DirectXⅡ

[**Chapter 1 DirectXTK(DirectX ToolKit)を使用した3Dモデル表示** 3](#_Toc513346169)

[**1.1 cmoファイル** 3](#_Toc513346170)

[**1.2 Hands-On Lesson\_01を使用して、cmoファイルを作成してみる。** 4](#_Toc513346171)

[**1.3 DirectXTK(DirectX Tool Kit)** 4](#_Toc513346172)

[**1.3.1 cmoファイルのロード** 4](#_Toc513346173)

[**1.3.2 ３Ｄモデルの表示** 4](#_Toc513346174)

[**1.3.3 後始末** 5](#_Toc513346175)

[**1.4 章末テスト** 5](#_Toc513346176)

[**Chapter 2　SkinModelクラス** 6](#_Toc513346177)

[**2.1 SkinModelのクラス宣言** 6](#_Toc513346178)

[2**.2 SkinModel::Init関数の定義** 7](#_Toc513346179)

[2**.3 SkinModel::UpdateWorldMatrixの定義** 7](#_Toc513346180)

[**2.4 SkinModel::Draw関数の定義** 8](#_Toc513346181)

[**2.5 SkinModelクラスの使い方** 8](#_Toc513346182)

[**2.6 章末テスト** 10](#_Toc513346183)

[**Chapter 3 SkinModelDataManagerクラス** 11](#_Toc513346184)

[**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** 11](#_Toc513346185)

[**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** 11](#_Toc513346186)

[**3.3 Flyweightパターンとは？** 12](#_Toc513346187)

[**3.3 SkinModelDataManagerクラス(FlyweightFactoryクラスにあたる)** 12](#_Toc513346188)

[**3.3.1 std::map** 12](#_Toc513346189)

[**3.3.2 Hads-On std::mapを使ってみよう。** 13](#_Toc513346190)

[**3.3.3 SkinModelDataManager::Load関数(FlyweightFactory::getFlyweightにあたる)** 13](#_Toc513346191)

[**3.4 SkinModelクラスを改造** 14](#_Toc513346192)

[**3.4　リソースの開放** 15](#_Toc513346193)

[**3.5 章末テスト** 15](#_Toc513346194)

[**Chapter 4 スキンアニメーション** 16](#_Toc513346195)

[**4.1 スケルトン** 16](#_Toc513346196)

[**4.2　親子関係** 17](#_Toc513346197)

[**4.3 tksファイル** 17](#_Toc513346198)

[**4.4 Hands-On** 17](#_Toc513346199)

[**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。** 20](#_Toc513346200)

[**4.6 tksファイルのフォーマット** 22](#_Toc513346201)

[**4.8 Skeletonクラス** 23](#_Toc513346202)

[**4.9 Hands-On スケルトンの読み込みの処理を追いかけてみよう。** 25](#_Toc513346203)

[**4.10 バインドポーズ** 26](#_Toc513346204)

[**4.11 ローカル行列** 26](#_Toc513346205)

[**4.12 中間テスト(5分)** 26](#_Toc513346206)

[**4.13** 27](#_Toc513346207)

[**SkinModelクラスにSkeletonクラスのインスタンスを保持させて、使えるようにする** 27](#_Toc513346208)

[**4.12　GPUに送るためのボーン行列の計算（Skeleton::Update関数）** 30](#_Toc513346209)

[**4.12 Hands-On　ろくろ首 Unityちゃん！** 30](#_Toc513346210)

[**4.13 実習(時間 15分)** 30](#_Toc513346211)

[**4.14 親子関係のデータ構造はローカル座標系(親の座標系)で考えた方が簡単？** 30](#_Toc513346212)

# **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。

URL

<https://docs.google.com/forms/d/e/1FAIpQLSdgBkUtITKpgN_LlfZmZFO5gfK2jfzRf3qZ\ewAiN6Kr7KdKA/viewform?usp=sf_link>

QRコード



# **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\_01/Assets/modelData/unityChan.fbxを開いて、tksファイルを出力する方法を見ていきましょう。

　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\_01/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\_01/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 tksファイルのフォーマット**

では、Skeletonクラスの中身を見ていこうと思うのですが、その前に、バイナリデータをシンボル付きで表示することができる、バイナリエディタのtsxbinを使って、tksファイルのフォーマットを確認してみましょう。tsxbn500/ TSXBIN.exeを起動して、Lesson\_04\_01/Assets/modelData/unityChan.tksをドラッグアンドドロップしてみましょう。



すると、下記のように表示されると思います。



イメージとしては、下記のような構造体の配列のデータだと考えてください。

|  |
| --- |
| struct Bone{  int numBoneName; //ボーンの名前の長さ。  char\* boneName; //ボーンの名前。numBoneNameの大きさの配列として確保されている。  int parentBoneNo;　 //親のボーンの番号。  　 float bindPoseMatrix[4][4]; //バインドポーズの行列。  　 float invBindPoseMatrix[4][4]; //バインドポーズの逆行列。  }; |

## **4.8 Skeletonクラス**

　では、Skeletonクラスの実装について見ていきましょう。まずは、Lesson\_04\_01/skeleton.hを開いてください。

まずは、Boneクラスのメンバ変数から見ていきます。

|  |
| --- |
| /\*!  \*@brief ボーン。  \*/  class Bone {  public:  　　　・  　　　・  　　　省略  　　　・  　　　・  private:  std::wstring m\_boneName;  int m\_parentId = -1;  int m\_boneId = -1;  CMatrix m\_bindPose;  CMatrix m\_invBindPose;  CMatrix m\_localMatrix = CMatrix::Identity(); //!<ローカル行列。  CMatrix m\_worldMatrix = CMatrix::Identity(); //!<ワールド行列。  CMatrix m\_offsetLocalMatrix;  CVector3 　　m\_positoin = CVector3::Zero();  CVector3 　　m\_scale = CVector3::One();  CQuaternion m\_rotation = CQuaternion::Identity();  std::vector<Bone\*> m\_children; //!<子供。  }; |

m\_parentIdには親のボーンの番号、m\_boneIdには自分自身の番号が入ります。CMatrix型のメンバ変数として、m\_bindPose(バインドポーズ)、m\_invBindPose(バインドポーズの逆行列)、m\_localMatrix(ローカル行列)、m\_worldMatrix(ワールド行列) があります。これらの詳しい説明は後述します。m\_position、m\_scale、m\_rotationはBone::CalcWorldTRS関数を実行することで、ワールドの座標、拡大率、回転が格納されます。

続いて、Skeletonクラスのメンバ変数を見ていきましょう。

|  |
| --- |
| /\*!  \*@brief スケルトン。  \*/  class Skeleton {  public:  　　・  　　・  　　省略  　　・  　　・  private:  static const int BONE\_MAX = 512; //!<ボーンの最大数。  std::vector<Bone\*> m\_bones; //!<ボーンの配列。  std::vector<CMatrix> m\_boneMatrixs; //!<ボーン行列。  }; |

　m\_bonesはボーンの配列、m\_boneMatrixはボーン行列の配列です。続いて、SkeletonクラスのLoad関数について見ていきましょう。

|  |
| --- |
| bool Skeleton::Load(const wchar\_t\* filePath)  {  //tksファイルをオープン。  FILE\* fp = \_wfopen(filePath, L"rb");  if (fp == nullptr) {  return false;  }  //骨の数を取得。  int numBone = 0;  //tksファイルの先頭から、4バイト読み込む。  //先頭４バイトに骨の数のデータが入っている。  fread(&numBone, sizeof(numBone), 1, fp);  //骨を一本一本読み込んでいく。  for (int boneNo = 0; boneNo < numBone; boneNo++) {  int nameCount = 0;  //骨の名前の文字数を読み込む。  fread(&nameCount, 1, 1, fp);  //骨の名前を読み込めるだけのメモリを確保する。  char\* name = new char[nameCount + 1];  //骨の名前を読み込み。+1は終端文字列を読み込むため  fread(name, nameCount+1, 1, fp);  //親の骨番号を取得。  int parentNo;  fread(&parentNo, sizeof(parentNo), 1, fp);  //バインドポーズを取得。  CVector3 bindPose[4];  fread(&bindPose, sizeof(bindPose), 1, fp);  //バインドポーズの逆行列を取得。  CVector3 invBindPose[4];  fread(&invBindPose, sizeof(invBindPose), 1, fp);    //バインドポーズを表す行列を作成する。  CMatrix bindPoseMatrix;  memcpy(bindPoseMatrix.m[0], &bindPose[0], sizeof(bindPose[0]));  memcpy(bindPoseMatrix.m[1], &bindPose[1], sizeof(bindPose[1]));  memcpy(bindPoseMatrix.m[2], &bindPose[2], sizeof(bindPose[2]));  memcpy(bindPoseMatrix.m[3], &bindPose[3], sizeof(bindPose[3]));  bindPoseMatrix.m[0][3] = 0.0f;  bindPoseMatrix.m[1][3] = 0.0f;  bindPoseMatrix.m[2][3] = 0.0f;  bindPoseMatrix.m[3][3] = 1.0f;  //バインドポーズの逆行列。  CMatrix invBindPoseMatrix;  memcpy(invBindPoseMatrix.m[0], &invBindPose[0], sizeof(invBindPose[0]));  memcpy(invBindPoseMatrix.m[1], &invBindPose[1], sizeof(invBindPose[1]));  memcpy(invBindPoseMatrix.m[2], &invBindPose[2], sizeof(invBindPose[2]));  memcpy(invBindPoseMatrix.m[3], &invBindPose[3], sizeof(invBindPose[3]));  invBindPoseMatrix.m[0][3] = 0.0f;  invBindPoseMatrix.m[1][3] = 0.0f;  invBindPoseMatrix.m[2][3] = 0.0f;  invBindPoseMatrix.m[3][3] = 1.0f;    //マルチバイト文字列をワイド文字列に変換する。  wchar\_t boneName[256];  mbstowcs(boneName, name, 256);  //tksファイルからロードしたボーン情報を使って、新しいボーンを作成。  Bone\* bone = new Bone(  boneName, //ボーンの名前。  bindPoseMatrix, //バインドポーズを表す行列。  invBindPoseMatrix, //バインドポーズを表す行列の逆行列。  parentNo, //親の番号。  boneNo //ボーン番号。  );  //ちゃんと不要になったら削除。  delete[] name;  //ファイルから読み込んで、作成した骨の情報をリストにプッシュバック。  m\_bones.push\_back(bone);  }  //ファイルは開いたら、ちゃんと閉じる。。  fclose(fp);  //スケルトンの読み込みが完了したので、親の座標系での行列を求める。  for (int boneNo = 0; boneNo < m\_bones.size(); boneNo++) {  Bone\* bone = m\_bones[boneNo];  if (bone->GetParentId() != -1) {  //親がいる。  m\_bones.at(bone->GetParentId())->AddChild(bone);  //親の逆行列を取得する。h  const CMatrix& parentMatrix = m\_bones.at(bone->GetParentId())->GetInvBindPoseMatrix();  //骨のバインドポーズの行列に、親の逆行列を乗算して、親の座標系での行列を求める。  CMatrix localMatrix;  localMatrix.Mul(bone->GetBindPoseMatrix(), parentMatrix);  bone->SetLocalMatrix(localMatrix);  }  else {  //親がいない。  bone->SetLocalMatrix(bone->GetBindPoseMatrix());  }  }  //ボーン行列を確保。これは後々必要になってくる行列。  //まだ使ってないので無視してください。  m\_boneMatrixs.resize(m\_bones.size());  return true;  } |

tksファイルからボーンの数を読み込んで、その数の分だけボーンの情報を読み込んで、Boneクラスのインスタンスを生成して、ボーン配列を作成しています。ボーンの構築が終わったら、親の座標系でのローカル行列を作成しています(詳細は後述)。

## **4.9 Hands-On スケルトンの読み込みの処理を追いかけてみよう。**

　Lesson04\_01/Skeleton.cppの65行目にブレイクポイントを設置して、tksファイルの読み込みの処理を追いかけてみましょう。

## **4.10 バインドポーズ**

　バインドポーズとはアニメーションしていないときのポーズのことです。多くの場合で下記のようなポーズになります。



バインドポーズの行列が必要になる理由は後述します。

## **4.11 ローカル行列**

　ここでいうローカル行列とは、親の座標系での行列のことを指しています。親子関係を持っているデータ構造では、ワールド座標系で計算するよりも、親の座標系で計算したほうが簡単になります。(理由は4.14にて解説)

## **4.12 中間テスト(5分)**

　下記のURLのテストを行いなさい。

URL

<https://docs.google.com/forms/d/e/1FAIpQLSfDjjh18yuX1HYl0iGv2wDoppetC6Vf6zMZsZm32ALwM0ZxAQ/viewform?usp=sf_link>

QRコード



## **4.13**

## **SkinModelクラスにSkeletonクラスのインスタンスを保持させて、使えるようにする**

では、SkinModelクラスにSkeletonクラスのインスタンスを保持させて、使えるようにしてみましょう。まずは、下記のようにSkinModelクラスにメンバ変数を追加します。

Lesson\_04\_02/SkinModel.h

|  |
| --- |
| #pragma once  #include "Skeleton.h"  /\*!  \*@brief スキンモデルクラス。  \*/  class SkinModel  {  public:  　　　　・  　　　　・  　　　　・  　　　省略  　　　　・  　　　　・  　　　　・  private:  //定数バッファ。  struct SVSConstantBuffer {  CMatrix mWorld;  CMatrix mView;  CMatrix mProj;  };  ID3D11Buffer\* m\_cb = nullptr; 　　　　 //定数バッファ。  **Skeleton m\_skeleton; 　　　　 //!<スケルトン。**  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。  ID3D11SamplerState\* m\_samplerState = nullptr; //!<サンプラステート。  }; |

続いて、tksファイルの読み込み処理です。

Lesson\_04\_02/SkinModel.cpp(16行目)

|  |
| --- |
| void SkinModel::Init(const wchar\_t\* filePath)  {  **//スケルトンのデータを読み込む。**  **InitSkeleton(filePath);**  //定数バッファの作成。  InitConstantBuffer();  //サンプラステートの初期化。  InitSamplerState();  //SkinModelDataManagerを使用してCMOファイルのロード。  m\_modelDx = g\_skinModelDataManager.Load(filePath, m\_skeleton);  }  **void SkinModel::InitSkeleton(const wchar\_t\* filePath)**  **{**  **//スケルトンのデータを読み込む。**  **//cmoファイルの拡張子をtksに変更する。**  **std::wstring skeletonFilePath = filePath;**  **//文字列から.cmoファイル始まる場所を検索。**  **int pos = (int)skeletonFilePath.find(L".cmo");**  **//.cmoファイルを.tksに置き換える。**  **skeletonFilePath.replace(pos, 4, L".tks");**  **//tksファイルをロードする。**  **bool result = m\_skeleton.Load(skeletonFilePath.c\_str());**  **if ( result == false ) {**  **//スケルトンが読み込みに失敗した。**  **//アニメーションしないモデルは、スケルトンが不要なので**  **//読み込みに失敗することはあるので、ログ出力だけにしておく。**  **#ifdef \_DEBUG**  **char message[256];**  **sprintf(message, "tksファイルの読み込みに失敗しました。%s\n", message);**  **OutputDebugStringA(message);**  **#endif**  **}**  **}** |

SkinModel::Init関数の中からSkinModel::InitSkeleton関数が呼ばれています。InitSkeleton関数の中では、下記の例のように、cmoファイルのファイルパスをもとに、tksファイルのファイルパスを作成しています。

例)

Assets/modelData/UnityChan.**cmo**

↓

Assets/modelData/UnityChan.**tks**

これが、「4.4 Hands-On－⑥」でtksファイルの名前を、fbxファイルと同名にしていた理由です。

　アニメーションしないモデルはスケルトンが不要なので、ロードに失敗してもプログラムは停止せずに継続されます。読み込みに失敗した場合はVisualStudioの出力ウィンドウにログを出しています。

続いて、スケルトンの更新とGPUへの転送処理を見ていきましょう。

Lesson\_04\_02/SkinModel.cpp

|  |
| --- |
| 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);  **//スケルトンの更新。**  **m\_skeleton.Update(m\_worldMatrix);**  }  void SkinModel::Draw(CMatrix viewMatrix, CMatrix projMatrix)  {  DirectX::CommonStates state(g\_graphicsEngine->GetD3DDevice());  ID3D11DeviceContext\* d3dDeviceContext = g\_graphicsEngine->GetD3DDeviceContext();  //定数バッファの内容を更新。  SVSConstantBuffer vsCb;  vsCb.mWorld = m\_worldMatrix;  vsCb.mProj = projMatrix;  vsCb.mView = viewMatrix;  d3dDeviceContext->UpdateSubresource(m\_cb, 0, nullptr, &vsCb, 0, 0);  //定数バッファをGPUに転送。  d3dDeviceContext->VSSetConstantBuffers(0, 1, &m\_cb);  d3dDeviceContext->PSSetConstantBuffers(0, 1, &m\_cb);  //サンプラステートを設定。  d3dDeviceContext->PSSetSamplers(0, 1, &m\_samplerState);  **//ボーン行列をGPUに転送。**  **m\_skeleton.SendBoneMatrixArrayToGPU();**  //描画。  m\_modelDx->Draw(  d3dDeviceContext,  state,  m\_worldMatrix,  viewMatrix,  projMatrix  );  } |

SkinModel::UpdateWorldMatrix関数の中で、Skeleton::Update関数を呼び出しています。ここで、GPUに送るためのボーン行列が計算されます。

　SkinModel::Draw関数の中では、Skeleton::SendBoneMatrixArrayToGPU関数を呼び出して、ボーン行列をGPUに転送しています。

## **4.12　GPUに送るためのボーン行列の計算（Skeleton::Update関数）**

　ボーン行列は最終的にはGPUに転送されて、3Dモデルの頂点座標にボーン行列を乗算して、ワールド座標系に変換するために使用されます。しかし、スケルトンのボーン行列はローカル座標系(親の座標系)となっているため、このままでは使用できません。そこで、ワールド座標系のボーン行列を計算する必要があります。この計算を行っているのがSkeleton::Update関数です。

　例えば、下記のような階層構造を持っているボーンAのワールド行列は、下記の計算で求めることができます。

**あるモデルの階層構造**

|  |
| --- |
| ルートボーン  　　　　　ボーンＣ  　　　　　　　　ボーンＢ  　　　　　　　　　　　ボーンＡ |

**ボーンAのワールド行列の求め方**

|  |
| --- |
| ボーンＡのワールド行列 = 3Dモデルのワールド行列  ×ルートボーンのローカル行列  ×ボーンＣのローカル行列  ×ボーンＢのローカル行列  ×ボーンＡのローカル行列 |

　親から子供に向かって行列を乗算していくことで、ボーンのワールド行列を求めることができます。

では、この処理をもう少し分かりやすく解説します。話を簡単にするために、平行移動についてのみ考えてみようと思います。では下記の図を見て下さい。



　この図の場合、ツボのワールド座標は下記の計算で求まります。

*ツボのワールド座標 = リンクのワールド座標＋ツボのローカル座標*

*= (５,９,７)*

では、これを行列で考えてみましょう。行列は行列同士の乗算を行うことで、行列を合成することができました。上の図では、リンクのワールド行列(ワールド空間で5,4,7、平行移動する行列とツボのローカル行列(親の座標系で0,5,0、平行移動する行列)を乗算すると、

(5,９,７)平行移動する行列が求まります。いかがでしょうか、先ほど計算した、ツボのワールド座標と同じになりましたね。

ボーンのワールド行列を求めるときの式は、親から子に向かって行列を乗算することで、このような計算を行っていたのです。

では、Skeleton::Update関数を見ていきましょう。

Lesson\_04\_02/Skeleton.cpp

|  |
| --- |
| **void Skeleton::UpdateBoneWorldMatrix(Bone& bone, const CMatrix& parentMatrix)**  **{**  **//ワールド行列を計算する。**  **CMatrix mBoneWorld;**  **CMatrix localMatrix = bone.GetLocalMatrix();**  **//親の行列とローカル行列を乗算して、ワールド行列を計算する。**  **mBoneWorld.Mul(localMatrix, parentMatrix);**  **bone.SetWorldMatrix(mBoneWorld);**  **//子供のワールド行列も計算する。**  **std::vector<Bone\*>& children = bone.GetChildren();**  **for (int childNo = 0; childNo < children.size(); childNo++ ) {**  **//この骨のワールド行列をUpdateBoneWorldMatrixに渡して、**  **//さらに子供のワールド行列を計算する。**  **UpdateBoneWorldMatrix(\*children[childNo], mBoneWorld);**  **}**  **}**  　　・  　　・  　　・  　省略  　　・  　　・  　　・  void Skeleton::Update(const CMatrix& mWorld)  {  **//ここがワールド行列を計算しているところ！！！**  **for (int boneNo = 0; boneNo < m\_bones.size(); boneNo++) {**  **Bone\* bone = m\_bones[boneNo];**  **if (bone->GetParentId() != -1) {**  **continue;**  **}**  **//ルートが見つかったので、ボーンのワールド行列を計算していく。**  **UpdateBoneWorldMatrix(\*bone, mWorld);**  **}**  //ボーン行列を計算  for (int boneNo = 0; boneNo < m\_bones.size(); boneNo++) {  Bone\* bone = m\_bones[boneNo];  CMatrix mBone;  **//ワールド行列にバインドポーズの逆行列をかけたものがボーン行列？？？**  **mBone.Mul(bone->GetInvBindPoseMatrix(), bone->GetWorldMatrix());**  m\_boneMatrixs[boneNo] = mBone;  }  } |

　まず、Skeleton::Update関数の中で、スケルトンのルートとなるボーンを検索しています。そして、ルートボーンが見つかると、Skeleton::UpdateBoneWorldMatrix関数を呼び出して、ワールド行列を計算していきます。Skeleton::UpdateBoneWorldMatrix関数の中では、引数で渡されたボーンのワールド行列を計算して、さらにそのボーンに子供のボーンが存在したら、Skeleton::UpdateBoneWorldMatrix関数を再帰的に呼び出しています。これですべてのボーンのワールド行列の計算が行えます。

　・・・しかし、残念ながらまだこれではボーン行列は完成ではありません。Skeleton::Update関数でワールド行列を計算した後で、そのワールド行列にバインドポーズの逆行列を乗算したものがボーン行列になります。この詳細は後で説明します。

## **4.13 Hands-On　ろくろ首 Unityちゃん！**

では、ボーンを動かすことで、Unityちゃんのモデルが変形することを確認してみましょう。下記のコードをmain.cppに追加してください。

Lesson\_04\_02/main.cpp(116行目)

|  |
| --- |
| //Hands-On ろくろ首Unityちゃん！  //ボーンを名前で検索。  Bone\* bone = g\_unityChanModel.FindBone(L"Character1\_Neck");  //ローカルマトリクスを取得。  CMatrix localMatrix = bone->GetLocalMatrix();  if (GetAsyncKeyState(VK\_LEFT)) {  //左に1.0平行移動する行列を作成。  CMatrix mTrans;  mTrans.MakeTranslation({ 1.0f, 0.0f, 0.0f });  //平行移動する行列をローカル行列に乗算する。  localMatrix.Mul(localMatrix, mTrans);  //変更したローカル行列をボーンに反映。  bone->SetLocalMatrix(localMatrix);  } |

どうでしょうか？Unityちゃんの首が、ろくろ首のように伸びたのではないでしょうか？キャラクターのアニメーションとはこのように、ボーンを動かすことで実現されています。

## **4.14 Hands-On 親子関係のデータ構造はローカル座標系(親の座標系)で考えた方が簡単？**

　さて、4.13のハンズオンから、表題の理由が少し見えてきます。main.cppの下記のコードのコメントアウトを外して、有効にしてプログラムを実行してみてください。

|  |
| --- |
| //Hands-On UnityちゃんをY軸周りに回転させる。  //static変数は永続的な寿命を持つ。  static float angleY = 0.0f;  //Y軸周りに回転させる行列を作成。  CQuaternion qRotY;  qRotY.SetRotation(CVector3::AxisY, angleY);  angleY += 0.03f;  //X軸周りに回してから、Y軸周りで回す。  //クォータニオンは行列とは掛け算の順番が逆になるので注意！！！  qRot.Multiply(qRotY, qRot); |

そうすると、UnityちゃんがY軸周りに回転を始めたと思います。さて、では4.13のUnityちゃんの首を動かすコードを思い出してください。もし、ボーンの行列が親の座標系ではなく、ワールド座標系での行列だったらどうなっていたでしょう？ 左に動かすという平行移動行列は、Unityちゃんが現在どこを向いているのか？ということを考慮に入れて計算を行う必要が生まれたはずです。しかし、ボーンの行列はローカル座標系で考えればよいので、現在のUnityちゃんの姿勢など考える必要はありません。これがローカル座標系を使用する理由です。

## **4.15 中間テスト**

　下記のURLのテストを行いなさい。

URL

<https://docs.google.com/forms/d/e/1FAIpQLSdHnKQQdxpsGVBTtN0kf-zwhfhJKqNVE_atiBn7bPdFkeWbeQ/viewform?usp=sf_link>

QRコード



**4.16 スキンウェイト**

　スキンアニメーションはボーンを動かすことによって、そのボーンと関連付けされているモデルの頂点に対してボーンの行列を乗算することで実現されています。また、例えば1つの頂点を複数のボーンと関連付けを行うこともあります。肘の関節付近の頂点を考えてみてください。



肘付近の頂点は、前腕、上腕のボーンのどちらに関連付けを行うと、綺麗にアニメーションするのでしょうか？答えは両方のボーンに関連付けを行うと綺麗にアニメーションします。そこで、スキンウェイトという概念が出てきます。例えば、肘付近の頂点は、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、肘の頂点を変換する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、下記のような計算となります。

vDst = ( m0 ×vSrc ) × w0 + ( m1 × vSrc ) × w1

下記に疑似コードを記載します。

|  |
| --- |
| CVector3 vDst;  vDst = CMatrix::Mul( m0, vSrc ) ＊ w0 + CMatrix::Mul( m1, vSrc ) ＊ w1; |