Chapter 1 ウィンドウの作成～メッセージループ

　このチャプターではDirectX12を扱う前に、WindowsSDKを利用して、ゲームの絵を表示するためのウィンドウを表示、制御する方法について見ていきます。

1.1 WindowsSDKとDirectX12

　ウィンドウズ上で動作するゲームを作成するためには、まずウィンドウを作成する必要があります。このウィンドウを作成するAPIを提供しているのはDirectXではなく、WindowsSDKというソフトウェア開発キットになります。WindowsSDKは様々なAPIを含んでおり、その中にCreateWindow()関数、RgisterClassEX()関数、ShowWindos()関数など、ウィンドウアプリを作成するための様々なAPIがあります。実はDirectXもWindowsSDKの中に含まれているAPI群なのです。WindowsSDKを利用することによって、WindowsOS上で動作するグラフィカルなツールやゲームといったアプリを作ることができます。
　DirectX、正確にはDirectXの中のDirect3Dは3Dグラフィックスを扱うためのAPI群となっています。ゲームではDirect3Dを使って、リアルタイム3DCGの絵を作成して、その絵を作成したウインドウに表示しているわけです。

1.2 ウィンドウの作成

　では、まずはウィンドウの作成に関する処理を見ていきましょう。ウィンドウを作成するステップは下記のようになります。

1. ウィンドウクラスの登録
2. ウィンドウの作成
3. ウィンドウの表示

　では、各ステップ詳細に見ていきましょう。

1.2.1 ウィンドウクラスの登録

　ウィンドウクラスとは作成するウィンドウの基本データとなります。ウィンドウの作成は後述するCreateWindow()関数を利用するのですが、この時にどのウィンドウクラス情報を使うのか？といった指定をすることになります。
　ウィンドウの基本データの設定はWNDCLASSEXという構造体を利用します。基本データを設定することができたら、RegisterClassEx()関数を利用して、ウィンドウクラスを登録します。ウィンドウクラスのデータは使用するアイコン、マウスカーソル、ウィンドウの背景色などのデータを設定します。次のコードはウィンドウクラスの登録の疑似コードです。

```
// 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。
    MsgProc,                     // 【重要】メッセージプロシージャ(後述)
    0,                           // ウインドウクラス構造体の後ろに割り当てる補足バイト数。多くの場
合で0でよい。
    0,                           // ウインドウインスタンスの後ろに割り当てる補足バイト数。多くの場
```

```
合で0でよい。
    GetModuleHandle(nullptr),    // 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。
    nullptr,                     // アイコンのハンドル。NULLの場合はデフォルト。
    nullptr,                     // マウスカーソルのハンドル。NULLの場合はデフォルト。
    nullptr,                     // ウィンドウの背景色。NULLの場合はデフォルト。
    nullptr,                     // メニュー名。NULLだとメニューなし。
    L"MyGame",                   // ウィンドウクラスに付ける名前。
    nullptr                      // 16×16の小さいサイズのアイコン。
};

// ウィンドウクラスの登録。
RegisterClassEx(&wc);
```

さて、このデータの中で特に注目してほしいのが、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後ほど詳しく説明しますが、ゲーム以外のWindowsアプリを作る場合、特にGUIプログラミングを行う場合は心臓部分になるものとなります。

1.2.2 ウィンドウの作成

ウィンドウクラスの登録ができたら、CreateWindow()関数を利用して、ウィンドウを作成します。ウィンドウを作成する際に、どの名前を使って、どのウィンドウクラスを利用するのかを指定します。イメージとしてはウィンドウクラスがtkmファイル、ウィンドウがModelクラスのインスタンスといった感じです。次のコードはウィンドウを作成する疑似コードです。

```
// ウィンドウの作成
HWND hwnd = CreateWindow(
    L"MyGame",               // 使用するウィンドウクラスの名前。
    L"MyGame_00",            // ウィンドウの名前。
    WS_OVERLAPPEDWINDOW,     // ウィンドウスタイル。
    0,                       // ウィンドウの初期X座標。
    0,                       // ウィンドウの初期Y座標。
    1280,                    // ウィンドウの幅。
    720,                     // ウィンドウの高さ。
    nullptr,                 // 親ウィンドウのハンドル。
    nullptr,                 // メニューハンドル。
    hInstance,               // アプリケーションのインスタンス。
    nullptr,                 // WM_CREATEメッセージのlParamパラメータとして渡される引数。
);
```

CreateWindow()関数はウィンドウを識別するためのウィンドウハンドルを返してきます。ウィンドウハンドルというのは、ウィンドウを識別するためのハンドルです。マルチウィンドウのアプリケーションの場合、このウィンドウハンドルを利用してウィンドウを識別します。

1.2.2.1 ウィンドウハンドルとは？

CreateWindow()関数を利用するとHWND型のウインドウハンドルが返ってくることを学びました。ウィンドウハンドルというのはウィンドウを識別するために使用されると記述しましたが、ではどのように識別す

るのでしょうか？答えはアドレスで識別します。実はHWND型は次のコードのように、ただのHWND__という構造体のポインタ型の別名定義です。

```
// HWND__という構造体を定義する。
struct HWND__ {
    int unused;
};

// HWND__*を別名定義。
typedef struct HWND__* HWND;
```

私は昔C言語を学び始めて数か月ほどでウィンドウハンドルに出会い混乱した覚えがありますが、ようはCreateWindow()関数は作成したウィンドウを表すオブジェクトのアドレスを返してきているだけです。そのアドレスを利用して、ウィンドウを識別するわけです。

1.2.3 ウィンドウの表示

ウィンドウの表示はShowWindw()関数を利用します。ShowWindow()関数に表示したいウィンドウハンドルを渡すことでウィンドウを表示することができます。次のコードはウィンドウを表示する疑似コードです。

```
// 作成したウィンドウを表示状態にする。
ShowWindow( hwnd, nCmdShow );
```

1.3 メッセージループ

1.2節でウィンドウを作成して表示する処理について勉強しました。この処理はいわゆる初期化処理と呼ばれるものです。ウィンドウを作成することができたら、ユーザーのアクションに対して様々な反応を返す必要があります。例えば「×ボタンを押されたらウィンドウを終了させる」や、「マウスの右クリックでポップアップメニューを出す」などです。このユーザーの反応に対するアクションはメッセージループと呼ばれるループ内に記述されます。イメージとしてはゲームループに近いです。実際ゲームでは、このメッセージループの一部がゲームループとなります。メッセージループの中でユーザーからのメッセージが処理されて、後述するメッセージプロシージャという関数が呼び出されることになります。次のコードはメッセージループの疑似コードです。

```
// step-2 メッセージループを実装する。
MSG msg = { 0 };
// 終了メッセージが送られてくるまでループを回す。
while (WM_QUIT != msg.message) {
    // PeekMessage()関数を利用して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))
    {
        // 仮想キーメッセージを文字メッセージに変換する。
        // ユーザーからの文字入力を取得する必要がある場合は、本関数を呼び出す。
        TranslateMessage(&msg);
```

```
            // メッセージを処理する。
            // この関数の中でウィドウクラスで指定したメッセージプロシージャー関数が呼ばれる。
            DispatchMessage(&msg);
        }
    }
```

このメッセージループではPeekMessage()関数、TranslateMessage()関数、DispatchMessage()関数の３つの関数が利用されています。PeekMessage()関数はユーザーが行った操作をメッセージという形で取り出します。例えば、マウスを動かした、右クリックした、左クリックしたといった情報です。取得したメッセージは第一引数のMSG構造体の変数に格納されます。TranslateMessage()関数はユーザーからの文字入力を処理する必要がある場合に呼び出します。仮想キーメッセージとなっているデータを扱いやすい文字データに変換してくれます。最後にDispatchMessage()関数ですが、この関数の中でウィンドウクラスの作成時に指定したメッセージプロシージャ関数が呼ばれます。

1.4 メッセージプロシージャ

では、最後にメッセージプロシージャについて見ていきます。メッセージプロシージャとはユーザーからの入力を処理する関数です。ユーザーの操作に対しての反応はアプリによって違います。例えば、マウスの右クリックの場合、ポップアップメニューを表示するアプリもあれば、表示しないアプリもあります。このように、ユーザーの操作に対してのアクションはアプリによって異なるため、そのアクションをプログラミングする必要があります。このプログラムの窓口となるのがメッセージプロシージャです。下記のコードはマウスの左クリックのメッセージが来た時に、メッセージボックスを表示するメッセージプロシージャの疑似コードです。

```
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_LBUTTONDOWN:
        // マウスの左ボタンが押された。
        MessageBox(hWnd, L"マウスの左ボタンが押された!", L"通知", MB_OK);
        break;
    default:
        // それ以外はデフォルトの処理に飛ばす。
        return DefWindowProc(hWnd, msg, wParam, lParam);
    }

    return 0;
}
```

1.5 【ハンズオン】ウィンドウを表示してみよう

ではSample_01_01を使って、ウィンドウを表示するプログラムを実装していきましょう。
Sample_01_01/Game/Game.slnを立ち上げてください。

step-1 ウィンドウの初期化

まずはウィンドウの初期化です。ウィンドウを初期化する流れはウィンドウクラスの登録、ウィンドウの作成、ウィンドウの表示でした。その流れを思い出しながら、main.cppにリスト1.1のプログラムを入力してください。

[リスト1.1 main.cpp]

```
// step-1 ウィンドウの初期化
// 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、ゲームではほぼ不要なので
CS_CLASSDCでよい。
    MsgProc,                // メッセージプロシージャ。
    0,                      // 0でいい。
    0,                      // 0でいい。
    GetModuleHandle(nullptr),   // 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。
    nullptr,                // アイコンのハンドル。今回はnullptrでいい。
    nullptr,                // マウスカーソルのハンドル。今回はnullptrでいい。
    nullptr,                // ウィンドウの背景色。今回はnullptrでいい。
    nullptr,                // メニュー名。今回はnullptrでいい。
    L"MyGame",              // ウィンドウクラスに付ける名前。
    nullptr                 // NULLでいい。
};

// ウィンドウクラスの登録。
RegisterClassEx(&wc);

// ウィンドウの作成
HWND hwnd = CreateWindow(
    L"MyGame",              // 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。
    L"MyGame",              // 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。
    1280,                   // ウィンドウの幅。
    720,                    // ウィンドウの高さ。
    nullptr,                // 親ウィンドウ。今回はnullptrでいい。
    nullptr,                // メニュー。今回はnullptrでいい。
    hInstance,              // アプリケーションのインスタンス。
    nullptr                 // WM_CREATEメッセージのlParamパラメータとして渡される引数。
);
// 作成したウィンドウを表示状態にする。
ShowWindow( hwnd, nCmdShow );
```

さて、入力してもらうとわかると思いますが、多くのパラメータにnullptrが指定されています。このテキストではこれらのパラメータについて多くは説明しませんが、ゲームであればこれらのパラメータは多くの

場合でnullptrで十分です。もちろんこれらのパラメーターを利用するゲームもあると思いますが、必要になった時に調べるで十分です(ゲームのアイコンなどは売り物のゲームを作るのであれば変更する必要がありますね)。

step-2 メッセージループを実装する

　続いて、メッセージループの実装です。main.cppの該当するコメントの箇所にリスト1.2のプログラムを入力してください。
[リスト1.2 main.cpp]

```
// step-2 メッセージループを実装する。
MSG msg = { 0 };
// 終了メッセージが送られてくるまでループを回す。
while (WM_QUIT != msg.message) {
    // PeekMessage()関数を利用して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))
    {
        // 仮想キーメッセージを文字メッセージに変換する。
        // ユーザーからの文字入力を取得する必要がある場合は、本関数を呼び出す。
        TranslateMessage(&msg);
        // メッセージを処理する。
        // この関数の中でウィドウクラスで指定したメッセージプロシージャー関数が呼ばれる。
        DispatchMessage(&msg);
    }
}
```

step-3 メッセージプロシージャを実装する。

　最後にメッセージプロシージャを実装しましょう。main.cppにリスト1.3のプログラムを入力してください。

[リスト1.3 main.cpp]

```
// step-3 メッセージプロシージャを実装する。
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_LBUTTONDOWN:
        // マウスの左ボタンが押された。
        MessageBox(hWnd, L"マウスの左ボタンが押された!", L"通知", MB_OK);
        break;
    case WM_DESTROY:
        //  削除メッセージが来たので終了させる。
        PostQuitMessage(0);
        break;

    default:
```

```
            // それ以外はデフォルトの処理に飛ばす。
            return DefWindowProc(hWnd, msg, wParam, lParam);
        }

        return 0;
    }
```

　このメッセージプロシージャではWM_LBUTTONDOWN(マウスの左ボタンが押された)とWM_DESTROY(終了メッセージ)メッセージを処理しています。WM_DESTROYメッセージはアプリの×ボタンを押すなど、ユーザーがアプリを終了させる操作を行ったときに送られてくるメッセージです。今回のプログラムでは、WM_DESTROYメッセージを受け取ると、PostQuitMessage()関数を利用して、WM_QUITメッセージをメッセージキューにポストしています。このメッセージがポストされることで、リスト1.2のメッセージループを抜けることができるようになります。WM_LBUTTONDOWNとWM_DESTROYメッセージ以外はデフォルトのウィンドウメッセージ処理を行ってくれるDefWindowProc()関数を呼び出ししています。入力出来たら実行してみてください。図1.1のような何も表示されていないウィンドウが表示されたら成功です。

[図1.1]

1.6 評価テスト

下記のURLの評価テストを実施しなさい。
評価テスト

Chapter 2 DirectXの初期化～毎フレームの処理

　このChapterではDirectXの初期化～ゲームループでの毎フレームの処理を見ていきます。

2.1 DirectXとは？

　では、まずそもそもDirectXとはいったい何なのか見ていきましょう。DirectXとはマイクロソフトが開発したゲーム・マルチメディア処理用のAPI群です。これまでは主に3DCGを扱うDirect3Dを勉強してきましたが、それ以外にも次のようなAPIがあります。

名称	説明
DirectX Graphics Infrastructure	Direct3D 10以降のグラフィックス基盤API。デバイスとの通信など、ローレベルタスクを担当する。

名称	説明
Direct2D	Windows 7以降で利用可能な、新しい2次元グラフィックスAPI。バージョン1.0はWindows Vistaにもバックポートされた。
DirectWrite	Windows 7以降で利用可能な、新しい高品位テキスト描画API。バージョン1.0はWindows Vistaにもバックポートされた。
XAudio2	クロスプラットフォーム (WindowsとXbox) で共通に使える低レベルオーディオAPI
X3DAudio	WindowsとXboxの両方のプラットフォーム上で利用可能な、空間音響用ヘルパーライブラリ
DirectAnimation	2D Webアニメーション用
DirectX Media Objects	エンコーダー、デコーダー、エフェクトといったストリーミングオブジェクトのサポート。
Direct Storage	ゲームローディング（読み込み）の高速化。

DirectXの進化の歴史の大半はDirect3Dになるのですが、実はこのように様々なAPIが存在しています。このテキストでは主にDirect3Dを扱いますが、グラフィックカードなどのデバイスと通信を行うために、DirectX Graphics Infrastructure(DXGI)なども利用します。このテキストでは扱いませんが、本校のゲームエンジンではサウンドの再生にXAudio2とX3DAudioを利用しています。

2.2 Direct3Dの初期化

では、Direct3Dの初期化の処理について見ていきましょう。Direct3Dの初期化処理は非常に煩雑で大量のコードを書く必要があります。これらのコードの大半は誰が書いても同じようになりますし、一度書いてしまえば変更が発生することも少ないです。そのため、初期化処理の詳細を全て把握できていなかったとしてもゲームは作れますし、大半のゲーム開発者は生涯この知識を使うことはないかもしれません。そもそもDirectXを利用しない、AndroidやiOS向けのゲームの開発やPlayStation、NitendoSwitchの開発などではDirectXは利用しません。WindowsPCやXBox向けのゲームであればDirectXを利用しますが、大半の開発者はUnrealEngineやUnityを使いますし、独自のゲームエンジンを使っている開発者であっても、エンジン開発を行っている一部、しかもエンジン開発の分野においてもグラフィックエンジンの開発を行っている一部の開発者のみです。

ですので、これから非常に煩雑で、おそらく面白みの少くない処理について勉強していくことになりますが、全てを完全に把握する必要はありません。ただ、その中でもどのプラットフォームでも通用する知識というものはあります。例えばフレームバッファの知識や描画コマンドなどの知識です。エンジン開発などは行わないにしても、これらの知識があることによってRenderDocなどを利用したグラフィックデバッグの助けにはなることでしょう。

2.2.1 Direct3Dの初期化～D3Dデバイスの初期化～

では、まずはD3Dデバイスの初期化について見ていきましょう。D3Dデバイスとは GPUにアクセスするためのインターフェースです。ようは、D3Dデバイスを作成することによって、C++側からGPUに対して様々な命令を行うことができるようになります。D3DデバイスはDirect3DのD3D12CreateDevice()関数を利用することで作成することができます。リスト2.1はD3Dデバイスを作成する疑似コードです。

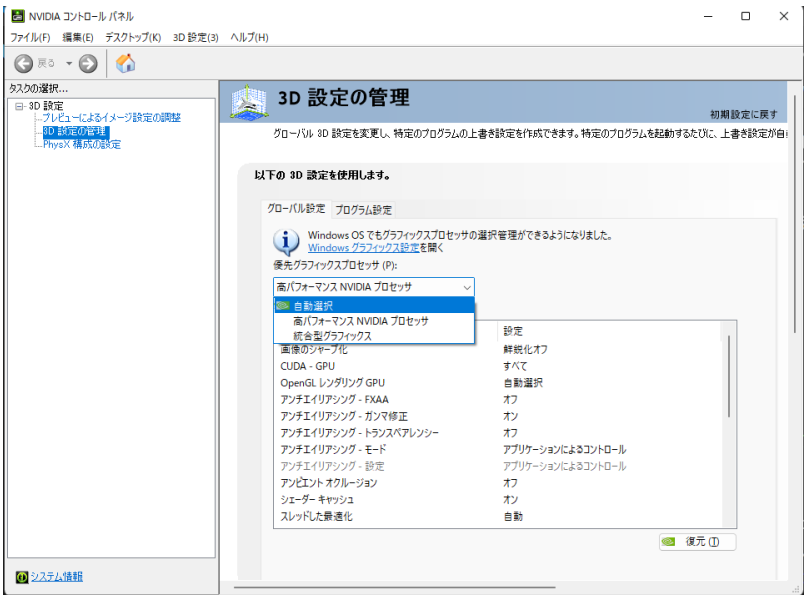
[リスト2.1]

```
ID3D12Device5* d3dDevice = nullptr;
D3D12CreateDevice(
    nullptr,                    // 使用するGPUのアダプタ。
    D3D_FEATURE_LEVEL_12_1,     // DirectX12のバージョン。
    IID_PPV_ARGS(&d3dDevice)    // D3Dデバイスにアクセスするインターフェース。
);
```

D3Dデバイスへの通信はID3D12Device*に格納されたD3Dデバイスオブジェクトのアドレスに参照して行います。ID3D12Deviceインターフェースは複数のバージョンがあり、使用する機能に応じて適切なインターフェースを使用する必要があります。疑似コードではID3D12Device5を利用しています。

D3D12CreateDevice()関数の第一引数は使用するGPUのアダプタとなっています。nullptrが指定されると、デフォルトのGPUが利用されます。複数のGPUが載っているパソコン、例えばゲーミングノートパソコンであれば、Intel製のGPUとNVIDIA製のGPUが乗っている場合があったりします。そのような時に、nullptrを指定しているIntel製のGPUが使われるか、NVIDIA製のGPUが使われるかは、パソコンの設定次第となります(図2.1)。

[図2.1 優先使用されるGPUの設定]



このような設定の影響を受けずに、使用するGPUを指定したい場合、例えばNVIDIAのGPUが載っている場合はそれを優先したいなどには、DXGIを使って、パソコンに載っているGPUアダプタを列挙して、使いたいアダプタを指定するということもできます。リスト2.2のコードは、NVIDIA製のアダプタを最優先で使用している疑似コードです。興味がある方は参照してみてください。

[リスト2.2]

```
enum GPU_Vender {
    GPU_VenderNvidia,   //NVIDIA
    GPU_VenderAMD,      //AMD
    GPU_VenderIntel,    //Intel
    Num_GPUVender,
};

ComPtr<IDXGIAdapter> adapterTmp = nullptr;
// 各ベンダーのアダプターを記憶する配列。
```

```
ComPtr<IDXGIAdapter> adapterVender[Num_GPUVender] = { nullptr };
// 最終的に使用するアダプタ。
ComPtr<IDXGIAdapter> useAdapter = nullptr;
SIZE_T videoMemorySize = 0;
// IDXGIFactory::EnumAdapters()関数を利用して、接続されているGPUアダプタを列挙していく。
for (int i = 0; dxgiFactory->EnumAdapters(i, &adapterTmp) != DXGI_ERROR_NOT_FOUND;
i++) {
    // IDXGIAdapter::GetDesc()関数を利用してアダプタ情報を取得。
    DXGI_ADAPTER_DESC desc;
    adapterTmp->GetDesc(&desc);
    if (wcsstr(desc.Description, L"NVIDIA") != nullptr) {
        // NVIDIA製
        adapterVender[GPU_VenderNvidia] = adapterTmp;
    }
    else if (wcsstr(desc.Description, L"AMD") != nullptr) {
        // AMD製
        adapterVender[GPU_VenderAMD] = adapterTmp;
    }
    else if (wcsstr(desc.Description, L"Intel") != nullptr) {
        // Intel製
        adapterVender[GPU_VenderIntel] = adapterTmp;
    }
}
// 使用するアダプターを決める。
if (adapterVender[GPU_VenderNvidia] != nullptr) {
    // NVIDIA製が最優先
    useAdapter = adapterVender[GPU_VenderNvidia];
}
else if (adapterVender[GPU_VenderAMD] != nullptr) {
    // 次はAMDが優先。
    useAdapter = adapterVender[GPU_VenderAMD];
}
else {
    // NVIDIAとAMDのGPUがなければビデオメモリが一番多いやつを使う。
    useAdapter = adapterMaxVideoMemory;
}

// D3Dデバイスを作成。
D3D12CreateDevice(
    useAdapter,
    D3D_FEATURE_LEVEL_12_1,
    IID_PPV_ARGS(&m_d3dDevice)
);
```

2.2.2 DirectXの初期化～コマンドアロケータ、コマンドリスト、コマンドキュー

　続いて、コマンドアロケータ、コマンドリスト、コマンドキューについて見ていきます。「コマンド」というのは「命令」という意味で、これらは全てGPUに対する命令に関連する項目です。CPUの仕事はこの命令、描画コマンドを作成していって、GPUに送るという者になります。リスト2.3のコードは描画コマンドを作成している疑似コードです。

[リスト2.3]

```
unsigned int commands[100];
commands[0] = 10;        // 10は頂点バッファをセット。
commands[1] = 0x1234;    // 頂点バッファのアドレス。
commands[2] = 20;        // 20はインデックスバッファをセット。
commands[3] = 0x2345;    // インデックスバッファのアドレス。
commands[4] = 100;       // 100は描画キック
commands[5] = -1;        // コマンドの終わりを表す番兵

// GPUに命令を送る。
Submit( commands );
```

　このコードはあくまで疑似コードですので、これでGPUに命令が送られるわけではないので注意してください。しかし、このコードはC++側で行っている描画コマンドの作成の処理を非常にシンプルに表しているものです。これから描画コマンドを生成するための複雑なプログラムを見ていきますが、本質的には疑似コードのような処理をしていると考えてください。

コマンドアロケータ

　コマンドアロケーターはコマンドを記憶するためのメモリ領域を確保するために使われるものです。コマンドアロケータによって確保されたメモリ上にコマンドが記憶されていきます。リスト2.4のコードはコマンドアロケーターの疑似コードです。
[リスト2.4]

```
int currentCommandPos = 0;
char commandAllocator[10*1024];

void* AllocCommand( int size )
{
    // 書き込み可能なアドレスを取得。
    void* mem = &commandAllocator[currentCommandPos];
    // 確保されたサイズ分だけ書き込み可能な位置を動かす。
    currentCommandPos += size;
    // 書き込み可能アドレスを返す。
    return mem;
}

int main()
{
    // 描画コマンドの先頭アドレス。
    void* commandTopAddres = commandAllocator;
    // コマンドを書き込む
    // 頂点バッファを設定する。
    // 頂点バッファの設定は8バイト使う。
    int* newCommand = (int*)AllocCommand( 8 );
    newCommand[0] = 10;      // 10が頂点バッファを設定する命令(4バイト)
    newCommand[1] = 0x1234; // 頂点バッファのアドレス(4バイト)

    // インデックスバッファを設定する。
    // インデックスバッファの設定は8バイト使う。
```

```
    newCommand = (int*)AllocCommand( 8 );
    newCommand[0] = 20;     // 20がインデックスバッファを設定する命令(4バイト)
    newCommand[1] = 0x2345; // インデックスバッファのアドレス(4バイト)

    // ドローコール
    // ドローコールの設定は4バイト使う。
    newCommand = (int*)AllocCommand( 4 );
    newCommand[0] = 100;    // 100がドローコールの命令(4バイト)

    // 最後にコマンドの終了の番兵を設定する。
    newCommand = (int*)AllocCommand( 4 );
    newCommand[0] = -1;     // -1が番兵

    // 生成されたコマンドをサブミット
    Submit(commandTopAddres);
}
```

リスト2.3のコードと比べると複雑になってきましたが、本質的には2.3のコードと同じです。コマンドアロケータを使って、コマンドの書き込み先のメモリを確保して、そこにコマンドを書き込んでいっているだけです。

コマンドリスト

続いてコマンドリストです。コマンドリストは作成されたコマンドのアドレスを記憶していくリストです。リスト2.4のコードはコマンドリストを利用している疑似コードです。
[リスト2.4]

```
// これがコマンドリスト。コマンドのアドレスを記憶していく。
std::list<void*> commandList;

int currentCommandPos = 0;
char commandAllocator[10*1024];

void* AllocCommand( int size )
{
    // 書き込み可能なアドレスを取得。
    void* mem = &commandAllocator[currentCommandPos];
    // 確保されたサイズ分だけ書き込み可能な位置を動かす。
    currentCommandPos += size;
    // 書き込み可能アドレスを返す。
    return mem;
}

int main()
{
    // 描画コマンドの先頭アドレス。
    void* commandTopAddres = commandAllocator;
    // コマンドを書き込む
    // 頂点バッファを設定する。
    int* newCommand = (int*)AllocCommand( 8 );
    newCommand[0] = 10;     // 10が頂点バッファを設定する命令(4バイト)
```

```
    newCommand[1] = 0x1234; // 頂点バッファのアドレス(4バイト)

    //【注目】コマンドリストに追加
    commandList.push_back(newCommand);

    // インデックスバッファを設定する。
    // インデックスバッファの設定は8バイト使う。
    newCommand = (int*)AllocCommand( 8 );
    newCommand[0] = 20;     // 20がインデックスバッファを設定する命令(4バイト)
    newCommand[1] = 0x2345; // インデックスバッファのアドレス(4バイト)

    //【注目】コマンドリストに追加
    commandList.push_back(newCommand);

    // ドローコール
    // ドローコールの設定は4バイト使う。
    newCommand = (int*)AllocCommand( 4 );
    newCommand[0] = 100;    // 100がドローコールの命令(4バイト)

    //【注目】コマンドリストに追加
    commandList.push_back(newCommand);

    // 最後にコマンドの終了の番兵を設定する。
    newCommand = (int*)AllocCommand( 4 );
    newCommand[0] = -1;     // -1が番兵

    //【注目】コマンドリストに追加
    commandList.push_back(newCommand);

    // 生成されたコマンドをサブミット
    Submit(commandList);
}
```

コマンドキュー

　最後はコマンドキューです。コマンドキューはFIFO(先入れ先だし)のデータ構造になっており、複数のコマンドリストをリストをキューに詰むことができます。DirectX12では最終的に、このコマンドキューにコマンドリストを積んで、GPUにコマンドを送ります。リスト2.5のコードはコマンドキューの疑似コードです。

[リスト2.5]

```
// これがコマンドキュー。
std::queue<std::list<void*>> commandQueue;
// コマンドリスト。
std::list<void*> commandList;
// コマンドアロケータ。
int currentCommandPos = 0;
char commandAllocator[10*1024];

void* AllocCommand( int size )
{
```

```
    // 書き込み可能なアドレスを取得。
    void* mem = &commandAllocator[currentCommandPos];
    // 確保されたサイズ分だけ書き込み可能な位置を動かす。
    currentCommandPos += size;
    // 書き込み可能アドレスを返す。
    return mem;
}

int main()
{
    // 描画コマンドの先頭アドレス。
    void* commandTopAddres = commandAllocator;
    // コマンドを書き込む
    // 頂点バッファを設定する。
    int* newCommand = (int*)AllocCommand( 8 );
    newCommand[0] = 10;     // 10が頂点バッファを設定する命令(4バイト)
    newCommand[1] = 0x1234; // 頂点バッファのアドレス(4バイト)

    // コマンドリストに追加
    commandList.push_back(newCommand);

    // インデックスバッファを設定する。
    // インデックスバッファの設定は8バイト使う。
    newCommand = (int*)AllocCommand( 8 );
    newCommand[0] = 20;     // 20がインデックスバッファを設定する命令(4バイト)
    newCommand[1] = 0x2345; // インデックスバッファのアドレス(4バイト)
    // コマンドリストに追加
    commandList.push_back(newCommand);

    // ドローコール
    // ドローコールの設定は4バイト使う。
    newCommand = (int*)AllocCommand( 4 );
    newCommand[0] = 100;    // 100がドローコールの命令(4バイト)
    // コマンドリストに追加
    commandList.push_back(newCommand);

    // 最後にコマンドの終了の番兵を設定する。
    newCommand = (int*)AllocCommand( 4 );
    newCommand[0] = -1;     // -1が番兵
    // コマンドリストに追加
    commandList.push_back(newCommand);

    //【注目】コマンドリストをキューに入れる。
    commandQueue.push( commandList );

    // 生成されたコマンドをサブミット
    Submit(commandQueue);
}
```

ここまでの一連の流れがDirectX12に置けるコマンド生成です。コマンドアロケータでコマンドを記憶する領域を確保して、そこにコマンドを書き込む。そのコマンドアドレスをコマンドリストに登録して、最終的にはコマンドリストをコマンドキューに詰んでGPUにコマンドを発行します。リスト2.6は実際にDirectX12を

利用した際のコマンドアロケータ、コマンドリスト、コマンドキューの活用のサンプルコードです。
[リスト2.6]

```
// D3Dデバイスを介してコマンドアロケータの作成する。
ID3D12CommandAllocator* commandAllocator = nullptr;
d3dDevice->CreateCommandAllocator(
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    IID_PPV_ARGS(&commandAllocator));

// D3Dデバイスを介してコマンドリストを作成する。
ID3D12GraphicsCommandList4* commandList = nullptr;
d3dDevice->CreateCommandList(
    0,
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    commandAllocator),
    nullptr,
    IID_PPV_ARGS(&commandList)
);

// D3Dデバイスを介してコマンドキューを作成する。
D3D12_COMMAND_QUEUE_DESC queueDesc = {};
queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
ID3D12CommandQueue commandQueue = nullptr;
d3dDevice->CreateCommandQueue(
    &queueDesc,
    IID_PPV_ARGS(&commandQueue)
);

// ゲームループ
while(true){
    // コマンドアロケータとコマンドリストをリセットする。
    commandAllocator->Reset();
    commandList->Reset(
        commandAllocator,   // 使用するコマンドアロケータを指定する。
        nullptr
    );

    // ここから描画コマンドをコマンドリストに詰んでいく。
    // キャラの頂点バッファを設定
    commandList->IASetVertexBuffers(0, 1, charaVertexBuffer);
    // キャラのインデックスバッファを設定
    commandList->IASetIndexBuffer( charaIndexBuffer);
    // キャラをドローする。
    commandList->DrawIndexedInstanced( charaIndexCount, 1, 0, 0, 0);

    // 背景の頂点バッファを設定
    commandList->IASetVertexBuffers(0, 1, bgVertexBuffer);
    // 背景のインデックスバッファを設定
    commandList->IASetIndexBuffer( bgIndexBuffer);
    // 背景をドローする。
    commandList->DrawIndexedInstanced( bgIndexCount, 1, 0, 0, 0);
```

```
    // コマンドリストをクローズする(番兵)。
    commandList->Close();

    // コマンドキューに詰むためにコマンドリストの配列を定義する。
    ID3D12CommandList* commandListArray[] = {
        commandList
    };

    // コマンドキューにコマンドリストを詰む。
    commandQueue->ExecuteCommandLists(
        1,                  // コマンドリストの数。
        commandListArray    // コマンドリストの配列。
    );
}
```

コマンドキューを使用する理由は、複数のコマンドリストを使えるようにするためです。
　近年のゲームは表示されるオブジェクトの数が大量になっており、描画コマンドの作成が大きなCPU側のボトルネックとなっていました。そこで、複数のスレッドで並列に描画コマンドを作成することができれば、CPUのボトルネックを大きく解消することができます。しかし、単純に描画コマンドの生成処理を並列にすればいいだけではありません。GPUの描画処理というのは処理の順番が重要になってきます。ですので、リスト2.5

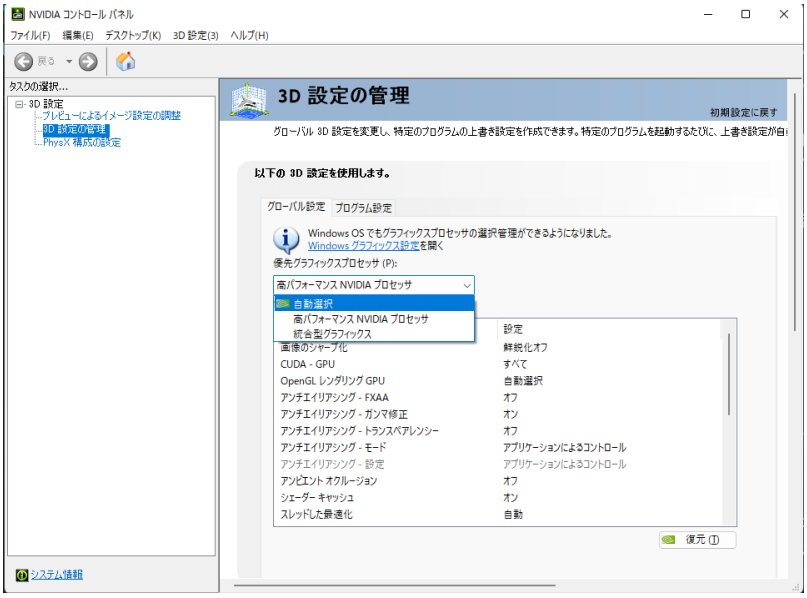
Chapter 2 リソースバインディング

　このチャプターではDirectX12におけるリソースバインディングの仕組みについて見てきます。

2.1 リソースバインディングとは？

　では、リソースバインディングについて解説をしたいのですが、これを解説するには昨今のGPUのアーキテクチャについて多少の前提知識が必要になるので、それについて勉強していきましょう。　昨今の高品質なゲームを快適に遊ぶためにはグラフィックカードが必須となります。現在のグラフィックカードは主にNVIDIA製のものとAMD製のモノがあります。(IntelのCPUに内蔵されているUHDグラフィックスなどのインテル製のチップもありますが、このチップはあくまでも一般ユーザー向けのチップとなっており、高品質なゲームを遊びたいゲームユーザー向けではないため、割愛します。)　これらのグラフィックカードの構成を大雑把に説明すると、演算を行うGPU、テクスチャなどのデータを記憶するVRAMに分類することができます(図2.1)。

[図2.1 GPUとVRAM～その1]



さて、グラフィックカードを大雑把に見てみるとGPUとVRAMがあることが分かりました。では、もう少し詳細にGPUのアーキテクチャについて見ていきましょう。図2.2を見てください。

[図2.2 GPUとVRAM～その2]

GPUの内部には多数のコアと呼ばれる演算基が含まれており、前述したシェーダープログラムの実行はこのコアを使って並列に実行されます。例えば、100万頂点のモデルの描画コールを実行した場合、このコアで分担して100万頂点の頂点シェーダーを実行していくことになります。また、VRAMにはシーンを描画するために必要な各種リソースが乗っています。さて、そろそろリソースバインディングに関する話に近づいているのですが、もう少しだけグラフィックカードのアーキテクチャを詳細に見ていきましょう。図2.3を見てください。

[図2.3 GPUとVRAM～その3]

図2.3ではコアの内部情報を詳細に記載しています。コアの内部には数値演算を行うための演算器と高速なメモリのレジスタがあります。この演算器でシェーダープログラムを実行していると考えてください。ここで重要なのはレジスタです。シェーダープログラムが直接アクセスできるメモリはこのレジスタに乗っているデータになります。さて、ここで少しGPUの動きを考えてみましょう。アプリケーション側からユニティちゃんを描画するためのドローコールが実行されるとGPUはユニティちゃんを描画するためにレンダリングパイプラインを実行していきます。このパイプライン上に頂点処理を行う頂点シェーダーやピクセル処理を行うピクセルシェーダーが実行されていくわけです。この時、頂点シェーダーでは当然VRAMに乗っているユニティちゃんの頂点バッファやインデックスバッファにアクセスする必要があります。ピクセルシェーダーではユニティちゃんのテクスチャにアクセスる必要があります。しかし、VRAMにはその他のオブジェクトの頂点バッファやテクスチャといったグラフィックリソースもVRAMに乗っているのです。では、GPUはどのようにして多数あるVRAM上のリソースから、Unityちゃんのグラフィックリソースを選べばいいのでしょうか。答えを言うと、GPUが自動的にユニティちゃんのグラフィックリソースを選ぶということはできません。プログラマが明示的にプログラムを記載して、ユニティちゃんのグラフィックリソースを指定する必要があります。もう少し具体的に言うと、GPUの各種レジスタに使用するグラフィックリソースのアドレスを設定するのです(図2.4)。

[図2.4 GPUとVRAM～その4]

イメージとしてはC++のポインタをイメージしてみてください。レジスタに使用するリソースのアドレスを設定して、そのアドレスを使ってリソースをロードしていきます。そしてこのレジスタと使用するリソー

スを関連付けすることをリソースバインディングといいます。DirectX12を利用して絵を描画するためには、後述するディスクリプタ、ディスクリプタヒープ、そしてルートシグネチャを活用して、描画したい絵に必要なリソースとレジスタの関連付けを行う必要があります。

2.1 ディスクリプタ

まずはディスクリプタから見ていきます。ディスクリプタとはグラフィックメモリ内のリソースの情報が記述されているデータです。リソースというのは、単なるメモリの塊となります。ディスクリプタはそのメモリの塊がなんなか、テクスチャなのか定数バッファなのかはたまたストラクチャードバッファなのか、といった情報が記述されています(図2.5)。
[図2.5 ディスクリプタ]


DirectX12のAPIを使ったリソースバインディングでは各種レジスタと、このディスクリプタを関連付けしていくことになります。

2.2 ディスクリプタヒープ

続いてディスクリプタヒープです。これはディスクリプタを記憶するためのメモリ領域です。ディスクリプタの配列のようなものだと考えてもらって買いません。DirectX12ではディスクリプタを作成する前に、ディスクリプタヒープを確保して、そこにリソース情報(ディスクリプタ)を記憶していきます(図2.6)。
[図2.6 ディスクリプタヒープ]


ディスクリプタヒープは単にディスクリプタを記憶するためのメモリ領域です。その領域にリソースの情報を記述していくことになります。下記のコードはディスクリプタヒープとディスクリプタの関係を示したC++の疑似コードです。

```
struct Descriptor{
    void* addr; // リソースのアドレス。
    int type;   // リソースの種類。
}
// ディスクリプタヒープを100バイト確保。
void* descriptorHeap = malloc( 100 );

Descriptor* descriptor = (Descriptor*)descriptorHeap;
// テクスチャの情報を書き込む。
descriptor[0].addr = 0x100;     // アドレス。
descriptor[0].type = 1;         // 1はテクスチャを表している。
// 次は定数バッファ。
descriptor[1].addr = 0x200;     // アドレス
descriptor[1].type = 2;         // 2は定数バッファを表している。
```

2.3 ルートシグネチャとディスクリプタテーブル

最後にルートシグネチャとディスクリプタテーブルです。正確にはここの話はディスクリプタテーブルに関する説明なのですが、ディスクリプタテーブルの情報はルートシグネチャの内部のデータとなるので、ルートシグネチャとディスクリプタテーブルという節タイトルになっています。ディスクリプタテーブルはレジスタ番号からディスクリプタのアドレスに変換するためのテーブルです。

2.2節までディスクリプタとディスクリプタヒープについて話をしてきましたが、結局ディスクリプタとレ

ジスタをどのように関連付けるのかは分かりませんでした。この関連付けを行うのがディスクリプタテーブルです。
　ディスクリプタテーブルの話に入る前に、そもそもテーブルというのが何なのか解説します。
テーブルというのは、何かのデータ(番号など)を使って、別のデータを取り出すためのデータ構造のことを指します。例えば、学生の出席番号を使って、学生の年齢を取得するテーブルについて考えてみましょう。このような場合C++であれば１次元配列が利用されます。次の疑似コードを見てください。

```
// 年齢は出席番号の順番で格納されている。
int studentAgeTable[5] = { 20, 19, 21, 19, 20 };
// 出席番号4番の学生の年齢を表示。配列の添え字は0から始まるので、出席番号-1を添え字とする。
printf("出席番号4番の学生の年齢 = %d", studentAgeTable[3]);
// 出席番号2番の学生の年齢を表示。配列の添え字は0から始まるので、出席番号-1を添え字とする。
printf("出席番号2番の学生の年齢 = %d", studentAgeTable[1]);
```

　いかがでしょうか。それほど難しい話ではなかったと思います。ディスクリプタテーブルもこれと全く同じです。ディスクリプタテーブルはレジスタ番号を使って、ディスクリプタのアドレスを取得します。下記のコードはディスクリプタテーブルを使って、レジスタとディスクリプタを関連付ける疑似コードです。

```
void* descriptorTable[5] = { 0x1FFF, 0x2FFFF, 0x0123, 0x3A5D, 0xB29F };
// 0番のレジスタに割り当てられているディスクリプタのアドレスを取得。
void* desciptorAddr_0 = descriptorTable[0];
// 2番のレジスタに割り当てられているディスクリプタのアドレスを取得。
void* desciptorAddr_2 = descriptorTable[2];
```

図2.7にDirectX12のリソースバインディングのイメージ図を示します。
[図2.7 リースバインディングのイメージ図]


2.4 リソースバインディングの流れ

　この節ではリソースバインディングの流れを疑似コードを交えて解説します。まずは大雑把な流れを押さえておきましょう。まずは初期化の流れです。

1. ディスクリプタヒープを確保
2. ディスクリプタヒープにディスクリプタの情報を書き込んでいく。
3. ディスクリプタテーブルを作成(ルートシグネチャを作成)　続いて、毎フレームのドローコールです。
4. ディスクリプタヒープを設定。
5. ディスクリプタテーブルを設定。
6. ドローコールを実行。

　では、まずは初期化の疑似コードを見ていきましょう。なお、今回はディスクリプタヒープに複数のオブジェクトのリソースが登録されているとします。

```
struct Descriptor{
    void* addr;      // リソースのアドレス。
```

```
    int type;       // リソースの種類。
};

// ディスクリプタヒープ。
Descriptor* g_descriptorHeap = nullptr;
// モンスターのデスクリプタテーブル。
Descriptor* g_monsterDescriptorTbl[10];
// 地面のディスクリプタテーブル。
Descriptor* g_groundDescriptorTbl[10];


void Init()
{
    const int MAX_DESCRIPTOR = 1000;    // ディスクリプタの最大数。
    // ディスクリプタヒープを確保する。
    g_descriptorHeap = malloc(sizeof(Descriptor) * 1000);
    // 怪物のモデルを描画するためのリソースをディスクリプタを書き込んでいく。
    // アルベドマップ
    g_descriptorHeap[0].addr = 0x1000;
    g_descriptorHeap[0].type = 1;
    // 法線マップ
    g_descriptorHeap[1].addr = 0x2000;
    g_descriptorHeap[1].type = 1;
    // メタリックスムースマップ
    g_descriptorHeap[2].addr = 0x2400;
    g_descriptorHeap[2].type = 1;

    // 続いて地面のモデルを描画するためのリソースをディスクリプタに書き込んでいく。
    // アルベドマップ
    g_descriptorHeap[3].addr = 0x3000;
    g_descriptorHeap[3].type = 1;
    // 法線マップ
    g_descriptorHeap[4].addr = 0x4000;
    g_descriptorHeap[4].type = 1;
    // スペキュラマップ
    g_descriptorHeap[5].addr = 0x4400;
    g_descriptorHeap[5].type = 1;

    // 続いてモンスターのディスクリプタテーブルを初期化する。
    // 0番レジスタはアルベドテクスチャ
    g_monsterDescriptorTbl[0] = &g_descriptorHeap[0];
    // 1番レジスタは法線マップ
    g_monsterDescriptorTbl[1] = &g_descriptorHeap[1];
    // 2番レジスタはメタリックスムースマップ
    g_monsterDescriptorTbl[2] = &g_descriptorHeap[2];

    // 続いて地面のディスクリプタテーブルを初期化する。
    // 0番レジスタはアルベドテクスチャ
    g_groundDescriptorTbl[0] = &g_descriptorHeap[3];
    // 1番レジスタは法線マップ
    g_groundDescriptorTbl[1] = &g_descriptorHeap[4];
    // 2番レジスタはメタリックスムースマップ
    g_groundDescriptorTbl[2] = &g_descriptorHeap[5];
}
```

今回の疑似コードでは、大きなディスクリプタヒープを確保して、そこに各種モデルのディスクリプタを登録していっています。しかし、必ずしもディスクリプタヒープは１つである必要はなく、複数のディスクリプタヒープを利用することも可能です。例えば、本校のエンジンのモデル表示処理では、モデルごとに１つのディスクリプタヒープが用意されています。このディスクリプタヒープにモデルの各マテリアルに設定されチエルテクスチャのディスクリプタを登録していますいます。では、続いて、毎フレームのドローコールの疑似コードを見ていきましょう。

```
void Render()
{
    // ディスクリプタヒープを設定。
    g_renderContext.SetDescriptorHeap(g_descriptorHeap);
    // モンスターを描画
        ・
        ・　他にも頂点バッファやインデックスバッファなど色々設定してるが省略。
        ・
    // モンスターのディスクリプタテーブルを設定。
    g_renderContext.SetDescriptorTable(g_monsterDescriptorTbl);
    // ドロー。
    g_renderContext.Draw();

    // 地面を描画
        ・
        ・　他にも頂点バッファやインデックスバッファなど色々設定してるが省略。
        ・
    // 地面のディスクリプタテーブルを設定。
    g_renderContext.SetDescriptorTable(g_groundDescriptorTbl);
    // ドロー。
    g_renderContext.Draw();
}
```

2.5

Chapter2 パイプラインステート

Chapter3 Zテスト

Chapter4 アルファテスト

Chapter5 カリング