



まずうちさあ…

方針を説明したいんだけど…聞いていかない? ああ〜いいっすねえ〜。

という事でこの授業の方針ですが、レンダリング系の技術を身に付けてもらうためのものでゲームを作るための授業ではございません。そこはよくご理解いただきますようよろしくお願いいたします。

基盤技術に興味がなく、3DCG レンダリングにも興味ないという人は、ほかのコースを受講されたほうがよいのかと思います。ただ、このコースを修了すれば必然的に様々な問題に対処できるようになっていると思いますので、特に 3D のゲームを作る底力が付くと思います (2D でも役に立つとは思いますが)

そこまで承知の上でこの授業を受講したい人はそのまま受けていただけるとありがたいと思います。

最近の流れ

CEDEC とかに参加して見えてきた、ちょっと最近の流れをご説明したいと思います。昨年の頭に DirectXRaytracing が正式にリリースされ、nVidia が RTX シリーズを発表しています。で、お分かりになる方がいればいいんですが、RTX はレイトレーシングを強化したグラボです。

それ以外ではなく『テンソルコア』というものが搭載されており、DirectML も見据えた作りになっているのではないかと思います。

GTX と RTX の違い

RTX の特徴としては

- ①レイトレーシング(DXR)に特化した GPU
- ②機械学習(DirectML)に特化した GPU

レイトレーシングとは？

英語で Ray Tracing と言って Ray(光線)目からビームを出すこのビームが物体に当たった時、その当たった座標の色を検出して、ピクセルを塗りつぶす。これを全画素について行うことで 3D の絵を表示する。

一般のゲームで用いられている 3D 表示のアルゴリズムはスキャンライン法といって『三角形塗りつぶしアルゴリズム』で塗りつぶして 3D 表示してました。三角形は最終的に 2D なんですけど、3D→2D の変換行列があって、すべて 2D の三角形に変換され、それをただ塗りつぶしてただけ。

レイトレーシング(くそ遅い)→スキャンライン(速い)。RTX(レイトレーシングを速くするためのハードウェア)、ついでに言うと、機械学習も速くなります(テンソルコア)。で、皆さんがいるこの教室の PC は今のところ RTX2070 ですので、ご研究されたい方は研究してください。

ちなみに CEDIL のサイトにいろいろとありますので、授業が余裕な人は研究をしておきましょう。

<https://cedil.cesa.or.jp/>

授業の流れ

授業の流れは大雑把に言うとこんな感じです。

基本的な目的は、DirectX12 を用いて 3D モデル(PMD…MikuMikuDance のモデル)を表示する。ということです。

- ①モデルの表示(ここが…しんどい、ほんとうにしんどい)
- ②セルシェード(たつのしー!!)
- ③ボーンでアニメーション(たつのしー!!)
- ④シャドウマップ(すごーい!!!)
- ⑤ポストエフェクト(わかんないや!)

もうちょっと細かく言うと

1. DirectX12 ポリゴンを出すまでがんばる(面倒だしシェータが必要だし即死)
2. ポリゴンに 3D 変換行列をかけて 3D 化する(行列が分かってれば割と大丈夫)
3. テクスチャ貼る(テクスチャは思ったより面倒なんやで?)
4. PMD モデルを読み込んで表示する(まずは頂点情報のみ)
5. 面を貼る(インデックス情報が必要)
6. シェーディングする(数学がクソ出てくる。内積とか内積とか内積とか)
7. 深度バッファを有効にする(めんどろ)
8. ボーン情報を読み込む
9. ボーンを回転させてみる
10. ボーンに合わせてスキニング(頂点ウェイトで頂点移動)する
11. テクスチャロータを作る
12. ポージングさせる
13. アニメーションさせる(リバーサイテレータ登場!!!)
14. ベジエで動かす(ニュートン法、二分法)
15. つぶれ影表示(行列で潰して黒く塗るだけ)
16. シャドウマップでセルフシャドウ(シャドウアクネがさ...)
17. 簡易トゥーンレンダリング
18. 輪郭線
19. アンチエイリアシング(輪郭線との相性最悪)
20. IK(いけるかな...)
21. ポストエフェクト(をするために必要な事)
22. 色調整(ポストエフェクト)
23. 画面を割る(法線テクスチャ+ポストエフェクト)
24. ガウスぼかし
25. ブルーム(縮小バッファ+ポストエフェクト(ガウス))
26. 被写界深度(深度値+縮小バッファ+ポストエフェクト(ガウス))
27. imgui 組み込み
28. ディファードレンダリング
29. インスタンスングで大量表示
30. SSAO(スーパーサードアートオンラインではない...冬休み中にはできるかな)
31. SSR(ガチャのことではない...いれいれ加減にしる!!)
32. インバースキネマティクス
33. 太陽光産卵(散乱)
34. 法線マップ(接ベクトルと従法線ベクトルが必要なんだよなあ...)
35. コンピュートシェータ(いけるのか?)

36. DirectXRaytracing

こんな感じで行けたらいいかな。どうかな？あくまでも願望です。せつかく RTX 部屋にいるのだから DXR やりたいとは思ってます。

目次

まずうちさあ…	1
最近の流れ	1
GTX と RTX の違い	2
授業の流れ	2
環境構築	6
ウィンドウ表示	9
HINSTANCE とか HWND とか	9
じゃあ実装	10
解説	14
基礎知識説明①	16
シェータ	16
頂点シェータ	17
ピクセルシェータ	18
ジオメトリシェータ	18
ハルシェータ(テセレーション)ドメインシェータ	19
コンピュートシェータ(GPGPU)	20
レンダリングパイプラインについて	22
ちょっとしたハードウェアの知識	24
GPU と CPU の違いについて	24
キャッシュメモリについて	28
DirectX 組み込みに入る前に	31
DirectX12 がそれ以前の DX と違うのはどこ？ここ？	32
仮想メモリ(仮想アドレス)とは	34
キャッシュメモリとか分岐予測とか	36
とにかく DirectX12 を動かそう(初期化編)	38
準備①(インクルードとリンク)	39
基本的な部分の初期化	39
画面に影響を与える準備	43
スワップチェーン	44

レンダーターゲットの作成.....	51
さて、いよいよ画面のクリアだ.....	56
コマンドを投げるために.....	56
コマンドリストとコマンドアロケータをリセット.....	57
コマンド:レンダーターゲットを設定.....	59
コマンド:レンダーターゲットをクリア.....	60
コマンド:クローズ.....	60
コマンドキューに投げる.....	60
スワップチェーン Present.....	61
実は色々間違ってるんです.....	61
フェンス.....	62
ではフェンスを実装しようか.....	67

まあとはいえずは作る環境を整えなくちゃね。

環境構築

今この教室の OS の設定はこんな感じになってます。

Windows の仕様	
エディション	Windows 10 Education
バージョン	1809
インストール日	2019/03/28
OS ビルド	17763.316

Windows の設定→システム→バージョン情報を見るとこんな感じでバージョンが書かれています。この数値は確認しておきましょう。重要です。

現在の最新は確か 1903 だったので、最新の機能を使いたい場合はバージョンアップする必要があります。まあその辺は開発に慣れてからにしましょう。

DirectX12は基本的にはWindowsSDKというSDK(Software Development Kit)の中に入っています。WindowsSDK は一応 VisualStudio をインストールする際に選択的にインストールされているので、恐らく皆さんの環境に最低限のものは入っているとは思いますが。

なので、特になんもしなくても開発そのものはできると思いますが、この WindowsSDK はできるだけ Windows のバージョンと合わせておいた方が良いでしょう。現在の WindowsSDK はバージョン 10.0.18362.0 ですが、これは Windows バージョンが 1903 でないとともに動作しないので、バージョンの管理はしっかりしておきましょう。

あと、後述する dx12.h や DirectXTex などの対応バージョンが食い違くと動作しないので気を付けましょう。ちょっと色々バージョン関連がややこしいので、おうちの PC で環境構築する時には気を付けましょう。ちなみに最新バージョンの SDK は

<https://developer.microsoft.com/ja-jp/windows/downloads/windows-10-sdk>

から入手できますが、インストールにクツソ時間がかかるので、時間があるときにやっておき

ましょう。あと確か Windows 最新バージョンは特定のグラボでブルスクレベルの不具合起こすらしいので家の PC のバージョン上げるときは自己責任でお願いします。1803 以上になってたら大丈夫です。

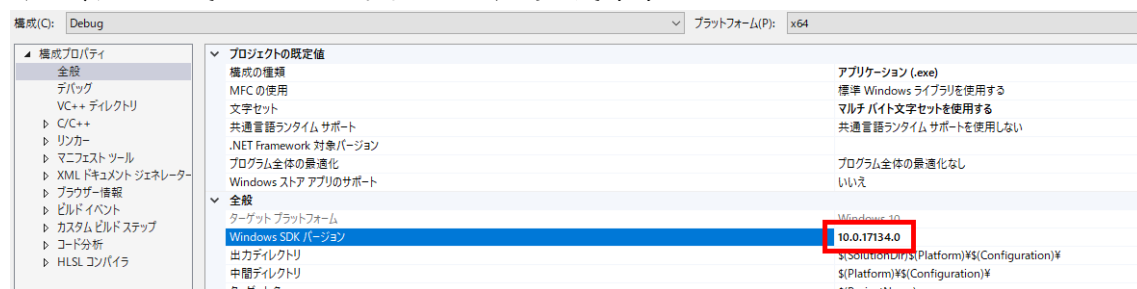
過去の SDK のバージョンへのリンクも一応貼っておきますので、家の PC が不安な人は 1803 に合わせておいた方が無難でしょう。

<https://developer.microsoft.com/ja-jp/windows/downloads/sdk-archive>

まあ、最初の方は Window の構築だけなので、しばらくはそこ関係しないので、ぼちぼち環境構築してってください。

あ、ちなみに VisualStudio2015 はもう DirectX12 の SDK に対応してないので、家の PC が 2015 の人は直ちに 2017 以降にしてください。なお VS2019 での動作確認はしてますので、VS2017 か VS2019 なら問題ないと思います。

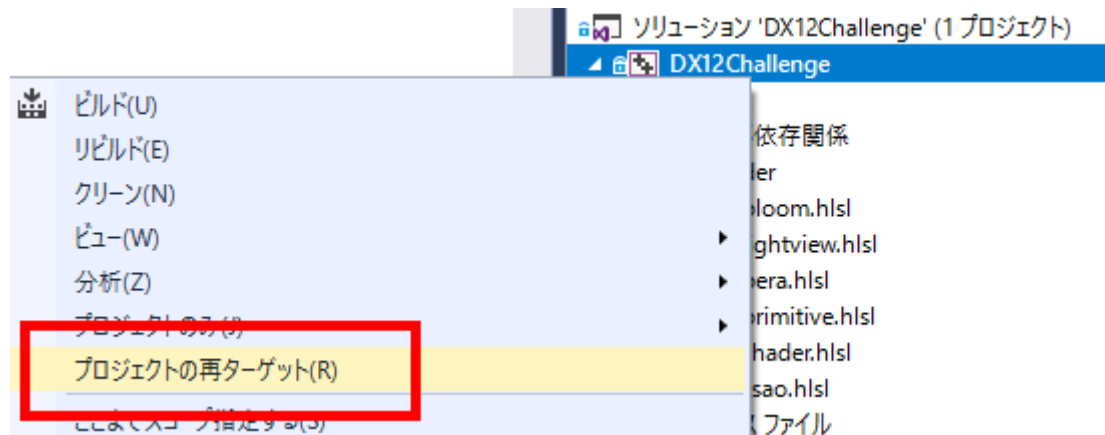
ちなみに VS 上での SDK バージョンの確認方法ですが



プロジェクトプロパティの↑を見れば書いてます。ちなみに授業で DirectX12 を導入したのは VS2015 の時代で、あの頃は VS 自体が対応してなくてトラブルばかりでした。あの時代に導入したのは見切り発車だったかな〜って思いますが、今は開発環境がスタンバイ OK なので、2 年前の先輩に比べれば恵まれてますよ本当に…。

ちなみに、この SDK バージョンが違う事でちょっと面倒なことが発生します。もし家のバージョンと学校のバージョンが食い違っていた場合、単純なコンパイルすらさせてくれません。

その場合は対象プロジェクトを右クリックして『再ターゲット』をしてください。



一応それで大丈夫なはずですが。

もし家のが上手くいかないという人はご相談ください。ノートPCの人はもってきて見せてもらうとこっちも助かります。

ウィンドウ表示

何はともあれウィンドウの表示を行いましょう。表示自体は DxLib なら `DxLib_Init()` で終わら
だっただけですけどね。WindowsAPI の場合、そうはいきません。とりあえずいつものように
`main.cpp` を作って、`main` 関数もしくは `WinMain` 関数を作っておいてください。

次に Application クラス作りましょう。シングルトンで作るときでしょうか。
で、DxLib の時と似たような感じで作っていきます。とりあえず

`Initialize()`

`Run()`

`Terminate()`

のそれぞれの関数を作っておきます。`main` 側からはこの3つを呼ぶだけにしておきたいです。
もちろん `Run` の中にメインループが入っているイメージです。

で、ウィンドウ作るときにやたらと『ハンドル』ってのが出てきます。

HINSTANCE とか HWND とか

`HINSTANCE` や `HWND` の頭にある `H` というのは `Handle` の略です。いろいろと例え方がありと思
いますが、それを操作するためのモノということで『ハンドル』って名前がついてるのです。ハ
ンドルっていうと



こういうものを想像すると思いますが、とくに `HWND` に関してはウィンドウ(車)を操作するた
めに必要な『鍵』と思ってもらったほうが良いと思います。



ウィンドウ(車)



必要!!



ハンドル(鍵)

一応 Windows とか DirectX 界限では当然のように `Handled-Body` パターン的なのが使用されて
いて、実際 DxLib におけるリソースのほとんどの戻り値もこれですね。あれは `int` で使いやす

いけどね。

ただ、Windows プログラミングにおいてこいつの型は単なる整数型(というかアドレス型)のくせに windows.h で typedef だかなんだかやってるせいで windows.h(windef.h)をインクルードしなければ使えないんですが、その値を Application クラス内で保持するためにはヘッダ側へのインクルードとなって、ちょっとイヤ。

こういった時に選択肢は3つくらいある。1 つではないと思ってください。プログラミングに一つの答えなんて存在しないのです。

これはプログラムするうえで身に着けておいてほしい考え方ですが、最初の解決策に飛びつかないでください。

必ずいくつか選択肢を見つけて、その中から明確な根拠で選んでください。場合によっては『一番シンプルで簡単そうだから』でもいいです。ただし、必ず選択肢をいくつか用意してください。

で選択肢ですが

1. 割り切ってヘッダでインクルードする
2. ハンドルをヘッダ側で使用せず cpp 側のグローバルな領域(cpp スコープ)で宣言、初期化、使用する
3. Window などのデコレートクラスもしくは DxLib のように別テーブルで int 管理する

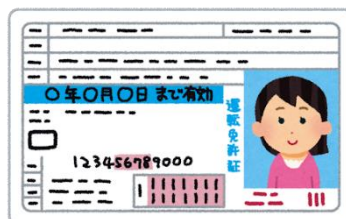
正直ここは後々の拡張性まで考えて、潤沢な時間さえあれば3番を用いたいところだけど、ここは2番くらいが時間的な意味でも妥当かなと思う。1番はやっぱ生理的にイヤ。

じゃあ実装

とりあえずウィンドウズのウィンドウを作るのに、DxLib の時は DxLib_Init で済んでただけど(フォントはそれだけじゃなくてデバイスとかその他初期化してくれてる)、ウィンドウを『作る』だけでまずは、Windows に自分の身分証明をする必要があります。

```
WNDCLASSEX w = {};  
w.cbSize = sizeof(WNDCLASSEX); //これ、何のために設定するのさ...?  
w.lpfnWndProc = (WNDPROC)WindowProcedure; //コールバック関数の指定  
w.lpszClassName = _T("DirectXTest"); //アプリケーションクラス名(適当でもいいです)  
w.hInstance = GetModuleHandle(0); //ハンドルの取得  
RegisterClassEx(&w); //アプリケーションクラス(こういうの作るからよろしくって OS に予告する)
```

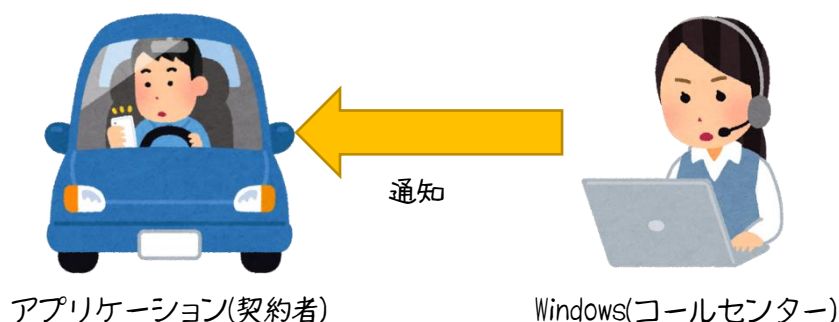
はい、このWNDCLASSEX という構造体は身分証明書みたいなものです。
レンタカー屋さん(Windows)に車を借りるためにまず身分証明書を提出しなければなりません。
犯罪に使われたら困りますからね…



身分証明書(WNDCLASS)

ちなみに `lpfnWndProc` というのは電話番号みたいなもので、何かがあったら問い合わせされるものです。Windowsではこれを関数ポインタを使って登録しており、コールバック関数といえます。

そのウィンドウに対して何か変化が要求されれば Windows 側からなにか通知が行われます。登録しておかなければそれに対応できないので、登録します。



で、いいよウィンドウを作っていくわけですが、大きさとか種類を指定しなければなりません。
レンタカー屋で借りるときも車種とか大きさとか伝えないとだめなのと同じですよ。

で、この大きさ指定がちょっとややこしくて、タイトルバーがある場合はタイトルバー込みの幅と高さを指定しないとタイトルバーやウィンドウ枠のぶん、大きさが変わってしまいますので、`AdjustWindowRect` 関数で補正します。

```
RECT wrc = { 0,0, WINDOW_WIDTH, WINDOW_HEIGHT };//ウィンドウサイズを決める
AdjustWindowRect(&wrc, WS_OVERLAPPEDWINDOW, false);//ウィンドウのサイズはちょっと面倒なので関数を使って補正する
```

```
HWND hwnd = CreateWindow(w.lpszClassName, //クラス名指定
    _T("DX12 テスト"), //タイトルバーの文字
    WS_OVERLAPPEDWINDOW, //タイトルバーと境界線があるウィンドウです
```

```

CW_USEDEFAULT, // 表示 X 座標は OS にお任せします
CW_USEDEFAULT, // 表示 Y 座標は OS にお任せします
wrc.right - wrc.left, // ウィンドウ幅
wrc.bottom - wrc.top, // ウィンドウ高
nullptr, // 親ウィンドウハンドル
nullptr, // メニューハンドル
w.hInstance, // 呼び出しアプリケーションハンドル
nullptr); // 追加パラメータ

```

このくらいのコードが必要になる。

で、ウィンドウ出るかい？まあ、出ないんだな、これが。『ウィンドウハンドル』というウィンドウの素を作っただけなんですよね。あくまでも車のキーをもらった状態でまだ運転していない。

ここでしくじることは 99.9% くらいないと思うけど、あ、最初に `#include <windows.h>` しといてね。

もし失敗した時にキャッチできるよう

```

if (hwnd == nullptr) {
    LPVOID messageBuffer = nullptr;
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        nullptr,
        GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPWSTR)&messageBuffer,
        0,
        nullptr);
    OutputDebugString((TCHAR*)messageBuffer);
    cout << (TCHAR*)messageBuffer << endl;
    LocalFree(messageBuffer);
}

```

のコードも追加しておいた方がいれね。まだウィンドウは出ないよ。

ただ、ここまでがウィンドウの初期化处理なので、これを `InitWindow` 的な関数を作って、その中に入れておいてください。

で、一応ウィンドウ出すのなんてハンドルがあればあとは ShowWindow 関数で終わるんだけど
ShowWindow(hwnd, SW_SHOW); //ウィンドウ表示

これはちょっと InitWindow に入れるのはやめておこう。どっちかというと Run に入りたい。

次に DxDlib の時にもあったと思うけどメインループだ。これは Run の中に書いてほしい。一応やり方としては無限ループがまして、ウィンドウ破棄のタイミングでループを抜けるイメージで。

```
if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) { //OS からのメッセージを msg に格納
    TranslateMessage(&msg); //仮想キー関連の変換
    DispatchMessage(&msg); //処理されなかったメッセージを OS に投げ返す
}
```

```
if (msg.message == WM_QUIT) { //もうアプリケーションが終わるって時に WM_QUIT になる
    break;
}
```

こんな感じでループ抜けを書いておく。

で、Terminate()あたりに

```
UnregisterClass(w.lpszClassName, w.hInstance); //もう使わんから登録解除してや
```

と書けば、一応ウィンドウ表示まで完成です。ひとまずお疲れ様。と言いたいところやけど、ひとつ忘れとったわ…いっつも忘れる。ウィンドウプロシージャを忘れてた。こいつは『コールバック関数』と言って、OS から呼ばれる関数を定義しとかなあかんのですよ。ということで定義
//めんどくせーし、あまりゲームに関係ないけどけど書かなあかんやつ

```
LRESULT WindowProcedure(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam) {
    if (msg == WM_DESTROY) { //ウィンドウが破棄されたら呼ばれます
        PostQuitMessage(0); //OS に対して『もうこのアプリは終わるんや』と伝える
        return 0;
    }
    return DefWindowProc(hwnd, msg, wparam, lparam); //規定の処理を行う
}
```

こいつはクラス内関数じゃなくて、通常の変数として宣言してください。結果的には main.cpp が

```
#include "Application.h"

int main() { // ①…コマンドラインありの時
//int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int){
    auto& app = Application::Instance();
    app.Initialize();
    app.Run();
    app.Terminate();
    return 0;
}
```

このようになるようにしておいてください。

解説

ちなみに軽く解説しておく…これ、面倒なんで昨年の授業のテキストから一部コピーしてくると

アプリケーションのハンドル

何なんでしょう…これはマイクロソフト系のプログラムでありがちなものなのですが、Handle-Body イディオムとも呼ばれるんですが意味合い的には DxLib におけるグラフィクスハンドルみたいなもんです。あれはロードした絵を操作するためのものでしたが、今回はアプリケーションを操作するための『ハンドル』だと思ってください。持ってくる方法は至って簡単

ウィンドウアプリケーションなら

```
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR, int cmdShow){
    ~中略~
}
```

この hInst がアプリケーションのハンドルにあたります。このハンドルはウィンドウを表示するために必要なものになります。

軽く理由を説明しておく…

ウィンドウを表示するのは『アプリケーション自身』に思えますが、実際は『OS(Windows)』です。ちょっと難しい概念なんですけどね。ディスプレイやマウスやキーボードやスピーカーなどのデバイス回りを制御するのは OS なんですよ。モバイル機器でも同様なんですけど、OS ってアホほど色々やってるんですわ。

で、そのデバイスの一つであるディスプレイに『ウィンドウ』を表示するのは OS の役割であり、

OSにその仕事をさせるためには『持ち主は誰か』をOSに教えておく必要があるのです。

…何となくわかりますかね？君のプログラムが直接ウィンドウ出してるわけじゃないんです。だからこのハンドルをOSに教えることによってウィンドウを表示したりするわけです。

ちなみにDirectXってのはこのOSがやっている仕事をDirectXが一部『ぶんどって』ドライバに対して直接命令を出し、より高速に描画処理をするためのものです。

なお、コンソールアプリケーションでも今実行中のプログラムのハンドルを得ることができまして、

GetModuleHandle という関数で取得できます。

```
HINSTANCE hInst=GetModuleHandle(nullptr);
```

あと、この授業を受けるときには徹底してほしいことが一つあって、それは

知らない関数が出てきたら、MSDNの関数を必ず確認しよう

です。OS周りやDirectX周りの関数は結構罠が多くて、きちんと読まないで予想外の仕様にハマる事になります。

<https://msdn.microsoft.com/ja-jp/library/cc429129.aspx>

ちなみに↑のリンクはGetModuleHandleのMSDNリファレンスです。『必ず』読むクセをつけましょう。マニュアル読め！ハードやライブラリの仕様読め！！はプロになってからももちろん徹底してください。読まずにドツボにハマる奴が多すぎる(プロでも)

ちょっとここでいい機会なので、僕の授業を受けるときの鉄則を書いておきます。

鉄の掟

- マニュアルは必ず読む(MSDNなどの信頼できる物を必ず隅から隅まで読んでください)
- 分からなかったらすぐに聞く(先生でも友人でもいいので、分からないままにしない事)
- 休まないように(基本的に、休むとワケ分からない事になります。そういうやつを僕はフォローするつもりは一切ないです。機能が実装できてなければ落第ですので気を付けてください)
- 寝ないように(出席しても寝てたら同じです。いや俺に面白みがなくて眠いのはわかるけど、それは改善しようと思ってるけど、眠ること自体は君の問題です。寝りゃあその分君は学費を無駄にしてるんです。家で十分な睡眠を取って、授業を聞かない時間を極力つくらないようにしてください。寝ててついていけない奴をフォローしません)
- 授業中のトイレも同様です。トイレに行っても基本授業は止まりません。授業中にブリュリュやられても困りますが、そこは自分で判断して可能な限り我慢してください(休み時間に

出すだけ出し切ってください)

- 放課後に少なくとも1時間は制作の時間を割り当ててください(それくらいじゃないとゲームコンテストにも就職活動にも間に合いません。世の中そんなに甘くはないです。)
- 学外の制作会(福大のハ耐など)や勉強会(Unity 勉強会とか UE4 勉強会など)に一度は参加しましょう。学校の狭い範囲内の価値観ばかり見ていると作るものがショボくなりがちです。逆に他校のを見ると自信がつくかもしれません。
- ↑と同じ意味で他校の発表会もチェックしておきましょう。TGS に行く人は企業ブースばかりでなく他校のブースをスパイしましょう。

さて解説に戻るがこのアプリケーションのハンドルを用いて OS にウィンドウを表示してもらうのだけど、これもまた結構面倒なのだ。

手順が

- 1.ウィンドウクラスの作成→登録(RegisterClass)
- 2.ウィンドウサイズの設定
- 3.ウィンドウオブジェクトそのものを生成(CreateWindow)
- 4.ウィンドウを表示>ShowWindow)
- 5.ループ

となります。このウィンドウクラスを作る際にアプリケーションハンドルが必要になります。また、ウィンドウクラスを作る際にはウィンドウプロシージャなるものも作る必要があります、結構面倒なのです。

次回以降自分でウィンドウ出す際にはこの手順を思い出してください。

で、これから DirectX12 だー!みんないくぞー!と言いたいところですが、ちょっと事前の知識がそれなりに必要なので、知っておきましょう。CG 検定の知識も同様ですが...

基礎知識説明①

シェータ

シェータ、シェータと言うとりますけれども、『誰やのあんた!?』って思ってる人も多いと思います。こいつは言うたら、表示に関わる言語で C/C++ と違うものです。GPU 上で動作する言語でございます。HLSL(High Level Shader Language)と言って、C 言語っぽい見た目はしておりますが、別物でございますので、ご注意ください。

シェータの種類は現在の所

- VS:頂点シェーダ(バーテックスシェーダ)
 - PS:ピクセルシェーダ(フラグメントシェーダ)
 - GS:ジオメトリシェーダ
 - HS:ハールシェーダ(デセレーション)DS:ドメインシェーダ
 - CS:コンピュートシェーダ
- などの種類があります。

最初に使われるのが恐らく頂点シェーダとピクセルシェーダでございますね。DX11 以降においては少なくとも VS と PS の2つを定義しないとそもそもポリゴンを 1 枚表示することもできません。

という事で、みなさん、この DX12 の授業ではシェーダは避けて通れないんです。フヒヒ(DX11 の頃からシェーダは必須でしたけどね)

ちなみにこの中に仲間外れがいます。CS:コンピュートシェーダです。そもそもシェーダというのは名前から想像できると思いますが、本来は陰影をつけるための計算をするものでした。

ところが、GPU 自体が並列処理に優れているという理由でシェーディングや幾何学と関係のない部分で使用されました。これを GPGPU と言い、それを行うためのシェーダをコンピュートシェーダと言います。ですから、この後に説明する『レンダリングパイプライン』の環から外れた存在なのです。

レンダリングパイプラインについてはのちほど解説します。

頂点シェーダ

その名の通り頂点をいじくりまわすシェーダです。3D オブジェクトが無数の頂点でできているのは知っていると思いますが、頂点情報が GPU に送られ、描画コマンドが走ると真っ先に実行されるシェーダです。

当たり前ですが、頂点情報は頂点の集合体にすぎません。ですから移動とかしませんし、カメラ変換とかもしませんし、そのままだと 3D なので 2D に変換してやる必要もあります。

それをやるのが頂点シェーダです。1つ1つの頂点につき1度実行されますので、1万頂点のモデルなら1万回実行されます。ただし、頂点情報は GPU 側にあり、シェーダも GPU 側で実行されるため超高速です。1万回でも一瞬です。

初歩的な主な仕事は座標変換行列データを CPU 側から渡してやって、その行列を頂点情報に乗算し最終的な座標に変換するのがお仕事です。

ピクセルシェータ

ピクセルシェータはその名の通りピクセルを塗りつぶすときに発行されるシェータです。頂点シェータで変換された頂点情報を『ラスタライザー』がラスタライズして(ピクセル情報に変換して)、その塗りつぶすべき 1ピクセル 1ピクセルに対してピクセルシェータが呼ばれます。

つまり、長方形ポリをウィンドウいっぱい 1 枚描画したとするとその解像度分のピクセルシェータが実行されます。例えば 1280x720 のウィンドウであれば 921,600 回実行されるわけです。怖いですね〜。ですから昔はピクセルシェータで複雑な計算はご法度で、DX9 の頃は演算回数制限があったほどです(超えてるとシェータコンパイル時にエラーが出ます)。

参考までに PixelShader1.0 の演算回数は 8 で、PixelShader2.0a の制限は 1024 です。一気に増えましたねえ…。

ちなみにシェーディングとかもピクセルシェータで行いますが、昔は処理量を減らすために頂点側でシェーディングして、あとはラスタライズ時の補間に任せるという安っぽいテクもありました。

ピクセルシェータの基本的な役割は

最終的に塗りつぶす色を決定する

です。このためにテクスチャの参照とかシェーディング計算とかやることになります。

ジオメトリシェータ

さて、ジオメトリシェータですが、こいつ、何なんすかねえ？

頂点とピクセルは分かった。ではジオメトリシェータとは何なの？ジオメトリとは幾何学という意味だ。

言うてしまうと、ジオメトリシェータによって新たな頂点を作ることができたりします。これにより全頂点からライトベクトル方向に引き延ばした頂点を作ることによって『ボリュームシャドウ』などを作ることできます。

ただ、ボリュームシャドウは最近あんまり聞かないので、たぶん実用的じゃないんじゃないか

なあと思ったりします。

ちなみに受け取るデータは『頂点』ではなく『プリミティブ』です。頂点一つ一つではなく三角形ひとつひとつです。ですからある意味『ポリゴンごと』の処理ができる唯一のシェータだったりします。

なのでポリゴン単位に色々とおもしろい事ができるはずっちゃあできるはずなんだけどねえ…。

ちなみにこういう事も出来るっぽいです。

<https://wlog.flatlib.jp/item/1070>

ああ～楽しそうなんじゃよ～。

とりあえずそういうのがあって、なんか活用できそうなアイデアがあったら使いたいと思います。まあ、業界の話をするとき、最近ちょっとジオメトリシェータ不人気じゃない?って思います。廃れていく可能性を感じます。

ジオメトリシェータがいなくなっちゃうヤバいやババ…。。

ハルシェータ(テセレーション)ドメインシェータ

次にハルシェータです。ハルシェータの話をするまえに『テセレーション』とは何かをお話しいたします。

テセレーションと言うのは、おおざっぱに言うとポリゴンを元の状態からさらに細かく分割する仕組みの事です。いわゆるサブディブ的な奴ですね。OpenSubdiv だので活用されていたり、また、ハイトマップ(高さマップ)と組み合わせることにより、ノーマルマップやパララックスマップみたいに『見せかけの』凸凹にするまでもなく、実際に凸凹を出現させることができるようになります。

正直、使ったことがないので良く分からないんですが、テセレーションの前にハルシェータを実行し、テセレーションの後にドメインシェータが実行されるようです。

どうもハルシェータが分割パッチのコントロールポイントを定義したり、分割の際のパラメータを定義するところのようです。実際の分割はテセレーションステージで行われますので…。

で、テセレーターが分割して、それがドメインシェータに渡されるようです。この分割後にで

きた新しい頂点に対して、頂点シェータと同じような事をする部分のようです。

…まあ時間があつたらデモ的なものを作ろうかなと思っていたりします。

最後に仲間外れのコンピュートシェータですが、これも使ったことはありません

コンピュートシェータ(GPGPU)

これは何かというと、描画に基本関係しないシェータです。シェータなのに描画に関係しないとはこれ如何に…？

ちなみにグラフィックスレンダリングパイプラインのどこにも ComputeShader はありません。(コンピューとパイプラインというのは存在するけど)

先にも言いましたが、レンダリングパイプラインの流れの外にあります。ではなぜシェータなのかと言うと、とにかく GPU 上で動くプログラムを慣例的に『シェータ』と言ってるからに過ぎません。

つまり ComputeShader というのは本来は CPU でやるべき数値計算を GPU 側でやってると思ってくれば良いです。GPU は速いというのがゲームや CG 業界以外にも知れ渡ってしまって、数値計算やディープラーニングや、仮想通貨マイニングに使われるようになってるわけです。

もちろんゲームでも物理計算だの衝突判定だので使用するので、ゲーム的にもこの GPGPU(汎用 GPU コンピューティング)は役に立っています。

使ったことないのであまり言うところが出そうなのですが、分かる部分でちょっと注意をしておきます。

『そんなに早いんなら全部 GPU に処理を渡せばええんちゃうのん？』

と思うかもしれませんが、ちょっと違うんですよ。CPU1 コアと GPU1 コアだと明らかに CPU の方が計算速度も速いし、複雑な演算も処理できます。1つ1つの性能は CPU の方が高いのです。

じゃあなぜ GPU が速い速いと言われているのかと言うと、画像の描画に特化して進化してきたため、演算自体にそれほど複雑な計算が必要ないコアを『大量に』並べることで高速化を図ってきたのです。

それなりのスペックの PC だと CPU がだいたい 8 コアくらいなのに対し、GPU は数千個…多分今のスーパーな GPU なら万言ってるんじゃないかと思います。調べてないから知らんけど。

まあ言うたら、数学の先生 1 人に対し、四則演算くらいしかできない中学生が 1000 人いて、中学生がそれぞれ手分けして作業するのと先生 1 人で作業するのと比べるようなもんやね。

やからあんまり複雑な命令を出すといくら 1000 人の中学生でも無理なものは無理だし、その代わり大量の単純計算なら圧倒的に 1000 人中学生の方が速い。

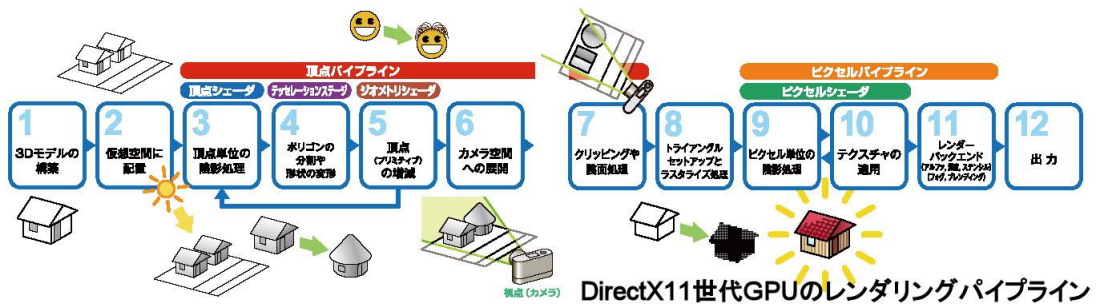
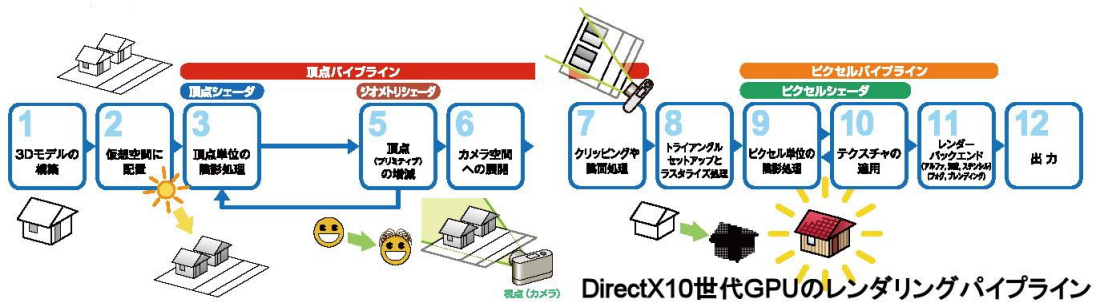
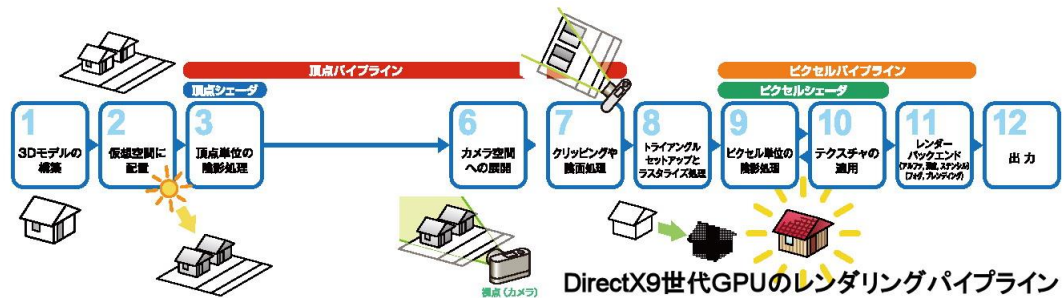
CPU と GPU はそういう違いがあると思ってください。だから、GPU は大量の頂点とか、大量のピクセルとかを処理するのが得意なわけやね。

ちなみにそういう理由から、GPU は全員で働く状況をお膳立てしてやれば最高のパフォーマンスを引き出せるって事です。

で、おぜん立てってのは並列化を阻害しないって事…並列化を阻害する要因はいくつかあるんですが、よく言われるのが『分岐』ですね。その他いろいろあるんですが、勉強不足でこれ以上は今は言えません。すんません。先に進みましょう。

レンダリングパイプラインについて

レンダリングパイプラインというのは3Dデータの入力からどのようにデータがやり取りされ、最終的な画面出力になっていくのかの流れを示したものです。以下にレンダリングパイプラインの移り変わりの図をパクってコピペします。



西川善司の3DゲームファンのためのE3最新ハードウェア講座
ちなみに『レンダリングパイプライン』自体はDirectX12も11と変わりません。ちなみにDX9の頃からピクセルシェーダー側ってあまり変わってないんですねえ。

んまあとはいえ、今後はどうなるかわかりませんからねえ。レンダリングパイプラインってのは上の図のような出力までの流れですね。一応シェーダーについてはさっき話したので、よく見てほしいのはラスター化とかね。

流れをとにかく把握しておいてほしい。ちなみに一度ラスター化までくればあとは基

本的にピクセルシェータを通して、レンダーターゲットにレンダリングって事なんですけど、レンダーターゲット二画面上に表示ではない事には注意しておいてください。基本的に裏画面に描画ですが、それ以外の部分にも書き込みます。それによっていろいろとテクがあったりするわけです。

で、DirectX12 をやっていく上ではこのレンダリングパイプラインの把握が非常に重要になってくるので、しっかり頭に叩き込んでおいてください。

ちなみにパイプライン処理に関しては Wikipedia が分かりやすいと思いますので <https://ja.wikipedia.org/wiki/%E3%83%91%E3%82%A4%E3%83%97%E3%83%A9%E3%82%A4%E3%83%B3%E5%87%A6%E7%90%86> 読んでおきましょう。

ちなみに Direct3D ではパイプラインにおけるそれぞれの位置に名前がついています。

1. InputAssembler(IA)ステージ：頂点情報とインデックス情報をもとにポリゴンメッシュを構成するためのモノ
2. VertexShader(VS)ステージ：頂点シェータ
3. HullShader(HS)ステージ：ハルシェータ
4. Tessellator(TS)ステージ：テセレータ(ポリゴンを分割するためのもの)
5. DomainShader(DS)ステージ：ドメインシェータ
6. Rasterlizer(RS)ステージ：ラスタライザ(三角形をピクセル化)
7. PixelShader(PS)ステージ：ピクセルシェータ
8. OutputMerger(OM)ステージ：レンダーターゲットへ出力するためのモノ

大雑把に言うところこんな感じです。それぞれ『ステージ』と呼ばれていますが、レンダリングパイプラインのどの辺に位置するのかを区別する単位だと思ってください。

なんでいちいち頭文字を表したかということ、Direct3D の関数にはこの頭文字を使った関数名がそれなりの数だけ出てくるからです。それではだいたいパイプラインが分かってきたかと思いますので、次にハードウェアの話をしていきます。

ちょっとしたハードウェアの知識

君たちはソフトウェアをプログラミングするので、ハードウェアの知識は要らないように思えるかもしれませんが、そうはいかないんですよ。昔は『最適化』のことを考えるときにハードウェアのことまで考えるという感じでハードウェアについて知ることが大事だったんですが、なんと DirectX12 では『最初のプログラミングの時点でそれなりに知っておかねばならない』という厳しい状況にあります。

GPU と CPU の違いについて

DirectX12 を使うにあたって、ある程度ハードウェアに関する知識を知っておくと便利なので、初歩的な部分を解説しておきます。あくまで初歩的なイメージを付けてもらうための解説なので正確さや詳細を気にする人は専門の本を読んでみてください。

まず、CPU と GPU は何なのかというと、どちらも『演算』を行うという点では共通しています。というより演算だけに関していうと CPU だけで事足りるし、そもそもグラフィックボード自体はディスプレイへの出力が主な役割でした。この時代まではあまり GPU って言葉は一般的に聞かれなかったかな。単なる『3D グラフィックスアクセラレータ』と呼ばれてました。

そのうち 3D アクセラレータとしての役割を持ち前述のラスライズおよびスキャンライン法や 3D に関する演算に特化した演算を行うようになってきて、GPU って言葉が聞かれるようになってきました。

とはいえ最初はシェータを書くこととかなかったし、あまり意識しなかったんだけど DirectX9 あたりでシェータを扱うようになってからは GPU を意識するようになってきました。

さて、この CPU と GPU は何が違うのか？『とにかく GPU を使ったら速いんじゃないの？』と思ってる人もいるかもしれませんが、ちよつと違います。もちろん速いのですが得意とする所が違います。まずは正確さはともかく簡単にイメージを書くと



CPU は難しい問題が解ける博士で

GPU は簡単な計算ができる『集団』

といったイメージを持ってもらうといいかなと思います。

というわけで得意なことが違うんですね。

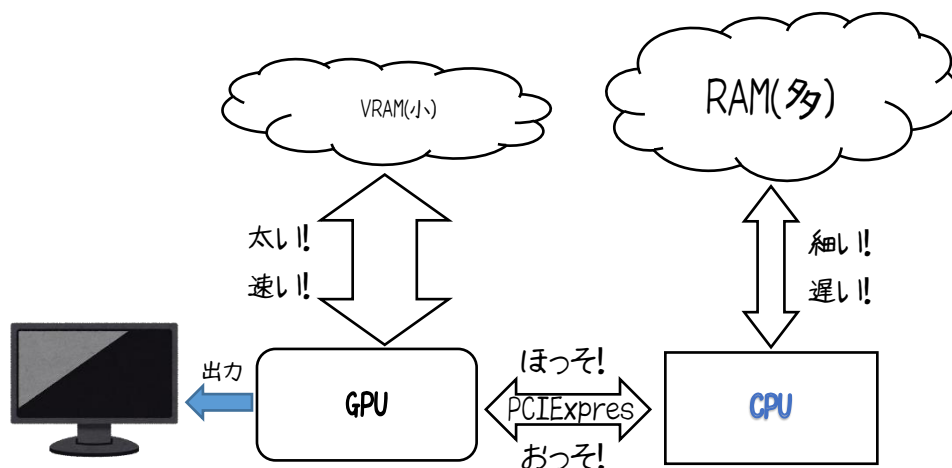
CPU と GPU を比較した場合、個々の能力は CPU 側に軍配が上がりますが、GPU は『数の暴力』で CPU を凌駕します。恐らく GPU はたくさんの頂点やピクセルに対する処理を行う事に特化させるためにそういう進化をしてきたのだと考えられます。



そういう特性もあって、初期の頃のシェータは命令数に限界があったりしました。ちょっと命令数が多いとシェータコンパイル時になんかエラーが出まくるんで工夫してたのを覚えています。今はちょっとくらい無理してもそうそう出ないですね(それでもあまり無理させると出そう)。

違いのもう一つ…メモリとの関係についてお話しておきます。ひとまずは CPU と GPU が分かれてるやつ(nVidia のグラボとか積んでる状態)についてお話しします(ちなみに分かれてる奴をヘテロジニアスといいます)

まずヘテロジニアスのメモリと CPU と GPU の関係の模式図を描くと…こんな感じ。



あくまで比較のため大げさに描いてる部分もありますが、イメージはだいたいこんな感じです。CPU のメモリは元から大いし増設もできるしで…最近のゲーミング PC とかだと 8~16GB とかかなあで、GPU のメモリに関しては 4~8GB なので、大体倍くらい違う事になりますね。

容量の事だけ考えればいいかということ、それだけじゃなくて重要になってくるのが『転送速度』です。当然ながら演算対象を置いておくメモリと、実際に演算を行う GPU や CPU との転送速度で処理速度に差が出てきます。

これが今のところですが、一般的には CPU⇄メモリより GPU⇄GPUメモリ(VRAM)のほうが10倍くらい速いと思われます。これも GPU の処理速度が速いと言われる要因ですね。

ちなみにこの転送経路を『バス(Bus)』と言い、転送速度の事を『帯域幅』とか『バンド幅』とか言ったりします。なんで速度の話に『幅』が出てくるのかというと、たくさんの種類の周波数を送れればそれだけ単位時間当たりの転送情報量が増えるからです。

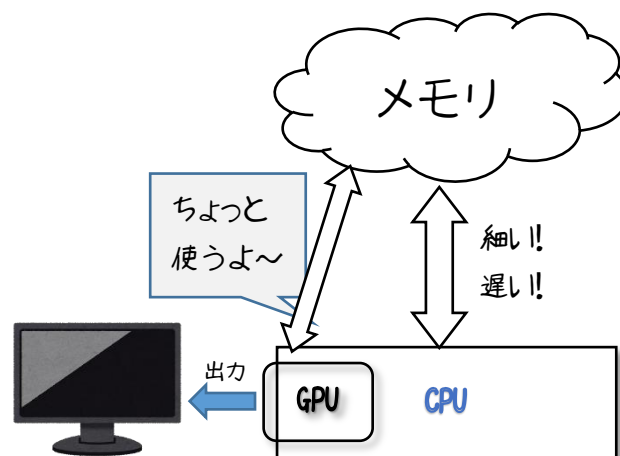
イメージでいうとバスは道路(文字通り乗るバスでも可)で、バンド幅ってのはその道路の幅(もしくはバスの本数)って考えてもらえばいいんじゃないかな。ほら、道幅が広いとさ、それだけ一度に通る車(情報量)が増えるじゃん？

更に CPU と GPU の間でデータを転送する『PCIExpress』という機構です(この先これも変わってくると思います)が、これは CPU⇄メモリよりさらに速度が遅い事が多いです。

GPU 側に全データを置きたくなりますが、ゲームに使用するデータは大きいので全部が全部 GPU 側に置いてしまうわけにもいきませんので、この辺の転送の話も頭に入れておいた方がいいですね。

次に統合型グラフィックス(Intel HD Graphics 等)についてですが、大体のイメージで言うと CPU の家の中に GPU が住まわせてもらってる感じです。

このためメインメモリと VRAM が共用されています。というかメインメモリの一部を VRAM として使用するといった感じです。



メインメモリを VRAM として使用するのを UMA(Unified Memory Architecture)と言います。

DirectX12 では UMA に関するパラメータもあったりするので、頭の片隅にでも置いておいて

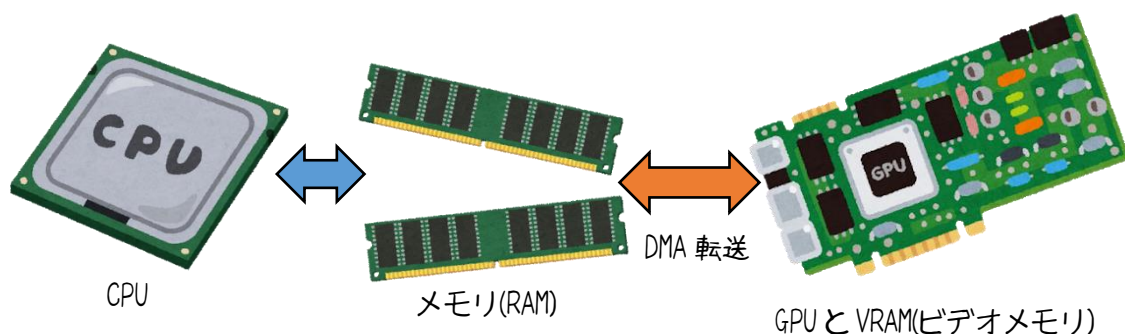
ください。

このため前述の PCIeExpress を介す必要はないのですが、VRAMとしてメインメモリを一部使用するためどっちみち転送速度の弱さは出てくると思います。

どっちが優れてるとかそういうのを言うつもりはないです。なんか論争になると思いますし。なんでここでディスクリート GPU と統合型グラフィックスの話をしたかという DirectX12 でグラフィックスメモリを確保したりデータを転送したりする際にこういうハードウェア周りの知識が必要そうなパラメータが出てくるからです。

まず、皆さんご存じかと思いますが、CPU と GPU がございしますが、ちよつとこの辺の関係がややこしいのです。まず一般的なデスクトップにみられる『ディスクリートな関係』についてお話しします。

ディスクリートな関係というのは CPU と GPU が分かれている構造の事です。



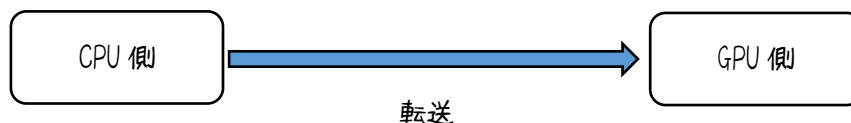
しれっと重要なことを書いてるんですけどね…まあひとまずはこういう関係になってると思ってください。

グラボの上には GPU と VRAM が載っているんですが、GPU 側で使われるメモリが VRAM だと思ってください…そう、ここで賢い人はお分かりになるかと思いますが、CPU が使ってるメモリと違うわけですから…転送が必要なんですよね。それを DMA 転送といいます。

DirectMemoryAccess の略です。ともかく RAM と VRAM が物理的に離れていることにより面倒な問題が発生します。

例えば PNG などの画像ファイルを読み込んで表示するとして、画像ファイルは RAM に読み込まれて、表示は VRAM にデータがないと表示されないわけです。言ってる意味は分かりますよね？

ということは…



で、この転送の速度のことを「バンド幅」「バンド周波数」とか言ったりします。バンド幅ってのは単位時間あたりにいっぺんにデータを転送できる量の事です。

どんなに CPU や GPU が早くなってもこの転送コストは確実にかかってしまいます。DirectX12 では、この転送を意識する必要がちょいちょい出てきます。逆に言うと最適化するために「意識させるように設計されている」とも言えます。ということで、なんとなく程度は知っておく必要があるという事です。

あと DMA 転送と書きましたが、データの転送を行うためには内部的に「コピー」という操作を行わねばならないのですが、そのお仕事を CPU にさせてしまうと CPU がその間止まってしまうため、処理落ちの原因になります。

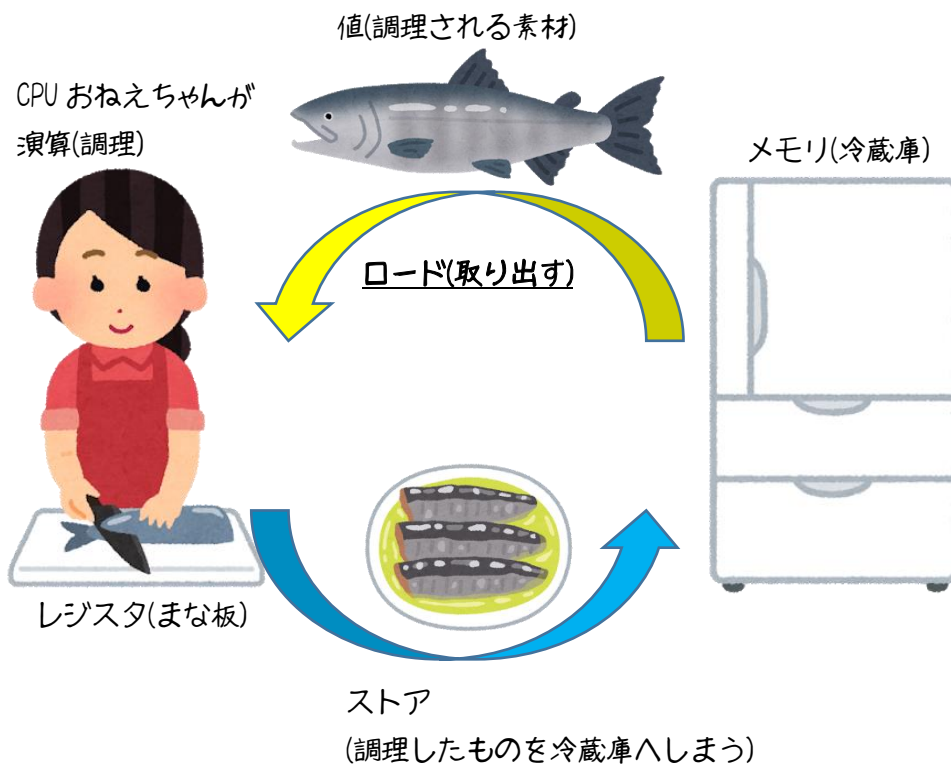
このため、PCIExpress(簡単に言うと CPU 側と GPU 側をつないでる線。実際にはマザーボードにくっついてるやつ)には DMA エンジンという転送のための仕組みがあり、CPU はそいつに対して「転送してくれー」って命令を出します。そしたら CPU の邪魔をすることなく転送が行われ、いつかは GPU にデータがコピーされます。

これに対して、CPU と GPU が一体型になってるものがあり、それは例えば Intel なら、Intel の CPU と Intel の HDGraphics という感じで、同じチップセットにあるものです。前述しましたがこれだと RAM が VRAM の役割を果たすので、転送のコストは減りますが、パワー不足は否めないですね。

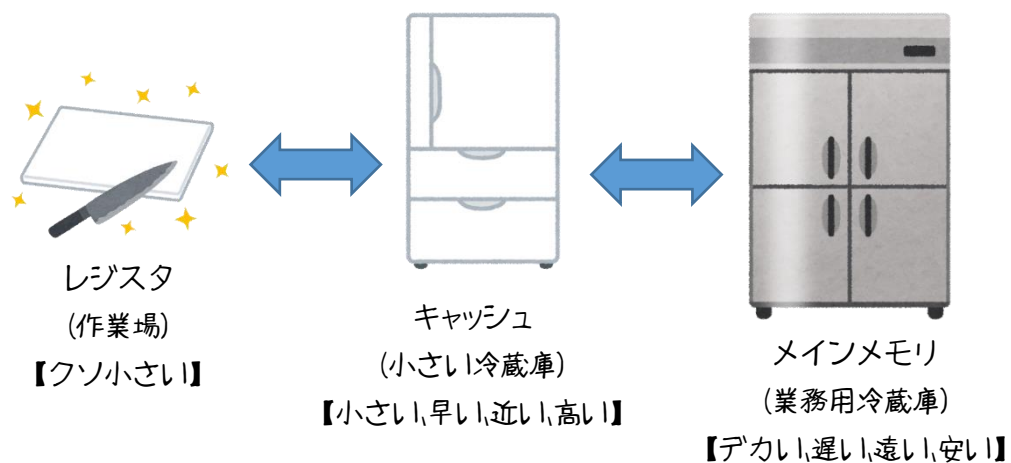
安いノート PC なんかはこれですね。場合によっては、省電力のために通常は HDGraphics のほうを使用し、負荷の高いゲームの場合は nVidia や Radeon などの専用グラボを使うことがあります。

キャッシュメモリについて

先ほどの図式ですが、CPU 側のことを考えると CPU とメモリしかなかったので、メモリから直接 CPU に読み込んで演算すると思ってる人も多いかと思います。前期で話したかもしれませんが、実際にはそうではありません。



こういう図を見せたことがあると思いますが、覚えていませんか？実際の構造はもう少し複雑で



メインメモリへのアクセスは遅くていちいちそこからロードしてくるのは時間がかかるため、近代以降の PC では『キャッシュ』と呼ばれるアクセスしやすい領域を持っておいて、何度もアクセスする場合はキャッシュにおいておけば、作業スピードが速くなるという事です。

分かりやすいところでいうとループ内の処理とかですね。

```
int sum=0;
for(int i=0;i<100;++i){
    sum+=i;
}
```

このiとsumなんかはキャッシュに載るため、こいつへのアクセスが速くなるという事です。もちろんキャッシュの容量が小さいため、載せ続けるわけにはいきません。いつかはいっぱいになります。

キャッシュの大きさにもよりますが、いっぱいになったらどうなるかというと、時間的に『使わない期間が長い』領域から消えていきます。もう使わへんやろということでそういう仕組みになっています。

例えば仮に、完全に仮にの話ですが、キャッシュが8バイトだとします。↑のiとsumでいっぱいの状態です。で、こういうプログラムになっているとします。

```
int sum1=0;
int sum2=0;
for(int i=0;i<100;++i){
    sum1+=i;
    sum2+=i*2;
}
```

このように変数を増やしてしまうとループのたびに変数がキャッシュから解放されてしまい、効率が悪くなります。なので、その場合は

```
int sum1=0;
int sum2=0;
for(int i=0;i<100;++i){
    sum1+=i;
}
for(int i=0;i<100;++i){
    sum1+=i*2;
}
```

としたほうが効率的になります。なお、キャッシュがそんなに小さいことはないのですが、これが効率的とは思わないでください。あくまでも例です。

DirectX 組み込みに入る前に

ひとつ言っておくと、DirectX12 はいまだに日本語訳されていない。多分翻訳されないでしょう。と思ったら翻訳されてやがる…

<https://docs.microsoft.com/ja-jp/windows/win32/direct3d12/directx-12-programming-guide>

これはもはや『翻訳する気がない』のではないだろうかと思う。もしそうなら君たちはチャンスなのだ。日本語訳されないそれだけで『参入障壁』となるからである。

ぶっちゃけこの責め苦に耐えられる奴らは、それでも参入障壁以前にクソ強い事を保証しよう。まあクソ強くてもセオリーは押さえんと負けるので、そこは学ばないといけないし、結局作品は作らないといけないんで、あまり油断しないようにしよう。

うん、でも今は Chrome の『翻訳する』があるから楽だよね。便利なツールはバンバン活用しよう。

まあ資料が英語だけだけど、大丈夫!!どうせプログラミング言語も英語みたいなもんだ!! (実は『英語』という言葉にビビってるだけということはある)

もっと言うと、一部の卒業生は結局英語の論文とか読む羽目になってるらしい。ゲーム開発者になるってのはそういうことです。ああ、楽そうだから他の職業にしよう? 本当に楽かな?

『自分が興味ある事には労力を惜しまない』習慣を今のうちに育てておくのは大事だと思います。ゲーム開発がそうでないというのなら、今のうちに別の何かを自分で探してください。僕もゲーム開発以外でのサポートはできませんので、自分で何とかしてください。

ちなみに DirectX12 の参考訳として

<https://www.isus.jp/games/direct3d-overview-part-1-closer-to-the-metal/>

があるので、一通り目を通しておいてもらおうと、英語のドキュメントも読みやすいと思います。

ちなみに MS のサンプルコードは武骨すぎるので

<https://sites.google.com/site/monshonosuana/directxno-hanashi-1/directx-143>

とか

http://www.project-asura.com/program/d3d12/d3d12_001.html

とか

<http://zerogram.info/?p=1746#more-1746>

とか

<https://qiita.com/em7dfggbcd9/items/483c60fa06bf10f510d7>

とか

http://d.hatena.ne.jp/shuichi_h/20150502

とか

<http://blog.techlab-xe.net/archives/3645>

ついでに

<https://qiita.com/Onbashira/items/25bfa7a0017d6d888e39>

を見ておくといいんじゃないでしょうか？

ちなみにサンプルコードのいくつかは ComPtr を使用していますが、個人的にはあれ使
うと『あ〜、サンプルぶっこ抜いてきただけですね〜』って感じがするので使いたくない
です。まあ、あれ使う意味は解放の際に InternalRelease が呼ばれて→Release()してくれる
からなんだけど…あまり好きじゃないなあ。

もし使用する際にはコメントで『内部で→Release()してくれる ComPtr を利用』してま
すを書いてればいいかな。理解せずに使用するってのがいちばん嫌われる。

あと、ZeroMemory 使ってる参考書や参考サイトあるけど、あれ使うのはやめようね!ダメ!
ゼツタイ!

あくまで参考程度に見ておいて、サンプルコードなどは直接使わないように注意してく
ださい…まあ DX12 相手にそれをやる勇気(無謀さ)のあるやつはそうそういないと思う
けど…。DxLib とか DX9 の開発と同じと思ってコピペをすると死ぬし、横着しようとする
と身をもって思い知ることになるであろう。

あと…喜べ!公式がついに日本語訳し始めたぞ!!!

<https://docs.microsoft.com/ja-jp/windows/win32/direct3d12/directx-12-programming-guide>

DirectX12 がそれ以前の DX と違うのはどこ?ここ?

とりあえずこの授業を聞いている人の中に DirectX11 の授業を受けた人がいないんだよね
(そんな時代になったんだなあ…遠い目)

というわけで、それまでとの違いってのが良く分からないと思います。逆に言うと混乱するこ
とがなくていいかな?こういうもんや!!!って思えば迷う事もないしね。

一応、技術記事とか読んでも『性能差が〜』とか言われてますが、どっちかつちゅうと

DirectX11 時代の問題点に触れておいた方がいゝのかもしれない。恐らく皆さんが DX11 を直接いじることはもうないと思いますのでお話として聞いておいてもらうといいですね。

まず DirectX11 の時はシェーダの切り替えとかステート(後々説明するけど、描画時の設定)の切り替えを 1 命令でやってたんですよ。で、この命令の後に描画される奴はすべてそのシェーダ、ステートで描画されるっていうルールだったんだよね。DXLib も同じなんだよね。

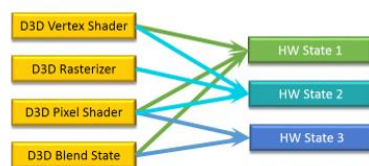
イメージわくかな？

で、1つのモデルの中でもこのステートはバンバン変わっちゃうわけ…一例を出すとモデルが複数のマテリアルでできている場合、描画時にステートを変更しながら別々で描画しなければならないわけ。もっと言うと、ステート切り替えのたびに GPU 命令を上書きするため CPU→GPU オーバーヘッドが発生し、まあ良くない状況になるわけだ。

で、このステート変更のコストがそれなりに高つくわけだ。

Direct3D 11 – Pipeline State Overhead

Small state objects → Hardware mismatch overhead



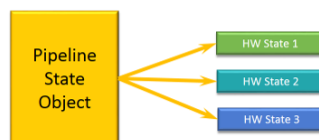
図で説明されてるサイトとかだとこんな感じですね。何となく切り替えコストが無駄になっているのが伝わるかなーと思います。何となくでいいですよ？無理に理解しようとしなくていいです。

で、12ではそういうのをまとめて GPU に投げたおいて、切り替えたいときは参照先…CPU で言う所のアドレスの数値を進めたり戻したりすることで切り替えを実現していると考えてください。

Direct3D 12 – Pipeline State Optimization

Group pipeline into single object

Copy from PSO to Hardware State



で、これを実現しているのがデスクリプタなんか…だったりするんですが、それはさておき、ちょっとこれ以上ここでやってしまうといつまで経っても初期化処理のコーディングにすら入れないので、ここからおおざっぱな話になるけど、

DirectX12 の思想の根底にはこの『おまとめ思想』というものが流れていると思っていただきたい。

コマンドリスト、コマンドキュー、デスクリプターヒープなどが出てきますがそれらは、DX11の時にバラバラだったものをまとめて効率化するためのものだと思ってください。

昨年の僕の設計の失策はこの DirectX12 の思想を理解しないまま DirectX11 の思想のままに設計してしまったために必要以上にややこしく、かつ非効率なものになってしまったということです。

あと、DirectX11 との違いをもう一つ挙げるとするならば『並列化』です。CPU→GPU 命令を逐次実行にするのではなく前の命令の完了を待たずに次の命令を出せるようにしています。このため DirectX11 では結果的に『スレッドセーフ』になっていた部分がスレッドセーフでなくなっており、そのため後述する『バリア』とか『フェンス』とかの仕組みが入ってきているわけです。

はい、DX11 と DX12 の違いのまとめ

メリット

- CPU→GPU の命令を完了復帰から即時復帰にすることで並列に命令を飛ばせるように
- メモリ→VRAM への細かい転送を減らせるような設計になっている
- 命令やらステートをまとめて扱う事で、スイッチングコストを減らせるように
- つまり工夫すれば速度が DX11 の時より上げられる設計になっている

デメリット

- 工夫できるような設計になっているが、工夫しないと寧ろ遅くなることもある
- 設計的に難しく面倒になっている
- 理解が困難。マニュアルが全部英語。情報が少ない。なんかライブラリが変化しすぎ

とまあ学習のハードルが上がっただけに思えますが、いい所も挙げておくけど

『思想に慣れたら、DirectX12 の方が楽に感じる』ので、さっさと慣れましょう。慣れるしか…ないっ!!

仮想メモリ(仮想アドレス)とは

『仮想メモリ』について、軽く説明しておきます。なんですかという、リファレンスに『仮想メモリ』って言葉がよく出てくるからです。

で、仮想メモリってことはバーチャルなメモリなん？って思う人もいると思いますが、その通

りです。実際のメモリとは違うが、実際のように使用できるメモリの事です。どういうことか
というと…

[https://msdn.microsoft.com/ja-jp/library/windows/hardware/hh439648\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/hardware/hh439648(v=vs.85).aspx)

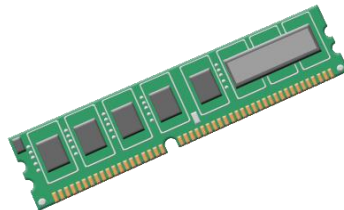
<https://ja.wikipedia.org/wiki/%E4%BB%AE%E6%83%B3%E8%A8%98%E6%86%B6>

にも書いてるんですが、GPUに限らず CPU の頃から『物理メモリアドレス』に対して『仮想メモ
リアドレス』ってのがああるわけ。

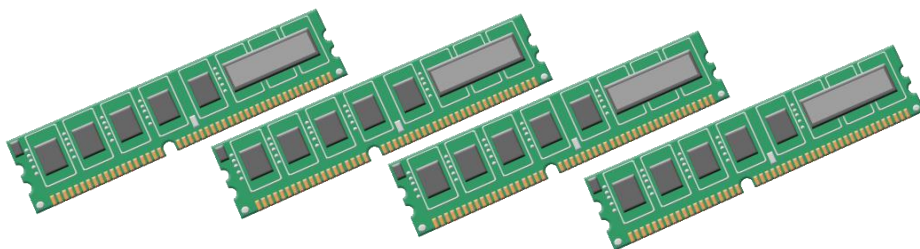
仕組みとしては、物理メモリをそのまま使うより、OS っていうか MMU(メモリマネジメントユニ
ット)がマネジメントした『仮想メモリ』を見たほうが便利なので、基本的にプログラムはこの
仮想メモリを通して物理メモリにアクセスしています。

では、なぜワンクッション置いてアクセスしてるんでしょう？物理メモリに直接アクセスし
たほうが速度も速くなりそうじゃない？なんでこんなけったいな仕組みを使っているんでし
ょうか？

一番の理由は巨大メモリにアクセスするためです。メモリってのはこういうものです。



で、もちろん一本差しではなく、2つ3つ4つ刺さっています。



大雑把に言うと大きなメモリ確保の場合、1本では足りなかったりメモリ間を跨いだりする
わけ。そうなると物理メモリ番地的には連続してないため大量のメモリ確保は不可能にな
るわけだ。

そこをマネジメントすることによって、あたかも連続した大きなメモリ空間であるかのよう
にハードウェアのメモリを見ることができるよう、仮想アドレスを通してメモリアクセスを
していると思ってください。

なお、GPU の仮想アドレスに関しても基本的な意味はこれと同じです。同じですが、GPU 側の仮想アドレスと言った場合、もしかしたらもう少し違う意味かもしれません。

もちろん GPU も GPU も仮想アドレス空間を持っているんですが、GPU 仮想アドレスと言った場合もしかしたら CPU/GPU 共有仮想メモリの事を指しているのかもしれませんが。そこはその時の説明の文章(英語?)を見ないと分かりませんが、とにかくドライバの中身をいじらない限りは物理アドレスに直接アクセスはできませんので、あまり用語に捕らわれることなく、普通にプログラムすればいいと思います。

キャッシュメモリとか分岐予測とか

ここからはマニアックすぎるので与太話として聞いてくれ。キャッシュメモリってのは知ってるかな？もちろんなんとなくは知ってる？

インターネットのキャッシュって知ってるかな？通常は Web サイトのデータというのはアクセスしてはじめてダウンロードされ、画面に表示されているんだけど、これを高速化するために何度もアクセスするようなデータはダウンロード HDD の TemporaryInternetCache という所に残骸が残っていくよね？

で、次にアクセスするときにダウンロードするのではなく、このキャッシュデータを見に行っていたりするんだ。大元の仕組みが今みたいなブロードバンドの時代じゃなくて、ダイヤルアップ回線使ってたナローバンドの時代だからこういう風になってるんだけど、昔は本当に重宝してたんだ。本当にクソ遅かったから。

なんだけど、今はブロードバンドでダウンロードが速いのと、著作権系のデータをローカル HDD に残さないような法整備の流れでこの仕組みもすたれつつある。

とまあ、歴史的な部分はさておき、キャッシュメモリの話だけどこれは CPU からのデータアクセスを高速化するための仕組みだ。

L1,L2,L3 キャッシュというのがあって、L1,L2,L3 の順にアクセス速度が速い……が、L1,L2,L3 の順に容量が小さい。また、演算するための CPU に近い位置に物理的な意味で配置される。

メモリ上のデータから、頻繁にアクセスするものをより分けてそれぞれのキャッシュメモリに置くことで、同じような数値の同じような計算を高速化している。

なので、プログラマ側がここを効率化しようと思ったら、一度に使用するデータはキャッシュに乗っけて一気に計算するように工夫する。ちなみに乗らなかった場合や、欲しいデータがない場合は一度キャッシュが破棄され、別のデータに乗っけて計算が行われる。ここにオーバーヘッドが発生する。

だからゲームプログラマは良く『キャッシュに載るように』とがなんとか言う。

次に分岐予測の話だけど、CPU 側の命令も『パイプライン処理』ってのをやっている。

<https://ja.wikipedia.org/wiki/%E5%91%BD%E4%BB%A4%E3%83%91%E3%82%A4%E3%83%97%E3%83%A9%E3%82%A4%E3%83%B3>

本来直列にシーケンシャルに実行されるものを並列に処理できる工程(ステージ)に分割して並列に処理している。これにより細かいスレッド化のような恩恵が得られている。

で、プログラムの実行の流れで条件分岐命令が分岐するかしないかをよそくしている。これを分岐予測と言う。これが何の役に立つのかと言うと前述のパイプライン処理をスムーズに並列化するためである。

ただし、この予測が外れると巻き戻りが発生し、並列パイプラインの恩恵が受けられなくなる。分岐的な処理は可能な限りなくした方がいい理由はこちらである。でもあまり神経質にならなくてもいいと思う。

分岐を減らした方がいい理由は高速化というより、可読性の問題ですね。高速化もちよつとだけありますけれどもね。分岐を減らしたいばっかりに変なコードを書くとそれはやっぱり遅くなりますしね。

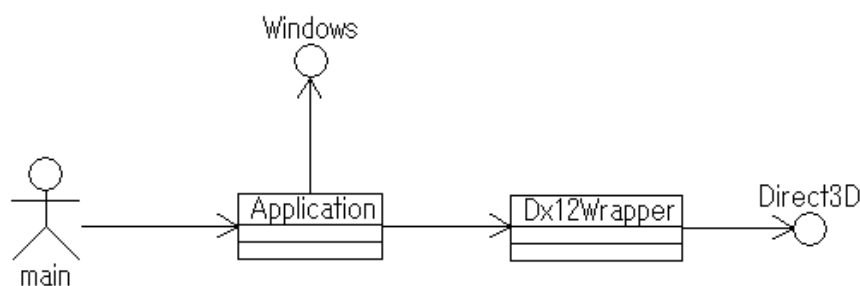
ちなみに for ループ処理の場合、毎回ループ条件が同じであるためほとんどの分岐予測が当たります。N 回ループなら N-1 回は必ず分岐予測が当たるわけです。

ともかく switch 文は要らないってことでいいですね。

とにかく DirectX12 を動かそう(初期化編)

うん。なんでさっきみたいな話を長々とやったかと言うと、これから DirectX12 を組み込むんだけど正直『なしてこんな手間かかるの？アホちゃうん？はあ〜つかえ(MS)やめたら？このゲーム(のためのライブラリづくり)』と言う気になっちゃうからです。

ちょっとその前にですね、Dx12Wrapper クラスを作っておいてください。理由はこれを Application クラスに載せてしまうと Application のコード量が増大しまくるからです。だから



こういうふうにしたい

ひとまず Dx12Wrapper クラスをウィザードかなんかで作っておいてください。なお、Direct3D の初期化にはウィンドウハンドルが必要なので、コンストラクタ等で受け取れるようにしておきましょう。

で、今から DirectX12 を初期化していきます。DxLib_Init 一行で済むような事はなく、DirectX12 を最低限使える状態にするまでに

基本初期化として

- D3D12Device(デバイス周り)
- DXGI SwapChain(画面フリップ周り)

を設定して、画面に影響を与えるためには

レンダーターゲットが必要です。レンダーターゲットっていうのは、絵を書き込むバッファの事です。これを画面とバインド(結び付けてやる)してやることによって画面上に絵が表示されるんです。

レンダーターゲットってのはピクセルの集合体です。つまりテクスチャと一緒にです。

DirectX12 ではテクスチャなど、VRAM を食いつぶすオブジェクトをリソースとして定義します。ID3D12Resource* という型で定義されます。

そして、当然のことながら GPU 上にそのメモリを確保する命令を出す必要があります。そのほか、Direct3D に対して描画関係の命令を出すにはコマンドリストという物が必要で、DirectX11 のように個別で命令を出すのではなく、まとめて命令を出します。

ということで

- D3D12DescriptorHeap
- D3D12CommandList(と D3D12CommandAllocator)
- D3D12CommandQueue

の初期化が必要となります。

で、先にも書きましたが、画面自体が『リソース』です。それを GPU 内に確保します。そして『更新』します。そしてほんとくと確保や更新を待たずに処理が進みます(実際にはスワップチェーンを生成した時点でレンダーターゲット用のリソースは作成済みであり、スワップチェーンから取得することになる。しかし、別レンダーターゲットを使用する際には自前でリソースを確保しなければならないことは覚えておこう)

ところで画面を更新する際には DxLib においては ScreenFlip がありましたね？

うん、で、確保、更新が完了しないまま ScreenFlip(Direct3D では Present 処理)すると…まずいですよ!!!

ということで、リソースバリア、フェンスなどで待ちやブロッキングを入れてあげる必要があります。

準備①(インクルードとリンク)

とりあえず必要なのは direct3d12 の定義なので

#include<d3d12.h>をします。

もうひとつおまけに

#include <dxgi1_6.h>します

次にリンクするために以下のコードをどっかのcppにリンクコードを書きます。

```
#pragma comment(lib, "d3d12.lib")
```

```
#pragma comment(lib, "dxgi.lib")
```

基本的な部分の初期化

ここからは先に書きましたが、既にラッパークラス Dx12Wrapper を作っている前提で話

をします。

で、メンバ変数として最低限

```
ID3D12Device* _dev = nullptr;  
ID3D12CommandAllocator* _cmdAllocator=nullptr;  
ID3D12GraphicsCommandList* _cmdList=nullptr;  
ID3D12CommandQueue* _cmdQue = nullptr;  
IDXGIFactory6* _dxgi = nullptr;  
IDXGISwapChain4* _swapchain = nullptr;  
ID3D12Fence* _fence = nullptr;
```

が必要になってくるわけだが、さて…まあインクルード問題…ぐぬぬ。

うん、皆さんは真似しないでいいんですけど前方宣言でなんとかするかな…それとももう include 解禁しちゃうかな…。

どの道、標準関数は include するしなあ。こんなところで悩んでてもなあ…。よし、

- 標準関数
- DirectX の関数
- Geometry.h などの基本構造体のやつ

は OK というルールにするかな。あんまし無理してもな…(´；ω；`)

という事で泣く泣く OK にする。まあ頻繁に変更がかかるものでもないしいいよね。

さて、ということでデバイスを生成します。

D3D12CreateDevice って関数です。

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

わあ英語だ。まあ慌てんな…そういう時はだな Chrome に翻訳させればええんじゃないよ。ここでは英語版と並行して読もう。

もちろん使い方とか引数の数とかは違うけど、だいたい概要は一緒なので気にすんな。

HRESULT D3D12CreateDevice(

 IUnknown *pAdapter, // nullptr で OK

 D3D_FEATURE_LEVEL MinimumFeatureLevel, // フィーチャレベル

 REFIID riid, // 後述


```
void                **ppDevice//後述
);
```

で、DirectX12 の場合、この最後の 2 つの引数がちょっと特殊なんだけど、マクロを使う必要があります。

IID_PPV_ARGS というマクロを使います。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/ee330727\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/ee330727(v=vs.85).aspx)

英語解説しかありません。

ともかく、これにデバイス用のポインタのポインタを入れて、CreateDevice に渡すと、REFID と中身の入ったデバイスを返してくれるという優れモノなのです。

この REFID は自分でどうこうするのは無理な ID なので、マクロを使うしかありません。おとなしくマクロのお世話になりましょう。

この IID_PPV_ARGS マクロは非常に何度も使用機会がありますので、覚えておきましょう。まあ、ここは大して重要なわけでもないのですが、ここで問題なのは第二引数のフィーチャレベルです。

https://docs.microsoft.com/ja-jp/windows/desktop/api/d3dcommon/ne-d3dcommon-d3d_feature_level

いくつかあるのが分かると思いますが、これ、DirectX のバージョンに対応してるのがなんとなくわかるでしょうか？

で、可能な限り新しいバージョンを使いたい場合にはどう書いたらいいのでしょうか？ちなみにハードウェアがそのフィーチャレベルに対応していなければ CreateDevice は失敗し、S_OK 以外を返します。

この状態で一番いいフィーチャレベルを選択するにはどうしたらいいのだろうか？対応していなければ失敗することが分かっているんだから、高いレベルから試せばいい。つまり、

```
D3D_FEATURE_LEVEL levels[] = {
    D3D_FEATURE_LEVEL_12_1,
    D3D_FEATURE_LEVEL_12_0,
    D3D_FEATURE_LEVEL_11_1,
    D3D_FEATURE_LEVEL_11_0,
```

```
};
```

で、フィーチャレベルを配列化しておきます。あとはループさせながら D3D12CreateDevice を実行し、成功したらループを抜けます。

```
D3D_FEATURE_LEVEL level = D3D_FEATURE_LEVEL::D3D_FEATURE_LEVEL_12_1;
HRESULT result = S_OK;
for (auto l : levels) {
    result = D3D12CreateDevice(nullptr, l, IID_PPV_ARGS(&_dev));
    if (SUCCEEDED(result)) {
        level = l;
        break;
    }
}
```

まあ、学校の PC ならどれか引けちゃうんで…多分 12_0 くらいが引けちゃうはず。

あー、言い忘れてたけど、CreateDevice だけでなく、DirectX ではポインタのポインタをひきすうで受け取るものがあるんですが、そういう時はまず変数をポインタで宣言しておいて、それに &つけてポインタのアドレスを示してる状態にして渡してあげます。

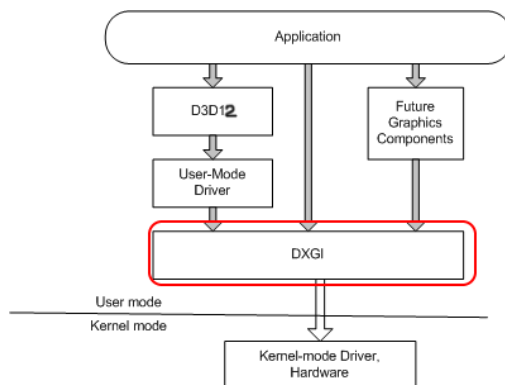
分からなかったり、納得できない人はすぐに言ってください。フォローの講義をしますので…このへん納得できないまま進むと死ぬんで遠慮なく聞いてください。

で、次ですが、DXGISwapchain です。こいつは画面のフリップとかに必要なものです。

で、ここに出てくる DXGI という言葉ですが、これもキーワードです。

[https://msdn.microsoft.com/ja-jp/library/ee415b71\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee415b71(v=vs.85).aspx)

I はインターフェースじゃなくてインストラクチャーなんやなあ…。とにかく DXGI は表示デバイスとグラボに関わる部分で、Direct3D の 1 個下にある(ドライバに近い)レイヤーなんですよ。



一応1.6の改善部分を見ると

<https://docs.microsoft.com/en-us/windows/desktop/direct3ddxgi/dxgi-1.6-improvements>

HDR 対応とか書いてますね。まあそういうのをやる部分って事です。
ともかく初期化しましょう。

dxgi1.6 で検索しましょう。

https://docs.microsoft.com/en-us/windows/desktop/api/dxgi1_6/

で IDXGIFactory6 があるわけですが、

https://docs.microsoft.com/en-us/windows/desktop/api/dxgi1_6/nndxgi1_6-idxgifactory6

これどうやって実体を作るのか書いてないんですよ。ひどくない？仕方ないんで公式サンプル見ると

CreateDXGIFactory1 を使ってるんだよね…大丈夫なん？

[https://msdn.microsoft.com/ja-jp/library/ee415212\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee415212(v=vs.85).aspx)

なんでか日本語やな…

```
HRESULT CreateDXGIFactory1(
```

```
    REFIID riid,
```

```
    void **ppFactory
```

```
);
```

引数はデバイスの生成の時と同じなので問題なさそうなんですけど、行けるのかなあ…ホントマ DirectX12 の仕様とマニュアルいれ加減固めろや…。

で、通るし、S_OK 返ってくるしでいいんだろうけど…納得いかん。

はき

ちなみに CreateDXGIFactory2 っていうのもあるんだけど、こいつは

DXGI_CREATE_FACTORY_DEBUG か 0 を受け取るもののようです。第一引数に 0 入れて成功するんで、別にどっちでもいいっぽいんです。引数パターンの違いだけみたいですね。

とりあえずここまでできたら基本的な初期化ができたということで…ここからがまた…地獄の始まり

画面に影響を与える準備

画面に影響を与えるためには前にも書きましたがまず表示画面のための

- スワップチェーン

- レンダーターゲットビュー(デスクリプタハンドル)

描画等の命令のための

- コマンドキュー
- コマンドリスト

ビューをメモリ上に配置するための

- デスクリプターヒープ

さらにさらにそれをまとめるための

- デスクリプターテーブル
- ルートシグネチャ

最後にレンダリングパイプラインをまとめた

- パイプラインステートオブジェクト

まとめまくりですねー。でもまだあるんだごめんね。

前にも言ったように命令が即時復帰のために待ちの仕組みを用意してあげなきゃならない。それが

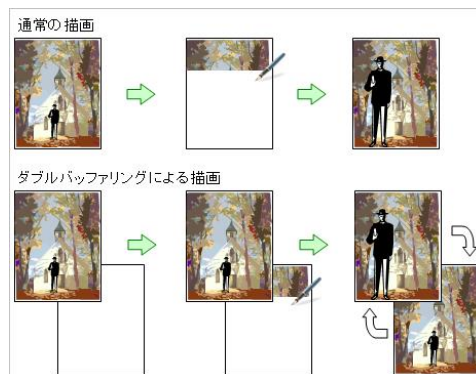
- フェンス

である

さて、これだけの D3D12 オブジェクトを用意する必要があるんだね。死ぬ。

スワップチェーン

スワップチェーンとは何かというと、DxLib の時に ScreenFlip()ってやってましたよね？



ダブルバッファリングと言って、表示すべきものをディスプレイに直接描画するのではなく、別のメモリに裏で書き込んでおいて、表示の直前でさっと入れ替えるものです。

そこは理解していますか？

オーケー、それならスワップチェーンは理解できると思います。こいつはその裏画面と表画面を入れ替える処理をコントロールするものなのだ。ちなみに ScreenFlip は 2 画面の入れ替えだが、スワップチェーンはそれ以上も想定しています。

ただし…大抵の場合は2画面で十分だと思います。

でスワップチェーンを作るときには、例によって CreateSwapChain 的な関数を使うんだけど、ウィンドウと関連付けるためにウィンドウハンドルとバインドする関数 CreateSwapChainForHwnd を使用する。

<https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404557>

こいつを見てくれ。どう思う？

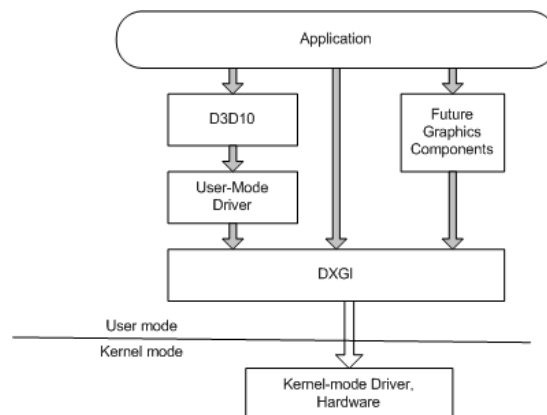
うーん。まだわからんかな？

こいつの持ち主が

IDXGIFactory6

になっている。つまり IDXGIFactory6 型のオブジェクトにアローは演算子を使ってコールしていくわけです。

のほうはまだわかりやすいかな(DirectX10 の説明だけど)



ご覧のように、かなりハードウェアに近い部分であることが分かります。

恐らくスクリーンフリップ(ダブルバッファリング)などの処理はここに含めておいた方がいいという判断なのでしょう。設計思想はよくわかりませんが。

ともかく

IDXGIFactory4 を使うのですが、こいつのインターフェイスを持ってくるには CreateDXGIFactory1 関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/ee415212\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee415212(v=vs.85).aspx)

ここは『知ってなきゃわからない』部分なので、ソースコード書いちゃいますが

```
IDXGIFactory4* factory = nullptr;  
result = CreateDXGIFactory1(IID_PPV_ARGS(&factory));
```

こうやって作ります。result が S_OK なのを確認してください。

さて、それではスワップチェーンの生成に取り掛かるんだが
一度これを読んでおいたほうがいい

https://www.isus.jp/wp-content/uploads/pdf/625_sample-app-for-direct3d-12-flip-model-swap-chains.pdf

比較的…比較的分かりやすいです。

CreateSwapChainHwnd を使用するのだが、まずは DXGI_SWAP_CHAIN_DESC1 についてみてみよう。たぶんスワップチェーンにおいてはこれが一番大事。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404528\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404528(v=vs.85).aspx)

DXGI_FORMAT

[https://msdn.microsoft.com/ja-jp/library/bb173059\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb173059(v=vs.85).aspx)

定義を見るとこうなってますね？

```
typedef struct _DXGI_SWAP_CHAIN_DESC1 {  
    UINT                Width; //書き込み先の幅(ウィンドウ幅と同じでOK)  
    UINT                Height; //書き込み先の高(ウィンドウ高と同じでOK)  
    DXGI_FORMAT         Format; //DXGI_FORMATの項を参照するように  
    BOOL                Stereo; //よく分からないので後で解説する  
    DXGI_SAMPLE_DESC    SampleDesc; //マルチサンプルの数と品質(countを1にQualityを0に)  
    DXGI_USAGE          BufferUsage; //バッファの使用法(あとで解説)  
    UINT                BufferCount; //バックバッファの数(2でいい)  
    DXGI_SCALING         Scaling; //DXGI_SCALING_STRETCHでいい  
    DXGI_SWAP_EFFECT     SwapEffect; //DXGI_SWAP_EFFECT_FLIP_DISCARDでいい  
    DXGI_ALPHA_MODE      AlphaMode; //DXGI_ALPHA_MODE_UNSPECIFIEDでいい  
    UINT                Flags; //0でいい  
} DXGI_SWAP_CHAIN_DESC1;
```

DXGI_FORMAT

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173059\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173059(v=vs.85).aspx)

DXGI_USAGE

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173078\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173078(v=vs.85).aspx)

DXGI_SAMPLE_DESC

[https://msdn.microsoft.com/ja-jp/library/bb173072\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb173072(v=vs.85).aspx)

DXGI_SCALING

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404526\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404526(v=vs.85).aspx)

DXGI_SWAP_EFFECT

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173077\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173077(v=vs.85).aspx)

DXGI_ALPHA_MODE

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404496\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404496(v=vs.85).aspx)

DXGI_SWAP_CHAIN_FLAG

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173076\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/bb173076(v=vs.85).aspx)

さて、書き込み幅はともかく他が良く分かりませんか？

というわけで、まずは Format から…これはビット数に関わってくるのですが、1 画素1バイトなら

＝横幅×高さ

で済むんですが、もしフルカラーの場合であれば 1ピクセルあたり R8ビット G8ビット B8ビット A8ビットを使用しています。この場合であれば

DXGI_FORMAT_R8G8B8A8_UNORM にしています。

なお、UNORM というのは何かというと

『符号なし正規化整数。n ビットの数値では、すべての桁が 0 の場合は 0.0f、すべての桁が 1 の場合は 1.0f を表します。0.0f ～ 1.0f の均等な間隔の一連の浮動小数点値が表されます。たとえば、2 ビットの UNORM は、0.0f、1/3、2/3、および 1.0f を表します。』

簡単に言うと 0～255 を 0.0～1.0 にしているものだと思います。例えば 128 だと 0.5 とかそういう事です。

なお特に初心者の間は、動かなくなった時に、どの時点でのバグが起きたか分かりづらいので、1つ1つ潰して行ってください。

次に USAGE ですが、これは

[https://msdn.microsoft.com/ja-jp/library/bb173078\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb173078(v=vs.85).aspx)

の中から選ぶんですが、今回は

`DXGI_USAGE_RENDER_TARGET_OUTPUT`

を使用します。

実は `DXGI_USAGE_BACK_BUFFER` かな〜って思ってたんですが、色々なサンプル見てると

`DXGI_USAGE_RENDER_TARGET_OUTPUT`

ばかりなのでひとまずこれにしておきます。で、画面更新が滞りなくできたら、その時に `BACK_BUFFER` に変えてみる実験をしようかと思います。ちなみにそれぞれの説明は

`DXGI_USAGE_BACK_BUFFER` サーフェスまたはリソースをバックバッファとして使用します。

`DXGI_USAGE_DISCARD_ON_PRESENT` このフラグは、内部使用のみを目的としています。

`DXGI_USAGE_READ_ONLY` サーフェスまたはリソースをレンタリングのみに使用します。

`DXGI_USAGE_RENDER_TARGET_OUTPUT` サーフェスまたはリソースを出力レンダーターゲットとして使用します。

`DXGI_USAGE_SHADER_INPUT` サーフェスまたはリソースをシェーダーへの入力として使用します。

`DXGI_USAGE_SHARED` サーフェスまたはリソースを共有します。

とあります。

となっているんですが、この説明を見ても `BACK_BUFFER` でもいいような気がするんですよ〜。というわけで、こういう疑問を君たちも持てるようになってください。(※追記、さっき検証したらどっちでも動きました。これもう分かんねえな…)

あと、Stereo に関してですが、ちょっと Google 翻訳にかけてみましょう。

ステレオ

全画面表示モードまたはスワップチェーン/バックバッファをステレオにするかどうかを指定します。ステレオの場合は TRUE。それ以外の場合は FALSE です。ステレオを指定する場合は、フリップモデルスワップチェーン(つまり、SwapEffect メンバーに `DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL` 値が設定されたスワップチェーン)も指定する

必要があります

という事らしいです。でもステレオ言うてもこれ音声の事ちゃうしなあ…。とりあえず良く分からないので、false にしておきます。

あ、そういえば今一度 CreateSwapChainForHwnd を見てみましょう。

<https://msdn.microsoft.com/ja-jp/library/windows/desktop/hh404557>

第一引数の説明を見てみてください。

pDevice [in]

For Direct3D 11, and earlier versions of Direct3D, this is a pointer to the Direct3D device for the swap chain. For Direct3D 12 this is a pointer to a direct command queue (refer to ID3D12CommandQueue). This parameter cannot be NULL.

例によって Google 翻訳

pDevice [in] Direct3D 11 およびそれ以前のバージョンの Direct3D では、これはスワップチェーンの Direct3D デバイスへのポインタです。**Direct3D 12 では、これはダイレクトコマンドキューへのポインタ**です(ID3D12CommandQueue を参照)。このパラメータは **NULL にすることはできません**。

おっとお？

どうも『コマンドキュー』とやらが必要なようですね。

つまり

```
result = dxgiFactory->CreateSwapChainForHwnd(dev,  
    hwnd,  
    &swapChainDesc,  
    nullptr,  
    nullptr,  
    (IDXGISwapChain1**)( &swapChain ));
```

ではなく

```
result = dxgiFactory->CreateSwapChainForHwnd(commandQueue,  
    hwnd,  
    &swapChainDesc,  
    nullptr,  
    nullptr,
```

```
(IDXGISwapChain1**>(&swapChain));
```

にすべきところですよ。

で見ればわかるように、コマンドキューを事前に作っておく必要があります。

早速コマンドキューを作りましょう。

コマンドキュー

コマンドキューは device の CreateCommandQueue 関数で生成できるのですが

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12device-createcommandqueue>

問題は第一引数です。COMMAND_QUEUE_DESC とって結構設定しなければならないのですが、これでもまだマシな方なんですよね…。

```
D3D12_COMMAND_QUEUE_DESC cmdQDesc = {};
```

```
cmdQDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
```

```
cmdQDesc.NodeMask = 0;
```

```
cmdQDesc.Priority = D3D12_COMMAND_QUEUE_PRIORITY_NORMAL;
```

```
cmdQDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
```

```
result = _dev->CreateCommandQueue(&cmdQDesc, IID_PPV_ARGS(&_cmdQue));
```

ちなみに一部引数の説明をヘキサドライブのブログでやられてますのでご参考にどうぞ

<https://hexadrive.jp/hexablog/program/13072/>

この戻り値が S_OK になるところをご確認ください。それができたらスワップチェーンも初期化できます。

スワップチェーン

ちなみに SWAPCHAIN_FLAGS に関しては

[https://msdn.microsoft.com/ja-jp/library/bb173076\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb173076(v=vs.85).aspx)

を見てやればだいたい何をしたらいいのかわかります。

DXGI_MODE_SCALING も同様です。

[https://msdn.microsoft.com/ja-jp/library/bb173066\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb173066(v=vs.85).aspx)

DXGI_SWAP_EFFECT に関してですが

https://docs.microsoft.com/en-us/windows/desktop/api/dxgi/ne-dxgi-dxgi_swap_effect

これは Present 関数呼び出し時に何をするかっていう話なんですけど、

ひとまず FLIP_DISCARD を選んでください。役割はフリップした後にディスカード(破棄)します。つまり Present 前に裏画面に書き込んでおき、Present でフリップし、前の画像は破棄するって意味です。

BufferCount はバックバッファの数なので 2 を指定(表画面と裏画面で2)してください。

つまりと、こうなります。

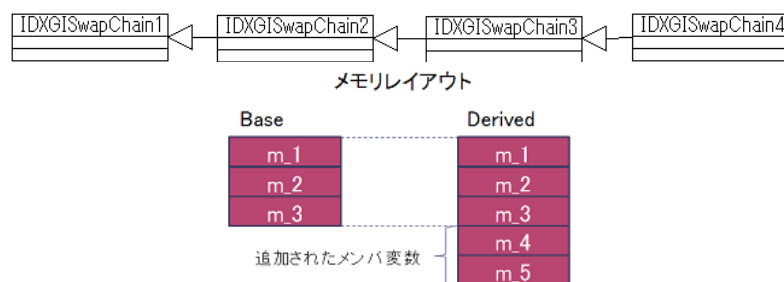
```
Size wsize = appH.GetWindowSize();
```

```

DXGI_SWAP_CHAIN_DESC1 swapchainDesc = {};
swapchainDesc.Width = wsize.w;
swapchainDesc.Height = wsize.h;
swapchainDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
swapchainDesc.SampleDesc.Count = 1;
swapchainDesc.SampleDesc.Quality = 0;
swapchainDesc.Stereo = false;
swapchainDesc.Scaling = DXGI_SCALING_STRETCH;
swapchainDesc.Flags = DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH;
swapchainDesc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_DISCARD;
swapchainDesc.BufferCount = 2;
result = _dxgi->CreateSwapChainForHwnd(_cmdQue, hwnd, &swapchainDesc,
    nullptr, nullptr, (IDXGISwapChain1**>(&_swapchain));

```

しつこいようですが、必ず result を確認してください。なお、最後の引数をキャストしてますが、IDXGISwapChain1 と IDXGISwapChain4 の関係が



という関係なので、キャストしても OK... なんだけどさあ... もうちょっと何とかありませんかねえ。一応継承におけるメモリレイアウトは…
 となるため、キャストしても問題ないわけです。
 とまあこれでスワップチェーンは終わりです。

レンダーターゲットの作成

ちょっと時間ないんで前のテキストまんまコピーしますが、
 というわけで今回必要なものは

- 2 枚のレンダーターゲット(フリップのために 2 枚)
- レンダーターゲットビュー
- デスクリプターヒープのサイズ(整数型)を記録
- デスクリプターヒープ
- デスクリプターハンドル

となります。メンドクサイですね。ほんと。

手順としては

1. デスクリプターヒープを作る
2. デスクリプターハンドルを作る
3. スワップチェーンからレンダーターゲットを取得
4. レンダーターゲットビューを作成

で、なんでレンダーターゲットを生成して使うまでになぜかデスクリプターとかいうのが必要なんだろう、

<https://docs.microsoft.com/ja-jp/windows/desktop/direct3d12/resource-binding>

によると、Descriptor とは『リソースバインディング』の基本単位のこと

リソースバインディングってのはリソースとシェーダ(パイプライン)のリンク(バインド)って事です。で、

<https://docs.microsoft.com/ja-jp/windows/desktop/direct3d12/creating-descriptors>

によると、レンダーターゲットビューもそのお仲間になってるわけだ。

ビューとデスクリプター

実は去年のテキストではデスクリプターヒープを『ビューみたいなもん』と記載していたが、そうではなく、ビューも『デスクリプタの一種』という扱いで抽象化されている。言い換えると各ビューとかサンプルとかの親がデスクリプタって感じ。

ああ、で、何度も当然のように『ビュー』って言ってますが、これが何なのか軽く説明しておくと『画像などのリソースとその見方のペア』です。例えばレンダーターゲットビューなら、表示画面のためのデータとその見方なので、RGBA ペンキ職人と元絵のペアみたいなもんだと思ってください。



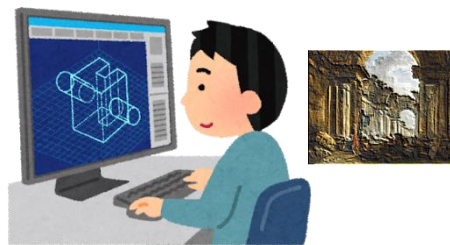
元のデータと、実際に塗る塗り方ね。後から出てきますが、深度値入りのもいて、同じ情報から深度値を書き込んでいく職人もいますので、データと、色んな見方があるわけで、それをまとめてビューと呼んでいるわけです。

で、DX11 までは扱いが別々だったんですが、これらを抽象化したデスクリプタという概念を作って、まとめられるようにしたものが Dx12 であり、これをまとめたものがデスクリプタテーブルです。ちなみにビューだけでなくサンプラーとかテクスチャ(シェーダリソースビュー)などもデスクリプタです。かなりまとめるつもりのようです。

デスクリプターヒープとデスクリプターテーブル

軽くデスクリプターヒープについて解説しておく...

『デスクリプタヒープ』という概念はデスクリプタテーブルと概念的な区別が難しいのですが



<https://docs.microsoft.com/ja-jp/windows/desktop/direct3d12/descriptor-heaps-overview>
を見ると

"Direct3D 12 does require the use of descriptor heaps, there is no option to put descriptors anywhere in memory."

要約すると...そもそもデスクリプタを単体で実体を作れるようになっておらず、とにかくデスクリプタヒープを利用しろということです。ヒープの特定の場所(アドレス)を特定のデスクリプタを割り当てるべきであり、通常の変数のように任意のメモリ位置にデスクリプタを作ることはできません。

一つのことです。意味が分からないかもしれませんが、要はビューを作りたければまずデスクリプターヒープを作って、その中にビュー定義を配置しなさいという事のようにです。

ヒープって言葉が出てきましたが分かりますか？プログラミングの時によく出てくる用語なんですけど、要は作業のために必要なメモリ領域。それを動的に確保しているその領域の事です(malloc だの new だので確保できる領域の事です)

デスクリプターヒープとデスクリプターテーブルの違いは、ビューを割り当てるための場所がヒープで、それを並べて活用できるようにまとめたのがデスクリプターテーブルってことです。

まとめると

- ビューとデスクリプタの関係はポリモーフィズムの子と親みたいなもん
- デスクリプタはデスクリプタヒープからしか利用できない
- デスクリプタテーブルはそのデスクリプタをインデックスで並べたもの

です。たしかに DX11 から来た人にとってはビューの代わりと言えはそうかもしれないが、それだとたぶん誤解してしまうので、ちょっと面倒な説明しました。で、

デスクリプターヒープを作るには

CreateDescriptorHeap 関数を使います。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788662\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788662(v=vs.85).aspx)

```
HRESULT CreateDescriptorHeap(  
    [in] const D3D12_DESCRIPTOR_HEAP_DESC *pDescriptorHeapDesc,  
        REFIID riid,  
    [out] void **ppvHeap  
);
```

第二、第三引数はいつものパターンですね。

問題は第一引数ですが、

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770359\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770359(v=vs.85).aspx)

を見ながらやっていきましょう。

今回はレンダーターゲットに使用するので、Type は

D3D12_DESCRIPTOR_HEAP_TYPE_RTV

ですね。ちなみに RTV は "Render Target View" の略です。

次に Flags ですが、特に指定しないのでデフォルトを表す NONE を使いましょう。

D3D12_DESCRIPTOR_HEAP_FLAG_NONE

次に NumDescriptors ですが、こいつはヘルプを見るだけじゃ分からない。だって

The number of descriptors in the heap.

うん…ヒープの中のデスクリプタ数って事だけど、ちょっと情報量少なすぎませんか？

なのでサンプルを見ながら考えましたが、とりあえず表画面と裏画面で2にしておきましょう。

最後に NodeMask ですが、こいつは**ゼロでいい**です。これは説明に

For single-adapter operation, set this to zero. If there are multiple adapter nodes, set a bit to identify the node (one of the device's physical adapters) to which the descriptor heap applies. Each bit in the mask corresponds to a single node. Only one bit must be set. See [Multi-Adapter](#).

って書いてるからです。

//----表示画面用メモリ確保-----

```
ID3D12DescriptorHeap* descriptorHeap = nullptr;
D3D12_DESCRIPTOR_HEAP_DESC descriptorHeapDesc = {};
descriptorHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV; //レンダーターゲットビュー
descriptorHeapDesc.NodeMask = 0;
descriptorHeapDesc.NumDescriptors = 2; //表画面と裏画面ぶん
descriptorHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
result = dev->CreateDescriptorHeap(&descriptorHeapDesc, IID_PPV_ARGS(&descriptorHeap));
これもまた S_OK が返ってくるまで頑張りましょう。これで 2 個のレンダーターゲットビュー
(デスクリプタ)を格納できるデスクリプターヒープができました。
```

格納先を確保したので、次に実際にここ↓にビューの定義を突っ込みましょう。

その前にデスクリプターヒープサイズを計算します。2つめは 1 つ目の後ろに配置したいので 1 つ目を書き込んだ後の場所を知りたいからです。

GetDescriptorHandleIncrementSize という関数を使用して計算します。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899186\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899186(v=vs.85).aspx)

ヘルプを見れば分かるようにいたって簡単です。ヒープのタイプを入れれば勝手に計算してくれます。

```
heapSize=dev->GetDescriptorHandleIncrementSize(DX12_DESCRIPTOR_HEAP_TYPE_RTV);
```

で終わりです。殺伐とした DirectX12 の中でこの関数は久々に心がほっこりするね。そしてレンダーターゲットビューを作る関数は当然のように CreateRenderTargetView ですから

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12device-createrendertargetview>

```
void CreateRenderTargetView(
```

```
    ID3D12Resource          *pResource, //ピクセルを書き込む本体
```

```
    const D3D12_RENDER_TARGET_VIEW_DESC *pDesc, //レンダーターゲットビューの仕様
```



```
D3D12_CPU_DESCRIPTOR_HANDLE      DestDescriptor//ヒープ内の場所
);
```

ご覧のように…3つとも定義が大変そう!!さてどうしたものか。

でも、リソースと VIEW_DESC は心配しなくていいです。実はスワップチェーンを作った時点で画面を表すリソースができているのだ。こっちはそれに対してビューを割り当てればいいだけです。逆に第3引数の扱いが少々面倒なため、

D3D12_CPU_DESCRIPTOR_HANDLE

```
DXGI_SWAP_CHAIN_DESC swcDesc = {};
dx12.GetSwapchain()->GetDesc(&swcDesc);

int renderTargetsNum = swcDesc.BufferCount;
//レンダーターゲット数ぶん確保
renderTargets.resize(renderTargetsNum);
//デスクリプタ1個あたりのサイズを取得
int descriptorSize = dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
for (int i = 0; i < renderTargetsNum; ++i) {
    result = dx12.GetSwapchain()->GetBuffer(i, IID_PPV_ARGS(&renderTargets(i)));//『キャンバス』を取得
    dev->CreateRenderTargetView(renderTargets(i), nullptr, descriptorHandle);//キャンバスと職人を紐づける
    descriptorHandle.ptr+= descriptorSize;//職人とキャンバスのペアのぶん次の所までオフセット
}
}
```

こんな感じになります。今回必要なものはレンダーターゲットの持つリソースを取得し、それをレンダーターゲットビューと関連付ける情報を作りデスクリプターヒープに書き込む…これだけです。

ちなみにデスクリプタテーブルに関してはまた後で記述します。たぶんシェータを書き始めないと『なんで?』っていうのが分からないから。

さて、いよいよ画面のクリアだ

コマンドを投げるために…

画面をクリアするためには『画面をクリア』というコマンドを発行する必要があり、コマンドを発行するという事は、コマンドリストとコマンドアロケータが必要になる。

既にコマンドキューは作っている(スワップチェーン作るとき)。

作り方はいたって簡単。

CreateCommandAllocator

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12device-createcommandallocator>

と

CreateCommandList

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12device-createcommandlist>

を使ってオブジェクトを作るだけ。

両方とも知りたいのはコマンドリストの種別…。D3D12_COMMAND_LIST_TYPE が知りたいのです。

https://docs.microsoft.com/ja-jp/windows/desktop/api/d3d12/ne-d3d12-d3d12_command_list_type

前のコマンドキューの時と同様に LIST_TYPE_DIRECT を選ぼう。つまり、ちなみに nodeMask はいつもの 0 でお願いします。

```
_dev->CreateCommandAllocator( D3D12_COMMAND_LIST_TYPE_DIRECT, IID_PPV_ARGS(&_cmdAllocator));  
_dev->CreateCommandList(0,D3D12_COMMAND_LIST_TYPE_DIRECT,_cmdAllocator,nullptr,IID_PPV_ARGS(&_cmdList));
```

さて、これで最低限の準備が整いましたので、画面に影響を与えてみましょうか…

コマンドリストとコマンドアロケータをリセット

まず、命令を出す前にいったんコマンドアロケータとコマンドリストをリセットします。

```
_cmdAllocator->Reset();  
_cmdList->Reset(_cmdAllocator, nullptr);
```

どちらもリセット命令ですね。ちなみにコマンドリストのほうのリセット命令の第二引数ですが、こっちは本来は nullptr ではなく、本来はパイプラインステートオブジェクトが入ります。

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12commandallocator-reset>

Allocator は、まともなメソッドは Reset しかないんやな…

<https://docs.microsoft.com/ja-jp/windows/desktop/api/d3d12/nf-d3d12-id3d12graphicscommandlist-reset>

どちらも HRESULT を返すので、必ず戻り値をチェックしよう。

ちなみに一連の命令を出すときはこの二つを Reset することになります。なお、アロケータ Reset の注意書きに書かれています。

「Unlike [ID3D12GraphicsCommandList::Reset](#), it is not recommended that you call **Reset** on the command allocator while a command list is still being executed.」

↓翻訳↓

「[ID3D12GraphicsCommandList :: Reset](#) とは異なり、コマンドリストがまだ実行されている間は、コマンドアロケータで **Reset** を呼び出すことはお勧めしません。」

うーん？コマンドリストはリセットかけてもいいの？一応コマンドリストのリセットの項目を見ても「明記」はされてないんですが…どっちにしてもリセットする時はちょっと気を付けておいた方がいいだろう。

それよりも気になるのは…

「アプリがリセットを呼び出す前に、コマンドリストは「閉じた」状態でなければなりません。コマンドリストが「クローズ」状態でない場合、リセットは失敗します。」

であるため、原則的にはクローズ処理の後にリセットを呼び出す必要があるということです。

ちなみにコマンドリストを「実行」するのは CommandQueue だから、そいつの実行が終わって(コマンドリスト内部の Close が完了して)からリセットすべきものだろう。

ちなみに DxLib における「命令」は命令を出すと即実行されていたイメージだけどこれは違う。

コマンドリストと言うリストにコマンドを溜めていくイメージで、溜めている間はまだ実行されない。必要な分を溜めた後で CommandQueue の ExecuteCommand を呼び出し順次実行されるイメージだ。

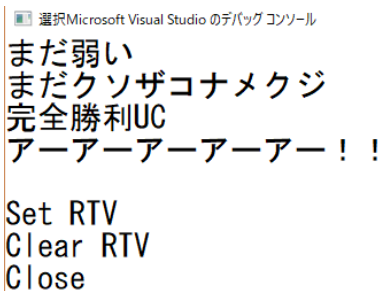
例えばこういうプログラムをコンソールで書いてみてくれ

```
std::vector<std::function<void(void)>> commandList;
commandList.push_back({}() {cout << "Set RTV" << endl; }); //命令1
cout << "まだ弱い" << endl;
commandList.push_back({}() {cout << "Clear RTV" << endl; }); //命令2
cout << "まだクソザコなメダリ" << endl;
commandList.push_back({}() {cout << "Close" << endl; }); //命令3
cout << "完全勝利UC" << endl;
cout << "アーアーアーアー！！" << endl;
cout << endl;
```

//コマンドキューのExecuteCommandみたいなもん

```
for (auto& cmd : commandlist) {  
    cmd();  
}
```

この例だと命令1と2と3がコマンドリストに登録されるが、その場では実行されず最後のループで一気に実行される。



選択Microsoft Visual Studio のデバッグ コンソール

```
まだ弱い  
まだクソザコなメクジ  
完全勝利UC  
アーアーアーアーアー！！  
  
Set RTV  
Clear RTV  
Close
```

こういうイメージでいい。

で、この ExecuteCommand 自体は即時復帰する(ここがちょっと厄介)が、まずは特に気にせずコマンドを投げていこう。

```
auto heapStart=_dscHeap->GetCPUDescriptorHandleForHeapStart();  
float clearColor[] = {1.0f,0.0f,0.0f,1.0f}; //クリアカラー設定  
_cmdAllocator->Reset(); //アロケータリセット  
_cmdList->Reset(_cmdAllocator, nullptr); //コマンドリストリセット  
_cmdList->OMSetRenderTargets(1, &heapStart, false, nullptr); //レンダーターゲット設定  
_cmdList->ClearRenderTargetView(heapStart, clearColor, 0, nullptr); //クリア  
_cmdList->Close(); //コマンドのクローズ
```

コマンド:レンダーターゲットを設定

OMSetRenderTarget というコマンドを使用します。

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12graphicscommandlist-omsetrendertargets>

```
void OMSetRenderTargets(  
    UINT NumRenderTargetDescriptors, //レンダーターゲットビュー数  
    const D3D12_CPU_DESCRIPTOR_HANDLE *pRenderTargetDescriptors, //ハンドル  
    BOOL RTsSingleHandleToDescriptorRange, //ひとまず false  
    const D3D12_CPU_DESCRIPTOR_HANDLE *pDepthStencilDescriptor //今は nullptr でいい  
);
```

ということで、こう

```
_cmdList->OMSetRenderTargets(1, &heapStart, false, nullptr); //レンダーターゲット設定
```

コマンド:レンダーターゲットをクリア

クリアは簡単…

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12graphicscommandlist-clearrendertargetview>

```
void ClearRenderTargetView(  
    D3D12_CPU_DESCRIPTOR_HANDLE RenderTargetView, //レンダーターゲットビュー  
    const FLOAT (4) ColorRGBA, //カラー(0.0~1.0 が4つ)  
    UINT NumRects, //0 でいい  
    const D3D12_RECT *pRects //nullptr でいい  
);
```

ということで…こう

```
_cmdList->ClearRenderTargetView(heapStart, clearColor, 0, nullptr); //クリア
```

コマンド:クローズ

これで最初の発行すべきコマンドはすべてなのでクローズします。

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12graphicscommandlist-close>

これはもう説明の必要ないでしょ。

ということで、あとはコマンドキューに投げるだけです。

コマンドキューに投げる

ExecuteCommandList 関数を呼びます

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12commandqueue-executecommandlists>

```
void ExecuteCommandLists(  
    UINT NumCommandLists, コマンドリスト数  
    ID3D12CommandList * const *ppCommandLists //コマンドリスト配列  
);
```

今回は一個しかないなのでコマンドリスト数は1でいいです。

```
_cmdQue->ExecuteCommandLists(1, cmdLists);
```

あとは Present 関数を呼べばいい

スワップチェーン Present

Present って贈り物の事じゃなくて、レンダリングってイメージをお願いします。レンダリングしてスワップします。ていうかスワップします。

[https://msdn.microsoft.com/ja-jp/library/bb174576\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb174576(v=vs.85).aspx)

ひとまず 0,0 でいいですのでやってみてください。運が良ければ



画面の色が変わります。運が良ければ。運が悪ければ変わりません。

実は色々間違ってるんです

実は色々と間違えてます。なので、運が悪ければ画面の色が変わりません。

間違っている点は

- 常に 0 番目のレンダーターゲットに書かれている
- コマンドキュー実行待ちをしていない

の 2 点です。

本来ダブルバッファリングであるため、常に裏画面に書いていきますが、それはフリップするたびに更新されるはず。つまり『現在の裏画面』番号を取得し、そいつをレンダーターゲットにして、再びフリップ後に裏画面指定を変更してやる必要があります。

SwapChain に GetCurrentBackBufferIndex 関数で裏画面番号を取得し、それを2つのレンダーターゲットのインデックスに指定します。

```
bbIndex = swapChain->GetCurrentBackBufferIndex();
```

で、ポインタのハンドルをコピーしとして ptr を
bbIndex*descriptorSize
だけオフセットしておく。

はい、これで表画面と裏画面が切り替わるようになります。次にフェンスの実装ですが、

フェンス

さて、非常に申し訳ないのですが、画面クリア程度であればフェンスなどの対処が不要と思っていたのですが、画面クリアですら非同期処理に対応しなければならぬのが DirectX12 のようです。

そもそも非同期処理とは？

マルチスレッドの話をしてしまうとかなり難しいので、簡単な話からしていきます。ゲームに限らず特定の処理を行う関数には

- 完了復帰(処理が完了するまでプログラムはストップする)
- 即時復帰(処理が完了してなくてもそのままプログラムカウンタは進む)

の2種類があります。これは裏で別スレッドが走っているんですが、DxLib においても FileRead_open などのはの指定によっては即時復帰と完了復帰が選べます。

http://dxlib.o.oo7.jp/function/dxfunc_other.html#R19N1

完了復帰ならばファイルの読み込みが終わるまではその関数から処理が返ってこないですし、即時復帰ならばファイルの読み込みが終わる終わらないに関わらず処理を返します。

前にも言ったかもしれませんが、ファイルアクセス(つまり HDD へのアクセス)は非常に重い処理で待ちが発生します。ちなみにゲーセン仕様のゲームの場合は1秒以上(60 フレーム以上)の待ちが発生した場合(画面更新を1秒以上行わない場合)は『ウォッチドッグ』という仕組みにより、強制再起動が発生します。

…怖いだろ？マルチスレッドとかなかった時代はファイル読み込みでこういう事が発生しないように相当工夫してたんだよ。

で、マルチスレッドにより非同期処理がデフォルトで入るようになって、読み込み中でも『NowLoading』を表すものを表示できるようになりました。一番秀逸なのは初代バイオハザードのドアが開くシーンです。あれ、ドアを開けている時間で一生懸命ロードしてたわけです。

ちなみに僕の大好きなゲーム『Dead Space』ではエレベータのシーンでレベルロードを行っているっぽいんです。昔のゲームは正面切って『NowLoading』出してましたが、最近のゲームではその時間をごまかすための工夫がより洗練されているようです。

で、ここで非同期ロードには欠かせない概念として『いつロード完了したか』を判断しなければならないわけです。ロードが完了してもいらない不完全なままそのデータを読み取ろうとすればそれはもうね、蛹を羽化前に開いちゃったり、孵化前の有精卵を割っちゃったりするようなもんですよあんな。

というわけできちんと準備できてるかどうか知らなければならないので通常はそのためのAPIなどが用意されている。例えばDxLibのFileRead系であれば

CheckHandleAsyncLoad

http://dxlib.o.oo7.jp/function/dxfunc_other.html#R21N2

などでチェックすることができます。

ちなみにループ内などでチェックしながら完了を待つことを『ポーリング』と言います。ゲームではこのポーリングを使用することが多いです。

[https://ja.wikipedia.org/wiki/%E3%83%9D%E3%83%BC%E3%83%AA%E3%83%B3%E3%82%B0_\(%E6%83%85%E5%A0%B1\)](https://ja.wikipedia.org/wiki/%E3%83%9D%E3%83%BC%E3%83%AA%E3%83%B3%E3%82%B0_(%E6%83%85%E5%A0%B1))

もう一つは完了時にイベント(コールバック関数コール)が発生するパターンです。PlayStation3などではこの方法がとられていました。

あと、非同期処理が顕著なのは『ネットワーク通信』があるゲームですね。結構ネットのデータのやり取りって遅いんです。当然パケットが大きくそして多ければ多

いほど時間がかかりますので、まずは送るパケットを工夫して小さくするところから始まりますが、ともかくここでも完了復帰は普通に使用されます。最後に DB へのアクセスもそうですね。

で、結局 DirectX12 ではどうなの？

ぶっちゃけ GPU と CPU のやり取りなんてのは通信と同じだと思っておいでください。特に DirectX12 においては、例えば GPU にコマンドを投げましたー。で、このコマンドの ExecuteCommand は即時復帰なのよ。

つまり今回の場合であれば画面クリアの実行が完了する前にスクリーンフリップが先に実行されてしまい、まあおかしいことになってしまうわけです。なんですか、というと、ホワイトボードや黒板をイレイサーで消している最中に黒板がフリップされたら困るだろう？



まずはそれを防止しなければなりません。面倒ですけどね。

DirectX にはフェンスという仕組み(D3D12 Fence)があり、それを使用することで GPU に投げた処理を「待つ」ことができます。

ここで

「いやどうせ GPU に投げた処理が完了するまでフリップを待たなきゃいけないんだったら DirectX11 の時みたいに完了復帰にすりゃいいじゃん」と思った君は賢いのだろう。

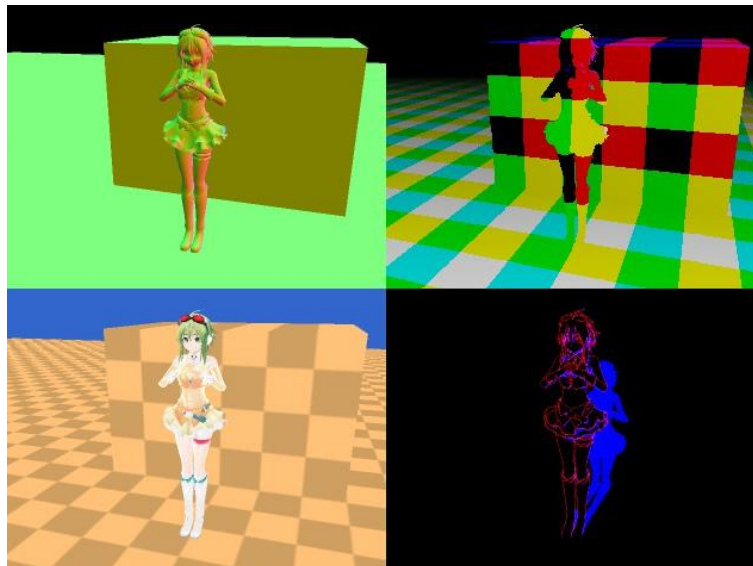


これには理由があるのだ。
 そもそもなんでこんなややこしいことをする羽目になったのかというと

DirectX9~11時代に様々なテクニックが生まれ『マルチパスレンダリング』が当たり前になり、
 ディファードレンダリングなどの手法が方々で使われるようになってきたのが原因じゃないかなと思う。

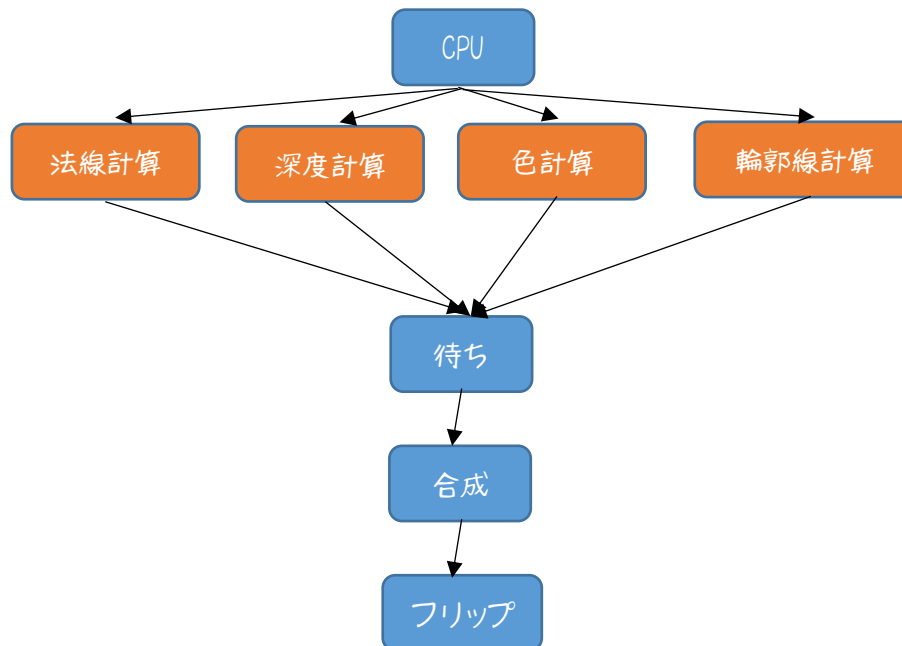
意味が分からないだろうから簡単に言うと。

一枚の画面を作るために



事前に↑の絵のような複数の情報を作っておいて、最後に合成するわけです。

普通に作っちゃうと左上をレンダリングして、右上をレンダリングして、左下をレンダリングして、右下をレンダリングして、最後に合成ってなるんですが、GPU がマルチコアであるにも関わらずシーケンシャル(順次実行)に処理するのは効率悪い



すっげー大雑把に言うところこういう感じにしているわけ。徒競走で4人の走者がいて、運営側は全員がゴールするまで待っておかなければならないみたいな。そういう状態です。

ちなみにこの仕組みに対応できているハードはまだ多くはなく PS4 や XboxOne などに対応していると思いますが GeForce GTX860 以前の PC では対応していないと思います。

フェンスの仕組み

フェンスの仕組み自体はクソ単純です。すぐに理解できると思います。



- 内部に UINT 型の変数を持っている
- GPU 側のコマンド処理が完了した時点で↑の UINT64 型変数を更新する
- CPU 側はこのカウントが更新されたかを見て待つかどうかを決める

ちなみにそれでも分かりづらいかもしれないのが

『GPU 側のコマンド処理が完了した時点で↑の UINT64 型変数を更新する』だけど、これは具体的に言うと

Signal(指定の値)

とやると、GPU 側の処理が完了し次第、フェンス値が指定の値に変更されるので(逆に言うと GPU 側のコマンド処理が完了するまではフェンス値は前の値のまま)、CPU 側としてはこの値を見ながら待つかどうかを決める。

な？クソ簡単じゃろ？

ではフェンスを実装しようか

やることはそれほど大変ではないです。まずはフェンスオブジェクトを作ります。

```
ID3D12Fence* _fence=NULLptr;
```

次に、更新していくためのフェンス値を定義しなければならない。上に書いてるように UINT64 型で定義しよう

```
UINT64 _fenceValue=0;
```

ちなみに GPU が持っている『フェンス値』は CreateFence 時に決定されます。

次にフェンスオブジェクトを生成します。CreateFence を使います。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899179\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899179(v=vs.85).aspx)

```
dev->CreateFence(初期値,DX12_FENCE_FLAG_NONE,IID_PPV_ARGS(いつもの));
```

で、例えばこう

```
dev->CreateFence(_fenceValue,DX12_FENCE_FLAG_NONE,IID_PPV_ARGS(&&_fence));
```

まあ、やろうとしてることは分かるでしょ？

さて、これで ExecureCommand の後あたりで CommandQueue::Signal 関数を呼び出します。

```
_commandQueue->Signal(フェンスオブジェクト,変えたい数値);
```

<https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899171>

で、注意点は、こいつの呼び出し元は Fence ではなく CommandQueue ってこと。自分のコマンドがすべて完了したら自分の中の内部の数値を第二引数の数値に変更します。

例えばこうですね。

```
++_fenceValue;  
_commandQueue->Signal(_fence, _fenceValue);
```

で、ここで注意してほしいことは Signal 関数は「待ってはくれない」という事です。「待つ処理」は自分で作らなければなりません。
ここで待つ方法としては2つあります。ポーリングとイベント通知です。

ポーリング

一番手っ取り早くて分かりやすい方法は「ポーリング」

Signal の後に

```
while(fence->GetCompletedValue() != _fenceValue){  
    //ナニモシマセン('・ω・`)  
}
```

ただねえ…これやっちゃうとぶっちゃけビジー状態になりっぱなしになるので、どうかとは思うけど、ゲームだから CPU 占有しちゃってもいいか。
…まあ、簡単でしょ？

で、もう ExecuteCommand の後に(すぐじゃなくても)待つことになるので、シグナルは飛ばしておきたい。ということで、セットでラップしておきましょう。

void

```
DirectX12::ExecuteCommand() {  
    _cmdQue->ExecuteCommandLists(1, (ID3D12CommandList* const*)&_cmdList);  
    _cmdQue->Signal(_fence, ++_fenceValue);  
}
```

void

```
DirectX12::WaitWithFence() {  
    while (_fence->GetCompletedValue() != _fenceValue)
```

```
        ;  
    }
```

ちなみに今回は分かりやすさ重視のため、ポーリングによる待ちを採用しています。
ちなみにこのやり方だと『完全に待ち』状態になってしまい、せっかく即時復帰になってるのにあまり意味がないですね。

今回はポリゴン表示以外にやる事がないため、ここを待ちにしておりますが、実際には別の作業をやるのが普通です。

まあ、それはさておきイベント通知による実装も紹介しておきます。

イベント通知

これはまあ簡単で、WinAPI によってイベントオブジェクトを作成し、`WaitForSingleObject` 関数を用いてイベントが飛んでくるまで待ち状態をさせるというものです。やってること自体はポーリングと変わらないのですが、どちらを選択するかで全体的な設計思想が変わってきますね。

シグナルを飛ばすところまではポーリングと同じなのですが、まずはイベントオブジェクトを `CreateEvent` を使用して作成します。

```
auto event = CreateEvent(nullptr, false, false, nullptr);
```

これでイベントオブジェクトができますので、フェンスオブジェクトの `SetEventOnCompletion` 関数を呼び出します。これは GPU に対する命令が完了したら設定したイベント通知が走るという物です。

```
_fence->SetEventOnCompletion(_fenceVal, event);
```

で、あとはスレッド用待ち関数 `WaitForSingleObject` でそのイベントを待ちます。

```
WaitForSingleObject(event, INFINITE);
```

終わったら `CloseHandle` 関数でクローズイベントをします。ここでは単純な待ちなので、`WaitForSingleObject` でガチ待ちしていますが、ここもポーリング時と同様に通知が入るまでは他の仕事をさせることができます。

違いは流れの違いだけなので、設計次第でどちらを選ぶか決めましょう。

イベント通知を使用する場合には、こう書き換えられます。

```
if(_fence->GetCompletedValue() != _fenceVal) {  
    auto event = CreateEvent(nullptr, false, false, nullptr);  
    _fence->SetEventOnCompletion(_fenceVal, event);  
    WaitForSingleObject(event, INFINITE); //ここで待ち  
    CloseHandle(event);  
}
```

さて、これで OK かな？ どうか？ まあ、まともには動いているとは思いますが。ただ、水面下でおかしなことになっているかもしれませんので念のため『デバッグレイヤー』を有効にしておいて、エラーが起きていたら通知するようにしてみましょう。

デバッグレイヤーを有効にする

DirectX12 にはデバッグをしやすくするためにデバッグレイヤーという機能がありまあす!! デフォルトではこいつが無効になっているため、オンにします。

何かのオブジェクトを作るとかではなく、スイッチをオンにするだけなので、

```
auto result = D3D12GetDebugInterface(IID_PPV_ARGS(&debugLayer));  
でデバッグレイヤーインターフェイスを取得して...  
debugLayer->EnableDebugLayer();
```

あくまでもデバッグ用なので、リリース時には外れるように #ifdef ディレクティブで場合分けしておいてください。

そうすると出るわ出るわ... エラーが

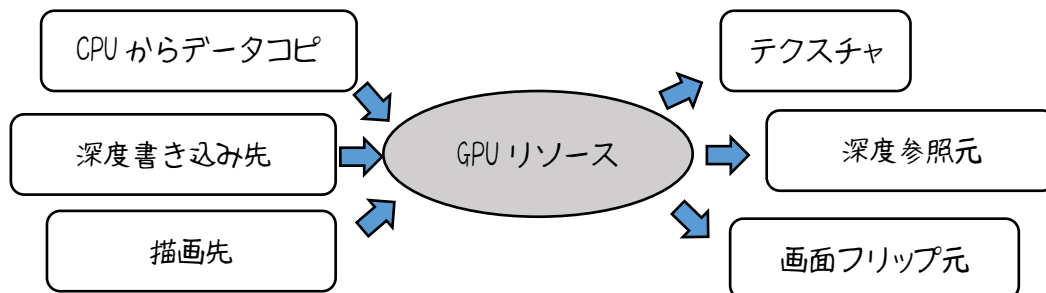
多分確定で出てくるのは リソースのバリア を行わずに、レンダーターゲット書き込みとフリップを行っているところからエラーが発生していると思います。
今はうまくいってるように見えますが、今後問題を引き起こすという事です。

ということで、バリアの話をやっていきます。

リソースバリア

DirectX12 におけるバリアってのは、リソースに対するバリアの事です。この理解がちょっと面倒だなあと感じます。バリアとフェンス…言葉の意味自体が似てるのも嫌な感じだと思っ
んですが…。

DirectX12 におけるリソース(GPU 上のメモリ領域)は様々な使われ方をします。同じリソース
が描画先(レンダーターゲット)になったり、テクスチャになったりします。



で『コマンドを投げた時にリソースが何の用途に使用されるのか』を設定するのが DirectX12 におけるバリアです。

バリアの考え方自体は DirectX12 に限らず『メモリバリア』と呼ばれ、フェンスとともにマルチスレッド用語です。

フェンスは前述の通り、処理の同期をとるための仕組みでしたが、メモリバリアというのは、元々の意味はそのメモリの可視性を保証するというバリアです。

マルチスレッドで特定のメモリが書き換わる可能性がある場合には、誰かがそれを見ようとする際には『見てる最中に値が変わらない』ことを保証する必要があります。

なお、GPU リソースの解釈というのは、それぞれの用途によって変わってきますので、それぞれ用のバリアをしてあげる必要があります。というわけで今回の件では…画面に使用されているバッファは

- レンダーターゲットとしての見方
 - スワップチェーン(フリップされるもの、画面に表示されるもの)としての見方
- がありますので、それぞれに合ったバリアを指定してあげる必要があります。ただこのバリアは…非常に設定が面倒です。

ともかくレンダーターゲットとして使用する部分…

OMSetRenderTarget の直前で「レンダーターゲット用」にリソースバリアを行います。

バリアを設定するには ResourceBarrier という関数を使用しますが

<https://docs.microsoft.com/ja-jp/windows/win32/api/d3d12/nf-d3d12-id3d12graphicscommandlist-resourcebarrier>

この第二引数の設定が面倒ですね。なぜならば

https://docs.microsoft.com/ja-jp/windows/win32/api/d3d12/ns-d3d12-d3d12_resource_barrier

```
struct D3D12_RESOURCE_BARRIER {
    D3D12_RESOURCE_BARRIER_TYPE Type; ///バリア種別
    D3D12_RESOURCE_BARRIER_FLAGS Flags; ///バリアフラグ
    union {
        D3D12_RESOURCE_TRANSITION_BARRIER Transition; //これしか使わん
        D3D12_RESOURCE_ALIASING_BARRIER Aliasing;
        D3D12_RESOURCE_UAV_BARRIER UAV;
    };
};
```

これは DirectX12 プログラミングでよくある union(共用体)です。なにかというと、TYPE の設定によって、そのあとの使用方法の解釈が変わってくるというかなり「C 言語の仕様」を熟知していないと混乱する仕様となっております。もちろん union に関してはご存じかと思いますが、軽く説明しておくとうの Transition, Aliasing, UAV は同じメモリ領域を指し示します。

「そんなことしたら、だめだろ!!! いれい加減にしろ!!!」とお叱りを受けそうですが、まへの Type で使用法を指定してるので、これが食い違ってなければ大丈夫です。はえ〜〜〜そんなだけメモリ節約したいんすねえ〜。

つまり、

今回は Transition しか使わないので、Type を D3D12_RESOURCE_BARRIER_TYPE_TRANSITION にしておけばいいわけです。

で、この Transition バリアはどういう風に指定するのかというと「事前状態」と「事後状態」を指定する必要があります。

例えば今回のバックバッファに関しては、作成の時点で(ていうかスワップチェーンに作られているので)、もともと PRESENT(フリップに使用する)状態として作られているので、レンダータ

ターゲットとして使用するにはステートを RENDER_TARGET にしてあげる必要があります。

なので…

```
D3D12_RESOURCE_BARRIER BarrierDesc = {};  
BarrierDesc.Type = D3D12_RESOURCE_BARRIER_TYPE_TRANSITION;  
BarrierDesc.Flags = D3D12_RESOURCE_BARRIER_FLAG_NONE;  
BarrierDesc.Transition.pResource = _backBuffers(bbIdx);  
BarrierDesc.Transition.Subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES;  
BarrierDesc.Transition.StateBefore = D3D12_RESOURCE_STATE_PRESENT;  
BarrierDesc.Transition.StateAfter = D3D12_RESOURCE_STATE_RENDER_TARGET;  
_cmdList->ResourceBarrier(1, &BarrierDesc);
```

となります。で、レンダリングした後は(といっても今はクリアだけなんですが)ステートをひっくり返して

```
BarrierDesc.Transition.StateBefore = D3D12_RESOURCE_STATE_RENDER_TARGET;  
BarrierDesc.Transition.StateAfter = D3D12_RESOURCE_STATE_PRESENT;  
_cmdList->ResourceBarrier(1, &BarrierDesc);
```

とします。これでデバッグレイヤーでもエラーが消えると思いますが…いかがでしょうか？