DirectXⅡ

[**Chapter 1 DirectXTK(DirectX ToolKit)を使用した3Dモデル表示** 3](#_Toc515207520)

[**1.1 cmoファイル** 3](#_Toc515207521)

[**1.2 Hands-On Lesson\_01を使用して、cmoファイルを作成してみる。** 4](#_Toc515207522)

[**1.3 DirectXTK(DirectX Tool Kit)** 4](#_Toc515207523)

[**1.3.1 cmoファイルのロード** 4](#_Toc515207524)

[**1.3.2 ３Ｄモデルの表示** 4](#_Toc515207525)

[**1.3.3 後始末** 5](#_Toc515207526)

[**1.4 章末テスト** 5](#_Toc515207527)

[**Chapter 2　SkinModelクラス** 6](#_Toc515207528)

[**2.1 SkinModelのクラス宣言** 6](#_Toc515207529)

[2**.2 SkinModel::Init関数の定義** 7](#_Toc515207530)

[2**.3 SkinModel::UpdateWorldMatrixの定義** 7](#_Toc515207531)

[**2.4 SkinModel::Draw関数の定義** 8](#_Toc515207532)

[**2.5 SkinModelクラスの使い方** 8](#_Toc515207533)

[**2.6 章末テスト** 10](#_Toc515207534)

[**Chapter 3 SkinModelDataManagerクラス** 11](#_Toc515207535)

[**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](#_Toc515207536)

[**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](#_Toc515207537)

[**3.3 Flyweightパターンとは？** 12](#_Toc515207538)

[**3.3 SkinModelDataManagerクラス(FlyweightFactoryクラスにあたる)** 12](#_Toc515207539)

[**3.3.1 std::map** 12](#_Toc515207540)

[**3.3.2 Hads-On std::mapを使ってみよう。** 13](#_Toc515207541)

[**3.3.3 SkinModelDataManager::Load関数(FlyweightFactory::getFlyweightにあたる)** 13](#_Toc515207542)

[**3.4 SkinModelクラスを改造** 14](#_Toc515207543)

[**3.4　リソースの開放** 15](#_Toc515207544)

[**3.5 章末テスト** 15](#_Toc515207545)

[**Chapter 4 スキンアニメーション** 16](#_Toc515207546)

[**4.1 スケルトン** 16](#_Toc515207547)

[**4.2　親子関係** 17](#_Toc515207548)

[**4.3 tksファイル** 17](#_Toc515207549)

[**4.4 Hands-On** 17](#_Toc515207550)

[**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。** 21](#_Toc515207551)

[**4.6 tksファイルのフォーマット** 22](#_Toc515207552)

[**4.8 Skeletonクラス** 24](#_Toc515207553)

[**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。** 26](#_Toc515207554)

[**4.10 バインドポーズ** 26](#_Toc515207555)

[**4.11 ローカル行列** 27](#_Toc515207556)

[**4.12 中間テスト(5分)** 27](#_Toc515207557)

[**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** 27](#_Toc515207558)

[**4.14　GPUに送るためのボーン行列の計算（Skeleton::Update関数）** 30](#_Toc515207559)

[**4.14.1　なんでバインドポーズの逆行列を乗算するの？** 33](#_Toc515207560)

[**4.15 Hands-On　ろくろ首 Unityちゃん！** 36](#_Toc515207561)

[**4.16 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？** 36](#_Toc515207562)

[**4.17 中間テスト** 37](#_Toc515207563)

[**4.19　アニメーションクリップ** 40](#_Toc515207564)

[**4.20　tkaファイル** 40](#_Toc515207565)

[**4.21 Hands-On** 40](#_Toc515207566)

[**4.22 Hands-On インゲームでtkaファイルをロードして、アニメーションを流してみる。** 43](#_Toc515207567)

[**4.23 実習（20分）** 44](#_Toc515207568)

[**4.24　中間テスト** 45](#_Toc515207569)

[**4.25 tkaファイルのフォーマット** 46](#_Toc515207570)

[**4.26 AnimationClipクラス** 47](#_Toc515207571)

[**4.27 Hands-On tkaファイルのロードの処理を追いかけてみよう。** 50](#_Toc515207572)

[**4.28 AnimationクラスとAnimationPlayControllerクラス** 51](#_Toc515207573)

[**4.28.1 ローカルポーズとグローバルポーズ** 51](#_Toc515207574)

[**4.28.2 なぜ複数のポーズを合成するの？** 51](#_Toc515207575)

[**4.28.3 Animation::Update関数とAnimationPlayController::Update関数** 53](#_Toc515207576)

[**4.29 Hands-On　モーション補間を行ってみよう。** 56](#_Toc515207577)

[**4.30 最後に** 56](#_Toc515207578)

[**Chapter 5　XInput** 57](#_Toc515207579)

[**5.1 スタティックリンクライブラリのリンク** 57](#_Toc515207580)

[**5.2　Hnads-on XInputライブラリをリンクしよう。** 57](#_Toc515207581)

[**5.3　Padクラス** 58](#_Toc515207582)

[**5.4 中間テスト** 64](#_Toc515207583)

# **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Lesson\_04\_01/Assets/animData/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。

## **4.14　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。

## **4.14.1　なんでバインドポーズの逆行列を乗算するの？**

　まず、これを説明するためには逆行列について説明する必要があります。ある行列matＡから逆行列matＡ^-1を求めたら場合、matA^-1はmatAの変換を打ち消す行列となります。つまり、例えばmatAがX軸上に5移動する行列だった場合、matA^-1はＸ軸上に－5移動する行列となります。よって、下記のような計算が成り立ちます。

***matA×matA^-1 = 単位行列***

　ここで肝となるのは「行列の逆行列は、もとの行列の変換を打ち消す行列になる。」ということです。つまり、バインドポーズの逆行列は、バインド

ポーズの変換を打ち消すものだということです。

　では、なぜバインドポーズの逆行列をボーンのワールド行列に乗算するのか見ていきましょう。

　下記の図のようなボーンに関連付けされている、モデルがあると考えてみてください。



骨を動かしたら、下記のような変形を望んでいると思います。



この図は正しい変形なのですが、この変形を行うためには、3Dモデルにボーンのワールド行列をそのまま乗算したのでは、この結果にはなりません。なぜなら、バインドポーズのボーンにすでに平行移動行列が入ってしまっているため、初期ポーズの分モデルの位置がずれてしまうからです。



この図は無理矢理ボックスの位置をずらしたのですが、ボーンのワールド行列をモデルに乗算すると、このような結果になってしまいます。実はモデルに乗算する必要のある行列は、ボーンのワールド行列ではなく、**ボーンのバインドポーズからの変化量を表す行列**なのです！話を簡単にするために、平行移動成分のみで考えてみましょう。例えば、バインドポーズがX軸上に5移動するボーン行列の場合、アニメーションした結果のボーンのワールド行列がX軸上に6移動するのであれば、ボーンに関連付けされている頂点はX軸上に１動かさなければなりません。つまり、ボーンのワールド行列から、バインドポーズの行列を打ち消した行列が必要になります！これが、ボーンのワールド行列にバインドポーズの逆行列を乗算している理由です。

## **4.15 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。

## **4.16 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。

## **4.17 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



**4.18 スキンウェイト**

　4.15で見たように、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が変形します。この変形は、モデルの頂点座標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。しかし、モデルの頂点は１つのボーンとだけ関連付けを行えばよいのでしょうか？例えば、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; |

この頂点変換は、非常に重い処理です。３Ｄモデルのすべての頂点に対してこの計算を行うことを考えて下さい。今のゲームであれば、1フレームで処理する必要のあるの頂点が100万を超えることはザラにあります。そのため、この計算は頂点シェーダーと呼ばれるGPUプログラミングで行われることがほとんどです。では頂点シェーダーを見ていきましょう。

Lesson\_04\_02/shader/model.fx(100行目)

|  |
| --- |
| /\*!--------------------------------------------------------------------------------------  \* @brief スキンありモデル用の頂点シェーダー。  \* 全ての頂点に対してこのシェーダーが呼ばれる。  \* Inは1つの頂点データ。VSInputNmTxWeightsを見てみよう。  -------------------------------------------------------------------------------------- \*/  PSInput VSMainSkin( VSInputNmTxWeights In )  {  PSInput psInput = (PSInput)0;  ///////////////////////////////////////////////////  //ここからスキニングを行っている箇所。  //スキン行列を計算。  ///////////////////////////////////////////////////  float4x4 skinning = 0;  float4 pos = 0;  {    float w = 0.0f;  for (int i = 0; i < 3; i++)  {  //boneMatrixにボーン行列が設定されていて、  //In.indicesは頂点に埋め込まれた、関連しているボーンの番号。  //In.weightsは頂点に埋め込まれた、関連しているボーンのウェイト。  skinning += boneMatrix[In.Indices[i]] \* In.Weights[i];  w += In.Weights[i];  }  //最後のボーンを計算する。  skinning += boneMatrix[In.Indices[3]] \* (1.0f - w);  //頂点座標にスキン行列を乗算して、頂点をワールド空間に変換。  //mulは乗算命令。  pos = mul(skinning, In.Position);  }  psInput.Normal = normalize( mul(skinning, In.Normal) );  psInput.Tangent = normalize( mul(skinning, In.Tangent) );    pos = mul(mView, pos);  pos = mul(mProj, pos);  psInput.Position = pos;  psInput.TexCoord = In.TexCoord;  return psInput;  } |

シェーダーに関しては、DirectXⅢの授業で詳しく見ていくことになるので、ここではさらっとだけ説明します。VSMainSkinは頂点シェーダーのエントリーポイントとなる関数で、すべての頂点に対して実行されるプログラムです。このプログラムは、CPU側(C++側)からモデルの描画命令(今回であれば、SkinModel::Draw関数の中で行われている)がGPUに送られると、頂点シェーダーが実行されます。今回、頂点シェーダーで見てほしいのは112行目～130行目までのプログラムです。ここがボーン行列を使って、頂点をワールド座標系に変換しているコードです。

## **4.19　アニメーションクリップ**

　ボーンのローカル行列を動かすことで、３Ｄモデルの頂点が移動することが確認できました。アニメーションクリップとは、３Ｄモデルのボーンのフレームごとのローカル行列を配列のような形式で保持しているデータです。下記のようなデータの配列だと考えてください。

|  |
| --- |
| /\*!  \*@brief キーフレーム。  \*/  struct Keyframe {  std::uint32\_t boneNo; //!<ボーン番号。  float time; //!<時間。  CMatrix transform; //!<ローカル行列  }; |

## **4.20　tkaファイル**

　アニメーションクリップはスケルトンデータなどと同様に、3dsMaxやMayaDCCツールで作成します。そして、これも同様にゲームででたーを使えるような形式で出力する必要があります。tkEngine2で制作を行っていた時に出力していた、tkaファイルがアニメーションクリップデータだったのです。

## **4.21 Hands-On**

　下記の手順に従って、3dsMaxでLesson\_04\_01/Assets/modelData/unityChanAnim.fbxを開いて、tkaファイルを出力する方法を見ていきましょう。（※ アニメーションのfbxファイルはAssetsフォルダの中にある必要はないので注意してください。今回はここに置いてあるだけで、fbxファイルはどこにあっても構いません）

　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　tkaファイルを出力したい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、拡大されたアニメーションクリップ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！！！)



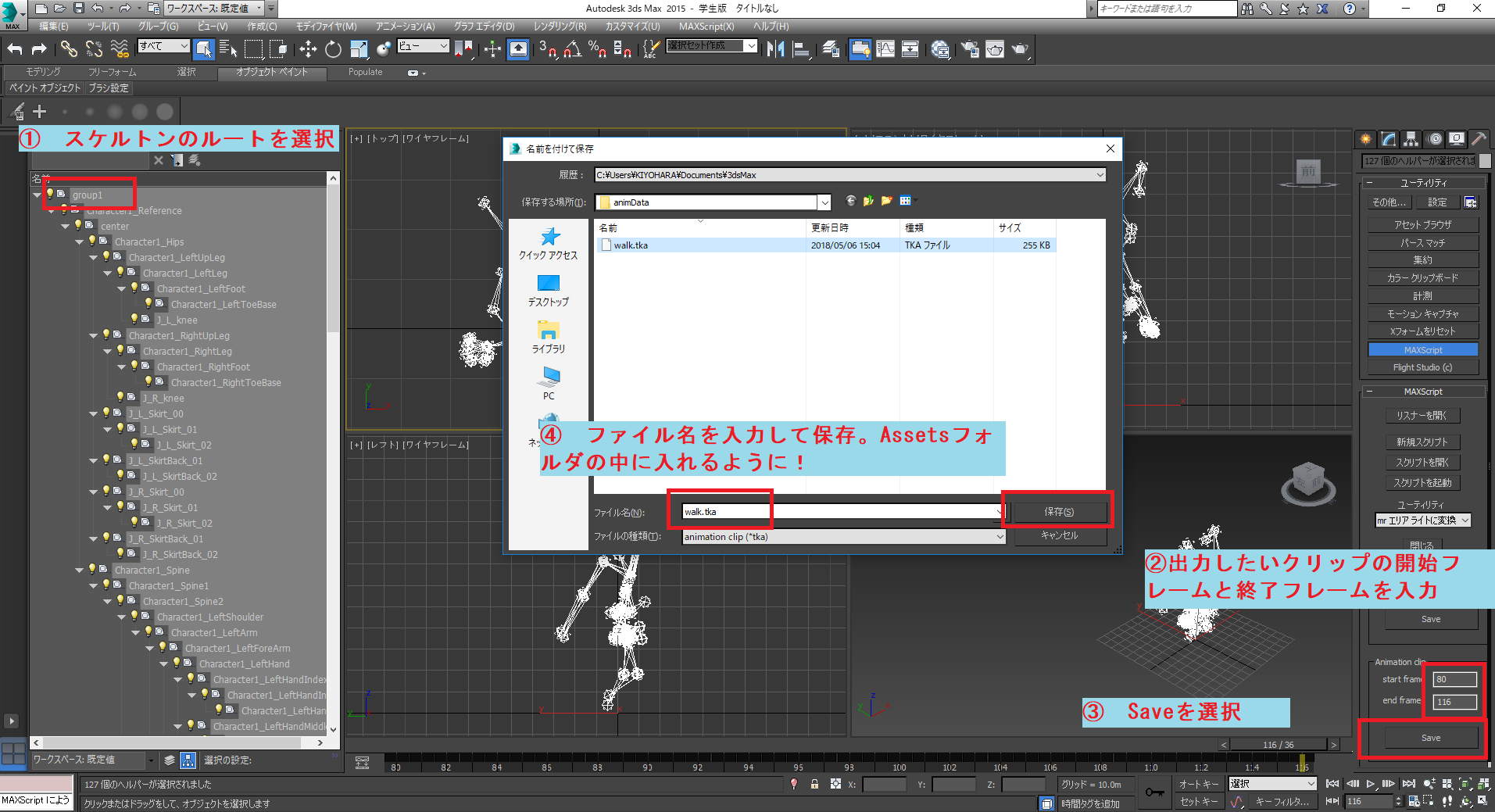
④　ユーティリティパネルから「スクリプトを起動」を選択して、tkExporter.msを起動する。



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。



⑦　出力したいアニメーションの開始フレームと終了フレームを入力して、tkaファイルを出力する。



* *参考　Unityちゃんの各種アニメーションクリップの開始フレームと終了フレーム*



## **4.22 Hands-On インゲームでtkaファイルをロードして、アニメーションを流してみる。**

　では、出力したtkaファイルを使用して、アニメーションを再生してみましょう。

Lesson\_04\_03/main.cppの「Hands-On 4.20-1 Animation.hとAnimatioinClip.hをインクルードする。」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 4.20-1 Animation.hとAnimatioinClip.hをインクルードする。  #include "graphics/animation/Animation.h"  #include "graphics/animation/AnimationClip.h" |

　続いて、main.cppの「Hands-On 4.20-2 AnimationClip配列とAnimationのグローバル変数を追加する。」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 4.20-2 AnimationClip配列とAnimationのグローバル変数を追加する。  Animation g\_unityChanAnimation; //アニメーション。  AnimationClip g\_unityChanAnimationClips[1]; //アニメーションクリップ。 |

今回はロードするアニメーションクリップは一つだけなので、アニメーションクリップの配列の要素数は１となっています。ロードするクリップの数が増える場合、配列の要素数も増やす必要があるので注意してください。

　続いて、アニメーションクリップのロードとアニメーションの初期化のコードを追加します。main.cppの「Hands-On 4.20-3 AnimationClipをロードしてとAnimationの初期化を行う。」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。

|  |
| --- |
| //tkaファイルの読み込み。  g\_unityChanAnimationClips[0].Load(L"Assets/animData/walk.tka");  g\_unityChanAnimationClips[0].SetLoopFlag(true);  //アニメーションの初期化。  g\_unityChanAnimation.Init(  g\_unityChanModel, //アニメーションを流すスキンモデル。  //これでアニメーションとスキンモデルが関連付けされる。  g\_unityChanAnimationClips, //アニメーションクリップの配列。  1 //アニメーションクリップの数。  ); |

　最後に、アニメーションの更新処理を追加します。アニメーションはAnimation::Update関数を呼び出すことによって、進行していきます。main.cppの「Hands-On 4.20-4 アニメーションを流す。」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 4.20-4 アニメーションを流す。  g\_unityChanAnimation.Update(1.0f / 30.0f); |

　ここまでのハンズオンが終了したら、F5で実行してみてください。うまくできていたら、Unityちゃんが歩きアニメーションを行っているはずです。

## **4.23 実習（20分）**

　下記の実習を行いなさい。

　①　Unityちゃんの走るアニメーションクリップを出力する。

　②　①で出力したアニメーションクリップをインゲームでロードして、再生できるよう

にする。

　③　キーボードのAが押されたら走りモーション、Ｂが押されたら歩きモーションを再

生できるようにする。キーボードの入力判定はGetAsyncKeyState関数を使用して、

下記のように行いなさい。

|  |
| --- |
| if (GetAsyncKeyState('A')) {  //ここに走りアニメーションを再生するプログラムを追加する。  } |

アニメーションは、Animation::Play関数を使用することで再生できます。

## **4.24　中間テスト**

下記のURLのテストを行いなさい。

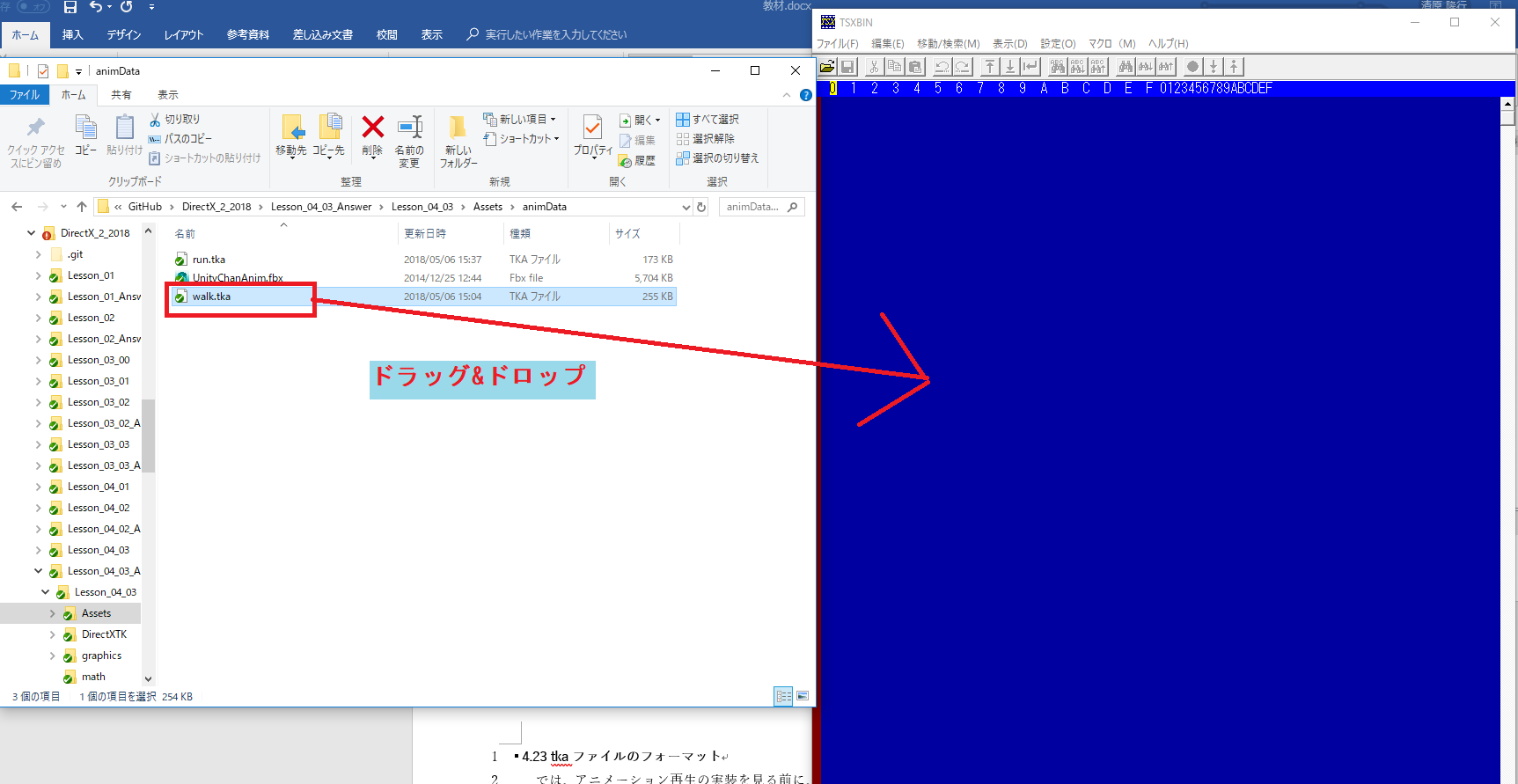
URL

<https://docs.google.com/forms/d/e/1FAIpQLSc1trXjZaOiIG7t0ESD6YWS8ziTkyam-9OrpeYCVLLPx3qpTg/viewform?usp=sf_link>

QRコード



## **4.25 tkaファイルのフォーマット**

　では、アニメーション再生の実装を見る前に、tsxbinを使用してtka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\_03\_Answer/Lesson\_04\_03/Assets/animData/ walk.tka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。



4.17でアニメーションクリップの構造体を記載していますが、あの構造体の配列のようなデータになっていると考えてください。

## **4.26 AnimationClipクラス**

　では、AnimationClipクラスについて見ていきましょう。AnimationClipクラスは1つのtkaファイルをロードして、ゲームで使える形式に変換するクラスです。まずはLesson\_04\_04/AnimationClip.hを開いて、クラス宣言を見てみましょう。

|  |
| --- |
| /\*!  \*@brief アニメーションクリップ。  \*/  #pragma once  //////////////////////////////////////////////////////////////////////  // 各種構造体  //////////////////////////////////////////////////////////////////////  /\*!  \*@brief アニメーションクリップのヘッダー。  \*/  struct AnimClipHeader {  std::uint32\_t numKey; //!<キーフレームの数。  std::uint32\_t numAnimationEvent; //!<アニメーションイベントの数。  };  /\*!  \*@brief アニメーションイベント  \*@アニメーションイベントは未対応。  \* やりたかったら自分で実装するように。  \*/  struct AnimationEventData {  float invokeTime; //!<アニメーションイベントが発生する時間(単位:秒)  std::uint32\_t eventNameLength; //!<イベント名の長さ。  };  /\*!  \*@brief キーフレーム。  \*/  struct Keyframe {  std::uint32\_t boneIndex; //!<ボーンインデックス。  float time; //!<時間。  CMatrix transform; //!<トランスフォーム。  };  /\*!  \*@brief キーフレーム。  \*/  struct KeyframeRow {  std::uint32\_t boneIndex; //!<ボーンインデックス。  float time; //!<時間。  CVector3 transform[4]; //!<トランスフォーム。  };  /\*!  \*@brief アニメーションクリップ。  \*/  class AnimationClip {  public:  //タイプ量が長ったらしくて、うざいのでstd::vector<KeyFrame\*>の別名定義。  using keyFramePtrList = std::vector<Keyframe\*>;  /\*!  \* @brief コンストラクタ  \*/  AnimationClip()  {  }  /\*!  \*@brief デストラクタ。  \*/  ~AnimationClip();  /\*!  \*@brief アニメーションクリップをロード。  \*@param[in] filePath ファイルパス。  \*/  void Load(const wchar\_t\* filePath);  /\*!  \*@brief ループする？  \*/  bool IsLoop() const  {  return m\_isLoop;  }  /\*!  \*@brief ループフラグを設定する。  \*/  void SetLoopFlag(bool flag)  {  m\_isLoop = flag;  }  /\*!  \*@brief  \*/  const std::vector<keyFramePtrList>& GetKeyFramePtrListArray() const  {  return m\_keyFramePtrListArray;  }  const keyFramePtrList& GetTopBoneKeyFrameList() const  {  return \*m\_topBoneKeyFramList;  }  private:    bool m\_isLoop = false;  **std::vector<Keyframe\*> m\_keyframes;**  **std::vector<keyFramePtrList> m\_keyFramePtrListArray;**  keyFramePtrList\* m\_topBoneKeyFramList = nullptr;  }; |

ヘッダーファイルの先頭から、各種構造体が記述されています。これらの構造体はtkaファイルの読み込みの使用される構造体です。

AnimationClipクラスの宣言は、太字になっているメンバ変数のm\_keyframesとm\_keyFramePtrListArrayのコメントに注目してください。後程、AnimationClip::Load関数を見るときに詳しく見ていきますが、m\_keyframesにはtkaファイルに入っている、キーフレームの情報が全て格納されています。m\_keyFramePtrListArrayは、そのキーフレームをボーン番号ごとに振り分けたものです。

では、AnimationClipクラスの肝となるAnimationClip::Load関数の実装を見ていきましょう。

Lesson\_04\_03\_Answer/Lesson\_04\_03/AnimationClip.cppを開いて下さい。

Lesson\_04\_04/AnimationClip.cpp(18行目)

|  |
| --- |
| void AnimationClip::Load(const wchar\_t\* filePath)  {  FILE\* fp = \_wfopen(filePath, L"rb");  if (fp == nullptr) {  #ifdef \_DEBUG  //ファイルが開けなかったときの処理。  char message[256];  sprintf(message, "AnimationClip::Load, ファイルのオープンに失敗しました。%ls\n", filePath);  MessageBox(NULL, message, "Error", MB\_OK);  //止める。  std::abort();  #endif  return;  }    //アニメーションクリップのヘッダーをロード。  AnimClipHeader header;  fread(&header, sizeof(header), 1, fp);    if (header.numAnimationEvent > 0) {  //アニメーションイベントは未対応。  //就職作品でチャレンジしてみよう。  std::abort();  }  //中身コピーするためのメモリをドカッと確保。  KeyframeRow\* keyframes = new KeyframeRow[header.numKey];  //キーフレームをドカッと読み込み。  fread(keyframes, sizeof(KeyframeRow), header.numKey, fp);  //もうデータのロードはすべて終わったので、ファイルは閉じる。  fclose(fp);  //tkaファイルのキーフレームのローカル業レは4x3行列なので  //ゲームで使用しやすいように、4x4行列に変換していく。  for (int i = 0; i < (int)header.numKey; i++) {  //ゲームで使用するKeyframeのインスタンスを生成。  Keyframe\* keyframe = new Keyframe;  //ボーン番号とか再生時間とかをコピーしていく。  keyframe->boneIndex = keyframes[i].boneIndex;  keyframe->transform = CMatrix::Identity;  keyframe->time = keyframes[i].time;  //行列はコピー。  for (int j = 0; j < 4; j++) {  keyframe->transform.m[j][0] = keyframes[i].transform[j].x;  keyframe->transform.m[j][1] = keyframes[i].transform[j].y;  keyframe->transform.m[j][2] = keyframes[i].transform[j].z;  }  //新しく作ったキーフレームを可変長配列に追加。  m\_keyframes.push\_back(keyframe);  }  //キーフレームは全部コピー終わったので、ファイルから読み込んだ分は破棄する。  delete[] keyframes;  //ボーン番号ごとにキーフレームを振り分けていく。  m\_keyFramePtrListArray.resize(MAX\_BONE);  for (Keyframe\* keyframe : m\_keyframes) {  m\_keyFramePtrListArray[keyframe->boneIndex].push\_back(keyframe);  if (m\_topBoneKeyFramList == nullptr) {  m\_topBoneKeyFramList = &m\_keyFramePtrListArray[keyframe->boneIndex];  }  }  } |

行っていることは、tkaファイルからキーフレームの情報をロードして、ゲームで使える形式に変換しているだけです。

## **4.27 Hands-On tkaファイルのロードの処理を追いかけてみよう。**

　では、下記のようにLesson\_04\_04/main.cppの167行目にブレイクポイントを設置して、AnimationClip::Load関数の処理を追いかけてみましょう。



## **4.28 AnimationクラスとAnimationPlayControllerクラス**

　最後にAnimationクラスとAnimationPlayControllerクラスについて見ていきましょう。AnimationPlayControllerクラスのインスタンスはAnimationクラスが保持しており、ゲームアプリケーション層が、このクラスを意識することはありません。この二つのクラスは下記のような役割を持っています。

　・AniationPlayControllerクラス

　　　　1つのアニメーションクリップを使用して、アニメーションを再生してローカルポ

ーズを作成する。

　・Animationクラス

　　　　複数のAnimationPlayController::Update関数を呼び出して、ローカルポーズを作

成して、ローカルポーズを合成して、グローバルポーズを作成する。

## **4.28.1 ローカルポーズとグローバルポーズ**

　ローカルポーズは、ようはスケルトンのローカル行列です。アニメーションクリップを再生して、その再生時間に対応する、ローカル行列をアニメーションクリップから取得してくることで作成されます。

　一方、グローバルポーズは複数のローカルポーズを合成したものとなります。そして、これが最終的なスケルトンのローカル行列となります。

## **4.28.2 なぜ複数のポーズを合成するの？**

　では、なぜ複数のポーズを合成するのでしょうか？これはアニメーション補完を行うためです。Lesson\_04\_04のプログラムはキーボードのAとBでアニメーションを切り替えることができますが、二つのアニメーションは滑らかに変化しておらず、即座に変化していると思います。これではクオリティの高いアニメーションを実装することは困難です。そこで、アニメーションを切り替えるときに、二つのアニメーションを補間するという考え方生まれます。二つのアニメーションを補完するということは、二つのアニメーションのローカル行列＝ローカルポーズが必要になります。そして、そのローカルポーズをブレンディング係数によって、補間してやればアニメーションは滑らかに切り替わります。

**ローカルポーズＡ　走りアニメーション**　　　　**ローカルポーズＢ　歩きアニメーション**

この二つのローカルポーズを例えば、ブレンディング係数0.5でブレンディングすると下記のようなグローバルポーズが得られます。

**グローバルポーズ　＝ ローカルポーズＡ×0.5 + ローカルポーズＢ×0.5**

****

## **4.28.3 Animation::Update関数とAnimationPlayController::Update関数**

　では、Animation::Update関数と、AnimationPlayController::Update関数を見ていきましょう。

Lesson\_04\_04/AnimationPlayController.cpp(28行目)

|  |
| --- |
| void AnimationPlayController::Update(float deltaTime, Animation\* animation)  {  if(m\_animationClip == nullptr){    return ;  }  const auto& topBoneKeyFrameList = m\_animationClip->GetTopBoneKeyFrameList();  m\_time += deltaTime;  //補完時間も進めていく。  m\_interpolateTime = min(1.0f, m\_interpolateTime + deltaTime);  while (true) {  if (m\_currentKeyFrameNo >= (int)topBoneKeyFrameList.size()) {  //終端まで行った。  if (m\_animationClip->IsLoop()) {  //ループ。  StartLoop();  }  else {  //ワンショット再生。  m\_currentKeyFrameNo--;  m\_isPlaying = false; //再生終わり。  }  break;  }  if (topBoneKeyFrameList.at(m\_currentKeyFrameNo)->time >= m\_time) {  //終わり。  break;  }  //次へ。  m\_currentKeyFrameNo++;  }  **//ボーン行列を計算していく。**  **const auto& keyFramePtrListArray = m\_animationClip->GetKeyFramePtrListArray();**  **for (const auto& keyFrameList : keyFramePtrListArray) {**  **if (keyFrameList.size() == 0) {**  **continue;**  **}**  **//現在再生中のキーフレームを取ってくる。**  **Keyframe\* keyframe = keyFrameList.at(m\_currentKeyFrameNo);**  **if (keyframe->boneIndex < m\_boneMatrix.size()) {**  **m\_boneMatrix[keyframe->boneIndex] = keyframe->transform;**  **}**  **else {**  **#ifdef \_DEBUG**  **MessageBox(NULL, "AnimationPlayController::Update : 存在しないボーンに値を書き込もうとしています。次のような原因が考えられます。\n"**  **"① tkaファイルを出力する時に、選択したルートボーンがスケルトンのルートボーンと異なっている。\n"**  **"② 異なるスケルトンのアニメーションクリップを使っている。\n"**  **"もう一度tkaファイルを出力しなおしてください。", "error", MB\_OK);**  **std::abort();**  **#endif**  **}**  **}**  } |

　太字の網掛けになっている箇所に注目してください。ここが、AnimationPlayControllerが再生中のアニメーションクリップから、ボーンのローカル行列を引っ張ってきて、ローカルポーズを作成している箇所です。メンバ変数のm\_boneMatrixにローカルポーズが設定されていています。

　続いて、Animation::Update関数を見てみましょう。

|  |
| --- |
| /\*!  \* @brief ローカルポーズの更新。  \*/  void Animation::UpdateLocalPose(float deltaTime)  {  m\_interpolateTime += deltaTime;  if (m\_interpolateTime >= 1.0f) {  //補間完了。  //現在の最終アニメーションコントローラへのインデックスが開始インデックスになる。  m\_startAnimationPlayController = GetLastAnimationControllerIndex();  m\_numAnimationPlayController = 1;  m\_interpolateTime = 1.0f;  }  **//AnimationPlayController::Update関数を実行していく。**  **for (int i = 0; i < m\_numAnimationPlayController; i++) {**  **int index = GetAnimationControllerIndex(m\_startAnimationPlayController, i );**  **m\_animationPlayController[index].Update(deltaTime, this);**  **}**  }  void Animation::UpdateGlobalPose()  {  //グローバルポーズ計算用のメモリをスタックから確保。  int numBone = m\_skeleton->GetNumBones();  CQuaternion\* qGlobalPose = (CQuaternion\*)alloca(sizeof(CQuaternion) \* numBone);  CVector3\* vGlobalPose = (CVector3\*)alloca(sizeof(CVector3) \* numBone);  for (int i = 0; i < numBone; i++) {  qGlobalPose[i] = CQuaternion::Identity;  vGlobalPose[i] = CVector3::Zero;  }  //グローバルポーズを計算していく。  **int startIndex = m\_startAnimationPlayController;**  **for (int i = 0; i < m\_numAnimationPlayController; i++) {**  **int index = GetAnimationControllerIndex(startIndex, i);**  **float intepolateRate = m\_animationPlayController[index].GetInterpolateRate();**  **const auto& localBoneMatrix = m\_animationPlayController[index].GetBoneLocalMatrix();**  **for (int boneNo = 0; boneNo < numBone; boneNo++) {**  **//平行移動の補完**  **CMatrix m = localBoneMatrix[boneNo];**  **vGlobalPose[boneNo].Lerp(**  **intepolateRate,**  **vGlobalPose[boneNo],**  **\*(CVector3\*)m.m[3]**  **);**  **//平行移動成分を削除。**  **m.m[3][0] = 0.0f;**  **m.m[3][1] = 0.0f;**  **m.m[3][2] = 0.0f;**    **//回転の補完**  **CQuaternion qBone;**  **qBone.SetRotation(m);**  **qGlobalPose[boneNo].Slerp(intepolateRate, qGlobalPose[boneNo], qBone);**  **}**  **}**  //グローバルポーズをスケルトンに反映させていく。  **for (int boneNo = 0; boneNo < numBone; boneNo++) {**  **CMatrix boneMatrix;**  **boneMatrix.MakeRotationFromQuaternion(qGlobalPose[boneNo]);**  **CMatrix transMat;**  **transMat.MakeTranslation(vGlobalPose[boneNo]);**  **boneMatrix.Mul(boneMatrix, transMat);**  **m\_skeleton->SetBoneLocalMatrix(**  **boneNo,**  **boneMatrix**  **);**  **}**    //最終アニメーション以外は補間完了していたら除去していく。  int numAnimationPlayController = m\_numAnimationPlayController;  for (int i = 1; i < m\_numAnimationPlayController; i++) {  int index = GetAnimationControllerIndex(startIndex, i);  if (m\_animationPlayController[index].GetInterpolateRate() > 0.99999f) {  //補間が終わっているのでアニメーションの開始位置を前にする。  m\_startAnimationPlayController = index;  numAnimationPlayController = m\_numAnimationPlayController - i;  }  }  m\_numAnimationPlayController = numAnimationPlayController;  }  void Animation::Update(float deltaTime)  {  if (m\_numAnimationPlayController == 0) {  return;  }  //ローカルポーズの更新をやっていく。  UpdateLocalPose(deltaTime);    //グローバルポーズを計算していく。  UpdateGlobalPose();  } |

Animation::Update関数のなかで、UpdateLocalPose関数とUpdateGlobalPose関数が実行されています。UpdateLocalPose関数の中では、AnimationPlayController::Update関数が呼び出されており、ローカルポーズが作成されていきます。UpdateGlobalPose関数では、作成されたローカルポーズを合成して、グローバルポーズを作成してスケルトンに流し込んでいっています。

## **4.29 Hands-On　モーション補間を行ってみよう。**

Lesson\_04\_04/main.cppの「Hands-On　モーション補間を行ってみよう。」を行ってみようと書かれている箇所に下記の変更を行って、モーション補間を行えるようにしてみましょう。

|  |
| --- |
| //Hands-On　モーション補間を行ってみよう。  if (GetAsyncKeyState('A')) {  //走りアニメーションを再生する。  g\_unityChanAnimation.Play(1**, 0.3f**);  }  if (GetAsyncKeyState('B')) {  //歩きアニメーションを再生する。  g\_unityChanAnimation.Play(0**, 0.3f**);  } |

網掛けになっている箇所が変更点です。0.3秒かけてアニメーションが切り替わっていきます。

## **4.30 最後に**

　とても長くて、難しいチャプターとなりました。しかし安心してください、この内容をすべて理解できているゲームプログラマはプロでも多くはありません。恐らく、これを学生が全部１から作成できたのであれば、もう就職作品の製作は終わり！と言えるくらいの難易度です。

しかし、このチャプターの内容をすべて理解できなくても、今回のサンプルで作成されたアニメーションのエンジンを使えば、就職作品のゲームでアニメーションを再生することは可能です。そして、エンジンを使うだけならば、それほど難しくはありません。

一昔前は3Dモデルのアニメーションを実装出来たらゲームプログラマとして一人前と呼ばれていました。しかし、昨今はUnity、UnrealEngine4の登場で、この手の知識がなくても簡単なゲームなら作れるようになっています。しかし、少し凝った処理を書こうとすると、すぐに基礎的な知識が必要となります。そのため、多くのゲーム会社からは、このような知識が求められています。

このアニメーションエンジンは、tkEngine2のものから、いくつか機能を削減してシンプルで分かりやすい実装にしています。今回削除された機能は下記のようなものです。

　・ルートモーション

　　　アニメーションから移動量を引っ張ってきて反映させるもの。

　・アニメーションイベント

　　　アニメーションクリップに打ち込まれたアニメーションイベントの再生。

　これらを自身の就職作品に組み込むことができれば、企業にアピールできる点となります。

　また、最新のゲームは、もっと複雑なアニメーションブレンディングを行っています。是非、それらにもチャレンジしてみてください。

# **Chapter 5　XInput**

　XInputはMicrosoftが提供している、Windows向けのゲームコントローラーからの入力値をアプリケーションで受け取れるようにするためのAPIです。コントローラーがXInputに対応している必要があり、未対応のコントローラーでは使用できません。この節ではサンプルプログラムのLesson\_05を使用して、XInputの使い方を学びましょう。

## **5.1 スタティックリンクライブラリのリンク**

　XInputはスタティックリンクライブラリとして提供されています。スタティックリンクライブラリを使用するためには、ゲームのプロジェクトにライブラリをリンクするように設定を行う必要があります。Lesson\_05をビルドしてみてください。そうすると下記のようなエラーが発生したと思います。



このエラーは、このプロジェクトでXInputの関数を使用しているのに、スタティックリンクライブラリがリンクされていないせいで、XInputGetState関数がないですよ、というエラーです。

## **5.2　Hnads-on XInputライブラリをリンクしよう。**

では、下記の動画を参考にして、ライブラリをリンクしてみましょう。

<https://youtu.be/WCoCrn9MMZU>

## **5.3　Padクラス**

　続いて、XInputから取得した情報を、ゲームで使いやすいように加工しているPadクラスについて見ていきましょう。まずはクラス宣言から見ていきましょう。Lesson\_05/HID/Pad.hを開いてください。

Lesson\_05/HID/Pad.h(13行目)

|  |
| --- |
| /\*!  \*@brief 仮想ボタン定義。  \*/  enum EnButton{  enButtonUp, //!<上。  enButtonDown, //!<下。  enButtonLeft, //!<左。  enButtonRight, //!<右。  enButtonA, //!<Aボタン。  enButtonB, //!<Bボタン。  enButtonX, //!<Xボタン。  enButtonY, //!<Yボタン。  enButtonSelect, //!<セレクトボタン。  enButtonStart, //!<スタートボタン。  enButtonRB1, //!<RB1ボタン。  enButtonRB2, //!<RB2ボタン。  enButtonRB3, //!<RB3ボタン。  enButtonLB1, //!<LB1ボタン。  enButtonLB2, //!<LB2ボタン。  enButtonLB3, //!<LB3ボタン。  enButtonNum, //!<ボタンの数。  }; |

13行目～仮想ボタンの定義として、EnButton型の定数が用意されています。ゲーム側のプログラムは、この定数を使用してパッドの入力状況を調べることになります。

Lesson\_05/HID/Pad.h(13行目)

|  |
| --- |
| /\*!  \*@brief 更新。  \*@details  \* １フレームに一度呼び出してください。  \* 複数回呼び出すと、トリガー入力が取れなくなるよ！！！  \*/  void Update(); |

パッドの入力情報を更新するためのUpdate関数です。この関数を１フレームに一度呼び出す必要があります。

Lesson\_05/HID/Pad.h(69行目)

|  |
| --- |
| /\*!  \*@brief ボタンのトリガー判定。  \*@param[in] button 調べたいボタン。enum EnButtonを参照。  \*@return trueが返ってきたらトリガー入力。  \*@code  //使用例(g\_padというPad型のグローバル変数が定義されているものとする。)  if( g\_pad.IsTrigger( enButtonA ) == true ){  //Aボタンが押されたときの処理を記述。  ・  ・  ・  }  \*@endcode  \*/  bool IsTrigger(EnButton button) const  {  return m\_trigger[button] != 0;  }  /\*!  \*@brief ボタンが押されているか判定。  \*@param[in] button 調べたいボタン。enum EnButtonを参照。  \*@code  //使用例(g\_padというPad型のグローバル変数が定義されているものとする。)  if( g\_pad.IsPress( enButtonA ) == true ){  //Aボタンが押されたときの処理を記述する。  ・  ・  ・  }  \*@endcode  \*@return trueが返ってきたら押されている。  \*/  bool IsPress(EnButton button) const  {  return m\_press[button] != 0;  } |

トリガー入力の判定を行うためのIsTrigger関数と、プレス入力(押しっぱなし)の判定を行うためのIsPress関数が定義されています。

Lesson\_05/HID/Pad.h(69行目)

|  |
| --- |
| /\*!  \*@brief 左スティックのX軸の入力量を取得。  \*@return -1.0～1.0の正規化された値を返す。  \*@code  //使用例(g\_padというPad型のグローバル変数が定義されているものとする。)  //左スティックのX軸方向の入力を取得する。  float lStickXF = g\_pad.GetLStickXF();  \*@endcode  \*/  float GetLStickXF() const  {  return m\_lStickX;  }  /\*!  \*@brief 左スティックのY軸の入力量を取得。  \*@return -1.0～1.0の正規化された値を返す。  \*@code  //使用例(g\_padというPad型のグローバル変数が定義されているものとする。)  //左スティックのX軸方向の入力を取得する。  float lStickYF = g\_pad.GetLStickYF();  \*@endcode  \*/  float GetLStickYF() const  {  return m\_lStickY;  }  /\*!  \*@brief 右スティックのX軸の入力量を取得。  \*@return -1.0～1.0の正規化された値を返す。  \*@code  //使用例(g\_padというPad型のグローバル変数が定義されているものとする。)  //右スティックのX軸方向の入力を取得する。  float rStickXF = g\_pad.GetRStickXF();  \*@endcode  \*/  float GetRStickXF() const  {  return m\_rStickX;  }  /\*!  \*@brief 右スティックのY軸の入力量を取得。  \*@return -1.0～1.0の正規化された値を返す。  \*@code  //使用例(g\_padというPad型のグローバル変数が定義されているものとする。)  //右スティックのY軸方向の入力を取得する。  float rStickXF = g\_pad.GetRStickYF();  \*@endcode  \*/  float GetRStickYF() const  {  return m\_rStickY;  } |

アナログスティックの入力を取得するための関数です。

続いて、Lesson\_05/HID/Pad.cppを開いて下さい。

Lesson\_05/HID/Pad.cpp(133行目)

|  |
| --- |
| /\*!  \*@brief パッドの入力を更新。  \*/  void Pad::Update()  {  //XInputGetState関数を使って、ゲームパッドの入力状況を取得する。  DWORD result = XInputGetState(m\_padNo, &m\_state.xInputState);  if (result != ERROR\_SUCCESS) {  //ゲームパッドが接続されていない。  return;  }  //接続されている。  m\_state.bConnected = true;  //ボタンの入力情報を更新。  UpdateButtonInput();    //アナログスティックの入力情報を更新。  UpdateAnalogStickInput();  } |

139行目のプログラムに注目してください。XInputのXInputGetState関数を使用して、ゲームコントローラーからの入力を取得しています。結果として、ERROR\_SUCCESSが返ってこなければ、ゲームパッドが接続されていないなどのエラーが発生しているので、returnで処理を終了しています。

　148行目のUpdateButtonInput関数でAボタンなどのボタンの入力情報を更新して、UpdateAnalogStickInput関数でRスティックなどのアナログスティックの入力情報を更新しています。

Lesson\_05/HID/Pad.cpp(20行目)

|  |
| --- |
| /\*!  \*@brief ボタンの入力情報を更新。  \*/  void Pad::UpdateButtonInput()  {  //XInputState.Gamepad.wButtonsに入力されたパッドの情報がビットパターンとして入っている。  //XINPUT\_GAMEPAD\_???には、対応するボタンのビットの位置に1が入っている。  //例えば、Aボタンは１３ビット目、Bボタンは１４ビット目に１が立つ場合、  //AとB両方が押されていたら、wButtonsには下記のような値になっている。  //0011 0000 0000 0000  //定数、XINPUT\_GAMEPAD\_Aは 0001 0000 0000 0000という値が入っているので、  //wButtonsとXINPUT\_GAMEPAD\_Aの論理積(&)を取ると、下記のような結果となる。  // 0011 0000 0000 0000(wButtons)  // & 0001 0000 0000 0000(XINPUT\_GAMEPAD\_A)  // = 0001 0000 0000 0000(wButtons & XINPUT\_GAMEPAD\_A の結果)  //このように、ビットパターンと論理積を取った時に、0出ない場合はそのパッドが入力されていると判断することができる。  //Aボタンが押されているか判定。  if ( ( m\_state.xInputState.Gamepad.wButtons & XINPUT\_GAMEPAD\_A ) != 0 ) {  if (m\_press[enButtonA] == 0) {  //1フレーム前は押されていないので、トリガー入力のフラグを立てる。  m\_trigger[enButtonA] = 1;  }  else {  //1フレーム前にも押されているので、トリガー入力は終わり。  m\_trigger[enButtonA] = 0;  }  //押されているフラグを立てる。  m\_press[enButtonA] = 1;  }  else {  //押されていない。  m\_trigger[enButtonA] = 0;  m\_press[enButtonA] = 0;  }  //実習 1 残りのボタンも入力をとれるようにしてみよう。  //残りのボタンのXINPUT\_GAMEPAD\_？？？は下記です。  // 十字キーの上ボタン 　 XINPUT\_GAMEPAD\_DPAD\_UP  // 十字キーの下ボタン 　 XINPUT\_GAMEPAD\_DPAD\_DOWN  // 十字キーの右ボタン XINPUT\_GAMEPAD\_DPAD\_RIGHT  // 十字キーの左ボタン XINPUT\_GAMEPAD\_DPAD\_LEFT  // Bボタン XINPUT\_GAMEPAD\_B  // Yボタン XINPUT\_GAMEPAD\_Y  // Xボタン XINPUT\_GAMEPAD\_X  // Backボタン XINPUT\_GAMEPAD\_BACK  // Startボタン XINPUT\_GAMEPAD\_START  // RB1ボタン XINPUT\_GAMEPAD\_RIGHT\_SHOULDER  // RB3ボタン(Rスティック押し込み) XINPUT\_GAMEPAD\_RIGHT\_THUMB  // LB1ボタン XINPUT\_GAMEPAD\_LEFT\_SHOULDER  // LB3ボタン(L私ティック押し込み) XINPUT\_GAMEPAD\_LEFT\_THUMB  //左トリガー(LB2ボタン)の入力判定。  if (m\_state.xInputState.Gamepad.bLeftTrigger != 0) {  m\_trigger[enButtonLB2] = 1 ^ m\_press[enButtonLB2];  m\_press[enButtonLB2] = 1;  }  else {  m\_trigger[enButtonLB2] = 0;  m\_press[enButtonLB2] = 0;  }  //右トリガー(RB2ボタン)の入力判定。  if (m\_state.xInputState.Gamepad.bRightTrigger != 0) {  m\_trigger[enButtonRB2] = 1 ^ m\_press[enButtonRB2];  m\_press[enButtonRB2] = 1;  }  else {  m\_trigger[enButtonRB2] = 0;  m\_press[enButtonRB2] = 0;  }  } |

UpdateButtonInput関数では、Aボタンなどの各種ボタンの入力を処理しています。XInputからのボタン入力情報は、各ボタンに対応するビットを立ているビットパターンとして取得できます。

　例） Aボタンが13ビット目、Bボタンが14ビット目として割り当てられているときに、

　　　A ボタンとBボタンの両方が押されていると、GamePad.wButtonsには下記のよう

なデータが設定されています。

0011 0000 0000 0000(2進数)

XInputが定義しているXINPUT\_GAMEPAD\_Aという定数には、13ビット目だけが1になっている、0001 0000 0000 0000という値が設定されています。そのため、GamePad.wButtonsとXINPUT\_GAMEPAD\_Aの論理積を計算して、その結果が0でなければ、Ａボタンが押されていると判定できます。

Lesson\_05/HID/Pad.cpp(93行目)

|  |
| --- |
| /\*!  \*@brief アナログスティックの入力情報を更新。  \*/  void Pad::UpdateAnalogStickInput()  {  //xInputState.Gamepad.sThumbLX、sThumbLYにLスティックのX方向のとY方向の入力情報がsigned short(-32,768 ～ 32,767)の範囲で入っている。  //この値がINPUT\_DEADZONE以下の場合は、入力されていないとする。スティックの遊びです。  //これがないとちょっと触れただけでキャラが動いてしまって、イライラすることとなります。  if ((m\_state.xInputState.Gamepad.sThumbLX < INPUT\_DEADZONE &&  m\_state.xInputState.Gamepad.sThumbLX > -INPUT\_DEADZONE) &&  (m\_state.xInputState.Gamepad.sThumbLY < INPUT\_DEADZONE &&  m\_state.xInputState.Gamepad.sThumbLY > -INPUT\_DEADZONE))  {  //何も入力されていないとする。  m\_state.xInputState.Gamepad.sThumbLX = 0;  m\_state.xInputState.Gamepad.sThumbLY = 0;  m\_lStickX = 0.0f;  m\_lStickY = 0.0f;  }  else {  //左スティックの入力量を-32,768 ～ 32,767から-1.0～1.0に変換する。  //なぜ変換するかというと、その方がゲームを作りやすいから。  if (m\_state.xInputState.Gamepad.sThumbLX > 0) {  //SHRT\_MAXはlimits.hに定義されている定数で、32767という値が入っています。  m\_lStickX = static\_cast<float>(m\_state.xInputState.Gamepad.sThumbLX) / SHRT\_MAX;  }  else {  //SHRT\_MINはlimits.hに定義されている定数で、-32768という値が入っています。  m\_lStickX = static\_cast<float>(m\_state.xInputState.Gamepad.sThumbLX) / -SHRT\_MIN;  }  if (m\_state.xInputState.Gamepad.sThumbLY > 0) {  m\_lStickY = static\_cast<float>(m\_state.xInputState.Gamepad.sThumbLY) / SHRT\_MAX;  }  else {  m\_lStickY = static\_cast<float>(m\_state.xInputState.Gamepad.sThumbLY) / -SHRT\_MIN;  }  }  //実習２　左スティックの入力の取得の仕方を参考にして、右スティックも入力をとれるようにしてみよう。  } |

　UpdateAnalogStickInput関数では、アナログスティックの入力をゲームで扱いやすいように加工を行っています。アナログスティックはX方向とY方向の二つの入力の値として取得することができて、XInputでは-32,768 ～ 32,767のsigned shortの値として取得できます。UpdateAnalogStickInput関数では、この値を-1.0～1.0に正規化を行っています(113行目～114行目)。

101行目～104行目のif文でINPUT\_DEADZONEより小さい入力は、無視されるようになっています。

　続いて、ゲーム側でのPadクラスの使い方を見ていきましょう。また、Padクラスのインスタンスはグローバル変数として、下記のようにPad.cppに定義されています。

Lesson\_05/Pad.cpp(9行目)

|  |
| --- |
| Pad g\_pad[Pad::CONNECT\_PAD\_MAX]; //g\_padの定義。 |

では、main.cppを見ていきましょう。

Lesson\_05/main.cpp(106行目)

|  |
| --- |
| //ゲームパッドの更新。  //foreachと型推論を使ってみる。  for (auto& pad : g\_pad) {  pad.Update();  } |

パッドの更新を行っています。C++11の機能のforeachと型推論(auto)を使用しています。

Lesson\_05/main.cpp(111行目)

|  |
| --- |
| //　Aボタンが押されたら走りアニメーションを再生する。  if ( g\_pad[0].IsTrigger(enButtonA) ) {  //走りアニメーションを再生する。  g\_unityChanAnimation.Play(1);  }  //Bボタンが押されたら歩きアニメーションを再生する。  if ( g\_pad[0].IsTrigger(enButtonB) ) {  //歩きアニメーションを再生する。  g\_unityChanAnimation.Play(0);  }  //Xボタンが押されたら走りアニメーションを補間付きで再生する。  if ( g\_pad[0].IsTrigger(enButtonX) ) {  //走りアニメーションを再生する。  g\_unityChanAnimation.Play(1, 0.3f);  }  //Yボタンが押されたら歩きアニメーションを補間付きで再生する。  if ( g\_pad[0].IsTrigger(enButtonY) ) {  //歩きアニメーションを再生する。  g\_unityChanAnimation.Play(0, 0.3f);  }  //上ボタンが押されたら、UnityちゃんのY軸方向の拡大率を増やす。  if (g\_pad[0].IsPress(enButtonUp)) {  g\_unityChanScale.y += 0.01f;  }  //下ボタンが押されたら、UnityちゃんのY軸方向の拡大率を減らす。  if (g\_pad[0].IsPress(enButtonDown)) {  g\_unityChanScale.y -= 0.01f;  }  //右ボタンが押されたら、UnityちゃんのX軸方向の拡大率を増やす。  if (g\_pad[0].IsPress(enButtonRight)) {  g\_unityChanScale.x += 0.01f;  }  //左ボタンがおされたら、UnityちゃんのX軸方向の拡大率を減らす。  if (g\_pad[0].IsPress(enButtonLeft)) {  g\_unityChanScale.x -= 0.01f;  }  //上ボタンが押されたら、UnityちゃんのY軸方向の拡大率を増やす。  if (g\_pad[0].IsPress(enButtonUp)) {  g\_unityChanScale.y += 0.01f;  }  //下ボタンがおされたら、UnityちゃんのY軸方向の拡大率を減らす。  if (g\_pad[0].IsPress(enButtonLeft)) {  g\_unityChanScale.y -= 0.01f;  } |

Padクラスのメンバ関数を使用して、パッドの入力に応じてUnityちゃんを操作しています。

## **5.4 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/1FAIpQLScCyX2aKiFOviyHPjWaZaciP3290bIKTuFiPCO6TUZJJ5WYqg/viewform?usp=sf_link>

QRコード



# **Chapter 6 カメラ**

　チャプター６ではカメラに関する復習をして、カメラ行列とプロジェクション行列を作成するための最低限のパラメーターと、更新処理だけを持っている注視点、視点などを保持している低レベルなカメラクラスを作成していきます。

## **6.1 カメラ行列とプロジェクション行列**

　3Dゲームにおける低レベルなカメラは、3Dモデルの頂点をワールド空間からカメラ空間、カメラ空間からスクリーン空間に変換する、カメラ行列とプロジェクション行列の作成を行えることが求められます。高レベルなカメラの処理となる、バネカメラ、TPSカメラなどは、この低レベルなカメラクラスのインスタンスの注視点や視点などを操作することで実装されています。

　低レベルなカメラクラスに最低限必要な仕様は下記のようなものとなります。

　① カメラ行列を作成することができる。

　② パースペクティブ射影変換行列を作成することができる。

　③ 平行投影変換行列を作成することができる。

　この3点が目的となります。また、２と３はともにプロジェクション行列を呼ばれるもので、遠近感がある絵を描画したい場合は２、遠近感がない絵を描画したい場合は３を使用します。一般的には、３Dシーンではパースペクティブ射影行列、HUDなどの2Dを描画する場合は平行投影行列を作成します。

## **6.1.1 カメラ行列**

　カメラ行列を作成するためには、下記の3点のパラメーターが必要です。

　①　注視点

　②　視点

　③　カメラの上方向(多くの場合で{ 0, 1, 0}でよい。)

カメラ行列の詳細はDirectXⅠの11.1を参照してください。

## **6.1.2 透視投影行列（プロジェクション行列）**

　透視投影行列を作成するためには、下記の4点のパラメーターが必要です。

　①　画角

　②　アスペクト比

　③　近平面

　④　遠平面

透視投影行列の詳細はDirectXⅠの11.2を参照してください。

## **6.1.3 平行投影行列（プロジェクション行列）**

　平行投影行列を作成するためには、下記の4つのパラメーターが必要です。

①　ビューボリュームの幅

②　ビューボリュームの高さ

　③　近平面

　④　遠平面

平行投影は遠近感がない絵が作成されるので、2D画像の表示や、3Dモデルを使って、２Dゲームを作成する場合などに使用されます。

　例)New スーパーマリオブラザーズ

<https://www.youtube.com/watch?v=pVah6vNK43g>

平行投影と透視変換の違いは下記の図のようになります。



ビューボリュームの高さ

ビューボリュームの幅

ビューボリュームの幅と高さは、投影面に移す範囲になります。幅と高さを大きくすれば、広範囲を移すことができます。ビューボリュームの幅と高さの比率は、スクリーンのアスペクト比と合わせる必要がある場合がほとんどです。

## **6.2 カメラクラスの実装**

では、カメラクラスの実装を見ていきましょう。Lesson\_07を開いてCamera.hを開いてください。

Lesson\_07/graphics/Camera.h

|  |
| --- |
| #pragma once  class Camera  {  public:  /\*!  \* @brief カメラ行列、プロジェクション行列の更新。  \*@details  \* 現在設定されている、注視点、視点、画角などの情報をもとに  \* カメラ行列とプロジェクション行列を更新します。  \* この関数を呼ばないと、カメラは動かないので注意が必要です。  \*/  void Update();  　　・  　　・  　　・  　　省略  　　・  　　・  　　・  private:  CMatrix m\_viewMatrix = CMatrix::Identity(); //ビュー行列。  CMatrix m\_projMatrix = CMatrix::Identity(); //プロジェクション行列。  CVector3 m\_target = CVector3::Zero(); //注視点。  CVector3 m\_position = CVector3::Zero(); //視点。  CVector3 m\_up = CVector3::Up(); //上方向。  float m\_viewAngle = CMath::DegToRad(60.0f); //画角。  float m\_far = 10000.0f; //遠い平面までの距離。  float m\_near = 1.0f; //近平面までの距離。  };  extern Camera g\_camera3D; //!<3Dカメラ。 |

Cameraクラスには、カメラ行列を作成するためのパラメーターを設定するためのsetterと、パラメーターを参照するためのgetterが用意されています。Camera::Update関数は、これらのパラメーターを元に、カメラ行列とプロジェクション行列を作成します。カメラ行列と透視変換行列を作成するためのパラメーターをメンバ変数として保持しています。

　では、続いて、Camera.cppを開いてください。

Lesson\_07/graphics/Camera.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Camera.h"  Camera g\_camera3D; //3Dカメラ。  void Camera::Update()  {  //ビュー行列を計算。  m\_viewMatrix.MakeLookAt(  m\_position,  m\_target,  m\_up  );  //プロジェクション行列を計算。  m\_projMatrix.MakeProjectionMatrix(  m\_viewAngle, //画角。  FRAME\_BUFFER\_W / FRAME\_BUFFER\_H, //アスペクト比。  m\_near,  m\_far  );  } |

Camera::Update関数の中で、CMatrix::MakeLookAt関数とCMatrix::MakeProjectionMatrix関数を使用して、カメラ行列と透視変換行列を作成しています。4行目には、Cameraクラスのインスタンスとしてg\_camera3Dというグローバル変数が用意されています。

　では、カメラを使う側のコードを見ていきましょう。

Lesson\_07/main.cpp(14行目)

|  |
| --- |
| //カメラを初期化。  g\_camera3D.SetPosition({ 0.0f, 100.0f, 300.0f });  g\_camera3D.SetTarget({ 0.0f, 100.0f, 0.0f }); |

カメラの視点と注視点を設定して、初期化を行っています

Lesson\_07/main.cpp(41行目)

|  |
| --- |
| g\_camera3D.Update(); |

main.cppの41行目では、カメラのUpdate関数を呼び出して、カメラ行列とプロジェクション行列の更新を行っています。今回のサンプルでは、カメラは動いていないので、

毎フレームCamera::Update関数を呼び出す必要はないのですが、カメラが動くゲームの場合は毎フレーム呼び出す必要があります。

続いて、プレイヤーを描画するコードを見てみましょう。

Lesson\_07/Player.cpp

|  |
| --- |
| void Player::Draw()  {  m\_model.Draw(  g\_camera3D.GetViewMatrix(),  g\_camera3D.GetProjectionMatrix()  );  } |

SkinModel::Draw関数の引数に、ビュー行列とプロジェクション行列を渡しています。

## **6.3 実習課題**

　Lesson\_07のCameraクラスを改造して、平行投影行列を作成できるようにして、Unityちゃんの描画を平行投影で行えるようにしてください。

　ヒント１

　　カメラクラスにビューボリュームの幅と高さを表す、float型のメンバ変数を追加する

必要があります。

　ヒント２

　　平行投影行列はCMatrix::MakeOrthoProjectionMatrix関数を使用すると作成すること

ができます。

## **6.4 中間テスト**

　下記のURLのテストを行いなさい。

# **Chapter 7 物理エンジンを活用した衝突判定**

このチャプターではBulletPhysicsという物理エンジンを活用した衝突判定について見ていきます。

## **7.1 物理エンジン**

3Dゲームにおいて衝突判定は非常に難しい分野のプログラムだったのですが、物理エンジンの登場によって、その難易度は大幅に下がりました。物理エンジンで有名なものはHavokPhysics、PhysXが挙げられます。ただし、基本的な使い方はどのエンジンも大きくは変わりません。この節では、衝突判定の話に行く前に物理エンジンとはどういうものなのか簡単に説明します。

　物理エンジンにはいくつか機能的な分類があります。軟体物理、破壊物理などなど。ここでは剛体物理エンジンについて見ていきます。剛体とは形の変わらない物体のことを指します。剛体物理エンジンを使うためには物理ワールドを作成する必要があります。そして、ユーザーは剛体を生成して、そのインスタンスを物理ワールドに登録することで物理シミュレーションを行うことができます。下記は簡単な剛体シミュレーションの行い方です。

1. *物理ワールドの初期化*

|  |
| --- |
| //コリジョンコンフィギュレーションを作成。  collisionConfig = new btDefaultCollisionConfiguration();  //コリジョンディスパッチャーを作成。  collisionDispatcher = new btCollisionDispatcher(collisionConfig);  //ブロードフェーズの作成。  overlappingPairCache = new btDbvtBroadphase();  //コンストレイントソルバーの作成。  constraintSolver = new btSequentialImpulseConstraintSolver;  //物理ワールドの作成。  dynamicWorld = new btDiscreteDynamicsWorld(  collisionDispatcher,  overlappingPairCache,  constraintSolver,  collisionConfig  );  //重力を設定。  dynamicWorld->setGravity(btVector3(0, -10, 0)); |

続いて、剛体を生成して物理ワールドに登録するやり方です。

1. *剛体の生成と登録*

|  |
| --- |
| //剛体の形状を表すオブジェクトを作成する。  //このサンプルでは半径0.5の球体コリジョンを作成。  btSphereShape shape = new btSphereShape(0.5f);  //座標と回転を使って剛体の初期位置を決める。  btTransform transform;  transform.setIdentity();  transform.setOrigin(btVector3(rbInfo.pos.x, rbInfo.pos.y, rbInfo.pos.z));  transform.setRotation(btQuaternion(rbInfo.rot.x, rbInfo.rot.y, rbInfo.rot.z, rbInfo.rot.w));  //モーションステートの生成。  myMotionState = new btDefaultMotionState;  //質量、モーションステート、コリジョン形状から剛体情報を用意して、剛体を作成。  btRigidBody::btRigidBodyConstructionInfo btRbInfo(rbInfo.mass, myMotionState, shape, btVector3(0, 0, 0));  //剛体を作成。  rigidBody = new btRigidBody(btRbInfo);  //作成した剛体を物理ワールドに追加。  dynamicWorld-> addRigidBody(rigidBody); |

さて、初期化と剛体の登録までは終わりました。次は物理シミュレーションを行う方法を見ていきましょう。下記のコードを毎フレーム呼び出してください。

1. 物理シミュレーションの実行

|  |
| --- |
| //1フレームの経過時間(単位は秒)を引数に渡して、物理シミュレーションを行う。  dynamicWorld->stepSimulation(1.0f / 60.0f ); |

では最後に物理シミュレーションを行った結果をゲームに引き渡すコードを見てみましょう。

1. *ゲームオブジェクトへの引渡し*

|  |
| --- |
| btTransform trans;  myMotionState->getWorldTransform(trans);  //ゲームのオブジェクトに位置と回転を引き渡す。  position = \*(D3DXVECTOR3\*)&trans.getOrigin();  rotation = \*(D3DXQUATERNION\*)&trans. getRotation(); |

簡単な剛体物理シミュレーションの使い方を説明しましたが、このチャプターの趣旨は物理エンジンを活用した衝突判定です。なので、実はこのチャプターでは必要なのは下記の３つの処理になります。

1. *物理ワールドの初期化*
2. 剛体の生成と登録
3. 物理シミュレーションの実行

## **7.2 コリジョン**

　コリジョンは衝突判定で用いられる用語で、コリジョン＝衝突という意味になります。物体同士のコリジョンを調べるためにはその物体の形状を決める必要があります。形状とは例えば下記のようなものがあります。

　・球体(SphereCollider)

　・箱(BoxCollider)

　・カプセル(CapsuleCollider)

　・3Dモデルと同じ形状をもったメッシュ(MeshCollider)

このようなコリジョンの形状のことをゲームエンジンのUnityではCollider(コライダー)と呼称しており、この教材でもその呼称を使用します。

多くのゲームではColliderに複雑な形状をもったMeshColliderを使うことはほとんどなく、SphereCollider、BoxCollider、CapsuleColliderを組み合わせて複雑な形状のColliderを作成しています。その理由はモデルから生成したMeshColliderは非常に詳細な形状を持っており、衝突判定の処理に時間がかかってしまいます。また、複雑な形状のColliderとの衝突判定は往々にして当たり抜けが発生しやすくなります。MeshColliderを使うとしても、ポリゴン数を減らした別データを用意することがほとんどです。

しかし、MeshCollider以外は実装するのが非常に簡単ですので、今回はモデルから自動的に生成することが可能なMeshColliderの作り方を見ていきます。

## **7.3 MeshCollider**

Lesson\_08/physics/ MeshCollider.cppを開いてください。このソースファイルの27行目から始まっているCreateFromSkinModel関数がMeshColliderを作成しているコートです。モデルを元に作成を行っているため、第一引数にSkinModelのインスタンスの参照を受け取っています。第二引数のoffsetMatrixはモデルの頂点をワールド座標系に変換させるための行列です。

MeshCollider.cpp(26行目)

|  |
| --- |
| model.FindMesh(**[&](const auto& mesh){** |

このSkinModel::FindMesh関数を使って、モデルのメッシュを検索しています。この関数は引数として、関数オブジェクトを受け取ります。網掛けになっている箇所はラムダ式と呼ばれるもので、無名の関数です。モデルを検索して、メッシュが見つかると、この関数が実行されます。autoはC++11から追加された型推論の機能で、文脈に合わせて、コンパイラが適切な型に変換してくれます。(便利！)

頂点バッファの作成　MeshCollider.cpp(29行目)

|  |
| --- |
| //頂点バッファを作成。  {  D3D11\_MAPPED\_SUBRESOURCE subresource;  HRESULT hr = deviceContext->Map(mesh->vertexBuffer.Get(), 0, D3D11\_MAP\_WRITE\_NO\_OVERWRITE, 0, &subresource);  if (FAILED(hr)) {  return;  }  D3D11\_BUFFER\_DESC bufferDesc;  mesh->vertexBuffer->GetDesc(&bufferDesc);  int vertexCount = bufferDesc.ByteWidth / mesh->vertexStride;  char\* pData = reinterpret\_cast<char\*>(subresource.pData);  VertexBufferPtr vertexBuffer = std::make\_unique<VertexBuffer>();  CVector3 pos;  for (int i = 0; i < vertexCount; i++) {  pos = \*reinterpret\_cast<CVector3\*>(pData);  //バイアスをかける。  mBias.Mul(pos);  vertexBuffer->push\_back(pos);  //次の頂点へ。  pData += mesh->vertexStride;  }  //頂点バッファをアンロック  deviceContext->Unmap(mesh->vertexBuffer.Get(), 0);  m\_vertexBufferArray.push\_back(std::move(vertexBuffer));  } |

29行目から53行目まではメッシュコライダーを作成するための頂点バッファを作成するコードです。

インデックスバッファの作成　MeshCollider.cpp(54行目)

|  |
| --- |
| //インデックスバッファを作成。  {  D3D11\_MAPPED\_SUBRESOURCE subresource;  //インデックスバッファをロック。  HRESULT hr = deviceContext->Map(mesh->indexBuffer.Get(), 0, D3D11\_MAP\_WRITE\_NO\_OVERWRITE, 0, &subresource);  if (FAILED(hr)) {  return;  }  D3D11\_BUFFER\_DESC bufferDesc;  mesh->indexBuffer->GetDesc(&bufferDesc);  //@todo cmoファイルはインデックスバッファのサイズは2byte固定。  IndexBufferPtr indexBuffer = std::make\_unique<IndexBuffer>();  int stride = 2;  int indexCount = bufferDesc.ByteWidth / stride;  unsigned short\* pIndex = reinterpret\_cast<unsigned short\*>(subresource.pData);  for (int i = 0; i < indexCount; i++) {  indexBuffer->push\_back(pIndex[i]);  }  //インデックスバッファをアンロック。  deviceContext->Unmap(mesh->indexBuffer.Get(), 0);  m\_indexBufferArray.push\_back(std::move(indexBuffer));  } |

54行目から75行目まではインデックスバッファを作成するためのコードです。

MeshCollider.cpp(77行目) btIndexMeshの生成

|  |
| --- |
| //インデックスメッシュを作成。  btIndexedMesh indexedMesh;  IndexBuffer\* ib = m\_indexBufferArray.back().get();  VertexBuffer\* vb = m\_vertexBufferArray.back().get();  indexedMesh.m\_numTriangles = (int)ib->size() / 3;  indexedMesh.m\_triangleIndexBase = (unsigned char\*)(&ib->front());  indexedMesh.m\_triangleIndexStride = 12;  indexedMesh.m\_numVertices = (int)vb->size();  indexedMesh.m\_vertexBase = (unsigned char\*)(&vb->front());  indexedMesh.m\_vertexStride = sizeof(CVector3);  m\_stridingMeshInterface->addIndexedMesh(indexedMesh); |

そして最後の99行目から112行目までのコードで、作成した頂点バッファ、インデックスバッファを用いてBulletPhysicsのbtIndexMeshを生成して、btTriangleMeshShapeを生成しています。これでMeshColliderの完成です。

## **7.4 RigidBodyの登録**

　MeshColliderを作成しただけではあたり判定を取ることはできません。物理エンジンを使って、あたり判定を行うためには剛体、RigidBodyを作成する必要があります。ここではbulletPhysicsのbtRigidBodyを直接触らずに、ラッパークラス(\*[[1]](#footnote-1))のRigidBodyを使います。/ Lesson\_08/physics/PhysicsStaticObject.cppの20行目から27行目にかけてサンプルとなるコードがあります。

|  |
| --- |
| //メッシュコライダーを作成。  m\_meshCollider.CreateFromSkinModel(skinModel, nullptr);  //剛体を作成、  RigidBodyInfo rbInfo;  rbInfo.collider = &m\_meshCollider; //剛体に形状(コライダー)を設定する。  rbInfo.mass = 0.0f;  rbInfo.pos = pos;  rbInfo.rot = rot;  m\_rigidBody.Create(rbInfo);  //剛体を物理ワールドに追加する。  g\_physics.AddRigidBody(m\_rigidBody); |

これで剛体を物理ワールドに登録することができました。剛体の質量を設定するときに0.0を設定していることに注意してください。質量を0.0にすることで、その剛体は静的になり、他の剛体と衝突しても動くことがなくなります。地面や、建物などのような動かない剛体は0.0を設定してください。

## **7.5 PhysicsStaticObject**

今回のサンプルでは、よく使用する静的オブジェクト(動かないオブジェクト)の作成を簡単にするためのFaçadeパターンのクラスのPhysicsStaticObjectクラスを用意しています。このクラスはMeshColliderのインスタンスとRigidBodyのインスタンスをメンバ変数として保持しています。

|  |
| --- |
| /\*!  \* @brief 静的物理オブジェクト  \*/  class PhysicsStaticObject{  public:  /\*!  \* @brief コンストラクタ。  \*/  PhysicsStaticObject();  /\*!  \* @brief デストラクタ。  \*/  ~PhysicsStaticObject();  /\*!  \* @brief メッシュの静的オブジェクトを作成。  \*@param[in] skinModel スキンモデル。  \*@param[in] pos 座標。  \*@param[in] rot 回転。  \*/  void CreateMeshObject(SkinModel& skinModel, CVector3 pos, CQuaternion rot);    private:  MeshCollider m\_meshCollider; //!<メッシュコライダー。  RigidBody m\_rigidBody; //!<剛体。  }; |

　背景などの静的オブジェクトはこのクラスを利用するとよいでしょう。

これで背景などの静的オブジェクトを物理ワールドに登録することもできるようになりました。単純な剛体同士の衝突判定であれば、全て物理エンジンが計算を行ってくれるようになりました。しかし、ユーザーが操作するゲームキャラクターの挙動を全て物理シミュレーション任せにすることはできません。なぜなら、ゲームキャラクターが移動する方向と速度の決定はユーザーが決定します、このときキャラクターの挙動は物理的に正しくない挙動をすることが多々あります (物理的に正しい挙動は操作性が悪く感じることがあります) 。また、物体に衝突した時のキャラクターの挙動もゲームの状況によって異なることもあるでしょう。反発する壁とかありそうですよね。そのため、ゲームキャラクターのコリジョン処理はアプリケーション側で実装したほうが都合の良い場合が多々あります。

次の節からは、キャラクターのコリジョン処理を実装するために必要な知識のコリジョン検出とコリジョン解決を見ていきましょう。

## **7.6コリジョン検出とコリジョン解決**

　キャラクターコントローラーの話をする前に、コリジョン検出とコリジョン解決の話をします。コリジョン処理には大きく分類するとコリジョン検出とコリジョン解決に分けることができます。コリジョン検出はCollider同士が衝突するかどうか、衝突する場合は衝突点を求めたり、衝突する面の法線を求めたりする処理のことをいいます。コリジョン解決は衝突する場合のめり込みの補正の処理を差します。

コリジョン検出は非常に難しいプログラムを実装することになるのですが、その難しい部分はBulletPhysicsが実装を行ってくれているので、ありがたくそれを使いましょう。下記はBulletPhysicsのレイキャストを使用した衝突判定のサンプルコードになります。

|  |
| --- |
| btTransform start;  start.setOrigin(btVctor3(10.0f, 10.0f, 10.0f)); //レイの始点を作成。  btTransform end;  end.setOrigin(btVector3(20.0f, 10.0f, 10.0f)); //レイの終点を作成。  ClosestConvexResultCallback cb //衝突したときに呼ばれるコールバックの関数オブジェクト。  dynamicWorld-> ConvexSweepTest(  shape, //レイを飛ばすオブジェクトの形状  start, 　　　　　　　　　　　　　　　　　//レイの始点  end, //レイの終点  cb 　　　　　　　　　　　　　　　　 //コールバックオブジェクト  );  if(cb.hasHit()){  //衝突した  } |

ConvexSweepTest関数にオブジェクトの形状を渡していることに注意してください。この形状はbtCapsuleShape、btBoxShape、btSphereShapeのインスタンスです。このレイキャストは下記の図のようなことをしていると考えて下さい。

レイ

始点

終点

上の図の始点と終点の間に物理ワールドに登録されている剛体がある場合は、衝突しているという結果になります。

　続いてコリジョン解決を見ていきましょう。コリジョン解決とはコリジョン検出を行った結果、衝突していることが分かった場合に、その衝突を解決するものです。例えば下記のようなケースを考えてみて下さい。

物体がこのような位置関係になる場合にコリジョン検出を行うと、衝突しているという結果が返ってきます。多くのゲームではこのようなときは下記のような位置関係に補正するプログラムが実行されていると思います。

めり込んだ分を押し戻す

青いオブジェクトが赤い矢印の方向に押し戻されています。これがコリジョン解決です。

## **7.7 キャラクターコントローラー**

　最近の物理エンジンには前節で勉強した典型的なゲームキャラクターのコリジョン検出とコリジョン解決を行ってくれるキャラクターコントローラーというものがあります。しかし前述したように、キャラクターの挙動というのはゲームによって細かい調整が行われます。そのため、キャラクターコントローラーが行っている処理を理解するということは無駄ではありません。この節ではサンプルプログラムのCharacterControllerクラスを例にして、処理を説明していきます。

Lesson\_08/physics/CharacterController.cppの123行目から282行目までのプログラムを見てみて下さい。この関数がコリジョン検出、コリジョン解決を行っている関数です。

*移動速度に従って、次の移動先を計算しているコード*

|  |
| --- |
| //次の移動先となる座標を計算する。  CVector3 nextPosition = m\_position;  //速度からこのフレームでの移動量を求める。オイラー積分。  CVector3 addPos = moveSpeed;  addPos \*= (deltaTime;  nextPosition += addPos; |

130行目～135行目までのコードは次の移動先を計算しているコードになります。

移動速度の単位はm/secになっているために、移動速度に1フレームの経過時間(1/60秒)を乗算しています。そして移動量が求まったら、それをnextPositionに加算しています。

*XZ平面でのコリジョン検出*

|  |
| --- |
| //現在の座標から次の移動先へ向かうベクトルを求める。  CVector3 addPos;  addPos.Subtract(nextPosition, m\_position);  CVector3 addPosXZ = addPos;  addPosXZ.y = 0.0f;  if (addPosXZ.Length() < FLT\_EPSILON) {  //XZ平面で動きがないので調べる必要なし。  //FLT\_EPSILONは1より大きい、最小の値との差分を表す定数。  //とても小さい値のことです。  break;  }  //カプセルコライダーの中心座標 + 高さ\*0.1の座標をposTmpに求める。  CVector3 posTmp = m\_position;  posTmp.y += m\_height \* 0.5f + m\_radius + m\_height \* 0.1f;  //レイを作成。  btTransform start, end;  start.setIdentity();  end.setIdentity();  //始点はカプセルコライダーの中心座標 + 0.2の座標をposTmpに求める。  start.setOrigin(btVector3(posTmp.x, posTmp.y, posTmp.z));  //終点は次の移動先。XZ平面での衝突を調べるので、yはposTmp.yを設定する。  end.setOrigin(btVector3(nextPosition.x, posTmp.y, nextPosition.z));  SweepResultWall callback;  callback.me = m\_rigidBody.GetBody();  callback.startPos = posTmp;  //衝突検出。  g\_physics.ConvexSweepTest((const btConvexShape\*)m\_collider.GetBody(), start, end, callback); |

続いて、移動先が求まったのでY成分を無視したXZ平面での移動だけ考えます。移動前の座標を始点、移動後の座標を終点とするレイを作成してレイキャストを行ってコリジョン検出を行っています。

*XZ平面でのコリジョン解決*

|  |
| --- |
| //当たった。  //壁。  CVector3 vT0, vT1;  //XZ平面上での移動後の座標をvT0に、交点の座標をvT1に設定する。  vT0.Set(nextPosition.x, 0.0f, nextPosition.z);  vT1.Set(callback.hitPos.x, 0.0f, callback.hitPos.z);  //めり込みが発生している移動ベクトルを求める。  CVector3 vMerikomi;  vMerikomi.Subtract(vT0, vT1);  //XZ平面での衝突した壁の法線を求める。。  CVector3 hitNormalXZ = callback.hitNormal;  hitNormalXZ.y = 0.0f;  hitNormalXZ.Normalize();  //めり込みベクトルを壁の法線に射影する。  float fT0 = hitNormalXZ.Dot(vMerikomi);  //押し戻し返すベクトルを求める。  //押し返すベクトルは壁の法線に射影されためり込みベクトル+半径。  CVector3 vOffset;  vOffset = hitNormalXZ;  vOffset.Scale(-fT0 + m\_radius);  nextPosition.Add(vOffset);  CVector3 currentDir;  currentDir.Subtract(nextPosition, m\_position);  currentDir.y = 0.0f;  currentDir.Normalize();  if (currentDir.Dot(originalXZDir) < 0.0f) {  //角に入った時のキャラクタの振動を防止するために、  //移動先が逆向きになったら移動をキャンセルする。  nextPosition.x = m\_position.x;  nextPosition.z = m\_position.z;  break;  } |

この176行目～205行目までがコリジョン解決を行っているコードです。この処理は**壁ずり**と呼ばれている壁に衝突した際の典型的なコリジョン解決となります。

## **7.7.1 壁ずり**

では壁ずりについて見ていきましょう。下記の図のようにキャラクターが移動した場合を考えてみて下さい。

移動した結果、この点で衝突した。

シンプルに考えたら、下記のような衝突解決が考えられるます。

**シンプルな衝突解決**

衝突点に戻して、半径分押し戻す。

しかし、この押し戻し処理を行うと、本来ユーザーが望んでいた移動量より少ない量しか移動しないため、ユーザーに大きなストレスを与えることとなります。

ユーザーが期待している移動先はここ。

171行目の#if 0 を#if 1に変更すると、シンプルな衝突解決を行っている処理に変更されます。どのような動作になるか確認してみてください。

## **7.7.2 壁ずりの実装**

　では、壁刷りの実装を見ていきましょう。壁刷りのコードは184行目～202行目までの処理となります。

Lesson08/character/CharacterController.cpp(184行目～203行目)

|  |
| --- |
| CVector3 vT0, vT1;  //XZ平面上での移動後の座標をvT0に、交点の座標をvT1に設定する。  vT0.Set(nextPosition.x, 0.0f, nextPosition.z);  vT1.Set(callback.hitPos.x, 0.0f, callback.hitPos.z);  //めり込みが発生している移動ベクトルを求める。  CVector3 vMerikomi;  vMerikomi = vT0 - vT1;  //XZ平面での衝突した壁の法線を求める。。  CVector3 hitNormalXZ = callback.hitNormal;  hitNormalXZ.y = 0.0f;  hitNormalXZ.Normalize();  //めり込みベクトルを壁の法線に射影する。  float fT0 = hitNormalXZ.Dot(vMerikomi);  //押し戻し返すベクトルを求める。  //押し返すベクトルは壁の法線に射影されためり込みベクトル+半径。  CVector3 vOffset;  vOffset = hitNormalXZ;  vOffset \*= -fT0 + m\_radius;  nextPosition += vOffset; |

このプログラムは下記のような計算を行っています。

② vT1(衝突点)

③ vMerikomi = vT0－vT1

④ hitNormalXZ

衝突した面の法線

⑤ fT0 = dot( vMerikomi, itNormalXZ)

めり込み量

⑥ vOffset = (fT0 + m\_radius) × hitNormalXZ

①　vT0(移動先)

移動先の座標(vT0)と衝突点の座標(vT1)を減算して、vMerikomiを計算します。そして、vMerikmiと衝突した面の法線(hitNormalXZ)との内積を計算します。内積には大きさ１のベクトルとの内積を計算すると、**そのベクトルに射影した長さが求まるという性質があります。**この性質を利用して、押し戻す量のvOffsetを計算しています。

## **7.8 下方向のあたり判定**

　XZ平面の移動での衝突解決を行うことができたら、最後は下方向のあたり判定を行っています。下方向の衝突解決は、XZ平面のときのように難しいことは考えずに、単に衝突点に戻しているだけです。

|  |
| --- |
| g\_physics.ConvexSweepTest((const btConvexShape\*)m\_collider.GetBody(), start, end, callback);  if (callback.isHit) {  //当たった。  moveSpeed.y = 0.0f;  m\_isJump = false;  m\_isOnGround = true;  nextPosition.y = callback.hitPos.y;  }  else {  //地面上にいない。  m\_isOnGround = false;  } |

## **7.9 衝突解決後の座標を戻り値で返す**

　すべての衝突解決が行えたら、解決後の座標を剛体に設定して戻り値として返しています。キャラクターコントローラーを使用する側は、この戻り値を次の移動先として使用すれば、静的オブジェクトにめり込むことなく移動できるようになります。

## **7.10 今後の課題**

　このキャラクターコントローラーには、ゲームによっては、解決しなくてはならない、いくつかの問題があります。

　①　上方向のあたり判定を取っていないため、ジャンプできるゲームで天井が低い場合

にあたり抜けが起きてしまう。

②　回転に対応していないため、動物などのあたり判定が難しい。

　③　動的オブジェクト(動くオブジェクト)とのあたり判定はサポートしていない。

今後の作品制作でこれらの課題を解決する必要に迫られたときは、ぜひ、自身の力で解決してみてください。

## **7.11　実習課題**

　Lesson\_08/Assets/modelData/mikyan.fbxをVisualStudioに追加して、みきゃんを動かないオブジェクトとして、マップに表示しなさい。また、みきゃんはPhysicsStaticObjectクラスの利用して物理ワールドに登録を行い、ユニティちゃんとめりこまないようにしなさい。

1. (\*) ラッパークラスとは、既存のクラスのインスタンスを保持して、そのクラスを使いやすくなるようにインターフェースを定義したり、使用できる機能を制限したりすることが目的のクラスのことです。 [↑](#footnote-ref-1)