オブジェクト指向と ゲームプログラミング

▶ DirectX Graphics 3D編 ― 第５回　カメラ ◀

カメラ

3DCGでは、ワールド空間にオブジェクトを配置してシーンを作成します。次に行うのは、ワールド空間のどこからどの方向をどのように見るのか、という「視点」の作成です。

視点は、ワールド空間内の任意の位置から任意の方向をとらえることができます。DirectX Graphicsでは、視点のことをカメラまたはビューと呼びます。カメラの設定は、行列で行います。カメラの座標と方向を表す行列と、どのように見えるかを表す行列を作成し、デバイスに適用します。その後、レンダリングを行うと、カメラがとらえたワールド空間内の画像が出力されるというわけです。

ビュー変換行列

カメラから見る画像は、「ワールド座標に配置された3Dオブジェクトの頂点」を「カメラを基準とした座標」に変換したものといえます。このカメラを基準とした座標をビュー座標系と呼びます。ワールド座標系からビュー座標系への変換は、ビュー変換行列と呼ばれる行列を用います。

3Dオブジェクトの頂点にワールド変換行列が掛けられることにより、ワールド座標系への変換が行われますが、これにビュー変換行列を掛けると、カメラから見た世界、つまりカメラを原点としたビュー座標系へと変換されます。

ビュー変換行列は、カメラのｘ軸を示すベクトルをｒ、カメラのｙ軸を示すベクトルをｕ、カメラのｚ軸を示すベクトルをｖ、カメラの位置を示すベクトルをｐとしたとき、以下のような行列になります。

$$\begin{pmatrix} r_x & u_x & v_x & 0 \\ r_y & u_y & v_y & 0 \\ r_z & u_z & v_z & 0 \\ -(R\cdot p) & -(U\cdot p) & -(V\cdot p) & 1 \end{pmatrix}$$

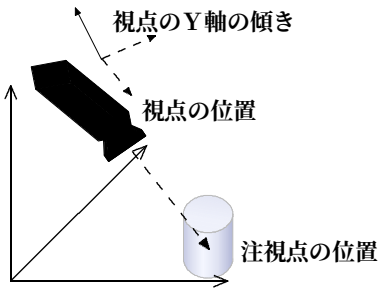
$R=(r_x, u_x, v_x)$…カメラの右方向ベクトル
$U=(r_y, u_y, v_y)$…カメラの上方向ベクトル
$V=(r_z, u_z, v_z)$…カメラの視点方向ベクトル

ビュー変換行列は、カメラの位置と方向を定義したものといえます。

LookAt方式

カメラを制御する場合、「視点の位置」「視点の方向」の２つの要素を用いますが、これは、ハンディカメラを扱うような感覚でカメラを制御することになります。この方式のほかに、テレビカメラのように、ある位置からある位置をとらえるLookAtという方式があります。

LookAt方式とは、直感的にカメラを扱うために考え出された方式で、DirectX Graphics以外の3Dライブラリでもよく使われています。LookAt方式では、カメラの制御に「視点の位置」「注視点の位置」「Ｙ軸の傾きを制御するベクトル」の３つの要素を用います。



LookAt方式でビュー変換行列を作成するには、まず注視点の位置から視点の位置を引き、正規化します。これは、カメラの視点方向(ｚ軸)を示すベクトル(ｖ)となります。次に、このベクトルとＹ軸の傾きを制御するベクトルの外積を求めます。すると、カメラの右方向(ｘ軸)を示すベクトル(ｒ)となります。さらに、ｒとｖの外積をとると、カメラの上方向(ｙ軸)を示すベクトルをｕが求まります。これで、カメラの(直交する)３つの軸が求まります。あとは、前述のビュー変換行列に代入し、位置と方向ベクトルの内積を計算するだけです。

Direct3DXは、LookAt方式でビュー変換行列を作成するD3DXMatrixLookAtLH関数を提供していますが、LookAt方式以外でビュー変換行列を作成する関数は提供していません。

```
// ビュー変換行列作成([0.0, 1.0, -5.0]から[0.0, 0.0, 0.0]を注視するカメラ)
D3DXMATRIX    view
D3DXMatrixLookAtLH(&view, &D3DXVECTOR3(0.0f, 1.0f, -5.0f),
                   &D3DXVECTOR3(0.0f, 0.0f, 0.0f), &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
```

ビュー変換行列の設定

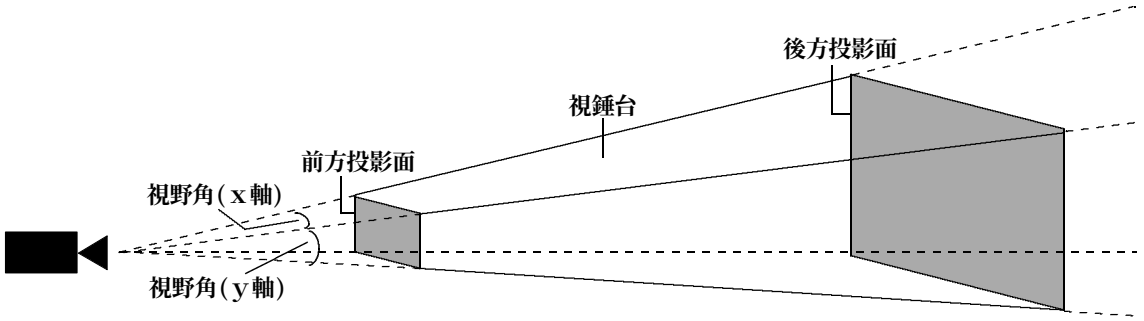
ビュー変換行列が作成できたら、IDirect3DDevice9::SetTransformメソッドを呼び出してシステムに設定します。以後、すべてのオブジェクトのレンダリング時に、設定したビュー変換行列が適用されます。

```
// ビュー変換行列設定(pD3DDeviceは初期化済みのDirect3DDevice9オブジェクト)
pD3DDevice->SetTransform(D3DTS_VIEW, &view);
```

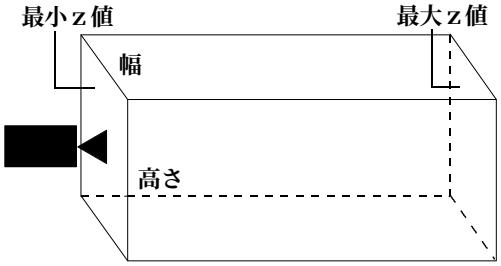
プロジェクション変換

ビュー変換の次に行うのが、「カメラでどのようにとらえるか」という変換です。これをプロジェクション変換(射影変換または投影変換)と呼びます。これは、カメラのレンズを設定する作業といえます。プロジェクション変換には、おもに透視投影(パースペクティブ射影：Perspective Projection)と正投影(正射影：Orthogonal Projection)が用いられます。

透視投影とは、「パースペクティブ：遠近」という言葉どおり、遠くのものが小さく見えるという見え方のことです。
透視投影では、投影面という概念を導入します。投影面とは、スクリーンのように世界を写し取る平面のことです。この投影面を２つ導入します。カメラに近いところと遠いところ、それぞれ前方投影面と後方投影面と呼びます。前方投影面より近い範囲は見えない、後方投影面より遠い範囲も見えない、というようにすると、見える範囲がピラミッドの上部を切り取ったような形に区切られることになります。この形を視錘台と呼びます。視錘台から完全にはずれているオブジェクトは描画されません。このようなオブジェクトのレンダリング処理を省くことにより、描画速度を向上させることができます。
次に、視野角(Filed Of View)という概念を導入します。これは、見える範囲の角度のことで、視線の中心からどの角度までを描画するかを指定するものです。視野角を小さくすると被写体はズームインされ、大きくすると被写体はズームアウトされます。

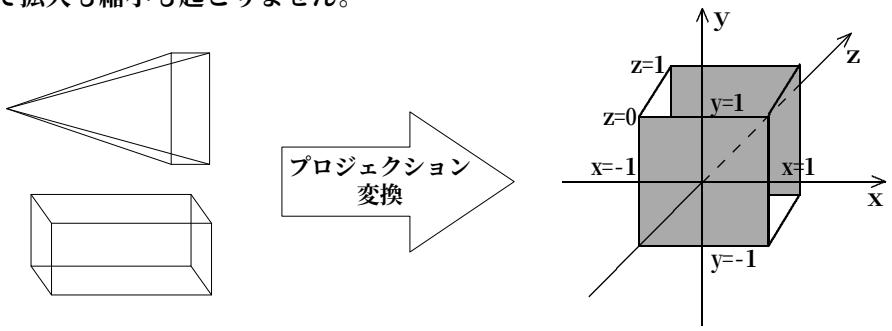


正投影とは、オブジェクトを平面的に見るという方法です。遠近感がないので、遠くにあるものが小さく表示されたりしません。正投影は、CADのような図形を正確に表示しなければならない分野で使われています。



プロジェクション変換では、カメラがとらえた空間(ビューボリューム)を「Ｘ＝－１～＋１」「Ｙ＝－１～＋１」「Ｚ＝０～＋１」の領域に収まるように変換します。この空間を射影空間と呼びます。

透視投影では、ビューボリュームがカメラの原点を頂点とした錐になっているため、遠くのものほど縮小されます。そのため、遠いものは小さくなり、遠近感がでます。正投影では、ビューボリュームが立方体なので拡大も縮小も起こりません。



プロジェクション変換行列

ビュー座標系に変換された頂点の座標を射影空間の座標(射影座標系)に変換するには、プロジェクション変換行列と呼ばれる変換行列を用います。ビュー座標系の頂点にプロジェクション変換行列を掛けることにより、カメラを基準とした座標から、実際にカメラから見た座標へ変換されます。

プロジェクション変換行列は、投影方法により異なる行列となります。通常、パースペクティブ射影行列の場合は、以下のように視野をもとに作成します。

$$\begin{pmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & zf/(zf-zn) & 1 \\ 0 & 0 & -zn*zf/(zf-zn) & 0 \end{pmatrix}$$

※h = cot(Y軸の視野角 / 2)
w = h / 投影面の横÷縦[＝アスペクト比]
zn = 視点から前方投影面までの距離
zf = 視点から後方投影面までの距離

正射影行列の場合は、以下のように作成します。

$$\begin{pmatrix} 2/w & 0 & 0 & 0 \\ 0 & 2/h & 0 & 0 \\ 0 & 0 & 1/(zf-zn) & 0 \\ 0 & 0 & zn(zn-zf) & 1 \end{pmatrix}$$

※h = ビューボリュームの高さ
w = ビューボリュームの幅
zn = ビューボリュームの最小ｚ値
zf = ビューボリュームの最大ｚ値

Direct3DXは、視野をもとにパースペクティブ射影行列を作成するD3DXMatrixPerspectiveFovLH関数や正射影行列を作成するD3DXMatrixOrthoLH関数など、いくつかの射影行列作成関数を提供しています。

```
// パースペクティブ射影行列の作成
D3DXMATRIX    projection;
::D3DXMatrixPerspectiveFovLH(&projection, ::D3DXToRadian(45.0f), 4.0f / 3.0f,
                             0.1f, 1000.0f);
```

プロジェクション変換行列の設定

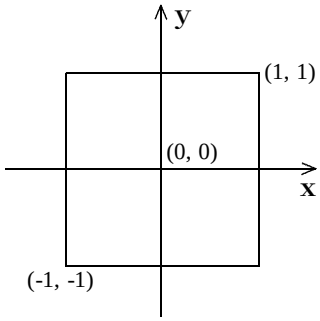
プロジェクション行列が生成できたら、IDirect3DDevice9::SetTransformメソッドを呼び出してシステムに設定します。以後、すべてのオブジェクトのレンダリング時に、設定したプロジェクション変換行列が適用されます。

```
// プロジェクション変換行列設定
pD3DDevice->SetTransform(D3DTS_PROJECTION, &projection);
```

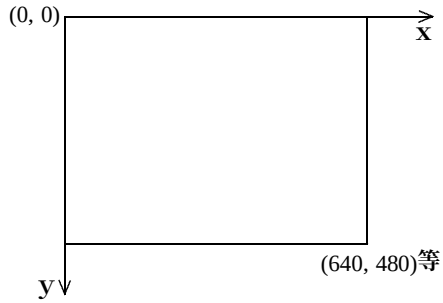
スクリーン座標系

3DCGを描画する際、最後に行われるのが射影座標系からスクリーン座標系への変換です。スクリーン座標系とは、画面のピクセルに１対１に対応する座標系のことです。射影座標系からスクリーン座標系への変換は、ビューポートスケーリング行列またはビューポート行列と呼ばれる変換行列が用いられます。

射影座標系とスクリーン座標系の違いは、中心の座標です。射影座標系では中央にあり、スクリーン座標系では左上にあるということです。また、ｙ軸の向きが逆になっていることも大きな違いです。



▲射影座標系



▲スクリーン座標系

ビューポートスケーリング行列は、画面の幅をWidth、画面の高さをHeightとしたとき、以下のようになります。

$$\begin{pmatrix} Width/2 & 0 & 0 & 0 \\ 0 & -Height/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Width/2 & Height/2 & 0 & 1 \end{pmatrix}$$

DirectX Graphicsでは、ビューポートスケーリング行列は、IDirect3DDevice9::SetViewportメソッドでビューポート(レンダリング結果を対象面のどこに描画するのか)の設定を行うことにより自動的に生成され、デバイスに適用されます。

最後に、モデル座標系からスクリーン座標系への変換をまとめると、

スクリーン座標 ＝ モデル座標 × ワールド行列 × ビュー行列 × 射影行列 × ビューポート行列

となります。

課　　題

CDXGraphics9クラスに、カメラを設定する機能を追加しましょう。

(１)カメラをカプセル化するクラスを作成します。カメラクラスのヘッダファイル(Camera.hpp)を以下のように作成しましょう。

－Camera.hpp－

```
/*
================================================================================================
                                オブジェクト指向ゲームプログラミング
    Programmed by Hibikino software. Copyright (c) 2005 Hibikino software. All rights reserved.
================================================================================================

  【対象ＯＳ】
     Microsoft Windows2000/XP

  【コンパイラ】
     Microsoft Visual C++ 2005

  【プログラム】
     Camera.hpp
                カメラクラスヘッダ

  【履歴】
     * Version    1.00       2005/04/dd  hh:mm:ss

================================================================================================
*/

#pragma once

/*************************************************************************************************/
/*                                インクルードファイル                                           */
```

```
/*****************************************************************************************************/
#include <d3dx9.h>

/*****************************************************************************************************/
/*                                          カメラクラス定義                                         */
/*****************************************************************************************************/
class CCamera {
public:

private:
};
```

(２)カメラクラスのソースファイル(Camera.cpp)を以下のように作成しましょう。

－Camera.cpp－

```
/*
=====================================================================================================
                                    オブジェクト指向ゲームプログラミング
     Programmed by Hibikino software. Copyright (c) 2005 Hibikino software. All rights reserved.
=====================================================================================================

  【対象ＯＳ】
     Microsoft Windows2000/XP

  【コンパイラ】
     Microsoft Visual C++ 2005

  【プログラム】
     Camera.cpp
                カメラクラス

  【履歴】
     * Version    1.00       2005/04/dd  hh:mm:ss

=====================================================================================================
*/

/*****************************************************************************************************/
/*                                          インクルードファイル                                     */
/*****************************************************************************************************/
#include "Camera.hpp"
#include <cassert>
```

(３)カメラクラスに、カメラを制御するのに必要なメンバを追加します。以下のプログラムを適切な場所に追加しましょう。

```
    IDirect3DDevice9*    m_pD3DDevice;

    D3DXVECTOR3          m_Position;
    D3DXQUATERNION       m_Rotation;

    D3DXMATRIX           m_View;
    D3DXMATRIX           m_Projection;
```

ビュー行列、プロジェクション行列をデバイスに設定する際に、Direct3DDevice9オブジェクトが必要になるので、メンバにしておきます。また、カメラの座標や角度、ビューおよびプロジェクション行列もメンバにしておきます。これらのメンバは、座標や角度をクラスの外に返したり、デバイスをリセットした際の復元処理に使用します。

カメラクラスでは、回転を制御するのにクォータニオンを用いています。クォータニオン(四元数)とは、ｘ，ｙ，ｚのベクトルと、ベクトルを軸とした回転角の４つの値で表現される数です。
クォータニオンを用いると、ジンバルロックという現象を回避することができます。ジンバルロックとは、ある軸の角度を９０度回転させたときに、別の軸の回転ができなくなる現象です。たとえば、フライトシミュレータで機首を真上したとき、ロール(ｚ軸)回転ができなくなるような現象です。
Direct3DXでは、クォータニオンを格納する構造体と、クォータニオンの計算を行う関数を提供しているので、とても簡単に扱うことができます。クォータニオンで回転を管理すると、ジンバルロックを回避できるだけでなく、回転の補間なども行うことができるようになります。

(４)カメラクラスのコンストラクタを作成します。以下のプログラムをソースファイルに追加しましょう(プロトタイプも忘れずに追加しましょう)。

```
/*****************************************************************************************************/
/*                                          コンストラクタ                                           */
/*****************************************************************************************************/
CCamera::CCamera(IDirect3DDevice9* pD3DDevice) : m_pD3DDevice(NULL)
{
    assert(pD3DDevice != NULL);

    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // 位置と角度の初期化
    m_Position = D3DXVECTOR3   (0.0f, 0.0f, 0.0f);
    m_Rotation = D3DXQUATERNION(0.0f, 0.0f, 0.0f, 1.0f);

    // ビュー行列の初期化
    ::D3DXMatrixIdentity(&m_View);
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &m_View);

    // プロジェクション行列の初期化
    ::D3DXMatrixIdentity(&m_Projection);
    m_pD3DDevice->SetTransform(D3DTS_PROJECTION, &m_Projection);
}
```

(５)カメラクラスのデストラクタを作成します。以下のプログラムを完成させ、ソースファイルに追加しましょう(プロトタイプも忘れずに追加しましょう)。

```
/*****************************************************************************************************/
/*                                          デストラクタ                                             */
/*****************************************************************************************************/
CCamera::~CCamera()
{
    m_pD3DDevice->???????();
}
```

(６)カメラの位置と角度からビュー変換行列を生成し、デバイスに設定するSetView関数を作成します。以下のプログラムを完成させ、適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                        ビュー変換行列設定                                         */
/*****************************************************************************************************/
void CCamera::SetView(const D3DXVECTOR3& inPosition, const D3DXVECTOR3& inRotation)
{
    m_Position = inPosition;

    // 角軸のクォータニオンを求める
    D3DXQUATERNION   rot_x, rot_y, rot_z;
    ::D3DXQuaternionRotationAxis(&rot_x, &D3DXVECTOR3(1.0f, 0.0f, 0.0f), D3DXToRadian( inRotation.x));
    ::D3DXQuaternionRotationAxis(&rot_y, &D3DXVECTOR3(0.0f, 1.0f, 0.0f), D3DXToRadian(-inRotation.y));
    ::D3DXQuaternionRotationAxis(&rot_z, &D3DXVECTOR3(0.0f, 0.0f, 1.0f), D3DXToRadian(-inRotation.z));

    // 方向設定
    m_Rotation = rot_x * rot_y * rot_z;
    ::D3DXMatrixRotationQuaternion(&m_View, &m_Rotation);

    // 位置ベクトル設定
    m_View._41 = -::D3DXVec3Dot(&D3DXVECTOR3(m_View._11, m_View._12, m_View._13), &m_Position);
    m_View._42 = -::D3DXVec3Dot(&D3DXVECTOR3(m_View._21, m_View._22, m_View._23), &m_Position);
    m_View._43 = -::D3DXVec3Dot(&D3DXVECTOR3(m_View._31, m_View._32, m_View._33), &m_Position);

    m_pD3DDevice->???????????(??????????, &m_View);
}
```

SetView関数の仕様は以下のとおりです。

SetView　設定

カメラの位置と角度からビュー変換行列を生成し、デバイスに設定します。

書式 void SetView(const D3DXVECTOR3& inPosition, const D3DXVECTOR3& inRotation);

inPosition　カメラの位置
inRotation　カメラの回転角

(７)LookAt方式でビュー変換行列を生成し、デバイスに設定するSetViewLookAt関数を作成します。以下のプログラムを完成させ、適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          ビュー変換行列設定                                         */
/*****************************************************************************************************/
void CCamera::SetViewLookAt(const D3DXVECTOR3& inEye, const D3DXVECTOR3& inAt, const float inRoll)
{
    // 上方ベクトル設定
    D3DXVECTOR3   up = D3DXVECTOR3(-::sinf(D3DXToRadian(inRoll)), ::cosf(D3DXToRadian(inRoll)), 0.0f);

    // ビュー行列設定
    ::??????????????????LH(&m_View, &inEye, &inAt, &up);
    m_pD3DDevice->※ここは各自考えましょう;

    m_Position = inEye;
    ::D3DXQuaternionRotationMatrix(&m_Rotation, &m_View);

}
```

SetViewLookAt関数の仕様は以下のとおりです。

SetViewLookAt　設定

LookAt方式でビュー変換行列を生成し、デバイスに設定します。

```
書void SetViewLookAt(const D3DXVECTOR3& inEye, const D3DXVECTOR3& inAt,
                     const float inRoll);
```

inEye	カメラの位置
inAt	カメラの注視座標
inRoll	カメラのｚ軸を回転軸とした回転角

(８)視野をもとにパースペクティブ射影行列を作成し、デバイスに設定するSetProjectionPerspective関数を作成します。以下のプログラムを完成させ、適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          遠近射影行列設定                                           */
/*****************************************************************************************************/
void CCamera::SetProjectionPerspective(const float inFovY, const float inWidth, const float inHeight,
                                       const float inNear, const float inFar)
{
    ::D3DXMatrix?????????????LH(&m_Projection, D3DXToRadian(inFovY), inWidth / inHeight, inZn, inZf);
    m_pD3DDevice->※ここは各自考えましょう;
}
```

SetProjection関数の仕様は次のとおりです。

SetProjection　設定

視野をもとにパースペクティブ射影行列を生成し、デバイスに設定します。

```
書void SetProjectionPerspective(const float inFovY,
                                const float inWidth, const float inHeight,
                                const float inNear,  const float inFar);
```

inFovY	視野の角度(ｙ軸)
inWidth	アスペクト比を求めるための視野の幅
inHeight	アスペクト比を求めるための視野の高さ
inNear	前方投影面
inFar	後方投影面

(９)正射影行列を作成し、デバイスに設定するSetProjectionOrtho関数を作成します。以下のプログラムを完成させ、適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          正射影行列設定                                             */
/*****************************************************************************************************/
```

```
void CCamera::SetProjectionOrtho(const float inWidth, const float inHeight,
                                 const float inZn, const float inZf)
{
    ::??????????????LH(&m_Projection, inWidth, inHeight, inZn, inZf);
    m_pD3DDevice->※ここは各自考えましょう;
}
```

SetProjectionOrtho関数の仕様は以下のとおりです。

SetProjectionOrtho　　設定

正射影行列を生成し、デバイスに設定します。

```
試void SetProjectionOrtho(const float inWidth, const float inHeight,
                          const float inNear,  const float inFar);
```

inWidth	視野の幅
inHeight	視野の高さ
inNear	視点から前方投影面までの距離
inFar	視点から後方投影面までの距離

(10)カメラの位置、向きを返す関数を作成します。以下のプログラムを適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          位置取得                                                 */
/*****************************************************************************************************/
D3DXVECTOR3 CCamera::GetPosition()
{
    return m_Position;
}

/*****************************************************************************************************/
/*                                          回転角取得                                               */
/*****************************************************************************************************/
D3DXVECTOR3 CCamera::GetRotation()
{
    D3DXVECTOR3   rotation;

    // 行列から軸を得る
    D3DXVECTOR3   y_axis(m_View._12, m_View._22, m_View._32);
    D3DXVECTOR3   z_axis(m_View._13, m_View._23, m_View._33);

    // y軸回転角
    rotation.y = ::atan2f(z_axis.x, z_axis.z);

    // x軸回転角
    const float  COS_Y = ::cosf(rotation.y);
    rotation.x = ::atan2f(COS_Y * z_axis.y, ::fabs(z_axis.z));

    // z軸回転角
    const float  SIN_Y = ::sinf(rotation.y);
    const float  COS_X = ::cosf(rotation.x);
    rotation.z = ::atan2f(COS_X * (SIN_Y * y_axis.z - COS_Y * y_axis.x), y_axis.y);

    // 度に変換
    rotation.x = D3DXToDegree(rotation.x);
    rotation.y = D3DXToDegree(rotation.y);
    rotation.z = D3DXToDegree(rotation.z);

    return rotation;
}
```

GetRotation関数が返す角度には誤差があります。また、求めた角度が複数解になる場合があり、最小のものを返すようになっています。そのため、x軸の角度に関しては、意図しない値が返される場合があります。なお、返す角度の範囲は-180度から+180度です。

(11)カメラを移動および回転させる関数を追加します。

カメラを現在向いている方向に移動させるMove関数、カメラを回転させるRotation関数、ある点を注視しながらその周りを回る(指定した座標を中心に公転する)LookAtRotation関数を作成します。

以下のプログラムを完成させ、適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                                移　動                                               */
/*****************************************************************************************************/
void CCamera::Move(const float inFront, const float inUp, const float inRight)
{
    // カメラの方向ベクトルを取り出す
    D3DXVECTOR3   right(m_View._11, m_View._12, m_View._13);
    D3DXVECTOR3   up   (※ここは各自考えましょう);
    D3DXVECTOR3   front(※ここは各自考えましょう);

    // 移動
    right      *= inRight;
    up         *= inUp;
    front      *= inFront;
    m_Position += front + up + right;
}

/*****************************************************************************************************/
/*                                                回　転                                               */
/*****************************************************************************************************/
void CCamera::Rotation(const float inX, const float inY, const float inZ)
{
    // 角軸のクォータニオンを求める
    D3DXQUATERNION   rot_x, rot_y, rot_z;
    ::D3DXQuaternionRotationAxis(&rot_x, &D3DXVECTOR3(1.0f, 0.0f, 0.0f), D3DXToRadian( inX));
    ::D3DXQuaternionRotationAxis(&rot_y, &D3DXVECTOR3(0.0f, 1.0f, 0.0f), D3DXToRadian(-inY));
    ::D3DXQuaternionRotationAxis(&rot_z, &D3DXVECTOR3(0.0f, 0.0f, 1.0f), D3DXToRadian(-inZ));

    // 回転
    m_Rotation *= rot_x * rot_y * rot_z;

    // ビュー行列の軸の設定
    ::D3DXMatrixRotationQuaternion(&m_View, &m_Rotation);
}

/*****************************************************************************************************/
/*                                                公　転                                               */
/*****************************************************************************************************/
void CCamera::LookAtRotation(const D3DXVECTOR3& inAt, const D3DXVECTOR2& inRotation)
{
    // 注視点を原点としたときの視点の座標を求め、ビュー行列をワールド行列にする
    m_View._41 = m_Position.x - inAt.x;
    m_View._42 = m_Position.y - inAt.y;
    m_View._43 = m_Position.z - inAt.z;

    // 角軸の回転行列を作成する
    D3DXMATRIX   rot_x, rot_y;
    ::D3DXMatrixRotationX(&rot_x, D3DXToRadian(inRotation.x));
    ::D3DXMatrixRotationY(&rot_y, D3DXToRadian(inRotation.y));

    // カメラを回転させる
    m_View *= rot_y * rot_x;

    // 視点を戻す
    m_Position = D3DXVECTOR3(m_View._41, m_View._42, m_View._43) + inAt;

    // クォータニオンの保存
    ::D3DXQuaternionRotationMatrix(&m_Rotation, &m_View);
}

/*****************************************************************************************************/
/*                                                更　新                                               */
/*****************************************************************************************************/
void CCamera::Update()
{
    // 位置ベクトルの設定
    m_View._41 = -::D3DXVec3Dot(&D3DXVECTOR3(m_View._11, m_View._12, m_View._13), &m_Position);
    m_View._42 = -::D3DXVec3Dot(&D3DXVECTOR3(m_View._21, m_View._22, m_View._23), &m_Position);
    m_View._43 = -::D3DXVec3Dot(&D3DXVECTOR3(m_View._31, m_View._32, m_View._33), &m_Position);

    m_pD3DDevice->SetTransform(※ここは各自考えましょう);
}
```

各関数の仕様は以下のとおりです。

Move 設定

カメラの向いている方向に進みます(画面に向かって奥、上、右に進みます)。ただし、カメラクラスの持つ座標を更新するだけで、デバイスには反映されません。

```
書式void Move(const float inFront, const float inUp, const float inRight);
```

inFront	奥行き方向への移動量
inUp	上方向への移動量
inRight	右方向への移動量

Rotation 設定

カメラを回転させます。ただし、カメラクラスの持つ角度を更新するだけで、デバイスには反映されません。

```
書式void Rotation(const float inX, const float inY, const float inZ);
```

inX	ｘ軸の回転量
inY	ｙ軸の回転量
inZ	ｚ軸の回転量

LookAtRotation 設定

現在の座標から指定された点を注視しながら、その周りを回ります。ただし、カメラクラスの持つ座標と角度を更新するだけで、デバイスには反映されません。

```
書式void LookAtRotation(const D3DXVECTOR3& inAt, const D3DXVECTOR2& inRotation);
```

inAt	凝視する座標(公転運動の中心座標)。
inRotation	ｘ軸(縦方向)とｙ軸(横方向)の回転量

Update 設定

Move, Rotation, LookAtRotation関数で更新された座標および角度をデバイスに適用します。

```
書式void Update();
```

※Update関数とSetView関数の「// 位置ベクトル設定」以降は、まったく同じプログラムです。よって、SetView関数を以下のように書き換えることができます。

```
void CCamera::SetView(const D3DXVECTOR3& inPosition, const D3DXVECTOR3& inRotation)
{
    m_Position = inPosition;

    // 角軸のクォータニオンを求める
    D3DXQUATERNION   rot_x, rot_y, rot_z;
    ::D3DXQuaternionRotationAxis(&rot_x, &D3DXVECTOR3(1.0f, 0.0f, 0.0f), D3DXToRadian( inRotation.x));
    ::D3DXQuaternionRotationAxis(&rot_y, &D3DXVECTOR3(0.0f, 1.0f, 0.0f), D3DXToRadian(-inRotation.y));
    ::D3DXQuaternionRotationAxis(&rot_z, &D3DXVECTOR3(0.0f, 0.0f, 1.0f), D3DXToRadian(-inRotation.z));

    // カメラの軸の設定
    m_Rotation = rot_x * rot_y * rot_z;
    ::D3DXMatrixRotationQuaternion(&m_View, &m_Rotation);

    Update();
}
```

(12)デバイスがリセットされたときの処理を作成します。

IDirect3DDevice9::Resetメソッドを呼び出すと、デバイスに設定されているすべての状態がデフォルトに戻ります。状態とは、ワールド変換行列、ビュー変換行列、プロジェクション変換行列、マテリアル、ライト、レンダリングステート、テクスチャ、テクスチャステージステート、バーテックスシェーダ、ピクセルシェーダなどです。

DirectX Graphicsは状態を復元する機能を提供していないので、あらかじめ設定値を保存しておき、リセット語に復元するといった処理が必要になります。

カメラを制御するのに必要なビュー変換行列、プロジェクション変換行列もリセット後はデフォルトに戻ります。しかし、カメラクラスでは、ビュー変換行列、プロジェクション変換行列をメンバ変数として保持しているので、リセット後に設定し直せば、リセット前の状態に復元できます。

デバイスのリセット前に呼び出すOnLostDevice関数と、リセット後に呼び出すOnResetDevice関数を完成させ、適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          デバイス消失処理                                         */
/*****************************************************************************************************/
void CCamera::OnLostDevice()
{
}

/*****************************************************************************************************/
/*                                          デバイスリセット処理                                     */
/*****************************************************************************************************/
void CCamera::OnResetDevice()
{
    Update();                                   // ビュー行列の設定
    m_pD3DDevice->※ここは各自考えましょう;      // プロジェクション行列の設定
}
```

OnLostDevice関数は、処理が何もないため本来は必要ありませんが、バックバッファクラスなど、リセット復元処理が必要なクラスと処理を統一するために定義しています。

(13)CDXGraphics9クラスに、カメラ機能を追加します。これは、カメラクラスのオブジェクトをメンバ変数にすること(集約)で行います。以下のプログラムを適切な場所に追加しましょう。

```
CCamera*    m_pCamera;
```

(14)(13)で追加したメンバを初期化します。以下のコンストラクタ初期化子を適切な場所に追加しましょう。

```
m_pCamera(NULL)
```

(15)DirectX Graphicsの初期化時に、カメラクラスのオブジェクトを生成します。以下のプログラムを適切な場所に追加しましょう。

```
// カメラ生成
m_pCamera = new CCamera(m_pD3DDevice);
```

(16)DirectX Graphicsの解放時に、カメラクラスのオブジェクトを解放します。以下のプログラムを適切な場所に追加しましょう。

```
// カメラ解放
delete m_pCamera;
m_pCamera = NULL;
```

(17)デバイスのリセット前に、カメラクラスのあるメンバ関数を呼び出します。以下のプログラムを完成させ、CDXGraphics9::Reset関数の適切な場所に追加しましょう。

```
// リセット前処理
m_pCamera->???????????();
```

(18)デバイスのリセット後に、カメラクラスのあるメンバ関数を呼び出します。以下のプログラムを完成させ、CDXGraphics9::Reset関数の適切な場所に追加しましょう。

```
// リセット後処理
m_pCamera->????????????();
```

(19)CDXGraphics9クラスに、カメラオブジェクトを返す関数を追加します。この関数により、CDXGraphics9クラスの外でもカメラの機能を使えるようになります。
以下のプログラムを適切な場所に追加しましょう。

```
CCamera* GetCamera() const { return m_pCamera; }
```

(20)以下のインライン関数を適切な場所に追加しましょう。

```
inline CCamera* DXGCamera() { return DXGraphics().GetCamera(); }
```

カメラの機能を呼び出すには「DXGraphics().GetCamera->メンバ名」という比較的長い名前を記述しなければなりません。上記のようなインライン関数を定義しておくと、「DXGCamera()->メンバ名」という短い名前で呼び出すことができるようになります。

(21)カメラの視野を設定してみます。以下のプログラムをCTestScene::CTestScene関数に追加しましょう。

```
// カメラ視野設定
DXGCamera()->SetProjection(30.0f, 640.0f, 480.0f, 0.1f, 1000.0f);
```

(22)カメラの位置と角度を変更してみます。以下のプログラムをCTestScene::CTestScene関数に追加しましょう。

```
// カメラ位置、角度設定
DXGCamera()->SetView(D3DXVECTOR3(0.0f, 1.0f, -5.0f), D3DXVECTOR3(0.0f, 0.0f, 0.0f));
```

(23)LookAt方式でカメラの位置、角度を設定してみます。(22)を以下のように変更しましょう。

```
// カメラ位置、角度設定
DXGCamera()->SetViewLookAt(D3DXVECTOR3(0.0f, 1.0f, -5.0f), D3DXVECTOR3(0.0f, 0.0f, 0.0f), 0.0f);
```

応用問題　入力装置によってカメラを移動、回転、公転させてみましょう。