（草稿）

DirectX12の初期化

ウィンドウ生成

{

LRESULT CALLBACK WindowProcedure(\_In\_ HWND hwnd, In\_ UINT uMsg,\_In\_ WPARAM wParam,\_In\_ LPARAM lParam);

//プロトタイプ宣言

1,CreateWindow,ShowWindowを呼ぶ

コード

int WinMain(HINSTANCE hInstance, HINSTANCE hPreInst, TCHAR\* lpszCmdLine, int nCmdShow)

{

const int Window\_width =1280;

const int Window\_height =640;

HWND hwnd;

WNDCLASSEX windowclass\_ex;

windowclass\_ex.cbsize =sizeof(WNDCLASSEX);

windowclass\_ex.lpfnWndProc = WndProc;

windowclass\_ex.lpszClassName = \_T("てきとー");

windowclass\_ex.hInstance = hInstance;

//ShowWindow,CreateWindowを呼ぶのに必要。

RegisterClassEx(&windowclass\_ex);

//メインウインドウを作成する

CreateWindow(windowclass\_ex.lpszClassName,

\_T("テキトー"),

WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT,

CW\_USEDEFAULT,

//表示x,y座標はOSに任せる

,

Window\_width,

Window\_Height,

nullptr,

nullptr,

windowClass\_ex.hInstance,

nullptr);

ShowWindow(hwnd,SW\_SHOW);

}

2,メイン関数にウィンドウプロシージャー<(アプリケーションに送られてきたメッセージを処理するための関数)を作成する

{

コード

LRESULT CALLBACK WindowProcedure(

\_In\_ HWND   hwnd,

  \_In\_ UINT   uMsg,

  \_In\_ WPARAM wParam,

  \_In\_ LPARAM lParam)

{

case:WM\_DESTROY:

{

PostQuitMessage(0);

//OSに対して「もうこのアプリは終わると伝える」

}

return 0;

}

return DefWindowProc(hwnd,msg,wparam,lparam);

//既定の処理を行う

https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-defwindowproca

}

3.ループ処理を行い、ウインドウを出したままにする

int WinMain(HINSTANCE hInstance, HINSTANCE hPreInst, TCHAR\* lpszCmdLine, int nCmdShow)

{

<ShowWindowの続き>

MSG msg ={};

while(true)

{

if(PeekMessage(&msg,nullptr,0,0,

PM\_REMOVE))

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

if(msg.message == WM\_QUIT)

{

break;

}

}

UnregisterClass(windowclass\_ex.lpszClassName,windowclass\_ex.hInstance);

https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-unregisterclassa

デバイス生成

{

デバイスとは？

コンピューターアーキテクチャ的には

{

デバイス自体は端末を指す。

スマホ、タブレットを思い起こしてもらえば簡単だろう。IoTもそうである。ただし、DirectX12においては、むしろデバイスドライバーとしての意味あいが強いと思われる。デバイスドライバーとは「デバイスを制御するプログラム」だ。

DirectX12では

仮想アダプタの事を指す。

{

用途

{

詳しくは後々述べるが、コマンドアロケーター、コマンドアロケーター、コマンドリスト、フェンス、リソース、

パイプラインステートオブジェクト、ヒープ、ルートシグネチャ、サンプラー、各種リソースビューを生成するのに使われる。

https://docs.microsoft.com/en-us/windows/win32/api/d3d12/nn-d3d12-id3d12device

}

デバイスの初期化

1.DXGIの初期化

{  
DXGIとは{

DirectX12よりもハードウェア寄りのAPIであり、主な目的は

・スワップチェーンオブジェクト（絵を二枚用意し、表で絵を見せて、裏で絵を描く。こうすることで

絵のちらつきを防ぐ）の生成

・ウインドウのサイズ再設定を制御する

・モニターの向きに応じて描画を変えることができる。

・ディスプレイモードを変える

このように、ディスプレイに関する制御を担うのがDXGIの役割である。

}

}

ID3D12Device\* m\_Device  = nullptr;

IDXGIAdapter\* m\_dxgiAdapter = nullptr;

D3D\_FEATURE\_LEVEL level;

for(const auto& appropriateLevel:level){

if(D3D12CreateDevice(this->mpAdapter, level, IID\_PPV\_ARGS(&mpDevice));

 ==S\_OK)

{

out\_level = appropriateLevel;

break;

}

IDXGIFactory6\* pDxgiFactory = nullptr;

int iAdapter = 0;

auto result =CreateDXGIFactory2 ();

std::vector<IDXGIAdapter> Adapters ={};

for(int i =0;i<pDxgiFactory->EnumAdapterByGpuPreference(i, **DXGI\_GPU\_PREFERENCE\_HIGH\_PERFORMANCE**, IID\_PPV\_ARGS(&pAdapter)) != DXGI\_ERROR\_NOT\_FOUND)

;i++)

https://docs.microsoft.com/en-us/windows/win32/api/dxgi1\_6/nf-dxgi1\_6-idxgifactory6-enumadapterbygpupreference

{

if(D3D12CreateDevice(pAdapter,level,\_uuidof(ID3D12Device),nullptr)==SUCEEDED)

{

Adapters.push\_back(pAdapter);

}

//todo assertでAdaptersのサイズが0でないか調べる

auto out\_adapter = Adapters[0];

auto result = D3D12CreateDevice(out\_adapter,out\_level, IID\_PPV\_ARGS(&mpDevice));

}

}

<https://docs.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-d3d12createdevice>

コマンド系インターフェイス

概要

{

　DirectX12のシステムは(ここは誤解している可能性があるから間違えてたら訂正する)GPUを並列処理させて処理を高速化する処理プロセス,マルチスレッディング(multi threading)を前提としている。

コマンド系のインターフェイスはそれを制御するデータ構造である。

}

なぜこのようなものが？

{

(DirectX12のリリースされた背景については、晩御飯を食べたら調べる)

DirectX12がリリースされた背景は、「グラフィック処理を始めとする、ビデオゲームのリアルタイム処理の高速化」である。そのために主に使用される技術がマルチスレッド技術、過去のDirect3Dが担っていたパフォーマンスチューニングの多くをプログラマー側に任せるといった所だ。マルチスレッド周りに関して特に大きな処理機構の刷新が図られている。

DirectX11ではimmediate contextが同じくcommandlistがGPUの命令を記録していた

が、この仕組みは並行処理を行わせるには問題があった(Not free threaded)設計であったため、GPUへの描画命令伝達速度が低下する要因、すなわちボトルネックとなっており、マルチスレッディングの妨げとなっていた。(

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/design-philosophy-of-command-queues-and-command-lists>

"Design Philosophy of Command Queues and Command Lists"の"Removal of the immediate context"を参照。

)

そこでこのimmediate contextを削除し、代わりにCommand AllocatorとCommand QueueでGPUに対する命令を蓄積、Command listを利用する処理の流れにおいて、複数のコマンドリストに並行してGPUへの命令を記録させることで並行性を持たせ、GPUへの描画命令を並行して処理できるようにしたのがDirectX12におけるコマンド系インターフェイスだ。

"Removal of the immediate context"

https://docs.microsoft.com/en-us/windows/win32/direct3d12/design-philosophy-of-command-queues-and-command-lists

コマンドアロケータ

{

アロケーターとは

{

計算機分野(コンピューター分野)では

メモリに置くリソースの領域確保や、逆にリソースをメモリから消去するメモリ管理を行う物を指す。

DirectX12のCommand allocatorにおける機能

{

2021/06/06に記入

}

ID3D12CommandAllocatorの役割{

command allocatorの役割は、

コマンドリストに確保する(恐らくGPUのVRAM)メモリ領域を

アプリケーションが制御できるようにすることである。

注意しておきたい。**Command Allocatorこそが後述するCommand listの本体であり、**

**ここにGPUへの命令が一旦格納される。**

作成するコマンド系のオブジェクトのCommand listのタイプ（D3D12\_COMMAND\_LIST\_TYPE)はコマンドのグループの中で一致させていなければならない。すなわち,Command Allocator,Command list,Command QueueのオブジェクトのD3D12\_COMMAND\_LIST\_TYPEは同じでなければならないのだ。

)

ID3D12CommandAllocatorの作成

{

・COMMAND\_LIST\_TYPE\_DIRECTを使う（コマンドリストもDirecttypeにする。）

CreateCommandAllocator(

**D3D12\_COMMAND\_LIST\_TYPE type,<ここで決定したTYPEは、このAllocatorに**

**GPUへの命令(以下コマンド)を格納させるために使うlistや、allocatorからコマンドを伝達させるのに使うQueueのTYPEと必ず一致させる。**

**(ここは隠しておく)**

  REFIID                  riid,

  void                    \*\*ppCommandAllocator<(**IID\_PPV\_ARGS(ID3D12CommandAllocatorのポインタのさらにポインタを格納する。つまり、&コマンドアロケーターのポインタと記述するか、後々使う**

**ComPtrと呼ばれるユニークポインタ（ＧＰＵへのデータ送信に関連するデータを扱うのに安全なポインタ）ならこのユニークポインタのアドレスを解放してから生成、さらにユニークポインタのアドレスを取得するReleaseandGetAddressOf()を記述する**

**(ここは隠しておく)**

**(なぜReleaseandGetAddressOf()なのかをきちんと追求しておく。調べていく過程で**

**もしかしたら危険な可能性がある。**

);

device->CreateCommandAllocator(COMMAND\_LIST\_TYPE\_DIRECT,(コマンドアロケーターのポインタのアドレス)

}

}

Command list

{

Command listとは？

{

GPUへの描画、または状態の変更命令を遅延実行によって記録するインターフェースの事を指す。

DirectX12における一つ一つのCommand Listは並行処理をさせても問題が無いようには設計されていないNot free-threadedではあるが、複数のCommand listは並行して命令を記録していくことができる。

Command Listの種類

{

主要なものと思われるlistは４種類あり、

第一のCommand listを

Direct Command lists,

第二のCommand listがbundles,

第三はCompute Command lists,

最後がCopy Command listsとなる。

bundleは何度も使いまわす事を前提に設計されたCommand listであり、

生成時にはDirect Command listより前もって処理時間がかかるが、その分随時再利用しやすい設計となっている。

Direct Commandlistは一回のみの利用を想定しており、再利用する場合は

**以前実行された命令が確実に実行されていなければならない**のだ。

どのような実行の流れになるかは、このURLに図が掲載されているのでこちらを見る事をお勧めする

"

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/recording-command-lists-and-bundles>

"

}

コード例{

result = device->CreateCommandList(0, D3D12\_COMMAND\_LIST\_TYPE\_DIRECT, m\_RenderthreadC\_Allocater.Get(), nullptr, IID\_PPV\_ARGS(m\_RenderthreadC\_list.ReleaseAndGetAddressOf()));

m\_RenderthreadC\_list->Close();

**ここは当然だが書き換えろ。**

\*\*\*\*\*\*\*\*\*\*\*\*解説を入れる\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

注意点

{

2021/06/06 "Creating command lists"を読んでおく。

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/recording-command-lists-and-bundles>

・2021/06/07"Recording command lists"を読んでおく

https://docs.microsoft.com/ja-jp/windows/win32/direct3d12/recording-command-lists-and-bundles

}

Command Queue

{

キュー(Queue)とは？

{

データ構造の一つ。

スーパーのレジ待ち行列を考えてみよう。

行列に入った人は、先に入った順に並んでいき、会計を受けられるのは一番先頭に並んでいた人である。キューも同じであり、行列に格納したデータは先に格納されたデータ順に並び、データを追い出す時は一番先に入っているデータが追い出される。

このような先入れ先だしの構造を、FIFO(First in First Out)と呼ぶ。

C++の標準ライブラリでは,std::queueがキューに当たる。

}

コマンドキュー(Command Queue)とは？

{

キューの説明と、コマンドの性質を踏まえると、

コマンドキューは

・コマンドリストによって蓄積した描画命令のデータを先入れ先出しで

保存し、

・貯めこんだ命令データをGPUに実行させるためのデータ構造

となる。

}

}

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/design-philosophy-of-command-queues-and-command-lists>

コマンドキューの作成

{

D3D12\_COMMAND\_QUEUE\_DESC desc = {};

desc.Type = D3D12\_COMMAND\_LIST\_TYPE\_DIRECT;

desc.Flags = D3D12\_COMMAND\_QUEUE\_FLAG\_NONE;

desc.NodeMask = 0;

desc.Priority = D3D12\_COMMAND\_QUEUE\_PRIORITY\_NORMAL;

result = device->CreateCommandQueue(&desc, IID\_PPV\_ARGS(m\_RenderthreadC\_Queue.ReleaseAndGetAddressOf()));

\*\*\*\*\*\*\*\*\*\*\*\*解説を入れる\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

}

}

スワップチェーン

{

スワップチェーンとは

画面のチラツキを防ぐアイツ

Introduction To 3D Game programming with DirectX12:kindle no3346

どうやって実現する？

ダブルバッファリングを使う。

https://docs.microsoft.com/ja-jp/windows/win32/direct3d9/page-flipping-and-back-buffering

→レンダーターゲットとは？

{

フレームバッファの事だ（"フレームバッファ（Frame Buffer）とはピクセル（に描画するための値）を敷き詰めたメモリ領域のことです。フレームバッファの各点に値を書き込むと、それがディスプレイのピクセルに反映される仕組みになっています。"）

ゼロからのos自作入門PDF版 p83

バッファ（GPUリソース）

https://www.itmedia.co.jp/anchordesk/spv/0402/13/news064.html

{

GPUに処理（絵を描く、計算する等）させたいデータ（テクスチャ、3Dモデル、テクスチャ、画面、GUI,計算（演算）データ）

→（詳しくはGPUを支える技術を見るといい）

の事。

スワップチェーンではレンダーターゲットが関わる

}

}

Render Target View(レンダーターゲットビュー)

{

ビューって？

→バッファ(GPUに処理させるデータ

(しつこいとは思うが(ここは隠しておく):

ポリゴン、数式によって計算されたデータ、深度、GUI,テクスチャ)

のVRAM内におけるメモリ領域を確保しておくための緩衝材)のGPUのためのマニュアル)を定義するもの

)

(

分かりやすい記述については、DirectX12の魔導書第四章,"頂点バッファ"の節を参照(電子書籍版ではNo.2413から

)

)

{

ディスクリプタ

→ビューに加えてサンプラーを包括するもの。

（DirectX12の魔導書:kindle版 no1974）

}

つまり、レンダーターゲットビューとは…

レンダーターゲット（フレームバッファ）の使い方を定義するもので、モニターへの絵の出し方をGPUに説明するものだ。

2021/06/08)詳しく記述

}

ディスクリプタヒープの作成

{

ディスクリプタヒープ？

サンプラーに加えたGPUリソースの使い方を定義した物を格納するメモリ領域（ヒープとはメモリ領域）

2021/06/08に詳しく

}

}

スワップチェーンの作成

1:SwapChainDescを記述する

{

例{

DXGI\_SWAP\_CHAIN\_DESC1 swapChainDesc = {};

swapChainDesc.BufferCount = 2;

swapChainDesc.Height = window->GetHeight();

swapChainDesc.Width = window->GetWidth();

swapChainDesc.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;

swapChainDesc.SampleDesc.Count = 0;

swapChainDesc.SampleDesc.Quality = 1;

swapChainDesc.SwapEffect = DXGI\_SWAP\_EFFECT\_FLIP\_DISCARD;

swapChainDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

swapChainDesc.Flags = 0;

swapChainDesc.Scaling = DXGI\_SCALING\_STRETCH;

}

詳細:

BufferCount:

スワップチェーンに用いるフレームバッファ(GPUのVRAMに格納されたピクセルデータの

メモリ領域を確保する緩衝材。ここからモニターに出力される絵を持ってくる)

の数を指す。

ここはFPS(Frame Per Second:一秒間に何枚絵を出すかを示した指標)に応じて

入力することになるが、ここではフロントフレームバッファとバックフレームバッファ（本来はフロントバッファとバックバッファだが分かりやすさ優先でフレームを付け加えた）

の二枚を用意するので、**2と入力する**。あるいは2の変数を格納するconst変数(例.**const int HOGE =2。**HOGEはもっとわかりやすい名前を付けてよい)を入力してマジックナンバーにしないようにしてもいいだろう。

Height:

ここには絵を出力する対象のウインドウ、あるいはモニター全体の解像度を入力する。

ここではウインドウに絵を出力するので、そのウインドウの高さを入力する。

Width:

横の解像度。幅である。

Heightと似た感じでウインドウの幅を入力する。

BufferUsage:

スワップチェーンに使うバッファの仕様用途を指定する。

ここではフレームバッファ（しつこいようだが：GPUのVRAMに格納されたピクセル情報のメモリ領域を確保する緩衝材）を使う。絵をRendertargetに出力するのでRENDER\_TARGET\_OUTPUTにする。

※なぜDXGI\_USAGE\_BACK\_BUFFERにはしないのかというと、マイクロソフトのページ（

<https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/dxgi-usage>

では「スワップチェーンを作成する際にDXGI\_USAGE\_BACK\_BUFFERと入力する必要はない」と記述されているので、今のところは**（todo:コードを実行して支障があるなら変更する）**

入力しない。

Count

DXGI\_SAMPLE\_DESCのメンバ変数。このディスクリプタはマルチサンプリング(**絵のジャギーを減らす技術:もっとよく調べる**)に関する

処理の指示書みたいなものである。

Countには、サンプルの数を入力する。値が大きければ大きいほど取れるサンプルのピクセルの数が多くなる。

ここでは一応マルチサンプルを使わないので0を入力するが、後々任意に調整できるように書き換える可能性もある。

Quality

サンプリングの品質。値が大きければ品質が上がり処理負担が増加してパフォーマンスが低下する。

ここでは最低品質である1を入力。

Format

用途が多様だが、ここでは「モニターに出力する色ベクトルの情報」を入力すると考えてよい。

色については後述するが、レンダーターゲットに出力する絵の色の多さは重要なため、精度の高いRGBA(RGBは赤緑青の三原色、Aは透明度のα)の情報を持たせたい。そのため32bit(4,294,967,296色)の精度を持つDXGI\_FORMAT\_R8G8B8A8\_UNORM

を入力する（2021/06/07)）

flags

詳細はドキュメントを参照してほしい。ここではレンダーターゲット(絵を出す先の事。例えばウインドウの場合や、モニターそのものだったり、ヘッドマウントディスプレイ(HMD)にも絵を出せる）のサイズをIDXGI::ResizeTargetで変更した時に、絵のサイズを変更するDXGI\_SWAP\_CHAIN\_FLAG\_ALLOW\_MODE\_SWITCHを入力する

[**https://docs.microsoft.com/en-us/windows/win32/api/dxgi/ne-dxgi-dxgi\_swap\_chain\_flag**](https://docs.microsoft.com/en-us/windows/win32/api/dxgi/ne-dxgi-dxgi_swap_chain_flag)

Scaling

<https://docs.microsoft.com/en-us/windows/win32/api/dxgi1_2/ne-dxgi1_2-dxgi_scaling>

resize（描画先（レンダーターゲット）のサイズが変更)された際、スワップチェーンはどのような動きを取るか

の指示の出し方。

ここでは、バックフレームバッファ、すなわち次フレームでモニターに出力する

絵のサイズをレンダーターゲットのサイズに合わせるように指示を出すDXGI\_SCALING\_STRETCHを入力しておく

※DXGI\_SCALING\_ASPECT\_RATIO\_STRETCHは、おおむねDXGI\_SCALING\_STRETCHと同じような指示だが、

描画範囲がレンダーターゲットのアスペクト比（ここも調べて説明)とフィットしない場合は上下か左右に黒帯を出すようになる。

2.DXGIFactoryを作るかどこからか持ってきて、{

スワップチェーンを作る

例.

auto hr = in\_factory->CreateSwapChainForHwnd(in\_desc.p\_queue, in\_desc.window.GetHWND(), &swapChainDesc, nullptr, nullptr, p\_swapChain.ReleaseAndGetAddressOf());

ここも必ず読む事。スワップチェーンの生成は意外と手間がかかるうえ、引数が多いので

クラッシュしやすい。引数の開設には必ず目を通しておこう。例えば、**HWNDにはNULLを入れてはならない**

<https://docs.microsoft.com/en-us/windows/win32/api/dxgi1_2/nf-dxgi1_2-idxgifactory2-createswapchainforhwnd>

}

２．RenderTargetviewを作る。

スワップチェーンだけでは描画処理の一連の動作を行うには不十分だ。

GPUにバックバッファとRendertargetをどのように扱うかの指示を飛ばしていないからである。

そこまでやらんといけないのかという声も挙げたくなりそうだが、DirectX12はプログラマーがチューニングしなければいけない箇所を増やして、アプリに考えさせる余地を減らす事で負担を減らすことで処理の高速化に繋がっているのだ。（アセンブリのデバッグ見てみたら挙動がわかるかもしれない）

そういうわけなので、GPUにRendertargetをどのように扱うかを記す指示書、RendertargetViewを書く。

1.DescriptorHeapの作成。

DescriptorHeapとは{

DescriptorHeapはDescriptor（GPUに処理させるデータ(くどいようだが書いておく)際、

どのように処理するかを記述したマニュアル)を格納する配列のようなメモリ領域である。

配列のようなというわけなので、このヒープには複数のDescriptorを格納することができる

その目的

{

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/descriptor-heaps-overview>

}

}

文章の終わり

・バージョン管理にgitを使おう。