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。

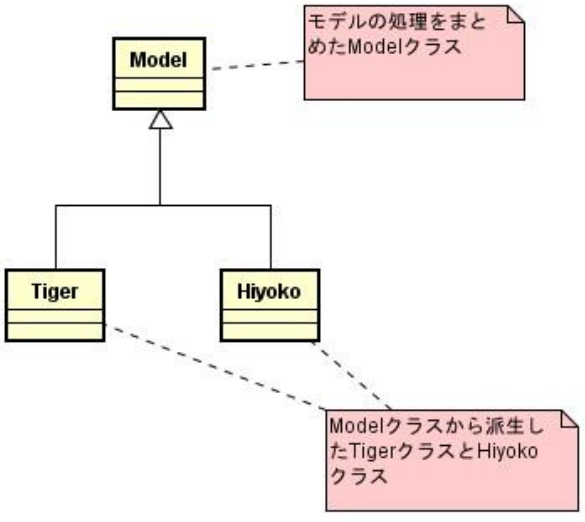
```
class Hiyoko{
};
```

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。



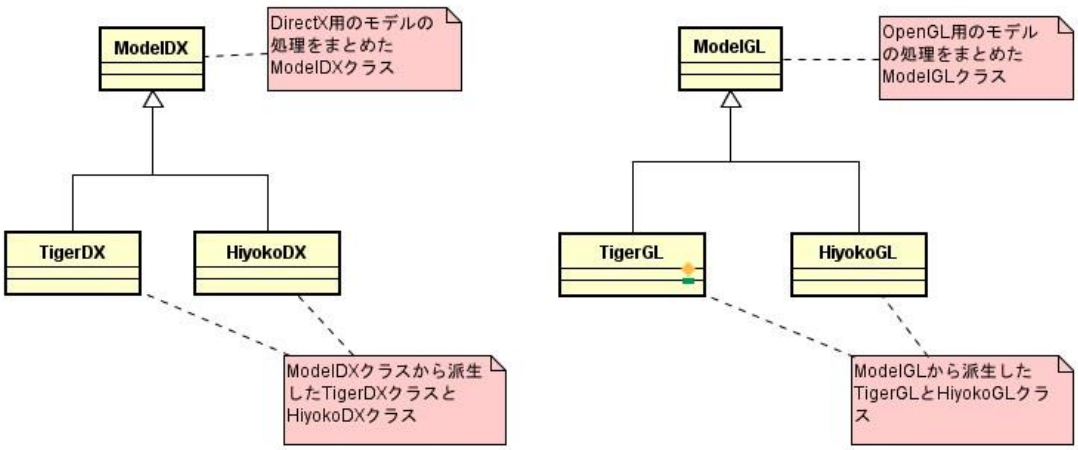
プログラムは下記のようなものになっているでしょう。

```
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

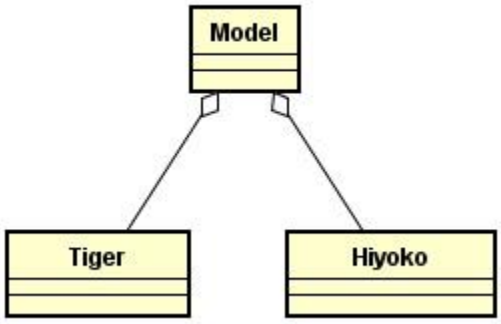
「継承よりも委譲を使おう」オブジェクト指向を用いた、よりよい設計を考える際に、継承を行う場合、先に移譲が行えないか検討することが推奨されています。では委譲とはなにか？これはあるクラスの責任を別のクラスに譲り渡すことです。ではもともとの虎クラスを見てみましょう。元々の虎のクラスは下記の二つの処理を正しく実行する責任がありました。

・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。

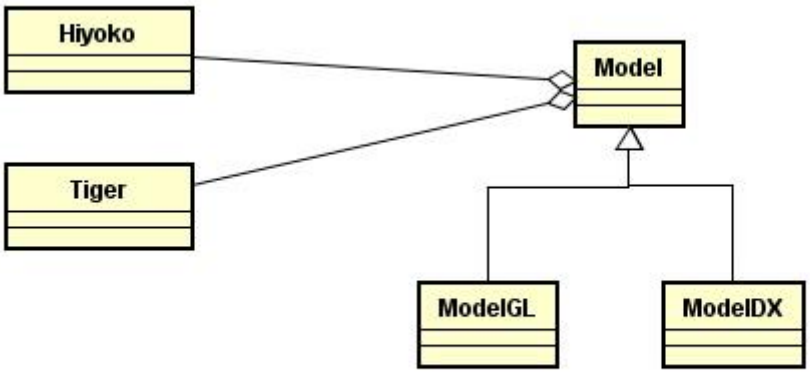
```
//モデルの基底クラス。
class Model{
public:
   virtual void Draw() = 0;   //純粋仮想関数。
};
//DirectX 用のモデルクラス。
class ModelDX : public Model{
public:
   void Draw();
};
//OpenGL 用のモデルクラス。
class ModelGL : public Model{
public:
   void Draw();
};
//トラクラス。
class Tiger {
   Model*    model;
public:
   //モデルのインスタンス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