Polygon

class KVector3

```
#pragma once

class KVector3
{
public:
    float x;
    float y;
    float z;

public:
    KVector3(float x=0.0f, float y=0.0f, float z=0.0f);
    KVector3(int tx, int ty, int tz) { x = (float)tx; y = (float)ty; z = (float)tz; }
    ~KVector3();
};//class KVector3
```

Length(), Normalize()

Projection Transform

Parallel Projection, Isometric Projection

Primitive

Primitives in DirectX 11

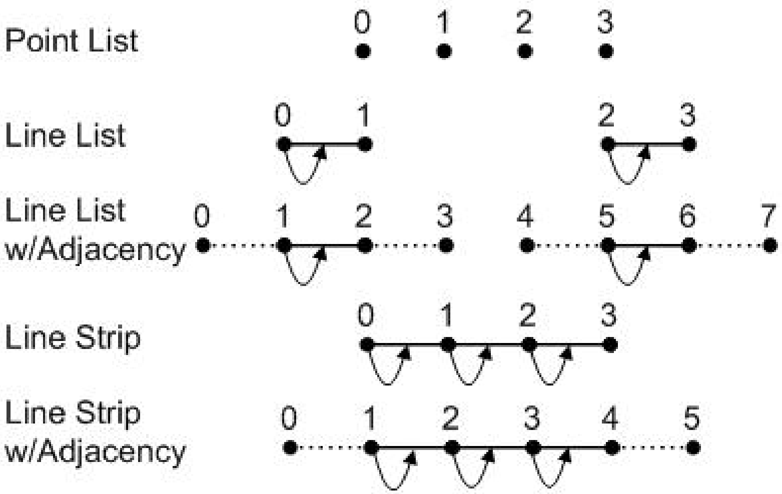


Fig. Primitive to draw points and line

Primitives to draw triangles

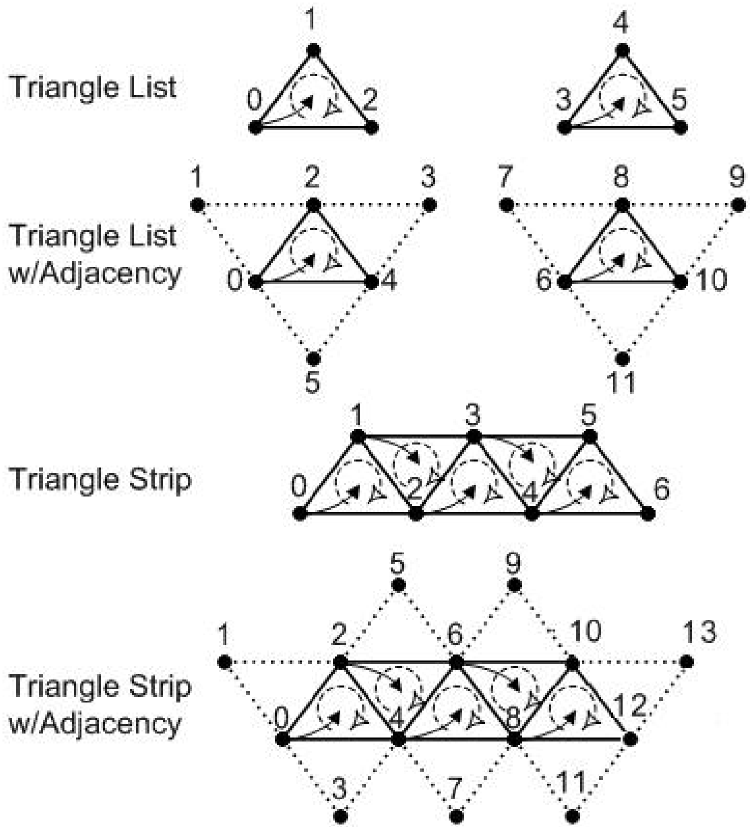


Fig. Primitives to draw triangle

Why Triangle?

One Plane, One Normal Vector

Mesh

Polygon

3D primitive

surface modelling

Triangle, Minimum Vertices and Convex Polygon

Vertex vs. Polygon

(Texture UV, Color, Point, Normal)

Indexed primitive

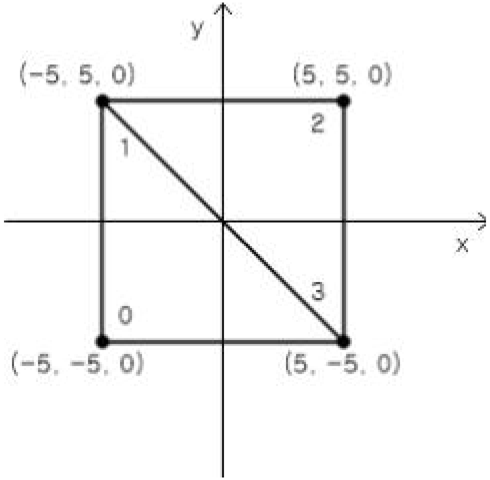


Fig. Rectangle with Indexed Primitive

vertex buffer

index buffer

2 Indices for Line

6 indices for Triangle

12 Indices for Rectangle

```
#include "KVector3.h"

class KPolygon
{
private:
    int         m_indexBuffer[100];
    int         m_sizeIndex;
    KVector3    m_vertexBuffer[100];
    int         m_sizeVertex;
    COLORREF    m_color;

public:
    KPolygon();
    ~KPolygon();

    void SetIndexBuffer();
    void SetVertexBuffer();
    void Render(HDC hdc);
    void SetColor(COLORREF color) { m_color = color; }
};//class KPolygon
```

array for simplicity

m_sizeIndex: number of indices in the index buffer

m_sizeVertex: number of vertices in the vertex buffer

Render()

Setting Vertex Buffer

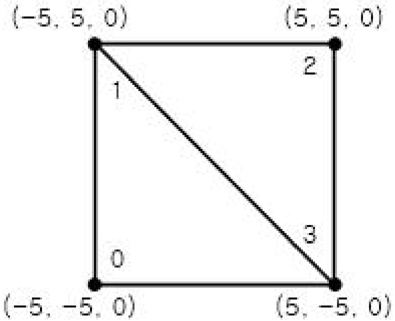


Fig. 4 Vertices for Rectangle

```
void KPolygon::SetVertexBuffer()
{
    m_vertexBuffer[0] = KVector3(-5.0f, -5.0f, 0.0f);
    m_vertexBuffer[1] = KVector3(-5.0f, 5.0f, 0.0f);
    m_vertexBuffer[2] = KVector3(5.0f, 5.0f, 0.0f);
    m_vertexBuffer[3] = KVector3(5.0f, -5.0f, 0.0f);
    m_sizeVertex = 4;
}//KPolygon::SetVertexBuffer()
```

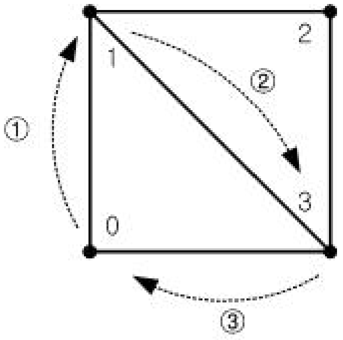


Fig. Triangle composed with 3 lines, 6 indices

Setting Index Buffer

```
void KPolygon::SetIndexBuffer()
{
    int buffer[] = {0,1,
                    1,3,
                    3,0,
                    1,2,
                    2,3,
                    3,1};
    for (int i=0; i<12; ++i)
```

```
    {
        m_indexBuffer[i] = buffer[i];
    }//for
    m_sizeIndex = 12;
}//KPolygon::SetIndexBuffer()
```

Indices are defined CW(Clockwise)

Render a Polygon

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

6: number of lines for primitive

DrawIndexedPrimitive()

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

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

        // draw line
        KVectorUtil::DrawLine(hdc, m_vertexBuffer[i1].x, m_vertexBuffer[i1].y
            , m_vertexBuffer[i2].x, m_vertexBuffer[i2].y, 2, PS_SOLID, color );

        // advance to next primitive
```

```
        counter += 2;
    }//for
}//DrawIndexedPrimitive()
```

Get 2 indices, then draw line between them.

Ignore z values

parallel projection --> perspective projection

MVC Design Pattern

(Model-View-Controller Design Pattern)

HAVE-A relationship in classes

Association, Aggregation and Composition

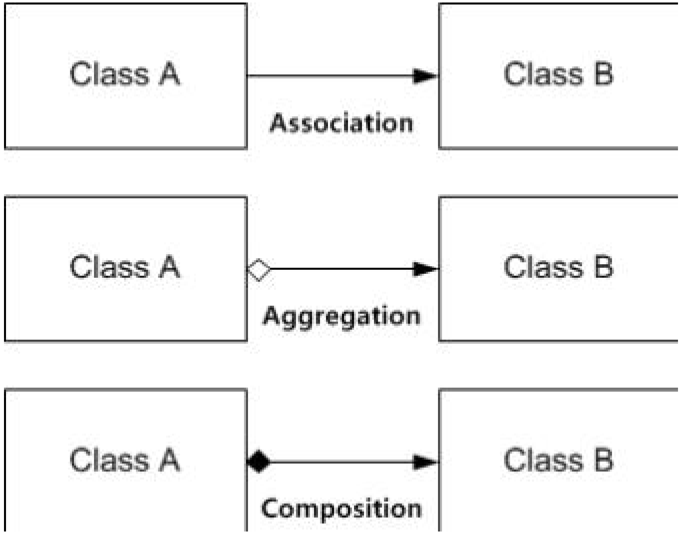


Fig. HAVE-A relationship: Association, Aggregation 및 Composition

Render() method in class KPolygon

① Is this class reusable when it moved to other platform?
② Is this class resuable when it moved to other project in the same platform?
--> Not good decision: implement Render() as a method of class KPolygon.

(1) Data
(2) Rendering
(3) Doing something with Data --> Implemented as Independent class.
(4) Control actions between Data and Rendering

(1) Model
(2) View
(4) Controller

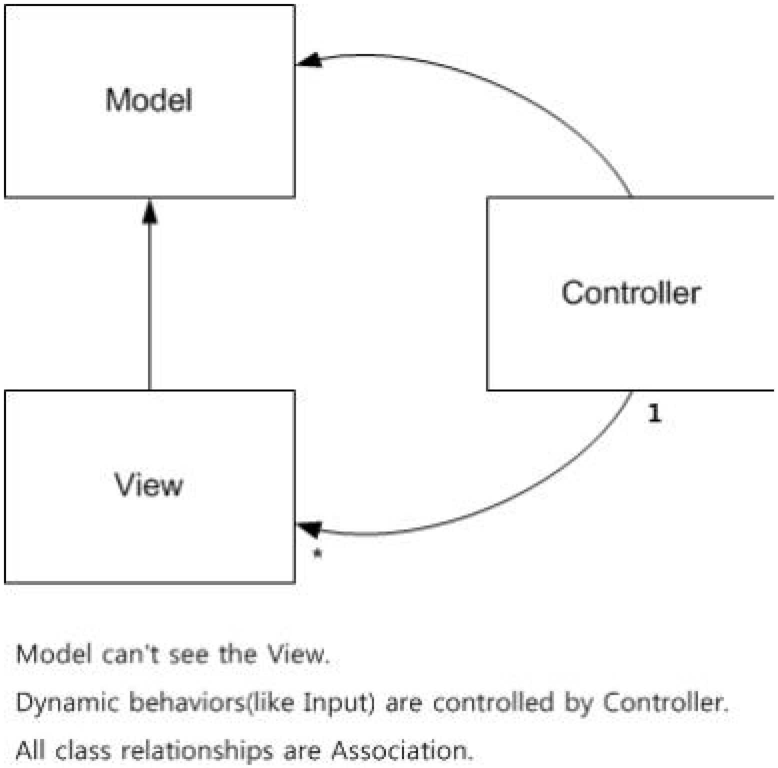


Fig. MVC Design Pattern

Document-View architecture of MFC(Microsoft Foundation Class)

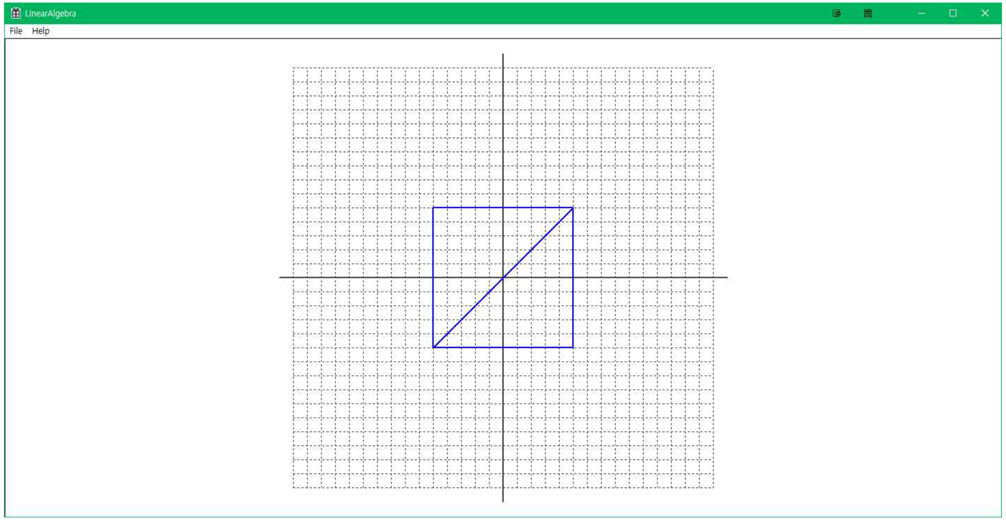
Character and Character Controller in Unity

Rendering Polygon

Implementing OnRender()

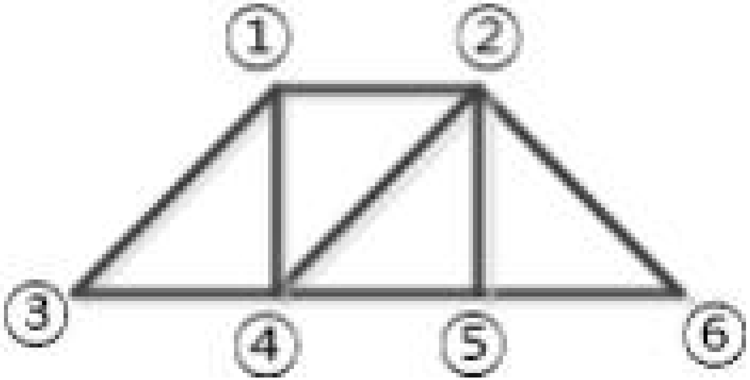
```
void OnRender(HDC hdc, float fElapsedTime_)
{
    KVectorUtil::DrawGrid(hdc, 30, 30);
    KVectorUtil::DrawAxis(hdc, 32, 32);

    KPolygon    poly;
    poly.SetIndexBuffer();
    poly.SetVertexBuffer();
    poly.Render(hdc);
}
```

Result



Practice

1) Define a trapezoid then draw it.



Step08: 3-dimension

3×3 Homogeneous matrix for 2D transformation
4×4 Homogeneous matrix for 3D transformation
2D rotation is 3D rotation about z-axis

3×3 Matrix for 2D rotation

$$\begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

4×4 Matrix for 3D rotation

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

3D Transform Matrices

1) translation

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

2) scaling

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} t_x & 0 & 0 & 0 \\ 0 & t_y & 0 & 0 \\ 0 & 0 & t_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

3) Rotation about z-axis

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

4) Rotation about x-axis

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

5) Rotation about y-axis

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin thea & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

composite transform

A: Rotation transform

B: Translation Transform

AB

Translation then Rotation

KMatrix4

class KMatrix4

```
#pragma once
#include "KVector3.h"

class KMatrix4
{
public:
    float   m_afElements[4][4];

    KMatrix4();
    ~KMatrix4();

    /// access matrix element (iRow,iCol)
    /// @param  iRow: row index
    /// @param  iCol: column index
    /// @return reference of element (iRow,iCol)
    float& operator()(int iRow, int iCol);
```

```
    KMatrix4 operator*(KMatrix4& mRight);
    KVector3 operator*(KVector3& vRight);
    KMatrix4 operator+(KMatrix4& mRight);
    KMatrix4& operator=(KMatrix4& mRight);

    KMatrix4 SetZero();
    KMatrix4 SetIdentity();
    KMatrix4 SetRotationX(float fRadian);
    KMatrix4 SetRotationY(float fRadian);
    KMatrix4 SetRotationZ(float fRadian);
    KMatrix4 SetScale(float fxScale, float fyScale, float fzScale);
    KMatrix4 SetTranslation(float x, float y, float z);
};//class KMatrix4
```

operator()() : access element

operator*(KVector3& rhs)

Note: There must be homogeneous divide.

operator*(KMatrix4&)

```
KMatrix4 KMatrix4::operator*(KMatrix4& mRight)
{
    KMatrix4 mRet;

    mRet.SetZero();
    for (int i=0; i<4; ++i)
    {
        for (int j=0; j<4; ++j)
        {
            for (int k=0; k<4; ++k)
            {
                mRet(i,j) += m_afElements[i][k] * mRight(k,j);
            }//for
        }//for
    }//for

    return mRet;
}//KMatrix4::operator*()
```

operator*(KVector3&)

```
KVector3 KMatrix4::operator*(KVector3& vLeft)
{
    KVector3 vRet;

    vRet.x = vLeft.x * m_afElements[0][0] +
             vLeft.y * m_afElements[0][1] +
             vLeft.z * m_afElements[0][2] +
             m_afElements[0][3];
    vRet.y = vLeft.x * m_afElements[1][0] +
             vLeft.y * m_afElements[1][1] +
             vLeft.z * m_afElements[1][2] +
             m_afElements[1][3];
    vRet.z = vLeft.x * m_afElements[2][0] +
             vLeft.y * m_afElements[2][1] +
             vLeft.z * m_afElements[2][2] +
             m_afElements[2][3];
    const float w = vLeft.x * m_afElements[3][0] +
        vLeft.y * m_afElements[3][1] +
        vLeft.z * m_afElements[3][2] +
        1.0f * m_afElements[3][3];
```

```
    vRet.x /= w; // homogeneous divide
    vRet.y /= w;
    vRet.z /= w;
    return vRet;
}//KMatrix4::operator*()
```

Hohogeneous divide

-->after transformation, w must be 1.

Rotation about z-axis

$$\begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

SetRotationZ()

```
KMatrix4 KMatrix4::SetRotationZ(float fRadian)
{
    SetIdentity();
    m_afElements[0][0] = cosf(fRadian);
    m_afElements[0][1] = -sinf(fRadian);
    m_afElements[1][0] = sinf(fRadian);
    m_afElements[1][1] = cosf(fRadian);
    return *this;
}//KMatrix4::SetRotationZ()
```

Full source of class KMatrix4.cpp

```
#include "stdafx.h"
#include "KMatrix4.h"
#include <math.h>

KMatrix4::KMatrix4()
{
    SetIdentity();
}//KMatrix4::KMatrix4()

KMatrix4::~KMatrix4()
{
}//KMatrix4::~KMatrix4()

float& KMatrix4::operator()(int iRow, int iCol)
{
    return m_afElements[iRow][iCol];
}//KMatrix4::operator()

KMatrix4 KMatrix4::operator*(KMatrix4& mRight)
{
```

```
    KMatrix4 mRet;

    mRet.SetZero();
    for (int i=0; i<4; ++i)
    {
        for (int j=0; j<4; ++j)
        {
            for (int k=0; k<4; ++k)
            {
                mRet(i,j) += m_afElements[i][k] * mRight(k,j);
            }//for
        }//for
    }//for

    return mRet;
}//KMatrix4::operator*()

KVector3 KMatrix4::operator*(KVector3& vLeft)
{
    KVector3 vRet;

    vRet.x = vLeft.x * m_afElements[0][0] +
               vLeft.y * m_afElements[0][1] +
```

```
                vLeft.z * m_afElements[0][2] +
                m_afElements[0][3];
    vRet.y = vLeft.x * m_afElements[1][0] +
                vLeft.y * m_afElements[1][1] +
                vLeft.z * m_afElements[1][2] +
                m_afElements[1][3];
    vRet.z = vLeft.x * m_afElements[2][0] +
                vLeft.y * m_afElements[2][1] +
                vLeft.z * m_afElements[2][2] +
                m_afElements[2][3];
    const float w = vLeft.x * m_afElements[3][0] +
        vLeft.y * m_afElements[3][1] +
        vLeft.z * m_afElements[3][2] +
        1.0f * m_afElements[3][3];
    vRet.x /= w; // homogeneous divide
    vRet.y /= w;
    vRet.z /= w;
    return vRet;
}//KMatrix4::operator*()

KMatrix4 KMatrix4::operator+(KMatrix4& mRight)
{
    KMatrix4 mRet;
```

```
    for (int i=0; i<4; ++i)
    {
        for (int j=0; j<4; ++j)
        {
            mRet(i,j) = m_afElements[i][j] + mRight(i,j);
        }//for
    }//for

    return mRet;
}//KMatrix4::operator+()

KMatrix4& KMatrix4::operator=(KMatrix4& mRight)
{
    memcpy( m_afElements, mRight.m_afElements, sizeof(m_afElements) );
    return *this;
}//KMatrix4::operator=()

KMatrix4 KMatrix4::SetZero()
{
    memset( m_afElements, 0, sizeof(m_afElements) );

    return *this;
```

```
}//KMatrix4::SetZero()

KMatrix4 KMatrix4::SetIdentity()
{
    SetZero();

    m_afElements[0][0] =
    m_afElements[1][1] =
    m_afElements[2][2] =
    m_afElements[3][3] = 1.f;

    return *this;
}//KMatrix4::SetIdentity()

KMatrix4 KMatrix4::SetRotationX(float fRadian)
{
    SetIdentity();
    m_afElements[1][1] = cosf(fRadian);
    m_afElements[1][2] = -sinf(fRadian);
    m_afElements[2][1] = sinf(fRadian);
    m_afElements[2][2] = cosf(fRadian);
    return *this;
}//KMatrix4::SetRotationX()
```

```
KMatrix4 KMatrix4::SetRotationY(float fRadian)
{
    SetIdentity();
    m_afElements[0][0] = cosf(fRadian);
    m_afElements[0][2] = sinf(fRadian);
    m_afElements[2][0] = -sinf(fRadian);
    m_afElements[2][2] = cosf(fRadian);
    return *this;
}//KMatrix4::SetRotationY()

KMatrix4 KMatrix4::SetRotationZ(float fRadian)
{
    SetIdentity();
    m_afElements[0][0] = cosf(fRadian);
    m_afElements[0][1] = -sinf(fRadian);
    m_afElements[1][0] = sinf(fRadian);
    m_afElements[1][1] = cosf(fRadian);
    return *this;
}//KMatrix4::SetRotationZ()

KMatrix4 KMatrix4::SetScale(float fxScale, float fyScale, float fzScale)
{
```

```
    SetIdentity();
    m_afElements[0][0] = fxScale;
    m_afElements[1][1] = fyScale;
    m_afElements[2][2] = fzScale;

    return *this;
}//KMatrix4::SetScale()

KMatrix4 KMatrix4::SetTranslation(float x, float y, float z)
{
    SetIdentity();
    m_afElements[0][3] = x;
    m_afElements[1][3] = y;
    m_afElements[2][3] = z;

    return *this;
}//KMatrix4::SetTranslation()
```

Transform Polygon

Add Transform() to the class KPolygon

```
#include "KMatrix4.h"

class KPolygon
{
private:
    int         m_indexBuffer[100];
    int         m_sizeIndex;
    KVector3    m_vertexBuffer[100];
    int         m_sizeVertex;
    COLORREF    m_color;

public:
    KPolygon();
     ~KPolygon();

    void SetIndexBuffer();
```

```
    void SetVertexBuffer();
    void Render(HDC hdc);
     void SetColor(COLORREF color) { m_color = color; }

    void Transform(KMatrix4& mat);
};//class KPolygon
```

Transform()

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

Rotation(), Scale(), Translation()? No! Transform()

Render() in LinearAlgebra.cpp

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

    KPolygon        poly;
    static float    s_fTheta = 0.0f;

    poly.SetIndexBuffer();
    poly.SetVertexBuffer();
    KMatrix4    rotX;
    KMatrix4    rotY;
    KMatrix4    translate;
    KMatrix4    transform;
    rotX.SetRotationX(3.141592f / 4.0f);
    rotY.SetRotationY(s_fTheta);
    s_fTheta += fElapsedTime_;
    translate.SetTranslation(5.0f, 5.0f, 0);
    transform = translate * rotY * rotX;
    poly.Transform(transform);
```

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

Result

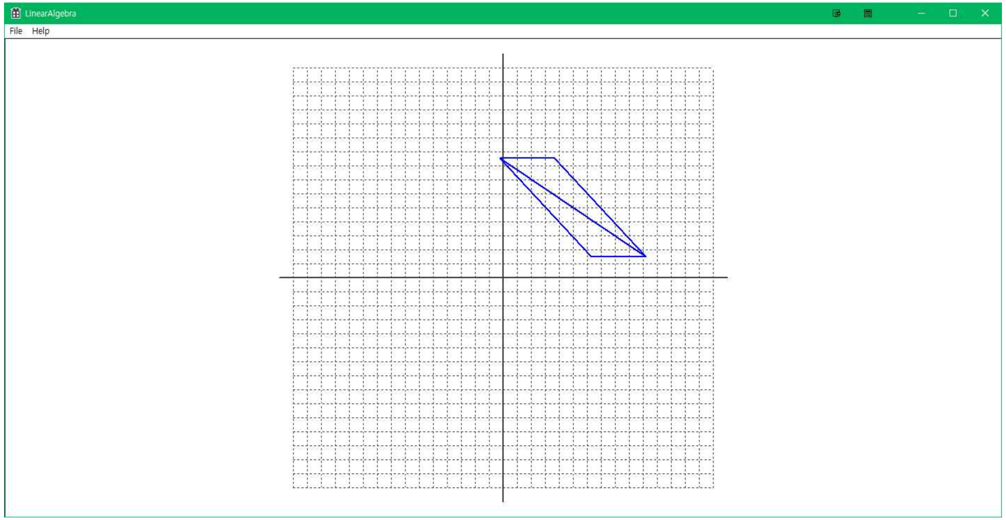
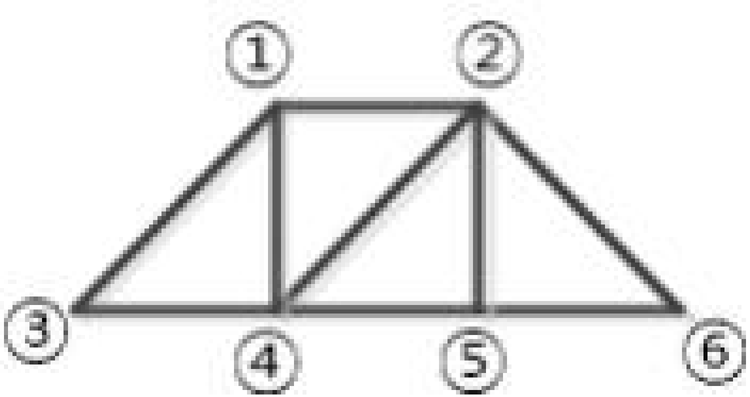


Fig. Rotating Rectangle in wireframe

Practice

1) Define a trapezoid polygon in 3D sapce then rotate it.



@