DirectXⅡ

[**Chapter 1 DirectXTK(DirectX ToolKit)を使用した3Dモデル表示** 2](#_Toc508219754)

[1.1 cmoファイル 2](#_Toc508219755)

[1.2 Hands-On Lesson\_00を使用して、cmoファイルを作成してみる。 3](#_Toc508219756)

[1.3 DirectXTK(DirectX Tool Kit) 3](#_Toc508219757)

[1.3.1 cmoファイルのロード 3](#_Toc508219758)

[1.3.2 ３Ｄモデルの表示 3](#_Toc508219759)

[1.3.3 後始末 4](#_Toc508219760)

[1.4 章末テスト 4](#_Toc508219761)

[**Chapter 2　SkinModelクラス** 5](#_Toc508219762)

[2.1 SkinModelのクラス定義 5](#_Toc508219763)

[2.2 SkinModel::Init関数の定義 6](#_Toc508219764)

[2.3 SkinModel::UpdateWorldMatrixの定義 6](#_Toc508219765)

[2.4 SkinModel::Draw関数の定義 7](#_Toc508219766)

[2.5 中間テスト 7](#_Toc508219767)

[**Chapter 3 SkinModelDataクラス** 8](#_Toc508219768)

[3.1 Hands-On 毎回ロードが発生することによる問題を確認する 8](#_Toc508219769)

# **Chapter 1 DirectXTK(DirectX ToolKit)を使用した3Dモデル表示**

## **1.1 cmoファイル**

3Dモデルを表示するには、頂点バッファやインデックスバッファといった3Dモデルを表現するためのデータをGPUに送り込む必要があります。これらのデータ、は3dsMaxやMayaで作成した3Dモデルの頂点情報やポリゴン情報から作成されます。3dsMaxから出力されるfbxファイルには、下記の図のように、頂点バッファやインデックスバッファと呼ばれるデータが記述されています。



これが頂点データ

これがインデックスデータ

fbxファイルは3Dモデルを表示するのに十分な情報が含まれているのですが、3dsMaxのシーン情報が出力されているため、カメラやライトの情報なども含まれており、モデル表示のみを考えた場合、情報過多になっています。そこで、VisualStudioの機能を使って、モデル表示に必要な情報のみ抽出したモデルフォーマットがcmoファイルです。

## **1.2 Hands-On Lesson\_01を使用して、cmoファイルを作成してみる。**

Tips

|  |
| --- |
| VisualStudioでビルドすると、Resourceフォルダにcmoファイルが作成されていることを確認してみよう。 |

## **1.3 DirectXTK(DirectX Tool Kit)**

DirectXTKはMicrosoft社が提供している、3Dモデルの表示や、テクスチャの読み込みなどを簡単に行えるようにしてくれるサポートライブラリです。

DirectXTKにはcmoファイルからモデルをロードして、表示する機能があります。この機能を使って、モデルを読み込みで表示させるコードを見ていきましょう。

## **1.3.1 cmoファイルのロード**

Lesson\_01/main.cpp(144行目)

|  |
| --- |
| //エフェクトファクトリ。  DirectX::EffectFactory effectFactory(g\_graphicsEngine->GetD3DDevice());  //テクスチャがあるフォルダを設定する。  effectFactory.SetDirectory(L"Resource/modelData");  //CMOファイルからモデルを作成する関数の、CreateFromCMOを実行する。  g\_teapotModel = DirectX::Model::CreateFromCMO(  g\_graphicsEngine->GetD3DDevice(), //第一引数はD3Dデバイス。  L"Resource/modelData/teapot.cmo", //第二引数は読み込むCMOファイルのファイルパス。  effectFactory, //第三引数はエフェクトファクトリ。  false //第四引数はCullモード。今は気にしなくてよい。  ); |

DirectX::Model::CreateFromCMO関数を使用して、cmoファイルをロードして、Direct::Modelクラスのインスタンスを作成しています。

　エフェクトファクトリは、テクスチャなどのロードの時に使用されるものです。SetDirectory関数を使用して、テクスチャがあるファイルパスを指定しています。

## **1.3.2 ３Ｄモデルの表示**

Lesson\_01/main.cpp(106行目)

|  |
| --- |
| DirectX::CommonStates state(g\_graphicsEngine->GetD3DDevice());  g\_teapotModel->Draw(  g\_graphicsEngine->GetD3DDeviceContext(),//D3Dデバイスコンテキスト。  state, //レンダリングステート。今は気にしなくてよい。  g\_worldMatrix, //ワールド行列。  g\_viewMatrix, //ビュー行列。  g\_projMatrix //プロジェクション行列。  ); |

DirectX::ModelクラスのDraw関数を使用して、モデルを表示しています。引数にワールド行列、ビュー行列、プロジェクション行列を渡しています。

## **1.3.3 後始末**

プログラムが終了、もしくはモデルの表示が不要になった場合は、頂点バッファやインデックスバッファ、テクスチャなどを破棄する必要があります。当然、DirectX::Modelを使っている場合も後始末を行う必要があるのですが、今回は**スマートポインタ**と呼ばれるものを使っているため、明示的に終了処理を書く必要はありません。

DirectX::Modelの変数を定義するときに下記のような不思議な記述をしたと思います。

|  |
| --- |
| **std::unique\_ptr<DirectX::Model>** g\_teapotModel; |

これがスマートポインタを使うときの定義の仕方です。g\_teapotModelはDirectX::Modelのスマート(賢い)ポインタになります。このスマートポインタはモデルが不要になると、自動的にアドレスを解放してくれます。

## **1.4 章末テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/WEG3RvWI9dM41Ozn2>

# **Chapter 2　SkinModelクラス**

Chapter 1では、DirectX::Modelクラスを使って、複数のモデルを表示していましたが、少々冗長なコードが多くなってしまいました。そこで、DirectX::Modelをもっと簡単に使えるようにするため、薄くラップしたSkinModelクラスを作成してみようと思います。SkinModelクラス定義は下記のようになります。

## **2.1 SkinModelのクラス定義**

Lesson02/SkinModel.h

|  |
| --- |
| /\*!  \*@brief スキンモデルクラス。  \*/  class SkinModel  {  public:  /\*!  \*@brief 初期化。  \*@param[in] filePath ロードするcmoファイルのファイルパス。  \*/  void Init(const wchar\_t\* filePath);  /\*!  \*@brief モデルをワールド座標系に変換するためのワールド行列を更新する。  \*@param[in] position モデルの座標。  \*@param[in] rotation モデルの回転。  \*@param[in] scale モデルの拡大率。  \*/  void UpdateWorldMatrix(CVector3 position, CQuaternion rotation, CVector3 scale);  /\*!  \*@brief モデルを描画。  \*@param[in] viewMatrix カメラ行列。  \* ワールド座標系の3Dモデルをカメラ座標系に変換する行列です。  \*@param[in] projMatrix プロジェクション行列。  \* カメラ座標系の3Dモデルをスクリーン座標系に変換する行列です。  \*/  void Draw( CMatrix viewMatrix, CMatrix projMatrix );  private:  CMatrix 　　　　　m\_worldMatrix; //!<ワールド行列。  std::unique\_ptr<DirectX::Model> m\_modelDx; //!<DirectXTKが提供するモデルクラス。  }; |

メンバ関数として、cmoファイルをロードしてSkinModelを初期化するInit関数、ワールド行列を更新するUpdateWorldMatrix関数、モデルを描画するDraw関数が宣言されています。

では、続いて各メンバ関数の定義を見ていきましょう。

## 2**.2 SkinModel::Init関数の定義**

Lesson\_02/SkinModel.cpp(4行目)

|  |
| --- |
| void SkinModel::Init(const wchar\_t\* filePath)  {  //エフェクトファクトリ。  DirectX::EffectFactory effectFactory(g\_graphicsEngine->GetD3DDevice());  //テクスチャがあるフォルダを設定する。  effectFactory.SetDirectory(L"Resource/modelData");  //CMOファイルのロード。  //CMOファイルからモデルを作成する関数のCreateFromCMOを実行する。  m\_modelDx = DirectX::Model::CreateFromCMO(  g\_graphicsEngine->GetD3DDevice(), //第一引数はD3Dデバイス。  filePath, //第二引数は読み込むCMOファイルのファイルパス。  effectFactory, //第三引数はエフェクトファクトリ。  false //第四引数はCullモード。今は気にしなくてよい。  );  } |

引数で受け取ったファイルパスを使っ

て、cmoファイルをロードしています。

## 2**.3 SkinModel::UpdateWorldMatrixの定義**

Lesson\_02/SkinModel.cpp(18行目)

|  |
| --- |
| void SkinModel::UpdateWorldMatrix(CVector3 position, CQuaternion rotation, CVector3 scale)  {  CMatrix transMatrix, rotMatrix, scaleMatrix;  //平行移動行列を作成する。  transMatrix.MakeTranslation( position );  //回転行列を作成する。  rotMatrix.MakeRotationFromQuaternion( rotation );  //拡大行列を作成する。  scaleMatrix.MakeScaling(scale);  //ワールド行列を作成する。  //拡大×回転×平行移動の順番で乗算するように！  //順番を間違えたら結果が変わるよ。  m\_worldMatrix.Mul(scaleMatrix, rotMatrix);  m\_worldMatrix.Mul(m\_worldMatrix, transMatrix);  } |

座標、回転クォータニオン、拡大率からワールド行列を計算しています。行列の乗算順番に注意してください。

## **2.4 SkinModel::Draw関数の定義**

Lesson\_02/SkinModel.cpp(33行目)

|  |
| --- |
| void SkinModel::Draw(CMatrix viewMatrix, CMatrix projMatrix)  {  DirectX::CommonStates state(g\_graphicsEngine->GetD3DDevice());  m\_modelDx->Draw(  g\_graphicsEngine->GetD3DDeviceContext(),  state,  m\_worldMatrix,  viewMatrix,  projMatrix  );  } |

ワールド行列、ビュー行列、プロジェクション行列を渡して、3Dモデルを描画しています。DirectX::CommonStatesは今は気にしなくて構いません。

# **2.5 SkinModelクラスの使い方**

では、main.cppのモデル表示処理をSkinModelクラスを使うように書き換えていきましょう。まず、SkinModelクラスのインスタンスを定義する必要があります。

Lesson\_02/main.cpp(4行目)

|  |
| --- |
| //////////////////////////////////////////////////////////////////  // グローバル変数。  ///////////////////////////////////////////////////////////////////  HWND g\_hWnd = NULL; //ウィンドウハンドル。  GraphicsEngine\* g\_graphicsEngine = NULL; //グラフィックスエンジン。  **SkinModel g\_teapotModel; //ティーポットモデル。**  CMatrix g\_viewMatrix = CMatrix::Identity(); //ビュー行列。  CMatrix g\_projMatrix = CMatrix::Identity(); //プロジェクション行列。  CMatrix g\_worldMatrix = CMatrix::Identity(); //ワールド行列。 |

グローバル変数にg\_teapotModelが追加されていることを確認してください。

続いて、モデルの読み込み処理です。

Lesson\_02/main.cpp(153行目)

|  |
| --- |
| /////////////////////////////////////////////////////////  //ティーポットモデルの初期化。  g\_teapotModel.Init(L"Resource/modelData/teapot.cmo"); |

読み込むcmoファイルのパスを指定しています。

続いて、ワールド行列の更新です。

Lesson\_02/main.cpp(94行目)

|  |
| --- |
| void Update()  {  CVector3 pos = { 0.0f, 0.0f, 0.0f };  CQuaternion qRot = { 0.0f, 0.0f, 0.0f, 1.0f };  CVector3 scale = { 1.0f, 1.0f, 1.0f };  //座標 0, 0, 0, 回転　なし(単位クォータニオン), 拡大 等倍で  //ワールド行列を更新する。  g\_teapotModel.UpdateWorldMatrix(  pos,  qRot,  scale  );  } |

最後に描画処理です。

|  |
| --- |
| ///////////////////////////////////////////////////////////////////  // 毎フレーム呼ばれるゲームの描画処理。  ///////////////////////////////////////////////////////////////////  void Render()  {    g\_graphicsEngine->BegineRender();  ///////////////////////////////////////////  //ここからモデル表示のプログラム。  //3Dモデルを描画する。  DirectX::CommonStates state(g\_graphicsEngine->GetD3DDevice());  g\_teapotModel.Draw(  g\_viewMatrix, //ビュー行列。  g\_projMatrix //プロジェクション行列。  );  //ここまでモデル表示に関係するプログラム。  ///////////////////////////////////////////  g\_graphicsEngine->EndRender();  } |

これでteapotモデルが表示されます。

|  |
| --- |
| *Tips*  *ゲームプログラミングでは、描画と更新(座標の更新とか)は別の関数に分ける方が一般的とされます。この二つを分ける理由としては、描画とゲームの進行を分離することによって、一時停止の機能などの実装が容易になるためです。*  *例えば、一時停止の処理は下記のように実装されます。*  void Update()  {  if ( g\_isPause == true ){  return ;  }  　　//ただ横に移動していくだけ。  pos.x += 1.0f;  //ワールド行列の更新。  g\_teapotModel.UpdateWorldMatrix(  pos,  qRot,  scale  );  }  ///////////////////////////////////////////////////////////////////  // 毎フレーム呼ばれるゲームの描画処理。  ///////////////////////////////////////////////////////////////////  void Render()  {  g\_graphicsEngine->BegineRender();  ///////////////////////////////////////////  //ここからモデル表示のプログラム。  //3Dモデルを描画する。  DirectX::CommonStates state(g\_graphicsEngine->GetD3DDevice());  g\_teapotModel.Draw(  g\_viewMatrix, //ビュー行列。  g\_projMatrix //プロジェクション行列。  );  //ここまでモデル表示に関係するプログラム。  ///////////////////////////////////////////  g\_graphicsEngine->EndRender();  } |

## **2.6 章末テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/bcBKr0UbGprcvsUl2>

# **Chapter 3 SkinModelDataクラス**

SkinModelクラスはモデルのロードとモデルの表示という二つの機能を持っています。Chapter３では、SkinModelクラスをモデル描画の機能のみにして、モデル表示用のデータを保持しているSkinModelDataクラスを作成して機能分解を行います。何を言っているのかイマイチわからない、もしくは、なぜわざわざ分離するの？と思った人がいるのではないでしょうか。では、先に目的を伝えておきましょう。データと表示処理を分離する目的は次のものとなります。

**「複数体、同じモデルを表示する場合でも、モデルデータのロードを一回だけにする」**

## 3.1 Hands-On 毎回ロードが発生することによる問題を確認する

　Lesson\_03をビルドして、ゲームを実行してください。下記のような絵が表示されると思います。

弾を発射するたびに、画面がカクついていたのではないでしょうか？これは、弾のSkinModelを初期化するたびに、モデルデータのロードが発生していたからです。この問題を解決するために、SkinModelの機能を分解して、モデルデータは使いまわして、複数の同じモデルを表示できるようにしてみたいと思います。