

# DirectX12

## ジョクノプログラミング ■



じごくへようこそ

やつふおー!!おひさー!!はじめましての人は初めまして、1年ぶりの人はお久しぶりにござります。DirectX12のお時間がやつてまいりました。

前期でももう死にかけたのに、後期でさらに地獄の責め苦を味わう…そんな目に遭えば



という気持ちになるかもしれない。それは仕方ない。



ぼくは君たちに敬意を表して手は抜かない。全力でお相手いたします。誰かが死にかけていても仕方ない。君たちが隣のお友達をフォローするのは自由だ。だがそれによって遅れても僕はフォローしない。少なくとも授業中はフォローしない。

そんなもので乗り越えられるほど DirectX12 は甘くないのだ。そして俺の信念も固まったのだ。昨年までは「学生の間に DirectX12 なんて教えて大丈夫か?」と自問自答していたし、事実ちょっと適切ではなかったかもしれない。

だが、今まさにグラフィックスプログラミング/パラダイムの時代に突入しつつあります。これはもう「間違ってない!!!!」と言えます。川野先生の暴走はもはや止まらないのです。

# 目次

はじめに .....	5
流れ .....	5
環境設定 .....	11
C++言語のおさらいとか追記とか .....	14
STLについて .....	14
おさらい .....	14
イテレータについて .....	16
リバースイテレータについて .....	16
stringについて .....	17
stringstreamについて .....	18
C++の新しい仕様 .....	19
右辺値参照とムーブセマンティクス .....	20
配列の範囲 .....	25
Emplacement(emplace,emplace_back) .....	25
(おまけ)minmax と iota .....	26
const と constexpr .....	27
まずはプロジェクトを作ろう .....	28
じゃあウインドウ出すか .....	30
アプリケーションのハンドル .....	34
基礎知識説明① .....	37
シェーダ .....	37
頂点シェーダ .....	38
ピクセルシェーダ .....	38
ジオメトリシェーダ .....	39
ハルシェーダ(テセレーション) .....	40
コンピュートシェーダ(GPGPU) .....	41
この辺書いてて思ったこと .....	42
レンダリングパイプラインについて .....	44
DirectX組み込みに入る前に .....	45
DirectX12がそれ以前のDXと違うのはどこ?ここ? .....	46
仮想メモリ(仮想アドレス)とは .....	48
キャッシュメモリとか分岐予測とか .....	50
DirectX12組み込み .....	51
準備①(インクルードとリンク) .....	52

基本の初期化.....	53
画面に影響を与える準備.....	57

# はじめに

最初に言っておかねばならないことがあります。

この授業の主眼は『ゲーム技術の基礎研究』です。残念ながら『ゲーム作り』ではありません。こ↑こ↓注意してください。

えー、じやあ何すんのさ?と思われるかもしれません、先ほども言いましたが基礎研究です。ゲームを作る根本の部分ですね。DXLIB がやってくれていた事(隠ぺいしてくれていた事)が何なのかな…ゲームエンジンがやってくれている事はどういう事なのかな…を知るために今回は DirectX12 を使用して MMD のキャラを動かしてみようと思っています。

半数の学生さんにとっては2度目なので、ああ、またあれか、あれなのがと思っている事でしょう。

とはいっても昨年と同じであれば3年生にこの授業を受けていただけ意味があまりないため、計画としては、去年のやつ+αで『ポストエフェクト』をやろうかと思っております。

やろうと思ってるポストエフェクトは

- 画面にヒビ入れる
- ブルーム
- 被写界深度

です。

まあ、昨年の授業を受けてない人、昨年のテキストを見たこともない人のために流れを言っておくと

## 流れ

1. とにかく DirectX12 ポリゴンを出すまでがんばる(面倒だしシェーダが必要だし即死)
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. MMD 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(めんどう)

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. WIC ローダを作る
12. DDS ローダを作る
13. ポージングさせる
14. アニメーションさせる(リバースイテレータ登場!!!)
15. ベジエで動かす(ニュートン法、二分法)
16. つぶれ影表示(行列で演して黒く塗るだけ)
17. シャドウマップでセルフシャドウ(シャドウアクネがさ…)
18. 簡易トゥーンレンダリング
19. 輪郭線
20. アンチエイリアシング(輪郭線との相性最悪)
21. IK(いけるかな…)
22. ポストエフェクト(をするために必要な事)
23. 色調整(ポストエフェクト)
24. 画面を割る(法線+ポストエフェクト)
25. ブルーム(ガウス+ポストエフェクト)
26. 被写界深度(深度値が重要ですねえ+ポストエフェクト)
27. インスタンシングで大量表示
28. 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…)
29. ディファードレンダリング
30. TBDR(小林先生のご提供となっております)

まあ、これが全部やれるとは俺も思っていない。時間がまるで足りないのだ。昨年よりかはスピーディにできるだろうけど、みんな死ぬでしょうし(笑)

まあシェーダを恐れずやれるようになっておくと、Unity 使おうが UE4 使おうがちょっとかっこいい!事ができてしまうので、シェーダには慣れておいた方がいいと思うよ。

あと、C++の効果的な使い方(?)についても、しつとやっていくので、頑張ってついていく。

で、ここまで説明に一切『ゲームの作り方』に関するものがない事からも『あつ…(察し)』だと思いますが、今期は『授業外でゲームを作ってください』

# 授業外でゲームを作ってください

大事な事ですね…。しんどい?しんどいよなあ…仕方ない。でも、やれ。



うーん。なんでそんなややこしいことを今やるのがと言うと

<https://www.youtube.com/watch?v=H3M07qR0i28>

のカメラ割れとか



の光が漏れている感じとか



のピントが合っている、外れているの感じとか

ゲームエンジンとかライブラリを使用しているとブラックボックスになってて、中身を理解していないと「アーティスト」や「プランナー」の「ああしたい、こうしたい」に対応できないことが多いんですよ。ちなみにゲームエンジンがキャラクターをアニメーションさせてますけど、あれDirectXが勝手にやってくれるんじゃないんですよ？数学とC++を駆使して実装してるんですよ？

僕らはプロです。プロを目指しています。



もちろんです。プロですから。

と胸を張れるようになるためには魔法使いレベルの事ができなきゃいけません。その辺の高校生ができるレベルができても自慢にならないし、その程度だったらなぜここにきて3~4年もやってんのさ。今すぐ仕事しろよ。

『ゲームエンジンの機能にないから実装できません』ではもうそれプロじゃないと思います。

正直ぼくがそういうやつがプロを名乗っていたら



野郎！ ぶっ殺してやる!!

と思います。

とはいって君たちの殆どに欠けているものがある。それは『知識』だ。CGの理論などは今か

ら1からやつていくのはしんどい…しんどいはずだ。

という事で、知識の正確なところはGoogle大先生に任せるとしたら人間は何をすればよいのだろうか？

そう、用語をある程度知つておかねばなるまい…。何故って？用語を知つていればGoogle先生へのお伺いのやり方がスムーズになります。また、そもそも用語を知らないと特定の技術の存在そのものを知らずに過ごしてしまうという事にもなりかねません。

昨今はゲームエンジンで、レンダリング部分やAIにおけるナビメッシュやビヘイビアツリーガブラックボックス化されて見えなくなっているけど、僕らプログラマはその見えない部分も意識しなければならない。ゲームエンジンの中の人が言ってるんだから間違いない。

<https://entry.cgworld.jp/column/post/201701-gameengine.html>

というわけで、UE4 やるにせよ Unity やるにせよ「プログラマ」を生業とするのならば中身をある程度理解しておく必要があるという事です。

さて、授業の大まかな流れですが、パンナムの研修の流れを参考にしてみましょう。まず、パンナムのはこんな感じでした。

- 1<sup>st</sup> week
  - Day 1: Direct3D12 基礎とポリゴン入門
  - Day 2: テクスチャマッピングとリソースバインディング、3D モデル
  - Day 3: 3D モデルのシェーディング、ライティング基礎
  - Day 4: レンダーターゲット、コマンドリスト
  - Day 5, 6: Shadow map
- 2<sup>nd</sup> week
  - Day 7: PBR と発展的なシェーディングテクニック
  - Day 8: Compute Shader
  - Day 9, 10: Deferred Rendering

パンナムの連中にできるんなら、俺たちにもできるはずだよなア？

…嘘です。

というか、大事な大事なアタックチャーンスではなく、大事な大事なスキニングとトゥー

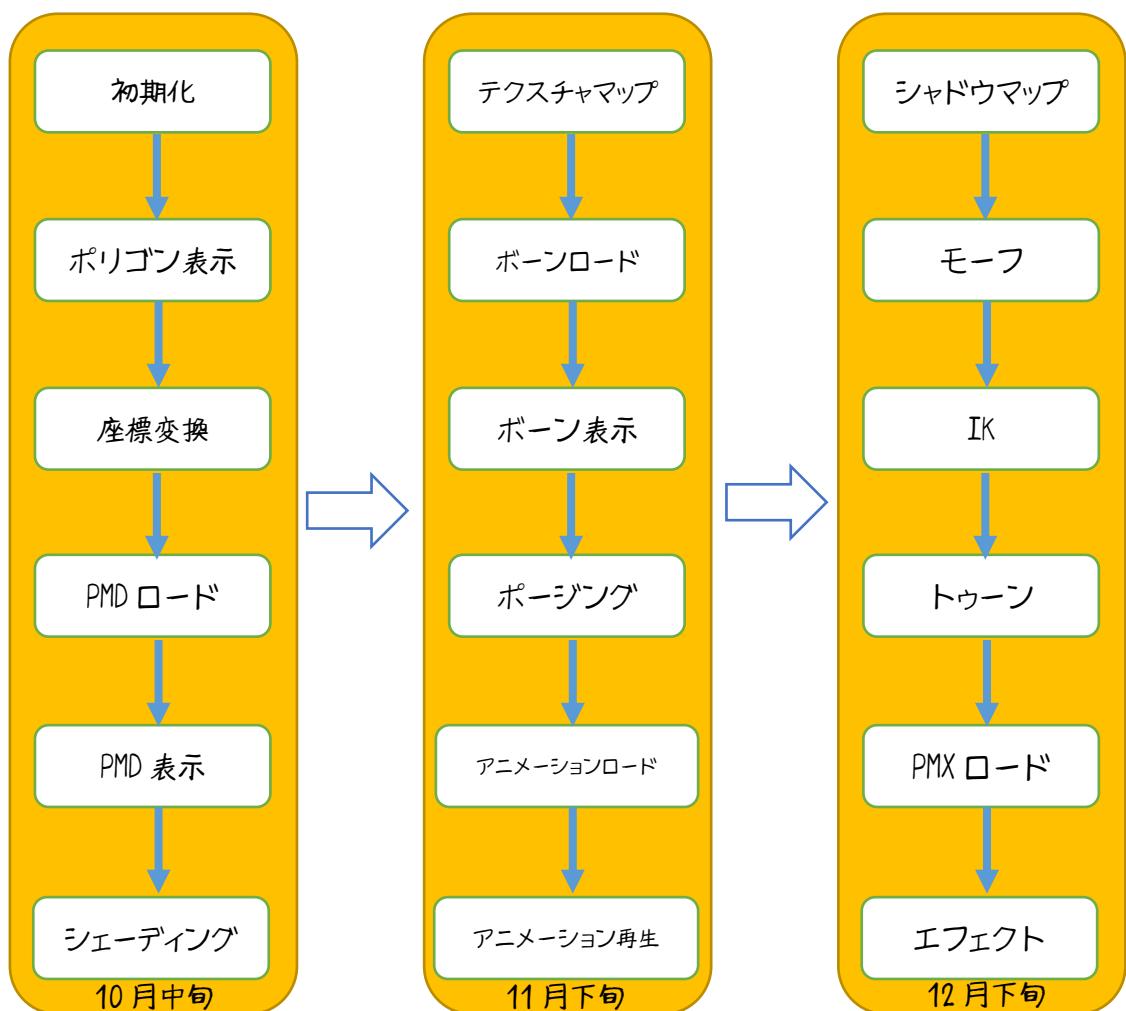
ンとポストエフェクトがないではありませんか!!!

という事で、ComputeShaderとかPBRを後回しにして、その代わりにスキニングとトゥーンを入れたいと思います。

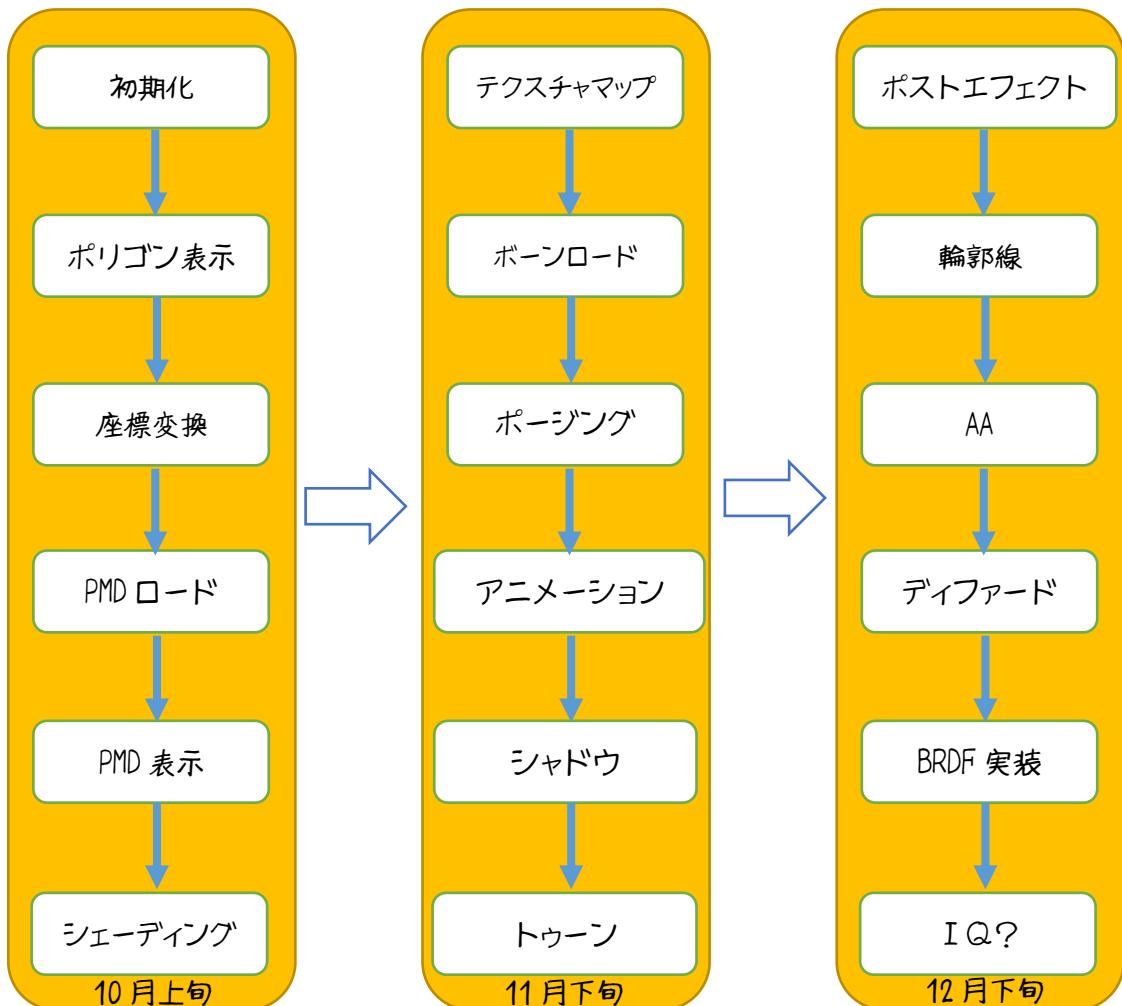
昨年まではやってなかったのですが、DX11からの定番としてディファードレンダリングというのがあるので、今年はそれを入れていきたいと思います。さすがに欲張りセットかな?

あと、昨年に失敗したこととして、僕も余裕がなかったからなんだけど、設計的な事後回しにしてたから、かなりクソコードになってたので、多少の設計はやりながら進めていこうと思います。

昨年まではこう…



今年は



こんな感じで行こうかなと思います。十分に時間ができたら、前期にやった IQ を DX12 で作ってみるというのもいいと思います。

辛いと思います。死ぬと思います。死にます。死にましょう。学期末にアレイズするんで安心してください。勝負はそこからだと言いたいところですが、次年度就職年次の皆さんはゾンビになってでもゲーム作ってもらいます。

## 環境設定

じゃあ早速作り始めよう!!と言いたいところですが、いちど最新の状態で環境設定をしておきたい。何故かと言うと DX12 は Windows SDK のバージョンが変わると同じコードが動かなくなったり、挙動が変わったりするので非常に面倒だが少なくとも WDK は揃えておきたい。

現在のところ WDK の最新版は 10.0.17134.12 である。ともかく WindowsSDK で検索したら WinSDKSetup.exe が落とせるので、落としたらインストールしてください。



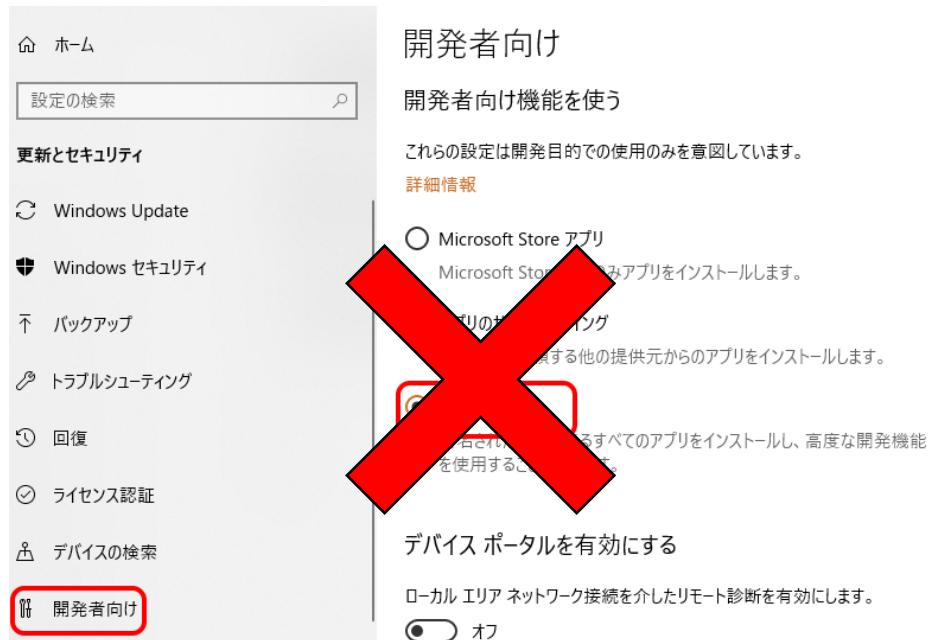
一応学校のサーバーにもインストーラを置いておきますが、インターネットにつながっていないとインストールできないのでご注意ください。

また、学校の外付けHDDにダウンロード済みのを入れてますので、遅い場合はそれを使用してください。

あ、そういえばこのバージョンの面倒くささがあるんで DX12 を活用したものを企業に送る際には SDK バージョンと Windows のバージョンは明記しておいた方がいいかも。

んでも、Windows の最新バージョンが 1803 ではあるんだけど、もしかしたら開発者モードじゃないと 1700 番台までしか更新できないかも…。

開発者モードにするには WindowsUpdate 画面で



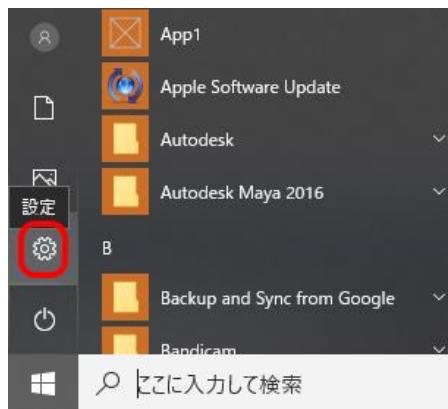
の状態にする必要があるのかなあ…もしくは

<http://www.atmarkit.co.jp/ait/articles/1807/24/news010.html>

に書かれてましたが、WindowUpdate の詳細設定にある延期日数によって出ないこともあります。

どっちにしてもちょっとめんどくさそうな部分(管理者権限が必要な部分)をいじることになりそうなので、最初の方はちょっと手間がかかると思います…とかいろいろとやってみましたが、何故か 1803(4 月に更新されたはずの最新版)に更新されません。もし現段階で 1803 なら更新の必要はありません。

バージョンの見方は

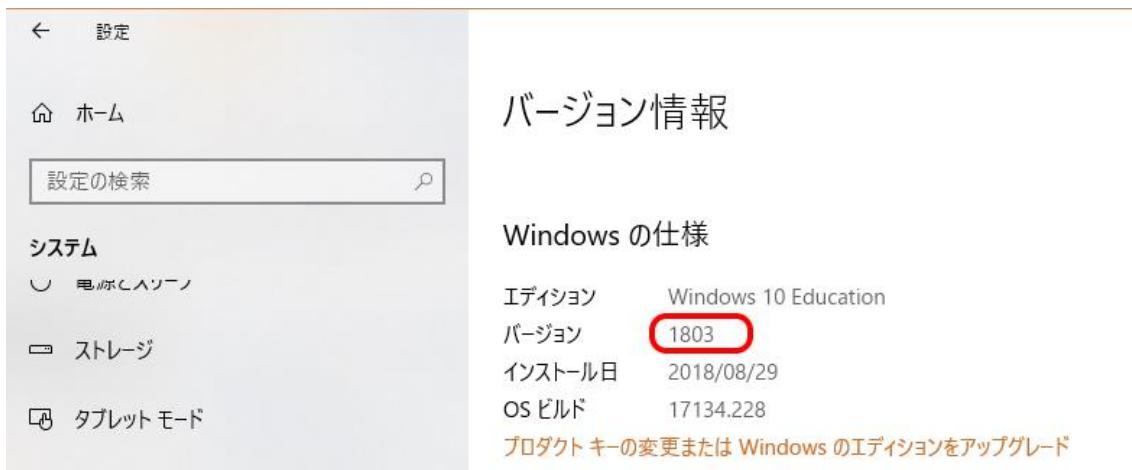


Window ボタン→歯車マーク

Windows の設定



そうすると右側に Windows のバージョンが表示されます。人によっては既に 1803 かもしれませんので、その人は Windows の更新は必要ありません。



そうでないならば直接インストールです。もちろん Windows Update 側に 1803 のインストールが見えてるならば、落とさないに越したことはありません。

<https://www.microsoft.com/ja-jp/software-download/windows10>

これをやるとガチで Windows の更新が始まります。クソ時間かかります。フリーズしどのんちゅうか?っていうくらい…。

無事終わったら Windows のバージョンが 1803 になっているはずですので、確認しておいてください。何度も再起動かかるんで待ちましょう。

さて、ここに手間取っている間に、昨年のテキストでも見ながら「予習」しておいてほしい。

## C++言語のおさらいとか追記とか

STLについて

おさらい

前回から当然のように STL を使っていると思いますが、とりあえず vector と map と list と set の区別はついているでしょうか?あと string に関してはな…。

コンテナ名	概要
vector	動的配列として使用できる。ガチでメモリが連続しているので、様々な用途に使える。 ただし、あまり push_back してると動的確保が頻繁に行われるため速度低下の原因となる。その場合は予め予測した大きさを reserve する。 また、要素が増えてくると配列同様に要素の挿入、削除の時間コスト

	が高いけれど、殆どの場合は気にならない(それ以外のメリットの方が大きい)
map	連想配列として使用できる。インデックスではなく文字列を使用したり、飛び飛びのインデックスの配列としても使える。 また、map の要素は結局のところ 2 つ値<Key, Value>のペアに過ぎないため、そう捉えると様々な応用が可能。
list	名前の通りリスト構造でできているコンテナ。vector と違って、メモリが連続していない。要素と要素がリンクによってのみ繋がっているため、挿入と削除のコストが一定。 ただし、検索のコストは、要素が増えるほどに増えていく。大抵の場合は vector を使っておいた方が幸せである。
set	要素がソート済みとなるコンテナ。まあほとんどの場合 vector や map で事足りる。

つまるところやっぱり vector と map で十分って事やな!!

また、前回もちょっとだけ出てきましたが、algorithm などを使うとコードの量をぐっと減らせます。

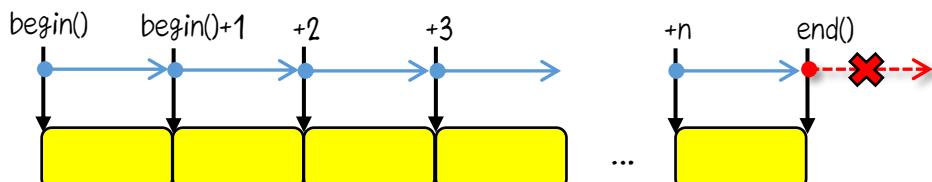
関数名	概要
for_each	指定された範囲内の要素に対して、特定の処理を行う
find	指定された値を検索しイテレータを返す
find_if	条件に合う要素を検索し、イテレータを返す
find_first_of	条件に合う要素の中で最初に見つけたイテレータを返す
find_last_of	条件に合う要素の中で最後に見つけたイテレータを返す(いらんかも)
min_element	要素の中から最も小さい要素を持つイテレータを返す
max_element	要素の中から最も大きい要素を持つイテレータを返す
sort	要素をソート
count	指定した値を持つ要素数を返す
count_if	条件に合致する要素数を返す
all_of	全てが条件を満たせば true
none_of	全てが条件を満たさなければ true
any_of	一つでも条件を満たせば true
fill	指定した範囲内に指定した値を代入。memset みたいになもん。
copy	指定した範囲内に、別の指定した範囲をコピー
copy_if	指定した条件を満たすもののみコピー

generate	指定した範囲に対して特定の関数を適用
transform	指定した範囲に対して特定の関数を適用した値を別のイテレータに代入(generateと似てるがちょっと違う)
remove	指定した範囲の要素を取り除く。実は削除されてないので注意。eraseとの組み合わせで本当に削除される。
remove_if	指定した条件に合致する要素を取り除く。↑と同様に削除されてないので注意。
replace	指定した値 A を別の指定した値 B に書き換える
replace_if	条件に合致する要素の値を、値 B に書き換える
unique	重複した要素を取り除く。実は削除されてない(略)
sample	指定された範囲内からランダムに要素をいくつか抽出する
shuffle	要素をミッドナイトシャッフルする

### イテレータについて

STL の vector などの map ってのは「コンテナ」と呼ばれて、色々なルールで要素の集合を格納するものです。そこは大丈夫だと思いますが、要素へのアクセスには色々とやり方があるんですよ。

で、一番基本的なアクセス方法が「イテレータ」と言うものを介すものです。今までしつつ何度も出て来てたんですが、基本的には begin() だの end() だので得ることができます。こいつはポインタのようなもんで、その要素のアドレス先頭を指示していると思ってく



ださい。

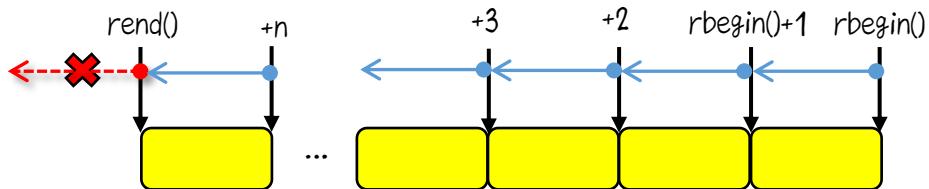
ここまでではいいかな?

そして、\*などで「値」にアクセスする場合はこのイテレータから先に向かう方向にデータを取ってくるイメージなのだ。なので↑の図のように end() はその先にデータがないため、end() に対して値アクセス要求を出すとクラッシュ(というよりアサート)するわけ。

大丈夫? この辺非常に大事なのでしっかりしてね? そしてこのイテレータには変な親戚がいるのだ。それが「リバースイテレータ」というやつ。

### リバースイテレータについて

リバースという名前から想像がつくと思いますが、逆方向イテレータです。begin() とか rend() から取得します。データとの関係を図にするとこんな感じ。



ただ、逆方向に向いているだけのイテレータだが、これが「スキンマッシュアニメーション」の時とかに意外と使えるのだ。なんとなく頭には入れておこう。

### stringについて

`string`は何度も使ってるからもう使う事に対して苦手意識は持っていないかなと思います。`string`は結局のところ `vector<char>`に文字列用のメソッドをいくつか追加しただけのものです。

文字列用のメソッドとは

関数名	概要
<code>+オペレータ</code>	文字列連結
<code>==オペレータ</code>	同じ文字列か比較
<code>c_str()</code>	C言語の文字列表現(つまり文字列ポインタ)を返す
<code>length()</code>	文字列の長さを返す
<code>substr()</code>	部分文字列を返す

こういう関数が独自にあります。それ以外にもいろいろあつたりするので、これ以外は自分で調べましょう。

ちなみに `string`についてですが、文字列をチョットどうこうしようと思った人ならこう考えるんじゃないだろうか?

そういうえば `string`ってのは `char` の集合体だよね? 文字って、1バイト文字だけじゃなくて「マルチバイト文字」ってあったよね? あれも `string`を使うの?

と、考えた人は良いところに目を付けていると思います。

実は `string`ってのは、`string`単品で宣言されているわけではなく、見てないところでこう宣言されています。

```
using string    = basic_string<char>; // シングルバイト文字
using wstring   = basic_string<wchar_t>; // ワイド文字
using u16string = basic_string<char16_t>; // UTF16文字
using u32string = basic_string<char32_t>; // UTF32文字
```

勘のいい人はお判りでしょうが、ワイド文字を扱う場合は `wstring` を使用します。当然ながら `string` と `wstring` には型の互換性がないため、もし変換をする際にはかなり面倒な処理が必要になります。

`MultiByteToWideChar` とか `WideCharToMultiByte` とかそういうのを介す必要があります。今の所は本題ではありませんので、飛ばしますが、文字列いじり始めると色々とややこしい事は心に留めておいてください。

### stringstreamについて

そういえば C 言語の時は、数字をフォーマットして文字に変換する関数として `sprintf` とかあったけど、`string` でそれに相当するものはないの？ `string` は連結と部分抽出しかできないの？

まあ `string` はそうなんだけどね。モチロンその辺の対応がない C++ ではない。

`stringstream`…つまり文字列ストリームと言うのがある。

例えば `cout` を使う際に

```
cout << "Hello World!" << endl;
```

なんて書くと標準出力に対して `HelloWorld` と出力できるだろう？ これの文字列版があるのだ。

まず

```
#include<sstream>
```

で使う準備だ。

次に文字列ストリーム用のオブジェクトを用意する。

```
ostringstream ss;
```

あとは `cout` の時と同じ要領で数字などをぶっこんでいく

```
ss << "Age=" << 42 << ", Height=" << 160;
```

などと書けば `ss` の中に "Age=42, Height=160" という文字列ストリームができている。確認した

ければ

```
cout << ss.str() << endl;
```

とでも書けばいい。なお str()ってのは文字列ストリームを文字列に変換する関数だ。

ちなみに 16進数とかにしたければ hex を書くことによってそれ以降が 16進数になる

```
ss << hex << "Age=" << 42 << ", Height=" << 160;
```

と書けば

```
"Age=2a,Height=a0"
```

という文字列が得られる。

ちなみに hex は 16進数、dec は 10進数、oct は 8進数である。

また桁数揃え等をしたければ setw を使用します。

0埋め等をしたければ setfill を使用します。

例えば

```
array<ostringstream,5> sss;
for (int i = 0; i < sss.size(); ++i) {
    sss[i] << "Texture_" << setw(3) << setfill('0') << i;
}
for (auto& ss : sss) {
    cout << ss.str() << endl;
}
```

とでも書けば

```
Texture_000
```

```
Texture_001
```

```
Texture_002
```

```
Texture_003
```

```
Texture_004
```

という出力が得られます。STLに関してはこんなもんですね…。

## C++の新しい仕様

前期の授業で C++の新しい仕様として、

auto, nullptr, 範囲 for 文, enum class ラムダ式

仕様	概要
auto 変数名	右辺値から型を推測して決定
nullptr	NULLとかタッセー奴じゃなくてちゃんとしたヌルオブジェクト
範囲 for 文	インデックスを指定しなくても、全要素のループを記述できる
enum class	先頭に型名を付加することで、従来の enum のような名前重複を防止
ラムダ式	{}でお手軽に関数オブジェクトを作れます

を紹介したわけなんですが…実は代表的で分かりやすい一部しか伝えてないんですね。

非常にありがたい資料があったので紹介しますが

<https://www.slideshare.net/Reputeless/c11c14>

をちょっと読んどいてください。157 ページと、川野先生に負けず劣らずボリューム多しですが、さっと目を通しておくといいと思います。

見る限り、意外と思ったのが代入におけるメモリコピーの無駄をなくす仕様に偏っており、逆に C++ らしくなったよなという印象です。

新しい仕様で知つておいた方がいいのは

- 右辺値参照とムーブセマンティクス
- 配列の範囲
- Emplacement
- minmax, iota

あと、なんとか constexpr がないんでそれも本当は追加したい。

まあ一番ややこしい奴から行きましょうか

## 右辺値参照とムーブセマンティクス

実はムーブセマンティクス自体は、前期の unique\_ptr の時に一度だけ出てきているんですね。std::move を使用して所有権の移動を行ってところで。

で、右辺値参照の話なんですが

代入の際の

左辺値 = 右辺値

の図式で考えちゃうとたぶん理解できないし誤解する。

## とりあえず右辺値が変数ではなく一時オブジェクトの場

合に限って 考えてほしい。ちょっと限定的な状況の話だ。

で、今回の限定的な状況の話においてはプログラマが意識してプログラムの方法を変えると  
かそういう事じゃなくて、しつつ C++のメモリ部分の仕様変更が行われてメモリ周りが効率化されたと考えてほしい。

だから僕らが頭を悩ます必要がなくて、逆に昔一生懸命効率化しようとしてた部分が不要になつたと思って良い。

ひとまず、右辺値も左辺値も変数である場合を見てみよう。

```
std::vector<int> lv;
std::vector<int> rv={0,1,2,3,4};
lv = rv;
for (auto v : lv) {
    cout << v << endl;
}
```

これは当然のように全値のコピーが発生します。これはいい。意図したとおりだから。しかし、もし以下のようないふり

```
lv = std::vector<int>(100, 0); //右辺値を破棄してもいいんだからコスト無駄じゃね?
for (auto v : lv) {
    cout << v << endl;
}
```

この場合も古い C++なら一時オブジェクトを生成して、さらにデータコピーが発生していたのだ。ところが新しい C++の場合ならば『所有権の移動』が行われコピーコストが発生しない。つまり、どういうことかと言うとコピーするまでもなく左辺に直接値が入るイメージである。

ここでちょっとした疑問が湧く。2つだ。

- 本当にコピーが発生していないのか？
- 所有権の移動と言うのは一時オブジェクトに参照が変わる事なのか？

いや、仕様なんやから信用しようや。確かにそうなんだけど気持ち悪いのだ。分かんないかな

あ…こういう気持ち。

まあ、そもそも皆さんとしても、検証もしないでっていうのは納得いけないでしょ？という事で検証

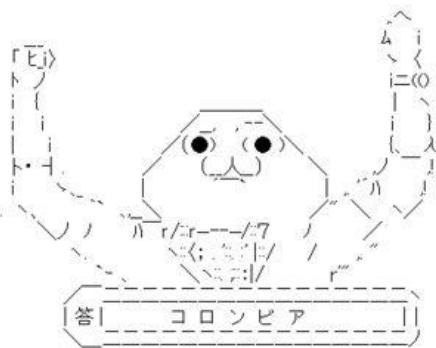
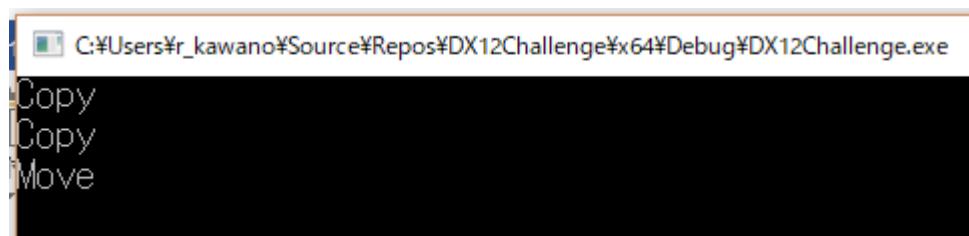
まずこういうクラスを作る。

```
struct Boo {  
    Boo& operator=(const Boo& b) {//コピー代入オペレータ  
        cout << "Copy" << endl;  
        return *this;  
    }  
    Boo& operator=(Boo&& b) {//ムーブ代入オペレータ  
        cout << "Move" << endl;  
        return *this;  
    }  
};
```

検証用なので適当である。ともかく重要なのはどちらが呼ばれるかである。

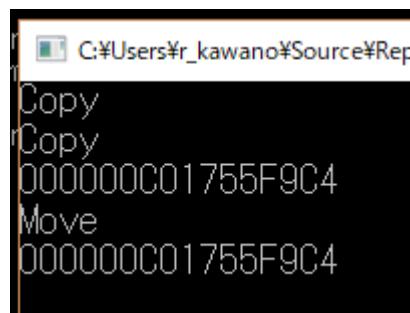
```
Boo a, b,c;  
b = c;  
a = b;  
a = Boo();
```

こういうコードを書いて結果を予想する。もし右辺値参照が言ってる通りの仕様であるならば Copy Copy Move と表示されるはずだ。実行!!



という事で、信用していいわけだ。おっともう一つ検証しなければならない。もし一時オブジェクトの所有権が左辺値に移るという事が、左辺値の参照先が一時オブジェクトを指し示すという事ならばアドレスが変わっているはず。逆に左辺値の値を直接書き換えていた状態ならアドレスは変わらない。さあどちらだ!!!

```
cout << hex << &a << endl;
a = Boo();
cout << hex << &a << endl;
```



The screenshot shows a terminal window with the following output:

```
C:\Users\r_kawano\Source\Rep
Copy
Copy
000000C01755F9C4
Move
000000C01755F9C4
```

オッケー!!安心してこの仕様に乗つかろう!!

ちなみにしれっと書いたけど

```
Boo& operator=(Boo&& b)
```

これがムーブオペレータである。ちなみに関数の戻り値として使用する場合だが

```
Boo GetBoo() {
    Boo b;
    return b;
}
```

(中略)

```
a=GetBoo();
```

とか書くとどうなんだろう?



アツハイ

ムーブしか発生してませんね。なるほどなるほど。

ちなみにこの右辺値参照を強制(つまりコピーを禁止)するにはどうするかというと

型名`&&` 变数名;  
の宣言を行う。

`Boo&& boo=GetBoo();`

であるとか

`Boo&& Foo = Boo();`

のように書くと右辺値は一時オブジェクトである事を強要される。つまり`&&`がつけられた左辺値には変数を右辺値に取ることはできない。

つまり

`Boo a, b, c;`

`Boo qoo = a; //OK`

`Boo& poo = a; //OK`

`Boo&& woo = a; //NG`

となる。あああああああああああややこしい。で、最後の行がNGになっているんだけど  
`std::move` を使えば強制的に所有権が移動し、OK 牧場となる。

`Boo&& hoo = std::move(a); //OK`

さて、この場合は流石に所有権の移動が行われているだろう。つまり

`cout << hex << &a << endl;`

`Boo&& hoo = std::move(a); //OK`

`cout << &hoo << endl;`

の結果は



A screenshot of a terminal window titled 'Terminal' showing two lines of memory addresses. The first line shows '0000005F6D8FF874' and the second line shows '0000005F6D8FF874'. The second line is highlighted with a yellow background.

となる。完全に所有者が`hoo`になってしまっている。では元の`a`はどうなっているのかという  
と

`cout << hex << &a << endl;`

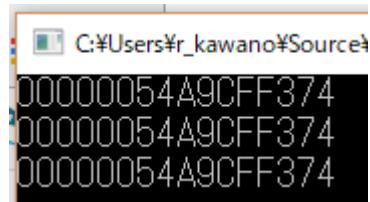
`Boo&& hoo = std::move(a); //OK`

`cout << &hoo << endl;`

`cout << &a << endl;`

と書いたところ

ということで、まだ`a`も所有権を持っていそうだが、一応仕様上は所有権がないということな



```
C:\Users\r_kawano\Source\repos\test>00000054A9CFF374
00000054A9CFF374
00000054A9CFF374
```

ので、どうなっても知らんよという事。つまり move してしまったら中身を使えると思うなという事。

もうガチでややこしい仕様やったわ…。ちょっと横道に逸れるだけのつもりやったのにガチ説明したわ。

## 配列の範囲

これはあれやな。配列の場所をイテレータとして使えるんやけど前まではポインタとそれ+範囲の先という指定をしておった。

つまり

```
int a[] = { 1,3,5,7,9,11 };
std::for_each(a, a + _countof(a), [](auto v) {cout << v << endl; });
```

こう書いていたのを

```
int a[] = { 1,3,5,7,9,11 };
std::for_each(begin(a), end(a), [](auto v) {cout << v << endl; });
```

こう書けるようになった。

いや~、でかい、でかいよこれは。ありがたい。

## Emplacement(emplace,emplace\_back)

例えばこのようなクラスを作る。

```
struct Vector3 {
    Vector3() {};
    Vector3(float inx, float iny, float inz):x(inx),y(iny),z(inz) {}
    float x, y, z;
};
```

こいつのベクタを作る。

```
vector<Vector3> vertices;
```

で、こいつに push\_back したいとする。但し、push\_back の引数は Vector 型であるため、  
vertices.push\_back(Vector3(1, 2, 3));

とする必要がある。

が、emplace\_back を使用すれば、Vector3 の一時オブジェクトを使う必要がない。

```
vector<Vector3> vertices;
```

```
vertices.push_back(Vector3(1, 2, 3)); // 一時オブジェクト生成 & コピー
```

```
vertices.emplace_back(4, 5, 6); // 直接生成 & 値の設定
```

```
vertices.emplace_back(7, 8, 9); // 直接生成 & 値の設定
```

```
std::for_each(vertices.begin(), vertices.end(), [](auto v) { cout << v.x  
<<, << v.y <<, << v.z << endl; });
```

ご覧のように emplace\_back の方がコード量も若干少なくなりますし、一時オブジェクトも作られないないので、場合によってはメモリの効率化にもつながります。若干だと思いますが。

### (おまけ)minmax と iota

最後はオマケみたいにはもんやな…。まずは minmax から

<https://cppref.jp.github.io/reference/algorithm/minmax.html>

『同じ型の 2 つの値、もしくは initializer\_list による N 個の値のうち、最小値と最大値の組を取得する。』

最後の引数 comp は、2 項の述語関数オブジェクトであり、これを使用して比較演算をカスタマイズすることができる。

例えば

```
auto mm=minmax({ 0,1,2,3,4 ,8,2,-6,-2,3,100,120,-12 });
```

```
cout << mm.first << "<=value<=" << mm.second << endl;
```

なんて実行すると

```
C:\Users\r_kawano\Source  
-12<=value<=120
```

なるほど

じゃあこれは…？

```
std::vector<int> rv = { 0,1,2,3,4 ,8,2,-6,-2,3,100,120,-12};
```

```
auto mm=minmax(rv.begin(), rv.end());
```

これはダメなんです。

begin の大きさと end の大きさを比べてしまうので、予想したような結果になりません。  
minmax はあくまで二つの値もしくは initializer\_list の大きい方と小さい方を返すだけっぽいです…惜しいなあ。

次に iota ですが、これは要素を連番で埋めるというものです。

```
#include<numeric>
```

で使います。

```
std::vector<int> rv = { 0,1,2,3,4 ,8,2,-6,-2,3,100,120,-12};  
std::iota(rv.begin(), rv.end(), 0);  
for_each(rv.begin(), rv.end(), [](auto v) {cout << v << endl; });
```

とやると

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12
```

となります。

役に立つかなあ…。

ちょっと minmax と iota は微妙やつたかも。

### const と constexpr

もう一つ新しい仕様として const のもっと厳密な奴と言うかコンパイル時 const にあたる constexpr と言うやつが追加されている。

もともと C 言語の時に

```
#define RIGHT_VALUE 16
```

てな感じで定数を定義していたやつを

```
const int RIGHT_VALUE=16;
```

って書いてたんだよね。間違ってないんだけど、今まで const 使ってたから分かるでしょ？

所詮 const ってそのスコープの中で書き換えが発生しないって事やから実行時に右辺値が分かつてなくても OK なんよね。それはそれでいいし、使える仕様なんですが define の代わりに

使う意味の const としては弱くなつたんよね。

それで出てきたのが constexpr です。

コンパイル時に右辺値が決定できないとエラーを吐くわけ。だからマジックナンバー回避のための定数などには const ではなく constexpr を使うのが最近の流れです。

だから

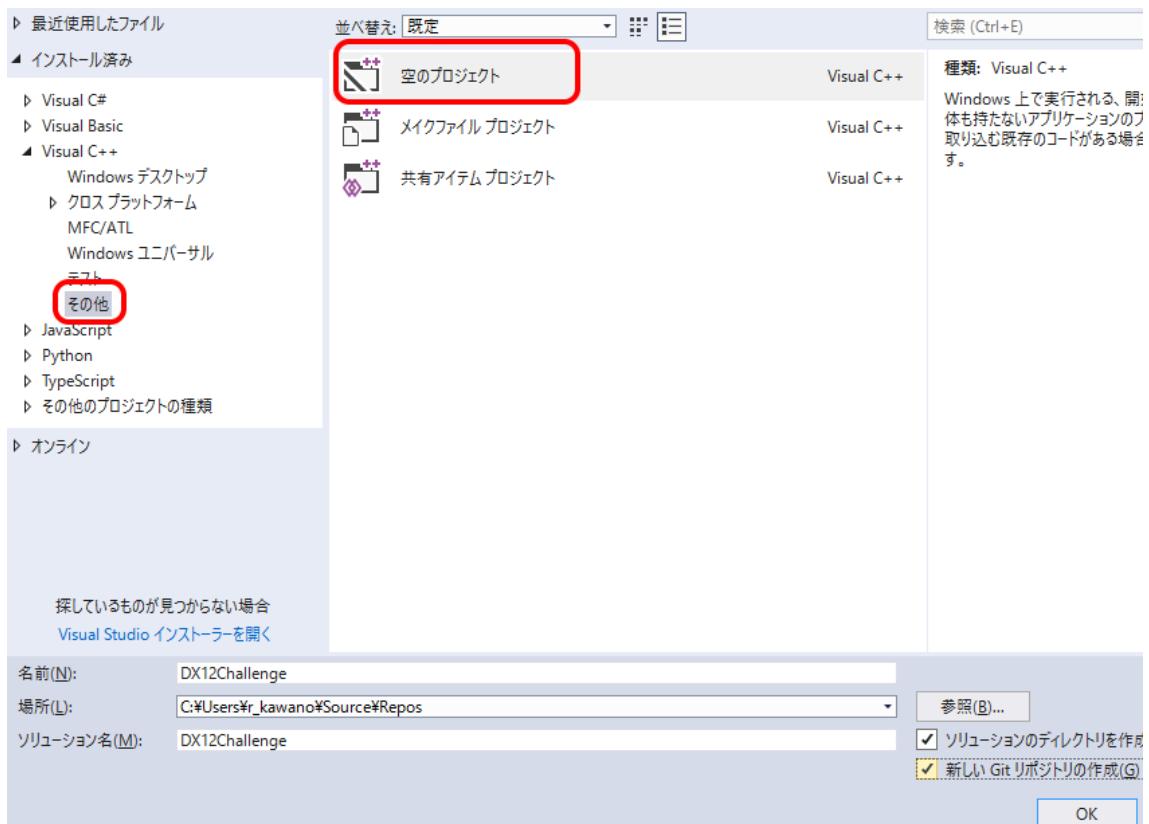
```
constexpr int ppp = Get(); // NG
```

```
const int qqq = Get(); // OK
```

というわけですね。

とりあえず新しい仕様としてはこんなもんかな。なんでこれ話してきたかと言うとたぶんこの先、僕がしつこく新しい仕様に沿ったコード書いてみんな混乱するかもしねへんので最初に言っておきました。

## まずはプロジェクトを作ろう



空のプロジェクトを作るとこからですね。

で、メイン関数を作るのでですが、mainでもWinMainでもどっちでもいいです。mainを出すとコンソール画面が出てくるくらいの違いしかないです(他にはHINSTANCE hInstでアプリハンドルを取ってこれるくらいしか違ひがない)。

ぼくはエラーが出しやすいという理由でコマンドラインの方を使います。

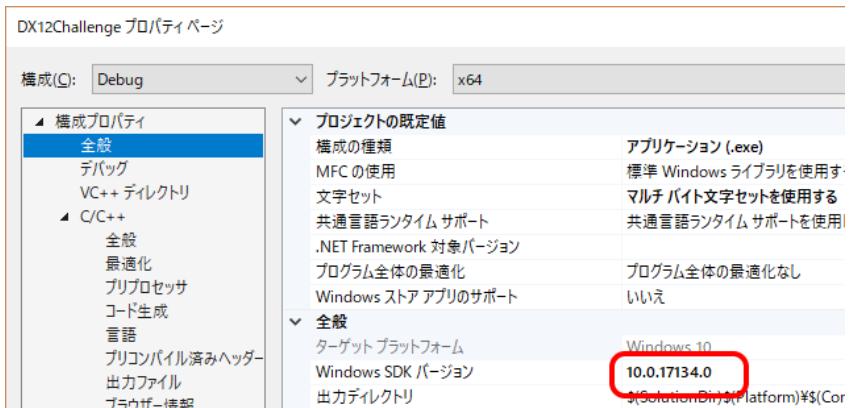
```
int main() { //①…コマンドラインありの時
    int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, int){ //②…コマンドラインなしの時
        cout << "Fuck You" << endl;
        getchar();
        return 0;
    }
}
```

別にどちらでも構いません。あ、Windows アプリケーションなので

```
#include<Windows.h>
```

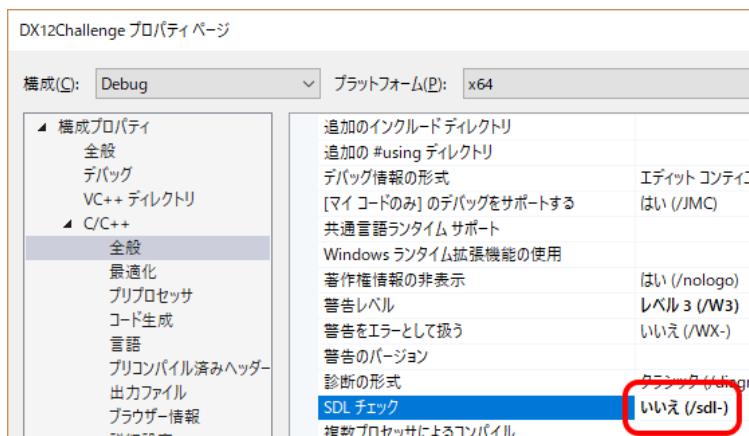
をインクルードはしておいてください。あ、ちなみにメイン関数がある cpp は main.cpp とします。ただただ main を実行するのみの関数ですね。

で、プロジェクトの設定に入りますが

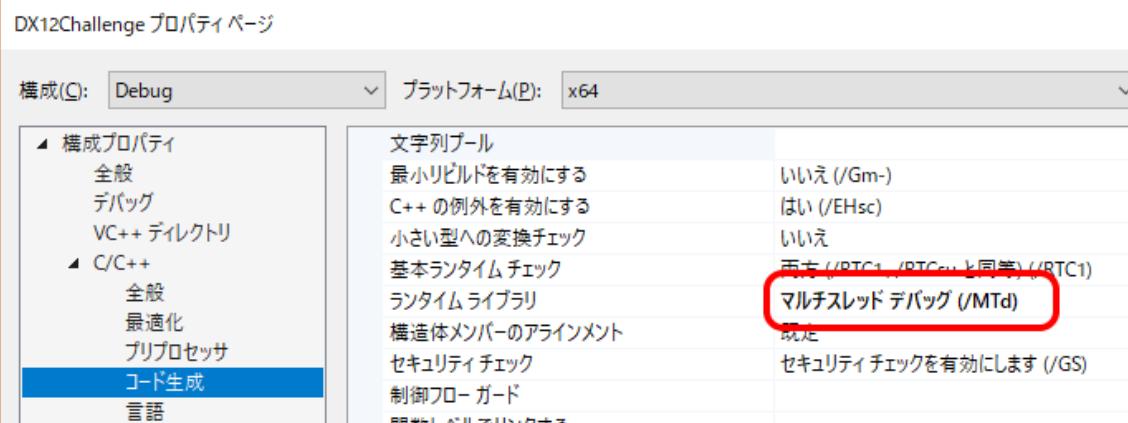


WindowsSDK が最新なのを確認してください。

次に C++ の全般で SDL チェックをいいえにします。



これしてないと、C言語標準のメモリ処理の関数やら文字列処理の関数やら出てきたときに\_sを使えとかローカルルール押し付けられてうざいので、こうしておきます。はい、次にちょっと面倒な部分ですが、コード生成を「マルチスレッドデバッグ」に書き換えます。



とりあえずウィンドウ出すまでならここまでで十分。

## じゃあウィンドウ出すか

ここから既に DxLib とは違いますので頑張りましょう。クソみたいに面倒くさいです。でも、細かく解説しません。ウィンドウ生成について細かく知りたい人は「猫でも分かるプログラミング」でも読んでください。

あまり main.cpp は汚したくないので Application クラス作りましょう。シングルトンで作っときましょうか。

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つておきます。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」ってのが出づります。

HINSTANCEとかHWNDとか

一応 Windows とか DirectX 界隈では当然のように Handled-Body / パターン的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だ。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。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 側のグローバル的な領域(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、闊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 番はやっぱり生理的にイヤ。

とりあえず Windows のウィンドウを作るのに、DxLib の時は DxLib\_Init で済んでたんだけど(ホ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る)、ウィンドウを「作る」だけで

```
WNDCLASSEX w = {};
w.cbSize = sizeof(WNDCLASSEX); // これ、何のために設定するのさ…?
w.lpfWndProc = (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)
```

```
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/プラメータ
```

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だわ。

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*)mssageBuffer);
cout << (TCHAR*)mssageBuffer << endl;
LocalFree(mssageBuffer);
}
```

のコードも追加しておいた方がいいね。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処理なので、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 に入るのはやめておこう。どっちかというと Run に入れたいい。

次に DxLib 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られるイメージで。

```
if (PeekMessage(&msg, NULL, 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); // 处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。言いたいところやけど、ひとつ忘れとったわ…いつも忘れる。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やなくて、通常の関数として宣言して置いてください。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。

ちなみに軽く解説しておくと…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 におけるグラフィッ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と…

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周りを制御するのは OS なんですよ。モバイル機器でも同様なんんですけど、OS ってアホほど色々やってるんですね。

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、きちんと読まないと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。マニュアル読み！ハードやライブラリの仕様読み!!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)
- 分からなかつたらすぐ聞く(先生でも友人でもいいので、分らないままにしない事)
- 休まないよう(基本的に、休むとワケ分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。寝りや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りないようにしてください。寝ててついで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し。
- ↑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 にウイ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、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。

# 基礎知識説明①

## シェーダ

シェーダ、シェーダ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。

シェーダの種類は現在の所

- 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:ハルシェーダ(テセレーションシェーダ)
  - 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りますね。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があります。

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。

初步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。

## ピクセルシェーダ

ピクセルシェーダ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ダです。頂点シェーダで変換された頂点情報を「ラスタライザ」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ダが呼ばれます。

つまり、長方形ペラポリをウインドウいっぱいに 1 枚描画したとするとその解像度分のピクセルシェーダが実行されます。例えば 1280×720 のウインドウであれば 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ダ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(超えてるとシェーダ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ダで行いますが、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という安っぽいテクノロジーがありました。

ピクセルシェーダの基本的な役割は  
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。

## ジオメトリシェーダ

さて、ジオメトリシェーダですが、こいつ、何なんすかねえ？  
頂点とピクセルは分かつた。ではジオメトリシェーダとは何なのだ？ジオメトリとは幾何学と言う意味だ。

言うてしまうと、ジオメトリシェーダによって新たな頂点を作ることができます。  
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で「ボリュームシャドウ」などを作ることもできます。



ただ、ボリュームシャドウは最近あまり聞かないるので、たぶん実用的じやないんじやないかなあと思ってたりします。

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ダだったりします。

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～。

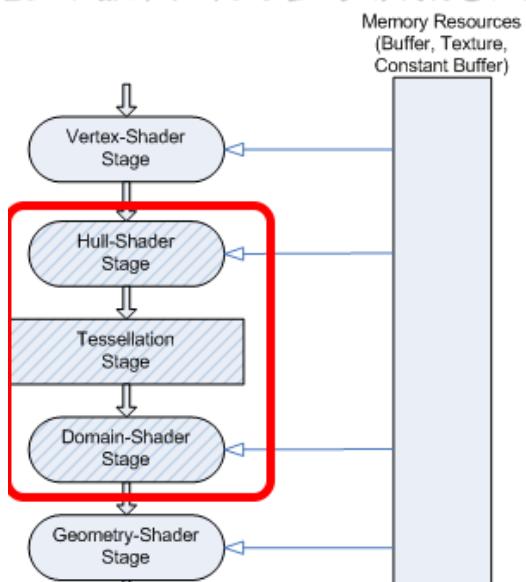
とりあえずそういうのがあって、なんか活用できそうなアイデアがあったら使いたいと思ひます。

## ハルシェーダ(テセレーション)

次にハルシェーダです。ハルシェーダの話をするまえに「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。いわゆるサブディフ的な奴ですね。OpenSubdivだったので活用されてたり、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ます。

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



どうも、ハルシェーダが分割/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ダに渡されるようです。この分割後につながった新しい頂点に対して、頂点シェーダ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つてしたりします。

最後に仲間外れのコンピュートシェーダですが、これも使つたことはありません

### コンピュートシェーダ(GPGPU)

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

ちなみにレンダリングパイプラインのどこにも ComputeShader はありません。

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。ではなぜシェーダなのかと言うと、とにかく GPU 上で動くプログラムを慣例的に「シェーダ」と言ってるからに過ぎません。

つまり ComputeShader というのはホンマは CPU でやるべき数値計算を GPU 側でやってると思ってくれればいいです。GPU は速いというのがゲームや CG 業界以外にも知れ渡つてしまつて、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つてゐるわけです。

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) は役に立つています。

使つたことないのであまり言うとぼろが出そうのですが、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のが、ちょっと違うんですよ。CPU 1コアと GPU 1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1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 18 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なら万言てるんじゃないかと思ひます。調べてないから知らんけど。

まあ言つたら、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はそういう違つてあると思っておいてください。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。

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、おせ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んです。すんません。先に進みましょう。

## この辺書いてて思ったこと

いや、本当、俺さ、時代の流れについていけないよなあって思う。俺がここにきて7~8年経つんだけど、来たばかりの時はだれもスマホ持つてなかつたよ。俺は持つたけど…みんなが持つようになってスマホ使わなくなつたけどさ。まあともかくそんな時代からスマートホンが当たり前になつて、その間にもゲームを取り巻く状況が色々と変わっていくって、ねえ？ ゲームエンジンが出てきたからプログラマは楽ができるようになって、実力がそうでもない！ 学生も就職できるようになってC++とかもう要らないって言われたりしたと思ったらやっぱりUnityがScriptableRenderPipeline発表してプログラマの負担が間違ひなく上がつてくると思うようになつたり、ソシャゲの方が収益率が高いからってみんなそっちに言つちやつたり、Steamが出てきますますコンシューマ勢が弱くなってきたかなと思つたら相変わらず任天堂がつよかつたり、てやつてる間にシエータはVSとPSだけだった時代から今みたいにVS, PS, GS, HS, DS, CSってな時代になつたり、ユーチューバーが出てきたと思ったらVtuberだしVRだし、VR元年とか言ってたソニーが一人負けしそうだしそれでも相変わらずクソゲークソ映画は量産されてるし仮想通貨が割とすごい事になつたりディープラーニングがかつてと別物として活躍したり昔はクソ遅くて実用に耐えな

いって言われてたレイトレーシングがリアルタイムで実現しようとしているしでもちろん7年ってのはそれくらい変化するのは当然なんだけど振り返るとホントすごいなと思うわけ。俺がゲーム業界の中にいた頃はそれほど変化してなかつた気がするんだが…。という事で何が言いたいのかと言うとみんな大変な時代にゲーム開発者を目指すんだねえってこと。

技術的にも文化的にも激動の時代になってるんですね。正直大学、専門学校はそれについていけない気がします。

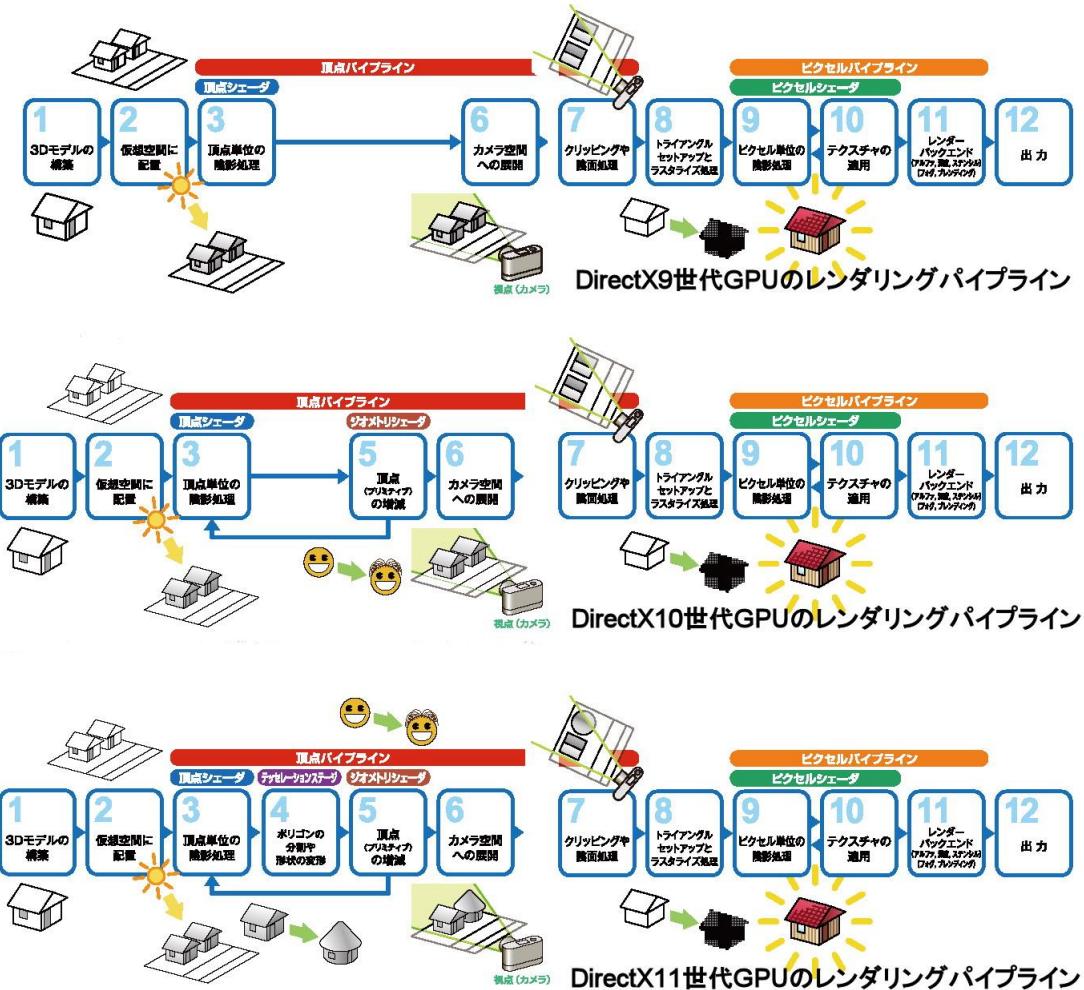
バンナムさんも「もはや学校にまかせておけませんから(キリッ)って言っちゃう状況。これは本当にそうで、我々がクソザコナメクジなのがほんま…ホンマに不甲斐ないつ…!!

さて、話を戻してではここで出てきたレンタリング/パイプラインとは何ぞや?

→次ページへ

## 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。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側ってあまり変わってないんですねえ。

んまあとはいっても、今後はどうなるか分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、ラスタライズとかね。

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ライズまでくればあとは基

基本的にピクセルシェーダを経て、レンダーターゲットにレンダリングって事なんですが、レンダーターゲットニ画面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ります。

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な事になりますので、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。

## 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。

<https://docs.microsoft.com/en-us/windows/desktop/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。

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になもんだ!!  
(実は『英語』という言葉にビビってるだけということはよくある)  
もつと言ふと、卒業生の〇野くんとかは卒業して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、楽しさ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ば、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](http://www.project-asura.com/program/d3d12/d3d12_001.html)

とか

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

とか

<https://qiita.com/em7dfggbcadd9/items/483cb0fa0bbf10f510d7>

とか

[http://d.hatena.ne.jp/shuichi\\_h/20150502](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/25bfa7a0017dbd888e39>

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』してますと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 使うやつは避ってよし。あれ、業界内でも嫌われてるんじゃないかな…。

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、横着しようと身をもって思い知ることになるであろう。

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!!!って思ってれば迷う事もないしね。

一応、技術記事とか読んでると「性能差が～」とか言われてますが、どっちがっちゅうと DirectX11 時代の問題点に触れておいた方がいいのかもしれません。恐らく皆さん DX11 を直接いじることはもうないと思いますのでお話しし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ね。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。

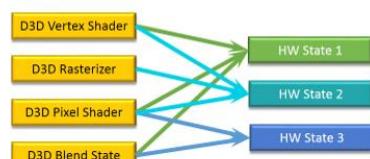
イメージわくかな？

で、1 つのモデルの中でもこのステートはパンパン変わっちゃうわけ…一例を出する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くつくわけだ。

#### Direct3D 11 – Pipeline State Overhead

Small state objects → Hardware mismatch overhead



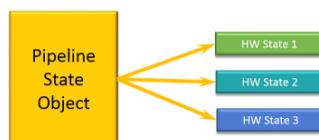
図で説明されてるサイトとかだとこんな感じですね。何となく切り替えコストが無駄になっているのが伝わるかなーと思います。何となくでいいですよ？無理に理解しようとしなくていいです。

で、12 ではそういうのをまとめて GPU に投げておいて、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れ。

#### Direct3D 12 – Pipeline State Optimization

Group pipeline into single object

Copy from PSO to Hardware State



で、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」と言うものが流れていると思つていただきたい。

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でますがそれらは、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しまっており、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ことで、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

## 仮想メモリ(仮想アドレス)とは

I 藤くんから『GPU 仮想アドレスって何ですか?』というご質問がありましたので、軽く説明しておきます。

[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)

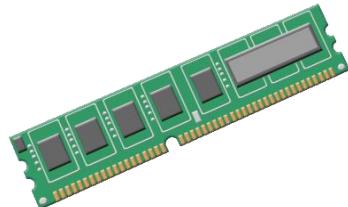
<https://ja.wikipedia.org/wiki/%E4%BB%AE%E6%83%B3%E8%A8%98%E6%86%B6>

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。

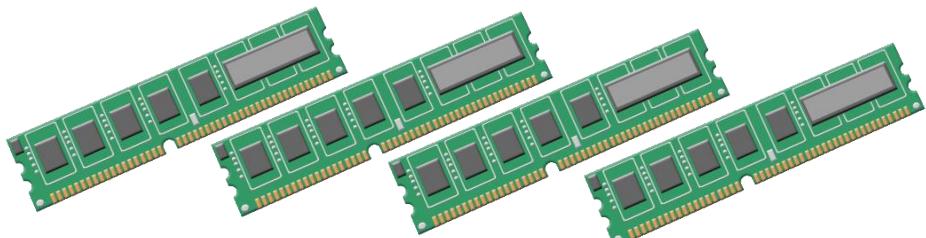
仕組みとしては、物理メモリをそのまま使うより、OSっていうか MMU(メモリマネジメントユニット)がマネジメントした「仮想メモリ」を見たほうが便利なので、基本的にプログラムはこの仮想メモリを通して物理メモリにアクセスしています。

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？なんでこんなかつい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本では足りなかつ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い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能够があるため、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。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と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も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つたりするんだ。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つけて一気に計算するように工夫する。ちなみに乗らなかつた場合や、欲しいデータがない場合は一度キャッシュが破棄され、別のデータを乗つ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」とかなんとか言う。

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こである。でもあまり神経質にならなくていいと思う。

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。高速化もちょっとだけありますけれどもね。

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。

## DirectX12 組み込み

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。

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。で、この命令も例にもれず『おまとめ』の対象であり、命令に関してはコマンドキュー、コマンドリストでおまとめするルールとなっております。

モチロン命令に関してもメモリ使っておまとめしますので、それ用のが必要になります。

ということで

- D3D12DescriptorHeap
- D3D12CommandList
- D3D12CommandQueue

の初期化が必要となります。

で、先にも書きましたけど、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。そしてほっとくと確保や更新を待たずに処理が進みます。

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(実際には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を入れてあげる必要があります。

ああ～しんどい。必要だからこういう状況になってるとは言え本当にしんどい。

## 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つかの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、皆さんには真似しなくて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/en-us/windows/desktop/api/d3d12/nf-d3d12-d3d12createdevice>

わあ英語だ。まあ、慌てんな…そういう時はだな DirectX11 のマニュアルを見ながら

DirectX12 のマニュアルを並行して読もう。大体似たような意味です。  
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 でおk  
    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)

英語解説しかございません。

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

いくつが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か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いければ 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

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 ならどれか引つかかるんで…多分 12\_0 くらいが引つかかるはず。

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いて、そいつに & つけてポインタのアドレスを示して渡してあげます。

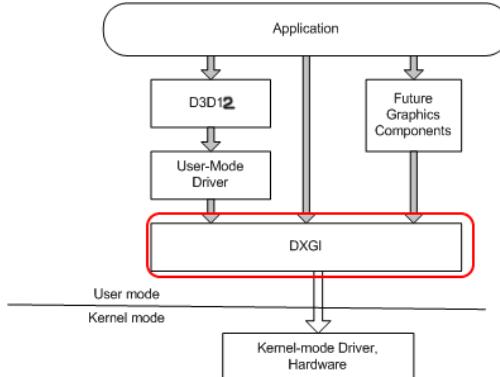
分からなかつたり、納得できない人はすぐに言ってください。フォローの講義をしますので…このへん納得できないまま進むと死ぬんで遠慮なく聞いてください。

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

で、ここで出てくる DXGI と言う言葉ですが、これもキーワードです。

[https://msdn.microsoft.com/ja-jp/library/ee415671\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee415671(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ね。



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

でIXGIFactory6があるわけですが

[https://docs.microsoft.com/en-us/windows/desktop/api/dxgi1\\_6/nn-dxgi1\\_6-idxgifactory6](https://docs.microsoft.com/en-us/windows/desktop/api/dxgi1_6/nn-dxgi1_6-idxgifactory6)

これどうやって実体を作るのか書いてないんですね。ひどくね？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が、行けるんかなあ…ホントマDirectX12の仕様とマニュアルレベルが減固めろや…。

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

- レンダーターゲット(リソース)
- 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

それら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

- 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。

## 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
- ディスクリプターヒープ
- ディスクリプタハンドル

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. スワップチェイ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

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“RenderTargetView”的略です。

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りませんでした。

The number of descriptors in the heap.

うう…ごめん、これでは何の情報量もないよ。

なのでサンプルを見ながら考えましたが、こいつは既に設定している画面のバッファ数と同

じで良いようです。つまり今回であれば2を指定しましょう。

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。

```
ID3D12DescriptorHeap* descriptorHeap = nullptr;
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。

…1段階目がこれだよ。

次にデスクリプターヒープサイズを計算します。

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);
```

終わりです。久々に心がほっこりするね。

DirectX12の特徴として、11の頃はバラバラにしてGPUに送っていた情報をまとめて送るつて流れになっています。

そうは言っても「ビュー」だの「サンプル」だの言われてもよー分からんだろうから軽く説明しつぶやく？

11の頃に説明したことそのままで

## ビューの解説

「コイツを視界とかビュー行列とか視点とかの、あのビューと勘違いすると途端にわけわからん事になるので注意しよう。あくまでもこの場合の『ビュー』はデータに対する『見方』の意味のビューだと思っておいてくれ。(モデルビューコントロール【MVC】を知つてたら理解が早いだろうけど…知らんだろうなあ)

ビューとは『データの塊へのリンクと、その使い方を定義するもの』と思ってくれ。意味がわかりづらいなら、ビューっていうのはコンピュータの中の『絵をバッファに描く職人さん』くらいに考えておいたら良いよ。

というわけです。

次にサンプラーの解説ですが

サンプラーってのは、テクスチャをサンプリングする人です。もう少し言うと、テクスチャをサンプリングする方法を知ってる奴です。そもそもテクスチャサンプリングっていうのが何かというと、uv 値(テクスチャにおける XY 座標みたいなもん)を元に、その UV 値に対応する『色』を取得することです。で、この色の取得の仕方に色々とあってそれの設定を知ってて、サンプリングするのがサンプラーです。

まあ、良く分からんかもしれないけど、今はきっちり分かる必要もないだろう。なんでかって？ 分かろうとすると『テクスチャアドレッシングモード』だのなんだのがまた出てきていつまで経っても DirectX12 の初期化すら終わらないからさ。

初回はさらっと流して、何度もプログラム組んでたらわかるよ。

ともかく今作ってるのはビューとサンプラーを乗せるジャパリバス『デスクリプタ』である。さて既にデスクリプタのためのヒープ(必要な領域)

で、いよいよデスクリプターハンドルの作成ですが、ちょっと毛色の違うものが出てきます。

CD3DX12\_CPU\_DESCRIPTOR\_HANDLE

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/mt186565\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/mt186565(v=vs.85).aspx)

です。これはお助けクラスです。だいたい頭に CD3DX12 って書いてたらお助けライブラリと思っておいてください。

コンストラクタにさっき作ったヒープを入れることによってひとまず生成されます。

ちょっと面倒なんだけど

CD3DX12\_CPU\_DESCRIPTOR\_HANDLE descriptorHandle(descriptorHeap);

ではうまくいかないのよね。

定義を見ると

```
const D3D12_CPU_DESCRIPTOR_HANDLE &o
```

なので、こいつの引数は `D3D12_CPU_DESCRIPTOR_HANDLE` 型である必要がある。ちょっとツラいけど、ここで

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/mt186565\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/mt186565(v=vs.85).aspx)

と

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn788648\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn788648(v=vs.85).aspx)

をもう一回見てみよう。見ての通り英語だ。僕もつらい。

Method	Description
GetCPUDescriptorHandleForHeapStart	Gets the CPU descriptor handle that represents the start of the heap.
GetDesc	Gets the descriptor heap description.
GetGPUDescriptorHandleForHeapStart	Gets the GPU descriptor handle that represents the start of the heap.

`GetCPUDescriptorHandleForHeapStart` を使用します。

「ヒープの開始を表す CPU ディスクリプタハンドルを取得します。」  
これっぽいですね。

一応儀式的に

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn899174\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899174(v=vs.85).aspx)

をさらっと読んだら

```
CD3DX12_CPU_DESCRIPTOR_HANDLE descriptorHandle(descriptorHeap->GetCPUDescriptorHandleForHeapStart());
```

てな感じで、デスクリプタハンドルを作りましょう。

## レンダーターゲット

ここが“できたらレンダーターゲットの仕組みを作りましょう。DirectX11 をやってた人たちは分かると思いますが、画面上にモノを表示するには…”というか画面に限らず何かに描画するにはレンダーターゲットというものが必要です。

レンダーってのは描画するって意味で、ターゲットはそのままの意味ですね。描画する先を設定するってことです。

で、最終的に必要になってくるのは「レンダーターゲット」と「レンダーターゲットビュー」です。ありがたいことにスワップチェインを作った時点でレンダーターゲット自体のメモリは確保されているんです。これはDirectX11もX9も同じです。

ということでひとまずはすべてのレンダーターゲットへの参照(ポインタ)を確保しておきます。というわけでスワップチェイン数ぶんのレンダーターゲットポインタの配列を作ります。で、レンダーターゲット自体は「テクスチャ」つまり「絵」と同じです。つまり今から絵を描こうとする「キャンバス」だと思ってください。

つーわけで

```
std::vector<ID3D12Resource*> renderTargets;
```

を宣言します。インターフェイスが ID3D12RenderTarget ではなく ID3D12Resource なのは前述のとおりレンダーターゲットは「絵」だからです。基本的に DirectX では絵のことをリソースと言っています(先に進むとリソースがさすものは「絵」だけではないことが分かるがそれはまた後程)

さて、レンダーターゲット数が必要なんだけど、これどうやって取得しよう。自分で設定したものだからどつかの定数にぶち込んでそれ使えばいいんだけど、正直スマートじゃない気がする。何とかならんか。

という事でスワップチェインから取得することにする。やり方は SwapChain::GetDesc で情報を取得。そしてその中の BufferCount からレンダーターゲット数を取得する。つまり

```
DXGI_SWAP_CHAIN_DESC swcDesc = {};
```

```
swapChain->GetDesc(&swcDesc);
```

```
int renderTargetsNum = swcDesc.BufferCount;
```

こういう事。わかる? 分からん人は正直に質問してね。いつも言ってるけどほっといたらしくんよ? 死ぬよ? これマジでそういうものよ?

でループを回しながら、レンダーターゲットビューの作成をやっていきます。

レンダーターゲットビューってのは「絵を描く職人とキャンバス」のことですが、DX11 の頃に比べるとちょっとばかりややこしいので一旦正解のコードを書きます。書きますが、皆さん自身のコードでこれを書き換えるという事を忘れないようにしてください。もしこの通りに書い

て『センターの言うとおりに書いたのに動かんかった』とか言っても知りません。僕の所では動いてますんで。

```
//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 = swapChain->GetBuffer(i, IID_PPV_ARGS(&renderTargets(i))); //スワップチェインから『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.Offset(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
}
```

で、この解説を今からやっていきますが、一応リザルトとか見ながらうまく動いてるのを確認してください。

最初に『デスクリプタインクリメントサイズ』ってのを取得していますが

[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)

これは何かというと、複数のレンダーターゲットビューを扱う場合はデスクリプタ内部にレンダーターゲットビューの配列(リストではないと思う)へのアドレスが必要で、配列になっているために、レンダーターゲットビューを定義するたびにオフセットしなければならないからです。

ジャバリバスで例えると、既にかばんちゃんが座っちゃったら、もうその席には座れないため次の席に座るんだけど、実は『次の席の場所』ってのがメモリ的に明確ではないため、あらかじめ席のサイズをとつといて、前の人人が席を占有するたびに場所情報を変更するって感じなのだ。

で、次にループの中に入るのだが、まずはスワップチェインから GetBuffer でスワップチェインが持っている『キャンバス』つまりリソースを取得します。

次にそれを元に CreateRenderTargetView でレンダーターゲットビューを作ります。これ、DirectX11だとレンダーターゲットごとに変数を作ったんですが、今回はデスクリプタの中に入っています。

CreateRenderTargetView がそいつを席に配置させてるので、その後で、デスクリプタの『席情報』をオフセットさせます。

ちなみにサンプラーに関しては今回はまだ使いません。ポリゴン出してテクスチャを張るときになると使います。

## ルートシグネチャー

またわけわかんない概念が出てきました。ルートシグネチャーです。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899208\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899208(v=vs.85).aspx)

これ読んでも良く分からなかったので、

<https://shobomaru.wordpress.com/2015/03/01/direct3d-12-update-at-idf14/>

を見ました。

『Root Signature は、Pipeline(全てのシェーダステージをまとめたもの、いわゆる Pipeline State Object(PSO))と対になっていて、Pipeline の型に合わせてアプリケーションが作る必要があるものです。』

よくわかりませんが…

RootSignature

| Descriptor Tables

| Descriptors

\ Constants

こういった構造になっているらしいです。

ともかく色々なものをまとめているという事はわかります。ともかく作っていきましょう。

```
ID3D12RootSignature* rootSignature=nullptr;//これが最終目的
```

```
ID3DBlob* signature=nullptr;
```

```
ID3DBlob* error=nullptr;
```

ちなみに ID3DBlob ってのは汎用的に使用するためのメモリオブジェクトだと思ってください。 Blob ってのは不定形ってな意味があります。興味があったら『不思議なプロビー』とか映画『the BLOB』を見ると分かりやすいかもしれません。

CreateRootSignature

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899182\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899182(v=vs.85).aspx)

を使います。

//ルートシグネチャの生成

```
result = dev->CreateRootSignature(0,  
    signature->GetBufferPointer(),  
    signature->GetBufferSize(),  
    IID_PPV_ARGS(&rootSignature));
```

で生成できるのですが、当然ながら signature が nullptr であるためクラッシュします。  
ではどのように signature を作るのかというと、

D3D12SerializeRootSignature を使用します。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn859363\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn859363(v=vs.85).aspx)

ここで第一引数である D3D12\_ROOT\_SIGNATURE\_DESC は Flags だけ指定すればよく

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986747\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986747(v=vs.85).aspx)

他は nullptr と 0 でいいので、

```
D3D12_ROOT_SIGNATURE_DESC rsd = {};  
rsd.Flags = D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT;
```

で十分です。これを D3D12SerializeRootSignature の第一引数に入れます。ちなみに

D3D12\_ROOT\_SIGNATURE\_FLAG\_ALLOW\_INPUT\_ASSEMBLER\_INPUT\_LAYOUT;

は

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn879480\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn879480(v=vs.85).aspx)

に書かれているように、

The app is opting in to using the Input Assembler (requiring an input layout that defines a set of vertex buffer bindings). Omitting this flag can result in one root argument space being saved on some hardware. Omit this flag if the Input Assembler is not required, though the optimization is minor.

に書かれているように、

「アプリケーションは、入力アセンブラー（頂点バッファバインディングのセットを定義する入力レイアウトが必要）を使用するようにオプトインしています。このフラグを省略すると、一部のハードウェアに 1 つのルート引数スペースが保存される可能性が

あります。入力アセンブラが不要な場合はこのフラグを省略しますが、最適化は軽微です。」

ということで、今回は「入力アセンブラ」は使用するので  
D3D12\_ROOT\_SIGNATURE\_FLAG\_ALLOW\_INPUT\_ASSEMBLER\_INPUT  
を指定します。

つまりこのように書くことになります。

```
D3D12_ROOT_SIGNATURE_DESC rsd = {};
```

```
rsd.Flags = D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT;
```

さて、これでできた RootSignatureDesc を使って、シリアル化していきましょう。

第一引数はこの rsd のアドレスを代入し、第二引数は D3D\_ROOT\_SIGNATURE\_VERSION\_1  
を指定しておけばいい。

残り 2 つは signature と error なので、アドレスをそのまま入れればよい。

さて、これで CreateRootSignature ができたらオッケーです。

通常であればここから「パイプラインステート」に入るんですが、今の所「パイプ  
ライン」に乗せるもの(頂点だのシェーダだの)がないのでちょっと後回しにしま  
す。

## 実際に画面に影響(色を変える)を与える

画面の色を変えるにはコマンドリストに「画面をクリア」コマンドを発行し、それ  
を実行すればいいわけです。大枠的にはこれだけです。

「画面をクリア」はバックバッファ(裏面)をクリアという事で、ひいてはレンダ  
ターゲットをクリアという事になります。

## レンターターゲットクリアコマンド発行

という事で ClearRenderTargetView 関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903842\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903842(v=vs.85).aspx)

DX11 やった人にはお馴染みなのではないでしょうか。

```
void ClearRenderTargetView(  
    D3D12_CPU_DESCRIPTOR_HANDLE RenderTargetView, // デスクリプタハンドル  
    const FLOAT ColorRGBA[4], // クリアカラー  
    UINT NumRects, // 最後の引数の矩形がいくつあるのか  
    const D3D12_RECT *pRects // 特定の領域だけクリアするのに使う「矩形」  
);
```

で、これをメインループの先頭あたりで呼び出します。

こいつはコマンドリストの持ち物なので…あとはわかるな?

```
_commandList->ClearRenderTargetView(descriptorHandle, clearColor, 0, nullptr);
```

さて…

実は↑の関数の第一引数がちょっとマズい。間違ってはいけないんだが、思い通りの挙動にはならない。でもとりあえずそれは知らないふりをしながら先を書いていく。

今は命令自体は画面のクリアだけなので、さっさとキューに対して実行命令を出していきましょう。

実行命令を出す前にやらなければいけないことがあって、それはコマンドリストを閉じるという事が必要です。Close 命令です。Close 命令を出さずに Executeしようとすると GPU 側に例外(実行時エラーみたいなやつ。クラッシュするわけではない)が発生します。

ぱっと見では分かりませんが、

出力の部分にはこんな感じのメッセージが出てしまいます。これが出るとマズいと思ってください。

ひとまずはクローズをクリア命令の直後で呼んでください。

クローズしたら漸く実行できます。実行はリストからするのではなくコマンドキーから行います。

## ExecuteCommandLists

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788631\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788631(v=vs.85).aspx)

これを使用すればすでに発行されているコマンドリストの内容を一気に実行します。

```
void ExecuteCommandLists(  
    (in) UINT           NumCommandLists, // コマンドリストの数  
    (in) ID3D12CommandList *const *ppCommandLists // コマンドリストポインタ配列の先頭アドレス  
);
```

ご覧の通り大して難しくはないです。ホンマはノーヒントでやって欲しいんだけど、時間もあまりないので書いておくと

```
_commandQueue->ExecuteCommandLists(1, (ID3D12CommandList* const*)&_commandList);  
こんな感じです。今回はコマンドリストが1つだけなので、第一引数は1でいいし、第二引数もキャストでなんとかしています。
```

コマンドリストの配列を使うならばここが1以上になりますし、第二引数も配列の先頭アドレスになります。

ただし、この場合キャストが面倒だし分かりにくいくらいもあるので一般的には

```
ID3D12CommandList* commandlist() = { _commandList };  
_commandQueue->ExecuteCommandLists(_countof(commandlist), commandlist);  
みたいな書き方をすることが多いです。
```

これで確かにコマンドは実行されるのですが、画面に変化は起こりません。大事なことがいくつか抜けているのです。

- コマンドアロケータのリセット
- コマンドリストのリセット
- 書き込むべきレンダーターゲットをどれにするのか指定する
- Present 関数で表画面と裏画面を入れ替える(ScreenFlipみたいな処理)
- 参照すべきレンダーターゲット(裏画面インデックス)がどれか調べる

まず、コマンドアロケータやコマンドリストをリセットするのは何でかというと、キューは実行したら勝手に消えるんですが、リストは残ってしまうし、アロケータもクリアしつかないといゴミが残るんでリセットする必要があります。

ループの先頭で

```
_commandAllocator->Reset();  
_commandList->Reset(アロケータ, ぬるぽ);  
やつといてください。
```

次に書き込むべきレンダーターゲットを指定するのですが、ここは DirectX11 の時と同じ関数 `OMSetRenderTarget` という関数を使用します。

じゃつかん話は変わりますが、DirectX の命令にはこういう OM だの IA だの良く分からない接頭語がつく関数があります。これは DirectX におけるグラフィックスパイプラインってのが、上から下に以下の図のようになっていて



この頭文字をとったモノなのね。で、今回使用する `OMSetRenderTarget` ってのは `OutputMerger` ステージにあたるもので

[https://msdn.microsoft.com/ja-jp/library/ee415707\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee415707(v=vs.85).aspx)

最終的に画面(レンダーターゲット)になんか出すべきところを設定するものです。ですから、書き込み先の指定などは `OutputMerger` の接頭辞がついているのです。こういうのもそのうち慣れれます。

それはともかく `OMSetRenderTarget` ですが

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986884\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986884(v=vs.85).aspx)

DirectX11の時から比べるとちょっと変わってます。

### DX11バージョン

```
void OMSetRenderTargets(  
    [in]          UINT           NumViews,  
    [in, optional] ID3D11RenderTargetView *const *ppRenderTargetViews,  
    [in, optional] ID3D11DepthStencilView     *pDepthStencilView  
);
```

### DX12バージョン

```
void OMSetRenderTargets(  
    [in]          UINT           NumRenderTargetDescriptors,  
    [in, optional] const D3D12_CPU_DESCRIPTOR_HANDLE *pRenderTargetDescriptors,  
    [in]          BOOL            RTsSingleHandleToDescriptorRange,  
    [in, optional] const D3D12_CPU_DESCRIPTOR_HANDLE *pDepthStencilDescriptor  
);  
_commandList->OMSetRenderTargets(1, &descriptorHandle, false, nullptr);
```

このようにしてはいけない。

既にオフセットしちゃってるってのもあるけど、それだけじゃない。何かというと、ここでセットされるレンダーターゲットは「裏画面」に当たるものでなければならぬ。

そしてその指し示すべきレンダーターゲットは Present を呼び出すたびに変更される。

Present 関数ってのはなにかを誰かにあげるって意味ではなく、表に出すって意味がって、そういう意味だと思います。つまり裏画面を表画面に持ってくるわけです。

ちなみに Present 関数はこう書いてください。

```
swapchain->Present(1,0);
```

とりあえずこれでフリップされるんですが、さっきも言ったように、これをやると裏画面が表になり、表画面が裏になる。

つまり「裏画面」を示すインデックスが変更されるのだ。

というわけである関数を呼び出す。

GetCurrentBackBufferIndex

である。

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn903675\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn903675(v=vs.85).aspx)

まあなんてことはない。

裏画面のインデックスを得るだけだ。

```
int bbIndex=swapchain->GetCurrentBackBufferIndex();
```

はい、これで裏画面インデックスが取得できるわけだから、OMSetRenderTarget の書き込み先もこれで指定しよう。

```
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(descriptorHeap->GetCPUDescriptorHandleForHeapStart(),  
bbIndex, descriptorSize);
```

で、これで取得した rtvHandle に対して OMSetRenderTarget すればよい。また、クリアすべきレンダーターゲットもこの rtvHandle にすべきである。

さて、クリアの際に色を変更すべきですが、色が中途半端では変化が分かりづらいですね。

クリアの部分は真っ赤にするくらいでいいんじゃないでしょうか。

```
const float clearColor() = { 1.0f, 0.0f, 0.0f, 1.0f };  
_commandList->ClearRenderTargetView(descriptorHandle, clearColor, 0, nullptr);
```

ちなみにカラーの範囲は 0~255 ではなく、0.0f~1.0f であることに注意してください。

ここが R8G8B8A8\_UNORM と関わってるんです。

余裕のある人は色に変化が出るようにしてみてください。

## 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必要”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ないのが 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.o07.jp/function/dxfunc\\_other.html#R19N1](http://dxlib.o.o07.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間に関わらず処理を返します。

前にも言ったかも知れませんが、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に入るようになって、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い「不完全なまま」データを読み取ろうとすればそれはもうね、蛹を羽化前に開いてちゃったり、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た。

というわけできちんと準備でき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.007.jp/function/dxfunc\\_other.html#R21N2](http://dxlib.o.007.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てくれていい!(特に DirectX12 においては)わけで、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、まあおかしなことになってしまふわけです。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(ID3D12Fence)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。

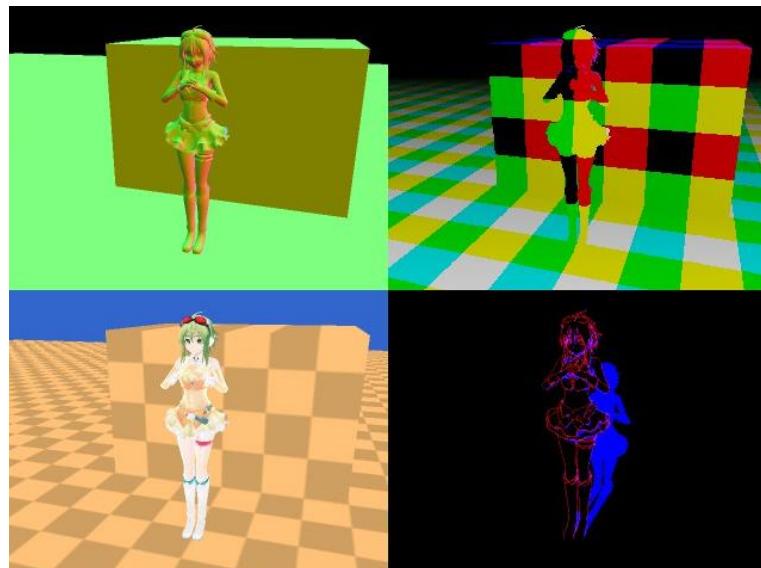
そもそもなんでこんなややこしいことをする羽目になったのかというと

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色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。

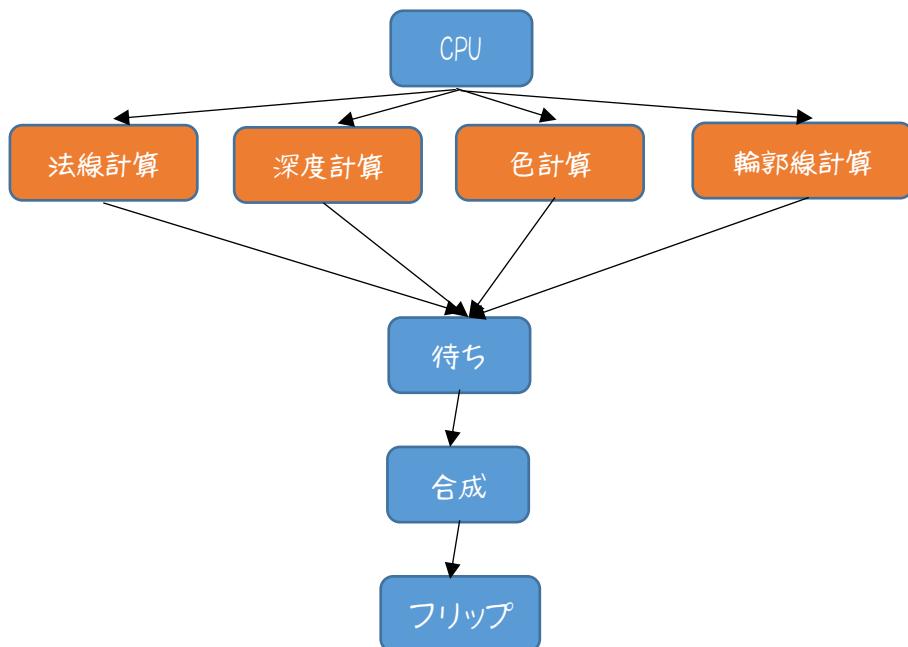
意味が分からぬんだろうから簡単に言うと。

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つておいて、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ため



すつぜー大雑把に言うと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つておかなければならぬみたいだ。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などは対応していると思いますがGeForceGTX860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、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 ExecuteCommand 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。

一番手っ取り早く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…これやつちや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？

とりあえずこれをやれば不具合はなくなると思うけど…どうかな？面倒だねえ。

うん、ここまで書いてて思ったけどさ、実はおかしくなる原因についてなんだけど「フリップが命令完了よりも先に実行される」が原因というよりは「命令完了の前にCommandAllocator や CommandList がリセットされている」事が原因みたい。

そりやそうか。実行中にリストがリセットされりやそりやおかしくなるわな。

(メモ)

d3d12\_1

d3dx12.h

directxmath.h

d3dx12.h はバージョン合わせないといけない…

Dx12Wrapper 的なクラスを作るか…名前面倒なので Dx12Wrapper でいいかな。

ウインドウハンドル HWND とアプリケーションインスタンス hInst が必要

たぶん、ルートシグチャとパイプラインステートオブジェクトの説明が最初は一番しんどい…話す側も聞く側もね。

(～メモ)