폴리곤(polygon, 다각형)

KVector3 클래스

우리는 3차원 공간상에 위치한 2차원 다각형을 정의하기를 원합니다. 3차원 공간의 다각형의 한 점을 표현하기 위해서 KVector3을 정의할 수 있습니다.

```
#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
```

KVector3은 3차원 점의 위치를 표현하기 위해서 x, y와 z값을 가집니다. 그리고 아직은 생성자와 파괴자외에 특별한 멤버 함수는 가지지 않습니다. KVector2에서 정의했던, Length(), Normalize()등은 아직 정의하지 않습니다. 3차원 물체를 2차원에 투영할 때, 단순하게 z값을 무시해 버리면, 현실적으로 보이지는 않지만 아주 쉽게 3차원 물체를 2차원에 그리는 것이 가능합니다. 그것은 투영 변환(Projection Transform)에서 **평행 투영(Parallel Projection)** 혹은 **등축 투영(Isometric Projection)**으로 알려져 있습니다.

우리는 Step7에서 폴리곤을 그리기 위해서, 또한 Step8에서 3차원 공간에서 회전된 폴리곤을 그리기 위해서도, 평행 투영을 사용할 것입니다.

프리미티브(Primitive)

도형을 표현하기 위한 기본 구조를 **프리미티브(Primitive)**라고 합니다. 아래는 DirectX 11이 지원하는 프리미티브들 입니다.

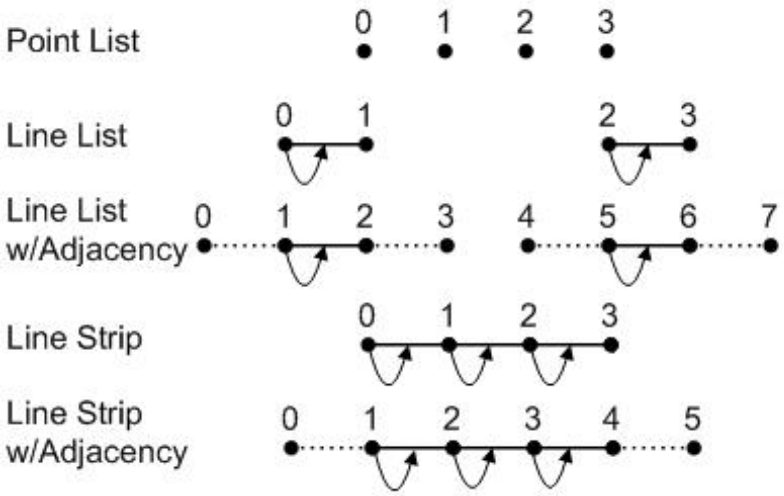


그림. 점과 라인을 그리기 위한 프리미티브가 있습니다.

점을 그리거나 라인을 그리는 프리미티브가 있습니다. 점이나 라인을 그릴 때, 점의 굵기나 라인의 굵기를 설정하는 것은 불가능합니다. 프리미티브가 정의되는 공간이, 2차원 평면이 아니라 3차원 공간이기 때문에 그렇습니다. 3차원 렌더링 라이브러리를 구현할 때, 최종 변환된 결과를 2차원 평면에 선으로 그려야 한다면, 선의 굵기나 색을 설정하는 것이 가능한데, 이 기능은 어떤 이유에서인지는 몰라도 구현되어 있지 않습니다.

그리고 물체를 나타내기 위해 물체의 표면을 삼각형의 조합으로 나타냅니다. 그렇게 하기 위해서 삼각형을 그리는 다양한 프리미티브가 있습니다.

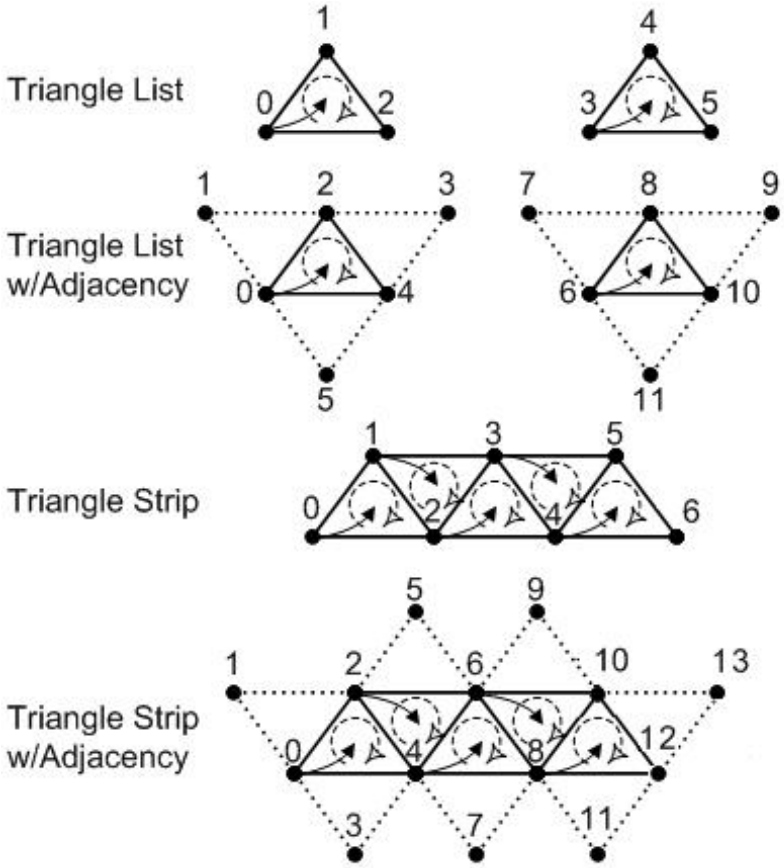


그림. 메시의 표면을 그리기 위한 프리미티브가 있습니다. 모두 삼각형을 사용합니다. 삼각형의 내부는 색이나 이미지로 채울 수 있습니다.

삼각형은 3개의 점으로 구성되며, 3차원 공간상에서 항상 하나의 평면을 결정하므로, 단 하나의 법선 벡터(Normal Vector)를 가집니다. 그래서 메시(Mesh)는 모두 삼각형을 사용하여 표현합니다.

폴리곤(Polygon)

3차원 **프리미티브(primitive, 기본 구성요소)**를 표현하는 방식은 여러가지입니다. 일반적으로 3차원 물체의 속을 전부 표현할 수 없으므로, 3차원 물체의 표면(surface)을 폴리곤(polygon)들로 표현하는데 이렇게 표현된 3차원 물체를

메시(mesh)라고 합니다. 폴리곤의 기본 단위로 삼각형(triangle)을 사용하는데, 이유는 **3개의 최소 버텍스(vertex)□로 표현되는 항상 볼록한 다각형**이기 때문입니다.

□ 폴리곤의 꼭지점을 점(point)이라고 하지 않고 정점(버텍스, vertex)라고 할 때, 버텍스는 점(point)의 위치 정보 (x,y,z) 이외에 추가적인 정보를 가진다는 의미입니다. 이러한 정보에는 텍스쳐(texture) 좌표 (u,v), 점의 색 (r,g,b,a), 점의 노멀(normal)벡터 (x,y,z) 등이 있습니다.

우리는 구현을 간단하게 하기 위해, 표면(surface)은 고려하지 않고 삼각형을 구성하는 선(line)만을 고려할 것입니다. 버텍스의 갯수와 선의 개수는 정형적인 비례관계가 없습니다. 이러한 것을 효과적으로 나타내기 위해 **인덱스드 프리미티브(indexed primitive)**를 사용할 수 있습니다. 아래 그림을 보세요.

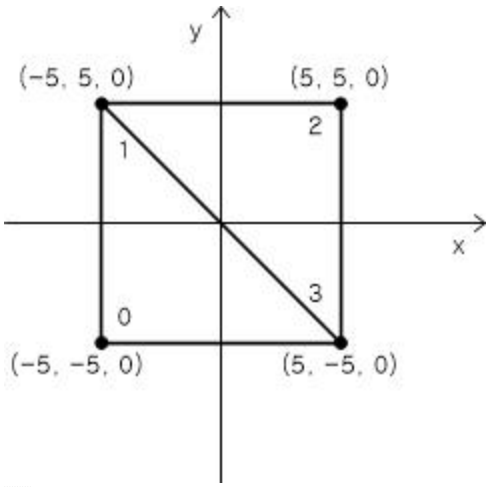


그림. 정사각형 표현: 표준 좌표계 - 오른손 좌표계를 사용하므로 화면속으로 들어가는 방향이 -z축이 됩니다 - 에서 정사각형을 정의합니다. 사각형의 왼쪽 아래 첫번째 삼각형을 구성하는 선들의 인덱스 표현은 (0,1), (1,3)와 (3,0)입니다. 오른쪽 위 두번째 삼각형의 인덱스는 (1,2), (2,3)과 (3,1)입니다. 그러므로 우리는 위 사각형을 렌더링 하기 위해 4개의 버텍스와 6개의 인덱스 쌍(pair)이 필요합니다.

다각형을 구성하는 버텍스를 별도의 버퍼에 관리하고 - 이것을 **버텍스 버퍼(vertex buffer)**라고 합니다 - 각 버텍스의 일련번호를 인덱스로 하는 별도의 버퍼 - 이것을 **인덱스 버퍼(index buffer)**라 합니다 - 를 관리합니다. 선을 그리기 위해서는 2개의 인덱스가 필요합니다. 삼각형을 그리기 위해서는 6개의 인덱스가 필요하며, 삼각형 2개로 구성된 사각형을 표현하기 위해서는 12개의

인덱스, 즉 6쌍의 인덱스가 필요하게 됩니다. 그래서 다각형을 나타내기 위한 다각형 클래스는 버텍스 버퍼와 인덱스 버퍼를 멤버로 가집니다. 구현된 다각형 클래스의 헤더는 다음과 같습니다.

```
#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
```

구현을 간단하게 하기 위해, 버텍스 버퍼와 인덱스 버퍼 모두 크기가 고정된 배열을 사용하였고, m_sizeIndex는 인덱스 버퍼에 들어 있는 인덱스의 갯수를, m_sizeVertex는 버텍스 버퍼에 들어있는 버텍스의 갯수를 의미합니다. 또한, 인덱스 버퍼와 버텍스 버퍼를 설정하는 함수를 가지며, 인덱스와 버텍스 버퍼로 표현되는 다각형을 렌더링하는 Render()함수를 가집니다.

그림에 표현된 정사각형을 위해서 버텍스 버퍼를 설정하는 함수는 다음과 같습니다.

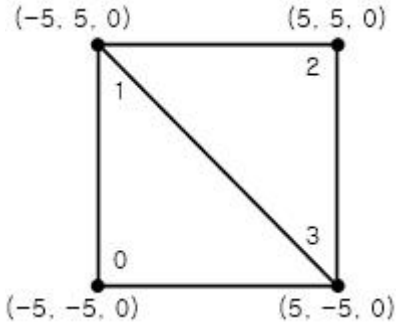


그림. 사각형 폴리곤을 위해서는 4개의 정점이 필요합니다.

```
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()
```

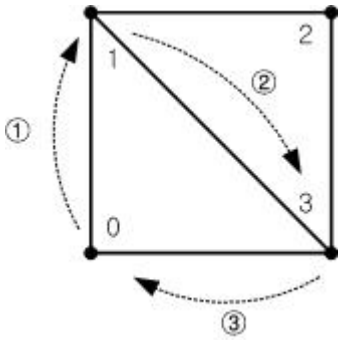


그림. 삼각형은 3개의 선으로 이루어져 있으므로, 6개의 인덱스가 필요합니다.

이제 인덱스를 설정하는 함수를 다음과 같이 작성할 수 있습니다.

```
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()
```

인덱스의 순서가 시계방향으로 정의되어 있다는 사실에 주목하세요. 지금은 사용하지 않지만, 이 순서는 면의 방향을 결정하는데 사용하므로 중요합니다. 지금의 예에서는 삼각형의 정점이 시계방향(CW, Clockwise)으로 사용된다고 가정하였습니다.

폴리곤의 렌더링 멤버 함수는 단순하게 DrawIndexedPrimitive()를 호출합니다.

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

DrawIndexedPrimitive()의 세번째 파라미터는 폴리곤을 구성하는 선의 개수입니다. 삼각형 2개로 구성된 사각형이므로 6을 전달합니다.

그러면 인덱스 버퍼와 버텍스 버퍼를 파라미터로 받아서 라인을 그리는 DrawIndexedPrimitive()를 다음과 같이 작성할 수 있습니다.

```
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()
```

함수의 동작은 어렵지 않습니다. 인덱스 버퍼에서 버텍스를 가리키는 2개의 인덱스를 차례대로 꺼내서 버텍스를 접근합니다. 그리고 선을 그립니다. 소스에서 특별히 주의할 점은 z값을 무시한다는 것입니다. 이것은 3차원 공간상의 점들의 z축에 의한 깊이 정보가 무시되므로, 멀리 있는 점이든 가까이 있는 점이든 xy-평면(xy-plane)상에서 같은 위치에 표현된다는 것을 의미합니다. 이것을 **평행 투영(parallel projectio)**이라고 하는데, 후에 **원근 투영(perspective projection)**으로 이 문제를 해결할 것입니다.

MVC 디자인 패턴(Model-View-Controller Design Pattern)

클래스간의 HAVE-A 관계는 크게 세가지 종류가 있습니다. 그것은

Association, Aggregation 및 Composition입니다.

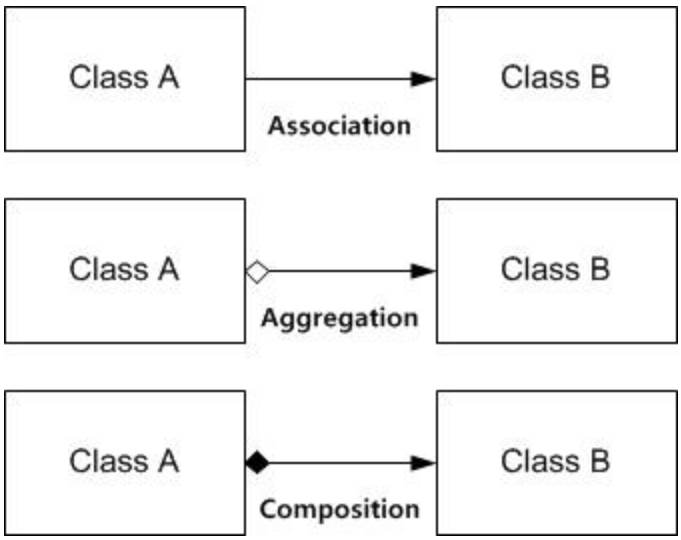


그림. 클래스 간의 HAVE-A 관계: Association, Aggregation 및 Composition

KPolygon이 Render()함수를 가지는 문제

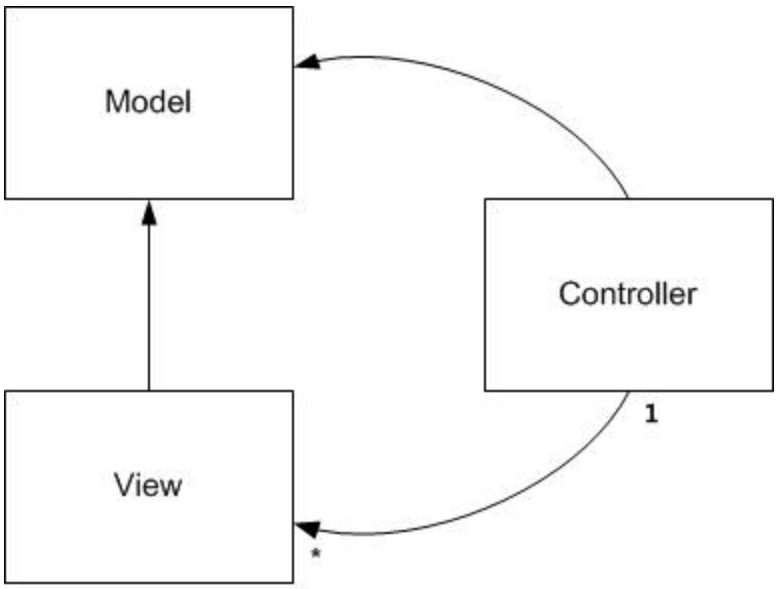
게임에 사용되는 범용 클래스를 제작할 때, 렌더 함수를 클래스의 멤버로 포함할 때는 신중한 결정이 필요합니다. KPolygon에 Render()를 포함하기로 한 결정에 대해서 생각해 봅시다.

(영상에서 MVC패턴에 대해서 설명할 것)
① 이 클래스를 다른 플랫폼으로 가져가면 재사용 가능한가?
② 이 클래스를 다른 프로젝트에 가져가면 재사용 가능한가?
--> Render()를 KPolygon의 멤버 함수로 구현하는 것은 좋은 설계가 아니다.

일반적으로 재사용 가능한 클래스를 설계할 때, (1) 데이터와 (2) 데이터를 가지고 렌더링 하는 부분 및 (3) 데이터를 참고하여 동적인 제어를 담당하는 부분을 분리하는 것은 좋은 습관 입니다. 이러한 디자인 패턴은 MVC패턴으로 알려져 있습니다.

(1)에 해당하는 부분을 Model, (2)에 해당하는 부분을 View, 그리고 (3)에 해

당하는 부분을 Controller라고 합니다.



Model can't see the View.

Dynamic behaviors(like Input) are controlled by Controller.

All class relationships are Association.

그림. MVC 디자인 패턴: 모델과 뷰를 분리하는 것은 좋은 프로그래밍 습관입니다. 이것을 확인하는 가장 좋은 질문은 다음과 같습니다. '이 클래스를 다른 플랫폼으로 가져가면 재사용 가능한가?' 이 질문에 '예'라고 답하기 위해서 일반적으로 그리기 함수는 클래스의 멤버가 되어서는 안 됩니다.

MFC(Microsoft Foundation Class) 라이브러리의 다큐먼트-뷰 구조(Document-View Architecture)라든가, Unity 엔진에서 캐릭터와 캐릭터 컨트롤러가 분리된 것 등, MVC는 다양한 플랫폼에서 쉽게 찾아 볼 수 있습니다.

KPolygon에서 Render()를 분리하는 것은 실습문제로 남깁니다.

폴리곤 렌더링

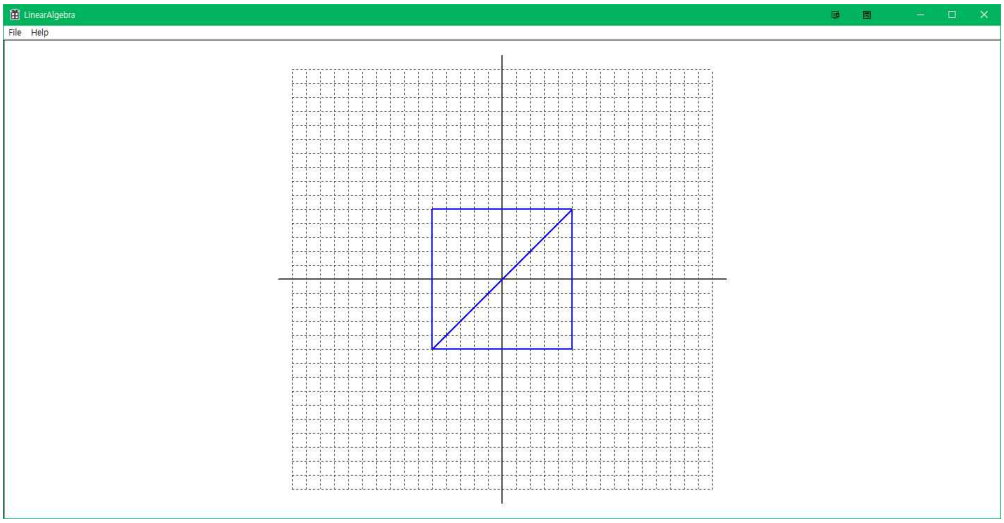
이제 폴리곤 객체를 생성하고 렌더링하는 함수를 아래와 같이 작성할 수 있

습니다.

```
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);
}
```

실행 결과는 다음과 같습니다.



Step08: 3차원으로의 확장

2차원 변환에서 3×3 호모지니어스 매트릭스를 변환 매트릭스로 사용하므로, 3차원 변환에서는 4×4 호모지니어스 매트릭스를 사용해야합니다. 2차원

공간에서 xy-평면상에서 회전하는 것은 3차원 공간에서 z축을 중심으로 회전하는 변환입니다.

2차원 회전 변환은 다음과 같은 3×3 행렬을 사용하여 나타냅니다.

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

이제 3차원 z축 회전 변환을 나타내기 위해서, 아래와 같이 기본 베이시스 (0,0,1)을 추가하여 4×4 행렬로 나타낼 수 있습니다.

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

이러한 방식으로 3차원 변환 매트릭스를 유도할 수 있습니다. 3차원에서 각 변환 매트릭스는 다음과 같습니다. 이를 유도하는 과정은 설명하지 않습니다 (실습문제 10).

1) 위치 변환(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) 크기 변환(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) z축 회전 변환

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

4) x축 회전 변환

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

5) y축 회전 변환

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

변환 매트릭스들이 주어졌을 때, 변환 매트릭스의 곱은 **합성 변환 (composite transform)**으로 나타냅니다. 예를 들면, A가 회전 변환, B가 위치 변환 매트릭스라고 가정해 봅시다.

AB

위 매트릭스곱은 (벡터가 오른쪽에 있는 열벡터라 가정하므로) B 위치 변환 후의, A회전 변환을 의미합니다. 열벡터가 매트릭스의 오른쪽에 있다고 가정하므로, 변환의 순서에 주의하세요.

KMatrix4

이제 3차원 변환에 사용할 KMatrix4를 구현합니다. 클래스의 헤더는 다음과 같습니다.

```
#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
```

KMatrix4 클래스는 행렬의 요소를 접근하기 위한 operator()()함수를 가집니다. 행렬간의 곱셈을 위한 operator*()함수를 가지고, 행렬과 KVector3과의 곱셈을 위한 operator*()함수를 가집니다. 이 부분을 구현할 때, 호모지니어스 나눗셈이 필요한다는 것을 명심하세요.

우리는 operator*(KMatrix4&) 함수를 다음과 같이 구현할 수 있습니다.

```
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*()
```

우리는 operator*(KVector3&) 함수를 다음과 같이 구현할 수 있습니다.

```
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*()
```

w를 구한 다음에, 호모지니어스 나눗셈을 적용하여 w를 1로 만드는 과정에 반드시 포함되어야 합니다.

z축에 대한 회전 변환은 다음과 같은 행렬입니다.

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

그러므로 SetRotationZ() 함수를 다음과 같이 구현할 수 있습니다.

```
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.cpp의 전체 소스는 다음과 같습니다.

```
#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()
```

다각형 변환 함수

다각형을 변환하는 멤버 함수는 Transform()입니다. Transform()은 변환 행렬을 인자로 받아서 다각형을 구성하는 모든 점들에 대해서 변환을 수행합니다. Step07에서 작성한 KPolygon클래스에 Transform() 멤버 함수를 추가합니다.

```
#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
```

폴리곤을 변환하는 Transform() 함수를 다음과 같이 추가할 수 있습니다.

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

Transform()은 폴리곤을 구성하는 모든 정점에 대해서 변환을 수행하는 간단한 함수입니다. 그리고, 회전, 크기, 이동 변환을 위해 각각의 함수가 존재하는 것이 아니라, 하나의 Transform()함수로 충분합니다.

마지막으로 LinearAlgebra.cpp에 위치한 Render()함수를 다음과 같이 수정합니다.

```
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);
}
```

수정된 Render()함수는 다각형을 x축을 중심으로 45도 회전하고, 다시 y축에 대해 일정한 각 만큼을 회전한 다음, 월드 축상의 (5, 5, 0)위치로 이동합니다. 프로그램의 출력결과는 아래 그림과 같습니다.

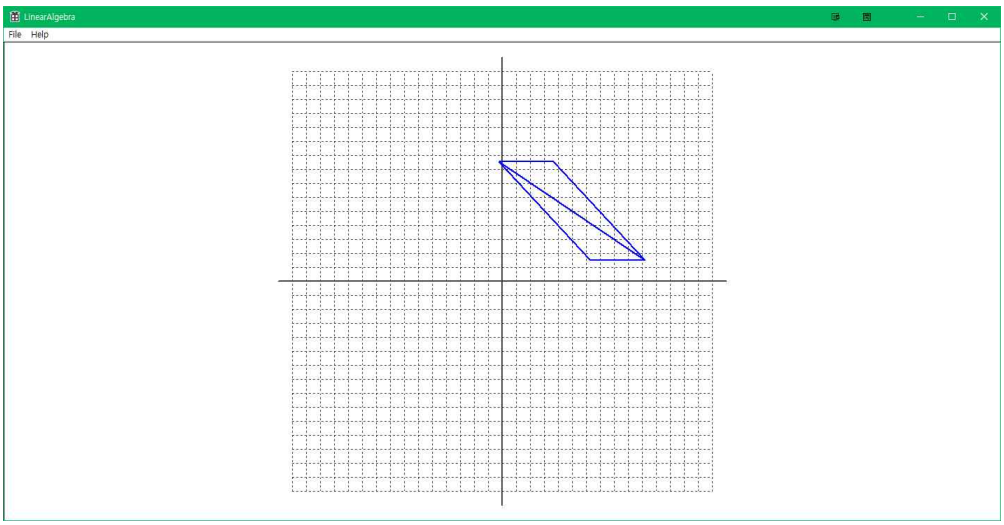


그림. 회전하는 와이어 프레임 사각형: x축으로 45도 회전된 사각형은 로컬 y축을 중심으로 계속해서 회전합니다. 사각형의 월드 상의 위치는 (100,100,0)입니다.

Step09: 프로젝션 변환

이제 원근 투영 매트릭스를 설정하는 멤버 함수를 구현할 차례입니다. 실세계(real world)에서 같은 크기라도 멀리 있는 물체는 더 작게 보입니다. 거리가 멀어짐에 따라 물체는 점점 작아져서 결국은 한 점으로 수렴하게 되는데 이 점을 **소실점(vanishing point)**이라고 합니다. 3차원 개체를 렌더링할 때 이와 같은 깊이값을 고려해서 렌더링하는 것이 (대부분의 경우) 바람직한데, 이

렇게 되도록 변환 매트릭스를 설정할 수 있는데 이 매트릭스를 **원근 투영 행렬(Perspective Projection Matrix)**라고 합니다. 구현을 간단하게 하기 위해 투영된 개체들이 렌더링 되는 평면을 항상 xy평면이라 가정하고, 아래 그림을 봅시다.

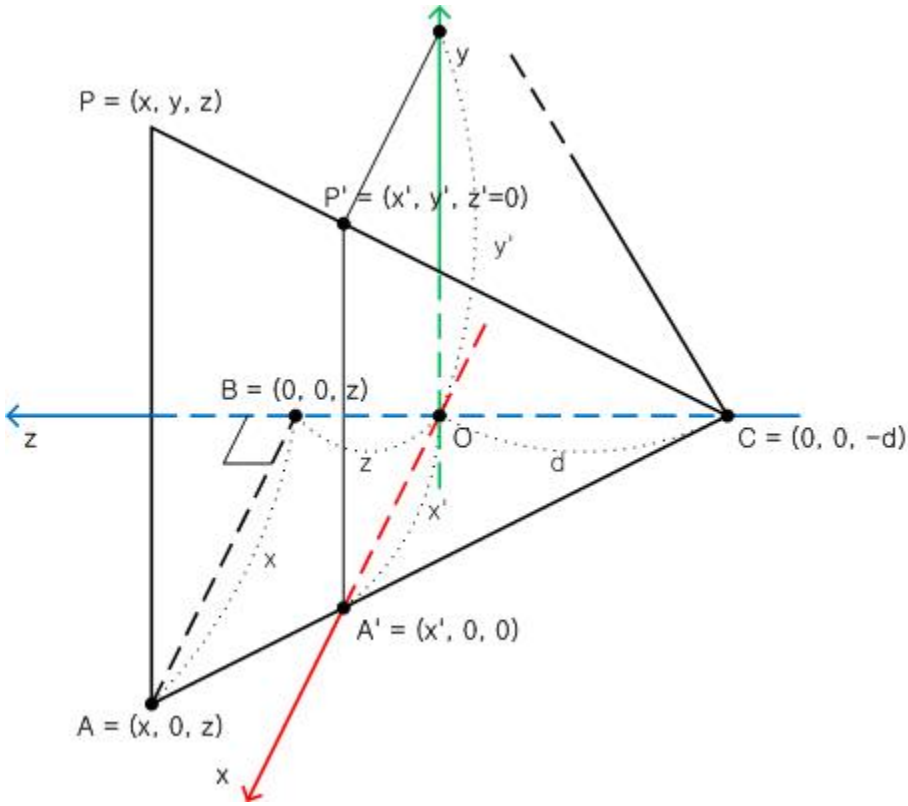


그림. 프로젝션 매트릭스의 설정:

눈(Eye)의 방향은 +z방향에 있고, 소실점을 C=(0,0,-d)라고 가정합니다. 점 P=(x,y,z)는 xy평면의 P'위치에 표시됩니다. 이제 우리의 목적은 소실점 C와 P가 주어졌을 때, P'을 구하는 것입니다. 이것은 삼각형 ABC와 삼각형 A'OC의 닮음 관계에 의해 유도할 수 있습니다. 높이와 밑변의 길이의 비율이 같으므로, 비례식은 다음과 같습니다.

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

x'에 대해 전개하면 다음과 같이 정리됩니다.

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

비슷한 방법으로, 아래의 식을 유도할 수 있습니다.

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

모든 식에 d/(z+d)가 있으므로, 호모지니어스 나눗셈의 분모가 $(z+d)/d$되게 선형 시스템을 구성해야 합니다. 그러면 w를 다음과 같이 구성해야 합니다.

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

그러면 (x', y', z', w)의 선형 시스템은 다음과 같습니다.

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

3차원 변환에서, 위의 선형 시스템의 결과 w==1이 되어야 하므로, 호모지니어스 나눗셈의 분모가 $(z+d)/d$가 되는 것입니다.

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

이제 아래와 같은 변환 행렬을 유도할 수 있습니다.

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

DirectX의 프로젝션 매트릭스 설정

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

그림. DirectX의 투영 행렬 설정에 관한 문서를 보면, 위 행렬이 처음 등장합니다. DirectX는 위치에 대해서 행벡터(Row Vector)를 사용하므로, 벡터와 행렬 곱셉에서 행렬의 왼쪽에 위치해야 합니다. 그러므로 DirectX는 행렬의 각 행이 베이시스를 의미합니다.

(아래 링크에서 DirectX의 프로젝션 매트릭스 설정에 대한 설명)
https://docs.microsoft.com/en-us/windows/desktop/direct3d9/projection-transform

이제 위치값 (x,y,z)를 프로젝션시키기 위해서는 (x,y,z,1)을 준비합니다. 그리고 아래의 투영행렬과 곱셉하여 투영 변환을 적용합니다.

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

매트릭스와 벡터의 곱의 결과 w값이 1이 아닌, (z+d)/d이므로 x', y', z'을 구하기 위해서는 매트릭스와 벡터의 결과로 구해진 벡터의 각 요소를 (z+d)/d로 나누어 주어야 합니다. 이것을 호모지니어서 나눗셈이라고 합니다.

(z+d)/d값 즉 w값은 포인트의 깊이 정보를 가지므로, DirectX나 OpenGl같은 그래픽 라이브러리는 이 값을 **z-버퍼(z-buffer)**라는 곳에 저장하고, 보이지 않는 부분을 제거하기 위해 이 정보를 유용하게 사용합니다(이 경우 z값을 저장하는 것이 아니라, w값을 저장하므로 **w-버퍼(w-buffer)**라고 합니다).

이제 프로젝션 매트릭스를 설정하는 멤버 함수를 다음과 같이 작성할 수 있습니다.

```
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
```

깊이감을 느끼기 위해 사각형보다는 육면체(cube)를 회전시켜 보도록 합시다. 육면체는 표준 좌표계 상에서 아래 그림과 같이 정의하였습니다.

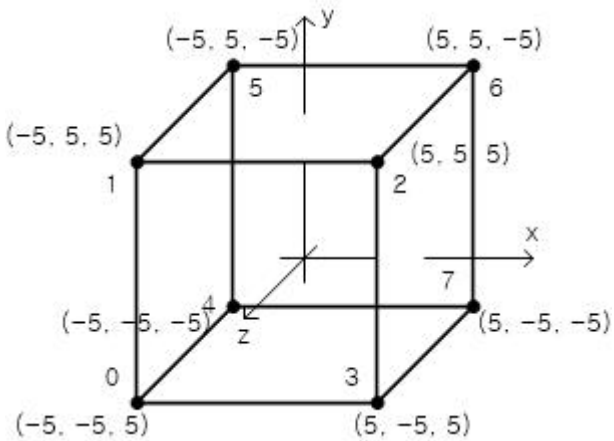


그림. 육면체(cube)의 구성

버텍스는 모두 8개이지만, 육면체의 테두리를 그리기 위해서 12개의 에지(edge)를 그려야 합니다. 보이지 않는 면을 렌더링하지 않는 것은 일단 무시합니다(실습문제 12). 인덱스 버퍼는 다음과 같습니다.

```
    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
    };
```

수정된 KPolygon의 구현 파일은 다음과 같습니다.

```
#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];

        //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()


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()
```

DrawIndexedPrimitive()함수의 내부에서 벡터의 Dot()연산과 Cross()연산이 사용된 부분을 주석처리하고 프로그램을 실행하면 결과는 다음과 같습니다.

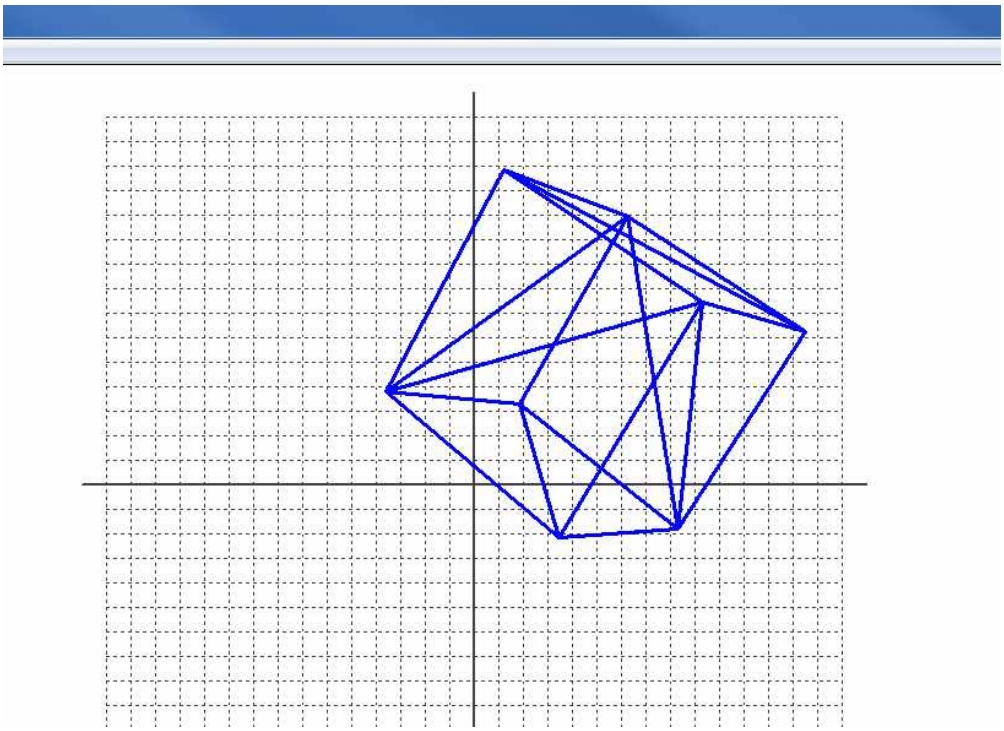


그림. 은면(Hidden Surface)제거없이 렌더링하는 육면체

육면체는 렌더링되지만, 보이지 않는 면을 제거하지 않아서 직관적이지 않습니다. 보이지 않는 면을 제거하는 기술을 **컬링(Culling)**이라고 합니다. 컬링에는 보이지 않는 면 제거(Hidden Surface Culling), 다른 물체에 의해서 가려진 면 제거(Occlusion Culling), 화면의 범위를 벗어나는 물체의 컬링(Viewport Culling)등이 있습니다.

보이지 않는 면을 제거하기 위해서는 추가적인 벡터 연산을 이해할 필요가 있습니다.

벡터의 추가적인 연산

벡터의 곱은 결과가 스칼라가 되는 **내적(dot product)**과 결과가 벡터가 되는 **외적(cross product)**이 있습니다.

내적(Dot Product)

u와 v가 이루는 각이 θ일때, 벡터 u와 v의 내적은 점(·) 연산자를 사용하여 u·v로 나타내며 다음과 같이 정의합니다.

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

3D 게임에서 내적은 두 벡터가 이루는 각을 구하기 위해서 많이 사용합니다. u = (u_1, u_2, u_3), v = (v_1, v_2, v_3)라 하고 u의 끝점을 P, v의 끝점을 Q라 합시다. 그리고 u,v가 이루는 각을 θ라 합시다. 그러면 코사인 법칙(law of cosines)에 의해 다음의 식을 유도할 수 있습니다.

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

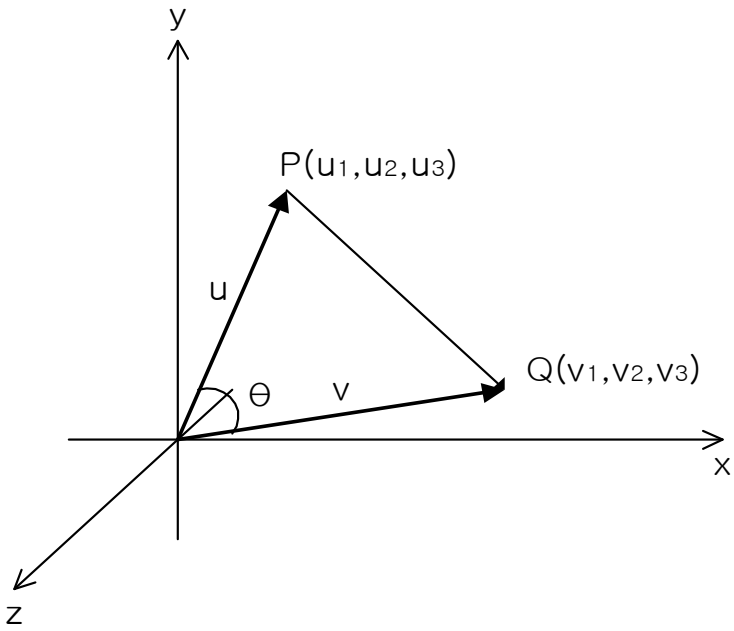


그림. 코사인 법칙: $|\overrightarrow{PQ}|^2 = |u|^2 + |v|^2 - 2|u||v|\cos\theta$가 성립합니다.

위 식을 전개하면 다음과 같습니다.

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

이제 식을 간단히 하면 다음과 같은 결과를 얻습니다.

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

$|u||v|\cos\theta$에 대해서 정리합니다. 그러면 다음과 같은 식을 얻습니다.

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

$|u||v|\cos\theta$는 어떤 의미일까요?

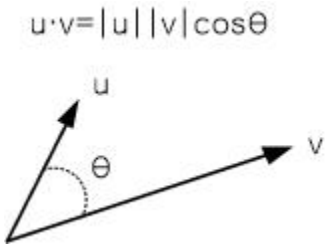


그림. 벡터의 내적

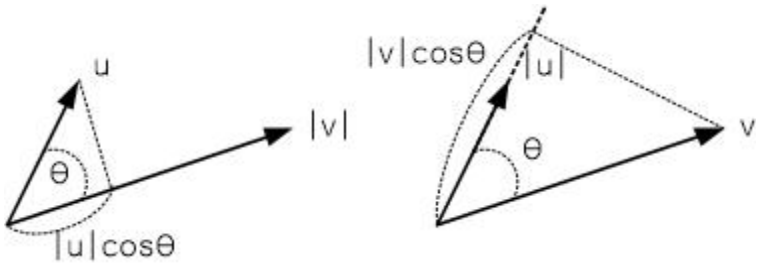


그림. 두 벡터의 내적: 벡터의 내적은 한 벡터를 대상 벡터에 직교 투영했을 때의 길이와 대상 벡터의 길이를 곱한 값입니다.

벡터 u와 벡터 v의 내적 u・v는 벡터 u를 벡터 v에 투영했을 때의 길이 $|u|\cos\theta$와 v의 길이 |v|를 곱한 값입니다. 이러한 내적은 한 벡터를 다른 벡터에 투영했을 때의 벡터를 구하거나, 두 벡터의 각을 구하기 위해서 사용할 수 있습니다. 또한 쿼터니언(Quaternion)등의 계산의 내부 연산으로 사용하기도 합니다.

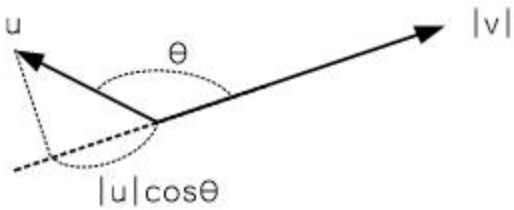


그림. 내적이 음수(negative number)인 경우: 내적이 음수라면 두 벡터는 $\pi/2$(90도)보다 큰 각으로 벌어져 있음을 나타냅니다.

벡터의 곱셈중의 하나인 내적(Inner Product, Dot Product) u·v는 다음과 같이 정의합니다.

$$u \cdot v = u_1 v_1 + u_2 v_2 + u_3 v_3 = |u||v|\cos\theta$$

이제 내적을 이용하면 두 벡터가 이루는 각을 구할 수 있습니다.

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

u·v = 0이면, 두 벡터는 서로 직각(perpendicular, orthogonal)입니다(실습문제1). 2차원 공간에서 n = (a,b)인 경우, 직선 ax + by + c = 0과 n이 서로 직교함을 증명할 수 있는데(실습문제 2), 이것은 3차원 공간으로 확대되어 n = (a, b, c)에 대해서, ax + by + cz + d = 0 평면과 (a, b, c) 벡터가 서로 직교함을 증명할 수 있습니다. 이렇게 면(plane)에 수직인 벡터를 **노멀(normal) 벡터(법선 벡터)**라고 하는데, 면의 방향을 결정하는 등 많은 곳에서 normal 벡터는 자주 사용됩니다.

행렬식(Determinant)

(동영상에서 행렬식의 기하학적인 의미를 설명할 것)

2×2행렬이 주어졌을 때, 이 변환이 기존의 면적이 1인 영역의 크기를 얼마만큼 변하게 하는지 구할 수 있습니다. 다음과 같은 행렬을 고려해 봅시다.

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

위 행렬은 (2,1)과 (-1,1)을 새로운 베이시스로 만듭니다. 그러면 표준 베이시스의 면적 1×1은 어떠한 크기로 변환 될까요? 그것은 아래 그림에서 빗금친 사각형의 면적을 구하는 것입니다.

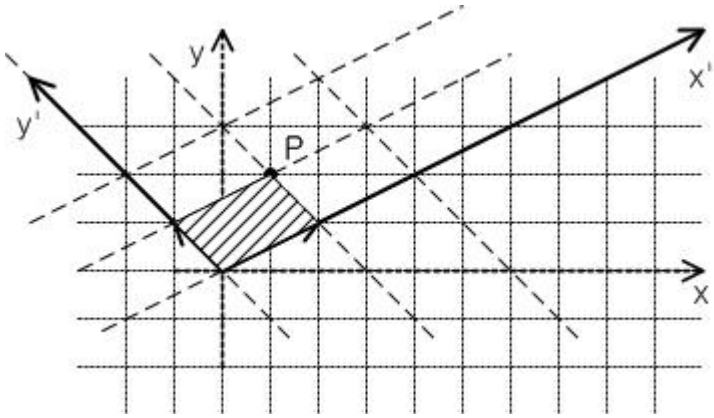


그림. 변환된 베이시스에 대해서 베이시스가 이루는 사각형의 단위 면적을 구할 수 있습니다. 이것을 행렬식이라고 합니다.

이제 다음 행렬 $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$에 대해 변환된 베이시스가 이루는 단위 면적을 다음과 같이 구할 수 있습니다. 행렬식은 ad-bc입니다.

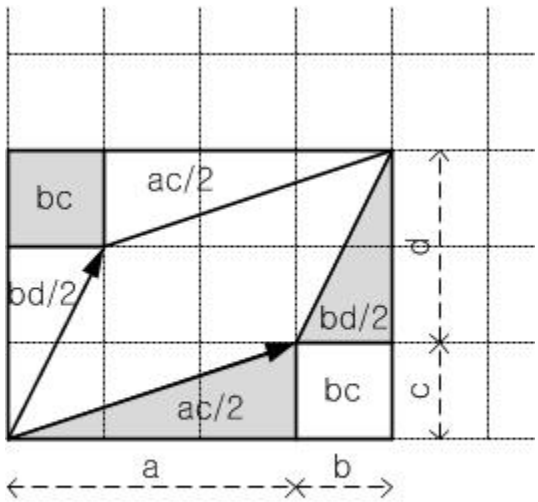


그림. 행렬식의 계산: (a,c)와 (b,d)를 베이시스로 하는 좌표계에서 단위 면적을 구할 수 있습니다

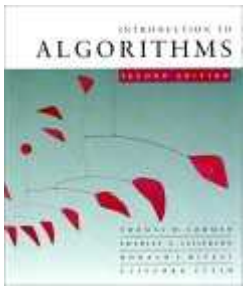
3차원 행렬에서 행렬식은 어떤 의미일까요? 그것은 변환된 베이시스가 이루는 3차원 공간에서 단위 부피를 의미합니다.

determinant for 2×2 matrix

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

Fig. determinant for 2×2 matrix

ex) Cormen, Introduction to Algorithm



Point in Polygon

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

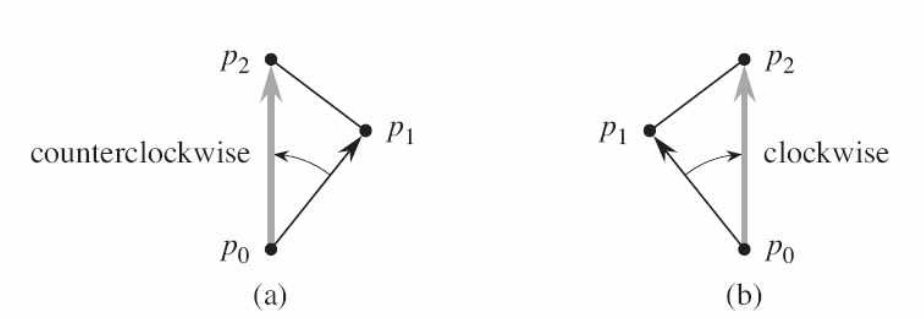
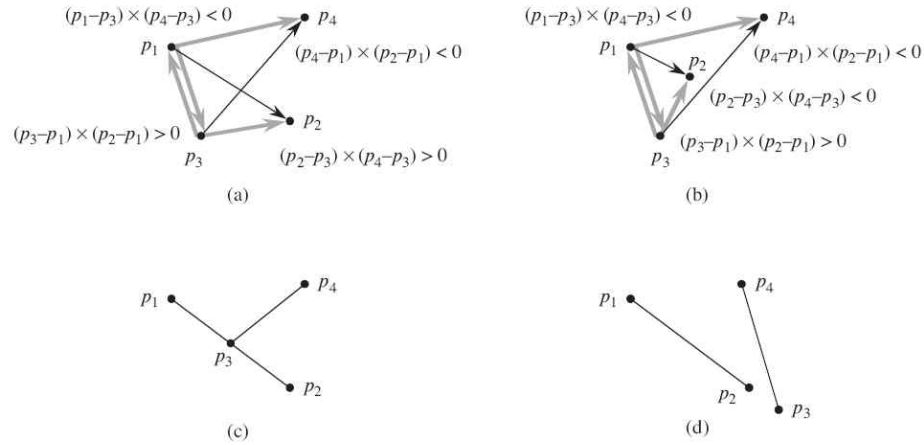


Figure 33.2 Using the cross product to determine how consecutive line segments p0p1 and p1p2 turn at point p1. We check whether the directed segment $\overrightarrow{p0p2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p0p1}$. (a) If counterclockwise, the points make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

Two Line Segment Intersection




```
SEGMENTS-INTERSECT (p1, p2, p3, p4)
1 d1 = DIRECTION (p3, p4, p1)
2 d2 = DIRECTION (p3, p4, p2)
3 d3 = DIRECTION (p1, p2, p3)
4 d4 = DIRECTION (p1, p2, p4)
5 if ((d1 > 0 && d2 < 0 ) || (d1 < 0 && d2 > 0 )) and
```

```
    ((d3 > 0 && d4 < 0 ) || (d3 < 0 && d4 > 0 ))
6    return TRUE
7 elseif d1 == 0 and ON-SEGMENT (p3, p4, p1)
8    return TRUE
9 elseif d2 == 0 and ON-SEGMENT (p3, p4, p2)
10    return TRUE
11 elseif d3 == 0 and ON-SEGMENT (p1, p2, p3)
12    return TRUE
13 elseif d4 == 0 and ON-SEGMENT (p1, p2, p4)
14    return TRUE
15 else return FALSE
```

DIRECTION $(\overrightarrow{p_i,p_j,p_k})$

1 return $(p_k - p_i) \times (p_j - p_i)$

ON-SEGMENT $(\overrightarrow{p_i,p_j,p_k})$

1 if $\min(x_i,x_j) \le x_k \le \max(x_i,x_j)$ and $\min(y_i,y_j) \le y_k \le \max(y_i,y_j)$

2 return TRUE

3 else return FALSE

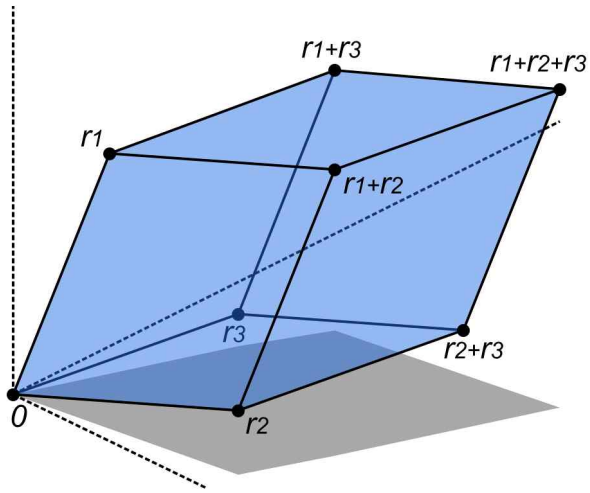


그림. 3×3행렬의 행렬식은 베이시스 단위가 이루는 체적의 부피입니다.

(행렬식의 계산)

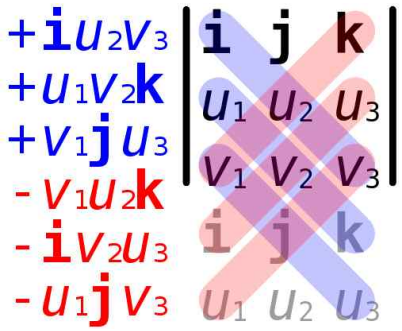


그림. 외적을 찾기 위한 소러스의 규칙(Sarrus Rule)

외적(Cross Product)

u = (u_1, u_2, u_3), v = (v_1, v_2, v_3)가 3차원 공간상의 벡터일 때, **외적(cross product) u×v**는 다음과 같이 정의합니다.

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

사실 외적은 모든 차원에 대해 **행렬식(determinant)**에 의해 형식적으로 정의될 수 있는데, 위의 경우는 3차원의 행렬식을 전개한 것입니다(실습문제 6). 내적의 결과는 스칼라이지만, 3차원 벡터의 외적의 결과는 벡터임을 주의하세요. 외적 u×v의 결과가 의미하는 벡터는 두 벡터 u,v에 모두 수직인 벡터이고, 그 크기는 다음과 같습니다(실습문제 7).

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

오른손 좌표계(RHS, right-hand coordinate system)를 쓰는 경우, 외적의 결과 벡터의 방향은 **오른손 규칙(right-hand rule)**에 의해 구할 수 있습니다.

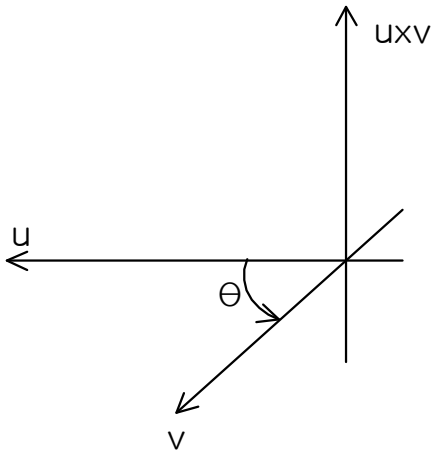


그림. 오른손법칙: 외적 벡터의 방향은 오른손을 오무렸을 때 엄지가 가리키는 방향입니다. 오른손을 폈을 때 손끝이 가리키는 방향이 u이고, 오므렸을 때 손끝이 가리키는 방향이 v라면, 엄지의 방향이 외적 벡터의 방향이 됩니다. 왼손 좌표계의 경우는 외적의 방향이 반대가 되므로 주의하세요.

외적 역시 2D/3D 프로그래밍에서 빈번하게 사용합니다. 면의 방향을 결정하기 위해서, 점이 직선이나 평면의 어느 쪽에 놓여있는지를 검사하기 위해서 외적을 사용할 수 있습니다.

내적과 외적의 사용 예를 한마디로 정의할 수는 없지만, 다음과 같이 빈번하게 사용되는 예가 있습니다.

"**내적은 두 벡터의 각을 구하기 위해서, 외적은 면의 방향을 결정하기 위해서 사용합니다.**"

이제 KVector3에 대해서 내적을 구하는 Dot()함수와 외적을 구하는 Cross() 함수를 아래와 같이 추가할 수 있습니다.

```
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;
}
```

보이지 않는 면을 판단하는 방법

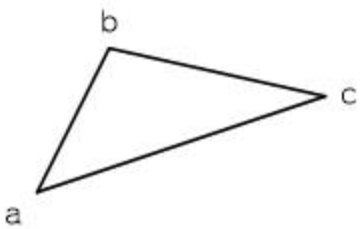


그림. 메시를 구성하는 한 삼각형 면이 카메라 시점에서 보이는지 안 보이는지를 판단하려고 합니다.

메시의 표면이 모두 삼각형으로 구성되었을 때, 메시의 한 삼각형 면이 현재 카메라 시점에서 보이지 않는다면 화면에 그리지 않아야 합니다. 이것을 판단하기 위해 먼저 면의 방향을 결정해야 하는데, 면의 방향은 벡터의 외적을 이용하면 구할 수 있습니다.

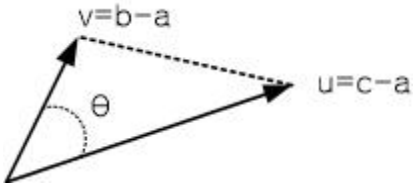


그림. 삼각형의 방향을 결정하기 위해, 삼각형을 구성하는 두 벡터를 구합니다.

삼각형 abc의 면의 방향은 면과 직각을 이루는 벡터로 표현할 수 있으며 이 벡터를 **법선 벡터(Normal Vector)**라고 합니다. 이 벡터를 구하기 위해 u=c-a, v=b-a로 두면, 두 벡터의 외적 u×v를 구하면 삼각형의 법선벡터가 됩니다.

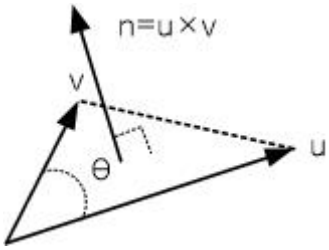


그림. u×v를 구하면 삼각형의 법선 벡터가 됩니다.

이제 이러한 방식으로 메시를 구성하는 모든 삼각형의 법선 벡터를 구할 수 있습니다. 이제 현 상태의 메시에 대해, 카메라를 향하는 방향벡터 Eye가 주어졌다고 합시다.

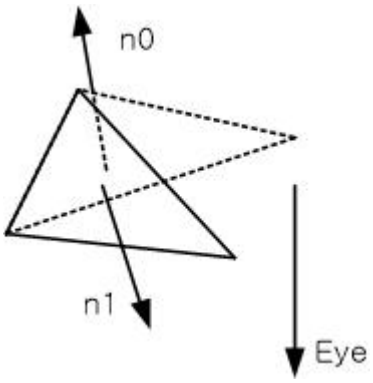


그림. 메시를 구성하는 모든 삼각형의 법선 벡터와 메시에서 카메라를 향하는 방향벡터 Eye가 주어졌다고 가정합시다.

이제 각 법선벡터와 Eye벡터의 내적을 취하면 삼각형이 현 시점에서 보이는지 어떤지를 판단할 수 있습니다.

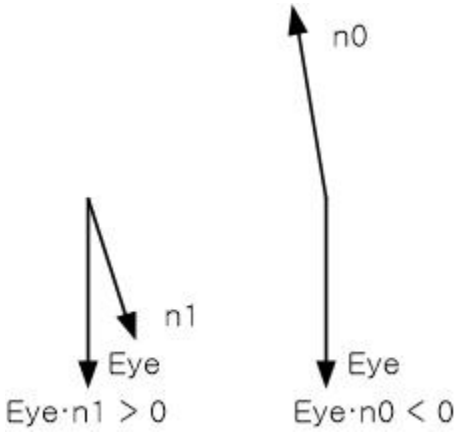


그림. 카메라를 향하는 벡터 Eye와 법선벡터의 내적이 0보다 크다면 그 면은 보이는 면이라는 뜻입니다.

법선벡터와 Eye 벡터의 내적을 구해서 그 값이 0보다 크다면 보이는 면입니다. 내적이 0보다 작다면 면이 시점의 반대 방향을 향한다는 의미이므로 보이지 않는 면이 됩니다.

일반적으로 그래픽 라이브러리에서는, 이 과정을 빠르게 하기 위해, 각 정점마다 법선 정보를 저장해 둡니다. 그리고 각 면의 법선은 정점의 법선을 보간해서 구하고, 시점 벡터 Eye와 내적을 구해서 그 값이 0보다 작다면 그 삼각형을 렌더링하지 않습니다.

보이지 않는 면을 제거하는 기능을 추가한 DrawIndexedPrimitive() 함수를 아래와 같이 구현할 수 있습니다.

```
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()
```

위 구현에서는 외적을 사용하여, 각 삼각형의 법선벡터를 구한 다음, forward 벡터와의 내적의 값이 0보다 커서, 그 면이 보이지 않는 면이라면 점선을 사용하여 삼각형을 렌더링하도록 하였습니다.

렌더링 파이프라인(rendering pipeline)

마지막으로 Render()함수를 작성하기 전에, 3D 장면이 렌더링되는 과정을 살펴봅시다. 이것은 최소한 다음의 3단계를 거칩니다.

1) 물체를 월드의 어디에 놓을 것인가를 결정합니다.
2) 놓여진 물체들 중 어디를 볼 것인가를 결정합니다.
3) 보이는 물체들을 어떻게 볼 것인가를 결정합니다.

위 단계를 각각 **월드 변환(world transform), 뷰잉 변환(viewing transform), 프로젝션 변환(projection transform)**이라고 합니다. 우리의 Render()함수에서는 카메라 시점의 방향이 (0,0,1)로 고정되었다고 가정하였습니다. 그래서 우리는 단지 월드 변환과 프로젝션 변환의 과정만을 고려할 것입니다.

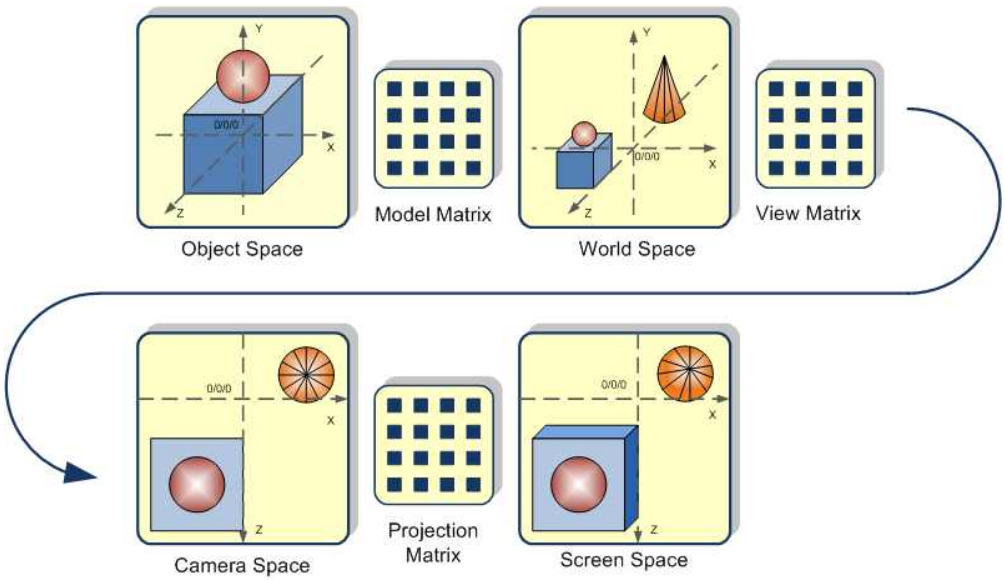


그림. 컴퓨터 그래픽스 변환 파이프라인: 모델 공간, 월드 공간, 카메라 공간을 거친후 프로젝션을 마치면 화면 공간에 렌더링됩니다.

이제 Render()함수는 다음과 같습니다.

```
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);
}
```

위 소스에서 아래 문장의 변환의 순서에 주의하세요.

matTransform = matTrans * matRotY * matRotX;

이것을 아래의 Transform()함수에 전달합니다.

```
poly.Transform( matTransform );
```

poly 객체의 변환의 순서는 다음과 같습니다.

1) x축에 대한 회전 변환
2) y축에 대한 회전 변환
3) 위치 변환

다음에 뷰잉 변환은 생략되었고, 프로젝션 변환을 거친 후 육면체가 렌더링 됩니다.

```
//poly.Viewing( matViewing );
poly.Transform( matProjection );
```

실행 결과는 아래와 같습니다. 프로젝션 매트릭스의 d값이 작다면 육면체의 외곡은 더 심해집니다.

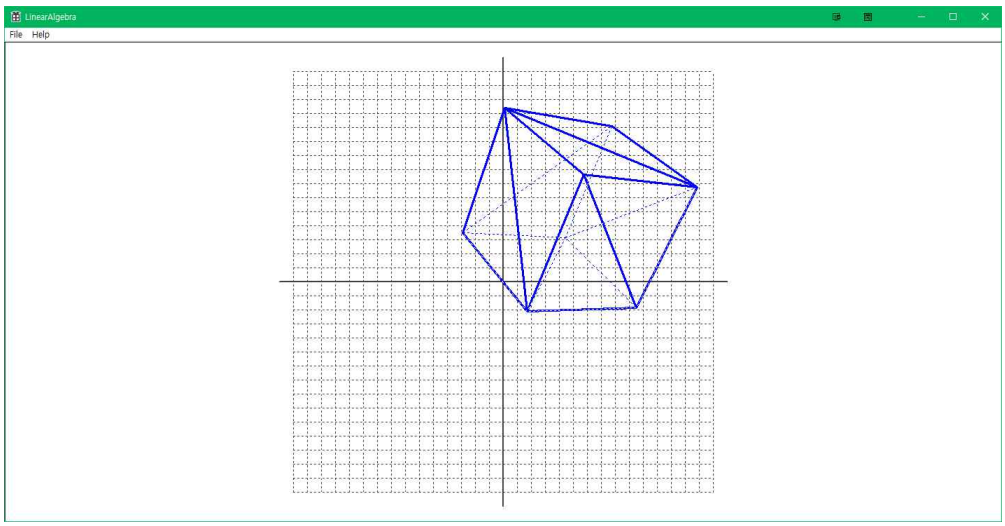


그림. 회전하는 육면체: 프로젝션된 육면체는 입체감이 나타난다.

이 장의 의도는 독자들이 3D 프로그래밍에 대한 전반적인 개념을 익히는 것입니다. 실제로 DirectX나 OpenGL을 사용하여 프로그래밍을 할 때 고려사항은 훨씬 많습니다.

실습문제

1. u·v = 0이면, 두 벡터는 왜 서로 직각입니까?

2. n = (a,b)와 직선 ax + by + c = 0이 서로 직교 함을 증명하세요(힌트: 직선상의 임의의 두 점이 이루는 벡터와 n의 내적이 0임을 보입니다).

3. (평면이나 공간상에서) 시점에서 멀리 떨어진 물체를 서서히 나타내게 하기 위해 거리 정보를 사용할 수 있습니다. 그렇게 하기 위해서는 시점방향의 거리에 대해 화면상에 표시될 물체들을 정렬하는 것이 필요한데, 이 때 내적을 이용할 수 있습니다. 구체적인 계산 예를 보이세요(힌트: 내적 a·b의 결과는 b에 대한 a의 투영(projection)입니다).

4. u = (a,b,c)를 normalize 하세요.

5. 평면 ax + by + cz + d = 0의 노멀 벡터 (a,b,c)가 노멀라이즈 되어 있을 때, 임의의 점 (v1, v2, v3)과 평면사이의 거리를 내적을 이용하여 구하세요.

6. 행렬식(determinant)에 대해 설명하세요. 외적은 행렬식에 의해 어떻게 정의할 수 있나요?

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

그림. 2×2행렬의 행렬식

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix} - b\begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix} + c\begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix}$$
$$= aei + bfg + cdh - ceg - bdi - afh.$$

그림. 3×3행렬의 행렬식

7. 외적의 크기가 |u×v| = |u||v|$\sin\theta$임을 증명하세요(힌트: 라그랑제의 항등원

(Lagrange's identity)).

$$\|\mathbf{a} \times \mathbf{b}\|^2 \equiv \det \begin{bmatrix} \mathbf{a} \cdot \mathbf{a} & \mathbf{a} \cdot \mathbf{b} \\ \mathbf{a} \cdot \mathbf{b} & \mathbf{b} \cdot \mathbf{b} \end{bmatrix} \equiv \|\mathbf{a}\|^2 \|\mathbf{b}\|^2 - (\mathbf{a} \cdot \mathbf{b})^2.$$

그림. 라그랑제 항등식

8. 올소노멀 베이시스(orthonormal basis)를 좌표계의 축으로 사용할 때의 이점은 무엇인가요?

9. 3차원 벡터 (x,y,z,1)가 변환된 (x',y',z',w)에서 w값이 어떻게 유용하게 이용될 수 있을까요?(힌트: 프로젝션 매트릭스)

10. 3차원 투영 변환 매트릭스를 유도하는 과정을 설명하세요.

11. 사이드 이펙트가 있는 연산자 함수가 레퍼런스를 리턴할 때의 이점은 무엇인가요?

12. 육면체의 보이지 않는 면을 렌더링하지 않도록 코드를 수정하세요(힌트: 내적과 외적)

13. KPolygon 클래스를 재사용 가능하도록 디자인하기 위해서는 Render() 멤버를 분리하는 것이 좋습니다. Render()를 분리하여 KPolygon클래스를 재사용 가능하도록 수정하세요.

[문서의 끝]