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クラス宣言は下記のようになります。

## **2.1 SkinModelのクラス宣言**

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 SkinModelDataManagerクラス**

SkinModelクラスはInit関数が呼ばれるたびに、モデルをロードしています。しかし、同じ3Dモデルを複数描画する場合、頂点バッファ、インデックスバッファ、テクスチャといったデータは共有することが可能です。例えば、今回のデモであれば、弾丸のモデルなどは使いまわすことができます。そこで、Flyweightパターンを利用して、モデルデータを共有させることができるSkinModelDataManagerクラスを作成してみようと思います。

## **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。



キーボードのAを押すと、弾を発射することができるのですが、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。

## **3.2 Hands-On ロードのカクツキが解消されていることを確認する**

Lesson\_03\_01はFlyweightパターンを使用して、モデルデータの共有を実装しています。ビルドして、ゲームを実行してみください。カクツキが解消されていると思います。

## **3.3 Flyweightパターンとは？**

　Flyweightにあたるクラス(これは共有したいクラスなので、今回の場合DirectX::Modelとなる)は直接newなどを使用して生成するのではなく、FlyweightFactoryクラスのgetFlyweight関数を使って、インスタンスを取得する(今回の場合、SkinModelDataManagerがFlyweightFactoryクラスにとなる)。この関数は新規に作成されるインスタンスであれば、インスタンスを生成して、それを再利用できるように保存して、そのインスタンスを返す。すでに作成されているインスタンスを取得する場合は、先ほど保存した、インスタンスを返すだけとなるため、無駄なオブジェクトの生成が発生しなくなる。

　では、具体例を見ていきましょう。

## **3.3 SkinModelDataManagerクラス(FlyweightFactoryクラスにあたる)**

　共有できるモデルデータ(頂点バッファ、インデックスバッファ、テクスチャなど)を保持しているのは、DirectX::Modelクラスです。Lesson\_03\_01では、SkinModelDataManagerクラスを使うことでインスタンスを共有できるようにしています。では、クラスの実装を見ていきましょう。

Lesson\_03\_01/SkinModelDataManager.h

|  |
| --- |
| /\*!  \*@brief スキンモデルデータマネージャー。  \*/  class SkinModelDataManager  {  public:  /\*!  \*@brief モデルをロード。  \*@param[in] filePath ファイルパス。  \*/  DirectX::Model\* Load(const wchar\_t\* filePath);  private:  //ファイルパスをキー、DirectXModelのインスタンスを値とするマップ。  //辞書みたいなものです。  std::map<  std::wstring, //wstringがキー。  std::unique\_ptr<DirectX::Model> //これが値。  > m\_directXModelMap;  }; |

メンバ変数にstd::mapのインスタンスを保持しています。これが生成したインスタンスを記憶しておくコンテナです。

## **3.3.1 std::map**

少し話がそれますが、std::mapはC++の標準テンプレートクラスで、キーと値のペアを保持できるコレクションです。(なんのことかわからないですね。)

下記に簡単なサンプルを示します。

|  |
| --- |
| int main()  {  std::map<std::string, int> gakuseiMeibo; //学生の名前をキー、年齢を値に持つマップ。  gakuseiMeibo.insert({ "上野", 19 }); //上野君の年齢を19歳で登録。  gakuseiMeibo.insert({ "黒瀬", 21 }); //黒瀬君の年齢を21歳で登録。  gakuseiMeibo.insert({ "土居", 20 }); //土居君の年齢を20歳で登録。  //上野君の年齢を取得して、表示する。  int age = gakuseiMeibo.at("上野");  std::cout << age << "\n";  //黒瀬君の年齢を取得して、表示する。  age = gakuseiMeibo.at("黒瀬");  std::cout << age << "\n";  //土居君の年齢を取得して、表示する。  age = gakuseiMeibo.at("土居");  std::cout << age << "\n";  return 0;  } |

このサンプルは、学生の名前をキー、年齢を値に持つマップを使っています。gakuseiMeiboに名前を渡すことで、値の年齢を取得することができるマップです。

SkinModelDataManagerクラスでは、モデルのファイルパスをキー、DirectX::Modelのインスタンスを値として登録できるマップを使用しています。

## **3.3.2 Hads-On std::mapを使ってみよう。**

Lesson\_03\_02を使ってstd::mapを使ってみましょう。

## **3.3.3 SkinModelDataManager::Load関数(FlyweightFactory::getFlyweightにあたる)**

Lesson\_03\_01/SkinModelDataManager.cpp(7行目)

|  |
| --- |
| DirectX::Model\* SkinModelDataManager::Load(const wchar\_t\* filePath)  {  DirectX::Model\* retModel = NULL;  //マップに登録されているか調べる。  auto it = m\_directXModelMap.find(filePath);  if (it == m\_directXModelMap.end()) {  //未登録なので、新規でロードする。  //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。  auto model = 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。  );  retModel = model.get();  //新規なのでマップに登録する。  m\_directXModelMap.insert({ filePath, std::move(model) });  }  else {  //登録されているので、読み込み済みのデータを取得。  retModel = it->second.get();  }  return retModel;  } |

SkinModelDataManager::Load関数では、m\_directXModelMap.find関数にモデルのファイルパスを渡して、DirectX::Modelのインスタンスが登録されているか調べています。そして、未登録の場合、cmoファイルをロードしてDirectX::Modelのインスタンスをm\_directXModelMapに登録しています。すでに登録済みの場合、登録されているインスタンスを返しています。

　SkinModelDataManagerクラスのインスタンスはグローバル変数として定義されています。

Lesson\_03\_01/SkinModelDataManager.h(24行目)

|  |
| --- |
| //g\_skinModelDataManagerのextern宣言。  //extern宣言は実態ではないので注意！  //コンパイラにどこかにあるから使ってねと教えるもの。  extern SkinModelDataManager g\_skinModelDataManager; |

Lesson\_03\_01/SkinModelDataManager.cpp(4行目)

|  |
| --- |
| //g\_skinModelDataManagerの実態。  SkinModelDataManager g\_skinModelDataManager; |

extern宣言は変数の実態ではないことに注意してください。実態はどこかに必ず定義する必要があります。

## **3.4 SkinModelクラスを改造**

　では、SkinModelDataManagerクラスを使用して、モデルをロードするようにSkinModelクラスを改造してみましょう。

Lesson\_03\_01/SkinModel.h(3行目)

|  |
| --- |
| /\*!  \*@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。  **DirectX::Model\* m\_modelDx; //!<DirectXTKが提供するモデルクラス。**  }; |

Lesson\_03\_01/SkinModel.cpp(5行目)

|  |
| --- |
| void SkinModel::Init(const wchar\_t\* filePath)  {  //SkinModelDataManagerを使用してCMOファイルのロード。  **m\_modelDx = g\_skinModelDataManager.Load(filePath);**  } |

網掛けになっている箇所が変更点です。

## **3.4　リソースの開放**

今回の変更で、g\_skinModelDataManagerがモデルデータをキャッシュするようになったため、明示的に開放処理を呼ばないとリソースは解放されません。下記のコードを確認してください。

Lesson\_03\_01/SkinModelDataManager.h(15行目)

|  |
| --- |
| /\*!  \*@brief モデルデータを全開放。  \*/  void Release(); |

Lesson\_03\_01/SkinModelDataManager.cpp(36行目)

|  |
| --- |
| void SkinModelDataManager::Release()  {  　　//mapを空にする。  m\_directXModelMap.clear();  } |

モデルのリソースを全開放したいとき、例えばゲーム画面からタイトル画面に戻る時とか、

はSkinModelDataManagerのRelease関数を呼び出すと、リソースは解放されます。

## **3.5 章末テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSdgBkUtITKpgN_LlfZmZFO5gfK2jfzRf3qZJewAiN6Kr7KdKA/viewform?usp=sf_link>

# **Chapter 4 スキンアニメーション**

　このチャプターではスキンアニメーションについて見ていきましょう。スキンアニメーションは、人肌など、関節が折り曲がったときに切れ目が発生してしまうと不自然に見えてしまうモデルのアニメーションに使われる手法で、現在、ほぼすべての３Dモデルのアニメーションに採用されている仕組みです。

## **4.1 スケルトン**

　スキンアニメーションを行うためには、まずスケルトンと呼ばれる階層構造のデータを作成する必要があります。下記のようにスケルトンは関節(ジョイント)と呼ばれる、剛体の断片で構成されるデータ構造です。



## **4.2　親子関係**

スケルトンは親子関係を持っているボーンの集合として表現されます。下記はスケルトンのサンプルプログラムです。

|  |
| --- |
| //ボーン。  struct Bone {  　　　・  　　　・  　　　省略  　　　・  　　　・  int 　　　　 parentBoneNo; //親のボーンの番号。  std::vector<int> childrenBoneNo; //子供のボーンの番号のリスト。  };  //スケルトン。  //スケルトンはボーンの集合。  struct Skelton{  std::vector<Bone> boneArray; //ボーンの配列。  }; |

## **4.3 tksファイル**

スケルトンデータは3dsMaxやMayaなどのDCCツールで作成するのですが、そのデータをゲームで使えるような形式で出力する必要があります。tkEngine2を使ってゲームを制作していた時に、tksファイルというものを出力していたと思います。実は、あれがスケルトンデータだったのです。

## **4.4 Hands-On**

下記の手順に従って、3dsMaxでのLesson\_04\_02/Assets/modelData/unityChan.fbxを開いて、tksファイルを出力から、ゲームで使用するためにVisualStudioに追加するところまでやってみましょう。

　① 3dsMaxを起動して、ファイル/読み込み/読み込みを選択する。



　② スケルトンを出力したいfbxファイルを開く。



　③ FBXインポートのダイアログボックスが開くので、拡大率を確認してインポートする。(拡大率が1.0になっていないと、拡大されたスケルトンが出力されてしまって問題が 起きるので、1.0になっていることをシッカリと確認する！！！)



　④　ユーティリティパネルから「スクリプトを起動」を選択して、tkExporter.msを起動する。



⑤　ユーティリティパネルにtkExporterが追加されたことを確認する。



⑥ スケルトンのルートを選択して、tksファイルを出力する。

　(fbxと同名にする！！！)



## **4.5 Hands-On インゲームでtksファイルをロードして、動作を確認する。**

　4.4のHands-Onでtksファイルをゲームで使用できるようになったので、tksファイルをロードしてみましょう。では、main.cppの「Hands-On 1 skeleton.hをインクルード。」と書かれている箇所に下記のコードを追加してください。

***Lesson\_04\_02/main.cpp***

|  |
| --- |
| //Hands-On 1 skeleton.hをインクルード。  #include "graphics/Skeleton.h" |

ヘッダーファイルをインクルードしたので、main.cpp

続いて、main.cppの「Hands-On 2 tksファイルをロード。」と書かれている箇所に下記のコードを追加してください。

***Lesson\_04\_02/main.cpp***

|  |
| --- |
| //Hands-On 2 tksファイルをロード。  Skeleton skeleton;  skeleton.Load(L"Assets/modelData/unityChan.tks"); |

これでスケルトンデータができます。

続いて、「Hands-On 3 ロードできていることを確認する。」と書かれている箇所に下記のコードを追加してください。

|  |
| --- |
| //Hands-On 3 ロードできていることを確認する。  int numBone = skeleton.GetNumBones();  for (int i = 0; i < numBone; i++) {  wchar\_t boneName[256];  swprintf(boneName, L"%s\n", skeleton.GetBone(i)->GetName());  OutputDebugStringW(boneName);  } |

OutputDebugStringWは文字列をVisualStudioの出力ウィンドウに出力する関数です。ロードに成功していると、下記のように表示されるはずです。



## **4.6　中間テスト**

　下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLSfcec90UdT3iiTOKB1dLeeL-HGWhOLT87ON9wjl9cwUKKyZnw/viewform?usp=sf_link>