Step09: Projection Transform

Perspective Projection, vanishing point C

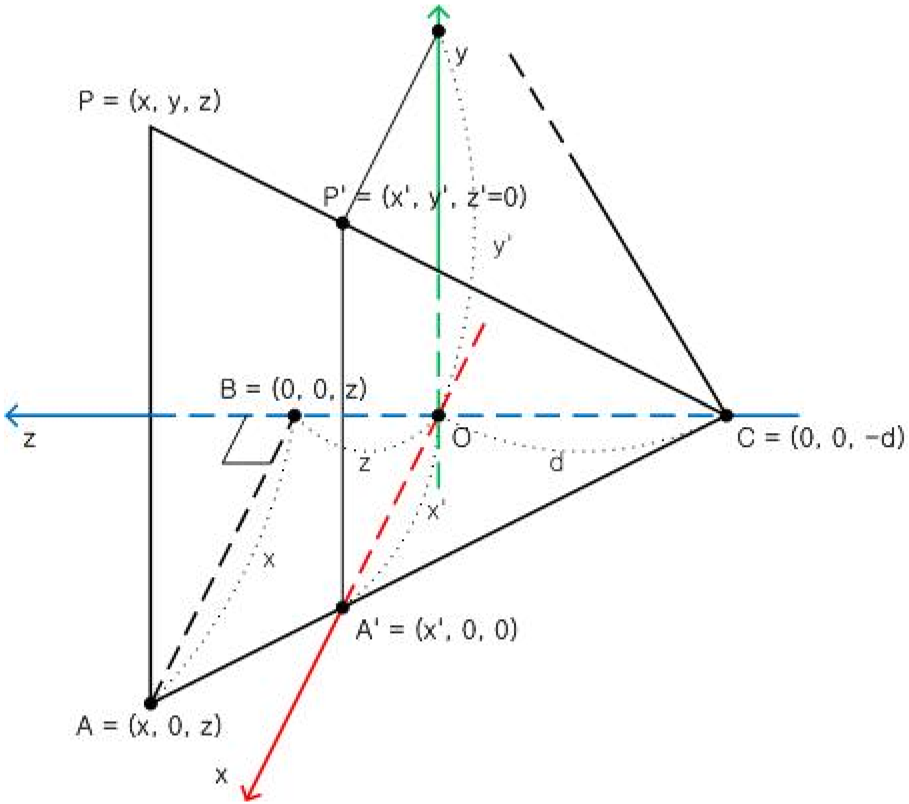


Fig. Setting Projection Matrix

Eye in +z direction
Vanishing point C=(0,0,-d)
P=(x,y,z) is projected to p' on xy-plane
input C, P --> output P'
similarity between triangle ABC and triangle A'OC.

Proportional expression

$$x : x' = (z+d) : d$$

Rearranging with respect to x'

$$x' = \frac{dx}{(z+d)}$$

Similarly we can derive:

$$y' = \frac{dy}{(z+d)}, \quad z' = \frac{dz}{(z+d)}$$

Common factor d/(z+d)

Denominator of homogeneous division can be $(z+d)/d$

$$w = (z+d)/d = \frac{1}{d}z + 1 = 0x + 0y + \frac{1}{d}z + 1$$

Linear equation for (x', y', z', w) will be:

$$\begin{aligned}
x' &= 1x + 0y + 0z + 0 \\
y' &= 0x + 1y + 0z + 0 \\
z' &= 0x + 0y + 1z + 0 \\
w &= 0x + 0y + z(1/d) + 1
\end{aligned}$$

For x'

$$x' = \frac{x}{\frac{(z+d)}{d}} = \frac{dx}{(z+d)}$$

Now homogeneous matrix for projection transform will be:

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

Projection Matrix in DirectX

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. In official DirectX documentation, above projection matrix will shows up. DirectX uses Row Vector

(Link for DirectX documents about Projection Transform)

https://docs.microsoft.com/en-us/windows/desktop/direct3d9/projection-transform

Homogeneous division after projection

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{vmatrix}$$

How can 'w' be used?

Depth buffer

z-buffer

w-buffer

```
KMatrix4 KMatrix4::SetProjection( float d )
{
    SetZero( );
    m_afElements[0][0] = 1;
    m_afElements[1][1] = 1;
    m_afElements[2][2] = 1;
    m_afElements[3][2] = 1.0f / d;
    m_afElements[3][3] = 1;

    return *this;
}//KMatrix4::SetProjection
```

Cube

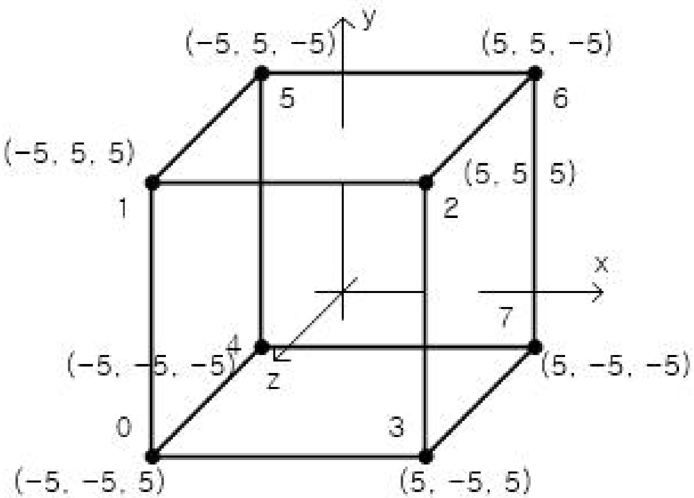


Fig. Indices and vertices for cube

8 vertices

12 edges

Setup Index Buffer

```
int buffer[] = {
    0,2,1,
    2,0,3,
    3,6,2,
    6,3,7,
    7,5,6,
    5,7,4,
    4,1,5,
    1,4,0,
    4,3,0,
    3,4,7,
    1,6,5,
    6,1,2
};
```

Modified KPolygon.cpp

```
#include "stdafx.h"
#include "KPolygon.h"
#include "KVectorUtil.h"

void DrawIndexedPrimitive( HDC hdc
    , int* m_indexBuffer            // index buffer
    , int primitiveCounter          // primitive counter
    , KVector3* m_vertexBuffer      // vertex buffer
    , COLORREF color )
{
    int   i0, i1, i2;
    int   counter = 0;

    for (int i = 0; i < primitiveCounter; ++i)
    {
        // get index
        i0 = m_indexBuffer[counter];
        i1 = m_indexBuffer[counter + 1];
        i2 = m_indexBuffer[counter + 2];

        // draw triangle
        KVectorUtil::DrawLine(hdc, m_vertexBuffer[i0].x, m_vertexBuffer[i0].y
```

```
            , m_vertexBuffer[i1].x, m_vertexBuffer[i1].y, penWidth, penStyle, color );
        KVectorUtil::DrawLine(hdc, m_vertexBuffer[i1].x, m_vertexBuffer[i1].y
            , m_vertexBuffer[i2].x, m_vertexBuffer[i2].y, penWidth, penStyle, color );
        KVectorUtil::DrawLine(hdc, m_vertexBuffer[i2].x, m_vertexBuffer[i2].y
            , m_vertexBuffer[i0].x, m_vertexBuffer[i0].y, penWidth, penStyle, color );

            // advance to next primitive
        counter += 3;
    }//for
}//DrawIndexedPrimitive()


KPolygon::KPolygon()
{
    m_sizeIndex  = 0;
    m_sizeVertex = 0;
    m_color = RGB(0, 0, 255);
}//KPolygon::KPolygon()

KPolygon::~KPolygon()
{
}//KPolygon::~KPolygon()

void KPolygon::SetIndexBuffer()
{
```

```
    int buffer[] = {
        0,2,1,
        2,0,3,
        3,6,2,
        6,3,7,
        7,5,6,
        5,7,4,
        4,1,5,
        1,4,0,
        4,3,0,
        3,4,7,
        1,6,5,
        6,1,2
    };

    for (int i=0; i<_countof(buffer); ++i)
    {
        m_indexBuffer[i] = buffer[i];
    }//for
    m_sizeIndex = _countof(buffer);
}//KPolygon::SetIndexBuffer()

void KPolygon::SetVertexBuffer()
{
    m_vertexBuffer[0] = KVector3( -5.f, -5.f,  5.f);
```

```
    m_vertexBuffer[1] = KVector3( -5.f,  5.f,  5.f);
    m_vertexBuffer[2] = KVector3(  5.f,  5.f,  5.f);
    m_vertexBuffer[3] = KVector3(  5.f, -5.f,  5.f);
    m_vertexBuffer[4] = KVector3( -5.f, -5.f, -5.f);
    m_vertexBuffer[5] = KVector3( -5.f,  5.f, -5.f);
    m_vertexBuffer[6] = KVector3(  5.f,  5.f, -5.f);
    m_vertexBuffer[7] = KVector3(  5.f, -5.f, -5.f);
    m_sizeVertex = 8;
}//KPolygon::SetVertexBuffer()

void KPolygon::Render(HDC hdc)
{
    ::DrawIndexedPrimitive(
        hdc,
        m_indexBuffer,      // index buffer
        12,                 // primitive(triangle) counter
        m_vertexBuffer,     // vertex buffer
        m_color );
}//KPolygon::Render()

void KPolygon::Transform(KMatrix4& mat)
{
    for (int i=0; i<m_sizeVertex; ++i)
    {
        m_vertexBuffer[i] = mat * m_vertexBuffer[i];
```

```
    }//for
}//KPolygon::Transform()
```

Output

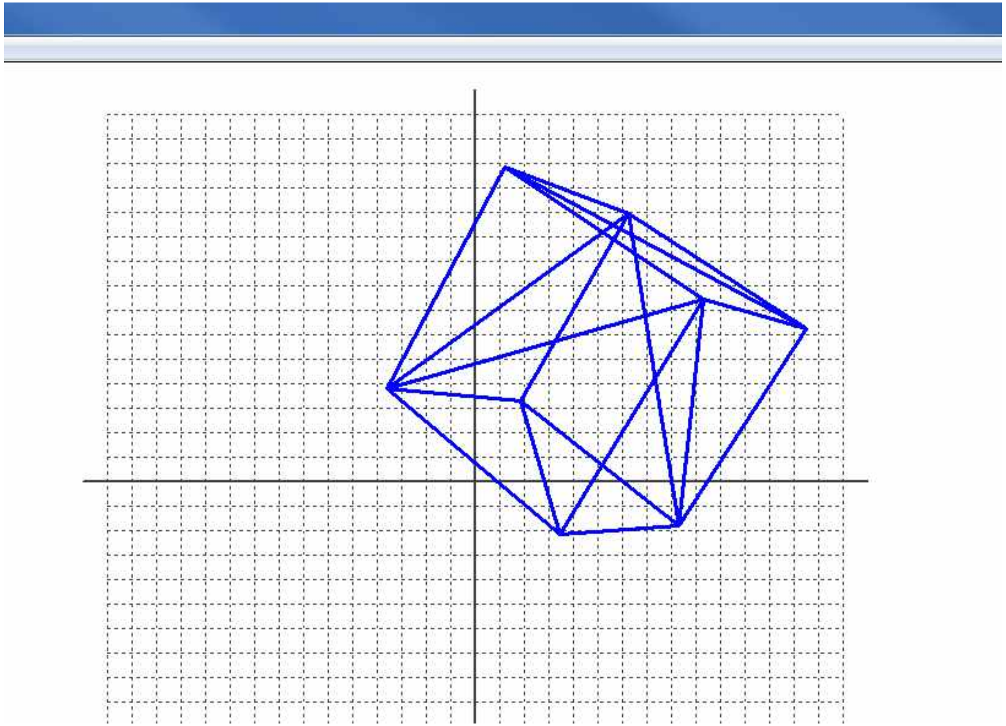


Fig. Cube without Hidden Surface removing

Culling

Hidden Surface Culling

Occlusion Culling

Viewport Culling

Additional Vector Operations

dot product and cross product

Dot Product(==Inner Product)

$$u \cdot v = \begin{cases} |u||v|\cos\theta, & \text{if } u \neq 0 \text{ and } v \neq 0 \\ 0, & \text{if } u = 0 \text{ or } v = 0 \end{cases}$$

$u = (u_1, u_2, u_3)$, $v = (v_1, v_2, v_3)$

u's tip is P

v's tip is Q

angle between u and v is θ

Derivation

Law of cosines

$$|\overrightarrow{PQ}|^2 = |u|^2 + |v|^2 - 2|u||v|\cos\theta$$

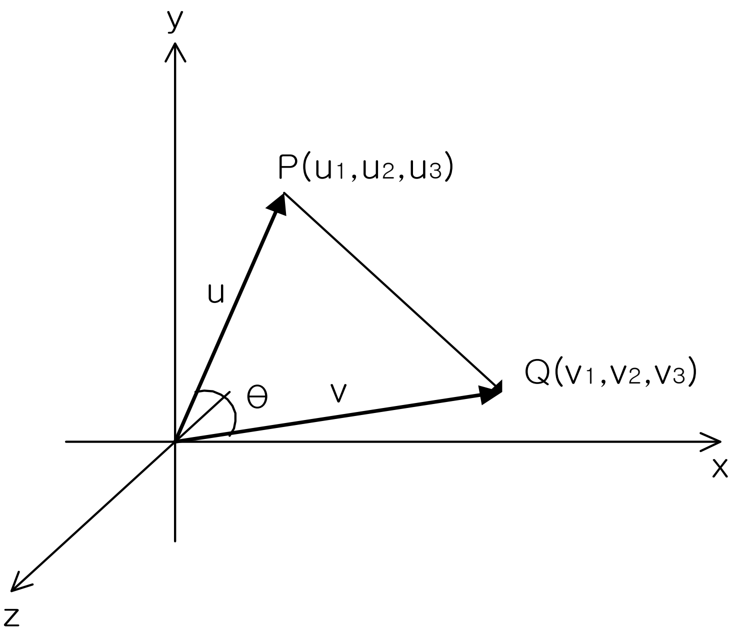


Fig. Law of cosine: $|\overrightarrow{PQ}|^2 = |u|^2 + |v|^2 - 2|u||v|\cos\theta$

expanding equation:

$$
\begin{aligned}
&v_1^2 - 2v_1u_1 + u_1^2 + v_2^2 - 2v_2u_2 + u_2^2 + v_3^2 - 2v_3u_3 + u_3^2 \\
&= u_1^2 + u_2^2 + u_3^2 + v_1^2 + v_2^2 + v_3^2 - 2|u||v|\cos\theta
\end{aligned}
$$

equation can be simplified:

$$
\begin{aligned}
&-2v_1u_1 - 2v_2u_2 - 2v_3u_3 \\
&= -2|u||v|\cos\theta
\end{aligned}
$$

Now

$$|u||v|\cos\theta = u_1v_1 + u_2v_2 + u_3v_3$$

What's the meaning of $|u||v|\cos\theta$?

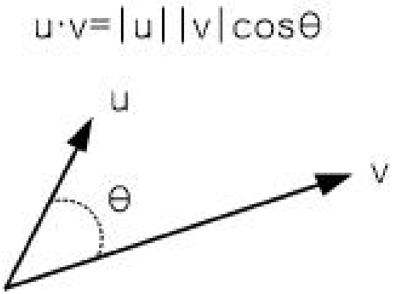


Fig. Inner product

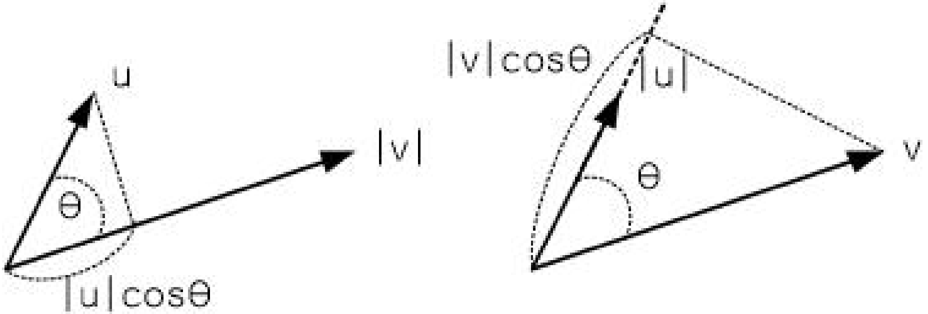


Fig. Inner Product: a vector is projected to other vector, dot product is multiplication of length of projected vector and other vector.

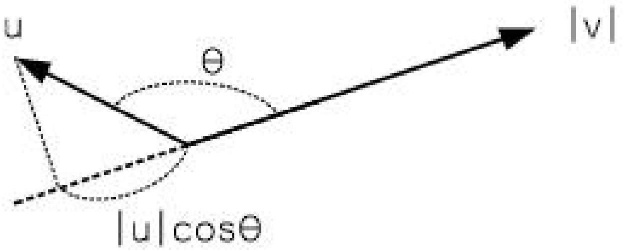


Fig. When dot product is less than 0: angle between two vector is greater than $\pi/2$ (90 degree)

Inner Product, Dot Product

$$u \cdot v = u_1v_1 + u_2v_2 + u_3v_3 = |u||v|\cos\theta$$

Angle between two vectors

$$\cos\theta = \frac{u \cdot v}{|u||v|}$$

if u·v == 0, then two vectors are perpendicular to each other.

Normal

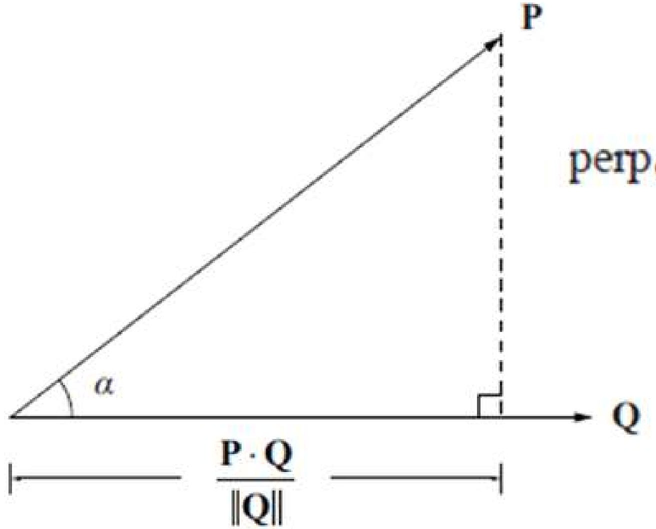
1). in 2-dimensional space, $n = (a,b)$ and line $ax + by + c = 0$ is perpendicular to each other

2) in 3-dimensional space, $n = (a, b, c)$ is normal vector for plane $ax + by + cz + d = 0$

Application : Vector Decomposition

- ✓ The situation often arises in which we need to decompose a vector **P** into components that are **parallel** and **perpendicular** to another vector **Q**.

$$\|\mathbf{P}\|\cos\alpha = \frac{\mathbf{P}\cdot\mathbf{Q}}{\|\mathbf{Q}\|}.$$



$$\begin{aligned}\operatorname{perp}_{\mathbf{Q}}\mathbf{P} &= \mathbf{P} - \operatorname{proj}_{\mathbf{Q}}\mathbf{P} \\ &= \mathbf{P} - \frac{\mathbf{P}\cdot\mathbf{Q}}{\|\mathbf{Q}\|^2}\mathbf{Q}.\end{aligned}$$

$$\operatorname{proj}_{\mathbf{Q}}\mathbf{P} = \frac{\mathbf{P}\cdot\mathbf{Q}}{\|\mathbf{Q}\|^2}\mathbf{Q}$$

Determinant

2×2 Matrix is given, we want to know area changes after transform.

$$\begin{vmatrix} 2 & -1 \\ 1 & 1 \end{vmatrix}$$

area 1×1 in standard basis will be changed in certain ratio.

Calculating size of hatched area.

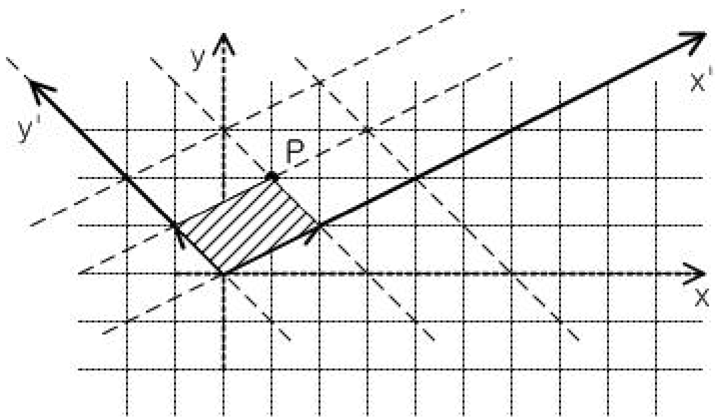


Fig. We can calculate area formed by transform basis, this is determinant.

Proof)

Ex) calculating unit area for matrix $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$

ad-bc.

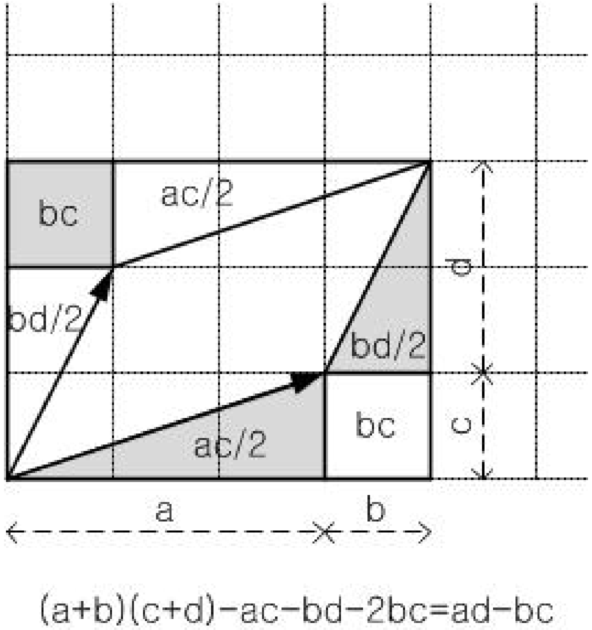


Fig. Calculation size formed by basis

determinant for 2×2 matrix

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

Fig. determinant for 2×2 matrix

Determinant in 3-dimensional transform.

It's unit volume.

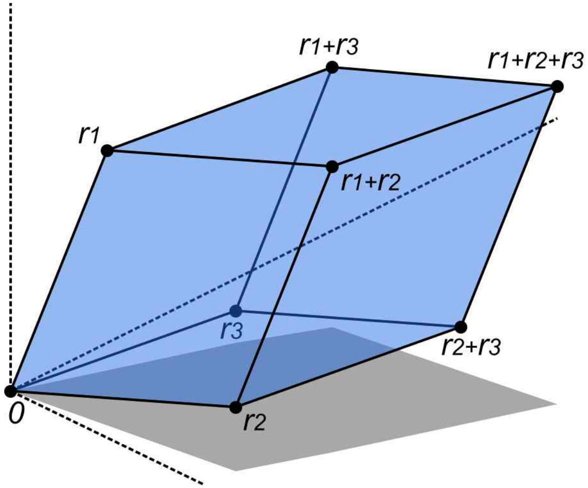


Fig. Determinant of 3×3 matrix: unit volume formed by 3 basis.

Cross Product

determinant for 3×3 matrix

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

Fig. determinant for 2×2 matrix

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

$$= aei + bfg + cdh - ceg - bdi - afh.$$

Fig. determinant for 3×3 matrix

Cross product can be defined by pseudo determinant

Sarrus Rule

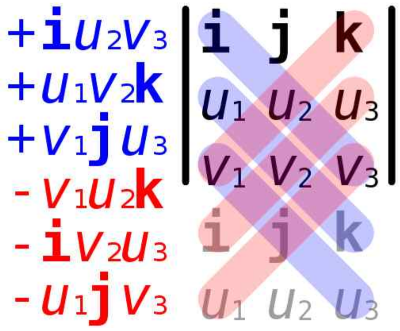


Fig. Sarrus Rule for Cross Product

cross product u×v

$u = (u_1, u_2, u_3)$, $v = (v_1, v_2, v_3)$

$$u \times v = (u_2v_3 - u_3v_2,\ u_3v_1 - u_1v_3,\ u_1v_2 - u_2v_1)$$

u×v is perpendicular to u and v(plane uv)

It's length:

$$|u \times v| = |u||v|\sin\theta$$

RHS, right-hand coordinate system

right-hand rule

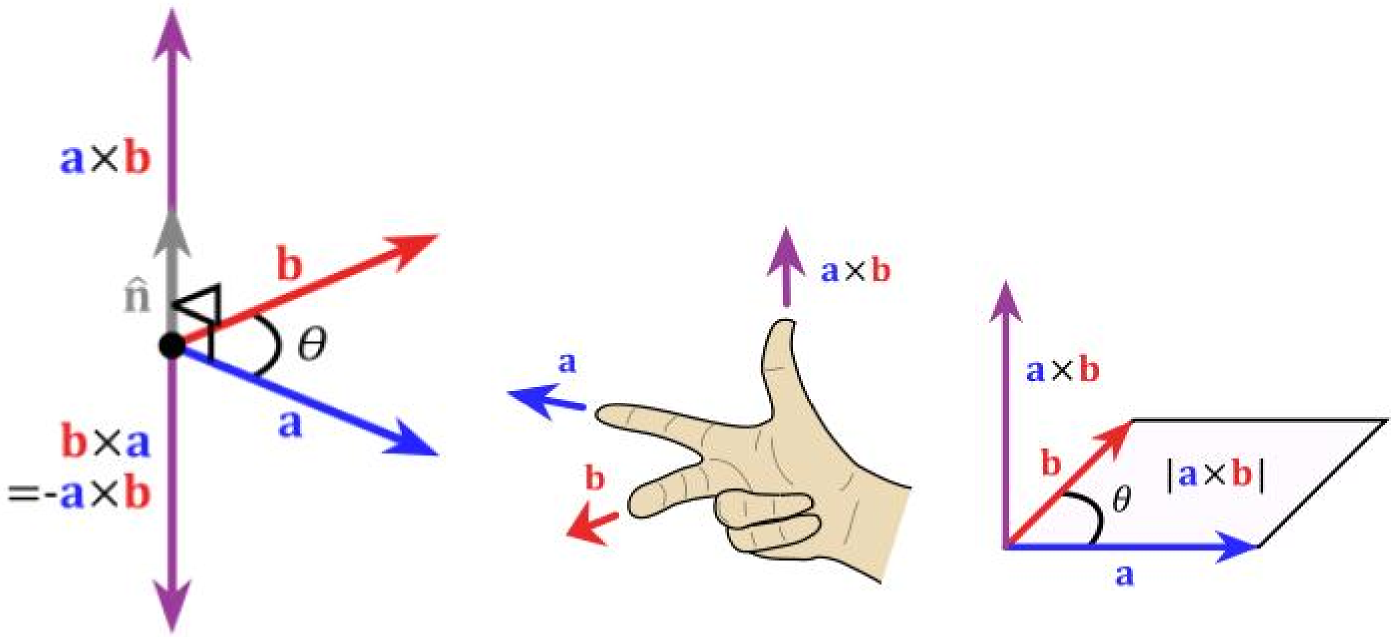


Fig. Right Hand Rule

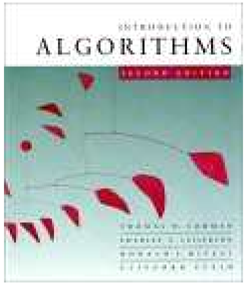
Usage:

determine face normal

determine point is in left or right space of plane

visible surface detection

ex) Cormen, Introduction to Algorithm



Point in Polygon

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 \, . \end{aligned}$$

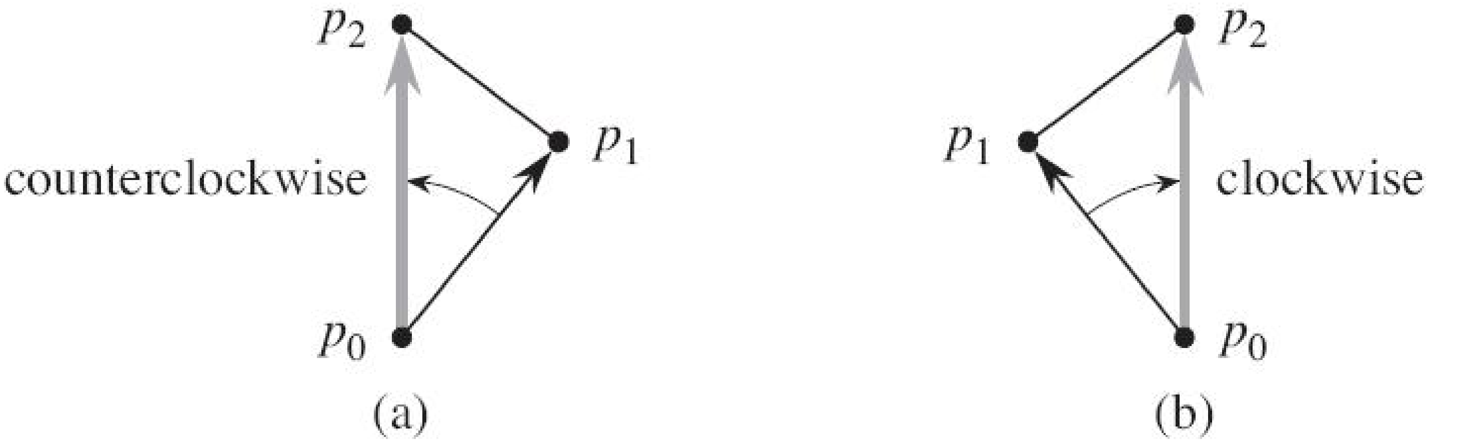
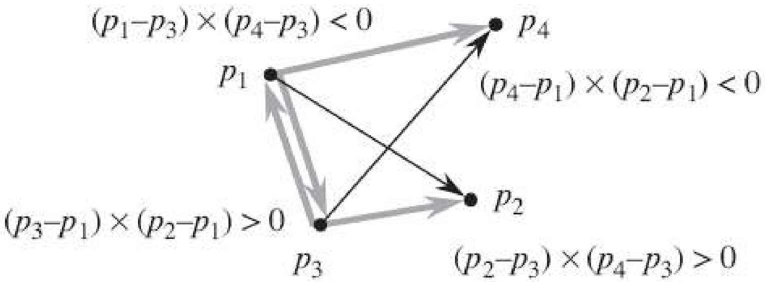
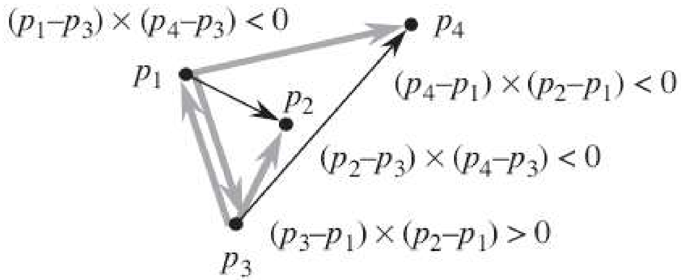


Figure 33.2 Using the cross product to determine how consecutive line segments p0p1 and p1p2 turn at point p1. We check whether the directed segment $\overrightarrow{p0p2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p0p1}$. (a) If counterclockwise, the points make a left turn. (b) If clockwise, they make a right turn.

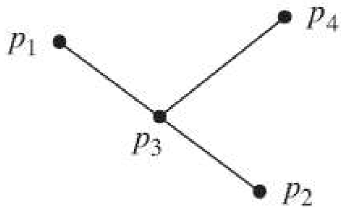
Two Line Segment Intersection



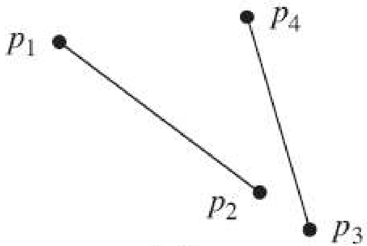
(a)



(b)



(c)



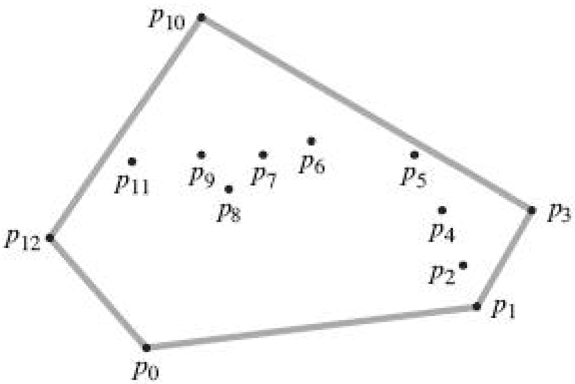
(d)

```
SEGMENTS-INTERSECT (p1, p2, p3, p4)
d1 = DIRECTION (p3, p4, p1)
d2 = DIRECTION (p3, p4, p2)
d3 = DIRECTION (p1, p2, p3)
d4 = DIRECTION (p1, p2, p4)
if ((d1 > 0 && d2 < 0 ) || (d1 < 0 && d2 > 0 )) and
   ((d3 > 0 && d4 < 0 ) || (d3 < 0 && d4 > 0 ))
    return TRUE
elseif d1 == 0 and ON-SEGMENT (p3, p4, p1)
    return TRUE
elseif d2 == 0 and ON-SEGMENT (p3, p4, p2)
    return TRUE
elseif d3 == 0 and ON-SEGMENT (p1, p2, p3)
    return TRUE
elseif d4 == 0 and ON-SEGMENT (p1, p2, p4)
    return TRUE
else return FALSE
```

```
DIRECTION (p_i, p_j, p_k)
1    return (p_k − p_i) × (p_j − p_i)
```

```
ON-SEGMENT (p_i, p_j, p_k)
1    if min(x_i, x_j) ≤ x_k ≤ max(x_i, x_j) and min(y_i, y_j) ≤ y_k ≤ max(y_i, y_j)
2        return TRUE
3    else return FALSE
```

Practice: Algorithm to determine a point in a polygon



Adding Dot() and Cross() for class KVector

```
inline KVector3 operator+(const KVector3& lhs, const KVector3& rhs)
{
    KVector3 temp(lhs.x + rhs.x, lhs.y + rhs.y, lhs.z + rhs.z);
    return temp;
}
inline KVector3 operator-(const KVector3& lhs, const KVector3& rhs)
{
    KVector3 temp(lhs.x - rhs.x, lhs.y - rhs.y, lhs.z - rhs.z);
    return temp;
}
inline float Dot(const KVector3& lhs, const KVector3& rhs)
{
    return lhs.x*rhs.x + lhs.y*rhs.y + lhs.z*rhs.z;
}
inline KVector3 Cross(const KVector3& u, const KVector3& v)
{
    KVector3 temp;
    temp.x = u.y*v.z - u.z*v.y;
    temp.y = u.z*v.x - u.x*v.z;
    temp.z = u.x*v.y - u.y*v.x;
    return temp;
}
```

Ex) Detecting invisible Surface

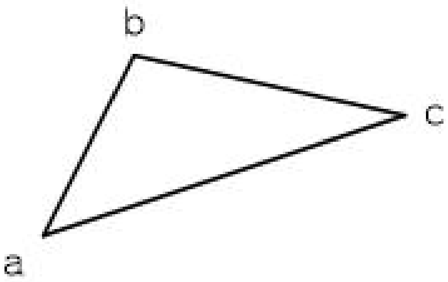


Fig. a triangle in mesh: decide this face is visible or not in camera position

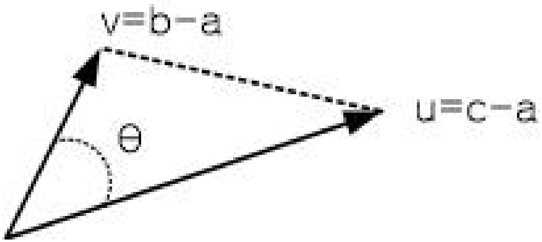


Fig. Calculating face normal

Normal Vector

u=c-a, v=b-a

cross product, u×v

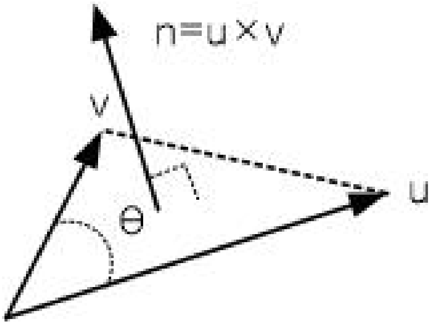


Fig. u×v is normal vector for triangle

Calculate normal for each triangle

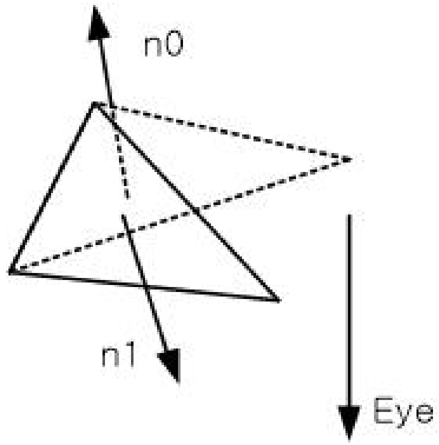


Fig. vector Eye look at camera

Dot product of Eye and normal

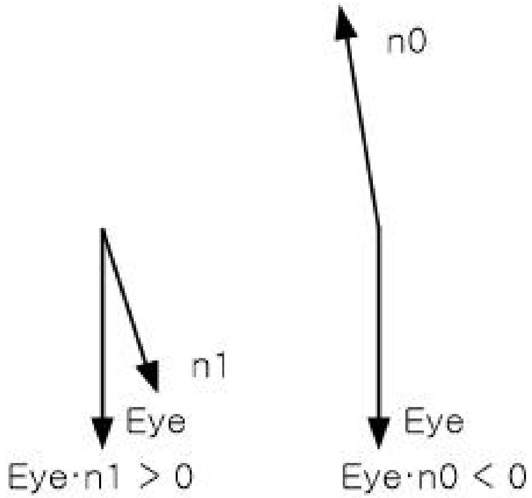


Fig. Face is visible if dot product is greater than 0

DrawIndexedPrimitive()

```
void DrawIndexedPrimitive( HDC hdc
    , int* m_indexBuffer            // index buffer
    , int primitiveCounter          // primitive counter
    , KVector3* m_vertexBuffer      // vertex buffer
    , COLORREF color )
{
    int   i0, i1, i2;
    int   counter = 0;

    for (int i = 0; i < primitiveCounter; ++i)
    {
        // get index
        i0 = m_indexBuffer[counter];
        i1 = m_indexBuffer[counter + 1];
        i2 = m_indexBuffer[counter + 2];

        KVector3 normal;
        normal  =  Cross(m_vertexBuffer[i0]  -  m_vertexBuffer[i1],  m_vertexBuffer[i0]  -
m_vertexBuffer[i2]);
        KVector3 forward(0, 0, 1);
        int penStyle = PS_SOLID;
        int penWidth = 3;
```

```
        if (Dot(forward, normal) > 0)
        {
            penStyle = PS_DOT;
            penWidth = 1;
        }

        // draw triangle
        KVectorUtil::DrawLine(hdc, m_vertexBuffer[i0].x, m_vertexBuffer[i0].y
            , m_vertexBuffer[i1].x, m_vertexBuffer[i1].y, penWidth, penStyle, color );
        KVectorUtil::DrawLine(hdc, m_vertexBuffer[i1].x, m_vertexBuffer[i1].y
            , m_vertexBuffer[i2].x, m_vertexBuffer[i2].y, penWidth, penStyle, color );
        KVectorUtil::DrawLine(hdc, m_vertexBuffer[i2].x, m_vertexBuffer[i2].y
            , m_vertexBuffer[i0].x, m_vertexBuffer[i0].y, penWidth, penStyle, color );

            // advance to next primitive
        counter += 3;
    }//for
}//DrawIndexedPrimitive()
```

vector forward

Rendering pipeline

1) Put a object in World space
2) Where to see
3) How to project

World transform
Viewing transform
Projection transform

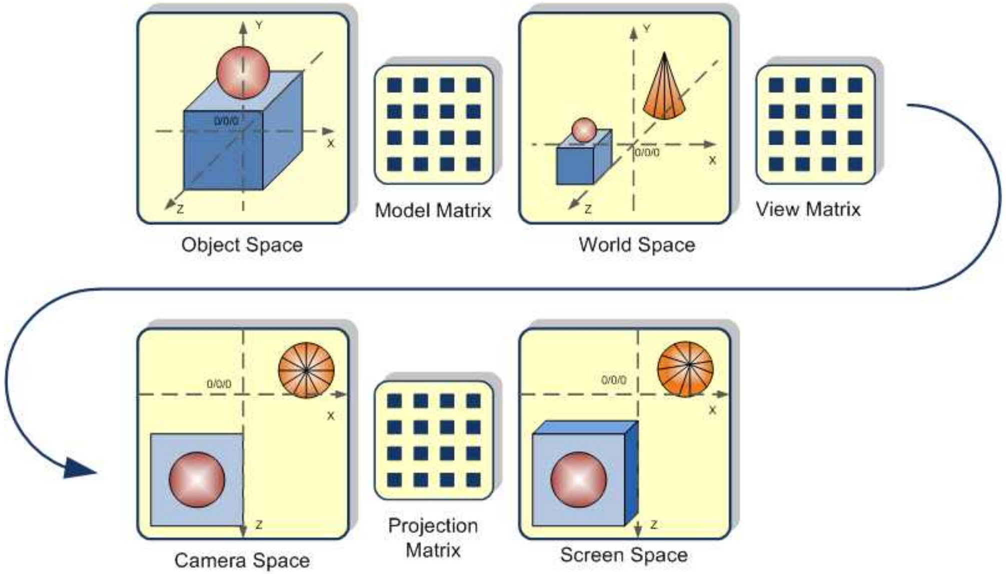


Fig. Graphics Pipeline

Render()

```
void OnRender(HDC hdc, float fElapsedTime_)
{
    KVectorUtil::DrawGrid(hdc, 30, 30);
    KVectorUtil::DrawAxis(hdc, 32, 32);
```

```
KPolygon        poly;
KMatrix4         matRotX;
KMatrix4         matRotY;
KMatrix4         matTrans;
KMatrix4         matTransform;
KMatrix4         matProjection;
static float    s_fTheta = 0.0f;

s_fTheta += fElapsedTime_ * 0.5f;
//matRotX.SetRotationX( 3.14f / 4.0f );
matRotX.SetRotationX(s_fTheta);
matRotY.SetRotationY(s_fTheta);
matTrans.SetTranslation( 5.f, 5.f, 0 );
//matTrans.SetTranslation(0.f, 0.f, 0);

matProjection.SetProjection(50.f);

matTransform = matTrans * matRotY * matRotX;
//matTransform = matRotY * matRotX*matTrans;

poly.SetIndexBuffer();
poly.SetVertexBuffer();
poly.Transform(matTransform);
//poly.Viewing( matViewing );
poly.Transform(matProjection);
```

```
    poly.Render(hdc);
}
```

Result

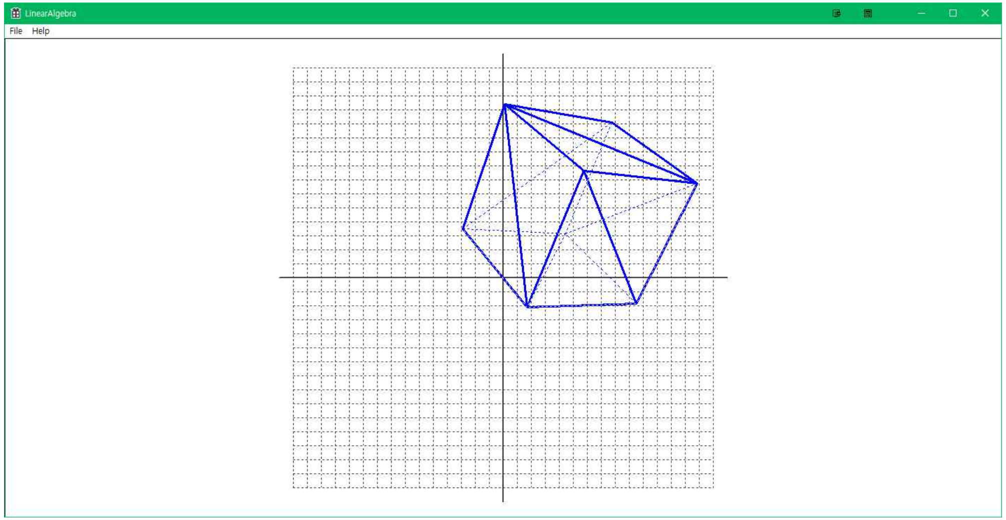
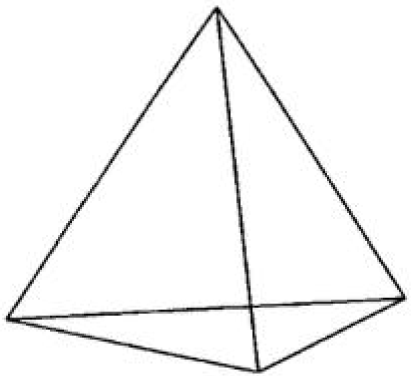


Fig. Rotating Cube

Practice: Define a tetrahedron and render it with back face culling.



tetrahedron

@