Computer Graphics

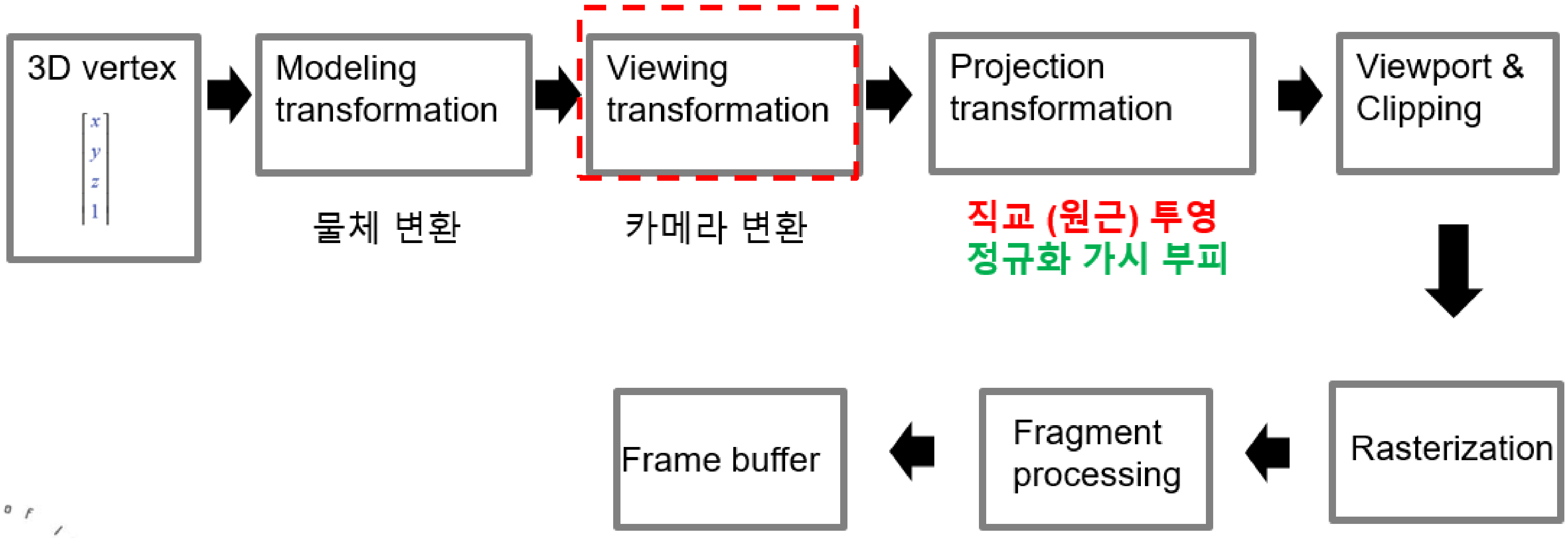
Prof. Jibum Kim

Department of Computer Science & Engineering

Incheon National University

- **Viewing transformation: gluLookAt()**

■ Graphics pipeline (OpenGL 2.x 기준)



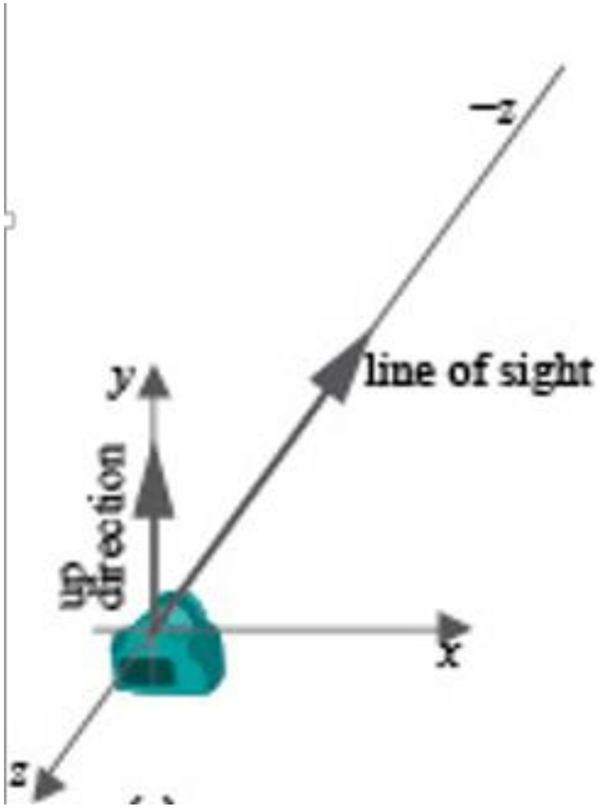
- 앞에서는 물체를 변환시키는 object transformation에 대해서 살펴보았다

- 이번에는 물체가 아닌 카메라의 위치를 변환시키는 viewing transformation에 대해서 알아보자

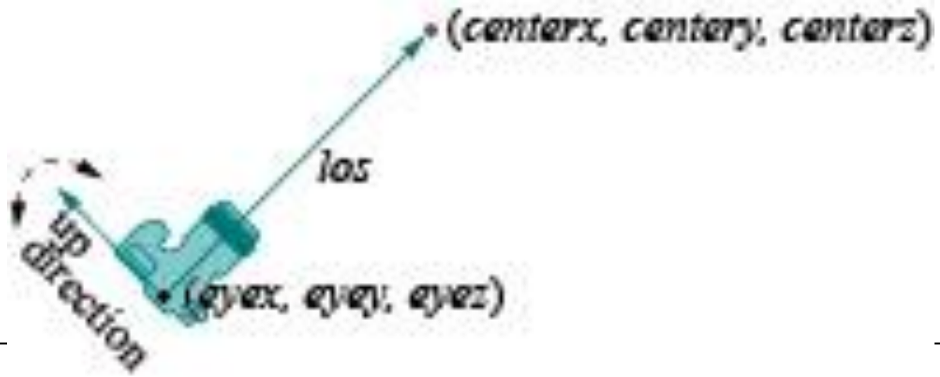
- **Viewing Frustum 을 사용시의 기본적인 가상의 카메라의 위치는 기본적으로 origin (0,0,0)에 위치하고 음의 z축 방향 (-z축 방향)을 바라본다고 가정하였다**

- **카메라가 바라 보는 방향 (즉, -z축 방향)을 line of sight (LOS)라 한다**

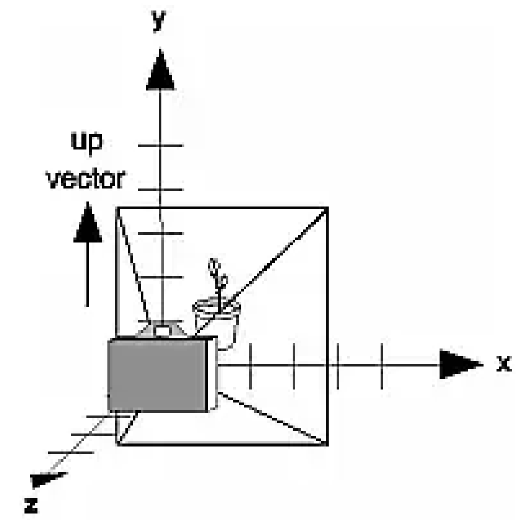
- Default Camera location and LOS in frustum
- 카메라 위치 (0,0,0)
- 방향 (LOS): -z direction
- 카메라 위쪽: +y direction



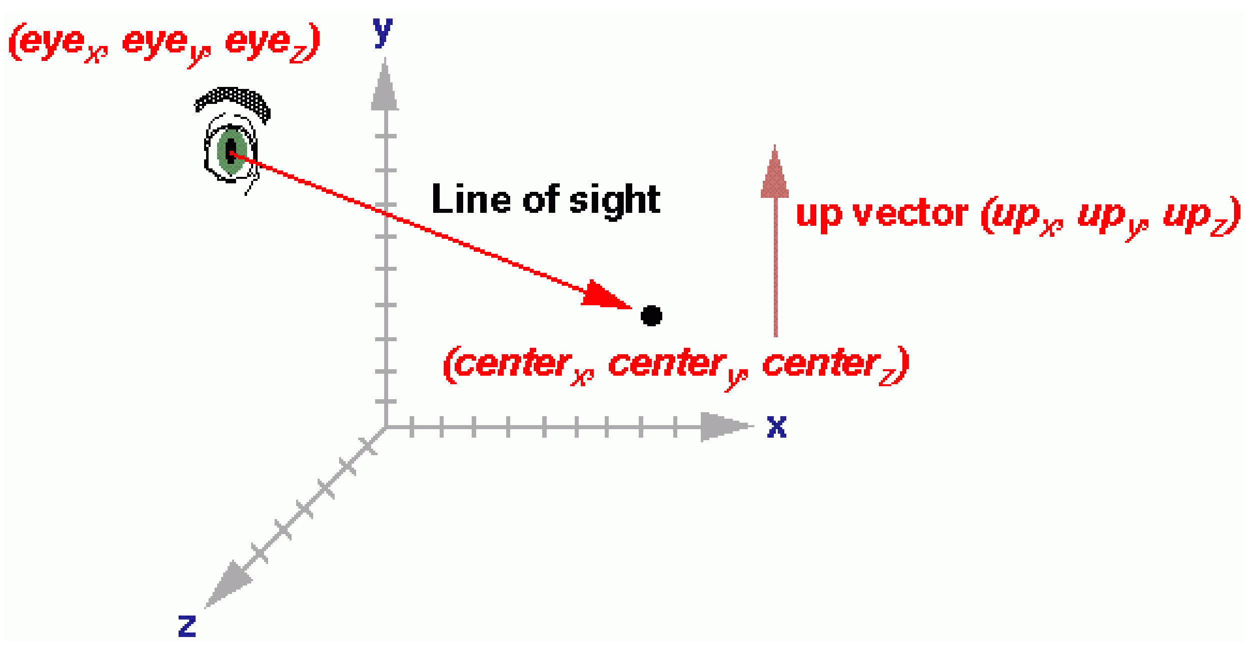
- OpenGL에서는 카메라의 위치, 보는 방향등을 바꿀 수 있는 함수를 제공하는데 그것이 gluLookAt 함수이다
- gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)
- OpenGL camera location: (eyex, eyey, eyez)
- OpenGL camera가 바라보는점 (centerx, centery, centerz)
- OpenGL camera의 up direction: (upx, upy, upz)
- 밑의 그림에서 LOS ? 카메라 위치에서 바라보는 점 연결



- 예: gluLookAt(0,0,15,0,0,0, 0,1,0);
- 카메라 위치?
- 보는 위치?
- 카메라의 위쪽 (up)?



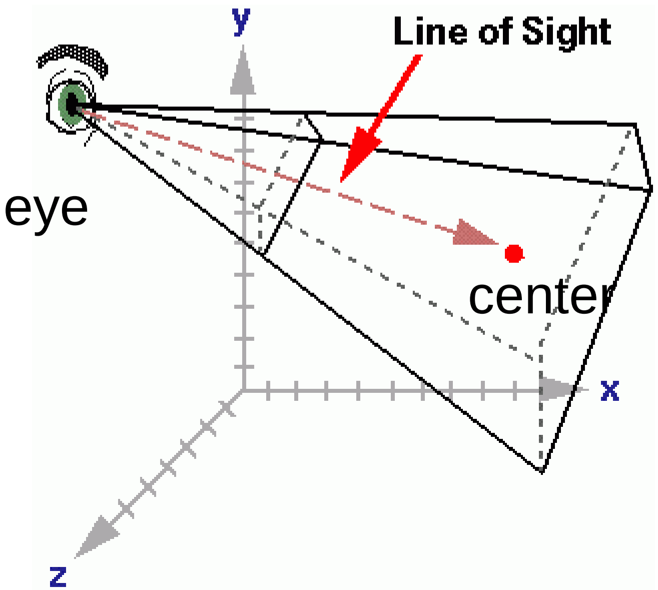
- **gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)**
- **Line of sight: (eyex, eyey, eyez)와 (centerx, centery, centerz)를 연결한 직선**



- Viewing transformation시의 viewing frustum
- The frustum is always aligned down the line of sight (LOS) (frustum은 LOS에 맞춰짐) Why?
- 사실 viewing frustum이라는 것은 가상의 절두체로서 물체를 원근감 있게 표현하려는데 사용된다. 카메라가 center를 바라보는 방향이 LOS 이므로 frustum도 LOS쪽으로 이동해야 한다
- (glFrustum으로 만든 것이 LOS로 이동)
- gluLookAt()함수를 사용해도 frustum의 모양은 변하지 않고 LOS에 따라 viewing frustum의 위치와 각도만 달라진다
- (알아서 frustum이 translate됨)

인천대학교

- **Viewing frustum stays connected to the eye (camera)**
- **The frustum is always aligned down the line of sight**
- **gluLookAt() 함수가 없을때의 viewing frustum이 LOS 방향으로 이동 (단, 원래 카메라와 viewing face사이의 거리는 동일)**



- gluLookAt() 함수에서 주로 사용하는 인자는 처음 6개의 인자이다
- 즉, camera의 위치 (eyex,eyey,eyez)와
- 쳐다보는 점 (centerx, centery, centerz)을 의미
- 원래 LOS와 up 방향은 수직이어야 하지만 정확한 계산이 어렵기 때문에 대부분의 경우 마지막 3개 인자 (upx, upy, upz)는 대부분 (0,1,0)으로 놓고 쓴다.

인천대학교

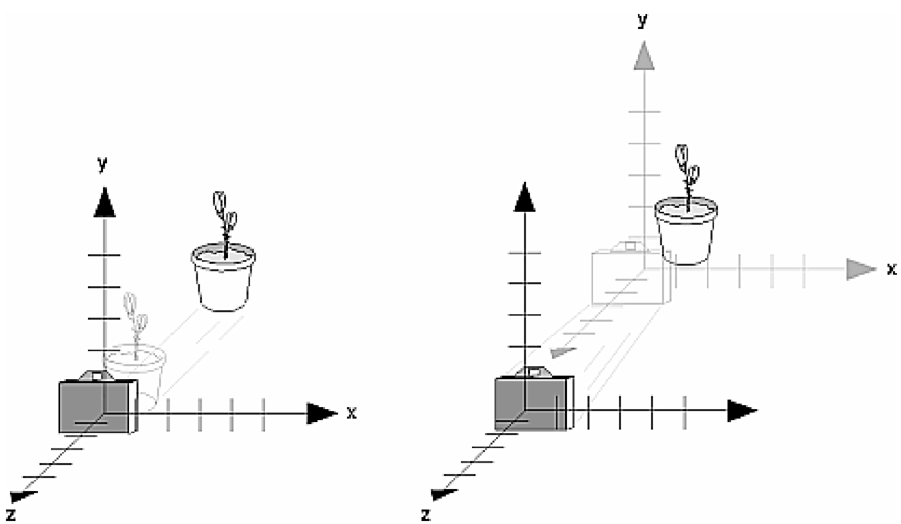
- glutObject 들은 최초에 중심이 (0,0,0)에 위치함 Frustum사용시 glTranslatef(0,0,-15)와 같이 물체를 이동시켜서 viewing frustum안으로 물체를 이동시켰다
- Q) 이때 카메라와 object 중심 사이의 거리는?
- 이번엔 glutObject를 (0,0,0)에 고정하고 gluLookAt 함수로 카메라의 위치를 이동시켜 보자
- gluLookAt(0,0,15,0,0,0, 0,1,0);
- 카메라를 z축(+)로 15만큼이동. 물체는 (0,0,0)에 고정
- Q) 이때 카메라와 object 중심 사이의 거리는?

1. glutobject를 보기 위해 그 물체를 z축으로 -15이동시켰다 (왼쪽 그림)

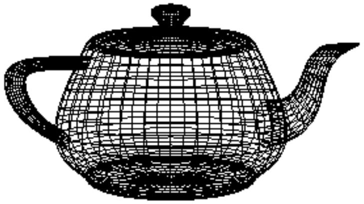
- glTranslatef(0,0,-15) => 카메라와 물체 사이의 거리=

2. 물체는 움직이지 않고 카메라를 z축으로 +15만큼 이동시켰다

- (오른쪽 그림) gluLookAt(0,0,15,0,0,0, 0,1,0);
- 카메라와 물체 사이의 거리=



- 예) glutWireTeapot(5.0); 을 frustum을 이용하여
- 사용시 이 두 가지가 실행 결과가 동일한지 확인해 보자
- 물체 이동 : glTranslatef(0.0, 0.0, -15.0);
- 카메라 이동 : gluLookAt(0,0,15,0,0,0, 0,1,0);



```
#include <GL/glut.h>

void Init()
{
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glColor3f(0.0, 0.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0);
}

void Display(){
  glClear(GL_COLOR_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0, 0, -15);
  glutWireTeapot(5.0);
glFlush();
 }
int main(){
  glutInitWindowSize(400,400);
  glutInitWindowPosition(100,100);
  glutCreateWindow("OpenGL Hello World!");
  Init();
  glutDisplayFunc(Display);
  glutMainLoop();
  return 0;
}
```

- **glTranslatef(0, 0, -15);**
- **를 comment out (주석 처리) 하고**
- **gluLookAt(0,0,15,0,0,0, 0,1,0);**

인천대학교

- gluLookAt(0,0,15,0,0,0, 0,1,0);
- 를 다음과 같이 바꾸어 보자 어떤 결과를 예상하는가? 왜 그런가?
- gluLookAt(0,0,15,0,0,-10, 0,1,0);

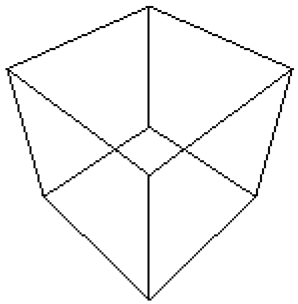
인천대학교

- 실제 gluLookAt(); 함수는 3D 물체를 입체 있게 보여주는데 유용하다
- 다음의 예제를 살펴보자

```
#include <GL/glut.h>

void Init()
{
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glColor3f(0.0, 0.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glFrustum(-5.0, 5.0, -5.0, 5.0, 5.0, 100.0);
}

void Display(){
  glClear(GL_COLOR_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(6,8,6,0,0,0, 0,1,0);
  glutWireCube(5.0);
  glFlush();
 }
int main(){
  glutInitWindowSize(400,400);
  glutInitWindowPosition(100,100);
  glutCreateWindow("OpenGL Hello World!");
  Init();
  glutDisplayFunc(Display);
  glutMainLoop();
  return 0;
}
```



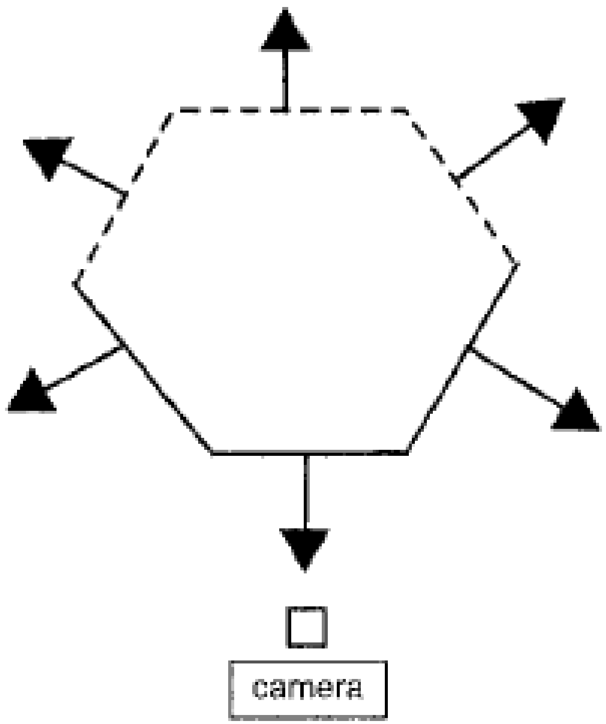
- 대부분의 경우 gluLookAt() 함수를 사용시 카메라의 up에 해당하는 마지막 세 인자는 (0, 1, 0)으로 둔다

- **Vector**

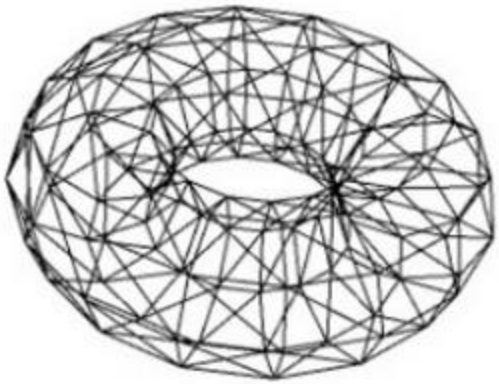
- 벡터 (vector)
- (Euclidean) Vector? a geometric object that has both a magnitude (크기) and direction (방향)

인천대학교

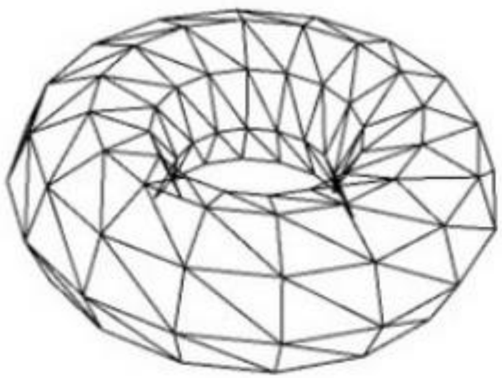
- 왜 컴퓨터 그래픽스에서 벡터가 중요한가?
- 1. Polygon에서 어떠한 면이 camera 위치에 대하여 앞면인지 뒷면 (후면, 이면)인지 따질 때 벡터를 사용한다



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

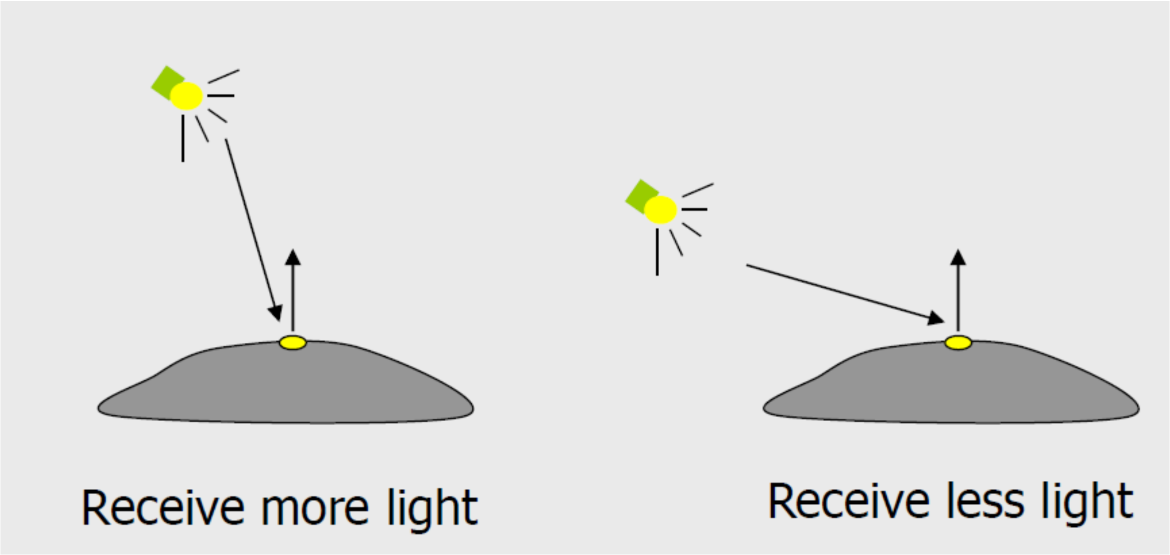


Torus drawn in wire-frame
without back face culling

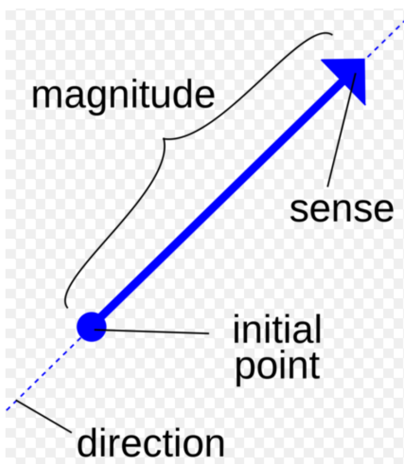


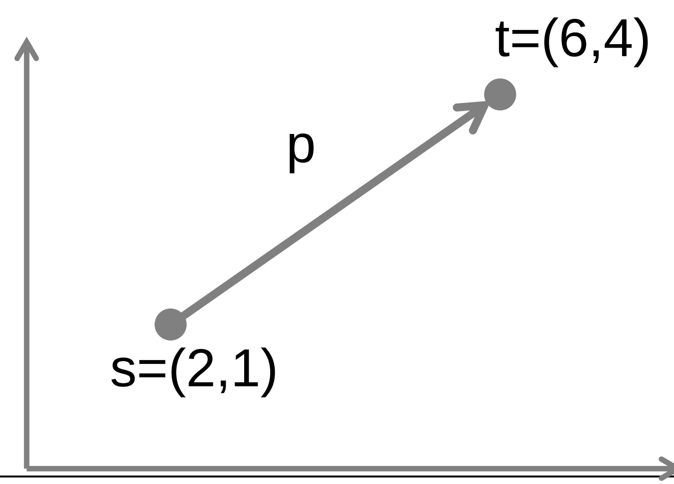
Torus drawn in wire-frame
with back face culling

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



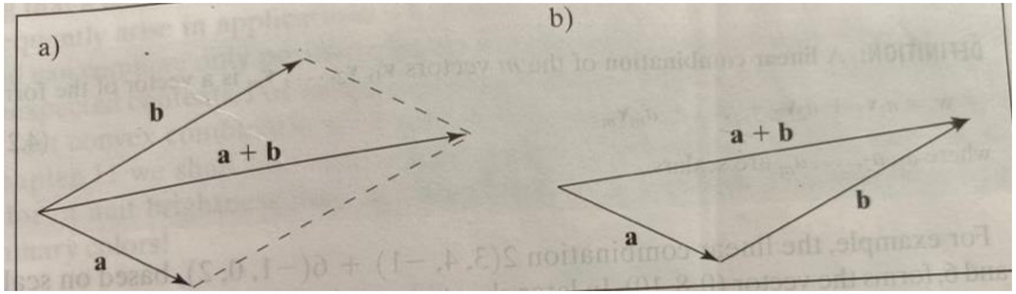
- Vector? **a geometric object that has both a magnitude (크기) and direction (방향)**
- **Magnitude: distance between two points (initial point=>end point)**
- **예) p: vector from point s to point t: p=t-s=(6,4)-(2,1)=(4,3)**
- **Magnitude=**$\sqrt{4^2+3^2}=5$
- **Vector는 보통 bold (강조) 혹은 italic, *p*, 혹은 화살표 표시** $\vec{p}$



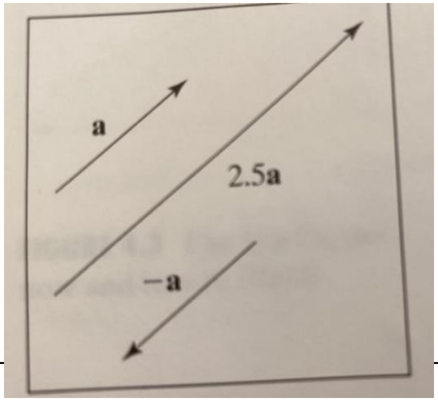


- Vector의 표기
- n-차원 벡터는 n-tuple로 표기
- $w=(w_1, w_2, \ldots, w_n)$

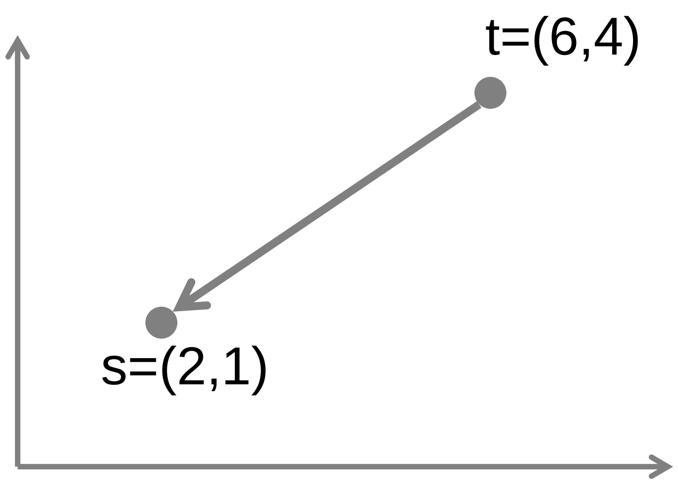
- Vector의 합
- a=(2, 5, 6) and b=(-2, 7, 1)
- a+b=(0, 12, 7)



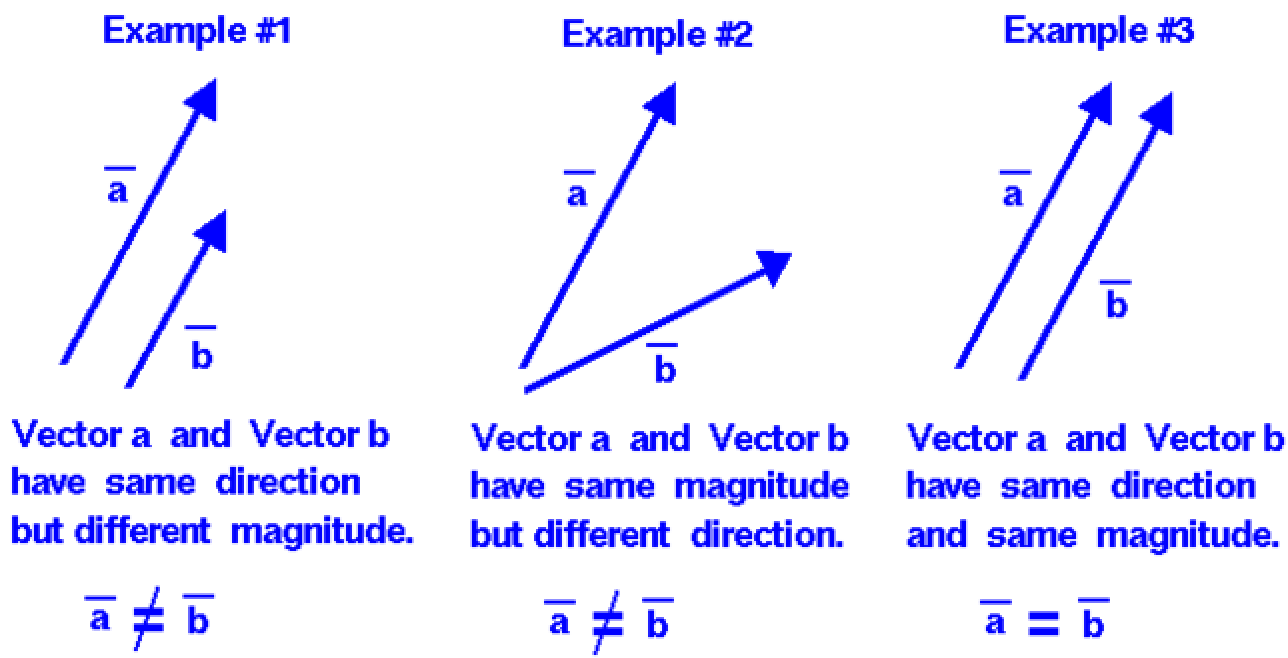
- Vector와 scalar와의 곱
- 6a=(12, 30, 36)



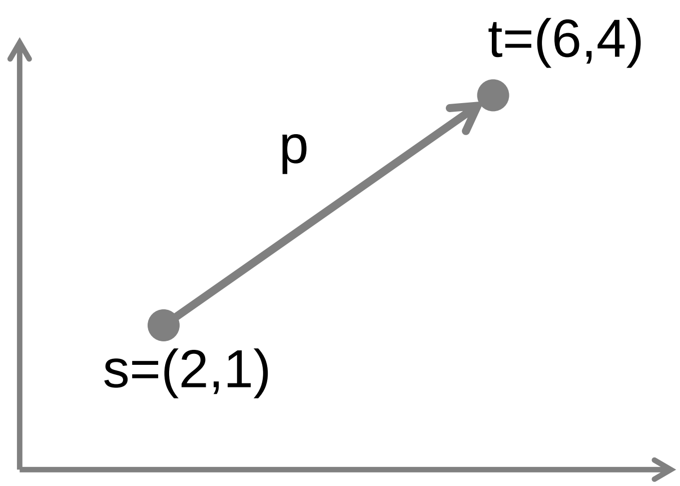
- Vector has a direction
- 예) vector from point t to point s: q=s-t=(2,1)-(6,4)=(-4,-3)
- Magnitude=$\sqrt{(2-6)^2+(1-4)^2} = 5$
- 즉, 방향이 바뀌면 (반대 방향이 되면) 부호가 바뀐다
- s->t로 가는 vector: p=(4,3),
- t->s로 가는 vector: q=(-4,-3)
- p≠q (opposite direction)
- Vector 표기: italic, 진하게, 화살표표시

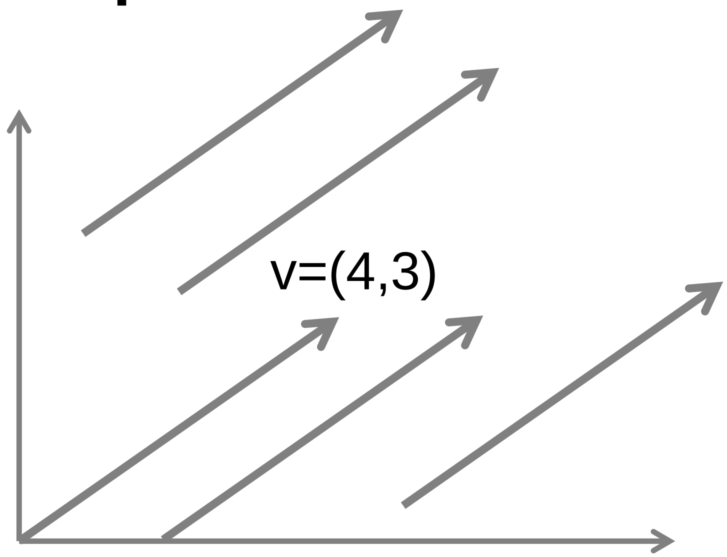


- **Vector의 특징**
- **두 vector는 크기 (magnitude) 와 방향 (direction) 이 모두 같아야 같은 vector이다**



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



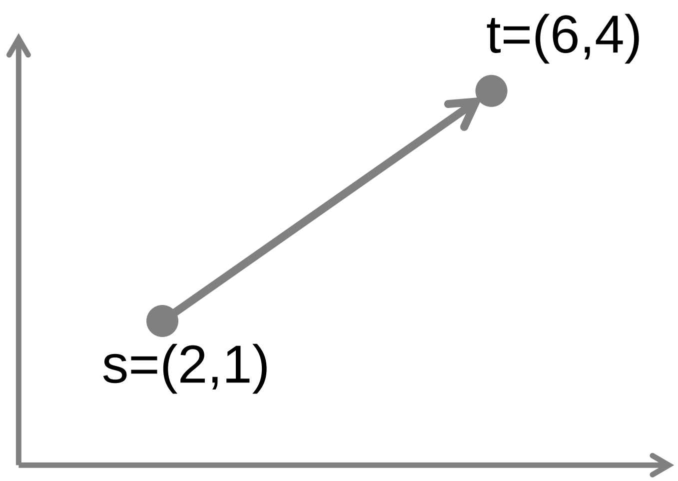


- **Normalized vector (정규화 벡터)**
- **In general, we use a normalized vector**
- **Vector with a magnitude 1 (크기가 1인 vector)**
- **3차원 vector p(x, y, z)의 magnitude, |p|=**$\sqrt{x^2+y^2+z^2}$
- **normalized vector: p'**

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

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

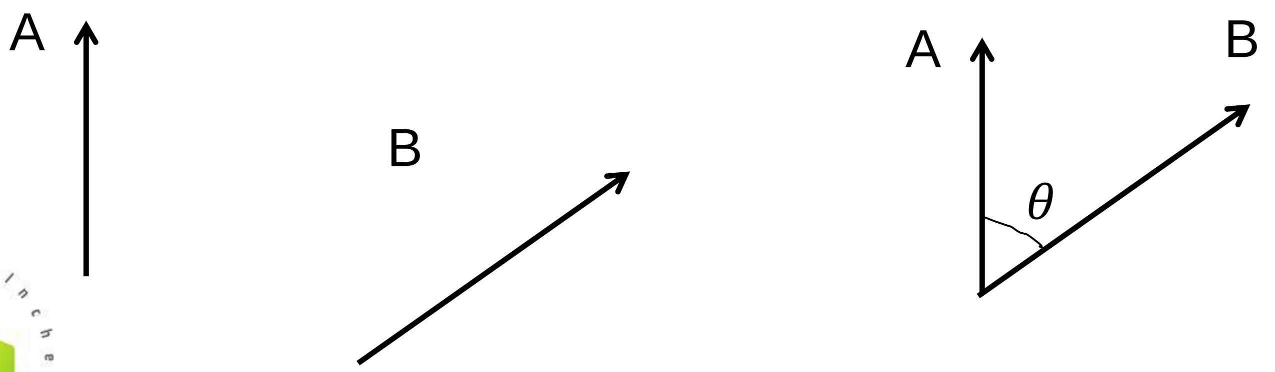
- **Ex) Normalized vector from point s to point t**
- **p=t-s=(6,4)-(2,1)=(4,3)**
- **|p|=**$\sqrt{4^2+3^2} = 5$
- **p'=**$\frac{p}{|p|} = (\frac{4}{5}, \frac{3}{5})$



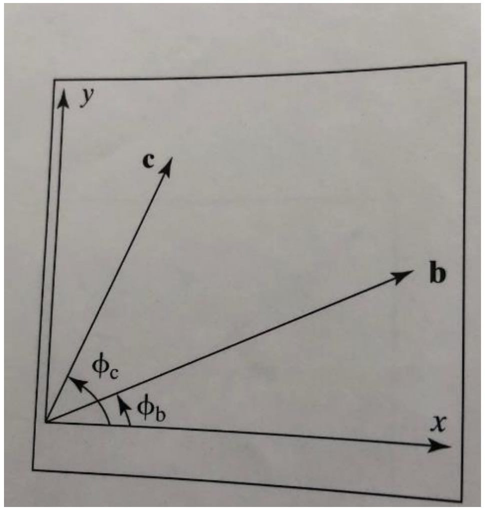
- **예) Normalized vector from point A (1, 3, 2) to point B(2,4,1)**
- ***P*=B-A =**
- ***P*의 magnitude=**
- **p'=**

- **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 !**



- 두 벡터 사이의 사이각, θ , 구하기
- $b = (|b| cos\ \varphi_b, |b| sin\ \varphi_b)$
- $c = (|c| cos\ \varphi_c, |c| sin\ \varphi_c)$
- $b \bullet c = |b||c| cos\ \varphi_b cos\ \varphi_c + |b||c| sin\ \varphi_b sin\ \varphi_c$
- $b \bullet c = |b||c| cos(\varphi_c\text{-}\varphi_b)$
- $b \bullet c = |b||c| cos(\theta)$
- $cos(\theta) = \dfrac{b \bullet c}{|b||c|}$

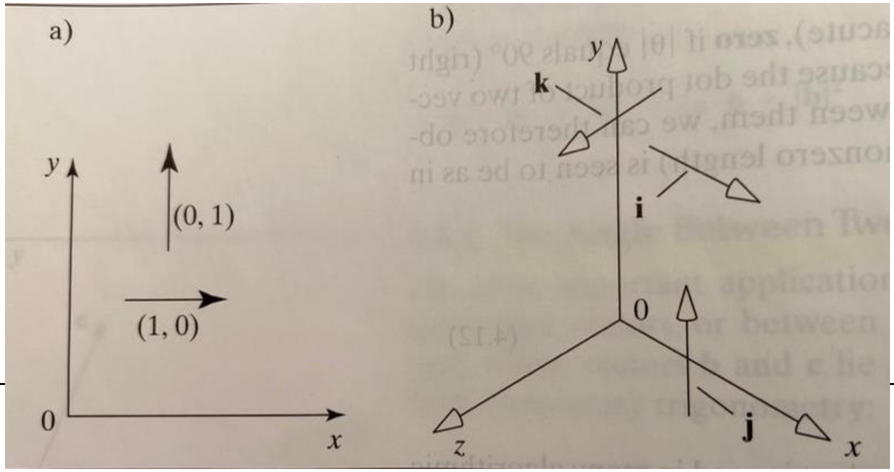


- 다음 두 벡터 *b*=(3,4)와 *c*=(5,2)사이의 각도를 계산해 보자

- 두 벡터 사이의 내적값을 통해 두 벡터의 사이각에 대해서 알 수 있다
- 두 벡터 A, B의 사이각: $\boldsymbol{\theta = cos^{-1}(\frac{A \cdot B}{|A||B|})}$, $\mathbf{cos(\theta) = \frac{A \cdot B}{|A||B|}}$
- 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)

인천대학교

- **Vectors b and c are perpendicular if *b* • *c*=0**
- **Other names for perpendicular are orthogonal and normal**
- **Standard unit vector**
- **The standard unit vectors in 3D have components:**
- ***i*=(1, 0, 0), *j*=(0, 1, 0), *k*=(0, 0, 1)**



- **두 벡터의 외적 (cross product)**

- 벡터의 외적은 기호 (x, cross product)로 표시되고
- 아래와 같인 연산할 수 있다. 벡터의 외적은 결과적으로 다시 벡터가 된다

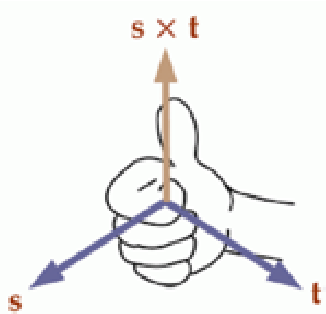
S: (sx, sy, sz),   t: (tx, ty, tz)

sxt=(sy*tz-sz*ty, -sx*tz+sz*tx, sx*ty-sy*tx)

- 행렬식 (determinant) 이용하면 기억하기 쉬움

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

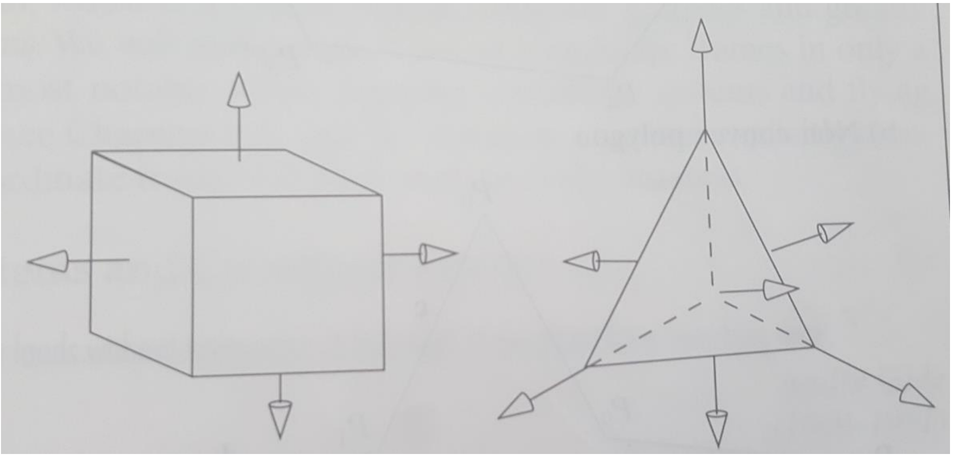
- **벡터 외적 (cross product)의 *sxt* , 의 특징**
- **1. 벡터 *s*와 벡터 *t*의 외적 결과도 *sxt* 도 벡터이다**
- **2. 벡터 *sxt* 는 벡터 *s* , 벡터 *t*와 perpendicular (직교)하다**
- **이를 이용하여 벡터 외적을 법선 벡터 계산시 사용**
- **3. 벡터 *sxt* 의 방향은 오른손 법칙을 이용하면 첫 벡터 s로 부터 둘째 벡터인 t를 향해 오른손 주먹을 감싸 쥐었을 때 엄지 손가락의 방향이다 (즉, *sxt* 와 *txs* 는 크기는 같지만 벡터 방향이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

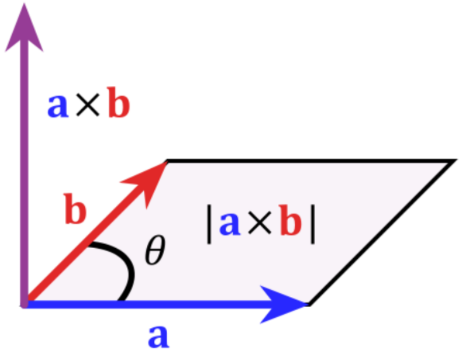


- 예: *i*x*j*=*k*, (*i*=(1, 0, 0), *j*=(0, 1, 0), *k*=(0, 0, 1)), i, j, k (standard unit vector), 오른손 법칙 이용

- **Normal vector (법선 벡터)**

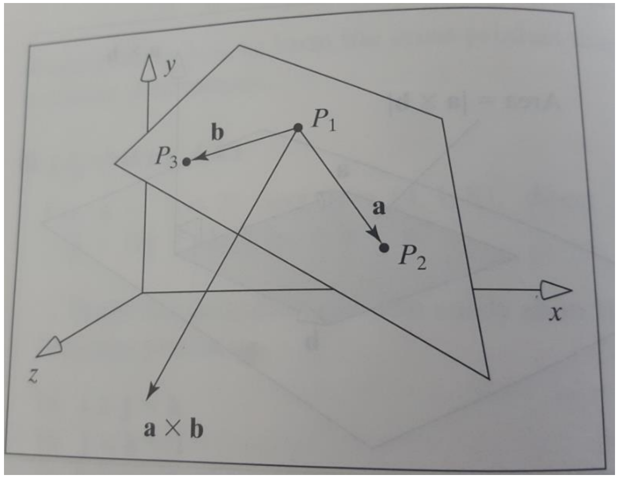
- **Normal vector (법선 벡터):** 평면에 수직인 vector
- 법선 벡터는 앞에서 배운 벡터 외적을 통해서 구할 수 있다
- 두 물체에 대하여 각 face (면)에 대하여 normal vector를 표시해 보면





인천대학교

- 평면에 대한 방정식이 아래와 같이 주어져 있는 경우에는
- $ax+by+cz+d=0$
- 법선 벡터 (*n*)는 *n*=(a, b, c)이다
- 평면의 방정식이 주어져 있지 않은 경우에는 그 평면에 있는 세 점 P1, P2, P3을 찾은 후 두 벡터 *a*=P2-P1, *b*=P3-P1을 구한 후 다음의 연산을 통해서 구할 수 있다
- *n=axb*



E.g., Find a normal vector to the plane through the points

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

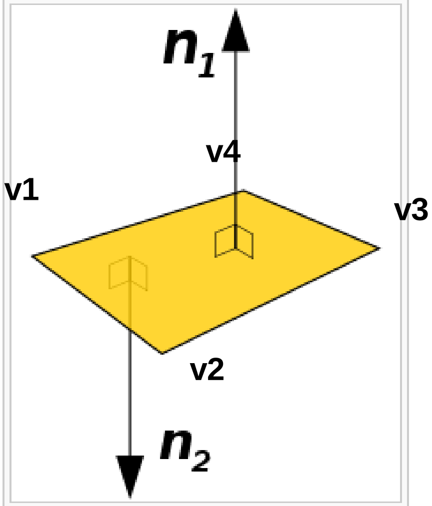
***a*=(1, 3, -2), *b*=(0, 2, 2), *n* (normal vector)**

- OpenGL에서는 vertex를 명시하는 순서에 따라서 법선 벡터의 방향이 달라지게 된다
- 컴퓨터 그래픽스에서 법선 벡터가 중요한 이유는 어떤 면이 공간상에서 어디를 향해 있는지를 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 are of a sphere or a cube