Computer Graphics

Prof. Jibum Kim

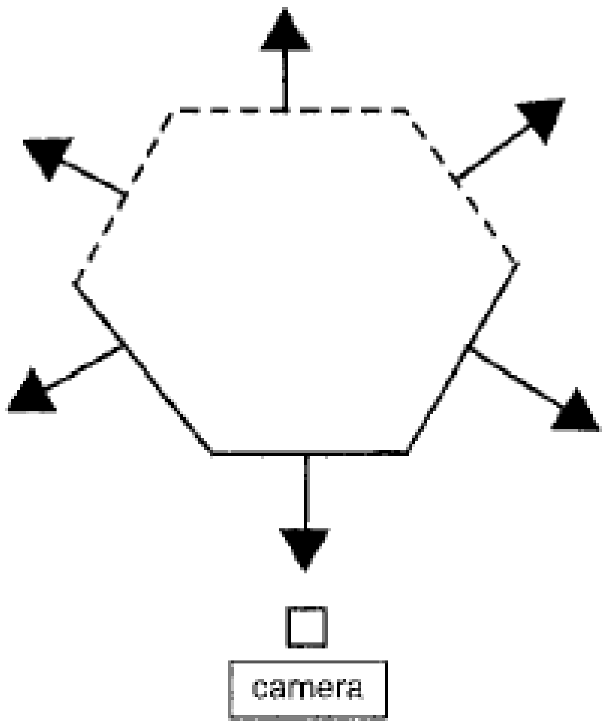
Department of Computer Science & Engineering

Incheon National University

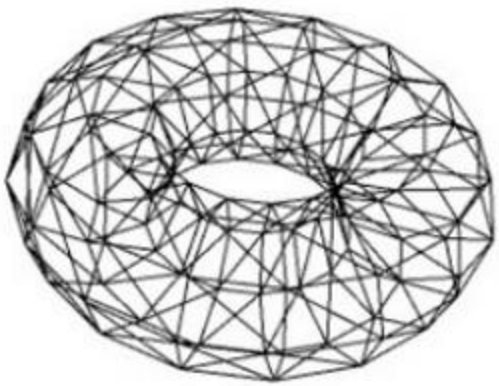
- **Vector**

- 벡터 (vector)
- (Euclidean) Vector: a geometric object that has both a magnitude and direction
- In constrast, points have position, but neither length nor direction
- Vectors correspond to various physical entities such as force, displacement, and velocity
- It is valuable to think of a vector geometrically as a displacement from one point to another

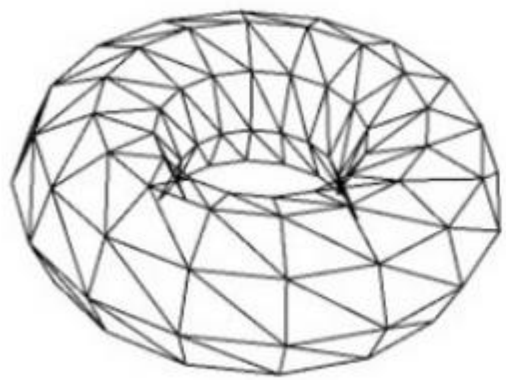
- 왜 컴퓨터 그래픽스에서 벡터가 중요한가?
- 1. Polygon에서 어떠한 면이 synthetic camera 위치에 대하여 앞면인지 뒷면 (후면, 이면)인지 따질 때 벡터를 사용한다



- 이를 통해서 **후면 제거 (backface culling)**를 수행 한다
- 아래와 같이 후면 제거를 하면 어떠한 점이 좋을까?

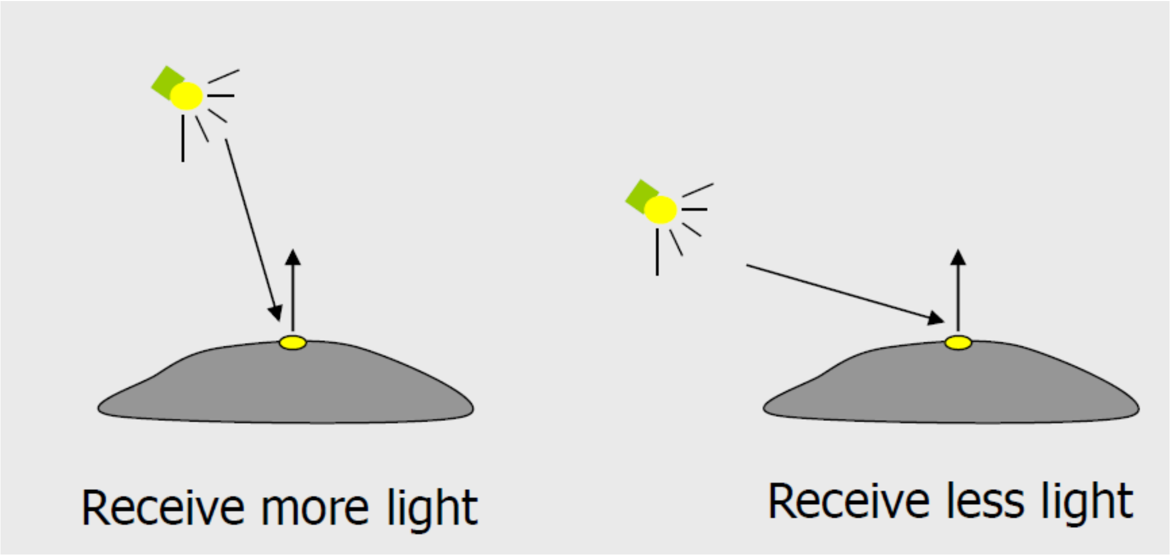


Torus drawn in wire-frame
without back face culling

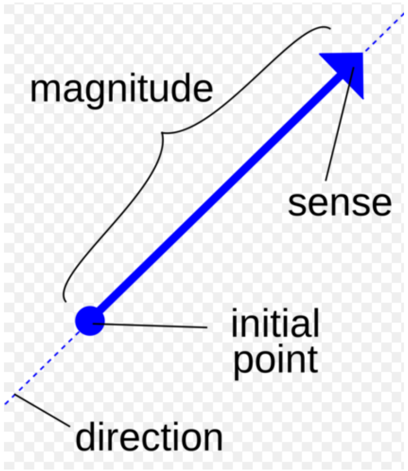


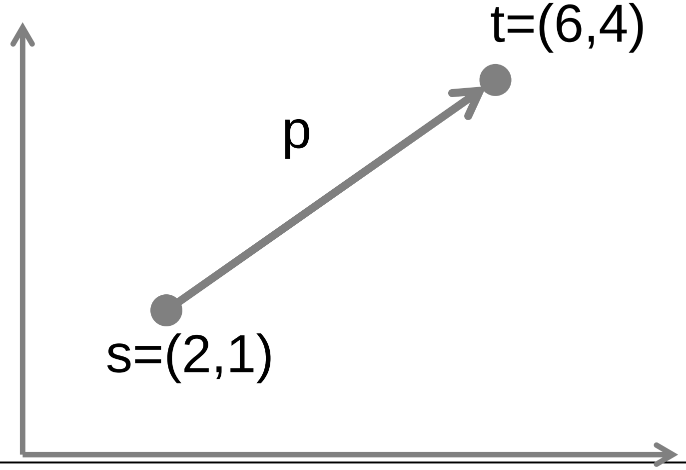
Torus drawn in wire-frame
with back face culling

- 2. 조명 (lighting) 처리시 필수적이다
- 광원 (빛이 나오는 곳)으로의 빛이 어느 정도 반사되는지 계산할 때 벡터를 사용한다

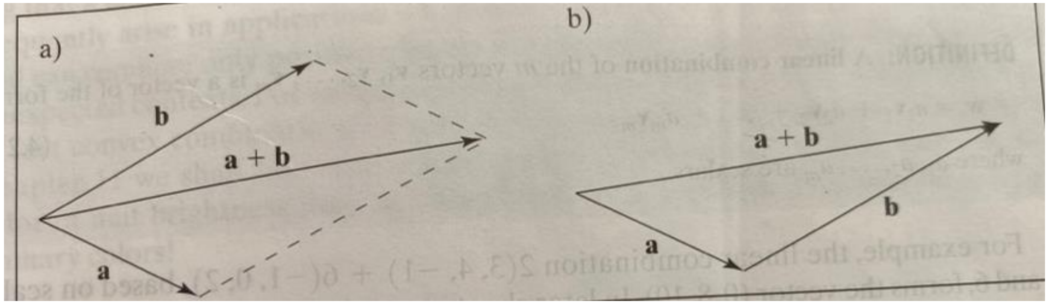


- **Vector의 magnitude: distance from its tail to its head**
- **예) p: vector from point s to point t: p=t-s=(6,4)-(2,1)=(4,3)**
- **Magnitude=**$\sqrt{4^2+3^2}=5$
- **Vector는 보통 bold (강조) 혹은 italic, *p*, 혹은 화살표 표시** $\vec{p}$

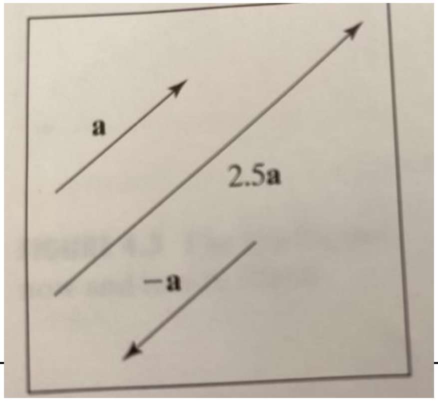




- **Vector의 표기**
- **n-차원 벡터는 n-tuple로 표기**
- $w=(w_1, w_2, \ldots, w_n)$

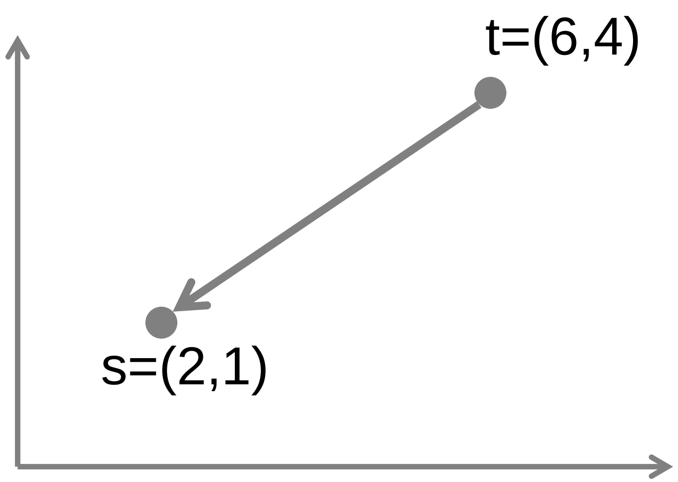


- **Vector의 합**
- **a=(2, 5, 6) and b=(-2, 7, 1)**
- **a+b=(0, 12, 7)**

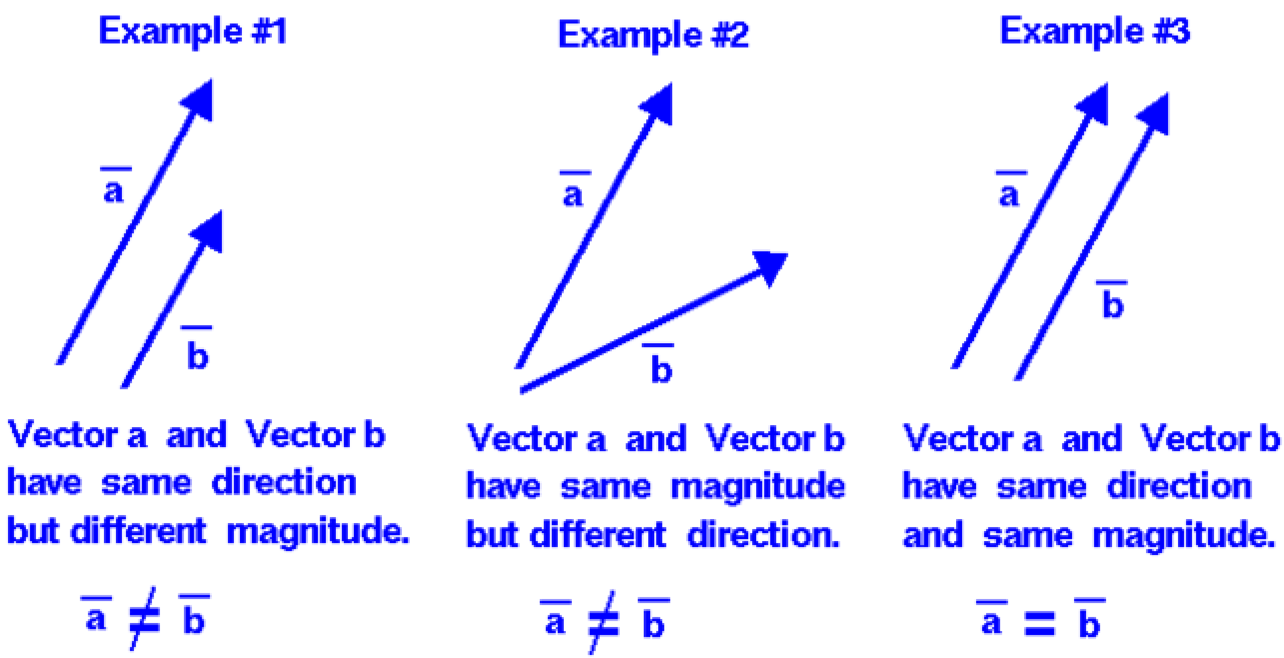


- **Vector와 scalar와의 곱**
- **6a=(12, 30, 36)**

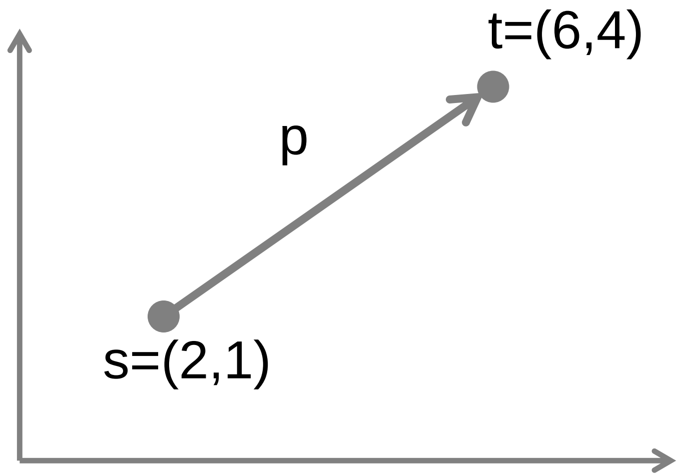
- Vector has a direction
- 예) vector from point t to point s: q=s-t=(2,1)-(6,4)=(-4,-3)
- Magnitude=$\sqrt{(2-6)^2+(1-4)^2}=5$
- 즉, 방향이 바뀌면 (반대 방향이 되면) 부호가 바뀐다
- s->t로 가는 vector: p=(4,3),
- t->s로 가는 vector: q=(-4,-3)
- p≠q (opposite direction)
- Vector 표기: italic, 진하게, 화살표표시

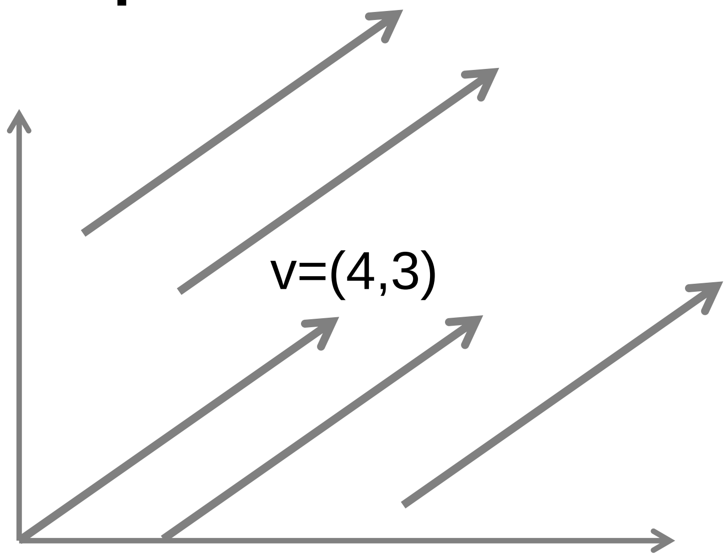


- Vector의 특징
- 두 vector는 크기 (magnitude) 와 방향 (direction) 이 모두 같아야 같은 vector이다



- Vector p from s to t : left
- These vectors are same as vector p

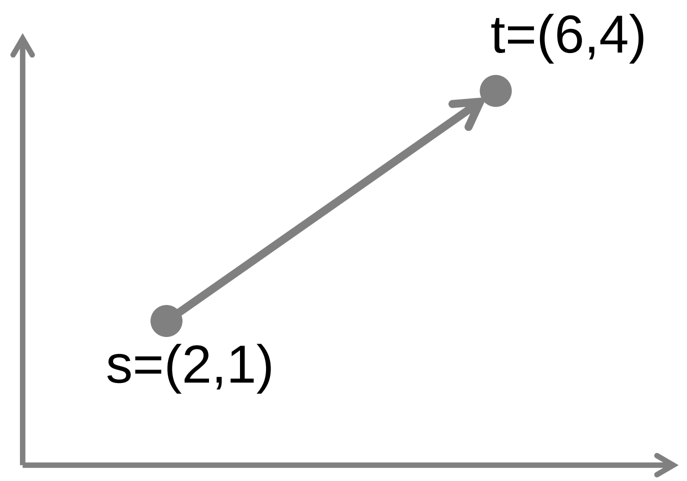




- Normalized vector (정규화 벡터)
- In general, we use a normalized vector
- Vector with a magnitude 1 (크기가 1인 vector)
- 3차원 vector p(x, y, z)의 magnitude, |p|=$\sqrt{x^2+y^2+z^2}$
- normalized vector: p'

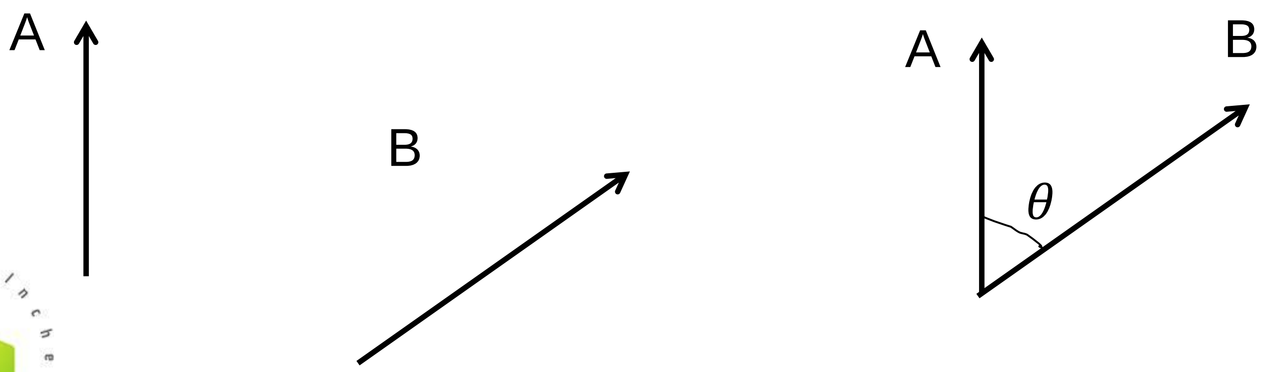
$$\boldsymbol{p}' = \left(\frac{x}{|\boldsymbol{p}|}, \frac{y}{|\boldsymbol{p}|}, \frac{z}{|\boldsymbol{p}|}\right)$$
$$= \left(\frac{x}{\sqrt{x^2+y^2+z^2}}, \frac{y}{\sqrt{x^2+y^2+z^2}}, \frac{z}{\sqrt{x^2+y^2+z^2}}\right)$$

- 예) Normalized vector from point s to point t
- p=t-s=(6,4)-(2,1)=(4,3)
- |p|=$\sqrt{4^2+3^2}=5$
- p'=$\frac{p}{|p|}=(\frac{4}{5},\frac{3}{5})$

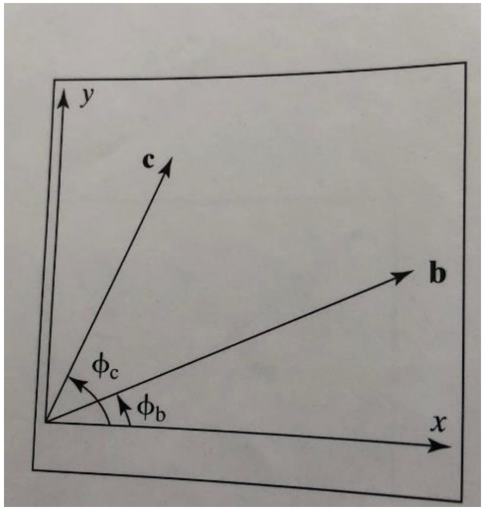


- **Dot product (Inner product) of vectors**
- **두 벡터의 내적**

- **두 vector *A*=(x1, y1, z1), *B*=(x2, y2, z2) 가 있다고 하면 두 벡터의 내적 (inner product), A • B**
- **A • B (벡터 A,B 사이의 내적)=x1*x2+y1*y2+z1*z2**
- **중요: 두 벡터 내적의 결과는 scalar 이다**
- **또한, 교환 법칙 A • B =B • A 도 항상 성립한다**



- 두 벡터 b 와 c사이의 사이각, θ , 구하기
- Vector b and c lie at angle φ_b, φ_c relative to the x-axis
- $b = (|b|cos\ \varphi_b, |b|sin\ \varphi_b)$
- $c = (|c|cos\ \varphi_c, |c|sin\ \varphi_c)$
- $b \bullet c = |b||c|cos\ \varphi_b cos\ \varphi_c + |b||c|sin\ \varphi_b sin\ \varphi_c$
- $b \bullet c = |b||c|cos(\varphi_c - \varphi_b)$
- $b \bullet c = |b||c|cos(\theta)$
- $cos(\theta) = \frac{b \bullet c}{|b||c|}$



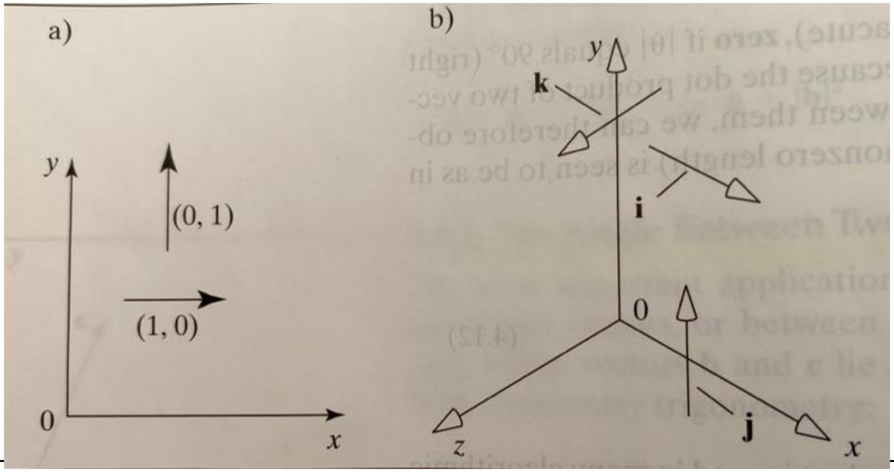
- 다음 두 벡터 *b*=(3,4)와 *c*=(5,2)사이의 각도를 계산해 보자

- 두 벡터 사이의 내적값을 통해 두 벡터의 사이각에 대해서 알 수 있다
- 두 벡터 A, B의 사이각: $\boldsymbol{\theta = cos^{-1}(\frac{A \cdot B}{|A||B|})}$, $\mathbf{cos(\boldsymbol{\theta}) = \frac{\boldsymbol{A \cdot B}}{|\boldsymbol{A}||\boldsymbol{B}|}}$
- Properties of inner product when two vectors are given
- 1. A • B=0, θ=90°
- 2. A • B >0 , 0<θ<90° (예각)
- 3. A • B <0, θ> 90 (둔각)

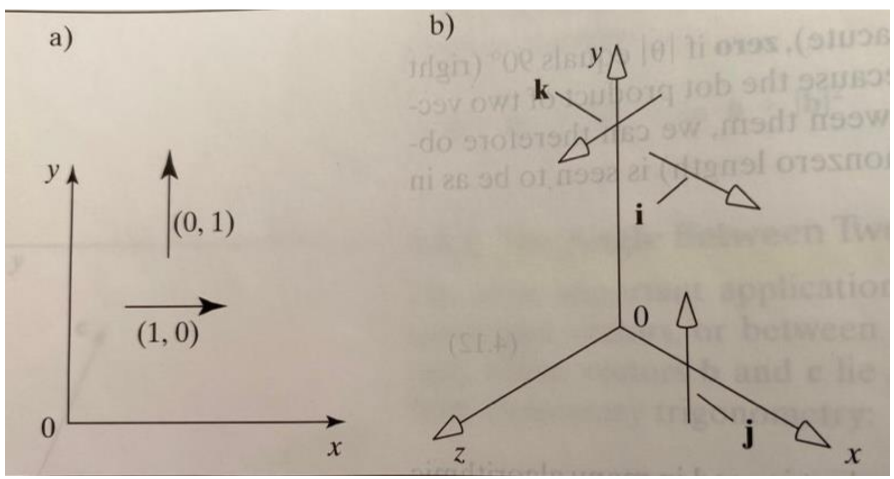
- **벡터 내적에 관련된 특징들**
- **(a) non-zero vectors u and v are perpendicular if and only if $\boldsymbol{u} \cdot \boldsymbol{v} = \mathbf{0}$**
- **(b) 교환 법칙: $\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{v} \cdot \boldsymbol{u}$**
- **(c) scalar c에 대하여 $(c\boldsymbol{u}) \cdot \boldsymbol{v} = c(\boldsymbol{u} \cdot \boldsymbol{v})$**
- **(d) 분배 법칙: $\boldsymbol{u} \cdot (\boldsymbol{v} + \boldsymbol{w}) = \boldsymbol{u} \cdot \boldsymbol{v} + \boldsymbol{u} \cdot \boldsymbol{w}$**

- Vectors b and c are **perpendicular if $b \cdot c=0$**
- Other names for perpendicular are orthogonal and normal

- The most familiar examples of orthogonal vectors are those aimed along the axis of 2D and 3D coordinate systems
- a) the 2D vectors (1, 0) and (0, 1) are mutually orthogonal unit vectors



- The standard unit vectors in 3D have components:
- *i*=(1, 0, 0), *j*=(0, 1, 0), *k*=(0, 0, 1)
- b) 오른손 좌표계에서 보여줌



- **Using these definitions,**
- **we can write any 3D vector such as (a, b, c) in the alterative form**
- **(a, b, c)= ai+ bj +ck**
- **예**
- **벡터 v=(2, 5, -1) 은 2i + 5j –k 로 표현 가능**

인천대학교

- **두 벡터의 외적 (cross product)**

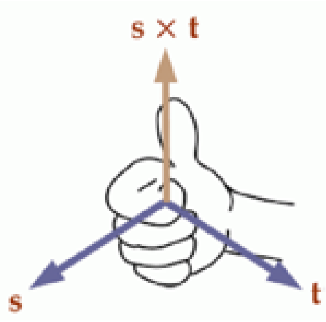
- The cross product of two vectors is another 3D vector
- The cross product is defined only for 3D vectros
- Given the 3D vectors a=(ax, ay, az) and b=(bx, by, bz), their cross product is denoted by a $\times$ b
- a $\times$ b =(ay*bz- az*by)i + (az*bx – ax*bz)j+ (ax*by-ay*bx)k
- 행렬식을 이용한 another form

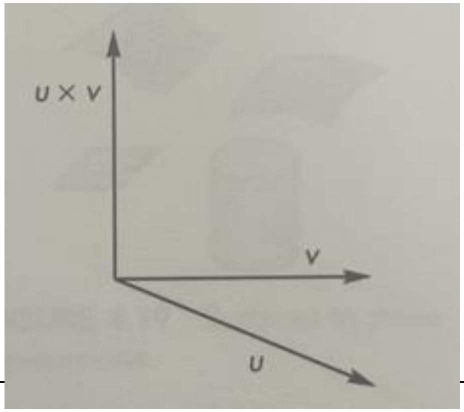
$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

- a $\times$ b = $\begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ \boldsymbol{ax} & \boldsymbol{ay} & \boldsymbol{az} \\ \boldsymbol{bx} & \boldsymbol{by} & \boldsymbol{bz} \end{vmatrix}$

- 예: 다음과 같이 두 벡터 *a*=(3, 0, 2)와 벡터 *b*=(4, 1, 8)가 주어져 있을 때
- axb=-2i-16j+3k임을 보이고 , bxa를 연산해 보자

- **벡터 외적 (cross product)의 *sxt* , 의 특징**
- 1. 벡터 *s*와 벡터 *t*의 외적 결과도 *sxt* 도 벡터이다
- 2. 벡터 *sxt* 는 벡터 *s* , 벡터 *t*와 perpendicular (직교)하다
- 이를 이용하여 벡터 외적을 법선 벡터 계산시 사용
- 3. 벡터 *sxt* 의 방향은 오른손 법칙을 이용하면 첫 벡터 s로 부터 둘째 벡터인 t를 향해 오른손 주먹을 감싸 쥐었을 때 엄지 손가락의 방향이다 (즉, ***sxt* 와 *txs* 는 크기는 같지만 벡터 방향이 반대**이다)





- 예: *i*x*j*=*k*, (*i*=(1, 0, 0), *j*=(0, 1, 0), *k*=(0, 0, 1)), i, j, k (standard unit vector), 오른손 법칙 이용

- **glm library를 사용한 벡터 내적, 외적연산**

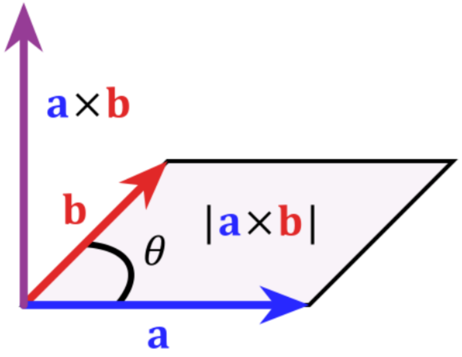
- glm에는 ‘dot’함수, ‘cross’함수, ‘normalize’함수등이 정의되어있다
- [Geometric functions (g-truc.net)]
- http://www.c-jump.com/bcc/c262c/c262samples/glm/Week08_lab4/glm_examples_cpp.htm

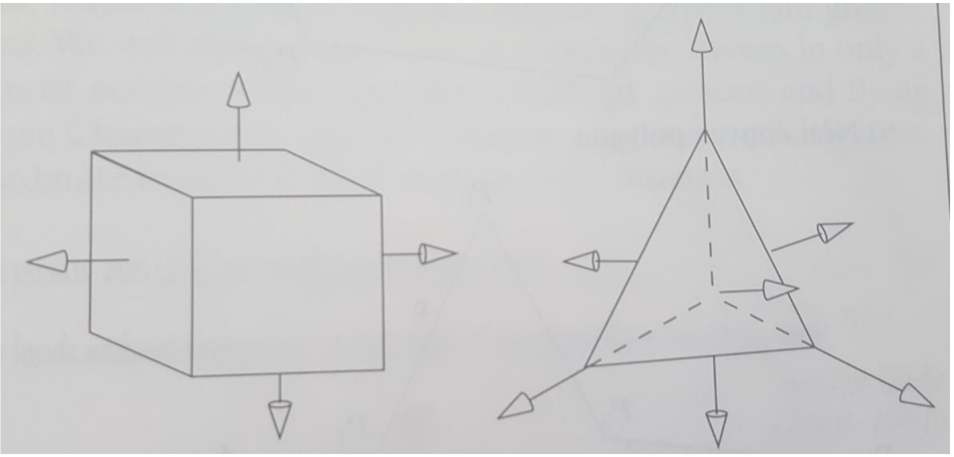
```
vCross = glm::cross(
    glm::vec3( 0.0f, 1.0f, 0.0f ), // unit-size vector along the y axis
    glm::vec3( 1.0f, 0.0f, 0.0f )  // unit-size vector along the x axis
    );
display_v3( "cross( y, x )", vCross );

glm::vec3 vA( 1.0f, 0.0f, 0.0f );
glm::vec3 vB( 0.0f, 1.0f, 0.0f );
float dot_product = glm::dot( vA, vA );
cout << "dot-product: " << dot_product << "\n\n";
```

```
#include<iostream>
#include<glm/glm.hpp>
#include <string>
using namespace::std;

void display_v3(string tag, glm::vec3 v3)
{
	cout << tag
		<< "\n| "
		<< v3.x << '\t'
		<< v3.y << '\t'
		<< v3.z << '\t'
		<< "\n"
		;
}

int main()
{
	glm::vec3 vCross = glm::cross(
		glm::vec3(3.0f, 0.0f, 2.0f), // unit-size vector along the x axis
		glm::vec3(4.0f, 1.0f, 8.0f)  // unit-size vector along the y axis
	);
	display_v3("cross( x, y )", vCross);
	glm::vec3 vA(3.0f, 4.0f, 0.0f);
	glm::vec3 vB(5.0f, 2.0f, 0.0f);
	float dot_product = glm::dot(vA, vB);
	cout << "dot-product: " << dot_product << "\n\n";
}
```

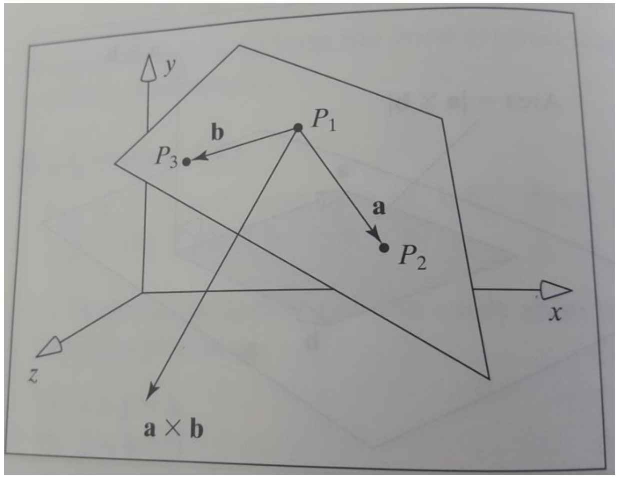
- **Normal vector (법선 벡터) to a plane**

- Normal vector (법선 벡터) to a plane (평면)
- 이는 벡터 외적을 통해서 구할 수 있다
- 두 물체에 대하여 각 face (면)에 대하여 normal vector를 표시해 보면





- 평면에 대한 방정식이 아래와 같이 주어져 있는 경우에는
- $ax+by+cz+d=0$
- 이 평면에 대한 법선 벡터 (*n*)는 *n*=(a, b, c)이다
- 평면의 방정식이 주어져 있지 않은 경우에는 그 평면에 있는 세 점 P1, P2, P3 (같은 직선 위에 있지 않은)을 찾은 후 두 벡터 *a*=P2-P1, *b*=P3-P1을 구한 후 다음의 연산을 통해서 구할 수 있다
- *n=axb*



예: find a normal vector to the plane through the points

P1=(1, 0, 2), P2=(2, 3, 0), P3=(1, 2, 4)

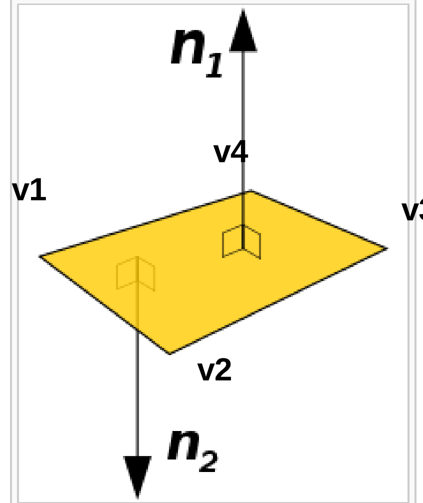
***a*=(1, 3, -2), *b*=(0, 2, 2), *n* (normal vector)**

- OpenGL에서는 vertex를 명시하는 순서에 따라서 법선 벡터의 방향이 달라지게 된다
- 컴퓨터 그래픽스에서 법선 벡터가 중요한 이유는 어떤 면이 **공간상에서 어디를 향해 있는지**를 나타내는 **면 방향 (orientation)**을 표시할 수 있기 때문이다
- 오른손 법칙을 이용하면
- **Polygon 정의 시: v1, v2, v3, v4 순서로 정의하면**

  n1 is a normal vector (윗 방향)

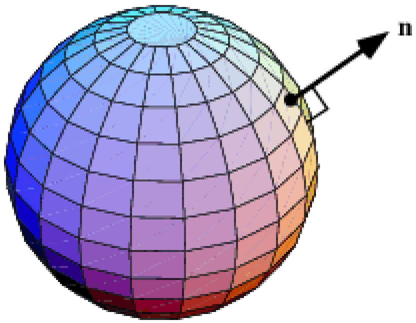
  Polygon 정의 시: v1, v4, v3, v2 순서로 정의하면

  n2 is a normal vector (아래 방향)



A polygon and two of its normal vectors

- closed surface에서의 normal vector는 outward direction만 사용한다고 생각하자
- A closed surface could be the surface area of a sphere or a cube



- **Viewing transformation**

- **OpenGL 좌표계 변환 순서**

1. Local (model) coordinate (모델 좌표계)

2. World coordinate (세계 좌표계)

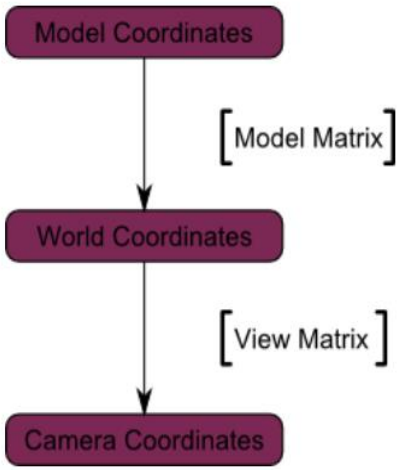
3. Eye coordinate (view, camera coordinate)

4. Clip coordinate (절단 좌표계)

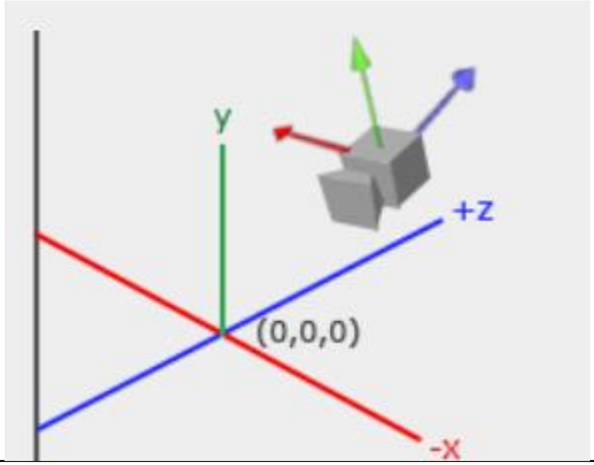
5. Normalized device coordinate (NDC)

6. Screen coordinate (화면 좌표계)

- **3. Eye coordinate**
- **(view, camera coordinate):**
- **All the vertex coordinates as seen from the camera's perspective as the origin of the scene**
- **The view matrix transforms all the world coordinates into eye coordinates that are relative to the camera's position and direction**



- **Eye coordinate system**
- **아래와 같이 camera의 중심을 원점 (origin)으로 생각하고 camera 중심으로 x, y, z축 처럼 세 개의 서로 수직 (직교)하는 세 개의 축을 가짐**
- [LearnOpenGL - Camera]()



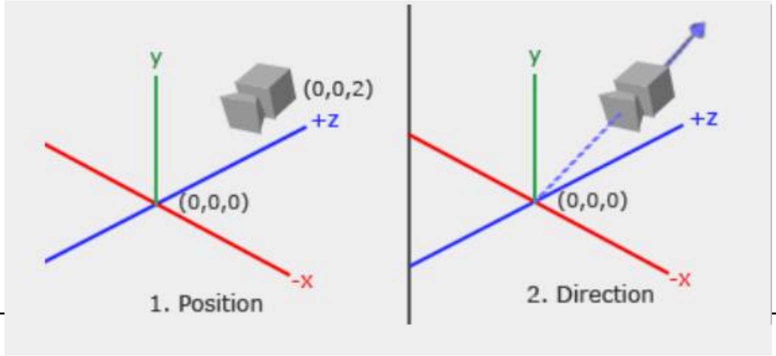
- **1. 카메라 위치**
- **카메라 위치는 세계 좌표로 줄 수 있음**
- **예: (0, 0, 3)**

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

인천대학교

University of Incheon

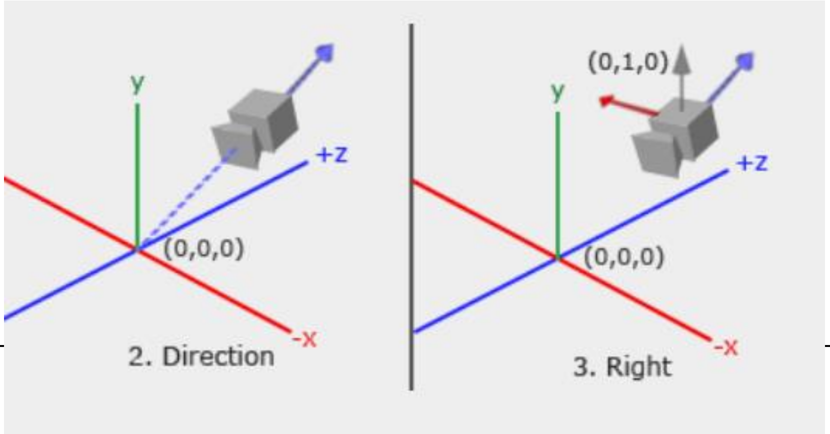
- 2. 카메라 기준 +z축 방향 (camera direction?)
- OpenGL에서는 기본적으로 카메라가 바라보는 방향을 -z축으로 설정한다
- -z축 벡터: (카메라가 바라보는 위치)-(카메라 위치)
- Reference point (카메라가 바라보는 위치), eye point (카메라 위치)
- 즉, 카메라가 바라보는 방향의 반대 방향의 벡터가 eye coordinate system에서는 +z축이다 (아래 파란색 벡터)
- 카메라 기준 +z축 벡터: (카메라 위치)-(카메라가 바라보는 위치)



- **카메라 기준 +z축 벡터:**
- **(카메라 위치)-(카메라가 바라보는 위치)**

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```

- **3. 카메라 기준 x축 (right)**
- 세계 좌표기준 up 벡터: (0, 1, 0) 정의. Why?
- 앞에서 카메라 기준 +z축 벡터
- up벡터와 +z축 벡터를 외적하여 카메라 기준 x축 구함
- **즉, (+x축 벡터) = (up 벡터) x (카메라 기준 +z 축 벡터)**
- 아래 빨간색 벡터 (오른손 법칙)

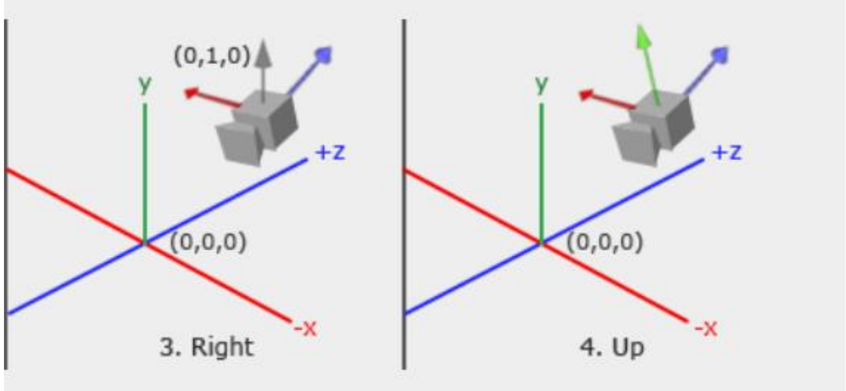


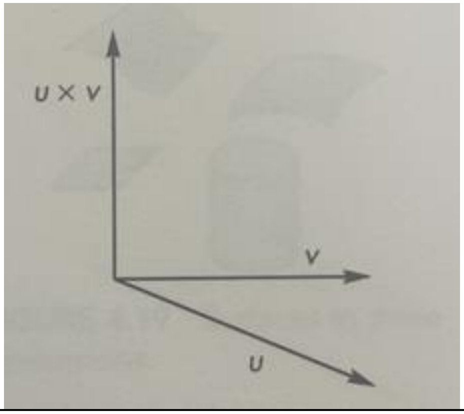
- **(+x축 벡터) = (up 벡터) x (카메라 기준 +z 축 벡터)**

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

인천대학교

- **4. 카메라 기준 +y축 방향**
- **카메라 기준의 +z축과 카메라 기준 +x축이 정해졌다면 벡터 외적을 통하여 +y축을 구할 수 있다**
- **즉, 카메라 기준으로**
- **+y축 벡터 = (+z축 벡터) x (+x축 벡터)**

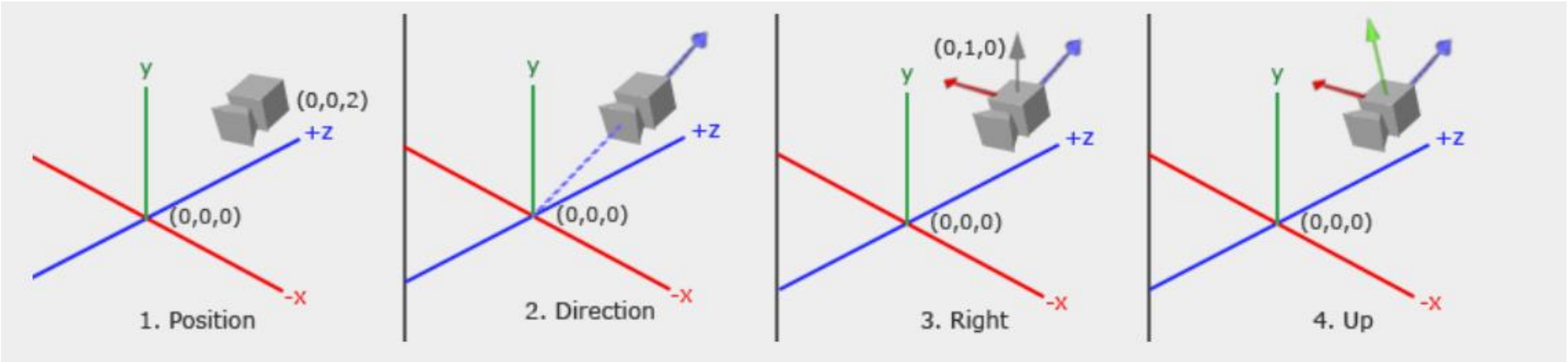




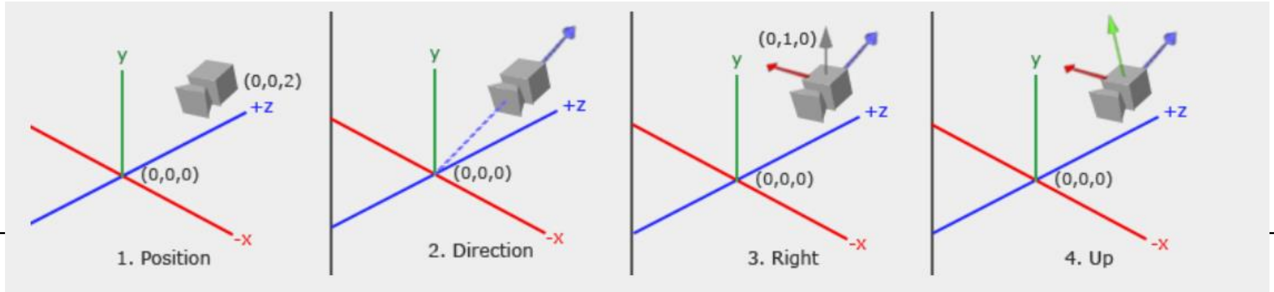
- **+y축 벡터 = (+z축 벡터) x (+x축 벡터)**

```
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

- **카메라 기준 좌표계인 eye coordinate system**



- 예: camera 위치가 세계 좌표계 기준으로 (4, 4, 4)이고 카메라가 (0, 1, 0)을 바라본다. 카메라 기준으로 +x, +y,+z축을 구해보자. 단, +x축을 구할때 세계 좌표기준 up 벡터: (0, 1, 0)를 사용하자
- 1. 카메라 기준 +z축 벡터: (4, 3, 4)
- 2. 카메라 기준 +x축 벡터 (right): (0, 1, 0) X (4, 3, 4)= (4, 0, -4)
- 3. 카메라 기준 +y축 벡터 (up): (4, 3, -4) X (4, 0, -4)=(-12, 32, -12)



- **View matrix**

- View matrix (v)는 world coordinate를 eye coordinate로 변환시키는 4x4 행렬이다
- V는 먼저 카메라의 위치를 (0, 0, 0)으로 translate시킨다. 그리고 카메라 기준의 x, y, z축을 실제 x, y, z축과 대응되도록 회전한다
- 즉, V=rotation matrix * translation matrix 형태인 4x4 행렬이다. 왜 translation먼저?
- [OpenGL Camera (songho.ca)]

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing transformation in OpenGL

- 앞에서 공부한 viewing transformation을 OpenGL에서 구현한 것이 lookAt 함수이다
- 이전 OpenGL에서는 gluLooatAt 함수로 구현되어 있고 최근 OpenGL에서는 glm librar를 통하여 glm::lookat 함수로 구현되어 있다
- gluLookAt creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.
- gluLookAt (khronos.org)

인천대학교

- The command gluLooatAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz) simulates OpenGL's camera first being moved to the location eye=(eyex, eyey, eyez), then pointed at center=(centerx, centery, centerz) and finally rotated about its line of sight (los) –the line joining eye to center – so that its up direction is one determined from up=(upx, upy, upz)
- 여기서 eye, center, up은 모두 세계 좌표

University of Incheon
인천대학교

- **gluLooatAt() function**

los= (centerx, centery, centerz)- (eyex, eyey, eyez)

los is a vector

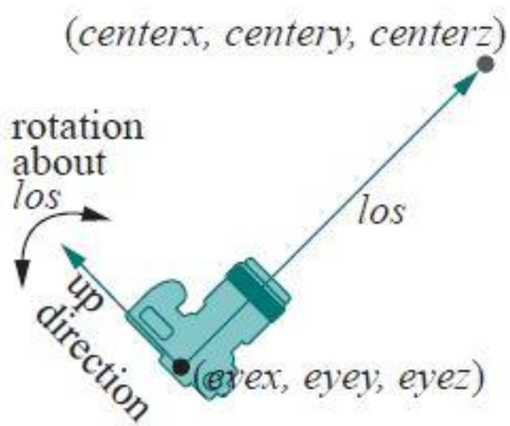


Figure 4.45: Camera pose determined by `gluLookAt(`*eyex*, *eyey*, *eyez*, *centerx*, *centery*, *centerz*, *upx*, *upy*, *upz*`)`.

- **예: gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0, 0);**

-

- **What if**
- **gluLookAt(0.0, 0.0, 15.0, 0.0, -10.0, 0.0, 0.0, 1.0, 0, 0);**

- **Chapter4/box.cpp에서**
- **glTranslatef()를**
- **gluLookAt(0.0, 0.0, 15.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)**
- **로 바꾸어보자. 결과가 달라지나? No. Why?**
- **Because its position relative to the frustum is the same in both**

- **(a) gluLookAt(0, 0, 15, 0, 0, 0, 0, 1, 0);**
- **(b) glTranslatef(0.0, 0.0, -15.0);**

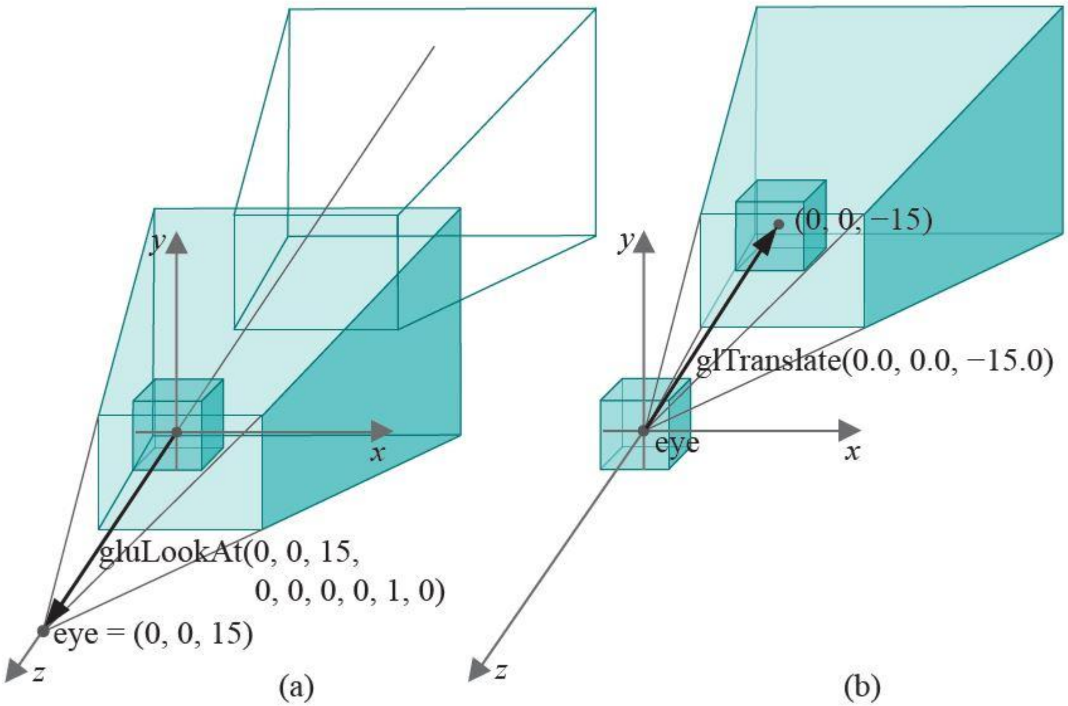
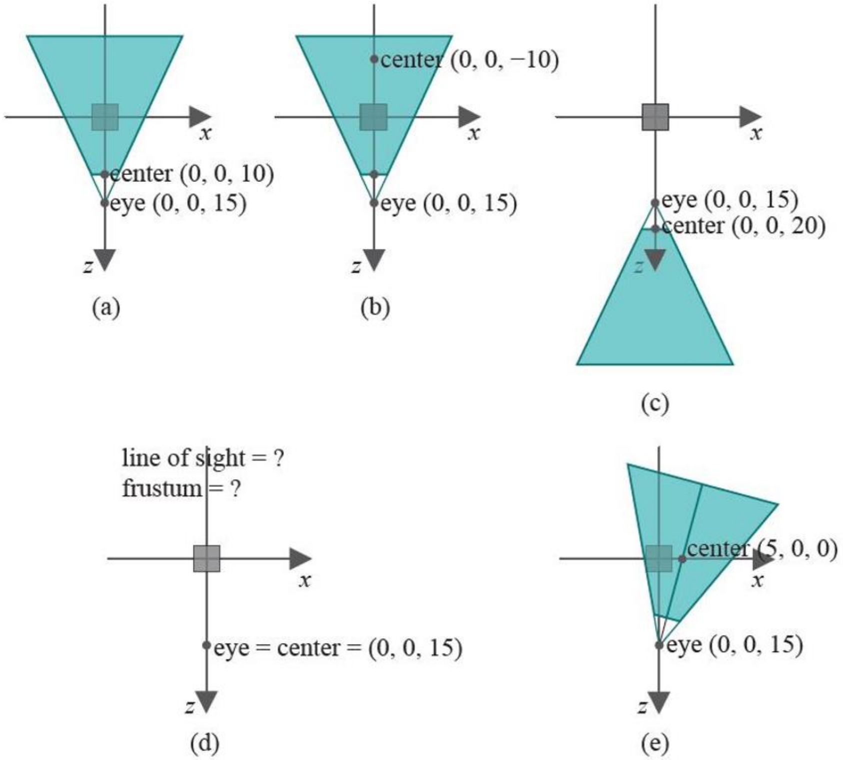


Figure 4.48: (a) `gluLookAt()`: the outlined frustum is the original viewing frustum, the solid one is where it's translated by the `gluLookAt()` call, the box doesn't move. (b) `glTranslatef()`: the viewing frustum doesn't move, rather the box is translated by the `glTranslatef()` call.

- 예: changing only the parameters centerx, centery, centerz – the middle three parameters – of the gluLookAt() call to the following:
- eye=0.0, 0.0, 15.0
- (a) 0.0, 0.0, 10.0
- (b) 0.0, 0.0, -10.0
- (c) 0.0, 0.0, 20.0
- (d) 0.0, 0.0, 15.0
- (e) 5.0, 0.0, 0.0



- **최근 OpenGL 버전에서는 glm library를 사용한 glm::lookat 함수가 있음**
- **출처: [Tutorial 3 : 행렬(매트릭스) (opengl-tutorial.org)]()**

```
glm::mat4 CameraMatrix = glm::lookAt(
    cameraPosition, // 월드 공간에서 당신의 카메라 좌표
    cameraTarget,   // 월드 스페이스에서 당신의 카메라가 볼 곳
    upVector        // glm::vec(0,1,0) 가 적절하나, (0,-1,0)으로 화면을 뒤집을 수 있습니다. 그래도 멋지겠죠
);
```

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
                   glm::vec3(0.0f, 0.0f, 0.0f),
                   glm::vec3(0.0f, 1.0f, 0.0f));
```