**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 実習課題**

　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 実習課題**

　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 実習課題**

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 実習課題**

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 |