**Chapter 1 　ウィンドウの作成**

DirectX11を使用してゲームを作るためには、まずウィンドウを作成する必要があります。このチャプターではウィンドウを作成する方法を学んでいきましょう。使用するサンプルプログラムはLesson\_00です。

**1.1 ウィンドウクラス**

ウィンドウを作成するためにはウィンドウクラスを登録する必要があります。(注意:

ここでいうクラスはC++のクラスとは一切関係がないので注意してください。単にマイクロソフトがウィンドウクラスと呼んでいるだけです。)

ウィンドウクラスの登録はWNDCLASSEXという構造体とRegisterClassExという関数を使用して下記のように記述します。

Lesson\_00/main.cpp(33行目)

|  |
| --- |
| //ウィンドウクラスのパラメータを設定(単なる構造体の変数の初期化です。)  WNDCLASSEX wc =  {  sizeof(WNDCLASSEX), //構造体のサイズ。  CS\_CLASSDC, //ウィンドウのスタイル。  　　　　　　 //ここの指定でスクロールバーをつけたりできるが、  　　　　　　 //ゲームでは不要なのでCS\_CLASSDCでよい。  MsgProc, //メッセージプロシージャ(後述)  0, //0でいい。  0, //0でいい。  GetModuleHandle(NULL), //このクラスのためのウインドウプロシージャがあるインスタンスハンドル。  //何も気にしなくてよい。  NULL, //アイコンのハンドル。アイコンを変えたい場合ここを変更する。  //とりあえずこれでいい。  NULL, //マウスカーソルのハンドル。NULLの場合はデフォルト。  NULL, //ウィンドウの背景色。NULLの場合はデフォルト。  NULL, //メニュー名。NULLでいい。  TEXT("Sample\_00"), //ウィンドウクラスに付ける名前。  NULL  };  //ウィンドウクラスの登録。  RegisterClassEx(&wc); |

これでウィンドウクラスが登録できました。

**1.2 ウィンドウの作成**

では続いて、1.1で作成したウィンドウクラスを使用してウィンドウを作成する方法を見ていきましょう。ウィンドウの作成にはCreateWindowという関数を使用します。

Lesson\_00/ main .cpp(54行目)

|  |
| --- |
| // ウィンドウの作成。  g\_hWnd = CreateWindow(  TEXT("Sample\_00"), //使用するウィンドウクラスの名前。  //先ほど作成したウィンドウクラスと同じ名前にする。  TEXT("Sample\_00"), //ウィンドウの名前。ウィンドウクラスの名前と別名でもよい。  WS\_OVERLAPPEDWINDOW, //ウィンドウスタイル。ゲームでは基本的にWS\_OVERLAPPEDWINDOWでいい、  0, //ウィンドウの初期X座標。  0, //ウィンドウの初期Y座標。  500, //ウィンドウの幅。  500, //ウィンドウの高さ。  NULL, //親ウィンドウ。ゲームでは基本的にNULLでいい。  NULL, //メニュー。今はNULLでいい。  hInstance, //アプリケーションのインスタンス。  NULL  ); |

CreateWindow関数は作成したウィンドウのハンドルを返してきます(ウィンドウのアドレスを返してくると考えて構いません)。このウィンドウハンドルは後々使用することになるので、グローバル変数のg\_hWndに記録しています。

ウィンドウの作成ができたら、ShowWindow関数を使用してウィンドウを表示します。

Lesson\_00/ main .cpp(69行目)

|  |
| --- |
| ShowWindow(g\_hWnd, nCmdShow); |

**1.3 メッセージループとメッセージプロシージャ**

ウィンドウアプリケーションのようなGUIアプリはコンソールアプリケーション(CUIアプリ)と異なり、ユーザーが明示的に終了させるまで、永続的にアプリケーションが起動しています。なので、起動したらすぐにアプリケーションが終了するのではなく、ユーザーがアプリケーションを閉じるまで、ずっと実行し続ける必要があります。これを実現するためのものがメッセージループです。サンプルプログラムにはShowWindowを実行した後で、下記のようなループがあります。

Lesson\_00/main.cpp(72行目)

|  |
| --- |
| //メッセージ構造体の変数msgを初期化。  MSG msg = { 0 };  while (WM\_QUIT != msg.message) //メッセージループ  {  //ウィンドウからのメッセージを受け取る。  これがメッセージループ  if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))  {  TranslateMessage(&msg);  DispatchMessage(&msg);  }  } |

**1.3.1 メッセージプロシージャ**

　ウィンドウズプログラムではユーザーの操作に対して、様々なアクションを起こす必要があります。このユーザーの操作によるメッセージを受け取っているのが、メッセージループ内のPeekMessage、TranslateMessage、DispatchMessageという関数です。この三つの関数を使用して、ユーザーからのメッセージを受けとると、メッセージプロシージャという特殊な関数が呼び出されます。これが1.1のウィンドウクラスの作成の時に登録した、メッセージプロシージャです。メッセージプロシージャはアプリケーションが独自に作成する必要がある関数で、アプリケーションごとに実装されています。では具体的なコードを見てみましょう。

Lesson\_00/main.cpp(10行目)

|  |
| --- |
| //メッセージプロシージャ。  //hWndがメッセージを送ってきたウィンドウのハンドル。  //msgがメッセージの種類。  //wParamとlParamは引数。今は気にしなくてよい。  LRESULT CALLBACK MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)  {  //送られてきたメッセージで処理を分岐させる。  switch (msg)  {  case WM\_DESTROY:  PostQuitMessage(0);  break;  case WM\_LBUTTONDOWN:  //マウスの左ボタンが押された。  MessageBox(hWnd, "マウスの左ボタンが押されたよ。", "お知らせ", MB\_OK);  break;  default:  return DefWindowProc(hWnd, msg, wParam, lParam);  }  return 0;  } |

これがウィンドウクラスを登録する時に指定していたメッセージプロシージャです。引数のmsgにユーザーが行った操作に関するメッセージが記録されています。このサンプルではWM\_LBUTTONDONW、マウスの左ボタンが押されたときのメッセージ、が送られてきたときにMessageBoxという関数を使用して、メッセージを表示しています。

**1.4 章末テスト**

　①　ウィンドウのサイズを1280×720に変更しなさい。

　②　ウィンドウの名前をGameDemoに変更しなさい。

　③　マウスの右ボタンが押されたときに「マウスの右ボタンが押されたよ」というメッセージを表示しなさい。「マウスの右ボタンが押された」というメッセージはWM\_RBUTTONDOWNです。

**Chapter 2 　DirectXの初期化と終了処理**

Chapter１でウィンドウを作成することができるようになりました。しかし、まだDirectXの機能を利用することはできません。このチャプターではDirectXを利用できるようにするための方法と、利用が終わるときの終了処理について学んでいきましょう。

**2.1 D3DデバイスとD3Dデバイスコンテキストとスワップチェインの作成**

DirectXを使用できるようにするためには、まずD3DデバイスとD3Dデバイスコンテキストとスワップチェインを作成する必要があります。

**2.1.1 D3Dデバイス**

　D3DデバイスはGPUなどのハードウェアにアクセスするためのインターフェースのようなものです。D3Dデバイスを作成することで、GPUに対して命令を行うことができるようになります。

**2.1.2 D3Dデバイスコンテキスト**

　D3DデバイスコンテキストはGPUへの命令を記録するためのものです。DirectXではD3Dデバイスに直接命令を送るのではなく、D3Dデバイスコンテキストを介してD3Dデバイスに命令を送ります。D3Dデバイスコンテキストは中間管理職のようなものです。

**2.1.2 スワップチェイン**

　スワップチェインは描画した絵を切り替える機能を提供するものです。DirectXを利用して絵をスクリーンに描画する場合、直接スクリーンに絵を描き込むのではなく(このスクリーンの映っている絵のことをフロントバッファといいます)、バックバッファと呼ばれる、現在は画面に表示されていないバッファに絵を描き込みます。そして絵の描き込みがすべて完了したら、バックバッファとフロントバッファを入れ替えて、描画した絵をスクリーンに映します。

例えば10フレーム目の状態

**モニター**



　　　　　　フロントバッファ　　　　　　　　　　　　バックバッファ

DirectXはバックバッファに11フレームの絵を描き込む。

11フレームの絵の描画がすべて完了すると、スワップチェインを使用してフロントバッファとバックバッファを入れ替えます。

11フレーム目の状態。

　　　　　　　　　　　　　　　　　　モニタ



　　　　　　　バックバッファ　　　　　　　　　　　　フロントバッファ

このようにすることによって、描画している最中の絵をユーザーに見せることがなくなり、ティアリングと呼ばれる現象を軽減させることができます。そして、この機能を提供するのがスワップチェインです。

D3Dデバイスとスワップチェインを作成するには、DXGI\_SWAP\_CHAIN\_DESC構造体とD3D11CreateDeviceAndSwapChainを使用します。

Lesson\_01/main.cpp(22行目)

|  |
| --- |
| //スワップチェインを作成するための情報を設定する。  DXGI\_SWAP\_CHAIN\_DESC sd;  ZeroMemory(&sd, sizeof(sd));  sd.BufferCount = 1; //スワップチェインのバッファ数。通常は１。  sd.BufferDesc.Width = 500; //フレームバッファの幅。  sd.BufferDesc.Height = 500; //フレームバッファの高さ。  sd.BufferDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM; //フレームバッファのフォーマット。  //R8G8B8A8の32bit。  sd.BufferDesc.RefreshRate.Numerator = 60; //モニタのリフレッシュレート。  //これがバックバッファとフロントバッファを  //入れ替えるタイミングになる。  sd.BufferDesc.RefreshRate.Denominator = 1; //２にしたら最大30fpsになる。1でよい。  sd.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT; //DXGI\_USAGE\_RENDER\_TARGET\_OUTPUTでよい。  sd.OutputWindow = g\_hWnd; //出力先のウィンドウハンドル。  sd.SampleDesc.Count = 1; //1でいい。  sd.SampleDesc.Quality = 0; //MSAAなし。0でいい。  sd.Windowed = TRUE; //ウィンドウモード。TRUEでよい。  //利用するDirectXの機能セット。DirectX10以上に対応しているGPUを利用可能とする。  //この配列はD3D11CreateDeviceAndSwapChainの引数として使う。  D3D\_FEATURE\_LEVEL featureLevels[] =  {  D3D\_FEATURE\_LEVEL\_11\_0,  D3D\_FEATURE\_LEVEL\_10\_1,  D3D\_FEATURE\_LEVEL\_10\_0,  };  //D3Dデバイスとスワップチェインを作成する。  D3D11CreateDeviceAndSwapChain(  NULL, //NULLでいい。  D3D\_DRIVER\_TYPE\_HARDWARE, //D3Dデバイスがアクセスするドライバーの種類。  //基本的にD3D\_DRIVER\_TYPE\_HARDWAREを指定すればよい。  NULL, //NULLでいい。  0, //０でいい。  featureLevels, //D3Dデバイスのターゲットとなる機能セットを指定する。  //今回のサンプルはDirectX10以上をサポートするので、  //それらを含むD3D\_FEATURE\_LEVELの配列を渡す。  sizeof(featureLevels)/sizeof(featureLevels[0]), //機能セットの数。  D3D11\_SDK\_VERSION, //使用するDirectXのバージョン。  //D3D11\_SDK\_VERSIONを指定すればよい。  &sd, //スワップチェインを作成するための情報。  &g\_pSwapChain, //作成したスワップチェインのアドレスの格納先。  &g\_pd3dDevice, //作成したD3Dデバイスのアドレスの格納先。  &g\_featureLevel, //使用される機能セットの格納先。  &g\_pd3dDeviceContext //作成したD3Dデバイスコンテキストのアドレスの格納先。  ); |

**2.2 レンダリングターゲットの作成**

　続いて、スワップチェインに設定されいてるバックバッファが書き込み先(レンダリングターゲット)であることをGPUに教えてやる必要があります。そのために、まずレンダリングターゲットを作成します。

Lesson\_01/main.cpp(65行目)

|  |
| --- |
| //書き込み先になるレンダリングターゲットを作成。  ID3D11Texture2D\* pBackBuffer = NULL;  g\_pSwapChain->GetBuffer(0, \_\_uuidof(ID3D11Texture2D), (LPVOID\*)&pBackBuffer);  g\_pd3dDevice->CreateRenderTargetView(pBackBuffer, NULL, &g\_backBuffer);  pBackBuffer->Release(); |

**2.3 ゲームループの作成**

　ゲームはCUIアプリと異なりユーザーが明示的に終了するまで、永続的に実行される必要があります。そして、60fpsのゲームであれば1秒間に60回、30fpsのゲームであれば1秒間に30回描画を行う必要があります。そこで、ゲームループと呼ばれるゲームが起動している間、ずっと実行されるループを作成しましょう。Chapter1でメッセージループという似たようなものを作成したと思います。これを利用しましょう。

Lesson\_01/main.cpp(185行目)

|  |
| --- |
| while (WM\_QUIT != msg.message) //メッセージループ  {  //ウィンドウからのメッセージを受け取る。  if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))  {  TranslateMessage(&msg);  DispatchMessage(&msg);  }  else {  //ウィンドウからのメッセージがないので、ゲームの処理を実行する。  GameUpdate();  }  } |

メッセージループにウィンドウからのメッセージが存在しないときはGameUpateという関数を呼び出すコードを追加しました。

では、GameUpdate関数を見てみましょう。

Lesson\_01/main.cpp(156行目)

|  |
| --- |
| ///////////////////////////////////////////////////////////////////  // 毎フレーム呼ばれるゲームの更新処理。  ///////////////////////////////////////////////////////////////////  void GameUpdate()  {  float ClearColor[4] = { 0.5f, 0.5f, 0.5f, 1.0f }; //red,green,blue,alpha  //描き込み先をバックバッファにする。  g\_pd3dDeviceContext->OMSetRenderTargets(1, &g\_backBuffer, NULL);  //バックバッファを灰色で塗りつぶす。  g\_pd3dDeviceContext->ClearRenderTargetView(g\_backBuffer, ClearColor);  /////////////////////////////////////////////////  // ここに3Dモデルなどを描画するコードを書いていく。  /////////////////////////////////////////////////  //バックバッファとフロントバッファを入れ替える。  g\_pSwapChain->Present(0, 0);  } |

これでバックバッファが灰色にクリアされて、画面に表示されます。

**2.4　DirectXの終了処理**

ゲームが終了したときに、正しくDirectXの終了処理を実行する必要があります。今回はDirectXの終了処理を行うためにReleaseDirectXという関数を用意しました。

Lesson\_01/main.cpp(71行目)

|  |
| --- |
| ///////////////////////////////////////////////////////////////////  // DirectXの終了処理。  ///////////////////////////////////////////////////////////////////  void ReleaseDirectX()  {  if (g\_backBuffer != NULL) {  g\_backBuffer->Release();  }  if (g\_pSwapChain != NULL) {  g\_pSwapChain->Release();  }  if (g\_pd3dDeviceContext != NULL) {  g\_pd3dDeviceContext->Release();  }  if (g\_pd3dDevice != NULL) {  g\_pd3dDevice->Release();  }  } |

あとは、ReleaseDirectX関数をゲームが終了する時に正しく呼び出されるようにしましょう。ゲームが終了する時はメッセージプロシージャーにWM\_DESTROYというメッセージが送られてくるので、そこでこの関数を呼び出しましょう。

Lesson\_01/main.cpp(95行目)

|  |
| --- |
| LRESULT CALLBACK MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)  {  //送られてきたメッセージで処理を分岐させる。  switch (msg)  {  case WM\_DESTROY:  **ReleaseDirectX();**  PostQuitMessage(0);  break;  default:  return DefWindowProc(hWnd, msg, wParam, lParam);  }  return 0;  } |

**2.5 章末テスト**

　①　画面クリアのカラーを赤にしなさい。光の三原色、RGBで赤はR = 1.0f、G = 0.0、Ｂ＝0.0で表されます。

**Chapter 3 ポリゴン**

**3.1 ポリゴンとは**

　ポリゴンとは3Dモデルを構成する最小要素の事で、ゲームの世界では多くの場合で３角形ポリゴンのことを指します。下記の図は3dsMaxを使用してUnityちゃんのポリゴンを可視化しているものです。



このように3Dモデルというのは３角形のポリゴンの集合体として表現されています。

**3.2 頂点バッファ**

　3Dモデルを構成する最小単位はポリゴンですが、そのポリゴンを定義するためにはポリゴンの頂点座標を決める必要があります。

頂点１

座標(0.0f, 10.0f, 0.0f)

頂点3

座標(-10.0f, 0.0f, 0.0f)

頂点2

座標(10.0f, 0.0f, 0.0f)

このように、三角形を構成する３頂点の座標を決定してやれば、ポリゴンを定義することができます。つまり、3Dモデルはポリゴンの数×３の頂点によって表現されていることになります。この頂点の集合が**頂点バッファ**と呼ばれます。

**3.2.1 頂点バッファの作成**

ではLesson\_02のサンプルコードを使用して、頂点バッファの作成を見ていきましょう。

Lesson\_02/main.cpp(196行目)

|  |
| --- |
| void InitTrianglePolygon()  {  //3頂点を定義する。これが頂点座標。  SVertex vertex[3] = {  { 0.5f, -0.5f, 0.0f, 1.0f },  { 0.0f, 0.5f, 0.0f, 1.0f },  { -0.5f, -0.5f, 0.0f, 1.0f },  };  //上で定義した頂点を使用して頂点バッファを作成する。  //頂点バッファを作成するためにはD3D11\_BUFFER\_DESCと  //D3D11\_SUBRESOURCE\_DATAを設定する必要がある。  D3D11\_BUFFER\_DESC bd;  ZeroMemory(&bd, sizeof(bd)); //構造体を0で初期化する。  bd.Usage = D3D11\_USAGE\_DEFAULT; //バッファーで想定されている読み込み、  //および書き込みの方法。  //取りあえずはD3D11\_USAGE\_DEFAULTでよい。  bd.ByteWidth = sizeof(vertex); //頂点バッファのサイズ。頂点のサイズ×頂点数となる。  bd.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER; //これから作成するバッファが  //頂点バッファであることを指定する。  D3D11\_SUBRESOURCE\_DATA InitData;  ZeroMemory(&InitData, sizeof(InitData));  InitData.pSysMem = vertex;  //頂点バッファの作成。  g\_pd3dDevice->CreateBuffer(&bd, &InitData, &g\_vertexBuffer);  //シェーダーをロード。(今は気にしなくてよい)  g\_effect.Load("Assets/shader/primitive.fx");  } |

作成された頂点バッファにアクセスするためのインターフェースはg\_vertexBufferに設定されます。

続いて作成した頂点バッファを使用して三角形を描画するコードを見ていきましょう。

Lesson\_02/main.cpp(238行目)

|  |
| --- |
| g\_effect.BeginRender(); //描画開始。  unsigned int vertexSize = sizeof(SVertex); //頂点のサイズ。  unsigned int offset = 0;  g\_pd3dDeviceContext->IASetVertexBuffers( //頂点バッファを設定。  0, //StartSlot番号。今は0でいい。  1, //バッファの数。今は1でいい。  &g\_vertexBuffer, //頂点バッファ。  &vertexSize, //頂点のサイズ。  &offset //気にしなくてよい。  );  //プリミティブのトポロジーを設定する。(詳しくはLesson\_3で解説を行う。)  g\_pd3dDeviceContext->IASetPrimitiveTopology(D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);  g\_pd3dDeviceContext->Draw( //描画命令。  3, //頂点数。  0 //開始頂点番号。  ); |

**3.2.3 頂点バッファの開放**

　頂点バッファが不要になったら正しく開放を行いましょう。下記のようなコードで開放を行うことができます。

Lesson\_02/main.cpp(106行目)

|  |
| --- |
| if (g\_vertexBuffer != NULL) {  g\_vertexBuffer->Release();  } |

**3.3 章末テスト**

①　三角形を二倍の大きさにして下記のように表示されるようにしなさい。



②　頂点バッファをもう一つ用意して、三角形を二つ描画して、下記の図のようになるようにしなさい。



課題２のヒント

① ID3D11Buffer\*型の変数を一つ追加する。例えばg\_vertexBuffer2のようなもの。

　② InitTrianglePolygonを参考にして、二つ目の頂点バッファを作成するための

　　 InitTrianglePolygon2のような関数を作成して、適切な場所から呼び出す。

　③ GameUpdate関数の一つ目の三角形の描画処理を参考にして、二つ目の三角形を描画する。

チェック項目

|  |  |  |  |
| --- | --- | --- | --- |
| 課題１ | 実装できた。 |  | 25点 |
| 課題２ | 新しい頂点バッファを作成することができた。 |  | 25点 |
| 課題2 | 例にあるような表示を行うことができた。 |  | 25点 |
| 課題2 | 新しく作った頂点バッファを開放するコードが書けた |  | 25点 |

**3.3 インデックスバッファ**

　インデックスバッファの説明をする前に、頂点バッファのみを用いて4角形を描画する

方法を見てみましょう。

Lesson\_03\_00/main.cpp(200行目)

|  |
| --- |
| //四角形を描画するので、三角形ポリゴンが二枚必要になる。  //なので6頂点を定義する。これが頂点座標  SVertex vertex[6] = {  { -0.5f, -0.5f, 0.0f, 1.0f }, //頂点１  { -0.5f, 0.5f, 0.0f, 1.0f }, //頂点２  { 0.5f, -0.5f, 0.0f, 1.0f }, //頂点３  { 0.5f, -0.5f, 0.0f, 1.0f }, //頂点４  { -0.5f, 0.5f, 0.0f, 1.0f }, //頂点５  { 0.5f, 0.5f, 0.0f, 1.0f }, //頂点６  }; |

四角形の描画は下記のように2枚の三角形を描画していることになります



二つ三角形を描画して4角形を表現しているため

頂点データが6個あることが分かります。頂点が6個なので、頂点バッファのサイズは、頂点のサイズ(4バイト×4)×超点数(６)となるので、96バイトになります。

　しかし、ここで無駄なデータがあることに気付けるでしょうか？頂点２、５と頂点３，４は全く同じ座標になっています。しかし、この重複を除外できると効率が良くなりそうです。この重複を除外するために使われるものがインデックスバッファです。

　インデックスバッファとは、**三角形を描画するために使用する頂点番号の配列です**。例えば先ほどのような四角形を描画するケースを考えてみましょう。今回使用する頂点バッファは重複を除外するので、下記のように用意します。



そして、ポリゴンの描画に、何番の頂点を使用するのか？というものを指定するのものがインデックスバッファと呼ばれるものです。例えば、1枚目のポリゴンを描画する場合は１，２，３番の頂点を使用して、2枚目のポリゴンは２，３，４番の頂点を使用するということを指定するようなものです。

　では具体的にコードを見ていきましょう。まずはインデックスバッファを作成しているコードを見ていきます。

Lesson\_03\_01/main.cpp(226行目)

|  |
| --- |
| ///////////////////////////////////////////////////////////////////  // インデックスバッファの初期化。  ///////////////////////////////////////////////////////////////////  void InitIndexBuffer()  {  //これがインデックス。頂点番号。  int index[6] = {  0,1,2, //三角形一つ目  2,1,3 //三角形二つ目  };  //上で定義したインデックスを使用してインデックスバッファを作成する。  //インデックスバッファの作成も頂点バッファと同様に、  //D3D11\_BUFFER\_DESCとD3D11\_SUBRESOURCE\_DATAを設定する必要がある。  D3D11\_BUFFER\_DESC bd;  ZeroMemory(&bd, sizeof(bd)); //構造体を0で初期化する。  bd.Usage = D3D11\_USAGE\_DEFAULT; //バッファーで想定されている読み込み、書き込みの方法。  //取りあえずはD3D11\_USAGE\_DEFAULTでよい。  bd.ByteWidth = sizeof(index); //インデックスバッファのサイズ。  bd.BindFlags = D3D11\_BIND\_INDEX\_BUFFER; //これから作成するバッファが  //インデックスバッファであることを指定する。  D3D11\_SUBRESOURCE\_DATA InitData;  ZeroMemory(&InitData, sizeof(InitData));  InitData.pSysMem = index;  //インデックスバッファの作成。  g\_pd3dDevice->CreateBuffer(&bd, &InitData, &g\_indexBuffer);  } |

int型の要素数６の配列を使用して、インデックスバッファを作成しています。D3D11\_BUFFER\_DESCの設定が少し違うだけで、頂点バッファの作成と大きく違う点はありません。

　続いて頂点バッファの作成を見ていきましょう。

Lesson\_03\_01/main.cpp(197行目)

|  |
| --- |
| ///////////////////////////////////////////////////////////////////  // 頂点バッファの初期化。  ///////////////////////////////////////////////////////////////////  void InitVertexBuffer()  {  //上で定義した頂点を使用して頂点バッファを作成する。  //頂点バッファを作成するためにはD3D11\_BUFFER\_DESCとD3D11\_SUBRESOURCE\_DATAを設定する必要がある。  S**Vertex vertex[4] = {**  **{ -0.5f, -0.5f, 0.0f, 1.0f }, //頂点１**  **{ 0.5f, -0.5f, 0.0f, 1.0f }, //頂点２**  **{ -0.5f, 0.5f, 0.0f, 1.0f }, //頂点３**  **{ 0.5f, 0.5f, 0.0f, 1.0f }, //頂点４**  **};**  //上で定義した頂点を使用して頂点バッファを作成する。  //頂点バッファを作成するためにはD3D11\_BUFFER\_DESCとD3D11\_SUBRESOURCE\_DATAを設定する必要がある。  D3D11\_BUFFER\_DESC bd;  ZeroMemory(&bd, sizeof(bd)); //構造体を0で初期化する。  bd.Usage = D3D11\_USAGE\_DEFAULT; //バッファーで想定されている読み込みおよび書き込みの方法。  //取りあえずはD3D11\_USAGE\_DEFAULTでよい。  bd.ByteWidth = sizeof(vertex); //頂点バッファのサイズ。頂点のサイズ×頂点数となる。  bd.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER; //これから作成するバッファが  //頂点バッファであることを指定する。  D3D11\_SUBRESOURCE\_DATA InitData;  ZeroMemory(&InitData, sizeof(InitData));  InitData.pSysMem = vertex;  //頂点バッファの作成。  g\_pd3dDevice->CreateBuffer(&bd, &InitData, &g\_vertexBuffer);  } |

先ほどと異なり、頂点の数が4つになっていることに注目してください。その他に変更点はありません。続いて描画コードを見ていきましょう。

Lesson\_03\_01/main.cpp(265行目)

|  |
| --- |
| ///////////////////////////////////////////////////////////////////  // 毎フレーム呼ばれるゲームの更新処理。  ///////////////////////////////////////////////////////////////////  void GameUpdate()  {  float ClearColor[4] = { 0.5f, 0.5f, 0.5f, 1.0f }; //red,green,blue,alpha  //描き込み先をバックバッファにする。  g\_pd3dDeviceContext->OMSetRenderTargets(1, &g\_backBuffer, NULL);  //バックバッファを灰色で塗りつぶす。  g\_pd3dDeviceContext->ClearRenderTargetView(g\_backBuffer, ClearColor);    /////////////////////////////////////////////////  // ここに3Dモデルなどを描画するコードを書いていく。  /////////////////////////////////////////////////  g\_effect.BeginRender(); //描画開始。  unsigned int vertexSize = sizeof(SVertex); //頂点のサイズ。  unsigned int offset = 0;  g\_pd3dDeviceContext->IASetVertexBuffers( //頂点バッファを設定。  0, //StartSlot番号。今は0でいい。  1, //バッファの数。今は1でいい。  &g\_vertexBuffer, //頂点バッファ。  &vertexSize, //頂点のサイズ。  &offset //気にしなくてよい。  );  **g\_pd3dDeviceContext->IASetIndexBuffer( //インデックスバッファを設定。**  **g\_indexBuffer, //インデックスバッファ。**  **DXGI\_FORMAT\_R32\_UINT, //インデックスのフォーマット。**  **//今回は32bitなので、DXGI\_FORMAT\_R32\_UINTでいい。**  **0 //オフセット0でいい。**  **);**  //プリミティブのトポロジーは  //トライアングルストリップを設定する。(詳しくはLesson\_3で解説を行う。)  g\_pd3dDeviceContext->IASetPrimitiveTopology(D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);  **g\_pd3dDeviceContext->DrawIndexed( //描画命令。**  **6, //インデックス数。**  **0, //開始インデックス番号。0でいい。**  **0 //開始頂点番号。0でいい。**  **);**  //バックバッファとフロントバッファを入れ替える。  g\_pSwapChain->Present(0, 0);  } |

網掛けになっている箇所が、頂点バッファのみを使用して描画する場合との変更点になります。描画する前にインデックスバッファを設定して、描画コールははインデックスバッファを使用して描画を行う、DrawIndexedを使用しています。

**3.3.1 メモリ使用量の違い**

では、メモリ使用量がどの程度変わったのか見てみましょう。

**① インデックスバッファを使用しない場合**

頂点サイズ(4バイト×4)×頂点数(6個) = 96バイト

**② インデックスバッファを使用する場合**

　　頂点サイズ(4バイト×4)×頂点数(4個) + インデックスのサイズ(4バイト)×インデックスの数(4個) = 80バイト

このように、インデックスバッファを使用した場合の方が16バイト少なくなりました。四角形の場合はポリゴン数が少ないので、その差は大きくありませんが、最先端のゲームであれば、モデルのポリゴン数が10万を超えることは珍しいことではありません。このような場合にメモリ使用量に大きな差が生まれることになります。

**3.4 インデックスバッファの解放**

　では、最後にインデックスバッファの解放を見ていきましょう。

Lesson\_03\_01/main.cpp(107行目)

|  |
| --- |
| if (g\_indexBuffer != NULL) {  g\_indexBuffer->Release();  } |

頂点バッファと全く同じです。作成したものは不要になったら解放する。これはとても大事なことです。

**3.5 章末テスト**

Lesson\_03\_01を改造して、下記のような表示ができるようにしなさい。

****

チェック項目

|  |  |  |
| --- | --- | --- |
| 頂点バッファを拡張することができた。 |  | 30点 |
| インデックスバッファを拡張することができた。 |  | 30点 |
| 実装できた。 |  | 40点 |

**Chapter 4 テクスチャ**

**4.1 テクスチャとは**

　3DCGの世界でテクスチャというと多くの場合で、３Ｄモデルの模様を表す画像データのことをさします。下記の図はUnity社がフリーで提供している３ＤモデルのUnityちゃんと、そのテクスチャです。

　3Dモデル



テクスチャ



ここまで勉強してきたように、3Dモデルというのは３角形ポリゴンの集合体です。つまり3Dモデルの模様は、三角形のポリゴンにテクスチャの模様を貼り付けて表現しているのです。では、ポリゴンにテクスチャを貼り付ける方法を見ていきましょう。

**4.2 UV座標**

　ポリゴンにテクスチャを貼り付けるときはUV座標というものを使用します。UV座標とはテクスチャ座標とも呼ばれる下記のようなものです。

　 (0.0、0.0)



V

(1.0、1.0)

例えばUV座標が(0.2、0.5)の場合は下記のテクセル(画素)を指します。

**0.2**



**0.5**

このUV座標を使って、ポリゴンに貼り付ける模様を決定します。

**4.3 頂点UV座標**

　ではポリゴンにテクスチャを貼り付ける具体例を見ていきましょう。次のようにポリゴンにテクスチャを貼り付けたい場合を考えてみましょう。

**ポリゴン**



**テクスチャ**

****

このようにテクスチャを貼り付けるのは、下記のように頂点にUV座標を埋め込むことで実現できます。



( 1.0,　1.0 )

( 0.0,　1.0 )

( 0.5,　0.0 )

**4.4　サンプルコード**

　ではサンプルコードを見て、具体的にどのようにプログラムを書いていくのか見ていきましょう。

　クスチャを貼り付けるためには頂点にUV座標を埋め込む必要があります。Chapter3までは頂点に頂点座標のデータしかありませんでしたが、そこにUV座標のデータを追加しましょう。

Lesson\_04/main.cpp(25行目)

|  |
| --- |
| //頂点構造体。  struct SVertex {  float position[4]; //頂点座標。4要素なのは今は気にしない。  **float uv[2]; //UV座標。これがテクスチャ座標**  }; |

SVertex構造体にfloatの要素数２の配列uvを追加しています。

続いて頂点バッファの作成を見ていきましょう。

Lesson\_04/main.cpp(213行目)

|  |
| --- |
| SVertex vertex[4] = {  //頂点１  {  //座標　position[4]  -0.5f, -0.5f, 0.0f, 1.0f,  //UV座標 uv[2]  0.0f, 1.0f  },  //頂点２  {  //座標　position[4]  0.0f, 0.5f, 0.0f, 1.0f,  //UV座標 uv[2]  0.5f, 0.0f  },  //頂点３  {  //座標　position[4]  0.5f, -0.5f, 0.0f, 1.0f,  //UV座標 uv[2]  1.0f, 1.0f  },  //頂点４  {  //座標　position[4]  0.5f, 0.5f, 0.0f, 1.0f,  //UV座標 uv[2]  1.0f, 0.0f  },  }; |

頂点バッファの情報にuv座標が追加されています。続いて、貼り付けるテクスチャをロードするコードを見ていきましょう。

Lesson\_04/main.cpp(372行目)

|  |
| --- |
| DirectX::CreateDDSTextureFromFileEx(  g\_pd3dDevice, //D3Dデバイス。  L"Assets/mikyan.dds", //読み込む画像データのファイルパス。  0, //今は気にしなくてよい。  D3D11\_USAGE\_DEFAULT, //今は気にしなくてよい。  D3D11\_BIND\_SHADER\_RESOURCE, //今は気にしなくてよい。  0, //今は気にしなくてよい。  0, //今は気にしなくてよい。  false, //今は気にしなくてよい。  nullptr, //今は気にしなくてよい。  &g\_texture //読み込んだテクスチャに  //アクセスするためのインターフェースの格納先。  ); |

続いて描画コードです。

Lesson\_04/main.cpp(342行目)

|  |
| --- |
| //テクスチャを設定。  g\_pd3dDeviceContext->PSSetShaderResources(0, 1, &g\_texture); |

ポリゴンを描画する前に貼り付けるテクスチャをGPUに教えています。

後に終了コードです。テクスチャが不要になったらきちんと開放しましょう。

Lesson\_04/main.cpp(110行目)

|  |
| --- |
| if (g\_texture != NULL) {  g\_texture->Release();  } |

**4.5 章末テスト**

　① Lesson\_04のプログラムを改造して、下記のように表示できるようにしなさい。



② Lesson\_04のプログラムを改造して、下記のように表示できるようにしなさい。



チェック項目

|  |  |  |
| --- | --- | --- |
| ①が実装できた。 |  | 50点 |
| ②が実装できた。 |  | 50点 |

**Chapter 5　Spriteクラス**

Chapter4で板ポリにテクスチャを貼り付けて２Ｄを表示する方法を勉強しました。Chapter5では2D表示を行うためのSpriteクラスを作成して、2Dの表示を簡単に行えるようにしてみましょう。

**5.1 章末テスト**

　まずVisualStudioのソリューションエクスプローラーで、クラスを追加したいプロジェクトを選択して、右クリック→追加→クラスを選択します。



　クラス追加のダイアログが出てくるので、追加を選択してください。



クラス名にSpriteと入力して、完了を押してください。



これでプロジェクトにSprite.cppとSprite.hというソースファイルが追加されました。



続いてSprite.hを開いて、クラス定義を作成していきます。

Spriteクラスに四角形ポリゴンを表示するためのメンバ変数を下記のように追加します。



Effectクラスが分からないよ？ というエラーが出ているので、Effect.hをインクルードしましょう。



頂点バッファ、インデックスバッファ、サンプラステート作成やテクスチャのロードを行うためのInit関数を宣言します。



絵を表示するためのDraw関数も追加しましょう。



続いてSprite.cppにSpriteクラスの実装を書いていきましょう。まず、Init関数を実装します。(\*注意 頂点バッファ、インデックスバッファ、サンプラステートの作成はこちらで用意した関数を使用して作成しています。)



続いてDraw関数を実装します。

Sprite.cpp



Spriteクラスの実装の最後に、デストラクタに後始末のコードを追加しましょう。



これでSpriteクラスは完成です。しかし、クラスは作成しただけでは何もできません。では続いて、クラスのインスタンスを作成して、２Ｄを表示してみましょう。

まず、main.cppにSpriteクラスの変数を追加します(これがSpriteクラスのインスタンス)。



g\_spriteクラスを利用できるようにするために、きちんと初期化を行いましょう。



初期化が行えたので、後はDraw関数を呼び出して絵を描画しましょう。



**5.2 章末テスト**

　Lesson\_05を使って、ここまでの内容を実装しなさい。

　チェック項目

|  |  |
| --- | --- |
| Spriteクラスを追加出来た。 | 10点 |
| Spriteクラスにメンバ変数を追加出来た。 | 10点 |
| SpriteクラスにInit関数とDraw関数の宣言を追加出来た。 | 10点 |
| SpriteクラスのInit関数を実装できた。 | 10点 |
| SpriteクラスのDraw関数を実装できた。 | 10点 |
| Spriteクラスのデストラクタで頂点バッファなどのリソースを開放できた。 | 10点 |
| main.cppにSpriteクラスのインスタンスを追加出来た。 | 10点 |
| Spriteクラスのインスタンスを初期化することができた。 | 10点 |
| ２Ｄが表示された。 | 20点 |

**Chapter 6 複数のスプライトの表示**

　このチャプターではChapter5のプログラムを改造した、サンプルプログラムのLesson\_06を使用して、複数のスプライトを表示する実習を行います。Lesson\_06のSpriteクラスには下記の機能が追加されています。

　・画像の幅と高さの指定。

　・画像の座標、回転、拡大率、ピボットの指定。

では、一つずつ見ていきましょう。

**6.1 画像の幅と高さの指定**

　下記のようにSprite::Init関数に幅と高さのパラメータが追加されています。



Spriteクラスの初期化時に幅と高さを指定することで、画像の大きさを変更することができます。

**6.2 画像の座標、回転、拡大率、ピボットの指定**

6.2.1 座標

2Dを画面上に表示するためには、どこに表示するのかを指定する位置を指定する必要があります。Lesson\_06のSpriteクラスではUpdate関数が追加されていて、その第一引数が2Dの位置となります。

このSpriteクラスでは、位置の指定は下記のような座標系で行います。



座標の指定は3次元のベクトルデータで指定できますが、今回は2D表示なので、XとYしか使用しません。Ｚの値は0.0でも指定しておいてください。

6.2.2 回転

　回転にはクォータニオンを使用します。SpriteクラスのUpdate関数の第二引数に回転を表すクォータニオンを指定してください。

6.2.3 拡大

　拡大は3次元のベクトルデータで指定できますが、こちらも座標の場合と同様にXとYしか使用しません。

6.2.4 ピボット

　ピボットとは基点という意味で、画像のどこを中心点とするかを指定します。ピボットは2次元のベクトルデータで指定します。ピボットの指定の仕方はUnityのuGUIの仕様に準拠しており、下記のようになります。

|  |  |
| --- | --- |
| ピボット = {0.0f, 0.0f} | 画像の左下 |
| ピボット = {0.5f, 0.5f} | 画像の中心 |
| ピボット = {1.0f, 1.0f} | 画像の右上 |

座標(0, 0)、ピボット(0, 0)の場合



座標(0, 0)、ピボット(0.5, 0.5)の場合



座標(0, 0)、ピボット(1, 1)の場合



**6.3 章末テスト**

　① Lesson\_06のプロジェクトに下記の画像データを追加しなさい。(ハンズオン)

　　Assets/sprite/hp.png

Assets/sprite/mp.png

Assets/sprite/hpmp\_bar.png

② Lesson\_06を改造して、下記のようなHUDを表示しなさい。



チェック項目

|  |  |
| --- | --- |
| ①ができた。 | ２０点 |
| ②　HPバーの背景が表示できた。 | ２０点 |
| ② HPバーが表示できた。 | ２０点 |
| ②　MPバーの背景が表示できた。 | ２０点 |
| ② MPバーが表示できた。 | ２０点 |

**Chapter 7 3Dモデルの表示**

　このチャプターではMicrosoft社が提供しているライブラリのDirectXTKを使用して、3Dモデルを表示する方法を学びます。

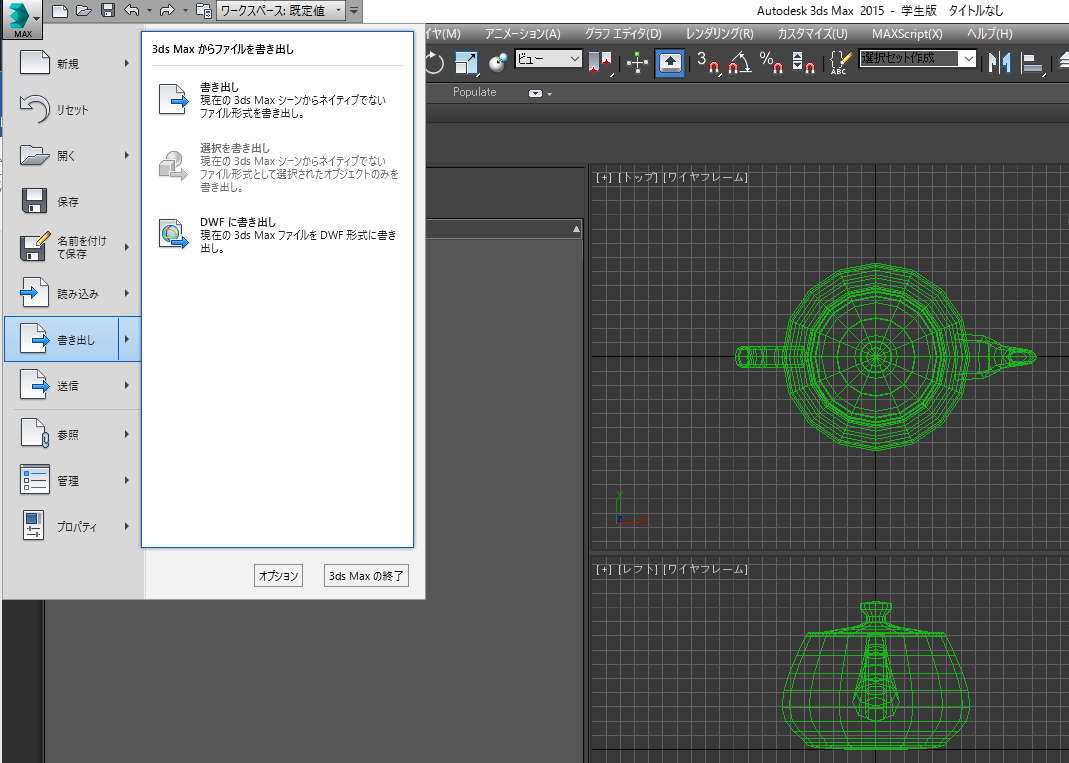
**7.1 FBXファイル**

　FBXファイルは3dsMaxやMAYAなどのDCCツールを販売しているAutodesk社が提供している3Dシーンファイルです。今回のサンプルでは、このFBXファイルを中間モデルデータとして活用しています。

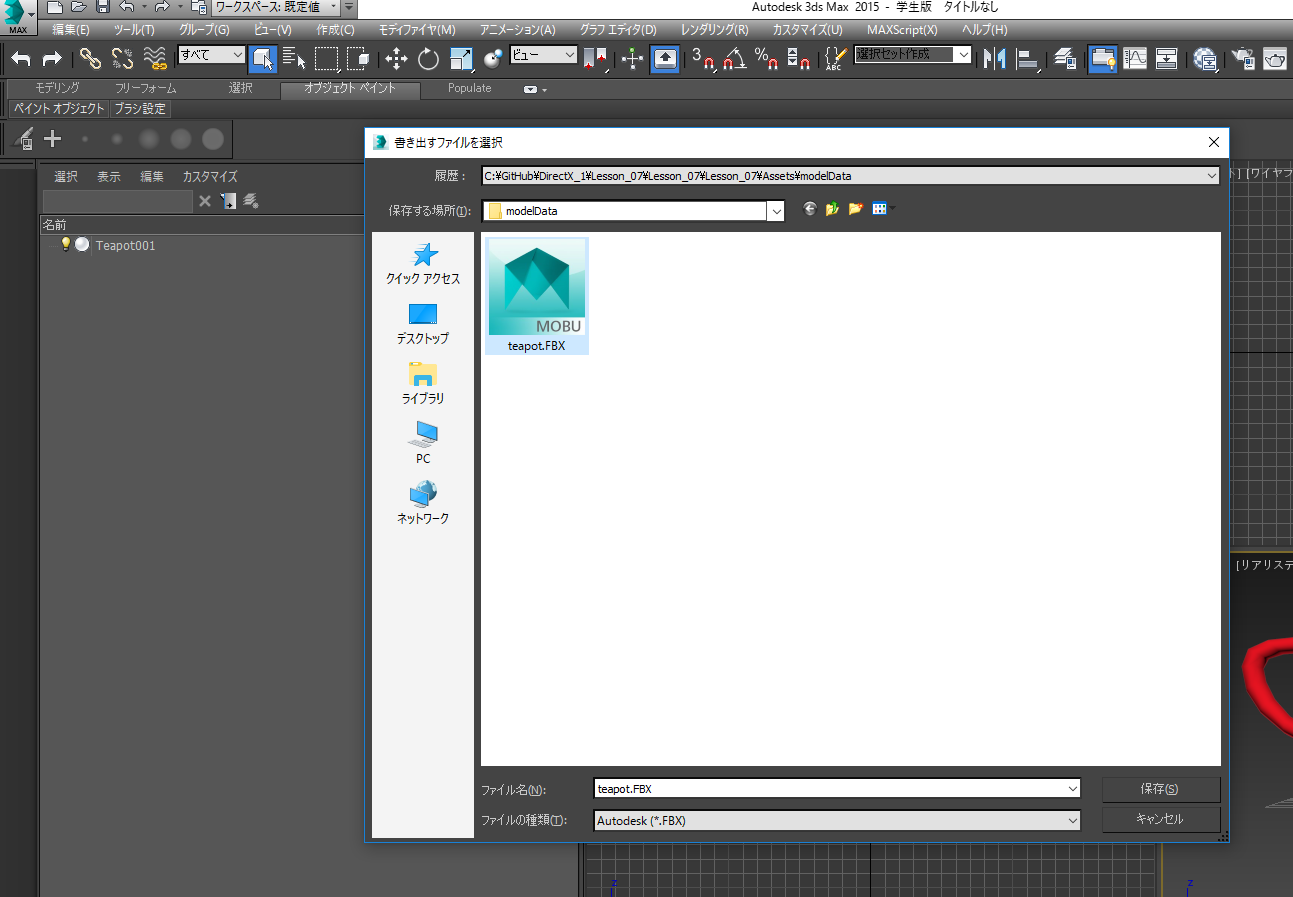
7.1.1 FBXファイルの作成の仕方。

　FBXファイルは3dsMaxを使用することで作成することができます。3dsMaxで3Dシーンを作成したら、下記のようにFBXファイルの書き出しを行ってください。

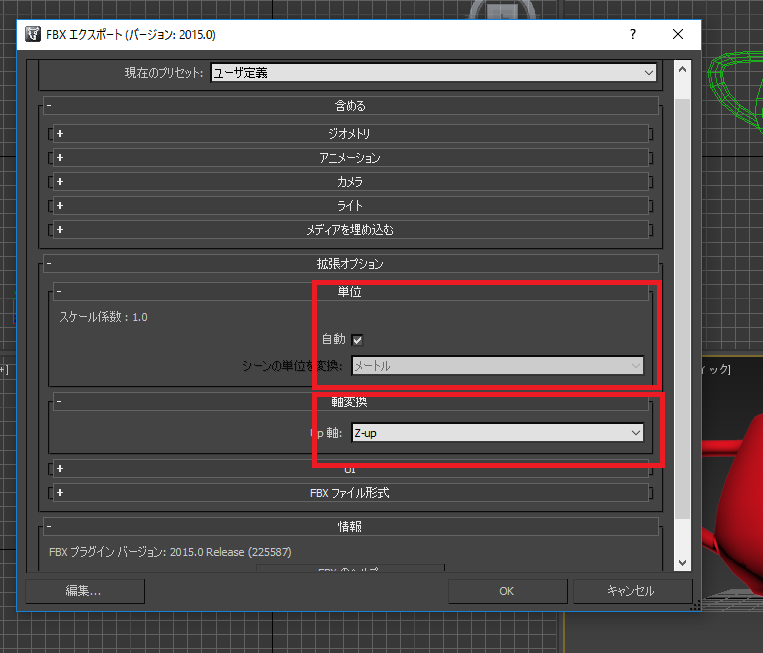
**①メニューから書き出しを選択**



**②　ファイルの種類をAutodesk(\*.FBX)になっていることを確認して、任意の場所に出力**



**③ 単位や軸変換などに気を付けながら出力**

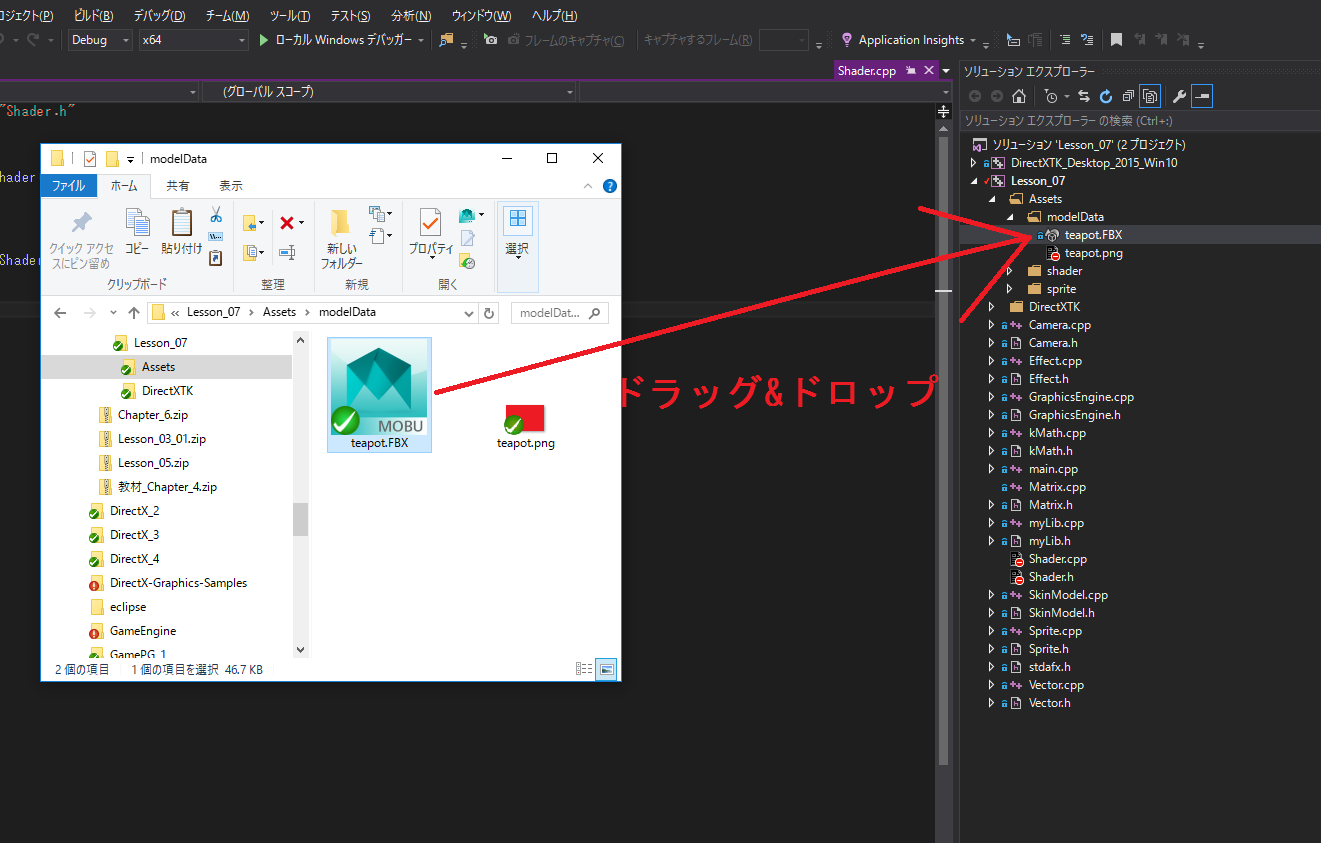
****

**7.2 CMOファイル**

FBXファイルは3Dシーンファイルなので、3Dモデルの情報だけではなく、カメラ、ライトなど様々な情報が入っています。そのため、3Dモデルを表示することだけを考えると、不要なデータが多数含まれており、データサイズが膨大になります。そこで、DirectXTKではFBXファイルをそのまま使用するのではなく、FBXファイルからCMOファイルを作成して使用します。

7.2.1 CMOファイルの作成

CMOファイルはVisualStudioの機能を使用して作成します。まず、CMOファイルに変換したいFBXファイルをVisualStudioに追加します。



これでプロジェクトをビルドするとResource/modelDataフォルダにteapot.cmoというファイルが作成されます。

FBXファイルを中間フォーマットとしてCMOファイルを作成するので、7.1で出力するFBXファイルはAssets/modelDataの下に出力するのをオススメします。

**7.3 SkinModelクラス**

　今回使用するサンプルプログラムのLesson\_07では、3Dモデルの表示を行うSkinModelクラスが用意されています。では、使い方を見ていきましょう。

main.cpp(23行目)

|  |
| --- |
| SkinModel g\_teapotModel; |

g\_teapotModelというスキンモデルクラスのインスタンスを定義しています。

main.cpp(167行目)

|  |
| --- |
| //SkinModelを初期化。  g\_teapotModel.Load(L"modelData/teapot.cmo"); |

Load関数を使用して、モデルのロードを行っています。

main.cpp(111行目)

|  |
| --- |
| //3Dモデルを描画する。  g\_teapotModel.UpateWorldMatrix(  g\_teapotPosition,  g\_teapotRotation,  g\_teapotScale  );  g\_teapotModel.Draw(g\_camera3D->GetViewMatrix(),g\_camera3D->GetProjectionMatrix()); |

UpdateWorldMatrix関数を使用して、ワールド行列を更新して、Draw関数を使用してモデルを描画しています。

**7.4 章末テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/k11ESFAwPwBy6Ye82>

QRコード



**Chapter 8 ベクトル**

　このチャプターでは3Dゲームを作る上でかかすことのできないベクトルについて見ていきます。

**8.1 概念**

　ベクトルとは大きさと方向を表す量です。例えば、3Dゲームではベクトルを使用して車の移動速度などを表すことがあります。

　自動車の速度は60km/hなどと表されていると思います。しかし、この表記だけでは自動車は北に進んでいるのか、南に進んでいるのか？はたまた東か西に進んでいるのか分かりません。ここで、ベクトルを使用することで、速度と方向を簡潔に表すことができます。

では、ベクトルで方向を表す方法を見ていきましょう。

北(0、1)

東(１、0)

西(-1、0)

南(0、－１)

北を（0、１）、南を（0、－１)、東を（１、0）、西を（－１、０）の2次元ベクトルで表しています。大きさが１のベクトルは正規化されたベクトルと呼ばれ、力を持たない方向のみを表すベクトルとして扱えます。この時に、例えば車が北に60km/hで進んでいる場合は、(0、60)と表されます。

では、車が下記の用に北東に進んでいる場合の方向はどのように表すのでしょうか。

北(0、1)

南(0、－１)

西(-1、0)

東(１、0)

V(車の進行方向を表すベクトル)

45°

Ｖは進行方向を表しているベクトルなので、大きさは１です。そこから、このベクトルのx成分とy成分は下記のように求まります。

V.x = cos(45°)≒ 0.707

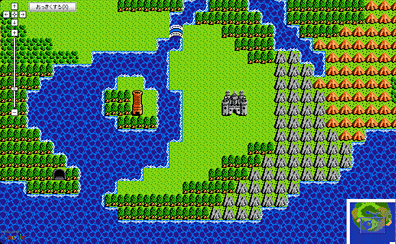
V.y = sin(45°)≒ 0.707

よって、車の進行方向は(0.707、0.707)と表されます。この時に車の速度が60km/hの場合、車の移動速度をベクトルで表すと(42.42、42.42)と表されます。

**8.2 ベクトルを座標として扱う**

　ゲームではベクトルを座標として扱うことが多々あります。世界の中心を原点(0、0)と定義して、下記のような座標系でオブジェクトの座標を扱います。

Y軸



Ｘ軸

原点(0,0)

例えば、上図の場合、お城の座標は(２０、１０)、ダンジョンの座標は(―５０、―３０)、塔の座標は(―１５、５)といったように表します。

これは３Ｄゲームの場合も同じです。



**8.3 ベクトルの加算**

　ベクトルは加算を行うことができます。例えば、ベクトルV1（２、４）にベクトルV2（５、３）を加算すると、下記のようなベクトルＶ３が求まります。

*V3=Ｖ１＋Ｖ２*

*＝（２＋５、４＋３）*

*＝（７、７）*

これを図示化すると下記のようになります。

Y軸

V3

V1

V2

Ｘ軸

ベクトルの加算は下記のように考えることもできます。

あるワールド空間上のオブジェクトの座標Ｐ０（２，４）にベクトルＶ１（５，３）を加算することで、そのオブジェクトをベクトル方向に動かすことができます。

*移動後のP０＝Ｐ０＋Ｖ１*

*＝（２＋５、４＋３）*

*＝（７、７）*

この特性から、例えば、ゲームキャラクターを移動させるときに、移動速度をベクトルで表現して、キャラクターの座標に速度を加算するといったことが行われます。

移動後のＰ０

V1

Ｐ０

Y軸

Ｘ軸

**8.4 中間テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/vs0JndWIgwNUpc8E2>

QRコード



**8.5 ベクトルの減算**

　ベクトルは減算を行うことができます。例えば、ベクトルV2（２、４）からベクトルV2（５、３）を減算すると、下記のようなベクトルＶ３が求まります。

*V3=Ｖ１―Ｖ２*

*＝（２―５、４―３）*

*＝（－３、１）*

これを図示化すると下記のようになります。

Y軸

V1

V2

V3

Ｘ軸

ベクトルの減算は下記のように考えることもできます。

あるワールド空間上のオブジェクトの座標Ｐ０（２、４）からベクトルＰ１（５、３）を減算すると、Ｐ１からＰ０に向かうベクトルＶ１を計算することができる。

V1 = P0－P1

*＝（２－５、４―３）*

*＝（－３、１）*

Y軸

Ｐ０



Ｐ１



V1

Ｘ軸

これは、P０からP1を減算して求まったV1というベクトルをP1に加算すると、P0と同じ座標になるということです。

この特性は非常に重要で、ゲームでよく使われます。では、ゲームでの実例を見ていきましょう。

**8.5.1 ２点間の距離を使ったオブジェクトのあたり判定**

最もゲームで使用されて、かつ高速なあたり判定は２点間の距離を使ったものです。例えば、マリオがコインをゲットする処理を考えてみましょう。このような処理は**「マリオとコインの距離が一定値以下、１０ｃｍ以下になったらゲットできる」**というようなプログラムで実現されています。この２点間の距離を計算すためには、マリオの座標とコインの座標を減算して、コインからマリオに向かうベクトル(マリオからコインに向かうベクトルでもＯＫ)を求めて、そのベクトルの長さを求めることで計算できます。

Y軸

減算して、このベクトルVを求める



　　　　　　　　　　　　　　　　　　　　　　　（７、５）



　　　　　　　　　　　 (３，２)

Ｘ軸

コインの座標(７、５)からマリオの座標(３，２)を減算して、上の図のベクトルV(４、3)を求めます。あとは、このベクトルの長さを求めることで、コインとマリオの距離を計算することができます。

　ベクトルの長さは３平方の定理を使用することで求めることができます。

V1





y

ｘ

上の図のように、補助線を入れてみるとベクトルV1は直角三角形の斜辺になっていることが分かります。直角三角形の斜辺の長さは三平方の定理を使えば求まります。直角三角形の斜辺の長さは、残る二つの辺の２乗の総和の平方根で求まります。

　V1の長さ=

(※３次元の場合はzも必要です！！！)

では、２点間の距離を求める疑似コードを見てみましょう。

|  |
| --- |
| CVector3 v = mario.m\_position – coin.m\_position;　//マリオの座標からコインの座標を減算する。  float len = sqrt(v.x \* v.x + v.y \* v.y + v.z \* v.z); //ベクトルVの長さを求める。  //sqrtはC言語の標準関数です。  //引数で渡された値の平方根を求めます。  if( len < 10.0f ){  //マリオとコインの距離が10cm以下になったら・・・  　　//ここにコインを取得するプログラムを記述する。  } |

**8.5.2 Hands-On演習\_1**

　サンプルプログラムのLesson\_09\_01を使って、Player.cppの31行目にコインとプレイヤーの距離を計算するプログラムを追加しましょう。

**8.5.3 CVector3::Length関数**

　これはDirectXの機能ではありませんが、多くのゲームエンジンのベクトルクラスには、ベクトルの距離を求める関数が用意されています。今回のサンプルのCVector3クラスにはベクトルの長さを求めるためのLength関数が用意されています。そのため、先ほどの疑似コードは下記のように書き換えることができます。

|  |
| --- |
| CVector3 v = mario.m\_position – coin.m\_position;　//マリオの座標からコインの座標を減算する。  float len = **v.Length();** //vの長さを求める。Length関数の中で行われていることは、sqrtを使うコードと同じ。  if( len < 10.0f ){  //マリオとコインの距離が10cm以下になったら・・・  　　//ここにコインを取得するプログラムを記述する。  } |

**8.5.4 Hands-On演習\_2**

　8.5.2で実装した距離の計算を行うプログラムを、Length関数を使用するように変更する。

**8.6 中間テスト(30分)**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/h4ZEttj7AlxpCUMw1>

QRコード



**8.7.1 オブジェクトの追従**

　では、続いてピクミンのようにプレイヤーに追従してくるキャラクターをプログラミングしてみましょう。ワールド空間上に点Ｐ０、Ｐ１があった場合に下記のような計算を行うと、Ｐ１からP０に向かって伸びるベクトルVが計算できました。

***V = P０―P１***

この計算を下記のように置き換えてみましょう。

***V=プレイヤーの座標―ピクミンの座標***

すると、ベクトルVはピクミンからプレイヤーに向かって伸びるベクトルになります。ベクトルVは方向とプレイヤーまでの長さが入っているベクトルになっているので、このベクトルを正規化して方向のみのベクトルにします。正規化はベクトルの各要素(x,y,z)をベクトルの長さで除算することで行えます。

***Vnorm = { V.x / |V|、V.y /|V|、V.z/|V| }***

ではベクトルの正規化を行う疑似コードを見てみましょう。

|  |
| --- |
| CVector3 v = player.m\_position – pikumin.m\_position;　//プレイヤーの座標からピクミンの座標を減算する。  float len = v.Length();  //ベクトルの各要素を長さで除算して、正規化されたベクトルを求める。  v.x = v.x / len;  v.y = v.y / len;  v.z = v.z / len; |

これでvはピクミンからプレイヤーへの方向を表すベクトルになりました。

このベクトルを使用して、下記のようにピクミンを動かすとプレイヤーに追従していきます。

|  |
| --- |
| pikumin.m\_position += v＊5.0f； //プレイヤ―に向かってピクミンが速度5.0で動いていく。 |

**8.7.2 Hands-On 演習\_1**

サンプルプログラムのLesson\_09\_02/Hands\_Onを使って、コインがプレイヤーに追従するように、Coin.cppの21行目からプログラムを追加しなさい。

**8.7.3 CVector3::Normalize関数**

　これもDirectXの機能ではありませんが、多くのゲームエンジンのベクトルクラスには、ベクトルを正規化する関数が用意されています。今回のサンプルのCVector3クラスにはベクトルを正規化するためのNormalize関数が用意されています。そのため、先ほどの疑似コードは下記のように書き換えることができます。

|  |
| --- |
| CVector3 v = player.m\_position – pikumin.m\_position;　//プレイヤーの座標からピクミンの座標を減算する。  **v.Normalize();** //ベクトルを正規化する。  pikumin.m\_position += v＊5.0f； //プレイヤ―に向かってピクミンが速度5.0で動いていく。 |

**8.7.4 Hands-On演習\_2**

　8.7.3で実装した追従を行うプログラムを、Normalize関数を使用するように変更する。

**8.8 中間テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/VF5wPMzXJoMJl3gE2>

QRコード



**Chapter 9 オイラー角とクォータニオン**

　このチャプターでは３Dオブジェクトの回転を表現するのオイラー角とクォータニオンについて見ていきます。

**9.1　オイラー角とは？**

　オイラー角とは、３Dオブジェクトの回転をロール・ピッチ・ヨーで表現したものとなります。

ロールはX軸周りの回転角度、ピッチはZ軸周りの回転角度、ヨーはY軸周りの回転角度を表します。例えば、ロール45°、ピッチ30°、ヨー40°と表現される場合、３Dオブジェクトは、X軸周りに45°、Z軸周りに30°、Y軸周りに20°回転していることになります。

**9.2　オイラー角の表現にベクトルを利用する。**

　オイラー角は3軸の回転角度を使って表現されるため、ゲームプログラミングでは、しばしばベクトルを使って表現されます。下記に疑似コードを記載します。

|  |
| --- |
| CVector3 rotation;  rotation.x = 30.0f; //ロール30°  rotation.y = 20.0f; //ヨー40°  rotation.z = 15.0f; //ピッチ15° |

**9.3 Hands-On 演習\_1**

Lesson\_11\_1を使用して、キャラクターをキーボードの入力に応じて回転するようにしなさい。

**9.4 オイラー角のデメリット**

　シンプルかつ直観的に回転を表現できるオイラー角ですが、下記のようなデメリットがあります。

　①　所定の方向の表現が一意に決まらない。

　　　　例えば、オイラー角では、（ロール135°）と（ロール45°、ピッチ180°、ヨー

180°）が同じ方向を表します。

　②　２つの角の補間が難しい。

これは①に起因するのですが、方向の表現が一意に決まらないため、２つの回転の補間が困難になります。

　③　ジンバルロックが起きる。

第二の回転軸で+-90°回転するとジンバルロックという、回転軸が失われる現象が起きてしまい、回転の自由度が失われてしまいます。オイラー角を利用している限り、この問題の簡単な解決策はありません。

では、次の節ではこれらの問題を解決することができる回転の表現である、クォータニオンについて見ていきましょう。

**9.5　クォータニオン（四元数）とは？**

　クォータニオンとは、xyzwの４要素で任意の軸周りの回転を表現することができる数字です。オイラー角と同様に、３Dオブジェクトの回転を表現することができ、9.4で紹介したオイラー角のデメリットをすべて解消できるものとなります。しかし、オイラー角と比べると直感的でなく、分かりにくいものとなっています。

**9.6 クォータニオンの定義**

　回転軸を(ｘ、ｙ、ｚ)、回転角度をθとしたときのクォータニオンは下記のように定義されます。

**（ｘ×sin(0.5×θ)、y×sin(0.5×θ）、z×sin(0.5×θ)、cos(0.5×θ))**

ここでは、この定義の数学的な意味は説明しません。このように定義されていると考えてください。

**9.7 クォータニオンの乗算**

　クォータニオンは乗算を行うことができます。乗算を行うことで、回転を加算することができます。例えば、Y軸周りに5°回した後で、さらにY軸周りに5°回すといったことが行えます。

下記はクォータニオンの乗算を行う疑似コードです。

|  |
| --- |
| CVector3 rotAxis = {0.0f, 1.0f, 0.0f}; //回転軸  CQuaternion qRot\_00, qRot\_01, qRot;  qRot\_00.SetRotationDeg(rotAxis, 5.0f);  qRot\_01.SetRotationDeg(rotAxis, 5.0f);  qRot.Multiply(qRot\_00, qRot\_01); //これでqRotはY軸周りに10°回転するクォータニンになる。 |

また、クォータニオンの乗算は交換法則が成り立たないことに注意してください。

上のコードですと、qRot\_00×qRot\_01とqRot\_01×qRot00は同じにならないことがあります。

**9.8 CQuaternionクラス**

　これはDirectXの機能ではありませんが、多くのゲームエンジンにはクォータニオンのクラスが用意されています。この後使用するサンプルプログラムにも、CQuaternionクラスがあります。(9.7の疑似コードでも使用しています)。このクラスにはクォータニオンを作成するためのSetRotationDeg関数、クォータニオンの乗算やベクトルの回転を行うMultiply命令などがあります。

**9.9 Hands-On 演習\_2**

Lesson11\_03を使用して、クォータニオンを使ってキャラクターを回転させるようにしましょう。

**9.10　球面線形補完**

　クォータニオンを利用する大きな理由として、球面線形補完を行えるというものがあります。ざっくりと説明すると、回転の補間を滑らかに行うことができるというものです。良く使われるのはアニメーションの補間などです。

　CQuaternionはSlerpという関数で球面線形補完の機能を提供しています。下記に疑似コードを記載します。

|  |
| --- |
| CQuaternion qStart;  CQuaternion qEnd;  qStart.SetRotationDeg(CVector3::AxisY(), 45.0f);  qEnd.SetRotationDeg(CVector3::AxisY(), -45.0f);  CQuaternion qRot;  qRot.Slerp( 0.2f, qStart, qEnd); //qStartとqEndを0.2で補完する。 |

**9.11　中間テスト**

下記のURLのテストを行なさい。

URL

<https://goo.gl/forms/24uOQoSq1G3jqnA03>

QRコード



**Chapter 10 行列**

　このチャプターは行列がゲームでどのように使われているのか？という点と、ゲームで使われる際に重要になってくる点について見ていきます。

**10.1 行列とベクトル**

　行列とベクトルは乗算することができて、その結果はベクトルになります。ベクトルに行列をかけることによって、ベクトルを変換することができるのです。例えば、下記のような行列があります。

　①　平行移動行列

　　　　→ベクトルを平行移動させます。

　②　回転行列

　　　　→ベクトルを回転させます。

　③　拡大行列

　　　　→ベクトルを拡大します。

　④　ワールド行列

→多くの場合で、①、②、③の行列を混ぜ合わせたもの。ローカル座標系のベクトルをワールド座標系に変換します。

　⑤　カメラ行列

　　　　→ワールド座標系のベクトルをカメラ座標系に変換します。

　⑥　透視投影行列

　　　　→カメラ座標のベクトルをスクリーン座標系に変換します。

**10.2　ワールド行列**

　ワールド行列はローカル座標系から、ワールド座標系に変換するための行列です。要は３Dモデルを移動、回転、拡大をさせるための行列です。Chapter８、９でベクトル、オイラー角、クォータニオンなどを使用して、3Dモデルを移動、回転させる方法を学びました。

それらのチャプターで必ず、下記のような関数を呼び出していたはずです。

***mode.UpdateWorldMatrix( m\_position, m\_rotation, m\_scale );***

3Dモデルを表示する時に、最終的に３Dモデルの位置、回転、拡大を決めるためにはワールド行列というものを作成する必要があります。実は、この関数の中で、渡された座標、回転、拡大率を使ってワールド行列を作成していたのです。

**10.2.1 なぜワールド行列を使うの？**

　3Dモデルをローカル座標系から、ワールド座標系に変換するということは、3Dモデルの頂点座標を移動、回転させるということです。では、なぜ座標変換に行列を使うのでしょうか？座標を動かすだけなら、ベクトルの足し算で出来ますし、回転もクォータニオンの乗算だけで可能です。では、なぜわざわざワールド行列などを用意するのでしょうか？理由はいくつかありますが、ざっくりと説明すると、計算がシンプルで早いからです。行列を使うと、平行移動→回転→拡大といった変換を一回の乗算で行うことができます。そのため、モデルの座標変換には行列が使われています。

**10.2.2 平行移動行列**

平行移動行列は下記のようになります。

ｘ軸上を１０、y軸上を２０、ｚ軸上を１５移動する行列＝

**10.2.3 回転行列**

　回転移動行列は下記のようになります。

x軸周りにθ回転 =

y軸周りにθ回転 =

z軸周りにθ回転 =

**10.2.4 拡大行列**

　拡大行列は下記のようになります。

X軸に2.0倍、Y軸に0.8倍、Z軸に4倍 =

**10.2.5 Hands-On**

Lesson\_12\_01を使って下記のHands-Onを行いましょう。

　①　平行移動行列を作って、ティーポットを動かす。

　②　回転行列を作って、ティーポットを回転させる。

　③　拡大行列を作って、ティーポットを拡大させる。

**10.2.6 行列と行列の乗算**

　行列は乗算することができます。行列同士を乗算した結果は行列になります。例えば下記のような乗算を行うことができます。

***行列A = 回転行列×平行移動行列***

この計算によって、出来上がった行列Aとベクトルの乗算を行うと、そのベクトルは回転してから平行移動します。つまり、行列Aは回転行列と平行移動行列を混ぜ合わせた行列になります。また、行列の乗算は交換法則が成り立ちません。

***回転行列×平行移動行列　≠　平行移動行列×回転行列***

これは非常に重要な性質ですので、覚えておいてください。

**10.2.7 Hands-On**

Lesson\_12\_01を使って下記のHands-Onを行いましょう。

　①　平行移動行列と回転行列を乗算して、ティーポットを動かす。

　②　平行移動と回転の乗算順番を変更すると、どうなるか確認する。

**10.2.8 ワールド行列の作成**

　ワールド行列はここまで見てきた、平行移動行列、回転行列、拡大行列を乗算することで作成できます。ワールド行列は下記の順番で乗算します。

***ワールド行列　＝　拡大行列×回転行列×平行移動行列***

基本的にこの順番は間違えてはいけません。拡大→回転→平行移動の順番です。

**10.3 中間テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/e3nZrXnDkCknPGFh1>

QRコード



**Chapter 11 カメラ行列と透視変換行列**

このチャプターでは3Dシーンを映すためのカメラの概念と、ワールド空間上の点を、カメラ空間に変換するカメラ行列と、スクリーンに絵を出すために、３Ｄシーンをスクリーン空間に変換する透視変換行列について見ていきます。

**11.1 カメラ行列**

**11.1.1 注視点と視点**

カメラにはカメラが移しているターゲットを表す注視点と、カメラの位置を表す視点があります。(前期ゲームプログラミングのLesson\_05を参照)。視点と注視点を動かすことでカメラを動かすことができます。

**11.1.2 カメラの上方向**

注視点と視点を決定しただけでは、まだカメラの姿勢を決定することはできません。カメラの姿勢を決定するためには、カメラの上方向を指定する必要があります。多くのゲームでは、上方向は(0、1、０)を指定してやればよいのですが、フライトシミュレーションのような、360度旋回するゲームを作成している場合は注意が必要です。なぜなら、カメラの上方向が(1、0、0)にもなりえるからです。

**11.1.3 カメラ行列**

　注視点、視点、カメラの上方向を決定すると、カメラ行列を作成することができます。

*Lesson13\_00/Camera.cpp(46行目)*

|  |
| --- |
| //ビュー行列の算出  m\_viewMatrix.MakeLookAt( m\_position, m\_target, m\_up ); |

このコードは毎フレーム呼ばれている、CameraクラスのUpdate関数の一部です。MakeLookAtはCMatrixクラスのメンバ関数で、引数の値を元にカメラ行列を計算してくれます。CameraクラスではUpdate関数が呼ばれるたびに、設定されている情報を元にカメラ行列を計算します。

**11.1.4 CMatrix::MakeLookAt関数**

　この関数の中でDirectXの機能を使って、カメラ行列を計算しています。

*Lesson13\_00/Matrix.h(195行目)*

|  |
| --- |
| /\*!  \* @brief 注視点、上方向、カメラ位置からカメラ行列を作成。  \*@param[in] position カメラ位置。  \*@param[in] target 注視点。  \*@param[in] up カメラの上方向。  \*/  void MakeLookAt( const CVector3& position, const CVector3& target, const CVector3& up )  {  DirectX::XMStoreFloat4x4(  &mat,  **DirectX::XMMatrixLookAtLH(position, target, up)**  );  } |

網掛けになっている箇所が、カメラ行列を計算してくれている関数です。

**11.1.5 Hands-On 視点と注視点を変えてみる**

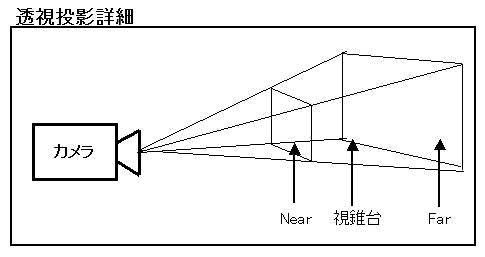
Lesson\_13を改造して、視点と注視点を変更してみましょう。

**11.2 透視変換行列**

では、続いてカメラ空間のオブジェクトをスクリーン空間に変換する透視変換行列について見ていきます。

**11.2.1 近平面と遠平面**

　これは、カメラ空間のどこから、どこまでをスクリーン空間に変換するかを決めるためのパラメータです。



近平面(Near)から遠平面(Far)の間にある、オブジェクトがスクリーンに表示されます。

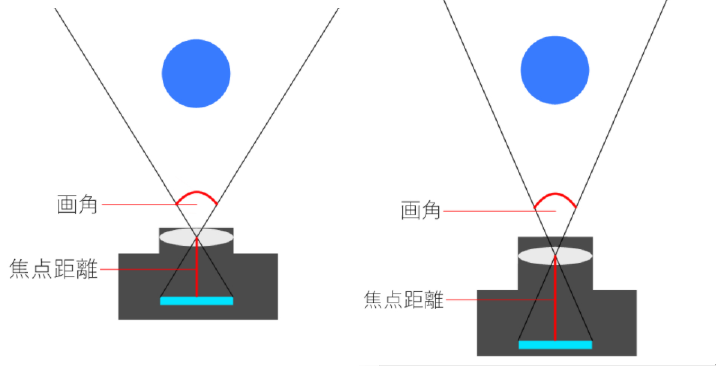
*Lesson13\_00/Camera.h(195行目)*

|  |
| --- |
| /\*!  \* @brief 遠平面を設定。  \*/  void SetFar( float fFar )  {  m\_far = fFar;  }  /\*!  \* @brief 近平面を設定。  \*/  void SetNear( float fNear )  {  m\_near = fNear;  } |

カメラクラスにはSetFarとSetNearがあり、近平面と遠平面を設定することができます。

**11.2.2　画角**

　画角も近平面と遠平面と同様に、下記の図のようにカメラが映す範囲を指定するものです。



近平面、遠平面、画角が決まると、スクリーンに投影する範囲が確定します。

*Lesson13\_00/Camera.h(161行目)*

|  |
| --- |
| /\*!  \* @brief 画角を設定。  \* @details  \* m\_updateProjMatrixFuncがenUpdateProjMatrixFunc\_Perspectiveの時に使用される。  \*/  void SetViewAngle(float viewAngle)  {  m\_viewAngle = viewAngle;  } |

カメラクラスにはSetViewAngle関数があり、画角を設定することができます。

**11.2.3 アスペクト比**

　最後にアスペクト比を設定します。アスペクト比とはスクリーンの縦横比のことです。例えば、解像度が1280×720のスクリーンの場合、アスペクト比は1.77≒1280/720です。

**11.2.4 透視変換行列**

近平面、遠平面、画角、アスペクト比を決定すると透視変換行列を作成することができます。

*Lesson13\_00/Camera.cpp(32行目)*

|  |
| --- |
| //透視変換行列を計算。  float aspect = FRAME\_BUFFER\_W / FRAME\_BUFFER\_H;  m\_projectionMatrix.MakeProjectionMatrix(  m\_viewAngle,  aspect,  m\_near,  m\_far  ); |

このコードは毎フレーム呼ばれている、CameraクラスのUpdate関数の一部です。MakeProjectionMatrixはCMatrixクラスのメンバ関数で、引数の値を元に透視変換行列を計算してくれます。CameraクラスではUpdate関数が呼ばれるたびに、設定されている情報を元に透視変換行列を計算します。

**11.2.5 CMatrix::MakeProjectionMatrix関数**

この関数の中でDirectXの機能を使って、透視変換行列を計算しています。

*Lesson13\_00/Matrix.h(162行目)*

|  |
| --- |
| /\*!  \* @brief プロジェクション行列を作成。  \*@param[in] viewAngle 画角。(ラジアン)。  \*@param[in] aspect アスペクト比。  \*@param[in] fNear 近平面。  \*@param[in] fFar 遠平面。  \*/  void MakeProjectionMatrix(  float viewAngle,  float aspect,  float fNear,  float fFar  )  {  DirectX::XMStoreFloat4x4(  &mat,  **DirectX::XMMatrixPerspectiveFovLH(viewAngle, aspect, fNear, fFar)**  );  } |

網掛けになっている箇所が、透視変換行列を計算してくれている関数です。

**11.2.6 Hands-On 近平面、遠平面、画角を変更してみる。**

　Lesson\_13を改造して、近平面、遠平面、画角を変更してみましょう。

**11.3　3Dモデル描画**

カメラ行列、透視変換行列は３Ｄモデルを描画する時に使用されます。今回のサンプルではティーポットのモデルを描画する時に、カメラ行列と透視変換行列を渡しています。

*Lesson13\_00/main.cpp(108行目)*

|  |
| --- |
| g\_teapotModel.Draw(g\_camera3D->GetViewMatrix(), g\_camera3D->GetProjectionMatrix()); |

3Dモデルがスクリーンに表示されるには、ワールド座標系→カメラ座標系→スクリーン座標系と変換する必要があります。実は、今までモデルを描画する時に渡していた引数がカメラに関係している引数だったのです。

**11.4　中間テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/1ngIkMAbSxh2wMI62>

QRコード



**Chapter 12 三人称カメラ**

このチャプターでは見下ろし方の三人称カメラの実装方法について見ていきます。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

**12.1　プレイヤーを追従**

　プレイヤーを追従させるためには、カメラの注視点と視点を動かす必要があります。多くの三人称カメラのゲームではプレイヤーが操作するキャラクターの座標を基準にして、注視点を決めます。そして、視点は注視点からの距離を保つように決定することによって、プレイヤーを追従するカメラが出来上がります。

下記に疑似コードを記載します。

|  |
| --- |
| //カメラの注視点から視点に向かうベクトルを計算する。  CVector3 toCameraPos = g\_camera->m\_position – g\_camera->m\_target;  //新しい注視点を計算する。注視点はプレイヤーのちょっと上を見る。  g\_camera->m\_target = g\_player->m\_position;  g\_camera->m\_target.y += 2.0f;  //新しい注視点から距離を保つように、新しい視点を計算する。  g\_camera->m\_position = g\_camera->m\_target + toCameraPos; |

**12.2 Hands-On プレイヤーを追従するカメラ**

　Lesson14\_01を使用して、プレイヤーを追従するカメラを実装しましょう。

**12.3 カメラの回転**

続いて三人称カメラの回転のさせ方を学んでみましょう。

**12.3.1 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、  CVector3 toPos = cameraPos – camearTarget; |

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。  CMatrix mRot;  mRot.MakeRotationY( CMath:: DegToRad(10.0f)); //第二引数はラジアン単位。  //ベクトルと行列を乗算して回転させる。  mRot.Mul(toPos); |

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; |

**12.3.2 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2の場合はこの軸で回す

視点１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、  CVector3 toPos = pos – target;  CVector3 vUP( 0.0f, 1.0f, 0.0f );  //外積を計算して、回転軸を求める。  CVector3 vRotAxis;  //外積を求める。  vRotAxis.Cross(toPos, vUP);  //大きさ1にするために正規化を行う。  vRotAxis.Normalize(); |

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。

|  |
| --- |
| CMatrix mRot;  //任意の軸周りの回転行列を作成。  mRot .MakeRotationAxis( vRotAxis, CMath::DegToRad(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。  mRot.Multiply( toPos )  //最後にtoPosとtargetから視点を計算する。  pos = target + toPos; |

**12.4 Hands-On**

Lesson14\_01を改造して、カメラを回転させられるようにしてみましょう。

**12.5　中間テスト**

下記のURLのテストを行いなさい。

URL

<https://goo.gl/forms/U8aI0t9lWXpyOUsL2>

QRコード



**12.6 カメラを考慮したキャラクターの移動**

　カメラを回転させることができるようになると、次は「カメラを考慮したキャラクターの移動」を行いたくなります。これまでは、前後に進みたい場合はキャラクターのZ座標に＋－、左右に進みたい場合はX座標に＋－していましたが、カメラが動くとそうはいきません。なぜなら、キャラクターの前後への移動がZ座標とは限らないからです。では、カメラを考慮してキャラクターを移動させる方法を見ていきましょう。

**12.7 カメラの方向に移動させる**

カメラを考慮して、前後に動かすには、カメラの方向を使って、移動させればよのです。

カメラの方向は下記の計算で求まります。

*V = 注視点―視点 ・・・・視点から注視点に向かって伸びるベクトルを求める。*

*V /= |V| ・・・・Vを正規化*

プログラムにすると下記のようになります。

|  |
| --- |
| CVector3 v = camera.m\_target – camera.position; //視点から注視点に向かって伸びるベクトルを求める。  v.Normalize();　//vを正規化して、大きさ１のベクトルにする。 |

これでカメラの方向が求まりました。キャラクターをカメラの奥方向に動かしたい場合は、このベクトルを使って、下記のようなコードを書けばいいことになります。

|  |
| --- |
| player.m\_position += v \* 5.0f; //カメラの方向に速度5.0で移動する。 |

**12.8 Hands-On**

Lesson14\_01を使って、キャラクターをカメラの方向に移動させてみましょう。

**12.9 XZ平面での移動**

　さて、うまくいったでしょうか？　12.8のハンズオンだけでは、カメラの向きによってキャラクターが宙に浮かんでしまったり、地面にめり込んでしまったりしてしまったはずです。例えばユーザーがコントローラーの上方向を入力して、カメラを考慮した移動を行いたい場合、多くの場合でY軸方向の移動は無視され、XZ平面での移動を期待しています。

そこで、先ほどのカメラの向きを求めるコードを下記のように改造してみましょう。

|  |
| --- |
| CVector3 v = camera.m\_target – camera.position; //視点から注視点に向かって伸びるベクトルを求める。  v.y = 0.0f; //y成分を打ち消して、XZ平面でのベクトルにする。  v.Normalize();　//vを正規化して、大きさ１のベクトルにする。 |

Y成分を0にすることで、vはXZ平面上のベクトルになりました。あとは、このベクトルを使ってキャラクターを移動させてやればいいのです。

**12.9 Hands-On**

Lesson14\_01を使って、キャラクターをカメラ方向に移動させましょう。(XZ平面での移動)

**12.10 左右の移動**

　最後にカメラを考慮した左右の移動を行う方法を見ていきましょう。前後に移動する場合は、カメラの進行方向(前ベクトル)を求めました。つまり左右の移動は、カメラの横ベクトルを求めればよいのです。このベクトルを求めるには外積の「二つのベクトルの外積は、その二つのベクトルに直行するベクトルである」という性質を利用します。

　では、横ベクトルを求めるコードを見てみましょう。

|  |
| --- |
| //まずカメラの前方向を求める。  CVector3 cameraForward = camera.m\_target – camera.m\_position;  cameraForward.y = 0.0f;  cameraForward.Normalize();  //続いてカメラの横ベクトルを求める。  CVector3 cameraRight;  cameraRight.Cross(cameraForward, {0.0f, 1.0f, 0.0f}); //カメラの前ベクトルとと上ベクトルとで外積を求める。  cameraRight.Normalize(); //正規化する。 |

これで、カメラの横ベクトルが求まりました。あとは、このベクトルを使って、キャラクターを動かしてやればよいのです。

|  |
| --- |
| player.m\_position += cameraRight \* 5.0f; //カメラの横方向に速度5.0で移動する。 |

**12.11　実習課題**

　Lesson14\_01を改造して、カメラを考慮した横移動を行えるしなさい。