Chapter 1

Xファイルを使用したアニメーションしないモデル表示。

1.1 Xファイル

Xファイルとは DirectX2.0 から導入されたモデルフォーマットで、DirectX9 までサポートされていました。DirectX10 以降は標準モデルフォーマットというものは用意されなくなり、自前でモデル表示処理を実装する必要があります。

現在はサポートされていないフォーマットのため、X ファイルの使い方を学ぶ意味は薄いように思えるかもしれませんが、そもそもどこの環境に行っても使えるモデルフォーマットなど存在しません。自前でエンジンを作っている会社は自分たちでモデルフォーマットを作成しています。しかし、モデルを表示するための基本的な概念はどのモデルフォーマットでも共通となっており、x ファイルを使用したモデル表示の仕方を学んでおけば、独自のモデルフォーマットを使用するライブラリに出会ったとしても、「似たような感じだな」というように思えるはずです。

> Tips
> モデルの表示に関して、どの環境でも通用する技術とは頂点バッファ、インデックスバッファ、シェーダーなど低レベルな知識になります。非常に重要な知識で先生は大好きな分野なのですが、２年生の前期にこれをやっていると就職作品の作成を開始するまでにアニメーションするモデル表示まで話を勧められません。ですので、この手の基礎の話は後期に行います。

1.2　X ファイルのロード

X ファイルを用いてモデルを表示するためには、D3DXLoadMeshFromX 関数を使用して X ファイルをロードして、ID3DXMesh のインスタンスを作成する必要があります。ID3DXMesh とは内部にモデルを表示するための頂点バッファやインデックスバッファを保持したモデルクラスのようなものです。この API は下記のように使用します。

```
D3DXLoadMeshFromX(
   "Tiger.x",                    //ファイルパス
   D3DXMESH_SYSTEMMEM,           //メッシュ作成のオプション。基本これでいい。
                                 //他にも大事なオプションはあるのだが、今は説明しない。
   g_pd3dDevice,                 //D3D デバイス。
   NULL,                         //ポリゴンの隣接情報の出力先。
                                 //モデルをロードするだけなら NULL でいい。
   &pD3DXMtrlBuffer,             //マテリアルバッファの出力先。後述。
   NULL,                         //NULL でいい。
   &g_dwNumMaterials,            //マテリアルの数の出力先。後述。
   &g_pMesh                      //ID3DXMesh のインスタンスの格納先。
)
```

これで ID3DXMesh のインスタンスが生成されました。

1.3 マテリアル

マテリアルとはモデルの質感を決定するためのものです。例えばテクスチャ、鏡面反射率などの設定を行うものです。D3DXLoadMeshFromX から取得できるテクスチャ以外のマテリアル情報は固定機能と呼ばれる、現在は廃れた機能の情報しか取得できないため。今回使用するサンプルでは使用しません。今回のサンプルではマテリアル情報はテクスチャを引っ張ってくるためだけに使用します。マテリアルからテクスチャを引っ張ってくるコードは下記のようになります。

```
D3DXMATERIAL* d3dxMaterials = ( D3DXMATERIAL* )pD3DXMtrlBuffer->GetBufferPointer();
//テクスチャ配列を new
g_pMeshTextures = new LPDIRECT3DTEXTURE9[g_dwNumMaterials];
//マテリアルの数だけループを回してテクスチャをロード。
for( DWORD i = 0; i < g_dwNumMaterials; i++ )
{
    g_pMeshTextures[i] = NULL;
    if( d3dxMaterials[i].pTextureFilename != NULL &&
        lstrlenA( d3dxMaterials[i].pTextureFilename ) > 0 )
    {
        // テクスチャを作成。
        if( FAILED( D3DXCreateTextureFromFileA( g_pd3dDevice,
                                                d3dxMaterials[i].pTextureFilename,
                                                &g_pMeshTextures[i] ) ) )
        {
            /テクスチャが見つからなかった。
            MessageBox( NULL, "Could not find texture map", "Meshes.exe", MB_OK );
        }
    }
}
```

1.4 エフェクトファイルのロード。

モデルを表示するためには、拡張子が.fx のエフェクトファイルと言われるものをロードする必要があります。このエフェクトファイルは HLSL という言語で記述されたシェーダープログラムになります。シェーダーは近年のゲームのグラフィックスを語る上で欠かすことのできない、非常に重要な要素になります。しかし、この話をするだけでかなりの時間がかかりますので、この話は後期に行います。今はこのように記述を行う必要があるのだなという風にだけ覚えておいてください。

エフェクトファイルのロードは下記のように行います。

```
//シェーダーをコンパイル。
HRESULT hr = D3DXCreateEffectFromFile(
            g_pd3dDevice,
            "basic.fx",
            NULL,
            NULL,
#ifdef _DEBUG
            D3DXSHADER_DEBUG,
#else
            D3DXSHADER_SKIPVALIDATION,
#endif
            NULL,
            &g_pEffect,
            &compileErrorBuffer
            );
if (FAILED(hr)) {
            MessageBox(NULL, (char*)(compileErrorBuffer->GetBufferPointer()), "error", MB_OK);
            std::abort();
}
```

1.5 モデルの描画処理

ここまでは全て初期化と言われる処理で、これでやっとモデルを表示することができます。では実際にモデルを描画するコードを見てみましょう。

```
//シェーダー適用開始。
g_pEffect->SetTechnique("SkinModel");
g_pEffect->Begin(NULL, D3DXFX_DONOTSAVESHADERSTATE);
g_pEffect->BeginPass(0);

//定数レジスタに設定するカラー。
D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);
//ワールド行列の転送。
g_pEffect->SetMatrix("g_worldMatrix", &g_worldMatrix);
//ビュー行列の転送。
g_pEffect->SetMatrix("g_viewMatrix", &g_viewMatrix);
//プロジェクション行列の転送。
g_pEffect->SetMatrix("g_projectionMatrix", &g_projectionMatrix);
 //回転行列を転送。
g_pEffect->SetMatrix( "g_rotationMatrix", &g_rotationMatrix );
//ライトの向きを転送。
g_pEffect->SetVectorArray("g_diffuseLightDirection", g_diffuseLightDirection, LIGHT_NUM );
//ライトのカラーを転送。
g_pEffect->SetVectorArray("g_diffuseLightColor", g_diffuseLightColor, LIGHT_NUM );
//環境光を設定。
g_pEffect->SetVector("g_ambientLight", &g_ambientLight);

//この関数を呼び出すことで、データの転送が確定する。描画を行う前に一回だけ呼び出す。
g_pEffect->CommitChanges();

// Meshes are divided into subsets, one for each material. Render them in
    // a loop
    for( DWORD i = 0; i < g_dwNumMaterials; i++ )
    {
        //テクスチャを設定。
        g_pEffect->SetTexture("g_diffuseTexture", g_pMeshTextures[i]);
        //描画。
        g_pMesh->DrawSubset( i );
    }

g_pEffect->EndPass();
g_pEffect->End();
```

よくわからないコードが多いかと思います。非常に長いコードになりましたが、これがモデルを表示するときに必要なコードになります。まだ、よく分からない部分がいくつかあるかと思いますが、今は構いません。少なくともワールド行列、ビュー行列、プロジェクション行列の設定などをしている部分を分かってもらうだけで今は十分です。

1.6 終了処理。

プログラムが終了、もしくはモデル表示が不要になった場合は、ここまでロードしたID3DXMeshやテクスチャ、エフェクトファイルなどを破棄する必要があります。下記に破棄を行うコードを記述します。モデルが不要になったら必ず終了処理を実行するように気をつけてください。

```
//テクスチャを破棄
if (g_pMeshTextures != NULL) {
   for (int i = 0; i < g_dwNumMaterials; i++) {
      g_pMeshTextures[i]->Release();
   }
   delete[] g_pMeshTextures;
}
//メッシュを破棄
if (g_pMesh != NULL) {
   g_pMesh->Release();
}
//エフェクトを破棄
if (g_pEffect != NULL) {
   g_pEffect->Release();
}
```

1.7 まとめ

Xファイルを使用してモデルを表示するためには下記の手順が必要でした。

① D3DXLoadMeshFromX 関数 を使用してXファイルをロードし、ID3DXMeshのインスタンスを作成する。(初期化時に一度だけ実行)

② D3DXLoadMeshFromX関数を使用して取得できたマテリアル情報を元にD3DXCreateTextureFromFileA関数を使用してテクスチャをロードする。(初期化時に一度だけ実行)

③ D3DXCreateEffectFromFile関数を使用してエフェクトファイルをロードする。(初期化時に一度だけ実行)

④ ロードした要素を使用してモデルの描画処理を記述する。(毎フレーム実行する)

実習課題

下記のURLから実習用のプログラムをpullして、実習を行ってください。

① トラをクラス化してみましょう。

トラのクラスの最低限の要求仕様。

・トラのクラスは下記のメンバ関数実装する。

Init関数を実装するようにしてく

Xファイル、テクスチャ、エフェクトファイルのロードなどの処理を記述する。

Update関数

ワールド行列の更新やトラの移動などを記述する関数。

Render 関数

トラの描画処理を記述する。

Release 関数

メッシュ、エフェクト、テクスチャなどを破棄するコードを記述する。

・トラのクラスは下記のメンバ変数を最低限保持する。

```
ID3DXEffect*     pEffect;
D3DXMATRIX    worldMatrix;
LPD3DXMESH    pMEsh;
LPDIRECT3DTEXTURE9* pMeshTextures
DWORD numMaterial;
```

② トラを２体出してください。

Chapter 2 モデルクラスの作成

Chapter1 で作成したトラクラスはトラ固有の処理と、モデルを表示するための処理が記述されていて、まだまだ設計に改善の余地があります。例えば Chapter1 で作成したサンプルプログラムにヒヨコのクラスを追加するケースを考えてみて下さい。恐らくあなたは下記のようなクラスを作ることを思いつくはずです。

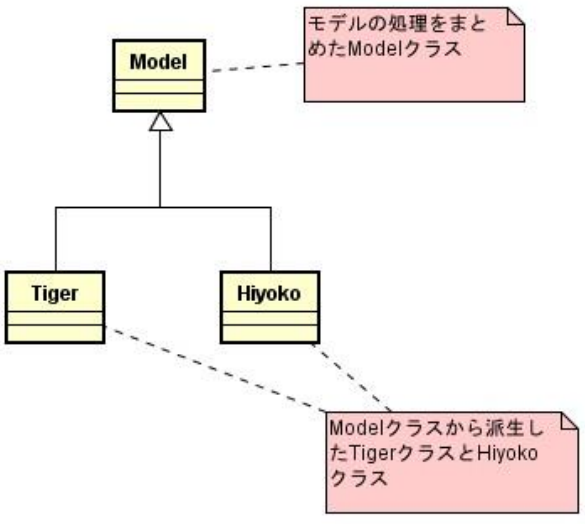
```
class Hiyoko{
};
```

当然ヒヨコクラスも３Ｄモデルを表示する必要があるので、モデルを表示するプログラムを記述していくはずです。そして、ヒヨコクラスの実装を進めていくうちにモデルを表示する処理の大部分がトラクラスの実装と共通であることに気付くと思います。ソフトウェア工学において、共通の処理をコピーアンドペーストで増やしていくことは、保守性、可読性、再利用性を大きく損なう行為になります。コピーアンドペーストでコードを複製していった場合、モデル表示プログラムに不具合があったときや拡張が必要になった場合、コピーアンドペーストを行った数だけ修正が必要になるのです。そして、人は作業の数が膨大になるほど、ヒューマンエラーを起こす確率が高くなるため、そのプログラムを保守する人は頭を抱えることになるでしょう。

2.1 継承 vs 移譲

2.1.1 継承

C++の継承を学んだプログラマであれば、この問題の解決に継承を使用しようと考えるでしょう。恐らく下記のようなクラスを設計すると思います。



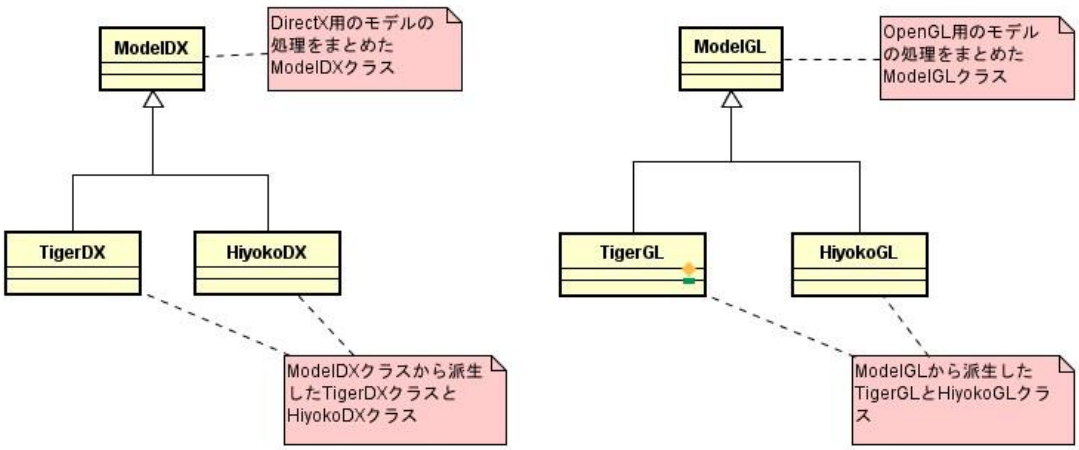
プログラムは下記のようなものになっているでしょう。

```
class Model{
    //定義は省略。
};
class Tiger：public Model
{
   //定義は省略。
};
class Hiyoko：public Model
{
    //定義は省略。
};
```

Tiger クラスと Hiyoko クラスから、共通するモデル関連の処理を抽出した Model クラスを作成して、Tiger と Hiyoko を Model の派生クラスにしています。これによって、モデル関連の処理への修正や、拡張の作業が発生した場合は、Model クラスの処理のみを変更すれば良くなります。コピーアンドペーストで処理を増やしていく実装に比べると、かなり改善されたと言えます・・・。しかしこの設計でもまだ大きな問題が起きるケースがあります。次節ではその問題について見ていきましょう。

2.1.2 組み合わせの爆発

では、先ほどの Tiger クラス、Hiyoko クラス、Model クラスについて見てみましょう。もともと Model クラスは Microsoft 社が提供する SDK の DirectX で実装をされていました。しかし、ある日クライアントから次のような要求が来ました。「DirectX が嫌いなユーザーも遊べるように OpenGL でも動作するように拡張して欲しい。」実際、昔このような要望を社内ツールの開発でデザイナーから受けたことがあります。この要望に応えるために、あなたは下記のように設計を変更しました。



この設計変更により、TigerDX と TigerGL、HiyokoDX と HiyokoGL は共通のコードが多数あるコピーアンドペーストと同じ保守性、拡張性、再利用性の低いクラスになってしまいました。

2.1.3 委譲

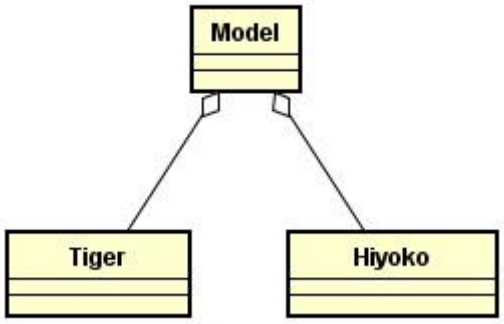
「継承よりも委譲を使おう」オブジェクト指向を用いた、よりよい設計を考える際に、継承を行う場合、先に移譲が行えないか検討することが推奨されています。では委譲とはなにか？これはあるクラスの責任を別のクラスに譲り渡すことです。ではもともとの虎クラスを見てみましょう。元々の虎のクラスは下記の二つの処理を正しく実行する責任がありました。

・トラの挙動(歩くとか走るとか)

・トラの表示する

この二つの処理のうち、「トラを表示する」という処理を、新しく Model といクラスを作成して責任を譲り渡します。これが委譲です。そして、トラクラスは Model クラスを継承するのではなく、Model クラスのインスタンスを保持する形に変更します。これがコンポジションや集約と呼ばれるものです。

では、委譲を使用した場合のクラス図を見てみましょう。



クラス図を見ても分かりにくいかと思いますので、実際のコードを見てみましょう。

```
class Model{
    //定義は省略。
};
class Tiger
{
   Model   model;    //Model のインスタンスを保持！！！
   //定義は省略。
};
class Hiyoko
{
    Model   model;    //Model のインスタンスを保持！！！
    //定義は省略。
};
```

Tiger、Hiyoko が Model クラスのインスタンスを保持しています。これがコンポジション、集約といわれるものです。では、なぜこれが継承を使用した設計より優れているのかを、先ほどの DirectX、OpenGL の話から考えてみましょう。

まず、DirectX 用の ModelDX クラスと OpenGL 用の ModelGL クラスを用意する必要があります。しかし、モデルクラスというのは往々にして共通のインターフェースを保持するものです。例えば Draw 関数とか。そこで、基底クラスに Model クラスを作成します。そして Tiger と Hiyoko クラスには Model クラスのポインタを保持させます。

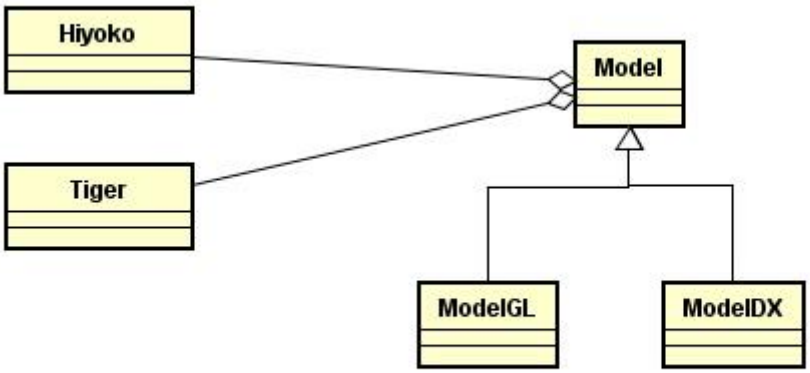
```
//モデルの基底クラス。
class Model{
public:
   virtual void Draw() = 0;   //純粋仮想関数。
};
//DirectX 用のモデルクラス。
class ModelDX : public Model{
public:
   void Draw();
};
//OpenGL 用のモデルクラス。
class ModelGL : public Model{
public:
   void Draw();
};
//トラクラス。
class Tiger {
   Model*    model;
public:
   //モデルのインスタンスを設定。
   void SetModel(Model* pModel)
   {
      model = pModel;
   }
};
//ヒヨコクラス。
class Hiyoko{
   Model*    model;
public:
   //モデルのインスタンスを設定。
   void SetModel(Model* pModel)
   {
      model = pModel;
   }
};
```

では、この設計の最後のトリックを紹介します。

```
Hiyoko hiyoko;   //ヒヨコ
Tiger tiger;     //トラ
void Func()
{

    if( オープンＧＬを使用する場合 ){
       hiyoko.SetModel(new ModelGL );
          tiger.SetModel(new ModelGL);
    }else if( DirectX を使用する場合 ){
       hiyoko.SetModel( new ModelDX );
          tiger.SetModel( new ModelDX );
    }
}
```

では最後にクラス図を見てみましょう。



いかがでしょうか。見事に冗長性が排除され、拡張性、保守性に優れた設計になっています。

2.2 まとめ

継承と委譲に関して、絶対に継承よりも委譲を使用しなさいというものではありません。ただ、設計の指針として継承よりも委譲を使おうという指針を頭に入れておくだけでも、設計はより優れたものになります。

Chapter 3 アニメーション

3.1 モーフィング

この節では頂点単位のアニメーションのモーフィングについて見ていきます。モーフィングはフェイシャルアニメーション(顔のアニメーション)でよく使われており、昨今のゲームには欠かすことのできない技術になっております。フェイシャルアニメーションはボーンを使用して実装することもできますが､最近のフォトリアルなゲームは役者の顔で３Ｄキャプチャーを行い、モーフターゲットとして使用することでリアルな表情を実現しています。

3.1.1 モーフターゲット

モーフィングを行うためには､モーフターゲットというデータが必要になります。モーフターゲットを簡潔に説明すると、例えばキャラクタを無表情から笑っている顔にアニメーションさせたい場合、無表情のモデルと笑っているモデルの二つを作成します。そして、無表情のモデルと笑っているモデルとで､0.0～1.0 のブレンディング率を使用して､同じ番号の頂点をブレンディングしていきます。頂点ブレンディングの計算式は下記になります。

モデル A の 100 番目の頂点を VA、モデル B の 100 番目の頂点を VB として、ブレンディング率を R とすると

モーフィング後の頂点 = VA * (1.0 -R) + VB * R

となる。

3.1.2 DirectX での頂点アクセス

実際にモーフィングを行うためにはモデルの頂点バッファにアクセスする必要があります。ここではモデルの頂点バッファにアクセスする方法を紹介します。今回はソフトウェアモーフィングを行いますので、CPU でモーフィングを行うことにします。

X ファイルをロードすると、ID3DXMesh のインスタンスを使用してモデルの表示などが行えます。このインスタンスを使用すれば、頂点バッファにアクセスすることができます。頂点バッファを取得するには ID3DXMesh:: GetVertexBuffer を使用します。

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
mesh->GetVertexBuffer(&vertexBuffer); //頂点バッファを取得。
```

頂点バッファとは、モデルの頂点情報をまとめて管理するバッファです。下記のようなバッファと考

えるとイメージしやすいのではないでしょうか。

```
//頂点
struct Vertex{
  D3DXVECTOR3   pos;   //座標
  D3DXVECTOR3   normal; //頂点の向きを表す法線。
  D3DXVECTOR2   uv;    //テクスチャをサンプリングするための UV 座標。
};

Vertex vertexBuffer[1256];  //頂点数が 1256 の頂点バッファ。
```

このコードは擬似コードなのですが、イメージはこのようになります。
さて、頂点データを書き換えるためには、CPU で頂点を書き換えている最中に GPU がその頂点バッファにアクセスできないようにロックをかける必要があります。頂点バッファのロックは LPDIRECT3DVERTEXBUFFER9 の Lock 関数を使用すれば実行できます。

```
char* pVertex;
vertexBuffer->Lock(0, desc.Size, (void**)&pVertex, D3DLOCK_DISCARD);
```

Lock 関数を使用すると頂点バッファをロックすることができ、pVertex に頂点バッファに対する生のメモリアドレスが格納されます。ロックを行ったあとは、pVertex を使って直接頂点バッファを書き換えることができます。
頂点の書き換えが完了したら、頂点バッファをアンロックする必要があります。アンロックを忘れてしまうと、GPU がいつまでたってもその頂点にアクセスすることができなくなるため、GPU がフリーズします。

```
vertexBuffer->Unlock();
```

3.1.2.1 頂点ストライド

頂点情報はモデルによって内容が変わります。例えばテクスチャを貼らないモデルであれば UV の要素はいらなくなりますし、ライティングを行わない場合は normal の要素がいらなくなることもあります。そのため、頂点にアクセスするときは一つの頂点のサイズが必要になります。一つの頂点のサイズは次のようなコードで取得できます。

```
//頂点バッファの定義を取得する。
D3DVERTEXBUFFER_DESC desc;
vertexBuffer->GetDesc(&desc);
//一つの頂点のサイズを計算する。
//desc.sizeには頂点バッファのサイズが入っているので、
//これを頂点数で除算してやれば一つの頂点のサイズがわかります。
```

```
int stride = desc.Size / mesh->GetNumVertices();
```

3.1.2.2

では頂点を書き換えるためのプログラムを見てみましょう。

```
D3DXVECTOR3* vertexPos;
//頂点バッファをロック
vertexBuffer->Lock(0, desc.Size, (void**)& vertexPos _B, D3DLOCK_DISCARD);
for (int vertNo = 0; vertNo < mesh->GetNumVertices(); vertNo++) {
    //頂点座標に+1.0 していく。
    vertexPos->x += 1.0f
    vertexPos->y += 1.0f:
    vertexPos->z += 1.0f

    //次の頂点へ。
    char* p = (char*)vertexPos;
    p += stride;
}
vertexBuffer->Ulock();
```

実習課題

モーフィングを学ぶ課題を使用して、ユニティちゃんがフェイシャルアニメーションできるようにしてください。

3.2 スキンアニメーション

すでにスケルトン(骨組み)を使用した階層アニメーションは勉強しましたが、ここまで勉強した階層アニメーションの手法では、複数のパーツに分かれているオブジェクトを描画するときに切れ目が発生したり、人肌のようなワンメッシュのモデルであっても関節のつなぎ目で不自然なアーティファクトが発生してしまいます。これを解決するための手法がスキンアニメーションまたはスキニングと言われるものです。

3.2.1 スキンウェイト

ではどのようにすればパーツの切れ目や、不自然なアーティファクトを除去することができるのでしょうか？例えば人体の腕について考えてみましょう。人の腕は肩から上腕、前腕、掌、手の指など多数のボーンが存在します。今回は上腕と前腕について考えてみましょう。

上腕と前腕をアニメーションさせる場合、3dsMax などの DCC ツールを使用して 3D モデルデータの上腕と前腕の各頂点がどのボーンに関連づいているかを設定することで、ボーンを使用した階層アニメーションが実現できます。さて、腕の各頂点にボーンと関連付けを行うと言いましたが、例えば肘の付近の頂点は上腕と前腕のどちらのボーンに関連付けを行えばいいでしょうか？どちらに関連付けを行っても不自然なアーティファクトが発生しそうです。これを解決するのがスキンウェイトと言われるものです。

では肘の話に戻します。肘のような骨と骨のつなぎ目の関節付近の頂点は上腕と前腕の二つのボーンに関連付けを行います。そして、例えば上腕のボーンに 0.4 の重みで影響を受けて、前腕のボーンに 0.6 の重みを受けるように設定します。この重みが**スキンウェイト**と呼ばれるものです。

では、肘の頂点をどのように変換するのか疑似コードを示します。肘の頂点を vSrc、上腕のボーン行列を m0、前腕のボーン行列を m1、上腕のボーンへのスキンウェイトを w0、前腕のボーンへのスキンウェイトを w1、変換後の頂点を vDst とした場合、下記のようなコードになります。

```
D3DXVECTOR4 vTmp;
//上腕のボーン行列で変換させた頂点座標を vTmp に代入。
D3DXVec4Transform(&vTmp, &vSrc, &m0);
//スキンウェイトを乗算して vDst に代入。
vDst = vTmp * w0;
//前腕のボーン行列で変換させた頂点座標を vTmp に代入。
D3DXVec4Transform(&vTmp, &vSrc,&m1);
//スキンウェイトを乗算して vDst に加算
vDst += vTmp * w1;
```

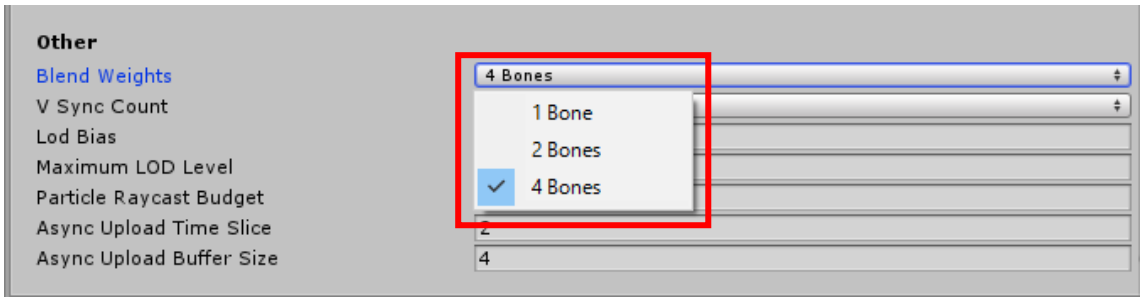
3.2.2 関連付けできるボーンの本数

スキンウェイトを設定できるようになれば、各頂点に関連付けできるボーンの数も増やすことができます。例えば肩の辺りの頂点であれば、胴体、上腕、首と３つのボーンと関連付けされているかもしれません。しかしこれらもスキンウェイトを使えば簡単に解決できます。胴体のボーンに 0.3、上腕に 0.3、首に 0.4 のスキンウェイトを設定すればいいのです。複数ボーンに関連付けできる場合のスキニングの疑似コードを下記に示します。

```
D3DXVECTOR3 dstPos = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
for (int boneNo = 0; boneNo < numBone; boneNo++)       //ボーンの本数分ループを回す。
{
    D3DXVECTOR4 vTmp;
    D3DXVec4Transform(&vTmp, &vSrc, &boneMatrixArray[IndexArray[boneNo]] );
    dstPos += vTmp * blendWeightsArray[iBone];
}
```

3.2.3 スキニングのパフォーマンス

スキニングはモデルのすべての頂点に対して行われます。最近のハイエンドのゲームであれば､キャラクタのモデルの頂点数が 10 万を超えることも珍しくありません｡そのため、スキニングはほとんどのケースで GPU などの高速なプロセッサで計算されます。また、前節の関連付けできるボーンの数に関しても、当然ですが数が少ないほど処理が高速になります。そのため関連付けできるボーンの数に上限を設けているゲームエンジンがほとんどです。下記の図は Unity のクオリティセッティングの図です。



Blend Weights の部分が関連付けできるボーンの上限になります。

3.2.4 実習

スキニングを学ぶ課題を使用して、スキニングを実装してみて下さい。今回は分かりやすくするために CPU でのスキニングのソフトウェアスキニングを実装してもらいます。

3.3 アニメーション付き X ファイル

DirectX9 でサポートされている X ファイルにはアニメーションデータを付随することができます。この節では、アニメーション付き X ファイルの作成の仕方を紹介します。

3.3.1 アニメーションデータ

X ファイルに付随するアニメーションデータは AnimationSet という名前で付随しています。AnimationSet のデータを見てみると、キーフレームが打たれており、そのキーの時に各ボーンがどのような姿勢になっているかという情報が付随されています。

3.3.2 アニメーション付き X ファイルの出力方法。

ではアニメーション付き X ファイルを出力する方法を動画を使用して説明します。今回は 3dsMax の kwxport.dle というプラグインを使用します。GitHub¥DirectXLesson_2¥動画¥アニメーション付き X ファイルの出力.mp4 を参照してください。

3.3.3 アニメーションの追加

kwxport.dle は一つの X ファイルに対して一つのアニメーションしか出力することができません。しかし、ゲームでは複数のアニメーションを使用して多彩なアクションを行うことでキャラクターに息を吹き込んでいます。ここでは X ファイルにアニメーションを追加して、複数のアニメーションを扱う方法を勉強します。こちらも動画を用意していますので、GitHub¥DirectXLesson_2¥動画¥アニメーションの追加.mp4 を参照してください。

4　実例で学ぶゲーム数学

この節では DirectX を使用して、ゲームでの数学の活用法を紹介していきます。この授業は数学の授業ではないため、公式の証明などの話はしません。先人の考えた公式をありがたく使わせてもらおうという趣旨の授業になります。

4.1　ベクトル

3D ゲームにおいて、ベクトルは多種多様な用途で活用されています。活用例を下記に示します。

・3D オブジェクトの座標
・移動速度
・3D オブジェクトの向き
・ポリゴンの向きを表す法線
・モデルの頂点を表す頂点座標
・etc

このように、ベクトルは 3D ゲームを作成するうえで欠かすことのできない要素になっています。では次の節からはゲームでよく使われるベクトルの使い方を見ていきましょう。

4.1.1　2 点間の距離

2 点間の距離の計算は衝突判定や、AI の敵発見の思考など、いろいろな箇所で多用される計算になります。3D 空間ではオブジェクトの座標は 3 次元のベクトルで表現されています。例えば、パックマンのようなゲームで、プレイヤーに食べ物が衝突すると食べ物が消滅する仕様を実装するケースを考えてみましょう。

プレイヤーの座標を playerPos、食べ物の座標を foodPos とする。

```
//まず食べ物からプレイヤーに向かうベクトルが計算される。
D3DXVECTOR3 toPlayer　= playerPos – foodPos;
//このベクトルの距離は三平方の定理を活用して求める。sqrt は平方根を求める関数。
float length = sqrt( toPlayer.x * toPlayer.x + toPlayer.y * toPlayer.y + toPlayer.z * toPlayer.z);
if(length < 0.2f){
   //食べ物とプレイヤーの距離が 0.2 以下になったら・・・
}
```

DirectX にはベクトルの長さを計算する関数 r が用意されています。上のコードを DirectX の関数を使用するように書き換えてみましょう。

```
//まず食べ物からプレイヤーに向かうベクトルが計算される。
D3DXVECTOR3 toPlayer　= playerPos – foodPos;
//このベクトルの距離は三平方の定理を活用して求める。sqrt は平方根を求める関数。
float length = D3DXVec3Length(&toPlayer);
if(length < 0.2f){
　//食べ物とプレイヤーの距離が 0.2 以下になったら・・・
}
```

4.1.2 ベクトルの正規化

ベクトルとは大きさと向きを持った情報です。ゲーム数学ではベクトルから大きさの要素を除外して、向きだけがほしい場合が多々あります。このように、ベクトルから大きさを除去して(大きさを 1 にする)、向きだけの情報を持ったベクトルにすることを正規化と言います。では、敵がプレイヤーに向かって移動していく仕様の実装で正規化の活用の具体例を見てみましょう。

敵の座標を enemyPos、プレイヤーの座標を playerPos とする。

```
//敵からプレイヤーに向かうベクトルを計算する。
D3DXVECTOR3 toPlayerDir = playerPos – enemyPos;
//toPlayerDir には敵からプレイヤーまでの向きと、大きさが入っているため正規化を行って、向きだけを抽出する。
//正規化とは、ベクトルの xyz の要素をベクトルの大きさで除算することで行える。
float len = D3DXVec3Length(&toPlayerDir);
toPlayerDir.x /= len;
toPlayerDir.y /= len;
toPlayerDir.z /= len;
//toPlayerDir がプレイヤーまでの方向ベクトルになったので、enemyPos をその方向に動かしていく。
enemyPos += toPlayerDir * 0.2f; //速度 0.2 で動かしていく。
```

DirectX には正規化を行う関数も用意されています。上のコードを DirectX の関数を使用するように書き換えてみましょう。

```
//敵からプレイヤーに向かうベクトルを計算する。
D3DXVECTOR3 toPlayerDir = playerPos – enemyPos;
//toPlayerDir を正規化する。
D3DXVec3Normalize(&toPlayerDir, &toPlayerDit);
//toPlayerDir がプレイヤーまでの方向ベクトルになったので、enemyPos をその方向に動かしていく。
enemyPos += toPlayerDir * 0.2f; //速度 0.2 で動かしていく。
```

4.1.3 内積

内積は 3D ゲームで最も多用される公式の一つと言ってもいいかもしれません。内積とは二つのベクトルから計算されるもので、ベクトル VA、VB の内積は下記のように定義されています。

VA・VB = VA.x * VB.x + VA.y * VB.y + VA.z * VB.z ・・・・・ ①

また余弦定理より

|VA| |VB| Cos θ = VA.x * VB.x + VA.y * VB.y + VA.z * VB.z・・・・・②

となる。

この①に関しては定義であり、このように決められています。内積とはこういうものであると昔の数学者が決めたのです。そして②については、余弦定理の方程式を解くことで容易に証明できます。しかしこの授業はそれを論じる授業ではないので、そこについて言及はしません。しかし、この②番目の定義からゲームで非常に有用に使える要素がいくつか見えてきます。

4.1.3.1 ベクトルのなす角

|VA| |VB| cos θ についてみてみましょう。ここで記述されている |VA| と |VB| は各ベクトルの長さを表しています。ではこの二つのベクトルが正規化された大きさ 1 のベクトルである場合、この公式は下記の様はものになります。

$1 \times 1 \times \cos\theta$

つまり、大きさ 1 のベクトル同士の内積はそのベクトル同士がなす角 θ の cos θ となります。C 言語には acos という cos θ を θ に戻す関数が存在します。これを使用することでベクトル同士のなす角を求めることができます。ではこれをどのような場面で使うのか考えてみましょう。

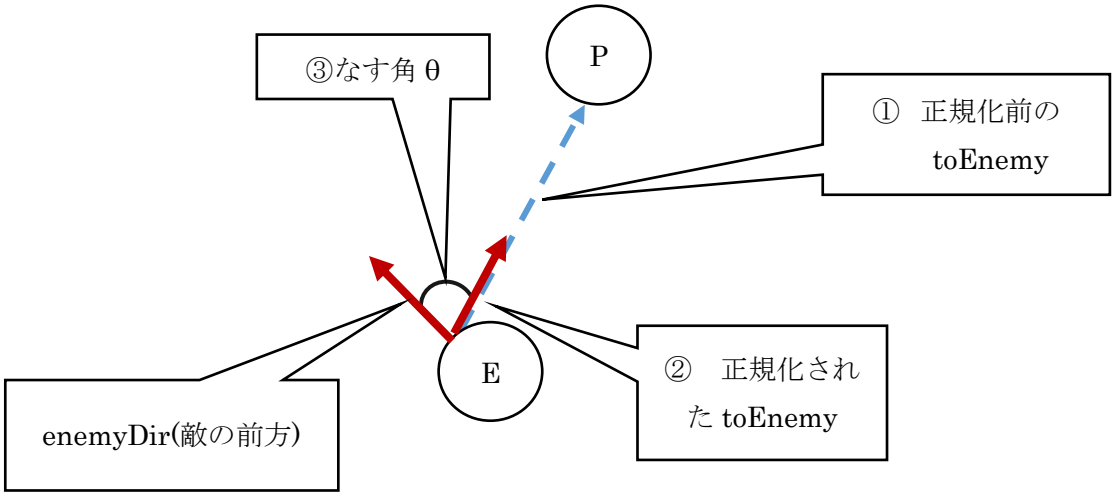
4.1.3.2 視野角の判定

メタルギアソリッド 1 の敵兵の AI について考えてみましょう。メタルギアソリッド 1 の敵兵は視野角というデータを持っていて、プレイヤーがその視野角の中に入るとプレイヤーを発見して追いかけてくる思考になっています。この視野角の判定は内積を使用すれば簡単に行うことができます。ではサンプルコードを見てみましょう。

プレイヤーの座標を playerPos、敵の座標を enemyPos、敵の前方方向を enemyDir とします。

```
//敵からプレイヤーに向かうベクトルを計算する。①
D3DXVECTOR3 toPlayer = playerPos – enemyPos;
//プレイヤーに向かうベクトルを正規化する。②
D3DXVec3Normalize(&toPlayer, &toPlayer);
//敵の前方方向と、プレイヤへの向きベクトルの内積を計算する。
float angle = D3DXVec3Dot(&toPlayer, &enemyDir);
//内積の結果は cos θ になるため、なす角 θ を求めるために acos を実行する。③
angle = acos(angle);
//これで angle にはラジアン単位の角度が入ったため、視野角の判定を行える。
if(fabsf(angle) < D3DXToRadian(45.5f)){
    //視野角 90 度以内に入った。
}
```

この計算を図示すると下記のようになります。

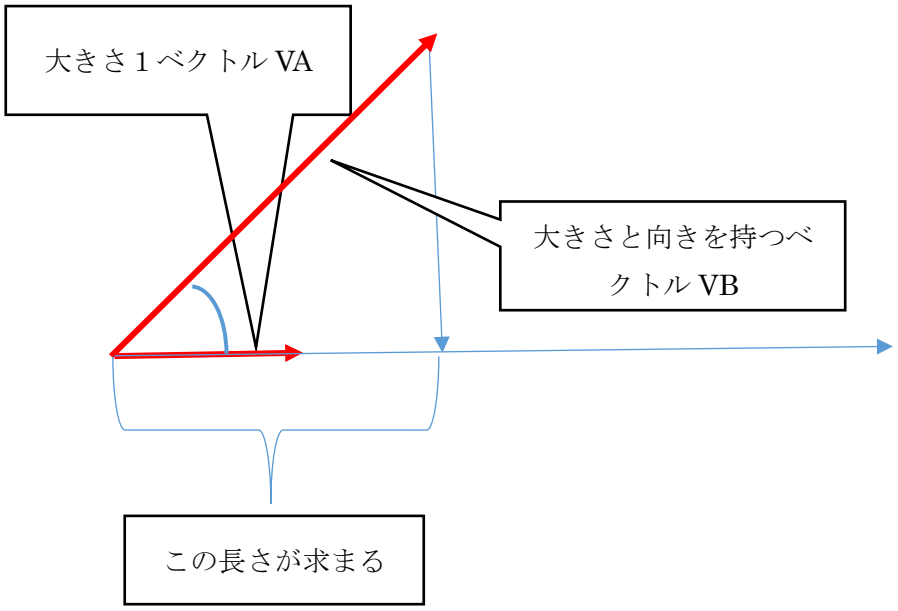


4.1.3.3 射影

|VA||VB|cosθ の公式から、もう一つゲームで使える重要な性質が見えてきます。それは**射影**と言われるもので、例えば VA が大きさ 1 の正規化されたベクトルで、VB が向きと大きさを持っているベクトル(非正規化)だとします。このとき、VA と VB の内積は下記のような計算になります。

$1 \times |VB| \cos\theta = |VB| \cos\theta$

これはつまり、VB のベクトルを VA の無限線分上に垂線を落として射影した長さということになります。

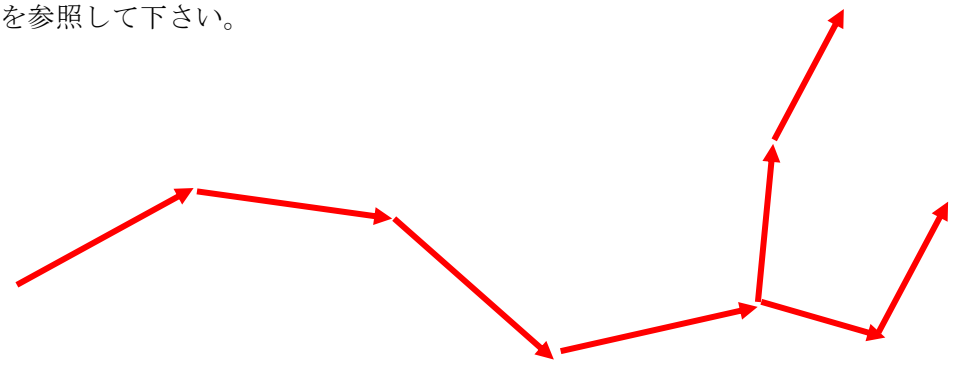


では、この計算が使用される例を見ていきましょう。

4.1.3.3　ゲーム大賞作品 SweetEngineer のコース定義の事例

グランツーリスモ、Forza やマリオカートのようなレースゲームではコースの逆走判定や、AI コースの分岐判定などを行うためにコース定義と言われるデータが使われることがあります。また、God of war やクラッシュバンディクー、今回ゲーム大賞に向けて作成された SweetEngineer などコースに沿ってカメラが移動するようなゲームにもコース定義のようなものが作成されています。God of war3 のカメラはコースドリブンカメラと言うような名前でサンタモニカスタジオで呼ばれていたそうです。
ではコース定義とはどういうものか簡単な例を見ていきましょう。

下記の図を参照して下さい。



このように、コースの向きと長さを保持しているベクトル上のデータが、SweetEngineer でコース定義と呼ばれていたものです。

レースゲームであれば AI はこのコース定義を参照して、現在自分が何処を走っているのかを知り、進行方向に向けてハンドルを切ることになります。
SweetEngineer ではプレイヤーが現在コースのどこにいるのかを調べて、カメラが適切な方向を向くような実装になっています。これは God of war3 のコースドリブンカメラに近い実装です。SweetEngineer ではプレイヤーが現在どのコースにいるのかを判定する際に内積を計算して、プレイヤーの座標をコース上に射影していました。ではそのサンプルプログラムを記述します。プレイヤの座標は playerPos とします。

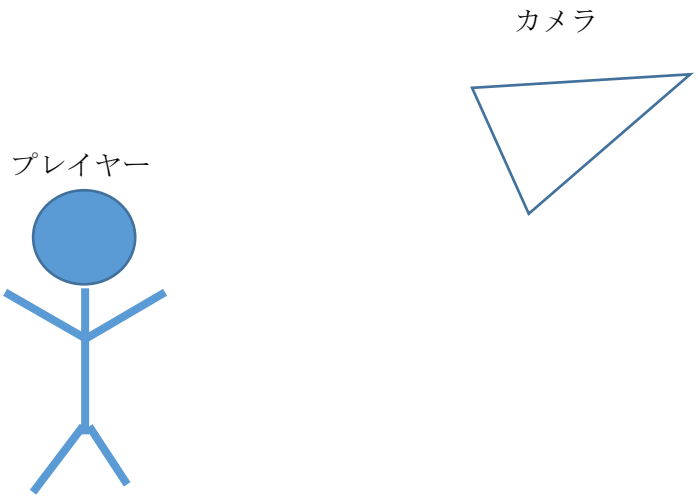
```
//コースのノード。1 ブロックに相当する。
struct CourceNode{
   D3DXVECTOR3 startPos;   //始点
   D3DXVECTOR3 endPos;     //終点
};
std::list< CourceNode* > courceNodes;   //コースのノードのリスト。

void FindCource()
{
    for(auto node : courceNodes){
      //座標がコース上に居るか調べる。
      D3DXVECTOR3 courceDir = node->endPos – node->startPos;
      //コースの 1 ブロックの長さを計算する。
      float courceLen = D3DXVec3Length(&courceDir);
      //コースの 1 ブロックの向きを計算する。
      D3DXVec3Normalize(&courceDir, &courceDir);
      //コースの始点から、プレイヤーの座標までのベクトルを計算する。
      D3DXVECTOR3 toPlayer = playerPos – node->startPos;
      //toPlayer と courceDir の内積を計算する。
      float playerPoInCourceDir = D3DXVec3Normalize(&toPlayer, &courceDir);
      if(playerPoInCourceDir > 0.0f && playerPoInCourceDir < courceLen){
          //コース上
      }
   }
}
```

これだけのコードではまだ幾つか問題が残るのですが、曲がり道の少ないコースであれば十分使えるコードになります。

4.2 三人称視点カメラの回転

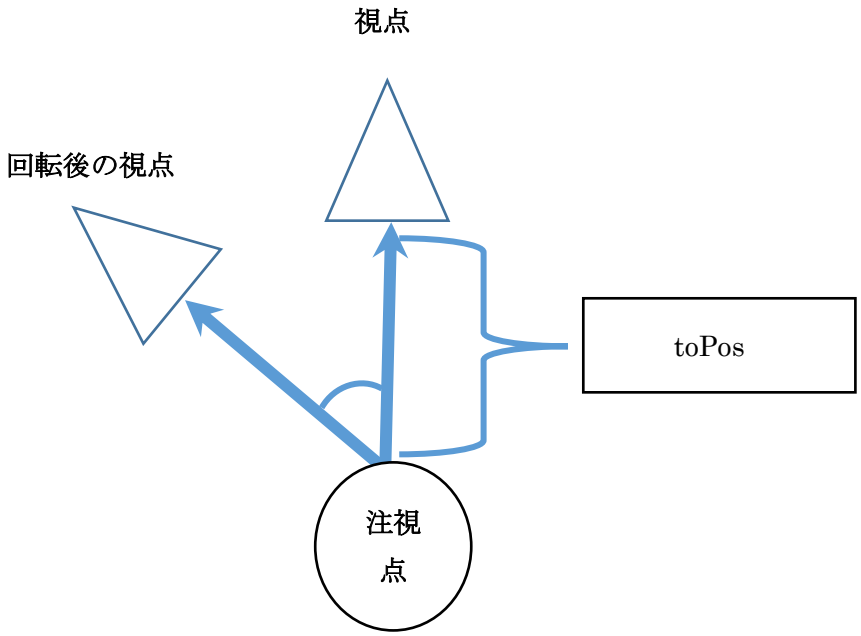
この節では三人称視点カメラの回転を通してベクトルの回転のさせ方を学んでみましょう。三人称視点のカメラとは下記のようなカメラのことをいいます。



4.2.1 Y 軸周りの回転

では、まずカメラを Y 軸周りに回転させる方法を見てみましょう。 下の図は真上から見た図になります。

カメラが Y 軸周りに回るということは、カメラの視点が注視点を中心にして回転することを意味しています。



視点を回転させるには、注視点から視点に向かって伸びるベクトルを回転させて、視点を計算してやればいいのです。

注視点から視点に向かうベクトルは下記のプログラムで求めることができます。

```
//注視点を target、視点を pos として、注視点から視点に向かうベクトルを toPos とすると、
D3DXVECTOR3 toPos = pos – target;
```

このベクトルを Y 軸周りに 10 度回転させるには、Y 軸周りに 10 度回転する行列を使って、ベクトルと乗算してやればいいことになります。

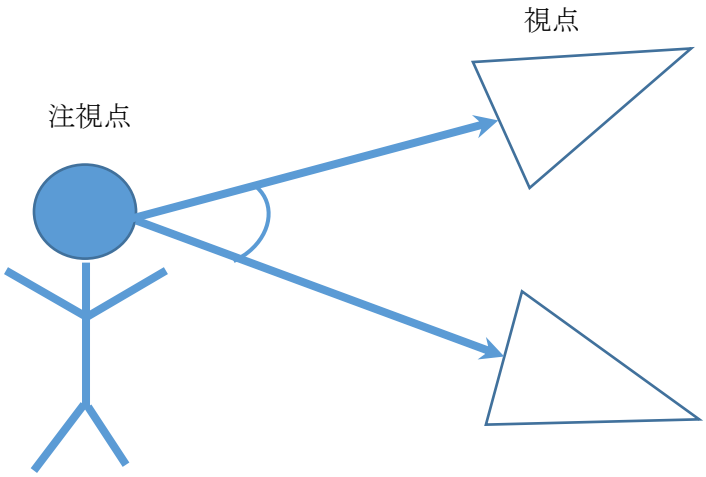
```
//Y 軸周りに 10 度回転する回転行列を作成。
D3DXMATRIX mRot;
D3DXMatrixRotationY(&mRot, D3DXToRadian(10.0f)); //第二引数はラジアン単位。
//ベクトルを回転させる。
//第一引数は計算結果の格納先。D3DXVECTOR4 型なのに注意。
//第二引数は回転元となるベクトル。
//第三引数は回転行列。
D3DXVECTOR4 vOut;
D3DXVec3Transform(&vOut, &toPos, &mRot);
```

そして、toPos を回転させることができたら、このベクトルと注視点を加算したものを視点とすればいいのです。

```
pos.x = target.x + vOut.x;
pos.y = target.y + vOut.y;
pos.z = target.z + vOut.z;
```

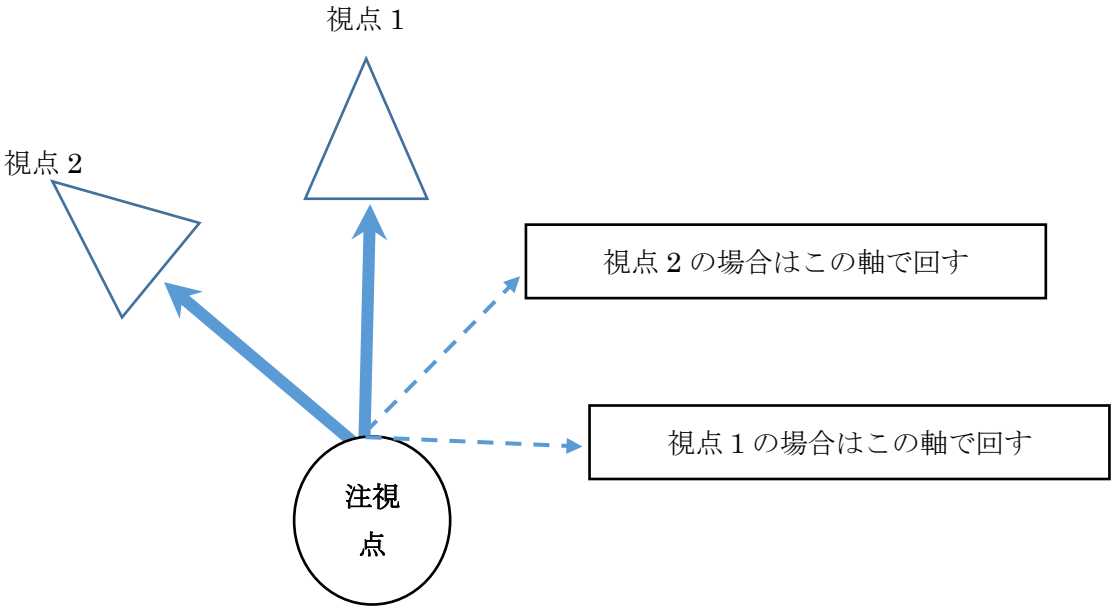
4.2.2 X 軸周りの回転

続いて、カメラの X 軸周りの回転を見ていきましょう。ここでいう X 軸周りの回転とは下記の回転を指します。



この回転も注視点から視点に向かうベクトルを回転させることに変わりはありません。先ほどと違う点は、回転させる軸の求め方です。先程は Y 軸周り固定でしたが、今回はカメラの Y 軸周りの回転を考慮して、回転軸を計算する必要があります。次のページの図を見てみてください。

真上から見た図



このように回す軸が変わることになります。この回転軸を求めるためには外積を活用します。外積とは下記のように定義されるものです。

ベクトル v0、v1 の外積の結果を v2 とすると
v2.x = v0.y×v1.z － v0.z×v1.y
v2.y = v0.z×v1.x － v0.x×v1.z
v2.z = v0.x×v1.y － v0.y×v1.x
となる。

ゲームでよく使われる、外積の特性に「外積の結果は二つのベクトルの直行するベクトルになる」というものがあります。直行とはなす角が 90 度で交わるということです。
この性質を使うことで、視点を回転させるベクトルが計算できます。

まず、注視点から視点に向かうベクトルを計算します。そしてそのベクトルと上方向のベクトルと外積を計算します。この結果が先ほど見た図の回転軸になるのです。

```
//注視点を target、視点を pos として、注視点から視点に向かうベクトルを toPos とすると、
D3DXVECTOR3 toPos = pos – target;
D3DXVECTOR3 vUP( 0.0f, 1.0f, 0.0f );
//外積を計算して、回転軸を求める。
D3DXVECTOR3 vRotAxis;
//第一引数が計算結果の格納先。
//第二引数と第三引数が外積に使用されるベクトル。
D3DXVec3Cross(&vRotAxis, &toPos, &vUp);
```

あとは、この軸周りに回転する行列を作成して、toPos を回転すればいいのです。

```
D3DXMATRIX mRot;
//任意の軸周りの回転行列を作成。
D3DXMatrixRotationAxis(&mRot, &vRotAxis,D3DXToRadian(10.0f));
//ベクトルを回転させる。
//第一引数は計算結果の格納先。D3DXVECTOR4 型なのに注意。
//第二引数は回転元となるベクトル。
//第三引数は回転行列。
D3DXVECTOR4 vOut;
D3DXVec3Transform(&vOut, &toPos, &mRot);

//最後に toPos と target から視点を計算する。
pos.x = target.x + vOut.x;
pos.y = target.y + vOut.y;
pos.z = target.z + vOut.z;
```