

DirectX12

初心者プログラミング■



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のお時間がやってまいりました。

前期でももう死にかけだったのに、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	7
流れ	7
環境設定	13
C++言語のおさらいとか追記とか	16
STLについて	16
おさらい	16
イテレータについて	18
リバースイテレータについて	18
stringについて	19
stringstreamについて	20
C++の新しい仕様	21
右辺値参照とムーブセマンティクス	22
配列の範囲	27
Emplacement(emplace,emplace_back)	27
(おかげ)minmaxとiota	28
constとconstexpr	29
まずはプロジェクトを作ろう	30
じゃあウィンドウ出すか	32
アプリケーションのハンドル	36
基礎知識説明①	39
シェーダ	39
頂点シェーダ	40
ピクセルシェーダ	40
ジオメトリシェーダ	41
ハルシェーダ(テセレーション)	42
コンピュートシェーダ(GPGPU)	43
この辺書いてて思ったこと	44
レンダリングパイプラインについて	46
DirectX組み込みに入る前に	47
DirectX12がそれ以前のDXと違うのはどこ?ここ?	48
仮想メモリ(仮想アドレス)とは	50
キャッシュメモリとか分岐予測とか	52
DirectX12組み込み	53
準備①(インクルードとリンク)	54

基本の初期化	55
画面に影響を与える準備	59
スワップチェイン	59
レンダーターゲットの作成	67
さて、いよいよ画面のクリアだ	73
コマンドを投げるために…	73
コマンドリストとコマンドアロケータをリセット	73
コマンド:レンダーターゲットを設定	75
コマンド:レンダーターゲットをクリア	76
コマンド:クローズ	76
コマンドキューに投げる	76
スワップチェーン Present	77
色々間違ってるんですけど	77
フェンス	78
ではフェンスを実装しようか	83
三角形の描画をしよう	85
頂点情報の設定と GPU 転送	85
頂点情報を作る	85
頂点/バッファ	86
頂点/バッファビュー	89
そんな事よりシェーダ書こうぜ	90
シェーダ読み込み	91
ルートシグネチャー	92
頂点レイアウト	99
パイプラインステートオブジェクト(PSO)	102
その他やらなければならぬ1事	105
リソースバリア	106
ビューポート	106
残り色々セット	108
ドロー!!ポリゴン!!!	109
うまくいかない場合	111
アプリがグラボを選ぶズエ…レルズエ…	111
四角形の描画をしよう	113
インデックス情報の設定と GPU 転送	114
インデックス配列を作る	114
インデックスバッファを作る	115

インデックスバッファをセット.....	115
ドロー(インデックスあり).....	115
テクスチャ貼りたいなあ.....	116
頂点情報にUVを追加.....	116
頂点シェーダ変更.....	117
テクスチャオブジェクト生成.....	117
書き込み.....	121
バリアとフェンス.....	122
シェーダリソースビューを作る.....	122
サンプラを設定.....	124
シェーダにテクスチャの受け取り側を記述する.....	125
ルートシグネチャを設定.....	126
毎フレームやること.....	126
リソースのL0とかL1について.....	129
DirectXTexture(WIC,DDS)を組み込み.....	132
行列で座標変換してみよう.....	134
行列おさらい.....	134
2D座標変換行列.....	135
定数バッファ.....	139
CPP側.....	139
シェーダ側.....	142
3D化してみる.....	142
XMVECTORについて.....	144
リファクタリング.....	146
ComPtrを使う.....	146
色々関数化する(コメントをきちんと書く).....	148
デスクリプター(テクスチャ,定数)まわりを支える設計.....	151
ヘッダだけ公開.....	155
ともかくPMDモデルを表示させよう.....	159
フォーマットを確認する.....	159
ヘッダ.....	159
頂点リスト.....	162
とりあえずモデル表示してみよう.....	163
BadAppleにしてみる.....	164
インデックス情報を読み込みましょう.....	165
シェーディングしてみる.....	166

深度バッファ	168
深度バッファとは	168
結局 DX12 では何をしなければならないの?	170
深度バッファの作成	171
深度バッファビューの作成	172
パイプラインステートオブジェクトに深度情報を追加	172
レンダーターゲットと深度バッファを関連付け	173
深度バッファをクリア(毎フレーム)	174
法線も座標変換	174
マテリアルを適用	176
マテリアルってなんや?	177
マテリアルデータ読み込み	178
クソコードでごめんなさい	180
2つの選えてないやり方	183
マテリアルのためのバッファ作成	184
ヒープとビューの作成	186
ルートシグネチャの設定	188
シェーダ	188
Draw 時の切り替え	188
テクスチャを入れよう!!!	189
UV 復活!!! UV 復活!!! UV 復活!!!	190
テクスチャの設定がない奴の取り扱い	191

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。こゝ注意してください。

えー、じゃあ何すんのさ?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)が何なのが…ゲームエンジンがやっている事はどういう事なのか…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、あれなのか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く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. 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、しつつやっていくので、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

大事な事ですしね…。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=H3M07qR0j28>

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。

とはいえる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つてゐ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。

- 1st 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
- 2nd 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

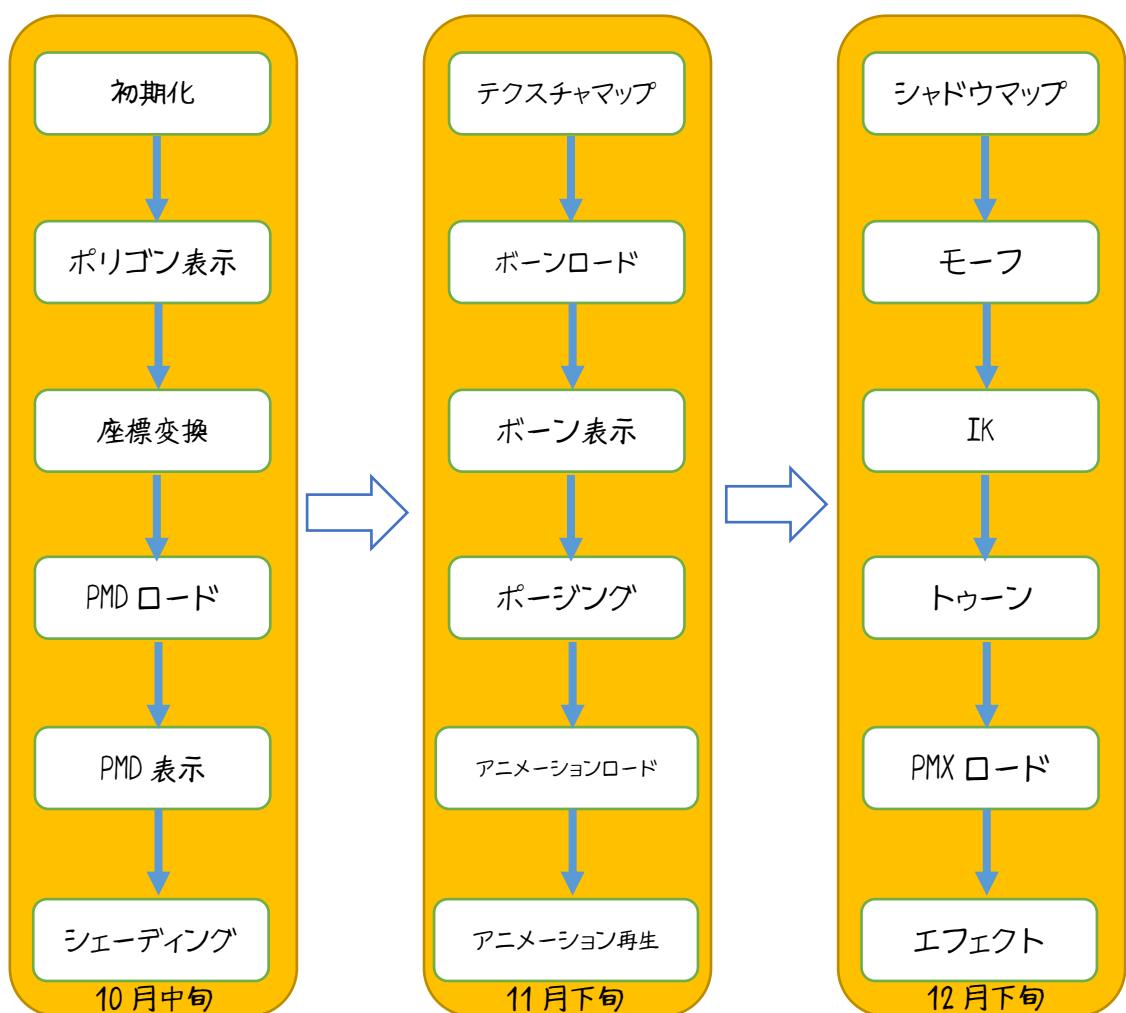
ンとポストエフェクトがないではありませんか!!!

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

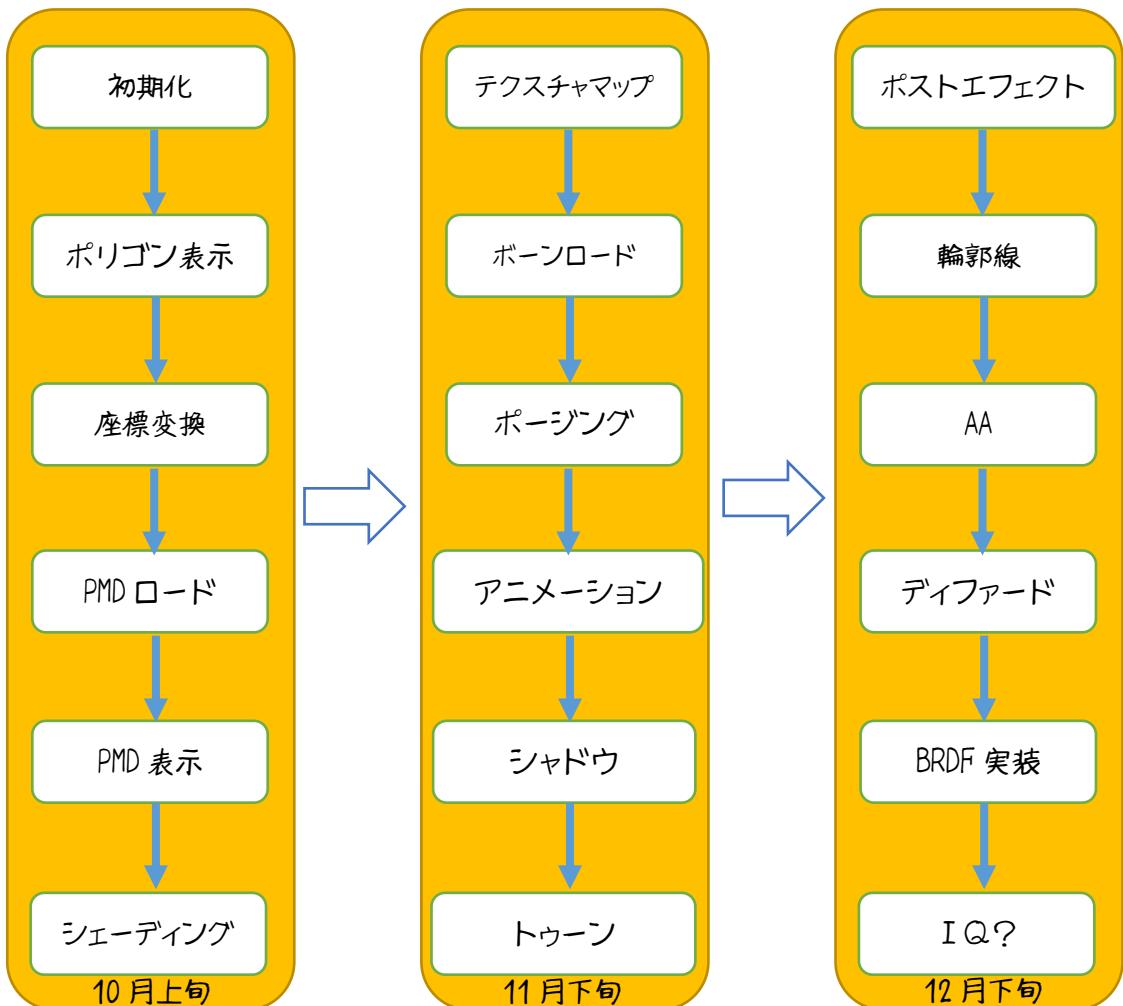
昨年まではやってなかつたのですが、DX11からの定番としてディファードレンダリングというのがあるので、今年はそれを入れていきたいと思います。さすがに欲張りセットかな?

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



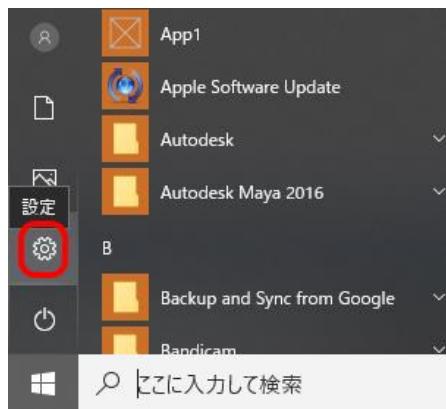
の状態にする必要があるのかなあ…もしくは

<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



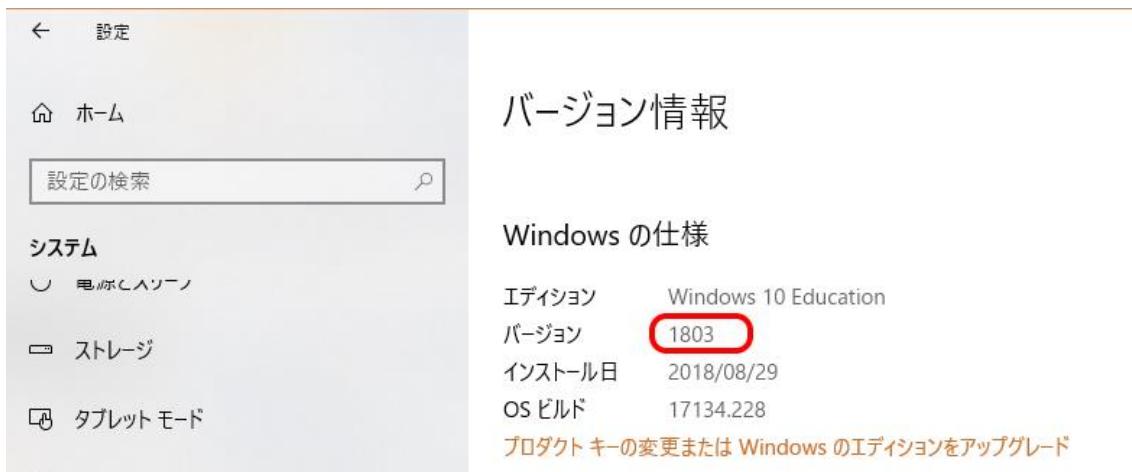
Window ボタン → 齒車マーク

Windows の設定

→システムを押すと、別画面が出てくるので左下の「バージョン情報」をクリック。

システム ディスプレイ、サウンド、通知、電源	デバイス Bluetooth、プリンター、マウス	電話 Android、iPhone のリンク	
個人用設定	アプリ	アカウント	
分辨率 1920 × 1080 (推奨)	向き 横		

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。フリーズしどんのんちゃうか?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	<p>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。</p> <p>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。</p> <p>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</p>

	高いが、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。

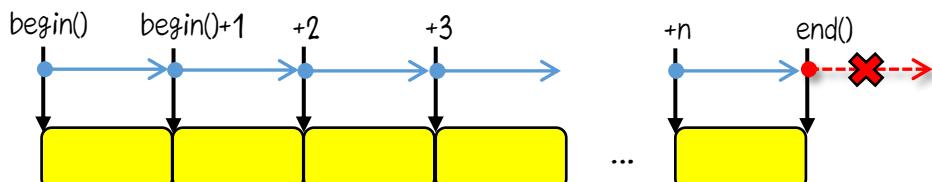
関数名	概要
for_each	指定された範囲内の要素に対して、特定の処理を行う
find	指定された値を検索しイテレータを返す
find_if	条件に合う要素を検索しイテレータを返す
find_first_of	条件に合う要素の中で最初に見つけたイテレータを返す
find_last_of	条件に合う要素の中で最後に見つけたイテレータを返す(いらんかも)
min_element	要素の中から最も小さい要素を持つイテレータを返す
max_element	要素の中から最も大きい要素を持つイテレータを返す
sort	要素をソート
count	指定した値を持つ要素数を返す
count_if	条件に合致する要素数を返す
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。今まで少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。

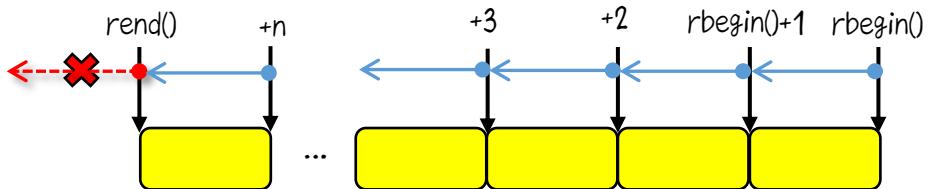
ここまで理解できましたか？

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。rbegin() 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「スキンメ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ったりするので、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 `Hello World` と出力できるだろう？こ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とかダ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>

をちょっと読み歩いてください。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 がなれin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。これはレリ。意図したとおりだから。しかし、もし以下のようないふる

```
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かないでしょ？という事で検証

まずこういうクラスを作る。

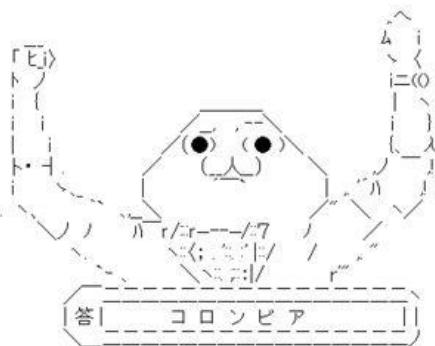
```
struct Boo {
    Boo& operator=(const Boo& b) { // コピーオペレータ
        cout << "Copy" << endl;
        return *this;
    }
    Boo& operator=(Boo&& b) { // ムーブオペレータ
        cout << "Move" << endl;
        return *this;
    }
};
```

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!!

```
C:\Users\kawano\Source\Repos\DX12Challenge\x64\Debug\DX12Challenge.exe
Copy
Copy
Move
```



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場合は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;
```

```
C:\Users\r_kawano\Source\Repl
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?

```
Move
```

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

```

C:\Users\r_kawano\Source
0000005F6D8FF874
0000005F6D8FF874

```

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\kawano\Source\repos\HelloDX12\HelloDX12> 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なもんやな…。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

```
C:\Users\kawano\S...
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 は微妙やったかも。

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 使ってき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

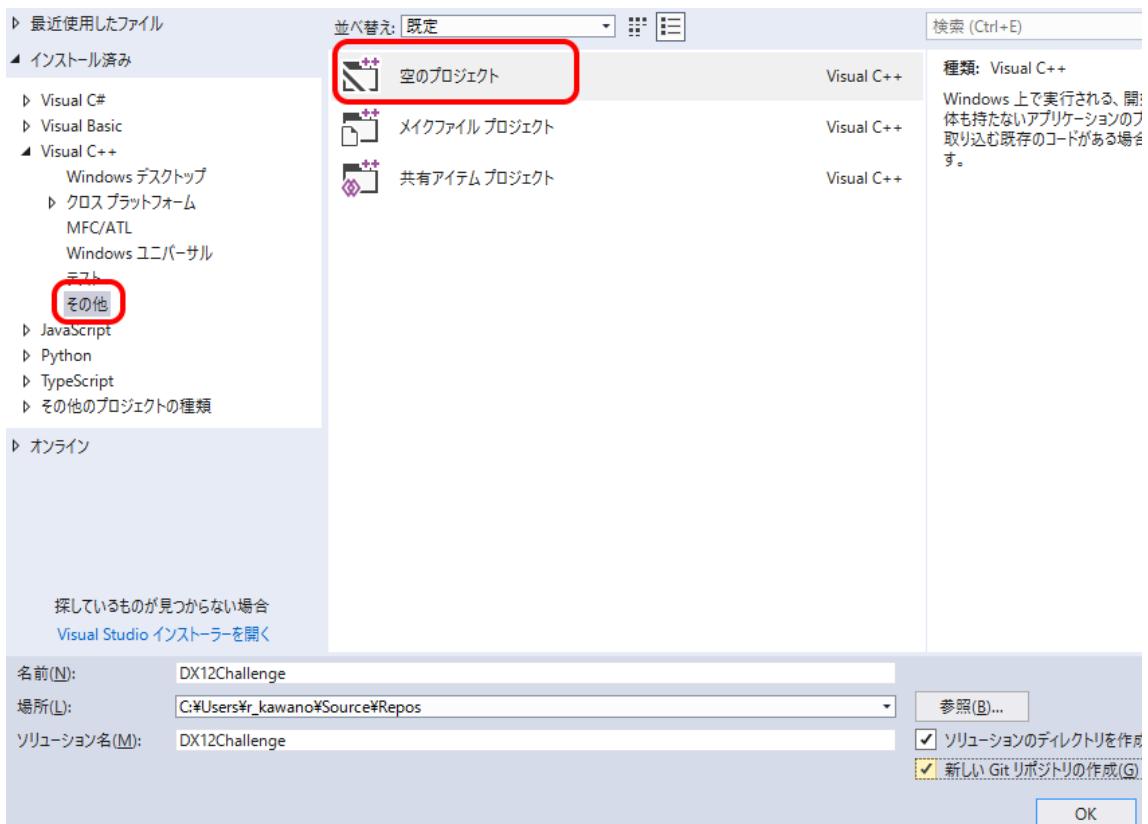
```
constexpr int ppp = Get(); // NG
```

```
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、僕がしつつ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れないん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はですが、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いがない)。

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;
```

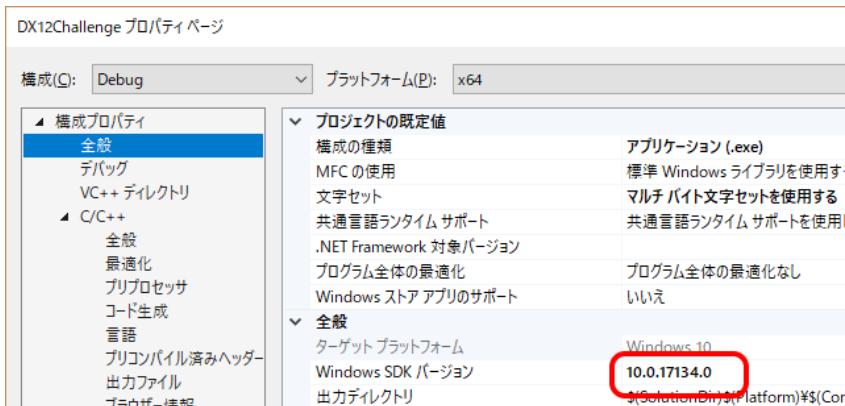
```
}
```

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>
```

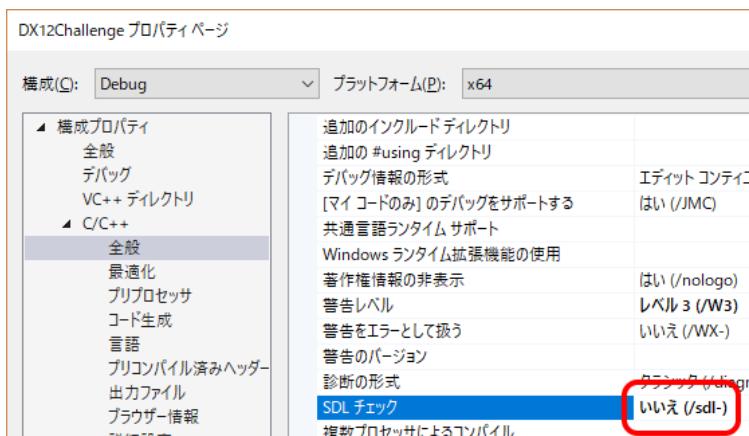
をインクルードはしておいてください。あ、ちなみにメイン関数がある cpp は main.cpp とします。ただただ main を実行するのみの関数ですね。

で、プロジェクトの設定に入りますが

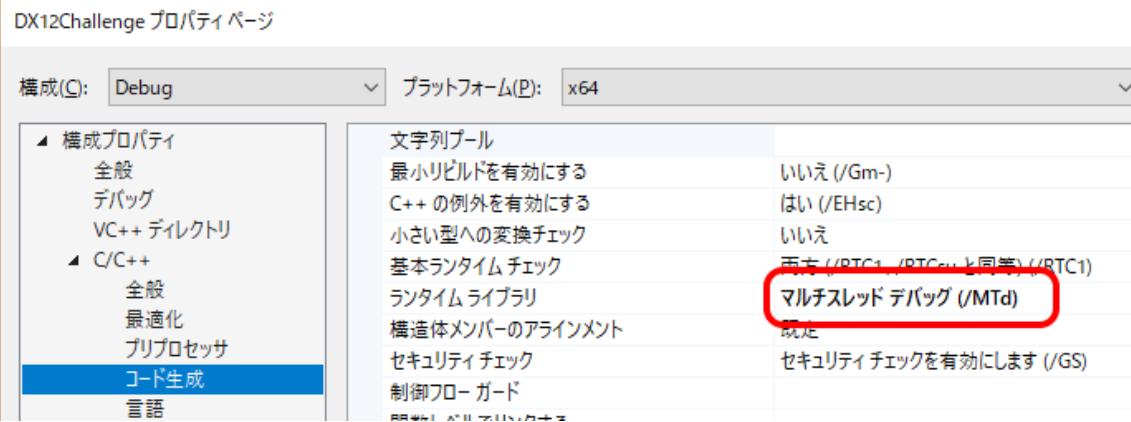


WindowsSDK が最新なのを確認してください。

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 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てきましょうか。

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りおきます。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」ってのが出でます。

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 番はやっぱり生理的にイヤ。

とりあえず「ウインドウズのウインドウを作るのに、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てる)、ウインドウを「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.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>しといでね。

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 に入れ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

```
HRESULT 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); // 標準の処理を行う
}
```

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下さい。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下さい。

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」だと思って下さい。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が、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ですが、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」と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)
- 休まないように(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。

最初に使われるも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 1280x720 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ともに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いう安っぽいテクノロジがありました。

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、たぶん実用的じゃないんじゃ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~。

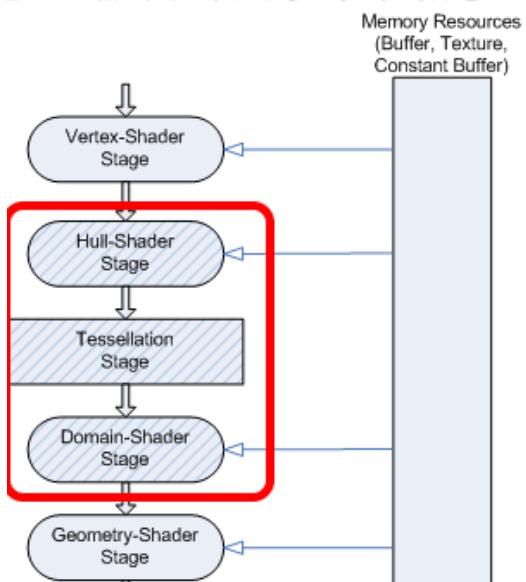
とりあえずそういうのがあって、なんか活用できそうなアイデアがあつたら使いたいと
思います。

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。いわゆるサブディビジョンですね。OpenSubdivだったので活用され
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れ
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
でき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) は役に立つています。

使つたことないのあまり言うとぼろが出そなうですが、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 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 はそういう違いが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『分歧』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ったよ。俺は持つてたけど…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って、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ってくると思うようになつたり、ソシヤゲの方が収益率が高いからってみんなそっちに言っちゃつたり、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、てやってる間にシェータは 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れ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った気がするんだが…。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。

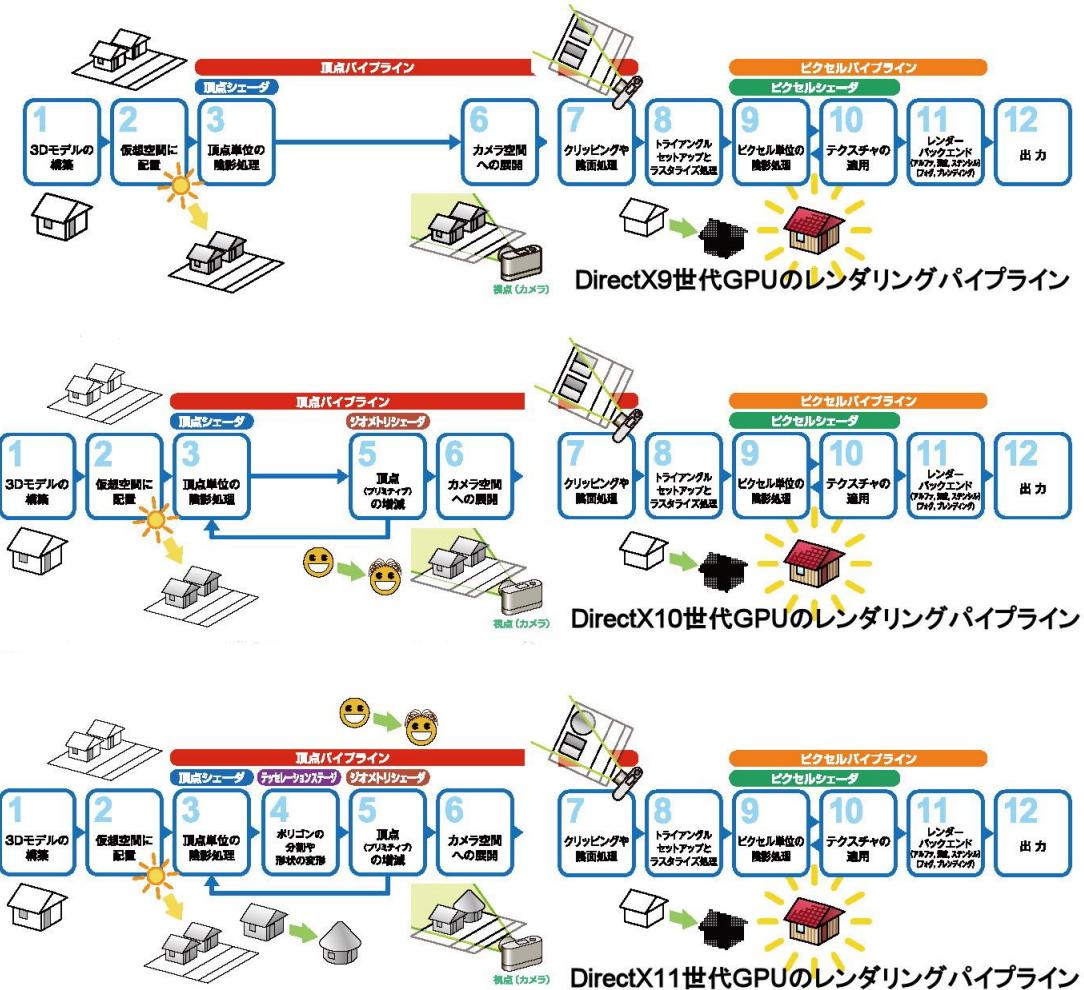
バンナムさんも「もはや学校にまかせておけませんから(キリッ)って言っちゃう状況。これは本当にそうで、我々がクソザコナメクジなのがほんま…ホンマに不甲斐ないつ…!!

さて、話を戻してではここで出てきたレンダ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由してレンダーランダムにレンダリングって事なんですが、レンダーランダムに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ニックがあります。

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要素になりますので、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%83%82%AB%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 はまだ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

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

とか

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

ついでに

<https://qiita.com/Onbashira/items/256fa7a0017dbd888e39>

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」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、横着しようとする身をもって思ふ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を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。

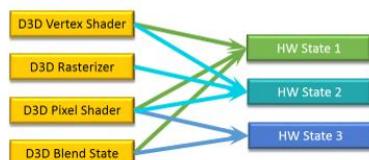
イメージわくかな?

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



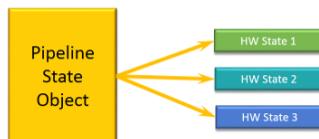
図で説明されてるサイトとかだとこんな感じですね。何となく切り替えコストが無駄になっているのが伝わるかなーと思います。何となくでいいですよ? 無理に理解しようとしなくていいです。

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そう

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)

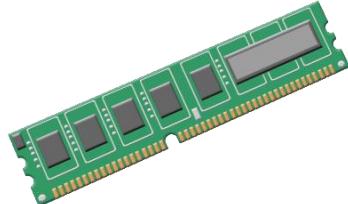
<https://ja.wikipedia.org/wiki/%E4%BB%AE%E6%83%B3%E8%A8%98%E6%86%B6>

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。

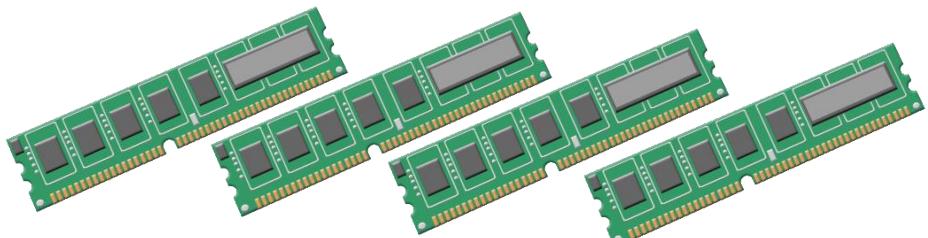
仕組みとしては、物理メモリをそのまま使うより、OSっていうか MMU(メモリマネジメントユニット)がマネジメントした「仮想メモリ」を見たほうが便利なので、基本的にプログラムはこの仮想メモリを通して物理メモリにアクセスしています。

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本では足りなかったりメモリ間を跨げたりするわけ。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事ができるため、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てしたりするんだ。大元の仕組みが今みたいに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

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。

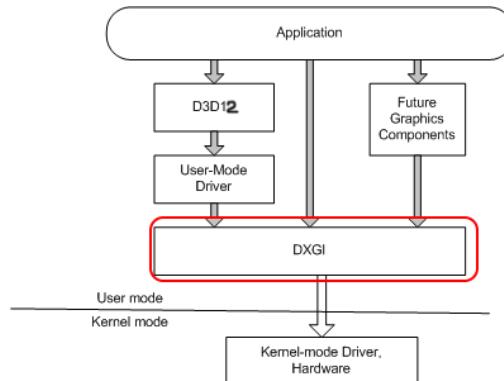
分からなかつたり、納得できない人はすぐに言ってください。フォローの講義をしますので…このへん納得できないまま進むと死ぬんで遠慮なく聞いてください。

で、次ですが、DXGISwapchain です。コレ1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/

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

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… 木ノマ DirectX12 の仕様とマニュアルレベルを減らめろや…。

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 カークル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

- スワップチェイン
- 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。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

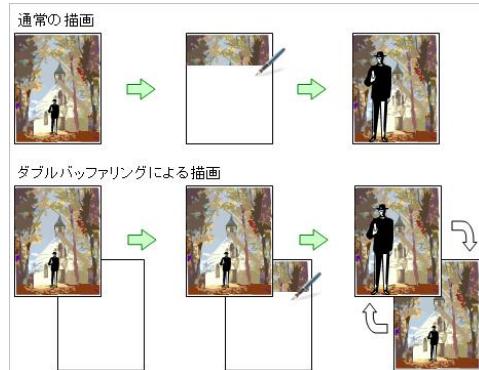
- フェンス

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。

スワップチェイン

スワップチェイ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イ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、スワップチェインはそれ以上も想定しています。

ただし…大抵の場合は 2 画面で十分と思います。

で、スワップチェインを作るときには、例によって CreateSwapChain 的な関数を使うんだけどウインドウと関連付けるためにウイ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?

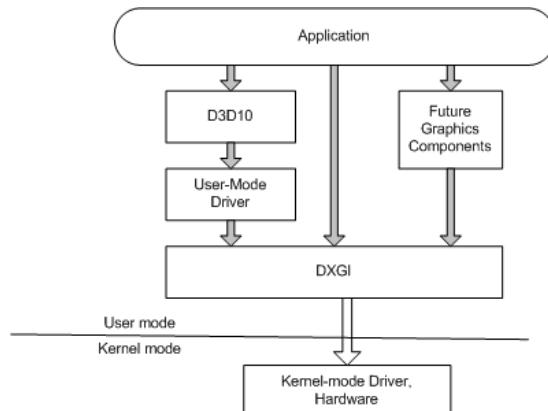
うーん。まだわからんかな?

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。

のほうが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ると思います。
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けど。

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「知ってなきやわからない」部分なので、ソースコード書いちやいますけど

```

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

```

こうやって作ります。result が S_OK のを確認してください。

さて、それではスワップチェイ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イ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)

定義を見るところなってますね?

```
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Quarityを0に)
    DXGI_USAGE BufferUsage; //バッファの使用法(あとで解説)
    UINT BufferCount; //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/windows/desktop/bb173072\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/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ね？

というわけで、まずは Format から…これはビット数が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ふと

「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ってください。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ーとして使用します。

`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ダ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ね。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やうしなあ…。とりあえず良く分からぬるので false にしておきます。

あ、そういうえば今一度 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));
```

にすべきってところです。

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。

急遽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 と言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インも初期化できます。

スワップチェイン

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入れたらいいのか分かります。

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が、

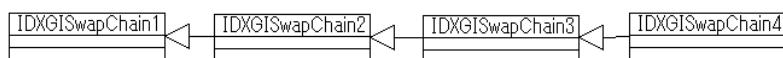
ひとまず FLIP_DISCARD を選んでください。役割はフリップした後にディスカード(破棄)します。つまり Present 前に裏画面に書き込んでおき、Present でフリップし、前の画像は破棄するって意味です。

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。

つまるところ、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なりませんかね



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。
ともあれこれでスワップチェインは終わりです。

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
 - ディスクリプターヒープ
 - ディ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

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が、

<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仲間になってるわけだ。

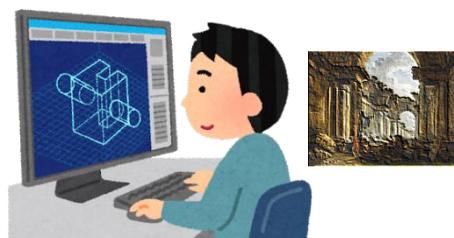
ビューとデスクリプター

実は昨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ラー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」って言ってますけど、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、深度職人てのもいて、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、色々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までは扱いは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ーです。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と…

「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。

つっこことです。意味が分からぬいかかもしれません、要はビューを作りたければまずデスクリプタヒープを作り、その中にビュー定義を配置しなさいという事のようです。

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)

デスクリプタヒープとデスクリプタテーブルの違いは、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ば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。

デスクリプタヒープを作るには

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" 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ぬんだって

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？

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(&descriptorHeap, 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オブジェクト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 の中でこの関数は々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](https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12-device-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

のメンバはptrしかないのですが、オフセットの仕様とかが意外と面倒だったりするため、ここで d3dx12.h を投入します。

<https://github.com/Microsoft/DirectX-Graphics-Samples/blob/master/Libraries/D3DX12/d3dx12.h>

とりあえずこのPDFを書いてる最中のバージョンは
Windows 10 April 2018 Update and Visual Studio 2017
なので、最初の設定をきっちりやってる人は大丈夫です。バージョンが違うと動かないこ

とがありますので、気を付けてください。

<https://docs.microsoft.com/en-us/windows/desktop/direct3d12/helper-structures-and-functions-for-d3d12>

正直このD3DXは使用したくないのですが、仕方ないです。

可能な限りブラックボックスにならない説明をすることを心がけますが、流石にこれがないとしんどくなってしまった。

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

一度この `CD3DX12_CPU_DESCRIPTOR_HANDLE` を見てもらうといいのですが、単なる元の

`D3D12_CPU_DESCRIPTOR_HANDLE` をラップしたクラスにすぎません。元の奴がまた内部に「ptr」を一個持ってるだけの構造体です。え？ 構造体も継承できるの？ できますよ。C++の場合の構造体とクラスの違いはアクセス指定だけですから。

DirectX系はこの手の継承してる奴が多いので、class と思ったら struct だったって事案が多いので気を付けてください。

そして、

```
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.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
}
```

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。たぶんシェーダを書き始めないと「なんで？」っていうのが分からなければから。

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 しかないんじゃない…

<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が

『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事です。

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 << "まだクソザコナメクジ" << 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。



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=_dsHeap->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->OMSetRenderTarget(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,//ひとつまず 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);
```

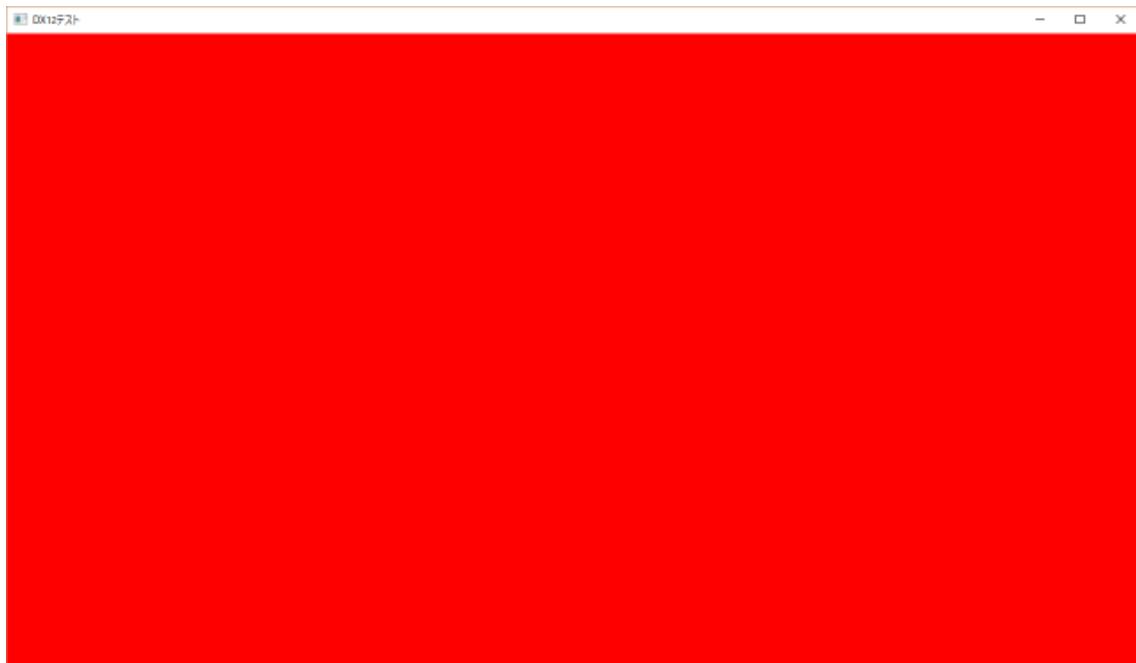
あとは Present 関数を呼べばいい

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でお願いします。レンダリングしてスワップします。と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 でいいですのでもう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。

色々間違ってるんです

実は色々と間違っています。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ますが、それはフリップするたびに変更されるはず。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

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ね!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ぬ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.0.007.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 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。

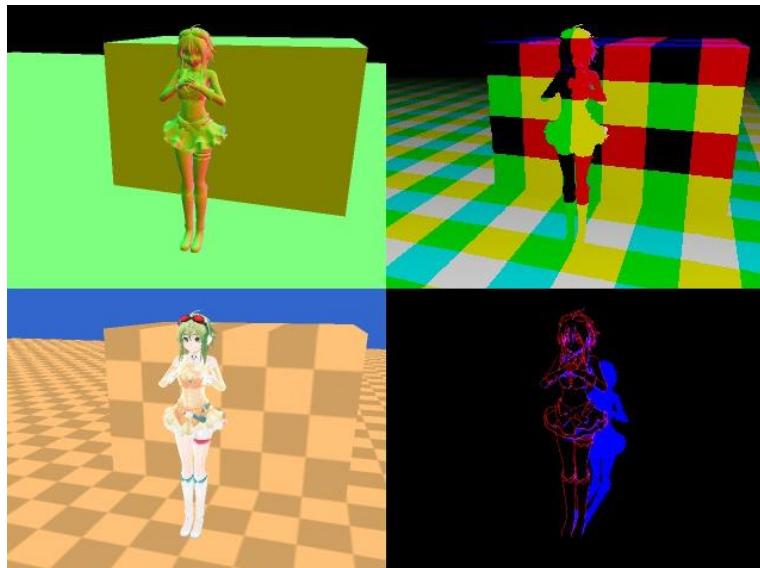
そもそもなんでこんなややこしいことをする羽目になったのかというと

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。

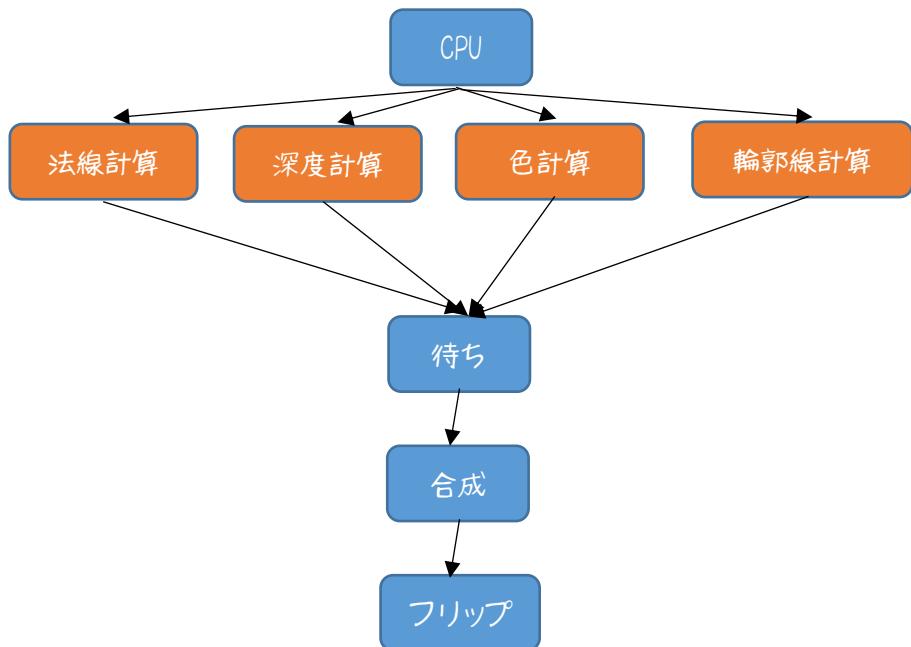
意味が分からぬだろうから簡単に言うと。

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ため



すげ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っておかなければならぬみたいだな。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 などは対応していると思いますが GeForce GTX 860 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 `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知れないのが

「GPU 側のコマンド処理が完了した時点で 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 占有しちゃつてもいいか。
…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() {
    _cmdQue->Signal(_fence, ++_fenceValue);
    while (_fence->GetCompletedValue() != _fenceValue)
        ;
}
```

三角形の描画をしよう

さて、いよいよ三角形の描画をしていきます。ここまでが本当に大変。

頂点情報の設定と GPU 転送

頂点情報を作る

今回は三角形を作っていきます。

頂点はいくつひとつかな? そう、3 頂点ですね?

んで、この3つの頂点の一つ一つにはどれくらいの情報量が必要かな? 座標情報が必要だからひとまずは X, Y, Z ですね。

まず構造体を作ってみましょう。

一応一番最初に便利ライブラリとして

#include<DirectXMath.h> をインクルードしているので、こいつを使えば数学的なところは幾分マシになるかなと思います。

ただし、こいつがちょっと面倒で、昨年の DirectX11 をやってる人にとってはちょっとだけ罷になっているのですが

float3 つぶんを表す XMFLOAT3 ってのがあるんですけど、こいつは DX11 の時にはそのまま使えました。しかし DirectXMath になってからはちょっと面倒で

DirectX::XMFLOAT3

って使い方になります。名前空間がくっついちゃってるんですね。面倒だと思う人は

```
using namespace DirectX;
```

って cpp の先頭(インクルードの後くらい)で書いてください。

くれぐれも言っておきますが、using namespace をヘッダ側で使用しないようにしてください。それは相当な悪手です。



さて、using namespace DirectX;を書いている前提で話を進めますけれども

```
struct Vertex{
    XMFFLOAT3 pos;//座標
};

こんなのは作ってください。一応意味は分かりますよね？そう
Vertex vertex;
vertex.pos.x=...
みたいにして頂点を定義して使うわけです。
とりあえず3点定義します。

//頂点情報の作成
Vertex vertices() = { {{0.0f,0.0f,0.0f}},
    {{ 1.0f,0.0f,0.0f }},
    {{ 0.0f,-1.0f,0.0f }} };
```

こんな感じで(正解とは言ってない)。次は頂点レイアウトの定義です。ちなみにこの「頂点の順序」は結構重要で、順序を間違えると表示されません。基本的に時計回りになるようにしてください。あとでどうにでもなりますが、理屈知らないと結構ハマる罠なので。

頂点バッファ

頂点情報をそのまま GPU 側に投げれるかというとそうではなくて、そんなに甘くもないのです。どうやって投げるのかと言うと頂点バッファおよび頂点バッファビューを使用して投げます。

まず頂点バッファを作ります。でも DirectX11 得意二キを罠にはめる情報も満載なのです。そ

そもそも ID3D12Buffer が存在しない。代わりに

ID3D12Resource を使用します。

<https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788709.aspx>

もう名前からして、11 の時より完全に「メモリの塊(リソース)」って感じがします。

ID3D12Resource* _vertexBuffer=nullptr;

とでも宣言しておいてください。

11までだったらこういうバッファを作りたければ GetBuffer だの CreateBuffer だのを使っていればよかった。だがそれはいけない。そのような関数は「もうない」のである。

代わりにあるのが CreateCommittedResource である。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899178\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899178(v=vs.85).aspx)

まあ…罷ですなあ。DirectX11 の CreateBuffer よりもパラメータ多いし…キツツレなホント。
ぶっちゃけパラメータ多くて面倒なので素直にサンプルに従います。

```
dev->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD), //CPUからGPUへ転送する用
    D3D12_HEAP_FLAG_NONE, //特別な指定なし
    &CD3DX12_RESOURCE_DESC::Buffer(sizeof(vertices)), //サイズ
    D3D12_RESOURCE_STATE_GENERIC_READ, //よくわからない
    nullptr, //nullptrでいい
    IID_PPV_ARGS(&_vertexBuffer)); //いつもの
```

とりあえずこう記述してください。僕もまだ DirectX12 の全体的な使用を把握しきれてないので、あまり細かいところになると良く分かりません(DX11なら VERTEX_BUFFER とかの指定で OK だったんですけど…)

ちなみに D3D12_RESOURCE_STATE_GENERIC_READ の部分に「良く分からぬ」と書いたらやいましたが、これ、日本語に訳しても

「これは、アップロードヒープに必要な開始状態です。可能であれば、アプリケーションは通常この状態を避け、実際に使用されている状態にのみリソースを移行してください。」

とか非常に不穏なことを書いていますし。正直な話ここでサンプル頼みになっちゃうのは非

常に悲しいし、申し訳ないけど俺の力不足です。

ともかくこれで頂点バッファができました。あとリザルトは確認しておいてくださいね。でもまんが DEFAULT を指定すると、書き込み不可のため MAP が失敗するのでやっぱ UPLOAD にしておいてください。

でもよく考えてください。頂点バッファは作ったけど中身が入っていませんよね？ 雑に言うと 器は作ったけど空っぽなわけです。今からここに中身(頂点情報)をねじこんでいく必要があります。

これねえ… DX11 の時代は初期情報を Create の時点で突っ込むこともできたんですが、 DirectX12 は Map すること前提なので… ホンマにハードル上がつとるわ。で、Map って何がって言うとすでに作ったバッファに対してこちらから書き込みをするときに使います。この Map した段階で内部的には GPU 側からこのバッファの参照ができなくなるためある意味 Lock に近いかな～って感じです。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788712\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788712(v=vs.85).aspx)

何かというと、ロックしといてメモリの番地を貰つといて、そこに対して書き込みするわけです。

第一引数はインデックスなのでとりあえずは 0 でいいです。第二引数はちょっとややこしいんですが、そのメモリの内容を読み込んで利用する時にのみ意味があるものとなります。ということ

`D3D12_RANGE range = { 0, 0 };`

適当な値を入れておいて、第二引数にセットします(もしかしたらこいつは nullptr 入れておけばいいかも知れません)

そして最後の引数でポインタを取得するのですが、こいつの型が void** なので、正直何でもいいんですけど、char* が unsigned char* のポインタでも突っ込んでおけばいいです。

で、Map 関数が終わった時点で↑のポインタのアドレスに頂点座標を書き込めば GPU に投げるためのデータとなるわけです。

ただ、そうは言っても単なるデータの塊なので、結局 memcpy や std::copy などでメモリコピーをしてあげる必要があります(これが構造体変数 1 個なら memcpy や std::copy 使わなくても行けるんですけどね)

ともかく頂点データの内容を↑のバッファにコピーして終わったら Unmap してください。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788713\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788713(v=vs.85).aspx)

_vertexBuffer->Unmap(インデックス,書き込み範囲を表すポインタ);

というわけでインデックスは Map の時と同様に 0 でよく、第二引数も nullptr でオッケー。

とりあえずこれで頂点バッファはできました。だけどまだ終わらなくて、次はこれを頂点バッファビューにして GPU に投げれるようにします。

頂点バッファビュー

頂点バッファビューを宣言します。

```
D3D12_VERTEX_BUFFER_VIEW _vbView={};
```

頂点バッファビューってのは、頂点バッファの全体の大きさとか1頂点当たりの大きさとかを知らせるための付加情報と頂点バッファを紐づけて GPU に投げるためのものです。DX11 いう所の VERTEX_BUFFER_DESC みたいなもんです。

まずは頂点バッファの GPU におけるアドレスを記録しておきます。

```
_vbView.BufferLocation=_vertexBuffer->GetGPUVirtualAddress();
```

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903923\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903923(v=vs.85).aspx)

次にストライド(頂点1つ当たりのバイト数)を指定します。実はストライドって歩幅って意味なんだけど、次のデータまでの距離を表すわけです。これは簡単で sizeof 使えばいい。

```
_vbView.StrideInBytes = sizeof(Vertex);
```

次にデータ全体のサイズを伝えます。

```
_vbView.SizeInBytes = sizeof(vertices);
```

で、このビューを最終的にはコマンドリストにて

```
_commandList->IASetVertexBuffers(0,1,&_vbView);
```

てな投げ方をするんですが、それはもうちょっと後でやります。

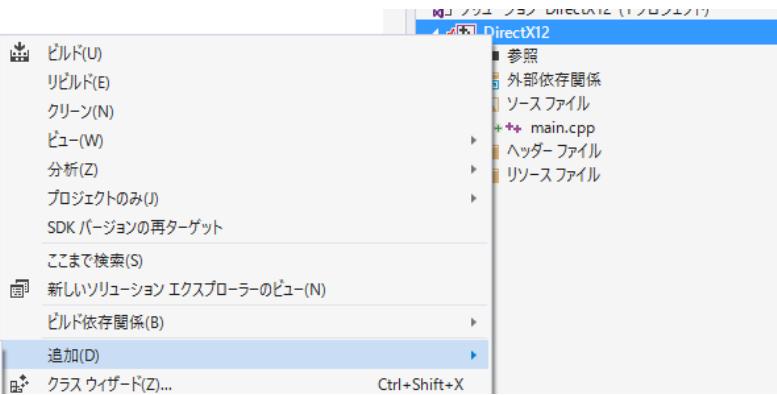
そんな事よりシェーダ書こうぜ

AD), //CPUからGPUへ転送する
3)), //サイズ
ない

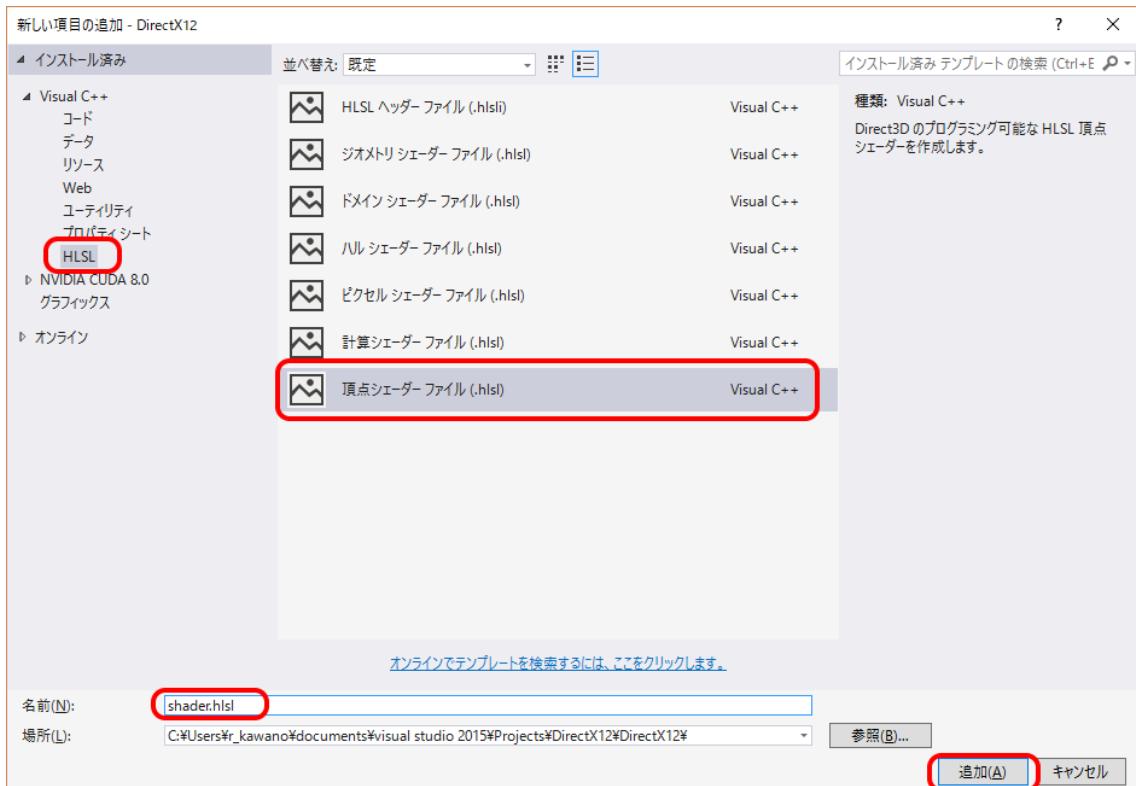
buff);

新しい項目(W)... Ctrl+Shift+A

存在する項目(O)... Shift+Alt+A



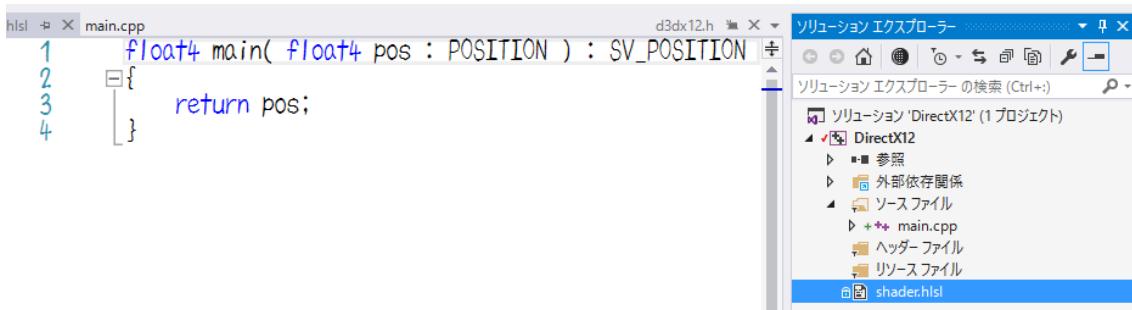
プロジェクトで右クリック→追加→新しい項目を選ぶと



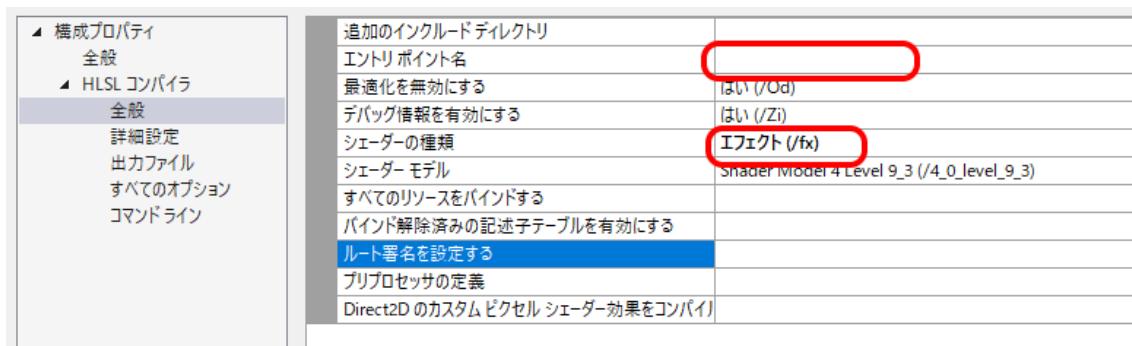
こんなのが出てくるので頂点シェーダファイルとして追加してください。とはいって実際には頂点シェーダとピクセルシェーダを共用しますので、VertexShaderではなく shader.hlsl にし

といてください→追加。

こうなるので, shader.hlsl で右クリック>プロパティ>



HLSL コンパイラ→全般



エントリポイント名を空白にして、シェーダの種類を「エフェクト」にしてください。これでピクセルシェーダを併用できます。

で、実は頂点シェーダは今までいいのでピクセルシェーダ自分で書いていきます。まだシェーダの書き方を知らないと思うのでこう書いてください。

あと、シェーダモデルは 5.0 にしてください。

で、コンパイルして通ればひとまずシェーダは大丈夫です。

とはいって、これは hlsl 側が終わったって意味で、C++ 側では今度はシェーダ読み込み処理を書かなければなりません。面倒ですね。

シェーダ読み込み

はい、久々のプロフですが、シェーダ用の宣言です。

```
ID3DBlob* vertexShader = nullptr;
ID3DBlob* pixelShader = nullptr;
```

次にこれに対してシェーダのコンパイルを行います。

```
result = D3DCompileFromFile(_T("shader.hlsl"), nullptr, nullptr, "BasicVS", "vs_5_0", D3DCOMPILE_DEBUG |  
D3DCOMPILE_SKIP_OPTIMIZATION, 0, &vertexShader, nullptr);  
result = D3DCompileFromFile(_T("shader.hlsl"), nullptr, nullptr, "BasicPS", "ps_5_0", D3DCOMPILE_DEBUG |  
D3DCOMPILE_SKIP_OPTIMIZATION, 0, &pixelShader, nullptr);
```

ちょっと長いんですけど、頑張って書いてください。

で、これ実行しようとするとリンクに怒られるので `d3dcompiler.lib` をリンクしてください。

さて、これで終わりと思うかね？まだまだですよ。あくまでも「シェーダをコンパイル」して「使える」状態にしただけなので、使ってあげないといけません。

ルートシグネチャー

またわけわから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)

こんなのが読んでも分からぬと思います。

ちなみに機械翻訳した結果…

ルートシグネチャ

ルートシグネチャは、グラフィックスパイプラインにどのタイプのリソースがバインドされているかを定義します。

このセクションで

<u>ルート署名の概要</u>	ルートシグネチャは app によって設定され、シェーダが必要とするリソースにコマンドリストをリンクします。グラフィックスコマンドリストは、グラフィックスと ルートルートシグネチャ の両方を有する。 計算コマンドリスト には、単純に 1 つの 計算ルート署名 があります。これらのルート署名は、互いに独立しています。
<u>ルート署名の使用</u>	ルートシグネチャは、ディスクリプターテーブル（レイアウトを含む）、ルート定数およびルート記述子の任意に配置された集合の定義である。各エントリには最大限のコストがかかります。そのため、アプリケーションでは、ルートシグネチャに含めるエントリの種類ごとにバランスを取ることができます。
<u>ルート・シグネチャの作成</u>	ルートシグネチャは、ネストされた構造を含む複雑なデータ構造です。これらは、以下のデータ構造定義（メンバの初期化を支援するメソッドを含む）を使用してプログラムで定義することができます。また、 HLSL (High Level Shading Language) で作成することもできます。これにより、レイアウトがシェーダと互換性があることをコンパイラが早期に検証できる利点があります。

<u>ルート署名の制限</u>	ルート署名は <u>プライム不動産</u> であり、厳格な制限と考慮する必要があります。
<u>ルートシグチャに直接定数を使用する</u>	アプリケーションは、ルートシグチャ内のルート定数をそれぞれ 32 ビット値のセットとして定義できます。それらは定数バッファとして HLSL(High Level Shading Language) で表示されます。歴史的な理由から、 <u>定数バッファは 4x32 ビットの値</u> の集合とみなされることに注意してください。
<u>ルートシグチャに直接記述子を使用する</u>	アプリケーションは、ディスクリプタヒープを通過することを避けるために、ディスクリプタを直接ルートシグチャに入れることができます。これらのディスクリプタは、ルートシグチャの領域（ルートシグチャの制限のセクションを参照）に多くのスペースを必要とするため、アプリケーションはそれを控えめに使用する必要があります。
<u>ルート署名の例</u>	次のセクションでは、複雑さが <u>空から完全完全に変化するルートシグチャ</u> を示します。
<u>HLSL でのルートシグチャの指定</u>	<u>HLSL でのルートシグチャの指定</u> Shader Model 5.1 は C++ コードでそれらを指定する代わりに使用できます。
<u>ルート署名バージョン 1.1</u>	Root Signature バージョン 1.1 の目的は、ディスクリプタヒープ内のディスクリプタが変更されないか、またはディスクリプタがポイントしても変更されない場合に、アプリケーションがドライバに指示できるようにすることです。これにより、ドライバが記述子またはそれが指すメモリが一定期間静的であることを知ることができる可能性のある最適化を行うことができるようになります。

Google 大先生翻訳がかなりの勢いでバグるほど難しいようです。



割と本気で殺しに来てるのがわかるだろう？

そこで

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

とか

<https://sites.google.com/site/monshonosuana/directxno-hanashi-1/directx-145>

を見ました。

ちなみに

プライム不動産→Prime real estate→「主要(重要?最優先?)な物理メモリ領域」とかそういうのだと思います。要は、ルートシグネチャも物理メモリを消費するので制限とコストについてしっかり考慮する必要がありますってことです。

ちなみに <https://docs.microsoft.com/ja-jp/windows/desktop/direct3d12/using-a-root-signature> をみると『各エントリには最大限のコストがかかります』と書かれているためメモリの管理には気を付けようぜという事だと思う。

ルートルートシグネチャ→compute root signature→単なる誤植っぽい。

空から完全完全→from empty to completely full→空っぽのやつから、完全にフルのやつまで…つまり、ルートシグネチャは中身空っぽの状態でも生成できるし、フルフルの状態もありうる。どちらの状況についても例を用いてご説明いたしますって事。

計算コマンドリスト、計算ルート署名→A compute command list will simply have one compute root signature.→これは、グラフィックス用のコマンドリストやらルートシグネチャに対して、コンピュートシェーダで使用するためのコマンドリスト(ID3D12CommandList)やらルートシグネチャがあるため、このような言い回しになっていると思います。

ちなみにルートシグネチャの説明として

『ルートシグネチャは、ディスクリプターブル(レイアウトを含む)ルート定数およびルート記述子の任意に配置された集合の定義である。各エントリには最大限のコストがかかります。そのため、アプリケーションでは、ルートシグネチャに含めるエントリの種類ごとにバランスを取ることができます。』←機械翻訳

おかしなのは、この概要の説明が、「概要」の部分になく「使用」の部分にあるんだよなあ。ちなみに概要の部分の説明では

『ルートシグネチャは app によって設定され、シェーダが必要とするリソースにコマンドリストをリンクします。グラフィックスコマンドリストは、グラフィックスとルートルートシグネチャの両方を有する。計算コマンドリストには、単純に 1 つの計算ルート署名があります。これらのルート署名は、互いに独立しています。』←機械翻訳

である。

ちなみにマニュアルの中の例…『空のルートシグネチャ』ってやつは使い物にならない上に、

恐らくそれすら作るのは簡単ではない。

正直とてもとてもややこしい(難しいわけではない…ややこしい。説明は難しいが…)

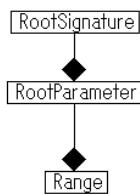
非常に作りが…本当に、本当にややこしい。

結局説明を入れても

<https://sites.google.com/site/monshonusuana/directxno-hanashi-1/directx-145>

と似たような説明になってしまふし、説明しても多分スルーしてしまう程わけわからんのです。

一応構造的には



こういうシンプルな構造にはなっています。で、RootParameter の1つ1つが、今まで何度か言っている DescriptorTable に当たります。正確には

```
D3D12_ROOT_PARAMETER rootParam = {};
rootParam.ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
rootParam.DescriptorTable.NumDescriptorRanges = 1;
```

こういう作りです。

コード見てもらった方が手っ取り早いと感じてこう書きました。実はルートパラメータというものは共用体状態になってて、DescriptorTable, Constants, Descriptor の三種のどれかの形態になっていて、どの形態になっているかは ParameterType で知らせるように作られています。

基本的に DescriptorTable を使っていくのですが、他の奴ももちろん知っておいた方がいい…でもとりあえず今は飛ばしていきます。知りたい人は

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

の解説を見ておくとよいと思います。

で、DescriptorTableについて色々と資料を漁りました。

<https://www.slideshare.net/DevCentralAMD/introduction-to-dx12-by-ivan-nevraev>

<https://software.intel.com/en-us/articles/introduction-to-resource-binding-in-microsoft-directx-12>

ちなみに DescriptorTable と DescriptorHeap は直接関連しているわけではありません。
これは知っておいてください。

「ディスクリプタヒープの主な目標は、できるだけ多くのレンダリングのためにすべてのディスクリプタを格納するために必要なだけ多くのメモリを割り当てます。」

「ディスクリプタ・テーブルは、ディスクリプタ・ヒープにオフセットします。ディスクリプタテーブルを切り替えることで、ヒープ全体を常に表示するようにグラフィックパイプラインを強制するのではなく、特定のシェーダが使用するリソースセットを変更するための安価な方法です。この方法では、シェーダはヒープ空間内のリソースをどこに見つけるかを理解する必要はありません。」

「言い換えれば、アプリケーションは、図 2 に示すように、異なるシェーダの同じディスクリプタヒープをインデックスする複数のディスクリプタテーブルを利用できます。」

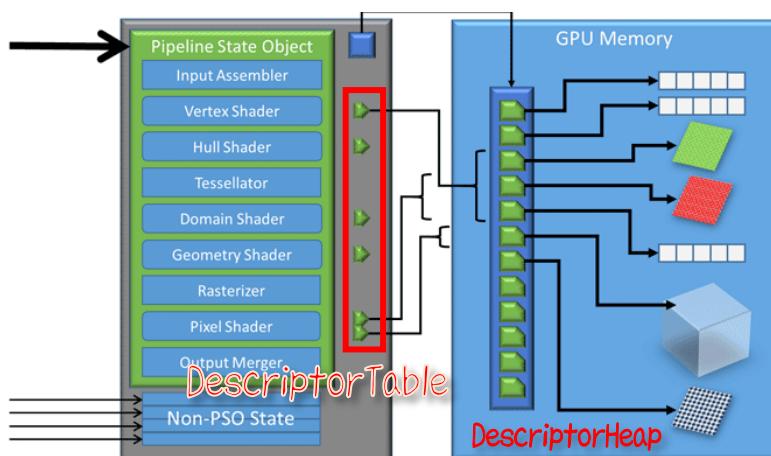
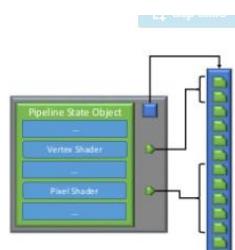


図2

ちょっと付け足してみたんだけど、DescriptorTableってのが、左側にある五角形のアレの事ですね。

Descriptor Tables

- Context points to active heap
- A table is an index and a size in the heap
- Not an API object
- Single view type per table
- Multiple tables per type

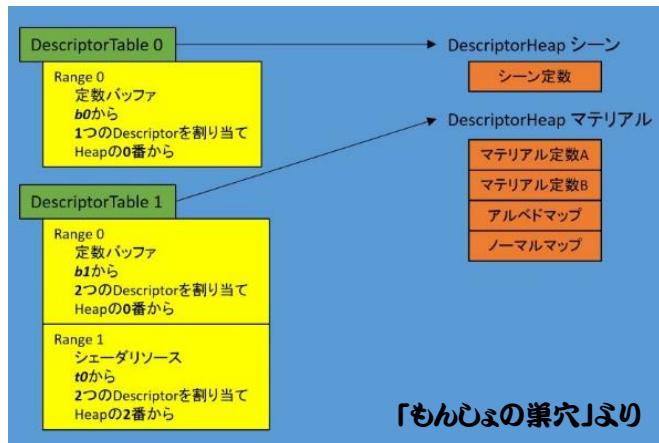


で、ディスクリプタテーブルはレンジと言うものを持ってて、これによってヒープの特定の場所にアクセスするんだけど、ヒープの場所の~(どつかアドレス)ではなくシェーダレジスタ番号の指定なんだよね。

ちょいイメージが違うんだわあ…。

まあ、レジスタ番号指定メンバの名前が `BaseShaderRegister` って名前だからまあ、そういう事なんだろうって察しはつくんですけどね。

もんじょ氏の図はもう少しだけ分かりやすくて、



アルベドとかノーマルは忘れて構造だけ見てね

まあ、マテリアルとマップがこんなお行儀よく並んでるかと言うとちょっと疑問なんだけど。まあ DirectX12においては「まとめて扱いたいから並べろ」って事かな。ちなみに「マテリアル定数」は定数バッファ(CBV)でアルベドとかノーマルはテクスチャ(SRV)です。別の種別のバッファが並んでるわけです。

で、レンジにはレンジには `RangeType` というメンバがあり、そいつがシェーダの種別を持っている(こちらで指定する)という構造になっている。

で、なんで「レンジ」かというと、Range…範囲と言う意味で、DX11までだったらこういう指定って一つ一つだったんですが、それらをまとめて扱えるようにして、いっぺんに指定するため「範囲」という意味の Range が使われているのだと思います。

変数名を決めた「思想」まで慮らないと、意味が分からぬしくみ…それが DirectX12

『DirectX12 沼へようこそ』

それはともかく何となく構造が見えてきたところで早速プログラミングしていきましょう。今回は単純なシェーダ2つだけなのでそれほど複雑にならないと思います。

```
ID3D12RootSignature* rootSignature=nullptr;//これが最終的に欲しいオブジェクト
ID3DBlob* signature=nullptr;//ルートシグネチャをつくるための材料
ID3DBlob* error=nullptr;//エラー出た時の対処
```

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。

ルートシグネチャ本体を作るためには…

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。入力アセンブラー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「入力アセンブラー(IA)」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です。ちなみに `1_1` を使用すると今の所失敗します。たぶん他に色々設定する必要があるのでしょうが、これ以上ここに時間をかけたくないので先に進みましょう。

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。

でもし、シェーダに対して値とかテクスチャとか渡すようになると、最初に言った `Range` とかなんとか必要になってきますが、それは必要になってからまた説明します。

頂点レイアウト

頂点レイアウトって何？

これはデータの塊がどういう意味を持つのかを知らせるものです。CPUの世界ではご覧のように `Float3` つで、頂点の座標を示しているのは分かってるんですが、GPUに投げられた時には

単なるバイトデータの塊なのです。

例えば↑のデータならこんな感じに見えます。

「ウフフフフフ…フフフフフ…ヤ…ウフフフフ」

なにわろとんねん。怖いわ。というわけで、これでは使い物にならんわけです。かといってテキストで投げたら GPU にとってはもっとワケわからんのです。ここで出てくるのが…

「このデータはこういう風に扱ってや」というデータ上で頂点情報がどのようにメモリ上にレイアウト(配置)されているのかを示す「頂点レイアウト」なのです。これを頂点情報とともにGPUに投げることによって、頂点情報をxyzとして認識できるわけです。

さて次に頂点レイアウトの定義ですが

D3D12_INPUT_ELEMENT_DESC

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770377\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770377(v=vs.85).aspx)

で定義します。これも DX11 のやつを参考に見てみます。

[https://msdn.microsoft.com/ja-jp/library/ee416244\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416244(v=vs.85).aspx)

まず、分らない用語が出てきます。

「セマンティクス」ってなんや?

初めて聞く言葉だと思いますが、これは「データの意味付け」くらいに思っておいたらいいです。

「HLSL セマンティクス」で検索すると

[https://msdn.microsoft.com/ja-jp/library/bb509647\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509647(v=vs.85).aspx)

POSITION(座標)だの COLOR(色)だのが出てきます。

実は POSITION も COLOR もどちらもシェーダにわたってくるときには float4 つぶんと表されます。

POSITIONならxyzw,COLORならrgbaですね。

まあ最初は POSITION のみでいいです。

SemanticIndex はしばらく 0 でいいです。同一セマンティクス要素は出でこないので。
次の DXGI_FORMAT ですが、これは FLOAT いくつ分のデータとかそういうのを記述します。
FLOAT 三つ分なので

[https://msdn.microsoft.com/ja-jp/library/ee418116\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee418116(v=vs.85).aspx)

を見ながら

DXGI_FORMAT_R32G32B32_FLOAT

を指定します。

この辺が面倒なのですが X32Y32Z32 なんという指定はないのです。GPU まわりは XYZ も RGB として表現したりしますので、そういうのにもう慣れてください。

次に入力スロットですが、これは 0 でいいです。そのうちスロットを複数使いますが、しばらくは 0 スロットしか使わないで 0 でいいです。

[https://msdn.microsoft.com/ja-jp/library/bb205117\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb205117(v=vs.85).aspx)

とか

http://marupeke296.com/DX10_No2_RenderBillboard.html

にスロットの話とかが書かれてますので、興味のある人は良く読んでおきましょう。

AlignedByteOffset は D3D11_APPEND_ALIGNED_ELEMENT を指定しておいてください。本来は数値を設定するのですが、それだとあまりにも面倒なんで。

次に InputSlotClass ですがこれも

D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA

を指定します。

最後の引数は 0 にしてください。それはヘルプに明記されています。

で、この頂点レイアウトは『配列にすべきものです。つまり今まで書いたのを構造体の配列にするように定義してください。』

// 頂点レイアウト

```
D3D12_INPUT_ELEMENT_DESC inputLayoutDescs[] = {
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_
    INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
};
```

あとはパイプラインステートにて

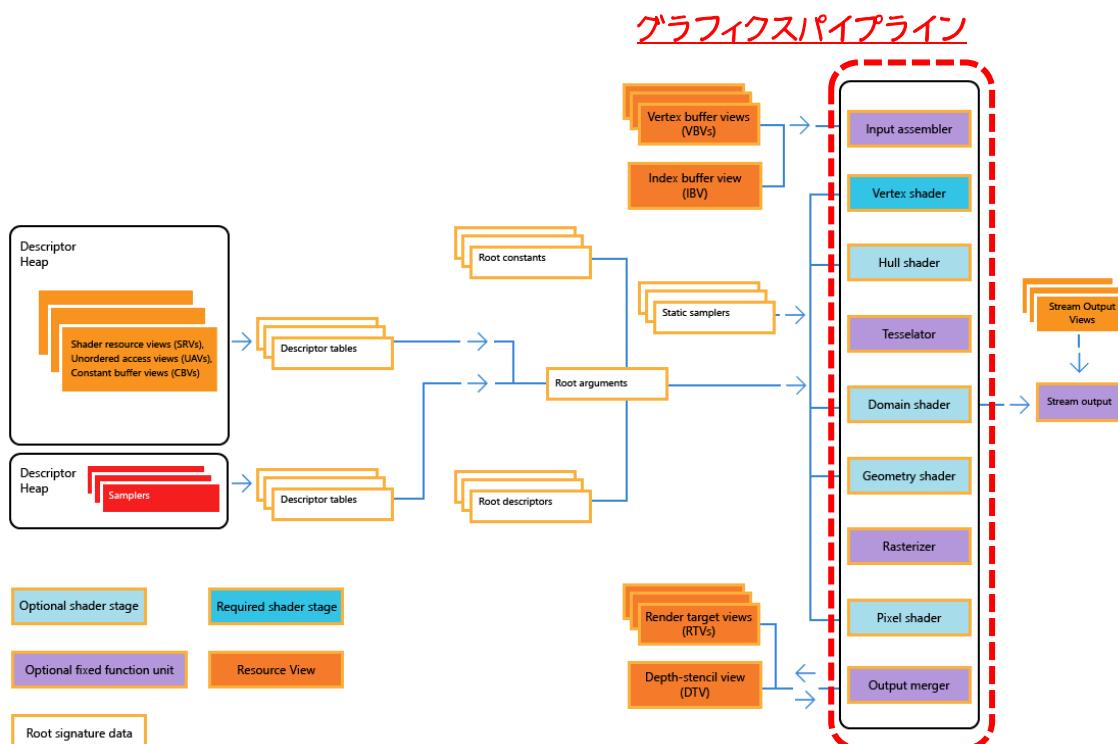
```
gpsDesc.InputLayout.pInputElementDescs = inputLayoutDescs; //  
gpsDesc.InputLayout.NumElements = _countof(inputLayoutDescs); //  
とでもしてやれば(^ω^)おつけ。
```

パイプラインステートオブジェクト(PSO)

さて、前にもちょっとだけ出て来てた「パイプラインステートオブジェクト(PSO)」を初期化していきましょう。

名前の通りパイプラインに関する情報をパンパン突っ込んでいきます。個人的にはあまりパイプラインと直接関係ないもの混ざっている気がするんですが…

まずグラフィクスピープラインと言うのは



↑こういうものでしたよね？

<https://docs.microsoft.com/en-us/windows/desktop/direct3d12/pipelines-and-shaders-with-directx-12> より

前にも言いましたが、DirectX12 は DirectX11 では「ラバーラ」になっているものくつつけたがる設計思想なのだ。

ステート系ってのはここまで話で具体的に言うと…このレンダリングパイプラインに関わる部分をまとめたものという事だ。

シェーダ系

- 頂点シェーダ(VS)
- ピクセルシェーダ(PS)
- ハルシェーダ(HS)
- ドメインシェーダ(DS)
- ジオメトリシェーダ(GS)

です。それに加えて

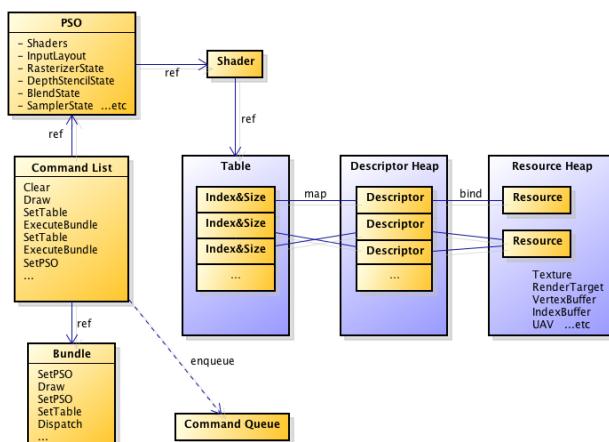
- 頂点レイアウト
- ラスタライザーステート
- ブレンドステート
- デプスStencilステート
- トポロジータイプ
- その他色々

あと、ルートシグネチャと関連付ける必要があるので

- ルートシグネチャ

ちなみに比較的、パイプラインステートとルートシグネチャの関係の分かりやすい図として

http://f.hatena.ne.jp/shuichi_h/20150502164707 の



あります。ああ～、なんとなく構造がわかつてくるですよ～。

これらの情報を

D3D12_GRAPHICS_PIPELINE_STATE_DESC 変数に入れておいて

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770370\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770370(v=vs.85).aspx)

とはいえ、正直クソ多いので、必要なものだけ入れておきます。

VS,PS,RasterizerState,BlendState,DepthStencilState,SampleMask,PrimitiveTopologyState,Num
RenderTargets,0番レンダーターゲットビューフォーマット、サンプルカウント
など…とはいえる初心者が分かる部分ではないので

```
VS=C3DX_SHADER_BYTECODE(vs);
PS=C3DX_SHADER_BYTECODE(ps);
RasterizerState=C3DX_RASTERIZER_DESC(D3D12_DEFAULT);
BlendState=CD3DX_BLEND_DESC(D3D12_DEFAULT);
DepthStencilState.DepthEnable=false;//今はファルスで
DepthStencilState.StencilEnable=false;//今はファルスで
D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE
レンダーターゲット数1
フォーマットは R8G8B8A8_UNORM
サンプルカウントは1で
```

あとはパイプラインステートを CreateGraphicsPipelineState でパイプラインステートを作
って…セットするだけです。これはコマンドリストのリセットの際に第二引数にパイプライ
ンステートを入れておけばいいのです。

CreateGraphicsPipelineState

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788663\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788663(v=vs.85).aspx)

で、以下のように書いてください。

//ルートシグネチャと頂点レイアウト

```
gpsDesc.pRootSignature = _rootSignature;
gpsDesc.InputLayout.pInputElementDescs= inputLayoutDescs;//
gpsDesc.InputLayout.NumElements = _countof(inputLayoutDescs);//;
```

//シェーダ系

```
gpsDesc.VS = CD3DX12_SHADER_BYTECODE(vsBlob);
gpsDesc.PS = CD3DX12_SHADER_BYTECODE(psBlob);
//使わない
```

```
//gpsDesc.HS;
//gpsDesc.DS;
//gpsDesc.GS;
//レンダーターゲット
gpsDesc.NumRenderTargets = 1; //注)このターゲット数と設定するフォーマット数は
gpsDesc.RTVFormats(0) = DXGI_FORMAT_R8G8B8A8_UNORM; //一致させておく事
```

注意点ですが設定したレンダーターゲット数以上にRTVFormatは設定しないでください。やっちはまいました。NumRenderTarget=1にした状態で8枚全部RGBAで設定したらエラります。とりあえず1枚なら0番目のみ設定しつければいいです(デフォルトはDXGI_FORMAT_UNKNOWN)

```
//深度ステンシル
gpsDesc.DepthStencilState.DepthEnable = false; //あとで
gpsDesc.DepthStencilState.StencilEnable = false; //あとで
gpsDesc.DSVFormat; //あとで

//ラスタライザ
gpsDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);

//その他
gpsDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
gpsDesc.NodeMask = 0;
gpsDesc.SampleDesc.Count = 1; //いる
gpsDesc.SampleDesc.Quality = 0; //いる
gpsDesc.SampleMask = 0xffffffff; //全部1

//gpsDesc.Flags; //デフォルトでOK
gpsDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE; //三角形
ID3D12PipelineState* _pipelineState=nullptr;
result = dev->CreateGraphicsPipelineState(&gpsDesc, IID_PPV_ARGS(&_pipelineState));
で、これでS_OKが返ってこない場合はシェーダとかレイアウトとかが間違えていると思いま
すので、確認しておいてください。
```

その他やらなければならぬ事

あともうちょっと…あともうちょっと我慢してくれ。もうすぐポリゴン出るから。
ここからやらなければならぬことは

- パイプラインステートをセット(コマンドリストのリセット時)
- ルートシグネチャをセット(SetGraphicsRootSignature)
- ビューポートのセット(RSSetViewports)

くらいなのだが、これに加えて、以前画面クリアするときには使ってない描画という処理を使ってるので、またちょっとだけ面倒なことをやらなければならない。
それは

『リソースバリア』である。

リソースバリア

これもフェンスと同じように、特定のリソースに対して読み込みと書き込みが同時に行われないようにする仕組みです。

で、今回ひとまずバックバッファリソースにバリアをかけておくため

ResourceBarrier

```
https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903898
_cmdList->ResourceBarrier(1,&CD3DX12_RESOURCE_BARRIER::Transition(_renderTargets[バックバッ
ファーム番号],D3D12_RESOURCE_STATE_RENDER_TARGET,D3D12_RESOURCE_STATE_PRESENT));
つまり
_cmdList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(_backBuffers(bbIdx),
        D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_PRESENT));
_cmdList->Close();
```

を使います。これをコマンドリストの一番最後に呼び出します。

あとはそれぞれの処理をこなしていく。

ビューポート

ビューポートってのは、これまでの話に比べると比較的簡単で、ディスプレイに対してレンダリング結果をどのように表示するかというものです。これは内部でレンダリング画像を『ビューポート変換』して、画面に表示しています。簡単ですのでやっていきましょう。

ん~、シンプルに言うとどこからどこまでの範囲にレンダリングするかってのを指定するものです。

必要なものは画面のサイズ…そしてデプスですが、たぶん良く分からぬと思いますので、デプスに関しては今は言うとおりにしてください。

RSSetViewportsって関数を使います。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903900\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903900(v=vs.85).aspx)

これはDX11と同じなのでそっち見て考えましょう。

[https://msdn.microsoft.com/ja-jp/library/ee419744\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419744(v=vs.85).aspx)

はい、今回は表示先はひとつだけなのでNumViewportsは1にしておいてください。

んで、ビューポート(D3D12_VIEWPORT)を普通に構造体オブジェクトとして作ってそのポイントを渡してください。

D3D12_VIEWPORTの指定は

[https://msdn.microsoft.com/ja-jp/library/ee416354\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416354(v=vs.85).aspx)

見てもらえば入れるものは分かると思います。左上は0,0でいいです。

で、MinDepth=0,MaxDepth=1にしておいてください。

```
//ビューポート設定
_viewport.TopLeftX = 0;
_viewport.TopLeftY = 0;
_viewport.Width = wsize.w;
_viewport.Height = wsize.h;
_viewport.MaxDepth = 1.0f;//カメラからの距離(遠いほう)
_viewport.MinDepth = 0.0f;//カメラからの距離(近いほう)

//なんとかシザー(切り取り)矩形も必要
_scissorRect.left = 0;
_scissorRect.top = 0;
_scissorRect.right = wsize.w;
_scissorRect.bottom = wsize.h;
(中略)
```

//ビューポートとシザーセット

既に作っているパイプラインステートオブジェクトをセット
→コマンドリストのリセット時に既に作っているパイプラインステートオブジェクトを入れる。

ルートシグネチャーをセット

既に作っているルートシグネチャーを

_commandList->SetGraphicsRootSignature

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788705\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788705(v=vs.85).aspx)

でセット。

ビューポートをセット

RSSetViewports

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903900\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903900(v=vs.85).aspx)

でセット

でもう一つ設定しなければならないんだけど、シザーっていうやつで、画面をどう切り取るかの指定もしなければならない。正直めんどいんだけど、これをやらないと表示されない。

RSSetScissorRects

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn903899\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn903899(v=vs.85).aspx)

これは聞いたって簡単。left,top,right,bottom を設定すればいいだけ。左上は 0 でいいから…あとはわかるな?

ここまでではいいんだが、最後にもう一つ、頂点バッファのセットも必要である。

[https://msdn.microsoft.com/ja-jp/library/ee419692\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419692(v=vs.85).aspx)

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn986883\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn986883(v=vs.85).aspx)

これも DX11 と同じなのでこれを見ながら

スロットは 0 でいい。Numbuffers は 1

で、次の引数に頂点バッファビューをセット。

そこまで終わったら

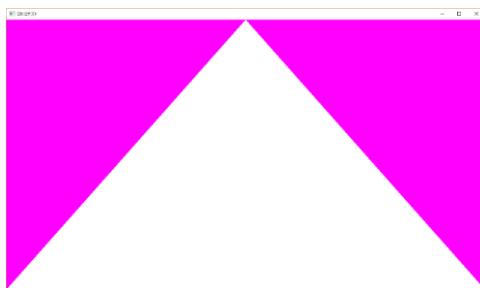
ドロー!!ポリゴン!!!

[https://msdn.microsoft.com/ja-jp/library/ee419594\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419594(v=vs.85).aspx)

_commandList->DrawInstanced(頂点数,インスタンス数,0,0)

で描画します。インスタンス数ってのは同じ奴をいくつも書くときに入れる奴なので、1でいいです。

うまくいけば…



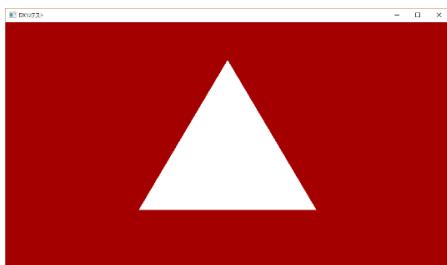
こんな感じの表示になります。右上の白い部分が今回描画している三角形です。

ここまでのかつてを考えると意外とあっさりですね…

例えば頂点をちょっといじると

```
Vertex vertices[] = { { { 0.0f, 0.7f, 0.0f } },
{ { 0.4f, -0.5f, 0.0f } },
{ { -0.4f, -0.5f, 0.0f } } };
```

こうなります。



さらにシェーダをいじって色を変えてみましょう。

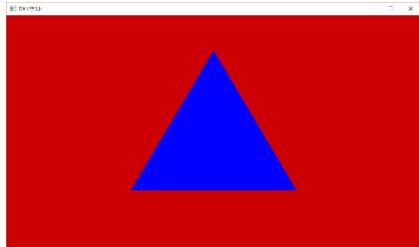
ピクセルシェーダの

```
return float4(1,1,1,1);
```

の部分を

```
return float4(0,0,1,1);
```

とすると



こうなります。でもこれは予想できて面白くない…。

というわけで、こう書いてみてください。

```
struct Out {  
    float4 svpos : SV_POSITION;  
    float4 pos : POSITION;  
};
```

//頂点シェーダ

```
Out BasicVS( float4 pos : POSITION )  
{  
    Out o;  
    o.svpos = pos;  
    o.pos = pos;  
    return o;  
}
```

//ピクセルシェーダ

```
float4 BasicPS( Out o):SV_Target  
{  
    return float4((o.pos.xy+float2(1,1))/2,1,1);  
}
```

どうなりました?



はい、勝手にグラデーションがつきました。これがシェーダの面白さです。

シェーダの構造体とかあとメンバに関しては C++ よりかなり柔軟です。いや C++ でも似たようなことはできるんですが、かなり面白いと思います。面白いと思ってほしいなあ。

何でこうなるかを考えてほしいのですが、ちょっと面倒なのは SV_POSITION のまま使おうとするとグラデがつかないことがあります。

たかだかポリゴン1枚出すのに100ページ越えちゃったよ…。やっとここまで **パンナムブートキャンプ1日目終了** っすよ。これは確かに新人にはキツイわあ…。

うまくいかない場合

ちなみに、ここまで正しくやってるのに三角形表示されないことがあります。原因としてグラボが対応していないことがあります(ノートの場合はインテル長友 HD グラフィックスになってるとか…ドライバの問題だったりするので、その場合は出力 exe ファイルに右クリックで高機能 GPU を割り当てるよう設定してテストしてみてください)。

すまんけどノートの HD グラフィクスの検証はちょっとやってないんですね(自前のノート PC も教務室のデスクトップ PC も nVidia しか選択されない仕様のため…ぶっちゃけめんどう)。



本当にすまない

デバイスの生成もできるし対応はしてるはずなんだけど、要所要所で挙動が違うんですね。検証にかける時間はちょっとないんですね…。

ちなみに、デバイスが複数ある場合に、こちらから(アプリケーション側から)グラボを選ぶといふのはできなくもない。

アプリが「グラボを選ぶズエ…レリズエ…」

面倒なんだけど、dxgifactory を作った後に「使用されているグラフィクスアダプタを列挙」という関数があるんで、列挙された中からええグラボを取りに行くという事はできます。

やり方はというと、今の CreateDevice の第一引数が nullptr なんだけど、そいつをきちんと指

定する。どうするかと言うとグラフィクスアダプタを列挙する。

```
std::vector<IDXGIAAdapter*> adapters;
IDXGIAAdapter* adapter = nullptr;
for (int i = 0; _dxgiFactory->EnumAdapters(i, &adapter) != DXGI_ERROR_NOT_FOUND; ++i) {
    adapters.push_back(adapter);
}
この中から NVIDIA の奴を探す
for (auto adapt : adapters) {
    DXGI_ADAPTER_DESC adesc = {};
    adapt->GetDesc(&adesc);
    std::wstring strDesc = adesc.Description;
    if (strDesc.find(L"NVIDIA") != std::string::npos) {//NVIDIAアダプタを強制
        adapter = adapt;
        break;
    }
}
```

そのアダプタを使って CreateDevice する。

```
D3D12CreateDevice(adapter, 1, IID_PPV_ARGS(&_dev))
```

いいね?

もうちょっと後になると怖い怖いテクスチャマッピングがお前らを襲うっ!!でもその前にその貼り付けるためのサーフェイスを作りましょう。

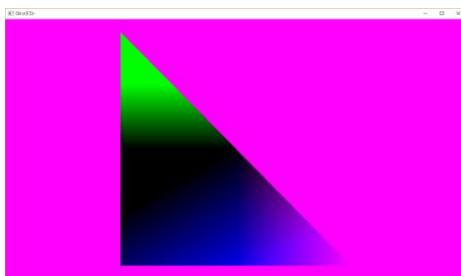
四角形の描画をしよう

まず、4頂点にしましょう。

//頂点バッファ生成

```
Vertex vertices[] = {
    XMFLOAT3(-0.5, -0.9, 0),
    XMFLOAT3(-0.5, 0.9, 0),
    XMFLOAT3(0.5, -0.9, 0),
    XMFLOAT3(0.5, 0.9, 0),
};
```

でも、これをそのまま表示したとしても



4角形にはなりません。

_cmdList->DrawInstanced(4, 1, 0, 0);

とやっても同じです。ではどうしましよう…? 2つくらいやりようがあります。

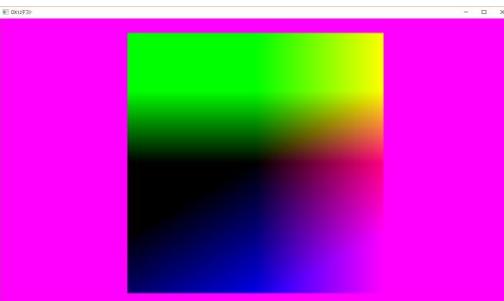
まずは、TRIANGLESTRIPにしてしまう事です。

_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);

通常は三角形と言えば三角形の集合 TRIANGLELIST がメジャーなのですが手っ取り早く矩形を表示するために TRIANGLESTRIP ってのが使われます。

<https://docs.microsoft.com/en-us/windows/desktop/direct3d9/triangle-strips>

簡単に言うと頂点を N 字もしくは Z 字の順になるように並べていくことでひとつながりの三角形を表現できるものです。



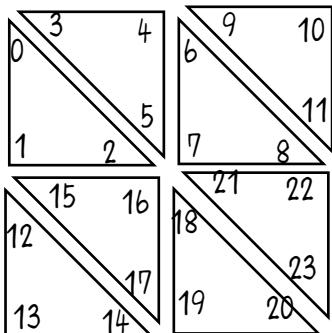
ね？簡単じゃろ？簡単だよなあ？

そして簡単すぎてゲイないので、TRIANGLELIST のまま4頂点で4角形になる方法を考える… 実はこ↑こ↓が重要で…後々お世話になる「インデックス」の概念を学ぶのだ。

と言いつつよく考えたら DxLib の頃からやってたよなあ…

インデックス情報の設定と GPU 転送

モデルデータなどは頂点の集合で、そして無数にあります。例えば



TRIANGLELIST で頂点を並べるとこういう図のように頂点が重なってしまうのですよ。真ん中の頂点なんか 2,5,7,16,18,21 の 6 頂点が重複することになります。頂点 1 つ当たりの情報は「座標3」「法線3」「UV2」「その他」で、一頂点当たりの情報がかなり多いんですね。ということで、頂点情報を減らすため、もつといふ一度に動かす頂点を減らすために、インデックスという番号で三角形の構成情報を渡すという方式を取っています。

なので、例えば単なる矩形(4角形)ですら 6 頂点必要なんですが 4 頂点で済ますためにはインデックスデータがあればいい。UI に使用する矩形であれば TRIANGLESTRIP でもいいんですが、モデルではインデックスありきなので、まあ、練習の意味もこめてインデックスにしてみましょう。

さて、すでに 4 頂点ありますから、頂点番号は 0,1,2,3 です。あ、いったん TRIANGLELIST に戻してくださいね。で、頂点 4 つを組み合わせて 2 つの三角形を作ってください。なお、インデックスは 6 個です。

この時、全ての三角形が時計回りになるように気を付けておいてください。ひっくり返ると面倒な事になります(描画されないんじゃないかな…)

さて、実際のインデックス情報も GPU のお気に召すような状態にしてあげなきゃいけません。

インデックス配列を作る

簡単すぎるか…

```
std::vector<unsigned short> indices = {0,2,1,2,3,1};
```

インデックスバッファを作る

頂点の時と同様に ID3D12Resource*でインデックスバッファ用変数を作つておいてください。

あと、頂点の時と同様に

```
result = dev->CreateCommittedResource(&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
                                         D3D12_HEAP_FLAG_NONE,
                                         &CD3DX12_RESOURCE_DESC::Buffer(indices.size() * sizeof(indices[0])),
                                         D3D12_RESOURCE_STATE_GENERIC_READ,
                                         nullptr,
                                         IID_PPV_ARGS(&indexBuffer));
```

で、これも頂点の時と同様にメンバ変数に `D3D12_INDEX_BUFFER_VIEW` の型の変数を作つてください。

そいつに

```
_ibView.BufferLocation = idxBuff->GetGPUVirtualAddress(); //バッファの場所
_ibView.Format = DXGI_FORMAT_R16_UINT; //フォーマット(shortだからR16)
_ibView.SizeInBytes = indices.size() * sizeof(indices[0]); //総サイズ
こんな感じで必要な情報を代入します。
```

できたらインデックス情報を転送しといてください。そこは自分でお願いします。やり方は頂点の時と同じです。

インデックスバッファをセット

`IASETINDEXBUFFER` 関数を使用して、インデックス情報をセットします。

```
_cmdList->IASETINDEXBUFFER(&_ibView);
_cmdList->IASETPRIMITIVE_TOPOLOGY(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

ドロー(インデックスあり)

簡単です。`DrawIndexedInstance` を使えばいい。

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

第一引数がインデックス数。第二引数がインスタンス数である事が分かれば…わかるじやろ？

テクスチャ貼りたいなあ…

さて、そろそろテクスチャを貼りたくなってきたと思います。バンナムブートキャンプの2日目です。テクスチャってのは絵の事です。

まあ、絵は何でもいいんですけどね。別に



こんなにも

それはともかくテクスチャを貼りましょう。

うん、「また」なんだ。すまない。

テクスチャ…めんどくせえんだ。DirectX11までとはわけが違うんだ。シェーダやデスクリプタは勿論ルートシグネチャとか絡んでくるんですわこれが。

まずは貼れる準備をしましょう

頂点情報に UV を追加

では手始めに Vertex 構造体に uv を追加しましょう。

XMFLOAT2 uv;

UV は 2float なので XMFLOAT2 を使用します。

初期化の部分は uv の情報を増やしておいてください。できるでしょ？

モチロンレイアウトも増えますので追加します。

```
{ "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
```

こうなるとモチロン頂点シェーダも変更しなきゃいけなくて…

頂点シェーダ変更

前にも言った通りグラフィックスパイプラインは簡略化して書くと
頂点情報→頂点シェーダ→ピクセルシェーダ→出力
で流れになってて「パイプライン」というのが、前のステージの出力が次のステージの入力
になっているわけで、今、頂点情報を変更したわけですからシェーダの方も uv の入力を追加する
必要があります。

やり方はとっても簡単。頂点シェーダに UV の項を追加するだけ

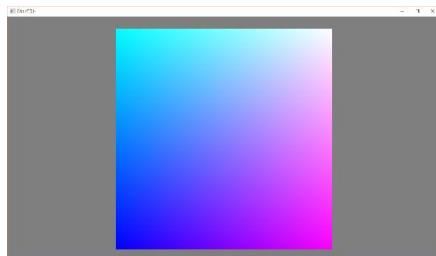
```
vs( float4 pos : POSITION ,float2 uv:TEXCOORD)
```

これで OK

ついでにピクセルシェーダ側もいじってみましょう。せっかく uv が入ってきたので色に反映させましょうか。出力構造体に uv を追加しといて…

```
return float4(output.uv.x,output.uv.y,1,1);
```

と書けば



こんな感じになるはずです。

テクスチャオブジェクト生成

はい、uv を作ったのは伊達や酔狂でもなんでもなく、画像を貼り付けるためです。そのためには GPU に流せる「テクスチャデータ」を作つてあげる必要があります。纂しのいい人はすでに分かってると思いますが… そう D3D12Resource です。

今一度

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

を見ましょう。

説明が

「ヒープがリソース全体を格納するのに十分な大きさでリソースがヒープにマップされるように、リソースと暗黙のヒープの両方を作成します。」←機械翻訳

「Creates both a resource and an implicit heap, such that the heap is big enough to contain the entire resource and the resource is mapped to the heap.」←原文

頂点シェーダの時とより、ちょっとパラメータの設定が面倒です。

CD3DX～一発ではなく、いちいちパラメータに入れるヒーププロパティとかリソースデスクリプションとか自分で書いてあげます。

// ひとまずこの通りに書いてください。

```
D3D12_HEAP_PROPERTIES heapprop = {};
heapprop.Type = D3D12_HEAP_TYPE_CUSTOM;
heapprop.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_WRITE_BACK;
heapprop.MemoryPoolPreference = D3D12_MEMORY_POOL_L0;
heapprop.CreationNodeMask = 1;
heapprop.VisibleNodeMask = 1;
```

色々ネットサーフィンしてペニーワイズとかチャーハンついでに DirectX12 のヒーププロパティについて調べましたが、少なくとも日本語のサイトで、これらの設定についてきちんと考察、記載しているサイトはありませんでした。ほぼみんな同じコードなので、サンプル丸写しかなと邪推しています。

一応公式のリファレンスはこれ

https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/ns-d3d12-d3d12_heap_properties

L0 とか L1 ってあのキャッシュの事ちゃうのん? CPU ページプロパティ? ページングのこと? だからページプロパティってなんなんですかあ! ??ごめんちょっと泣きそうなレベルで分からぬし。



恐らく推測としては CPU におけるページングだの L0 だの L1 だのの話なんだろうけど、説明

がほとんどないから本当に分からない。ここからそのページングとか L0L1 の詳細な話をするのかと!!! 時間ねーよ!!!!

それでもいちおう詳しいのはここかな。

https://shikihiiku.wordpress.com/2015/04/01/available_combinations_of_flags_when_allocating_resources_in_dx12/

ちなみに今回の場合だと

『D3D12_CPU_PAGE_WRITE_BACK を用いた場合

POOL_L0 のみリソース確保可能。

GPU で有効なアドレスを保持し、Map 可能です。

対象のアドレス領域には、WRITE_COMBINE や NOCACHE フラグが適用されていないので、一般的な writeback(通常のキャッシュが有効なメモリ)と考えられます。』

なぜこれを選ぶのか…正直わかりませんが、他の組み合わせでやろうとすると S_OK 出ないし、出たとしてもテクスチャ利用時に書き込めないとそういうのが発生してしまい、正直これに関しては分からんです。勉強ですまんがここはもう書かれてる通りにしてくれ。

https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/ne-d3d12-d3d12_cpu_page_property

ライトバックとか言われても…意味が分からん。

まあ

<https://www.weblio.jp/content/%E3%83%A9%E3%82%A4%E3%83%88%E3%83%90%E3%83%83%E3%82%A4>

F

を見れば

『CPU と記憶装置の間で記録を一時的に保管するキャッシュメモリーの動作方式のひとつで、CPU が記憶装置にデータを書き込む際、いったんキャッシュメモリーにデータを書き込み、処理の空き時間ができてからキャッシュメモリーからメインメモリーに書き込む方式のことである。

キャッシュと記憶装置に同時にデータの書き込みを行うライトスルー方式に比べて、キャッシュが記憶装置よりも記憶動作が高速であるという特性を活かすことができ、動作が高速になるという長所があるが、キャッシュとメインメモリーとの間でデータが必ずしも

整合するとは限らず、ライトスルー方式よりも制御が多少困難になるという短所もある。

まあ、これがテクスチャデータ作成にどう関わってるのか分からんのだが…ともかく重ね重ね申し訳ないのだがここだけはサンプルの通りにしておいてほしい。

ということでヒーププロパティ D3D12_RESOURCE_DESC を設定していきます。

https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/ns-d3d12-d3d12_resource_desc

D3D12_RESOURCE_DIMENSION Dimension; //何次元テクスチャか(TEXTURE2D)	
UINT64 Alignment; //先頭からなので0	
UINT64 Width; //テクスチャ幅	
UINT Height; //テクスチャ高さ	
UINT16 DepthOrArraySize; //リソースは2Dだし配列でもないので1	
UINT16 MipLevels; //ミップ使うまでは0レベル	
DXGI_FORMAT Format; //例によって R8G8B8A8_UNORM	
DXGI_SAMPLE_DESC SampleDesc; //count1, quality0	
D3D12_TEXTURE_LAYOUT Layout; //わかりません(UNKNOWN)というより決定できない状態	
D3D12_RESOURCE_FLAGS Flags; //NONE(ほかのオプション見たらわかるけど、これしかない)	

一個一個見ていくてもいいけど、ひとまずオブジェクトを作りたいと思います。

ちなみに LAYOUT の SWIZZLE はデータの並びがややこしい(モートン符号化みたいになびになってる)んでやめとこう←近いピクセルが隣接する並びなので圧縮効率がいいことがあるんだけど、やめとこう。敢えて今は沼には踏み込むまい。

とりあえずここまで読んで、32ビットで幅256、高さ256のテクスチャオブジェクトを作ってみてください。

自分で考えて書くのもお勉強です…というかそれが正しいお勉強です。センセーが書くまで待ってる人は置いていきます。

(待ちます)

(しばらくお待ちします)

いかがでしょうか?

よし、S_OK が返ってる前提で行きます。

書き込み

ちなみにこいつを Map しようとすると失敗します。どういう事?

という事で Map ではなく別の方法を取ります。WriteToSubresource という関数を使用します。

<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12resource-writetosubresource>

ホンマ正攻法が通用しないの腹立つわあ…。

ちなみに第一引数は 0 にしておきます。問題の第二引数ですが、

「リソースデータをコピーする宛先サブリソースの部分を定義するボックスへのポインタ。NULL の場合、データはオフセットのないデスティネーションのサブリソースに書き込まれます。ソースの寸法は、目的地に適合しなければなりません([D3D12_BOX を参照](#))。」

空のボックスはノーオペレーションになります。先頭の値がボトムの値以上であるか、または左の値が正しい値以上であるか、または前の値が後の値より大きいか等しい場合、ボックスは空です。ボックスが空の場合、このメソッドは何も操作を実行しません。」

ん? nullptr でもいいって事ですかね? ちょっと検証してないんですが、

WriteToSubresource(インデックス(0)、ボックス、データポインタ。横一列のデータ量、全画像のデータ量(バイト))

このボックスについてだけ、画像の上下左右と前と後ろを入れます。上下左右は画像の大きさで決めればいい。ただ front と back は 0,1 にしてください。一応今回は画像を用意していないのでノイズなんかの模様がビットマップでもぶち込んでしまいましょう。

ちなみにサンプルでは WriteToSubresource ではなく、UpdateSubresource を使っています。ただし、D3DX12 の関数だし、これ使ったやり方は中間リソース作って使わなきゃだし、必然性が良く分からぬ。

<https://docs.microsoft.com/en-us/windows/desktop/api/direct3d12/nc-direct3d12-updatesubresources1>

ので使いません。ちなみにソースコード内部に潜っていくと、結局 WriteToSubresource と同じような事をやっています。CopyBufferRegion を呼び出して何やらやっているようです。

UpdateSubresources を使ったやり方は後で紹介をしますので、両方やってみても構いません。

```

D3D12_RESOURCE_DESC resdesc = {};
resdesc=textureBuffer->GetDesc();
D3D12_BOX box = {};
box.left = 0;
box.right = (resdesc.Width);
box.top = 0;
box.bottom = (resdesc.Height);
box.front = 0;
box.back = 1;
result = textureBuffer->WriteToSubresource(0, &box, データの塊ポインタ, 1行のデータサイズ,
全体のデータサイズ);
こんな感じですかね。

```

バリアとフェンス

で、このテクスチャ書き込みも結局 GPU 側への書き込み命令なのでリソースバリアが必要になります。

```

_cmdLists->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(_textureBuffer,
D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));
_cmdLists->Close();
_cmdQ->ExecuteCommandLists(1, (ID3D12CommandList* const*) &_cmdLists);

```

あとはいつものようにフェンス待ち

シェーダリソースビューを作る

さて、書き込みもできしたことだしテクスチャビューに当たるシェーダリソースビュー(SRV)を作りたいところではあるが、ここで一つ考えたのだが、

レンダーターゲットビューとテクスチャリソースビューはヒープ上で共存できるかと言う話だが…まあまずはレンダーターゲットビュー作った時のヒープ作成のコードを見直してみよう。

```
D3D12_DESCRIPTOR_HEAP_DESC descHeapDesc = {};
```

```

descHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
descHeapDesc.NodeMask = 0;
descHeapDesc.NumDescriptors = 1;
descHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
result= _dev->CreateDescriptorHeap(
    &descHeapDesc,
    IID_PPV_ARGS(&_descriptorHeap)
);

```

…つまりそういうことだ。

ヒープを作る時点で、そのヒープは「何用か?」は決まっていると思われる。だって Type が決まっちゃうんだもんよ。

じゃあいちいちサイズ測る必要…なくね?

```
auto rtvSize=_dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
```

と思ってたんだが、

D3D12_DESCRIPTOR_HEAP_TYPE_NUM_TYPES

なるものもあるようだ。ただ、これを使ってるサンプルがないのと詳しい説明もないのとでちょっと怖いので使ってないです。

ケツ論としては

ConstantBufferView と ShaderResourceView と UnorderedAccessView はまとめてよいが、
RenderTargetView と DepthStencilView とはまとめてはならない。

つまりところ、深度まで使おうと思ったらヒープは 3 種類必要という事になる。

というわけで定数バッファ、テクスチャバッファのためのヒープをいっちょ作りましょう。そのためデスクリムゾン…デスクリプタヒープが 3 種類。なので、元のレンダーターゲットに使用しているデスクリプタヒープの名称を変えておきましょう。

```
ID3D12DescriptorHeap* _rtvDescHeap;//RTV(レンダーターゲット)デスクリプタヒープ
ID3D12DescriptorHeap* _dsvDescHeap;//DSV(深度)デスクリプタヒープ
ID3D12DescriptorHeap* _rgstDescHeap;//その他(テクスチャ、定数)デスクリプタヒープ
```

ちなみに現行のレンダーターゲットに関しては _rtvDescHeap にしてください。

でテクスチャに関しては rgstDescHeap と言う名前にしていますが、これに関しては名前を後

で変えるかもしれません、今の所はシェーダのレジスタに関するヒープって事で `rgstDescHeap` って名前にしています。

ヒープができたらシェーダリソースビューです。

```
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MipLevels = 1;
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
```

はい、上4つのパラメータはだいたい分かることと思いますが、最後の1パラメータが謎ですねえ。

https://docs.microsoft.com/ja-jp/windows/desktop/api/d3d12/ne-d3d12-d3d12_shader_component_mapping

「この列挙型を使用すると、SRV は、メモリフェッチ後にシェーダ内の4つのリターンコンポーネントにメモリをルーティングする方法を選択できます。各シェーダコンポーネント(0..3)(RGBAに対応)のオプションは次のとおりです。SRV フェッチ結果からのコンポーネント 0..3 または強制 0 または強制 1。」

D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING を指定すると、デフォルトの1:1マッピングが示されます。そうでない場合は、マクロ D3D12_ENCODE_SHADER_4_COMPONENT_MAPPING を使用して任意のマッピングを指定できます。」

ともかく最初の一歩なのでデフォルトをやっておきましょう。

さて、これで `CreateShaderResourceView` が使えますね。

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

あとは `RenderTargetView` とやることは同じです。

出来上がったものが `ShaderResourceView` オブジェクトですがひとまずはそこまで。あとはルートシグネチャとコマンドリストのお仕事です。

サンプラを設定

サンプラは DxLib の時にも話しましたが uv が 0~1 の範囲を越えた時にどういう扱いをするのかというものです。

```
D3D12_STATIC_SAMPLER_DESC samplerDesc = {};
samplerDesc.Filter = D3D12_FILTER_MIN_MAG_LINEAR_MIP_POINT;//特別なフィルタを使用しない
samplerDesc.AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;//縁が繰り返される(U方向)
samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;//縁が繰り返される(V方向)
samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;//縁が繰り返される(W方向)
samplerDesc.MaxLOD = D3D12_FLOAT32_MAX;//MIPMAP上限なし
samplerDesc.MinLOD = 0.0f;//MIPMAP下限なし
samplerDesc.MipLODBias = 0.0f;//MIPMAPのバイアス
samplerDesc.BorderColor = D3D12_STATIC_BORDER_COLOR_TRANSPARENT_BLACK;//エッジの色(黒透明)
samplerDesc.ShaderRegister = 0;//使用するシェーダレジスタ(スロット)
samplerDesc.ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;//どのくらいのデータをシェーダに見せるか(全部)
samplerDesc.RegisterSpace = 0;//0でいいよ
samplerDesc.MaxAnisotropy = 0;//Filter が Anisotropy の時のみ有効
samplerDesc.ComparisonFunc = D3D12_COMPARISON_FUNC_NEVER;//特に比較しない!(ではなく常に否定)
これはルートシグネチャのパラメータにほっておいてください。
```

あー、ルートシグネチャを設定する前にシェーダ設定した方がいいね。

シェーダにテクスチャの受け取り側を記述する

それでは shader.hlsl の先頭に

```
Texture2D<float4> tex:register(t0);
SamplerState smp:register(s0);
```

と書いてください。

テクスチャの **0番レジスタ** とサンプラの **0番レジスタ** を設定します。CPU 側から投げたテクスチャやサンプラが↑の tex や smp に入ります。

ちなみに、番号の前の t だの s だのに関しては

[https://msdn.microsoft.com/ja-jp/library/ee418530\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee418530(v=vs.85).aspx)

見るといいと思います。

ともかく CPU 側からのデータを受け取る準備ができました。後はピクセルシェーダにてテクスチャの画素値を参照するようにすればいいのです。

ということで、Texture2D の Sampler 関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/bb509695\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509695(v=vs.85).aspx)

tex.Sample(サンプラオブジェクト,UV 座標);

のようにして使います。ちなみにピクセルシェーダでのみ有効です。

では、チヨット頑張って書いてみてください。シェーダエラーが起きなくなるまで頑張りましょう。

で、得られた画素値ですが…ひとまずは、その値をそのまま返すようにしてください。

ルートシグネチャを設定

はい、これがまた面倒…前に話したようにルートシグネチャ→ルートパラメータ→レンジの階層構造になっています。

サンプラはともかく、テクスチャが例によってルートパラメータとレンジの指定が必要になってきますのでやっていきます。

今回はテクスチャ(シェーダリソースビュー)なのでレンジは

```
descRange.RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV; //シェーダリソース
descRange.BaseShaderRegister = 0; //レジスタ番号
descRange.NumDescriptors = 1;
descRange.OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
```

であり

```
rootParam.ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
rootParam.DescriptorTable.NumDescriptorRanges = 1;
rootParam.DescriptorTable.pDescriptorRanges = &descRange; //対応するレンジへのポインタ
rootParam.ShaderVisibility = D3D12_SHADER_VISIBILITY_PIXEL; //ピクセルシェーダから参照;
```

であるとして、

それぞれ設定していきましょう。

毎フレームやること

ルートシグネチャのセット

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

```
_cmdList->SetGraphicsRootSignature(_rootSignature);
```

ComputeRootSignature と間違えないよう注意です。

次はテクスチャ用デスクリプタヒープのセットですね。

SetDescriptorHeap を使います。

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

```
_cmdList->SetDescriptorHeaps(1, &_rgstDescHeap);
```

ちなみに、第一引数がセットしたいヒープの数で、第二引数がヒープ配列です。1つしか指定しないので、ポインタでオッケー。

最後にデスクリプターテーブルのどれを使用するかを指定します。

SetGraphicsRootDescriptorTable

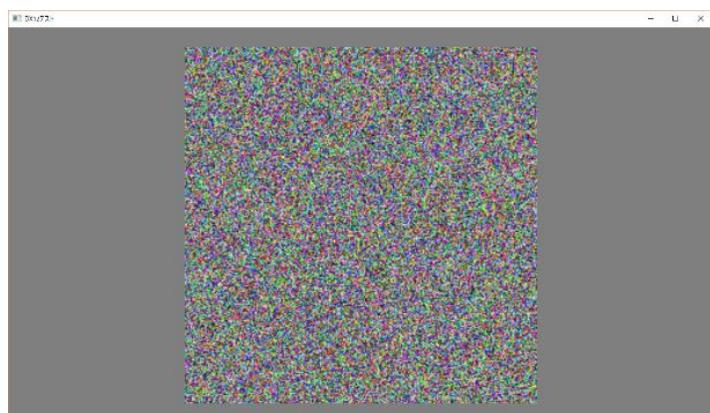
<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12graphicscommandlist-setgraphicsrootdescriptortable>

今回はデスクリプターテーブルが一つしかないし、レジスタに関連してるヒープも一つしかないので、

```
_cmdList->SetGraphicsRootDescriptorTable(0, hstart);
```

てな感じでいいと思います。

ここまでが“適切にできていれば”…



このようにノイズでもなんでも設定した絵が出ます

昨年とかは自分でビットマップ読み込んでどうこうしたりしてたんですが、今は自分が好きな絵を表示したいでしょうからここいらでテクスチャ読み込み関連のヘルパライブラリを投入します。

ヘルパライブラリを使わずに BMP から表示したい人は

```

ImageData*  

BMPLoader::Load(const char* filePath) {  
  

    BITMAPFILEHEADER fHeader = {};  

    BITMAPINFOHEADER iHeader = {};  

    FILE* fp = fopen(filePath, "rb");  

    if (fp == nullptr) return nullptr;  

    fread(&fHeader, sizeof(fHeader), 1, fp);  

    fread(&iHeader, sizeof(iHeader), 1, fp);  

    assert(iHeader.biBitCount == 24);  

    ImageData* ret = new ImageData(iHeader.biWidth, iHeader.biHeight);  

    auto buff=ret->GetBufferPointer();  
  

    //BMPが24bitの場合32ビットに変換する  

    //もっと言うとXBGR⇒RGBAに変換する  

    for (int line = iHeader.biHeight - 1; line >= 0; --line) {  

        for (int count = 0; count < iHeader.biWidth * 4; count += 4) {  

            unsigned int address = line*iHeader.biWidth * 4;  

            unsigned char bgr(3);  

            unsigned char rgba(4);  

            fread(bgr, sizeof(char), 3, fp);  

            rgba(0) = bgr(2); //b→r  

            rgba(1) = bgr(1); //g  

            rgba(2) = bgr(0); //r→b  

            rgba(3)=255;  
  

            //memcpyとか使いたくなないのでござるが…  

            memcpy(buff + address + count, &rgba, sizeof(unsigned int));  
  

        }  

    }  
  

    fclose(fp);  

    return ret;  

}

```

↑これでも参考にしてください。あ、ImageDataってのは単なる中間クラスです。

```

///イメージデータ
class ImageData
{
    friend ImageLoader;

public:
    enum class DataType {
        unknown,
        bmp,
        png,
        tga
    };

private:
    DataType _dataType;
    unsigned int _width;
    unsigned int _height;
    std::vector<char> _data;

public:
    ImageData(int width,int height);
    ~ImageData();
    unsigned int GetWidth();
    unsigned int GetHeight();
    char* GetBufferPointer();

};

こういう僕のプログラムが Web のどこかにある画像読み込みを参考にしてもらえばいいかと思います。

```

リソースのL0とかL1について

ちょっと分からぬままにしておくのが気持ち悪いので色々と考えたことを書きます。一応キーになっているのがUMA(ユニファイド・メモリー・アーキテクチャ)です。

<https://ja.wikipedia.org/wiki/%E3%83%A6%E3%83%8B%E3%83%95%E3%82%A1%E3%82%A4%E3%83%89%E3%83%A1%E3%83%A2%E3%83%AA%E3%82%A2%E3%83%BC%E3%82%AD%E3%83%86%E3%82%AF%E3%83%81%E3%83%A3>

ちなみに昔は似たような名前の仕組みに DMA(ダイレクトメモリアクセス)っていう仕組みが

あって、DirectX9 より前の時代はコソツにお世話になっていた。

https://ja.wikipedia.org/wiki/Direct_Memory_Access

まあ、それはともかく UMA だ。未確認生物ではない。

『メインメモリを CPU だけでなく、他のデバイスにも共有して使用するメモリアーキテクチャの一つである。』

『メインメモリの一部を VRAM の一部として扱い、CRT(CRT コントローラー)に DMA 転送することで、画面を表示していた。DMA が動作中 CPU はメモリバスの使用権を失い、画面表示中は CPU 本来の能力を発することができなかった。そこで計算などの用途において DMA を停止し、CPU がメインメモリをフルにアクセスできるようにすることが一般的だった。』

『現代のこの方式の応用もまた、VRAM をメインメモリにマッピングする 場合に用いられていることが多い。このアーキテクチャが PC に適用された時は、CPU 本来の性能を発揮できないことから嫌われた。そこで CPU 動作速度の低下を避けるため、メモリバスの周波数を CPU 本来のバス周波数より高く設定し、CPU からのメモリアクセスをさまたげないよう工夫されるようになった。CPU がメモリにアクセスする際に GPU は目的の演算を遅延することから、表示にジッターが現れることが多かった。後にこれはメモリアクセス方式を工夫したり、あるいは最終的に表示に使うメモリのみを、グラフィックボードに搭載することで解決した。しかし、さらに時代が下って再び採用され始める。PC に限らず、ワークステーションでも一般的であった。3D アクセラレーターにおいて、VRAM だけでなくテクスチャメモリ、イメージキャプチャ結果を保持するメモリとしても使われた。』

ほんまに書き方が不親切だなと思う。もう少し平易に書けないものだろうか。いやいや説明していただけてる事がありがたいかな。

『VRAM』の『UMA』の項目も見ておいた方がいいかもね

<https://ja.wikipedia.org/wiki/VRAM#UMA>

こっちの方が分かりやすいかな。

『メインメモリの領域から他用途のメモリを間借りすることを UMA(ユニファイドメモリアーキテクチャ)という。』

ひとつですね。これが一番適切ですね。

ともかく、

<https://www.slideshare.net/mistercteam/d3-d12-anewmeaningforefficiencyandperformance>

の34ページの説明および

https://docs.microsoft.com/ja-jp/windows/desktop/api/d3d12/ne-d3d12-d3d12_memory_pool

D3D12_MEMORY_POOL_L0 メモリプールは L0 です。L0 は物理システムメモリプールです。アダプタがディスクリート / NUMA の場合、このプールの CPU 帯域幅は大きくなり、GPU の帯域幅は小さくなります。アダプターが UMA の場合、このプールだけが有効です。

D3D12_MEMORY_POOL_L1 メモリプールは L1 です。L1 は、通常、物理ビデオメモリプールとして知られています。 L1 は、アダプタがディスクリート / NUMA の場合にのみ使用でき、GPU の帯域幅が大きく、CPU がアクセスできない場合もあります。アダプタが UMA の場合、このプールは使用できません。

L0 は CPU 側のメモリ共有メモリとして使用し GPU 側から見れる状態でのメモリプール。であり、L1 は GPU 側でのメモリプールではないかと思われます。どういう事がというと、

(この場合の L0, L1 ってのが CPU の L1, L2 とはちょっと違うので注意してほしい。CPU 側のメモリか GPU 側のメモリかということで 0 を CPU 側 1 を GPU 側にしているのだと思う)

<https://pc.watch.impress.co.jp/docs/column/1month-kouza/b46073.html>

の話も読んどくといいと思うけど、CPU 側から GPU 側へデータを転送する際にはそれなりのコストがかかる。物理的に離れているものもあるが、そもそも構造が違うので転送時にはそれぞれの特性に応じたコストがかかる。

恐らく DirectX11 まではメインアダプタになっているグラボのドライバによってこのあたりの選択を自動化していたのだと思われる。オンボードならば L0 で刺し GPU なら L1 を利用するとかそういう具合に。

それならば一番の疑問点だが、なぜそのように使いやすさに逆行するような仕組みにしてしまったのかだが、恐らく GPU を複数刺したり、さらにその状態でオンボードまでフルで回転させてしまいたいなら、それはアプリケーション開発者側が制御しなければならないということで、今回のような仕組みになっていると推測できる。

ちなみに CD3DX の言うように、サンプル通りにやれば DirectX11 がやってたことと似たような状態にできるというのが、サンプルの言いたいことじゃないだろうかと思う。

DirectXTex(WIC,DDS)を組み込み

…これ組み込むとちょっと大きさになるからちょっと避けたいのと、こいつも常に更新繰り返していて、下手なタイミングで組み込むと動かなくなる可能性があるので、今回は最新にしたばかりだし大丈夫だろうと思います。

まず

<https://github.com/Microsoft/DirectXTex>

を丸ごと落としてきます。

DirectXTex/Desktop_2017_Win10.sln

を開きます。ビルドします。ビルド時にビットは合わせておいてください。

ビルドすると DirectXTex.lib ができます。定義は DirectXTex.h にあります。プロジェクトからここにパスを通しておいてください。

一応パスは DXTEX_DIR ということにしておきます。

DirectXTex-master\DirectXTex

に合わせておけば

プロジェクトプロパティで

追加のインクルードディレクトリに \$(DXTEX_DIR) を

追加のライブラリディレクトリに \$(DXTEX_DIR)\Bin\Desktop_2017_Win10\x64\Debug

をそれぞれ記述します。

で、

#include<DirectXTex.h>

と

#pragma comment(lib,"DirectXTex.lib")

をそれぞれ記述。

これで一応の準備ができました。ちなみに WIC 単体で使用するだけならこんな大掛かりなものは必要なく、件のソースコードの WICTextureLoader12.h, cpp をそのまま自分のライブラリにぶち込んでいいと思います。その場合は

LoadWICTextureFromFile

という関数を使います。ファイルパスが WCHAR 披いてなんど色々と面倒かもしれません。

`LoadWICTextureFromFile(デバイス, パス, &テクスチャバッファ, 受け取るデータポインタ, 空のサブリソースデータ);`

というような使い方をします。一見テクスチャバッファに書き込んでくれるように見えますが、書き込んでくれません。書き込むためのおせん立てを整えてくれるだけです。

ちなみに `LoadFromWICFile` はそれすらしてくれないっぽいです。メモリ上に WIC テクスチャデータを読み込むくらいのようです。

`BMP`、`PNG`、`JPG` などのファイルは WIC という仕組み `Windows Imaging Component` なので、`LoadFromWICFile` とかそういう関数を使用します。

こんな感じで指定します。

```
TexMetadata metadata;
ScratchImage img;
auto result = DirectX::LoadFromWICFile(L"venom.png", WIC_FLAGS_NONE, &metadata, img);
ちなみに画像情報の本体は ScratchImage になります。うーん metadata はぶっちゃけいらな
いんじゃないかな。お助け関数なので特にヘルプもないし困ったな…。ともかく img にデータ
が入ります。で、GetImage って関数があったので
auto simg= img.GetImage(0,0,0);
こんな風にしてみました。第一引数はミップレベルなので 0 で合っているんですが、第二引数
がアイテム。第三引数がスライスってなってるんだけど、どちらも良く分からぬ。
```

パラメータが曖昧なのを気に入らない人は `GetPixel` すればデータ部分が取ってこれますので、それと `metadata` を組み合わせて `WriteToSubresource` しましょう。そうすれば



こんな感じに出力されます。

DDS に関してはまた必要になってからやりましょう。関数が別になるだけです。一応それ以外にもミップマップがどうこうあるんですが、それはミップマップを見ないと分からないと思うので、今回のこれはここで一旦切りましょう。

行列で座標変換してみよう

これもねえ…DirectX11 ならすぐ終わる話なんだけどもねえ。DirectX12 だとそもそもいかないんだわあ…例によってデスクリプターヒープ、デスクリプターテーブル、ルートシグネチャ、レジスタだのなんだの…あとアライメントの問題もあつたりして、非常になあ…って感じなんですわ。

まずは DxLib がやってるみたいに 2D ピクセル座標を D3D の画面座標に変換する部分を作つてみましようか。

行列おさらい

「行列」自体は前期もやつたし大丈夫だよね？ 縦横に値が並んでるアレだよ。

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

こういうやつやな？ で、使うのはこいつの乗算のみ。乗算のやり方は知ってるつけ？ 左辺値の行と右辺値の列をそれぞれかけて足したものがその要素の値になるんだ。

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

こういうことなわけやな？ この時に注意すべきことはなんやつたかな？

そう、乗算の順序やな？ たとえば

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

と

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 23 & 34 \\ 31 & 46 \end{pmatrix}$$

とは全然違う結果になる

当たり前だよなあ？

というわけで乗算の順序が本当に大切なのだ。

座標変換系の行列を主に使う事になるんだけど、

回転 → 平行移動

と

平行移動→回転

とは全然違う結果になるだろう? 行列の乗算の順序はこの通りにしてほしいのだ。↑の例ならば(↓は数学的に掛け算すると思ってくれ)もし、回転して平行移動なら

$$\begin{pmatrix} \text{平行} \\ \text{移動} \end{pmatrix} \begin{pmatrix} \text{回} \\ \text{転} \end{pmatrix}$$

のようになるし、逆に平行移動して回転なら

$$\begin{pmatrix} \text{回} \\ \text{転} \end{pmatrix} \begin{pmatrix} \text{平行} \\ \text{移動} \end{pmatrix}$$

となる。掛け算の順序がそのまま変換の順序を表していると思ってほしい(この場合見た目は逆だが)

あとこれはこの後に説明する XMATRIX を使用する際には乗算の順序は右側に右側にかけていってほしい。ここが混乱のしどころだが…まあ逆に直感的かもしれません。

例えば Y 軸回転の関数が XMMatrixRotationY で、平行移動が XMMatrixTranslation だとして回転して平行移動ならば

```
XMMatrixRotationY(XM_1DIV2PI)*XMMatrixTranslation(1, 2, 3);
```

と書くべきで逆に平行移動してから回転なら

```
XMMatrixTranslation(1, 2, 3)*XMMatrixRotationY(XM_1DIV2PI);
```

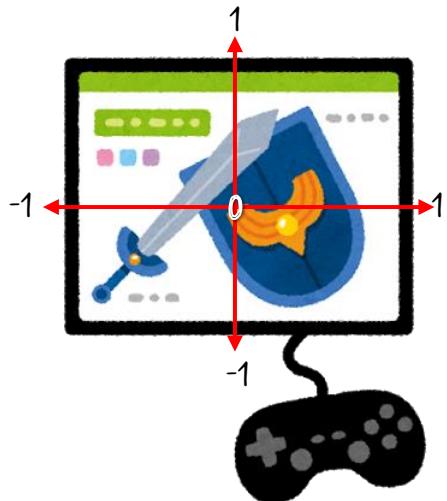
と書いてほしい。

まあこっちの方が直感的に考えられて便利でしょ? うん、そなんだこれならそもそも数学的な乗算順序を教えなくてもいいんじゃないのか。そう思うだろう。…なんだけど、シェーダ側の乗算関数においては、数学の方が採用されている。

頭が混乱しそうになるので注意してほしい。

2D 座標変換行列

ひとまず今回作るのはピクセル指定で画面上に表示する行列を作りたい。何の役に立つかと言うと UI とかに使うのだ。UI はピクセル単位で指定したいだろう? そうでもない?



ところが画面の大きさに関わらず座標はこういう指定だ。

画面…クライアント(ビューポート)サイズがいくらであろうとも-1~1という値になる。言つてしまえば正規化された状態になるんやな?

さてここで無理やりにでもピクセル指定をしたいとするとどうすればいいだろうか? 例えば画面のサイズが1280*720であると仮定しよう。

となると

$$(0,0) \rightarrow (-1,1) \cdots ①$$

$$(1280,720) \rightarrow (1,-1) \cdots ②$$

ここまですぐわかるかな。じゃあ画面の真ん中はどういう変換かな? ちょっと自分で考えてみてくれ。ノートかなんかに絵と数値を書きながら。

$$(1280/2,720/2) \rightarrow (0,0) \cdots ③$$

はい。これらすべて満たすような変換を考えてみよう。皆さんができるんですよ? なに僕が書くまで待ってるんですか? この程度…中学生でもできますよ?(大ヒント)

まあ連立方程式立てればすぐ分かるんだけど。変換って結局こういう事やん?

$$m(x_0 + a) + n(y_0 + b) = (x_1, y_1)$$

で、この x_0 だの y_0 だのにさっきの①~③の式の値を当てはめてやりやれ! ていうか①と②だけで十分だつたりする。

じゃあ①だが

$$(m(0+a), n(0+b)) = (-1, 1)$$

と

$$(m(1280+a), n(720+b)) = (1, -1)$$

の連立方程式を解きましょう

$$(ma, nb) = (-1, 1)$$

$$(1280m+ma, 720n+nb) = (1, -1)$$

を連立させると

$$(1280m, 720n) = (2, -2)$$

$$(640m, 360n) = (1, -1)$$

$$m = 1/640$$

$$n = -1/360$$

であり、

$$a = -640$$

$$b = 360$$

と言えます。

1280 は幅だから w として、720 は高さだから h だとすると

$m = 2/w$ であり、 $n = -2/h$ であり、 $a = -w/2$ であり、 $b = -h/2$ となりますね。最初の式に代入すると

$$\left(\frac{2}{w} \left(x - \frac{w}{2} \right), \frac{-2}{h} \left(y - \frac{h}{2} \right) \right)$$

となります。なんかもうちょっとわかりやすくしたいので

$$\left(\frac{2x}{w} - 1, 1 - \frac{2y}{h} \right)$$

とするとシンプルですね。中坊程度の連立方程式すら分からぬソザコナメクジでも心配ないよ。たとえば横だけ考えて 0~1280 が -1~1 になるんだから、幅は 2/1280 倍ってのは分かるよね？ああもうここが分からんかったら話にならないので、身の振り方を考えたほうがいいのでは？

ともかく、座標を幅の半分で割ればいいことは分かるんだけど、そうなると範囲が 0~2 になってしまって、-1 すれば -1~1 になって終わりですね。結局 1 をひくことになるんだね。縦方向も同様で、同じ結果になります。

まあここまで、たぶん中学校を卒業できた人なら大丈夫でしょう。分からないなら中学校に戻りましょう？

これから高校レベルに一気に上がりますが、今回のこれを座標変換として、この座標変換を一気にやる行列を考えてください。つまり

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2x}{w} - 1 \\ 1 - \frac{2y}{h} \\ 1 \end{pmatrix}$$

ってなる $abcdefghi$ を考えろって言ってるんだよ!!! 言わせんな恥ずかしい。
この程度の事は自分で考えろよ。考えないと力にならんぞホンマ。何のために朝っぱらから学校にきて高え金払ってんのかマジ考えてくれよ。

当然ながら単位行列なら

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

こうなります。当たり前だよなあ。あとは拡大縮小の事を考えると

$$\begin{pmatrix} \frac{2}{w} & 0 & 0 \\ 0 & -\frac{2}{h} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2x}{w} \\ -\frac{2y}{h} \\ 1 \end{pmatrix}$$

こうなるね。

あとは平行移動と考えればいいから

$$\begin{pmatrix} \frac{2}{w} & 0 & -1 \\ 0 & -\frac{2}{h} & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{2x}{w} - 1 \\ 1 - \frac{2y}{h} \\ 1 \end{pmatrix}$$

となりますね。ここまで読み飛ばした奴はもう何考えてんの? そんな生き方してて面白い?



ここまでが散々だった人は、今のうちに行列やり直しこう。今後は座標系に関しては、ほぼ行列しか出てこんぞ。

ともかく、この行列を作ってシェーダ側にぶん投げてそのまま乗算すればええんやな。

定数バッファ

さて、今回のデータは行列なんじゃけど。頂点でもインデックスでもなければテクスチャ(リソース)でもない。となるとどうすればええんでしょ?いや別にテクスチャとしてぶん投げても一向にかまわんのですが、最初つからそれはアカンやろとか、教える側の思惑もあって、定数バッファという奴で投げます。

CPP 側

だいたい今はシェーダリソース、シェーダリソースビューと同じなので

`ID3D12Resource* _cBuff;`

とかが必要になるわけやなあ…。多分昨年の授業とかだと、新しくヒープ作って、ってやってたけどせっかくだからテクスチャのやつと統合してみようかと思う。もし分かりづらかったら言ってください。DX11 脳だとこの辺は気持ち悪いはずなんで…。

まずはいつものようにバッファ作りましょうか。

一応、

`CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD)`

でもいいし

`D3D12_HEAP_PROPERTIES cbvHeapProp = {};`

`cbvHeapProp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;`

`cbvHeapProp.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;`

`cbvHeapProp.CreationNodeMask = 1;`

`cbvHeapProp.VisibleNodeMask = 1;`

`cbvHeapProp.Type = D3D12_HEAP_TYPE_UPLOAD;`

でもいい。とりあえず同じ意味。

```
result = dev->CreateCommittedResource(&cbvHeapProp,
    D3D12_HEAP_FLAGS::D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(数量*((ひとつ当たりサイズ + 0xff)&~0xff)),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&cbo->_buffer));
```

とりあえずサンプルコードを書いておく。何故かというとちょっとした部分で意外とややこしいからだ。一応書いておくけど、これ書いて終わりにしないでくれ。そんなことをしても全く意味がない。

まず、

(ひとつ当たりサイズ + 0xff)&~0xff

の部分の意味を考えてほしい。自分で、自分の頭で考えてほしい。考えられないなら授業は無駄。この授業を受けない! もうがれり! どこへなりとも行ってくれ。

正解とか間違いとかどうでもいいんだ。考えるという事が大事で、それができないならゲームプログラマなどなるべきではない。

とりあえず分かりづらい場合は、ひとつ当たりのサイズを S として、 $0xff$ を 255 として考えてくれ。

$(S+255)&~255$

ちなみに~の演算的意味は分かるよな? ビット反転だ。ちなみに数値の方は $size_t$ だと思ってくれ。つまりビット反転とは 1111111111111111111111111100000000 でな感じだ。そいつと & 演算。& 演算の前に 255 を足している。これは通常の足し算。

分かりませんかねえ。基本情報とかこういう問題出ないんですかねえ。何のための資格なんですかねえ。

答えを言う前にこれを見てくれ。

<https://docs.microsoft.com/en-us/windows/desktop/direct3d12/upload-and-readback-of-texture-data>

の下の方で

Constant data reads must be a multiple of 256 bytes from the beginning of the heap (i.e. only from addresses that are 256-byte aligned).

と書いてある。敢えて日本語訳しないけど、大体言つてることは分かりますよね? ちなみに DX11 までは 16 バイトアライメントだったはずなんだけど… どうしてそうなった!!!

はい、ここまで読めばお分かりですね。サイズを 256 の倍数にしてるんですわ。

さて、ここでサイズ 256 アライメントにしてなかった場合の挙動については「不確定」つまり何が起きても知らんよってこと。という事で、うまくいくかもしけんし、うまくいかないかも

しれん。だが、やめておいた方がいい。ていうかうまくいかない。ほんま何のエラーも出さずに応答なしになつたりするからマジ勘弁って感じです。

さて、昨年の授業ではまた新たにデスクリプターヒープを作つたりしてましたが、今年は先にテクスチャのために作ったデスクリプターヒープを使ってみましょう。

ただし、デスクリプタの数が増えますので、DescriptorNum の数は増やしてください。増やせよ？ 言ったからな？

で、テクスチャで作ったデスクリプタの後ろにくつける形で、定数バッファビューを作りましょう。

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

```
handle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc = {};
cbvDesc.BufferLocation = _cBuff->GetGPUVirtualAddress();
cbvDesc.SizeInBytes = size;
_dev->CreateConstantBufferView(&cbvDesc, handle);
```

さて、これで後ろにくついたことになるね。

じゃあ次はルートシグチャだ。これもパラメータを増やす必要がある。

```
desc_range(1).RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV; // 定数バッファ
desc_range(1).BaseShaderRegister = 0; // レジスタ番号
desc_range(1).NumDescriptors = 1;
desc_range(1).OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
```

はい。当然レンジの数が増えましたので NumRanges も増やしてください。増やせよ？

あと、今回はピクセルシェーダだけでなく頂点シェーダでも使用するため

```
root_param.ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL
```

としておきます。

あとは先ほど確保したバッファをマップして、先に書いた行列を放り込んでおいてください。
え? 分からん? ちょっとは考えや?

シェーダ側

前にも書いたと思いますが、定数バッファは `b` とレジスタ番号で指定します。今回は行列なので

```
matrix mat : register(b0)
```

とでもしておきます。次に頂点に対して行列を乗算しますが、*演算子ではなく、`mul` 関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/ee418335\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee418335(v=vs.85).aspx)

行列*ベクトル

ならば

`mul(行列, ベクトル)`

と書きます。

```
output.pos = mul(mat, pos);
```

はい、うまい事行けばそのまま反映されます。

3D化してみる

いよいよ待ちに待った3D化です。そんなに難しいことなくて

- ワールド行列
- カメラ(ビュー)行列
- 射影(ピースペクティブ)行列

の3つを3Dとして設定すればそれで終わりです。

ワールドは回転とか平行移動ですので、`XMMatrixRotation` なんとかや `XMMatrixTranslation` を使えばいいですね。

そして、これは DXLib でも出てきた

カメラ行列

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

射影行列

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

です。ここから 3D 用の行列を丁寧に作っていきましょう。

合成するんですが

World→View→Projection の順序で乗算するので、WVP 行列とも言います。現在のペラポリゴンを 0,0,0 中心に置いておいて、それを Z 値 -20 ~ -50 くらいの所から見てるって感じのカメラ座標で画角は PI/2 くらいにしておきましょうか?

がんばれ

うまくいけば



こんな感じで回転してくれます。

サンプルコードとしては…

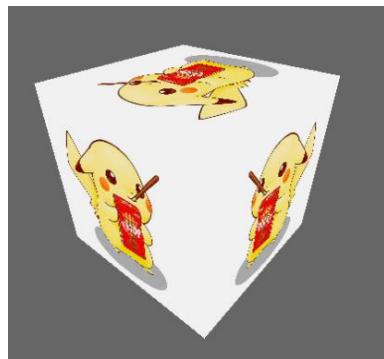
```
auto matrix = XMMatrixRotationY(angle);
auto eye = XMFLOAT3(0, 0, -30);
```

```

auto target = XMFLOAT3(0, 0, 0);
auto up = XMFLOAT3(0, 1, 0);
matrix *= XMMatrixLookAtLH(XMLoadFloat3(&eye), XMLoadFloat3(&target), XMLoadFloat3(&up));
matrix *= XMMatrixPerspectiveFovLH(XM_PIDIV2, static_cast<float>(wsizew) /
static_cast<float>(wsizeh), 0.1f, 300.0f);
*_matrix = matrix;

```

こんな感じですね。それができたらもうちょっと頑張って



こんな感じの立方体にしてくるくる回してみてください

XMVECTORについて

XMMatrix 系の関数において引数の型として XMVECTOR というのを使います。こいつがちょっとクセモノで

```

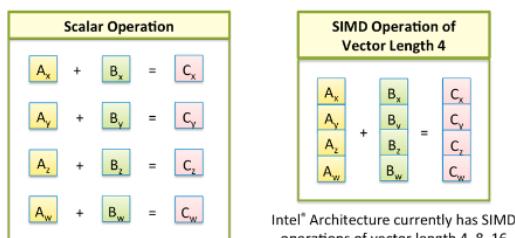
struct Vector3{
    float x,y,z;
};

```

みたいにな作りにはなってないんですね。残念ながら。そういう作りになっているのは XMFLOAT3 ですよ。でも Matrix 系の関数は XMVECTOR を要求しやがる…なぜか？

これはどう見ても SIMD 型演算に合わせるためにこうなっています。

<https://ja.wikipedia.org/wiki/SIMD>



ごらんのように 4 ついつぱんに計算する仕組みがあるんです

こういう仕組みに対応するには無理やりにでも4つにしないとスピードが出ないんです。FLOAT3にして4つめ無視すりゃいいじゃんと思うかもしれませんが、メモリ上で4つワンセットになつていないと不具合が起きるようになっています。

このためメンドクセエ XMVECTOR なんてものを使っているわけです。

かといって僕ら人間にとっては使いにくい。このため XMFLOAT3↔XMVECTOR を支援する関数として

- XMLoadFloat3
- XMStoreFloat3

というのがあります。Load と Store ってのが「？」って感じだと思いますがアセンブラーの時代にレジスタに乗つけたり下ろしたりしてた時の命令の名残ですね。ちなみにこの場合の Store は「お店」ではなく「保存する」「保管する」ってなイメージです。ちなみにお店の意味も色々な商品を「保管してる」とこから来てるわけです。

シェーディングはもう少し先(モデル読み込んでから)でいいでしょう。

リファクタリング

もう、すぐにでもミクさん読み込んでしまいたいんですけど、今のコードのままだとちょっとミクさん読み込みまで行くまでにコードがカオスなスパゲッティになっちゃうと思います。



既に頭の中がスパゲッティになってる人もいると思いますが…

既に Dx12Wrapper が結構な状況になっているでしょう？

まあ完璧なリファクタリングなんてできないと思いますので、簡単なところからやっていきましょう。

- ① Releaseで解放する奴は ComPtr を使う
- ② 色々と関数化する
- ③ RootSignature→DescriptorTable→Range と DescriptorHeap の仕組みを使いややすくクラス設計しなおす

くらいでしょうかね。③のウェイトが重いですね。

まず簡単なところからやっていきましょう。

ComPtr を使う

ComPtr ってのは、Microsoft が作ったスマートポインタの一種なんだけれど、解放のタイミングで Release 関数を呼び出してくれるスマートポインタです。

MS 系の特定のクラスは解放の際に delete ではなく、自身の Release 関数を呼び出すため、これが使えると判断しました。

ちなみに wrl.h をインクルードして使います。ネームスペースがちょっと面倒で

`Microsoft::WRL::ComPtr<ID3D12RootSignature>`

2つくついてちょっと嫌になりますよ～

ちなみに ComPtr の定義見れるんで中身を見ると

```

unsigned long InternalRelease() throw()
{
    unsigned long ref = 0;
    T* temp = ptr_;
    if (temp != nullptr)
    {
        ptr_ = nullptr;
        ref = temp->Release();
    }
    return ref;
}

```

とやってくれてますんで、今の所デストラクタで

`Dx12Wrapper::~Dx12Wrapper()`

```

{
    _cmdAlloc->Release();
    _cmdQ->Release();
    _cmdList->Release();
    _rtvDescHeap->Release();
    _dev->Release();
    _swapchain->Release();
    _dxgiFactory->Release();
}

```

とやってるのが不要になるわけです。現状の所はほぼ `Dx12Wrapper` が全部つかいでるので自分で `ScopedPtr`(スコープ内でのみ有効なスマート STL にはない) ので自分で実装する…テンプレートの勉強にもなる)でも実装してやってやればいいけど、今後そうもない場合もあるので、今回は `ComPtr` を使います。

ちなみに WRL ってのは Windows(Template)RuntimeLibrary の略です。なぜ T を省いた…

んで、こいつが VS2017 のデフォルト設定とちょいとばかり相性悪くて妙なエラーが出る。正直分からんのだが C7510 エラーが発生する。どういう事がと言うと

<https://social.msdn.microsoft.com/Forums/ja-JP/255e1c5c-33bb-438b-afa0-8dc38de24f31/comptr1246312521124731236420351123601239412356?forum=vcgeneralja>

うん、なるほど、/permission パラメータが悪さしているっぽいので、外しておきましょう。ちなみに「外していいのか?」と思うかもしれないけど、この仕様に関しては

『# 皮肉なことに C++ 言語の次バージョンで typename を省略可能にしようと議論されていますw』
らしいですので、まあ大丈夫なんちゃうかな?

ちょっと面倒な部分があるのは… IID_PPV_ARGS に入る部分は、このスマホのアドレスを渡せない。結果的に

```
ID3D12DescriptorHeap* rgstDescHeap = nullptr;
D3D12_DESCRIPTOR_HEAP_DESC descHeapDesc = {};
descHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
descHeapDesc.NodeMask = 0;
descHeapDesc.NumDescriptors = 2;
descHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
_dev->CreateDescriptorHeap(&descHeapDesc, IID_PPV_ARGS(&rgstDescHeap));
_rgstDescHeap.Attach(rgstDescHeap);
```

こういう形の書き方になる。Attach でポインタをセットします。ついでに生ポインタを欲しい場合は

```
_cmdList->Reset(_cmdAlloc.Get(), _pipeline.Get());
```

と書く。

面倒だったり「美しくない」と感じるなら使わなくてもいいかな。

色々関数化する(コメントをきちんと書く)

こういうわけのわからないプログラミングこそ、細かすぎるレベルで関数化した方がいいと思います。

とりあえず分かりやすいところで言うと

1. 関数が 100 行越えてるところ(あれば)
2. デスクリプターヒープ作成の部分
3. ルートシグネチャ設定部分
4. パイプラインステート設定部分

ちなみに僕のプログラムの場合、2～4を関数化すると自然に1がなくなります。もし2と3を関数化しても100行越えてる部分があればそこも関数化して100行越えないようにしてください。

ついつい調子に乗って

//コマンド系初期化

```
void InitCommandSet();
```

//スワップチェーン初期化

```
void InitSwapchain();
```

//フェンスの初期化

```
void InitFence();
```

//コンスタントバッファの初期化等

```
void InitConstants();
```

//頂点バッファの初期化(インデックスバッファもな)

```
void InitVertices();
```

//シェーダの初期化

```
void InitShaders();
```

//テクスチャの初期化

```
void InitTexture();
```

//レンダーターゲットビューのためのデスクリプタヒープを作成

```
void InitDescriptorHeapForRTV();
```

//レジスタ系のデスクリプタヒープを作成

```
void InitDescriptorHeapForRegister();
```

//ルートシグネチャの初期化

```
void InitRootSignature();
```

//パイプラインステート初期化

```
void InitPipelineState();
```

まで作りましたが、問題ないね。

名前はどうでもいいし、分け方も各自で考えてほしい。これは僕の一例に過ぎないので、正解とかそういうのはないと思ってくれ。

流石にソースコードは見せませんよ。甘えんな。ええかげんにしなさい!!



ちょっとアドバイスだけど、クソザコナメクジこそ、関数化するたびに動作確認すべき。クソザコナメクジは関数化するだけでバグを混入してしまう。いくつかやってしまった後だともうどうしようもなくなるぞ!!!

驕らず高ぶらず動作確認をしよう。やりすぎ感あるが、こうなる。

//色々初期化

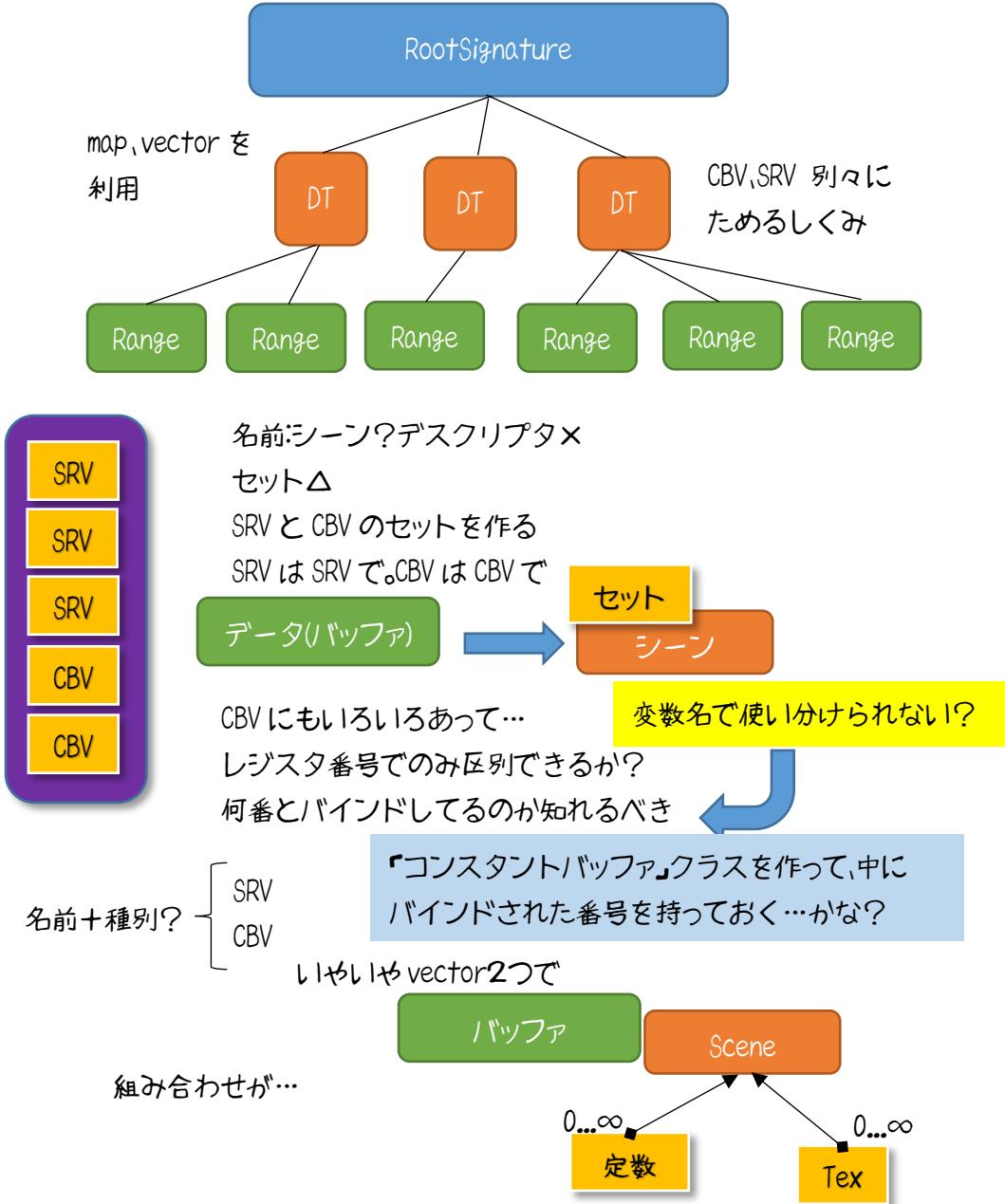
```
InitCommandSet();
InitSwapchain();
InitDescriptorHeapForRTV();
InitRenderTargetViewFromSwapchain();
InitFence();
InitVertices();
InitShaders();
InitDescriptorHeapForRegister();
InitRootSignature();
InitPipelineState();
InitTexture();
InitConstants();
```

さて、関数化ができる前提で行きます。

最後の難関だ。これ俺もノートにあーでもないこーでもないってやった結果なので、それなりに難しいと思ってくれ。

デスクリプター(テクスチャ、定数)まわりを支える設計

とりあえず、俺が書いてるメモをなんとか図にしてみる。



くらいのことはノートに走り書きしどんのじゃ。お前さんは書いとるか? この辺を考えようするとノートが汚くなるんじゃ。

ノートをきれいに仕上げようなどと思つてはいけない!!!!

誰に見せるんじゃ…あほか。時間の無駄じゃい。自分の頭を整理するためにやるんじゃ。アウト

プットするためにやるんじゃ。センセーがノートのチェックなんてしねーからのびのび書け。

あのな? 学習ってのはな?

情報の習得⇒**思考**⇒アウトプット⇒フィードバック⇒**思考**⇒最初へ戻る

をどれくらい繰り返したかなんじゅれ!!!! 繰り返してこそ定着するんじゅれ!!!! 愚直にやれとは言わん。できるだけ効率の良い方法を考えてくれ。でも、前を写してただけの奴は「思考」がすっぽり抜け落ちとるんじゅれ!!!! なんば繰り返しても時間の無駄無駄無駄無駄無駄無駄ア!!!

考えてるって言うなら何で傍らにガチャガチャ書いてるノートとか紙がないんじゅれ!!!! 書かずにやっていけるほど甘いアキテクチャちゃうんじゅれ!!!!

あのな……高校の頃にホンマは「基礎解析学」「代数幾何」「線形代数」「確率統計」

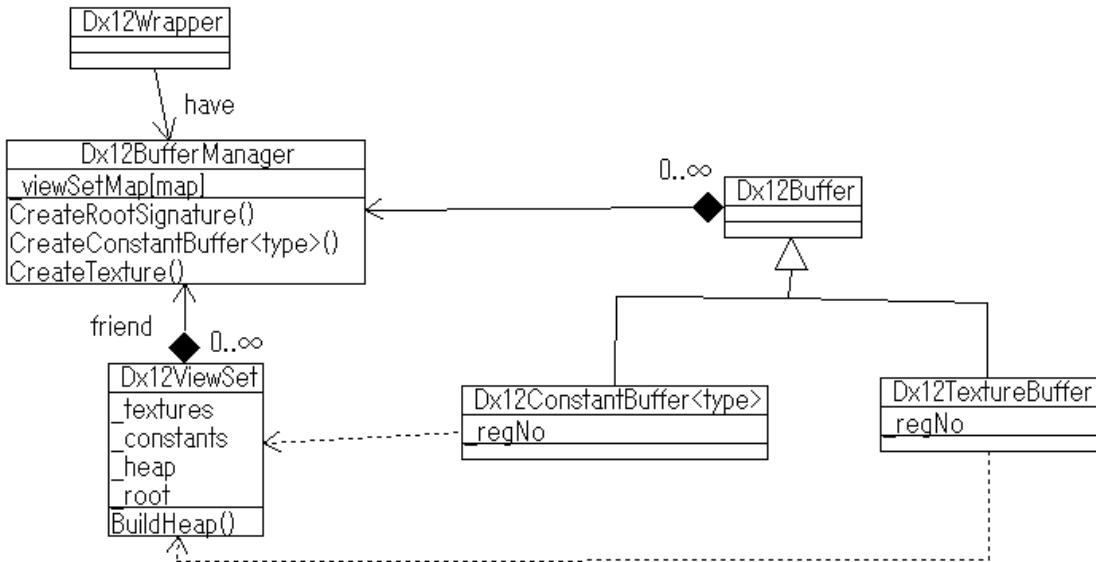


までやってるべきところをわざわざピタゴラスの定理やら連立一次方程式とかから教えないかん状況から3~4年、いや2~3年でゲームプログラマになろうってんだからそりゃキツイよ!! 難しいよ!!! でも自分で選んだんじゅん!?

現在できることとできるようになりたい事の間にギャップがあるなら、そこに犠牲は支払う必要がありますよそりや。

ノート取るなり放課後にやるなり最大限の努力をしろよホンマに!!! イヤ? キツイ? 長い人生のこの期間くらい我慢できない? そんな人はもうアキラメロン。早急に身の振り方を考えましょう。寝坊とかする休みする奴とかもう論外。しらんよもう。

はいはいということで、クラス設計に入ります。



大雑把に設計しました。大雑把でも既にめんどくさそうなのは感じられますね…。ちなみに第一案にすぎませんし、一つのことですが後でガラッと変えるかもしれません。一応書いておくと、**ViewSet** の **BuildHeap** は **private** 関数で、**Dx12BufferManager** が **CreateRootSignature** するタイミングで呼び出す予定です。

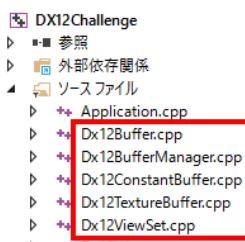
あのな、ゲームプログラマにはな? 設計力も求められるとんねん。コードをキレイに書く技術もそうだけどまとめあげる力も求められんねん。で、そこは『クリエイティブ』やねん。という事はセンサーの丸写しするんじゃなくて『俺ならこう設計する』って考えながらやらんとかはつかへんねん。これはあくまでも参考程度に考えといてや?

丸写ししてる奴!!!! そんな奴は確実に失敗する。失敗して気がつけば立て直し図れるけど、気が付かないとそのまま空回りして終わるぞ!!!!だから今言っておく!!!自分の!!!頭を!!!使え!!!このクソザコナメクジがつ!!!!!!

あ、ちなみに **ConstantBuffer** のところだけなんか<Type>とかついてるやん?

これ **ConstantBuffer** の中身の型やね。テンプレートなテクニックを使おうかと思ってます。
あまり難しい事はしたくないんですけど…まあ仕方ないズラね。

ともかく、後で修正かかると思うけど、僕はもう行きますわー。早速クラス作りまーす。



で、ちょっとテクニカルなのが予想される Dx12ConstantBuffer から見ていきます。

```
#include "Dx12Buffer.h"

class Dx12BufferManager;

///定数バッファ基底クラス

class Dx12ConstantBufferBase : public Dx12Buffer
{
private:
    Microsoft::WRL::ComPtr<ID3D12Resource> _cbuf;
public:
    Dx12ConstantBufferBase();
    virtual ~Dx12ConstantBufferBase() = 0;
};

///定数バッファ基底クラスから派生した定数バッファクラス

template <typename T>
class Dx12ConstantBuffer : public Dx12ConstantBufferBase
{
friend Dx12BufferManager;

private:
    T* _mappedAddress;
    //直接作らせない
    Dx12ConstantBuffer() {}
    Dx12ConstantBuffer(const Dx12ConstantBuffer&)
        代入オペレータオーバーロードもなし
public:
    ~Dx12ConstantBuffer() {}
};

…大丈夫かい? 息してますか~? テンプレ入ってますが意味はわかるかな?
```

Dx12Buffer は protected でレジスタ番号を持っている事を前提としてます。

で、そんなこんなでコンスタントバッファとテクスチャバッファのクラスを作ったらひとまず Wrapper のそれぞれのバッファ生成の部分をこれで置き換えます。

なんのためにそういうややこしいことをやるのかというと、使用するテクスチャとコンスタントバッファをグループピングしてしまいたいという意図があります。

何故グループピングなんてすんのかというと、使用する定数バッファビュー(CBV)とテクスチャビュー(SRV)は最終的にひとつのデスクリプターヒープに並べたい。また、デスクリプターテーブルのレンジにおいては CBV と SRV は区別しておく必要があるので、それぞれの vector を作っておいて、それを ViewSet クラスにでも持たせておいて、後から一気にデスクリプタ設定関係を構築しようという魂胆なのです。

意図するところはお分かりになられますか？

そうやないと、デスクリプタ構築のためにあらかじめテクスチャやら定数やらの使用数が分かつてなきやいけないことになるため、脳味噌側のメモリ節約したいからです。また、DX12 の場合はレジスタ番号の指定をレンジが握っている事になるため、実際のバッファがどのレジスタに放り込まれているのかの情報の距離が遠いです。このため、それぞれのバッファクラスにレジスタ番号を持たせておきます。

レジスタ番号は手動で入力するか、自動で入力するかが迷いどころですが、シェーダは自前で番号を書くので、本来は手動で入力すべきだと思います。ただ、その場合はレジスタ番号がぶりがなしあうかのチェックは必要となりますのでご注意ください。

ヘッダだけ公開

一応ヘッダだけ書いておきます

DxViewSet.h

```
class Dx12ViewSet
{
    friend Dx12BufferManager;

private:
    std::vector<std::shared_ptr<Dx12TextureBuffer>> _textures;
    std::vector<std::shared_ptr<Dx12ConstantBuffer>> _constants;

public:
    Dx12ViewSet();
    ~Dx12ViewSet();
    void BuildHeapAndViews();
}
```

```

};

Dx12Buffer.h
///DirectX12のバッファ系基底クラス
class Dx12Buffer
{
    friend Dx12BufferManager;

protected :
    ComPtr<ID3D12Resource> _resource;

public:
    ComPtr<ID3D12Resource> GetResource()const;
    Dx12Buffer();
    virtual ~Dx12Buffer() = 0;
};

Dx12BufferConstant.h
class Dx12BufferManager;
///定数バッファ基底クラス
class Dx12ConstantBufferBase : public Dx12Buffer
{
    friend Dx12BufferManager;

protected:
    size_t _buffSize;
    void* Map();

public:
    Dx12ConstantBufferBase();
    virtual ~Dx12ConstantBufferBase() = 0;
    size_t GetBufferSize() { return _buffSize; }
};

///定数バッファ基底クラスから派生した定数バッファクラス
///
template <typename T>
class Dx12ConstantBuffer : public Dx12ConstantBufferBase
{
    friend Dx12BufferManager;

private:
    T* _mappedAddress;
};

```

```

//直接操作させない
Dx12ConstantBuffer() {};
Dx12ConstantBuffer(const Dx12ConstantBuffer&);
void operator=(const Dx12ConstantBuffer&);

public:
    ~Dx12ConstantBuffer() {};
    void UpdateValue(T& value) {
        *_mappedAddress = value;
    }
};

Dx12BufferManager.h
class Dx12BufferManager
{
private:
    //定数バッファアライメントサイズを返す()
    size_t GetConstantBufferAlignedSize(size_t size);
    ID3D12Resource* CreateConstantBufferResource(size_t size);
    Dx12Wrapper& _dx;

public:
    Dx12BufferManager(Dx12Wrapper& dx);
    ~Dx12BufferManager();

    Dx12TextureBuffer* CreateTextureBuffer(const char* groupname, size_t width, size_t height);

    Dx12TextureBuffer* CreateTextureBufferFromFile(const char* groupname, const wchar_t* filepath);

///コンスタントバッファを作成します
///@param groupname グループ名(ワンセットにするための識別名)
template <typename T>
inline Dx12ConstantBuffer<T>* CreateConstantBuffer(const char* groupname) {
    auto ret = new Dx12ConstantBuffer<T>();
    ret->_buffSize = GetConstantBufferAlignedSize(sizeof(T));
    ret->_resource.Attach(CreateConstantBufferResource(ret->_buffSize));
    ret->_mappedAddress = static_cast<T*>(ret->Map());
}

```

```
    return ret;  
}  
};  
こんな感じですね。次の章は次ページからです
```

ともかく PMD モデルを表示させよう

あーもう前置きが長えんだよ!!俺は可愛いモデルを表示したいんだよふざけんなもう 150 ページ超えてんだぞ!!10 月まで間に合うのか!?

いつまで引っ張る気だよどこぞのバラエティ番組じゃあるまいしはよせえよ。

と皆様のお怒りの声が聞こえてきそうなので、さっさと PMD 表示までやっちゃいましょうね。二回目兄貴にとっては、ここらへんは昨年とほぼ変わらないのでつまんないかも知れませんが、初回兄貴はここからが楽しいのではないでどうか?

フォーマットを確認する

いやもう確認するまでもないんですが、一応参考文献的な出どころはきちんと書いておきた
いので書いておくと「通りすがりの記憶」というサイトに書いてあります。

https://blog.goo.ne.jp/torisu_tetosuki/e/209ad341d3ece2b1b4df24abfb19dbe4

この手のデータの構成はだいたい

ヘッダ→データという流れになっていて、さらにこの『データ』もまたヘッダ部とデータ部で構成されてたりします。ヘッダ→データの階層構造ですね。

ヘッダ

https://blog.goo.ne.jp/torisu_tetosuki/e/ac6aa40b23783309c8db5023356debba

ヘッダは基本難しい部分はないのですが、例えばこれを読み込もうとすると皆さんはフリーに考えて上のページからコピペして

```
struct PMDHeader{
    char magic[3]; // "Pmd"
    float version; // 00 00 80 3F == 1.00 // 訂正しました。
    char model_name[20];
    char comment[256];
}

なんてやると思う。そして
PMDHeader pmdheader;
fread(&pmdheader, sizeof(pmdheader), fp);
```

なんてやると思う。いや、いいよ？ここまでもし自分で考えてやれたら上等ですもん。逆に僕がなんかするまで待ってるようじゃ、そりやドラマとしての姿勢を疑いますねえ。

自分で考えてコーディングできない奴よりよっぽどマシ。ほめてつかわす。

だが、いきなり問題が発生する…うまくいかへんのや



ま、とりあえずは読み込んで中身を確認してくれい。

ぐっちゃぐちゃやろ? こうなるともうミンチさんやで。うーん初見さんでなんでこうなるか
わかる人あるか?

初見じゃまず分からぬと思ふけど、これはバイトアライメントってのが関係している。今までにもしけつと出てきたけど、何のことかよくわからなかつたかもしねない。ただ、こうやって普通に fread するだけでデータが壊れちゃう以上しつかり理解してほしい。

ここから昨年のテキストまんま

Windowsというか、最近のOSは、特に指定をしない場合4バイト区切りで処理を行おうとします。もしプログラムが4バイト区切りじゃない時はコンパイラが無理やり4バイト区切りにしようとします。結果として

- 最初の3バイトは“Pmd”という文字列
 - 次にfloat型でバージョン情報(1.00)4バイト
 - その次はモデルの名前が20バイト
 - 最後にコメントが256バイト

先頭 3 バイトという指定が悪さを行なうわけです。ここでコンパイラは Windows のために 3 バイト部分を 4 バイトとして扱おうとします。

ちなみに `fread` などでのバイト数指定では本来期待通りの挙動をします。

しかし構造体を展開する際に先頭の 3 バイトを 4 バイトとして扱い、余った 1 バイトは「padding」と言って詰め物をします。

で、別にファイル自体は詰め物をしないし、読み込みの処理も指定通り行われてしまうため結果としてもう version から狂ってくるわけです。

つまり、犯人は fread ではなく構造体だったのだ。sizeof() をした時点でこちらの期待通りの数値にならないのだ。

C 言語の構造体は…いやコンパイラか? まあコンパイラの方というべきか…処理系依存というべきか…まあ難しい話なんですねー。

簡単に言うとね、32bit マシンならメモリが 32bit ごと(つまり 4 バイト)ごとで区切られているんですね…大雑把に言うとだけど。



で、とある変数がこのメモリにアクセスしようとした時には 4 バイトごとにアクセスしようとします。これには理由があって、メモリへのアクセスやファイル読み込みの際に 1 バイト毎に読み取っていたら効率が悪いため 4 バイトごと読み取っているわけです。

というか、コンピュータは 4 バイトアクセスで最適化設計されているので、1 バイト毎にアクセスすると、4 バイトアクセスより 1 バイトアクセスのほうが効率が悪いんです。

今回みたいに 3 バイトデータが混じっていると、こういうアクセスになる。



なので、コンパイラがご親切にも(おせつかれにも?)「padding(詰め物)」してやって、あなたのプログラムを速くしてあげましょうって事なのだ。

まあ SIMD の話で既にこういう話は察しがついてると思うので、これ以上は深く解説しない。で、これを解消するにはどうすればいいのだろうか?

- pragma pack をいじる
- 読み込み時に先頭 3 文字だけ別物として fread する

という二つの手段があるが、2番目の方がたぶん理解的にはすっきりする。`#pragma pack`については各自調べて自分の責任の下やっておいてくれ。

まあまあ、これができたらさっそく頂点だ。

頂点リスト

https://blog.goo.ne.jp/torisu_tetosuki/e/5a1b1be2fb10b7838dfcbb010389707

頂点にもヘッダみたいなものがある。それが「頂点数」だ。これは読み込むバイト数にもかかわってくるが読み取り側のメモリ確保のためにもここに頂点数があるのがありがたい。まず4バイトを `unsigned int` として読み込もう。何頂点あるかな？

あ、ちなみに上のページでの `DWORD` ってのが初見の人もいるかもしれないけど、これはダブルワードって言って4バイト符号なし整数のことだ。それだけ知ってればいい。

さて頂点数がわかつたら頂点データロードだけどフォーマットを確認しておこう。

コピペ…

```
float pos(3); // x, y, z // 座標
float normal_vec(3); // nx, ny, nz // 法線ベクトル
float uv(2); // u, v // UV 座標 // MMD は頂点 UV
WORD bone_num(2); // ボーン番号 1, 番号 2 // モデル変形(頂点移動)時に影響
BYTE bone_weight; // ボーン 1 に与える影響度 // min:0 max:100 // ボーン 2 への影響度は、(100 - bone_weight)
BYTE edge_flag; // 0:通常、1:エッジ無効 // エッジ(輪郭)が有効の場合
```

38バイト…これはもう頭に…来ますよ。

フツーに読ませる気ゼロですわ。

どうしようか？どうしようね？どこの教科書にも載ってませんよ？自分の頭で考える癖をつけてください(今までのほとんども教科書なんかには載ってねーけどな。世の中はクソだ。大人はうそつきだ。だから自立しろ頼るな。)

ただ、ちょっとだけ救いはある。この構造の場合、パディングが最後にしか追加されないので。ということは `read` バイトを制限してやれば構造体に細工などいらないわけだ。だから、直感的な解決法で申し訳ないが

```
const unsigned int vertex_size = 38;
std::vector<char> pmdvertices(vertexCount*vertex_size);
```

fread(pmdvertices.data(), pmdvertices.size(), 1, fp);

と書ける。ただ、塊として扱うとはいえ char 配列ってのは乱暴かもしれないるので気に入らない人、もしくは頂点情報に細工をする場合はもうちょっと知恵を使わないとだめだろうが今は表示最優先。

まあ、何はともあれ頂点を読み始めたわけだから実はもうミクさんを、いや、ミクさんらしきモノを表示することはできる。さっそく見てみたいだろ？

とりあえずモデル表示してみよう

ひとまず、頂点の座標だけを表示する形でやってみてくれ。インデックスはまだないので、DrawInstanced でやってくれ。あと、トポロジーは POINT で。立方体の時とはデータが変わっているが、データのストライド(38 バイト)さえしっかりしてればなんかしら表示できるだろう。がんばれ。

え？ 教えてくれないのかって？

いやいや、ここまでやってこれた皆様ならきっとできますよ。

やる必要がある事は

- 頂点データを入れ替える
 - 頂点レイアウトを変更する
 - シェーダを変更する(ピクセルシェーダ…出力を黒一色に)
 - 画面クリア色を変更する(白一色に)
 - トポロジーを POINT に変更
 - DrawIndexedInstance⇒DrawInstance に
 - もちろん頂点数は変更しといてね
- そんなにやることないので頑張って。

うまいこといけば



こんな出ます

ちなみに僕が1回やらかした例として…

```
auto result=_dev->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
    D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(sizeof(vdata.size())),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&_vertexBuff)
);
```

さあ何が発生しているか分かるかい?

```
&CD3DX12_RESOURCE_DESC::Buffer(sizeof(vdata.size())),
```

ここやね。まあぶっちゃけると1頂点のデータすら乗りませんと。そりやあきまへんわ。

ひとまずミクさんらしきものが出てればOKです。

BadAppleにしてみる



こういう状態にしたい

現在は頂点データをもとにただただ頂点を点として表示しているだけなので、影絵すら作れません。

じゃあトポロジを POINT から TRIANGLELIST に戻せばいいんじゃないの? と思った人?



こうなります

なんかホラーみたいでこれはこれでありかもしれません、結局のところ根本の解決になりません。

TRIANGLELIST って事は面です。三角形の面です。面を構成するには頂点情報だけでは足りませんでしたよね?

そう…インデックスですよね?

インデックス情報を読み込みましょう

https://blog.goo.ne.jp/torisu_tetosuki/e/7cebae143cbbb9bae38ebbefef228c05b

簡単ですね。unsigned int でインデックス数を読み込んだら、unsigned short のインデックス集合を作成して、あとはインデックスとして読み込めばいいのです。

ちょっと、ここ自分で考えてやってみてくれないかなあ。ちなみに頂点の直後にインデックスデータがあるから、普通に fread すればいいけるはずだよ。

うまいこといけば



こんなん出ます

さあ、いよいよそれっぽくなってきました。もっと 3D っぽくしてみましょう

シェーディングしてみる

ついにきましたね。シェーディングってのは陰影をつけるという意味です。いったん分かりやすくするために反転してみます。



で、陰影をつけるためには法線情報が必要になります。既に法線は含まれておりますので、レイアウトに NORMAL を追加します。

```
"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
```

シェーダ側も変更します。

```
vs( float4 pos : POSITION, float4 normal:NORMAL)
```

```
struct Output {
    float4 svpos : SV_Position;
    float4 normal:NORMAL;
};
```

これで法線情報を使用することができます。ひとまず法線をそのまま色として出力してみましょう。



うーん。す…すりーディー?

まあ、これはあくまでも「法線」なので、最終的にライトベクトルと内積をとることで明るさを計算できます。

表面の明るさを決める最もシンプルな式として「ランパートの余弦則」ってのがあります。これは非常に簡単で、ライトのベクトルと物体の法線ベクトルのなす $\cos \theta$ によって明るさが決まるというものです。

$$I_{\text{lambert}} = k \cos \theta$$

内積にはこういう式がありましたね?

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta$$

また、この双方のベクトルが正規化済みなら、

$$I_{\text{lambert}} = k (\mathbf{a} \cdot \mathbf{b})$$

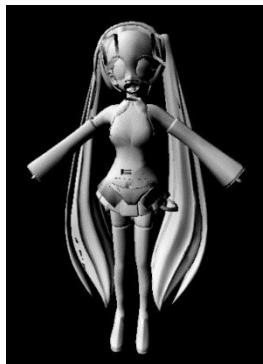
今回この k は 1 としてもいいので、あと HLSL では内積関数は dot ってのがありますので、それを使いましょう。正規化は normalize です。

法線は正規化されてると考えればいいので

```
float3 light = float3(-1,1,-1)
light = normalize(light)
brightness = dot(normal,light)
でいいでしょう。
```

こいつが明るさを示すので、RGB 全てにこれを適用すればいいです。

```
float4(brightness,brightness,brightness,1);
```



一応シェーディングはできているようです…が?
どうもおかしなことになっているようですね。

何故かな?何故だと思うね?

深度バッファ

はい、今まで何度か話には出てきてた「深度バッファ」ですね。深度バッファって何でしたっけ?

<https://ja.wikipedia.org/wiki/%E3%83%90%E3%83%83%E3%83%95%E3%82%A1>

<https://msdn.microsoft.com/ja-jp/library/cc324546.aspx>

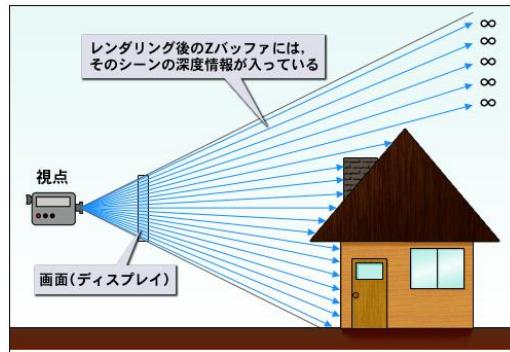
そう…深度バッファ法とは、視点からの距離を各ピクセルで…保持しておいて、今から色を塗ろうという時にその深度と今持っている深度を比較して、既に手前が塗りつぶされなければ破棄するという仕組みです。

深度バッファとは

<https://msdn.microsoft.com/ja-jp/library/cc324546.aspx>

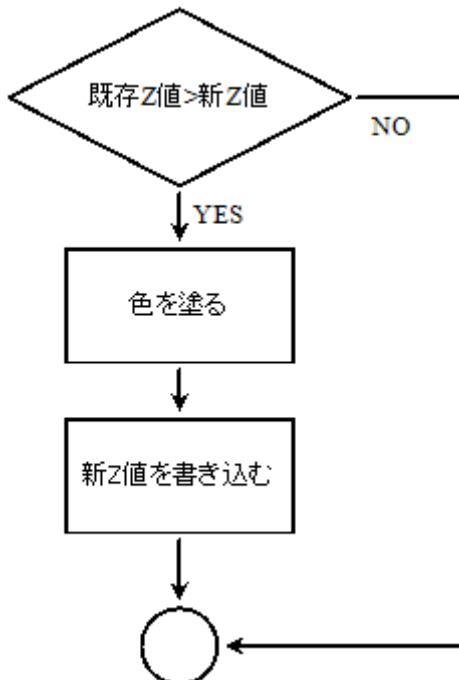
「深度」というのは何かというと、カメラから見た時のカメラからの距離の事です。いわゆる Z 値というのですが、UE4であったり3dsMaxであったりが Z を上方向として定義しているため、 Z 値って言うのが不適切になっちゃって、そのせいで「深度」というようになったんですね。

で、深度バッファってのは何かというと、画面上に色を乗せていく際に同時に深度値(Z 値)をピクセルに書き込んでいきます。その書き込み先を深度バッファというのです。



で、この深度バッファがどのように働くのかというと、今からそのピクセルを書こうとするときに Z テスト(深度テスト)というのを行います(タイミングとしてはピクセルシェーダ後… ラスタライザが終わった段階でテストすればいいのに… DX12 をクソ難しくする暇あつたら この辺どうにかしろよ)。

で、深度テストと言うのは既に書き込まれている深度値と今から書き込もうとする深度値を比較して、今から書き込もうとする深度値が既に書き込まれている深度値よりも小さければ描画&新しい深度値を書き込み、そうでなければ描画も深度値更新もしないということです。



す。

フローチャートにするとこんな感じですね。

で、この深度テストの仕組みのために深度バッファを作らなければならない(DX9 時代は深度テスト=TRUE にすれば終わってたのですが…)

残念ながら

`gpsDesc.DepthStencilState.DepthEnable=false;`

を true にすれば OK 的な代物ではありません。実際これを true にすると表示されません。お

そらくはバッファがないため比較ができずに常に深度テストが失敗するような結果になっているんでしょう。

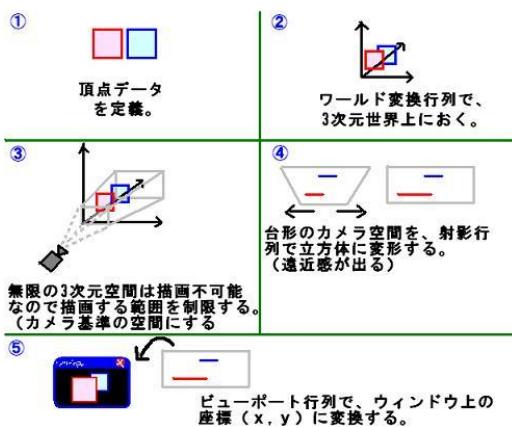
さて、というわけでちょっと不思議なバッファ…デプスStencilバッファを作つていましょ。え? 深度バッファじゃないの? という疑問はごもっともではあるのだが、なんかそういう仕様になっているのだ。ちなみに深度バッファは通常の float と同様に 32bit のバッファである。

なお、Stencilも使用したい場合はこのバッファのビットをStencilと深度値で折半して使用することになる。ちなみにStencilが 8bit で深度値が 24bit という風になる。

ともかく今回は深度値の事だけ考えればよいのでやっていこう。ちなみにこの時の深度値の範囲は 0~1 である。一番近いところが 0 で一番遠いところが 1 である。

「え? いやいやだって見える範囲を 0.1f~100.0f にしてるのにそれはねーよ WWW と思った人には『いいね!!』をくれてやろう。そういう疑問を持つのは正しい。」

以前に、「プロジェクト行列は視錐台を厚さ 1cm の板に変換する」というような話をしたのを覚えているだろうか…。そう、厚さ 1cm の板になってしまふのだ。このため最も近いところの z 値が 0.0 となり、最も遠いところが 1.0 となるように変換されているのだ。



出典:(ゲームプログラマを目指すひと)

この④やね。どうも遠近法の所にはかり目が言ってしまうのだが、このプロジェクト行列変換は z 値を 0~1 に正規化する役割も持っているのだ。

結局 DX12 では何をしなければならないの?

さて話を戻して深度バッファを作つていこう。もちろんいつものように深度バッファを作るだけではなく、色々とやってあげないといけないのだが…

1. 深度バッファ作成

2. 深度バッファビュー作成(デスクリプタヒープとかビューとか)
 3. パイプラインステートオブジェクトに深度バッファの設定を追加
 4. 深度バッファビューをレンダーターゲットと関連付け(毎フレーム)
 5. 深度バッファビューを毎フレームクリア
- …結構やることあるね。うまくいけば



このように適切な立体感をもって表示されます

深度バッファの作成

レンダーターゲットとは違い、スワップチェインにテクスチャが乗っているわけではないため、自分で制作する必要があります。作り方はテクスチャとして作ってもらえばいいです。あ、深度なので、フォーマットがチョイとばかり違いますが…

深度バッファはフリップの必要がないので、1つ作ればいいです。

`CreatedCommittedResource` を使ってリソース(バッファ本体)を作っていく。ほぼほぼテクスチャの時と同様なのでアレを参考に考えてほしい。

関数化するならテクスチャの生成は

`InitTextureForDSV`

で、

デスクリプタヒープと言うかビューは

`InitDescriptorHeapForDSV`

あたりで作ってしまえばいいかなと思います。

```
depthResDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
depthResDesc.Width = wsize.width; // 画面に対して使うバッファなので画面幅
depthResDesc.Height = wsize.height; // 画面に対して使うバッファなので画面高さ
depthResDesc.DepthOrArraySize = 1;
depthResDesc.Format = DXGI_FORMAT_D32_FLOAT; // 必須(大事) デプスですしあれ
depthResDesc.SampleDesc.Count = 1;
depthResDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL; // 必須(大事)
```

```
depthHeapProp.Type = D3D12_HEAP_TYPE_DEFAULT;//デフォルトでよい
depthHeapProp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;//別に知らなくてもOK
depthHeapProp.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;//別に知らなくてもOK
```

//このクリアバリューが重要な意味を持つので今回は作っておく
`D3D12_CLEAR_VALUE _depthClearValue = {};`
`_depthClearValue.DepthStencil.Depth = 1.0f;//深さ最大値は1`
`_depthClearValue.Format = DXGI_FORMAT_D32_FLOAT;`
 深さ最大値が1なのはわかるね?プロジェクト変換の結果、ニアファーガ0~1に正規化される
 からである。

```
result = dev->CreateCommittedResource(&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
                                         D3D12_HEAP_FLAG_NONE,
                                         &depthResDesc,
                                         D3D12_RESOURCE_STATE_DEPTH_WRITE, //デプス書き込みに使います
                                         &_depthClearValue,
                                         IID_PPV_ARGS(&_depthBuffer));
```

リザルトは確認しておきましょう。

深度バッファビューの作成

ClearDepthStencilView を使って作ります。これもほかのビューと同じですね。ビューデスクリプションとデスクリプターヒープが必要です。

```
D3D12_DESCRIPTOR_HEAP_DESC _dsvHeapDesc = {};//ぶっちゃけ特に設定の必要はないっぽい
ID3D12DescriptorHeap* _dsvHeap = nullptr;
_dsvHeapDesc.NumDescriptors = 1;
_dsvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_DSV;
result = dev->CreateDescriptorHeap(&_dsvHeapDesc, IID_PPV_ARGS(&_dsvHeap));
dev->CreateDepthStencilView(_depthBuffer,&dsvDesc, _dsvHeap->GetCPUDescriptorHandleForHeapStart());
CBV やら SRV みたいに兼用はできないので、深度バッファのためだけのヒープになります。
```

パイプラインステートオブジェクトに深度情報を追加

```
psoDesc.DepthStencilState.DepthEnable=true;//深度バッファを使うぞ
```

psodesc.DepthStencilState.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL; // DSV 必須
psodesc.DepthStencilState.DepthFunc = D3D12_COMPARISON_FUNC_LESS; // 小さいほうを通すぞ
とりあえず深度バッファを使うぞという事を明示します。ちなみに MASK_ALL ってのは「常に深度値を書き込む」という意味です。ちなみに「深度値を書き込まない」ということもでき、その時は MASK_ZERO になります。(αとの組みで使用することがあります)

次に FUNC_LESS ですが、これは深度テストの結果、大きいほうか小さいほうかどちらを採用するのかというものです。今回は深度値が小さい(カメラからの距離が近い)方を採用するので LESS にします。

次にここでも深度バッファのフォーマットを明示しなければなりませんので、

psodesc.DSVFormat = DXGI_FORMAT_D32_FLOAT; // 必須(DSV)

とします。

ではここまで設定したうえでパイプラインステートオブジェクトの生成が S_OK されることを確認してください。

されなければどこかが間違っています。

レンダーターゲットと深度バッファを関連付け



「一緒にポテトはいかがですか」
を三回連續で断った瞬間氣を失い、
目が覚めると彼と二人きりの密室
にいた

「レンダーターゲットのセットですね。一緒に深度バッファもいかがですか?」

ということで本来はレンダーターゲットと深度バッファは一緒にすべきものだったりします。なので、OMSetRenderTarget には深度Stencilビューを入れる場所が最初から用意されています。現在の OMSetRenderTarget をご確認ください。

```
_commandList->OMSetRenderTargets(1,&rtvHandle,false,nullptr);
```

という風になっていると思いますが、これの第3引数が nullptr になっていますね？定義を確認してみましょう。

[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)

第4引数が

pDepthStencilDescriptor (in, optional)

Type: const **D3D12_CPU_DESCRIPTOR_HANDLE***

A pointer to a **D3D12_CPU_DESCRIPTOR_HANDLE** structure that describes the CPU descriptor handle that represents the start of the heap that holds the depth stencil descriptor.

つまりところここに深度ステンシルデスクリプタハンドルを入れるってことです。つまり、

```
OMSetRenderTargets(1,&rtvHandle,false,&_dsvHeap->GetCPUDescriptorHandleForHeapStart());
```

こんな感じですね。で、これだけでは「まともに」機能しません。深度バッファは毎フレームクリアする必要があります。

深度バッファをクリア(毎フレーム)

`ClearDepthStencilView` という関数を使います。どうクリアするのかと言うと Z 値を無限大…と言いたいところですが、1 でクリアします。なぜ 1 かと言うと前にも話した通り、ビューポリューム `near~far` を 0~1 の範囲に正規化しているからです。

つまり 1 で初期化するという事は最初の Z 値が `far` になるわけで、クリッピングボリューム内に「見える」オブジェクトの Z 値は全て 1 未満だからオブジェクトに当たるたびに小さくなってしまいます。これをクリアしなければならないのです。

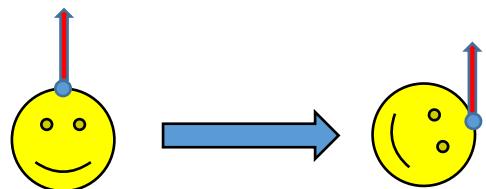
ちなみにこの機能により アルファブレンディングとの関係がうまくいかないことがあります、それはまあ…仕方ないと思ってください。そのうちその話をします。

ここまでがうまくいけば 3D のミクさんが石膏みたいに感じで表示されるはずです。
頑張りましょう。

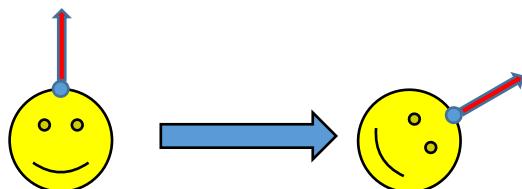
法線も座標変換

実際の話、モデルが回転する場合は座標だけでなく、法線も回転させなければいけません。

もし座標だけを回転させている状態のままならこうなります



イメージはなんとなくわかりますかね? 当然ながらこれではダメで



こうしたいわけ。当たり前だよなあ。

ところが現状としてはひとつの matrix に world, view, projection みつつの行列が合成されている状況で渡っているので話は単純ではない。悲しい事に合成されてしまうと元には戻せないんだ。



悲しいなあ…

いや、逆行列を持ってれば元に戻せるんだけど結局「逆行列」を渡さなければならぬるので、そもそも合成して渡すのではなく合成は GPU 側にお願いしておくって事でやろうかなと思った。でも頂点ごとにその計算が行われてしまうのは無駄でよろしくない。このため

- 合成前ワールド行列
- WVP 合成済行列

という渡し方にしようかと思います。渡すための構造体として

```
struct TransformMatrices {
    DirectX::XMATRIX world; //合成前ワールド
    DirectX::XMATRIX wvp; //WVP合成済み
};
```

という事でいいがでしょ? コンスタントバッファはそれに合わせて作ります。僕のプログラムだとこんな感じで渡します。

```
TransformMatrices tm = {};
tm.world = XMMatrixRotationY(angle);
```

```

tm.wvp = tm.world *
    XMMatrixLookAtLH(XMLoadFloat3(&eye), XMLoadFloat3(&target), XMLoadFloat3(&up)) *
    XMMatrixPerspectiveFovLH(XM_PIDIV2, static_cast<float>(wscale.w) /
static_cast<float>(wscale.h), 0.1f, 300.0f);
_constantBuffer.reset(_buffMgr->CreateConstantBuffer<TransformMatrices>("sample"));
_constantBuffer->UpdateValue(tm);
てな感じでシェーダ側も
cbuffer mat:register(b0) {
    matrix world;
    matrix wvp;
}
(中略)
pos = mul(wvp, pos);
normal = mul(world, normal);
こんな感じでシンプルにしましょう。

```

マテリアルを適用



この辺からみんなはマルチスレッドの面倒くささを思い知ることになるだろう…

細かい事は後で説明するけど、ひとまず DX11 時代のやり方を軽く言っておくと

- マテリアルごとにインデックス区切る
- その区切ったインデックスごとに描画する
- その描画の直前でマテリアル(定数バッファの内容)を切り替える

というやり方で OK だった。

なんとなく直感的に感じるだろう?ところが DX11 ではそうはいけないのだ。

残念ながら。

どういう事がと言うと、DX12 の GPU に対する命令はほぼ全てコマンドリストに溜めてからコマンドキーで実行と言う流れである。

そして定数バッファの内容を切り替えるという事は Map した内容に対して変更を加えるという事である。これは CPU 側がマップ内容を書き換えるという事になる。CPU 側の処理である。

これ、DX11であれば特に問題なかったのだが、さっきも言ったようにコマンドリストをコマンドキューでまとめて処理するといった性質上…どんなにプログラムの上で書き換え ⇒ 描画という命令をループで繰り返したところで結局



このようになってしまふのである。言ってる意味わかるかな？

はあ～(クソデカため息)あほくさ。昨年はこの概念が良く分かってなかつたので、かなり死にました～。



昨年の俺

ということで面倒だという事を急頭に置いたうえでマテリアルの適用(着色とか)を行っていきましょう。

マテリアルってなんや？

マテリアル(material)ってのはCGで言う場合は『表面材質』を表します。なぜ『表面』なのかと言うとCGの世界はペラペラの集合体だからです。中身が詰まってないポリゴンメッシュ集合なので、こういう言い方をします。

で、簡単に言うと着色のデータとかになるんやけど、基本的には『古典的』レンダリングの

- ディフューズ(拡散反射)
- スペキュラー(鏡面反射)
- アンビエント(下駄:…環境光反応成分)

になります。CG検定を受験する人は、これらについて理解しておきなあ？

どうかな？

一応拡散反射ってのは、表面がざらついてる場合の反射で入射光反転ベクトルと法線ベクト

ルの内積に比例する。基本的にはこの色がベースになる。っていうとちょっと語弊があるけど、まあMMDにおいてはそう思って支障はない。

スペキュラってのはハイライトに使用されるもので、光が反射したベクトルと、視線ベクトル反転ベクトルの内積のさらに n 乗(パワー)によって決まる。このパワー値が大きいほどハイライトが鋭くなる(より金属っぽくなる…つやつやする)。逆にこれが小さいとぬるーんって感じのハイライトになる)

アンビエントってのは環境光って言って、そのままやと暗すぎるから下駄を履かせるときに加算する。この段階ではその程度の理解でいいです。CG検定受ける奴はきちんと勉強するように。

マテリアルデータ読み込み

テクスチャなどを設定することになります。

https://blog.goo.ne.jp/torisu_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32

では「材質リスト」って言ってますね。

いものの流れです。

```
DWORD material_count; // 材質数
t_material material(material_count); // 材質データ(70Bytes/material)
```

- t_material

```
float diffuse_color[3]; // dr, dg, db // 減衰色
float alpha; // 減衰色の不透明度
float specularity;
float specular_color[3]; // sr, sg, sb // 光沢色
float mirror_color[3]; // mr, mg, mb // 環境色(ambient)
BYTE toon_index; // toon?.bmp // 0.bmp:0xFF, 1(01).bmp:0x00 ••• 10.bmp:0x09
BYTE edge_flag; // 輮郭、影
DWORD face_vert_count; // 面頂点数 // 面数ではありません。この材質で使う、面頂点リスト上のデータ数です。
char texture_file_name[20]; // テクスチャファイル名またはスフィアファイル名 // 20バイトぎりぎりまで使える(終端の0x00は無くても動く)
```

はい、また「クソ」データ構造ですね。2つのBYTE直後に\パディングが発生しますね。ふざけんな。これホントMMDの作者は作るときにどうしてたのかな?もしかしてpragma pack(1)してたのか?

まあこのマテリアルデータは GPU に直投げするわけではないので、パディングに気を付けさえすればそれほどややこしくはないと思います。ループ読み込みしちゃってもいいんじゃないでしょうか？

とりあえず色のメインデータは「古典的」レンダリングの場合ディフューズですから、ひとまずこいつさえあればいいです。

こいつもインデックスデータの直後にあるので、まずはマテリアル数を読み込みましょう。マテリアル数が DWORD ってのも逆に疑問を感じるのですが…今まであれだけケチつておいてお前 4,294,967,295 もマテリアルがあるとでも思ってんのか？

まあいいや。読み込んでみてください。ミクさんなら 17 個くらいですから確認してください。マテリアルは頂点とかに比べたら圧倒的に少ないのですが、けちるメリットはないんだけどなあ。

```
struct PMDMaterial {
    float diffuse_color[3]; // dr, dg, db // 減衰色
    float alpha; // 減衰色の不透明度
    float specularity; // スペキュラ乗数
    float specular_color[3]; // sr, sg, sb // 光沢色
    float mirror_color[3]; // mr, mg, mb // 環境色(ambient)
    unsigned char toon_index; // toon???.bmp //
    unsigned char edge_flag; // 輪郭、影
    // パディング 2 個が予想される…ここまでで 46 バイト
    unsigned int face_vert_count; // 面頂点数
    char texture_file_name[20]; // テクスチャファイル名
};

std::vector<PMDMaterial> _materials;
(中略)
for (auto& material : _materials) {
    fread(&material, 46, 1, fp); // 直値はあとで修正しとく
    fread(&material.face_vert_count, 24, 1, fp); // 直値はあとで修正しとく
}
```

で、この取ってきたマテリアルを利用してミクさんに色をつけたろか？って事やな。

さて、ここまでええんや、ここまでではな…

さて、ここからがお悩みどころやで……

マテリアルデータは複数あります。そしてマテリアルごとに描画を分けます。ここまではいい。

さて、マテリアルは描画しつつ切り替わっていきます。そうやないと色分けでけまへんからな?

ところが以前にも話したように



という問題がある。

昨年はこれに対して

「全マテリアルについてのバッファを生成し、それぞれをヒープに乗っけていく」

「そのヒープの参照を切り替えていくことでマテリアルを切り替えていく」

あ、それでさ、昨年のコード見たんだけど、↑の戦略は間違ってないとして…ひどいなー。

やっぱわかつてなかつたんやな……。どういうコードかちょっと見せましょか?

クソコードでごめんなさい

昨年のコード(ママ)

このコードはレンダリング関数内のコードです。

```
for (auto& mat : _materials) {
    *cbuffTemp = *cbuffer;
    cbuffTemp->diffuse = mat.diffuse;
    cbuffTemp->alpha = mat.alpha;
    cbuffTemp->specularity = mat.specularity;
    cbuffTemp->specular = mat.specular;
    cbuffTemp->ambient = mat.ambient;
    cbuffTemp->existTexture = (mat.texture != nullptr);
    cbuffTemp->existSPA = false;
    ID3D12DescriptorHeap* texDescHeap() = { textureCreator.GetDescriptorHeap() };
}
```

```

_commandList->SetDescriptorHeaps(1, texDescHeap);
if (mat.toonIdx >= 0 && mat.toonIdx<10) {
    cbuffTemp->existToon = true;
    _commandList-
>SetGraphicsRootDescriptorTable(_toonTextures(mat.toonIdx+1)->GetRootParameterIndex(),
_toonTextures(mat.toonIdx+1)->GPUDescriptorHandle());
}
else {
    cbuffTemp->existToon = false;
}
if (mat.texture != nullptr) {
    if (mat.texture->GetType() == default) {
        _commandList->SetGraphicsRootDescriptorTable(mat.texture-
>GetRootParameterIndex(), mat.texture->GPUDescriptorHandle());
    }
    else {
        cbuffTemp->existTexture = false;
        cbuffTemp->existSPA = true;
        _commandList->SetGraphicsRootDescriptorTable(mat.texture-
>GetRootParameterIndex(), mat.texture->GPUDescriptorHandle());
    }
}
cbuffTemp = (CBuffer*)((char*)cbuffTemp + GetSizeOf256Alignment(cbuffTemp));
ID3D12DescriptorHeap* descHeaps2() = { cbo->GetDescriptorHeap() };
_commandList->SetDescriptorHeaps(1, descHeaps2());
_commandList->SetGraphicsRootDescriptorTable(cbo->GetParameterIndex(), handle);
handle.ptr += descSize;

_commandList->DrawIndexedInstanced(mat.indexCount, 1, indexOffset, 0, 0);
indexOffset += mat.indexCount;
}
間違ってはない!…間違ってはないねんで?
```

でも…これはひどい…。結局分かってない!…DX11脳のままコーディングしていたことが窺えますね。

まあ仕方ない…仕方ない…昨年は本当に申し訳ございませんでした。が、本当に日本語資料もロクにない中やりながら、授業日数は進んでいくし焦ってたんだよ～。わかつてくれ～。

とはいって、昨年授業を受けてた人でも、もしかしたらこのコードの酷さがわかつてないのかもしれません。

ちょっとここから「経験済み」の人ための解説になるけど…初見の人は察してください。初見でもなんとなく分かるように説明しますし(少なくとも僕はそのつもり)、もしかしたら経験済みの人でも良く分からぬかも知れない。

そこはこれまでやってきたバッファの話とか、デスクリプターヒープの話とかをイメージできているかどうかにかかっている。

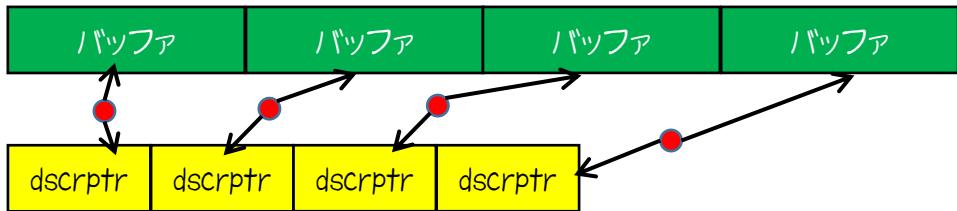
ひとまずやっている事を説明するな? ちなみに、ゼーカー/バッファは(256バイト)*マテリアル数ぶん確保してるし、デスクリプタ数もマテリアル数ぶん確保するで? …もうここでピンと来ると思うけど、

予め確保しておいて、さらにコンスタントバッファビューを作るときに

```
// 指定デスクリプタ数ぶんのビューを作る
for (int i = 0; i < b->_descriptorNum; ++i) {
    D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc = {};
    cbvDesc.BufferLocation = buffLoc;
    buffLoc += (b->_bufferSizeOfOne + 0xff) & ~0xff;
    cbvDesc.SizeInBytes = (b->_bufferSizeOfOne + 0xff) & ~0xff;
    dev->CreateConstantBufferView(&cbvDesc, handle);
    handle.ptr += descSize;
}
```

とやっており、バッファずらしの部分とヒープ上のビューの関連付けを予め行っている。間違つてはいけない。間違つてないけど、もう少しシンプルに書けるはずだし、そもそもレンダリンググループ内でのような事(マテリアルの設定切り替え)をする必要がない。

図に書くと



こうなつとるわけやな。で、レンダリングの時にこれ(ヒープ)を切り替えながら進めとるわけや。

しかしよく考えたら頭が悪い。処理的にもエントロピー(複雑度合い)的にも。処理的に頭が悪い部分とはどこかというと、このバッファへの内容の代入を毎フレーム行っている部分や。既にバッファは確保してるし、関連付けられたデスクリプタも既に用意している…つまりレンダリンググループ外でバッファの中身を埋めてしまえば

```
for (auto& mat : _materials) {
    handle.ptr += descSize;
    _commandList->DrawIndexedInstanced(mat.indexCount, 1, indexOffset, 0, 0);
}
```

で済むはずだよなあ…。あたりまえだよなあ。こうやつたとしてもちょっと気に入らないのはマテリアル数ぶんのビューができることになつてしまふので、何だかなーって思う。

2つの冴えてないやり方

「冴えてない」ってのはもっと冴えたやり方がきっとあるはずだと思うので、こういう書き方をしています。

まず一つ目のやり方は前述のとおり、バッファとビューをマテリアル数ぶん作り、それを切り替えていくというやり方です。

このデメリットとしては、バッファがもつたしない気がするし、ビューももつたしない気がする。

もう一つのやり方としては…コンスタントバッファを Map で更新するのではなく CopyBufferRegion などのリスト系コマンドで投げてやる方法です。その場合

Copy⇒Draw⇒Copy⇒Draw⇒Copy⇒Draw⇒Copy⇒Draw と積んであげれば

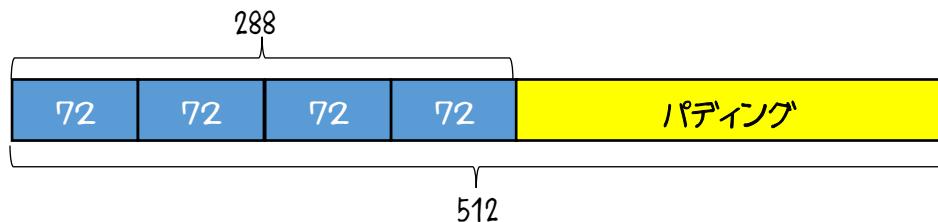


のようにすることができます。これはこれでコピーコマンドが増えることで負担が高まるんじゃないのかと言う懸念もあります。どっちがいいのかは検証してみないとわかりませんね。

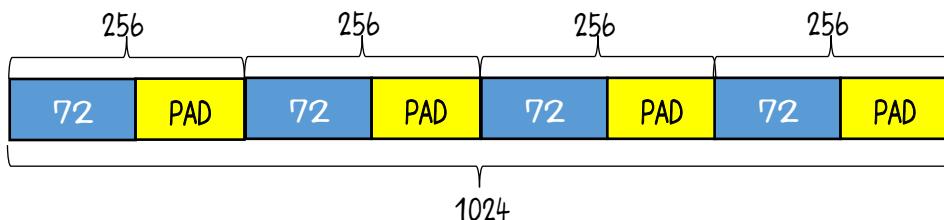
まず、最初のやり方からやってみましょう。

マテリアルのためのバッファ作成

ひとまず前者のやり方を採用して作ってみましょう。マテリアル数分バッファを用意します。今回はディフューズだけ投げてみます。僕の予想が正しければ昨年よりバッファも節約できるかもしれませんし(結論から言うとダメでした)…。



こういう風にはできません。指定 BufferLocation 自体が 256 アライメントである必要があるため(S_OK が返るし、その場では何も起きないけど突然クラッシュする原因となる)



こういう無駄な事になってしまいます。悲しい。こうなると1つ1つバッファ作っても占有領域的には同じことだよなあ…。

最初にルールを決めましょう

- レジスタ番号は1とします(0は既に座標変換で使用しているため)。
- ビューは1マテリアルごとに1つとします。
- ルートパラメータはWVPと別にします。

- もちろんヒープもWVPと別にします。

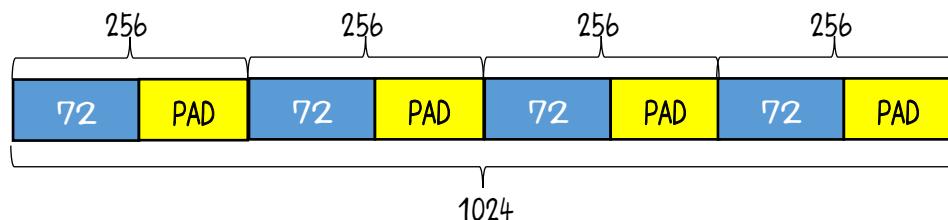
本当にこの「マテリアルの切り替え」は「鬼門」と呼んでもいい。上記の256区切りが本当にバグを引き起こしやすいしたかだが



を表示するまでに大勢死者が出るであろう(予言)。まだテクスチャも貼ってないのに…

まずはバッファを作つていこうか…

どつみち



といった状況なのでそもそもバッファは別々に作つてもいいしまとめてもいい。君の自由だ。
自由選択です。つまり

```
size = (size + 0xff) & ~0xff; // 256責め
auto result = _dev->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
    D3D12_HEAP_FLAGS::D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(size*mats.size()), // 256*マテ数
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&_materialBuff));
);
```

としてもいいし

```

int midx = 0;
for (auto& mbuff : _materialsBuff) {
    auto result = _dev->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
        D3D12_HEAP_FLAGS::D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(size),
        D3D12_RESOURCE_STATE_GENERIC_READ,
        nullptr,
        IID_PPV_ARGS(&mbuff));
    Material* material = nullptr;
    result = mbuff->Map(0, nullptr, (void**)&material);
    *material = mats(midx);
    mbuff->Unmap(0, nullptr);
    ++midx;
}

```

としてもいい。モチロンどっちにするかで、後々の書き方も変わってくるとは思いますが、トータルでのバッファ占有領域も変わらないし、デスクリプタ通してみたらどっちみち同じような見え方するので、自分がやりやすい方法でいいと思います。

今回はビューも分ける予定だし個人的にはバッファを分けてしまった方が考え方的には楽になつていいけかなと思ひます。

このあたり、ホント最終的にはパターンの組み合わせを作って、状況に寄つての最適解を検証する必要はあると思います。今回は分かりやすい方法(マテリアル数ぶんのバッファオブジェクトを作る)を選択します。

これが問題になるようなパターンあるかな?今は思いつかない。

ヒープとビューの作成

ここはバッファの作り方が違うと、この対応も違うので注意してください。

ヒープ自体はもう作れるでしょ?ちょっと作ってみましょうよ。但し今回はデスクリプタの数はマテリアル数と同じなので

```
descHeapDesc.NumDescriptors = mats.size();
```

とやっておく。

もし、バッファをマテリアルごとに分けているならば

```
for (auto& m : mats) {
    desc.BufferLocation = _materialsBuff[idx]->GetGPUVirtualAddress();
    _dev->CreateConstantBufferView(&desc, handle);
    handle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
    ++idx;
}
```

あ…コードを丸写ししても、まったく実力にはならんぞ?書く俺も悪いのかもしれんが何をやっているのかを考えて書いてね?例えば「ここは何をやってるの?」「どうしてこう書いてるの?」って訊かれたら答えられますか?

例えば↑のコードなら

`desc.BufferLocation = _materialsBuff[idx]->GetGPUVirtualAddress();`
ビューとバッファのアドレスをバインドしています。今回はバッファはまとめずに配列管理にしているため、`GetGPUVirtualAddress`をそのまま入れています。

`_dev->CreateConstantBufferView(&desc, handle);`
`desc` 設定を元にヒープ上にビュー情報を作成します。

`handle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);`
ヒープへの書き込み位置を進めています。

と言った具合にです。

なお、1つのバッファでやった場合は

```
for (auto& m : mats) {
    desc.BufferLocation = _materialBuff->GetGPUVirtualAddress();
    _dev->CreateConstantBufferView(&desc, handle);
    desc.BufferLocation += size; //もちろん256アライメントプラスする
    handle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
}
```

えーと、赤字の部分がポイントですが、前にも言ったように、いっぺんにやったところで 256 アライメント並びにせざるを得ないため、バッファロケーションが 256 ごとに並んでいると思ってください。

ルートシグネチャの設定

前にも書きましたが、今回のマテリアルはレジスタ番号を1番とするため、別レンジを作らなければなりません。また、座標変換と寿命というがスコープが違うため、パラメータも別とします。

まずはレンジの設定。簡単ですが追加しときます。

```
//"b1"もつくるぞー
range.NumDescriptors = mats.size(); //マテリアル数
range.RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV; //定数バッファやで
range.BaseShaderRegister = 1; //レジスタ番号は1ですよ
range.OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
ranges.push_back(range);
```

ルートパラメータは別にしたいので

```
rootparam.ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
rootparam.DescriptorTable.NumDescriptorRanges = 1;
rootparam.DescriptorTable.pDescriptorRanges = &ranges.back();
rootparam.ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
rootparams.push_back(rootparam);
```

とにかく追加しとく。もちろん「パラメータ数」も増やしておくことを忘れずに。

これで今回のマテリアルを追加する準備ができました。

シェーダ

あとは shader 側ですが、レジスタ番号1なので

```
cbuffer material : register(b1) {
    float3 diffuse;
}
```

とします。これで diffuse 色が付くので着色します。Brightness に乗算しておいてください。

Draw 時の切り替え

```
unsigned int offset = 0;
auto matHandle = _materialHeap->GetGPUDescriptorHandleForHeapStart();
```

```

ID3D12DescriptorHeap* matdescHeaps() = { _materialHeap };

_cmdList->SetDescriptorHeaps(1, matdescHeaps);

for (auto& m : _model->Materials()) {
    _cmdList->SetGraphicsRootDescriptorTable(1, mathandle);

    mathandle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);

    auto idxcnt = m.face_vert_count;

    _cmdList->DrawIndexedInstanced(idxcnt, 1, offset, 0, 0);

    offset += idxcnt;
}

```

PMD のマテリアルには、そのマテリアルに割り当てるべきインデックスの数が記録されています。

それを利用して色分けを行います。このマテリアルをすべて描画するとすべての面を描画したことになります。

これをマテリアルに分けるためにループを作り、そこに DrawIndexed～を実行します。
描画しつつビューを切り替えていくことで色分けを行うことができます。

うまくいけば…



まあ、色々問題はあるけど色分けはできましたね。

テクスチャを入れよう!!!

さあ、ご覧のように色分けができたところまではいいのですが、目が怖いですね。

実はミクさん。ほとんどはディフューズカラーで色分けできるんですが、目だけはテクスチャでできているんですよね。

FF78 material[5].edge_flag	UU
FF79 material[5].face_vert_count	000001E0
FF7D material[5].texture_file_name[0]	65 79 65 32 2E 62 6D 70 00 FD FD FD FD FD FD
FF80 material[5].texture_file_name[16]	FD FD FD FD
FF91 material[6].diffuse_color[0]	3DED9168 3DED9168 3DED9168
FF9D material[6].alpha	3F800000
FFA1 material[6].alpha	40400000

どうやら eye2.bmp という名前のようにです。

一応絵は



一応 BMP なので、透過は入っていない…そして周囲が黒って事を考えると 0,0,0 は抜く仕様なのかな?

そうは言っても、アルファという事にしちゃうと、深度バッファ問題が発生するため、特に考えずに貼り付けてみましょう。

UV 復活!! UV 復活!! UV 復活!!

一度は封印したテクスチャの設定を復活させます。



ピロリロリロリロッ

Output vs(float4 pos : POSITION, float4 normal:NORMAL, float2 uv:TEXCOORD)

とか

```
struct Output {
    float4 sypos : SV_Position;
    float4 normal:NORMAL;
    float3 pos:POSITION;
    float2 uv : TEXCOORD;
};
```

とか

return float4(tex.Sample(smp, input.uv).rgb*diffuse...

などを復活させます。あとはレイアウトを復活させれば UV データは入ってるはずなので

```
{ "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_INPUT_CLASSI
```

```
FIFICATION_PER_VERTEX_DATA,0}
```

まで復活させねば…



あれれえ? おつかしいなあ。

テクスチャの設定がない奴の取り扱い

まあ、わかつててやったんですけどね。えーとですね。今回のミクモデルに関してですがテクスチャは目のみに貼られていて、それ以外はノーテクスチャでUV値が(0,1)固定なんですね。で、テクスチャがあるとかないとか判断できないため

```
tex.Sample(smp, input.uv).rgb*diffuse
```

というコードを書くと



の左下の色がサンプリングされて、全ての頂点が黒くなるわけです。というわけで全身黒になっているわけ。これを無理やり白っぽいテクスチャにすると



このようにやっぱりミクさんが表示されるわけです

でも、テクスチャがない時にまでテクスチャの影響を受けたくないですよね。さて、どうしたものか…ここは状況によって考えてもらいたいんですが、色々と考えてみました。

- テクスチャが指定されてない場合は白テクスチャを割り当てとく(多分これが簡単だし速い)
- シェーダを切り替える(いやあ…パイプラインステート複数作らにゃいけんし…まあ仕方

ないのかなあ…)

- フラグ管理する(GPU 側にテクスチャ使用するかどうかを教える…ピクセルシェーダがちょっと忙しくなるかな)
- ↑の派生になるが、使用するテクスチャ数を渡す(マルチテクスチャ前提。意外と↑より速いと思われる)

おそらくこれのどれを選ぶかはその時の状況次第だし、これら以外にもやり方はあるだろう。非常に申し訳ないが、正直現場的にどのようなやり方をしているのかなんて想像するしかない。

(僕は現役時代は 2D 屋だったし、テクスチャは必ず 1 枚だけ貼られるという状況だったので、本当に想像するしかない…面白ないと言わざるを得ない。そろそろ僕も教育的なところからはいなくなるべきだと思います…嘘教えたくないしさ)

まあ、ちなみにそれぞれの解説を簡単にしておくと、

最初のやり方については 4×4 くらいの白テクスチャを用意しておいて、それを乗算すればいいと思ってます。

現在のコードが

`tex.Sample(smp, input.uv).rgb * diffuse * brightness`

なので `tex.Sample` の部分が白ならば、`diffuse * brightness` と同じ意味になるからです。

用語テスト～TGS 明けの確認～

- (1) vector とか map の begin()とかで返される「繰り返し」「巡回」に使用する変数を何というか?
- (2) (1)を逆方向から進めるものを何というか?
- (3) vector 型の変数 v に要素を追加するときに通常 push_back を使用するが、一時オブジェクトを作らず直接要素を追加したいときは何というメンバ関数を使うか?
- (4) 入力アセンブラー→頂点シェーダ→テセレータ関連→ジオメトリシェーダ→???→ピクセルシェーダ→出力
のような一連の流れの事を何というか?
- (5) (4)において???に入る言葉は何か?
- (6) DirectX12において、画面をフリップする役割を持つオブジェクトを何というか?
- (7) DirectX12において、データをレンダリングして、最終的に出力されるターゲットの事を何というか?
- (8) DirectX12 のコマンドは基本的に即時復帰となっている。このため特定の部分で「待ち」をしておかないと面倒なことになる。これを制御するために DirectX12 に組み込まれているオブジェクトは何という名前か?
- (9) DirectX10 以降においてはデータに対して、その「データの解釈」という設定がペアになっている。この設定の事を一言で何というか?

- (10) 実際にポリゴンなどをレンダリングするためには様々な種類の(9)を用意する必要がある。(9)やサンプラーなどと一緒にまとめて扱うための上位概念が DirectX12 から導入されたが、それは何というか?
- (11) (10)をメモリ上に配置するために確保される記憶領域の事を何というか?
- (12) DirectX10 以降では、ポリゴン1枚表示するのにも C++ ではないが、C/C++ によく似たプログラムで GPU が解釈するためのプログラムを書く必要がある。この事を何というか?
- (13) DirectXにおいて(12)を表現するための言語を何というか?
- (14) DirectX では、頂点だけだとデータの無駄が発生するために頂点の組み合わせからポリゴンを構成するためのデータを利用するようになった。このデータの事を何というか?
- (15)(10)や(11)を(12)で利用するためにレジスターというものを使用し、その範囲を設定するためにレンジというオブジェクトがある。これをまとめたためのオブジェクトを何というか?
- (16) 複数の(15)やサンプラーをまとめたための DirectX12 から導入されたオブジェクトを何というか?
- (17) 頂点データは座標や法線や UV の情報が含まれており、そのままではデータの塊に過ぎないため利用できない。このため頂点○○○○○を設定する必要がある。それは何か?
- (18)(17)を設定する際に POSITION だの TEXCOORD だの NORMAL だのと言った文字列でその意味を明確にしておく。この事を○○○○。○○という。何か?