



茶番

何とも挑戦的なタイトルと、挑戦的な授業だと思う。何しろプロでも使ってる人がそれほど多くないであろう DirectX12 を君たちに教えようというのだ。俺は間違っているのかもしれない。

だが俺からみんなに言える事が一つある。それは

**他の奴らと同じ戦略をとったところで
勝つ可能性はあんまり無い！**

お手々ついで、足並みそろえてなど、クソっくらえだ!! ファッキン同調圧力!!!



同調圧力なんか大っ嫌いだ！ ドーカ!!

そんなもんに屈する奴らはどうせ全員仲良く諦めるハメになるんだ。周りが DX9 しかやってない時に DX11 をやっていたやつらは就職活動を勝ち抜いたのだ。これは事実なのだ。ゲーム開発者ってのはな!!! 他を出し抜くことが求められるとんじや!!! そして今、周囲では DX11 が当たり前になってきている。では DX12 に行くしかないじゃなけり!!!



皆が DirectX11 をやるようになつたら… DirectX12 をやるしかないじゃなし!!!!
仕方ない!……求人数は増えているが、求められるレベル自体は下がつてない。それどころか
年々上がつていて。これが意味するところは君たちが難しい事に対するチャレンジャーである
ことが求められているのだ。



あまりにも脅しすぎたかな。既に DX11 などを始めている一部の意識高い学生さん以外のほとんどうが DxLib で開発している事だろう。勉強の指針になるかもしれない。ここに DxLib で開発した時と、DirectX12 で開発した時の違いを述べてみる。

DxLib との違い。

- ①: まず環境設定が大変
 - ②: めちゃくちゃインクルード文が増える
 - ③: ライブライのリンクも滅茶苦茶増える
 - ④: ウィンドウ出すまでがクソ面倒
 - ⑤: レンダリングパイプラインを知っておく必要がある
 - ⑥: 「シェーダ」というのを記述しないと絵が表示されない
 - ⑦: 必要な基礎知識だけでも頭パンクするレベル
 - ⑧: 直接手を突っ込んでいろいろやれる
- どうみても數え役満レベル。君たちに耐えられるのだろうか?
- ⑨に関しては、⑦までこなせた人間からすれば逆にありがたいことではあるのだが、ここまで到達するのが結構大変。死にそう?

まあ人間、そう簡単には死ねないので大丈夫大丈夫。頑張ろう。

内容

| | |
|-------------------------------|----|
| 茶番 | 1 |
| はじめに | 10 |
| 予定 | 10 |
| 膨大なる用語(本当にすまない!) | 12 |
| 環境設定 | 19 |
| ウィンドウを出すまで | 21 |
| アプリケーションのハンドル | 22 |
| Direct3D の初期化 | 30 |
| Direct3D の初期化について | 31 |
| Direct3DDevice | 31 |
| コマンドまわり | 37 |
| スワップチェイン | 43 |
| ディスクリプタとレンダーターゲット | 49 |
| レンダーターゲット | 54 |
| ルートシグネチャー | 57 |
| 画面に影響を与えよう(画面を特定の色でクリア) | 60 |
| レンダーターゲットクリアコマンド発行 | 60 |
| フェンス | 66 |
| そもそも非同期処理とは? | 66 |
| で、結局 DirectX12 ではどうなの? | 68 |
| フェンスの仕組み | 71 |
| ではフェンスを実装しようか | 71 |
| ポリゴンを表示しよう | 73 |
| 頂点情報を作る | 73 |
| 頂点レイアウト | 75 |
| 頂点バッファ | 77 |
| 頂点バッファビュー | 79 |
| そんな事よりシェーダ書こうぜ | 80 |
| シェーダ読み込み | 81 |
| パイプラインステートオブジェクト(PSO) | 82 |
| その他やらなければならない事 | 84 |
| リソースバリア | 84 |
| ビューポート | 85 |
| 残り色々セツト | 86 |

| | |
|----------------------------------|-----|
| テクスチャマッピング | 91 |
| UV 座標を付加..... | 92 |
| 頂点情報構造体に UV を追加..... | 92 |
| 頂点情報に UV 座標を追加..... | 93 |
| 頂点レイアウトを追加..... | 93 |
| シェーダ側の引数を追加..... | 93 |
| テクスチャリソースの作成..... | 94 |
| ビットマップ読み込み..... | 97 |
| テクスチャバッファへの書き込み..... | 98 |
| シェーダリソースビューの作成..... | 99 |
| サンプラ | 101 |
| サンプラの設定..... | 101 |
| ルートシグネチャへサンプラを適用する..... | 102 |
| シェーダ側にサンプラとテクスチャの設定を書き加える..... | 103 |
| シェーダリソースビュー用のデスクリプタを登録..... | 104 |
| 色化け対処 | 106 |
| ビットマップ以外への対処(今はおまけ的な話)..... | 110 |
| 3D 化してみよう..... | 111 |
| 行列について再学習..... | 111 |
| 行列による座標変換..... | 113 |
| アフィン変換(アフィン行列)..... | 114 |
| それぞれの行列を作ってみよう | 118 |
| ワールド行列..... | 118 |
| カメラ行列(ビュー行列)..... | 119 |
| プロジェクション行列(射影行列)..... | 120 |
| 行列を合成しよう..... | 121 |
| 座標変換データを GPU に送ろう..... | 123 |
| 定数バッファを作ろう..... | 123 |
| ちょっとバッファとビューについてのたとえ話..... | 128 |
| メッショの表示..... | 130 |
| メッショ(PMD)の表示..... | 131 |
| 4バイトアライメントに注意..... | 134 |
| インデックスさんデータを使って「面」を表示していこう | 140 |
| 立体感つけよう..... | 142 |
| ワールドと、ビュープロジェクションを分割..... | 144 |
| 深度バッファの冒険..... | 147 |

| | |
|---------------------------------|-----|
| 深度バッファとは..... | 147 |
| 結局 DX12 では何をしなければならないの? | 149 |
| 深度バッファの作成..... | 149 |
| 深度バッファビューの作成..... | 150 |
| パイプラインステートオブジェクトに深度情報を追加..... | 151 |
| レンダーターゲットと深度バッファを関連付け..... | 151 |
| 深度バッファをクリア(毎フレーム)..... | 152 |
| 色をつけよう..... | 153 |
| 準備 | 155 |
| DX11ほど甘くない色分け… | 158 |
| まずはディフューズ成分を GPU に投げよう | 158 |
| なんで? | 160 |
| 先人のコードを見てみよう | 160 |
| じゃあ俺実装 | 162 |
| テクスチャを読み込んでモデルに張り付けて表示しよう | 164 |
| クラス設計 | 167 |
| PMD ファイル情報からテクスチャをロード | 171 |
| じゃあテクスチャを回してみよう | 172 |
| ボーン(骨地獄)..... | 177 |
| ボーンって何? | 177 |
| スキニング(キンメッシュアニメーション)とは? | 178 |
| ボーン情報 | 179 |
| ボーン情報の「表示」..... | 181 |
| ボーン用シェーダ | 184 |
| ボーンの回転..... | 186 |
| ツリー反映の準備 | 191 |
| ツリー構造の構築 | 192 |
| 再帰 | 195 |
| 親子構造の中で複数の回転を行う | 197 |
| 最後に補足 | 200 |
| 妙なクラッシュ | 200 |
| 対処法 | 202 |
| SIMD 命令(SSE)とは | 204 |
| スキニング | 205 |
| 頂点データについて | 205 |
| ボーン ID | 205 |

| | |
|---------------------------------|-----|
| ボーン ID に悩まされる..... | 205 |
| min16uint の検証..... | 207 |
| ボーン用定数/バッファ(ボーン数×行列)を作る..... | 208 |
| 実際にポージングしてみよう..... | 212 |
| リファクタリング(コードをキレイにしましょう)..... | 215 |
| 待てあわてるなこれは vector の罠だ..... | 215 |
| 次は定数/バッファ(テクスチャも合わせるか?)周り | 219 |
| こまけ一部分..... | 223 |
| BMP 以外も読めるようにしよう..... | 227 |
| LoadWICTextureFromFile..... | 228 |
| MutiByteToWideChar..... | 229 |
| 進行とか知るか! やが! そんな事より実験だ!..... | 231 |
| LoadWICTextureFromFile の改造..... | 235 |
| sph とか spa ってなに?..... | 237 |
| せつかくだからスペキュラもアンビエントも入れる..... | 241 |
| ポージング..... | 243 |
| ウォータニオンとは..... | 244 |
| 複素数のおさらい..... | 244 |
| 四元数 | 246 |
| ウォータニオンを使用して任意軸回転する..... | 246 |
| ポーズデータのロード..... | 247 |
| 一方、クライアント側では… | 248 |
| おい、アニメーションしろよ!..... | 251 |
| どういうデータ構造にしようかな… | 251 |
| 実装 | 253 |
| 基本 | 253 |
| 指定フレームにおけるポーズを取得する..... | 254 |
| 実際にリアルタイムで動かしてみよう..... | 256 |
| 適切なフレームで適切なポーズをとるには..... | 257 |
| ポーズを補間してアニメーションしよう | 259 |
| VMD ファイルの罠に陥る..... | 265 |
| ベジエとニュートン(ニュートン・ラフソン)法と私 | 266 |
| ニュートン法の実装..... | 273 |
| ニュートン法実装後の課題..... | 276 |
| 二分法 | 278 |
| ちょっと細かいところ修正..... | 280 |

| | |
|-------------------------|-----|
| 白飛びの対処 | 280 |
| 何か透けてる | 282 |
| ちょっと横道に逸れてる | 285 |
| シェーダデザイナ | 285 |
| 影を落とす | 290 |
| 影行列 | 290 |
| ちょっとだけお悩み | 294 |
| シャドウマップ | 296 |
| その概念 | 296 |
| 演習準備 | 300 |
| さて、その意気で円柱も作っちゃおう | 310 |
| シャドウマップ実装 | 312 |
| ノッファの確保 | 312 |
| 2つの見方(深度/ノッファ/シェーダリソース) | 314 |
| 2回レンダリング(2パスレンダリング) | 316 |
| ライトの「とりあえずの」座標を決める | 320 |
| 悪夢の深度値比較つ…! | 324 |
| 深度値と距離を同じ土俵に… | 326 |
| UV値はどうするのか? | 327 |
| 比較 | 327 |
| トゥーンとかやってみよう | 333 |
| LookUpTable | 335 |
| なんかうまくいかない | 337 |
| エッジ(輪郭線) | 339 |
| 背面法(反転法?) | 341 |
| 輪郭線抽出フィルター | 348 |
| ガチでマルチパスレンダリング | 351 |
| アンチエイリアシング | 356 |
| はじめに | 356 |
| DirectX12での実装 | 358 |
| MSAA用レンダーターゲットを作る | 362 |
| 結果の検証が結構大変 | 364 |
| まとめ | 366 |
| コード | 367 |
| DDSファイルのロード | 371 |
| DXTC,ETC,PVRTC | 371 |

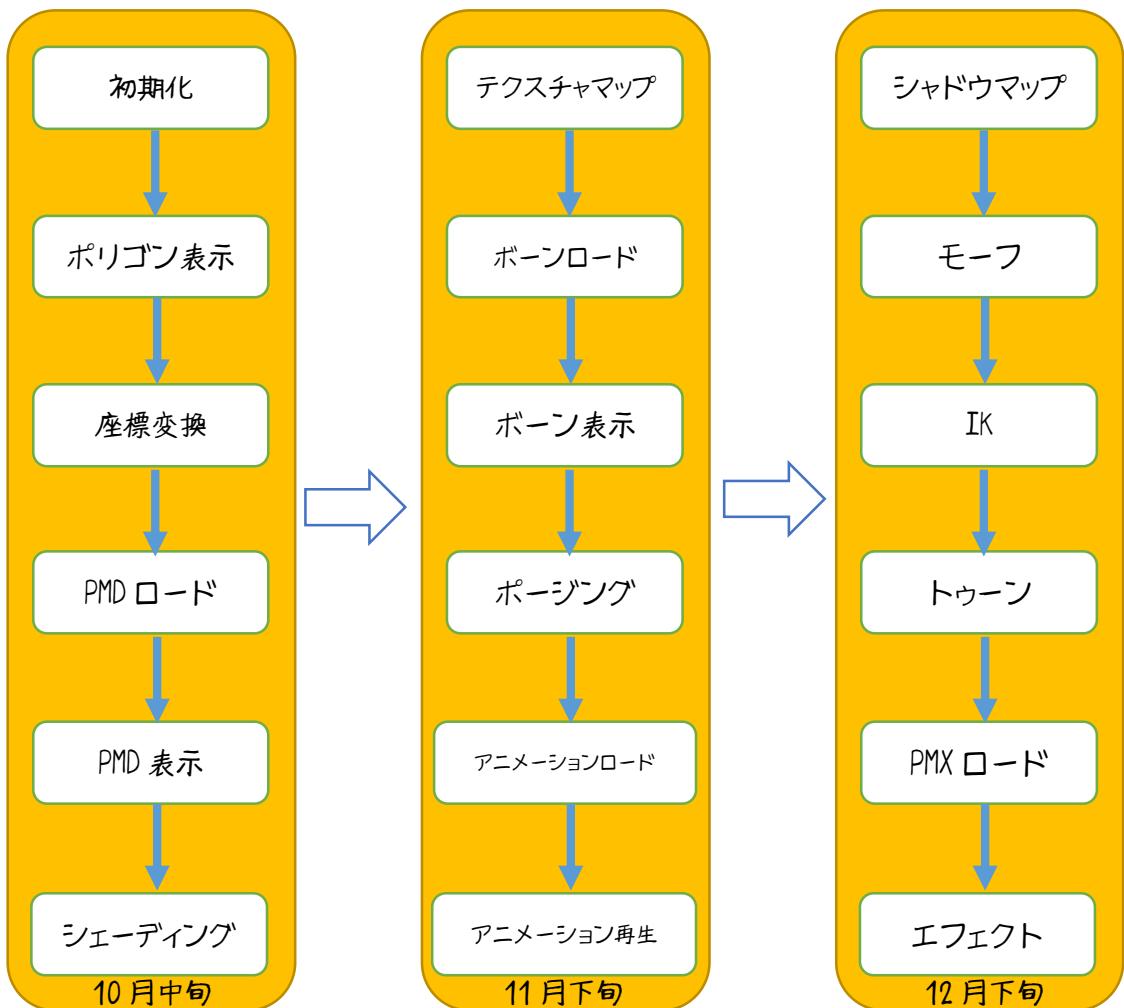
| | |
|--------------------------------|-----|
| 再び DirectXTex-master の中へ | 376 |
| クソ UpdateSubresources | 379 |
| クソに対する対処法 | 380 |
| クソまみれの中で得られたもの | 382 |
| IK(インバースキネマティクス) | 389 |
| CCD-IK とは … | 390 |
| 高校数学だけで IK っぽくしてみよう | 391 |
| 二次元における CCD-IK | 393 |
| 三次元における CCD-IK | 395 |
| LookAt 行列の作成 | 400 |
| 納得いかないんですが… | 407 |
| 夢はひろがりんぐ | 407 |
| 何で納得いくってないのか&実験 | 407 |
| まっすぐにに対する解決案 | 408 |
| LookAt 行列関数の改造 | 412 |
| いよいよ実装である | 412 |
| PMD から IK 情報を取得する | 413 |
| CCD_IK 関数について | 415 |
| CCD-IK の実装 | 417 |
| 呼び出し側 | 421 |
| 角度制限 | 421 |
| 確認コード | 424 |
| 仕上げ(軸の補正) | 426 |
| 仕上げ(ベクトル最大長による IK 位置の補正) | 427 |
| ここで今更ですがルートシグネチャについて | 430 |
| それなりにルートシグネチャについては分かってきた | 431 |
| でもでも、やっぱりさっぱりわからん人のために | 431 |
| ハッフルについて | 432 |
| デスクリプターヒープについて | 435 |
| 改めてルートシグネチャについて | 438 |
| パイプラインステート | 440 |
| 綺麗なコード? を書く | 442 |
| 課題提出のごあんない | 442 |
| 就職に向けて | 444 |

はじめに

茶番の後に「はじめに」が来るのもどうかと思うけど、最初に言いたいことを言っておきました。

とりあえずシラバズ的なものを示しておきましょう。とはいっても大体の流れがたぶん昨年と同じで対象が DX11⇒DX12 になったくらいの違いですので、流れ図は昨年のモノを流用します。

予定



こんな感じで進んでいくんじゃないかなあ…と思っております。ちなみに音の再生に関しては真面目にやると間に合いません(キチンとやろうとすると再生するだけで結構な知識が必要です)ので、既存のライブラリの CRI ADXあたりを使用することをお勧めします。

<http://www.adx2le.com/>

予定を見れば分かるように、「ゲームの制作」はこの授業の中では行いません。ですが就職活動には「ゲームの制作」は必須です。どうすればいいのでしょうか?



それはね、

この授業外で作るしかないんだよオオオオオオ

嘘じゃないし本気ですよ？そして本気でやった先輩たちはこのクソ地獄の中でも作品作つていたんです。

キツツリいかもしれないけど、今年の就職状況を見てると、この地獄を乗り越えなければ就職はかなり厳しいです。逆に言うと

「このクソ地獄をなんとか乗り越えれば、諦めなければ何とかなる。」

今期は一応授業で「チーム制作」に割り当ててますが、チーム制作をする場合も自分が何処を担当したのかを明確に言えるようじやないとせっかくチーム制作しても就職活動の助けになりませんのでご注意ください。なお、この授業で「ゲームの作り方」は教えませんので予めご了承ください。

ちなみにスマホゲーの会社などは受ける際には DX11 などでスキニングをしているだけでも高評価の対象になるっぽい（内定貰った学生談）ので、本気で死ぬほど難しいですが最低限スキニングまではついてきてください（ちなみに言うとスマホゲー会社とはいえ東証一部上場企業なので採用側の意識が高いので評価されたのも。ショボい会社の場合は評価そのものができるないんじゃないかな…。）

「だったら DX11 で十分じゃね？」確かにそうなのだが、どの道数年後には DX12 以降が当たり前になってるわけだし、いずれ移行しなきゃならない。そう考えるとアドバンテージとれる今やるべきだと思う。少なくとも俺はそう思ってる。

コンシューマゲーム会社はそこからさらに高い技術力と知識を要求されますのでそれは十分認識しておいてください（今年はコンシューマはクリエータ料と専攻料の二大巨頭才が一人ずつしかコンシューマ企業には内定していない）

そろそろこれも通用しなくなるとか思ってたけど、意外と通用してるので、現場の新人のレベルってあまり上がってないんですね…。

さて、「コンシューマの場合はそこからさらに高い知識と技術が要求される」と言いましたが、それはつまり他の学生がやってないこと…少なくとも同級生がやってないような技術を見つける必要があります。

そのためにはねえ…世の中にどういう技術があるのかを表層だけでも(用語だけでも)ざっと眺めておいて、自分の研究分野というのを持っておいたほうがいいと思います。そこまでやつてもゲーム会社への就職活動は大変なんですね。

では参考までにいろいろな用語を書いておきます。ぶっちゃけ年々増えていますので、大変だなあって思います。たぶんみんながこれを読んでいる間にまた用語が増えちゃう…そういうもんです。

膨大なる用語(本当にすまない)

- デザインパターン
- 関数型プログラミング
- クロージャ
- スマートポインタ
- STL
- MVVM
- WPF
- C++以外でよく使用される言語(C#, python, javascript)
- Git/Subversion
- ガベージコレクション
- マルチスレッド
- スレッドセーフ
- ライブラリ/リンク
- コンパイル/リンク/ビルド
- DCCツール
- DX12/Vulkan/Metal/OpenGL ES
- ディフューズ/アンビエント/スペキュラー
- ランダート反射

- 頂点シェーダ(VS)
- ピクセルシェーダ(PS)
- フラグメントシェーダ(FS)
- ジオメトリシェーダ(GS)
- ハルシェーダ(HS)
- コンピュートシェーダ(CS)
- HLSL/GLSL
- トゥーンレンダリング/セルシェーディング
- レイトレーシング
- レイマーチング
- ボリュームレンダリング
- コマンドバッファ
- テクスチャ
- ミップマップ
- 隠面消去
- カリング
- ラスタライズ
- テッセレーション
- UV
- 法線ベクトル/接線ベクトル/従法線ベクトル
- Z/バッファ/深度バッファ
- ddx/ddy(偏微分)
- ステンシルバッファ
- パンチスルー
- ディザパターン
- サンプラー
- GPU
- レンダリングパイプライン
- トゥーンレンダリング
- モーションブラー
- ボーン
- スキニング
- IK(インバースキネマティクス)
- ノーマルマップ
- ファー
- アンビエントオクルージョン

- ディファードレンダリング
- シャドウマップ
- VSM(バリアンスシャドウマップ)
- 環境マップ
- ディスペレスメントマッピング
- サブサーフェススキャッタリング(SSS)
- サブディビジョンサーフェス
- 平行光線/点光源/スポットライト
- フレネル反射
- スネルの法則
- デプスフォグ
- LUT(ルックアップテーブル)
- カスケードシャドウマップ
- PBR(物理ベースレンダリング)
- NPR(ノンフォトリアリスティックレンダリング)
- アルベド/ラフネス/メタリック
- プロシージャルOO(例:プロシージャルテクスチャ)
- HDR/SDR
- ガンマ
- BRDF
- パーティクル
- クオータニオン
- 球面線形補間
- マイクロファセット関数
- 画角
- パースペクティブ
- メモリ/グラボ(VRAM)
- GeForceGTXOO
- RadeonOO
- ナビゲーションメッシュ
- 遺伝的アルゴリズム
- A*アルゴリズム
- ビヘイビアツリー
- ディープラーニング
- $\alpha\beta$ 戻り
- HSB/HSV

- 色相環/表色系
- RGB/sRGB/adobeRGB/CMYK
- 同期/非同期
- TCP/UDP
- NAT
- サーバークライアント/P2P
- クラウド
- AWS
- サーバーの垂直分割/水平分割
- データベース/クラスタ/キー
- オーサリングツール
- KPI(←ソシャゲプランナー向け)
- ユーザビリティ
- ユーザーエクスペリエンス(UX)
- アクションとリアクション/官能性
- VR/AR/MR

最初のほうの用語はプログラミングにおける用語で、その次がCG用語(僕がこっち系だからこれが多いね)その後あたりで基本的な用語が来て、基本的なグラボが来て、人工知能系の用語が来て、その後は色彩的な用語が来て、そのあとはネットワークとかインフラ回りの用語が来て、最後はソシャゲとかの企画の用語で、VR/AR/MRは、最後の奴はMR(複合現実)ってやつです。多くて本当に申し訳ない。



とは言えすべてを知っておく必要があるわけではなく、次年度の皆さんには特に、この中から自分の研究課題を決めてゲームを作りながら取り組んでほしい。

とてもじゃないけど出来そうにないと思うかもしれないけど、やりようによってはできます。コンシューマ狙うなら本当に頑張ろう!!!

とは言え授業でこの半分くらいのところは網羅するとは思いますので、プラスアルファくらいで考えてください(楽ではないですが)

ところで僕はCG系が一番教えて、AI系はちょっとだけ教えて、ネットワークやインフラ系は正直期待しない!もうがれい!です。全部はやれないつす。あとゲームエンジンに関しては「ワタシ、ユニティ、チョット、デキル」、「ワタシ、アンリアル、チョット、デキル」程度なら分かりますのでそっち系で分からぬことがありますたら一旦はご質問ください。

ちなみにDX12ができればDX11は余裕だと思いますので、ここを乗り越えれば自信もっていいと思いますよ。少なくとも現段階のスマホゲー会社は何とかなるんじゃないでしょうか。とはいってもレベルが低すぎる会社だとそこに重点を置いてない可能性もあるので、そこは戦略的に進めるべきです(Aiming以上の会社なら評価されると思いますが、地方の某社未満(何處とは言いません)の会社では評価されないと思いますので気を付けてください)

モチロン、ここで言ってきたことは、現役のゲームプログラマでもない僕が言っている事なので、実際にはさらに進んでいるだろう。という事で現在の技術トレンドにはアンテナを張っておいてほしい

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

ここには最先端の資料が置かれています。全てではないのである程度分かるようになつたら現行のゲームをよ～～く観察して「どのように作られているのか」を推測する力をつけて、実際に同じものを実装してみてください。今僕が実装したいのはコレ…

<http://tech.cygames.co.jp/archives/2987/>

雲表現ですね。リアルタイムにレイマーチングでボリュームレンダリングするんだから工夫しないとまずまともな速度は出ないはず。まずその遅さを実感してみて、そつから速くしていきたい。仕事しながらだとなかなか作らないので、福工大の八耐とかの機会に作ってみようと思っている。

cygames社内では、開発者教育が盛んなようで、頻繁に勉強会が行われているようです。

<http://tech.cygames.co.jp/>

ここに次から次に資料が上がっています。皆さんへのお勧めは資料はコレ

<http://tech.cygames.co.jp/archives/2617/>

<http://tech.cygames.co.jp/archives/2621/>

<http://tech.cygames.co.jp/archives/2430/>

まあ、これくらいは読んでおいてください。また、専攻科3年とかはもっと攻めてほしいので物理ベースレンダリングの話

<http://tech.cygames.co.jp/archives/2129/>

<http://tech.cygames.co.jp/archives/2296/>

<http://tech.cygames.co.jp/archives/2339/>

<http://tech.cygames.co.jp/archives/2488/>

や

<http://tech.cygames.co.jp/archives/2484/>

<http://tech.cygames.co.jp/archives/2487/>

などを読んで、研究に活かしてほしいかなーと思っています。グラフィクスに偏ったのでAIとかも

<http://tech.cygames.co.jp/archives/2272/>

<http://tech.cygames.co.jp/archives/2364/>

<http://tech.cygames.co.jp/archives/2853/>

その他

<http://tech.cygames.co.jp/archives/2843/>

<http://tech.cygames.co.jp/archives/2820/>

<http://tech.cygames.co.jp/archives/2259/>

<http://tech.cygames.co.jp/archives/2937/>

<http://tech.cygames.co.jp/archives/2950/>

<http://tech.cygames.co.jp/archives/2961/>

<http://tech.cygames.co.jp/archives/3009/>

<http://tech.cygames.co.jp/archives/3027/>

資料の量がすごいですね。なおこういう資料を公開しているのは Aiming や KLab も同様なので、あっちも見ておくといいとも思います。

Aiming

<https://developer.aiming-inc.com/>

KLab

http://www.klab.com/jp/technology/archive/contents_type=28

また、SlideShare はこういうのの宝庫です。

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

ただ、目的のモノを探すのは大変です。資料をよく公開している個人を特定し、その人の名前で検索すればいくつか出でてきます。

大圖衛玄氏

https://www.slideshare.net/MoriharuOhzu?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview

内村創氏

https://www.slideshare.net/nikuque?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview

三宅陽一郎氏

https://www.slideshare.net/youichiromiyake?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview

などです。他にももっとありがたい資料がたくさんあると思いますが、そこは自分で調べてください。

長々と書きましたがまずは環境設定からやっていきましょう。

環境設定

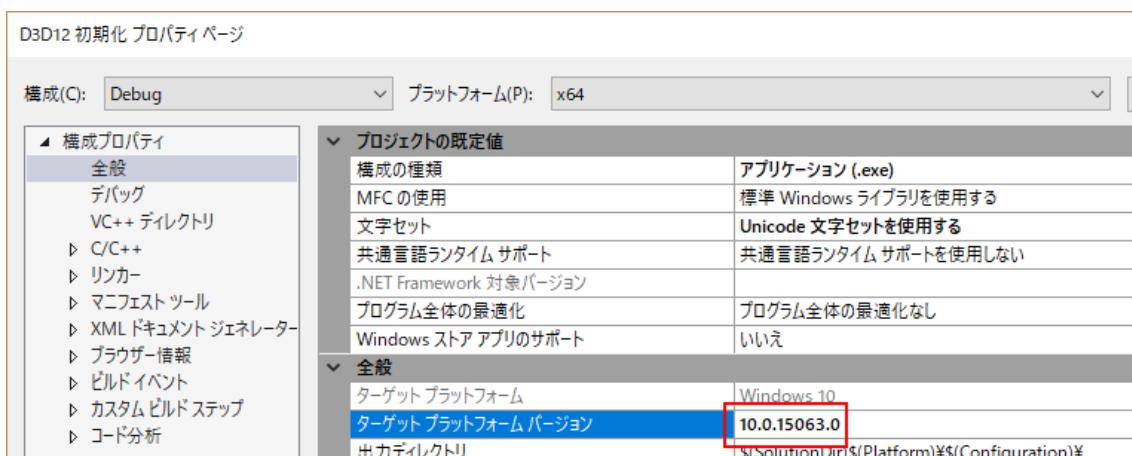
DirectX12 を使えるようにするまでの環境設定が結構ややこしい。

DirectX12 を使用する時に助けになる d3dx12.h が標準で入っていないのである…。対処法としては…

- ①使わないで、別の便利ライブラリを使用する
 - ②落としてきて使えるようにする
- があり、ちなみにいうとターゲットプラットフォームバージョンによって対処が変わってくる。

まず確認してほしいのですが、Visual Studio で適当な C++ プロジェクトを作った時のプロジェクトの設定を見てみよう。

まず「全般」の「ターゲットプラットフォームバージョン」を見てください。



自分のここがどうなっているのか確認してください。10.0～になつていないと、DirectX12 の開発ができません。

皆さんの PC がどうはは分かりませんが、下手をすると 8.1 までしかない可能性があります。この場合…面倒なのですが、最新版の Windows SDK をダウンロードしてインストールするか、
<https://developer.microsoft.com/ja-jp/windows/downloads/windows-10-sdk>

プロジェクトの新規作成 → Visual C++ → Windows 10 / ユニバーサルなんかをクリック → インストールが始まります。

最新版 Windows SDK なら 10.0.15063.0 になりますし、ユニバーサル何とかなら、10.0.14393.0 になります(この 15063 は VS2017 じゃないとまともに動かないるので、今回は 14393 にしつくのです)。

| |
|----------------------|
| 10.0.14393.0 |
| 10.0.15063.0 |
| 8.1 |
| <親またはプロジェクトの既定値から継承> |

この辺の作業は、最初からターゲットプラットフォームバージョンに 10 番があれば必要ない作業ですが、如何でしょうか？

ほんまは一番いいのは最新版の WindowsSDK をインストールしてから、さらに言うと VisualStudio2017 にすることです。まあ学校の PC はそうそう新しい VS をインストールするのも大変なので、希望者は個人でやっておいてください。管理者権限が必要であればそこは対処します。

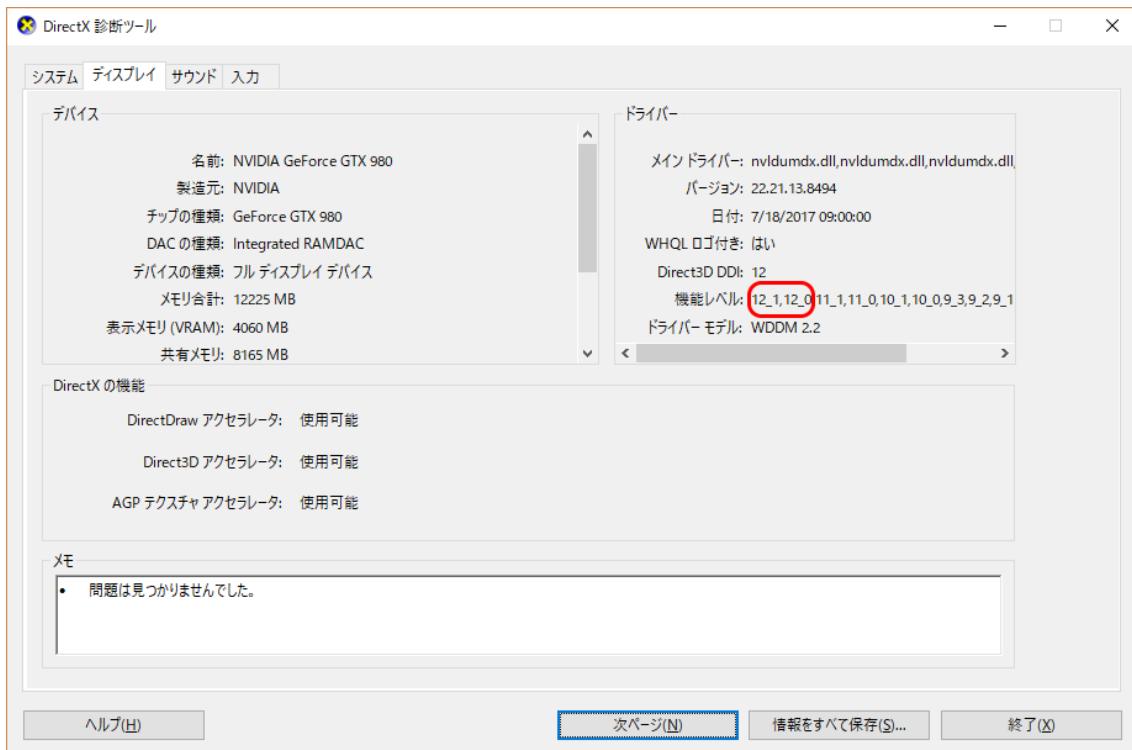
少なくとも家の PC は最新版でやっておくことをお勧めします。が、授業では 10.0.14393.0 をベースに進めていこうと思います。

一応、DirectX のランタイムのバージョンは dxdiag で確認できます。学校のは基本的には DirectX12 になっているはずです。

で、ここで問題というか、残念なことが発覚… DirectX12 じたいは使えるのですが、ひとまずウインドウズキーを押して、dxdiag と入力してみてください。



こんなのが出てくると思います。DirectX バージョンが DirectX12 である事が分かると思います。では「次ページ」を押してください。



デバイスとか、ドライバーの状況が表示されると思います。ドライバーの機能レベルを見てください。ここに DirectX の「フィーチャレベル」というものが表示されているはずです。

僕の家の PC は 12_1,12_0 までサポートしているのですが、この教室の PC も隣の教室の PC も、機能レベルが 11_1 が最高なのです。だいたい GeforceGTX980 未満の PC やモバイル系だとこうなっています。仕方ないです。

つまり DirectX12 から搭載された機能が使えないわけです。まあ、シェーダ機能部分以外は DirectX12 なので、このままでいますが、初期化の時にちょっと注意が必要になりますので、心に留めておいてください。

ウインドウを出すまで

DxLib を使用せずにウインドウを出すには [Windows.h をインクルード](#)する必要があります。手順としては

1. Windows.h をインクルード
2. アプリケーションインスタンスハンドルを取ってくる
3. ウィンドウを作る準備をする
4. ウィンドウを作ってウィンドウハンドルを取得する

5. ウィンドウを表示する
6. すぐにウィンドウが閉じないようにメインループで止めておく

こんな感じです。簡単でしょ?とは言いません。それなりに面倒です。何しろ今まで DxLib_Init() で済ませていたのですから。

※あと、_T("") がコンパイルエラーを起こすことがありますので、その場合は tchar.h をインクルードししてください。

まずアプリケーションインスタンス"ハンドル"って何だ?

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

ウィンドウアプリケーションなら

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

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

軽く理由を説明しておくと…

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

で、そのデバイスの一つであるディスプレイに「ウィンドウ」を表示するのは OS の役割であり、OS にその仕事をさせるためには「持ち主は誰か」を OS に教えておく必要があるのです。

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

ちなみに DirectX ってのはこの OS がやっている仕事を DirectX が一部「ぶんどってドライバ

に対して直接命令を出し、より高速に描画処理をするためのものです。

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

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。

ちょっとここでいい機会なので、僕の授業を受けるときの鉄則を書いておきます。

鉄の掟

- マニュアルは必ず読む(MSDNなどの信頼できる物を必ず隅から隅まで読んでください)
- 分からなかつたらすぐ聞く(先生でも友人でもいいので、分らないままにしない事)
- 休まないよう(基本的に、休むとワケ分らない事になります。僕も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)
- 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ヨボくなりがちです。

さて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)
- 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。

ひとまずはメイン関数にこの通りに打ち込んでください。

ウィンドウクラス登録

```
WNDCLASS w = {};
w.lpfnWndProc = (WNDPROC)WindowProcedure;//コールバック関数の指定
w.lpszClassName = _T("DirectXTest");//アプリケーションクラス名(適当でいいです)
w.hInstance = GetModuleHandle(0);//ハンドルの取得
RegisterClass(&w); //アプリケーションクラス
```

ウィンドウサイズ設定

```
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補正する
```

ウィンドウ生成

```
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
NULL, //親ウィンドウハンドル
NULL, //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
NULL); //追加/ラメータ
```

ウィンドウ表示

```
ShowWindow(hwnd, SW_SHOW); //ウィンドウ表示
```

ちなみに僕はウィンドウアプリでもコンソールウィンドウやっちゃう方(テキスト出力とかもしやすい)なので GetModuleHandle を使用していますが、皆さんはこの通りにする必要はありません。WinMain からやっても大丈夫です。

で、どうせ「WindowProcedure が存在しない」とか言ってエラーが出るので、関数を作ってください。こんな感じでいいです。

```
//めんどくせーし、あまりゲームに関係ないけど書きかなあかんやつ
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); //既定の処理を行う
}
```

さて、ここまでが書いて、コンパイルが通ったら実行してみましょう。うまく書ければ「ウインドウ」が表示されるはずです（一瞬だけ）

もしウインドウが表示されない人がいたら言ってください。たいていはウインドウハンドルが取得できていないか、クラスの登録ができるないかです。失敗してたら CreateWindow の戻り値が nullptr になってしまふはずです。

ウインドウ表示するだけでこの手間なのよ～!!!

あとちなみに上のコードを覚えようとする人がいますが、必要ありません。プロでも覚えてないでいいです。ただ「何をやってて、何が必要か、表示するまでにどういう仕組みになっているのか」をだいたい把握してればOKです。

さて、表示はしたものの一瞬で消えるためウンドループを作つてあげなければなりません。この辺は DXLIB と同じですね？

ただ DXLIB と違つて ProcessMessage 関数ではなく、別のやり方を行います。あの ProcessMessage は中でモノごつつい色々やってんのよね…。まあそれはともかくゲームループ回すだけなら

```
MSG msg = {};
while (true) { // 基本無限ループ
    if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) { // OSから投げられてるメッセージを msg に格納
        TranslateMessage(&msg); // 仮想キー関連の変換(ぶっちゃけゲームには関係ない)
        DispatchMessage(&msg); // 処理されなかつたメッセージを 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;
    }
}
```

で十分。ちょっとこれでウンドウを出してみてください。そして×ボタンを押して閉じてみて下さい。きちんと終了しましたか？

さて、ここまで色々と関数が出てきたのでささっと確認しましょう。

RegisterClass 関数

<https://msdn.microsoft.com/ja-jp/library/cc410975.aspx>

(。・ω・)ん？

『RegisterClass

<https://msdn.microsoft.com/ja-jp/library/ms633576.aspx>

ウィンドウクラスを登録します。同様の機能を持つ RegisterClassEx 関数をお使いください。

なん…だと? いつから RegisterClass は推奨されなくなったのだ…。という事を確認するためにもマニュアルは見ておくべきなのです。

という事で RegisterClassEx を使用するように書き換えてみます。

```
WNDCLASSEX w = {};
w.cbSize = sizeof(WNDCLASSEX); //これ、何のために設定するのさ…?
w.lpfnWndProc = (WNDPROC)WindowProcedure; //コールバック関数の指定
w.lpszClassName = _T("DirectXTest"); //アプリケーションクラス名(適当でいいです)
w.hInstance = GetModuleHandle(0); //ハンドルの取得
RegisterClassEx(&w); //アプリケーションクラス
```

ほぼ変わってないんですが、何故か WNDCLASSEX のサイズを入れとかないとウィンドウ生成に失敗します。これは意義するところがよくわからんです。

改めて RegisterClassEx のマニュアルを見ましょう。

<https://msdn.microsoft.com/ja-jp/library/cc410996.aspx>

アトム(ATOM)がどうこう言ってますが、



ぶつちやけまあ要らないです。ただ、UnregisterClass で、クラスは破棄しておきましょう。

<https://msdn.microsoft.com/ja-jp/library/cc364845.aspx>

ちなみに今回出てる「クラス」は C++ の「クラス」とは別モノなので混同しないようにしてください。名前同じだから混同すると思うけど、とにかく違うって認識でお願いします。

AdjustWindowRect

<https://msdn.microsoft.com/ja-jp/library/cc430250.aspx>

例えば 640,480 で指定した場合、ウィンドウのサイズを 640,480 にしてしまうと、ウィンドウの表示部分が少し小さくなってしまいます。どういう事がというと、タイトルバーと枠線のぶんだけ表示領域が小さくなるんですよね。

なので、そこを考慮してサイズを計算しなおしてくれるのが、この AdjustWindowRect なのです。

CreateWindow

<https://msdn.microsoft.com/ja-jp/library/cc410713.aspx>

引数の部分もしっかり読んでください。そして今回の引数がどういう意味を持つてそういう値を入力されたのかを考えながら読みましょう。たぶん今のレベルで全部理解するのは難しいでしょうが、理解しようと思ってみてください。

あと、書き忘れてましたが、ウインドウクラスの中の

w.lpfnWndProc = WindowProcedure

ですが、これ、前にも言ったように、ウインドウの表示もウインドウの×ボタンを押したりキー入力したりも OS を介してやってるんですが、その応答をどうするかという事で、関数ポインタを渡してるんですよね。こういうのを『コールバック関数』といいます。

で、OS は何かしらイベントが発生したらこの関数ポインタへ処理を投げる。そういう仕組みになってるんだ。ちなみにここを nullptr にしたらどうなるんだろう…クラッシュします。そういう事です。

で、ShowWindow で表示する…と。

<https://msdn.microsoft.com/ja-jp/library/cc411211.aspx>

で、いろいろとパラメータがあるんですが、SW_SHOW を指定します。

色々変えてみて、どうなるのか見てみましょう。

で、次に無限ループの部分ですが PeekMessage を見てみましょう。

<https://msdn.microsoft.com/ja-jp/library/cc410948.aspx>

似たような名前で DxLib の PostProcessMessage がありますが、あれソースコード見ると予想以上に色々とやっています。なので、名前は似ていますが、ちょっと違うと思っておいてください。

ともかく PeekMessage を見てみましょう。

『着信した送信済みメッセージをディスパッチ(送出)し、スレッドのメッセージキューにポスト済みメッセージが存在するかどうかをチェックし、存在する場合は、指定された構造体にそのメッセージを格納します。』

なんだこの『「アルシのルシがコクーンをページ』的な文章は…。本当に MSDN はこういう文章

が多いいんですが、我慢して読むんだ。理解しなくていいから。

ともかく今のウインドウに対してなんか変化があつたら、OS が「メッセージ」って奴を飛ばすんだ。でも一度にドドドッと飛んでくることがあるんだけど受け取れるのは一度に1つずつなので「キュー」という所に溜まっていく(正確にはデータ構造が Queue 型のメモリに溜まっている)。で、この ProcessMessage ってのは、そのキューにメッセージが溜まっているかどうかを確認し、溜まつていれば先に格納されたものからメッセージを取り出します。

なのでこれを呼び出した後は第一引数の msg にウインドウズからのメッセージが入っていきます。入っていない時は 0 が返ります。

…つまるところ、これを呼び出すと msg に OS からのメッセージが入ります。

TranslateMessage ですが、

<https://msdn.microsoft.com/ja-jp/library/cc364841.aspx>

「仮想キーメッセージを文字メッセージへ変換します。文字メッセージは、呼び出し側スレッドのメッセージキューにポストされ、次にそのスレッドが GetMessage または PeekMessage 関数を呼び出すと、その文字メッセージが読み取られます。」

これも「何じゃらほい」って感じで、そもそも「仮想キー」メッセージって何やねん。と思う。おそらくはキーボード入力処理において、ドライバから飛んできたキーボードイベントで取得したそのままのキーコードをプログラマがわかる「仮想キーコード」に変換した上で msg に格納します。ぶっちゃけゲームには要らないかもーって思う。WM_KEYDOWN など、キーボード関係のイベントを処理するなら必要なんですがね。

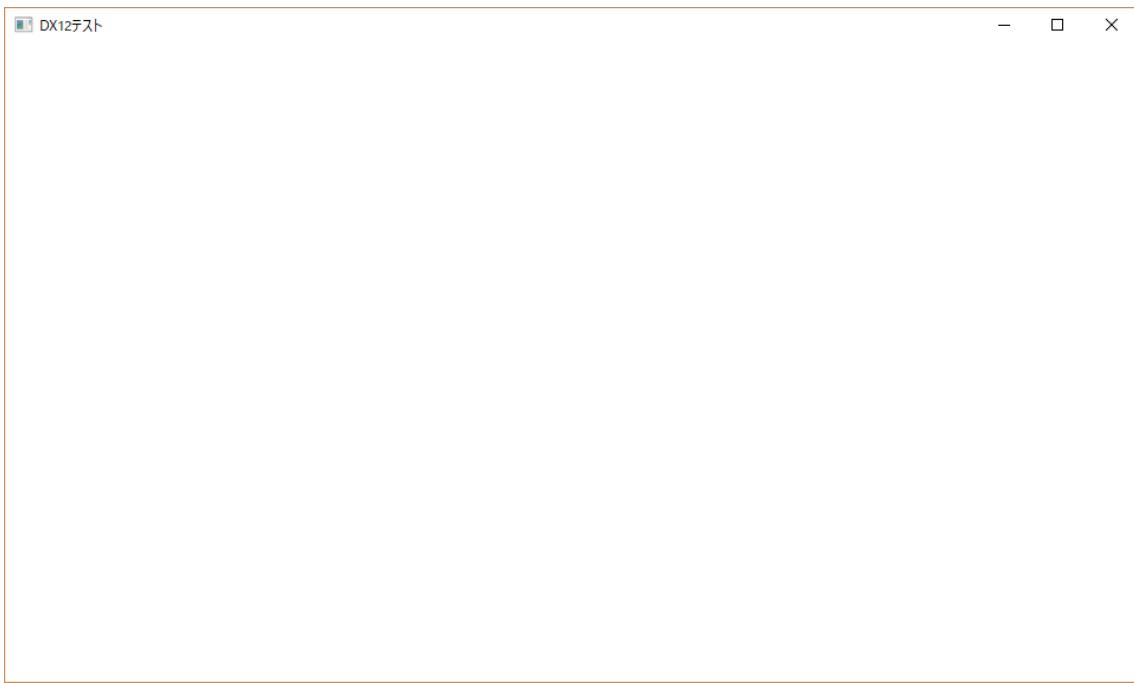
次に DispatchMessage

<https://msdn.microsoft.com/ja-jp/library/cc410766.aspx>

この処理はウインドウプロシージャ(WindowProcedure)にメッセージを投げます。で、投げるんだから普通に WindowProcedure をコールすればいいんですけど、そこはまた OS を介して投げたいので、そのため DispatchMessage という関数があるという認識でよいと思います。あまり深くまで理解しようとすると OS 作るレベルの理解が必要なのでそこはこの程度の認識で OK。

ちなみに msg.message==WM_QUIT はウインドウを破棄するときに発生するのでこのタイミングでブレイクしてループ抜け→アプリケーション終了します。

普通に



って出て、右上の×を押してコンソール画面ごと落ちれば大丈夫です。

Direct3D の初期化

最初にも書いてたが、そもそも必要なヘッダファイルが相當に増えている。DxLib の時は DxLib.h をインクルードすれば良かつたのだが…DirectX12 にするとこうなる。

```
#include <windows.h> // ウィンドウ出すのに必要  
#include <d3d12.h> // DirectX12を使うのに必要  
#include <d3dx12.h> // DirectX12を若干使いやすくするためのヘッダ  
#include <dxgi1_4.h> // DXGIを扱うのに必要(DX12ではDXGI1.4が使われてる)  
#include <D3Dcompiler.h> // シェーダコンパイラ(シェーダ解釈回り)で必要  
#include <DirectXMath.h> // 数学系の便利なのが入ってるヘッダ
```

ちなみに d3dx12.h は最初から環境にインストールされているわけではなく、自分でとつてこなければならぬ。別に取ってくるのが必須ではないがそれはそれで大変だと思います。



で、ここで(特に家でプログラミングするときの)注意点なんですけれども、前述したプラットフォームバージョンによって d3dx12.h のバージョンも変わるんですね。

ちなみに 10.0.14393.0 の場合だと、最新版の d3dx12.h ではコンパイルエラーが出るんですよ。ここで GitHub から特定のバージョンを落としてこなければならないのですが、それなりに難しいんですね…(ちなみにプラットフォームが 10.0.15603 以降ならば最新版でも OK)

で、たぶん皆さんは Git とか一部の人しか使ったことないですよね? 使えと言わなくとも使っておいてもらえると助かるんだけどね。授業の時間は限られてるし、それ以外にも教えるべきことは多いしですね。

長々と書きましたが、つまるところ半数の人は困難だと思いますので、サーバに 10.0.14393.0

用の d3dx12.h を置いておきます。

[¥132sv¥gakuseigamero¥rkawano¥DirectX12¥d3dx12.h](#)

今回の開発で覚えといてほしいのは、環境設定…大変だろ？意外と死ねるんだこれが。実際仕事の時はこの環境構築が思いのほか時間を食ってしまうので気を付けておきましょう。学校だとこうやってセンサーがやっておいてくれたりするんだけど、ゲームプログラマはこういう事を全て自分でやらなければならぬんで大変なのよ？

情報収集から何から自分でやらなきゃいけないのよ？お金を貰うってそういう事なのよ？動かなくても労力に見合わなくても誰にも文句は言えないのよ？次年度就職の人たちはキモに銘じておくように。

さて、実際に上のようにインクルードを行って、特にエラーが出なければそのまま進んでいきましょう。

Direct3D の初期化について

さて Direct3D についてですが、以前にもちょっと書きましたが、本来は Windows OS がデバイスドライバに対して働きかける部分を一部ぶんどってやるための仕組みです。というわけで必ず必要なのはまず

Direct3DDevice

『Direct3DDevice』という、DirectX がデバイスにアクセスする部分のインターフェイスですね。これを作らないとどうしようもないです。ちなみにこの『Direct3DDevice』は DirectX9 時代からあるもので、昔はこれさえ初期化すればよかったんですよねえ…(遠い目) どちらにせよ DirectX12 でも必要なので作っちゃいましょう。

さて、デバイスの作成は DirectX12 の場合は D3D12CreateDevice という関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770336\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770336(v=vs.85).aspx)

現在のところ英語ドキュメントしかありません。



だが、やるしかない(ああ、周囲のプロゲームプログラマが揃いも揃って手を出してないのは

そういうことがあ…)

頑張って読んでいこう。Parameters ってのが引数の説明です。IUnknown ってのが第一引数だが、Pass NULL to use the default adapter,

とか書いてるので、ここは nullptr にしておいて、デフォルトのアダプタを使わせていただこう。

ちなみに「アダプタ」ってのは今は「物理的な意味での」ディスプレイだと思っておいてください。

では次に D3D_FEATURE_LEVEL だが、これは学校の PC においては 11_1 にせざるを得ない。

次の引数だが

Type: REFIID

The globally unique identifier (GUID) for the device interface. This parameter, and ppDevice, can be addressed with the single macro IID_PPV_ARGS.

等と書いてある。GUID を指定しろとある。分かんねーよそんなもん。なんだが、もう少し読んでいこう。

「このパラメータとデバイスは IID_PPV_ARGS によって対処できる」ってな感じかな。

ということで IID_PPV_ARGS を見てみよう。

[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)

Used to retrieve an interface pointer, supplying the IID value of the requested interface automatically based on the type of the interface pointer used. This avoids a common coding error by checking the type of the value passed at compile time.

長いので Google 翻訳にかけてみる。

「使用されたインターフェースポインタのタイプに基づいて、要求されたインターフェースの IID 値を自動的に提供するインターフェースポインタを取得するために使用されます。これにより、コンパイル時に渡される値の型をチェックすることにより、共通のコーディングエラーを回避します。」

どうやあ…？まあつまるところ、ポインタの型から判断して IID 値を自動的に計算しそれによりエラーを回避できるという。それは分かった。そのあと、の説明を読んでみよう。

Parameters

pType

An address of an interface pointer whose type T is used to determine the type of object being requested. The macro returns the interface pointer through this

parameter.

Return value

This macro does not return a value.

ひとまずここまで読んでみよう。インターフェースのポインタを入れるとある。でその型が要求されているもの。このマクロはインターフェースポインターを返す。

また、このマクロは値を返さない。お前は何を言っているんだ？返すって言ったり返さないって言つたり情緒不安定か？

これはよくわからないので定義を見てみよう。

```
#define IID_PPV_ARGS(ppType) __uuidof(**(ppType)), IID_PPV_ARGS_Helper(ppType)
```

なるほど。ロクでもない説明より余程分かりやすい。

つまり、ポインタを渡すと

ポインタの型に応じた GUID，適切にキャストしたポインタ

に変換してくれます。

つまり、第3引数と第4引数をまとめて

IID_PPV_ARGS(ポインタ)

で済ますことができるわけだ。

つまり

```
HRESULT
```

```
result=D3D12CreateDevice(nullptr,D3D_FEATURE_LEVEL_11_1,IID_PPV_ARGS(&dev));
```

といいうわけだ。ひとまずこのまま実行してみてくれ。あ、dev の型を言っておかなかった。しかしこの辺は自分で判断できるようになっておいてほしい（マニュアルに書いてあるので）が

```
ID3D12Device* dev=nullptr;
```

である。

さて、実行してみてくれ。

おや？ リンカエラーがでますか。

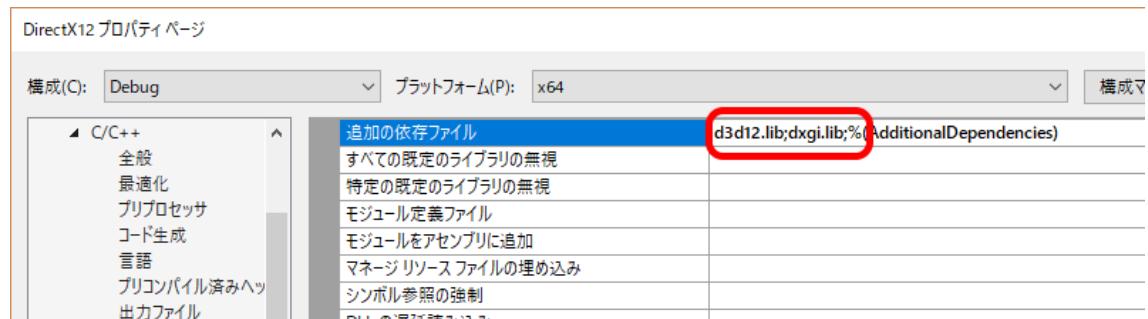
LNK2019 未解決の外部シンボル _D3D12CreateDevice@16 が関数 _main で参照されました。

なるほど。この手のリンカエラーが起きたら lib ファイルをリンクできていないのだなと判断できるようになっておこう。

これをリンクするには二つのやり方がある。まず一つは、インクルード文の後くらいで、関数の外側で

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

と書いておくか、もしくは、プロジェクトのプロパティの追加の依存ファイルに



としておくとよいでしょう。両方やっても意味ないのでどちらかにしておいてください。

とりあえず、先ほど書いたコードの result が S_OK になっているのを確認しよう。もしできてなければどこか間違っています。頑張ってください！

ともかくこの CreateDevice を呼べるようにしようとしてここまで書いてきたが、これでは将来性がない。つまり 12 が大丈夫なマシンに行っても 11 が選ばれてしまう。これはよろしくない。

11 までだったらデバイスを nullptr にして投げれば最高バージョンが取得できただが、12 バージョンではそれができないっぽい。今のところ。

例えばこのように…

```

D3D_FEATURE_LEVEL levels[] = {
    D3D_FEATURE_LEVEL_12_1,
    D3D_FEATURE_LEVEL_12_0,
    D3D_FEATURE_LEVEL_11_1,
    D3D_FEATURE_LEVEL_11_0,
};

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

```

という感じで一番レベル12の機能が取れるようにしておいた方がいいですね。本当はこういうのも自分で思いつけるようになって欲しいんですが現状ではなかなか難しいでしょうし、なぜこのようなコードになっているのかを各自考えてください。

ともかくこれでデバイスの作成はできたのでメインループ抜けた後に
`dev->Release();`
と書いておきます。`Release()`はそのオブジェクトを解放するという事です。ちょっとわけあって`delete`ではないのです。DirectXはこんなのが多いので注意してください。

で、デバイスを作ったらそれで終わりかというとそうではなく、画面に色々と表示するために
は

- スワップチェイン
- レンダーターゲット

が必要で、さらに DirectX12からは

- コマンドアロケータ
- コマンドキュー
- コマンドリスト

などが必要で、さらに

- ディスクリプター(DX11 時代でいう所のビュー)

というのも出てきます。まあこいつは DirectX11 時代の「ビュー」を知つてればそれほど理解は

難しくないのですが、所見だと意味わからないと思います。
そのほかにもエンスだのドリアだのマルチスレッド関連のやつがパンパン出てきますので、完璧に理解する必要はないけど、頑張って、頑張って全体的なイメージは把握しておいてください。

コマンドまわり

実はこの辺の考え方は OpenGL に近いのですが OpenGL にも「コマンドバッファ」という考え方があり、(恐らくは)それに近い考え方だと思います。

DX12 には「コマンドリスト」というものがあり、それが OpenGL のコマンドバッファに当たるものと考えられます。

軽く説明しつゝ、グラフィックス周りの命令は「命令⇒即実行」ではなく「命令⇒命令をどつかに溜め込んでおく⇒一気に実行」

という仕組みで動きます。

ドラクエとか初期の FF のようなコマンドバトルのコマンドを作るときのことを考えてみてください。



例えば「たたかう」コマンドを入れたら即攻撃ではないですよね? 即攻撃になってほしいのは格闘ゲームとかのアクションゲームの時で、↑のようなゲームの場合は一旦全員のコマンドを入力すると一気にコマンドが実行されましたよね?

DirectX12 における「コマンド」もそういうイメージで考えてください。

なんでそういう面倒なことになっているのかというと、描画周りの命令ってのは GPU ハックセスします。ところが GPU へのアクセスはいつでもやっていいわけではありません(ディスプレイへの描画とかしてたりするので)。

ということで命令を受け取るにはディスプレイのロックを行い、グラフィックメモリを書き込み可能な状態にして、そこで初めて命令を受け取ることができます。そして命令を受け取ったら書き込み不可に戻して、ディスプレイへの描画を行います。

そして GPU はロックしたりロック外したりやるわけですが、このコストが結構高い。つまり処理時間はかかるしグラボの寿命を縮めたり発熱を促進したりするわけです。

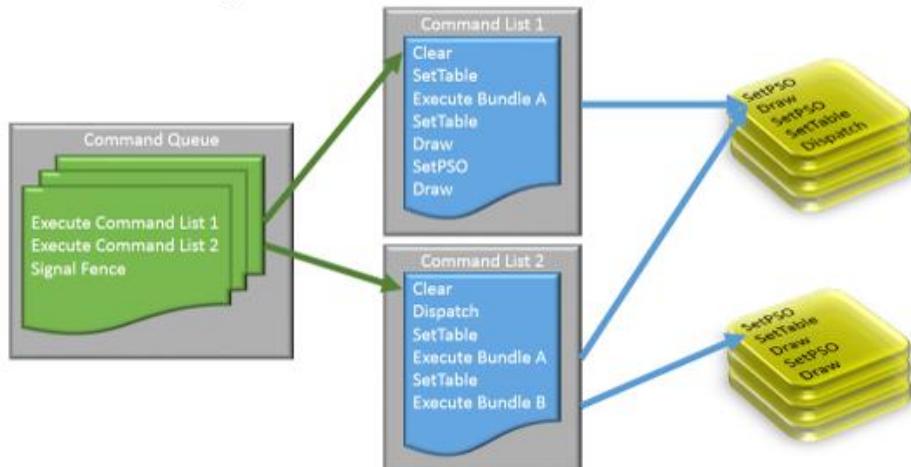
そういう理由があって、命令をため込んでおいて一気に GPU に転送するために「コマンド バッファ」を使用するわけですが、その名前が「コマンドリスト」になっているとでも思っておけばいいです。

で、ここがちょっとややこしいんですけど、コマンド周りでは
「コマンドリスト」
「コマンド キュー」
ってのがあります。アルゴリズムとデータ構造が分かっている人ならあれ?って思うでしょう。

コマンドのリストとコマンドのキュー。どちらもデータ(命令)を溜めていくために使用されるという事は予測できます。

ちょっとこの辺周りを説明するものとして、拾ってきた画像があるのですが、こういう構造のようです。

Command Queue

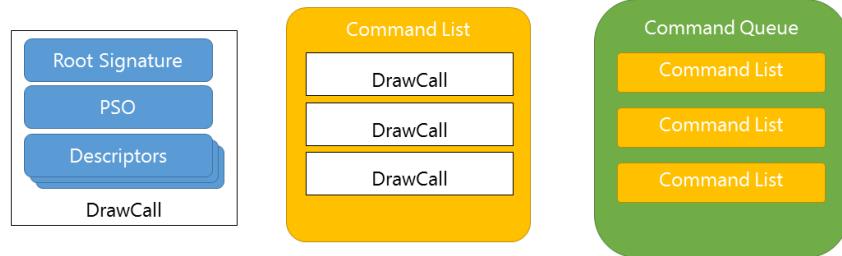


<https://www.isus.jp/games/direct3d-12-overview-part-6-command-lists/>

と色々と書いてありますが、正直読んでも分かりません。正確に言うと説明できるほどの理解がでてないです。この図から判断するに、様々な命令をコマンドリストに溜めておいて、それぞれのコマンドリストを実行のためのキューとして溜めておくのがコマンド キューです。

で、上の図でコマンドリストが複数あるのですが、これは1と2は別スレッドにて実行されるイメージです。ここでマルチコアの概念が絡んでくるのでちょっと難しいので今の所はなんとなく命令を溜めているものとイメージしてください。

ちなみに別のサイトでも



<https://shobomaru.wordpress.com/2015/04/20/d3d12-command/>

のような説明がされており、コマンドリストとコマンドキューは集約関係にあるようです。細かい話は実際に使っていけば分かると思いますので、今の所はそういうイメージで。ただし、↑のサイトでも注意されているように、別のコマンドキューにコマンドリストを放り込んだ場合その実行順序やスレッドセーフは保証されませんよ。という事には注意してください。ということです。

一応僕の持っている本では

- 「コマンドリストは主にCPU側からの働きかけ」
- 「コマンドキューは主にGPU側からの働きかけ」

と書かれています。でもこの説明はちょっと怪しいなあと思います。たぶん正確に理解している人じたしいが少ないと思いますので、皆も今の段階ではあまり深く思いつめないほうが多いと思います。

ともかくコマンドリストとコマンドキューがあるという事を頭に入れておいてください。

で「溜めていく」ためにはそのためのメモリ確保の仕組みを作つておかなければならずそのためには

「コマンドアロケータ」というものが必要になってきます。というわけでこの3つをコントロールするための変数を作ります。

//コマンド周り

```
ID3D12CommandAllocator* _commandAllocator = nullptr; //コマンドアロケータ  
ID3D12CommandQueue* _commandQueue = nullptr; //コマンドキュー  
ID3D12GraphicsCommandList* _commandList = nullptr; //コマンドリスト
```

では作成していきましょう。まずはコマンドアロケータから。作成するには CreateCommandAllocator を使用します。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788655\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788655(v=vs.85).aspx)

はいまた

```
REFIID          riid,  
[out] void      **ppCommandAllocator
```

のパターンです。こういうパターンに早く気付けるかどうかが、使いこなすための差となって表れてきます。繰り返し意識的に練習あるのみです頑張りましょう。

どういうパターンかというと IID_PPV_ARGS を使うあのパターンですよ。つまり

dev->CreateCommandAllocator(後で説明します, IID_PPV_ARGS(&_commandAllocator));
という感じになります。

次に何を考えなきゃいけんかというと第一引数。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770348\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770348(v=vs.85).aspx)

これの中から選ばなければならぬのですが、そろそろ英語もきついかなー。誰か翻訳してないかなー。

<http://hexadrive.jp/hexablog/%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%A0/13072/>

おお!!!神よ!!!

ちなみにこの「ヘキサドライブ」は業界でも有名な技術力を誇る会社です。まあ最近は某スマホゲーの会社に引き抜かれたりしてて、体力減ってるけど頑張ってるみたいで。技術を追求したい人にはお勧めの会社です。

今回はシンプルに行きたいで

D3D12_COMMAND_LIST_TYPE_DIRECT

を使用します。

キチンと戻り値を確認して S_OK が返るのを確認しておいてください。

次にコマンドリストを作りましょう。

CreateCommandList を使用します。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788656\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788656(v=vs.85).aspx)

```

HRESULT CreateCommandList(
    [in] UINT           nodeMask, //0でいいよ
    [in] D3D12_COMMAND_LIST_TYPE type, //D3D12_COMMAND_LIST_TYPE_DIRECTでいいよ
    [in] ID3D12CommandAllocator *pCommandAllocator, //アロケータ(↑で作ったやつ)
    [in, optional] ID3D12PipelineState *pInitialState, //nullptrでオッケー
    REFIID             riid, //例の奴
    [out] void          **ppCommandList //例の奴
);

```

はい、という事なので、CreateCommandList に関しては↑の説明を見ながら自分で設定してみましょう。そろそろ慣れてね。

ちなみに第4引数を nullptr にしても良い理由は

This is optional and can be NULL. If NULL, the runtime sets a dummy initial pipeline state so that drivers don't have to deal with undefined state.

という事です。

俺訳

「これは nullptr でもオッケー。もし nullptr にしたら、未定義のステートを解決しなくてもいいように、ダミーの初期化/パイプラインステートを設定します。」

Google 翻訳

「これはオプションで、NULL でもかまいません。NULL の場合、ランタイムは、ドライバが未定義状態を処理する必要がないように、ダミーの初期/パイプライン状態を設定します。」

うーん。Google 翻訳のほうが分かりやすいですね。皆さんも Google 翻訳は活用していくようにないましょう。

それでは実行して、リザルトが S_OK であることを確認してください。

さて、よいよコマンド周りの最後ですね。コマンドキューです。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788657\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn788657(v=vs.85).aspx)

```

HRESULT CreateCommandQueue(
    [in] const D3D12_COMMAND_QUEUE_DESC *pDesc, //ちょっとこれは説明が必要ですね
    REFIID             riid, //ひとつもの
    [out] void          **ppCommandQueue //ひとつもの
);

```

);

第一引数でいきなり説明が必要なので説明しておきます。定義がこのような状況になっているときは事前に D3D12_COMMAND_QUEUE_DESC 型の変数を作つておいて、そいつのアドレス(&)を投げてあげる必要があります。

```
D3D12_COMMAND_QUEUE_DESC desc={};
```

で、中身をしつかり記述しておかないと意味がないので、こいつの仕様を見てみましょう。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903796\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903796(v=vs.85).aspx)

NodeMask

For single GPU operation, set this to zero. If there are multiple GPU nodes, set a bit to identify the node (the device's physical adapter) to which the command queue applies. Each bit in the mask corresponds to a single node. Only 1 bit must be set. Refer to [Multi-Adapter](#).

うーん。教えて Google 翻訳先生。

「単一の GPU 操作では、これをゼロに設定します。複数の GPU ノードがある場合は、コマンドキューが適用されるノード（デバイスの物理アダプタ）を特定するビットを設定します。マスク内の各ビットは単一のノードに対応します。1ビットのみ設定する必要があります。マルチアダプタを参照してください。」

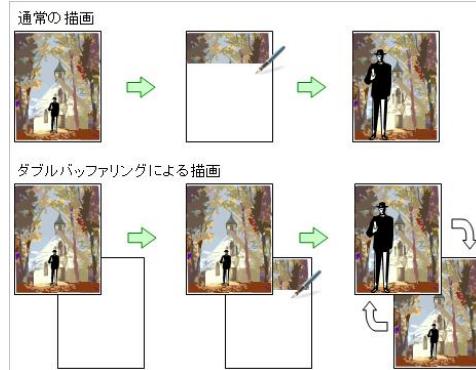
らしいです。今回は普通に GPU を使うので、ここでは 0 にしておきましょう。さて、ここまでヒントから CreateCommandQueue を作つて、S_OK が返るか確認してみてください。

さて、そこまでできればコマンド周りは終了です。

次はスワップチェインです。

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う。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可能である。

ただし…大抵の場合は 2 画面で十分である。今の君たちには意味ないね!!! こいつに関しては DirectX11 の頃と同じです。

で、スワップチェインを作るときには、例によって CreateSwapChain 的な関数を使うんだが、ウィンドウと関連付けるためにウィンドウハンドルとバインドする関数 CreateSwapChainHWnd 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

IDXGIFactory4

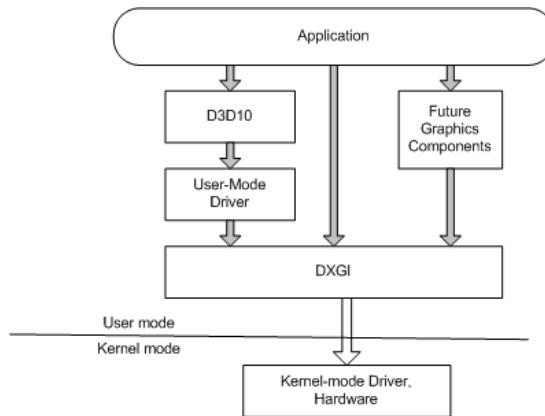
等という聞いたことのないものになっている。これは DirectX11 でもそうだったのだが DXGI という概念に軽く触れておく必要がある。

https://ja.wikipedia.org/wiki/Windows_Display_Driver_Model#DXGI

に書いてあるが

[https://msdn.microsoft.com/ja-jp/library/bb205075\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb205075(v=vs.85).aspx)

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ふ判断なのでしょう。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.jsus.jp/wp-content/uploads/pdf/b25_sample-app-for-direct3d-12-flip-model-.pdf

[swap-chains.pdf](#)

比較的…比較的分かりやすいです。

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

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

DXGI_FORMAT

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

定義を見るところうなってますね？

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

DXGI_SCALING

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

DXGI_SWAP_EFFECT

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

DXGI_ALPHA_MODE

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

DXGI_SWAP_CHAIN_FLAG

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

さて、書き込み幅はともかく他が良く分かりませんね？

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

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 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_R32G32B32_UNORM にしています。

なお、UNORM というのは何かといふと

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

いや、おそらくは 32 ビット使用して小数点を作っているのは分かるんですけどね。

次に USAGE ですが、これは

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

の中から選ぶんですが、今回は

DXGI_USAGE_RENDER_TARGET_OUTPUT

を使用します。

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

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

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

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

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

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

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

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

となっているんですが、この説明を見ても BACK_BUFFER でもいいような気がするんですよね。というわけで、こういう疑問を君たちも持てるようになってください。

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/バックバッファー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？

危ない危ない。DirectX11までのパターンで Deviceを入れるところだったぜ。というわけで既に生成しているコマンドキューを入れましょう。

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。DirectX11やってる人は逆に引つかかる部分なのでご注意ください。

この戻り値が S_OK になるところをご確認ください。

これでスワップチェインは終わりです。次はディスクリプタです。

ディスクリプタとレンダーターゲット

さて、またわけのわからない用語が出てきました。ホンマに未知の用語をポンポン出してくるのやめてくれへんかな…

あと、僕も「デスクリプター」と言ったり「ディスクリプター」と言ったりするかもしれません。綴りが Descriptor ので、カタカナ的にはどっちでもいいかなーって思っています。あまり気にしないでください。

で、詳細な説明を日本語でやってくれているサイトが

<https://www.jsus.jp/games/introduction-to-resource-binding-in-microsoft-directx-12/>

なんだが、案の定良く分からぬ。ポイントを抜き出しておくと
「ディスクリプターは、メモリーにストアされるリソースを表します。ディスクリプターは、GPU 固有の不透過な形式で GPU へのオブジェクトを記述するデータブロックです。ディスクリプターを簡単に考えると、DirectX11における古い“ビュー”システムの代替です。さらに、DirectX11 のシェーダー・リソース・ビュー (SRV) と順不同のアクセスビュー (UAV) など異なるディスクリプター・タイプに加え、DirectX12 は、サンプラーと定数バッファービュー (CBV) のようなディスクリプターもあります。」

さて、理解を深めるために DirectX11 の「ビュー」について説明しておこうか。

[https://msdn.microsoft.com/ja-jp/library/ee422117\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee422117(v=vs.85).aspx)

ビューっていうと View で、なんか 3DCG に関連しそうな名前なんだが、そうではないし、ビューポート(Viewport)のビューとも違う。非常に面倒な用語だ。この辺が紛らわしいために DX12 では名前をディスクリプタにしたのではないかと思えるほどだ。

こいつはつまるところ、ダイナリの塊…データの塊。要はそのデータの「見方」が分からぬ奴にとては何なのが分からぬ塊。

そいつを意味のあるデータにするために必要な仕組みとでも思ってもらえばいいだろう。昨年の DX11 のテキストの説明では

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

てな感じだ。もうちょっと言うと、CPU 側のデータを GPU に転送できるようにするために必要なものだ。単純に言うと「データ」と「その使い方」のセットだ

今回のディスクリプタもとりあえずはそういう仕組みを形作っているものと思っていければいいと思う。

また、同時に考えなければならぬのが、これは DX9 からある概念なのだが
レンダーターゲット

という概念がある。本当に初めての概念、用語ばかりで大変だと思うが DirectX 故致し方なし。
[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dd756755\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dd756755(v=vs.85).aspx)

レンダーターゲットってのは「絵を描くキャンバス」くらいに考えておいてもらえばいいです。DxLib の時でもこれは働いているのですが、意識しなくていいようになってました。

今回はこれを意識しなければならず、そのためにメモリの確保もしなければなりません。面倒ですが仕方ないです。

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
- デスクリプタヒープのサイズ(整数型)を記録
- ディスクリプタヒープ
- ディスクリプタハンドル

となります。メンドクサイですね。ほんと。

手順としては

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

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

デスクリプタヒープを作るには

CreateDescriptorHeap 関数を使います。

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

HRESULT CreateDescriptorHeap(

```
(in) const D3D12_DESCRIPTOR_HEAP_DESC *pDescriptorHeapDesc,
      REFIID                      riid,
(out)    void                  **ppvHeap
);
```

第二、第三引数はいつものパターンですね。

問題は第一引数ですが、

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

を見ながらやっていきましょう。

今回はレンダーターゲットに使用するので、Typeは

D3D12_DESCRIPTOR_HEAP_TYPE_RTV

ですね。ちなみに RTV は“RenderTargetView”的略です。

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

D3D12_DESCRIPTOR_HEAP_FLAG_NONE

次に NumDescriptors ですが、こいつはヘルプを見るだけじゃ分かりませんでした。

The number of descriptors in the heap.

うう…ごめん、これでは何の情報量もないよ。

なのでサンプルを見ながら考えましたが、こいつは既に設定している画面のバッファ数と同じで良いようです。つまり今回であれば**2**を指定しましょう。

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

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

って書いてるからです。

```
ID3D12DescriptorHeap* descriptorHeap = nullptr;
result = dev->CreateDescriptorHeap(&descriptorHeapDesc, IID_PPV_ARGS(&descriptorHeap));
```

これもまた S_OK が返ってくるまで頑張りましょう。

次にデスクリプターヒープサイズを計算します。

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

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

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

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

終わりです。久々に心がほっこりするね。

ちなみに「デスクリプター(Descriptor)」って何かっていうと、意味的には「記述するもの(記述子)」なんですが、それだと実用の意味と離れているので、言い換えると

「GPUに渡すためのビューとかサンプラーを乗つけるための箱」



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

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

11の頃に説明したことそのまま言うと

ビューの解説

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

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

というわけです。

次にサンプラーの解説ですが

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

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

初回はさらっと流して、何度もプログラム組んでたらわかるよ。

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

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

CD3DX12_CPU_DESCRIPTOR_HANDLE

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

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

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

ちょっと面倒なんだけど

CD3DX12_CPU_DESCRIPTOR_HANDLE descriptorHandle(descriptorHeap);

ではうまくいかないのよね。

定義を見ると

const D3D12_CPU_DESCRIPTOR_HANDLE &0

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

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

と

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn788648\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn788648(v=vs.85).aspx)
をもう一回見てみよう。見ての通り英語だ。僕もつらい。

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

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

一応儀式的に

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn899174\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899174(v=vs.85).aspx)
をさらっと読んだら

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

てな感じで、デスクリプタハンドルを作りましょう。

レンダーターゲット

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

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

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

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

とする「キャンバス」だと思ってください。

つーわけで

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

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

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

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

```
DXGI_SWAP_CHAIN_DESC swcDesc = {};
swapChain->GetDesc(&swcDesc);

int renderTargetsNum = swcDesc.BufferCount;
```

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

でループを回しながら、レンダーターゲットビューの作成をやっていきます。

レンダーターゲットビューってのは「絵を描く職人とキャンバス」のことですが、DX11 の頃に比べるとチョットばかりややこしいので一旦正解のコードを書きます。書きますが、皆さん自身のコードでこれを書き換えるという事を忘れないようにしてください。もしこの通りに書いて「センターの言うとおりに書いたのに動かんかった」とか言っても知りません。僕の所では動いてますんで。

```
//レンダーターゲット数ぶん確保
renderTargets.resize(renderTargetsNum);

//デスクリプタ1個あたりのサイズを取得
int descriptorSize = dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
for (int i = 0; i < renderTargetsNum; ++i) {
```

```
result = swapChain->GetBuffer(i, IID_PPV_ARGS(&renderTargets[i])); //スワップチェインから「キャンバス」を取得  
dev->CreateRenderTargetView(renderTargets[i], nullptr, descriptorHandle); //キャンバスと職人を紐づける  
descriptorHandle.Offset(descriptorSize); //職人とキャンバスのペアのぶん次の所までオフセット  
}
```

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

最初に「デスクリプタインクリメントサイズ」ってのを取得していますが
[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn99186\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn99186(v=vs.85).aspx)

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

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

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

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

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

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

ルートシグネチャー

まーたわけわかんない概念が出てきました。ルートシグネチャーです。

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

これ読んでも良く分からなかったので、

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

を見ました。

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

よくわかりませんが…

RootSignature

| Descriptor Tables

| Descriptors

└ Constants

こういった構造になっているらしいです。

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

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

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

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

ちなみに ID3DBlob ってのは汎用的に使用するためのメモリオブジェクトだと思ってください。

Blob ってのは不定形ってな意味があります。興味があったら「不思議なプロビー」とか映画「the BLOB」を見ると分かりやすいかもしれません。

CreateRootSignature

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

を使います。

//ルートシグネチャの生成

```
result = dev->CreateRootSignature(0,  
signature->GetBufferPointer(),  
signature->GetBufferSize(),
```

```
IID_PPV_ARGS(&rootSignature));
```

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

D3D12SerializeRootSignature を使用します。

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

ここで第一引数である D3D12_ROOT_SIGNATURE_DESC は Flagsだけ指定すればよく

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

他は nullptr と 0 でいいので、

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

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

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

D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT;

は

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

に書かれているように、

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

に書かれているように、

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

ということで、今回は「入力アセンブラ」は使用するので
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT
を指定します。

つまりこのように書くことになります。

```
D3D12_ROOT_SIGNATURE_DESC rsd = {};  
rsd.Flags = D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT;  
さて、これでできた RootSignatureDesc を使って、シリアル化していきましょう。  
第一引数はこの rsd のアドレスを代入し、第二引数は  
D3D_ROOT_SIGNATURE_VERSION_1 を指定しておけばいい。
```

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

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

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

ちょっとですね？ここまでやれば一応 DirectX から画面に影響を与えることができるんで、さっさとそこをやっていきたいかなって思うわけです。これ以上画面に変化が現れない！初期化だと戦争が起きますので…



というわけで暴動が起きないようにひとまずは画面に影響を与えましょう。

画面に影響を与えよう(画面を特定の色でクリア)

まあ簡単に言うとですね、DirectX から働きかけて画面の色を変更して、確かにウインドウを DirectX がジャックしているというのを実感してみましょうってことです。

そして、その程度の事であればすぐにでも可能な状態になっているという事です。

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

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

レンダーターゲットクリアコマンド発行

という事で ClearRenderTargetView 関数を使用します。

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

DX11 やった人にはお馴染みなのではないでしょうか。

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

で、これをメインループの先頭あたりで呼び出します。

こいつはコマンドリストの持ち物なので…あとはわかるな?

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

さて…

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

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

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

ぱっと見では分かりませんが、

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

ひとまずはクローズをクリア命令の直後で呼んでください。

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

ExecuteCommandLists

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

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

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

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

```
_commandQueue->ExecuteCommandLists(1, (ID3D12CommandList* const*)&_commandList);
```

こんな感じです。今回はコマンドリストが1つだけなので、第一引数は1でいいし、第二引数もキャストでなんとかしています。

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

ただし、この場合キャストが面倒だし分かりにくいってのもあるので一般的には

```
ID3D12CommandList* commandList = { _commandList };
_commandQueue->ExecuteCommandLists(_countof(commandList), commandList);
```

みたいな書き方をすることが多いです。

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

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

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

ループの先頭で

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

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

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



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

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

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

それはともかく OMSetRenderTarget ですが

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

DirectX11 の時から比べるとちょっと変わってます。

DX11 バージョン

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

DX12 バージョン

```
void OMSetRenderTargets(
    [in]          UINT           NumRenderTargetDescriptors,
    [in, optional] const D3D12_CPU_DESCRIPTOR_HANDLE *pRenderTargetDescriptors,
    [in]          BOOL           RTsSingleHandleToDescriptorRange,
    [in, optional] const D3D12_CPU_DESCRIPTOR_HANDLE *pDepthStencilDescriptor
);
```

どうですか？違ひが分かりますか？

頭の悪い答え：“ふくざつになっている”



どこがどう複雑になっているのか答えないと言えません。見たまんまでいいのでもうちょっと細かく考えましょう。

引数が増えている

View→CPU_DESCRIPTOR_HANDLE になっている

ごめん、それくらいだった。俺も頭が悪い。

で、

_commandList->OMSetRenderTargets(1, &**descriptorHandle**, false, nullptr);

このようにしてはいけない。

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

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

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

ちなみに Present 関数はこう書いてください。

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

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

つまり「裏画面」を示すインデックスが変更されるのだ。

というわけである関数を呼び出す。

```
GetCurrentBackBufferIndex
```

である。

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

まあ、なんてことはない。

裏画面のインデックスを得るだけだ。

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

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

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

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

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

クリアの部分は真っ赤にするくらいでいいんじゃないでしょうか。

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

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

ここが R8G8B8A8_UNORM と関わってるんです。

余裕のある人は色に変化が出るようにしてみてください。

フェンス

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

そもそも非同期処理とは？

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

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

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

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

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

前にも言ったかもしれません、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「Now Loading」出してましたが、最近のゲームではその時間をごまかすための工夫がより洗練されているようです。

で、ここで非同期ロードには欠かせない概念として「いつロード完了したか」を判断しなければならないわけです。ロードが完了してもいいなし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.0.007.jp/function/dxfunc_other.html#R21N2

などでチェックすることができます。

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

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

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

あと、非同期処理が顕著なのは「ネットワーク通信」があるゲームですね。結構ネットのデータのやり取りって遅いんです。当然パケットが大きくそして多ければ多いほど時間がかかりますので、まずは送るパケットを工夫して小さくするところから始まりますが、ともかくここでも完了復帰は普通に使用されます。最後にDBへのアクセスもそうですね。

ですから、みんな大好きスマホネトゲにおいては多用されているシステムなので、この辺の理解は JK(常識的に考えられる)の範囲内なわけです。



さて、ここでネットワーク通信ゲーム(MMORPGなど)について考えてみましょう。

たとえば MMORPG などではネットワーク上のプレイヤーたちの座標情報が通信されていて、その情報を元に画面上にほかのプレイヤーを表示させているわけです。

で、先ほども言ったようにネットワーク通信ってのは時間がかかるため、他プレイヤーの座標を取得する命令を出して、返ってくる間に「表示」しなければならないことがあるのですが、データがない以上は不適切な場所に表示することになり、ゲームが崩壊することもあるわけです。

で、結局 DirectX12 ではどうなの?

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

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



まずはそれを防止しなければなりません。面倒ですけどね。

DirectXにはフェンスという仕組み(ID3D12Fence)があり、それを使用することでGPUに投げた処理を「待つ」ことができます。

ここで

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



これには理由があるのだ。

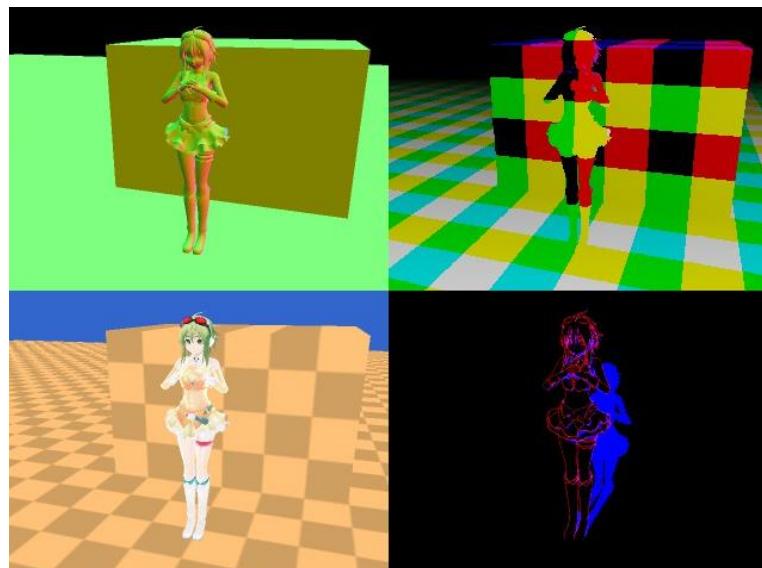
そもそもなんでこんなややこしいことをする羽目になったのかというと

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

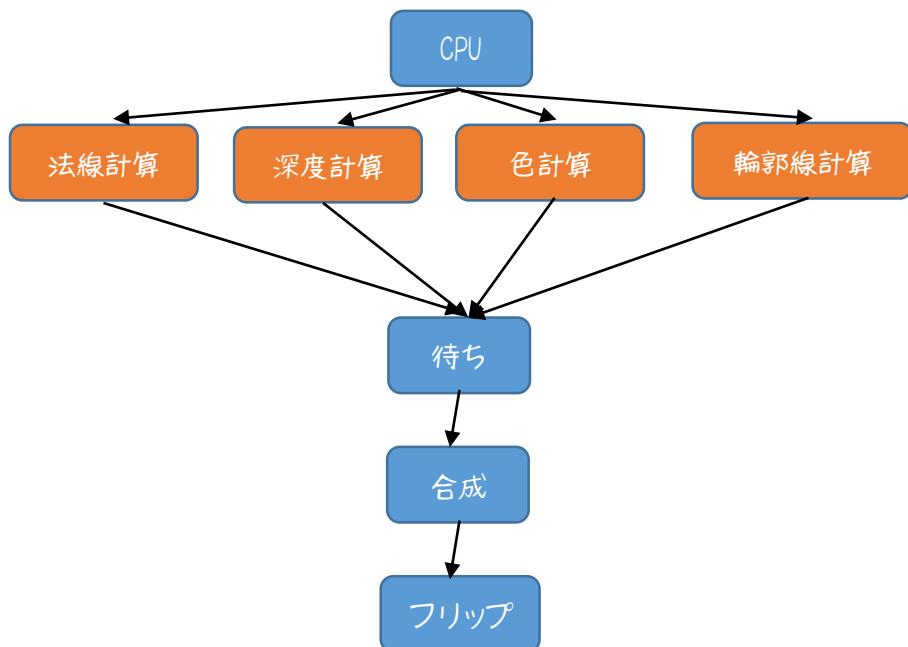
意味が分からぬんだろうから簡単に言うと。

一枚の画面を作るために

事前に↑の絵のような複数の情報を作つておいて、最後に合成するわけです。



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



すつぜー大雑把に言うとこういう感じにしているわけ。徒競走で4人の走者がいて、運営側は全員がゴールするまで待つておかなければならぬみたいだ。そういう状態です。

ちなみにこの仕組みに対応できているハードはまだ多くはなくPS4やXBoxOneなどは対応していると思いますがGeForceGTX860以前のPCでは対応していないと思います。

フェンスの仕組み

フェンスの仕組み自体はクソ単純です。すぐに理解できると思います。



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

ちなみにそれでも分かりづらいかも知れないのが

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

Signal(指定の値)

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

な?クソ簡単じゃろ?

ではフェンスを実装しようか

やることはそれほど大変ではないです。まずはフェンスオブジェクトを作ります。

```
ID3D12Fence* _fence=nullptr;
```

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

にUINT64型で定義しよう

```
UINT64_fenceValue=0;
```

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

次にフェンスオブジェクトを生成します。CreateFence を使います。

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

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

で、例えばこう

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

まあ、やろうとしてることは分かるでしょ？

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

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

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

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

例えばこうですね。

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

で、ここで注意してほしいことは Signal 関数は「待ってはくれない」という事です。
「待つ処理」は自分で作らなければなりません。

一番手っ取り早く分かりやすい方法は「ポーリング」

Signal の後に

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

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

…まあ簡単でしょ？

とりあえずこれをやれば不具合はなくなると思うけど…どうかな？面倒だねえ。

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

そりやそりや。実行中にリストがリセットされりやそりやおかしくなるわな。

ポリゴンを表示しよう

さて、やっとこさポリゴンを表示できるところまで来ましたね？これまたポリゴンを表示するための準備がちょっと面倒なんですが頑張ってやっていきましょう。

ポリゴンを表示させるための

- 頂点情報(頂点、座標のみ)
 - 頂点レイアウト
 - 頂点バッファ
 - 頂点バッファビュー
 - 頂点シェーダ
 - ピクセルシェーダ
 - パイプラインステートオブジェクト
- が少なくとも必要です。

また多いですね。でも頂点情報そのものは DxLib を使用しても同じですし、頂点シェーダ、ピクセルシェーダに関してはこれまた DxLib を使ってても結局同じです。

まずは頂点情報を作っていきましょう。

頂点情報を作る

今回は三角形を作っていきます。

頂点はいくつひつようかな？そう、3 頂点ですね？

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

まず構造体を作ってみましょう。

一応一番最初に便利ライブラリとして

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

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

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

DirectX::XMFLOAT3

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

using namespace DirectX;

って cpp の先頭(インクルードの後くらい)で書いてください。

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



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

```
struct Vertex{  
    XMFLOAT3 pos;//座標  
};
```

こんなのは作ってください。一応意味は分かりますよね？ そう

Vertex vertex;

vertex.pos.x=...

みたいにして頂点を定義して使うわけです。

とりあえず3点定義します。

//頂点情報の作成

```
Vertex vertices() = { {{0.0f,0.0f,0.0f}},  
                      {{ 1.0f,0.0f,0.0f }},  
                      {{ 0.0f,-1.0f,0.0f }} }
```

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

頂点レイアウト

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

で定義します。これもDX11のやつを参考に見てみます。

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

まず、分らない用語が出てきます。

「セマンティクス」ってなんや？

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

「HLSL セマンティクス」で検索すると

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

POSITION(座標)だの COLOR(色)だのが出てきます。

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

POSITIONなら xyzw, COLORなら rgbaですね。

まあ最初は POSITION のみでいいです。

SemanticIndex はしばらく 0 でいいです。同一セマンティクス要素は出でこないので。

次の DXGI_FORMATですが、これは FLOAT いくつ分のデータとかそういうのを記述します。

FLOAT 三つ分なので

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

を見ながら

DXGI_FORMAT_R32G32B32_FLOAT

を指定します。

この辺が面倒なのですが X32Y32Z32 なん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のです 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。

頂点バッファ

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

まず頂点バッファを作ります。でも DirectX11 得意ニキを罠にはめる情報も満載なのです。そもそも ID3D12Buffer が存在しない。代わりに

ID3D12Resource を使用します。

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

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

ID3D12Resource* _vertexBuffer=nullptr;

とでも宣言しておいてください。

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

代わりにあるのが CreateCommittedResource である。

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

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

```
dev->CreateCommittedResource(  
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD), //CPUからGPUへ転送する用  
    D3D12_HEAP_FLAG_NONE, //特別な指定なし  
    &CD3DX12_RESOURCE_DESC::Buffer(sizeof(vertices)), //サイズ
```

```
D3D12_RESOURCE_STATE_GENERIC_READ,//よくわからない  
nullptr,//nullptrでいい  
IID_PPV_ARGS(&_vertexBuffer));//いつもの
```

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

ちなみに D3D12_RESOURCE_STATE_GENERIC_READ の部分に「良く分からぬ」と書いたら、私が、これ、日本語に訳しても

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

とか非常に不穏なことを書いていますし。正直な話ここでサンプル頼みになっちゃうのは非常に悲しいし、申し訳ないけど俺の力不足です。

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

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

これねえ…DX11 の時代は初期情報を Create の時点で突っ込むこともできたんですが、DirectX12 は Map すること前提なので…ホンマにハードル上がつるわ。

で、Map って何って言うとすぐ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ね、「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。ということで

`D3D12_RANGE range = { 0, 0 };`

適当な値を入れておいて、第二引数にセットします(もしかしたらこいつは nullptr 入れてお

けばいいかも知れません)

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

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

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

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

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

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

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

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

頂点バッファビュー

頂点バッファビューを宣言します。

D3D12_VERTEX_BUFFER_VIEW _vbView={};

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

まずは頂点バッファの GPU におけるアドレスを記録しておきます。

_vbView.BufferLocation=_vertexBuffer->GetGPUVirtualAddress();

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

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

_vbView.StrideInBytes = sizeof(Vertex);

次にデータ全体のサイズを伝えます。

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

で、このビューを最終的にはコマンドリストにて

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

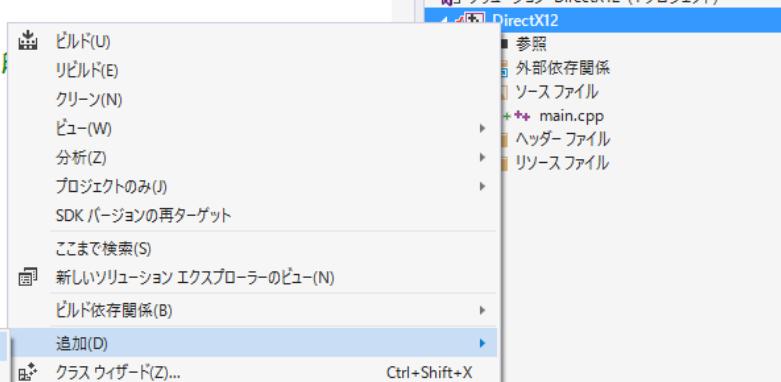
てな投げ方をするんですが、それはもうちょっと後でやります。

そんな事よりシェーダ書こうぜ

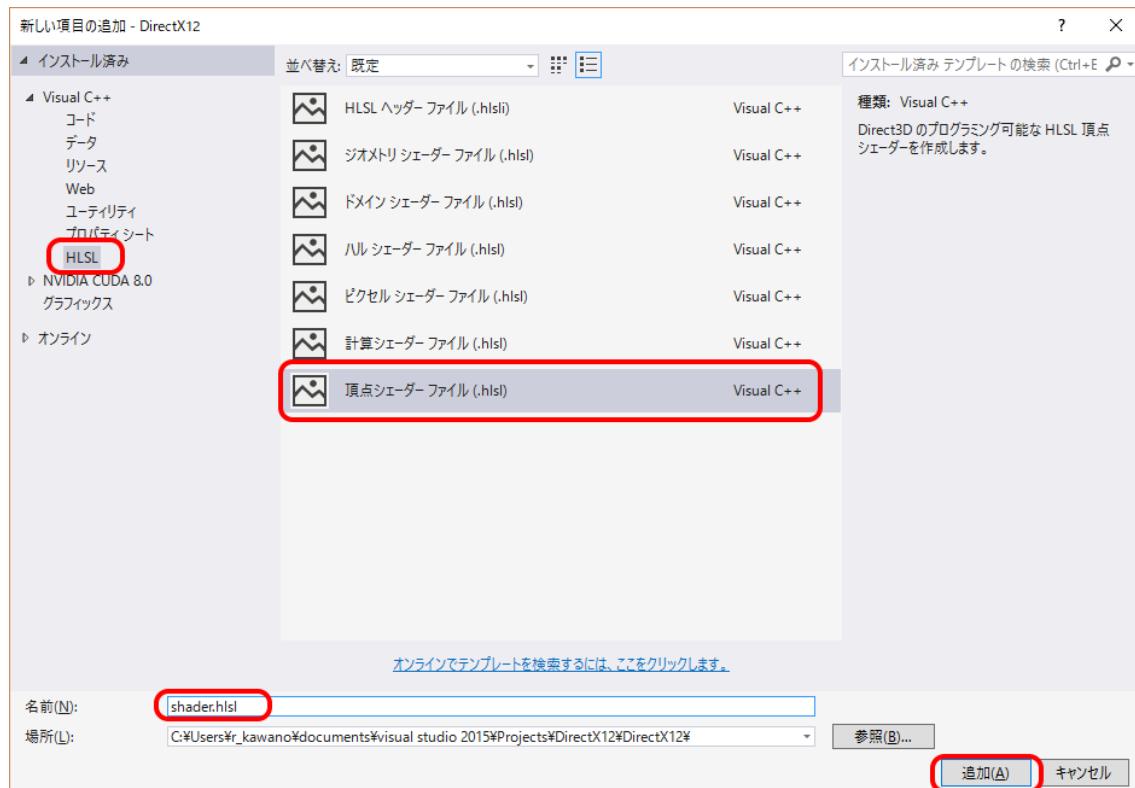
AD), //CPUからGPUへ転送する
3)), //サイズ
,ない

```
buff);
```

新しい項目(W)... Ctrl+Shift+A

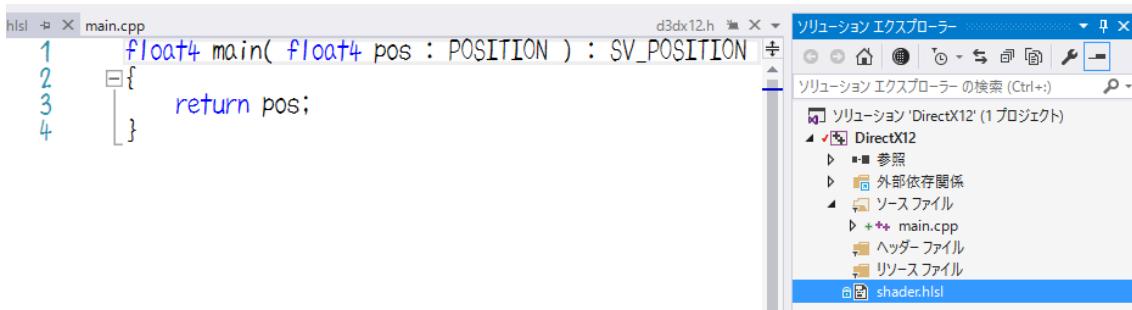


プロジェクトで右クリック→追加→新しい項目を選ぶと

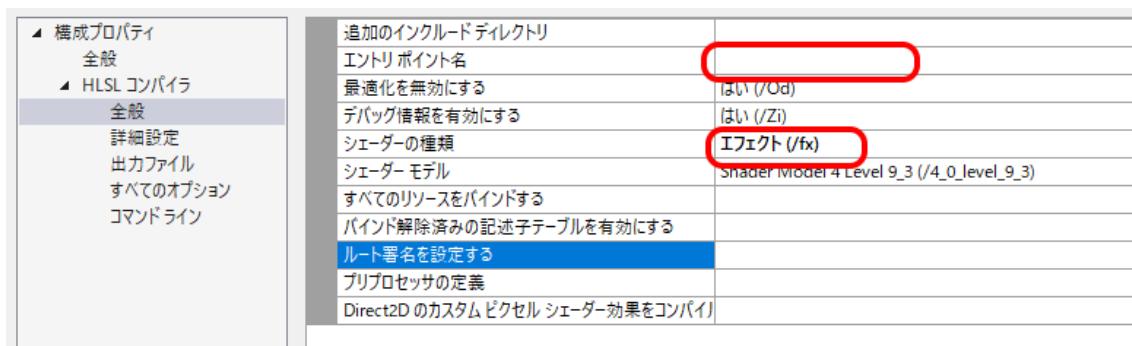


こんなのが出てくるので頂点シェーダファイルとして追加してください。とはいって実際には

頂点シェーダとピクセルシェーダを共用しますので、VertexShaderではなく、`shader.hlsl` にし
ていてください→追加。



こうなるので、`shader.hlsl` で右クリック>プロパティ>
HLSL コンパイラ→全般



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

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

あと、シェーダモデルは 5.0 にしてください。

で、コンパイルして通ればひとまずシェーダは大丈夫です。

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

シェーダ読み込み

はい、久々のプロフですが、シェーダ用の宣言です。

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

```
ID3DBlob* pixelShader = nullptr;  
次にこれに対してシェーダのコンパイルを行います。  
result = D3DCompileFromFile(_T("shader.hlsl"), nullptr, nullptr, "BasicVS", "vs_5_0", D3DCOMPILE_DEBUG |  
D3DCOMPILE_SKIP_OPTIMIZATION, 0, &vertexShader, nullptr);  
result = D3DCompileFromFile(_T("shader.hlsl"), nullptr, nullptr, "BasicPS", "ps_5_0", D3DCOMPILE_DEBUG |  
D3DCOMPILE_SKIP_OPTIMIZATION, 0, &pixelShader, nullptr);  
ちょっと長いんですけど、頑張って書いてください。  
で、これ実行しようとするとリンクに怒られるので d3dcompiler.lib をリンクしてください。
```

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

ここで漸くパイプラインステートオブジェクトの出番です。

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。

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習性がある。

これもそのひとつで、前のデスクリプタガービューなどをまとめたのに対して、ステート系をまとめてしまします。

ステート系ってのはここまで話で具体的に言うと…

- ルートシグネチャ
- 頂点レイアウト
- 頂点シェーダ
- ピクセルシェーダ

です。それに加えて

- 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

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、正直クソタレなので、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;//今はファルスで
DepthStencilState.StencilEnable=false;//今はファルスで
D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE
レンダーターゲット数1
フォーマットは R8G8B8A8_UNORM
サンプルカウントは1で
```

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

CreateGraphicsPipelineState

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

で、以下のように書いてください。

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC gpsDesc={};
gpsDesc.BlendState=CD3DX12_BLEND_DESC(D3D12_DEFAULT);
gpsDesc.DepthStencilState.DepthEnable=false;
gpsDesc.DepthStencilState.StencilEnable=false;
gpsDesc.VS = CD3DX12_SHADER_BYTECODE(vs);
gpsDesc.PS = CD3DX12_SHADER_BYTECODE(ps);
```

```

gpsDesc.InputLayout.NumElements=sizeof(inputLayoutDescs)/sizeof(D3D12_INPUT_ELEMENT_DESC)
;
gpsDesc.InputLayout.pInputElementDescs = inputLayoutDescs;
gpsDesc.pRootSignature = rootSignature;
gpsDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
gpsDesc.RTVFormats(0)=DXGI_FORMAT_R8G8B8A8_UNORM;
gpsDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
gpsDesc.SampleDesc.Count=1;
gpsDesc.NumRenderTargets=1;
gpsDesc.SampleMask=0xffffffff;
ID3D12PipelineState* _pipelineState=nullptr;
result = dev->CreateGraphicsPipelineState(&gpsDesc,IID_PPV_ARGS(&_pipelineState));
で、これでS_OKが返ってこない場合はシェーダとかレイアウトとかが間違えていると思いま
すので、確認しておいてください。

```

その他やらなければならぬ事

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

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

くらいなのだが、これに加えて、以前画面クリアするときには使ってない描画という処理を使
ってるので、またちょっとだけ面倒なことをやらなければならぬ。
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「バックバッファにバリアをかけておくため

ResourceBarrier

<https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903898>
_commandList->ResourceBarrier(1,&CD3DX12_RESOURCE_BARRIER::Transition(_renderTargets[1].ffd / フッファフレーム番号),D3D12_RESOURCE_STATE_RENDER_TARGET,D3D12_RESOURCE_STATE_PRESENT));

を使います。これをコマンドリストの一番最後に呼び出します。

あとはそれぞれの処理をこなしていく。

ビューポート

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

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

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

RSSetViewportsって関数を使います。

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

これは DX11 と同じなのでそっち見て考えましょう。

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

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

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

D3D12_VIEWPORT の指定は

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

見てもらえば入れるものは分かると思います。左上は 0,0 でいいです。

で、MinDepth=0,MaxDepth=1 にしておいてください。

残り色々セット

既に作っているパイプラインステートオブジェクトをセット

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

ルートシグネチャーをセット

既に作っているルートシグネチャーを

_commandList->SetGraphicsRootSignature

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

でセット。

ビューポートをセット

RSSetViewports

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

でセット

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

RSSetScissorRects

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

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

ここまでではいいんだが、最後にもう一つ、頂点バッファのセットも必要である。

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

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

これも DX11 と同じなのでこれを見ながら

スロットは 0 でいい。Numbuffers は 1

で、次の引数に頂点バッファビューをセット。

そこまで終わったら

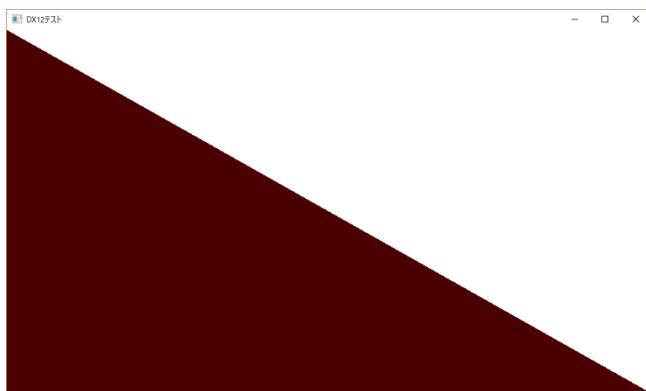
[https://msdn.microsoft.com/ja-jp/library/ee419594\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419594(v=vs.85).aspx)

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

で描画します。

うまくいけば…

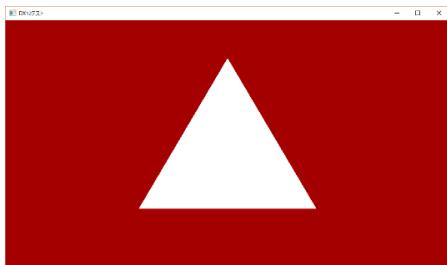
こんな感じの表示になります。右上の白い部分が今回描画している三角形です。



今は左上が $(-1, 1)$ で右下が $(1, -1)$ という状態で三角形を定義しているからこうですが、例えば頂点をちょっといじると

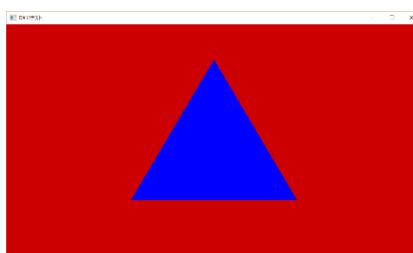
```
Vertex vertices[] = { { { 0.0f, 0.7f, 0.0f } },
{ { 0.4f, -0.5f, 0.0f } },
{ { -0.4f, -0.5f, 0.0f } } };
```

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。



何でこうなるかを考えてほしいのですが、ちょっと面倒なのは SV_POSITION のまま使おうとするとグラデがつからないんですね。

今回いじったシェーダはレンダリングパイプラインと呼ばれる仕組みの3と8に相当します。なお、頂点データはベクトデータでありピクセルシェーダにおいてはピクセルのデータつまりラスタライズ済みのデータです。



このラスタライズの時に色々なことが起こっている(POSITIONとかUVとかNORMALは線形補間がかかります)ので、今後はそこも頭に入れておく必要が出てきます。

ちなみに今の状態を2DのDXLibみたいな指定にしたければ

```
pos.xy = float2(-1,1) + pos.xy / float2(480, -270);
```

こう書きます。なお、今は画面幅960、画面高さ540設定なのでこう書いてます。抽象的に書くならば

```
pos.xy=float2(-1,1)+pos.xy/float2(画面幅の半分, -画面高さの半分)
```

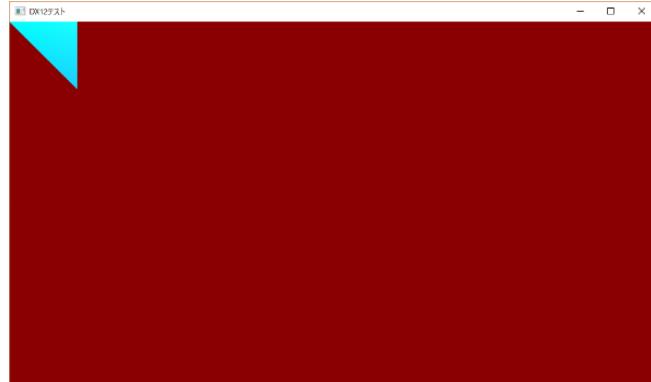
とします。これが何をやっているかは後で話しますが、これの何がうれしいかと言うとピクセル数値通りに画面上に表示されるようになります。そうするとHUD的なUIを作るときに正確に置けるのでこれを覚えとくと重宝すると思います。

この状態で

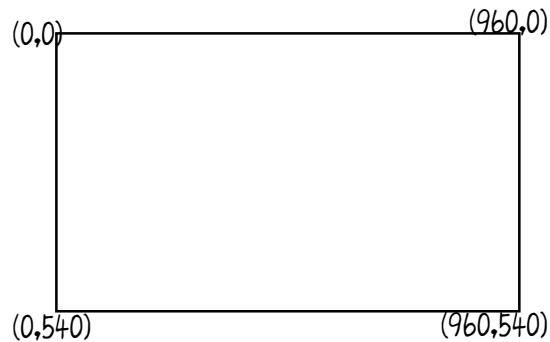
```
Vertex vertices() = { { { 0.0f,0.0f,0.0f } },  
                      { { 100.0f,0.0f,0.0f } },  
                      { { 100.0f,100.0f,0.0f } } };
```

という指定をすると

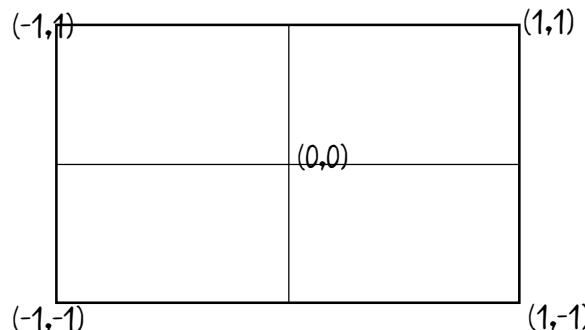
このように思った通りの場所に表示されるようになります。



で、この仕組みですが発想はいたって簡単です。



↑のような座標系を



このような座標系に変換したいわけです

真っ先に思いつくのが画面幅ピクセル数で割ることですが、そんなことをすれば $0 \sim 1$ の範囲となり、マイナス方向がなくなってしまいます。よく考えてください。 $-1 \sim 1$ までの範囲というのは幅的にはどれくらいですか？2ですよね。

つまり $0 \sim 2$ の範囲になるように割り算を行い、そこから -1 をすれば $-1 \sim 1$ の範囲に収まります。つまり、まずそれぞれ画面幅の半分、高さの半分で割ってあげれば $0 \sim 2$ の範囲内に収まります。次にそれから -1 をそれぞれ引けばいい。

ところがY方向にちょっと問題が…。

それは DxLib みたいにしようとすれば Y は下方向になるため、正負を逆転する必要があります。

結果として

```
pos.xy = float2(-1,1) + pos.xy / float2(480, -270);
```

こういう事になります。

ここで注目してほしいのはシェーダ言語における演算は XY とか XYZ をいつまでも行えることです。

ということで単純ポリゴン表示のお話はこれで終了です。

テクスチャマッピング

ちょっと予定と違ってテクスチャマッピングを先にしようと思います。何故かというと先走つてモデル表示した人たちが躊躇するのがここじゃないかなーって思うからです。

ではテクスチャを貼り付けます。貼り付けるのですが、三角形だとちょっと貼り付けづらいので頂点をいじって



このように長方形表示になるようにしてください。これに関しては TRIANGLESTRIP にしたほうが楽です。

```
_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
```

さて、これにテクスチャを貼り付けるとします。テクスチャってのは「絵」です。



こういうやつ

これが上のポリゴンに張り付ければOKって言う事です。これをやるまでに必要な知識と行動は

- UV座標(すぐ終わる)
- テクスチャリソースの作成(まだ平和)
- 画像ファイルロード(くっそメンドクサイ。完全敗北したDX11テクスチャロード)
- 画像ファイルの内容をテクスチャリソースへ流し込み(コピー)
- シェーダリソースビュー作成(比較的平和)
- サンプラ作成(平和)
- シェーダをテクスチャ対応(大丈夫)

申し訳ないけどポリゴン表示の時くらいうんざりするのでごめんけど、ごめんけど頑張ってください。

UV座標を付加

そもそもUVって何者か知っていますか？絵をポリゴンに貼り付けるときに、絵のどの部分と頂点を対応させて貼り付けるかというのを指定するものです。



で、絵に対して上記のような座標が割り当たっている。これがUV座標系です。UV座標は0~1の範囲内です。これを越えると貼りつかないか、繰り返されるかどちらかになります(その辺の設定はテクスチャアドレッシングモードにてやりますが、今は特に考えなくていいです)

頂点情報構造体にUVを追加

UVはFLOAT2つぶんなのだ。簡単なのだ。それを頂点構造体に追加するのだ。XMFLOAT2で大丈夫なのだ。

//頂点情報

```

struct Vertex{
    XMFLOAT3 pos;//頂点座標
    XMFLOAT2 uv;//UV
};

```

頂点情報に UV 座標を追加

例えばこうなのだ。

```

Vertex vertices() = { { { 50.0f, 250.0f, 0.0f },{0,0} },
    { { 50.0f, 50.0f, 0.0f },{0,1} },
    { { 250.0f,250.0f,0.0f },{1,0} },
    { { 250.0f,50.0f,0.0f },{1,1} } };

```

言いたいことはわかったかい？

頂点レイアウトを追加

しかしこのままでは GPU 側が UV 情報が増えたことが分からない。頂点レイアウトを追加すべき。今は POSITION しか入っていない頂点レイアウトに次の一行を追加してください。

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

シェーダ側の引数を追加

さて、ここでシェーダ側のコードだがまずは引数に

`float2 uv:TEXCOORD`

を追加してください。

一応検証のためにピクセルシェーダにこの UV を投げれるようにしてください。ピクセルシェーダに投げれるようにするためには

```

struct Out {
    float4 svpos : SV_POSITION;//システム座標
    float4 pos : POSITION;//座標(補間あり)
    float2 uv : TEXCOORD;//UV座標(補間あり)
};

```

このような構造体を作って頂点シェーダの戻り値はこの構造体型を返すようにします。

```

Out BasicVS( float4 pos : POSITION ,float2 uv:TEXCOORD){
    Out o;
    中略
    return o;
}

```

```

}

で、ピクセルシェーダ側も
float4 BasicPS( Out o):SV_Target
にしておいて、
return float4(o.uv,1,1);
とすれば、全体的にうまくいっていれば

```



こんな感じで染め物みたいになるでしょう。頑張ってください。
ここまで、第一段階…UV 座標への対応は終わりだ。さっさと次行くぞっ!!!時間ないのだ。

テクスチャリソースの作成

テクスチャリソースも頂点リソースと同様に CreateCommittedResource を使用して生成します。

[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)

ここで必要なのは

D3D12_HEAP_PROPERTIES(HEAP_TYPE_DEFAULT)
D3D12_RESOURCE_DESC の作成…頂点 / バッファの時は
CD3DX12_RESOURCE_DESC::Buffer(sizeof(vertices)),
で良かったのだが、テクスチャの場合はこれではダメで、きちんとテクスチャとして指定してあげなきゃいけないです。面倒だけど頑張ろう。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903813\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903813(v=vs.85).aspx)

に DepthOrArraySize は 2D テクスチャの時は 1 にしろって書いてあるから

ひとまずこんな感じで

```

D3D12_RESOURCE_DESC texResourceDesc = {};
texResourceDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
texResourceDesc.Width = 256; // とりあえず決め打ちな
texResourceDesc.Height = 256; // とりあえず決め打ちな
texResourceDesc.DepthOrArraySize = 1;

```

```

texResourceDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
texResourceDesc.SampleDesc.Count = 1;
texResourceDesc.Flags = D3D12_RESOURCE_FLAG_NONE;
texResourceDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;

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

ID3D12Resource* _textureBuffer = nullptr;
result = dev->CreateCommittedResource(&hprop,
                                         D3D12_HEAP_FLAG_NONE,
                                         &texResourceDesc,
                                         D3D12_RESOURCE_STATE_GENERIC_READ,
                                         nullptr,
                                         IID_PPV_ARGS(&_textureBuffer));

```

で、S_OK が返ってくるのを確認してください。そしてここから先は割とさらっとした感じに見えますが、正直なところ悩んで、デバッグして、仕様書見て理解を更新して、を繰り返した結果です。

別にありがたがれというわけではなく、「理解」が中途半端だと余計な回り道をしてしまう。でもその回り道が理解にとっては逆道だったりするので、その辺のニュアンスは分かっていただきたれ！と思っています。だからみんなに見せてない！研究用のコードはクソ汚いです。

ちなみに MSDN 以外に参考にしたサイトが

<https://qiita.com/em7dfggbcadd9/items/b5a9b71abae29d8bda50>
<https://glhub.blogspot.jp/2016/04/directx12rootsignature.html>
<http://zerogram.info/?p=1746#more-1746>
<https://sites.google.com/site/monshonosuana/directxno-hanashi-1/directx-146>

などですが、DirectX12 になって以降もバージョンによってちょいちょい違いがあり、それぞれ言ってることが違ってたりするんですよね…その辺の判断もある程度理解してこない！と見抜けない！のです。

まあ最終的には表示できるんですけどね。本当にめんどう。でもね、大人げないんですけど、サンプル丸写しとかしたくなれいんですよホント。クソ難しいけど可能な限り理解したいんですね。そしてその理解したところを伝えていきたいんですね。

大人げないつすよホントに。「ここをこうしたほうがみんな分かりやすいかな」とか思っていじった瞬間に壊れますからね DirectX12 は。

でも、最後はこれが出るので、感動します。それは保証するんやで。



がんばるぞ!!

さて、今から作成したテクスチャリソースに対してデータを突っ込んでいきます。色々とテクスチャデータに関しては…本当に色々あるんですが、今回は比較的対処しやすい 32bitBMP 画像を使用していきます。

サーバーに

[¥¥132sv¥gakuseigamero¥rkawano¥DirectX12¥texturesample.bmp](#)

があります。プロパティを見て

| イメージ | |
|--------|-----------|
| 大きさ | 256 × 256 |
| 幅 | 256 ピクセル |
| 高さ | 256 ピクセル |
| ビットの深さ | 32 |

ビットの深さが 32 になっていることをご確認ください。

もしくはバイナリエディタで開いて

| | |
|---------------------------|----------|
| Header.bfType | 4D42 |
| Header.bfSize | 0004008A |
| Header.bfReserved1 | 0000 |
| Header.bfReserved2 | 0000 |
| Header.bfOffBits | 0000008A |
| Info.bmiHeader.biSize | 0000007C |
| Info.bmiHeader.biWidth | 00000100 |
| Info.bmiHeader.biHeight | 00000100 |
| Info.bmiHeader.biPlanes | 0001 |
| Info.bmiHeader.biBitCount | 0020 |

BitCount が 32→0x20 であることを確認してください。

これ、通常の BMP の場合 8~24 なんですが、チョイと加工して 32bit にしています。理由はテクスチャのフォーマット指定に

DXGI_FORMAT_R8G8B8_UNORM

的なのがなくて、DXGI_FORMAT_R8G8B8A8_UNORM とかしかないのであります。

もちろんプログラム内で対処のしようはあるのですが、今から色々と詰め込んでしまうのもよくないので、ひとまずはこれで行きます。

ビットマップ読み込み

さて、当たり前のようにビットマップを読み込んでいきます。なぜこれに1つの項目を使っていいかって? LoadFile(ファイル名)みてえにはいけねえからだよ!!!

ひとまずビットマップの構造についてですが…

http://www.umekkii.jp/data/computer/file_format/bitmap.cgi

見てもらうか、バイナリエディタで bmp を読んでもらえば分かりますが、ヘッタ側があってデータがあるという、ごくごく一般的なデータ構造になっているため初心者(を脱しようと思っている者)にはちょうどいいのです。

構造としてはヘッタ側が BITMAPFILEHEADER と BITMAPINFOHEADER でできています。ここに必要な情報が入っています。

で、fread なりなんなり使って、ファイルを読み込んでいくわけですが、バイナリは前期で読み込んだことがあるよね? うーん。とりあえず

```
BITMAPFILEHEADER bmpfileheader = {};
BITMAPINFOHEADER bmpheader = {};
```

をヒントとして書いておくから、ヘッダデータを読み込んでみてください。ちょっとずつ梯子を抜いていきますよ？

ここからビットマップデータ分の読み込みを行っていくわけですが、中に biSizeImage ってのがあるので、それがデータそのもののサイズだと思ってください。

そのサイズで

```
std::vector<char> data;  
data.resize(bmpheader.biSizeImage);
```

とでもやれば、領域が確保できますので、ここからはノーヒントでビットマップデータ全て読み込んでください。

そのあとで先ほど作ったテクスチャバッファに対して書き込みを行います。

テクスチャバッファへの書き込み

関数自体は簡単です。

WriteToSubresource 関数を使います。

<https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn914416>

迷うのは第二引数の D3D12_BOX ですがこれは簡単です。

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。 front と back は 0,1 にしてください。

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))

これで書き込みが終わったと思ったらそうじゃない。そうじゃないのだ。

リソースへの書き込みもまた即時復帰なのでリソースバリアおよびフェンス待ちが必要なのだ。

```
_commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(_textureBuffer,  
D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));  
_commandList->Close();  
_commandQueue->ExecuteCommandLists(1, (ID3D12CommandList* const*) &_commandList);  
  
// そして待ち  
_commandQueue->Signal(_fence, ++_fenceValue);
```

```
while (_fence->GetCompletedValue() != _fenceValue);
```

ちなみに今回 WriteSubresource 関数を使用しましたが、
で、ここからシェーダリソースビューを作ります

シェーダリソースビューの作成

```
D3D12_DESCRIPTOR_HEAP_DESC texDescriptorHeapDesc = {};
texDescriptorHeapDesc.NumDescriptors = 1;
texDescriptorHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
texDescriptorHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;

ID3D12DescriptorHeap* descriptorHeapSRV = nullptr;
result = dev->CreateDescriptorHeap(&texDescriptorHeapDesc, IID_PPV_ARGS(&descriptorHeapSRV));

unsigned int stride = dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
D3D12_CPU_DESCRIPTOR_HANDLE srvHandle = {};
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;

srvHandle = descriptorHeapSRV->GetCPUDescriptorHandleForHeapStart();
dev->CreateShaderResourceView(_textureBuffer, &srvDesc, srvHandle);
```

ひとまずコードを書きましたが、ここから色々と説明していく。

D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV

の意味ですが、CBV=ConstantBufferView(頂点でもテクスチャもないコンスタントバッファの
ビュー)で、SRV=ShaderResourceView(テクスチャの事ね)で、UAV=UnorderedAccessView らしい。
コンスタントバッファとシェーダリソースは DX11 の頃からあるので、まだ言いたいことは分
かるんですが、なんだそのアンノーダードリソースビューってのは…と思ったら DirectX11 か
らあったみたいね。

<http://wlog.flatlib.jp/item/1411>

うーん。よくわからぬけれどたぶん、コンピュートシェーダとかを使用する時とかに使うんじ
ゃないかな。今回は考えなくてもいいや。

次に D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE ですが、これは文字を見て予想がつく。「データはシェーダから参照可能」というものだろう。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn859378\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn859378(v=vs.85).aspx)

中を見ると説明は

「D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE は、シェーダによる参照のためにコマンドリストにバインドされていることを示すために、ディスクリプタヒープ上にオプションで設定できます。このフラグを付けずに作成されたディスクリプタヒープでは、CPU メモリにディスクリプタをステージングしてから、シェーダの可視ディスクリプタヒープにコピーすることができます。しかし、アプリケーションがシェーダの可視ディスクリプタヒープに直接ディスクリプタを作成し、CPU 上に何かをステージングする必要はありません。」

と書かれています。正直よくわからない。そもそもディスクリプタヒープという名前が連発すると、これ、名前変えたほうがいいんじゃないかなと思う。

まあまとめるとシェーダから参照可能なものって思つておけばいいよ。

ちなみに

srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;

だが、これは、

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903814\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903814(v=vs.85).aspx)

に書かれているように

「The default 1:1 mapping can be indicated by specifying D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING, otherwise an arbitrary mapping can be specified using the macro D3D12_ENCODE_SHADER_4_COMPONENT_MAPPING. See below.」

D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING を使うとデフォルトの1:1マッピングが使用されるらしい逆に聞きたいんだけど、1:1マッピング以外を使用する時ってどういうパターン？

だめだ…ちょっと思いつかない。ということでデフォルトでいいと思います。

あとは CreateShaderResourceView を使用すればいいだけです。とはいもののそのために必要なものは、シェーダリソースビューハンドルを置くべき場所のディスクリプタハンドルなので、GetCPUDescriptorHandleForHeapStart を使用しています。

もし、ここまででうまい事いかない部分があつたら教えてください。

さて…ここまででテクスチャへの書き込みと、シェーダリソースビューの準備はできたと。残るはサンプラ設定と、それに伴うルートシグネチャの変更ですね。

サンプラ

サンプラーってのは sampler っていうもので、テクスチャから画素を得るときに、どういうルールで画素を持ってくるのかを設定するものだ。

必須なものかといわれると必ずしも描画のために必要なものではないのだが、サンプラーがない場合はデータとして「何ピクセル目」の画素値をとってくるような命令を出す必要があります。これは画像の特質によっては非常に面倒なことになるため通常はサンプラを使用してテクスチャマッピングを行います。

サンプラの設定

D3D12_STATIC_SAMPLER_DESC を使用して設定していきます。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986748\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986748(v=vs.85).aspx)

さっきも言ったようにサンプラーっていうのは、どういうルールでテクスチャの「画素値」を取得するのかということ。

```
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;//絵が繰り返される(U方向)  
samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;//絵が繰り返される(V方向)  
samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;//絵が繰り返される(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)

こんな感じ。

最後の奴は比較が云々言うとりますが、これは良く分からんんですね。サンプラーの段階で何と比較するというのだろう…書き込みの段階での比較は分かるんですけどねえ。で、案の定

D3D12_COMPARISON_FUNC_NEVER

にしても

D3D12_COMPARISON_FUNC_ALWAYS

にしても結果は変わりません。あんまり意味がないのかな。ともかくサンプラーを設定しました。

ルートシグネチャへサンプラーを適用する

既に書いているルートシグネチャの設定ですが

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

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

こんな感じになっていると思いますが、今一度ドキュメントを見てみましょう。

[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)

```
UINT NumParameters; // パラメータ数
```

```
const D3D12_ROOT_PARAMETER *pParameters; // パラメータ配列ポインタ
```

```
UINT NumStaticSamplers; // サンプラー数
```

```
const D3D12_STATIC_SAMPLER_DESC *pStaticSamplers; // サンプラー配列ポインタ
```

```
D3D12_ROOT_SIGNATURE_FLAGS Flags; // ルートシグネチャフラグ(前のと同じでいい)
```

というわけで、サンプラーを追加したのでサンプラーもルートシグネチャに登録してあげる必要があります。

ちなみに面倒なのは、このサンプラーを登録するとすると、同時にルートパラメータも登録しないといけないというオマケつきです(じゃないとパイプラインステートがしくじります)

ではルートパラメータについてみてみましょう。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn879477\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn879477(v=vs.85).aspx)

うわあ…久々にヘヴィな雰囲気だね。

ParameterType は普通に D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE でいいでしょう。

次にデスクリプターテーブル(DescriptorTable)だけどここには

「デスクリプターレンジ」というなんかまたけたいなものを指定する必要があります。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn859380\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn859380(v=vs.85).aspx)

直訳すると「デスクリプターの範囲」なんんですけどねえ…。

```
D3D12_DESCRIPTOR_RANGE_TYPE RangeType; // 範囲種別
```

```
UINT NumDescriptors; // デスクリプタ数
```

```
UINT BaseShaderRegister; // シェーダレジスタ(スロットに相当)
```

```
UINT RegisterSpace; // レジスタの範囲(大きさ?)
```

```
UINT OffsetInDescriptorsFromTableStart; // D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND でいい
```

さて、範囲種別なんですが、ここはシェーダリソースビューを使うので

```
D3D12_DESCRIPTOR_RANGE_TYPE_SRV
```

を指定します。ちょっとほかに関しては自分で考えてみてください。分からぬ場合はドキュ

メントとよ～～くにらめっこしよう。

ともかくこれでレンジの設定ができるので、レンジを設定した変数のポインタをルートパラメータの pDescriptorRanges に突っ込みましょう。

最後に ShaderVisibility ですが、こいつは、テクスチャの内容はピクセルシェーダから見えてほしいので、

D3D12_SHADER_VISIBILITY_PIXEL

とします。

で、ここまでルートパラメータ設定は終わるので、皿のこのルートパラメータのポインタをルートシグネチャの pParameters に放り込んでください。

シェーダ側にサンプラとテクスチャの設定を書き加える

ここまで CPU 側の設定はほぼほぼ終わりです。GPU 側は何をしたらいいのかと言うと、テクスチャとサンプラをそれぞれ GPU 側に用意します。実は GPU 側で指定するレジスタ番号というやつが、CPU でのスロットやらレジスタに対応していて、投げたテクスチャ情報がそこに入ります。

それでは shader.hlsl の先頭に

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

と書いてください。

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

ちなみに、番号の前の t だの s だのに関しては

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

見るといいと思います。

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

ということで、Texture2D の Sampler 関数を使用します。

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

```
tex.Sample(サンプラオブジェクト, UV 座標);
```

のようにして使います。ちなみにピクセルシェーダでのみ有效です。

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

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

シェーダリソースビュー用のデスクリプタを登録

まずは現在使用中のシェーダリソースビューをとります。取得の仕方は既に使っている descriptorHeapSRV のようなテクスチャ用のヒープをとってくれればいいです。

関数は CommandList の SetDescriptorHeaps です。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903908\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903908(v=vs.85).aspx)

関数名から予想はつくと思いますが、こいつも配列を望んでいます。まあ、今回は一つしかないので、ポインタのポインタを渡さざるを得ないんですが。

```
_commandList->SetDescriptorHeaps(1, (ID3D12DescriptorHeap* const*)&descriptorHeapSRV);  
こんな感じになるんですが、真ん中の引数の const に注意してください。元のポインタに  
const がついた入れ子みたいなもんです。わかんない人は要素数1の配列を作って、それを利  
用してください。
```

次に

SetGraphicsRootDescriptorTable をセットします。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903912\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903912(v=vs.85).aspx)

この第一引数は 0 でいいんですが、第二引数はどうしましょうか…。

GetGPUDescriptorHandleForHeapStart

を使用します。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899175\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn899175(v=vs.85).aspx)

これで、シェーダリソースビューのデスクリプタハンドルヒープを取得して、
SetGraphicsRootDescriptorTable

に突っ込めば…こんな感じに青葉ちゃんが表示されるようになります。



ん?今、青くなつたよね。青葉ちゃんだけに

【審議中】

($\wedge_{\omega} \wedge_{\omega}$) ($\wedge_{\omega} \wedge_{\omega}$) $\wedge \wedge$
 $\wedge_{\omega} \cup \wedge_{\omega}$ つと /($\wedge_{\omega} \wedge_{\omega}$)
| \cup ($\wedge_{\omega} \wedge_{\omega}$) と / $\wedge_{\omega} \wedge_{\omega}$
 $\wedge_{\omega} \wedge_{\omega}$ / $\wedge_{\omega} \wedge_{\omega}$

色化け対処

なんか青みがかったないかな…?何故かな~?というわけで、データの並びを改めて見てみたいと思います。

```
00 E4 E2 E8 00 E4 E2 E8 00 E6 E4 EA 00 E5 E3 E9  
00 E5 E3 E9 00 E5 E3 E9 00 E5 E3 E9 00 E5 E3 E9  
00 E7 E3 E9 00 E9 E2 E9 00 E9 E3 E8 00 E6 E3 E8  
00 E5 E4 E8 00 E3 E5 E6 00 E3 E4 E6 00 E2 E3 E7  
00 E3 DF E5 00 D0 C9 D1 00 C6 BD C6 00 C9 BE C9  
00 C7 BC C6 00 C4 BC C3 00 AF AB AF 00 7A 7E 80  
00 54 61 5D 00 48 5E 53 00 43 5D 4F 00 3D 5D 49  
00 39 5C 43 00 39 59 40 00 38 58 3C 00 37 59 3B  
00 32 57 39 00 30 56 38 00 30 56 38 00 2F 55 37
```

これ見てピンとくる人いるかなー?この現象とこのデータの並びを見たとき僕は



ってなりました。どういう事がと言うと、通常であれば並びは RGBA の並びになっていると思われるのですが、無理やり 24bitBMP を 32bitbmp にしているので、ARGB になっているわけです。まあこの程度の事は、シェーダの柔軟性をもってすれば解決できます。

RGBA が実は XRGB であり、X 部分が 0 であるので無効であるとします。そうなると R は使えないとになります。あ、そしてもうと言うと、並びはこうでした。XBGR。これはシェーダ側はどう書けばいいのでしょうか…ちょっと自分で考えてみてください。ちなみに現在の色化け状況は

```
tex.Sample(smp,o.uv).rgba
```

と同様の状態です。

さあ、考えてみよう。先ほども言いましたが、R 部分は無効だから使ってはいけない。有効なのは gba の部分のみ。さらに色の並びもひっくり返っている…シェーダは柔軟…つまり…

```
return float4(tex.Sample(smp,o.uv).rgb,1);
```

とやればかわいい青葉ちゃんが表示されるわけです。



やったぜ!!

と思っていたら問題発生。これ、まともそうに見えますが、実は偶然UV値が反転しちゃって気づかなかったんですよね…実は

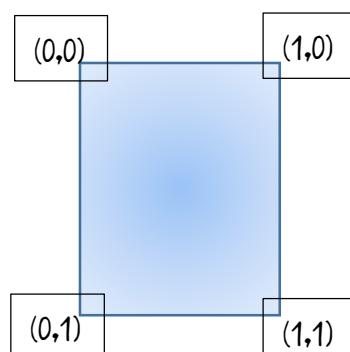


こういう風になってて正解

何故でしょか…理由があるのです。

<https://ja.wikipedia.org/wiki/%E3%83%93%E3%83%83%E3%83%88%E3%83%9E%E3%83%83%E3%83%97%E7%94%BB%E5%83%8F#.E5.BA.A7.E6.A8.99>

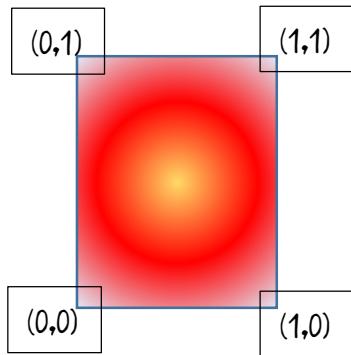
通常であれば2DやUVの座標系は…



こういう並びになっているはずだし、僕もそう思っていました。ところが違ったのです。

http://www.umekkii.jp/data/computer/file_format/bitmap.cgi

何故か左下から右上へとデータが流れているんですよね。つまり



つまり上下が反転しているのです。そりや馬鹿正直に表示すれば青葉ちゃんはひっくりかえるよなあ…(•ω•)

で、これ、右下から左上であれば単なる「反対読み」で何とかなるんですが、そういうわけでもなく、上下のみ反転なのでかなり面倒なのです。

そういう事を考えていくとやっぱり WIC のお世話になるのが妥当かな～って思つたりもします。あと今更ですが相手が Bitmap であれば LoadImage 関数が使用できるんですよね…でも一つ言っておくと LoadImage 使ってビットマップ読み込みするのは逆に面倒かもしれません。

Windows 特有のめんどくささがあるので…手間自体は同じかもっと面倒。ホワイトボックスであるぶん素直に fread で読み込んで加工することをお勧めします。

```
for (int line = bmpheader.biHeight - 1; line >= 0; --line) {
    for (int count = 0; count < bmpheader.biWidth*4; count += 4) {
        unsigned int address = line * bmpheader.biWidth * 4;
        fread(&data[address + count], sizeof(unsigned char), 4, fp);
    }
}
```

こういう感じですね。やってることはわかるでしょうか…？外側のループは下から 1 ラインずつ上がってて、内側のループで左から右まで読み込んでいます。

あと、今回は 32bit ビットマップでしたが、結局のところ MMD で使用されている bmp は 24bit です。さて、如何なものか。

ちなみにこれは DirectX11 のころから指摘されている模様

<https://gamedev.stackexchange.com/questions/125975/directx-11-r8g8b8-24bit-format-without-alpha-channel>

で、結局のところ 24bit を読み込むときに無理やり 32bit にしなければならず、24bit は HDD の容量を削る以外のメリットはないわけだ。ちなみに DX11 の時は

D3DXCreateTextureFromFile って関数がすべてやってくれていたのだが、結局は中で 32bit に変換していたのだと思われます。つまり…

```
data.resize(bmpheader.biWidth*bmpheader.biHeight * 4);

for (int line = bmpheader.biHeight - 1; line >= 0; --line) {
    for (int count = 0; count < bmpheader.biWidth*4; count += 4) {
        unsigned int address = line*bmpheader.biWidth * 4;
        data[address+count] = 0;
        fread(&data[address+ count + 1], sizeof(unsigned char), 3, fp);
    }
}
```

こんな感じの事をやる必要があるというわけや。ああめんどくさ



ビットマップ以外への対処(今はおまけ的な話)

とは言え問題はまだまだ山積みなのです。これ、ビットマップならばまだ対処のしようはあるのですが、png やら jpg になったときどうします? ちなみにデフォルトミクさんファミリーであれば、すべて BMP なので特にここは無理して乗り越えなくても良いでしょう。

ただ、デフォルトのミクさんルカさんメイコさんハクさんリンレンカイトネルであれば全部 BMP なので問題ないんですが、世の中に出回ってるモデルは png,jpg,tga が多用されています。

いや、安心させる話をすると png も jpg も libpng や libjpeg があるので、まあなんちゃないんですが、その辺いっぺんにやってくれるライブラリ(というかソースコード)もあるにはあるので紹介しておきます。

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

DirectXTex というライブラリがあります。Microsoft 提供で Github で公開されており、割と安心して使える部類ではないかと。

ちなみに tga については DirectXTex も対応しておらず、結局



いつかはこういう顔になるのは明白なのですが…(ぶっちゃけ TGA を使うメリットが思いつかないんだけど…たいていしてサイズは減らないし…シグネチャが先頭にないし…プログラマからするとクソみたいなフォーマットですよ!!何でこんなもん未だに使われてるんや!!)

ひとまずここから後はモデルを表示するところまで先延ばしにして、次の話題に入ります。次は 3D 化の話です。

3D化してみよう

お待たせいたしました。いよいよやってまいりました。3Dの世界へようこそ。



もうだめだあおしまいだあ

どうやって3D化するのか?もしかしてポリゴンにz値を入れたら勝手に3Dになるとでも思った?

残念。行列と幾何学的な変換が必要でした。ここからは数学的な地獄が始まります。

3D化するには3つの変換行列…の合成行列が必要となります。

どんな行列かと言うと

- ワールド行列(モデルを回転させたり移動させたり)
- ビュー行列(カメラ行列)←DX12における「ビュー」とは別物なので注意
- プロジェクション行列(スクリーンに射影する)

の

3つほどとなります。言うても行列を忘れてる人もいると思いますのでちょっとおさらいしましょう。

行列について再学習

座標の変換は数学の中でも「行列」というのを使っていきます。

すごい重要で、それ程難しくもないのに、世間知らずの教育委員会の糞野郎どもが高校カリキュラムから外しやがったので、馴染みのない分野となってしまった悲しいモノです。



「行列など社会で必要ない」とか、ゲームプログラマーを舐めとんのかつ!!!!ボケ!カス!アホンタラア!!!
ITのプログラマも今後はもっと行列の知識が必要になってくるのに…世間知らずの教育委員会のバカどもが!!

最初に大雑把に言うと $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ のように、数字を縦横に並べたものです。

ゲームプログラムにおいてはまず「乗算」しか使いませんので、乗算のルールだけ軽く説明し

ます。

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

という風に行列の乗算(しばしば×は省略されます)を求めたいとしたときに、この場合は答えも2×2行列になりますので、左上(1行1列目)、右上(1行2列目)、左下(2行1列目)、右下(2行2列目)を求める必要があります。

で、ルールとしては1行1列目を求めたい場合は…左式の1行目と右式の1列目を「かけて足す」です。つまり

$$\begin{pmatrix} 1 * 5 + 2 * 7 & ? \\ ? & ? \end{pmatrix} = \begin{pmatrix} 19 & ? \\ ? & ? \end{pmatrix}$$

と、こうなるわけです。

同様に右上(1行2列め)を求めたければ左式の1行目と右式の2列目をかけて足します。

$$\begin{pmatrix} 19 & 1 * 6 + 2 * 8 \\ ? & ? \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ ? & ? \end{pmatrix}$$

こんな感じです。大事です。さて下段に行きます。2行目と1列目をかけて足します。

$$\begin{pmatrix} 19 & 22 \\ 3 * 5 + 4 * 7 & ? \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & ? \end{pmatrix}$$

最後です。

$$\begin{pmatrix} 19 & 22 \\ 43 & 3 * 6 + 4 * 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

つまり一般化すれば

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

こういうことです。何となくわかりますかね？ちなみに行列には特徴があって「非可換」って特徴です。

通常僕らが使用している「数」は $3 * 5 = 5 * 3$ が成り立つし、 $a * b = b * a$ が成り立つでしょ？ところがこの行列とやらにはそれが成り立たないのです。

$$A \times B \neq B \times A$$

なのです。

試しに先ほどどの

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

を自分で計算してみて？ぜんぜん違う答えになるから。

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 5*1 + 6*3 & 5*2 + 6*4 \\ 7*1 + 8*3 & 7*2 + 8*4 \end{pmatrix} = \begin{pmatrix} 23 & 34 \\ 41 & 46 \end{pmatrix}$$

ね？似ても似つかないでしょ？

ですから行列ってのは乗算順序が超重要なのです。プログラミングにおいてはこれが最重要なので、今日はここを覚えて帰ってください。

ここまででは知ってると思いますし、まあ前期の数学のテストの状況を見ると細かい計算に不安がありますが、そこは大丈夫ひとまず知っておくべきことは順序が重要。くどいようですがね。

とりあえず教育委員会なるものはぶつ潰していいと思います。

行列による座標変換

さて教育委員会への熱い風評被害を与えたところで、この行列…何に使うのでしょうか？

それは座標を変換するためなのです。座標の移動や回転を効率的に行えるからな"のです。



ちなみにモデルってのは1万頂点くらいあるわけですよ。で、座標変換ってのは頂点に対する変換なのよね。例えば平行移動して回転したモデルをスクリーンに投影したいとする。そうすると変換するのに必要な計算回数は

1. 回転変換
2. 平行移動変換
3. カメラに合わせる変換
4. スクリーンに投影させる変換

の4回になります。本当はコレどろころじゃすまないんですが、少なくともこれくらい必要であると。で、これを1万頂点に対して処理するなら4万回の計算が必要になります。

ここで行列の性質として「行列は合成できる」→「変換は合成できる」という特性が役に立ちます。とりあえず比較的分かりやすい変換としてアフィン行列(アフィン変換)を考えてみましょう。

アフィン変換(アフィン行列)

<https://ja.wikipedia.org/wiki/%E3%82%A2%E3%83%95%E3%82%A3%E3%83%B3%E5%86%99%E5%83%8F>

Wikipedia の説明は数学的に正確すぎて分かりづらいため

<https://qiita.com/yuba/items/7fb6a49adfd08fa4bbd8>

でも見ておきましょう。

簡単に言うと、アフィン変換ってのは、平行移動、回転、拡大縮小、あと使わないけど「せん断」という操作を行って頂点の座標を別の座標へと移すものです。

例えば回転ならばコブラのマシンはサイコ・ガンで

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

こういう式になると思います。これを回転行列と言います。拡大縮小も、X 方向に N 倍、Y 方向に M 倍拡大縮小するのならば

$$\begin{pmatrix} N & 0 \\ 0 & M \end{pmatrix}$$

となります。でもここで問題が発生します。じゃあ平行移動はどうやって表現しようか? 平行移動ってのは現在の XY 座標に例えば(A,B)を足すだけであるから、元の XY の影響を受けてはいけない!…が、

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

式を見れば明らかのように、このままでは何をどうやつたって a も b も c も d も xy の影響からは逃れられない…という事で考え出されたのが「同次座標系」というものです。

何かというと、X 行 Y 行に加えてもう一行…1 を加えた座標系を考えます。これにより平行移動も拡縮や回転と同時に表現できるようになりました。これが座標変換において革新をもたらします。

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

これが同時座標系の座標表現ね。

で、まあゆーたら xy の2次元座標系なのに3行になってるわけや。ではなく 1 が入っとる。ともかくそれはちょっとこういうもんやと思っておいて。この後に意味が分かるから。

3行になってしまったからには変換行列も3行3列にせなあかんな。とすると、こうなる。

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + c \\ dx + ey + f \\ gx + hy + i \end{pmatrix}$$

さて…おわかりいただけただろうか? c, f, i が xy の影響を受けていないという事を。つまりここを平行移動として使う事ができるという事だ。 1 がかけられているため、3列目がそのまま平行移動成分となるのだ。つまり単位行列で書くとこういうこと

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

で、前にも書いたけど、変換行列というのは合成できるわけ。つまり…

例えば、 (a, b) だけ平行移動したいとすると…

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

こうなるわけで、じゃあ、回転して平行移動ならどうなるか? というと

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

こういう行列になる。これを合成してみよう。紙に書くなりなんなりして計算してみてください。なお、数学的なルールとしては順序が左に左にかけていく事になるので注意(通常だと右に右にかけていくものだが)。あと、DirectX は左手系なので、右に右にかけることになりますが、ひとまずはこれを計算してみてください。

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & a \\ \sin\theta & \cos\theta & b \\ 0 & 0 & 1 \end{pmatrix}$$

こうなるはずです。この形を見ると確かに回転と平行移動が合成されていることがわかりますね?

あともう一つ言うと、原点から離れた場所で、その場で回転したい場合はちょっとややこしくて3つの操作が必要になる(「回転」の操作は必ず原点中心に行われるため)

1. 原点へ移動(現在位置をそのままマイナス)

2. 回転(原点中心回転)
3. 元の座標へ平行移動(原点からもとの座標をプラス)

となる。何故かはノートに書いたり自分の手で手遊びしながら思い浮かべてほしい。となると

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix}$$

こうなる。これもできれば自分で計算してみてほしいのだが

$$\begin{aligned} & \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & -a\cos\theta + b\sin\theta \\ \sin\theta & \cos\theta & -a\sin\theta - b\cos\theta \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos\theta & -\sin\theta & -a\cos\theta + b\sin\theta + a \\ \sin\theta & \cos\theta & -a\sin\theta - b\cos\theta + b \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

という風になり、ちょっとややこしいのだが、この行列一つで先ほどの 3 操作を一つにまとめて表現できるわけだ。

ちなみに数学の慣習として平行移動行列を T と書き、回転行列を R と書くのですが、元に戻す行列を T' とすると求めた新しい行列 M は

$$M = TRT'$$

と書けます。このように合成した行列を作ることができるので、頂点ひとつひとつに行列を一個一個やってくのに比べると処理を減らすことができます。

…ていう感じになってると思ってください。

ちなみに平行移動と回転と拡大縮小の行列以外にも「ビュー行列(カメラ行列)」「プロジェクション行列」ってのがあってそれぞれこんな感じ

$$\begin{aligned} Z &= \text{Normalize}(E - P) \\ X &= \text{Normalize}(U \times Z) \\ Y &= Z \times X \end{aligned}$$

$$M_{\text{view}} = \begin{pmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ -P \cdot X & -P \cdot Y & -P \cdot Z & 1 \end{pmatrix}$$

ビュー行列

$$\begin{aligned}
 & \left(\begin{array}{cccc|cccc}
 \frac{2}{right-left} & 0 & 0 & 0 & 1 & 0 & 0 & -\frac{right+left}{2} \\
 0 & \frac{2}{top-bottom} & 0 & 0 & 0 & 1 & 0 & -\frac{top+bottom}{2} \\
 0 & 0 & \frac{-2}{far-near} & 0 & 0 & 0 & 1 & \frac{far+near}{2} \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
 \end{array} \right) \\
 = & \left(\begin{array}{cccc|cccc}
 \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\
 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\
 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\
 0 & 0 & 0 & 1
 \end{array} \right)
 \end{aligned}$$

プロジェクション(投影行列)

難しいですね。中身についてはまたあとでお話しましょう。で、ちなみにモデルに対する回転、拡大縮小、平行移動をまとめて「ワールド行列(ワールド変換)」もしくは「モデル行列(モデル変換)」と言い、「W」で表すことが多いです。

その後に書いたカメラ行列(ビュー行列)は「V」で表すことが多いです。最後の射影行列は英語で「プロジェクション」というため「P」で表すことが多いです。文字通り

W×V×P の意味です。

前述のとおり

1. 平行移動(原点へ)
2. 回転変換(原点、中心回転)
3. 平行移動変換(元の座標へ)
4. カメラに合わせる変換
5. スクリーンに投影させる変換

これだけの操作があるわけですが、これを1万ポリゴンに適用した場合の計算量は1万×5操作=5万操作になるわけだが、この行列の合成できるという特性をもってすれば操作は「原点へ平行移動し回転し元の座標へ戻しカメラ座標へ合わせスクリーンに投影する」行列をかけねばいいだけなので、計算回数は1万となる。あ、最初に合成する計算を考えると

5+10000で10005となるわけだけど、本来かかるはずだった計算量50000と比べるとずいぶんと小さいでしょ？また、最近のCPUに特殊な命令で計算させたり、GPUに行列計算させると通常の計算よりも高速になります(最近のプロセッサは並列計算が超得意なため)

という事でゲームプログラミングというか、DirectXやOpenGLでは行列とその合成が頻繁に

使われます。

ワールドビュープロジェクションこれらを合わせて(合成して)WVP 変換とか言ったりします。
ここまではいいかな?

それぞれの行列を作ってみよう

まあ行列を作るなんてクソ難しそうだね。え? 完全に理解した?



そういうギャグは命を縮めるぞ

一応ね、行列の構造とか行列の中身については知っておいてほしいのはやまやまなんだけ
ど、プロジェクト自作するとか言っちゃうとたぶん誰かが死ぬのでそこはお便利機能
に頼りましょう。DirectX もそこまで無慈悲ではないです。

まず DirectXMath.h をインクルードしていると思いますが、これをインクルードしていると
3D プログラミングに必要なやつが大抵そろっていると思っていい。もちろん行列もだ。

名前は XMMATRIX です。

ワールド行列

で、だいたい行列の初期化では「単位行列」を代入します。というわけで

`XMMATRIX world=XMMatrixIdentity();`

なんて書くわけですよ。そしたらですねアホエディタがですね。

`XMMATRIX world=XMMatrixIdentity();`

ご覧のように赤フニャ出しそるんですわ。なめこんのが貴様。で、この原因はですね。
DirectXMath.h 側が DirectX::XMMatrixIdentity() を定義していて、それはいいんですが、バカが
DirectXMathMatrix.inl でも XMMatrixIdentity() を定義しとるんですね。

エディタが混乱して文法エラー出しどるわけですけどこの DirectXMathMatrix.inl は誰からも
インクルードされてないのよね…。だから試しに赤フニャ出た状態でコンパイルしてみてく
ださい。コンパイルが通ると思います。つまりインテリセンスのバグです。

お前なんか全然インテリでもない! シセンスもない! わこの野郎!!

ど~もしても赤フニヤ気持ち悪いって人は関数の前に Direct:: って書いてください。

まあそれはともかく、こういう便利な関数があるので、これを利用して行列を作つていきましょうという事だ。

ちなみに XMMatrixIdentity() は「単位行列」を返すものです。
ひとまずは回転とかさせないでこれをワールド行列とします。

カメラ行列(ビュー行列)

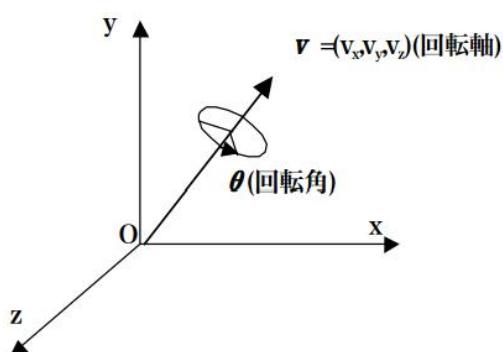
次にカメラ行列を作ります。XMMatrixLookAtLH 関数を使用します。

[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)

まあ今までのに比べると比較的分かりやすいでしょ? 日本語だし。ただめんどくせーのはそれぞれの引数。

第一引数に視点、座標、第二引数に注視点、座標、最後に上ベクトルを入れます。この上ベクトルを入れる理由はあとで話します。視点は勿論カメラの座標。注視点は「注視するもの」の座標だから、TPS とかではマウスカーソルが指す先とかになるね。

最後の上ベクトルについてですが、簡単に言うと
、視点から注視点を見るベクトルだけだと回転しちゃうでしょ? だから上ベクトルを基準として与えてるのよ。



「上ベクトル」がないとくるくる回っちゃう

とりあえずしばらくの間は上ベクトル 0,1,0 で問題ありません。ライトシミュレーターみたいに上がくるくると回っちゃう奴のときだけ気にすればいいです。地上にいる間は決め打ちでも構いません。

視点を eye、注視点を target、上ベクトルを upper として定義するとこんな感じ

```
XMVECTOR eye = {0,0,-10};  
XMVECTOR target = {0,0,0};  
XMVECTOR upper = {0,1,0};
```

視点の Z が -10 なのは、現在の頂点座標が 0 だからです。視点に近づきすぎると消えるからです。なお、それぞれの型が XMVECTOR である点に注意してください。XMFLOAT3 ではありません。これには計算効率的な理由があってこうなってるんですが、しばらくは「何故か型が違う」という認識でいいでしょう。

XMMatrixLookAtLH という関数を使います。ちなみに LH は LeftHand(左手系)の略です。

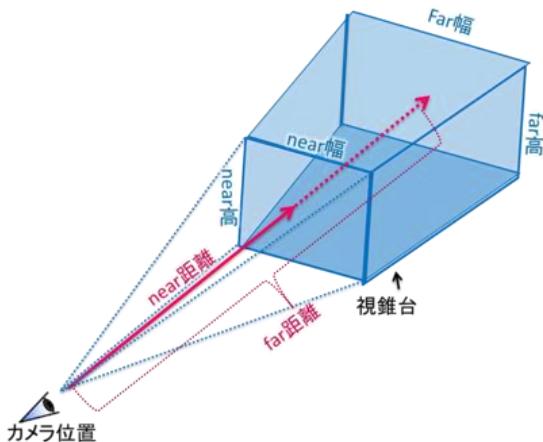
[https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixlookatlh\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixlookatlh(v=vs.85).aspx)
XMMATRIX camera = XMMatrixLookAtLH(eye, target, upper);

これでもう camera の中にカメラ行列が入ります。残りはプロジェクション行列だけです。

プロジェクション行列(射影行列)

ワールドやカメラはともかく、このプロジェクション行列はなじみがないものだと思います。簡単に言うとプロジェクション行列は奥行きを表現するための行列です。遠くに行けば行くほど小さくなります。遠近法の計算をする行列なんです。

計算的に言うと 3D 空間上で以下のような 3D 台形のような感じになっているものを

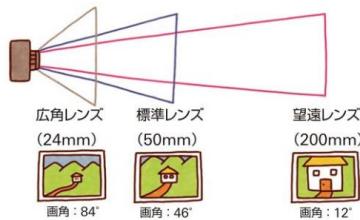


無理くりスクリーン板の形にしてしまうような変換です。見ての通り手前は小さく遠くは大

きい…というものを1枚板にするものだから遠いものが小さくなるわけです。この台形(視錐台)の広がり角度を「画角」と言ったりします。

この画角が広ければ広いほど、ピースがきつくなります。画角が狭ければ、ピースがゆるやかになります。

カメラに詳しい人なら分かると思いますが、望遠レンズと広角レンズの違いみたいなもんです。



広角だとちょっと離れただけで無茶苦茶小さくなります。望遠だと離れてもそれほどピースがかかりません。まあそういうものがあると知っておけばいいです。3Dにおいて画角を動的に変更する場面があるとすると、スピードアップというかブーストした時に画角を広角にすることによってよりスピード感を増すというそういう手法もあつたりします。

まあ現段階ではどうせよくわがんねーと思いますので、ちゃんと作ります。プロジェクション行列は

XMMatrixPerspectiveFovLH 関数を使用して計算します。MS の説明をよく読んでください。

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

さて、このパラメータが良く分からぬと思いますが、第一引数は先ほど説明した画角です。 $0^\circ \sim 90^\circ$ くらいの中で適切な角度を指定してください。

次に高さと幅のアスペクト比についてですが、これは簡単に言うと縦横比のことです。横縦比かな？

で、さっきの視錐台の絵を考えながら、近いほうと遠いほうの奥行きをそれぞれ入れてみてください。カメラとオブジェクトの位置を考慮しながらね。

これでプロジェクション行列(ピースペクティブ行列)ができます。

行列を合成しよう

簡単です。乗算すればいいだけ。

```
XMMATRIX matrix = world*camera*projection;
```

で、これ、二次元座標系と違うので、頂点の座標はそれに合わせてまた変更しておいてください。

さて、あとはこの合成した行列を GPU 側に投げなければならぬ…どうすればいいのでしょうか？

テクスチャでもない、頂点データでもない…これを GPU 側に投げるには一体どうしたら…。

という所で出てくるのがコンスタントバッファ(定数バッファ)です。

[https://msdn.microsoft.com/ja-jp/library/ee422115\(v=vs.85\).aspx#Shader_Constant_Buffer](https://msdn.microsoft.com/ja-jp/library/ee422115(v=vs.85).aspx#Shader_Constant_Buffer)

座標変換データを GPU に送ろう

もちろん行列データをそのまま送れるわけではありません。何かしらのバッファにいれてなげないと GPU 側では受け取ってくれません。それが定数バッファです。

定数バッファを作ろう

必要なものは

```
ID3D12Resource* _constantBuffer=nullptr;  
ID3D12DescriptorHeap* _cbvDescHeap=nullptr;
```

で、これを作るために

```
D3D12_DESCRIPTOR_HEAP_DESC  
D3D12_CONSTANT_BUFFER_VIEW_DESC  
D3DX_HEAP_PROPERTIES
```

が必要です。もうそろそろ慣れてきましたかね。

まず CreateCommittedResource で _constantBuffer を作りましょう。

ああ、そういうばっちょっと面倒くさい制限が定数バッファにはあるんですよこれが。

256 バイト境界

なんのこっちゃ…と思うかもしれません。DirectX 系にはこういうのが多いんですよ。何かと計算を高速化するために区切りを切りのいい整数値にする必要があります。なお、256 のどこが「区切りがいい」のか、プログラマだったらわかるよな?

で、実はこの制約。僕もまた初耳です。DX11 の頃はこのような制約を意識する必要はなかったように思えるのですが…まあ恐らく内部でパディングしてくれただけかもしれません。

DX11 の時の制約と言えば…

[https://msdn.microsoft.com/ja-jp/library/ee418725\(v=vs.85\).aspx#TypeUsageGuidelines](https://msdn.microsoft.com/ja-jp/library/ee418725(v=vs.85).aspx#TypeUsageGuidelines)

16 バイト境界にあわせると…そういうのがあります。↑のは SIMD だの SSE だので高速化するために必要な制約なんですが、今回の 256 バイト境界ってなんなんでしょう。そしてこの 256 バイトに関する説明がほとんどないんですよこれがまた。

ヒントは DX12 のサンプルコードでした。

```
cbvDesc.SizeInBytes = (sizeof(SceneConstantBuffer) + 255) & ~255; // CB size is required to be 256-byte aligned.
```

この1行のみですよ。ふざけんなっての!!!インターネットでも探ししましたが
「256バイト境界らしい」、「256バイト揃えじゃないとダメらしい」とらしいらしいの連発ばかり。
俺はMSDNの公式見解が聞きたいんだぜ!!それを見るまで信用しないんだぜ!!!

を見つけました

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn903925\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn903925(v=vs.85).aspx)

の真ん中くらいに書いてありました。

The root signature must be specified if and only if the command signature changes one of the root arguments.

For root SRV/UAV/CBV, the application specified size is in bytes. The debug layer will validate the following restrictions on the address:

- **CBV – address must be a multiple of 256 bytes.**
- Raw SRV/UAV – address must be a multiple of 4 bytes.
- Structured SRV/UAV – address must be a multiple of the structure byte stride (declared in the shader).

CBV つまり定数バッファのアドレスは 256 バイトの倍数でなければならない…と。ひどいよ、
こんなのがんまりだよ。



(` ; ω ; `)ウッ…

ちなみに d3d12.h の中に

D3D12_CONSTANT_BUFFER_DATA_PLACEMENT_ALIGNMENT=256
という定数も見つけてしまいました。これはもう疑う余地もなく 256 バイトアライメント
のようです。

ちなみに 256 バイトアライメントについて書いてる日本語のサイト

<http://gameproject.jp/20160814-02/>

<https://glhub.blogspot.jp/2016/07/dx12-getcopyablefootprints.html>

なに?テクスチャのピッチも256バイトアライメント…だと?

<http://www5d.biglobe.ne.jp/~noocyte/Programming/Alignment.html>

基本的なことが書いている

まあ…その…なんだ。愚痴っても仕方ない頑張ろう。

まずはデスクリプタヒープから作りましょう。

D3D12_DESCRIPTOR_HEAP_DESC を設定

[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)

```
D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc = {};
cbvHeapDesc.NumDescriptors = 1;
cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;//シェーダから見えますように
cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;//コンスタントバッファです
result = dev->CreateDescriptorHeap(&cbvHeapDesc, IID_PPV_ARGS(&_cbvDescHeap));//いつもの
```

次は定数バッファそのものを生成。そのために CreateCommittedResource を使うのですが、

[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)

当然ながら頂点バッファの時とも、テクスチャの時とも設定が違います。

D3D12_HEAP_PROPERTIES はあまり設定的には難しくないです。

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770373\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn770373(v=vs.85).aspx)

今回はページプロテイもメモリプールもアンノウンでオッケーです。

です。またマスクはどちらも1にしておいてください。

```
D3D12_HEAP_PROPERTIES cbvHeapProperties = {};
cbvHeapProperties.Type = D3D12_HEAP_TYPE_UPLOAD;
cbvHeapProperties.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;
cbvHeapProperties.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;
cbvHeapProperties.VisibleNodeMask = 1;
cbvHeapProperties.CreationNodeMask = 1;
```

次にリソース設定ですが、前述の256バイト境界が出てきます。めんどう。

```
cbvResDesc.Dimension = D3D12_RESOURCE_DIMENSION_BUFFER;//単なる1次元バッファなので
cbvResDesc.Width = (sizeof(XMMATRIX) + 0xff)&~0xff;//256アライメント
cbvResDesc.Height = 1;//1次元なんで1でいい
cbvResDesc.DepthOrArraySize = 1;//深さとかないんで
```

```
cbvResDesc.MipLevels = 1; //ミップとかないんで  
cbvResDesc.SampleDesc.Count = 1; //これ1に意味ないと思うんだけど無いと失敗  
cbvResDesc.Layout = D3D12_TEXTURE_LAYOUT_ROW_MAJOR; //SWIZZLEとかしねーから。
```

あとはいつもの感じでリソースを生成してください。

とりあえずできたリソースを_constantBufferとして話を進めます。
あといつものようにコンスタントバッファビューも必要です。本当に面倒だな。
cbvDesc.BufferLocation=_constantBuffer->GetGPUVirtualAddress();
cbvDesc.SizeInBytes= (sizeof(XMMATRIX) + 0xff)&~0xff; //256アラインメント
dev->CreateConstantBufferView(&cbvDesc, _cbvDescHeap->GetCPUDescriptorHandleForHeapStart());
あと、一応言っておくとここで 256 アラインメントしてればリソース側の 256 アラインメントがなくても実行に支障はないようです。ないようですが怖いのでどちらも 256 アラインメントしています。

あとはマップして中に行列を放り込んで行きたいところなのですが、またもやルートシグネチャに「定数バッファを使うよ」と教えてあげなければならぬようです。

で、ルートシグネチャの所に戻ってもらって、

```
ルートシグネチャーDesc.NumParameters = 2;  
ルートシグネチャーDesc.pParameters = ルートパラメータ;
```

としてほしいのですが、勘のいい人ならお分かりのとおり、パラメータが増えるため、ルートパラメータを配列にしなければなりません(もちろんベクタでもOK)。とりあえずルートパラメータを配列化して

```
ルートパラメータ[1].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;  
ルートパラメータ[1].DescriptorTable.NumDescriptorRanges = 1;  
ルートパラメータ[1].DescriptorTable.pDescriptorRanges = &定数バッファ用デスクリプタレンジ;  
ルートパラメータ[1].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
```

実は設定自体はテクスチャの時とほとんど変わらない。ShaderVisibilityがAllもしくはVertexになることと、定数バッファ用デスクリプタレンジを作つてそれを突っ込むことくらいだ。

で、定数バッファ用デスクリプタレンジとシェーダリソースビュー用の違いはというと

RangeTypeだけだ。ご想像のとおり、SRVをCBVにすればいい。それだけだ。

ここまで設定したら、もういちどルートシグネチャー設定がうまくいっているのかを確かめよう。

問題なければ最後の仕上げだ。

あとで説明はするので、こんな感じでループ前にこう書いてくれ。

```
D3D12_RANGE range = {};
result = _constantBuffer->Map(0, &range, (void**)(&matrixAddress));
*matrixAddress = matrix;
```

今回はUnmapしなくていい。

次にループの中で、デスクリプタヒープのセットを行う。

```
_commandList->SetDescriptorHeaps(1, &_cbvDescHeap);
_commandList->SetGraphicsRootDescriptorTable(1, _cbvDescHeap->GetGPUDescriptorHandleForHeapStart());
```

あとはシェーダ。今回は頂点シェーダのみにする。まず、定数バッファの受け取り先を書く。

これは関数外で描いてくれ

```
cbuffer mat:register(b0) {
    float4x4 wvp; //WorldViewProjectionぎょうれつ
}
```

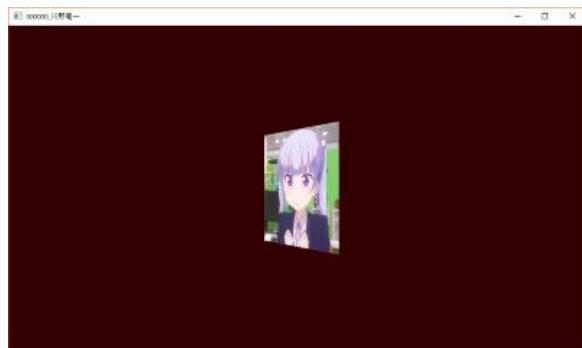
で、頂点シェーダの中身だが、引数の座標を表すパラメータをposとすると、posに対して行列の内容を反映させる。乗算だ。この乗算の順序がCPP側と逆になるので注意。

つまり CPP 側であれば右に右に乗算していたが、シェーダ上では数学のとおり左に左にかけていく。

そして行列の掛け算は * ではなく mul 関数を使用してほしい。行列の乗算には * は使えないらしい。

うまい事いければ、きちんと表示されるだろう。

うまい事いった人はカメラの座標を変えてみたり、world を回転行列に変えてみたりしてほしい。



ワールドに XMMatrixRotationY を使って回転させればこのようになります。奥行きがあるように見えるだろう？

ちょっとパッファとビューについてのたとえ話

パッファとビューについての考え方だが、ジュースとストローみたいな感じでとらえてほしい。



ジュースがパッファで、ストローがビューね？で



これではお気に召さないのである



GPU はパッファというジュースを渡してもそのままでは飲んでくれないのだ。

ストロー(ビュー)を付けてあげないと飲んでくれない。

非常にわがままな奴だが、こちらは仕事をお願いする立場だ。我慢しよう。逆に考えるんだ。ストローを渡すだけで気持ちよく仕事をしてもらえると。



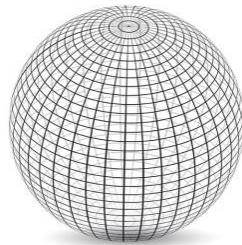
このとおりである

で、これは後から出てくるのだが、ストローを複数ぶつ刺して使う場面が出てきます。どういう場面かは今の所言えないけれど、「ある」という事を知つておいてください。

メッシュの表示

さて、次は当然というかなんといふかメッシュの表示である。メッシュとは何だろう？それは3Dオブジェクトを構成する頂点(辺、面)によって、様々な形を表現しているものを「メッシュ」と言います。

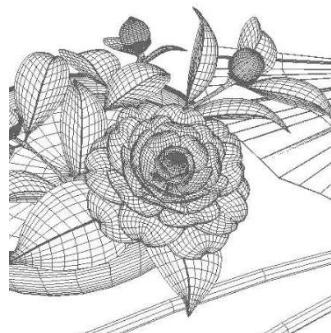
何故かと言うと



上のモデルのようなものを表しているのがメッシュと言います。図を見れば分かるようにストッキングみたいにな穴ぼこになります。このような網タイツ的な構造を「メッシュ」って言います。



こういうのをメッシュというために3Dにおけるポリゴンが三角形の集合体であることから、モデルそのものをメッシュと言うようになりました。



網タイツみたいでしょ？

さて、そのメッシュ(モデル)ですが、様々なフォーマットがあります。今回はメインターゲットとしてPMDを使っていこうと思います。MikuMikuDanceで使用されている最も基本的なモデルデータです。

もちろん、世の中で広く使われているモデルは FBX とかそういうのなんだけど PMD は必要最低限のデータしかないだけに扱いやすく(ちょっとクセはあるんだけど)、あとがわいいモデルデータを選び放題ってメリットもあるね。というわけで PMD データを読み込んでそれを表示していきましょう。

メッシュ(PMD)の表示

PMD を使用するにはデータの仕様の把握が必須です。一応 PMD に関しては「通りすがりの記憶」というサイトに全て載っています。

http://blog.goo.ne.jp/torisu_tetosuki/e/209ad341d3ece2b1b4df24abf619dbe4

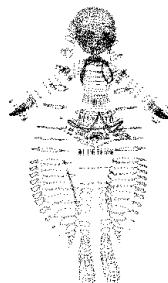
正直どうやって解析したのか分からん。が、ありがたく利用させてもらいます。データは

- ヘッタ1
- 頂点リスト2
- 面頂点リスト3
- 材質リスト4
- ボーンリスト5
- IK リスト6
- 表情リスト7
- 表情枠用表示リスト8
- ボーン枠用枠名リスト9
- ボーン枠用表示リスト10

という。この記事 8 年前の記事ですよ。確かに古いフォーマットと言われても仕方ないかなーって思いますが、まあ無意味に大変なところをやるよりはこっちの方がいいんじゃないかなと。

まあ PMD にも先頭 3 バイトくらいは無意味な大変さがあるのですが…

ひとまずまずは練習としてプレーンなミクさんを表示するところからやっていきましょう。まずは「頂点データのみ」からミクさんを表現します。



こんな感じのが出力されるところまでやりましょう

ひとまずヘッタデータを取得しますが、どれが正しいのかわからないので、まず色々と準備し

ます。

まず MikuMikuDance を落としてきてください。必須じゃないんですが、最終出力のイメージを持つといった方がいいと思います。また、データ解析のためにサーバの DirectX12 の中の PMD.SYM を自分のローカルに持っていてください。

ちなみに“初音ミク.pmd”をバイナリエディタで開くと

J000050 ED 64 00 00 80 3F 89 89 B9 83 7E 83 4E 00 7nd ?初音ミク
J0001FD FD 50 6F 7C 79 4DPolyM
J0020FD F9 70 93 82 83 66 33 88 83 66 91 58 83 95 81 90+モモデルデータ
J003046 8F 89 89 B9 83 7E 83 4E 20 76 65 72 2E 31 2E F初音ミク ver.1.
J004033 0A 28 95 8A 97 90 89 88 8E 5A 91 0E 89 9E 83 3 (物理演算対応)
J005082 83 66 83 8B 29 0A 04 83 82 83 66 83 8A 83 95 (c)ル モデリノ
J006083 4F 09 81 46 82 A0 82 C9 82 DC 82 B3 8E 81 0A グル : あにまさ氏
J007033 66 81 5B 83 95 9E CF 8A B7 09 81 46 8B 9E 20 データ変換 : 京
J00808F 4F 90 6C 8E 81 0A 43 6F 70 79 72 69 67 68 74 秋人氏 Copyright
J009009 81 46 43 52 59 50 54 4F 4E 20 46 55 54 55 52 : CRYPTON FUTUR
J00A045 20 40 45 44 49 41 20 20 49 4E 43 00 FD FD FD E MEDIA, INC ...
J0080FD FD
J0000FD FD FD

こんな風になっていますが、PMD.SYM をシンボルファイルとして使うと…

```
header.magic[0]
header.version
header.model_name[0]
header.model_name[16]
header.comment[0]
header.comment[16]
header.comment[32]
header.comment[48]
header.comment[64]
header.comment[80]
header.comment[96]
header.comment[112]
header.comment[128]
header.comment[144]
header.comment[160]
header.comment[176]
header.comment[192]
header.comment[208]
header.comment[224]
header.comment[240]
vert_count
vertex[0].pos[0]
vertex[0].normal_vec[0]
vertex[0].uv[0]
vertex[0].bone_num[0]
vertex[0].bone_weight
vertex[0].edge_flag
vertex[1].pos[0]
00000234C
3F95844D 418D1A37 BE9C91D1
3F48E5AF BEA8474B BF068654
00000000 3F800000
0003 0000
64
00
3FA60419 A18ED0A51 BF9E915C
```

そこそこ意味のあるデータが見えてきます。まあここまで見ても分からぬるのでひとまずデータフォーマットを説明した後に中身を確認していきましょう。

ヘッダデータは

http://blog.goo.ne.jp/torisu_tetosuki/e/acbaa40b23783309c8db5023356debba

に説明があるのでですが、

- 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

という感じで入っていきます。

さて、それではこれをそれぞれ fread してみましょうか。ちょっとそれぞれ fread して確かめてみてください。

この最初の3バイトの事を通常「シグネチャ」って言いますが、今回はDirectX12のルートシグ

ネチャと間違えやすいのでファイル種別を表す文字列と思っておきましょう。つまりいくつかのファイルフォーマットで言われてる「ファイルタイプ」と呼んでおきましょう。これが PMD の場合最初の 3 バイトです。最初の 3 バイトを読み込んで PMD であることを確認してください。

次に 4 バイト読み込んで、1.00 であることを確認してください。

次に 20 バイト読み込んで「初音ミク」であることを確認してください。

その次に 256 バイト読み込んで

「PolyMo 用モデルデータ：初音ミク ver.1.3
(物理演算対応モデル)

モデリング : あにまさ氏
データ変換 : 京 秋人氏
Copyright : CRYPTON FUTURE MEDIA, INC.
と書かれているのを確認してください。

そしてその次の 4 バイトを読み込んで頂点数(9036)を確認してください。

できましたか？

できたらこれらを一つの構造体にまとめて、いっぺんに読み込んでみてください。ReadFile の回数は少なめに越したことはないので。

さて…どうでしょう？とりあえずやってからこれ以降の話は聞いてね。

先にネタバレ見るようなズルはするなよ？

で、結果としては

| | | |
|---|-----------|---|
| ▶ | type | 0x006ff100 "Pmd..." |
| ▶ | version | -9.44167896e-30 |
| ▶ | name | 0x006ff108 "演ケミク" |
| ▶ | comment | 0x006ff11c "olyMo用モデルデータ：初音ミク ver.1.3..." |
| ▶ | vertexNum | 0x4d000023 |

ご覧のようにデータが壊れてしまします。

さて…何故なんでしょう?ひとつひとつでやった時には大丈夫だったのに…なぜなんでしょう。これには非常に面倒な問題が隠されているのです。

4バイトアライメントに注意

DirectXと関わらなくてもアライメント問題は絡んできます。ほんまは DxLib だろうと何だろうと絡んでくる問題ですね。

本当は MMD の作者がフォーマット作るときにこの辺の配慮があつたら特に直面する問題ではなかつたのですが…実はこういう意味でも学習に最適かなーって思うわけです。

どういう事を説明します。

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 などでのバイト数指定は通常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 バイトと扱い、余った 1 バイトはパディ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。

構造体アライメント って問題が発生しているのです。これを考慮しないとデータがガンガンぶつ壊れてしまう。例えば、今回の場合はバージョンがトンでもない数になったり、頂点数が 0 になったりする。もうね、こうなつたらデータは使い物にならん。

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…処理系依存といふべきか…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。



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(おせつかれにも?)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。

そういうわけで予想もしない数が入っているのである。まああくまでもこれはメモリの話なので、ファイル上ではまったく関係ない話なのだが…。

さて、これを解決するには2つの選択肢がある。

まずはsignature(3)だけ別にして読み込む。つまり、

```
struct PMDHeader{
```

```
    float version; // バージョン
```

```
    char name(20); // 名前
```

```
char comment[256];//コメント  
unsigned int vertexCount;//頂点数  
};
```

とし、

```
fread(&pmdfiletype,1,3,fp);//シグネチャ読み込み  
fread(&header,sizeof(header),1,fp);
```

とするのである。

次に#pragma packを使う方法。

C言語で開発する場合、このアライメント問題は結構出てくる。特にメモリヶ切ってた昔はそうだったのだろう。ということで、言語仕様として既に対処されていたりする。

要は、ムリヤリまとめる単位を変えてしまうのだ。通常は4バイトになっているそれを1バイトにすれば、余計な詰め物は発生しない。

で、C++言語には#pragma packってのがあるんですよ。アライメントの丸めるバイト数を指定するディレクティブですね。こいつに一時的に1を設定します。

ですから

さっき作ったマテリアル構造体の宣言前に pack(1)として、終わったら規定値にするために pack()にします。つまり

#pragma pack(1)

構造体宣言

#pragma pack()

最後にデフォルトに戻す pack()を忘れないようにね。

直値を入れてもいいし、シグネチャを外してもいいし、#pragma pack をやってもいいけど、とにかく 283 バイト目まで飛びました。

ここから fread 関数で 4 バイト読み込んで unsigned int 型変数に入れてください。ミクであ

れば9036くらいの数が入っていれば正解です。

```
unsigned int vsize=0;  
どこかで頂点数を表す変数を宣言しておいて…  
fread(&vsize,sizeof(unsigned int),1,fp);  
こういう感じで記述して、vsizeに頂点数っぽいのが入っていれば成功です。ミクモデルだと  
9000ちょっとくらい。
```

如何でしょうか？

では、実際に頂点情報を読み込んでいきますが、頂点はFLOAT3つではありません。

http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b1be2fb10b7838dfcbb0d010389707

はい、これくらい色々な情報が入っています。1頂点に入ってるのに、これが9000個くらいあるというわけです。

38バイトです。38*9000くらいなわけです。多いよ。そして、最後にあるWORDとかBYTEがまた曲者ですわ。どちらにせよ38バイトではまたパディングが発生してしまい、例えば

```
struct PMDVertex {  
    XMFLOAT3 pos; //座標(12バイト)  
    XMFLOAT3 normal; //法線(12バイト)  
    XMFLOAT2 uv; //UV(8バイト)  
    unsigned short bornNum(2); //ボーン番号(4バイト)  
    unsigned char bornWeight; //ウェイト(1バイト)  
    unsigned char edgeFlag; //輪郭線フラグ(1バイト)  
};
```

というような構造体を作ったとします。そして sizeof(PMDVertex)を測ると…40バイトという答えが返ってきます(本当は38なのに)

これもまたデータが狂う原因なので、仕方なくパッキング1を使用するか、もう38ってわかつてんんだから頂点全部読み込むってことで

```
fread(適当なバッファ,38*頂点数,1,fp);
```

とやります。つまりパック1にして

```
std::vector<PMDVertex> _vertices(pmdheader.vertexNum);  
fread(&_vertices[0], sizeof(PMDVertex), pmdheader.vertexNum, pmdFp);  
とやるか  
std::vector<char> _vertices(38 * pmdheader.vertexNum);
```

```
fread(&_vertices[0], _vertices.size(), 1, pmdFp);
```

とやるか。結局はデータの塊になって GPU に流し込まれるんだからここでの「型」はそれほど重要ではないのです。

これで読み込んできたデータを頂点情報として使用しますが、まだ「インデックス」データがないため、面を構成できません。このまま無理やり面を構成すれば



このように痛いミクさんになります刺さりそうです

なので、一旦皆さんにはこのデータがミクさんであることを納得していただきたいので、ちょっと書き方を変えます。ひとまず TRIANGLESTRIP にしている部分を POINTLIST に変えてください。

次にデータのサイズとストライドが変わっているため

```
_vbView.SizeInBytes=頂点データ総量;
```

```
_vbView.StrideInBytes=頂点一つ当たりのサイズ;//11バッキングしていないなら38直書きでいいです  
この部分も修正しておいてください。
```

あと、まだテクスチャデータとか乗つけてないので UV を一時的に外しましょう。

```
D3D12_INPUT_ELEMENT_DESC inputLayoutDescs[] =  
{  
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},  
    {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }  
};
```

もちろん頂点シェーダからも外します。

```
BasicVS( float4 pos : POSITION/* ,float2 uv:TEXCOORD*/ )
```

ピクセルシェーダ側も、UV がないのでただ色を返すだけにします。

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

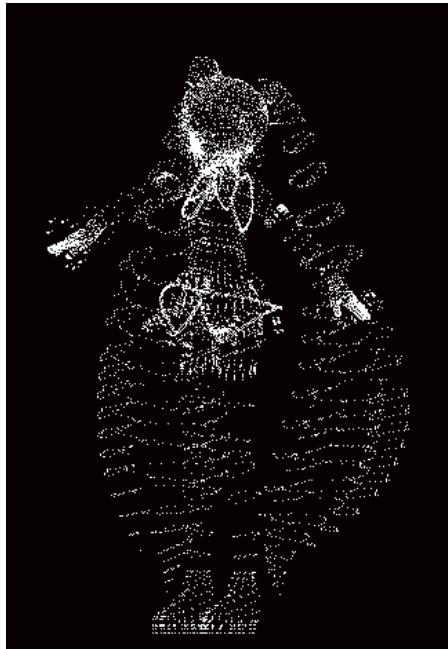
あとは描画部分を修正します。DrawInstancedです。

頂点数が増えちゃったので

_commandList->DrawInstanced(pmdheader.vertexNum, 1, 0, 0);

のように実際に入ってくる頂点数に変更します。

ここまでやって、



やっとこんな感じにミクさんが表示されます

もしミクさんの場所がおかしい場合には、カメラの位置を変更してください。ミクさんは足元が〇なので、どちらかというと上方にあります。視点と注視点の高さは10くらいが適切でしょう。

で、これを見れば、適切に頂点から面を作ればミクさんのシルエットくらいにはなるんじゃないかな?っていう気になってくると思います。さあでは次にミクさんシルエットを作ってみましょう。

ちなみに"pmd"フォーマットならば他のモデルでも表示できますので「俺の嫁」がいる人は是非俺の嫁を表示してください。

あとチョイと嘘教えちゃったかもしれないるので補足しておくと

DrawInstanced(頂点数, インスタンス数, 頂点オフセット, インスタンスオフセット);

で、このインスタンス数ってのを三角形の数って教えてましたが、これ間違えです。

すみません。嘘を教えてしました。

インスタンス数ってのは、実体の数。現段階ではあまり気にしなくていいんですが、画面上に同じ物体をたくさん表示させるときなどに使うと効率的なものです。まあ「自分の分身をいくつ作るか」と思っておいてください。

というわけで、今回はミクさん一人いれば十分なので、インスタンス数は1にしてください。

```
cmdList->DrawInstanced(pmdheader.vertexNum,1,0,0);
```

というわけですね。

インデックスさんデータを使って「面」を表示していこう

ではこの頂点の羅列を「面」にしていきましょう。そのためには一つ一つの頂点を「どうつなげていくか」というデータが必要です。



くそかわいい!

そのデータの事を頂点インデックスと言います。インデックスは頂点配列の配列番号を3つ組み合わせて「面」を表現します。しかしアレだな。あのアニメシリーズはベクトルだのインデックスだのと楽しいアニメですわ。

さて、そのインなんとかさんはどこにいるのかと言うと、頂点データのすぐ後です。

http://blog.goo.ne.jp/torisu_tetosuki/e/7cebae143cb6b9bae38ebbe228c05b

| | |
|----------------------------|---|
| vertex[9035].pos[0] | BEFEF9DC 418C7E5D BF79096C |
| vertex[9035].normal_vec[0] | BE8F31B1 BEB2875B BF650069 |
| vertex[9035].uv[0] | 3F351B71 3F4E6A55 |
| vertex[9035].bone_num[0] | 0005 0005 |
| vertex[9035].bone_weight | 64 |
| vertex[9035].edge_flag | 00 |
| face_vert_count | ここまで頂点データ |
| face_vert_index[0] | 0000AFBF |
| face_vert_index[8] | 0AE8 0AE9 0AEA 0AE8 0AEA 0AEB 0AEB 0AEC |
| face_vert_index[16] | 0AED 0AEB 0AED 0AE8 0AEE 0AEF 0AED 0AEE |
| face_vert_index[24] | 0AED 0AEC 0AF0 0AF1 0AEF 0AF0 0AEF 0AEE |
| face_vert_index[32] | 0AF2 0AF3 0AF1 0AF2 0AF1 0AF0 0AF4 0AF5 |
| face_vert_index[40] | 0AF3 0AF4 0AF3 0AF2 0AF4 0AF6 0AF7 0AF4 |
| | 0AF7 0AF5 0AF8 0AF9 0AF7 0AF8 0AF7 0AF6 |

インデックス数

ここまで頂点データ

つまり頂点データを全て読み終わっていれば、その後にはインデックス数が入っているはずです。では読み込んでみましょう。

unsigned int4バイトです

一応ミクデータなら44991が入ってくるはずです。その後でインデックスを読み込むので

確保しておく必要があります。一つ当たり 2 バイトなので

```
std::vector<unsigned short> indices(indicesNum);
```

で確保しておきましょう。これで 2 バイト × インデックス数を読み込むことができます。

で、まあ皆さんのご予想どおりこいつもバッファにして送らなければなりません。とはいって
クスチャや頂点バッファの時よりかはまだマシです。まず頂点バッファの時と同じようにまず
CreateCommittedResource でバッファ作成

D3D12_INDEX_BUFFER_VIEW を定義し、BufferLocation に GPUVirtualAddress() をセット

そのほかのパラメータは自分で考えて入れてみよう。フォーマットは R16_UINT だ。

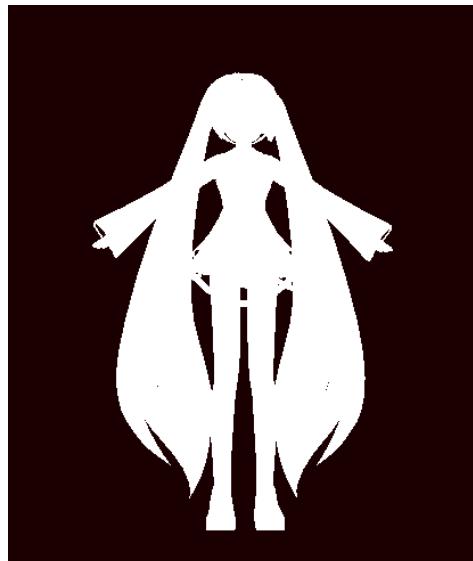
最後にデータのマップやね。これも頂点バッファの時と同じでいい。

次にやるべきことは頂点バッファビューをセットしている部分まで行って、
インデックスもセットする。IASetIndexBuffer でね。

でトポロジーをトライアングルリストにしてくれ。

最後に DrawInstanced を DrawIndexedInstanced に書き換えて、そのうえで頂点数を入れて
いる部分にインデックス数を入れてくれ。

うまくいけば…こうなる



頑張れ!!

立体感つけよう

せっかくここまで来たので立体感をつけよう。



ちょっとおかしな感じですが、理由は後程言います(深度/バッファ設定していないからです)
ひとまず 3D 作ってる実感は大事ですので陰影をつけてみましょう。

ここは実は簡単で、既に「法線情報」ってのがデータに含まれています。Normalってやつですね。面に垂直なベクトルの事です。

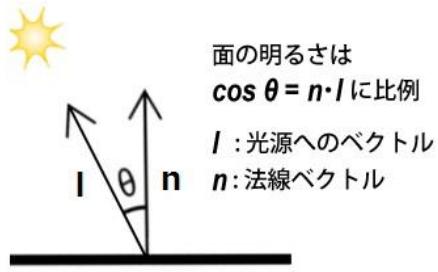
ここで CG の基礎となる有名な理論を紹介しておきますが、Lambert シェーディングとか Lambert の余弦則とか呼ばれる理論です。

<https://ja.wikipedia.org/wiki/%E3%83%A9%E3%83%B3%E3%83%99%E3%83%AB%E3%83%88%E3%81%AE%E4%BD%99%E5%BC%A6%E5%89%87>

なんかクソ難しい事を書かれていますが、簡単に考えるならば、物体表面の輝度(明るさ)は光源へのベクトルと自分の法線ベクトルとのなす角度 θ の $\cos \theta$ に比例する

物体表面の輝度は $\cos \theta$ に比例する

という事です。



いや、これは本来の式をものごつう簡略化して言ってるので、そこはご了承ください。また、最近は Lambert の余弦則ではリアルな質感が追及できないという事で、PBR(物理ベースレンダリング)が採用されています(UE4など)

初心者に PBR やるのは無謀なのでお手軽に立体感を表現できるこの方法でやっていきます。
で、ここで思い出してほしいのが

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

ですね。これを変形すると

$$\frac{\vec{a} \cdot \vec{b}}{|\vec{a}| |\vec{b}|} = \cos \theta$$

a と b が正規化済みならば

$$\hat{a} \cdot \hat{b} = \cos \theta$$

となります。つまり

$$\text{明るさ} = \hat{a} \cdot \hat{b}$$

とすることができます。

ちなみに内積は HLSL では `dot(a,b)` で表します。

さて、前にも書きましたが法線データ自体は入っているのでここからいじるべきはレイアウトに法線 `NORMAL` を追加し、頂点シェーダの引数に法線を入れることです。

つまりレイアウトには

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

を追加し、

頂点シェーダの引数に

```
float3 normal: NORMAL
```

を追加し、

ピクセルシェーダの戻り値を

```
float3 light = normalize(float3(-1,1,-1)); //光源へのベクトル(平行光源)
float brightness = dot(o.normal, light); //内積となります
return float4(brightness, brightness, brightness, 1);
```

とします。

やってみてください。ただ写すのではなく意味を考えながらね。それぞれの関数は各自調べてみてください。

[https://msdn.microsoft.com/ja-jp/library/bb509611\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509611(v=vs.85).aspx)

normalizeは正規化、dotは内積を表しています。

今はただただ内積だけでやってますが、それなりに陰影もついてるでしょ？

さて、回転させるとどこかおかしい事に気づくと思います。これはですね、今は頂点の座標をWVPで変換してますよね？それと同じように、回転にも対応させなければいけないんですよ。

という事で頂点シェーダの部分で法線にも

```
o.normal=mul(wvp,normal)
```

と書けばいいかなーって思うんですが、これは実際には間違えです。何故かと言うと法線に対してカメラ行列もプロジェクション行列も適用してはいけません。

何故かと言うと光は3D座標系の住民です。今それをピクセルシェーダで定義してはいますが、投影後の住民ではありません。つまりそれに対応する「法線」にはカメラ行列もプロジェクション行列も適用してはいけないのです。

でも今のWVPは三つの行列がまとまってしまっている…どうしたものか…と、悩むくらいだったらWとVPは分離しちゃいましょう。

頂点当たりの計算回数は増えてしまいますが、こういう事ならば仕方ありません。

ワールドと、ビュープロジェクションを分割

まあ、別に難しくはないです。送るべきものがちょっと増えちゃうんで構造体化しちゃいましょう。

```
struct BaseMatrices{
    XMATRIX world; //ワールド
    XMATRIX viewproj; //ビュープロジェクション
```

};

という風に二つに分けます。どうせケチったって 256 バイトは送られてしまうので、いいでしょ。

で、送るべきものを増やしてしまいましたので、定数バッファの生成の部分とシェーダ側の修正をする必要があります。

```
cbvResDesc.Width = (sizeof(BaseMatrix) + 0xff)&~0xff;//256 アライメント
```

ここ…

```
cbvDesc.SizeInBytes= (sizeof(BaseMatrix) + 0xff)&~0xff;//256 アライメント
```

ここくらいですね。実は値は変わってないんですが一応ね。

これでデータは送られるはずなので、HLSL(シェーダ)側にも受け取り的な部分を書いておきます。定数バッファのあの構造体みたいなやつあったでしょ？

その行列もワールドとビュープロジェクションの二つを作ってください。

```
cbuffer mat:register(b0) {  
    float4x4 world;  
    float4x4 viewproj;  
}
```

あとは頂点シェーダの時に mul するだけ。

```
pos = mul(mul(viewproj,world), pos); // float2(-1, 1) + pos.xy / float2(480, -270);  
o.svpos = pos;  
o.pos = pos;  
o.normal = mul(world,normal);
```

あとほんまは「回転成分」のみを抽出してノーマルに乗算しなきゃいけないんだけど、それはもう少し後でボーンの実装まで行ってからにします。

ところで陰影をつけてしまうと、おかしな状態が見えてしまうようになるのではないか…？



これを見て、初心者は「法線が裏返ってる!」と思ったらしいです。

確かにそんな感じにも見えますね。ですが、原因は別のところにあります。

メッシュ描画ってのは、本来こういいうものなのです。じゃあ MMD はどうやって対処しているのか？どうやって対処しているのかと言うと『深度バッファ』というものを使っているのです。

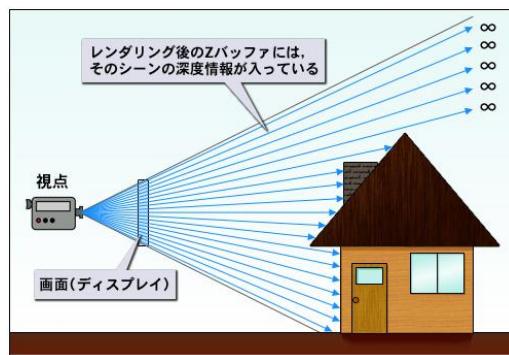
深度バッファの冒険

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。



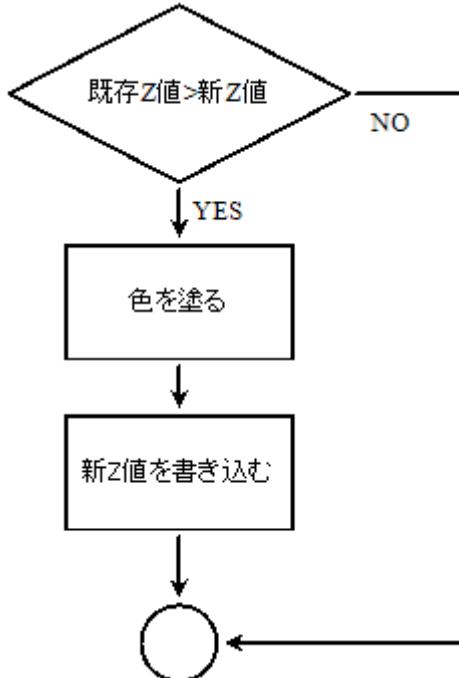
で、この深度バッファがどのように働くのかというと、今からそのピクセルを書こうとすると
きに Z テスト(深度テスト)というのを行います(タイミングとしてはピクセルシェーダ後…
ラスタライザが終わった段階でテストすりゃいいのに…DX12 をクソ難しくする暇あつたら
この辺どうにかしろよ)。

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

`gpusDesc.DepthStencilState.DepthEnable=false;`

を true にすれば OK 的な代物ではありません。実際これを true にすると表示されません。おそらくはバッファがないため比較ができずに常に深度テストが失敗するような結果になつているんでしょう。

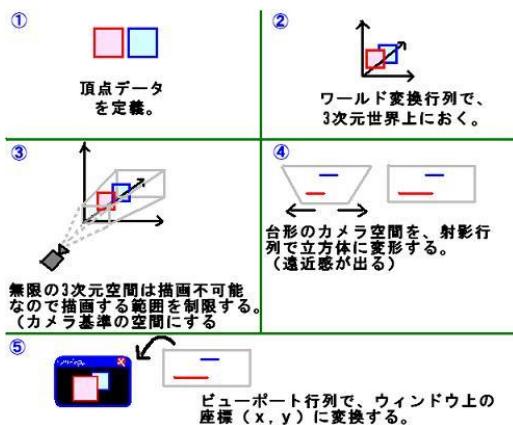
さて、というわけでちょっと不思議なバッファ…デプスStencilバッファを作つていきましょう。え？ 深度バッファじゃないの？ という疑問はごもっともではあるのだが、なんかそういう仕様になっているのだ。ちなみに深度バッファは通常の float と同様に 32bit のバッファである。

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

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

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

以前に、「プロジェク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ば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バッファを作っていく。もちろんいつものように深度バッファを作るだけではなく、色々とやってあげないといけないのだが…

1. 深度バッファ作成
 2. 深度バッファビュー作成(デスクリプター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

まずはテクスチャと同様に `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。

```

depthResDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
depthResDesc.Width = WINDOW_WIDTH;//画面に対して使うバッファなので画面幅
depthResDesc.Height = WINDOW_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;

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());

```

パイプラインステートオブジェクトに深度情報を追加

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

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

次にここでも深度バッファのフォーマットを明示しなければなりませんので、

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

とします。

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

されなければどこかが間違っています。

レンダーターゲットと深度バッファを関連付け



「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気を失い、
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つたりします。
なので、OMSetRenderTarget には深度ステンシルビューを入れる場所が最初から用意されて

います。現在の OMSetRenderTarget をご確認ください。

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

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

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

第 4 引数が

pDepthStencilDescriptor (in, optional)

Type: const **D3D12_CPU_DESCRIPTOR_HANDLE***

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

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

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

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

深度バッファをクリア(毎フレーム)

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

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

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

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

色をつけよう

今ついているのは「陰影」のみで「色」がついていません。色が欲しいですねえ。そもそもこのPMDデータにおいて「色」はどのように設定されているのでしょうか…。3DCGソフトとかやってるのなら知っていると思いますが、色とか見た目とかを設定するのは「マテリアル」と言います。

ではPMDにおけるマテリアルデータとはどこでどうか…。見てみましょう。

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

ここです。「材質データ」材質ってのは英語で言うと「マテリアル」ですから間違いないですね。どのようなデータとして入っているのでしょうか…

```
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(基本色です)
float alpha; // 減衰色の不透明度
float specularity; // スペキュラの強さ
float specular_color[3]; // sr, sg, sb // スペキュラ色
float mirror_color[3]; // mr, mg, mb // 環境色(ambient color)
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、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)
```

きつついな、おい。そしてやっぱりアライメントが絶対絡んでくるよ!!!

作者の底すらない悪意を感じる…。で、ここでポイントになるのは「面頂点数」です。これはなんなんでしょう…。

そうですねえ。試しにバイナリエディタで材質の欄を見てください。

| | |
|---------------------------------|---|
| serial_count | 00000011 |
| serial[0].diffuse_color[0] | 3E083127 3F1EB852 3F36C8B4 |
| serial[0].alpha | 3F800000 |
| serial[0].specularity | 40A00000 |
| serial[0].specular_color[0] | 00000000 00000000 00000000 |
| serial[0].mirror_color[0] | 3D883127 3E9EB852 3EB6C8B4 |
| serial[0].toon_index | 00 |
| serial[0].edge_flag | 01 |
| serial[0].face_vert_count | 00000C06 |
| serial[0].texture_file_name[0] | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| serial[0].texture_file_name[16] | 00 00 00 00 |
| serial[1].diffuse_color[0] | 00000000 3F1096BC 3F1096BC |
| serial[1].alpha | 3F800000 |
| serial[1].specularity | 41200000 |
| serial[1].specular_color[0] | 3E800000 3E800000 3E800000 |
| serial[1].mirror_color[0] | 00000000 3EB4BC6A 3EB4BC6A |
| serial[1].toon_index | 02 |
| serial[1].edge_flag | 01 |
| serial[1].face_vert_count | 00003EF1 |
| serial[1].texture_file_name[0] | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| serial[1].texture_file_name[16] | 00 00 00 00 |
| serial[2].diffuse_color[0] | 3F4CCCCD 3F3645A2 3F1FBE77 |
| serial[2].alpha | 3F800000 |
| serial[2].specularity | 40C00000 |
| serial[2].specular_color[0] | 3E19999A 3E19999A 3E19999A |
| serial[2].mirror_color[0] | 3F000000 3EE3D70A 3EC7AE14 |
| serial[2].toon_index | 01 |
| serial[2].edge_flag | 01 |
| serial[2].face_vert_count | 00002C64 |
| serial[2].texture_file_name[0] | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| serial[2].texture_file_name[16] | 00 00 00 00 |
| serial[3].diffuse_color[0] | 3E23D70A 3E23D70A 3E23D70A |
| serial[3].alpha | 3F800000 |
| serial[3].specularity | 41700000 |
| serial[3].specular_color[0] | 3ECCCCCD 3ECCCCCD 3ECCCCCD |
| serial[3].mirror_color[0] | 3DCCCCCD 3DCCCCCD 3DCCCCCD |
| serial[3].toon_index | 02 |
| serial[3].edge_flag | 01 |
| serial[3].face_vert_count | 000013C8 |
| serial[3].texture_file_name[0] | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |

face_vert_count に注目します。あー、10進数にした方がいいですね。うん、ゼーんぶ足してみてください。

どうなるでしょうか？

…ひとまず合計してみてくださいよ。もちろん計算機を使っていいので。やってみてください。何かが見えてくるはずです。

そうですね。

ゼーんぶ足したらインデックスの数と一致しますよね？つまりインデックス全体をマテリアルごとに割ってるデータ構造なのです。

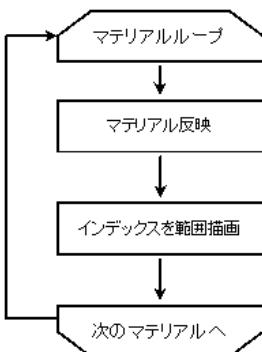
さて、マテリアルデータを利用してポリゴンに色を付けるとすると、どういう手順を用いればいいかなあ…？

色はとりあえず DiffuseColor(3)を使えばいいです。

問題は他の点です。どうやってマテリアルごとに色を付ければいいんでしょうか。ここでヒントになるのがインデックスの数ですね。

そう。インデックスの範囲でマテリアルが違うってことです。でも分割するにはどうしたらいいんでしょうか…？

ちょっと面倒なのですがマテリアルごとにループします。



準備

まずはマテリアルデータを読み込みます。インデックスデータの直後なのでまだ見つけやすいと思います。

ひとまずマテリアル数を読み込んでください。ミクさんなら 17 個くらいのマテリアルがあるはずですので、読み込んだ結果も確認してください。

マテリアル数を読み込んだら、ひとまずマテリアルデータを全て読み込むのですが、

```
struct PmdMaterial {  
    XMFLOAT3 diffuse; // 基本色(拡散反射色)  
    float alpha; // アルファ色
```

```

float specularity;//スペキュラ強さ
XMFLOAT3 specular;//スペキュラ(反射色)
XMFLOAT3 mirror;//アンビエント
unsigned char toonIdx;//トゥーンのインデックス
unsigned char edgeFlag;//輪郭線フラグ
unsigned int vertexCount;//vertexCountだけインデックス数
char textureFilePath[20];//テクスチャがあるときテクスチャパス
};

で、これどうせループ内で GPU にあげるのは別形式になるのでもう pragma pack しておいてください。

```

で、読み込んでください。

これをどう利用するのかと言うと、それぞれの色情報コンスタント/バッファに混ぜて GPU 側に投げます。↑の構造体をそのまま GPU に投げない理由はテクスチャ名とかを GPU にそのまま投げても全く意味がないからです。つまりこのデータは GPU に投げて使う事を想定されたデータではないということです。

まあ、それが悪いのほか面倒なんですが…そして結構ぼく個人はハマってしまったわけなのです。ほぼ徹夜でやって、あきらめて寝て起きたら、まあ、そういうことがと思う。そういう感じの面倒さでした。

こういう時は潔く誰かのサンプルでも見るべきでしたね…。ちなみに MS のサンプルは今回役に立ちませんでした。

DX11 の感覚だけでやるとホンマにキツいんで…。

ともかくそれはおいおい分かると思いますので、まずは GPU とのやりとりができる準備をしていきましょう。ひとまずは既に作成している定数/バッファにくっつける形で。

という事で、投げるための構造体も変わりますのでそちら辺の修正はしてください。

次に GPU(HLSL)側ですが、

```

cbuffer mat:register(b0) {
    float4x4 world;
    float4x4 viewproj;
}

```

```
    float3 diffuse; // 基本色(ディフューズ)  
}
```

↑のように色情報を受け取れるようにしておきます。

次にちょっと面倒なのですが、今、インデックス全部を
`_commandList->DrawIndexedInstanced(indexCount, 1, 0, 0, 0);`
ってやっていると思いますが、これをマテリアルごとに分割します。

例えば先ほどのマテリアルデータから diffuseだけ抽出したいとします。

```
struct Material{  
    XMFLFLOAT3 diffuse;  
};  
std::vector<Material> materials(materialCount);  
的な感じにしちゃいます。
```

```
for(int i=0;i<materials.size();++i){  
    ドロー関数  
}
```

てな感じにします。で、このドロー関数は

```
_commandList->DrawIndexedInstanced(indexCount, 1, 0, 0, 0);  
なのですが、この第3引数がミソなのです。  
https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn903874\(v=vs.85\).aspx  
StartIndexLocationと書いてあります。  
これは…インデックスのデータオフセットだと思ってください。…そしてロード時に取得した  
PMDマテリアルデータを_pmdMaterialsだとすると
```

```
_commandList->DrawIndexedInstanced(_pmdMaterial[i].indexCount, 1, offset, 0, 0);
```

さて…皆さんには一つ課題です。↑のoffsetに当たる部分を計算して、もとの石膏像と同じものが表示される状態にしてみてください。

ヒントはね…offsetはどんどん加算されていきます。
何を加算していくのかと言うと、当然インデックス数です。

よし、それでは実際の色分けだ。

DX11ほど甘くない色分け…



で、実際に色分けをする部分ですが、これが思いのほか難しかったです…というか難しさに紛れてやらかしている凡ミス等があって、原因の特定がしんどかったです。

この手の『初見ライブラリ』におけるバグはその原因が『ライブラリの理解不足』によるものなのか『単なる凡ミス』なのかが見分けにくいのです。割とその時の精神状態に左右されてしまったりします…いや、ホンマ弱音吐くようですが、今回ホンマにしんどかったです…。

でもね、こういうバグに悩んだ時間は無駄ってわけじゃなくて、結局原因を探していく過程でライブラリの理解が深まるわけです。だから二周目になると極端に理解度が上がります。何度か言ってますが、そういうわけなので、余裕がある人は予習か復習かしておくとたぶんすぐにセンサーを越えることができます。

この授業においてただ単にセンサーは1周目の人がなっているにすぎません。

さあ、この長文を読んだら、どれくらい泣きそうになってたか分かると思う。心して実装してほしい。

まずはディフェューズ成分を GPU に投げよう

これは簡単だと思います。

普通に行列と同じようにディフェューズ成分をくっつけます。じゃあ、こんな感じで書けばいいのかな？

```

for (int i = 0; i < materialNum; ++i) {
    cbAddress->diffuse = mat.diffuse;
    描画処理
}

```

こんな感じで…やってみてください。



ん?

うん、そなんだ。すまない。

そういうことなんだ。そううまくいかないのだ。え? GPU のアドレスを直で書き換えるんじゃないの? うん、書き換えるんだけどさ…えーっとね

cbAddress->diffuse = mat.diffuse;

これは GPU のメモリの内容を書き換えるものやねん。
で、

commandList->DrawIndexedInstanced(_pmdMaterial[i].indexCount, 1, **offset**, 0, 0);

これね、ミキフルーンの苗木…じゃなくて、前にも言ったけど、この命令は ExecuteCommand 時に別スレッドで実行されるイメージなのね? で、その Draw 命令の中でピクセルシェーダが走って、先ほどの diffuse を参照するわけだ。そうなるとどうなると思うね?



まあ、推測なんだけどつまるところ Draw が実行されるときには、最後に代入した diffuse しか有效になつていなければいいんだよね(たぶん)
で、最初にやってみたことは

なんで?

```
for(略){  
    cbAddress->diffuse = mat.diffuse;  
    (中略)  
    _commandList->SetDescriptorHeaps(1, &_cbvDescHeap);  
    _commandList->SetGraphicsRootDescriptorTable(1, handled);  
    _commandList->DrawIndexedInstanced(_materials[i].vertexCount, 1, indexOffset, 0, 0);  
    (中略)  
}
```

こんな感じでやってみたが、結果変わらず。理由としては確かにコンスタントバッファの再設定は行われてはいるが、結局のところ GPU メモリ変更のタイミングが違うため、同じ結果になっていると思われる……じゃあ一体どうしたら



また出た

というわけで、まあ困った時の MS サンプルなわけだが……、MS のサンプルは送るべき定数データが 1 種類の時のサンプルしかないので、正直今回の件に関しては不適合なのである。
ここまで来たら不本意だが仕方ない。

先人のコードを見てみよう

<http://zerogram.info/?p=1746#more-1746> (ZeroGram 氏)

http://www.project-asura.com/program/d3d12/d3d12_005.html (AsuraProject 氏)

頭の良い先人たちの PMD 表示コードを見てみよう。一応 2 つのサンプルを持ち出したのは、理由があつて、片方だけだと間違っている可能性が「ぐっと」高くなるからだ。ちなみに 2 サンプルくらいでは間違いの可能性はそれほど低くならないのだが、別にコード丸写しするわけではなく、「動いているコードを見て考察する」ためなので、レリ。のだ。言い訳がましいが、まあ背に腹は代えられぬのだ。

とりあえず二つのプロジェクトを「ConstantBufferView」で検索してみると

↓こういうコード(ZeroGram)と

```
for (auto& cb : cbs){  
    if(cb){  
        pass.apCBuff(ic) = cb->Res.pRes;  
        pDev->CreateConstantBufferView(&cb->View, pass.Heap.GetCPUHandle(pass.idxCBVHeap, ic));  
    }else{  
        pass.apCBuff(ic) = nullptr;  
        pDev->CreateConstantBufferView(nullptr, pass.Heap.GetCPUHandle(pass.idxCBVHeap, ic));  
    }  
    ++ic;  
}
```

↓こういうコードがあった

```
for( size_t i=0; i<m_ModelData.Materials.size(); ++i )  
{  
    m_ModelMB.Update( &m_ModelData.Materials(i), size, offset );  
    m_pDevice->CreateConstantBufferView( &bufferDesc, m_Heap(DESC_HEAP_BUFFER).GetHandleCPU(1 + u32(i)) );  
    bufferDesc.BufferLocation += size;  
    offset += size;  
}
```

とりあえずこのコードから推測できることは…以前から例えている「ジュースとストロー」の例でいうとこういうイメージだろう



よくカップルが使う(という都市伝説がある)あれですよ



独りで使ってはいけないやつですよ

うん、まあ、何が言いたいのかというと、コンスタントバッファ(ジユース)は必要な分(マテリアル数ぶん)だけ用意する必要がもちろんあるんだが、それよりも重要なのはコンスタントバッファビュー(ストロー)を人数分(マテリアル数ぶん)用意する事である。もちろん適切な場所にストローをぶつ刺してだ。

今回の両氏が共通して用いている手法は、一つのコンスタントバッファにすべてのマテリアル情報を入れておき、マテリアル数ぶんのコンスタントバッファビューを作成するわけだ。

じゃあ俺実装

しかし…両氏はどこ情報でこの手法を思いついたのだろうか…多分彼らもどこかのサンプルではあるんだろうけど…ああ、技術者としての技能の差を感じずにはいられない。
そこでコンスタントバッファビュー作成の部分をこう書いてみた。

```
for (int i = 0; i < materialNum; ++i) {  
  
    D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc = {};  
  
    cbvDesc.BufferLocation = bufferLocation;  
  
    bufferLocation+=1個当たりのサイズ(アライメント済み);  
  
    cbvDesc.SizeInBytes = 1個あたりのバッファサイズ;  
  
    heapstart.ptr+= dev->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);  
  
    dev->CreateConstantBufferView(&cbvDesc, heapstart);  
  
}
```

しかしクラッシュ。そしてこのクラッシュがたちが悪い…。クラッシュ位置が毎回変わる(クラッシュしないこともあった)。ということはスレッドセーフではない部分があるという事が…などと考えてしまい…これのせいで月曜日は徹夜したが解決できなかったのだが、実は凡ミスだったのだ。

相手が難しい奴の場合、難しさに紛れてアホなりグが紛れ込む…私は深刻なりグだと思い込み、悩んで徹夜しても治らないのでDirect11にしたくなつたが



というわけで、まあ徹夜しても分からん以上どうせ凡ミスだとあたりを付けたら

//定数デスクリプタヒープの作成

```
D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc = {};
```

```

cbvHeapDesc.NumDescriptors = 1;
cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;//シェーダから見えますように
cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;//コンスタントバッファです
result = dev->CreateDescriptorHeap(&cbvHeapDesc, IID_PPV_ARGS(&_cbvDescHeap));//1つの

```

もうね、これを見た瞬間



こんな気持ちになりました

マテリアル数が1つのビューがいるんだから、ここが1でいいわけねーだろ!!というわけで、これをマテリアル数に変更。フラッシュしなくなりました。

あとはガンガン行こうぜってな話なわけです。

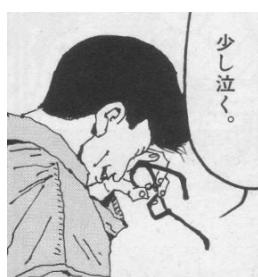
描画部分を

```

_commandList->SetDescriptorHeaps(1, &_cbvDescHeap);
auto handled = _cbvDescHeap->GetGPUDescriptorHandleForHeapStart();
for(中略){
    (中略)
    handled.ptr += GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
    _commandList->SetGraphicsRootDescriptorTable(1, handled);
    _commandList->DrawIndexedInstanced(_materials[i].vertexCount, 1, indexOffset, 0, 0);
    (中略)
}

```

こんな感じにすればいいわけです。俺が寝ずに2日間必死で悩んだ部分が説明するとこの程度になってしまふのがちょっと悲しい。



一コマもたねーじゃねーか!!

というわけで残るはミクさんの目玉部分。

テクスチャを読み込んでモデルに張り付けてを表示しよう

さあいよいよミクさん表示まであと少しです。現在の所



このようになっております。怖いです。

目の部分に関してはテクスチャで



こういうビットマップがあるわけです

eye2.bmp というやつです。

これに関しては……PMD 同じフォルダに入れておきますがロードするには注意が必要です。当然と言えば当然なのですがパスは PMD からの相対パスです。

つまり全てプロジェクトフォルダ直下に入れているのならいいのですが、もし PMD モデルを PMD フォルダなどに入れている場合は注意が必要です。指定ファイル名の手前に PMD のあるフォルダの名前を付加してあげないといけません。

std::string が使えるのならば

```
//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  
//場合によってはそれがゴチャゴチャである可能性もある。  
//ともかく末尾の\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npos(-1→0xffffffff)を返すため  
int pathIndex1=path.rfind('\\');  
int pathIndex2 = path.rfind('\\');  
int pathIndex = max(pathIndex1,pathIndex2);  
std::string folderPath = path.substr(0,pathIndex);  
folderPath += "/"; //最後はセパレータが消えるため(↑の行を pathIndex+1 にしても可)
```

これでフォルダがとてこれるので、この結果とファイル名を連結します。

で、ちょっともう今は色々な理由から直指定します。既に青葉ちゃんをロードしている部分を eye2.bmp にしましょう。

```
FILE* fp = nullptr;  
fp=fopen("miku/eye2.bmp", "rb");  
(中略)
```

で、ロードおよび GPU にデータは投げれてるので後は準備を整えるだけです。以前に書いた UV を復活させます。

レイアウト

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

シェーダ

```
Out BasicVS( float4 pos : POSITION, float3 normal:NORMAL, float2 uv:TEXCOORD)
```

とし、ひとまずはエラーが出ず、クラッシュしないことをご確認ください。またシェーダの中のテクスチャを取り出して絵に変えていくところを見てください。

で、ひとまず目にテクスチャを張るのですが、テクスチャを張るときと、そうじゃないときでちよつと場合分けする必要があります。何故ならばテクスチャを貼っていなければ部分の UV は恐らくすべて(0,0)だから(0,1)だからになっているため出力した結果…



なんだこれウルトラ怪獣紹介じゃねーか
つまり



の左上がまっくろであるために全体的に黒になつたのだ。けして呪いではない。という事で先ほども説明させていただきましたが場合分けをする必要があります。場合分けをする条件が何かといふと「テクスチャデータがあるかなしか」である。

では、例えばブール型を用いて場合分けをすることを考えてみよう。こちらからは定数バッファを通して「テクスチャ存在フラグ」を GPU 側に投げ、GPU 側はこれを見て処理を切り替えるというわけです。

テクスチャがあるかなしか判定するのは簡単で、PMD のデータの「テクスチャファイル名」の先頭が「¥0」であるかどうかを判定すればいいだけです。

```
_materials[i].existTexture = (_pmdMaterials[i].texturePath[0] != '¥0');
```

これを GPU 側に投げて判定すればいいだけですね。

```
float3 color = existTex ? tex.Sample(smp, data.uv).rgb : diffuse;  
return float4(color*brightness,1);
```

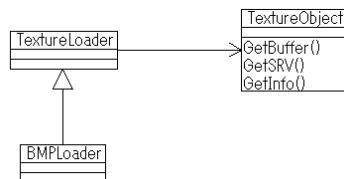
何をやっているかは…分かりますよね？ここまでできれば「かわいいミクさんが表示されますよ。



さて、現在の所テクスチャが目玉一つなので、それほど問題にはなっていないですが、PMDの中にはテクスチャをたくさん持つものがあります。

それを考慮すると現在のテクスチャロード ⇒ シェーダリソースビューの作成までの流れはまとめてしまった方が良いと考えます。

日々のクラス設計的な話だが、覚悟はよいかな？まあ被害は最小限に留めるつもりなんだが

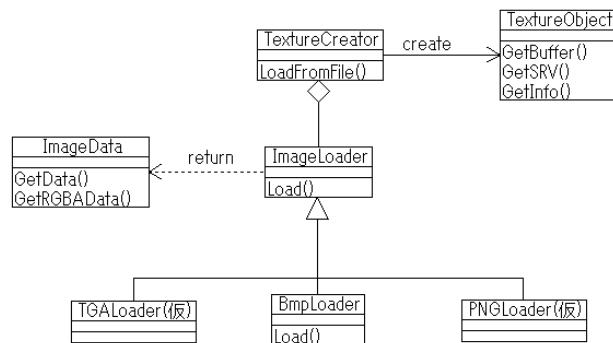


こんな感じ。今回は BMP から読み込んだので BMP ローダを作ってるがたぶん TGA だの PNG だのは必須なので、拡張可能にしている。

ちなみに TextureObject 内にはバッファとビューを持たせて、あとついでに、画像の幅と高さを取得できるようにします。

クラス設計

あ、チョットだけクラス設計変更…一晩寝たら考えが変わりました。



PNGとかTGAはあくまでも仮です

ImageLoader(<-インターフェースクラス)そのものはデータをロードしてメモリ上に持ってくるのみ。そこからバッファとか作るのが TextureCreator という体(てい)にしました。

ちなみに TextureCreator は

```
class TextureObject;
class ImageLoader;
```

```

///テクスチャ生成器
class TextureCreator
{
private:
    //マップを使用しているのは、拡張子から適切なローダを選択し
    //また、ファイルパスとテクスチャオブジェクトをペアにすることにより多重ロードを避
    //けるため
    std::map<std::string, ImageLoader*> _loaders; //拡張子とローダのペア
    std::map<std::string, std::weak_ptr<TextureObject>> _textureobjects; //ファイルパ
    //スとテクスチャオブジェクトのペア
public:
    TextureCreator();
    ~TextureCreator();
    //ファイルをロードしてその結果生成されたテクスチャオブジェクトを返す
    //呼び出し側はファイルの種別については考慮しなくてよい
    //ただ、生成されたテクスチャオブジェクトが返るのみ
  
```

```
///@param filepath ファイルパス
///@retval notnullptr そのファイルを元に作られたテクスチャオブジェクト
///@retval nullptr ファイルが見つからないかエラーが発生した
std::shared_ptr<TextureObject> LoadFromFile(const char* filepath);
};
```

こんな感じにしています。まあ、一つの参考としてみてください。
このクラスの人なら大体どういう風にしようかってのは分かるよね?…てな感じで色々と考えてたらもう全体的にクラス設計すべきやなと言う風に考えた。
DirectX12 初期化周り(デバイス、スワップチェイン、DXGI など、あとコマンド周りも)はシングルトンクラスを持って行ってー。で、テクスチャ周りは TextureObject の中にに入れちゃって。コードのデータは TextureLoader でロードして、中間形式は ImageData とかそんな感じ。

まあ、時間あげるから、しばらく設計を自分で考えて、コードを整理してみて。自分の頭で考えて整理することで見えてくることもあるから。クラス設計のそれなりの基礎は前期で教えるはずだからちょっとトライしてみましょ。

いきなりコードを書き始めるより、いったん図に書いてみたほうがいいと思います。たぶんコード書き始めると、クラスの図そのものを書き換えたくなっちゃう。

コーディング ⇔ クラス設計

を行ったり来たりすることになると思うが、それが力になります。知識だけではまだ力ではないです。練習だと思って頑張りましょう。

ちなみにテクスチャロード→使える状態について部分は DirectX11(2010 June 時代)であれば [https://msdn.microsoft.com/en-us/library/windows/desktop/ff476283\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ff476283(v=vs.85).aspx) D3DX11CreateShaderResourceViewFromFile なんていう関数一発でやってくれてたので、クラス設計とかがムズ化良ければ、ひとまず関数一つにまとめてしまうのもありかもね。

そしてね~?↑のやつのドキュメントを見て気になる記述がひとつ…

Note Instead of using this function, we recommend that you use these:

- [DirectXTK library \(runtime\)](#), **CreateXXXTextureFromFile** (where XXX is DDS or WIC)

- [DirectXTex](#) library (tools), **LoadFromXXXFile** (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then **CreateShaderResourceView**

ん?今…

いや、この記述は DirectX11 用のライブラリ(ヘルバーコード)の記述なんですね…。ちょっと DirectXTex を見てみましょう。

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

( walbourn Updated for latest d3dx12.h (10.0.16299.15))

なるほど、DX12 にも対応している…と、だが気になるのは WDK バージョンが 16299 であることだ。

現在授業で想定しているのは 14393。そのまま落としてきたところで使えない可能性は高い。



まだだ、こらえるんだ

検証するまでもう少し待て。勿論、コードを見るのは OK。もし自分のプロジェクトに組み込むなら自己責任でお願いします。

ちなみに D3Dx12.h が 15063 に対応したのが 2017 年 4 月のこと。コードを取得するのならば、それ以前のものを取得するのがいいだろう。

では次に DirectXTK

<https://github.com/Microsoft/DirectXTK12/wiki/DirectXTK>

なるほど。すでに DirectX12 用のものがあるようだ。

ちなみにこいつも最新版は 16299 である。

で、とりあえずこれを書いている今は日曜の…いや月曜の午前 2 時半。そして風邪をひいています。

ということで、これ以上深入りすると授業に差し支えるので、一言。

どちらにせよ

CreateShaderResourceViewFromFile に当たるものはありません。

ただ、LoadFromFile と、CreateShaderResourceView はあるようです。

ちなみに DirectXTex 側には、それにあたるものがないようです。ただ、ヒントにはなるでしょう。おそらくかつての ShaderResourceView のやり方とかなり変わっちゃったので、その辺の対応ができないんでしょう。

僕の予想だと根本から作り直さないと恐らく、以前のようには使えるようにならないからです。

恐らく使える部分があるとすれば「ファイルから読み込む部分」のみです。ホントにホラー映画とかサスペンス映画の懐中電灯並みに使えないわあ…。

あと ZeroGram 氏によれば

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

2016年4月の記事の時点で「便利な関数や DirectXTex や DirectXTK も用意されています」とのことなので、このライブラリ自体が新しいリリージョンにしか対応していない可能性があります。

まあ今のうちに苦労しといて VS2017 導入後に楽をしつつゲームガンガン作っていこうぜってなところですね。個人で作る分には VS2015 を使ってた方がいいでしょう。

とりあえずそれのフォーマットロードの部分だけ(DirectX12 に関わってない部分)引っこ抜いて採用するといいんじゃないかな。

で、結局のところ TextureObject が内部にテクスチャ/バッファとデスクリプターヒープを内包するようにしている。

つまりこうだ。

```
//テクスチャオブジェクト
class TextureObject
{
    friend TextureCreator;
    ID3D12Resource* _buffer;
```

```

ID3D12DescriptorHeap* _descHeap;
public:
    TextureObject();
    ~TextureObject();
    ///バッファ情報を返す
    ID3D12Resource* GetBuffer();
    ///ヒープ情報を返す
    ID3D12DescriptorHeap* GetDescriptorHeap();
};

```

としておき、TextureCreator から、データを流し込めるようにしておく。ここまではいいかな？

じゃあ PMD ファイルの情報を元にテクスチャもロードできるようにしてみよう

PMD ファイル情報からテクスチャをロード

では以前に書いたやり方で、PMD ファイルパスとテクスチャ名をドッキングする関数を作つてみる。

```

std::string
GetRelativeTexturePathFromPmdPath(const std::string& path,const char* texturename) {
    int pathIndex1 = path.rfind('/');
    int pathIndex2 = path.rfind('¥');
    int pathIndex = max(pathIndex1, pathIndex2);
    std::string texturepath = path.substr(0, pathIndex);
    texturepath += "/"; //最後はセパレータが消えるため(↑の行をpathIndex+1にしても可)
    texturepath += texturename;
    return texturepath;
}

```

こんなのは作つておけば

```

for (int i = 0; i < _pmdMaterials.size(); ++i) {
    _materials[i].diffuse = _pmdMaterials[i].diffuse;
    _materials[i].indexCount = _pmdMaterials[i].indexCount;
    if (_pmdMaterials[i].texturePath[0] == '¥') continue;
    std::string path = GetRelativeTexturePathFromPmdPath(pmdfilepath, _pmdMaterials[i].texturePath);
    _materials[i].texture = textureCreator.LoadFromFile(path.c_str());
}

```

このような形で必要なテクスチャをロードすることができます。なお、_materials(i) の texture の型は TextureObject で中にバッファとヒープが入っています。

じゃあテクスチャを回してみよう

という事で、マテリアルごとにテクスチャも変更してみたい。一応今、テクスチャオブジェクトが紐づいているので

```
for (auto& mat:_materials) {  
  
    *cbufTemp = *cbuffer;  
  
    cbufTemp->diffuse=mat.diffuse;  
  
    cbufTemp->existTexture=(mat.texture!=nullptr);  
  
    cbufTemp= (CBuffer*) ((char*)cbufTemp + ((sizeof(CBuffer) + 0xff)&~0xff));  
  
    if (mat.texture != nullptr) {  
  
        ID3D12DescriptorHeap* texDescHeap[] = { mat.texture->GetDescriptorHeap() };  
  
        _commandList->SetDescriptorHeaps(1, texDescHeap);  
  
        _commandList->SetGraphicsRootDescriptorTable(0, texDescHeap[0]->GetGPUDescriptorHandleForHeapStart());  
  
    }  
  
    _commandList->SetGraphicsRootDescriptorTable(1, handle );  
  
    handle.ptr+=descSize;  
  
    _commandList->DrawIndexedInstanced(mat.indexCount, 1, indexOffset, 0, 0);  
  
    indexOffset+=mat.indexCount;  
}
```

これが“できるかどうかですよね…。

で、ちょっと嫌な予感がするのでこれでOKと思うのはちょっと待っておいてください。ちょっと先に進んでから考えましょう。

ちなみに僕のマシンでここまで事を表示しようとすると



こうなります

怖いです。



流石に見てられないで当初の予定通り

ダメでした。エラーは置きませんでしたが、何も表示されませんでした。



あー!!!めんどくせー!!!

というわけで結局定数/ドッファと同じようなことをやらなければならぬようですね。

では、ひとつ手っ取り早いやり方を考えてみましょう。

とりあえずドッファはそのままにして、デスクリプターヒープのみ共通にしましょう。ということで、TextureCreator がデスクリプタの大元を持っておき、TextureObject の中にはドッファとデスクリプターヒープ番号(インデックス)かヒープハンドルを持たせるようにしましょう。

と、言った感じにクラス分けしてればこういう仕様変更したいときに簡単でしょ?あと、↑の「ひとまずの解決策」ですが、がちやがちやとプログラミングしてるとときは設計に悩むばかりで思いつかなかったんですが、ボクシングの練習で5ラウンドミット打って死にそうになつた後に思いつきました。頭を真っ白にしてしまうのも時には必要です(ちなみに僕の練習時間は22~23時…家に帰ってコーディングに起こすほど)の体力は残ってませんでした)

あ、これを書いている時点で、このやり方でうまくいくかどうかは検証しておりません。ちょっと大急ぎでやります。

…とか思ってたら、授業用のノートパソコンだと動いたやつた…でもやっぱり教師用の PC だと画面に何も表示されない…(•ω•)。ちょっと PC によって動いたり動かなかつたりすると気持ち悪いので両方動くように検証しておきます。

うーん。ちょっとうまく行っちゃつたものは仕方ないのでこのままいきましょう。ただ、後からこちらの検証が終わつたらどのPCでも動くようにそっちに移行していきましょう。

さて、検証中なのが、デスクリプターヒープを一本化してやってみたがこうなった



まだ怖い

緑色になってしまった。一体どういうことなのだろう。正直そろそろ分からなくなってきた。

うーん。こうなってくるとやっぱり
Descriptor…DescriptorHeap…DescriptorTable…RootSignature
の正確な理解が必要になってくるのかもしれません。

今一度

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

を見てみます。みんなも見ておきましょう。

…えーと、まだ正確に理解していないんですが、注目したのは
『DescriptorHeap 内に CBV, SRV, UAV は混在可能』
という所です。

こういう事を敢えて書いているという事でもう一度自分のコードを見てみました。

```
///テクスチャバッファヒープセット
ID3D12DescriptorHeap* texDescHeap[] = { textureCreator.GetDescriptorHeap() };
_commandList->SetDescriptorHeaps(1, texDescHeap);
(中略)
```

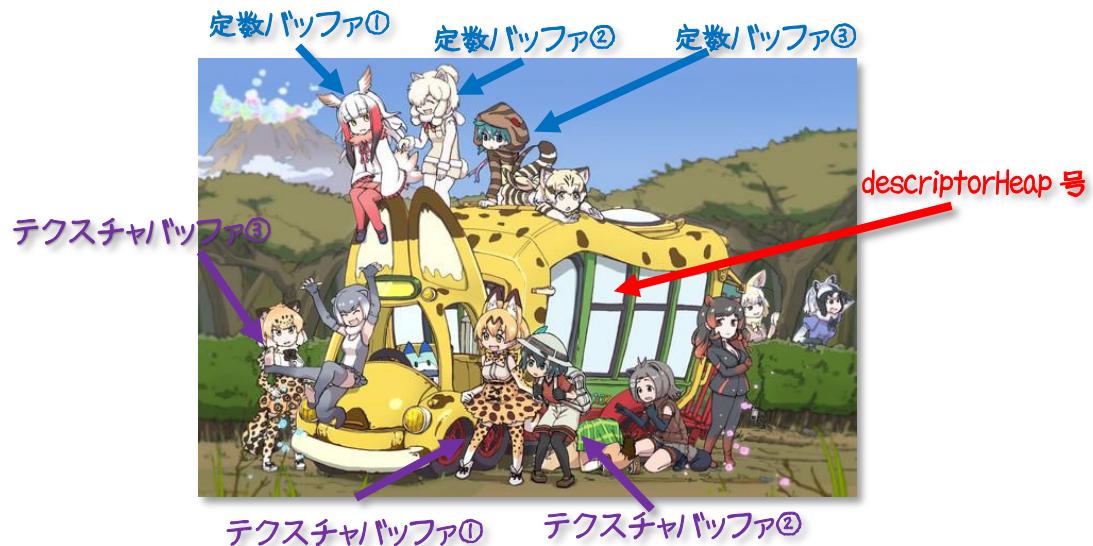
```

///定数バッファ的なヒープセット
_commandList->SetDescriptorHeaps(1, &_cbvDescHeap); //←あっ！
(中略)
for (auto& mat:_materials) {
    if (mat.texture != nullptr) {
        ID3D12DescriptorHeap* texDescHeap[] = { textureCreator.GetDescriptorHeap() };

        _commandList->SetDescriptorHeaps(1, texDescHeap); //あっ…あああ…
        _commandList->SetGraphicsRootDescriptorTable(0, texHandle);
        texHandle.ptr += descSize;
    }
}
(中略)
_commandList->SetGraphicsRootDescriptorTable(1, handle );
(中略)
}

```

うん。良く分かった。逆に言うとテクスチャのデスクリプターヒープと定数バッファのデスクリプターヒープを DirectX12 は区別してないってことやね。ちなみに↑のコードに書いてある SetDescriptorHeaps の第一引数は「インデックス」ではなく「デスクリプターヒープの数」ですから、おそらく同じところに塗りつぶさると考えられます。そう考えると確かに表示されなくなるわな…なんで NotePC ではうまくいくんだろう…
で、ともかく要は



こんなイメージでいいんじゃないかなと

まあでもそこまでしなくてもですね。あることに気づいたらなんですよねえ……さっきの所を見ると。

つまり、デスクリプターヒープのセットが切り替わるという事は、今までの不具合は結局コン

スタンプ/バッファとテクスチャ/バッファを持つてデスクリプタヒープの切り替えが適切に行われていなければ原因があるわけで、

```
for (auto& mat:_materials) {
    if (mat.texture != nullptr) {
        //テクスチャのデスクリプタヒープのセット
        ID3D12DescriptorHeap* texDescHeap[] = { textureCreator.GetDescriptorHeap() };
        _commandList->SetDescriptorHeaps(1, texDescHeap);
        _commandList->SetGraphicsRootDescriptorTable(0, mat.texture->GPUDescriptorHandle());
    }

    *cbufTemp = *cbuffer;
    cbufTemp->diffuse=mat.diffuse;
    cbufTemp->existTexture=(mat.texture!=nullptr);
    cbufTemp= (CBuffer*)((char*)cbufTemp + ((sizeof(CBuffer) + 0xff)&~0xff));
    ///定数バッファ的なヒープセット
    _commandList->SetDescriptorHeaps(1, &cbvDescHeap); //これを忘れていた
    _commandList->SetGraphicsRootDescriptorTable(1, handle );
}
```

とやると、僕のマシンでも表示されます。というか、前に説明したやり方でうまくいく方がおかしい…。ともかくここまでやれば靈夢さんもこの通りである。



良かった…本当に良かった

ただ処理効率の事を考えるとおそらくまとめてしまった方がいいのでしょうか。それはもう少し後で考えましょう。そしてさっさと次の段階へ行きましょう。

地獄入 souha

ボーン(骨地獄)



という事でボーン(bone)の話です。

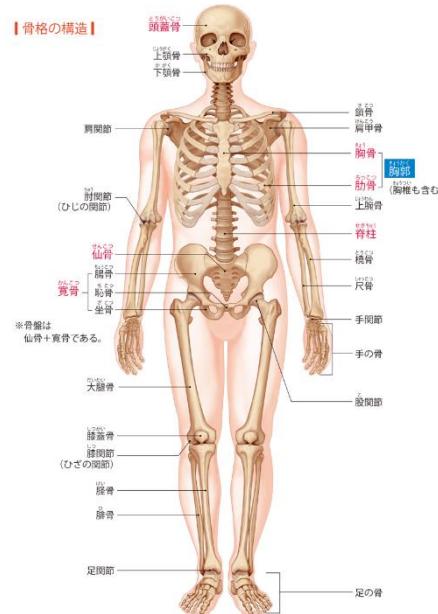
ここからは地獄です。おそらく骨しか残らない人も多いのではないかなどと思います。でも骨は捨ってやるから安心しろ。



という人は無理しなくてもいいです

ボーンって何？

ボーンってのは、骨やねん。ただ君らの体に 206 個くらいあるあの骨とだいたいイメージとしては同じ。



こんな風に皮膚と筋肉の下にあるものというイメージとしては同じ。でも実際にはどちらか

と言うと棒人間の「棒」ってイメージがっているだろう。



実際のMMDにおけるボーンとは



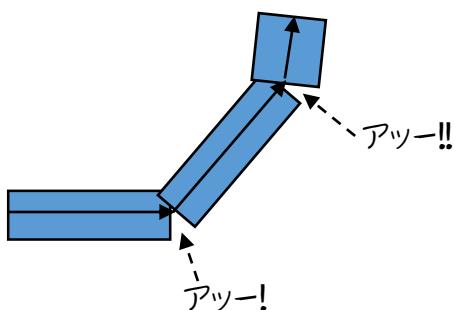
ああ～くっそかわいいんじゃ～

表面に○▷がたくさんあるやろ？これがMMDにおけるボーンなんじゃ。Blenderでいうとアーマチュアってやつかな？

で、動かしてみれば分かることですが、ボーンを動かせばモデルの頂点がそれに沿うように移動します。これによってポージングが可能になります。

スキニング(スキンメッシュアニメーション)とは？

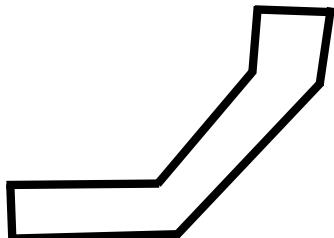
ポージングをするためには結局のところ、ボーンに頂点がくっついていかなければならぬわけだけど、きれいに「関節らしく」動かすためには1つの頂点に関わるボーンが一つでは足りないのである。なぜならば…



スキニングしていないと上の例のように重なってしまう部分や切れ目が出てしまう部分が出てきます。

昔のバーチャファイター(2まではスキニングしていない)ならいざ知らず、最近のゲームのモデルにおいてメッシュがめり込んだり切れたりするのはよろしくありません(とはいえモバ)

イルなどはハーチャ2方式をとっている場合もあります。スキニングは処理が重いので…)
というわけで、特に関節近辺は1頂点が複数のボーンの影響を受けるように修正し、



このように各頂点がうまい具合に一致するようにすれば、ある程度自然な感じでポージングできるようになるわけです。

今回はこの「スキンメッシュアニメーション」が後半戦のメインディッシュになります。

ボーン情報

まずはボーンデータについて見てきましょう。と、その前に今一度頂点データのおさらいをしましょう。

http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b1be2fb10b7838dfcbb0d010389707

中を見てください。頂点情報の中にボーン情報があると思います。既に書きましたが、ボーンは頂点を動かすためのモノなので、頂点側に「どのボーンに関係するのか」情報が入っています。

さて、今お話しした理由により頂点が複数のボーンの影響を受けることがあることが分かつたと思います。

頂点情報のUVの下にボーン番号が二つある。先ほど言ったように頂点は複数のボーンの影響を受けるようになっています。で、ここに2つあるという事は、PMDにおいては最大2つのボーンの影響を受ける(逆に言うと2つのボーンの影響しか受けない)←PMXではここが2つとは限らない…)

で、その後にボーン影響度が(0~100)で設定できるようになっています。これはその頂点が影響度によってどちらのボーンにどのくらい動かされるかがわかるわけです。

つまり、ある頂点 $V(x, y, z)$ があり、それがボーン A とボーン B に影響を受けているとして、それぞれ影響度が $p, (1-p)$ と設定されているとします(影響度は足して1になるように)。

そうすると新しい頂点 $V'(x', y', z')$ は

$$V'(x', y', z') = pA * V(x, y, z) + (1 - p)B * V(x, y, z)$$

と表すことができるわけです。

大体概要はわかりましたか? この辺からどうしても数学的記述が増えていくので、ホント覚悟しましょう。

さて、実際の PMD に於ける影響度はちょっと変わっていて、
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 型なのに注意ね？さらに 1.0f に当たる部分が 100 になっているのにも注意…こんなところで情報量を減らしてもあまり意味が無いんですけどねえ…。
これが各頂点に設定されているってのを覚えておきましょう。

…まあ、そろは言っても、ボーンとスキニングをいっぺんにやると死にます。
いや、切っても切れない関係なんんですけど、ほんマいっぺんにやらん方が良いです。

さて、そういうことで、ボーン…今はボーンのことだけを考えましょう。
で、ボーンが絡んでくると基本モデル以外の場合また大変なので、ひとまずまたモデルはミクさんに戻しておきましょう。

ボーン情報は、マテリアル情報の次に入っています。まずはボーン数を取得しましょう。
ここで注意すべきことがあります。
ボーン数は…2バイトなんですよ。
まあ良いです。読み込んでください。おそらくミクさんなら 122 くらいだと思います。次にボーン情報そのものを読み込みましょう。

http://blog.goo.ne.jp/torisu_tetosuki/e/b384b3f52d0adbca1c46fd315a9b17d0
うへん。39 バイトか…。あえて文句を言わせてもらうなら、この「ボーンの種類」と「IK ボーン番号」は最後においたほうが良かったんじゃないかな…

仕方がない…。

ひとまず

//ボーン情報

```
struct BoneInfo{
    char bone_name[20]; // ボーン名
    WORD parent_bone_index; // 親ボーン番号(ない場合は0xFFFF)
    WORD tail_pos_bone_index; // tail 位置のボーン番号
    BYTE bone_type; // ボーンの種類
    WORD ik_parent_bone_index; // IKボーン番号(影響IKボーン。ない場合は0)
    XMFLOAT3 bone_head_pos; // x, y, z // ボーンのヘッドの位置
};
```

これをロードしましょうか。

```
for (auto& b : bones){
    fread(b.bone_name, sizeof(b.bone_name), 1, fp);
    fread(&b.parent_bone_index, sizeof(b.parent_bone_index), 1, fp);
```

```
    fread(&b.tail_pos_bone_index, sizeof(b.tail_pos_bone_index), 1, fp);
    fread(&b.bone_type, sizeof(b.bone_type), 1, fp);
    fread(&b.ik_parent_bone_index, sizeof(b.ik_parent_bone_index), 1, fp);
    fread(&b.bone_head_pos, sizeof(b.bone_head_pos), 1, fp);
}

こんな感じで。
```

で、大体取得できたらこの情報を「表示」していきましょう。

ボーン情報の「表示」

ボーン情報は、数値で見てもわけわからりやしません。やっぱりエディタみたいに目で見える必要があると思います。

ボーンの中で可視化できるのは「座標」情報です。

座標情報はどこに入っているのでしょうか？

```
char bone_name[20];
WORD parent_bone_index;
WORD tail_pos_bone_index;
BYTE bone_type;
WORD ik_parent_bone_index;
XMFLOAT3 bone_head_pos;
```

どこでしょう？

懸命な皆さまなら、お気づきかとは存じますが、最後の bone_head_pos そして… tail_pos_bone_index です。

「え？ tail_pos_bone_index って WORD(unsigned short)なのに座標情報なの？」

って思ったひとは、まだまだプログラマとしては甘いですね。

プログラマは…特にゲームプログラマは探偵としての素養も大事なのです。推測することが大事です。

index という名前に注目しましょう。あー、確かに、面としての index って感覚があると罷かもしけないなあ。

これは配列の番号なのです。配列の「インデックス」とか言ったりするでしょ？ そういう感覚を持ってください。

で、そのインデックス…そして配列ってのは今読み込んだばかりの「ボーン」の配列…つまり、 bones(bones(i).tailpos_bone_index).bone_head_pos
が、お尻ポジションになります。

なのでもしボーンのベクトルを計算するのならば

```
boneVec(i)= bones(bones(i).tailpos_bone_index).bone_head_pos- bones(i).bone_head_pos;
```

なんていう風になります。

ボーンを可視化するには、このヘッド→テールをラインリストとして表示します。

じゃあ…やろうか

データとしてはインテックスデータを作っても良いんですけど…面倒なので頂点データのみでラインリストを構築します。ですので読み込み用のボーン構造体と、PMDMesh 内のボーン構造体は区別して作りましょう。

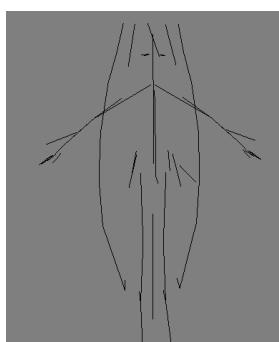
つまりロード用としては

```
//ボーン情報
struct BoneInfo{
    char bone_name[20]; // ボーン名
    WORD parent_bone_index; // 親ボーン番号(ない場合は0xFFFF)
    WORD tail_pos_bone_index; // 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
    BYTE bone_type; // ボーンの種類
    WORD ik_parent_bone_index; // IKボーン番号(影響IKボーン。ない場合は0)
    XMFLOAT3 bone_head_pos; // x, y, z // ボーンのヘッドの位置
};
```

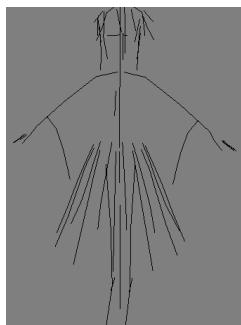
とします。そして表示したい情報はとりあえず座標だけなのでひとまず

```
struct Bone{
    XMFLOAT3 headpos;
    XMFLOAT3 tailpos;
};
```

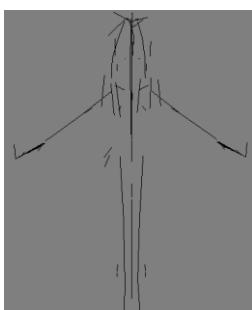
とでも定義する。で、ここに値を入れて、頂点バッファ作って、ボーン用レイアウトと、頂点シェーダ作って、うまいこと表示するとこうなります。



ミクさん



霊夢さん



我那覇さん

まあ、ボーン情報だけ見ても大体誰か何となく分かるものです。我那覇さんだけ、指の先がおかしいですが、これはおそらく「奇妙なボーン」が入っているためでしょう。

手順を書いておきます。

- ①PMDLoader 時にファイルからボーン情報をロード
- ②ロード後に頂点データに変換(BoneInfo 型→Bone 型)
(この時に、テール番号が〇のものは一旦省いておきましょう)
- ③②で作ったボーン配列を使って BoneVertexBuffer を作りましょう
(ストライドは sizeof(XMFL0AT3) でいいでしょう…Bone ではないぞ)
- ④ボーン用に頂点シェーダを作ります。引数を POSITION だけにすればいいです。
- ⑤④の頂点シェーダを元に頂点シェーダオブジェクトとボーン用レイアウトを作ります。
(レイアウトも POSITION のみでいい)
- ⑥ループ前に頂点シェーダ、頂点バッファ、レイアウトを入れ替えればボーンが可視化されるはずです。

ひとまずはこれだけのヒントからやってみましょう。

どうですか？できましたか？注意点を上げるならば、ボーン情報の頂点には色がついていため、ピクセルシェーダ側は
return float4(0,0,0,1);
とでもしてあげましょう。

ああ…でもまだ頂点シェーダ、ピクセルシェーダ、レイアウト動的に切り替えるのをやってなかつたね。

そつからやな…DX11の時のように簡単に切り替えられるかなあ。
まあ、やり方としてはシェーダ作つといて、パイプラインステートオブジェクトを複数作つといて、実行時に切り替えるというやり方が簡単かなあと思います。

ボーン用シェーダ

ボーン用のシェーダを作りましょう。

とりあえず今のシェーダにアペンドする形で、ボーン用シェーダを作ります。あくまでもデバップ用のモノなので凝る必要はありません。

```
//ボーン用頂点シェーダ
float4 BoneVS(float4 pos : POSITION):SV_POSITION
{
    pos = mul(mul(viewproj, world), pos);
    return pos;
}
```

```
//ボーンピクセルシェーダ
float4 BonePS(float4 pos:SV_POSITION) :SV_Target
{
    return float4(0,0,0,1);
}
```

この程度で。

で、見れば分かると思いますが、頂点シェーダの入力が一つになっているのでレイアウトも新しいのを作る必要がありますね

```
{ "POSITION",0,DXGI_FORMAT_R32G32B32_FLOAT,0,D3D12_APPEND_ALIGNED_ELEMENT,D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA,0 },
```

これオンリーなレイアウトを作つておきます。で、標準のと同じように頂点シェーダピクセルシェーダをロードします。

ちなみに、お勧めしておきますが、恐らく複数のシェーダを読むようになると同じようなコードが増えてしまつますので、頂点シェーダロードとピクセルシェーダロードは関数化してお

いた方がいいでしょう。

ともかくボーン用のシェーダもロードして、bonevs とか boneps とかそういう変数名にしておいてください。

そこまでできたら、通常のパイプラインステートオブジェクトの生成後にでも、このボーン用のやつも生成しておきます。で、すでに通常パイプラインステートは生成後なので

```
gpsDesc.VS = CD3DX12_SHADER_BYTECODE(bonevs);
gpsDesc.PS = CD3DX12_SHADER_BYTECODE(boneps);
gpsDesc.InputLayout.NumElements = sizeof(boneLayoutDescs) / sizeof(D3D12_INPUT_ELEMENT_DESC);
gpsDesc.InputLayout.pInputElementDescs = boneLayoutDescs;
で
result = dev->CreateGraphicsPipelineState(&gpsDesc, IID_PPV_ARGS(&_bonePSO));
当たり前だけど、result の確認は忘れないように。
```

で、これでボーン用のパイプラインステートオブジェクトができたわけだ。あとは事前に作っておいたボーン頂点情報を頂点バッファにセット。トポロジーもすり替えておきます。

```
_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_LINELIST);
_commandList->IASetVertexBuffers(0, 1, &_boneVBV);
```



すり替えておいたのさ!

もちろん、今回はインデックスを使用していないので

```
_commandList->DrawIndexedInstanced
```

ではなく

```
_commandList->DrawInstanced
```

を使用します。あとアロケータリセット時のパイプラインステートオブジェクトのすり替えも忘れないように

```
result = _commandList->Reset(_commandAllocator, _bonePSO);
```

ボーンの回転

それでは、ボーンを回転させてみせましょう。

回転なんんですけど、これは数学の時間に口を酸っぱくして言ってるように、原点に平行移動して、回転して、元の座標に平行移動します。

というわけで、左肘を回転させてみましょう。

左肘は「左ひじ」という名前で登録されていますので、とってきます。このため `map→index` 配列を予め用意しておきましょう。

そして

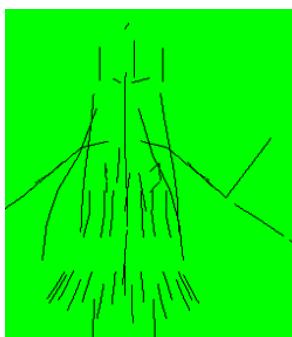
①「左ひじ」を検索しインデックスを得る

②「左ひじ」のテール位置を、ヘッド位置を中心に回転させる

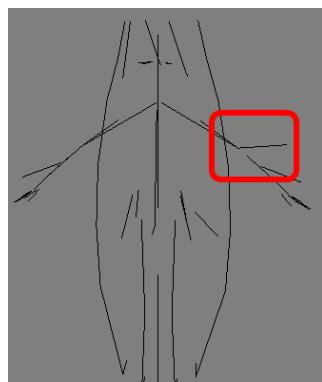
③頂点再セット

このようにするためにボーン頂点／ドッファを変更可能にしておきましょう。

で、ループ前に左ひじを捻じ曲げてみましょう。今は CPU 側でいじってみるのです。



90°ならこうなりますが…ね。45°だとこんな感じ。



さて、やり方を書いておこう。

PMDMesh 側に「名前で検索できる」インターフェイスを作ります。

このため、`std::map` を使用します。

```
std::map<std::string, int> _boneMap;
```

キーは文字列型。値は `int` 型です。

それで、読み込み後に、ボーン情報を作る時についでにこれも作ります。

```
mesh->_boneMap[b.bone_name] = idx;
```

これをボーンロードループ内で行ってあげます。

そしたらこんな感じのデータ(マップ)ができます。

| | less |
|----------------|-----------|
| ▷ [comparator] | allocator |
| ▷ [allocator] | |
| ▷ ["センター"] | 0 |
| ▷ ["右つま先 I K"] | 86 |
| ▷ ["右ひさ"] | 69 |
| ▷ ["右ひし"] | 49 |
| ▷ ["右肩"] | 47 |
| ▷ ["右手首"] | 50 |
| ▷ ["右小指 1 "] | 63 |
| ▷ ["右小指 2 "] | 64 |
| ▷ ["右小指 3 "] | 65 |
| ▷ ["右親指 1 "] | 52 |
| ▷ ["右親指 2 "] | 53 |
| ▷ ["右人指 1 "] | 54 |
| ▷ ["右人指 2 "] | 55 |

こんな風になってればだいたいオッケー

これで名前からインデックスを検索することができます。何の準備かと言うと、アニメーションデータである VMD は名前で動かすボーンを登録しているからです。ともかく名前からインデックスを得る仕組みを作ります。

インデックスがわかれば、ボーンのヘッドとテールの座標も分かりますね？

```
int elbowIdx = mesh->BoneMap()["左ひじ"];
```

例えば、左ひじのインデックスはこんな感じで取ってれます。じゃあこの左ひじをどうやって扱うか？

で、ここからちょっと特殊に感じるかもしれません、XMFLOAT3 の情報を XMVECTOR 型に変換します。理由は「行列」とのやり取りを行うためです。

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.loading.xmlloadfloat3\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.loading.xmlloadfloat3(v=vs.85).aspx)

という関数を使います。Load って名前が変換に似つかわしくないので、ロードとストア(レジスタだのアセンブリだのやつてれば出てくる用語)の対応を取っているためです。(アセンブリで load 命令ってのと store 命令ってのがある)

```
XMVECTOR offsetVec = XMLoadFloat3(&mesh->Bones()(elbowIdx).headpos);
```

```
XMVECTOR tailPos = XMLoadFloat3(&mesh->Bones()(elbowIdx).tailpos);
```

このような感じで、それぞれの XMFLOAT3 を XMVECTOR に変換します。

なお、XMVECTOR てのは

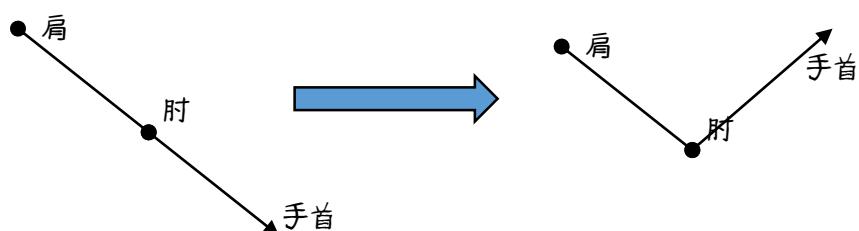
[https://msdn.microsoft.com/ja-jp/library/ee420742\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee420742(v=vs.85).aspx)

と書いてますが、よくわからないので、定義ヘジャンプすると、行列みたいな扱いであること

がわかります。要はただ単に乗算やらなんやらできるように行列の形をとっており、関数を介するすることで XMVECTOR と XMFLOAT3 とのやりとりができますよってこと。XMFLOAT3 と XMMATRIX が直接やり取りできれば楽なのにねえ…。
面倒ですが、仕方ありません。仕様です。

というわけでひとまず「左ひじから手首まで」の「前腕ベクトル」を作成し、それを回転させてまた左ひじにくっつけるということをやってみます。

headpos はそのままで、tailpos を「headpos 中心に回転」させます。あ、別に今回の場合は「中心に戻して～回転させて～元の座標に戻す」なんてことは必要なくて、普通に回転でオッケーです。最初から「肘中心回転」ってわかってますから。これが面倒になるのはもう少し後です…ご安心ください。



さて、現在の状況を見てみればわかるように、腕のボーンが折れ曲がっています。腕のボーンが折れ曲がるということはどういう事がというと…



ボーンを文字通り「骨」として、スキンを文字通り皮膚とするとですよ？ どういう痛ましい状況になるかはお分かりですね？ メッシュへの適用法は後で言いますんで待ってください。



あれ…予想外…昔やった時は、



こんな感じだったんですけど…まあどっちにしても痛ましいですね。

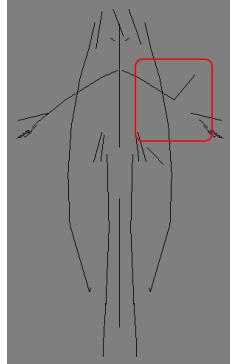
参考コード

//左ひじまげる

```
unsigned short idx = _boneMap("左ひじ");
//idxは左ひじインデックスである
//左ひじボーンを左ひじを中心に90° 回転させてみよう
XMFLOAT3& headpos = _bones[idx].headpos;//前腕ベクトルを作つとく
XMVECTOR lowerarm = XMVectorSubtract(_bones[idx].tailpos, headpos); //ベクトル型に変換
XMMATRIX elbowMat = XMMatrixRotationZ(XM_PIDIV2); //90° 回転行列
elbowVec = XMVector3Transform(elbowVec, elbowMat); //行列をベクトル型に適用

XMStoreFloat3(&lowerarm, elbowVec); //ベクトル型をfloat3に変換
_bones[idx].tailpos = headpos + lowerarm; //ヘッドに足す
```

ボーンだけの表示だと、こう

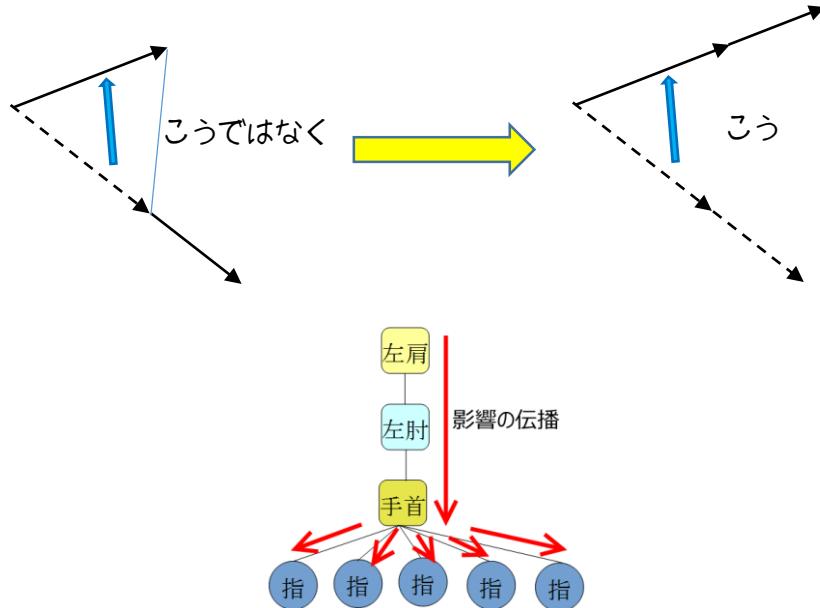


肘を曲げたら当然肘から先まで回転しなければならないのに、前腕ボーンだけが曲がっています。これでは腕が千切れても仕方がない。

何故かわかりますか？それはこの構造がまだツリー構造になっていないからです。

特定のボーンへの座標変換は、そのボーンから先のボーンにまで伝播する必要があるんですね。

肩から指先まで回転を伝播させるにはツリー構造を構築する必要があります。



そういうわけで、きちんとツリー構造を構築していきましょう。ここで「座標変換行列」の特性を思い出してください

『乗算することで合成される』

でしたね？ということは「親指」「人差し指」「中指」「薬指」「小指」は手首の影響を受け、「手首」は左肘の影響を受け、「左肘」は左肩の影響を受けます。

つまり行列の乗算＝座標変換の合成だから

1. 左肩頂点=左肩
2. 左ひじ頂点=左ひじ×左肩
3. 左手首頂点=左手首×左ひじ×左肩
4. 左親指=左親指×左手首×左ひじ×左肩

こういう感じですね。

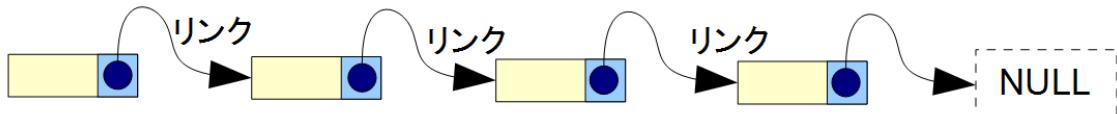
こういう感じなのを効率良くやるには、今回話している「ツリー構造」と「再帰」の理解が必要です。

ツリー構造…

[http://ja.wikipedia.org/wiki/%E6%9C%A8%E6%A7%8B%E9%80%A0_\(%E3%83%87%E3%83%BC%E3%82%BF%E6%A7%8B%E9%80%A0\)](http://ja.wikipedia.org/wiki/%E6%9C%A8%E6%A7%8B%E9%80%A0_(%E3%83%87%E3%83%BC%E3%82%BF%E6%A7%8B%E9%80%A0))

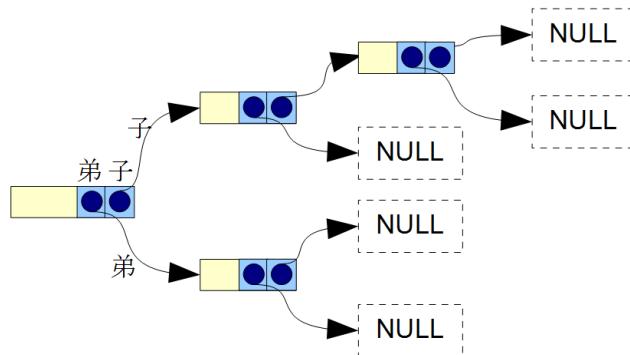
リンクとだいたい同じなんんですけど、下の図のようにリンクは一つのノードから一つのノード

ドヘしか行かないんですけど、



ツリーの場合は、一つのノードから複数のノードがあるわけ

これがC言語の場合ならツリーってのは



こういう風に構築するもんなんだろうけど C++ は vector とかがあるので

```
struct BoneNode{  
    std::vector<unsigned short> children;//子だくさん  
};  
std::vector<BoneNode> _bonenodes;
```

こういうのを用意します。中に保持するのはインデックスデータのみです。結局は整数型の二次元配列みたいになっちゃってますが…BoneNode の children に入るのはボーンのインデックスです。

さらにこれだけでは使い物にならないので、このインデックスから、対象となる行列を指定できるようにする必要があります。このため、先程書いたように、全てのボーンにおける合成行列を代入するものも必要になります。

ツリー反映の準備

なので、先程 head と tail と作りましたが、同じ数だけの matrix を入れられるようにしましょ。

つまり

```
std::vector<XMMATRIX> _bonematrixes;
```

こうします。

そしてボーン情報分のメモリを開けます。

```
_bonematrixes.resize(boneNum);
```

その上でループの中で

```
_bonematrixes[i]=XMMatrixIdentity();
```

こうします。XMMatrixIdentity()は単位行列【何もしない行列】を返します。

あと、余談ですがSTLにはalgorithmってのがあってですね。その中のfillって関数を使うと
std::fill(_bonematrixes.begin(), _bonematrixes.end(), XMMatrixIdentity());

この一行で全てのボーンマトリクスをXMMatrixIdentity()にしちゃえるんですよ。こういう便利なものをまとめた#include<algorithm>ってのがあります。この手のやつ全部覚えるのは大変なので、今は「そういうのもあるのか」くらいに思っておきましょう。



こんなノリで

アルゴリズムに関しては機会があつたらまた解説します。

これでボーン情報には全て単位行列が入っている状態です。前にも言ったようにここに目的の行列を入れます。

何も加工しなければXMMatrixIdentity()のままでいいのですが、ここに「左ひじを曲げる」という操作を追加しようとするならば当然左ひじ以降は全て「左ひじを曲げる」の影響を受けます。

ツリー構造の構築

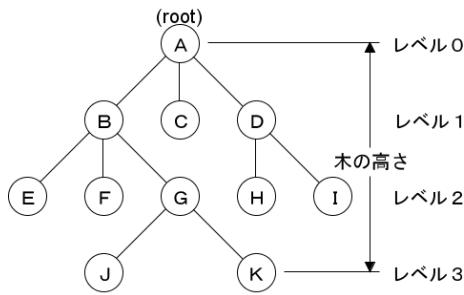
「左ひじ以降」というのはツリーにおける左肘から先のノードのことです。

おっと、まだツリー構造を構築してなかつたですね。本来はこのように書きます

```
struct BoneNode{
```

```
    int index; //インデックス  
    std::vector<BoneNode> child;  
};  
BoneNode _root;
```

ボーンのツリー構造をメッシュの中で宣言します。rootって名前なのは、ツリー構造ってのは、ツリーの最初の部分を「ルート(根っこ)」というからです。



ツリーはいわば「フォルダ構成」みたいなもんだからルートってのは 何だと思ってくれ。ルートディレクトリとか言うでしょ？

さて、こういう構造になるように、ロード時にツリーを構成していきます。今回は簡単にするために、BoneNode をボーン数分作ります。ツリーの構築の仕方としては 30 点ですが、この辺をきっちりやろうとするとけっこう難しいので、一旦こうします。

```
struct BoneNode{
    std::vector<unsigned short> child;//自分の子どもたち
};

std::vector<BoneNode> _bonenodes;
```

ぶっちゃけた話、`std::vector<BoneNode>`の中に `std::vector<BoneNode>` があつたりして、もう混乱の元になりそうですが、今回の BoneNode はただ単に「自分の子供達を管理する」構造体だと思ってください。そしてそれが **全てのボーンに割り当たっている**。つまり全てのボーンがこれを持っている…そう思ってください。

きれいなツリー構造ならば実は `_bonenodes[0]` 以外いらなくなるんですけどね。
(※きれいなツリー構造ってのは、必ずルートから末端までが一つの道で示される…が PMD の構造はモデルによってはちょっと怪しい…ように見える。検証はしていない。)
はい、`_bonenodes` はボーン数分確保してればいいです。その上でロードの際に

```
if (b.parent_bone_index != 0xffff){//親ボーンがある場合
    ret->_bonenodes(b.parent_bone_index).childIndex.push_back(idx); //ボーンツリー構築
}
```

こんな感じで各ボーンノードに子のインデックスを入れていきます。正直な所、これはツリーではありませんが、同じように動作すれば「広義の意味ではツリー」なんですよ（まあちょっと我慢してくれ）。

ともかくこれで子が分かります。ちなみに「初音ミク.pmd」のルートノードの子は 6 個です。

ともかく「左ひじ」の子が分かるので、それら全てを「左ひじ回転」にもとづいて座標変換させてみましょう。

さて、肘の変形にまた戻りましょう。肘中心の変型は

```

//左ひじ変形
int elbowIdx = mesh->BoneMap()("左ひじ");
XMVECTOR offsetVec = XMLoadFloat3(&mesh->Bones()(elbowIdx).headpos);
XMVECTOR tailpos = XMLoadFloat3(&mesh->Bones()(elbowIdx).tailpos);

XMMATRIX bone = XMMatrixTranslationFromVector(-offsetVec);
bone *= XMMatrixRotationZ(XM_PIDIV4); //45° 回転
bone *= XMMatrixTranslationFromVector(offsetVec);
こんな感じで bone って中に肘中心変形が入っています。
実際に変形させる時には

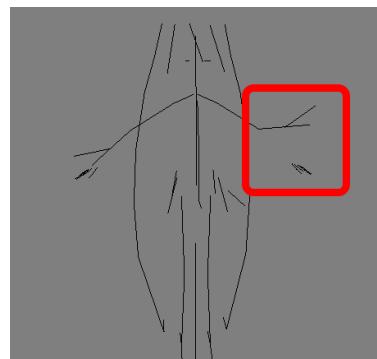
//肘変形
tailpos = XMVector3Transform(tailpos, bone);
XMStoreFloat3(&mesh->Bones()(elbowIdx).tailpos, tailpos);

左ひじの子に伝播するには

//左ひじの子変形
std::vector<PMDMesh::BoneNode>& nodes = mesh->BoneNodes();
for (auto& idx : nodes(elbowIdx).childIndices){
    XMVECTOR hpos = XMLoadFloat3(&mesh->Bones()(idx).headpos);
    XMVECTOR tpos = XMLoadFloat3(&mesh->Bones()(idx).tailpos);
    hpos = XMVector3Transform(hpos, bone);
    tpos = XMVector3Transform(tpos, bone);
    XMStoreFloat3(&mesh->Bones()(idx).headpos, hpos);
    XMStoreFloat3(&mesh->Bones()(idx).tailpos, tpos);
}

```

こんな感じで変形できますが、これだと



手首までは運動しますが、指先が運動しませんね。そりやそうだ。これを指先まで…つまりゴール(指先末端)まで伝播するには「再帰」という方法を使います。

再帰

聞いたことはありますよね?「再帰構造」

再帰ってのは…簡単に言うと

```
void Func(){  
    Func(); // 関数の中で自分自身を呼び出している  
}
```

のように、自分自身を呼び出す関数のことです。しかしこの例だとダメなことは分かりますよね?

そう…無限ループと同じなんです。そこで「再帰構造」には必ず「終了条件」というものが必要です。つまり、ある特定の条件を満たすと「次を呼び出さない」。逆に言うと「条件を満たす」と次を呼び出すことです。

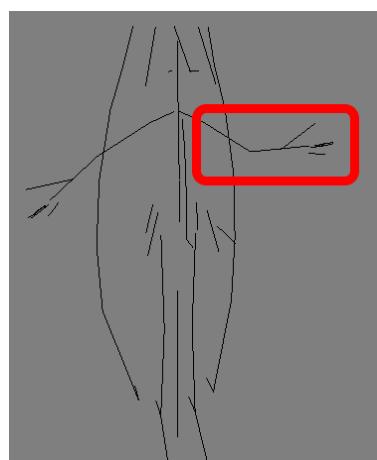
今回の場合次を呼び出す条件は「子がない」ってことですね。

つまり疑似コード的には

```
void Func(node){  
    for(auto& node:node.children){  
        Func(node);  
    }  
}
```

というわけです。この場合、子供がないければ呼び出されることはできませんからね。それが「終了条件」となるわけです。ツリー構造を利用する場合、この「再帰呼び出し」が必須となりますので、覚えておきましょう。

さて、ここで「自分で考えろ」課題です。



自分で考えて、末端までボーンを回転させてください。

できましたか?ここでちょっと待つよ?

なに？一行も書かずに待ってるの？



何度も言ってますが、自分で書かないとかになりませんよ？

再帰は知ってる。やることも分かってる。そこで書き進めようとしている人はダメだよ？



というわけで、自分で考えない潜在的ナマケモノを牽制したところで再帰のコードについてお話ししましょ。再帰のコードを書く際に重要なことは

1. 終了条件(継続条件)

2. 処理

3. 再帰関数呼び出し

です。

中でも終了条件は非常に重要なので脳内に叩き込んでおきましょう。これ不完全だといとも簡単に無限ループするからな

さて、今回の「終了条件(継続条件)」は何でしょうか？それは「末端かどうか」ですね？

そして処理は「肘回転」です

最後に再帰関数を呼び出しますが、これは自分の子すべてに対して呼び出しを行うので for もしくは for_each を使用します。

///再帰関数

///@param bonenodes ボーンノードベクタ

///@param bones ボーン(ヘッドとテール)

///@param index 曲げるべきインデックス

///@param

```
void ApplyRecursiveBones(std::vector<BoneNode>& bonenodes, std::vector<Bone>& bones, unsigned short index,
const XMATRIX& matrix) {
```

//終了条件チェック

```
if (bonenodes.empty() || index == 0) return;
```

//処理

```
//ヘッダを回転
```

```

XMVECTOR headposvec = XMLoadFloat3(&bones(index).headpos);
headposvec = XMVector3Transform(headposvec, matrix);
XMStoreFloat3(&bones(index).headpos, headposvec);

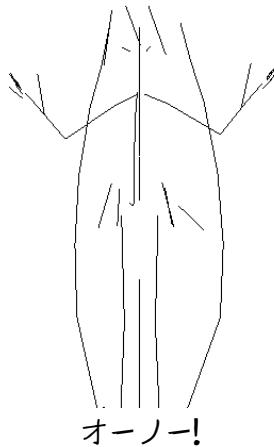
//テールを回転
XMVECTOR tailposvec = XMLoadFloat3(&bones(index).tailpos);
tailposvec = XMVector3Transform(tailposvec, matrix);
XMStoreFloat3(&bones(index).tailpos, tailposvec);

//再帰
for (auto cidx : bonenodes(index).children) { //チルドレンの数がそのまま継続条件になっている
    ApplyRecursiveBones(bonenodes, bones, cidx, matrix); //自分の関数をまた呼び出す
}
}

```

例えばこんな感じですね。

更にこいつらの呼び出し側も関数化すればこの通り…



さて、このままスキニングまで突入したいところですが、まだ弱い。ここまでならまだできる。
そこまで難易度は高くなれ。

では、腕回転→肘回転→手首回転を組み合わせてみてください。どうしたらいいんでしょう
か？

親子構造の中で複数の回転を行う

この辺から流れが変わってきます。

以前に

1. 左肩=左肩

2. 左ひじ=左ひじ×左肩
3. 左手首=左手首×左ひじ×左肩
4. 左親指=左親指×左手首×左ひじ×左肩

こういいうのを説明したかな～って思いますが、覚えてますかね？この辺ほんとにデリケートなので、居眠りして記憶が断片的に途切れると命取りですよ？

最終的に

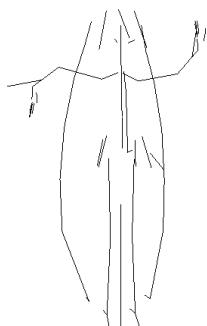
//ボーン曲げ実験

```
BendBoneMatrixes(_boneMap("左腕"), _bonenodes, _bones, _bonematrixes, XMMatrixRotationZ(XM_PIDIV4));
BendBoneMatrixes(_boneMap("左ひじ"), _bonenodes, _bones, _bonematrixes, XMMatrixRotationZ(XM_PIDIV4));
BendBoneMatrixes(_boneMap("左手首"), _bonenodes, _bones, _bonematrixes, XMMatrixRotationZ(XM_PIDIV4));

BendBoneMatrixes(_boneMap("右腕"), _bonenodes, _bones, _bonematrixes, XMMatrixRotationZ(-XM_PIDIV4));
BendBoneMatrixes(_boneMap("右ひじ"), _bonenodes, _bones, _bonematrixes, XMMatrixRotationZ(XM_PIDIV4));
BendBoneMatrixes(_boneMap("右手首"), _bonenodes, _bones, _bonematrixes, XMMatrixRotationZ(XM_PIDIV4));

ApplyBoneByMatrixes(_bonenodes, 0, _bones, _bonematrixes);
```

こんな感じで書いたら



こうなるようにしたい

どうしたらいいんでしょうか？

さあ、果たして…正解や如何に？自分で考えてさっきのヒントに頭を使って考えてみてください。

この先を読み進めずに、自分の頭で考えてください。

ひとまず、ボーンを曲げるという処理はただ曲げるだけという事にします。

```
///ボーン行列曲げ(あくまでも行列を曲げるだけ)

///@param bonemap ボーンマップ

///@param bonenodes ボーンノード

///@param bonematrixes ボーン行列配列

///@param bones ボーン配列(ヘッドとテールのやつ)

///@param boneName 曲げたいボーン名

///@param boneMat 曲げたいボーン行列

void BendBoneMatrixes(unsigned short idx, std::vector<BoneNode>& bonenodes, std::vector<Bone>& bones, std::vector<XMATRIX>& bonematrixes, const XMATRIX& matrix) {

    auto& bone = bones[idx];

    auto& bonenode = bonenodes[idx];

    auto& bonematrix = bonematrixes[idx];

    //行列を回転させる

    XMVECTOR offsetVec = XMLoadFloat3(&bone.headpos);

    bonematrix = XMMatrixTranslationFromVector(-offsetVec);

    bonematrix *= matrix;

    bonematrix *= XMMatrixTranslationFromVector(offsetVec);

}

}


```

で、その曲げた結果を再帰的に反映する関数を作ります。

```
///ボーン行列配列でボーンを曲げてみます(先に曲げたやつのを実際に適用)

///@param bonenodes ボーンノード

///@param index 再帰用(先頭は大抵の場合0)

///@param bones ボーン配列(ヘッドとテールのやつ)

///@param bonematrixes ボーン行列配列

void ApplyBoneByMatrixes(std::vector<BoneNode>& bonenodes, unsigned short index, std::vector<Bone>& bones, std::vector<XMATRIX>& bonematrixes) {

    //終了条件チェック

    if (bonenodes.empty()) return;

    //処理

    auto& bonenode = bonematrixes[index];

    auto& bone = bones[index];

    auto& bonemat = bonematrixes[index];

    //ヘッタを回転

}

}


```

```

XMVECTOR headposvec = XMLoadFloat3(&bone.headpos);

headposvec = XMVector3Transform(headposvec, bonemat);

XMStoreFloat3(&bone.headpos, headposvec);

//テールを回転

XMVECTOR tailposvec = XMLoadFloat3(&bone.tailpos);

tailposvec = XMVector3Transform(tailposvec, bonemat);

XMStoreFloat3(&bone.tailpos, tailposvec);

//再帰

for (auto cidx : bonenodes[index].children) { //チルドレンの数がそのまま継続条件になっている

    auto& cbone = bonematrixes[cidx];

    cbone = cbone*bonemat; //大事なのはここです。順番に注意ね。末端から根元に乘算していきます。何でかは逆にすると分かります

    ApplyBoneByMatrixes(bonenodes, cidx, bones, bonematrixes); //自分の関数をまた呼び出す

}

}

```

はい、非常にちっちゃえコードになりますが、こんな感じです。もちろんもっと効率の良い書き方はありますので、模索してください。

さて、ボーン単品ならここまで、何とかかんとか理解したと思って良いでしょう。理解してなかつたらもう一回やり直す勢いで頑張ってください(大変だと思うけど…頑張ろう)

申し訳ないが、こつから次年度就職の皆さんには全神経を開発に集中しないと間に合いません。本當です。

最後に補足

この辺になってくると、XMMATRIX をパンパン使っていく事になると思います。そうなると色々と意味不明なトラブルに見舞われるようになります。これ辛い。本当につらい。

妙なクラッシュ

これは DirectX11 時代にあったことなのですが、実行時に妙なバグによってクラッシュすることがあります。ぱっと見原因が分からぬはずです。こういう場合は仕様の見落としが関係していることが多いです。

<http://mofo.pns.to/?#2010-07-26>

に書いてある。

「xnamath を利用してたら、実行時にエラーが発生するようになりました。デバッガコンパイルしたプログラムを通常実行した場合に発生するのですが、エラーの理由はアライメントのせいでした。**SSE を利用する場合、変数のアドレスが 16 の倍数の位置にいなければいけない**のですが、new で動的に作ったものだと問題が起ります。対処法は new をオーバーライドして_aligned_malloc に置き換えるしかないのですが、そうなると確実に_aligned_free と対応させないといけないので、他のライブラリと複合的に使う場合の対処に不安が残ります。」

というような現象があります。

これもまあ MSDN に明記されているのですが

[https://msdn.microsoft.com/ja-jp/library/ee418725\(v=vs.85\).aspx#TypeUsageGuidelines](https://msdn.microsoft.com/ja-jp/library/ee418725(v=vs.85).aspx#TypeUsageGuidelines)

に書いてあるんですが

ここにも「16 バイトのアライメントが必要」と書かれています。

で、この手のエラーの特徴として、リビルドして一発目にエラーが発生し、二回目以降はエラーが発生しない…そんな感じの現象だと思います。

まあ Microsoft 社としては「16 バイトアライメントを違反すると何が起きても知りませんよ?」ってスタンスです。今回で言うとこの「クラッシュ」です。

それもあって「値渡し」が推奨されないんですねわかりません。

SIMD とか SSE に関しては後述しますが

「これらの型がローカル変数として使用される場合はこれらの型を自動的にスタック上に正しく配置し、グローバル変数として使用される場合はデータセグメント内に配置します。正しい規則を使用し、これらの型をパラメーターとして関数に渡すこともできます（詳細については、「[呼び出し規則](#)」を参照してください）。」

…まあつまるところ「演算に使用する XMMATRIX の先頭アドレスは 16 の倍数じゃないとダメよ?」ってこと。

よくわからないと思いますが、new はもちろんクラスメンバであるとか、の場合は通常の構造体配置のように配置されるため「4 バイトアライメント」は行われますが「16 バイトアライメント」ではないことがあるため、メンバ変数に配置した時点で規約違反的な使い方になります。

まあ、アライメントに関しては既に定数バッファで面倒なことになっているので何となくわかっているとは思いますが…。

対処法

そういう現象に対する対処法ですが、皆さんはこう思われるでしょう。
『いや、アライメントの配置なんてプログラマが制御できるわけじゃねーんだからコンパイラが勝手に配置しやがったらどうしようもねーじゃん？それとも適切に配置する方法とかあるの？』
なるほどごもっともでございます。

ひとまず落ち着いて XMMATRIX 単品を見るとですね。まあ…こう書いているわけですよ

```
_declspec(alignment(16)) struct XMMATRIX
{
    #ifdef _XM_NO_INTRINSICS_
        union
        {
            XMVECTOR r(4);
            struct
            {
                float _11, _12, _13, _14;
                float _21, _22, _23, _24;
                float _31, _32, _33, _34;
                float _41, _42, _43, _44;
            };
            float m(4)(4);
        };
    #else
        XMVECTOR r(4);
    #endif

    XMMATRIX() XM_CTOR_DEFAULT
    #if defined(_MSC_VER) && _MSC_VER >= 1900
        constexpr XMMATRIX(FXMVECTOR R0, FXMVECTOR R1, FXMVECTOR R2, CXMVECTOR R3) : r{ R0,R1,R2,R3 }
    {}
    #else
        XMMATRIX(FXMVECTOR R0, FXMVECTOR R1, FXMVECTOR R2, CXMVECTOR R3) { r(0) = R0; r(1) = R1;
        r(2) = R2; r(3) = R3; }
    #endif
}
```

```

XMMATRIX(float m00, float m01, float m02, float m03,
         float m10, float m11, float m12, float m13,
         float m20, float m21, float m22, float m23,
         float m30, float m31, float m32, float m33);
explicit XMMATRIX(_In_reads_(16) const float *pArray);

#ifndef _XM_NO_INTRINSICS_
    float      operator() (size_t Row, size_t Column) const { return m(Row)(Column); }
    float&    operator() (size_t Row, size_t Column) { return m(Row)(Column); }
#endif

    XMMATRIX& operator= (const XMMATRIX& M) { r(0) = M.r(0); r(1) = M.r(1); r(2) = M.r(2);
r(3) = M.r(3); return *this; }

    XMMATRIX    operator+ () const { return *this; }
    XMMATRIX    operator- () const;

    XMMATRIX& XM_CALLCONV    operator+= (FXMMATRIX M);
    XMMATRIX& XM_CALLCONV    operator-= (FXMMATRIX M);
    XMMATRIX& XM_CALLCONV    operator*= (FXMMATRIX M);
    XMMATRIX& operator*= (float S);
    XMMATRIX& operator/= (float S);

    XMMATRIX    XM_CALLCONV    operator+ (FXMMATRIX M) const;
    XMMATRIX    XM_CALLCONV    operator- (FXMMATRIX M) const;
    XMMATRIX    XM_CALLCONV    operator* (FXMMATRIX M) const;
    XMMATRIX    operator* (float S) const;
    XMMATRIX    operator/ (float S) const;

    friend XMMATRIX    XM_CALLCONV    operator* (float S, FXMMATRIX M);
};


```

ちょっと勢い余って全部取ってきちゃいましたが、必要なのは最初の1行だけですね。ここに書いてある`__declspec(alignment(16))`という記述はこいつを配置するときは16バイトアライメントにしろよっていうコンパイラに対する指定だったりします。

つまり、普通に宣言すれば勝手に16バイト境界になるのです。じゃあ何故おかしくなるのか

というと、いくつかりパターンがありますが、当たり前ですが、mallocなどで動的に「バイト数指定」で確保した場合がそれにあたりますね。これはただ単にメモリ上に〇〇バイト確保しておけばいいから 16 バイトアライメントの保証はできないのです。

という事で気を付けてほしいのが malloc で確保する時と、あとは構造体の中のメンバとして入れていて new されるパターンね…。この時も 16 バイトアライメントにはなりません。DX11 時代にやらかしたパターンで言うと PMDMesh というクラスの中にボーン行列の配列を持たせていて、

```
pmdmesh = new PMDMesh(ファイル名);
```

みたいな使い方してたわけですよ。そりやああんた 16 バイトアライメントになりませんよってことですね。

というわけで、お気を付けください。

じゃあ構造体に入れて new して使いたい場合はどうしたらいいんでしょう？

非常に面倒ですが、関数内などで Stack を利用することになります。

```
XMMATRIX bone = bonematrixes(index);
XMMATRIX mat = bonematrixes(cidx);
mat=mat*bone;
bonematrixes(idx) = mat;
```

Stack の一時変数に退避させておいて、そっちに計算させたうえでまた戻しとるわけです。まあこういう対処法くらいしかないので…もしくはこいつを内包しているクラス自身を 16 バイトアライメントにするしかないですね。

で、こういう 16 バイトアライメントとか、そういう制約がなんであるのかっていう話ですが、インテルの CPU とかには SIMD 命令なんてあるので、その

SIMD 命令(SSE)とは

定義はここを読んで下さい。

https://ja.wikipedia.org/wiki/Streaming SIMD_Extensions

読んでもよくわかりません。

簡単に言うと、「いつぺんに浮動小数演算をものごつついスピードでやってくれるやつ」です。う~ん。128 ビット…つまり 16 バイト分(float*4)いつぺんに計算します。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

ご覧のような演算が一回で終わります。ていうか実際は…

このレベルで計算されます。

ともかく↓のように 16 バイト分いっぺんに計算する都合上、16 バイトアライメントになっているわけです。

$$\left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right) \left(\begin{array}{c|c|c|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right)$$

ちなみに SSE に関しては

<http://www.officedaytime.com/tips SIMD.html>

に書いてますが、とても使いこなせない。少なくとも僕は嫌です。必要に迫られない限り覚えたくもないです。

その辺を既に組み込まれているのが XMATRIX と XMMatrix 系の計算なのです。ここまで、大雑把な説明ですが、お分かりいただけたでしょうか？

さて、では忘れている人も多いかもしませんが、このボーンをいじる本当の目的を実行に移していきたいかなと思います。

スキーリング

スニーキングではありません。スキーリングです。事前にちょっと説明してた「頂点を重みづけで座標変換する」を行うものです。という事で、表示がまたメッシュに戻ります。

頂点データについて

ボーン ID

当然ながらボーン情報…つまり座標変換行列は頂点の座標変換に使われます。

そしてどのボーンがどの頂点に影響をあたえるのか…というと…思い出してください。

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

ボーンの番号は bone_num と言うのに入っています。2 バイト 2 つ分で表され、それぞれ二つのボーン番号に対応します。

WORD bone_num(2); // ボーン番号 1、番号 2

まあ、大丈夫ですかね？これを GPU 側に投げます。

ボーン ID に悩まされる

そういうことなのでレイアウトには

```
{ "BONENO", 0, DXGI_FORMAT_R16G16_UINT, 0, D3D12_APPEND_ALIGNED_ELEMENT, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
```

と、追加したいところです。ところがこの状態…ちょっと問題ありなんですよ。CPU 側ではなく GPU 側で…

では頂点シェーダ側はどう書きますか？

`uint2 boneid : BONENO`

と追加する？残念…これではうまくいかないんだ。そりやね、 $16+16=32$ ビットしか情報来てないのに $32*2$ ぶんを取ろうとしてもダメだよね？CPU 側 `UINT2` と言うと、2 ビット整数型みたいなイメージがあるんだけど、この定義だと `uint` 二つ分って意味なんだよね。

さあ、うまくいかない…どうしよう。

で、DX11 時代の説明では `uint` でとってきてビットシフトで分割してたりしてたんですが…一つ試したいことがあります。

`min16uint` なるものがあるらしい…。

[https://msdn.microsoft.com/en-us/library/windows/desktop/bb509646\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509646(v=vs.85).aspx)

<https://docs.microsoft.com/ja-jp/windows/uwp/gaming/glsl-to-hlsl-reference>

これが有効なのかどうかを試してみたい。

ちなみに参考までに DX11 の時にやってた手法書いときます。

ビット演算が出てきます。まさかのビット演算…でも大丈夫。HLSL できちんと規定されてるから良いんです。

[https://msdn.microsoft.com/ja-jp/library/bb509631\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509631(v=vs.85).aspx)

つまり 32 ビットで渡しておいて、ビット演算によって上位 16 ビットと下位 16 ビットを分割します。

ここは別にシェーダとかそういう話じゃなくて、皆さんお得意の基本情報の話ですよ？しばらく考えてみてください。

そう…ビットシフトと、&演算を使うんだ。

つまり

`uint boneid : BONENO`

で受け取っておいて

`boneid & 0xffff`

と

`(boneid & 0xffff0000) >> 16`

に分割する。

これで得られた2つのIDがその頂点に影響を与えるボーンIDだと分かるわけ。

…うん、色々な知識が必要なんだね。あーめんどくさい。

もちろんこれをするとならば、32ビット整数として渡す必要があるため、

```
{ "BONENO", 0, DXGI_FORMAT_R32_UINT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },  
としておきます。
```

だったわけなんだけど、こういう面倒な事をやらなくてもいいかも。

では次に影響度だが、こいつは8ビットだし0～100なので、ちょっとイラッとする。

```
{ "WEIGHT", 0, DXGI_FORMAT_R8_UINT, 0, D3D12_APPEND_ALIGNED_ELEMENT,  
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
```

さて、ここまでやって、果たしてmin16uintが有効なのかどうか検証してみる

min16uintの検証

普通にやろうとすると、スキーリング実装するまで分からないので、もうちょっと気楽に検証できないか考えてみる。こういう思考って大事よ？後になればなるほど検証のコストは増えるからね。

まずはシェーダ側の受け取りを

```
Output BasicVS( float4 pos : POSITION, float3 normal : NORMAL, float2 uv:TEXCOORD,  
min16uint2 boneno:BONENO, min16uint weight:WEIGHT)
```

こうやっておいて

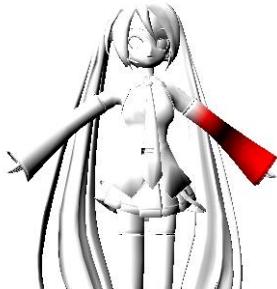
Outputを書き換えて

```
struct Output{  
    float4 svpos:SV_POSITION;  
    float4 pos:POSITION;  
    float3 normal:NORMAL;  
    float2 uv:TEXCOORD;  
    float4 color:COLOR;  
};  
で  
if (boneno(0)==19 || boneno(1)==19) {  
    o.color=float4(1, 0, 0, 1)*(float(weight)/100.0);  
}
```

で、ピクセルシェーダで

```
color = data.color.rgb;
```

ですね。



どうやら大丈夫そうですね。よかつたよかつた。さて…、ここからが大変かもしれません。

ボーン用定数バッファ(ボーン数×行列)を作る

ここでボーン行列を「すべて」GPU 側に投げる必要が出てくるのです。

何故かと言うと

頂点は1度に GPU 側に渡されています。このため、「必要なぶんだけ」ボーン情報を与えるってことは事実上不可能なのです。

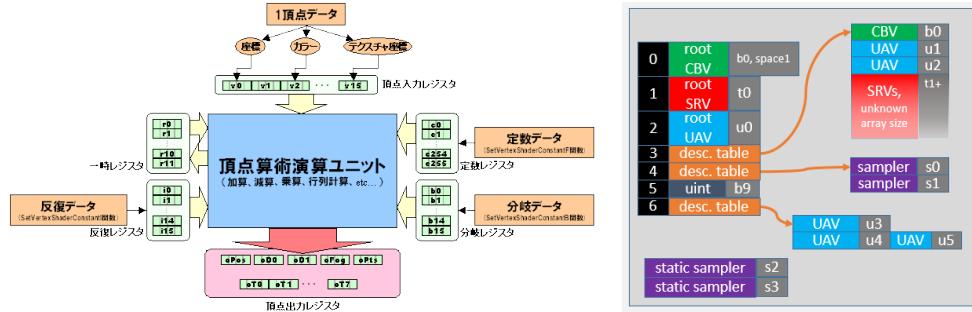
…まあできるのかもしれないけど、今の僕は知らない…と言い続けてはや3年。どうしたらいいんでしょうか?



結局わかんないので全部投げます。あともう一つ難点があつて、シェーダ側では動的な配列が使用できません。このため無駄なようですが、考えられる最大数のボーン行列を投げることになります。ちょっとどうにかしたいところなのですが…。ともかく **256個** 投げましょう。ミクさんだと 122 ですが、靈夢さんだと 204 あります。このため一般的なモデルで 128では足りないと思われるためです。

じゃあまた例のコンスタントバッファに追加で良くね?と思われると思いますが、あのコンスタントバッファはループ内で情報を入れなおしています。それが必要だったので…ただ今回のような場合は毎フレーム 1 回のコピーで済ませたいので新しくコンスタントバッファを作ります。

ちなみにレジスタの概要図がこちらです。



一応定数レジスタはたくさん持てるんですが、なるべくなら増やさないほうがいいとされています(プログラムも面倒になるしね)が、今回の場合は、情報の種類が違いすぎるのと、とにかく多いのとで、分けてしまった方が良いという判断です。

まずはルートシグネチャに、定数を増やすよ～と教えておきます。

//定数バッファ(レジスタb1用)

```
descriptorRange[2].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV;
descriptorRange[2].BaseShaderRegister = 1;
descriptorRange[2].NumDescriptors = 1;
descriptorRange[2].RegisterSpace = 0;
descriptorRange[2].OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
```

ポイントはシェーダーレジスタの部分ですね。

さて実際に定数を増やします。

float(4 バイト)*16*256=16384 バイト…つまり 16KB である。まあ毎フレーム送られる量としては大したサイズじゃないかな。だってテクスチャで 16KB なんて超小さい方でしょ? おもったより大したサイズじゃないです…ただ、これが数キャラいると考えるとつらい…。』

あ、でも恐らくですが、正直に 256 投げる必要はないです。ひとまず必要な分だけぶん投げてみましょう。

実装してみた感想を言うと…またやっちゃんついた感がありますね。いつもの

『困難さと凡ミスの狭間で』

ってやつです。昼の 12 時～19 時くらい潰れてしまいました。木曜日金曜日は頭が疲労しているのでなおさら凡ミスが増えてしまって痛いですね。

ちなみに最初にドはまりしたのが、ヒープを2つ作って、SetDescriptorHeaps に複数乗つけたんですが、何故かうまくいかず、モノごつつい悩んだんです。

例えばこんな感じにしてたわけです。

```
ID3D12DescriptorHeap* descHeaps[] = {cbvHeap, boneCBVHeap};  
_commandList->SetDescriptorHeaps(2, descHeaps);
```

で、これエラーも出ないけど何も表示されなくなりました。つまりダメなわけです。ちなみに行列がきちんと渡せてないのかなと思ってすべて単位行列にしたけど何も表示されず。頭を悩ませていたところにこの記事が…

http://masafumi.cocolog-nifty.com/masafumis_diary/2016/01/id3d12graphicsc.html

「D3D12GraphicsCommandList::SetDescriptorHeaps って's'がつくだけあって複数の ID3D12DescriptorHeap を一括でセットできるけど、その時の注意点として同じタイプの DescriptorHeap は複数セットできない」のね。』



ひどすぎやしませんかね

ホントかよ…公式のどこにも明記されてないんだけどそういういやこのサンプルコード見て も Heaps なのにも関わらず1個しか乗つけてないサンプルばつかだし、ZeroGram 氏のコード 中にも

```
void RenCommandX12::SetDescriptorHeap(IDescHeapPtr heap)  
{  
    if (pCurHeap != heap) {  
        ID3D12DescriptorHeap* ppHeaps[] = { heap.Get() };  
        pCmd->SetDescriptorHeaps(_countof(ppHeaps), ppHeaps);  
        // 複数DescriptorHeapを設定できない  
        // SRVとCRVは1つしかだめ！  
        pCurHeap = heap;  
    }  
}
```

こんなん書かれてるし…もう僕は叫びましたよ。



もうどんだけ罠張りや気が済むんだよ…まあ確かにね、ヒープ配列のつけて「配列中のどのヒープを使うか」なんていう関数がない以上、そりゃあそうんだろうけど…

でもそれはリファレンス中に明記しろよクソったれが!!!



うん、取り乱しましたよ流石に。こちとら慣れない英語読みまくって脳みそ疲れとんのじゃ!!

という所で脳みそが疲労したのでこの後凡ミス(バッファの名前を間違える等)を連発し、5時間が見る見る間になくなってしまいました。ケロが出そう。

ちなみにこの後に実装を解説しますが、レジスタとバッファとデスクリプターテーブルは、先に使用しているコンスタント/バッファと分けるべきですが、デスクリプターヒープに関して

は分けても分けなくてもいいです。SetDescriptorHeaps の回数を減らしたいならまとめるべきですが、そこまでやるならテクスチャともまとめてしまっていいだろうと思います。

実際にポージングしてみよう

正直ここまで痛めつけられたのでこの辺の理解は深まってきた。だから授業に先んじて色々とやっておられる皆さんはそうでない人より理解度が高いと思います。痛い目に遭った方が理解度は上がる。これは間違いない。



とまあこんなのが書いてるからドウンドウン時間がなくなっていくのですが。

手順を書きます

1. ルートシグネチャにボーン用の設定を追加する
2. ボーン行列 256 個/バッファ作る
3. 定数/バッファビューを設定する
4. でもしヒープを分けてるならループ内でボーンデスクリプターヒープをセット
5. ボーンデスクリプターテーブルをセット

てな感じになります。くれぐれも言っておきますが、この後にコード書いていきますが、そのまま口ボットのように写すのはおやめください。「何やってるのか」を考えながら自分の頭で解釈してコードを書いてください。

君の目的は単位を取る事？それとも高い技術を身に着ける事？前者なら早めに身の振り方を考えておいた方がいいと思います。

```
//定数バッファ(レジスタb1用)
```

```
descriptorRange(2).RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_CBV;  
descriptorRange(2).BaseShaderRegister = 1;  
descriptorRange(2).NumDescriptors = 1;  
descriptorRange(2).RegisterSpace = 0;  
descriptorRange(2).OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
```

```
//定数バッファ用(レジスタ0)
```

```

rootparam[2].DescriptorTable.NumDescriptorRanges = 1;
rootparam[2].DescriptorTable.pDescriptorRanges = &descriptorRange(2);
rootparam[2].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
rootparam[2].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;

D3D12_ROOT_SIGNATURE_DESC rsd = {};
rsd.Flags = D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT;
//ここに今から「レンジ」とサンプラーを設定する。
rsd.NumStaticSamplers = 1;
rsd.pStaticSamplers = &samplerDesc;
rsd.NumParameters = 3;//テクスチャと定数バッファの合計数
rsd.pParameters = rootparam;

```

でルートシグネチャを作り一の

```

//ボーン用の定数バッファ作成
ID3D12Resource* _boneCB = nullptr;
ID3D12DescriptorHeap* _boneDescHeap = nullptr;

D3D12_HEAP_PROPERTIES cbvHeapProp2 = {};
D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc2 = {};

cbvHeapProp2.CPUPageProperty = D3D12_CPU_PROPERTY_UNKNOWN;
cbvHeapProp2.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;
cbvHeapProp2.CreationNodeMask = 1;
cbvHeapProp2.VisibleNodeMask = 1;
cbvHeapProp2.Type = D3D12_HEAP_TYPE_UPLOAD;

cbvHeapDesc2.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
cbvHeapDesc2.NumDescriptors = 1;
cbvHeapDesc2.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;

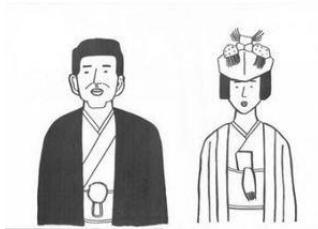
result = dev->CreateCommittedResource(&cbvHeapProp2,
D3D12_HEAP_FLAGS::D3D12_HEAP_FLAG_NONE,
&CD3DX12_RESOURCE_DESC::Buffer((sizeof(XMMATRIX))*_bonematrixes.size() + 0xff)&~0xff),
D3D12_RESOURCE_STATE_GENERIC_READ,
nullptr,

```

```

    IID_PPV_ARGS(&_boneCB));
定数バッファつくりーの
result = dev->CreateDescriptorHeap(&cbvHeapDesc2, IID_PPV_ARGS(&_boneDescHeap));
D3D12_CONSTANT_BUFFER_VIEW_DESC boneCBVDesc = {};
boneCBVDesc.BufferLocation = _boneCB->GetGPUVirtualAddress();
boneCBVDesc.SizeInBytes = (sizeof(XMMATRIX)*_bonematrixes.size() + 0xff)&~0xff;
auto boneH=_cbvDescHeap->GetCPUDescriptorHandleForHeapStart();
boneH.ptr += cHeapHandleOffset;
dev->CreateConstantBufferView(&boneCBVDesc, boneH);//
ヒープとビューを作りーの

```



トツギーノ

じゃなくて、まだあって、ボーン行列コピーの

```

XMMATRIX* boneBuffer = nullptr;
result = _boneCB->Map(0, &cbRange, (void**)&boneBuffer);
std::copy(_bonematrixes.begin(),_bonematrixes.end(), boneBuffer);

```

GPUにボーンバッファ渡しーの

```

//ボーンバッファを渡す
auto boneGH=_cbvDescHeap->GetGPUDescriptorHandleForHeapStart();
boneGH.ptr += cHeapHandleOffset;
_commandList->SetDescriptorHeaps(1, &_cbvDescHeap);
_commandList->SetGraphicsRootDescriptorTable(2, boneGH);
ひょうじーの

```



やつたぜ

リファクタリング(コードをキレイにしましょう)

流石にそろそろまたリファクタリングしたい!(俺が)。

しつこいようですが、「リファクタリング」の際には、バージョン管理ツールを使用しましょう。そして「望みどおりに動いている状態」でコミットしておいてください。いつでも戻せるようにという事と、何処をいじったら悪くなつたかというのを知るためです。

今回リファクタリングしたいのはコンスタント/バッファ周り(ルートシグネチャ→デスクリプターヒープ+バッファ)と PMD ロード周りですね。あとついでに BMP だけでなく他のロードできるようにしたいかなと思います。

PMD ロード周りはともかく、コンスタント/バッファ周りはどう整理しようかなあ。ともかく手始めにあのルートシグネチャのレンジとかパラメータの部分を vector 化するところから始めましょうか。

待てあわてるなこれは vector の罠だ



ま…またまたやられていただきましたアン!

というわけでいきなりハマりましたので、僕の失敗を書いておきます。

ひとまずこういう関数を作りました。

//パラメータやレンジをルートシグネチャのために追加する

///@param rootParams(out) ルートパラメータ配列(ベクタ)

```

///@param descRanges(out) デスクリプタレンジ配列(ベクタ)
///@param visibility シェーダビジュアリティフラグ
///@param type レンジタイプ
///@param regno レジスタ番号
void AddParameterAndRangeForRootSignature(std::vector<D3D12_ROOT_PARAMETER>& rootParams,
                                         std::vector<D3D12_DESCRIPTOR_RANGE>& descRanges,
                                         D3D12_SHADER_VISIBILITY visibility, D3D12_DESCRIPTOR_RANGE_TYPE type, unsigned int regno) {
    D3D12_DESCRIPTOR_RANGE desc_range = {};
    desc_range.RangeType = type;//レンジ種別
    desc_range.BaseShaderRegister = regno;//レジスタ番号
    desc_range.NumDescriptors = 1;
    desc_range.OffsetInDescriptorsFromTableStart = D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND;
    descRanges.push_back(desc_range);

    D3D12_ROOT_PARAMETER root_param = {};
    root_param.DescriptorTable.NumDescriptorRanges = 1;
    root_param.DescriptorTable.pDescriptorRanges = &descRanges.back();//対応するレンジへのポインタ
    root_param.ShaderVisibility = visibility;
    root_param.ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
    rootParams.push_back(root_param);
}

```

まあ普通に悪くないですよね。この関数自体はさ。
でね？

```

AddParameterAndRangeForRootSignature(_rootParams, _descriptorRanges, D3D12_SHADER_VISIBILITY_PIXEL, D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 0);
AddParameterAndRangeForRootSignature(_rootParams, _descriptorRanges, D3D12_SHADER_VISIBILITY_ALL, D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 0);
AddParameterAndRangeForRootSignature(_rootParams, _descriptorRanges, D3D12_SHADER_VISIBILITY_ALL, D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1);

```

こういう風に呼び出すわけですよ。何処もおかしなところはないと思します。
で、これで実行するとルートシグチャ生成に失敗します(正確に言うとシリアルライズの段階で失敗します)

これ、僕はしばらく悩みました…30分くらい。

そしてデバッガを見ました。

| | |
|------------------|--|
| [capacity] | 3 |
| [allocator] | allocator |
| [0] | {ParameterType=D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE (0) DescriptorTable={NumDescriptorRanges=1 ...} ...} |
| ParameterType | D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE (0) |
| DescriptorTable | {NumDescriptorRanges=1 pDescriptorRanges=0x09864940 {RangeType=-572662307 NumDescriptors=3722304989 ...}} |
| Constants | {ShaderRegister=1 RegisterSpace=159795520 Num32BitValues=0} |
| Descriptor | {ShaderRegister=1 RegisterSpace=159795520 } |
| ShaderVisibility | D3D12_SHADER_VISIBILITY_PIXEL (5) |

ん?

何でこうなってるんですか?教えてください!何でもしますから。

(。•ω•)ん?

ちなみに代入したばかりの時はきちんとした値が入っています。関数を抜けたばかりでもきちんとした値が入っていました。

2個目が追加された時に値が壊れていきました…。

謎は全て解けたツツツツツツ!!すべてベクタのせいだSTLの仕様のせいなのだアーッ!!

DirectXにいぢめられ、STLにすらバカにされ…ぼくはもう、生きてるのが嫌になった



まあ、いつものことなんんですけどね

さて、どういうことなのか。もしSTLの中身とかに興味がある人は自分で考えてみましょう。考えるまでもないつすかね?

割とSTL初心者にとっては興味深い話だと思うのでせつかくだから解説します。

今回 vector の push_back メソッドを利用して要素を追加しております。で、この push_back メソッドが曲者なんですよ。

push_back するって事はさ、メモリが 1 個余分に必要になるわけじゃん？ あとベクタのお約束として「連続したメモリ」になっている事が仕様になっています。



例えば↑の図のようにあらかじめ 4 つあるところに 1 つ push_back しようとするとします。この時すでに 5 個分のメモリ領域が使える状況にあるのならば問題ありません。ところが push_back するまでは、メモリは使用していないわけで、メモリは貴重なので、使ってなかつたら即他の奴が持つていきます。



つまり↑みたいにならなければなりませんが、Vector の役割は連続したメモリを確保するため、5 個が確保できる別のアドレスを探して 5 個を確保するわけです。



↑みたいにならなければなりませんが、元のアドレスが変わっちゃうわけ。となると、今回の場合は問題になるのはどこかというと

`root_param.DescriptorTable.pDescriptorRanges = &descRanges.back(); // 対応するレンジへのポインタ`
この部分です。

最初のアドレスを pDescriptorRanges に代入しているわけですが、descRanges の中のアドレスは push_back するたびに変わってしまう可能性があります。もし、今大丈夫であってもそのうちこの↓が発生します。じゃあこれに対処するにはどうしたらいいのか？

対処その①: vector の reserve を使用する。

そう、既に習ってると思いますが、resize ではなく、reserve はあらかじめ使用する領域を確保しておくという命令です。これを事前に必要な分確保していれば push_back のたびにアドレスが変わることはなくなります。

ただ、このやり方の場合、あらかじめパラメータの最大数を知ってないと最大数を越えた瞬間にこの問題が発生しますので、まあ、安心はできないかな…と。

という事で考えられる一つの策としてはさつきと同じように作っていいんだけど
pDescriptorRanges の部分に関しては、ルートシグネチャ作成前に設定しなおすのが良いかなと思いません。

つまり

```
//デスクリプタレンジアドレスの再設定
for (int i = 0; i < _rootParams.size(); ++i) {
    _rootParams[i].DescriptorTable.pDescriptorRanges = &_amp;descriptorRanges(i); //再設定
}
//シグネチャシリアル化
result = D3D12SerializeRootSignature(&rsd,
    D3D_ROOT_SIGNATURE_VERSION_1,
    &signature,
    &error);
```

こういうことだ。

次は定数バッファ(テクスチャも合わせるか?)周り

定数バッファ(やテクスチャ)周り…苦しめられたしね。どうしていいようか…。

ひとまず定数バッファ単品を何とかしようか。定数バッファに関連するものは

- 定数バッファ
- 定数バッファ用デスクリプターヒープ
- 定数バッファ用デスクリプターテーブル
- 定数バッファビュー
- 定数バッファをマップしたポインタ(アドレス)

という所ですね。

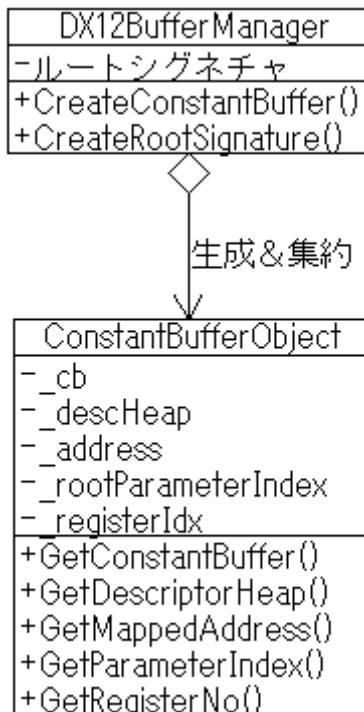
まあ、実はデスクリプターテーブルやバッファビューに関しては「使用者側」からすると、特に必要ない。マップしたポインタも結局は定数バッファから取ってくるんだから、要らないいちやいらない。ただ、これをいちいち必要なときに Map してから取り出そうとすると面倒なので、定数バッファ作った時点でマップしておいてもいいんじゃないかな。

クラス図で書くとこんな感じかな。まだ確定じゃないよ

| ConstantBufferObject |
|-----------------------|
| - _cb |
| - _descHeap |
| - _address |
| + GetConstantBuffer() |
| + GetDescriptorHeap() |
| + GetMappedAddress() |

じゃあ、バッファとかヒープはいつ作んのかって話ですが、

まあ「いつ」ってのは別にいつでもいいんですけどね、どっちかというと、これを誰が生成するのかってところですね。うーん、結局のところ、ルートシグネチャがコンスタントバッファの数とか知つておく必要があるため、ルートシグネチャの持ち主い…ですかねえ。



うんこの「持ち主の名前」がなかなか付けづらいんだよね…どうしようかな。
あまり `Manager` とかつけたくないんだけど…まあ一時的にはいいかな。
ちなみにルートシグネチャを生成した段階でコンスタントバッファを作るの禁止。というか
正確にはルートパラメータとかレンジを増やすの禁止。

ということで `assert` を使用して事前条件を決めておきます。`assert` で事前条件を設定しておくのは一つの手だと思ってください。…言つてもなかなかうまい事キマらんなあ…。クラス設計はやっぱり難しいですよ。でも正直 DX12 についての理解がここまで来ない」とたぶん適切な設計ってできなかつたと思うんだ。

今頃になって ZeroGram 氏とか ProjectASURA 氏の設計がなぜ、ああいったまどろっこしい形になつたのか理解したよ…。

でも理解するまではコード丸写しなんかしても全く意味ないからね。ちょっとしくじると自分で対応できないからさ。まあやせ我慢して、ここまで苦労してきた意味はあったと思うんだ。

ひとまずこんな感じで

```

///定数バッファオブジェクト
///このクラスの中に定数バッファ、デスクリプタヒープなどをまとめておく
///生成された時点でマップされておりそのアドレスは内部に持っている
///このオブジェクトが削除されるときにUnmapされます

class ConstantBufferObject
{
    friend DX12BufferManager;//DX12BufferManagerからは見えるように

private:
    ID3D12Resource* _buffer;//コンスタントバッファ
    ID3D12DescriptorHeap* _descHeap;//デスクリプタヒープ
    char* _address;//マップ後のアドレスを返す
    unsigned int _rootParameterIndex;//パラメータインデックス
    unsigned int _registerIndex;//レジスタ番号

    unsigned int _descriptorIndex;//DX12BufferManagerが持っているデスクリプタ配列へのインデックス
    unsigned int _descriptorNum;//必要なデスクリプタの数

    size_t _bufferSizeOfOne;//ひとつあたりのバッファサイズ
    unsigned int _heapHandleOffset;

public:
    ConstantBufferObject();
    ~ConstantBufferObject();
    ///定数バッファを返します
    ID3D12Resource* GetBuffer();

    ///デスクリプタヒープを返します
    ID3D12DescriptorHeap* GetDescriptorHeap();

    D3D12_GPU_DESCRIPTOR_HANDLE GetGPUHandle();

    ///マップされたアドレスを返します
    char* GetMappedAddress();

    ///レジスタ番号を返します
    unsigned int GetRegisterNo()const;
}

```

```

    ///パラメータインデックスを返します
    unsigned int GetParameterIndex()const;

};

さて、「リファクタリングは少しずつ」が鉄則。まずは「まとめる」ところからやっていきましょう。
もちろんコレ1つを生成する奴も作ります。

class ID3D12RootSignature;
class ID3D12DescriptorHeap;
class ConstantBufferObject;

//DirectX12の定数バッファ周りとかルートシグネチャ周りの制御クラス
class DX12BufferManager
{
private:
    //定数バッファ周り
    std::vector<ConstantBufferObject*> _bufferObjects;
    std::vector<ID3D12DescriptorHeap*> _descriptorHeaps;

    //ルートシグネチャ周り
    ID3D12RootSignature* _rootSignature;
    std::vector<D3D12_ROOT_PARAMETER> _rootParams;
    std::vector<D3D12_DESCRIPTOR_RANGE> _descRanges;

    bool _createdRootSignature;//ルートシグネチャ生成済みか

public:
    DX12BufferManager();
    ~DX12BufferManager();

    ///定数バッファオブジェクトを生成する
    ///@param size 1つ当たりのサイズ
    ///@param num バッファの数
    ///@param preCBO ヒープを共通で使いたい定数バッファオブジェクト(省略可)
    ///@return 定数バッファオブジェクト
    ConstantBufferObject* CreateConstantBufferObject(size_t size, unsigned int num,unsigned int

```

```

regno, D3D12_SHADER_VISIBILITY visibility, const ConstantBufferObject* preCBO=nullptr);
    void CreateDescriptorHeaps();
    void CreateRootSignature();
    unsigned int AddParameterAndRangeForRootSignature(D3D12_SHADER_VISIBILITY visibility,
D3D12_DESCRIPTOR_RANGE_TYPE type, unsigned int regno);
    ID3D12RootSignature* GetRootSignature();
};

こういう感じですね。

```

んで、今回も色々と凡ミスしてハマったりしました。本当に Git が役に立ちました。結構バグるとすぐ動かなくなるのが DirectX12 だからね。何度も言うようだけど必ず動いている状態になるたびにコミットしてね。

だいたい今回のリファクタリングだけで 3 回くらいコミットしています。

| ID | 作成者 | 日付 | メッセージ |
|----------|------|---------------------|---------------------|
| ▲ ローカル履歴 | | | |
| 20dc6a09 | 川野竜一 | 2017/11/17 19:45:38 | ルートシグネチャ移行完了 |
| e0c99b3e | 川野竜一 | 2017/11/17 19:05:16 | リファクタリング第二段階 |
| f33ed962 | 川野竜一 | 2017/11/17 17:57:03 | なんとカリファクタリングが第一段階OK |

「もうリファクタリングする暇ねーよ」って人はそのままやっちゃってください。

コマケー部分

コマケー部分で結構苦労しましたよ。

皆さんご存知のように、非常に設定がややこしいのです。リファクタリングするときに、まあここに関しては色々と、本当に色々と気を使いながらクラス設計する必要があります。

1. 定数/バッファを作る
2. 全部作ったたらデスクリプターヒープを全部作る
3. 最後にルートシグネチャを作る

```

DX12BufferManager::CreateConstantBufferObject(size_t size, unsigned int num, unsigned int
regno, D3D12_SHADER_VISIBILITY visibility, const ConstantBufferObject* preCBO) {
    //ルートシグネチャ生成後は扱えません
    assert(!_createdRootSignature);
}

```

```

auto& dx12=DirectX12::GetInstance();
auto dev = dx12.Device();
ConstantBufferObject* cbo = new ConstantBufferObject();
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;

auto result = dev->CreateCommittedResource(&cbvHeapProp,
D3D12_HEAP_FLAGS::D3D12_HEAP_FLAG_NONE,
&CD3DX12_RESOURCE_DESC::Buffer(num*((size + 0xff)&~0xff)),
D3D12_RESOURCE_STATE_GENERIC_READ,
nullptr,
IID_PPV_ARGS(&cbo->_buffer));

cbo->_descriptorNum = num;
cbo->_bufferSizeOfOne = size;

if (preCBO == nullptr) {
    cbo->_descriptorIndex = _descriptorHeaps.size();
    _descriptorHeaps.push_back(nullptr);
} else{
    cbo->_descriptorIndex = preCBO->_descriptorIndex;
}

D3D12_RANGE cbRange = { 0, 0 };
cbo->_buffer->Map(0, &cbRange, (void**)&cbo->_address);

cbo->_rootParameterIndex =AddParameterAndRangeForRootSignature(visibility,
D3D12_DESCRIPTOR_RANGE_TYPE_CBV, regno);

_bufferObjects.push_back(cbo);
return _bufferObjects.back();

```

}

と

```
DX12BufferManager::CreateDescriptorHeaps() {
    //ルートシグネチャ生成後は扱えません
    assert(!_createdRootSignature);

    auto& dx12 = DirectX12::GetInstance();
    auto dev = dx12.Device();
    std::vector<unsigned int> descriptorNums(_descriptorHeaps.size());
    for (auto& b : _bufferObjects) {
        descriptorNums[b->_descriptorIndex] += b->_descriptorNum;
    }

    HRESULT result = S_OK;
    for (int i = 0; i < _descriptorHeaps.size(); ++i) {
        D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc = {};
        cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
        cbvHeapDesc.NumDescriptors = descriptorNums[i];
        cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
        result = dev->CreateDescriptorHeap(&cbvHeapDesc,
            IID_PPV_ARGS(&_descriptorHeaps[i]));
    }

    for (auto& b : _bufferObjects) {
        b->_descHeap = _descriptorHeaps[b->_descriptorIndex];
    }

    auto descSize = dev-
        >GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
    std::vector<unsigned int> descriptorOffsets(_descriptorHeaps.size());

    for (auto& b : _bufferObjects) {

        auto buffLoc = b->_buffer->GetGPUVirtualAddress();
        auto handle = b->_descHeap->GetCPUDescriptorHandleForHeapStart();

        auto& offset = descriptorOffsets[b->_descriptorIndex];
        handle.ptr += offset;
    }
}
```

```

    b->_heapHandleOffset = offset;
    //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;
        offset += descSize;

    }
}

と

DX12BufferManager::CreateRootSignature() {
    auto& dx12 = DirectX12::GetInstance();
    auto dev = dx12.Device();

    ID3DBlob* signature = nullptr;//シグネチャシリアル化用
    ID3DBlob* error = nullptr;//エラー取得用
    //サンプラー…ここは適切じゃないかもしないけど
    D3D12_STATIC_SAMPLER_DESC samplerDesc = {};
    samplerDesc.Filter = D3D12_FILTER_MIN_MAG_MIP_LINEAR;
    samplerDesc.AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
    samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
    samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
    samplerDesc.ComparisonFunc = D3D12_COMPARISON_FUNC_ALWAYS;
    samplerDesc.MaxAnisotropy = 0;
    samplerDesc.MaxLOD = D3D12_FLOAT32_MAX;
    samplerDesc.MinLOD = 0;
    samplerDesc.MipLODBias = 0;
    //ルートシグネチャーを宣言
    D3D12_ROOT_SIGNATURE_DESC rsd = {};
    rsd.Flags = D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT;
}

```

```

//ここに今から「レンジ」とサンプラーを設定する。
rsd.NumStaticSamplers = 1;
rsd.pStaticSamplers = &samplerDesc;
rsd.NumParameters = _rootParams.size(); //テクスチャと定数バッファの合計数
rsd.pParameters = &_rootParams[0];

//デスクリプタレンジアドレスの再設定
for (int i = 0; i < _rootParams.size(); ++i) {
    _rootParams[i].DescriptorTable.pDescriptorRanges = &_descRanges[i];
}

//シグネチャシリアルライズ
auto result = D3D12SerializeRootSignature(&rsd,
    D3D_ROOT_SIGNATURE_VERSION_1,
    &signature,
    &error);

//ルートシグネチャの生成
result = dev->CreateRootSignature(0,
    signature->GetBufferPointer(),
    signature->GetBufferSize(),
    IID_PPV_ARGS(&_rootSignature));
_createdRootSignature = true;
}

```

てな感じです。

BMP以外も読めるようにしよう

前にも一度紹介しましたが、DirectXTexという公式のライブラリを用いれば BMP 以外も読め るようになります。

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

落とします。

適当な場所に解凍します。解凍したらコンパイル(ビルド)します。

プロジェクトは DirectXTex_Desktop_2015 と DirectXTex_Desktop_2015_Win10 がありますが、

Win10の方にしておきましょう。

この時、アーキテクチャ(x86 なのか 64 なのか)は自分が今 DirectX12 で開発しているものと合わせておきましょう。

そうすると、まあまずビルドは通ると思います。

そうすると

DirectXTex\Bin\Desktop_2015_Win10\Win32\Debug

の中に、ライブラリ DirectXTex.lib ができているはずです。…とやろうと思ったが、今回は WIC だけが必要なので、

WICTextureLoader12.cpp と WICTextureLoader12.h を自分のプロジェクトに持ってきてリビルドしよう。

多分通るとは思う。そして使い方だが、

LoadWICTextureFromFile

を使用する。

とりあえずキャラクターを我那覇さんに変更します。



我那覇さんは全部のテクスチャが jpg ので、それらのビットマップが読み込めずにこのような感じになります。

LoadWICTextureFromFile

ところでこの LoadWICTextureFromFile にはちょっとばかり面倒なところがあります。それはこの関数が wchar_t(Unicode)にしか対応していないという事です。そして PMD ファイル内のファイル名指定は char(ASCII or SJIS)だという事です。

つまりそもそも皆さんの感覚だと「文字型」は 1 バイト…なのだと思います。ところが wchar_t は確かに「文字型」ですが、こいつは 1 文字を 2 バイトとしています。つまりそもそも「型が違う」わけです。

型が違うものを別の型として扱うには「キャスト」か「変換」が必要になってきます。「キャスト」はご存知のように float 型を int 型に変えたり、int 型を float 型に変えたりするものです。それに対して「変換」というのはまったく別の意味にしてしまうものです。

例えば'A'という文字は、48という数値でもありますので、(int)キャストをすると 48 という数値になります。それはいいですね？ただし

'3'という文字を(int)にキャストするとどうなるでしょう？3 という数値になつてしまい、そこですが、実際には 51 とかいう数値が入ってしまいます。これは「文字コード」であり、その中の文字が示す「意味」とは関係ない数値です。

では'3'という文字から 3 という数値を得たければどうすればいいでしょうか？そういう時に変換関数を使います。今回の 3 の例で言うと atoi 関数を使用することになるでしょう。

それと同じような感覚で「char」が示すものを「wchar_t」に変換する必要があります。その変換関数ですが MultiByteToWideChar という関数です。

<https://msdn.microsoft.com/ja-jp/library/cc448053.aspx>

余談ですがこれ、逆もあって WideCharToMultiByte 関数もあります

<https://msdn.microsoft.com/ja-jp/library/cc448089.aspx>

ともかく今回はこの MultiByteToWideChar を使いましょう。

MutiByteToWideChar

これちょっと使い方にクセがあるんでよく読んでおきましょう。通常のマルチバイトとワイドキャラ(Unicode)は、文字数というかバイト数が事前に計算できるとは限りません。どういう事なのかというと、

例えば

This is a pen.

は通常であれば 14 文字なので、14 バイト + null 文字で 15 バイトといったところでしょう…。で、通常なら倍の 28 バイト + 2 バイトで 30 バイトとかになるはずなのですが、そうとは限らないのです。ですから、よくありがちなのが、ワイド文字を受け取るバッファを作るときに

```
malloc(sizeof(パス名)*2);
```

なんてやつちやう人がいますけど、それはやめましょう。やめてください。そこでリファレンス

をよく見てみましょう。

戻り値

cchWideChar に 0 以外の値を指定し、関数が成功すると、*lpWideCharStr* が指すバッファに書き込まれたワイド文字の数が返ります。

cchWideChar に 0 を指定し、関数が成功すると、変換後の文字列を受け取るバッファに必要なサイズ（ワイド文字数）が返ります。

ご覧の通りだよ!!

つまり…勘のいい人なら分かると思いますが、1回の変換において、この関数は2回呼ばれます。1度目はバイト数を測るため。2度目は実際に変換された文字列を入れるため。

というわけで、GetUnicodeFromSJIS なんていう関数を作ります。

```
std::wstring GetUnicodeFromSJIS(const char* str);
```

ちなみに *wstring* は *std::string* の Unicode 版です。

さらに上の解説に書いてますが、戻り値はバイト数ではありません。ワイド文字の数です。つまり1度目のコールで取得した「文字の数」ぶんだけ *wstring* を確保すればいいのです。

```
//SJIS→Unicode変換
//@param str 入力文字列(SJIS)
//@return 出力文字列(Unicode)
std::wstring
TextureCreator::GetUnicodeFromSJIS(const char* str) {
    //予め文字数を取得
    auto bytesize = MultiByteToWideChar(CP_ACP,
                                         MB_PRECOMPOSED | MB_ERR_INVALID_CHARS,
                                         str, -1, nullptr, 0);

    //返す用の文字列を宣言し、必要な文字数で初期化
    std::wstring wstr;
    wstr.resize(bytesize);
```

```

//変換して返す
bytesize = MultiByteToWideChar(CP_ACP,
    MB_PRECOMPOSED | MB_ERR_INVALID_CHARS,
    str, -1, &wstr[0], bytesize);
return wstr;
}

```

うーん。ではここまでやれば LoadWICTextureFromFile は使えますかね？

ひとまずこの時点で動いている状態でコミットして…実験しましょう。



進行とか知るか！バカ！そんな事より実験だ！

前回の授業ではホントにすまんかった。うまくいくと思っていたのだ。

ちょっとなめてた。

あと、言い訳を言わせてもらうなら、やっぱり資料が少なすぎる…OTL
DirectX11 の時はうまくいってたんだよう…。

さて、WIC 関係のサンプルコードを見ているとやっぱり UpdateSubresources を使ってデータを流し込んでいるところを見ると LoadWICTextureFromFile では中身が更新されていない状況のようだ。

では UpdateSubresources を見ると

[https://msdn.microsoft.com/query/dev14.query?appId=Dev14IDEF1&l=EN-US&k=k\(UpdateSubresources\);k\(DevLang-C%2B%2B\);k\(TargetOS-Windows\)&rd=true](https://msdn.microsoft.com/query/dev14.query?appId=Dev14IDEF1&l=EN-US&k=k(UpdateSubresources);k(DevLang-C%2B%2B);k(TargetOS-Windows)&rd=true)

となっている。だいたいわかるが pIntermediate ってのが謎である。なんだっそら!!!
Intermediate ってのは中間形式だの仲介者だのそういう意味でつかわれる言葉なのだが、これを使うのさ…。

という話なのだが、ちょっと今更だが、CPU から GPU へ渡す仕組みの部分が関係している話なのだ。

んで、LoadWICTextureFromFile じたいは成功するんですよ。S_OK返ってくるんですよ。でも結局最終的な出力として



フケイデアルゾ

メジエド様みたいになっちゃいます。

原因は

textureBuffer->WriteToSubresource

が失敗するからです。何をどうやっても失敗してしまいます。原因を見つけるのにかなり苦労しましたが、まあ原因じたいがわかったので解説していきます。

元々、WriteToSubresource が通っていた時のテクスチャのヒーププロパティの設定は…

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

でした。

で、

LoadWICTextureFromFile

の中身を調べてみたわけですよ。で、かなり潜ってみてわかったのですが…

D3DX12_HEAP_PROPERTIES defaultHeapProperties(D3D12_HEAP_TYPE_DEFAULT);

と初期化されました。

まあ、これ、この部分を

D3D12_HEAP_PROPERTIES defaultHeapProperties = {};

defaultHeapProperties.Type = D3D12_HEAP_TYPE_CUSTOM;

defaultHeapProperties.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_WRITE_BACK;

defaultHeapProperties.MemoryPoolPreference = D3D12_MEMORY_POOL_L0;

defaultHeapProperties.CreationNodeMask = 1;

defaultHeapProperties.VisibleNodeMask = 1;

とすれば、いいんですけどね。



まだメジエドちゃん

体のテクスチャが表示されていませんが、これは別の原因によるものなので、ひとまずこれで解決するのですが…でも、ぶっちゃけた話をしてると、MSが作っているライブラリの中身なんて書き換えたくなれ!のが本音です。何故か?というと、他人の奴の中身を下手にいじると別の予期しない部分で後々バグが出て悩まされるというのもよくあることです。

ということで、別の手を考える必要があります。

さて、ところで、なぜ D3D12_HEAP_TYPE_DEFAULT だと WriteToSubresources で書き込めないのでしょうか…

色々と資料を見ても分からなかつたので不本意ですが、解説サイト(日本語)を見てみます。

https://shikihuiku.wordpress.com/2015/03/31/about_3d12_heap_type_upload/

「D3D12_HEAP_TYPE_DEFAULT フラグは、いわゆる GPU 側のメモリ(VidMem)を確保するためのものです。Map()することはできないので、他の Resouce からコピーしてデータを書き換えます。」

なるほど。ちなみに CUSTOM は

「D3D12_HEAP_TYPE_CUSTOM フラグを用いると、アプリケーション側で明示的にリソースの配置されるメモリが VidMem か SysMem か、Map 可能かなどを指定することができます。」

と書かれています。

さらにもう一つのヒープ指定の D3D12_HEAP_UPLOAD ですが、ちょっと説明が長いです。

「D3D12_HEAP_TYPE_UPLOAD フラグは、Heap/Resource を確保する際に指定するフラグで、CPU から GPU にデータを転送する用途で使用する Heap に付けます。この領域は、Map() 可能で、CPU から情報を書き込めるだけでなく、GPU から直接参照することができるようにも設定できます。そのため、DX のアプリケーション内では大変有用で、ConstantBuffer をここに書き込んで Shader から参照したり、VB/IB などを書き込んで、これをそのまま使用したり、または、**その内容を D3D12_HEAP_TYPE_DEFAULT 領域にコピー** をしたりすることができます。」

ん?

なるほど…。そういう事が…。

ちなみに ZeroGram 氏も ProjectASURA 氏も WriteSubresource ではなく UpdateSubresource を使用していました。おそらくは WIC などのロード関数が DEFAULT 設定になっており、直接書き込むことができないため WriteSubresource を使えないと早々に判断したのでしょう。

では何を使って転送するのか…

前述した UpdateSubresources を使うようです。

[https://msdn.microsoft.com/query/dev14.query?appId=Dev14IDEF1&l=EN-US&k=k\(UpdateSubresources\);k\(DevLang-C%2B%2B\);k\(TargetOS-Windows\)&rd=true](https://msdn.microsoft.com/query/dev14.query?appId=Dev14IDEF1&l=EN-US&k=k(UpdateSubresources);k(DevLang-C%2B%2B);k(TargetOS-Windows)&rd=true)

いや、あるのは知つたんですが、この引数の内容が意味不明だったので使わなかつたんです。

何處が意味不明だったのかって？前にも書いたけど、pIntermediateなのよね。さて…ここまでなんとか我慢して読み進めてこれた諸君には分かっているだろう？

そう、WIC の中で使われているのが、DEFAULT であり、こいつは GPU にメモリを確保し、Map も書き換えもできないため、UPLOAD で作ったバッファを pIntermediate で作って、それをコピーするイメージです。

つまり、

- ①LoadWICTexture でロードする
- ②UPLOAD 指定でバッファ作る
- ③UploadSubresources でデータを流し込む
- ④バリアで待つ

ていうかさ… LoadWICTextureFromFile って名前ならロードしてバッファに投げるところまでやってくれよ…あ、それは逆にちょっと問題かな。Microsoft が作ったとはいえ、ある意味有志が作ったモノなので、ドキュメントがないのも痛いよね…。

で、UploadSubresources を使ってみたのですが、結果を言うと、3 時間以上費やして…うまくいきませんでした。一応うまくいくはずなんですが、これ以上悩んでも授業に支障が出るため、LoadWICTextureFromFile を改造して一旦先に進みましょう。その後で UploadSubresource のやり方について軽く解説して先に進んでしまいましょう。

LoadWICTextureFromFile の改造

WICTextureLoader12.cpp の中に CreateTextureFromWIC という関数があって、その関数の最後の方に

```
CD3DX12_HEAP_PROPERTIES defaultHeapProperties(D3D12_HEAP_TYPE_DEFAULT);
```

と書いてある部分があります。こいつをコメントアウトして WriteSubresource が通るように、以下のように書き換えてあげます。

```
D3D12_HEAP_PROPERTIES defaultHeapProperties = {};
defaultHeapProperties.Type = D3D12_HEAP_TYPE_CUSTOM;
defaultHeapProperties.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_WRITE_BACK;
defaultHeapProperties.MemoryPoolPreference = D3D12_MEMORY_POOL_L0;
```

```
defaultHeapProperties.CreationNodeMask = 1;  
defaultHeapProperties.VisibleNodeMask = 1;  
で、呼び出し側コードを
```

```
D3D12_RESOURCE_DESC resdesc = {};  
resdesc=textureBuffer->GetDesc();  
D3D12_BOX box = {};  
box.left = 0;  
box.right = (resdesc.Width); //imagedata->GetWidth();  
box.top = 0;  
box.bottom = (resdesc.Height); //imagedata->GetHeight();  
box.front = 0;  
box.back = 1;  
result = textureBuffer->WriteToSubresource(0, &box, decodedData.get(), subres.RowPitch,  
subres.SlicePitch);  
とします。
```

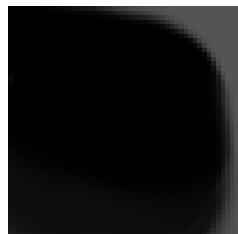
ひとまずこれでテクスチャが表示されます。なお、これでも恐らく何となくメジエド状態のは維持されていますが部分的にテクスチャが適用されているのは分かると思います。



ではなぜお肌のテクスチャが剥げてるのでしようか？

その秘密は…hada.spa にあります。実は*.spa ファイルってのは中身は完全なる bmp ファイルになってて、単純に拡張子を spa にしているだけなのだ!!!!

さて、それでは拡張子を bmp にしてみて、中身を見てみよう…



実は spa および sph に関してはちょっとややこしい話が関連してるので、これは後に回したいと思いますので、拡張子が spa や sph だった場合は「一旦」読み込まないようになります。

そうすると、このように…



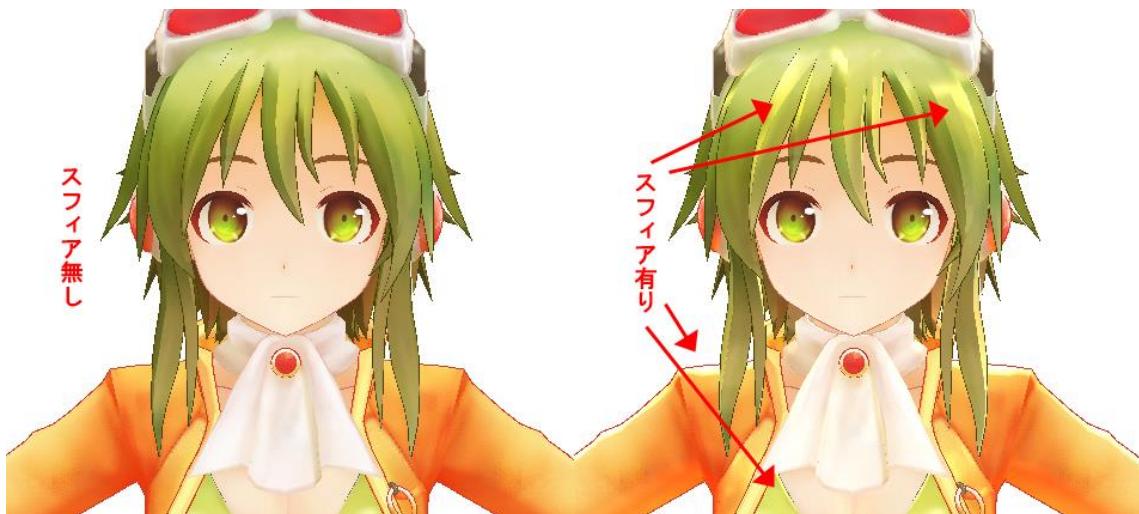
ある程度は思った通りの画像になります
とりあえずここで一旦の区切りと行きましょう。

sph とか spa ってなに？

これはちょっと面倒な話なのですが、MMDにおいて spa とか sph とか出できたらそれは「スマップ」と呼ばれるものです。

で、このスフィアマップって概念…MMDくらいでしか聞いたことないんですね。環境マップの一つなんですけど、まあ簡単に言うと視線の反射ベクトルの方向を見たときに何を写すべきかというマップなのよね。

MMDのモデルにおいては、よく髪の艶とかに応用されてたりします。



髪の艶の場合はどっちかというとスペキュラマップみたいな意味で使われてますけどね。あと今回の件であれば「肌艶」ですね。

まあ、スフィアマップなんて用語はCG検定には出ないので、理屈だけ大まかに分かって実装できればそれでいいです。

とりあえず一番簡単に「それっぽく」見せる実装としては反射ベクトルとかそういうのを抜きにして、法線ベクトルをUVに割り当ててみます。

ちなみにspaは加算で使用されるマップです。とりあえずレジスタ1番(†1)に入っていると仮定してシェーダを組むと

```
Texture2D<float4> spa:register(†1);
```

こうして、あと、こいつは「加算」なので

うまい事計算してあげれば

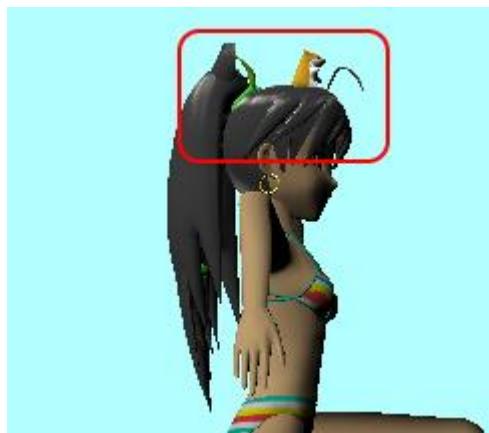


このようにぬるぬるてかてかになります。

ピクセルシェーダ側で、

```
color += spa.Sample(smp, data.normal.xy*float2(0.5, -0.5) + float2(0.5, 0.5)).rgb;
```

のように書いてあげると、

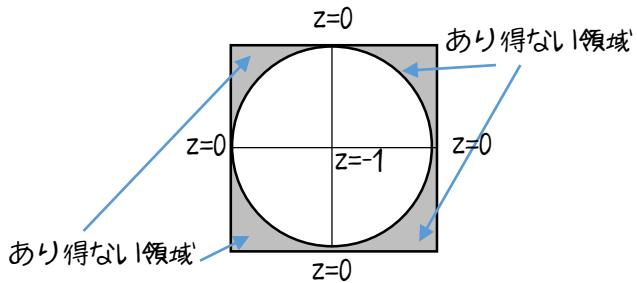


このように艶的なのが入ります。

なお、↑のピクセルシェーダについてですが、

理屈をお話ししていきます。

スフィアマップのテクスチャは図のように、正方形の領域に内接するように書かれた円の形をしています。



そしてモデルにおける法線ベクトルを (nx, ny, nz) とすると、正規化済みのため
 $nx^2 + ny^2 + nz^2 = 1$ が成り立っています。

もし、 $z=0$ で法線が完全にそっぽを向いていたとしても $nx^2 + ny^2 = 1$ が成り立つはずです。

つまり円です。更に言うと $z \neq 0$ 以外の実数であれば xy は円の内部になるはずです。
 つまり、法線の xy を uv に使用しても円の外側に出ることはないとため、見た目おかしくはならないのです。ただしこのやり方は、簡易版なので、より、正解に近くするにはきちんと、視線の反射ベクトルを作つて…

```
float3 vray = data.pos - eye; // 視線ベクトル
vray = reflect(vray, data.normal); // 視線の反射ベクトル
vray = normalize(vray);
color += spa.Sample(smp, vray.xy / 2 * float2(1, -1) + float2(0.5, 0.5)).rgb;
```

のようにします。

で、これでも正確ではなくて、実は`data.pos`は $(-1 \sim 1, -1 \sim 1, 0 \sim 1)$ 空間に正規化されているため、きちんとした「視線ベクトル」を得たければ Camera および Projection 行列をかけず、ボーン行列とワールド行列のみをかけるための `origPos` ってのを作ります。作るときに注意点ですが…

```
struct Output {
    float4 spos:SV_POSITION;
    float4 pos:POSITION0;
    float4 origpos:POSITION1;
    float3 normal:NORMAL;
    float2 uv:TEXCOORD;
};
```

POSITION 系が2つあるので、ひとつずつ0,1を末尾につけてください。セマンティクスに同じのがあるとコンパイル通らないようです。

```
output.origpos = mul(m, pos);
pos = mul(mul(viewproj, m), pos);
```

こんな感じに、分けておきます。で、視線ベクトルは

```
float3 vray = data.origpos - eye;
vray = normalize(vray);
```

とします。

この vray を元に

```
vray=reflect(vray, data.normal);
color += spa.Sample(smp, vray.xy * float2(0.5, -0.5) + float2(0.5, 0.5)).rgb;
```

とします。

せつかくだからスペキュラもアンビエントも入れる

もうせつかくだから、この流れついでにスペキュラもアンビエントも入れましょう。データとしてはあるわけだし…。

手順

①GPUに投げるデータを増やす(アルファ、スペキュラ強さ、スペキュラ色、アンビエント)

②hlsl側で受け取る

③ライトの反射ベクトルを作る

④スペキュラ成分を計算

⑤アンビエントを加算

こんな感じ。

まずは GPU 側に投げるデータを増やします。

CBufferを変更します。

```
struct CBuffer { //ワールド行列とViewProj行列
    XMATRIX world; //ワールド行列
    XMATRIX viewproj; //カメラ、プロジェクション
    XMFLOAT3 diffuse; //ディフューズ
    float alpha; //アルファ
    float specularity; //スペキュラ強さ
    XMFLOAT3 specular; //スペキュラ色
    XMFLOAT3 ambient; //環境光成分
    int existTexture; //テクスチャあるか
    int existSPA; //スフィアマップあるか
    XMFLOAT3 eye; //視点座標(ピクセルシェーダに「視線」をつくるため)
```

```
};  
当然ながら渡せるものは全部入れておきます。  
cbuffTemp->diffuse = mat.diffuse;  
cbuffTemp->alpha = mat.alpha;  
cbuffTemp->specularity = mat.specularity;  
cbuffTemp->specular = mat.specular;  
cbuffTemp->ambient = mat.ambient;
```

もちろんhlsl側では受け取れるようになっています。

```
cbuffer mat : register(b0) {  
    matrix world;  
    matrix viewproj;  
    float3 diffuse;//拡散反射成分(基本色)  
    float alpha;//拡散反射成分(基本色)  
    float specularity;//スペキュラ強さ  
  
    float3 specular;//スペキュラ色  
    float3 ambient;//環境光色  
  
    int existTex;//テクスチャあるか  
    int existSPA;//スフィアマップあるか  
    float3 eye;//視点  
}
```

すりいぶん増えてしまいましたね。

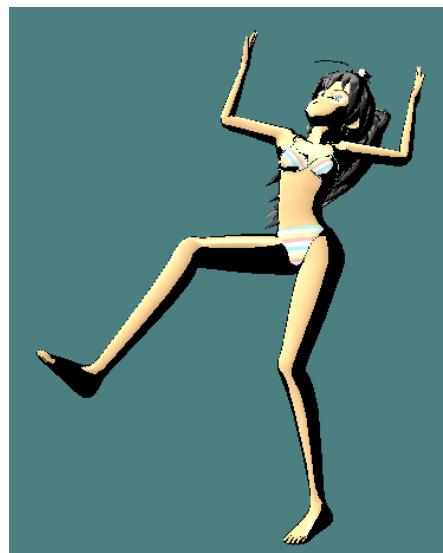
ピクセルシェーダにてスペキュラ計算

```
float3 vray = data.origpos - eye;//視線ベクトル  
vray = normalize(vray);//正規化  
float spec=saturate(pow(dot(reflect(-light, data.normal),-vray),specularity));//スペキュラ計算
```

あとはこれを既に計算済みのカラーに足してあげます。ちなみに saturate 関数に疑問を持っている人がいるかもしれないるので解説しておくと、こいつは出てきた数値を 0~1 の範囲にクランプしてくれるものです。

これをつけてないとスペキュラ値がマイナスになってしまったときに、色が暗くなってしまうからです。

まあ面白い効果が得られたので、見せておく。



ちょっとグラスホッパーぽくて僕は好きかな
で、この `saturete` して持ってきたスペキュラ値と、スペキュラをかけて、アンビエントを足す
と



このような感じで表示されます

ポージング

ついにやってきましたよ。ポージングです。アニメーションの前段階です。
で、MMD にはポーズデータという `*.vpd` ってデータがあるんですが、こいつはテキストファイルなんですね。正直面倒なので、VMD ファイルでポーズを取らせようと思います。
`¥¥132sv¥gakuseigamero¥rkawano¥MMD`
に `VMD.SYM` があります。これを自分の `TSXBIN.exe` のフォルダに入れて、VMD ファイルを開いてみましょう。

| | |
|----------------------------|--|
| eader.VmdHeader[0] | 56 6F 63 61 6C 6F 69 64 20 4D 6F 74 69 6F 6E 20 Vocaloid Motion |
| eader.VmdHeader[16] | 44 61 74 61 20 30 30 30 32 00 00 00 00 00 00 Data 0002 |
| eader.VmdModelName[0] | 8F 89 89 B9 83 7E 83 4E 00 FD FD FD FD FD FD FD FD 初音ミク |
| eader.VmdModelName[16] | FD FD FD FD |
| otion_count | 0000008C |
| otion[0].BoneName[0] | 83 5A 83 93 83 5E 81 5B 00 FD FD FD FD FD センター |
| otion[0].FlameNo | 00000000 |
| otion[0].Location[0] | 00000000 00000000 00000000 |
| otion[0].Rotatation[0] | 00000000 00000000 00000000 3F800000 |
| otion[0].Interpolation[0] | 14 14 00 00 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkk |
| otion[0].Interpolation[16] | 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 00 kkkkkkkk |
| otion[0].Interpolation[32] | 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 00 00 kkkkkkkk |
| otion[0].Interpolation[48] | 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 00 00 00 kkkkkkkk |
| otion[1].BoneName[0] | 8F E3 94 BC 90 67 00 FD FD FD FD FD FD FD 上半身 |
| otion[1].FlameNo | 00000000 |
| otion[1].Location[0] | 00000000 00000000 00000000 |
| otion[1].Rotatation[0] | 00000000 00000000 00000000 3F800000 |

ご覧のとおりです。今回重要なのは「ボーン名」と「Rotation」です。

ところでこの Rotation…何に対する回転なんでしょうね？

ここで今までの Rotation に関する考え方を変えなきゃいけないんですが、準備はいいかな？

今までみなさんは X 軸回転、Y 軸回転、Z 軸回転のような回転しかしてきませんでした。

その回転だけで、全てがまかんえると思つてますか？

残念ながらそうは行かないんですね…。理由は後からぼちぼち言いますけど、「任意軸回り」の回転」というものを使用します。

で、これも通常通りの回転でやってもいいんですが、それだと

(n_x, n_y, n_z) を軸に θ だけ回転する場合

$$\begin{pmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_x n_y (1-\cos\theta)-n_z \sin\theta & n_z n_x (1-\cos\theta)+n_y \sin\theta \\ n_x n_y (1-\cos\theta)+n_z \sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_y n_z (1-\cos\theta)-n_x \sin\theta \\ n_z n_x (1-\cos\theta)-n_y \sin\theta & n_y n_z (1-\cos\theta)+n_x \sin\theta & n_z^2(1-\cos\theta)+\cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

まあこんな式があるんですけど残念ながらシンプルじゃないんですね。

ということで「クォータニオン」という数学的な概念が利用されます。

クォータニオンとは

<https://ja.wikipedia.org/wiki/%E5%9B%9B%E5%85%83%E6%95%B0>

http://marupeke296.com/DXG_No10_Quaternion.html

クォータニオンの話をする前にちょっと聞いておきましょう。皆さんには「虚数」というのを知っていますでしょうか？

複素数のおさらい

そう、2乗するとマイナスになる数です。大抵の場合は虚数を Imaginary Number の略で i とおきます。聞いたことはありますね？

そしてこの「虚数」と「実数」を組み合わせたものを「複素数」と言います。

複素数は

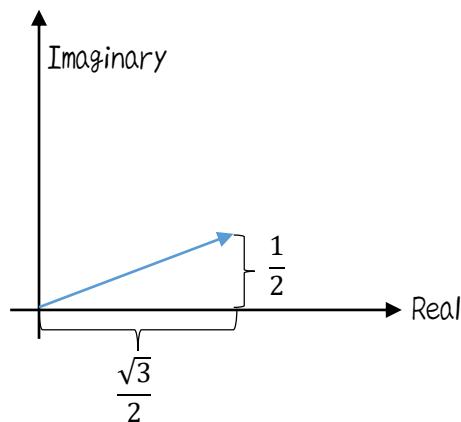
$$z=a+ib$$

と表されます。複素数のことを英語で Complex Number と言います。ちなみに「実数」は Real Number です。 a の事を「実部」と言い、 b の事を「虚部」と言います。で、こいつですね。面白いことに「回転」を表すことができるんですよ。Real を横軸に、Imaginary を縦軸に考えてください。

そうすると 30° 回転とは

$$\frac{\sqrt{3} + i}{2}$$

となります。

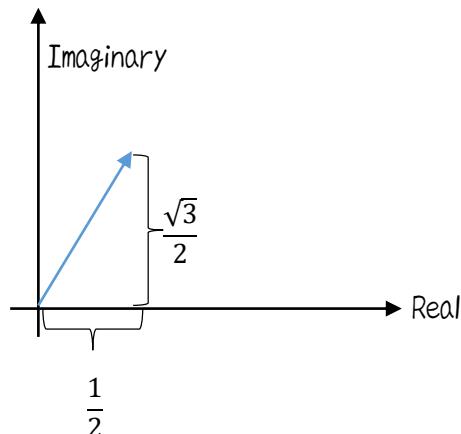


この面白いところは「乗算」すると回転が進むことがあります。

例えば、 $30^\circ + 30^\circ = 60^\circ$ だが、これは複素数を使うと

$$\left(\frac{\sqrt{3}+i}{2}\right)\left(\frac{\sqrt{3}+i}{2}\right) = \left(\frac{\sqrt{3}^2 + 2\sqrt{3}i + i^2}{2^2}\right) = \frac{(3-1)+2\sqrt{3}i}{4} = \frac{1+\sqrt{3}i}{2}$$

つまり 60° ということになります。



お分かりいただろうか？これが複素数である。一般化して言うと

$$Z = \cos \theta + i \sin \theta$$

なわけだ。なんで急にこういう話をしたのか? というと、クオータニオンってのが複素数を3Dに拡張したものだからだ。

ともかく、複素数を用いて回転を表すことができることはわかりましたね?

四元数

クオータニオンは3Dに拡張された複素数なので、以下のように表します。

$$Q = ix + jy + kz + w$$

です。なお、 i, j, k はどれも次元の異なる虚数記号で

$$i^2 = j^2 = k^2 = -1$$

$$ij = k, jk = i, ki = j$$

$$ijk = -1$$

が成り立つような「虚数?」と定義されます。クオータニオンの場合 w を実部といい、 i, j, k を虚部と言います。Unityにおける「回転」を表す Quaternion

こいつがモノごつい便利なのは「任意軸回りの回転」をさっきの式よりかはシンプルに記述することができる点です。

クオータニオンを使用して任意軸回転する

まず、任意ベクトル V 周りの θ 回転のクオータニオンは

$$Q = \left(\frac{V_x \sin \theta}{2}, \frac{V_y \sin \theta}{2}, \frac{V_z \sin \theta}{2}, \frac{\cos \theta}{2} \right)$$

と表されます。MMD データの場合、4つの数値が指定されていますが、それはこういう並びだと思ってください。ともかくクオータニオンで格納されているのはありがたいことです。

このクオータニオンを回転行列にするには

$$\begin{pmatrix} Q_x^2 + Q_y^2 - Q_z^2 - Q_w^2 & 2(Q_y Q_z - Q_x Q_w) & 2(Q_y Q_w + Q_x Q_z) \\ 2(Q_y Q_z - Q_x Q_w) & Q_x^2 - Q_y^2 + Q_z^2 - Q_w^2 & 2(Q_y Q_w - Q_x Q_y) \\ 2(Q_y Q_w - Q_x Q_z) & 2(Q_z Q_w + Q_x Q_y) & Q_x^2 - Q_y^2 + Q_z^2 - Q_w^2 \end{pmatrix}$$

こういう回転行列を作ります。別にシンプルではないですね。それでもクオータニオン使う理由はジンバルロック防止と球面線形補間のためです。

クオータニオンに関しては「道具」と割り切ってください。いつもは「理屈」をしつかり教える僕ですが、こればっかりは「理屈」を知るのはコスパが悪いです。

何故かと言うと、「テンソル」「スピノール」などの概念を知る必要が出てくるからです。ちょっとやめておきましょう。

さて、今回の MMD のデータに関しては既にクオータニオンで入っているので、これを利用するに

は

XMMatrixRotationQuaternion 関数を利用します。

https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixrotationquaternion.aspx

ま、それはともかくひとまずはロードしましよう。

ポーズデータのロード

http://blog.goo.ne.jp/torisu_tetosuki/e/bc9f1c4d597341b394bd02bb4597499d

を見ましょう。

```
struct VMDHeader {
```

```
    char VmdHeader(30); // "Vocaloid Motion Data 0002"
```

```
    char VmdModelName(20); // カメラの場合:"カメラ・照明"
```

```
};
```

ヘッダを読み込んだら次の8バイトがモーションデータ数です(ヘッダ32バイトでいいじゃん
…なんでその2バイトをケチるのか…)

そして次にモーションデータですが、以下のようになっています。

```
struct VmdMotion { // 111 Bytes // モーション
```

```
    char BoneName(15); // ボーン名(くそが!!!!)
```

```
    DWORD FrameNo; // フレーム番号(読み込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
```

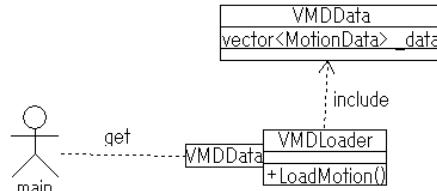
```
    BYTE Interpolation(64); // (4)(4)(4) // 補完(ベジエーデータ)
```

```
};
```

…またツツツツ!!!

なんなんだよ15バイトってよ!!!おどりやくそ!!!

仕方ない…PMDMesh&PMDLoaderと同様にVMDDataというクラスとVMDLoaderを作りましょう。



今回はバーテックス/ドッファもコンスタント/ドッファも使わないのでとにかくロードしましよう。話はそれからだ。ここは特にヒントなしでもできるでしょ?

あ、でもVmdMotionはこう書き換えておいたほうが便利かな?

```
struct VmdMotion { // 111 Bytes // モーション
```

```
    char boneName(15); // ボーン名(くそが!!!!)
```

```

    unsigned int frameNo; // フレーム番号(読み込時は現在のフレーム位置を0とした相対位置)
    XMFLOAT3 location; // 位置
    XMFLOAT4 rotation; // Quaternion // 回転
    unsigned char interpolation[64]; // [4][4][4] // 補完(ベジェーデータ)
};

違いますかね？

```

で、とりあえずメイン側に対して今教える必要があるのは「ボーン名」と「クォータニオン」ですね。

```

//モーション情報(1個あたり)
struct MotionData{
    std::string boneName;//ボーン名
    XMVECTOR quaternion;//クォータニオン
};

```

ですから先程の構造体を pack(1)でも何でも良いので、とにかく読み取って、その後で VMDData クラスに必要なものを入れてください。今は必要なものだけいいです。そっちのほうが考えやすいから。でももちろんこれも複数あるので std::vector で配列しといてください。

分かりませんか？

構造的にはですね。

- ①VMDData の中に vector<MotionData> が入ってて、それをクライアント側に提供します
- ②VMDLoader は VMDData を new 生成して、それをクライアント側に返します
- ③VMDMotion はロード時に使用するだけ

というわけです。

ここまで頑張つづいてこれたんだから、ここは僕のソースコードなんか見ずに考えられるはず。

あ、最後に fclose は忘れないようにね。忘れやすいからね。

一方、クライアント側では…

ひとまずクライアント側は VMDLoader をインクルードしてください。そして LoadMotions 関数を呼び出してください。いや、ロード関数名はなんでもいいんですけど、ともかくロードしてください。

```
VMDData* vmddata = vmdloader.LoadMotions("pose.vmd");
```

そしてこの vmddata にモーションデータ(ボーン名とクォータニオン)が入りまくっています。次にやるべきことは、リファクタリング時に DeformBones 関数を作ったと思いますが、こいつを改造します。リファクタリングしないところが面倒になってたんですよ。やっててよかったです。

もともと DeformBones は自分でボーン名と回転を指定していたので

```
void DeformBones(PMDMesh* mesh, VMDDATA* vmddata)
{
    //左ひじ変形
    int elbowIdx = mesh->BoneMap()["左ひじ"];
    XMVECTOR offsetVec = XMLoadFloat3(&mesh->Bones()(elbowIdx).headPos);
    XMFLOAT3 tailPos = XMLoadFloat3(&mesh->Bones()(elbowIdx).tailPos);
    XMMatrix bone = XMMatrixTranslationFromVector(-offsetVec);
    bone *= XMMatrixRotationZ(XM_PIDIV2);
    bone *= XMMatrixTranslationFromVector(offsetVec);
    mesh->BoneMatrices()(elbowIdx) = bone;

    int shoulderIdx = mesh->BoneMap()["左手首"];
    offsetVec = XMLoadFloat3(&mesh->Bones()(shoulderIdx).headPos);
    tailPos = XMLoadFloat3(&mesh->Bones()(shoulderIdx).tailPos);
    bone = XMMatrixTranslationFromVector(-offsetVec);
    bone *= XMMatrixRotationZ(XM_PIDIV4);
    bone *= XMMatrixTranslationFromVector(offsetVec);
    mesh->BoneMatrices()(shoulderIdx) = bone;

    int wristIdx = mesh->BoneMap()["左手首"];
    offsetVec = XMLoadFloat3(&mesh->Bones()(wristIdx).headPos);
    tailPos = XMLoadFloat3(&mesh->Bones()(wristIdx).tailPos);
    bone = XMMatrixTranslationFromVector(-offsetVec);
    bone *= XMMatrixRotationZ(XM_PIDIV2);
    bone *= XMMatrixTranslationFromVector(offsetVec);
    mesh->BoneMatrices()(wristIdx) = bone;
}
```

こんな感じのソースコードだったと思いますが、もう自分でやる必要はないのです。何故ならボーン名と回転情報は今作った vmddata に入っているんですから。

というわけで元の直指定ソースコードは潔く消しちゃいます。あとは全てのボーンモーションデータに対して処理を行いたいので当然 for 文を使います。vmddata の持つデータで回します。

あと、やることは↑の肘曲げとほぼ同じです。回転がクォータニオンになるだけです。

1. それぞれのボーン起点をとってくる
2. 中心移動行列を作る
3. クォータニオン回転行列をかける
4. 元の位置移動行列をかける

https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixrotationquaternion.aspx

を見ると難しそうですが、とにかくロードしたクォータニオン値をいれて見れ。

うまくいけば



こうなるはずじゃけえ。

ちなみに肩、肘、上半身、首をまわしています。なおIK切ってモーションを作れば下半身もMMDで設定したとおりに動きます。



おめでとう!!君はついに2つ目の山を超えたのだ。

ソースコード的にはこうするだけです。

```
for(auto& pose : poses){  
    XMVECTOR q=XMLoadFloat4(&pose.quaternion);  
    BendBoneMatrixes(_boneMap[pose.boneName],  
        _bonenodes,  
        _bones,  
        _bonematrixes,  
        XMMatrixRotationQuaternion(q));  
}
```

次はこれをアニメーションさせてみましょう

おい、アニメーションしろよ!

ついにここまで来たか。

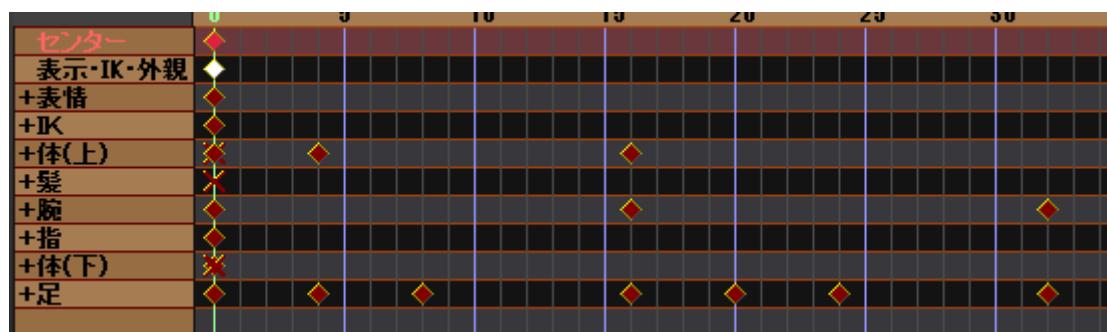
MMD 上でとらせたポーズを反映できているのだから、アニメーションなんて簡単だろう?と思うのかもしれないけど、そうは甘くない。

結構頭使うし、大変だ。

大変な部分としては

1. 時間の概念(タイムライン)について考えなければならぬ
2. キーフレームについて考えなければならぬ
3. 一つのボーンに対して関連するポーズが一つではない
4. クォータニオン値補間(球面線形補間)を考える必要がある
5. (今年はリニア補間だけじゃなくてベジエ補間もやりたいでござる…)

こうですね。で、MMD のモーションデータというのは下のように「タイムライン」上に「キーフレーム」が乗っかっているデータです。



さて、これをプログラムとして実装するのならばどうすればいいのでしょうか?

実装する前に構造を考えよう…。

どういうデータ構造にしようかな…

自分で考えてください。

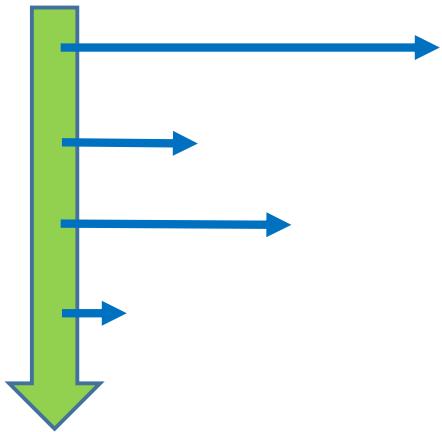
図を描くなり、実験的なプログラムを書くなりして、どうやつたらいい感じに実装できるのかどうかを考えてみてください。

何度も言うようですが、「答えだけ教えてもらう」のは力になりません。君たちに潤沢に時間があるのならばともかく残念ながら時間はありません。そういうわけで授業でやってる事程度は確実に身に着けてほしいのです。

さらに何度も言うようですが、「前を写す」ような写経学習は、時間と気力の無駄遣いです。即刻やめましょう。金と時間と精神力がもったいないです。

さあ考えましょう。

とりあえず僕が考えたものを書いておきます。以下のような考え方のもとに作ってみようと思いました。



MMD のタイムラインそっくりに作ります。縦方向は勿論ボーンを表していますし、横方向はキー フレームを格納するものです。vector の二次元配列でもいいのですが、VMD ファイル内の指定で「ボーン名」の指定があるため、これは map と vector が良いだろうと思います。つまり

```
std::map<std::string, std::vector<Pose>> _motions;
```

のような構造になるかな…と思います。僕はね。もっといい方法を思いついてる人はむしろ教えてあげてください。この哀れな中年に。

ともかくお察しのようなデータを作ります。



```
//フレームとクォータニオン
struct Pose {
    Pose(unsigned int f, const XMVECTOR& q) :frameNo(f), quaternion(q) {}
    unsigned int frameNo;
    XMVECTOR quaternion;
};

typedef std::vector<Pose> Motion_t;
typedef std::map<std::string, Motion_t> Animation_t;
```

お察しの通りですかね？ちなみに `typedef` は知っていますよね？特に説明の必要はないですね？

実装

基本

そして必要なデータを流し込みます。

```
for (auto& pose : poses) {  
    animation[pose.boneName].push_back(Pose(pose.frameNo, XMLoadFloat4(&pose.quaternion)));  
}
```

どうや？

まあ、ここまで大した話じゃない。

ひとまずはこれで作った `animation` データから、前回のポーズを表示できるようにしてみてください。

できましたか？とりあえずはそれぞれのボーンが最初のフレームか最後のフレームの情報を返すようにしましょう。

そうすると…

```
for (auto& a : animation) {  
    BendBoneMatrices(_bonemap[a.first], _bonenodes, _bones, _bonematrices, XMMatrixRotationQuaternion(a.second.back().quaternion));  
}
```

と、書ける。とはいっても、最後のフレームだけを取って来てるだけなので使い物にはならない。

あと、余談だけど `first` とか `second` の意味は分かりますか？

マップにおける `first` とか `second` ってのは…単なる連想配列のキーと値程度の関係です。`std::pair` というものが STL に用意されており、これはただ単に二つの型のペアを格納するためのテンプレートにすぎません。で、`map` 構造というのはつまるところこの `pair` の格納庫に過ぎないわけ。

格納庫に過ぎないわけなんだけど、仕組みとして、`pair` のキー(`first`)から値(`second`)を取得できる仕組みを `map` の内部に持っている。その一つが「連想配列」である。

これがあるために

`bones(ボーン名)=値`

等という書き方ができるようになっているわけだ。で、`a.first` にはキーつまりこの場合はボーンの名前が入っているため、いちいち「値」を見に行かなくても、ボーンの名前が取得できるというわけ。ひとまず Map 状態にしたうえで元と同じようなポーズを決めてくれ。



残念だが、ここまででは下準備に過ぎない。

さて、ここで指定フレームにおける「すべてのボーンの回転情報が欲しい」と仮定する。とりあえず今は補間とか考えずに考えましょう。

指定フレームにおけるポーズを取得する

サーバーから "swing.vmd" というファイルを取得してください。これは腕を上下に動かすだけのデータです。

このデータは 0 フレーム目と 30 フレーム目と 60 フレーム目にそれぞれキーフレームが入っています。

では 0 を入れたら 0 フレーム目のポーズ、30 を入れたら 30 フレーム目のポーズ、60 を入れたら 60 フレーム目のポーズをとるようにしてみましょう。

言ってしまうと

```
Poseforframe(0);
```

```
Poseforframe(30);
```

```
Poseforframe(60);
```

とやれば、それぞれ

0 フレ



30 フレ



60 フレ

こうなればいいと思います。

実装？

そろそろ自分で考えてほしいなあ…もう就職近いんでしょう？

```
for (auto& a : animation) {
    auto it=std::find_if(a.second.begin(), a.second.end(), [frameno](const Pose& p) {return p.frameNo == frameno; });

    if (it == a.second.end()) continue;

    BendBoneMatrixes(_boneMap[a.first], _bonenodes, _bones, _bonematrixes, XMMatrixRotationQuaternion(it->quaternion));
}
```

こうだよ(便乗)

ちなみに僕は find_if を使っています。

find_if 知ってる？

```
#include<algorithm>
で使えるようになるアレだよ!!
find_if(探索開始位置,探索終了位置,ヒット条件関数オブジェクト)
で、探索条件に合致する要素のイテレータが返るよ!!!!これ多分あとで役に立つから使い方覚
えといてね!!!
ちなみにヒット条件の関数オブジェクトは「ラムダ式」を使ってるよ!!!覚えてるかな!?
```

実際にリアルタイムで動かしてみよう

では、実際にリアルタイムで動かしてみましょう…どうなるかなー。
特に工夫しないと…

```
while (true) { // 基本無限ループ
    PoseForFrame(frameno);
    ApplyBoneByMatrixes(_bonenodes, 0, _bones, _bonematrixes);
```

こういう書き方になると思いますが、これだと数秒もたたないうちに



こうなります

何故でしょうか?



違います

これはアニメーションさせようとすると起こることなのですが、ボーンをいちいち初期化してあげないと、動かした行列に対して、さらに行列が乗算するためにドゥンドゥン明後日の方向に飛んでいくわけです。つまりやらなければならないことは、ボーンを適用する前に初期化することです。

え?すべてのボーンの初期化の方法が分からぬって?そんなもん自分で考えるボケが!!
std::fill(_bonematrixes.begin(), _bonematrixes.end(), XMMatrixIdentity());

ああ?これ?これはちょっとしたコピペミスだよ。STLのアルゴリズムを使ったfillだよ知ってるだろ?beginからendまでを第三引数の値で埋めるのさ。

と、まあ茶番はともかく、そこまでやればたしかに時間によって動いてはくれますが、これだと一瞬しかそのポーズをとってくれないんですね。

適切なフレームで適切なポーズをとるには

通常、アニメーションにおいては、特に「元に戻れ」という指定がない限りもとに戻ってほしくないわけ。

じゃあ特定のフレームの時にはどうするべきかというと、最後に表示してたポーズ状態を維持しておけばいいわけ。

ところで「最後に表示してたポーズ状態」ってなんだろう?

| | 15 | 20 | 25 | 30 | 35 |
|----------|----|----|----|----|----|
| 全ての顔 | | | | | |
| 表示・IK・外観 | | | | | |
| +表情 | | | | | |
| +センター | | | | | |
| +体(上) | | | | | |
| +腕 | | | | | |
| +指 | | | | | |
| +体(下) | | | | | |

例えば35フレームの時にはこの30フレームにある「キーフレーム」を参照します。さつきみたいに合致するフレームだったら簡単だけど、今回みたいに「35未満のうち一番でかい要素」を取ってきたい場合はどうしたらいいんだろうか?

先ほどのfind_ifを覚えていませんか?覚えてない人は寝てたか健忘症を疑った方がいいでしょうね。

さて、このfind_ifの条件を変えればよくなじですか?

```
auto it=std::find_if(a.second.begin(), a.second.end(), [frameno](const Pose& p) {return p.frameNo <= frameno; });
と思うじゃん?
```



残念。find_ifは先頭から探索するので、どうやっても0番フレームが引つかっちゃいます。

ではどうすべきか?前期に使った「リバースイテレータ」を思い出しましょ



さあ記憶をリバースするんだ

そう、反対側から探索するイテレータでした。これを使用するのだ。使い方はいたって簡単。`begin`と`end`を`rbegin`と`rend`にするだけ。たったそれだけ!!!

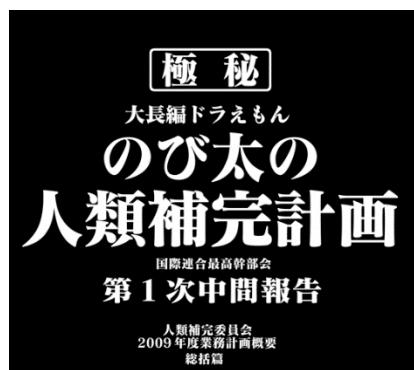
さあ、やってみましょ

腕がピコピコと動けば成功です。

「いや、俺がやりたいのはそうじゃなくて、もうちょっとアニメーションしてるといふがなんといふか」

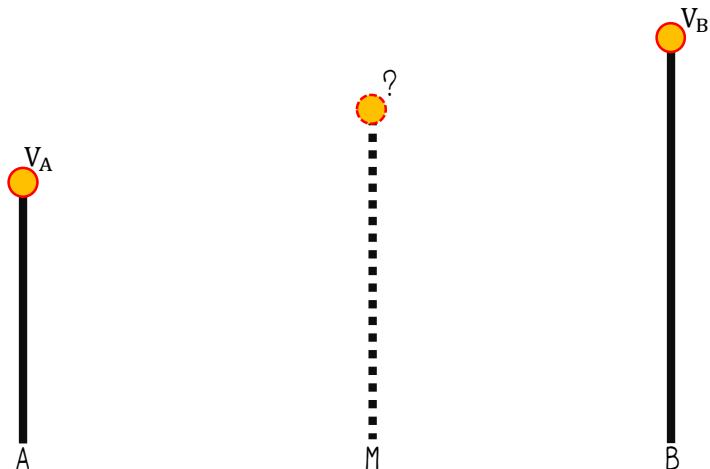
ごもっとも。そうだね。ここからは補間を楽しもうか

ポーズを補間してアニメーションしよう



字が違います

間を補うほうの補間です。英語で言うと *interpolation*。国際警察みたいな名前ですが、補間という意味です。



例えば、図のように A 地点と B 地点があったとします。そしてそれぞれの「値」が V_A および V_B だったとします。

ここで知りたいのは A と B の間に M という地点があった場合、M の「値」はどのように考えればよいのかという事です。数学的に。

もし M がちょうど A と B の中間地点にあったとしたらどうでしょう？中間地点にあればそれは小学生でもだいたいわかります。

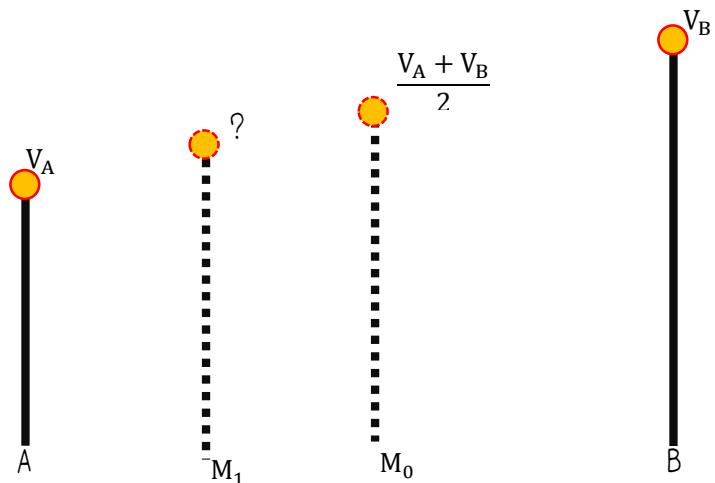
「足して 2 で割る」

です。

えーと、これくらい分からぬ」とちょっと賢い小学 3 年生に負けますよ？どうせ小学生とか言うと君らエロい事しか考えてないんだろう？



分からぬ人は小学生に戻ったつもりで考えましょう。では、もう少し考えを進めます。



さて、先ほど求めた M と A のちょうど中間地点に M₁ があったとするとその値は？当然そこも足して 2 で割るため

$$M_1 = \frac{V_A + \frac{V_A + V_B}{2}}{2}$$

となりますよね？これは整理すると

$$M_1 = \frac{3V_A + V_B}{4}$$

となります。ピンときませんかね？もし M₀ と B の間を M₂ とすると今度は

$$M_2 = \frac{V_A + 3V_B}{4}$$

となりますね？

まだピンとこない人はもっともっと分割してみましょう。必ず自分の頭で考えて、そして感じてください。

$$\frac{7V_A + V_B}{8}, \frac{6V_A + 2V_B}{8}, \frac{5V_A + 3V_B}{8}, \frac{4V_A + 4V_B}{8}, \frac{3V_A + 5V_B}{8}, \frac{2V_A + 6V_B}{8}, \frac{V_A + 7V_B}{8}$$

ハア…ハア…どうよ!!

流石にもうお分かりですね? AとBの間の「値」は、Aの何分の1かとBの何分の1かを足したものになっていて、もつと言うと、Aの係数とBの係数を足すと1になるようになっていますよね?

更に、Aに近ければより A の値が大きく反映され、B の値の影響度は小さくなります。これを一般化した式にすると

$$V_t = (1 - t)V_A + tV_B$$

となります。 $t+(1-t)=1$ ですし、例えば t に 0.5 を入れれば分かるように、ちょうど中間の値になります。そして、 $t=0$ のときは v_A となり、 $t=1$ のときは v_B となります。

この十の値ですが

$$t = \frac{M_t - A}{B - A}$$

とすることで計算できます。分かります？ $M_t = A$ のときは $t=0$ となりますし $M_t = B$ のときは 1 になります。さらに中間地点である $M_t = \frac{A+B}{2}$ の時は $t=0.5$ になりますよね？

ということで、この \dagger を用いて線形補間をやってみましょう。ちょっとネタバレしつくと、線形補間は覚えておいてほしいのですが、今回の場合は線形補間がちょっとだけ不適切ではあります。でもなぜ不適切なのは見ない?と分からぬるので、このまま進んでいきましょう。

ひとまず一つの単純なアイディアとして

最終行列 = 変換行列 $A^*t +$ 変換行列 $B^*(1-t)$

という計算をしてみましょう。ループ1回を1フレームとして考えましょう。となると、ポーズのための行列がAとBの2つ必要というわけですので、プログラミング的にはそこが面倒な部分ですね。

Aの方は既に計算している部分だからいいとして、問題はBの部分をどうしようかという話ですね。

さて、どうしましよう？

revit++

ですか？

そうは行かないんですね。こいつはリバースイテレータ…ですから++すると逆方向に向かうので、「次」ではなく「前」になります。

じゃあ、こう？

revit--

まあ、悪くないね。でも…revit==rbegin()である場合にはエラーが起きます。

if(revit!=rbegin()){

 revit--

}

でもいいんですけど…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it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ね。これに関しては「イテレータ」に関する理解がもう一段階進まないとピンとこないと思いますので、今は簡単に「頭を刺してお尻を刺してお尻を刺してお尻を刺すイテレータか」の違いだとなんとなく思っておいてください。

ともかく baseを使えばいいのですが、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。

でももし rit がOKで base()がend()になっている場合ならば rit 100%で構いません。つまり

```
auto it = rit.base();
if (it == a.second.end()) {
    BendBoneMatrices(_boneMap[a.first], _bonenodes, _bones, _bonematrixes, XMMatrixRotationQuaternion(rit->quaternion));
}
else {
    float t = static_cast<float>(frameNo - rit->frameNo) / static_cast<float>(it->frameNo - rit->frameNo);
```

```

        BendBoneMatrices(_boneMap[a.first],
        _bonenodes,
        _bones,
        _bonematrixes,
        XMMatrixRotationQuaternion(it->quaternion)*t+ XMMatrixRotationQuaternion(rit->quaternion)*(1.0f-
        t));
    }
}

```

てな感じですね。やろうとしている事はわかりますかね?
でも、これでやろうとすると実はおかしなことになります。

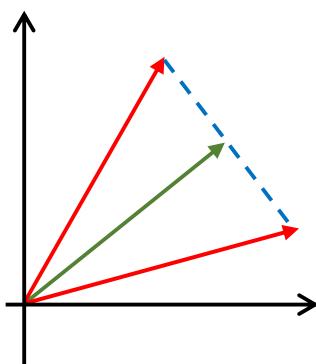


腕の長さおかしくね?

これが…線形補間の限界…!!

腕は現実的には「長さが変わりません」

で、線形補間というのは、本当に線形補間なのです。2つの点の間を直線で結んでその間に+として補間するものです。つまり…



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?
これはちょいと特殊な補間をしてあげないといけないのだ。

で、球面線形補間っていうのを使うんだけど、これは簡単に言うとさっきの線形補間を角度の補間として補間してあげるものなのだ。

簡単にといふか、君らの知ってる範囲でいふとやね

$$A(x_A, y_A) \rightarrow B(x_B, y_B)$$
$$M = A(1-t) + Bt = (x_A(1-t) + x_B t, y_A(1-t) + y_B t)$$

これ↑が通常の線形補間で、球面線形補間ってのは

$$A(r_A, \theta_A) \rightarrow B(r_B, \theta_B)$$
$$M = A(1-t) + Bt = (r_A(1-t) + r_B t, \theta_A(1-t) + \theta_B t)$$

といふわけです。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ふイメージね。

まあ、面倒くさそうだ。だがまた都合がいいのが「ウォータニオン」ってのが「極座標表現」みたいになもんなのでそれを利用すれば球面線形補間となる。関数名はXMQuaternionSlerpという関数を使います。

https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk quaternion.xmquaternionslerp.aspx

こいつは2つのウォータニオンベクトルを指定して、最後に t を指定することで、球面線形補間をやってくれます。

これは補間したベクトルを返しますので、

XMMatrixRotationQuaternion(XMQuaternionSlerp(rit->quaternion, it->quaternion, t))
とします。

ここまでがうまくいけば、腕が短くなることなく腕を振ることができていると思います。

ここでほかのいろいろなモーションもうまくいくかどうかを検証してみましょう。
charge.vmdを再生してみてください。これはチャージマン研!の変身シーンのポーズなのが



妙なところでアニメーションが止まると思ひます。

これには罠があるのです。VMDの罠がね!!!

VMD ファイルの罠に陥る

VMD ファイルにはちょっとした罠が仕掛けられている。大したことじゃないんだが、いや…重大な罠だな。これを放置するとチャージマン研の変身シーン以上にカッコ悪くなる。

それはだな…VMD のキーフレーム情報は「登録」した順に並んでいるのであって、決してフレーム番号ごとに並んでいるわけではないということだ。

つまり、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 フレーム時点での挙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://cpprefjp.github.io/reference/algorithm/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をするの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(), [] (MotionData& a, MotionData& 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。

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。

ベジエとニュートン(ニュートン・ラフソン)法と私

はい、この授業もかなりマニアックなところまでやってきました。

正直しんどいと思います。俺もしんどい。そう、普通に補間するだけで十分だろ？お前は良くやったよ…そんな声が聞こえるようだが、俺はプログラミングキチガイ



みんなが嫌がることをやるのが俺なのさ!!

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 VMD の説明では

//// 補間の補足 ////

BYTE Interpolation(b4); // (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(クオータニオン)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、

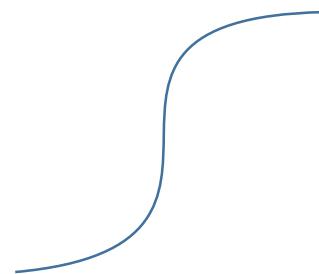
```
// Xax,Yax,Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,
// Yax,Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,
// Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,00,
// Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,00,00
```

こんな感じで書いています。正直面倒くさいやで。

さて、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し、何をしたいのかさっぱりわからぬ。`unsigned char*b4` 個のデータなんだが、どう見ても `Xay` 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と思うんだが…。

…まあ言っても仕方ないか。

と、それは置いておいて、最大の悩みはこれだ。



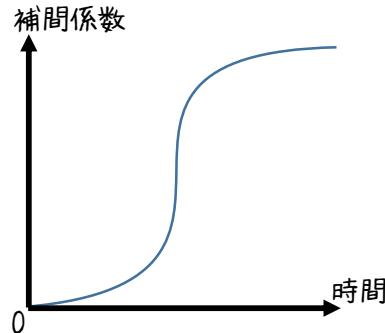
はい、これはパラメトリック曲線の一種であるため

$$y = y(t)$$

$$x = x(t)$$

という具合に、それぞれ t によって値が決まる関数の組み合わせなのである。これを $y=f(x)$ の形にしたい。

何故ならば図のように、補間係数が時間の関数になっているからである。…というか縦軸が横



軸の関数にならなければならぬのだ。実用として。

数式で書くと

$$t = t(x)$$

が必要なのだと…これさえ求めれば

$$y = y(t(x))$$

とすることができる。ところでこの場合の $t(x)$ とはなんだろうか?

所謂「逆関数」である。高校でやったことがある人もいると思うが

$y = f(x)$ ならば

$x = f^{-1}(y)$ が存在するというわけだ。もうわからんねえな、これ。

ところでベジエ曲線というのは三次の場合

$$x = x(t) = t^3 x_4 + 3t^2(1-t)x_3 + 3t(1-t)^2x_2 + (1-t)^3x_1$$

および

$$y = y(t) = t^3 y_4 + 3t^2(1-t)y_3 + 3t(1-t)^2y_2 + (1-t)^3y_1$$

だが、こういった三次式の逆関数を求めるのは困難を極めるのだ。

<https://ja.wikipedia.org/wiki/%E4%B8%89%E6%AC%A1%E6%96%B9%E7%A8%8B%E5%BC%8F#.E3.82.AB.E3.83.AB.E3.83.80.E3.83.8E.E3.81.AE.E5.85.AC.E5.BC.8F>

逆関数を求める

• 3次方程式の解

$$x = 3t^2 - 2t^3$$

```
rules = Solve[3 t^2 - 2 t^3 == x, t]
{{t → 1/2 (1 - 1/(1 + 2 x + 2 Sqrt[-x + x^2])^(1/3) - (-1 + 2 x + 2 Sqrt[-x + x^2])^(1/3)), 
t → 1/2 + (1 + I Sqrt[3])/4 (1 - I Sqrt[3]) (1/(1 + 2 x + 2 Sqrt[-x + x^2])^(1/3)), 
t → 1/2 + (1 - I Sqrt[3])/4 (1 + I Sqrt[3]) (1/(1 + 2 x + 2 Sqrt[-x + x^2])^(1/3))}}
```

今回の場合はもうすこしだけ簡単することは可能である。何故かと言うと $x_1=0$ で、
 $x_4=\max$ であることが分かっているからだ。少なくとも x_1 の項は消えるんやで。

…ところが結局三次の項が残っている以上はろくでもない計算が残っているのだなあ。

でも、MMD の作者は何とかしているのだ。数学の専門家でもないはずだから、なんか抜け道はあるはず…あるはず。

ということで、

<https://ja.wikipedia.org/wiki/%E3%83%8B%E3%83%A5%E3%83%BC%E3%83%88%E3%83%B3%E6%B3%95>
<http://qiita.com/PlanetMeron/items/09d7eb204868e1a49f49>

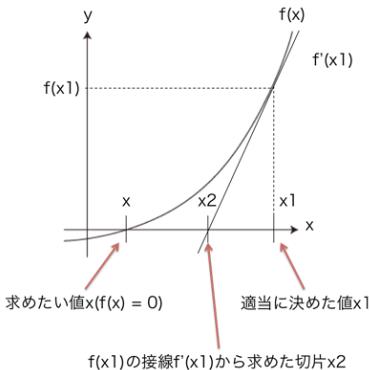
<http://ringsbell.blog117.fc2.com/blog-entry-803.html>

ニュートン・ラフソン法って奴を見つけました。

まず、この方法の目的は

$$t(x) = 0$$

となる x を見つけることとすると



微分法によって、求めたい答え(x)に近づけていくというものです。試行回数を増やせば限りなく x に近づきます。

さて、目的は先ほど

$$x = t^3x_4 + 3t^2(1-t)x_3 + 3t(1-t)^2x_2 + (1-t)^3x_1$$

の式において、 x は分かっており、知りたいのは t であり、その t を用いて

$$y = t^3y_4 + 3t^2(1-t)y_3 + 3t(1-t)^2y_2 + (1-t)^3y_1$$

を求めるのである。つまり、↑における x の式からニュートンラフソン法を用いて t を算出すれば、後は大したことではないのである。

で、ニュートンラフソン法ってのはこうやって解きます。

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

何これ分からん？



漸化式い…ですかねえ。

まあ、言うほど難しくはないのでちょっとだけ我慢して聞いてください。今回の話は「微分」が入ってきますのでちょっとそこだけが難しいかもしれません。

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つて、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、

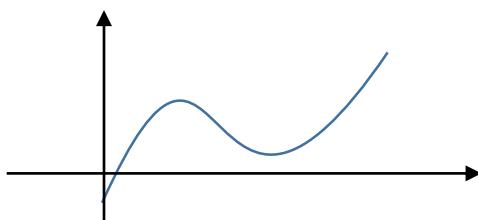
x_0 は定数ですので、傾きも定数で算出されるというわけです。

えーと、大丈夫? ついてきてますか? なんでこうなるかは時間があつたら解説します。独学で
やりたい人は

[¥¥132sv¥gakuseigamero¥rkawano¥数学¥微分積分.pdf](#)

でも見ておいてください。ともかく特定の座標における傾きを出すために使うのが微分です。

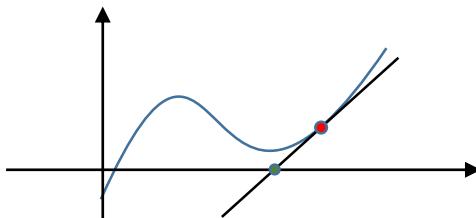
で、ニュートンラフソン法といいのは、これを用いて、方程式の解を近似するものですが、



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。で、三次曲線以上って解を求め
るのが結構大変なんですよね。という事で、コンピュータを使うときは近似を用います。

そのための近似法のひとつとして、先ほど紹介したニュートンラフソン法ってのがあるわけ
です。やり方としては、

- ①曲線上の適当な点を打ちます。
 - ②その点の傾き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



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0になるところなので

$$0 - f(x_0) = f'(x_0)(x - x_0)$$

という式が成り立ちます。これをxの式にします。

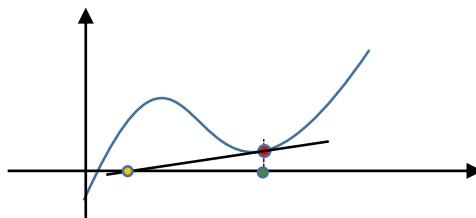
$$-\frac{f(x_0)}{f'(x_0)} = x - x_0$$

移項すると

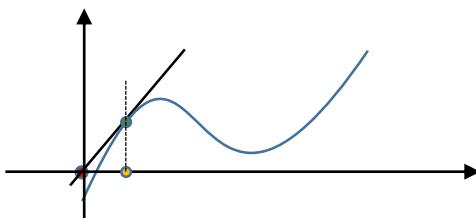
あと

となりX座標が求まります。

そこからさらにそのX座標を求めます。



で、そのX座標における接線を求めます。



と、まあこれを繰り返していくうちに解に近づいていくというわけです。

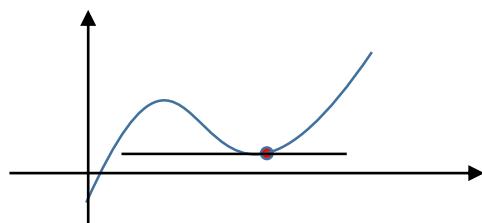
ともかく、これをさっさと数式にしたいところだが、その前にまずやることがある。それは式を簡単にできるという事だ。



を見れば分かる通り「端点」は固定(はじめは0で最後は1)である。つまり、先ほどの式は

$$x = t^3 + 3t^2(1-t)x_3 + 3t(1-t)^2x_2$$

と書くことができる。もう一つある特徴は、MMDの傾きがマイナスになることはないのです。これもし傾きがマイナスになると…というか、プラスとマイナスが混在しているとかなり面倒なんですよね。要は



接線の傾きがこんな感じになる箇所がある場合、答えに全然近づかないですよね？

ただ、プラスとマイナスが混在しなくとも〇になる箇所はあります。↑の例の中間地点ですね？この時は0になった部分で計算を打ち切ってOKです。なぜって+によってxが変わらないから問題ないのです。

ともかくこの方式によって求めたい+の近似値を得ることができます。もちろん何回も繰り返さなければならぬので、そこはループを用います。

ところで

$$x = t^3 + 3t^2(1-t)x_3 + 3t(1-t)^2x_2$$

を移行すると

$$0 = t^3 + 3t^2(1-t)x_3 + 3t(1-t)^2x_2 - x$$

であり、この式が0に近づくような+を求めればいいわけです。あと、初期値は $t_0=x$ でいいと思います。

つまり

$$t_0 = x$$

であり

$$f(t) = t^3(1 + 3x_2 - 3x_3) + t^2(3x_3 - 6x_2) + t(3x_2) - x$$

であるから

$$f'(t) = 3t^2(1 + 3x_2 - 3x_3) + 2t(3x_3 - 6x_2) + 3x_2$$

である。

もちろん、x₃やらx₂やらx₁は定数なので、プログラムが求めてくれる。

また、

$$f(t_n) = 0$$

になった場合は計算を打ち切ってもいい。解が求まっているからだ。

さらに

$$f'(t_n) = 0$$

になった場合も計算を打ち切ろう。何故なら 0 除算になるからだ。

あとはこの式から

$$t_{n+1} = t_n - \frac{f(t_n)}{f'(t_n)}$$

でループしてあげればいい。

ニュートン法の実装

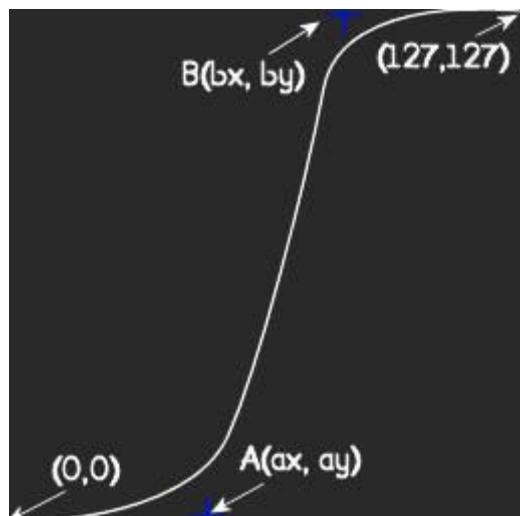
まずはこの方法をとる前に、ベジエのデータがどうなっているのか見てみよう…。

http://blog.goo.ne.jp/torisu_tetosuki/e/bc9f1c4d597341b394bd02bb4597499d

BYTE Interpolation[24]; // おそらく[6][4](未検証) // 補完

ぐぬぬ…未検証かよ。辛いなあ。

なんかこんな図が書いてある…



ん？じゃあ、必要なのは A(ax,ay) と B(bx,by) だけでよくね？何でこんなつくりにしてんの…
まあ～インパース kinematics 用やろなあ。

未検証ってのも怖いしなあ…。まあ、男は度胸。なんでもやってみるのさ。

// 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**, Yax,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

で、FK(フォワードキネマティクス)のときに使用するのは↑の赤い部分だけです…無駄ですか…まあ、後々 IK をやる時に役に立つかもなんで我慢しちましょ。

さて、番号的には 3,7,11,15 なので、

```
ax = interpolation(3)
ay = interpolation(7)
bx = interpolation(11)
by = interpolation(15)
```

これは…あまりにもマジックナンバーすぎるでしょう…('ω; `)

まあ仕方あるまいよ。

(0,0), A(ax,ay), B(bx,by), (127,127) の 3 次(4 点)ベジエ

のことなので、

ax/=127.0f

```
ay/=127.0f;  
bx/=127.0f;  
by/=127.0f;
```

とします。これによりそれぞれの範囲が0~1となります。

さて、ここから何とかかんとか近似解を求める関数を作りましょう。とりあえず先ほどの式の通りに関数を作った結果こうなりました。

```
///X値からBezierのY値を(ニュートン法を用いて)返す  
float GetBezierYValueFromXWithNewton(float x,const XMFLOAT2& a,const XMFLOAT2& b) {  
    if (a.x == a.y&&b.x == b.y) return x;//直線になってるのでx=yである  
    float t = x;//初期値はxと同じでいい  
    float k0 = 1 + 3 * a.x-3*b.x;  
    float k1 = 3*b.x-b*a.x;  
    float k2 = 3*a.x;  
    for (int i = 0; i < 16; ++i) {  
        float r = (1 - t);  
        float kt = r*r*r * 0 + 3 * r*r*t*a.x + 3 * r*t*t*b.x + t*t*t * 1-x;  
        float ft = (t*t*t)*k0 + t*t*k1 + t*k2 - x;  
        if (ft <= 0.0005&&ft >= -0.0005) break;//適当なところで計算打ち切り  
        float fdt = (3 * t*t*k0 + 2 * t*k1 + k2);  
        if (fdt == 0) break;//0除算防止  
        t = t - ft / fdt;  
    }  
    float r = (1 - t);  
    return 3*r*r*t*a.y+3*r*t*t*b.y+t*t*t;
```

まずこの関数を作った。あとは補間/パラメータを適用していくわけだが読み込んだものを持っておく必要があるため構造体を変更します。

```
struct Pose {  
    Pose(unsigned int f, const XMVECTOR& q,BYTE ax, BYTE ay, BYTE bx, BYTE  
by) :frameNo(f), quaternion(q) {  
        bz1 = XMFLOAT2((float)ax/127.f, (float)ay / 127.f );  
        bz2 = XMFLOAT2((float)bx / 127.f, (float)by / 127.f );  
    }
```

```

    unsigned int frameNo;
    XMVECTOR quaternion;
    XMFLOAT2 bz1;//ベジエ要素①
    XMFLOAT2 bz2;//ベジエ要素②
};


```

で、これを読み込みつつ代入

```

for (auto& pose : poses) {
    animation[pose.boneName].push_back(Pose(pose.frameNo,
        XMLoadFloat4(&pose.quaternion),
        pose.interpolation(3),
        pose.interpolation(7),
        pose.interpolation(11),
        pose.interpolation(15)));
}

```

あとは適用

```
t=GetBezierYValueFromXWithNewton(t, it->bz1, it->bz2);
```

さて、これで理論上ベジエ補間ができるわけですが…ま～だ微妙に違うんですよね。細かい検証はともかく、ちょっとやってみましょう。

ニュートン法実装後の課題

とりあえずニュートン法を用いて求めたい値を計算してみたわけだが、やっぱり本家と微妙に様子が違う…。色々と悩みまくってみたけど、ここはひとまず DxLib のソースコードを見てみよう…。

```

if( Linear )
{
    RateH = Rate ;
}
else
{
    int l ;
    float T, InvT ;

```

```

for( l = 0, T = Rate, InvT = 1.0f - T ; l < 32 ; l ++ )
{
    float TempX = InvT * InvT * T * X1 + InvT * T * T * X2 + T * T * T -
Rate ;
    if( TempX < 0.0001f && TempX > -0.0001f ) break ;
    T -= TempX * 0.5f ;
    InvT = 1.0f - T ;
}
RateH = InvT * InvT * T * Y1 + InvT * T * T * Y2 + T * T * T ;
}

```

う～ん。発想は同じだとは思うんだよね…。

ただ、式が微妙に違う…。特に

$T -= TempX * 0.5f ;$

なんだこれ？さらに近似かな？ちょっと裏が取れてないのまだ飛びつくのは早い…。

似た数式は

http://d.hatena.ne.jp/babu_babu_baboo/20120618/1340004207

にもあって、

<http://umeru.hatenablog.com/entry/2015/12/03/131844>

を見ると

「ニュートン法」「二分法」というキーワードが見える。

二分法で検索すると…

<http://c5h12pentan.blog.fc2.com/blog-entry-3.html>

とか

さらにはベジェクリッピングなどというのも出てきた。

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

とか

また勉強しなきゃいけんところが増えました。

二分法

さて、二分法ですが、原理自体は非常にシンプルで

<https://ja.wikipedia.org/wiki/%E4%BA%8C%E5%88%86%E6%B3%95>

によれば

例えば $f(x)=0$ となる点を探したいとする。

まず

1. $f(x_1)$ と $f(x_2)$ で符号が異なるような 2 点を探す。

2. x_1 と x_2 の中間地点である x_m を定め $f(x_m)$ を計算する。 $(x_m = \frac{x_1+x_2}{2})$

3. x_1 と x_2 のうち $f(x_m)$ と符号が同じ方を x_m に入れ替える(例えば $f(x_1)$ と $f(x_m)$ の符号が同じなら $x_1 = x_m$ とする。)

4. 1に戻る

というアルゴリズムになっています。

シンプルなので、直感的にも分かりやすいかなと思います。ただし Wikipedia に気になる記述があつて、それは「収束が遅い」ということです。

で、先ほどの DxLib のソースコードについてですが、

```
for( I = 0, T = Rate, InvT = 1.0f - T ; I < 32 ; I ++ )  
{  
    float TempX = InvT * InvT * T * X1 + InvT * T * T * X2 + T * T * T - Rate ;  
    if( TempX < 0.0001f && TempX > -0.0001f ) break ;  
    T -= TempX * 0.5f ;  
    InvT = 1.0f - T ;  
}
```

あつ…

これは二分法っぽい…何でか? というと 0.5 をかけている。つまり 2 で割っている。でも、中間点じゃないよね? 符号の比較もしないよね? 一体どういう事だろう…

$T = T - TempX * 0.5$

このコードね…。

上の式を数式にすると

$$f(t) = r^2 t + rt^2 + t^3 - x$$

であり、

$$t_{n+1} = t_n - \frac{r_n^2 t_n + r_n t_n^2 + t_n^3 - x}{2}$$

である。

なかなか強敵。

とりあえず二分法なのに符号の判定をしていないのは何故なんだろう…？

というわけで恐らく「単純な2分法ではない」と考えられる。どう解釈しても二分法ではないのだ。

ここで

$$f(t_n) = r_n^2 t_n + r_n t_n^2 + t_n^3 - x$$

の式について考えてみる。これ、例えば適当な場所…例えば $t=0.5$ だとする。もし、0.5で正解ならば $r_n^2 t_n + r_n t_n^2 + t_n^3$ は x と同じ数になるため、 $f(t_n) = 0$ となる。

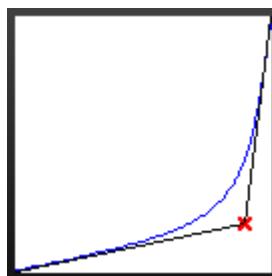
ここで

$$g(t_n) = r_n^2 t_n + r_n t_n^2 + t_n^3$$

と定義する。 $f(t)=g(t)-x$ である。

ちなみに、ベジエが曲がっていなければ必ず一発目で $t=x=g(t)$ となるはずである。

ここで例えば後半戦から上がっていくパターン



こういうのを考えてみる

この場合だと最初 $x=0.7$ のとき、初期値として $t=x$ を代入すれば当然ながら

$$r_n^2 t_n + r_n t_n^2 + t_n^3 < 0.7$$

となるため、 $f(t)<0$ となる。もうちょっと t を増やさないと x には近づかないわけだ。ちなみに求めたいたが何処にあるのかは分からぬんだけど、

$$r_n^2 t_n + r_n t_n^2 + t_n^3 = x$$

となる t が知りたいだけである。だからこの DxLib の思想的には「 $g(t)=x$ 」じゃなければ、近づくように足したり引いたりするよってな感じで実際の値との差分として使用している。差分

がマイナスなら $+$ はプラスすべきで、差分がプラスなら $+$ はマイナスすべきなので、元の $+$ からこの差分を引いている。

で、増やしたり減らしたりする個所に $f(x)$ の半分を表す 0.5 を乗算しているのは二分法の考え方だろう。符号の役割は「差分がプラスかマイナスか」で変化するため二分法と同じことになる。

さて…仕組みが納得いったところで自分のプログラムを修正してみよう…。とりあえず関数の名前は変更する。

あ、あと 3 が乗算されてない理由ですが、これ、interpolation 読み込む時点で 3 がかけられていましたので、まあ…そういうことです。

で、この改良版二分法でもチェックしましたが、結果として言うと、同じでした。全く同じでした。まあ、それほどおかしいわけでもないからいいか…ちょっと気になる程度なので、今はちょっと置いておきます。

ちょっと細かいところ修正

白飛びの対処

今、なんか表示がアレで、テクスチャ部分の見え方がおかしなことになってるんですよね…白くくすむ…というか…。



なんか、環境光を適用したところからこんな感じになってしまいました。

どうも PMDにおいては、環境光成分をただ足すだけではダメなようです。この辺はわりとルールがまちまちなので、MMD の出力を見ながら「それっぽく」していくしかないようです。

ということで、テクスチャに対して環境光を適用しないようにすると…



きれいですが、微妙に不自然です。モデルを変えると…



流石に鮮やかすぎて浮いています。なのでちょっと面倒ですが、こう書きました。

```
float3 color = existTex ? tex.Sample(smp, data.uv)*(dbright+ambient) :  
diffuse*dbright+ambient;
```

ちょっと無理やりっぽいですが…。



わりと鮮やかに出力しつつも、影もきちんと入っていると思います。

もしくは

```
float3 color = existTex ? tex.Sample(smp, data.uv)*max(dbright,ambient) :  
diffuse*dbright+ambient;
```

こうすると、



まあ割とそれっぽく出ます。

何か透けてる

↑の響ちゃんを見ると分かると思いますが、ブラが透けています。透けブラです。いやらしいです。このままでもいいんですが、僕はこのままでもいいと思うんですが、本来は透けるはずではないので、ブラを見えないようにします。

問題は今は描画がアルファに対応していないからです。ちなみに後ろから見るとパンツも透けています。いいです。

原因は

| | |
|--|---|
| E71material[20].face_vert_count | 00010000 |
| E7FBmaterial[20].texture_file_name[0] | 68 69 62 69 6B 69 5F 74 65 63 35 2E 6A 70 67 00 hibiki_tec5.jpg |
| E80Bmaterial[20].texture_file_name[16] | FD FD FD FD |
| E80Fmaterial[21].diffuse_color[0] | 3F000000 3F000000 3F000000 |
| E81Bmaterial[21].alpha | 00000000 |
| E81Fmaterial[21].specularity | 40A00000 |
| E823material[21].specular_color[0] | 00000000 00000000 00000000 |
| E82Fmaterial[21].mirror_color[0] | 3F333333 3F333333 3F333333 |
| E83Bmaterial[21].toon_index | 00 |
| E83Cmaterial[21].edge_flag | 00 |
| E83Dmaterial[21].face_vert_count | 00003588 |
| E841material[21].texture_file_name[0] | 68 69 62 69 6B 69 5F 74 65 63 34 2E 6A 70 67 00 hibiki_tec4.jpg |

ここです。ブラのマテリアルを見てください。

アルファが0です。まず、手取り早い方法を使います。

[https://msdn.microsoft.com/ja-jp/library/bb943995\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb943995(v=vs.85).aspx)

discardってのがあります。これは関数でも何でもなく「ステートメント」です。

ページを見れば分かる通り、break,continue,discard,do,for,if,switch,whileの一種です。で、説明はたった一言

現在のピクセルの結果を出力しません

そういうことです。いや、流石にこの説明はひどすぎるでしょう…手を抜くのもいい加減にしろ!!!

英語だ英語!!!

[https://msdn.microsoft.com/en-us/library/windows/desktop/bb943995\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb943995(v=vs.85).aspx)

"Do not output the result of the current pixel."

このやろー!!!

…まあ仕方ないですね。

では僕の理解をお話ししたしましょう。

ピクセルシェーダの結果をなかったことにする

これがdiscardです。

もつと言ふと、そもそもピクセルシェーダに入つてなかつたことにする。描画対象になつていなければ、結果的にピクセルは塗りつぶされないし、値も書き込まれません。

というわけで

```
if(alpha==0) discard;
```

とすると最初から出力されません。こいつを利用すると色々と面白い事ができるようになります。

ますが、それはもうちょっと後で。そしてこれで出力すると



そういうことだ

ちょっと横道に逸れてみる

ちょっと皆さん実装でお悩みのようなので、時間の猶予を与えましょう。ということで、横道に逸れてみましょう。ここからしばらくは余裕のある人だけ聞いてください。

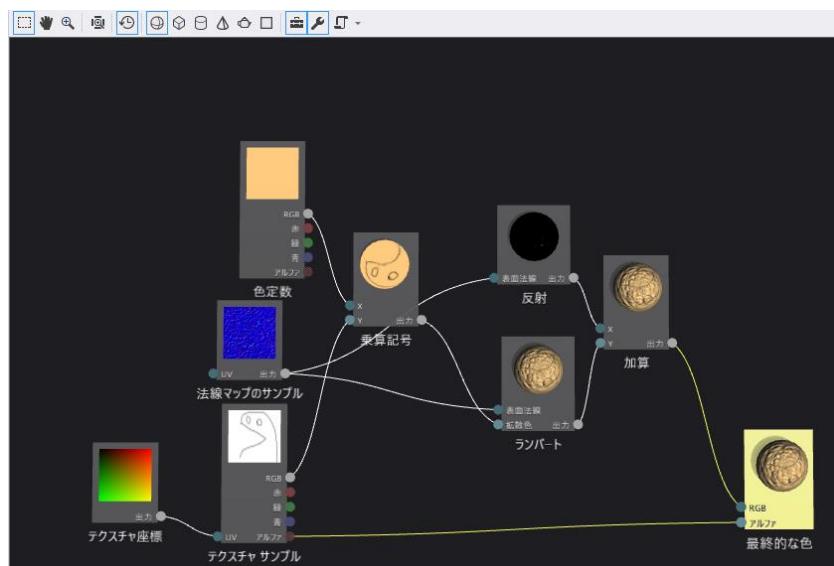
シェーダデザイナ

VisualStudio にはシェーダデザイナという機能がついています。

ファイル→新規作成→グラフィックス→「視覚シェーダ・グラフ」

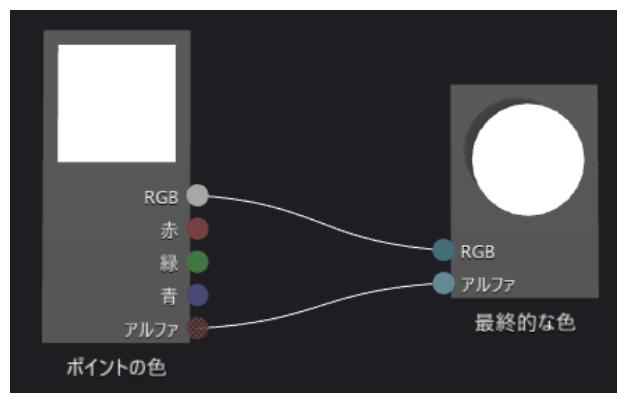
を選択すると、シェーダのファイルをグラフィカルに作ることができるツールが立ち上がる。

UnrealEngine4のマテリアルエディタみたいなやつですね。



こんなやつ

モチロン最初は何もない。



マジなにもない

例えばここにランダムシェーディングをしたいとする。

まず「ポイントの色」は不要なので、deleteで削除してください
左側にツールレーダーがあるので、そこから「ランパート」というのを選択して D&D してください。

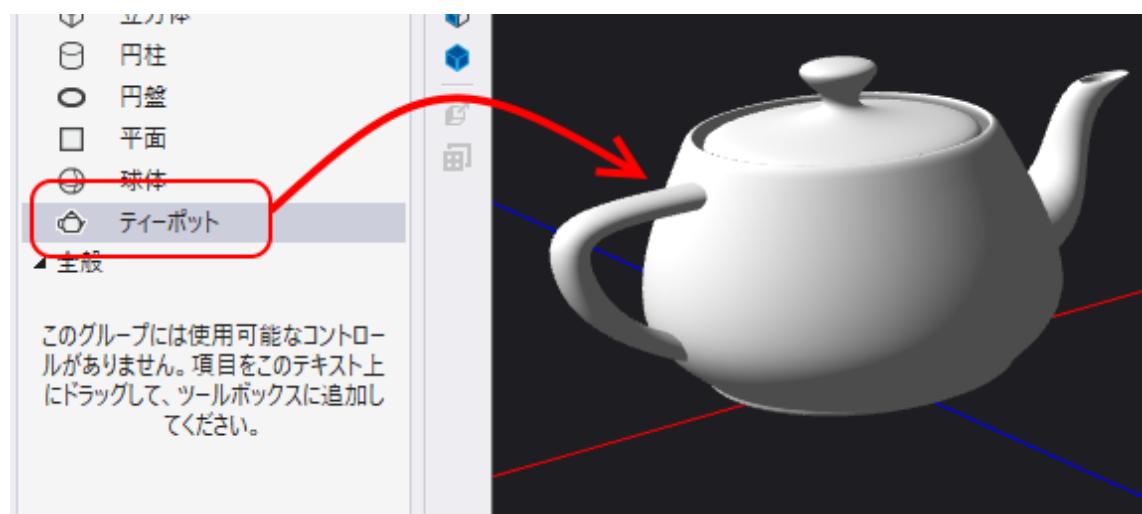


で、ランパートの「出力」を「最終的な色」の RGB につないでください。

上図のようになると思います。ここで一旦保存しておいてください(dgsi)

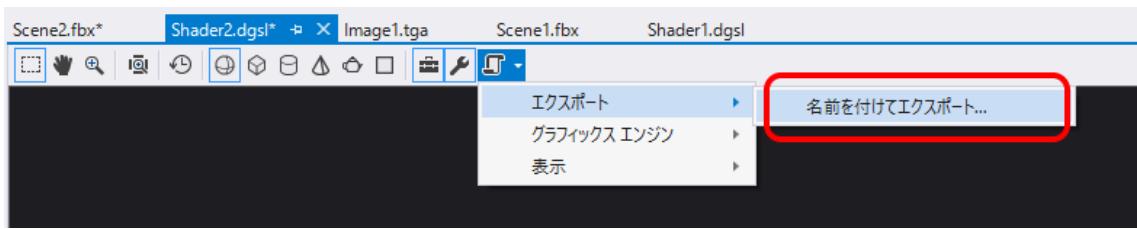
で、これを FBX に反映させたいので、また、新規作成→ファイル→グラフィックス→3Dシーン(fbx)としてください。

真っ暗な画面になると思いますが、例えばティーポットを D&D するとティーポットが出てきます。



で、このプロパティの中に「効果」ってあると思います。で、この中にファイル指定があると思いますので、先ほど作った dgsi ファイルを指定してください。

そうすると、ちょっと見え方が変わったんじゃないかなと思います。で、これはありがたいことに hlsi に出力もできます。



メニューになんかエクスポートってあるので、これに「名前を付けてエクスポート」をやります。で、hlslで出力すると

```
P2F main(V2P pixel)
{
    P2F result;

    // we need to normalize incoming vectors
    float3 surfaceNormal = normalize(pixel.normal);
    float3 surfaceTangent = normalize(pixel.tangent.xyz);
    float3 worldNormal = normalize(pixel.worldNorm);
    float3 toEyeVector = normalize(pixel.toEye);

    // construct tangent matrix
    float3x3 localToTangent = transpose(float3x3(surfaceTangent, cross(surfaceNormal, surfaceTangent) *
pixel.tangent.w, surfaceNormal));
    float3x3 worldToTangent = mul((float3x3)WorldToLocal4x4, localToTangent);

    // transform some vectors into tangent space
    float3 tangentLightDir = normalize(mul(LightDirection[0], worldToTangent));
    float3 tangentToEyeVec = normalize(mul(toEyeVector, worldToTangent));

    // BEGIN GENERATED CODE
    float3 local0 = LambertLighting(tangentLightDir, float3(0.00000f, 0.00000f, 1.00000f),
MaterialAmbient.rgb, AmbientLight.rgb, LightColor[0].rgb, float3(1.00000f, 1.00000f, 1.00000f));
    result.fragment = CombineRGBWithAlpha(local0, 1.00000f);
    // END GENERATED CODE

    if (result.fragment.a == 0.0f) discard;

    return result;
}
```

および

```
float3 LambertLighting(
    float3 lightNormal,
    float3 surfaceNormal,
    float3 materialAmbient,
    float3 lightAmbient,
    float3 lightColor,
    float3 pixelColor
)
{
    // compute amount of contribution per light
    float diffuseAmount = saturate(dot(lightNormal, surfaceNormal));
    float3 diffuse = diffuseAmount * lightColor * pixelColor;

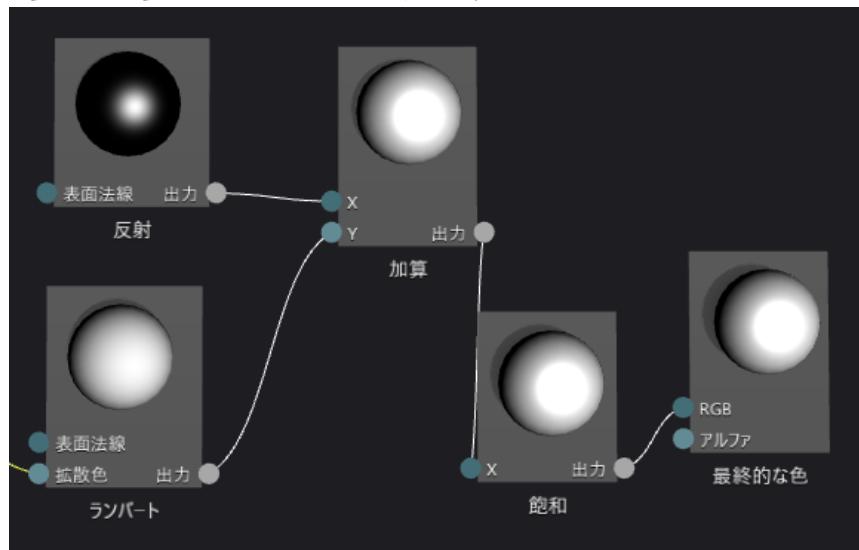
    // combine ambient with diffuse
    return saturate((materialAmbient * lightAmbient) + diffuse);
}
```

うへん。これ、そのまま使えそうにないですね。というわけで参考程度に扱うといいでしょう。

次にフォンシェーディング(スペキュラ)を実装してみましょう。

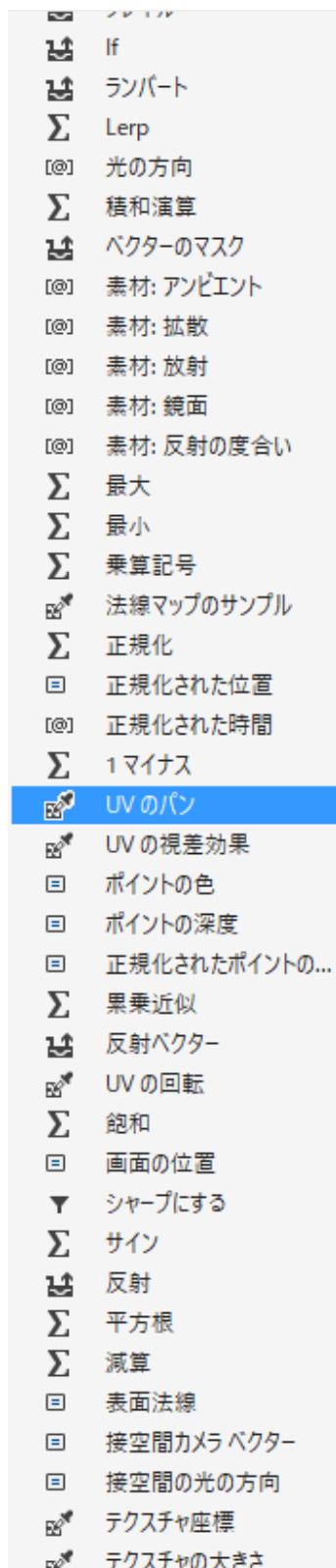
DGSL 画面に移動して

今度は、「反射」と「加算」を追加し、つなげてみます。



このように各ノードを追加することで出力(最終的な色)が決まってきます。

ツールバーには色々と



これと出力結果を見比べながら様々なエフェクトを使用してHLSL書きだしてやれることを増やしていきましょう。

影を落とす



影行列

最も簡単にそれっぽいものができるであろうモノが、これ。

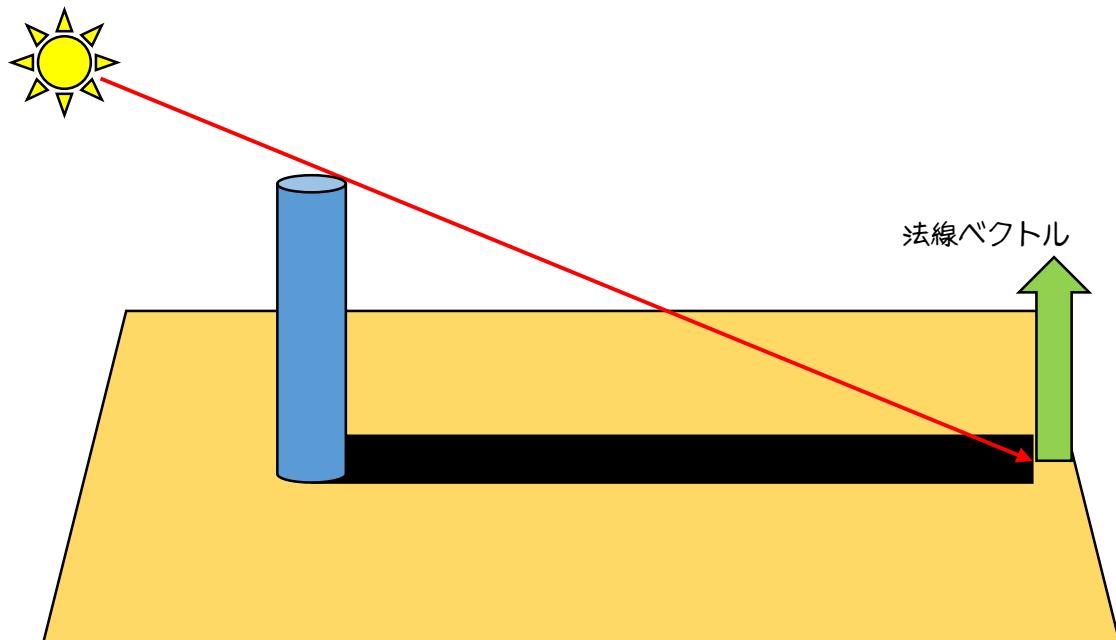
行列を用いて、光の方向と地面の法線ベクトルから、頂点をどこに落とすべきかを計算する。で、それをやる行列を作る。そしてそれを描画する際に黒で描画すればいい。

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、光の方向と法線ベクトルで決まると言ったが、図のような関係になつていると考へると、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。

で、ライトベクトルと法線ベクトルの内積をとれば、ライトベクトル1個あたりでどれくらい地面に近づくかが分かる。

ライトベクトルを \vec{L} とし、正規化された法線ベクトルを \hat{N} とすると

ひとまず、一回分の高さを d とおくと

$$k = -\vec{L} \cdot \hat{N}$$

となる。

で、頂点座標を P と置き、その位置ベクトルを \vec{P} とすると平面から頂点までの高さは

$$h = \vec{P} \cdot \hat{N}$$

となる。

そして、ライトベクトルがいくつあれば平面に到達するかは

$$\frac{h}{k}$$

でわかる。ことは分かるだろう？ということはとある点 P が平面に影を落とすときの平面上の対応する影頂点の座標は

$$P + \vec{L} \frac{h}{k}$$

となる。で、これを行列として扱う場合、元座標 P は行列にかけられるものとしてつかう。つまり影行列を S とすると

$$P' = PS$$

である。これは

$$(x', y', z', 1) = (x, y, z, 1) \begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & 1 \end{pmatrix}$$

である。この?はどういう行列になるべきだろうか？

$$\left(x + L_x \frac{h}{k}, y + L_y \frac{h}{k}, z + L_z \frac{h}{k}, 1 \right) = (x, y, z, 1) \begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & 1 \end{pmatrix}$$

まず、左辺値に生の x, y, z の項がある以上、

$$\left(x + L_x \frac{h}{k}, y + L_y \frac{h}{k}, z + L_z \frac{h}{k} \right) = (x, y, z) \begin{pmatrix} 1+? & ? & ? & ? \\ ? & 1+? & ? & ? \\ ? & ? & 1+? & ? \\ ? & ? & ? & ? \end{pmatrix}$$

であることは間違いない。

さらに、

$$h = \vec{P} \cdot \hat{N}$$

であり、Pは、それぞれのx,y,zであることから

$$\left(x + L_x \frac{h}{k}, y + L_y \frac{h}{k}, z + L_z \frac{h}{k} \right) = (x, y, z) \begin{pmatrix} 1 + \frac{N_x L_x}{k} & \frac{N_x L_y}{k} & \frac{N_x L_z}{k} & ? \\ \frac{N_y L_x}{k} & 1 + \frac{N_y L_y}{k} & \frac{N_y L_z}{k} & ? \\ \frac{N_z L_x}{k} & \frac{N_z L_y}{k} & 1 + \frac{N_z L_z}{k} & ? \\ ? & ? & ? & 1 \end{pmatrix}$$

であることがわかる…が、果たして4行目、4列目はどうするべきなんだろうか？

良く分からぬ。

ということでここで観念して

<https://msdn.microsoft.com/ja-jp/library/cc372900.aspx>

を見ることにする。

P = normalize(Plane);

L = Light;

d = dot(P, L)

| | | | |
|---------------|---------------|---------------|---------------|
| P.a * L.x + d | P.a * L.y | P.a * L.z | P.a * L.w |
| P.b * L.x | P.b * L.y + d | P.b * L.z | P.b * L.w |
| P.c * L.x | P.c * L.y | P.c * L.z + d | P.c * L.w |
| P.d * L.x | P.d * L.y | P.d * L.z | P.d * L.w + d |

となっている。とりあえず今回はディレクショナルライトなので、w=0とすると

| | | | |
|---------------|---------------|---------------|---|
| P.a * L.x + d | P.a * L.y | P.a * L.z | 0 |
| P.b * L.x | P.b * L.y + d | P.b * L.z | 0 |
| P.c * L.x | P.c * L.y | P.c * L.z + d | 0 |
| P.d * L.x | P.d * L.y | P.d * L.z | d |

で、全体を d で割ると、

$$\begin{array}{cccc} P.a * L.x/d + 1 & P.a * L.y/d & P.a * L.z/d & 0 \\ P.b * L.x/d & P.b * L.y/d + 1 & P.b * L.z/d & 0 \\ P.c * L.x/d & P.c * L.y/d & P.c * L.z/d + 1 & 0 \\ \textcolor{red}{P.d * L.x/d} & \textcolor{red}{P.d * L.y/d} & \textcolor{red}{P.d * L.z/d} & 1 \end{array}$$

であるから、さっきの式とほぼ同じですよね？

で、4行目なんですが、こいつに 1 がかけられるため、それぞれ平行移動を表します。これは $P.d$ をかけているのが分かりますよね？平面の方程式の d って何でしたっけ？そう。オフセットでしたね？

で、このオフセットに対して L/d をかけてるわけですが、 d は光線と法線の内積でした。これに $P.d$ を書けることによって、平面からのズレ（オフセット）ぶんにライトベクトルが何個分必要かつていう事になるわけです。

まあ、というわけで、ポリゴンを特定の平面に押しつぶす行列ができるという事は分かったと思います。

今のワールド行列に、`XMMatrixShadow` の行列を乗算してみよう…



ペシャンコになっているのが、わかるだろう？

さて、んじゃあこれを通常の描画と組み合わせてみよう…

ちょっとだけお悩み

そう、DirectX11だったら

1. XMMatrixShadowでペシャンコにする
2. ペシャンコにしたものを描画する
3. 行列を元の状態に戻す
4. 通常の描画を行う

で済む話なのだが、DirectX12だとそれはいけない。という事で悩む。

まず、やり方としては定数バッファは、実行時に書き換えたとしても、描画時には最後の変更しか反映されないことは以前にやって思いました。だから、定数バッファを切り替え、デスクリプターヒープの指定を変更するというやり方が一つ。

次に、もうシャドウ変換をかけた行列もバッファに追加してしまい、シェーダ側で場合分けする。もちろん場合分けをするのならば結局定数デスクリプターヒープを切り替えなければならぬ…

これは…面倒。ちょっと実験するだけなのに、楽なやり方で行きたいのになあ…。

結局結論としては、通常用と、影用のふたつのデスクリプターヒープを用意する必要があります。

一番手っ取り早い考え方としては「複数マテリアル」の時と同じ考え方をするといったところでしょうか。

つまりハンドルのポインタをずらすあのやり方ですよ…。

色々とやってみたが、手っ取り早くやるなら、最初から「マテリアルの一つ」としてシャドウを割り当てることですね。

まずは MaterialNum を MaterialNum+1 で定義しておきます。

//マテリアルに+1しているのは「影」のぶんです

```
auto cbo = dxbuffManager.CreateConstantBufferObject(sizeof(CBuffer), materialNum+1, 0,  
D3D12_SHADER_VISIBILITY_ALL);
```

で、通常の描画のあとで

```

cbuffTemp->world *= XMMatrixShadow(XMLoadFloat4(&p), XMLoadFloat4(&l));
cbuffTemp->diffuse = XMFLOAT3(0,0,0); //黒く塗れ!
cbuffTemp->alpha = 1.0f;
cbuffTemp->specularity = 0;
cbuffTemp->specular = XMFLOAT3(0, 0, 0); //黒く塗れ!
cbuffTemp->ambient = XMFLOAT3(0, 0, 0); //黒く塗れ!
cbuffTemp->existTexture = false;
cbuffTemp->existSPA = false;

```

で設定して、その後で、全身を描画します。

```

_commandList->SetGraphicsRootDescriptorTable(cbo->GetParameterIndex(), handle);
_commandList->DrawIndexedInstanced(indexCount, 1, 0, 0, 0);
うまくいけば…

```



てな感じで、あたかも足元に床があるかのように影が描画されます
さて、ここまでほんのジャブ…。

ここからが本番!!!せいぜい悩むがいいさ!!!

シャドウマップ

シャドウマップ……。

それは、地獄の入り口でございます…。

シャドウマップに必要な概念を最初に書いておきます

- ライトビュー(ライトの場所から見たら何が見えるか)
- 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(深度値を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。真ん中の「テクスチャへの深度値の書き込み」が一番の難関だと思います。DirectX11ではやれたが…果たして12ではどうかな?

でも割とイメージもしやすいため、そこまで苦労はしないと思いますが…まあ頑張ってやっていきましょう。

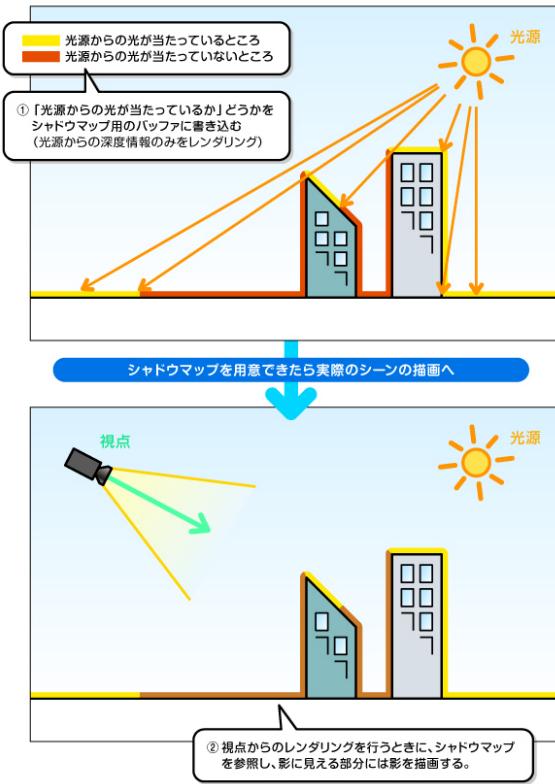
超が付くほどの余裕があれば、このシャドウマップの問題点とその解決法について話してみたいかなとは思っています。

その概念

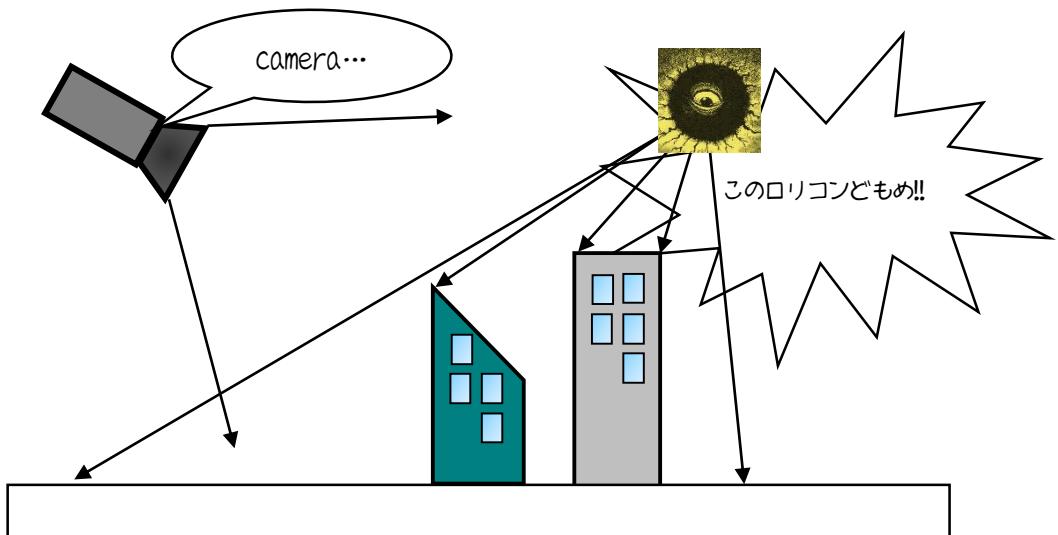
そもそも影が落ちるということはどういうことなのかというのを今一度考えてみましょう。

「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。

それについて <http://news.mynavi.jp/column/graphics/024/> の解説が分かりやすいとは思いますが、この図が超分かりやすいので使いましょう。



この図に全てが書かれています。この図よりも分かりやすい図ってないんじゃないかなあ…。
 まあ当たり前の話ですよね？図を見れば。
 この「当たり前」の事を塔やって実装するのか…これ最初に考えたやつは天才だと思います。
 上の図のオレンジの部分は手前の物体に光が遮蔽されて光が届いてないわけです。
 それはどういうことなのか？
 そうですね。
 光源を光の発生源であると同時に視点だと思ってください。



と、言うことは光源を視点として考えた場合、そこ(光源=視点)から見えているものにのみ光が当たるというわけです。

見えないものには光が当たらない…そういうことだ。

なるほど、理屈はわかった。

- 光源から見えている&&カメラから見えている→明るく見える
- 光源から見えていない&&カメラから見えている→暗く見える
- カメラから見えていない→問題外

こういうわけか。

さて、んじゃあ特定の点に対して「遮蔽されているかどうか」をどうやって判断したら良いんだろう？

二段階に分けて処理をすることになります。

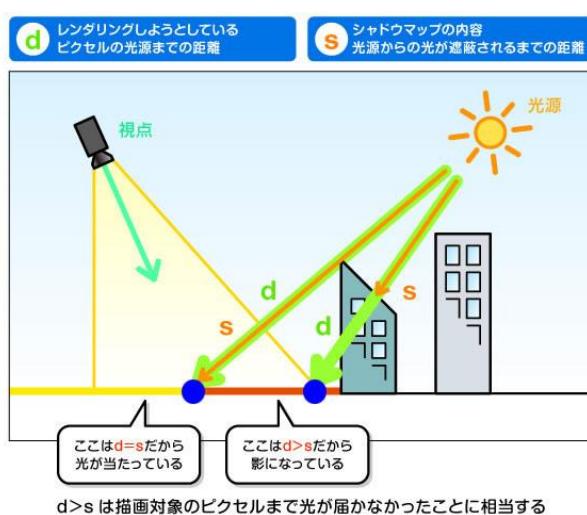
第①段階：ライトからのレンダリング（深度値）をテクスチャに書き込む

第②段階：通常のカメラレンダリングを描画する

この時の第②段階が重要。

描画の際に、描画しようとしている点が「ライトから見えているかどうか」をチェックしなければならない。この「チェック」とは何をチェックするのか？

そこでまたさっきのページの別の図を見てみましょう。



ホンマにこの図は秀逸です。わあ…パーカクト!!

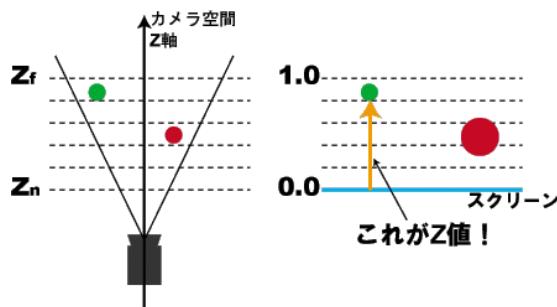
チェックとは…特定の点の座標の「光源からの距離」が「光線遮蔽物より遠くにあるのかどうか」である。

s の内容は第①段階で計算されます。ただし $0.0 \leq s \leq 1.0$ である。こうなる理由は後で話します。

じゃあ d はどうやって計算すれば良いんでしょう? 光源位置を L とし、対称座標を P として

$d = \text{length}(P - L);$

とでもするか…だがここで問題があるのだ。先程も言ったように $0.0 \leq s \leq 1.0$ である。つまりところ単位系がぜんぜん違うのと似たような状態になっているのでこれに合わせなければならぬ。



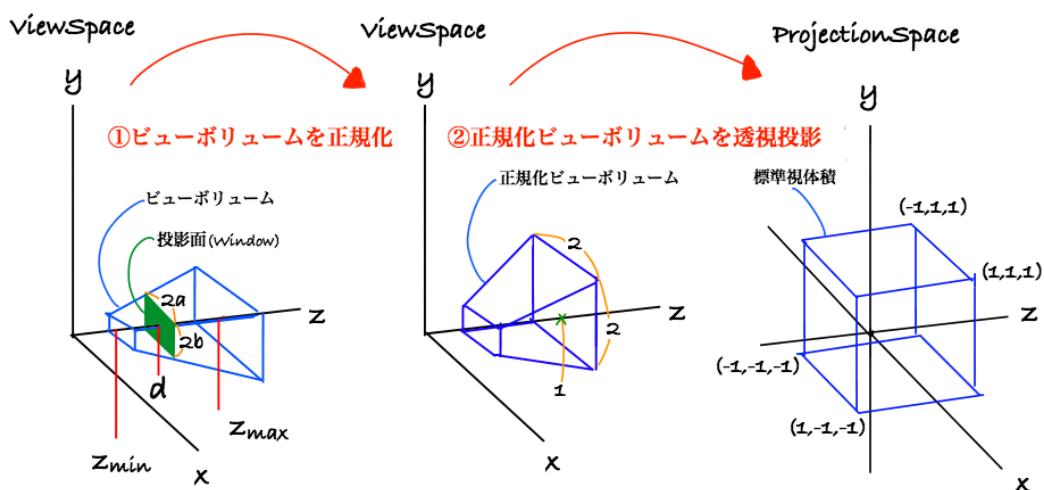
ちなみにそもそも何故 $0.0 \leq s \leq 1.0$ なのか?

<http://www.slideshare.net/todoroki/2-28983824>

よくわからんね…

<http://esprog.hatenablog.com/entry/2016/04/09/172925>

こっちのほうが分かりやすいかな…



そもそも射影行列ってのが、カメラから見た視錐台を「正規化デバイス座標系」に変換するものです。上の図のようにカメラから見たバーチャル空間を正規化デバイス座標系に変換し、そ

れをスクリーンに書き出します。スクリーンは二次元平面なので Z 方向は一番手前のもののみ描画します。

ところで画面を見ればわかりますが、どれも -1 ~ 1 の範囲になってますが、結局これって深度バッファに書き込む時に最終的に「テクスチャ」に書き込む必要があるので 0 ~ 1 の範囲にしなければいけません。つまり 1 を足して 2 で割っています。

はい…ちょっと長々と書いたけどつまるところ、結果が 0 ~ 1 になるようになっています。

もちろん比較関数もこれが必要なので、何をする必要があるのかというと、カメラから見た デプス値(0 ~ 1 に正規化された「光源からの距離」と、最初に光源から取ったデプス値を比較するわけです。

さて、比較というわけですが、ライトから見た「カメラ行列」で変換した値と、第①で取った深度値を比較すれば良いわけです。

演習準備

今回は「ガチの影」を作っていくので、影を落とす先が必要になります…つまりまずは床が必要になります。

また、遮蔽物があっても適切に影が落ちることを確認するために障害物も置きましょうか。

うーん。

メッシュと同じ頂点情報でやっちゃうと過剰情報な気がしますので、PMD 表示用とは異なるものを作りましょう。

つまり…

Plane クラス

Cylinder クラス

を作りましょう。う~ん。順調に進みすぎてるし、ひとまず仕様だけ言います。ちょっとクラス設計的な意味でね。

Plane は「平面」って意味で、Cylinder は円柱、円筒って意味です。

Plane はコンストラクタに幅と奥行を与えると法線(0,1,0)の平面の頂点バッファを生成します。

Cylinder はコンストラクタに、半径と高さと分割数を入れると円柱の頂点バッファを生成します。

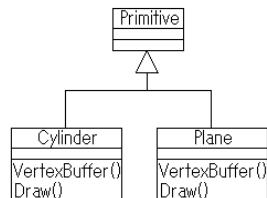
どちらも VertexBuffer() 関数で頂点バッファを返します。

頂点バッファは返しますが、今回のこいつらはどうせプリミティブ(原始的な)形状なので Draw() 関数もこいつらに持たせてあげていいでしょう。

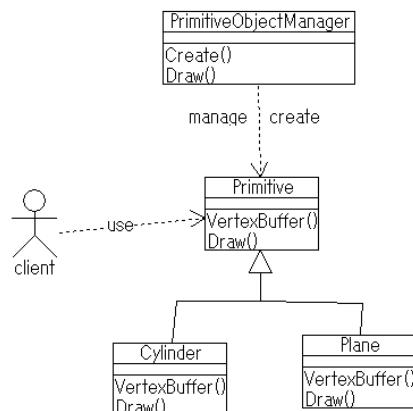
基本的にはこいつらは内部で全て完結するようにしてあげましょう。

また、内部で使用するシェーダはプリミティブ共通のものを使うようにすればいいでしょう。

つまり…



う~ん。そうなってくると結局は PrimitiveObjectManager なんて作って…Create 関数と Draw 関数作って…作ったプリミティブを操作する…って感じにするなら



こうでしょうかね。うーん、やっぱりこんな構造にするんじゃねえかよお前ふざけんな。
まあまあ、最初から完成品にする必要はないのだ。

ひとまずは最初に言った仕様を満たせば良い。

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。

//円柱

```
class Cylinder : public PrimitiveObject
{
public:
    Cylinder(float radius, float height, unsigned int div);
    ~Cylinder();
    ID3D12Resource* VertexBuffer();
};
```

//平面

```

class Plane : public PrimitiveObject
{
public:
    Plane(float width, float depth, float nx, float ny, float nz);
    ~Plane();
    ID3D11Buffer* VertexBuffer();
};


```

とりあえず頂点情報のみで床と遮蔽物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 PrimitiveVertex{
    XMFLOAT3 pos;
    XMFLOAT3 normal;
    XMFLOAT3 color;
    XMFLOAT2 uv;
    PrimitiveVertex(){
        pos=XMFLOAT3(0,0,0);
        normal=XMFLOAT3(0,0,0);
        color=XMFLOAT3(0,0,0);
        uv=XMFLOAT2(0,0);
    }
    PrimitiveVertex(XMFLOAT3& p, XMFLOAT3& norm, XMFLOAT2& 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にこんな名前にしている。
        pos = p;
    }
};


```

```

        normal = norm;
        uv = coord;
    }

    PrimitiveVertex(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
 例えはプリミティブ頂点を宣言する時に

`PrimitiveVertex p(-10, -0.2, 10.f, 0, 1, 0, 0, 0);`

などと書けます。

更に言うと、vector への `push_back` などでは

```

std::vector<PrimitiveVertex> vertices;

vertices.push_back(PrimitiveVertex(-10, -0.2, 10.f, 0, 1, 0, 0, 0));
vertices.push_back(PrimitiveVertex(10, -0.2, 10.f, 0, 1, 0, 1, 0));
vertices.push_back(PrimitiveVertex(-10, -0.2, -10.f, 0, 1, 0, 0, 1));
vertices.push_back(PrimitiveVertex(10, -0.2, -10.f, 0, 1, 0, 1, 1));

```

とでもすればいい。これで頂点をつくって頂点ノッファを返せばいい。あとはレイアウトと、シーケンスをセットすれば床も表示されるようになります。

あとは円柱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ぬれいかもしれないるので、ヒントを言おうか。

- 円柱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
- プリミティブトポジが TRIANGLESTRIP であれば N 字を並べればよい
- プリミティブトポジが TRIANGLELIST ならば長方形一つに n 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<PrimitiveVertex> 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 = r*cos((XM_2PI / float(div))*(float)i);
    vertices[i * 2 + 1].pos.z = r*sin((XM_2PI / float(div))*(float)i);
    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柱を表示させてください。と、言いたいところですが、DirectX12 はそう簡単にいられないかな…。

とりあえず思考をまとめておくと、必要なものは…

- 頂点バッファ
- 頂点バッファビュー

と、まあこれくらいで、テクスチャバッファとか定数バッファに比べたら大したことはないんだが、問題は



プリミティブと PMD モデルとでは頂点 レイアウトも使用するシェーダも違う

これは初体験…だろ？

とりあえず、そのレイアウトとかシェーダだけど PrimitiveCreator が保持しておくことにしましょう。

ちなみに PrimitiveCreator は

```
// 前方宣言
class PrimitiveObject;
struct ID3D12PipelineState;

class PrimitiveCreator
{
private:
    // プリミティブ用のパイプラインステート
    ID3D12PipelineState* _pipelineState = nullptr;

public:
    PrimitiveCreator();
    ~PrimitiveCreator();

    /// この関数内でプリミティブ用のレイアウト、シェーダ
    /// パイプラインステートが初期化されます
    void Init();

    /// プリミティブ用のパイプラインステートが内部でセットされます
```

```

///Primitive系のDraw命令を出す前に必ず呼んでください
void SetPrimitiveDrawMode();

///平面オブジェクトを作る
///@param width 平面の幅
///@param depth 平面の奥行き
///@param y 平面のY座標
PrimitiveObject* CreatePlane(float width, float depth, float y);

///円柱オブジェクトを作る
///@param radius 半径
///@param height 高さ
///@param div 分割数
PrimitiveObject* CreateCylinder(float radius, float height, unsigned int div);
};

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。

```

うまいこといけば床が表示されますが…



ごらんのように床の表面に影らしきものがグジヤグジヤーっと表示されます

この事を Z-Fighting と言います。Z 値の事を深度値と言うようになって長い年月が経ちま

したが、この Z-Fighting という言葉はいつまで経っても Depth-Fighting とは呼ばれませんし、深度ファイティングなんて言う風にも言われません。

…それじゃあ Z の呼称のままでよかつたんじゃないのか？ Z バッファ… Z ソート… Z 値… なんでそんな中途半端なことするん？

ともかく、↑のような Z ファイティングなどという現象が発生します。原因はというと、影の座標も $y=0$ であり、平面の座標も $y=0$ である。こうなるとコンピュータは…というかグラボはどうちらを優先して表示したらよいのか分からぬのだ。

正確に言うとその座標における影の深度値(カメラからの距離)と、その座標における平面の深度値を測って、よりカメラに近いほうを採用するのだが、この場合はどちらも正解という事になる。

そういう中途半端な状況だと、結局浮動小数点誤差でどっちつかずな答えとなってしまう。これに対する対処法は、地面と影の間にほんの少しだけ隙間を開けてやることだ。

- 影をちょっと上に上げる
- 床をちょっと下に下げる

のどちらかである。

ためしに影をちょっとだけ上に上げてみよう…

```
XMVECTOR plane = { 0,1,0,-0.0005f }; // 0.0005fだけ上に上げて Z ファイティングを防止
```

とでもやれば隙間ができるため、Z ファイティングは発生しなくなる。同様の問題はシャドウマップでも発生するので、その場合は影の方をカメラ方向にずらしてやる必要があります。

あ、もしかしたら単なる床表示で詰まってる人がいるかもしれませんので、書いておくと

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。

うまくいけば。



こんな感じで出力されるのがわかるだろう?
チョット浮かすだけでミライティングを改善できるのだ。すごいだろう?

さて、その意気で円柱も作っちゃおう
ひとまずノーヒントでおながいします。



普通にやると靈夢さんと被るので、平行移動行列 XMMatrixTranslation でも使ってずらしてください。

と言いたいところなのですが…まさか、ああ、まさか……。またあの問題か…。ホンマに DX11 のときはこんなもんかんたんだったのに…。

コンスタント/バッファ周りは悩まされるなあ…仕組みじやなくて、設計がね…。

手っ取り早くやるならまた「マテリアルとみなす」方式でやるんだけど…納得いかへんなあ…。

ということでちょっと設計変更について考えてますが、やっぱり幾何学変換とマテリアルは分けるべきかなーって考えていたりします。

今はちょっと面倒だしひとまずマテリアルでやっていきましょうか…。ひとまずね。ここで設計とかをやれる余裕のある人はしっかり設計してください。

//+1の部分は影行列のため
//「影」をマテリアルのひとつとしてとらえる

```

//さらに+1しているのはプリミティブの一部用…。
//実際この数は最終的には現実的ではなくなるため、
//もうマテリアルと行列に関しては分けるべきであろうと思う
auto cbo = dxbuffManager.CreateConstantBufferObject(sizeof(CBuffer), materialNum+2, 0,
D3D12_SHADER_VISIBILITY_ALL);

```

としておき、あとはもうパターンだからわかりますよね？バッファとハンドルを進めてあげて描画です。

```

handle.ptr += descSize;
cbuffTemp = (CBuffer*)((char*)cbuffTemp + GetSizeOf256Alignment(cbuffTemp));
*cbuffTemp = *cbuffer;
cbuffTemp->world = XMMatrixTranslation(-15,0,0)*cbuffTemp->world;
_commandList->SetGraphicsRootDescriptorTable(cbo->GetParameterIndex(), handle);

```

円柱をちょっと横にずらします。

すると以下の画像のようになりますが、影に注目すると円柱の下に影が潜って不自然ですね？



これが射影行列の限界っ…!!

ここでシャドウマップですよ。

でもシャドウマップってのは前にも話したように、光源の位置から対象をカメラ撮影しなければならないため、その絵を焼き付ける紙(フィルム)が必要…



撮影してもフィルムが入ってなかつたら残せないからね

シャドウマップ実装

ノーマルマップの確保

まあ、デジカメの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ノーマルマップの部分を参考にしてください。

```
ID3D12DescriptorHeap* shadowDH=nullptr;  
ID3DResource* shadowBuffer=nullptr;
```

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 shadowmapHD = {};  
shadowmapHD.NumDescriptors = 1;  
shadowmapHD.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  
ID3D12DescriptorHeap* _shadowDH = nullptr;  
result = dev->CreateDescriptorHeap(&shadowmapHD, IID_PPV_ARGS(&_shadowDH));
```

デスクリプターヒープはこんなもんでいいんですが、問題は深度ノーマルマップのほう。
通常の深度ノーマルマップ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。ちなみに言うと、サイズはどつちみち2の乗数ぶんが乗つかるので、それも考慮すると…

```
size_t ssize=max(WINDOW_WIDTH,WINDOW_HEIGHT);  
こういう感じででかい方を取って来て  
///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 Roundup2Multiplier(size_t size) {  
    size_t bit = 0x80000000;  
    for (size_t i = 31; i >= 0; --i) {  
        if (size&bit)break;  
        bit >>= 1;  
    }  
    return bit << 1;  
}
```

みたいにな関数を用意して…と、しつと書いたけど、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な、これは…という事で、この後は自分で考えてくれたまえ。分からぬ？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こうしました。

```
return size+((bit << 1)-size)%bit;
```

なんでなのか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ら、

```
ssize = Roundup2Multiplier(ssize);
```

で適切なサイズを測っておいて…

```
D3D12_RESOURCE_DESC shadowmapResDesc = {};  
shadowmapResDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;  
shadowmapResDesc.Width = shadowmapResDesc.Height = ssize;
```

```
shadowmapResDesc.Format = DXGI_FORMAT_R32_TYPELESS; //D32_FLOATだと後で使いにくいので
shadowmapResDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
shadowmapResDesc.DepthOrArraySize = 1;
shadowmapResDesc.MipLevels = 1;
shadowmapResDesc.SampleDesc.Count = 1;
shadowmapResDesc.SampleDesc.Quality = 0;
shadowmapResDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;
こんな感じに設定しておきます。なお、FormatはD32_FLOATだと後々使いにくいので
R32_TYPELESSにしておきます。なんかR32_FLOATにするとINVALIDARGが返ってくるので、こ
こはいったんTYPELESSにしておきます。何でだろう…?
```

あとは既に作っている深度バッファビューと同じように作ります。

```
D3D12_CLEAR_VALUE clearValue = {};
clearValue.Format = DXGI_FORMAT_D32_FLOAT;
clearValue.DepthStencil.Depth = 1.0f;
clearValue.DepthStencil.Stencil = 0;

result = dev->CreateCommittedResource(&shadowHeapProperties,
                                         D3D12_HEAP_FLAG_NONE,
                                         &shadowmapResDesc,
                                         D3D12_RESOURCE_STATE_DEPTH_WRITE,
                                         &clearValue,
                                         IID_PPV_ARGS(&_shadowBuff));
```

ここまで通常の深度バッファと同じです。

2つの見方(深度バッファ/シェーダリソース)

何が違うのかというとこのバッファに対する「見方」が二つある…つまり「ビュー(デスクリプタヒープ)」が二つできることになります。

まあこの辺はDirectX11の時にやったのと同じ発想ではあるんだけど、データはひとつ。見方はふたつって事やね。

真実はいつも一つ!!なんだけど、何とかねえ…見方によって、利用法も意味も変わってきちゃうわけですよ。その見方がDX11の場合はビューであり、DX12の場合はデスクリプタヒープのよ

うです。



同じ人です

というわけで、先ほど作ったバッファに対して二つの見方を作つてあげましょう。
まずは深度バッファの見方…これは自分のソースコードの深度バッファデスクリプターヒープの作り方を参考にして作つてください。

```
D3D12_DESCRIPTOR_HEAP_DESC shadowmapDSV_DHD = {};
shadowmapDSV_DHD.NumDescriptors = 1;
shadowmapDSV_DHD.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
ID3D12DescriptorHeap* _shadowDsvDH = nullptr;
result = dev->CreateDescriptorHeap(&shadowmapDSV_DHD, IID_PPV_ARGS(&_shadowDsvDH));
```

次にシェーダリソースビューとしての「見方」を作つてあげます。

```
//シェーダリソースとして
D3D12_DESCRIPTOR_HEAP_DESC shadowmapSRV_DHD = {};
shadowmapSRV_DHD.NumDescriptors = 1;
shadowmapSRV_DHD.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
shadowmapSRV_DHD.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
```

```
//深度値用デスクリプターヒープの作成(シェーダリソース)
ID3D12DescriptorHeap* _shadowSrvDH = nullptr;
result = dev->CreateDescriptorHeap(&shadowmapSRV_DHD, IID_PPV_ARGS(&_shadowSrvDH));
```

てな感じです。

これで二つの見方ができました。あとは関連付けるだけです。

//デプスステンシルビューと関連付け

```
D3D12_DEPTH_STENCIL_VIEW_DESC shadowDSVDesc = {};
shadowDSVDesc.ViewDimension = D3D12_DSV_DIMENSION_TEXTURE2D;
shadowDSVDesc.Format = DXGI_FORMAT_D32_FLOAT;
dev->CreateDepthStencilView(_shadowBuff, &shadowDSVDesc, _shadowDsvDH->GetCPUDescriptorHandleForHeapStart());
```

//シェーダリソースビューと関連付け

```
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Format = DXGI_FORMAT_R32_FLOAT;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MipLevels = 1;
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
dev->CreateShaderResourceView(_shadowBuff, &srvDesc, _shadowSrvDH->GetCPUDescriptorHandleForHeapStart());
```

2回レンダリング(2パスレンダリング)

1ターン2回攻撃みたいでかっこいいですが、まあ面倒です。

えーと、もうね、通常の描画とは完全に別として考えてください。んで、順序としては

1. ライトビュー用レンダリング
2. 通常レンダリング

の順番で実行します。

ライトビューってのは光源からの撮影ってことだけどとにかく今はそのままの状態で撮影してください。カメラが光源というイメージになります。まだまだシャドウマップは先が長いのです(それほどでもないけど)なので、今は2パスレンダリングを実行できるという所を目指します。

まず1パス目のライトビューですが、これはどうせ深度バッファしか書き込みませんのでマテリアルなんか考えなくていいです。ただしワールドビュープロジェクション行列は適用してください。あとボーンも適用してね。

となると…

```
//-----シャドウマップ用処理-----
_commandList->SetGraphicsRootSignature(dxbuffManager.GetRootSignature());
_commandList->ClearDepthStencilView(_shadowDsvDH->GetCPUDescriptorHandleForHeapStart(),
D3D12_CLEAR_FLAG_DEPTH, 1.0f, 0.0f, 0, nullptr);
//シャドウマップ用レンダーターゲットに切り替えしーの
_commandList->OMSetRenderTargets(0, nullptr, false, &_shadowDsvDH-
>GetCPUDescriptorHandleForHeapStart());
//アスペクト比変更(オンマはパースペクティブも変更する必要があるがそれは後でやる)
viewport.Width = ssize;
viewport.Height = ssize;
scissorRect.right = ssize;
scissorRect.bottom = ssize;
_commandList->RSSetViewports(1, &viewport);
_commandList->RSSetScissorRects(1, &scissorRect);
cbuffer->diffuse = XMFLOAT3(0, 0, 0);
cbuffer->alpha = 1.0f;
cbuffer->specularity = 0.0f;
cbuffer->specular = XMFLOAT3(0, 0, 0);
cbuffer->ambient = XMFLOAT3(0, 0, 0);
cbuffer->existTexture = false;
cbuffer->existSPA = false;
ID3D12DescriptorHeap* shadowDescHeaps() = { cbo->GetDescriptorHeap() };
_commandList->SetDescriptorHeaps(1, shadowDescHeaps());
_commandList->SetGraphicsRootDescriptorTable(cbo->GetParameterIndex(), cbo-
>GetGPUHandle());
_commandList->IASetIndexBuffer(&_indexBufferView);
_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
_commandList->IASetVertexBuffers(0, 1, &_vbView);
_commandList->DrawIndexedInstanced(indexCount, 1, 0, 0, 0);

//必要ないと思うけど念のため
_commandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(_shadowBuff,
D3D12_RESOURCE_STATE_DEPTH_WRITE,
D3D12_RESOURCE_STATE_PRESENT));
```

```
_commandList->Close();

dx12.ExecuteCommand();
//GPU側の実行を待つ
dx12.WaitWithFence();
_commandAllocator->Reset();
result = _commandList->Reset(_commandAllocator, _pipelineState);
```

これが1パス目。これによりシャドウバッファに深度値が書き込まれます。これを後々シェーダリソースとして利用する必要がありますが、ひとまずはそれは置いといて、2パス目を今まで通り描画してみてください。

以前と変わってなければ成功です。おかしければどつかでやらかしています。

さて、1パス目のレンダリング結果をとにかく見えるようにしたいのでちょっと細工します。まずはシェーダリソースビューを使えるようにしたい…その場合はまたルートシグネチャに登録する必要があるので、それを行います。僕はすでに関数を作っているので、

```
///深度テクスチャをルートシグネチャへ(レジスタ番号2番…ルートシグネチャインデックス4番)
dxbuffManager.AddParameterAndRangeForRootSignature(D3D12_SHADER_VISIBILITY_PIXEL,
D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 2);
```

これでオッケーです。なお、この関数はルートシグネチャインデックス番号を返すので、それを保持しておけば問題ないです。

さて、レジスタ番号2番に入れる事にしましたので、シェーダ側も対応します。

```
Texture2D<float> shadowmap:register(t2);
```

ちなみに深度値しか書き込む気はないので、`Texture2D<float>`です。`Texture2D<float4>`とかではありません。

で、ちょっと分かりやすいようにプリミティブ描画時にこのテクスチャを表示してみましょう。

```

//プリミティブ表示用ピクセルシェーダ
float4 PrimitivePS(PrimOutput inp) : SV_Target
{
    float3 light = normalize(float3(-1,1,-1)); //平行光線の逆ベクトル
    float3 color = shadowmap.Sample(smp, inp.uv);
    return float4(color, 1);
}

```

次はこれを通常描画時にレジスタ 2 番へセットします。先ほどの関数でルートシグネチャにおけるインデックス数はわかっているので、

```

ID3D12DescriptorHeap* shadowDescHeap() = { _shadowSrvDH };
_commandList->SetDescriptorHeaps(1, shadowDescHeap);
_commandList->SetGraphicsRootDescriptorTable(shadowmapRSIndex, _shadowSrvDH-
>GetGPUDescriptorHandleForHeapStart());

```

こういう風にします。でも表示は…



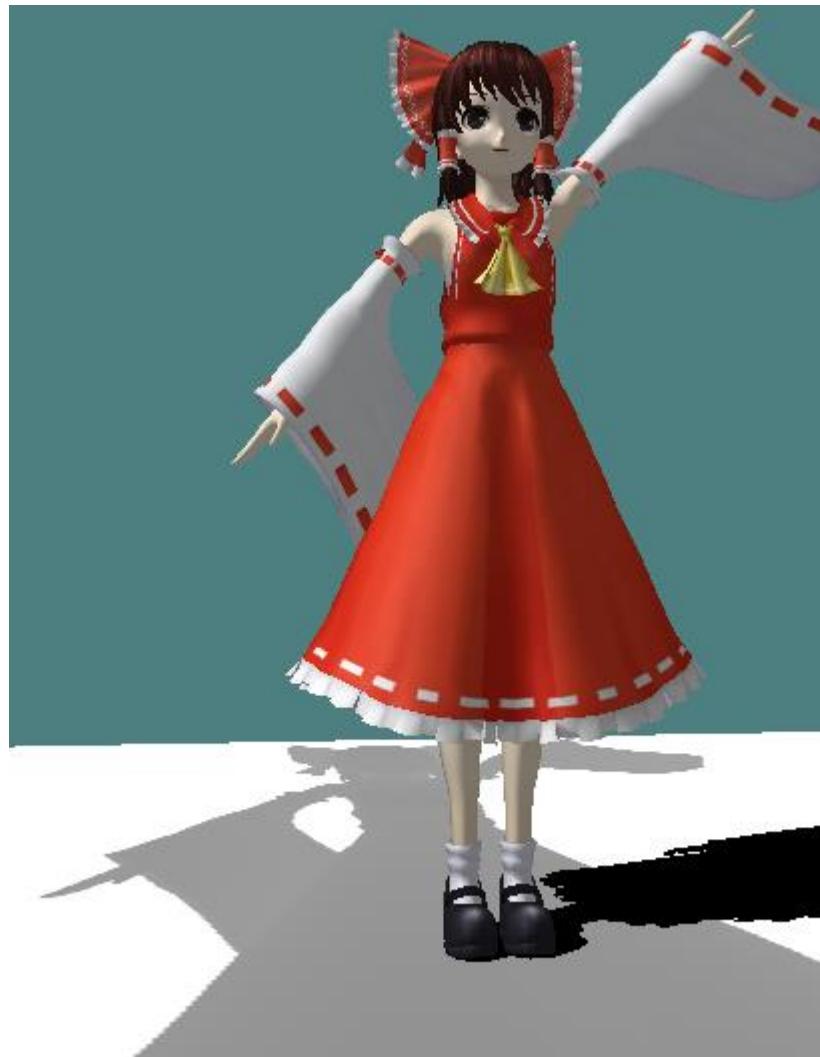
なんか薄くて分かりづらいですねえ…

なので、分かりやすくします。

`float3 color = shadowmap.Sample(smp, inp.uv);`
の部分を指数的に分かりやすくします。

```
float3 color = pow(shadowmap.Sample(smp, inp.uv),10);
```

くらいやれば差が顕著になり



こうなります

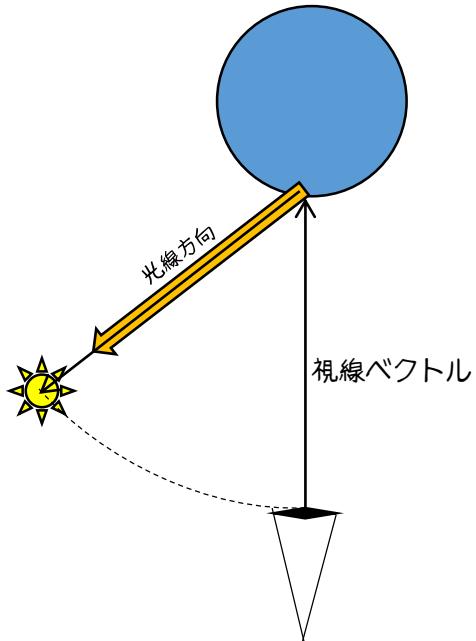
今日はここまでやってみてください。今からライトマップしていくので、`XMMatrixShadow`の影は消しておいてください。

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ります。

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。

つまり最初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

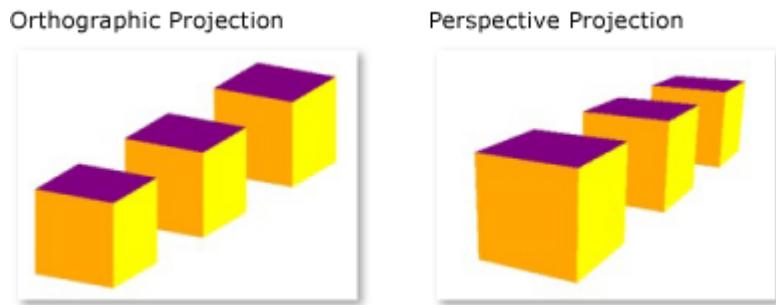
```
lightpos = toLight*(target-eye).Magnitude();
```

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ション行列などあるのでしょうか?

あるのです。ペースをかけない時点でプロジェクションとは言えないのですが、どちらみち 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クォータービューみたいな3Dゲームもありますので、知つ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.xmmatrixorthographich\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixorthographich(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;

とすれば、ライトから見た(リースをかけない)画像が見れます。



なお、ちょっと太ましいのは、縦横同じ大きさのを無理やり画面サイズに合わせているからです。アスペクト比の狂いにすぎません。

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、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といてください

さい。

さて、このライトビューProjection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Projectionを通常にするようにしてください。

ここで注意点ですが、



このように表示されてるので、これを影だと思う人もいますが、違います。

違うんです。これはただ単に深度値を色にしてプリミティブに貼り付けているだけです。ここからが悪夢の本番です。

悪夢の深度値比較…!

ここからの話はまさに悪魔的…

そうですね…。



ひとまず下準備としてライトの座標をコンスタントバッファに混ぜといてください。

```
cbuffer mat : register(b0) {
    matrix world;
    matrix viewproj;
    float3 diffuse;//拡散反射成分(基本色)
    float alpha;//拡散反射成分(基本色)
    float specularity;//スペキュラ強さ

    float3 specular;//スペキュラ色
    float3 ambient;//環境光色

    int existTex;//テクスチャあるか
    int existSPA;//スフィアマップあるか
    float3 eye;//視点
    float3 lightpos;//ライト座標(仮)
}

struct CBuffer {//ワールド行列とViewProj行列
    XMATRIX world;//ワールド行列
    XMATRIX viewproj;//カメラ、プロジェクション
    XMFLOAT3 diffuse;//ディフューズ
    float alpha;//アルファ
    float specularity;//スペキュラ強さ
    XMFLOAT3 specular;//スペキュラ色
    XMFLOAT3 ambient;//環境光成分
    int existTexture;//テクスチャあるか
}
```

```

int existSPA;//スフィアマップあるか
XMFLOAT3 eye;//視点座標(ピクセルシェーダに「視線」をつくるため)
XMFLOAT3 lightpos;//ライト座標(仮)
};

```

もちろん、この lightpos にはライトの座標を忘れずに代入しておいてください。
さて、

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の範囲にされているのである。さらに言うと、既に画像化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UV値が分からぬ

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。

三次元上の調べたい座標に対して `lightviewproj` 行列を適用する。その座標を仮に `shadowpos` とする。

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。これで既に持っている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 $x:(-1\sim 1), y:(-1\sim 1), z(0\sim 1)$ 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…
 $-1\sim 1 \Rightarrow 0\sim 1$ 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(1,-1))*0.5;
```

となる。

さて、比較である。

比較

```
float Id=output.shadowpos.z;
```

この値と深度マップの値を比較するつまり

```
if(Id > shadowmap.Sample(smp, inp.uv).r){  
    //暗くする  
}
```

というわけだ。ちなみに今の状況であれば、シャドウマップの r 値だけでもそれなりに影っぽ

いのは出るので

```
float2 uv=(float2(1,1)+shadowpos.xy*float2(1,-1))*0.5;  
float lightdepth = shadowmap.Sample(smp, uv);  
return float4(lightdepth, lightdepth, lightdepth, 1);
```

で、以下のような影っぽいのは出るのだが…



一見、これで良さそうにも思えるのだが、違うんだな、これが。よく見てください。これは影ではなく z 値を貼り付けただけのものです。影のように見えるのは lightviewproj で変換した後の shadowpos の xy を uv として使っているからそう見えるだけなのです。ですから、



物体が手前に来ても影が出ちゃう。

というわけで、影の計算はきちんとやらねばならないのです。

```
if (inp.shadowpos.z > lightdepth) {  
    brightness *= 0.7f; //暗くする  
}  
return float4(brightness, brightness, brightness, 1);
```

そこでこのように書きます。そうすれば…



このように正しい影が出るでしょう。ちなみにセルフシャドウについてですが…

自分自身への影はそのままだと



このようにノイズが入るので

ちょっと下駄を履かせて

```
if (Id > lightdepth+0.0005f) {  
    dbright *= 0.7f; //暗くする  
}
```

としましょう。そ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

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。嫌だな。とりあえずここまでで一旦今年の最低目標までは到達しましたので、今日はあとは質問タイムとします。

シャドウをパースペクティブにすることもできるのですが、その場合はちょっと注意点があつて、パースがかかっているって事はそれぞれの座標にパースがかかっているという事なので、ちょっと特殊な処理が必要になります。

パースペクティヴをかけたものと比較する場合
x,y,z を w で割らないと期待する値になりません。

つまり…

```
float2 uv = (float2(1, 1) + inp.shadowpos.xy / inp.shadowpos.w * float2(1, -1)) * 0.5f;  
float Id = inp.shadowpos.z / inp.shadowpos.w;
```

とやらねばならないのです。ああめんどくさ。え？理由をもっと詳しく知りたい？

<https://msdn.microsoft.com/ja-jp/library/cc372898.aspx> を見るがいい！

| | | | |
|---|---|----------------------|---|
| w | 0 | 0 | 0 |
| 0 | h | 0 | 0 |
| 0 | 0 | zf / (zf - zn) | 1 |
| 0 | 0 | -zn * zf / (zf - zn) | 0 |

そして、コレ1つに(x,y,z,1)をかけるがいい！

$$\left(xw, yh, \frac{zf}{zf - zn} (z - zn), z \right)$$

これによりすべての座標変換が z 値の影響を受けることになります。なお、ニアとファーが例えば 1.0f ~ 100.0f だとすると Z=ZN=1 のときは Z=0 に変換され、Z=ZF=100 の時は (xw, yh, zf, zf) となります。ここで「同時座標系」というものについて知っておきましょう。

この名前で検索してもなかなか出てこないのだが、そもそもは CG 用語でも何でもなく、数学的な座標の話らしい。通常の座標系は (x, y, z) なのだが、同時座標系と言うのは $w \neq 0$ の実数 w を用いて、座標を (wx, wy, wz, w) と表すものようだ。通常の座標系仕様においては全てを w で割ることにより $(x, y, z, 1)$ というユークリッド幾何学空間に変換される（これがいわゆる「座標系」）。

という事は、 w が 1 ではない場合は、すべてを w で割ることにより、空間上の座標を得られるわけである。このことから透視変換（パースペクティヴ）においては、計算後の座標に対して w で割ることが求められるのだ。

一方平行投影の場合は

[https://msdn.microsoft.com/en-us/library/windows/desktop/bb205347\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb205347(v=vs.85).aspx)

$$\begin{matrix} 2/(r-l) & 0 & 0 & 0 \\ 0 & 2/(t-b) & 0 & 0 \\ 0 & 0 & 1/(zf-zn) & 0 \\ (l+r)/(l-r) & (t+b)/(b-t) & zn/(zn-zf) & 1 \end{matrix}$$

このような行列であるため、Wに変化はなく、ややこしい計算(除算)が必要ないものである。

トゥーンとかやってみよう

さて、シャドウマップまでやっちゃっておじさんちょっと満足しちゃったんだけど、ゲームに必要な技法はまだまだあるよ。ここからはセンサーの趣味もあって好き勝手やっていくよー♪

まず“トゥーンシェーダ”とか“セルシェーダ”とかいう技法なんだけど、これに関しては好みといふか意見が分かれますね。洋ゲー好きな人はセルシェーダなんていらんと思うだろうし、日本的にはセルシェーダありきだと思う。例えば…

みんな大好きプリキュアも、トゥーンシェーダなしに普通のレンダリングをしてしまうところの通りである。



アメリカ人とかはこっちの方が好きそうだが…

これでは日本の(大きな)子供たちには受けが悪いだろう。そもそも現在のレンダリング結果とMMD本体におけるレンダリング結果の違いにも似たような部分を感じていることだろう。

というわけでMMDにはどうやらデフォルトでトゥーンレンダリングの機能が搭載されているようだ。

なお、この辺の表現に関しては、ゲームにおいてはアイドルマスターが神なので

<http://www.4gamer.net/games/105/G010549/20100903012/>

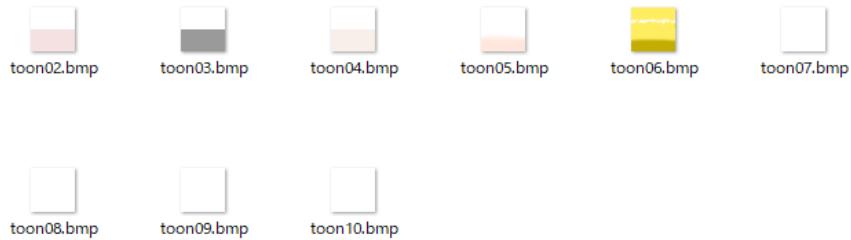
の記事を見ておいた方がいいよ。

ちなみにニコ動でもいまだに動画が見れるよ

<http://www.nicovideo.jp/watch/1306301817>

まあ、2010年の時点では革新的ではあったんだけど、わりかし今だと当たり前な感じ。でも学生さんに説明するならこれくらい枯れた技術の方がちょうどいいかもしれない。

とりあえずミクミクダンスをダウンロードして、中身のDataフォルダを開いてみると…



この通りトゥーンの中にはたくさんの bmp が用意されています。で、PMD データはどう対応しているのかと言うと…

```

material[3].alpha          3F800000
material[3].specularity    41700000
material[3].specular_color[0] 3ECCCCCD 3ECCCCCD 3EC
material[3].mirror_color[0] 3DCCCCCD 3DCCCCCD 3DC
material[3].toon_index      02
material[3].edge_flag       01
material[3].face_vert_count 000013C8
material[3].texture_file_name[0] 00 00 00 00 00 00 00
material[3].texture_file_name[16] 00 00 00 00

```

トゥーンインデックスとこのファイル名が対応しているんですねえ…さて、というわけでもた新しくレジスタを用意しましようかね…テクスチャレジスタ 3 番(+3)ですね。ちなみに toon00 の場合はトゥーンを使用しません。どうしましようかねえ…トゥーン用のピクセルシェーダとそうでないシェーダを分けなければいけないか、マテリアルごとにトゥーン有り無しフラグを作るか…どうしましょうか。

まあ…分かりやすいのはフラグですね。とはいってピクセルシェーダでの分歧は…分歧は…で、僕がピクセルシェーダにおける分歧処理を嫌がるのは理由があって

<http://www.js.utsunomiya-u.ac.jp/pearlab/ja/18/gpu/>
<https://pc.watch.impress.co.jp/docs/2007/0326/kaigai346.htm>

に書いてあるように、GPU ってのは「たくさんの」「単純な仕事」を高速にやるように設計されているので、分歧は可能な限りない方がいい。

もちろん、パイプラインステートの切り替えのコストもそれなりにあるのだが、すべてのピクセルで分歧するよりかはマシかなーって思います。ちょっと嫌なのは同じようなコードが増えてしまうという事なんですね…

そこでシェーダの中身を関数化します。シェーダも当然のように関数化できるんですよ～。つまり今まで書いてたピクセルシェーダのコードをそっくりそのまま別の関数にコピーします。

```

//標準ピクセルシェーダ
float4 BasicPS(Output data) :SV_Target
{
    return float4(GetColorForBasicPS(data),alpha);
}

ひとまずはその状態で元通りに動くのを確認してください。その後にコピペしてトゥーンの
ピクセルシェーダを作ります。

//標準トゥーンピクセルシェーダ
float4 ToonPS(Output data) :SV_Target
{
    //ここにトゥーンのコードを書きまくります
    return float4(GetColorForBasicPS(data),alpha);
}

```

LookUpTable

ひとまずはトゥーンテクスチャの受け取り先を書きます。

```
Texture2D<float4> lut:register(t3);
```

レジスタ3番にLUTテクスチャを設定します。LUTというのはLookUpTableの略で、よくCLUT(Color LookUp Table)と書かれるものです。

まあ、何が」というと色の濃淡を独自の階調にしたい場合に使用するもので、toon01.bmpのようなテクスチャになっています。これをどう使用するのかと言うと例えばディフューズの明るさが出たとしますよね？これは0~1ですね？

この明るさをLUTを使用して、階調を操作します。つまり



のような画像の場合0.5付近で0と灰色に分かれています。通常であればゆるやかに0→1となるところをこれにてパキッと変えています。これによりアニメのような結果を得るために…それがLUTです。もちろんアニメのような効果ばかりに使用するわけではなく様々な事に使用されます。パレット的な事にも使用しますが、今回はV方向をbrightnessに合わせてやるところからやってみましょう。

既にシェーダ側の設定ができているので、今度は渡す側…CPU側のコードを書きましょう。まずは3番目のテクスチャを使用する用意です。

```
auto toonRSIndex=dxbuffManager.AddParameterAndRangeForRootSignature(D3D12_SHADER_VISIBILITY_PIXEL, D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 3);
```

3番レジスタに割り当てる準備です。次にテクスチャをロードします。

```
//トゥーンテクスチャのロード  
_toonTextures.push_back(nullptr);  
  
for (int i = 1; i <= 10; ++i) {  
    char filepath[] = "pmd/toon00.bmp";  
    sprintf(filepath, "pmd/toon%02d.bmp", i);  
    _toonTextures.push_back(textureCreator.LoadFromFile(filepath, 3, toonRSIndex));  
}
```

0番は使用しないのでそのままです。

あとはループしながら

```
if (mat.toonIdx > 0) {  
    _commandList->SetGraphicsRootDescriptorTable(_toonTextures[mat.toonIdx]->GetRootParameterIndex(),  
    _toonTextures[mat.toonIdx]->GPUDescriptorHandle());  
}
```

ですね。さて、このLUTを適用するところですが、例えば

```
float3 c = lut.Sample(smp, float2(0, dbright));
```

等と言う風にすると…



うへん。確かにぺたーんとしてしまいましたが、ちょっと白すぎやしませんかね…。ということでちょっと調整してみましたが…

なんかうまくいかない



へー?



あれー?

いや、こんな風にしてるんですけどね?

```
float3 c = float3(dbright, dbright, dbright); //
```

```
if (existToon)
```

```
    c = lut.Sample(smp, float2(0, dbright));
```

```
float3 color = existTex ? tex.Sample(smp, data.uv)*max(c, ambient) : max(diffuse*c, ambient);
```

うまくいかんもんだなあ…。と思ったら、良くテクスチャを見たら、上が明るくて下が暗いので

...

```
c = lut.Sample(smp, float2(0, 1.0f-dbright));
```

とすればいいんだね。



でもなんかおかしい…何が違うんだろう…
で、いろいろと調整しつつどうにもうまくいかないので、カラー成分を外して、LUTのみでレンダリングしてみた…。



ここでおかしなことに気が付きました。緑色の髪の色のトーンが赤いのです。普通にやつたらこれはあり得ない…。ということで一度仕様を見直してみました。

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

```
『BYTE toon_index; // toon???.bmp // 0.bmp:0xFF, 1(01).bmp:0x00 … 10.bmp:0x09』
```

な、なんだってー!!!!

なんでそういう仕様にしたよ…
はいわかりました。1ズレてますね。インデックス+1が名前に反映される仕組みです。



はい、それっぽくなりました。ちなみに靈夢さんも…
それっぽい感じです。



エッジ(輪郭線)

ちなみに輪郭線に関してはリムライトと反対の事をやってやればいいのです。例えばリムライトは視線ベクトルとの内積が0 近辺の時に明るくするものでした。後はこれを逆にすれば輪郭線を出せます。この方法が一番手っ取り早いんですが、

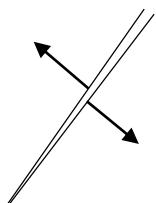
```
float dotVN = dot(-vray, data.normal);  
とでも書いておいて  
if (abs(dotVN) < 0.3f) {  
    color = float3(0, 0, 0);  
}
```

のように書いておけば、何気に輪郭線入ります。



でもスカートの裾に出ていません
なぜでしょうか？

このやり方は、シンプルなだけに弱点があります。スカートのような物体の場合モデルの作りは

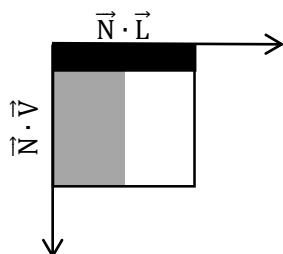


こんな風になってて裾の先の法線が存在しない状態になっているんですね。という事で輪郭線が出てきていません。

そう、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で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んじゃないでしょうか。

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」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)。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



ぐちやぐちやですが、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。そろそろこれ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。

とはいって、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, 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,
min16uint2 boneno : BONENO, min16uint 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…？



これはいったい…

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] | 3F1090B0 3F1090B0 3F1090B0 |
| 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。さっ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
```

やってる事はいいんですが、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 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



こういうことだ

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.jaigiri.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。

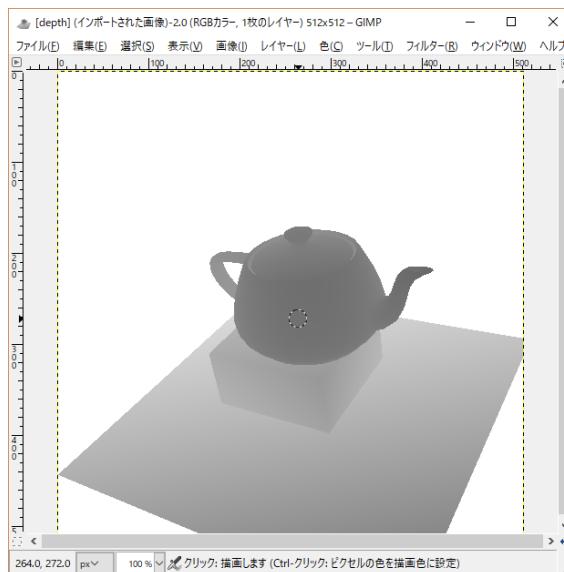
さて、10の周囲が10であればこのフィルタをかけた結果は0になりますね？ $10+10+10+10+(-4)*10=0$ です。

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.DirectX12>

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。



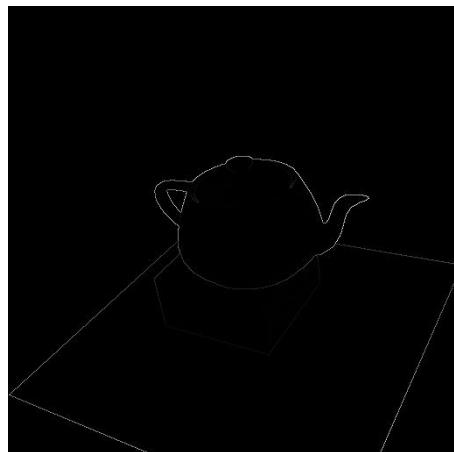
こうなりますね？これでフィルター→汎用→コンボリューションって押すと



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…

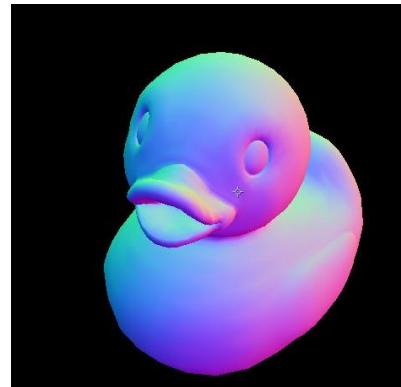


おっ!?

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。

ガチでマルチパスレンダリング

ここまでやろうとしてくると、ガチでマルチパスレンダリング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(深度バッファ法)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/ピッ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、例によつて 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  
}
```

これを正しく間違いなく適用してやると



ひとまず輪郭線、トゥーンについてはこんな感じで終了です。

そのうちMMDにはトゥーン拡張みたい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…

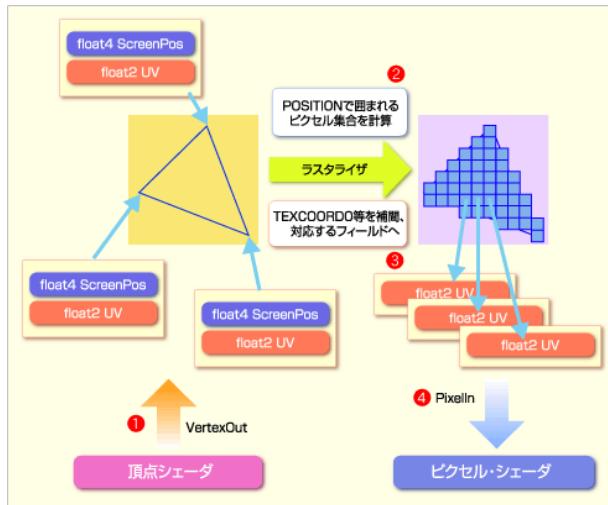


流石にエッジが汚いんですねえ…

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。つまり塗るか塗らないかの中間のやつが必要になるわけです。線で描くとこう

拡大



左寄り右のほうがギザギザ感が少なくなっているのがわかるだろう?

このように、これ(ギザギザ感)をマシにするには「アンチエリアシング」という技術を用います。これにはメリットとデメリットがあります。

メリット: エッジがきれいになる

デメリット: クソコスト(時間)がかかる。G-Buffer が使えなくなる)

で、ちょっと不穏な事に DirectX11 の AA と比べて DirectX12 の AA は難しそうだのなんだの言われてるのかなり心配なんですが…やってみましょう。

とりあえず DirectX11 の手順で考えると

まず初期化の部分で

① CheckMultisampleQualityLevels を使用し、可能なクオリティをチェック(しないとエラー)

② ①のチェックで「使えるクオリティの最大値」を決定する

で、使用できるクオリティの最大値が得られたらそれに合わせて、テクスチャの設定を変更し

ます。具体的にはテクスチャ/バッファを作る部分において
SampleDesc の指定をする部分があるが、ここに対して非アンチエリならば
SampleDesc.Count=1
SampleDesc.Quality=0;
とするだろうが、ここがアンチエリの場合、最初のチェックで得られた適切なカウントとクオリティを設定する。

また、レンダーターゲットビューの設定で
ViewDimension=D3D11_RTV_DIMENSION_TEXTURE2D
から
ViewDimension=D3D11_RTV_DIMENSION_TEXTURE2DMS
に変更する。MS はモビルスケルなどではなく、マルチサンプリングの略である。
また、この部分に関しては勿論深度バッファビューも書き換えるべきだし、それに対応しているシェーダリソースビューも書き換えるべきである。

非常に面倒。

さて、これが DirectX12になるとどれほど面倒になるんだろうね。まあ、男は度胸、なんでもやつてみるのさ

DirectX12での実装

やれるのか…？

まず CheckMultisampleQualityLevels を探しましたが、そんな関数ありません。

その代わりに

CheckFeatureSupport という関数を使うようです。クソが!!なぜ名前を変える!!!

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn788653\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn788653(v=vs.85).aspx)

当然の英語!!!!

使い方としては第一引数に欲しいデータの種別を入れると第二引数にそのデータが返ってくるらしい。

今回はマルチサンプルのクオリティを知りたいので

D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS

を指定します。

そうすると第二引数に帰ってくるのですが、こいつは void ポインタなのでそれ用の型を用意しておきます。型は

D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS です。

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn859387\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn859387(v=vs.85).aspx)

あれ?まさか DX11 よりお手軽?

と思って

```
//マルチサンプルクオリティ最大値をチェック  
D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS qualitylevels = {};  
result = _dev->CheckFeatureSupport(D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS, &qualitylevels, sizeof(qualitylevels));
```

なんて書いてみました。結果は

E_FAIL でした。甘くねえよな、人生って奴は…。今一度ヘルプを見てみましょう。

下の方に書いてあるサンプルも見てみます…。

ははあ…ということはあれか、この関数は DX11 の時と同じで、結局「チェック」しかしてくれない」と見た。

つまり、D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS の中の値をいじくりまわしながら、合致する奴を見つけなきゃいけない…と。やっぱり面倒なんだね。

設定すべきは 4 引数。

というわけで再挑戦。試しにこう書いてみた。

```
//マルチサンプルクオリティ最大値をチェック  
D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS qualitylevels = {};  
qualitylevels.SampleCount = 1;  
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));
```

今度は S_OK が返ってきた。ちなみにあえて NumQualityLevels には何も入れてないが、これは 1 が入っている。SampleCount が大丈夫な数値なら 1 が入っているようだ。逆にダメなら 0 が入っているっぽい。

ということで、2 にしてみて S_OK が返ってきたらその先の処理もやってみよう。

そして、実験を繰り返した結果 8 くらいまで大丈夫だったのでその設定で

```
_dxgi->CreateSwapChainForHwnd
```

を呼ぶと失敗。色々とやってみましたが失敗しました。チェックは OK だったのに何故!? 正直良く分からぬいのだ。いや、ここで成功してもらわんと困るんですが…

という事で海外のサイトにまで手を伸ばしてみましたが…

<https://github.com/Microsoft/DirectXTK12/wiki/Simple-rendering>

のいちばん下の方にこう書いてあります。

『Technical Note

The ability to create an MSAA DXGI swap chain is only supported for the older "bit-blt" style presentation modes, specifically `DXGI_SWAP_EFFECT_DISCARD` or `DXGI_SWAP_EFFECT_SEQUENTIAL`. The newer "flip" style presentation modes `DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL` or `DXGI_SWAP_EFFECT_FLIP_DISCARD` required for Universal Windows Platform (UWP) apps and Direct3D 12 doesn't support creating MSAA swap chains--**attempts to create a swap chain with `SampleDesc.Count > 1` will fail**. Instead, you create your own MSAA render target and explicitly resolve to the DXGI back-buffer for presentation as shown here. See the [SimpleMSAA12](#) sample.』

うん、お前ふざけんな。

意味わからんかもしけんが、to create a swap chain with `SampleDesc.Count>1` will fail.と書いてある。つまり、

『サンプラーデスクのカウントを1以上にしてスワップチェインを作成しようとする試みは失敗するだろう。』



くっそ…予言者みたいになこと言いやがって!!的中したよ!!この野郎!!!だいぶ悩んだよ!!だいたいこういう大事な事を公式のWebサイトにも、日本のWebサイトにも情報がないってどういうことなの?もうホントにDX11でやれてたことを悉く潰して来よるなア…。なんなんだ。

俺が英語見て「あつ…察し」ができるほどに英語できるからすぐに分かったモノの…。分かるかこんなもん!!

さて、ではどう対処すればよいのだろうか…

Instead, you create your own MSAA render target and explicitly resolve to the DXGI back-buffer for presentation as shown here.

と書いてあります。

「まあできないんで自分でMSAAのレンダーターゲットを作つて、明示的にDXGIのバッファを割り当ててちょ。」

そして最後にこう書いてあるわけだ。

「See the [SimpleMSAA12](#) sample.」

ほほう…。まあつまり今回の問題を解説するとですね…今までレンダーターゲットの一つは元からディスプレイ用にあるものを使つていたけど、自分でテクスチャ作つて、それを割り当てるようにしてちょっとすことよね。

しかしこのサンプル…UWPなんんですけど、大丈夫なんすかねえ…？まあ、やり方さえ分かればいいんでサンプルの環境には大して興味がないんですが…しかし困ったねホント。

そしてこのサンプルはクソでかれいで落とすときは注意しよう、な!!

で、ソースコードを読んでみました。

とりあえずポイントとなりそうな部分をピックアップしてみました。

- スワップチェイン生成時にはCountとQualityは1,0である
- ひとまずバッファフォーマットも通常通り
- CreateCommittedSubresourceでレンダーターゲットやら深度バッファを作つている
- ↑のバッファにてサンプルカウントの指定を行つている(クオリティ=0?)
- レンダリング時にResolveSubresourceを使用している

わざとか!!わざとややこしとるんちゃうんか!?

取り乱して済まんが、こんなにややこしくする理由はあまりないと思うんだけどなあ…素直にバッファをMSAA対応させれば済む話ちゃうんかともうねアホかとバカかと。

まあ明らかに最大のポイントはResolveSubresourceやろなあ

ちなみにZeroGram氏のHPを見てみると…

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

「SwapChain 作成時にマルチサンプルを指定するとエラーになります。マルチサンプルを使うには、別にレンダーターゲットを作成、描画結果を ResolveSubresource 関数を使用し通常のテクスチャに変換する必要があります（変換先を SwapChain のフレームバッファに）。またマルチサンプルのレンダーターゲットをテクスチャにして SwapChain のフレームバッファに描画する方法もあります（たぶん、ResolveSubresource の内部処理？）。

ResolveSubresource を使用するには、テクスチャの状態を

D3D12_RESOURCE_STATE_RESOLVE_DEST、D3D12_RESOURCE_STATE_RESOLVE_SOURCE に変更します。」

等と書いておりました。

最初にこのサイトを見てればよかつたよ…。

さて、ともかく分かっている事は…

1. スワップチェインを作った時点で内部にレンダーターゲット(RT)は作られている
2. スワップチェイン内部の RT に対してマルチサンプリングは使用できない

これに対する解決策として

1. レンダーターゲット用のテクスチャ別にを用意し、そこにレンダリングする
2. ↑のテクスチャを ResolveSubresource で rendertargets(i) にコピーする

を考えるという事だ。なお、ZeroGram 氏の書いてある RESOLVE_DEST,RESOLVE_SOURCE に関してですが、これはどうもバリアの時のパラメータのようです。

これは Microsoft のサンプルでも同様でした。

ともかく MSAA 用レンダーターゲットを作らねばならぬようだ。面倒だね

MSAA 用レンダーターゲットを作る

とはいってこれも相当苦労した。

ひとまず、通常通りテクスチャ作るところまでは同じですが、これをマルチサンプリング可の指定およびレンダーターゲット可の指定をしなければならない。

そしてまた罠がはられていたという…

レンダーターゲットとして使うので flags に D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET を追加する。追加ってのは他にフラグがあった場合 OR 演算で追加する。ここはもう言わずともわかるよね。分からぬ人は正直に言いなさい。

で、それ罠の部分ですけど SampleDesc.Quality に 1 を入れると失敗確定。←これすぐ一盲点だった。1日費やした…泣きそう。確かにサンプルコードとか ZeroGramさんのコードを見ると直打ちしてるし…。

じゃあ Check で帰ってくる Quality はなんの意味があるのさ…ふざけんな!

というわけで

```
//-----MSAA用レンダーターゲット用バッファ作成-----
D3D12_RESOURCE_DESC msaaResDesc = {};
msaaResDesc.Width = swcDesc.BufferDesc.Width;
msaaResDesc.Height = swcDesc.BufferDesc.Height;
msaaResDesc.Format = swcDesc.BufferDesc.Format;
msaaResDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
msaaResDesc.MipLevels = 1;
msaaResDesc.Flags = D3D12_RESOURCE_FLAG_NONE;
msaaResDesc.Flags |= D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET;
msaaResDesc.DepthOrArraySize = 1;
msaaResDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
msaaResDesc.Alignment = 0;
msaaResDesc.SampleDesc.Count = dx12.GetMSAAParam().samplecount;
msaaResDesc.SampleDesc.Quality = 0; //てめえふざけんな!
```

で、これを元にしながらノックファを作つてみる。ひとまずはこんな感じで

で、あとはデスクリプタヒープとビューを作る。この部分は通常のレンダーターゲットと同様なので作っていきます。

```
D3D12_RENDER_TARGET_VIEW_DESC rtvDesc = {};
rtvDesc.Format = swcDesc.BufferDesc.Format;
rtvDesc.ViewDimension = D3D12_RTV_DIMENSION_TEXTURE2DMS; //マルチサンプリング
```

```
dev->CreateRenderTargetView(
    _msaaTex, &rtvDesc,
    _msaaDescriptorHeap->GetCPUDescriptorHandleForHeapStart());
```

で、これで作ってみました。

結果の検証が結構大変

作ってみたのですが、意外と結果の検証が大変であることが分かりました。知らないと「あ、アンチエリしてる!!」と思っちゃうそう。

ちなみに、



アンチエリしているように見えるでしょ？

でもアンチエリしてないんですよ。なんかスクショ取って貼り付けると勝手にアンチエリつづか／＼イリニアかかってる状態になります。

ちなみにプロジェクトの拡大機能を使ってもアンチエリがかかるので、一見成功したように思えてました。

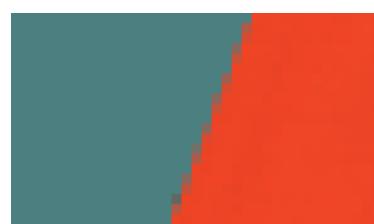
思えてたんですけどねえ…。↑のような罫によって結局は「失敗してた」事がわかりました。どうやって分かったかと言うと Windows 標準の「拡大鏡」とか使うとわかりました。Windows キー & "+"とか "-"とかで拡大縮小表示ができます。そうすると…

こんな感じでした。



あ、これはアンチエリになってないですね

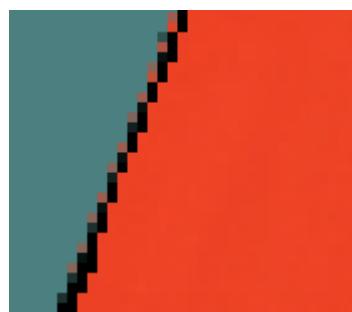
で、原因是デプス(深度)のアンチエリをかけてなかつたからで…。デプスもマルチサンプリングするとこの通り…



アンチエリがかかるっています。クオリティはちょっと「うーん」という感じかもしれませんけどもかくアンチエリはかかるっています。

もし、この結果(MSAA)に不満があるという人は様々な手法を試してみましょう。FXAAだのTXAAだのなんだと思いますので、お試しあれ。

ちなみに、この状態で輪郭線を復活させると、こうなる



あら、きたない!!

よく見ると、確かに輪郭線にもアンチエリがかかるているのが分かるが、モデルのアンチエリの内側に輪郭線表示がされている。

これにより、アンチエリかけたほうが汚く感じてしまう事になるのである。悲しい。とりあえず普通にアンチエリするだけでもかなり苦労したので、ここから先は「今後の課題」としておこう。

ちなみにこういう時に強いのが背面法。シンプルなアルゴリズムであるだけに、こういう時の悪影響をあまり受けにくいという特徴があります。

↓背面法輪郭線＋アンチエイリアス



とりあえず…まあきれいになっている…と思う

まとめ

これまでの事をまとめると…

アンチエイリアスをやりたいかつ DX の機能でどうにかしたいならば

1. MSAA のサンプリング数を設定することができる限界を CheckFeatureSupport で確認
2. スワップチェインを MS にすることはできないので「MSAA 用の RT テクスチャ」を作る
3. それ用の「RT テクスチャ」の SampleDesc の Count を 1～CheckFeatureSupport で確認した最大数のどれかにする
4. SampleDesc の Quality は 0 にしておくこと
5. RT 用テクスチャなので D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET 指定を追加する
6. レンダーターゲットビューを作る際に ViewDimension の TEXTURE2D を TEXTURE2DMS にする
7. 深度もやらないとアンチエリは結局効果がないため DSV も SampleDesc.Count を RTV と同じにする
8. SampleDesc.Quality はやっぱり 0 にします
9. ViewDimension は D3D12_DSV_DIMENSION_TEXTURE2DMS にします
10. ループ内のレンダーターゲットを 2 で得られた MSAA 用 RT テクスチャにする
11. レンダーターゲットをクリアする
12. レンダーターゲットに書き込む

13. 深度/ピッファをクリアする
14. 深度/ピッファに書き込む
15. RT および DSV の書き込みが終わったら、ResolveSubresource を使用します

コード

まず、サンプリングカウントチェック

```
D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS qualitylevels = {};
qualitylevels.SampleCount = サンプル数;
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.isAA = true;
    _msaaparam.samplecount = qualitylevels.SampleCount;
    _msaaparam.quality = qualitylevels.NumQualityLevels;
}
```

MSAA レンダーターゲット用テクスチャ作成

```
/// ピッファの作成
D3D12_RESOURCE_DESC msaaResDesc={};
msaaResDesc.Width=swcDesc.BufferDesc.Width;
msaaResDesc.Height= swcDesc.BufferDesc.Height;
msaaResDesc.Format=swcDesc.BufferDesc.Format;
msaaResDesc.Dimension=D3D12_RESOURCE_DIMENSION_TEXTURE2D;
msaaResDesc.MipLevels=1;
msaaResDesc.Flags=D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET;
msaaResDesc.DepthOrArraySize=1;
msaaResDesc.Layout=D3D12_TEXTURE_LAYOUT_UNKNOWN;
msaaResDesc.SampleDesc.Count=dx12.GetMSAAParam().samplecount;
msaaResDesc.SampleDesc.Quality=0; // 0だという裏を取ろう

D3D12_HEAP_PROPERTIES hp = {};
hp.Type = D3D12_HEAP_TYPE_CUSTOM;
```

```

hp.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_WRITE_BACK;
hp.MemoryPoolPreference = D3D12_MEMORY_POOL_L0;
hp.CreationNodeMask = 1;
hp.VisibleNodeMask = 1;

ID3D12Resource* _msaaTex=nullptr;
result = dev->CreateCommittedResource(&hp,
                                         D3D12_HEAP_FLAG_NONE,
                                         &msaaResDesc,
                                         D3D12_RESOURCE_STATE_RESOLVE_SOURCE,
                                         nullptr,
                                         IID_PPV_ARGS(&_msaaTex));

ID3D12DescriptorHeap* _msaaDescHeap = nullptr;
D3D12_DESCRIPTOR_HEAP_DESC msaaDescHeapDesc = {};
msaaDescHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
msaaDescHeapDesc.NodeMask = 0;
msaaDescHeapDesc.NumDescriptors = 1;
msaaDescHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
result = dev->CreateDescriptorHeap(&msaaDescHeapDesc, IID_PPV_ARGS(&_msaaDescHeap));

D3D12_RENDER_TARGET_VIEW_DESC msaaViewDesc={};
msaaViewDesc.Format=swcDesc.BufferDesc.Format;
msaaViewDesc.ViewDimension=D3D12_RTV_DIMENSION_TEXTURE2DMS;
dev->CreateRenderTargetView(_msaaTex,
                            &msaaViewDesc,
                            _msaaDescHeap->GetCPUDescriptorHandleForHeapStart());

```

深度/ピッファ設定

```

depthResourceDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
depthResourceDesc.Width = WINDOW_WIDTH;
depthResourceDesc.Height = WINDOW_HEIGHT;
depthResourceDesc.Format = DXGI_FORMAT_R32_TYPELESS;//D32_FLOATだと後で使いにくいので
depthResourceDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
depthResourceDesc.DepthOrArraySize = 1;

```

```
depthResourceDesc.MipLevels = 1;
depthResourceDesc.SampleDesc.Count = dx12.GetMSAAParam().samplecount;;
depthResourceDesc.SampleDesc.Quality = 0;
depthResourceDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;
```

ビューも変更

```
depDesc.Format = DXGI_FORMAT_D32_FLOAT;
depDesc.ViewDimension = D3D12_DSV_DIMENSION_TEXTURE2DMS;
depHandle = depDescHeap->GetCPUDescriptorHandleForHeapStart();
dev->CreateDepthStencilView(_depthBuffer, &depDesc, depHandle);
```

毎フレーム描画部分

```
//MSAA用のデスクリプタヒープをレンダーターゲットとする
_commandList->OMSetRenderTargets(1, &_msaaDescHeap->GetCPUDescriptorHandleForHeapStart(),
false, &depHandle);
(中略)
```

```
result = _commandList->Reset(_commandAllocator, _pipelineState);
```

//↑にて、通常描画がすべて終了している

//↓で既にMSAAバッファに描画済みのデータをバックバッファに転送

```
D3D12_RESOURCE_BARRIER barriers(2) =
{
    CD3DX12_RESOURCE_BARRIER::Transition(
        _msaaTex,
        D3D12_RESOURCE_STATE_RENDER_TARGET,
        D3D12_RESOURCE_STATE_RESOLVE_SOURCE),
    CD3DX12_RESOURCE_BARRIER::Transition(
        renderTargets(bbIndex),
        D3D12_RESOURCE_STATE_PRESENT,
        D3D12_RESOURCE_STATE_RESOLVE_DEST)
};

_commandList->ResourceBarrier(2, barriers);

_commandList->ResolveSubresource(renderTargets(bbIndex), 0, _msaaTex, 0,
DXGI_FORMAT_R8G8B8A8_UNORM);
```

```
//最後のバリアーも忘れないよう
_commandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(renderTargets(bbIndex),
        D3D12_RESOURCE_STATE_RESOLVE_DEST,
        D3D12_RESOURCE_STATE_RENDER_TARGET));

_commandList->Close();
dx12.ExecuteCommand();

//GPU側の実行を待つ
dx12.WaitWithFence();
```

といった具合だ。

一応今更感あるが、アンチエリについての記事を貼り付けておこう

<http://www.4gamer.net/games/120/G012093/20121125002/>

まあ確かにこの説明を見ると「奥行き情報のないところでは機能しない」と書かれてあるため、そもそも RT バッファをマルチサンプリング設定しても、深度バッファがそのままなら機能しない」という事だ。

ちなみに↑の記事にも書いてあるが、色の中間色を計算してるので、HDRになった時に正確な色を計算することができない。今後はいつくらいから HDR 主流になるか分からんが、その時代が来れば新しいアルゴリズムが求められるだろう…

ちなみに↑の記事においては TXAA が HDR にもうまく対応できそうとのことなので nVidia の Web サイトに行ってみて研究してはどうだろうか？

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…皆さんもプログラムしてるからお分かりのように、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 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 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

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、ほつとんど同じです。

…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って大きく分けると

- 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。

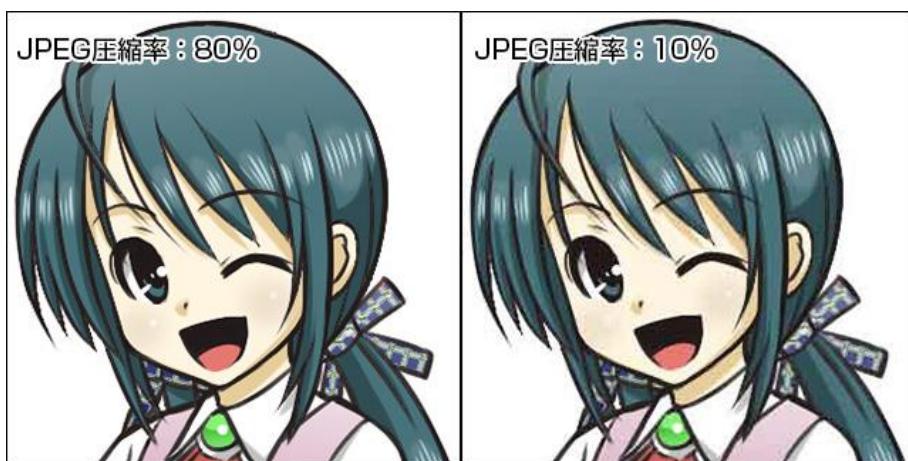
それによりパレット部分を無視すれば圧縮率は理論上1/4となる。

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い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…

分かりづらいかもしれません、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」など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、この手の「ノイズが入る」系の圧縮方式は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 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言うとクソ圧縮なんですよね。とはいっても使われてるのに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 だときりが悪くて、16 にしようとなった時に、人間の識別範囲が広いのが縁だからという事です。

し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 にされてしまいます。 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

$$512 \rightarrow 64$$

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「なくて『品質が〜!!』とか言ってるんですか
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…

効果的な場合

- ドット絵的な場合(<DXTC 等でやると確實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

やらない方がいい場合

- 写実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(どの道大体)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っている場合(インデックスを補間したら何が起こるか…分かるね? 分からない? 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いかな?)

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います。

で、かなり本題から外れてしましましたが実際の DDS ファイルのロードについて話してみようと思います。

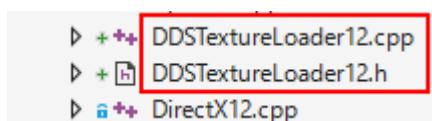
前の時に BMP や PNG をロードする仕組みとして WICTextureLoader12.cpp があったでしょ? あのノリで DDSLoader も用意します。

再び DirectXTex-master の中へ

もう忘れちゃってるかもしれません、DirectXTex-master の中を見ます。

LoadWICTextureFromFile に相当するものが DDS にもあればいいですね。そうすると DDSTextureLoader というフォルダがありますので中を見ると DDSTextureLoader12.cpp というファイルがありますね。bingo!!! だったらいいんですけどどうでしょうか。

まあ、足踏みする時間などないので、こまけえ事は考えずにプロジェクトに追加だ!!



そしてビルドしてみる。まあ通るか。通るのか。まあまだ利用していないし、もう一つ言うと DDS のプログラムはプログラムで完結してるし、WIC よりかは使いやすそうだ。

まずはヘッダファイルを見てください。

```
// Standard version

///メモリ上のDDSファイルをロード(デコード)してID3D12Resourceとして返す。
///@param d3dDevice(In) デバイス
///@param ddsData(?) 対象データのアドレス(エンコードされた状態)
///@param texture(out) テクスチャオブジェクト(こいつがメインで欲しいもの)
///@param subresources(out?) サブリソースデータ受け取り用配列の参照
///@param maxsize 受け取るべき配列のサイズ(…だと思う)
///@param alphaMode アルファモード受け取り用?
///@param isCubeMap これも受け取り用かな?キューブマップ用?
///@return 読み込み(デコード)結果のフラグ
HRESULT __cdecl LoadDDSTextureFromMemory(
    _In_ ID3D12Device* d3dDevice,
    _In_reads_bytes_(ddsDataSize) const uint8_t* ddsData,
    size_t ddsDataSize,
    _Outptr_ ID3D12Resource** texture,
    std::vector<D3D12_SUBRESOURCE_DATA>& subresources,
    size_t maxsize = 0,
    _Out_opt_ DDS_ALPHA_MODE* alphaMode = nullptr,
    _Out_opt_ bool* isCubeMap = nullptr);

///DDSファイルをロードしてID3D12Resourceとして返す。
///@param d3dDevice デバイス
///@param szFileName ファイルパス
///@param texture(out) データ受け取り用ID3D12Resourceオブジェクト
///@param ddsData(?) 対象データのアドレス(デコード済みデータ配列)
///@param subresources(out?) サブリソースデータ受け取り用配列の参照
///@param maxsize 受け取るべき配列のサイズ(…だと思う)
///@param alphaMode アルファモード受け取り用?
///@param isCubeMap これも受け取り用かな?キューブマップ用?
///@return 読み込み(デコード)結果のフラグ
HRESULT __cdecl LoadDDSTextureFromFile(
    _In_ ID3D12Device* d3dDevice,
```

```

_In_z_ const wchar_t* szFileName,
_Outptr_ ID3D12Resource** texture,
std::unique_ptr<uint8_t[]>& ddsData,
std::vector<D3D12_SUBRESOURCE_DATA>& subresources,
size_t maxsize = 0,
_Out_opt_ DDS_ALPHA_MODE* alphaMode = nullptr,
_Out_opt_ bool* isCubeMap = nullptr);

```

コメントはワシが書いたので、実際には日本語のコメントなどないっ!!!

ちなみに_In_だの_Out_だのについては

<https://msdn.microsoft.com/ja-jp/library/hh916382.aspx>

を見て理解してください。

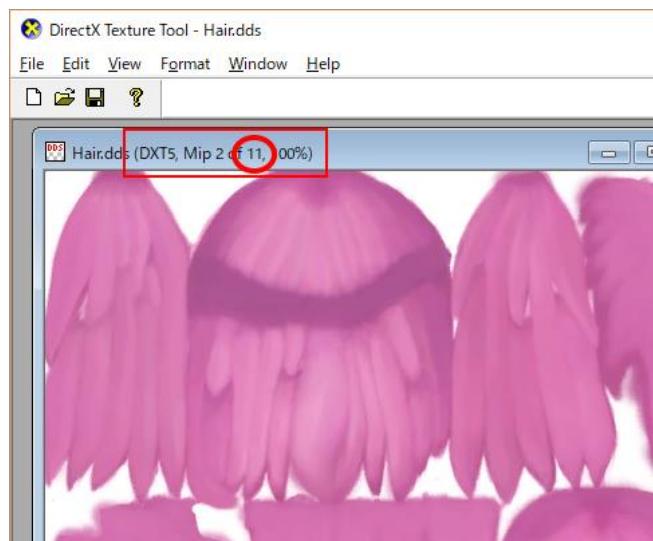


まずこれを実装しようとして最初に直面した問題が…

まあ僕は WIC と同じような感じで行けるだろーって思ってたんですよ。実際。そしたら罠と言うか、僕がただ単に DDS の特性を忘れてただけとも言えるのですが、

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~~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った



ご覧のありさまだよ!

という事で、2枚目以降のテクスチャがうまくできていないようです。今の所

- テクスチャデータの読み込み
- テクスチャデータの生成

はうまくいっているようです。ですので『転送』に失敗していると思います。フェンスカリリアの失敗で2枚目以降が不正になってるのではないかと思うのですが…。

ちなみに注意点としては DDS 形式の場合 R8G8B8A8 的な形式は間違いで、それ用のフォーマットを指定してやる必要があります。フォーマットに関しては読み込み時に分かってるので、それを代入することになります。

クソ UpdateSubresources

さて、理屈から言うと、ここまでやればすぐにでも終わったはず…なのであるが、またクソ仕様のために今回は2週間ほど悩んだ。授業期間中じゃなくてよかった…本当に…良かった。

で、俺の2週間をつぶされた理由が本当にクソムカつくんですが、UpdateSubresources のク

ソ仕様なんですよね…



ともかく、少なくとも現バージョン(10.0.15063.0)においては、UpdateSubresource がまともに機能していない(ミップありの 2 枚目以降の拳動が怪しい)…これは間違いない!(キリッ)

サンプルコードも 1 枚読み込んだらそれで終わっている(複数の Mip つき DDS を連続で読み込むことはやってない)しまあはっきり言って検証されているのかどうかも怪しい。

MS のサンプルコードが寄ってたから UpdateSubresources の方ばかり使ってるから、俺もそれでやってたけど、結局それが悪さしてるっぽい。WriteSubresource で正解。ただし恐らく最新 WDK では deprecated 関数なので、バージョン上がつたら UpdateSubresources に変更すべきなのだろう少なくとも 10.0.15063.0 における UpdateSubresources の拳動はおかしい。

という見切りをつけるまでに 2 週間かかってしまったのだ!!!!許さんぞ!!!!

ちなみにこれ、10.0.1607 以降のバージョンで治ってるかどうかの保証はないっ……!!(現在の最新バージョンは 1709 だが…まだ試してはいけないっ…!!)

クソに対する対処法

ともかく、15~16 では少なくとも GeForce GTX 100 では動かないことが分かったので、もうここはこのまま突っ走るのをあきらめて考えました。このそびえ立つクソに対する僕の対処法は…やっぱり WriteSubresource を使う事でした。

WIC の時と同様、まずは Write 用にヒープの指定をやり直します。

DDSTextureLoader のヒーププロパティの指定を

```
CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
```

ではなく

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

にして、今度は UpdateSubresources を使うべき部分をこうします。Write は 1 枚ずつしか書き込めませんので以下のように for ループを使います。こういう使い方は久しぶりですね。インデックスを WriteToSubresource の第一引数に使用するため、こういう場合は仕方ないですね。

```
for (int i = 0; i < subreses.size(); ++i) {  
    result = textureBuffer->WriteToSubresource(i, nullptr, subreses[i].pData,  
    subreses[i].RowPitch, subreses[i].SlicePitch);  
}  
dx12.GetCommandList()->Close();
```

Box なんていいらんかったんや…ということで nullptr にしています。

はい、そこまでやれればミップ付き DDS であっても



はいこの通り

ちょっと気になるのは色がくすむんですよね~。これは元からある問題でもあるんですが…

ちなみにテクスチャカラーのみ表示すると…



このように違和感はあまりない…かな?

これに通常のシェーディングを行うと



もうすでに顔の色に違和感を覚えますね…なんだろうね
あんまり考えても仕方ないので、ちょっとここは考えずに先に進みましょう。

クソまみれの中で得られたもの

クソクソ言うとりますけれども、まあ失った2週間はただ失ったわけではない。クソと向き合っているうちに DirectX12に対する理解を深められたし、ツールなどに対する知見も得られました。

そうは言っても高すぎる代償ではあったけどね。

はい、得られたものは何かというと、あの顔以外真っ黒になる原因を調べようとしたときに、そもそもテクスチャ情報が GPU に渡っているのか？という疑問を持ちました。

そうなってくると必要なのは計測や調査に使用するソフトウェア。

昔(DirectX9 時代)であれば PixForWindows と言うものがございまして、これがレンダリング状況やそれにかかったコストを計測してくれたものでございます。

で、DirectX12 の SDK の中にはそれが入っておりませんでしたので、DX9 の PixForWindows を使ってみましたが、落ちました…というか動作が停止しました。

恐らくシグナルとか、フェンスとかそのあたりで固まっているようで、原因も分からぬし、結論としては古いツールは使えないという結論に至りました。

という事で、PixForWindows DirectX12 で検索すると MS の Github にそれがありました。ありましたか、はっきり言って使い物になりませんでした…。まともな情報が得られない…。

ということで見つけたのが

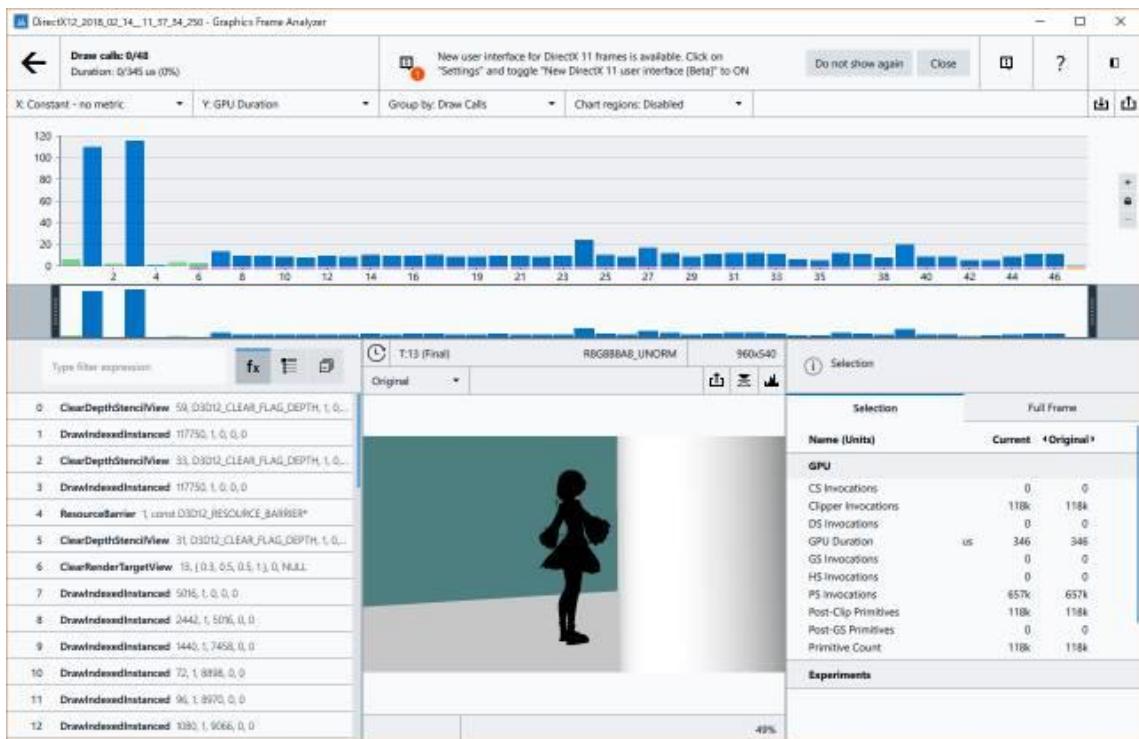
Intel Graphics Performance Analyzer

というツールです。フリーなのかどうなのはちょっと分かりませんが、役に立ったしいまだにお金も請求されてませんし、評価期間過ぎたから更新料などとも言われてません。まあ二週間なので、これが 1 ヶ月後には言われるかも知れませんが…

<https://software.intel.com/en-us/gpa>

あ、FreeDownload って書いてますね。これを使わない手はないですね。

起動して自分のプログラムにアタッチすると…



こんな感じでコストだの何だのが表示されてボトルネックとかに対処しやすくなります…
これで段階的に見ていったのですが、結局テクスチャは真っ黒でした。

で、転送されていないかと言うとそうではなく、転送されたテクスチャが真っ黒だったのです。つまり、サブリソース(ロード後の生データ)を、テクスチャへ書き込むのが間違っている事が分かりました。というわけで `UpdateSubresources` が犯人であるとあたりを付けることができ、`WriteSubresource` で対処するという結論に至りました。

というわけで、闇雲に思い悩むより先に「計測する」これは大切な思想なので覚えておきましょう。

闇雲に思い悩むより先に「計測する」

これを今回は覚えて帰ってください。僕の上司の言葉で「ツマミをいじっても問題は解決せんへんぞ!!まずは何が問題なのがを正確に見極めろ!!」ってのがありました。これは僕もその通りだと思います。

更なるクソの中に潜ってみた結果…

幸いと言えるがどうかわかりませんが、UpdateSubresource のコードは d3dx12.h の内部にソースコードがインラインで記述されているために中身の構造、処理の手順を知ることができます。

さて、潜ってみたところ、キーになる関数がいくつか…

- D3D12Device::GetCopyableFootprints
- D3D12CommandList::CopyTextureRegion
- D3D12CommandList::CopyBufferRegion

が見つかりました。いちおうそれぞれ

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986878\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/dn986878(v=vs.85).aspx)

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn903862\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn903862(v=vs.85).aspx)

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn903856\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn903856(v=vs.85).aspx)

ですね。まあ別に見なくてもいいのですが、やたら面倒な事をやっている。

ちなみに Copy○○Region というのは範囲を指定して Intermediate(中間バッファ)から、最終的に持っていくリバッファにデータをコピーしています。ちなみにどちらもバッファです。

ちなみに、Texture と Buffer とありますが、これは Intermediate のヒープ指定時に Buffer の指定を行っていれば Buffer になるし、テクスチャの指定を行えば Texture 側が使用されます。なお、ここでは Buffer を使用しています。これは後程説明します。

元のデータはと言うと、この手前で Intermediate をマップして、普通に memcpy 的な事をやっており、コンスタントバッファの変更の時と同じようなことをやっています。

さて、ここで一つ疑問が湧いてくるのではないかと思いませんか…

Intermediate(中間バッファ)って何で必要なの!?



まあ、そうなるわな…

なんでなんでしょうね？

そこでマイクロソフトのいわゆるホワイトペーパーを見てみましょう。

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn899215\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899215(v=vs.85).aspx)

いつも通り英語ではあるんですが…

『Uploading 2D or 3D texture data is similar to uploading 1D data, except that applications need to pay closer attention to data alignment related to row pitch. Buffers can be used orthogonally and concurrently from multiple parts of the graphics pipeline, and are very flexible.』

自動翻訳すると…

『2D または 3D のテクスチャデータをアップロードすることは、1D データをアップロードするのと似ていますが、アプリケーションでは行のピッチに関連するデータの配置に細心の注意を払う必要があります。バッファは、グラフィックスパイプラインの複数の部分から直交して同時に使用でき、非常に柔軟です。』

まあ、言いたいことは分かるんだが、まだ核心ではない気がする。ちなみに「直交」ってのは「かぶらない」程度に理解しておけばいいでしょう。さらに

『Applications must upload data via **ID3D12GraphicsCommandList::CopyTextureRegion** or **ID3D12GraphicsCommandList::CopyBufferRegion**. Texture data is much more likely to be larger, accessed repeatedly, and benefit from the improved cache-coherency of non-linear memory layouts than other resource data. When buffers are used in D3D12, applications have full control on data

placement and arrangement associated with copying resource data around, as long as the memory alignment requirements are satisfied.]

お?気になる関数名が出てきましたね?自動翻訳してみましょう。

「アプリケーションは、ID3D12GraphicsCommandList::CopyTextureRegion または ID3D12GraphicsCommandList::CopyBufferRegion を介してデータをアップロードする必要があります。テクスチャデータは、より大きく、繰り返しアクセスされ、他のリソースデータよりも改善された非線形メモリレイアウトのキャッシュ一貫性の恩恵を受ける可能性が非常に高い。」バッファが D3D12 で使用される場合、アプリケーションは、メモリの配置要件が満たされている限り、リソースのデータをコピーすることに関連するデータの配置と配置を完全に制御します。]

ああはい、そうですか…。

Copying

D3D12 methods enable applications to replace D3D11 [UpdateSubresource](#), [CopySubresourceRegion](#), and resource initial data. A single 3D subresource worth of row-major texture data may be located in buffer resources. [CopyTextureRegion](#) can copy that texture data from the buffer to a texture resource with an unknown texture layout, and vice versa. Applications should prefer this type of technique to populate frequently accessed GPU resources, by creating large buffers in an UPLOAD heap while creating the frequently accessed GPU resources in a DEFAULT heap that has no CPU access. Such a technique efficiently supports discrete GPUs and their large amounts of CPU-inaccessible memory, without commonly impairing UMA architectures.

Note the following two constants:]

「コピー D3D12 メソッドは、アプリケーションが D3D11 UpdateSubresource, CopySubresourceRegion、およびリソースの初期データを置き換えることを可能にします。1 つの 3D サブリソースの行主要テクスチャデータがバッファリソース内に配置されてもよい。CopyTextureRegion はそのテクスチャデータをバッファから未知のテクスチャレイアウトを持つテクスチャリソースにコピーでき、その逆も可能です。アプリケーションは、CPU アクセスのない DEFAULT ヒープに頻繁にアクセスする GPU リソースを作成しながら、UPLOAD ヒープに大きなバッファを作成することにより、頻繁にアクセスされる GPU リソースにデータを入れるために、この種のテクニックを推奨します。このような技術は、一

般に UMA アーキテクチャを損なうことなく、個別の GPU とその大量の CPU にアクセスできないメモリを効率的にサポートします。』

うん、流石に自動翻訳に無理が出てきたね。僕なりに修正翻訳してみよう。

『D3D12 のメソッドは、アプリケーションが D3D11 における UpdateSubresource、CopySubresourceRegion、およびリソース初期化データ(initial data)と置き換えて考えられます。

1 つの 3D サブリソースの row-major(後述)テクスチャデータがバッファリソース内に配置されてもよい。CopyTextureRegion はそのテクスチャデータをバッファから未知のテクスチャレイアウトを持つテクスチャリソースにコピーすることができ、その逆も可能です。

アプリケーションは、CPU アクセスのない DEFAULT ヒープに頻繁にアクセスする GPU リソースを作成しながら、UPLOAD ヒープに大きなバッファを作成することにより、頻繁にアクセスされる GPU リソースにデータを入れるために、この種のテクニックを推奨します。

このような技術は、一般に UMA アーキテクチャを損なうことなく、個別の GPU とその大量の CPU にアクセスできないメモリを効率的にサポートします。』

はい、ようやく言及されましたね？。Upload ヒープに大きなバッファを作成することによりアクセス効率と言うか、キャッシュ効率を上げようとしているのでしょうか。そしてその大きなバッファは一次元のデータバッファの事だね。

というわけで、いちいち面倒なんだが、その一次元データとして Intermediate を作り、書き込み先は Texture2D として定義されているため、生データは memcpy で Intermediate で書き込み、それを CopyBufferRegion で書き込み先へコピーする。

終わり。

ちなみに row-major というのは column-major の対義語で、データの格納の仕方が行ずつなのか、列ずつなのかという事を言っている。

例えば

(■(a_11&a_12&a_13&a_14@a_21&a_22&a_23&a_24@a_31&a_32&a_33&a_34@
a_41&a_42&a_33&a_34))

という行列があるとして、それを 1 次元の配列に代入する際に

IK(インバースキネマティクス)

地獄中の地獄ここにありつ……!!!

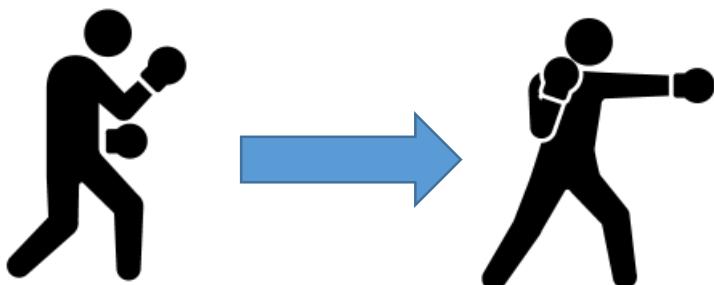
ついにきました。インバースキネマティクス。

もう前回まで影もエフェクトもやっちゃったし、これに入ってもいいでしょ。

このテキスト中最難関なのではないかなーと思います。

そもそもインバースキネマティクスとは何でしょう。知ってる人は知ってるかもしれませんのが、なんていうかな~。日本語で言うと「逆運動学」っていうんですけど、日本語で言われても分かりませんよね。

例えばですね



こう…パンチするわけです。

で、フツーにこのポーズを取らせようとするならば、肩を 45° 回転させて、肘を 90° 回転させてパンチポーズにするわけですね？

で、そういうやり方のことをフォワードキネマティクス(FK)と言います。

これを逆から考えるのが逆運動学で、拳の位置から肘などの位置を逆算するわけです。これをインバースキネマティクスと言います。

例えば



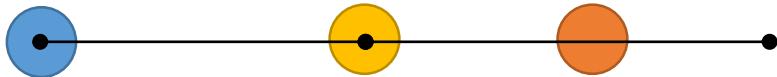
図のように3つの要素があると考えてください。赤が肩で、黄色が肘で、紫が拳にあたると見てください。

そうなると一つの制約が出てきます。

「骨の長さは変わらない」

です。関節の角度や位置は変わっても、骨の長さだけは変わりません。変わったら大変なことになりますよね？

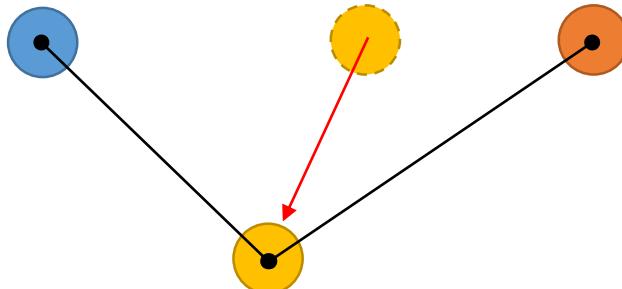
ここで拳を肩に近づけていきます。



さて、ただ近づけただけでは、拳から骨が飛び出てしまい、使い物になりません。でも骨の長さは変えられません…どうしたらいいのでしょうか？

この場合、肘の位置を変更します。つまり

このように肘の位置を「骨の長さが一定になり、拳の位置が所定の位置に来るよう」場所を



変更します。

で、今回は CCD-IK と言うのを使ってやっていきます。色々と IK の種類はありますし、MMD では CCD-IK を使ってるらしいので、それを使います。

CCD-IK とは…

さつきから当たり前のように CCD-IK と言ってますが、コイツは Cyclic Coordinate Inverse Kinematics の略です。長いので CCD-IK と呼ばれています。

様々な Inverse Kinematics の手法の中でこれが MMD に使用されているのは

- 理論が簡単
- 計算が速い(比較的)

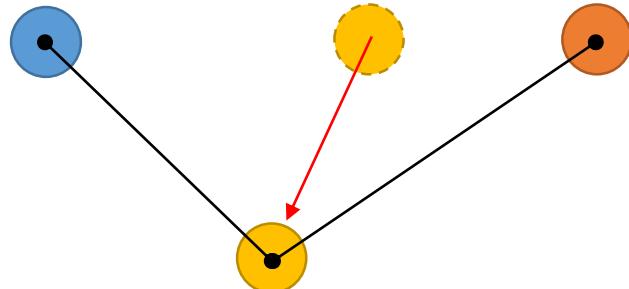
などの理由です。その他の IK を見ましたがよく分かりませんでした。ヤコビアン IK とかワオータニオン IK は直感的に分かりづらかったです。

高校数学だけで IK っぽくしてみよう

えっ!? 高校数学で IK を!?



難しいかもしれません、理屈を分かる上で必要なのでやっていきます。



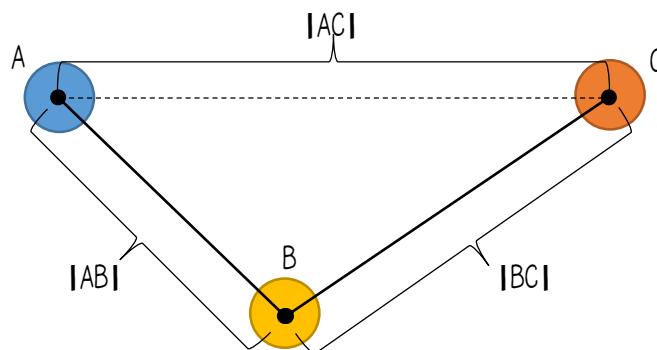
このこの、黄色の位置が知りたい。赤と紫の位置はわかつており、その間の「長さ」は既知であると、そういうわけだ。

仮に赤を A とし、黄色を B とし、紫を C とする。

そうすると、既知のものは

座標 A, C および $|AB|, |BC|$ である。

また、A と C が既知であるため $|AC|$ は $|C-A|$ で求まりますよね? つまりこれも既知。最終的に知りたい $|B$ の値は B の座標。



さて、これだけの材料から B の座標を求めましょう。まず、B から辺 AC に対して垂線を引きましょう。AC と垂線の交点を M とします。

ここまででは良いですかね？

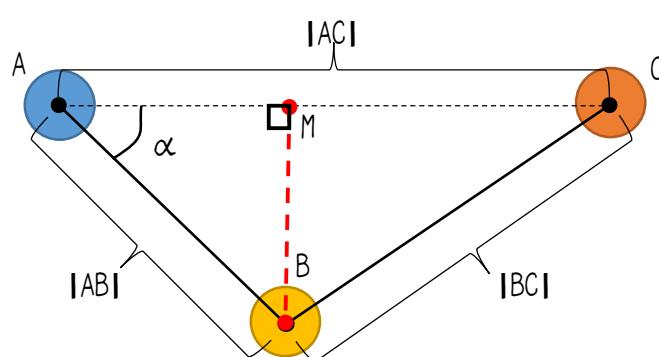
直角三角形となりますから、 $|\overrightarrow{AM}| = |\overrightarrow{AB}| \cos \alpha = \frac{\overrightarrow{AB} \cdot \overrightarrow{AC}}{|\overrightarrow{AC}|}$ となりますね？ただしベクトル \overrightarrow{AB} がわからなければ、まだ使いのにはなりません。早急に点してはダメですよ？

ここで余弦定理を使います。 $|BC|$ を求めるという体で使います。BC は既知ですが、これは後で使うからです。余弦定理は

$$c^2 = a^2 + b^2 - 2ab \cos \theta$$

こんなやつでしたね？今回の三角形に当てはめると

$$|BC|^2 = |AC|^2 + |AB|^2 - 2|AB||AC|\cos\alpha$$



です。

この式の中で既知ではないのは $\cos \alpha$ だけです。ここで先程の

$$|\overrightarrow{AB}| \cos \alpha = \frac{\overrightarrow{AB} \cdot \overrightarrow{AC}}{|\overrightarrow{AC}|}$$

と連立させられないかどうか考えます。おやおや、 $|\overrightarrow{AB}| \cos \alpha$ が使えそうですね？

$$|BC|^2 = |AC|^2 + |AB|^2 - 2|AB||AC|\cos\alpha$$

を変形すると

$$|AM| = |AB| \cos \alpha = \frac{|AC|^2 + |AB|^2 - |BC|^2}{2|AC|}$$

で、この $|AB| \cos \alpha$ ってのは A から M までの距離ですから座標 M は、A からベクトル AC 方向に $|AB| \cos \alpha$ だけ進んだ部分ということになります。ですから

$$M = A + \frac{\overrightarrow{AC}(|AB| \cos \alpha)}{|AC|}$$

さらにこの $|AB| \cos \alpha$ は $\frac{|AC|^2 + |AB|^2 - |BC|^2}{2|AC|}$ と書けるので、

$$M = A + \left(\frac{\overrightarrow{AC}}{|AC|} \right) \left(\frac{|AC^2| + |AB^2| - |BC^2|}{2|AC|} \right) = A + \overrightarrow{AC} \left(\frac{|AC^2| + |AB^2| - |BC^2|}{2|AC^2|} \right)$$

となるので、既知の情報だけで M が表せます。そうすると座標 M が求まりますね？

ただ、求めたいのは座標 M ではなく、B です。どうすれば良いのでしょうか？

$$B = M + \overrightarrow{MB}$$

ですから、ベクトル \overrightarrow{MB} を求めれば終わりです。なお、 $|\overrightarrow{MB}|$ はピタゴラスの定理から

$$|\overrightarrow{MB}| = \sqrt{|AB^2| - |AM^2|}$$

ですね。あとは向きがわかれれば良いのですが、向きはベクトル \overrightarrow{AC} と直行しているため

$$\overrightarrow{AC}^R = \begin{pmatrix} \cos 90^\circ & -\sin 90^\circ \\ \sin 90^\circ & \cos 90^\circ \end{pmatrix} \overrightarrow{AC}$$

で求められます。 \overrightarrow{AC}^R が大きさ情報を持っていると使いにくいためこれを正規化したのと $|MB|$ を乗算します。そうすると座標 B は

$$B = M + \frac{\overrightarrow{AC}^R |\overrightarrow{MB}|}{|\overrightarrow{AC}^R|}$$

と表せます。ここで B を求めりやれりなので、反復するまでもなく、B の座標は確定的に明らか。ちなみに回転方向が正の方向と負の方向が考えられますが、どちらでも正しいです。通常は関節に使うので、どちらかはユーザーが決めることがあります。

まあ、ここまで話ならベクトルと内積と、サインコサインしか使ってないので、頭のレリ！高校生にもなんとか分かる話です。

さて、ここからが本番ですよ。間の関節が複数あるパターンです。それだけで解を求めるのが複雑になってしまいますが、頑張りましょう。

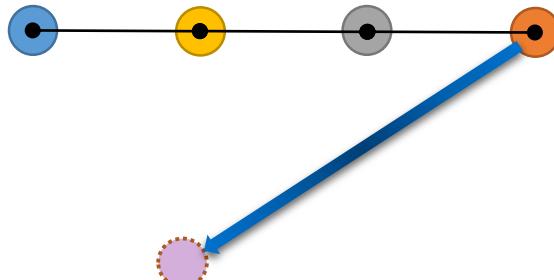
二次元における CCD-IK

さて、実際に CCD-IK をやっていくわけですが、いきなり 3D でやっていくと大変なので、2D に於ける解説からして行こうかと思います。

一文で説明すると

『コントロールポイント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ロート方向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。

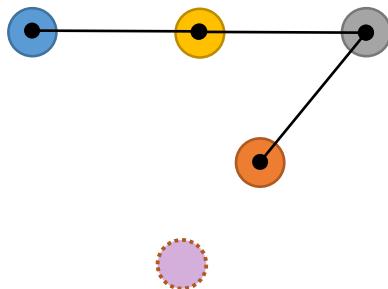


図のように紫を動かしたいとします。なお、赤はIKの影響を受けない端点とします。そうなると間の移動対象は黄色と橙ですよね？

で、先ほどと一緒に各コントロールポイント間の距離は変化しない…と。

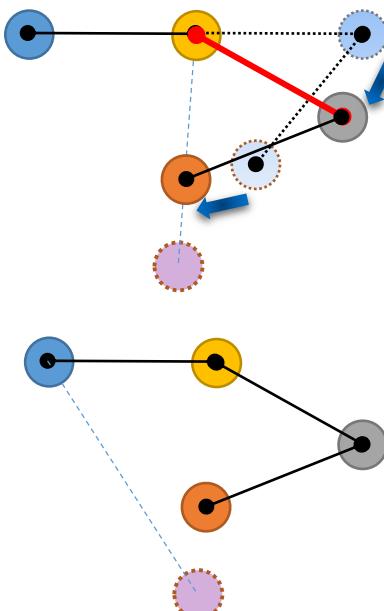
まずは対象コントロールポイントを「目的地に最も近づくように」回転させます。

当然ながら届いてません。かつ、間のコントロールポイントを回転させれば更に近づきそう



ですよね？

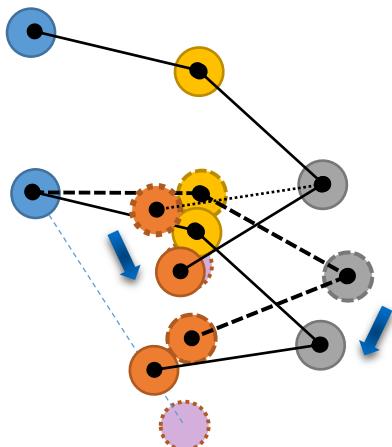
では一つ遡って黄色→橙の線を回転させます。



結果としてかなり近づいていますが、もうひと頑張りです。

今度は赤→黄の線を回転させます。

これで一周したわけですが、まだ離れてますので、この状態でまた末端からやり直します。



これを一致範囲内に入るまで、もしくは繰り返し制限回数を超えるまで繰り返します。
これが CCD-IK(2D)です。

三次元における CCD-IK

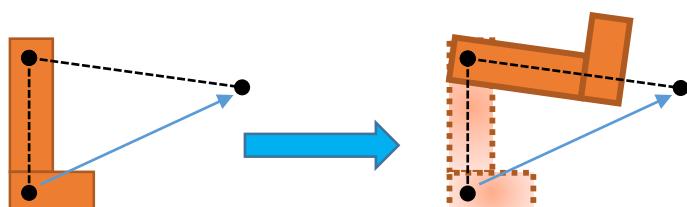
原理は 2D の CCD-IK と同じでいいです。面倒(というかトラブルメーカー)なのは回転の「軸」を決定することです。

で、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。

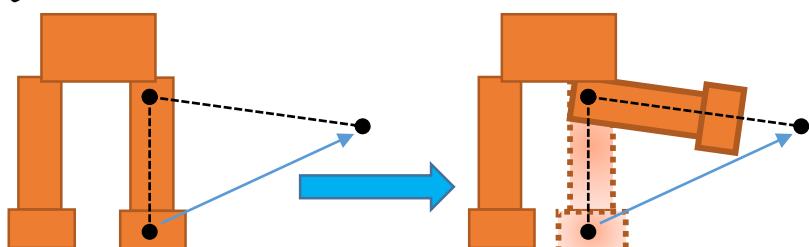
では、何ベクトルと何ベクトルで外積をとるのか?を考えましょう。

どの軸を中心に回転すべきなのかについて考えてみれば分かるのではないか…

例えば真横から足の IK を見た場合



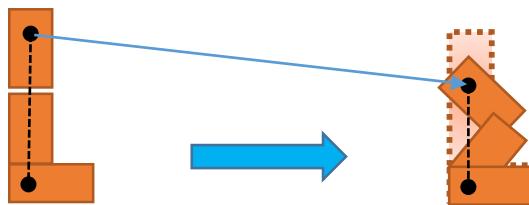
図のように足 IK を前にキック的に出そうとすると、当然軸は X 方向といふか横方向の軸になります。



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ふかと方向になりますね？

あ、まだ正解じゃないですよ？でも、ちょっと自分で考えて、あーそーなんだなーって思ってください。

ひとまずこの考えをもとに実装してみてください。足を上げるところくらいまで。
これには穴があります。たとえば「しゃがみ」を考えてください



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、それ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ね？

何故かって？

ターゲットへの「角度」が変わってないからだよ!! そうですね？

ベクトルの方向が同じままなので、これでは外積が出せません。意味は分かりますかね？

分からぬ人のためにちょっとだけ解説すると

ベクトルA(x_a, y_a)とベクトルB(x_b, y_b)とします。簡単のため二次元でやりますね？

この外積は

$$A \times B = x_a y_b - x_b y_a$$

ですね？ここでベクトルAとベクトルBが同じ方向を向いていなければ

$$B = kA = (kx_a, ky_a)$$

と書けますね？BはAのk倍のことです。

さて、その前提で外積を出してみましょう。どうなりますか？

$$A \times B = x_a y_b - x_b y_a = x_a k y_a - k x_a y_a = k x_a y_a - k x_a y_a = 0$$

という感じでゼロになります。外積は同じ方向を向くとゼロになります。

これは三次元でも同様で

$$A \times B = (y_a k z_a - z_a k y_a, z_a k x_a - x_a k z_a, x_a k y_a - k x_a y_a) = (0, 0, 0)$$

つまり、同じ方向を向いたベクトルは外積をとっても意味がないわけです。

じゃあ、しゃがみ時はどうしてるのかな？と考えよう。

実は3年前のテキストでは、分からぬあまりにこう考えました。

「さて、PMDには「ひざ」のとる角度が特殊だといろんな資料に書いてあったので、信じることにしましょう。

ということでIK名が「右ひざ」もしくは「左ひざ」ならばX軸方向にしか稼働しないらしいので、強制的に軸をX軸にしちゃいましょう。

簡単です。

```
if(対象IKが"右ひざ"もしくは"左ひざ"){
    軸.x=-1;
    軸.y=0;
    軸.z=0;
}

それ以外の場合は外積で軸を作りましょう。』と言ってました。参照ソースコードをお見せしますが、
///CCD_IKを行い、結果を返す。
///@param ikmap IKデータから取ってきたIKボーン名とIKリストデータのマップ情報
///@param boneInfo ボーン名によっては特殊な動きをする必要があるため「名前」取得のためボーン情報への参照を受け取る
///@param ikname 対象IKボーン名
///@param location 動かしたい先の座標
///@note 重要なのは…というが必要なのはそれぞれのボーンをどれだけ回転するかの情報だけ
///だが、CCDであるため途中経過の座標も重要なので、関数内部にテンポラリとして保持しておく
///関数を抜けたら用済みである
void CCD_IK(std::map<std::string,IKList>& ikmap,std::vector<BoneInfo>& boneInfo,const char*
ikname,XMFLOAT3& location){
    int ikIdx=ikmap(ikname).boneIdx;//IKボーンマトリクス番号を取得
    std::vector<unsigned short>& itn=ikmap(ikname).boneIndices;//IKに対応するボーンのインデックス
    std::vector<unsigned short>::iterator it=itn.begin();

    std::vector<XMFLOAT3> tmpLocation(itn.size());//IKから支点までのボーン座標(IKは含まない)
    XMFLOAT3 ikpos=_boneoffsets(ikIdx);//IKの座標
    XMFLOAT3 targetpos=ikpos+location;//IKの座標(目的地やから変わらへん)

    int nodecount=itn.size();//チェーンノード数

    for(int i=0;i<nodecount;++i){
        tmpLocation(i)=_boneoffsets(itn(i));
    }
    for(int c=0;c<40;++c){
        for(int i=0;i<nodecount;++i){
            if(ikpos==targetpos){
                break;
            }
        }
    }
}
```

```

        }

XMFLOAT3 originVec=ikpos-tmpLocation(i); //もとの先っちょIKとさかのぼりノードでベクトル作成

XMFLOAT3 transVec=targetpos-tmpLocation(i); //目標地点とさかのぼりノードでベクトルを作成

//ベクトル長が小さすぎる場合は処理を打ち切る

if(abs(Length(transVec))<0.0001 || abs(Length(originVec))<0.0001){

    return;
}

//軸作成

XMFLOAT3 norm=Normalize(originVec)^Normalize(transVec); //ちなみに^は外積として扱っている

const char* name=boneInfo(iIn(i)).boneName;

if(strcmp(boneInfo(iIn(i)).boneName,"右ひざ")==0 ||

strcmp(boneInfo(iIn(i)).boneName,"左ひざ")==0){

    norm.x=-1;//norm;

    norm.y=0;

    norm.z=0;

} else{

    if(Length(norm)==0){

        return;
    }
}

//角度計算

XMVECTOR rot=XMVector3AngleBetweenNormals(XMLoadFloat3(&Normalize(originVec)),XMLoadFloat3(&Normalize(transVec)));

rot*=0.5;

//角度が小さすぎる場合は処理を打ち切る

if(abs(rot.m128_f32(0))==0.000f){

    return;
}

float strict=(2.0f/(float)nodecount)*(float)(i+1);

if(rot.m128_f32(0) > strict){

    rot.m128_f32(0)=strict;
}

```

```

XMVECTOR q=XMQuaternionRotationAxis(XMLoadFloat3(&norm),rot.m128_f32(0));

//ボーンの変換行列を計算
XMFLOAT3 offset=_boneoffsets(itn(i));
XMMATRIX RotAt=XMMatrixTranslation(-offset.x,-offset.y,-offset.z)*
XMMatrixRotationQuaternion(q)*
XMMatrixTranslation(offset.x,offset.y,offset.z);
_boneMatrixes(itn(i))=_boneMatrixes(itn(i))*RotAt;

offset=tmpLocation(i);

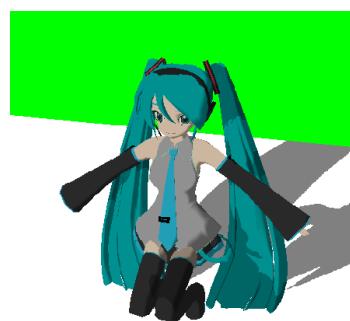
//理論上の変換行列を計算
RotAt=XMMatrixTranslation(-offset.x,-offset.y,-offset.z)*
XMMatrixRotationQuaternion(q)*
XMMatrixTranslation(offset.x,offset.y,offset.z);

//理論値を更新
ikpos*=RotAt;
for(int j=i-1;j>0;--j){
    tmpLocation(j)*=RotAt;
}
}

}

と。

```



当時はこれで上の画像のように、きちんと動いたものだから「これが正解じゃ!!」と思って当時の学生にドヤ顔で教えていたんですが、よくよく考えると、足を開いた状態でしゃがむと、中心軸もX軸に平行じゃないわけだから、おかしなことにならんしかな?

ちなみに当時のコードでやってみたらこうなりました。



まともそうに見えますが、やっぱり膝が不自然なんですよね。というか、特定の角度の時にやっぱりおかしなことになります。一瞬なのでスクショ取れてませんけど。

3年前より僕は知識と技術は上がっているはず…ならば、何かあの頃に思いつかなかつたことも思いつけるはず…そうか!!軸のことばかり考えるからこうなるんだ!!全く違うことを考えなければならない! そう思うのだ!!!

三年前の俺になくて、今の俺にあるもの…。

そう…それは、

LookAt 行列

LookAt 行列というのは、特定のベクトルを特定の方向に向かせるための行列です。ちなみに「LookAt 行列」って名前は僕が勝手に名前付けてるだけなので、インターネットで探しても出てきません。

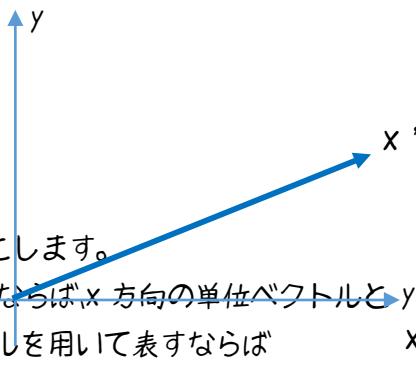
いや、出てきた。gluLookAt ですね。これは OpenGL の関数で、でもこれはカメラに使うやつなので、今回の用途には合いませんし、XMMatrix 系にも LookAt は存在しません。

僕が LookAt, LookAt 言ってるのは、Unity の Transform の関数に LookAt ってあるからです。そいつは、オブジェクトの \vec{z} 方向を任意のベクトル方向に向かわせるって関数です。

この関数はたぶん今回の IK だけでなくいろんな所で使うことになると思います。例えば敵がこちらを発見して顔をこちらに向けるとか、そういう事をするために必要なので、作っておきましょう。

LookAt 行列の作成

今回も最初は 2D で考えてみよう。結果的には回転を表すが回転角は分からず、ベクトルによる方向指示のみとする。で、向かせたいベクトルを \vec{x}' とする。



この x' は正規化されているとします。
で、角度を用いずに x' を表すならば、 x 方向の単位ベクトルと y 方向の単位ベクトル…つまりどちらも長さが1のベクトルを用いて表すならば

$$x' = m\vec{x}_i + n\vec{y}_i$$

と表せます。

m, n は任意の実数で、 x_i, y_i はそれぞれ x 方向、 y 方向の単位ベクトル(長さ1)です。だから別に x_i, y_i ってのはいらないっちゃいらないけどね。意味的にそう書いてるだけです。なんでわざわざこれを書いているのかというと、「ベクトル」ということにしていれば、目的地をベクトルの足し算で表すことができますね?
通常であれば座標 (a, b) というのは

また、 x' は正規化済みですから。

$$\sqrt{m^2 + n^2} = 1$$

とします。

ここまでよろしいでしょうか?

で、やりたいことは元の X ベクトルに行列 M をかけて X' になる。そういう行列を考えたいとします。

元の X は $(1, 0)$ なので

$$(1, 0) \times \begin{pmatrix} a & b \\ c & d \end{pmatrix} = (m, n)$$

こうしたいわけ。そうなると当然

$$(1, 0) \times \begin{pmatrix} m & n \\ ? & ? \end{pmatrix} = (m, n)$$

となるだろう。ただしこのままでは

$$(0, 1) \times \begin{pmatrix} m & n \\ ? & ? \end{pmatrix} = (?, ?)$$

となってしまい、 X 軸方向のことしか考えてない。 Y' は X' から 90° ずれてる部分なので本当は

$$(0, 1) \times \begin{pmatrix} m & n \\ c & d \end{pmatrix} = (-n, m)$$

となってほしい。

そうなると

$$(0,1) \times \begin{pmatrix} m & n \\ -n & m \end{pmatrix} = (-n, m)$$

となる。これが新しいY軸のベクトルだ。

どこかで見た形だなーって思った人は感がレル。コブラのマシンはサイコガンのあれだ。回転行列と同じような形になっている。さらに

$$\sqrt{m^2 + n^2} = 1$$

であるから、ますますもって m, n は \sin, \cos と対応しているのである。

回転行列に対応しているということは、特定の点を回転させるためには、それぞれの新しい軸との内積になっていることがわかるだろうか？

つまり、意味合いからすると、それぞれの軸に対する内積が、それぞれの軸の成分だったわけで、それを 3D に拡張すると、どういう話になるのでしょうか？

結局は 3D の時もそれぞれの軸に対する内積をとり、それを全部合成したものが新しい座標となります。それができるような行列を作ればいい。

まとめいとう。

- X 軸との内積が X 座標
- Y 軸との内積が Y 座標
- Z 軸との内積が Z 座標

これは分かると思いますが、これがなんかしら変換された後の XYZ に対しても同様である。つまり、特定の方向を向かせたければ、それぞれの X, Y, Z に当たるベクトルをそれぞれ計算し、それに対して「内積」をとってしまえばいいわけです。

じゃあ「どこか向く」と言った場合、3 軸必要かつていうと、そうじゃなくて、カメラ行列の時もそうでしたが、二つあれば外積で直交ベクトル出せるので十分です。

つまり

向かせたい方向ベクトル: Z 軸

うーん。ここでアッパーべクトルとか言って安易に(0,1,0)ってやっても良いんですが、センターから末端 IK が「真下」を向いてると結構大変なことになります。

くっそー!! 結局ここで軸を決めないといけないのか…。

うーん。重力方向は避けたいのでX軸を決めよう。右ベクトルとして(1,0,0)を使おう。

ここで二次元の時を思い出してほしいんですが

$$\begin{pmatrix} m & n \\ -n & m \end{pmatrix} = \begin{pmatrix} \text{回転後のX軸} \\ \text{回転後のY軸} \end{pmatrix}$$

になっていますから、これを3Dに拡張すると

$$\begin{pmatrix} \text{回転後のX軸} \\ \text{回転後のY軸} \\ \text{回転後のZ軸} \end{pmatrix}$$

になります。これは言い換えると

$$\begin{pmatrix} x'_x & x'_y & x'_z \\ y'_x & y'_y & y'_z \\ z'_x & z'_y & z'_z \end{pmatrix}$$

となります。

です。この確認は(0,0,1)が(z'_x, z'_y, z'_z)となればオッケーなんですが、計算してみてください。なりますよね？

そうすると

$$\begin{aligned} (1,0,0) &\rightarrow (x'_x, x'_y, x'_z) \\ (0,1,0) &\rightarrow (y'_x, y'_y, y'_z) \\ (0,0,1) &\rightarrow (z'_x, z'_y, z'_z) \end{aligned}$$

という変換になるのが分かると思います。こういう行列を返す関数を作ればいいです。

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。ここまで大丈夫でしょうか？

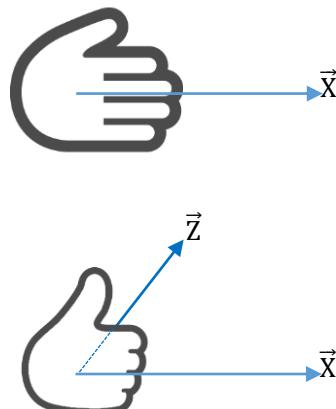
ベクトルの向きをあまり考えなしに、適当な順序で外積とっちゃうと大変なことになります。

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「右ねじの法則」によって決まっています。高校の電磁気学で出てきたあれですね。



忘れたとか知らないって人までサポートしてたら、ページ数が膨大になるので、詳しくは
<https://ja.wikipedia.org/wiki/%E5%8F%B3%E6%89%8B%E3%81%AE%E6%B3%95%E5%89%87>
 を見てくれ。

具体的には、右手系の場合、二つのベクトルから直行するベクトルを出すときには、まず手のひらをシュッとさせて外積の左辺値の向きに右手の指先が向くようにする(図は $\vec{x} \times \vec{z}$ のとき)



次に右辺値ベクトル(\vec{z})の向きに指先だけカクっと向ける。その状態でグッジョブする。
 この時の親指の方向が \vec{y} になるというわけだ。
 これが高校の理科の時間に習う「右ねじの法則」である。であるが、これは右手座標系の話である。確かに言われてみればそうなのだ。
 つまり系が変われば、それに合わせるには外積の順序も変えなければならない…これは恥ましいことだが、最初に系を確定させて、それに全てを合わせるということが…わかるだろう?つまり DirectX の座標系を扱うときは「左ねじの法則」だ。ややこしいね。
 ということを考えて書いてみた関数が以下の通りだ。

```
// 特定の方向を向かす行列を返す関数
XMMATRIX LookAtMatrix(XMFLOAT3& lookat, XMFLOAT3& right){
    XMVECTOR vz = XMVector3Normalize(XMLoadFloat3(&lookat));
    XMVECTOR vx = XMVector3Normalize(XMLoadFloat3(&right));
    XMVECTOR vy = XMVector3Normalize(XMVector3Cross(vx, vz));
    vx = XMVector3Normalize(XMVector3Cross(vz, vy));
    vy = XMVector3Normalize(XMVector3Cross(vz, vx));
    vx = XMVector3Normalize(XMVector3Cross(vy, vz));
```

```

XMMATRIX ret = XMMatrixIdentity();

XMFLOAT3 fvx, fvy, fvz;
XMStoreFloat3(&fvx, vx);
XMStoreFloat3(&fvy, vy);
XMStoreFloat3(&fvz, vz);

ret._11 = fvx.x; ret._12 = fvx.y; ret._13 = fvx.z;
ret._21 = fvy.x; ret._22 = fvy.y; ret._23 = fvy.z;
ret._31 = fvz.x; ret._32 = fvz.y; ret._33 = fvz.z;

return ret;
}

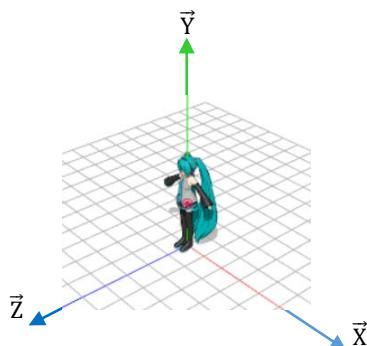
```

確認のために、キャラクターの Z 軸を特定の方向に向けるのをやってみてください。
と書いたところで、いろいろと問題が発生した。

僕のプログラムで起きた既知(キチ)の問題は以下の通りだ

- 真正面のはずなのにお尻を向ける
- モデルが縮む
- モデルが消える

お尻を向ける…というか Z 軸が反転してしまう原因是、座標系にある。MMD を起動し、エディタを見てもらえばわかるが、



右手系だよこれ!!どうして DirectX なのに左手系にしなかった!!またぶん他の CG ソフトに合わせたんでしょう。

ともかく、つまるところ Z 方向が最初からお尻向けてるんですね。これは仕方ない。

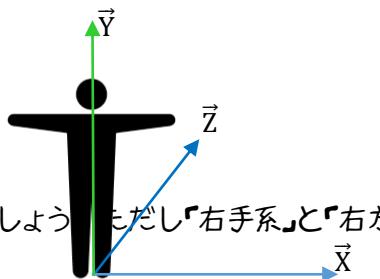
対応策がいくつか考えられますが、

- 表示の際に Y 軸 180°回転させとくのと LookAt はお尻向きで設定する
- 読み込みの際に左手系にしてしまう(Z 値反転させてインデックス逆にするかカーリング逆にする)

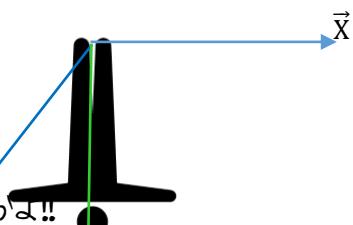
こんな感じでしょう。どちらにするのかは皆さんにお任せします。正解はないんです。

次に「モデルが縮む」ですが、これは単純に出来上がったそれぞれのベクトルをノーマライズし忘れていただけでした。

最後の「モデルが消える」ですが、これは \vec{z} を反対に向けた時に地面の反対側に行ってました。どういう事がというと、



この \vec{z} をこっちに向けてみましょう。ただし「右手系」と「右方向」は変えずに…。
そうすると、



こうなるわけです。「吊られた男」かよ!!
さて、ここにきて思い当たるのが「アップ(+)Y)ベクトル」です。カメラやモデルにおいて向きを決めるときには、たいていの場合はアップ(上方向)ベクトルです。つまりここを固定しておかないとい、逆さになるから、right でも left でもなく上ベクトルを基準にしたのですね。
つまり、X 軸固定にしたときは、 \vec{z} 反転の際に X 軸周りに 180° 回転したのに対して、アップベクトルを使用する場合は \vec{z} 反転の時に Y 軸周りに 180° 回転しているというわけです。

通常、人間が後ろを向くときは Y 軸中心に回転するからで、Y 軸を基準にするのが通常は正しいというか、都合がいいことになります。

ということで、先ほどの関数を改変するならば

- 引数には Upper と Right の両方のベクトルを渡しておく
- 通常は Upper を基準として LookAt 行列を作成
- 真上を向くときなど Upper が向かせたい向きと同じになってしまっていれば、その時は Right のほうを使用する

といった感じだろうか。

まだハグ混入しそうだが、今のところつじつま合わせるためににはこれが簡単だろう。

さて、ここでまた CCD-IK の話に戻る。

納得いかないんですが…

まあCCD-IKの正式な記事はComputerGraphicsGemsJP2012の記事の作者である

<http://mukai-lab.org/library/ik-legacy.html>

氏のやり方を参考にさせてもらったのですが…このやり方でうまくいくんです。うまくいくんですけど…今冷静に考えると納得行かないんですね。

うん、まあもう時間ないんで『論文に書いてるんだからその通りで正しいんだ』でもいいんですけど、そんなもんエンジニアの姿でも教育者の姿でもないんですね。



夢はひろがりんぐ

んで、そこまでしてなんでIKをやりたいのかというと、FKでもエエんちゃうのか?というと

<https://ja.osdn.net/projects/mmdmotion-java/wiki/MMDIKSolver>

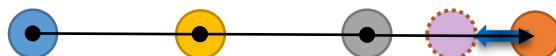


こんな風に特定の点を追いかけて体全体を動かすとか、バランスをとるとかやりたいわけです。これができれば階段とかで



手付けアニメーションだと階段の高さによってアニメーションを変えなければならないのですが、足の位置との当たり判定で全身のポーズを決めることが可能になるわけです。

何で納得いってないのか&実験



例えば図のように平行に視点に近づけるように端点を動かそうとするとします。当然ながら『長さを変えてはいけない原則』があるため、関節を曲げなければいけません。

ところで CCD-IK ってどういう法則で曲げていきましたつけ? そう、ボーンを現在のエフェクタ

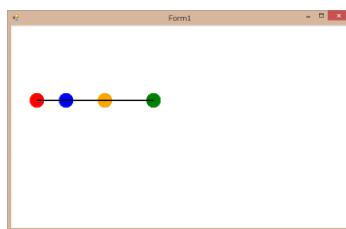
(コントロールポイント…つまるところ動かしたい先の座標)に向くように(最も近づくように)曲げてあげるんですが…上の図を良く見てください。

エフェクタの位置は何処にありますか?

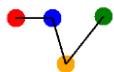
そう。最初に配置されていたボーン直線の真上です。そしてそこから全てのボーンが一直線上に並んでいます。

既にボーンの向きはベクトル的には目的地を向いています。これでは曲げようがないまんね?

試しにこういうアプリケーションを作って実験してみました。



で、端点を左に押し込むと…



あれ? 曲がってしまいましたね? 予想外です…が、あることに気が付きました…人間だもの。押し込む際にまっすぐやっているつもりがちょっと上下に行くわけよね。

そう考えて、Shift キーを押しながら移動させると Y 座標が変わらないようにします。そうすると…



詰まりました。なるほどそういう単純なことだったのか…TL。

ちなみにほんのちょっとでも曲げた状態で Shift 移動させると、正しく IK を行います。つまりピーンとまっすぐにしている時は CCD-IK であったとしても詰まっちゃうってことです。良かった。安心した。まあ、ニンゲンでも関節伸ばした状態で関節方向に骨を押し込むと関節外したり関節破壊したりできますもんね…。

まっすぐに対する解決案

というわけでまずは今回の実験結果を大前提に考えましょう。そう考えると膝を曲げる動きの場合には2つの可能性が考えられます。

- 予め曲げられている
- まっすぐ押し込むとされた場合には適当な方向にちょっとだけ曲げてやる

3年前のスクワットがうまく行っているのは前者の状況だったからだと推測できます。後者の対

処をしてるならプログラムに何らかの痕跡があるはずですが、ありませんでしたし…。

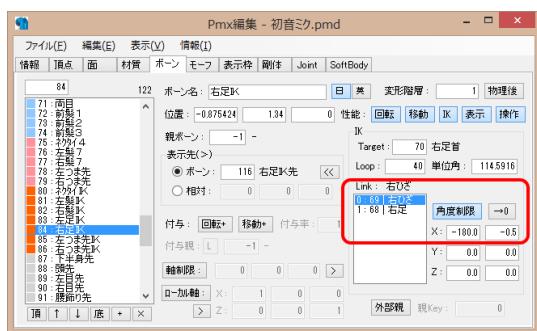
では、何はともあれ、IK を実装していきたいところですが、まだ解決というか決めていかなければならぬことがあります。曲げる軸について考える

二次元ならば軸は完全に「 Z 軸」なので良いんですが、3D の場合、軸をどう取るのかっていうのがかなり悩ましいという話は前にしました。それを解消するために LookAt を作ったはずなんですが、結局「制限角度」を正しく設定するためには必要なんだよね。

なんとかというと、軸ベクトルが反対側向いてると角度の符号が変わっちゃうんだよね…。

角度制限がないなら基本的には LookAt のみでやっちゃってオッケー（結局固定すべき軸は決めておく必要はあるんだけど）。そうではあるんだけど人間の骨は大抵制限がついてるからね。仕方ないね。

さて、では手始めに「ひざ」について考えよう。一応 PMXEditor でミクさんを開いてみると



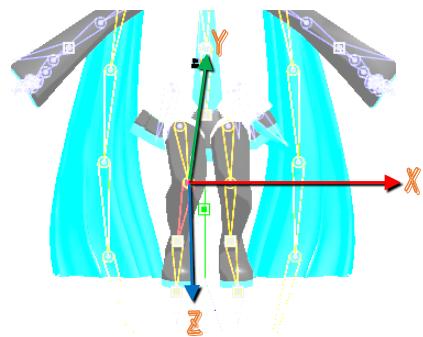
こんな感じです。こんな感じ言われてもよーわからへんといった具合なんでしょうね。右側のパラメータを見てもらうと分かるように、角度制限がついてますが $-0.5^\circ \sim -180.0^\circ$ ということで、予めちょっとだけ曲げられているのが分かると思います。

まあ、モデル自体はそういう風に作られていることもあり、3年前はうまく行ったんでしょう。ただ、前にもお話ししたように「ひざ」だからといって、回転軸を強制的に X 軸にするようでは



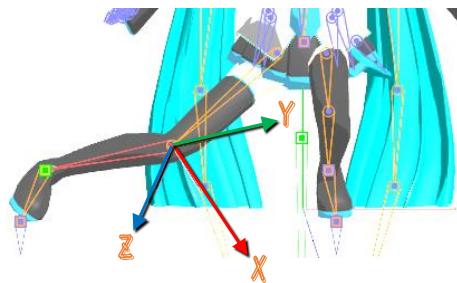
こんなふうになった時にうまく行きません。

原因を言うと、普通にしゃがんだ時にはこうなんです。



Xが右方向を向いています(画像は右手系なので注意してね)。この時はワールドX軸中心に回しても問題ないんですが

まあ、当たり前の話なんんですけど、↑の図の座標系のX軸中心って意味なのよね。



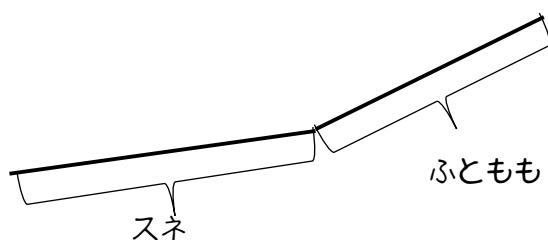
ということは、何かと何かの外積をとって、このX軸を計算しなければならないわけだ。



この図を見ると、右足IKは右ひざと右足に対してリンクを持っているのがわかりますね？2つのリンクを持っているわけです。この数は可変ではあるんですが、大抵の場合は1~2です。

で、何度か試してみてわかるのは、太ももとヒザ(スネ)の座標系(軸)は一致させておかないと、おかしなことになるということです。

例えばこの「位置関係」が正しかったとしても、それぞれの座標系「軸」がズれていれば膝関節でねじれが発生するわけです。



経験上わかると思いますが「ありえない」ですよね？ヒザとスネの向きが違うということは、それはもう関節外れてますから。

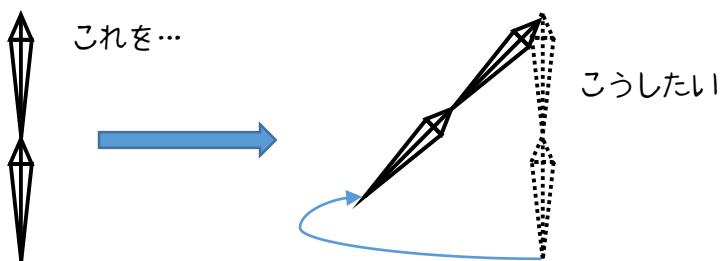
ただ、別の生き物や機械とかの場合はその限りじゃないです。ですが、多分、向きは同じにしておかないと IK を使う時には「解」がたくさん出てきてやりづらいと思いますので、統一しておいたほうが良いでしょう。

で、膝を使用する場合、X 軸を中心に曲げるという事だが、これは股関節を開いているときに、不具合を起こす。

そうだよね。さっきも書いたけど、股関節の開きに合わせて膝の X 軸方向を回転させなきゃならない。

どれくらい回転させたらいいんだろう。で、やっぱりここで「度」とか使いたくない。ベクトルだけでなんとかならないものだろうか。

ということで考えました。



2D なら簡単に求められるところだが、3D の場合だと、内積出して角度計算して外積で軸出して、その軸周りの回転行列を求める…となる。

もう少し簡単にいいかないものが？

「いや、特定のベクトル A を特定のベクトル B に向かせたいんでしょ？じゃあ LookAt で作った行列でいいんじゃね？」

そうだわな。ただ、あれは 2 ベクトルが特定のベクトルに向くように作っているので、

$$R_{ZB} \cdot \vec{Z} \Rightarrow \vec{B}$$

であって、

$$R_{AB} \cdot \vec{A} \Rightarrow \vec{B}$$

ではない。

そういうことだ。俺は $A \Rightarrow B$ が欲しいんだよ!!!さて、行列のことを考えまくって、なんとか R_{AB} を出せないか考えた。

みんなもしばらくは自分で考えてみてくれ。

うーん、武器が足りない!。ほかに LookAt で分かるものは…ああ!! そうだ!!

$$R_{ZA} : \vec{Z} \Rightarrow \vec{A}$$

というわけだ。さらにこれを逆に考えると

$$R_{ZA}^{-1}$$

つまり、 $\text{LookAt}(\vec{Z} \Rightarrow \vec{A})$

ちなみに今回のベクトルは、長さを変えない、移動しないベクトル → 回転のみということだ。

$$\vec{A} \Rightarrow \vec{B}$$

はちょっと変形すると

$$\vec{A} \Rightarrow \vec{Z} \Rightarrow \vec{B}$$

となることは、わかるよね? A から B まで回転するには、 $Z \Rightarrow B$ と $A \Rightarrow Z$ を組み合わせれば

$$A \Rightarrow B = A \Rightarrow Z \times Z \Rightarrow B$$

だから

$$A \Rightarrow B = (Z \Rightarrow A)^{-1} \times Z \Rightarrow B$$

というわけ。つまり

XMMATRIXTranspose(LookAt(A, up, right)) * LookAt(B, up, right);

とでもしてやればいい。

LookAt 行列関数の改造

これをを利用して、任意ベクトル A を任意ベクトル B へ向かわせる LookAt 関数のオーバーロード関数を作成しよう。元の LookAt は 27.4.1 を見返してくれ。

大雑把に描くとこんな感じ。

```
XMMATRIX LookAt(XMFLOAT3& origin, XMFLOAT3& lookat, XMFLOAT3& up, XMFLOAT3& right){  
    XMMATRIX tmp=LookAt(origin,up,right);  
    tmp=XMMATRIXTranspose(tmp)*LookAt(lookat,up,right);  
    return tmp;  
}
```

これを使用すれば、元の IK ベクトルを IK ターゲット 移動後 IK ベクトルにする行列ができるります。

いよいよ実装である

長かった…ホンマに長かった。そして苦しめられた…。某クラスのインフルエンザアウトブレイクによりカミサマから猶予期間を与えてもらったにも関わらず、最終的にはまだ不具合が

残っている状況にある…申し訳ない。CCD-IK の論文を理解して 2D の IK を実装するのと、MMD の IK を実装することの間にはまだまだ深い深い谷があったのだ。



ただただ申し訳ないが、それを言っても前に進まないので、実装しよう。最初に言っておくけど、まだ不具合が残っていることは認識しておいてほしい。やっぱりヒザ周りが難しいのだ。

もし、完璧にしたい人がいるなら、オープンソースの MMD エディタや MMD 再生機や Java や Javascript でやっている強者もいるので、そちらを見ながら研究してほしい。

http://dxlib.0.007.jp/DxLib/DxLibMake3_17a.zip

<https://ja.osdn.net/projects/mmdmotion-java/wiki/MMDIKSolver>

<http://www.nicovideo.jp/mylist/49125767>

<https://jthird.net/>

<https://github.com/edvakf/MMD.js>

<https://code.google.com/archive/p/nymmd/>

※ちなみに現在 jThreee は開発停止 & サイト閉鎖されています。

投げっぱなしに思われても仕方ないのだが、うーん。まあキチンとやろうとすると、集中した潤沢な時間が必要だってことです。

それはともかく、実装してみましょうってのがこの章の趣旨である。

だが、ここに入る前にも最初に言ったように「IK は十分条件であって必要条件ではない」のでそこまでしなくてもいい。余裕があったらでいい。特に次年度就職年次はそんな時間があるなら就活用の作品を作つたほうがいい。

さて、言い訳は済ませたので実装しよう。

PMD から IK 情報を取得する

久々に PMD のデータを取得するぞ。IK の情報だ。

http://blog.goo.ne.jp/torisu_tetosuki/e/445cbbbe75c4b2b22c22b473a27aaaae9

さて、見てもらえばわかるが、相当アレだ。データの場所はボーン情報を読み切った後のところなので、ボーンを全読み込みして今のみんなは素直に読み込んでもらわればいい。

まず、最初の2バイトが IK データの数だ。2バイト読み込んで数を確認しよう。デフォルトの初音ミクなら、7個になっているはずだ。

| | | |
|-------|------------------------------|-------------------------|
| 6B51F | bone[121].bone_head_pos[0] | 3FA499A6 418A43E9 BFA54 |
| 6B52B | ik_data_count | 0007 |
| 6B52D | ik_data.ik_bone_index | 0050 |
| 6B52F | ik_data.ik_target_bone_index | 004B |
| 6B531 | ik_data.ik_chain_length | 03 |
| 6B532 | ik_data.iterations | 000F |
| 6B534 | ik_data.control_weight | 3CF5C28F |
| 6B538 | ik_child_bone_index[0] | 0008 0007 0006 |
| 6B53E | ik_data.ik_bone_index | 0051 |
| 6B540 | ik_data.ik_target_bone_index | 004C |
| 6B542 | ik_data.ik_chain_length | 05 |
| 6B543 | ik_data.iterations | 0008 |

確認してくれ。

で、結構クセモノなデータである。IK データひとつあたり

2+2+1+2+4+可変長*2

という状況だ。もうアライメントを意識していないのは分かった。そこは諦めてる。

だが、最後の IK 影響ボーン番号が「可変長」だから、必要なデータを「ガ-っと」取得することは不可能なのだ。

IK データを一つ一つ丁寧に取得しなければならない。面倒だが、データの総量はそれほど多くもないはずなので、コストはそこまでかからないんじゃないかな…とは思います。ちなみにこの IK の後にスキンデータ(表情データ)が控えているわけだが、IK を実装しなくても、表情を出したい場合は IK を丁寧に読み込まなくてはスキンデータの先頭すらわからないのだ!!!

もはや文句言う気にもならない(いや、IK 再生を完成させられない僕にそんな資格など最初からないのだ)

構造体こんな感じかな

```
///@brief IKリスト
///IKボーンとターゲットボーンの二つがあるのは、
///それでベクトルを作るためあります。
struct IKList{
    unsigned short boneIdx; ///IKボーンインデックス
    unsigned short tboneIdx; ///ターゲットボーンインデックス
    unsigned char chainLen; ///さかのぼりボーン数
    unsigned short iterationNum; ///巡回回数
    float restriction; ///制限角度
    std::vector<unsigned short> boneIndices; ///さかのぼりボーン番号
};
```

で、アライメントの関係があるのでメンバを一つ一つ読み込むか、#pragma pack(1)を使ってください。

僕は最後のメンバが可変長である以上どのみちガーッととってこれないので、一つ一つ丁寧に読み込みします。読み込んでください。

そんなに数はないので、おかしなデータになってないかどうかご確認ください。

もちろんこの IKList は配列状態になっていますのでいつも通り

```
std::vector<IKList> _iklist;  
として読み込んでいきます。
```

この時にちょっとだけ注意点ですが、VMD データとしては IK であるとかの情報があるのではなく、ボーンの状態だけが格納されています。

このため、後から IK と辻褄を合わせるように、IKList はベクタではなく、マップで持ったほうが良いかもしれません。やりようは色々あるので、各自ご研究あらんことを。

CCD_IK 関数について

作りましょう。そうしましょう。ややこしいんですけどね。ちなみに通常の回転と IK とどちらを優先すべきかというと、IK の方です。

MMD のエディタ見れば分かりますが、IK 動かさない限り IK 回りの回転は反映されません。IK を切ると反映されますがね？

まあ…ぶっちゃけた話をすると…股関節→膝→足首であれば、三点なので、この場合に関しては CCD-IK を使わざとも軸さえ決まれば余弦定理で行けるとは思うんですけどね…うーん制御点が 3 点以下とそれ以上で場合分けしてもいいかなー。

2つのベクトル間の角度を知るための関数が用意されています。

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweennormals\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweennormals(v=vs.85).aspx)

XmVector3AngleBetweenNormals はベクトル間の角度を返すものだ。

ちょっと疑問なのが「ベクトルを返します。各要素に、N1 と N2 の間のラジアン角度が複製されます。」

の下線の部分…。確かにラジアン値がすべての要素に同じ値が入っている…これ、意味あんのか？float 一個返したほうが速くね？

って思うんだけど、定義まで見ると、どうも SIMD 拡張命令を使用しているらしい。申し訳ないがあまり検証する気にはなれない。ともかく 0 番要素を取ってくればいいだろう。

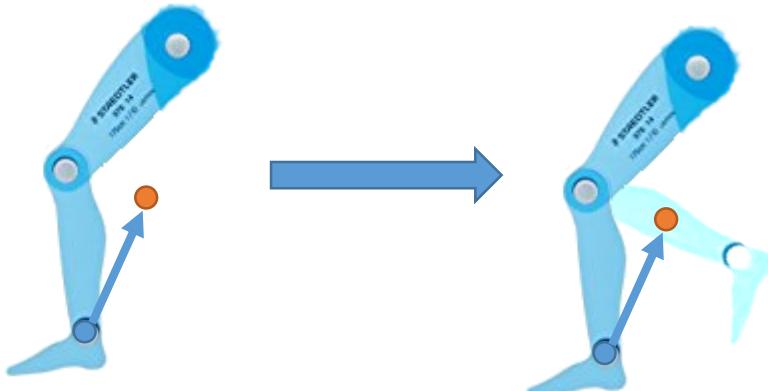
ベクトル値を受け取るが、ベクトル値ではないことに注意してくれ。

んで、最初は 2D のつもりで作っていきましょう。今は膝のことだけ考えて



こんなイメージで作っていきます。つまり回転は全て X 軸を中心とします。2D の解説は既にやっていますので、それを元に解説していきます。

まず、膝を曲げることを想定します。



例えば足首の IK を図のように移動した場合、膝を中心にしてスネを回すわけですが、間の角度を計算して、その差分だけ回転させてあげれば良いわけです。

まずそこを考えましょう。先程の `XMFLOAT3AngleBetweenNormals` には 2 つのベクトルがあればいいので、

図の2つのベクトルがわかれればいいですね？



そういうことで

- ひざから現在のIK座標ベクトル
- ひざから目的のIK座標ベクトル

を作ります。

CCD-IK の実装

```
XMFLOAT3 originVec = ikOriginPos - bone.headpos; // IKとさかのぼりノードでベクトル作成  
XMFLOAT3 targetVec = ikTargetPos - bone.headpos; // 目標とさかのぼりノードでベクトル作成
```

こんな感じで。

ちなみに PMD の IKList が持っている ik インデックスは末端から根っこに向かってインデックスが配置されてるので、別に逆からとかにしなくていい。

つまり

```
for (int i = 0; i < iklist.ikchainLen; ++i)
```

でいいわけです。さらに前述の2つのベクトルから軸をつくりましょう。外積を取るわけですが、事前に正規化しておきましょう(正規化する必要はないですが、一応)

// 正規化します

```
originVec = Normalize(originVec);  
targetVec = Normalize(targetVec);
```

// 外積から軸を作成します

```
XMFLOAT3 axis = Normalize(Cross(originVec, targetVec));
```

はい、これで軸が出来上がるわけですが、

// もしひざ系なら、X軸を回転軸とする

```
if (bone.name.find("ひざ") != std::string::npos){  
    axis.x = -1;  
    axis.y = 0;  
    axis.z = 0;
```

```

}else{
    if (Length(axis) == 0.0f){
        return;//外積結果が0になってるなら使えません
    }
}

```

とりあえず「ひざ」だけは強制的に X 軸なので、上のコードのようにボーン名を確認して「ひざ」という文字列が含まれていれば、強制的に X 軸の回転とします。

文字列に特定の文字列が含まれているかどうかを確認するためには C 言語では strstr という関数を使いましたが、C++ では find で検索できます。

find の戻り値は「見つかった場所」を数値で返してきます。

なお、見つからなかった場合は std::string::npos を返しますので、これ以外なら「ひざ」という文字列があったという証拠になります。

ただし上のコードでは axis.x=-1 としていますが、これは PMD が右手系で、デフォルトでお尻向けていているため、左手系に合わせるために -1 にしています。

さて、ここまでではいいでしようか？

ここまでできたら、「軸」が決定していますので、あとは間の角度です。間の角度は

XMVector3AngleBetweenNormals

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweennormals\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweennormals(v=vs.85).aspx)
か

XMVector3AngleBetweenVectors

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweenvectors\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweenvectors(v=vs.85).aspx)

もしくは自作の角度計算関数を使いましょう。

二つのベクトルの内積 = $\cos \theta$ なので、 $\arccos(\text{内積})$ で間の角度は求められます。

でも面倒なので出来合いの関数を使ったほうが楽だし、多分高速だし、今回は XMVectorBetweenNormals を使用しましょう。

以前にも書きましたが、この関数は無駄に XMVECTOR 型で返してきますので、こう書いてもいいですね。

//ふたつのベクトルの間の角度を計算(制限角度演算のため)

```
float angle = XMVector3AngleBetweenNormals(XMLoadFloat3(&originVec), XMLoadFloat3(&targetVec)).m128_f32(0);
```

で、角度が出ましたので『そのための回転行列』を作ります。



そのための回転行列を作るためには、軸と回転角度が必要ですが、それはすでに分かっていきます。これのために使用できる関数は2種類です。

XMMatrixRotationAxis

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

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

かもしくは

XMMatrixRotationQuaternion

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

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

です。クオータニオンはよくわかつてないし、ひとまずはまだわかる XMMatrixRotationAxis を使いします。

これは特定の軸中心に回転する行列を作ります。

数式にすると

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_xn_y(1-\cos\theta)-n_z\sin\theta & n_zn_x(1-\cos\theta)+n_y\sin\theta & 0 \\ n_xn_y(1-\cos\theta)+n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_yn_z(1-\cos\theta)-n_x\sin\theta & 0 \\ n_zn_x(1-\cos\theta)-n_y\sin\theta & n_yn_z(1-\cos\theta)+n_x\sin\theta & n_z^2(1-\cos\theta)+\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

こういう感じで既にわけわからないですが、ともかくこれで (n_x, n_y, n_z) ベクトル中心に回る回転行列が作れるわけです。

ついでに言うと、クオータニオンのほうは

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) & 0 \\ 2(q_1q_2 + q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) & 0 \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

こういう式です。 θ がクォータニオンなので、わけわからんないっすね!!
ともかく回転させます。

```
XMMATRIX rotMat = XMMatrixRotationAxis(XMLoadFloat3(&axis), angle);
```

で、ここで得られた `rotMat` という行列が「回転そのもの」を表していますが、これをボーン行列に乗算すればいい…そう考えていませんか？ そう考えていた時期が僕にもありました。当然のことですが、回転はそのままだと「原点中心回転」になります。というわけで例によって「回転中心を原点に移動する行列 × 原点中心回転行列 × 元の座標に戻す行列」を作る必要がありますね？ そういうことです。作ります。

どこを中心にしていいのかというと、ボーンの起点ですよね？ 今回はボーンの `headpos` がそれに当たりますから

// オフセットを考慮した行列を作る(原点に移動→回転→元の座標)

```
XMMATRIX mat = XMMatrixTranslation(-bone.headpos.x, -bone.headpos.y, -bone.headpos.z) *  
    rotMat *  
    XMMatrixTranslation(bone.headpos.x, bone.headpos.y, bone.headpos.z);
```

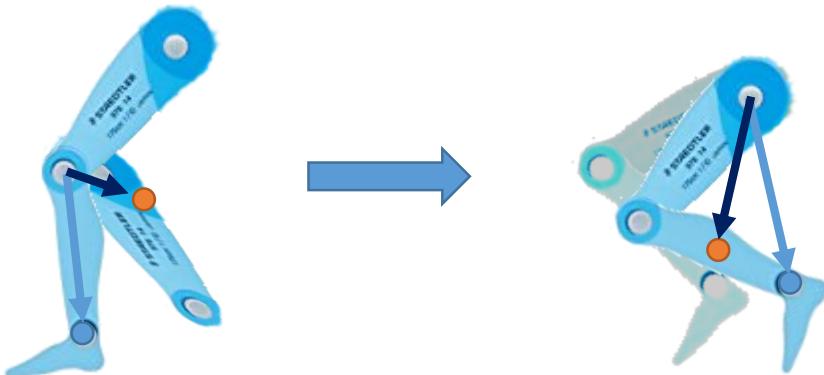
こうしたことですね。横着して `rotMat`=ってやるとあとで痛い目を見ますので、もったいなし
ようですが、新規でマトリクスを作つとしてください。
あとはそれをもとのボーン行列にかけてあげればよい

// 変換行列を計算(オフセットを考慮)

```
mesh.BoneMatrixes()(ikboneIdx) = mesh.BoneMatrixes()(ikboneIdx) * mat;
```

まあ、サイクリック(繰り返し)をしないなら、ここまで話で終わりなのですが、CCD-IK なので、
当然繰り返しをします。

つまりサイクリック中のコントロールポイント(ボーン起点)を CCD-IK の間保持しておく必要



があります。

だったら `bone.headpos` を動かせばいいと思うかもしれません、最終的には IK→FK にして
ポーズを決めるため、`bone` の場所は動かしたくないです。

なので一時変数を使用します。

```
//まずはIKの間にあるボーンの座標の一時変数配列を作つて、値をコピーする
//理由はIK再帰する毎にボーン座標が変更されるからです(元の座標は必要なので一次変数に格納)
std::vector<XMFLOAT3> tmpBonePositions(iklist.ikchainLen);
for (int i = 0; i < iklist.ikchainLen; ++i){
    tmpBonePositions[i]=mesh.Bones()( iklist.ikboneIndices(i)).headpos;
}
```

さて、そうなるといちいちベクトルも更新しなければならないため

```
XMFLOAT3 originVec = ikOriginPos - bone.headpos;//IKとさかのぼりノードでベクトル作成
```

```
XMFLOAT3 targetVec = ikTargetPos - bone.headpos;//目標とさかのぼりノードでベクトル作成
```

の箇所は

```
XMFLOAT3 originVec = ikOriginPos - tmpBonePositions(i);//もとの先っちょIKとさかのぼりノードでベクトル作成
```

```
XMFLOAT3 targetVec = ikTargetPos - tmpBonePositions(i);//目標地点とさかのぼりノードでベクトルを作成
```

となるでしょう。

あとはこれを繰り返せば良いわけです。

呼び出し側

また、忘れてならないのが呼び出し側ですが、

```
std::string name = "右足IK";
```

```
CcdIkSolve(*_mesh, name, XMFLOAT3(-0.0, _iky, 0));
```

```
RecursiveApplyBones(_mesh->BoneNodes(), 0, _mesh->Bones(), _mesh->BoneMatrices());
```

CCDIK やったあとに Recursive 処理は忘れないようにしてください。

角度制限

角度制限をつけなければ、足の関節があらぬ方向へぶつ飛びます。

というわけで、角度制限をつけたいのですが、仕様が良くわからぬので PMDEditor の readme.txt を見てみます。すると…

●[IK]

IK リスト : IK ボーンとして機能するボーンの一覧。

→

IK(数値) : 対応する IK ボーン Index

Target(数値) : IK ボーンの位置にこのボーンを一致させるように IK 処理が行われる

IK ループ回数(整数) : IK 処理での計算回数(最大 255)

単位制限角(実数) : 一回の IK 計算での制限角度(数値は rad 値の模様? | $1.0=4[\text{rad}]$ (230 度程度) $180 \text{ 度} \cdot 0.7854 = 3.141592/4$)

影響下ボーンリスト : IK の影響下にあるボーン一覧 | IK 接続先に近い方からリスト順にする必要がある

と、書かれてました。なんだこの…うーんこの…。数値が rad 値とは書いてるが、どうやらそうではないらしい。

つまり制限角度に書かれている数値を 4 倍したものがラジアン(弧度法)になるようだ…。ややこしい。

ちなみに足 IK の制限角度は 0.5 となっていたので、ラジアンは 2 くらい…だいたい 144° くらい。PMDEditor で見てもそうなのだからそうなのだろう。

そして疑問に思うのが、なぜ 4 分の 1 にまでしたんだろうってこと。別に容量的にも同じだし実際に使用する際も角度ラジアンをそのまま渡したほうがいいはずだ。
いいはずだが…何でなんですかね? まあいいや、ともかく進みましょう。

```
float strict = iklist.limitAngle * 4; // 制限角度は持ってきた角度の4倍  
// それ以上に曲げられないようにとしく  
angle = min(angle, strict);  
angle = max(angle, -strict);
```

それでもなんか荒ぶるなあ… PMDEditor の `readme.txt` をもっと読んでみる。

○deg ボタン

単位制限角を角度で入力／**有効範囲は 0-180 度程度**(あくまで解析情報からの推測値となります)

などと書いてある。

…これも適用してみたが、結局荒ぶるため、各実装系を参考にしてみた所いくつか共通点がありました。

制限角度は一つ一つの制限ではなく、合計の回転角度制限(サイクリック 1 回あたり)というこのようです。

例えば

// で? なんか `cos` から角度を出して 2 で割って…

`Rot = 0.5f * _ACOS(Cos);`

// コントロールウェイト(演算一回の制限角度)*(j+1)*2よりも Rot が大きいのなら
// その制限角度にしてしまう。

`if(Rot > IKInfo->ControlWeight * (j + 1) * 2)`

```

Rot = IKInfo->ControlWeight * (j + 1) * 2;

というコードや

maxangle = (i + 1) * ik.control_weight * 4;
theta = Math.asin(sinTheta);
if (vec3.dot(targetVec, ikboneVec) < 0) {
    theta = 3.141592653589793 - theta;
}
if (theta > maxangle) {
    theta = maxangle;
}
q = quat4.set(vec3.scale(axis, Math.sin(theta / 2) / axisLen), tmpQ);
q[3] = Math.cos(theta / 2);

というコードを見つけました。そもそも本家が実装系の説明全くしてないもんなあ…。手探りなんだよなあ。それでも共通点は
● 「2で割っていること」
● 「制限角度を出てきた角度の4倍していること」

```

となっています。

そして以下のHPの説明を読んでみる…相当難しいけど

<http://d.hatena.ne.jp/edvakf/20111102/1320268602>

「極北PさんのPMDEditorのReadme（一番下に引用した）の説明によると、指定された値×4 radianまでしか曲がれないらしい。なので、上で θ を求めたときに、この単位制限角とのmaxを取る必要がある。

もう一つのポイントは、「ひざ」という名の付くボーンはX軸方向にしか動かないというもの。（PMXだと好きなボーンに角度制限が付けられるらしいが）

これをやるにはちょっと面倒なことをしないといけない。なぜなら、先ほど求め

た \tilde{R}_0^{\prime} などは、いわば「修正『合成』回転」になるため、そこから「修正『個別』回

転 R_0^{\prime} を求め、これに角度制限をかけ、さらにそれを合成回転に直さないといけない。（回転軸制限は個別回転に適用されるもので、合成回転にではない）」

原文ママ

何言ってるか良くわからない。

でも、個別角度は求まっているから、ここに書かれているようなややこしいことはしなくていい

いんじやなしいかな。

個別が求まっているから(i+1)をかけるのはちょっと置いておいて、注目してみたのは「2で割っている」という部分。

半分にしてみよう…。

```
angle *= 0.5;
```

さて…どうなるかな。



…なん……だと？

足首が本当に IK の解の座標になっているのかを確認しましょう。普通に考えると、サイクリックしなければ本来の座標の半分くらいの位置にしか来ないはずです。

確認コード

現在のIKの座標を画面上で確認できるようにしてみましょう。



画像のようなマーカーを表示させましょう。

指定された足のIK座標と、実際の足首が乖離していないか確認しましょう。

「ビルボード」でやっても良いんですが、それだとオブジェクトに隠れてしまいます。最前面に表示させたい時はHUDと同様に、2Dとして表示したいと思います。MMDのマーカーもそうですが、視点からの距離に関係なく同じ大きさで表示されています。

このことから、あのマーカーはビルボードというよりHUDの一種だと思います。

じゃあどうやって実装しますか?

そろそろ自分で考えていただきたい。大抵の教えられることは今まで教えてきました。この程度の事であれば自分で考えて、もうそろそろセンサーの手を離れるべき時です。

ヒント…

- シェーダは HUD 用のを使用すると楽
- 4点クアッドポリゴンを使用
- 点(座標)に対してモデルの WVP を乗算した座標を中心に 2D ポリゴンを表示
- 最後に深度を無効にして表示

たまには自分で考えて実装しようぜ。

ちなみに

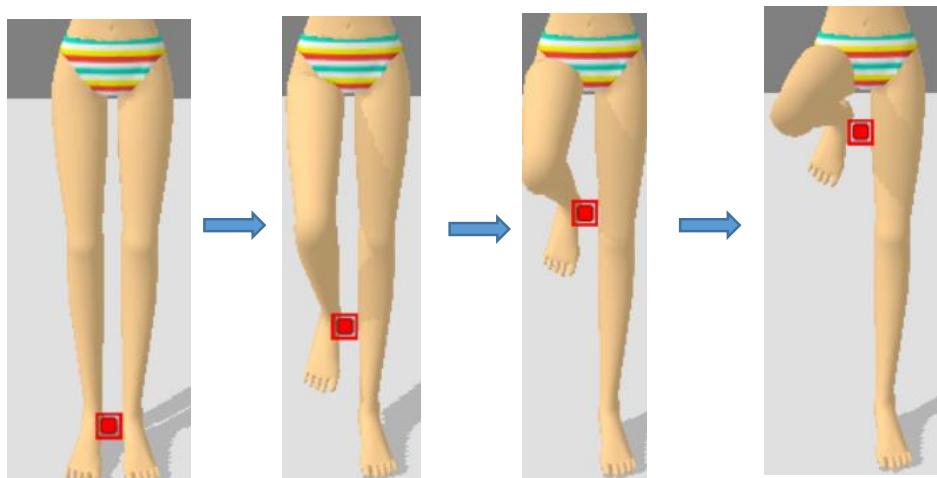


こういう画像を dds としてサーバーにアップしているのでよければ使ってみてください。
"dds"という拡張子は駒染みがないかも知れませんが、DirectX 用の画像フォーマットです。

dds が分からなければ自分でマーカーを作って構いません。フリーにテクスチャとしてロードすれば良いんですけどね？

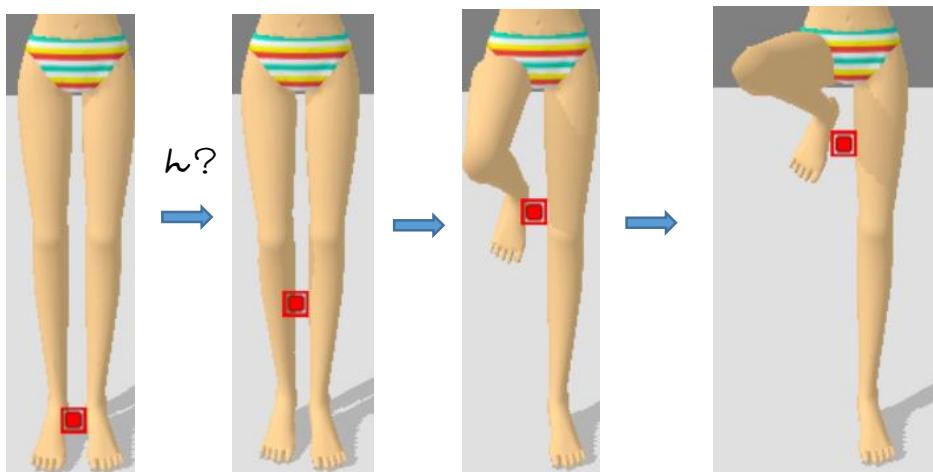
ともかく、実装してみて確認してみました。

右足の IK にマーカーを合わせています。



一応、マーカーに合わせて足が上がっているのが分かると思います。

それではサイクリックなしと比較してみましょう。



一目瞭然ですが、特に二番目を見るとおかしなことが分かりますね？マーカーが膝に来るまでは全然動かさず、そこを超えるとIKとして追従します。

というわけで、ソリューションとしては、制限角度をつけたうえで2で割るということのよう

です。
どこかに明確な使用があればいいんですけどね。ホント手探りですわー。

今回のマーカーみたいに、挙動を一目で比較できる「デバッグツール」を自分で考えておいたほうがいいでしょう。

仕上げ(軸の補正)

足を斜めにした時に応するコードを書きましょう。

以前も書きましたが、このままでは足を開いた時に、足の角度が不適切になります。IKについていきません。



これに対応するためにはループに入る前に最根っこボーンとIK(元の場所、移動先)のベクトルについて外積を取り、回転を計算します。

//ボーンの根っこ部分(IKから最も遠いボーン)からIKの元の座標へのベクトルを作つておく(軸作成用)

```
XMFLOAT3 ikOriginRootVec = ikOriginPos - tmpBonePositions(iklist.ikchainLen - 1);
```

//ボーンの根っこ部分(IKから最も遠いボーン)から移動後IK座標へのベクトルを作つておく(軸作成用)

```
XMFLOAT3 ikTargetRootVec = ikTargetPos - tmpBonePositions(iklist.ikchainLen - 1);
```

この2つのベクトルから回転行列を計算しましょう。

```
XMMATRIX matIkRot = LookAtMatrix(Normalize(ikOriginRootVec), Normalize(ikTargetRootVec),
XMFLOAT3(0, 1, 0), XMFLOAT3(1, 0, 0));
```

とやってもいいし、

2つのベクトルの軸と回転を取得した上でその回転行列を求めて構いません。

```
rootAxis=Cross(Normalize(ikOriginRootVec), Normalize(ikTargetRootVec));
rootAngle=XMVector3BetweenNormals(ikoriginrootvec, iktargetrootvec);
matIkRot=XMMatrixAxisAngle(rootAxis,rootAngle);
```

こんな感じ。

いかがですやろ？

これで得られた軸と角度で、「ひざ X 軸」に補正を行ってやる。

そうすると



ぐらんのように、完璧ではないですが、きちんと IK に追従するようになります。

仕上げ(ベクトル最大長による IK 位置の補正)

足の長さは変わらないので、IK の場所を足の長さを超えるほど長くならないようにしましょう。

大した話ではないです。所謂「CLAMP」すればいいのです。



こうですか？

ちがいます。

これですか？



ちがいます。

値に上限と下限を設けてやればいいのです。XNAMathには便利な関数があります

XMVector3.ClampLength(ベクトル, 最小長, 最大長)

[https://msdn.microsoft.com/ja-](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3clamplength(v=vs.85).aspx)

[jp/library/microsoft.directx_sdk.geometric.xmvector3clamplength\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3clamplength(v=vs.85).aspx)

という関数があります。

その名の通り、ベクトルをクランプした結果を返してくれます。

ターゲットを動かすときに、元のオリジナルの長さ(膝が曲がってない)を上限としていれば、それより遠くに行くことはありません。

例えば

```
float ikmaxLen = Length(ikOriginRootVec);
```

とでも設定しておいて

```
vec=XMVector3.ClampLength(vec, 0.1, ikmaxLen);
```

とでも書いてあげればいいのです。

もちろん、こいつはベクトルなので、

```
XMStoreFloat3(&ikTargetRootVec, vec);
```

と書いてやれば、元の float3 型変数になります。ただしこいつはベクトルであって、座標でないため、このベクトルを作ったもとの座標を足してあげる必要がありますので、注意してください。

```
ikTargetPos = ikTargetRootVec + tmpBonePositions(iklist.ikchainLen - 1);
```

とでもやれば、補正後の IK ターゲット座標が得られます。

まだまだ問題が残っていますが、結構きりがないし、僕にもなんとかわからないし！
力不足ですみません。

ここで今更ですがルートシグネチャについて

そろそろなんとなく見えてきたかも…と思いつつ。やはり良く分かってない。

そもそも「DirectX12におけるルートシグネチャは何のためにあるのか」は

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

を見ると

『Direct3D12の新しいリバインドの仕組みでは、頻繁に更新されるパラメータに GPU のレジスタやリネーミングパスが効率よく動作できるようにするため、シグネチャとパラメータという2つの概念が追加されました。』

うーん。何かしら理由はあるんだろう。ただ、この程度の理由にしては煩雑すぎる…。

次に

<http://www.4gamer.net/games/210/G021013/20160318178/>

にも

『DirectX12で効果的にされたのが、『Root Signature』という仕組みだ。これは何かを簡単に説明すると、シェーダが使う定数をレジスタにバインドするという、DirectX12で実装された機能なのだが、これを用いることで、シェーダの最適化が行えるようになったという。』

『DirectX11までは、こうした細かな指定はできなかった。正確には、DirectX側が自動的にやってくれていたので、アプリケーション側で最適化することがそもそもできなかったそうだ。それが、DirectX12では指定できるようになったので、シェーダの効率を上げられるようになったというわけである。』

らしい。うーん。ハードウェアの事が分からんとこの辺の意義が全く分からん。



要はレジスタの扱いを自由にできるようになったからシェーダに使用するレジスタへのアクセスを最適化できるようになったという理解で…今の所はオッケーかな。それにしても煩雑すぎる…。

ともかく使っていく時に気を付けるべきことはなんかリッファ関連はいちいちルートシグネチャに登録しなければならないっぽいです。面倒です。ただ、おそらくそれをすることでア

クセスのスピードが効率化されるとかそういう事ではないかなあ…と思ひます。

それなりにルートシグネチャについては分かつてきたり

何度かリファクタリングとかしているうちに分かつてきたり

はっきり言って、ルートシグネチャのせいでクラス設計まで複雑になってしま…公式ドキュメントでは

[https://msdn.microsoft.com/en-us/library/windows/desktop/dn899123\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899123(v=vs.85).aspx)

等となっている。

で、定数バッファのリファクタリングの時にも見てきたように、それぞれの役割としては

- 定数リソース(バッファ): GPU 側が使用できるバッファ(CPU 側から確保する)
- デスクリプタヒープ: バッファを GPU 側からどう見るか、どう使うか
- ルートシグネチャ: レジスタ番号など、GPU 側から見るように必要なパラメータをデスクリプタと関連付ける

という理解でいいかなと思います。

また、良い解説を見つけたので、シェアしておきます。

<http://dench.flatlib.jp/d3d/d3d12/descriptor>

でもでも、やっぱりさっぱりわからん人のために

とはいっても、僕もよく分かつてないのですが、流石にここまでやってくると何となく理解が深まってくるもので、今一度解説します。

理解しづらい概念としては

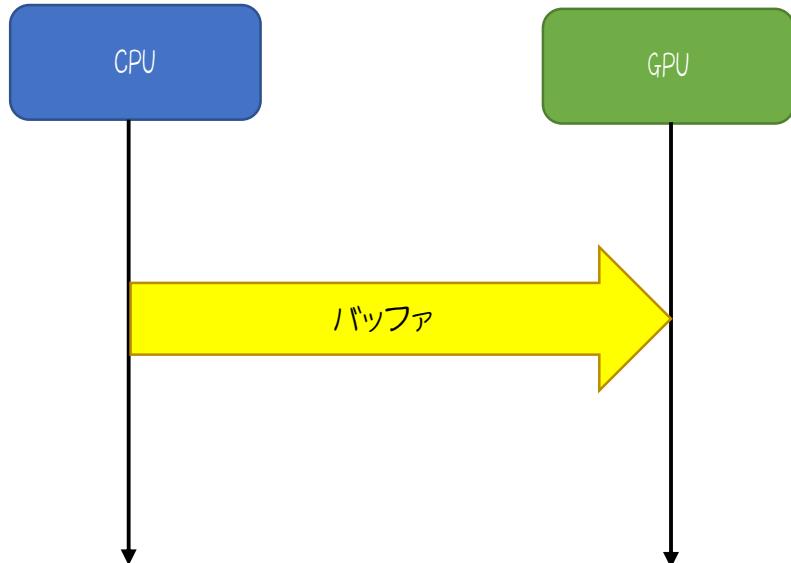
- ルートシグネチャ
- バッファ(リソース)
- デスクリプタヒープ
- パイプラインステート

があると思います。まあ、バッファはともかくなあ…。

バッファはとにかく頂点だろうとテクスチャだろうと定数の集合体だろうと区別してない。とにかく GPU に送る情報のために CreateCommittedResource で作られるのが「バッファ」である。これは多分↑の中では一番理解しやすいだろう。

バッファについて

今まで DX12 以前に作ってきた「バッファ」の概念とちょっと違う。



そもそも「バッファ」という言葉の日本語の意味は「緩衝材」である。つまり直接のやりとりをするのではなく、一時的なメモリに情報を退避させておいてその内容を別の所から利用するといった具合だ。

だから厳密に言うと、ただ適当なアドレスにメモリを確保しただけではそれはバッファとして成立していない。例えば直接アクセスが憚られる 2 者がいるとすると、この 2 者がその確保したメモリを通して情報をやり取りする事により「これはバッファである」と言える。

分かりやすい例がデータのロードである。何かしらの外部ファイル(HDD 内のファイルなど)を利用する場合、例えば `printf` などでファイルの内容を表示したいとする。その場合はほぼ必ずメモリ上に読み込んで、そのメモリの内容を `printf` するだろう？この時の一時メモリの事をバッファというわけだ。

そう考えると、CPU と GPU のやり取りにおいて、そのまま情報をやり取りするのではなく、VRAM 上の特定の場所をバッファとして利用するから、そこを書き換えてくれたら GPU 側はそれを見て処理をするよってなメモリ上の場所が DirectX における「バッファ」なのだ。ちなみに VRAM ってのは表示用に確保されている GPU 上のメモリの事である。

まとめて言うと、ここでいうバッファってのは GPU と CPU の橋渡し役とでも思っていただければいいんだろう。

イメージが湧きにくい人はパンコ屋のやり取りをイメージしてくれ。

パチンコ玉をそのままお金に換えると賭博罪なので、おまわりさんは発砲します。



ところが不思議な事に間に一枚がますと途端におとなしくなります。



ジャパンの法律良く分カラナイのですが、そういう仕組みデス。

直接にやっちゃうとマズくても間に面倒な中間者を置けば通ることが結構あります。マネーロンダリングなんかもそうですよね。どちらも「公然とした抜け穴」です。

パチンコ店で言う所の「景品」が CPU と GPU の間の「バッファ」と思っていただければよいのではないでしょうか?

社会もプログラムも似たような仕組みで動いています。あとついでに数学もね? 微分の時を

考えてくれよ？

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

こんななんやん？そもそも $h=0$ にしてもええんやけど「0 除算したら数学の神様が怒り狂いほ
すよって、中間者として『ホンマは0 やけど、一時的に0 じゃない扱いしてや』という変数 h を
用意しますで？」

で、うま～～いこと分母の h が消えたら、

『すんまへん、実はそいつ0 やってん』と言って、 h が乗算されてるものを消していく…そういう
寸法なんや。数学の神様も騙せるんやからそりやプログラムで使われないはずがない。

まあ、ちょっとこれは脱線なんやけど、バッファってのはそういう『そのままでやり取りが困
難な場合に、間に入れると便利な中間者』と思ってもらえばいいよ。

さて、先に説明したこの『バッファ』…そのままで使い物にならない…いいね？
アツハイな皆様は、私と一緒にもう少し地獄に付き合ってもらうよ。

デスクリプターヒープについて

うん、これは DX11 における「ビュー」の存在を知つてれば理解しやすいかなと思うんだけど…まあ 2 年生の大半は知らんやろし、あまりビュービュービュービュー言つてると混乱させそうなので、さっさと説明しますわ。

こいつはな? 先に説明した「バッファ」を GPU 側がどう解釈したらいいのかを設定するものなんやで。まあ~意味合い的には「頂点レイアウト」に似てるかなあ。

どっちにしても「バッファ」単品じゃ事実上使い物にならへんというわけや。例えば人間にとつてデータの塊なんていうのは色気も何にもないものですわ。



00101101101010101011111100
100011010011011011011101010
1111101101101101101110101010
0000001110000101010101011
1010101110000000011100100
00010100111100010011010001
0101011010101010101010101
10010101010101010101010101

ところがこいつに意味付けと言うか解釈のルールと言うか、そういうのをやってやると



明らかにありがたさと言うか、価値が変わってくるわけだ
だが、それに子だらうとぼちゃ子だらうと読み方が分からなければ単なる二進数である

何度も言うようだけど「データの塊」そのものには価値がない。復号化できない暗号を施されたようなもんですわ。

で、そのデータの塊の「解釈の仕方」を GPU 側に教えるのが「デスクリプターヒープ」である。名前の由来は正直良く分からぬ。というかこの名前のせいで「何者なのか」「どういう役割

なのが』を分からなくなってる人も多いと思う。俺もその一人だ。

で、色々な資料を読んでも正直サッパリではあるのだが、とりあえずプログラムを書いていく過程で理解が深まつたりしたので、その辺を説明していこう。

先にも書いたがしばらく使って分かった『意味』は

バッファの解釈の仕方を GPU に教えるためのモノ

である。まさに DX11 の『ビュー』そのものなのである。

名前が良くない…

- Descriptor(寄附記述子)
- Heap(積み上げる、ヒープメモリ)

であり、『解釈の仕方』を仄めかすような意味はない…ないんだよ。とりあえず無理やり解釈するなら descriptor は『describe するもの』と考えてみる。そうすると describe ってのは『言い表す、説明する』『描写する、表現する』『特徴づける』『類型化する』『図を描く』という意味があるらしい。やっぱり『解釈する』的な意味合いはないのだが、あえて言うなら『説明する』であろうか。『やってきたデータについて GPU に説明する』と言ったところか。

ところで後半のヒープは当然のようにヒープメモリだと思うんだが、なんでそういう『ヒープ』状態になっているのかというと解釈がひとつじゃないからである。そもそもこの『解釈』には『バッファのどの部分(アドレス)』からがその『データの始まり』であるかという情報も含まれる…BufferLocation などというパラメータに覚えはないだろうか?

そう、いくつかのデータをまとめている場合、特に定数バッファの場合には、例えばマテリアルをまとめていたりするだろう。その場合『解釈』は集合体となるため、解釈もまたメモリの塊と言う相當にややこしい事になる。さすがにありがたいことに

GetDescriptorHandleIncrementSize(ヒープ種別)

でサイズは分かるので、解釈が何処にあるのかを探し出し、その解釈を GPU に教えてやって、GPU はその『解釈』を元にバッファを解釈する…というもうなんのこっちゃという仕組みとなっている。

ぶっちゃけ俺自身も何でそういうややこしい仕組みにしているのかは分からぬ。分からぬものがもはやどうしようもない。



ともかく DescriptorHeap はバッファの解釈をするものだと理解しておいていい。

そして更なる混乱を引き起こすのが、この DescriptorHeap を作るためにどのような解釈をすればいいのかの設定を行うのだが、そしてそいつの型の名前は…

お 前 はもうちよつと考へて名 前 つけるや
D3D12_DESCRIPTOR_HEAP_DESC

である。



そんなん敬遠されるわ…。

もうちよつと考へて名前つけるや。だって最後の DESC って DESCRIBE の略語なんですよ?『説明する』『特徴づける』って意味の…絶対に名前の付け方を失敗しとるやろ…これ DX13 になつたら消えるか変わるかするんじゃないかな?

いやホント、変数名つけるときに相当悩むんですよ本当に。

まあでも Microsoft 御大がつけた名前である以上は、それを利用する我々はそれに従わざるを得ないんだろう…OTL

さて、このクソのようにややこしいデスクリプターヒープもだいたいその「役割」についてはご理解いただけたのではないかと思う。

ちなみにこのデスクリプターヒープは GPU からも CPU からもアクセス可能である。この「解釈」の情報には CPU 用と GPU 用があって、それも使い分ける必要があるのだが、それがそれぞれ

GetCPUDescriptorHandleForHeapStart();

と

GetGPUDescriptorHandleForHeapStart();

である。

で、こいつから得られるのはぶっちゃけた話単なるアドレスである。定義を見てもらえば分かるが、内部にポインタを持っているだけである。なんでこんなケッタイな型にしてるかっちはじめ、ほかの用途に使わせない…ただそれだけだと思います。

あと、デスクリプターヒープの本体は「ハンドル」のポインタですので、そこはよろしくお願ひいたします。

デスクリプターヒープはその程度の理解でいい…と思う。よろしいか？

よろしい、ならばルートシグネチャだ

改めてルートシグネチャについて

こいつも名前が良くないと思うのです。ひとまずそれはさておき、おおざっぱな意味を説明しましょう。

リソース(リソース)と GPU のスロットを結びつけるための仕組み

です。ここまでプログラムしてる人ならピンと来ると思いますが、

```
Texture2D<float4> tex:register(t0);
Texture2D<float4> spa:register(t1);
Texture2D<float> shadowmap:register(t2);
Texture2D<float4> lut:register(t3);
Texture2D<float> doutline:register(t4);
```

```

cbuffer mat : register(b0) {
    matrix world;
    matrix viewproj;
    matrix lightviewproj;
    float3 diffuse;//拡散反射成分(基本色)
    float alpha;//拡散反射成分(基本色)
    float specularity;//スペキュラ強さ

    float3 specular;//スペキュラ色
    float3 ambient;//環境光色

    int existTex;//テクスチャあるか
    int existSPA;//スフィアマップあるか
    float3 eye;//視点
    float3 lightpos;//ライト座標(仮)
    int existToon;//

}

cbuffer bone : register(b1) {
    matrix boneMat[512];
}

こいつ(定数バッファやテクスチャバッファのスロット)とそれぞれのバッファをバインドする
(紐づける)ための仕組みが「ルートシグネチャ」ですよ、と。

```

とりあえずそういう理解でいいと思います。そういう理解であるならばルートシグネチャのキモはどこにあるのかと言うとルートシグネチャの設定の時に出てくる

DescriptorTable

です。もっと言うとこの中の

D3D12_DESCRIPTOR_RANGE

なんですが、この中のパラメータにシェーダレジスタを設定する部分があるので、そいつに割り当てます。

という風に解釈すると、意外とルートシグネチャってのは大した事ない！という風に思えてくるんじゃないかな。

要はシェーダ側で記載しているレジスタ番号に合わせて、ルートシグネチャのデスクリプターテーブルのレンジの配列インデックスを、実際にバッファを割り当てるときに

```
_commandList->SetGraphicsRootDescriptorTable(cbo->GetParameterIndex(), handle);
```

と書けばいいわけで、↑のコードの第一引数はルートシグネチャの配列インデックスで、ハンドルってのはその「バッファの解釈情報」を置いてあるハンドルを指定しているわけです。

で、このハンドル部分には、バッファのアドレスとその使い方が格納されており、それによりGPUからバッファを正確に解釈して利用することができるわけです。

最後にパイプラインステートですが…まあこれはそれほど難しくはないと思いますって言うか皆さんだいたい分かりますよね？

パイプラインステート

パイプラインステートの役割はひとことで言うと

メッシュを表示するときの設定をまとめたもの

です。もうそれだけですよ。ちなみに設定すべきものは

- シェーダ
- デプス
- 頂点レイアウト
- ラスタライザ設定
- レンダーターゲットフォーマット設定
- プリミティブトポロジー

てな感じです。

ちなみに面倒なんですが、このパイプラインステートを作る際にはルートシグネチャと紐づけておく必要があるので、順序としては

ルートシグネチャの生成→パイプラインステートの生成
の順序で行う必要があります。

まあこんな感じです。

ここまで言っても正直良く分からん人は正直に言ってください。分からんまんま卒業させる
のも俺としても後味悪いんでさあ…。

綺麗なコード？を書く

よく、就職の面接練習の時に「きれいなコードを書くことを心がけています」と言つて言うんだけど、きれいなコードってなんやねん？

それぞれの定義によって曖昧なのよね。これ。更に言うとこれって「当たり前」の事だから、直接言うなとは言わんけど「売り」にするのは良くない。他にとりえがないように思われるだろう。

課題提出のごあんない

さて、最終課題の提出時期が近づいてまいりました。

この授業は単位数が半端じゃないので、提出しなければ自動的に進級条件が偽(false)となります。

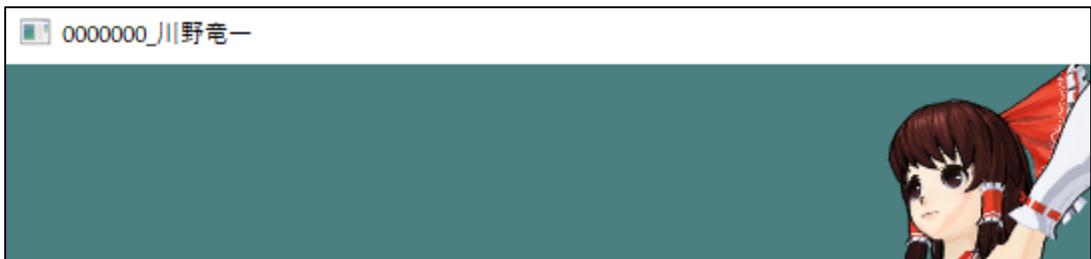
みんなとサヨナラしたければ、どうぞ。次年度に上がりたければ、とにかく提出してください。これについて抗議があれば本日中に言ってください。それがなければそれでは承したものとします。

提出期限：2/2(金)18:00まで。

提出場所：[¥132sv¥gakuseigame¥game¥b44 教室¥川野¥後期最終課題](#)

レギュレーション(提出要件)

自分の学籍番号と名前を書いたフォルダを作つて、自分の作ったEXEと実行に必要な素材を入れてください。



アプリケーションのタイトルバーには、自分の学籍番号と名前が書いてあるようにしてください。これががない場合はもう受け付けませんのでご注意ください。

最低限実装されているべき機能

- モデルの表示(カラー、法線、テクスチャなどの質感も)

- 光源と法線によって陰影がついている
- スキニング(ポーズやアニメーションがMMDで設定したとおりになるように)
- シャドウイング(シャドウマップもしくはステンシルシャドウなど)が出ている

できれば実装されているべき機能

- LUTによるトーン表現
- 反転法による輪郭線表現
- テプスによる輪郭線表現
- MSAA

頑張って実装してほしい機能(A~B以上)

- IK(インバースキネマティクス)
- モーフ
- インスタンシング
- 別スレッドでデータ読み込み
- DirectX12を利用して何かしらゲームを作る

これを自前でできたら素晴らしい(無条件S判定)と思う機能

- FootIK(段差などで自然に足を設定)
- PBR(物理ベースレンダリング)
- レイマーチング法を利用した雲表現など
- ナビメッシュを作つてそれを元に経路探索
- GPUパーティクル
- SSS(サブサーフェスキャッタリング)
- EffekseerをDX12化
- 自分でエンジン作っちゃう
- その他なんかすごいやつ(凄さが分からんかったら説明して)
- 商用ゲームと言って差し支えないゲーム

ともかく、最低限の要件はこなしたうえでレギュレーションを守つて、提出期限までに提出してください。

就職に向けて

はい、ここから専攻科3年生とクリエイティブ2年生は本格的に就職活動が始まっていくと思います。ひとまず今後は自覚を持って就職活動をしていく準備をしなければなりません。

<https://note.mu/gamerdj/n/nb1eebf5b5183>

という記事を見つけましたので、一通り読んでおきましょう。一応ゲームプログラマを目指すひとのための記事ですが、ゲームプランナーとか、その他の職業に就いても大枠は似たようなモノなので、読んでおきましょう。

まあ、この時期にきて、具体的に受験計画を立ててないのは論外だと思います。
学校が紹介してくれる？そんな受け身の人間は多分受からない。受け身の姿勢が治った時に受かり始めるかもしれませんけどね。

ここまで何度も面接練習をしましたけど、何處を受験するのか考えてない人が多すぎる。

就職活動はなあ……戦争なんだよ!!!

戦術が必要なんだわ。戦争っていう比喩が苦手なら「お見合い」なのだ!! 戦争もお見合いも情報戦なんだよ!!! 想像してみ？お前はお見合いの相手も知らんでお見合いすんのか？するかもしれへんけど…イヤだろそんなの？

いやさ、「俺は誰でもいいから結婚したいんです」とて人もいるかもしれないけど、そんなの絶対後悔するし、お前「誰でもいい」とて奴がOKもらえると思ってんのか？

PBR 自前で実装してた〇氏ですら、その辺を見誤ってちょっと苦労してたっぽいよ。

〇 先輩を越える奴なんて、そうそういねーだろ？なのに何でどこを受験するのかとか考えてない人がいるんだろう…うまくいくわけねーだろそんなの…。

逆に自分がどういう仕事をしたくて、それをやるためににはどういう会社に行ったら良さそうなのかを想定して、そこで受験してうまく行くように戦術を立て、作品作って勉強して行ったら確率はグーンと上がるんや!!

もちろん戦術を立てるには会社の情報が必要なので、会社のWebページを隅から隅まで見て、どういう会社か把握している事が重要なんや。ここで間違えてほしくないのは隅から隅まで研究して、書かれていない事まで推理して、確率を上げる。そこまでしないと君らは受からんのよ。実際の話。

何度も言うけど、ライバルは九州大学の理系の奴らとか、会社によっては東大京大ってこともあり得るんやで？そんなダケモンと比べられるんだから戦術立ててないと勝ち目ねーよ。



だからやれ、今すぐにだ

あと、会社選びの注意点だけど

楽しそうな職場
安定してそう
丁寧に教えてくれそう

とかに惹かれていくのはやめておいた方がいいだろう。自分がやりたい事ができそうかどうか、自分が成長したい方向に成長できるかどうかで決めよう。

というか上記のキーワードに惹かれる奴は、会社への依存心がぶんぶんするからだ。こういう動機で会社を決める奴には、だいたいエンジニアとしてロクな奴はない。覚えとけ。

