



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!……求人件数は増えているが、求められるレベル自体は下がっていない。それどころか
年々上がっている。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。そこでここに 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: ライブラリ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

どうみても数え役満レベル。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 | 7 |
| 予定 | 7 |
| 膨大なる用語(本当にすまない!) | 9 |
| 環境設定 | 16 |
| ウィンドウを出すまで | 18 |
| アプリケーションのハンドル | 19 |
| Direct3D の初期化 | 27 |
| Direct3D の初期化について | 28 |
| Direct3DDevice | 28 |
| コマンドまわり | 34 |
| スワップチェイン | 40 |
| ディスクリプタとレンダーターゲット | 46 |
| レンダーターゲット | 51 |
| ルートシグネチャー | 54 |
| 画面に影響を与えよう(画面を特定の色でクリア) | 57 |
| レンダーターゲットクリアコマンド発行 | 57 |
| フェンス | 63 |
| そもそも非同期処理とは? | 63 |
| で、結局 DirectX12 ではどうなの? | 65 |
| フェンスの仕組み | 68 |
| ではフェンスを実装しようか | 68 |
| ポリゴンを表示しよう | 70 |
| 頂点情報を作る | 70 |
| 頂点レイアウト | 72 |
| 頂点バッファ | 74 |
| 頂点バッファビュー | 76 |
| そんな事よりシェーダ書こうぜ | 77 |
| シェーダ読み込み | 78 |
| パイプラインステートオブジェクト(PSO) | 79 |
| その他やらなければならない事 | 81 |
| リソースバリア | 81 |
| ビューポート | 82 |
| 残り色々セツト | 83 |

| | |
|----------------------------------|-----|
| テクスチャマッピング | 88 |
| UV 座標を付加..... | 89 |
| 頂点情報構造体に UV を追加..... | 89 |
| 頂点情報に UV 座標を追加..... | 90 |
| 頂点レイアウトを追加..... | 90 |
| シェーダ側の引数を追加..... | 90 |
| テクスチャリソースの作成..... | 91 |
| ビットマップ読み込み..... | 94 |
| テクスチャバッファへの書き込み..... | 95 |
| シェーダリソースビューの作成..... | 96 |
| サンプラ | 98 |
| サンプラの設定..... | 98 |
| ルートシグネチャへサンプラを適用する..... | 99 |
| 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..... | 100 |
| シェーダリソースビュー用のデスクリプタを登録..... | 101 |
| 色化け対処 | 103 |
| ビットマップ以外への対処(今はおまけ的な話)..... | 107 |
| 3D 化してみよう..... | 108 |
| 行列について再学習..... | 108 |
| 行列による座標変換..... | 110 |
| アフィン変換(アフィン行列)..... | 111 |
| それぞれの行列を作ってみよう | 115 |
| ワールド行列..... | 115 |
| カメラ行列(ビュー行列)..... | 116 |
| プロジェクション行列(射影行列)..... | 117 |
| 行列を合成しよう..... | 118 |
| 座標変換データを GPU に送ろう..... | 120 |
| 定数バッファを作ろう..... | 120 |
| ちょっとバッファとビューについてのたとえ話..... | 125 |
| メッショの表示..... | 127 |
| メッショ(PMD)の表示..... | 128 |
| 4バイトアライメントに注意..... | 131 |
| インデックスさんデータを使って「面」を表示していこう | 137 |
| 立体感つけよう..... | 139 |
| ワールドと、ビュープロジェクションを分割..... | 141 |
| 深度バッファの冒険..... | 144 |

| | |
|---------------------------------|-----|
| 深度バッファとは..... | 144 |
| 結局 DX12 では何をしなければならないの? | 146 |
| 深度バッファの作成..... | 146 |
| 深度バッファビューの作成..... | 147 |
| パイプラインステートオブジェクトに深度情報を追加..... | 148 |
| レンダーターゲットと深度バッファを関連付け..... | 148 |
| 深度バッファをクリア(毎フレーム)..... | 149 |
| 色をつけよう..... | 150 |
| 準備 | 152 |
| DX11ほど甘くない色分け… | 155 |
| まずはディフューズ成分を GPU に投げよう | 155 |
| なんで? | 157 |
| 先人のコードを見てみよう | 157 |
| じゃあ俺実装 | 159 |
| 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 | 161 |
| クラス設計 | 164 |
| PMD ファイル情報からテクスチャをロード | 168 |
| じゃあテクスチャを回してみよう | 169 |
| ボーン(骨地獄) | 174 |
| ボーンって何? | 174 |
| スキニング(キンメッシュアニメーション)とは? | 175 |
| ボーン情報 | 176 |
| ボーン情報の「表示」 | 178 |
| ボーン用シェーダ | 181 |
| ボーンの回転 | 183 |
| ツリー反映の準備 | 188 |
| ツリー構造の構築 | 189 |
| 再帰 | 192 |
| 親子構造の中で複数の回転を行う | 194 |
| 最後に補足 | 197 |
| 妙なクラッシュ | 197 |
| 対処法 | 199 |
| SIMD 命令(SSE)とは | 201 |
| スキニング | 202 |
| 頂点データについて | 202 |
| ボーン ID | 202 |

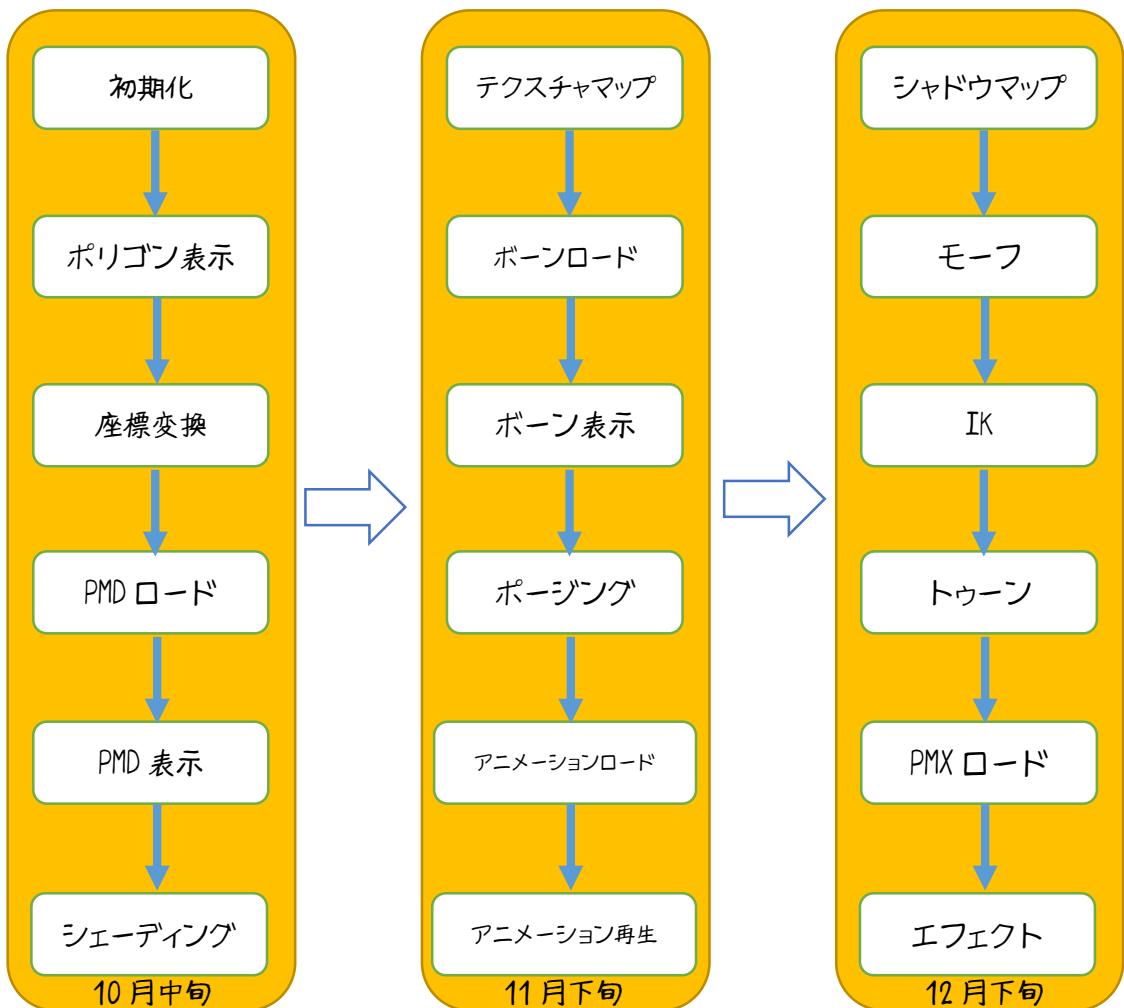
| | |
|--------------------------------|-----|
| ボーン ID に悩まされる..... | 202 |
| min16uint の検証..... | 204 |
| ボーン用定数バッファ(ボーン数×行列)を作る..... | 205 |
| 実際にページングしてみよう..... | 209 |
| リファクタリング(コードをキレイにしましょう)..... | 212 |
| 待てあわてるなこれは vector の罠だ..... | 212 |
| 次は定数バッファ(テクスチャも合わせるか?)周り | 216 |
| BMP 以外も読めるようにしよう | 220 |
| LoadWICTextureFromFile..... | 221 |
| MutiByteToWideChar..... | 222 |
| 進行とか知るか! やが! そんな事より実験だ!..... | 224 |
| ここで今更ですがルートシグネチャについて | 227 |
| それなりにルートシグネチャについては分かつてきた..... | 228 |
| クオータニオン..... | 228 |
| 球面線形補間..... | 228 |
| アニメーションデータ(VMD)のロード | 228 |
| アニメーションを再生..... | 228 |

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つていきましょう。

環境設定

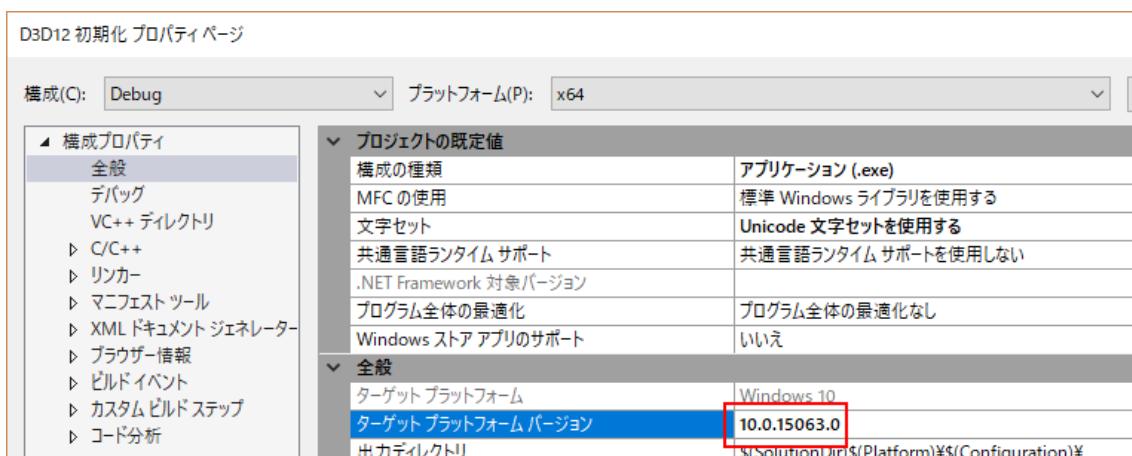
DirectX12 を使えるようにするまでの環境設定が結構ややこしい。

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。

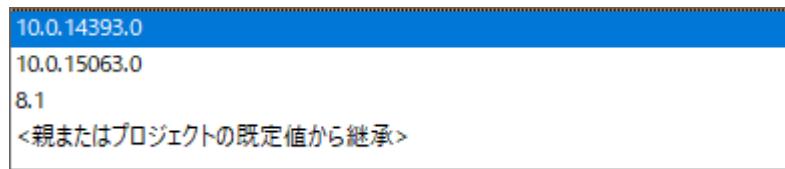


自分のここがどうなっているのか確認してください。10.0～になつていないと、DirectX12 の開発ができません。

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)。



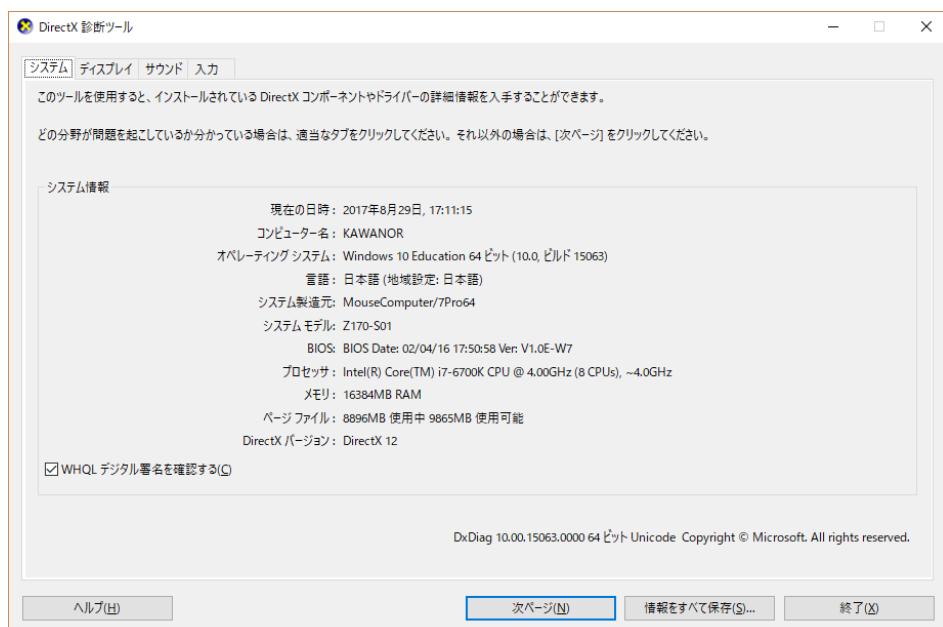
この辺の作業は、最初からターゲットプラットフォームバージョンに 10 番があれば必要ない作業ですが、如何でしょうか？

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。

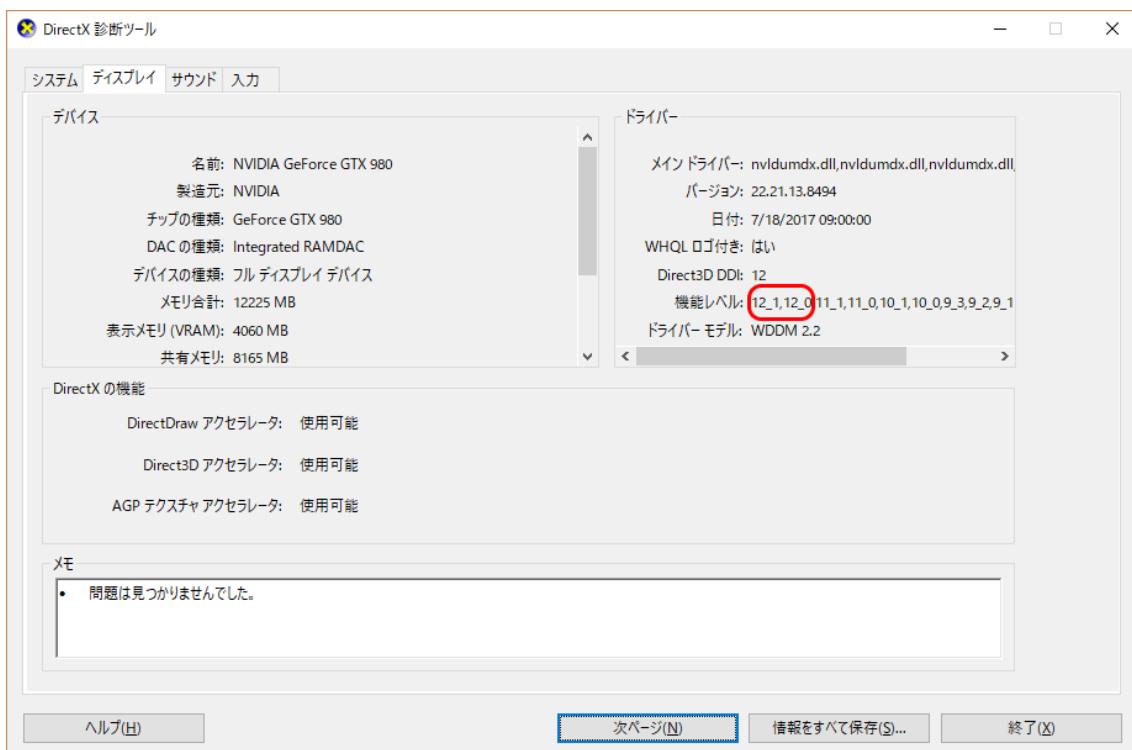
少なくとも家の PC は最新版でやっておくことをお勧めします。が、授業では 10.0.14393.0 をベースに進めていこうと思います。

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、ここで問題といふか、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つたりしますので、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ョボくなりがちです。

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ないのかです。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。

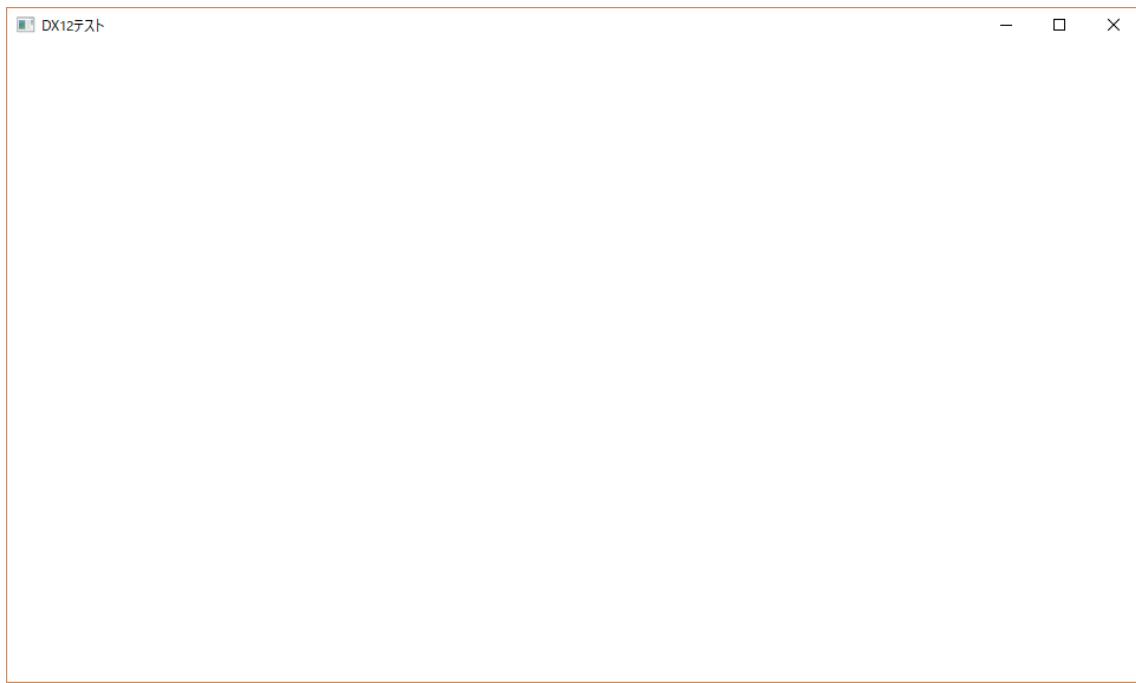
次に DispatchMessage

<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ね…(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。口く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。

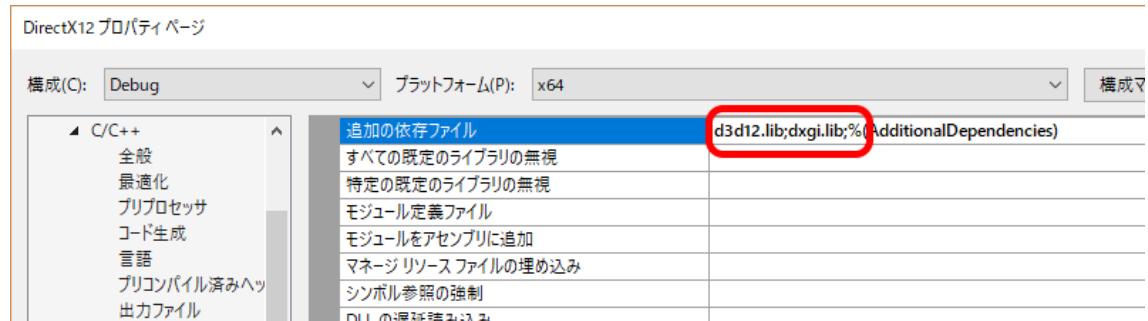
LNK2019 未解決の外部シンボル _D3D12CreateDevice@16 が関数 _main で参照されました。

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ければだ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。もしできてない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? 即攻撃になつ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。

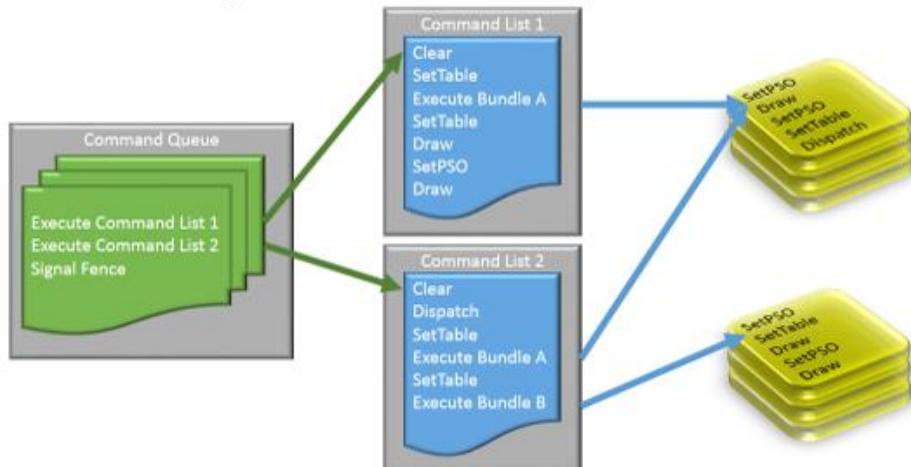
そういう理由があって、命令をため込んでおいて一気に GPU に転送するために「コマンド バッファ」を使用するわけですが、その名前が「コマンドリスト」になっているとでも思っておけばいいです。

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

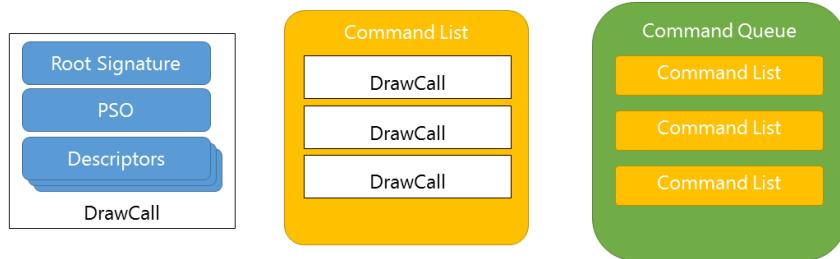


<https://www.isus.jp/games/direct3d-12-overview-part-6-command-lists/>

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          iid,  
[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。」

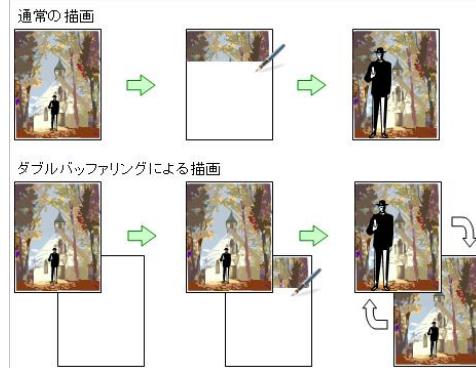
らしいです。今回は普通に GPU を使うので、ここでは 0 にしておきましょう。さて、ここまでヒントから CreateCommandQueue を作つて、S_OK が返るか確認してみてください。

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

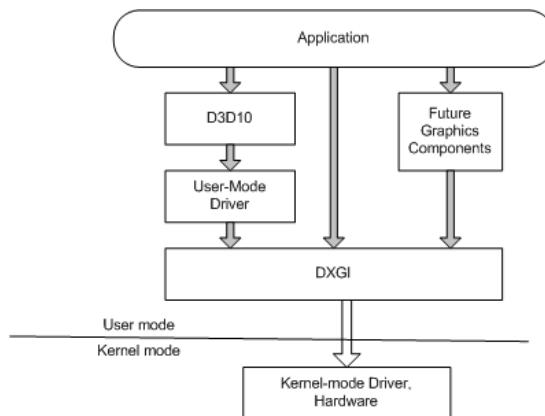
等という聞いたことのないものになっている。これは DirectX11 でもそうだったのだが DXGI という概念に軽く触れておく必要がある。

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/625_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をそのまま言うと

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、ディ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います。これが出て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。

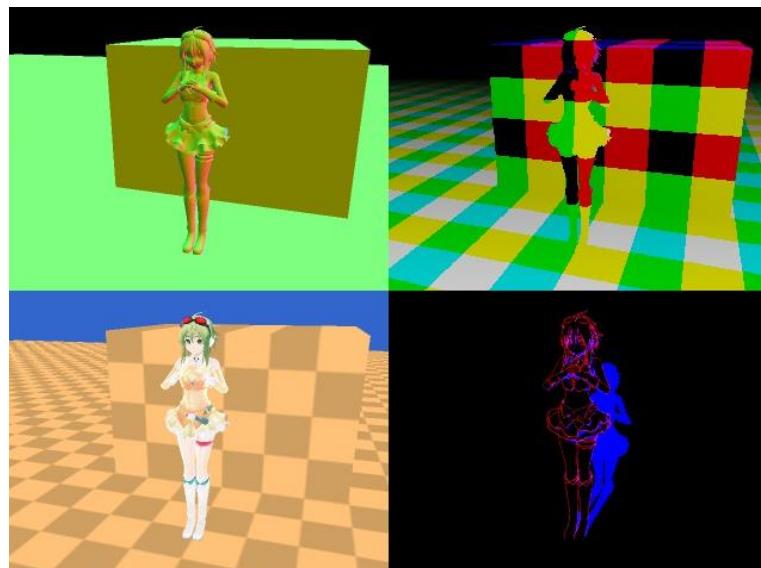
そもそもなんでこんなややこしいことをする羽目になったのかというと

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。

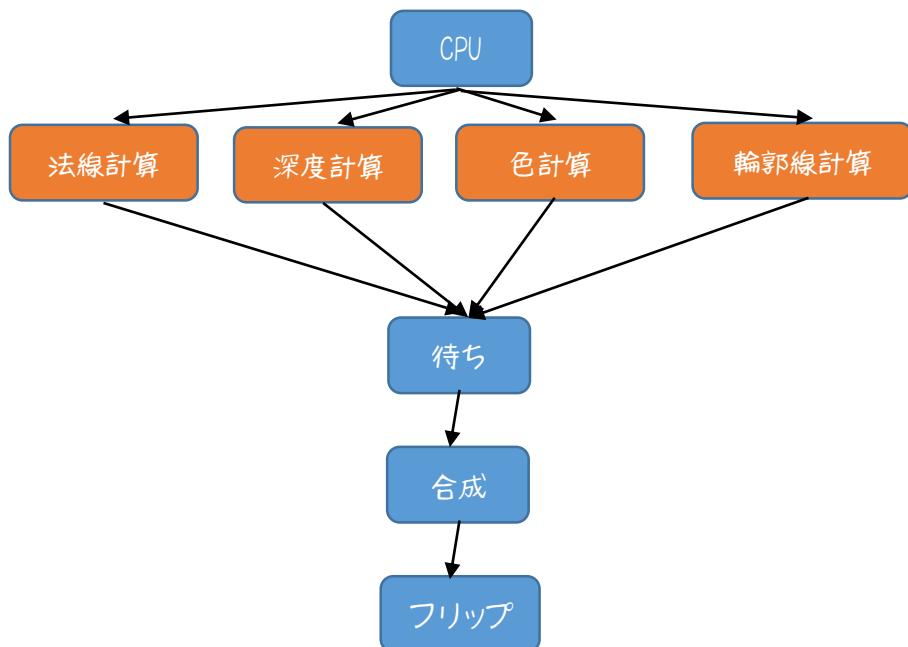
意味が分からぬんだろうから簡単に言うと。

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



すつづ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…これやつちやうとぶつちやけビジー状態になりっぱなしになるので、どうかとは思うけど、ゲームだから CPU 占有しちゃってもいいか。

…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ます。

「ウフフフフフ…フフフフフ…ヤ…ウフフフフ」

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

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

さて次に頂点レイアウトの定義ですが

D3D12 INPUT ELEMENT DESC

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

で定義します。これもDX11のやつを参考に見てみます。

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

まず、分らない用語が出てきます。

「セマンティクス」ってなんや？

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

「HLSL セマンティクス」で検索すると

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

POSITION(座標)などの COLOR(色)などが出てきます。

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…キツツク的なホント。
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);
```

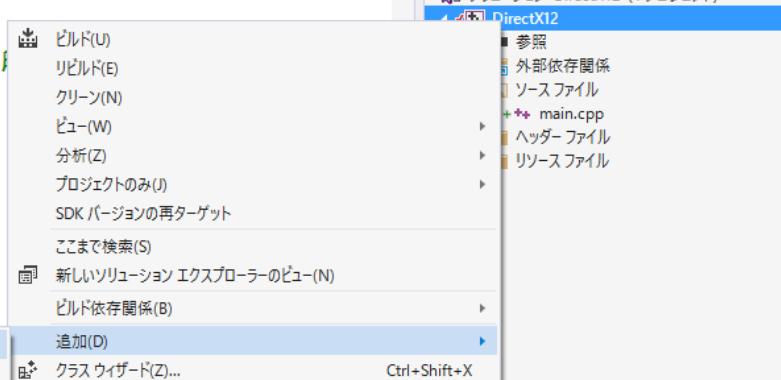
てな投げ方をするんですが、それはもうちょっと後でやります。

そんな事よりシェーダ書こうぜ

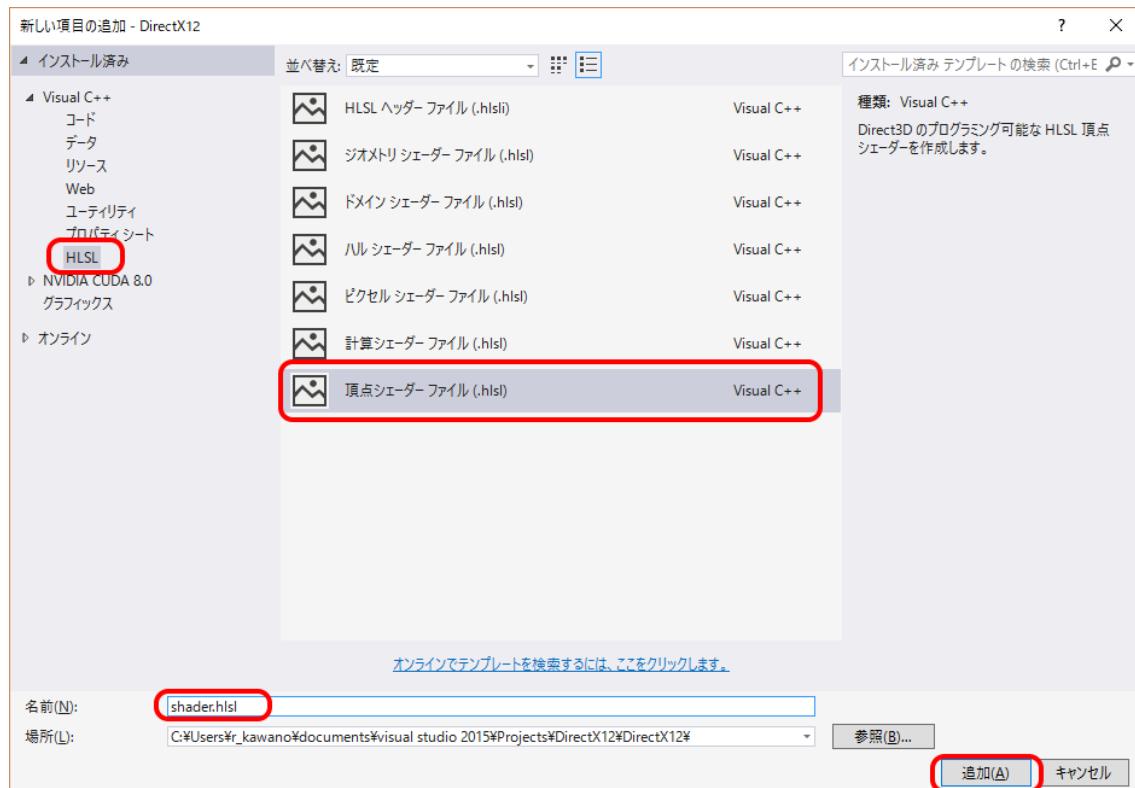
AD), //CPUからGPUへ転送する
3)), //サイズ
,ない

```
buff);
```

新しい項目(W)... Ctrl+Shift+A

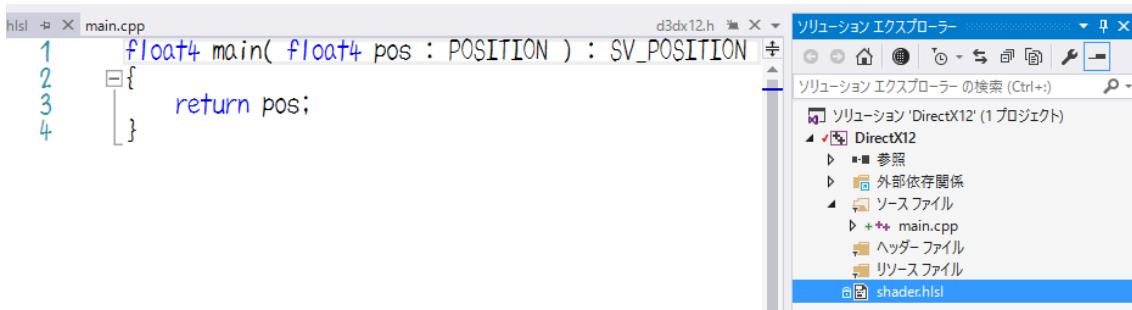


プロジェクトで右クリック→追加→新しい項目を選ぶと

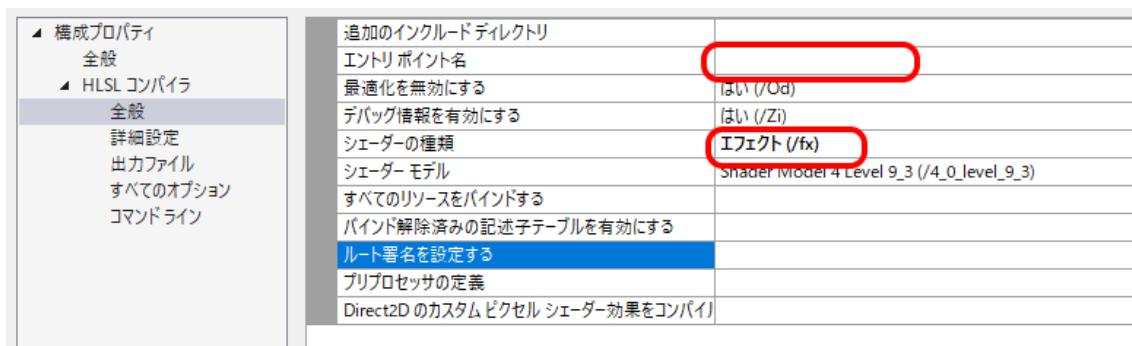


こんなのが出てくるので頂点シェーダファイルとして追加してください。とはいって実際には

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 コンパイラー→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を自分で書いていきます。まだ
シェーダの書き方を知らないと思うのでこう書いてください。

あと、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] ピック / 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

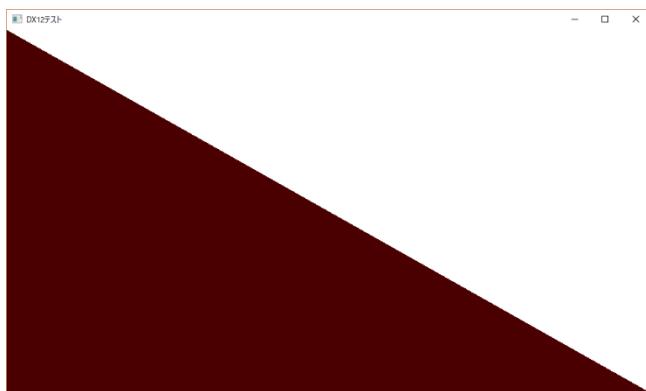
[https://msdn.microsoft.com/ja-jp/library/ee419594\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419594(v=vs.85).aspx)

_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…

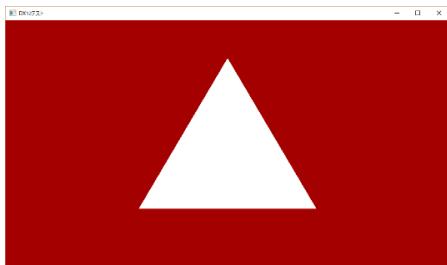
こんな感じの表示になります。右上の白い部分が今回描画している三角形です。



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

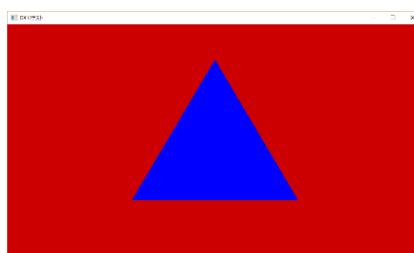
```
Vertex vertices[] = { { { 0.0f, 0.7f, 0.0f } },
{ { 0.4f, -0.5f, 0.0f } },
{ { -0.4f, -0.5f, 0.0f } } };
```

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;
}

```

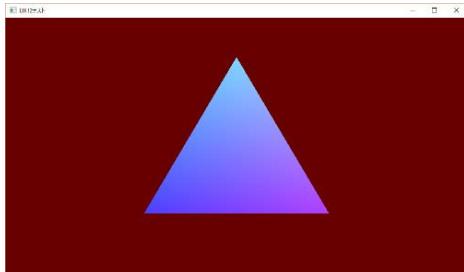
//ピクセルシェーダ

```

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)
```

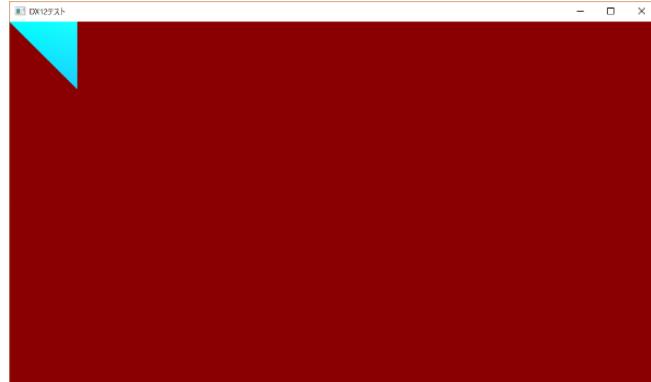
とします。これが何をやっているかは後で話しますが、これの何がうれしいかと言うとピクセル数値通りに画面上に表示されるようになります。そうするとHUD的なUIを作るときに正確に置けるのでこれを覚えとくと重宝すると思います。

この状態で

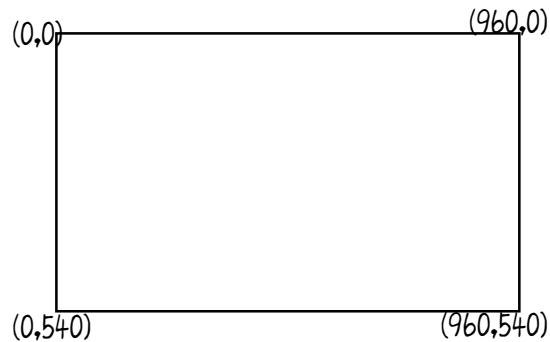
```
Vertex vertices() = { { { 0.0f,0.0f,0.0f } },  
                      { { 100.0f,0.0f,0.0f } },  
                      { { 100.0f,100.0f,0.0f } } };
```

という指定をすると

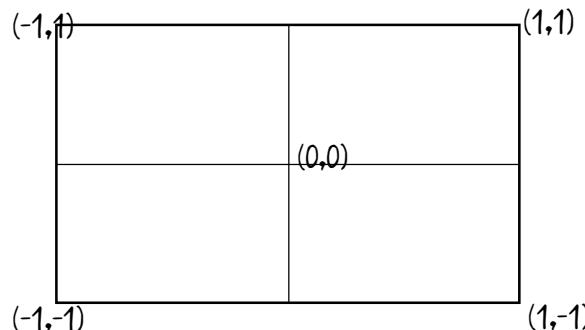
このように思った通りの場所に表示されるようになります。



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0~1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。-1~1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0~2の範囲になるように割り算を行い、そこから-1をすれば-1~1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0~2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…頂点リソースの時は

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。

サンプラー

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ーを使用してテクスチャマッピングを行います。

サンプラーの設定

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 になります。

ちなみに、番号の前の t だの s だのに関しては

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

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

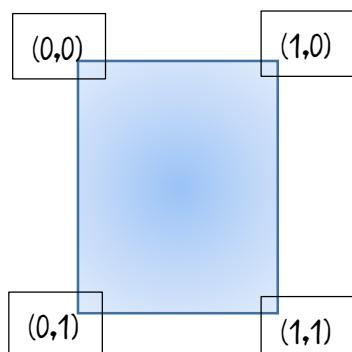


こういう風になってて正解

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>

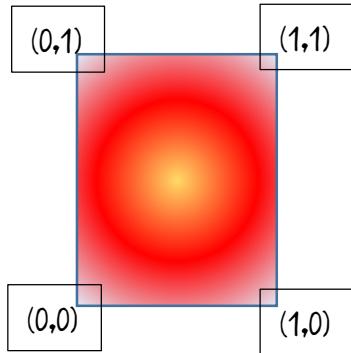
通常であれば2DやUVの座標系は…



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

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

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

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

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、プロジェクト行列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)

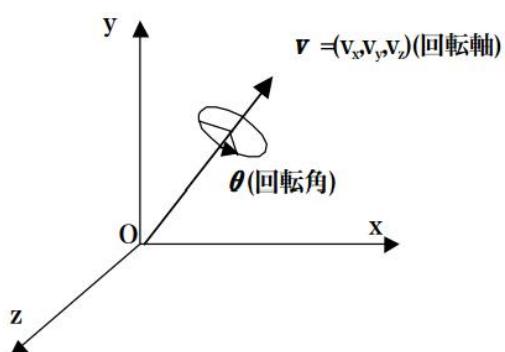
次にカメラ行列を作ります。XMMatrixLookAtLH 関数を使用します。

[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。

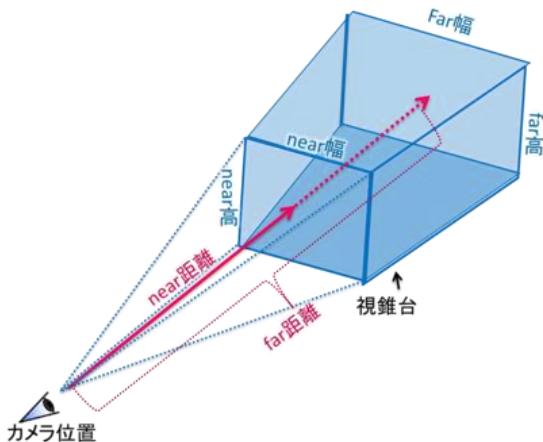
[https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixlookatlh\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixlookatlh(v=vs.85).aspx)
XMMATRIX camera = XMMatrixLookAtLH(eye, target, upper);

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

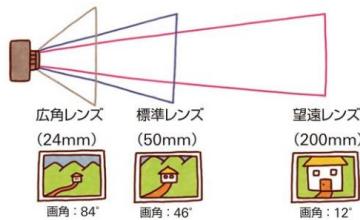


無理くりスクリーン板の形にしてしまうような変換です。見ての通り手前は小さく遠くは大

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っちゃくなります。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っちゃと作ります。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ひますが、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、そういうふう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次元なんでもいい

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
}
```

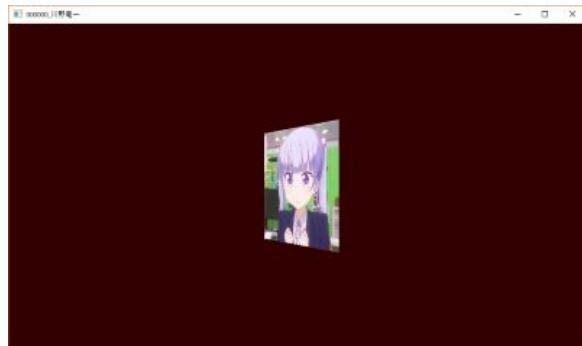
で、頂点シェーダの中身だが、引数の座標を表すパラメータをposとすると、posに対して行列の内容を反映させる。乗算だ。この乗算の順序がCPP側と逆になるので注意。

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。



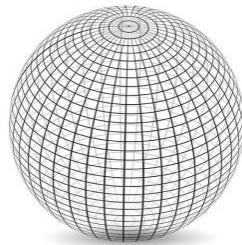
このとおりである

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。

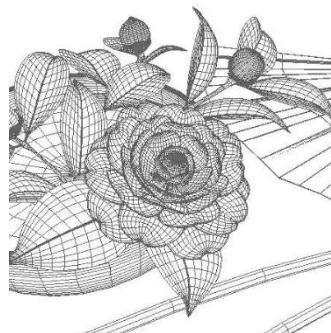
何故かと言うと



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)、あとか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

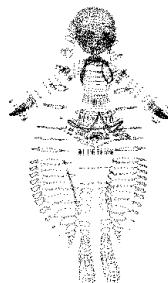
正直どうやって解析したのか分からん。が、ありがたく利用させてもらいます。データは

- 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ない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?ひとつひとつでやつ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 から狂つてくるわけです。

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…処理系依存というべきか…まあ難しい話なんですわー。

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);
```

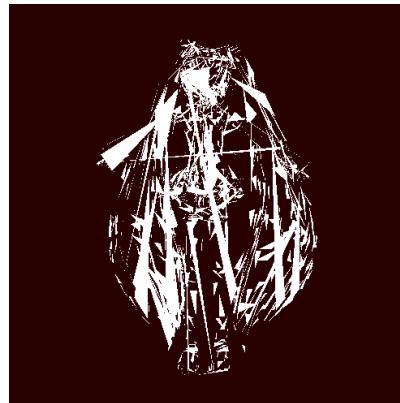
とやります。つまりパック1にして

```
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);
```

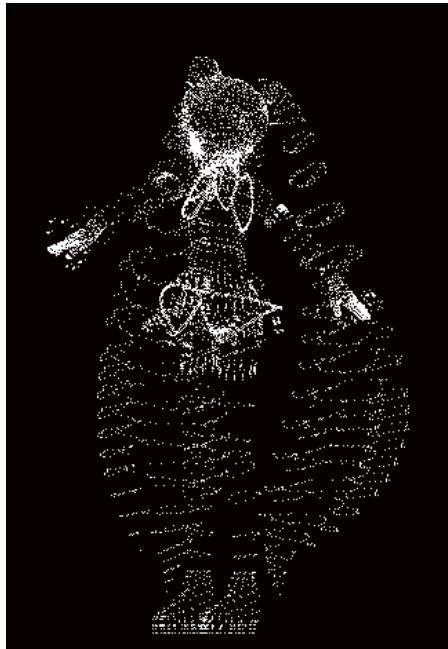
あとは描画部分を修正します。DrawInstancedです。

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、その直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 int 4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。

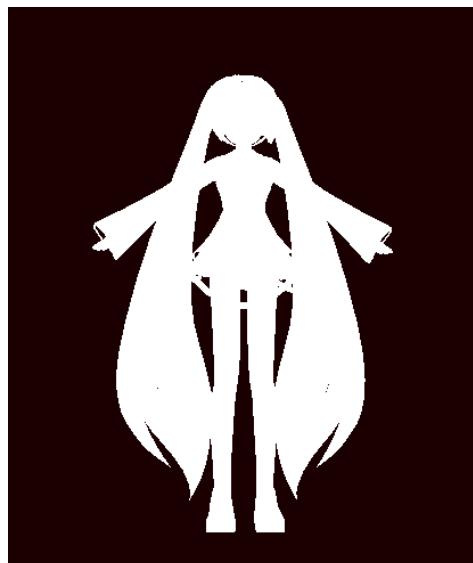
最後にデータのマップやね。これも頂点バッファの時と同じでいい。

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。

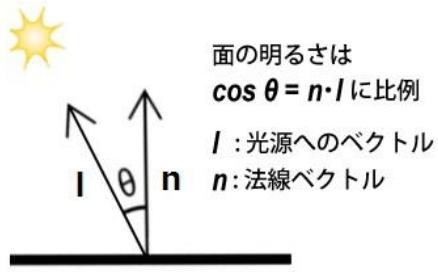
ここで CG の基礎となる有名な理論を紹介しておきますが、Lambert シェーディングとか Lambert の余弦則とか呼ばれる理論です。

<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。

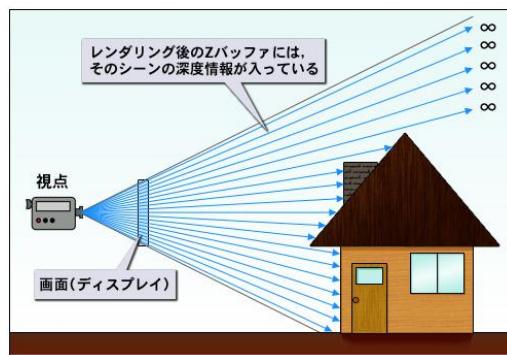
深度バッファの冒険

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。



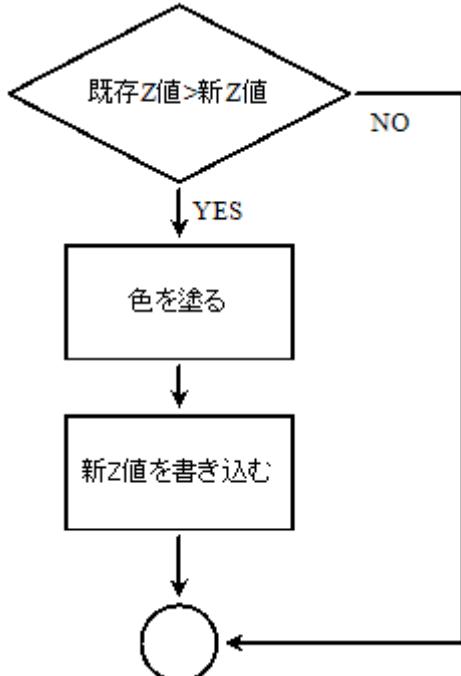
で、この深度バッファがどのように働くのかというと、今からそのピクセルを書こうとすると
きに Z テスト(深度テスト)というのを行います(タイミングとしてはピクセルシェーダ後…
ラスタライザが終わった段階でテストすりゃいいのに…DX12 をクソ難しくする暇あつたら
この辺どうにかしろよ)。

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っているんでしょう。

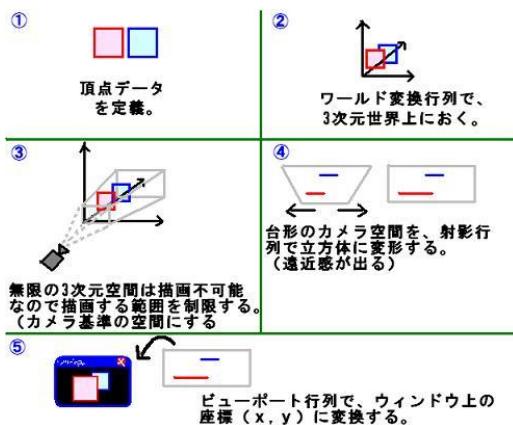
さて、というわけでちょっと不思議なバッファ…デプスStencilバッファを作っていくましょ。え？ 深度バッファじゃないの？ という疑問はごもっともではあるのだが、なんかそういう仕様になっているのだ。ちなみに深度バッファは通常の float と同様に 32bit のバッファである。

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

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

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

以前に、「プロジェク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。

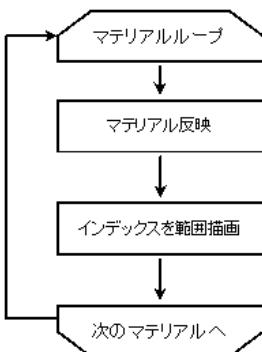
さて、マテリアルデータを利用してポリゴンに色を付けるとすると、どういう手順を用いればいいかなあ…？

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{  
    XMFBLOCK3 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。そなまくいかないのだ。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(ジュース)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つたのだが、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「ツグ」が紛れ込む…私は深刻な「ツグ」だと思いつ込み、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。

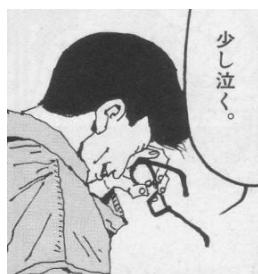
描画部分を

```

_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\を削られればいいので、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);
```

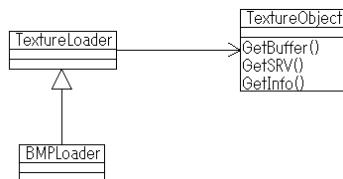
何をやっているかは…分かりますよね？ここまでできれば「かわいいミクさんが表示されますよ。



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

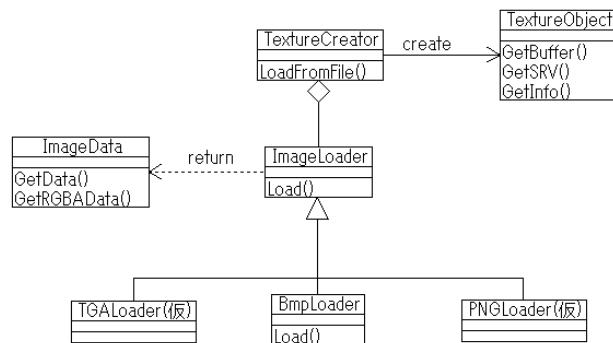


こんな感じ。今回は BMP から読み込んだので BMP ローダを作ってるがたぶん TGA だの PNG だのは必須なので、拡張可能にしている。

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、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 も用意されていません」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。

とりあえずそれぞ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] == '\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)
```

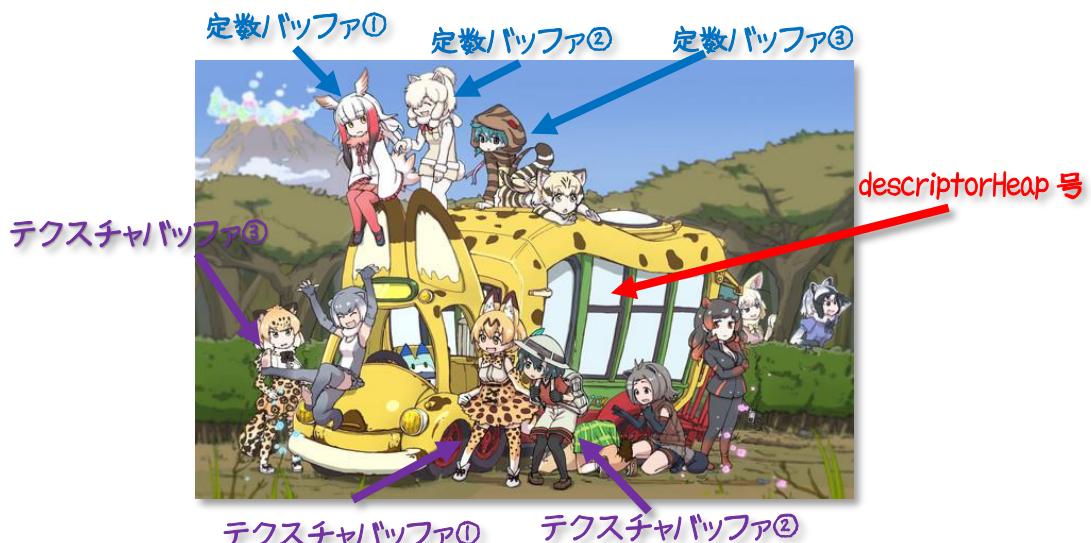
```

///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。

地獄入ナニマ

ボーン(骨地獄)



という事でボーン(bone)の話です。

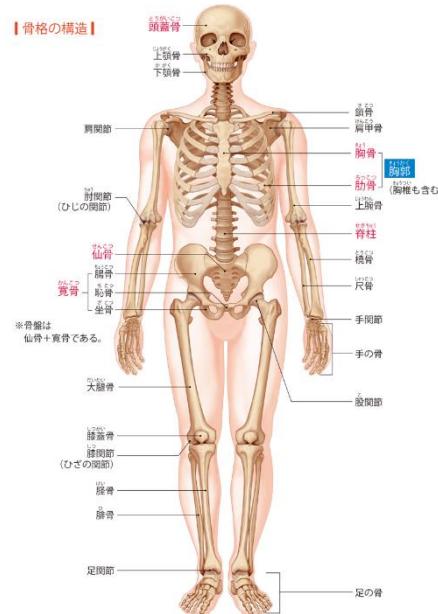
ここからは地獄です。おそらく骨しか残らない人も多いのではないかなどと思います。でも骨は捨ってやるから安心しろ。



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



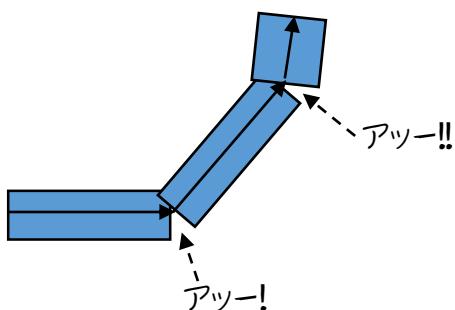
ああ～くっそかわいいんじゃ～

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？

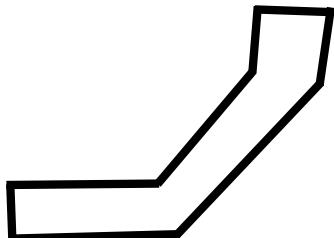
ポージングをするためには結局のところ、ボーンに頂点がくっついていかなければならぬわけだけど、きれいに「関節らしく」動かすためには1つの頂点に関わるボーンが一つでは足りないのである。なぜならば…



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「データ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。

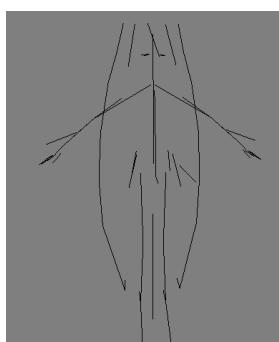
つまりロード用としては

```
//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
};
```

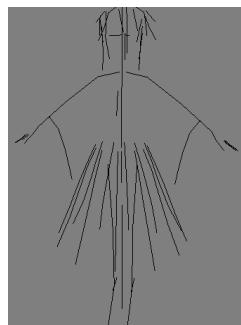
とします。そして表示したい情報はとりあえず座標だけなのでひとまず

```
struct Bone{
    XMFLOAT3 headpos;
    XMFLOAT3 tailpos;
};
```

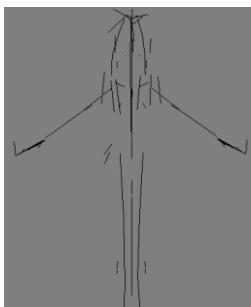
とでも定義する。で、ここに値を入れて、頂点バッファ作って、ボーン用レイアウトと、頂点シェーダ作って、うまいこと表示するとこうなります。



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

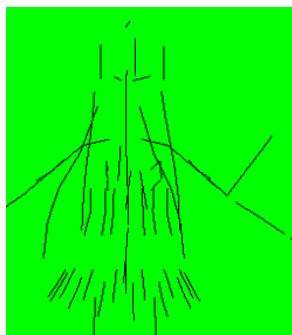
①「左ひじ」を検索しインデックスを得る

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

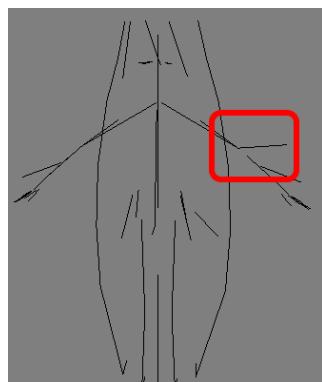
③頂点再セット

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

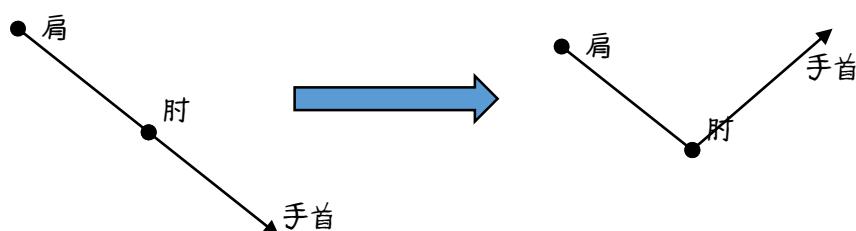
[https://msdn.microsoft.com/ja-jp/library/ee420742\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee420742(v=vs.85).aspx)

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。



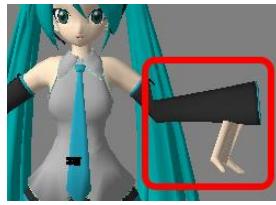
さて、現在の状況を見てみればわかるように、腕のボーンが折れ曲がっています。腕のボーンが折れ曲がるということはどういう事がというと…



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。

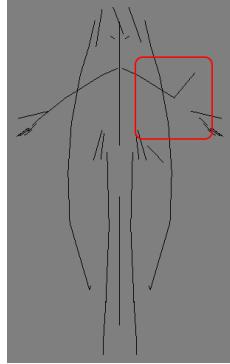
参考コード

//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

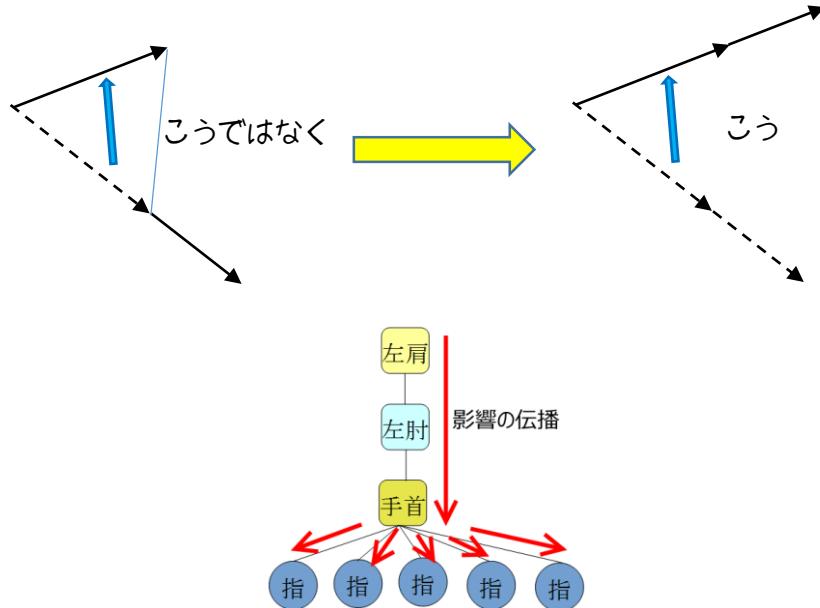


肘を曲げたら当然肘から先まで回転しなければならないのに、前腕ボーンだけが曲がっています。これでは腕が千切れても仕方がない。

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。

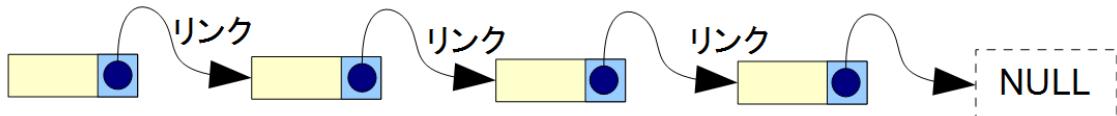
こういう感じなのを効率良くやるには、今回話している「ツリー構造」と「再帰」の理解が必要です。

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))

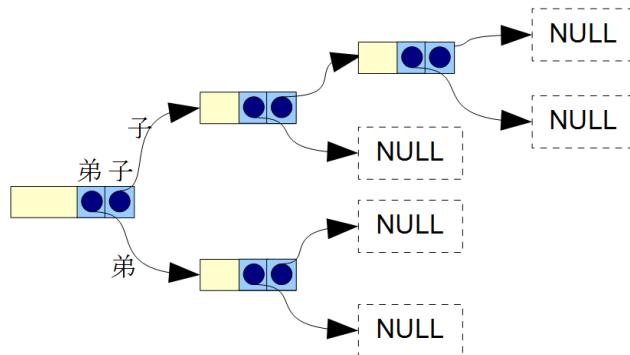
リンクとだいたい同じなんんですけど、下の図のようにリンクは一つのノードから一つのノード

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
う。

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

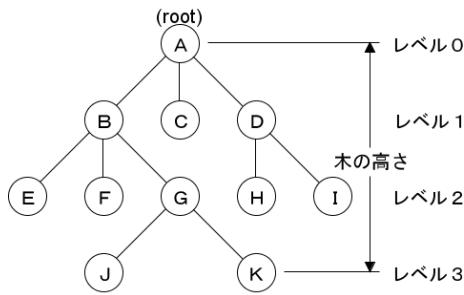
「左ひじ以降」というのはツリーにおける左肘から先のノードのことです。

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);

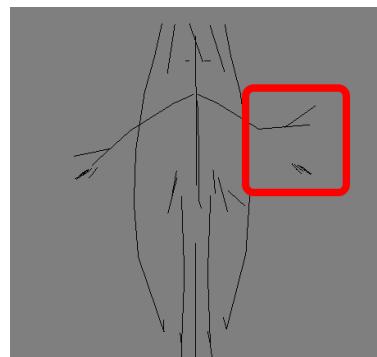
XMMATRIX bone = XMMatrixTranslationFromVector(-offsetVec);
bone *= XMMatrixRotationZ(XM_PIDIV4); //45° 回転
bone *= XMMatrixTranslationFromVector(offsetVec);
こんな感じでboneって中に肘中心変形が入っています。
実際に変形させる時には

//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。

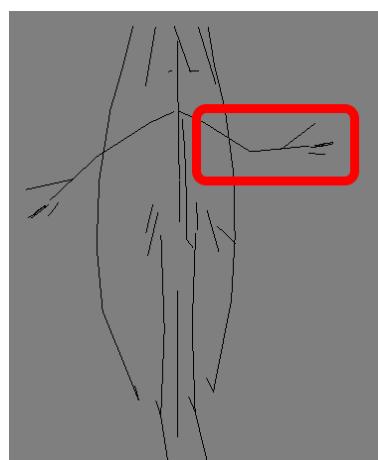
今回の場合次を呼び出す条件は「子がない」ってことですね。

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);

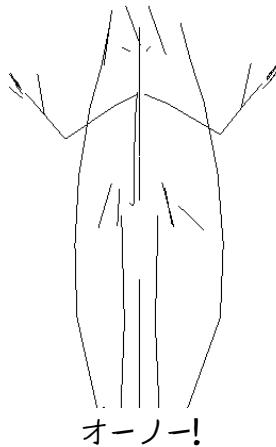
//テールを回転
XMVECTOR tailposvec = XMLoadFloat3(&bones(index).tailpos);
tailposvec = XMVector3Transform(tailposvec, matrix);
XMStoreFloat3(&bones(index).tailpos, tailposvec);

//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

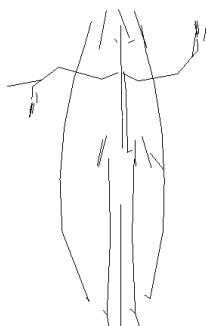
//ボーン曲げ実験

```
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

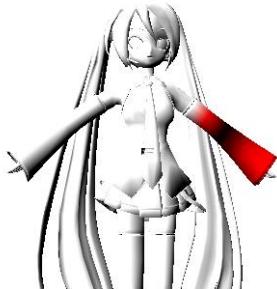
Outputを書き換えて

```
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。

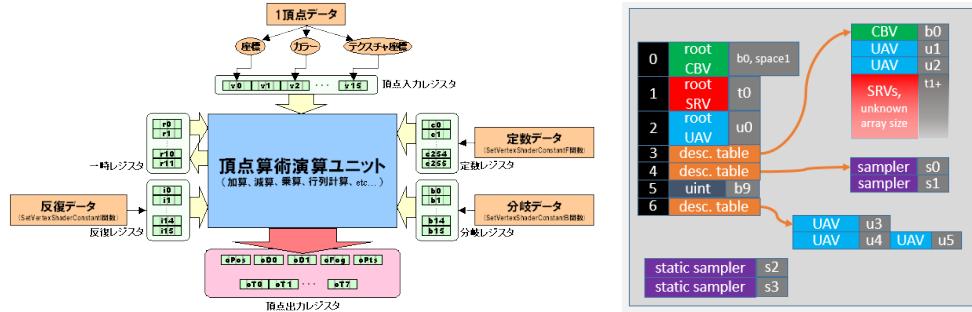
…まあできるのかもしれないけど、今の僕は知らない…と言い続けてはや3年。どうしたらいいんでしょうか?



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;

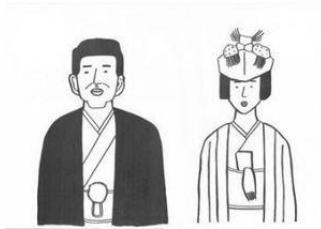
result = dev->CreateCommittedResource(&cbvHeapProp2,
D3D12_HEAP_FLAGS::D3D12_HEAP_FLAG_NONE,
&CD3DX12_RESOURCE_DESC::Buffer((sizeof(XMMATRIX))*_bonematrixes.size() + 0xff)&~0xff),
D3D12_RESOURCE_STATE_GENERIC_READ,
nullptr,

```

```

    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。

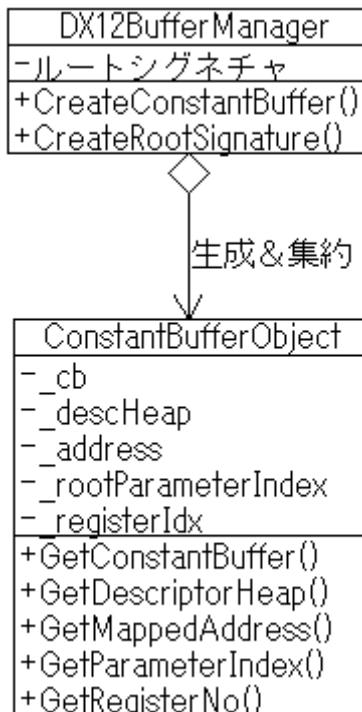
まあ、実はデスクリプターテーブルやバッファビューに関しては「使用者側」からすると、特に必要ない。マップしたポインタも結局は定数バッファから取ってくるんだから、要らないいちやいらない。ただ、これをいちいち必要なときに Map してから取り出そうとすると面倒なので、定数バッファ作った時点でマップしておいてもいいんじゃないかな。

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。結構ハ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。

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 なので、それらのビットマップが読み込めずにこのようないきなりになります。

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という数値になってほしいところですが、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、よくありがちのが、ワイド文字を受け取るバッファを作るときに

```
malloc(sizeof(1パス名)*2);
```

なんてやつっちゃう人がいますけど、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いますが、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。

```
        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。なんだつそら!!!

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、これをどう使うのさ…。

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んだ。

んで、LoadWICTextureFromFile じたれ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、まあ原因じたれ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…

`CD3DX12_HEAP_PROPERTIES defaultHeapProperties(D3D12_HEAP_TYPE_DEFAULT);`

と初期化されました。

ぶっちゃけた話をすると、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いのが本音です。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ったので不本意ですが、解説サイト(日本語)を見てみます。

https://shikihiiku.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…。そういう事か…。

ちなみに ZeroGram 氏も Project ASURA 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、あるのは知ってたんですが、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。

何処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、

- ① LoadWICTexture でロードする
- ② UPLOAD 指定でバッファを作る
- ③ UploadSubresources でデータを流し込む
- ④ バリアで待つ

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

『Direct3D 12 の新しいリbind 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に、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側が自動的にやつてくれていたので、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つたそうだ。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。



さっぱりわからん

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プア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、おそらくそれをして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…と思います。

それなりにルートシグネチャについては分かつてきた

クォータニオン

球面線形補間

アニメーションデータ(VMD)のロード

アニメーションを再生