Computer Graphics

Prof. Jibum Kim

Department of Computer Science & Engineering

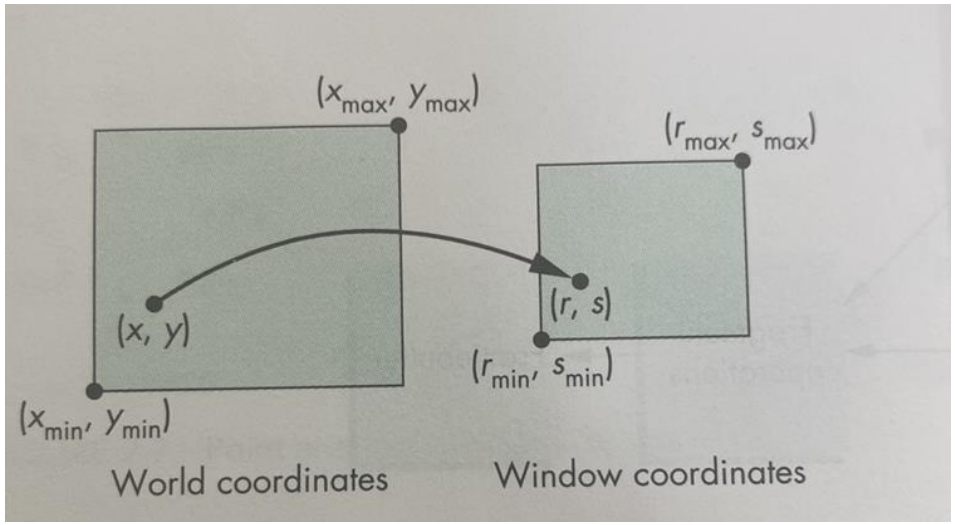
Incheon National University

- **Window**

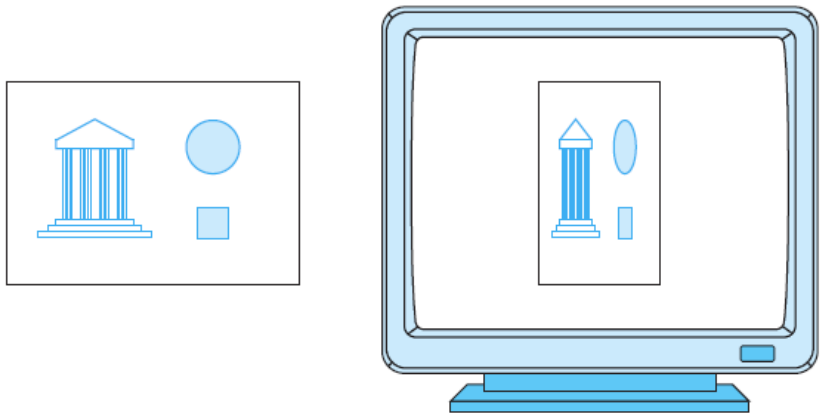
- We use window or screen (OpenGL) window to denote a rectangular area of our display

- A window has a height and width and because the window displays the contents of the framebuffer, positions in the window are measured in screen coordinates where the units are pixels

- **Mapping from world coordinates (세계 좌표) to window (screen) coordinates**
- **가시 공간안의 세계좌표로 정의된 위치는 OpenGL window의 screen coordinates로 mapping되어야 한다**



- The aspect ratio of a rectangle is the ratio of the rectangle's width to its height
- The independence of the object, viewing, and window specification can cause undesirable side effects if the aspect ratio of the viewing rectangle (가시 공간) is not the same as the aspect ratio of the window specified for the canvas
- If they differ, objects are distorted on the screen



- 예: Aspect ratio mismatch (distortion). Why?
- (a) 가시 공간 (b) window

- We can avoid this distortion if we ensure that the 가시 공간 and window have the same aspect ratio

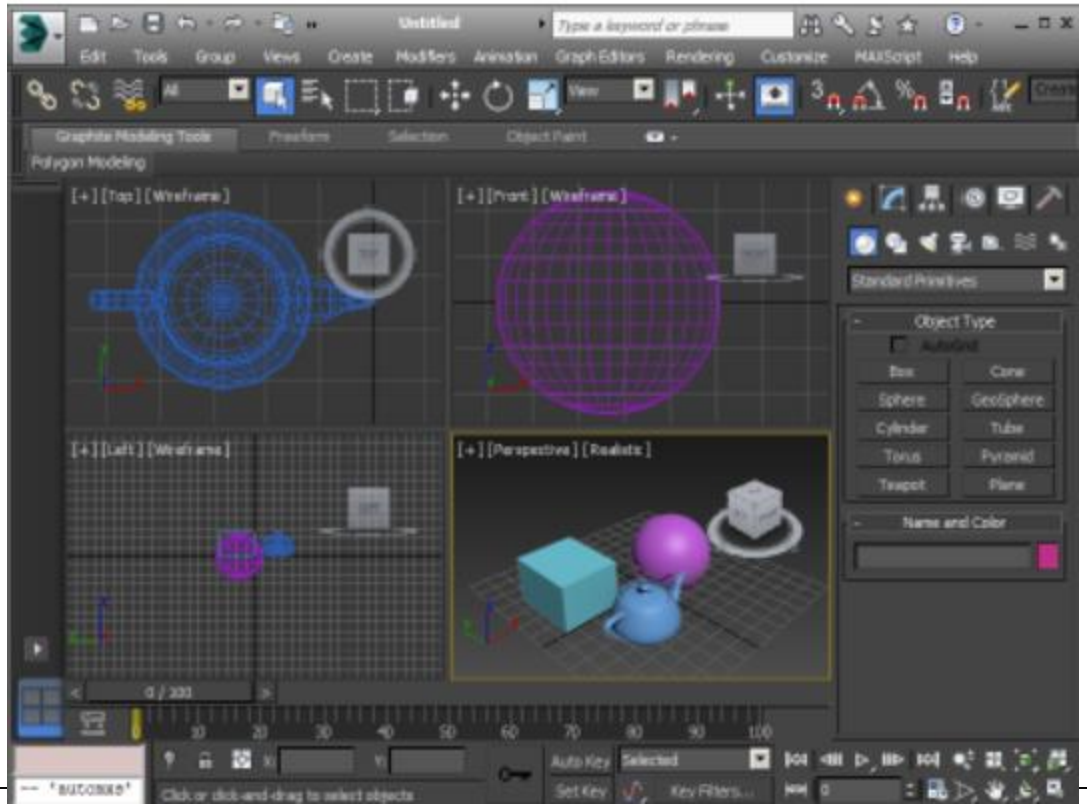
- 예: gluOrtho2D(0.0, 100.0, 0.0, 100.0)이면 window의 aspect ratio도 1:1 (1)로 맞춰줌

- Another more flexible method is to use the concept of a viewport

- **Viewport (뷰포트)**

- 지금까지는 가시공간에서 그린 물체를 window 전체를 사용하여 나타내었다
- 하지만, 어떤 경우에는 가시 공간에 그린 물체를 window의 일부만 사용하여 표현하고 싶은 경우도 있을 수 있다
- A viewport is a rectangular area of the displaying window
- By default, it is the entire window, but it can be set to any smaller size in pixels

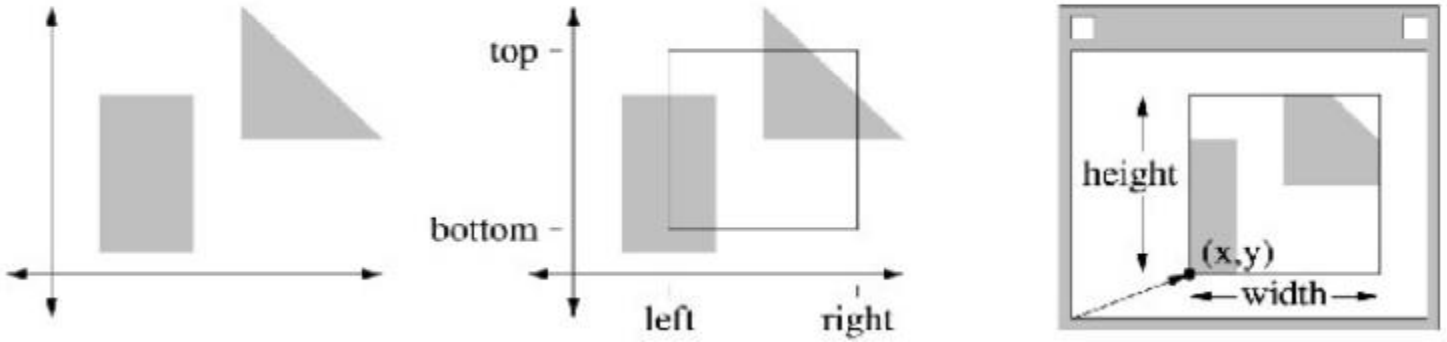
- **Viewport example**
- **3D Studio Max: divide the window into 4 viewports**



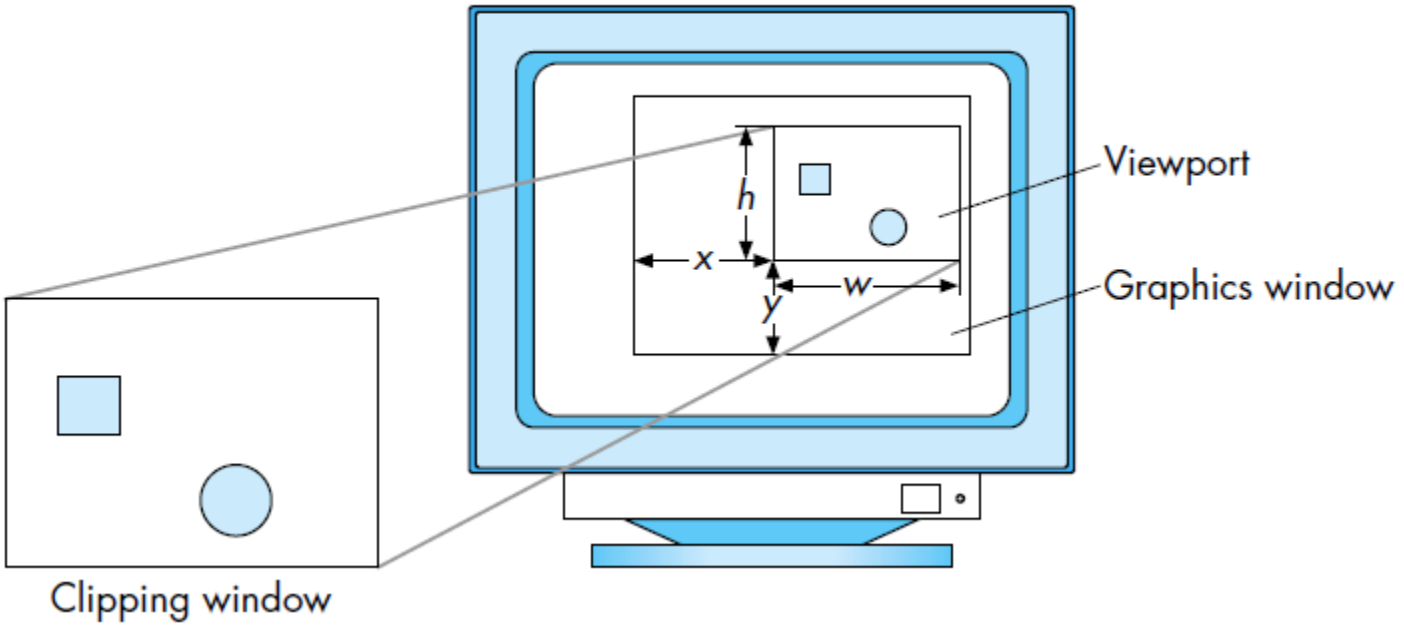
- **가시 공간과 viewport**

1. 가시 공간 밖에 있는 물체는 잘려서 보이게 된다 (clipping)

2. Viewport를 사용하면 가시 공간에서 그려진 물체는 window 전체가 아닌 viewport에만 그려지고 viewport에 mapping 된다

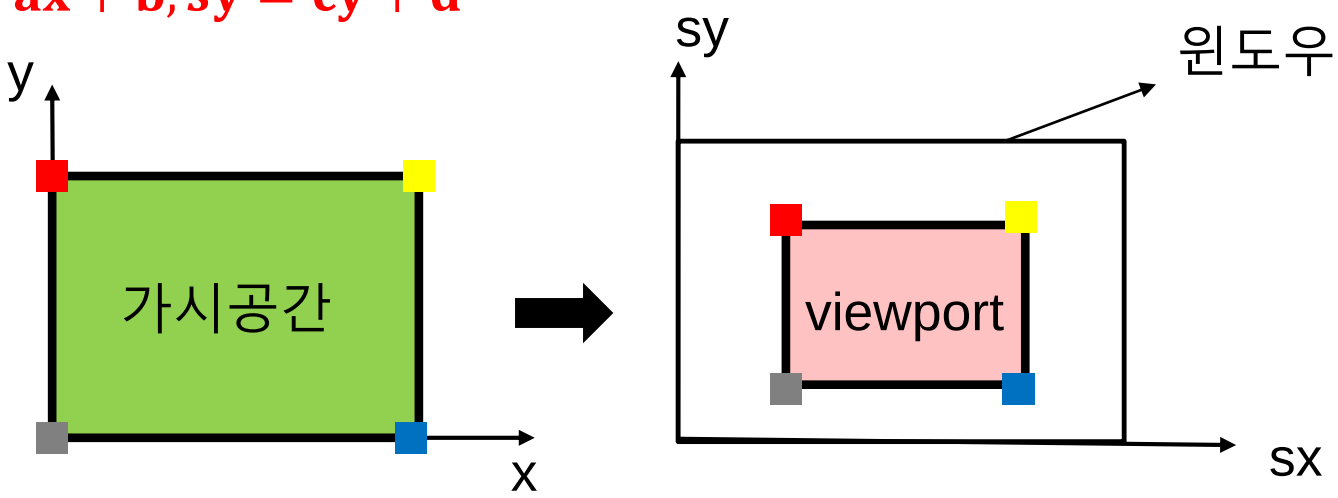


- OpenGL에서의 viewport 사용: glViewport(x, y, w, h)
- (x, y): lower-left corner of the viewport
- w: width, h: height



- 가시공간에서 viewport로의 mapping

- 가시 공간과 viewport의 크기 및 모양이 아래와 같다고 하면 가시 공간과 viewport는 모두 직사각형이므로 각각의 끝점이 각각의 끝점으로 mapping 된다고 하면 이렇게 mapping 시키는 선형 함수 (f)를 찾으면 된다. 꼭 이렇게 mapping될 필요는 없음
- $f: (x, y) \rightarrow (sx, sy)$
- 가로축과 세로축을 구분하여 각각의 mapping 함수를 찾고자 한다
- $sx = ax + b, sy = cy + d$

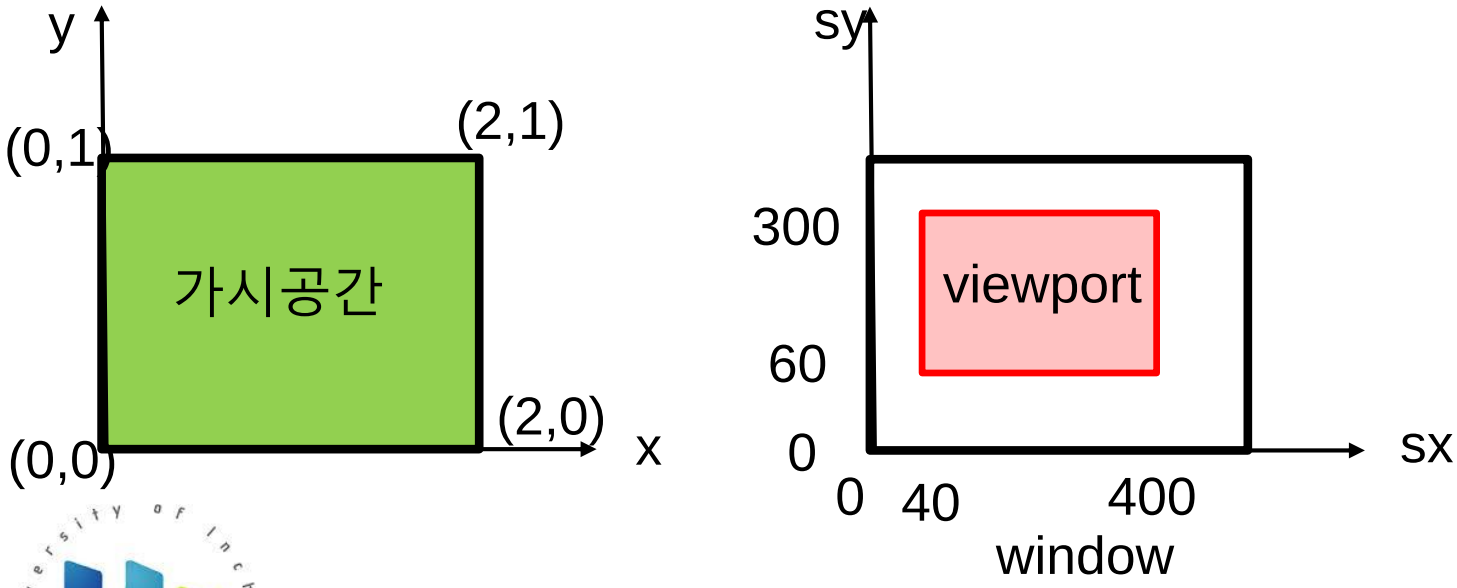


예) 가시 공간과 window 및 viewport가 주어져 있을 때 가시 공간=>viewport로의 선형 함수를 찾아보자. 가로축 선형 mapping. $Sx = ax + b, a$와 b 찾음

gluOrtho2D(0.0, 2.0, 0.0, 1.0);

glutInitWindowSize(500,400);

glViewport(40, 60, 360, 240);

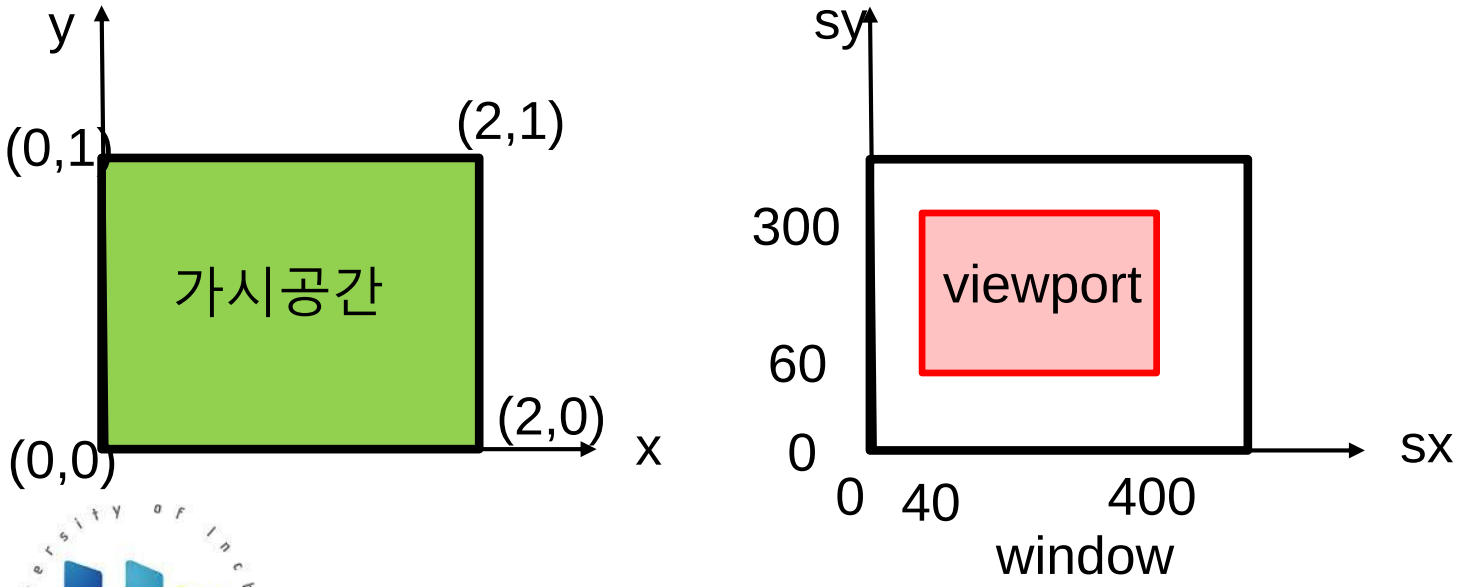


예) 가시 공간과 window 및 viewport가 주어져 있을 때 가시 공간=>viewport로의 선형 함수를 찾아보자. 세로축 선형 mapping. $Sy = cy + d, c$와 d 찾음

gluOrtho2D(0.0, 2.0, 0.0, 1.0);

glutInitWindowSize(500,400);

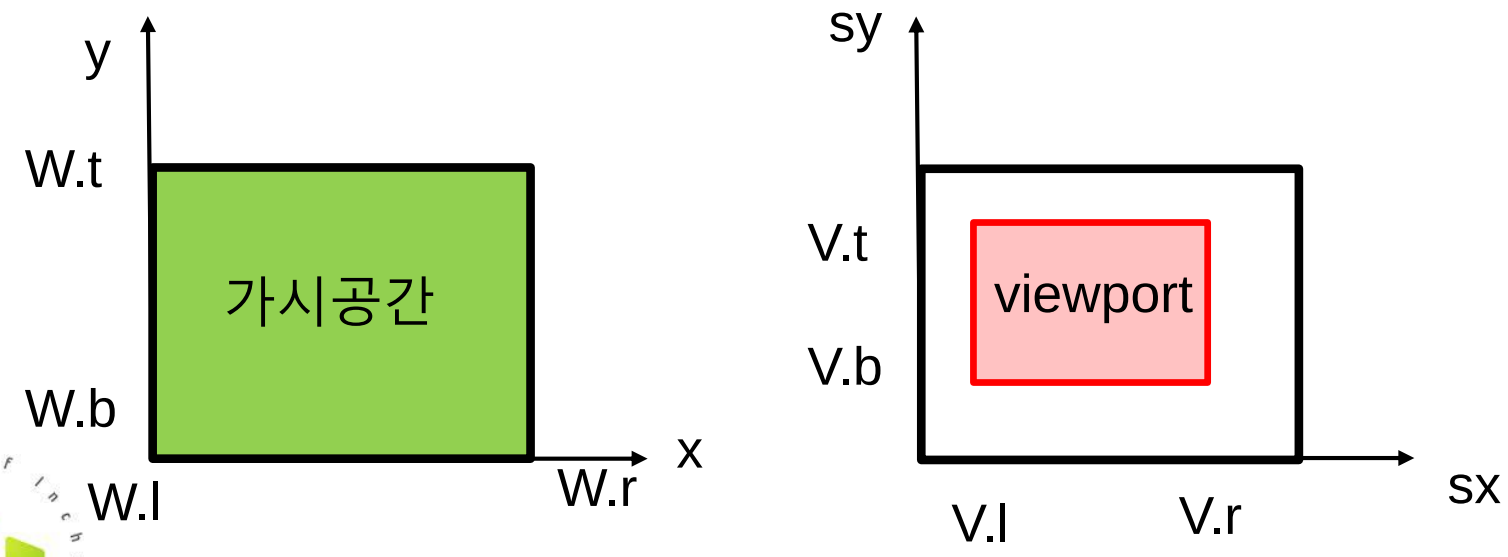
glViewport(40, 60, 360, 240);



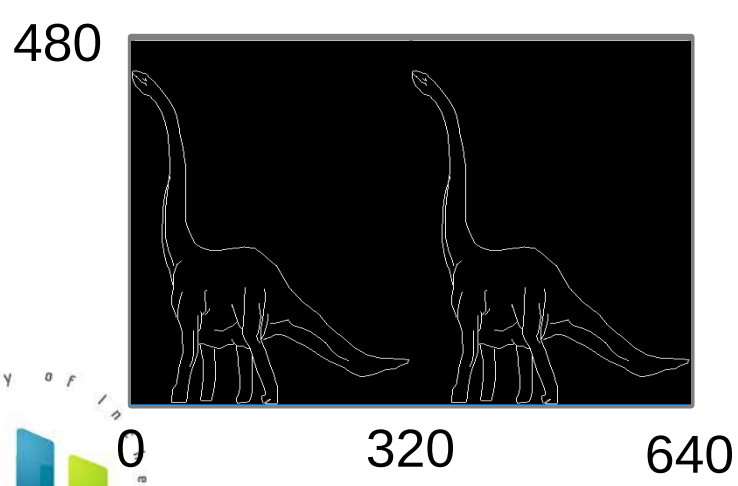
정리하면

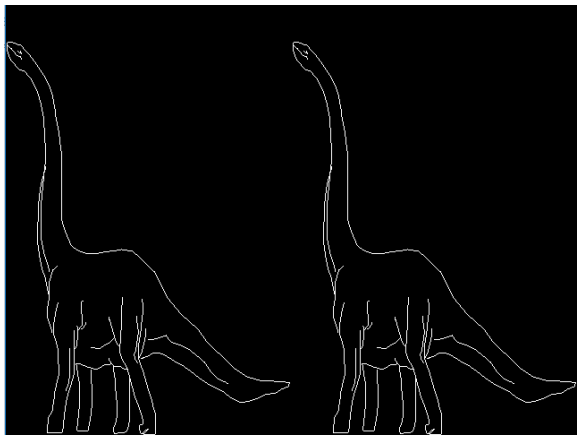
- $Sx = 180x + 40$
- $Sy = 240y + 60$

- **가시공간에서의 viewport mapping**
- $sx = Ax + c, sy = By + D$
- $A = \frac{V.r - V.l}{W.r - W.l}, C = V.l - A * W.l$
- $B = \frac{V.t - V.b}{W.t - W.b}, D = V.b - B * W.b$

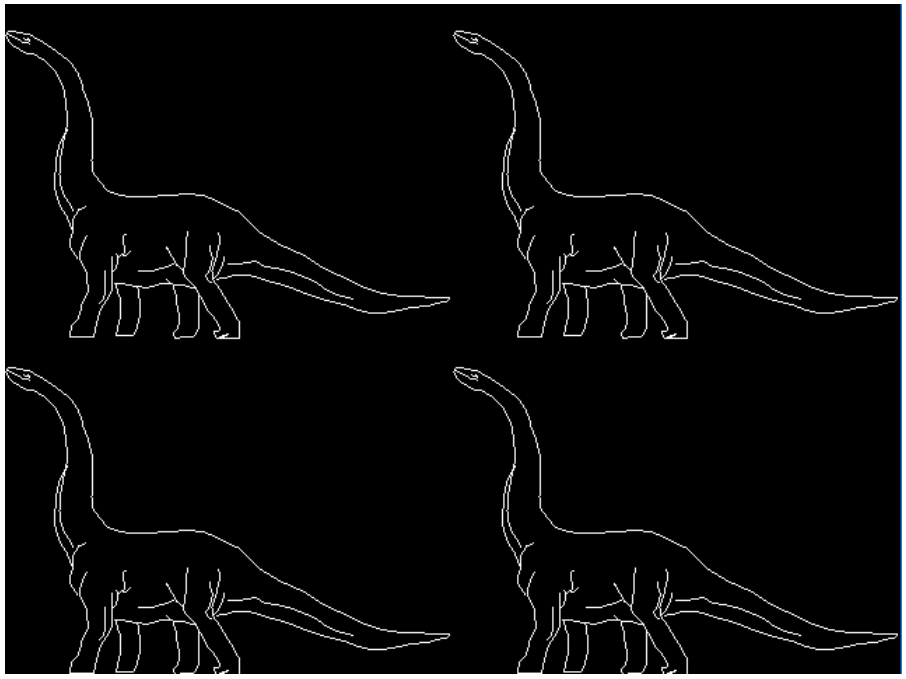


- 예) 앞에서 실습했던 dinosaur 예제를 OpenGL의 glViewport 함수를 2번 사용하여 다음과 같이 만들어 보았다
- glViewport(x, y, w, h)
- 아래와같이 screen window를 동일한 크기의 2개로 분할한 viewport를 설정하였다
- https://www.dropbox.com/s/bco8whsacgmc99k/dinosaur_1.txt?dl=0





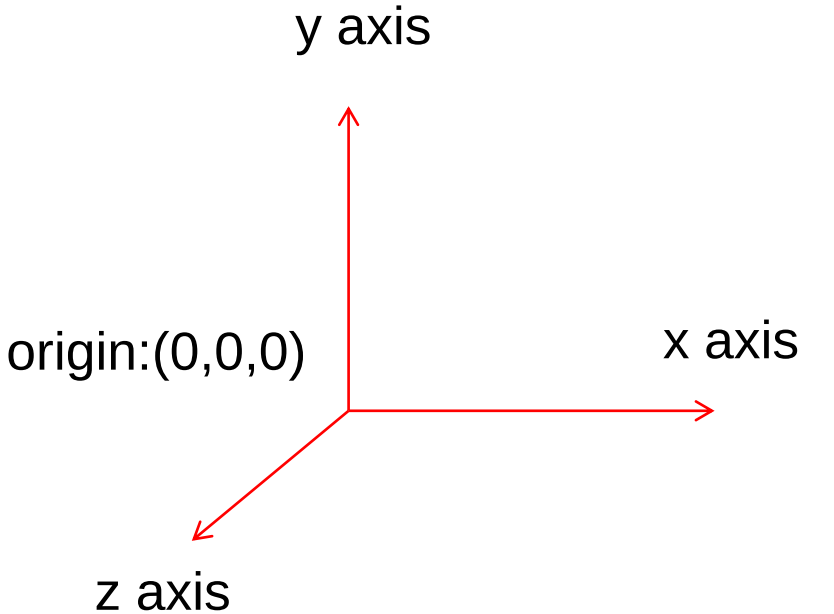
- 예: 이번에는 viewport를 여러 번 설정하여 다음과 같이 dinosaur가 4번 반복되어 보이게 만들어보자
- Nested for loop를 사용해 보자

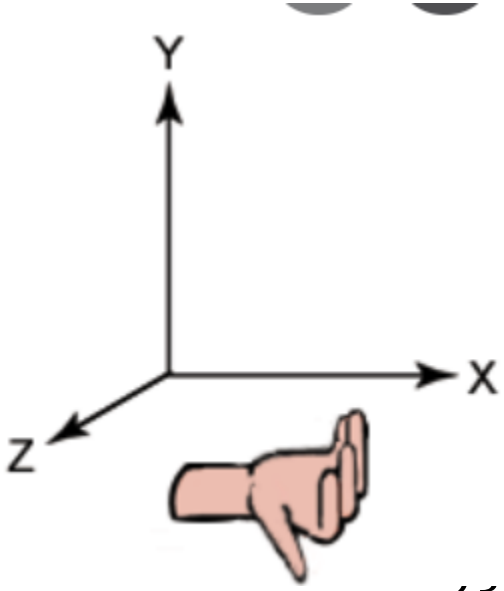


- Screen window에 dinosaur를 복사해서 여러 개의 카피본을 넣어서 만든 이와 같은 것을 tiling이라 한다. 아래 예
- 반복문을 사용하면 손쉽게 tiling이 가능하다. 어떻게 가능할까?

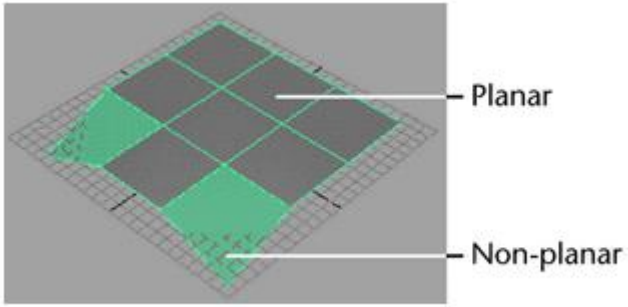
- **Orthogonal projection (직교 투영)**
- **가시 부피 (viewing box)**

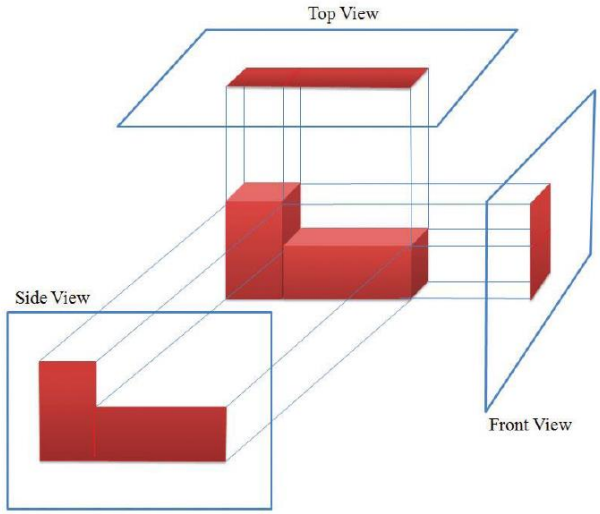
- **3차원 좌표계: 직교 좌표계 (Cartesian coordinate system)를 사용. 원점에서 서로 직각으로 교차하는 3개의 좌표축 벡터로 이루어짐**
- **Specify direction in each axis: +z, -z, +y, -y, +x, -x**
- **오른손 법칙: +x축의 끝에서 +y축의 끝을 향해 오른손 주먹을 말아 쥐었을 때 엄지 방향이 +z축이다 (OpenGL에서 사용)**



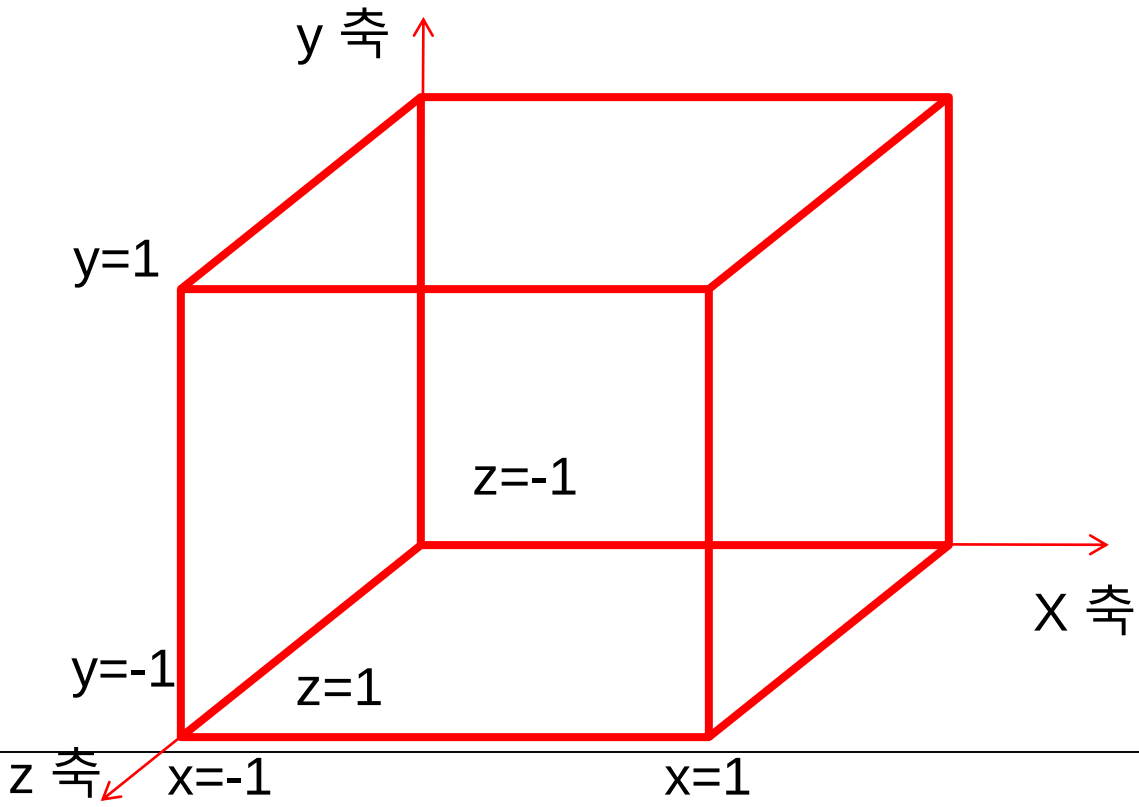


- **1. Orthogonal (orthographic) projection**
- **The simplest view and OpenGL's default, is the orthographic projection**

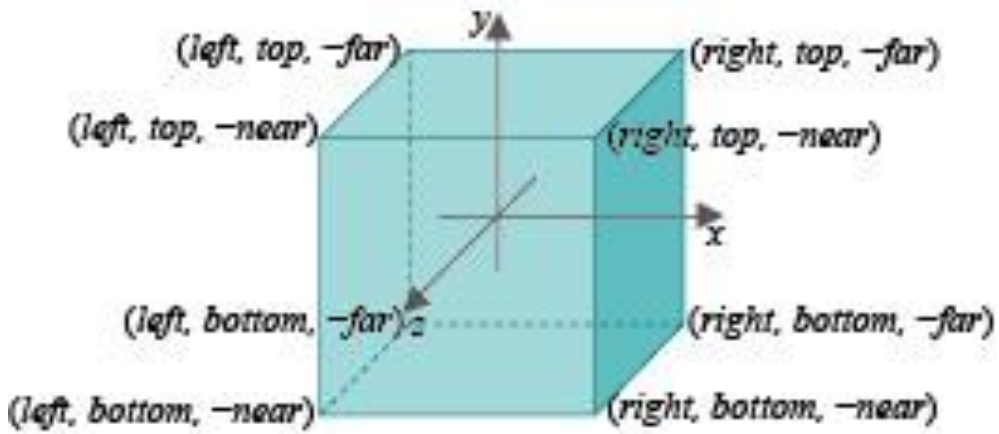




- **By default, object can be seen only if it's coordinate is specified between x=-1~1 and y=-1~1. In z-axis, it should be set between z=-1~1.**
- **This cube is called a viewing volume (가시 부피) or viewing box. Note that this is virtual !**



- OpenGL에서의 viewing volume 및 orthogonal projection
- 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

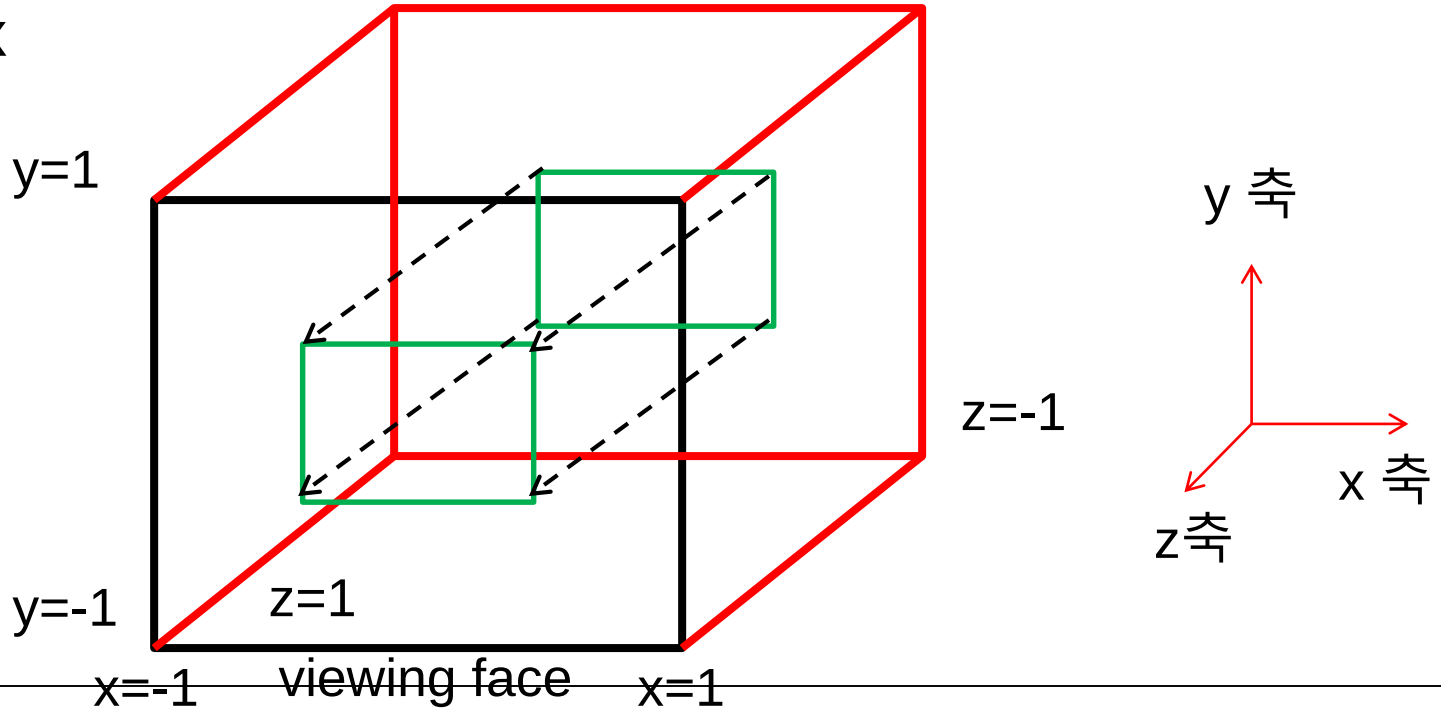


- For example, let
- glOrtho(-1, 1, -1, 1, -1, 1),
- and draw a viewing volume (box)
- also specify each corner

Viewing volume (box)를 그려보자

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

- **If the projection statement is glOrtho() as in square.cpp, then the rendering process is two-step**
- **1. Shoot: first, objects are projected perpendicularly onto the front (viewing) face (z=-near) of the viewing box**



- **최초 square.cpp에서의 shoot 예**
- **glOrtho(0.0, 100.0, 0.0, 100.0, -1.0, 1.0)**

```
glBegin(GL_POLYGON);
glVertex3f(20.0, 20.0, 0.0);
glVertex3f(80.0, 20.0, 0.0);
glVertex3f(80.0, 80.0, 0.0);
glVertex3f(20.0, 80.0, 0.0);
glEnd();
```

- **Orthogonal projection후에 polygon을 이루는 4개의 vertex의 x, y좌표는 유지되고 z값만 – near(+1)로 바뀌게 된다**

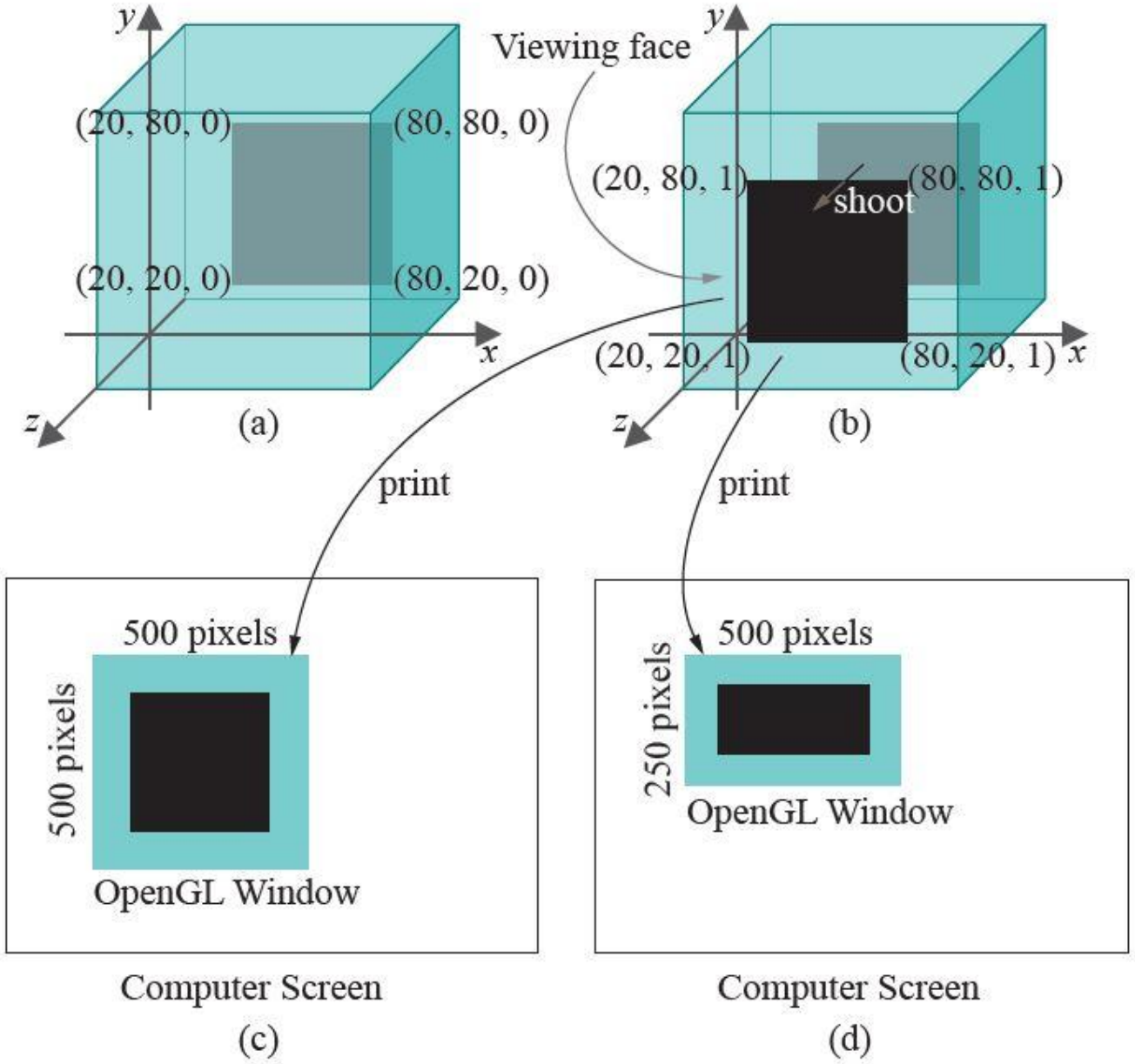


Figure 2.8: Rendering with `glOrtho()`.

- **2. Print: next, the viewing face is proportionately scaled to fit the rectangular OpenGL window. This step is like printing the film on paper**

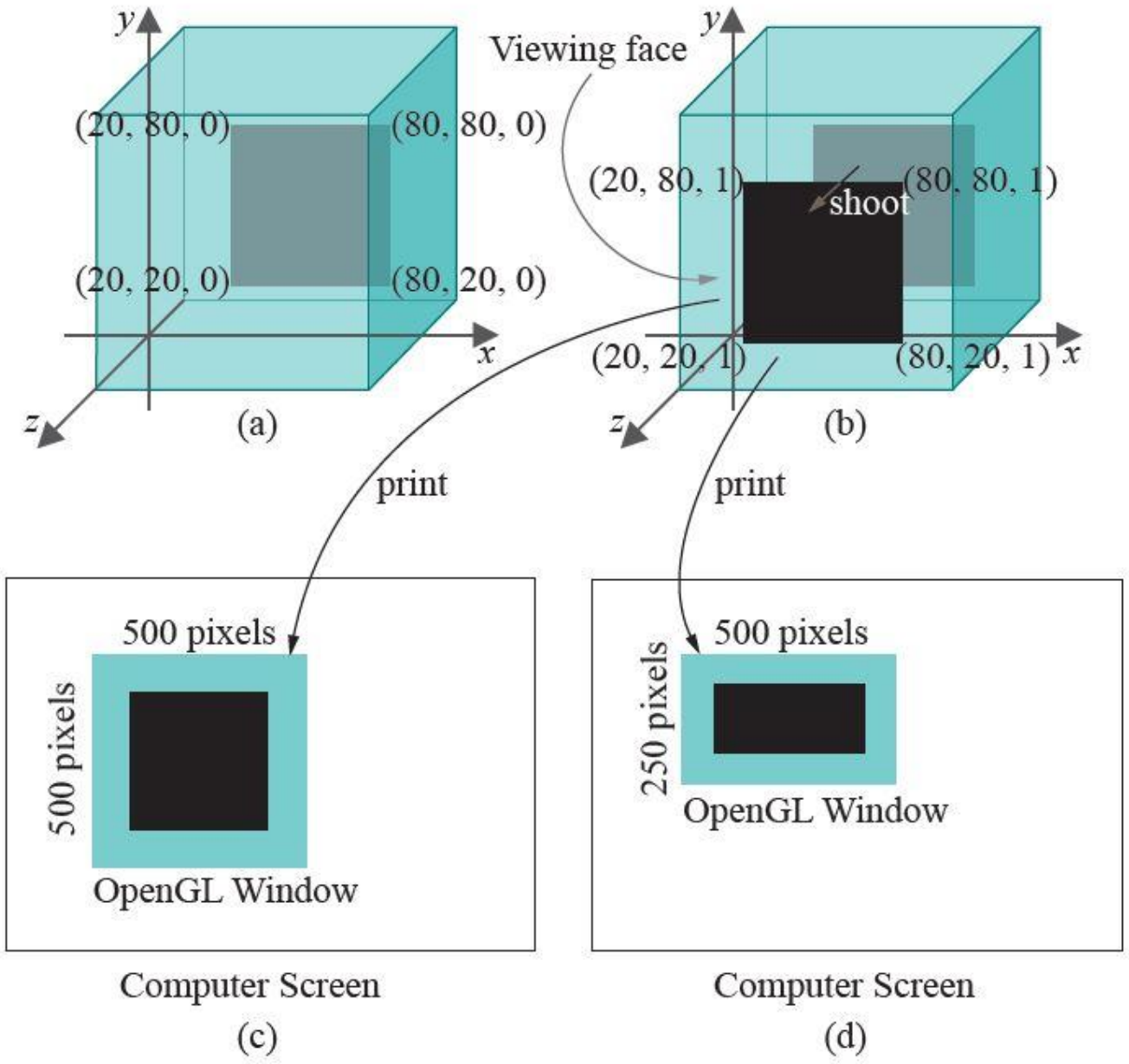
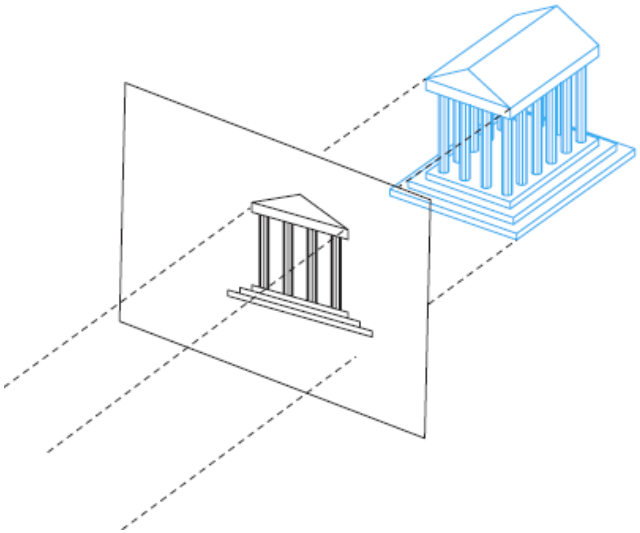
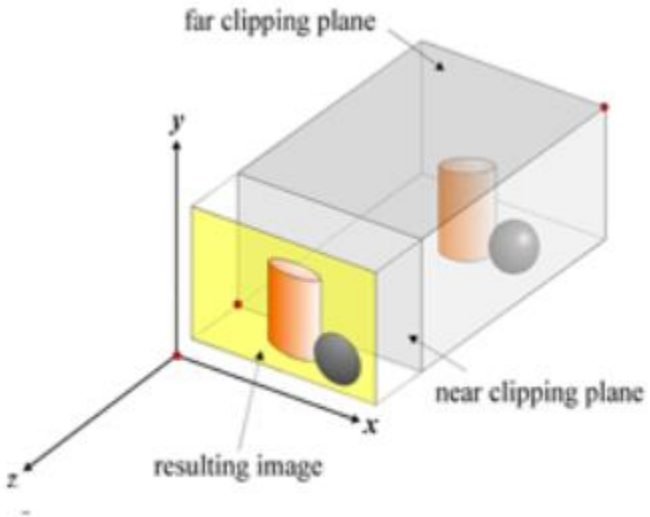


Figure 2.8: Rendering with `glOrtho()`.

- **Mathematically, the orthogonal projection is what we would get if the camera in our synthetic-camera (가상의 카메라) model had an infinitely long telephoto lens and we could then place the camera infinitely far from our objects**



- **Orthogonal projection in OpenGL**



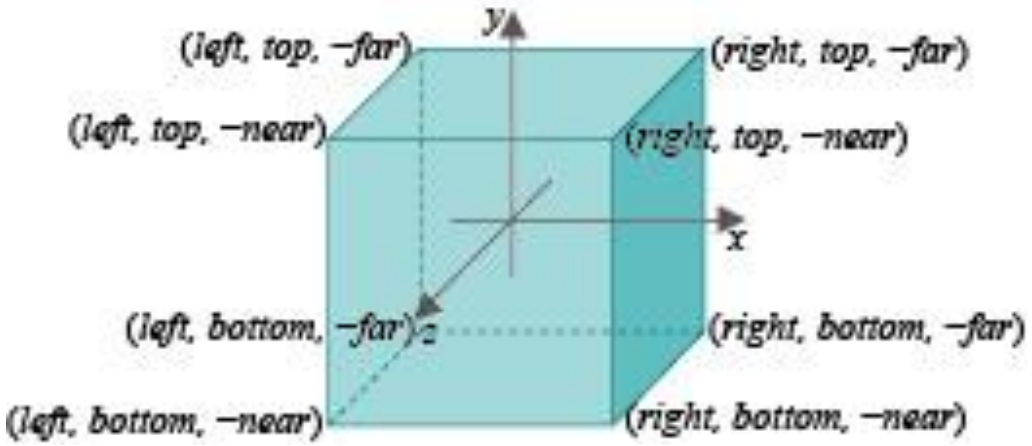
- OpenGL에서의 기본적인 viewing volume은 아래와 같은 형태의 cube이다
- x=-1~1, y=-1~1, z=-1~1
- glOrtho 함수를 이용하면 가시 부피를 변경 가능하다

- 최초 square.cpp 파일에서 ‘glOrtho’를 다음과 같이 바꿔보고 결과를 예상해보자
- glOrtho(20.0, 80.0, 20.0, 80.0, -1.0, 1.0);

- **직교 투영과 동차좌표**

- 직교 투영의 경우
- glOrtho(left, right, bottom, top, near, far)
- 으로 생성된 viewing volume안의 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

- **원근 투영의 필요성**

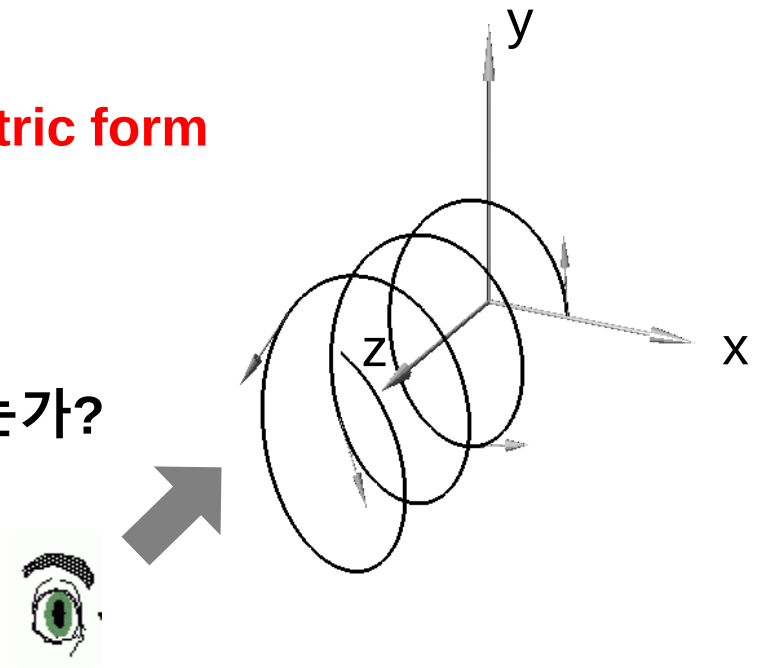
- **직교투영은 이해하기는 쉽지만 현실적이지 않다.** 사람 눈에서 멀리 떨어져 있는 물체는 가까이 있는 물체보다 작게 투영되어야 보다 현실적이다

- 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, $-10\pi \leq t \leq 10\pi$

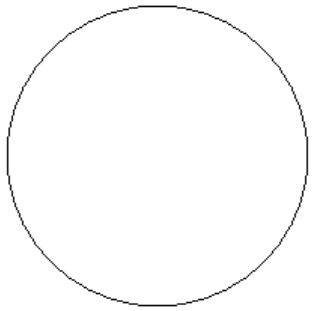
t값이 커짐에 따라 Helix는 어떻게 이동하는가?



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
- Chapter2/Helix 예제



- 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