

Chapter 04 - 동명대학교 게임공학과 3D 그래픽스 프로그래밍

카메라 모델과 투영의 기초



“젊은이, 수학에서는 뭔가 이해하는 것이 아니야. 그냥 익숙해지는 것일 뿐이지.”
“Young man, in mathematics you don't understand things. You just get used to them.”
– 폰 노이만 John Von Neumann

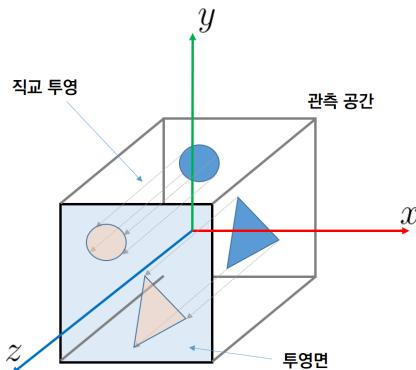
이 장에서 생각할 문제

- ❖ OpenGL의 투영 개념에 대한 이해
- ❖ 카메라의 특성을 조작하여 원하는 장면 그려내기
- ❖ 카메라 모델의 수학적 표현 방법 이해

3.1 직교 투영의 기본 개념

앞서 우리는 점들의 좌표를 OpenGL에 제공하였다. 이 좌표에 따라 기하 객체의 모양이 달라지는 것은 당연하다. 그런데 이 기하 객체가 그려질 때 화면에 어디에 그려지는지를 결정하는 방법은 무엇일까? 우리 눈에 관찰되는 공간은 좌표의 어디에서 어디까지를 담을 수 있을까?

이러한 것을 결정하는 것이 카메라 모델이다. 가장 단순한 카메라 모델은 직교 투영^{orthographic projection}이다. 직교 투영은 그림 1과 같이 관측 공간이 x , y , z 축에 정렬되어 있다고 가정할 때, 관측 공간내의 좌표를 z 축에 수직인 투영면에 옮기는 것으로 이해하면 된다. 이것은 x 와 y 축 좌표는 그대로 유지하고, z 축 값은 투영면의 z 값인 z_{proj} 으로 고치면 된다.



[그림 1] 직교 투영의 기본 개념

따라서 간단한 직교 투영은 다음과 같은 좌표 변환이라고 이해할 수 있다. 3차원 좌표 공간을 \mathbf{V} 라고 하면, 직교 투영은 다음과 같은 변환이다.

$$f : \mathbf{V} \rightarrow \mathbf{V}$$

그리고 이 변환함수는 다음과 같이 간단히 표현할 수 있을 것이다.

$$f(x, y, z) = (x, y, z_{proj})$$

이러한 변환이 이루어지고 나면 그림 1의 투영면으로 표시된 사각형 영역으로 좌표가 옮겨지며, 이 사각형 영역을 그리기 원도우에 대응시켜서 출력하면 3차원 공간을 2차원에 투영한 결과 이미지를 얻을 수 있는 것이다.

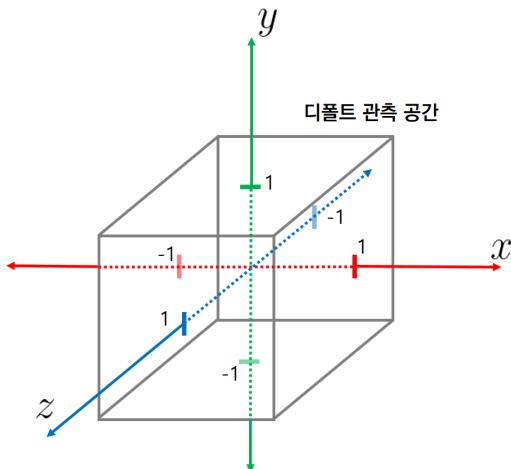
이 관측 공간을 설정하는 일은 `glOrtho` 함수를 이용한다. 관측공간은 축에 정렬된 유효체의 형태로 지정할 수 있으므로 z 축에서 관측 공간을 바라보면 왼쪽 경계를 이루는 x 좌표, 오른쪽 경계를 이루는 x 좌표, 그리고 관측공간의 아래쪽 경계를 만드는 y 좌표, 위쪽 경계를 만드는 y 좌표, 그리고 z축 위에서 바라볼 때 가까운 쪽 경계가 되는 z 좌표, 면 쪽 경계가 되는 z 좌표만 지정하면 된다. 이 각각의 값들을 left, right, bottom, top, near, far라고 하면 이 값들이 다음과 같이 `glOrtho`에 제공되면 된다.

```
glOrtho(left, right, bottom, top, near, far)
```

우리가 앞서 살펴본 코드에서 이 관측 공간을 설정하는 일을 하지 않아도 그림은 그려졌다. 이것은 아무런 설정을 하지 않아도 자동으로 적용되는 관측공간이 있다는 것이다. 자동으로 적용되는 공간은 다음과 같은 `glOrtho` 명령을 호출한 것과 같은 동작을 한다.

```
glOrtho(-1, 1, -1, 1, -1, 1)
```

이를 통해 설정되는 관측공간은 그림 2와 같다.



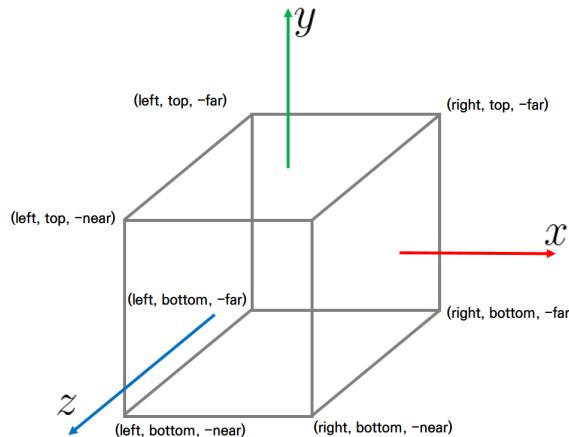
[그림 1] 기본적인 직교 투영에 의해서 상^{image}에 담기는 관측 공간

이 공간을 정규 관측 공간^{canonical viewing volume}이라고 부른다. 이 관측 공간 내에 있는 모든 좌표는 -1에서 1 사이의 값을 가지게 된다. 만약 x 좌표가 -1이라면 OpenGL 렌더링이 이루어지는 윈도우의 왼쪽 끝에 그리면 된다. x 좌표가 1이면 오른쪽 끝이 될 것이다. y 축

좌표 값이 1이면 윈도우의 상단, -1이면 하단에 배치되도록 하여 그림을 그리게 된다. z 값은 실제로 투영면의 z 값으로 바꾸지 않고 그대로 유지한다. 이것은 물체가 앞뒤로 가리는 경우 앞쪽에 있는 물체가 그려지고 뒤쪽의 물체는 그리지 않기 위해서이다.

3.2 정규 관측 공간이 아닌 경우의 좌표 변환

우리는 다양한 이유로 정규 관측 공간이 아닌 관측 공간을 설정하고 그 공간 안에 있는 객체들을 렌더링하는 작업을 한다. 이때는 glOrtho 함수를 이용하여 6개의 좌표값을 지정하면 된다. 이때 지정되는 관측공간은 그림 3과 같다.



[그림 3] 정규 관측 공간이 아닌 임의의 관측 공간

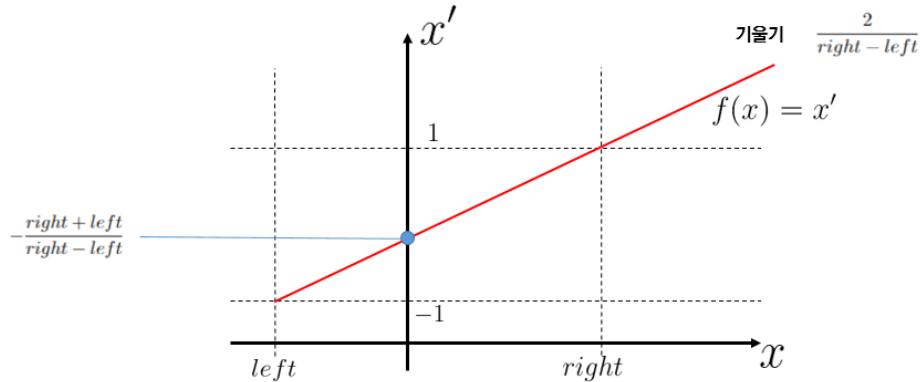
이제 관측 공간 내의 좌표는 x 는 $[\text{left}, \text{right}]$, y 는 $[\text{bottom}, \text{top}]$, 그리고 z 좌표는 $[-\text{far}, -\text{near}]$ 의 범위를 갖게 된다. 이것을 윈도우에 그대로 대응하는 방법이 있지만, OpenGL은 이것을 우선 정규 관측 공간내의 좌표로 변경하는 일을 한다. 이것은 좌표의 범위를 다음과 같이 수정하는 것이다.

$$x: [\text{left}, \text{right}] \Rightarrow x': [-1, 1]$$

$$y: [\text{bottom}, \text{top}] \Rightarrow y': [-1, 1]$$

$$z: [\text{near}, \text{far}] \Rightarrow z': [-1, 1]$$

이를 위해서 x 좌표가 실제로 어떻게 바뀌는지 생각해 보자. 그림 4와 같이 x 좌표가 $left$ 라면 x' 는 -1 로, $right$ 라면 1 로 바꾸는 1차 함수를 생각해 보자. 이 함수는 $f(x) = x'$ 로 표현할 수 있다. 이 함수의 기울기는 $\frac{2}{right - left}$ 라는 것을 쉽게 알 수 있으며, 이 함수의 절편은 $-\frac{right + left}{right - left}$ 라는 것을 약간의 수고로 계산하면 알 수 있다.



[그림 4] 관측 공간내 좌표 x 와 대응하는 정규 관측 공간 내 좌표 x' 사이의 관계

따라서 x 좌표는 다음과 같은 함수에 의해 정규 관측 공간으로 옮겨질 수 있다.

$$x' = f(x) = \left(\frac{2}{right - left} \right) x - \frac{right + left}{right - left}$$

비슷한 방식으로 y 좌표도 변경할 수 있다.

$$y' = \left(\frac{2}{top - bottom} \right) y - \frac{top + bottom}{top - bottom}$$

z 좌표는 부호가 바뀐다는 점을 고려하면 다음과 같이 변경된다.

$$z' = - \left(\frac{2}{far - near} \right) z - \frac{far + near}{far - near}$$

이것을 행렬과 벡터의 곱으로 표현하면 다음과 같은 선형 변환으로 이해할 수 있다.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

3차원 좌표를 표현하는 벡터가 4 개의 원소를 가지고 있는데, 이는 동차 좌표계라는 것으로 나중에 다시 깊이 다룰 것이다. 여기서는 좌표 뒤에 1이 추가되어 있다고만 생각하자. 원래의 좌표에 행렬이 곱해져서 새로운 좌표가 된다. 이 새로운 좌표는 관측 공간 내의 좌표를 정규 관측 공간으로 옮겼을 때에 얻게 되는 좌표이다. 따라서 정규 직교 투영이라는 것은 이 행렬을 곱하는 것을 의미한다. 이렇게 좌표를 변환하고 나면 어떤 좌표는 -1과 1의 범위를 벗어날 것이다. 이렇게 벗어나는 좌표들은 그리기 대상에서 제외된다.

`glOrtho(-1, 1, -1, 1, 1, -1)`을 설정했다고 가정하면 이 행렬이 어떻게 되는지 살펴 보라. 다음과 같은 행렬을 얻을 것이다.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

이것은 항등행렬^{identity matrix}이다. 이제 앞에서 다루었던 코드를 생각해 보자. OpenGL 위짓은 세 가지 콜백이 구현되어야 하는데, 윈도우의 크기가 변경될 때 호출되는 `resizeGL`이 있었다. 여기에 사용되는 코드로 다음과 같은 일을 한 기억이 날 것이다.

```
glMatrixMode(GL_PROJECTION)
glLoadIdentity()
```

투영을 할 때 사용되는 행렬이 투영 행렬^{projection matrix}이다. OpenGL에서 객체들이 가진 모든 좌표는 화면에 그려지기 전에 이 투영 행렬이 곱해진다. 이것은 정규 관측 공간으로 좌표를 변환하는 것이다. 이 투영행렬을 바꾸면 화면에 그려지는 내용이 달라지게 된다. 투영 행렬을 변경하고 싶으면 어떤 행렬을 조작할 것인지를 결정하는 `glMatrixMode`를 호출하여 투영 행렬 변경 상태로 만들어야 한다. 이때 이 함수에 인자로 `GL_PROJECTION`이라는 값을 넘기면 된다. 이후에 행렬 변경 명령을 제공하면 투영 행렬이 변경된다. 우리는 `glLoadIdentity`라는 함수를 호출하였는데, 이것은 현재의 행렬을 항등행렬로 변경하라는 의미이다. 즉 `glOrtho(-1, 1, -1, 1, 1, -1)`을 호출한 것과 같은

효과가 된다. 이것은 정규 관측 공간이 z 축 방향으로 뒤집어진 것과 같다. 2차원 화면에 그려질 때에 그 차이를 알 수 없을 것이다.

우리가 원하는 투영 행렬을 만들고 싶으면 행렬을 바로 지정하거나, 행렬 조작 함수를 사용할 수 있다. 대표적인 투영 행렬 조작 함수가 지금까지 살펴본 glOrtho 함수이다. 이제 이것을 이용하여 직교 투영의 결과가 달라지도록 해 보자.

3.3 직교 투영 제어하기

지금부터 구현할 프로그램은 하나의 윈도우 내에 두 개의 OpenGL 위짓을 설치할 것이다. 이 중에 하나는 우리가 제어하는 glOrtho의 인자에 따라 투영된 좌표를 그려내는 것이고, 다른 하나는 glOrtho에 의해 변경된 관측 공간을 관찰하는 것이 목적이다.

우선 그림을 그리기 위해 몇 가지 함수들을 정의하자. 다음 함수는 축을 그리는 함수이다.

```
def drawAxes() :  
    glBegin(GL_LINES)  
    glColor3f(1,0,0)  
    glVertex3f(0,0,0)  
    glVertex3f(1,0,0)  
    glColor3f(0,1,0)  
    glVertex3f(0,0,0)  
    glVertex3f(0,1,0)  
    glColor3f(0,0,1)  
    glVertex3f(0,0,0)  
    glVertex3f(0,0,1)  
    glEnd()
```

다음으로 구현할 함수는 glOrtho에 사용되는 6개의 값을 이용하여 육면체인 관측 공간의 테두리를 그리는 함수이다.

```
def drawBox(l, r, b, t, n, f): # glOrtho가 만드는 공간(육면체)을 가시화  
    glColor3f(1, 1, 1)  
    glBegin(GL_LINE_LOOP)  
    # 앞면
```

```

glVertex3f(l,t,-n); glVertex3f(l,b,-n);
glVertex3f(r,b,-n); glVertex3f(r,t,-n)
glEnd()

glBegin(GL_LINE_LOOP)
# 뒷면
glVertex3f(l,t,-f); glVertex3f(l,b,-f);
glVertex3f(r,b,-f); glVertex3f(r,t,-f)
glEnd()

glBegin(GL_LINES)
glVertex3f(l,t,-n); glVertex3f(l,t,-f)
glVertex3f(l,b,-n); glVertex3f(l,b,-f)
glVertex3f(r,b,-n); glVertex3f(r,b,-f)
glVertex3f(r,t,-n); glVertex3f(r,t,-f)
glEnd()

```

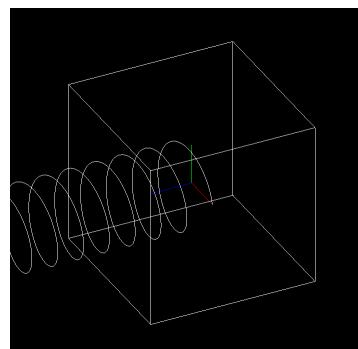
그리고 다음으로 구현한 함수는 z 축 방향으로 감겨있는 코일 모양을 객체를 그리는 함수이다.

```

def drawHelix():
    glBegin(GL_LINE_STRIP)
    for i in range(1000):
        z = i/10
        x, y = 0.5*math.cos(z), 0.5*math.sin(z)
        glVertex3f(x, y, z/100)
    glEnd()

```

이러한 함수들을 호출하여 그림을 그리면 다음과 같은 장면이 그려질 것이다.



[그림 5] 그리기의 대상이 되는 객체들

윈도우는 QMainWindow를 상속받은 클래스 MyWindow를 선언하고, 여기에 두 개의 OpenGL 위젯을 설치할 것이다. 다음과 같은 방법으로 구현할 수 있을 것이다. 코드를 보면 레이아웃을 하나 설정하고, 여기에 두 개의 OpenGL 위젯인 self.glWidget1과 self.glWidget2를 배치하였다. 각각의 위젯을 생성할 때 생성자에 인자를 제공하여 다른 특성을 갖도록 하는데, observation이라는 인자를 하나는 False, 다른 하나는 True로 설정하였다. 이것에 따라 두 위젯의 동작이 다르게 이루어질 것이다.

```
class MyWindow(QMainWindow):

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

        ### GUI 설정

        central_widget = QWidget()
        self.setCentralWidget(central_widget)

        gui_layout = QHBoxLayout() # CentralWidget의 수평 나열 레이아웃

        # 배치될 것들 - 정점 입력을 받기 위한 위젯
        central_widget.setLayout(gui_layout)

        ##### OpenGL Widget 추가
        self.glWidget1 = MyGLWidget(observation=False) # OpenGL Widget
        self.glWidget2 = MyGLWidget(observation=True)
        gui_layout.addWidget(self.glWidget1)
        gui_layout.addWidget(self.glWidget2)
#####
```

OpenGL 위젯은 QOpenGLWidget을 상속받은 MyGLWidget 클래스로 구현할 것이다. 이 클래스의 생성자에 넘겨지는 인자로 observation이라는 인자가 있는데, 이것이 False이면 glOrtho로 설정한 투영이 적용되어 렌더링이 될 것이고, True이면 관측 공간 자체를 관찰할 수 있도록 구현될 것이다. 이 클래스의 생성자는 다음과 같이 구현할 수 있다. 멤버 변수로 self.observation이 있으며, 이 값은 인자로 넘어온 값이 전달된다. 그리고 glOrtho에 적용된 left, right, bottom, top, near, far의 값 역시 멤버 변수로 관리한다. 그리고 생성자에서 이 값을 설정하였다.

```

class MyGLWidget(QOpenGLWidget):

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

        self.left = self.bottom = self.near = -1
        self.right = self.top = self.far = 1

```

OpenGL 위젯에서 중요한 세 가지 콜백 함수 중 초기화 함수는 간단히 버퍼를 지우는 색을
검정으로 지정하는 일만 하자.

```

def initializeGL(self):
    # OpenGL 그리기를 수행하기 전에 각종 상태값을 초기화
    glClearColor(0.0, 0.0, 0.0, 1.0)
    if self.observation:
        gluLookAt(1.2, 1.5, 1, 0, 0, 0, 0, 1, 0)

```

화면 크기가 변경되었을 때, 혹은 화면이 처음으로 그려질 때 호출되는 resizeGL에는 우리가 관리하고 있는 관측 공간의 크기 값을 이용하여 glOrtho를 호출하는 작업을 수행한다. 이것은 관측 공간 관찰 모드가 아닌 위젯에서만 그러하다. 관측 공간을 관찰하는 위젯, 즉 self.observation이 True인 위젯은 고정된 관측공간을 갖도록 할 것이다. 가장 먼저 실행되는 코드는 수정되는 행렬이 투영 행렬이 되도록 glMatrixMode를 호출하여 GL_PROJECTION을 지정한다. 그리고 행렬을 초기화하기 위해 glLoadIdentity를 호출한다. 다음으로 우리가 원하는 투영을 적용하기 위해 glOrtho를 호출하고 있다.

```

def resizeGL(self, width, height):
    # 카메라의 투영 특성을 여기서 설정
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    if not self.observation:
        glOrtho(
            self.left, self.right,
            self.bottom, self.top,
            self.near, self.far)
    else :
        glOrtho(
            -2, 2, -2, 2, -100, 100)

```

이제 그리기 이벤트를 처리하는 paintGL을 구현해 보자. 여기에서는 조작되는 행렬이 투영 행렬이 아니라 모델링과 카메라의 위치 등을 제어하는 행렬이라는 것을 명시하기 위해 glMatrixMode에 GL_MODELVIEW를 지정한다. 모델뷰 행렬을 조작하는 것에 대해서는 나중에 깊이 다룰 것이다. 여기서도 우선 glLoadIdentity 함수를 통해 모델뷰 행렬을 초기화한다. 다음으로 관측 공간을 관찰하는 self.observation이 True인 위짓이라면 카메라의 위치를 옮겨서 관측 공간을 볼 수 있도록 gluLookAt을 호출한다. gluLookAt도 다시 다루겠지만, 여기서는 앞의 세 개의 숫자가 카메라가 놓인 위치, 다음 세 숫자가 카메라가 쳐다보는 지점, 그리고 마지막 세 숫자는 카메라의 자세를 알기 위해 필요한 카메라 위쪽 방향 벡터를 의미한다. 이 값을 지정함으로써 관찰용 위짓은 관측 공간 밖으로 나가서 관측 공간을 볼 수 있게 된다. 그리고 앞서 구현했던 각종 그리기 함수를 호출하여 그림을 그린다.

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

    if self.observation:
        gluLookAt(1.2, 1.5, 1, 0, 0, 0, 0, 1, 0)

    drawHelix()
    drawAxes()
    drawBox(
        self.left, self.right,
        self.bottom, self.top,
        self.near, self.far)

    # 그려진 프레임버퍼를 화면으로 송출
    glFlush()
```

이러한 내용을 종합하여 다음과 같이 동작하는 코드를 작성할 수 있다.

glOrtho를 이용한 투영 제어 실험

```
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

def drawAxes():
    glBegin(GL_LINES)
    glColor3f(1,0,0) # red x axis
    glVertex3f(0,0,0); glVertex3f(1,0,0)
    glColor3f(0,1,0) # green y axis
    glVertex3f(0,0,0); glVertex3f(0,1,0)
    glColor3f(0,0,1) # blue z axis
    glVertex3f(0,0,0); glVertex3f(0,0,1)
    glEnd()

def drawHelix():
    glColor3f(1,1,1)
    glBegin(GL_LINE_STRIP)
    for i in range(1000):
        angle = i/10
        x, y = math.cos(angle), math.sin(angle)
        glVertex3f(x, y, angle/10)
    glEnd()

def drawBox(l, r, b, t, n, f): # glOrtho가 만드는 공간(육면체)을 가시화
    glColor3f(1, 1, 1)
    glBegin(GL_LINE_LOOP)
    # 앞면
    glVertex3f(l,t,-n); glVertex3f(l,b,-n); glVertex3f(r,b,-n);
    glVertex3f(r,t,-n)
    glEnd()

    glBegin(GL_LINE_LOOP)
    # 뒷면
    glVertex3f(l,t,-f); glVertex3f(l,b,-f); glVertex3f(r,b,-f);
    glVertex3f(r,t,-f)
    glEnd()

    glBegin(GL_LINES)
    glVertex3f(l,t,-n); glVertex3f(l,t,-f)
    glVertex3f(l,b,-n); glVertex3f(l,b,-f)
    glVertex3f(r,b,-n); glVertex3f(r,b,-f)
    glVertex3f(r,t,-n); glVertex3f(r,t,-f)
    glEnd()

class MyGLWidget(QOpenGLWidget):

```

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

    self.left = self.bottom = self.near = -2
    self.right = self.top = self.far = 2

def initializeGL(self):
    pass

def resizeGL(self, w, h):
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    if self.observation:
        glOrtho(-4, 4, -4, 4, -100, 100)
    else:
        glOrtho(self.left, self.right, self.bottom, self.top,
                self.near, self.far)

def paintGL(self):

    self.projection_update()

    glClear(GL_COLOR_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    if self.observation:
        gluLookAt(
            1, 0.7, 0.5, # 눈의 위치
            0, 0, 0, # 점다보는 목표 지점 위치
            0, 1, 0 # 카메라의 상향 벡터
        )

        drawAxes()
        drawHelix()
        drawBox(self.left, self.right, self.bottom, self.top, self.near,
                self.far)

    def projection_update(self):
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        if self.observation:
            glOrtho(-4, 4, -4, 4, -100, 100)
        else:
            glOrtho(self.left, self.right, self.bottom, self.top,
                    self.near, self.far)
```

```
class MyWindow(QMainWindow):
    def __init__(self, title=''):
        super().__init__()
        self.setWindowTitle(title)

        ## GUI 설정
        central_widget = QWidget()
        self.setCentralWidget(central_widget)
        gui_layout = QHBoxLayout()

        central_widget.setLayout(gui_layout)

        self.glWidget1 = MyGLWidget()
        self.glWidget2 = MyGLWidget(observation = True) # 관측용 OpenGL 위젯

        gui_layout.addWidget(self.glWidget1)
        gui_layout.addWidget(self.glWidget2)

    def keyPressEvent(self, e):
        if e.key() == Qt.Key.Key_A:
            self.glWidget1.left -= 0.1
            self.glWidget2.left -= 0.1
        elif e.key() == Qt.Key.Key_S:
            self.glWidget1.left += 0.1
            self.glWidget2.left += 0.1
        elif e.key() == Qt.Key.Key_D:
            self.glWidget1.right -= 0.1
            self.glWidget2.right -= 0.1
        elif e.key() == Qt.Key.Key_F:
            self.glWidget1.right += 0.1
            self.glWidget2.right += 0.1
        elif e.key() == Qt.Key.Key_Q:
            self.glWidget1.top += 0.1
            self.glWidget2.top += 0.1
        elif e.key() == Qt.Key.Key_W:
            self.glWidget1.top -= 0.1
            self.glWidget2.top -= 0.1
        elif e.key() == Qt.Key.Key_Z:
            self.glWidget1.near += 0.1
            self.glWidget2.near += 0.1
        elif e.key() == Qt.Key.Key_X:
            self.glWidget1.near -= 0.1
            self.glWidget2.near -= 0.1
        elif e.key() == Qt.Key.Key_V:
            self.glWidget1.far += 0.1
            self.glWidget2.far += 0.1
        elif e.key() == Qt.Key.Key_C:
```

```

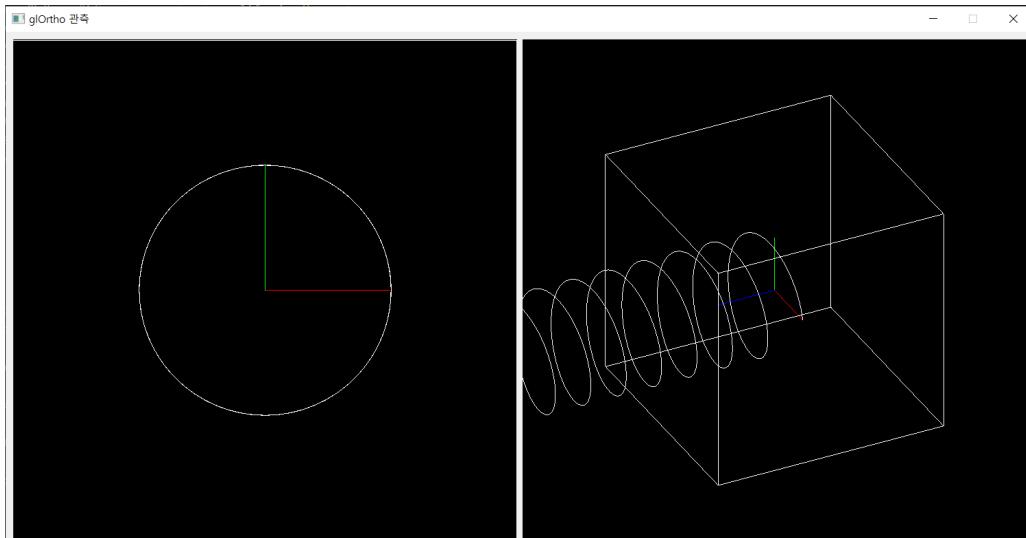
        self.glWidget1.far -= 0.1
        self.glWidget2.far -= 0.1

    self.glWidget1.update()
    self.glWidget2.update()

def main(argv = sys.argv):
    app = QApplication(argv)
    window = MyWindow('glOrtho 관측')
    window.setFixedSize(1200, 600)
    window.show()
    app.exec()

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

```



[그림 6] glOrtho로 설정한 관측 공간에 따른 렌더링 결과(좌)와 관측공간 관찰 결과(우)

키보드를 이용하여 glOrtho에 적용되는 파라미터를 변경할 수 있도록 해보자. MyGLWidget 클래스에 projection_update()라는 메소드를 하나 만들어 두자. 현재 설정된 관측 공간의 크기에 따라 glOrtho 함수를 부르는 역할을 수행한다.

```

def projection_update(self) :
    # 카메라의 투영 특성을 여기서 설정
    glMatrixMode(GL_PROJECTION)

```

```

glLoadIdentity()
if not self.observation:
    glOrtho(
        self.left, self.right,
        self.bottom, self.top,
        self.near, self.far)
else :
    glOrtho(
        -2, 2, -2, 2, -100, 100)

```

paintGL 함수는 이 투영행렬 설정을 먼저 하고 그리기를 수행하도록 변경해 보자.

```

def paintGL(self):

    self.projection_update()
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    ...

```

MyWindow 클래스에서 다음과 같이 키보드 처리 함수를 구현하여 OpenGL 위젯이 관리하는 관측공간의 크기를 변경하도록 하자.

```

def keyPressEvent(self, e):
    if e.key() == Qt.Key.Key_A:
        self.glWidget1.left -= 0.1
        self.glWidget2.left -= 0.1
    elif e.key() == Qt.Key.Key_S:
        self.glWidget1.left += 0.1
        self.glWidget2.left += 0.1
    elif e.key() == Qt.Key.Key_D:
        self.glWidget1.right -= 0.1
        self.glWidget2.right -= 0.1
    elif e.key() == Qt.Key.Key_F:
        self.glWidget1.right += 0.1
        self.glWidget2.right += 0.1
    elif e.key() == Qt.Key.Key_Q:
        self.glWidget1.top += 0.1
        self.glWidget2.top += 0.1
    elif e.key() == Qt.Key.Key_W:
        self.glWidget1.top -= 0.1
        self.glWidget2.top -= 0.1
    elif e.key() == Qt.Key.Key_Z:

```

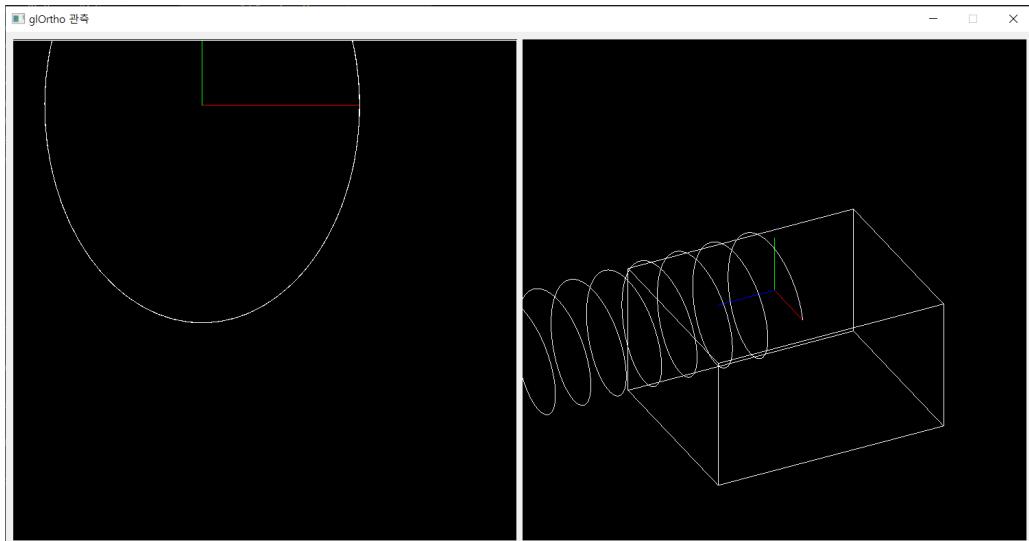
```

    self.glWidget1.near += 0.1
    self.glWidget2.near += 0.1
elif e.key() == Qt.Key.Key_X:
    self.glWidget1.near -= 0.1
    self.glWidget2.near -= 0.1
elif e.key() == Qt.Key.Key_V:
    self.glWidget1.far += 0.1
    self.glWidget2.far += 0.1
elif e.key() == Qt.Key.Key_C:
    self.glWidget1.far -= 0.1
    self.glWidget2.far -= 0.1

self.glWidget1.update()
self.glWidget2.update()

```

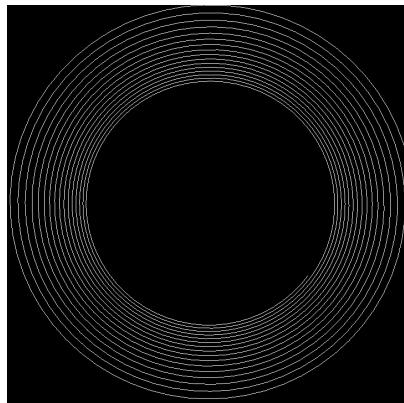
이제 키보드를 a, d, s, w 키를 눌러서 관측 공간의 크기를 제어할 수 있을 것이다. 이 관측 공간에 따라 최종 렌더링이 어떻게 이루어지는지는 두 개의 OpenGL 위젯 가운데 왼쪽에서 확인할 수 있다.



[그림 7] 직교 투영의 관측 공간을 제어하여 얻는 렌더링 결과(좌)와 변형된 관측 공간(우)

이렇게 우리는 직교 투영방법에 대해 이해할 수 있다. 그런데, 이러한 투영 방식은 실제로 우리 눈이 관찰하는 모습과는 다른 결과를 나타낸다. 우리 눈에는 멀리 있는 물체는 작게, 눈에 가까운 물체는 크게 보인다. 직교 투영 기법은 극단적인 망원 렌즈라고 할 수 있는데, 아무리 멀리 있는 물체라도 그 크기가 작게 나타나지 않는 것이다. 이것은 현실에서 있을 수

없는 관찰 방법이다. 실제 우리 눈에는 먼 물체가 작게 보이기 때문에 나선형 모양으로 감겨 있는 코일은 그림 8의 왼쪽 위짓에 나타난 것과 같이 보여야 할 것이다.

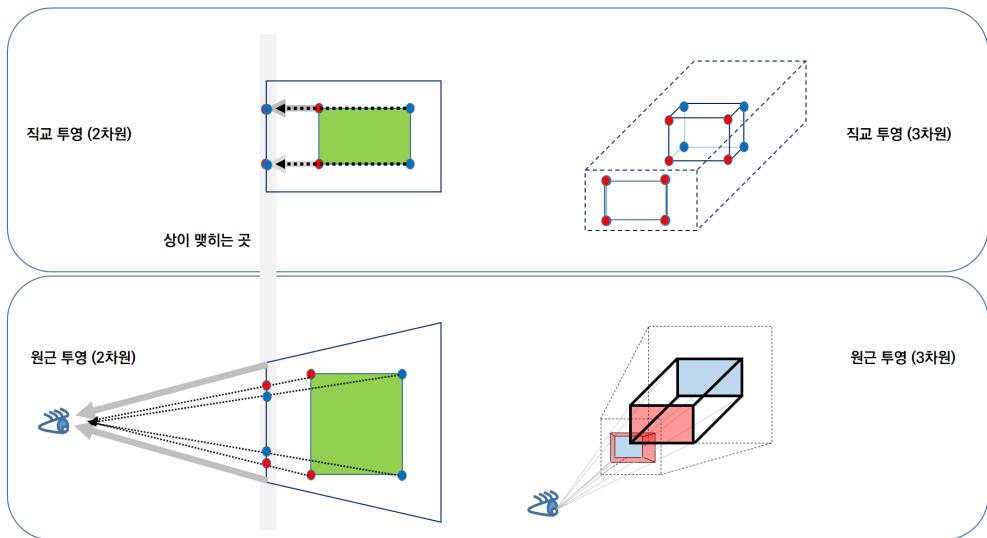


[그림 8] 우리 눈이 관측하는 원근감이 존재하는 이미지에 담겨야 할 결과

3.4 원근이 파악되는 투영

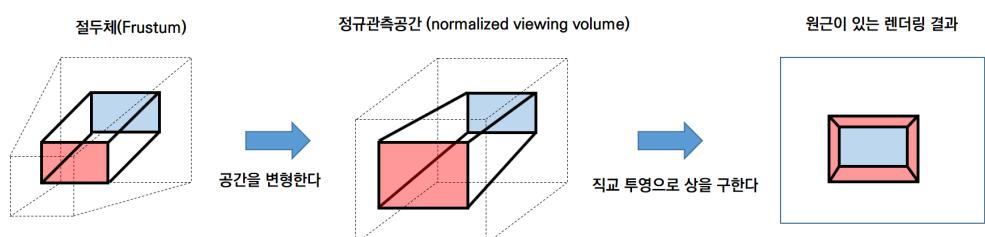
실제 카메라나 우리 눈으로 세계를 관찰하면 가까운 물체는 크게, 멀리 떨어진 물체는 작게 보인다. 그 이유는 눈으로 감지하는 빛은 눈이라는 하나의 관측 지점으로 모이면서 상image를 형성하게 되며, 이렇게 모이는 선은 앞에서 살펴본 직교 투영과 달리 서로 평행하지 않기 때문이다. 이러한 원근 투영을 설정하는 방법은 glFrustum과 gluPerspective가 있다. 직교 투영과 원근 투영의 차이는 그림 9와 같이 설명할 수 있다. 그림에서 확인할 수 있는 것처럼 직교 투영은 좌표가 상이 맷히는 평면에 수직으로 투영되며, 이 투영선은 모든 점에 대해 평행하다. 왼쪽에 있는 2차원 그림으로 이해해 보자. 직교 투영에서는 상이 맷히는 평면에서 가까이 있는 빨간 두 점과, 멀리 있는 파란색 두 점이 상이 맷힐 때에 똑같은 간격을 가진채로 좌표가 결정된다. 그런데 아래쪽 원근 투영을 살펴보면 빨간색 두 점과 파란색 두 점이 모두 관찰자의 눈에 모이도록 투영이 이루어지는데, 가까운 거리의 빨간색 두 점은 조금 덜 모인 상태에서 상이 맷하고, 멀리 있는 파란색 두 점은 더 많이 모인 뒤에 상이 맷혀 그 간격이 더 좁아지게 된다.

3차원 공간으로 확장해 보면, 직교 투영은 우리가 이미 살펴본 직육면체 모양의 관측 공간이 된다. 그런데 원근이 구별되는 투영을 위해서는 오른쪽 아래와 같이 눈과 가까운 쪽은 좁고 멀리 있는 면을 가지는 육면체가 되어야 한다. 이를 절두체Frustum이라고 한다. 직교 투영에서



따라서 원근이 있는 투영을 위해서는 직교 투영에서 `glOrtho`를 이용하여 관측 공간의 모양을 설정하듯이 원근 투영에 필요한 이 절두체의 모양을 지정하면 된다. 이를 가능하게 하는 함수가 `glFrustum`이다.

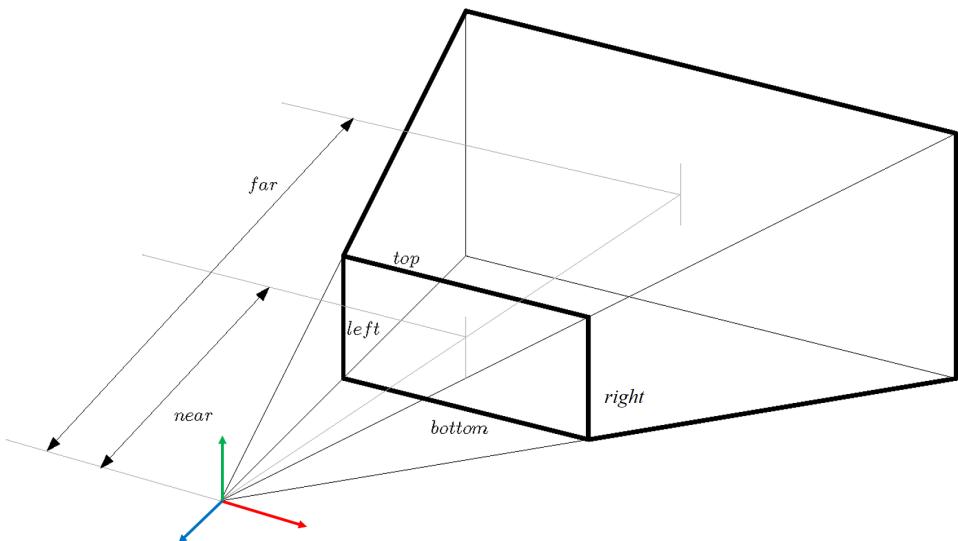
개념적으로 절두체 내에 있는 객체에 대해 투영선을 따라 상이 맷히는 곳까지 가서 좌표를 구하면 된다. 그런데 실제로 원근이 있는 투영 좌표의 계산은 그림 10과 같은 방법으로 이루어진다. 이것은 절두체 내의 객체가 가진 좌표를 정규 관측 공간 내의 좌표로 변환하는 것을 의미한다. 즉 절두체의 모양을 정규 관측 공간으로 변형하는 것이다. 그러면 이 변형에 따라 객체의 좌표가 변할 것이다. 절두체는 시점에서 가까운 곳의 면이 더 좁기 때문에 정규 관측 공간으로 변형하면 앞쪽의 공간이 더 크게 확장되는 것과 같다. 이렇게 변화를 일으킨 뒤에는 앞에서 살펴본 직교 투영을 정규 관측 공간에서 수행하면 되는 것이다. 그러면 최종적으로 앞쪽 객체는 커지고 멀리 있는 객체는 작아지는 투영이 이루어지게 된다.



[그림 10] 절두체 내의 객체가 렌더링되는 과정

절두체를 지정하는 방법은 `glFrustum`을 사용한다. 이 함수는 `glOrtho`와 동일한 파라미터를 가지는데, 다음과 같이 파라미터들을 입력 받고, 그림 11과 같은 절두체를 생성해 투영이 이루어지게 한다. 이때 투영선이 집중되는 한 점, 즉 관측자의 눈이 놓이는 위치는 좌표 $(0,0,0)$ 이 된다.

```
glFrustum(left, right, bottom, top, near, far)
```



[그림 11] `glFrustum`을 이용한 절두체의 생성

`glFrustum`은 원근 투영 행렬을 하나 만들어 내고, 이 행렬을 현재의 투영 행렬에 곱하는 작업을 수행한다. OpenGL은 이 곱의 결과로 얻어지는 행렬을 투영 행렬을 사용하여 투영을 실시한다. 따라서 일반적으로 사용되는 방법은 다음과 같다.

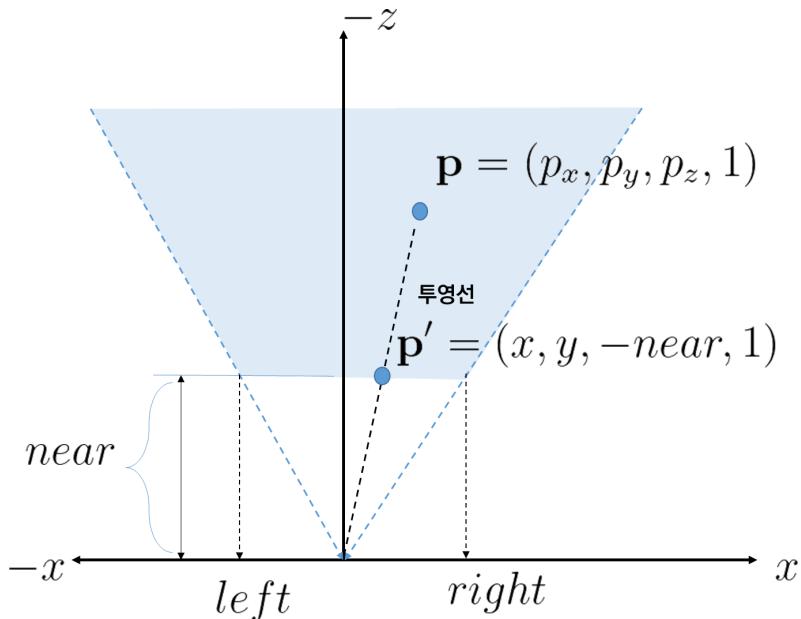
```
glMatrixMode(GL_PROJECTION)    # 투영 행렬 설정 모드로 지정  
glLoadIdentity();               # 투영 행렬을 항등행렬 I로 지정  
glFrustum(l, r, b, t, n, f)    # 현재 투영 행렬에 새로운 행렬을 곱한다.
```

위의 코드는 투영 행렬을 우선 항등 행렬 I 로 만든 뒤에 `glFrustum`이 표현하는 행렬 M 을 곱하는 것이다. 만약 `glLoadIdentity`를 빼뜨리면 어떻게 될까? 이 부분은 윈도우의

크기가 변경될 때마다 호출되는 `resizeGL`이나, 그리기 이벤트가 발생하면 호출되는 `paintGL` 등에 담기게 될 것인데, 그때 마다 \mathbf{M} 을 투영 행렬에 곱할 것이다. 따라서 호출되는 횟수가 n 이면 투영 행렬은 \mathbf{M}^n 이 되어 우리가 원하는 행렬이 되지 않을 것이다. 따라서 매번 항등 행렬을 불러 초기화하고, 행렬 \mathbf{M} 을 곱하는 형식을 취한다.

3.5 glFrustum에 의해 얻어지는 엎근 투영 행렬 구하기

그림 10을 통해 `glFrustum`을 통해 얻어지는 행렬은 지정된 좌표로 결정되는 절두체를 모든 축 방향으로 $[-1, 1]$ 의 범위를 차지하는 정규 관측 공간으로 변형하는 것과 같다는 것을 알 수 있다. 이러한 원리에 따라 `glFrustum`에 의해 결정되는 투영행렬을 유도해 보자.



[그림 12] 절두체 내의 좌표 \mathbf{p} 가 상을 맺는 위치 \mathbf{p}'

우선 그림 12에서 절두체 안에 있는 좌표 \mathbf{p} 가 상을 맺는 위치 \mathbf{p}' 를 계산해 보자. 닮은 삼각형의 사이의 일치하는 비례 관계를 이용하여 $p_x : p_z = x : -near$ 라는 것을 쉽게 알 수 있다. 따라서 다음과 같은 결과를 얻을 수 있다.

$$x = -near \frac{p_x}{p_z}$$

이렇게 해서 얻은 좌표 x 는 정규 장치 좌표계로 옮겨져야 한다. 즉 $[-1, 1]$ 의 범위로 변경된다. 만약 x 가 left이면 -1 , right이면 1 이 되게 하면 된다 이것은 직교 투영의 행렬을 유도할 때 다루어 본 문제이고, 다음과 같이 변환하면 된다.

$$x' = \left(-\frac{2near}{right - left} \right) \frac{p_x}{p_z} - \frac{right + left}{right - left}$$

비슷한 방법으로 y 좌표도 변환할 수 있다.

$$y' = \left(-\frac{2near}{top - bottom} \right) \frac{p_y}{p_z} - \frac{top + bottom}{top - bottom}$$

그럼데 z 좌표의 투영은 다소 고민을 해야 한다. x 와 y 좌표의 변화를 보면 원래의 좌표 p_x 와 p_y 를 p_z 로 나누는 작업을 포함하고 있다. 이러한 변환 방식과 동일하게 p_z 가 z' 로 변환되게 만들기 위해 다음과 같이 미지수 α 와 β 를 도입하여 z' 를 표현해 보자.

$$z' = \frac{\alpha}{p_z} + \beta$$

이 식은 p_z 가 near인 경우에는 -1 , far인 경우에는 1 의 값을 가져야 한다. 따라서 미지수를 구하기 위해 이 값들을 넣어 두 개의 식을 얻어 보자.

$$-1 = \frac{\alpha}{near} + \beta$$

$$1 = \frac{\alpha}{far} + \beta$$

이 이원일차연립방정식을 풀면 쉽게 다음과 같은 α 와 β 를 구할 수 있다.

$$\alpha = \frac{near \cdot far}{far - near}, \quad \beta = \frac{far + near}{far - near}$$

따라서 z' 는 다음과 같다.

$$z' = \frac{near \cdot far}{(far - near)p_z} + \frac{far + near}{far - near}$$

지금까지 동차좌표는 $(x, y, z, 1)$ 과 같이 마지막 좌표가 1이 되는 것만을 사용했지만, 이 값은 임의의 값 w 가 될 수 있다. 이때 w 가 1이 아니라면 다음과 같은 방식으로 마지막 좌표가 1인 좌표로 바꿀 수 있다. 이 두 좌표는 동일하다.

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$$

즉, w 로 모든 값을 나눈 것과 같다. 이러한 동차좌표에 대한 이해는 변환을 다루는 장으로 조금 미루고, 이 특징만을 기억하고 이용하여 다음과 같은 관계식을 구해 보자.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} -x' \cdot p_z \\ -y' \cdot p_z \\ -z' \cdot p_z \\ -p_z \end{pmatrix}$$

앞에서 구한 좌표 변환을 이용하여 이것을 다음과 같이 표현할 수 있다.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} -x' \cdot p_z \\ -y' \cdot p_z \\ -z' \cdot p_z \\ -p_z \end{pmatrix} = \begin{pmatrix} \frac{2near}{right-left}p_x + \frac{right+left}{right-left}p_z \\ \frac{2near}{top-bottom}p_y + \frac{top+bottom}{top-bottom}p_z \\ -\frac{far+near}{right-left}p_z - \frac{2near \cdot far}{far - near} \\ -p_z \end{pmatrix}$$

이것은 결국 다음과 같이 좌표 \mathbf{P} 에 변환 행렬 \mathbf{M} 을 곱하는 것으로 표현할 수 있다. 이때 이 변환 행렬이 바로 원근 투영 행렬이다.

$$\mathbf{M} = \begin{pmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far - near} & \frac{-2near \cdot far}{far - near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

3.6 절두체 관찰과 렌더링 결과 확인하기

glFrustum이 관찰하는 공간과 그 결과를 확인하기 위한 코드를 작성해 보자. 이 코드는 앞서 살펴보았던 glOrtho에 의한 관측공간을 확인하는 코드와 비슷할 것이다. 우선 절두체 공간을 그려내는 함수를 구현해 보자. glFrustum(l, r, b, t, n, f)가 호출되면, 이를 그리기 위해 drawFrustum에도 동일한 파라미터를 제공한다. 관측공간에서 앞쪽 가까운 면^{near plane}의 왼쪽, 오른쪽 끝 x 좌표와 아래쪽과 위쪽 y 좌표는 각각 l, r, b, t로 제공되면 이를 그대로 이용하면 된다. 그리고 앞쪽 면의 z 좌표는 $-n$, 뒤쪽 면의 z 좌표는 $-f$ 로 하면 된다. 문제는 뒤쪽 면^{far plane}의 좌우와 상하는 제공되지 않았다는 점이다. 이것은 나타내는 값을 L, R, B, F라고 하면, 간단한 비례식으로 그 값을 쉽게 구할 수 있다.

$$L = \frac{l}{n}f, \quad R = \frac{r}{n}f, \quad B = \frac{b}{n}f, \quad T = \frac{t}{n}f$$

```
def drawFrustum(l, r, b, t, n, f):
    L = l * (f/n)
    R = r * (f/n)
    B = b * (f/n)
    T = t * (f/n)
    glColor3f(1,1,1)
    glBegin(GL_LINE_LOOP)
    glVertex3f(l,t,-n)
    glVertex3f(l,b,-n)
    glVertex3f(r,b,-n)
    glVertex3f(r,t,-n)
    glEnd()
    glBegin(GL_LINE_LOOP)
    glVertex3f(L,T,-f)
    glVertex3f(L,B,-f)
    glVertex3f(R,B,-f)
    glVertex3f(R,T,-f)
    glEnd()
    glBegin(GL_LINES)
    glVertex3f(l,t,-n)
    glVertex3f(L,T,-f)
    glVertex3f(l,b,-n)
    glVertex3f(L,B,-f)
    glVertex3f(r,b,-n)
    glVertex3f(R,B,-f)
    glVertex3f(r,t,-n)
    glVertex3f(R,T,-f)
    glEnd()
```

glFrustum에 의한 투영은 관찰자의 눈이 원점, 즉 (0,0,0)의 위치에 있고, z 축 음의 방향을 쳐다보는 것으로 가정한다. 그리고 가까운 평면은 $-n$ 의 z 좌표를 가지고, 면쪽은 $-f$ 가 된다. 따라서 관찰이 되는 물체는 z 축으로 음의 방향에 놓여 있어야 한다. 따라서 관찰하려고 하는 나선 모양의 물체의 좌표를 조금 옮겨 놓도록 하자.

```
def drawHelix():
    glBegin(GL_LINE_STRIP)
    for i in range(1000):
        z = i/10
        x, y = 0.5*math.cos(z), 0.5*math.sin(z)
        glVertex3f(x, y, -1.5-z/100)
    glEnd()
```

다음으로 resizeGL 함수 내에서 glOrtho 대신에 glFrustum을 호출하여 원근 투영이 일어나게 하면 된다. 이때 이 관측공간을 관찰하는 OpenGL 위젯에서는 그대로 glOrtho를 부르도록 하였다.

```
def resizeGL(self, width, height):
    # 카메라의 투영 특성을 여기서 설정
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    if not self.observation:
        glFrustum(
            self.left, self.right,
            self.bottom, self.top,
            self.near, self.far)
    else :
        glOrtho(
            -4, 4, -4, 4, -100, 100)
```

다음으로 렌더링을 수행하는 paintGL 함수를 수정해 보자. 이곳에서는 drawBox가 아니라 drawFrustum 함수를 호출하여야 할 것이다. 두 개의 OpenGL 위젯 가운데 절두체를 관찰하기 위해 만들어지 위젯에서는 관찰의 위치를 변경해야 할 것이다. 적절한 위치로 관찰자의 시점을 옮기고 절두체의 앞면 중앙을 관찰하기 위해 관찰 목표지점은 (0,0,-self.near)가 되도록 하였다.

```

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

    if self.observation:
        gluLookAt(1.5,1.0,0.2, 0,0,-self.near, 0,1,0)

    drawHelix()
    drawAxes()
    drawFrustum(
        self.left, self.right,
        self.bottom, self.top,
        self.near, self.far)

    # 그려진 프레임버퍼를 화면으로 송출
    glFlush()

```

키보드를 이용하여 glFrustum에 사용되는 파라미터를 조정하는 것은 앞에서 다룬 코드를 그대로 사용하면 될 것이다. 따라서 이를 정리하면 다음과 같은 코드가 된다.

glFrustum을 이용한 원근 투영 관측 공간 제어 실험

```

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

def drawAxes() :
    glBegin(GL_LINES)
    glColor3f(1,0,0)
    glVertex3f(0,0,0)
    glVertex3f(1,0,0)
    glColor3f(0,1,0)
    glVertex3f(0,0,0)
    glVertex3f(0,1,0)
    glColor3f(0,0,1)
    glVertex3f(0,0,0)

```

```

glVertex3f(0,0,1)
glEnd()

def drawFrustum(l, r, b, t, n, f):
    L = l * (f/n)
    R = r * (f/n)
    B = b * (f/n)
    T = t * (f/n)
    glColor3f(1,1,1)
    glBegin(GL_LINE_LOOP)
    glVertex3f(l,t,-n)
    glVertex3f(l,b,-n)
    glVertex3f(r,b,-n)
    glVertex3f(r,t,-n)
    glEnd()
    glBegin(GL_LINE_LOOP)
    glVertex3f(L,T,-f)
    glVertex3f(L,B,-f)
    glVertex3f(R,B,-f)
    glVertex3f(R,T,-f)
    glEnd()
    glBegin(GL_LINES)
    glVertex3f(l,t,-n)
    glVertex3f(L,T,-f)
    glVertex3f(l,b,-n)
    glVertex3f(L,B,-f)
    glVertex3f(r,b,-n)
    glVertex3f(R,B,-f)
    glVertex3f(r,t,-n)
    glVertex3f(R,T,-f)
    glEnd()

def drawHelix():
    glBegin(GL_LINE_STRIP)
    for i in range(1000):
        z = i/10
        x, y = 0.5*math.cos(z), 0.5*math.sin(z)
        glVertex3f(x, y, -1.5-z/100)
    glEnd()

class MyGLWidget(QOpenGLWidget):

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

        self.left = self.bottom = -0.5

```

```

        self.right = self.top = 0.5
        self.near = 1.5
        self.far = 2.5

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

    def resizeGL(self, width, height):
        # 카메라의 투영 특성을 여기서 설정
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        if not self.observation:
            glFrustum(
                self.left, self.right,
                self.bottom, self.top,
                self.near, self.far)
        else :
            glOrtho(
                -4, 4, -4, 4, -100, 100)

    def paintGL(self):

        self.projection_update()

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

        if self.observation:
            gluLookAt(1.5,1.0,0.2, 0,0,-self.near, 0,1,0)

        drawHelix()
        drawAxes()
        drawFrustum(
            self.left, self.right,
            self.bottom, self.top,
            self.near, self.far)

        # 그려진 프레임버퍼를 화면으로 송출
        glFlush()

    def projection_update(self):
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        if not self.observation:
            glFrustum(
                self.left, self.right,

```

```

        self.bottom, self.top,
        self.near, self.far)
else :
    glOrtho(
        -4, 4, -4, 4, -100, 100)

class MyWindow(QMainWindow):

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

        ##### GUI 설정
        central_widget = QWidget()
        self.setCentralWidget(central_widget)
        gui_layout = QHBoxLayout() # CentralWidget의 수평 나열 레이아웃

        # 배치될 것들 - 정점 입력을 받기 위한 위치
        central_widget.setLayout(gui_layout)

        ##### OpenGL Widget 추가
        self.glWidget1 = MyGLWidget(observation=False) # OpenGL Widget
        self.glWidget2 = MyGLWidget(observation=True)
        gui_layout.addWidget(self.glWidget1)
        gui_layout.addWidget(self.glWidget2)

    def keyPressEvent(self, e):
        if e.key() == Qt.Key.Key_A:
            self.glWidget1.left -= 0.1
            self.glWidget2.left -= 0.1
        elif e.key() == Qt.Key.Key_S:
            self.glWidget1.left += 0.1
            self.glWidget2.left += 0.1
        elif e.key() == Qt.Key.Key_D:
            self.glWidget1.right -= 0.1
            self.glWidget2.right -= 0.1
        elif e.key() == Qt.Key.Key_F:
            self.glWidget1.right += 0.1
            self.glWidget2.right += 0.1
        elif e.key() == Qt.Key.Key_Q:
            self.glWidget1.top += 0.1
            self.glWidget2.top += 0.1
        elif e.key() == Qt.Key.Key_W:
            self.glWidget1.top -= 0.1
            self.glWidget2.top -= 0.1
        elif e.key() == Qt.Key.Key_Z:
            self.glWidget1.near += 0.1

```

```

        self.glWidget2.near += 0.1
    elif e.key() == Qt.Key.Key_X:
        self.glWidget1.near -= 0.1
        self.glWidget2.near -= 0.1
    elif e.key() == Qt.Key.Key_V:
        self.glWidget1.far += 0.1
        self.glWidget2.far += 0.1
    elif e.key() == Qt.Key.Key_C:
        self.glWidget1.far -= 0.1
        self.glWidget2.far -= 0.1

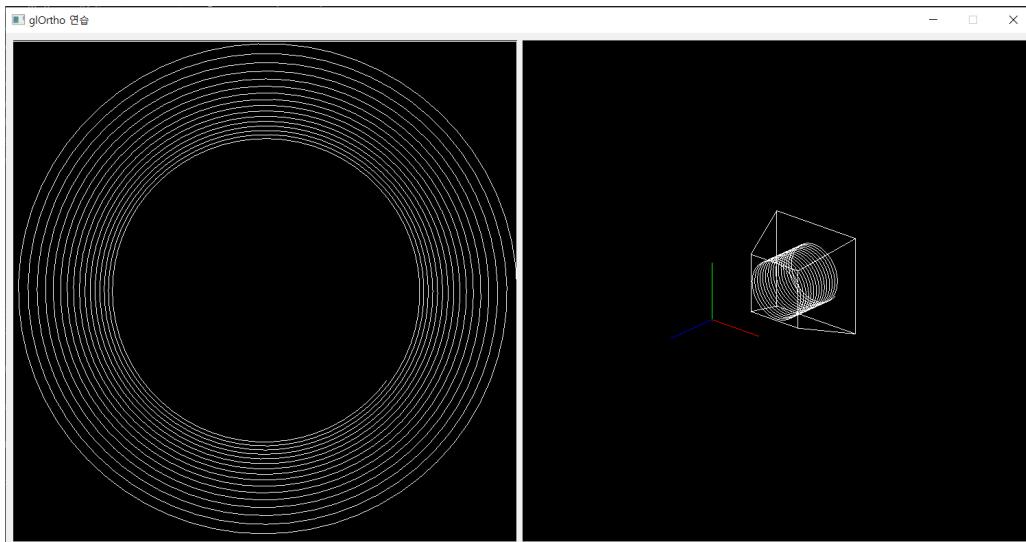
    self.glWidget1.update()
    self.glWidget2.update()

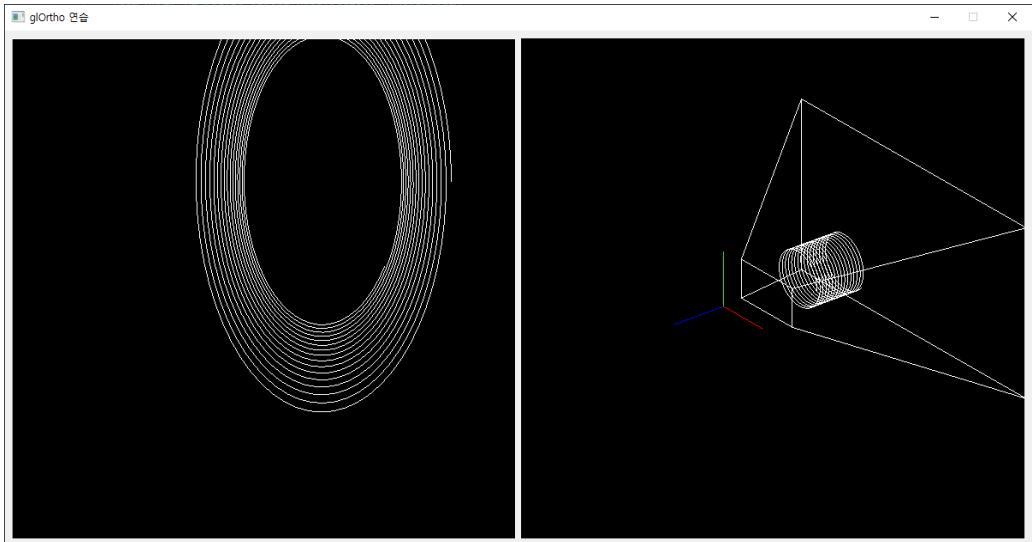
def main(argv = []):
    app = QApplication(argv)
    window = MyWindow('glOrtho 연습')
    window.setFixedSize(1200, 600)
    window.show()
    app.exec()

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

```

그림 13은 이 코드를 실행했을 때에 얻게 되는 결과이다. 키보드를 이용하여 glFrustum의 파라미터를 변경할 수 있으며, 이를 통해 glFrustum이 만들어 내는 공간과 그 공간 내부의 객체가 어떻게 렌더링 되는지를 확인할 수 있다.





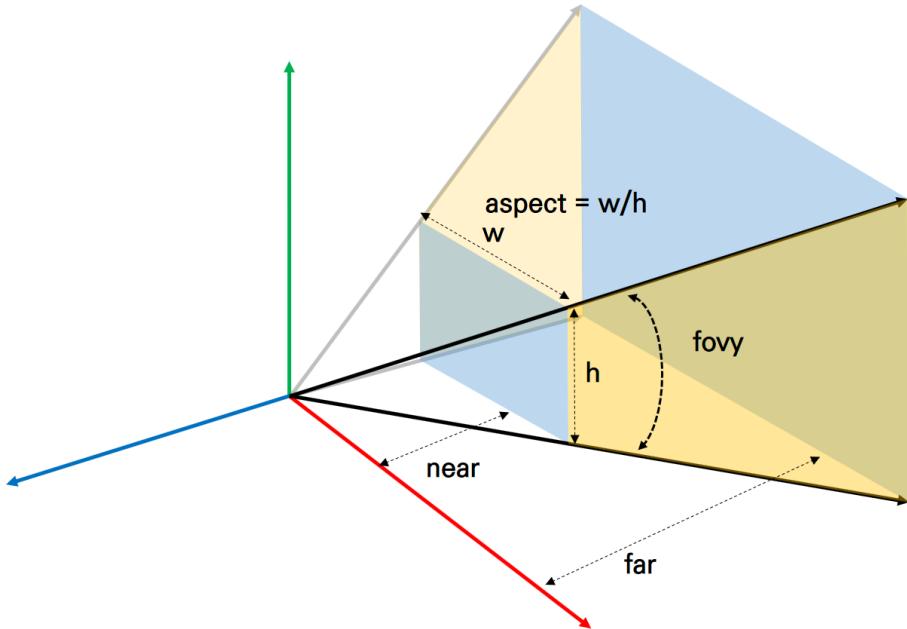
[그림 13] glFrustum 파라미터를 변경하여 투영 결과 변화 관찰하기

3.7 gluPerspective를 이용한 편리한 투영 설정

glFrustum을 이용하여 원근 투영을 결정하는 것이 OpenGL의 기본적인 접근법이다. 그런데, glFrustum의 파라미터를 이용하여 만들어지는 관측공간은 그다지 직관적이지 않다. 그래서 사람들이 조금 더 이해하기 쉬운 파라미터를 이용하여 원근 투영을 결정하는 방법이 있는데, 이것이 gluPerspective이다. 이 함수는 다음과 같은 함수 원형prototype을 갖는다.

gluPerspective(fovy, aspect, near, far)

fovy는 y축 시야각 field of view angle in the y direction을 의미한다. 이 값은 라디안radian이 아니라도 degree를 사용한다. aspect는 절두체 앞면과 뒷면의 종횡비를 의미한다. 이 값이 렌더링이 최종적으로 이루어지는 윈도우 공간의 종횡비가 일치하는 것이 바람직하다. near와 far는 지금까지 glFrustum에서 사용했던 것과 같은 의미이다.



[그림 14] gluPerspective 파라미터의 이해

OpenGL은 glFrustum을 이용하여 원근 투영을 설정하므로 gluPerspective가 호출된 경우, 여기에서 설정된 파라미터를 바탕으로 glFrustum에 적용되는 파라미터를 계산하고, 이를 이용하여 절두체를 생성한다. 이때 glFrustum에 필요한 파라미터는 다음과 같이 계산된다.

`glFrustum(left, right, bottom, top, near, far)`

`gluPerspective(fovy, aspect, near, far)`

$$top = near \cdot \tan \frac{\theta}{2}$$

$$bottom = -top$$

$$right = top \cdot aspect$$

$$left = -right$$

gluPerspective의 파라미터가 더욱 직관적이기는 하지만, glFrustum에 비해 다소 제한적인 설정만 가능하다는 것을 알 수 있다. 계산된 top과 bottom은 x 축,, 그리고 right와 left는 y 축을 기준으로 대칭을 이루고 있다. 따라서 대칭 형태가 아닌 절두체는 gluPerspective를 통해 구현할 수 없다는 것이다.

3.8 카메라를 움직여 FPS 형태로 돌아다녀 보기

카메라의 위치는 기본적으로 원점에 놓여 있고, z 축 음의 방향을 쳐다 보는 것으로 되어 있다. 그런데 이렇게 제한적인 카메라 위치와 방향으로는 3차원 공간에 대한 다양한 관찰을 할 수가 없다. 그래서 OpenGL에서는 gluLookAt이라는 함수를 제공한다. 우리는 glOrtho와 glFrustum이 만들어내는 관측 공간을 확인하기 위해 카메라를 gluLookAt을 이용하여 옮겨 보았다. 이 함수의 원형은 다음과 같다.

```
gluLookAt(eye_x, eye_y, eye_z, at_x, at_y, at_z, up_x, up_y, up_z)
```

이 함수는 카메라가 (eye_x, eye_y, eye_z)에 놓여 있고, (at_x, at_y, at_z) 지점을 쳐다 보는데, 카메라의 위쪽 방향이 (up_x, up_y, up_z)를 향하도록 자세를 잡았을 때에 관찰되는 장면이 그려지게 한다. 그런데, 실제로 OpenGL에서는 카메라를 원점에서 옮길 수가 없다. 그러면 이것은 어떻게 구현될 것일까?

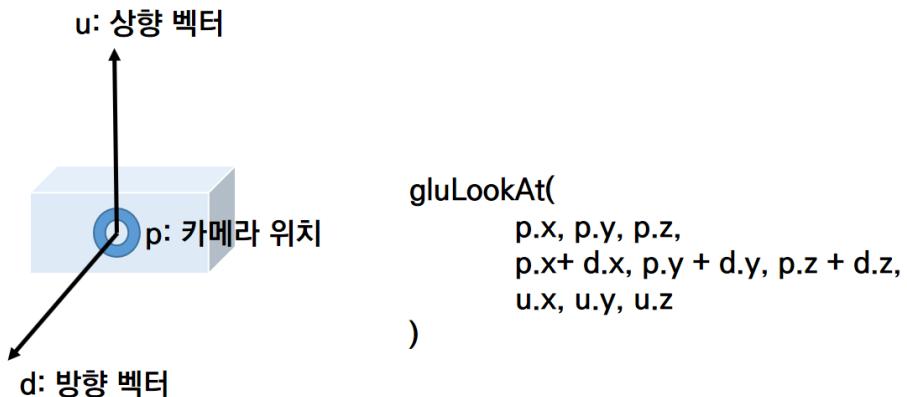
카메라의 위치를 옮겨서 관찰되는 장면은 물체를 카메라의 반대로 옮기는 것과 동일한 결과가 된다. 즉 카메라의 위치를 원점이 아니라 (eye_x, eye_y, eye_z)로 옮긴 것은, 카메라는 원점에 그대로 두고 모든 물체를 (-eye_x, -eye_y, -eye_z)로 옮긴 것과 동일한 결과가 될 것이다. 이렇게 물체들을 모두 옮겨 놓는 일은 모델뷰 행렬이 하는 일이다. 따라서 카메라의 위치를 옮기는 작업 gluLookAt은 glMatrixMode(GL_MODELVIEW)를 호출하여 모델뷰 행렬을 조작하는 모드에서 호출된다.

OpenGL은 세 종류의 행렬을 가지고 있다. 텍스처 행렬, 모델뷰 행렬, 투영 행렬이 그 세 가지 행렬이다. 이 가운데 가상 공간 내의 좌표를 결정하는 행렬은 모델뷰 행렬과 투영 행렬이다. 모델뷰 행렬은 공간 내에서 객체들의 좌표를 변환 transform하여 새로운 좌표로 만드는 역할을 한다. 투영 행렬을 앞서 살펴본 바와 같이 가상 객체를 투영면에 옮겨 놓는 데에 사용되는 행렬이다. OpenGL에서 그리는 모든 객체의 좌표는 우선 모델뷰 행렬에 곱해져서 자리를 잡고, 투영 행렬에 곱해져서 평면으로 옮겨지는 것이다. 모델뷰 행렬에 대해서는 변환을 다루면서 본격적으로 살펴볼 것이다.

키보드를 이용하여 카메라를 제어하는 프로그램을 작성해 보자. 카메라는 위치 \mathbf{p} 와 쳐다보는 방향 \mathbf{d} 로 표현할 수 있을 것이다. 카메라의 상향 벡터는 언제는 y 축 방향인 $(0, 1, 0)$ 으로 가정하자. 그러면 gluLookAt에 이 카메라 상태를 전달하는 방법은 다음과 같을 것이다.

`gluLookAt(px, py, pz, px + dx, py + dy, pz + dz, 0, 1, 0)`

좀 더 일반적인 경우를 생각하면 그림 14와 같이 이해할 수 있다.



[그림 15] 카메라 위치와 관찰 방향, 상향 벡터를 기준으로 gluLookAt 파라미터 결정하기

카메라를 전진시키는 것은 매우 간단하다. 카메라를 이동 시키는 간격이 s 라고 하면 다음과 같이 카메라의 위치를 바꾸면 된다. 이때 방향 벡터 \mathbf{d} 는 바뀌지 않는다.

$$\mathbf{p}^{new} = \mathbf{p} + s\mathbf{d}$$

이때, 방향은 바뀌지 않다. 뒤로 물러서는 동작은 다음과 같다.

$$\mathbf{p}^{new} = \mathbf{p} - s\mathbf{d}$$

그러면, 카메라의 방향을 바꾸는 것은 어떻게 하면 될까? 이것은 카메라가 xz 평면에 있고, 상향 벡터가 $(0, 1, 0)$ 이라는 가정을 하면 y 축 기준으로 얼마나 회전을 했는지로 계산하면

될 것이다. 이 회전의 정도를 θ 라고 하자. 그러면 x 와 z 좌표를 다음과 같이 결정하면 회전 각도에 따라 카메라의 방향이 회전을 하게 될 것이다.

$$\mathbf{d} = (\sin \theta, 0, \cos \theta)$$

회전각이 0이라고 생각해 보자. 그러면 $\mathbf{d} = (0, 0, 1)$ 이 되어 z 축 방향을 쳐다보게 될 것이다. 그리고 회전각이 증가하면 y 축을 따라 회전이 이루어지게 된다. 따라서 회전을 요구하는 키보드 입력이 들어오면 θ 만 변경하고, 이를 바탕으로 방향 벡터를 새롭게 계산한 뒤, gluLookAt을 호출하면 된다.

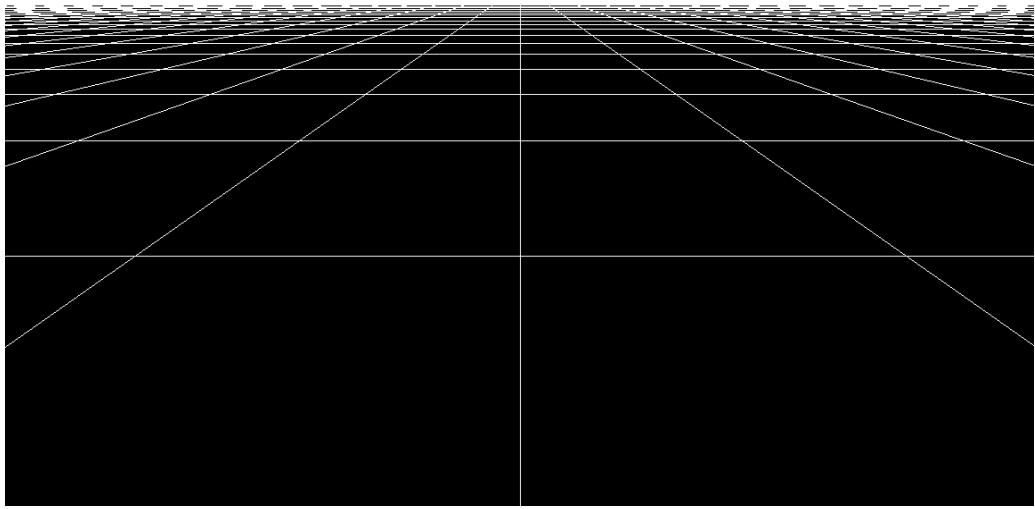
이제 자유롭게 카메라를 옮겨 가상 공간을 다닐 수 있는 프로그램을 작성해 보자. 우선 삼각함수를 사용하기 위해 math 패키지를 임포트한다. 그리고, 벡터 연산을 다루기 위해 넘파이를 사용할 것이므로 numpy도 임포트한다. 넘파이는 파이썬의 기본 자료형인 리스트 list와 비슷하지만, 훨씬 더 빠른 접근 속도와 벡터화된 연산을 지원하여 과학기술 계산에 표준적으로 사용되는 패키지이다. 이에 대한 자세한 설명은 부록을 참고하기 바란다.

```
import math
import numpy as np
```

다음으로 카메라가 이동하는 것을 확인하기 위한 배경으로 바닥면을 그리는 함수를 만들어 보자. 이 함수는 다음과 같이 구현할 수 있을 것이다.

```
def drawPlane() :
    glBegin(GL_LINES)
    for i in range(100):
        glVertex3f(i-50, 0, -50)
        glVertex3f(i-50, 0, 50)
        glVertex3f(-50, 0, i-50)
        glVertex3f(50, 0, i-50)
    glEnd()
```

이 함수는 그림 16과 같은 바닥면을 표현하는 격자를 그려낼 것이다.



[그림 16] 바닥면 격자의 모양

다음으로 OpenGL 위젯을 상속받아 클래스를 하나 만들자. 이 클래스는 카메라의 위치와 방향, 그리고 방향을 제어하는 각도를 가지게 될 것이다. 따라서 다음과 같이 생성자를 구현할 수 있다.

```
class MyGLWidget(QOpenGLWidget):  
  
    def __init__(self, parent=None):  
        super().__init__(parent)  
  
        self.p = np.array([0.0, 0.0, 0.0], dtype=float)  
        self.d = np.array([0.0, 0.0, 1.0], dtype=float)  
        self.angle = 0.0
```

이 OpenGL Widget에 gluPerspective를 이용하여 투영을 설정하고, gluLookAt을 이용하여 관리되고 있는 카메라 위치에서 카메라 방향을 쳐다보기 하는 코드는 다음과 같다.

```
def resizeGL(self, width, height):  
    # 카메라의 투영 특성을 여기서 설정  
    glMatrixMode(GL_PROJECTION)  
    glLoadIdentity()  
    gluPerspective(60, width/height, 0.1, 100)
```

```

def paintGL(self):

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

    gluLookAt(self.p[0], self.p[1]+0.6, self.p[2],
              self.p[0]+self.d[0], self.p[1]+self.d[1], self.p[2]+self.d[2],
              0, 1, 0)

    drawPlane()
    glFlush()

```

일반적인 PyQt 애플리케이션을 하나 만들고, 윈도우를 부착한 뒤에 이 OpenGL 위젯을 달면 될 것이다. 그리고 키보드 입력에 따라 카메라의 위치를 바꾸거나 방향을 바꾸는 이벤트 처리 메소드를 윈도우 클래스에 구현해 보자.

```

def keyPressEvent(self, e):
    s = np.array([0.1])
    angle = self.glWidget.angle
    if e.key() == Qt.Key.Key_W:
        self.glWidget.p += s*self.glWidget.d
    elif e.key() == Qt.Key.Key_S:
        self.glWidget.p -= s*self.glWidget.d
    elif e.key() == Qt.Key.Key_A:
        angle += 0.01
        self.glWidget.d = (math.sin(angle), 0, math.cos(angle))
    elif e.key() == Qt.Key.Key_D:
        angle -= 0.01
        self.glWidget.d = (math.sin(angle), 0, math.cos(angle))

    self.glWidget.angle = angle
    self.glWidget.update()

```

이제 키보드를 이용하여 바닥면 위를 자유자재로 이동할 수 있을 것이다. 사용되는 키는 A-W-S-D로 일반적인 FPS 게임 등에서 사용되는 방식이다. 전체 코드를 정리하면 다음과 같으며, 실행 결과는 그림 17에 나타나 있다.

FPS 방식으로 카메라를 제어하는 프로그램

```

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

def drawPlane() :
    glBegin(GL_LINES)
    for i in range(100):
        glVertex3f(i-50, 0, -50)
        glVertex3f(i-50, 0, 50)
        glVertex3f( -50, 0, i-50)
        glVertex3f( 50, 0, i-50)
    glEnd()

class MyGLWidget(QOpenGLWidget):

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

        self.p = np.array([0.0,0.0,0.0], dtype=float)
        self.d = np.array([0.0,0.0,1.0], dtype=float)
        self.angle = 0.0

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

    def resizeGL(self, width, height):
        # 카메라의 투영 특성을 여기서 설정
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(60, width/height, 0.1, 100)

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

        gluLookAt(self.p[0], self.p[1]+0.6, self.p[2],
                  self.p[0]+self.d[0], self.p[1]+self.d[1], self.p[2]+self.d[2],
                  0, 1, 0)
        drawPlane()
        glFlush()

```

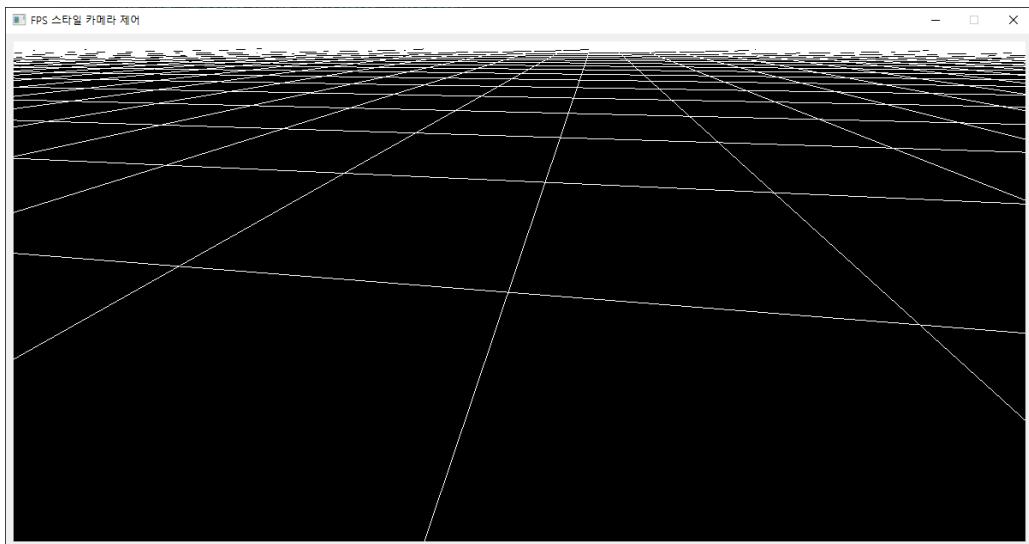
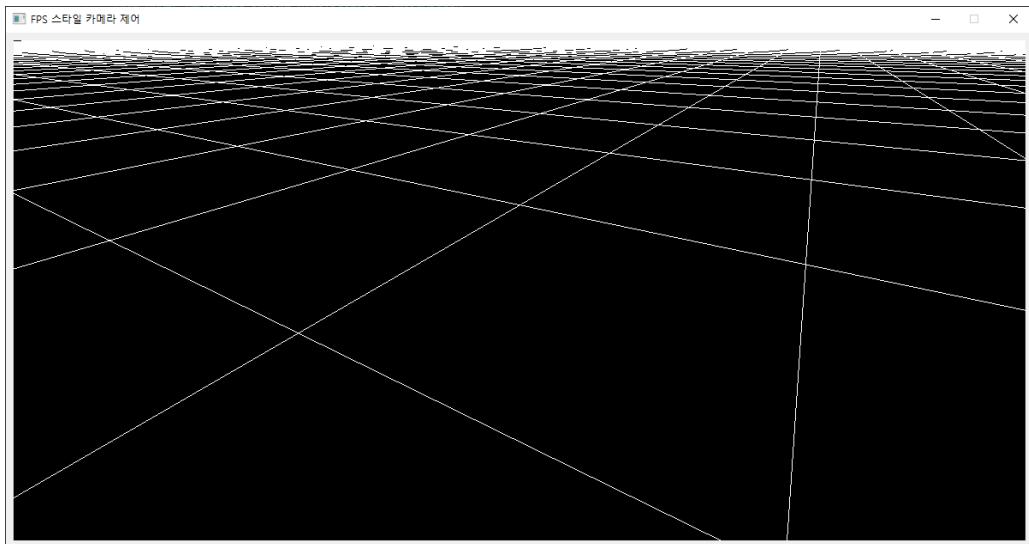
```
class MyWindow(QMainWindow):
    def __init__(self, title=''):
        QMainWindow.__init__(self) # QMainWindow 슈퍼 클래스의 초기화
        self.setWindowTitle(title)

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

    def keyPressEvent(self, e):
        s = np.array([0.1])
        angle = self.glWidget.angle
        if e.key() == Qt.Key.Key_W:
            self.glWidget.p += s*self.glWidget.d
        elif e.key() == Qt.Key.Key_S:
            self.glWidget.p -= s*self.glWidget.d
        elif e.key() == Qt.Key.Key_A:
            angle += 0.01
            self.glWidget.d = (math.sin(angle), 0, math.cos(angle))
        elif e.key() == Qt.Key.Key_D:
            angle -= 0.01
            self.glWidget.d = (math.sin(angle), 0, math.cos(angle))
        self.glWidget.angle = angle
        self.glWidget.update()

def main(argv = []):
    app = QApplication(argv)
    window = MyWindow('FPS 스타일 카메라 제어')
    window.setFixedSize(1200, 600)
    window.show()
    app.exec()

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



[그림 17] 카메라 이동 제어 결과