} = (l_r \cdot m_r, l_g \cdot m_g, l_b \cdot m_b)$$

이 계산은 눈에 감지되는 빛의 색상을 결정한다. 같은 색상이라도 밝고 어두운 곳이 있는데, 이러한 음영을 고려해야 입체적인 물체로 보이게 된다. 이러한 음영은 광강도 $I^{\text{light intensity}}$ 에 의해 결정되는데, 이 광강도의 값은 스칼라 I^{scalar} 값 I 로 표현한다. 그러면 실제로 눈에 보이는 색은 광원과 재질에 의해 결정되는 색상 c 와 광강도 I 에 의해 다음과 같이 결정된다.

$$\kappa = I\mathbf{c}$$

앞서 설명한 바와 같이 풍 모델은 각각의 반사 요소를 모두 따로 계산한다. 난반사 요소와 관련된 값에는 아래첨자 d 를 붙이고, 정반사에는 s , 주변광 반사에는 a 를 붙이면 눈에 관측되는 색상은 다음과 같다.

$$\kappa = I_a \mathbf{c}_a + I_d \mathbf{c}_d + I_s \mathbf{c}_s$$

이때, 주변광의 광강도는 상수 값을 가지므로 I_a 는 1로 설정할 수 있다. 따라서 위 식은 다음과 같이 바꿀 수 있다.

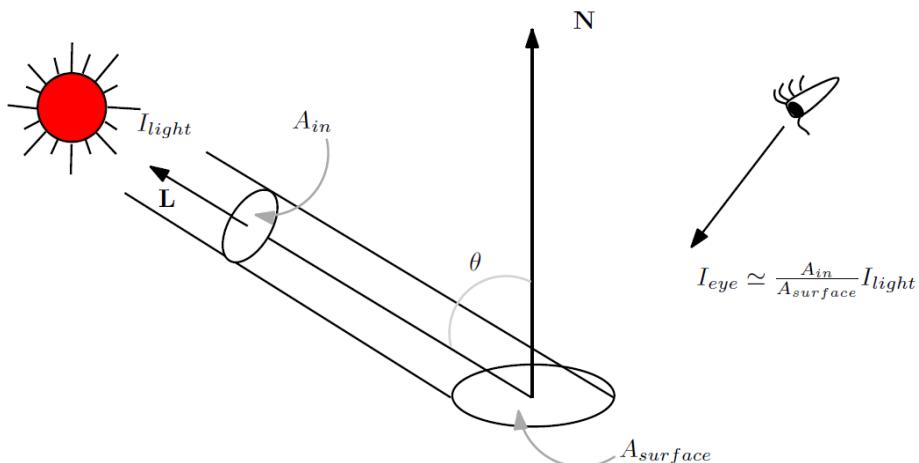
$$\kappa = \mathbf{l}_a \otimes \mathbf{m}_a + I_d \mathbf{l}_d \otimes \mathbf{m}_d + I_s \mathbf{l}_s \otimes \mathbf{m}_s$$

이제 눈에 관찰되는 색을 결정하기 위해서는 난반사의 광강도 I_d 와 정반사 광강도 I_s 를 계산하는 법을 알아야 한다.

풍 모델 광강도^{intensity}의 계산 - 난반사 모델

풍 모델에서 우리가 계산해야 하는 광강도는 난반사 광강도 I_d 와 정반사 광강도 I_s 이다. 이 가운데 우선 난반사 광강도를 먼저 구해 보자. 우리가 가정하는 난반사는 완벽한 난반사로

모든 방향에 동등하게 빛이 퍼진다. 따라서 눈이 어디에 있는지 동일한 색상 관찰이 이루어진다. 따라서 난반사는 눈의 움직임에 따라 변하는 하일라이트^{highlight}는 표현하지 못하며, 색을 칠하려고 하는 한 지점에 대해 어디서 쳐다 보든지 동일한 밝기 정도를 결정한다. 이 밝기는 얼마나 많은 에너지(빛)가 해당 지점에 떨어지는지에 달려 있다. 이러한 개념은 그림 16에서 확인할 수 있다. 빛이 처음에 I_{light} 의 강도로 표면에 떨어진다. 이때 입사하는 각도에 수직으로 빛을 절단했을 때, 일정한 면적 A_{in} 을 통과하는 빛은 표면의 $A_{surface}$ 면적에 흩어져 떨어지게 된다. 눈으로 관찰되는 에너지는 이 면적의 비인 $A_{in}/A_{surface}$ 에 비례하게 된다.



[그림 16] 난반사 광강도 결정 방법의 개념적 이해

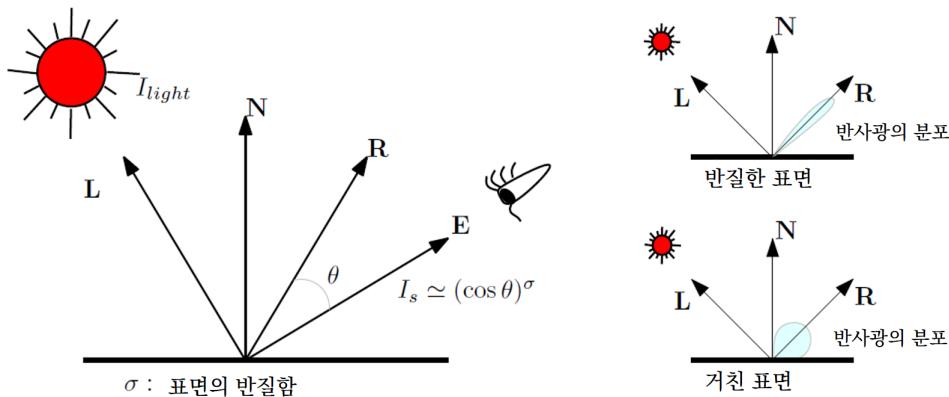
이 값은 광원 벡터 \mathbf{L} 과 법선 벡터 \mathbf{N} 이 일치할 때 최대이며, 90도를 이룰 때 0이 된다. 이 값은 두 벡터의 내적, 즉 두 벡터가 이루는 각 θ 의 코사인^{cosine}에 비례한다는 것이 램버트 반사^{Lambertian reflectance} 모델이다. 따라서 다음 식과 같이 광강도를 계산해야하는 지점에서 빛을 향하는 방향벡터 \mathbf{L} 과 표면 법선벡터 \mathbf{N} 의 내적으로 I_d 를 구할 수 있다

$$I_d = \cos \theta = \mathbf{L} \cdot \mathbf{N}$$

퐁 모델 광강도^{intensity}의 계산 - 정반사 모델

정반사는 거울과 같이 입사각에 대칭되는 방향으로 반사되는 것이다. 그런데, 실제 물체들은 이런 이상적인 정반사가 아니라 반사 방향으로 빛이 강하게 진행하기는 하지만 다른 방향으로 조금씩 빛이 나간다. 퐁 모델에서 사용하는 정반사 모델은 거울과 같은

반사가 아니라 반사 벡터 \mathbf{R} 중심으로 퍼지는 반사를 표현한다. 이 반사는 반사 벡터 \mathbf{R} 근처에서 강하게 관찰되기 때문에 눈을 \mathbf{R} 근처로 가져가야 강한 빛을 볼 수 있다. 따라서 정반사의 광강도 I_s 는 \mathbf{R} 과 \mathbf{E} 의 사잇각의 코사인에 연관된다. 그리고 물체의 재질에 따라 \mathbf{R} 방향으로 집중되는 정도가 달라진다. 이것은 물체의 반질함(shininess)에 의해 결정되는데, 이 반질함을 σ 라고 하자. 그러면 이 반질함을 지수로 하는 모델을 만들 수 있다. 그림 17은 이러한 정반사의 강도가 결정되는 방식을 개념적으로 보이고 있다.



[그림 17] 정반사의 강도를 계산하는 방법에 대한 개념적 설명

이러한 개념을 수식으로 표현하면 다음과 같다.

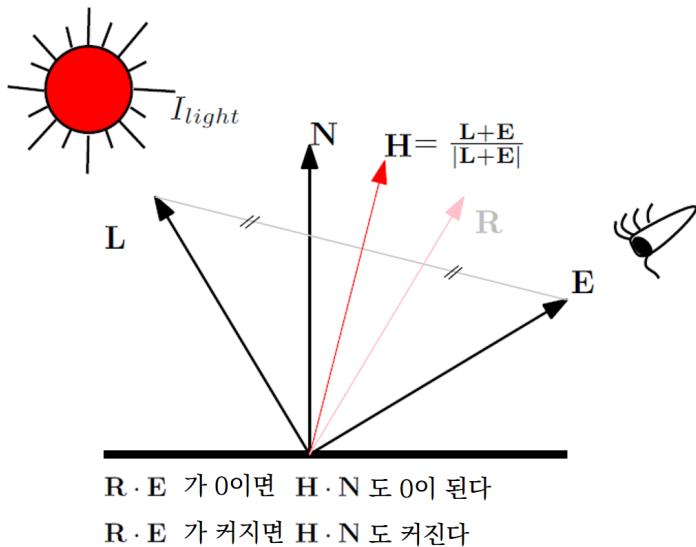
$$I_s = \cos\theta = (\mathbf{R} \cdot \mathbf{E})^\sigma$$

수정된 풍 모델 - 블린^{Blinn} 모델

풍 모델은 정반사 광강도 계산에서 반사 벡터 \mathbf{R} 을 계산해야 한다. 블린^{Blinn}은 따로 이 반사벡터 대신에 반 벡터(halfway vector)라는 것을 도입하였다. 반 벡터 \mathbf{H} 는 광원 벡터 \mathbf{L} 과 시선 벡터 \mathbf{E} 를 더해서 다음과 같이 정규화하면 된다.

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{E}}{|\mathbf{L} + \mathbf{E}|}$$

이러한 반벡터의 개념과, 이를 활용하여 반사 벡터 없이 정반사 광강도를 계산할 수 있는 이유가 그림 18에 나타나 있다.



[그림 18] 반 벡터를 이용한 광강도 계산이 가능한 이유

이러한 관찰을 근거로 정반사 광강도는 반사벡터 **R**이 아니라, 반 벡터를 이용하여 다음 같이 구할 수 있다는 것이 블린이 제안한 수정 모델이다.

$$\kappa = \mathbf{l}_a \otimes \mathbf{m}_a + (\mathbf{L} \cdot \mathbf{N}) \mathbf{l}_d \otimes \mathbf{m}_d + (\mathbf{H} \cdot \mathbf{N})^\sigma \mathbf{l}_s \otimes \mathbf{m}_s$$

9.5 OpenGL에서 조명 다루기

이 절에서는 실제 OpenGL 코드를 사용하여 물체에 조명을 비추는 작업을 수행할 것이다. 앞에서 다룬 풍 모델은 OpenGL이 기본적으로 사용하는 음영 계산 기법, 즉, 셰이딩 기법이다. 따라서 특별한 코딩^{coding}을 하지 않고 조명과 재질에 관한 정보만 OpenGL에 넘기면 광강도 계산을 알아서 처리하여 준다. 우리가 설정해야 할 것은 조명을 사용할 것이라는 것과, 조명의 색과 물체의 색을 결정하는 것이다. 조명과 재질 설정 조명과 재질은 난반사^{diffuse}, 정반사^{specular}, 주변광^{ambient}으로 구분하여 각각 설정해야 한다. 따라서 총 6 가지의 색상 설정이 필요하다. 그리고 재질의 반질거림^{shininess}까지 설정하면 된다. 다음으로

설정해야 하는 것은 조명의 위치이다. 조명의 위치는 동차좌표로 표현하는데, 마지막 성분이 1이면 점광원이고, 0이면 방향광원^{directional light source}라고 생각하면 된다. 다음과 같이 조명 계산에 필요한 값들을 준비하자.

```
mat_spec = [1.0, 1.0, 1.0, 1.0]
mat_diff = [0.0, 1.0, 1.0, 1.0]
mat_ambi = [1.0, 1.0, 1.0, 1.0]
mat_shin = [120]

lit_spec = [1.0, 1.0, 1.0, 1.0]
lit_diff = [1.0, 1.0, 1.0, 1.0]
lit_ambi = [0.0, 0.0, 0.0, 0.0]

light_pos = [1.0, 1.0, 1.0, 0.0]
```

이것은 다음과 같은 설정과 같다. 색상 값으로 네 개의 원소를 사용하는데, 빨강, 녹색, 파랑 채널의 강도와 함께 마지막 숫자는 불투명^{opacity} 정도를 의미한다.

$$\mathbf{m}_s = (1, 1, 1, 1)$$

$$\mathbf{m}_d = (1, 1, 1, 1)$$

$$\mathbf{m}_a = (1, 1, 1, 1)$$

$$\sigma = 120/128$$

$$\mathbf{l}_s = (1, 1, 1, 1)$$

$$\mathbf{l}_d = (1, 1, 1, 1)$$

$$\mathbf{l}_a = (1, 1, 1, 1)$$

$$\mathbf{L} = (1, 1, 1)$$

이상의 코드는 실제로 광원과 재질을 설정하는 것이 아니라 값만 준비해 둔 것이다. 이 값을 실제로 조명과 재질에 적용해야 한다. 조명의 특성을 설정하는 함수는 `glLight*`이며, 재질 특성을 설정하는 것은 `glMaterial*`이다. 이를 이용하여 다음 코드와 같이 물체의 재질과, 광원의 특성을 설정한다. 이 코드를 살펴보면, 우선 `LightSet` 함수는 조명과 재질의 색상을 지정한다. 그리고 나서 `GL LIGHTING`을 활성화하여 OpenGL이 조명을 사용하도록 한다. 또한 오픈지엘 고정 파이프라인이 사용할 수 있는 조명 가운데 어떤 조명을 사용할 것인지를 설정하고 있다. 이 경우 `GL LIGHT0` 하나만을 사용한다.

LightPositioning은 광원의 위치를 설정하고 있다. 이렇게 광원이 설정되면 법선 벡터를 가진 각 정점은 풍 모델에 의해 그 밝기와 색상이 결정된다.

```
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)
```

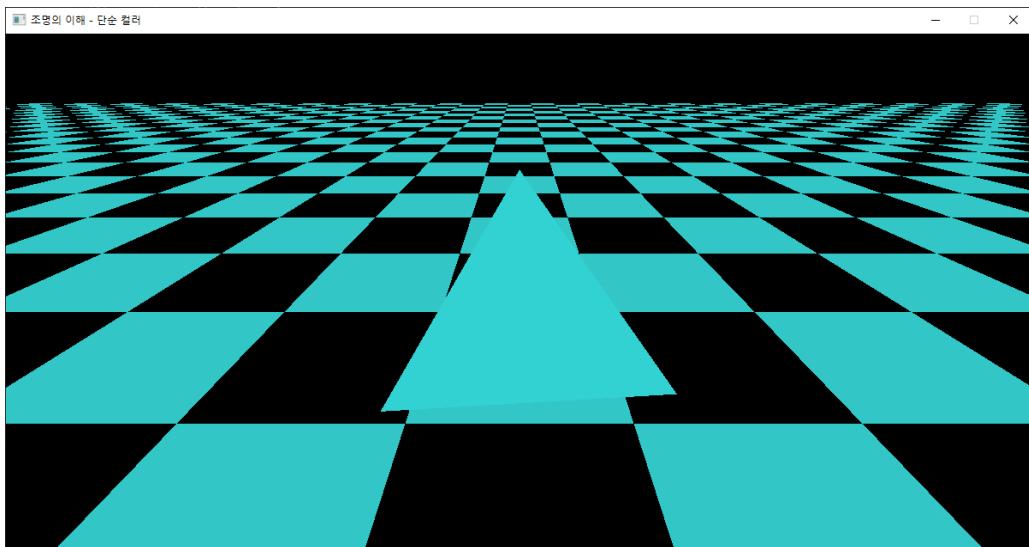
LightSet 함수는 OpenGL의 초기화 메소드 initializeGL에서 호출하면 된다. 그리고, 카메라 설정이 이루어지고 나면 좌표계의 기준이 변경되기 때문에, LightPositioning은 이러한 변경이 이루어진 뒤에 적용되어야 한다. 이 작업은 OpenGL 위젯의 paintGL 메소드에서 이루어진다. 그러면 다음과 같이 두 메소드를 수정한다.

```
class MyGLWidget(QOpenGLWidget):
    ...

    def initializeGL(self):
        ...
        LightSet()

    def paintGL(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        gluLookAt(0,7,10, 0,2,0, 0,1,0)
        LightPositioning()
        ...
```

이를 실행하면 다음과 같이 앞서 지정한 색들이 완전히 무시되고 단조로운 단색의 결과를 얻게 될 것이다. 삼각형으로 구성된 물체 뿐만 아니라 바닥도 같은 색이 된 것을 알 수 있다.

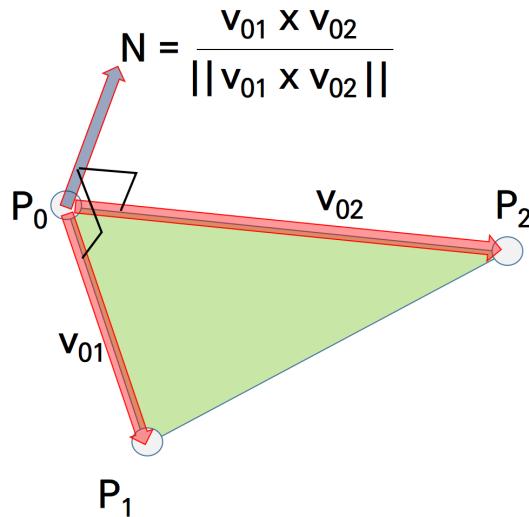


[그림 19] 법선 벡터가 지정되지 않아 음영이 나타나지 않는 모델

우리는 glEnable(GL_LIGHTING)을 이용하여 조명 모델을 사용하는 것을 활성화하였다. 그리고 이때 조명으로 동작하는 것이 LIGHT_0이다. 이 조명 개체를 사용하기 위해 glEnable(GL_LIGHT0)로 실행하였다. 그리고, 조명 계산에 필요한 조명의 색과 물체의 색을 지정하였다. 조명 계산은 이 값들에 의해서 이루어지지 우리가 glColor3f를 이용하여 입력한 색상은 고려되지 않는 것이다. 그러면 이 물체가 전혀 입체로 보이지 않는 이유는 무엇일까? 그것은 조명과 재질은 설정했지만 이 물체의 표면이 어느 방향으로 향하고 있는지를 설명할 수 있는 법선 벡터 \mathbf{N} 이 지정되지 않았기 때문이다.

이 법선벡터는 각 정점별로 설정될 수 있다. 만약 한 정점에 설정하고 변경하지 않는다면 이어서 나타나는 모든 정점에 대해서 동일한 법선이 사용된다. 법선 벡터는 어떻게 구할 수 있을까? 그림 20을 살펴보자. 세 개의 점으로 구성된 삼각형이 있다. 이 삼각형이 이루는 평면의 법선을 \mathbf{N} 이라고 하자. 이때 평면 위에서 두 개의 벡터 $\mathbf{v}_{01} = \mathbf{P}_1 - \mathbf{P}_0$ 과 $\mathbf{v}_{02} = \mathbf{P}_2 - \mathbf{P}_0$ 를 구할 수 있다. 법선 벡터는 그림에서 볼 수 있는 바와 같이 이 두 벡터 모두와 수직인 벡터이다. 따라서 이 두 벡터의 가위곱^{cross product}과 같은 방향이다. 따라서 길이가 1인 정규화된 법선 벡터는 이 가위곱을 정규화하면 된다. 따라서 법선 벡터는 다음과 같이 구할 수 있다.

$$\mathbf{N} = \frac{\mathbf{v}_{01} \times \mathbf{v}_{02}}{||\mathbf{v}_{01} \times \mathbf{v}_{02}||}$$



[그림 20] 세 점으로 구성된 삼각형의 법선 벡터 \mathbf{N} 계산하기

우선 법선 벡터가 명백한 부분을 고쳐 보자. 평면을 이루는 사각형들이다. 이들은 법선 벡터가 $(0, 1, 0)$ 이므로, drawPlane을 수정하여 다음과 같이 만들어 보자. 법선 벡터를 지정하는 것은 glNormal3f 함수이다.

```
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):
            ...
            ...
```

다음으로 세 개의 삼각형에 적절한 법선 벡터를 지정해 보자. 미리 계산해서 지정할 수도 있겠지만, 그것보다는 컴퓨터가 계산하여 법선을 지정할 수 있도록 하는 방식을 사용해

보자. 법선 벡터를 계산하기 위해서는 벡터의 연산이 필요하다. 이것은 넘파이가 쉽게 할 수 있도록 다양한 API를 제공한다. 따라서 정점의 위치들을 우선 넘파이 배열로 만들어 보자.

```
def paintGL(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    ...

    P0 = np.array([0,5,0])
    P1 = np.array([-3,1,3])
    P2 = np.array([ 3,1,3])
    P3 = np.array([ 0,1,-3])
```

다음으로 각 면을 그릴 때에 사용한 glColor3f 코드는 모두 없애고, 해당 면을 그릴 때에 적용할 법선을 계산한다. 우선 첫번째 삼각형의 법선을 계산하자. 넘파이의 cross 함수는 두 벡터의 외적을 구해 준다. 그리고 넘파이의 서브 패키지인 linalg는 다양한 선형대수 계산을 지원하는데 벡터의 크기는 norm으로 구할 수 있다. 이를 이용하여 법선 \mathbf{N} 을 계산할 수 있다.

```
def paintGL(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    ...
    glBegin(GL_TRIANGLES)
    u = P1-P0
    v = P2-P0
    N = np.cross(u, v)
    norm = np.linalg.norm(N)
    N = N / norm
```

이렇게 계산한 법선은 벡터 형식이기 때문에 glNormal3f가 아니라 glNormal3fv에 넘겨주면 된다. 다른 삼각형에 대해서도 동일한 방식으로 법선을 구하고 적용한다.

```
def paintGL(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    ...

    glBegin(GL_TRIANGLES)
    u = P1-P0
```

```

v = P2-P0
N = np.cross(u, v)
norm = np.linalg.norm(N)
N = N / norm
glNormal3fv(N)
glVertex3fv(P0)
glVertex3fv(P1)
glVertex3fv(P2)

u = P2-P0
v = P3-P0
N = np.cross(u, v)
norm = np.linalg.norm(N)
N = N / norm
glNormal3fv(N)
glVertex3fv(P0)
glVertex3fv(P2)
glVertex3fv(P3)

u = P3-P0
v = P0-P0
N = np.cross(u, v)
norm = np.linalg.norm(N)
N = N / norm
glNormal3fv(N)
glVertex3fv(P0)
glVertex3fv(P3)
glVertex3fv(P1)

glEnd()

```

이를 전체적으로 정리하면 다음과 같은 코드가 될 것이다.

조명 설정과 법선 적용이 완료된 그리기 코드

```

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

```

```

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()

mat_spec = [1.0, 1.0, 1.0, 1.0]
mat_diff = [0.0, 1.0, 1.0, 1.0]
mat_ambi = [1.0, 1.0, 1.0, 1.0]
mat_shin = [120]

lit_spec = [1.0, 1.0, 1.0, 1.0]
lit_diff = [1.0, 1.0, 1.0, 1.0]
lit_ambi = [0.0, 0.0, 0.0, 0.0]

light_pos = [1.0, 1.0, 1.0, 0.0]

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)

```

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

        self.angle = 0

    def initializeGL(self):
        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()

    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(0,7,10, 0,2,0, 0,1,0)
        LightPositioning()

        glCallList(self.planeList)

        P0 = np.array([0,5,0])
        P1 = np.array([-3,1,3])
        P2 = np.array([ 3,1,3])
        P3 = np.array([ 0,1,-3])

        glRotatef(self.angle, 0, 1, 0)
        self.angle += 1

    glBegin(GL_TRIANGLES)
    u = P1-P0
```

```

v = P2-P0
N = np.cross(u, v)
norm = np.linalg.norm(N)
N = N / norm
glNormal3fv(N)
glVertex3fv(P0)
glVertex3fv(P1)
glVertex3fv(P2)

u = P2-P0
v = P3-P0
N = np.cross(u, v)
norm = np.linalg.norm(N)
N = N / norm
glNormal3fv(N)
glVertex3fv(P0)
glVertex3fv(P2)
glVertex3fv(P3)

u = P3-P0
v = P0-P0
N = np.cross(u, v)
norm = np.linalg.norm(N)
N = N / norm
glNormal3fv(N)
glVertex3fv(P0)
glVertex3fv(P3)
glVertex3fv(P1)

glEnd()

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

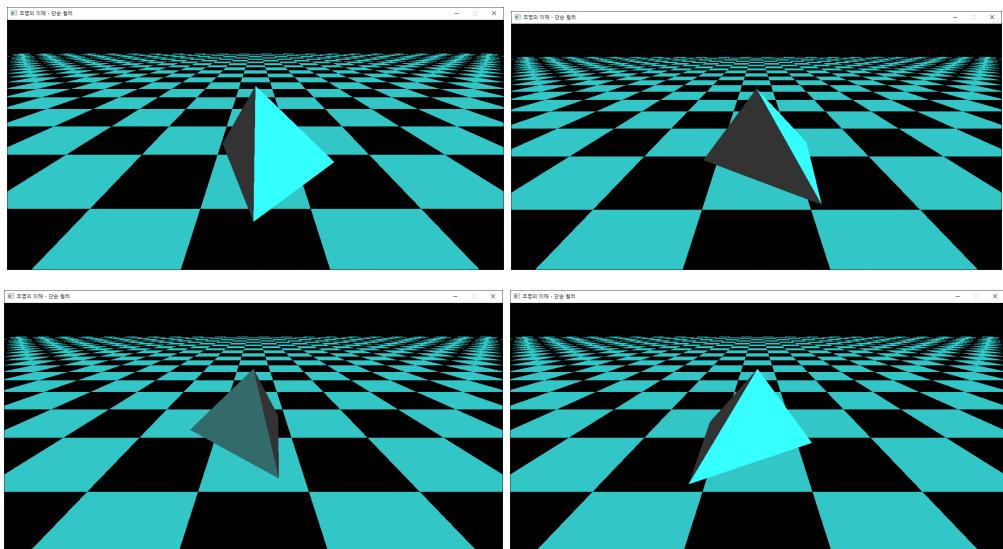
    def keyPressEvent(self, e):
        self.glWidget.update()

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)
```

이 코드를 실행하면 다음과 같이 조명 환경에 따라 적절한 반사가 이루어지는 결과를 얻을 수 있을 것이다.



[그림 21] 조명과 볍선이 제대로 설정된 결과