



まずうちさあ…

方針を説明したいんだけど…聞いていいかい?ああ~いいっすねえ~。

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

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

ここまで承知の上でこの授業を受講したい人はそのまま受けただけだとありがたいと思います。

最近の流れ

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

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

GTX と RTX の違い

RTX の特徴としては

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

レイトレーシングとは?

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

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

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

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

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

授業の流れ

授業の流れは大雑把に言うとこんな感じです。

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

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

もうちょっと細かく言うと

1. DirectX12ポリゴンを出すまでがんばる(面倒だしシェーダが必要だし即死)
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/ド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. テクスチャローダを作る
12. ポージングさせる
13. アニメーションさせる(リバースイテレータ登場!!!)
14. ベジエで動かす(ニュートン法、二分法)
15. つぶれ影表示(行列で演して黒く塗るだけ)
16. シャドウマップでセルフシャドウ(シャドウアクネがさ…)
17. 簡易トゥーンレンダリング
18. 輪郭線
19. アンチエイリアシング(輪郭線との相性最悪)
20. IK(いけるかな…)
21. ポストエフェクト(をするために必要な事)
22. 色調整(ポストエフェクト)
23. 画面を割る(法線テクスチャ+ポストエフェクト)
24. ガウスぼかし
25. ブルーム(縮小ドッファ+ポストエフェクト(ガウス))
26. 被写界深度(深度値+縮小ドッファ+ポストエフェクト(ガウス))
27. imgui組み込み
28. ディファードレンダリング
29. インスタンシングで大量表示
30. SSAO(スーパー・ソードアートオンラインではない…冬休み中にはできるかな)
31. SSR(ガチャのことではない…レリ! 加減にしろ!!)
32. インバースキネマティクス
33. 太陽光産卵(散乱)
34. 法線マップ(接ベクトルと従法線ベクトルが必要なんだよなあ…)
35. コンピュートシェーダ(いけるのか?)

36. DirectXRaytracing

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

目次

まずうちさあ…	1
最近の流れ	1
GTX と RTX の違い	2
授業の流れ	2
環境構築	12
ウィンドウ表示	15
HINSTANCE とか HWND とか	15
じゃあ実装	16
解説	20
基礎知識説明①	22
シェーダ	22
頂点シェーダ	23
ピクセルシェーダ	24
ジオメトリシェーダ	24
ハルシェーダ(テセレーション)ドメインシェーダ	25
コンピュートシェーダ(GPGPU)	26
レンダリングパイプラインについて	28
ちょっとしたハードウェアの知識	30
GPU と CPU の違いについて	30
キャッシュメモリについて	34
DirectX 組み込みに入る前に	37
DirectX12 がそれ以前の DX と違うのはどこ?ここ?	38
仮想メモリ(仮想アドレス)とは	40
キャッシュメモリとか分岐予測とか	42
とにかく DirectX12 を動かそう(初期化編)	44
準備①(インクルードとリンク)	45
基本的な部分の初期化	45
画面に影響を与える準備	49
スワップチェイン	50

レンダーターゲットの作成.....	57
さて、いよいよ画面のクリアだ.....	62
コマンドを投げるために…	62
コマンドリストとコマンドアロケータをリセット	63
コマンド:レンダーターゲットを設定.....	65
コマンド:レンダーターゲットをクリア.....	66
コマンド:クローズ.....	66
コマンドキューに投げる.....	66
スワップチェーン Present.....	67
実は色々間違ってるんです	67
フェンス	68
ではフェンスを実装しようか	73
デバッグレイヤーを有効にする.....	76
リソースバリア	77
ポリゴンを表示しよう.....	79
頂点を作ろう	79
頂点バッファ	81
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	92
ルートシグネチャー	93
頂点レイアウト	100
パイプラインステートオブジェクト(PSO)	103
その他やらなければならない事	106
リソースバリア	107
ビューポート	107
残り色々セット	109
ドロー!!ポリゴン!!!	109
うまくいかない場合	112
アプリがグラボを選ぶズエ…レリズエ…	112
四角形ポリゴンにしてみよう	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	115
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	116
ドロー(インデックスあり)	116

そんな事よりテクスチャ貼るうぜ.....	117
頂点情報にUVを追加.....	117
頂点シェーダ変更.....	117
テクスチャオブジェクト生成.....	118
書き込み	124
バリアとフェンス.....	125
シェーダリソースビューを作る.....	125
サンプラーを設定.....	128
シェーダにテクスチャの受け取り側を記述する.....	128
ルートシグネチャを設定.....	129
毎フレームやること.....	130
画像ファイルを読み込んで表示する.....	133
CopyTextureRegionによる転送.....	139
d3dx12.h(CD3DX～)を導入する.....	146
行列で座標変換してみよう.....	150
行列おさらい.....	150
2D座標変換行列.....	151
定数バッファ	155
CPP側	155
シェーダ側	158
3D化してみる.....	158
XMVECTORについて	160
リファクタリング.....	162
ComPtrを使う.....	162
色々関数化する(コメントをきちんと書く).....	164
デスクリプター(テクスチャ、定数)まわりを支える設計.....	167
ヘッダだけ公開.....	171
ともかくPMDモデルを表示させよう.....	175
フォーマットを確認する.....	175
ヘッダ	175
頂点リスト	178
とりあえずモデル表示してみよう	179
BadAppleにしてみる.....	180
インデックス情報を読み込みましょう	181
シェーディングしてみる.....	182
深度バッファ	184

深度/ドッファとは.....	184
結局 DX12 では何をしなければならないの?	186
深度/ドッファの作成.....	187
深度/ドッファビューの作成.....	188
パイプラインステートオブジェクトに深度情報を追加.....	188
レンダーターゲットと深度/ドッファを関連付け.....	189
深度/ドッファをクリア(毎フレーム).....	190
法線も座標変換.....	190
マテリアルを適用.....	192
マテリアルってなんや?	193
マテリアルデータ読み込み.....	194
クソコードでごめんなさい.....	196
2つの冴えてないやり方.....	199
マテリアルのためのドッファ作成.....	200
ヒープとビューの作成.....	202
ルートシグネチャの設定.....	204
シェーダ.....	204
Draw 時の切り替え.....	204
1項: マテリアルに合わせてテクスチャを貼る	210
2項: テクスチャの有無でマテリアルに不具合が起きないように	220
3項: 他のモデルも試してみよう(特殊なテクスチャファイル指定)	224
4項: スペキュラとアンビエントの実装.....	235
5項: テクスチャファイルが tga や dds だった時の対応	239
6項: トーンシェーディング	242
リファクタリング.....	250
便利なクラスや構造体やマクロを積極的に使用する.....	251
用途によって簡単な分類を行う	256
ポージングしようぜエ…(レリーズエ…)	263
概要	264
ボーンとはいったい…	264
スキニング(スキンメッシュアニメーション)	264
ツリー構造と再帰	267
具体的にどういうやり方でポージングしていくの?	270
ボーン情報をロード	272
ツリーを構築	273
特定ボーンの回転	277

準備	277
実験(だいたい)結果が予想できるやつ).....	282
ボーン中心回転(原点中心回転ではない).....	282
子々孫々末代まで回転を伝播する.....	283
VMD ファイルを読み込む.....	286
クオータニオンって何ですかねえ…	287
データを読み込む.....	289
データの加工.....	290
クライアント側.....	291
アニメーションしてまう.....	293
フレーム補間しよう.....	299
リバースイテレータと base().....	300
あれ?	301
球面線形補間.....	303
VMD モーションデータの罠	306
補間曲線	309
ベジェ曲線…とは?	310
ニュートン法(ニュートン・ラフソン法)	312
ベジェによるイーズインイーズアウトを実装してみよう	319
その他今のうちにやっておきたい事	322
可変フレームレート状態で拳動を合わせる.....	322
マルチパスレンダリング	323
大雑把な解説	324
最初の実践	324
ビュー用ヒープ作る.....	325
リソース作る.....	326
ビュー作る	326
レンダーターゲット切り替え	326
ペラポリ作る.....	327
ペラポリ表示用ルートシグネチャを作る	328
ペラポリシェーダを作る	329
ペラポリ用レイアウトを作る	330
ペラポリ用パイプラインステートを作る	330
ペラポリ表示部分を作る	330
ここまで間違いやすいポイント	331
加工してみよう	332

モノクロ化	332
反転	333
ポスタリゼーション?	333
軽い単純ぼかし(平均化)	334
エンボス	334
シャープネス(エッジ強調)	335
簡単な画像処理的輪郭線抽出	336
ガウシアンぼかし(簡易版)	337
ガウシアンぼかし(ちゃんとしたやつ)	339
ガラスフィルタ(画面歪み)シェーダ	347
シャドウマップ	352
影行列(潰し影)…ウソ影	352
シャドウマップの導入(マルチパスの応用編)	362
やつてもやつてもバグが取れないるので、リファクタリング	363
土日を犠牲にしてリファクタリングした結果	364
ズバッと解決	365
シャドウマップのしくみ	367
準備	368
概要	368
設計	369
実装	369
シャドウマップ本編	374
手順	374
ライトビュー行列を追加	375
ノットファの確保	375
ライトの「とりあえずの」座標を決める	378
ライトからの描画	381
悪夢の深度値比較…!	383
深度値と距離を同じ土俵に…	383
UV値はどうするのか?	384
比較	385
セルフシャドウ	386
簡単なPCF(percentage-closest-filtering)で影をマシに	388
Effekseer組み込み	391
とにかく必要なものを集めてくる	391
一旦実験までやってみよう	393

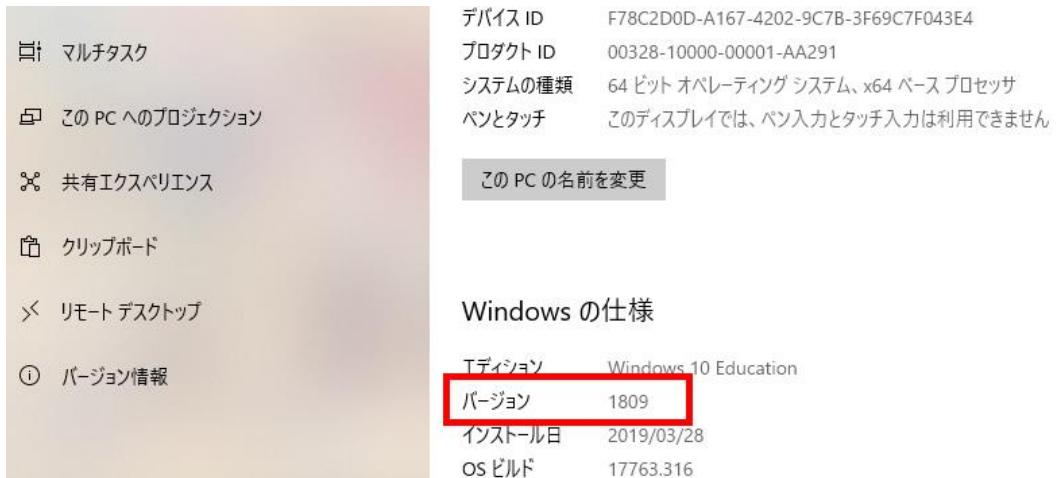
DDS ファイルのロード	397
DXTC, ETC, PVRTC	397
臨時の話	403
リアルタイムレイトレーシングのセッショングについて	403
輪郭線的なやつ	405
試しにリムライトの考え方を応用してみよう	405
トゥーンマップを用いる方法	407
背面法(反転法?)	407
輪郭線抽出フィルター	415
深度パスレンダリング結果を応用	418
アンチエイリアシング	423
はじめに	423
SSAA(SuperSampling Anti-Aliasing)	424
MSAA(MultiSampling Anti-Aliasing)	426
FXAA?(Fast Approximate Anti-Aliasing)	428
ガチめ FXAA	430
インバースキネマティクス(IK)	434
インバースキネマティクス(IK)とは	434
高校数学(余弦定理)を用いた簡易的なインバースキネマティクス	436
CCD-IK のしくみ	440
2D における CCD-IK 実装について	443
PMD における CCD-IK について	448
その他 VMD や IK の仕様を考慮する	471
マルチレンダーターゲットとその応用について	477
ピクセルシェーダの出力を複数にする(色, 法線)	477
色情報と法線情報からディファードシェーディングを行う	481
高輝度成分抽出とぼかしからブルーム(光の漏れ)を実装	484
ブルームの原理	484
ブルームの準備	486
縮小リッパへの書き込み	489
深度値からぼかす範囲を決め簡易的な被写界深度を実装	494
被写界深度について	494
CG における被写界深度	495
プログラムによる実装	496
スクリーンスペースアンビエントオクルージョン(SSAO)	501
アンビエントオクルージョンとは	501

数式から考えるアンビエントオクルージョン.....	502
スクリーンスペースアンビエントオクルージョン(SSAO)の実装.....	507
アンビエントオクルージョンまで一気に実装.....	509
imgui	518
imguiについて.....	518
imguiの組み込み.....	519
imguiの活用例.....	530
DirectXTK	538
DirectXTKの概要.....	538
DirectXTK12の入手.....	538
DirectXTKの組み込み.....	540
特定ののフォントで特定の文字列を表示する.....	541
さまざまなフォント、文字列を表示してみよう.....	544
レイマーチング	547
レイマーチング基本というか初歩の話.....	547
2Dにおける距離関数.....	548
fmodで大量生産.....	550
3Dにおける距離関数とレイマーチング	553
法線ベクトルを返してみる.....	555
そろそろ提出物.....	556

まあとはいっては作る環境を整えなくちゃね。

環境構築

今この教室の OS の設定はこんな感じになっています。



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

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

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

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

あと、後述する d3dx12.h や DirectXEx などの対応バージョンが食い違うと動作しないので気を付けましょう。ちょっと色々とバージョン関連がややこしいので、おうちの PC で環境構築する時には気を付けましょう。ちなみに最新バージョンの SDK は

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

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

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

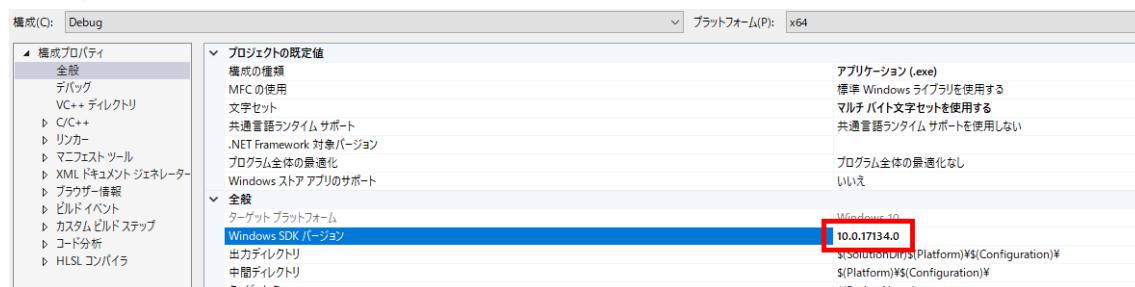
過去の SDK のバージョンへのリンクも一応貼っておきますので、家の PC が不安な人は 1803 に合わせておいた方が無難でしょう。

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

まあ、最初の方は Window の構築だけなので、しばらくはそこ関係ないので、ぼちぼち環境構築していくください。

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

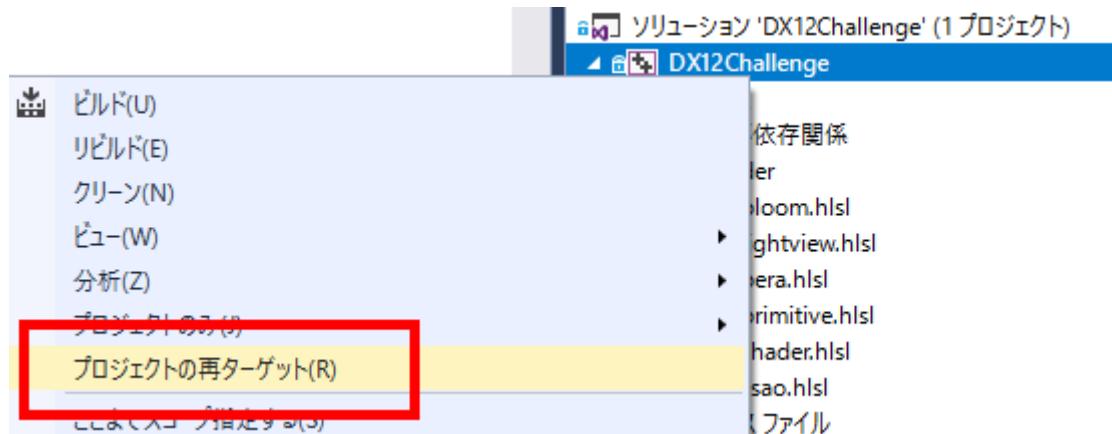
ちなみに VS 上での SDK バージョンの確認方法ですが



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

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

その場合は対象プロジェクトを右クリックして「再ターゲット」をしてください。



一応それで大丈夫なはずです。

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

ウィンドウ表示

何はともあれウィンドウの表示を行いましょう。表示自体は DxLib なら DxLib_Init()で終わりだったんですけどね。Windows API の場合、そうはいきません。とりあえず一つものように main.cpp を作って、main 関数もしくは WinMain 関数を作つておいてください。

次に Application クラス作りましょう。シングルトンで作つときましようか。DxLib の時と似たような感じで作つていきます。とりあえず

Initialize()

Run()

Terminate()

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

で、ウィンドウ作るときにやたらと「ハンドル」ってのが出てきます。

HINSTANCE とか HWND とか

HINSTANCE や HWND の頭にある H というのは Handle の略です。いろいろと例え方があると思いますが、それを操作するためのモノということで「ハンドル」って名前がついてゐるのです。「ハンドル」というと



こういうものを想像すると思いますが、とくに HWND に関してはウィンドウ(車)を操作するために必要な「鍵」と思つてもらつたほうがいいと思います。



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が戻り値もこれですね。あれは int で使いやす

いけどね。

ただ、Windows プログラミングにおいてこいつの型は単なる整数型(というかアドレス型)のくせに windows.h で typedef だかなんだがやつてるせいで 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つて、ちょっとイヤ。

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

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

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

で選択肢ですが

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

正直ここは後々の拡張性まで考えて、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。

じゃあ実装

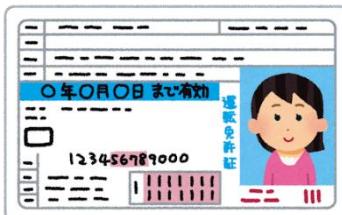
とりあえず Windows のウィンドウを作るのに、DxLib の時は DxLib_Init で済んでたんだけど(ホントはそれだけじゃなくてデバイスとかその他初期化してくれてる)、ウィンドウを作るのは、まずは Windows に自分の身分証明をする必要があります。

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

はい、このWNDCLASSEXという構造体は身分証明書みたいなもんです。

レンタカー屋さん(Windows)に車を借りるためにまず身分証明書を提出しなければなりません。

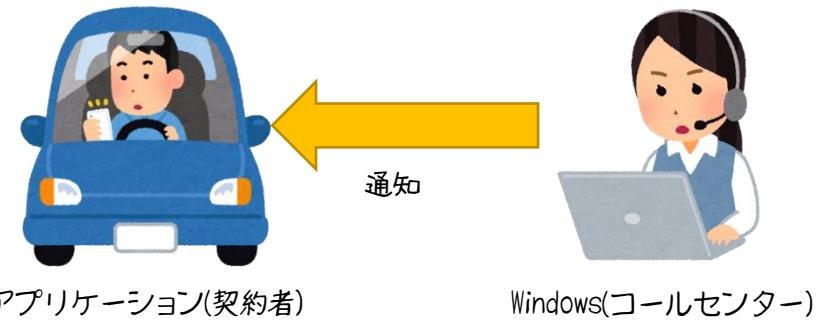
犯罪に使われたら困りますからね…



身分証明書(WNDCLASS)

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

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



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

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

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

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

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

このくらいのコードが必要になる。

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

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

もし失敗した時にキャッチできるよう

```
if (hwnd == nullptr) {  
    LPVOID mssageBuffer = nullptr;  
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |  
        FORMAT_MESSAGE_IGNORE_INSERTS,  
        nullptr,  
        GetLastError(),  
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),  
        (LPWSTR)&mssageBuffer,  
        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、一応ウイ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。

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るイメー
ジで。

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

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

こんな感じでループ抜けを書いておく。

で、Terminate()あたりに

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

と書けば、一応ウイ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。ウイ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どかなあかんのですよ。ということで定義
//めんどくせーし、あまりゲームに関係ないけど書かなあかんやつ

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

こいつはクラス内関数やなくて、通常の関数として宣言しといてください。結果的には
main.cpp が

```
#include "Application.h"

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

このようになるようにしておいてください。

解説

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こぴーしてくると

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のなのですが、Handle-Body イディオムとも呼ばれるんですが意味合い的には DxLib におけるグラフィックスハンドルみたいなもんです。あれはロードした絵を操作するためのものでしたが、今回はアプリケーションを操作するための「ハンドル」だと思ってください。持ってくる方法は至って簡単

ウイ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。このハンドルはウイ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…

ウインドウを表示するのは「アプリケーション自身」に思えますが、実際は「OS(Windows)」です。ちょっと難しい概念なんですけどね。ディスプレイやマウスやキーボードやスピーカーなどのデバイス周りを制御するのは OS なんですよ。モバイル機器でも同様なんですが、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「ウインドウ」を表示するのは OS の役割であり、

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

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

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

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

GetModuleHandleという関数で取得できます。

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

あと、この授業を受けるときには徹底してほしいことが一つあって、それは知らない関数が出てきたら、MSDNの関数を必ず確認しようです。OS周りや DirectX周りの関数は結構罠が多くて、きちんと読まない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に聞く(先生でも友人でもいいので、分らないままにしない事)
- 休まないよう(基本的に、休むとワケ分からぬ事になります。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(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らないようにしてください。寝ててついていけない奴をフォローしません)
- 授業中のトイレも同様です。トイレに行っても基本授業は止まりません。授業中にブリュリュ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はなく他校のブースをスパイしましょう。

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

手順が

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

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

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

基礎知識説明①

シェーダ

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(リテックスシェーダ)
 - 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) DS:ドメインシェーダ
 - CS:コンピュートシェーダ
- などの種類があります。

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

という事で、みなさん、この DX12 の授業ではシェーダは避けて通れないんです。フヒヒ(Dx11 の頃からシェーダは必須でしたけどね)

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

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

レンダリングパイプラインについてはのちほど解説します。

頂点シェーダ

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

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

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 側から渡してやつて、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てきます)。

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

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

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った。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。

言ってしまうと、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?

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つたりします。

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

ちなみにこういう事も出来るっぽいです。

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

ああ～楽しそうなんじゃよ～。

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

ジオメトリシェーダがいらないなくなっちゃうや／＼いや／＼…。

ハルシェーダ(テセレーション)ドメインシェーダ

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」とは何かをお話しします。

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

正直、使ったことがないので良く分からぬいんですが、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リ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)

これは何かといふと、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…？

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

先にも言いましたが、レンダリングパイプラインの流れの外にあります。ではなぜシェーダなのかと言うと、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 といふのはホンマは 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 は速いといふのがゲームや CG 業界以外にも知れ渡つてしまつて、数値計算やディープラーニングや、仮想通貨マイニングに使われるようになつてゐるわけです。

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

使つたことないのであまり言うとぼろが出そなうですが、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 が速いと言わされているのかと言うと、画像の描画に特化して進化してきたため演算自体にそれほど複雑な計算が必要ないコアを「大量に」並べることで高速化を図ってきたのです。

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

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

やからあんまり複雑な命令を出すといいくら 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。

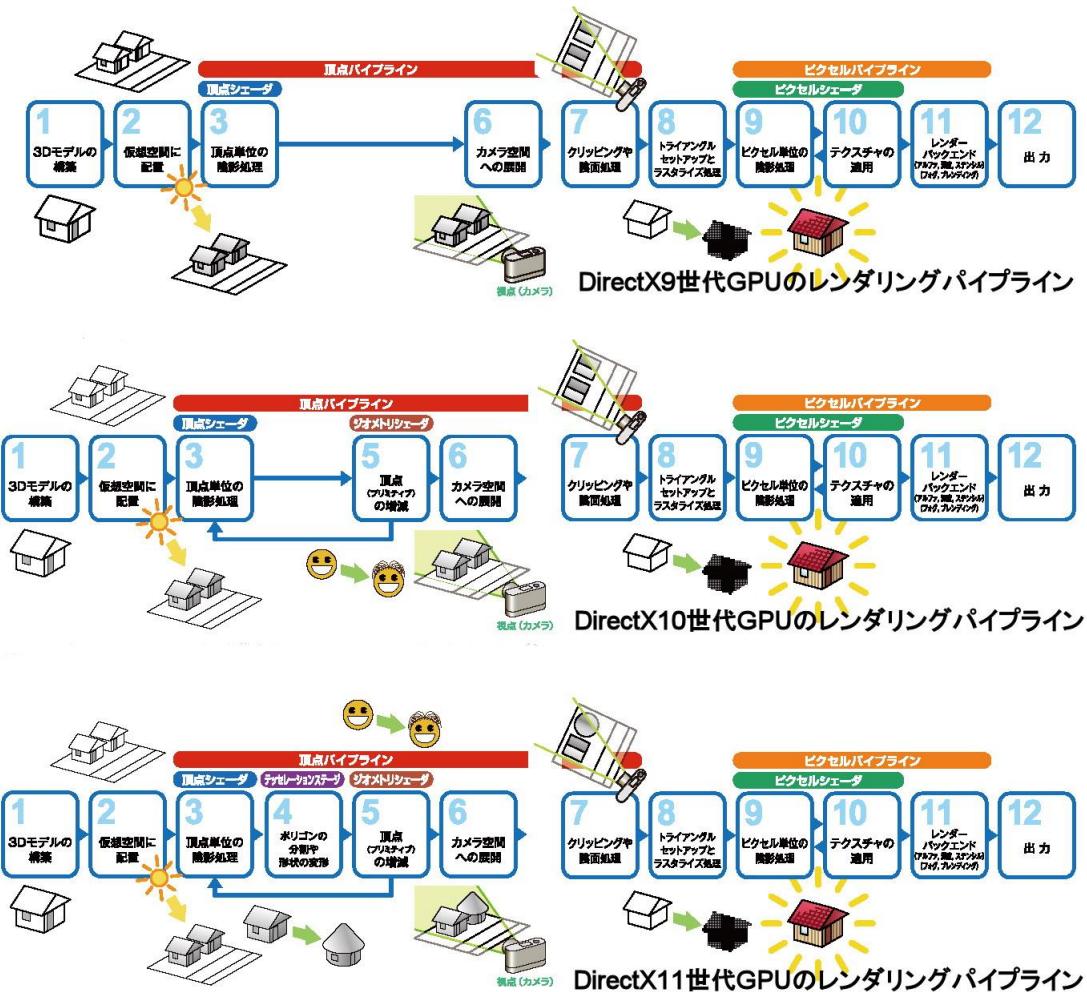
CPU と GPU はそういう違いがあると思っておいてください。だから、GPU は大量の頂点とか、大量のピクセルとかを処理するのが得意なわけやね。

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

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

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ックってコピペします。



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

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

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

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

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

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

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

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

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

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

ちょっとしたハードウェアの知識

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

GPU と CPU の違いについて

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

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

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

とはいっても最初はシェーダを書くことしかなかったし、あまり意識しなかつたんだけど DirectX9あたりでシェーダを扱うようになってからは GPU を意識するようになってきました。

さて、この CPU と GPU は何が違うのか? 「とにかく GPU を使つたら速いんじゃないの?」と思つてゐる人もいるかもしれません、ちょっと違います。もちろん速いのですが得意とする所が違います。まずは正確さはともかく簡単にイメージを書くと



CPU は難しい問題が解ける博士で



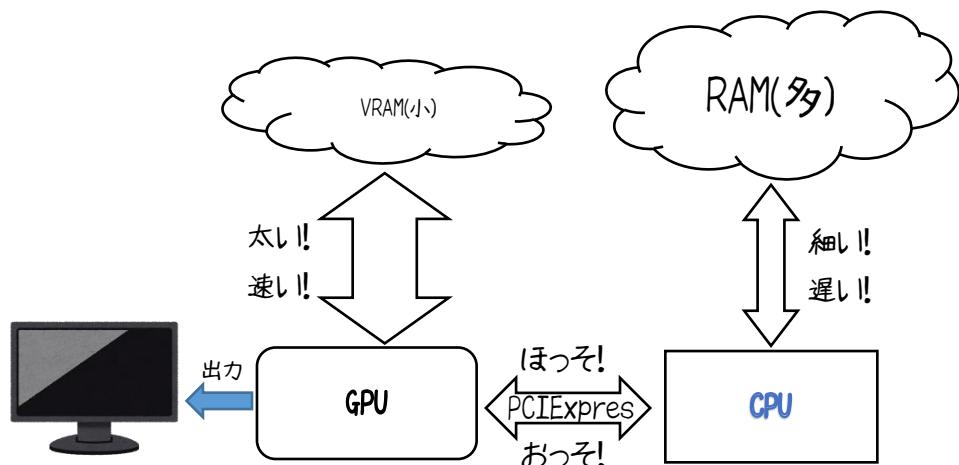
GPU は簡単な計算ができる「集団」
といったイメージを持ってもらうといいかなと思います。
というわけで得意なことが違うんですよね。

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

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

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

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



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

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

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

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

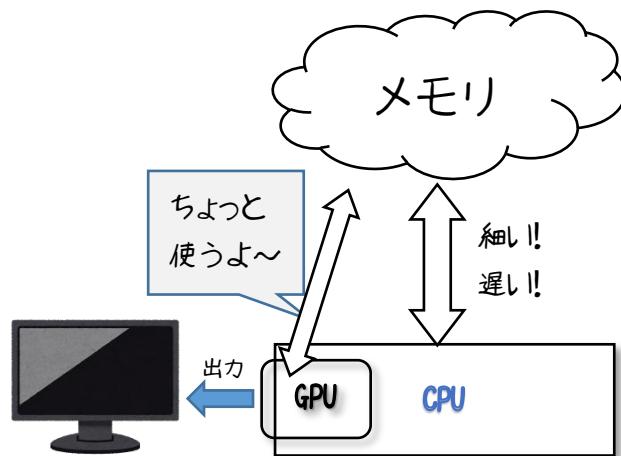
イメージでいうとバスは道路(文字通り乗るバスでも可)で、バンド幅ってのはその道路の幅(もしくはバスの本数)って考えてもらえばいいんじゃないかな。ほら、道幅が広いとさ、それだけ一度に通る車(情報量)が増えるじゃん？

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

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

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

このためメインメモリとVRAMが共用されています。というかメインメモリの一部をVRAMと



して使用するといった感じです。

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

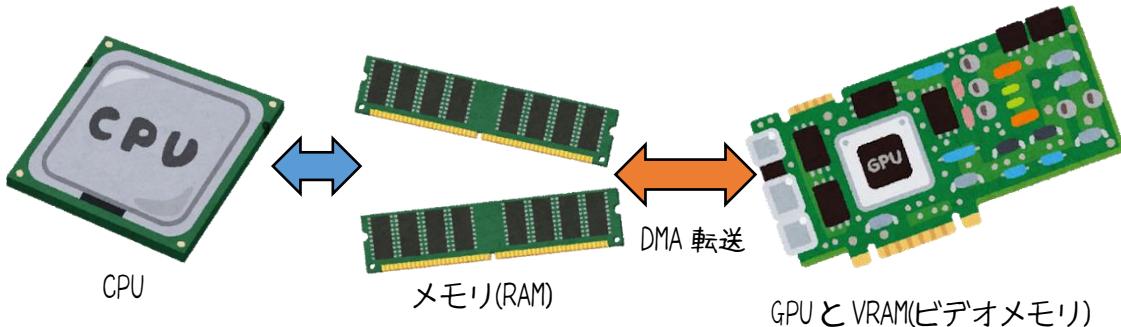
DirectX12 では UMA に関するパラメータもあつたりするので、頭の片隅にでも置いておいてください。

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

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

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

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



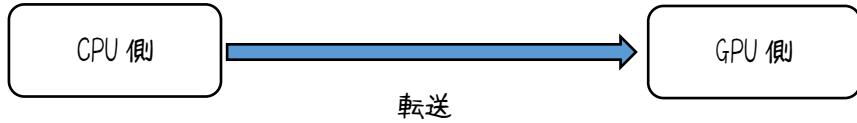
しつと重要なことを書いてるんですけどね…まあひとまずはこういう関係になってると思ってください。

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

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

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

ということは…



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

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

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

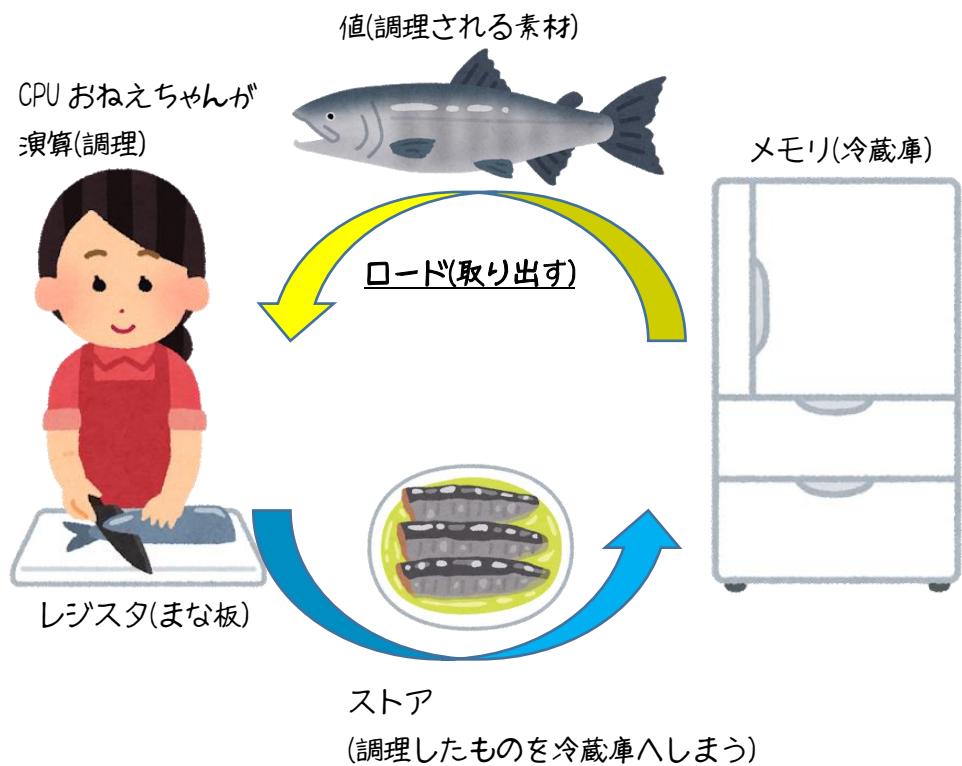
このため、PCIExpress(簡単に言うと CPU 側と GPU 側をつなぐ線。実際にはマザーボードにくっついてるやつ)には DMA エンジンという転送のための仕組みがあり、CPU はそいつに対して「転送してくれー」と命を出します。そしたら CPU の邪魔をすることなく転送が行われ、いつかは GPU にデータがコピーされます。

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

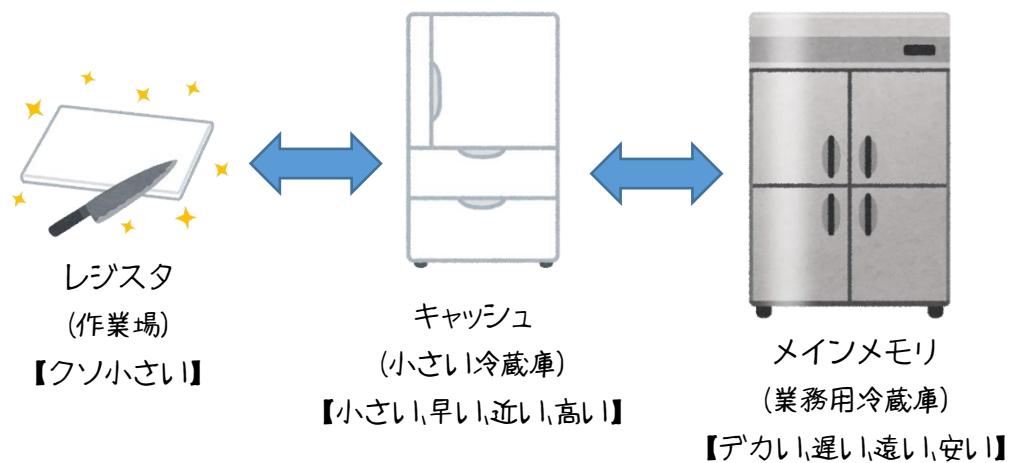
安いノート PC なんかはこれですね。場合によっては、省電力のために通常は HDGraphics のほうを使用し、負荷の高いゲームの場合は nVidia や Radeon などの専用グラボを使うことがあります。

キャッシュメモリについて

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



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



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

分かりやすいところでいようとループ内の処理とかですね。

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

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

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

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

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

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

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

としたほうが効率的になります。なお、キャッシュがそんなに小さいことはないので、これが効

率的とは思わないでください。あくまでも例です。

DirectX組み込みに入る前に

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

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

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

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

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

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

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

「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/483c60fa06bf10f510d7>

とか

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

とか

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

ついでに

<https://qiita.com/Onbashira/items/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』しますと書いてればいいかな。理解せずに使用するってのがいちばん嫌われる。

あと、ZeroMemory 使ってた参考書や参考サイトあるけど、あれ使うのは、やめようね!ダメ! ゼッタイ!

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

あと…喜べ! 公式がついに日本語訳し始めたぞ!!!

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

DirectX12 がそれ以前の DX と違うのはどこ? ここ?

とりあえずこの授業を聞いている人の中に DirectX11 の授業を受けた人がいないんだよね(そんな時代になつたんだなあ…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ぬと思います。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!!!って思つ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、どっちかっちゅうと DirectX11 時代の問題点に触れておいた方がいいのかもしれん。恐らく皆さんが DX11 を直接いじることはもうないと思いますのでお話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。

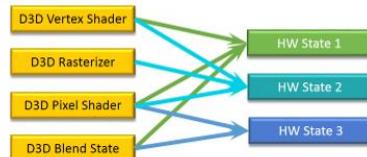
イメージわくかな?

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



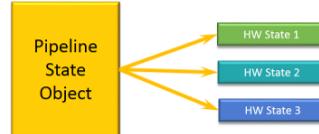
図で説明されてるサイトとかだとこんな感じですね。何となく切り替えコストが無駄になっているのが伝わるかなーと思います。何となくでいいですよ? 無理に理解しようとしなくていいです。

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ださい。

Direct3D 12 – Pipeline State Optimization

Group pipeline into single object

Copy from PSO to Hardware State



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

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

コマンドリスト、コマンドキュー、デスクリプターヒープなどが出でますが、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

とまあ学習のハードルが上がっただけに思えますが、慣れ所も挙げておくけど
『思想に慣れたら、DirectX12 の方が楽に感じる』ので、さっさと慣れましょう。慣れるしか…ないつ!!

仮想メモリ(仮想アドレス)とは

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

で、仮想メモリってことはバーチャルなメモリなん？って思う人もいると思いますが、その通りです。実際のメモリとは違うが、実際のように使用できるメモリの事です。どういうことが」ということ…

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

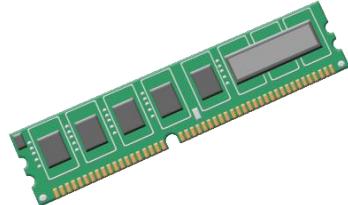
<https://ja.wikipedia.org/wiki/%E4%BB%AE%E6%83%B3%E8%A8%98%E6%86%B6>

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

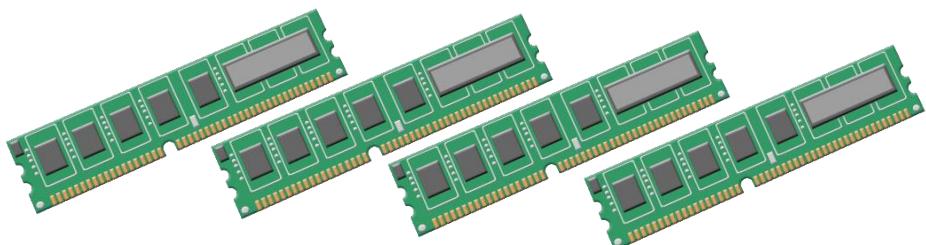
仕組みとしては、物理メモリをそのまま使うより、OSっていうかMMU(メモリマネジメントユニット)がマネジメントした「仮想メモリ」を見たほうが便利なので、基本的にプログラムはこの仮想メモリを通して物理メモリにアクセスしています。

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

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



で、もちろん一本差しではなく、2つ3つ4つ刺さっています。



大雑把に言うと大きなメモリ確保の場合、1本では足りなかつたりメモリ間を跨げたりするわけ。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のかもしれません。そこはその時の説明の文章(英語?)を見ないと分かりませんが、とにかくドライバの中身をいじらない限りは物理アドレスに直接アクセスはできませんので、あまり用語に捕らわれることなく、普通にプログラムすればいいと思います。

キャッシュメモリとか分岐予測とか

ここからはマニアックすぎるの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たりするんだ。大元の仕組みが今みたいなブロードバンドの時代じゃなくて、ダイヤルアップ回線使ってたナローバンドの時代だからこういう風になってるんだけど、昔は本当に重宝してたんだ。本当にクソ遅かったから。

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

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

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

メモリ上のデータから、頻繁にアクセスするものをより分けてそれぞれのキャッシュメモリ

に置くことで、同じような数値の同じような計算を高速化している。

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

だからゲームプログラマは良く「キャッシュに載るように」とかなんとか言う。

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

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

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

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

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

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

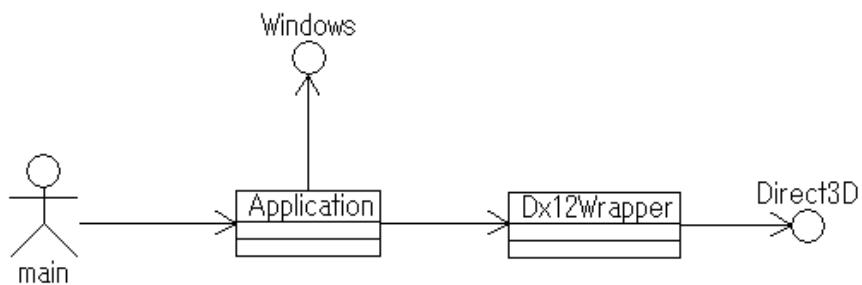
ちなみに for ループ処理の場合、毎回ループ条件が同じであるためほとんどの分岐予測が当たります。N 回ループなら N-1 回は必ず分岐予測が当たるわけです。

ともかく switch 文は要らないってことでいいですね。

とにかく DirectX12 を動かそう(初期化編)

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

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



こういうふうにしたい

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

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

基本初期化として

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

を設定して、画面に影響を与えるためには

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

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

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

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

ということで

- D3D12DescriptorHeap
- D3D12CommandList(と D3D12CommandAllocator)
- D3D12CommandQueue

の初期化が必要となります。

で、先にも書きましたけど、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(実際にはスワップチェインを生成した時点でレンダーターゲット用のリソースは作成済みであり、スワップチェインから取得することになる。しかし、別レンダーターゲットを使用する際には自分でリソースを確保しなければならないことは覚えておこう)

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

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

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

準備①(インクルードとリンク)

とりあえず必要なのは direct3d12 の定義なので

#include<d3d12.h>をします。

もうひとつおまけに

#include <dxgi1_6.h>します

次にリンクするために以下のコードをどつかのcppにリンクコードを書きます。

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

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

基本的な部分の初期化

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

をします。

で、メンバ変数として最低限

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

が必要になってくるわけだが、さて…まあ、インクルード問題…ぐぬぬ。

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

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

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

はOKというルールにするかな。あんまし無理してもな…(｀；ω；｀)

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

さて、ということでデバイスを生成します。

D3D12CreateDeviceって関数です。

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

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

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

```
HRESULT D3D12CreateDevice(  
    IUnknown* pAdapter, //nullptrでお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れど、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をしましょう。

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

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

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

で、可能な限り新しいバージョンを使いたい場合にはどう書いたらいいのでしょうか？ちなみにハードウェアがそのフィーチャレベルに対応していないければ 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？対応していないければ失敗することが分かっているんだから、高いレベルから試せばいい。つまり、

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

```
};
```

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

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

まあ、学校の PC ならどれか引つかかるんで…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。

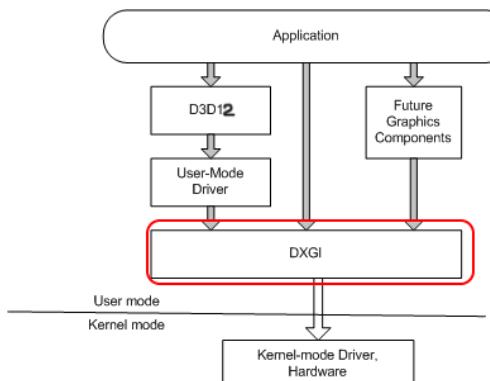
分からなかったり、納得できない人はすぐに言ってください。フォローの講義をしますので…このへん納得できないまま進むと死ぬんで遠慮なく聞いてください。

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

で、ここで出てくる 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)

なんですか日本語やな…

HRESULT CreateDXGIFactory1(

REFIID riid,

void **ppFactory

);

引数はデバイスの生成の時と同じなので問題なさそうなんですが、行けるんかなあ…ホンマDirectX12の仕様とマニュアルレベルが減固めろや…。

で通るし、S_OK返ってくるしでいいんだろうけど…納得いかん。

はき

ちなみにCreateDXGIFactory2ってのもあるんだけど、こいつは

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

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

画面に影響を与える準備

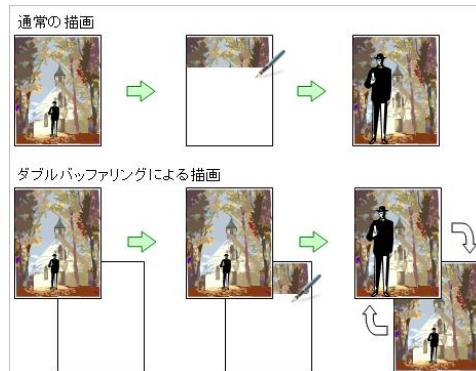
画面に影響を与えるためには前にも書きましたがまず表示画面のための

- 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
- デスクリプターヒープ
さらにさらにそれをまとめるための
- デスクリプターテーブル
- ルートシグネチャ
最後にレンダリングパイプラインをまとめた
- パイプラインステートオブジェクト
まとめまくりですねー。でもまだあるんだごめんね。
前にも言ったように命令が即時復帰のために待ちの仕組みを用意してあげなきゃならぬ
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 や似た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？

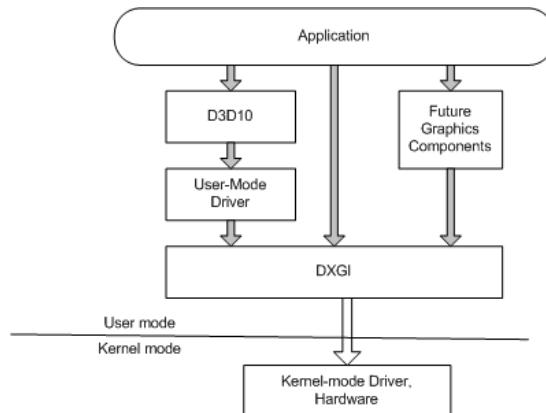
うーん。まだわからんかな？

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)



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)

定義を見るとこうなってますね？

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

DXGI_FORMAT

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

DXGI_USAGE

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

DXGI_SAMPLE_DESC

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

DXGI_SCALING

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

DXGI_SWAP_EFFECT

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

DXGI_ALPHA_MODE

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

DXGI_SWAP_CHAIN_FLAG

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

さて、書き込み幅はともかく他が良く分かりませんね？

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

二横幅 × 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 といふのは何かといふと

「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 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ログが起きたか分からづらいので、1つ1つ演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

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

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

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

DXGI_USAGE_READ_ONLY サーフェスまたはリソースをレン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、さつき検証したらどっちでも動きました。これもう分かんねえな…)

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

ステレオ

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

必要があります

という事らしいです。でもステレオ言うてもこれ音声の事ちゃうしなあ…。とりあえず良く分からぬるので、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。

急速コマンドキューを作りましょう。

コマンドキュー

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

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

問題は第一引数です。COMMAND_QUEUE_DESC と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よね…。

```
D3D12_COMMAND_QUEUE_DESC cmdQDesc = {};  
cmdQDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;  
cmdQDesc.NodeMask = 0;  
cmdQDesc.Priority = D3D12_COMMAND_QUEUE_PRIORITY_NORMAL;  
cmdQDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;  
result = _dev->CreateCommandQueue(&cmdQDesc, IID_PPV_ARGS(&_cmdQueue));
```

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/ディッファの数なので 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。

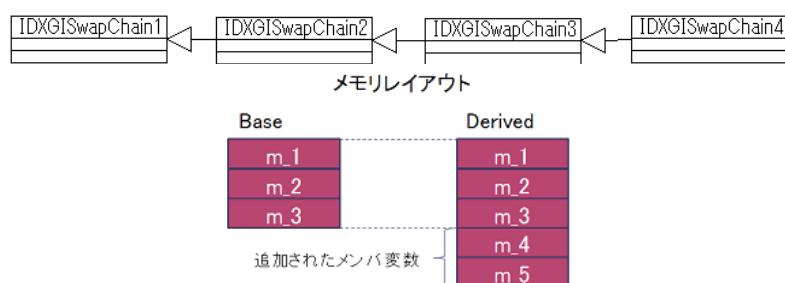
```
Size wsize = appH.GetWindowSize();
```

```

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」の基本単位のことだ

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。

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

によると、レンダーターゲットビューもそのお仲間になってるわけだ。

ビューとデスクリプター

実は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。



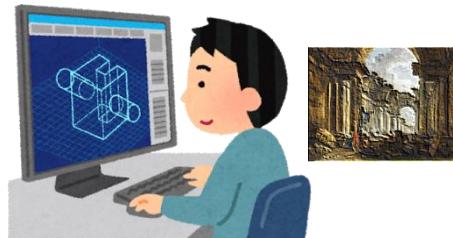
元のデータと、実際に塗る塗り方ね。後から出てきますが、深度職人てのもいて、同じ情報から深度値を書き込んでいく職人もいますので、データと、色々な見方があるわけで、それをまとめてビューやと呼んでいるわけです。

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つて、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。

デスクリプタヒープとデスクリプタテーブル

軽くデスクリプタ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。

一つのことです。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つて、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知れないが、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] REFIID riid,  
    [out] void **ppvHeap  
) ;
```

第二、第三引数はいつものパターンですね。

問題は第一引数ですが、

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

を見ながらやっていきましょう。

今回はレンダーターゲットに使用するので、Typeは

D3D12_DESCRIPTOR_HEAP_TYPE_RTV

ですね。ちなみにRTVは“RenderTargetView”的略です。

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

D3D12_DESCRIPTOR_HEAP_FLAG_NONE

次にNumDescriptorsですが、こいつはヘルプを見るだけじゃ分からぬ。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にしておきました。

最後に NodeMaskですが、こいつはゼロでいいです。これは説明に
For single-adapter operation, set this to zero. If there are multiple adapter
nodes, set a bit to identify the node (one of the device's physical adapters) to
which the descriptor heap applies. Each bit in the mask corresponds to a single
node. Only one bit must be set. See [Multi-Adapter](#).

って書いてるからです。

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

格納先を確保したので、次に実際にこゝにビューの定義を突っ込みましょう。
その前にデスクリプタ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/dn99186\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn99186(v=vs.85).aspx)

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

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

で終わりです。殺伐とした DirectX12 の中でこの関数は久々に心がほっこりするね。そしてレ
ンダーターゲットビューを作る関数は当然のように CreateRenderTargetView ですから
<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12device-createrendertargetview>

```
void CreateRenderTargetView(  
    ID3D12Resource                      *pResource, //ピクセルを書き込む本体  
    const D3D12_RENDER_TARGET_VIEW_DESC *pDesc, //レンダーターゲットビューの仕様
```

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

ご覧のように…3つとも定義が大変そう!!さてどうしたものが。

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

D3D12_CPU_DESCRIPTOR_HANDLE

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

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

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JC" << 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るのでコマンドリスト数は1でいいです。

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

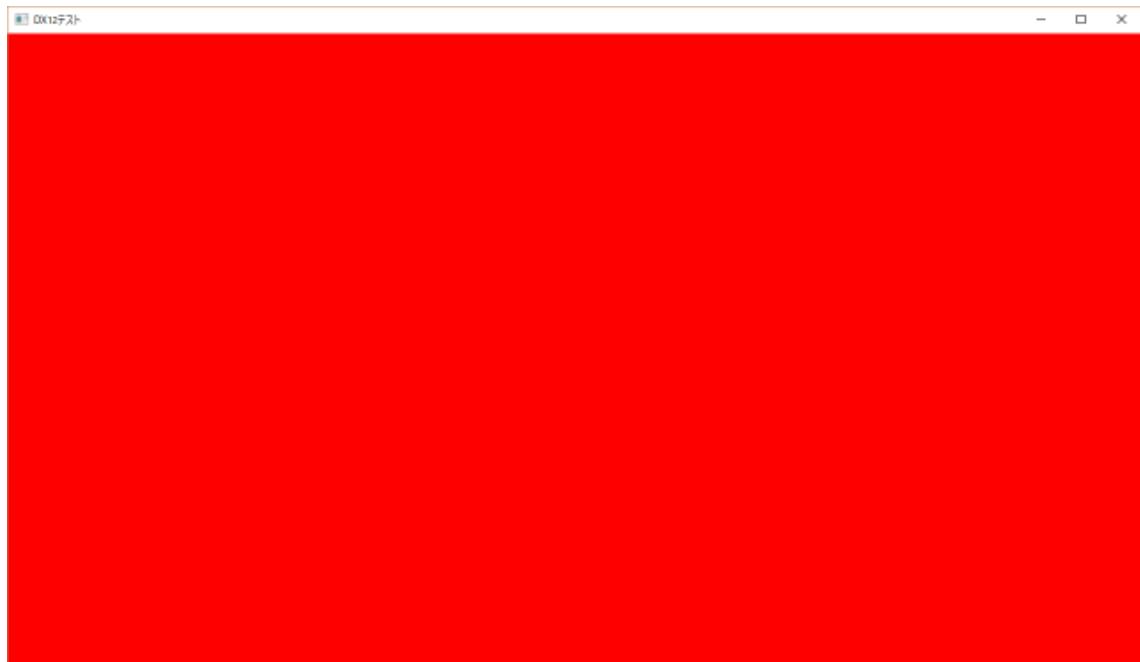
あとは Present 関数を呼べばいい

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。ていうかスワップします。

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

ひとまず 0,0 でいいですのでもってみてください。運が良ければ



画面の色が変わります。運が良ければ。運が悪ければ変わりません。

実は色々間違ってるんです

実は色々と間違っています。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
- コマンドキュー実行待ちをしていない

の 2 点です。

本来ダブルバッファリングであるため、常に裏画面に書いていきますが、それはフリップするたびに変更されるはず。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 つのレンダーラン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つていたのですが、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.0.007.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けない不完全なままデータを読み取ろうとすればそれはもうね、蛹を羽化前に開けちゃったり、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.o.oo7.jp/function/dxfunc_other.html#R21N2

などでチェックすることができます。

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

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

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

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

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

で、結局 DirectX12ではどうなの？

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

つまり今回の場合であれば画面クリアの実行が完了する前にスクリーンフリップが先に実行されてしまい、まあおかしなことになってしまふわけです。なんとかというと、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。



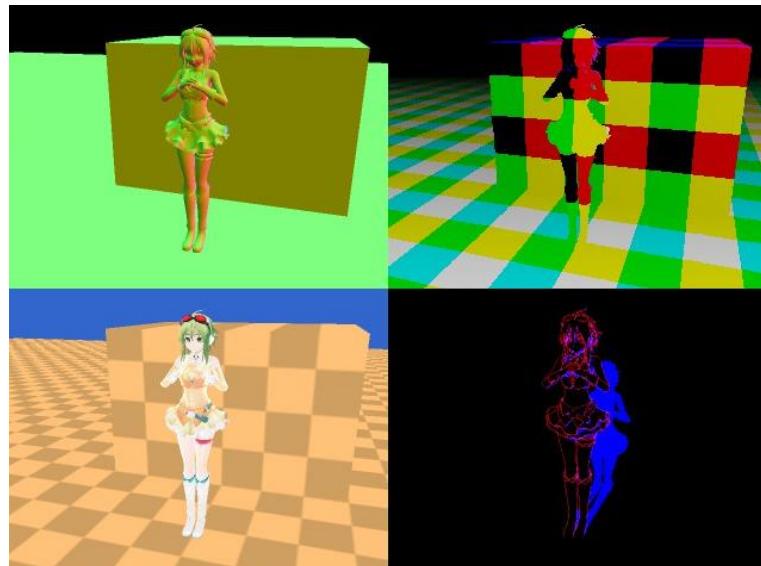
これには理由があるのだ。

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方々で使われるようになってきたのが原因じゃないかなと思う。

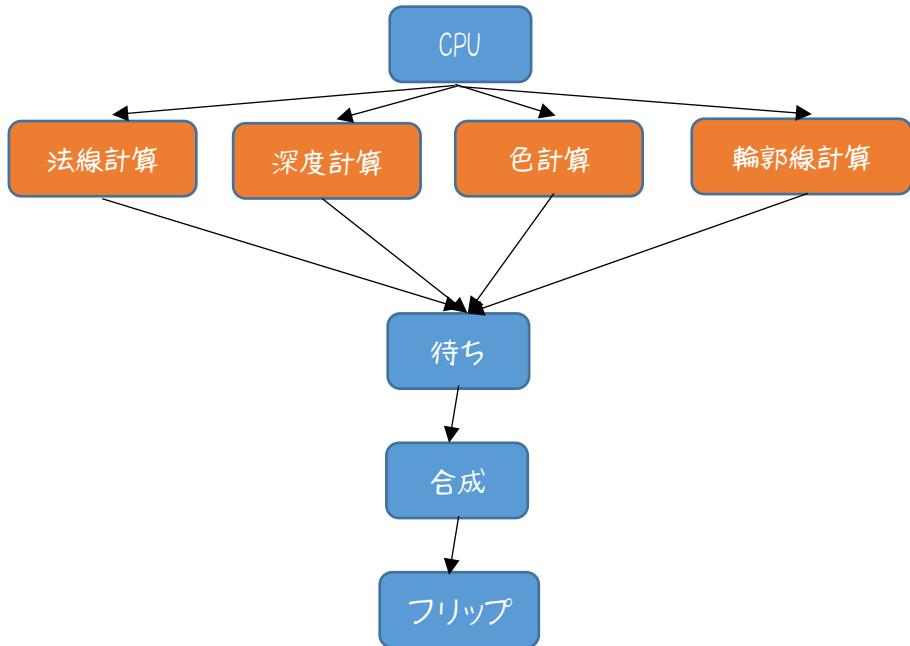
意味が分からぬだろうから簡単に言うと。

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。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。
ここで待つ方法としては2つあります。ポーリングとイベント通知です。

ポーリング

一番手っ取り早くて分かりやすい方法は「ポーリング」

Signal の後に

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

ただねえ…これやつちやうとぶつちやけビジー状態になりっぱなしになるので、どうかとは思うけど、ゲームだから CPU 占有しちゃってもいいのか。
…まあ、簡単でしょ？

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

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

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

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

まあ、それはさておきイベント通知による実装も紹介しておきます。

イベント通知

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

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

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

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

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

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

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

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

違いは流れの違いだけなので、設計次第でどちらを選ぶか決めましょう。

イベント通知を使用する場合には、こう書き換えられます。

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

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

デバッグレイヤーを有効にする

DirectX12 にはデバッグをしやすくするためにデバッグレイヤーという機能があります!! デフォルトではこの一つが無効になっているため、オンにします。

何かのオブジェクトを作るとかではなく、スイッチをオンにするだけなので、

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

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

そうすると出るわ出るわ…エラーが、ああ、あと DXGI 関連のエラーが出なかつたりするので、DXGIFactory を作るところもこう書き換えます。

```
CreateDXGIFactory2(DXGI_CREATE_FACTORY_DEBUG, IID_PPV_ARGS(&dxgiFactory));  
これで DXGI 関連のデバッグ情報も出力されます。
```

多分確定で出てくるのは リソースのドリアを行わず、レンダーターゲット書き込みとフリップを行っているところからエラーが発生していると思います。

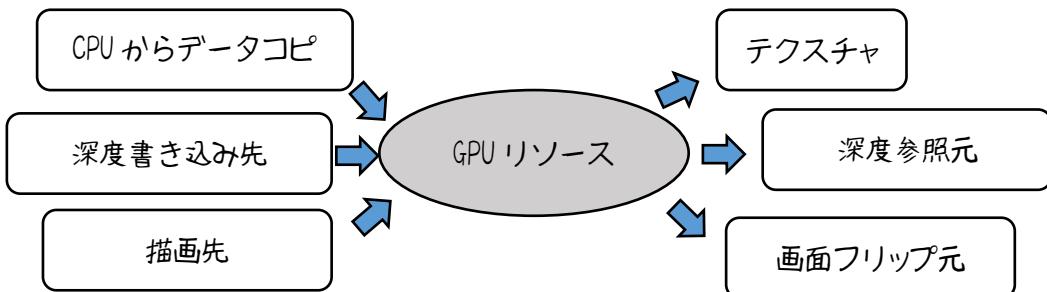
今はうまくいってるように見えますが、今後問題を引き起こすという事です。

ということで、バリアの話をやっていきます。

リソースバリア

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

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



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

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

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

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

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

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

は…非常に設定が面倒です。

ともかくレンダーターゲットとして使用する部分…

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

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

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

この第二引数の設定が面倒ですね。なぜならば

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

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

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

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

つまり、

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

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

例えば今回のバックバッファに関しては、作成の時点では(というかスワップチェーンに作られてるので)もともと PRESENT(フリップに使用する)状態として作られているので、レンダーターゲットとして使用するにはステートを RENDER_TARGET にしてあげる必要があります。

なので…

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

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

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

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

とはいっても画面の色を変えるだけじゃつまらないですね。ちゃっちゃとポリゴンを表示しちゃいましょう。

ポリゴンを表示しよう

さて、ポリゴンを表示していくわけですが、ここからは本格的に「グラフィクスピープライン」もしくは「レンダリングペイプライン」を意識する必要があります。まず頂点を定義するところから…

頂点を作ろう

今回は三角形を作っていくわけですが、頂点はいくつ必要でしょうか？ そう、3つですね。では1頂点あたり必要な情報はどのくらいでしょうか…まずは座標の情報が最低限必要ですね。

一応最終的には3D座標にするためx,y,zの3つの情報をfloatで持たせることを考えましょう。

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

というのを作つてもいいんですが、ここはDirectXMathというライブラリの恩恵を受けましょう。ちょっと数学関数や構造体に関しては自前実装は手間がかかり過ぎますので…。

という事で、まずは必要な数学ライブラリをインクルードします。

```
#include<DirectXMath.h>  
特にリンクは必要ないのでこのまま使いましょう。
```

先ほどのVector3みたいのは既に用意されていてXMFLOAT3という名前の構造体で定義されています。なお、XMVECTORの方が適切そうですが少し使い方が難しいため、今はXMFLOAT3を使っておきます。

なお、数学ライブラリの名前空間がDirectXなので、using namespace DirectX;を使いましょう。

```
using namespace DirectX;  
(中略)  
XMFLOAT3 vertices(3); //3頂点
```

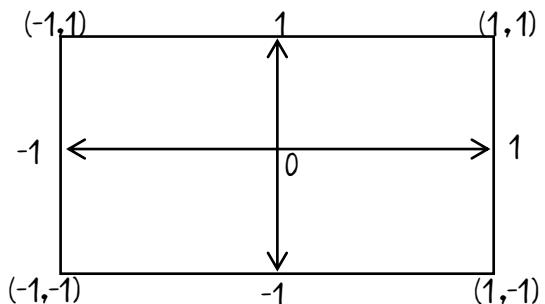
としてもいいですし、もしコーディングルール等の関係でusing namespaceが使えない場合は

```
DirectX::XMFLOAT3 vertices(3); //3頂点
```

でも構いません。で、それぞれの頂点を定義します。

```
XMFLOAT3 vertices() = {  
    {-1.0f,-1.0f,0.0f} , //左下  
    {-1.0f,1.0f,0.0f} , //左上  
    {1.0f,-1.0f,0.0f} , //右下  
};
```

ちなみに何も座標変換しない状況だとこの-1だの1だの0ってのがどこに対応しているのかというと



こんな感じです。

つまりウインドウの左上が(-1,1)で右下が(1,-1)です。2D 座標系といふか UV 座標系にそっくりですが、上がプラスで下がマイナスです。で、画面のアスペクト比にかかわらず縦も横もどちらも-1~1 と、つまり幅が2であるため、そのままのイメージで出力すると必ず横長になります。

後々これは考慮しなければなりませんが今は多少横広になつても構いません。

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ぶん投げます。

ID3D12Resource を使用します。

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

ID3D12Resource* _vertexBuffer=nullptr;

とでも宣言しておいてください。

これを作るには CreateCommittedResource を使用します。

[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)

HRESULT CreateCommittedResource(

const D3D12_HEAP_PROPERTIES *pHeapProperties, // ヒープ設定構造体のアドレス

D3D12_HEAP_FLAGS HeapFlags, // 特に指定がないので D3D12_HEAP_FLAG_NONE で可

```

const D3D12_RESOURCE_DESC *pDesc, //リソース設定構造体のアドレス
D3D12_RESOURCE_STATES InitialResourceState, //GPU 側からは読み取り専用なので GENERIC_READ
const D3D12_CLEAR_VALUE *pOptimizedClearValue, //使わないもので nullptr でいい
REFIID riidResource, //一つもの
void **ppvResource //一つもの
);

```

こんな仕様になっているため

特に曲者っぽいのが D3D12_HEAP_PROPERTIES と D3D12_RESOURCE_DESC ですね。というかヒープの設定とリソースの設定って、どちらも意味的にはメモリを示してるっぽくて良く分からぬいね。

じゃあまず、HEAP_PROPERTIES から見ていきましょうか。

```

struct D3D12_HEAP_PROPERTIES {
    D3D12_HEAP_TYPE Type; //ヒープ種別(Mapする必要があるなら UPLOAD)
    D3D12_CPU_PAGE_PROPERTY CPUPageProperty; //ページング設定(PAGE_PROPERTY_UNKNOWN で可)
    D3D12_MEMORY_POOL MemoryPoolPreference; //メモリプールどこ?(POOL_UNKNOWN で可)
    UINT CreationNodeMask; //単一アダプタなら 0 で OK
    UINT VisibleNodeMask; //単一アダプタなら 0 で OK
};

```

さて、頂点ヒープ設定についてですが、かなりややこしいです。正直なところ、正確に説明できるか分かりません。ええ、わからないのです。というわけでここは「僕はこう解釈しました」だと思ってください。

ヒープ種別

まず、ヒープの種別ですが、

```

enum D3D12_HEAP_TYPE {
    D3D12_HEAP_TYPE_DEFAULT, //CPU からアクセスできない(map できない)
    D3D12_HEAP_TYPE_UPLOAD, //CPU からアクセスできる(map できる)
    D3D12_HEAP_TYPE_READBACK, //CPU から読み取れる
    D3D12_HEAP_TYPE_CUSTOM //カスタムヒープ(カスタムなんでもこの後がややこしい)
};

```

このような種類があります。ポリゴンを表示するために使用されるヒープは DEFAULT か

UPLOAD が殆どという事になると思います。なので、この二つについて説明します。MSDN の説明を全面的に信用して書きます。

D3D12_HEAP_TYPE_DEFAULT

DEFAULT は CPU からアクセスできません。この指定でヒープを作ったうえで後述する Map 関数でアクセスしようとすると失敗します。ただしバンド幅が最大に広く(つまりアクセスが早い)ため、GPU 側のみでやりとりするためのヒープとして向いています。

D3D12_HEAP_TYPE_UPLOAD

次に UPLOAD ですが、名前からして CPU のデータを GPU にアップロードするために使用するものようです。このため UPLOAD で作ったヒープは CPU からアクセスできます。

つまり Map 関数で中身を書き換えることが可能ですが、便利ですが CPU、GPU 双方からのアクセスが DEFAULT に比べると遅いため、CPU から一回限りの書き込み、GPU から一回限りの読み込みをするような場合に推奨されるようです。

一応上2つのどちらかを頂点に対しては使う事になりますが、他二つも軽く紹介しておきます。

D3D12_HEAP_TYPE_READBACK

READBACK というのは読み戻し専用のヒープです。つまり CPU 側から見れるという事。CPU から可読のためか、バンド幅は広くないようです(比較的遅い)。GPU で加工・計算したデータを CPU 側で活用するための物でしょう。

D3D12_HEAP_TYPE_CUSTOM

最後に CUSTOM ですが、これがちょっとややこしいです。実はここを CUSTOM 以外(UPLOAD や DEFAULT)で指定している場合、ページング設定やメモリプール設定は後述する UNKNOWN を指定すればよく、非常に楽なのですが、CUSTOM だとこのページング設定やメモリプールをきちんと設定しなければならなくなります。

というわけなので、頂点/バッファの時は UPLOAD、UNKNOWN、UNKNOWN でいいわけです。

ページ設定

前述の理由により、頂点/バッファ作成では D3D12_CPU_PAGE_PROPERTY_UNKNOWN でいいのですが、一応他に何があるのか見てみましょう。基本的にここからの話は CUSTOM の時にしか関わってきませんので、今の所は読み飛ばしてもらって結構です。

```
enum D3D12_CPU_PAGE_PROPERTY {
    D3D12_CPU_PAGE_PROPERTY_UNKNOWN, // 考えなくていい
    D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE, // CPU からアクセス不可
    D3D12_CPU_PAGE_PROPERTY_WRITE_COMBINE, // ライトコンバイン
    D3D12_CPU_PAGE_PROPERTY_WRITE_BACK // ライトバック
};
```

何の話やーって感じですね。

CPU からアクセス不可はまだ分かるにせよ WRITE_COMBINE と WRITE_BACK って何の違いがあるんでしょうか…?

D3D12_CPU_PAGE_PROPERTY_UNKNOWN

これはヒープ種別が CUSTOM 以外の時に使う設定です。特に設定しなくてもそれぞれに合わせた設定をしてくれます。逆に言うと CUSTOM にすると UNKNOWN は許されず、このあたりの設定を自分でいじる事になるわけですね。

D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE

読んで字の如く、CPU からのアクセス不可です。要はこのヒープは GPU 内での計算に限定して使用してねということですね。ヒープ種別 DEFAULT の時と同じ状態になるのではないかと思われます。

D3D12_CPU_PAGE_PROPERTY_WRITE_BACK

WRITE_BACK に関してですが、CPU のキャッシュにライトバック方式と言うのがあり、それをイメージするといいのかなと思います。

どういう方式なのかと言うと、通常 CPU が演算を行い、その結果をメモリに書き戻すわけですが、この時にメモリとキャッシュに同時に書き込むのをライトスルー方式と言い、これに対して CPU の演算結果をまずはキャッシュに書き込み、適切なタイミングでメモリに書き込まれる方法を「ライトバック」と言います。

恐らく CPU→GPU へ転送する場合に CPU のメモリをキャッシュとして使用もしくは文字通り CPU 側のキャッシュメモリをキャッシュとして、時間の空きができ次第、順次 GPU 側のメモリに転送する方式だと考えられます。すみませんが、ここは自信を持ってこうだ!と言えないのが正直なところです。

D3D12_CPU_PAGE_PROPERTY_WRITE_COMBINE

では WRITE_COMBINE は何かといふと、COMBINE(結合)と言う意味から推測できるように、送るべきデータをある程度の大きさのデータにまとめて転送する方式を表しています。このまとめて転送するのを「バーストモード」と言ったりするようです。注意点としてはこのまとめて送る際には順序は考慮されないので、使いどころは気をつけた方が良さそうです。

WRITE_BACK と WRITE_COMBINE は CPU からメモリにデータを書き込んで GPU 側に転送と言う流れは共通しているように思えます。転送の方式に差があるという所ですね。CPU アーキテクチャの話だと転送の際に少しずつキャッシュに乗つけて順次転送するのが Write-Back で、キャッシュに乗つけず一気に転送するのが Write-Combine と言ったところかなと思います。一応メモリプールとの組み合わせは決まっているようなので、後の項で表にします。

メモリプール設定

次にメモリプール設定ですが、これは以下のようになっています。

```
enum D3D12_MEMORY_POOL {
    D3D12_MEMORY_POOL_UNKNOWN, // CUSTOM 以外の時はこれでいい
    D3D12_MEMORY_POOL_L0, // システムメモリ(アダプタが UMA のとき…GPU バンド幅狭)
    D3D12_MEMORY_POOL_L1 // ビデオメモリ(アダプタが NUMA のとき…GPU バンド幅広)
};
```

まず、CUSTOM 以外の時は UNKNOWN で構わないといふのは、CPU ページ設定と同様です。次に L0 と L1 についてですが、これはよくキャッシュに L1~L2 とかあります、その事だと思って良いかなと思います。

L0について

L0 はシステムメモリを表しています。アダプタ(グラボ)の状況によって変わってくるようです。MSDN の解説を見る限り、オンボード(UMA)の場合は、この L0 しか選択肢がないようです。もし、nVidia や AMD などのディスクリートグラボを参照している場合に L0 を選ぶと CPU バンド幅が広く、GPU バンド幅が狭くなるようです。

L1について

L1 はビデオメモリを表しています。これはオンボード(UMA)の場合には使用できないパラメータです。nVidia や AMD のディスクリートグラボを参照している場合にのみ使用できます。L1 設定にすると GPU 用のバンド幅が広くなりますが、CPU 側からのアクセスができなくなりま

す。

組み合わせ

ここまで読んでもらったことをまとめて表にしておきます。これ以外の組み合わせでは私の環境では INVALIDARG が返ってきました。

ヒープ種別	ページ設定	メモリプール	RESOURCE_STATE	Map 可?
DEFAULT	UNKNOWN	UNKNOWN	GENERIC_READ	不可
UPLOAD	UNKNOWN	UNKNOWN	GENERIC_READ	可能
READBACK	UNKNOWN	UNKNOWN	COPY_DEST	可能
CUSTOM	NOT_AVAILABLE	L0/L1	GENERIC_READ	不可
	WRITE_BACK	L0のみ	GENERIC_READ	可能
	WRITE_COMBINE	L0/L1(?)	GENERIC_READ	可能

※とはいえ、nVidia のグラボでしか検証していないため、AMD 等の時の挙動は各自で検証してほしいと思います(あと、全ての組み合わせ検証を完璧にやったわけでもないため、使う時は用途に合わせて各自検証してください)。あと COMBINE&L1 の組み合わせが不安定です。

あと、Map 可?という項目がありますが、Map については次項で説明しますが、簡単に言うとこのバッファを通常の CPU 側メモリアクセスのように扱えるようにするための処理を Map といい、それが可能かどうかという事です。

さて、長々と設定について話してきましたが、ひとまず頂点データは CPU 側から設定するものなので、Map で設定するとして UPLOAD で作ろうと思います。モーフ(フレンドシェイプ)などを作らない限りは頂点データは毎フレーム 1 回くらいしか参照しないでしょうし、UPLOAD で問題ないかと思います。ほかの指定に比べるとパフォーマンスが悪いため頻繁にアクセスがある場合には別設定を検討する必要があるとは思います。

後々パフォーマンスを考慮したり、用途に合わせて、これらのバッファの作り方を選択していくことになりますが、まずは単純に頂点を GPU に送りたいので、パフォーマンスの事は置いといて UPLOAD を使用していきます。

リソース設定構造体

うえへ、あんなにやったのにまだあるの~?って思ってるでしょうけど、もうひと踏ん張りです。頑張りましょう。

リソースの設定には D3D12_RESOURCE_DESC という構造体を設定します。

```

struct D3D12_RESOURCE_DESC {
    D3D12_RESOURCE_DIMENSION Dimension; // バッファに使うので BUFFER
    UINT64 Width; // 幅で全部賄うので sizeof(全頂点)
    UINT Height; // 幅で表現してるので 1
    UINT16 DepthOrArraySize; // 1 でいい
    UINT16 MipLevels; // 1 でいい
    DXGI_FORMAT Format; // 画像じゃないので UNKNOWN で
    DXGI_SAMPLE_DESC SampleDesc; // Count=1 で
    D3D12_TEXTURE_LAYOUT Layout; // D3D12_TEXTURE_LAYOUT_ROW_MAJOR で
    D3D12_RESOURCE_FLAGS Flags; // NONE で OK
};


```

Width とか Height はテクスチャの時だと画像の幅と高さを表しますが、今回はテクスチャではありません。単なる頂点情報の集合体です。Format なんてあるわけないだろ!! というわけでコメントに書いてあるような指定になります。

なお、SampleDesc に関してはアンチエイリアシングをするときのパラメータとして有效なもので、アンチエイリアシングを自動でやらないのだったら、ここは Count=1 で構いませんし、そもそも頂点座標の集合体なので、アンチエイリアシングも必要なし。本当だったら 0 にしたいところですが、それだとデータがない事になってしまいますので、1 を渡します。

もっとも悩ましいのは TEXTURE_LAYOUT なのですが、これ LAYOUT 列挙型を見ると UNKNOWN が適切なようにも思えますが、今回の場合 UNKNOWN 指定をすると、自動で最適なレイアウトを設定しようとするのですが、テクスチャのレイアウトなので不適切です。というわけで、メモリが最初から終わりまでそのまま連続している事を示す D3D12_TEXTURE_LAYOUT_ROW_MAJOR を指定します。

頂点/バッファ生成

さて、ここまで話した内容で、CreateCommittedResource でリソースを作つてみましょう。

```

D3D12_HEAP_PROPERTIES heapprop = {};
heapprop.Type = D3D12_HEAP_TYPE_UPLOAD;
heapprop.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;
heapprop.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;

D3D12_RESOURCE_DESC resdesc = {};
resdesc.Dimension = D3D12_RESOURCE_DIMENSION_BUFFER;

```

```

resdesc.Width = sizeof(vertices); //頂点情報が入るだけのサイズ
resdesc.Height = 1;
resdesc.DepthOrArraySize = 1;
resdesc.MipLevels = 1;
resdesc.Format = DXGI_FORMAT_UNKNOWN;
resdesc.SampleDesc.Count = 1;
resdesc.Flags = D3D12_RESOURCE_FLAG_NONE;
resdesc.Layout = D3D12_TEXTURE_LAYOUT_ROW_MAJOR;

ID3D12Resource* vertBuff = nullptr;
result = _dev->CreateCommittedResource(
    &heapprop,
    D3D12_HEAP_FLAG_NONE,
    &resdesc,
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&vertBuff));

```

最後の CreateCommittedResource の戻り値に S_OK が返るかどうかをご確認ください。もし S_OK 以外が返っている場合は、どこかの設定間違いだと思いますので、見直してください。

ともかくこれで頂点バッファができました。あ、リザルトは確認しておいてくださいね。でも、すまんが 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。

頂点データ転送

DirectX12において転送は Map でやります。

Map って何か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…もしかしなくとも nullptr でOKです)

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* や 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)

```
XMFLOAT3* vertMap=nullptr;  
result=vertBuff->Map(0, nullptr, (void**)&vertMap);  
std::copy(std::begin(vertices), std::end(vertices), vertMap);
```

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。だけどまだ終わりじゃないです。まだまだデータの塊に過ぎないのでこれまた「ビュー」が必要です。

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 でい

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れど、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);
```

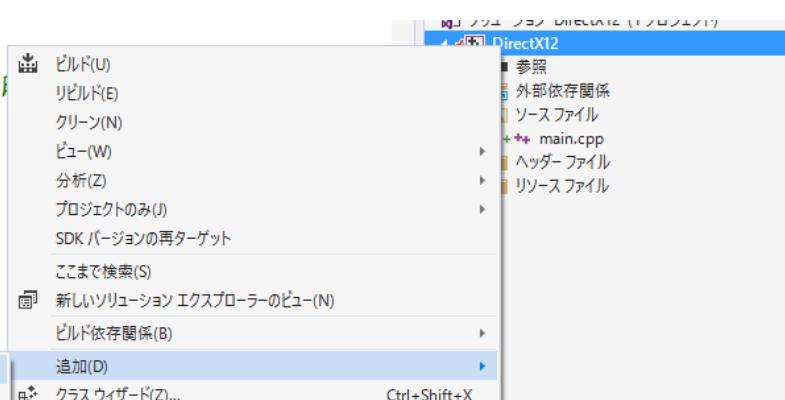
で、このビューを最終的にはコマンドリストにて

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

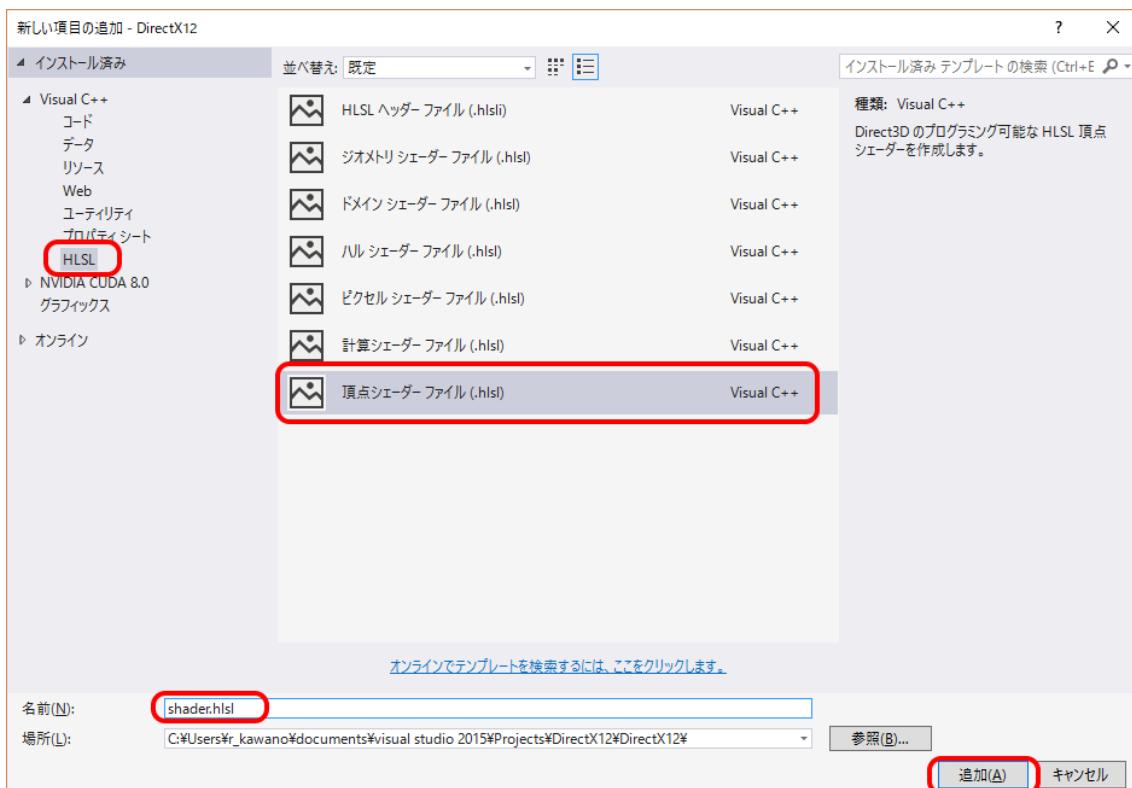
てな投げ方をするんですが、それはもうちょっと後でやります。

そんな事よりシェーダ書こうぜ

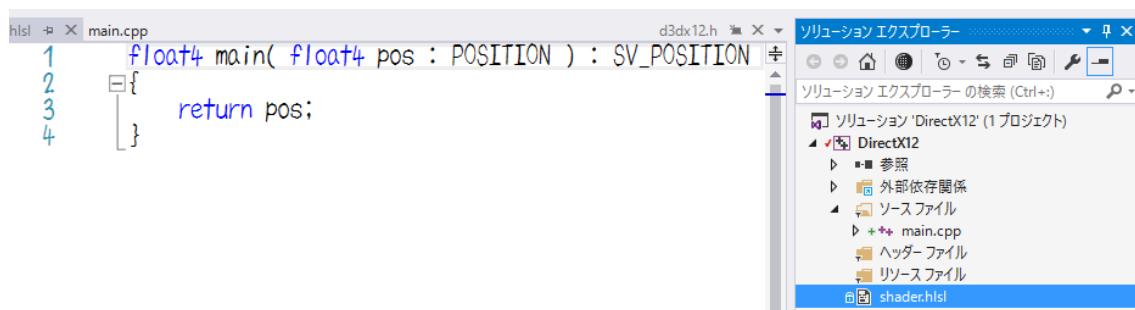
```
AD), //CPUからGPUへ転送する  
}); // サイズ  
,  
,  
,  
uff);
```



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(実際は別にどのシェーダでも構いません。どーセ後で「頂点シェーダファイル」ではなくしますので)。シェーダフ

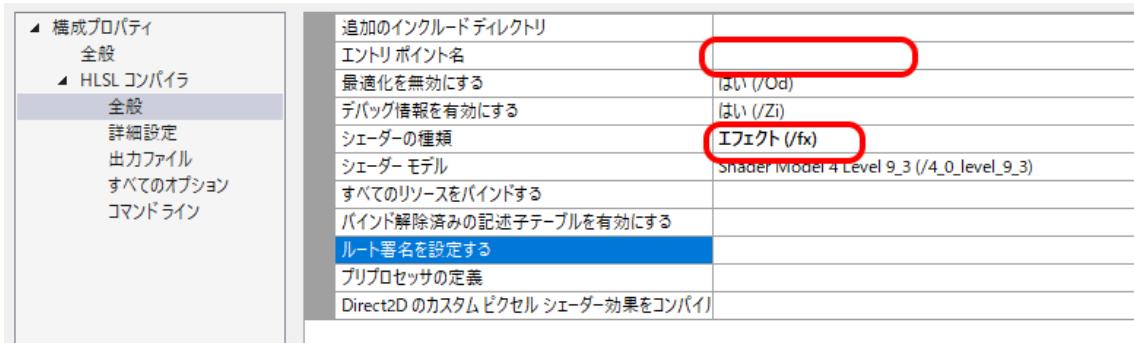


イルの名前は shader.hlsl にしといてください。元からあるソースコードは全消しして

```
float4 BasicVS(float4 pos:POSITION) :SV_POSITION{
    return pos;
}
float4 BasicPS(float4 pos:SV_POSITION):SV_TARGET{
    return float4(1,1,1,1);
}
```

とても書いておいてください。意味的なところは後で解説します。

その上で shader.hsls で右クリック>プロパティ>HLSL コンパイラ→全般を選択し以下の設定を出します。



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のままでいいのでピクセルシェーダを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、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 hsls 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.hsls"), nullptr, nullptr, "BasicVS", "vs_5_0", D3DCOMPILE_DEBUG |  
D3DCOMPILE_SKIP_OPTIMIZATION, 0, &vertexShader, nullptr);  
result = D3DCompileFromFile(_T("shader.hsls"), 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ひます。

ちなみに機械翻訳した結果…

トピック

説明

ルート署名の概要 ルート署名はアプリによって構成され、シェーダーで必要なリソースにコマンド一覧をリンクします。グラフィックス コマンド リストには、グラフィックスと計算ルート署名の両方があります。計算コマンド リストには、計算ルート署名が 1 つだけあります。これらのルート署名は、互いに独立しています。

ルート署名の使用 ルート署名は記述子テーブル（そのレイアウトを含む）、ルート定数、ルート記述子の任意に整理されたコレクションの定義です。各エントリのコストには上限名がありため、アプリケーションでは、ルート署名に含めるエントリの各種類の数との間でバランスを取ることができます。

ルート署名の作成 ルート署名は入れ子になった構造を含む複雑なデータ構造です。これらは、下のデータ構造の定義を使用してプログラムによって定義できます（メンバーの初期化に役立つメソッドを含む）。または、上位レベル シェーダー言語 (HLSL) で作成できます。これには、コンパイラーによって、レイアウトとシェーダーとの互換性が早期に検証されるという利点があります。

ルート署名の制限

ルート署名は、主要な不動産であり、厳密な制限と考慮するコストがあります。

ルート署名で定数を直接使う アプリケーションでは、それぞれ 32 ビット値のセットとして、ルート署名でルート定数を定義できます。これらは HighLevel Shading Language (HLSL) で定数バッファーとして扱われます。履歴の理由上、定数バッファーは 4×32 ビット値のセットとして表示されていることに注意してください。

ルート署名で記述子を直接使う 記述子ヒープを実行しないで済むように、アプリケーションでは記述子をルート署名に直接置くことができます。これらの記述子はルート署名で多くの領域を占めるため（ルートの署名の制限のセクションを参照）、アプリケーションで使用するはこれらを慎重に使用する必要があります。

ルート署名の例 次のセクションでは、空の状態から完全なフルまで、さまざまな複雑さのルートの署名を示します。

トピック	説明
<u>HLSL でのルート署名の指定</u>	C++ コードでの指定の代替方法として、HLSL Shader Model 5.1 でルート署名を指定できます。

ルート署名バージョン 1.1 の目的は、記述子ヒープ内の記述子が変わらない場合またはデータ記述子のポイント先が変わらない場合、ドライバーを示すよう名バージョンにアプリケーションを有効にすることです。これにより、ドライバーのオプションを最適化して、ドライバーがポイントする記述子またはメモリが一定の時間静的であることを認識できる場合があります。

Google 大先生翻訳がかなりの勢いでドグるほど難しいようです。



割と本気で殺しに来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。」←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。説明は難しいかも…)

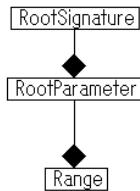
非常に作りが…本当に、本当にややこしい。

結局説明を入れても

<https://sites.google.com/site/monshonosuana/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でこう書きました。実はルートパラメータという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://shobomaru.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。

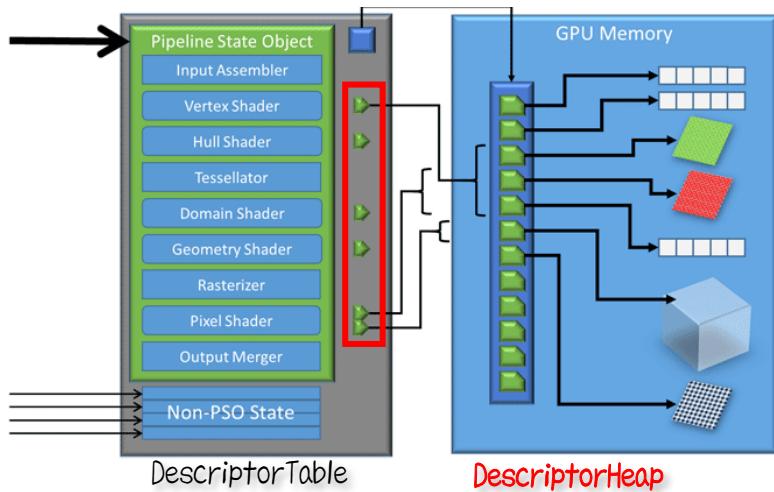
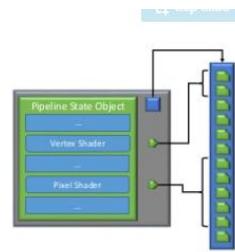


図2

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

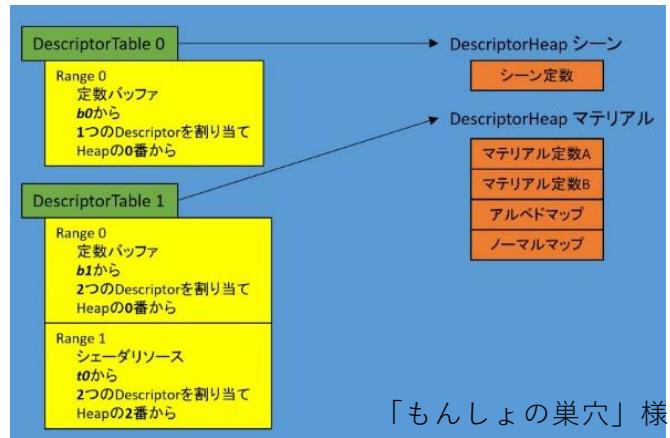


で、ディスクリプタテーブルはレンジと言うものを持ってて、これによってヒープの特定の場所にアクセスするんだけど、ヒープの場所の~(どつかアドレス)ではなくシェーダレジスタ番号の指定なんだよね。

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。

もんし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、意味が分からぬいしくみ…それが DirectX12

『DirectX12 滔へようこそ』

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。興味があつたら「不思議なプロビー」とか映画「the BLOB」を見ると分かりやすいけかもしれません。

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（頂点バッファバインディングのセットを定義する入力レイアウトが必要）を使用するようにオプトインしています。このフラグを省略すると、一部のハードウェアに1つのルート引数スペースが保存される可能性があります。入力アセンブラが不要な場合はこのフラグを省略しますが、最適化は軽微です。』

ということで、今回は『入力アセンブラ(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うが、これ以上ここに時間をかけたくないのと先に進
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、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てます。

「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)だけの COLOR(色)だけのが出でてきます。

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てこないので。

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 なんて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

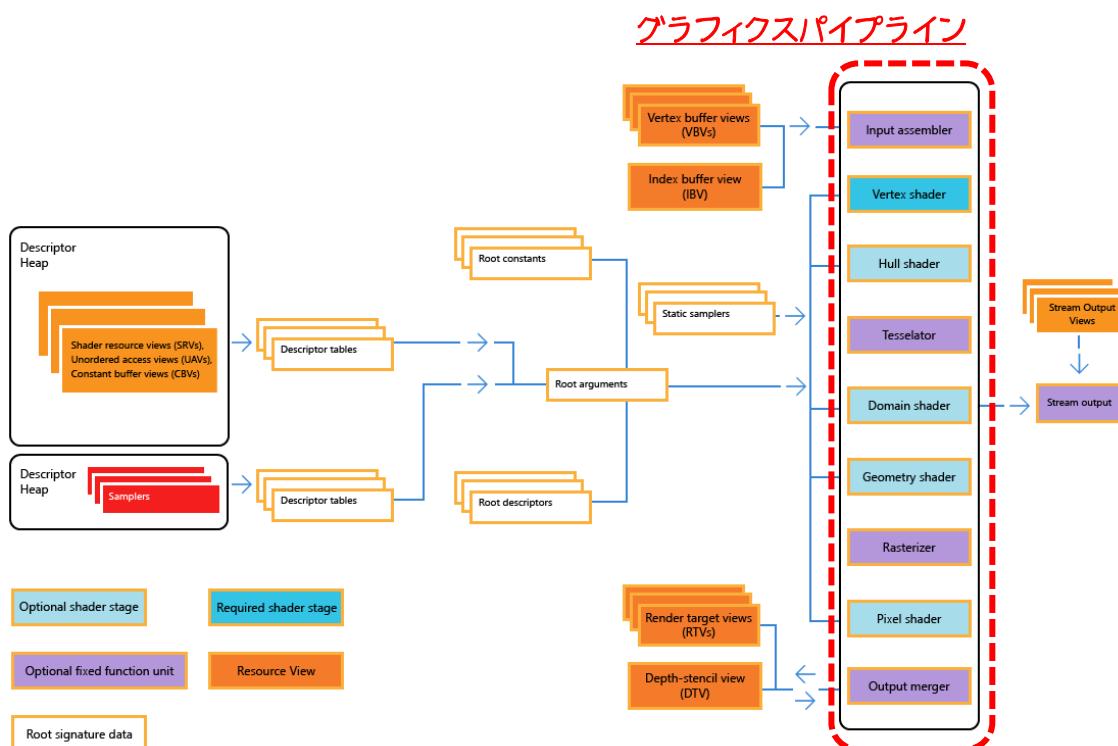
```
gpsDesc.InputLayout.pInputElementDescs = inputLayoutDescs; //
gpsDesc.InputLayout.NumElements = _countof(inputLayoutDescs); //
とでもしてやれば( ^ω^)おつけ。
```

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ではバラバラになっているものくっ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す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

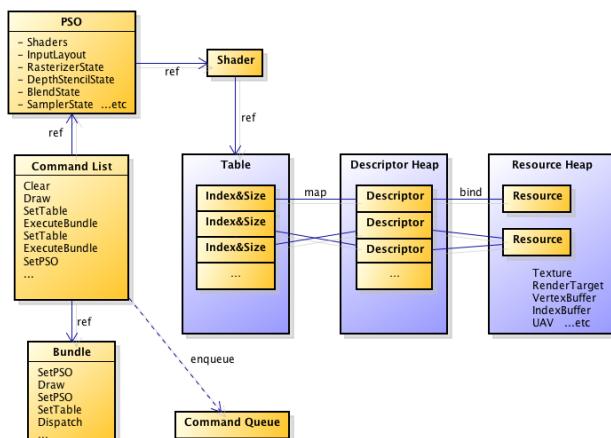
- 頂点レイアウト
- ラスタライザーステート
- ブレンドステート
- デプスステンシルステート
- トポロジータイプ
- その他色々

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



があります。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)

とはいって、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…とはいって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;//今はフルスクリーン
DepthStencilState.StencilEnable=false;//今はフルスクリーン
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
```

```
gpusDesc.NumRenderTargets = 1;//注)このターゲット数と設定するフォーマット数は  
gpusDesc.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番目のみ設定しとけばいいです(デフォルトはDXGI_FORMAT_UNKNOWN)

```
//深度ステンシル  
gpusDesc.DepthStencilState.DepthEnable = false;//あとで  
gpusDesc.DepthStencilState.StencilEnable = false;//あとで  
gpusDesc.DSVFormat;//あとで  
  
//ラスタライザ  
gpusDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);  
  
//その他  
gpusDesc.BlendState=CD3DX12_BLEND_DESC(D3D12_DEFAULT);  
gpusDesc.NodeMask = 0;  
gpusDesc.SampleDesc.Count = 1;//いる  
gpusDesc.SampleDesc.Quality = 0;//いる  
gpusDesc.SampleMask = 0xffffffff;//全部1  
//gpusDesc.Flags;//デフォルトでOK  
gpusDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;//三角形  
ID3D12PipelineState* _pipelineState=nullptr;  
result = dev->CreateGraphicsPipelineState(&gpusDesc, 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…じゃなくても可)
- 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ぬ。

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(バックバッファフレーム番号),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ひますので、

テプラ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れを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)

を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設定

```
_cmdList->RSSetViewports(1, &_viewport);  
_cmdList->RSSetScissorRects(1, &_scissorRect);
```

残り色々セツト

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)

これはレートって簡単。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!んだけど、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

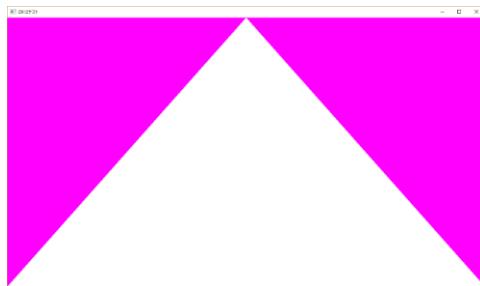
ドロー!!ポリゴン!!!

[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

いです。
うまくいけば…



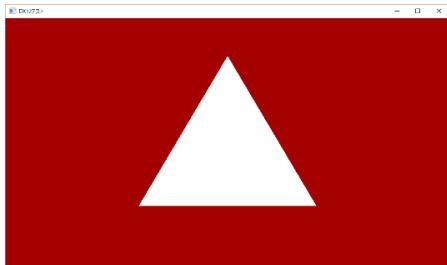
こんな感じの表示になります。右上の白い部分が今回描画している三角形です。

ここまで苦労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

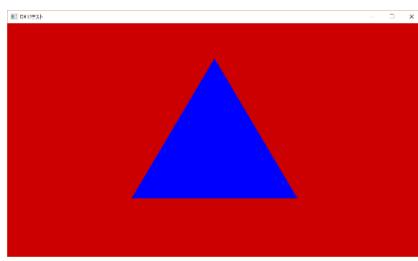
```
Vertex vertices[] = { { { 0.0f, 0.7f, 0.0f } },
{ { 0.4f, -0.5f, 0.0f } },
{ { -0.4f, -0.5f, 0.0f } } };
```

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わからないことですね。

あ、忘れてた…トポロジの設定を忘れてた…。

```
_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

すみません、どうがんばっても表示されなかつた人、本当にすみません…。

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)。



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よね。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)グラボを選ぶというのはできなくもない。

アプリがグラボを選ぶズエ…レリーズエ…

面倒なんだけど、dxgifactory を作った後に「使用されて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<IDXGIAdapter*> adapters;  
IDXGIAdapter* 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 = {};
```

```

adpt->GetDesc(&adesc);
std::wstring strDesc = adesc.Description;
if (strDesc.find(L"NVIDIA")!= std::string::npos) {//NVIDIAアダプタを強制
    adapter = adpt;
    break;
}
}

そのアダプタを使って CreateDevice する。
D3D12CreateDevice(adapter, 1, IID_PPV_ARGS(&_dev))
いいね？

```

四角形ポリゴンにしてみよう

で、ここにいるみんなは賢いのでちょっと自分で考えて4角形にしてみてください。

ヒントは頂点の数…と、プリミティブを TRIANGLESTRIP にしてみることです。

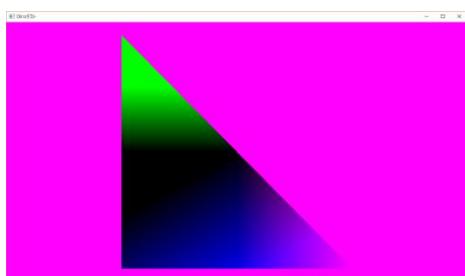
まず、4頂点にしましょう。

```

//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。

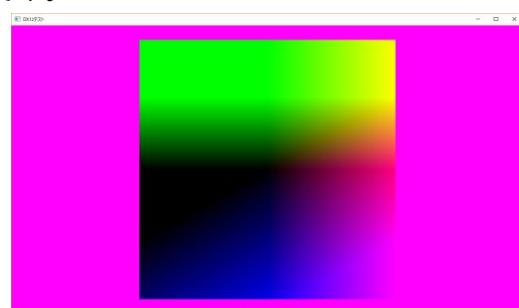
まずは、TRIANGLESTRIPにしてしまう事です。

```
_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 2 字の順序になるように並べていくことでひとつつなぎりの三角形を表現できるものです。



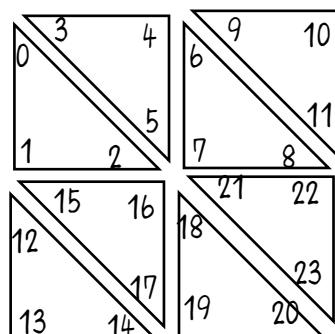
ね?簡単じゃろ?簡単だよなあ?

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。

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、もつというと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->IASetPrimitiveTopology(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…わかるじゃろ？

そんな事よりテクスチャ貼ろうぜ



ひとまず何かしら画像を用意してください。どっちかというと自分がお気に入りの画像がいいでしょう。なぜなら画像の色が違つてたり、出力がおかしい場合に変があれば気づきやすいからです。

今回正方形に張り付ける予定ですので、なるべく正方形の画像を用意してください。欲を言えば幅と高さが 2^n になっているものがいいです。

まずは貼れる準備をしましよう

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クスピー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

でな流れになってて「ペイプライン」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を追加しとして…

```
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。察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](https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12-device-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ベルついでに 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? 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!??

まあ分かる範囲で説明します。どうやら文字通りの意味でもなさそうです。

ヒープ種別

まず、ヒープの種別ですが、

```
enum D3D12_HEAP_TYPE {  
    D3D12_HEAP_TYPE_DEFAULT, //CPU からアクセスできない(map できない)  
    D3D12_HEAP_TYPE_UPLOAD, //CPU からアクセスできる(map できる)  
    D3D12_HEAP_TYPE_READBACK, //CPU から読み取れる  
    D3D12_HEAP_TYPE_CUSTOM //カスタムヒープ(カスタムなんでこの後がややこしい)  
};
```

このような種類があります。ポリゴンを表示するために使用されるヒープは DEFAULT か UPLOAD が殆どという事になると思います。なので、この二つについて説明します。MSDN の説明を全面的に信用して書きます。

D3D12_HEAP_TYPE_DEFAULT

DEFAULT は CPU からアクセスできません。この指定でヒープを作ったうえで後述する Map 関

数でアクセスしようとすると失敗します。ただしバンド幅が最大に広く(つまりアクセスが早い)ため、GPU側のみでやりとりするためのヒープとして向いています。

D3D12_HEAP_TYPE_UPLOAD

次に UPLOADですが、名前からして CPU のデータを GPU にアップロードするために使用するものようです。このため UPLOADで作ったヒープはCPUからアクセスできます。
つまり Map 関数で中身を書き換えることが可能です。便利ですが CPU、GPU 双方からのアクセスが DEFAULT に比べると遅いため、CPU から一回限りの書き込み、GPU から一回限りの読み込みをするような場合に推奨されるようです。

一応上2つのどちらかを頂点に対しては使う事になりますが、他二つも軽く紹介しておきます。

D3D12_HEAP_TYPE_READBACK

READBACK というのは読み戻し専用のヒープです。つまり CPU 側から見れるという事。CPU から可読のためか、バンド幅は広くないようです(比較的遅い)。GPU で加工・計算したデータを CPU 側で活用するための物でしょう。

D3D12_HEAP_TYPE_CUSTOM

最後に CUSTOMですが、これがちょっとややこしいです。実はここを CUSTOM 以外(UPLOAD や DEFAULT)で指定している場合、ページング設定やメモリプール設定は後述する UNKNOWN を指定すればよく、非常に楽なのですが、CUSTOMだとこのページング設定やメモリプールをきちんと設定しなければならなくなります。

といふわけなので、頂点/バッファの時は UPLOAD、UNKNOWN、UNKNOWN でいいわけです。

ページ設定

前述の理由により、頂点/バッファ作成では D3D12_CPU_PAGE_PROPERTY_UNKNOWN でいいのですが、一応他に何があるのか見てみましょう。基本的にここからの話は CUSTOM の時にしか関わってきませんので、今の所は読み飛ばしてもらって結構です。

```
enum D3D12_CPU_PAGE_PROPERTY {
```

```
D3D12_CPU_PAGE_PROPERTY_UNKNOWN,//考えなくていい  
D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE,//CPU からアクセス不可  
D3D12_CPU_PAGE_PROPERTY_WRITE_COMBINE,//ライトコンバイン  
D3D12_CPU_PAGE_PROPERTY_WRITE_BACK//ライトバック  
};
```

何の話やーって感じですね。

CPU からアクセス不可はまだ分かるにせよ WRITE_COMBINE と WRITE_BACK って何の違いがあるんでしょうか…?

D3D12_CPU_PAGE_PROPERTY_UNKNOWN

これはヒープ種別が CUSTOM 以外の時に使う設定です。特に設定しなくともそれぞれに合わせた設定をしてくれます。逆に言うと CUSTOM にすると UNKNOWN は許されず、このあたりの設定を自分でいじる事になるわけですね。

D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE

読んで字の如く、CPU からのアクセス不可です。要はこのヒープは GPU 内での計算に限定して使用してねということですね。ヒープ種別 DEFAULT の時と同じ状態になるのではないかと思われます。

D3D12_CPU_PAGE_PROPERTY_WRITE_BACK

WRITE_BACK に関してですが、CPU のキャッシュにライト/バック方式と言うのがあり、それをイメージするといいのかなと思います。

どういう方式なのかと言うと、通常 CPU が演算を行い、その結果をメモリに書き戻すわけですが、この時にメモリとキャッシュに同時に書き込むのをライトスルー方式と言い、これに対して CPU の演算結果をまずはキャッシュに書き込み、適切なタイミングでメモリに書き込まれる方法を「ライトバック」と言います。

恐らく CPU→GPU へ転送する場合に CPU のメモリをキャッシュとして使用もしくは文字通り CPU 側のキャッシュメモリをキャッシュとして、時間の空きができ次第、順次 GPU 側のメモリに転送する方式だと考えられます。すみませんが、ここは自信を持ってこうだ!と言えないのが正直なところです。

D3D12_CPU_PAGE_PROPERTY_WRITE_COMBINE

では WRITE_COMBINE は何かというと、COMBINE(結合)と言う意味から推測できるように、送るべきデータをある程度の大きさのデータにまとめて転送する方式を表しています。このまとめて転送するのを「バーストモード」と言ったりするようです。注意点としてはこのまとめて送る際には順序は考慮されないので、使いどころは気をつけた方が良さそうです。

WRITE_BACK と WRITE_COMBINE は CPU からメモリにデータを書き込んで GPU 側に転送と言う流れは共通しているように思えます。転送の方式に差があるという所ですね。CPU アーキテクチャの話だと転送の際に少しずつキャッシュに乗つけて順次転送するのが Write-Back で、キャッシュに乗つけず一気に転送するのが Write-Combine と言ったところかなと思います。一応メモリプールとの組み合わせは決まっているようなので、後の項で表にします。

メモリプール設定

次にメモリプール設定ですが、これは以下のようになっています。

```
enum D3D12_MEMORY_POOL {
    D3D12_MEMORY_POOL_UNKNOWN, // CUSTOM 以外の時はこれでいい
    D3D12_MEMORY_POOL_L0, // システムメモリ(アダプタが UMA のとき…GPU バンド幅狭)
    D3D12_MEMORY_POOL_L1 // ビデオメモリ(アダプタが NUMA のとき…GPU バンド幅広)
};
```

まず、CUSTOM 以外の時は UNKNOWN で構わないといいのは、CPU ページ設定と同様です。次に L0 と L1 についてですが、これはよくキャッシュに L1~L2 とかあります、その事だと思って良いかなと思います。

L0 について

L0 はシステムメモリを表しています。アダプタ(グラボ)の状況によって変わってくるようです。MSDN の解説を見る限り、オンボード(UMA)の場合は、この L0 しか選択肢がないようです。もし、Nvidia や AMD などのディスクリートグラボを参照している場合に L0 を選ぶと CPU バンド幅が広く、GPU バンド幅が狭くなるようです。

L1について

L1はビデオメモリを表しています。これはオンボード(UMA)の場合には使用できないパラメータです。nVidiaやAMDのディスクリートグラボを参照している場合にのみ使用できます。L1設定にするとGPU用のバンド幅が広くなりますが、CPU側からのアクセスができなくなります。

組み合わせ

ここまで読んでもらったことをまとめて表にしておきます。これ以外の組み合わせでは私の環境ではINVALIDARGが返ってきました。

ヒープ種別	ページ設定	メモリペア ル	RESOURCE_STATE	Map可?
DEFAULT	UNKNOWN	UNKNOWN	GENERIC_READ	不可
UPLOAD	UNKNOWN	UNKNOWN	GENERIC_READ	可能
READBACK	UNKNOWN	UNKNOWN	COPY_DEST	可能
CUSTOM	NOT_AVAILABLE	L0/L1	GENERIC_READ	不可
	WRITE_BACK	L0のみ	GENERIC_READ	可能
	WRITE_COMBINE	L0/L1(?)	GENERIC_READ	可能

※とはいえ、nVidiaのグラボでしか検証していないため、AMD等の時の挙動は各自で検証してほしいと思います(あと、全ての組み合わせ検証を完璧にやったわけでもないため、使う時は用途に合わせて各自検証してください)。あとCOMBINE&L1の組み合わせが不安定です。

あと、Map可?という項目がありますが、Mapについては次項で説明しますが、簡単に言うとこのバッファを通常のCPU側メモリアクセスのように扱えるようにするための処理をMapと言い、それが可能かどうかという事です。

さて、長々と設定について話してきましたが、ひとまず頂点データはCPU側から設定するものなので、Mapで設定するとして UPLOAD で作ろうと思います。モーフ(ブレンドシェイプ)などをやらない限りは頂点データは毎フレーム1回くらいしか参照しないでしょうし、UPLOADで問題ないかと思います。ほかの指定に比べるとパフォーマンスが悪いため頻繁にアクセスがある場合には別設定を検討する必要があるとは思います。

後々パフォーマンスを考慮したり、用途に合わせて、これらのバッファの作り方を選択していくことになりますが、まずは単純に頂点を GPU に送りたいので、パフォーマンスの事は置いといて UPLOAD を使用していきます。

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 O 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」([HYPERLINK "https://msdn.microsoft.com/DD3973CC-043E-486E-9403-B46D8B7DE644#D3D12_BOX" を参照](https://msdn.microsoft.com/DD3973CC-043E-486E-9403-B46D8B7DE644#D3D12_BOX))。

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か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、これ使ったやつの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/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よこ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。

つまるところ、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と思いますが、最後の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。

サンプラーを設定

サンプラーは 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あっておいてください。  
これはルートシグネチャのパラメータにあ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.hsls 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ーの 0番レジスタを設定します。CPU 側から投げたテク
スチャやサンプラーが↑の tex や smp になります。

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)

を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、チョット頑張って書いてみてください。シェーダー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

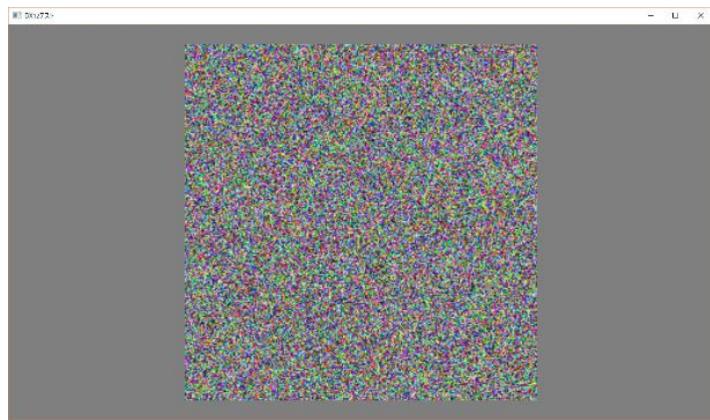
<https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12graphicscommandlist-setgraphicsrootdescriptortable>

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、皆は自分が好きな絵を表示したいでしょ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);  
            bgr[0] = bgr[2];  
            bgr[2] = bgr[0];  
            bgr[1] = bgr[1];  
            bgr[2] = bgr[3];  
            bgr[3] = 0;  
            for (int i = 0; i < 3; i++)  
                rgba[i] = bgr[i];  
            rgba[3] = 255;  
            buff[address] = rgba[0];  
            buff[address + 1] = rgba[1];  
            buff[address + 2] = rgba[2];  
            buff[address + 3] = rgba[3];  
        }  
    }  
    return ret;  
}
```

```

        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。

画像ファイルを読み込んで表示する

これが授業だったら画像ファイル読み込みも全て自分で作るので、紙幅も考慮してここは DirectXTex を使用したいと思います。

まずは DirectXTex を以下のサイトからダウンロードしてください。Git でダウンロードしてもいいですし、ZIP で落として解凍してもらって構いません。

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

これを落として使用する際には WindowsSDK のバージョンに注意しましょう。バージョンがかけ離れていると正しく動作しない可能性があります。一応ここでは Git 使ってない人用に ZIP で落とした前提で話します。

解凍したら DirectXTex-master.zip という名前のファイルがありますので、解凍して中にあるソリューションを選択します。自分が使用しているバージョンの VisualStudio が 2017 であれば、

DirectXTex/Desktop_2017_Win10.sln を開いてください。もちろん使用している VS が 2019 の場合は 2019 の方を選んでください。

ビルドするのですが、そもそも SDK バージョンが違うとビルド出来ないのでその場合はプロジェクトで右クリックして「再ターゲット」してください。

ビルドでライブラリを作りますが、その際には今の自分のプロジェクトに合わせて Win32/x64 および Debug/Release を選択してください。例えば自分のプロジェクトの設定が x64 で Debug なら、このライブラリも合わせて x64 の Debug にしてビルドしてください。

あとはここにパスを通します。環境変数を作つてもいいですし、直アドレスを指定してもいいです。僕は環境変数を作つてプロジェクト設定で指定するのをお勧めします。

例えば私の場合だと

DirectXTex-master\DirectXTex

を環境変数で DXTEX_DIR という別名として登録しておき、

VisualStudio の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。

なお、環境変数設定後は一度 VisualStudio を再起動しなければならないため、動いている VisualStudio を落としてから、また立ち上げ直してください。

で、うまくいけば、まずはインクルードディレクトリは見えているはずなので、

#include <DirectXTex.h>

と書いたら、いったんコンパイルして、見えていることを確認してください。

次にリンクですが

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

と書いてビルドしてみてエラーが出ないことを確認してください。さて、これで DirectXTex を使う準備ができました。

(注意)まだロードしていないので検証できないと思いますが、もし LoadFromWICFile の結果がインターフェイスがないとかなんとかいうエラーメッセージが出ることがあります。その場合は

```
auto result = CoInitializeEx(0, COINIT_MULTITHREADED);
```

を最初に記述してください。そもそも LoadFromWICFile は Windows 側の機能を使用してロードするものなのですが、それを利用するには COM にお伺いを立てねばならず、運が良ければそのまま機能しますが、そうじゃない場合は機能しませんのでご確認ください。

画像は基本的なファイル形式のロードに関しては LoadFromWICFile で事足りると思います。

これは bmp や png や jpg に対応しています。

ただし MMD に使用されているテクスチャファイルは他にも tga や dds があります。これをロードする場合はまた別の関数を使用します。でもまずは一般的なファイル形式を読み込んで張り付けてみましょう。

用意するファイルは何でも構いませんが、一応 Windows10 の PC でサムネイルが見れる状態の

ファイル(bmp,png,jpgなど)を使用してください。できれば正方形でもっと言うと画像のサイズが2のn乗(256とか1024とか)になってるファイルを選んだ方がいいでしょう。
少し著作権とかが怖いので、自分で作ったCGをテクスチャとしました。こういうやつ



ともかくこれが、画面上のポリゴンに張り付ければいいわけです。ちなみに img フォルダの中に textest.png という名前で保存しています。

それではさっそく LoadFromWICFile を使用しますが、この関数のファイルパス文字列は wchar_t 型であるため、ちょっとだけ指定が面倒です。今回のテストでは文字列定数なので L をつけるだけでどうにかなりますが、後々 PMD ファイルから読み取った文字列を変換する必要があるため、手間がかかるという覚悟はしておいてください。

ちなみに、この DirectXTex というライブラリと後で使用する DirectXMath は名前空間が DirectX なので、本当は DirectX:: 関数名といった書き方になりますが、頻繁に書かなければなりませんし文字数節約のために cpp のヘッタインクルードの後あたりで

```
using namespace DirectX;
```

と書いておき、Direct::といちいち書かないようにしていますので、ご注意ください。

さて、LoadFromWICFile は

```
result=LoadFromWICFile(ファイルパス, フラグ, メタデータ, スクラッチイメージ);
```

という形で呼び出します。それぞれのパラメータに関しては後述しますが、これで得たデータからバッファに必要なサイズ等の情報を得て、そして画像情報を WriteToSubresource で転送することになります。関数の中身は公開されているため、興味がある人は中身を見てみると勉強になると思います。

まずファイルパスですが、注意すべき点がひとつだけ。型が wchar_t ですので、もし文字列リテラルならば、文字列の前に L"OO.png" のように L を付けてください。

次の WIC_FLAGS_NONE についてですが、特別な事をしないという意味で、基本的には読み込ん

だデータを素直に使うという事です。ほかの指定では強制的に RGB や SRGB にするといったようなフラグがありますが、本書では NONE 以外のフラグは使用しませんので他のフラグの解説は割愛します。

次のメタデータというのは TexMetadata 構造体で、画像ファイルに関する情報(幅、高さ、フォーマットなど)が入っており、これを参照してデータの解釈に使いましょう。

最後の ScratchImage オブジェクトの中に実際のデータが入ります。生データは GetImage 関数で取得できますので、それを WriteToSubresource で転送するといった流れになります。

先ほどのノイズの出力がきちんとできていれば意外と簡単だと思います。まずは生データの取り出し方ですが、先ほど言ったように GetImage です。ちょっと引数が曲者で

```
Img->GetImage(ミップレベル, アイテム, スライス);
```

という第一引数以外がちょっと謎な感じですが、第二引数はテクスチャ配列使用時のインデックスで、第三引数は少し前にも書いたように 3D テクスチャにおける深さです。

ヒントが全然ないし、サンプルコードを見ても GetImage(0,0,0)ばかりなので分かりづらいですね。ともかく生データを以下のコードで取得します。

```
//WICテクスチャのロード
TexMetadata metadata = {};
ScratchImage scratchImg = {};
result = LoadFromWICFile(L"img/textest.png", WIC_FLAGS_NONE, &metadata, scratchImg);
auto img = scratchImg.GetImage(0, 0, 0); //生データ抽出
```

さて、この img もまた構造体になっていて、

- format: 画像データフォーマット
- width: 画像幅
- height: 画像高さ
- rowPitch: 1 行のデータサイズ
- slicePitch: 1 枚のデータサイズ
- pixels: 生データアドレス

となっています。

あれ? メタデータいらなくね?と思われそうですが、ミップレベル数とかテクスチャ配列サ

ノイズとかの情報がメタデータ側に入っているので、やっぱり必要ですね。

それでは、ノイズの時に作った ResourceDesc を書き換えて、このデータ用にしちゃいましょう。

変更のあった部分だけ記述しますね？

```
resDesc.Width = metadata.width;//幅  
resDesc.Height = metadata.height;//高さ  
resDesc.DepthOrArraySize = metadata.arraySize;  
resDesc.MipLevels = metadata.mipLevels;  
resDesc.Dimension = static_cast<D3D12_RESOURCE_DIMENSION>(metadata.dimension);
```

最後のところだけキャストしてるのは、中身が同じ数値の enum ですが、型名だけが違うのでこうします。多分 10,11,12 と共用してるライブラリなのでそういうもんだと思います。

あとは WriteToSubresource のデータコピー部をノイズテクスチャから pixels に変更します。

```
result = texbuff->WriteToSubresource(0,  
                                     nullptr,//全領域へコピー  
                                     img->pixels,//元データアドレス  
                                     img->rowPitch,//1ラインサイズ  
                                     img->slicePitch//1枚サイズ  
                                     );
```

さて、これで実行するとどうなるでしょうか？僕はこうなりました。



あれれ？色が青くなってるよ？おかしいなあ…？これはロードした画像自体が持つてるフォーマットによって違うと思いますが、実は先ほどの変更点に formatを入れていませんでした。画像情報の並びが R8G8B8A8 とは限らない！という事を知ってほしくて敢えて変更してませんでした。

さて、それではリソースの format も変更しますが、ここを変更するとビューも同時に変更しないとビューの生成に失敗しますので注意してください。

リソースフォーマット部分↓

```
resDesc.Format = metadata.format;
```

シェーダリソースビューフォーマット部分↓

```
srvDesc.Format = metadata.format;
```

これで想定した画像が表示される事でしょう。



しかし、画像フォーマットによっては、それでも元の絵よりも暗く見えたり、コントラストが上がって見えたりするかもしれません。僕の環境では暗くてコントラストが上がって見えました。

これは「ガンマ補正」と呼ばれるものが関係しており、元画像が SRGB フォーマットである場合はガンマ補正データとなっており、それをガンマ補正なしの RGBA レンダーターゲットに表示しようとするためこの問題が発生します。元の色を出したければ、レンダーターゲットから修正します。

なお、スワップチェーンのフォーマットを SRGB にしてはいけません。スワップチェーン生成が失敗します。あくまでもレンダーターゲットビュー生成部分だけです。

コードを見るとスワップチェーン情報をもとに作られているのが分かりますね。

```
for (int i = 0; i < swcDesc.BufferCount; ++i) {
    result = _swapchain->GetBuffer(i, IID_PPV_ARGS(&_backBuffers[i]));
```

```

    _dev->CreateRenderTargetView(_backBuffers[i], nullptr, handle); //ここを変更
    handle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
}

```

さて、CreateRenderTargetView の第二引数が nullptr になっており、スワップチェーンに準拠することになっていますが、ここを書き換えます。まず SRGB 用のレンダーターゲット設定をループ外で作っておきます。

```

//SRGBレンダーターゲットビュー設定
D3D12_RENDER_TARGET_VIEW_DESC rtvDesc = {};
rtvDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM_SRGB;
rtvDesc.ViewDimension = D3D12_RTV_DIMENSION_TEXTURE2D;

```

そしてこれを先ほどの CreateRenderTargetView の第二引数に入れてあげます。

```
_dev->CreateRenderTargetView(_backBuffers[i], &rtvDesc, handle);
```

ここがうまく実装できれば元通りの画像と同じような色で表示されると思います。



さて、これでテクスチャの貼り付けはひと段落です(※ただしこのやり方の場合、バッファフォーマットとレンダーターゲットフォーマットに食い違いが生じるため DebugLayer をオフにしている場合に ERROR が表示されます…どうせえと)

もしかしたら WriteToSubresource でうまくいってない人もいるかもしれませんので、次の章に行く前に別の転送方法による実装を書いておきます。うまくいってる人は読み飛ばしてもらって大丈夫です。

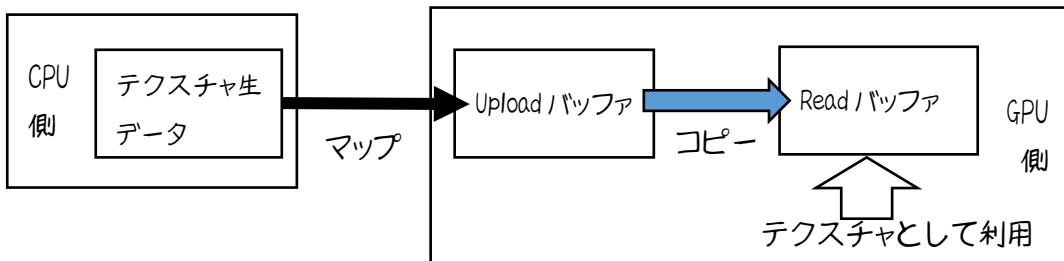
CopyTextureRegion による転送

ハードウェアの話になりますが、ディスクリート GPU(グラボが別刺しの時など)の時に、WriteToSubresource では転送がうまくいかない、もしくは動作が遅くなることがあるようです。

これは以前にも書きましたが、CPU→GPUへの転送の際の話で、`WriteToSubresource`はCPU主導でGPUへ転送するためUMA(UnifiedMemoryAccess)の際には優位性を発揮しますが、DMA(DirectMemoryAccess)転送には向いていないためです(大雑把に言うと)。

で、その場合はどうするかというと、まずUpload用のリソースをGPU上に作ります。ここは頂点バッファなんかと同じです。これをそのままテクスチャとして使用できればいいのですが、テクスチャとして使用するには読み出し用リソースである必要があります。

まったくもってややこしいのですが、大雑把な図を書くと



このような形で「CPU側の生のテクスチャデータ」をシェーダリソースとして扱える状態にします。なんでこんなややこしいことするの!?とお思いの読者の方もおられると思いますが、僕もそう思います。

グラボの内部のメモリには色々種類があって、前にも書いたようにグラボメモリは「書き込み特化型」と「読み出し特化型」があり、書き込み特化型の方はMap可能だけどクソ重いため、頻繁にデータ参照されるテクスチャとして利用するのは向きだからこういうやり方になります。

ということで、納得していただいたという事にして、実際のプログラミングをしていきましょう。前述の説明からわかるように「2種類のリソース」を作る必要があります。で、片方はUploadなのでDirectX11の時と同様にMapでテクスチャデータを書き込んでいきます。ここまでDirectX11やってる人なら大丈夫でしょうが、注意点がひとつ。ここで作るリソースはあくまでも中間バッファであり、テクスチャ用のものではないため、フォーマットにRGBAなどを指定してはいけません。単なる連續したメモリだという指定にします。

次はUpload用メモリからテクスチャのための読み出し用メモリへのコピーです。ここがCopyTextureRegionの使いどころになります。ここでうまいことコピーができればシェーダリソースとして参照可能になるということです。

ですから作業手順は

- ① アップロード用リソースの作成
 - ② 読み出し用リソースの作成
 - ③ アップロード用リソースへテクスチャデータを Map でコピー
 - ④ アップロード用リソースから読み出し用リソースへ CopyTextureRegion でコピー
- となります。

では①から

アップロード用リソースの作成

まずはヒープ設定

```
//まずは中間バッファとしてのUploadヒープ設定
D3D12_HEAP_PROPERTIES uploadHeapProp = {};
uploadHeapProp.Type = D3D12_HEAP_TYPE_UPLOAD; //←Upload用にする
//↓はUpload用に使用すること前提なのでUNKNOWNでOK
uploadHeapProp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;
uploadHeapProp.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;
uploadHeapProp.CreationNodeMask = 0; //単一アダプタのため0
uploadHeapProp.VisibleNodeMask = 0; //単一アダプタのため0
```

次にリソース設定です

```
D3D12_RESOURCE_DESC resDesc = {};
resDesc.Format = DXGI_FORMAT_UNKNOWN; //単なる塊なのでUNKNOWN
resDesc.Dimension = D3D12_RESOURCE_DIMENSION_BUFFER; //単なるバッファとして
resDesc.Width = img->slicePitch; //データサイズ
resDesc.Height = 1; //
resDesc.DepthOrArraySize = 1; //
resDesc.MipLevels = 1;
resDesc.Layout = D3D12_TEXTURE_LAYOUT_ROW_MAJOR; //連續したデータですよ
resDesc.Flags = D3D12_RESOURCE_FLAG_NONE; //とくにフラグなし
resDesc.SampleDesc.Count = 1; //通常テクスチャなのでアンチエリしない
resDesc.SampleDesc.Quality = 0; //
```

そしてアップロード用リソース作成

```
//中間バッファ作成
ID3D12Resource* uploadbuff = nullptr;
```

```

result = _dev->CreateCommittedResource(
    &uploadHeapProp,
    D3D12_HEAP_FLAG_NONE,//特に指定なし
    &resDesc,
    D3D12_RESOURCE_STATE_GENERIC_READ,//CPUから書き込み可能
    nullptr,
    IID_PPV_ARGS(&uploadbuff)
);

```

リソースステートを GENERIC_READ にするのを忘れないようにしてください。CPU から Map できるようにするためです。

コピー先リソース作成

次はコピー先のバッファを作ります。

ヒープ設定

//テクスチャのためのヒープ設定

```

D3D12_HEAP_PROPERTIES texHeapProp = {};
texHeapProp.Type = D3D12_HEAP_TYPE_DEFAULT;//テクスチャ用
texHeapProp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;
texHeapProp.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;
texHeapProp.CreationNodeMask = 0;//單一アダプタのため0
texHeapProp.VisibleNodeMask = 0;//單一アダプタのため0

```

リソース設定(変数は使いまわし)

//リソース設定(変数は使いまわし)

```

resDesc.Format = metadata.format;
resDesc.Width = metadata.width;//幅
resDesc.Height = metadata.height;//高さ
resDesc.DepthOrArraySize = metadata.arraySize;//2Dで配列でもないので1
resDesc.MipLevels = metadata.mipLevels;//ミップマップしないのでミップ数は1つ
resDesc.Dimension = static_cast<D3D12_RESOURCE_DIMENSION>(metadata.dimension);
resDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;

```

あとはリソース作成

//テクスチャバッファ作成

```

ID3D12Resource* texbuff = nullptr;
result = _dev->CreateCommittedResource(
    &texHeapProp,

```

```

D3D12_HEAP_FLAG_NONE,//特に指定なし
&resDesc,
D3D12_RESOURCE_STATE_COPY_DEST,//コピー先
nullptr,
IID_PPV_ARGS(&texbuff)
);

```

結果が S_OK であることを確認してください。次は Upload リソースに Map でコピーします。

Upload リソースへ Map する

ここは頂点バッファやインデックスバッファに対して行ったことと同じことをやればいいです。

```

uint8_t* mapforImg = nullptr;//image->pixelsと同じ型にする
result = uploadbuff->Map(0, nullptr, (void**)&mapforImg); //マップ
std::copy_n(img->pixels, img->slicePitch, mapforImg); //コピー
uploadbuff->Unmap(0, nullptr); //アンマップ

```

以上です。

あとはこれをテクスチャ用リソースへコピーするだけです。ここが一番厄介なんですが…。

CopyTextureRegion で Upload からコピー

まず CopyTextureRegion ですが、これはコマンドリストの関数です。ここからはコピー元もコピー先もグラボ上なので GPU に対する命令となります。

という事は、フェンスを用いて待ちを入れないといけません。またテクスチャ用リソースが「コピー先」のままで、バリアを用いて COPY_DESC → PIXEL_SHADER_RESOURCE にする必要があります。

ともかくまずは CopyTextureRegion 関数を確認してみましょう。

```

void CopyTextureRegion(
    const D3D12_TEXTURE_COPY_LOCATION *pDst, //グラボ上のコピー先アドレス
    UINT DstX, //コピー先領域開始 X(0 でいいです)
    UINT DstY, //コピー先領域開始 Y(0 でいいです)
    UINT DstZ, //コピー先領域開始 Z(0 でいいです)
    const D3D12_TEXTURE_COPY_LOCATION *pSrc, //グラボ上のコピー元アドレス
    const D3D12_BOX *pSrcBox //コピー元領域ボックス(nullptr でいいです)
);

```

このような定義になっています。気になるのは D3D12_TEXTURE_COPY_LOCATION ですね。これ

も定義を確認しましょう。

```
struct D3D12_TEXTURE_COPY_LOCATION {
    ID3D12Resource *pResource; //これはリソース入れいやれ！
    D3D12_TEXTURE_COPY_TYPE Type; //コピー種別(FOOTPRINT か INDEX を選ぶ)
    union { //出たよ…共用体
        D3D12_PLACED_SUBRESOURCE_FOOTPRINT PlacedFootprint; //コピー領域に関する情報
        UINT SubresourceIndex; //インデックス
    };
};
```

はい、また出ました共用体。もう慣れましたね？共用体を使用するためにどちらを使うかを Type で指定しますがコピー元には D3D12_TEXTURE_COPY_TYPE_PLACED_FOOTPRINT を選択します。

となると察していると思いますが、D3D12_PLACED_SUBRESOURCE_FOOTPRINT 構造体についても見ておく必要があります。

```
struct D3D12_PLACED_SUBRESOURCE_FOOTPRINT {
    UINT64 Offset; //0 よさそう
    D3D12_SUBRESOURCE_FOOTPRINT Footprint; //げつ、また構造体かよ…
};
```

ということで、さらに Footprint の型も見てみます。

```
struct D3D12_SUBRESOURCE_FOOTPRINT {
    DXGI_FORMAT Format; //フォーマット
    UINT Width; //幅
    UINT Height; //高さ
    UINT Depth; //深さ
    UINT RowPitch; //1 行あたりのバイト数
};
```

ちなみにフットプリントというのは、メモリ占有領域に関する情報を示します。これは大丈夫ですね？

さて、ざっと書きましたがこの情報をコピー元とコピー先と両方に作ってあげなきゃいけません。でここにもちょっと罠的な仕様があるんですが、MSDN の解説を見ると

- 『Type が D3D12_TEXTURE_COPY_TYPE_PLACED_FOOTPRINT の場合、pResource はバッファリソースを指す必要があります。』
- 『Type が D3D12_TEXTURE_COPY_TYPE_SUBRESOURCE_INDEX の場合、pResource はテクスチャリソースを指す必要があります。』

と書かれています。これがずっと入ってくる人は問題ないんですが僕はいまリマリしました。何を言っているのかというと、upload バッファにはフットプリントを指定し、テクスチャバッファにはインデックスを指定しなければならないという事です。

さて、それを踏まえてプログラミングしていきましょう。

```
D3D12_TEXTURE_COPY_LOCATION src = {}, dst = {};
//アップロード側設定
src.pResource = uploadbuff; //中間バッファ
src.Type = D3D12_TEXTURE_COPY_TYPE_PLACED_FOOTPRINT; //フットプリント指定
src.PlacedFootprint.Offset = 0;
src.PlacedFootprint.Footprint.Width = metadata.width;
src.PlacedFootprint.Footprint.Height = metadata.height;
src.PlacedFootprint.Footprint.Depth = metadata.depth;
src.PlacedFootprint.Footprint.RowPitch = img->rowPitch;
src.PlacedFootprint.Footprint.Format = img->format;
//コピー先設定
dst.pResource = texbuff;
dst.Type = D3D12_TEXTURE_COPY_TYPE_SUBRESOURCE_INDEX;
dst.SubresourceIndex = 0;
```

ここまで書けば準備完了です。

```
_cmdList->CopyTextureRegion(&dst, 0, 0, 0, &src, nullptr);
```

ただし、ここで終わりではありません。この項目の最初に書いた通りバリアとフェンスをやってあげる必要があります。

バリア

```
D3D12_RESOURCE_BARRIER BarrierDesc = {};
BarrierDesc.Type = D3D12_RESOURCE_BARRIER_TYPE_TRANSITION;
BarrierDesc.Flags = D3D12_RESOURCE_BARRIER_FLAG_NONE;
```

```
BarrierDesc.Transition.pResource = texbuff;
BarrierDesc.Transition.Subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES;
BarrierDesc.Transition.StateBefore = D3D12_RESOURCE_STATE_COPY_DEST;//ここ重要
BarrierDesc.Transition.StateAfter = D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE;//重要
```

そして Execute で転送実行

```
_cmdList->ResourceBarrier(1, &BarrierDesc);
_cmdList->Close();
//コマンドリストの実行
ID3D12CommandList* cmdlists[] = { _cmdList };
_cmdQueue->ExecuteCommandLists(1, cmdlists);
```

そして待ち

```
_cmdQueue->Signal(_fence, ++_fenceVal);
if (_fence->GetCompletedValue() != _fenceVal) {
    auto event = CreateEvent(nullptr, false, false, nullptr);
    _fence->SetEventOnCompletion(_fenceVal, event);
    WaitForSingleObject(event, INFINITE);
    CloseHandle(event);
}
```

さて、ここまでがなんとかエラーなく記述できれば、WriteToSubresource の時と同様にテクスチャ情報が転送されるはずですから、用意したテクスチャが張り付けられて表示されるはずです。

今度こそテクスチャを張り付ける部分が終わりました。お疲れ様です。

d3dx12.h (C030X～) を導入する

ここまで敢えて d3dx12.h を導入してきました。

その理由は理解が浅いうちにブラックボックスを使いたくなかったからですが、ここまでリソースの作成やバッファのコピー、バリアの設定等を面倒に設定してきましたのでコード量がかなり増えています。それを整理するために d3dx12 を導入したいと思います。

もし中身に疑問が湧いてもここまで読み進めていただけた皆様なら、なんとなく d3dx12.h のコードを理解することができると思います。もし難しければ前の項を読み直しながら理解すればいいと思います。

もし導入しないと決めてる読者はそのままでも構いません。今後のソースコードは d3d12x.h を利用する部分が出てきますが、ここまでついてこれた人は利用しないコードを自分で書けると思います。

えっ？ d3dx12.h あるのかよ？ ふざけるな!! と思う人もいるかもしれません。昔のように高機能なものではなく最低限の… 要はここまでやってきたコードを整理する程度のものでしかないです。しかしコードを整理できるのも確かなので、思い切って導入しましょう。

場所はどこにあるのかというとマイクロソフト公式が GitHub に置いてる「サンプル」の中 있습니다。え？ ライブドアじゃないの？ サンプルなの？ と思うかもしれません。僕も同感です。さらに言うとあくまでサンプルなので適宜更新されています。SDK とのバージョンが食い違っているとコンパイルすら通らないことがあります。

そういう意味であまり使いたくなかったというのもここまで使用を引き延ばした理由ではあります。まあその辺に関しては DirectXTexture も同様ですね。ともかく落としてきましょう。サンプルすべて落としても構いませんが必要なのは d3dx12.h だけなので、

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

からダウンロードしていただくか、サンプルそのものを ZIP か Git クローンで取得するかもしくは DirectXTexture のプロジェクトの内部に d3dx12.h がありますので、それを使用してもいいと思います。

既に DirectXTexture にパスを通しておるので、こちらを参照したいと思います。ということで

```
#include<d3dx12.h>
```

と書きます。これで利用可能となりました。具体的な利用部分について説明しますと例えば頂点バッファを作る際には

```
D3D12_HEAP_PROPERTIES heapprop = {};//頂点バッファ用ヒープ設定  
heapprop.Type = D3D12_HEAP_TYPE_UPLOAD;  
(中略)  
D3D12_RESOURCE_DESC resdesc = {};//頂点バッファ用リソース設定  
resdesc.Width = sizeof(vertices);  
(中略)  
ID3D12Resource* vertBuff = nullptr;
```

```

result = _dev->CreateCommittedResource(
    &heapprop,
    D3D12_HEAP_FLAG_NONE,
    &resdesc,
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&vertBuff));

```

という風にヒープ設定とリソース設定を細かく設定して頂点バッファを作っていましたが、この部分が

```

ID3D12Resource* vertBuff = nullptr;
result = _dev->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD), // UPLOADヒープとして
    D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(sizeof(vertices)), // サイズで適切な設定してくれる
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&vertBuff));

```

これだけで済むようになります。当然コードの量をかなり減らせます。d3dx12.h は中身を見れるので何をやっているのか知りたい人は見てみましょう。

リソース作成部分がかなり短く書けたところで次はバリアの設定についても見てみましょう。

```

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

```

この部分です。バッファの状態を PRESENT から RENDER_TARGET へ遷移させるバリアですね。これを CD3DX の機能で記述すると

```
_cmdList->ResourceBarrier(
```

1,

```
&CD3DX12_RESOURCE_BARRIER::Transition(_backBuffers[bbIdx],D3D12_RESOURCE_STATE_PRESENT,  
D3D12_RESOURCE_STATE_RENDER_TARGET)//これだけで済みます  
);
```

などと、行数をかなり減らせる代表的なものだけ紹介しましたが、これ以外にも便利な関数やクラスや構造体がありますので、適宜使用すればかなりコードがスッキリすると思います。

なお、12 項で説明した CopyTextureRegion に関しても UpdateSubresource という関数が用意されており、ある程度はコード量を減らせると思います（結局バッファは2つ必要なんですか）。

ともかくここまで話を理解できたかどうかは d3dx12.h の関数やクラスのコードを読んでみて「なぜこうなっているんだろう？」と考えて「こういうことかな？」「こうかもしれない」「これはこんな時に使えそうだ」と自分で判断できるようになれば、もう大丈夫だと思います。

逆に苦言を呈しておきますと、この d3dx12.h はサンプルや DirectXTexture についているものであり、内部の関数やクラスに関しては「公式でありながら公式的な説明が乏しい」ため、内部で何をやっているかを理解しないままにやみくもに使用するのは危険だと思います。

あくまでも内部で何をやっているかを理解したうえで使用されることをお勧めいたします。

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 だとそもそもいけないんだわあ…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？ 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 \Rightarrow 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);
```

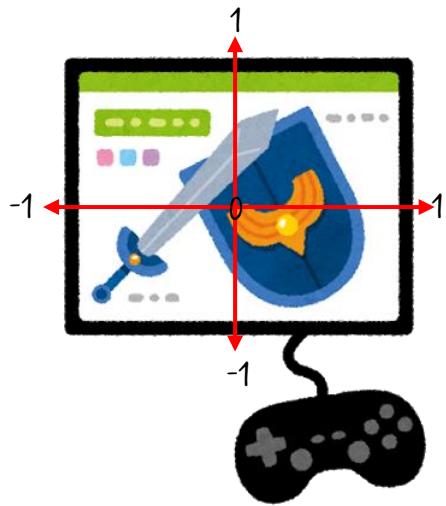
と書いてほしい。

まあこっちの方が直感的に考えられて便利でしょ？うん、そう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だけで十分だったりする。

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ね。

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。もうがいいのだ。
ほんマ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？ 言つたからな？

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ネチャだ。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)

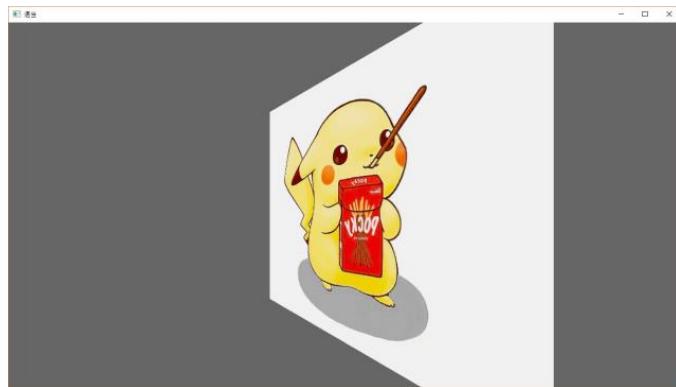
です。ここから 3D 用の行列を丁寧に作っていきましょう。

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…

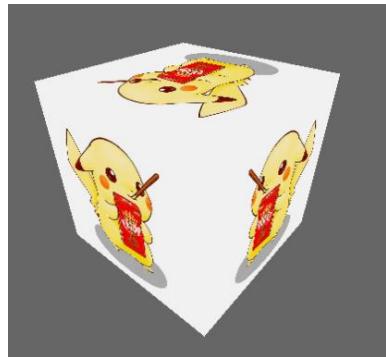
```
auto matrix = XMMatrixRotationY(angle);
auto eye = XMFLOAT3(0, 0, -30);
auto target = XMFLOAT3(0, 0, 0);
auto up = XMFLOAT3(0, 1, 0);
matrix *= XMMatrixLookAtLH(XMLoadFloat3(&eye), XMLoadFloat3(&target), XMLoadFloat3(&up));
```

```

matrix *= XMMatrixPerspectiveFovLH(XM_PIDIV2, static_cast<float>(wsize.w) /
static_cast<float>(wsize.h), 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

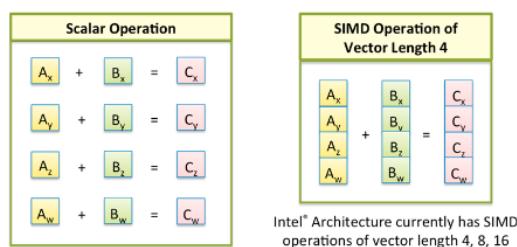
```
struct Vector3{
```

```
    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、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」とこから來てるわけです。

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うが…

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、これ一つ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つくっ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。現状の所はまだ `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ポインター)にはない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-33b6-4386-afa0-8dc38de24f31/comptr1246312521124731236420351123601239412356?forum=vcgeneral.ja>

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* rgstDesHeap = 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(&rgstDesHeap));
_rgstDesHeap.Attach(rgstDesHeap);
```

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

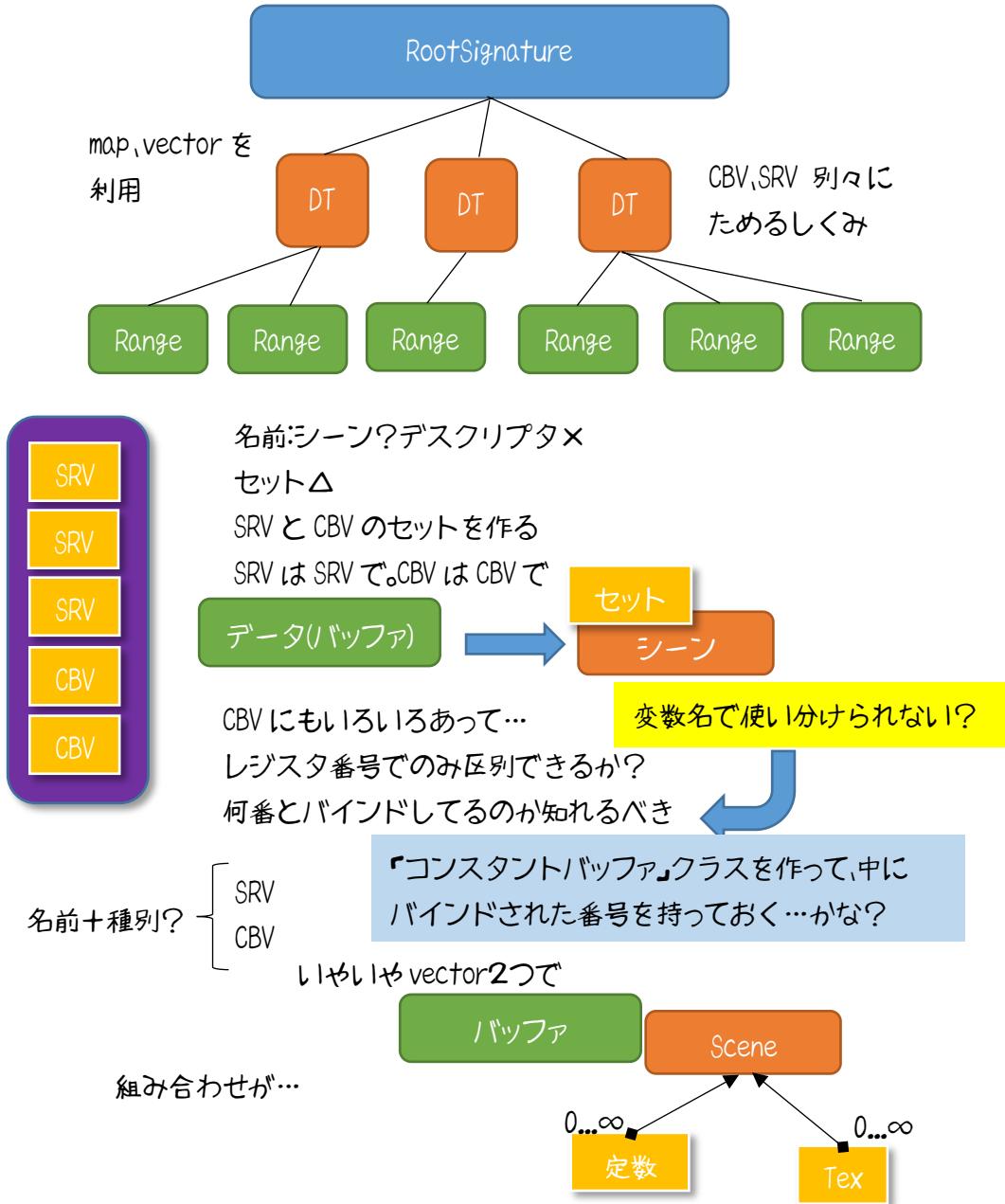
```
InitCommandSet();
InitSwapchain();
InitDescriptorHeapForRTV();
InitRenderTargetViewFromSwapchain();
InitFence();
InitVertices();
InitShaders();
InitDescriptorHeapForRegister();
InitRootSignature();
InitPipelineState();
InitTexture();
InitConstants();
```

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とんのじや。お前さんは書いとるか? この辺を考えようとするとノートが汚くなるんじや。

ノートをきれいに仕上げようなどと思ってはいけない!!!!

誰に見せるんじや…あほか。時間の無駄じやれ。自分の頭を整理するためにやるんじや。アウトプットするためにやるんじや。センセーがノートのチェックなんてしねーからのがび書け。

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ゃれ!!!!繰り返してこそ定着するんじゃれ!!!!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ゃ!!なんば繰り返しても時間の無駄無駄無駄無駄無駄ア!!!

考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ゃあ!!!書かずにやっていけるほど甘いアキテクチャちゃんじゃれ!!!!

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」

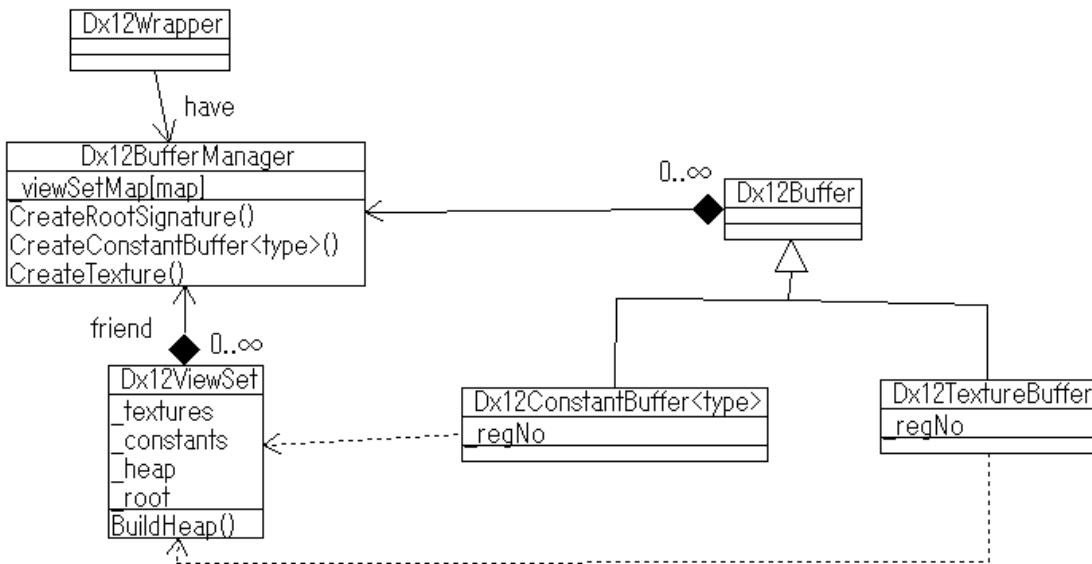


までやってるべきところをわざわざピタゴラスの定理やら連立一次方程式とかから教えないかん状況から3~4年、いや2~3年でゲームプログラマになろうってんだからそりゃキツイよ!!難しいよ!!!でも自分で選んだんじゃん!?

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ゃ。

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。

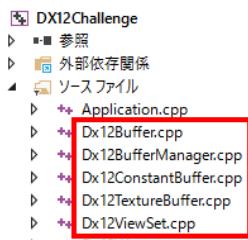
あのな、ゲームプログラマにはな? 設計力も求められるとん。コードをキレイに書く技術もそうだけどまとめあげる力も求められんねん。で、そこは「クリエイティブ」やねん。という事はセンサーのを丸写しするんじゃなくて「俺ならこう設計する」って考えながらやらんとかはつかへんねん。これはあくまでも参考程度に考えといいてや?

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戻れるけど、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つ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つたらひとまず 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れいかど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;
```

```

//直接作らせない
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 170 ページ超えてんだぞ!!10 月だぞ間に合うのか!?
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しませんが、初回兄貴はここからが楽しいのではないでしょか?

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/acbaa40b23783309c8db502335bdebba

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のはつりに考えて上のページからコピペして

```
struct PMDHeader{  
    char magic(3); // "Pmd"  
    float version; // 0000803F == 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てたら上等ですもん。逆に

僕がなんかするまで待ってるようじゃ、そりやプログ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おるか？

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知れない。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っ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 / パテ

イング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。

////

アライメントしていない場合は前の 4 バイトのケツ 1 バイトと、次の 4 バイトの 3 倍とを合わせることになり、処理が非常に重くなる。こういう理由でアライメントって仕組みが入ってしまう。

なので、コンパイラがご親切にも（おせつかいにも？）パディング（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

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/5a1b1be2fb10b7838dfc6bd010389707

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。それだけ知つてればいい。

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。

フリ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、padding 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。さっさと見てみたいだろ？

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



こんなん出ます

ちなみに僕が1回やらかしたノグとして…

```
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。まあぶつちやけると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/7cebae143cb6b9bae38ebbefc228c05b

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、

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

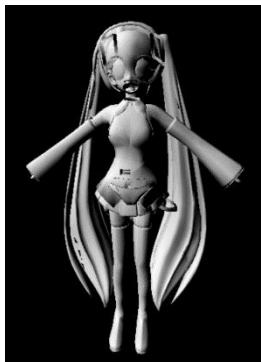
今回この k は 1 としてもいいので、あと HLSL では内積関数は dot ってのがありますので、それを使いましょう。正規化は normalize です。

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/%E6%8C%83%E3%83%90%E3%83%83%E3%83%95%E3%82%A1>

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

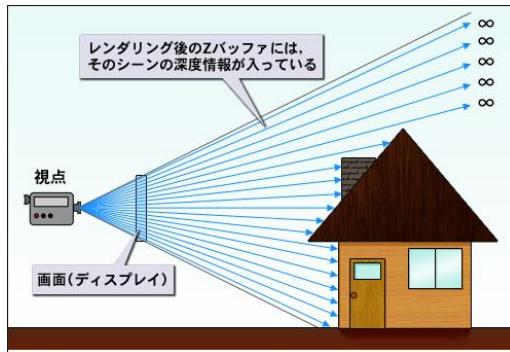
そう…深度バッファ法とは、視点からの距離を各ピクセルで…保持しておいて、今から色を塗るという時にその深度と今持っている深度を比較して、既に手前が塗りつぶされていれば破棄するという仕組みです。

深度バッファとは

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

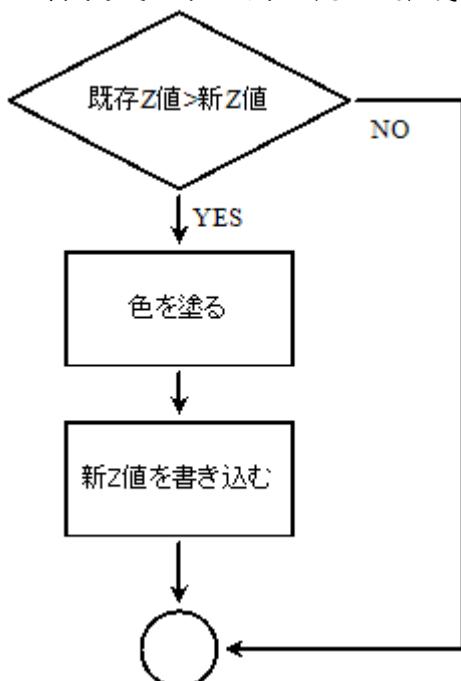
「深度」というのは何かというと、カメラから見た時のカメラからの距離の事です。いわゆる Z 値というのですが、UE4であったり3dsMaxであったりが Z を上方向として定義しているため、 Z 値って言うのが不適切になっちゃって、そのせいで「深度」というようになったんですね。

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ぬ(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

そらくはノ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。

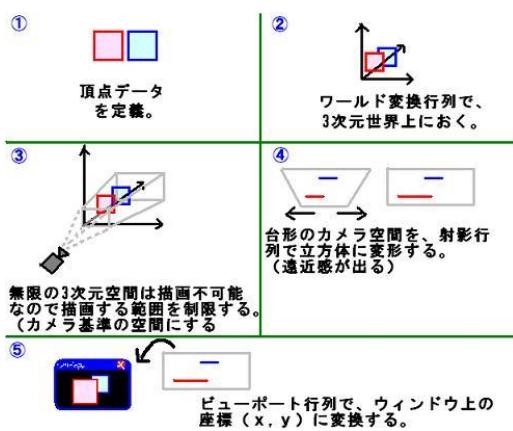
さて、というわけでちょっと不思議なノッファ…デプスStencilノッファを作つていましょ。え？ 深度ノッファじゃないの？ という疑問はごもっともではあるのだが、なんかそういう仕様になっているのだ。ちなみに深度ノッファは通常の float と同様に 32bit のノッファである。

なお、Stencilも使用したい場合はこのノッファのビットを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、「プロジェクション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、このプロジェクション行列変換は 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ノッファを作つていこう。もちろんいつものように深度ノ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. 深度ノッファ作成

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 = ws.size.width;//画面に対して使うバッファなので画面幅
depthResDesc.Height = ws.size.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)デプスですしおすし
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？プロジェクション変換の結果、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; // 小さいほうを通過するぞ  
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、

```
posoDesc.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



「ご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。ご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。

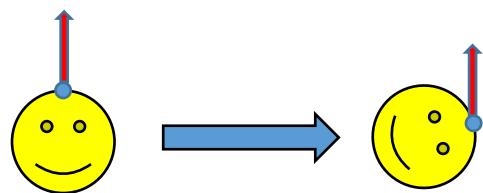
ちなみにこの機能により アルファブレンディングとの関係がうまくいかないことがあります、それはまあ…仕方ないと思ってください。そのうちその話はします。

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

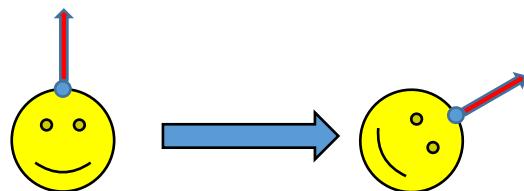
法線も座標変換

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ないので、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うのである。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～。

ということで面倒だという事を念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ゐるはず。当たり前だよなあ？

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。

スペキュラー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」とって言ってますね。

いつもの流れです。

```
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,sq,sb // 光沢色  
float mirror_color(3); // mr,mq,mb // 環境色(ambient)  
BYTE toon_index; // toon?? bmp // 0.bmp:0xFF, 1(01).bmp:0x00 ... 10.bmp:0x09  
BYTE edge_flag; // 輪郭、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、HYPERLINK  
"http://blog.goo.ne.jp/torisu_tetosuki/e/7cebae143cb6b9bae38ebbef228c05b" 面頂点リスト  
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に直接渡すわけではないので、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

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っておいてお前 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わけぢ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
    /// パーティング2個が予想される…ここまでで4bit
    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, 4b, 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) {  
    *cbufTemp = *cbuffer;  
    cbufTemp->diffuse = mat.diffuse;  
    cbufTemp->alpha = mat.alpha;  
    cbufTemp->specularity = mat.specularity;  
    cbufTemp->specular = mat.specular;  
    cbufTemp->ambient = mat.ambient;  
    cbufTemp->existTexture = (mat.texture != nullptr);  
    cbufTemp->existSPA = false;  
    ID3D12DescriptorHeap* texDescHeap() = { textureCreator.GetDescriptorHeap() };  
    _commandList->SetDescriptorHeaps(1, texDescHeap);  
    if (mat.toonIdx >= 0 && mat.toonIdx < 10) {  
        cbufTemp->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口クにない!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~。
```

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口クにない!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~。

とはいっても、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ない。

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は説明しますし(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、一ゼーカー/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? …もうここでピント来ると思うけど、

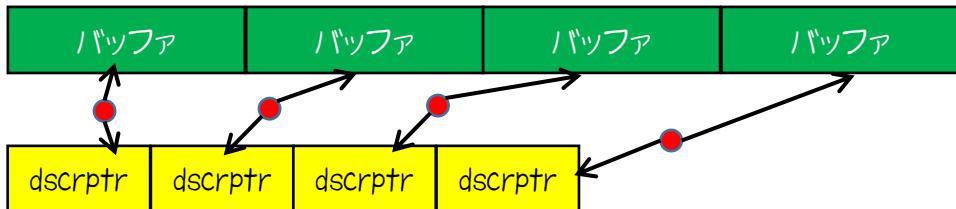
予め確保しておいて、さらにコンスタントバッファビューを作るときに

//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、バッファアドレス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。間違つてはなし。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ってしまふので、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、バッファがもったいない気がするし、ビューももったいない気がする。

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

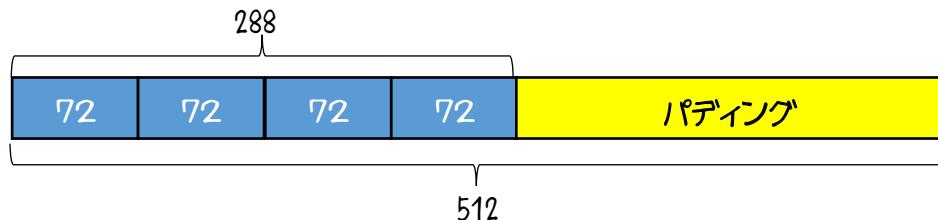


のようにすることができます。これはこれでコピーコマンドが増えることで負担が高まるんじゃないのかと言う懸念もあります。どっちがいいのかは検証してみないとわかりませんね。

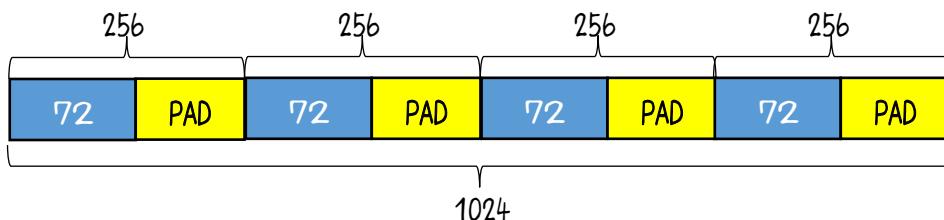
まず、最初のやり方からやってみましょう。

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でした。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。

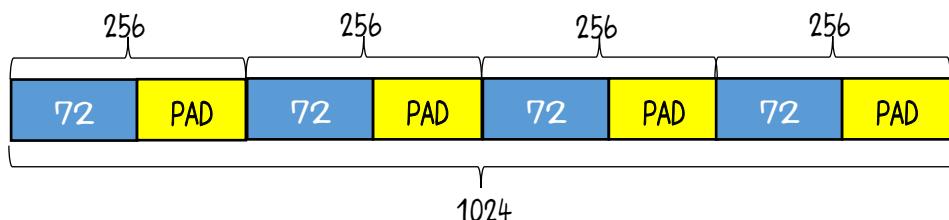
本当にこの「マテリアルの切り替え」は「鬼門」と呼んでもいい。上記の 256 区切りが本当にバグを引き起こしやすいしたかだか



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…

どっちみち



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 貢め
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います。

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。

ヒープ自身はもう作れるでしょ?ちょっと作つてみましょうよ。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つ

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か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。

あと、既に一部でディフューズアンビエントスペキュラーを入れてる人がいるので、もうやっちゃいますけどね？ 例えば

```

struct Material {
    Material() {}
    Material(DirectX::XMFLOAT3& d, DirectX::XMFLOAT3& s, DirectX::XMFLOAT3&
a) : diffuse(d), specular(s), ambient(a) {}
    DirectX::XMFLOAT3 diffuse; //拡散反射

```

```

    DirectX::XMFLOAT3 specular; //鏡面反射
    DirectX::XMFLOAT3 ambient; //環境光成分
};

こういう風に作っていたとします。

```

そうすると HLSL 側は

```

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

```

するわけで、そこにディフューズスペキュラーアンビエントが入ってくるわけです。順当にいくとこの3要素と、ザラピカゲタを乗算すればいいわけですが

ちなみに鏡面の方はハーフベクトルではなく、直接反射を計算すると

$$R = I_{in} - 2N \cdot (I \cdot N)$$

$$I_{specular} = (R \cdot V_{eye})^n$$

てな感じの式になります。ちなみに R は反射ベクトルの計算をしています。



ハーフベクトルの方が計算負荷は低いので、実際にスペキュラいうたらこっち(ハーフベクトル)が採用されることが多い…が、今回はきちんと反射ベクトルを作ろう…reflect 関数でな!!

つまり↑の式の1行目は

```

float3 mirror=reflect(light,input.normal);
となる。

```

あとはこいつと視線ベクトルの内積を取って
`float spec=pow(dot(mirror,toeye),n)`
 で、取れた明るさを乗算加算すれば、それっぽくなります。
`float spec = saturate(dot(reflect(-light, input.normal), ray));`
`spec = pow(spec, 10);`
`float brightness = saturate(dot(light, input.normal.xyz));`
`return float4(saturate(diffuse*brightness+specular*spec+ambient),1);`
 ただしこの時に面倒な問題に直面します。



なんか赤くね？

昨年の僕も見落としていたんですけどね…なんかこういう仕様があるみたいなのねん。

[https://msdn.microsoft.com/ja-jp/library/ee418340\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee418340(v=vs.85).aspx)

「各構造体は、次の変数に次の 4 成分ベクトルを開始させます。これは、構造体の配列にパディングを生成する場合があります。いずれの構造体の結果のサイズも、必ず `sizeof(4 要素ベクトル)` によって均一に割り切ることができます。」

ちなみに構造体にまとめてみてもダメですね。

```
struct Material {
    float3 diffuse;
    float3 specular;
    float3 ambient;
};

cbuffer material : register(b1) {
    Material mat;
}
```

`sizeof(float4)` 仕様にパディングが入るようです。お前この野郎。ということで、もう諦めて `float4` にしました。

```

cbuffer material : register(b1) {
    float4 diffuse;
    float4 specular;
    float4 ambient;
}

```

モチロン C++ 側も XMFLOAT4 状態にしないとズレるので、それに対処しさえすれば



おっ

さて、ちょっと疑問なのは

```

float4(saturate(diffuse.rgb*brightness+specular.rgb*spec+ambient.rgb),1);
の環境光に何も乗算されてないという所が気になりますので、検証しましょう。

```

```
return ambient;
```

で環境光(ゲタ)のみ表示しましょう。そして MMD の照明 0 の状態を見てみましょう。



んまあ、だいたいの色味は同じ感じかな。

じゃあ足しときやいいのか。あと、データにある alpha と specularity(乗算値)ですが、それぞれ

diffuse の w(4 番目)と specular の w(4 番目)に割り当てましょう。

(余談ですが、久々に MMD 立ちあげたら視野角が 30 あることに気づきました…それ合わせておくといいかもしません。)

```
auto& m = mats.back();  
m.diffuse.w = pm.alpha;  
m.specular.w = pm.specularity;  
  
しといて  
spec = pow(spec, specular.a);  
return float4(saturate(diffuse.rgb*brightness+specular.rgb*spec+ambient.rgb),diffuse.a);  
とすれば無駄なく使えついでですね。
```



なんか白っぽいんだよなあ…トーン入れたら変わるかな?

1項：マテリアルに合わせてテクスチャを貼る

テクスチャファイルのパスの合成

PMDに使用されているテクスチャファイルのパスについてですが、少し注意が必要です。

マテリアルの中にテクスチャファイルパスがありますが、これはあくまでも「PMD ファイルから見た相対パス」です。

このためマテリアルから取得したパスのままパス解決を行うとアプリケーションからの相対パスとなってしまいます。もちろん実際のテクスチャがあるパスではないため、ロードが失敗してしまいます。

ではどうすればいいのでしょうか？

「PMD のファイルパスとテクスチャファイルパス(マテリアルデータ内にある文字列)を合成する」のです。えっ？パスの合成？めんどくさそう…いえいえ、多少は手間がかかりますが、テクニカルな事はしませんのでご安心を。

例えば PMD ファイルのパスが "model/miku.pmd" で、その中のマテリアルで指定されているテクスチャが "img/tex.png" だったとします。「仮に」の話ですよ？

さて、この場合アプリケーションから見たテクスチャのパスは…そう "model/img/tex.png" ですよね。という事はどうにかして2つの文字列から、この文字列を作ればいいわけです。一応 Windows API にフォルダ名を取得する API がありますが、それは使いません。std::string の機能だけで合成します。

手順は…

1. モデルのパスの「ファイル名」が邪魔なのでそれを省く
 2. ↑の省いたパスにマテリアルデータ内の「テクスチャファイルパス」をくっつける
- となります。2は std::string なら簡単で + 演算子で結合できます。

ここで使うのは std::string の

- rfind: 逆から検索して文字が見つかった場所のインデックスを返す
- substr: 文字列の一部を抽出

2つの関数です。察しの良い人はプログラムまで頭に浮かんでると思いますが、こうします。あ、その前に pmd のパスは modelPath に入っていて、テクスチャのパスは texPath に入っているとします。ひとまずアプリケーションから見たテクスチャのパスを返す関数を作ってみます。

```

///モデルのパスとテクスチャのパスから合成パスを得る
///@param modelPath アプリケーションから見たpmdモデルのパス
///@param texPath PMDモデルから見たテクスチャのパス
///@return アプリケーションから見たテクスチャのパス
std::string GetTexturePathFromModelAndTexPath(const std::string& modelPath, const char*
texPath) {
    auto folderPath = modelPath.substr(0, modelPath.rfind('/'));
    return folderPath + texPath;
}

```

さて、用心深い人ならこう思うのではないかでしょう？『いやフォルダセパレータは Windows の場合は'/'とは限らんぞ？'¥'かもしれないぞ？』騙されんぞ？と。そこまで対応するならば関数内の処理はこうなります。

```

//ファイルのフォルダ区切りは¥と/の二種類が使用される可能性があり
//ともかく末尾の¥か/を得られればいいので、双方のrfindをとり比較する
//int型に代入しているのは見つからなかった場合はrfindがepos(-1→0xffffffff)を返すため
int pathIndex1 = modelPath.rfind('/');
int pathIndex2 = modelPath.rfind('¥');
auto pathIndex = max(pathIndex1, pathIndex2);
auto folderPath = modelPath.substr(0, pathIndex);
return folderPath + texPath;

```

で、ここでパス問題は解決した…かに見えるんですが、まだまだ終わらないのです。何故かというと、今、テクスチャデータのロードは LoadFromWICFile という関数を使用しているかと思います。

ここで一つ問題が発生します。pmd のテクスチャファイルパス文字列の型は char*です。これはまあ普通だな？って感じですが LoadFromWICFile の第一引数(つまりパス)は wchar_t*です。そもそも型が違うのです。軽く解説します。

char と wchar_t について

通常文字や文字列は1文字当たりのバイト数が1バイトです。8bitです。普通ですね？ですが全角文字などを使用する場合には char での表現には限界があります。

そこで出てきたのが wchar_t という「ワイド文字」というものです。これは1文字を 8bit 以上で表現する事によって、日本語等を使えるようにしたもので。もちろん中国語韓国語…特に

アジア圏の言葉にも対応しています。他にもベトナム語やアジア圏文字コードにまともに取り組もうとすると本当に大変。

細かくやっていくとそれだけで本1冊で済まないような分野なので、ここではさらっと

- `char` は1文字を1バイトで表現
- `wchar_t` は1文字を1バイト以上で表現

と思っておいてください。先ほども書きましたがそもそも型が違うため、そのまま代入しようとするとエラーになります。勘のいい読者様ならピンと来ていると思いますが、変換が必要になってくるのです。

リテラルなら `L` をつけるだけでいいんですが、元々 `char*` の文字列として持ってきたものを `wchar_t*` に変換するのは面倒です。面倒ですがやらざるを得ません。変換には Windows API の関数に

- `MultiByteToWideChar`
- `WideCharToMultiByte`

というものがあります。今回はワイド文字列に変換したいので前者の `MultiByteToWideChar` を使用します。このマルチバイトと言うのが `char*` 側に当たるのですがこれもコードページと言ふのがあって

一応日本語の文字コードというのが

- SJIS(コードページ 932)
- EUC-JP(コードページ 51932)
- UTF-8(コードページ 65001)

という種類があります。デフォルトの Visual Studio は現在 SJIS を採用しており、PMD のファイル指定も SJIS となっているため文字コードが SJIS の前提で書いていきますので注意してください。もし UTF-8などを利用する場合はそれに合わせて書き換えてください。

ひとまずここで頭に入れておいてほしいのは

- `char(1byte)` と `wchar_t(1以上byte)` という表現が C 言語の中にある
- `wchar_t` が必要な時は状況に応じてリテラルでは `L` や `_T()` を使用する

- MultiByteToWideChar でマルチバイト文字→ワイド文字変換する

ということです。それでは実際の変換プログラムを書いていきます。

char*→wchar_t*変換

MultiByteToWideChar 一発で wchar_t* が返ってくるならいいんですが、手順が少々面倒で 2 回呼び出す必要があります。

- 一回目: char を wchar_t に変換した時の「文字数」を取得する(wchar_t の配列確保のため)
- 二回目: wchar_t 配列を渡し、変換した文字列を受け取る

という手順になります。どういう事がというと、char 配列の文字列から wchar_t の文字列を作るため、受け取るための wchar_t の配列が必要になるのですが、サイズは変換するまで分からぬいわけです。

このため、1 回目の呼び出しでは、受け取り用の wchar_t 配列の先頭アドレスの代わりに nullptr を渡す事により wchar_t の要素数を得ることができます。この要素数でリサイズし、1 回目の呼び出しでその文字列をバッファにコピーを行うわけです。

何度も使うと思いますので、関数化しておきます。

```
//string(マルチバイト文字列)からwstring(ワイド文字列)を得る
///@param str マルチバイト文字列
///@return 変換されたワイド文字列
std::wstring
GetWideStringFromString(const std::string& str) {
    //呼び出し1回目(文字列数を得る)
    auto num1 = MultiByteToWideChar(CP_ACP,
        MB_PRECOMPOSED | MB_ERR_INVALID_CHARS,
        str.c_str(), -1, nullptr, 0);

    std::wstring wstr;//stringのwchar_t版
    wstr.resize(num1);//得られた文字列数でリサイズ

    //呼び出し2回目(確保済みのwstrに変換文字列をコピー)
    auto num2 = MultiByteToWideChar(CP_ACP,
        MB_PRECOMPOSED | MB_ERR_INVALID_CHARS,
```

```

    str.c_str(), -1, &wstr[0], num1);

assert(num1 == num2); //一応チェック
return wstr;
}

```

これでやっとテクスチャファイル名文字列を LoadWICfile 関数に渡す準備ができました。
それでは PMD マテリアルを利用している部分に移動してください。

マテリアルに対応したテクスチャパッファを作成する

色々とやり方はあるのですが、シンプルにいきたいと思います。

- パッファに関してはテクスチャの指定がないリソースは nullptr を入れておく
 - シェーダリソースビューはマテリアルと同数
 - シェーダリソースビューとマテリアルのビューは同じデスクリプタヒープに置く
 - ルートパラメータはマテリアルとこのテクスチャはまとめる(レンジは分ける)
- こんな感じで実装したいと思います。

と、その前に PMD ファイルオープンの部分をこう書き換えておきます。

```

string strModelPath = "Model/初音ミク.pmd";
auto fp = fopen(strModelPath.c_str(), "rb");

```

この strModelPath をテクスチャロードの時に使いますのでこのようにしています。さてまずはロード→リソース作成→データコピーまでの一連の流れを関数化しましょう。少し長くなりますが…

```

ID3D12Resource* LoadTextureFromFile(std::string& texPath) {
    //WICテクスチャのロード
    TexMetadata metadata = {};
    ScratchImage scratchImg = {};

    auto result = LoadFromWICFile(GetWideStringFromString(texPath).c_str(),
        WIC_FLAGS_NONE,
        &metadata,
        scratchImg);
    if (FAILED(result)) {

```

```

        return nullptr;
    }

    auto img = scratchImg.GetImage(0, 0, 0); //生データ抽出

    //WriteToSubresourceで転送する用のヒープ設定
    D3D12_HEAP_PROPERTIES texHeapProp = {};
    texHeapProp.Type = D3D12_HEAP_TYPE_CUSTOM;
    texHeapProp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_WRITE_BACK;
    texHeapProp.MemoryPoolPreference = D3D12_MEMORY_POOL_L0;
    texHeapProp.CreationNodeMask = 0; //単一アダプタのため0
    texHeapProp.VisibleNodeMask = 0; //単一アダプタのため0

    D3D12_RESOURCE_DESC resDesc = {};
    resDesc.Format = metadata.format;
    resDesc.Width = metadata.width; //幅
    resDesc.Height = metadata.height; //高さ
    resDesc.DepthOrArraySize = metadata.arraySize;
    resDesc.SampleDesc.Count = 1; //通常テクスチャなのでアンチエリしない
    resDesc.SampleDesc.Quality = 0; //
    resDesc.MipLevels = metadata.mipLevels;
    resDesc.Dimension = static_cast<D3D12_RESOURCE_DIMENSION>(metadata.dimension);
    resDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN; //レイアウトについては決定しない
    resDesc.Flags = D3D12_RESOURCE_FLAG_NONE; //とくにフラグなし
    /// ピッファ作成
    ID3D12Resource* texbuff = nullptr;
    result = _dev->CreateCommittedResource(
        &texHeapProp,
        D3D12_HEAP_FLAG_NONE, //特に指定なし
        &resDesc,
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
        nullptr,
        IID_PPV_ARGS(&texbuff)
    );
    if (FAILED(result)) {
        return nullptr;
    }
}

```

```

        result = texbuff->WriteToSubresource(0,
                                              nullptr,//全領域へコピー
                                              img->pixels,//元データアドレス
                                              img->rowPitch,//1ラインサイズ
                                              img->slicePitch//全サイズ
                                              );
    if (FAILED(result)) {
        return nullptr;
    }
    return texbuff;
}

```

で、この呼び出し側(PMD マテリアルループ)はこうします。

```

for (int i = 0; i < pmdMaterials.size(); ++i) {
    if (strlen(pmdMaterials[i].texFilePath) == 0) {
        textureResources[i] = nullptr;
    }
    //モデルとテクスチャパスからアプリケーションからのテクスチャパスを得る
    auto texFilePath = GetTexturePathFromModelAndTexPath(strModelPath,
                                                          pmdMaterials[i].texFilePath);
    textureResources[i] = LoadTextureFromFile(texFilePath);
}

```

さて、こうするとリソースロードできてるところはコピー済みのバッファが入っており、ロードできなかった場合(指定がそもそもなかったりパスが間違ってたり)には nullptr が入っています。

シェーダリソースビューを作る

次にバッファを指定するためのシェーダリソースビューを作ります。いつもだったらデスクリプタヒープを作るところですが、今回はマテリアルのヒープに相乗りさせてもらいます。つまり現在以下のようになっているマテリアル用デスクリプタヒープを

CBV	CBV	CBV	CBV
-----	-----	-----	-----

以下のような並びにしたいわけです。バッファの方じゃないので気を付けてください。

CBV	SRV	CBV	SRV	CBV	SRV	CBV	SRV
-----	-----	-----	-----	-----	-----	-----	-----

というわけで、まずデスクリプタ数を増やします。今のところマテリアルの設定が

```
D3D12_DESCRIPTOR_HEAP_DESC materialDescHeapDesc = {};
materialDescHeapDesc.NumDescriptors = materialNum;//マテリアル数ぶん
となっていると思いますので、これを2倍にします。
materialDescHeapDesc.NumDescriptors = materialNum*2;//マテリアル数x2
ヒープの設定自体はこれで終わりです。今までの面倒さから考えると拍子抜けですね。での、このヒープ内にシェーダリソースビュー設定をねじ込んでいきます。入れ物は確保しているので、あとは整然と置いていくだけです。
マテリアルのコンスタントバッファビューを作っているところを書き換えますが、ループ前にあらかじめシェーダリソースビューの基本的な部分は定義しておきます。
```

```
////通常テクスチャビュー作成
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;//デフォルト
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;//後述
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;//2Dテクスチャ
srvDesc.Texture2D.MipLevels = 1;//ミップマップは使用しないので1
```

あとはループ内でロードした内容を元にフォーマットを変えていけばOKです。ロードの結果、バッファがnullptrになっている時も、シェーダリソースビューは作ってしまいます。

```
auto matDescHeapH = materialDescHeap->GetCPUDescriptorHandleForHeapStart();
auto inc=_dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
for (int i = 0; i < materialNum; ++i) {
    //マテリアルコンスタントバッファビュー
    _dev->CreateConstantBufferView(&matCBVDesc, matDescHeapH);
    matDescHeapH.ptr += inc;
    matCBVDesc.BufferLocation += materialBuffSize;

    //シェーダリソースビュー(ここからが追加部分)
    if (textureResources[i] != nullptr) {
        srvDesc.Format = textureResources[i]->GetDesc().Format;
    }
    _dev->CreateShaderResourceView(textureResources[i], &srvDesc, matDescHeapH);
    matDescHeapH.ptr +=inc;
}
```

さて、これで必要なビューが作られたわけです。あとはデスクリプタレンジとルートパラメータです。

テクスチャ用デスクリプタレンジを追加し、ルートパラメータを書き替える

今のデスクリプタレンジ部分は

```
//定数ひとつ目(座標変換用)
```

```
descTb1Range[0].NumDescriptors = 1; //定数ひとつ  
descTb1Range[0].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV; //種別は定数  
descTb1Range[0].BaseShaderRegister = 0; //0番スロットから  
descTb1Range[0].OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
```

```
//定数ふたつめ(マテリアル用)
```

```
descTb1Range[1].NumDescriptors = 1; //デスクリプタヒープはたくさんあるが一度に使うのは1つ  
descTb1Range[1].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV; //種別は定数  
descTb1Range[1].BaseShaderRegister = 1; //1番スロットから  
descTb1Range[1].OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
```

となっていると思いますので、下にシェーダリソースビューのレンジを追加します。

```
//テクスチャ1つ目(↑のマテリアルとペア)
```

```
descTb1Range[2].NumDescriptors = 1; //テクスチャひとつ  
descTb1Range[2].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV; //種別はテクスチャ  
descTb1Range[2].BaseShaderRegister = 0; //0番スロットから  
descTb1Range[2].OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
```

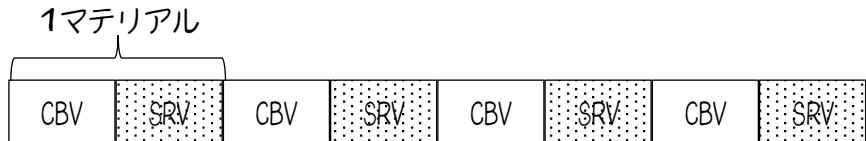
さて、↑のコメントにある「↑のマテリアルとペア」という言葉を意識しておいてください。ちょっとした意味があるんです。そして意識したままルートパラメータの書き換えですが、1 カ所だけです。

```
rootparam[1].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;  
rootparam[1].DescriptorTable.pDescriptorRanges = &descTb1Range[1]; //レンジの先頭アドレス  
rootparam[1].DescriptorTable.NumDescriptorRanges = 2; //デスクリプタレンジ数←ここ  
rootparam[1].ShaderVisibility = D3D12_SHADER_VISIBILITY_PIXEL; //ピクセルシェーダから見える
```

このようにデスクリプタレンジ数だけ書き換えればいいです。最後に描画(Draw まわり)部分です。

描画ループ回り

ここもそれほど変更の必要はないのですが、デスクリプタヒープの並びが



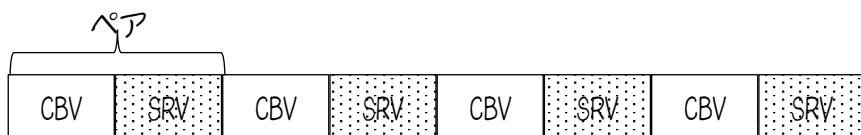
1マテリアルあたりのヒープに2つ分使用しています。ですので、もともとDrawの際にインクメントサイズを算出し、その分だけポインタを進めていたと思いますが、これを2倍にする必要があります。このため、

```
auto cbvsrvIncSize = _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV)*2;//2倍
for (auto& m : materials) {
    _cmdList->SetGraphicsRootDescriptorTable(1, materialH);
    _cmdList->DrawIndexedInstanced(m.indicesNum, 1, idxOffset, 0, 0);
    materialH.ptr += cbvsrvIncSize;//次ビューのために2倍進める
    idxOffset += m.indicesNum;
}
```

のようになります。「え？ マテリアルの固定ビューとシェーダリソースビューの切り替えはいらないの？」と思った読者さんは鋭いと思います。

これは事前にデスクリプタヒープの並びとデスクリプタテーブルの並びを隣にすることによって、2つのビューを同時に使う事ができるという技なのです。

しつこいでですが、デスクリプタヒープの並びはこうなっています。



で、ルートパラメータにおいては

```
rootparam[1].DescriptorTable.NumDescriptorRanges = 2;
```

とやっていました。これは2つのレンジ(CBV と SRV)がデスクリプタテーブルに並んでいる事を示しており、同じようにデスクリプタヒープでビューが並んでいればいちいち切り替えるのではなく2つ同時に指定することができるわけです。

この辺は慣れないし難しいと思いますが、どの道すぐ同じような事をする必要が出てきますので、ぼちぼち慣れていくください。

HLSL(シェーダ)側のコード

シェーダ側のコードも簡単です。テクスチャとディフューズ色を乗算すればいいのですから

```
float3 light = normalize(float3(1, -1, 1));
float brightness = dot(-light, input.normal);
return float4(brightness, brightness, brightness, 1)*diffuse*tex.Sample(smp, input.uv);
```

このようにすれば、ランパートの余原則で得られた輝度と、マテリアルのディフューズカラーと、テクスチャが乗算され、テクスチャが適用された状態になる…はず。

さて、ここまでがきちんとコーディングできていれば、ミクさんの目にテクスチャが…



貼られ…ん?

貼られ…



貼られてはいるようだが…なんで真っ黒なの…?

2項： テクスチャの有無でマテリアルに不具合が起きないように

さて、目的である目のテクスチャは貼られたのですが、他のマテリアルが真っ黒になってしまいました。これは何故でしょう？

原因

ピクセルシェーダを見てください。

```
float3 light = normalize(float3(1, -1, 1));
float brightness = dot(-light, input.normal);
```

```
return float4(brightness, brightness, brightness, 1)*diffuse*tex.Sample(smp, input.uv);
```

こんな感じではないでしょうか？ちなみにテクスチャファイル指定は5番目だけに入っています。あとは全部テクスチャなし(rgbaがnullptr)です。

テクスチャなしの場合は nullptr を指定しています。テクスチャ用rgbaを nullptr として CreateShaderResourceView を作った時の挙動はMSDNによると「null バインディングが保障され読み取り 0 で書き込み破棄されます」とのことです。

つまりテクスチャがないすべてのピクセルにおいて、

```
return float4(0,0,0,0)*diffuse;
```

となっていると思われます。え？これだと透明になるはずでしょ？って思われるかもしれません、まだアルファブレンドを有効にしていないため、真っ黒になります。アルファを有効にしたうえでこれをやると、目玉だけが浮いていてわかりづらいので、今のところはアルファを OFF にしています。

となると、真っ黒になるのは理由が分かりましたね？ではどう対処しましょう…シェーダ側ではrgbaが nullptr かどうかなんて知りようがありません。どうしたらいいんでしょうか？

白テクスチャを作成し、テクスチャがないときは白テクスチャを割り当てる
色々とやり方は考えられますが、シェーダ側に色々書きたくないので「白テクスチャ」を作り、テクスチャが nullptr の時はビューアがこの「白テクスチャ」を指定するようにします。

「白テクスチャ」とは何でしょうか？もう一度ピクセル色決定のコードを見てください。

```
return float4(brightness, brightness, brightness, 1)*diffuse*tex.Sample(smp, input.uv);
```

こうですよね？

これはテクスチャがあればうまくいくけど、テクスチャがないと真っ黒になりました。だから本当は、テクスチャがない場合には

```
return float4(brightness, brightness, brightness, 1)*diffuse*float4(1,1,1,1);
```

となってほしい。先ほども言いましたが、テクスチャが nullptr かどうかはシェーダ側から分かりません。

なので「白テクスチャ」という float4(1,1,1,1)に当たるものを作つてあげればいいのです。作れ

ばいいのでロードなどは行いません。中身を 1,1,1,1 で初期化するだけです。

白でありさえすればいいので、大きさはテクスチャの最小単位の 4x4 にしましょう。やり方はもうそろそろ解説しなくても分かると思いますので、コードだけ書きます。

```
ID3D12Resource*  
CreateWhiteTexture() {  
  
    D3D12_HEAP_PROPERTIES texHeapProp = {};  
    texHeapProp.Type = D3D12_HEAP_TYPE_CUSTOM;//  
    texHeapProp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_WRITE_BACK;//  
    texHeapProp.MemoryPoolPreference = D3D12_MEMORY_POOL_L0;//  
    texHeapProp.CreationNodeMask = 0;//  
    texHeapProp.VisibleNodeMask = 0;//  
  
    D3D12_RESOURCE_DESC resDesc = {};  
    resDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;  
    resDesc.Width = 4;//幅  
    resDesc.Height = 4;//高さ  
    resDesc.DepthOrArraySize = 1;  
    resDesc.SampleDesc.Count = 1;  
    resDesc.SampleDesc.Quality = 0;//  
    resDesc.MipLevels = 1;//  
    resDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;  
    resDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;//  
    resDesc.Flags = D3D12_RESOURCE_FLAG_NONE;//  
  
    ID3D12Resource* whiteBuff = nullptr;  
    auto result = _dev->CreateCommittedResource(  
        &texHeapProp,  
        D3D12_HEAP_FLAG_NONE, //特に指定なし  
        &resDesc,  
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,  
        nullptr,  
        IID_PPV_ARGS(&whiteBuff)  
    );  
    if (FAILED(result)) {  
        return nullptr;
```

```

        }

        std::vector<unsigned char> data(4 * 4 * 4);
        std::fill(data.begin(), data.end(), 0xff); //全部255で埋める

        //データ転送
        result = whiteBuff->WriteToSubresource(0, nullptr, data.data(), 4*4, data.size());
        return whiteBuff;
    }
}

```

さて、これで float4(1,1,1,1) テクスチャができました。あとはテクスチャがないときにこれを割り当てるようにすればいいだけです。

マテリアルループに移動してください。なお「白テクスチャ」は変数名 whiteTex とします。テクスチャ指定が nullptr である場合の部分をよく見てください。

```

for (int i = 0; i < materialNum; ++i) {
    //マテリアル固定バッファビュー
    _dev->CreateConstantBufferView(&matCBVDesc, matDescHeapH);
    matDescHeapH.ptr += incSize;
    matCBVDesc.BufferLocation += materialBufferSize;

    if (textureResources[i] == nullptr) {
        srvDesc.Format = whiteTex->GetDesc().Format;
        _dev->CreateShaderResourceView(whiteTex, &srvDesc, matDescHeapH);
    } else{
        srvDesc.Format = textureResources[i]->GetDesc().Format;
        _dev->CreateShaderResourceView(textureResources[i],
            &srvDesc,
            matDescHeapH);
    }
    matDescHeapH.ptr += incSize;
}

```

とやってやれば、nullptr の時は白テクスチャとなり、ディフューズ色がそのまま出力されることがなりますので、結果は



やつたぜ!

3項： 他のモデルも試してみよう (特殊なテクスチャファイル指定)

漸くモデル出力まで来ましたが他のモデルはどうでしょう？

ここで紹介するのは MikuMikuDance に付属のモデルのみで話しますね。世の中には有志によって作られたいろんな PMD モデルがありますが、作られ方によってはややこしくなったりしますし、使用許諾の問題もありますので、読者様でお気に入りのモデルがありましたら、個人の責任と判断でそれを使っていただければと思います。

基本モデル以外のテクスチャやボーンへの個別対応は読者様で対応していただきますようお願いします。ここでは基本のみという事で、繰り返しますが MikuMikuDance に付属のモデルのみの話をていきます。

まず特殊なテクスチャについて話しますので、モデルは「巡音ルカ.pmd」を使用します。ファイルロード部分のファイル名を巡音ルカ.pmd に変更すると…巡音ルカのモデルが表示されま



すが

何かおかしいですよね？特に目が…。

スフィアマップテクスチャ名(sph,spa)との融合を外す
さて、巡音ルカ.pmd のマテリアルファイルを解析すると目の部分のファイル名指定が
"eyelu.bmp*a4.sph"

となっています。確かにこれではロードできませんね。モデルがあるフォルダの中に"eyelu.bmp"というファイルもありますし"04.sph"というファイルもあります。eyelu.bmpは目のテクスチャだろうなという事は分かりますが、この04.sphとは何なんでしょう？

これは「スフィアマップ」と言って、MMDにおいてあたかも周囲の景色が映り込んでいるかのように見せるテクスチャの事です。「スフィアマップ sph sph」等で検索してもらえばたくさん見つかるかと思います。

スフィアマップ自身の対処は後にすることとして、まずは文字列を分離して目のテクスチャ等がきちんと読み込まれるようにしましょう。

まず、セパレータが'*'アスタリスクであることが分かりますので、モデルのフォルダ名を取得するときに'/'でモデル名と分離したのを思い出しましょう。で、その分離したファイル名のどちらがスフィアマップか分かりませんので「拡張子」で判断するようにしましょう。

という事で以下の2つの関数を作りましょう。

- ファイル名を分離する関数(SplitFileName)
- 拡張子を得る関数(GetExtension)

どちらも↑で書いたように'/'で区切った方法が使えますね。特定の文字の位置をfindもしくはrfindで探して、その場所をもとにsubstrで新しい部分文字列を作り出す。

そんなに難しくはないと思いますので、関数のコードを書きます。まずは拡張子を得る関数。これは簡単ですね。

```
// ファイル名から拡張子を取得する
// @param path 対象のパス文字列
// @return 拡張子
string
GetExtension(const string& path) {
    int idx = path.rfind('.');
    return path.substr(idx+1, path.length() - idx-1);
}
```

次に区切り文字で分離する関数です。

```

///テクスチャのパスをセパレータ文字で分離する
///@param path 対象のパス文字列
///@param splitter 区切り文字
///@return 分離前後の文字列ペア
pair<string, string>
SplitFileName(const string& path, const char splitter='*') {
    int idx = path.find(splitter);
    pair<string, string> ret;
    ret.first = path.substr(0, idx);
    ret.second = path.substr(idx+1, path.length()-idx-1);
    return ret;
}

```

さてこれを分離するかどうかの判断ですが、algorithm の count 関数を使用します。string に対して count 関数を使用すると、特定の文字がいくつあるのかを返してくれます。これが1以上なら分離すべきという事です。

という事で、テクスチャファイルロードの部分はこう書き換えられます。

```

string texFileName = pmdMaterials[i].texFilePath;
if (count(texFileName.begin(), texFileName.end(), '*') > 0) {//スプリッタがある
    auto namepair=SplitFileName(texFileName);
    if (GetExtension(namepair.first) == "sph" ||
        GetExtension(namepair.first) == "spa") {
        texFileName = namepair.second;
    }
    else {
        texFileName = namepair.first;
    }
}
//モデルとテクスチャパスからアプリケーションからのテクスチャパスを得る
auto texFilePath = GetTexturePathFromModelAndTexPath(strModelPath, texFileName.c_str());
(略)

```

まずここまでやってみましょう。ここではいったん「スフィアマップ」は無視します。これらを適用するのはちょっと面倒なので次の項に回して、まずは基本テクスチャが表示されてるのを確認してください。



まあ基本テクスチャって「目」だけなんですが…

スフィアマップ sph を「一応」適用する

「一応」という言葉に引っ掛かりを覚える人も多いと思いますが、まずは手っ取り早く効果を感じてもらうために簡易的な実装から始めます。

さて、まずは sph(乗算スフィアマップ)の実装からやっていこうと思います。なぜこちらから実装するのかというと、計算についてちょっと考えてみてください。乗算という事は掛け算です。つまり色を決定するために

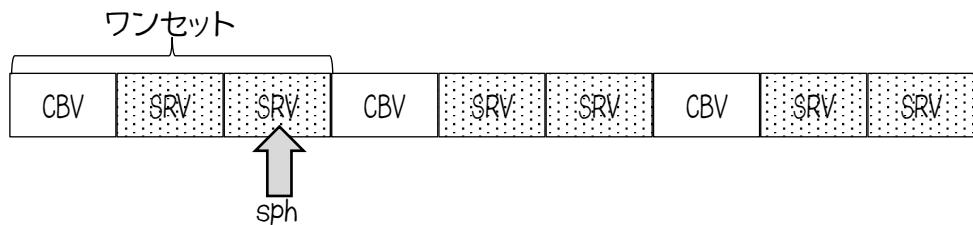
出力色 = 輝度 * ディフューズ色 * テクスチャ色 (nullptr の時はホワイト) * sph

というような式になるはずです。そうなるとテクスチャ色と同様に「指定がない場合」の色を決めておかねばなりませんが、乗算の場合それが既に作っている「白テクスチャ」を流用すればいいわけです。

となってくると必要なのは

- 「スフィアマップ(乗算)指定」があった場合のドッファの確保とデータ転送
- スフィアマップ(乗算)用の SRV の作成
- スフィアマップ(乗算)用のスロットの作成(デスクリプタレンジの拡張)

となります。そうなると賢明な読者には予想がつくと思いますが、デスクリプタヒープの並



びが

このようになります。となると描画の際のハンドルのポインタの進め方も最初の 3 倍になることが分かると思います(最終的には sph も含めて 4 倍になる)。

はい、それではまずはデスクリプタヒープの「デスクリプタ数」を増やしましょう。

```
materialDescHeapDesc.NumDescriptors = materialNum * 3; //マテリアル数ぶん(2→3)
```

あとはこれまた同様にロード部分を作り、増えたヒープに対して内部にシェーダリソースビューを追加して、合わせてデスクリプタテーブルのレンジを変更したりしていくだけです。

ロード部分ですが sph だったときに特定のバッファに入るようになります。

まずはテクスチャデータの読み込みループをしている部分に移動しましょう。そこに sph 用のバッファ配列(vector)を宣言します。

```
vector<ID3D12Resource*> sphResources(materialNum);
```

で、拡張子が“sph”だった時に sphResources(n) に代入すればいいわけです。それ以外の時は nullptr が入ってるという具合にしておきましょう。

はい、これで sphResources の各要素にバッファのアドレスが入っている(指定されていれば)と仮定します。

あとはビューア作成です。以下のコードをループ内のテクスチャ用ビューア作成の直後に書きましょう。

```
if (sphResources[i] == nullptr) {
    srvDesc.Format = whiteTex->GetDesc().Format;
    _dev->CreateShaderResourceView(whiteTex, &srvDesc, matDescHeapH);
}
else {
```

```

    srvDesc.Format = sphResources[i]->GetDesc().Format;
    _dev->CreateShaderResourceView(sphResources[i], &srvDesc, matDescHeapH);
}

matDescHeapH.ptr += incSize;

```

テクスチャの時と同じですね。指定された時のバッファの場所が違うだけです。次はデスクリプターテーブル(レンジ)です。これはすごく簡単です。既に作っているテクスチャ用レンジのNumDescriptorsの数を増やすだけです。

```

//テクスチャ1つ目(↑のマテリアルとペア)
descTblRange[2].NumDescriptors = 2;//テクスチャ二つ(基本と sph)

```

次にドロー時のヒープ内のポインタの増え方も2→3にします。

```

auto cbvSrvIncSize = _dev-
>GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV)*3;//ここ
for (auto& m : materials) {
    _cmdList->SetGraphicsRootDescriptorTable(1, materialH);
    _cmdList->DrawIndexedInstanced(m.indicesNum, 1, idxOffset, 0, 0);
    materialH.ptr += cbvSrvIncSize;
    idxOffset += m.indicesNum;
}

```

まずはここまでやって元通りに表示されるかどうかを確認してください。それができたらシェーダ側を書き換えていきます。

まず、シェーダ側に sph 用のテクスチャ変数を書き加えます。0 番スロットの t0 は既に使用しているため t1 を sph として使いますので、

```
Texture2D<float4> sph:register(t1); //1番スロットに設定されたテクスチャ
```

という風に指定します。次にピクセルシェーダですが

```

return float4(brightness, brightness, brightness, 1)*//輝度
        diffuse//ディフューズ色
        tex.Sample(smp, input.uv)//テクスチャカラー
        sph.Sample(smp, input.uv); //スフィアマップ(乗算)

```

としてみてもいいんですが、これでは「スフィアマップ」としては意味が通りません。どういうことがというとテクスチャの UV と同じなんですよね…。詳しい説明の前にスフィアマップの中身を見てみましょうか。

Model フォルダの中に "metal.sph" があると思いますので、それを画像ファイルエディタかビューワで開いてみてください。あ、拡張子 sph では開けない読者もいると思いますので、その場合には sph の拡張子を "bmp" に変更してください。え？ 大丈夫なのか？ 大丈夫なのでやってください。



つやーん

このように金属がつやつやしたような画像が入っていると思います。これがスフィアマップの正体です。実は中身は bmp だったり png だったりするのですが、それをスフィアマップとして使用するために拡張子を sph にしているだけだったのです。

ちょっと読み込みモデルを「初音ミク metal.pmd」にしてもらうとわかりやすいのですが今のまま表示してみても

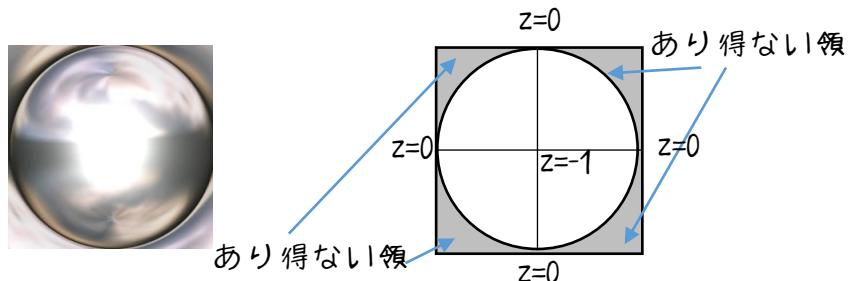


つやつやしない! じゃないか!! ふざけるな!!

となります。前にも話しましたが UV を決める要素は本当は「視線ベクトルの反射ベクトル」によって UV を決めるべきなんですが、そもそも今は「視線ベクトル」がないので、視線ベクトルなしに「それっぽく」表示させてみましょう。

通常、法線ベクトルというのは正規化されているため $x^2 + y^2 + z^2 = 1^2$ です。これは大丈夫ですね？ この式の z の項を右辺に移項すると $x^2 + y^2 = 1^2 - z^2$ になりますね？ で、先ほども言ったように法線ベクトルは正規化されておりまますので $-1 \leq z \leq 1$ ですよね？

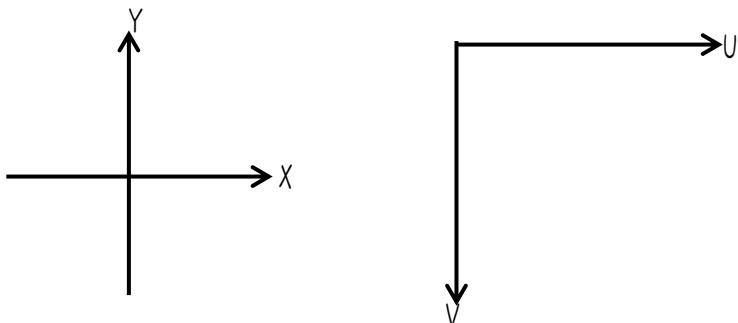
ここで、法線ベクトル(x, y, z)のうち、2D 部分(x, y)だけについて考えてみましょう。法線ベクトルが真正面に向いているとします。そして z を無視します。



そうすると $x^2 + y^2 \leq 1^2$ になりますので、法線ベクトルの XY 成分は実は円の内部を表しているといえるわけです。円の外側はあり得ない領域です。これを UV 値として利用すれば現在の法線の方向のピクセル値を取得するため、周囲が映り込んでいるように見えるわけです。

ただし、このスフィアマップのテクスチャの UV 値は $U(0 \sim 1), V(0 \sim 1)$ であり法線ベクトルの XY は $X(-1 \sim 1), Y(-1 \sim 1)$ であるため、このままでは範囲が異なっているため期待するピクセルをとできません。このため XY 値を加工します。どう加工するのかというと、マイナスがあり得ないため、両方に 1 を足すと、 $X(0 \sim 2), Y(0 \sim 2)$ になります。これを半分にすると $X(0 \sim 1), Y(0 \sim 1)$ となるため範囲も対応してくれます。

試しに $X=0, Y=0$ をこの式に使うと $(0+1)/2=0.5$ ですので、 $(0.5, 0.5)$ となり画像の真ん中になりますので、意図した場所を示していますね？ということで、これでそれっぽくなると言いたいところですが、Y 軸(V 軸)方向に問題があります。



はい、反対側に向いてるんですね。Y を反対に向けるために -1 を乗算する必要がありますね？ということで最終的には

$$(u, v) = ((x, y) + (1, -1)) / 2 * (1, -1)$$

まとめると

$$(u, v) = ((x, y) + (1, -1)) * (0.5, -0.5)$$

になります。ということでピクセルシェーダ側でこう書きます。

```

float2 normalUV = (input.normal.xy + float2(1, -1))*float2(0.5, -0.5);
return float4(brightness, brightness, brightness, 1) //輝度
        diffuse //ディフューズ色
        tex.Sample(smp, input.uv) //テクスチャカラー
        sph.Sample(smp, normalUV); //スフィアマップ(乗算)

```

のようにしてやります。このように設定してやれば、スフィアマップの期待した部分のピクセルを取得できますので、



つやつやしているのが分かると思います。

さて、あとは spa(加算スフィアマップ)ですが、ここまでついて来てる人には「ツッパやデスクリプタヒープの設定はお分かりかと思いますので、ひとまずノーヒントでやってみてください。たまには自分で考えてやってみたい」と身につきませんからね。

ちなみに“初音ミク metal.pmd”的場合だと spa が含まれていなければ、“巡音ルカ.pmd”を表示してみてください。また「加算」であるためテクスチャ指定がない場合には「黒テクスチャ」にする必要がありますので、それも考慮してご自身で考えてやってみてください。



このように髪に艶が出ていれば正解ですが…？

うまくいかない読者様のために sph の時と異なる点をかいつまんでコードを見せますね？

①デスクリプタヒープの数を増やす

```

materialDescHeapDesc.NumDescriptors = materialNum * 4; //マテリアル数ぶん(定数1つ、テクスチャ  
3つ)

```

②「黒テクスチャ」の作成

```
auto blackTex = CreateBlackTexture();
```

③加算テクスチャのためのビューを作る

```
if (spaResources[i] == nullptr) {  
    srvDesc.Format = blackTex->GetDesc().Format;  
    _dev->CreateShaderResourceView(blackTex, &srvDesc, matDescHeapH);  
}  
  
else {  
    srvDesc.Format = spaResources[i]->GetDesc().Format;  
    _dev->CreateShaderResourceView(spaResources[i], &srvDesc, matDescHeapH);  
}  
  
matDescHeapH.ptr += incSize;
```

④デスクリプタレンジのサイズを変更する

```
descTblRange[2].NumDescriptors = 3; //テクスチャ3つ(基本と sph と spa)
```

⑤Draw時の進め方を増やす

```
_dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV)*4;
```

⑥ピクセルシェーダに加算の式を追加する

```
return float4(brightness, brightness, brightness, 1) //輝度  
    *diffuse //ディフェューズ色  
    *tex.Sample(smp, input.uv) //テクスチャカラー  
    *sph.Sample(smp, normalUV) //スフィアマップ(乗算)  
    +spa.Sample(smp, normalUV); //スフィアマップ(加算)
```

加算なので+であるところに注意してください。

現在のカメラの向きも反映させる

ただし今のままであればカメラとの位置関係が考慮されません。どういうことがというと法線の方向がワールドにおける法線方向を向いているため、このままであれば正しいスフィアマップといえません。(厳密にはここからの実装も「完全に正しい」とは言えないのですが…一応MMDエディタの出力に準拠した見え方になればOKという意味で「正しい」とします)

さて、現在のカメラの向きを反映させたいのですが、そのためにはシェーダ内での viewproj を分

割して view と proj にする必要があります。でもこれは簡単ですね？cpp 側の構造体を view と proj で分けてしまって

```
XMMATRIX world;//ワールド行列  
XMMATRIX view;//プロジェクション行列  
XMMATRIX proj;//プロジェクション行列
```

のように分けてあげればいいでしょう。代入の部分も分けてしまいましょう。そこはもう簡単なのでコードはここでは書きませんので読者自身で書いてください。

次にシェーダ側ですが、シェーダ側の受け取り側も view と proj を分けてください。そして view 変換後の法線が欲しいため、ノーメンリレーの構造体に vnormal というビューバンドル後法線を作ります。

```
struct Output {  
    float4 svpos:SV_POSITION;//システム用頂点座標  
    float4 pos:POSITION;//システム用頂点座標  
    float4 normal:NORMAL0;//法線ベクトル  
    float4 vnormal:NORMAL1;//ビューバンドル後法線ベクトル  
    float2 uv:TEXCOORD;//UV 値  
}
```

で、ワールド変換した法線ベクトルに対してビューバンドル後法線を vnormal に代入します。

```
output.vnormal = mul(view, output.normal);
```

そしてピクセルシェーダ側でこの vnormal をスフィアマップの UV に利用します。

```
float2 sphereMapUV = input.vnormal.xy;
```

としておき、スフィアマップの UV に適用します。

```
sph.Sample(smp, sphereMapUV)  
spa.Sample(smp, sphereMapUV)
```

ここまでがきちんとければ、MMD と比較しても「それっぽい」艶が出てくると思います。次はスペキュラとアンビエントを作っていくましょう。

4項：スペキュラとアンビエントの実装

アンビエント(環境光成分)

まず、アンビエント(環境光成分)は簡単で、既にマテリアルにアンビエントが含まれているため先ほどのスフィアマップまで加算した上に

```
+float4(ambient, 1); //アンビエント
```

を追加すればいいだけです。本当はライトを定義する際に「環境光」を与えて、それと乗算すべきですが、それはもう少し後にしましょう。

そうすると以下のようになってしまい、目が白っぽくなってしまいます。



という事で、アンビエントカラーにテクスチャカラーを乗算するようにします。そうすると…

```
float4 color = tex.Sample(smp, input.uv); //テクスチャカラー  
return float4(brightness, brightness, brightness, 1) //輝度  
    *diffuse //ディフューズ色  
    *color //テクスチャカラー  
    *sph.Sample(smp, normalUV) //スフィアマップ(乗算)  
    +spa.Sample(smp, normalUV) //スフィアマップ(加算)  
    +float4(color*ambient, 1); //アンビエント
```

このようになり、表示のアンビエントにもテクスチャが加味されて



目の部分が白っぽくなることはなくなります。いったんアンビエントはこれでOKにしておいて、次に鏡面反射成分(スペキュラー)を実装しましょう。

スペキュラー(鏡面反射成分)

鏡面反射成分を実装するには「視線ベクトル」が必要です。そのため最初に作ったコンスタントバッファである変換行列用の構造体に手を加えます。もともとこのような構造体だと思います。

```
struct MatricesData {
```

```
XMMATRIX world;
XMMATRIX viewproj;
};
```

これに「視点座標」を追加しようと思いますが、その場合は float3 つまり cpp 側では XMFLOAT3 になりますので MatricesData という構造体名が少々実態と異なってきます。ということで名称の変更と XMFLOAT3 eye; を追加しましょう。

構造体名は SceneData という事にしておきましょう。VS を使用しているならば右クリックして「名称の変更」を使用すれば楽に名前を変更できると思います。

```
//シェーダ側に渡すための基本的な環境データ
struct SceneData {
    XMMATRIX world;//ワールド行列
    XMMATRIX viewproj;//ビュープロジェクション行列
    XMFLOAT3 eye;//視点座標
};
```

そしてこの eye に視点座標を代入してしまえばいいわけです。代入したらシェーダから見えるようになりますので、

```
//定数バッファ0
cbuffer SceneData : register(b0) {
    matrix world;//ワールド変換行列
    matrix viewproj; //ビュープロジェクション行列
    float3 eye;//視点
};
```

のようにして視点座標を得ます。ここからレイベクトルを作り、ピクセルシェーダに渡せるようにしましょう。頂点シェーダ側で

```
output.ray = normalize(pos.xyz - eye); //視線ベクトル
```

とします。output.ray は「float3 ray:VECTOR;//ベクトル」で受け渡し構造体に追加したものです。セマンティクスが VECTOR ですが、これは予約語ではなくユーザー定義セマンティクスです。こういうのも OK です。

で、レイが得られますのでピクセルシェーダ側で反射ベクトルを作りましょう。ちなみに反射

ベクトルは reflect 関数で、対象のベクトルと法線ベクトルを与えれば作れます。

今回はスペキュラーですが、スペキュラーの式は最初の章で話したように

$$I_s = I_{in} K_s (\cos\theta)^n$$

という物です。で、この $\cos\theta$ はライトと法線ベクトル…ではなく、ライトの反射ベクトルと視線ベクトルの間の角度ですので、視線ベクトル反射ベクトルを R とすると

$$I_s = I_{in} K_s (\hat{R} \cdot \hat{V})^n$$

になります。

光のベクトルはディフューズの時に使った

```
float3 light = normalize(float3(1, -1, 1)); //平行光線ベクトル
```

を用います。あとは、パイプラインからドケツリレーで渡された法線ベクトルとレイベクトルがありますので、それを用いて…

```
//光の反射ベクトル
```

```
float3 refLight = normalize(reflect(light, input.normal.xyz));
float specularB = pow(saturate(dot(refLight, -input.ray)), specular.a);
```

ちなみに `specular.a` には↑の式における n 乗の n にあたります。なお初見の関数と思われる `pow`, `saturate` について解説しますと

- $\text{pow}(x, n) \Rightarrow x^n$ を返す。基本的にシェーダ側で n 乗を表すにはこの関数を使う必要があります。
- $\text{saturate}(x) \Rightarrow x$ の範囲を 0~1 にクランプする $\min(1, \max(0, x))$ と同じだがノーコスト

となっています。`dot` が内積を表しているのは問題ないですね？

さてこの計算結果の `specularB` が鏡面反射成分の輝度となりますので、これにマテリアルから取得したスペキュラ成分の `rgb` を乗算します。分かりやすくするためにスペキュラのみで着色すると…

```
return float4(specularB*specular.rgb, 1);
```



このように鋭い反射が映し出されると思います

このままだとミクさんに見えませんので

```
float4 texColor = tex.Sample(smp, input.uv); //テクスチャカラー  
return diffuse * diffuse * texColor//ディフューズ色  
+ float4(specularB * specular.rgb, 1)//スペキュラー+  
+ float4(ambient*0.5*texColor, 1); //アンビエント(明る過ぎなので0.5かけています)
```

のようになります。まずはディフューズ・スペキュラー・アンビエントを“初音ミク metal.pmd”に適用する

と



このようになります

髪の毛が白いですが“初音ミク metal.pmd”的場合、髪のカラーは sph 側に入ってるで復活させましょう。スフィアマップ乗算(sph)を復活させましょう。

```
sph.Sample(smp, sphereMapUV) //スフィアマップ(乗算)
```

をディフューズに乗算してください。そうすると



このように髪の色が反映されます。ここまでやってきた画素値の計算部分をまとめるとこうなります。

```
float4 BasicPS(Output input) : SV_TARGET{  
    float3 light = normalize(float3(1, -1, 1)); //光の向かうベクトル(平行光線)  
    float3 lightColor = float3(1, 1, 1); //ライトのカラー(1, 1, 1で真っ白)
```

```

//ディフェューズ計算
float diffuseB = dot(-light, input.normal);

//光の反射ベクトル
float3 refLight= normalize(reflect(light, input.normal.xyz));
float specularB = pow(saturate(dot(refLight, -input.ray)), specular.a);

//スフィアマップ用UV
float2 sphereMapUV = input.vnormal.xy;
sphereMapUV = (sphereMapUV + float2(1, -1)) * float2(0.5, -0.5);

float4 texColor = tex.Sample(smp, input.uv); //テクスチャカラー
return diffuseB//輝度
    *diffuse//ディフェューズ色
    *texColor//テクスチャカラー
    *sph.Sample(smp, sphereMapUV)//スフィアマップ(乗算)
    + spa.Sample(smp, sphereMapUV)*texColor//スフィアマップ(加算)
    + float4(specularB*specular.rgb, 1)//スペキュラー
    + float4(texColor*ambient*0.5, 1); //アンビエント(0.5にしてます)
}

```

はい、いっただんここまでスペキュラ(鏡面反射光)もアンビエント(環境光)も終わりとします。
基本が終わりました。

5項： テクスチャファイルが tga や dds だった時の対応

基本モデルを扱ってる間は問題ないと思いますがインターネット上の PMD ファイルの中にはテクスチャが tga や dds の時があり、それに対応していないとテクスチャが張られないようになります。ですから拡張子文字列によって呼び出す関数を切り替えられるようにします。

PMD でよく(たまに)使用される画像ファイルは WIC にも tga や dds があり対応する関数が存在します。このため使用する可能性のある関数を列挙すると

- LoadFromWICFile: bmp や png などの Window のデフォルトで読める基本的な画像ファイル
- LoadFromTGAFile: tga などの一部の 3D ソフトで使用されているテクスチャファイル
- LoadFromDDSFile: DirectX 用の圧縮テクスチャファイル

このような感じになります。

if 文をすらすら書いて場合分けしてもいいのですが、C++も進化していることですし、もう少しだけオシャレな感じで実装したいと思います。今回はラムダ式と STL の map を応用したいと思います。

まずロード用ラムダ式格納のための型を定義します。

```
using LoadLambda_t = function<HRESULT(const wstring& path, TexMetadata*, ScratchImage&)>;
```

これはロード用のラムダ式のための型です。このラムダ式と拡張子文字列のマップを作ります。

```
map < string, LoadLambda_t> loadLambdaTable;
```

それではこの loadLambdaTable に対応表を作っていきましょう。

```
loadLambdaTable[“sph”] = loadLambdaTable [“spa”] = loadLambdaTable [“bmp”] = loadLambdaTable  
[“png”] = loadLambdaTable [“jpg”] = [] (const wstring& path , TexMetadata* meta, ScratchImage&  
img)→HRESULT {  
    return LoadFromWICFile(path.c_str(), 0, meta, img);  
};  
  
loadLambdaTable [“tga”] = [] (const wstring& path, TexMetadata* meta, ScratchImage& img)-  
>HRESULT {  
    return LoadFromTGAFfile(path.c_str(), meta, img);  
};  
  
loadLambdaTable [“dds”] = [] (const wstring& path, TexMetadata* meta, ScratchImage& img)-  
>HRESULT {  
    return LoadFromDDSFfile(path.c_str(), 0, meta, img);  
};
```

多少複雑になってしまったが、やっていることはご理解いただけるでしょうか？単なる関数テーブルでもよかったです。引数の数などが異なりましたので、ラムダ式でワンクションを置いています。ここまで用意したら呼び出しだすが

```
auto wtexpath = GetWideStringFromString(texPath); //テクスチャのファイルパス  
auto ext = GetExtension(texPath); //拡張子を取得  
auto result = loadLambdaTable[ext](wtxpath,  
    &metadata,  
    scratchImg);
```

これで bmp,png,jpg,sph,spa,tga,dds に対応できます。あ、ファイルパス指定がなくて拡張子がない場合はここでエラーが発生してしまいますので、実際には拡張子が既に存在するかのチェックが必要です。そこは各自チェックコードを入れておいてください。これでテクスチャフォーマットが TGA や DDS のファイルに対応できるようになりました。

それではマテリアルの最後の締めくくりとしてトゥーンシェーディングを行いましょう。

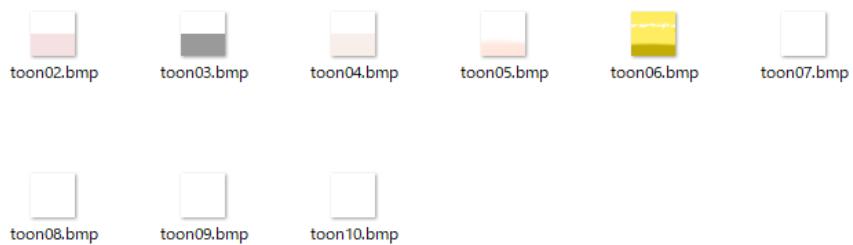
6項：トゥーンシェーディング

トゥーンシェーディング(セルシェーディング)とは

トゥーンシェーディングとは、輝度の変化に手を加える(具体的には段階的に変化する輝度の変化をいじったり着色したりする)ことによってアニメのような見え方にするためのテクニックです。セルシェーディングとも言います。

MMDではカラールックアップテーブル(CLUT)という手法を用いてトゥーンを実現しています。

MMDの中身のDataというフォルダがあり、その中を見てもらうと



このような小さなビットマップデータが並んでいると思います。これがカラールックアップテーブルになります。

これをコピーして、ひとまずプロジェクトフォルダの下に toon というフォルダ作って、そこに入れてください。その前提で話を進めていきます。

この画像を使って何をするのかというと、例えば輝度決定の関数はランダートの余弦則により $\cos \theta$ に比例するので、連続的です。



連続的であるがゆえにこのような見え方になるんですね。正直怖いです。昔のゲームとかでよくあったシェーディングだと思します。これにセルシェード(輪郭線なし)をかけることで

みたいにな見た目にします。

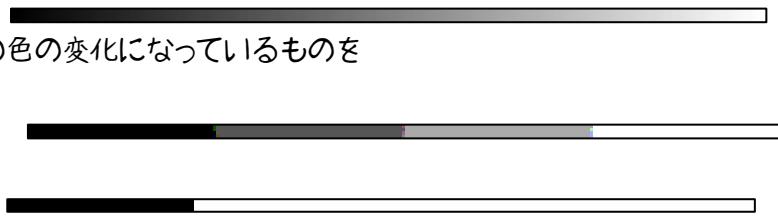
特に日本のアニメのセルっぽいのは色々とありますし、パキッとした色の変化が特徴です。

カラールックアップテーブル(CLUT)の活用

ということで、どうするのかというとランダートの結果

こんな感じの色の変化になっているものを
例えば

とか



のようにしたいと考えます。あくまでも例なので、これ以外にもいろいろありますし、着色もされます。これをやるにはどのようにしたらいいのでしょうか。式(=次方程式など)を作つて変化関数を作つたり、if文で分岐したりしてこれを実現してもいいのですが先ほど見た



このような画像が役に立つのです

どういうことがというとランダートの余弦則の取りうる範囲はせいぜい $0 \sim 1$ です。ということはこの $0 \sim 1$ を↑の画像の V 値に対応させて考えてやればいいわけです。上下がひっくり返っているため疑似コードで表すと

出力色 = $1 - \text{輝度}$

となります。すると例えば↑のようなトゥーンデータを用いれば輝度 0.5 くらいより明るければ真っ白で、 0.5 より暗ければ灰色が輝度値として使用されます。

ちなみにこのカラーレックアップテーブル(CLUT)は先ほども見てもらいましたように全部で 10 個。+ モデル特有の CLUT の指定も可能になっています。

本当は検索の順番としては

マテリアルのトゥーン番号 → (拡張部分の)トゥーンファイル名データ → そのファイルパスをロード

てな感じにすべきなのですが、「拡張部分」なので、最初はトゥーン番号から固定でひとまず toon フォルダからとってきましょう。

と、その前に…トゥーンの指定は各マテリアルで被つてるので、同じテクスチャがロードされてメモリ上に展開されてしまうことになります。これはもったいないです。また、モデルの中にはテクスチャが複数マテリアルに適用されていることもあります。ここは今のうちに効率化を図ろうと思います。こういう同じリソースや処理が被る可能性のある部分に適用するのにちょうどいいのが Flyweight パターンです。

Flyweight パターンを用いたリソースの再利用

パターンなど」というと「クラス図が～」と思うかもしれません、今回のこれはそんなに御大

層なものではありません。

簡単に言うと STL の map を使用して、「ファイルパス」をキーとして、「テクスチャバッファ」を値とするテーブルを作成します。つまり

```
//ファイル名パスとリソースのマップテーブル  
map<string, ID3D12Resource*> _resourceTable;
```

これを見ればだいたい予想できますよね？まず LoadTextureFromFile 関数の頭にこう書きます。

```
ID3D12Resource*  
LoadTextureFromFile(std::string& texPath) {  
    auto it=_resourceTable.find(texPath);  
    if (it != _resourceTable.end()) {  
        //テーブルに内にあったらロードするのではなくマップ内の  
        //リソースを返す  
        return *it;  
    }  
(略)
```

一応解説しますと、今回の find 関数はマップの中からキー文字列(ファイルパス)を探し、要素が存在していれば _resourceTable.end() ではないため、ロード済みと判断しそのイテレータの実態(リソースポインタ)を返します。

では、そうでなかつた場合…つまりロード処理が終わったときにどうすべきかというと、これは簡単で LoadFileFromFile 関数の最後でリソースを返す前にマップテーブルへ登録するだけです。

```
(略)  
_resourceTable[texPath] = texbuff;  
return texbuff;  
}
```

これにより次に同じパスの画像ファイルをロードしようとしたときに、再びロードするのではなく既にロード済みのバッファを返すのです。これも広義の Flyweight パターンとみなせます。

まずはロード関数をこれに変更したうえでまともに動くかどうかを確認してください。正常に動いたらトゥーンのロードに移りましょう。

トゥーンインデックスからのトゥーンのロード

前にも書きましたが、トゥーンのロードについては本当は拡張部分を見に行って決めるのですが、「拡張」なので、拡張がなかった時代のやり方でまずはやってみましょう。

```
インデックス 0xff:toon00.bmp  
インデックス 0x00:toon01.bmp  
インデックス 0x01:toon02.bmp  
:  
:
```

というルールになっているため、インデックスを toon00.bmp に変換する処理が必要になります。stringstream を使ってもいいのですが、今回のパターンでは固定長ですし、オーバーダックスに sprintf を使用したいと思います。

例えばこう宣言しておいて

```
char toonFileName[16];
```

これに sprintf でインデックス(仮に toonIdx とする)をもとに文字列を作るには

```
sprintf(toonFileName, "toon%02d.bmp", toonIdx + 1);
```

とします。ではこの文字列をもとにプロジェクトの直下の toon フォルダ(この頃の最初に作ってトゥーン画像をコピーしましたね?していない人はしておきましょう。)から取得するようにしましょう。

(なお、SDL をオンにしていると sprintf はコンパイルエラーを起こしますので、その場合は SDL をオフにするか sprintf_s をご使用ください)

トゥーンはマテリアルに必ず入っているのでマテリアルループの先頭で

```
for (int i = 0; i < pmdMaterials.size(); ++i) {  
    // トゥーンリソースの読み込み  
    string toonFilePath = "toon/";  
    char toonFileName[16];  
    sprintf(toonFileName, "toon%02d.bmp", pmdMaterials[i].toonIdx + 1);  
    toonFilePath += toonFileName;  
    toonResources[i] = LoadTextureFromFile(toonFilePath);
```

(中略)

のようにして読み込みます。

デフォルトグラデーションテクスチャの作成

基本的にはロードできていますが、ロードが失敗した場合には nullptr が入っています。もし
かしたらトゥーンを使ってほしくないマテリアルがあって、そこには無効な数値が入っている
場合もあるため以下のようなテクスチャを作ります。

```
// トゥーンのためのグラデーションテクスチャ
ID3D12Resource* CreateGrayGradationTexture() {
    D3D12_RESOURCE_DESC resDesc = {};
    resDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    resDesc.Width = 4; // 帯
    resDesc.Height = 256; // 高さ
    (略)
    // 上が白くて下が黒いテクスチャデータを作成
    std::vector<unsigned int> data(4 * 256);
    auto it = data.begin();
    unsigned int c=0xff;
    for (; it != data.end(); it+=4) {
        auto col = (c << 0xff) | (c << 16) | (c << 8) | c;
        std::fill(it, it+4, col);
        --c;
    }
    result = gradBuff->WriteToSubresource(0,
                                           nullptr,
                                           data.data(),
                                           4 * sizeof(unsigned int),
                                           sizeof(unsigned int)*data.size());
    return gradBuff;
}
```

で、このテクスチャを適当な変数に入れておきます。

```
auto gradTex = CreateGrayGradationTexture();
```

デスクリプタヒープのサイズを増やしてビューを作成

この上でビューを作りますが、これは spa の直後に入れますので、デスクリプタヒープのサイズをまた一つ増やして、spa の後ろでビューを作成してください。

```
if (toonResources[i] == nullptr) {
    srvDesc.Format = gradTex->GetDesc().Format;
    _dev->CreateShaderResourceView(gradTex, &srvDesc, matDescHeapH);
}
else {
    srvDesc.Format = toonResources[i]->GetDesc().Format;
    _dev->CreateShaderResourceView(toonResources[i], &srvDesc, matDescHeapH);
}
matDescHeapH.ptr += incSize;
```

こんな感じですね。トゥーン読み込みに成功してたらそのまま割り当てて、トゥーンが nullptr だったら gradTex のほうを使用します。

デスクリプタヒープのサイズ増やすのを忘れないでくださいね？

レンジの数を増やしてトゥーンに対応

descTblRange[2].NumDescriptors = 4; //テクスチャ4つ(基本と sph と spa とトゥーン)
なんとこれで終わりです。

描画時のデスクリプタヒープのポインタの進め方を増やす

```
auto cbvSrvIncSize = _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV)*5;
for (auto& m : materials) {
    _cmdList->SetGraphicsRootDescriptorTable(1, materialH);
    _cmdList->DrawIndexedInstanced(m.indicesNum, 1, idxOffset, 0, 0);
    materialH.ptr += cbvSrvIncSize;
    idxOffset += m.indicesNum;
}
```

cpp 側はこれでいいので、hlsl 側に行きましょう。

HLSL を変更してトゥーンに対応

まずはトゥーンテクスチャを受け取るための変数を用意します。

```
Texture2D<float4> toon:register(t3); //3番スロットに設定されたテクスチャ(トゥーン)
```

次はピクセルシェーダでこの toon を利用する部分ですが前にも書いた通り、輝度をテクスチャの V 座標に置き換えます。ただしここで注意点ですが、トゥーンのテクスチャは図のように上が明るくしたが暗い画像になっています。



このため、そのまま輝度値 ⇒ V 座標にしてしまうと、輝度=0 のとき最も明るくなってしまい反転したような見た目になってしまいます。HLSL で値を反転するのは簡単で、1-value とすればいいのです。

ということで輝度値計算の部分にこう書きます。

```
//ディフューズ計算  
float diffuseB = saturate(dot(-light, input.normal));  
float4 toonDif = toon.Sample(smp, float2(0, 1.0 - diffuseB));
```

さて、ここで出てきた toonDif を、ピクセル値計算時の輝度の代わりに使用します。

```
return saturate(toonDif // 輝度(トゥーン)  
    * diffuse // ディフューズ色  
    * texColor // テクスチャカラー  
    * sph.Sample(smp, sphereMapUV) // スフィアマップ(乗算)  
    + saturate(spa.Sample(smp, sphereMapUV) * texColor) // スフィアマップ(加算)  
    + float4(specularB * specular.rgb, 1) // スペキュラー  
    + float4(texColor * ambient * 0.5, 1) // アンビエント
```

ここまで特にバグなく実装出来れば…



おお、トゥーンになった…？

ちょっとおかしいですよね？

暗くなるはずのほっぺたの下部分が急に明るくなっていますね？これはどういうことでしょう？

トゥーン用にサンプラーを作る

よく結果を観察してください。

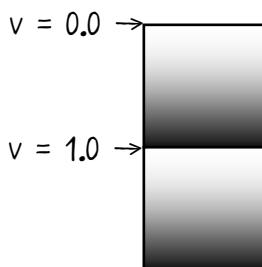
ほっぺやあごの下が妙に明るくなっています。ということはディフューズの結果=0の時が怪しいですねえ…？

もし輝度が0だったと仮定すると

$$v \text{ 座標} = 1.0f - \text{輝度値};$$

ですから v 座標=1.0となります。そななるとどうなりますかね…？真っ暗？真っ白？

これは現在のサンプラーのテクスチャアドレッシングモードが WRAP であるため 1.0 の時に



御覧のように今のままで 1.0 のときに真っ黒ではなく真っ白になる可能性があります。ということでトゥーン用に WRAP でなく CLAMP のサンプラーを作る必要があります。という事でノンシグネチャのサンプラーを書き換えます。

```
D3D12_STATIC_SAMPLER_DESC samplerDesc[2] = {};  
  
samplerDesc[0].AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP; // 横繰り返し  
samplerDesc[0].AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP; // 縦繰り返し  
samplerDesc[0].AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP; // 奥行繰り返し  
samplerDesc[0].BorderColor = D3D12_STATIC_BORDER_COLOR_TRANSPARENT_BLACK; //  
samplerDesc[0].Filter = D3D12_FILTER_MIN_MAG_MIP_POINT; // 補間しない(ニアレストネイバー)  
samplerDesc[0].MaxLOD = D3D12_FLOAT32_MAX; // ミップマップ最大値  
samplerDesc[0].MinLOD = 0.0f; // ミップマップ最小値  
samplerDesc[0].ComparisonFunc = D3D12_COMPARISON_FUNC_NEVER; // samplerDesc[0].ShaderVisibility  
= D3D12_SHADER_VISIBILITY_PIXEL; // ピクセルシェーダ可視  
samplerDesc[0].ShaderRegister = 0; // シェーダースロット番号を忘れないように  
samplerDesc[1] = samplerDesc[0]; // 変更点以外をコピー  
  
samplerDesc[1].AddressU = D3D12_TEXTURE_ADDRESS_MODE_CLAMP; // 繰り返さない  
samplerDesc[1].AddressV = D3D12_TEXTURE_ADDRESS_MODE_CLAMP; // 繰り返さない  
samplerDesc[1].AddressW = D3D12_TEXTURE_ADDRESS_MODE_CLAMP; // 繰り返さない  
samplerDesc[1].ShaderRegister = 1; // シェーダースロット番号を忘れないように
```

```
rootSignatureDesc.pStaticSamplers = samplerDesc;  
rootSignatureDesc.NumStaticSamplers = 2; //数の変更を書いてください。
```

HLSL 側でトゥーンサンプラーの対応

これで 0 番と 1 番のサンプラーを用意して 1 番を CLAMP にしてますので、これをトゥーン用のサンプラーとしましょう。

```
SamplerState smp:register(s0); //0番スロットに設定されたサンプラー  
SamplerState smpToon:register(s1); //1 番スロットに設定されたサンプラー(トゥーン用)
```

あとはトゥーンの UV 座標決定部分のサンプラーをこれに入れ替えるだけです。

```
float4 toonDif = toon.Sample(smpToon, float2(0, 1.0 - diffuseB));
```

これでやっと正しいトゥーン表示となります。



とりあえず基本の見た目はこれでいいでしょう。

さてこれでただの基本的な表示までという事であれば、目的とするところまでは来れました。

リファクタリング

ここからはアニメーションに進むのですが、ここまでプログラミングをしてくるとかなりゴチャゴチャしているかなと思います。

正直説明するほうもこのままではしんどくなってしまった。というわけで「リファクタリング」を行います。実はここまで説明のためのサンプルプログラムは main.cpp 一本でやってきましたが、説明している僕のコードが 1200 行を超えており、メンテナンス的にも危険な状態になっています。

プログラミングに慣れている読者は既にリファクタリングどころか適切な設計までやっている事と思いますので、そういう読者はこの章を読み飛ばしていただいて結構です。第3部へ移動してスキンメッシュアニメーションの実装に進んでください。

便利なクラスや構造体やマクロを積極的に使用する

クラス設計から直したいところですが、その前に便利なクラスや構造体やマクロを積極的に使用することで、コードの見通しをよくしましょう。

それぞれの構造体の紹介

- CD3DX12_HEAP_PROPERTIES : GPU リソースを作る時のヒーププロパティのヘルパー
- CD3DX12_RESOURCE_DESC : GPU リソースを作る時のリソース設定のヘルパー
- CD3DX12_RESOURCE_BARRIER : バリア設定のヘルパー

などは既に第 6 章の最後で紹介してお勧めしておりますので、お使いになられている方も多いと思いますが、それ以外の便利なものについてもご紹介いたします。

まず、パイプライン周りから

- CD3DX12_RASTERIZER_DESC : 面倒なラスタライザ系を便利に設定できる構造体
- CD3DX12_BLEND_DESC : 面倒なブレンド周りを便利に設定できる構造体
- CD3DX12_SHADER_BYTCODE : シェーダ周囲を便利に設定できる構造体

ルートシグネチャ回り

- CD3DX12_DESCRIPTOR_RANGE : デスクリプタレンジを便利に初期化できる構造体
- CD3DX12_ROOT_PARAMETER : ルートパラメータを便利に初期化できる構造体
- CD3DX12_STATIC_SAMPLER_DESC : スタティックサンプラーを便利に初期化できる構造体
- CD3DX12_ROOT_SIGNATURE_DESC : ルートシグネチャ設定を便利に初期化できる構造体

それ以外

- CD3DX12_VIEWPORT : ビューポート設定を便利に初期化できる構造体
- CD3DX12_RECT : 矩形を便利に初期化できる構造体
- CD3DX12_CLEAR_VALUE : クリア値を便利に初期化できる構造体
- CD3DX12_CPU_DESCRIPTOR_HANDLE : CPU デスクリプタハンドルを便利に操作
- CD3DX12_GPU_DESCRIPTOR_HANDLE : GPU デスクリプタハンドルを便利に操作

といった具合です。多いですね。

なんで最初から教えてくれなかつたんだ!とお怒りの声が聞こえてきそうですが、中身がどうなってるのかを知らないままに使い続けるのは怖いものです。

初心者の段階だからこそ中身を意識して使うべきだと僕は思います。ていうか僕自身の感覚と経験からいいうといきなり CD3DX 見せられてもブラックボックスみたいで理解できなくて気持ち悪かったです。

ComPtr<T>テンプレートクラス(正確には WRL::ComPtr)

この ComPtr<T>テンプレートクラスは、DirectX12 のオブジェクトに対して特に有効なものです。働き自体は STL の shared_ptr と同じなのですが、解放されるときの挙動が違います。

shared_ptr ならば、そのポインタを参照している関数がすべてなくなつたときに対象のアド

レスに対して delete を行い、メモリの解放を行います。

ところが DirectX12 の Create○○ 等で作られるオブジェクトというのは、CPU メモリだけでなく GPU メモリも確保していたりするため、解放時には delete ではなく明示的に Release 関数を呼び出す必要があります。

では、通常の shared_ptr ではこれが呼び出されないため、例えば CreateCommittedResource などに shared_ptr を用いるのは不適切です。

ではどうするのかというと、この ComPtr を使用します。この ComPtr は参照数がゼロになった場合に Release() 関数を呼び出してくれるので、shared_ptr ならば delete で開放されるタイミングでオブジェクトの Release() 関数が呼び出されるため、適切に解放処理を行ってくれるというわけです。

便利なマクロを使おう

マクロって言っても既に使ってると思いますが、HRESULT の評価をするときの

- SUCCEEDED(result): 成功してたら true を返す
- FAILED(result): 失敗してたら true を返す

あと、使うかどうかわかりませんが、DirectXMath のマクロにこういうのもあります。

- XMConvertToDegrees : 弧度法を度数法に変換
- XMConvertToRadians : 度数法を弧度法に変換

実際に CD3DX12～構造体を使ってリファクタリング

ここまで話した構造体を使うと、例えば

```
D3D12_RESOURCE_DESC resDesc = {};  
resDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;  
resDesc.Width = 4; //幅  
resDesc.Height = 256; //高さ  
resDesc.DepthOrArraySize = 1;  
resDesc.SampleDesc.Count = 1;  
resDesc.SampleDesc.Quality = 0; //  
resDesc.MipLevels = 1; //  
resDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;  
resDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN; //レイアウトについては決定しない  
resDesc.Flags = D3D12_RESOURCE_FLAG_NONE; //とくにフラグなし  
↑こういうコードが…  
auto resDesc = CD3DX12_RESOURCE_DESC::Tex2D(DXGI_FORMAT_R8G8B8A8_UNORM, 4, 256);  
こうなります。すごい!!!
```

ちなみにヒーププロパティもCD3DX12を使えばある程度小さくなるんですが、RESOURCE_DESCほど劇的ではないです。これがDEFAULTやUPLOADなら短いんですが、CUSTOMの場合いちいち指定しなければならないですからね。

ちなみに

```
D3D12_HEAP_PROPERTIES texHeapProp = {};  
texHeapProp.Type = D3D12_HEAP_TYPE_CUSTOM;//特殊な設定なのでdefaultでもuploadでもなく  
texHeapProp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_WRITE_BACK;//ライトバックで  
texHeapProp.MemoryPoolPreference = D3D12_MEMORY_POOL_L0;//転送がL0つまりCPU側から直で  
texHeapProp.CreationNodeMask = 0;//單一アダプタのため0  
texHeapProp.VisibleNodeMask = 0;//單一アダプタのため0  
↑これが  
auto texHeapProp = CD3DX12_HEAP_PROPERTIES(D3D12_CPU_PAGE_PROPERTY_WRITE_BACK,  
                                            D3D12_MEMORY_POOL_L0);
```

こうなります。短くなつていいのですが、最初からこれ使われてしまつたら「何やってるのかわからぬ!」ですよね?ほぼブラックボックスのように見えてしまいます。そう思ったので最初は使用をためらつていたのです。

ちなみにロードテクスチャ用のバッファに関してはミップ数も指定する必要がありますのでそこは

```
auto texHeapProp = CD3DX12_HEAP_PROPERTIES(D3D12_CPU_PAGE_PROPERTY_WRITE_BACK,  
                                            D3D12_MEMORY_POOL_L0);  
auto resDesc = CD3DX12_RESOURCE_DESC::Tex2D(metadata.format,  
                                             metadata.width, metadata.height,  
                                             metadata.arraySize, metadata.mipLevels);
```

といった具合になります。だいぶ短くはなりますね。

ただ、例えばデプスステンシルのためのバッファに関しては…

```
D3D12_RESOURCE_DESC depthResDesc = {};  
depthResDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;//  
depthResDesc.Width = window_width;//  
depthResDesc.Height = window_height;//  
depthResDesc.DepthOrArraySize = 1;//  
depthResDesc.Format=DXGI_FORMAT_D32_FLOAT;//  
depthResDesc.SampleDesc.Count = 1;//  
depthResDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;//  
depthResDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
```

```
depthResDesc.MipLevels = 1;  
↑これが  
↓こうなるので  
auto depthResDesc = CD3DX12_RESOURCE_DESC::Tex2D(DXGI_FORMAT_D32_FLOAT,  
                                                 window_width, window_height,  
                                                 1, 1, 1, 0,  
                                                 D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL);  
確かに短くはなりましたが、却ってパラメータの意味するところが分かりづらくなっている  
気がするため、こういう物に関しては使わないほうがいいと思います。短くするのが目的では  
ありませんしね。
```

ここからはいちいち比較しませんが、変更があった部分をそれぞれ示していきます。まず、パイ
プラインステート内のラスタライザの部分は…

```
gpipeline.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);  
gpipeline.RasterizerState.CullMode = D3D12_CULL_MODE_NONE; // カリングしない
```

こうなります。D3D12_DEFAULT で基本設定になるのですが、その場合は背面カリングが ON に
なってしまうので、MMD 仕様に合わせてカリング OFF にしています。元のコードを見れば分か
りますが、かなりコード量が減りますので、ここは使用すべきだと思います。
同様にブレンドステートの部分ですが…

```
gpipeline.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
```

1 行で済んでしまいます。もしブレンドステートに変更があれば↑の BlendState に追加で設
定していけばいいだけですね。次にシェーダを登録する部分ですが

```
gpipeline.VS = CD3DX12_SHADER_BYTECODE(_vsBlob);  
gpipeline.PS = CD3DX12_SHADER_BYTECODE(_psBlob);
```

これはそれぞれ 1 行減っただけですね。

CD3DX12 をレンジヤルートパラメータ周りに使用すると
//レンジ
CD3DX12_DESCRIPTOR_RANGE descTb1Range[3] = {};//テクスチャと定数の 2つ
descTb1Range[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);//定数[b0] (座標変換用)

```

descTblRange[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 1); //定数[b0] (マテリアル用)
descTblRange[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 4, 0); //テクスチャ4つ
//ルートパラメータ
CD3DX12_ROOT_PARAMETER rootparam[2] = {};
rootparam[0].InitAsDescriptorTable(1, &descTblRange[0]); //座標変換
rootparam[1].InitAsDescriptorTable(2, &descTblRange[1]); //マテリアル周り

```

結構短くなりましたね。サンプラ部分なんかは

```

CD3DX12_STATIC_SAMPLER_DESC samplerDesc[2] = {};
samplerDesc[0].Init(0);
samplerDesc[1].Init(1, D3D12_FILTER_ANISOTROPIC,
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, D3D12_TEXTURE_ADDRESS_MODE_CLAMP);

```

こうなります。

レンジルートパラメータ、サンプラ周りは使ったほうがいいですね。でも…ルートシグネチャ設定は使ったありがたみがそんなにないです↓

```

CD3DX12_ROOT_SIGNATURE_DESC rootSignatureDesc = {};
rootSignatureDesc.Init(2, rootParams,
    2, samplerDescs,
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
行数はあまり減りませので、可読性と天秤にかけるとどうかなーって感じですね。ここは使うか使わないかは読者の判断にお任せです。

```

その他の CD3DX12～です。

CD3DX12_VIEWPORT はちょっと面白いです。第一引数にバック/dffアのリソースポインタを渡せば、その内容から幅と高さを計算してくれるので、

```
CD3DX12_VIEWPORT viewport(_backBuffers[0]);
```

で済みます。

ただしちょっと変わったビューポートを作ろうと思ったら細かく指定しなければならないので、そういう場合にはあまりありがたみはないです。

次にシザーリングのための矩形に CD3DX12_RECT を使う例ですが全然ありがたくないです。

```
CD3DX12_RECT scissorRect(0, 0, window_width, window_height);
```

↑横に並んだだけやないか!!

CD3DX12_CLEAR_VALUE ですが、今のところクリア値を設定するのが深度/dffアのみなので、

```
CD3DX12_CLEAR_VALUE depthClearValue(DXGI_FORMAT_D32_FLOAT, 1.0f, 0);
```

こんな感じになります。

もしカラーのクリア値なら

```
float clrColor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

```
CD3DX12_CLEAR_VALUE rtClearValue(DXGI_FORMAT_R8G8B8A8_UINT, clrColor);
```

こうなりますが、こっちはあまりありがたくないし、あまり使う機会はないかと思います。

最後に

```
CD3DX12_CPU_DESCRIPTOR_HANDLE
```

と

```
CD3DX12_GPU_DESCRIPTOR_HANDLE
```

ですが、これは個人的にはそれほどうれしくないです。ご覧いただければ分かると思いますが、

```
CD3DX12_CPU_DESCRIPTOR_HANDLE
```

```
matDescHeapH(materialDescHeap->GetCPUDescriptorHandleForHeapStart());
```

初期化がこんな形になります。そして得られるものといえば Offset 関数くらいです。

```
matDescHeapH.Offset(incSize);
```

関数の作用はご想像の通り、ptr+=offset です。あまりありがたくない上に意味が伝わりづらくなる気がするので、僕はこれを使いません。使うかどうかは読者の判断にお任せいたします。

用途によって簡単な分類を行う

簡単に言うと PMD 特有の部分と、アプリケーション内で共通する処理の部分を分けてしまおうという事です。

概要

こういう時には、共通部分と特有の部分を箇条書きにして分けていくといいと思います。最終的な結果まで見据えると難しくなるので、現状を分かりやすく分類すると思ってください。

PMD 特有

- PMD データロード部分
- ワールド行列まわり
- 頂点とインデックスまわり
- PMD 用デスクリプタヒープ(マテリアル、テクスチャ)
- PMD 用パイプライン(ルートシグネチャ、シェーダなど)

- PMD 用描画部分まわり
- スフィアマップやトーンテクスチャまわり

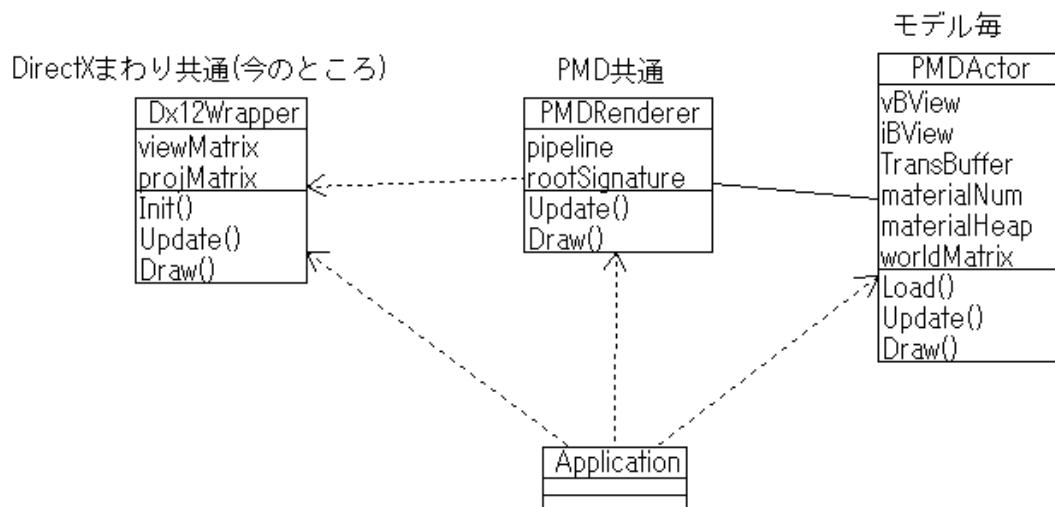
共通部分

- 初期化まわり(ウィンドウ、DXGI、DirectX12 等)
- レンダーターゲット、深度/バッファまわり
- ビュー&プロジェクションまわり
- テクスチャ/バッファ生成まわり
- 共通テクスチャまわり
- メインループ
- 後処理まわり

ひとまずはこんなところでしょう。

PMD 周辺についてもモデルを数体表示することを想定するとパイプライン周りは共有化しておいて、それ以外の部分(バッファ作成部分)は共有しないとかいろいろありますので、いっぽんはそこまでクラスの構成を考えてみましょう。

大雑把にこのような感じにします。



今回は PMD の個別部分と共有部分を分けています。

で、分けたときにもしかしたらプログラミングの罠に直面する人が居るかもしれませんので、特に危ない部分を解説します。

XMMATRIX の罠

まず、**PMDActor** に `transBuffer` と書いていますが、これは座標変換のためのバッファだと思つ

てください。現時点では world 変換が入っているとします。

そして、PMDActor オブジェクトは new されて生成されるものとします。ここで問題が発生します。

え? どういう事なの? どういうことなのかを解説する前に DirectXMath の XMMATRIX の型の宣言を見てみましょう。ちょっと見づらいですが、このように定義されています。

```
_declspec(alignment(16)) struct XMMATRIX
{
    XMVECTOR r[4];
    (略)
};
```

さて、ここで注目すべきは _declspec(alignment(16)) の部分です。ここがかなりの曲者なのです。これは前述の SIMD 演算の効率性のために「16 バイトアライメント」を強制するものです。ちなみに

ローカルのスタック変数として使用する分にはコンパイラが↑のキーワードよりアライメントを尊重して配置してくれますので問題は起きません。が、XMMATRIX をヒープに確保するか、もしくは XMMATRIX をメンバに持つ構造体やクラスを new(つまり動的メモリ確保)する場合に問題が発生します。

new などの動的メモリ確保はそのままでは、ヒープメモリの特定のアドレスにメモリを確保します。これは動作自体が malloc と同じだと思ってください。もちろんこの時のメモリ確保のアライメントは 32bit ならば 4 バイトアライメント、64bit なら 8 バイトアライメントとれます…ん?

あれれ? それってヤバくない? 16 バイトアライメントが強制されている構造体はそれを期待しているのに 4 バイト境界、8 バイト境界にメモリ確保されたらお約束を守ってないとになり、動作がおかしくなったりクラッシュしたりするかもしれない(僕の環境では実際にきました)

乗算などの演算を行うまでは特に問題にならないのですが、このバイト境界がずれたまま乗算などの演算を行うとバグることがあります。

という事で、動的確保をするときもバイト境界を指定できるものを使用します。用意されて

いるものとしては_aligned_malloc(確保バイト数, バイト境界)という関数があります。
なんとなくお分かりかと思いますがこの関数は第二引数で指定されたバイト境界の位置に
動的確保を行います。ところがnewを使いたいときはどうすればいいのでしょうか?

例えば、対象の XMMATRIX をラップするような構造体を作り、そいつの new をオーバーロード
します。

```
struct Transform {
    //内部に持つてるXMMATRIXメンバが16バイトアライメントであるため
    //Transformをnewする際には16バイト境界に確保する
    void* operator new(size_t size);
    DirectX::XMMATRIX world;
};

void*
PMDActor::Transform::operator new(size_t size) {
    return _aligned_malloc(size, 16);
}
```

もしくは XMMATRIX を new する際に placement new を行います。

```
auto p = _aligned_malloc(sizeof(XMMATRIX), 16);
auto m = new(p) XMMATRIX;
```

あとは、単なる行列の入れ物として使う用途であるならば XMMATRIX である必要はなく、
XMFLOAT4x4 を用いればいいです。中身は XMMATRIX と同じですが、SIMD に最適化されてないため、アライメントの問題は起こりません(ただし SIMD の恩恵は受けられません)。

なので、乗算等の計算は XMMATRIX で行い、GPU に渡す時の入れ物は XMFLOAT4x4 とすればいい
かなと思います。

今のところ筆者のプログラムでは問題は起きておりませんが、作り方や環境によっては↑の
問題が起きると思いますので、解説しておきました。

結果としての Application.cpp

リファクタリング後の全ソースコードを書くわけにもいきませんので整理後の
Application.cpp の内容を記載します。

```
#include "Application.h"
```

```

#include "Dx12Wrapper.h"
#include "PMDRenderer.h"
#include "PMDActor.h"

//ウィンドウ定数
const unsigned int window_width = 1280;
const unsigned int window_height = 720;

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

void
Application::CreateGameWindow(HWND &hwnd, WNDCLASSEX &windowClass) {
    HINSTANCE hInst = GetModuleHandle(nullptr);
    // ウィンドウクラス生成&登録
    windowClass.cbSize = sizeof(WNDCLASSEX);
    windowClass.lpfnWndProc = (WNDPROC)WindowProcedure; // コールバック関数の指定
    windowClass.lpszClassName = _T("DirectXTest"); // アプリケーションクラス名(適当でいいです)
    windowClass.hInstance = GetModuleHandle(0); // ハンドルの取得
    RegisterClassEx(&windowClass); // 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に预告する)

    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
    // ウィンドウオブジェクトの生成
    hwnd = CreateWindow(windowClass.lpszClassName, // クラス名指定
        _T("DX12リファクタリング"), // 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
windowClass.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

}

SIZE
Application::GetWindowSize()const {
    SIZE ret;
    ret.cx = window_width;
    ret.cy = window_height;
    return ret;
}

void
Application::Run() {
    ShowWindow(_hwnd, SW_SHOW);//ウィンドウ表示
    float angle = 0.0f;
    MSG msg = {};
    unsigned int frame = 0;
    while (true) {
        if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        //もうアプリケーションが終わるって時にmessageがWM_QUITになる
        if (msg.message == WM_QUIT) {
            break;
        }

        //全体の描画準備
        _dx12->BeginDraw();
    }
}
```

```

//PMD 用の描画パイプラインに合わせる
_dx12->CommandList()->SetPipelineState(_pmdRenderer->GetPipelineState());
//ルートシグネチャも PMD 用に合わせる
_dx12->CommandList()->SetGraphicsRootSignature(_pmdRenderer->GetRootSignature());

_dx12->CommandList()->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

_dx12->SetScene();

_pmdActor->Update();
_pmdActor->Draw();

_dx12->EndDraw();

//フレップ
_dx12->Swapchain()->Present(1, 0);
}

}

bool
Application::Init() {
    auto result = CoInitializeEx(0, COINIT_MULTITHREADED);
    CreateGameWindow(_hwnd, _windowClass);

    //DirectX12 ラッパー生成&初期化
    _dx12.reset(new Dx12Wrapper(_hwnd));
    _pmdRenderer.reset(new PMDRenderer(*_dx12));
    _pmdActor.reset(new PMDActor("Model/初音ミク.pmd", *_pmdRenderer));

    return true;
}

void
Application::Terminate() {
    //もうクラス使わんから登録解除してや
}

```

```
UnregisterClass(_windowClass.lpszClassName, _windowClass.hInstance);  
}
```

```
Application&  
Application::Instance() {  
    static Application instance;  
    return instance;  
}
```

ポージングしようゼエ…(いいズエ…)



何気ないポージングが…↑ ポーズ(A ポーズ)先生を傷つけた…。
ついにきましたポージング。まあ今までの話も十分ややこしかったけど、ここからはアルゴリズム&数学的にややこしい。

概要

ボーンとはいったい…

ボーンと言うのは「骨」です。Blender で言うとアーマチュアです。ボーンと言う名前は本当に良くつけたなと思います。割とイメージ通り。ただ、太さの無い骨…いや、「実体のない骨」、「計算上の骨」とでも思っておいてください。MMD でいうと



上の図における②の集合体の事です。言うたら関節の座標を起点としたベクトルの集合体という事です。

実際に MMD のボーンを回転させれば分かりますが、回転操作をすると関節を起点としてそこから先のパートが回転します。

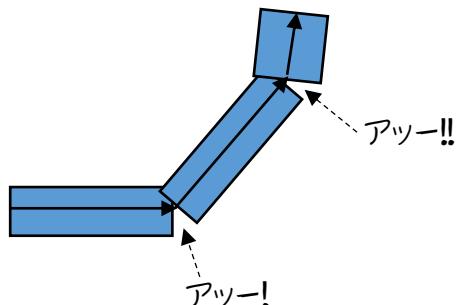
ただ、ボーンのしくみだけでは肉と骨が分離してしまいますので、ここでもう一つの仕組みが必要なのです。

要はボーンに肉を運動させるそういう仕組みです。

スキニング(スキンメッシュアニメーション)

肉を骨に運動させることをスキニングと言います。スニーキングではないので間違えないようにお願いします。

まあ、実は昔のゲームでは肉は運動してたけど「皮膚が伸びない」という状況だったんで

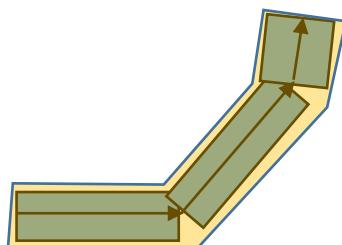


こういう状況になっていました。バーチャファイター2くらいまではこれだったんですが、職人さんが切れ目をみせないよう工夫してたので気になりませんでしたが…今は本当にびっくりするくらいに人間のように皮膚がくっついてきていますよね？



今では当然のように見えますけど、「スキニング」というのがない時代はロボットみたいになつたんですわ。

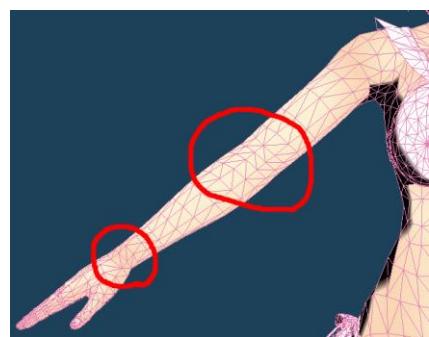
で、今ではさっきの切れ目がある図の重なっている部分および分離している部分をイイ感じに補正(重みづけ乗算)して



こんな感じにして切れ目をなくしています。当然ながらカクカクしてしまうので実際にはモーテリングの時点で関節部分のポリゴン分割数を多めにして、破綻が出にくくないようにしています。蛇腹ストロー(曲がるストロー)と同じ原理ですね。



蛇腹(分割数が多い)になっているからスムーズに曲がる



実際にモデルを見ても関節近辺で分割数増やしてるのがわかります
んで、この関節曲げをスムーズに見せるために、「ウェイト」というのを各頂点に設定しどのボーンからどのくらいの影響を受けるのかを全頂点に対して設定しておきます。

その関節に近い部分は2つ以上のボーンの影響を受けることになるわけです。

ちなみにそのデータが、頂点データにおけるボーンウェイトに当たります。

vertex[36].pos[0]	3FA6978E 4073E5C8 3EC85879
vertex[36].normal_vec[0]	3E9F0401 3E65D180 BF6C75D7
vertex[36].uv[0]	00000000 3F800000
vertex[36].bone_num[0]	0010 000F
vertex[36].bone_weight	46
vertex[36].edge_flag	00
vertex[37].pos[0]	3FBE872A 4093999A 3F19B3D1
vertex[37].normal_vec[0]	3EB88A54 3E0FC2D7 BF6C12B7
vertex[37].uv[0]	00000000 3F800000
vertex[37].bone_num[0]	0010 000F
vertex[37].bone_weight	28
vertex[37].edge_flag	00
vertex[38].pos[0]	3FB06F6A 409449BA 3F281062
vertex[38].normal_vec[0]	BFF02E3A 3E992F2E BF4FF0D9
vertex[38].uv[0]	00000000 3F800000
vertex[38].bone_num[0]	0010 000F
vertex[38].bone_weight	28

例えば37番頂点は0x10番ボーンと0x0f番ボーンから影響を受けるわけです。で、その割合がbone_weightで定義されており、100のウェイトをどう分けるかと言う計算になっています。

37番頂点なら

16番ボーン*40/100+15番ボーン*60/100と言った具合になっています。

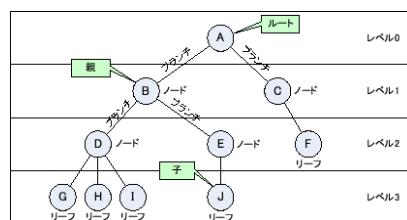
またここもけつたいは作りでfloatで設定して、+と-にすりゃレル1に変にけちりやがつて1バイトにして、100分率でやっています。

…その反動がPMXでは異様なくらいややこしいんだけどね…まあともかくそういうの頂点を動かすことによっていろいろな形にポージングしてくれるわけだ。

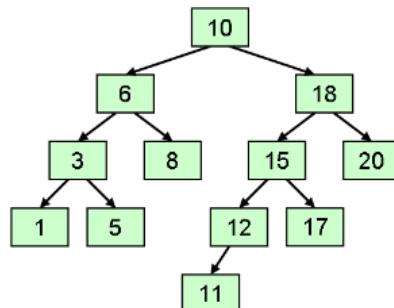
さて、これを実装していく前にちょっとしたアルゴリズムとデータ構造を知っておいてほしい。ツリー構造と再帰実行だ。

ツリー構造と再帰

普通に「アルゴリズムとデータ構造」だから基本情報を通ってる人は知っていて当たり前なんですかねえ…？



図うへくらいは見たことあるでしょ？二分木なら見たことあるかな？バイナリソートで見たことあるよなあ？



まあ、いずれにせよ木構造(ツリー構造)ってのはな？

- 親子構成になっていること
- 親1人に対して子供複数(兄弟がいる)
- もちろん親→子→孫→ひ孫→と言う具合に何階層にもなっている
- 親子はなんかしらのリンクで繋がっている(繋がり方は問わない)
- 親は子を知る方法がある。子が親を知ってるかどうかは任意(場合による)

こういう構造の事です。親と子のつながりが「1対1以上」になってしまったりやツリー構造。もし「1対1」ならそりやリンクリストって話よ。

次にツリー構造をなめていく事を「トラバースする」って言うんですが、この時にただの登録

順にやっていくんじゃアリストとか配列と変わらない。ツリー構造のアドバイantageを活かすのに適しているのが再帰である。

再帰ってのは非常に簡単な話で関数 Function があったとする。

そしてその関数は関数内の処理において自分自身を呼び出す。

```
void Function(){  
    Function();  
}
```

当たり前の話ですが、↑の状況だと「無限ループ」と同じになって帰ってきません。なお、while や for の無限ループと違ってそのうち「スタックオーバーフロー」を起こします。



であるため再帰処理を行う際には「継続条件(or 終了条件)」が必ずどこかで成立するように気を遣わないといけません。例えばこうします。

```
void Function(int a,int b) {  
    std::cout << a << std::endl;  
    if(a+b<100)//継続条件  
        Function(b,a+b);  
}
```

終了条件ならば

```
void Function(int a,int b) {  
    if (b > 100) return;//終了条件  
    std::cout << a << std::endl;  
    Function(b,a+b);  
}
```

と記述します。どっちでもいいです。必ず終了するようになってねばね。

ちなみに↑の例ですと1方向にのみ進むのでツリーとは関係ないようですが例えは

```
void Traverse(オブジェクト){
```

```

for(auto& child : オブジェクト->Children()){
    Traverse(child);
}
}

```

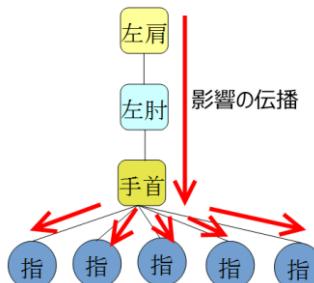
とやれば親子構造をきれいになめ回していくようになりますね？

ちなみにツリー構造と再帰処理は字句解析などにも使用されます。時間がある人は小手調べに

$3+2*(3+6)*(4-2)+16$

と言う文字列が渡されたら計算結果として 45 を出せるようなプログラムを書いてみてもらおうとまあ色々と力がつくんじゃないかなあと思います。ただそれやり始めると終わらないので、土日にでもやつといてください。

まあそれはともかく何故ここでツリー構造と再帰の話をしたのかと言うと、ボーンの構造と言うのはツリー構造になっているからです。例えば左肩からですが…



子が一つでなく複数になっています。ちなみにツリーの実装ですが本当に色々なやり方があって、リスト的に作るなら

```

struct Node {
    (内容)
    Node* child;//子ノードへのリンク
    Node* sibling;//弟へのリンク
}

```

と言う風に作ります。

普通のリンクリストに「弟」が追加されただけです。ちなみに brother とか sister でもいいんですが、たいていの場合は sibling って単語を使います。どうでもいい事ですが「しぶりん」と発音します。



ほんとどうでもいいですねすみません

もしくは親が全子供を知っているパターン

```
struct Node {  
    (内容)  
    std::vector<Node*> children; // 子供たちノードへのリンク  
}
```

子供が親を知っているパターン

```
struct Node {  
    (内容)  
    Node* parent; // 親へのリンク  
}
```

PMDのボーンデータ構造がこれに当てはまります。ただ、プログラムで全部再帰的に動かそうとするとちょっと良くない構造に思えます。何故ならこれを効率的に行うためには全リーフ(ツリーの末端)がどれか知っておく必要があるからです。

だからやっぱり親→子への流れがいいかなーって思っています。

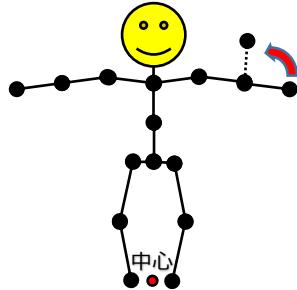
具体的にどういうやり方でポージングしていくの？

ここまで話でもピンと来てない人もいるかもしれないのでもう少し具体的な話をしよう。人体でもロボットでもそうなんですが、とにかく「骨」にあたる部分と言うのは伸び縮みしません。「俺の骨は伸び縮みするぞおージョジョー！」とかいう人は言ってください。大学病院に売り飛ばします。ていうかそれもう骨ちゃうやろ…

ともかく「骨」と言っても結局は(伸び縮みしない)データにすぎません。どういうデータかというと骨にくつづいてる頂点を「回転」させるためのデータです。基本的にボーンと言うのは関節を中心とした「回転」しかしません。

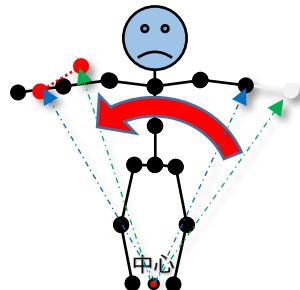
「拡大縮小」も「平行移動」もしません。さっきも言ったように骨は伸び縮みしないし、平行移動したらそりやお前…関節外れてますよね。

というわけで任意軸周りの回転情報が入っています。じゃあ 3×3 行列でいいのかと言うとそうでもなくて、

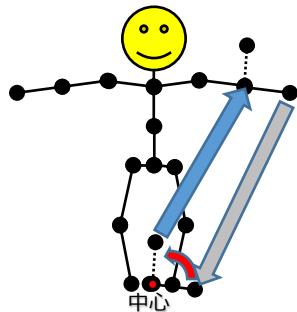


こいつの肘を曲げて前腕を動かしたいとする

そしてモデルの中心はたいていの場合股間の下の足元である。すると回転だけだと全ての回転がここを中心に行われることになり…



まあ割とえげつない事になります。棒人間だからいいけどこれモデルでやるとショック受けます?なのでさっさみたいなことをやりたければ



ボーン起点を中心に平行移動→回転→ボーン起点を元の座標に移動

という操作が必要になります。このため結局 4×4 行列が必要になります。また、ボーン起点とか言うてますけれども、この起点も親関節が曲がっている場合はそれを考慮してあげないといけなくなります。

つまり…例えば肩→肘→手首という構造になってるなら

(※平行移動⇒回転⇒戻し移動をとりあえず「回転」と呼ぶ)

肩：肩回転

肘：肩回転*肘回転

手首：肩回転*肘回転*手首回転

と言う風に末端に行けば行くほど親の影響を受けまくります。当然ですね？

まあ、ともかく面倒くさいので、この面倒な回転行列によって、頂点の座標を変換していくわけですが、スキーリングで説明したように、ウェイトで行列の乗算具合を調整します。つまり例えば肘辺の頂点が行列 A と行列 B に影響を受けていて、その影響度が + と 1- だとすると

$$P' = (tA + (1 - t)B)P$$

てな感じで行列がブレンドされます。面倒ですねえ。

理屈はここまで、さっさと実装しましょう。

ボーン情報をロード

とにかく手始めにボーン情報をロードしてみます（まだロード確認だけでナニモシマセンよ？）データ構造はこうなっています。

https://blog.goo.ne.jp/torisu_tetosuki/e/b384b3f52d0adbca1c46fd315a9b17d039 バイトです。

bone[17].bone_name[0]	8D B6 8C A8 00 FD 左肩
bone[17].bone_name[16]	FD FD FD FD
bone[17].parent_bone_index	0001
bone[17].tail_pos_bone_index	0012
bone[17].bone_type	00
bone[17].ik_parent_bone_index	0000
bone[17].bone_head_pos[0]	3E0BA27B 41810A36 3E879F45

全部で 6 種類のデータ。頂点に比べりや少ないですね。

まずボーンの数を予め取得してきますが気を付けてください。

bone_count	007A
------------	------

2バイトだこれ!!!

これを忘れるトドウンドウンズしていきます。どちらにせよ読み込まない事には始まらないので読み込んでいきましょう。

とりあえずもうヒントコード書かないから自分で読み込んでください。今まで何度も同じような事やってんだから分かるでしょ？

中身を見て、こんな感じでボーン名が壊れてないりやたぶん大丈夫です。ちなみに↑のボーン

```
b.sized.bones { size=0x0000000000000095 } + bone_index) + sizeof(b.tail_pos_bone_index), 1, fp);  
b.bone_type  
b.ik_paren  
b.bone_head  
b.ikNum  
m.sizeof  
「つ飛ばす  
= 0; i <  
b.4, SEEK  
d.char ikc  
ikchainNun  
b.6, SEEK  
[capacity] 0x0000000000000095  
[allocator] allocator  
[0x00000000] {bone_name=0x000002b42f42d400 "全ての親" parent_bone_index=0xffff tail_pos_bone_index=0x0001 ...}  
[0x00000001] {bone_name=0x000002b42f42d428 "センター" parent_bone_index=0x0000 tail_pos_bone_index=0x0002 ...}  
[0x00000002] {bone_name=0x000002b42f42d450 "グループ" parent_bone_index=0x0001 tail_pos_bone_index=0x0000 ...}  
[0x00000003] {bone_name=0x000002b42f42d478 "上半身" parent_bone_index=0x0002 tail_pos_bone_index=0x0004 ...}  
[0x00000004] {bone_name=0x000002b42f42d4a0 "上半身 2" parent_bone_index=0x0003 tail_pos_bone_index=0x0061 ...}  
[0x00000005] {bone_name=0x000002b42f42d4c8 "胸親" parent_bone_index=0x0004 tail_pos_bone_index=0x0062 ...}  
[0x00000006] {bone_name=0x000002b42f42d4f0 "右胸" parent_bone_index=0x0005 tail_pos_bone_index=0x0063 ...}  
[0x00000007] {bone_name=0x000002b42f42d518 "左胸" parent_bone_index=0x0005 tail_pos_bone_index=0x0064 ...}  
[0x00000008] {bone_name=0x000002b42f42d540 "ジッパー" parent_bone_index=0x0004 tail_pos_bone_index=0x0065 ...}  
[0x00000009] {bone_name=0x000002b42f42d568 "右肩" parent_bone_index=0x0004 tail_pos_bone_index=0x000a ...}  
[0x0000000a] {bone_name=0x000002b42f42d590 "右腕" parent_bone_index=0x0009 tail_pos_bone_index=0x000d ...}  
[0x0000000b] {bone_name=0x000002b42f42d5b8 "右腕捩" parent_bone_index=0x000a tail_pos_bone_index=0x000c ...}  
[0x0000000c] {bone_name=0x000002b42f42d5e0 "右腕捩先" parent_bone_index=0x000b tail_pos_bone_index=0x0000 ...}
```

名に「捩」という文字がありますが、ボーンの名前に「捩」だの「捩」だのいう名前がついてる奴には気付けてください。ちょっとばかし(いや非常に)ややこしいです。まずはブレーンなミクさんで練習しましょう。

まあうまくいってりゃロードはOKです。ここからツリー構造を構築していきましょう。なお、さっきのツリーの説明だけ聞くと全てのボーンが一つにつながってそうなイメージがあるかもしれません、PMDのボーンデータはそうでもないのでお気を付けください。

ツリーを構築

ここ、毎年悩みどころなんですよね…。ツリーの話ですが、その前にまず言っておきますが、ボーン数に応じた行列配列を投げることになります。ちなみに投げられる行列配列の数は固定なので256個投げます。

で、その行列を頂点に乗算するため、行列は投げなきゃいけない。配列順序通り。つまり投げる配列は予め決めておいて、ツリー自体の「内容」はインデックスになるわけだ。

あと、曲げたいボーンは「ボーン名」で指定できる必要があるためそこはmapを使うのがいいだろう…。

ボーン名で探した先のやつのインデックスからその下のツリーが分かる状態にしなければならない…。

というわけで実装に必要な情報、条件は

- 連続した行列(配列)
- ボーン名でインデックスを検索できるように

- 親が子を知っているリンク(複数)

これを組み合わせるわけです。当然3つが「ドラ」になってしまっているわけではないのでそれぞれの特性を考慮しなければなりません。

な?現実のプログラミングって奴ア検定の覚えゲーみたいにはいかねえだろう?ここを自前でできるようになればそりやあお前、ショボい検定ゼ~んぶ獲ったの以上に実力ついとるで?

もう基本的なデータ構造勢ぞろいだね!!!これで基本情報技術者試験もバツチリダネ!!!ヤッタネ!!!

さて、まず当然ながらボーンの配列が絶対に必要なのでこうなる。

```
vector<XMMATRIX> boneMats;
```

これは簡単。他の奴はインデックスを知っておけばいい事になるが…
それではこうかな?

```
map<string,int> boneMap;
```

ところがこれではここからツリーを下ることができない…。ということで次のツリー構造を考えるがこれは『子が複数』いるわけだ。

となると

```
struct BoneNode{
    int boneIdx;//ボーン行列配列と対応
    XMFFLOAT3 startPos;//ボーン始点(関節初期座標)
    XMFFLOAT3 endPos;//ボーン終点(次の関節座標)
    vector<BoneNode*> children;//子供たちへのリンク
};
```

というツリーの元が考えられる。つまり、こいつをマップで探せるようにすればいいかなと思います。

```
map<string,BoneNode> _boneMap;
```

当然のように読み込んだPMD側のデータはこうなっています。

```
struct PMDBone {
    char boneName[20]; // ボーン名
```

```

        unsigned short parentBoneIndex; // 親ボーン番号(ない場合は0xFFFF)
        unsigned short tailBoneIndex; // tail位置のボーン番号(チェーン末端の場合は
0xFFFF 0 →補足2) // 親：子は1:多なので、主に位置決め用
        unsigned char boneType; // ボーンの種類
        //パディングが入りますぞ
        unsigned short ikParentBoneIndex; // IKボーン番号(影響IKボーン。ない場合は0)
        //パディングが入りますね
        DirectX::XMFLOAT3 bonePos; // x, y, z // ボーンのヘッドの位置
    };
    (中略)
    std::vector<PMDBone> _bones;

```

で、今度は使用者側は

```

std::vector<XMMATRIX> _boneMatrices;//ボーン行列転送用
std::map<std::string,BoneNode> _boneMap;//ボーンマップ

```

というのを用意しておく。まあ、行列配列自体はこのボーン数分確保すればいいので

```

//ボーン配列初期化
_boneMatrices.resize(_model->Bones().size());
std::fill(_boneMatrices.begin(), _boneMatrices.end(), XMMatrixIdentity());

```

このコードの説明…もう皆さんにはできますよね？できますよね？

次にツリーを考慮せずマップ情報を構築しようとすると…

```

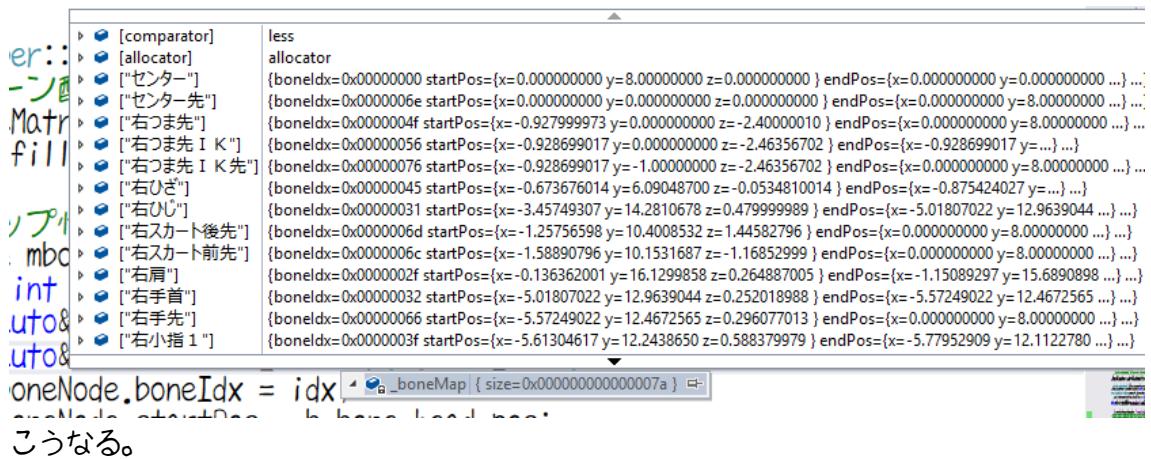
//マップ情報を構築
auto& mbones = _model->Bones();
for (int idx = 0; idx < mbones.size(); ++idx) {
    auto& b = _model->Bones()(idx);
    auto& boneNode = _boneMap(b.boneName);
    boneNode.boneIdx = idx;
    boneNode.startPos = b.bonePos;
    boneNode.endPos = mbones[b.tailBoneIndex].bonePos;
}

```

こうなる。

一旦これで実行してみよう。

うまいことやれてねば



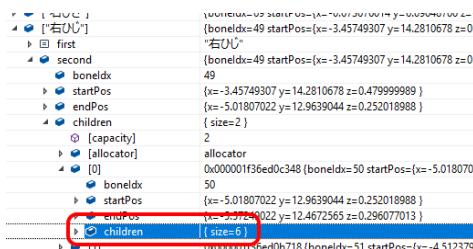
こうなる。

つまりマップにきっちり必要な情報が入っている状態である。こうなっていれば次の段階へ行ける。ツリー構築だ。

ループで回しながら親のベクタにドウンドゥン追加していく。

```
for (auto& b : _boneMap) {
    if (mbones(b.second.boneIdx).parentBoneIndex >= mbones.size()) continue;
    auto parentName = mbones(mbones(b.second.boneIdx).parentBoneIndex).boneName; // ikのボーンと間違えないように
    _boneMap(parentName).children.push_back(&b.second);
}
```

うまい事ツリーが構築できていれば



右ひじの2つ下のノードが6本になっています

感がいい人は肘の先のボーンが2本で、手首から先のボーンが6本であることに違和感を覚えると思います。

が、これは何故か肘から「手首」と「袖」の二つのボーンが飛び出しているからです。

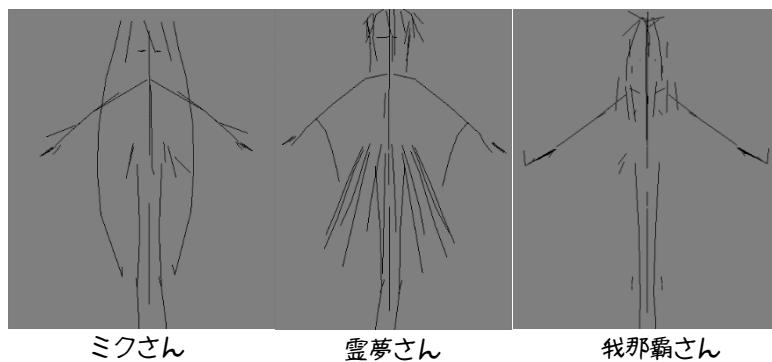
また、指が6本に見えるのは指5本に加えて「手先」という「非表示ボーン」があるからです。

今は無視しといついでです。(非表示ボーンは基本的に作業用なので再生時には意味がありません)

ここまでやれば「ボーンツリー」が構築できることになります。

ここでちょっと迷ってるのは「ボーンの表示」をやるかどうか…かなあ。さっさとスキニング実装してゲームを作ることを最優先に考えると…別にボーンの表示はしなくてもいいかなあって思っています。

例年だとデフォルト用にってのとボーンの理解用に



こういう針金みたいなの作って、こいつを曲げたりしてたんですけどね…まあ時間がない中でやるようなものじゃないよね。という事で今年はこれやりません(要望があればやりますが…)

特定ボーンの回転

ポージングへの第一歩です。ミクさんの腕を曲げてみようと思いまます。

準備

その前に、頂点情報をボーン設定に必要な分をすべて投げてあげましょう。今一度思い出して
…

```
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
```

とりあえず2つのボーン番号を入れましょう。ちょうどいい事にR16G16_UINT的なフォーマット

トがあります。

```
{ "BONENO", 0, DXGI_FORMAT_R16G16_UINT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
```

ちなみにセマンティクスを BONENO としていますが、これは『ユーザー定義セマンティクス』に当たります。ともかく2つの16バイトを投げましょう。

ちなみに min16uint という型があります

<https://docs.microsoft.com/ja-jp/windows/desktop/direct3dhlsl/dx-graphics-hlsl-scalar>

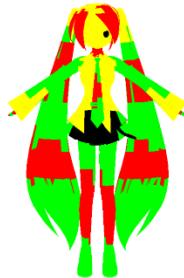
日本語情報がないのでちょっと不安がありますが、きちんと使えますので、安心してください。

```
min16uint2 boneno : BONENO)
```

2個受け取るんで型名の後に2を付ければ2個セットになります。試しにこのインデックスを使って

```
return float4((float2)(input.boneno%2),0,1);
```

てな感じで出力すると



このようになります

なんとなく『何故か』は分かりますかね？影響ボーン番号を%2したものを色に使っているため交互に色が変わるように形になります。ちなみに

```
return float4((float2)(input.boneno/128.0f),0,1);
```

と言う式にすると(128で割ってるのはボーン数が122だからです)



こうなります

影響ボーンの番号が変化していっているのが分かりますね。

次に影響度を受けとります。

影響度はデータとしては11バイトなので、レイアウト的には

```
{ "WEIGHT", 0, DXGI_FORMAT_R8_UINT, 0, D3D12_APPEND_ALIGNED_ELEMENT,  
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
```

とします。

シェーダ側ですが

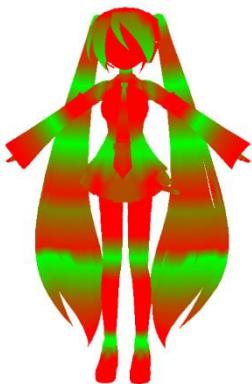
```
min16uint weight:WEIGHT
```

とします。min8uint的なのがあったら使ったんですが、それがないっぽいので。一応データは取ってこれてるんでOKとします。

試しに頂点シェーダ側でこうして

```
float w = weight / 100. f;  
output.color = float3(w, 1 - w, 0);  
ピクセルシェーダ側でこうしたら  
return float4(input.color, 1);
```

こうなりました。影響度がせめぎあっているのが分かると思います。



影響度の戦い!!

まあ片方がWでもう片方が1-Wなら当然せめぎあうわなあ…。それはともかくこれでボーンIDと影響度がシェーダ側に渡されているのが分かったと思います。

あとはボーンによる回転情報を投げてあげるわけですが、全投げします。シェーダ側としてはいくつ来るかわからないので512MATRIXを受け取る準備だけしておきます。b0,b1は使ってないので、b2を割り当てようかと思います。

///ボーン行列

```
cbuffer bones : register(b2) {  
    matrix boneMats(512);  
}
```

そして

ルート/パラメータは新しく作りましょう。

ヒープもビュー1つだけですが新しく作りましょう

となるとこんな感じですかね。

```
std::vector<DirectX::XMATRIX> _boneMatrices;//ボーン行列転送用  
std::map<std::string,BoneNode> _boneMap;//ボーンマップ  
ID3D12Resource* _boneBuffer;//ボーンバッファ  
ID3D12DescriptorHeap* _boneHeap;//ボーン用ヒープ
```

あとは「いつもの」ようにバッファ作ってどうぞ。あとはボーン用ヒープ作って定数バッファビューも作ります。

///バッファの作成

```
_boneBuffer.Attach(_buffMgr->CreateConstantBufferResource(size));
```

```
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_CBV_SRV_UAV;  
auto result = _dev->CreateDescriptorHeap(&descHeapDesc,  
IID_PPV_ARGS(_boneHeap.GetAddressOf()));
```

```
D3D12_CONSTANT_BUFFER_VIEW_DESC desc = {};  
desc.BufferLocation = _boneBuffer->GetGPUVirtualAddress();
```

```

desc.SizeInBytes = size;
auto handle = _boneHeap->GetCPUDescriptorHandleForHeapStart();
_dev->CreateConstantBufferView(&desc, handle);

_boneBuffer->Map(0, nullptr, (void**)&_mappedBones);

```

マテリアルより楽なのはマテリアルごとに切り替えが必要ない事です。久しぶりの新しいルートパラメータとヒープなので間違えないようにね？

```

// "b2" をつくるぞ
descTb1Range[3].NumDescriptors = 1; //
descTb1Range[3].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV;
descTb1Range[3].BaseShaderRegister = 2;
descTb1Range[3].OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;

rootparams[2].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
rootparams[2].DescriptorTable.NumDescriptorRanges = 1;
rootparams[2].DescriptorTable.pDescriptorRanges = &descTb1Range[3];
rootparams[2].ShaderVisibility = D3D12_SHADER_VISIBILITY_VERTEX;

```

ノーファ全部ぶんぬげるで、コマンドリスト発行も必要なし。ちなみにボーン情報はピクセルに一切関係しないので、可視範囲を頂点シェーダのみにしています。

マテリアルより…楽でしょ？

で、それができたらひとまず全てのマトリックスを XMMatrixIdentity で埋めます。モチロン Map したものを埋めなきゃだめですよ？

```
std::copy(_boneMatrices.begin(), _boneMatrices.end(), _mappedBones);
```

その上でシェーダ上ですべての頂点に対して、ボーン行列を乗算して元通りに表示される（単位行列の乗算だから変化しないはず…！）のご確認ください。

```

float w = weight / 100.f;
matrix m = boneMats(boneno.x)*w + boneMats(boneno.y)*(1-w);
pos = mul(m, pos);

```

とくに変化が出てなければ、ひとまずここまでで準備が終わりです。

実験(だいたい結果が予想できるやつ)

ひとまず肘にあたる部分を回転させてみましょう。マップを「左ひじ」で検索して、そいつにZ軸中心回転行列を書き込んでください。

//実験

```
_boneMatrices[_boneMap("左ひじ").boneIdx] = XMMatrixRotationZ(XM_PIDIV4);
```

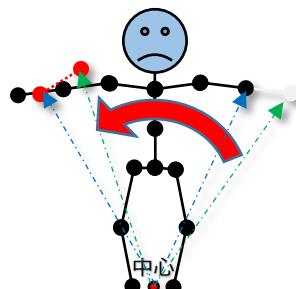
うまくいけば…



ああああああああああああああああ

うまくいくのか!?これ!?ホンマか!?嘘ちゃうんか!?ミクさんの腕どないなつとんねーん!!!

原因をいくつか言うてしまえば



これが起きてるんですわ。ひとまずこれから修正しまひよか?

ボーン中心回転(原点中心回転ではない!)

前にも言いましたが肘中心に回転するためには、一旦原点に戻してから回転して元の座標に戻さねばならない…ところで原点に戻すというがどうやつたら原点に戻るのだろうか?

ここでボーン情報を思い出してほしい。

「ボーンの座標」的なデータがあったと思います。アレを使用します。

あれこそ関節の座標です。つまりあの座標分マイナスしたものを現在の座標に足せば原点に平行移動します。そこから回転して、元の関節の座標を足してあげます。

また、以前にも言いましたが、行列は乗算することで↑のような操作を合成することができます。

// 実験

```
auto elbow = _boneMap["左ひじ"];
auto vec = XMLoadFloat3(&elbow.startPos);
_boneMatrices[elbow.boneIdx] = XMMatrixTranslationFromVector(XMVectorScale(vec, -1)) * XMMatrixRotationZ(XM_PIDIV2) * XMMatrixTranslationFromVector(vec);
```

ちなみに座標に-1をScaleしてるのは原点に戻すための平行移動ベクタを作るためです。
さて、このようにしてあげれば…



ううんん??

さて、何故だと思うね?

いや、肘から前腕までが回転して、それ以降がついてきていいからだろ…つまり自分の回転を自分の子々孫々に伝播しなければならないのだ!!!



子々孫々末代まで回転を伝播する

ツリー構造において、自分の子供にあたる奴ら…リーフに到達するまでドウンドウン回転情報を乗算してやる必要があるのだ。

そういう時に使えるのが以前にも説明した「再帰」です。再帰的ってのは英語で Recursive といいます。

RecursiveMatrixMultiplyとかそういう名前の関数を再帰的に使います。

まず、普通に再帰する事だけを考えて関数を作ると、こう。

```
Dx12Wrapper::RecursiveMatrixMultiply(BoneNode& node) {
```

```

        for (auto& cnode : node.children) {
            RecursiveMatrixMultiply(cnode);
        }
    }

```

ソリーガまともにできていれば、スタックオーバーフローになることなく子々孫々をトラバースできます。

でもこれだけじゃ意味がないねえ。親の行列が子に乗算されるようにしないと。ただ、この時順番には気を付けてほしいのだが…まあいいや。とりあえず乗算してみてそれが再帰的に乗算されるようにしてください。

色々と考えて自分で実装してみましょう。僕を頼らないでください。

例えば

```

void Dx12Wrapper::RecursiveMatrixMultiply(BoneNode& node, XMATRIX& inMat) {
    _boneMatrices[node.boneIdx] *= inMat;
    for (auto& cnode : node.children) {
        RecursiveMatrixMultiply(*cnode, _boneMatrices[node.boneIdx]);
    }
}

```

みたいに書いて、肘曲げた後にでもルートからこれをやってやれば…

```

XMATRIX rootmat = XMMatrixIdentity();
RecursiveMatrixMultiply(_boneMap["センター"], rootmat);

```



やったぜ

ついでに右の肘もやってあげましょう。



やりまくったぜ

肩も曲げてみましょう



コロンビアだぜ

一応他のキャラもやってみますが



靈夢さんOK



我那霸さんOK



あれ?失敗すると思ったのに…雲雀さんOK

大体抢险りボーンが入ってるとここらへんで失敗するんですが…。



こんな風になるはずなんんですけどねえ…

まあ問題ないならいいでいいか…進みましょか!

VMD ファイルを読み込む

VMD ファイルってのは MMD のモーション用のデータです。モーションはちょっと難しいのですが、まずはポージングすることだけ考えましょう。

サーバーに pose.vmd というのを置いてますので、それを利用してポージングします。

https://blog.goo.ne.jp/torisu_tetosuki/e/bc9f1c4d597341b394bd02b64597499d

見てフォーマットを確認しましょう。

一応サーバーに vmd.sym も置いてあるので必要な人は活用してください。入れとけば

header.VmdHeader[0]	36 6F 63 61 6C 6F 69 64 20 4D 6F 74 69 6F 6E 20	Vocaloid Motion
header.VmdHeader[16]	44 61 74 61 20 30 30 30 32 00 00 00 00 00 00	Data 0002
header.VmdModelName[0]	8F 89 89 B9 83 7E 83 4E 00 FD FD FD FD FD FD FD FD	初音ミク
header.VmdModelName[16]	FD FD FD FD
motion_count	0000007A	
motion[0].BoneName[0]	83 5A 83 93 83 5E 81 5B 00 FD FD FD FD FD	センター
motion[0].FrameNo	00000000	
motion[0].Location[0]	00000000 00000000 00000000	
motion[0].Rotation[0]	00000000 00000000 00000000 3F800000	
motion[0].Interpolation[0]	14 14 00 00 14 14 14 14 6B 6B 6B 6B 6B 6Bkkkkkkkk
motion[0].Interpolation[16]	14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 00kkkkkkkk
motion[0].Interpolation[32]	14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 00kkkkkkkk
motion[0].Interpolation[48]	14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 00kkkkkkkk
motion[1].BoneName[0]	8F E3 94 BC 90 67 00 FD FD FD FD FD FD	上半身
motion[1].FrameNo	00000000	

で、ぱっと見何のデータだこれ…って感じですよね。まあ一気に解説するのはちょっと困難なので、ひとまずは「名前」と「回転」のデータだけを取得します。

で、回転のデータは当然 Rotation の部分なのですが、これ4要素あります。普通に皆さんのが考えてる回転だと3つでいいはずですよね？

あのケチケチ MMD のフォーマットがこれって…？ 製作者兄貴が多重人格である可能性が微粒子レベルで…!!

その疑問を解消するために上記サイトでフォーマットを見ましょう。

```
char BoneName(15); // ボーン名  
DWORD FrameNo; // フレーム番号(読み込み時は現在のフレーム位置を0とした相対位置)  
float Location(3); // 位置  
float Rotation(4); // Quaternion // 回転  
BYTE Interpolation(64); // (4)(4)(4) // 補完
```

ん？今…クオータニオンって書いてたよね？(英語で)

MMD 回転クオータニオン説

つまり僕らが最初に思い描く X 回転 Y 回転 Z 回転ではないのだ。クオータニオンは日本語で四元数。四元数は4つの数値。4つの数値だから float4 つぶん必要。つまりこれはクオータニオンで間違いない! Q.E.D。

クオータニオンって何ですかねえ…

クオータニオンって当然のように言ってますけど、クオータニオンってのは

<https://qiita.com/kenjihiranabe/items/945232fbde58fab45b81>

<http://www.mss.co.jp/technology/report/pdf/18-07.pdf>

<https://www.f-sp.com/entry/2017/06/30/221124>

<https://ja.wikipedia.org/wiki/%E5%9B%9B%E5%85%83%E6%95%B0>

http://marupeke296.com/DXG_No10_Quaternion.html

https://qiita.com/edo_m18/items/5db35bb0112e281f840e

なるほど!!! わからん!!!

でもご安心を!!! ゲイツ兄貴もそこまで鬼ではございません

XMMatrixRotationQuaternion

っていう関数があります。これはこの「クオータニオン」から任意軸回りの回転行列を生成するものです。

https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixrotationquaternion.aspx

X 軸回転 + Y 軸回転 + Z 軸回転だと色々と不都合もありますしね…

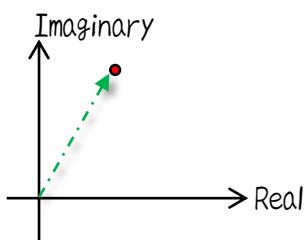
ちなみに「クオータニオン」とは何かとかやってたら 1 年終わっちゃいますので、やめましょう。軽く説明しておくと複素数($z=a+bi$)の概念を三次元に拡張したものだと思って良いと思います。

虚数は知っていますね? 二乗して -1 になる奴です。そいつと実数を組み合わせたものを複素数と呼びます。

ちなみに複素数で 2 次元の回転(角度)を表す事もできます。例えば

$$\frac{1 + \sqrt{3}i}{2}$$

という複素数は 60° を表します。どういう事が分かりますか? 二次元の座標系を書いて横軸

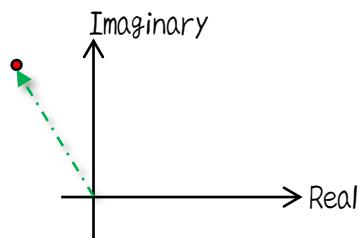


を実部(R)とし、縦軸を虚部(I)とします。つまり虚数値が出てきたらそれを Y 軸の値として、それ以外を X の値とします。そうすると上の複素数は
と言う感じになり 60° を表しているという事にします。

例えば角度を 120° にしたいとすると 60° の複素数を乗算します。つまり

$$R_{120} = R_{60}R_{60} = \frac{1 + \sqrt{3}i}{2} \cdot \frac{1 + \sqrt{3}i}{2} = \frac{(1 + \sqrt{3}i)^2}{4} = \frac{-2 + 2\sqrt{3}i}{4} = \frac{-1 + \sqrt{3}i}{2}$$

となり、新たにグラフに書き込んでみますよ



さらに 60° 回転します ($\frac{1+\sqrt{3}i}{2}$ を乗算します)。そしたらどうなる? 180° だよねえ。

$$R_{180} = \frac{-1 + \sqrt{3}i}{2} \times \frac{1 + \sqrt{3}i}{2} = -\frac{4}{4} = -1$$

これもうグラフに書くまでもなく、回転と運動しているよね。この虚数部分を 3 次元に拡張したのがフォータニオンなんだ。それぞれの軸に当たるフォータニオンは通常の虚数と同じように二乗すれば -1 になります。3 つあるのでそれぞれ i, j, k とあり、概念上 4 次元になっています。複素数と同じように

$$Q = aR + bi + cj + dk$$

実部と i, j, k はそれぞれ全て直交しています。そして妙な法則があります。

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = k, ji = -k$$

$$jk = i, kj = -i$$

$$ki = j, ik = -j$$

まあざつとこういう数であることを知っておけばいいと思います。で、こいつが空間内の回転に非常に役に立つのだが、それを何故かとやり始めるとテンソルだのユニタリ行列だの共軛だの出てくるので、この際ブラックボックスでいいと思います。

とにかくフォータニオンは

- 複素数を 3D に拡張したもの
- とにかく回転に使える(オイラー回転と違ってジンバルロックなどの不具合を起こさない)

い)

- 中身については気にしない方がいい
のでございまーす。

データを読み込む

VMDMotion というクラスを作りましょう。その中でデータを読み込んで必要な情報を返すようなクラスにしましょう。(やるこた PMDModel クラスと同じです)

ひとまず読み込んだだけをやってみましょう。もうソースコードとかいらないよね?

https://blog.goo.ne.jp/torisu_tetosuki/e/bc9f1c4d597341b394bd02bb4597499d
を見れば分かるよね。アライメントに気を付ければ大丈夫だよ。

まあ、最初の 50 バイトは無駄データだわな…

```
fseek(fp, 50, SEEK_SET);
fread(&keyframeNum, sizeof(keyframeNum), 1, fp);
```

次にキーフレームデータ。モーションデータってのは特定のボーンの特定のフレームにおける回転角度(とかその他状態)を示します。エディタで言うと

◆のことです/

とにもかくにも、ファイルの中でこれが「登録された順」に並んでいます。この順序が後々ぼくらを苦しめる…。

ひとまず最初の 0 フレームしかないデータを作ったので(pose.vmd)そのポーズ通りになるようにしていくのが目的です。ともかくすべて読み込みましょう。

```
for (auto& keyframe: keyframes) {
    fread(keyframe.boneName, sizeof(keyframe.boneName), 1, fp); // ボーン名
    fread(&keyframe.frameNo, sizeof(keyframe.frameNo) + // フレーム番号
          sizeof(keyframe.location) + // 位置 (IK のときに使用予定)
          sizeof(keyframe.quaternion) + // クオータニオン
          sizeof(keyframe.interpolation), 1, fp); // 補間ベジエデータ
}
```

読み込んだら中身が壊れてないかご確認ください。全てのボーン名がまともなら大丈夫でしょう。

さて…データは読み始めたのでこれをさっさと使えるデータへと加工しましょう。

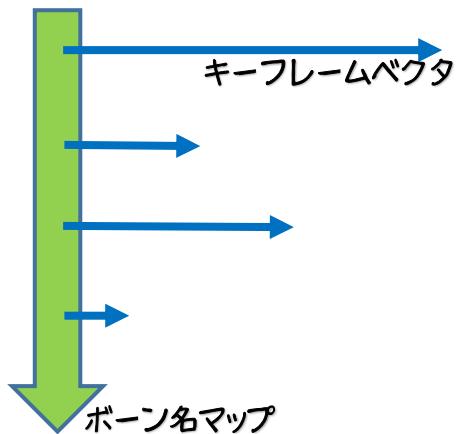
データの加工

/今一度 MMD のエディタを見てみましょう。

縦軸に「対応するボーン名」が書かれていて、横にキーフレームが登録されているような状況です。もちろんこの「キーフレーム」に

- フレームNo.(数値)
- 回転情報
- IK 移動情報
- 補間曲線データ

が含まれているわけですがひとまずはフレームNo.と回転情報だけでいいでしょう。また、エディタのデータ配置を簡略化して書くと



このような形になっているわけです。何となく分かりますかね？つまり例えばキーフレーム情報を

```
struct Keyframez{  
    int frameno;  
    XMFFLOAT4 quaternion;  
};
```

とかやっちゃんります。これが横に複数並ぶので(ポーズデータの場合は並んでないが)
vector<Keyframe> _keyframes;

となります。さらにこれがボーン数ぶんあるし、ボーンの名前で検索できるようになつとした方が便利っぽいのでマップにしておきます。

```
map<string,vector<Keyframe>> _animation;  
みたいなのを作つければ
```

```

for (auto& f : keyframes) {
    _animationdata[f.boneName].emplace_back(Keyframe(f.frameNo,f.quaternion));
}

```

こうするだけでえ…あとはわかるな?
あとはクライアント側から見れるように適当な関数を作つて…

```

const std::map<std::string, std::vector<Keyframe>>&
VMDMotion::AnimationData()const {
    return _animationdata;
}

```

これをクライアント側から参照できるようにすればいい。

クライアント側

もうここまで来たら全然難しくない。既に
void

```

Dx12Wrapper::RecursiveMatrixMultiply(BoneNode& node,XMMATRIX& inMat) {
    _boneMatrices(node.boneIdx) *= inMat;
    for (auto& cnode : node.children) {
        assert(node.boneIdx >= 0);
        RecursiveMatrixMultiply(*cnode, _boneMatrices(node.boneIdx));
    }
}

```

```

void Dx12Wrapper::RotateBone(const char* bonename,const DirectX::XMFLOAT4& q) {
    auto& bonenode = _boneMap(bonename);
    auto vec = XMLoadFloat3(&bonenode.startPos);
    auto quaternion = XMLoadFloat4(&q);
    _boneMatrices(bonenode.boneIdx) =
        XMMatrixTranslationFromVector(XMVectorScale(vec, -1))*
        XMMatrixRotationQuaternion(quaternion)*
        XMMatrixTranslationFromVector(vec);
}

```

こういう関数を作っているだろうから後はモーションに入っているボーン全てに対してモーション内のクオータニオンを適用してやればいいわけ
//ポージング適用

```

for (auto& boneanim : _motion->AnimationData()) {
    auto& keyframe=boneanim.second.back();
    RotateBone(boneanim.first.c_str(),keyframe.quaternion);
}

//ソリーラバース
XMMATRIX rootmat=XMMatrixIdentity();

RecursiveMatrixMultiply(_boneMap["センター"],rootmat);

```

とでもしてやればデータ通りのポーズをとるはずです。

ちなみにサーバに上げてるのは



こういうデータなので



こうなってればOK

ということでひと段落。

ちょっとお疲れさん。

アニメーションしてまう

「ポージングできんじゃーん!!」

「お、そうだな。アニメーションなんてすぐだよ」

「そうだよ」

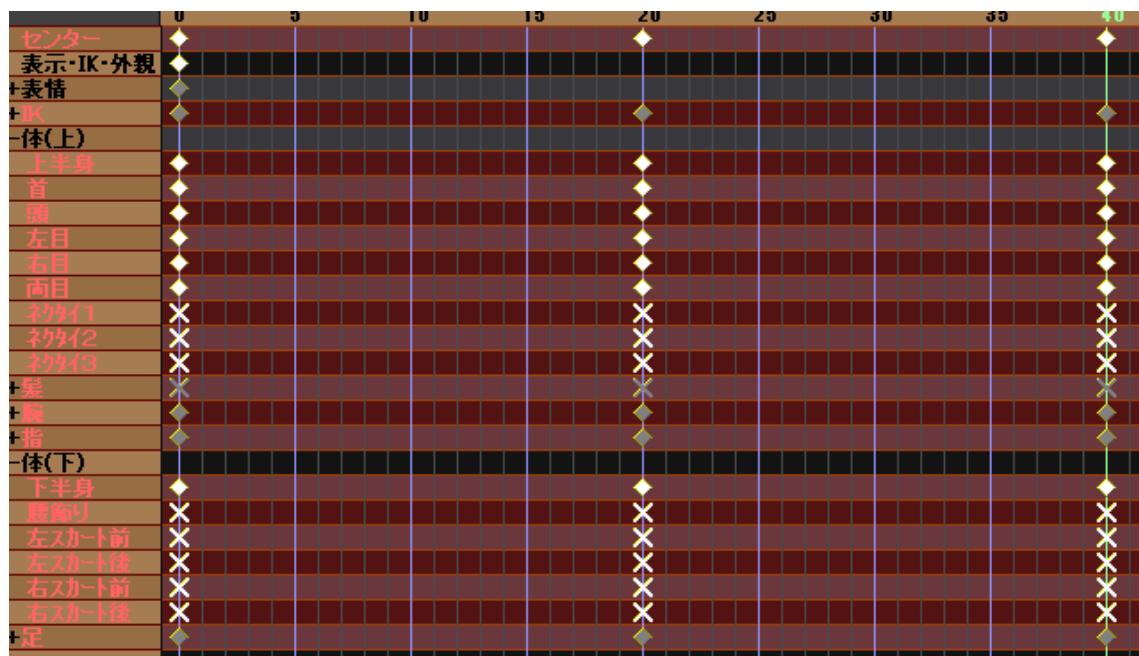
「俺もやったんだからさ!!」

とか思つてるとたぶん痛い目を見るので、痛い目に遭いたくない人は今のうちにリバースイテレータとか見直しておきましょう。

ん? ポージング出来たらあとはそれを補間するだけちゃいますのん?

そう甘くないんだコレが。

例えば



こういう風にきれいに並んでいればいいんですが、そんなのは稀です。実際のモーションは本当にバラバラです。

まあいいや。ひとまず1つのボーンだけを動かすデータを作りましょうか。以前のポーズから右ひじだけを15フレーム毎に動かすデータを作りました。



first.vmd という名前で保存しています。

さて、これをどう実装していくかって話ですよ。

既にモーションデータは名前で検索できるようにしています。更に言うと

```
std::map < std::string, std::vector<Keyframe> > _animationdata;
```

という作りにしているため特定のボーンの中に複数キーフレームがあり、それも
`vector<Keyframe> keyframes = _animationdata("右ひじ");`
 などで取得できる状態である。

たとえば時間ごとにフレームが進むとして、そのフレームに対応したボーン情報を取ってこ
 れればいい。ちなみにMMDは30フレームなので2回ループで1フレームくらいでちょうどいいで
 しょう。

という事で、前にやったポーズの

```
//データからポージング適用
for (auto& boneanim : _motion->AnimationData()) {
    auto& keyframe=boneanim.second.back();
    RotateBone(boneanim.first.c_str(), keyframe.quaternion());
}

//ツリーをトラバース
XMMATRIX rootmat = XMMatrixIdentity();
RecursiveMatrixMultiply(_boneMap["センター"], rootmat);
```

を毎フレームやることになります。んまあ、それはそれとして…フレームに対応させるならどうしましようかねえ…。↑のやつは関数化しておきましょうか。

ひとまず現在のフレームと一致するデータを探して、見つかったら適用。見つからなかったら放置(現在の状況を維持する)としましょうかね。

```
void
Dx12Wrapper::MotionUpdate(int frameno) {
    //データからポーリング適用
    for (auto& boneanim : _motion->AnimationData()) {
        auto& keyframes = boneanim.second;
        auto frameIt = std::find_if(keyframes.begin(), keyframes.end(),
            (frameno)(const Keyframe& k) {return k.frameNo == frameno; });
        if (frameIt == keyframes.end()) continue;
        RotateBone(boneanim.first.c_str(), frameIt->quaternion);
    }
    //ツリーをトラバース
    XMATRIX rootmat = XMMatrixIdentity();
    RecursiveMatrixMultiply(_boneMap["センター"], rootmat);
}
```

ということでUpdateの中で

```
MotionUpdate(frameno / 2);
std::copy(_boneMatrices.begin(), _boneMatrices.end(), _mappedBones);
```

を毎フレーム呼んでやる。

結果…



となる。

なんかといふと

再帰関数の中で

```
_boneMatrices[node.boneIdx] *= inMat;
```

と言う箇所がある。これが乗算であるが故に毎フレームどんどんずれていく。毎フレームと言うのは1/60である。あつといふ間に異形の生物になってしまふわけだ。

なので無駄なようであるが毎フレーム初期化をしてやる。ClearDrawScreen みたいになもんだと思ってもらえばレリケイかな。

```
std::fill(_boneMatrices.begin(), _boneMatrices.end(), XMMatrixIdentity()); //初期化  
MotionUpdate(frameno / 2);  
std::copy(_boneMatrices.begin(), _boneMatrices.end(), _mappedBones);
```

と言う具合になる。

ただ、これだと現在の実装だと



あれ？ そんなばかかな…

よくよく見ると一瞬だけポーズをとっていることがわかる。そう…今の実装ではダメなのだ。

何処がダメかと言うとここだ。

```
find_if(keyframes.begin(), keyframes.end(), [frameno](const Keyframe& k) {return  
k.frameNo == frameno;});
```

あ、しつと知ってる体(ついで)説明しちゃつたけど、このプログラム…何してるか分かります？「分かる」ってのは「他人に説明できる」ってことだよ？できます？

まず find_if

https://cpprefjp.github.io/reference/algorithm/find_if.html

そしてラムダ式

```
(frameno)(const Keyframe& k )->bool{ // `->bool` は省略可能
    return k.frameno == frameno;
}
```

ご説明可能かな?

find_if は簡単に言うと第三引数に書いてある関数オブジェクトが true の要素を返します。
入れるべきは「関数オブジェクト」か「ラムダ式」です。

さて、今回のラムダ式は↑のようになっています。なお、frameno がクロージャとして入れられていますが、これは引数でもないんですね。今回のようなパターンの時に役に立ちます。

なお、find_if 等が要求する関数は、要素一つ一つの参照を引数として、ブール型を返す関数です。

クロージャは関数定義時に決まるものなので、今回ここで使用している frameno は find_if ループ中に変化することはありません。ということで、こいつと比較しているわけです。

知識量と言うか情報量が多いんですけど、大丈夫でしょうか? お判りでしょうか? 説明できますでしょうか?

で、今の問題は現在のフレーム NO と合致したもの返して、そのクオータニオンを適用するのですが、イコールで結ぶと、そのフレームでしか合致しないため一瞬で元に戻ってしまってました。こいつに継続的なポーズを取らせるにはどうしたらよいのでしょうか?

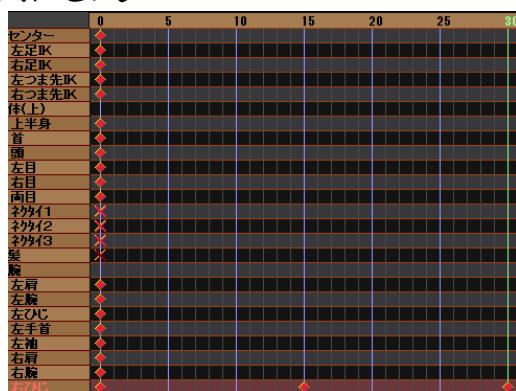
今のフレームよりも前に当たるキーフレームを対象にすればいい。そう思いますよね? つまり

```
return k.frameno <= frameno;
```

こうすればいいとお思いでしょうか? そう思うならこれで実行してみましょう。



たしかにポーズはとるけど…動かねえな
もう一度よく考えてみてください。



例えばこの状況で 30 フレーム以下のキーフレームを検索すると…
アッターー！

もう…お分かりですね？30 フレーム以下ってどういうことですか？そう…30 フレームも 15 フレームも 0 フレームもみんなまとめて「30 フレーム以下」なわけです。

というわけで「N フレーム以下のキーフレームで、その中で最大フレームの「キーフレーム」」が必要なわけです。

どうすればいい？

反対から検索すればいいんじゃねえの？

どうやったら反対側から検索できるの？

リバースイテレータだよ馬鹿野郎!!! 伏線張ってたじゃねえか!!! という事で、find_if に突っ込むイテレータをリバースにしてしまえば反対側から検索できます。え？リバースイテレータが分からん？rbegin と rend だよ。

さて、コードは書きませんので、検索をリバースイテレータにしてください。そうすればミクさんがぎこちなく、なんでやねんしてくれるとおもいます。



しかしまだまだぎこちない…もっとぬるりと動いてほしい。

フレーム補間しよう

なぜぬるりと動かないかと言うと、フレームからフレームに一気に動いているからです。というわけでフレーム補間を行います。

情報はキーフレームにしか入っていないため、キーフレームとキーフレームの間を線形線形します。

線形補間の式は分かりますかね？

例えば A の状態(行列)から B の状態(行列)まで 30 フレームかかるとして A から 21 フレームの時の状態を表すとするとどうすればいいでしょうか？

$$M = A \frac{30 - 21}{30} + B \frac{21}{30}$$

A のフレームを F_A とし、B のフレームを F_B とし、現在のフレームを F とすると

$$M = A \frac{(F_B - F_A) - (F - F_A)}{F_B - F_A} + B \frac{F - F_A}{F_B - F_A}$$

これをプログラムで表現すればいいのです。まあ元々

$$M = (1 - t)A + tB$$

が「線形補間の式」という事が頭に入っていれば

$$t = \frac{F - F_A}{F_B - F_A}$$

とすればOKであることが分かりますね。あとは自分で考えて…

やつておしまい!!



アラエラサツサー!!

できましたか? 行列も掛け算や足し算ができるので、試してみてください。

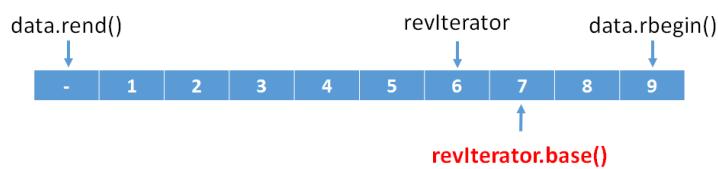
ちょっと早く分かりづらいかも知れないるので、分りづらかったらフレームの更新をゆっくりにしてみてください。

リバースイテレータと base()

ところでリバースイテレータを使えと言いましたが reverse_iterator には base()って関数があります。

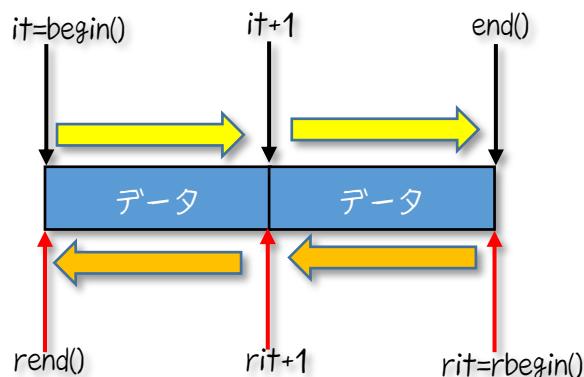
http://en.cppreference.com/w/cpp/iterator/reverse_iterator/base

こいつでイテレータにすることができるんですが、実はこの base()結果の イチ が指し示すものは



うん、つまりリバースイテレータの一つ後になるんですよね。前に話したと思いますが、イテレータは実際のデータの手前を指している。リバースイテレータは実際のデータのお尻を指している。2つしか要素がないコンテナのイテレータを考えるとで、イテレータからデータを見るときに↑の矢印の方向にデータを見るので、 $it+1$ と $rit+1$ はイテレータの位置自体は同じなのですが、指し示すデータが違います。

このため $rit+1$ から $base()$ を使用して(リバースでない)イテレータにしたとき、指し示すデータは同じではなく、「次の」データを指します。



ともかく $base$ を使えばいいんですが、 $rit.base()$ が $end()$ でないとも限らないのでチェックはしておきましょう。

ともかくそれで補間してアニメーションするようにしてください。

できましたか？

あれ？

でも、何かおかしな感じしませんか？



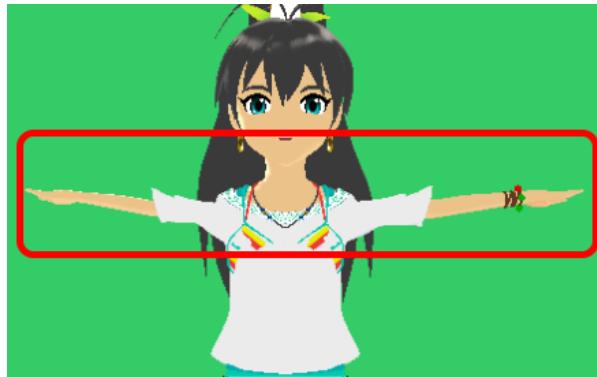
腕…短くね？

ミクさんは手が隠れてるので分かりづらいですね。



やはり短い

ということでもっとあからさまなモーション swing.vmd を用意してみたので見てみましょう。



これはひどい

ちなみにぼくはこういう実装にしてるんですけど。

```
void RotateBone(const char* bonename, const DirectX::XMFLOAT4& q, const DirectX::XMFLOAT4&
q2=DirectX::XMFLOAT4(), float t = 0.0f);
(中略)
void
Dx12Wrapper::RotateBone(const char* bonename,const DirectX::XMFLOAT4& q, const DirectX::XMFLOAT4&
q2 ,float t ) {
    auto& bonenode = _boneMap(bonename);
    auto vec = XMLoadFloat3(&bonenode.startPos);
    auto quaternion = XMLoadFloat4(&q);
    auto quaternion2 = XMLoadFloat4(&q2);
    _boneMatrices(bonenode.boneIdx) = XMMatrixTranslationFromVector(XMVectorScale(vec, -1))*
        (XMMatrixRotationQuaternion(quaternion)*(1.0f-t) +
        XMMatrixRotationQuaternion(quaternion2)*t) *
    XMMatrixTranslationFromVector(vec);
}
```

こういう関数を作つておいて…

```
auto frameIt = std::find_if(keyframes.rbegin(), keyframes.rend(), [frameno](const
Keyframe& k) {return k.frameNo <= frameno;});
```

```

if (frameIt == keyframes.end()) continue;
auto nextIt=frameIt.base(); //通常のイテレータに戻している(リバースを通常にすると自動的に
次の内容を示すことになる)
if (nextIt == keyframes.end()) {
    RotateBone(boneanim.first.c_str(), frameIt->quaternion);
}
else {
    float a = frameIt->frameNo;
    float b = nextIt->frameNo;
    float t = static_cast<float>(frameNo - a)/(b-a);
    RotateBone(boneanim.first.c_str(), frameIt->quaternion, nextIt->quaternion, t);
}

```

どこがいけないんでしょうかねえ…？

うん、まあこれ、知ってないと気づきづらいんで別に気づかなくても責めません。ただこういう「あからさまに短い」ようなときに「おかしいな」と思えるようにしてください。そういう「感覚」がないとゲームデザイナーさんとかアーティストさんとうまくやっていけません。

球面線形補間

で、

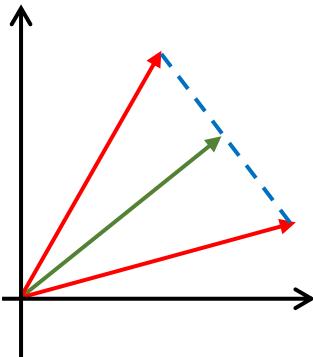
$$M = (1 - t)A + tB$$

というのは「線形補間」なんですが、これはマジ線形(1次関数)なんです。つまり直線的にしか動かない。直線的にしか動かないという事はつまり…

2つの点の間を直線で結んでその間を+として補間しているだけ…

図のように2つのベクトルの間のベクトルは長さが短くなっているのが…分かるだろう？

腕もまた…短くなるのだ。



これはちょっと特殊な補間をしてあげないといけないのだ。簡単に言うと極座標における「角度 θ 」を補間するイメージです。だから線形補間ってのはベクトルに対して行うと

$$A(x_A, y_A) \rightarrow B(x_B, y_B)$$

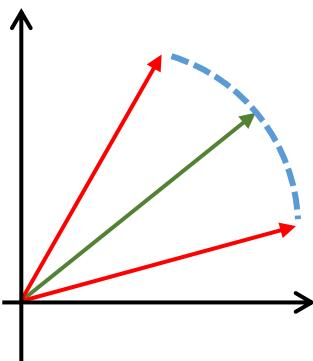
$$M = A(1-t) + Bt = (x_A(1-t) + x_B t, y_A(1-t) + y_B t)$$

これ↑が通常の線形補間で、球面線形補間ってのは

$$A(r_A, \theta_A) \rightarrow B(r_B, \theta_B)$$

$$M = A(1-t) + Bt = (r_A(1-t) + r_B t, \theta_A(1-t) + \theta_B t)$$

というわけです。極座標系で補間するのが球面線形補間です。腕などは長さが一定であることを考えると、角度の部分に対して $(1-t)$ と t を乗算して足すというイメージね。図で言うと



こんな感じになって自然な感じになることが分かります。じゃあどうすればいいんでしょうか？

http://marupeke296.com/DXG_No57_SheareLinearInterWithoutQu.html

と、自分でやってもいいのですが、クオータニオンは球面線形補間が得意なのです。何しろ角度の塊ですから。

まあ、そうは言ってもマイクロソフトさんが球面線形補間関数を作ってくれておりますので、そいつを使いましょう。

XQuaternionSlerp という関数です。

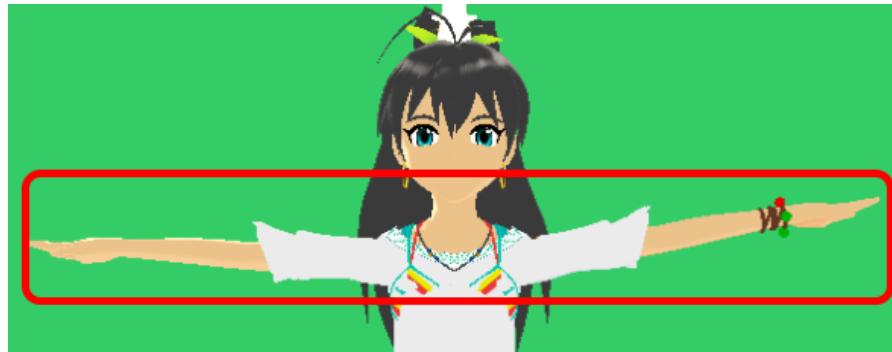
<https://msdn.microsoft.com/ja-jp/>

[jp/library/microsoft.directx_sdk quaternion.xmquaternionslerp.aspx](http://msdn.microsoft.com/jp/library/microsoft.directx_sdk quaternion.xmquaternionslerp.aspx)

XQuaternionSlerp(クオータニオン1, クオータニオン2, 補間係数+);
のように使用します。

つまり

XMMatrixRotationQuaternion(XQuaternionSlerp(quaternion, quaternion2, +))
とやってやれば…



まともな長さのが分かるかと思います
おめでとう。ここまであらかた再生は大丈夫だと思う。

VMD モーションデータの罠

VMD モーションデータは実は出力の際にソートなどは一切行われていないようです。つまり、ただただ「データを登録した順」にデータが並んでいます。

それを検証するためにわざと妙なデータを作りました。サーバの charge.vmd を開いてください。そしてアニメーション繰り返しとかしないようにしてみてください。

そうすると、特に対処を入れてないと



こういう中途半端なポーズで止まります(本当はチャージング GO ポーズなので片腕を上げているはず)

この原因は前回ちょっとだけお話しした「データは登録された順に並んでいるだけでソートされてない」というのが原因になっています。

例えば

0→30→60

と並んでいるところに後から 50 フレームを追加するとデータの並び的には

0→30→60→50

となり、50 フレーム時点での拳動が無視されてしまうわけだ。

ところが本来は

0→30→50→60

にならないとうまくいかないわけだ。

じゃあどうすればいいのかというと、…どうすればいい?

まあ順当に考えて「ソート」ですね。これも algorithm に頼ります。

<http://cppref.jp.github.io/reference/algorithms/sort.html>

というわけなんだが、この例のやり方ではうまくいかない。何故なら、どの数値を基準に並べ替えるのかが明記されていないからだ。

つまり…

```
template <class RandomAccessIterator, class Compare>

void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

こちら側…ソート関数まで明記されているソートを使う必要がある。でも既に `find_if` とかを使ったみんなには分かっていると思うがラムダ式を使えばカンタンなのである。

```
std::sort(begin(), end(), [](Keyframe& a, Keyframe& b){return a.frameNo < b.frameNo; });

というわけだな。分かるかな？
```

左側が小さければ `true`。そうでなければ `false`…つまり小さい順に並ぶということだ。

別に `vector` を `set` にしても構いません。その際にはちょっとだけまた面倒っちゃ面倒ではある…。軽く `set` について解説しておく `map` の `key+value` が `value` だけになった形である。`map` というのは `key` を基準として「赤黒木」というのを構築してソートしているのだが、`set` は `value` 自身を基準とするのだ。

ただ、`value` 側はたいていの場合ユーザー定義構造体になっている事が多く大小つけられないのでため普通に

```
struct Test{
    Test() {}
    Test(int f, float t) :frameno(f), test(t) {}
    int frameno;
    float test;
};
```

(中略)

```
std::set<Test> s;
```

なんて書くとコンパイラに怒られてしまいます。大小つけられへんやん!!!と。

じゃあどうすんねん…実は `Set` は比較関数をセットしないと、このように怒ります。怒りますが組み込み型の `int` や `float` では特に怒ることもなくデフォルトでソートしてくれます。これは予め `Lesser` という比較式が入っていて "`<`" で比較しているためです。

じゃあどうすればいいのかと言うと、いくつかやり方があります

- <演算子をオーバーロードする
- 比較式をラムダ式とかで突っ込む
- もう set は使わない

まず、<演算子をオーバーロードするって話ですが、これは簡単です。

```
bool operator<(const Test& a, const Test& b) {  
    return a.frameno < b.frameno;  
}
```

例えば frameno の大小でのみソートする場合は↑のように書けばいいのです。

ただ、これではそもそも Test 構造体の扱いに影響を及ぼしかねない…演算子をオーバーロードしちゃってるからね。ではオペレータオーバーロードではなく、もっと別の方法はないものだろうか…

実は set の第二引数は関数オブジェクトの「型」を指定できるようになっている。また、関数オブジェクトの実体は初期化時に引数として与えてやることができる。つまり宣言時に

```
set < Test, function<bool (const Test&, const Test&)>> s([](const Test& a, const Test&b)  
{return a.frameno < b.frameno;});
```

こういう事をやってやる。これもOKだ。

最後にだが、set を使わないという選択肢だ。実はソートで「並び変えられ」てさえいれば、アルゴリズムは問わない。スタイルも問わない。

このため「そもそも使わない」という選択肢がでてくる。ちなみにスピードはどうかというとおそらくソート済み vector の方が早い。

毎フレームキーフレームが更新されるならともかく、キーフレーム自体は更新されないと読み込んで最初の一発目にソート関数をかけておけば良いだろう。

恐らく、ここまでやれば、思った通りに動くことだろう…。長かった。本当に長かった。

もうちょっとだけ続くんで待ってください。ゲームは自主的に作っておいてください。

さてあとはオマケみたいになもんだけど「補間曲線」についてやっていこうと思います。

補間曲線

MMD の左下のこの部分…



これはベジエ曲線や。そこまではええで。そしてこの情報は VMD の中に入っとるやで。
ええねん…それはええねん。

ちなみに『通りすがりの記憶』の VMD の説明では

//// 補間の補足 ////

BYTE Interpolation(64); // (4)(4)(4) // 補完

// 補間用の曲線

// (0,0), A(ax,ay), B(bx,by), (127,127) の 3 次(4 点)ベジエ

// A:左下の+, B:右上の+

/

// モーションの補間パラメータの並び順(MMD 板の情報)

// 回転は 4 軸(ウォータニオン)だが、1 個にまとめられているので注意

// X 軸 Y 軸 Z 軸 回転

// A(Xax,Xay)(Yax,Yay)(Zax,Zay)(Rax,Ray)

// B(Xbx,Xby)(Ybx,Yby)(Zbx,Zby)(Rbx,Rby)

// とすると、

// Xax,Yax,Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,

// Yax,Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,

// Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,00,

// Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,00,00

こんな感じで書いてます。正直面倒くさいやで。

何故このような記録のされ方をしているのか分からぬ。結局データは重複している(Xax や
ら Xbx やらね)し、何をしたいのかさっぱりわからぬ。unsigned char* 64 個のデータなんだが、
どう見ても Xax や Xbx などが 4 つもかぶっているのがわかると思う。こんな感じで 64 バイト
使うのならば一つ一つの数値を float(4 バイト)にして、それを 16 個使ったほうがよっぽど良
いんじゃないでしょうか…。

まあ、いいや。ともかく↑の 64 個のデータにおいてボーン回転に使われているのは実は

Rax, Ray, Rbx, Ry だけである。2 次元 3 次ベジエのために必要な最小限のデータです。

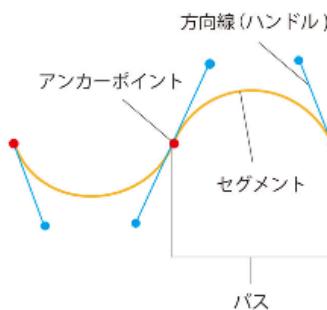
ちなみに一つ一つの取りうる値が 0~255 までなので、別にキャラ型でも構なしつつも構わないですね。

ところでここで困ったことがある。…その前にベジエ曲線といいのは CG 的にはご存じだろうか？まあ、要は何に使われているものかっていう話だけど…一応フォトショップや GIMP で使われているし、フォントデータといいのはベジエデータである。

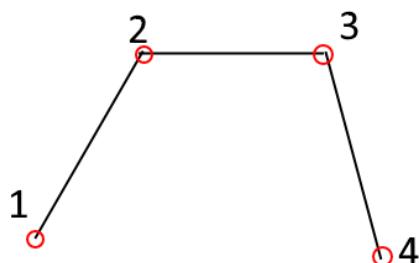
ベジエ曲線…とは？

何かといふと平面上(空間上)4 点から曲線を構成するというのだ。ツールなどでは「曲線をきれいに描くためのツール」として知られている。

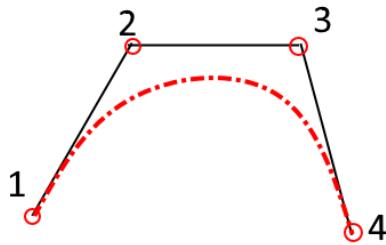
通常であれば 4 点しかなければならぬものは 3 つの直線でしかない。閉じた系なら 4 つの直線だが、別にそこは大差ない。



図のようにアンカーポイントとハンドルで構成されており、生成される曲線はアンカーポイントを通るが、ハンドルは通らない。これ CG 検定でも出るぞ？ 基本的には 4 点であれば直線なら 4 点をきっちり通るよね？

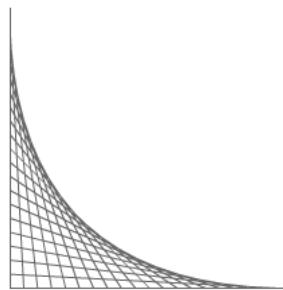


ところがベジエ曲線(3 次)の場合、通るのは 1 番と 4 番のみで、間の 2~3 は通らない。通るための点ではなく「計算のための点」として扱われる。結果として…



こうなります。

ちなみに自分で書くこともできますが…



これは2次ベジェですが定規1本で書けます

ぶっちゃけ3次ベジェも定規1本あれば書けます

分かりやすい解説だと

<http://blog.sigbus.info/2011/10/bezier.html>

があるが、この三次ベジェになっているのがMMDの補間曲線だ。数式で書けるのだが、意外と難しくはない。

理屈的には線形補間とほぼ同じである。 $M = (1 - t)A + tB$ が理解できていれば問題ない。

さっきの4点を $P_1 \sim P_4$ とすると

$$M = (1 - t)^3 P_1 + 3(1 - t)^2 t P_2 + 3(1 - t)t^2 P_3 + t^3 P_4$$

と言う式になります。ちなみに2次ベジェだと

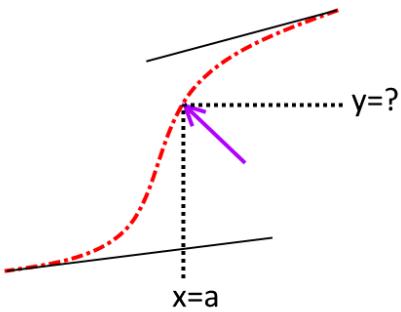
$$M = (1 - t)^2 P_1 + 2(1 - t)t P_2 + t^2 P_3$$

になります。なんとなく法則性はわかりますよね？

さて、今回は三次ベジェだから

$$M = (1 - t)^3 P_1 + 3(1 - t)^2 t P_2 + 3(1 - t)t^2 P_3 + t^3 P_4$$

で座標は分かるんですが、肝心の媒介変数 t がわからない!。 t がわかれれば x も y もわかるのだが、知りたいのはそれではないのだ。



例えばこういう曲線において $x=a$ だった時の y 座標を知りたいときはどうだろう
式で表せているのだから $y=f(x)$ の式にできそうなものであるが…。たとえば $x=0.5$ の時を想像してもらえば分かるが、簡単な定数の時ですら $y=f(x)$ を求めるのは困難である。
もちろん 0 の時と 1 の時は固定であるから $x=0, 1$ のとき $y=x$ である。だがそれ以外の時は求めようがないのだ。

ここで重要なのが「近似値」の考え方である。今までの皆さんには方程式と言えば「答えが…」はっきりした答えが出るものがかりを扱ってきたんだろう？ 例えば

$$ax^2 + bx + c = 0$$

を満たす x は

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

と言う風にはっきりとイコールで結ばれる値だ。これをはじき出すのが困難である場合に「近似値」は非常に役に立つのだ。

ニュートン法(ニュートン・ラフソン法)

x から y (今回は t だけ)を高速に計算したい…そんなお客様のご要望にお応えするのがニュートン法(もしくは二分法)という近似計算です。

ニュートン法は微分を応用した近似計算法です。

<https://ja.wikipedia.org/wiki/%E3%83%8B%E3%83%A5%E3%83%BC%E3%83%88%E3%83%B3%E6%B3%95>

x から最終的には y を求めたいので、一旦 x と y を分離して考えます。

目的は

$$x = (1-t)^3 x_1 + 3(1-t)^2 t x_2 + 3(1-t)t^2 x_3 + t^3 x_4$$

を満たす t を求めることです。何故かと言うと変数は x ですが、それを満たす t が分からな

いと γ が求まらないからです。

γ が求まればあとは

$$y = (1-t)^3 y_1 + 3(1-t)^2 t y_2 + 3(1-t)t^2 y_3 + t^3 y_4$$

の y は↑の式にその γ の値を入れてやれば求まるわけです。

なお、3次方程式の一般解は

$$\begin{aligned} x &= \frac{\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3}{3\sqrt[3]{2}a} - \\ &\quad \frac{\sqrt[3]{2}(3ac - b^2)}{3\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3} - \frac{b}{3a} \\ x &= -\frac{1}{6\sqrt[3]{2}a}(1 - i\sqrt{3}) \\ &\quad \frac{\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3 +}{3 \times 2^{2/3}a\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3} - \\ &\quad \frac{(1 + i\sqrt{3})(3ac - b^2)}{3 \times 2^{2/3}a\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3} - \\ x &= -\frac{1}{6\sqrt[3]{2}a}(1 + i\sqrt{3}) \\ &\quad \frac{\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3 +}{3 \times 2^{2/3}a\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3} - \\ &\quad \frac{(1 - i\sqrt{3})(3ac - b^2)}{3 \times 2^{2/3}a\sqrt[3]{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3} - \\ x &= \frac{b}{3a} \end{aligned}$$

となっており、死ぬほど複雑なことがお分かりいただけだと思います。

ちなみに、 4×4 行列の逆関数も死ぬほどややこしく実用的には「ガウスの消去法」というテクを用いて求めるのですがこういふ「死ぬほどややこしい」時はそういう毒いテクニックや近似を用いることになります。

ニュートン法の説明を見ると、こういう風にして近似値を求めるらしいです。

$$t_{n+1} = t_n - \frac{f(t_n)}{f'(t_n)}$$

これは漸化式と言いますが、漸化式って知っていますかね…数列とかで出てくるんですが…まあ、大した話じゃないです。例えば

$$a_{n+1} = a_n + 1$$

だとすると数列が $1, 2, 3, 4, 5, 6, \dots$ となる。これは分かるかな? ゲームプログラミングにおいては漸化式的な計算によって難しい数式を簡単に表現する事をよくやっています。

例えばジャンプするという場合、高校の物理では

$$y = \int_{t_s}^{t_e} v - gt \ dt = \left[vt_e - \frac{gt_e^2}{2} - \left(vt_s - \frac{gt_s^2}{2} \right) \right]$$

みたいに説明されてて、特に $t_s = 0$ としてかかった時間を t として公式

$$y = vt - \frac{gt^2}{2}$$

としていますが、これって結局ものごとつ小さい+に対して

v_0 = 初速度

として

$$v_{n+1} = v_n - g$$

$$y_{n+1} = y_n + v_n$$

としてるようなもんなのね。これ自体は皆さんのがゲームループでやっている事と一緒に思いませんか？

ちなみに今回使う「微分」は超初心的な話なので皆さんにも分かっていただけたと思います
…たぶん。

微分ってのはひとことで言うと「2次曲線とかり次曲線の『接線の傾き』を求めるためのものです」

その計算も非常に簡単で

$$f(x) = ax^n + bx^{n-1} + cx^{n-2} \dots wn + q$$

ってな式の場合、乗数を係数として掛け算してやって、さらに乗数を1引きます。つまり

$$(x^a)' \rightarrow ax^{a-1}$$

元の数のマイナス1

こんな感じの計算を全ての要素について計算してあげます。そうするとさっきの例だと

$$f'(x) = nax^{n-1} + (n-1)bx^{n-2} + (n-2)cx^{n-3} + \dots w$$

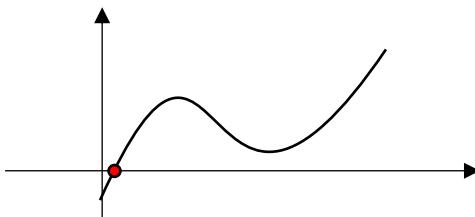
となります。なお、 x_0 の時の傾きを求めたければ

$$\text{傾き}_0 = na(x_0)^{n-1} + (n-1)b(x_0)^{n-2} + (n-2)c(x_0)^{n-3} + \dots w$$

となります。なお、

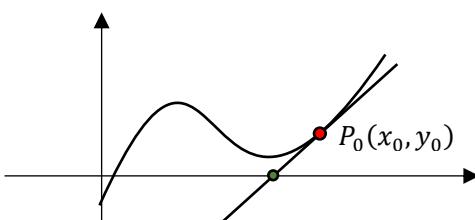
x_0 は定数ですので、傾きも定数で算出されるというわけです。

基本的な微分の話が終ったところで、ニュートンラフソン法というのは、これを用いて、方程式の解を近似するものなのですが、



例えばこういう3次曲線とX軸との交点を求めたいとします。三次曲線の解を求めるのがかなりやつかりです。ここでニュートンラフソン法の出番です。こいつの目的はひとまずはn次関数の関数のグラフとX軸との交点を求める 것입니다

- ①曲線上に適当な点を打ちます。
 - ②その点の傾き(微分で求まる)から直線の式を求めます
 - ③直線の式からX座標との交点を求めます
 - ④③の点のY座標が0でないなら①に戻ります
- というわけです。図で示すと以下のようになります



適当な点を打ち接線を求めます。

そうすると図のようにY軸との交点がわかりますね？ちなみに交点は、上図のY座標がOになるところで、適当な点P₀(x₀, y₀)における傾きはa = f'(x₀)で、X座標はx₀であるためy₀ = f(x₀)です。ところで直線の方程式は皆さんご存じ

$$y = ax + b$$

で、特定の点(x₀, y₀)を通る別式は

$$y - y_0 = a(x - x_0)$$

です。今回の「適当な点」から伸びた直線を考えると

$$y - f(x_0) = f'(x_0)(x - x_0)$$

です。そして、それがX軸(y=0)と交点を持つとすると

$$0 - f(x_0) = f'(x_0)(x - x_0)$$

という式が成り立ちます。これをxの式にします。

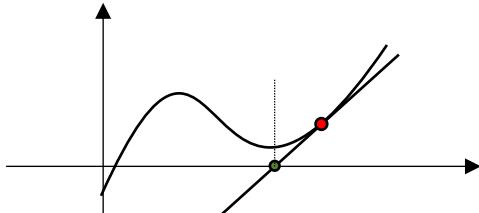
$$-\frac{f(x_0)}{f'(x_0)} = x - x_0$$

となりますので移項すると

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

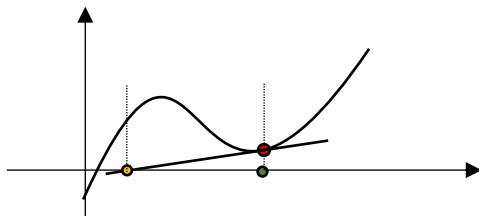
となり、接線が x 軸と交差する点が求まります。ここまで読んで、何のためにこんなけつたいね計算をやってるのか疑問に思う人もおられると思いますので、図を用いて説明していきます。とりあえずここまで説明で「適当な点から引いた接線と x 軸との交点を求める」という事をやろうとしている事じたしは伝わったかな~と思いますので、その前提で説明します。

接線と x 軸との交点を求める作業を繰り返すと何が起きるのでしょうか…?

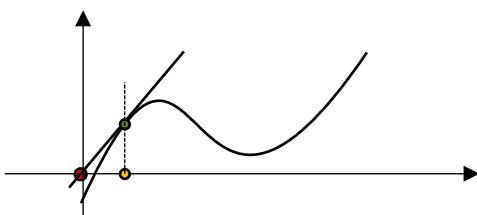


まず、先ほど計算した x 軸との交点における x 座標が分かります。この x 座標を x_1 とします。そうすると y 座標は $f(x_1)$ です。そこで先ほどと同じ計算を行います。

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$



結果として真の値に近づこうとしているのがわかりますね？



もちろん必ず順調に近づいていくと言う訳ではありませんが、基本的には真の値に近づいていきます。もちろん近似であるため完全に一致するわけではなく、ある程度近づいた時点で計算を打ち切ります。

「…あれ？なんか騙されそうになってたけど、今は t を求める話だったよね？なんで x を求める話になんてんの！？」

と思われるかもしれません、元々の目的を思い出してください。

$$x = (1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 x_3$$

を満たす t を求めることがでした。つまり x は分かっているし、 $x_0 \sim x_3$ は定数です。そうすると

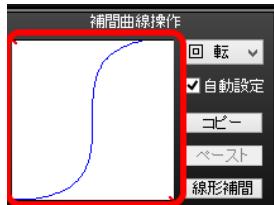
$$(1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 x_3 - x = 0$$

となります。ややこしいですがもう少しだけ我慢してください。↑の式は

$$At^3 + Bt^2 + Ct + D = 0$$

の形になることが分かります。そしてこの式の結果が 0 になるような t をニュートン法を用いて求めねばいいわけです。

そして、MMD のフレーム間補間ベジエの場合この式はもう少し簡単になります。もう一度エディタのベジエの部分を見てください。



実は端点である P_0 と P_3 の座標は動かすことができません。 $P_0 = (0,0)$ ですし $P_3 = (1,1)$ です。これを先ほどどの式に当てはめてみます。

$$(1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 x_3 - x = 0$$

この式は

$$(1-t)^3 0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 1 - x = 0$$

ですから

$$t^3 + 3t^2(1-t)x_2 + 3t(1-t)^2x_1 - x = 0$$

となり、 t の次数でまとめると

$$t^3(1 + 3x_1 - 3x_2) + t^2(3x_2 - 6x_1) + t(3x_1) - x = 0$$

となります。あとはこの式をニュートン法で解いた時の t を求めねばいいわけです。サンプルを

https://github.com/boxerprogrammer/Mathematics/tree/master/Bezier_3/BezierTime

に置いておりますので、参考にされたい方は見ておいてください。

二分法

ニュートン法と同じような役割を果たすものに 2 分法という手法もあります。これはニュートン法の目的と同様に $f(x) = 0$ となる x を探すものです。今回の件で言うと $f(t) = 0$ ですね。この方法は理屈自体はすごく簡単ですので、こちらのほうが納得いくかもしれません。ニュートン法に比べて収束が遅いという特徴があります。

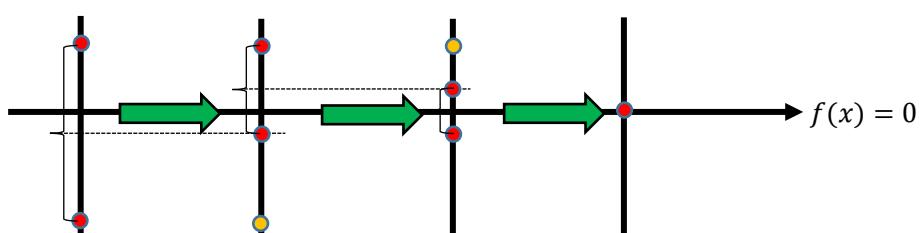
実はこの方法いろいろと難はあるますが、確かにニュートン法よりも理解しやすいため、まずは解説します。

まず $f(x) = 0$ を挟んでしまうような $f(a)$ および $f(b)$ を考えます(式が分からぬ以上不明なのですが、ひとまずこのまま読み進めてください)

$f(x) = 0$ となる x を求めるのが目的であるため $f(a)f(b) < 0$ とします。(掛け算の結果が負 \rightarrow $f(a)$ と $f(b)$ の符号が違う $\rightarrow f(x) = 0$ を挟む) というわけです)

上記を満たすような初期値 a, b が見つかったら次に $f\left(\frac{a+b}{2}\right)$ を計算します。 $f\left(\frac{a+b}{2}\right)$ の符号を確認し、 $f(a)$ が $f(b)$ のうち、符号が同じものと入れ替えます。

これにより挟み撃ち的に段階的にプラスとマイナスから段々 $f(x) = 0$ に近づくはず…というのが二分法のアルゴリズムです。図にすると以下のような形で収束するはずです



図のようにある程度 $f(x)=0$ に近づくか、あらかじめ決めておいた「繰り返し回数」に到達したら計算を打ち切るという事になります。

$f(x) = 0$ という目的地はニュートン法と同じですが収束へのアプローチが違いますね。ちなみにニュートン法よりも収束速度は遅いです(つまりループ回数を多く要求されます)。これだけを言うとニュートン法一択の気がしますが、ニュートン法のほうは式と初期値によっては収束しないことがあります。

で、2分法のほうは実装をしようと考えると一つ問題が発生します。それは「 a と b をどうやって決めるのか」という問題です。

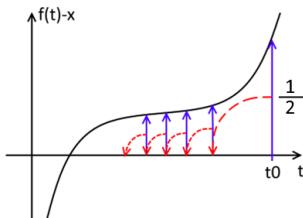
最初の前提として $f(a)$ と $f(b)$ で 0 を挟まないといけないので適当に選んだ結果が $f(a)f(b) > 0$ だと詰みです。というわけで今回の問題に限定して少し工夫を加える必要があります。

工夫と言うほどのものではないのですが、意識すべきことが今回の場合は値域が $0.0 \leq x \leq 1.0$ で定義域が $0.0 \leq t \leq 1.0$ です。よって挟み撃ちの開始位置は決め打ちで $a=0.0f, b=1.0f$ とします。多少収束までに時間がかかるかもしれません、仕方ないですね。

これまた先ほどの GitHub にソースコードを上げていますが、なかなか収束しません。収束の速さはニュートン法に軍配が上がりますので、特に問題が起きない限りニュートン法が妥当な気はしますが、今回の件限定で何かもっといい方法はないのでしょうか…?

もう一つおまけの方法(半分刻み法【勝手に命名】)

これは独自の方法(DxLib のソースコードを参考にしました)ですが本当に今までの話からすると、かなり分かりやすい方法だと思います。おそらく収束はニュートン法に負けますが、とにかく分かりやすいです。まず先ほどの式の $f(t)$ の結果を見ます。そして $f(t)$ の半分を現在の



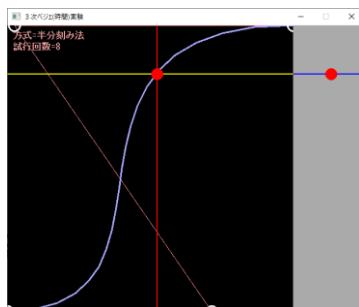
t から引いたものを次の t とします。

これを繰り返していくとそのうち嫌でも $f(t)=0$ の点に近づいていくと言う訳です。理屈はこうです。

- プラスだったらマイナス方向へ、マイナスだったらプラス方向へ進む
- 高さの半分であるため 0 との交点があれば必ずそのうち 0 に収束する
- 大きく離れているときは大きく動くため収束の効率がそれなり

というわけです。ちなみにニュートン法の時にご紹介した実験プログラムにおいてはこの方法はニュートン法と同じくらいの性能を示しました。計算回数は少ないですしこれでいいかなと思います。

https://github.com/boxerprogrammer/Mathematics/tree/master/Bezier_3/BezierTime



さて、理屈がお分かりになったと思いますので、いよいよ実装していきましょう。

ベジェによるイーズインイーズアウトを実装してみよう

実験プログラムの検証の結果「半分刻み」で 12 回以上試行が安定しているようなので、半分刻み 12 回試行で実装していきましょう。

プログラムは先ほどの式をそのまま使えばいいのでシンプルです。実装場所は PMDActor の private 関数でいいでしょう。今のところはここ以外で使用することは想定していませんし。

float

```

PMDActor::GetYFromXOnBezier (float x, const XMFLOAT2& a, const XMFLOAT2& b, uint8_t n) {
    if (a.x == a.y && b.x == b.y) return x; //計算不要
    float t = x;
    const float k0 = 1 + 3 * a.x - 3 * b.x; //t^3の係数
    const float k1 = 3 * b.x - 6 * a.x; //t^2の係数
    const float k2 = 3 * a.x; //tの係数

    //誤差の範囲内かどうかに使用する定数
    constexpr float epsilon = 0.0005f;
    //tを近似で求める
    for (int i = 0; i < n; ++i) {
        //f(t)求めまーす
        auto ft = k0 * t*t*t + k1 * t*t + k2 * t - x;
        //もし結果が0に近い(誤差の範囲内)なら打ち切り
        if (ft <= epsilon && ft >= -epsilon) break;
        t -= ft / 2; //刻む
    }
    //既に求めたいtは求めているのでyを計算する
    auto r = 1 - t;
    return t * t*t + 3 * t*t*r*b.y + 3 * t*r*r*a.y;
}

```

このような形となります。あとは補間部分に組み込んでいくだけです。

MotionUpdate 内で t を普通に現在の時間から求めている箇所があると思います。

```

auto t = static_cast<float>(frameNo - rit->frameNo) /
    static_cast<float>(it->frameNo - rit->frameNo);

```

ここにベジエによる補間を追加します。しかしながら、越えなければならない問題があります。それはベジエの制御点 P_1 および P_2 をどうやって得るかという事です。これは VMD の仕様によるとキーフレームデータのクオータニオンの直後にある 64 バイトのベジエデータから取得します。

ただし、非常に面倒な配置になっています。ボーンの変形に対するすべての変化に対してベジエ設定が存在するためこのような面倒なことになっているのです。

ボーンにかかるベジエの情報で含まれているのは大きく2つ

- X,Y,Z 方向の平行移動に関するベジエ補間
- 回転に関するベジエ補間

「平行移動」ってなんだよ。ボーンは回転だけだろ!!と思われると思いますが、これは IK(インパ

ースキネマティクス)を用いた際にソルバ(コントロールポイントみたいなもの)の平行移動によって回転を制御するため、このソルバの動きは平行移動となります。このためこのようなパラメータがあるのです。IK に関しては後の章で説明しますので、今は回転の事だけ考えてください。

ちなみにベジエの2次元的座標($0 \leq x \leq 1, 0 \leq y \leq 1$)によって補間具合が決まりますので、必要なのは回転に関する制御点データ

$$R_1 = (x_{R1}, y_{R1}) \quad R_2 = (x_{R2}, y_{R2})$$

の4データのみとなります。それ以外は

$$X_1 = (x_{X1}, y_{X1}) \quad X_2 = (x_{X2}, y_{X2})$$

$$Y_1 = (x_{Y1}, y_{Y1}) \quad Y_2 = (x_{Y2}, y_{Y2})$$

$$Z_1 = (x_{Z1}, y_{Z1}) \quad Z_2 = (x_{Z2}, y_{Z2})$$

なおベジエデータ 64 個の並びは以下のような形になっていて

x_{X1}	x_{Y1}	x_{Z1}	x_{R1}	y_{X1}	y_{Y1}	y_{Z1}	y_{R1}
x_{X2}	x_{Y2}	x_{Z2}	x_{R2}	y_{X2}	y_{Y2}	y_{Z2}	y_{R2}
x_{Y1}	x_{Z1}	x_{R1}	y_{X1}	y_{Y1}	y_{Z1}	y_{R1}	x_{X2}
x_{Y2}	x_{Z2}	x_{R2}	y_{X2}	y_{Y2}	y_{Z2}	y_{R2}	0
x_{Z1}	x_{R1}	y_{X1}	y_{Y1}	y_{Z1}	y_{R1}	x_{X2}	x_{Y2}
x_{Z2}	x_{R2}	y_{X2}	y_{Y2}	y_{Z2}	y_{R2}	0	0
x_{R1}	y_{X1}	y_{Y1}	y_{Z1}	y_{R1}	x_{X2}	x_{Y2}	x_{Z2}
x_{R2}	y_{X2}	y_{Y2}	y_{Z2}	y_{R2}	0	0	0

といふなんともけつたいな並びになっております。キャラの動きの補間に利用するには 1~2 行目だけで充分です。つまり最初の 16 バイトですね。さらに今回は回転だけ考えればいいため、配列番号 3,7,11,15 が必要なデータとなります。そこでまずはキーフレーム構造体にベジエ情報を追加します。

(実は 1 段目のデータがどうも想定と違うっぽいので、2 段目から取得したほうがいいですね) ちなみにベジエの座標値は($0 \leq x \leq 127$) で($0 \leq y \leq 127$) と、これまた中途半端(おそらくは char の取りうる値で範囲を決めていると思われる)なようですが、データがそうである以上従わざるをえないでの従います。

ということで、構造体にベジエの位置を

//キーフレーム構造体

```
struct KeyFrame {
    unsigned int frameNo; //フレームNo.(アニメーション開始からの経過時間)
    DirectX::XMFLOAT2 p1, p2; //ベジエの中間コントロールポイント
    DirectX::XMFLOAT3 q; //ベジエの回転
```

```

KeyFrame(unsigned int fno, XMVECTOR& q, const XMFLOAT2& ip1, const XMFLOAT2& ip2):
    frameNo(fno),
    quaternion(q),
    p1(ip1),
    p2(ip2){}
};

そして VMD データ読み込みの際に
//VMDのキーフレームデータから、実際に使用するキーフレームテーブルへ変換
for (auto& f : keyframes) {
    _motiondata[f.boneName].emplace_back(KeyFrame(f.frameNo,
        XMLoadFloat4(&f.quaternion),
        XMFLOAT2((float)f.bezier[3+15] / 127.0f, (float)f.bezier[7+15] / 127.0f),
        XMFLOAT2((float)f.bezier[11+15] / 127.0f, (float)f.bezier[15+15] / 127.0f)));
}

```

とでもやってやります。これでベジエの中間点を得ることができました。で、得られた p1,p2 を用いてベジエ補間をかけてやります。既に作っている GetYFromXOnBezier を用いて先ほどの t 値を求めた式の部分にコードを追加して

```

auto t = static_cast<float>(frameNo - rit->frameNo) /
    static_cast<float>(it->frameNo - rit->frameNo);
t = GetYFromXOnBezier(t, it->p1, it->p2, 12);

```

と書いてやればベジエ補間を考慮した動きとなるでしょう。

その他今のうちにやっておきたい事

可変フレームレート状態で拳動を合わせる

DxLib では画面更新の際にフレーム待ちしてくれるんで、よほどの処理落ちをしない限りは PC によって拳動が変わることはありません。しかし現在の実装では待ちを入れないため、各 PC でものごつ高速になつたりします。

少なくとも MMD の速度(30fps)に合わせて動かしたい

30fps ということはつまり…

1 秒間(1000 ミリ秒)あたりに 30 フレーム動くので $1000/30=33.333333333\dots$ である。

ということは経過ミリ秒/33.33333=経過フレーム数となる。このため

```
static auto lastTime = GetTickCount();  
MotionUpdate(static_cast<float>(GetTickCount() - lastTime) / 33.3333f);
```

とすれば、フレームレート可変のまま動きは一定になる。なお、↑のやり方だとループにならないので、ループにするにはモーションの長さに 33.33333f をかけたところで lastTime を更新すればいい。

```
if (GetTickCount() - lastTime > _motion->Duration() * 33.3333f) {  
    lastTime = GetTickCount();  
}
```

なお、GetTickCount()は「精度が低い」と言われる事が多いため、気になる人は timeGetTime()を使えばいいと思います（winmm.lib のリンクが必要）。さらに精度を求めたかつたら QueryPerformanceCounter を使用すればいいと思います。

これ以上説明するのが面倒なのでぼくは使わないです。各自調べてやつといてください。

マルチパスレンダリング

ああ～ついにここまでやってきてしもたんじやああ～～びっくりするねえ…。ここでいう「マルチパスレンダリング」は一度普通にドッファにレンダリングしておいてその内容をテクスチャとして利用するというものです。

実際の話、今レンダリングしてそのキャンバスが既にテクスチャなので、怖がることはあります。

ただ、レンダリング結果をテクスチャにするという事は、そのテクスチャを表示するためのペラポリゴンも必要なのでメンドクサイっちゃめんどくさいです。ただ、これ覚えとくと、テクスチャに対して色々と加工する⇒レンダリング結果に加工するのと同じ。

というわけで、画面の色合いを変えたり、画面割ったり、画面ぼかしたり、レンダリング結果をさらにゲーム内オブジェクト（テレビ画面など）にハメコミ合成したり。

そういう事ができるようになるので、一気に表現の幅が広がります。

ちなみに後述する、影をオブジェクトに落とす「シャドウマップ」もこのマルチパスレンダリングがあるからこそ実装可能なのです。

余談ですが、朝の弱い僕がなぜか4時に起きてこの原稿を書いています。

まあ無駄話はこれくらいにして、実際にやっていきましょう。行きすぎィ。

大雑把な解説

マルチパスのキモは何度もレンダリングを行う所にあります、レンダリング結果をテクスチャとして使用するという事で、一つのリソースに対して RTV と SRV の二つの見方をするという所です。



結構これがややこしくハマる人がいるので油断しないようにしてください。

最初の実践

まず、画面の大きさの RGBA テクスチャを作ってください。今のみんななら大したことないと思います。そしてレンダーターゲットビューおよびシェーダリソースビューを作ります。

↑の RGBA テクスチャを RTVくんと SRVくんで使用しあう事になります。

次に現在の SetRenderTarget の書き込み先をその作ったレンダーターゲットビューに変更します。

で、レンダリング。当然何も映らない。何故かと言うとさっき書き込んだのは画面(バック/ドッファ)ではなく、テクスチャとして使用する画像だからだ。ここまでが 1 パス目。

ここから以降は、↑で 1 パス目レンダリングした後の話になります。ポリゴンレンダリング先がテクスチャなので、そのテクスチャを何かに張り付けて、それをまたレンダリングすればいい事がわかります。一番最初にテクスチャ貼ってペラボリ表示したこと思い出してください。

なので、そのテクスチャを貼り付ける先(ペラポリ)を4頂点で作つとります。

次にそのペラペラポリゴン(最初に作ったやつ)を用意します。

ペラペラポリゴンを画面上に表示するプログラムを作ります。レイアウトやパイプラインステートが変わるので、それも必要分用意します。

1つのレンダーターゲットにペラポリゴンをレンダリング

ペラペラポリゴンに先ほどのテクスチャを貼り付ける。

終わり。

…え？ 雜スギル？

モーしょーがなれいなーのびたくんはー

ビュー用ヒープ作る

さっきも言ったように、2つのビューが必要なので作っておきます。

```
D3D12_DESCRIPTOR_HEAP_DESC descHeapDesc={};  
descHeapDesc.Flags=D3D12_DESCRIPTOR_HEAP_FLAG_NONE;  
descHeapDesc.NodeMask=0;  
descHeapDesc.NumDescriptors=1;//どうせ1個ずつしか使わないんで…  
descHeapDesc.Type=D3D12_DESCRIPTOR_HEAP_TYPE_RTV;//あっ、そつかあここで指定するから  
...  
auto result=_dev->CreateDescriptorHeap(  
    &descHeapDesc,  
    IID_PPV_ARGS(_heapFor1stPathRTV.GetAddressOf())  
,  
  
descHeapDesc.Type=D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;//これを忘れるなよ  
  
result=_dev->CreateDescriptorHeap(  
    &descHeapDesc,  
    IID_PPV_ARGS(_heapFor1stPathSRV.GetAddressOf()));
```

真ん中のタイプ切り替えを忘れるといきなり落ちますのでよ～く見とけよ見とけよ～。
また、一つのヒープに RTV と SRV を混在することはできないので、RTV 用と SRV 用二つ用意します。

リソース作る

これはバッファと同じサイズでいいのでその大きさで作っておいてください。なお、その際には RESOURCE_DESC を D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET にしないとビュー作成時に失敗するので注意しておいてください。

ビュー作る

あとはその1つのバッファに対して二つの見方(ビュー)を作るだけです。
割とそのままでいいです。

レンダーターゲットビュー

```
_dev->CreateRenderTargetView(バッファ, nullptr, _heapFor1stPathRTV->GetCPUDescriptorHandleForHeapStart());
```

シェーダリソースビュー

```
//シェーダリソースビューを生成
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Format = desc.Format;
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MipLevels = 1;
_dev->CreateShaderResourceView(バッファ, &srvDesc, _heapFor1stPathSRV->GetCPUDescriptorHandleForHeapStart());
```

こんな感じで特にクラッシュしなければ大丈夫でしょう。

レンダーターゲット切り替え

レンダーターゲットをさっきのビューに切り替えてください。あ、もちろん深度バッファは使うので、それは前の奴を設定しといてください。

```
//111ス目
```

```
_cmdList->OMSetRenderTargets(1,&_heapFor1stPathRTV->GetCPUDescriptorHandleForHeapStart(),false,dsvs);
```

当然のように画面からモデルさんが消えます。描画先が違うんですから当然ですね。ここまでが1パス目です。

ペラポリ作る

1枚長方形ポリゴンを作ります。横-1～1。縦-1～1。Z値0あたりでつくります。座標情報とUV情報だけで十分だよなあ？

昔作ったやつが残ってたらそれでええんちゃいますやろか。なくなってたら流石にそこは自分で考えて、どうぞ。

```
Vertex vertices[]={  
    XMFLOAT3(-1,-1,0),XMFLOAT2(0,1),//正面  
    XMFLOAT3(-1,1,0),XMFLOAT2(0,0),//正面  
    XMFLOAT3(1,-1,0),XMFLOAT2(1,1),//正面  
    XMFLOAT3(1,1,0),XMFLOAT2(1,0),//正面  
};  
//ペラバッファ生成  
auto result = _dev->CreateCommittedResource(  
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),  
    D3D12_HEAP_FLAG_NONE,  
    &CD3DX12_RESOURCE_DESC::Buffer(sizeof(vertices)),  
    D3D12_RESOURCE_STATE_GENERIC_READ,  
    nullptr,  
    IID_PPV_ARGS(&_peraVB)  
);  
  
_peraVBV.BufferLocation = _peraVB->GetGPUVirtualAddress();  
_peraVBV.SizeInBytes = sizeof(vertices);  
_peraVBV.StrideInBytes = sizeof(Vertex);
```

頂点情報はこれで大丈夫かなと思います。

ペラポリ表示用ルートシグネチャを作る

えーと、また1から作ると思ってください。ですからシェーダもペラ用シェーダを作るとお考え下さい。

なので、レジスタ番号もまた0番から。

今回はテクスチャ1枚を使いたいだけなので、

```
D3D12_ROOT_PARAMETER peraparam = {};
peraparam.ParameterType = D3D12_ROOT_PARAMETER_TYPE_SRV;
peraparam.Descriptor.RegisterSpace = 0;
peraparam.Descriptor.ShaderRegister = 0;
peraparam.ShaderVisibility = D3D12_SHADER_VISIBILITY_PIXEL;
```

というルートパラメータを作り新しくperaRS(ペラルートシグネチャ)を作ります。

これでOKだ…そんな風に思っていた時期もありました。

いや、ルートシグネチャ自体は通るんですが、パイプラインステートでミスマッチとか云々言われるんですわ。

この原因といふか、もうルール的にダメなのは決定的なので、その根拠を確かめるべく

<https://docs.microsoft.com/en-us/windows/desktop/direct3d12/using-descriptors-directly-in-the-root-signature>

見てみました。やっぱり公式を…確実やな!!

一応該当すると思われる部分は(Google日本語翻訳だと)

「ルートシグネチャでサポートされている唯一のタイプのディスクリプタは、SRV/UAVフォーマットに32ビットのFLOAT/UINT/SINTコンポーネントしか含まれていなければなりません。ソースのCBVおよびSRV/UAVです。フォーマット変換はありません。ルートのUAVには、それらに関連付けられたカウンタはありません。ルート署名の記述子は、個別に個別の記述子として表示されます。これらの記述子は、動的に索引付けすることはできません。」

一応原文も

The only types of descriptors supported in the root signature are CBVs and SRV/UAVs of buffer resources, where the SRV/UAV format contains **only 32 bit FLOAT/UINT/SINT components**. There is no format conversion. UAVs in the root cannot have counters associated with them. Descriptors in the root signature appear each as individual separate descriptors - they cannot be dynamically indexed.

などとあり、恐らく Texture2D はこの対象外ではないかと思われます(だから mismatch などと言っていた可能性が…?)

また、ちょっと気になったのがちょっと上に書いてあった部分…

(日本語訳)

『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ネ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、ルートシグネチャの領域(ルートシグネ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。』

ん?

これって…

Applications can put descriptors directly in the root signature to avoid having to go through a descriptor heap. These descriptors take a lot of space in the root signature (see the root signature limits section), so applications have to use them sparingly.

ああそうですか。まあ基本はやっぱり DescriptorTable 使えてことですねえ。やっぱり初心者
ノイバノイじやねえか!!ふざけるな!!! (追真)

しゃあない。

いつものようにデスクリプタヒープとデスクリプタテーブルを作って、どうぞ。

といいうかよく考えたらテクスチャのフォーマットとか幅とか高さとかそういうのってビューとして伝える必要があるわけだし、ビューを使うってことは、デスクリプタヒープが必要ってことだし、そう考えるとやっぱりデスクリプタテーブルじゃないとダメみたいですね。
また君かあ。壊れるなあ…。

俺ここで 4 時間以上ハマったぞクソッタレ!!!! (パイプライン通すまで発覚しないため…)

Descriptor は甘え。はつきりわからんだね。

ペラポリシェーダを作る

新しくペラポリシェーダを作ります。ファイル名は pera.hlsl とでもしておきます。

```

//ペラポリシェーダ
//テクスチャとサンプラー
Texture2D<float4> tex:register(t0); //通常テクスチャ
SamplerState smp:register(s0)
struct Output{
    float4 svpos:SV_POSITION;
    float2 uv:TEXCOORD;
}
//頂点シェーダ
Output vs(float4 pos:POSITION, float2 uv:TEXCOORD)
{
    Output output;
    output.svpos=pos;
    output.uv=uv;
    return output;
}
//ピクセルシェーダ
float4 ps(Output input):SV_Target{
    return tex.Sample(smp,input.uv);
}

```

ペラポリ用レイアウトを作る

ご自分の定義に合わせてください。

```

D3D12_INPUT_ELEMENT_DESC peraLayoutDescs() = {
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0 ... (中略) ...
}

```

ペラポリ用パイプラインステートを作る

ここでルートシグネチャの仕様により何度も失敗…ふざけるな!!!

まあ、変更点はレイアウトとシェーダだけなので、特に記載する必要はないでしょう。

ペラポリ表示部分を作る

通常のレンダリングができているとして話を進めます。

本番のバッファに書き込むわけですから、レンダーターゲットは本番の奴に書き込んでください。

プリミティブポロジは TRIANGLESTRIP だと長方形の場合は都合がいいですねえ。

あとは頂点バッファをセット。今回は4頂点だけなので、インデックスは不要でしょう。

DrawInstanced(4,1,0,0)

でいいと思います。なお、頂点バッファビューを指定し忘れのないように。うまくいけば元通りモデルが踊っている姿が見れると思います。なお、↑の頂点数を3とかにすると「レンダリング結果を貼られたテクスチャ」が表示されているのが分かると思います。

ここまで間違いやすいポイント

ここまで実装するうえで…ぼくがやらかしちゃった部分を書いておきます。

- バッファ(Resource)の作成時

パラメータに注意しましょう。今回のようにレンダーターゲットとして使用する場合には

リソース DESC を

```
resDesc.Flags = rtvFlag ? D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET:  
D3D12_RESOURCE_FLAG_NONE;
```

としています。RTVとして使えるようにするために

また、元々

```
D3D12_RESOURCE_STATE_GENERIC_READ
```

としていると思いますが、この部分も

```
rtvFlag ? D3D12_RESOURCE_STATE_RENDER_TARGET : D3D12_RESOURCE_STATE_GENERIC_READ
```

としています。

これにより、RTVにも使用できるリソースとなります。

- シェーダリソースビューのフラグ

次にシェーダリソースビューを作るときに僕がやらかしてた部分ですが

```
descHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;  
descHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
```

- 描画時にビューポート、シザーリングの指定

ああ～、これは忘れるんじゃあ～!!!

ていうかこれ忘れやすいわ。

```
_cmdList->RSSetViewports(1,&_viewport);
_cmdList->RSSetScissorRects(1,&_scissorRect);
```

- コマンドアロケータのリセット忘れ

```
//ペラポリ描画
_cmdAlloc->Reset();
_cmdList->Reset(_cmdAlloc.Get(), _peraPipeline.Get());
これは忘れちゃいけん(戒め)
```

- レンダーターゲットテクスチャに対してのルートパラメータのセットし忘れ

```
_cmdList->SetDescriptorHeaps(1, _heapFor1stPathSRV.GetAddressOf());
_cmdList->SetGraphicsRootDescriptorTable(1, _heapFor1stPathSRV-
>GetGPUDescriptorHandleForHeapStart());
```

とりあえず、以上のことには気をつけましょう。おれもやらかしたんだからさ!!!

加工してみよう

CG 検定試験前ですしちょうどいいのでちょっと色々と加工してみましょう。

モノクロ化

```
float b = dot(float3(0.298912f, 0.586611f, 0.114478f),rgb);
```

で輝度を計算します。計算式自体は様々な式がありますが、細かい事を考えなければだいたいどれも似たようなもんです。それはいいんですが

何故内積を取っているか分かりますか？ねえ。単なる色情報であって法線ベクトルとかでもないのに、何故定数との内積を取っているんでしょう…自分で考えてください。

ちなみにモノクロからちょっといじると…



古いゲームボーイみたいな表示になります

ああ、これ、そういうれば後述する「ポスタリゼーション」的な加工も組み合わせてるので、ほんのちょっと難易度は高いです。

反転

クソ簡単ですね。自分で考えてほしいですが、こっちも面倒なんでもうコードで説明します。

```
float4 col = tex.Sample(smp, input.uv);
return float4(1 - col.rgb, col.a);
```

説明の必要ないですよね。自分でコード読んで考えてください。ここまできて分からぬい人は…はあ～あほくさ…



なんか敵キャラみたいでええな

ポスタリゼーション？

```
return float4(tex.Sample(smp, input.uv).rgb - fmod(tex.Sample(smp, input.uv).rgb, 0.25f),
1);
```

/そもそもポスタリゼーションの意味わかりますかあ～？CG検定で出、出ますよ？階調変更の事です。

基本的にはグラフが階段状になってると思っていいと思います。結果的に階調が落ちます。今はフルカラーで表示してますので、そのまま落とすと

こんな感じになります！

落とし具合は色の各要素で調整した方がいいかもです。またポスタリゼーションで画像検索すると

こういう画像が出てくる！

単純に色調を落としているというより、ディザ処理が入っているように見えます。

デザ実装はまた後に回しますが、まあ、色々と表現手法があると思ってください。

軽い単純ぼかし(平均化)

```
float w,h,level;  
  
tex.GetDimensions(0,w,h,level);  
float dx = 1.0f / w;  
float dy = 1.0f / h;  
float4 ret = float4(0, 0, 0, 0);  
  
ret += tex.Sample(smp, input.uv + float2(-2*dx, 2*dy)) / 9.0f;  
ret += tex.Sample(smp, input.uv + float2(2*dx, 2*dy)) / 9.0f;  
ret += tex.Sample(smp, input.uv + float2(0, 2*dy)) / 9.0f;  
  
ret += tex.Sample(smp, input.uv + float2(-2*dx, 0)) / 9.0f;  
ret += tex.Sample(smp, input.uv) / 9.0f;  
ret += tex.Sample(smp, input.uv + float2(2*dx, 0)) / 9.0f;  
  
ret += tex.Sample(smp, input.uv + float2(-2*dx, -2*dy)) / 9.0f;  
ret += tex.Sample(smp, input.uv + float2(2*dx, -2*dy)) / 9.0f;  
ret += tex.Sample(smp, input.uv + float2(0, -2*dy)) / 9.0f;
```

9近傍をゼーんぶ足して9で割ってます。簡単に言うと自分と周囲8ピクセルの平均をとってそのピクセルの画素値としています。

dx,dy の係数をどんどん上げていけばボケ具合は強くなっていますが、ある程度以上になると、不自然になります。

画像処理ソフトみたいなきれいなボケってどうやって作るんでしょうか？実はフォトショップなどはガウスぼかしというぼかしテクを使っているんですが、非常に面倒だし重いのでちょっと後に回します。今回みたいなのを「平均化フィルタ」と言います。

エンボス

フォトショップに「エンボス」って加工があるんですが、これは「浮彫」「レリーフ」みたいな感じで、表面に絵の模様通りに凹凸がついているようにする加工です。左上を浮き上がらせ、右下

をへこませます。そのため

2 1 0

1 1 -1

0 -1 -2

とします。



わかりますかね？

//エンボス

```
ret += tex.Sample(smp, input.uv+float2(-dx,-dy))*2;
ret += tex.Sample(smp, input.uv + float2(0, -dy))* 1;
ret += tex.Sample(smp, input.uv + float2(-dx, 0)) * 1;
ret += tex.Sample(smp, input.uv + float2(dx, 0)) * -1;
ret += tex.Sample(smp, input.uv + float2(0, dy)) * -1;
ret += tex.Sample(smp, input.uv + float2(dx, dy)) * -2;
```

シャープネス(エッジ強調)

0 -1 0

-1 5 -1

0 -1 0

全てを足して1になるようにして、中心に5をかけて強調し、周囲を-1乗算します。それだけで



こんな感じになります

簡単な画像処理的輪郭線抽出

```
float4 ret = tex.Sample(smp, input.uv);
ret = ret * 4 -
    tex.Sample(smp, input.uv + float2(-dx, 0)) -
    tex.Sample(smp, input.uv + float2(dx, 0)) -
    tex.Sample(smp, input.uv + float2(0, dy)) -
    tex.Sample(smp, input.uv + float2(0, -dy));
float b = dot(float3(0.298912f, 0.586611f, 0.114478f), 1 - ret.rgb);
b = pow(b, 4);
return float4(b,b,b,1);
```

これは完全に「ぼくがとっさにかんがえた手っ取り早い輪郭線抽出コード」なので、適切かどうかわからませんが、少なくとも輪郭線を表示できます。

半分から下を輪郭表示します /

これは画像処理の知識があれば、思い付きで作れます。なんかの本を見る必要もありません。要はお勉強しろって事さ。

一応、やってる事は、まず輪郭抽出のため

0 -1 0

-1 4 -1

0 -1 0

こういうフィルタをかけます。実はこのフィルタ。GIMPやフォトショップでもあるため、ちょっと試してみるといいと思います。これは隣の色との色差を強調するものです。どういう事がと言う

と、例えば隣のピクセルが同じ色だったとします。

そうした場合、計算結果は0となり、真っ黒になります。しかし色に差がある場合はどうなるでしょうか? 例えば(64,0,64) (0,64,0)(64,0,64)という並びだったとします。そうすると、(-255,255,-255)になり、その部分は黒くなりません。

saturateをつかっても使わなくてもいいんですが、ひとまず↑のフィルタをかけると

/こうなります。これはこれで近未来な感じでいいですね
/もしこれを線画っぽくしたいと思うのであれば、もう一工夫必要です。ひとまず白黒化して
反転しましょう。やり方は説明いたしません。

薄いですねえ

powしましたよ(何でかは言わない…考えろ)

/b=pow(b,b);

まあ…マジかな

ちなみにpow値上げれば上げるほど線が濃くなります。理由は…考えろ!

ガウシアンぼかし(簡易版)

ガウスぼかし、もしくはガウシアンぼかしといふぼかしがあります。一応さっきの平均化フィルタの応用で作れるんです…まあ作りましょうか。

<http://htsuda.net/archives/2064>

の記述に

1/256	4/256	6/256	4/256	1/256
4/256	16/256	24/256	16/256	4/256
6/256	24/256	36/256	24/256	6/256
4/256	16/256	24/256	16/256	4/256
1/256	4/256	6/256	4/256	1/256

5×5画素

こんなのがあるんでそのまま適用してみましょう。

ret=ret*36/256;

dx *=2;

```
dy*=2;  
//今のピクセルに8近傍のピクセル値を加算  
ret+=tex.Sample(smp,input.uv+float2(-2*dx,2*dy))*1/256;  
ret+=tex.Sample(smp,input.uv+float2(-1*dx,2*dy))*4/256;  
ret+=tex.Sample(smp,input.uv+float2(0*dx,2*dy))*6/256;  
ret+=tex.Sample(smp,input.uv+float2(1*dx,2*dy))*4/256;  
ret+=tex.Sample(smp,input.uv+float2(2*dx,2*dy))*1/256;  
  
ret+=tex.Sample(smp,input.uv+float2(-2*dx,1*dy))*4/256;  
ret+=tex.Sample(smp,input.uv+float2(-1*dx,1*dy))*16/256;  
ret+=tex.Sample(smp,input.uv+float2(0*dx,1*dy))*24/256;  
ret+=tex.Sample(smp,input.uv+float2(1*dx,1*dy))*16/256;  
ret+=tex.Sample(smp,input.uv+float2(2*dx,1*dy))*4/256;  
  
ret+=tex.Sample(smp,input.uv+float2(-2*dx,0*dy))*6/256;  
ret+=tex.Sample(smp,input.uv+float2(-1*dx,0*dy))*24/256;  
//既に計算済み  
ret+=tex.Sample(smp,input.uv+float2(1*dx,0*dy))*24/256;  
ret+=tex.Sample(smp,input.uv+float2(2*dx,0*dy))*6/256;  
  
ret+=tex.Sample(smp,input.uv+float2(-2*dx,-1*dy))*4/256;  
ret+=tex.Sample(smp,input.uv+float2(-1*dx,-1*dy))*16/256;  
ret+=tex.Sample(smp,input.uv+float2(0*dx,-1*dy))*24/256;  
ret+=tex.Sample(smp,input.uv+float2(1*dx,-1*dy))*16/256;  
ret+=tex.Sample(smp,input.uv+float2(2*dx,-1*dy))*4/256;  
  
ret+=tex.Sample(smp,input.uv+float2(-2*dx,-2*dy))*1/256;  
ret+=tex.Sample(smp,input.uv+float2(-1*dx,-2*dy))*4/256;  
ret+=tex.Sample(smp,input.uv+float2(0*dx,-2*dy))*6/256;  
ret+=tex.Sample(smp,input.uv+float2(1*dx,-2*dy))*4/256;  
ret+=tex.Sample(smp,input.uv+float2(2*dx,-2*dy))*1/256;
```



平均ぼかしと比べてどうですかね?

これもうわからんねえな。

ということでもうちょっときちんとやってみましょう。

ガウシアンぼかし(ちゃんとしたやつ)

そもそも何故「ガウスぼかし」などという名前がついているのでしょうか?これは確率分布関数の一つ、ガウス分布(正規分布)から来ている。

<https://ja.wikipedia.org/wiki/%E6%AD%A3%E8%A6%8F%E5%88%86%E5%B8%83>

ちょっと分かりづらいのでもう少し分かりそうなサイトを見る

<http://zelli.j.hatenablog.com/entry/20140809/p1>

ふんふん。なるほど。わからん。

いちおう手元の本によると

$$f(x) = c \exp \left(-\frac{x^2}{2\sigma^2} \right)$$

/と言う関数らしい。

こういう形になる関数です。いわゆる正規分布と言うやつですね。なんでコンピュータグラフィックスの話に確率/統計の話が?って思うかもしれません。ひとまずこの「形」が重要なんです。あと、コンピュータグラフィックスには意外と確率/統計が出てくるので、勉強しておきましょう。

ちなみに \exp ってのはエクスponエーシャルと言って、ネイピア数… $e=2.718\dots$ って数なんだけど

軽く説明しておくと

$$f(x) = e^x$$

という関数はちょっと特殊で、 $f'(x) = e^x$ なのだ!!つまり微分しても結果が変わらないという非常に不思議な関数なのだ。

で、 \exp ってのは $\exp(x) = e^x$ という事を表している。つまり先ほどの式は

$$f(x) = c e^{-\frac{x^2}{2\sigma^2}}$$

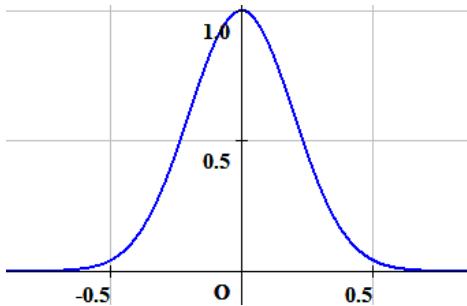
という意味になるのだが、まあ酷く分かりづらい式になる。どうせ \exp 関数を使用するのだからああいった描き方になっているだけなのだ。

ちなみに FunctionView というフリーソフトで $\sigma=5$ としてグラフを描画してみると

?

陽関数 $y=f(x)$

y_1 $e^{-(x^2)/2(5^2)}$



こういう感じの図になります

ものごとが難しそうに思えるかもしれません。が、単純にこういう「ガウス関数」が、あってそれが「ガウスばかし」の元になってるっていう事を知ってれば十分です。細かい数式はググればいいですしどうせ計算するのはコンピュータですし、標準関数に \exp ってあるので

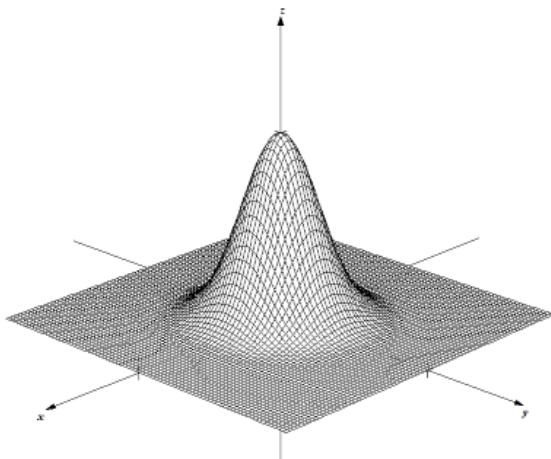
<http://www.c-tipsref.com/reference/math/exp.html>

それほど難しく考へることはあります。

また、実際には↑の図のように 2 次元ではなく 3 次元であるため

$$z = f(x, y) = c^2 \exp \left(-\frac{x^2}{2\sigma^2} \right) \exp \left(-\frac{y^2}{2\sigma^2} \right)$$

こういう式になって



こういうグラフになります

まあまあ、そうビビんなくていい。

別に分からなくても「使えりゃいい」のだ。さて、ここまでやってグラフはいいけどこれが何なのさ？

と思う方もおられるだろう…実はこれが「重み(ウェイト)」…「係数」となるのだ。ちょっと前のプログラムを思い出してほしい。

```
ret += tex.Sample(smp, input.uv + float2(-2 * dx, 2 * dy)) * 1 / 256;
ret += tex.Sample(smp, input.uv + float2(-1 * dx, 2 * dy)) * 4 / 256;
ret += tex.Sample(smp, input.uv + float2(0 * dx, 2 * dy)) * 6 / 256;
ret += tex.Sample(smp, input.uv + float2(1 * dx, 2 * dy)) * 4 / 256;
ret += tex.Sample(smp, input.uv + float2(2 * dx, 2 * dy)) * 1 / 256;
```

下線で示した部分が実は重み(ウェイト)にあたります。

でも、もう一つだけいいかな？まず平均化ぼかしをかけた時のテーブルは

$$\frac{1}{9} \quad \frac{1}{9} \quad \frac{1}{9}$$

$$\frac{1}{9} \quad \frac{1}{9} \quad \frac{1}{9}$$

$$\frac{1}{9} \quad \frac{1}{9} \quad \frac{1}{9}$$

としていました。また、3x3 のガウス的ぼかしテーブルは例えば

$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

となります。この二つのテーブルを見て、何か気が付くことは…ないかな？

気づけよオラアアアアアッ!!!! 小学生でも気づくぞ!!!



そう、全部足すと1になるんだよ。ていうかそうしないと明るさが変わっちゃうからね。じゃあどうするのかと言うと

```
std::vector<float> weights(weightNum);
float total=0.0f;
float x=0.0f;
for(auto& wgt:weights){
    wgt=expf(-x*x/(2*s*s));
    total+=wgt;
    x+=
}
//足して1にするようにする
for(auto& wgt:weights){
    wgt/=total;
}
```

とします。もしくは average 関数を使ってもいいですがまあそこはお任せします。ちなみにこの処理は CPU 側ですね。で、このウェイト値を GPU に投げます。

まあ、それはいいんだけどややこしいのはここから。正直今遊んでいる頂点シェーダにちょっとだけ頑張ってもらいます。一応手元にある本のばかし方を行います。

現在のピクセルを真ん中と考えます。そうすると

のように左右対称です。ここからはほぼその参考にした書籍(DirectX9シェーダプログラミングブック)通りにやってみましょう…左右のずらし方とウェイトは同じであるため、ひとまず計算は片方だけやればいいと考えます。ただ最終的にはそれも全部足されてしまうため

最後に $\text{total} * 2$ を行います。但し真ん中分は引くので、

$\text{total} * 2 - 1$

とします。なぜ最後に -1 なのかというと

$wgt = \exp(-0 * 0 / (2 * s * s));$

というのは $e^0 = 1$ であるから 1 を引いています。

```
float s=5.0f; //ここをいじるとボケ具合が変わる
```

```
float x=0.0f;
```

```
float total=0.0f;
```

```
for(auto& wgt:_gaussianW.w){
```

```
    wgt = exp(-(x*x)/(2*s*s));
```

```
    total += wgt;
```

```
    x+=1.0f;
```

```
}
```

```
total=total*2.0f-1; //2をかけて1を引いてるのは↑のループは左側だけなので右側もあるよ  
の意味で2倍しています。ただ真ん中は1回でいいので1を引いています。
```

```
//足して1になるようにする
```

```
for(auto& wgt:_gaussianW.w){
```

```
    wgt /= total;
```

```
}
```

さて、シェーダの方ですが、結構ややこしいですよ。ずらした UV の計算の半分を頂点シェーダに肩代わり してもらいます。

まず前にも使用した GetDimensions 関数で画像の幅と高さを取得します。

```
tex.GetDimensions(0,w,h,level);
```

でずらしたものを事前計算しておいて、それをピクセルシェーダに投げたいんですが、UV 値として投げたいと思います。ちなみに TEXCOORD(n)として、TEXCOORD0～TEXCOORDn-1 まで使えます。

[https://msdn.microsoft.com/ja-jp/library/ee418355\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee418355(v=vs.85).aspx)

ちなみに n は「サポートされてるリソースの数」とのことですが、いくつやねん。どうやって調べるねん。

まあ手元の本やと 8 個でやっとるから、ちょっとそれに合わせてみよか(ソースコードまで合わせてはいけない。合わせないぞ!!)

```
float2 uv:TEXCOORD0;
float2 uv1:TEXCOORD1;
float2 uv2:TEXCOORD2;
float2 uv3:TEXCOORD3;
float2 uv4:TEXCOORD4;
float2 uv5:TEXCOORD5;
float2 uv6:TEXCOORD6;
float2 uv7:TEXCOORD7;
```

増えますねえ!!

で、こいつらに 2 ピクセルずつずらした UV 値を入れていきます。

```
output.uv1=uv+float2(-1/w,0);
output.uv2=uv+float2(-3/w,0);
output.uv3=uv+float2(-5/w,0);
output.uv4=uv+float2(-7/w,0);
output.uv5=uv+float2(-9/w,0);
output.uvb=uv+float2(-11/w,0);
output.uv7=uv+float2(-13/w,0);
```

入れすぎイ!!

さらに先ほど「GetDimension」で得た幅高も再利用したいのでこれもピクセルシェーダに渡します。適当なセマンティクスを勝手に定義して…

```
float2 size:SIZE;
```

を追加。モチロン代入

```
output.size=float2(w,h);
```

あと、ウェイト受け取り部分ですが、

```
cbuffer Weight:register(b0){  
    float4 wghts(2);  
};
```

こんな感じで受け取っておきます。float wghts(8)にしていいのは float4 アライメントの所為か、中身がズレるからです(これも 3 時間くらいハマりました)

あとは、頂点シェーダの反対側の値を作つて足しこんでいきます。

```
float2 offsetx = float2(14 / w, 0);  
ret = ret * wghts(0).x;  
ret += wghts(0).y * (tex.Sample(smp, input.uv1) + tex.Sample(smp, input.uv7 + offsetx));  
ret += wghts(0).z * (tex.Sample(smp, input.uv2) + tex.Sample(smp, input.uv6 + offsetx));  
ret += wghts(0).w * (tex.Sample(smp, input.uv3) + tex.Sample(smp, input.uv5 + offsetx));  
ret += wghts(1).x * (tex.Sample(smp, input.uv4) + tex.Sample(smp, input.uv4 + offsetx));  
ret += wghts(1).y * (tex.Sample(smp, input.uv5) + tex.Sample(smp, input.uv3 + offsetx));  
ret += wghts(1).z * (tex.Sample(smp, input.uv6) + tex.Sample(smp, input.uv2 + offsetx));  
ret += wghts(1).w * (tex.Sample(smp, input.uv7) + tex.Sample(smp, input.uv1 + offsetx));
```

さて、そこまでやれば…!!



結構あがらさまにボケましたね

これで終わり…? まさか。

とぼけちゃつてえ…縦方向があるでしょッ…!!! というわけで、このレンダリング結果をさらに一時バッファにレンダリングしておいて、今度は縦方向に同じことをやります。

1パス増えるんだよなあ……ガウスは正直コスト高いんすわあ…だから大抵は縮小テクスチャを作つて(ただ単にテクスチャの作成の際に幅とか高さを半分とかにしてやります)いわけです)

縦方向にも無事入れてあげれば…



こんな感じでいい感じにボケてくれます

ちょっと僕がハマつた点としては、ペラ1パス目とペラ2パス目の共通命令をバンドル化して、パイプラインだけ変更しようとしてたんですが、これが上手くいきませんでした。何故かと言うと、バンドルを作るときにパイプライン指定を行ってしまうので、バンドル実行時にパイプラインの指定が内部に入っている事になります。

つまり共通部分をバンドル化してパイプラインだけ変更ってのはできませんでした。できるって人はネタ提供お願いします(4時間くらい色々試したり、色々と調べましたが解決には至りませんでした)

というわけで、ここはこんなもんで勘弁してください。

ガラスフィルタ(画面歪み)シェーダ

市販のゲームの中にはスクリーンにひびが入ったり弾痕が残ったりして表示画像がゆがんで表示される表現があります。実はこれは法線マップ画像を用いて画面をゆがませて、屈折に似た効果を演出しています。画面に就いた水滴の表現なんかもこれですね。

ところで説明に出てきた「法線マップ」とは何でしょうか。

これは本来であれば物体表面に凸凹をつけるために普通のテクスチャの代わりに「法線ベクトル」を張り付けるものです。

本来凸凹のない物体表面に凸凹を施す。これが法線マップである。それが歪みシェーダとどう関係しているのでしょうか?

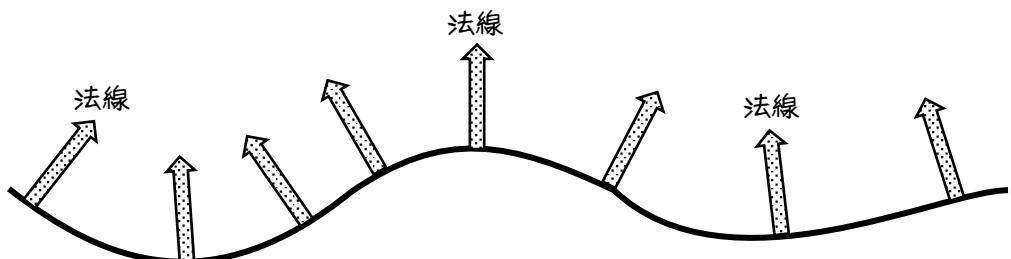
仕組みとしては「凸凹に見えるシール」を凸凹のない面に貼っているようなものです。そもそもどういった理屈で凸凹がついて見えるのかと言うと平たい面の法線が



であるのに対して、凸凹な面は

のようになっている。このため、法線があちこちに向いている状態だ。

ちなみにデュフューズやスペキュラーの明るさは、法線ベクトルに依存している事を思い出してほしい。



もちろんマッピングするには画像である必要があるため、法線も画像として保存されてしまう。

その保存されているデータは法線方向 N_x, N_y, N_z を RGB に変換して画像として保存できるようにしています。

単純な変換式を書くとこうです。

$$(R, G, B) = \left(\frac{1.0 + N_x}{2}, \frac{1.0 + N_y}{2}, \frac{1.0 + N_z}{2} \right)$$

法線情報は正規化されているため、それぞれの成分は-1~1に収まります。しかし色情報はマイナスがないため0~1の範囲に収める必要があります。そのためには1を足して2で割ればいいため、上記のような計算式で保存されています。

このため、RGB画像になってしまった法線情報をベクトルに逆変換するには2倍して1を引きます。

$$(N_x, N_y, N_z) = (2R - 1.0, 2G - 1.0, 2B - 1.0)$$

となりますので、最終的にはhslでこの計算をやってあげないといけないわけです。

なお、一般的な法線マップはZ方向を面の法線ベクトルとしているため例えばもともと(0,0,1)のベクトルをRGBに直した場合(0.5,0.5,1.0)となり、普通は大半が薄い青色になるかと思います。



歪みに使用する場合、法線マップ画像の色が(0.5,0.5,1.0)からどれくらいズレているかが、その部分の歪みの強さになります。

それではこの「法線マップ画像」を用いて画面をゆがませてみましょう。まずは画像の読み込みを行います。普通の画像としてのロードでいいです。

まずは変数と関数の準備

```
//歪みテクスチャ用  
ComPtr<ID3D12DescriptorHeap> _effectSRVHeap;  
ComPtr<ID3D12Resource> _effectTexBuffer;  
bool CreateEffectBufferAndView();
```

歪みテクスチャとペラのテクスチャを同一デスクリプタヒープに格納するかどうか迷うところですが、利用の性質が明確に違うため、ここは分けておきます。なお、歪みテクスチャならばdistortionとしたいところですが、エフェクト全般に使用する可能性があるためeffectという名前にしています。

```

bool
Dx12Wrapper::CreateEffectBufferAndView() {
    //ノーマルマップをロードする
    if (!LoadPictureFromFile(L"normal/crack_n.png", _distortionTexBuffer)) {
        return false;
    }
    //ポストエフェクト用のデスクリプタヒープ生成
    D3D12_DESCRIPTOR_HEAP_DESC heapDesc = {};
    heapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
    heapDesc.NumDescriptors = 1;
    heapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
    auto result = _dev->CreateDescriptorHeap(&heapDesc, IID_PPV_ARGS(&_distortionSRVHeap));

    //エラー対処は各自で

    //ポストエフェクト用テクスチャビューを作る
    auto desc=_distortionTexBuffer->GetDesc();
    D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
    srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
    srvDesc.Format = desc.Format;
    srvDesc.Texture2D.MipLevels = 1;
    srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
    _dev->CreateShaderResourceView(
        _distortionTexBuffer.Get(),
        &srvDesc,
        _distortionSRVHeap->GetCPUDescriptorHandleForHeapStart());
}

```

次にこのテクスチャをシェーダから見れるようにするためにルートシグネチャにスロット設定を追加します。

```

range[2].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;//t
range[2].BaseShaderRegister = 1;//1
range[2].NumDescriptors = 1;

```

今回はヒープを分離して作っていますので、ルートパラメータも別に用意します。

```
rp[2].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;//  
rp[2].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;  
rp[2].DescriptorTable.pDescriptorRanges = &range[2];  
rp[2].DescriptorTable.NumDescriptorRanges = 1;
```

合わせてパラメータ数を増やしておくのを忘れないようにしてください。
rsDesc.NumParameters = 3;

あとはピクセルシェーダ側に受け取り用の変数を用意し…

```
Texture2D<float4> distTex : register(t1);
```

この distTex を元にテクスチャ参照位置をずらしている。このズレ具合が歪みを表現する。この時表現の範囲を RGB から XYZ にする変換を忘れないように行います。

$$(N_x, N_y, N_z) = (2R - 1.0, 2G - 1.0, 2B - 1.0)$$

また、最大 1 ズレるわけですが 1 はテクスチャ座標の端を表し、大きすぎるため適当な少數を乗算して調整します。

```
float2 nmTex= distTex.Sample(smp, input.uv).xy;  
nmTex = nmTex * 2.0f - 1.0f;  
//nmTexの範囲は-1～1だが、幅1がテクスチャ1枚の  
//大きさであり-1～1ではゆがみすぎるため0.1を乗算している  
return tex.Sample(smp, input.uv+nmTex*0.1f);
```

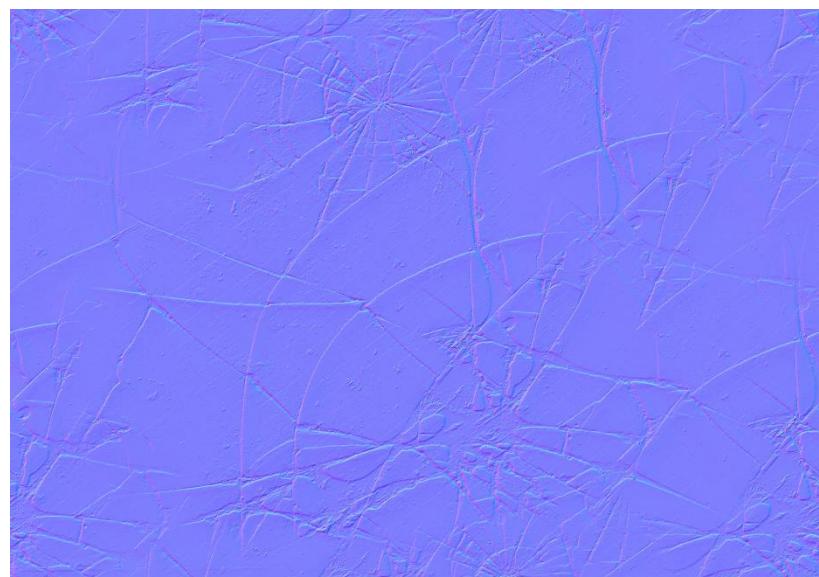
最後に Draw 時にデスクリプターテーブルとルートパラメータのインデックスをしてやれば終了です。

```
_cmdList->SetDescriptorHeaps(1, _distortionSRVHeap.GetAddressOf());  
_cmdList->SetGraphicsRootDescriptorTable(2,  
    _distortionSRVHeap->GetGPUDescriptorHandleForHeapStart());
```

ここまでが「**グ**なくかければノーマルマップの形に添って歪んでくれるでしょう。



今回の歪みは分かりやすいようにかなり大きさに歪ませましたが、実際にはさりげない感じのマップを使用します。



このくらいのぱっと見歪んでるか歪でないのか分からないマップでも



このように割れガラスの表現には十分となります
ひとまず「簡単な」ポストエフェクトは以上です。

シャドウマップ

フヒヒ…ついに来ちゃいました。

最初に予告してたアレですよ!!!影を落とすんですよ!!!

いや~長かったなあんもお~辞めたくないりますよ~プログラム。

シャドウ!シャドウ!オウ冷えてつか~?ノッヂエ冷えてますよ~。とはいえ影を落とすためには色々とおせん立てが必要なので、とりあえずまずは手っ取り早く『漬し影』で軽くお茶濁します…いや、数学の勉強のためと思ってちょっと聞いてくれよ。

影行列(漬し影)…ウノ影

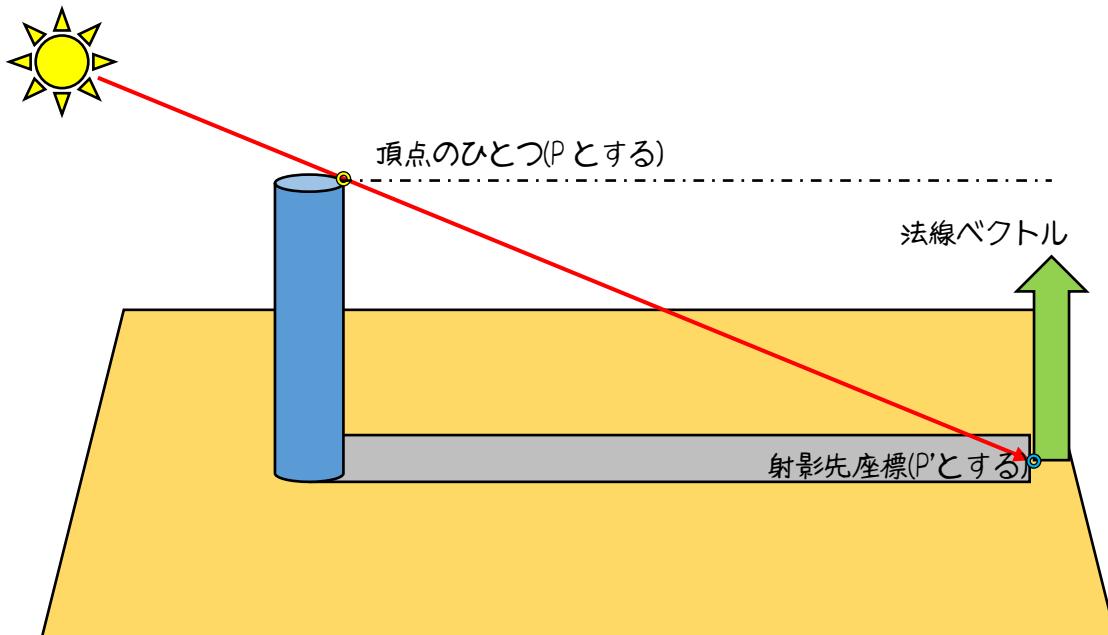
行列を用いて光の方向と地面の法線ベクトルから、頂点をどこに落とすべきかを計算する。で、それをやる行列を作る。そしてそれを描画する際に黒で描画すればいい。

二言で言うと

「つぶす。黒く塗る。」

である。

行列は最初から用意されていて、`XMMatrixShadow` というのがあるが、一応後学のために仕組みは言っておこう。仕組みはこうだ。



影というのは、光の方向と平面の法線ベクトルで決まると書いたが、図のような関係になっていると考える。

影が落ちる…これを元々の図形が地面に投影されると考えると、3D空間上の点 P が『地面(平

面)に投影され点 P' になる。と言える。

となると、要は↑の図における「直角三角形」の形がわかれればよい事になる。

直角三角形の形ってのは、ああ、↑の図だとちょっと誤解を招きますね。光線ベクトルと法線ベクトルと地面の位置(高さ…原点からのオフセット)が分かればいいって事。

どういう事がと言うと既に「古典的レイトレーシング」をやりこみ済みの君らにはお分かりの事と思いますが…ライトベクトルは3D空間上においては「傾き」を意味すると思って、それと平面との交点を求めればいい。

ここで

- 平面の法線ベクトル(正規化済)と元の図形(モデル)の頂点座標との内積
 - 平面の法線ベクトル(正規化済)と光線ベクトルとの内積
- を求めます。

最初に頂点座標と平面の法線ベクトルとの内積をとります。これは「平面と頂点との最短距離」つまり頂点から地面への「垂線の長さ」を測るためのものです。

なんでそんなものを測るのかと言うと地面までの距離によってどこまで影が伸びるかが決まるからです。

さあ、高さ(切片)はわかった(実はまだ分かってないんだが)。次は傾きだ。傾きはライトベクトルそのものなんだけど、ライトベクトル1回分でどれくらい地面に近づくかが分かればそれをライトベクトルに乗算して得られた座標が影が落ちた先の座標である。

そしてまたライトベクトル1回でどれくらい地面に近づくかも内積によって分かりますね?何でかって、ライトベクトルと法線ベクトルの内積はライト1回あたりどれくらい地面に近づくかの値になるからです。

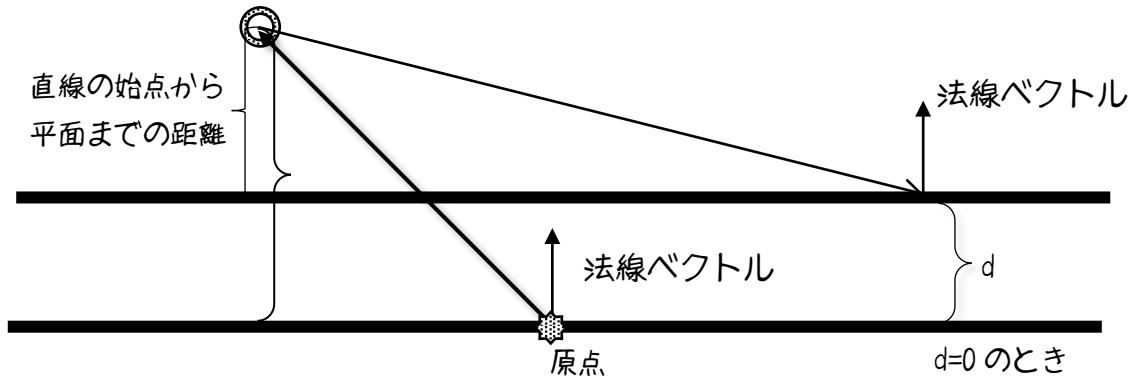
つまり

$$a = \frac{\vec{P} \cdot \vec{N}}{\vec{L} \cdot \vec{N}}$$

こんな感じです。

が罠が仕掛けられています。実はこのままでは平面といふのは原点を通る平面になってしまいます。これでは(0,0,0)を通らない平面は表現できません。

さて、どうすればいいと思ひます？



原点までの垂線の長さから地面はオフセットしていると考えます。オフセットが d だとすると、法線方向に浮き上がっているわけですから実際の距離は

$$\vec{P} \cdot \vec{N} - d$$

なわけだ。

ちなみに余談だけど、高校で習う「平面の方程式」は

$$ax + by + cz + d = 0$$

という方程式なんだ。 (a,b,c) は法線ベクトル(正規化済)らしい。けど、この式については「覚えやすい」からこうなってるんだろうけど正直納得いかない。

何故かと言うとこの式の場合、 d が法線方向にマイナスの向きにオフセットされるのだ。

例えば xz 平面に平行な平面を考えたとします。で y のプラス方向にちょっと(d だけ)浮き上がってると考えてください。

この場合、法線ベクトルは $(0,1,0)$ でするので平面の方程式に当てはめると

$$0x + y + 0z + d = 0$$

となるのですが、よく考えて式を整理してみてください。

$$y = -d$$

ん？ へへへへ？

浮き上がったはずが逆に沈んだるやんけ!!!! 覚えやすさを重視した結果…意味的(図形的)に混乱しやすくなつとるやん!!!

これだから受験覚えゲー数学は…と言いたくなる。ホンマ高校のセンセーは自分の頭で考え

て教えてほしい。現場の人間からしてみたらそんなカビの生えたような教育指導要領などクソつ飛らえなのだ。

まあ、ともかくそんなこんなで、地面までの距離が分かりましたと。あとはライトベクトルの内積で割って傾きが出ているわけだから求めたい座標は

$$P' = P + \vec{L} \frac{\vec{P} \cdot \vec{N} - d}{\vec{L} \cdot \vec{N}}$$

というわけだ。

さらにこれを行列にするのならばどうすればいいかな？

$$(x', y', z', 1) = (x, y, z, 1) \begin{pmatrix} S_{11} & S_{12} & S_{13} & S_{14} \\ S_{21} & S_{22} & S_{23} & S_{24} \\ S_{31} & S_{32} & S_{33} & S_{34} \\ S_{41} & S_{42} & S_{43} & S_{44} \end{pmatrix}$$

となる行列 S を考えましょう。

ちなみに傾き α を

$$\alpha = \frac{\vec{P} \cdot \vec{N} - d}{\vec{L} \cdot \vec{N}}$$

とすると求めたい点 P' は $P' = P + aL$ なので

$$P' = (x + a L_x, y + a L_y, z + a L_z)$$

である。それはそうなんだけど、実は $P \bullet N$ の P もまた元の (x, y, z) 座標であるから…これもう分かんねえな？一旦 $\alpha = \alpha$ の式を分解してみよう。

$$\alpha = \frac{\vec{P} \cdot \vec{N}}{\vec{L} \cdot \vec{N}} - \frac{d}{\vec{L} \cdot \vec{N}}$$

更に $\vec{L} \cdot \vec{N}$ が定数になることから

$$\alpha = \vec{P} \cdot \frac{\vec{N}}{\vec{L} \cdot \vec{N}} - \frac{d}{\vec{L} \cdot \vec{N}}$$

となる。もうここまで来たらついてきててもついてこなくとももう文句言わない。ワシの意地ぢや。

ちなみに全体を $\vec{L} \cdot \vec{N}$ で割る事になっている事を覚えておこう。

また、

$$\vec{P} \cdot \frac{\vec{N}}{\vec{L} \cdot \vec{N}}$$

P との内積をとる部分があるが、 P は「行列を乗算する座標」である。それとの内積だが、ちょっとここでヒントと言うか豆知識だが、ベクトルと行列の乗算の際、ベクトルとの内積をとりたければ「縦にXYZ成分を並べればいい」という事を覚えておきましょう。

どういう事がと言うと

$$(x, y, z) \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

の、乗算結果を考えると

$$(ax + dy + gz, bx + ey + hz, cx + fy + iz)$$

となっていますが、結果をよく見ると、 x 成分が (xyz) と (adg) の内積である事が分かりますね？これは様々な所で利用しますので、このテクは覚えておきましょう。自分で行列を作る場合はこのように「元」と「欲しい結果」を先に考えてそれを満たすような行列を考えていきまます。

さて、先ほどの部分…。 $\vec{L} \cdot \vec{N}$ だが、もはや定数だし、この後面倒になるので $G = \vec{L} \cdot \vec{N}$ とおきます。そうすると

$$P' = P + \vec{L} \frac{\vec{P} \cdot \vec{N} - d}{\vec{L} \cdot \vec{N}} = P + \vec{L} \left(\frac{\vec{P} \cdot \vec{N}}{G} - \frac{d}{G} \right)$$

だから

$$(x', y', z', 1') = (x, y, z, 1) \begin{pmatrix} 1 + \frac{N_x}{G} L_x & \frac{N_x}{G} L_y & \frac{N_x}{G} L_z & 0 \\ \frac{N_y}{G} L_x & 1 + \frac{N_y}{G} L_y & \frac{N_y}{G} L_z & 0 \\ \frac{N_z}{G} L_x & \frac{N_z}{G} L_y & 1 + \frac{N_z}{G} L_z & 0 \\ -\frac{d}{G} L_x & -\frac{d}{G} L_y & -\frac{d}{G} L_z & 1 \end{pmatrix}$$

です。

計算すると

$$x' = x + \frac{x N_x L_x}{G} + \frac{y N_y L_x}{G} + \frac{z N_z L_x}{G} - \frac{d L_x}{G}$$

$$y' = y + \frac{x N_x L_y}{G} + \frac{y N_y L_y}{G} + \frac{z N_z L_y}{G} - \frac{d L_y}{G}$$

$$z' = z + \frac{x N_x L_z}{G} + \frac{y N_y L_z}{G} + \frac{z N_z L_z}{G} - \frac{d L_z}{G}$$

となります。なお $xN_x + yN_y + zN_z = \vec{P} \cdot \vec{N}$ であるから

$$x' = x + \frac{L_x}{G}(\vec{P} \cdot \vec{N}) - \frac{d L_x}{G} = x + \frac{L_x}{G}(\vec{P} \cdot \vec{N} - d)$$

当然のようにこれは x, y, z 同じことになるので変換後の座標は

$$\left(x + \frac{L_x}{G}(\vec{P} \cdot \vec{N} - d), y + \frac{L_y}{G}(\vec{P} \cdot \vec{N} - d), z + \frac{L_z}{G}(\vec{P} \cdot \vec{N} - d) \right)$$

$$P' = P + \frac{L}{G}(\vec{P} \cdot \vec{N} - d) = P + \vec{L} \frac{(\vec{P} \cdot \vec{N} - d)}{\vec{L} \cdot \vec{N}}$$

ということで、求めたい値になることが分かります。あー、クソめんどくさ('A')

ということで、面倒な思いをさせたところで申し訳ないのですが、これをやる関数がありまます!!

ただ、一応平面の方程式と法線の話とかライトの話とかしとかないといパラメータ分からないのと、まあ、その、なんだ…行列の練習ですよ。どっちみち何処かで「欲しい」行列が用意されてない事がある。その時に投げだしてちゃあつまらんわけよ。

ちなみに D3DX 時代は中身まで公開してたので

<https://msdn.microsoft.com/ja-jp/library/cc372900.aspx>

見てみましょう。

ちなみに $d=dot(P,L)$ とありますが、 P は plane なので、 P というのは平面の法線ベクトルだというのが分かります。

つまり、1 行 1 列目は

$$L_x N_x + \vec{L} \cdot \vec{N}$$

という事になりますが、

$$G = \vec{L} \cdot \vec{N}$$

としていたことを思い出したうえで行列を表記すると

$$\begin{pmatrix} L_x N_x + G & L_y N_x & L_z N_x & L_w N_x \\ L_x N_y & L_y N_y + G & L_z N_y & L_w N_y \\ L_x N_z & L_y N_z & L_z N_z + G & L_w N_z \\ L_x N_w & L_y N_w & L_z N_w & L_w N_w + G \end{pmatrix}$$

こんな感じらしいんですけどね…？

「ライトの w 成分が 0 の場合、原点からのディレクショナル ライトを表す」などと書いてあるため、これらを 0 にしてみましょう。

$$\begin{pmatrix} L_x N_x + G & L_y N_x & L_z N_x & 0 \\ L_x N_y & L_y N_y + G & L_z N_y & 0 \\ L_x N_z & L_y N_z & L_z N_z + G & 0 \\ L_x N_w & L_y N_w & L_z N_w & G \end{pmatrix}$$

さて、全て G で割ってみましょう

$$G \begin{pmatrix} \frac{L_x N_x}{G} + 1 & \frac{L_y N_x}{G} & \frac{L_z N_x}{G} & 0 \\ \frac{L_x N_y}{G} & \frac{L_y N_y}{G} + 1 & \frac{L_z N_y}{G} & 0 \\ \frac{L_x N_z}{G} & \frac{L_y N_z}{G} & \frac{L_z N_z}{G} + 1 & 0 \\ \frac{L_x N_w}{G} & \frac{L_y N_w}{G} & \frac{L_z N_w}{G} & 1 \end{pmatrix}$$

さて、自分が作ったのと比べてみましょう。

$$(x', y', z', 1') = (x, y, z, 1) \begin{pmatrix} 1 + \frac{N_x}{G} L_x & \frac{N_x}{G} L_y & \frac{N_x}{G} L_z & 0 \\ \frac{N_y}{G} L_x & 1 + \frac{N_y}{G} L_y & \frac{N_y}{G} L_z & 0 \\ \frac{N_z}{G} L_x & \frac{N_z}{G} L_y & 1 + \frac{N_z}{G} L_z & 0 \\ -\frac{d}{G} L_x & -\frac{d}{G} L_y & -\frac{d}{G} L_z & 1 \end{pmatrix}$$

なんかスゲー惜しい感あるなあ。G が邪魔やで、G が。

ちなみに同次座標についてちょっと考えてみましょう。同次座標と言うのは

<http://www.elis.hokkai-s-u.ac.jp/~kikuchi/ma2/chap10a.html>

<http://www.osakac.ac.jp/labs/niizeki/monkey/section-3.html>

http://wwwb.pikara.ne.jp/ogawa-giken/image_process/image_062.html

例えば float4 ベクタで (x, y, z, w) という座標が示されていれば実際の座標は

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w} \right)$$

となるのだ。どういう事かと言うともし同時座標 w が 1 でなく w であれば

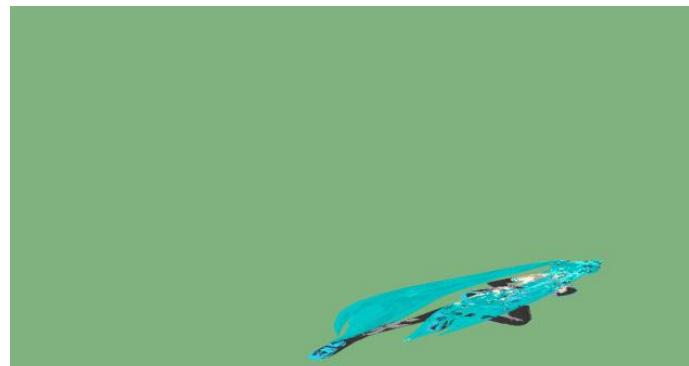
$$(x, y, w) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & 0 & 1 \end{pmatrix} = (x + w t_x, y, w)$$

となり、 t_x 移動時は同次座標 w のとき wt_x となり、このことは逆に考えると、同次座標系において wt_x 動いても実際には t_x しか動いていない事となる。つまり 4 行 4 列目が 1 以外 ($=w$) の場合は実際の座標的には全て w で割られているのと同じことになる…まあなんか正直に言うとなんか納得いかないんだが…まあいいや。

ともかくライトと法線の情報が必要という事は分かったと思う。試しに

```
auto p=XMFLOAT4(0,1,0,0); // 平面の方程式  
auto l=XMFLOAT4(-1,1,-1,0); // ライト座標  
matrix*=XMMatrixShadow(XMLoadFloat4(&p), XMLoadFloat4(&l));
```

とやってみると



ご覧のようにつぶれてしまいます。ですが、これは影に利用できそうだという事がわかります。なお、こいつの場合はマテリアルの使い分けが必要ありませんので



こうなります

ちなみに影は単なる座標変換なので、インスタンシングを用いて影を描画してみましょう。モデルの個別の描画は今 `PMDActor::Draw` 関数内でやっていますので、しばらくぶりにこの中の `DrawIndexedInstanced` 関数を変更します。

```
cmdList->DrawIndexedInstanced(material.indicesNum, 2, indexOffset, 0, 0);
```

あとは1枚目への描画を行っている頂点シェーダに最後の引数として

```
uint instNo: SV_InstanceID
```

を追加します。描画時にインスタンス数を2にしていれば頂点には0番と1番が入ってくるはずですので、

```
output.pos = mul(world, mul(conBone, pos));  
if (instNo == 1) {  
    output.pos = mul(shadow, output.pos);  
}
```

のようにIDが1番の時には影行列を乗算するようにします。そうすると



このように平面への射影描画と通常の描画の2つが描画されます

あとは地面を黒くするだけです。ピクセルシェーダにインスタンスIDを渡したいので、

```
struct Output {  
    float4 svpos : SV_POSITION;  
    float4 pos : POSITION;  
    float4 normal : NORMAL;  
    float2 uv : TEXCOORD;  
    uint instNo:SV_InstanceID;  
};
```

のように頂点シェーダの出力にインスタンス番号を追加します。

```
output.instNo = instNo;
```

そしてピクセルシェーダに渡せるようにしておき、あとはピクセルシェーダ側でこの番号を参照して1番ならば黒く塗ります。

```
if (input.instNo==1) {  
    return float4(0, 0, 0, 1);  
}
```

以上です。難しい部分はほとんどないため影が黒く描画されるようになっているかと思います。この影はポリゴンを潰している影に過ぎませんので先に述べたような問題点はあります
が、コストもあまりかかりずにきれいな影を出力できるため、状況が許せば（キャラがいる地
面が一つの平面で構成されている等）使える影だと思います。



潰したポリゴンを黒くして影っぽくなりました。

例年であればここから



このように影が落ちるところまでやりたいところですが、そこに時間かけるべきじゃないかなーって思います。プログラムも無駄に複雑になってしまいますしね。

ですので(MMDではこの方法で「セルフシャドウ以外の」影を落としています)、今回はもう次のシャドウマップに入っていこうと思います。

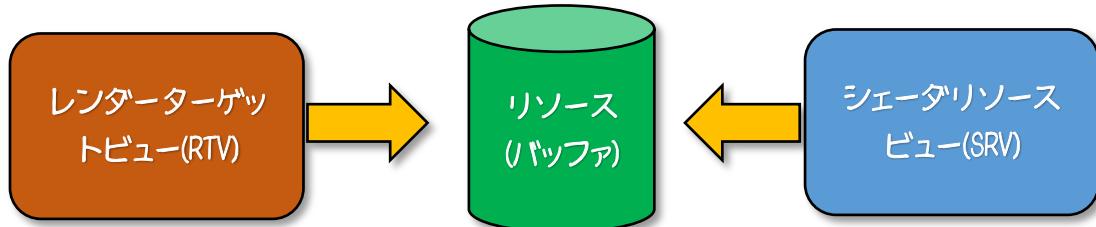
シャドウマップの導入(マルチパスの応用編)

シャドウマップに必要な概念(手順)を最初に書いておきます

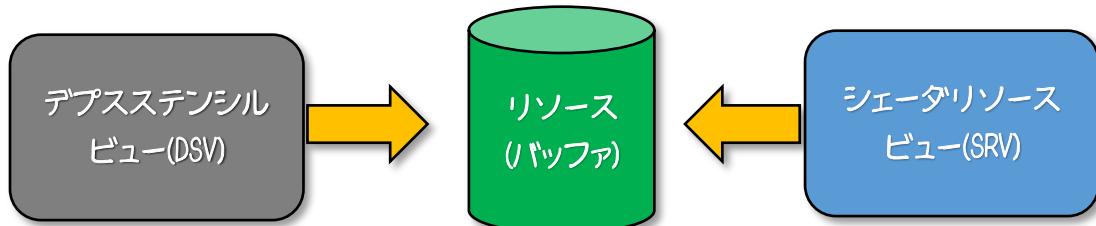
- ライトビュー(ライトの場所からはいチーズ!!)
- テクスチャへの深度値の書き込み(深度値職人大忙し)
- 深度値の比較(深度値見ながら影にするかどうか決める)

大雑把に言うとこんなもんです。既にマルチパスにしているのでそれほど抵抗はないかと思いますが…。

前はリソースに対して



RTV と SRV の2つの見方でしたが



今回は DSV と SRV の二つの見方となります。もう忘れちゃってるかもしれません、デプスマッテンシルビューと言うのは深度バッファを書き込むための見方(ビュー)です。

どちらにせよ RTV を使った時と同様にデプス(深度値)を書き込めるようなリソース(リバッファ)を作成し、2つのビューを作成することが必要になります。まずはそれを作りましょう。とはいっても、既にデプスバッファは用意しているため、特別に新しく作る必要はありません。

ちなみに深度値を書き込むとだいたい白っぽくなります。何故かと言うとまず 1.0 でクリアして、カメラから遠ければ遠いほど明るくなるためです。

しかし真っ黒…これは…と思ってバッファのパラメータとか、シェーダとか、ルートシグネチャとか、パイプラインステートとか、ビューとか調べてみましたかダメでした。

まあここで悩む過程でバリアとかもっと深く調べたりして、知識は補強できたのはプラスなんですが、結局解決しなかったので作り直します(もう深度バッファを新しく作る所からやります。それでもダメなら土日で Wrapper を完全に作り直します。)

そんなわけで、すんません今日はちょっと勘弁してください。

とりあえず皆さんには既にある深度バッファに対して、シェーダリソースビューとして見れるビューを作ってください。この時注意すべきなのは リソースの作り方のフォーマットの部分で フォーマットを D32 から R32_TYPELESS にしといてください。 R32_TYPELESS ってのはバッファとしてのビット数は決まっているけど、フォーマットはビュー側で決めてねって事です。

D32 にしちゃうとシェーダリソースビューで R32_FLOAT として使おうとすると失敗するので、気を付けるべき点はそこくらいかなと思います。ちょっとすまんですが、自分でやってみてください。

次に気付けるべき点としては、ペラポリレンタリング時に、元の深度バッファのクリアを行ってはダメで、もちろんレンダーターゲット指定の際にも深度バッファの指定は nullptr にしておいて、ペラポリ用のパイプラインステートでは DepthEnable を false にしておいてください。

ここに参考コードを書きたいのですが、バグってるので、下手に見せない方がいいかと…バグが感染しますからね。

と思ってたけど、まあ一応ここまで説明をそのままコードにしたら動くみたいなので、僕のバグみたいですね。

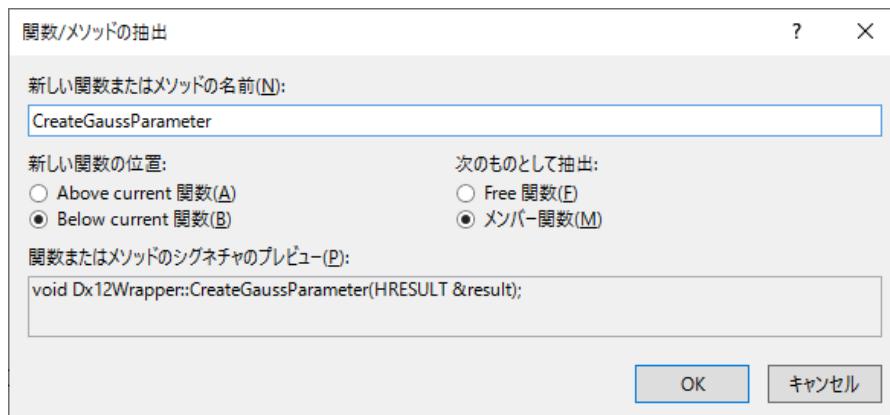
やってもやってもバグが取れないのに、リファクタリング
やってもやってもバグが取れないのに、リファクタリングしてたのですが、もう潜在バグが見つかる見つかる。ついでにデバッグル레이ヤーが吐くエラーにも対応しました。でも治りませ

ん。

で、リファクタリングしてると気が付いたのですが、Visual Studio にリファクタリングの機能があるのですよ。いや C# とかにあるのは知ってたんですけど、C++になかったのよね。しつこく追加されてて、これがあるとかなりリファクタリングが楽になるよ。

まず、まとめたいコードをガーッと選択して、その状態で右クリック。

『クイックアクションとリファクタリング』って出てくるので Extract 関数を選択。



するとこんなのが出てきますので、メソッド名(関数名)を書いて OK を押してください。
するともう自動でね、ヘッダにも cpp にも関数宣言と定義ができるのよね!!!!

これ、クソ楽!!!! 楽だからリファクタリングが楽しい!!!! たのしー!!!

他にも変数名や関数名の変更が一気にできちゃう(置換みたいな副作用なし)ので、ホンマリ ファクタリングが捲りますわーほんま。

土日を犠牲にしてリファクタリングした結果

なんの成果も…得られませんでしたアーッ!!! マジかよ。

とりあえず徹底的にリファクタリングしてみた結果、色々と潜在バグが見つかったので、とりあえずみんなも気を付けるようにね的ポイントを書いておく

- ConstantBuffer 指定の部分はルートパラメータを Descriptor にしなければならないのに Constant32Bit としてた(よく動いてたな)
- テクスチャ/バッファ作成の際に、書き込みしないテクスチャ(ロード情報を書き込まない)にて事に、書き込み用の設定してた
- コンスタントバッファは 256 アライメントじゃないといけないのに 32 バイト指定とか

やってた(ホンマなんで動くん?)

- 構造体のダミーメンバが残ってた(64バイト超えちゃう実験のやつ)
- よく考えたら 1stPath じゃなくて 1stPass だったね。疲れてたんだね!!
明らかに潜在バグを潰した上に、なんかエラーログも出てたので、対処。

色々と修正した(バリアの部分とか)おかげでエラーログも出なくなつたけど治らないよ!!

割と原型留めないレベルでリファクタしたけど治らなかつたのでラッピング作り直したけど治らなかつたよ(ー;ω;ー)ウッ…

なんで…!?

ちなみにマイクロソフトのサンプルの最新版は既に WDK のバージョンが上がってて、現状の環境では動かなかつたよ!!! コンパイルすら通らないよ!!! バージョンアップ多すぎませんかねえ!!

という事で、ここで立ち往生してもしかたないので、授業はそのまま進めていきます!! 治つたらすぐ追いつくからな!!!

さて、とりあえず俺の事はほっておいて先に進むんだツツツツツツツツ!!!

ズバリと解決

悩み始めて1週間にして、やっと解決しました。

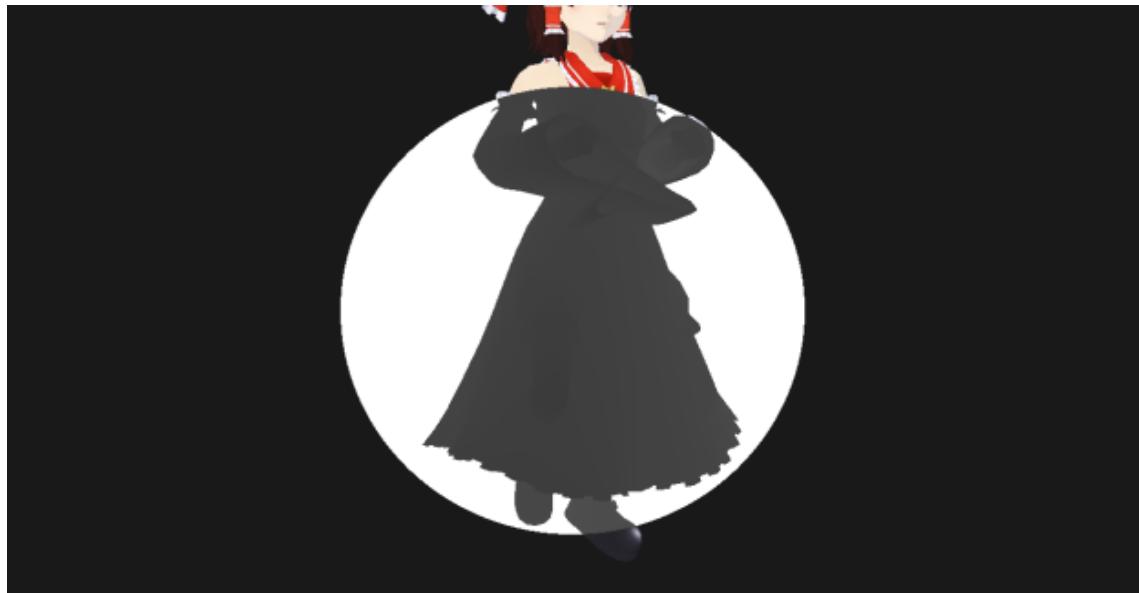
正直焦ってました…。もう CEDEC+KYUSHU も近いのに何こんなところで詰まっちゃってんのかと… 予想はしましたが、本当に バカみたいにバグ でした。でも、僕の脳内の「間違えやすい」データベースに強く刻み込まれたので、同じことを繰り返しはしないでしよう。

さて、原因ですが… ペラボリの SamplerDesc が、何へ故へかあ、
`samplerDesc.Filter=D3D12_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR;`
このようになっており、ただリニア補間すればいいのに COMPARISON が入ってるせいで、「比較」が行われていたようです。

で、まずい事にその下で

```
samplerDesc.ComparisonFunc=D3D12_COMPARISON_FUNC_NEVER;
```

としていたため、何も塗りつぶさずに終わってました。COMPARISON を外せば…



やったぜ…。1週間悩んで糞まみれのノグを見つけると、ああ～気が狂う程気持ちええんじや。

ノグファとかビューの部分ばかり見てサンプラーを見る時間が少なかったのがこれだけ時間がかかった原因ですね。一度は I 君がサンプラー怪しいんじゃないですかと指摘していたのに、「まさかフィルター部分ではあるまい」と COMPARISON_FUNC とかばかり見てたのも良くなかったです。惜しい所には来てたんですが…。

まあ、ともかく、めでたく修正完了したので晴れ晴れとして気持ちでシャドウマップやっていきましょう。

シャドウマップのしくみ

シャドウマップで一番面倒なのはマルチパスの部分なので、そこはもう済んでいる。実は山場がもう終わっているのだ。

そもそも影が落ちるといふのはどうしたことなのかといふのを今一度考えてみましょう。
「影ができる」ということは「光源からの光線が遮られている」事に他ならないわけですね。

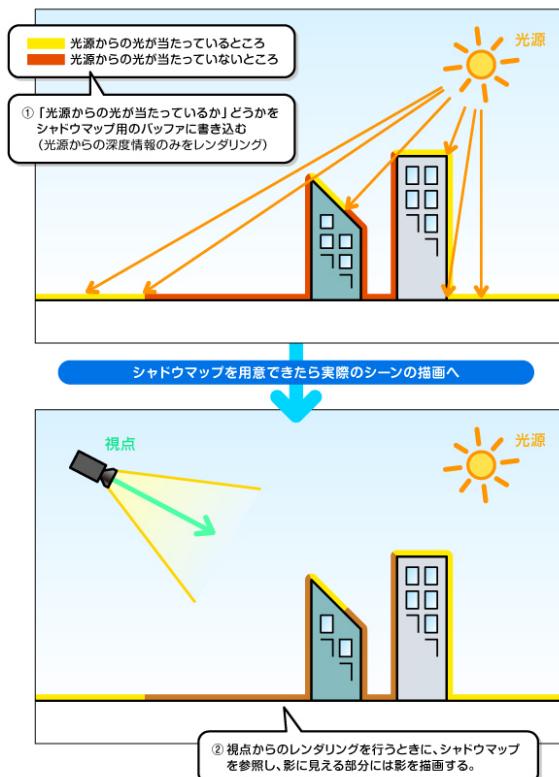
考え方としては、別のカメラ(現在のカメラが1カメなら、2カメだと思ってほしい)で光源から撮影して、そのカメラ(2カメ)に映った部分は直接光が当たっているわけだから、通常通りに描画する。

そしてカメラ(2カメ)に写っていなかった部分は直接光が当たっていない。つまり暗くなるというわけだ。で、1カメを現像する段階で2カメの深度情報を確認。

1カメのピクセル描画の際にそのピクセルの3D空間座標を2カメ座標変換した時の Z 値(深度値)を計算し、現像済みの2カメ深度より深ければ(大きければ)影だから暗く描画する。

これだけである。

ワカリヤスイ図が



ですね。

ちなみに図では模式的には点光源なんですが、今回は光源が平行光線なので、平行光線を扱っていきます。

準備

さて、既に僕以外の人は深度をテクスチャにできるでしょうからさっそく影を落としていきたいのですが、残念ながら影を落とす先がまだありません(自分自身に影が落ちますがちょっとわかりづらいので)

概要

というわけで、床と、余裕があればなにかプリミティブなオブジェでも置きたいけなと思います。

PrimitiveMesh という基底クラスを作つて、Plane(平面)、Cube(立方体)、Cone(円錐)あたりを作つてください。

Plane はコンストラクタに幅と奥行を与えると法線(0,1,0)の平面の頂点ノーフラを生成します(もしくは法線ベクトルを与えて自由な方向の平面を作る仕様でもいい)。

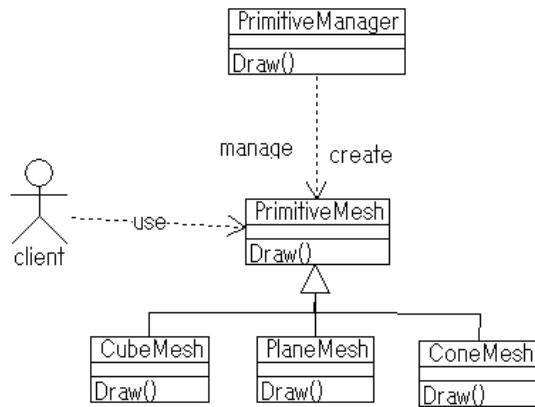
Cone はコンストラクタに、半径と高さと分割数を入れると円錐の頂点ノーフラを生成します。Cube は縦横高さがあればいいんじゃないかな。この辺は IQ もどき作った時を思い出してください。

まあ、考えてみてください。頭の体操です。Plane はすぐだと思いつますが…。

共通仕様としてはコンストラクタで設定。Draw で描画というシンプルなものにしたいかなと思います。また、今回のプリミティブは座標、法線(余力があるなら UV も)情報があればいいです。

PMD とは頂点の仕様が違うので、パイプラインステートは別々ですね。

設計



こんな感じで行こうかなと思います。各自こういう図は書きなれておきましょう。PlaneMesh やら CubeMesh やら作るとどうせ汎化したくなるので PrimitiveMesh という関数を作つてします。

実装

コンストラクタに必要なものを入れたら内部的に頂点バッファを作りましょう。

ひとまずコレの実装をお願いします。

```
using Microsoft::WRL::ComPtr;
///平面メッシュクラス
class PlaneMesh : public PrimitiveMesh
{
private:
    ComPtr<ID3D12Resource> _vbuffer;
    D3D12_VERTEX_BUFFER_VIEW _vbv;
public:
    //幅と奥行きから平面メッシュを作る
    PlaneMesh(ID3D12Device* dev, const DirectX::XMFLOAT3& pos, float width, float depth);
    ~PlaneMesh();
    void Draw(ID3D12GraphicsCommandList* cmdlist);
};
```

とりあえず頂点情報のみで床を表示できるようなシェーダとレイアウトを用意しましょう。

入れる情報は座標情報と法線情報と…uvだけでいいですね。頂点カラーのつけたい人は各自設定してください。

あとあと無駄なところは省くとして、ひとまずレイアウトと頂点バッファを内部で作りましょう。簡単な床から始めましょう。

で、頂点1つあたりの情報を格納するための構造体を作るんですが、コンストラクタで初期値を設定できるようにしといてあげると楽ですよ。

こんな感じで

///プリミティブ頂点型

```
struct PrimVertex {
```

```
    XMFFLOAT3 pos;
```

```
    XMFFLOAT3 normal;
```

```
    XMFFLOAT2 uv;
```

```
    PrimVertex(){
```

```
        pos=XMFFLOAT3(0,0,0);
```

```
        normal=XMFFLOAT3(0,0,0);
```

```
        uv=XMFFLOAT2(0,0);
```

```
}
```

```
    PrimVertex(XMFFLOAT3&p,XMFFLOAT3&norm,XMFFLOAT2&coord){
```

//入力変数名は、自分のメンバと重ならないようにするためだけにこんな名前にしている。

```
        pos=p;
```

```
        normal=norm;
```

```
        uv=coord;
```

```
}
```

```
    PrimVertex(float x,float y,float z,float nx,float ny,float nz,float u,float v){
```

```
        pos.x=x;
```

```
        pos.y=y;
```

```
        pos.z=z;
```

```
        normal.x=nx;
```

```
        normal.y=ny;
```

```
        normal.z=nz;
```

```
        uv.x=u;
```

```
        uv.y=v;
```

```
    }  
};
```

あとはそれぞれの(Plane や Cylinder)クラスのコンストラクタで頂点情報を作っていってやる。

まあ Plane は4頂点しか指定する必要が無いので楽だわな。

先程も言ったけど、この手の構造体の初期化を作つておくと便利な理由は
例えばプリミティブ頂点を宣言する時に

```
PrimVertex p(-10,-0.2,10.f,0,1,0,0,0);
```

などと書けます。

```
array<PrimVertex, 4> vertices{  
    { XMFLOAT3(pos.x - width / 2, pos.y, pos.z - depth / 2), XMFLOAT3(0,1,0), XMFLOAT2(0,0) },  
    { XMFLOAT3(pos.x - width / 2, pos.y, pos.z + depth / 2), XMFLOAT3(0,1,0), XMFLOAT2(0,1) },  
    { XMFLOAT3(pos.x + width / 2, pos.y, pos.z - depth / 2), XMFLOAT3(0,1,0), XMFLOAT2(1,0) },  
    { XMFLOAT3(pos.x + width / 2, pos.y, pos.z + depth / 2), XMFLOAT3(0,1,0), XMFLOAT2(1,1) }  
};
```

とでもすればいい。これで頂点をつくって頂点バッファを返せばいい。あとはレイアウトと、シエーダをセットすれば床も表示されるようになります。

あとは円錐はこの応用で作ってくれ…とはいいうものの、円柱の頂点を自動で作る方法とか
知らないかもしないので、ヒントを言おうか。

- 円錐と言っても結局は正多角形柱
- ループを使う
- ループ回数は引数で渡した分割数
- 頂点位置には sin,cos を利用
- ループの要素が++されるたびに角度は $2\pi / \text{分割数}$ 進む
- 円柱側面は長方形
- 長方形は三角形2つぶん
- プリミティブポロジが TRIANGLESTRIP であれば N 字を並べればよい
- プリミティブポロジが TRIANGLELIST ならば長方形一つに6頂点…頂点の並びに注意
- 法線は中心から外側に向かうように伸びている…つまり円柱の中心から側面頂点まで
- UV は…まあよきにはからえ(考えるのが面倒なら貼らなくても良い)

```
std::vector<PrimVertex> vertices(div*2+2);  
for(int i=0;i<=div;++i){  
    vertices[i*2].pos.x=r*cos((XM_2PI / float(div))*(float)i);
```

```

vertices(i*2).pos.z=r*sin((XM_2PI / float(div))*(float)i);
vertices(i*2).pos.y=0;

XMFLOAT3 norm=vertices(i*2).pos;
XMStoreFloat3(&vertices(i*2).normal,XMVector3Normalize(XMLoadFloat3(&norm)));

vertices(i*2).uv.x=(1.f / float(div))*float(i);
vertices(i*2).uv.y=1.0f;

vertices(i*2+1).pos.x= 0;
vertices(i*2+1).pos.z=0;
vertices(i*2+1).pos.y=height;

vertices(i*2+1).normal=vertices(i*2).normal;

vertices(i*2+1).uv.x=(1.f / float(div))*float(i);
vertices(i*2+1).uv.y=0.0f;
}

こんな感じかな?

```

とりあえず画面上に床と円錐を表示させてください。と言いたいところですが、そう簡単にいいかないかな…。

とりあえず思考をまとめておくと、必要なものは…

- 頂点/ドッファ
 - 頂点/ドッファビュー
- とまあこれくらいで

```

///プリミティブメッシュ管理
class PrimitiveManager
{
private:
    std::vector<std::shared_ptr<PrimitiveMesh>> _primitives;
    ComPtr<ID3D12PipelineState> _pipeline;
    ComPtr<ID3D12RootSignature> _rs;
    ComPtr<ID3D12Device> _dev;

```

```

public:
    PrimitiveManager(ComPtr<ID3D12Device> dev);
    ~PrimitiveManager();

    ///平面を作成
    ///@param pos 中心座標
    ///@param width 幅
    ///@param depth 奥行き
    std::shared_ptr<PlaneMesh> CreatePlane(const DirectX::XMFLOAT3& pos, float
width, float depth);

    ///描画準備
    void BeginDraw(ID3D12GraphicsCommandList3* cmdlist);

    ///描画
    void Draw(ID3D12GraphicsCommandList3* cmdlist);
};

てな感じで作ります。そろは言っても一気に作るのは大変なので平面から作っていきましょう。

```

Plane の Draw はこんな感じです

```

void
Plane::Draw(){
    DirectX12& dx=DirectX12::GetInstance();
    auto cmdlist=dx.GetCommandList();
    cmdlist->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
    cmdlist->IASetVertexBuffers(0,1,_vbview.get());
    cmdlist->DrawInstanced(4,1,0,0);
}

```

なのですが、この Draw 系を動かす前にやっておかねばならぬことが一つあって、それはグラフィックスパイプラインステートの切り替えです。

先ほどの PrimitiveCreator の SetPrimitiveDrawMode の中身ではなにをやるのかというと、

```
void
```

```
PrimitiveCreator::SetPrimitiveDrawMode(){
    DirectX12::GetInstance().GetCommandList()->SetPipelineState(_pipelineState);
}
```

この関数を通常 Draw が終わった後にでも呼ぶわけです。

えーと、もしかしたらプリミティブ用のシェーダも書けないかもしれませんので言っておくと

```
//プリミティブ表示用頂点シェーダ
PrimOutput PrimitiveVS(float4 pos:POSITION, float3 normal:NORMAL, float3 color:
COLOR, float2 uv:TEXCOORD)
{
    PrimOutput o;
    o.svpos=mul(mul(viewproj,world),pos);
    return o;
}
```

```
//ボーン表示用ピクセルシェーダ
float4 PrimitivePS(PrimOutput inp):SV_Target
{
    return float4(1,1,1,1);
}
```

こんな感じ。レイアウトの方は、これに合わせて作っておいてくれ。

レイアウトとシェーダがきっちりあっていれば、グラフィックスパイプラインステートの生成は成功するはずなので、それが成功するまで試行錯誤してみてください。

シャドウマップ本編

手順

では、今まで何度もなんとなく喋っていますが、手順を言います。

1. ライト側から撮影
2. 1の結果を深度のみ現像(DSVのみ指定してレンダリング→SRVとして使用)→LVSERV
3. 通常のカメラ側から撮影
4. 現像する

- (ア) ピクセルシェーダにて対象の3D座標を得る(SVPOSではなくPOS)(x,y,z)
- (イ) ↑のPOSをライトビュー変換かける($(x,y,z) \Rightarrow (x',y',z')$)
- (ウ) ↑で得た x',y' をuv座標に変換 $(+1,-1) * (0.5 - 0.5)$
- (エ) ↑のLVSRRのuv座標部分の値と z' を比較し z' が大きいなら暗くする

以上

ライトビュー行列を追加

```
struct TransformMatrices{ //float68=272byte
    DirectX::XMMATRIX world; //float16
    DirectX::XMMATRIX camera; //float16
    DirectX::XMMATRIX wvp; //float16
    DirectX::XMMATRIX lvp; //float16: ライトビュー行列
    DirectX::XMFLOAT4 eye; //float4
};
```

ちなみにこ↑こ↓で総計 256 を越えます。コンスタント/バッファ君は当然の権利のように 512 バイト分食いつぶすで。

さらにシェーダとの辻褷を合わせるために関連する hlsl 側も合わせて変更
//定数レジスター

```
cbuffer Mat : register(b0){
    matrix world; //ワールド
    matrix view; //ビュー
    matrix wvp; //合成済み
    matrix lvp; //ライトビュープロジェクション
    float4 peye; //視点,
};
```

ひとまずは何も起きません。

バッファの確保

まあデジカメもメモリがなかつたら撮影できないわけです。ですから、そのためのメモリを確保します。作り方は既に作っている深度バッファの部分を参考にしてください。

```
//シャドウマップ用
ComPtr<ID3D12DescriptorHeap> _shadowDsvHeap;
ComPtr<ID3D12DescriptorHeap> _shadowSrvHeap;
ComPtr<ID3D12Resource> _shadowBuffer;
void CreateShadowmap();
```

そしてそのメモリはテクスチャとして使用する必要があるためテクスチャとしてメモリの確保を行います。また、いつものRGBAではなく「深度値(Z値)」を書き込むためのものであるためいつものテクスチャとはちょっと指定が違ってきます。

```
D3D12_DESCRIPTOR_HEAP_DESC desc={};  
desc.NumDescriptors=1;  
desc.Type=D3D12_DESCRIPTOR_HEAP_TYPE_DSV;//深度ステンシル  
//深度値用デスクリプターヒープの作成  
result=dev->CreateDescriptorHeap(&shadowmapHD,IID_PPV_ARGS(&_shadowMapHeap));
```

デスクリプターヒープはこんなもんでいいんですが、問題は深度バッファのほう。
通常の深度バッファならば幅と高さは今見ている画面と同じでいいのですが、なにぶん光でござりますので、画面がワイドだからといって、光はワイドにやならんのですよ。

ということで縦横が同じようなサイズでレンダリングします。

```
size_t ssize=max(WINDOW_WIDTH,WINDOW_HEIGHT);  
こういう感じでかい方を取って来て…うーん。2の累乗サイズにしてみましょうか。  
///与えられた数値を2の乗数に切り上げる(32ビット版)  
size_t RoundupPowerOf2(size_t size){  
    size_t bit=0x8000000;  
    for(size_t i=31;i>=0;-i){ //1個ずつビットを下ろしていく  
        if(size&bit)break;//立っていたらそこで抜ける  
        bit>>=1;  
    }  
    return bit<<1;//ひとつ行き過ぎてるんで戻る  
}
```

みたいけな関数を用意して…と、しつつ書いたけど、やってる事の意味はお分かりだろうか？一番左のビットを探し出してそれを左に1ずらしているのだ。…あー、しまった。ちょうど 2^n の場合に余計なサイズになる(16のとき32になる)な、これは…という事で、この後は自分で考えててくれたまえ。分からぬい？いや…これくらい自分でできないと、そろそろ独り立ちせんとなアーッ!!!

うーん。

わからんのか、この戯けがアーッ!!いや、別にif文使っても良くてよ？あと僕はあまりとか~

は使いたくないので最後の戻り値の部分をこうしました。

```
return size+((bit<<1)-size)%bit;
return bit+(bit%size); //
```

なんでなのは各自考えることと、パズルだと思って、自分のやり方をあみ出してみてください。
ともかく関数ができたたら、

```
auto size=Dx12HelperFunc::RoundupPowerOf2(max(wsize.w,wsize.h));
でサイズ測っておいて
D3D12_RESOURCE_DESC resDesc={ };
resDesc.Dimension=D3D12_RESOURCE_DIMENSION_TEXTURE2D;
resDesc.DepthOrArraySize=1;
resDesc.Width=size;
resDesc.Height=size;
resDesc.Format=DXGI_FORMAT_R32_TYPELESS;
resDesc.SampleDesc.Count=1;
resDesc.SampleDesc.Quality=0;
resDesc.Flags=D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;
resDesc.Layout=D3D12_TEXTURE_LAYOUT_UNKNOWN;
resDesc.MipLevels=1;
resDesc.Alignment=0;
フォーマットを R32_TYPELESS にするの忘れずに。
```

あとは既に作っている深度/dffファブリックと同じように作ります。

```
//DSVヒープ作成
D3D12_DESCRIPTOR_HEAP_DESC heapDesc={ };
heapDesc.NumDescriptors=1;
heapDesc.Type=D3D12_DESCRIPTOR_HEAP_TYPE_DSV;
heapDesc.Flags=D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
heapDesc.NodeMask=0;
result=_dev->CreateDescriptorHeap(
    &heapDesc,
    IID_PPV_ARGS(_shadowDsvHeap.GetAddressOf()));
assert(SUCCEEDED(result));
```

```

D3D12_DEPTH_STENCIL_VIEW_DESC dsvdesc={ };
dsvdesc.Format=DXGI_FORMAT_D32_FLOAT;
dsvdesc.ViewDimension=D3D12_DSV_DIMENSION_TEXTURE2D;
dsvdesc.Flags=D3D12_DSV_FLAG_NONE;
dsvdesc.Texture2D.MipSlice=0;
_dev->CreateDepthStencilView(_shadowBuffer.Get(),
    &dsvdesc,
    _shadowDsvHeap->GetCPUDescriptorHandleForHeapStart()
);

```

```

//SRVヒープ作成
heapDesc.Type=D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
heapDesc.Flags=D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
heapDesc.NodeMask=0;
result=_dev->CreateDescriptorHeap(
    &heapDesc,
    IID_PPV_ARGS(_shadowSrvHeap.GetAddressOf()));
assert(SUCCEEDED(result));

```

```

D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc={ };
srvDesc.Format=DXGI_FORMAT_R32_FLOAT;
srvDesc.Shader4ComponentMapping=D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.ViewDimension=D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MipLevels=1;
srvDesc.Texture2D.PlaneSlice=0;
srvDesc.Texture2D.MostDetailedMip=0;

_dev->CreateShaderResourceView(_shadowBuffer.Get(),&srvDesc,_shadowSrvHeap-
>GetCPUDescriptorHandleForHeapStart());

```

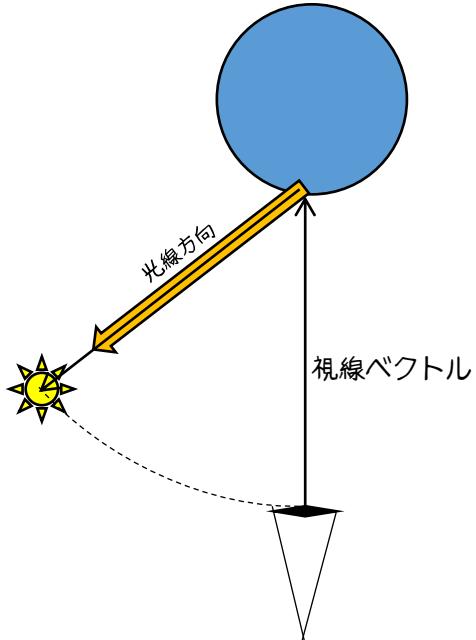
ここまででは通常の深度/バッファ DSV&SRV と同じです。

ライトの「とりあえずの」座標を決める

現在の「ライト」は平行光線なので、「ライトの座標」などというものはありません。でも前から言っているように、シャドウマップのためには「ライト(カメラ)」からの映像を撮っておく必要がありますので「ライトのとりあえずの座標」が必要になってきます。

で「ライトの座標」は平行光源なので、絶対座標ではなく相対座標として考えるようにならし
よう。

方向はもちろん今の平行光線のままでいいのですが…一つのアイデアとして



こういう図で考えてみてください。影の解像度があまり不自然にならないためにはだいたい
始点と注視点の距離を統一しておけばいいでしょう。

つまり初めに始点と注視点の距離を測っておき、注視点に正規化済み光線ベクトル*距離を
足せば、求めたい「光源の位置」をねつ造できます。それっぽければいいのです。

つまり…

光源の位置については

```
lightpos = toLight*(target-eye).Magnitude();
```

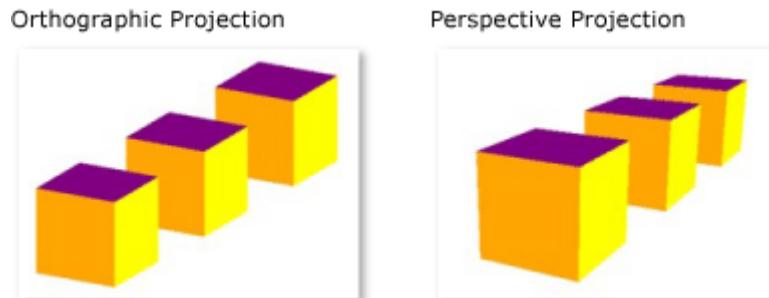
という風に考えればいい事がわかります。なお、注視点はそのままでいいでしょう。

で、これを元にライトビューとライトプロジェクション行列を作ります。ライトビューは簡単
ですね。さっき作った lightpos を視点として XMMatrixLookAtLH すればいいのです。

次にライトプロジェクションですが、XMMatrixPerspectiveFovLH を使用する…と言いたいところですが、今回は平行光源なのでペースがかかるっていないとみなします。ペースがかかるない

プロジェクト行列などあるのでしょうか？

あります。パースをかけない時点でプロジェクトとは言えないのですが、どっちみち 2D 空間につぶす必要があるので、この行列も必須なのです。



ちなみに 3D でパースをかけないことなんてあるの？って思うかもしれません、海外のよくあるゲームではウォータービューみたいな 3D ゲームもありますので、知っておいた方がいいですね。



FEZ ってゲームです

それはさておき、パースをかけないための関数は XMMatrixOrthographicLH という関数です。

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

ここにも書いてありますが、パースをかけない変換を正射影と言い、そういう行列を正射影行列と言います。

この関数は

```
XMMATRIX lightproj=XMMatrixOrthographicLH(幅、高さ、ニア、ファー);
```

と言うような指定にします。

この lightview と lightproj を使って、

```
wvp.viewproj=lightview*lightproj;
```

ちなみに幅と高さですが、正射影の場合は 1024 とかにするとかなり小さくなっちゃいますので、40x40 とかでいいと思います。ずいぶん小さいと感じるかもしれません、正射影ってこんなもんなのよね。正直分からないです()

<https://msdn.microsoft.com/ja-jp/magazine/dn745869.aspx>

に解説が書いてはありますが、ここでは 40x20 とかになってます。正直、単位が良く分からんです。一応根拠としては、通常は画面が -1~1 の範囲であるとして、僕のプログラムでは円柱が 15 だけ中心からずれて回転しています。そこから半径 5 の円柱なので、だいたい -20~20 で 40 にしています。

あとはライトビューを見て、カメラの範囲内にオブジェクトが収まるようになってればよいかなと思います。あと、ニアとファーはライトビューからの距離に合わせて設定してください。

さて、このライトビュープロジェクションが影の元として使用されることが分かったところで、今

①ライトビューの描画

②通常描画

という流れになっていると思いますので、通常描画のビュープロジェクションを通常にするようにしてください。

ライトからの描画

ライトからの描画についてですが、余計なものはそぎ落としましょう。マテリアルも必要ありませんし、レンダーターゲットへの書き込みも必要ありません。

つまり

- レンダーターゲットは null。デプスは書き込む
- マテリアルごとにインデックスを分けず、一気に書き込む
- シェーダも頂点操作以外の作業は必要なし

ということでシェーダ lightview.hlsl を作りましょう。

で、描画してみてください。画面に何も映らないので画面の一部に深度を表示しておきましょう。

ここまでで必要になるであろうメンバ変数、メンバ関数を書いておくと…

//シャドウマップ用

ComPtr<ID3D12DescriptorHeap> _shadowDsvHeap;

ComPtr<ID3D12DescriptorHeap> _shadowSrvHeap;

ComPtr<ID3D12Resource> _shadowBuffer;

ComPtr<ID3D12PipelineState> _shadowmapGPS;

ComPtr<ID3D12RootSignature> _shadowmapRS;

void CreateShadowmapHeap(); // ドッファヒープ作って

void CreateShadowmapRS(); // ルートシグチャ作って

void CreateShadowmapGPS(); // グラフィクスピライイン作って

void DrawLightView(); // ライトからの撮影

うまいこと書き込みが成功すると



このようにライトの深度が書き込まれているのがわかります(深度値の描画に関してはやらないでもいいですが、ともかく書き込まれている事は確認しましょう)。あとはこれを利用するだけです。

悪夢の深度値比較つ…!

ここからの話はまさに悪魔的…

そうですなあ…。

以前にライトからの深度値と比較することによって影を作ると言ったな?

何と比較するのだろう?

そりやあもちろんライトからの距離だよね?さて…ではその距離はどう測る?

distance(originalpos, lightpos)

という風にカメラからの距離を測るか?確かにカメラからの距離は測れる…が、これは完全に実距離である。

深度値マップの中に記録されている深度値は0~1の範囲に「正規化」されているのである。さらに言うと、既に画像化(UV&深度値)されている深度値マップのどのピクセルを使ったらいいのか?分からぬ!。

以下のように、二つの問題が浮上する。

- 0~1の深度値と、実際の「距離」をどう比較していいか分からぬ!
- 深度値マップのどのUV値を採用すればいいのかが分からぬ!

深度値と距離を同じ土俵に…

とにかく深度値と距離を同じ土俵に乗せるべきである。でないと…



このように相いれないこととなってしまう。まあ現実の話だと、2次元キャラを3次元に持ってくる方が手取り早い世の中になってしまいつつあるが、今回の影の話では3次元を2次元の世界に持って行った方が手取り早いのである。

どういう事がと言うと、座標の方を深度値の次元に持ってくるわけだ。

三次元空間上の調べたい座標…その座標を仮に pos とする。これは生 3D 座標だから範囲は正規化されていない。これに対して lightviewproj 行列を適用する。

そうするとライトビューにおける「0~1につぶれた座標」が得られる。これで既に撮影済みの深度値と距離が同じ土俵に乗るというわけだ。

とりあえずこれで一つ目の問題(距離をどう比較するのか)が解決するのだろう。

UV 値はどうするのか？

次に UV 値についてだが、座標に対して先ほど説明した lightviewproj 行列をかけることでライトから見た空間の座標になっている。ということは、元の座標が $x:(-1\sim1), y:(-1\sim1), z(0\sim1)$ の範囲内に収まる状況になっている。というわけである。もともと深度マップはこの範囲を画像にしたものなので、xy を UV 値に対応させればいい…

んだけど、そのままというわけにはいかなくてねえ。何でかと言うと、範囲が違う…
 $-1\sim1 \Rightarrow 0\sim1$ にしなければならないし、Y 方向は反転しているため

```
uv=(float2(1,-1)+shadowpos.xy*float2(0.5,-0.5));
```

となる。いつもの計算だから分かりますよね？とりあえずこの uv 使って床面に色を付けてみるところなる。



しかし影ではない。単なる深度マップである。深度値を比較しない事には正しい影を落とすことはできない。

さて、比較である。

比較

```
float ld=ライトビュー変換後の座標.z;  
この値と深度マップの値を比較する。つまり
```

```
if(ld > 深度マップ.Sample(smp,uv)){  
    //暗くする  
}
```

で、以下のような影が



たとえばちょっとだけ暗くするなら

```
if(ライトビュー変換座標.z > 取得した深度値){  
    brightness*=0.7f;//暗くする  
}  
return float4(brightness*rgb,1);  
等と書きます。理屈は説明してるから分かりますよね？
```

上の `lightdepth` は既にレンダリングしてロードした部分の値が入っています。こいつと、現在のピクセルの座標を比較。

現在の深度値(カメラから見た Z 値)の方が大きければ影が落ちるという事です。

セルフシャドウ

ちなみにセルフシャドウ(自分自身への影)に関しても同じ理屈でできます。ただ注意点がちょっとあります。

モデル用のシェーダに先ほどの処理を適用すると



なんだこれは…たまげたなあ…。

と言った具合にノイズが入ってしまいます。

こういうノイズの事を「シャドウアクネ」と言うらしいです。ちなみに acne ってのは「にきび」って意味らしいです。

これは何故かと言うと、ライトが最初に当たる部分では、深度値が

ライトマップと、ライトビュー変換後座標. Z で一致してしまうために

ライトビュー変換座標. Z > 取得した深度値

の比較時に、予め取得しておいた深度値と「フライティング(深度フライティング)」的な事が発生するからです。つまり浮動小数点演算の結果が場所によって変わってしまいます。

というわけで、せこいやり方ですが、ちょっと下駄($\text{epsilon}=0.0005f$ くらい)を履かせて
if($\text{ld} > \text{lightdepth} + \text{epsilon}$) {
 $\text{dbright} *= 0.7f$; //暗くする
}

とします。

ちなみに epsilon というのは Wikipedia によれば

- 数学で、 ε - δ 論法などで見られるように非常に小さな数を表す記号としてよく用いられる。
らしいです。

ともかく、こうするとセルフシャドウもキレイに入って



こうなります(しかしこれノイジーですね)

全体的なノイズはなくなったんですが、影のエッジがガタガタしてますね。解像度の問題だつたり、光線に平行に近い面だと伸びてしまいます。これを軽減するには

- シャドウマップの解像度を上げる

- 深度傾斜/ダイアス
- カスケードシャドウマップ(CSM)などのヴァリアンスシャドウマップ(VSM)などのを利用する

と、色々と対処がありますが、それはもう少し後にしましょう。

簡単な PCF(percentage-closest-filtering)で影をマシに

これは本当は「半影」を得るための手法らしいですが、半影気味にすることで、いまのジャギジャギしている影を少しあはマシにしたいと思います。

テクスチャに対する考え方もちよつと変わってきます。簡単に言うとエッジ的な部分をぼかすための手法です。実装の手順を簡単に言うと

- ①まずサンプラー増やす(シャドウマップ用)
- ②サンプラーの関数を LESS_EQ にする。
- ③サンプラーのフィルタを COMPARISON_MIN_MAG_MIP_LINEAR にする
- ④シェーダ側で "SamperComparisonState" を s2 として追加
- ⑤テクスチャの Sample ではなく、SampleCmp or SampleCmpLevelZero を用いた値を取得(フィルタによりエッジ部分がぼけている)
- ⑥この値を用いて lerp する。例えば 0.5~1.0 に対して 1.0 がリニアかければ影のエッジ部分が 0.5~1.0 に補間され、影がぼける。

さて、昨年まで謎のままにしてたこの「COMPARISON」ですが、英語で比較を意味します。
テクスチャはただ色を取得したり、輝度を取得してくるだけではなく、「比較の結果」を返すこともできます。
比較の結果とはもちろん既存の色との比較になるわけですが…今回は深度値なのと、そもそもテクスチャの内容取得関数名が違います。

通常: テクスチャ.Sample

影に使う: テクスチャ.SampleCmp

普通の Sample は色情報を返すものでしたので、float4 もしくは float が帰ってきます。本来であれば SampleCmp は比較の結果を 0.0 か 1.0 かで返します。そして中間の位置に関してはサンプラーのフィルタ(ポイントや、ダイリニアなどの補間形式)が適用されます。

つまり 0.0 と 1.0 のちょうど中間の場合、ポイントなら 0.0 や 1.0 がどちらかであり、バイリニアであれば 0.5 になります。これを利用します。

そこでまずシャドウマップ用の比較サンプラーを用意します。

普通のサンプラーではダメで `SamplerComparisonState` という比較サンプラーを使用します。

```
SamplerComparisonState shadowSmp : register(s2);
```

そして先ほど紹介した `SampleCmpLevelZero` 関数を使用します。

```
lightDepthTex.SampleCmpLevelZero(shadowSmp, UV座標, 比較対象の値);
```

比較サンプラーの比較対象の値といふのが、最後の引数になります。で、この「比較対象の値」と指定 UV の値を比較して 1.0 や 0.0 が返ります。通常のサンプラーだと本当に 1.0 や 0.0 なので、この結果にバイリニアがかかりますので、実際の値は 0.0~1.0 の間にあります。

```
float3 posFromLightVP=input.tpos.xyz / input.tpos.w;
float2 shadowUV = (input.tpos.xy/input.tpos.w+float2(1,-1))*float2(0.5,-0.5);
float depthFromLight = lightDepthTex.SampleCmpLevelZero(
    shadowSmp,
    shadowUV,
    posFromLightVP.z-0.005f);
```

注意点…これは Cmp に限りませんが、ライト行列が `PerspectiveFovLH` の場合は 同次座標で割るのを忘れないでください。ここはかなりややこしい話ですが、プロジェクション行列を適用した後の座標とかかわっています。プロジェクション行列を今一度確認してみましょう。

$$(x \ y \ z \ 1) \begin{pmatrix} \cot\left(\frac{\text{fov}}{2}\right) \frac{\text{screen}_h}{\text{screen}_w} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\text{fov}}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{z_{far}}{z_{far} - z_{near}} & 1 \\ 0 & 0 & -\frac{z_{near}z_{far}}{z_{far} - z_{near}} & 0 \end{pmatrix}$$

このようになっています。なお `fov` は画角です。`cot` は `tan` の逆数です。仮に今が画角 90° とします。そうすると $\cot\left(\frac{90^\circ}{2}\right) = 1$ です。

$$(x \ y \ z \ 1) \begin{pmatrix} \frac{screen_h}{screen_w} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{z_{far}}{z_{far} - z_{near}} & 1 \\ 0 & 0 & -\frac{z_{near}z_{far}}{z_{far} - z_{near}} & 0 \end{pmatrix}$$

なお $A_{spect} = \frac{screen_w}{screen_h}$ とすると

これで計算すると $(x, y, z, 1)$ が $\left(\frac{x}{A_{spect}} \ y \ \frac{zz_{far}}{z_{far} - z_{near}} - \frac{z_{near}z_{far}}{z_{far} - z_{near}} \ z \right)$ となります。Z 項をまとめると

$\left(\frac{x}{A_{spect}} \ y \ \frac{z_{far}(z - z_{near})}{z_{far} - z_{near}} \ z \right)$ となります。仮に入力が $z = z_{near}$ だった場合は

$\left(\frac{x}{A_{spect}} \ y \ 0 \ z_{near} \right)$ です。

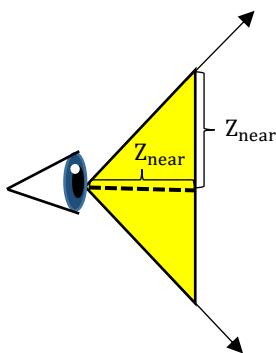
仮に入力が $z = z_{far}$ だった場合は $\left(\frac{x}{A_{spect}} \ y \ \frac{z_{far}(z_{far} - z_{near})}{z_{far} - z_{near}} \ z_{far} \right) = \left(\frac{x}{A_{spect}} \ y \ z_{far} \ z_{far} \right)$

となります。

そして頂点シェーダからラスタライザに渡されるときにシステム座標は同次座標(W 項)で XYZ が割られます。結果的に

$$\left(\frac{x}{A_{spect}z_{near}} \ \frac{y}{z_{near}} \ 0 \ 1 \right)$$

となります。ここで near 側の「画面左上の座標」について考えます。現在の例では画角 90° でやっているため図のように上座標は $y = Z_{near}$ と同じになります。



左座標は縦 × アスペクト比となりますので $x = -A_{spect}Z_{near}$ となります。これを先ほどの結果に代入すると

$$\left(\frac{-A_{spect}Z_{near}}{A_{spect}Z_{near}} \ \frac{z_{near}}{z_{near}} \ 0 \ 1 \right) = (-1, 1, 0, 1)$$

となります。同様に far も

ラスタライズ時に $\begin{pmatrix} \frac{x}{A_{spect}} & y & z_{far} & z_{far} \end{pmatrix}$ の z_{far} で割られるため $\begin{pmatrix} \frac{x}{A_{spect}z_{far}} & \frac{y}{z_{far}} & 1 & 1 \end{pmatrix}$ となり、

far 時の左上は $x = -A_{spect}z_{far}, y = z_{far}$ となるため $\begin{pmatrix} -A_{spect}z_{far} & \frac{z_{far}}{z_{far}} & 1 & 1 \end{pmatrix} = (-1 \ 1 \ 1 \ 1)$

で計算の辻褄が合っていることが分かります。疑わしい人はほかの画角、ほかのアスペクト比、適当な座標で手計算で検証することをお勧めします。

ちなみに Ortho の場合は

$$\begin{pmatrix} \frac{2}{Screen_w} & 0 & 0 & 0 \\ 0 & \frac{2}{Screen_h} & 0 & 0 \\ 0 & 0 & \frac{1}{z_{far} - z_{near}} & 0 \\ 0 & 0 & -\frac{z_{near}}{z_{far} - z_{near}} & 1 \end{pmatrix}$$

こうです。これはどうやっても $w=1$ になるため w で割ってやる必要はありません。ortho も perspective も両方対応させるつもりなら w で割っておきましょう。ただやみくもに割ればいいという物でもありませんので、そこだけはご注意ください。

Effekseer 組み込み

さあ、割としんどかった Effekseer 組み込みをやっていきます。何がしんどいってプログラム以前の部分がしんどいんですね。

ちなみにものごつ時間がかかるので、バックグラウンドで作業したほうがいいです。
あと、手順間違えると Git からやり直しが多いため、しっかり手順を見ましょう。

とにかく必要なものを集めてくる

SWD 氏のアドバイス

「授業の形式がわからないのですなんとも言えないのですが、git から取得して git-lfs でバイナリを取ってきて依存関係は Submodule で取得して Cmake のビルドが常に通るようにしています。これができないのならビルド済みの Lib を配った方がいいかもしれません。」

EffekseerForDX12 インストール手順

①自分のどこかのフォルダに

<https://github.com/effekseer/Effekseer.git>

のクローンを作る。

②`https://github.com/Kitware/CMake/releases/download/v3.16.0/cmake-3.16.0-win64-x64.zip`
から CMAKE を取得

SubModuleはどうやつたらいいんだろう….`gitmodules` が関係しているのだろうか…

「他人のリポジトリをビルドするときには、clone 直後に `git update submodule --init --recursive` と打つ、だけ覚えておけばいいかと」

「現場だとそれができないと仕事にならない技術がもっと増えてきてます…」

怒られました。しいーまシェーン!!

ちなみに実際には `git submodule update ~` なので注意。

というわけで③版以降の手順

③クローンしたディレクトリ上で "`git submodule update --init --recursive`" として、
実行

④(*ここ要注意!!慎重にやってください)CMAKE します。

しますが「ソースセットのバージョン」に注意。デフォルトで VS2019 となっていますので、もし
VS2017 でビルドする場合は明示的に指定する必要があります。コマンドラインでは…

`cmake -G "Visual Studio 15 Win64" CMakeLists.txt`

とします。もちろん 2019 用ならデフォルトでいいです。この辺は CMAKE 使うときは気をつけ
ましょう。間違うと最初からやり直しです(ぼく何回もやり直しになりました)

⑤`Effekseer.sln` が出来ているはずなので、起動しソリューションをビルドする

ここまでできればデバッグの Lib は

`Effekseer\Dev\Cpp\Debug`

ありますし、レンダラは

`Effekseer\Dev\Cpp\EffekseerRendererDX12\Debug`

あります。

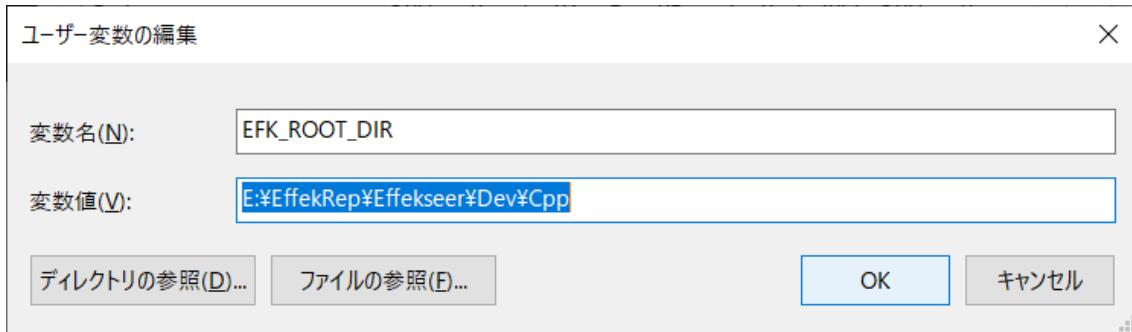
あとサードパーティが

`Effekseer\Dev\Cpp\3rdParty`

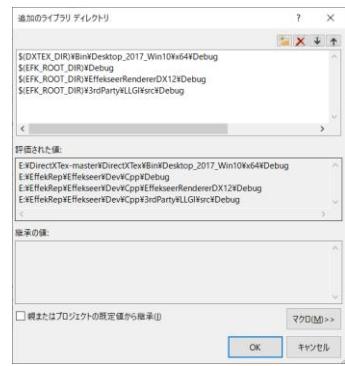
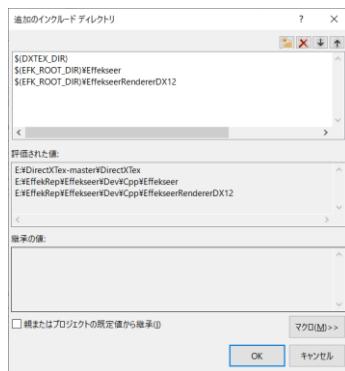
あります。

各リンク、環境変数を設定しておきましょう。

とりあえず僕は Cpp までを環境変数に設定しておいて…



こうしました。あとはビルド時にエラー言われるとおりに片っ端から設定していきます。



ここからが、クソみたれに大変です。

一旦実験までやってみよう

インクルード

```
#include<Effekseer.h>
#include<EffekseerRendererDX12.h>
```

(中略)

```
//Effekseerの基盤部分を初期化
auto format = DXGI_FORMAT_R8G8B8A8_UNORM;
efkRenderer = ::EffekseerRendererDX12::Create(_dx12->Device(), _dx12->CmdQue(), 2,&format, 1,
```

```

false, false, 2000);
efkManager = ::Effekseer::Manager::Create(2000);

// 描画用インスタンスから描画機能を設定
efkManager->SetSpriteRenderer(efkRenderer->CreateSpriteRenderer());
efkManager->SetRibbonRenderer(efkRenderer->CreateRibbonRenderer());
efkManager->SetRingRenderer(efkRenderer->CreateRingRenderer());
efkManager->SetTrackRenderer(efkRenderer->CreateTrackRenderer());
efkManager->SetModelRenderer(efkRenderer->CreateModelRenderer());

// 描画用インスタンスからテクスチャの読み込み機能を設定
// 独自拡張可能、現在はファイルから読み込んでいる。
efkManager->SetTextureLoader(efkRenderer->CreateTextureLoader());
efkManager->SetModelLoader(efkRenderer->CreateModelLoader());

// エフェクト発生位置を設定
auto efkPos = ::Effekseer::Vector3D(0, 0.0f, 0.0f);

//メモリプール
efkMemoryPool= EffekseerRendererDX12::CreateSingleFrameMemoryPool(efkRenderer);
//コマンドリスト作成
efkCmdList=EffekseerRendererDX12::CreateCommandList(efkRenderer, efkMemoryPool);
//コマンドリストセット
efkRenderer->SetCommandList(efkCmdList);

// 投影行列を設定
efkRenderer->SetProjectionMatrix(
    ::Effekseer::Matrix4x4().PerspectiveFovLH(90.0f / 180.0f * 3.14f, 1280.f/720.f, 1.0f,
100.0f));

// カメラ行列を設定
efkRenderer->SetCameraMatrix(
    ::Effekseer::Matrix4x4().LookAtRH(::Effekseer::Vector3D(0.0f, 10.0f, -
25.0f), ::Effekseer::Vector3D(0.0f, 15.0f, 0.0f), ::Effekseer::Vector3D(0.0f, 1.0f, 0.0f)));

// エフェクトの読み込み

```

```
effect = Effekseer::Effect::Create(efkManager, (const EFK_CHAR*)L"effect/test.efk");

まずはこのビルドが通るようにしてみて下さい(動くとは言ってない)
ビルドが通るようにするためにには少なくとも Effekseer.lib,
EffekseerRendererDX12.lib, LLGI.lib の3つのライブラリをリンクする必要があります。
```

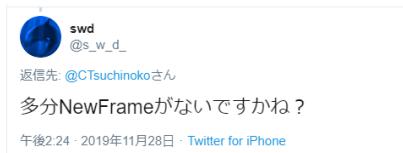
さらに苦戦することになるのですが…

まず、マネージャの Play 関数で再生を開始します。このときハンドルが得られるので変数に格納しておきます。

- BeginCommandList()と EndCommandList()で Draw 系命令を囲む
- マネージャの Draw 関数を呼び出す
- BeginCommandList の前にメモリプールの NewFrame()を呼び出す

これだけ…なんですが、DX12 用のヘルプがどこにもないうえにチュートリアルも何もないのでも、正直「そりや作者はわかるだろうけどさあ…」って状態でした。

聞いて正解。



わかるかこんなもん。しかもメモリプールのメンバだよこいつは。
もう一度言う!わかるか!!!といふわけで毎フレームループの部分を書きますが、

```
if (efkManager->Exists(efkHandle)) {
    effekManager->StopEffect(efkHandle);
}

efkHandle = effekManager->Play(effect, Effekseer::Vector3D(0, 0, 0));
```

こう書いてスペースキー押下時に発生させつつ
efkMemoryPool->NewFrame();
::EffekseerRendererDX12::BeginCommandList(efkCmdList, _dx12->CmdList());
efkRenderer->BeginRendering();
efkManager->Draw();
efkRenderer->EndRendering();

```
::EffekseerRendererDX12::EndCommandList(efkCmdList);
```

このようにして描画の処理を行います。

DDS ファイルのロード

ちょっともうおまけ的な流れになって来てるんですが、未だに DDS(DXT) 等の理解が必要だったりするので、一応この話も入れておきます。

UE4 や Unity などのエンジンの時代だからこそ知つておく必要があるんですよね…知らないと割と痛い目に遭う事もあるので知つておきましょう。

ちなみに「俺はスマホゲーソシャゲーに行くから関係ないもーん」と思つてる人もいるかもしねないけど、どう考へてもスマホゲーの方がこの概念は大事なので、知つておこう。

もし DirectX 関係をいじつていくと、そのうち dds フォーマットと言うのに出くわすだろう。これは bmp や png などと同じように「画像ファイル」の一種である。

ちなみに dds は何の略かと言うと「DirectDraw Surface」の略である。あれ？ やっぱりスマホゲー関係ないじゃん？ と思うかもしれない。いやいや、話を最後まで聞け。

DXTC、ETC、PVRTC

この圧縮方式は、細部の差こそあれスマホゲームに大いに活用されている。というのもこの DDS の圧縮方式は DXTC と言って、後で話す仕組みによって圧縮されている。

通常であれば PNG や JPG に圧縮したところで、ロードして GPU に転送する時点で…ここまでやってきた皆様ならば、お分かりのように、一度 RGBA に「展開(解凍)」されてしまうのだ。

つまり、png などでどんなに圧縮していたとしても内部メモリ的には、32bit ビットマップと同じ容量を食っているのと変わらないわけだ。

まずここは押さえておいてくれ。

そこで考え出されたのは DXTC(BC) というもので、これは $\frac{1}{8} \sim \frac{1}{4}$ くらいの圧縮率で、特筆すべきはこの圧縮率のまま GPU へ転送できるという部分だ。

メモリが貧弱なスマホゲームにおいてはこれを使わない手はなく、そもそも Unity の設定で「モバイル」ってやると、画像フォーマットが勝手にこの圧縮方式となってしまうのだ!!!

ちなみに Android 系では ETC という圧縮方式が使用されている。圧縮方式は DXTC(BC) とほぼ同じなのだが、微妙に違う。

さらにiphone系ではPVRTCという形式が使用されており、これも概念的にはほぼ同じである。

それぞれ細かい解説がOptpixのブログに書かれています(俺も自分のブログで昔書いたけど、記事を思い出せない…)

DXTC(BC)

http://www.webtech.co.jp/blog/optpix_labs/format/4013/

ETC

http://www.webtech.co.jp/blog/optpix_labs/format/5882/

PVRTC

http://www.webtech.co.jp/blog/optpix_labs/format/4930/

それぞれしっかりとご覧になれば「あつ…(察し)ふーん」と気づくと思いますが、ほっとんど同じです。

…と、非常に効果が高いため特にスマホゲームでバンバン使用されているのですが、こいつにも落とし穴があるのです。

それは…非常に劣化しやすいのです。

あー、テクスチャファイルは可逆圧縮と、不可逆圧縮があるのは知ってるよね？

PNGなんかは可逆圧縮なんだけど、JPGやらGIFは不可逆圧縮。DDSも不可逆圧縮。

で、不可逆圧縮にもパターンがあつて大きく分けると

- 色数を減らす圧縮(GIFなど)
 - 色数は減らないけど汚くなる圧縮(JPGなど)
- があります。

例えばGIFなんかは使用できる色の数を256未満に限定し、この256のパレット(インデックスで指定できる)を作る。この256パレットは32bitでも可である。ただし絵の部分はRGBA32bitではなく256インデックス8bitとする。

それによりパレット部分を無視すれば圧縮率は理論上1/4となる。

ただし、当然ながら色数はもともと最大 4294967296 色表現できたものが 256 色しか使えないため、使用できる色数は減る。そういうことだ。ただ、色鮮やかでピクセル数が多めの写真画像ならともかく、ドット絵やキャラ絵では最大色数を使う事はあまりないだろう。

特にドット絵などでは 256 色すら使いきれないことは、ドット絵を描いたことのある人なら分かるだろう？（FFV やメタルスラッグは 256 に抑えてるのは驚異的だが…ちなみに初代マリオなんて 4 色だぞ？）

ともかく、2D のレトロゲームにおいては「色をインデックスで管理して減らす」という方法はかなり効果的であった事が分かる。そもそも DirectX5 くらいまでは「パレット」がハードウェア的に標準搭載されていたはず。

はい、ちょっと長くなりましたが、次に話す圧縮方式は色数を減らすのではなく品質を落として圧縮するというものです。

ネットに転がってた解説の奴を引っ張ってきましたが…/ 分かりづらいかもせんが、エッジ近辺に細かいノイズが入っているのがわかるでしょうか？（この辺を「ほとんど見た目が同じだからいいじゃん」と言うと、ゲームアーティストにぶん殴られますので注意しましょう。）

ちょっと話が逸れますが、この手の非可逆圧縮の圧縮方式は 1 ピクセルニ 1 色データという対応関係は当てはまりません。

こういう「情報の持ち方」が想像と違うものがあるのがテクノロジーの世界なので「なるほど、そういうものもあるのか」と言う風に思ってください。

JPEG はまず「フーリエ変換」という方法を使って画像を周波数成分に分解します。

<https://www.slideshare.net/ginrou799/ss-46355460>

ですから JPEG の「情報量」は周波数の種類の数で決まります。例えば画像の色が单一 Sin 波状態になってしまえば情報量は非常に少くなります。

しかし通常であれば無限に近い周波数の種類が必要となるため、圧縮しようとすると確実に劣化します。JPG には品質の指定があると思いますが、これは低品質になればなるほど、高周波成分からカットされます → 周波数の数が減る → 圧縮。

まあこの説明はおおざっぱなので、専門で勉強したい人は画像処理の本でも読んで勉強しましょう。

で、本題に入りますがDXTC,ETC,PVRTCは

- 色数が減るうえに品質も下がる

という、クオリティの面からだけ言うと使いたくないような圧縮なんですよね。とはいっても使われてるのは理由があるのですが、それはもう少し後で話します。

まずアルファ以外の24bitが16bitに色数が減ります。

RGB(8,8,8)がRGB(5,6,5)

になります。まずここで色の劣化。ちなみにGが6なのは15だとさりが悪くて、16にしようとなつた時に、人間の識別範囲が広いのが縁だからという事です。

そしてさらに劣化するんですがちょっとややこしいのですが…4x4ピクセルを2つの代表色とインデックス()で表現するのです。つまり、

$16 \times 24\text{bit} \Rightarrow 2 \times 16\text{bit} + 16 \times 2\text{bit}$

$384\text{bit} \Rightarrow 64\text{bit}$

つまり1/6の情報量になります。ちなみに α 値も8bit \rightarrow 4bitにされてしまします。 α 値に関しては、全ピクセルに4bitが割り当たるため、

$16 \times 32\text{bit} \Rightarrow 2 \times 16\text{bit} + 16 \times 2\text{bit} + 16 \times 4\text{bit}$

$512\text{bit} \Rightarrow 128\text{bit}$

圧縮率は1/4となります。

なお、これはDXT2~5の場合で、DXT1の時はさらに減ります。

DXT1の時は α がないのでそもそも

512→64

で圧縮率1/8であり、アルファが入ったとしても1ビット(抜くか抜かないか)しか使わないの
で、

565→555とし、残りの1bitをアルファとして使用します。

と、まあ結構な圧縮率になるのですが、先ほども言ったように割とクソ品質です。

そして前にも言ったようにゲームエンジンにおいて「モバイル」と指定すると、テクスチャリ
ソースの大半が自動的にこの形式になります。3Dのテクスチャであればあまり違和感がない
のですが、UIとか、2Dのドット絵までこの形式にされると問題です。

で、意外とこの知識がないスマホゲーム開発者が多くて「品質が〜!!」とか言ってるんですが
もうね…。

うん、ここで2Dドットゲームばかり開発してきた某ゲーム会社の人間からすると、「こういう
時こそ温故知新じゃねーの?」って思うわけですよ。

つまり「パレット」の復活を考えるわけです。一応、そのテクスチャの総色数が256未満であれ
ばGIF同様にパレット圧縮できるはずだと思うわけです。ただ、これも時と場合…適切な状況
以外でやってもあまり効果がないことは心に留めておきましょう。

このパレットを使うべき時は…

パレットの方がいい！

- ドット絵的な場合(←DXTC等でやると確実に劣化するためドット絵にとって大事な1ド
ットがつぶれてしまう。ドット絵の場合は色数が少ないというのもある)
- 画像サイズ»色数であること(256色でサイズが16x16未満の絵とかだと意味がない!…
どころかサイズが増える。ドット絵なら関連アニメーションをまとめた状態とかだと効
果が高い)
- エッジをきちんと意図通りに表現した場合
- バイリニアフィルタなどのフィルタをかけてない場合

DXTCの方がいい！

- 写真的なものや様々な色がグラデーションしている場合

- エッジがそれほど明確ではない場合
- 3Dに張り付けるテクスチャ(どの道大いに拡縮するからあまり重要ではないことが多い)
- バイリニアフィルタがかかるついている場合(インデックスを補間したら何が起るか…分かるね?分からぬ?インデックス1番とインデックス2番を補間して1.5になったら困るでしょ?インデックスが整数型以外になることの恐ろしさが分からぬかな?)

てな感じです。ちなみにパレットに関してはLUTの応用でできます…というかパレットがLUTそのものなので別に難しくもないかなと思ひます。

で、かなり本題から外れてしましましたが実際のDDSファイルのロードしてみようと思ひます。

前の時にBMPやPNGをロードするついでにDDSローダも作ったでしょ?それをそのまま使えばいい。

ただ、これを実装しようとして最初に直面した問題が…

DDSは基本的にミップマップを持っている…
この特性を忘れておりました。「ミップマップは考えないようにしよう…」などと言っておりましたが、ご覧の通り

全部で11個あるようです。

今までのように1枚だけのデータでそれがすべてであった時と、ミップマップのように複数のデータが格納されている時で、扱いが同じはずがねーだろ!!というわけですね。

まあ、当たり前の話なのですが、ゲームプログラムのリソースの仕様とか知らないとだいたいそう認識する。

問題はこの11個の画像のうちどれを使用すべきなのかという事だ。どれを選べばいいのだろう…というか、選ぶのまでこっちでやってたらミップマップとかやってらんねーわけですよ。

というわけで「11個のミップマップ」という事でテクスチャを作りたいわけですよ。

/で、まあ色々と苦労して、ミップマップ状態データを貼り付けてこうなりました。

ご覧のありさまだよ!

さて、何が起きているのでしょうか?

フォーマットが違うんですよ。

いつもの R8G8B8A8_UNORM とかじゃないんですね。既に話したようにグラボまで圧縮フォーマットを持ち込むので、特殊なフォーマットになっています。

DXGI_FORMAT_BC1_UNORM
DXGI_FORMAT_BC2_UNORM
DXGI_FORMAT_BC3_UNORM
DXGI_FORMAT_BC4_UNORM
DXGI_FORMAT_BC5_UNORM

などですが、ひとまずこれも 4 バイトで計算しておいてください。画面上に最終的に表示する際に 4 バイトなので、4 バイト WriteToSubresource を使う時は 4 バイトの設定にしておいてください。

臨時の話

リアルタイムレイトレーシングのセッションについて

この前の CEDEC+KYUSHU でリアルタイムレイトレーシングのトーン表現とシャドウについて的な講演がありました。その資料が上がっていました。

<https://speakerdeck.com/katsushisuzuki/cedec-plus-kyushu2019-toon-rendering-by-directx-raytracing>

もちろんまだ僕らは DXR やってないので①～③については「なんのこっちゃ？」かもしれません。④、⑤については少なくとも参考になるかなと思います。

- ①レンダリングフロー(DXR x Compute)
- ②DXR のきほん
- ③AO とアニメーション
- ④ドットシェーダ
- ⑤アウトラインシェーダ

ただ DXR の部分でもちょっと見ておきたいのがこのスライド

▶ レンダリングフロー全体

DELIGHT WORKS

DXR



Compute



表示までにかなりの手間(パス数)がかかっているのが分かるかなと思います。Unity や UE4 でもしつれっとポストエフェクトつかってますのでお気を付けください。

次にドットシェーダですが、今回のこの説明は今までの資料でもありそうでなかつたものなので(いわゆる Tri-Planer を分かりやすく説明してくれている)読んでおいたほうがいいかと思います。



2Dのドット



UV座標をもとに描画

3Dのドット



XYZ座標をもとに描画

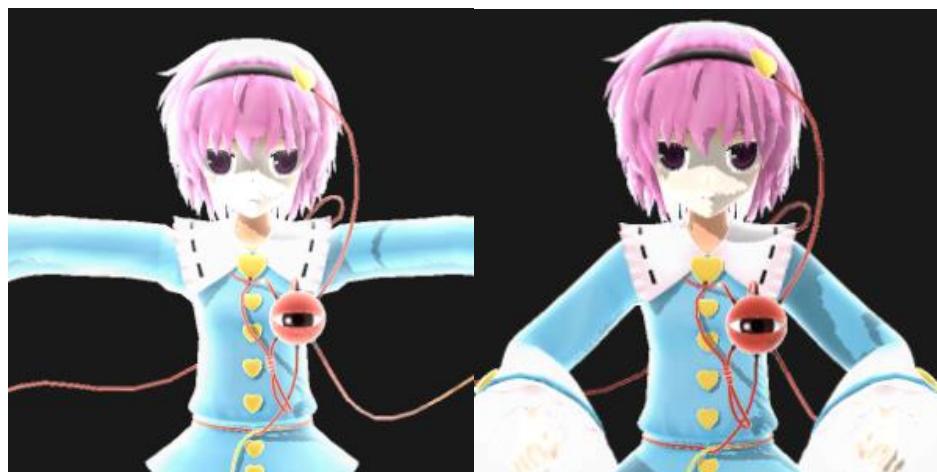
輪郭線的なやつ

色々とやり方あります。

試しにリムライトの考え方を応用してみよう

まず、リムライトとは逆光というか、背後から光を当ててキャラクターの側面を明るくする表現手法の事です。

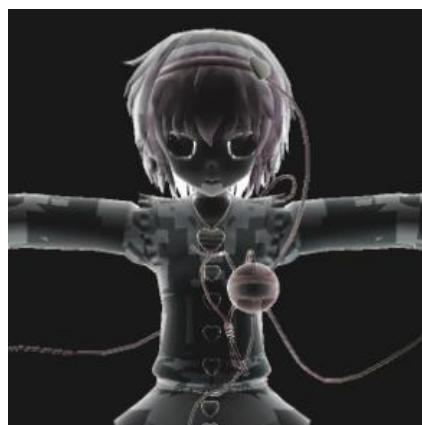
手っ取り早くそういう効果を出すためには…これもそろそろ自分で思いつく頃合いかなとは思うんですが、視線ベクトルと法線ベクトルの内積をとり、それを反転させることで…場合によってはその反転の数値を n 乗することで、リムライトの光漏れの広さを調整できます。



1乗

2乗

ご覧のような効果を得ることができます。なお、これを α 値にも影響が出るようになると…

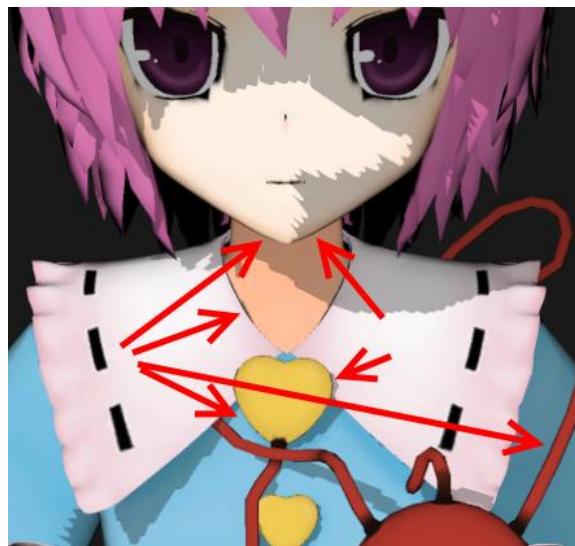


このように幽霊的な感じを出すことができます

なんちゃらソウル的なのを作りたいときにちょっと応用するといいんじゃないでしょうか。

で、本題は輪郭線ですが、この白くなっている部分を黒くするだけで輪郭線みたいになると思

いませんか?やってみましょう…。

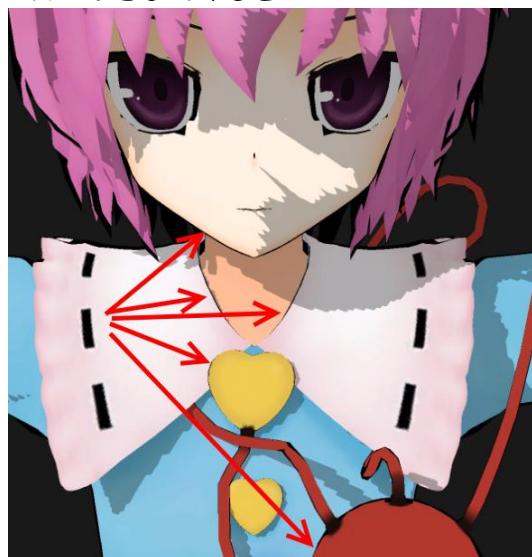


ん…まあ…輪郭線?

このやり方の難点としては

- エッジ近辺で法線が直交付近を向いてないと意味がない
- エッジ近辺で法線がゆるやかに変化していないと意味がない
- エッジを太くしようとするとディフューズっぽいグラデが出る
なので、ちょっとトゥーンの輪郭線に応用するのは向きなようです。

改良してちょっと閾値的な使い方をしてみると…



のようにだいぶエッジっぽくはなります。応用すると大神みたいな演出できそう?できそうじやない?

```

float edge = abs(dot(ray, input.normal)) > 0.25f ? 1 : 0;
return float4(float3(edge, edge, edge) * col * shadow, diffuse.a);

```

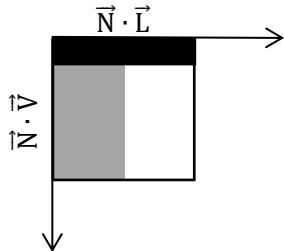
こんな感じで作りました。リムライトをやってみたい人はこれを逆に考えてみてください。

トゥーンマップを用いる方法

ここで書いた輪郭線の手法はあくまでも「手っ取り早く」「それっぽく」の手法である。ガチでやろうとすると実現のための手法は多いし、それなりに手間はかかるして大変だったりします。

ちなみにさっきの輪郭線抽出の例ですが、if文を書くのがイヤな場合はトゥーンテクスチャ側に細工を施しておきます。

よくあるのはこういう構造にしておきます。



ちなみに N は法線ベクトルで L はライトベクトルで、 V はビューベクトルです。やろうとしている事は分かりますよね？そう。 u を N と L の内積にすることで、濃淡に対するトゥーンシェーディングを行い、 v を N と V の内積にすることで、付近を黒く塗りつぶして、それが輪郭線として見えるようになるわけです。これだったら自動的に輪郭線が入り、if文を使わなくても良くなります。初心者はここら辺から入るのがいいでしょう。

じゃあそれ以外の輪郭はと言うと…有名どころで言うと「背面法」と「デプスバッファ法」と「法線バッファ法」があります。

一番わかりやすいのは「背面法(反転法?)」かな…？というわけで背面法から説明します。

背面法(反転法?)

以前にも言ったことがあると思いますが、原則的にポリゴンと言うのは表側が描画され、裏側は描画されません(MMDは両面デフォルト)。で、それがどちらかを決めているのが、頂点が右回りに配置されているか左回りに配置されているかです。まあ、それはともかく「表」と「裏」があ

るわけです。背面法のしくみは

- ①描画すべきモデルをもう一体同じ個所に描画する
- ②この①で描画する際に頂点を頂点の法線方向に少し拡げる
- ③②の面を反転させる
- ④②の面を黒(もしくはエッジに合わせた色)で塗る

という手順になります。さて、①番は簡単でしょう。問題は②番以降ですが、自分で考えて想像つくでしょうか？

簡単ですね。頂点の座標を法線方向に広げるんですから

```
pos.xyz += o.normal*0.2;
```

みたいになります。そうすると



ちょっと太ましくなりました

これが法線方向に広げるという事です。

次に「面を反転させる」ですが、ラスタライザーステートをいじります。ラスタライザーステートをいじるという事はつまり、またパイプラインステートオブジェクトが別に必要という事です。面倒ですねえ。

試しに今の状態で面を反転させてみましょう。どうやつたらいいのでしょうか？どちらの面を表示するかと言うのは「カリング」という指定で行います。現状カリングはオフです。

D3D12_CULL_MODE_NONE

と書かれている部分を…

D3D12_CULL_MODE_FRONT

にします。

そうすると、表面がカリング(表示対象から外す)されるため



ぐちゃぐちゃですが、こういう感じ

裏面だけが描画されているため、手前のサーフェイスは描画されず、奥がわのサーフェイスだけが描画されることになります。これを黒く塗ります。この黒い部分が輪郭線として機能することになります。

そのうえで元のモデルを通常通りに描画すれば「輪郭線のあるモデル」のように見えます。おそらく MMD はこの方法(背面法)を採用していると思われます。

さて、輪郭線モデルとホンマもんのモデルは今話したようにカリングの設定も違いますし、シェーダも変わりますのでまたパイプラインステートオブジェクトが増えます。そろそろこの管理も考えていかないとですね。

ベーシックな PSO を内部に持っておいて、必要な部分だけ修正したものを返す。そして、管理はマネージャ側がやるっていうような、そういうの必要だよね。

とはいえ、そういうのは各自考えていただくとして、とりあえず実装してみます。

```
///輪郭線用パイプラインステートオブジェクト
ID3D12PipelineState* _outlineGPS = nullptr;
ID3DBlob* outlineVS = nullptr; // コンパイル済み頂点シェーダ
ID3DBlob* outlinePS = nullptr; // コンパイル済みピクセルシェーダ
result = CompileVertexShaderFromFile(_T("shader.hlsl"), "OutlineVS", outlineVS);
result = CompilePixelShaderFromFile(_T("shader.hlsl"), "OutlinePS", outlinePS);
gpsDesc.VS = CD3DX12_SHADER_BYTECODE(outlineVS);
gpsDesc.PS = CD3DX12_SHADER_BYTECODE(outlinePS);
gpsDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_FILL_MODE_SOLID,
D3D12_CULL_MODE_FRONT, false, 0, 0, 0, false, false, 0,
```

```
D3D12_CONSERVATIVE_RASTERIZATION_MODE_OFF);  
result = dev->CreateGraphicsPipelineState(&gpsDesc, IID_PPV_ARGS(&_outlineGPS));
```

はい、↑のコードのシェーダ側なんですが、

```
//輪郭線頂点シェーダ  
Output OutlineVS(float4 pos : POSITION, float3 normal : NORMAL, float2 uv :  
TEXCOORD, min1uint2 boneno : BONENO, min1uint weight : WEIGHT)  
{  
    float wgt1 = float(weight) / 100.0;  
    float wgt2 = 1.0 - wgt1;  
    Output o;  
  
    matrix m = world;  
  
    m = mul(world, boneMat(boneno(0)) * wgt1 + boneMat(boneno(1)) * wgt2);  
    o.origpos = mul(m, pos);  
    pos = mul(mul(viewproj, m), pos);  
    o.normal = mul(m, normal);  
    pos.xyz += (o.normal * 0.05); //法線方向に引き延ばす  
    o.pos = o.svpos = pos;  
  
    return o;  
}
```

であり、

```
//輪郭線ピクセルシェーダ  
float4 OutlinePS(Output data) : SV_Target  
{  
    return float4(0,0,0,1); //黒く塗れ!(別に黒じゃなくてもいいよ)  
}
```

です。

これでうまい事行くかと思いきや結果は



確かに輪郭線らしきものは表現されているが…？



これはいいたい…

恐らく一番の実害は口の部分…おそらく小さな空間内で複雑な形をしているものは法線方向に引き延ばすと思いもよらない形になってしまうのではないかと思われます。

ではMMDはどのようにやっているのでしょうか…？

とりあえずフラグの場所を確認します。2か所ありました。まずは頂点情報

vertex[4019].edge_flag	00
vertex[4020].pos[0]	BF05119D 414F4F0E 3F0F2E49
vertex[4020].normal_vec[0]	BEFFA7A3 3EBA17CE 3F49577E
vertex[4020].uv[0]	00000000 3F800000
vertex[4020].bone_num[0]	0001 0009
vertex[4020].bone_weight	0A
vertex[4020].edge_flag	00

次にマテリアル情報…

material[4].diffuse_color[0]	3F1090BC 3F1090BC 3F1090BC
material[4].alpha	3F800000
material[4].specularity	40A00000
material[4].specular_color[0]	3E19999A 3E19999A 3E19999A
material[4].mirror_color[0]	3EB4BC6A 3EB4BC6A 3EB4BC6A
material[4].toon_index	00
material[4].edge_flag	01
material[4].face_vert_count	000000D32

どっちを使えばいいのだろうか…

ひとまず頂点側をいじってみましょう。頂点ごとにエッジフラグがついてるので、有効にするには頂点レイアウトにエッジフラグも追加します。

ちなみにちょっと面倒な事に

『BYTE edge_flag; // 0:通常、1:エッジ無効 // エッジ(輪郭)が有効の場合』
らしいです。1が無効らしいので…



なんでじゃ…普通1が輪郭線アリで、0がナシでしょ

ということで

`pos.xyz += (o.normal*0.1)*(1-edge); // 法線方向に引き延ばす`
てな感じのコードにしました。さつきより輪郭線を太くしてみてます。



うーん汚い

元の0.05にしてみても



うーん

何か見落としているのだろうか…。となんでなんでと悩んでたんですが、しょーーもないミスでした。

ここね

```
pos = mul(mul(viewproj, m), pos);
o.normal = mul(m, normal);
pos.xyz += (o.normal * 0.05); // 法線方向に引き延ばす
```

やってる事はレリインですが、順番がね…気づかない？

そう、プロジェクション変換をした後に法線方向に引き延ばしたので、おかしなことになるわけです。という事で、ちょっとまどろっこしいですが

```
o.origpos = mul(m, pos);
o.normal = mul(m, normal);
pos.xyz = o.origpos.xyz + (o.normal.xyz * 0.05) * (1 - edge); // 法線方向に引き延ばす
pos = mul(viewproj, pos);
```

という風に順序を変更してあげる必要があります。そうすれば



ふう…

輪郭線はできればピクセルで太さを指定したいとします。これを実現するには一体どうしましようか…？

まず当然の話ですが「輪郭線」は二次元のものです。つまり XY 方向にしか厚みは必要ありません。

次に法線方向は XYZ です。

さて…このような状況に対応するにはどのようにしたらよいのでしょうか？法線方向への引き延ばしが、2 次元上で 1 ピクセルになるようにすればいい…つまり例えは X 方向だけを考えると

0～WINDOW_WIDTH ⇔ -1～1

つまり 1 ピクセルというのは $2/\text{SCREEN_WIDTH}$ にあたるわけです。例えば適当に法線方向に引き延ばした結果、最終的に 5 pixel になるのならばこれを 5 で割ってあげる必要があるわけです。

つまり通常の頂点座標をプロジェクション変換かけた後の結果の座標を P_0 とし、引き延ばした後の座標を P_1 とすると

$P_1 - P_0$ を 1 ピクセル… $2/\text{SCREEN_WIDTH}$ になるようにするのだから

$$\text{float2 outlineRatio} = (P_1 - P_0) \div \frac{2}{\text{SCREEN_WIDTH}}$$

そしてこの outlineRatio で法線ベクトルを割ってやればいい。

ということで

```
float4 p0 = mul(viewproj, o.origpos);
float4 p1 = mul(viewproj, pos);

p0.xy /= p0.w;
p1.xy /= p1.w;

float outlineRatio = distance(p1.xy,p0.xy)*480.0f;
if (outlineRatio > 1.0f) outlineRatio = 1.0f / outlineRatio;
pos.xyz = o.origpos.xyz + (o.normal*outlineRatio);
こんな感じで。
if 文の部分は、こうしたかないと法線が真正面に来た時にぶつ飛ぶんでこうしています。
```

で、ここまで一生懸命やってきたんですが結局



次に別のエッジの出し方を考えてみましょう。

輪郭線抽出フィルター

ちょっとした画像処理の基礎知識として「ラプラシアンフィルタ」というのを考えてみましょう。

<https://algorithm.joho.info/image-processing/laplacian-filter/>

<https://qiita.com/shim0mura/items/5d3cbef873f2dd81d82c>

<http://asura.iraigiri.com/DirectX/dx10.html>

うーん。そういうわけでね？グラフィックスプログラムをやる場合はCGの知識だけではなく
画像処理の知識も必要になるのだ。そしてそれは今後ますます顕著になるのだ。
もう一つ輪郭線系統で覚えておいた方がいいのが「ソーベルフィルタ」ですね。

<http://www.mis.med.akita-u.ac.jp/~kata/image/sobelprev.html>

<http://asura.iagiri.com/DirectX/dx11.html>

どちらも「エッジを強調する」フィルタです。ただし結果はちょいちょい異なるようですので、その辺は興味があれば比較したり調べたりしてみましょう。

で、これららへんの説明を見ると微分だのなんだの言っておりますが、結局のところは周囲の画素値と比較して、変化が大きければ大きいほど値がでかくなるフィルタみたいに思ってればいいと思います。例えばラプラシアンフィルタなどは

0	1	0
1	-4	1
0	1	0

のようになっておりますが、これはどういうことがと言うと、それぞれ現在の画素の位置を中心として自分と周囲の画素値に対して乗算を行い、足してあげます。例えば画素値が横に10,10,10,40,40,40

のように並んでいたとします。今はちょっと縦は考えません。おおざっぱに仕組みをとらえてもらえばいいです。

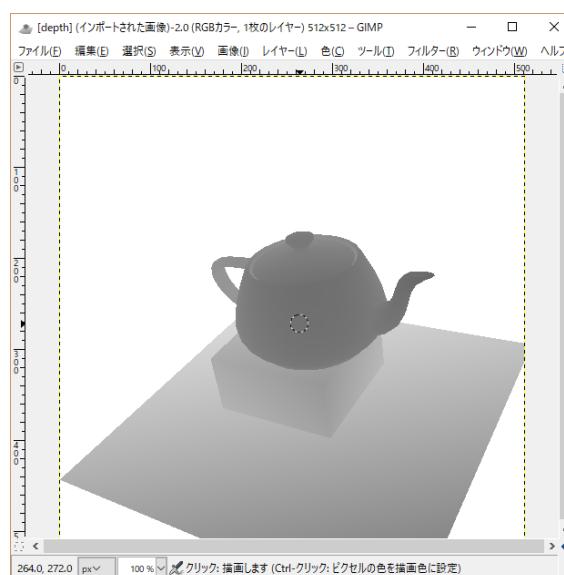
さて、10の周囲が10であればこのフィルタをかけた結果は0になりますね？ $10+10+10+10+(-4)*10=0$ です。

ところが隣が40である場合… $10+10+10+40-40=30$ となります。

言うても良く分からぬと思ひますので実験してもらいます。サーバの

<http://132sv.yakuseigamero.yrkawano.yDirectX12>

のフォルダの中に depth.png ってのがあると思ひます。こいつを GIMP で開いてみてください。



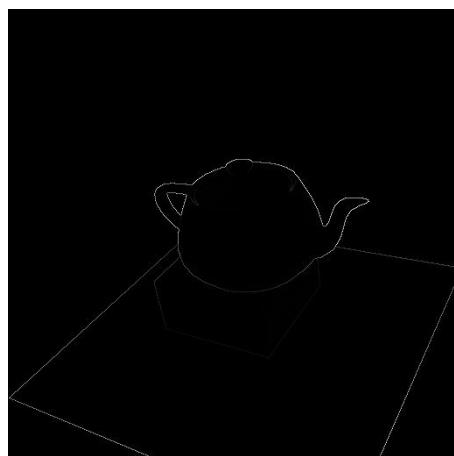
こうなりますね？これでフィルター→汎用→コンボリューションって押すと



こんな風な画面が出る

行列にラプラシアンフィルタ通りに入力してOKを押してみてください。

なお透明度のチェックは外しておいてください。そうすると…

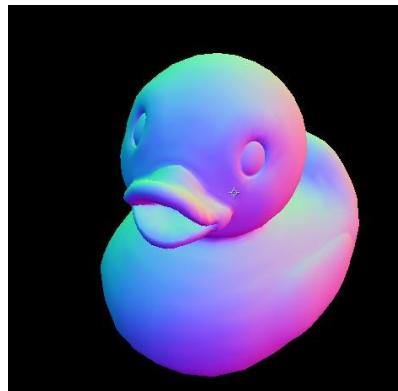


おっ!?

イイ感じに深度方向の輪郭線がとれそうです。

これで裾の輪郭線も表現できそうですね。

ちょっとついでに法線の輪郭線もとってみましょう。



こいつに対しても同様にラプラシアンすると…



ちょっと微妙かも…

…ひとまずは、深度バッファに対してラプラシアンフィルタをかけるところまで実装してみましょう。

深度パスレンダリング結果を応用

深度パスレンダリング結果が必要になってきます。以前にも話しましたが、最近のレンダリングは1パスで収まっている事は殆どないです。スマホゲームとかならともかく、コンシューマ系はマルチパスレンダリングの知識必須ですし、今後はスマホゲームでも必須になってくるかと思います。

<http://zerogram.info/?p=1070>に書いてるみたいにね。

やり方としてはレンダーターゲットを複数用意しておいて(シャドウマップの時にやつたね?)テクスチャとして使えるような状態にしておく。で、そこにテクスチャの内容を書き込んでいくときに合成を行ってわけさ。

だからそれぞれのパスの手順を書くと

- ①パス目:ライトからの深度値レンダリング
- ②パス目:カメラからの深度値をレンダリング
- ③パス目:通常のレンダリング+深度値を元にした輪郭線描画

と、まあ言うほどうまくいくとは思えませんが、やるだけやってみましょう。

既にライトビューは取れているので、ひとまずは②から実装してみましょう。ひとまず今までの背面法のコードが邪魔なので、一旦コメントアウトしましょう。

さて、言っておくがここから先は僕も手探りだ。うまくいかなくても馬鹿にしないように。

さて必要なものを教える…

- ①輪郭線抽出用深度バッファ
- ②輪郭線用パイプラインステートオブジェクト
- ③輪郭線抽出用シェーダ

これくらい…ですかねえ。

①の「深度バッファ」はライトビューの所でもやったと思いますが、上みたいに深度バッファとして&テクスチャバッファとして作りたかったら、ライトビューの時にやったように1つのバッファに2つの見方(デスクリプターヒープ)が必要になりますね。

②,③に関してはシェーダがチョイと違うくらいです。

さあ、やってみましょう。ここまで説明がそこそこ理解できているのならばやれると思います。

今回のキモは深度バッファのみなので、ライトビュー同様にレンダーターゲットビューに書き込む必要はありません。シェーダ側も特にひねる必要はないでしょう。

```
//輪郭線のための(深度バッファ法)頂点シェーダ
Output DOutlineVS(float4 pos : POSITION, float3 normal : NORMAL, float2 uv :
TEXCOORD, min16uint2 boneno : BONENO, min16uint weight : WEIGHT, min16uint
edge : EDGE)
{
    float wgt1 = float(weight) / 100.0;
```

```

float wgt2 = 1.0 - wgt1;
Output o;

matrix m = mul(world, boneMat(boneno(0)) * wgt1 + boneMat(boneno(1)) *
wgt2);
o.origpos = mul(m, pos);
o.normal = mul(m, normal);

pos = mul(viewproj, o.origpos);

o.pos = o.svpos = pos;
return o;
}

```

基本部分は BasicVS と同じなので、別関数化してそれから呼び出すのも良いでしょう。なお、BasicVS よりもシンプルにしています。

あとは

```

//輪郭線(深度/ツッファ法)用のシェーダ
ID3DBlob* doutlinevs = nullptr;
ID3DBlob* doutlineps = nullptr;
result = CompileVertexShaderFromFile(_T("shader.hlsl"), "DOutlineVS",
doutlinevs);
result = CompilePixelShaderFromFile(_T("shader.hlsl"), "DOutlinePS",
doutlineps);

```

こうして

```

gpsDesc.VS = CD3DX12_SHADER_BYTECODE(doutlinevs);
gpsDesc.PS = CD3DX12_SHADER_BYTECODE(doutlineps);
ID3D12PipelineState* _doutlineGPS = nullptr;
result = dev->CreateGraphicsPipelineState(&gpsDesc,
IID_PPV_ARGS(&_doutlineGPS));

```

こんな感じですね。なお、outline の先頭に d がついているのは深度(depth)のつもりです。

で、今回も必要なのはとりあえず深度だけなので、レンダリング自体は

```
_commandList->DrawIndexedInstanced(indexCount, 1, 0, 0, 0);
```

で良いと思います。

ちょいちょい実験しつつ、ラプラシアンフィルタを実行すると



こんな感じでエッジがとれています。これはいけそうですね。

ちなみに、ラプラシアンフィルタ(4近傍)の定義に合わせてシェーダ内にこういう関数を作りました。

```
//とりあえず8近傍全て用意したが4近傍で十分
//float d1 = doutline.Sample(smp, uv + float2(w * -1, h * 1)); //1左下
float d2 = doutline.Sample(smp, uv + float2(w * 0, h * 1)); //2:下
//float d3 = doutline.Sample(smp, uv + float2(w * 1, h * 1)); //3:右下
float d4 = doutline.Sample(smp, uv + float2(w * -1, h * 0)); //4:左
float d5 = doutline.Sample(smp, uv + float2(w * 0, h * 0)); //5:中
float d6 = doutline.Sample(smp, uv + float2(w * 1, h * 0)); //6:右
//float d7 = doutline.Sample(smp, uv + float2(w * -1, h * -1)); //7:左上
float d8 = doutline.Sample(smp, uv + float2(w * 0, h * -1)); //8:上
//float d9 = doutline.Sample(smp, uv + float2(w * 1, h * -1)); //9:右上

float v = d2 + d4 + d6 + d8 + (d5 * -4.0); //ラプラシアンフィルタ(4近傍計算)
v = v > 0.001f ? 1.0 : 0.0;
return (1 - v);
```

この計算でラプラシアンをやってくれるというわけです。あとは描画の際にちょいちょい工夫すればOK。

既に doutline の深度 / ディファに書き込みは行われているんであとはこの画像を持ってきて UV 値に適切なものを入れるだけですが、それほど難しくもなくて、例によって xy から取って

くればいいのです。

```
uv = data.pos.xy / data.pos.w; // wで割らないとまともな値になりません  
uv = (uv * float2(1, -1) + float2(1, 1)) * 0.5f; // (-1~1) → (0~1) 変換  
float v = GetLaplacianValue(uv); // ラプラシアンフィルタをかける  
if (v == 0) { // ラプラシアン値が0(輪郭線)ならば輪郭線を描画  
    return float4(0, 0, 0, 1); // 輪郭線です  
}
```

これを正しく間違なく適用してやると



ひとまず輪郭線、トーンについてはこんな感じで終了です。

そのうちMMDにはトーン拡張みたいなのがあるらしいので後々はそれを説明しようかと思います。

アンチエイリアシング

はじめに



このネタも昨年、一昨年はやってなかったですね。結構アンチエイリアシングは面倒だしややこしいし、輪郭線のアルゴリズムとも相性が悪いみたいなんですが…

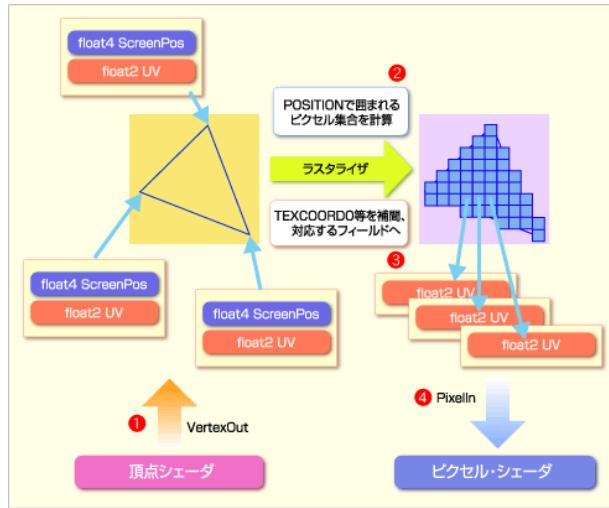


流石にエッジが汚いんですねえ…

そもそもなぜこういう事になるのかと言うと、ポリゴンをスキャンラインで描画していくわけなんですが、「あるピクセルを塗るか塗らないか」をラスタライザが決めてるのだ



どういう事がと言うと、ピクセル単位で塗るか塗らないかを決定するという事は当然



んだけど、そりやあガタガタになるよなって思います。

拡大



例えば、↑のような直線描画であれば、直線の方程式との親和率？網羅率？カバレッジ？から α を設定したりしますが…いちいちレンダリングのたびに網羅率なんて設定できません&上の作業をやるのがラスタライザであるためこちらで制御できる部分でもないんだよなあ…。

SSAA(SuperSamplingAnti-Aliasing)

クソ簡単ですねフォレハ。

スーパーサンプリングってのは何かちゅうと、倍くらいのハイレゾでレンダリングして、平均化すればアンチエリアシングと同様になります。

手順は

1. すつげー高解像度でレンダリング
2. 縮小(縮小時の画素値はニアレストネイバーではなくバイリニア等で計算)

で、これ簡単すぎなんです。

2パス目のサンプリングステートをバイリニアにしておけば、1パス目のレンダーターゲットを例えば縦横2倍(ピクセル数4倍)にしておけば、勝手に4ピクセル平均計算してくれるので特にテクニカルな事はないです。

ただ、注意点として、例えばレンダーターゲットの解像度を縦横2倍にしたならば、ScissorRectおよびViewportの大きさも縦横2倍にしなければ辻褄が合わなくなります。

このやり方は正攻法なだけに、品質は高いです。もちろんレンダーターゲットの解像度を上げれば上げるほどクオリティは上がるでしょう。

ただ、当然ながらピクセルシェーダが4倍の回数走る事になるため、処理負荷が高くなります。恐らくレンダリングクオリティ重視のエロゲーのAAなどはこれを使用しているのではないかと思われます(知らんけど~)。

```
_firstPassRTBuffer.reset(_buffMgr->CreateTextureBuffer(desc.Width*2, desc.Height*2));
```

こんな感じで2倍バッファ作っておいて……。

深度も

```
auto rtdesc=_firstPassRTBuffer->GetResource()->GetDesc();
D3D12_RESOURCE_DESC depBufDesc = {};
depBufDesc.Width = rtdesc.Width;
depBufDesc.Height = rtdesc.Height;
(中略)
auto result = _dev->CreateCommittedResource(
    &depthprop,
    D3D12_HEAP_FLAG_NONE,
    &depBufDesc,
    D3D12_RESOURCE_STATE_DEPTH_WRITE,
    &clearValue,
    IID_PPV_ARGS(_dsvBuffer.GetAddressOf()))
);
```

こんな感じで作って、忘れちゃいけないのがシザーリミット

```

auto vp = _viewport;
auto sr = _scissorRect;
auto desc = _firstPassRTBuffer->GetResource()->GetDesc();
vp.Width = desc.Width;
vp.Height = desc.Height;
sr.right = desc.Width;
sr.bottom = desc.Height;
_cmdList->RSSetViewports(1, &vp);
_cmdList->RSSetScissorRects(1, &sr);

```

このように最初に作ったレンダーターゲットの大きさが反映されるようにしておきます。モチロン、画面分割などある場合はこの限りではありません。

MSAA(MultiSamplingAnti-Aliasing)

これは、SDK の機能を使いましょう。

原理自体は SSAA とほぼ同様ですが、SDK で一応サポートされています。ただブラックボックスになってしまって、一つ使うと、調整的なところはできないでしょう。

ちなみに MSAA と SSAA の原理が同様な点とは高解像度と言う部分です。

ただし、高解像度なのは深度値部分のみにしておく。高解像度において深度値が急激に変化する部分はエッジであるため、深度値からカバレッジ(ポリゴンが占める割合)計算を行い、ラスタライズ時に重みづけ計算を行います。

これはラスタライザの協力が必要…つまりハードウェアサポートがされています。なので、どうやるかというと、今まで Sampler の Quality だの Count だのの指定がありましたよね？

で、今回のこれに関してはハードウェアのご協力が必須なので、CheckFeatureSupport を用いてその機能が使用できるかどうかを問い合わせます。

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

```

D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS qualitylevels = {};
qualitylevels.SampleCount = 8; // チェックしたいサンプルカウント
qualitylevels.Format = DXGI_FORMAT_R8G8B8A8_UNORM;

```

```

qualitylevels.Flags = D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;
result = _dev->CheckFeatureSupport(D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS, &qualitylevels,
sizeof(qualitylevels));
if (result == S_OK&&qualitylevels.NumQualityLevels > 0) {
    _msaaparam.samplecount = qualitylevels.SampleCount;
    _msaaparam.quality = qualitylevels.NumQualityLevels-1;
}
else {
    _msaaparam.samplecount = 1;
    _msaaparam.quality = 0;
}

```

上の例では 8 が通らなかつたら MSAA なしの即落ち状態になつてますが、一つずつおりたいな
ら

```

D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS qualitylevels{};

qualitylevels.SampleCount = 8;
qualitylevels.Format = rtv.Format;
qualitylevels.Flags = D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;

while (msaaQualityDesc.SampleCount > 1) {
    HRESULT hr = m_device->CheckFeatureSupport(D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS,
                                                &qualitylevels, 1);
    if (SUCCEEDED(hr)) {
        break;
    }
    msaaQualityDesc.SampleCount--;
}

```

などと書くことになると思います。で、ここで得られたサンプル数をサンプラーステートの
SampleCount や Quality に合わせていきます。

ところがそれで使用できるかと言うとそうではないです。

ResolveSubresource を使用して、マルチサンプル ⇒ 非マルチサンプル変換を行わねばなりません。

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

面倒くさくて、頭にきますよ。

ハードウェアサポートされているのはありがたいのですが、仕様に至るまでの段階が面倒。そして致命的な部分としては、深度値でエッジ検出しているため、テクスチャそのものがジヤギってたり、深度が変化なしの時に効果が出づらいという事があります。

FXAA?(Fast Approximate Anti-Aliasing)

<https://www.4gamer.net/games/120/G012093/20121125002/>

そういうものもあるらしい。説明を読む限り「きわめてシンプル」らしいのでやってみようかと思う。

「ピクセル色を周囲と比較して輝度差を調べ、輝度差があるピクセルの色は周囲と混ぜ合わせる」

なんじゃこりや。言わんとする意味はわからんでもないんだが…こういう事？

1. 周囲の輝度値を調べる
2. 周囲の輝度値との微分値を得る
3. 変化が激しければ周辺をぼかす

こういうこと？

```
//輝度値とかいうとるのでグレースケール変換して、輪郭線抽出
float3 gray = float3(0.298912f, 0.586611f, 0.114478f);
float b = dot(gray, ret.rgb);
b *= 8;

b -= dot(gray, tex.Sample(smp, input.uv + float2(-dx, -dy)).rgb);
b -= dot(gray, tex.Sample(smp, input.uv + float2(dx, -dy)).rgb);
b -= dot(gray, tex.Sample(smp, input.uv + float2(-dx, dy)).rgb);
b -= dot(gray, tex.Sample(smp, input.uv + float2(dy, dy)).rgb);

b -= dot(gray, tex.Sample(smp, input.uv+float2(-dx,0)).rgb);
b -= dot(gray, tex.Sample(smp, input.uv + float2(0, -dy)).rgb);
b -= dot(gray, tex.Sample(smp, input.uv + float2(dx, 0)).rgb);
b -= dot(gray, tex.Sample(smp, input.uv + float2(0, dy)).rgb);
```

こうやっておいて…

```
//輪郭線抽出フィルタ作動
//周囲との差がそれなりにあったら(輪郭線がそれなりにくっきりしてたらぼかす)
if (abs(b) > 0.8){
    return float4(
        //自分と周囲8近傍平均
        ( tex.Sample(smp, input.uv + float2(-dx, - dy)).rgb
        + tex.Sample(smp, input.uv + float2(0, - dy)).rgb
        + tex.Sample(smp, input.uv + float2(dx, - dy)).rgb
        + tex.Sample(smp, input.uv + float2(-dx, 0)).rgb
        + ret.rgb
        + tex.Sample(smp, input.uv + float2(dx, 0)).rgb
        + tex.Sample(smp, input.uv + float2(-dx, dy)).rgb
        + tex.Sample(smp, input.uv + float2(0, dy)).rgb
        + tex.Sample(smp, input.uv + float2(dx, dy)).rgb)/9,ret.a);
}
else {
    return ret;
}
```

とかやってみた。



うわ、遅いのが分かりづれえ〜。でも一番右、ボケてる感があるので、
やっぱりあのやり方じゃあエッジがぼけちゃうんだな。
一応 FXAA はエッジがボケるとは書いてあるが、なんかちょっと違う気がする…

もっと良く読んでみよう。

がチめ FXAA

FXAA
サブピクセルを使わないAA

1. 通常のレンダリングを行う(サブピクセルは不要)
2. 周辺ピクセルとの輝度差を計算する
3. エッジの向きと長さの検出
4. エッジにあわせて周辺のピクセルと色をブレンド



4Gamer.net
<http://www.4gamer.net/>

こつ、これわ…

ん?どうやって勾配なんて出すんだよ…。無茶言うなよ…でも勾配とか出せたら色々と役に立つかも…。

よくわからんが

<https://github.com/NVIDIAAGameWorks/D3DSamples/blob/master/samples/FXAA/media/FXAA.hls>
!

の内容を見てみる。

ちなみに FXAA のページは

https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf
ですけどね。

すごいコードだ…内積、行列…asm…アセンブラ?

一応、<http://hikita12312.hatenablog.com/entry/2017/12/16/160429> を見る限り、このコードは「使うもの(利用するもの)」扱いっぽいが、それも謎まらん。

まあ、ともかく最初は「使ってみて」それから考えましょう。

ひとまず FXAA.hlsl を落として来ましょう。そして#include を pera.hlsl シェーダ側に書いておきましょう。

で、そのままだとシェーダコンパイル失敗しますので、シェーダコンパイルの引数を
D3DCompileFromFile(L"pera.hlsl", nullptr, D3D_COMPILE_STANDARD_FILE_INCLUDE, "vs",
"vs_5_0", D3DCOMPILE_DEBUG | D3DCOMPILE_SKIP_OPTIMIZATION, 0, &peravs, &error);

とします。第三引数ですね。今まで nullptr だったと思います。
これで普通に実行されるのを確認したら…使ってみましょう。

#include 文の前に

```
#define FXAA_PC 1
#define FXAA_HLSL_5 1
#include"FXAA.hlsl"
```

ちょっと直値で嫌ですか…手取り早く関数を呼んで値を返してみましょうか。

```
FxaaTex InputFXAAATex = { smp, tex };
return FxaaPixelShader(
    input.uv,                                // FxaaFloat2 pos,
    FxaaFloat4(0.0f, 0.0f, input.size.x, input.size.y), // FxaaFloat4 fxaacConsolePosPos,
    InputFXAAATex,                            // FxaaTex tex,
    InputFXAAATex,                            // FxaaTex fxaacConsole360TexExpBiasNegOne,
    InputFXAAATex,                            // FxaaTex fxaacConsole360TexExpBiasNegTwo,
    float2(1.0,1.0)/input.size,                // FxaaFloat2 fxaaqualityRcpFrame,
    FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f),       // FxaaFloat4 fxaacConsoleRcpFrameOpt,
    FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f),       // FxaaFloat4 fxaacConsoleRcpFrameOpt2,
    FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f),       // FxaaFloat4 xaaConsole360RcpFrameOpt2
    0.75f,                                    // FxaaFloat fxaaqualitySubpix,
    0.16bf,                                    // FxaaFloat fxaaqualityEdgeThreshold,
    0.0833f,                                   // FxaaFloat fxaaqualityEdgeThresholdMin,
    8.0f,                                      // FxaaFloat fxaacConsoleEdgeSharpness,
    0.125f,                                     // FxaaFloat fxaacConsoleEdgeThreshold,
    0.05f,                                     // FxaaFloat fxaacConsoleEdgeThresholdMin,
    FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f)        // FxaaFloat fxaacConsole360ConstDir,
);
```

結果…



エッジ検出ができるのは間違いなさそうなんだけど…

寧ろエッジ強調になってるような…なんだこれ。パラメータ間違ってんのか入力テクスチャデータがアレなのが…。エッジが黒くなってるんだよなあ…

ちょっとコード見てみたけど、ちょっとなんか α の値を違う用途に使ってるっぽいなあ…ということは…戻り値の α 値を使わないようにしたらどうだろうか。つまり

```
float4 ret = tex.Sample(smp, input.uv);
FxaaTex InputFXAATex = { smp, tex };
return float4(FxaaPixelShader(
    input.uv, // FxaaFloat2 pos,
    FxaaFloat4(0.0f, 0.0f, 0, 0), // FxaaFloat4 fxaacConsolePosPos,
    InputFXAATex, // FxaaTex tex,
    InputFXAATex // FxaaTex fxaacConsole360TexExpBiasNegOne,
```

```

InputFXAATex,           // FxaaTex fxaacConsole360TexExpBiasNegTwo,
float2(1.0,1.0)/input.size,      // FxaaFloat2 fxaaqQualityRcpFrame,
FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f), // FxaaFloat4 fxaacConsoleRcpFrameOpt,
FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f), // FxaaFloat4 fxaacConsoleRcpFrameOpt2,
FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f), // 
0.75f, //SUBPIX,           // FxaaFloat fxaaqQualitySubpix,
0.166f, //EDGE_THRESHOLD,    // FxaaFloat fxaaqQualityEdgeThreshold,
0.0833f, //EDGE_THRESHOLD_MIN, // FxaaFloat fxaaqQualityEdgeThresholdMin,
8.0f,           // FxaaFloat fxaacConsoleEdgeSharpness,
0.125f,          // FxaaFloat fxaacConsoleEdgeThreshold,
0.05f,           // FxaaFloat fxaacConsoleEdgeThresholdMin,
FxaaFloat4(0.0f, 0.0f, 0.0f, 0.0f) // FxaaFloat fxaacConsole360ConstDir,
).rgb,ret.a);

```

こうやってみると…



ああ～一応 FXAA やな～

まあちょっと中を見てみたいと α の扱いを確定できないんですが、「使う」ぶんにはこれでえんちゃうかなと思います。

ちなみに「輝度値」ということで、ちょっと細かく FXAA.hlsl をいじってみた。

723 行目あたりに

```
#if (FXAA_GRAY_AS_LUMA == 1)
    FxaaFloat Fxaaluma(FxaaFloat4 rgba) { return dot(float3(0.298912f, 0.586611f, 0.114478f),

```

```

    rgba.rgb); }

#ifndef FXAA_GREEN_AS_LUMA
    FxaaFloat Fxaaluma(FxaaFloat4 rgba) { return rgba.w; }
#endif
#ifndef FXAA_GRAY_AS_LUMA
    FxaaFloat Fxaaluma(FxaaFloat4 rgba) { return rgba.y; }
#endif

```

こんな感じに書いておいて…

```

#define FXAA_GRAY_AS_LUMA 1
#define FXAA_PC 1
#define FXAA_HLSL_5 1
#include "FXAA.hsl"

```

とします。

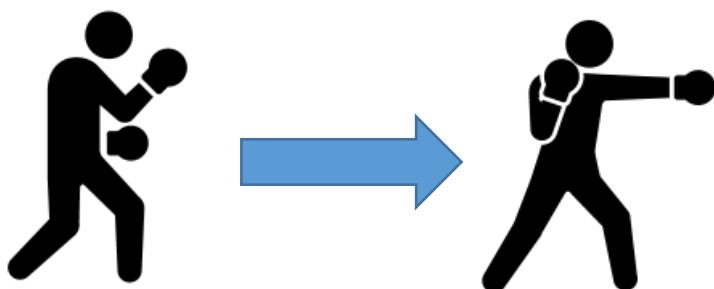
インバースキネマティクス(IK)

はつきり言ってこの章は超難度です。分からなくなったら次の章へ移動しても構いません。それくらい難しい。おそらくはこの本の中で最も難しいのではないか…。

インバースキネマティクス(IK)とは

インバースキネマティクスとは英語で書くと Inverse Kinematics で日本語で言うと「逆運動学」といいます。言葉だけ並べられてもわかりませんよね？

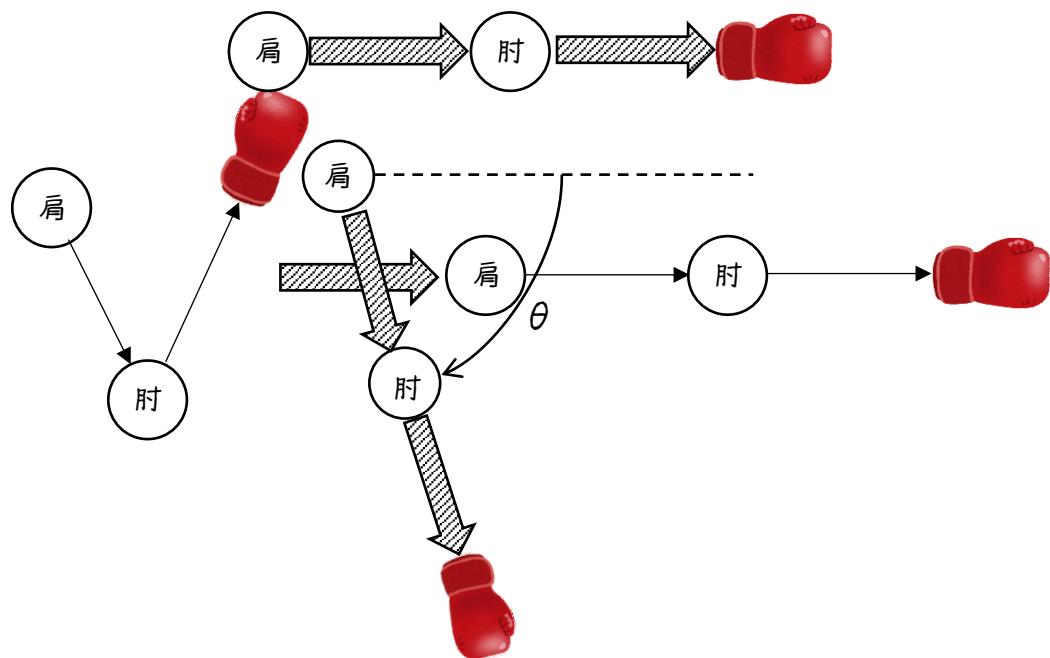
例えば以下のようにパンチを出す時の肩→肘→手首(拳)の位置関係について考えてみます。



拳を出しているときは肘を伸ばしていく、拳を顎に引き寄せていく(基本の構え)時は肘を畳んでいます。これを模式図で書くと
このようになるのは分かると思います。ではここで肩→肘→拳の位置関係を見てみましょう。
タコやイカなどの関節がない動物でない限り肩→肘→拳の間の長さは一定長ですね?

前にも言いましたがこういう制約(骨の長さは変えられない)があるために基本的に回転しかできません。

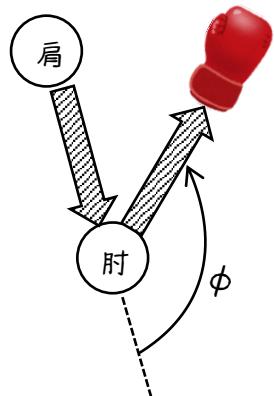
このためもし伸ばした状態から畳んだ状態にするためには



まず肩を中心に時計回りに θ 回転し…

肘を中心に ϕ 回転します。

といった具合に伸ばした状態のポーズの回転と畳んだ状態のポーズの回転をコントロールすることによって、腕をパンチを出したり引いたりします…が、アニメーションを作る側としてはこれではしんどいのです。まあ、腕ならまだしも足の場合などは、例えば階段のように段

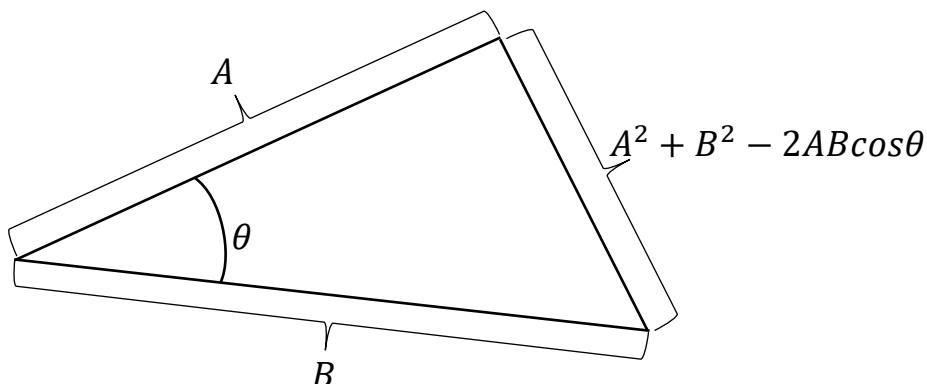


差があるところにいるとして、その回転を考えるのは大変です。

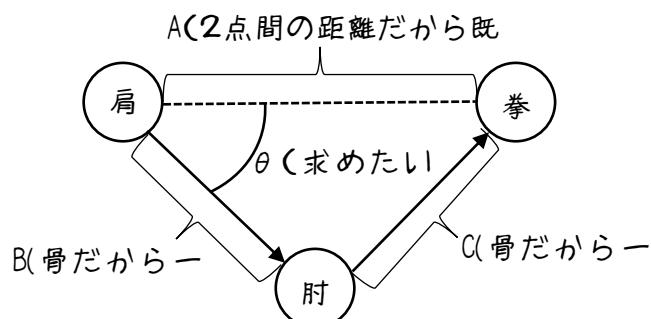
そこで基点ボーンと端点ボーンの位置関係から、その間のボーンの位置や回転を計算する。これがIK(インバースキネマティクス)です。先ほどの腕の例であれば肩→肘→拳の3点しかないので肘の位置は簡単に求められます。

高校数学(余弦定理)を用いた簡易的なインバースキネマティクス

「余弦定理」というのを覚えておりませんでしょうか？三角形のある点を挟む辺の長さと、その間の角度がわかればその点の向かいの長さが分かるというものです。



さて、ここで肩→肘→拳について考えてみますが、原則的に関節間の長さは変わりません。そのうえで肩と拳の位置が分かっている(肩と拳の直線距離が分かっている)とすると以下の図のようになります。



このように辺A、辺B、辺Cの長さは分かっています。ここからθを求めるには

$$C^2 = A^2 + B^2 - 2AB\cos \theta$$

を变形して…

$$\cos \theta = \frac{A^2 + B^2 - C^2}{2AB}$$

C言語にはacosというcosθの値からθを求める関数があります。数学的にはarccosといいますので

$$\theta = \arccos \left(\frac{A^2 + B^2 - C^2}{2AB} \right)$$

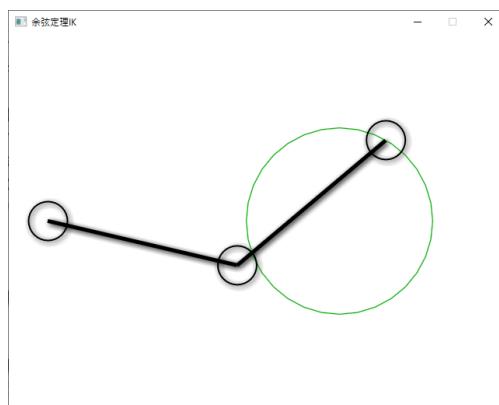
となります。簡単なのでサンプルプログラムを作ってみましょう。ここでは 2D 実装ですがまずは理解するための実験という事で考えてください。

```
Vector2f linearVec = positions[2] - positions[0];
float A = linearVec.Length(); // 点0から点2までの長さ(例えば肩から拳までの距離)
float B = edgeLens[0]; // 辺0の長さ(例えば上腕の骨)
float C = edgeLens[1]; // 辺1の長さ(例えば前腕の骨)
linearVec = linearVec.Normalized();
float angleFromX = atan2(linearVec.y, linearVec.x);
float theta = acosf((A*A + B*B - C*C) / (2 * A*B));
angleFromX += theta;
if (A < B + C) {
    positions[1] = positions[0] +
        Vector2f(cosf(angleFromX), sinf(angleFromX)) * edgeLens[0];
} else {
    positions[1] = positions[0] + linearVec * edgeLens[0];
}
```

プログラム全体は

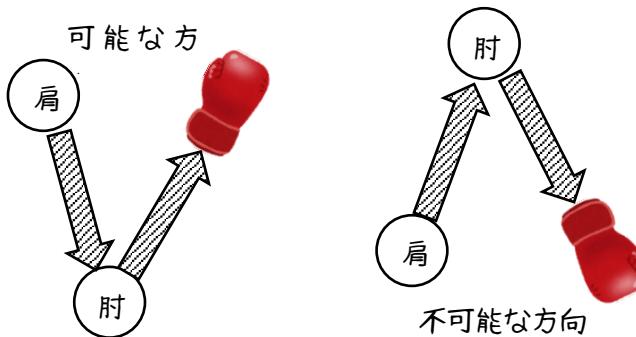
<https://github.com/boxerprogrammer/Mathematics/tree/master/CosineFomulaIK>

に置いてありますが、理屈自体は↑のコードで十分だと思います。数式をそのままプログラムにした形です。動かしてみれば分かりますが、点 0 と点 2 を動かせば真ん中にあたる点 1 が自動で動くのが分かると思います。



2D で、かつ間が 1 点しかないのであれば、難しいことをせずこの実装で十分だと思いますが、この実装でも一つの問題が残っています。それは「どちら側に曲げるか問題」です。

コンピュータプログラムは人間の関節の特性など知りません。つまり



こういった区別はつきません

今回の2Dの実験プログラムにおいては肩→拳よりも、肩→肘が時計回り方向に曲がるよう

にプログラムしていますが、実際MMDなどではどうやっているのでしょうか？

MMDのIKデータというデータの中に「角度制限」という要素があり、これによって曲がる方向

に制約をかけているようです。とはいってもそれは実際のMMD実装の話なので、それはあとでやり

ましょう。そしてまだ考えなければいけない問題があります。

2Dの例では回転角度を求めましたがその回転軸は当然Z方向です。

3Dはどうでしょうか？3Dにおいては回転の基本となる軸を考えなければいけません。

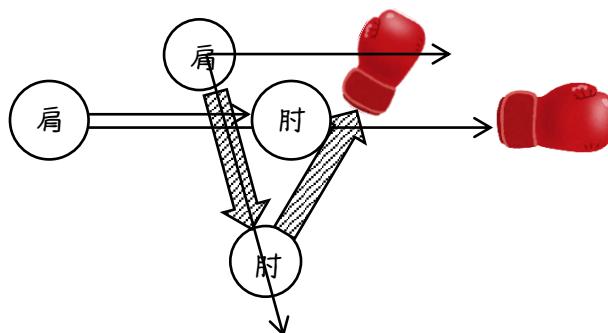
そしてその時の回転軸はX軸やY軸やZ軸などの分かりやすいものではなく「自然に関節が

曲がるためにの軸」である必要があります。つまり任意軸で回転させる必要がありますね？

角度の制限については置いといてとりあえずは「軸」について考えてみましょう。軸を作るには

最低限2つのベクトルを基準にする必要があります。それが分かれば外積を計算して座標

系を作るのですが、何と何を基準にしましょうか…。



まず考えられるのは上図のように最初のボーンまでのベクトルと、起点→端点のベクトルから

軸を作る方法です。これには一つ弱点があって、肩→肘→拳が一直線の場合は二つのベクト

ルの方向が一致してしまい、外積がまともなベクトルにならないのです。

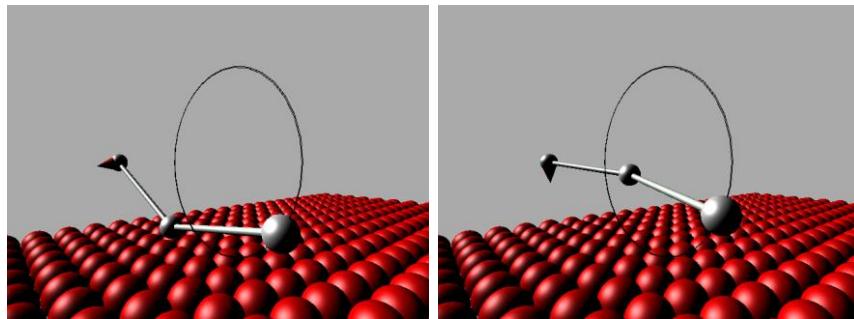
$$(a, b, c) \times (na, nb, nc) = (nbc - nbc, nca - nca, nab - nab) = (0, 0, 0)$$

なのでこの時に軸を決定しようとしてはいけません。しかし、困ったことに曲げ始めのときは

一直線になっているもので、そもそも「最初から曲がってる」場合のほうが少ないので。

そこで固定で特定の軸を使用することとしました。ルールとしてはボーンがその軸に直交するように回転するようにします。で、特定の軸は $(1,0,0)$ や $(0,1,0)$ や $(0,0,1)$ でいいんですが、これがボーンと同じ方向を向いてしまっては本末転倒なので、向きづらいベクトルを使用します。例えばキャラから見て真右などです。キャラが回転していなければ $(1,0,0)$ ですね。もし右ベクトルと外積した結果ゼロベクトルになるようなら $(0,0,1)$ などを使用します。ちなみに直交させるために2回外積を行います。

この結果として、始点を固定して端点が同じ動きをしたとしても、軸の方向によって間の点の動きが変わってしまいます。つまり軸の方向が重要であることが分かります。



図のように、軸の方向が変化すると折れ曲がる方向が変わります。これは余弦定理 IK だけではなく IK を作る時全てにおいて考慮しなければならない現象です。

なお、後述しますが MMD ではボーンの名前で軸の方向を決めていたりするため、なかなか特殊なことをする必要があります。

CCD-IKのしくみ

さて間の点が1つを超えると余弦定理によって角度を決定することができません。それを解決するための様々なIKの手法があります。

- CCD-IK
- パーティクル IK
- クオータニオン IK
- ヤコビアン IK

などがありますが、今回は比較的容易に実装できるCCD-IKについて考えてみます。

CCD-IKはCyclic Coordinate Descent IKの略で、日本語で言うと「巡回座標下降IK」みたいな名前になるのでしょうか、普通はCCD-IKと呼ばれています。

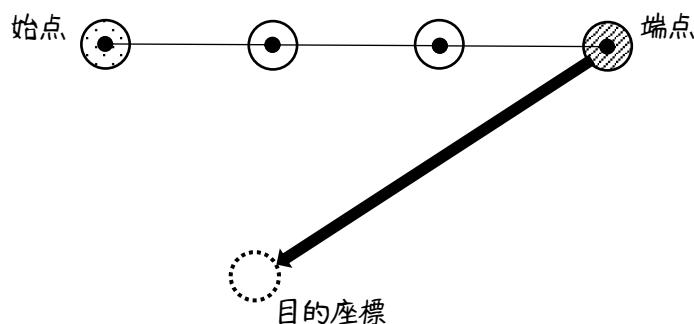
ただ、略語や英語のままではアルゴリズムを推測できないと思いますので、この日本語の意味からなんとなくイメージしてほしいのですが、順繰りに何回も繰り返します。またDescentというものは下降と書いてますが「逆順」と考えてください。

一文で説明すると

「コントロールポイント(端点、ボーン)を特定の場所に移動させるために、掴んだコントロールポイントからルート方向(始点、ボーン)へ遡るように回転処理(最終地点に近づくように)を行い、それを繰り返し近似座標を得る」

って事です。

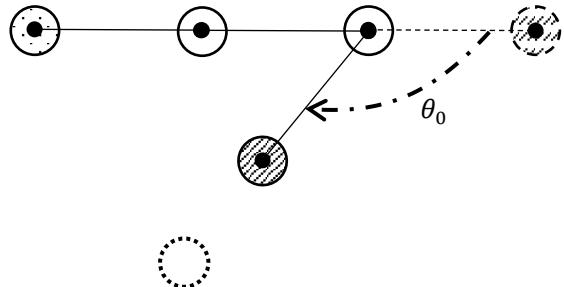
言葉だけではわかりづらいと思いますので、図で説明します。



まず、図のように端点を目的座標へ移動させたいとします。そのまま動かしてしまうと骨が伸びてしまうので、端点の1つ手前の点を中心とした回転だけを用いて可能な限り近づけます。

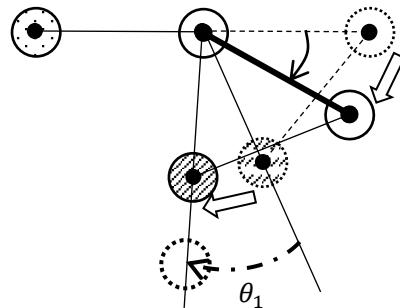
しかし、「端点の1つ手前の点を中心とした回転」だけでは目的の座標まで遠いとします。骨は

伸ばせませんから下図の状態が精一杯です。



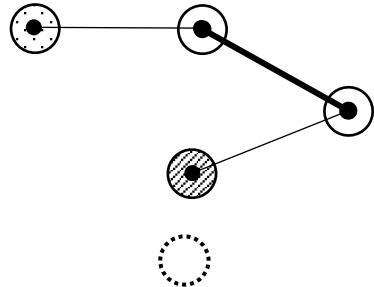
これでは意味がありませんね。しかし「この回転」ではここが限界です。そこでひとつボーンを遡ります(<この部分が逆順(Descent)ってわけです)。

一つ遡って、これも端点が限りなく目的地に近づくように回転します。



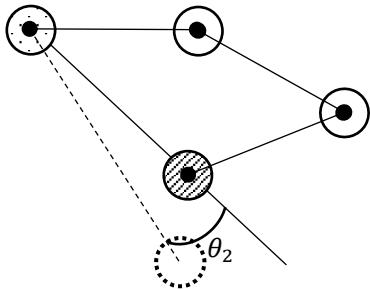
ちなみに回転量に関してですが、上図を見てもらえばわかりますが「今、回転の中心としてる点」と端点とを結ぶ線が、回転中心点と目的座標を結ぶ点に重なるように回転させます。

さて、↑の図だとゴチャゴチャしてしまってますが、末端から遡るように2度回転した結果

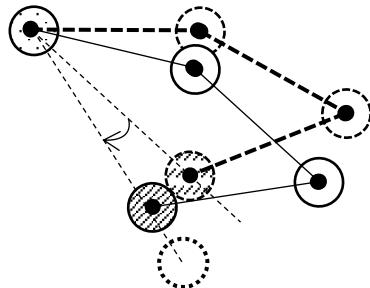


こうなりました。確かに近づいていますがまだ遠いですね。ここまで来たらルートを中心に回転するしかありませんね。

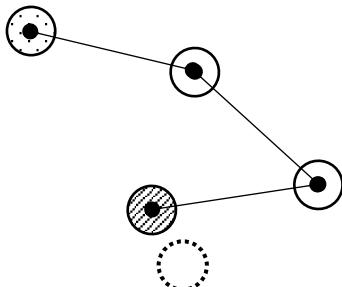
ではルート点から目的点まで線を引いてみて、どれくらいの角度が必要かを考えましょう。



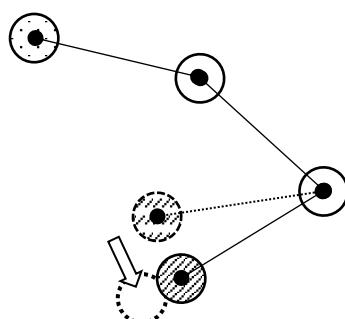
では、回転させてみましょう。



さて、こういう状況になりました。



まま、まだちょっとだけ離れてますが、1巡しちゃいましたね？どうしましようか？なんだか末端をもう一度回転させればかなりいい感じになりそうですね？そう、CCD-IKは試行回数の許す限りグルグルグルグル回転させて目的点に近づくような手法なのです。ついでに2巡目も一回だけ見てみましょうか。



上図のようにかなり近づきましたし、2巡目のどこかで許容範囲内に收まりそうなのが分かること思います。

ただし、この手法は言ってしまえば「余弦定理」のように答えが決まってる訳ではなく試行しながら近いところを探していくと言う訳で、近似のようですね。

実装においてはループ回数制限(試行回数制限)や、一回当たりの回転角度制限などを設定す

ることになると思います。そしてこれも当然の話ですが『どちら側に曲げるか問題』が各関節ごとに発生する事になります。

余弦定理 IK よりもかなり実装のハードルが上がりますので、ヤバいと思ったら次章に移ってくださいね。

20における CCD-IK 実装について

余弦定理 IK の時と同様にまずは 2D の CCD-IK の実装(4 点)について考えてみましょう。

特に制限を設けずに CCD-IK を実装してみると以下のようなプログラムになります。

角度制限なし

```
//繰り返し回数だけ繰り返す
for (int c = 0; c < cyclic; ++c) {
    //末端から攻めていく
    auto rit = positions.rbegin();
    ++rit;//末端との角度で計算するため、末端は省く
    for (; rit != positions.rend(); ++rit) {
        //回転角度計算
        Vector2f vecToEnd = positions.back().pos - rit->pos;//末端と現在のノードのベクトル
        Vector2f vecToTarget = target - rit->pos;//現在のノードからターゲットへのベクトル
        auto tarLen = vecToTarget.Length();
        auto endLen = vecToEnd.Length();
        if (tarLen == 0.0f) continue;//現在ノードからターゲットまでの距離
        if (endLen == 0.0f) continue;//現在ノードから末端ノードまでの距離
        vecToEnd.Normalize();
        vecToTarget.Normalize();
        //2D だから角度計算は単純(3D の場合こうはいかないです)
        float angle = atan2f(Cross(vecToEnd, vecToTarget), Dot(vecToEnd, vecToTarget));
        if (angle == 0.0f) continue;
        //回転行列を得る
        Matrix mat = MultipleMat(
            TranslateMat(rit->pos.x, rit->pos.y),
            MultipleMat(RotateMat(angle)),
            TranslateMat(-rit->pos.x, -rit->pos.y));
        //現在のノードから末端まで回転する
        auto it = rit.base();
        for (; it != positions.end(); ++it) {
```

```

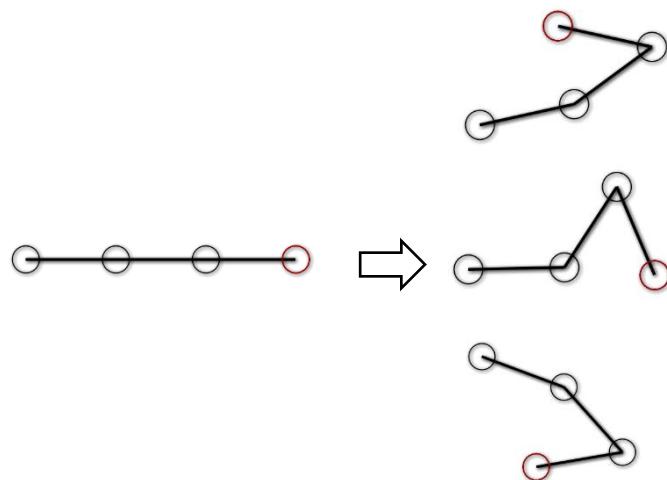
    it->pos = MultipleVec(mat, it->pos);
}

}

//末端の座標がターゲットと「ほぼ」一致したら計算を打ち切る
if ((positions.back().pos - target).Length() < epsilon) {
    break;
}
}

```

大して難しくないですよね。結果は下図のような形になります。



右端の丸を動かします。ほかの点はそこに合うように動いています。制限角度がなければほぼ正しく動作しています。

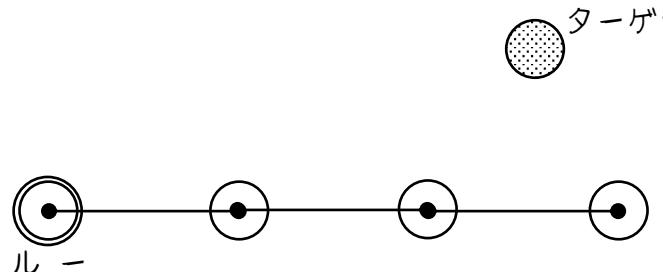
しかし、実際の人体に適用する場合には、関節は曲がる方向が決まっており、そのためには角度の制限が必要です。

角度制限あり

角度制限がつくとちょっと話がややこしくなるし、コード量も増えます。まず分かりやすい例で実装を考えてみましょう。

- 「根っこ」は角度制限なし
- 「根っこ以外」は時計回り方向に回るが、反時計回り方向に回らない

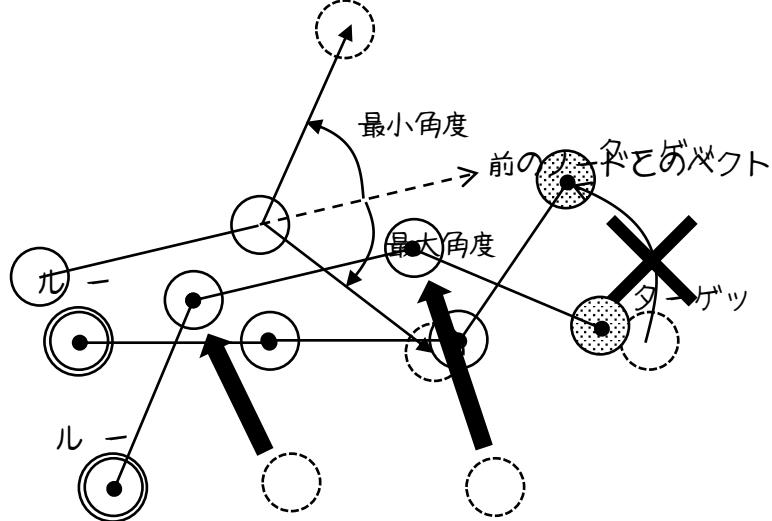
という条件としてみましょう。その条件の下でターゲットが以下のような状況になっている



とします。

もし角度制限がなければ末端手前を回転させてターゲットに合うようにするのですが、今の条件では反時計回りが制限されてるため回転することができません。そこでこの回転はあきらめて代わりに一つ手前のノードに戻りますがこいつにも回転制限がかかっています。結局ルートノードを回転させてから辻褄が合うように CCD-IK を行うしかありません。

このように動けばいいのですが、そううまくいくのでしょうか…？ 実際のプログラムについて



て考える前に「制限角度」はどこどこの間の角度を指しているのかを考えてみましょう。膝や肘の動きを考えるとこの図のように「前のノードとのベクトル」からの角度を制限することになるかと思います。ということで制限なしの時には考えなかつた「前のノード」について考える必要が出てきます。

ちなみに「前のノード」が存在しないルートノードの制限をすることはできません。角度比較対象がないからです。

実装の際に気を付けるべきは現在の座標からの回転差分を制限するのではなく、回転差分によって変化した現在のベクトルと前のベクトル間の角度を制限する事になりますのでお気を付けください。

この事を元にコーディングすると制限部分は以下のようなコードになります。

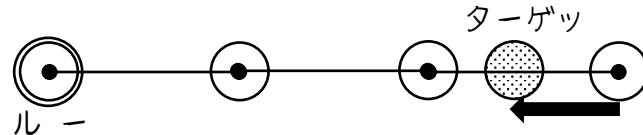
```
if (vecPrevious.Length() > 0) {  
    //今と前のベクトル間の角度を測っておく
```

```

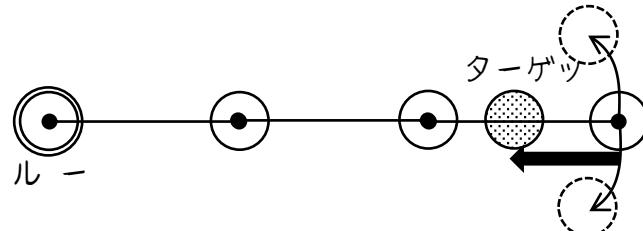
//現在のやつと次のやつの間のベクトルを算出
auto currentVec=rit.base()->pos - rit->pos;//現在のベクトル
//前のベクトルと現在のベクトルから角度を計算
auto anglePrev = atan2(Cross(vecPrevious, currentVec), Dot(vecPrevious, currentVec));
//現在の角度に差分を加算
auto angleForLimit = angle + anglePrev;
//結果の角度に対して制限クランプする
auto tmpAngle = Clampf(angleForLimit, rit->limit.minimum, rit->limit.maximum);
//もし制限されてたらその分戻す
angle += tmpAngle - angleForLimit;
}

```

実際の制限を行っているのは最後の 2 行(コメント行省く)ですね。これで角度制限つき CCD-IK が動作します。ただし、このままですると実際には「きれいにまっすぐ」ターゲットを根っこ方向に移動させたときに「角度に変化なし」と判断されて動かなくなることがあります。



それを防止するために上図のようなときは無理やり時計回りか反時計回りに強制的にちょっとだけ動かします。ただし、この時には制限角度に引っ掛からないように動かしましょう。



実際にプログラムを書くと

```

//ベクトルが完全に同一(もしくは反対向き)だった場合はほんのちょっと曲げる
//ただしリミットに引っかからないよう
if (fabs(dot-1.0f) < epsilon){
    //ターゲットまでの角度が同一でかつターゲットがノードの間にいるならば
    if (tarLen<endLen) {
        if (fabsf(rit->limit.minimum) < fabsf(rit->limit.maximum)) {
            angle = 0.01f;//ちょい回転
        }else {
            angle = -0.01f;//ちょい回転
        }
    }
}

```

```
    }  
}  
} といった具合になります。
```

ここまで 2D における CCD-IK のサンプルコードは

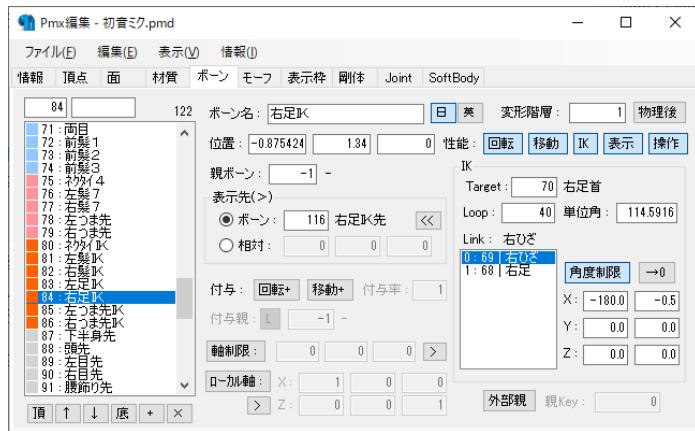
<https://github.com/boxerprogrammer/Mathmatics/tree/master/CCD-IK/CCD-IK>

に置いていますので、興味がある人は動かしてみてください。

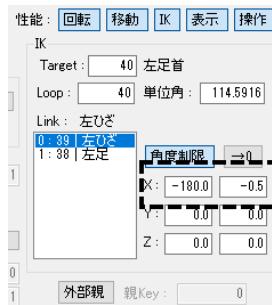
PMOにおけるCCD-IKについて

このままの流れだと制限角度付きで CCD-IK の 3D 版を用意しようというところなのですが、PMD の CCD-IK の場合、制限角度がかなり特殊なつくりをしているため頑張って 3D に移行してもちょっともったいないので、実装を先にしていこうかと思います。

PMXEditor というツールがあり、これが PMD や PMX のセットアップをするツールなのですぐ



今回は別にこのツールを使っていくわけではありません。なぜお見せしたのかというと、このツールの右下を見てください。



いっけん制限角度が設定されているように見えるのですが、実は PMD ファイルには制限角度は出力されていません。おそらく制限角度は PMX から設定できるようになったのではないかと思います。ちなみに PMX ファイルにおいては制限角度はきちんと出力されています。

じゃあ PMD はどうやって制御しているのかというとボーン名に「ひざ」が含まれている場合は「仕様として強制的に上記のような設定」にしています。

ちょっと納得いかないのですが、ひとまず相手が PMD の場合はそういう仕様になっていますので「ひざ」が含まれてるかどうかで制限角度を決めます。直値を入れているような気分で、気持ち悪い気がしますが「そういう仕様なのでしかたない」と割り切ってください。

ちなみにデータ内で「コントロールウェイト」というものがあるのですが、これはエディタ上の「単位角」に対応しており、一度に動かせる角度という事になっています。

まずは軸を決めましょう

さて、軸を決める場合も「ひざ」かそうでないかで結果が変わりますが、まずは「ひざ」でない場合の IKについて考えましょう。

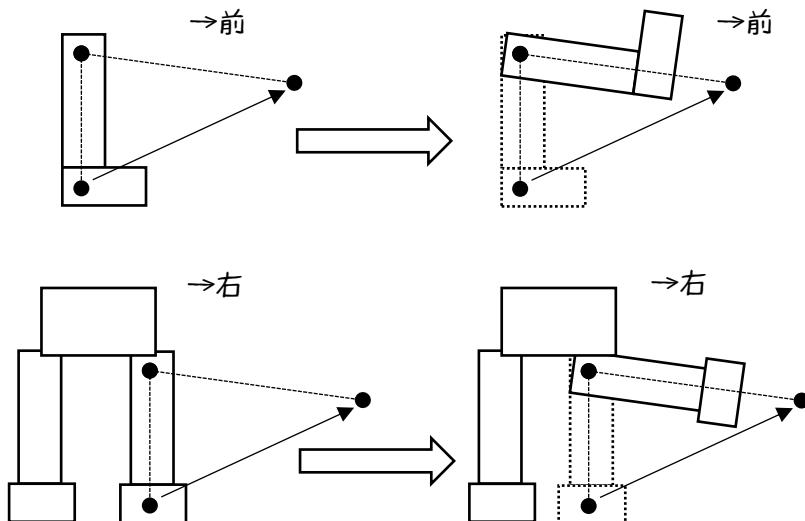
どうやって決定するのかというと「外積」を使います。「外積」が「2つのベクトルに直交する」事を利用します。

では、何ベクトルと何ベクトルで外積をとるのか？を考えましょう。

どの軸を中心回転すべきなのかについて考えてみれば分かるのでは無いでしょうか…

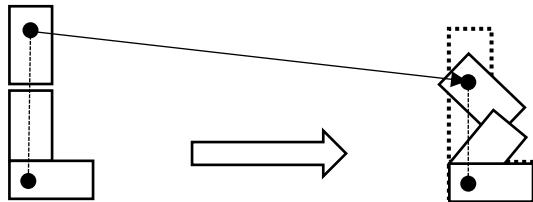
例えば真横から足の IK(説明のための IK なので「ひざ」ではないと思ってください)を見た場合は

下図のように足 IK を前にキック的に出そうとするとき、当然軸は X 方向というか横方向

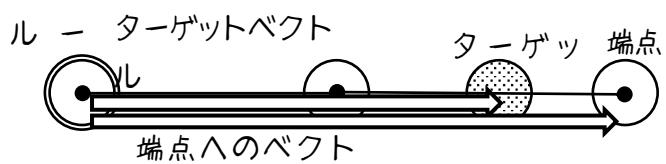


の軸になります。

また、例えば図のように、開脚気味に足を動かそうとするなら軸は前向きというか X 方向になりますね？これを実装しようとする場合、「ルートから元の端点までのベクトル」と「ルートからターゲットまでのベクトル」から外積をとって軸を作ればいいような気がします。



図のように、しゃがみであれば足の IK とセンターとの距離が短くなるため、そのつじつまを合わせるために膝が曲がるわけですが、先ほどの考え方(だけ)だとうまくいかないことがわかりますか? 外積は何と何でとったつけ? 端点までのベクトルと、ターゲットまでのベクトルでした。その場合



図のようにターゲットベクトルと端点ベクトルの方向が一致してしまいます。この状況は 2D の時は「ちょっと回転」でなんとかなっていましたのですが、3D の場合そもそも軸が作れない(=回転できません)。

どういうことがというと 2D の場合は問答無用で Z 軸中心に回転させれば済むのですが、3D の場合はまず軸を決めないと回転することができないので、↑のようにベクトルの方向が一致してしまうと軸が作れずにインバース kinematics ができなくなってしまいます。

納得いかない読者もいると思いますので、代数的に説明しますね? 同じ方向を向いているという事は 2 つのベクトルを A と B とすると

$$B = kA = (kx_a, ky_a)$$

という関係になっていますね? 同じ方向を向いているので定数倍と言う訳です。この状態で A と B の外積を計算してみましょう。

$$A \times B = (y_a k z_a - z_a k y_a, z_a k x_a - x_a k z_a, x_a k y_a - k x_a y_a) = (0, 0, 0)$$

ご覧のように 0 ベクトルになってしまい、軸として使い物になりません。この場合は仕方ないので計算用に上ベクトル(0,1,0)が右ベクトル(1,0,0)と、ターゲットベクトルとの 2 回外積を利用します。

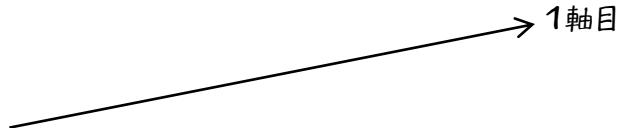
この「2回外積」はいろいろところで使用されている手法ですので覚えておいたほうが良いかと思います。例えば上ベクトルとの 2 回外積について思い浮かべてください。前にカメラ行列を作る時に上ベクトルがついてましたよね?

あの上ベクトルと同じなのですが、今回の上ベクトル(0,1,0)は「仮の上ベクトルです」。前回は「軸を固定するために上ベクトルを刺していた」というような説明の仕方をしましたが実際の目的は

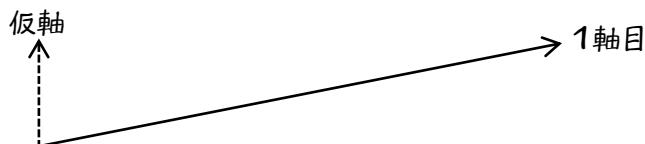
回転に必要な「座標軸」を作る

ことです。「座標軸」ということは「直交する3軸」が必要になります。実際には直交2軸あれば外積で3軸目を算出できるのですが、ともかく「直交する3軸」が必要だと思ってください。

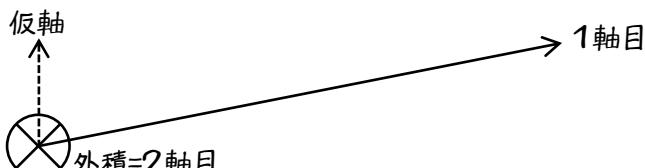
1軸目(カメラなら $\text{eye} \rightarrow \text{target}$)。ボーンの場合はターゲットへのベクトルが軸になる)ですがこれは既知のものなので問題なし。



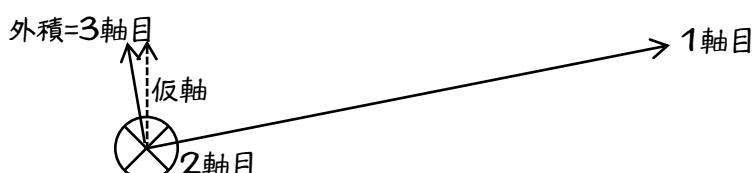
そしてこの軸を固定するためのもう1軸が必要になるのですが、この軸を作るためのベクトルは「今は別に直交してなくてもいい」です。一つ条件があるとすれば1軸目と違うベクトルであること(とはいえ反対向きもダメですが)です。と言う訳で「仮の」上ベクトルを用意します(別に上ベクトルじゃなくてもいいです)。



ここまでよろしいでしょうか? で、この仮軸と1軸目のベクトルを外積します。結果として両方に直交するベクトルができます。



外積の結果が仮軸と1軸目に直交しているのでこの図では手前(もしくは奥へ)向いてるベクトルだと思ってください。そして最後にこの1軸目と2軸目の外積をとります。そうすると双方に直交するベクトルであるため「直交する3軸」を作ることができ、これが「回転に必要な



座標系」となります。

先ほども述べましたが2軸目まで決めれば十分なことが分かりますね。

では「ひざ」の場合はどうなるのでしょうか? 本当にX軸(右ベクトル $(1,0,0)$)でよいのでしょうか?

正直なところここからは作者に聞いてみないとはっきりしたことは分からぬいため推測となります。が、右ベクトルではなく、右ベクトル \vec{R} とターゲットベクトル \vec{T} の外積からベクトル \vec{C} を作成し、今度はその \vec{T} と \vec{C} の外積から \vec{R}' ベクトルを生成しそれを用いるのだと思います。

一つの式として書くと…

$$\vec{R}' = \vec{T} \times (\vec{R} \times \vec{T})$$

ですね。この R' をひざの回転軸とすると仮定して作っていきます。

PMD から IK 情報を読み込む

ボーンデータの直後に IK のデータが入っているので、いつものようにまずは読み込みからやってみましょう。

ボーンデータを読み込み切った後の最初の 2 バイトが IK データの数になります。デフォルトのミクさんならば 7 個ですので、まずはそれを確認してみましょう。

```
uint16_t ikNum=0;  
fread(&ikNum, sizeof(ikNum), 1, fp);
```

次はこれまた 1つ1つのように 1つ1つの IK に関するデータを読み込んでくるのですが、IK の場合は他のデータにない面倒さがあります。それは IK チェーンという物があるのですが、これはその IK に関連するノードだと思ってもらえばいいのですが、その数は「可変」です。そのため構造体内に可変長配列を用意します。

```
struct PMDIK {  
    uint16_t boneIdx;//IK 対象のボーンを示す  
    uint16_t targetIdx;//ターゲットに近づけるためのボーンのインデックス  
    uint8_t chainLen;//間のノードがいくつあるか  
    uint16_t iterations;//試行回数  
    float limit;//一回当たりの回転制限  
    vector<uint16_t> nodeIdx;//間のノード番号  
};
```

当然 chainLen がアライメントに引っ掛かってしまいますので考慮した読み込みにする必要があります。今回はそもそもいっぺんに読み込むこともできないので無理して #pragma pack(1) をするメリットもありませんね。ひとつずつ読み込むなら chainLen は構造体に含める必要はないですね。

```
struct PMDIK {  
    uint16_t boneIdx;//IK 対象のボーンを示す  
    uint16_t targetIdx;//ターゲットに近づけるためのボーンのインデックス  
    uint16_t iterations;//試行回数  
    float limit;//一回当たりの回転制限  
    vector<uint16_t> nodeIdxes;//間のノード番号
```

```

};

vector<PMDIK> pmdlkData(ikNum);
for (auto& ik : pmdlkData) {
    fread(&ik.nodeldxes, sizeof(ik.nodeldxes), 1, fp);
    fread(&ik.targetIdx, sizeof(ik.targetIdx), 1, fp);
    uint8_t chainLen = 0;//間にいくつノードがあるか
    fread(&chainLen, sizeof(chainLen), 1, fp);
    ik.nodeldx.resize(chainLen);
    fread(&ik.iterations, sizeof(ik.iterations), 1, fp);
    fread(&ik.limit, sizeof(ik.limit), 1, fp);
    if (chainLen == 0) continue;//間ノード数が0ならばここで終わり
    fread(ik.nodeldxes.data(), sizeof(ik.nodeldxes[0]), chainLen, fp);
}

```

読み込んでみて、中身が壊れていないことをご確認ください。なお、デバッグ用として読み込み後に以下のようないいコードを書いておくと IK の関係性を一目で見ることができ便利です。

```

//IK デバッグ用
auto getNameFromIdx = [&](uint16_t idx)->string {
    auto it = find_if(_boneNodeTable.begin(), _boneNodeTable.end(),
        [idx](const pair<string, BoneNode>& obj) {
            return obj.second.bonelIdx == idx;
        });
    if (it != _boneNodeTable.end()) {
        return it->first;
    }
    else {
        return "";
    }
};

for (auto& ik : pmdlkData) {
    std::ostringstream oss;
    oss << "IK ボーン番号=" << ik.bonelIdx << ":" << getNameFromIdx(ik.bonelIdx) << endl;
    for (auto& node : ik.nodeldxes) {
        oss << "#t ノードボーン=" << node << ":" << getNameFromIdx(node)<<endl;
    }
    OutputDebugString(oss.str().c_str());
}

```

}

といったコードを書いておけばアドップ出カウンドウに以下のように構造が出来ます。

```
IKボーン番号=80:初めIK
ノードボーン=8:左脚/3
ノードボーン=7:左脚/2
ノードボーン=6:左脚/1
IKボーン番号=81:左腕IK
ノードボーン=16:左腕6
ノードボーン=15:左腕5
ノードボーン=14:左腕4
ノードボーン=13:左腕3
ノードボーン=12:左腕2
IKボーン番号=82:右腕IK
ノードボーン=46:右腕6
ノードボーン=45:右腕5
ノードボーン=44:右腕4
ノードボーン=43:右腕3
ノードボーン=42:右腕2
IKボーン番号=83:左足IK
ノードボーン=39:左足
ノードボーン=38:左足
IKボーン番号=84:右足IK
ノードボーン=88:右ひざ
ノードボーン=89:右足
IKボーン番号=85:左つま先IK
ノードボーン=40:左足首
IKボーン番号=86:右つま先IK
ノードボーン=40:右足首
```

少なくともきちんと読み込んでいることはわかります。また、この例ですとツリーと逆順にさかのぼるように設定されていることが分かると思います。

VMD から「平行移動情報」も読み込む

ここまでボーンに関しては回転ばかりだったため、平行移動情報は読み込んでいませんでしたが IK やセンターに関しては平行移動のデータが入っています。もし IK を実装するのであればこの平行移動情報も読み込まねばなりません。まずはキーフレーム構造体を変更します。

```
///キーフレーム構造体
struct KeyFrame {
    unsigned int frameNo;//フレームNo.(アニメーション開始からの経過時間)
    XMVECTOR quaternion;//クオータニオン
    XMFLOAT3 offset;//IK の初期座標からのオフセット情報
    XMFLOAT2 p1, p2;//ベジェの中間コントロールポイント
    KeyFrame(unsigned int fno,XMVECTOR& q,XMFLOAT3& ofst,
             XMFLOAT2& ip1,const XMFLOAT2& ip2):
        frameNo(fno),
        quaternion(q),
        offset(ofst),
        p1(ip1),
        p2(ip2){}
};
```

IK ボーンを利用するための下準備

さらにボーン種別(回転、IK など)によって挙動も変わってきますので、これも取ってこれるようにしておきましょう。種別の enum は PMDActor.cpp 内部でのみ使用するもので、cpp 側の無名名前空間内で定義すればいいでしょう。

```

namespace {//無名名前空間

    //ボーン種別
    enum class BoneType {
        Rotation,//回転
        RotAndMove,//回転＆移動
        IK,//IK
        Undefined,//未定義
        IKChild,//IK 影響ボーン
        RotationChild,//回転影響ボーン
        IKDestination,//IK 接続先
        Invisible//見えないボーン
    };
}

```

これは判別時の右辺にしか使用しないため、データ側は uint32_t や uint16_t で良いと思います。

さらにボーンノードにおける IK 親ボーンも必要となってくるため、ボーンノード構造体にボーン種別と親ボーン IK 番号を付加してこのように変更します。

```

struct BoneNode {
    uint32_t boneIdx;//ボーンインデックス
    uint32_t boneType;//ボーン種別
    uint32_t ikParentBone;//IK 親ボーン
    DirectX::XMFLOAT3 startPos;//ボーン基準点(回転中心)
    std::vector<BoneNode*> children;//子ノード
};

```

また、ここからはインデックスによる指定ばかりになるためボーン名やボーンノードをインデックスで検索できるようにしておくための下準備をしておきます。まずはメンバ変数に名前配列とボーンノード配列を用意します。

```

vector<string> _boneNameArray;//インデックスから名前を検索しやすいように
vector<BoneNode*> _boneNodeAddressArray;//インデックスからノードを検索しやすいように

```

そして PMD ファイル読み込み後のボーン情報構築時にここにも値が入るようにしておきます。

```

_boneNameArray.resize(pmdBones.size());
_boneNodeAddressArray.resize(pmdBones.size());
for(int idx=0;idx<pmdBones.size();++idx){
    auto& pb=pmdBones[idx];
    auto& node=_boneNodeTable(pb.boneName);
    (中略)
    //インデックス検索がしやすいように
    _boneNameArray(idx)=pb.boneName;
    _boneNodeAddressArray(idx)=&node;
}

```

IK 解決のための関数を場合分けできるようにしておく

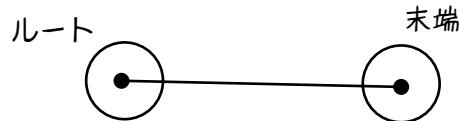
IK の解説でもお話ししましたが、IK の解決法はなにも CCD-IK ばかりではありません。もし「よりシンプルでより効率がいい方法」があるならそれを使うべきです。

例えば IK に影響する点が 3 点しかなく、IK ルートノードと末端ノードに追随するノードと中間ノードしかない とすると以前にお話しした 「余弦定理 IK」 のほうがいいわけです。



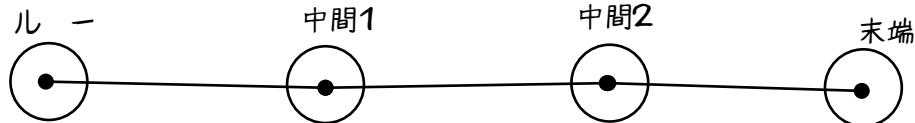
例えばこれはデフォルトの初音ミクモデルであれば例えはそれこそ「右足」→「右ひざ」→「右足首」の部分です。

さらに言うと MMD の IK はルートとターゲットノードしかないものもあり、この場合は 2 点であり IK なんてものを使うまでもなく 2 点間のベクトルを求めれば済む話です。



例えばこれはデフォルト初音ミクモデルであれば「右足首」→「右つま先」にあたります。

では CCD-IK はというと、ルートから末端を含めて 4 点以上の影響点を持ちます。仮に 4 点だとすると以下のようになり 余弦定理では解決できないため CCD-IK が必要となります。



ミクさんでいうと、ここが「ネクタイ」の IK にあたります。ここが一番めんどうですね。とにかくそれぞれの計算において手間が全然違います。同じ手法でやっていては非効率なため、場合分けをできるようにしていきましょう。

まず対象となる3つの関数を定義します。

```
///CCD-IK によりボーン方向を解決
```

```
///@param ik 対象 IK オブジェクト
```

```
void SolveCCDIK(const PMDIK& ik);
```

```
///余弦定理 IK によりボーン方向を解決
```

```
///@param ik 対象 IK オブジェクト
```

```
void SolveCosineIK(const PMDIK& ik);
```

```
///LookAt 行列によりボーン方向を解決
```

```
///@param ik 対象 IK オブジェクト
```

```
void SolveLookAt(const PMDIK& ik);
```

そして IK データの中の nodeIdxes の数(IK の影響を受けるボーンノードの数-1)によって場合分けするのです。

```
void PMDActor::IKSolve() {
    for (auto& ik : _ikData) {
        auto childrenNodesCount = ik.nodeIdxes.size();
        switch(childrenNodesCount) {
            case 0://間のボーン数が 0(ありえない)
                assert(0);
                continue;
            case 1://間のボーン数が 1 のときは LookAt
                SolveLookAt(ik);
                break;
            case 2://間のボーン数が 2 のときは余弦定理 IK
                SolveCosineIK(ik);
                break;
            default://3 以上の時は CCD-IK
                SolveCCDIK(ik);
        }
    }
}
```

```

    }
}

}

```

ひとまず実装はそれなりに難しいため実装のために必要な武器を用意しましょう。

LookAt 行列の作成

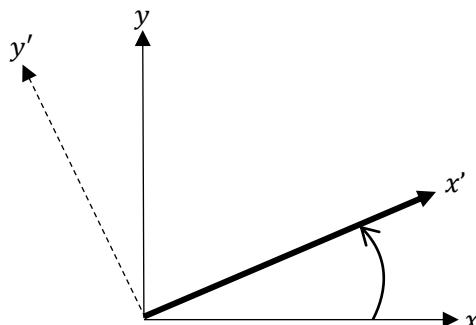
LookAt 行列という行列を定義しましょう。これはかなり強力な武器となりますので、しっかりと理解して実装できるようにしましょう。

そもそも何を目的とする行列かというと X 軸なり Y 軸なり Z 軸なりを特定の方向(ベクトル)に向かせるように回転するための行列です(具体的には Z 軸を特定の方向に向かせます)

本来の用途は敵を自機方向に向かせたりするためのものです。今回はボーンの方向を特定の方向に向かせるために用います。

まず簡単に考えるために 2D で考えてみましょう。ふつうはキャラクタの回転を行いたい場合は回転角度が必要になるため、向かせたい方向のベクトルと X 軸との角度を測り…といった具合に、まずはベクトルを角度に変更する必要があると思います。

しかしベクトルをベクトルのまま座標系を回転させられるとしたらどうでしょう?かなり



便利ですね。まずは 2D の計算からやってみましょう。

図のようにベクトル \vec{y} をベクトル \vec{x}' にするような行列を作ればそれが回転ベクトルとなるため、座標系そのものを回転させ \vec{y} を \vec{y}' にすることができる行列となります。つまりこの回転角度を用いて座標系の回転をすることができるようになります。

ここで基底ベクトルというものを考えます。基底ベクトルというのは長さが 1 であり、それぞれの軸方向を向いてるベクトルを言います。簡単に言うと

$$\vec{x}_1 = (1, 0)$$

であり

$$\vec{y}_1 = (0, 1)$$

です。簡単ですね。普通はベクトル 자체を

$$\vec{v} = (a, b)$$

のようく表すのですが、この「基底ベクトル」を用いて上の \vec{v} を表すと

$$\vec{v} = a\vec{x}_1 + b\vec{y}_1$$

となります。ここまでよろしいですか？

何てことないようく思えますが、この考えが大事なのです。「座標軸を回転させる」という事をこの「基底ベクトルを回転させる」とみなします。つまり

$$\begin{cases} \vec{x}'_1 = (m, n) \\ \vec{y}'_1 = (-n, m) \end{cases} \text{ただし } \sqrt{m^2 + n^2} = 1$$

ここで先ほどの \vec{v} を回転させた \vec{v}' について考えてみましょう。すると

$$\vec{v}' = a\vec{x}'_1 + b\vec{y}'_1$$

であることが分かるかと思います。

つまり回転させた座標軸方向に v のそれぞれの成分ぶん進むわけです。そしてここで実際の座標について考えてみましょう。

$$\vec{v}' = a\vec{x}'_1 + b\vec{y}'_1 = a(m, n) + b(-n, m) = (am - bn, an + bm)$$

そしてこの計算を行える行列について考えてみましょう。まず中身のわからない行列

$$M = \begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

を用意します。そして

$$(a, b) \times \begin{pmatrix} p & q \\ r & s \end{pmatrix} = (am - bn, an + bm)$$

を満たす行列 M を求めればいいことになります。となると必然的に

$$(a, b) \times \begin{pmatrix} m & n \\ -n & m \end{pmatrix} = (am - bn, an + bm)$$

であることが分かります。試しに $(1, 0)$ と $(0, 1)$ にこの行列を乗算すると

$$(1, 0) \times \begin{pmatrix} m & n \\ -n & m \end{pmatrix} = (m, n)$$

$$(0, 1) \times \begin{pmatrix} m & n \\ -n & m \end{pmatrix} = (-n, m)$$

となり、軸が回転していることがわかります。ところで

$$M = \begin{pmatrix} m & n \\ -n & m \end{pmatrix}$$

となります。この形どこかで見たことがないでしょうか？そうです、回転行列です。 $\sqrt{m^2 + n^2} = 1$ であるため、 m と n は \cos と \sin の関係になっているといえます。ただしここでは角度を測っていませんね？

つまり角度を測らずに回転行列が作れたという事です。ベクトルが分かっていさえすればいいのです。当たり前のことに思えますがここを忘れるとながらなくなるので覚えておきましょう。

この回転行列を用いて回転するという事はどういう事でしょうか？

$$(a \ b) \begin{pmatrix} m & n \\ -n & m \end{pmatrix} = (am - bn \ an + bm)$$

それぞれの要素を見るとちょうど「回転後の軸のそれぞれの成分に対して、対象のベクトル成分を内積した形になっている」と思います。分かりづらいかもしれません、こう書いてみると少しばかりやさしいかと思います。

$$M = \begin{pmatrix} m & n \\ -n & m \end{pmatrix} = \begin{pmatrix} \text{回転後 X 軸} \\ \text{回転後 Y 軸} \end{pmatrix} = \begin{pmatrix} X' \text{軸の X 成分} & X' \text{軸の Y 成分} \\ Y' \text{軸の X 成分} & Y' \text{軸の Y 成分} \end{pmatrix}$$

これを3Dに拡張すると

$$\begin{pmatrix} \text{回転後 X 軸} \\ \text{回転後 Y 軸} \\ \text{回転後 Z 軸} \end{pmatrix} = \begin{pmatrix} X' \text{軸の X 成分} & X' \text{軸の Y 成分} & X' \text{軸の Z 成分} \\ Y' \text{軸の X 成分} & Y' \text{軸の Y 成分} & Y' \text{軸の Z 成分} \\ Z' \text{軸の X 成分} & Z' \text{軸の Y 成分} & Z' \text{軸の Z 成分} \end{pmatrix}$$

となります。LookAt 行列に関しては「方向だけ」つまり「回転行列の一種」なので、平行移動成分はなしで考えます。つまり LookAt 行列は

$$\begin{pmatrix} X' \text{軸の X 成分} & X' \text{軸の Y 成分} & X' \text{軸の Z 成分} & 0 \\ Y' \text{軸の X 成分} & Y' \text{軸の Y 成分} & Y' \text{軸の Z 成分} & 0 \\ Z' \text{軸の X 成分} & Z' \text{軸の Y 成分} & Z' \text{軸の Z 成分} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} X'_x & X'_y & X'_z & 0 \\ Y'_x & Y'_y & Y'_z & 0 \\ Z'_x & Z'_y & Z'_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

となります。もし確認したければ X 軸(1,0,0)と Y 軸(0,1,0)と Z 軸(0,0,1)がそれぞれ回転後の軸になるかどうか確認すればいいので、疑わしいと思う読者は手計算してみて下さい。

$$\begin{aligned} (1,0,0) &\rightarrow (x'_x, x'_y, x'_z) \\ (0,1,0) &\rightarrow (y'_x, y'_y, y'_z) \\ (0,0,1) &\rightarrow (z'_x, z'_y, z'_z) \end{aligned}$$

このようになるのが分かると思います（気づいた読者も多いと思いますが、結局左側は単位行列になるので当然の結果なのです）

さてこれで LookAt 行列ができるかというとそう甘くもないのです。どうやってそれぞれの回転後の軸を決めるのでしょうか？まず向かせたいベクトルは「敵からみた自機」や「ルートボーンから見たターゲットボーン」の2点を結べば作れます。

問題はもう一軸です(2軸が判明すれば外積で最後の1軸は求まります)。これって何かに似ていますよね? そう「ビュー行列」です。あれは中身がこのLookAt行列の逆行列みたいな状態になっているのです(ただしカメラ座標も考慮されるため平行移動成分がありますが)。

ビュー行列を作る時には「上ベクトル」を渡していたと思います。あれは真上を向いてさえいなければ外積2回で座標軸が作れてしまうからです。

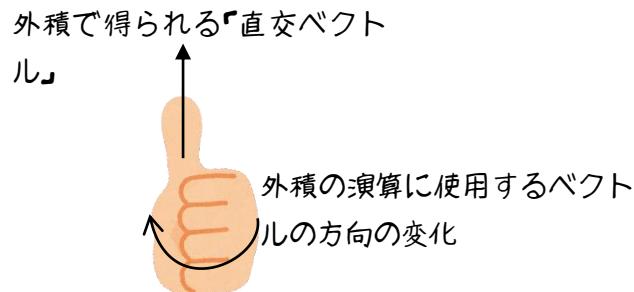
ここで外積に関して注意点ですが、外積は掛け算の順序によって符号が入れ替わります。例えば

$$(x_a, y_a, z_a) \times (x_b, y_b, z_b) = - (x_b, y_b, z_b) \times (x_a, y_a, z_a)$$

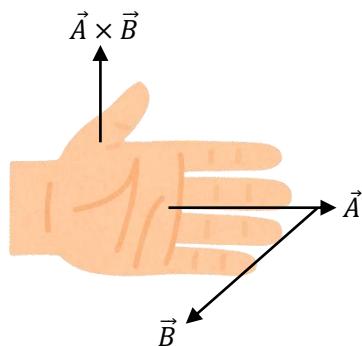
というわけです。ベクトルで向きが入れ替わるってことは、真反対に向くわけです。ベクトルの向きを考えず外積とっちゃうと大変なことになります。

というわけで順序を考えなければならんのですが、どういう法則で外積後のベクトルの向きが決まるのかというと「(右 or 左)ねじの法則」によって決まっています。これは右手系の時は右ねじ、左手系の時は左ねじなのです。

今は左手系を扱っているため左ねじで考えますが、具体的には
このような形で外積の方向が決まります。



例えば3Dにおける $\vec{A} \times \vec{B}$ の方向が知りたければまず親指以外の指先を \vec{A} に向けます。



その後にベクトル \vec{B} 方向に握りこむようにしたときに親指の向いている方向が外積による

直交ベクトルというわけです。よほど手が柔らかくない限り逆向きにはならないため、これで外積の方向がわかります。これによって軸をコントロールします。ともかくビュー行列作成の XMMatrixLookAtLH 関数に倣って補助ベクトルを使って LookAt できる関数を作りましょう。

```
///Z 軸を特定の方向を向かす行列を返す関数
///@param lookat 向かせたい方向ベクトル
///@param up 上ベクトル
///@param right 右ベクトル
XMMATRIX LookAtMatrix(const XMVECTOR& lookat, XMFLOAT3& up, XMFLOAT3& right) {
    //向かせたい方向(z 軸)
    XMVECTOR vz = lookat;
    //((向かせたい方向を向かせたときの)仮の y 軸ベクトル
    XMVECTOR vy = XMVector3Normalize(XMLoadFloat3(&up));
    //((向かせたい方向を向かせたときの)y 軸
    XMVECTOR vx = XMVector3Normalize(XMVector3Cross(vz, vx));
    XMVECTOR vx = XMVector3Normalize(XMVector3Cross(vy, vz));
    vy = XMVector3Normalize(XMVector3Cross(vz, vx));
    ///LookAt と up が同じ方向を向いてたら right 基準で作り直す
    if (abs(XMVector3Dot(vy, vz).m128_f32[0]) == 1.0f) {
        //仮の X 方向を定義
        vx = XMVector3Normalize(XMLoadFloat3(&right));
        //向かせたい方向を向かせたときの Y 軸を計算
        vy = XMVector3Normalize(XMVector3Cross(vz, vx));
        //真の X 軸を計算
        vx = XMVector3Normalize(XMVector3Cross(vy, vz));
    }
    XMMATRIX ret = XMMatrixIdentity();
    ret.r[0] = vx;
    ret.r[1] = vy;
    ret.r[2] = vz;
    return ret;
}
```

かなり XMVECTOR や XMMATRIX の SIMD 系のメンバが見えてしまってるので、ちょっとわかりづらいかもしれません。XMMATRIX は XMVECTOR を縦に 4 つ並べただけで、XMMATRIX のメンバの r は XMVECTOR を 4 つ並べただけです。さらに言うと XMVECTOR は 32bit の float(m128_f32) を横に 4 つ並べただけです。2 次元配列のようになっていると思ってもらえばいいですね。

しかしこの関数は「 z 軸を特定の方向に向かせる関数」です。このままでは「任意のベクトルを特定の方向に向かせる」ことはできません。

ですので少し面倒ですがひと手間かけます。

- ① z 軸を任意のベクトルに向ける行列を計算する
- ② ①で計算した行列の逆行列(回転だけなので転置でOK)を計算する
- ③ z 軸を特定の方向に向かせる行列を計算する
- ④ ②でできた行列と③でできた行列を乗算する

という手順を行えば任意のベクトルを特定の方向に向かせる行列を作ることができます。

```
///特定のベクトルを特定の方向に向けるための行列を返す
///@param origin 特定のベクトル
///@param lookat 向かせたい方向
///@param up 上ベクトル
///@param right 右ベクトル
///@retval 特定のベクトルを特定の方向に向けるための行列
XMMATRIX LookAtMatrix(const XMVECTOR& origin, const XMVECTOR& lookat, XMFLOAT3& up,
XMFLOAT3& right) {
    return XMMatrixTranspose(LookAtMatrix(origin, up, right)*
        LookAtMatrix(lookat, up, right));
}
```

では LookAt 行列ができましたので先に進みましょう。

IK ボーンがルートと末端しかない時の解決

既に作っている LookAt 回転による解決を行います。先に作成しておいた SolveLookAt 関数の実装を行いましょう。既に LookAt 関数は作っておりますので、ルートと末端それぞれの座標を XMVECTOR として代入してあげればいいだけです。

やり方としては、IK を動かす前のルートノードとターゲットノードから作ったベクトルと、IK を動かした後のルートノードとターゲットノードから作ったベクトルから回転行列を作ればいいのですが、単純に先ほどの LookAtMatrix 関数を使えば解決できます。

void

```

PMDActor::SolveLookAt(const PMDIK& ik) {
    //この関数に来た時点ではノードはひとつしかなく、チェーンに入っているノード番号は
    //IK のルートノードのものなので、このルートノードから末端に向かうベクトルを考える
    auto rootNode=_boneNodeAddressArray[ik.nodexes[0]];
    auto targetNode = _boneNodeAddressArray[ik.boneIdx];

    auto rpos1 = XMLoadFloat3(&rootNode->startPos);
    auto tpos1 = XMLoadFloat3(&targetNode->startPos);

    auto rpos2 = XMVector3TransformCoord( opos1,_boneMatrices[ik.nodexes[0]]);
    auto tpos2 = XMVector3TransformCoord( tpos1, _boneMatrices[ik.boneIdx]);

    auto originVec = XMVectorSubtract(tpos1,rpos1);
    auto targetVec = XMVectorSubtract(tpos2,rpos2);

    originVec = XMVector3Normalize(originVec);
    targetVec = XMVector3Normalize(targetVec);
    _boneMatrices[ik.nodexes[0]]=LookAtMatrix(originVec, targetVec,
                                                XMFLOAT3(0, 1, 0), XMFLOAT3(1, 0, 0));
}

```

これは簡単だと思います。次に簡単なのは余弦定理 IK なので実装していきたいと思います。

余弦定理 IK の実装

ここからはルートから末端までの点が3点しかない(つまり角度を余弦定理で計算できる)パターンについて解説していきます。余弦定理 IK に関してはこの章(第12章)の第2項を参照しながら読み進めてください。

まず、余弦定理 IK にとって重要なのはルートと末端と、それぞれのボーンの長さです。中間の点には自動計算されるため長さを測る以外の意味はありません。



```

vector<XMVECTOR> positions;//IK構成点を保存
std::array<float, 2> edgeLens;//IKのそれぞれのボーン間の距離を保存

```

中間点は↑のコードの「ボーン間の距離」を保持するためにあると思ってください。次に座標変換後のIKボーンの座標を取得しておきます。

```
//ターゲット(末端ボーンではなく、末端ボーンが近づく目標ボーンの座標を取得)
auto& targetNode = _boneNodeAddressArray[iK.boneIdx];
auto targetPos = XMVector3Transform(
    XMLoadFloat3(&targetNode->startPos),
    _boneMatrices[iK.boneIdx]);
```

そのままの順序で余弦定理IKを計算してもいいのですが、データ構造上順序が末端からになっているため逆順にします。

```
//IKチェーンが逆順なので、逆に並ぶようにしている
//末端ボーン
auto endNode = _boneNodeAddressArray[iK.targetIdx];
positions.emplace_back(XMLoadFloat3(&endNode->startPos));
//中間及びルートボーン
for (auto& chainBoneIdx : iK.nodeIdxes) {
    auto boneNode = _boneNodeAddressArray[chainBoneIdx];
    positions.emplace_back(XMLoadFloat3(&boneNode->startPos));
}
//ちょっと分かりづらいと思ったので逆にしておきます。そうでもない人はそのまま
//計算してもらって構ないです。
reverse(positions.begin(), positions.end());
```

で、前にやった2Dの時と同様に予め長さを測っておきます。

```
//元の長さを測っておく
edgeLens[0] = XMVector3Length(XMVectorSubtract(positions[1], positions[0])).m128_f32[0];
edgeLens[1] = XMVector3Length(XMVectorSubtract(positions[2], positions[1])).m128_f32[0];
```

長さが取得できたので、あとは現在のルートボーンと末端ボーンの座標を取得します。

```
//ルートボーン座標変換(逆順になっているため使用するインデックスに注意)
positions[0] = XMVector3Transform(positions[0], _boneMatrices[iK.nodeIdxes[1]]);
//真ん中はどうせ自動計算されるので計算しない
//先端ボーン
positions[2] = XMVector3Transform(positions[2], _boneMatrices[iK.boneIdx]);
```

ただし上のコードにおいては末端ボーンの座標変換を IK ボーンにしています。理由は末端ボーンは IK ボーンに追随し、そちらが優先になっているからです。

あとは 2D の時と同じです。なお、以前の説明を見ながら A と B と C のベクトルに対応させて考えてみて下さい。変数名も前の

//ルートから先端へのベクトルを作っておく

```
auto linearVec = XMVectorSubtract(positions[2], positions[0]);
float A = XMVector3Length(linearVec).m128_f32[0];
float B = edgeLens[0];
float C = edgeLens[1];
linearVec = XMVector3Normalize(linearVec);
```

//ルートから真ん中への角度計算

```
float theta1 = acosf((A*A + B * B - C * C) / (2 * A*B));
```

//真ん中からターゲットへの角度計算

```
float theta2 = acosf((B*B + C * C - A * A) / (2 * B*C));
```

次に軸を作りますがその前に、ひざかどうかを判別するために PMD 読み込み時に「ひざ」が含まれているインデックスを収集しておきます。

```
vector<uint32_t> _kneeIdxes;
```

こういうメンバ変数を作つておいて

```
_kneeIdxes.clear();
```

```
for (int idx = 0; idx < pmdBones.size(); ++idx) {
```

(中略)

```
    string boneName = pb.boneName;
```

```
    if (boneName.find("ひざ") != std::string::npos) {
```

```
        _kneeIdxes.emplace_back(idx);
```

```
}
```

```
}
```

のように「ひざ」の番号を収集しておきます。そうすると

//「軸」を求める

//もし真ん中が「ひざ」であった場合には強制的に X 軸とする。

```

XMVECTOR axis;

if (find(_kneeldxes.begin(), _kneeldxes.end(), ik.nodeIdxes[0]) == _kneeldxes.end()) {
    auto vm = XMVector3Normalize(XMVectorSubtract(positions[2], positions[0]));
    auto vt = XMVector3Normalize(XMVectorSubtract(targetPos, positions[0]));
    axis = XMVector3Cross(vt, vm);
} else {
    auto right = XMFLOAT3(1, 0, 0);
    axis = XMLoadFloat3(&right);
}

```

このように比較的シンプルに軸生成を場合分けできるのではないか。回転角度と軸が分かつたらあとは適用するだけです。

```

//注意点…IK チェーンは根っこに向かってから数えられるため 1 が根っこに近い
auto mat1=XMMatrixTranslationFromVector(-positions(0));
mat1*=XMMatrixRotationAxis(axis,theta1);
mat1*=XMMatrixTranslationFromVector(positions(0));

auto mat2=XMMatrixTranslationFromVector(-positions(1));
mat2*=XMMatrixRotationAxis(axis,theta2-XM_PI);
mat2*=XMMatrixTranslationFromVector(positions(1));

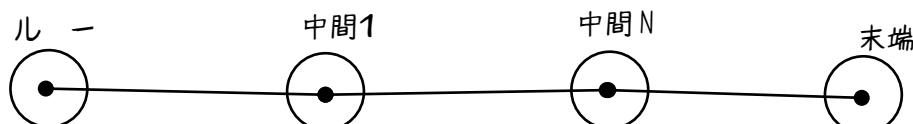
_boneMatrices(ik.nodeIdxes(1))*=mat1;
_boneMatrices(ik.nodeIdxes(0))=mat2*_boneMatrices(ik.nodeIdxes(1));
_boneMatrices(ik.targetIdx)=_boneMatrices(ik.nodeIdxes(0));

```

最後の部分が少し怪しいため後々修正することになるかもしれません。ともかく今は動かすことを第一に考えてください。

CCD-IK の実装

いよいよ関連する点が 4 つ以上の時です。これ以上は CCD-IK などの IK 技術を用いざるを得ません。



もちろん 4 つとは限りません。初音ミクモデルのネクタイ IK は 4 点ですが、髪 IK は 6 点です。モデルによってはそれ以上もあるかもしれません。対象の点が増えれば増えるほど IK 解決の結果が発散しやすくなるため慎重に実装しましょう。

まず今一度確認しますが、CCD-IK はニュートン法のように近似法みたいなものなので、ターゲットと末端の座標がある程度近づいたらそこで計算を打ち切ります。それ以外の場合は IK ごとに試行回数上限が決められているため、その上限まで計算を繰り返します。

まずはターゲットの座標を座標変換します。

```
//ターゲット
auto targetBoneNode = _boneNodeAddressArray[ik.boneIdx];
auto targetOriginPos = XMLoadFloat3(&targetBoneNode->startPos);
```

これは動かすためには当然必要ですね。そして次にこのターゲットの座標に座標変換をかけるのですが、親の座標変換が計算の邪魔になるため逆行列でいったん無効化しておきます。

```
auto parentMat = _boneMatrices[_boneNodeAddressArray[ik.boneIdx]->ikParentBone];
XMVECTOR det;
auto invParentMat = XMMatrixInverse(&det, parentMat);
auto targetNextPos = XMVector3Transform(targetOriginPos,
    _boneMatrices[ik.boneIdx] * invParentMat);
```

次にボーンの座標を動かしながら実際のそれぞれのボーンの回転角度を求めるために現在の座標を計算用に保存しておきます。

```
//末端ノード
auto endPos = XMLoadFloat3(&_boneNodeAddressArray[ik.targetIdx]->startPos);
//中間ノード(ルートを含む)
for (auto& cidx : ik.nodexes) {
    bonePositions.push_back(XMLoadFloat3(&_boneNodeAddressArray[cidx]->startPos));
}
```

さらに回転行列も記録しておく必要があるため末端以外のボーン数ぶん行列も確保しておきます。

```
vector<XMMATRIX> mats(bonePositions.size());
fill(mats.begin(), mats.end(), XMMatrixIdentity());
```

ここで PMD データ特有のものなのですが、「1 回あたりの回転制限」について少し癖があるようです。例えば PMD エディタ上で 6.8° と指定した場合、実際のデータ上ではおよそ 0.03 になっ

ていました。度数法→弧度法変換の結果に思えますがそれにしても値が適切ではありません。

例えば度数法の 6.8° は弧度法であればおよそ 0.119 になるはずが $1/3$ 未満という事になります。ここから考えられるのはおそらく度数法を 180° で割った数値が登録されているという事です。試しに 6.8 を 180 で割ってみると、 $0.037777\cdots$ となります。

この事から、この値をラジアンとして使用するには XM_PI を乗算する必要があります。

```
auto ikLimit = ik.limit*XM_PI;
```

あとは CCD-IK の繰り返し計算を for ループで行っていけば CCD-IK による点の解決ができます。まずはコードを見ていきましょう。少し長いですし、DirectXMath 特有の書き方などがあり分かりづらいかもしれません。とにかく読みましょう。

何これ?となりそうな部分を事前に書いておきますと何度か m128_f32[0] という記述が出てきますが、これは XMVECTOR という構造体の 1 番目の要素です(xyzw の x のことです)。では見ていきましょう。

```
//ik に設定されている試行回数だけ繰り返す
for (int c = 0; c < ik.iterations; ++c) {
    //ターゲットと末端がほぼ一致したら抜ける
    if (XMVector3Length(XMVectorSubtract(endPos, targetNextPos)).m128_f32[0] <= epsilon) {
        break;
    }
    //それぞれのボーンを遡りながら角度制限に引っ掛からないように曲げていく
    //bonePositions は CCD-IK における各ノードの座標をベクタ配列にしたものです
    for (int bidx = 0; bidx < bonePositions.size(); ++bidx) {
        const auto& pos = bonePositions[bidx];
        //対象ノードから末端ノードまでと対象ノードからターゲットまでのベクトル作成
        auto vecToEnd = XMVectorSubtract(endPos, pos); //末端へ
        auto vecToTarget = XMVectorSubtract(targetNextPos, pos); //ターゲットへ
        //両方正規化
        vecToEnd = XMVector3Normalize(vecToEnd);
        vecToTarget = XMVector3Normalize(vecToTarget);

        //ほぼ同じベクトルになってしまった場合は外積できないため次のボーンに引き渡す
        if (XMVector3Length(XMVectorSubtract(vecToEnd, vecToTarget)).m128_f32[0] <= epsilon) {
```

```

        continue;
    }

    //外積計算および角度計算

    auto cross = XMVector3Normalize(XMVector3Cross(vecToEnd, vecToTarget)); //軸になる
    // ↓ 便利な関数だけど中身は cos(内積値)なので 0~90° と 0~-90° の区別がない
    float angle = XMVector3AngleBetweenVectors(vecToEnd, vecToTarget).m128_f32[0];
    angle = min(angle, ikLimit); //回転限界を超てしまった時は限界値に補正

    XMMATRIX rot = XMMatrixRotationAxis(cross, angle); //回転行列作成

    //原点中心ではなく pos 中心に回転
    auto mat = XMMatrixTranslationFromVector(-pos)*
        rot*
        XMMatrixTranslationFromVector(pos);

    mats[bidx] *= mat; //回転行列を保持しておく(乗算で回転重ね掛けを作つておく)

    //対象となる点をすべて回転させる(現在の点から見て末端側を回転)
    for (auto idx = bidx - 1; idx >= 0; --idx) { //自分を回転させる必要はない

        bonePositions[idx] = XMVector3Transform(bonePositions[idx], mat);
    }

    endPos = XMVector3Transform(endPos, mat);

    //もし正解に近くなつたらループを抜ける
    if (XMVector3Length(XMVectorSubtract(endPos, targetNextPos)).m128_f32[0] <= epsilon) {
        break;
    }
}
}

```

2D の CCD-IK についてはすでに説明していますので、特筆すべき処理を挙げると

```

//ほぼ同じベクトルになつてしまつた場合は外積できないため次のポーンに引き渡す
if (XMVector3Length(XMVectorSubtract(vecToEnd, vecToTarget)).m128_f32[0] <= epsilon) {

    continue;
}

```

のように軸を計算できない状況であれば計算を行わない部分と、次に軸を計算できる場合はそれをもとに軸を作つて、回転角度を評価する部分ですね。

```

//外積計算および角度計算
auto cross = XMVector3Normalize(XMVector3Cross(vecToEnd, vecToTarget)); //軸になる

```

```
// ↓便利な関数だけど中身は cos(内積値)なので 0~90° と 0~-90° の区別がない
float angle = XMVector3AngleBetweenVectors(vecToEnd, vecToTarget).m128_f32[0];
angle = min(angle,ikLimit); //回転限界を超えた時は限界値に補正
```

そして最後に、実際のボーン行列に反映するために、ここで行われる CCD-IK による回転処理を乗算で蓄積していく、それを記録していく部分です。

```
mats[bidx] *= mat; //回転行列を保持しておく(乗算で回転重ね掛けを作つておく)
```

これくらいです。座標解決そのものは 2D のときと変わらないのですが「3D では軸を作らなければならぬ」ため、外積によって軸を作るという部分が必要なのと、最終的には「点」ではなく、「ボーン回転行列」が必要であるためこれらのコードが必要になるのです。

そしてこのままで mats 配列に行列が代入されただけですので、ボーン行列に割り当てていきます。

```
int idx = 0;
for (auto& cidx : ik.nodexes) {
    _boneMatrices[cidx] = mats[idx];
    ++idx;
}
```

もちろん、当然のようにこのままで親子関係が破たんしていますので、ルートボーン以降の再帰処理を行ってあげます。

```
auto rootNode = _boneNodeAddressArray[ik.nodexes.back()];
RecursiveMatrixMultiply(rootNode, parentMat, true);
```

以上です。ここまでを特に問題なく書ければ、こちらであらかじめ用意しておいたスクワットモーションくらいはできるようになるでしょう。

基本的なところはこれで十分だと思いますが、各自で用意した PMD ファイルと VMD ファイルでは不具合が起きると思います。そこは読者自身で解決していただきますようよろしくお願ひいたします。

その他 VMD や IK の仕様を考慮する

ここまで実装すると、色々と対応できていない部分があるため、最後に細かい部分への対応を書いておきます。

IK を切る(IK を無効にする)

MMD でモーションを作った方はご存知だと思いますが、キーフレーム単位で IK を個別に切ることもできます。逆に言うと、IK を切ったモーションを動かす時に IK が有効であるような処理をしていると壊れてしまいます。

そこで IK のオン、オフのフラグをモーションデータから探したいのですが、これが少し離れた場所にあります。IK のオン、オフは後から MMD に追加された仕様らしくデータとしては末尾にあります。ネット上の資料も少ない部分なのでここから先は特に注意しましょう。

*スキン以降の VMD フォーマットは、VMD が MMD 等から出力されたバージョンによって違っため、拾い物を使用する際には注意が必要です。もし、以前のバージョンの場合データの対象箇所が違ったため、拾い物でおかしなことが発生した場合は MMD で読み込んでもう一度出力し直すなどしてください。

詳しくは後述しますが、独特の部分がありますので、ここは軽く読み飛ばさずにじっくり読んでください。まずは IK オンオフのデータの場所を探しましょう。前述したように少し遠いです。

ヘッタ → モーションデータ → 表情データ → カメラデータ → ライトデータ → セルフ影データ
→ IK オンオフデータ

今はモーションデータまで読み込んでいる状態なので間に表情とカメラとライトのデータを読まないと到達できない事がわかります。ここは固定ではないですし fseek による読み飛ばしもできませんので読んでしまいます。

まず表情データ構造体からセルフ影データ構造体まで作って、パッケージと読み込んでいきます。今回の主役は IK オン/オフなので細かい所は気にしないようにしましょう。

```
#pragma pack(1)
// 表情データ(頂点モーフデータ)
struct VMDMorph {
    char name[15];//名前(パディングしてしまう)
    uint32_t frameNo;//フレーム番号
    float weight;//ウェイト(0.0f~1.0f)
}; // 全部で 23 バイトなので pragmapack で読む
#pragma pack()
```

```

uint32_t morphCount = 0;
fread(&morphCount, sizeof(morphCount), 1, fp);
vector<VMDMorph> morphs(morphCount);
fread(morphs.data(), sizeof(VMDMorph), morphCount, fp);

#pragma pack(1)
//カメラ
struct VMDCamera {
    uint32_t frameNo; // フレーム番号
    float distance; // 距離
    XMFLOAT3 pos; // 座標
    XMFLOAT3 eulerAngle; // オイラー角
    uint8_t Interpolation[24]; // 補完
    uint32_t fov; // 視界角
    uint8_t persFlg; // パースフラグ ON/OFF
}; //61 バイト(これも pragma pack(1)の必要あり)
#pragma pack()

uint32_t vmdCameraCount = 0;
fread(&vmdCameraCount, sizeof(vmdCameraCount), 1, fp);
vector<VMDCamera> cameraData(vmdCameraCount);
fread(cameraData.data(), sizeof(VMDCamera), vmdCameraCount, fp);

// ライト照明データ
struct VMDLight {
    uint32_t frameNo; // フレーム番号
    XMFLOAT3 rgb; // ライト色
    XMFLOAT3 vec; // 光線ベクトル(平行光線)
};

uint32_t vmdLightCount = 0;
fread(&vmdLightCount, sizeof(vmdLightCount), 1, fp);
vector<VMDLight> lights(vmdLightCount);
fread(lights.data(), sizeof(VMDLight), vmdLightCount, fp);

```

```

#pragma pack(1)
// セルフ影データ
struct VMDSelfShadow {
    uint32_t frameNo; // フレーム番号
    uint8_t mode; //影モード(0:影なし、1:モード 1、2:モード 2)
    float distance; //距離
};

#pragma pack()

uint32_t selfShadowCount = 0;
fread(&selfShadowCount, sizeof(selfShadowCount), 1, fp);
vector<VMDSelfShadow> selfShadowData(selfShadowCount);
fread(selfShadowData.data(), sizeof(VMDSelfShadow), selfShadowCount, fp);

```

随所に#pragma pack(1)～#pragma pack()がありますが、ただのアライメント回避ですので気にせずここまで読み進めましょう。ここまでが読み込めたらいいよいよ目的のIK有効無効データとなりますが、今までのデータに比べると特殊な形になっています。

有効無効の設定がキーフレームごとに一括で設定データが入っているような感じです。MMDのエディタ上の設定項目部分を見ればわかりますので、確認しておいてください。

ともかくここからは気を遣って読み込んでいきますがその前にどういうデータにすればいいのか考えましょう。まずデータはキーフレームごとに存在し、その中にオンオフ設定できるIKボーン名とオンオフフラグが記述されています。

そこでまずはこのような構造体を用意します。

```

//IK オンオフデータ
struct VMDIKEnable {
    uint32_t frameNo;//キーフレームがあるフレーム番号
    std::unordered_map<std::string, bool> ikEnableTable;//名前とオンオフフラグマップ
};

```

あとはこれが複数ありますのでベクタ配列にでも入れておきます。ここはメモリの連續性は重要ではありませんのでvectorじゃなくてlist等でも構いません。その辺の議論をする書籍ではないので手っ取り早いvectorにしておきます。

```
vector<VMDIKEnable> _ikEnableData;
```

あとは少々面倒な読み方ですが読み込んでいきます。まずはそもそもデータ数を読み込み

```
//IKオンオフ切り替わり数
uint32_t ikSwitchCount=0;
fread(&ikSwitchCount, sizeof(ikSwitchCount), 1, fp);
```

あとはキーフレームごとにループしつつ読み込むだけです。

```
_ikEnableData.resize(ikSwitchCount);
for (auto& ikEnable : _ikEnableData) {
    //キーフレーム情報なのでまずはフレーム番号読み込み
    fread(&ikEnable.frameNo, sizeof(ikEnable.frameNo), 1, fp);
    //次に可視フラグがありますがこれは使用しないので1バイトシークでも構いません
    uint8_t visibleFlg = 0;
    fread(&visibleFlg, sizeof(visibleFlg), 1, fp);
    //対象ボーン数読み込み
    uint32_t ikBoneCount = 0;
    fread(&ikBoneCount, sizeof(ikBoneCount), 1, fp);
    //ループしつつ名前とON/OFF情報を取得
    for (int i = 0; i < ikBoneCount; ++i) {
        char ikBoneName[20];
        fread(ikBoneName, _countof(ikBoneName), 1, fp);
        uint8_t flg = 0;
        fread(&flg, sizeof(flg), 1, fp);
        ikEnable.ikEnableTable[ikBoneName] = flg;
    }
}
```

これでデータが手元にある状態になりますのでIK適用時に名前とフラグを見て処理を飛ばすかどうかを決めましょう。モーション更新の際に

```
//前にもやったようにIKオンオフ情報をフレーム番号で逆から検索
auto it=find_if(_ikEnableData.rbegin(),_ikEnableData rend(),
[frameNo](const VMDIKEEnable& ikenable) {
    return ikenable.frameNo <= frameNo;
```

```
});
```

この辺が対象の IK 情報ですが、対象となるデータがなければもちろん end()になるのでチェックしておいてください。

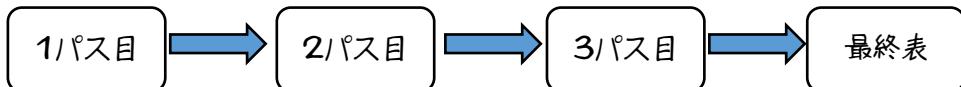
```
for (auto& ik : _ikData) {//IK 解決のためのループ
    if (it != _ikEnableData.end()) {
        auto ikEnableIt = it->ikEnableTable.find(_boneNameArray[ik.boneIdx]);
        if (ikEnableIt != it->ikEnableTable.end()) {
            if (!ikEnableIt->second) {//もし OFF なら打ち切れます
                continue;
            }
        }
    }
}
```

(略)

かなり面倒だったと思いますが、以上で MMD の IK に関しては本当に終了です。お疲れ様でした。

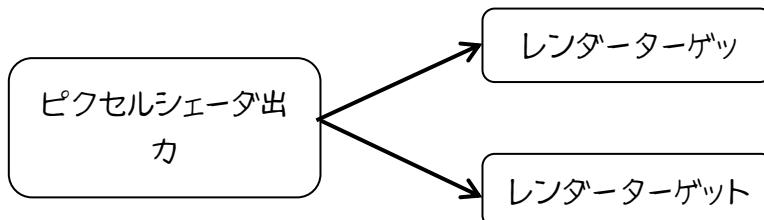
マルチレンダーターゲットとその応用について

マルチレンダーターゲットと聞いて「あれ？もうそれはやつたんじゃない？」と思われるかもしないため違いを説明します。ここまでレンダーターゲットを複数使ってきたのは



このように前のパスを次のパスで利用したり加工したりするものでした。これをマルチパスレンダリングと言います。

今回の「マルチレンダーターゲット」とは言葉通り「一回の描画におけるレンダーターゲットが複数」つまりピクセルシェーダの出力先が複数になるものです。



ちなみに上の図のようにレンダーターゲット0とレンダーターゲット1に出力する場合、二つのレンダーターゲットの解像度は同じでなければいけません。この仕様は当たり前のようにですが僕はここでハマりましたので一応お気を付けください。

ピクセルシェーダの出力を複数にする(色, 法線)

まずピクセルシェーダを書き換える前にパイプラインステートにおけるレンダーターゲットの数を変える必要があります。

1枚目のレンダーターゲットを増やします。複数あればいいだけですので、1枚目用バッファを配列化しておきます(※別に配列でなくても構いません。出力先が複数あればいいだけです)。まずはレンダーターゲットを2つにしましょう。

現在は

```
ComPtr<ID3D12Resource> _peraResource;
```

となっている部分を配列かvectorにしましょう。とりあえずarrayを使用して2つぶん。

```
std::array<ComPtr<ID3D12Resource>, 2> _pera1Resources;
```

リソースの種類は同じなので単純にループに差し替えましょう。

```
for (auto& res : _pera1Resources) {
```

```
    result = _dev->CreateCommittedResource(&heapProp,  
        D3D12_HEAP_FLAG_NONE,  
        &resDesc,  
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,  
        &clearValue,
```

```

    IID_PPV_ARGS(res.ReleaseAndGetAddressOf()));

if (!CheckResult(result)) {
    //エラー対処
}

}

レンダーターゲットビューもそれぞれ作ります。
for (auto& res : _peralResources) {
    _dev->CreateRenderTargetView(res.Get(),
        &rtvDesc, handle);
    handle.ptr +=

    _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
}

(*デスクリプタヒープ作成の際のデスクリプタ数を増やしておくのを忘れずに)
シェーダリソースビューもそれぞれ作ります。
for (auto& res : _peralResources) {
    _dev->CreateShaderResourceView(res.Get(),
        &srvDesc,
        handle);
    handle.ptr +=

    _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
}

描画時のバリアも2つぶん用意します。
for (auto& res : _peralResources) {
    Barrier(res.Get(),
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
        D3D12_RESOURCE_STATE_RENDER_TARGET);
}

次にパイプラインステート側の設定を複数レンダーターゲットにします。レンダーターゲット数と、それぞれのレンダーターゲットのフォーマットを指定します。
pIsDesc.NumRenderTargets = 2;//レンダーターゲット数
pIsDesc.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM;
pIsDesc.RTVFormats[1] = DXGI_FORMAT_R8G8B8A8_UNORM;

ひとまずこの状況でエラーなく出力できているかどうかを確認しましょう。次は本当に複数の情報を出力しますよ？

既にレンダーターゲット先を2つにしているため1枚目と2枚目…別の情報を出力してみま

```

しょう。ピクセルシェーダ側でやることは簡単です。2種類出力できればいいのです。

C++側はレンダーターゲット指定(OMRenderTargets)の部分を複数指定にするだけです。

```
auto handle= _peraRTVHeap->GetCPUDescriptorHandleForHeapStart();
handle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
D3D12_CPU_DESCRIPTOR_HANDLE rtvs[2] = {
    _peraRTVHeap->GetCPUDescriptorHandleForHeapStart(),
    Handle
};
_cmdList->OMSetRenderTargets(2, rtvs, false,
    &_dsvHeap->GetCPUDescriptorHandleForHeapStart());
```

勿論クリアも複数になっていますね。

```
for (auto& rt : rtvs) {
    _cmdList->ClearRenderTargetView(rt, clsCir, 0, nullptr);
}
```

HLSL側はどうやって指定しましょう。そこはC言語と同じで複数の情報を出力したければ構造体を利用します。

```
struct PixelOutput {
    float4 col:SV_TARGET0;//カラー値を出力
    float4 normal:SV_TARGET1;//法線を出力
};
```

//ピクセルシェーダ

```
PixelOutput PS(Output input) {
```

のように出力するターゲットごとに SV_TARGET(n)を用意します。なお出力できるのは今のところ SV_TARGET0～SV_TARGET7までの8個となっていますので8個までやりくりすることになります。まずは色と法線を出力しましょう。

色はここまで計算した色を出力すればいいです。法線を追加で出力していきましょう。

```
PixelOutput output;
output.col = float4(ret.rgb*shadowWeight, ret.a);
output.normal.rgb = float3((input.normal.xyz+1.0f)/2.0f);
output.normal.a = 1;
return output;
```

ひとまずは以上です(※法線の部分はカラー値として出力できるように加工しておきましょう。)

まずはこのまま実行してみましょう。結果はまだ変わらないはずです。どこかでやり忘れたればエラーが起きているはずなのでご確認ください。次はこの法線が出力されているかどうかを確認します。

今レンダーターゲットの 2 つ目に出力したわけですから、うまくいっていれば対応する 2 枚目のシェーダ「リソースビュー」から「法線情報」が見れるはずです。

まずペラボリ用のルートシグネチャのレンジ内の 1 枚目テクスチャ指定部分の NumDescriptors を 1 から 2 に増やしましょう。

```
range[1].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;//t  
range[1].BaseShaderRegister = 0;//0  
range[1].NumDescriptors = 2;//t0, t1
```

注意すべきは、もし他のテクスチャを利用していく場合は、他のレジスタを 1 つずつずらさなければならぬのでそこは各自で対応してください(t1→t2, t2→t3 など)

(↓例)

```
//通常カラー、法線(2枚)  
range[1].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;//t  
range[1].BaseShaderRegister = 0;//0  
range[1].NumDescriptors = 2;//t0, t1
```

//歪みテクスチャ用

```
range[2].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;//t  
range[2].BaseShaderRegister = 2;//1  
range[2].NumDescriptors = 1;//t2
```

//深度値テクスチャ用

```
range[3].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;//t  
range[3].BaseShaderRegister = 3;//2  
range[3].NumDescriptors = 2;//t3, t4
```

ペラ描画シェーダ側も合わせて

```
Texture2D<float4> tex : register(t0); //通常カラー
```

```

Texture2D<float4> texNormal : register(t1); //法線

Texture2D<float4> distTex : register(t2);
//深度値用
Texture2D<float> depthTex : register(t3); //デプス
Texture2D<float> lightDepthTex : register(t4); //ライトデプス

とします。この結果をまずは表示させてみましょう。ビューポート指定をいじってもいいのですがDraw回数が増えて面倒なので私はピクセルシェーダ側だけでやってみました。
if (input.uv.x < 0.2 && input.uv.y < 0.2) { //深度出力
    float depth = depthTex.Sample(smp, input.uv * 5);
    depth = 1.0f - pow(depth, 30);
    return float4(depth, depth, depth, 1);
} else if (input.uv.x < 0.2 && input.uv.y < 0.4) { //ライトからの深度出力
    float depth = lightDepthTex.Sample(smp, (input.uv - float2(0, 0.2)) * 5);
    depth = 1 - depth;
    return float4(depth, depth, depth, 1);
} else if (input.uv.x < 0.2 && input.uv.y < 0.6) { //法線出力
    return texNormal.Sample(smp, (input.uv - float2(0, 0.4)) * 5);
}

```

細かい所はともかく、今回注目すべきは最後の else if 以降の部分ですね。法線を出力できているかどうかです。ちなみに今回は UV 値を 5 分割して左側に情報を出すようにしています。正常に出力できていれば



このように左側に法線が出力されるようになります
次の項ではこの別々に出力したものを利用してみましょう。

色情報と法線情報からディファードシェーディングを行う

本書ではライティングが平行光線のみであるため、ディファードシェーディングの意味は全く

ないので原理だけでも触れておきたいと思います。なお、この項の実装は学習を目的としたテストのようなものなので Git 等を用いて後から戻せるようにしておいてください。

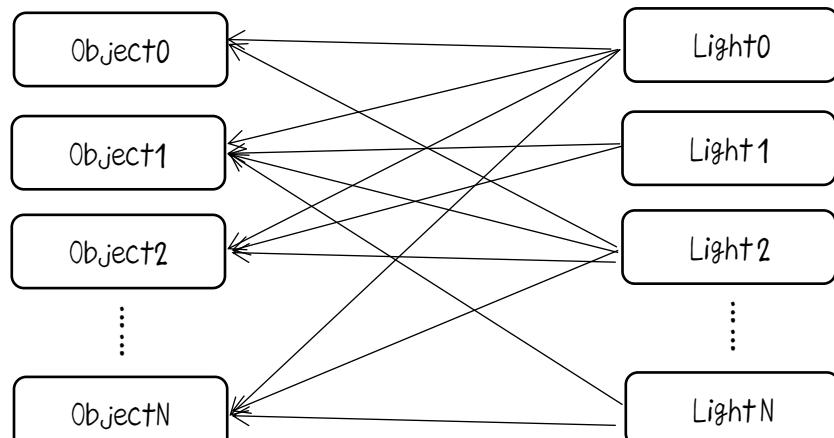
そもそもディファードシェーディングの目的は「光源が増えた際に光源の数だけシェーディングを行わなくてはならない」ため、光源と対象モデルが増えると組み合わせ的にコストが増大してしまいます。このデメリットを軽減するにディファードシェーディングがあります。

原理としては前項で行ったように法線情報を別で出力しておきます。そして特徴的なのは「1 パス目でシェーディング計算を行わない」ことにあります。

1 パス目ではシェーディングを行わない色情報のみを出力し、2 パス目でテクスチャ化した法線情報をもとにシェーディングを行います。このようにシェーディング計算を後から行うために「遅延シェーディング」「ディファードシェーディング」と呼ばれます。

また、1 パス目で出力された「色のみ出力テクスチャ」「法線情報テクスチャ」その他の事を G-Buffer と呼びます。色々なところで耳にする単語なので覚えておきましょう。

ともかく後のパスでこの G-Buffer の内容をもとにシェーディングしたり加工したりしていると思ってください。例えばシェーディングの場合、全てのオブジェクトと光源を計算するよ



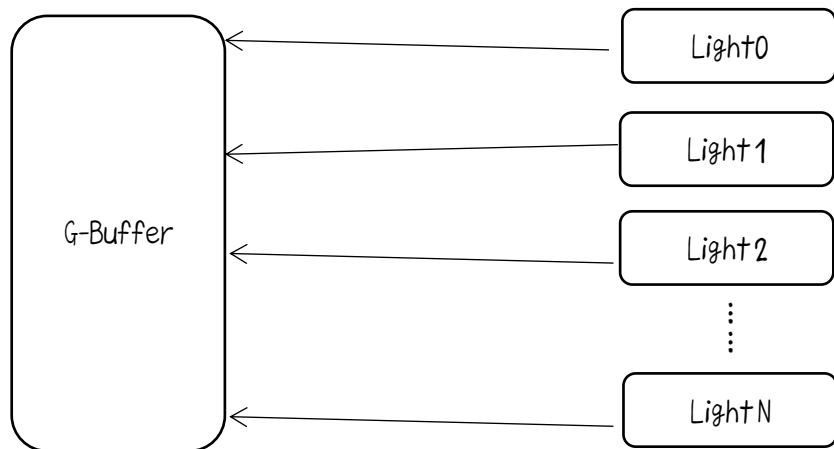
りは1つの出力済みの法線と計算する方がコストがかからないわけです。

普通のシェーディングの場合ですと上の図のように組み合わせが大変ですが
ディファードシェーディングにおいては、上の図のように比較的シンプルに計算できるわけです。

それではまず1パス目のシェーディング計算をいったん無効にしてみましょう。

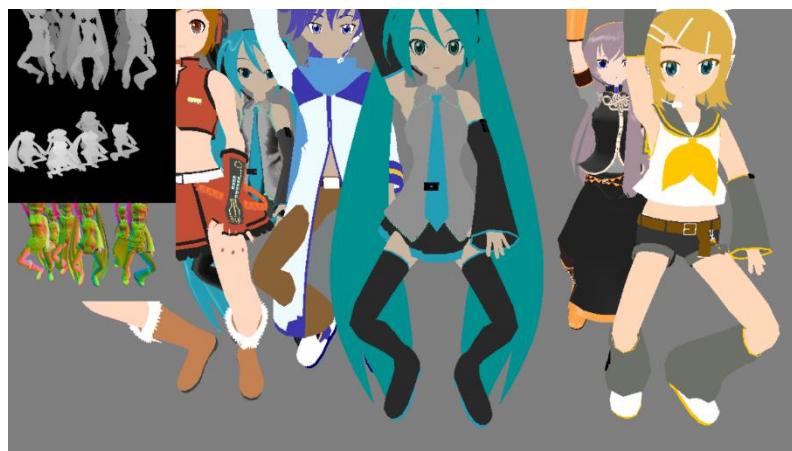
```
PixelOutput output;  
  
output.col = float4(spaCol+sphCol*texCol*diffuse);  
  
output.normal.rgb = float3((input.normal.xyz + 1.0f) / 2.0f);  
  
output.normal.a = 1;  
  
return output;
```

ちなみにこのコードの diffuse は「PMDに設定されているマテリアルの色」であり、シェーディ



ングにおける diffuse 計算とは関係ありませんので注意してください。

この状態で出力すると、シェーディングが行われておらず（光源に関係ない）スフィアマップやテクスチャは適用しています）平坦な感じで出力されます。なお、遅いのが分かりづらくなるためシャドウマップは切っています。



そして2パス目でこの情報をもとにシェーディングを行います
ピクセルシェーダ側で

```

float4 normal=texNormal.Sample(smp, input.uv);
normal = normal * 2.0f - 1.0f;
float3 light = normalize(float3(1.0f, -1.0f, 1.0f));
const float ambient = 0.25f;
float diffB = max(saturate(dot(normal.xyz, -light)), ambient);
return tex.Sample(smp, input.uv)*float4(diffB, diffB, diffB, 1);

```

今回テスト的な実装なので、ライトやアンビエントは臨時のものですが、ひとまずディフューズの計算をペラボリ側(2/4ス目)で行っているのが分かるかと思います。



なおトゥーン等の適用はしていないため、最初の方に実装したような質感になります。ともかく法線とカラーを別々に出力して、後からシェーディングすることができるることはご理解いただけたかなと思います。今回のこれはテスト的な実装ですので、いってん普通の「フォワードシェーディング」に戻して次の話をします。

高輝度成分抽出とぼかしからブルーム(光の漏れ)を実装

ブルームってのは何かというと「ものごつつい明るい部分」の表現のひとつです。とはいってもCGでは「ものごつ明るい」言うても、せいぜい **float4(1,1,1,1)** です。少なくとも SDR ディスプレイならそれ以上は扱えません。

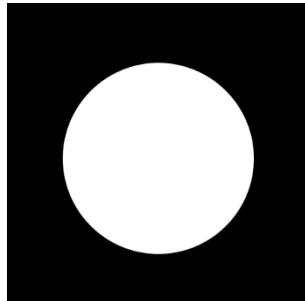
ブルームの理屈

「ものごつつい明るい」表現方法としてすぐに思いつくのは…

- 光が漏れる(あふれだす)
- 周囲を暗くする

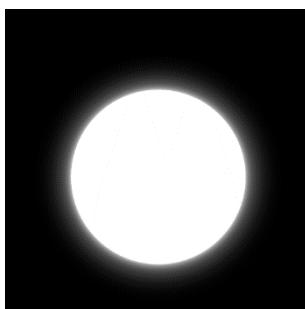
くらいでしょうか…で、1つめをブルームまたはグローと言います。色々と用語があるようですが本書ではブルームとします。

理屈はとても簡単です。例えば「フォトショップ」などの画像加工ソフトで黒背景の真ん中に白い円を描いてみましょう。



このままだと単なる白い円です。これ以上は明るくなりません。

このレイヤーを複製します。そのレイヤーのブレンドモードを「スクリーン(加算)」にしてから「ガウスぼかし」をかけてみましょう。



当然白い部分が最初の円からはみ出します。さらにこのレイヤーを複製してぼかしをかけてみると

このように光が漏れているように見えます。人間はこれを見て「ああ、光が漏れるくらいごつつい明るいんだな」と解釈します。

さて、それではこれから「高輝度」に当たる部分をマルチレンターゲット的に抽出して、その一つをぼかして「ブルーム」するわけですが、一つ問題があります。それは…

「めっちゃ広範囲にぼかす」

必要がある事です。画像加工ソフトでぼかした時に分かったと思いますが半径20ピクセルくらいぼかないとブルームに見えません。しかしながらここまでやったぼかしはせいぜい半径7です。そして当然半径が大きくなればなるほどコストは増大してしまいます。そこで利用するのが

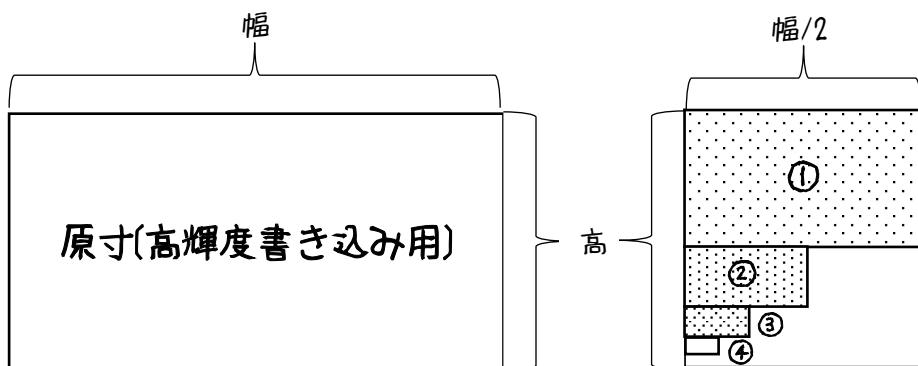
「縮小リッファ」

です。取てて解像度の低いリッファに画像をコピーしてそのうえでボカし、間はバイリニア補間をかけることで多少品質は落ちますが広範囲にボカシをかけることができます。つまり

- ① 解像度を下げる
- ② 低品質ぼかし
- ③ 高解像度に合成する際に勝手にバイリニアかかる(サンプラの指定による)

⇒広い範囲にボカシがかかったのと同じことに!!!!ということです。

また、使用するバッファに関してですが半分→半分と作っていくため



図のようにしていくことになります。つまりバッファとしては原寸のものが1枚と幅を半分にしたものが1枚あれば十分です。原寸の方はマルチレンダーターゲットで出力しましょう。
縮小バッファの方は原寸レンダリング後のパスでビューポートを変更しながら描画します。

ブルームの準備

まずはバッファ変数を用意します。

```
array<ComPtr<ID3D12Resource>, 2> _bloomBuffer; // ブルーム用バッファ
```

そして実際の領域を先ほど解説した仕様で作ります。ここまでできたら細かい説明はしません。

コードは例として示しますが、読者各自の手で実装してください。

```
for (auto& res : _bloomBuffer) {  
    result = _dev->CreateCommittedResource(&heapProp,  
        D3D12_HEAP_FLAG_NONE,  
        &resDesc,  
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,  
        &clearValue,  
        IID_PPV_ARGS(res.ReleaseAndGetAddressOf()));  
    resDesc.Width >>= 1;  
    (略)  
}
```

バッファができるので例によってデスクリプタの数を増やします。

```
// (1つ目RT3枚、2つ目RT1枚)
```

```
heapDesc.NumDescriptors = 4;  
result = _dev->CreateDescriptorHeap(&heapDesc, (略))
```

そしてデスクリプタの3つめにブルーム用の RTV を追加

```
// 3枚目(ブルーム用RTforペラ1)  
_dev->CreateRenderTargetView(_bloomBuffer[0].Get(),
```

```
&rtvDesc, handle);
```

シェーダリソースビュー側も増やしておきます。

```
heapDesc.NumDescriptors = 4;//1~3(ペラ1用)、4(ペラ2用)
```

```
heapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
```

```
//3つ目(ブルームテクスチャ)
```

```
_dev->CreateShaderResourceView(_bloomBuffer[0].Get(),  
&srvDesc,  
handle);
```

バッファとビュー側の準備ができたのでペラ用ルートシグネチャのレンジも増やします

//通常カラー、法線、高輝度(3枚)

```
range[1].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;//t  
range[1].BaseShaderRegister = 0;//0~2  
range[1].NumDescriptors = 3;//t0, t1, t2
```

次に1枚目の出力レンダーターゲット数も増やします。

```
pIsDesc.NumRenderTargets = 3;//レンダーターゲット数  
pIsDesc.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM;  
pIsDesc.RTVFormats[1] = DXGI_FORMAT_R8G8B8A8_UNORM;  
pIsDesc.RTVFormats[2] = DXGI_FORMAT_R8G8B8A8_UNORM;
```

もちろんOMSetRenderTarget部分も書き換えます。

```
CD3DX12_CPU_DESCRIPTOR_HANDLE handles[3];  
D3D12_CPU_DESCRIPTOR_HANDLE baseH=  
    _pераRTVHeap->GetCPUDescriptorHandleForHeapStart();  
auto incSize=  
    _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);  
uint32_t offset = 0;  
for (auto& handle : handles) {  
    handle.InitOffsetted(baseH, offset);  
    offset += incSize;  
}
```

これからも増えかねないため(最大8個)、数が増えても対応できるようにしておきました。

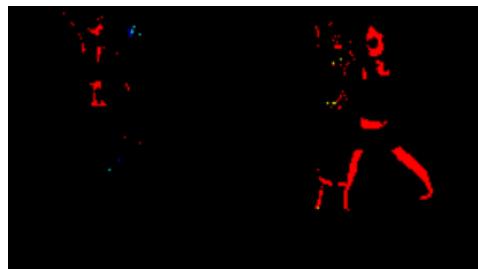
ではまず高輝度部分出力からやってみましょう。ピクセルシェーダ側の出力先を増やします。

```
struct PixelOutput {  
    float4 col:SV_TARGET0://通常のレンダリング結果  
    float4 normal:SV_TARGET1://法線  
    float4 highLum:SV_TARGET2://高輝度(High Luminance)  
};
```

で、高輝度の定義はどうしましょうか。発光物でもない限りブルームを起こさないので、今回のモデルにはどれも発光物がありません。ひとまず最終出力が1.0を超えている部分に色を付けてみます。

```
output.highLum = (ret > 1.0f);
```

↑のコードは最終出力結果(saturate前)が1.0を超えている成分を1.0でそれ以外を0として出力させます。これを見てみると。

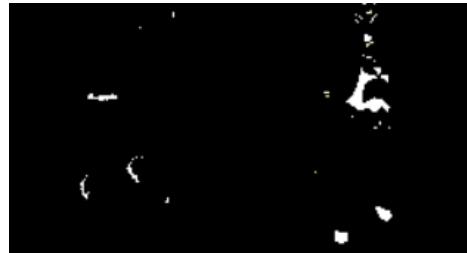


このように赤い部分が目立ちました

これは肌の色が赤寄りであるため1.0を超えたものが一部の髪の毛の色を除いては赤しかなかったことを表しています。このままでは怖い出力になりそうであるためいったん白黒にしてから輝度値を計算してみます。

```
float y = dot(float3(0.299f, 0.587f, 0.114f), output.col);  
output.highLum = y>0.99f? output.col :0.0;
```

そうすると



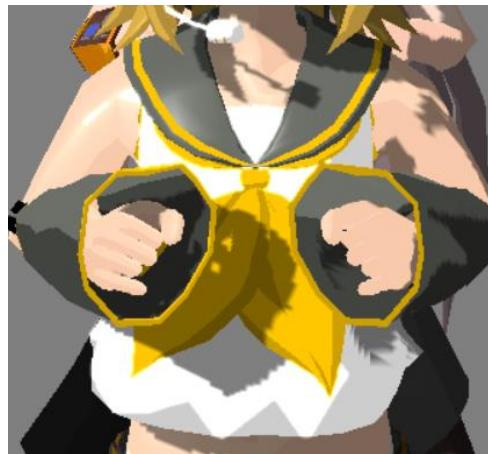
白い部分が抜き出されました

ひとまずこれをを使ってブルームしてみましょう。前に使用した5x5ガウシアンブラーを使ってみます。

```
return tex.Sample(smp, input.uv) +Get5x5GaussianBlur(texHighLum, smp, input.uv, dx, dy);
```

(※なお Get5x5GaussianBlur は 5x5 テーブルでぼかした値を返す自作関数です)

通常出力に高輝度部分をぼかして加算してるだけですがどうなるでしょうか? 高輝度の部分が多め所を見てみてもブルームしているように見えません。かなり大袈裟にブラーしなければならないようです。



では縮小バッファに出力していきましょう。簡単です。ただただぼかしつつ出力していくだけです。ぼかすだけのブルーム用シェーダを作りましょう。

シェーダはピクセルシェーダのみの変更で大丈夫です。そしてブルーム用の Draw 関数 DrawBloom を作りましょう。

縮小バッファへの書き込み

縮小バッファのための領域は既に確保しているためあとは書き込んでいくだけです。まずこのバッファに書き込みそしてテクスチャとして利用できるようにビューを追加します。

//4枚目(ブルーム用縮小バッファ用RT)

```
_dev->CreateRenderTargetView(_bloomBuffer[1].Get(),
    &rtvDesc, handle);
handle.ptr += _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
//中略
//4つ目(高輝度縮小テクスチャ)
_dev->CreateShaderResourceView(_bloomBuffer[1].Get(),
    &srvDesc,
    handle);
handle.ptr +=
    _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
```

次にシェーダを作ります。ぼかして描画するだけのシェーダのため BlurPS という名前にしましょう。

```

//メインテクスチャを 5x5 ガウスでぼかすピクセルシェーダ
float4 BlurPS(Output input) : SV_Target
{
    float w, h, mplevels;
    tex.GetDimensions(0, w, h, mplevels);
    return Get5x5GaussianBlur(tex, smp, input.uv, 1.0/w, 1.0/h);
}

```

(※なお汎用的なシェーダ名になっている理由はこの後の被写界深度にもこのシェーダを使用するからです。)

シェーダを書いたのでパイプラインもボカシ専用のものをつくれます。

```
ComPtr<ID3D12PipelineState> _blurPipeline; //画面全体ぼかし用パイプライン
```

パイプラインの内容は通常のペラ描画と同じでピクセルシェーダが違うだけです。

```

result = D3DCompileFromFile(L"pera.hlsl", nullptr, nullptr, "BlurPS", "ps_5_0",
    0, 0, ps.ReleaseAndGetAddressOf(), errBlob.ReleaseAndGetAddressOf());
gpsDesc_PS = CD3DX12_SHADER_BYTECODE(ps.Get());
result = _dev->CreateGraphicsPipelineState(&gpsDesc,
    IID_PPV_ARGS(_blurPipeline.ReleaseAndGetAddressOf()));

```

(※エラー処理は省略)

そして描画です。ボカシのための縮小バッファ書き込み処理なので

```
void DrawShrinkTextureForBlur();
```

という名前の関数を作ります。この関数内で通常描画後に手元にある高輝度成分テクスチャをビューポートとシザーリミットを縮小しながら描画していきます。ここはDrawShrinkTextureForBlur 関数を一気に書いていきますね。

```

void
Dx12Wrapper::DrawShrinkTextureForBlur() {
    _cmdList->SetPipelineState(_blurPipeline.Get());
    _cmdList->SetGraphicsRootSignature(_peraRS.Get());

    //頂点バッファセット
    _cmdList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
    _cmdList->IASetVertexBuffers(0, 1, &_peraVBV);
}

```

```

//高輝度成分バッファはシェーダリソースに
Barrier(_bloomBuffers[0].Get(),
D3D12_RESOURCE_STATE_RENDER_TARGET,
D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE);

//縮小バッファはレンダーターゲットに
Barrier(_bloomBuffers[1].Get(),
D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
D3D12_RESOURCE_STATE_RENDER_TARGET);

auto rtvHandle= _peraRTVHeap->GetCPUDescriptorHandleForHeapStart();
auto srvHandle = _peraSRVHeap->GetGPUDescriptorHandleForHeapStart();
//4つめに移動
rtvHandle.ptr +=

    _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV) * 3;
//レンダーターゲットセット
_cmdList->OMSetRenderTargets(1, &rtvHandle, false, nullptr);

//テクスチャは1枚目の3つめのレンダーターゲット
srvHandle.ptr +=

    _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV) * 2;
//1パス目高輝度をテクスチャとして使用
_cmdList->SetDescriptorHeaps(1, _peraSRVHeap.GetAddressOf());
_cmdList->SetGraphicsRootDescriptorTable(0, srvHandle);

auto desc = _bloomBuffers[0]->GetDesc();
D3D12_VIEWPORT vp = {};
D3D12_RECT sr = {};
vp.MaxDepth = 1.0f;
vp.MinDepth = 0.0f;
vp.Height = desc.Height/2;
vp.Width = desc.Width/ 2;
sr.top = 0;
sr.left = 0;
sr.right = vp.Width;
sr.bottom = vp.Height;

```

```

for (int i = 0; i < 8; ++i) {
    _cmdList->RSSetViewports(1, &vp);
    _cmdList->RSSetScissorRects(1, &sr);
    _cmdList->DrawInstanced(4, 1, 0, 0);
    //書いたら下にずらして次を書く準備
    sr.top += vp.Height;
    vp.TopLeftX = 0;
    vp.TopLeftY = sr.top;
    //幅も高さも半分に
    vp.Width /= 2;
    vp.Height /= 2;
    sr.bottom = sr.top + vp.Height;
}
//縮小バッファをシェーダリソースに
Barrier(_bloomBuffers[1].Get(),
D3D12_RESOURCE_STATE_RENDER_TARGET,
D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE);
}

ポイントは幅と高さを縮小しつつ一つのバッファに書き込んでいくところですね。これで通常のベラポリ描画で受け取り、ばかして加算します。

```

```

Texture2D<float4> texHighLum : register(t2); //高輝度
Texture2D<float4> texShrinkHighLum : register(t3); //縮小バッファ高輝度

```

この texShrinkHighLum に縮小バッファが入っていますので、全て加算します。

```

float4 bloomAccum = float4(0, 0, 0, 0);
float2 uvSize = float2(1, 0.5);
float2 uvOfst = float2(0, 0);
for (int i = 0; i < 8; ++i) {
    bloomAccum += Get5x5GaussianBlur(texShrinkHighLum, smp, input.uv*uvSize+uvOfst,dx,dy);
    uvOfst.y += uvSize.y;
    uvSize *= 0.5f;
}

```

書き込みの時と同様に UV を縮小に合わせてずらして加算します。ループが終わった時にすべての合計が bloomAccum に入っているはずですので

```

return tex.Sample(smp, input.uv) + //通常テクスチャ
Get5x5GaussianBlur(texHighLum, smp, input.uv, dx, dy) + //1枚目高輝度をぼかし
saturate(bloomAccum); //縮小ぼかし済み

```

合計した値を返してあげれば



このような出力になります。あれ？でも左下に何か光ってますね？これは並べた縮小バッファをぼかしたうえでバイリニア補間がかかっているためこのような結果になってしまっているのです。

例えばガウス関数に

```

float l1 = -dx, l2 = -2 * dx;
float r1 = dx, r2 = 2 * dx;
float u1 = -dy, u2 = -2 * dy;
float d1 = dy, d2 = 2 * dy;
l1 = max(uv.x + l1, rect.x) - uv.x;
l2 = max(uv.x + l2, rect.x) - uv.x;
r1 = min(uv.x + r1, rect.z-dx) - uv.x;
r2 = min(uv.x + r2, rect.z-dx) - uv.x;
u1 = max(uv.y + u1, rect.y) - uv.y;
u2 = max(uv.y + u2, rect.y) - uv.y;
d1 = min(uv.y + d1, rect.w-dy) - uv.y;
d2 = min(uv.y + d2, rect.w-dy) - uv.y;
return float4(
    tex.Sample(smp, uv + float2(l2, u2)).rgb
    + tex.Sample(smp, uv + float2(l1, u2)).rgb*4
    + tex.Sample(smp, uv + float2(0, u2)).rgb*6
    + tex.Sample(smp, uv + float2(r1, u2)).rgb*4

```

```
+ tex.Sample(smp, uv + float2(r2, u2)).rgb
```

(略)

のような処理を入れてもダメで、ダイリニア補間の方が大きな効果が出ているため大した効果はありません。本格的に解消するには影響が出ない範囲のマージンをとるなり別のテクスチャとして出力するなり、ミップマップ化する必要があるでしょう。

ここでは紙面の都合で行いませんが各自実装してください。ひとまず縮小ドッファを別々で作って試しましょう。

深度値からぼかす範囲を決め簡易的な被写界深度を実装

マルチレンダーターゲットとはあまり関係がないのですが、縮小ドッファでのぼかしをやりましたので、被写界深度についてもやっていこうと思います。

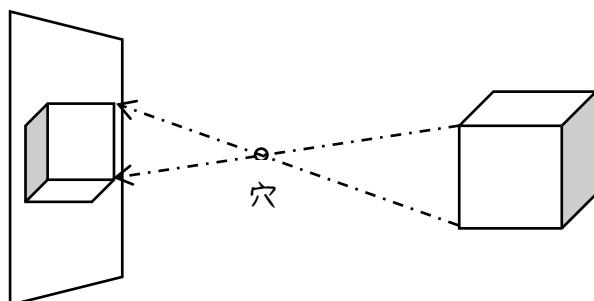
被写界深度について

被写界深度とはピントが合った範囲の事を指します。CGにおいてはピントが合った範囲以外をぼかすことを意味します。

そもそも「ピントが合っている」とはどういうことでしょうか？

それは空間上有る1点から入ってくる光がフィルム上の1つ点で結像する状態の事です。ちなみに理論上光の座標を1点に結像させるピンホールカメラというものがあります。レンズも何も使用せず、紙に針の穴を空けて撮影します。

これは穴が非常に小さい1点なので、外界の光がその穴を通して、中にある感光紙に感光させるしくみなんですが、

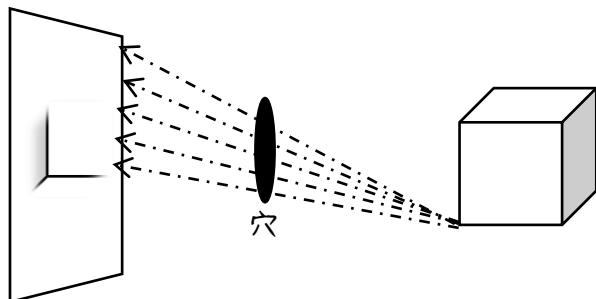


図のように1つの光が1つの点を通ってフィルムに結像するためほとんどボケが発生しません。ちなみに被写体をかなり近づけてしまうと結像が元の大きさ以上になってしまふため、その場合は当然ながらボケてしまいます。

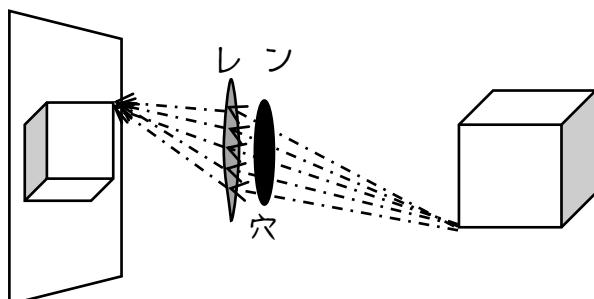
普通の撮影ではそこまで近づけることはめたいため上図のような状態がCGの状態だといえます。CGは普通にレンダリングするとボケませんからね。

(※実際にはピンホールカメラは入ってくる光の量が非常に少ないため十分な感光時間が必要になります。
例えば風景を撮影するために1時間くらい放置します。)

ピンホールカメラでは光量が少なく、撮影までに時間がかかってしまうため、ある程度の広さを持った円で光を受け取り、十分な光量を瞬間に受け取れるようにしたもののが一般的なカメラなのです。



その場合広い範囲から入ってくる光と言うのは放射してしまい、レンズを付さないとひどくボケてしまうためレンズを用いて光が収束するようにしているのです。



これで1点に収束すればよかつたのですが、そうはいかないのです。

例えば虫眼鏡で光を収束させて、何かを燃やそうという時、焦点距離がズレると光が広がってしまうのです。ピッタリ1点に収束したところから近づけても離しても燃えにくくなります。そのピッタリ1点に収束したところが『ピントが合っている』わけです。それ以上離れた部分はピントがずれてる…つまり1点に収束してない→複数の場所の光が1点に合成されてしまう→ボケるということになります。

そしてこのピントが合う範囲のことを被写界深度と言いますが、カメラの絞り(光の量を調節)やレンズが望遠か広角かで、ピントが合っている範囲が変わってくるのです。

CGにおける被写界深度

ここまでやってお分かりのように、CGには被写界深度など存在しません。実は適切な範囲をぼかすことで『それっぽく』見せてるだけなのです。

ではピントが合ってるかどうかってのはどう判断するのでしょうか?先ほど『焦点距離』という用語が出てきましたよね?

そうです。視点からの距離、つまり深度によって決まるのです。

特定の深度を「ピントが合っている」に設定しておき、そこから離れていくにつれぼかしてしまえばいいのです。

プログラムによる実装

実装に入りますがともかくボカシは必要です。それなりに広範囲のボカシが必要ですので、フレームと同じ要領でボカシを作ります。

深度が特定の範囲内であればぼかす。そうでなければぼかさないってことなので、まずボケ画像の用意をしましょう。

ひとまずは既にフレームでボケ画像を作っておりますので、同じように作りましょうか…。今回は高輝度とかではなく普通にぼかしを作りましょう。

```
ComPtr<ID3D12Resource> _dofBuffer; //被写界深度用ぼかしバッファ
```

ちなみに_dofはDepthOfField(被写界深度)の略です。なお、高輝度抽出ではないため1つめのバッファは必要ないです。縮小バッファのみ作ればいいです。

```
Dx12Wrapper::CreateBlurForDOFBuffer() {
```

```
    (略)
```

```
    resDesc.Width >>= 1; //縮小バッファなので大きさ半分でいい
```

```
    HRESULT result = _dev->CreateCommittedResource(&heapProp,
```

```
        D3D12_HEAP_FLAG_NONE,
```

```
        &resDesc,
```

```
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
```

```
        &clearValue,
```

```
        IID_PPV_ARGS(_dofBuffer.ReleaseAndGetAddressOf()));
```

```
    (略)
```

```
}
```

レンダーターゲットとシェーダリソースビューは合わせて各自で増やしてください。このプログラムの例では高輝度縮小、通常縮小の順にしていますので、後ろにくっつける形にしています。

次にフレーム用で使ったDrawShrinkTextureForBlur関数と一緒に縮小バッファを作れるようにしておきます。まずブラー用のパイプラインを通常&高輝度の2レンダーターゲット出力仕様に変更します。

```
gpsDesc.NumRenderTargets = 2; //通常&高輝度  
gpsDesc.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM;  
gpsDesc.RTVFormats[1] = DXGI_FORMAT_R8G8B8A8_UNORM;
```

次に OMSetRenderTarget の指定レンダーターゲットを増やします。

```
auto rtvBaseHandle= _peraRTVHeap->GetCPUDescriptorHandleForHeapStart();
auto rtvIncSize = _dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandles(2) = {};
//4つめ、5つめを使用
rtvHandles(0).InitOffsetted(rtvBaseHandle,rtvIncSize * 3);
rtvHandles(1).InitOffsetted(rtvBaseHandle, rtvIncSize * 4);
//レンダーターゲットセット
_cmdList->OMSetRenderTargets(2, rtvHandles, false, nullptr);
```

2つ出力できるようにしています。先ほども書きましたが、高輝度縮小、通常縮小の順です。次はピクセルシェーダ側です。縮小/ドッファ書き込み部分のシェーダの部分は

```
struct BlurOutput {
    float4 highLum:SV_TARGET0;//高輝度(High Luminance)
    float4 col:SV_TARGET1;//通常のレンダリング結果
};
```

のような構造体を作りおき

```
BlurOutput BlurPS(Output input)
{
    float w,h,miplevels;
    tex.GetDimensions(0, w, h, miplevels);
    float dx = 1.0 / w;
    float dy = 1.0 / h;
    BlurOutput ret;
    ret.col= Get5x5GaussianBlur(tex, smp, input.uv, dx, dy, float4(0, 0, 1, 1));
    ret.highLum= Get5x5GaussianBlur(texHighLum, smp, input.uv, dx, dy, float4(0, 0, 1, 1));
    return ret;
}
```

のように通常と高輝度をそれぞれ返します。このまま縮小/ドッファ書き込みを行えば、高輝度も通常もレンダーターゲットに書き込まれます。

特にエラー等が起きていないければこの時点でペラが使用できるテクスチャが増えているは

ずなのでペラ用ルートシグネチャのレンジをさらに増やします。

```
//通常カラー、法線、高輝度、縮小高輝度、縮小通常(5枚)
range[1].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;//+
range[1].BaseShaderRegister = 0;//0~2
range[1].NumDescriptors = 5;//t0,t1,t2,t3,t4
```

あとはこのぼかされた画像を使用して、距離に応じてぼかしていくたいと思います。どうやって距離でボケている感を出せばいいでしょうか。ボケている画像とボケていない画像の線形補間で対応しようと思います。

ただし線形補間対象が複数になっているため、少し工夫を行います。また、深度値はなかなか変化が出づらいため pow 関数を用いて、近い場所でも変化が出るようにします。

実験的に画面真ん中の深度と比較するようにプログラムすると…

```
//画面真ん中の深度の差を測る
float depthDiff=
    abs(depthTex.Sample(smp, float2(0.5,0.5)) - depthTex.Sample(smp, input.uv));
depthDiff = pow(depthDiff, 0.5f);
uvSize = float2(1, 0.5);
uv0fst = float2(0, 0);
float t = depthDiff*8;
float no;
t = modf(t, no);
float4 retColor(2);
retColor(0)= tex.Sample(smp, input.uv);//通常テクスチャ
if (no == 0.0f) {
    retColor(1)=
        Get5x5GaussianBlur(texShrink, smp, input.uv*uvSize + uv0fst, dx, dy,
        float4(uv0fst, uv0fst + uvSize));
} else{
    for (int i = 1; i <= 8; ++i) {
        if (i - no < 0) continue;
        retColor(i-no)=
```

```

Get5x5GaussianBlur(texShrink, smp, input.uv*uvSize + uv0fst,
dx, dy, float4(uv0fst, uv0fst + uvSize));
uv0fst.y += uvSize.y;
uvSize *= 0.5f;
if (i - no > 1) {
    break;
}
}

return lerp(retColor(0),retColor(1),t);

```

まず depthDiff = pow(depthDiff,0.5f)の部分で近地でも差が出るようにしています。そしてここからがミソですが、

```
float t = depthDiff*8;
```

```
float no;
```

```
t = modf(t, no);
```

この部分もとの depth 値の差は最大 1.0 ですが、複数の縮小ノッファとブレンドできるように 8 倍しています。そして modf で整数部と小数部に分けます。

そしてそのあとのコードで、距離に応じた縮小ノッファを取得。retColor(0)と retColor(1)に代入します。あとは小数部を用いて lerp 補間をかければ、深度値の差でボケるようになります。

フォーカスを画面真ん中深度に合わせてレンダリングすると、距離に応じてボケているのが分かるかと思います。



手前にフォーカスが合っているため後方のキャラがぼけているのが分かります。

スクリーンスペースアンビエントオクルージョン (SSAO)

アンビエントオクルージョンとは

アンビエントオクルージョン(環境遮蔽)とはその名の通り、特定の点において周囲の環境光がどれくらい遮られているのかを表現するものです。下の図のように面が密接するところが



暗くなります。

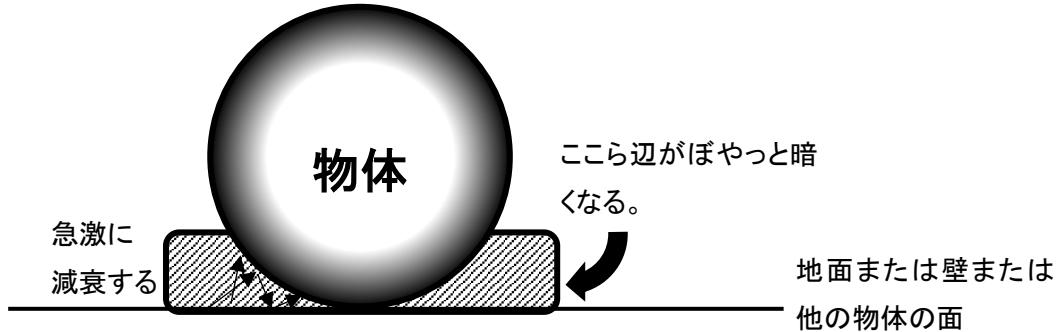
シワやすきまの中に光が吸い込まれていくような表現をすることでモデルの「実在感」を高めます。キャラクタに使うと、いかにも CG といった質感からフィギュアのような質感になります。

環境光と言うのは今まで「暗くなりすぎないための底上げ」として考えていましたが、本来はその名の通り周囲の物体に反射した光がさらに反射して無数に反射を繰り返しそこら中に光をばらまいた結果、環境の明るさを構築していると思ってください。

無数に反射を繰り返すって事もよく考えてください。完全な鏡面反射でもない限り、光と言うのは減衰します(実際には鏡面反射でも空気中で減衰してしまいます)。そして物体に吸収されなかつた光だけが反射するわけです。ともかく反射を繰り返すと光が弱くなると考えてください。

環境についてですが、直接光が当たってなくても、それなりの明るさがあると、あの光は満遍なく届くかと言うとそうではないんですね。たとえば狭くなっている部分は、

反射回数が多くなる…つまり、かなり早く光が弱くなるという事です。



この「狭い隙間が暗くなっている」表現するのがアンビエントオクルージョンです。

とりあえず今はイメージでいいんですが、大体掴めますかね？直接光の影じゃなくても狭い溝とかそういうのって暗いですよね？人体で言うと脇とか膝の裏とかがだいたいぼやーっと暗くなってるんですよ。これを人体でやるとなんというかフィギュア的な質感が出てきます。私はこれを「実在感が出る」と呼んでいます。

これを計算(プログラム)で表現するにはどうすればいいのでしょうか？

数式から考えるアンビエントオクルージョン

まずアンビエントオクルージョンと言えば以下のような式で表されます。

$$A_p = \frac{1}{\pi} \int_{\Omega} V(x_p, \vec{\omega}) \cos \theta d\omega = \frac{1}{\pi} \int_0^{2\pi} \int_0^{\frac{\pi}{2}} V(x_p, \vec{\omega}) \cos \theta \sin \theta d\theta d\varphi$$

この式を見て最初に思い浮かぶのは「なぜπで割っているんだろう」という事ですが、それも含めてお話ししていきます。かなり数学のウェイトが大きい話ですので、耐えられなればこの項は飛ばしていただいて構いません。

数式の話をする前に分かっておいてほしいことがあります。それは学校の試験ではないのですから数式を展開しなくてもいいし積分の計算をしなくてもいいのです。数式が意味するところをイメージ出来ればそれでいいのです。

実際のプログラムになったときには数式とは違う実装になっていると思います。それでも数式の話をするのは、CGの書籍や論文が数式の形で説明されることが多いからです。数式から目を背けていては新しい技術にアクセスすることができないのです。

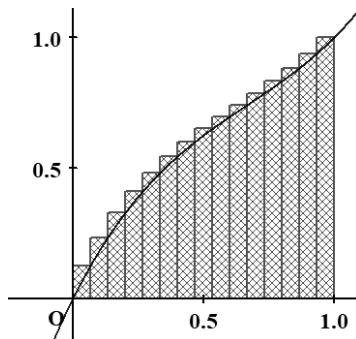
中には「を見た瞬間にアレルギーを起こしてしまう人もいるかもしれない」ですがここまで読んだなら腹をくくりましょう。ともかく「 \int 」は Σ と同じ意味で「特定の範囲内の値の総和」という意味です。

同じ意味ならばなぜ違う記号なのでしょうか?ひとまずは大雑把に、 Σ が総和する世界は「整数の世界(デジタル?)」で、 \int が総和する範囲は「実数の世界(アナログ?)」てな感じに理解しておけばいいです。

$$\sum_{k=0}^n a_k$$

例えばこの式は k が $0 \rightarrow n$ まで変化していきます。 k の変化は $0, 1, 2, 3, 4, 5, \dots$ のように整数で増えていく。この時の a_k を全て足すという意味です。

ところがこのやり方だと、アナログな変化をする数式には対応できないのです。下図のように本来の総和と比べるとガタガタの結果になります。



一定の数で割ってもいいのですが、結局整数で表せる限界に囚われます。

ところが実際の自然現象はアナログな事ばかりなわけ、そこで「極限の考え方」が編み出され、アナログ的に全て加算しようというのが「積分」つまり \int と言う訳です。

CGとコンピュータ。デジタルの権化のようなものです。であればforループを回して総和を求めれば簡単ですし範囲も「有限」です。

これに対して現実の自然現象としての光の総和を考えるときは \int で考える必要があります。

このため先ほどのアンビエントオクルージョンの数式は積分の式で表しているのです。

しかし \int はアナログですので範囲じたいが有限であっても加算対象は「無限」となってしまいます。解ける数式なら手計算で解いて結果を見てもいいのですが

$$A_p = \frac{1}{\pi} \int_{\Omega} V(x_p, \vec{\omega}) \cos\theta d\omega = \frac{1}{\pi} \int_0^{2\pi} \int_0^{\frac{\pi}{2}} V(x_p, \vec{\omega}) \cos\theta \sin\theta d\theta d\phi$$

$V(x_p, \vec{\omega})$ の項はサンプル点の状況によって変化するので手計算で解くことは難しいのです。

$V(x_p, \vec{\omega})$ が意味するところは「特定の方向の光が遮蔽されているか、されてないか」の true/false の値です。方向によって1だったり0だったりすると思ってください。

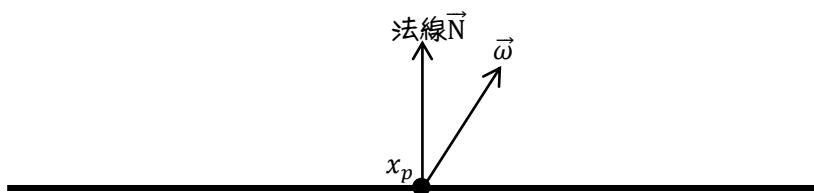
この方向を「総和」するのですが、実際に空間を見ながらでないと値は分かりません。つまり「この数式は解く必要がない」のです。高校教育までだと「解かなきやいけない」気になりますが、解こうとしなくていいです。

この数式は解かれることが期待しているのではなく…前にも述べたように「数式が言つてい

る事を理解してイメージされることを期待している」わけです。

ではこの数式の言わんとすることはどのような事なのでしょう。

まず $V(x_p, \vec{\omega})$ についてですが、これはに対して「点 x_p に対して $\vec{\omega}$ 方向からの環境光が届いてるかどうか」を yes(1) か no(0) で答えるものだそうです。私は「遮蔽されているかどうか」で yes(1) で no(0) にしますが、結局は同じことになります。遮蔽率であるため、例えば片方の率を t とするともう片方は $1.0 - t$ となり確定するため、どちらで考えても同じことです。



ちなみに $\vec{\omega}$ を見ると角度変化量を表している気になる読者もいると思いますが、これは対象点から任意の方向へ向かうベクトルを表しているだけです。 $\vec{\omega}$ は単なるベクトルです。

とはいえてそんな関数どうやって作ったらいいのでしょうか? どうやって光が遮蔽されているのか計算したらいいのか? とお思いかもしれません、数式はある意味ただの「言葉」です。計算は(まだ)しくていいのです。

とにかく $V(x_p, \vec{\omega})$ の意味するところが $\vec{\omega}$ 方向が遮蔽されているかどうかを調べるって事を意味してると思ってくれればいいです。

じゃあ次に後ろの $\cos \theta$ についてだけど、この $\cos \theta$ は単にランパートの余弦則の $\cos \theta$ だ。環境光とは言え、真正面から光が当たれば最も明るくなるし、真横から当たれば明るくない。そう言う意味の $\cos \theta$ であり、 θ は当然のように法線ベクトル N と適当なベクトル ω の間の角度だ。

最後に $\int_{\Omega} \dots d\omega$ の部分ですが、ちょっとサイトに書いてない情報ですがこの手の数式で Ω が出きたらこれは半球面を表していると思ってください。ちなみにこの場合 Ω (オーム)ではなく Ω (オメガ)って読みになる事が多いですね。ややこしい。ちなみに Hemisphere の略で H もよく使われます。全球なら S(Sphere) が使われます。

ともかくベクトルを半球面全てに飛ばし、その遮蔽率を計算。そしてその際にランパートを考慮して $\cos \theta$ が乗算されているのが分かります。とりあえずこの事を理解すればプログラムは書くことができます。

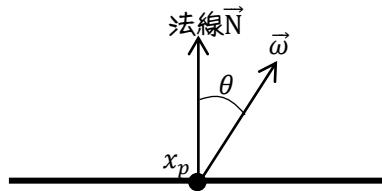
これで終わりでもいいんですが、もうちょっと数学的に続きをやってましょう。

疑問が残るのが「π」で割っている部分の事です。なぜπで割っているのでしょうか？結論から言いますと、これは結果を「正規化」するために割っているのです。

半球の表面積を知っている人は「あれ？どうせ割るなら、表面積の $4\pi r^2$ を2で割って $2\pi r^2$ のような気がしますが、式をよく見てみましょう。

$$A_p = \frac{1}{\pi} \int_{\Omega} V(x_p, \vec{\omega}) \cos\theta d\omega$$

$\cos\theta$ の項がありますね。これは何ベクトルと何ベクトルとの $\cos\theta$ かというと、



法線 \vec{N} ベクトルと任意の方向 $\vec{\omega}$ ベクトルのなす角度の θ です。特定の点にあらゆる方向から光が当たりますが、その微小な部分においてもランダートの余弦則が働くためです(本当は放射照度等の理由によるのですが、今はそう理解しておいてください)。この項があることを前提に積分していきます。

なお、今は半球面を表しており半球面全ての方向における $V(x_p, \vec{\omega}) \cos\theta$ 積分をするわけです。

このとき積分しつつ変化していくのは $\vec{\omega}$ のみです。それがありとあらゆる方向の結果を総和しますので

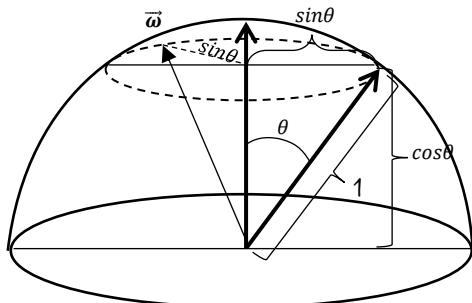
$$A_p = \frac{1}{\pi} \int_{\Omega} V(x_p, \vec{\omega}) \cos\theta d\omega = \frac{1}{\pi} \int_0^{2\pi} \int_0^{\frac{\pi}{2}} V(x_p, \vec{\omega}) \cos\theta \sin\theta d\theta d\varphi$$

となります。

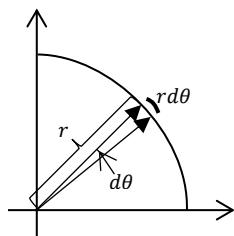
まずは何故 $\frac{1}{\pi} \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \sin\theta d\theta d\varphi$ の式が半球を表しているかの話を指定します。

式の中に急に出てきた $\sin\theta$ は図形的にどこを示しているのでしょうか。

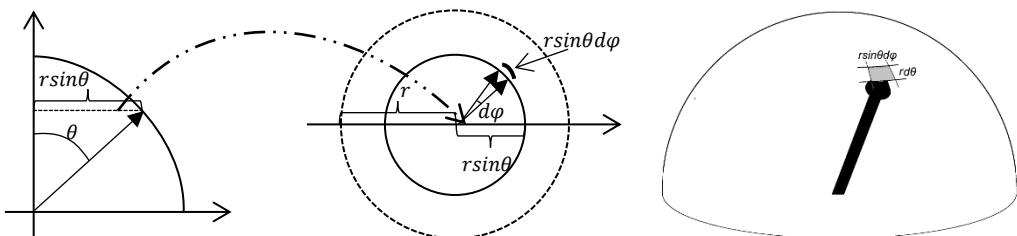
次の図を見てください



このように縦方向は θ で変化し、その後で法線軸方向に1回転(φ 回転)します。積分における $d\theta, d\varphi$ は角度の微小変化を表しています。まず $d\theta$ の事を考えます。



角度が微小角度 $d\theta$ 変化したときの弧の長さは $rd\theta$ となります。しかし弧の長さはあくまでも弧の長さに過ぎません。これを面積にするには直交する方向の微小変化幅が必要ですが、幅はどうなっているのでしょうか？今度は微小角度変化は $d\varphi$ です。ですので弧の大きさは θ の時と同様に $rd\varphi$ としたい所なのですが、そうはいきません。



ご覧のように角度によって円の半径が変わります。そしてこの場合円の半径は r ではなく $rsin \theta$ となります。微小の世界においても $rsin \theta \leq r$ なのです。そのことを踏まえたうえで微小面積を考えると微小面積 $dS = rsin \theta d\varphi r d\theta$ となります。これを半球全てに積分しますので

$$S = \int_{\Omega} dS = \int_0^{2\pi} \int_0^{\frac{\pi}{2}} r^2 sin \theta d\theta d\varphi$$

となって、これは確かに $S = \frac{4\pi r^2}{2} = 2\pi r^2$ なのですが、少し前に書いたように放射照度(ランバートの \cos)の項が入ってくるため実際の式は

$$L_A = \int_0^{2\pi} \int_0^{\frac{\pi}{2}} r^2 sin \theta cos \theta d\theta d\varphi = r^2 \int_0^{2\pi} \int_0^{\frac{\pi}{2}} sin \theta cos \theta d\theta d\varphi$$

となります。これを展開する事を考えます。 sin と cos がそのままでは計算しづらいため

$$sin(2\theta) = sin(\theta + \theta) = sin\theta cos\theta + sin\theta cos\theta = 2sin\theta cos\theta$$

を逆に利用します。

$$L_A = r^2 \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \frac{\sin(2\theta)}{2} d\theta d\varphi = \frac{r^2}{2} \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \sin(2\theta) d\theta d\varphi$$

と、変形します。なお \sin の積分は

$$\int_a^b \sin(nx) dx = \frac{1}{n} [-\cos nx]_a^b = \frac{1}{n} (-\cos b + \cos a)$$

の法則が成り立ちますので

$$\frac{r^2}{2} \int_0^{2\pi} \frac{1}{2} [-\cos 2\theta]_0^{\frac{\pi}{2}} d\varphi = \frac{r^2}{4} \int_0^{2\pi} -\cos\left(\frac{2\pi}{2}\right) + \cos(0) d\varphi = \frac{r^2}{4} \int_0^{2\pi} 2d\varphi$$

そして残った積分を計算します。

$$\frac{r^2}{4} \int_0^{2\pi} 2d\varphi = \frac{r^2}{4} [2\varphi]_0^{2\pi} = \frac{r^2}{4} 4\pi = \pi r^2$$

ここで半径 $r = 1$ だとすると積分の結果は π となります。

つまり アンビエントオクルージョンの式を π で割っている理由は $V(x_p, \bar{\omega})$ の結果が全て true だった場合の値を全て足すと π になるから なのです。

実際には溝や隙間の部分ではある程度の確率で `false` になります。この確率 = 環境光遮蔽率 ということです。

実際のプログラムにおいては別に π で割る必要はなく、全試行中のいくつが遮蔽されているかを測るためのプログラムになります。

スクリーンスペースアンビエントオクルージョン(SSAO)の実装

概要

前項までの計算でプログラムとしての手順を考えてみます。そうすると

- ①ある点から半径 r の半球面に対して遮蔽されてるかどうかを判定する
- ②判定の結果に $\cos\theta$ を乗算する
- ③②の総和を求める
- ④正規化する(判定が全て `true` だった時の結果で割る)

という手順で考えればよいことが分かります。

このように言葉で箇条書きにすると簡単に思えますが、レイトレーシングのような処理を行う事となりかなり処理不可が高くなります。

そのため比較的処理不可を抑えた「スクリーンスペースアンビエントオクルージョン(SSAO)」を実装していくこうと思います。

スクリーンスペースアンビエントオクルージョン(SSAO)とは「見える範囲だけを考慮する」アンビエントオクルージョンです。なお他にも「スクリーンスペース」という名前の手法を見つけたらそれも「見えている範囲だけが対象である」と覚えておきましょう。これ以降は文字数節約のために「SSAO」と表記します。

SSAO の理屈

「見えている範囲」と強調しましたがどういうことを意味しているのでしょうか。それは「11番目のレンダリング結果」(深度、法線ベクトル)を用いて遮蔽されているのかどうかをチェックします。

大雑把に説明すると現在の UV 値における元の座標を復元し、予め決めておいた試行回数だけ遮蔽を計算し

- ①逆 Projection 行列を定数バッファから得る
- ②UV を XY 座標に、深度値を Z 座標に変換し、元座標を復元
- ③以下をサンプル数(試行回数)だけ繰り返す

- I ランダム方向(大きさは固定)にベクトルを飛ばす
 - II $x_p + \vec{o}$ (現在の場所から指定ベクトルだけ変位した座標)が遮蔽されているかどうか調査
 - III $\vec{N} \cdot \vec{o}$ つまり $\cos\theta$ を計算
 - IV IIIの結果を総和変数 A に加算、IIの結果遮蔽されたいたら総和変数 B にも加算
- ④遮蔽率 = 総和変数 B / 総和変数 A として計算

これで求まった遮蔽率が高ければほど遮蔽されているため、 $(1.0 - \text{遮蔽率})$ を色に乗算すれば環境遮蔽を実装できそうです。

準備(ランダムテクスチャを作成)

残念ながらピクセルシェーダにはランダムの関数がありませんが、乱数を発生させることはできます。理屈を説明するよりもコードを見ていただいたほうが早いと思います。

```
//現在のUV値を元に乱数を返す
float random(float2 uv) {
    return frac(sin(dot(uv, float2(12.9898, 78.233)))*43758.5453);
}
```

この関数が乱数を生み出します。`frac` は小数部を返す関数です。それに対して uv 値と固定値の間の内積をラジアンとして `sin` をとりその値を定数倍したものを内積として使います。これで本当にランダム値になるの?と思いますが理屈があってランダムになりますので解説は数学の項で行いますので、ともかく使って乱数を発生させていきましょう。

アンビエントオクルージョンまで一気に実装

実装時はアンビエントオクルージョンのパスは最終パスとは別にしようと思います。本書ではアンビエントオクルージョン単体でレンダリングし、乗算を行う事にします。

そのために必要なのは

- アンビエントオクルージョン用バッファ
- アンビエントオクルージョン用ビュー
- アンビエントオクルージョン用シェーダ
- アンビエントオクルージョン用パイプライン
- アンビエントオクルージョン用描画処理

ですので早速作っていきましょう。

```
ComPtr<ID3D12Resource> _aoBuffer;
ComPtr<ID3D12PipelineState> _aoPipeline;
bool CreateAmbientOcclusionBuffer();
void DrawAmbientOcclusion();
```

を宣言しておいていつもの手順でアンビエントオクルージョン用バッファを作りましょう。

```
bool Dx12Wrapper::CreateAmbientOcclusionBuffer() {
    auto& bbuff = _backBuffers[0];
    auto resDesc = bbuff->GetDesc();
    resDesc.Format = DXGI_FORMAT_R32_FLOAT;
    D3D12_HEAP_PROPERTIES heapProp = CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT);
    D3D12_CLEAR_VALUE clearValue = {};
    clearValue.Color[0] = clearValue.Color[1] = clearValue.Color[2] =
        clearValue.Color[3] = 1.0f;
    clearValue.Format = resDesc.Format;
    HRESULT result = S_OK;
    result = _dev->CreateCommittedResource(&heapProp,
        D3D12_HEAP_FLAG_NONE,
        &resDesc,
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
```

```

    &clearValue,
    IID_PPV_ARGS(_aoBuffer.ReleaseAndGetAddressOf()));

if (!CheckResult(result)) {
    assert(0);
    return false;
}

return true;
}

```

アンビエントオクルージョンにおいて必要なのは乗算用の明度情報のみですので、R32_FLOATで確保しておきます。

次にレンダーターゲットビューとシェーダリソースビューですが、いつものように後ろに追加すると管理が分かりづらくなると感じたため、アンビエントオクルージョン用のデスクリプタヒープ(RTVもSRVも両方とも作りましょう)

宣言

```

ComPtr<ID3D12DescriptorHeap> _aoRTVDH;
ComPtr<ID3D12DescriptorHeap> _aoSRVDH;

```

生成

```

bool Dx12Wrapper::CreateAmbientOcclusionDescriptorHeap() {
    //RTV 用ヒープ作成
    D3D12_DESCRIPTOR_HEAP_DESC desc = {};
    desc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    desc.NodeMask = 0;
    desc.NumDescriptors = 1;
    desc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
    auto result = _dev->CreateDescriptorHeap(&desc,
        IID_PPV_ARGS(_aoRTVDH.ReleaseAndGetAddressOf()));

    //RTV 作成
    D3D12_RENDER_TARGET_VIEW_DESC rtvDesc = {};
    rtvDesc.ViewDimension = D3D12_RTV_DIMENSION_TEXTURE2D;
    rtvDesc.Format = DXGI_FORMAT_R32_FLOAT;
    _dev->CreateRenderTargetView(_aoBuffer.Get(), &rtvDesc,
        _aoRTVDH->GetCPUDescriptorHandleForHeapStart());
}

```

```

//SRV 用ヒープ作成
desc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
desc.NodeMask = 0;
desc.NumDescriptors = 1;
desc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
result = _dev->CreateDescriptorHeap(&desc,
    IID_PPV_ARGS(_aoSRVDH.ReleaseAndGetAddressOf()));

//SRV 作成
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Format = DXGI_FORMAT_R32_FLOAT;
srvDesc.Shader4ComponentMapping =
    D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Texture2D.MipLevels = 1;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
_dev->CreateShaderResourceView(_aoBuffer.Get(), &srvDesc,
    _aoSRVDH->GetCPUDescriptorHandleForHeapStart());

return true;
}

```

として作っておいてあげます。

次にシェーダ側ですが、いったん形だけ作っておきましょう。

//SSAO 処理のためだけのシェーダ

```
Texture2D<float4> normtex:register(t1); //1 パス目の法線描画
```

```
Texture2D<float> depthtex:register(t6); //1 パス目の深度テクスチャ
```

```
SamplerState smp:register(s0);
```

//SSAO(乗算用の明度のみ情報を返せばよい)

```
float SsaoPs(Output input) : SV_Target
```

```
{
```

```
    return 1.0f;
```

```
}
```

今回は明度のみを扱う float 値を返します。必要なのは法線情報と深度情報のみです。このサンプルコードでは +1 番と +6 番に合わせていますが、読者のプログラムの状況に合わせて番号は調整してください。

シェーダプログラムを用意しましたのでパイプラインを作りましょう。

//SSAO 用

```
result = D3DCompileFromFile(L"ssao.hlsl", nullptr, nullptr, "SsaoPs", "ps_5_0", 0, 0,
    ps.ReleaseAndGetAddressOf(), errBlob.ReleaseAndGetAddressOf());
```

(エラー処理は省略)

```
gpsDesc.NumRenderTargets = 1;
```

```
gpsDesc.RTVFormats[0] = DXGI_FORMAT_R32_FLOAT;
```

```
gpsDesc.RTVFormats[1] = DXGI_FORMAT_UNKNOWN;
```

```
gpsDesc.BlendState.RenderTarget[0].BlendEnable = false;
```

```
gpsDesc.PS = CD3DX12_SHADER_BYTECODE(ps.Get());
```

```
result = _dev->CreateGraphicsPipelineState(&gpsDesc,
```

```
IID_PPV_ARGS(_aoPipeline.ReleaseAndGetAddressOf()));
```

(エラー処理は省略)

注意すべき点として、SSAO の書き込み先の BlendEnable は false にしておいてください。R32_FLOAT なのに BlendEnable=true にしてしまうと、 $\alpha=0$ でブレンダリングされてしまい、せっかくのアンビエントオクルージョンの結果が見えなくなってしまいます。

この状態で描画処理を行います。今回の関数は SSAO 用のバッファに書き込みます。なお、必ず 1 パス目レンダリングの後に処理を行ってください。以下が DrawAmbientOcculusion 関数です。

```
void
```

```
Dx12Wrapper::DrawAmbientOcculusion() {
```

(バリア処理は省略しています)

```
auto rtvBaseHandle = _aoRTVDH->GetCPUDescriptorHandleForHeapStart();
```

```
_cmdList->OMSetRenderTarget(1, &rtvBaseHandle, false, nullptr);
```

```
_cmdList->SetGraphicsRootSignature(_peraRS.Get());
```

```
auto wsize = Application::Instance().GetWindowSize();
```

```
D3D12_VIEWPORT vp = CD3DX12_VIEWPORT(0.0f, 0.0f, wsize.width, wsize.height);
```

```
_cmdList->RSSetViewports(1, &vp); // ビューポート
```

```
CD3DX12_RECT rc(0, 0, wsize.width, wsize.height);
```

```
_cmdList->RSSetScissorRects(1, &rc); // シザーリング
```

```
_cmdList->SetDescriptorHeaps(1, _peraSRVHeap.GetAddressOf());
```

```
auto srvHandle = _peraSRVHeap->GetGPUDescriptorHandleForHeapStart(); // 法線テクスチャのため
```

```

    _cmdList->SetGraphicsRootDescriptorTable(1, srvHandle);

    _cmdList->SetDescriptorHeaps(1, _depthSRVHeap.GetAddressOf());
    auto srvDSVHandle = _depthSRVHeap->GetGPUDescriptorHandleForHeapStart(); //深度テクスチャのため
    _cmdList->SetGraphicsRootDescriptorTable(3, srvDSVHandle);

    _cmdList->SetDescriptorHeaps(1, _sceneHeap.GetAddressOf());
    auto sceneHandle = _sceneHeap->GetGPUDescriptorHandleForHeapStart();
    _cmdList->SetGraphicsRootDescriptorTable(5, sceneHandle);

    _cmdList->SetPipelineState(_aoPipeline.Get());
    _cmdList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
    _cmdList->IASetVertexBuffers(0, 1, &_peraVBV);
    _cmdList->DrawInstanced(4, 1, 0, 0);
    (バリア処理は省略しています)

}

```

なお、SSAOの処理を行う際に座標変換の処理が必要になってきますので、シーンの情報(ビュー行列やプロジェクション行列)をヒープにセットして渡しておくのを忘れないでください。
さて、この結果はアンビエントオクルージョン/マップに出力されているはずですので、最終マップへのドロー時にこのアンビエントオクルージョンをテクスチャとして利用します。

最終描画時にアンビエントオクルージョン用のリソースバインディングを行う

```
//SSAOテクスチャ
_cmdList->SetDescriptorHeaps(1, _aoSRVDH.GetAddressOf());
_cmdList->SetGraphicsRootDescriptorTable(4,
    _aoSRVDH->GetGPUDescriptorHandleForHeapStart());

```

今回はルートパラメータは4番に割り当てていますが、読者のプログラムの状況によって番号は調整してください。この書籍のサンプルとしては+8としてルートパラメータ4で設定しています。

次にペラシェーダ側にアンビエントオクルージョンの結果が見られるように

```
Texture2D<float> texSSAO : register(t8); //SSAO
を追加し、
```

```
(略)

else if (input.uv.x < 0.2&&input.uv.y < 0.8) { //AO
    float s=texSSAO.Sample(smp, (input.uv - float2(0, 0.6)) * 5);
    return float4(s, s, s, 1);
}
```

を追加しましょう。左下に真っ白のテクスチャが出ていれば正解です。

そこまでがうまくいいたら、いよいよアンビエントオクルージョンシェーダを作成します。ピクセルシェーダだけでよくピクセルシェーダ関数名は SsaoPs とします。

まず、1パス目の結果を取得する受け皿を用意します。

```
//SSAO処理のためだけのシェーダ
Texture2D<float4> normtex:register(t1); //1パス目の法線描画
Texture2D<float> depthtex:register(t6); //1パス目の深度テクスチャ
```

そしてこのシェーダ内では元座標を復元する必要があるため「逆プロジェクション行列」を受け取れるようにしておきます。元から CPP 側にあるシーン構造体に逆行列用の変数を追加しておきましょう。

```
struct SceneMatrix {
    XMATRIX view;//ビュー
    XMATRIX proj;//プロジェクション
XMATRIX invProj;//逆プロジェクション
    XMATRIX lightCamera;//ライトから見たビュー
    XMATRIX shadow;//影行列
    XMFLOAT3 eye;//視点
};
```

中身はとっても簡単。XMMatrixInverse でプロジェクションの逆行列を作って invProj に代入するだけです。

```
XMVECTOR det;
_mappedScene->invproj = XMMATRIXInverse(&det, _mappedScene->proj);
```

なお、変数 det には「固有値」が入ります。もし逆行列が得られない場合には det がゼロになります。今回は気にしなくていいです。nullptr でも構いません。

これでシェーダ側で逆行列が使えるようになるので、以下の構造体をシェーダ側に書いておいて利用できるようにしておきましょう。

```
cbuffer sceneBuffer : register(b1) {
    matrix view;//ビュー
    matrix proj;//プロジェクション
    matrix invproj;//逆プロジェクション
    matrix lightCamera;//ライトビュープロジェ
    matrix shadow;//影行列
    float3 eye;//視点
};
```

余計なものまで混じっていますが、元の構造体とのつじつまを合わせるために残しています。

いよいよ SsaoPs を記述していきましょう。数学の頃でお話しした内容を思い出しながらコードを見てください。

```
//SSAO(乗算用の明度のみ情報を返せればよい)
float SsaoPs(Output input) : SV_Target
{
    float dp = depthtex.Sample(smp, input.uv);//現在の UV の深度

    float w, h, mplevels;
    depthtex.GetDimensions(0, w, h, mplevels);
    float dx = 1.0f / w;
    float dy = 1.0f / h;

    //SSAO
    //元の座標を復元する
    float4 respos = mul(invproj, float4(input.uv * float2(2, -2) + float2(-1, 1), dp, 1));
    respos.xyz = respos.xyz / respos.w;
    float div = 0.0f;
    float ao = 0.0f;
    float3 norm = normalize((normtex.Sample(smp, input.uv).xyz * 2) - 1);
    const int trycnt = 256;
```

```

const float radius = 0.5f;
if (dp < 1.0f) {
    for (int i = 0; i < trycnt; ++i) {
        float rnd1 = random(float2(i*dx, i*dy)) * 2 - 1;
        float rnd2 = random(float2(rnd1, i*dy)) * 2 - 1;
        float rnd3 = random(float2(rnd2, rnd1)) * 2 - 1;
        float3 omega = normalize(float3(rnd1,rnd2,rnd3));
        omega = normalize(omega);
        //乱数の結果法線の反対側に向いてたら反転する
        float dt = dot(norm, omega);
        float sgn = sign(dt);
        omega *= sign(dt);
        //結果の座標を再び射影変換する
        float4 rpos = mul(proj, float4(respos.xyz + omega * radius, 1));
        rpos.xyz /= rpos.w;
        dt *= sgn;//正の値にして cos θ を得る
        div += dt;//遮蔽を考えない結果を加算する
        //計算結果が現在の場所の深度より奥に入ってるなら遮蔽されているので加算
        ao += step(depthtex.Sample(smp,
            (rpos.xy + float2(1, -1))*float2(0.5f, -0.5f)), rpos.z)*dt;
    }
    ao /= div;
}
return 1.0f - ao;
}

```

まず元の座標を復元するには uv 値と深度に対して逆プロジェクションを乗算してやります。

```

float4 respos = mul(invproj, float4(input.uv*float2(2, -2) + float2(-1, 1), dp, 1));
respos.xyz = respos.xyz / respos.w;

```

最後に w で割るのを忘れないでください。これで元座標の復元ができます。次にテクスチャ化している法線ベクトルも復元しますが、これはおなじみ 2 倍して 1 を引くだけです。

```

float3 norm = normalize((normtex.Sample(smp, input.uv).xyz * 2) - 1);

```

ここからが $\frac{1}{\pi} \int_{\Omega} V(x_p, \vec{\omega}) \cos \theta d\omega$ 式の本体部分です。for ループと乱数を使用して実装しています。trycnt が試行回数で、radius は調査のためのベクトル $\vec{\omega}$ の長さです。乱数と言っても 3D ベクトルのため 3 つ必要です。なお random 関数は 0.0~1.0 の範囲であるためテクスチ

ヤ化した法線ベクトルと同様に 2 をかけて 1 を引いてベクトルを作っています。

(※ここは事前にランダムテクスチャを作つておいて、それを参照するだけの方が速いと思います)

次が工夫ポイントですが、乱数ベクトルなので本当にどこに向くのか分かりません。しかしそれでは困ります。法線方向を中心とした半球になつてもらう必要があります。

そのため法線ベクトルとランダムベクトルの内積を取り、マイナス値ならランダムベクトルを反転します。これで強制的に法線を中心とした半球内に収まるようになります。

//乱数の結果法線の反対側に向いてたら反転する

```
float dt = dot(norm, omega);
float sgn = sign(dt);
omega *= sign(dt);
```

なお、`sign` 関数は「符号を取得する」関数です。これを乗算することによって無理やり法線の半球内に持って行つているのです。

次に遮蔽を考えずに乱数を飛ばした結果の総和を計算しておきます。これは最後に遮蔽計算した総和から割るためです。数式でいうと π の部分に当たります。

```
dt *= sgn;//正の値にして  $\cos \theta$  を得る
div += dt;//遮蔽を考えない結果を加算する
```

dt はもともとの内積値ですが、マイナスの可能性があるため符号を乗算して正の値にしておきます。そしてこれが $\cos \theta$ 項の役割も持ちます。`div` 変数にためておき総和として後で使用します。

これだけでは終わりません。現在の座標に得られたランダムベクトル $\vec{\omega}$ を足した座標が「遮蔽されているか」を判別する必要があります。

```
ao += step(depthtex.Sample(smp, (rpos.xy + float2(1, -1))*float2(0.5f, -0.5f)), rpos.z)*dt;
```

この `step(depthtex.Sample(smp, (rpos.xy + float2(1, -1))*float2(0.5f, -0.5f)), rpos.z)` の部分が数式でいうと $V(x_p, \vec{\omega})$ に当たります。そして dt が $\cos \theta$ 項であることがわかります。これを全加算して総和を記録します。

ループが終わつた後に

```
ao /= div;
```

としていますが、これが数式の $\frac{1}{\pi}$ の部分になります。結果の `ao` が「遮蔽率」となります。あくまでも遮蔽率ですので 1.0 で完全遮蔽となります。

乗算値として使用するには反転する必要があるため $1.0 - \alpha$ を乗算値として返しています。これをペラポリ描画で見てみると



このような画像となっており、狭い部分が暗くなっているのが分かると思います。これをもとのレンダリング結果に乗算してあげればアンビエントオクルージョンがレンダリング結果に反映されます。



なお、分かりやすくするために影や被写界深度の処理は外しています

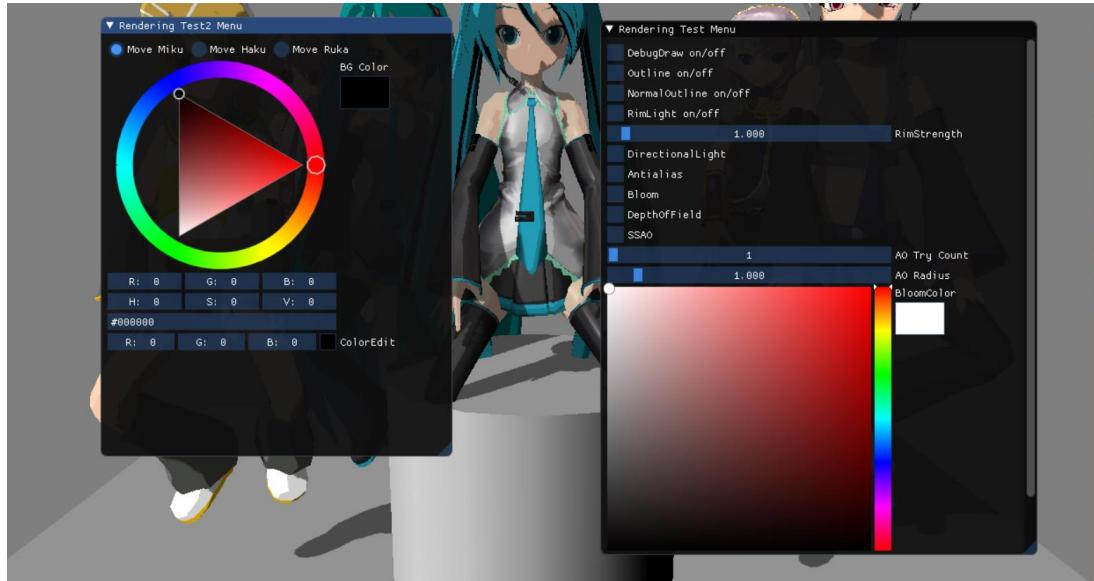
ImGui

ここでは技術デモの設定変更もしくはデバッガ時の GUI ディップア確認などの表示切り替えに使えるライブラリ imgui を紹介します。

ImGuiについて

ImGui とは非常に簡単に MDI 形式(親ウィンドウ…この場合はクライアント領域の中に複数のウィンドウを持つもの)の GUI を提供してくれるライブラリです。GitHub にて公開されています。

<https://github.com/ocornut/imgui>



名前の由来は immediate mode GUI の略です。immediate mode というのは retained mode に対する考え方なのですが、簡単に言うと

- immediate mode: 描画命令の時点で表示場所や大きさなどの情報を指定して描画する
 - retained mode: 描画命令とは別に表示場所や大きさなどの情報を与えて描画する
- このような意味になっています。使用する分にはあまり気にしなくても使えますので忘れてしまっても構いません。

ともかく図のように出力画面中に GUI ウィンドウを出せるようなライブラリです。一般的なウィンドウアプリ GUI の基本的な機能はそろっており、よく使用するものでは

- チェックボックス(Checkbox)
- ラジオボタン(RadioButton)
- スライドバー(SliderInt, SliderFloat)
- カラーピッカー(ColorPicker3, ColorPicker4)

などがあります。そのほかにもいろいろありますが、本書ではこの4種類を試していきます。

ImGui の組み込み

組み込みから描画までの手順はこうなります。

- ① 公式サイトからライブラリを取得する(zip でダウンロード、Git クローンどちらでも構いません)
- ② 初期化処理を行う(ウィンドウハンドルや DX12 デバイスその他が必要)
- ③ 描画前処理(仕様上 NewFrame を呼び出す必要がある)
- ④ 描画処理(Begin～描画 GUI 指定～End)
- ⑤ 描画後処理(Renderer 関数を呼び出し DX12 のコマンドリストに命令を積む)

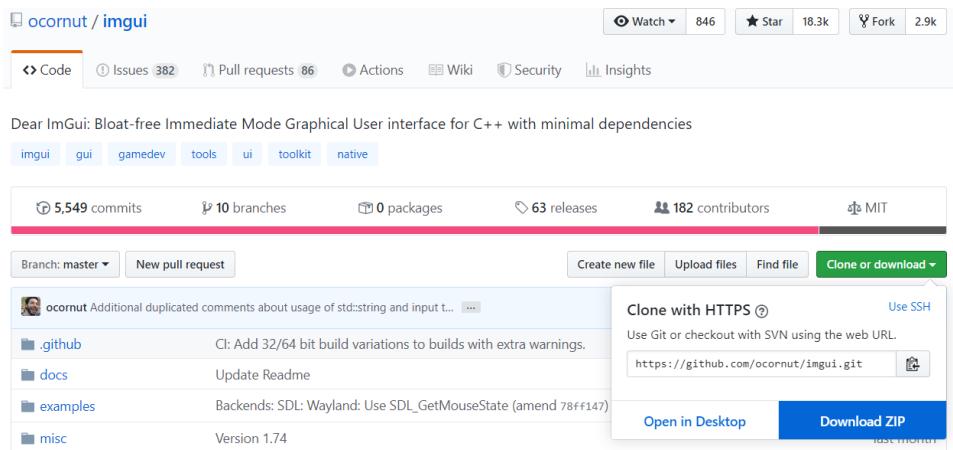
それではさっそく組み込んでいきましょう。

公式サイトからライブラリをダウンロード

まずは先ほど紹介した URL

<https://github.com/ocornut/imgui>

から必要なソースコードを取得します。今回は zip でダウンロードする例を示します。

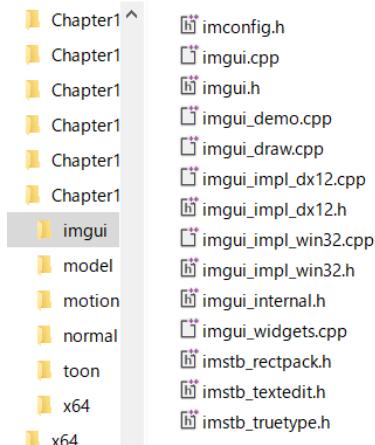


前述の URL にアクセスするとこのような画面になりますので Clone or download をクリックし Download ZIP なりクローンするなりしてソースコードを入手してください。クローンするか ZIP を解凍すると以下の構成になっていると思います。

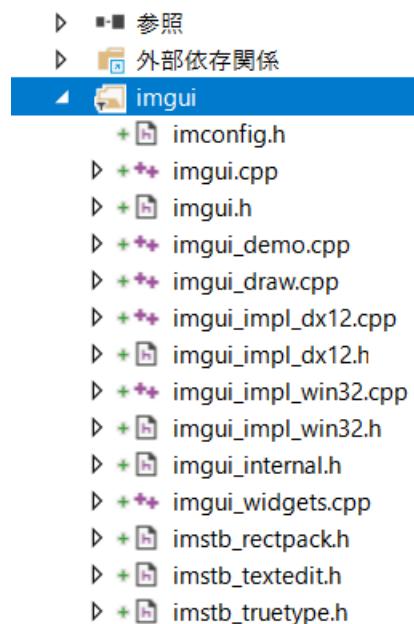
.github	ファイル フォルダー
docs	ファイル フォルダー
examples	ファイル フォルダー
misc	ファイル フォルダー
.editorconfig	EDITORCONFIG ... 1 KB
.gitattributes	テキスト ドキュメント 1 KB
imconfig.h	C/C++ Header 7 KB
imgui.cpp	C++ Source 487 KB
imgui.h	C/C++ Header 213 KB
imgui_demo.cpp	C++ Source 247 KB
imgui_draw.cpp	C++ Source 166 KB
imgui_internal.h	C/C++ Header 124 KB
imgui_widgets.cpp	C++ Source 353 KB
imstb_rectpack.h	C/C++ Header 21 KB
imstb_textedit.h	C/C++ Header 53 KB
imstb_truetype.h	C/C++ Header 188 KB
LICENSE.txt	テキスト ドキュメント 2 KB

今回はこれをライブラリ化するのではなく、必要な h, cpp ファイルを自分のプロジェクトに追加することで組み込みを行います。必要なファイルは上のスクリーンショットに見えているファイルだけではなく、examples フォルダの中から imgui_imple_win32.h, imgui_imple_win32.cpp, imgui_imple_dx12.h, imgui_imple_dx12.cpp の 4 ファイルを取り出して一緒にフォルダに置いておきます。

それでは自分のプロジェクト側の準備ですが、まず自分のプロジェクト内のフォルダに imgui というフォルダを作つてそこに先ほどのファイル(h, cpp)をコピーします。



必要なものがそろっているかどうかを確認しておきましょう。次にプロジェクトに追加しましょう。プロジェクト側でも imgui フィルタを作つてまとめておきましょう。



一旦この状態でプロジェクトをビルドしてみましょう。問題なくビルドできたのであればまずは大丈夫だと考えていいです。もしエラーが起きたらエラーメッセージをよく見てください。必要なファイルがそろっているか等を確認して、それでも原因が見当たらなければネット上で状況を正確に記述して助けを求めるましょう。

問題なければ Application.cpp(Dx12Wrapper.cpp ではなくクライアント側に)に以下の include

文を書いて下さい。

```
#include "imgui/imgui.h"
#include "imgui/imgui_impl_win32.h"
#include "imgui/imgui_impl_dx12.h"
```

ライブラリとしてではなく自分のフォルダに置いたため<>ではなく""で記述します。今回は imgui フォルダの下に置いたため上記のように書いてますが、読者それぞれの状況でフォルダ指定は柔軟に対応してください。インクルードしたら念のためいったんコンパイルしておきましょう。

続いて初期化処理を行います。初期化処理は少しややこしいので漏れがないよう注意してください。

ImGui の初期化

ImGui の初期化にはウインドウハンドルやデバイスやコマンドリストが必要なので、自分のプログラムのウインドウ初期化や DirectX12 の初期化が終わった後で行います。

まず ImGui 用の DescriptorHeap が必要なのですが、それは ImGui 側ではなくクライアント側で作っておく必要がありますので、Dx12Wrapper クラスメンバとして 1 つのデスクリプタヒープを生成する関数を作成します。種別は CBV_SRV_UAV とします。

```
//ImGui 用のヒープ生成(宣言)
ComPtr<ID3D12DescriptorHeap> CreateDescriptorHeapForImGui();

//ImGui 用のヒープ生成(定義)
ComPtr<ID3D12DescriptorHeap>
Dx12Wrapper::CreateDescriptorHeapForImGui() {
    ComPtr<ID3D12DescriptorHeap> ret;
    D3D12_DESCRIPTOR_HEAP_DESC desc = {};
    desc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
    desc.NodeMask = 0;
    desc.NumDescriptors = 1;
    desc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
    _dev->CreateDescriptorHeap(&desc, IID_PPV_ARGS(ret.ReleaseAndGetAddressOf()));
    return ret;
}
```

```
}
```

この関数を Dx12Wrapper クラスの初期化時にでも呼び出しておきます。

```
//初期化時に呼び出す
_heapForImgui = CreateDescriptorHeapForImgui();
if (_heapForImgui == nullptr) {
    return false;
}
```

さらに先ほどの関数で得られたヒープを保持するためにメンバ変数として、_heapForImgui を用意し、Application 側からアクセスできるように HeapForImGui 関数でヒープアドレスを返します。

```
ComPtr<ID3D12DescriptorHeap> _heapForImgui;//ヒープ保持用
ComPtr<ID3D12DescriptorHeap> GetHeapForImgui(); //imgui 用のヒープアクセサ(宣言)
//imgui 用のヒープアクセサ(定義)
ComPtr<ID3D12DescriptorHeap> Dx12Wrapper::GetHeapForImgui() {
    return _heapForImgui;
}
```

これで imgui に必要なヒープができたので Application 側に戻りましょう。まずは imgui そのものの初期化を行います。(imgui のコンテキストは今回使わない)ので戻り値チェックだけで OK)

```
if (ImGui::CreateContext() == nullptr) {
    assert(0);
    return false;
}
```

このコードはどのプラットフォームでも必ず呼び出す必要があります。次に Windows 用の初期化を行います。

```
bool bInResult = ImGui_ImplWin32_Init(_hwnd);
if (!bInResult) {
    assert(0);
    return false;
}
```

引数にはウインドウハンドルを入れてください。ここまで簡単なのですが DirectX12 用の初期化は少しややこしいです。

```
bInResult=ImGui_ImplDX12_Init(_dx12->Device(),//DirectX12 デバイス  
3,//frames_in_flight と説明にはあるが flight の意味が掴めず(後述)  
DXGI_FORMAT_R8G8B8A8_UNORM,//書き込み先 RTV のフォーマット  
_dx12->GetHeapForImgui().Get(),//imgui 用デスクリプタヒープ  
_dx12->GetHeapForImgui()->GetCPUDescriptorHandleForHeapStart(),//CPU ハンドル  
_dx12->GetHeapForImgui()->GetGPUDescriptorHandleForHeapStart();//GPU ハンドル
```

まず一つ目の引数はデバイスを渡します。これは問題ないと思います。

問題は次の引数です。これは関数の引数名には frames_in_flight と書かれており、flight の意味が掴めないので、ソースコードを見てみましたが、頂点バッファとインデックスバッファの複合構造体です。

ソースコードをある程度読んでも用途の正確なところが判明しませんでしたが複数の頂点バッファ、インデックスバッファを用意することで GPU の処理を待たずに次の子マントリスト発行ができるようになっているものと思われます。サンプルでは 3 が指定されていたため 3 を入れておきます(1 でも動きはします)

次はフォーマットですね。これは描画先の RTV のフォーマットと合わせておけばいいでしょう。次が先ほど作ったヒープが役に立ちますね。ヒープのポインタを入れてください。
(※この例では ComPtr を使用していますので Get() で渡しましょう)

最後の二つはヒープから得られる CPU アドレスハンドルと GPU アドレスハンドルですが、これってヒープ渡してるんだから向こうでやってくれればいいような気がしますが、仕様に従っておきましょう(オープンソースなので気に入らなければ変更可能です)。

さて、一応これで初期化部分は完了いたしました。もしここまでエラーもしくは初期化の失敗等がありましたらどこか間違えてないかご確認ください。

次はいよいよ画面上に imgui を表示させていきます。

imgui の表示

ループの中で「表示」コマンドを発行していきますが、ここも少しあやこしいです。
手順は NewFrame を呼び出す(複数)。Begin～描画～End。Render 関数を呼び出す。という流れです。

まずは NewFrame を呼び出します。Win32 用と DirectX12 用と imgui 本体用の3つを呼び出しま

す。

```
//imgui描画前処理
ImGui_ImplDX12_NewFrame();
ImGui_ImplWin32_NewFrame();
ImGui::NewFrame();

これは仕様なので従っておきましょう。次にウインドウ定義部分ですが
ImGui::Begin("Rendering Test Menu");

ImGui::SetWindowSize(ImVec2(400, 500), ImGuiCond_FirstUseEver);

ImGui::End();
```

このように記述します。それぞれ説明していきますが Begin では imgui ウィンドウの名前を定義します。この文字列が imgui タイトルバーに表示されます。SetWindowSize はウィンドウの大きさ指定ですね。

あとから動的に変更できますがデフォルトがかなり小さくて見えづらいため、いったんここで決めておかないといどこに表示されたか分からなければほど小さいウインドウになることがあります。

この例では幅 400 の高さ 500 にしています。用途に合わせてここは後から動的に変更できますので、各自で好きな大きさに設定しておきましょう。

これで表示できるかというとそうではありません。ウィンドウの定義が終わったら(複数のウィンドウの場合も同様) Render 呼び出し始めて描画されます。これも DirectX12 では少し手間です。通常であれば

```
ImGui::Render();
```

で済むのですが、DirectX12 の場合この結果をそのまま DirectX12 のコマンドとして投げてあげる必要がありますので、Render 関数呼び出し後に以下のように記述してください。

```
_dx12->CmdList()->SetDescriptorHeaps(1, _dx12->GetHeapForImgui().GetAddressOf());
ImGui_ImplDX12_RenderDrawData(ImGui::GetDrawData(), _dx12->CmdList());
```

まずコマンドリストに imgui のヒープをセットします。その後の ImGui_ImplDX12_RenderDrawData 関数で imgui 描画処理を指定のコマンドリストに乗つけます。この後にコマンドの Execute が来るようにしていれば imgui が正しく実行され



このように実行結果にウィンドウが表示されるでしょう

ただしこのウィンドウ。移動することも大きさを変更することもできません。これは Windows プログラミングを知っている人はピンと来るかもしれません、ウィンドウプロシージャの部分をいじる必要があります。最初の方で作った関数なので忘れているかもしれません、既に

```
LRESULT WindowProcedure(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam) {
    if (msg == WM_DESTROY) {
        PostQuitMessage(0);
        return 0;
    }
    return DefWindowProc(hwnd, msg, wparam, lparam);
}
```

このような関数があるので探してみてください。これは OS からウィンドウにメッセージが飛んで来た時に呼び出されるコールバック関数でしたね？ここに imgui 用のメッセージ処理を書いてあげるので。特別な事をしないならば難しい事はなく

```
ImGui_ImplWin32_WndProcHandler(hwnd, msg, wparam, lparam);
を呼び出してあげればいいだけです。ウィンドウメッセージをそのまま imgui のウィンドウメッセージとして処理するという関数です。
```

ただし、この関数の本体は imgui_win32_impl.cpp にあるため、使用する場合には extern 指定でプロタイプ宣言する必要があります。つまり

```
extern LRESULT ImGui_ImplWin32_WndProcHandler(HWND, UINT, WPARAM, LPARAM);
LRESULT WindowProcedure(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam) {
    if (msg == WM_DESTROY) {
        PostQuitMessage(0);
        return 0;
    }
}
```

```
ImGui_ImplWin32_WndProcHandler(hwnd, msg, wparam, lparam);  
return DefWindowProc(hwnd, msg, wparam, lparam);  
}
```

のように記述します。

これで imgui のウインドウのタイトルバー部分をつかんで動かしたり、右下を引っ張ってサイズを変えたりすることができるようになったはずです。とはいっても今までは単なるウインドウに過ぎませんので UI コントローラを付けていってあげましょう。

UI コントローラを一通り(4種類)表示してみよう

前にも挙げました

- チェックボックス(Checkbox)
- ラジオボタン(RadioButton)
- スライドバー(SliderInt,SliderFloat)
- カラーピッカー(ColorPicker3,ColorPicker4)

まずは1つずつ表示してみましょう。なおラジオボタンは複数必要であるため3つ表示します。

基本的な文法は

```
ImGui::コントロール名("ラベル文字列",結果を入れるアドレス,オプション);
```

です。最初の ImGui は単なる namespace です。一応戻り値があって、値の変更があれば true、なければ false が返されます。

色々と説明するよりも表示してしまった方が分かりやすいと思いますので、まずは以下のように記述してみてください。

```
bool bInChk = false;  
ImGui::Checkbox("CheckboxTest", &bInChk);  
  
int radio = 0;  
ImGui::RadioButton("Radio 1", &radio, 0); ImGui::SameLine();  
ImGui::RadioButton("Radio 2", &radio, 1); ImGui::SameLine();  
ImGui::RadioButton("Radio 3", &radio, 2);  
  
int nSlider = 0;  
ImGui::SliderInt("Int Slider", &nSlider, 0, 100);  
float fSlider = 0.0f;  
ImGui::SliderFloat("Float Slider", &fSlider, 0.0f, 100.0f);  
  
float col3[3] = {};  
ImGui::ColorPicker3("ColorPicker3", col3, ImGuiColorEditFlags::ImGuiColorEditFlags_InputRGB);
```

```

float col4[4] = {};
ImGui::ColorPicker4("ColorPicker4", col4,
ImGuiColorEditFlags::ImGuiColorEditFlags_PickerHueWheel |
ImGuiColorEditFlags::ImGuiColorEditFlags_AlphaBar);

```

実行すれば以下のようにさまざまなコントロール UI が表示されるはずです。



一つ一つ説明していきます。

Checkbox は第二引数に入れたアドレスに true か false が 1 か 0 で入っています。オンオフ切り替えの物に使えそうですね。

RadioButton は第二引数に入れたアドレスに現在選択中のボタンの数値(第三引数)が入っています。なお RadioButton 命令の後ろの SameLine は「次のコントロール UI を同じ行に表示する」という命令です。このためこの例のラジオボタンは 1 行に並んでいるというわけです。これは複数の中から 1 つ選択するというような事柄に使えそうですね。

次に SliderInt と SliderFloat ですが、これはどちらもスライドバーを表示するものです。戻り値が整数型か浮動小数点数型かの違いだけです。なお、第三引数は最小値、第四引数は最大値を表しています。制限がある数値の指定に使えそうですね。

なお、SliderOOには SliderInt2, SliderInt3, SliderInt4 のように後に数値がついているものがありますが、これは複数の値をまとめて指定するためのスライダーです。例えば SliderFloat3 ならば

```

float fSlider[3] = {};
ImGui::SliderFloat3("Float Slider", fSlider, 0.0f, 100.0f);

```

のように書けば



のように3つまとめて表示されます。ベクトル的なものを指定するのに使えそうですね。

お次はカラーピッカーです。これも SliderOOと同じように3と4の違いは得られる値の数。
それだけです。最もシンプルに書くなら

```
float col4[4] = {};
ImGui::ColorPicker4("ColorPicker4", col4);
```

のような指定になるでしょう。何も指定しないとデフォルトになります。デフォルトでは



このような表示となり α 値の指定がグラフィカルにできません。 α を指定したい場合は
`ImGuiColorEditFlags::ImGuiColorEditFlags_AlphaBar`をオプション(第三引数)に指定してあげます。オプションは or 演算で複数指定することも可能で、最初の例で書いたように
`ImGuiColorEditFlags_PickerHueWheel | ImGuiColorEditFlags_AlphaBar`

のように色相をホイール型(環型)で表示し、 α のバーも表示することができます。そうすると



このように表示されます

その他にもいろいろスタイルはありますので、各自で試してみてください。
今回のコード例では受け取る側の変数を UI の設定の直前に書きましたが、それではループ
内で値が宣言されているため、すぐに元に戻ってしまいます。
そのため実際には受け取り側の変数をループの外で宣言するか static で宣言するかして、内

容の変更を保持するようにしてください。ここでは例として static で書いておきますが、static を使用しない場合はループ外に置くなりしてください。

```
static bool bInChk = false;  
ImGui::Checkbox("CheckboxTest", &bInChk);  
  
static int radio = 0;  
ImGui::RadioButton("Radio 1", &radio, 0); ImGui::SameLine();  
ImGui::RadioButton("Radio 2", &radio, 1); ImGui::SameLine();  
ImGui::RadioButton("Radio 3", &radio, 2);  
  
static int nSlider = 0;  
ImGui::SliderInt("Int Slider", &nSlider, 0, 100);  
  
static float fSlider[3] = {};  
ImGui::SliderFloat3("Float Slider", fSlider, 0.0f, 100.0f);  
  
static float col3[3] = {};  
ImGui::ColorPicker3("ColorPicker3", col3, ImGuiColorEditFlags_::ImGuiColorEditFlags_InputRGB);  
  
static float col4[4] = {};  
ImGui::ColorPicker4("ColorPicker4", col4,  
ImGuiColorEditFlags_::ImGuiColorEditFlags_PickerHueWheel |  
ImGuiColorEditFlags_::ImGuiColorEditFlags_AlphaBar);
```

imgui の活用例

ここまで imgui の基本的な仕様に関してはご理解いただけたと思いますので、ここからは活用例を書いていきます。あくまでも例ですので読者それぞれの活用法を考えて実装してください。

動的に変更したいものとして

- デバッグ表示の ON/OFF
- アンビエントオクルージョンの ON/OFF
- セルフシャドウ ON/OFF
- 画角(30°~150°)
- 光線ベクトル(xyz ベクトル)
- 背景色の変更
- ブレームの色付け

実装してみたいと思います。手順としては

- ① UI 表示部分を作る
- ② Dx12Wrapper 側にインターフェイスを用意する

③ シェーダ側に用意が必要なら必要に応じて以下のものも準備する

- (ア) ルートシグネチャ
- (イ) デスクリプタヒープ
- (ウ) 定数バッファ
- (エ) シェーダ側の変更

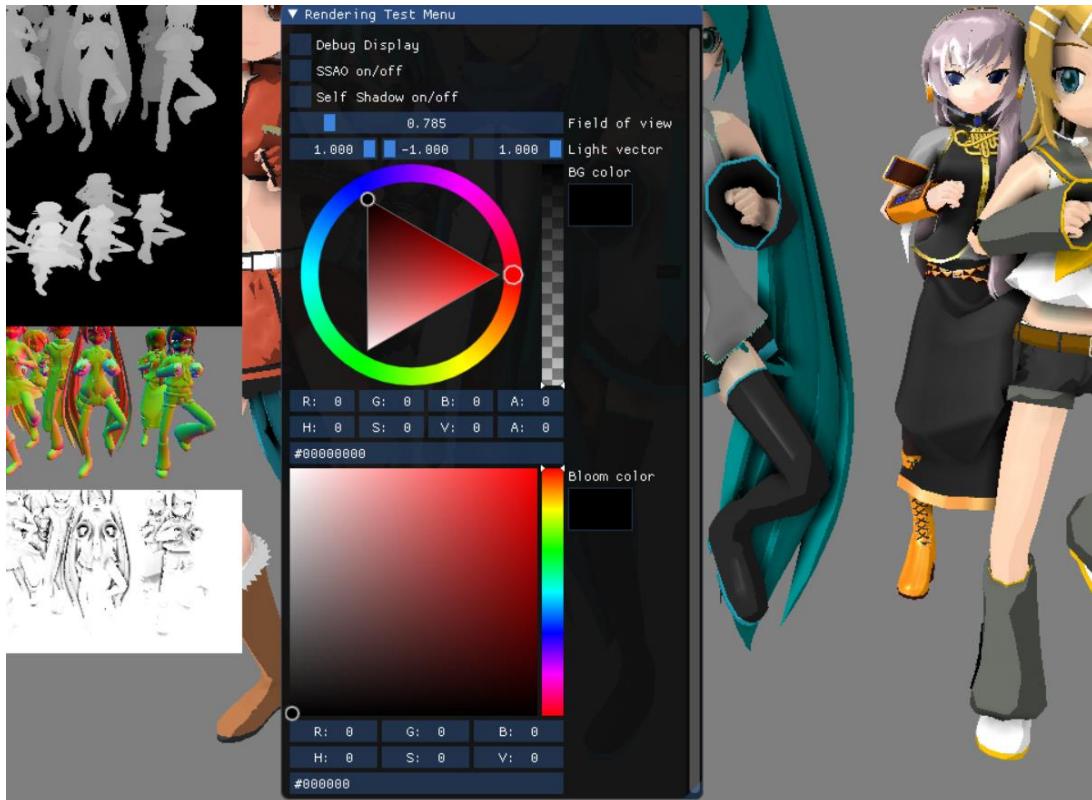
といった具合です。ではさっそくやっていきましょう

UI 表示部分を作る

これは先ほどの例を今回の仕様に合わせれば簡単ですね。なおラベルの日本語表示は保証されてないため英語表記にします。

```
static bool bInDebugDisp = false;  
ImGui::Checkbox("Debug Display", &bInDebugDisp);  
  
static bool bInSSAO = false;  
ImGui::Checkbox("SSAO on/off", &bInSSAO);  
  
static bool bInShadowmap = false;  
ImGui::Checkbox("Self Shadow on/off", &bInShadowmap);  
  
constexpr float pi = 3.141592653589f;  
  
static float fov = pi / 4.0f;  
  
ImGui::SliderFloat("Field of view", &fov, pi/6.0f, pi*5.0f/6.0f);  
  
static float lightVec[3] = { 1.0, -1.0, 1.0f };  
ImGui::SliderFloat3("Light vector", lightVec, -1.0f, 1.0f);  
  
static float bgCol[4] = { 0.5f, 0.5f, 0.5f, 1.0f };  
ImGui::ColorPicker4("BG color", bgCol,  
ImGuiColorEditFlags::ImGuiColorEditFlags_PickerHueWheel |  
  
    ImGuiColorEditFlags::ImGuiColorEditFlags_AlphaBar);  
static float bloomCol[3] = {};
```

```
ImGui::ColorPicker3("Bloom color", bloomCol);
```



ひとまず見た目がこうなっていれば第一段階はOKです。

Dx12Wrapper 側にインターフェイスを用意する

次はこの内容を画面に反映させる準備をしていきます。現在のフラグや数値を反映させるためには Dx12Wrapper に値を教えてあげる必要がありますので、値を変更できるように関数を public で用意します。

```
void SetDebugDisplay(bool flag); // デバッグ表示のON / OFF  
void SetSSAO(bool flag); // アンビエントオクルージョンのON / OFF  
void SetSelfShadow(bool flag); // セルフシャドウON / OFF  
void SetFov(float fov); // 画角(30° ~150° )  
void SetLightVector(float vec[3]); // 光線ベクトル(xyzベクトル)  
void SetBackColor(float col[4]); // 背景色の変更  
void SetBloomColor(float col[3]); // ブルームの色付け
```

次に imgui の内容を Dx12Wrapper に伝えるためにこれらの関数の呼び出しコードをメインループに書いていきます。先ほど定義したメインループ内の static 变数の内容をそのまま渡せばいいです。

```
// Dx12Wrapper に対して設定を渡す  
_dx12->SetDebugDisplay(bInDebugDisp);  
_dx12->SetSSAO(bInSSAO);
```

```
_dx12->SetSelfShadow(blnShadowmap);  
_dx12->SetFov(fov);  
_dx12->SetLightVector(lightVec);  
_dx12->SetBackColor(bgCol);  
_dx12->SetBloomColor(bloomCol);
```

このうち、CPU 側の変更だけで何とかなりそうな部分をまずは実装していきましょう。

表示内容に変更が反映されるようにする

まずは FOV ですね。既に Fov の変更の処理を入れていれば既に Fov 変更の結果は反映されるはずですが、そうでない場合は

```
void  
Dx12Wrapper::SetFov(float fov) {  
    _fov = fov;  
}
```

および

```
_mappedScene->proj = XMMatrixPerspectiveFovLH(  
    _fov,  
    static_cast<float>(wsize.width) / static_cast<float>(wsize.height),  
    1.0f,  
    100.0f);
```

の処理を入れておいてください。次は背景色ですが、これは背景色を変更するとデバッグ出力に警告がでますので注意してください（リソース作成の際のクリア値と ClearRenderTarget のクリア値が食い違っている場合は【パフォーマンス上の問題がある】と警告を発します）。

まずクリア色をメンバ変数の _bgColor でクリアするようにしておきます。

```
_cmdList->ClearRenderTargetView(handle, _bgColor, 0, nullptr);
```

この状態で _bgColor を外側から変更するようにします。

```
void  
Dx12Wrapper::SetBackColor(float col[4]) {  
    copy_n(col, 4, begin(_bgColor));  
}
```

あとはシェーダ側の変更及び渡す構造体の変更、さらに場合によってはルートシグネチャの変更も必要になります。まず構造体の変更だけで行けそうな部分はライトベクトルとシャドウマップ ON/OFF フラグです。これは現在のシーン構造体にライトベクトルを追加すればいいです。この時 hsls 側のアライメントに注意して配置を行いましょう。

```
struct SceneMatrix {  
    DirectX::XMATRIX view;//ビュー
```

```

DirectX::XMMATRIX proj;//プロジェクション
DirectX::XMMATRIX invproj;//プロジェクション
DirectX::XMMATRIX lightCamera;//ライトから見たビュー
DirectX::XMMATRIX shadow;//影行列
DirectX::XMFLOAT4 lightVec;//ライトベクトル(シェーダ側アライメントを防ぐため float4 に)
DirectX::XMFLOAT3 eye;//視点
bool isSelfShadow;//シャドウマップ ON/OFF
};

また、ライトベクトルとシャドウマップフラグの受け取り用の変数も作っておきます。こいつは XMFLOAT3 でもいいでしょう(XMFLOAT4 でも可)。
DirectX::XMFLOAT3 _lightVec;
bool _isSelfShadow;
あとは SetLightVector 関数で普通に代入し
void
Dx12Wrapper::SetLightVector (float vec[3]) {
    _lightVec.x = vec[0];
    _lightVec.y = vec[1];
    _lightVec.z = vec[2];
    SetCameraSetting();
}

シェーダ側に渡すメモリにも反映させます。
_mappedScene->lightVec.x = _lightVec.x;
_mappedScene->lightVec.y = _lightVec.y;
_mappedScene->lightVec.z = _lightVec.z;

また、シャドウマップのライトカメラの座標も変わりますので、
XMVECTOR lightVec = -XMLoadFloat3(&_lightVec);
auto lightPos= targetPos+XMVector3Normalize(lightVec)*
    XMVector3Length(XMVectorSubtract(targetPos, eyePos)).m128_f32[0];
_mappedScene->lightCamera = XMMatrixLookAtLH(lightPos, targetPos, upVec)*
    XMMatrixOrthographicLH(40, 40, 1.0f, 100.0f);

のようにライトからのカメラ座標も変更しておきます。そうしたら次はシェーダ側です。CPU 側で lightVec を追加しましたので BasicShader.hlsl(1 枚目シェーダ) のコンスタントバッファ部分に lightVec を追加します。
//シーン管理用スロット
cbuffer sceneBuffer : register(b1) {
    matrix view;//ビュー

```

```

matrix proj;//プロジェクトン
matrix invproj;//プロジェクトン
matrix lightCamera;//ライトビュープロジェ
matrix shadow;//影行列
float4 lightVec;//ライトベクトル
float3 eye;//視点
bool isSelfShadow;//シャドウマップフラグ
};

```

ではこれを使用してシェーディングの部分も変更します。

```
float3 light = normalize(lightVec);
```

これで imgui のライトベクトルの座標を変更したときに正しく影の落ちる部分やシェーディングが行われていれば正常な処理という事です。次はシャドウマップフラグで影を落とすかどうかを設定しましょう。シャドウマップを行っている部分をまるまるフラグの if 文で囲みます。

```

float shadowWeight = 1.0f;
if (isSelfShadow) {
    (中略)
    shadowWeight = lerp(0.5f, 1.0f, depthFromLight);
}

```

あとは imgui のシャドウフラグをオンにしたりオフにしたりしてみて影が落ちるかどうかが変化するのを確認してください。

残りは新たに定数バッファが必要になります。どれも 2 パス目以降のフラグなので PostSetting という名前にして

```
cbuffer PostSetting : register(b1) {
    bool isDebugDisp;//デバッグ表示フラグ
    bool isSSAO;//SSAO 有効フラグ
    float3 bloomColor;//ブルームの着色
};
```

という定数バッファにしておきます。次は CPU 側の話です。必要なものは

定数バッファ(CreateCommittedResource)

定数用ディスクリプタヒープ(CreateDescriptorHeap)

定数バッファビュー(CreateConstantBufferView)

で、後はペラ用のルートパラメータを増やすだけです。

```
struct PostSetting{
```

```
    bool isDebugDisp;//デバッグ表示
```

```

bool isSSAO;//SSAO オン
DirectX::XMFLOAT4 bloomColor;//ブルームカラー
};

ComPtr<ID3D12Resource> _postSettingResource;
PostSetting* _mappedPostSetting;
ComPtr<ID3D12DescriptorHeap> _postSettingDH;
bool CreatePostSetting();
さて、CreatePostSettingですが、
bool
Dx12Wrapper::CreatePostSetting() {
    auto bufferSize = AlignedValue(sizeof(PostSetting),
D3D12_CONSTANT_BUFFER_DATA_PLACEMENT_ALIGNMENT);
    //まずはバッファを作る
    auto result = _dev->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(bufferSize),
        D3D12_RESOURCE_STATE_GENERIC_READ,
        nullptr,
        IID_PPV_ARGS(_postSettingResource.ReleaseAndGetAddressOf()));
    (エラー処理は略)
    //デスクリプタヒープ
    D3D12_DESCRIPTOR_HEAP_DESC desc = {};
    desc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
    desc.NodeMask = 0;
    desc.NumDescriptors = 1;
    desc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
    result = _dev->CreateDescriptorHeap(&desc,
IID_PPV_ARGS(_postSettingDH.ReleaseAndGetAddressOf()));
    (エラー処理は略)
    //ビュー作成
    D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc = {};
    cbvDesc.BufferLocation = _postSettingResource->GetGPUVirtualAddress();
    cbvDesc.SizeInBytes = bufferSize;
    _dev->CreateConstantBufferView(&cbvDesc,
        _postSettingDH->GetCPUDescriptorHandleForHeapStart());
}

```

```

result = _postSettingResource->Map(0, nullptr, (void**)&_mappedPostSetting);
(エラー処理は略)
return true;
}

```

となります。この関数を初期化の時に呼び出してあげれば、バッファとビューレジストアの準備は完了です。次に渡すべきルートパラメータが増えているため、そこも実装してあげます。

//セッティング

```

range[5].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV;//b
range[5].BaseShaderRegister = 1;//b
range[5].NumDescriptors = 1;//b1
range[5].OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;

```

と書いておきます。あとは描画前にルートパラメータの5番目に PostSetting が来るようになります。

//セッティング

```

_cmdList->SetDescriptorHeaps(1, _postSettingDH.GetAddressOf());
_cmdList->SetGraphicsRootDescriptorTable(5,
                                         _postSettingDH->GetGPUDescriptorHandleForHeapStart());

```

あとはシェーダ側でまずはデバッグのオンオフを試してみましょう。

```

if (isDebugDisp) {//デバッグ出力
    if (input.uv.x < 0.2 && input.uv.y < 0.2) {//深度出力
        float depth = depthTex.Sample(smp, input.uv * 5);
        depth = 1.0f - pow(depth, 30);
        return float4(depth, depth, depth, 1);
    }
    else if (input.uv.x < 0.2 && input.uv.y < 0.4) {//ライトからの深度出力
        float depth = lightDepthTex.Sample(smp, (input.uv - float2(0, 0.2)) * 5);
        depth = 1 - depth;
        return float4(depth, depth, depth, 1);
    }
    else if (input.uv.x < 0.2 && input.uv.y < 0.6) {//法線出力
        return texNormal.Sample(smp, (input.uv - float2(0, 0.4)) * 5);
    }
    else if (input.uv.x < 0.2 && input.uv.y < 0.8) {//AO
        float s = texSSAO.Sample(smp, (input.uv - float2(0, 0.6)) * 5);
        return float4(s, s, s, 1);
    }
}

```

```
}
```

次に SSAO ですが、ここはしっかりと作っていれば簡単です。

```
if (iSSAO) {
    retcol.rgb *= texSSAO.Sample(smp, input.uv); //SSAOを乗算
}
```

あとはブルームの色ですが、もし真っ白にブルームを出力していれば、渡した bloomColor を乗算するだけです。

```
return float4(col.rgb*texSSAO.Sample(smp, input.uv), col.a) +
    float4(bloomAccum.xyz*bloomColor, bloomAccum.a);
```

これで imgui によって見た目を変える方法が分かったと思いますので、読者のほうで変更した リパラメータを考えてみて、見た目を様々に変えてみましょう。

DirectXTK

今週末(1/10)が GFFAward 提出締め切りなので、今週は授業というよりそちらを優先してほしいと思います。

という事で、あまり本編に関係なさそうな要素をピックアップしてやっていきます。

DirectXTK の概要

次に紹介するライブラリは DirectXTK(DirectX ToolKit)です。基本的には DirectX のヘルパクラス、ヘルパ関数集となっており、比較的簡単に DirectX12 製のゲームやアプリケーションを作れるようなライブラリです。サウンドやフォント、その他の機能も充実しているライブラリです。

ライブラリの全貌を知ってしまえば「あれ? DirectX12 のプログラム、もっと簡単に書けたんじゃない?」という気持ちになるかもしれません。ただし本書の目的としてはそこをあえて書くことでハードウェアに近い部分に触ることを目的としています。また DirectXTK には当然ながら MMD の再生機能はついておりませんので、ここまでややこしい話をしてきました。

ここまでを理解してこれた読者なら、きっと DirectXTK を効果的に使う事ができると思いま す。Microsoft 製のオープンソースライブラリですので、コードを読んで勉強にもなるでしょう。それでは早速 DirectXTK を入手します。

DirectXTK12 の入手

既に使用してプログラミングしている読者もおられると思いますが、DirectX12 対応の

DirectXTK は通常版とは別の場所にありますので注意してください

通常版:<https://github.com/microsoft/DirectXTK>

DirectX12 対応版:<https://github.com/microsoft/DirectXTK12>

今回は DirectXTK12 を使用しますので、Effekseer の時と同様に適当なフォルダを作り、そこに git clone してください。

```
git clone https://github.com/microsoft/DirectXTK12.git
```

クローンできたら、中にある

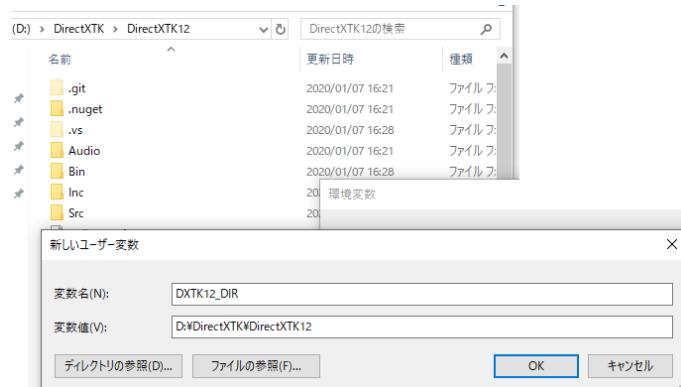
DirectXTK_Desktop_【VisualStudio の バージョン】_Win10.sln
を起動します。

例えば、現在使用中の VisualStudio が 2019 であれば

DirectXTK_Desktop_2019_Win10.sln

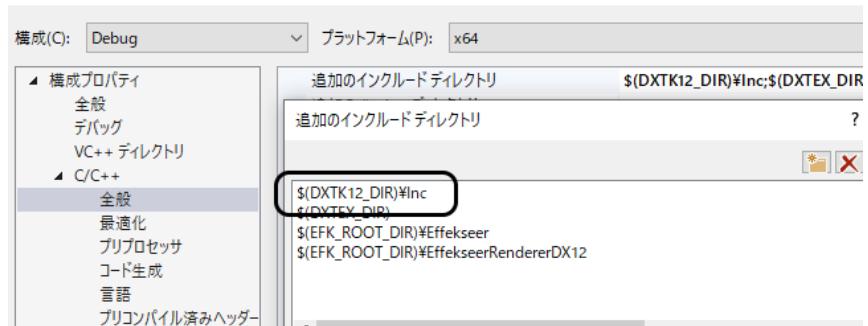
をダブルクリックして起動してください。立ち上がったら早速ソリューション内にあるプロジェクト「DirectXTK12」をビルドしてください。ビルドが終わったら、ライブラリを利用するのに必要なインクルードファイルは Inc フォルダに、ライブラリファイルは Bin\Desktop_2017_Win10\x64\Debug フォルダに入っています。なお Release ビルドの場合は Bin\Desktop_2017_Win10\x64\Release フォルダです。

このライブラリを利用するためには、いつものように環境変数を作りましょう。

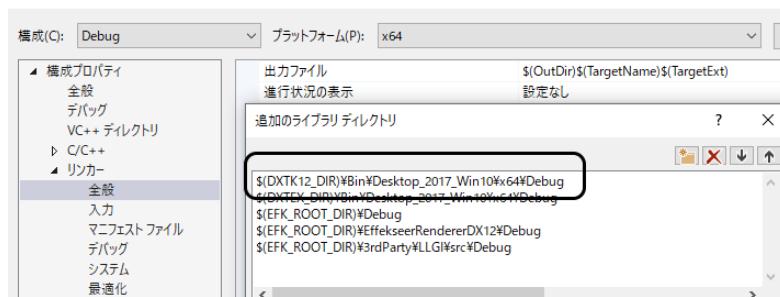


最後に OK を押して環境変数を有効にするのを忘れないでください。あとはクライアント側プロジェクトの環境設定です。

C/C++ → 全般 → 追加のインクルードディレクトリに \$(DXTK12_DIR)\Inc と指定してください。

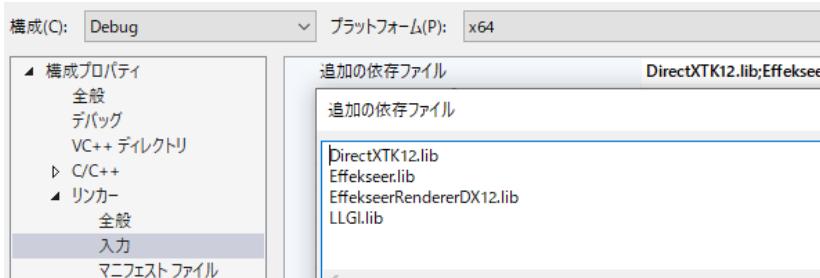


次はライブラリの設定です。リンク→全般→追加のライブラリディレクトリに



\$(DXTK12_DIR)\Bin\Desktop_2017_Win10\x64\Debug(もしくは Release)

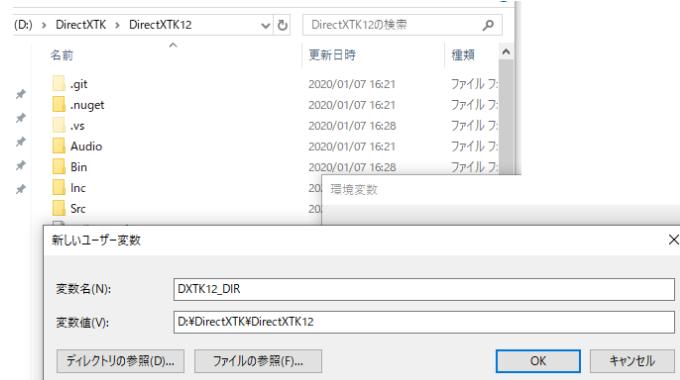
を指定します。次はリンク→全般→追加の依存ファイルに DirectXTK12.lib を追加します。



これにて準備完了です

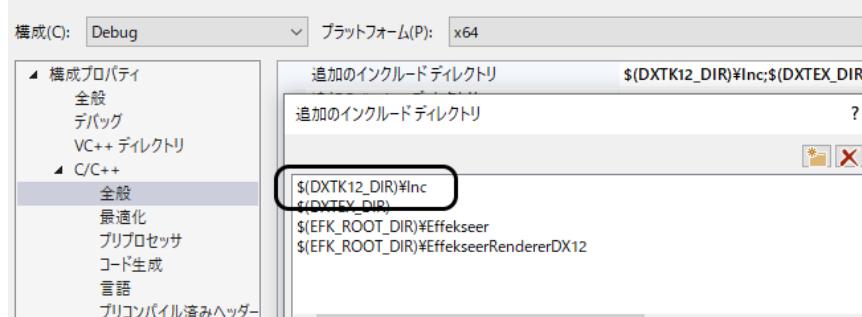
DirectXTK の組み込み

このライブラリを利用するためにつまのように環境変数を作りましょう。

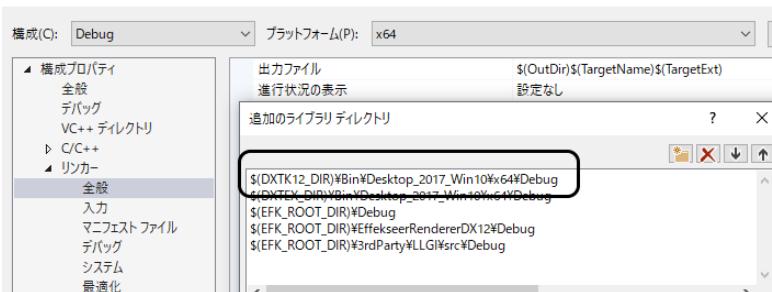


最後に OK を押して環境変数を有効にするのを忘れないでください。あとはクライアント側プロジェクトの環境設定です。

C/C++→全般→追加のインクルードディレクトリに \$(DXTK12_DIR)\Inc と指定してください。

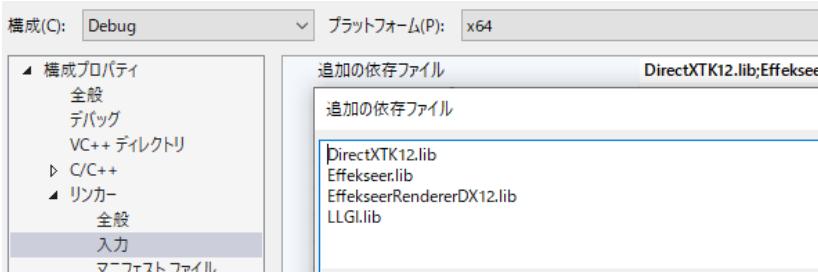


次はライブラリの設定です。リンク→全般→追加のライブラリディレクトリに



\$(DXTK12_DIR)\Bin\Desktop_2017_Win10\x64\Debug(もしくは Release)

を指定します。次はリンク→全般→追加の依存ファイルに DirectX12.lib を追加します。



これにて準備完了です

特定のフォントで特定の文字列を表示する

まずはインクルード文が通るのかどうかを確認したいのですが、今回はフォント表示のみでないので以下の2つのみインクルードします。

```
#include<SpriteFont.h> //文字列を表示するのに必要なもの
```

```
#include<ResourceUploadBatch.h> //DXTK 系列のリソースを使用するのに必要なもの
```

まずはインクルード文を書いた後でコンパイルしてみて下さい。パス通しが失敗していればここでエラーが発生します。

次に Dx12Wrapper から必要なデータを取得できるように以下の関数を public で作ってください。

```
//ビューポートを返す
```

```
D3D12_VIEWPORT GetViewPort() const;  
//SpriteFont用のDescriptorHeapを返す  
ComPtr<ID3D12DescriptorHeap> CreateDescriptorHeapForSpriteFont();
```

次は Application 側で必要なオブジェクトをメンバ変数もしくはグローバルスコープで宣言します。

```
DirectX::GraphicsMemory* _gmemory = nullptr;//グラフィックスメモリオブジェクト  
DirectX::SpriteFont* _spriteFont=nullptr;//フォント表示用オブジェクト  
DirectX::SpriteBatch* _spriteBatch= nullptr;//スプライト表示用オブジェクト
```

GraphicsMemory オブジェクトを直接操作することはありませんが、内部でシングルトン状態となっておりこのオブジェクトの本体を保持しておく必要があります。

フォント表示用オブジェクト SpriteFont はもちろん必要ですが、これを表示するために SpriteBatch オブジェクトが必要になりますので、この3つのオブジェクトを生成します。

Dx12Wrapper の初期化が終ったところで以下のようにして3つのオブジェクトを生成します。

GraphicsMemory 初期化

```
_gmemory = new DirectX::GraphicsMemory(_dx12->Device());
```

SpriteBatch 初期化

```
DirectX::ResourceUploadBatch resUploadBatch(_dx12->Device());  
resUploadBatch.Begin();  
DirectX::RenderTargetState rtState(DXGI_FORMAT_R8G8B8A8_UNORM, DXGI_FORMAT_D32_FLOAT);  
DirectX::SpriteBatchPipelineStateDescription pd(rtState);  
_spriteBatch= new DirectX::SpriteBatch(_dx12->Device(), resUploadBatch, pd);
```

ここが少しあややこしいのですが、SpriteBatch を初期化するために ResourceUploadBatch および SpriteBatchPipelineStateDescription オブジェクトが必要であるため上記のようなコードになっていますなお、名前から推測できると思いますがスプライト用のテクスチャデータを転送するためのオブジェクトなので、転送がすんだら解放されてもいいものなのでローカル変数で定義しています。

次に SpriteFont オブジェクトの初期化ですが、上記の SpriteBatch 初期化の直後に以下のように書いてください。

```
_heapForSpriteFont=_dx12->CreateDescriptorHeapForSpriteFont();  
spriteFont= new DirectX::SpriteFont(_dx12->Device(),
```

```

resUploadBatch,
L"font/fonttest.spritefont",
heapSpriteFont->GetCPUDescriptorHandleForHeapStart(),
heapSpriteFont->GetGPUDescriptorHandleForHeapStart());

```

なお、ImGui の時と同様にフォント用のヒープ(CBV_SRV_UAV)を用意して取得しておきます。これも後から使いますので、メンバ変数がグローバルスコープで宣言してください。
なお L"font/fonttest.spritefont" はフォントデータの相対パスです。

https://github.com/boxerprogrammer/directx12_samples/blob/master/Chapter19/font/fonttest.spritefont

から取得してください。なおこのフォントデータは自作することもできますが、それは次の項で説明します。

ともかく spriteFont を読み込んだらこれを GPU 側に転送する処理を記述します。

```

auto future=resUploadBatch.End(_dx12->CmdQue());//転送
//待ち
_dx12->WaitForCommandQueue();
future.wait();

```

これでフォントデータが GPU 側に転送されました。次にフォントを表示するために spriteBatch に対してビューポートを設定します。

そこまでできたら準備完了です。メインループに移動して、すべての表示が終わった後で以下の処理を書いてください。

```

_dx12->CmdList()->SetDescriptorHeaps(1, _heapForSpriteFont.GetAddressOf());
_spriteBatch->Begin(_dx12->CmdList());
spriteFont->DrawString(_spriteBatch, "Hello World", DirectX::XMFLOAT2(102, 102),
DirectX::Colors::Black);
spriteFont->DrawString(_spriteBatch, "Hello World", DirectX::XMFLOAT2(100, 100),
DirectX::Colors::Yellow);
_spriteBatch->End();

```

黒と黄色を少しずらして表示しているのは、文字を立体的に見せるためです。この結果画面

✖ LNK2001 外部シンボル "IID_ID3D12Device" は未解決です。
✖ LNK1120 1 件の未解決の外部参照

上に HelloWorld と表示されれば成功です。早速ビルドしてみましょう…と言いたいところですが、もしかしたら以下のリンクエラーが出るかもしれません。

「IID_ID3D12Device は未解決です」

理由は DirectX12.lib じしんが IID_ID3D12Device を参照しているのですが、定義がどこにも見つからないためです。これは IID_PPV_ARGS のときに取得している GUID というものを参照するためのモノなのですが、その ID を直接指定している箇所が DirectX12.lib 内にあるためです。外部ライブラリなので、ここはおとなしく必要なライブラリを追加しましょう。必要なライブラリの名前は

dxguid.lib

なので #pragma comment(lib,"dxguid.lib") でもプロジェクト設定でもいいのでライブラリの参照コードを書いておいてください。

ここまでができたら実行してみて下さい。左側に HelloWorld と黄色い文字が影付きで表示されれば成功です。



なお、なぜか imgui のメニューを開くとフォントが消えてしまうのですが、本書執筆時点では原因が分かっておりません。申し訳ありません。

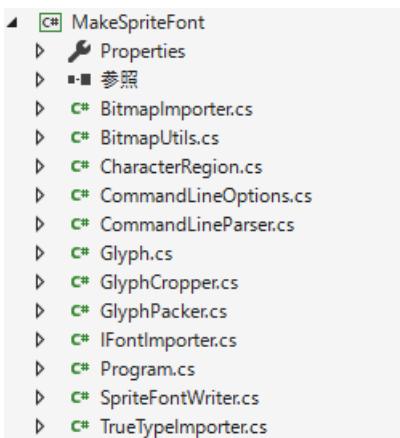
さまざまなフォント、文字列を表示してみよう

前項で使用したフォントでは日本語が使えません。また今回はメイリオを使用しましたが、特殊なフォントを使ったり日本語を表示したりしたいと思います。その場合は文字セット自分で作る必要があります。

この項では文字セットを作る方法を示します。必要なものはこの章の最初で紹介した DirectXTK の通常版: <https://github.com/microsoft/DirectXTK> です。

「え？ それって 12 に対応していないんじゃないの？」と思われるかもしれません。必要なのはこの中にある「MakeSpriteFont」というプログラムです。

ひとまずは ZIP でもよいので DirectXTK のプログラムをダウンロードしてみて下さい。そしてその中にある DirectXTK_Desktop_1 ディレクトリ\デージョン\Win10.sln を起動してください。中に MakeSpriteFont というプロジェクトがあると思います。



このプロジェクトをビルドしてください。(デバッグをすることもないと思いまして、Release でよいでしょう)

ビルドすると、当該プロジェクトの bin/Debug(もしくは Release) に MakeSpriteFont.exe というアプリケーションが出来ていると思います。このアプリケーションは GUI ではなくコマンドラインで使う物なのでダブルクリックしても何も起きません。

ということでコマンドプロンプトを起動して、exe があるフォルダまで移動します。そのまま exe をコマンドラインから実行してみて下さい。いろいろと説明が出てくると思いますが、使用方法は

MakeSpriteFont.exe "フォント名" 出力ファイル名.spritefont オプションです。このオプションが重要なのですが、例えば「メイリオ」で大きさ 36pt で「ひらがな/カタカナ」を使用したい場合は

MakeSpriteFont.exe "メイリオ" meiryo.spritefont /FontSize:36 /CharacterRegion:0x3000-0x30ff と指定します。CharacterRegion というのが「表示できる対象文字(unicode 指定)」となります。

例えばこの手法で作ったフォントを使うならば自分のプロジェクト内の font フォルダに作成した meiryo.spritefont をコピーしておき、フォント読み込み部分を

```
spriteFont = new DirectX::SpriteFont(_dx12->Device(),  
    resUploadBatch,  
    L"font/meiryo.spritefont",  
    _heapForSpriteFont->GetCPUDescriptorHandleForHeapStart(),  
    _heapForSpriteFont->GetGPUDescriptorHandleForHeapStart());
```

とし、文字列表示部分を

```

spriteFont->DrawString(_spriteBatch, L"こんにちはハロー", DirectX::XMFLOAT2(102, 102),
DirectX::Colors::Black);
spriteFont->DrawString(_spriteBatch, L"こんにちはハロー", DirectX::XMFLOAT2(100, 100),
DirectX::Colors::Yellow);

```

とすれば(日本語の場合 L を付けるのを忘れないように)、以下のようにひらがなカタカナが表示されます。



さて、ここで疑問がわくと思います。

「うん、わかった。ひらがなとカタカナは表示されたけど、これ漢字とか表示できないの?」

できますがかなり問題があります。漢字を含めてしまうと膨大な数のフォントグリフ(ベジエ曲線)をビットマップにレンダリングする事になり、それには膨大な時間(下手をすると数十時間)がかかってしまいます。

必要な文字列だけ抽出するようなプログラムを作ったほうが早いです。どういうことがいようと、フォームなどに文字列を入力すると、必要な文字だけが入った文字範囲を出力してくれるようなアプリがあれば楽に設定ができるそうです。

サーバーに MakeSpriteFontCommandGenerator というファイルがあるのでコピーして実行してみて下さい。



↑の上段のテキストボックスにテキストを入力するとリージョンが表示されるのが分かると

思います。

これをそのままコマンドラインに張り付けて実行します。そうしたら使いたい文字が全部入った文字セットができるおりますので、それを用いて表示させてみて下さい。

レイマーチング

レイマーチング、一部で最近はやっているようですが、古典的レイトレーシングをクリアしているみなさんにとってはたいして難しくないものです。あと一時期にどつかで、画面の中央からの距離で加工をする/しないを決定したのを覚えているでしょうか?画面の真ん中に円形のものができましたよね?

実際はアレの応用です。

レイマーチング基本といふか初步の話

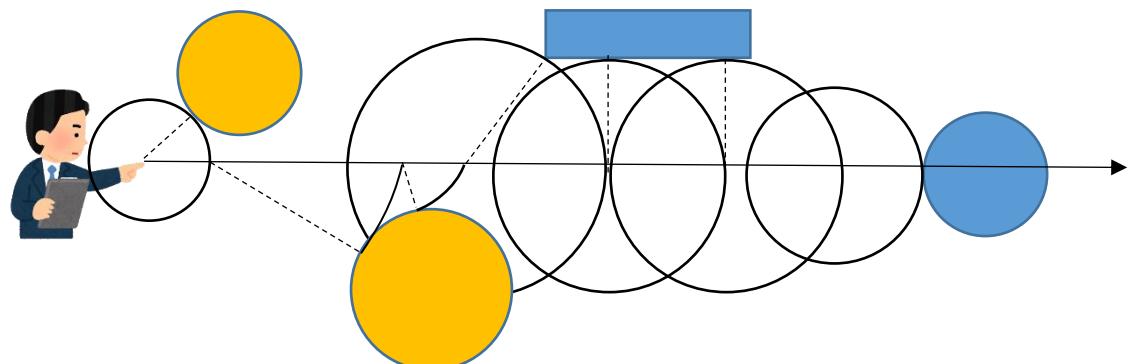
レイトレーシングは衝突点を求めるところから話が始まっていて、球体と直線の交点を求めるのが出発点でした。

レイマーチングはその出発点が少し違っていて SDF(Signed Distance Function)つまり距離関数(符号付き)を用いて物体表面を記述するものです。

例えば「球体との距離であれば」

```
float SDFSphere(float3 pos, float3 center, float r){  
    return length(center-pos)-r;  
}
```

よくレイマーチングの模式図として



こういう図が用いられます。いきなりこれを見せられても何が何だか分かりませんね?後

から振り返って見れば、適切な図であったことは分かるのですが、初心者に対しては「ナンジャラホイ」と印象を持たれてしまうんじゃないかなあ…って思います。

ひとまずこいつは「数学」からのアプローチではなく「アルゴリズム的アプローチ」で考えたほうがよさそうです。

- ①視点と視線ベクトルを取得する
- ②始点=視点とする
- ③始点から最も近いオブジェクトとの距離を求める(絶当たり)
- ④③で得られた距離の分だけ現在の視点から視線ベクトル方向に進んだ所を始点とする
- ⑤③に戻る

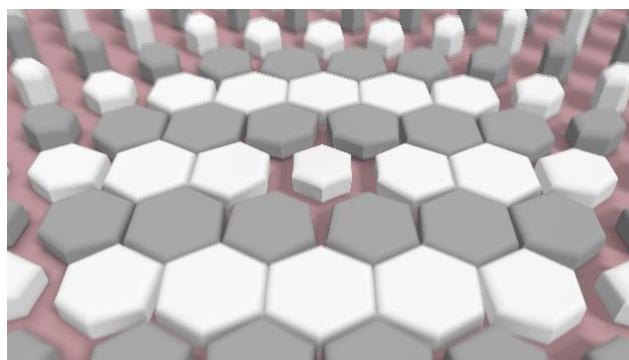
これを数学的に書くと

$$P_0 = \text{視点}, V = \text{視線ベクトル(正規化済)}$$

$$P_{n+1} = P_n + V * \text{SDF}(P_n, \text{Object}_{\min})$$

てな感じになると思います。もう何を言ってるんだか分からないと思います。ちなみに SDF は特定のオブジェクトとの最短距離です。

ともかく、一番近いところの最短距離を測って進んでいくことは分かりましたかね？ここでレイマーティングとか言うと以下のような画像を思い浮かべると思います。



そう、幾何学的、そして繰り返し…

先ほどのアルゴリズムでは結局は交点までいかにしてたどり着くのか…という事にしか触れていませんね？

さてさて…このアルゴリズムとこの結果は何の関係があるんでしょうか？それは距離関数を使っているところに秘密があります。

2Dにおける距離関数

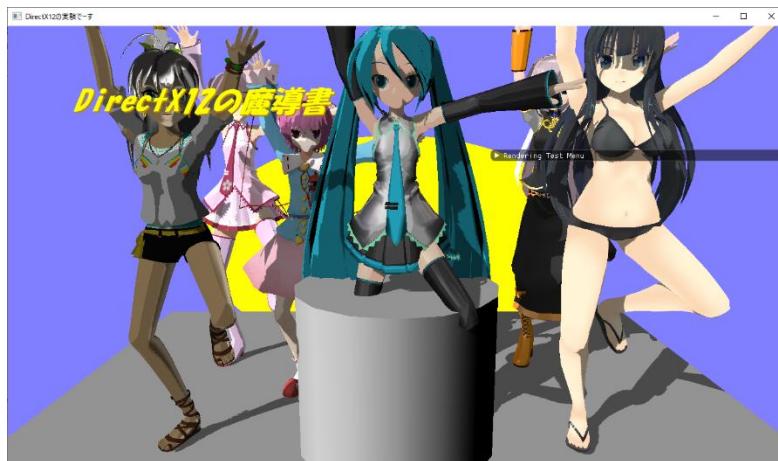
例えば、`SDFCircle2D` を以下のように定義します。中心からの距離によってその値を返します。

```
float SDFCircle2D(float2 xy, float2 center, float r) {  
    return length(center - xy) - r;
```

```
}
```

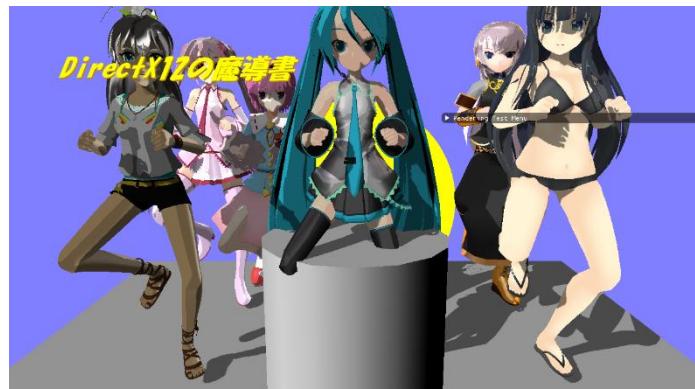
```
float sdf = SDFCircle2D(inputnpos.xy, float2(0, 0), r);
if (sdf < 0) {
    return float4(1, 1, 0, 1);
}
else {
    return float4(0.5, 0.5, 1, 1);
}
```

と書くと



となりますが、円の大きさが横広ですので、アスペクト比を乗算しましょう。

```
float2 aspect = float2(w / h, 1);
float sdf = SDFCircle2D(inputnpos.xy * aspect, float2(0, 0), r);
```



背景が日の丸になってしまいます。イデオロギー的な意味はないので無視してください。うーん、でもなんかレイマーチングになりそうになりますね。そこで距離関数をもう少しひじります。

fmodで大量生産

何かといふと fmod を使用します。

fmod 関数は浮動小数電数において、整数型における%演算子のような働きをします。つまり「割った余り」を返すのですが、これがミソです。fmod なので浮動小数点数で割った余りを返します。

0.14 を 0.1 で割った余りは 0.04 という風になりますが、これを利用することで、繰り返し表現を実現することができます。

例えば uv 値に対して 0.1 を fmod してやれば 0.1 の余りが出ます。これにより

```
sdf = length(fmod(input.uv, 0.1))-0.1f;  
if (sdf < 0) {  
    return float4(1, 0.5, 0, 1);  
}else {  
    return float4(0.5, 0.5, 1, 1);  
}
```



こんな感じになります

何故なのかはわかりますね？これはまず範囲を uv 両方向 0~0.1 になり、そこから 0.1 を引くことで -0.1~0.0 となります。length は距離を測るので三平方状態になり、このように左上中の円ができます。但しよく見ればわかるようにアスペクト比が反映されてないですね。

```
float2 aspect = float2(w / h, 1);  
sdf = length(fmod(input.uv*aspect, 0.1))-0.1f;
```

とすることによってアスペクト比が調整されます。あとは中心を真ん中に持ってくるだけです。

```

float SDFLatticeCircle2D(float2 xy, float divider) {
    return length(fmod(xy, divider) - divider / 2) - divider / 2;
}

```

という関数を作ります。この中にアスペクト比を考慮した uv 値を入れてやれば



このように円表示になります。なんかのオープニングムービーみたいになっていました。ここまでやって思ったことは「一般にレイマーチングって言われてる奴って、距離関数は使うけど本当の意味でのレイマーチングやってなくね?」ってことです。

だって

- ①視点と視線ベクトルを取得する
- ②始点=視点とする
- ③始点から最も近いオブジェクトとの距離を求める(縦当たり)
- ④③で得られた距離の分だけ現在の視点から視線ベクトル方向に進んだ所を始点とする
- ⑤③に戻る

って言ってるけど、これ縦当たりじゃないじゃん…縦当たりするまでもなくっていうか、どれに当たるか最初から決まってるよね?

ってこと。ひとまずは 2D に関してはそう思えますよね。ひとまずはそう思っておきましょう。次にこれを疑似的に立体化したいと思います。まずは疑似的に深さ情報を作ります。

```

float sdf = SDFLattice2D(uv*aspect, divider);
if ( sdf<0) {
    sdf = abs(sdf)*(1.0/divider); //一※
}

```

```

    return float4(sdf,sdf,sdf,1);
};


```

これがどんな結果になるか分かりますか？sdf は※マークの行により 0.0~1.0 となり、中心



が 1.0 で周囲が 0.0 になります。この結果を `return float4(sdf,sdf,sdf,1);` で出力するとこのようになります。まあ実際深度値は手前のほうが小さいのでこれは少し違うんですけどね。それではここから無理やり法線ベクトルを算出し、光を当ててみましょう。

```

sdf = abs(sdf)*(1.0 / divider);
float2 xy = (fmod(uv*aspect, divider) - (divider / 2))*(1.0 / divider)*float2(1,-1);
float3 sdfnormal = normalize(float3(xy, -sdf));

```

前述のように sdf は反対側に向けています。で uv 値を SDF 関数と同様の変換を行え、 nx, ny を計算、そこから法線ベクトルを作ります。そしてその法線ベクトルとライトベクトルの内積を

```

sdf = max(saturate(dot(normalize(float3(-1, 1, -1)), sdfnormal))), 0.15f);
return float4(float3(sdf, sdf*0.8, sdf*0.8), 1);

```

のようになれば…



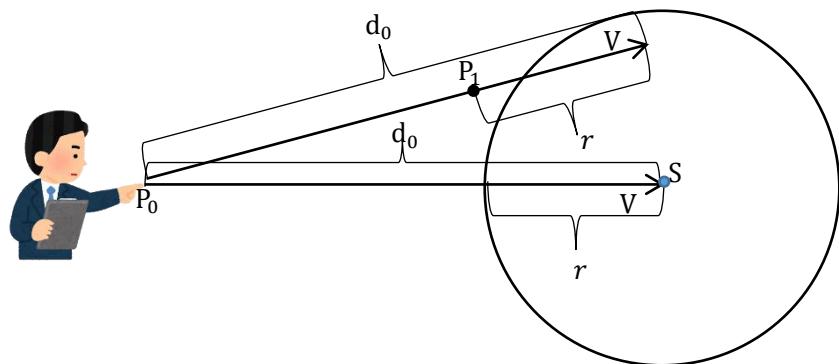
ほら、こんなに簡単にたくさん球体が。こんなに綺麗にできましたよ？

3Dにおける距離関数とレイマーチング

3Dになると途端にレイマーチングになります。でも距離関数が重要なのは変わりません。ただしかなり面倒です。

普通にレイトレーシングやらなきゃいけません。まずは再び1つの球体を「レイマーチング」で表示します。レイマーチングの場合は「固定ループ」を用います。ただこれをやると…重い…ループ回数に注意。

まずは1つの球体について考えます。球体の真正面を向いていればきれいにヒットするのですが…上下左右にある程度ズレると1回ではヒットしなくなります。



球体の距離関数は $\text{length}() - r$ です。↑の図で言うと

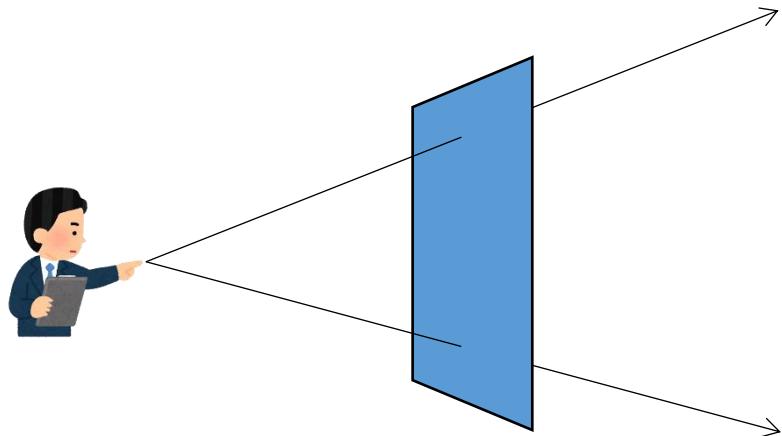
$$d_n = S - p_n$$

$$P_{n+1} = P_n + \hat{V}(d_n - r)$$

こんな感じですかね。これを繰り返して、 $d_n < \varepsilon$ になつたら処理を打ち切る…と。ちなみにイ

プシロンは十分に小さな値ね。と言う訳で真正面は1回で処理が終わると言う訳。

さて、レイトレーシングであるからには「スクリーン」の概念と視点の概念が必要なのでこれも用意します。



とはいっても、数学の時間のレイトレと違うのはピクセルシェーダに入った時点で、スクリーンの

場所は決まっているという事。あとは視点を仮にでも用意して視線ベクトルを作り、そこからレイマーチングを行えばいいのだ。とはいっても uv とやっちゃうといろいろアレなので、pos でやります。pos 言うても SV_POSITION の場合保管されてないので、なければ POSITION の変数作ってスクリーンが -1~1 の体でやります。視点は z=1 のど真ん中という感じでいいでしょう。つまり

```
float3 eye = float3(0, 0, -2.5); // 視点  
float3 tpos = float3(input.tpos.xy * aspect, 0);  
float3 ray = normalize(tpos - eye); // レイベクトル(このシェーダ内では一度計算したら固定)  
float rsph = 1.0f; // 球体の半径  
for (int i = 0; i < 64; ++i) {  
    float len = SDFSphere3D(eye, rsph);  
    eye += ray * len;  
    if (len < 0.001f) {  
        return float4((float)(64-i)/64.0f, (float)(64-i)/64.0f, (float)(64-i)/64.0f, 1);  
    }  
}  
return float4(1, 0, 0, 1);
```

みたいに考えましょう。あ、SDFSphere3D は自分で考えてみてね。

```
float SDFSphere3D(float3 pos, float r) {  
    return length(pos - float3(0, 0, 5)) - r;  
}
```

0,0,5 は仮の球体の中心です。



一応陰影は到達回数によってつけています。もちろん正しい陰ではありませんが、球をレンダリングしているのが重要なのです。さて、これをあとは fmod で繰り返してみましょう。

```

float SDFSphereLattice3D(float3 pos, float divider, float r) {
    return length(fmod(pos, divider) - divider / 2) - r;
}

```

としておき、呼び出し側は

```
float len = SDFSphereLattice3D(abs(eye), rsph*2, rsph/2);
```

とでもしてやれば



このように、ザ・レイマーチングってかんじになります。

球体以外の距離関数の参考サイトは

<http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>

ですので、いろいろ形状を試してみて遊んでみて下さい。

法線ベクトルを返してみる

今の戻り値は距離関数の距離ですので、法線は別的方式で返してあげる必要があります。hlsl で C++ の参照みたいにことをしたい場合はどうするのか、今更ですがやってみましょう。

やり方は簡単です。

引数の型名の前に `out` をつけて関数宣言を行います。そんだけです。入力と出力なら `inout` に

します。今回は出力のみなので `out` としましょう。それで球の距離関数を定義すると…

```
float SDFModSphere3D(float3 xyz, float divider, float r,out float3 normal)
```

こんな感じになります。

で、例えばこの関数の中で `normal` に法線を返すようにすると

```
normal = normalize(fmod(abs(xyz), divider) - divider / 2)*float3(1,1,-1);
```



このようになります。あ、ランパートは自分で計算してください。法線に `abs` つけてるんで、ちょっとおかしなことになってますが、そこは各自で調整お願いします。

そろそろ提出物

まあ、多分みんなそこそこできてると思うんで、できてるものを
¥stfs¥APC_ABCC クリエイティブ¥gakuseigame¥川野部屋¥652 教室制作提出場所
に挙げてください。

期限は 1/31(金)です。

プロジェクト上げるとデカすぎるので、`exe` と必要リソースのみ上げてください。
上げた人が分かるようにタイトルバーに学籍番号氏名を書いてください。