**Chapter 1**

**Xファイルを使用したアニメーションしないモデル表示。**

**1.1 Xファイル**

XファイルとはDirectX2.0から導入されたモデルフォーマットで、DirectX9までサポートされていました。DirectX10以降は標準モデルフォーマットというものは用意されなくなり、自分でモデルフォーマットを考える必要があります。

現在はサポートされていないフォーマットのため、Xファイルの使い方を学ぶ意味は薄いように思えるかもしれませんが、そもそもどこの環境に行っても使えるモデルフォーマットなど存在しません。自前でエンジンを作っている会社は自分たちでモデルフォーマットを作成しています。しかし、モデルを表示するための基本的な概念はどのモデルフォーマットでも共通となっており、xファイルを使用したモデル表示の仕方を学んでおけば、独自のモデルフォーマットを使用するライブラリに出会ったとしても、「似たような感じだな」というように思えるはずです。

|  |
| --- |
| Tips  モデルの表示に関して、どの環境でも通用する技術とは頂点バッファ、インデックスバッファ、シェーダーなど低レベルな知識になります。非常に重要な知識なのですが、２年生の前期にこの内容をやっていると就職作品の作成が間に合わないため、前期に詳細な説明は行いません。 |

**1.2 　Xファイルのロード**

Xファイルを用いてモデルを表示するためには、D3DXLoadMeshFromX関数を使用してXファイルをロードして、ID3DXMeshのインスタンスを作成する必要があります。ID3DXMeshとは内部にモデルを表示するための頂点バッファやインデックスバッファを保持したモデルクラスのようなものです。このAPIは下記のように使用します。

Lesson\_01/main.cpp(183行目～187行目)

|  |
| --- |
| D3DXLoadMeshFromX(  "Tiger.x", 　　　　　　　　　　//ファイルパス  D3DXMESH\_SYSTEMMEM,　//メッシュ作成のオプション。基本これでいい。  　　　　　　　　　　　　　　 //他にも大事なオプションはあるのだが、今は説明しない。  g\_pd3dDevice, //D3Dデバイス。  NULL,　　　　　　　　　　　 //ポリゴンの隣接情報の出力先。  //モデルをロードするだけならNULLでいい。  &pD3DXMtrlBuffer, //マテリアルバッファの出力先。後述。  NULL, 　　　　　　　　　　 //NULLでいい。  &g\_dwNumMaterials,　　　　 //マテリアルの数の出力先。後述。  &g\_pMesh 　　　　　　　　　 //ID3DXMeshのインスタンスの格納先。  ) |

これでID3DXMeshのインスタンスが生成されました。

**1.3 マテリアル**

マテリアルとはモデルの質感を決定するためのものです。例えばテクスチャ、鏡面反射率などの設定を行うものです。D3DXLoadMeshFromXから取得できるテクスチャ以外のマテリアル情報は固定機能と呼ばれる、現在は廃れた機能の情報しか取得できないため。今回使用するサンプルでは使用しません。今回のサンプルではマテリアル情報はテクスチャを引っ張ってくるためだけに使用します。マテリアルからテクスチャを引っ張ってくるコードは下記のようになります。

Lesson\_01/main.cpp(188行目～202行目)

|  |
| --- |
| // マテリアルバッファを取得。  D3DXMATERIAL\* d3dxMaterials = ( D3DXMATERIAL\* )pD3DXMtrlBuffer->GetBufferPointer();  g\_pMeshTextures = new LPDIRECT3DTEXTURE9[g\_dwNumMaterials];  for( DWORD i = 0; i < g\_dwNumMaterials; i++ )  {  g\_pMeshTextures[i] = NULL;  //テクスチャを作成する。  D3DXCreateTextureFromFileA( g\_pd3dDevice,  d3dxMaterials[i].pTextureFilename,  &g\_pMeshTextures[i] );  }  // マテリアルバッファを解放。  pD3DXMtrlBuffer->Release(); |

**1.4 エフェクトファイルのロード。**

モデルを表示するためには、拡張子が.fxのエフェクトファイルと言われるものをロードする必要があります。このエフェクトファイルはHLSLという言語で記述されたシェーダープログラムになります。シェーダーは近年のゲームのグラフィックスを語る上で欠かすことのできない、非常に重要な要素になります。しかし、この話をするだけでかなりの時間がかかりますので、この話は後期に行います。今はこのように記述を行う必要があるのだなという風にだけ覚えておいてください。

　エフェクトファイルのロードは下記のように行います。

Lesson\_01/main.cpp(42行目～60行目)

|  |
| --- |
| void LoadEffectFile()  {  LPD3DXBUFFER compileErrorBuffer = NULL;  //シェーダーをコンパイル。  HRESULT hr = D3DXCreateEffectFromFile(  g\_pd3dDevice,  "basic.fx",  NULL,  NULL,  D3DXSHADER\_SKIPVALIDATION,  NULL,  &g\_pEffect,  &compileErrorBuffer  );  if (FAILED(hr)) {  MessageBox(NULL, (char\*)(compileErrorBuffer->GetBufferPointer()), "error", MB\_OK);  std::abort();  }  } |

**1.5 モデルの描画処理**

ここまでは全て初期化と言われる処理で、これでやっとモデルを表示することができます。では実際にモデルを描画するコードを見てみましょう。

Lesson\_01/main.cpp(207行目～242行目)

|  |
| --- |
| void RenderTiger()  {  //シェーダー適用開始。  g\_pEffect->SetTechnique("SkinModel");  g\_pEffect->Begin(NULL, D3DXFX\_DONOTSAVESHADERSTATE);  g\_pEffect->BeginPass(0);  //定数レジスタに設定するカラー。  D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);  //ワールド行列の転送。  g\_pEffect->SetMatrix("g\_worldMatrix", &g\_worldMatrix);  //ビュー行列の転送。  g\_pEffect->SetMatrix("g\_viewMatrix", &g\_viewMatrix);  //プロジェクション行列の転送。  g\_pEffect->SetMatrix("g\_projectionMatrix", &g\_projectionMatrix);  //ライトの向きを転送。  g\_pEffect->SetVectorArray("g\_diffuseLightDirection", g\_diffuseLightDirection, LIGHT\_NUM );  //ライトのカラーを転送。  g\_pEffect->SetVectorArray("g\_diffuseLightColor", g\_diffuseLightColor, LIGHT\_NUM );  //環境光を設定。  g\_pEffect->SetVector("g\_ambientLight", &g\_ambientLight);  g\_pEffect->CommitChanges(); //この関数を呼び出すことで、データの転送が確定する。描画を行う前に一回だけ呼び出す。    // Meshes are divided into subsets, one for each material. Render them in  // a loop  for( DWORD i = 0; i < g\_dwNumMaterials; i++ )  {  g\_pEffect->SetTexture("g\_diffuseTexture", g\_pMeshTextures[i]);  // Draw the mesh subset  g\_pMesh->DrawSubset( i );  }    g\_pEffect->EndPass();  g\_pEffect->End();  } |

よくわからないコードが多いかと思います。非常に長いコードになりましたが、これがモデルを表示するときに必要なコードになります。まだ、よく分からない部分がいくつかあるかと思いますが、今は構いません。少なくともワールド行列、ビュー行列、プロジェクション行列の設定などをしている部分を分かってもらうだけで今は十分です。

**1.6 終了処理。**

プログラムが終了、もしくはモデル表示が不要になった場合は、ここまでロードしたID3DXMeshやテクスチャ、エフェクトファイルなどを破棄する必要があります。下記に破棄を行うコードを記述します。モデルが不要になったら必ず終了処理を実行するように気をつけてください。

Lesson\_01/main.cpp(246行目～260行目)

|  |
| --- |
| void CleanupTiger()  {  if (g\_pMeshTextures != NULL) {  for (int i = 0; i < g\_dwNumMaterials; i++) {  g\_pMeshTextures[i]->Release();  }  delete[] g\_pMeshTextures;  }  if (g\_pMesh != NULL) {  g\_pMesh->Release();  }  if (g\_pEffect != NULL) {  g\_pEffect->Release();  }  } |

**1.7 まとめ**

　Xファイルを使用してモデルを表示するためには下記の手順が必要でした。

1. D3DXLoadMeshFromX関数 を使用してXファイルをロードし、ID3DXMeshのインスタンスを作成する。(初期化時に一度だけ実行)
2. D3DXLoadMeshFromX関数を使用して取得できたマテリアル情報を元にD3DXCreateTextureFromFileA関数を使用してテクスチャをロードする。(初期化時に一度だけ実行)
3. D3DXCreateEffectFromFile関数を使用してエフェクトファイルをロードする。(初期化時に一度だけ実行)
4. ロードした要素を使用してモデルの描画処理を記述する。(毎フレーム実行する)

**実習課題**

Lesson01のコードを改造して下記の実習を行ってください。

1. トラをクラス化してみましょう。

トラのクラスの最低限の要求仕様。

・トラのクラスは下記のメンバ関数実装する。

Init関数

　　　　Xファイル、テクスチャ、エフェクトファイルのロードなどの処理を記述

する。

Update関数

　　ワールド行列の更新やトラの移動などを記述する関数。

Render関数

　　 トラの描画処理を記述する。

Release関数

　　　メッシュ、エフェクト、テクスチャなどを破棄するコードを記述する。

・トラのクラスは下記のメンバ変数を最低限保持する。

　　　ID3DXEffect\*　 pEffect;

D3DXMATRIX worldMatrix;

LPD3DXMESH pMEsh;

LPDIRECT3DTEXTURE9\* pMeshTextures

DWORD numMaterial;

1. トラを２体出してください。

**Chapter 2 モデルクラスの作成**

Chapter1で作成したトラクラスはトラ固有の処理と、モデルを表示するための処理が記述されていて、まだまだ設計に改善の余地があります。例えばChapter1で作成したサンプルプログラムにヒヨコのクラスを追加するケースを考えてみて下さい。恐らくあなたは下記のようなクラスを作ることを思いつくはずです。

class Hiyoko{

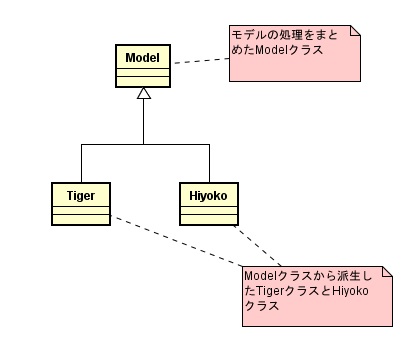
};

当然ヒヨコクラスも３Ｄモデルを表示する必要があるので、モデルを表示するプログラムを記述していくはずです。そして、ヒヨコクラスの実装を進めていくうちにモデルを表示する処理の大部分がトラクラスの実装と共通であることに気付くと思います。ソフトウェア工学において、共通の処理をコピーアンドペーストで増やしていくことは、保守性、可読性、再利用性を大きく損なう行為になります。コピーアンドペーストでコードを複製していった場合、モデル表示プログラムに不具合があったときや拡張が必要になった場合、コピーアンドペーストを行った数だけ修正が必要になるのです。そして、人は作業の数が膨大になるほど、ヒューマンエラーを起こす確率が高くなるため、そのプログラムを保守する人は頭を抱えることになるでしょう。

**2.1 継承vs移譲**

**2.1.1 継承**

　　C++の継承を学んだプログラマであれば、この問題の解決に継承を使用しようと考えるでしょう。恐らく下記のようなクラスを設計すると思います。



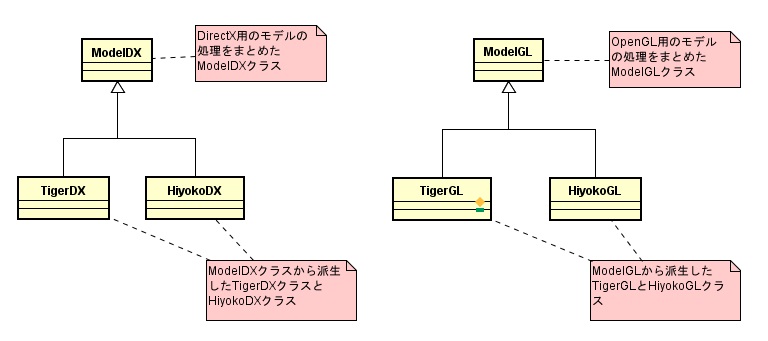
　プログラムは下記のようなものになっているでしょう。

|  |
| --- |
| class Model{  //定義は省略。  };  class Tiger : public Model  {  　//定義は省略。  };  class Hiyoko : public Model  {  //定義は省略。  }; |

TigerクラスとHiyokoクラスから、共通するモデル関連の処理を抽出したModelクラスを作成して、TigerとHiyokoをModelの派生クラスにしています。これによって、モデル関連の処理への修正や、拡張の作業が発生した場合は、Modelクラスの処理のみを変更すれば良くなります。コピーアンドペーストで処理を増やしていく実装に比べると、かなり改善されたと言えます・・・。しかしこの設計でもまだ大きな問題が起きるケースがあります。次節ではその問題について見ていきましょう。

**2.1.2 組み合わせの爆発**

　では、先ほどのTigerクラス、Hiyokoクラス、Modelクラスについて見てみましょう。もともとModelクラスはMicrosoft社が提供するSDKのDirectXで実装をされていました。しかし、ある日クライアントから次のような要求が来ました。「macユーザーも遊べるようにOpenGLでも動作するようにして欲しい。」この要望に応えるために、あなたは下記のように設計を変更しました。



この設計変更により、TigerDXとTigerGL、HiyokoDXとHiyokoGLは共通のコードが多数あるコピーアンドペーストと同じ保守性、拡張性、再利用性の低いクラスになってしまいました。

**2.1.3 委譲**

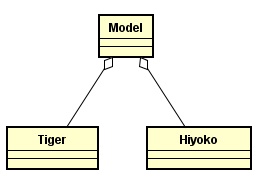
「継承よりも委譲を使おう」オブジェクト指向を用いた、よりよい設計を考える際に、継承を行う場合、先に移譲が行えないか検討することが推奨されています。では委譲とはなにか？これはあるクラスの責任を別のクラスに譲り渡すことです。ではもともとの虎クラスを見てみましょう。元々の虎のクラスは下記の二つの処理を正しく実行する責任がありました。

・トラの挙動(歩くとか走るとか)

・トラの表示する

この二つの処理のうち、「トラを表示する」という処理を、新しくModelといクラスを作成して責任を譲り渡します。これが委譲です。そして、トラクラスはModelクラスを継承するのではなく、Modelクラスのインスタンスを保持する形に変更します。これがコンポジションや集約と呼ばれるものです。

では、委譲を使用した場合のクラス図を見てみましょう。



クラス図を見ても分かりにくいかと思いますので、実際のコードを見てみましょう。

|  |
| --- |
| class Model{  //定義は省略。  };  class Tiger  {  **Model model; //Modelのインスタンスを保持！！！**  　//定義は省略。  };  class Hiyoko  {  **Model model; //Modelのインスタンスを保持！！！**  //定義は省略。  }; |

　Tiger、HiyokoがModelクラスのインスタンスを保持しています。これがコンポジション、集約といわれるものです。では、なぜこれが継承を使用した設計より優れているのかを、先ほどのDirectX、OpenGLの話から考えてみましょう。

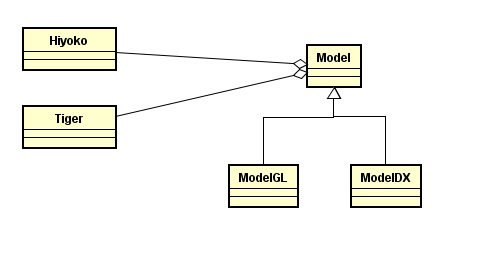
　　まず、DirectX用のModelDXクラスとOpenGL用のModelGLクラスを用意する必要があります。しかし、モデルクラスというのは往々にして共通のインターフェースを保持するものです。例えばDraw関数とか。そこで、基底クラスにModelクラスを作成します。そしてTigerとHiyokoクラスにはModelクラスのポインタを保持させます。

|  |
| --- |
| //モデルの基底クラス。  class Model{  public:  virtual void Draw() = 0; //純粋仮想関数。  };  //DirectX用のモデルクラス。  class ModelDX : public Model{  public:  void Draw();  };  //OpenGL用のモデルクラス。  class ModelGL : public Model{  public:  void Draw();  };  //トラクラス。  class Tiger {  Model\* model;  public:  //モデルのインスタンスを設定。  void SetModel(Model\* pModel)  {  model = pModel;  }  };  //ヒヨコクラス。  class Hiyoko{  Model\* model;  public:  //モデルのインスタンスを設定。  void SetModel(Model\* pModel)  {  model = pModel;  }  }; |

では、この設計の最後のトリックを紹介します。

|  |
| --- |
| Hiyoko hiyoko; //ヒヨコ  Tiger tiger;　　 //トラ  void Func()  {    if( オープンＧＬを使用する場合 ){  hiyoko.SetModel(new ModelGL );  tiger.SetModel(new ModelGL);  }else if( DirectXを使用する場合 ){  hiyoko.SetModel( new ModelDX );  tiger.SetModel( new ModelDX );  }  } |

　　では最後にクラス図を見てみましょう。



　　いかがでしょうか。見事に冗長性が排除され、拡張性、保守性に優れた設計になっています。

**2.2 まとめ**

継承と委譲に関して、絶対に継承よりも委譲を使用しなさいというものではありません。ただ、設計の指針として継承よりも委譲を使おうという指針を頭に入れておくだけでも、設計はより優れたものになります。

**実習課題**

1. Lesson02/Questionを改造して、虎と車を描画できるようにしなさい。車クラスは虎クラスのコピー&ペーストで作成しなさい。車のモデルはcar.xを読み込んで表示しなさい。
2. ①のプログラムを改造して、モデルクラスを作成しなさい。そして、そのモデルクラスを使用して虎と車のモデル表示処理を委譲するようにしなさい。

**Chapter 3 アニメーション**

**3.1 モーフィング**

この節では頂点単位のアニメーションのモーフィングについて見ていきます。モーフィングはフェイシャルアニメーション(顔のアニメーション)でよく使われており、昨今のゲームには欠かすことのできない技術になっております。フェイシャルアニメーションはボーンを使用して実装することもできますが、最近のフォトリアルなゲームは役者の顔で３Dキャプチャーを行い、モーフターゲットとして使用することでリアルな表情を実現しています。

**3.1.1 モーフターゲット**

モーフィングを行うためには、モーフターゲットというデータが必要になります。モーフターゲットを簡潔に説明すると、例えばキャラクタを無表情から笑っている顔にアニメーションさせたい場合、無表情のモデルと笑っているモデルの二つを作成します。そして、無表情のモデルと笑っているモデルとで、0.0～1.0のブレンディング率を使用して、同じ番号の頂点をブレンディングしていきます。頂点ブレンディングの計算式は下記になります。

　モデルAの100番目の頂点をVA、モデルBの100番目の頂点をVBとして、ブレンディング率をRとすると

　　モーフィング後の頂点 = VA \* (1.0 -R) + VB \* R

となる。

**3.1.2 DirectXでの頂点アクセス**

実際にモーフィングを行うためにはモデルの頂点バッファにアクセスする必要があります。ここではモデルの頂点バッファにアクセスする方法を紹介します。今回はソフトウェアモーフィングを行いますので、CPUでモーフィングを行うことにします。

　Xファイルをロードすると、ID3DXMeshのインスタンスを使用してモデルの表示などが行えます。このインスタンスを使用すれば、頂点バッファにアクセスすることができます。

頂点バッファを取得するにはID3DXMesh:: GetVertexBufferを使用します。

|  |
| --- |
| **LPDIRECT3DVERTEXBUFFER9 vertexBuffer;**  **mesh->GetVertexBuffer(&vertexBuffer); //頂点バッファを取得。** |

頂点バッファとは、モデルの頂点情報をまとめて管理するバッファです。下記のようなバッファと考えるとイメージしやすいのではないでしょうか。

|  |
| --- |
| **//頂点**  **struct Vertex{**  **D3DXVECTOR3 pos; //座標**  **D3DXVECTOR3 normal; //頂点の向きを表す法線。**  **D3DXVECTOR2 uv; //テクスチャをサンプリングするためのUV座標。**  **};**  **Vertex vertexBuffer[1256]; //頂点数が1256の頂点バッファ。** |

このコードは擬似コードなのですが、イメージはこのようになります。

さて、頂点データを書き換えるためには、CPUで頂点を書き換えている最中にGPUがその頂点バッファにアクセスできないようにロックをかける必要があります。頂点バッファのロックはLPDIRECT3DVERTEXBUFFER9のLock関数を使用すれば実行できます。

|  |
| --- |
| char\* pVertex;  vertexBuffer->Lock(0, desc.Size, (void\*\*)&pVertex, D3DLOCK\_DISCARD); |

Lock関数を使用すると頂点バッファをロックすることができ、pVertexに頂点バッファに対する生のメモリアドレスが格納されます。ロックを行ったあとは、pVertexを使って直接頂点バッファを書き換えることができます。

頂点の書き換えが完了したら、頂点バッファをアンロックする必要があります。アンロックを忘れてしまうと、GPUがいつまでたってもその頂点にアクセスすることができなくなるため、GPUがフリーズします。

|  |
| --- |
| vertexBuffer->Unlock(); |

**3.1.2.1 頂点ストライド**

頂点情報はモデルによって内容が変わります。例えばテクスチャを貼らないモデルであればUVの要素はいらなくなりますし、ライティングを行わない場合はnormalの要素がいらなくなることもあります。そのため、頂点にアクセスするときは一つの頂点のサイズが必要になります。一つの頂点のサイズは次のようなコードで取得できます。

|  |
| --- |
| //頂点バッファの定義を取得する。  D3DVERTEXBUFFER\_DESC desc;  vertexBuffer->GetDesc(&desc);  //一つの頂点のサイズを計算する。  //desc.sizeには頂点バッファのサイズが入っているので、  //これを頂点数で除算してやれば一つの頂点のサイズがわかります。  int stride = desc.Size / mesh->GetNumVertices(); |

**3.1.2.2**

では頂点を書き換えるためのプログラムを見てみましょう。

|  |
| --- |
| D3DXVECTOR3\* vertexPos;  //頂点バッファをロック  vertexBuffer->Lock(0, desc.Size, (void\*\*)& vertexPos \_B, D3DLOCK\_DISCARD);  for (int vertNo = 0; vertNo < mesh->GetNumVertices(); vertNo++) {  //頂点座標に+1.0していく。  vertexPos->x += 1.0f  vertexPos->y += 1.0f:  vertexPos->z += 1.0f  //次の頂点へ。  char\* p = (char\*)vertexPos;  p += stride;  }  vertexBuffer->Ulock(); |

実習課題

　モーフィングを学ぶ課題を使用して、ユニティちゃんがフェイシャルアニメーションできるようにしてください。

**3.2 スキンアニメーション**

すでにスケルトン(骨組み)を使用した階層アニメーションは勉強しましたが、ここまで勉強した階層アニメーションの手法では、複数のパーツに分かれているオブジェクトを描画するときに切れ目が発生したり、人肌のようなワンメッシュのモデルであっても関節のつなぎ目で不自然なアーティファクトが発生してしまいます。これを解決するための手法がスキンアニメーションまたはスキニングと言われるものです。

**3.2.1 スキンウェイト**

　ではどのようにすればパーツの切れ目や、不自然なアーティファクトを除去することができるのでしょうか？例えば人体の腕について考えてみましょう。人の腕は肩から上腕、前腕、掌、手の指など多数のボーンが存在します。今回は上腕と前腕について考えてみましょう。

　上腕と前腕をアニメーションさせる場合、3dsMaxなどのDCCツールを使用して3Dモデルデータの上腕と前腕の各頂点がどのボーンに関連づいているかを設定することで、ボーンを使用した階層アニメーションが実現できます。さて、腕の各頂点にボーンと関連付けを行うと言いましたが、例えば肘の付近の頂点は上腕と前腕のどちらのボーンに関連付けを行えばいいでしょうか？どちらに関連付けを行っても不自然なアーティファクトが発生しそうです。これを解決するのがスキンウェイトと言われるものです。

では肘の話に戻します。肘のような骨と骨のつなぎ目の関節付近の頂点は上腕と前腕の二つのボーンに関連付けを行います。そして、例えば上腕のボーンに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、肘の頂点をどのように変換するのか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、下記のようなコードになります。

|  |
| --- |
| D3DXVECTOR4 vTmp;  //上腕のボーン行列で変換させた頂点座標をvTmpに代入。  D3DXVec4Transform(&vTmp, &vSrc, &m0);  //スキンウェイトを乗算してvDstに代入。  vDst = vTmp \* w0;  //前腕のボーン行列で変換させた頂点座標をvTmpに代入。  D3DXVec4Transform(&vTmp, &vSrc,&m1);  //スキンウェイトを乗算してvDstに加算  vDst += vTmp \* w1; |

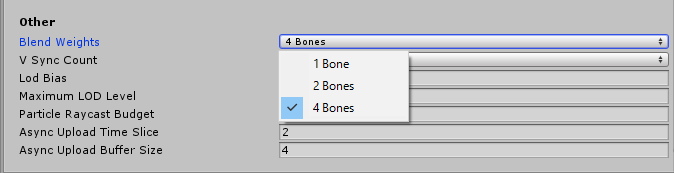
**3.2.2 関連付けできるボーンの本数**

　スキンウェイトを設定できるようになれば、各頂点に関連付けできるボーンの数も増やすことができます。例えば肩の辺りの頂点であれば、胴体、上腕、首と３つのボーンと関連付けされているかもしれません。しかしこれらもスキンウェイトを使えば簡単に解決できます。胴体のボーンに0.3、上腕に0.3、首に0.4のスキンウェイトを設定すればいいのです。複数ボーンに関連付けできる場合のスキニングの疑似コードを下記に示します。

|  |
| --- |
| D3DXVECTOR3 dstPos = D3DXVECTOR3(0.0f, 0.0f, 0.0f);  for (int boneNo = 0; boneNo < numBone; boneNo++) //ボーンの本数分ループを回す。  {  D3DXVECTOR4 vTmp;  D3DXVec4Transform(&vTmp, &vSrc, &boneMatrixArray[IndexArray[boneNo]] );  　dstPos += vTmp \* blendWeightsArray[iBone];  } |

**3.2.3 スキニングのパフォーマンス**

スキニングはモデルのすべての頂点に対して行われます。最近のハイエンドのゲームであれば、キャラクタのモデルの頂点数が10万を超えることも珍しくありません。そのため、スキニングはほとんどのケースでGPUなどの高速なプロセッサで計算されます。また、前節の関連付けできるボーンの数に関しても、当然ですが数が少ないほど処理が高速になります。そのため関連付けできるボーンの数に上限を設けているゲームエンジンがほとんどです。下記の図はUnityのクオリティセッティングの図です。



Blend Weightsの部分が関連付けできるボーンの上限になります。

**3.2.4 実習**

　スキニングを学ぶ課題を使用して、スキニングを実装してみて下さい。今回は分かりやすくするためにCPUでのスキニングのソフトウェアスキニングを実装してもらいます。

**3.3 アニメーション付きXファイル**

DirectX9でサポートされているXファイルにはアニメーションデータを付随することができます。この節では、アニメーション付きXファイルの作成の仕方を紹介します。

**3.3.1 アニメーションデータ**

　Xファイルに付随するアニメーションデータはAnimationSetという名前で付随しています。AnimationSetのデータを見てみると、キーフレームが打たれており、そのキーの時に各ボーンがどのような姿勢になっているかという情報が付随されています。

**3.3.2 アニメーション付きXファイルの出力方法。**

　ではアニメーション付きXファイルを出力する方法を動画を使用して説明します。今回は3dsMaxのkwxport.dleというプラグインを使用します。動画\アニメーション付きXファイルの出力.mp4を参照してください。

**3.3.3 アニメーションの追加**

kwxport.dleは一つのXファイルに対して一つのアニメーションしか出力することができません。しかし、ゲームでは複数のアニメーションを使用して多彩なアクションを行うことでキャラクターに息を吹き込んでいます。ここではXファイルにアニメーションを追加して、複数のアニメーションを扱う方法を勉強します。こちらも動画を用意していますので、動画\アニメーションの追加.mp4を参照してください。

**実習**

1. 「アニメーション付きXファイルを出力する課題」を使って、アニメーション付きのユニティちゃんを表示しなさい。
2. ユニティちゃんに走りアニメーションとジャンプアニメーションを追加しなさい。アニメーションが追加出来たら、再生して正しく追加出来ていることを確認しなさい。

**4 実例で学ぶゲーム数学**

　この節ではDirectXを使用して、ゲームでの数学の活用法を紹介していきます。この授業は数学の授業ではないため、公式の証明などの話はしません。先人の考えた公式をありがたく使わせてもらおうという趣旨の授業になります。

**4.1 ベクトル**

3Dゲームにおいて、ベクトルは多種多様な用途で活用されています。活用例を下記に示します。

　・3Dオブジェクトの座標

　・移動速度

　・3Dオブジェクトの向き

　・ポリゴンの向きを表す法線

　・モデルの頂点を表す頂点座標

　・etc

このように、ベクトルは3Dゲームを作成するうえで欠かすことのできない要素になっています。では次の節からはゲームでよく使われるベクトルの使い方を見ていきましょう。

**4.1.1 2点間の距離**

　2点間の距離の計算は衝突判定や、AIの敵発見の思考など、いろいろな箇所で多用される計算になります。3D空間ではオブジェクトの座標は3次元のベクトルで表現されています。例えば、パックマンのようなゲームで、プレイヤーに食べ物が衝突すると食べ物が消滅する仕様を実装するケースを考えてみましょう。

プレイヤーの座標をplayerPos、食べ物の座標をfoodPosとする。

|  |
| --- |
| //まず食べ物からプレイヤーに向かうベクトルが計算される。  D3DXVECTOR3 toPlayer = playerPos – foodPos;  //このベクトルの距離は三平方の定理を活用して求める。sqrtは平方根を求める関数。  float length = sqrt( toPlayer.x \* toPlayer.x + toPlayer.y \* toPlayer.y + toPlayer.z \* toPlayer.z);  if(length < 0.2f){  //食べ物とプレイヤーの距離が0.2以下になったら・・・  } |

DirectXにはベクトルの長さを計算する関数rが用意されています。上のコードをDirectXの関数を使用するように書き換えてみましょう。

|  |
| --- |
| //まず食べ物からプレイヤーに向かうベクトルが計算される。  D3DXVECTOR3 toPlayer = playerPos – foodPos;  //このベクトルの距離は三平方の定理を活用して求める。sqrtは平方根を求める関数。  **float length = D3DXVec3Length(&toPlayer);**  if(length < 0.2f){  //食べ物とプレイヤーの距離が0.2以下になったら・・・  } |

**4.1.2 ベクトルの正規化**

　ベクトルとは大きさと向きを持った情報です。ゲーム数学ではベクトルから大きさの要素を除外して、向きだけがほしい場合が多々あります。このように、ベクトルから大きさを除去して(大きさを１にする)、向きだけの情報を持ったベクトルにすることを正規化と言います。では、敵がプレイヤーに向かって移動していく仕様の実装で正規化の活用の具体例を見てみましょう。

敵の座標をenemyPos、プレイヤーの座標をplayerPosとする。

|  |
| --- |
| //敵からプレイヤーに向かうベクトルを計算する。  D3DXVECTOR3 toPlayerDir = playerPos – enemyPos;  //toPlayerDirには敵からプレイヤーまでの向きと、大きさが入っているため正規化を行って、向きだけを抽出する。  //正規化とは、ベクトルのxyzの要素をベクトルの大きさで除算することで行える。  float len = D3DXVec3Length(&toPlayerDir);  toPlayerDir.x /= len;  toPlayerDir.y /= len;  toPlayerDir.z /= len;  //toPlayerDirがプレイヤーまでの方向ベクトルになったので、enemyPosをその方向に動かしていく。  enemyPos += toPlayerDir \* 0.2f; //速度0.2で動かしていく。 |

DirectXには正規化を行う関数も用意されています。上のコードをDirectXの関数を使用するように書き換えてみましょう。

|  |
| --- |
| //敵からプレイヤーに向かうベクトルを計算する。  D3DXVECTOR3 toPlayerDir = playerPos – enemyPos;  //toPlayerDirを正規化する。  D3DXVec3Normalize(&toPlayerDir, &toPlayerDit);  //toPlayerDirがプレイヤーまでの方向ベクトルになったので、enemyPosをその方向に動かしていく。  enemyPos += toPlayerDir \* 0.2f; //速度0.2で動かしていく。 |

**4.1.3 内積**

　内積は3Dゲームで最も多用される公式の一つと言ってもいいかもしれません。内積とは二つのベクトルから計算されるもので、ベクトルVA、VBの内積は下記のように定義されています。

　VA・VB = VA.x \* VB.x + VA.y \* VB.y + VA.z \* VB.z ・・・・・ ①

また余弦定理より

|VA||VB|Cosθ= VA.x \* VB.x + VA.y \* VB.y + VA.z \* VB.z・・・・・②

となる。

この①に関しては定義であり、このように決められています。内積とはこういうものであると昔の数学者が決めたのです。そして②については、余弦定理の方程式を解くことで容易に証明できます。しかしこの授業はそれを論じる授業ではないので、そこについて言及はしません。しかし、この②番目の定義からゲームで非常に有用に使える要素がいくつか見えてきます。

**4.1.3.1　ベクトルのなす角**

|VA||VB|cosθについてみてみましょう。ここで記述されている|VA｜と|VB|は各ベクトルの長さを表しています。ではこの二つのベクトルが正規化された大きさ１のベクトルである場合、この公式は下記の様はものになります。

　　1×1×cosθ

つまり、大きさ１のベクトル同士の内積はそのベクトル同士がなす角θのcosθとなります。C言語にはacosというcosθをθに戻す関数が存在します。これを使用することでベクトル同士のなす角を求めることができます。ではこれをどのような場面で使うのか考えてみましょう。

**4.1.3.2 視野角の判定**

　メタルギアソリッド1の敵兵のAIについて考えてみましょう。メタルギアソリッド1の敵兵は視野角というデータを持っていて、プレイヤーがその視野角の中に入るとプレイヤーを発見して追いかけてくる思考になっています。この視野角の判定は内積を使用すれば簡単に行うことができます。ではサンプルコードを見てみましょう。

プレイヤーの座標をplayerPos、敵の座標をenemyPos、敵の前方方向をenemyDirとします。

|  |
| --- |
| //敵からプレイヤーに向かうベクトルを計算する。①  D3DXVECTOR3 toPlayer = playerPos – enemyPos;  //プレイヤーに向かうベクトルを正規化する。②  D3DXVec3Normalize(&toPlayer, &toPlayer);  //敵の前方方向と、プレイヤへの向きベクトルの内積を計算する。  float angle = D3DXVec3Dot(&toPlayer, &enemyDir);  //内積の結果はcosθになるため、なす角θを求めるためにacosを実行する。③  angle = acos(angle);  //これでangleにはラジアン単位の角度が入ったため、視野角の判定を行える。  if(fabsf(angle) < D3DXToRadian(45.5f)){  //視野角90度以内に入った。  } |

この計算を図示すると下記のようになります。

③なす角θ

1. 正規化前のtoEnemy

②　正規化されたtoEnemy

enemyDir(敵の前方)

**4.1.3.3 射影**

　|VA||VB|cosθの公式から、もう一つゲームで使える重要な性質が見えてきます。それは**射影**と言われるもので、例えばVAが大きさ１の正規化されたベクトルで、VBが向きと大きさを持っているベクトル(非正規化)だとします。このとき、VAとVBの内積は下記のような計算になります。

1×|VB|cosθ = |VB|cosθ

これはつまり、VBのベクトルをVAの無限線分上に垂線を落として射影した長さということになります。

大きさ１ベクトルVA

大きさと向きを持つベクトルVB

この長さが求まる

では、この計算が使用される例を見ていきましょう。

**4.1.3.3 コース定義の事例**

　グランツーリスモ、Forzaやマリオカートのようなレースゲームではコースの逆走判定や、AIコースの分岐判定などを行うためにコース定義と言われるデータが使われることがあります。また、God of warやクラッシュバンディクーなどコースに沿ってカメラが移動するようなゲームにもコース定義のようなものが作成されています。God of war3のカメラはコースドリブンカメラと言うような名前でサンタモニカスタジオで呼ばれていたそうです。

ではコース定義とはどういうものか簡単な例を見ていきましょう。

下記の図を参照して下さい。

このように、コースの向きと長さを保持しているベクトル上のデータが、コース定義と呼ばれていたりします。レースゲームであればAIはこのコース定義を参照して、現在自分が何処を走っているのかを知り、進行方向に向けてハンドルを切ることになります。

ではシンプルなAIを例に出して、コース定義と内積の射影について見ていきましょう。

内積の課題\_01/Question/game/CourceDef.cppを見てください。

|  |
| --- |
| //コース定義のデータテーブル。  CourceDefData g\_courceDatas[] = {  {  D3DXVECTOR3(0.000000f, 0.000000f, 0.000000f),  D3DXVECTOR3(0.000000f, 0.000000f, -36.480076f),  },  {  D3DXVECTOR3(0.000000f, 0.000000f, -36.480076f),  D3DXVECTOR3(44.080250f, 0.000000f, -104.641632f),  },  {  D3DXVECTOR3(44.080250f, 0.000000f, -104.641632f),  D3DXVECTOR3(128.082169f, 0.000000f, -129.202194f),  },  {  D3DXVECTOR3(128.082169f, 0.000000f, -129.202194f),  D3DXVECTOR3(170.723145f, 0.000000f, -196.723740f),  },  }; |

これが今回のサンプルのコース定義です。CourceDefDtaはノードの始点になるD3DXVEC3型のstartPosというメンバ変数と、ノードの始点になるendPosというメンバ変数持っている構造体です。

では、コース定義を利用してコース上での移動を行っているコードを見てみましょう。内積の課題\_01/Question/game/Player.cppを見てみてください。

|  |
| --- |
| D3DXVECTOR3 moveDir = D3DXVECTOR3(0.0f, 0.0f, 0.0f);  //パス移動を行う。  //プレイヤーの座標をコース定義に射影していく。  D3DXVECTOR3 courceDir = g\_courceDatas[currentCourceNodeNo].endPos - g\_courceDatas[currentCourceNodeNo].startPos;  //コースの長さを求めてから、コースの向きを求める。  float len = D3DXVec3Length(&courceDir);  D3DXVec3Normalize(&courceDir, &courceDir);  //プレイヤーがコース上のどの位置にいるかを調べる。  D3DXVECTOR3 playerPosInCourceNode = position - g\_courceDatas[currentCourceNodeNo].startPos;  float t = D3DXVec3Dot(&playerPosInCourceNode, &courceDir);  //進行方向はコースの向き。  moveDir = courceDir;  bool isPossibleMove = true;  if (t > len) {  //次のノードへ。  currentCourceNodeNo++;  if (currentCourceNodeNo >= COURCE\_NODE\_NUM) {  //終端まで移動した。  //移動はもうできない。  currentCourceNodeNo = COURCE\_NODE\_NUM;  isPossibleMove = false;  }  }  //終了ノードに向かって移動する。  if (isPossibleMove) {  //移動出来たら。  position += moveDir \* moveSpeed;  } |

currentCourceNodeNoは現在のコース定義のノード番号です。プレイヤーは現在のコースノード上を移動します。そして現在のコース上へプレイヤーの座標を射影して、ノードの長さ以上になっていたら次のノードに移動するというコードです。

**実習課題**

　カメラはプレイヤーを追尾する処理になっていますが、カメラの向きもコースの向きに合わせて方向を変えるようにしなさい。

**4.2 三人称視点カメラの回転**

　この節では三人称視点カメラの回転を通してベクトルの回転のさせ方を学んでみましょう。三人称視点のカメラとは下記のようなカメラのことをいいます。

カメラ

　　　　　　　　プレイヤー

4.2.1 Y軸周りの回転

　では、まずカメラをY軸周りに回転させる方法を見てみましょう。 下の図は真上から見た図になります。

　　　　　　　　　　　　　　　　　　　視点

注視点

カメラがY軸周りに回るということは、カメラの視点が注視点を中心にして回転することを意味しています。

**視点**

**回転後の視点**

toPos

視点を回転させるには、注視点から視点に向かって伸びるベクトルを回転させて、視点を計算してやればいいのです。

注視点から視点に向かうベクトルは下記のプログラムで求めることができます。

|  |
| --- |
| //注視点をtarget、視点をposとして、注視点から視点に向かうベクトルをtoPosとすると、  D3DXVECTOR3 toPos = pos – target; |

このベクトルをY軸周りに10度回転させるには、Y軸周りに10度回転する行列を使って、ベクトルと乗算してやればいいことになります。

|  |
| --- |
| //Y軸周りに10度回転する回転行列を作成。  D3DXMATRIX mRot;  D3DXMatrixRotationY(&mRot, D3DXToRadian(10.0f)); //第二引数はラジアン単位。  //ベクトルを回転させる。  //第一引数は計算結果の格納先。D3DXVECTOR4型なのに注意。  //第二引数は回転元となるベクトル。  //第三引数は回転行列。  D3DXVec3TransformCoord(&toPos, &toPos, &mRot); |

そして、toPosを回転させることができたら、このベクトルと注視点を加算したものを視点とすればいいのです。

|  |
| --- |
| pos = target + toPos; |

4.2.2 X軸周りの回転

　続いて、カメラのX軸周りの回転を見ていきましょう。ここでいうX軸周りの回転とは下記の回転を指します。

　　　　　　　　　　　　　　　　　　　　　　　　　視点

　　　　　　　 注視点

この回転も注視点から視点に向かうベクトルを回転させることに変わりはありません。先ほどと違う点は、回転させる軸の求め方です。先程はY軸周り固定でしたが、今回はカメラのY軸周りの回転を考慮して、回転軸を計算する必要があります。次のページの図を見てみてください。

真上から見た図

　　　　　　　　　　　　　　　 視点1

視点2

視点１の場合はこの軸で回す

視点2の場合はこの軸で回す

このように回す軸が変わることになります。この回転軸を求めるためには外積を活用します。外積とは下記のように定義されるものです。

|  |
| --- |
| ベクトルv0、v1の外積の結果をv2とすると  v2.x = v0.y×v1.z － v0.z×v1.y  v2.y = v0.z×v1.x － v0.x×v1.z  v2.z = v0.x×v1.y － v0.y×v1.x  となる。 |

ゲームでよく使われる、外積の特性に「外積の結果は二つのベクトルの直行するベクトルになる」というものがあります。直行とはなす角が90度で交わるということです。

この性質を使うことで、視点を回転させるベクトルが計算できます。

まず、注視点から視点に向かうベクトルを計算します。そしてそのベクトルと上方向のベクトルと外積を計算します。この結果が先ほど見た図の回転軸になるのです。

|  |
| --- |
| //注視点をtarget、視点をposとして、注視点から視点に向かうベクトルをtoPosとすると、  D3DXVECTOR3 toPos = pos – target;  D3DXVECTOR3 vUP( 0.0f, 1.0f, 0.0f );  //外積を計算して、回転軸を求める。  D3DXVECTOR3 vRotAxis;  //第一引数が計算結果の格納先。  ５//第二引数と第三引数が外積に使用されるベクトル。  D3DXVec3Cross(&vRotAxis, &toPos, &vUp); |

あとは、この軸周りに回転する行列を作成して、toPosを回転すればいいのです。

|  |
| --- |
| D3DXMATRIX mRot;  //任意の軸周りの回転行列を作成。  D3DXMatrixRotationAxis(&mRot, &vRotAxis,D3DXToRadian(10.0f));  //ベクトルを回転させる。  //第一引数は計算結果の格納先。D3DXVECTOR4型なのに注意。  //第二引数は回転元となるベクトル。  //第三引数は回転行列。  D3DXVec3TransformCoord(&toPos, &toPos, &mRot);  //最後にtoPosとtargetから視点を計算する。  pos = target + toPos; |

**実習**

1. TPSカメラ/Questionを改造して、キーボードの左右のキーの入力でカメラがY軸回転をするようにしなさい。
2. TPSカメラ/Questionを改造して、キーボードの上下のキーの入力でカメラがX軸回転をするようにしなさい。
3. ②の実習でできたカメラの上下の回転に可動域を設定して、真上と真下にカメラが来ないようにしなさい。

**5 XInput**

XInputはWindows向けコントローラ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は使用できません。この節ではXInputの使い方と仮想キーについて学びましょう。

**5.1 スタティックリンクライブラリのリンク**

　XInputを使用するためにはスタティックリンクライブラリをリンクする必要があります。

下記の動画を参考にして、あなたのプロジェクトにスタティックリンクライブラリを追加してください。

**5.2 実習**

XInput/Questionを使用してxboxコントローラーが刺さっていない場合はキーボードでエミュレートができるようにlib/Pad.cppを改造しなさい。

(ヒント tkEngine/Input/tkPad.h、tkEngine/Input/tkPad.cppを参考にしなさい。)

**6　レベルエディタ**

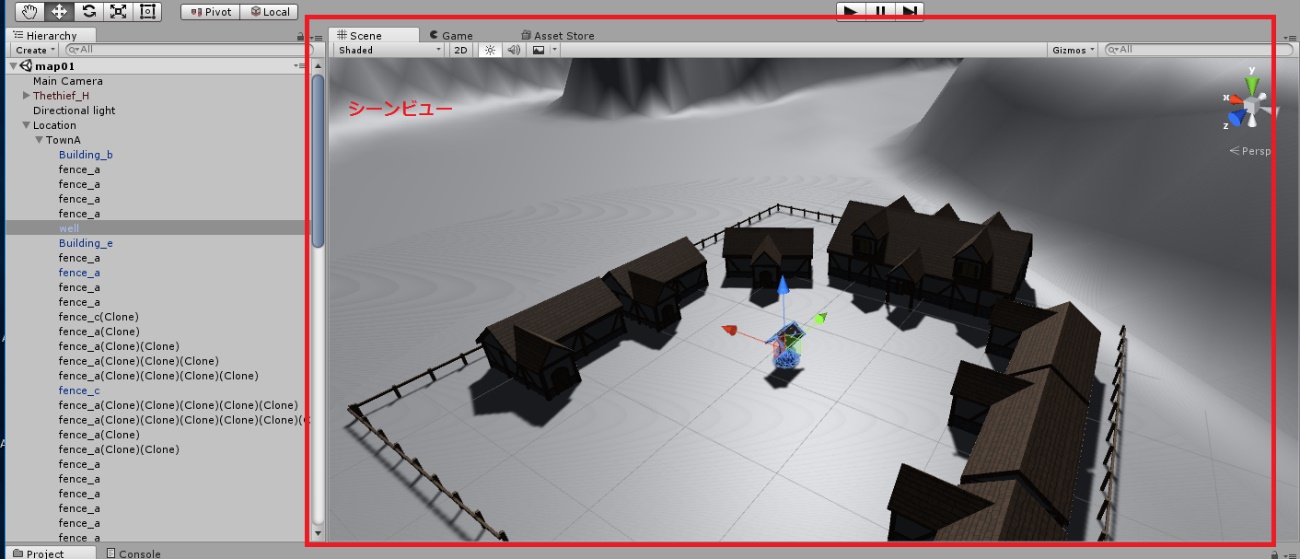
ゲームを作る際に必須のツールとしてレベルエディタがあります。このチャプターではUnityのレベルエディタを拡張してDirectX製のゲームでも使用できるようにしてみましょう。

**6.1 レベルエディタとは？**

レベルエディタとはマップを作るためのエディタのことをさします。レベルという言葉は日本では誤解されて広まっていますが、レベル＝ステージを意味します。つまりレベルエディタとはステージを作成するエディタということです。

**6.2 Unityのエディタ拡張**

　 UnityやUnrealEngineといった商用のツールには元々レベルエディタの機能が搭載されています。シーンビューにオブジェクトを配置して、マップを作ることができますよね？



あれがレベルエディタです。Unityなどのエンジンが登場する前、ほとんど全てのゲーム会社が内製エンジンを作っていた時代は、各社独自のレベルエディタを作成していました。3ds MaxやMayaのプラグインを作成したり、Windowsアプリとしてレベルエディタを作成したりなどして、オブジェクトの配置情報を出力していました。

　今回はUnityのエディタ拡張という機能を使って、Unityのシーンに配置されたオブジェクトの情報をDirectXで使える形で出力する方法を見ていきましょう。

**6.3 tkTool**

DirectX\_2/tkToolをUnityエディタで開いてください。このUnityプロジェクトを使えばシーンに配置されたオブジェクトの情報をC++で使える形で出力することができます。

**6.3.1 エディタ拡張**

　この内容は知識として知っている程度で構いません。読み終わったら忘れてしまっても大丈夫です。Unityエディタはかなり自由にエディタの拡張を行うことができ、エディタの拡張をC#スクリプトで行うことができます。とても簡単に拡張ができますので、何かツールを作りたいなと思ったときに、Unityを使ってみるというのはいいアイディアだと思います。

　tkTools/Assets/Editor/OutputLocationInfo.csというファイルを開いてみてください。これがシーンの情報をC++で使える形で出力しているスクリプトです。

OutputLocationInfo.cs(25行目～60行目)

|  |
| --- |
| {  GameObject location = GameObject.Find("Location");  Transform[] locs = location.GetComponentsInChildren<Transform>();  string outputTxt = "";  //マップチップを出力。  foreach (Transform tr in locs)  {  MapChip mapChip = tr.GetComponent<MapChip>();  if (!mapChip)  {  //マップチップが取得できないオブジェクトは無視。  continue;  }  string modelName = TrimModelName(tr.name);  outputTxt += string.Format("//{0}\n", modelName);  outputTxt += "{\n";  outputTxt += string.Format("\t\"{0}\",\n", modelName);  outputTxt += string.Format("\tD3DXVECTOR3({0:f}f, {1:f}f, {2:f}f), //平行移動\n", tr.position.x, tr.position.y, tr.position.z);  Quaternion rot = tr.localRotation \* mapChip.invRot;  rot \*= tr.parent.localRotation;  outputTxt += string.Format("\tD3DXQUATERNION({0:f}f, {1:f}f, {2:f}f, {3:f}f ), //回転\n", rot.x, rot.y, rot.z, rot.w);  outputTxt += "},\n";  }  try  {  using (StreamWriter sw = new StreamWriter(outputLocFilePath, false, Encoding.UTF8))  {  sw.Write(outputTxt);  }  }  catch { }  outputTxt = "";  } |

26行目を見てみてください。Locationというゲームオブジェクトを検索しています。27行目ではLocationのゲームオブジェクトの子供たちを取得しています。そして、30行目～47行目のループで子供たちの配置情報を出力しています。ここで作成された配置情報はC++のヘッダーファイルの形式で出力されています。

**6.4 出力されたヘッダーファイルをC++で使うには？**

では、出力されたヘッダーファイルの情報をどうやってC++側で使うのか見ていきましょう。

CollisionDemo/game/map.cppを開いてください。

Map.cpp(5行目～7行目)

|  |
| --- |
| static SMapChipLocInfo mapChipInfo[] = {  #include "locationInfo.h"  }; |

とても奇妙なコードが書かれています。今まで#includeをこのように使ったことはないのではないでしょうか？このトリックを理解するためには#includeの本質を理解する必要があります。

**6.4.1 プリプロセッサ**

　これまで、プログラムを編集するとVisualStudioの機能(F5やCtrl+F5とか)を使って実行ファイルをビルドしていたと思います。さて実は実行ファイルが出来上がるまでに複数のソフトウェアが実行されているのは知っていたでしょうか？実行ファイルが出来上がるまでに次のようなソフトウェアが実行されています・

1. プリプロセッサ

#includeや#pragmaなどのプリプロセッサキーワードを解析してソースコードを整形する。

1. コンパイラ

ソースファイルをマシン語に翻訳して、objファイルを作成する。

1. リンカ

2で作成されたobjファイルやスタティックリンクライブラリを結合して実行ファイルを作成する。

このように、exeファイルが出来上がるまでにいくつかのステップがあります。今回は①のプリプロセッサについて見ていきましょう。

**6.4.2 ソースファイルの整形**

　ソースコードに記述されている#から始まるキーワードはプリプロセッサ命令と呼ばれるものです。例えば次のようなコードの場合

|  |
| --- |
| #define VALUE 10  void main()  {  int a = VALUE;  } |

プリプロセッサによって次のように整形されます。

|  |
| --- |
| Void main()  {  int a = 10;  } |

では#includeはどのような整形が行われるのでしょうか？実は#includeは**指定されているファイルの中身をソースファイルに展開する**ということを行います。つまる、ヘッダーファイルに記述されていることを、そのままコピー＆ペーストしてくれるのです。

例えばlocationInfo.hの中身が次のようになっているとします。

LocationInfo.h

|  |
| --- |
| //Building\_b  {  "Building\_b",  D3DXVECTOR3(11.72f, 1.70f, -9.14f), //平行移動  D3DXQUATERNION(0.00f, 0.00f, 0.00f, 0.00f ), //回転  },  //fence\_a  {  "fence\_a",  D3DXVECTOR3(-19.36f, 0.35f, 34.94f), //平行移動  D3DXQUATERNION(0.00f, 0.00f, 0.00f, 0.00f ), //回転  },  //fence\_a  {  "fence\_a",  D3DXVECTOR3(-16.36f, 0.35f, 34.94f), //平行移動  D3DXQUATERNION(0.00f, 0.00f, 0.00f, 0.00f ), //回転  }, |

これのファイルを次のように#includeした場合、

|  |
| --- |
| static SMapChipLocInfo mapChipInfo[] = {  #include "locationInfo.h"  }; |

プリプロセッサによって次のようにコードが整形されます。

|  |
| --- |
| static SMapChipLocInfo mapChipInfo[] = {  //Building\_b  {  "Building\_b",  D3DXVECTOR3(11.72f, 1.70f, -9.14f), //平行移動  D3DXQUATERNION(0.00f, 0.00f, 0.00f, 0.00f ), //回転  },  //fence\_a  {  "fence\_a",  D3DXVECTOR3(-19.36f, 0.35f, 34.94f), //平行移動  D3DXQUATERNION(0.00f, 0.00f, 0.00f, 0.00f ), //回転  },  //fence\_a  {  "fence\_a",  D3DXVECTOR3(-16.36f, 0.35f, 34.94f), //平行移動  D3DXQUATERNION(0.00f, 0.00f, 0.00f, 0.00f ), //回転  },  }; |

これでSMapChipLocInfoの配列テーブルが出来上がりました。

**6.5 マップの構築**

ではmapChipInfoテーブルを使用して、マップを構築している処理を見ていきましょう。

CollisionDemo/game/Map.cppを開いてください。

Map.cpp(17行目～30行目)

|  |
| --- |
| void Map::Init()  {  //配置されているオブジェクトの数を計算。  int numObject = sizeof(mapChipInfo) / sizeof(mapChipInfo[0]);  for (int i = 0; i < numObject; i++) {  //オブジェクトを一個ずつロードしていく。  //まずはMapChipのインスタンスを動的に生成。  MapChip\* mapChip = new MapChip;  //マップチップの情報を渡して初期化。  mapChip->Init(mapChipInfo[i]);  //動的配列にプッシュ。  mapChipList.push\_back(mapChip);  }  } |

20行目でsizeof演算子を使用して、テーブルの大きさを求めています。21行目からのループで配置情報を一つずつ参照しながら、マップチップを作成していっています。28行目に生成したMapChipのインスタンスを動的配列に追加していることに注意してください。

**6.6 マップチップ**

　続いてマップチップについて見ていきます。マップチップは配置されているオブジェクト一つ分を表すクラスです。そのため、Init関数で渡された配置情報をもとにモデルのロードなどを行っています。

MapChip.cpp(19行目～25行目)

|  |
| --- |
| //読み込むモデルのファイルパスを作成。  char modelPath[256];  sprintf(modelPath, "Assets/modelData/%s.x", locInfo.modelName);  //モデルをロード。  modelData.LoadModelData(modelPath, NULL);  //ロードしたモデルデータを使ってSkinModelを初期化。  model.Init(&modelData); |

**6.7 実習**

tkToolを使用して、マップの配置を自由に変更しなさい。tkToolの使い方は「動画/tkTool使用方法.mp4」を参照しなさい。