はじめに

このチャプターでは、現在のキャラクターアニメーションのデファクトスタンダードである、スキンアニメーションの仕組みについて勉強していきます。

Opt.4.1 概要

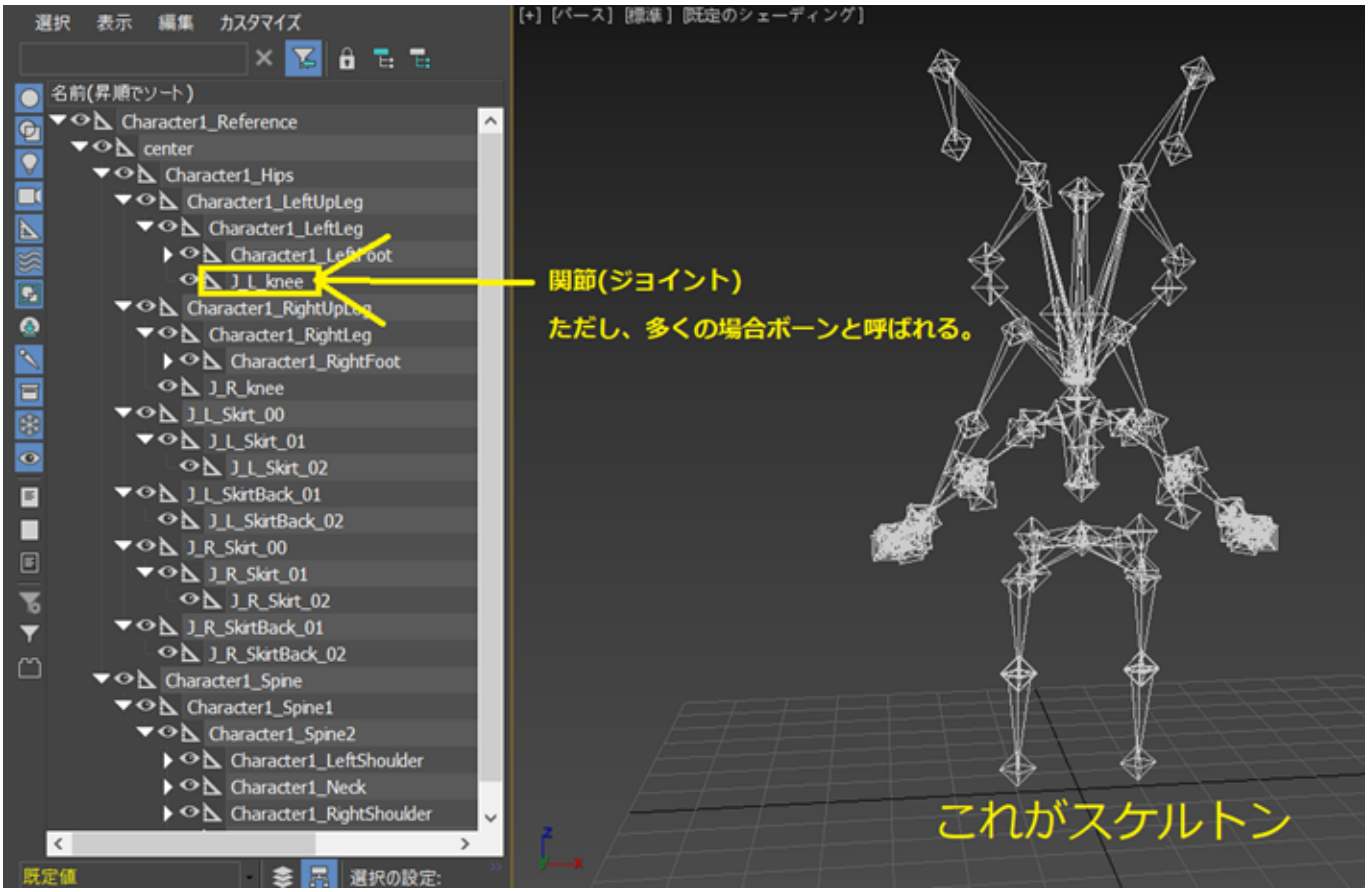
　スキンアニメーションとはスケルトンと呼ばれる骨情報を使って、キャラクターをアニメーションさせる手法です。スケルトンは骨の集合で、その骨を動かすことで、キャラクターをアニメーションさせることができます。スキンアニメーションで扱う、アニメーションデータは、各キーフレームごとに、スケルトンの骨の位置、回転、拡大率を記憶しているデータです。このアニメーション手法のことを、キーフレームアニメションといいます。３Ｄモデルの頂点には、どの骨の影響を受けるのかという情報が埋め込まれています。例えば、前腕の頂点には、影響をうける前腕の骨の番号が埋め込まれています。では、以降ではこれらについて、詳細に解説をしていきます。

Opt.4.2 スケルトン

　スケルトンはボーン(骨)の配列です。ボーンは、その骨の位置、回転、拡大情報を保持しています。そしてスケルトンはボーンの配列です。プログラム的には次のようなデータ構造になります。

```
// ボーン。
struct Bone{
    Vector3 position;       // 位置。
    Quaternion rotation;    // 回転。
    Vecto3 scale;           // 拡大率。
};
// スケルトン。
struct Skeleton{
    std::vector<Bone>   boneArray;  // 骨の配列。
};
```

　スケルトンは3dsMaxやMayaなどのDCCツールで作成します。図Opt4.1はUnityちゃんのスケルトンです。
図Opt4.1 3dsMax



DCCツールで作成された、スケルトンデータはゲーム中に利用されるので、ゲームで扱える形式で出力する必要があります。本校のエンジンにはtksファイルというスケルトンデータが用意されています。このデータは、3dsMaxでスキンモディファイアが設定されている、モデルデータを、tkExporterを利用して、tkmファイルを保存することで、自動的に出力されています。スキンアニメーションを行うためには、まずこのスケルトンデータをロードして、アニメーションさせたいモデルとバインドさせる必要があります。

Opt4.3 【ハンズオン】スケルトンとモデルとバインドさせて、骨を動かしてみよう。

さて、スケルトンの詳細な説明をする前に、とりあえず、手を動かしてスケルトンのイメージをつかんでみましょう。ここまでに説明してきたように、スケルトンとモデルをバインドさせることができたら、骨を動かすことで、キャラクターをアニメーションさせることができます。この骨を動かす情報が記憶されているのが、アニメーションデータなのですが、アニメーションデータがなくても骨を動かすことはできます。このハンズオンでは、Optional_04_01を使って、ゲームコントローラーの操作で、ユニティちゃんの首の骨を動かして、ろくろ首ユニティちゃんを作ってみましょう。では、Optional_04_01/Optional_04_01.slnを立ち上げてください。

step-1 スケルトンデータをロードする。

まずは、スケルトンデータをロードする必要があります。本書にはスケルトンを扱うためのSkeletonクラスが用意されています。このクラスを利用して、スケルトンデータをロードしましょう。main.cppにリスト-O4.1のコードを入力してください。
[リスト-O4.1 main.cpp]

```
// step-1 スケルトンデータをロードする。
Skeleton skeleton;
skeleton.Init("Assets/modelData/unityChan.tks");
```

Skeleton::Init()関数に指定した、unityChan.tksというファイルがスケルトンデータです。ファイル名を間違えないように気を付けて下さい。

step-2 モデルデータをロードして、スケルトンと関連付けする。

　続いて、モデルデータをロードして、スケルトンと関連付けしましょう。リスト-O4.2のコードを入力してください。
[リスト-O4.2 main.cpp]

```
// step-2 モデルデータをロードして、スケルトンと関連付けする。
ModelInitData modelInitData;
// tkmファイルのファイルパスを指定する。
modelInitData.m_tkmFilePath = "Assets/modelData/unityChan.tkm";
// シェーダーファイルのファイルパスを指定する。
modelInitData.m_fxFilePath = "Assets/shader/model.fx";
// ノンスキンメッシュ用の頂点シェーダーのエントリーポイントを指定する。
modelInitData.m_vsEntryPointFunc = "VSMain";
//【注目】スキンメッシュ用の頂点シェーダーのエントリーポイントを指定。
modelInitData.m_vsSkinEntryPointFunc = "VSSkinMain";
//【注目】スケルトンを指定する。
modelInitData.m_skeleton = &skeleton;

// 初期化情報を使って、モデルを初期化。
Model model;
model.Init(modelInitData);
```

　モデル初期化情報に、step-1でロードした、スケルトンのアドレスを指定していることに注目してください。これで、モデルとスケルトンの関連付けが行われます。
　また、もう一点、スキンメッシュ用の頂点シェーダーのエントリーポイントを指定している箇所にも注目してください。ここでは、まだ詳細は説明しませんが、スケルトンを使って、モデルを動かすには、専用の頂点シェーダーを作成する必要があります。今回は私の方で用意していますので、皆さんが実装することはありませんが、スケルトンを利用して、3Dモデルを動かすためには、スケルトンを利用した座標変換を行う頂点シェーダーを作成する必要があるということだけ、覚えておいてください。

step-3 コントローラーの入力で頭の骨を動かす。

　ここからはゲームループの処理です。ゲームコントローラーの入力で骨を動かすプログラムを実装しましょう。リスト-O4.3のプログラムを入力してください。
[リスト-O4.3]

```
// step-3 コントローラーの入力で頭の骨を動かす。
// 骨の名前でボーンIDを検索する。
int boneId = skeleton.FindBoneID(L"Character1_Neck");
// 検索したボーンIDを使って、ボーンを取得する。
Bone* bone = skeleton.GetBone(boneId);
// ボーンのローカル行列を取得する。
Matrix boneMatrix = bone->GetLocalMatrix();
```

```
// コントローラーを使って、ローカル行列の平行移動成分を変化させる。
boneMatrix.m[3][0] -= g_pad[0]->GetLStickXF();
boneMatrix.m[3][1] += g_pad[0]->GetLStickYF();
// 変更したボーン行列を設定する。
bone->SetLocalMatrix(boneMatrix);
```

ここで、操作しているボーンの情報が行列であることに注意してください。こちらも詳細は後述しますが、最終的にボーンの情報は頂点シェーダーに送られて、頂点の座標変換で利用されます。頂点シェーダーで座標変換は、頂点座標に行列を乗算することで行えました。ですので、ボーンの位置、回転、拡大などの情報も行列でGPUに送る必要があります。行列の3行目の成分は平行移動成分なので、ここでは直接行列をいじって、骨を動かしています。ここで動かしてるボーンは親の座標系での行列です。

step-4 スケルトンを更新する。

続いて、スケルトンを更新します。step-3でボーンの行列を動かしましたが、実はあの行列は最終的にGPUに送られる行列ではありません。Skeleton::Update()関数を呼び出すことで、最終的にGPUに送られるボーン行列が計算されます。リスト-O4.4のプログラムを入力してください。
[リスト-O4.4]

```
// step-4 スケルトンを更新する。
// Skeleton::Update()関数の中で、ワールド行列を計算している。
skeleton.Update(model.GetWorldMatrix());
```

step-5 モデルをドロー。

最後にモデルをドローするいつものコードを書きましょう。リスト-O4.5のプログラムを入力してください。
[リスト-O4.5]

```
// step-5 モデルをドロー。
model.Draw(renderContext);
```

入力出来たら実行してみてください。正しく実装できていると、ゲームコントローラの左スティックを使って、ユニティちゃんの首を動かすことができます。

Opt4.4 ローカル行列とワールド行列

ボーンは位置、回転、拡大を表す情報として、行列を保持しています。行列を保持している理由は、先ほど説明したように、最終的にGPUで座標変換を行う場合に、行列が必要になるからです。しかし、ボーンが保持している行列は一つだけではありません。ここでは、その中でも特に重要な「ローカル行列」と「ワールド行列」について解説していきます。ワールド行列は分かりやすいと思います。イメージ通り、ボーンのワールド空間上での位置、回転、拡大を表す行列です。最終的にGPUに送られて頂点シェーダーで利用される行列です。では、「ローカル行列」とはいったい何なのでしょうか。ここでのローカル行列は「親のボーン座標系での行列」を指しています。この後詳しく勉強するキーフレームアニメーションデータには、時間ごとの各ボーンの位置、回転、拡大率の情報が記憶されています。つまり、行列が記憶されているのです

が、ここで記憶されている行列は、ワールド行列ではなく、親のボーン座標系での行列、つまりローカル行列です。スケルトンは親子関係を持っている階層構造のデータです。例えば、前腕の骨の親は上腕の骨になります。親子関係のあるデータは親が動くと子供が動きます。上腕の骨を動かすと前腕の骨が動きますよね。親のボーンの座標系というのは、親のボーンを原点とする空間のことです。例えば、前腕の骨の親は上腕の骨となるため、前腕の骨のローカル行列は、上腕の骨を原点とした座標系の行列です。

　では、なぜローカル行列を使うのか。結論を端的にいうと、親の座標系で動かした方が、扱いやすいことが多いからです。例えば、ワンピースのルフィを扱ったゲームを作っていることを考えてみてください。ルフィはゴムゴムの実を食べたゴム人間なので、腕を伸ばすことができます。このような処理を実装したい場合、まさに、骨を利用して、腕を伸ばすのですが、では前腕の腕を、腕を伸ばしている方向に伸ばしたいとします。このとき、骨をワールド空間で扱うと、腕を伸ばす方向はルフィの回転、腕をどこに伸ばしているかということを考慮する必要が出てきて、プログラムが複雑になってしまいます。しかし、ローカル座標系で考えれば、そんなことは気にする必要がなくなります。このような要因から、親子構造のデータを扱う場合は、ローカル座標系で考えた方がプログラム的に扱いやすくなるのです。

Opt4.5 ワールド行列の計算

　さて、アニメーションのしやすさや、動かしやすさから、cpp側ではボーンの行列はローカル行列で扱われるのですが、最終的にGPUに送るときはワールド行列になっている必要があります。このローカル行列からワールド行列を計算しているのが、Skeleton::Update()関数です。

　では、ローカル行列からワールド行列はどのように計算されるのでしょうか。答えは、スケルトンのルートボーン(最上位の親)から子供に向かって、ローカル行列を掛け算していって、最終的なワールド行列を求めます。例えば、図Opt4.2のような階層構造を持っているボーンAのワールド行列を計算する場合は、図Opt4.3のような計算を行います。

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

　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。図Opt4.4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)となり、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。