Computer Graphics

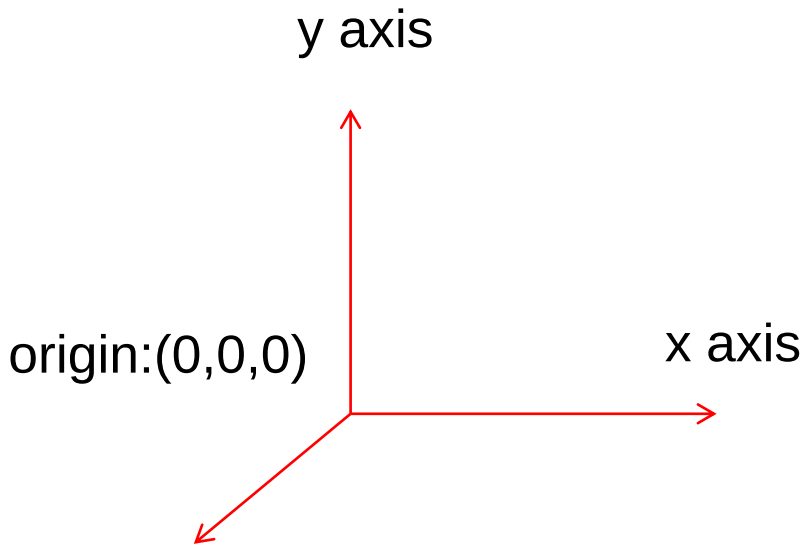
Prof. Jibum Kim

Department of Computer Science & Engineering

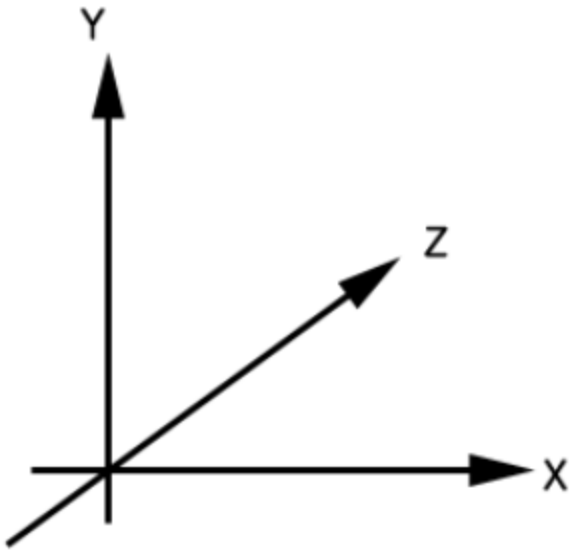
Incheon National University

- **Orthogonal projection (직교 투영)**
- **가시 부피 (viewing box)**

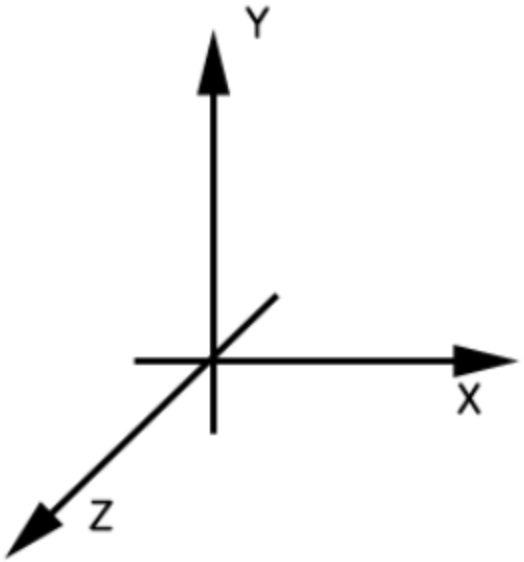
- **3차원 좌표계: 직교 좌표계 (Cartesian coordinate system)를 사용. 원점에서 서로 직각으로 교차하는 3개의 좌표축 벡터로 이루어짐**
- **Specify direction in each axis: +z, -z, +y, -y, +x, -x**
- **오른손 법칙: +x축의 끝에서 +y축의 끝을 향해 오른손 주먹을 말아 쥐었을 때 엄지 방향이 +z축이다 (OpenGL에서 사용)**



- DirectX에서는 **왼손좌표계**를 사용



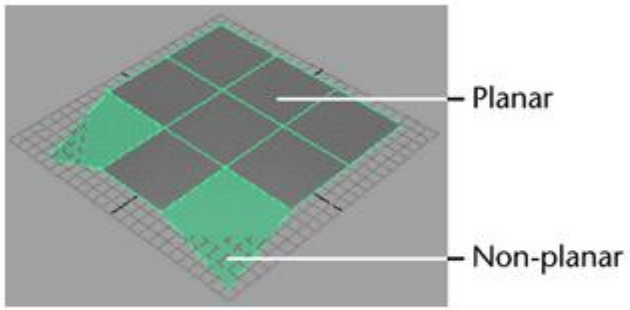
왼손 좌표계

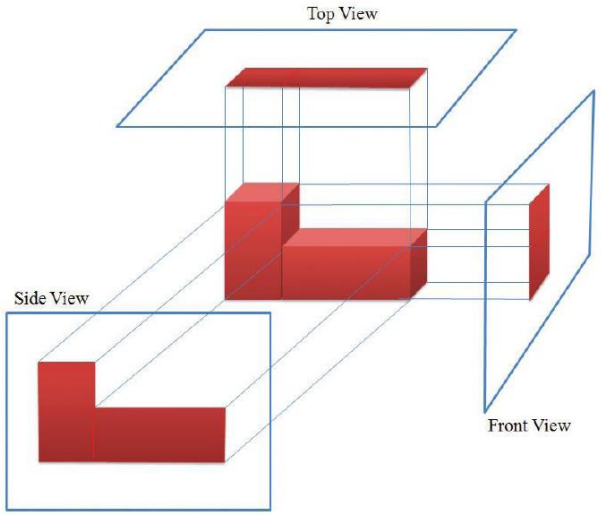


오른손 좌표계

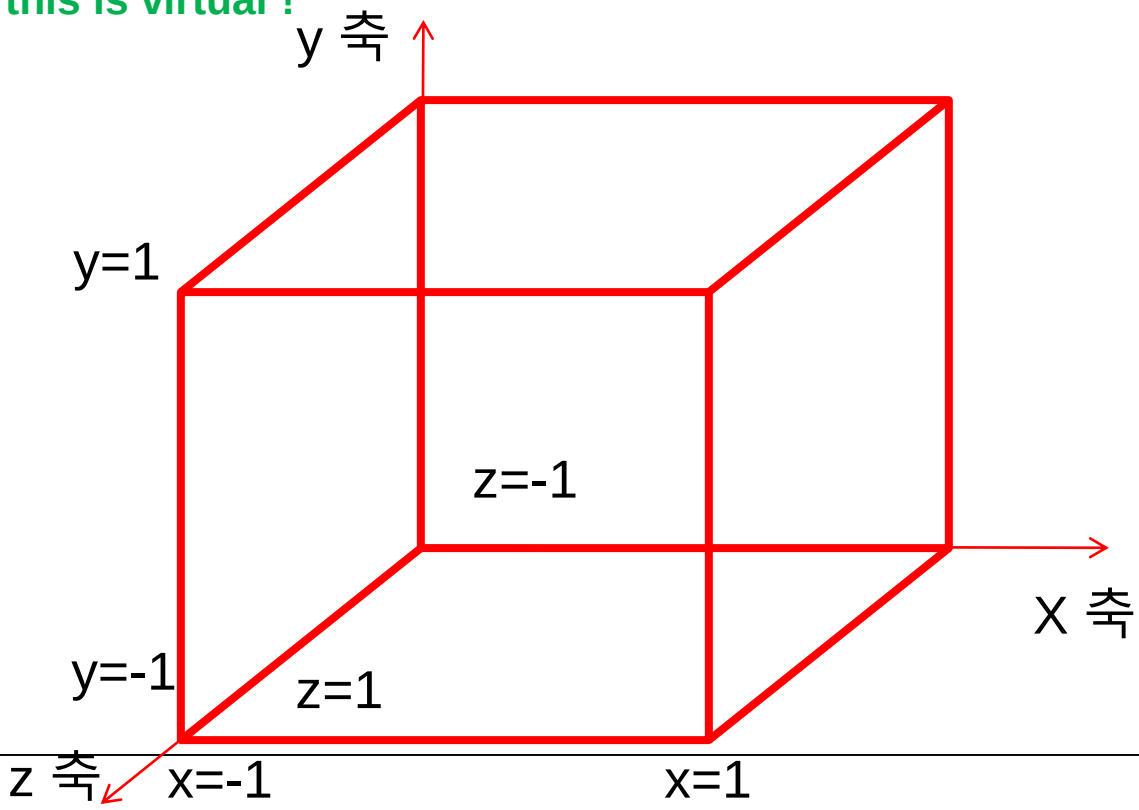
- **1. Orthogonal projection (orthographic projection, 직교 투영)**
- **Default projection in OpenGL**
- **First, projection means 3D objects are projected onto a planar surface (2D)**

Example of an orthogonal projection





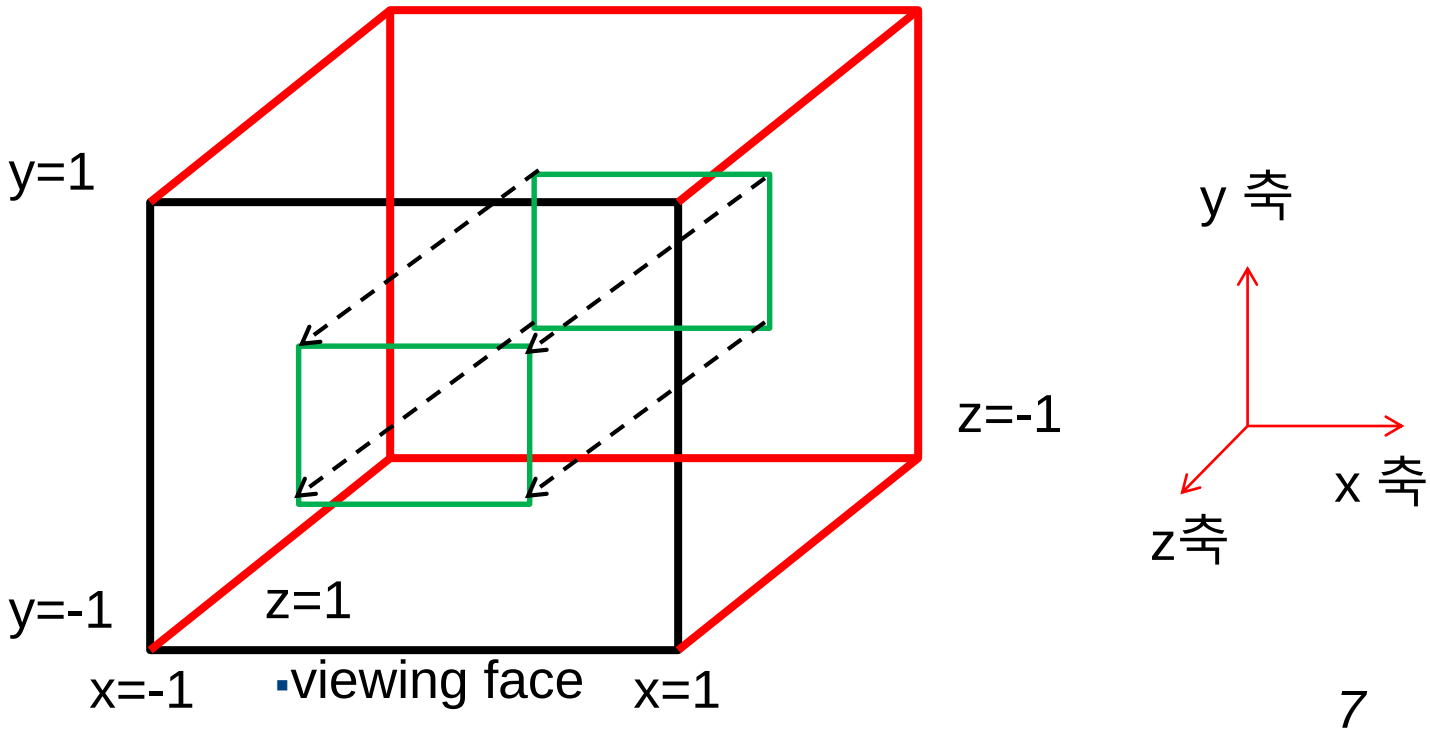
- **By default, object can be seen only if it's world coordinate is specified between x=-1~1 and y=-1~1. In z-axis, it should be set between z=-1~1.**
- **By default, in order to see the object, world coordinates should be specified between x=-1~1, y=-1~1, z=-1~1. This cube is called a viewing box (가시 부피)**
- **Note that this is virtual !**



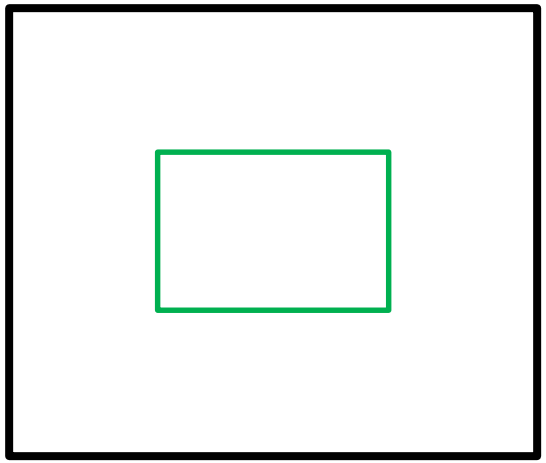
- Default orthogonal projection in OpenGL (투영을 별도로 설정하지 않으면)

- An object (green color) inside the viewing box (red color) is projected onto z=1, perpendicularly (수직으로 투영된다, 상이 맺힌다)

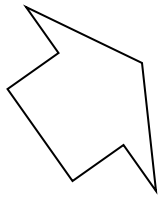
- So, z value of the object is ignored. The face where z=1 is called a front face (viewing face)



- 물체는 orthogonal projection시 z축으로 수직으로 내려온다 실제 viewer (camera)가 보는 것은 사람과 가장 가까운 2차원 사각형 (검은색) 면이다 (front face 혹은 viewing face라 한다). 이해를 쉽게 하기 위해 camera가 z축으로 무한히 먼 곳 (+z 방향으로 무한히 먼 곳)에 위치해 있다고 가정하고 –z 방향을 바라본다고 가상으로 가정한다 (실제로 default 카메라 위치는 다름)



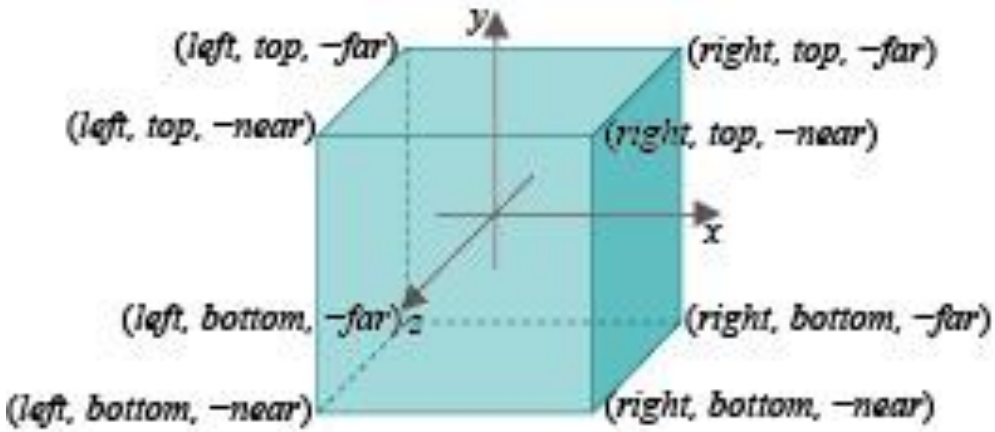
- Viewing face



-z direction

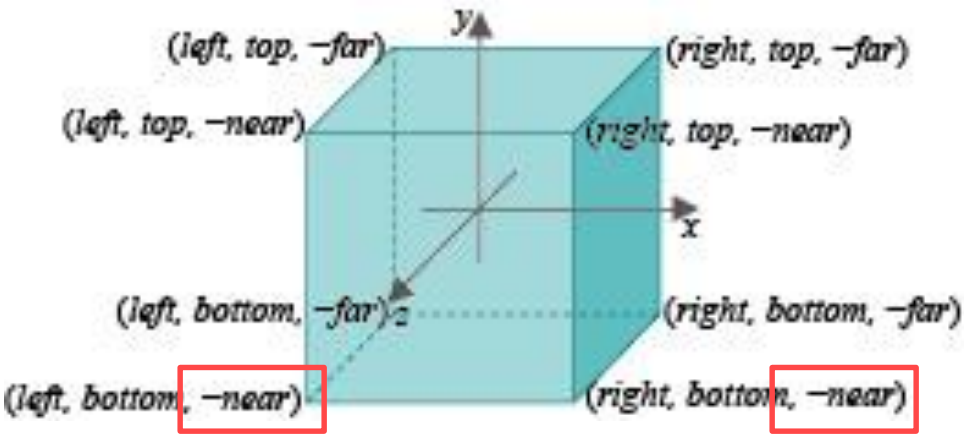
- OpenGL에서의 기본적인 가시 부피 아래와 같은 형태의 cube이다
- x=-1~1, y=-1~1, z=-1~1
- glOrtho 함수를 이용하면 가시 부피를 변경 가능하다

- glOrtho(left, right, bottom, top, near, far)
- x_{min}=left, x_{max}=right
- y_{min}=bottom, y_{max}=right
- z_{min}=-far, z_{max}=-near



- For example, let
- glOrtho(-1, 1, -1, 1, -1, 1),
- and draw a viewing box
- also specify each corner

- **glOrtho(left, right, bottom, top, near, far)**
- **Note that: front face is –near**

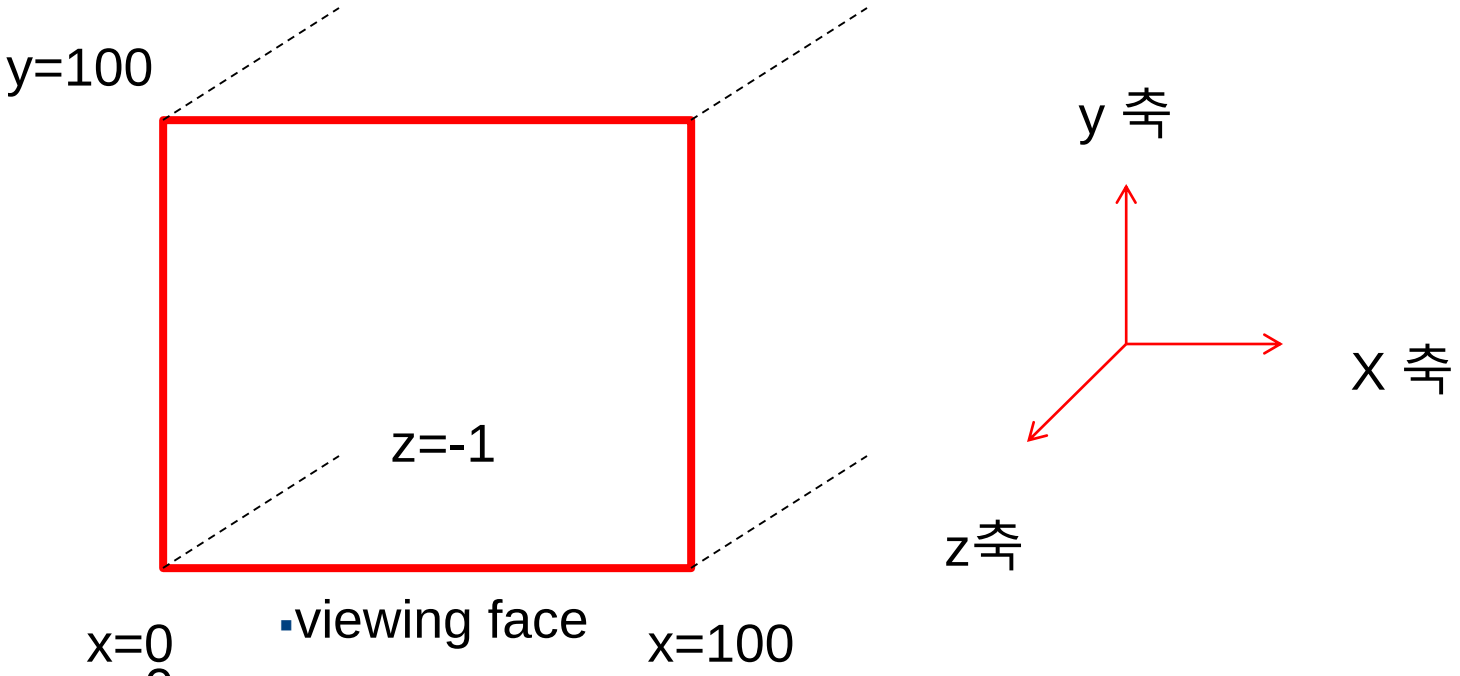


가시 부피 (viewing box)를 그려보자

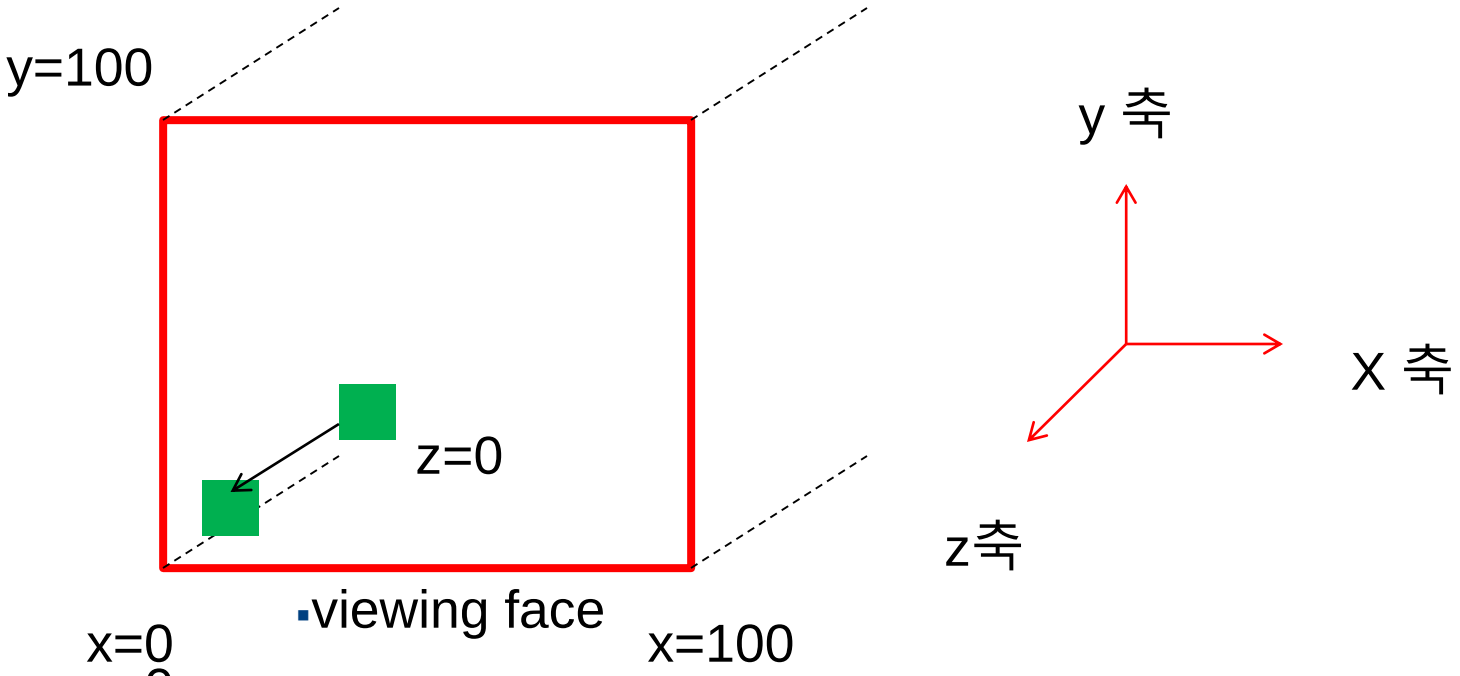
glOrtho(-2,2,-2,2,-1,1)

```
#include <GL/glut.h>
void Display(){
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0.0, 1.0, 0.0);
   glBegin(GL_QUADS);
   glVertex3f(10,10, 0.0);
   glVertex3f(20,10, 0.0);
   glVertex3f(20,20, 0);
     glVertex3f(10,20 ,0);
  glEnd();
  glFlush();
 }
int main(){
  glutInitWindowSize(200,200);
  glutInitWindowPosition(100,100);
  glutCreateWindow("OpenGL Hello World!");
    glClearColor(1.0, 0.0, 0.0, 0.0);
  glOrtho(0, 100, 0, 100, -1, 1);   //orthogonal projection
  glutDisplayFunc(Display);
  glutMainLoop();
  return 0;
}
```

- **Viewing box:** **glOrtho(0, 100, 0, 100, -1, 1);**

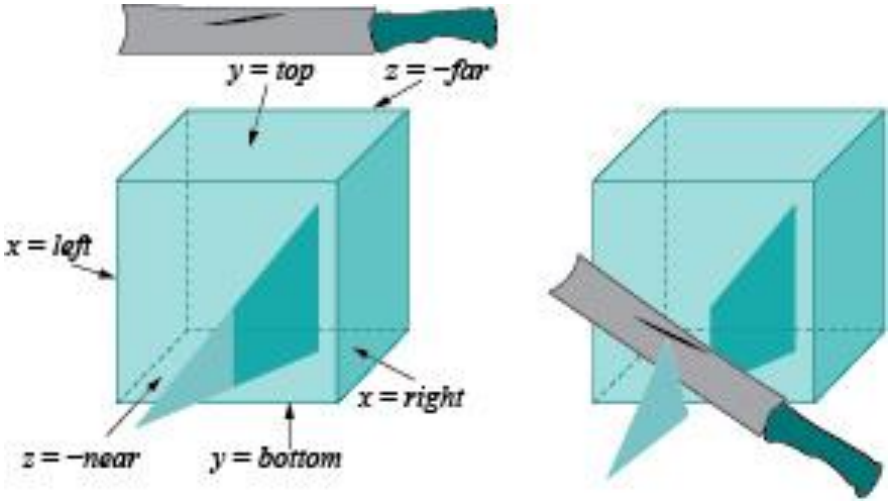


- **So, the rectangle at z=0 is projected to z=1**
- **z=-near로 직교 투영된다**



- **Clipping**

- **Clipping is a fundamental task in graphics, needed to keep those parts of an object that lie outside a given region from being drawn**
- **아래 예: 가시 부피 바깥에 위치한 object의 clipping**



- Clipping 알고리즘은 굉장히 많이 개발되어 있다
- OpenGL에서는 내부에서 구현된 clipping 알고리즘이 자동으로 실행되기 때문에 프로그래머가 clipping 알고리즘을 구현할 필요는 없다
- 후에 rasterization 부분에서 여러 가지 중요한 clipping 알고리즘들에 대해서 배울 예정이다
- Line clipping algorithm
- Polygon clipping algorithm

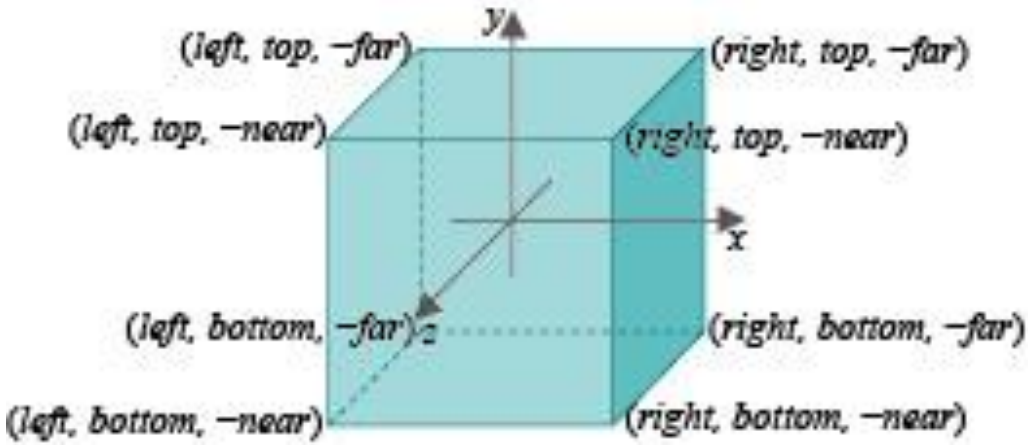
```
#include <GL/glut.h>
void Display() {
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0, 1.0, 0.0);
glBegin(GL_QUADS);
glVertex3f(-20, -20, 0);
glVertex3f(80, 20, 0);
glVertex3f(120, 120, 0);
glVertex3f(20, 80, 0);
glEnd();

glFlush();
}
int main() {
glutInitWindowSize(200, 200);
glutInitWindowPosition(100, 100);
glutCreateWindow("OpenGL Hello World!");
glClearColor(1.0, 0.0, 0.0, 0.0);
glOrtho(0, 200, 0, 200, -1, 1);//orthogonal projection
glutDisplayFunc(Display);
glutMainLoop();
return 0;
}
```

- **직교 투영과 동차좌표**

- 직교 투영의 경우
- glOrtho(left, right, bottom, top, near, far)
- 으로 생성된 가시 부피 안의 점 P(x, y, z)는
- P’(x’, y’, z’)= (x, y, -near)로 투영되고, x-y평면상의 좌표는 보존된다
- 즉, x’=x, y’=y, z’=-near
- 이를 행렬로 나타내면

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



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

- 이와 같이 3차원 좌표를 3개의 요소로 표시하지 않고 차원을 하나 높여서 4차원으로 표현한 것을 **동차 좌표 (homogeneous coordinate)** 이라 한다. 후에 변환 부분에서 동차 좌표에 대해서 자세히 배운다
- 이 경우 동차 좌표를 쓰면 장점이 뭘까?

- **원근 투영의 필요성**

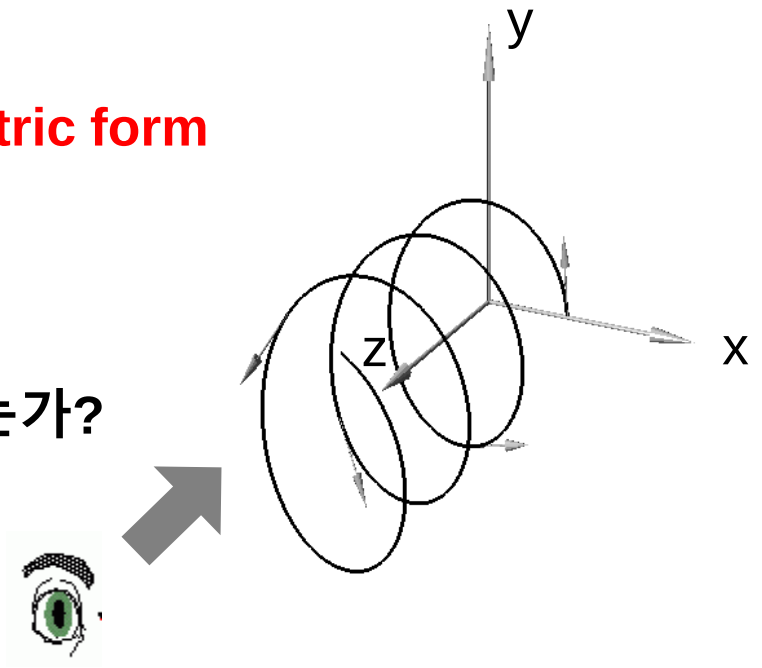
- **직교투영은 이해하기는 쉽지만 현실적이지 않다.** 사람 눈에서 멀리 떨어져 있는 물체는 가까이 있는 물체보다 작게 투영되어야 보다 현실적이다

- 하지만, 직교투영에서는 물체가 z축으로 어디에 위치해있는지는 물체가 가시 부피 안에 있는 한 상관이 없었다. 즉, 카메라 (viewer)와 물체와의 거리가 무시 된다

- 가시 부피 안에 있으면 **z값을 무시하고 z=-near인 viewing face에 투영**되었다

- 원근 투영의 필요성에 대해서 이해하기 위하여 다음의 helix 예에 대해서 살펴보자
- 3D Helix의 radius (반지름, R), parametric form
- x=R*cos(t), y=R*sin(t), z=t-60,
- R=20, $-10\pi \leq t \leq 10\pi$

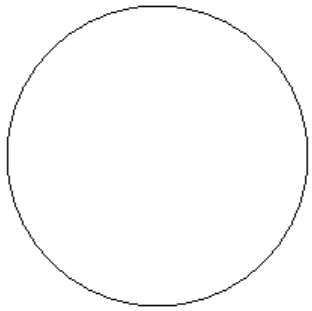
t값이 커짐에 따라 Helix는 어떻게 이동하는가?



직교 투영을 사용시 helix가 가시부피 안에 있다면 어떻게 보일까?

Viewer (카메라)는 기본적으로 +z에 서서 –z 축 방향을 바라본다고 하자

- 가시부피를 아래와 같이 정하고 직교투영을 사용하였다
- glOrtho(-50.0, 50.0, -50.0, 50.0, 0.0, 100.0);
- 어떻게 보이는지 확인해보자
- https://www.dropbox.com/s/3wb7bw4b46xvg32/helix.txt?dl=0



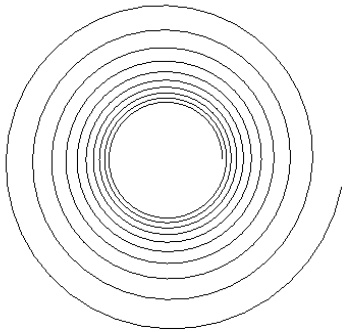
- 이런 helix를 사람의 실제 눈에 가까운 투영 방법 (원근 투영)을 사용하면 어떻게 보일까?
- 다음에 배울 glFrustum() 함수를 사용하여 원근 투영을 사용해 보았다

- glOrtho 부분을
- glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0);
- 로 바꾸어보자

- z값의 변화에 따라 원의 크기가 달라져 보이고 보다 현실감 있게 helix가 보인다
- 실제 사람 눈은 어떤 사물이 사람 눈과의 거리와 가까우면 같은 사물이어도 크게 보이고 사람 눈과의 거리와 멀면 같은 사물이어도 작게 보인다 => 이를 원근투영이라 한다
- OpenGL에서 viewer (카메라)는 +z 방향에 서서 –z 축 방향을 바라본다고 했으므로

  viewer와의 거리는 z축에서의 거리를

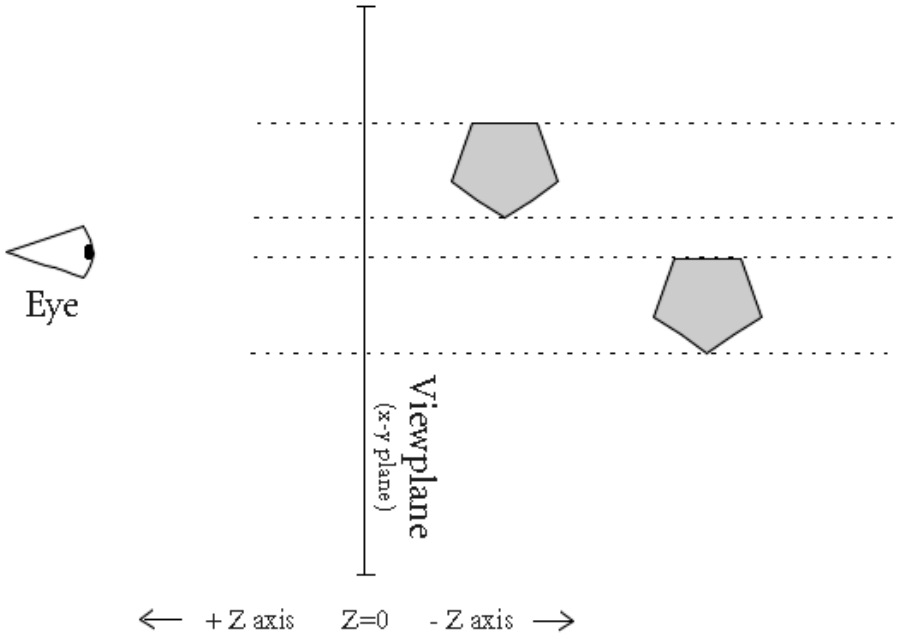
  의미한다



- **Perspective projection (원근 투영)**

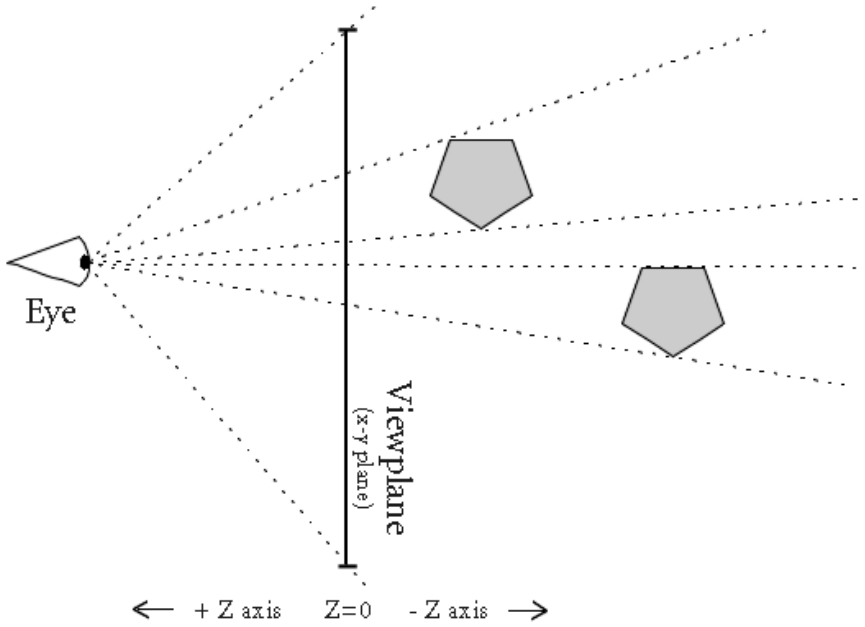
- 원근 투영의 예 (르네상스 그림)
- 카메라로부터 멀리 있는 물체는 작게 보이고
  카메라로부터 가까이 있는 물체는 크게 보인다

- **Orthogonal projection (직교 투영)**
- **가시 부피 (viewing box) 안의 같은 모양, 크기의 물체는 z축으로 서로 다른 위치에 있어도 같은 크기로 viewing face (plane)에 투영 되고 같은 크기로 보인다**

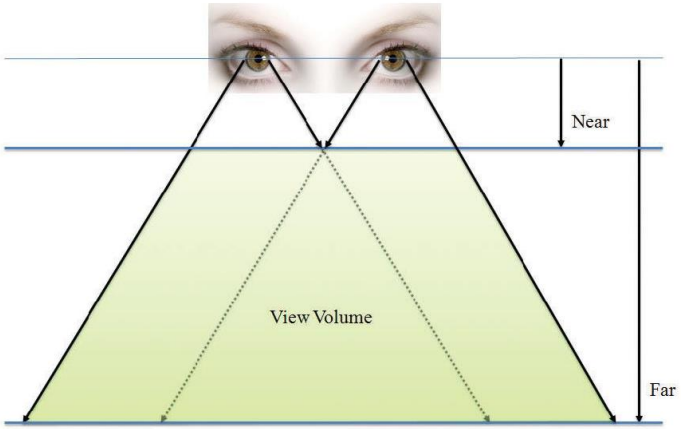


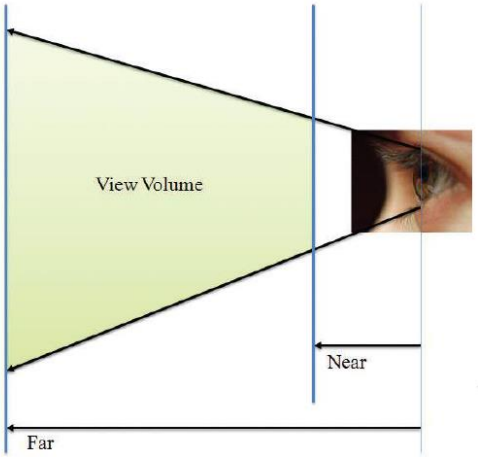
- 원근 투영의 필요성
- 보다 현실과 가깝게 만들기 위해서는 보는 사람 입장에서 같은 크기의 물체라도 보는 사람과 가까이 있는 물체는 크게 보이고 먼 물체는 가깝게 보이게 하는 것이 보다 사실적 이다
- 어떻게 하면 눈으로부터 멀리 있는 물체는 작게 보이고 가까이 있는 물체는 크게 보이게 할 수 있을까?
- 투영점을 사용한다

- **Perspective projection (원근 투영)**
- **가시 부피 (viewing frustum)안의 모든 물체는 눈 (eye)으로, 즉, 투영점 (center of projection, apex) 으로 수렴 (투영) 된다**
- **Viewing frustum안의 같은 모양, 크기의 물체가 있을 때 z축으로 서로 다른 위치에 있다면 다른 크기로 viewing face에 투영된다**
- **같은 크기의 물체라도 눈으로부터의 거리가 멀다면 작게 보이고 가깝다면 크게 보인다**



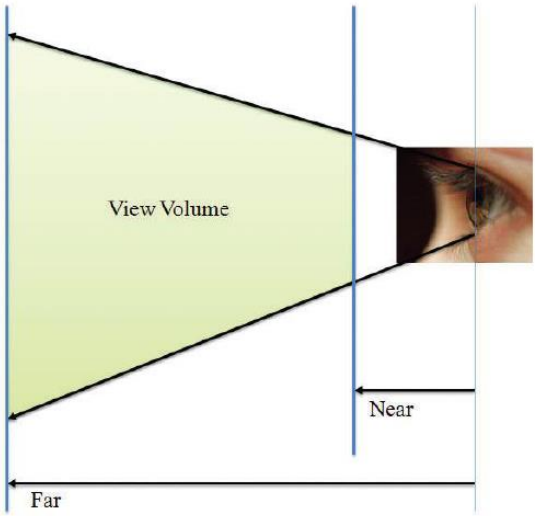
- OpenGL의 Perspective projection은 실제 사람의 눈 (카메라)과 비슷하게 가상으로 만들어져 있다
- 실제 사람의 눈은 좌, 우 가 있고 시야 각이 있다
- 어떤 물체가 카메라와 너무 가까이 있으면 상이 맺히지 않아서 어느 정도 거리가 필요하다. 두 개의 시야 각이 처음으로 만나는 위치에서부터 물체를 제대로 볼 수 있다 (눈으로부터 near만큼 거리)
- 또한, 사람의 눈 (카메라)은 무한히 멀리 있는 물체까지 볼 수 있는 것이 아니므로 최대 볼 수 있는 한계를 정해놓는다 (눈으로부터 far 만큼 거리)



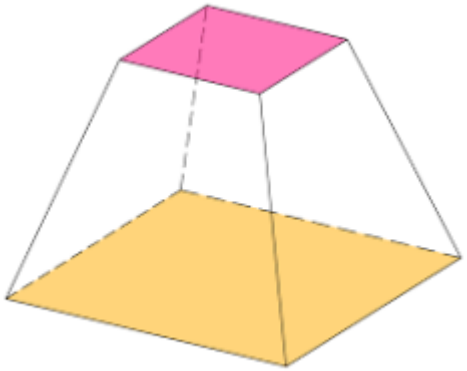


- Perspective projection에서는 이와 같이 눈으로 볼 수 있는 가상의 영역을 viewing frustum이라 한다(why frustum ? truncated pyramid)
- Orthogonal projection의 viewing box와 모양이 다르다
- 단, viewing box와 마찬가지로 물체가 viewing frustum안에 있어야 물체가 보인다고 가정한다

Viewing frustum

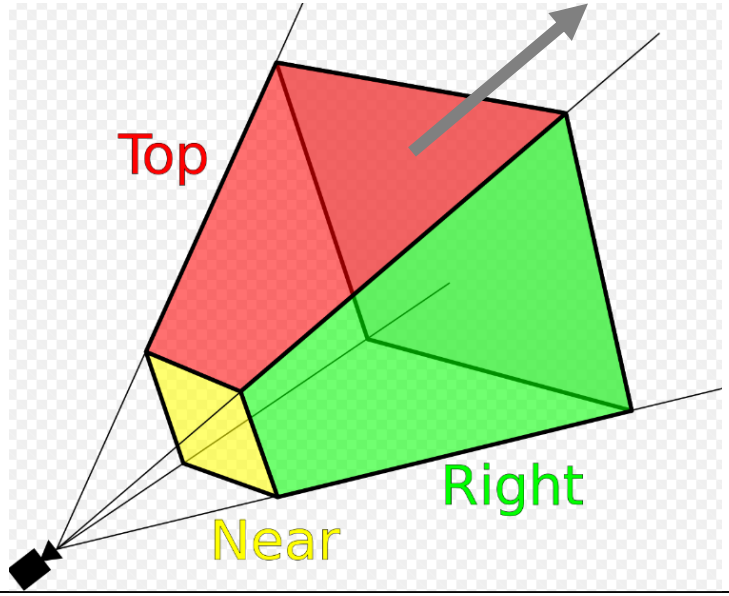


- **Viewing frustum의 형태: 사람 시야의 좌, 우, 위 아래의 시야 각을 모두 고려하면 아래와 같은 frustum (잘라진 피라미드) 형태의 viewing frustum이 만들어 진다**
- **원근 투영 시에 물체는 이 viewing frustum안에 있어야만 보인다고 가정한다**

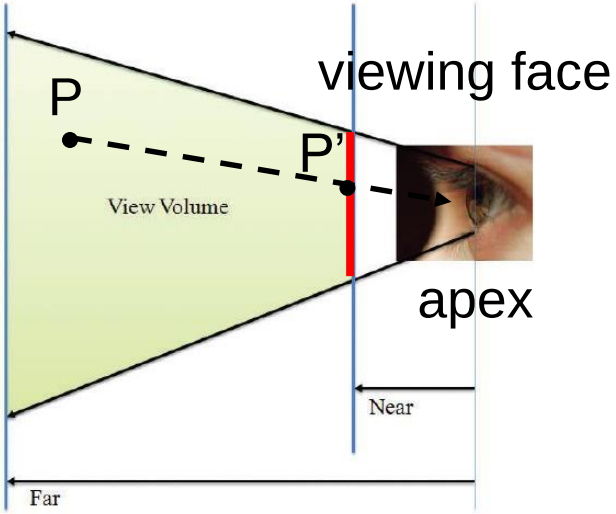


frustum

viewing frustum



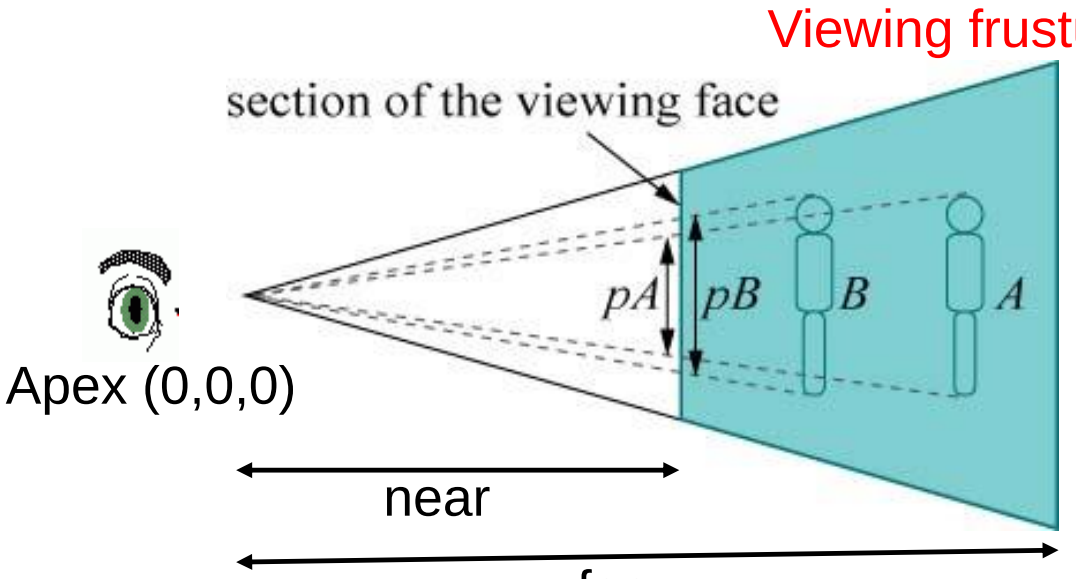
- 가정: 물체는 두 눈의 시야 각이 최초로 만나는 (z=-near)위치에 투영(projection)된다고 가정한다
- 이러한 원근 투영 (perspective projection)시 viewing frustum 안의 물체는 물체와 눈의 위치(apex)를 연결한 선이 z=-near와 만나는 위치(viewing face)에 투영된다
- 아래 그림에서 점 P는 P'로 투영된다



OpenGL 원근 투영의 기본 가정: 사람 눈 (카메라)의 위치는 : (0, 0, 0) 이고 이를 apex라고 한다. 카메라는 –z축 방향을 바라본다고 가정한다

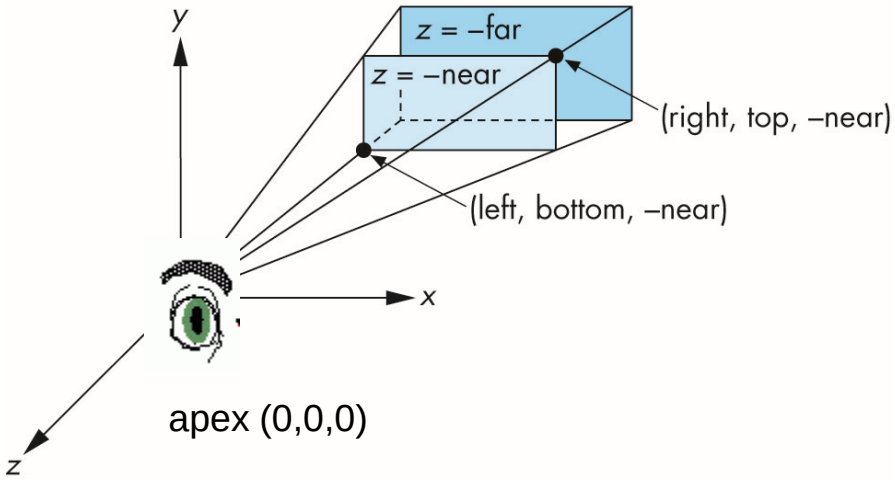
Viewing frustum의 안에 있는 물체는 카메라 (apex)와 그 물체를 연결한 선이 viewing face에 닿는 곳에 투영된다

Q) 똑같은 크기 (모양)의 2개의 물체가 frustum안에 있을 때 viewer와 가까이 있는 물체는 크게 보이고 멀리 있는 물체는 작게 보일까?

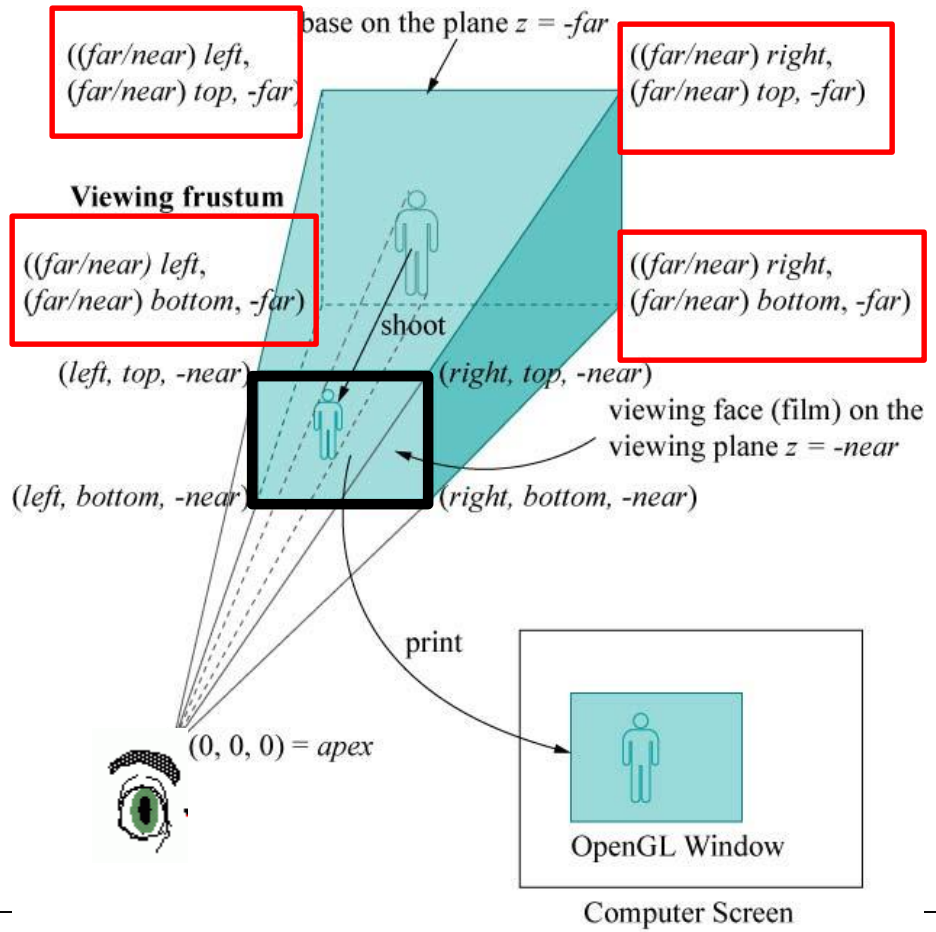


- **OpenGL에서 viewing frustum을 사용하는 방법 : glFrustum**

- viewing frustum을 그리는 방법
- viewing frustum안의 물체가 투영되는 평면: z=-near
- 닮은꼴 삼각형을 이용하여 viewing frustum의 각 corner점을 구해보자



- **glFrustum(left, right, bottom, top, near, far)**



glOrtho vs glFrustum

차이: 뒷면에 있는

네 corner 점의 (x,y)위치가

(far/near)배 만큼

커짐

- 단, glFrustum에서 near와 far값은 모두 양수여야 하고 near<far 값이어야 한다

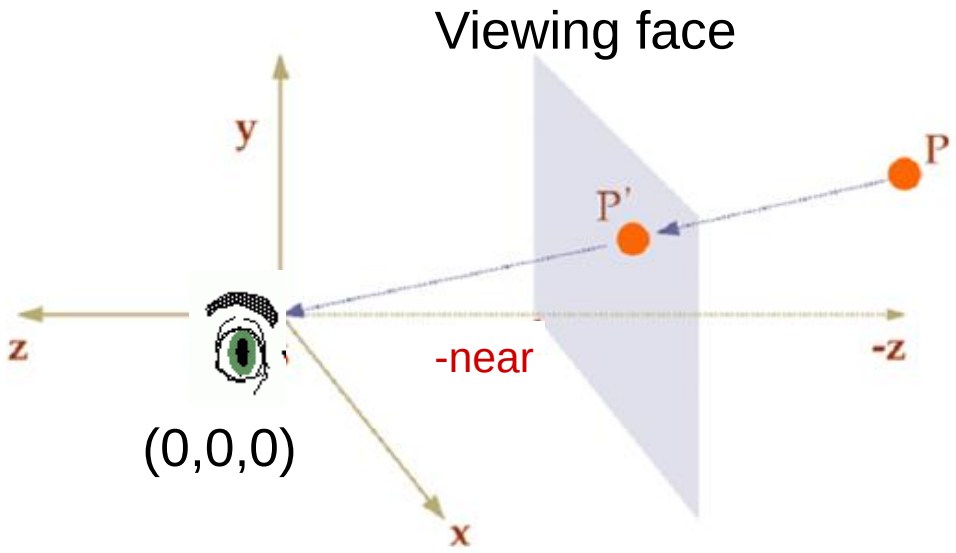
- Now, let’s draw a viewing frustum of
- glFrustum(-15.0, 15.0, -10.0, 10.0, 5.0, 50.0)

- **원근 투영시 투영되는 위치**

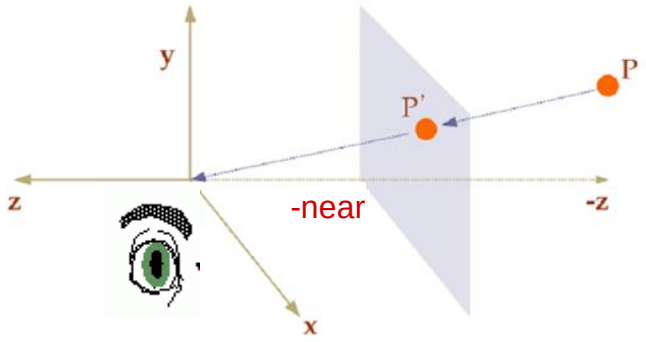
- 그렇다면 viewing frustum안의 어떤 점 (P)이 viewing face의 어디 (P’)에 투영되는지 생각해보자
- 앞에서 봤지만 원근투영에서는 viewing face에서 멀어질수록 물체가 축소되어 보인다. 그렇다면 물체가 얼마나 축소되어 보이는지 생각해보자
- Q) 극단적으로 어떤 물체가 Z=-far에 위치한다면 viewing face에서 얼마나 축소되어 보일까?

Viewing Frustum안의 어떤 점 P=(x,y,-z) 이 viewing face에 있는 P'=(x', y', z')로 투영 된다고 할때 P'의 좌표를 구해보자. 아래 그림과 같이 Viewing face의 z 좌표가 z=-near 라면 z'=-near.

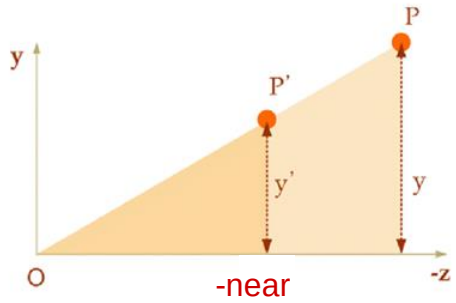
- P'=(x', y', -near) 이므로 x'와 y' 만 구하면 된다



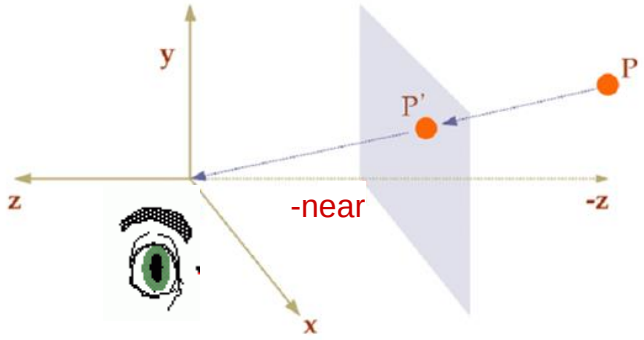
- P=(x,y,-z) , P'=(x', y', z')=(x',y',-near) 이것을 옆에서 바라보면



- 삼각형의 닮은꼴 공식 사용: z:y = near : y' => y'=(near/z)y

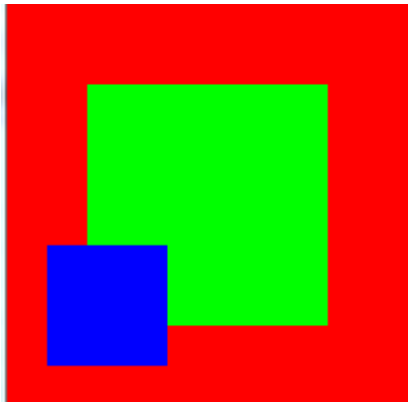


- 같은 방법으로 계산 해보면
- x’=(near/z)x
- 즉,
- P =(x, y, -z) 가 viewing face에 투영되면
- P’=(x’, y’, z’)=((near/z)x, (near/z)y, -near)



- 예) glFrustum(0, 100, 0, 100, 5, 50) 로 주어져 있을 때
- P1: (20, 80, -10)
- P2: (20, 80, -20)
- 이 두점이 viewing face에 투영된 좌표를 구해보자
- P1’=?
- P2’=?
-

- **https://www.dropbox.com/s/psvjtwtqf5qhyaa/frustum.txt?dl=0**

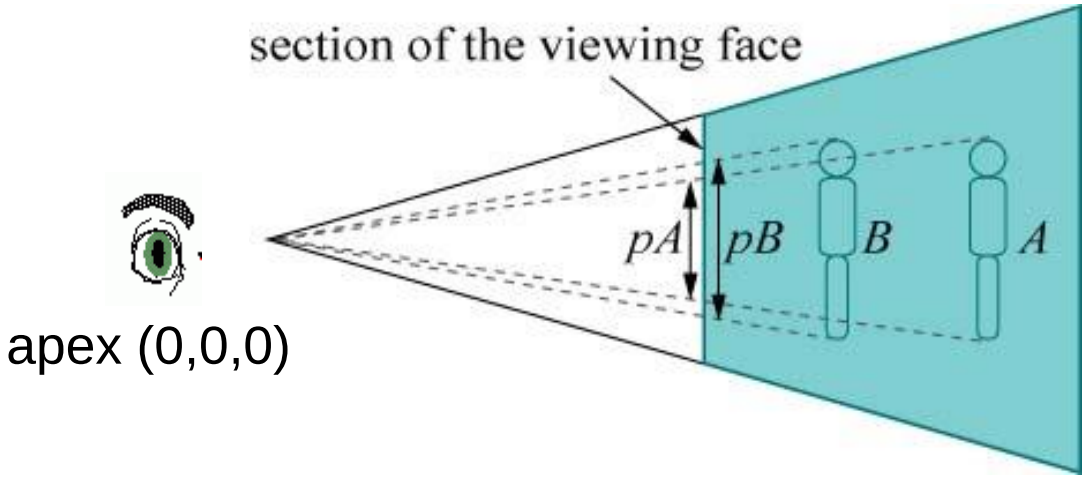


Q) glFrustum을 사용시 near 와 far 값은 모두 양수만 사용한다 그 이유는?

glFrustum(0, 100.0, 0, 100.0, 5.0, 50.0);

glFrustum(-15.0, 15.0, -10.0, 10.0, 5.0, 50.0)

****



- 앞에서 보았던 glFrustum을 사용한 예제

- https://www.dropbox.com/s/sz4wfpjdtm1xsxn/frustum_helix.txt?dl=0