

## Chapter 06 -

동명대학교 게임공학과 3D 그래픽스 프로그래밍

# 파일에 담긴 기하 객체 읽고 그리기



“소프트웨어 재사용 전에, 우선은 사용부터 가능해야 한다.”

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

– 랄프 존슨 Ralph Johnson

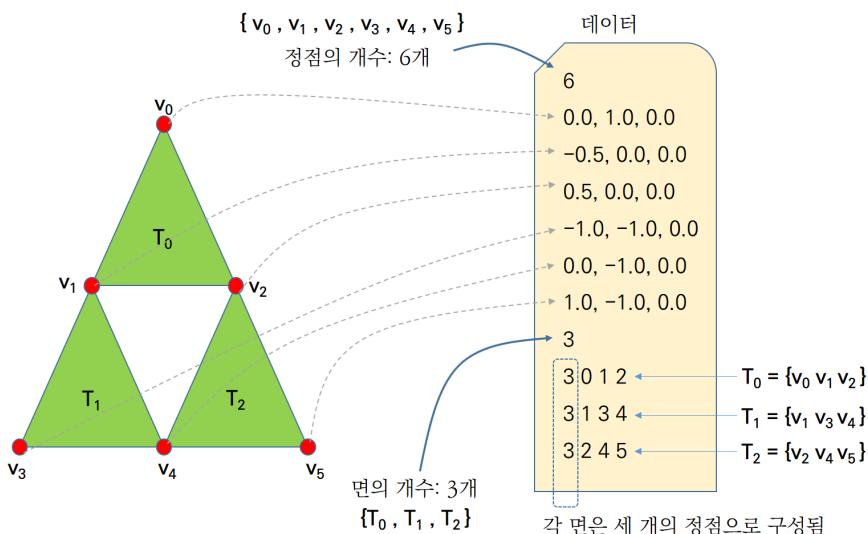
### 이 장에서 생각할 문제

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

## 6.1 간단한 메시 데이터 포맷

기하 객체의 정보를 담기 위해 우리는 간단한 메시<sup>mesh</sup> 데이터 포맷을 만들어 보려고 한다. 우리가 다룰 기하 객체는 삼각형을 기본 요소로 할 것이다. 삼각형을 표현하기 위해서는 정점이 필요하고, 하나의 삼각형을 구성하는 데에 세 개씩의 정점을 가져다 써야 할 것이다. 우리는 앞 장에서 `glDrawElements`를 다루면서 정점의 리스트와, 각 면에 사용할 정점을 정수 인덱스로 표시하는 방식을 살펴본 바가 있다. 이러한 방식으로 데이터를 파일에 담는다면 다음과 같은 간단한 양식을 생각할 수 있을 것이다.

데이터 양식	실제 데이터 예시
정점의 개수 ( $nV$ )	6
정점 0의 좌표	0.0, 1.0, 0.0
정점 1의 좌표	-0.5, 0.0, 0.0
정점 2의 좌표	0.5, 0.0, 0.0
...	-1.0, -1.0, 0.0
정점 $nV-1$ 의 좌표	0.0, -1.0, 0.0
면의 개수 ( $nF$ )	1.0, -1.0, 0.0
(면 0 구성 정점의 수 $k$ ) (정점 번호) ... $k$ 개	3
(면 1 구성 정점의 수 $k$ ) (정점 번호) ... $k$ 개	3 0 1 2
...	3 1 3 4
(면 $nF-1$ 구성 정점의 수 $k$ ) (정점 번호) ... $k$ 개	3 2 4 5



[그림 1] 기하 객체의 예시와 이를 표현하는 메시 데이터

이러한 메시 데이터를 읽어 들일 수 있는 클래스를 하나 만들어 보자. 메시 데이터를 읽어 들이는 클래스의 이름을 MeshLoader라고 짓고, 두 개의 메소드method를 구현할 계획이다. 이 두 메소드는 메시 데이터를 읽어 들여서 정보를 저장하는 loadData와 이 정보를 바탕으로 그림을 그리는 draw이다. 기본적인 모양은 이렇게 될 것이다.

```
class MeshLoader:  
    def __init__(self):  
        ...  
    def loadData(self, filename) :  
        ...  
    def draw(self):  
        ...
```

이 클래스가 관리할 정보는 정점의 개수 nV, 면의 개수 nF, 정점 배열 vertexBuffer, 그리고 면을 구성하는 정점의 인덱스들을 담는 정점 인덱스 배열 idxBuffer이다. 따라서 클래스 생성자는 이 값들을 0이나 None으로 설정하는 일만 하자.

```
def __init__(self):  
    self.nV = 0  
    self.nF = 0  
    self.vertexBuffer = None  
    self.idxBuffer = None
```

다음으로 데이터를 읽어 들이는 loadData 메소드를 작성하자. loadData 메소드에는 인자로 읽어들일 파일의 이름 filename을 제공하도록 하였다. octahedron.txt라는 파일에 다음과 같은 데이터가 담겨 있다고 가정해 보자.

```
6  
1 0 0  
0 -1 0  
-1 0 0  
0 1 0  
0 0 1  
0 0 -1  
8  
3 4 0 1  
3 4 1 2  
3 4 2 3
```

```
3 4 3 0  
3 5 1 0  
3 5 2 1  
3 5 3 2  
3 5 0 3
```

6개의 점을 가지고, 8 개의 면을 만들고 있다. 각각의 면은 3 개의 점으로 구성되므로 각 면은 모두 삼각형이다. 이것은 8면체를 나타내는 데이터이다. 이 데이터를 코드가 저장된 디렉토리에 두고 './octahedron.txt'를 파일 이름으로 넘기면 읽는 것이다. 읽기 동작은 다음과 같이 구현할 수 있다.

```
def loadData(self, filename) :  
    with open(filename, "rt") as inputFile:  
  
        ... # 파일 다루기 동작을 여기에 작성
```

우선 이 메소드는 `with open()` 으로 시작한다. `open`은 파일을 여는 동작을 한다. 첫 번째 인자는 파일의 이름이고 두 번째 인자는 파일을 열 때 사용하는 옵션이다.

`open(filename, option)`

`with` 문을 사용하지 않고 `open()`으로 연 파일은 다 쓰고 나면 `close()`로 닫아야 한다. 그런데, `with` 문 안에 `open`을 사용해 파일을 열면 조금 다르다. 파일은 컨텍스트 매니저로 다뤄지는 대표적인 객체로서 `__enter__()`와 `__exit__()` 메소드를 가지고 있다. `__enter__()`에 의해 반환된 파일 객체는 `as handle`에 의해 그 이름이 `handle`로 다루어지게 된다. 위의 예에서는 `as inputFile`을 사용하여 이후의 동작에서 `inputFile`을 이용하여 값을 읽어 오고 한다. 'rt'라는 옵션은 읽기를 의미하는 'r'와 텍스트 파일임을 의미하는 't'로 이루어져 있다. 다양한 파일 열기 옵션에 대해서는 파일썬 기본 자료를 참고하기 바란다. 파일을 다루는 작업이 끝나고 나면 `with` 문이 종료된다. 이때, `with` 문을 사용 하지 않았다면 `inputFile.close()`를 이용하여 파일을 닫아줘야 한다. 그러나, 이 파일 닫기는 파일을 여는 것과 쌍을 이루는 동작으로 이미 파일 객체의 `__exit__()` 메소드에 담겨 있다. `with` 문은 종료하면서 이 `__exit__()` 메소드를 부르므로 파일을 자동으로 닫힌다. 따라서 `close()`를 호출하는 일을 하지 않아도 되는 것이다.

```

def loadData(self, filename) :
    with open(filename, "rt") as inputFile:

        self.nV = int(next(inputFile)) # 정점의 개수
        self.vertexBuffer = np.zeros(shape = (self.nV*3, ),
                                     dtype=float)

```

파일을 열고 나면, 파일에서 데이터를 가져오는 일을 구현해 보자. 파일에서 하나의 행을 문자열로 가져오는 함수는 next(file)이다. 위에서는 next(inputFile)로 표현되어 있다. 이렇게 읽은 결과는 문자열로 전달이 된다. 따라서 위의 예에서 octahedron의 첫 행인 ‘6’이 반환될 것이다. 이것은 숫자 6으로 변환되어 self.nV에 저장된다. 그리고 나면 6 개의 정점을 담을 정점 좌표 배열을 생성해야 한다. 이를 위해 넘파이(numpy)를 사용하자. 넘파이가 임포트되었고 np라는 별칭으로 사용하도록 지정되었다고 하자. 이것은 다음과 같은 임포트가 이루어졌다는 것이다.

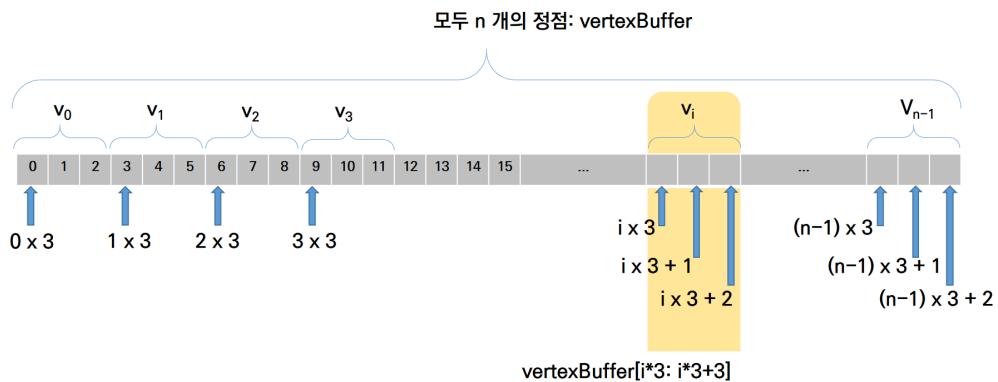
```
import numpy as np
```

그러면 np.array(...)를 이용하여 넘파이 배열을 생성할 수 있다. self.vertexBuffer를 정점을 담을 배열로 사용하자. 정점의 개수는 self.nV이고, 하나의 정점은 3 개의 좌표로 구성되므로 배열은  $3 * \text{self.nV}$  개의 숫자로 이루어진 1차원 배열이다. 이것은 넘파이 배열을  $(\text{self.nV} * 3, )$ 의 형태로 만들면 된다. 우선은 데이터가 읽히지 않았으므로 공간만 잡고 이 공간을 모두 0으로 채우기 위해 넘파이의 zeros 함수를 사용하자. 사용되는 좌표의 자료형은 부동소수점이 되어야 하므로 명시적으로 자료형을 지정하는 것도 포함했다.

```
self.vertexBuffer = np.zeros(shape = (self.nV*3, ), dtype=float)
```

이 공간에는 모두 n 개의 정점이 가진 좌표가 담기게 된다. 하나의 정점이 3 개의 좌표값을 갖기 때문에 이 공간의 아래 그림과 같이  $n \times 3$  개의 원소를 가진 배열로 표현된다. 정점이  $\mathbf{v}_0$ 부터  $\mathbf{v}_{n-1}$ 까지 번호가 붙여 있다고 하면,  $i \times 3$  인덱스가  $\mathbf{v}_i$  정점의 첫 좌표가 기록되는 곳이 된다. 넘파이 배열은 슬라이싱을 통해 배열내의 일부 영역을 효율적으로 접근할 수 있다. 다음으로 구현되는 부분은 파일에서 정점의 개수만큼 행을 읽고 각 행에 공백 문자로 띠어져 있는 세 개의 숫자를 읽어 저장하는 것이다. 이 작업은 self.nV 횟수 만큼 이루어지며, i 번째 데이터를 읽을 때 읽어들인 세 숫자는 self.vertexBuffer[i\*3: i\*3+3]이 표현하는 부분에 대입되면 된다. 문자열에 split() 메소드를 사용하면 특정한 문자를 분리 문자로 사용하여 이 문자를 기준으로 원래의 문자열을 여러 개의 문자열로 분리한 뒤

리스트에 담는다. `split()` 메소드에 인자를 넣지 않으면, 공백<sup>white space</sup>를 기준으로 분리를 한다. 한 행에는 한 정점의 좌표 세 개가 들어 있기 때문에 `next(inputFile).split()`를 실행하면 세 개의 문자열이 담긴 리스트가 된다. 이것을 `vertexBuffer`의 일정 영역에 대입하는 것이다. 이때 문자열이 담기지 않고 배열의 데이터 형이 부동소수점으로 변환되어 저장된다. 저장되는 영역에 대한 이해를 위해 그림 2를 참고하라.



[그림 2] 1차원 배열로 구현된 정점 배열의 구조

데이터에 따라 좌표값이 매우 클 수도 있고, 작은 값을 가질 수도 있어 카메라를 어디에 두고 관찰해야 할지 정하기 어려울 수 있다. 우리는 메시를 구성하는 좌표값의 최소가 -1, 최대가 1인 범위 내에 있도록 고칠 수 있다. 이것은 정점 배열에 들어 있는 값 중에 가장 큰 값과 작은 값을 찾아 둘 중에 절대값이 큰 값을 찾아 그 크기로 전체 배열을 나누어 버리면 된다. 그러면 모든 좌표는 [-1, 1]의 범위에 들게 될 것이다.

```

def loadData(self, filename) :
    with open(filename, "rt") as inputFile:

        self.nV = int(next(inputFile)) # 정점의 개수
        self.vertexBuffer = np.zeros(shape = (self.nV*3, ),
                                     dtype=float)

        for i in range(self.nV) :
            self.vertexBuffer[i*3: i*3+3] = next(inputFile).split()
            # 한 줄을 읽어 공백으로 분리 저장

        coordMin = self.vertexBuffer.min()
        coordMax = self.vertexBuffer.max()
    
```

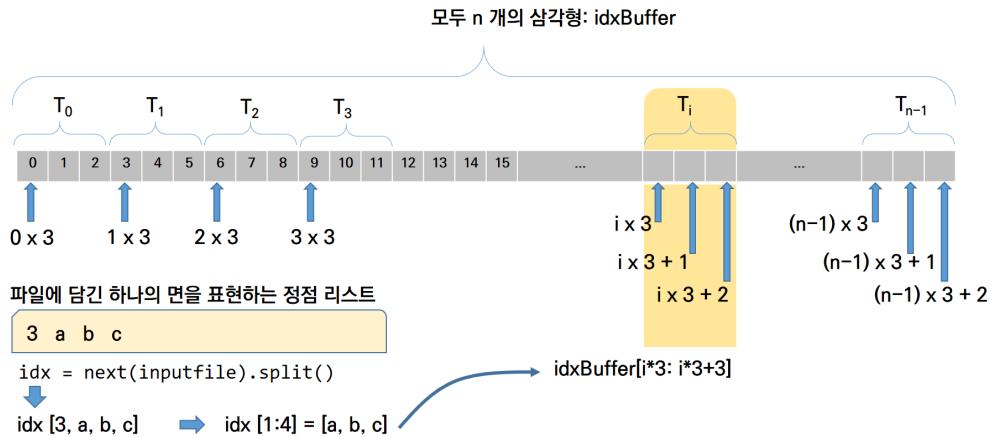
```
    scale = max([coordMin, coordMax], key=abs)
    self.vertexBuffer /= scale
```

다음으로 우리는 면에 대한 정보를 읽어야 한다. 면에 대한 정보는 면의 개수로 시작한다. 이것은 정점의 개수를 읽는 방식과 같이 다음처럼 구현할 수 있다. 그리고, 면의 개수가 결정되면 면을 구성하는 정점 인덱스가 채워질 인덱스 배열도 준비하자. 하나의 면을 구성하는 정점의 개수가 3 개가 아니고 4 개나 더 많을 수도 있지만, 이 책에서는 설명은 간단히 하기 위해 3개의 정점, 즉 삼각형으로 각각의 면이 이루어진다고 가정하자. 그러면 면의 개수가  $nF$  개이며, 필요한 정점은  $nF \times 3$  개가 된다. 그리고 각각의 정점은 좌표를 기록하는 것이 아니라 몇 번 째 정점인지 그 번호만 기록하면 된다. 따라서 하나의 면에 세 개의 정수가 필요하고, 전체적으로는  $nF \times 3$  개의 정수로 전체 데이터를 다룰 수 있다.

```
def loadData(self, filename) :
    with open(filename, "rt") as inputFile:
        ...
        self.nF = int(next(inputFile))
        self.idxBuffer = np.zeros(shape = (self.nF*3, ), dtype = int)
```

이제  $nF$  개의 행을 반복적으로 읽으면서 데이터를 인덱스 배열에 채우면 된다. 이때 읽어들인 행의 첫번째 원소는 정점의 개수를 의미하는데, 무조건 3이라고 가정하고 있다. 따라서 4 개의 숫자 중에 뒤의 세 개만 사용하면 된다. 읽어들인 정수가 idx에 저장되어 있다면 idx[1], idx[2], idx[3]을 사용하면 되고, 이는 idx[1:4]로 표현할 수 있다. 코드에서 읽어들인 내용이 인덱스 배열에 어떻게 저장되는지를 그림 3에서 보여주고 있다.

```
def loadData(self, filename) :
    with open(filename, "rt") as inputFile:
        ...
        for i in range(self.nF):          # 삼각형의 개수
            idx = next(inputFile).split()
            self.idxBuffer[i*3: i*3+3] = idx[1:4]
            # 한 줄을 읽어 공백으로 분리해 첫 원소 제외하고 저장
```



[그림 3] 면을 구성하는 정점 리스트를 읽어 저장하는 방법

지금까지의 내용을 정리하면 다음과 같은 loadData 메소드가 구현된다.

```

def loadData(self, filename) :
    with open(filename, "rt") as inputfile:

        self.nV = int(next(inputfile)) # 정점의 개수
        self.vertexBuffer = np.zeros(shape = (self.nV*3, ), 
                                     dtype=float)

        for i in range(self.nV) :
            self.vertexBuffer[i*3: i*3+3] = next(inputfile).split()
            # 한 줄을 읽어 공백으로 분리 저장

            coordMin = self.vertexBuffer.min()
            coordMax = self.vertexBuffer.max()
            scale = max(coordMin, coordMax)
            self.vertexBuffer /= scale

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

        for i in range(self.nF):          # 삼각형의 개수
            idx = next(inputfile).split()
            self.idxBuffer[i*3: i*3+3] = idx[1:4]
            # 한 줄을 읽어 공백으로 분리해 첫 원소 제외하고 저장

```

이제 이렇게 저장된 데이터를 이용하여 그림을 그리는 코드를 작성해 보자. 이것은 기본적으로 하나의 면을 구성하는 정점 리스트를 읽은 뒤에, 이 정점 리스트에 담긴 인덱스를 이용하여 정점 좌표를 찾아 오면 된다. 하나의 면마다 GL\_LINE\_LOOP을 이용하여 그림을 그리면 각 면은 선으로 그려진 삼각형 모양을 관찰할 수 있다. 코드로 표현하면 다음과 같다.

```
def draw(self):
    glColor3f(0,1,0.5)
    for i in range(self.nF):
        glBegin(GL_LINE_LOOP)
        vIdx = self.idxBuffer[i*3: i*3+3]
        v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
        glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
        glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
        glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
    glEnd()
```

이렇게 만들어진 클래스를 사용할 때는 다음과 같이 사용할 수 있다. 이 코드는 octahedron.txt라는 파일명으로 저장된 메쉬를 읽고 그리는 것이다.

```
class MyGLWidget(QOpenGLWidget):

    ...

    def initializeGL(self):
        # OpenGL 그리기를 수행하기 전에 각종 상태값을 초기화
        glClearColor(0.0, 0.0, 0.0, 1.0)

        self.meshLoader = MeshLoader()
        self.meshLoader.loadData('octahedron.txt')

    ...

    def paintGL(self):
        ...

        gluLookAt(2,0.5,2, 0,0,0, 0,1,1)
        self.meshLoader.draw()
```

전체적으로 동작하는 프로그램 코드를 작성하면 다음과 같은 형태가 될 것이다.

### 메쉬 데이터 클래스를 이용한 기하객체 그리기

```
from OpenGL.GL import *
from OpenGL.GLU import *
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow
from PyQt6.QtWidgets import QWidget, QHBoxLayout
from PyQt6.QtOpenGLWidgets import QOpenGLWidget
from PyQt6.QtCore import *
import math
import numpy as np

class MeshLoader:
    def __init__(self):
        self.nV = 0
        self.nF = 0
        self.vertexBuffer = None
        self.idxBuffer = None

    def loadData(self, filename) :
        with open(filename, "rt") as inputFile:

            self.nV = int(next(inputFile)) # 정점의 개수
            self.vertexBuffer = np.zeros(shape = (self.nV*3, ), dtype=float)

            for i in range(self.nV) :
                self.vertexBuffer[i*3: i*3+3] = next(inputFile).split()
                # 한 줄을 읽어 공백으로 분리 저장

            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):          # 삼각형의 개수
                idx = next(inputFile).split()
                self.idxBuffer[i*3: i*3+3] = idx[1:4]
                # 한 줄을 읽어 공백으로 분리해 첫 원소 제외하고 저장
```

```

def draw(self):
    glColor3f(0,1,0.5)
    for i in range(self.nF):
        glBegin(GL_LINE_LOOP)
        vIdx = self.idxBuffer[i*3: i*3+3]
        v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
        glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
        glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
        glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
        glEnd()

class MyGLWidget(QOpenGLWidget):

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

    def initializeGL(self):
        # OpenGL 그리기를 수행하기 전에 각종 상태값을 초기화
        glClearColor(0.0, 0.0, 0.0, 1.0)

        self.meshLoader = MeshLoader()
        self.meshLoader.loadData('octahedron.txt')

    def resizeGL(self, width, height):
        # 카메라의 투영 특성을 여기서 설정
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.1, 1000)

    def paintGL(self):

        glClear(GL_COLOR_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()

        gluLookAt(2,0.5,2, 0,0,0, 0,1,1)
        self.meshLoader.draw()
        glFlush()

class MyWindow(QMainWindow):

    def __init__(self, title=''):
        QMainWindow.__init__(self) # QMainWindow 슈퍼 클래스의 초기화
        self.setWindowTitle(title)

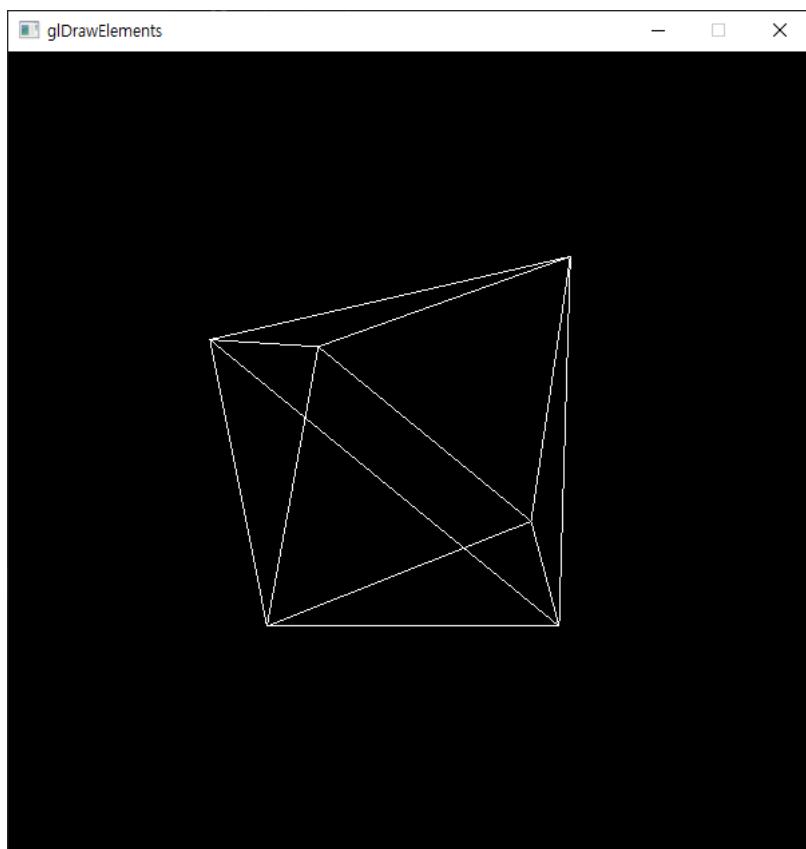
        self.glWidget = MyGLWidget() # OpenGL Widget
        self.setCentralWidget(self.glWidget)

```

```
def main(argv = []):
    app = QApplication(argv)
    window = MyWindow('glDrawElements')
    window.setFixedSize(600, 600)
    window.show()
    app.exec()

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

이 코드를 실행하면 octahedron.txt의 정보가 다음 그림 4와 같이 가시화되어 나타날 것이다.



[그림 4] 메시 데이터를 읽고 가시화한 결과

## 6.2 메시의 면 그리기 - 깊이 버퍼의 필요성

메시의 면을 그리기 위해서는 동일한 정점을 사용하면서 GL\_TRIANGLES 프리미티브를 이용하여 그리면 된다. 정점마다 색을 지정하면 삼각형을 칠할 수 있는데, 색은 정점의 좌표에 따라 결정되도록 하자. 그런데 색상은 R, G, B 세 가지 채널마다 0에서 1까지의 값을 가질 수 있는데 정점의 좌표값은 우리가 구현한 코드에 의하면 -1에서 1까지의 값을 가질 수 있다. 따라서 이를 0에서 1 사이의 값으로 바꾸는 작업이 필요하다.

이것은 좌표값에 1을 더하여 [0,2]의 범위에 들어가게 한 뒤 2로 나누면 된다. 모든 원소에 1을 더하기 위해서는 넘파이의 브로드캐스팅broadcasting을 적용하면 된다. 이것은 다음과 같이 구현할 수 있다.

```
self.vertexBuffer[v0*3: v0*3+3]+np.array([1])
```

이것은 3개의 원소를 가진 배열에 하나의 원소를 가진 배열 [1]을 더하는 것인데, 하나의 원소를 가진 배열은 더해지는 배열의 원소와 개수를 맞춰 [1,1,1]인 것처럼 모든 원소에 더해진다. 이를 브로드캐스팅이라고 하며 넘파이를 이용하여 대규모 배열을 다룰 때에 매우 유용하면서 계산의 효율성을 가져다 주는 방법이다. 자세한 내용은 부록을 참고하라.

이렇게 해서 얻은 값들을 모두 2로 나누고 싶을 때도 브로드캐스팅을 사용한다.

```
self.vertexBuffer[v0*3: v0*3+3]+np.array([1]))/2
```

그러면 그리기 코드는 다음과 같이 바뀔 것이다.

```
def draw(self):
    glColor3f(1,1,1)

    glBegin(GL_TRIANGLES)
    for i in range(self.nF):
        vIdx = self.idxBuffer[i*3: i*3+3]
        v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
        glColor3fv((self.vertexBuffer[v0*3: v0*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
        glColor3fv((self.vertexBuffer[v1*3: v1*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
        glColor3fv((self.vertexBuffer[v2*3: v2*3+3]+np.array([1]))/2)
```

```

glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
glEnd()

glColor3f(0,0,1)
for i in range(self.nF):
    glBegin(GL_LINE_LOOP)
    vIdx = self.idxBuffer[i*3: i*3+3]
    v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
    glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
    glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
    glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
    glEnd()

```

이를 이용하여 그림을 그리면 그림 2와 같이 나타날 것이다. 무엇인가 이상한 것을 발견할 수 있을 것이다. 이 결과는 앞에 있는 면이 뒤에 있는 면을 가리는 현상을 보여주지 못 한다.

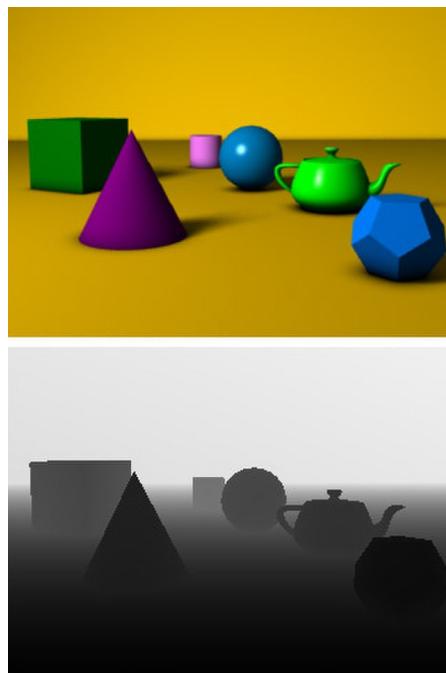


[그림 2] 면의 전후를 고려한 렌더링이 이루어지지 않는 결과

이 문제는 그려지는 픽셀이 카메라에서 얼마나 멀리 떨어져 있는가를 고려하지 않았기 때문이다. 카메라가 원점에 있고,  $z$  축 음수 방향을 쳐다보고 있다면, 그려지는 각 픽셀의  $z$

좌표값을 고려하여 멀리 있는 픽셀보다 가까이 있는 픽셀이 화면에 그려지는 데에 우선권을 갖도록 하는 일이 필요하다. OpenGL은 그리기 코드에 의해 무수히 많은 픽셀을 그리는데 우리 눈에 관찰되는 색상 정보는 색상 버퍼<sup>color buffer</sup>에 기록된다. 그런데 픽셀을 그릴 때에 색상 뿐만이 아니라 카메라로부터의 거리를 함께 기록한다고 가정해 보자. 그러면 픽셀을 그릴 때, 기록된 거리값을 참고해서 새롭게 그리려고 하는 픽셀이 기존의 픽셀보다 카메라에 가까이 있을 경우에만 실제로 픽셀 그리기를 실행하면 될 것이다. 이것을 깊이 테스트<sup>depth test</sup>라고 부른다. 그리고  $z$  좌표값, 혹은 깊이 값을 저장한 버퍼를 깊이 버퍼<sup>depth buffer</sup>라고 한다.

$z$ -버퍼라고도 불리는 깊이 버퍼는 영상을 생성할 때에 사용되는 여러 버퍼 가운데 하나이다. 대표적인 두 버퍼는 색상 버퍼와 깊이 버퍼인데, 색상 버퍼는 그림 3의 위쪽과 같이 우리 눈에 관찰되는 이미지를 그대로 옮겨 놓은 것이라 할 수 있다. 그런데, 이러한 색상 버퍼를 구성할 때 픽셀을 그릴 것인지 말 것인지를 결정할 때는 픽셀의 색상이 아니라 픽셀의 깊이값을 저장한 아래쪽의 깊이 버퍼를 참고해야 한다. 이 깊이 버퍼를 참고하여 새로운 픽셀이 기록된 깊이에 비해 더 얕은 곳에 있으면 픽셀을 그리는 일이 실제로 이루어지게 된다. 이 경우 색상 버퍼의 내용이 바뀔 것이고, 깊이 버퍼의 내용도 갱신된다.



[그림 3] 간단한 3차원 공간을 표현하는 색상 버퍼(위)와 대응하는 깊이 버퍼(아래)<sup>1</sup>

---

<sup>1</sup> T-tus, CC BY 2.0 via Wikimedia Commons

깊이 버퍼를 이용하여 깊이 테스트를 하려면 glEnable을 이용하여 GL\_DEPTH\_TEST를 활성화하여야 한다. OpenGL의 디폴트 상태는 깊이 테스트를 수행하지 않는 것이다. 따라서 다음과 같은 코드를 초기화 함수에서 실행한다.

```
glEnable(GL_DEPTH_TEST)
```

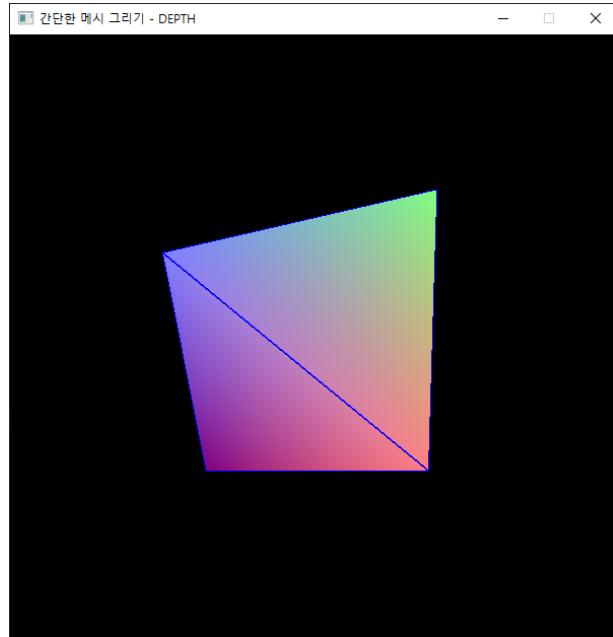
매 프레임 그림을 그릴 때 우리는 다음과 같이 색상 버퍼를 지우고 새로 준비하였다.

```
glClear(GL_COLOR_BUFFER_BIT)
```

이와 같은 방식으로 깊이 버퍼 역시 새로 준비해야 한다. 이것은 다음과 같은 방식으로 이루어진다.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

이상과 같이 깊이 테스트를 수행하도록 하고 매 프레임마다 깊이 버퍼를 지우고 새로 시작하면 우리가 기대하는 바와 같이 앞에 있는 물체가 뒤의 물체를 가리게 된다.



[그림 4] 깊이 버퍼가 사용되고 깊이 테스트를 수행하여 렌더링한 결과

깊이 버퍼를 사용하는 데에는 앞에서 본 코드에서 MyGLWidget 클래스만 수정하면 된다. 수정은 아래와 같이 진하게 표시된 두 줄만 변경하면 된다.

### 깊이 버퍼를 적용한 코드

```
class MyGLWidget(QOpenGLWidget):

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

    def initializeGL(self):
        # OpenGL 그리기를 수행하기 전에 각종 상태값을 초기화
        glClearColor(0.0, 0.0, 0.0, 1.0)

        self.meshLoader = MeshLoader()
        self.meshLoader.loadData('octahedron.txt')
        glLineWidth(2)
        glEnable(GL_DEPTH_TEST)

    def resizeGL(self, width, height):
        # 카메라의 투영 특성을 여기서 설정
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.1, 1000)

    def paintGL(self):

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()

        gluLookAt(2, 0.5, 2, 0, 0, 0, 0, 1, 1)
        self.meshLoader.draw()
        glFlush()
```

다양한 메시를 준비해 두었다. 아래 저장소를 살펴보면 우리가 사용한 octahedron.txt 뿐만 아니라 다른 메시들도 얻을 수 있을 것이다.

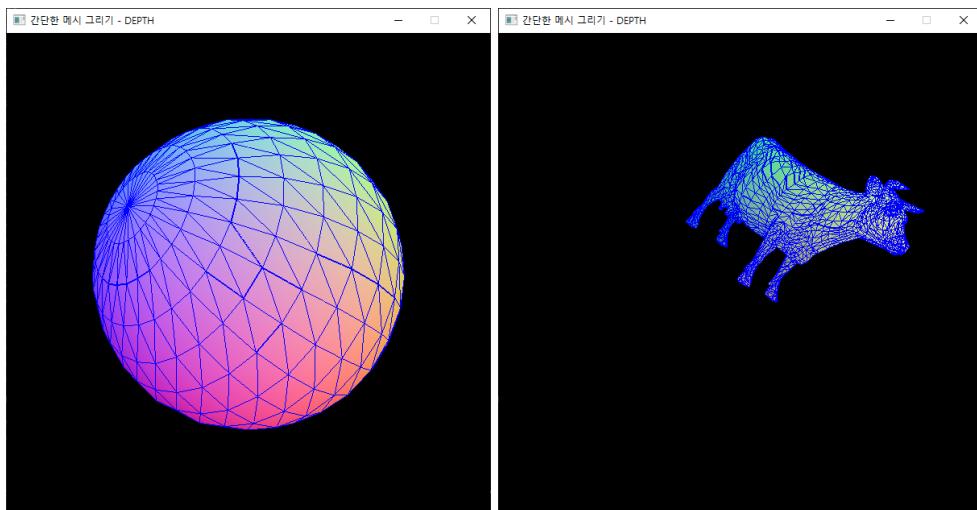
<https://github.com/dknife/opendata/tree/main/mesh>

예를 들어 다음의 구<sup>sphere</sup> 메시를 다운로드해서 앞의 코드에서 파일 이름을 변경하여 로딩해 보자.

<https://github.com/dknife/opendata/raw/main/mesh/sphere.txt>

<https://github.com/dknife/opendata/raw/main/mesh/cow.txt>

다음과 같은 결과를 얻을 수 있을 것이다.

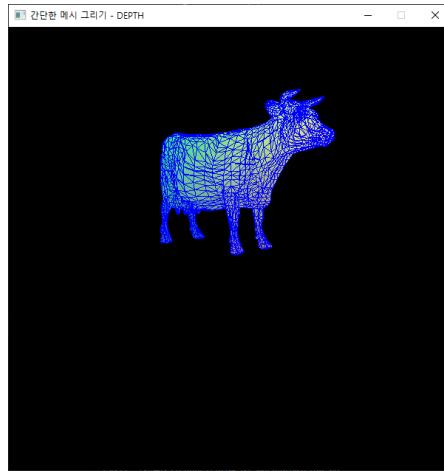


[그림 5] 다양한 메시를 읽기

### 6.3 객체의 위치와 방향을 바꾸기

OpenGL에는 glTranslatef, glRotatef와 같은 함수가 있어 그리기 대상이 되는 객체를 이동<sup>translate</sup>하거나, 회전<sup>rotate</sup>할 수 있다. 다음과 같이 paintGL을 바꾸어 그 결과를 살펴 보자. 우선은 아무런 변화 없이 메시를 그대로 그려보자. 카메라의 상향 벡터를 y 축을 향하게 하였다.

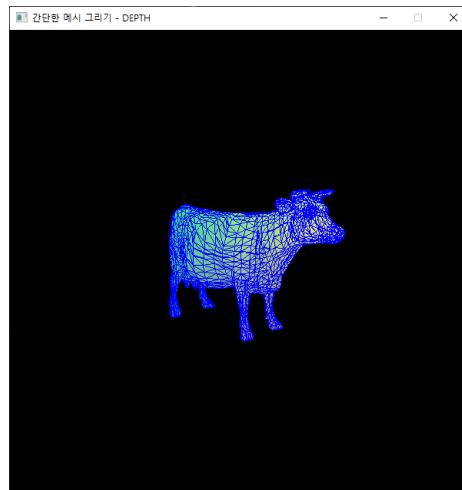
```
def paintGL(self):
    ...
    gluLookAt(2,0.5,2, 0,0,0, 0,1,0)
    self.meshLoader.draw()
```



[그림 6] 메시를 이동하거나 회전시키지 않고 그린 모양

이제 이 코드를 수정하여 소를 조금 아래로 내려보자.  $y$  좌표를 0.5 정도 줄이면 될 것이다. 이때 `glTranslatef(0, -0.5, 0)`을 사용한다.

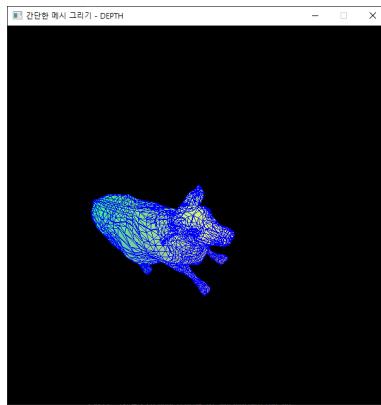
```
def paintGL(self):
    ...
    gluLookAt(2,0.5,2, 0,0,0, 0,1,0)
    glTranslatef(0, -0.5, 0)
    self.meshLoader.draw()
```



[그림 7] 메시를 아래쪽으로 이동한 결과

회전도 가능하다.  $x$  축 방향으로 45도 회전을 시키고 싶으면 다음과 같이 구현하면 된다.

```
def paintGL(self):
    ...
    gluLookAt(2,0.5,2, 0,0,0, 0,1,0)
    glTranslatef(0, -0.5, 0)
    glRotatef(45, 1, 0, 0)
    self.meshLoader.draw()
```



[그림 7] 메시를 회전시킨 결과

PyQt의 타이머를 이용하여 1000분의 1초 간격으로 타이머 이벤트처리를 수행하고, 이를 바탕으로 메시를 계속해서 회전시키는 애니메이션을 만들어 보자. 우선 MyOpenGLWidget을 다음과 같이 변경하여 self.angle 멤버 변수를 추가하고, 그림을 그릴 때마다 이 값이 증가하도록 한다. 그리고 이 증가된 값을 이용하여 메시를 회전시키기 위해 glRotatef 함수에 이 값을 넘긴다.

```
class MyGLWidget(QOpenGLWidget):

    def __init__(self, parent=None):
        super().__init__(parent)
        self.angle = 0

    def initializeGL(self):
        # OpenGL 그리기를 수행하기 전에 각종 상태값을 초기화
        glClearColor(0.0, 0.0, 0.0, 1.0)

        self.meshLoader = MeshLoader()
```

```

        self.meshLoader.loadData('cow.txt')
        glLineWidth(2)
        glEnable(GL_DEPTH_TEST)

    def resizeGL(self, width, height):
        # 카메라의 투영 특성을 여기서 설정
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.1, 1000)

    def paintGL(self):

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()

        gluLookAt(2, 0.5, 2, 0, 0, 0, 0, 1, 0)

        glTranslatef(0, -0.5, 0)
        glRotatef(self.angle, 1, 0, 0)
        self.meshLoader.draw()
        self.angle += 1.0
        glFlush()

```

애니메이션이 이루어지려면 paintGL이 계속해서 반복 호출되어야 한다. 이것은 MyWindow 클래스에 타이머를 설치하고, 이 타이머에 의한 이벤트가 발생할 때마다 OpenGL 위젯을 갱신하도록 하는 것이다. 다음과 같이 구현 가능하다.

```

class MyWindow(QMainWindow):

    def __init__(self, title=''):
        QMainWindow.__init__(self) # QMainWindow 슈퍼 클래스의 초기화
        self.setWindowTitle(title)

        self.glWidget = MyGLWidget() # OpenGL Widget
        self.setCentralWidget(self.glWidget)

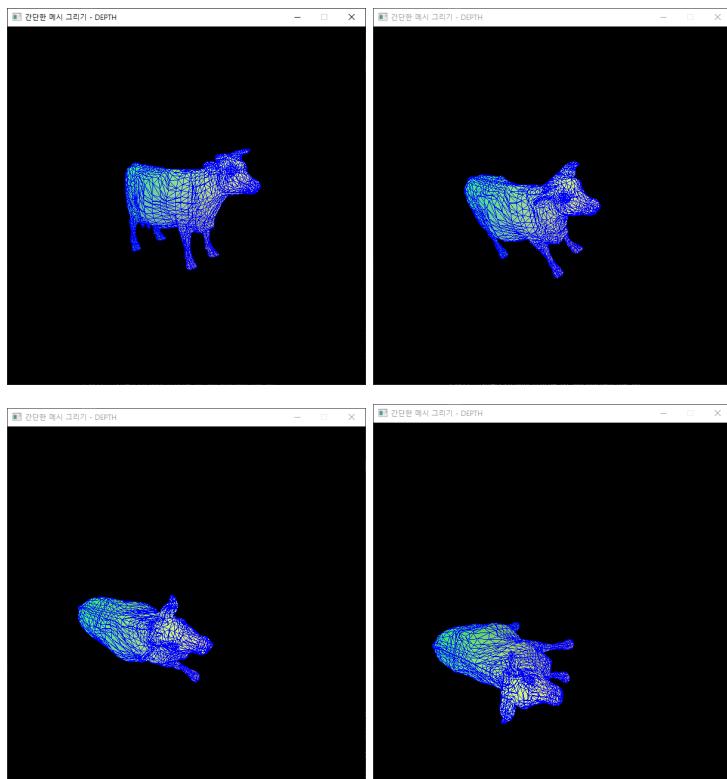
        self.timer = QTimer(self)
        self.timer.setInterval(1) # 1 밀리초
        self.timer.timeout.connect(self.timeout)
        self.timer.start()

    def timeout(self):

```

```
self.glWidget.update()
```

이제 이를 실행하면 다음과 같이 메시가 계속해서 회전하는 결과를 얻을 것이다.



[그림 8] 타이머에 의해 애니메이션이 이루어지는 결과

동작하는 전체 코드를 정리하면 다음과 같다.

#### 메시 회전 애니메이션

```
from OpenGL.GL import *
from OpenGL.GLU import *

import sys

from PyQt6.QtWidgets import QApplication, QMainWindow
from PyQt6.QtWidgets import QWidget, QVBoxLayout
from PyQt6.QtOpenGLWidgets import QOpenGLWidget
```

```

from PyQt6.QtCore import *
import math
import numpy as np

class MeshLoader:
    def __init__(self):
        self.nV = 0
        self.nF = 0
        self.vertexBuffer = None
        self.idxBuffer = None

    def loadData(self, filename) :
        with open(filename, "rt") as inputFile:

            self.nV = int(next(inputFile)) # 정점의 개수
            self.vertexBuffer = np.zeros(shape = (self.nV*3, ), dtype=float)

            for i in range(self.nV) :
                self.vertexBuffer[i*3: i*3+3] = next(inputFile).split()
                # 한 줄을 읽어 공백으로 분리 저장

            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):          # 삼각형의 개수
                idx = next(inputFile).split()
                self.idxBuffer[i*3: i*3+3] = idx[1:4]
                # 한 줄을 읽어 공백으로 분리해 첫 원소 제외하고 저장

    def draw(self):
        glColor3f(1,1,1)

        glBegin(GL_TRIANGLES)
        for i in range(self.nF):
            vIdx = self.idxBuffer[i*3: i*3+3]
            v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
            glColor3fv((self.vertexBuffer[v0*3: v0*3+3]+np.array([1]))/2)
            glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
            glColor3fv((self.vertexBuffer[v1*3: v1*3+3]+np.array([1]))/2)
            glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])

```

```

        glColor3fv((self.vertexBuffer[v2*3: v2*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
        glEnd()

        glColor3f(0,0,1)
        for i in range(self.nF):
            glBegin(GL_LINE_LOOP)
            vIdx = self.idxBuffer[i*3: i*3+3]
            v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
            glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
            glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
            glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
            glEnd()
    
```

```

class MyGLWidget(QOpenGLWidget):

    def __init__(self, parent=None):
        super().__init__(parent)
        self.angle = 0

    def initializeGL(self):
        # OpenGL 그리기를 수행하기 전에 각종 상태값을 초기화
        glClearColor(0.0, 0.0, 0.0, 1.0)

        self.meshLoader = MeshLoader()
        self.meshLoader.loadData('cow.txt')
        glLineWidth(2)
        glEnable(GL_DEPTH_TEST)

    def resizeGL(self, width, height):
        # 카메라의 투영 특성을 여기서 설정
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.1, 1000)

    def paintGL(self):

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()

        gluLookAt(2,0.5,2, 0,0,0, 0,1,0)

        glTranslatef(0, -0.5, 0)
    
```

```

glRotatef(self.angle, 1, 0, 0)
self.meshLoader.draw()
self.angle += 1.0
glFlush()

class MyWindow(QMainWindow):

    def __init__(self, title=''):
        QMainWindow.__init__(self) # QMainWindow 슈퍼 클래스의 초기화
        self.setWindowTitle(title)

        self.glWidget = MyGLWidget() # OpenGL Widget
        self.setCentralWidget(self.glWidget)

        self.timer = QTimer(self)
        self.timer.setInterval(1) # 1 밀리초
        self.timer.timeout.connect(self.timeout)
        self.timer.start()

    def timeout(self):
        self.glWidget.update()

def main(argv = []):
    app = QApplication(argv)
    window = MyWindow('간단한 메시 그리기 - DEPTH')
    window.setFixedSize(600, 600)
    window.show()
    app.exec()

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

```

glRotatef와 glTranslatef가 실제로 수행하는 일에 대한 깊이 있는 이해는 다음 장의 변환에서 살펴보도록 하자. 여기에서는 객체를 이동시키고 회전시킬 수 있다는 것만 이해하도록 하자. 이 함수들은 3차원 공간 내에서 물체의 좌표를 변경하는 모델링에 해당한다. 따라서 모델링 행렬을 변경하여 그 결과를 만들어내는 것이다. 카메라 렌즈 조작은 glMatrixMode에 GL\_PROJECTION을 사용한 뒤 행렬을 조작했는데, 이 동작들은 glMatrixMode에 GL\_MODELVIEW를 적용한 뒤 모델뷰 행렬을 수정하도록 지정한 뒤에 이루어져야 한다. 이 역시 변환을 다루는 장에서 상세히 설명할 것이다.

## 6.4 속도 개선하기

앞 장에서 다룬 속도 개선 방법을 이용하여 메시 그리기 코드의 성능도 개선할 수 있을 것이다. 우리가 살펴본 방식은 디스플레이 리스트를 사용하거나 정점 배열을 사용하는 것이었다. 이를 이용하여 빠른 렌더링이 가능한 메시 읽기 클래스를 만들어 보자. MeshLoader 클래스에 다음과 같이 원래의 draw 메소드는 그대로 두고, make\_displayList와 draw\_list라는 메소드를 추가한다. make\_displayList 메소드는 원래의 그리기 메소드인 draw를 이용하여 리스트를 만든다. draw\_list 메소드는 이렇게 만들어진 리스트를 호출하여 그리기를 실행한다.

```
def draw(self):
    glColor3f(1,1,1)
    glBegin(GL_TRIANGLES)
    for i in range(self.nF):
        vIdx = self.idxBuffer[i*3: i*3+3]
        v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
        glColor3fv((self.vertexBuffer[v0*3: v0*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
        glColor3fv((self.vertexBuffer[v1*3: v1*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
        glColor3fv((self.vertexBuffer[v2*3: v2*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
    glEnd()
    glColor3f(0,0,1)
    for i in range(self.nF):
        glBegin(GL_LINE_LOOP)
        vIdx = self.idxBuffer[i*3: i*3+3]
        v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
        glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
        glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
        glVertex3fv(self.vertexBuffer[v2*3: v2*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)
```

이제 이 새로운 메소드를 사용하면 된다. MyGLWidget의 initializeGL과 paintGL 메소드를 다음과 같이 고친다.

```
class MyGLWidget(QOpenGLWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.angle = 0

    def initializeGL(self):
        glClearColor(0.0, 1.0, 1.0, 1.0)

        self.meshLoader = MeshLoader() # 메시 로더 생성
        self.meshLoader.loadData('./skull.txt')
        self.meshLoader.make_displayList()
        glEnable(GL_DEPTH_TEST)

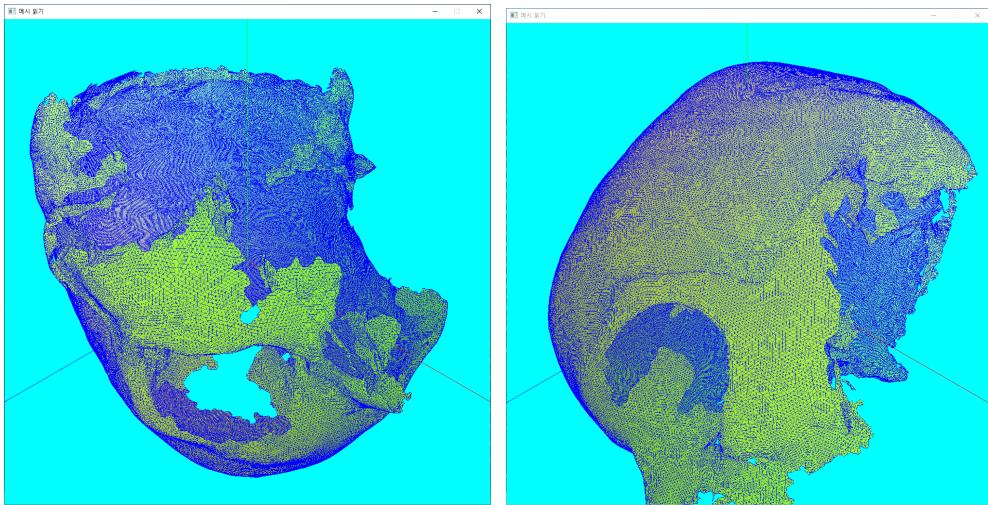
    def resizeGL(self, width, height):
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.01, 100)

    def paintGL(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        gluLookAt(0.7,0.7,0.7, 0,0,0, 0,1,0)

        drawAxes()

        glRotatef(self.angle, 0, 1, 0)
        glTranslatef(0, -0.5, 0.0)
        self.meshLoader.draw_list()
        self.angle += 5
```

이것을 실행하면 메시 그리기의 속도가 대폭 개선된 것을 확인할 수 있을 것이다. 메시 데이터 파일이 있는 곳에 skull.txt 파일이 있을 것이다. 이 파일에는 133,009개의 정점을 이용하여 248,999개의 삼각형이 담겨 있다. 이 파일을 읽어 애니메이션을 수행하여도 빠르게 애니메이션이 이루어지는 것을 확인할 수 있을 것이다.



[그림 9] 13만 개 수준의 정점과 25만개 수준의 삼각형으로 이루어진 메시의 가시화

앞 장에서 우리는 정점 배열을 준비하고 이를 glDrawArrays를 이용하여 그리거나, 정점 배열과 인덱스 배열을 준비한 뒤에 이를 glDrawElements를 이용하여 그리는 방법에 대해 살펴본 바가 있다. 이를 이용하여 빠른 렌더링을 하는 방법은 다음과 같이 구현할 수 있을 것이다.

먼저 정점 좌표를 담는 vertexBuffer와 각 정점의 색상을 담은 colorBuffer를 따로 준비한다.

```
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.colorBuffer = None
        self.list = None
```

그리고 loadData 메소드에서는 정점을 읽은 뒤에 이를 바탕으로 colorBuffer의 내용도 채운다.

```
class MeshLoader:
    ...
    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)
    self.colorBuffer = 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]
        self.colorBuffer[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.colorBuffer /= scale
    self.colorBuffer = (self.colorBuffer + np.array([1])) / 2.0

...

```

그림을 그리는 draw 메소드는 지금 다루는 주제와는 관계 없지만, 색상 정보를 담은 배열이 준비되었으므로 이렇게 고쳐둘 수 있을 것이다.

```

class MeshLoader:
    ...
    def draw(self):
        glColor3f(1,1,1)

        glBegin(GL_TRIANGLES)
        for i in range(self.nF):
            vIdx = self.idxBuffer[i*3: i*3+3]
            v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
            glColor3fv(self.colorBuffer[v0*3: v0*3+3])
            glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
            glColor3fv(self.colorBuffer[v1*3: v1*3+3])
            glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
            glColor3fv(self.colorBuffer[v2*3: v2*3+3])
            glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
        glEnd()

...

```

다음으로 우리가 관심 있는 새로운 그리기 코드는 draw\_elements라는 메소드로 구현해보자. 앞 장에서 살펴본 glDrawElements와 glDrawArrays를 이용하여 구현할 것이다. 우리는 세 개의 배열을 가지고 있다. 이것은 MeshLoader 클래스의 멤버인 vertexBuffer, colorBuffer, idxBuffer이다. 이들을 사용하기 위해서는 먼저 MyGLWidget의 OpenGL 초기화 함수인 initializeGL에서 해당 배열을 사용할 수 있도록 하고, 배열의 참조값을 GPU가 사용할 수 있도록 해야 한다. 다음과 같은 코드가 추가되면 된다.

```
class MyGLWidget(QOpenGLWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.angle = 0

    def initializeGL(self):
        glClearColor(0.0, 1.0, 1.0, 1.0)

        self.meshLoader = MeshLoader() # 메시 로더 생성
        self.meshLoader.loadData('./skull.txt')
        self.meshLoader.make_displayList()

        glEnable(GL_DEPTH_TEST) #####
        glEnableClientState(GL_VERTEX_ARRAY)
        glEnableClientState(GL_COLOR_ARRAY)
        glVertexPointer(3, GL_FLOAT, 0, self.meshLoader.vertexBuffer)
        glColorPointer(3, GL_FLOAT, 0, self.meshLoader.colorBuffer)
        glPointSize(1)
```

그러면 다시 MeshLoader로 돌아와서 draw\_elements를 다음과 같이 구현할 수 있다.

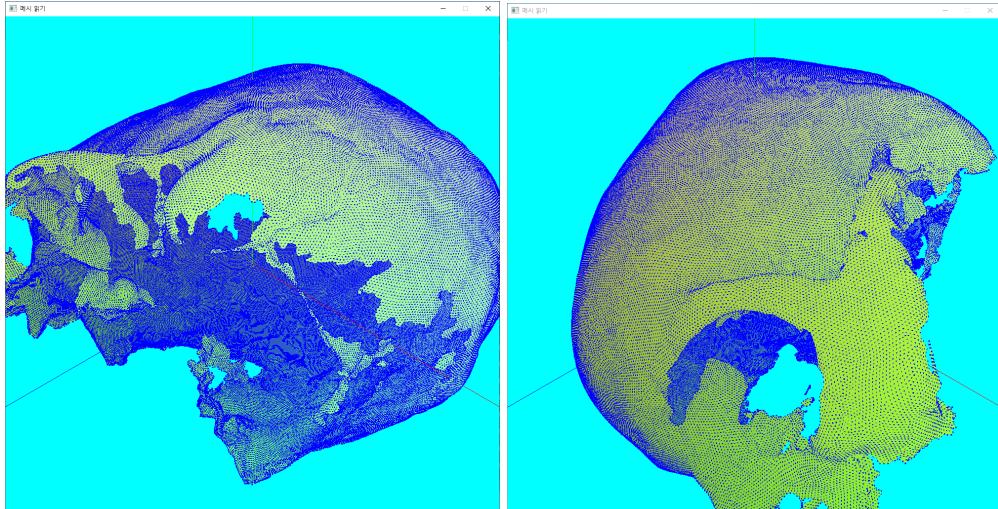
```
class MeshLoader:
    ...
    def draw_elements(self):
        glEnableClientState(GL_COLOR_ARRAY)
        glDrawElements(GL_TRIANGLES, self.nF * 3, GL_UNSIGNED_INT,
                      self.idxBuffer )

        glDisableClientState(GL_COLOR_ARRAY)
        glDrawArrays(GL_POINTS, 0, self.nV)
```

이때 면을 그릴 때는 glDrawElements를 사용하였고, 이때는 색상 배열을 사용하도록 설정하여 정점마다 다른 색이 지정되게 하였다. 그러나 점을 찍어서 표현하는 glDrawArrays 코드가 사용되기 전에는 glDisableClientState를 이용하여 색상 배열을

사용하지 않게 하였다. 따라서 모든 점이 같은 색으로 칠해지게 된다. 이렇게 구현된 코드를 사용하기 위해서는 MyGLWidget의 paintGL에서 MeshLoader의 draw\_list 메소드가 아니라 이 draw\_elements 메소드를 호출하면 된다.

```
class MyGLWidget(QOpenGLWidget):
    ...
    def paintGL(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        ...
        self.meshLoader.draw_elements()
```



[그림 10] glDrawElements와 glDrawArrays를 이용한 메시 그리기 결과

다음 코드는 디스플레이 리스트 방식과 정점 배열 방식을 비교할 수 있는 프로그램의 완성 코드이다. 이 코드를 실행하고 L 키를 누르면 리스트 모드 혹은 정점 배열 그리기 모드로 전환된다. 두 방식 모두 빠르게 렌더링을 할 수 있음을 확인할 수 있다.

#### 디스플레이 리스트와 정점 배열 방식의 비교

```
from OpenGL.GL import *
from OpenGL.GLU import *
```

```

import sys

from PyQt6.QtWidgets import QApplication, QMainWindow
from PyQt6.QtWidgets import QWidget, QHBoxLayout
from PyQt6.QtOpenGLWidgets import QOpenGLWidget
from PyQt6.QtCore import *
import math
import numpy as np

class MeshLoader:
    def __init__(self):
        self.nV = 0
        self.nF = 0
        self.vertexBuffer = None
        self.idxBuffer = None

    def loadData(self, filename) :
        with open(filename, "rt") as inputFile:

            self.nV = int(next(inputFile)) # 정점의 개수
            self.vertexBuffer = np.zeros(shape = (self.nV*3, ), dtype=float)

            for i in range(self.nV) :
                self.vertexBuffer[i*3: i*3+3] = next(inputFile).split()
                # 한 줄을 읽어 공백으로 분리 저장

            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):          # 삼각형의 개수
                idx = next(inputFile).split()
                self.idxBuffer[i*3: i*3+3] = idx[1:4]
                # 한 줄을 읽어 공백으로 분리해 첫 원소 제외하고 저장

    def draw(self):
        glColor3f(1,1,1)

        glBegin(GL_TRIANGLES)
        for i in range(self.nF):
            vIdx = self.idxBuffer[i*3: i*3+3]

```

```

        v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
        glColor3fv((self.vertexBuffer[v0*3: v0*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
        glColor3fv((self.vertexBuffer[v1*3: v1*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
        glColor3fv((self.vertexBuffer[v2*3: v2*3+3]+np.array([1]))/2)
        glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
    glEnd()

    glColor3f(0,0,1)
    for i in range(self.nF):
        glBegin(GL_LINE_LOOP)
        vIdx = self.idxBuffer[i*3: i*3+3]
        v0, v1, v2 = vIdx[0], vIdx[1], vIdx[2]
        glVertex3fv(self.vertexBuffer[v0*3: v0*3+3])
        glVertex3fv(self.vertexBuffer[v1*3: v1*3+3])
        glVertex3fv(self.vertexBuffer[v2*3: v2*3+3])
    glEnd()
}

class MyGLWidget(QOpenGLWidget):

    def __init__(self, parent=None):
        super().__init__(parent)
        self.angle = 0

    def initializeGL(self):
        # OpenGL 그리기를 수행하기 전에 각종 상태값을 초기화
        glClearColor(0.0, 0.0, 0.0, 1.0)

        self.meshLoader = MeshLoader()
        self.meshLoader.loadData('cow.txt')
        glLineWidth(2)
        glEnable(GL_DEPTH_TEST)

    def resizeGL(self, width, height):
        # 카메라의 투영 특성을 여기서 설정
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.1, 1000)

    def paintGL(self):

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_MODELVIEW)

```

```
glLoadIdentity()

gluLookAt(2,0.5,2, 0,0,0, 0,1,0)

glTranslatef(0, -0.5, 0)
glRotatef(self.angle, 1, 0, 0)
self.meshLoader.draw()
self.angle += 1.0
glFlush()

class MyWindow(QMainWindow):

    def __init__(self, title=''):
        QMainWindow.__init__(self) # QMainWindow 슈퍼 클래스의 초기화
        self.setWindowTitle(title)

        self.glWidget = MyGLWidget() # OpenGL Widget
        self.setCentralWidget(self.glWidget)

        self.timer = QTimer(self)
        self.timer.setInterval(1) # 1 밀리초
        self.timer.timeout.connect(self.timeout)
        self.timer.start()

    def timeout(self):
        self.glWidget.update()

def main(argv = []):
    app = QApplication(argv)
    window = MyWindow('간단한 메시 그리기 - DEPTH')
    window.setFixedSize(600, 600)
    window.show()
    app.exec()

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