

2016

# 地獄の DirectX11

君はこの地獄に耐えられるか！？

お花遊園地



# 目次

はじめに .....	11
1.1    だいたいの流れ .....	12
1.2    注意事項 .....	13
1.3    できる奴らは自分で .....	14
1.4    はじめの最後に… .....	17
2    知識的なこと .....	18
2.1    用語 .....	20
2.2    体系 .....	23
2.3    アンテナ .....	24
3    制作準備 .....	26
3.1    Gitについて .....	26
3.1.1    基本的な使い方(ローカル) .....	26
3.1.2    基本的な使い方(サーバー) .....	31
4    ウィンドウの表示 .....	36
4.1    関数ポインタとコールバック .....	37
5    DirectX11初期化 .....	43
5.1    初期化時に必要な用語の解説 .....	46
5.2    初期化中に出てきたOMとかRSとかってなんなのさ? .....	51
5.3    初期化の大まかな流れについて .....	52
5.4    きちんと初期化できているかどうかチェック .....	53
5.5    初期化処理のクラス化 .....	55
5.5.1    シングルトンクラス .....	57
5.5.2    構造体とかクラスのプロトタイプ宣言 .....	63

6	三角ポリゴンの表示.....	65
6.1	頂点を配列で定義.....	66
6.2	頂点/バッファのセット(書き換え).....	67
6.1	頂点/バッファの作成.....	69
6.2	プリミティブポロジの設定.....	70
6.3	頂点シェーダとピクセルシェーダを書く.....	71
6.4	シェーダのロード&コンパイル.....	73
6.4.1	頂点シェーダオブジェクトの作成.....	74
6.4.2	頂点レイアウト.....	75
6.4.3	ピクセルシェーダオブジェクトの作成.....	77
6.5	頂点シェーダ、ピクセルシェーダのセット.....	78
6.6	頂点描画.....	79
7	座標変換な"のです".....	83
7.1	行列とは.....	83
7.2	アフィン変換.....	85
7.3	3D化実装.....	88
7.3.1	XNAMath.....	88
7.3.2	ワールド行列を作成.....	89
7.3.3	ビュー行列を作成.....	89
7.3.4	プロジェクション行列を作成.....	90
7.4	コンスタント/バッファ(CPU側).....	92
7.5	コンスタント/バッファ(GPU側).....	93
7.6	コンスタント/バッファの動的変更.....	94
7.7	おまけ(もうちょっと回転をわかりやすく).....	96

8	PMD ファイルロード .....	98
8.1	バイナリエディタを使ってみよう .....	98
8.2	バイナリファイル(PMD)をロードするやで(°)(°)ミ .....	101
8.2.1	ヘッダファイルをロード .....	102
8.2.2	4バイトアライメント問題 .....	102
8.3	頂点情報をロード .....	105
8.4	なんでこれだけで表示されるのん? .....	108
8.5	インデックス情報とは .....	111
8.6	インデックスさんのロード .....	112
8.7	インデックスを使った面描画 .....	112
9	ここで一旦休憩 .....	114
9.1	STL のお話 .....	114
9.1.1	vector について .....	115
9.1.2	map について .....	119
9.2	コーディング(規約?)について .....	121
9.3	コメントルール .....	123
9.3.1	コメント基本(書くべきこと) .....	123
9.3.2	嘘コメント書くな .....	126
9.3.3	まとめ .....	127
9.3.4	public メンバ関数の仕様はヘッダ側に書こう .....	128
9.3.5	見て分かることはいちいちコーディングしない! .....	128
9.3.6	細けえコメントを書くよりコードそのものをわかりやすくしよう .....	129
9.4	クソコード .....	133
9.5	リファクタリング .....	136

9.6	今回のリファクタリングの注意点.....	137
9.6.1	左辺値がありませんエラー.....	137
10	法線情報を使って立体感与えちゃうぞ!!.....	139
10.1	立体感とは?.....	142
10.2	ランバートの余弦則.....	143
10.3	間違えました.....	146
11	テクスチャ(シェーダリソース)を貼ろう.....	148
11.1	テクスチャロード&セット.....	148
11.2	レイアウト変更.....	151
11.3	UV値からテクスチャの「色」を抜き出す.....	152
12	マテリアルで分けよう.....	154
12.1	マテリアル数ロード.....	156
12.2	マテリアルデータロード.....	156
12.3	マテリアルデータをGPUに渡す準備.....	157
12.4	マテリアル毎に色を変える.....	160
12.5	マテリアル毎にテクスチャを….....	162
13	モデルをフォルダ分けしたい.....	166
13.1	std::string(文字列型).....	168
14	背面カリングについて.....	171
14.1	ラスタライザステートを生成して、セット.....	172
15	うまく表示されない.....	175
15.1	PMD特有のテクスチャルール.....	175
15.2	アルファブレンディングの話をしよう.....	177
15.3	リクエストにお答え.....	182

15.4	スフィアマップ(嘘).....	187
15.5	Lat式ミクさん奮闘記.....	191
16	ボーン .....	197
16.1	そもそもボーンてなんだ? .....	197
16.2	ボーンをいじるだけでいいのか? .....	198
16.3	ボーン情報の取得.....	200
16.4	ボーン情報の「表示」.....	201
16.5	ボーンの回転.....	205
16.6	ボーンツリー.....	208
16.6.1	ツリー反映の準備.....	210
16.6.2	ツリー構造の構築.....	211
16.6.3	再帰 .....	213
16.7	アッ!! .....	217
16.7.1	どういうことなの? .....	218
16.7.2	対処法 .....	218
16.7.3	SIMD 命令(SSE)とは.....	219
17	スキニング .....	221
17.1	ボーン ID .....	221
17.2	最初の困難 .....	221
17.3	影響度 .....	222
17.4	ボーン配列(256 個)用コンスタントバッファ作成.....	223
17.5	頂点シェーダ側での処理.....	224
17.6	重み付け加算.....	227
17.7	後から思ったんだけど… .....	229

18	そろそろまたコード整理.....	230
18.1	リファクタリング.....	230
19	ポージング .....	233
19.1	クオータニオンとは.....	234
19.1.1	複素数のおさらい.....	234
19.1.2	四元数 .....	236
19.2	クオータニオンを使用して任意軸回転する .....	236
19.3	ポーズデータのロード.....	237
19.4	一方、クライアント側では… .....	239
20	アニメーション .....	242
20.1	構造を考える.....	243
20.2	typedef .....	244
20.3	std::mapについてもう少し詳しく .....	245
20.4	フレームレートを安定させる.....	248
20.5	フレームと連動させる.....	250
20.6	algorithm::find_if を使ってみよう .....	251
20.7	std::reverse_iterator(逆イテレータ).....	253
20.8	ボーン間の補間.....	254
20.9	Slerp(球面線形補間).....	256
20.10	VMD ファイルの罠 … .....	257
20.10.1	ソート(std::sort).....	258
20.10.2	そもそも挿入時に並び替えすればよくね? .....	259
20.11	最後の仕上げ .....	259
21	シャドウマップ .....	266

21.1	XMMatrixShadow を使う.....	267
21.1.1	三次元平面の方程式.....	267
21.2	シャドウマップの概念.....	276
21.3	演習準備 .....	281
21.4	エフェイティング.....	287
21.5	2つのレンターターゲット.....	290
21.6	ライトからの見た目.....	291
21.7	レンターターゲットの切り替え.....	292
21.8	影用シェーダリソースビュー .....	293
21.9	ライトビュー用シェーダ.....	295
21.10	ライトビューテクスチャの利用.....	303
21.11	ライトビューテクスチャのUV値は? .....	304
21.12	いよいよ比較だ.....	305
22	デバッグ用HUDを作る.....	308
22.1	HUD行列を作る.....	311
22.2	HUDシェーダ作る.....	311
22.3	HUD頂点データ(頂点/ドッファ)作る.....	312
22.4	ループの前で全部そろえとく .....	312
22.5	描画 .....	313
22.6	おかしなシャドウへの対処.....	315
22.7	ライトビューを引く .....	315
22.8	ライトビューの画角を広げる.....	316
22.9	Orthogonal(平行投影).....	316
22.10	点光源もどき .....	318

22.11	まだまだおかしい円柱の影.....	318
23	何度目のリファクタリングだ? .....	321
23.1	ちょっと突っ込んだ基本の話.....	321
23.1.1	テンプレート.....	324
23.1.2	コピー構造関数.....	334
23.2	コーディング規約.....	335
23.3	設計 .....	336
23.3.1	典型的なやつ.....	336
23.3.2	XMMATRIX に悩む.....	344
23.4	Warning を黙らせる… .....	348
23.5	僕の Model.h を公開します .....	350
23.6	僕の Renderer.h を公開します .....	352
23.7	シェーダエラー時にとつ捕まえやすくする.....	358
23.8	const と constexpr について .....	360
24	Effekseer 組み込み.....	364
24.1	とにかくダウンロードして遊ぼう .....	364
24.2	ひとまずサンプルを動かそう .....	365
24.3	『俺の』プロジェクトに組み込んでみよう .....	369
24.3.1	インクルードパスとライブラリパスを解決しよう .....	369
24.3.2	ひとまずコンパイルおよびビルドを通す .....	372
24.4	さあ本格的インテグレーションだ .....	373
24.4.1	初期化処理 .....	374
24.4.2	基本設定 .....	374
24.4.3	カメラとかの設定 .....	375

24.5	特定の場所にエフェクトを出そう.....	378
24.5.1	カメラクラスの改造.....	380
25	アニメーション切り替え.....	390
26	アンチエイリアシング.....	391
27	インバースキネマティクス(IK).....	395
27.1	CCD-IK とは… .....	397
27.2	高校数学だけで IK っぽくしてみよう .....	397
27.3	二次元における CCD-IK .....	400
27.4	三次元における CCD-IK .....	402
27.4.1	LookAt 行列の作成.....	408
27.4.2	納得いかないんですけど… .....	416
27.4.3	夢はひろがりんぐ .....	417
27.4.4	何で納得いってないのか&実験 .....	417
27.4.5	まっすぐに対する解決案.....	418
27.4.6	曲げる軸について考える .....	419
27.4.7	LookAt 行列関数の改造 .....	423
27.5	いよいよ実装である.....	424
27.5.1	PMD から IK 情報を取得する .....	425
27.5.2	CCD_IK 関数について .....	427
27.5.3	CCD-IK の実装 .....	429
27.5.4	呼び出し側 .....	434
27.5.5	角度制限 .....	435
27.5.6	確認コード .....	438
27.5.7	仕上げ(軸の補正).....	441

27.5.8	仕上げ(ベクトル最大長によるIK位置の補正).....	442
28	ビルボード .....	444
28.1	数学的な話 .....	447
28.2	実装 .....	448
28.2.1	カメラを動かせるようにしよう.....	449
28.2.2	カメラを動かせるようになってついでに .....	452
28.2.3	ビルボード.....	453
28.2.4	ビルボードクラス.....	456
29	音関連 .....	462
29.1	CRIADX2LE .....	462
29.1.1	ライセンスについて.....	463
29.1.2	組み込み.....	465
29.2	XAudio2について.....	469
29.2.1	ソースボイスを作成する.....	472
29.2.2	サウンドデータパッファを作成する .....	474
29.2.3	波形データを作る.....	475
30	おすすめの書籍、サイト.....	479
31	最終課題 .....	481
32	ベジエ曲線について .....	482
32.1	センサーも悩むやで .....	485

## はじめに

ついにやってまいりましたこの時がツツツツツ!!

まあ「地獄の」というほどには地獄ではないんですけど。3DCGが分かってない人にとっては地獄…そして死にます。

でもね、これをすべて習得できれば就職活動がグッと有利に進められるという諸刃の剣。素人にはお勧めできない。ここからはマジでキツいし逃げ出したくなる奴も出てくるだろう。だが結局のところかけた労力の分だけが成果に繋がるのだ。



半年後の結果は今までのそして今からの努力との等価交換なんぢ? 前期までである程度思い知つたでしょ?

『頑張ってるのに成果が出てない』等というのは弱者の戯言なのだ。結果が努力を表すのだ。

正直心してついてきて欲しい。そして最後までついてこれた人は自信を持っていい。はつきり言って並の専門学校では教えてないことをこれからやるのだから…。

ただし、それだけに楽ではない。

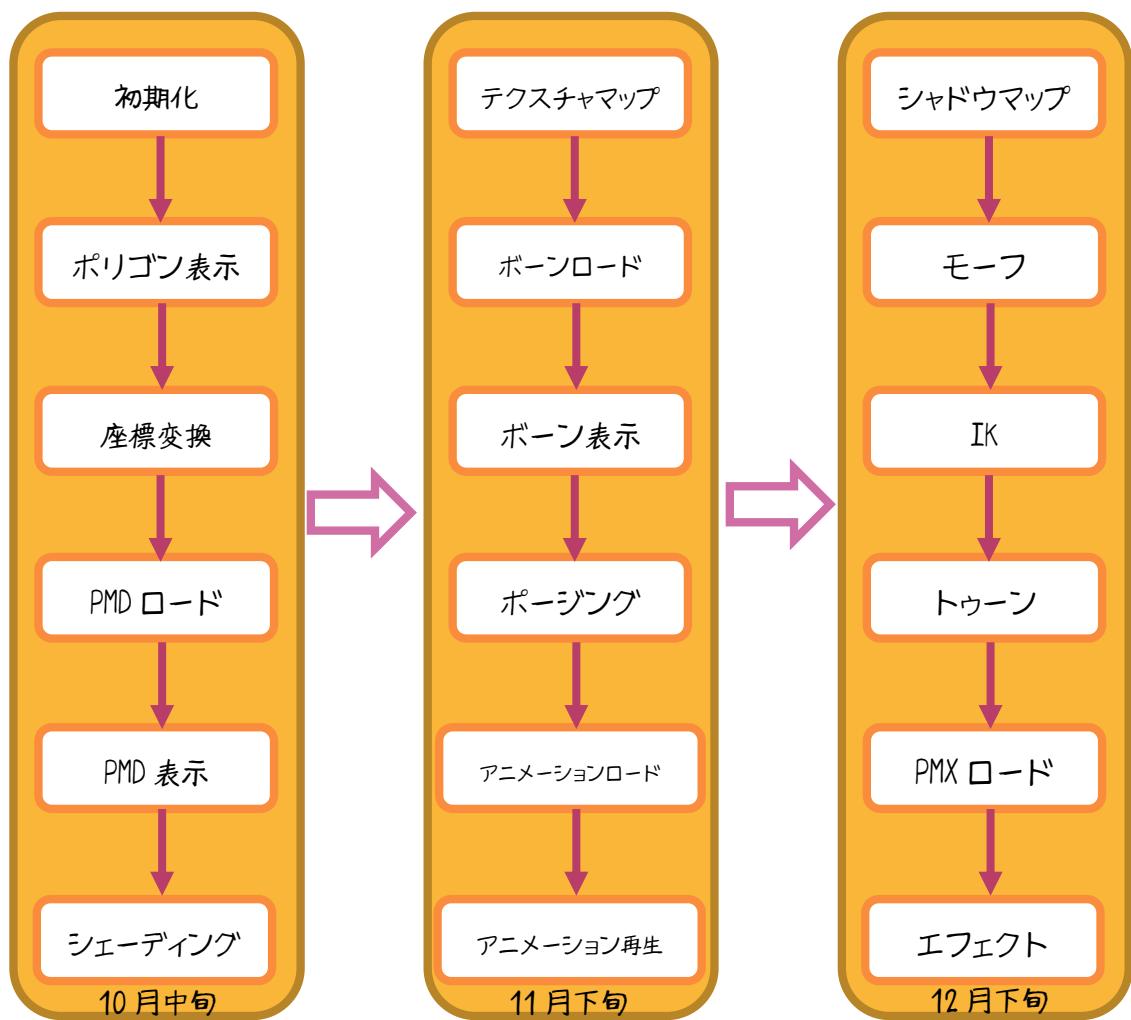
楽ではないのだ…代償もなく得られるものなどないのだ…。これからやって行くことは、DXLib やゲームエンジンに頼らずにゲームを作っていくこと。

つまり画像がどのようにして画面上に表示されているのかを理解する…そこからはじめなければならないのだ。

## 1.1 だいたいの流れ

この授業ではゲームそのものの作り方はほとんど教えないし、DirectX11を理解した後でゲーム開発ってやってると、次年度就職年次の人には確実に間に合わない。なので、ゲーム開発は自主的にやっておくことを強くお勧めします。

DxLibでも何でもいいのでこの授業外でゲームを作っておいて、Direct3Dの理解がそれなりにできてきたら、DxlibのプログラムをDirect3Dに移植するなどするといいでしょう。ちなみに全体の流れは以下のとおり



大きく分けて3つのフェーズに分かれます。

「最低ライン」は一番左の部分です。あ、「最低ライン」ってのは「単位をあげる最低ライン」ね。「就職できる最低ライン」と間違えないようにね。

少なくとも真ん中のフェーズをクリアしないとコンシューマのゲームプログラマは厳しいでしょう。

ちなみに「フロム・ソフトウェア」レベルを目指す人は、一番右までクリアした上で、さらにそこを越えて行きましょう。



まあそうは言ってもそこまでビビる事はない。技術的に足りなくとも悲観する事あ

一番右に至っては数学のスキルとCGの知識が必須(分かってねーと手も足も出ない)なのだが、恐らくそこを十分に教えて言ってる時間はない。なので今のうちからガンガン予習しておこう…脅しでも何でもなく事実です。

## 1.2 注意事項

でも、ものごつい残酷なことを言うと、授業について来れない人がいると授業が遅れます。遅れるということは全員が「学ぶ機会を失う」ということです。理解できますね？

ここは30~40人いるわけです。たまにセンサーを10分も20分も質問で拘束する人がいますが、これは僕を拘束しているだけでなく、全員の学ぶ機会もそれくらい消費しているという自覚を持ちましょう。質問すんなと言ってるわけじゃなくて、その自覚を持ってくれって事。全員がフロムに行ける確率を減らしているんだぜ？

ちなみに難関だろと思われる部分は「シャドウマップ」と「インパースキネマティクス」だ。ここに入る頃には色々と自学自習をしておく必要があるし、授業について来れてない場合は、なんとかついてくる必要がある。

悪いがこれから先は「センサーの教え方が悪いからだ」と喰いている暇はない。別に俺の免罪のために言ってるわけじゃなく、時間がないのだ。君たち自身のために心しておいてくれ。

あ、あとボクは C++er なので、STL 使いまくります。少なくとも std::vector は使用しまくりまくりますので、vector 知らない人は今のうちに周りに聞いて自学自習してください。

ゴメンけどそこは授業でやってる場合ちゃうんやで。

### 1.3 できる奴らは自分で

授業では基本的な事しかやらないので授業以上の事をやりたければ自分でやらざるを得ない。上で予告した以外のことはまあまずやらないだろう。つまりそつから先は自分で獲っていくしかない。

ちなみにそういう情報の仕入先は

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

これはプロのゲームクリエータのテクの資料があるサイトだ。正直プロテクなのでレベル高すぎなのが多いが、この半年でセンサーを超えるならちょうどいいだろう。

またそこまで行かなくても

3D グラフィックススマニアックス: <http://news.mynavi.jp/column/graphics/029/>

ヘキサドライブデモ: <http://hexadrive.jp/lab/demo/>

をひと通り読むと、実際のゲームで使われている基礎技術が分かる。この中で自分が興味あるものを調べて実装していくといい。一つ実装するだけでも随分とレベルアップになるだろう。

ただ、やっぱりどう実装したらしいのか分からぬ場合もある。どんな天才にでもある。そういう場合は細かい解説や実際のコードを見るといい。

ゲームつくろー:[http://marupeke296.com/ProShader\\_main.html](http://marupeke296.com/ProShader_main.html)

t-pot:<http://t-pot.com/program/index.html>

Project Asura:<http://www.project-asura.com/program.html>

MarverickProject:<http://maverickproj.fc2.com/pg00.html>

以下はちょっとおまけだが好きな技術である

ビームマンP:<https://www43.atwiki.jp/beamman/pages/13.html>

ZeroGram:<http://zerogram.info/?cat=4>

あと本気でMMD再生機を実装したいならBulletを勉強しておくといい。

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

<http://zerogram.info/?p=1558>

MMDの中で実際に使用されている物理エンジンだ。物理エンジンにはいくつか種類があるので、興味がある人は調べておくといい。

Bullet,PhysX,Havokなど:

<https://ja.wikipedia.org/wiki/%E7%89%A9%E7%90%86%E6%BC%94%E7%AE%97%E3%82%A8%E3%83%B3%E3%82%B8%E3%83%B3>

ちなみにUnityやUE4などはPhysXを使用しています。

なお、物理エンジンまで行かなくて、フゾーに当たり判定がやりたいためなら

ゲームつくろー(衝突判定):[http://marupeke296.com/COL\\_main.html](http://marupeke296.com/COL_main.html)

を見ておくといいでしょう。

アタマぶつ飛んでる人はUnrealEngine4のソースコードを見ておいてください。流石にそこまで行くと僕も面倒は見きれないです。

あついでにネットワークに関してですが、これも授業でやるつもりはないです。

<http://kidoom.hatenadiary.jp/entry/20110507/1304754628>

<http://testcording.com/?p=1818>

でも見て、基礎的な部分をおさえたら、あとはプログラミングやっていけば覚えるでしょう。

ただ、相当の能力…相当の時間がない限り、就職活動までにここまでやった挙句にネットワーク化するのはおすすめしません。まあまず時間が足りません。既にある程度のものができているなら兎も角、そうでない人は無理です。プロでも無理な時間でしょう。

ネットワークゲームを作りたいなら、DX11はDX11…ゲームエンジン等使ってネットワークゲーム作るのは分けておいたほうがいいでしょう。いや今まで作った人はいるし、不可能じゃないけど、すげえ大変な上に時間足りないよ。ちなみにこれができた奴は…

- フロム・ソフトウェア
- レベルファイブ
- サイバーコネクトツー
- ガンバリオン
- システムソフト・アルファー

に行ってます。まあ…当然といえば当然ですね。まあそういうわけなのでやってやれないわけじゃない。でももう本当にこつから先プログラミングに命を捧げる覚悟じゃないと無理です。

さあ何となくこの授業の雰囲気が分かったところで早速開発に入っていこうか。マンがC/C++の文法が分かってない人は…特にC言語の文法サポートしないからよろしくな…悪いなあ、もう次年度就職年次後期なんだ。んな事やってる暇アねえんだよ!!!

#### 1.4 はじめの最後に…

このクラスだから…信頼はしてるんだけど、それでも毎年出てくるゲス野郎のせいで僕は疑心暗鬼だ。だから言っておく。絶対にしてはならないこと…

### ①進捗に関して嘘をつくな！！

毎年いるんですけど…全然出来てないのに、実行結果もソースコードも見せない…そして期末に提出できない…そんな奴がいる!!いるんだよマジで!!!これ一番アタマ来るんだよ!!!できていなければ出来てないってさっさと言えッ!!!

### ②他人の提出物をコピーして提出 すんな！！

いや～、これいるんですよ。40分の1くらいの確率ですけど…アンタこれ犯罪だよ？カンニング見つかったのと一緒にだよ？全教科0点だよ？いやさ、この学校は甘いから全教科0点になって出来ないこと知ってやってるんだろうけど、この教科6単位だよ？落とした瞬間に留年だよ？お母さんに「そこを何とか」と泣きつかれたこともあるけど…ええ年こいて親に迷惑かけんな!!!!ほんまグーで殴りそうになったわ。

### ③言い訳すんな！提出しろ！！

バイトが忙しいんだろう、色いろあるんだろう…だが、提出すべきものは提出しろ。あと後でも話すけど『USBをなくしました/壊れました』は一切聞かないからな!!!

君らの先輩には上の3つをコンボしたクソ野郎がいたんだけど、ちょっとホンマに殴りそうになった…。「できてます」→「USBなくしました」→「〇〇君のを参考にしたら名前もそのままでした」→「鹿児島の実家に帰ってるんで提出できません」のクソコンボ…そんなクソ野郎相手にする時間はないのだ。

## 2 知識的なこと

知識の話をすると、どうも否定的な反応が帰ってくることが多いのですが、就職活動においても、もちろん開発においても武器になります。重要なのは情報の取捨選択。そして如何に自分のものとして使いこなしていくかだ。



知識は何よりも宝になる。そして重荷にならない生きていくための力だ。

と、ホムンクルスもこう申しております。

以前にも言ったかもしれないけど、今はグーグル先生に聞けば大抵のことは分かる時代だ。

だが「グーグル先生への聞き方」によって、必要とする情報へのアクセス速度が格段に違うのだ。

「用語」一つとっても「やりたいこと」がゲームプログラミングにおいてはどう呼ばれているかを知つていればすぐにアクセスできるのに、その用語を知らないだけで延々と探した挙句に最後までわからないことなんてザラにある。

知識は用語だけじゃない…用語だけを暗記してればいいとか言うのは何処ぞのサコ受験生だけである。受験はそれでいいかも知れないが、就職活動の場合は十分ではない。「体系」を理解する事が重要だ。まあ全体的に網羅しないと「体系」は見えてこないけど、体系が見えてくる

と「理解度が急上昇する」と「なんか用語もスラスラ覚えられる」などという嬉しい副産物の恩恵を受けられる。

だから DirectX11 を勉強する前に言っておく。ちょっとくらい分からんでも「止まるな!!」分からぬものを分からぬままにするのは良くないが、そこで停止してしまうのはもっと良くない。

一旦は流れのままにやってみて区切り区切りで振り返ってみよう。僕の授業も止まるつもりはないので、必ず放課後に遅れば取り戻してくれ。必ずその日のうちに…もしくはその週のうちに授業で進んだところまでは実行できるようにしてくれ。

余談だが「暗記」は「余計な情報をそぎ落とせば削ぎ落とすほどたくさん覚えられる」と思っている人も多いみたいだが、それは間違いだ。コンピュータならそれでいいが人間はそう簡単じゃない。

連想によって記憶力ってのは強化されるんだ。よく言う「芋づる式に思い出す」って言うが、ある事柄を思い出すために既に知っている内容との連携が強ければ強いほど思い出しやすい。記憶ってのは溜めるのが目的じゃなくて、引き出すのが目的なんだから芋のつるは多いほうが寧ろいいのだ。

僕が数学の授業でいちいち証明までやってんのはそういうことだ。意味があってやってんだよ。

さて、話がそれたが「体系」を理解しておけばその周辺の「用語」を思い出すのも思い出しやすくなる。これは学習のヒントだ…覚えておこう。

後、最後は「アンテナ」だ。これは開発というより「就職活動」で重要なことだが、技術者としては技術に対するアンテナ(つまり興味の方向性)が重要となる。

今は理解できなくても、自分が将来やりたい仕事に伴うものへの興味は向けておいたほうがいい。具体的には…CG表現に興味があるならそういう本を読むとか買うとかサイトを見るとかする。

ネットワークやらサーバ系に興味があるなら、実験的にプログラムを作る(簡単なメッセージチャーソフトでも何でもいい)とか、サーバを立ててみる(PCを一台潰さなくてもVirtualBoxでubuntuをインストールして、データベースサーバ立てたりしてもいい)

AI(人工知能)について興味があるなら、自分が実装できる範囲のAIを作る。また、FFで使用されていたような先進的なAIの実装調べてみる。

とにかくゲームを作りたい人は、ハイペースでゲームを作りながらも、売れているゲームがなぜ面白いのか、ここはどう実装しているのか考えて徹底的にUXやら技術やら調べたり、いろんな制作イベント(勉強会とかハ耐とか)に出てみよう。

とかそういう奴。そういうアンテナが張れてないと「お前ホントにその仕事したいの?」と疑われる。そこは就職で結構重要な部分なので興味の方向を自覚し、そこに対しては徹底的に闇雲に知識を吸収しよう。

## 2.1 用語

- デザインパターン
- 関数型プログラミング
- スマートポインタ
- STL
- 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/ビルド
- コンパイル/リンク/ビルド
- DCCツール
- ディフューズ/アンビエント/スペキュラー

- ランバート反射
- 頂点シェーダ
- ピクセルシェーダ
- フラグメントシェーダ
- ジオメトリシェーダ
- ハルシェーダ
- コンピュートシェーダ
- HLSL/GLSL
- 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
- Z/ワッファ/深度/ワッファ
- ステンシル/ワッファ
- サンプラー
- GPU
- レンダリングパイプライン
- トーンレンダリング
- モーションブラー
- ユーザビリティ
- ユーザーエクスペリエンス
- 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
- 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
- ディスプレースメントマッピング
- サブサーフェススキャッタリング
- 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
- カスケードシャドウマップ
- PBR(物理ベースレンダリング)
- アルベド/ラフネス/メタリック
- HDR
- 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)
- GeForce GTX 100
- Radeon 100
- 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
- $\alpha\beta$  戻り
- HSB/HSV
- RGB/sRGB/adobeRGB/CMYK

とかまあ、そういう感じ。これ以外にも知っておくべきことは多いっていうか、もう把握できないレベルまで来ている。まずはこれをグーグル先生で調べてみましょう。僕はここでは教えませんよ？自分で Google で調べるのもまた勉強です。ここで「バカバカしい…どうせ単位には関係ないんでしょう」と思って実際に手を動かして調べない人は、それは学校の単位が取れてもゲームプログラマにはなれないパターンです。

とにかくプログラマなんだから、プログラムに関する事は過剰なくらい調べておきましょう。ゲームのプログラムは CG に関する部分が多いので今の君達には理解できないう部分も多いけどとにかく調べよう。

これをイヤイヤ調べてるようじゃダメ。覚えようとしたり理解しようと無理するのも良くない。とにかくワクワクしながら調べて眺めて「ほ~ん…」くらいに思ってください。

## 2.2 体系

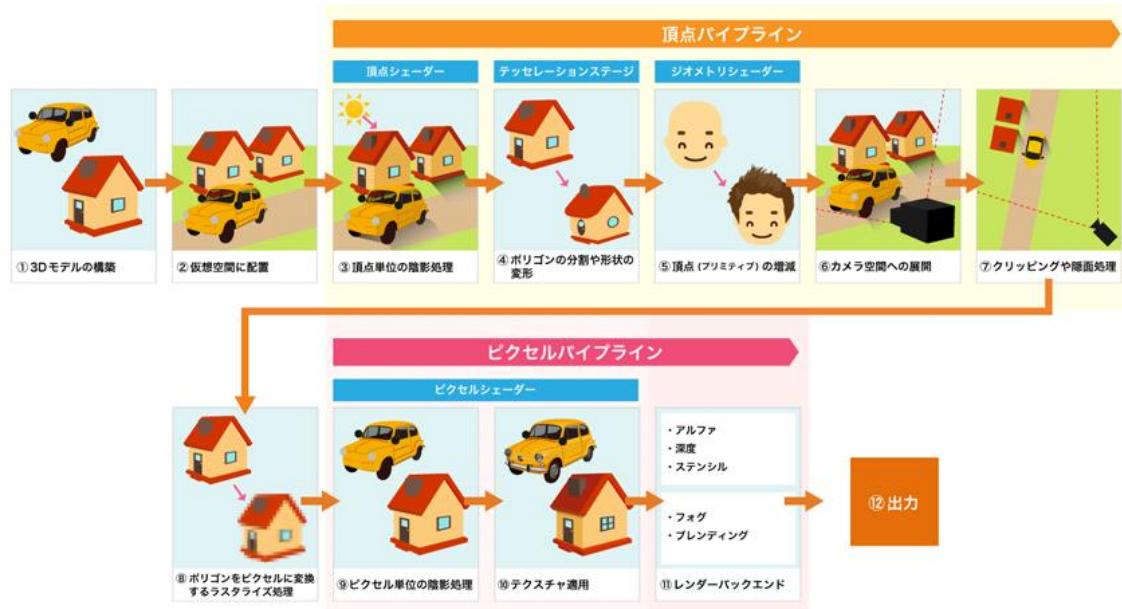
『体系』ごと勉強するのは結構大変なんだけど、今自分が勉強していることの全体像を掴んでいることは重要な事だ。今、自分が勉強しているのは全体のどの辺なのか?そういう事を考えながら勉強していくとより効果的だ。

例えば C++なら Wikipedia をざーっと見るんや

<https://ja.wikipedia.org/wiki/C%2B%2B>

んで C++の機能にどんなもんがあるのか大体把握しつく。なお、このウィキペは相当簡略化して書いているので、全体像をつかむには逆にちょうどいいきなりデカイものを見ると絶対挫折するからな。

こう考えると色々なものに体系がある。例えば画面上に CG を表示する過程を全体的に把握するにはこの図を見るといい。



かなりわかりやすい図だと思う。

『レンダリングパイプライン』で検索するとこういう図がいくつてくる。例えばこの図において今自分が頂点シェーダーをいじっているのであれば自分が『頂点パイプライン』の最初の方にいるんだなって事がわかる。

これを分かってないと、自分が今何をしているのかすぐに分からなくなる。気をつけろ!!

出来る限り図で説明されているものがいいが、そもそも行かないもので、なんとか頑張って把握して欲しい。

とにかく知りたいことがあつたらその全体的なことをとにかく早く把握する。これが大事です。

## 2.3 アンテナ

正直どこにアンテナを張るのかは、人によって違いますので、それぞれが決めて欲しいが、自分のやりたい仕事に繋がるようなアンテナを張っていてくれ。

ともかくゲームプログラミングに関するアンテナを張りたければ CEDEC の講演を聞くなり、資料を漁るなりしてみよう。

ホントに沢山の資料がありますので、暇な時に見るといいでしょう。

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

特に今年(2016)は資料の公開が異常に早かったので良かったです。業界がどういう技術を使っているのか、どういう技術者を欲しているのかがわかるでしょう。

ただ、あれもこれもなんてやってるとまず読みなくなりますので、CGなら CG 系を見る。ネットワークならネットワークを見る方がいいと思います。

あくまでも「アンテナ」なので、無理して理解することはないですが、本当に好きな方向を見つけたら多分、実装せずにはおれないはずです。

サンプル実行して、なかのソースコードを見るくらいはするでしょう。それしない人は多分そっち方向はそんなに好きじゃないんでしょう。

ちなみにスマホゲーム方面に進みたい人は Vulkan にもアンテナを貼っておきましょう。家が Windows10 の人は DirectX12 をためしてみるのもいいでしょう。

学校の授業でやってる事は、まあ悪いが全体教育なので、良くも悪くも平均的なところまでしか教えられない。学校の授業をこなしてると安心してちゃダメだ。その先を行け。

**自分の望み通りの会社に行きたきや授業で  
やった以上のことをやれ！センセーを超  
えてみせろ！！君たちにはそれができるは  
ず。**

### 3 制作準備

#### 3.1 Gitについて

Gitってのはバージョン管理システムという仕組みを使って、ファイルの履歴を保持、更新管理などを行うもので、以下のメリットがあります。

- 「変更したらなんか引いた」時に、原因を特定しやすくなるぞ!!
- ボタン一発で昔の状況に戻ることができるぞ!!
- チーム開発の時に「責任の所在」を明らかにできるぞ!!
- 「差分」を保存していくスタイルなのでハードディスク容量を節約できるぞ!!

「心配症だなあ…自分が作ったプログラムなんだから全て正確に把握しているよ!!」なんて思う人は認識甘いですよー。

#### 一ヶ月前の自分のコードは他人のコード

と思ってください。おじさんは記憶力がアレなので一週間前ですら把握できません。

まあ大した手間がかかるわけでもないのでぜひ使いましょう。

##### 3.1.1 基本的な使い方(ローカル)

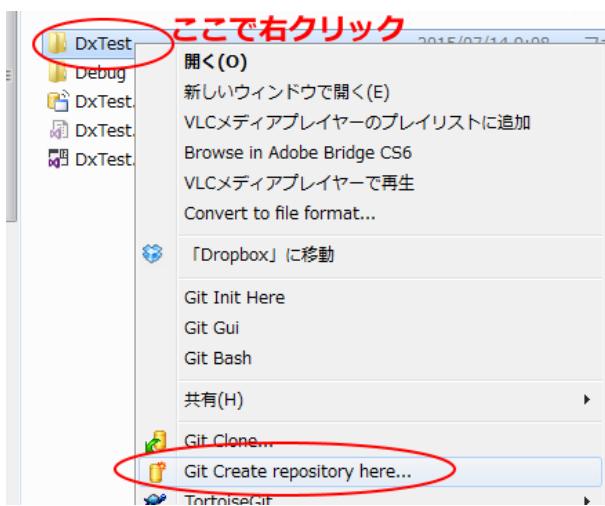
ローカルでだけ使用するのならば超カンタンです。専門的なことはともかくまずは使ってみましょう。

##### とにかく使ってみよう

前に僕のクラスにいたひとは知ってると思うので、とりあえずおさらいと思ってやっていきましょう。

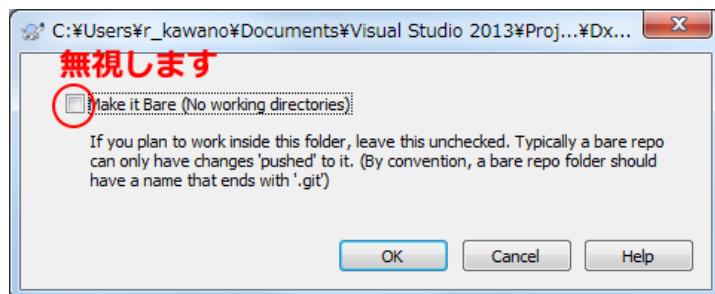
まず、みなさんのDropBoxの中にフォルダを作って…まあここではDx11Gitという名前にでもしておきましょう。このフォルダで右クリック

「リポジトリを作成」

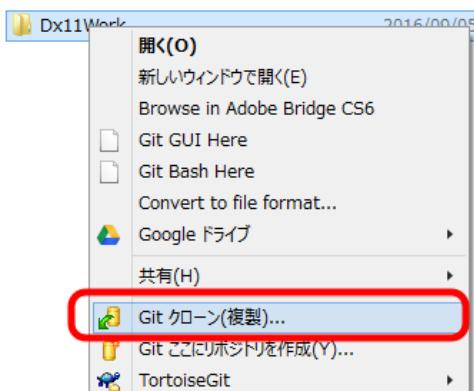


とします。

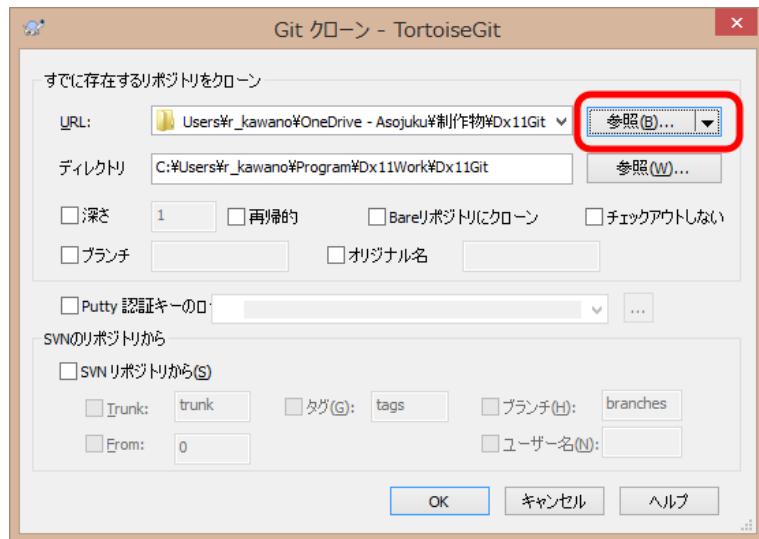
Bare がどうこう言われることがありますか無視します(チェックしません)。そうするとフォルダの中に「隠しファイル」の.git フォルダができていると思います。



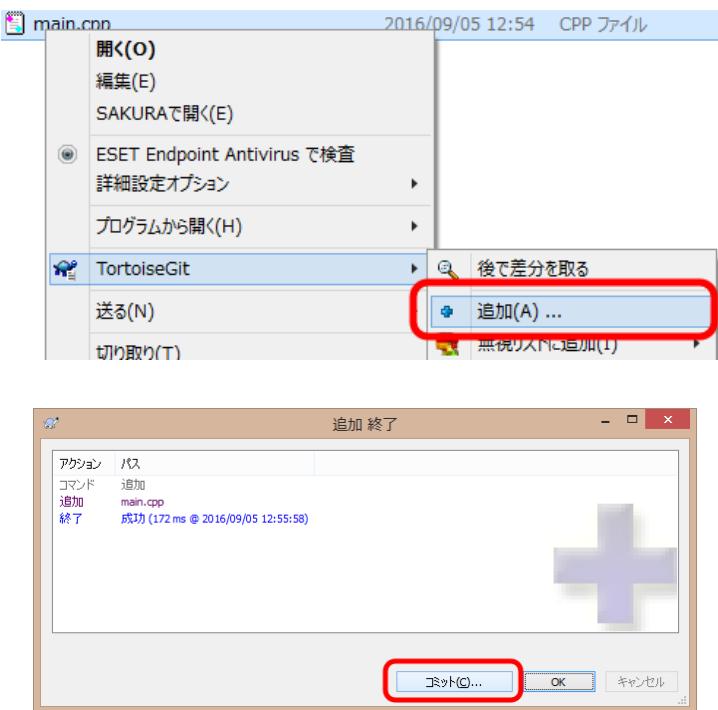
これで準備ができました。ここはあくまでもリポジトリフォルダなのでワーキングフォルダを自分のドキュメントフォルダかデスクトップフォルダに作ります。名前は Dx11Work とかでいいでしょう。そのフォルダで右クリック「Git クローン」してください。



Git クローンの元は最初に自分が作ったリポジトリです。「参照」とかで自分のさっき作ったフォルダを選択してください。



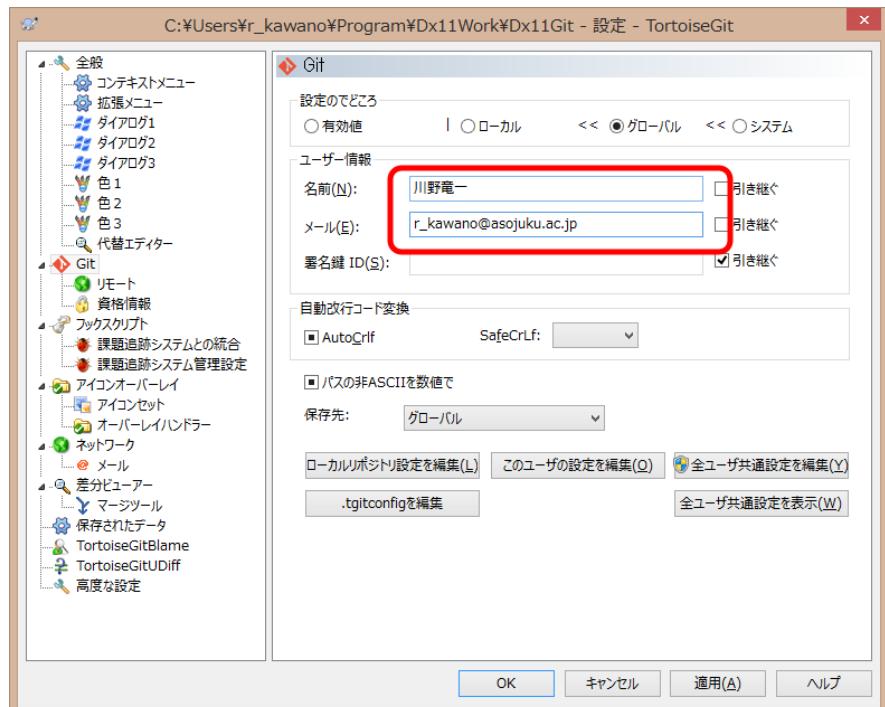
で、例えば DX11Git フォルダの中に main.cpp とか置いて「追加」して、「コミット」します。



あ、この時に Git の設定をきちんとやってないと怒られます(大したことじゃないです)。設定します。



はいを押したら、次の画面が出てくるので、名前とメールアドレスを設定します。これだけでOK



そうすると「コミットのコメント」を求められます。これは今回追加したのはどういう意味があるのか?ということを書いてください。意味が無いなら『main.cpp を追加』くらいで構いません。今後は変更点をコメントとして分かりやすく残していきます。

これで履歴ができました。

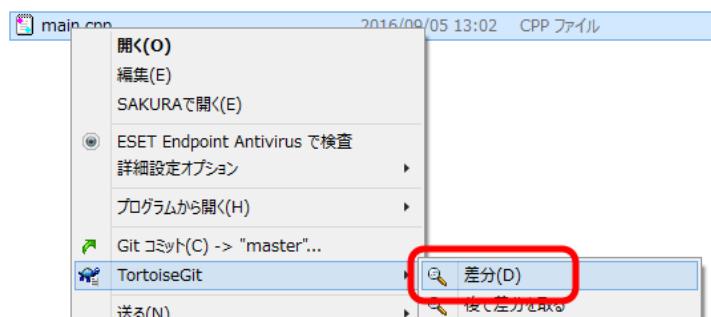
次にどこか変更してみてください。そして右クリックして Git のメニューを見ると、差分ってのがあると思います。

その場合どの行が変更されたのかとかの情報を見れます。

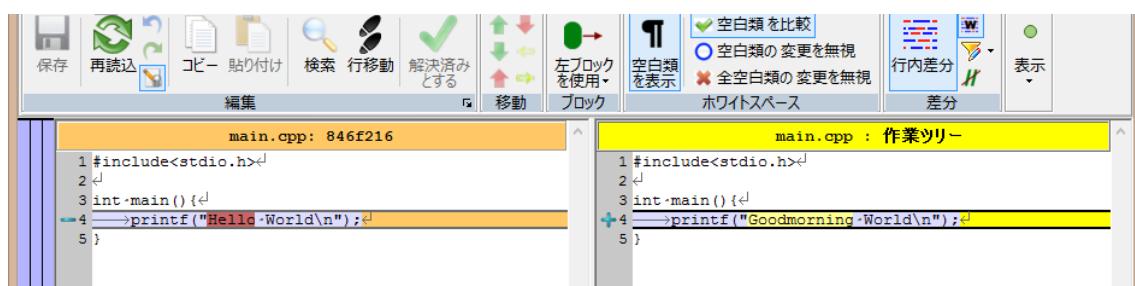
例えばこんなふうに「変更した箇所」を教えてくれるわけだ。

自分で開発していてもある時動かなくなつて、昨日までは動いていたのに…あれ？俺どこ変更したっけ？って思うことってあるだろう？

そんな時は「差分」を見るんだ



例えば HelloWorld の cpp をコミットしたとする。そしてその後に GoodmorningWorld なんて変更した上で「差分」を見ると、変更した行がハイライトされ、その中の変更箇所が分かるようになっている。



「カバカしい」と思っているそこの君!!!自分がどこを変更したのか分からなくなつて泣きていてこないようね!!!ここまで言っても泣きつく人なんて相手にしてあげられないんだからねっ!!!

ここで紹介したのは Git のあくまでも「ローカル」の機能のみですので、サーバーの機能については、今後チーム制作とかで考えることになるでしょう。

また、インターンシップに行つたら、大抵の場合は Git(もしくは subversion)を使わされるので、その時に深く覚えればいいでしょう。

### 3.1.2 基本的な使い方(サーバー)

自分でサーバ立てて Git を活用するのは結構大変なので、チームでモノを作るようになってからにしましょう。

まずは練習として、特定のサーバ(参考になりそうなプログラム)からのクローンを作ってソースコードリーディングに使いましょう。

<https://www.unrealengine.com/ja/ue4-on-github>

↑ UnrealEngine4 のソースコードを見れるので見ておこう。もちろんアカウント作ってないと見れないぞ!!

#### GitHub を使ってみよう

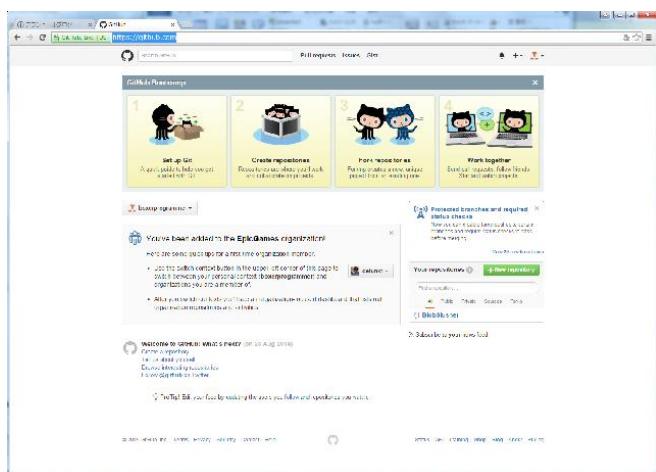
とりあえず GitHub っていうサイトがあるんだけど、そこは通常「リポジトリ」(チームで作るのに必要な Git のサーバ置き場)を作るのにお金がかかる。

だけど、「オープンソース」ライセンスならばリポジトリを作るのにお金がかからないというなかなかイカしたサイトなのだ。

とりあえず君がプログラマの端くれなれば GitHub にアカウントを作っておこう。

<https://github.com/>

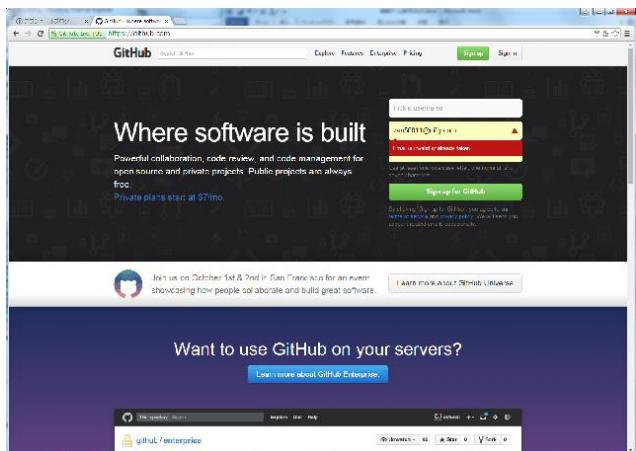
に行けば、



こういう可愛らしいサイトがあるとおもいます。

あ…

もしかしたらアカウントをやってないから…



こんな感じになっていたりするかもね。

そうなついたら「サインアップ」(新しくアカウントを作ること)をやってみよう。

右側のメールアドレスに自分のメアドを入力して「Signup」だ。

色々と書かなきゃいけないことがあると思うけど、そこは艦これとか刀剣乱舞とかやりこんでいるみんなだ!! こういうものの登録方法が分からぬいわけはないよね。

さあ、サインアップし給え。

さて、そうやってサインアップして、自分の情報を入れたら準備完了だ。

どつかテキトーにフォルダを作って、そこに特定のリポジトリのクローンを作つてみましょう。  
うへん、ひとまず UnrealEngine4 のソースコードでも落としてこようか…?

UnrealEngine のアカウントは作つてあるよね?

作つてない?

じゃあそつちも急いで作っちゃおう。

作つたら

<https://github.com/EpicGames/UnrealEngine>

からのクローンを作ってみよう。

どうだい?

このソースコードは宝の山だ。分かるところだけでも良いからぜひ見ておくと良い。

ど～～～～してもクローンが作れないけどソースコードは見たいとかそういう人は  
DownloadZIPってメニューもあるからそっちからダウンロードしてくれ。

### *Backlog を使ってみよう*

これはローカルが使いこなせたら是非試してみて欲しい。

<http://www.backlog.jp/license/?ref=main>

にアクセスすると「…あれ?どれも有料プランじゃん!!」と思うかもしれないが右下の方にち  
っちゃく「フリープラン」が記載されてある。

いずれのプランも30日間無料でお試しいただけます。

ベーシックプラン ¥2,000/月	プレミアムプラン ¥9,800/月	おすすめ マックスプラン ¥16,800/月	プラチナプラン ¥50,000/月	オンプレミス型 エンタープライズ ¥49,500/年~
ストレージ / 1GB	ストレージ / 30GB	ストレージ / 100GB	ストレージ / 300GB	ストレージ / 無制限
プロジェクト数 / 5	プロジェクト数 / 100	プロジェクト数 / 無制限	プロジェクト数 / 無制限	プロジェクト数 / 無制限
ユーザ数 / 30	ユーザ数 / 無制限	ユーザ数 / 無制限	ユーザ数 / 無制限	ユーザ数 / 20人~

30日間無料お試し      30日間無料お試し      30日間無料お試し      30日間無料お試し      詳しくはこちら

※ 価格は税抜価格です  
1プロジェクト、10人までのフリープランはこちら

全ての機能の違いを見る ▾

まあ…あまり使ってほしくなさそうなのは見て取れるがそこは…使おう。

まずはユーザを作つてプロジェクトの追加をしよう。ただしこのプロジェクト  
は1つまでなので適切なものを選ぼう。

ちなみに後で説明するが Backlog 以外にもこういうサービスはあるのでそちらを使おう(ただし殆どが英語だがな)

まずは慣れるために Backlog を紹介したってわけだ。

プロジェクトの作り方は…分かるだろう?

作ったらプロジェクト→プロジェクト名→Git と進んでくれ。そうすると

### ⑤ アクションゲーム (KITI\_VIL)

The screenshot shows the Backlog application interface. At the top, there's a navigation bar with tabs: ホーム, 課題の検索, 課題の追加, Wiki, ファイル, Subversion, Git (which is highlighted in green), and a placeholder for another tab. A yellow tooltip message 'SSH 公開鍵の登録' (SSH Public Key Registration) appears, stating 'Backlog の Git レポジトリに SSH でアクセスする場合は、公開鍵の登録が必要です。' (If you access the Git repository on Backlog via SSH, you need to register the public key). Below this, the main area is titled 'リポジトリ一覧' (Repository List). It shows a list of repositories, with 'splatter' selected. The repository details for 'splatter' include its last updated time (最終更新日時 : 2016/03/28 15:33) and a commit history section. Underneath, it says 'subversionのものをgitに移行' (Migrate from subversion to git). The 'HTTP' URL is listed as [https://tsuchinoko.backlog.jp/git/KITI\\_VIL/splatter.git](https://tsuchinoko.backlog.jp/git/KITI_VIL/splatter.git), and the 'SSH' URL is listed as [tsuchinoko@tsuchinoko.git.backlog.jp:/KITI\\_VIL/splatter.git](tsuchinoko@tsuchinoko.git.backlog.jp:/KITI_VIL/splatter.git). To the right of the repository list, there's a sidebar titled '最近の更新' (Recent Updates) which lists several recent commits made by 'tsuchinoko'.

こんなのでてくると思うので、HTTP の部分をコピーしておいてくれ。あとは Git クローンを作る時のリポジトリをこのアドレスにする。

それでクローンを作るとユーザー名とパスワードを求められるので入力。コミット→プッシュするたびにユーザー名&パスワードを求められるので忘れないようにね。

Git やるとプッシュだのプルだのいう用語がよく出てくるが、最終的にプッシュしないとサーバーには反映されないってのは覚えておいてくれ。忘れるとチーム制作は死ぬるぞ？

サーバーを介する大雑把に言うとこんな感じだ。まあ細かいところでマージだのロールバックだの何だの出てくるがやつてりや覚えるので必要になった時に覚えて欲しいんだが

- ロールバック:昔のバージョンに戻すこと
- リバート:今の作業を取り消すこと
- マージ:自分の変更分と他人の変更分をいい感じに合成すること

この辺は実際にチーム制作やらないと直面しないので…まあ直面してからでいいでしょう。

とにかく今は↑のような操作があって、それはチーム制作の時に役に立つって事をアタマに入れておきましょう。

フリーのGitリポジトリサーバーを紹介しておきます(ただし英語)

- GitHub(無料だが完全に公開状態。上げただけで即オープンソース化!!)
- BitBucket(無料で非公開にできる…が5名にしかコミット権がない)
- Assembla(1個だけプライベートリポジトリが作成可能…3名までの制限)

まあ一人でやってるうちは英語以外はあまり気にする必要はないです。数人で開発するときにはGitHubかBitBucketでしょうかね。別に学生作品はお金が絡んでいるわけでもないので、GitHubでいいんじゃないでしょうか?

慣れてきたら自分で選んでもいいし、自分でサーバを立ててGitLabを作つてもいいと思います。

## 4 ウィンドウの表示

ゲームエンジンもフレームワークも DxLib 等をも使用せずにウィンドウを表示しようとするとどれくらい面倒なのが体験していただきたいと思います。

ちなみにまずはウィンドウズアプリとしてプロジェクト作って、WinMainで動かすところまでやっていきますが実はコンソールとして作ってもウィンドウを出すことができます。こういうのって意外と本には書いてないよね…。

DxLib で作った時も WinMain は作ったと思います。そこは DxLib でも DirectX11 でも変わりませんので、まずは WinMain を作ってウィンドウ表示の準備をしましょう。

プロジェクトの作り方は大丈夫ですね？流石にそこは前期まででおしまいですよ？前期だけで2～3回やってるはずですからね。

できない人は周りの友達に教えてもらってください。

あと前期と違って前でコーディングなんてしてあげないんだから。

とはいえる、最初のところだけ難しいので書く

```
#include<Windows.h>
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int ){
    return 0;
}
```

とりあえずここでコンパイルだけ通します。通りましたか？通らない人はプログラムかプロジェクトの設定かのどちらかが間違っています。

前期の授業を思い出しながら(前期のテキスト見ながら)なんとかしてください。悪いが後期は時間がないのだ。そんな所に関わってられない。

さて、それでは WinMain にこれを打ち込んでくれ

```
//ウィンドウクラスの作成と登録
WNDCLASS w=WNDCLASS();
w.lpfWndProc=(WNDPROC)WindowProcedure;
w.lpszClassName="ClassName";//クラス名
w.hInstance=GetModuleHandle(0);
RegisterClass(&w);//ウィンドウクラス登録

//ウィンドウサイズ設定
RECT wrc={0,0,640,480};
::AdjustWindowRect(&wrc,WS_OVERLAPPEDWINDOW,false);

//ウィンドウの生成
HWND hwnd>CreateWindow(w.lpszClassName,
    "ウィンドウタイトル",//ウィンドウタイトル
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    wrc.right-wrc.left,
    wrc.bottom-wrc.top,
    NULL,
    NULL,
    w.hInstance,
    NULL);

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

解説は後です。

で、おそらくは w.lpfWndProc でエラーが発生するのでウィンドウプロシージャを作つておく。ウィンドウプロシージャってのはウィンドウに対してなんかしらメッセージが飛んで来たら実行される関数のことだ。この関数ポインタを渡すのだが…

## 4.1 関数ポインタとコールバック

関数ポインタってのは知ってるかい? ていうか僕も前期でさんざん教えたので何となくは分かっていることと思うけど、通常の変数にポインタがあるように、関数にもポインタがある。

それを渡してるわけだ。

何故かというと、所謂「コールバックシステム」を実現するためです。コールバックシステムってのは、ウィンドウズからの応答があった際に、その関数をウィンドウズがコールできるようにするものです。

なんでそんな事をするのかというと、OSってのは君たちが思っているよりもいろいろなことをしている。キーボードやマウスの入力検知などもそうなのだ。要はハードウェアの入力、ハードウェアへの出力はOSを介して行われている。ここ結構大事なので覚えといてね。

ちなみにキーボード入力もウィンドウの×ボタンを押すのも全部OSを介している。例えば「ユーザーがウィンドウを終わらす行為をした」とてのはWM\_DESTROYなんてメッセージが送られてくるが、どうやって送るのかというと、そのコールバック関数を呼ぶことで送っている。

ほんまはきっちり理解するためには自分でコールバックシステムを作ったほうがええんやが…そんな時間はないのでちゃっちゃいくで?とりあえずこういう関数を作るんやで

```
//ウィンドウプロシージャ(あとで説明します)
LRESULT CALLBACK WindowProcedure(HWND hwnd,
    UINT msg,
    WPARAM wpal,
    LPARAM lpal)
{
    if(msg==WM_DESTROY){
        PostQuitMessage(0);
        return 0;
    }
    return DefWindowProc(hwnd,
        msg,
        wpal,
        lpal);
}
```

これがさっき言うとったウィンドウプロシージャやな。

うん…まあ今回の授業ではWM\_DESTROYくらいしか使わへんのやけど…まあそういうシステムのために定義する必要があるくらいに思っておいてや(ぶっちゃけキー入力はウィンドウプロシージャで受け取るのは不適切やしな)。

ここまで書けたら取りあえずはウインドウ出るやろ?

でも一瞬で消えるやん?どうすればいい?DxLibの時にも似たのがあったな?

そうだね、無限ループだね。で、そのままやると処理が固まるので、  
ProcessMessage 命令を入れていたと思います。

で、ProcessMessage は思いのほか沢山の事をやってます。嘘やと思うんなら  
DxLib のソースコード見てみ?くっそ長いで…。

ともかく固まる事のないようにしなければならない。何故固まってしまうのだろうか?

それはアプリケーションのループは処理を専有し、OS へ処理を戻さない(正確に言つて OS からの処理に反応しない)ため、「固まつた」ようになつてしまうわけです。そうなればキーボードやマウスも効かなくなる。Xボタンを押しても落ちないというわけ。これがフリーズやね。

…こうならないために、

PeekMessage を使用します。

というわけで、固まらないために今回使用するのは

- PeekMessage
- TranslateMessage
- DispatchMessage

の3つです。

それぞれリファレンスを見てください。現場に行って一番先輩から怒られる言葉ナンバー1は「マニュアル読め!!」です。

隅から隅まで読む気持ちで読んでください。そうは言ってもよくわからないで  
しょうから今はひとまず眺めてください。マニュアルを確認するクセが大事です。

PeekMessage

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

TranslateMessage

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

DispatchMessage

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

まず PeekMessage ですが

「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。」

と書いてます。どういう意味でしょうか。MSDN は読んでも分からないことが多いです。こういう時はいろんなサイトを見ながら、意味を推測します。どうしてもわからんねーときは、英語の方を見てみるのも一つの解です。

結構誤訳があるんですね～。ちなみに Dispatch が「送出」と訳されていますがこれは適切なのでしょうか？英語のまま専門用語として考えてみます。

<http://www.weblio.jp/content/%E3%83%87%E3%82%A3%E3%82%B9%E3%83%91%E3%83%83%E3%83%81>

OS が処理の内容や用途に応じて動作の割当を行う。これが適切っぽい。

そもそも着信とか送信とか意味がわからんのよねえ…。

ともかく、OSってのはこの場合、ウインドウに教えるべきメッセージをどんどん「キュー」に溜めて行ってるわけ。PeekMessageってのはそれを見に行ってなんか溜まってるメッセージあつたら指定した構造体に格納するわけだ。

ちなみに、よく聞かれるんだけど PeekMessage と GetMessage の違いってなんなの?ってのがあります。GetMessage はメッセージが来るまで待機します。PeekMessage はメッセージが来てるかどうかチェックして、あってもなくても戻ります。

ゲームにおいて「待機」ってのは具合が悪い。何も入力がなくても動くべきでしょ?だから GetMessage は不適切なわけ。あとよくインターネットでこういう実装を見かけますが…

```
if (PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE)) {  
    if (!GetMessage (&msg, NULL, 0, 0)) /* メッセージ処理 */  
        return msg.wParam;  
  
    TranslateMessage(&msg);  
  
    DispatchMessage(&msg);  
}
```

全くの無駄やで…。これを書くんやつたら

```
if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE)) {  
    if (msg.message == WM_QUIT) {  
        break;  
    }  
}
```

```
TranslateMessage(&msg);  
DispatchMessage(&msg);  
}
```

こうすべきやで。

意味がわからないかもしないが、さっきも言ったように PeekMessage と GetMessage は同じ意味で、2つを同時に呼び出すのは意味が無い。この実装してるのは恐らく PeekMessage の最後の引数を NOREMOVE にしてしまい、メッセージが残ったから、GetMessage して辻褷合わせようとしてるだけやで。

あと、DirectX のサンプルでもなんとか GetMessage を使っているので、それに引きずられているだけだと思う。何故サンプルがそうなっているかは正直不明だなあ…僕の解釈がちやうんやろか。

ともかく真似したらアカンコードもネットには沢山ある。気をつけよう。

ちなみに TranslateMessage は仮想キーコードメッセージで飛んできたのを文字メッセージに変換するものやで。

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

最後に DispatchMessage なんやけど、これは

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

ウインドウプロシージャにメッセージをディスパッチするらしい…

ん?

え?どういうことなの…。直接ウインドウプロシージャ呼ぶのと何が違うの?と思われるが、これは一旦OSを介してから呼ぶ。だから呼ばれるタイミングが違う。

DispatchMessageをコールすると即時ウインドウプロシージャが実行されるわけではなく、状況によっては遅れることになる。そういうもんだと思ってください。

結果として、こう書いとけばいい

```
while(true){  
    if(PeekMessage(&msg,NULL,0,0,PM_REMOVE)){  
        ::TranslateMessage(&msg);  
        ::DispatchMessage(&msg);  
    }  
    if(msg.message==WM_QUIT){  
        break;  
    }  
}
```

ちなみにWM\_QUITはウインドウのXボタンが押されたり、Alt+F4が押されたりした時に発生します。break;してるんでwhileを抜けてMain関数を抜けるってわけです。

## 5 DirectX11 初期化

↓にDirectX11の初期化関数を書いておく。書いておくがこれをそのままエラーが発生するだけなので自分できちんと考えて、コンパイル通るようにまでしてください。

```
//DirectX11初期化関数  
HRESULT InitDirect3D()  
{  
    //デバイスとスワップチェーンの作成  
    DXGI_SWAP_CHAIN_DESC sd={};//構造体初期化  
    //ここからスワップチェインに対する指定をひたすら書いていく  
    sd.BufferCount = 1;//バックバッファの数=1
```

```
sd.BufferDesc.Width=WINDOW_WIDTH;//バックバッファ幅  
sd.BufferDesc.Height=WINDOW_HEIGHT;//バックバッファ高  
sd.BufferDesc.Format=DXGI_FORMAT_R8G8B8A8_UNORM;//バックバッファのフォーマット
```

#### ARGB

```
sd.BufferDesc.RefreshRate.Numerator=60;//分子  
sd.BufferDesc.RefreshRate.Denominator=1;//分母  
sd.BufferUsage=DXGI_USAGE_RENDER_TARGET_OUTPUT;//このサーフェスはレンダーターゲットとして使用する  
sd.OutputWindow=g_hWnd;//ウィンドウハンドルを指定  
sd.SampleDesc.Count=1;//サンプリング数  
sd.Windowed=TRUE;//ウィンドウモード
```

```
D3D_FEATURE_LEVEL pFeatureLevels = D3D_FEATURE_LEVEL_11_0;//使用したいフィーチャレベル(DirectX11モード)
```

```
D3D_FEATURE_LEVEL* pFeatureLevel = NULL;//実際に使用可能なフィーチャレベル配列へのポインタ
```

```
HRESULT result = D3D11CreateDeviceAndSwapChain(NULL, //規定のアダプタを使用  
D3D_DRIVER_TYPE_HARDWARE, //ハードウェアアクセラレータを使用  
NULL, //ソフトウェアラスタライザ(いらん)  
0, //0でいい(特に指定しない)  
&pFeatureLevels//使いたいフィーチャレベルへのアドレス  
, 1//要求レベルはいくつあるの?=1(フィーチャレベルの数です)  
, D3D11_SDK_VERSION, //ここは固定  
&sd, //スワップチェイン指定  
&g_pSwapChain, //得られるスワップチェイン  
&g_pDevice, //得られるデバイスピント  
pFeatureLevel, //実際に使用できるフィーチャレベル配列へのポインタ  
&g_pDeviceContext); //得られるデバイスコンテキスト
```

```
//バックバッファーのレンダーターゲットビュー(RTV)を作成  
ID3D11Texture2D *pBack;  
//バックバッファのサーフェイスをシェーダリソース(テクスチャ)として抜き出す  
g_pSwapChain->GetBuffer(0, __uuidof( ID3D11Texture2D ), (LPVOID*)&pBack);  
//そのテクスチャをレンダーターゲットとするようなレンダーターゲットビューを作成  
g_pDevice->CreateRenderTargetView( pBack, NULL, &g_pRTV );
```

```

pBack->Release();

//デプスステンシルビュー(DSV)を作成
//デプスステンシルビューってのは所謂「Zバッファ」を有効にするために必要
//Zバッファってのは描画順によらず前後のオブジェクトを正確に描画するのに必要
D3D11_TEXTURE2D_DESC descDepth;
descDepth.Width = WINDOW_WIDTH;//画面幅
descDepth.Height = WINDOW_HEIGHT;//画面高
descDepth.MipLevels = 1;//ミップマップとかいらないので1でいい
descDepth.ArraySize = 1;//テクスチャの数
descDepth.Format = DXGI_FORMAT_D32_FLOAT;//深度バッファのビット数と型の指定(32ビットFloat型)
descDepth.SampleDesc.Count = 1;//1でいいよ
descDepth.SampleDesc.Quality = 0;//今はアンチエリしないのでいらない。
descDepth.Usage = D3D11_USAGE_DEFAULT;//ここはこれ固定で
descDepth.BindFlags = D3D11_BIND_DEPTH_STENCIL;//このテクスチャは「深度値」保存用に使うよと明記
descDepth.CPUAccessFlags = 0;//とりあえず0でいい
descDepth.MiscFlags = 0;//とりあえず0でいい
g_pDevice->CreateTexture2D( &descDepth, NULL, &g_pDS );//深度値用テクスチャの生成
g_pDevice->CreateDepthStencilView(g_pDS, NULL, &g_pDSV );//作ったテクスチャ元にデプスステンシルビューを生成

//ビューポートの設定(これがないとモノを表示した時に表示されません)
D3D11_VIEWPORT vp;
vp.Width = WINDOW_WIDTH;//表示画面の幅(ウィンドウサイズとは違う)
vp.Height = WINDOW_HEIGHT;//表示画面の高さ
vp.MinDepth = 0.0f;//最小深度値(最小Z値)
vp.MaxDepth = 1.0f;//最大深度値(最大Z値)
vp.TopLeftX = 0;//表示部分左上の左
vp.TopLeftY = 0;//表示部分左上の上
g_pDeviceContext->RSSetViewports(1, &vp);

//レンダーターゲットビューとデプスステンシルビューをセット
g_pDeviceContext->OMSetRenderTargets(1, &g_pRTV, g_pDSV );

```

```
    return S_OK;  
}
```

さて、書いてもらって既にお腹いっぱい感があるかもしれないけど、なんか長いし、ワケワカラ  
ン用語も出てきたりと…ちょっとうんざりしていることだと思う。頑張ろう。

## 5.1 初期化時に必要な用語の解説

初期化に必要な基本的な用語と知識

### デバイス(ID3D11Device)

最も基本になるクラスである。理解を進めていくうちに、おいおい分かってくると思う  
けど、とにかく全体的な管理をやってくれるクラスである。

まあ簡単に言うとハードウェアに色々とお願いするために必要なやつである。

### デバイスコンテキスト(ID3DDeviceContext)

描画関係の基本となるクラスである。GPU先生と深い関わりを持っている。つまり  
このGPU先生にコマンド投げる役割を持っているのである。

シェーダ投げたり、画面クリアしたりするときにお世話になります。

### スワップチェーン(IDXGISwapChain)

ゲームってのは画面がちらつかないよう、複数の画面を用意して、それを切り替えるこ  
とにより綺麗に画像を表示しているわけなんだけど、そのために表画面と裏画面が必  
要になる、そいつを管理するクラス。

### ビュー

この名前が非常に紛らわしいんだが DirectX11 やってるとレンダーターゲットビュー  
だの、シェーダー・リソースビューだのが出てくる。

コイツを視界とかビュー行列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。

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やつなら、ビューティーの「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。

### レンダーターゲットビュー(ID3D11RenderTargetView)

こいつはレンダリング結果の出力先(レンダーターゲット)を管理するクラスというわけ。通常の出力先はウィンドウの中なわけだから、ウィンドウの中をレンダーターゲットとして設定しておけば良い(デフォルトでそうなっている)。

### ダブルバッファリング

こいつを説明するには画面のチラツキについて知らなければならないのだが、そもそもウィンドウに絵を描画するってのは、ものすごい高速で1ピクセルずつ画面に色をつけてるわけ。

で、それをコンピュータが一所懸命書いたり消したりしてるわけなんだけど、その過程が見えてしまうと画面が一瞬消えたりして非常に見苦しい。

これに対応するため、表画面と裏画面という二つのバッファを用意し、表画面(見えている画面)に描画するのではなく、裏画面(見てない画面)に描画し、書き終わったら(適切なタイミングで)表と裏を入れ替えるという技法。

DirectX9の時は、この辺意識しなくとも自動でやってくれてたんだけど、DirectX11の場合は明示的に設定しないかなければならない。めんどう。

### フィーチャーレベル

機能のレベルのこと。要はこのプログラムはDX10の機能を使うのか DX11の機能を使うのか、それとも DX12の機能を使うのか…そういうレベルを指定します。ちなみに 10,11,12 と言いましたが、実際には

10.0, 10.1, 11.0, 11.1, 11.2, 11.3, 12.0

などのようにマイナー番号まで振られています。今回の例では 11.0 を採用する感じですね。もし 11 全般にサポートしたい場合は、フィーチャーレベルを配列で指定して、その後にフィーチャ数を指定します。

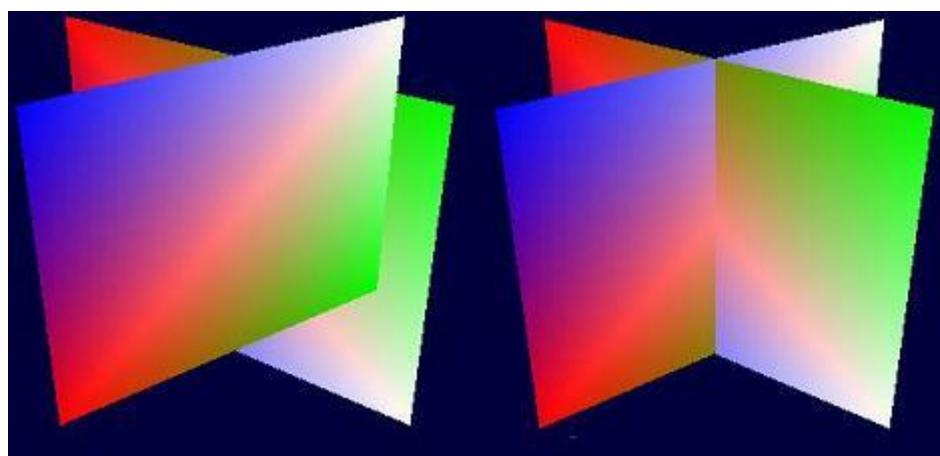
### Zバッファ(深度バッファ)

数年前までは Zバッファといふバッファがあつて、こいつは「Zバッファ法」という技法で使用されるバッファ(メモリ領域…正確にはテクスチャ領域)の事である。

ゾーナンス法ってのは、物体を画面上に描画する際に $Z$ 値(視点からの距離)も一緒に書き込む。イメージしづらいかもしれないが、コンピュータが1ピクセル描画すると同時にそのピクセルが3D空間上で視点から離れている距離を書き込む。

これにより重なった物体が正確に表現することができる。逆にこれがないと、見えないものが見えてしまったりすることとなる。

ゾーナンス法を使ってるものと使ってないものの違い↓



なんとなくおわかりだろうか？

本来はゾーナンス法を用いければ右側のようにクロスするはずだが、ゾーナンス法が入っていないと左のように「クロスしない」図形になる。

昔はゾーナンス法なしで描画していたので、ソートといって、オブジェクトごとソートしていた。あとゲーム的なテクニックで「そもそも重ならないようにしていった」です。

ともかくゾーナンス法のおかげでこれが可能になったのです。

ちなみに DirectX11ではゾーナンス法とは言わず、深度ゾーナンス法と言う。 $Z$ が使えないくなつた理由は UE4 のように「 $Z$ 方向は奥行きでなく、上である」というのが出てきたからである。

### ミップマップ

<https://ja.wikipedia.org/wiki/%E3%83%9F%E3%83%83%E3%83%97%E3%83%9E%E3%83%83%E3%83%97>

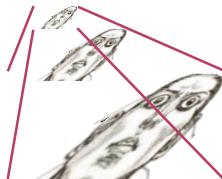
多分今回は使用する機会がないが、軽く説明しておきます。ミップマップはまずゲームでしか使わないものです。

どこで使われるのかといふと「テクスチャの拡大縮小」で使用されます。

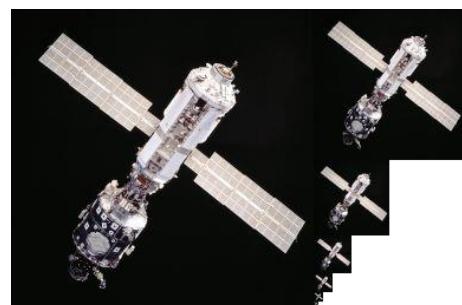
そもそも3Dだと奥行きがあるため、通常にテクスチャは拡大縮小されています。遠近法がありますからね。

テクスチャを拡大縮小するんですが、リアルタイムのテクスチャ拡大縮小ってのはそれほど綺麗じゃないんです。

このため、元の大きさから縮小した画像と、元の画像を組み合わせて一つのテクスチャとします。



イメージとしてはこんな感じ。縦横半分半分にどんどん小さい画像を組み合わせます。メモリモデル的には



こんな感じで配置されています。この授業ではまず使ないので、こういうものがあるってことを知つたらいいです。ちなみによく言われることですが、当然ながらメモリ使用量は増大します。

## ビューポート

3DCGってのは、座標変換(プロジェクション変換)をすることで3Dの頂点情報を2Dにつぶして表示しています。それはわかりますかね? 3Dに遠近法を加味しつつ2D平面上に射影している。そんなイメージです。

でもそれだけで終わらないのよね。その変換だけじゃ3D→2Dだけで表示すべき画面のことは考えられていません。画面上で最終的に表示すべき場所への変換をビューポート変換といいます。

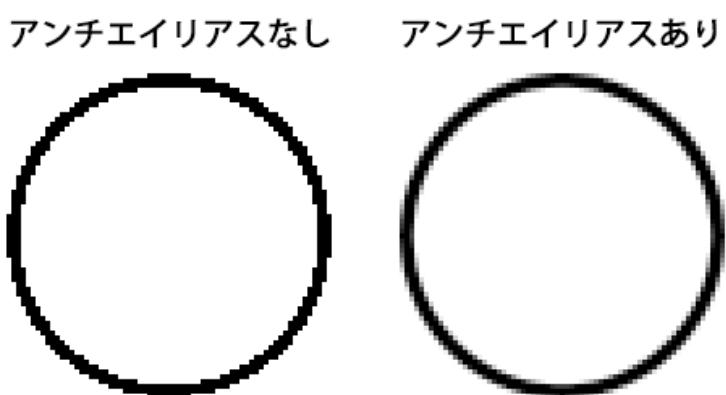
ですからこの設定を忘れるとなにかが表示されません。結構忘れやすいので気をつけましょう。

### サンプリング

サンプリングって色々な場面で使われる言葉ですが、CGにおけるサンプリングは1ピクセルの色をどうやって決めるのかってんのに関係している。

特に今回のDirectX11の場合は、アンチエイリアシングのために使うパラメータで、一般的なアンチエイリアシングは「マルチサンプリング」という技法を使った際のことだ。

3DCGでは特に工夫せずにレンダリングすると、ピクセル毎に色を決定する関係上、工ッジがたがたになることがある。これをジャギーという



ご覧のとおりですわ。

それをちょっとマシにするのが「アンチエイリアシング」と言います。

[https://msdn.microsoft.com/ja-jp/library/bb172267\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb172267(v=vs.85).aspx)

で、色を決定する際に複数のポイントから色を決定して、その平均をとるなどしてジャギをマシにするのをマルチサンプリングアンチエイリアシングといいます。

DX11ではその機能が実装されていて、初期化のパラメータで決定することができます。

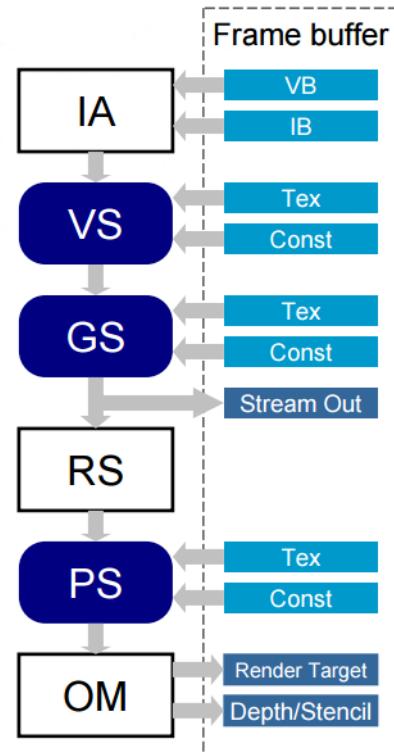
今回は使いませんが、知つとくべきことです。

最初はこれくらいを知っておけば十分でしょう。

## 5.2 初期化中に出てきた OM とか RS とかってなんのさ？

これは初期化だけではなく、色々なところに出てくるのですが

- New pipeline in Direct3D
- IA = Input Assembler
- RS = Rasterizer Stage
- OM = Output Merger



Direct3D のシェーダーパイプラインはいくつかのステージ(役割順序)に分かれています。その頭文字をとったものに過ぎません。

ただ、この頭文字をアタマに入れておくと、グーグルセンセーやインテリセンス先生にお願いするときに便利です。

IA(入力アセンブラー): 後で出てくる頂点バッファとインデックスバッファをまとめる系…要は頂点情報(とインデックス情報)を GPU 側に投げるときに使われます。

RS(ラスタライザステージ): ラスタライザ(3D のベクタ情報をラスタライズする関係の部分)に出てきます)ビューポート設定もラスタライザと関係しているのでこれに属します。

OM(アウトプットマージャー): 新しく描画されるピクセルと、バッファに既に描画されている効果を結合し、新しいピクセル値を生成する。

こんな感じです。なお、VS, GS, PS はシェーダ関連です。これらは全て GPU 側に係る処理なので DeviceContext が面倒を見ることになります。

### 5.3 初期化の大まかな流れについて

初期化が問題なく出来たところで、今回の初期化処理のおおまかな流れについて説明します。

- ① 最初は CreateDeviceAndSwapchain 関数で、ハードウェアとアプリケーションをつなぐ
- ② 次に画面そのものをレンダーターゲット化する。これによりレンダーターゲットに加えた変更を画面に反映する(そのための準備ができた)
- ③ 次にデプスバッファ(テクスチャ)を作成し、深度Stencilビューを作る。これにより深度バッファ法が有効になる。
- ④ 最後にビューポートを指定することで、レンダーターゲットへ描画したものを適切にウインドウに表示させる。

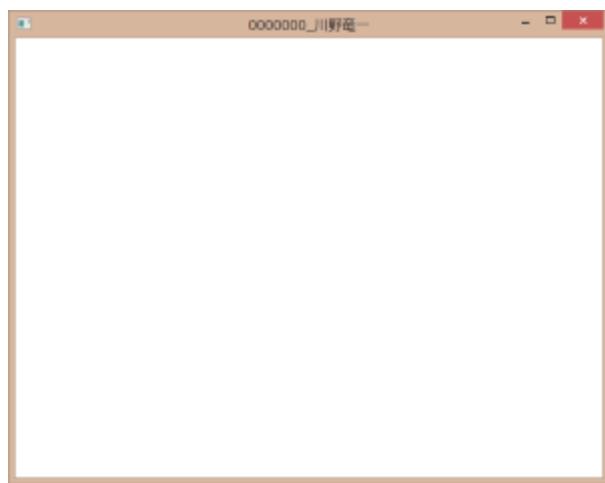
そういう感じの流れです。

## 5.4 きちんと初期化できているかどうかチェック

正しく初期化できているか…つまり Direct3D アプリ側とハードウェアが正常につながっているかどうかのチェックを行います。

どうチェックすればいいのかというと、DirectX の画面を変更する関数をコールして、予想通りに画面の変更が行われれば正常に動作している。ハードウェアとつながったと思っていいです。

特に何もしなければ、現在はこのような画面だと思います。



この画面に変化が現れればいいというわけです。一番簡単な確認方法として、画面の色を変更してみましょう。画面の色を変更するには

- 画面クリア処理

をやればいいんだけど、これだけをやっても画面は変更されません。何故なら「裏面」に書いてるからです。

裏面と表面を入れ替えるのは Present 関数を使用します。ちなみにフリップ関連はスワップチェインなので \_swapchain->Present なんていうふうに実行します。

はい、というわけで使用する関数は

`ID3D11DeviceContext::ClearRenderTargetView()`

[https://msdn.microsoft.com/ja-jp/library/ee419570\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419570(v=vs.85).aspx)

となります。皆さんには大丈夫だと思いますが、使い方はわかりますね？まさか

```
ID3D11DeviceContext::ClearRenderTargetView(パラメータ);
```

みたいになれば使い方はしませんよね？毎年いるんですけど…。あのね、MSDNでこう書いている場合は、ID3D11DeviceContext型のオブジェクトからコールしろって意味なのよね。だから

\_context->ClearRenderTargetView(パラメータ);が正しい。

なお、こんな人はいけないと思いませんけど

```
void ClearRenderTargetView(  
    ID3D11RenderTargetView* pRenderTargetView, //レンダーターゲットへのポインタ  
    const FLOAT ColorRGBA[4] //色情報(RGBA)  
) ;
```

をkopipeしても動きませんよ？これも毎年いますけど、これをkopipeして「動きません」って…動くかボケ!!!

ちなみに第一引数には既に作ったレンダーターゲット。第二引数には色配列を入れてください。色配列は

```
float color[4]={1.0f , 0.0f , 0.0f , 1.0f}; //RGBA
```

で指定して入れとけばいいです。気が利く人は、色を変更したり、動的に色が変わるようにしてみてください。

あ、忘れてましたね。最後にPresent関数を呼びましょう。

```
_swapchain->Present(0,0);
```

とりあえず引数は0,0でいいです。

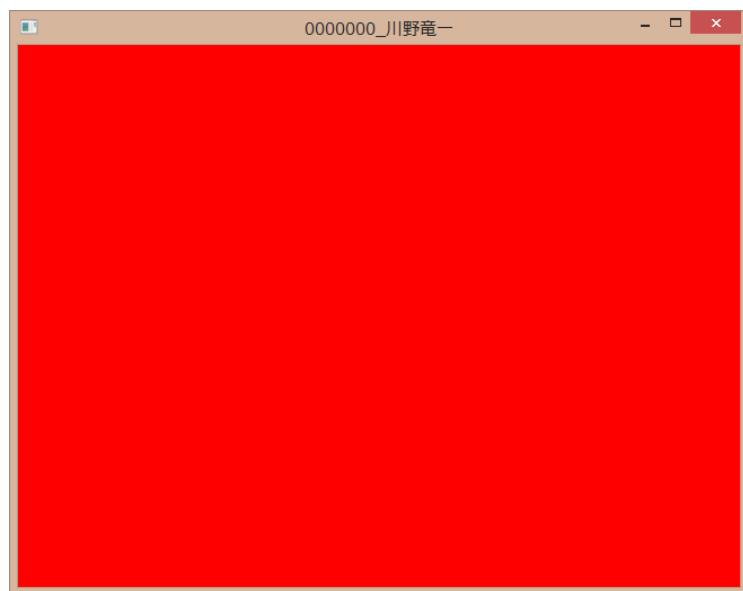
一応意味はあるんですけど、ごくごく分かりにくいです。

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

書いている説明は単純なんですけど、垂直同期の話とかハードウェアの理解が必要になります。

とりあえず今は0,0にしてください。

うまく行けばこんな感じで色が変わった画面が表示されます。

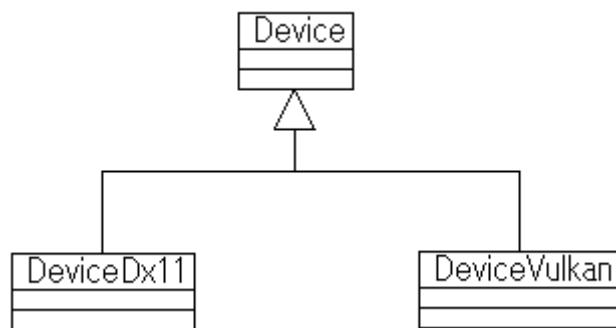


## 5.5 初期化処理のクラス化

昨年はクラス分からん人が多かったのでクラス化するのためらってましたが、みんな分かるっぽいので遠慮しないでクラス化します。そっちのが楽です。

名前は DeviceDx11 ってクラスにします。

できれば最終的にこんな構造にしたいんですが、まあそれは将来の夢ですね。



とにかく DeviceDx11 を作りましょうか…。

変数として内部に

- `HWND _hwnd;`
- `IDXGISwapChain* _swapchain;`
- `ID3D11Device* _device;`
- `ID3D11DeviceContext* _context;`

を持っておきましょう。そして、`Init` 関数と、デバイス、デバイスコンテキストへのアクセスを提供しましょう。

```
class DeviceDx11{  
  
private:  
  
    HWND _hwnd;  
  
    IDXGISwapChain* _swapchain;  
  
    ID3D11Device* _device;  
  
    ID3D11DeviceContext* _context;  
  
    DeviceDx11();  
  
    DeviceDx11(const DeviceDx11&);  
  
public:  
  
    static DeviceDx11& Instance(); // インスタンスを返す。  
  
    // 初期化に成功したら true、失敗したら false を返す。  
  
    bool Init(HWND hwnd);  
  
    IDXGISwapChain* SwapChain(); // _swapchain を返す  
  
    ID3D11Device* Device(); // _device を返す  
  
    ID3D11DeviceContext* Context(); //  
  
};
```

というわけでさっきの `InitDirect3D` をこっちに移植しましょう。

### 5.5.1 シングルトンクラス

うむむ…当たり前のようにシングルトンクラスの体で書いちゃったけど、知らない人とか忘れた人とか分からなくなっちゃった人が多そうですねえ。

Singletonパターンの解説をすると…アプリケーション全体の中でたった一つだけのインスタンス(実体)が存在することを許されるクラスって意味です。どういう意味かというと

```
class A{
```

```
};
```

```
A a;//
```

```
A b;//
```

などと書くとAっていう実体が2つあるわけです。これを許さないクラス。

なんでこれが必要なのがというとゲーム全体、アプリケーション全体を管理するクラスが2つ以上あつたら困るでしょう？

これを1つだけっていう事を保証するクラス。これがシングルトンクラスです。まあ…言い方を変えると「お行儀の良いグローバルオブジェクト」です。

とにかくそのクラスの実体がアプリケーション全体で一つだけになるようにすればいいんですけど、このためにはAppの実体がstaticである必要があります。

ただしこれだけでは当然Appはいくらでもできてしまいます。このため、自由に実体を作らせないようにしなければなりません。

どうすればいいのでしょうか？

そう…ですね。これは知りてないとできないことなので、いきなり答えを言います。

それは「コンストラクタをprivateにする!」です。

これ、どういうことがわかりますか？

意味はわかるでしょうけど、意義は、ちょっとむずかしいと思います。

さらっとだけ説明します。クラスで private に設定しているメンバ(関数、変数)は呼び出し側からは見えないという話は覚えてますか? private に設定している関数は呼び出せないんです。

ところで、「コンストラクタ」も当然ながら関数ですので、private に設定してしまうと、この関数を呼び出せなくなります。

ところで、「コンストラクタ」はいつ呼び出されるのでしょうか? そう、実体ができる瞬間でしたよね?

こいつを private にするということはどういうことかというと→「実体が作れない」ことになります。

???? 実体を作れないって? それでどうやってプログラミングするの!!

馬鹿なの? 死ぬの?

いやいや、「どうにか出来る」から、この手法が使われてるんですよ、これが。これもまた、知らないいと思いつかないと思いませんので、結論から言っておきます。

……フヒヒ。

「**実体(インスタンス)をクラス自身に作らせる**」んです。いくら private であっても、そのクラス自身は使える(見える)んだから、そいつに作らせちゃおうってわけwww。どうだ驚いたか!!



フヒィーーー! そんなトンデモナイ思考に痺れる憧れるッ!!!

その「**トンデモナイやり方**でできた実体」を返す関数を作ればいい。ということになりますよね？

あれれエー？でも、それってその関数を呼ぶたびに実体が作られてしまうから、結局トラブルを解決できなくない？そこで使うのがローカル static 変数(と static 関数)です。

ローカル static は C 言語でもあるので、わかつておいて欲しいですが、ある関数内で static 宣言すると、それはアプリケーションが終了するまで残り続けます(ローカルのくせに)。

例えば、この実体を返す関数を Instance() という関数にするのならば、こんな感じになります  
...

///アプリケーションクラス

```
class App
```

```
{
```

**private:**

```
App();
```

**public:**

```
~App();
```

///App のシングルトン実体を生成し、それを返す。

```
static App& Instance(){
```

```
    static App app;
```

```
    return app;
```

```
}
```

```
void TestFunc(){
```

```
    cout << "シングルトンクラス生成済み" << endl;
```

```
};  
};
```

というクラスを作つておいて、この機能を使いたいときは

```
App::Instance().TestFunc();
```

のように呼び出すかもしくは

```
App& app=App::Instance();
```

```
app.TestFunc();
```

のように呼び出す。

…おわかりだろうか？シングルトンクラスを作る手順を書くと

1. コンストラクタを `private` にする
2. インスタンスを返す `public` アクセサ(今回は `Instance()` 関数)を作る
3. アクセサから「参照」とつてきてそれを利用する

これが一連の流れです。がつ…ダメ!! それでは 50 点…………!!

まだまだ複数のインスタンスは作成可能!!! なぜならまだこのクラスは代入したりコピーしたりが可能だからだ!!!

現在の状況を見てください。関数内で `App` 型の `app` 変数を `static` 宣言しています。その参照を返す関数を作りさえすれば、いつでもどこでも「たったひとつだけの」`App` 型の実体にアクセスすることができます。

でも、これだけでは足りません。シングルトンクラスとはいって、「コピー」することはできてしまします。コピーするには、以下のように初期化時に他の App 型の変数で初期化することができます。

```
App app2 = app;
```

この場合、App の「コピー・コンストラクタ」というコンストラクタが呼ばれます。これは、以下の様なコンストラクタで宣言されるもので、宣言しないなら自動でこのコンストラクタは自動で生成されています。

```
App(const App& app);
```

この関数が呼ばれて、既に作られているものから、新しいクラスを生成することができてしまします。ちなみに代入もできないように(イコール演算子を無効化)しておかなければなりません。

え？ イコール演算子(=)を無効化することなんてできるの？

…できます。

C++には演算子オーバーロードという機能があって、あるクラスに対する演算を自分で勝手に定義できるわけです。もちろんニも例外ではありません。

ちなみにこれを使って、Vector2D(ベクトル 2D)クラスや Vector3D(ベクトル 3D)の足し算引き算掛け算を定義できますね。

それはともかく、演算子をオーバーロードするには、operator ってやつを使います。

自分の型の代入演算子オーバーロードはこんなふうに定義できます。

```
App& operator=(const App&);
```

これで、代入もコピーも禁止されました。ちなみに、このコピー・コンストラクタと、二オペレータは、シングルトンの性質上、使われることはなくなりますので、実装は書かなくてもOKです。ここまで話を総合して、Appというシングルトンを作るならば以下のようなクラス定義になります。

```
///アプリケーションクラス

class App

{

private:

    App();
    App(const App&);

    App& operator=(const App&);

public:

    ~App();

    bool Start();

    ///App のシングルトン実体を生成し、それを返す

    static App& Instance() { // クラス自身から呼び出すので static
        static App app; // app は static なので 世界に一つ しか存在しません
        return app;
    }

};
```

ここまでを理解した上で、DeviceDx11 クラスをシングルトン化しましょう。

はい、初期化処理を DeviceDx11 クラスに移植した上で Direct3D の初期化ができたことを確認できたら、次行きましょう。

### 5.5.2 構造体とかクラスのプロトタイプ宣言

ああ、あとついでなので構造体とかクラスのプロトタイプ宣言についても解説しておきます。

「プロトタイプ宣言」って知っていますか？C言語では前方参照とか、前方宣言とか呼ばれるものなので、何となく聞いたことはあると思いますが…

プロトタイプ宣言というのは、関数や構造体などを作成する際に、前もってその存在をコンパイラに知らせておくための方法です。宣言だけして、中身を定義していないものです。

C言語では恐らく「関数のプロトタイプ宣言」がよく使われると思いますが、C++ではクラスや構造体のプロトタイプ宣言が使われることになります。もちろん関数のプロトタイプ宣言も使われますが、それは知ってるでしょ？

ということで、構造体とかクラスのプロトタイプ宣言の話をします。

恐らく今回の DeviceDx11 クラスを作成する際に IDXGISwapChain だの ID3D11Device だのをメンバに追加する際に必要なので、DeviceDx11.h の先頭で

```
#include<d3d11.h>
```

と書いたのではないかと思います。

うん、別に間違いないじゃないし、就職で減点されることもないとは思います。

ただ、ヘッダにさらにヘッダをインクルードするのはよろしくないって思想があつたりします。なお、C++の場合、継承元はインクルードすべきなんですがね。

そう、今いじっているのは DeviceDx11.h なので更にここに #include<d3d11.h> を置くのはあまり良くないのだ。

そもそも ID3D11Device は実体ではない（ポインタとして宣言されてる）ので、ヘッダが ID3D11Device の定義を知っている必要はないのだ。

C++の作法として、インクルードしなくてもいいものは極力インクルードしないって思想があるのだ。これはもちろん意味があって、フリーにコンパイルが重くなるってのがあるのだ。だから出来る限りインクルード分は cpp ファイルの方に書くべきであって、hの方に書くべきではない。

余談だが、hの方に書くべきではないものに

```
using namespace OO;
```

がある。これにはまつとうな理由があるが、ともかくヘッダファイルはできるだけシンプルにしておきたい。

というわけで d3d11.h をインクルードしない。でもそうするとエラーが発生する。そんなとき役に立つのがプロトタイプ宣言。

```
class DeviceDx11
{
private:
    HWND _hwnd;
    ID3D11Device* _device;
    ID3D11DeviceContext* _context;
    IDXGISwapChain* _swapchain;
```

だけだと当然宣言されてないので怒られるのだが、ここで

```
struct ID3D11Device;
struct ID3D11DeviceContext;
struct IDXGISwapChain;
class DeviceDx11
{
private:
    HWND _hwnd;
    ID3D11Device* _device;
    ID3D11DeviceContext* _context;
    IDXGISwapChain* _swapchain;
```

と書くことにより怒られない。もちろんcpp側ではインクルードしなければならないので注意が必要だけどね。

あ、プロトタイプ宣言自体のやり方は

構造体の実装を書かずに

struct 構造体名;

クラスの場合は

class クラス名;

とするだけです。簡単でしょ？これで#include<d3d11.h>をインクルードしなくても済むようになります。

## 6 三角ポリゴンの表示

さあ…いよいよちょっとしたハードルがやってきました。三角形ポリゴンの表示。これは DirectX9まではまだ簡単だったんですけど、DX11ともなるとそうもいきません。

なぜならば…

ポリゴンを表示するにはシェーダを書かなければならぬからだ!!!!

というわけで「たかが」三角ポリゴン表示に恐らく数時間(3時間くらい)かかるであろう。覚悟しこう。

ちなみに最初に表示するポリゴンは、2Dのポリゴンだ。何しろ3D座標変換のための行列をまだ数学で教えていないからね。

三角ポリゴン表示までの手順を大きく分けると、

1. 頂点を配列で定義(頂点情報というメモリの塊)
2. 頂点バッファの作成(先に定義した頂点情報を入れる)
3. 頂点バッファのセット
4. プリミティブオペレーションの設定
5. 頂点シェーダとピクセルシェーダを書く
6. 頂点シェーダとピクセルシェーダをロード
7. 頂点シェーダとピクセルシェーダをセット
8. 頂点描画命令

うん、またかだか三角形ポリゴン表示にこの手間がかかるのだ。それでも DX11 地獄のほんの入り口に過ぎないのだ。

## 6.1 頂点を配列で定義

ここはみんなにとっては簡単だと思いますが…。

今回は手っ取り早く作るためにホントに頂点情報のみが入っているデータを作ります。

```
struct Position{//頂点1つあたりの情報  
    float x,y,z;  
};
```

これが頂点1つあたりの情報になります。あとはこれを配列化するだけ。

```
Position vertices() = {  
    {x0, y0, 0},  
    {x1 ,y1 ,0},  
    {x2 ,y2 ,0},  
};
```

頂点は二次元座標系基準のでいいので、適当に定義してください。

## 6.2 頂点バッファのセット(書き換え)

頂点バッファは作っただけじゃGPUはそれを知りません。持っている頂点情報はGPU君に送信しないとGPUは表示することすらできません。

まあなんということはない。

IASETVERTEXBUFFERS

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

を使用します。これはGPUとのやり取りなので、案の定DeviceContextの持ち物ですね。

```
void IASETVERTEXBUFFERS(  
    UINT StartSlot,//スロット(0でいい)  
    UINT NumBuffers,(バッファの数=1でいい)  
    ID3D11Buffer *const *ppVertexBuffers,//頂点バッファ配列へのポインタ  
    const UINT *pStrides,//ストライド値配列へのポインタ  
    const UINT *pOffsets//オフセット配列へのポインタ  
);
```

入力スロットなんですけど、これって「スロット」って何なの?って思う人もいると思いますが、投げられるバッファの番号…と言ってもよくわからないか…うーん、頂点バッファはもちろん複数投げができるんですけど、イメージとしては道路のレーンみたいなもんですね。



それぞれの料金所に車が入っていくように頂点情報が入っていくと思ってください。より正確なモデルとしては複数のホースに水とか流し込んでるイメージのほうが近いです。

今回は一つしか使わないのとりあえず0にしといてください。道路の話だと0番レーンだけ使うってことです。

第二引数のバッファの数なんんですけど、これはバッファは配列で流し込む事ができるので、その配列数を聞いています。

今回は配列の数=1なので1にしてます。

次がバッファ配列のアドレスですね。これも配列なら配列名なんんですけど、1つだけなので、さっき作ったバッファに&つけとけばいいです。理由はわかりますね？

同様に「ストライド値」「オフセット値」も配列扱いだ。ちなみに「オフセット」は分かるだろうか？基準からどれくらいズしているのかを表すのだ。バッファデータを先頭から読み取るとは限らない。途中からの値が必要なときもある。その場合に使用する。が、今回は先頭データから読み取るのでオフセットは0でいい。

では「ストライド」とはなんだろうか？

<http://eow.alc.co.jp/search?q=stride&ref=sq>

うーん。「大股で歩く？」とか書いてるが、この場合は「歩幅」の意味が近い。つまり1頂点あたりのバイト数を聞いているのだ。

今回はfloat型のx,y,zが1頂点辺のデータなので sizeof(float)\*3というわけだ。  
sizeof(Position)のほうが良いかな。そういうことだ。

そうなると

```
unsigned int stride=sizeof(Position);
```

といったところか。

で

```
context->IASetVertexBuffer(0,1,&vb,&stride,&offset);
```

という感じでプログラムすればバッファがGPUにセットされる。

…ふう。

長いな、ホント。でもまだ終わらんよ？

次はプリミティブトポロジの設定だ

## 6.1 頂点バッファの作成

先程作った頂点はそのままでは GPU に投げることができません。どういう状態なら投げることができるのかといふと「頂点バッファ」という状態にして投げることができます。

そこで CreateBuffer 関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/ee419781\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419781(v=vs.85).aspx)

で、最後の引数でバッファを受け取るのはわかるんだけど、第一、第二引数がよくわからんね。  
まあ、第二引数は初期化データってことなので、さっきの頂点データを放り込めばいいことはなんとかわかるか。

とりあえずここを見る

[https://msdn.microsoft.com/ja-jp/library/ee416048\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416048(v=vs.85).aspx)

うん、なんか構造体をきっちり定義すればいいことがわかる。

とりあえず BufferWidth については、さっきの頂点データのサイズを書いてくれ。つまり sizeof(Position)\*3 とか sizeof(float)\*9 である。

まあ…今回の場合は sizeof(vertices) でオッケーだったりはするんだけどね。

Usage に関しては、D3D11\_USAGE\_DEFAULT

bindFlags は、頂点バッファなので D3D11\_BIND\_VERTEX\_BUFFER

CPU アクセス必要ないので、CPUAccessFlags=0 で

MiscFlags も 0 でいいです。

ByteWidth は、バッファの全サイズを…つまり float\*3\*3 な

そこまで書けたら次は SUBRESOURCEDATA ですが簡単です。

pSysMem にさっきの頂点情報のアドレスを入れれば終わりです。

```
D3D11_SUBRESOURCE_DATA initData={};
```

```
initData.pSysMem=vertices;
```

で CreateBuffer すれば、頂点バッファ化された頂点情報が得られます。

頂点バッファ名は vBuffer とか vb とでもしておけばいいでしょう。僕は面倒なので vb にしておきますが、DX初心者の皆さんにはきっちり vertexBuffer と書いたほうが良いかも知れませんね。

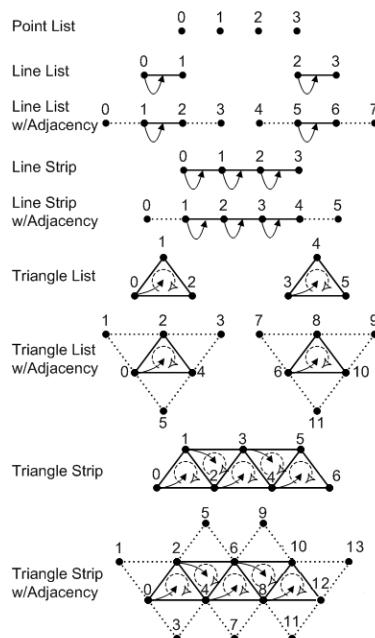
さて、CreateBuffer でバッファの生成ができたと思いますので、いよいよ 頂点情報を Direct3D に（グラボに）送ることができます。

そうやって作った頂点バッファをセットしないと GPU には送られません。

## 6.2 プリミティブトポロジの設定

プリミティブトポロジって言葉が既に意味不明なんだが、一応言っておくと、

「もらった頂点情報をどんなふうに利用するのか」というルールのことなのだ。



ここを見て分かるやつは相当カンガレル！というか頭がレル！ですね。

簡単に言うと「頂点を頂点リストとして扱う」「頂点と頂点データから辺データとして扱う」「頂点3つを面データとして扱う」とかいうのを設定するものです。

今回は三角形面データなので TriangleList ですね。

IASETPRIMITIVE TOPOLOGY 関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/ee419690\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419690(v=vs.85).aspx)

引数はひとつだけなので簡単です。

ここから一つ選ぶだけ

[https://msdn.microsoft.com/ja-jp/library/ee416253\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416253(v=vs.85).aspx)

で、三角形リストでやっていく予定なので D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST を指定してください。

さて、もうひと頑張りですよ？

いよいよお待ちかね「シェーダ」を書いてもらいます。

### 6.3 頂点シェーダとピクセルシェーダを書く

シェーダ書き始めると、まあプロっぽい感じがするな。

とりあえずプロジェクトで追加→「新しい項目」ってやるとダイアログボックスが出るので「HLSL」を探してください。



こういう画面が見えると重いです。実際「HLSL ヘッダーファイル」以外だったらなんでも良いので、適当に「頂点 シェーダ ファイル」ってのでも選択してください。

とりあえず名前は勝手に決められそうになるので、「基本のシェーダ」って意味で BaseShader とでもしておきましょう。

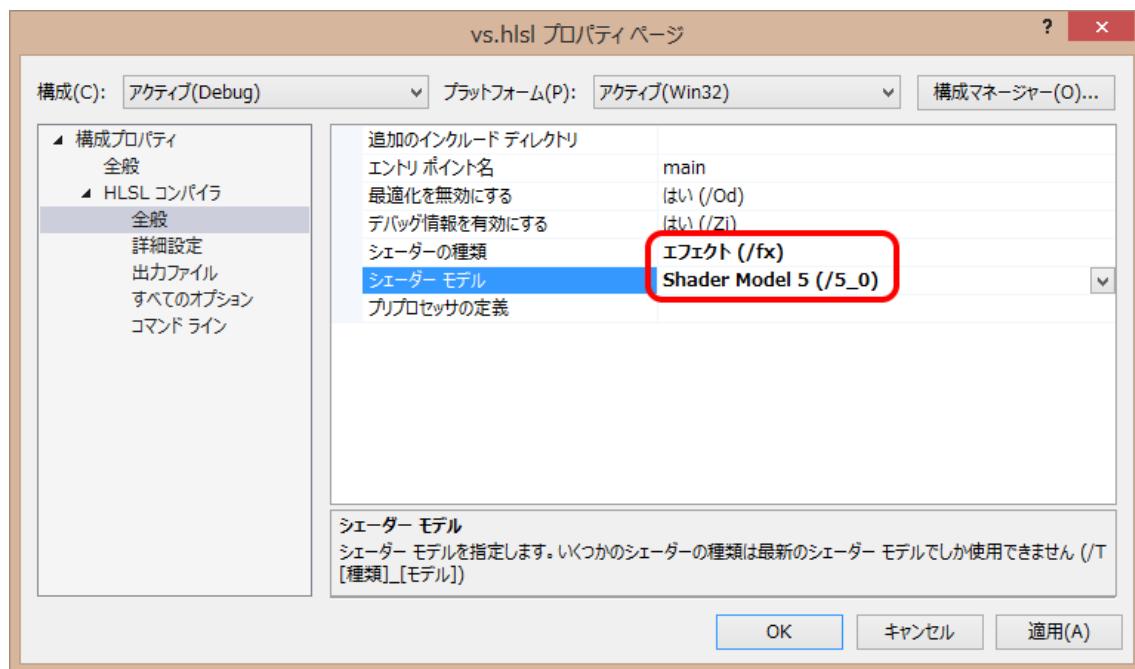
そうするとプロジェクトに追加されていますね。デフォルトの頂点シェーダが既に書かれています。今回は main 関数を使用しないので

関数名を VS とでもしておきましょう(VertexShader の略ね)

さて、これで頂点シェーダはオッケーなんんですけど、実行しようとしてみてください。怒られま  
すよね？

なんか main がないとダメとか言って怒られるので、黙らせます。

さっき作ったシェーダ(hlsl)を右クリック→プロパティ→全般



その部分をこう書き換えます。そうしないと main 関数ないと怒りますよ。

さて実行してみてください。できたでしょうか？ではピクセルシェーダを書いていきます。

プログラムの規模が小さい間は頂点シェーダファイルの中にピクセルシェーダ書いたほうが  
わかりやすいと思いますので、もう、いつぺんにこう書いちやいます。

```
float4 BaseVS( float4 pos : POSITION ) : SV_POSITION
{
    return pos;
}
float4 BasePS(float4 pos : POSITION) : SV_Target
{
```

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

何やってつかはまた三角形の表示ができてからお話をしたいと思いますので、今はこんな感じでシェーダコンパイル通してください。

さて、シェーダが書けたのであとはロードです。

## 6.4 シェーダのロード&コンパイル

シェーダをロードするには

D3DX11CompileFromFile 関数を使用します

[https://msdn.microsoft.com/ja-jp/library/ee416856\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416856(v=vs.85).aspx)

こいつは頂点シェーダおよびピクセルシェーダ両方で使えるやつですが、これ一個では頂点シェーダオブジェクトとピクセルシェーダオブジェクトをそれぞれ一つずつしか生成できないので2回同じ関数で同じファイルをロードすることになりますが、まあしょうがないと思つてください。

まずは頂点シェーダです。

```
//シェーダロード&コンパイル  
ID3DBlob* compiledShader = nullptr;  
ID3DBlob* shaderError = nullptr;  
  
HRESULT result = D3DX11CompileFromFile("vs.hlsl", //シェーダファイル名  
    nullptr, //DX10を併用するときに必要。使わないならヌルポでいい  
    nullptr, //DX10を併用するときに必要。使わないならヌルポでいい  
    "BaseVS", //関数名  
    "vs_5_0", //プロファイル名(シェーダバージョンやね)  
    0, //コンパイルオプション特に指定なし  
    0, //実行時オプション特に指定なし  
    nullptr, //スレッド使わないし指定なし  
    &compiledShader,  
    &shaderError,  
    nullptr //即時復帰関数時に使うが今回は完了復帰なのでnullptrでいい  
);
```

こんな感じでロード&コンパイル書いてください。書けたら実行してリザルト値を確認してください。エラーが起きてなければ次に行きます。

実はロード&コンパイルがおわっただけでは足りないんですね。長いね…ホント。

はい、コンパイル済みシェーダができるんですけど、これじゃあまだGPUに投げられないんですわ。

GPUに投げられるようにするにはこのコンパイル済みシェーダを頂点シェーダオブジェクトに変換する必要があります。

#### 6.4.1 頂点シェーダオブジェクトの作成

CreateVertexShaderです。

[https://msdn.microsoft.com/ja-jp/library/ee419807\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419807(v=vs.85).aspx)

カンガレル人はこれを見るだけで書けるかなーと思しますが…。まあ知らないと難しいかな。

```
HRESULT CreateVertexShader(  
    const void *pShaderBytecode,//コンパイル済みシェーダコード  
    SIZE_T BytecodeLength,//↑のシェーダコードのサイズ  
    ID3D11ClassLinkage *pClassLinkage,//使わないのでもnullptrでいいでしょう  
    ID3D11VertexShader **ppVertexShader//受け取りたい頂点シェーダオブジェクト  
);
```

これだけ見ると簡単そうなんですけど、んじゃあ例えば第一引数はどうすんのさ?って思うんですね

さつき作ったコンパイル済みシェーダのID3DBlobってオブジェクトは、GetBufferPointer()って関数とGetBufferSize()って関数があります。

```
ID3D11VertexShader* vs=nullptr;  
  
HRESULT result = device.Device()->CreateVertexShader(shader->GetBufferPointer(),  
                                                    shader->GetBufferSize(),  
                                                    nullptr,  
                                                    &vs);
```

これでGPUに投げることのできる頂点シェーダができました、と。

でも、コンパイル済みシェーダの出番はこれで終わらないんですね。頂点シェーダの場合はもう一つだけ…GPUへのデータ入り口の設定がいるんですね。

それが頂点レイアウトの設定です。ええ? プリミティプトポロジの設定だけではなく、「頂点レイアウト」と?なんだそれは…。

#### 6.4.2 頂点レイアウト

はい、頂点レイアウトってのは頂点1個あたりにどんなデータがどんな配置で入っているのかを指定します。今回は頂点のデータだけなので POSITIONだけあれば十分です。作り方は

```
D3D11_INPUT_ELEMENT_DESC inputElementDescs[] = {  
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },  
};
```

こんな感じです。中身的には

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

ここを見ていただきたいんですけど、最初から全理解するのはまだ無理だと思います。

そもそも「セマンティクス」って何なんですかねえ…。分かんない人が多いと思います。

ね？理解するには基礎知識が足りてないんですね。ぼちぼち行きましょうや。

ちなみに「セマンティクス」ってのはこの場合「流れてくるデータの扱い方、捉え方」を表しています。なお、今回は“POSITION”というセマンティクスですが、これは「座標」を表していることは分かりますね？

float の3つぶんのデータが渡ってきた時点ではそれを色として扱うか、座標として扱うか分からんでしょう？それに意味付けを行う文字列を渡すと思ってください。

そもそも Semantics ってのは意味論って意味なんですよねえ…ともかく今渡したデータは「座標」として扱いますよって約束をしました。

その後の0は…うーん。ここは0以外見たことないですね。

##### SemanticIndex

要素のセマンティクス インデックスです。セマンティクス インデックスは、整数のインデックス番号によってセマンティクスを修飾するものです。セマンティクス インデックスは、同じセマンティクスの要素が複数ある場合にのみ必要です。たとえば、4x4 のマトリクスには4個の構成要素があり、それぞれの構成要素にはセマンティクス名として matrix が付けられますが、4個の構成要素にはそれぞれ異なるセマンティクス インデックス(0, 1, 2, 3)が割り当てられます。

う～ん。これは複数を具体的に使ってる所を見ないと分からないですね。

ともかく今は0にしておいてください。

次 FORMATですが…これも相当慣れない自分で設定できないと思います。とりあえず今回は「DXGI\_FORMAT\_R32G32B32\_FLOAT」を指定していますが、これは 32bit の float を RGB3つ分を用意している。

実際は XYZ として使うのだが、シェーダでは XYZ であっても RGB として設定したりする。この辺はシェーダ慣れないと違和感がなくなってきた。要は慣れです。

スロットは今回に関しては 0 でいいでしょ。次のはオフセットになってまして、今回 0 って書いてますが、2行目以降を書くときは D3D11\_APPEND\_ALIGNED\_ELEMENT にしちゃいます。

その次は入力データクラスというちょっと謎なやつですが

[https://msdn.microsoft.com/ja-jp/library/ee416243\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416243(v=vs.85).aspx)

この中から選ぶんですけど

頂点データということで、D3D11\_INPUT\_PER\_VERTEX\_DATA でいいでしょう。

最後のは 0 でいいです。なんですか」というと

#### InstanceDataStepRate

バッファーの中で要素の 1 つ分進む前に、インスタンス単位の同じデータを使用して描画するインスタンスの数です。頂点単位のデータを持つ要素（スロット クラスは D3D11\_INPUT\_PER\_VERTEX\_DATA に設定されています）では、この値が 0 であることが必要です。

とあるので…まあそういうことです。

ちょっと説明が長くなりましたが、CreateInputLayout の第一引数にはこれを入れます。

で、第二引数は、今回は要素これ一つなので 1 って書いていいですが、実はこの後ドンドン増えていきますので、今のうちにきちんと考えましょう。

つまりところ D3D11\_INPUT\_ELEMENT\_DESC の配列の数が入ればいいので

sizeof(inputElemDescs)/sizeof(D3D11\_INPUT\_ELEMENT\_DESC)

を入れておけばいいかなと思います。

で、次のパラメータにさっき作った「コンパイル済みシェーダ」オブジェクトを入れます。入方は当然ながら

`compiledShader->GetBufferPointer()`と `compiledShader->GetBufferSize()`を使用します。

最後の引数が受け取るためのポインタのアドレスです。

つまりこんな感じ

```
ID3D11InputLayout* inputLayout = nullptr;
result = device->CreateInputLayout(inputElemDescs,//レイアウト配列
                                     sizeof(inputElemDescs) / sizeof(inputElemDescs[0]),//配列数
                                     compiledShader->GetBufferPointer(),//コンパイル済み頂点シェーダ
                                     compiledShader->GetBufferSize(),//コンパイル済み頂点シェーダサイズ
                                     &inputLayout);//セットしたいレイアウトデータ
```

ここで作成されたレイアウトデータをセットするのは簡単で `IASetInputLayout` を使います。引数一つなので簡単ですね。

[https://msdn.microsoft.com/ja-jp/library/ee419688\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419688(v=vs.85).aspx)

ちょっと自分で考えてセットしてください。

ここまででコンパイル済み頂点シェーダ利用部分はひとまず終りとなります。使い終わったコンパイル済みシェーダは開放する必要があるので

```
compiledShader->Release();
```

で解放させてください。DirectXでは「解放」をこのように `Release` 関数でやることが多いので

でまだ頂点シェーダをロードしただけなので、ピクセルシェーダも必要です。

#### 6.4.3 ピクセルシェーダオブジェクトの作成

ここからはピクセルシェーダに対してロード＆コンパイル→ピクセルシェーダオブジェクト生成をやっていきます。既にコンパイル済みシェーダは開放しているので

なので再びピクセルシェーダのロードコンパイルをかけます。無駄じゃないです。どの道頂点シェーダとピクセルシェーダは分けて作らないといけないんですね。

頂点シェーダのときとほぼ同じだし面倒なので特に説明しませんが、こんな感じで設定します。

```
//ピクセルシェーダのロード
```

```

result = D3DX11CompileFromFile("vs.hlsl", //シェーダファイル名
    nullptr, //DX10を併用するときに必要。使わないならヌルポでいい
    nullptr, //DX10を併用するときに必要。使わないならヌルポでいい
    "BasePS", //関数名
    "ps_5_0", //プロファイル名(シェーダバージョンやね)
    0, //コンパイルオプション特に指定なし
    0, //実行時オプション特に指定なし
    nullptr, //スレッド使わないし指定なし
    &compiledShader, //これがいる奴やな
    &shaderError, //エラーオブジェクト
    nullptr);

```

ピクセルシェーダの方はフツーにここからピクセルシェーダオブジェクトを生成したら終わりです。

```

ID3D11PixelShader* ps = nullptr;
result = device.Device()->CreatePixelShader(compiledShader->GetBufferPointer(),
    compiledShader->GetBufferSize(), nullptr,
    &ps);
compiledShader->Release();

```

はいっ!!長かったね。ゴクローさん。

これでやっと次に行ける。もうひと頑張りだ。

## 6.5 頂点シェーダ、ピクセルシェーダのセット

あ、ここまで来たら大したことしません。

セットだけです。

VSSetShader と PSSetShader を使います。

[https://msdn.microsoft.com/ja-jp/library/ee419766\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419766(v=vs.85).aspx)

[https://msdn.microsoft.com/ja-jp/library/ee419719\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419719(v=vs.85).aspx)

さて、簡単です。

第一引数には先程作った頂点シェーダ、ピクセルシェーダオブジェクトを入れて、第二、第三引数は今は nullptr, 0にしておきましょう。

うん、[https://msdn.microsoft.com/ja-jp/library/ee419534\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419534(v=vs.85).aspx)

をみても正直クラスリンクエージとか分かんなレ(•ω•)レいや、分かるけどちょっと優先順位が低いので今は nullptr, 0にしちゃいましょう。

それで VS と PS を設定しましょう。

それでここは終わりです。拍子抜けでしょ？

さて、オーラスです。

## 6.6 頂点描画

これもシンプルイズベストです。

既に頂点バッファはセットしているので、あとは描画命令出すだけです。

Draw 命令です。

[https://msdn.microsoft.com/ja-jp/library/ee419589\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419589(v=vs.85).aspx)

Present 関数の直前にでも入れてください。

今回は3頂点なので

context->Draw(3, 0)

にしてみましょう。

これで三角形が表示される…はず？

だけどされない人がいると思います。それは…デプスステンシルバッファのクリアをしてないからです。

そもそも設定しなけりゃいけないんですけどね。まあクリアしときましょう。

[https://msdn.microsoft.com/ja-jp/library/ee419569\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419569(v=vs.85).aspx)

第一引数は、デプスステンシルビューを入れましょ。

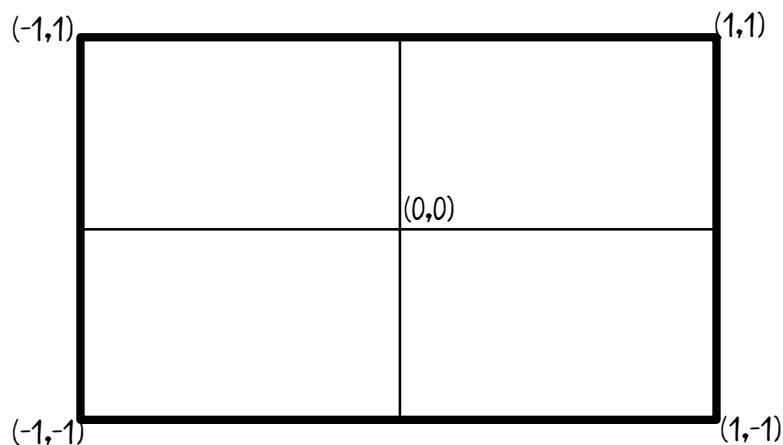
第二引数は D3D11\_CLEAR\_DEPTH にしましょう。

第三引数は 1.0f にしちゃいます。これはそのうち意味がわかります。

最後の引数は 0 で

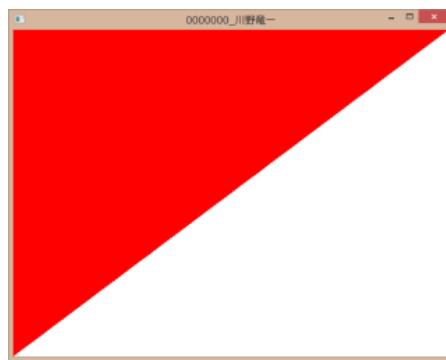
これで三角形が表示されると思います。

ちなみに特に頂点シェーダをいじらないとこんな感じになります。



中心を 0 として、左を -1、右を 1、上を 1、下を -1 という座標系になります。

画面に何も表示されてない人は、ここからはみ出していいのか、もしくは頂点の順番が時計回りにきちんとになっているのかを確認してください。うまいこといったらこんな感じです。



例えばこの場合の頂点情報は

```
Position vertices() = {  
    {1,1,0.0},  
    {1,-1,0.0},  
    {-1,-1,0.0}}
```

```
};
```

こんな感じです。

ちなみにピクセルシェーダの

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

この部分を様々に変更して遊んでみてください。

なお、頂点シェーダをいじると完全に2D座標系のように扱えます。

```
pos.xy -= float2(640 / 2, 480 / 2);
pos.xy /= float2(640 / 2, 480 / 2);
pos.y = -pos.y;
```

これで2D座標系のように扱えます。なお、やりようによってはこのような感じで色をグラデーションにしてしまうことも可能です。その場合は POSITIONだけじゃなく、COLORなどを使用する必要があります。

```
//頂点シェーダからの出力と
//ピクセルシェーダへの入力を兼ねている
//頂点情報構造体
struct OutputP{
    float4 pos:SV_POSITION;//システム座標をこっちに移動
    float4 col:COLOR;//色情報を含ませる(グラデーションのため)
};

//頂点シェーダ
OutputP BaseVS(float4 pos : POSITION)
{
    OutputP output;
    output.pos = pos;//座標情報をまるごと代入

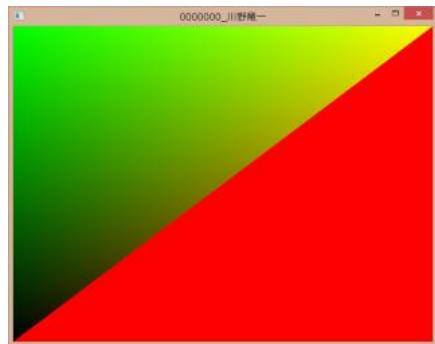
    //2D座標系に変換
    output.pos.xy -= float2(640 / 2, 480 / 2);
    output.pos.xy /= float2(640 / 2, 480 / 2);
    output.pos.y = -output.pos.y;

    //(-1~1にクランプされている座標情報をカラーへ代入)
    output.col = output.pos;
    //カラーのXY(つまりRG)間に座標情報を変換
    output.col.xy = (output.col.xy+float2(1,1))/float2(2,2);
    return output;
}

//ピクセルシェーダ
float4 BasePS(OutputP output) : SV_Target
{
    return float4(output.col.rgb,1);
}
```

シェーダはシェーダ内で struct 使って構造体が使えますので上記のように頂点シェーダからの出力構造体を定義します。これがピクセルシェーダへの入力構造体を兼ねます。

上記のシェーダの演算結果は座標が色に変換されるので



このような結果になります。

この原理を応用すればシェーディングまでできそうですよね。

ともかく一つの目標へたどり着いた感じですね。

さて、でもこのままではDirect3Dを使っているにも関わらず2Dですよね。

「えっ？それは頂点の設定で $z$ 座標に0を入れてるからちゃいますのん？」と思ってる人もいるでしょうが…違います。

今のままではどれほど $z$ 値をいじっても2Dしか表示されません。座標変換というのが必要なのです。



なのです。

さあ行きましょう。めくるめく数学、座標変換の世界へ

## 7 座標変換な"の"です"

### 7.1 行列とは

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。すっこい重要で、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!!!!

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。

たとえば

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

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 $2 \times 2$ 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列目)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。

試しに先ほどの

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

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？

DAKARA 行列ってのは乗算順序が重要なのです。

さて、どうしてわざわざこんなややこしいものを持ち出してまでやるのかというと、理由は「**座標変換**」を効率的にできるからです。

## 7.2 アフィン変換

行列の偉いところは、行列の乗算をすることにより、それぞれの変換処理を合成することができる点です。

実際、回転処理ってのはほぼ確実に3回の変換が必要になります。

物体を原点に戻す変換(平行移動)→回転→物体を元の座標へ戻す(平行移動)

基本的に座標変換は「平行移動に始まり平行移動に終わる」のです。というわけで座標の変換が複数回必要になりますが、最終的に数万頂点になるものに対して、そんな変換をかけていては無駄が多すぎるので、事前処理として「合成行列」を作っちゃおうってわけです。

そのためにはもう一つの面倒な概念「同次座標系」って概念を軽く知しておく必要がありま  
す。いや大したことじゃない。

2次元座標系なら通常は

(x,y)じゃん？

これだと実は「平行移動」に対応できない! DA!!!

説明のために回転行列を話すけど、二乗元の場合はこういう行列になります。

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

これに対して(x,y)を行列としてかけるとこうなります。

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{pmatrix}$$

で、ご覧のように変換されるわけです。

これで回転を表すんですが、平行移動はこれで表現できないのです。残念ながら。

2行2列のままだと、どうしてもxの変換にyが絡むことになり、変換にそれぞれの変数を絡めないためにはもう一つ…1という余計な行を追加する必要があります。その一つの「余計な次元」を含む座標系を「同次座標系」と言います。

同次座標系での表現はこうなります。

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

これ、計算するとわかりますけど、変わりません。変わらない行列を「単位行列」っていうんですけど…

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

ご覧のようになにも変わりません。これは知っておきましょう。うーん。単位行列を紹介したかったわけではなくてですね…

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

こんな感じで平行移動部分に関わるとこと、回転/拡大縮小に関わる部分が住み分けられている事に留意してください。

例えば、(a,b)だけ平行移動したいとすると…

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

こうなるわけです。ここで回転、拡大縮小、平行移動を同時に扱うことができるようになりました。

さて、スマノがこれが本題ではないのだ。

最初にも書いたが、回転したい場合は「原点へ平行移動」→「回転」→「元の位置へ平行移動」が必要である。

これを合成するところなる。

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

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」「プロジェクション行列」つてのがあってそれぞれこんな感じ

$$Z = \text{Normalize}(E - P)$$

$$X = \text{Normalize}(U \times Z)$$

$$Y = Z \times X$$

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

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & \frac{-2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{right+left}{2} \\ 0 & 1 & 0 & -\frac{top+bottom}{2} \\ 0 & 0 & 1 & \frac{far+near}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

難しいですね。これはまたあとでお話しましょう。

## 7.3 3D 化実装

とにかくまずは3D化実装してみましょう。

DirectXで行列を扱うには、昔はd3dx9.hというDirectXの中に入っていたんですが、最近は「そんなもん自前で作れ」とてスタンスらしいです。でも行列系の関数自作なんてやってられないでの、そのための便利なライブラリを紹介します。

### 7.3.1 XNAMath

xnamath.hをインクルードしてください。

そうすると XMATRIX系の行列を利用することができます。

XnaMathにおいては行列は

```
XMMATRIX matrix;
```

と宣言します。

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.reference.xmmatrix\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.reference.xmmatrix(v=vs.85).aspx)

ちなみに行列の初期化にはXMMatrixIdentity()を使用します。これは「単位行列」を返すもので

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

こんな状態の行列です。こいつは右からかけようと左からかけようと計算の結果が変わらないという面白い行列です。こいつで初期化することでニュートラル状態の行列にします。

さて、3Dにおける変換行列で重要なのは大きく3つあって

- モデル行列(ワールド行列)
- ビュー行列
- プロジェクション行列

などというものがります。

それぞれ説明しますが、ワールド行列ってのは一番分かりやすく、モデルが平行移動したり拡大したり、回転したりするときに使用されます。

次にビュー行列ですがこれはカメラ変換行列とも呼ばれ、カメラの座標と向きに応じてモデルの見え方を変換する行列です。

最後にプロジェクション行列ですが、これは「遠近法」のための行列で遠くにあるものほど小さくなることを実現する行列です。「画角」などで画面の歪み方が変わってきます。

大雑把に説明するとこんな感じで、こんな感じの行列を作っていきます。

### 7.3.2 ワールド行列を作成

ひとまずはワールド行列を作りましょう。簡単です。回転と平行移動をすればいいだけです。なお、回転する場合は通常、モデルの中心を原点に持っていきます。まあ「モデルの中心」なんて今回はわからないので、回転行列だけで作ります。

回転行列を作るには XMMatrixRotationY 関数を使います。

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

ちなみに行列関係のやつは

XMMATRIX…まで書くとインテリセンスくんが色々出してくるので、英語の感じからカンで探して、使い方は MSDN を見るというやり方がおすすめです。今回は XMMatrixRotationY を使います。

```
XMMATRIX world=XMMatrixRotationY(angle);
```

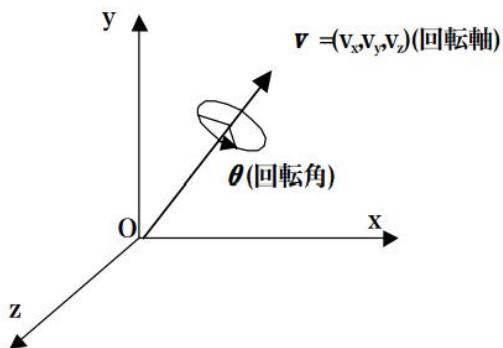
てな感じで適当な角度(ただしラジアンに限る)を渡しておいてください。

### 7.3.3 ビュー行列を作成

これはカメラをシミュレートするための行列です。

- 視点
- 注視点
- 上

の3つの要素から「視点から注視点を見る行列」を作ります。「上って何なの?ってよく聞かれるんですけど、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。」



名前は XMMatrixLookAtLH なのさ。

[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)

とりあえず適当に後ろの座標から、今おいてる三角形を見るようにしてください。上ベクトルは  $0,1,0$  でいいです。

```
XMVECTOR eye = {0,0,-3};
```

```
XMVECTOR target = {0,0,1};
```

```
XMVECTOR up = {0,1,0};
```

とにかく僕はこんな感じで設定しました。

```
XMMATRIX view = XMMatrixLookAtLH(eye, target, up);
```

意味はわかりますよね?

最後にプロジェクション行列を作ります。

#### 7.3.4 プロジェクション行列を作成

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。

コレはちょっと難しい。

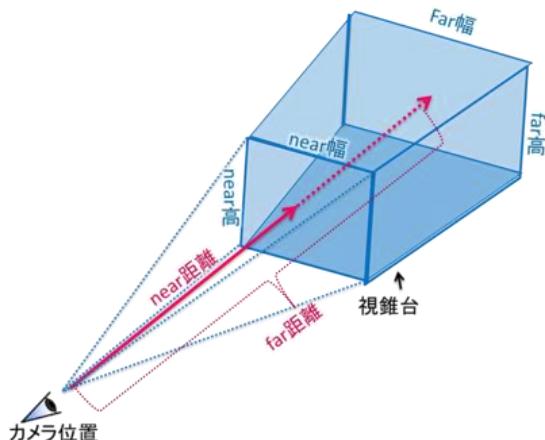
[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.matrix.xmmatrixperspectivefovih\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixperspectivefovih(v=vs.85).aspx)

第一引数の「トップダウン視野角」ってのはいわゆる「画角」と言うものです。詳しくはCGのセンターに質問してほしいんだけど、要はどれくらい距離に応じて小さくするのかって値。

「画角」っていうくらいだから角度(ただしラジアンに限る)で指定する。45°くらい設定すればいいでしょ。

アスペクト比は分かるでしょ? アス比ですよ。今回は640/480なのでそれを入れておきましょう…おっと…答えが整数にならないように注意してくれよ?

最後にニアファーだけど、これはクリッピングボリュームって言って、このボリュームを超えたら描画しないというルール設定のためのパラメータだ。



何となくわかりますかね? そこに書いてる台形の中身しか表示しないってこと。

今回はだいたい0.1f~100fまでの範囲で表示したいので、ニアを0.1ファーを100.fくらいに設定しておいてくれ。

最後に全部かける

```
XMMATRIX matrix=world*view*projection;
```

乗算の順番はワールド、ビュー、プロジェクションの順だ。これを間違うとおかしなことになるから気をつけてくれ。

さて、これで必要な行列は全部作りました。と。

じゃあどうやってこれを GPU に渡すの？

## 7.4 コンスタントバッファ(CPU 側)

そこで出てくるのがコンスタントバッファなのです。

頂点バッファは頂点を渡すためのもの。

コンスタントバッファってのは「定数」を渡すためのものです。

定数と言っても、実行時に値の変更は可能だけどね。とにかくコンスタントバッファと呼ばれるものに、さっきの値を入れてしまおう。

作り方は頂点バッファのときとほぼ同じだ。違いはアクセスフラグと Usage と BindFlags くらいだ。

当然ながら BindFlags は D3D11BIND\_CONSTANT\_BUFFER だ。

[https://msdn.microsoft.com/ja-jp/library/ee416041\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416041(v=vs.85).aspx)

今回は CPU 側からバッファの内容を書き換えまくるので CPU アクセスフラグと Usage が変更される。

Usage はダイナミックだ

[https://msdn.microsoft.com/ja-jp/library/ee416352\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416352(v=vs.85).aspx)

そして CPU アクセスは CPU\_ACCESS\_WRITE にしておこう

[https://msdn.microsoft.com/ja-jp/library/ee416074\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416074(v=vs.85).aspx)

あとはほぼいつも通りだ。

```
//行列用コンスタントバッファの作成
D3D11_BUFFER_DESC cbuffdesc= {};
cbuffdesc.Usage = D3D11_USAGE_DYNAMIC;//CPUから書き換えたらいGPUはそれを読み取る
cbuffdesc.ByteWidth = sizeof(XMMATRIX);
cbuffdesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;//コンスタントバッファとして
cbuffdesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;//CPUからの書き換え可
```

とりあえず初期値も入れておこう

```
D3D11_SUBRESOURCE_DATA initconstant = {};
initconstant.pSysMem = &matrix;
ID3D11Buffer* cbuffer = nullptr;
result = device.Device()->CreateBuffer(&cbuffdesc, &initconstant, &cbuffer);
```

こんな感じで。

リザルトがオッケーだったらコンスタントバッファをセットしよう

[https://msdn.microsoft.com/ja-jp/library/ee419762\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419762(v=vs.85).aspx)  
//コンスタントバッファのセット  
device.Context()->VSSetConstantBuffers(0, 1, &cbuffer);

今はスロットを 0 として登録してくれ

さて、これだけでは見た目に影響は出てこない。

そこでシェーダ側にも変更を加える。

## 7.5 コンスタントバッファ(GPU 側)

セットされたコンスタントバッファを使用するにはシェーダ側をいじることになる。シェーダプログラムの先頭に cbuffer cbuff{};

と書いてくれ

[https://msdn.microsoft.com/ja-jp/library/ee418283\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee418283(v=vs.85).aspx)

cbuffer はこの文字列通りに書く必要があるが、cbuff は特になんでもいい。構造体名みたいになもんだ。

cbuffer は予約語なので、かならずこう書いてくれ。

で、cbuff の中に

```
matrix mat;
```

と書こう。

この mat にさっき渡した行列が入っているぞ!!

じゃあどう使うのか? というと、座標変換の行列式を思い出してほしいのだが…

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

こんな感じだったと思う。シェーダでは掛け算は mul 関数を使用する。

つまり

```
OutputP output;
output.pos = mul(mat, pos); // 座標情報をまとめて代入
```

などと書けば、この output.pos は既に変換後の座標(つまりカメラ変換を書けられた後に 3D → 2D 変換で潰されている頂点座標)になっている。

というわけで、ともかくこの状態で実行してみよう。

実行が正常であれば回転角を変えてみて、きちんと変更されることを確認しよう。

## 7.6 コンスタントバッファの動的変更

ただ、これだとアニメーションしないので、3D 感が薄いかも知れない。

ということで、コンスタントバッファを動的に変更してみよう。まあそのために DYNAMIC の CPU\_ACCESS\_WRITE にしたんだけどね。

今回使用する関数は

Map～Unmap

だ。

[https://msdn.microsoft.com/ja-jp/library/ee419694\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419694(v=vs.85).aspx)

[https://msdn.microsoft.com/ja-jp/library/ee419753\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419753(v=vs.85).aspx)

正直面倒なのと、概念が分かりにくいので、深いところまでは説明しない予定だが、簡単に言うと、フレーム毎に既に GPU にセットしているコンスタント/バッファにアクセスして、書き換えてやるうってわけだ。

使い方はちょっとややこしい。

Map 関数を使うと、その情報はロックされる(GPU 側から書き換えることもできなきや、値の参照もできなくなる)。

だが、ロックされる代わりに CPU 側が値を書き換えることができるようになる。

```
HRESULT Map(
    ID3D11Resource *pResource, // 対象バッファ
    UINT Subresource, // インデックス番号(0 でいいでしょ)
    D3D11_MAP MapType, // 今回は書き込むし D3D11_MAP_WRITE_DISCARD を使う
    UINT MapFlags, // オプショナルらしいので今回は 0 やね
    D3D11_MAPPED_SUBRESOURCE *pMappedResource // ここに値が入る(ここの値を書き換える)
);
```

で、ロックされてるんで、書き換えた後は必ず Unmap が必要というわけだ。

```
D3D11_MAPPED_SUBRESOURCE mappedSubresource = {};
device.Context()->Map(cbuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedSubresource);
```

で、mappedSubresource にマップしたポインタが入ってるんだが、こいつをそのまま書き換えるのではなく、こいつが持っている pData を書き換える。

書き換え方は人それぞれだが、教科書的には(教科書なんてないがな)こう書いて有ることがおおい。

```
memcpy(mappedSubresource.pData, &matrix, sizeof(XMMATRIX));
```

だが、俺は memcpy が嫌いだ。どうしよう。

実はこう書いてもいい。

```
*(XMMATRIX*)mappedSubresource.pData=matrix;
```

おわかりだろうか?

あ、最後にあんまップを忘れないようにな。

```
device.Context()->Unmap(cbuffer, 0);
```

第二引数は、マップしたときのサブリソース番号なので、今回はゼロでいい。

これをを利用して、毎フレーム角度を更新できるようにしてください。

## 7.7 おまけ(もうちょっと回転をわかりやすく)

で、これでは回転しているのがわかりづらいので、クアッドポリゴンにしましょう。で、頂点が TRIANGLE\_LIST だと面倒なので TRIANGLE\_STRIP にしてください。

で、TRIANGLE\_STRIP の場合「時計回りが表」ではなく「N の字もしくはエの字が表」です。

ですから、こんな感じ。

```
Position vertices() = {  
    {-1,-1,0.0}, //左下  
    {-1,1, 0.0 }, //左上  
    {1,-1,0.0}, //右下  
    { 1, 1, 0.0 }, //右上  
};
```

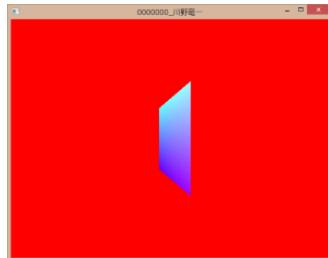
～中略～

```
context->Draw(4,0); //頂点数 4
```

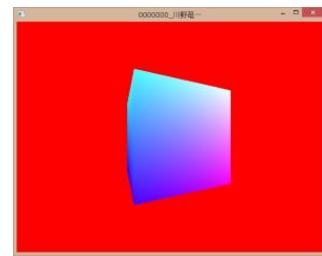
ただし、これだと背面カーリング(裏を表示しない)ために回転しているというより、なんか途中で消えちゃいますね。

なので、裏面も作りましょう。全部で 6 頂点になります。

回転しても平面がくるくるしているように作ってください。静止画だとわかりづらいですが、こんな感じですね。

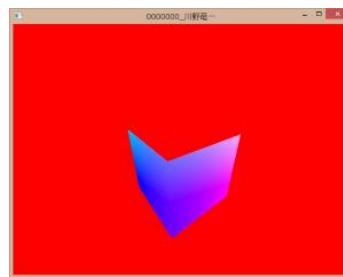


ここまでできたら、更に頂点を増やして、Z軸方向にも厚みを持たせてください。

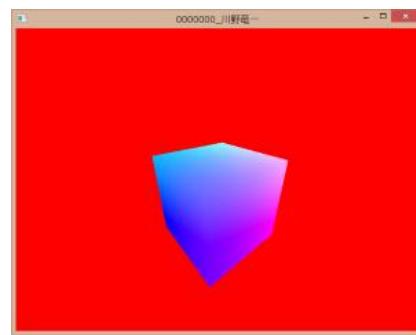


ペラ紙を一周させてこんな感じに見えるようにしてください。(静止画だとわかりづらいですが、立方体みたいになっています。)

上下の蓋がないので、ちょっと上から見ると、こういう情けない状況になっています。



かわいそうだと思ったら蓋を付けてください。Draw 命令は増えますけど。



おや?ゲーム科の様子が…

おめでとう!君は2Dプログラマから3Dプログラマに進化した!!!

## 8 PMD ファイルロード

さて、君が3Dを表示できることはわかった。

だが今のままでは立方体などしか表示できない!

# それは、ツマラナイ

やっぱり可愛い3Dモデルをちゅつちゅしたい。もちろん刀剣乱舞キャラみたいにかっこいい系でもかまわないのだが…ともかくメッシュ(3Dモデル)をロードすっべ。

ミクミクダンスのモデルデータはPMDと言って、バイナリファイルでできます。

PMDの特徴は中身が比較的シンプルで分かりやすいってのがあります。FBXとかCOLLADAとかのデータは汎用性には優れていますが、余計なデータが多いので、ゲームに向いているとは思いません。

未だにゲームに使用するデータは「シンプル」な「バイナリ」が適しているかなと思います。うっかりFBXに手をだすとたぶん…痛い目にあうと思います。

FBXはあくまでも「中間データ」くらいに思っておけばよいでしょう。

ところで「バイナリファイル」ってのは、ぱっと見、何を意味しているんか分からぬいようなデータですが、特定のツールを使用すればだいたい何を言っているのかがわかります。

### 8.1 バイナリエディタを使ってみよう

サーバーのToolのバイナリエディタから、tsxbn400をダウンロードして、解凍してください。

で、rkawano の素材の PMD の中にミクモデルデータが入っているのでそれをバイナリエディタで見てみよう

```
00000050 6D 64 00 00 80 3F 8F 89 89 B9 83 7E 83 4E 00 Pnd ?初音ミク
00010FD FD 50 6F 6C 79 4D .....PolyM
000206F 97 70 83 82 83 66 83 8B 83 66 81 5B 83 5E 81 o用モデルデータ・
0003046 8F 89 89 B9 83 7E 83 4E 20 76 65 72 2E 31 2E F初音ミク ver.1.
0004033 0A 28 95 A8 97 9D 89 89 8E 5A 91 CE 89 9E 83 3 (物理演算対応・
0005082 83 66 83 8B 29 0A 0A 83 82 83 66 83 8A 83 93 cフル) モデリン
0006083 4F 09 81 46 82 A0 82 C9 82 DC 82 B3 8E 81 0A グ : あにまさ氏
0007083 66 81 5B 83 5E 95 CF 8A B7 09 81 46 8B 9E 20 データ変換 : 京
000808F 48 90 6C 8E 81 0A 43 6F 70 79 72 69 67 68 74 秋人氏 Copyright
0009009 81 46 43 52 59 50 54 4F 4E 20 46 55 54 55 52 : CRYPTON FUTUR
000A045 20 4D 45 44 49 41 2C 20 49 4E 43 00 FD FD FD E MEDIA, INC ...
000B0FD FD ...
000C0FD FD ...
.....
```

お前は何を言ってるんだ。



おまえは何を言っているんだ

と感じるほどに何を言っているのかわかりませんが、コンピュータはこちらのほうがわかりやすいのです。

バイナリエディタはこの「コンピュータにわかることば」をそのまま16進数で表示するものです。もしくは機械の言葉を人類が分かる言葉に直訳してくれるようなもんです。

とりあえずそのまま開くと、ただ単に16進数で表示してるので  
人間がわかりやすくするために「シンボル」というのを使います。

はい、何のことかわからないので、僕が作った SYMBOL ファイルを(rkawano¥MMD¥PMD.SYMD) / バイナリエディタを解凍したフォルダに入れてください。

で、再読み込みすると

header.magic[0]	50 6D 64
header.version	3F800000
header.model_name[0]	8F 89 89 B9 83 7E 83 4E 00
header.model_name[16]	FD FD FD FD
header.comment[0]	50 6F 6C 79 4D 6F 97 70 83
header.comment[16]	81 5B 83 5E 81 46 8F 89 89
header.comment[32]	65 72 2E 31 2E 33 0A 28 95
header.comment[48]	91 CE 89 9E 83 82 83 66 83
header.comment[64]	66 83 8A 83 93 83 4F 09 81
header.comment[80]	82 B3 8E 81 0A 83 66 81 5B
header.comment[96]	81 46 8B 9E 20 8F 48 90 6C
header.comment[112]	72 69 67 68 74 09 81 46 43
header.comment[128]	46 55 54 55 52 45 20 4D 45
header.comment[144]	43 00 FD FD FD FD FD FD FD
header.comment[160]	FD FD FD FD FD FD FD FD FD
header.comment[176]	FD FD FD FD FD FD FD FD FD
header.comment[192]	FD FD FD FD FD FD FD FD FD
header.comment[208]	FD FD FD FD FD FD FD FD FD
header.comment[224]	FD FD FD FD FD FD FD FD FD
header.comment[240]	FD FD FD FD FD FD FD FD FD
vert_count	0000234C
vertex[0].pos[0]	3F95844D 418D1A37 BE9C91D1
vertex[0].normal_vec[0]	3F48E5AF BEA8474B BF068654
vertex[0].uv[0]	00000000 3F800000
vertex[0].bone_num[0]	0003 0000
vertex[0].bone_weight	64
vertex[0].edge_flag	00
vertex[1].pos[0]	3FA60419 418EDA51 BE9FB15C

まあ、ある程度意味があるデータであることがわかります。今回はここから「頂点情報」だけを抜き出し画面上に頂点を表示させたいと思います。

PMDデータフォーマットは、インターネットで探せばいくらでも出てくるので、興味がある人は探して欲しいですが、簡単に言うと、「ヘッタ部分」と「データ部分」に分かれています。

PMD: [http://blog.goo.ne.jp/torisu\\_tetosuki/e/209ad341d3ece2b1b4df24abf619dbe4](http://blog.goo.ne.jp/torisu_tetosuki/e/209ad341d3ece2b1b4df24abf619dbe4)

PMX: <https://wwwb.atwiki.jp/vpvpwiki/pages/284.html>

こまけえ話は後で解説するので、

vert\_count というところに注目してください。ここに総頂点数が書かれています。16進で書かれていますが、ミクのデータならだいたい 8000~10000 くらいでしょう。

思いの外、頂点1つあたりのデータが多いですね。でも今回注目すべきなのは pos のみ。

## 8.2 バイナリファイル(PMD)をロードするやで (°) (°)ミ

ファイルを読み出すには

- ①ファイルをオープンする
- ②ファイルをリードする
- ③ファイルをクローズする

これだけ。正直 C# も変わらないし、この流れはほとんどの言語において変わらない。P

ログラマになろうと思うなら「必ず」おさえておかなければならない。

さて、ファイルの操作はオープンから始まるわけなのだがここで

C 言語標準の fopen を使うか、Windows 標準の CreateFile を使うか迷ってしまう。fopen  
はシンプルだが、シンプル過ぎて後々面倒なことになることもある。

CreateFile 側は引数が多いけれど、色々と指定ができるのだが、そもそも殆どの学生が普通  
にファイルオープンの仕組みすら理解してそうにないので fopen の方を使おう。

ちなみに fopen 系を使うときは

```
#include<cstdio>
```

をインクルードしておきましょう。

ここを乗り越えられたやつだけ後から CreateFile, ReadFile で読み込みすればいい…あと  
でオマケ的にそっちの話はします。とりあえず fopen で行きましょう。

<http://www.orchid.co.jp/computer/cschoo/CREF/fopen.html>

- ①ファイルオープン

基本的には、

```
FILE*fp=fopen("ファイル名","rb");
```

って感じでオープンします。ちなみに rbってのはバイナリで、リードするよっていう意味です。

## ②ファイルリード

実際にファイルからデータを取得します。

ちなみに関数は fread です。

<http://www.orchid.co.jp/computer/cschoo/CREF/fread.html>

ちょっと今回はいきなり「頂点数」にアクセスしたいので

fseek って関数を使用します。

<http://www.orchid.co.jp/computer/cschoo/CREF/fseek.html>

これはニコニコ動画のシークバーみたいのはもんで、見たい情報のところにジャンプしたい時に使用します。

### 8.2.1 ヘッダファイルをロード

とりあえずネタバレしとくと頂点データがあるところは 283 バイト目なので、そこまでジャンプします。(ゆーても 283 はマジックナンバーなので、あとできちんと解説します。)

```
fseek(fp, 283, SEEK_SET);
```

ちなみに SEEK\_SET はファイルの先頭から、って意味です。

で、なんで 283 かって言うと、PMD ファイルにおいては、ヘッダサイズは固定で、それが 283 なんですね。

内訳は 3 バイトがシグネチャ(署名)で、次の 4 バイトがバージョン。次の 20 バイトがファイル名、その次の 256 バイトがコメント。これは PMD のファイルフォーマットルールで決まってるんで、仕方ないです。

$3+5+20+256=283$  で 283 なのです。ここがマジックナンバーなのが気持ち悪いので sizeof(PMDHeader) などと書きたいところですが、ここで面倒くさい問題が発生します。

### 8.2.2 4 バイトアライメント問題

そう。これはバイナリをいじってると遭遇する可能性がある問題で、構造体の要素のスタートが 4 の倍数バイトからスタートってのがあります。

さっきの例だと、先頭に 3 バイトあるやん? こいつが悪さして…

## 構造体アライメント

って問題が発生する。これを考慮しないとデータがガソリ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りました、頂点数が〇になったりする。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るべきか…処理系依存といるべきか…まあ難しい話なんですねー。

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



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

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

今回みたいに3バイトデータが混じっていると、こういうアクセスになる。



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

なので、コンパイラがご親切にも(おせつかれにも？)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{  
  
    //char signature(3); //シグネチャー"Pmd"  
  
    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(signature, 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。特にメモリケチってた者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。

最後にクローズするのがルールですが、まだまだ「頂点数」を読み込んだだけですから、ここからが本番です YO。

### 8.3 頂点情報をロード

さあ、いよいよ頂点情報をロードします。

頂点数がわかってるわけですから、あとは「頂点 1 つあたりのバイト数」が分かれば何バイト読み込めばいいのか…分かりますね？

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/5a1b1be2fb10b7838dfcbb0d010389707](http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b1be2fb10b7838dfcbb0d010389707)

を見てください。

•頂点リスト

```
DWORD vert_count; // 頂点数  
t_vertex vertex(vert_count); // 頂点データ(38Bytes/頂点)
```

と書いてます。

t\_vertex の中身は後で話すとして、ともかく頂点データ1つあたり 38 バイトです。また…アライメント問題起こすようなバイト数にしゃがって…ぐぬぬ。



頂点数を「決め打ち」は嫌ですが…仕方ない。const int vertex\_size=38; とでも定義しておいてください。いや、ヘッダー同様に定義できなくもないんですが、今は面倒なので、決め打ちにしつきましょう。

ともかく今は「アライメント」のせいで面倒なことになってるっていうのを認識して頂点情報を読み込みましょう。

さて、次に頂点情報を入れる「箱」の確保です。きれいに整理されている「箱」でもいいですが、早くミクちゃん見たいでしょ？

ということで手抜きします。モチベーションのためには手抜きもやむなしです。要はすべての頂点のバイト数が入る箱が必要で、そのサイズは頂点数\*38 であることが分かっている。

malloc とか new とか嫌いなので vector<char>を使います。中でやってるこたあ一緒です。

```
std::vector<char> vertices(頂点数*38);
```

これで箱の準備は完了です。あ、vectorを使うときは#include<vector>を忘れないように。

じゃあロードですね。

```
fread(&vertices[0],vertices.size(),1,fp);
```

とかやってロードします。ちなみにこの場合の vertices は単に受け取った『データの塊』に過ぎないんで、勘違いしないように。

あと、知らない人もいると思うのですが、ベクタの戻り値アドレスへのアクセスは上記のように

`&変数名[0]`

でアクセスします。要は(0)番目の要素に&をつけて、配列先頭アドレスとして返してるだけです。

ともかく頂点のロードができましたのであとは表示です。

他にも vertices の配列名としてアクセスしてたところは &vertices[0] でアクセスするよう に、また sizeof(vertices) でサイズ指定してたところは vertices.size() にしてください (bytewidth とか subresource の部分とかね)

あと、データのストライド変わってるんで 38 にしといてください。

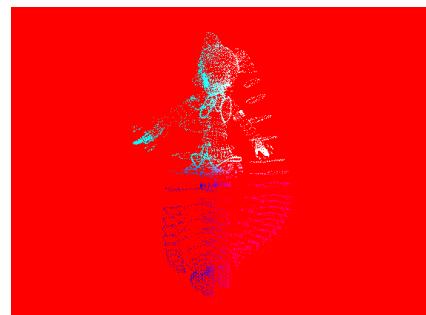
まだ頂点情報しかないので、「面表示モード」から「頂点表示モード」に変えてやらねばおかしなことになります。

そこでプリミティブポリジを『POINTLIST』に変更します。

D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST → D3D11\_PRIMITIVE\_TOPOLOGY\_POINTLIST

これな。

うまいこといくとこうなります。



変に色をつけてるからわかりづらいですかね…。白地に黒にするとこんな感じです。



何気にミクさんぽいのが表示されているのがわかるだろう?これがミクさんの頂点だ。しつかり拝んでおきましょう。

## 8.4 なんでこれだけで表示されるのん?

疑問に思っていただければ狙い通りなんんですけどね。

今回、こんなに簡単に頂点表示ができるとは思ってなかつたひとが結構多いのではないかと思ひます。ポイントは

- 頂点情報の構造
- ストライド
- レイアウト

にあります。

今一度

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/5a1b1be2f610b7838dfcbb010389707](http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b1be2f610b7838dfcbb010389707)

見てください。

構造体にするとこんな感じですね。

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

最後の 2 バイトのせいで 38byte などという中途半端なバイト数になってるのはこの際置いておきます。

この章の最初にも書きましたが、今回欲しいのは頂点情報だけです。

つまり最初の sizeof(float)\*3 バイトってのは、実は最初に三角ポリゴンを表示したのと変わらないんですね。頂点座標そのものは同じデータタイプなんです。そしてそれが頂点データの先頭に来ている。ここまではいいかな? 本当にいいかな?

そして次に考るべきことは GPU に頂点を渡すときに情報として渡した「ストライド」です。これは何を意味しているのがわかりますか? 「1 頂点あたりのサイズ」というより、もっと正確に言うと「次の頂点までのバイト数」って事なんだ。

つまり、「頂点、座標情報」+38バイトニ「次の頂点、座標情報」になっているわけ。

で、レイアウトは今

```
{"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0},
```

こうなってるんだけど、これは送られてきた頂点1つあたりのデータの塊からどうやって情報を解読するか…データの塊をどう見るかってことなんだ。

そして、最初のFLOATみっつを POSITIONとして見よう(後のことば知らない→次は38バイト先を見よう)

となって、見事頂点情報だけを抽出できているわけなのだ。

うーん。今まででは「データの塊」といわゆる「塊データ」って概念というか意識がなかったからかなり難しく感じるかもしれない。まあでもコンピュータってのは実際文字で何かを理解してるわけじゃなくて、データの塊と、その解釈方法しかないんだよね。

これに慣れるには何か少しずつ人間らしさを捨てていかなければならぬのかかもしれない。



で、他のデータが要らないものか?って言うとそうじゃなくて、今はまだ不要なので無視しているだけです。中には座標、法線、UV、ボーン、影響度、エッジフラグというデータがありますが、座標以外はちょっと説明が必要なので後回しにしてるだけです。

先にやるべきことがあるんです。次行ってみよう!次!!!

## 8.5 インデックス情報とは

先にやるべきことは何か?というと「インデックス情報」のロードが必要なんです。「なに? イン…なんとか?」



初めて聞く人も多いかもしれないけど、このインデックスと言うのは「面表示」をするために必要なのです。どういう事がというと、頂点だけでは面が作れないのです。

「馬鹿言うな!!!さっき立方体作ったじゃねーか!!!お前もう忘れちまったのかよお!!!」

残念ながらこのミクさんデータは先程も言いましたが「頂点情報のみ」なのです。頂点はいくら集まても面ではないのです。…そうです。

### 「どう頂点を繋げるか情報」

が欠けているのです。…分かんない?んじゃあさっきの POINTLIST を TRIANGLELIST にしてみれば分かるよ。



インデックスさん情報がないと、ただ単に頂点が登録された順番に三角形を作ろうとします。でもご覧のようにこれはミクさんではなく「ミクさんだった何か」になってしまいます。

というわけで、頂点をつなぐ情報…インデックス情報をロードして、せめて「影絵」にしてあげましょう。

## 8.6 インデックスさんのロード

例によって、ここを見ましょう。

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/7cebae143cb6b9bae38ebbefc228c05b](http://blog.goo.ne.jp/torisu_tetosuki/e/7cebae143cb6b9bae38ebbefc228c05b)

どうやら頂点情報をロードしきったところに『face\_vert\_count』があり、その後にインデックス情報があるようです。

ちなみに face\_vert\_count はインデックスの数です。インデックスとはこの場合、頂点一つ一つの番号を指定するためのものです。番号とはただ単に先頭から順の「通し番号」です。

そしてインデックスは WORD(つまり unsigned short)で宣言されています。2 バイトです。

つまるところ、PMD では 65535 頂点以上は扱えない計算です。

別にいいですけど。

ともかくロードしましょう。

頂点ロードはすでに終わっているので、ファイルリードカーソルはインデックス数の所に来ているはずです。

```
fread(&indexCount, sizeof(indexCount), fp);
```

とでもしてあげればインデックス数は得られます。

後はこのインデックス数だけインデックスを読み取ればいいわけです。

やり方は…分かりますよね？

インデックスをロードできたらあとは頂点バッファと同様にインデックスバッファを作つて、セットして、描画するだけです。

## 8.7 インデックスを使った面描画

うまく行けば



という風に出ますが、まだ準備が足りません。ほぼ頂点バッファの作成と同じなのでバッファ作成のところは D3D11\_BIND\_INDEX\_BUFFER 使えとしか言いませんが、セットがちょっと面倒。

[https://msdn.microsoft.com/ja-jp/library/ee419686\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419686(v=vs.85).aspx)

説明によると…

インデックス バッファー内のデータのフォーマットを指定する DXGI\_FORMAT です。インデックス バッファー データに使用可能なフォーマットは 16 ビット (DXGI\_FORMAT\_R16\_UINT) および 32 ビット (DXGI\_FORMAT\_R32\_UINT) の整数に限られます。

今回インデックスはさつき見たように WORD(unsigned short) の 2 バイトなので R16 の方を使いましょう。

さて、これでインデックスのセットまでできました。

あとは描画命令ですが、

Draw 命令を DrawIndexed に変更しましょう。

[https://msdn.microsoft.com/ja-jp/library/ee419591\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419591(v=vs.85).aspx)

第一引数にインデックス数。第二引数にインデックスオフセット。第三引数に頂点のオフセットを指定すればいいです。

DrawIndexed(インデックス数, 0, 0) でいいですよ。

それでうまく行けば影絵みた感じになるはずです。

## 9 ここで一旦休憩

話をしよう。

あれは今から22年前…

### 9.1 STL のお話

はい、ここで STL を知らない子どもたちのために STL の説明をしようと思います。

STL とは Standard Template Library の略で、C++ の「テンプレート」の機能で便利なライブラリを作った人がいて、それが 1994 年に正式に C++ の標準として組み込まれた。それが STL だ。

C++ を勉強するうえでは STL の理解は必須といつてもいい。そういう奴です。

例えば、C 言語を勉強する上で printf とか fscanf とか fopen とか sin とか cos とかの標準関数は必ず通る道でしょ？

そういうのと同じと思っておけばいいです。

ちなみに STL に関しては

<http://episteme.wankuma.com/stlprog/>

が詳しいですが、ちょっと古いとは思います。ちなみに Wikipedia もあり

[https://ja.wikipedia.org/wiki/Standard\\_Template\\_Library](https://ja.wikipedia.org/wiki/Standard_Template_Library)

に書かれているとおりです。

で、基本的なコンテナ…うーん、表現が難しいんですが他のオブジェクトを格納するオブジェクトのことだと思っておいてください。だから配列もある意味コンテナなんんですけど、そういう意味での「コンテナ」ってのが STL には沢山用意されています。

基本コンテナ

- vector: 配列のように扱える
- map: 連想配列のように扱える(キー重複なし)
- list: リストのように扱える
- stack: スタックのように扱える
- queue: キューのように扱える
- deque: 双方向キューのように扱える

- set:ルールに従って並ぶ(値重複なし)
- multiset:ルールに従って並ぶ(値重複あり)
- multimap:連想配列のように扱える(キー重複あり)

基本的なやつだけでこれだけある。array やら priority\_queue などの基本的じゃないやつも含めると大変なので、これくらいにしておく。

でもゲームでよく使うコンテナは

- vector…配列のように扱えるため、動的配列の代わりとして多用される
- map…キーと値のペアを格納し「連想配列」のように扱えるため、リソース管理などに使われる。

くらいだろう。

ではここでは vector と map について解説しよう。

#### 9.1.1 vector について

vector ってのは動的配列を実装するものです。動的配列を作ろうとするとき、一部の偏屈な人は vector を使わずに頑張って

```
int* a=new int[要素数];
```

などと宣言します。これも間違いじゃないんだけど、いろんな問題を含んでいる。まず、動的確保なので malloc と同様に「解放」を明示的に行わなければならないんだが、その解放も面倒で

```
delete() a;
```

などと書かなければならぬ。非常に面倒。なので vector を積極的に使おう。

まず、vector を使うには、vector をインクルードします。

```
#include<vector>
```

これで準備完了であります。

で、こいつを使うには std のひとつだよって意味をこめて、std:: を頭に付ける必要があります。

なので、例えば int のベクタ配列を作りたければ

```
std::vector<int> vectorhairetu;
```

なんていう風に宣言します。一般化して書くと

```
std::vector<配列にしたい型> 変数名;
```

となります。こいつは、std::vector<配列にしたい型> がまたひとつの型として扱われますので、要は、構造体みたいな扱いができるってことですね。つまり関数の引数として渡したい場合は

```
void Sample(std::vector<Aho>& ahos ){  
    // うほー  
}
```

って言う風に書くことができます。ここでのポイントは &。なんで & をついているのかというと、前にも言ったように、基本的に C++ の代入は「コピー」であって「参照」ではない話をしました。

つまり、関数の引数として、

```
void Sample(std::vector<Aho> ahos )
```

という定義をした場合、引数に入れた際に、ベクタ配列のコピーが発生する。コピーって結局すべての要素のコピーなわけ。つまり、こいつの数が 10000とかあった場合、非常に処理時間が勿体無いことになる。

なので、vector を引数として渡す際には & をつけることをおすすめします。

さて、このベクタ配列がすぐれものなのは、ホントに配列っぽく使えることですね。難しい言い方をすると

「一連の要素が連續してメモリ上に配置されることが保証されている」

わけです。

なお、

```
vector<int> vec(要素の数);
```

とでも宣言してやって、

例えば、先頭の要素にアクセスしたければ

```
vec[0]=56;
```

などというふうに、配列でやったような感じでアクセスできる。これは実は、C++の演算子オーバーロードのおかげなのだ。いわゆる添字演算子オーバーロードである。

しかし、ここでそれ言ってもわかる人あまりいない。それ困る。やらない。俺、定義だけ書く

```
int& operator[](int i){ return i 番目の要素; }
```

こんな感じ。分かる人だけわかればいい。分かりたい人は自分で勉強するね。

じゃあよくある配列の先頭のアドレスを返すってやつはどうやるのか?

それは

`&vec[0]`

とやってあげる。これが、ベクタ配列の先頭のアドレスとなる。あくまでも0番目の要素のアドレスを返してあげることに注意してください。

ここでよくやる勘違いは

`vec`

とか

`&vec`

なんですが、これではダメなのです。そもそも型が違うのでコンパイラから当たり前に怒られますけどよく意味を考えましょう。

`vec` はベクタ型のオブジェクトそのもの。`&vec` はベクタ型のオブジェクトのアドレス。`&vec[0]` はベクタ型の0番要素のアドレスを表しています。

大雑把に言うと、`vector` 型ってのはこんな構造なんです。



あくまでも大雑把な話な?

例えば`&vec` ってやると、先頭の要素ではなく `vector` ヘッダのアドレスになっちまう。そこで0番目の要素にあえて`&`をつけて先頭アドレスからアクセスさせているわけなんだ。

ベクタ配列の大きさですが、

宣言時に

`vector<型> 变数名(配列の大きさ);`

って、指定する方法と、

`vector<型> 变数名;`

`変数名.resize(配列の大きさ);`

って指定する方法があります。どっちでも、お好きな方で…

ちなみに配列要素数は

`変数名.size();`

でいいともとてこれる。便利でしょ？

次に `map` の話をしよう

### 9.1.2 `map` について

`map` についてですが、これは何なのかといふと「連想配列」です。

そもそも「連想配列」という言葉がわからないかもしれません、簡単に説明すると…通常の配列の場合、要素を特定するには整数のインデックスを使用しますよね？

`a[0]=1;`

`a[1]=5;`

`a[2]=7;`

ね？これは分かるよね？でもたまにこの「インデックスに文字列とか使えへんかなー？」って思ったことないですか？

```
a("ぬこ")=10;
```

```
a("イツヌ")=5;
```

```
a("うさぎ")=1024;
```

なんていう風に…それができるのが std::map なわけです。

ちなみに、配列の中に入っている文字列のことを「キー」と言い、そのキーでアクセスできる中身を「値」と言います。

↑の例だと「ぬこ」がキーで「10」が値ですね。何となくお分かりいただけただよ？

まだそういう需要は無いのかもしれないけどね。よくやるのはリソース管理で「ファイルパス」を「キー」とし、リソース本体(データの塊)へのポインタを「値」としておいて、同じファイルパスでロードが発生しそうになつたら、既にロード済みのリソースへのポインタを返すなんていうテクニックがあつたりします。

例えば Load 関数があるとして、雑魚キャラをロードするとします。

```
for(ザコ s){  
    ザコ s.mem=Load(ザコ s(i).path);  
}
```

などという事をやってた場合、同じリソースをロードしようとすると無駄なロードが発生します。なので予め

```
std::map<std::string,char*> _enemyFilePathMap;
```

なんてのをクラスのメンバ変数として持つておいて、

```

for(ザコ s){
    if(_enemyFilePathMap.find(ザコ s(i).path)==_enemyFilePathMap.end()){
        ザコ s.mem=Load(ザコ s(i).path);
        _enemyFilePathMap(ザコ s(i).path)= ザコ s.mem;
    }else{
        ザコ s.mem=_enemyFilePathMap(ザコ s(i).path);
    }
}

```

なんていうふうにやれば、同じやつを二度ロードするコストは軽減できるわけです。なお、ハードウェア的に、時間がかかる順に

CD/DVD→HDD→SSD->>>メモリ→L2 キャッシュ→L1 キャッシュ

なので、ファイルへのアクセスは極力減らすべきである。どんなにメモリ効率を上げたとしてもハードディスクへのアクセスが多ければそれで台無しになる。

こんなもんかな。

## 9.2 コーディング(規約?)について

ぶっちゃけた話、僕個人的にはコーディング規約の話をするのはあまり好きではないです。

なんとかというと、C++の…特にゲーム系のコーディングスタイルの規約なんて、

# 「組織によってまちまち」

だからです。

きれいに書くというより「汚くならない」ように「無駄に遅くならない」ように注意しながら書く。あと「一貫性を持って書く」

これが基本方針ではあります。ここに関してはどの組織も全会一致だと思います。ちなみになぜ「組織によってまちまち」なのかというと…ここに Google コーディング規約があります。

<http://www.textdrop.net/google-styleguide-ja/cppguide.xml>

天下の Google 大先生のコーディング規約なのだから、真似しておいて間違いはないだろう? と思うかもしれません…わかり易い例として「インデント」で検索してみてください。

「スペースだけを使い、インデントはスペース 2 つにしてください。」

インデントにはスペースを使います。コードにはタブを使ってはいけません。タブキーを押したときにはスペースが入力されるようエディタを設定しておきましょう。」

ちなみにこれには Google なりの理由があるのですが、一般的な「分かりにくく」コードの例としては「ネストが深い」というのがあります。

この「マリアナ海溝ネスト」を防ぐために僕などはタブ 8 で非スペースにしてます。

で、先程話した Google なりの理由としては Google の聴講生の検索エンジンの中では  $12 \times 12$  行列とかが計算で使用されたりするらしいです。このため for 文のネストだけで必然的にこんでもないことになるらしいです。

そのような事情であればネストがちょっと深くなつたくらいでコードが画面外に出ることは避けておきたい…そういう意味で出てきた規約だと思います。

ちなみにマイクロソフト大先生のコーディング規約もそれなりにクセがあります。一時期前はハンガリアン記法を好んで使ってたくらいだしね(かなり前から時代遅れ)

なので最初に言っておきますが、「どこかのコーディング規約が最高に汎用性があって、生産性も向上する」などということはありませんから!!! 残念!!!!!! 夢を壊してごめんねー!!!!

## 9.3 コメントルール

コーディング規約の一つではありますが、コメントのルールについて軽く触れておきましょう。

1. 書くべき所には書くこと
2. 嘘は絶対に書かないこと
3. 書く時はコードの意味ではなく意図がわかりやすくなるように書くこと
4. 意味のないことを書かないこと
5. 関数の使用法はヘッダ側に書くこと
6. 見て分かることはいちいちコメントしないこと
7. 分かりにくいうちにコードをコメントで誤魔化す前に分かりやすくできるかを考えよ

### 9.3.1 コメント基本(書くべきこと)

「書くべき所」とはどこかというと、

- 関数の使用法、
- クラス/構造体の説明
- コードを読む際の注意点
- コードの意図を書こう(自明な時は必要ない)

ですね。上2つは言わずもがな。…といふか、このコメントがなかなか書けない場合、その人の文章能力が著しく低いか、もしくは「関数、クラス/構造体の作り方が不適切」ってのがあります。

関数の作り方の鉄則として、「1関数1機能」ってのがあります。要は複数の機能をもたせんなって事。ちょっと極端な例だと

Get なんとか()

Set なんとか()

って関数をまとめて

GetSetなんとか()

って作るようなもんです。YA ME RO!!!

こういうわかりやすい例なら兎も角、初心者のうちは分からぬまま色々な機能をもたせちゃうものです。そうなるとコメントしづらくなる。そう言った「設計を見直す」ために説明コメントを書くクセを付けましょう。

例えば関数コメントならこんな感じ

```
//点を打つ  
//x 打つX座標  
//y 打つY座標  
//color 点の色  
DrawPoint(int x, int y,unsigned int color);
```

こんな感じで。ちなみにDoxygenコメントという、ドキュメント生成ツールではもう少しルールがあって、ドキュメント生成ツールに出力したい行は///コメントにします。照れてんじゃねーよ!!!

```
///点を打つ  
///x 打つX座標  
///y 打つY座標  
///color 点の色  
DrawPoint(int x, int y,unsigned int color);
```

ここではDoxygenについては触れませんが、ソーソーのがあるってのは知つておきましょ  
う。話はそれましたが、関数とクラス/構造体の説明を書く習慣をつけましょう。

あ、PMDLoaderとかの場合は

///ファイルパスをもとにクライアントにPMDMeshを提供する

```
class PMDLoader{  
  
    ~中略~  
  
};
```

あとはコードを読む際の注意点を書いとくと良いと思います。

例えば…これは[エキサドライブ](#)のHPに書いてあった例ですけど

```
1. switch( mode ) {  
2.     case MODE_INITIALIZE:  
3.         initialize();  
4.  
5.     case MODE_UPDATE:  
6.         update();  
7.         break;  
8. }
```

最初のケースの中、initialize(); の後にbreakが入っていないのはバグじゃないか?と思いま  
すよね。でも以下のように一文入っているだけで、

```
1. switch( mode ) {  
2.     case MODE_INITIALIZE:  
3.         initialize();  
4.         // breakせずに引き続き更新処理を行います  
5.  
6.     case MODE_UPDATE:  
7.         update();  
8.         break;
```

ああ、これは意図された処理なんだ、と理解できます。

ソースコードだけではよく分からぬ微妙なニュアンスも残せるわけです。

とか、そういう部分ね。注意書きしどかないと「なんや、ここ break 忘れてるやんけ。足しといたろ」なんて思われちゃうわけです。アカンですよね？そういう注意書きは必要です。

### 9.3.2 嘘コメント書くな

次に、嘘コメントは絶対に書かないことを約束してください。

うーん。

「わざわざそんなことしねーよ」と思ふ人もいますが、現実にあるわけです。

原因としては

1. 単なる悪意
2. 酔ってる/寝ぼけてる
3. コード/仕様の理解不足
4. コードを書き換えた結果、結果として「嘘」になってしまった。

などがあります。まあ悪意のある人には退場してもらうとして、コーディングする時には意識明瞭であることを確認してコーディングしましょう。

で、現実的によくあるパターンが3番と4番ね。

コードや仕様についての理解が浅い場合、今の自分の理解では正しいと思っていても実は間違っているって事があります。

可能な限り MSDN のリファレンス等で裏を取って正確なコメントを心掛けましょう。といってもこれはある程度は仕方ないので、自分の理解が上がって間違っていることに気づいたらコメントも修正してください。自信がない時は MSDN で参照した URL をコメントに書いておいて「この仕様によりこのように記載しました」と書いておきましょう。

最後に長い期間コーディングをしていると、クラス、関数を仕様変更することがあります。あとは中の細かい挙動などは頻繁に変わるでしょう。

そこで実装の挙動を変更したにも関わらずコメントをそのままにしていれば、それは「嘘コメント」になります。

### 9.3.3 まとめ

で、ここで一つアドバイス。

「コメントの量が多すぎると、実装変更したときに変更すべきコメントが増え、面倒くさい。結果として嘘コメントが増える。つまりクソコードになる。」

コメント書けばいいってもんじゃねえんだよ。もう一つアドバイス

「コメントには挙動そのものを書くよりも、そのコードの『意図』が分かるように書こう。」

DirectXに慣れないいうちは CreateBuffer の挙動について書くのも良いけど、慣れてきたらそういうのはいらない。「何故そこで CreateBuffer を使うのか」が分かるようにしよう。

### 9.3.4 public メンバ関数の仕様はヘッダ側に書こう

どういう事がというと、あるクラスの public メンバ(特にメンバ関数)は誰に対して公開してるんでしょう? 誰が知つておくべきことなんでしょう?

そう…クライアント側(呼び出し側/使用者)ですよね?

それならばコメントはヘッダ側に書くべきです。『え? cpp 側でもいいじゃん?』って思った人はよく考えてください。

ヘッダ側と cpp 側は、一般的にはどちらのほうが行数が多いか?

そうだね。実装を書いている cpp 側のほうが行数が多いけだ。ということはコードの複雑性(情報エントロピー)が大きいといふわけだ。



簡単に言うと cpp 側のコメントを見るのは面倒くさい(つまり時間がかかる)わけだ。出来る限り利用者は cpp を見なくても良いようにするのが親切というものだろう? というわけで public メンバ(特にメンバ関数)の説明は基本的にはヘッダに書くものだ。

もう一つの理由を言うと、例えば PMDLoader をライブラリ化したいとする。そうなった場合、cpp ファイルのコードは見えなくなる→当然 cpp に書いたコードもなくなる。つまり説明が見えなくなるわけだ。

『ヘッダと cpp 両方に同じコメントを書いてはだめなの?』という意見もあるが、個人的には反対だ。何故かというと同じコメントを書けば、修正するときに2倍の労力がかかる。もしくは cpp/h のどちらかの修正を忘れる→カオス。だから基本的にメンバ関数の説明コメントはヘッダ側に書くべきだ。

### 9.3.5 見て分ることはいちいちコーディングしない 例えばアホな例なら

```
for(int i=0;i<20;++i)//ループの中を 20 回繰り返す  
printf("HelloWorld");//"HelloWorld"と標準出力に出力する  
などだ。アホでしょ？printf の意味も知らない1年生ならまあ書いてもいいけど、こんなコードを見ると僕なんか
```



と言いたくなる。やめろ。文字数が増えるということは前述のエントロピーが増えるってことだ。君らも文字がぎっつり書かれてる小説より、文字があんまりかかれてない小説のほうが「わかりやすい」でしょ？そういうことだ。

先にも書いたが『意図』を書こう。↑の例なら 20 回繰り返す理由はなんだ？そこで標準出力する理由はなんだ？

ていうかこんなのはコメント書がなくても良い。うざい。

### 9.3.6 細けえコメントを書くよりコードそのものをわかりやすくしよう

昨年の卒業生にいたんですけど、ゲーム会社への提出作品で、こういうコードを書いてた人がいました。

```
switch(charType){  
    case 0://スライム  
        (中略)  
    break;
```

```
case 1://コウモリ  
    (中略)  
    break;  
  
case 2://キメラ  
    (中略)  
    break;  
  
case 3://ドラゴン  
    (中略)  
    break;  
  
default://どれにも当てはまらなかつたら…  
    (中略)  
}
```

ここまで読み進めた人ならわかると思いますが、色々問題があります。でもここではコメントされてる部分に絞って考えましょう。

さて、case文というのは、switch文のカッコの中身を判別してジャンプする機能です。  
まず、0,1,2,3という直値を誤魔化すためにコメントを入れて分かりやすくしようとしている。僕は怒ったぞ



「おどりや!!!4年にもなって enum の使い方も知らんのかボケエ!!!」

switch 文の時は、本当に純粹な数値が必要な場合以外は enum を使いましょう。さっきのを書き換えるならば

```
enum EnemyType{  
    et_slime,//スライム  
    et_bat,//コウモリ  
    et_chimera,//キメラ  
    et_dragon//ドラゴン  
};  
  
switch(charType){  
    case et_slime://スライム  
        (中略)  
        break;  
    case et_bat://コウモリ  
        (中略)  
        break;  
    case et_chimera://キメラ  
        (中略)  
        break;  
    case et_dragon://ドラゴン  
        (中略)  
        break;  
    default://どれにも当てはまらなかつたら…  
        (中略)  
}
```

基本的にはこれで良いんだけど、ちょっとコメントの話からは外れちゃうんだが、話しておこう。

switch文はその特性のため、行数は増えるしネストは深くなる。

そこでcaseの中身を関数化してしまいましょう(第一段階)

et\_slime://スライム

```
    return CreateSlime();
```

et\_bat://コウモリ

```
    return CreateBat();
```

et\_chimera://キメラ

```
    return CreateChimera();
```

et\_dragon://ドラゴン

```
    return CreateDragon();
```

僕のようなキチガイプログラマはこれすら…むむつてなります(関数ポインタを使えばネストしなくて良いんじゃね?)が、これ以上やっても可読性はそこまで上がらないので、そこまでやらないでいいです。

```
Func_t arrayOfCreateEnemy[]={CreateSlime,CreateBat,CreateChimera,CreateDragon};
```

```
return arrayOfCreateEnemy(enemyType);
```

↑僕がやりたがるキチガイコードです。ぱっと見わからぬでしょ?流石にここまでやらなくてOK。

コメントに関しては、こんなもんかね。

コーディング規約よりも先に知つとくべきなのは「クソコード」にならないこと。というわけで「クソコード」について学びましょう。

## 9.4 クソコード

クソコードとは何か…

クソコードってのは

「**俺様を怒らせてしまうようなコード**」の事である。



1. 読めないコード
2. 要領の悪いコード
3. 意図がわからないコード

と、

<http://www.slideshare.net/rootmoon/ss-38278479>

ここのサイトに書いてました。ともかく俺様が怒るってことは、そういうソースコードを見る企業の人も激おこってことよ。つまり合格しない…まあ実際はやんわりと落とすんだけどね。まあどんなコードがプログラマンに怒りを齎すのかはここも見ておいたほうがいいでしょ。

<http://www.pro.or.jp/~fuji/mybooks/cdiag/index.html#mokuji10>

こんなコードを書いていると

「コイツとは一緒に仕事したくなれんなあ」って思われるわけ。もちろん「なんか動きません」とか言って見せられたコードが「クソコード」の場合、もう見る気しません…そうですね。

僕の前の会社で実際にあった話ですけど、ネットワーク周り担当してた奴のコードが酷くて、まあメンテに困ってたわけですわ。そこで耐えかねた上司(テクニカルディレクター)がそいつ

が作ったコードを全消しして、「こんなものはゴミだ!!」とたった一行のコメントにしてしまったという…。もちろん代替コードはその上司自ら書きました。

結局コードの量は減り、わかりやすくなり、更には処理が速くなりました。それくらい良いコードと糞コードには差があるんですねー。そして皆さんにはできるかぎりクソコードにならないように気をつけていただきたい。

ちなみに「クソコードbingo」なるものがあり

識別子 スペルミス	エラー 揉み消し	乱調 インデント	トトロジー コメント	マジック ナンバー
巨大共通 ライブラリ	最長不倒 関数	マリアナ 海溝 ネスト	中括弧 省略汗	通らない コードブロック
多重継承	大富豪 プログラ ミング	名実 不一致 関数	一人二役 関数	ミックスイン
遺棄 ライブラリ 依存	複数言語 混在実装	ステート 依存	テスト困難	実行時 警告洪水
グローバル 変数	型キャスト	量産 コピペ 実装	デバッグ コード残骸	永久不滅 TODO

まあここに書いてあるものが必ずしも悪いわけじゃないんですけど、とにかく一つ一つ説明するのもんどうせーけどこんな風に「禁じ手」みたいにははあるわけ。

でも赤枠で囲んでいるのは「絶対悪」です。真面目にコーディングしてたらまずお目にかかることはないんですが…同僚がこれをやった瞬間に殺意が湧くわけです。

とりあえず俺の独自のコーディングルールを書くと

- 原則として1関数が100行を超えない
- 原則としてグローバル変数は使用しない(シングルトンも必要最小限)
- マジックナンバーは絶対避ける

- コメントアウトしたコードまず使わないから消してしまおう(人力検索を阻害する)
- 引数では可能な限りポインタでなく「参照」を用いる
- タブ幅は8(好みの問題←深いネストを避けるため)
- 生成と破棄がワンセットのものはクラスを利用する
- クラスのメンバはは可能な限り private または protected にする
- クラス名、変数名、関数名は可能な限り英語を用いる
- クラス名や構造体の名前は先頭大文字で、単語ごとに大文字にする
- ビットシフトを多用しない!(わかりづらい!しコンパイラの最適化を阻害する)
- 変数名の先頭は小文字。先頭以降は単語ごとに大文字にする。
- できるだけ変数名に短縮形は使わない!(ただしループ変数は除く)
- #define 系はすべて大文字
- #define はできるだけ使わずに const 定数を使う
- #define マクロはできるだけ使わずにテンプレートを使う
- テンプレートを使いすぎない
- なるべく可能な限り new~delete を使わないよう工夫する
- なるべくポインタでなく参照を使用する
- STL を使える場面では使用する
- 組み込み関数を使用する前には穴が空くほどリファレンスを読む
- メンバ変数の先頭に\_(アンダーバー)を書いて宣言する(好みの問題)
- ハングarian記法は使用しない!(好みの問題)
- どこぞのサンプルをそんままコピペしない!(必ず理解して使うこと)

こんな感じです。

あくまでも「俺ルール」なので従う必要はありませんが、自分で作ったルールは遵守するよう心がけましょう。

ちなみに敢えてオススメする本があるとすると

### ゲームプログラマのためのコーディング技術

という本です。

ただ、この本の主張もちょっとクセがあるので、実際に買う前に立ち読みをするか

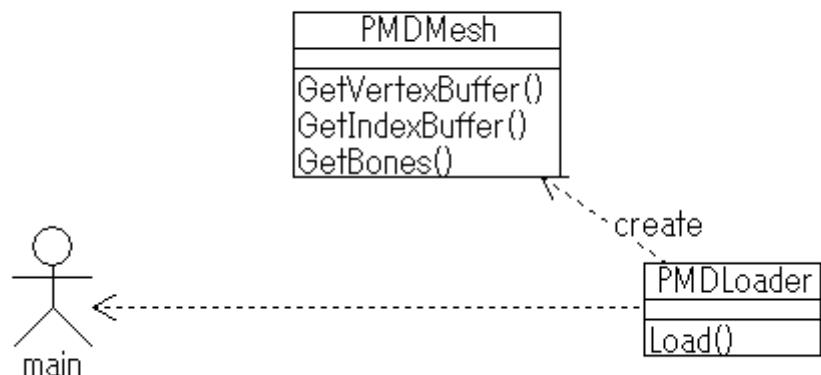
さて、それではリファクタリングに入っていこう。(ちなみに「リファクタリング」ってのはすでに動いているコードを、挙動を変えないようにコードをキレイキレイにしていくことです。  
ノドグった状態でやるべきことでないので注意してください。)

とりあえず指針としては「モデルクラス」もしくは「メッシュクラス」を作って、今回やったメッシュのロードからデータ展開までを一つのクラスの中に入れておいてください。

何故かと言うと、今のままで1種類のキャラロードにしか耐えられないからです。

## 9.5 リファクタリング

ひとまず PMDLoader クラスを作りましょう。



PMDLoader は内部で PMD をオープン／リードして必要な物を詰めこんだ PMDMesh を返します。

簡単でいいので、実装の前にこういった図を書く習慣をつけましょう。書式はなんでもいいです。ちなみにこの画像のツールは UMLMemo というシンプルなツールです。

まあ UML 書式を極めたところで、ゲーム業界への就職可能性は 0.01% も向上しないと思いまして、クラスの関係を「だいたい」示してればいいですよ。

記法理解のテストが有るらしいけど高いしゃめといたほうが良いと思うよ。

ともかくこんな(↑)イメージで分割してみましょう。

一気にやっちゃうのはオススメしません。リファクタリングする時は「拳動を変えない」事が前提なので、ある程度時間をかけてやるものですね。

つまり、ゆっくり、少しずつが「リファクタリング」のコツだと思ってください。



スピードが命のプログラミングですが、リファクタリングだけは少しずつ進みましょう。

## 9.6 今回のリファクタリングの注意点

### 9.6.1 左辺値がありませんエラー

ちょっと言い忘れていましたけど、

```
device.Context()->IASetVertexBuffers(0, 1, &(mesh->GetVertexBuffer()), &stride, &offset);
```

ここでエラーが出ると思います。

原因としては、C++としてこれを許してはいけない理由があるんですね。GetVertexBuffer()そのものは使えるんですよ？でも、そのアドレスは使えないんです。何故ならば…よく考えてください。

```
ID3D11Buffer* vb;
```

の

```
&vb
```

ってのは、上で ID3D11Buffer のアドレスを指し示すために宣言された vb という変数のアドレスを表していますよね？それは良いと思います。ところが

```
mesh->GetVertexBuffer()
```

の場合 mesh->\_vertexbuffer を返していますが、

```
&(mesh->GetVertexBuffer())
```

は`&mesh->_vertexbuffer`のアドレスではなく`mesh->GetVertexBuffer()`のアドレスを返しているわけです。

つまり、関数のアドレスです。こいつをバッファのポインタのポインタとして渡しても使いないわけです。ちょっと分かりにくいけど分かるかな？

## 10 法線情報を使って立体感与えちゃうぞ！！

さて、簡単に立体感を与えちゃってみようか!!!!

実はさっきまで無視してたけど、頂点一つ一つに職人さんが心を込めて「法線情報」を埋め込んでいるんだ!!!!



それを利用して、なんとなく立体感を与えてみよう!!!

やり方は至って簡単!!!!既に法線情報が入っているのだから、それを色に反映するようにすれば良し!!!!

さて、ひとまず法線が無視されてるので、無視しないように使用。レイアウトのところを見てください。

```
{ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
```

こんな感じですよね？座標情報しかないので、ここに法線情報を追加しましょう。

```
{ "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT,  
D3D11_INPUT_PER_VERTEX_DATA, 0 },
```

なお、`D3D11_APPEND_ALIGNED_ELEMENT`についてですがこれは知ておくと便利なやつで、知らないと面倒なやつ。

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

この説明の

**nticName**  
シェーダー入力署名でこの要素に関連付けられている HLSL セマンティクスです。

**nticIndex**  
要素のセマンティクス インデックスです。セマンティクス インデックスは、整数のインデックス番号によってセマンティクスを修飾するものです。セマンティクス インデックスある場合にのみ必要です。たとえば、 $4 \times 4$  のマトリクスには 4 個の構成要素があり、それぞれの構成要素にはセマンティクス名として `matrix` が付けられるセマンティクス インデックス (0, 1, 2, 3) が割り当てられます。

**st**  
要素データのデータ型です。「[DXGI\\_FORMAT](#)」を参照してください。

**Slot**  
入力アセンブラーを識別する整数值です（「入力スロット」を参照してください）。有効な値は 0 ~ 15 であり、`D3D11.h` で定義されています。

**edByteOffset**  
(省略可能)各要素間のオフセット (バイト単位) です。前の要素の直後で現在の要素を定義するには、`D3D11_APPEND_ALIGNED_ELEMENT` を指定できます。

**SlotClass**  
単一の入力スロットの入力データ クラスを識別します（「[D3D11\\_INPUT\\_CLASSIFICATION](#)」を参照してください）。

**nceDataStepRate**  
バッファーの中で要素の 1 つ分進む前に、インスタンス単位の同じデータを使用して描画するインスタンスの数です。頂点単位のデータを持つ要素 (スロット

よく見てください

### AlignedByteOffset

(省略可能)各要素間のオフセット (バイト単位) です。前の要素の直後で現在の要素を定義するには、`D3D11_APPEND_ALIGNED_ELEMENT` を使用すると便利です。必要に応じてパッキング処理も指定できます。

わかるか？

には、

**D3D11\_APPEND\_ALIGNED\_ELEMENT** を  
使用すると便利です。必要に応じてパッキング処

こういうことだ。

だからマニュアルはきっちり読めてんだよ。

手を抜くためにマニュアルはよ~~~~~く読みましょう。

さて、これで法線情報を使えるようになりましたので、シェーダ側を書き換えます。入力に法線情報が入ってきたので、頂点シェーダのパラメータにこれを追加してください。

`float4 normal : NORMAL`

で、頂点情報構造体にも `normal` 追加してください。

```
struct OutputP{
    float4 pos:SV_POSITION;//システム座標をこっちに移動
    float4 normal:NORMAL;//色情報を含ませる(グラデーションのため)
};
```

で、引数の normal を渡してあげます。

```
output.normal = normal;
```

今度はピクセルシェーダ側で

```
return float4(output.normal.rgb,1); //
```

としてみてください。



このように立体感与えちゃったかな?状態になると思います。でも、これは嘘の立体感。こつからが本番です。

まあ、その前にこの解説をしていきたいと思います。

なんでこれが「何となく立体的に」見えるのかってのもそうですけど、どういう情報が見えているのか?何のための情報なのか?って解説をこの後やっていきます。

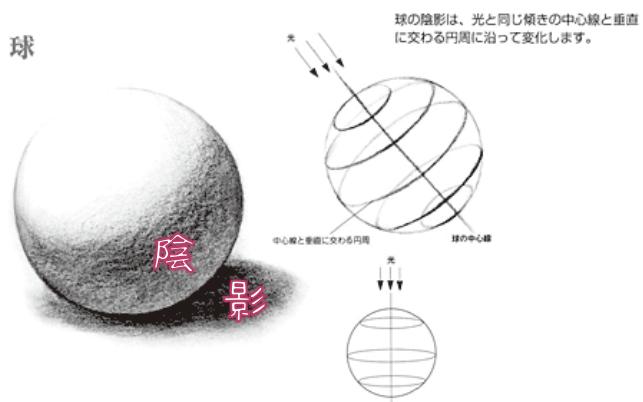
## 10.1 立体感とは？

お客様、立体感、立体感言うとりますけれども…どういうことが分かってますかね？絵を描く習慣のあるひとは何となく直感的にやっていると思うんですが、

物体には明るい部分と暗い部分がありますよね？

光が当たっている面は明るくて、当たっていない部分は暗いですよね？それで立体感を出しているわけです。

ちなみにデッサン教室の資料が的確なので持ってきてました。



ちなみに、CG的には「かげ」には二種類あって、「影」と「陰」があり、今回はこの「陰」の表現についてお話ししていきます（「影」の方は思いの外難しいのです。）

さて、この「陰」について、昔の人がとっても簡単な法則を思いついちゃったので、ひとまずそれについて紹介しましょう。

Lambert の余弦則というやつです。

## 10.2 ランバートの余弦則

<https://ja.wikipedia.org/wiki/%E3%83%A9%E3%83%B3%E3%83%90%E3%83%BC%E3%83%88%E5%8F%8D%E5%B0%84>

2006-2013 H. SHIOZAWA <http://vlab.org>

### 10.4 反射光の計算モデル

拡散反射光(p.123)

- ランバートの余弦則
  - 光がどの方向から入射しても、全方向に均等に拡散
  - 入射角余弦の法則より、表面の明るさは入射角の $\cos\theta$ に比例

$$I = K_d I_d \cos\theta$$

上から照らされると 明るくなる 横から照らされると 暗くなる

I<sub>d</sub>: 入射光の拡散反射成分 I: 反射光  
K<sub>d</sub>: 物体表面の拡散反射率 θ: 入射角

□ 入射角余弦の法則

- 単位面積あたりに当たる入射光の量は入射角の $\cos\theta$ に比例

法線ベクトル  
入射光  
 $\cos\theta$   
 $\theta$   
1

環境反射光(p.122)

- 環境光による拡散反射光
  - 環境光は四方八方から均等に当たるので方向がない
  - 常に同じ色に見える

$$I = K_a I_a \quad (K_a: \text{環境光の反射率})$$

はい、こういうやつです。法線とライトへの方向との角度の $\cos\theta$ に明るさは比例するんじやね？

ということをランバートさんが言ったわけです。これがランバートの余弦則なんですけど、ここで数学のこの式を思い出して欲しい…覚えてるかな？

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

内積の結果はベクトルのそれぞれの大きさとその間の角度 $\theta$ とすると、 $\cos\theta$ はそのまま内積と比例関係にあることが分かりますね？

ちなみに言うと、ベクトル $a$ と $b$ が正規化(大きさを1にする)済みならば、内積= $\cos\theta$ になるわけでそうなると元の明るさ $I$ は

$$I = k_d (\vec{a} \cdot \vec{b}) + I_{ambient}$$

って感じになります。ちなみに $I_{ambient}$ は暗くなりすぎないようにするための単なる「計算下駄」です。

さて、これを使用してシェーダを書いてみましょう。

既に法線情報を取得できているので、そこから内積をとりましょう。

シェーダで内積は `dot` と言う関数を使います。

ついでにいうと正規化の関数は `normalize` と言う関数です。

ちなみにシェーダについて調べたければ `hlsl` で検索すると良いと思います。

[https://msdn.microsoft.com/ja-jp/library/ee418317\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee418317(v=vs.85).aspx)

さて、ランパートの余弦則を使うには光へのベクトルが必要なのでこう定義します。

```
float3 light = float3(-1, -1, -1);
```

で、正規化

```
light = normalize(light);
```

で、内積を取ってそれを明るさとして使ってください。

```
dot(light,normal)
```

うまくいけば



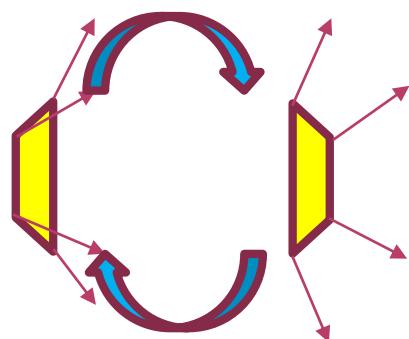
こんな感じに石膏像みたいになります。がんばってください。

でもまだ終わってないんだ。

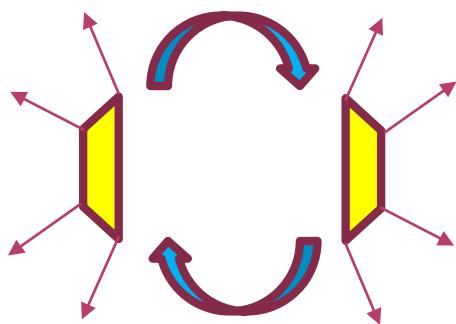
回すために座標を回転させてるよね?ということは法線も回転させなければ矛盾が出るんだ。



後ろを向くとこんな感じになってしまいます。何故かと言うと頂点の座標が回転しても、法線の方向がそのままなので、こうなるんだ。



これでは背中は一生真っ黒になります。ですから法線ベクトルも回転させましょう。



つまり行列の回転を法線ベクトルにも適用します。

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

## 10.3 間違えました

申し訳ございません!!!

ワタクシとしたことが初歩的なミスをしてしまいました!!!（学生様からもご指摘をいただきました!!!）

何かつちゅーと!!!!!!

output.normal=mul(mat,normal);

ここ!!!!

ここ間違いなんですよ!!このmatの中身はなんでしょうかね!!!そう!!!この中身は

world\*view\*projection

でしたね!?

ごめんな?法線ベクトルに対しては「回転しか適用したらアカン」のよ。そもそも法線ベクトルは2Dに演す必要がないので、乗算はworldだけでいい。

どうする必要があるのかというと、worldとview\*projectionを分離する必要があるということです。

そこでworldとview\*projectionを同時に格納する構造体を作ります。

```
struct WorldAndCamera{
    XMATRIX world;
    XMATRIX camera;
};
```

WorldAndCamera wac;

wac.world = world;

wac.camera = view\*proj;

と言った具合に2つ入れられる構造体を作って、そのバイ二フアを作つて送る。

そしてGPU側では2つ受け取ることにする。

この構造体の順番がそのまま GPU での順番になりますから、GPU 側もこう書き換えましょう。

```
cbuffer cbuff{
    matrix world; //ワールド座標
    matrix camera; //カメラ(ビュー*/ペースペクティブ)
}
```

で、座標情報にはこの world と camera を乗算したものを乗算し、ノーマルにはワールドのみを乗算します。

で、注意点としては

- 行列の乗算にも mul を使用してください
- シェーダでの乗算は左からかけてください

つまり、world と camera を乗算するというのは

```
matrix m = mul(camera,world);
```

というわけです。わかりますかね？CPU 側の乗算記号の時は右に右に書いていけばよかつたんですが、シェーダ側では左からかけるのが原則だと思ってください。

うまくいけば、こうなるはずです。



さて、立体感に関してはここまで…。次はテクスチャを張っていきましょう。

## 11 テクスチャ(シェーダリソース)を貼ろう

最初に言っておくと、一般的に3Dで「テクスチャ」と言うものは「シェーダリソース」と呼ばれるものとして取り扱われます。

簡単に言うと「GPUに渡す事のできる大きなデータの塊」のことです。そのデータの塊をGPU側がどうやって扱うのかを定義していくって感がなければいいです。

ああ、そもそもテクスチャってのが分からぬいかもしれないんで言っておくと、今あつかってるモデルを見ると石膏像みたいでしょ？これじゃあ可愛くないわけです。

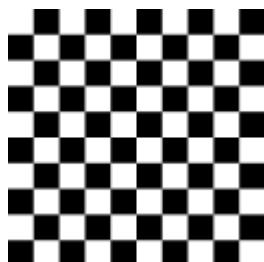
これを可愛くするために、シール的なやつを貼っていく…そのシールをテクスチャって言います。

それを貼るために必要なのが大雑把に言って

- シェーダリソース(テクスチャ)のロード&セット
- UV情報(レイアウト変更)
- サンプラーステート(サンプラ作成)

です。

とりあえずはテストとしてシェーダにテクスチャ投げて、それを表示したいと思います。こいつを使いましょう。



いわゆる市松模様と呼ばれる模様です。

UV情報は既にPMD頂点データの中に含まれていますので、それを使いましょう。

### 11.1 テクスチャロード&セット

肝心のテクスチャのロードについてですが、

D3DX11CreateShaderResourceViewFromFile

[https://msdn.microsoft.com/ja-jp/library/ee416885\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416885(v=vs.85).aspx)

でロードします。

市松模様はサーバに置いておりるので、ダウンロードしていつものフォルダに置いてください。

ひとまずは result を確認しましょ。ロードに失敗してたら S\_OK 以外が返ってるはずです。  
ああ…引数？

もうそろそろ自分で考えていただきたいのですがねえ…何のために URL 見せてるんだと思ってんだよ。



HRESULT D3DX11CreateShaderResourceViewFromFile(

```
①ID3D11Device *pDevice,//いつも使ってるやつだよわかるだろ  
②LPCTSTR pSrcFile,//テクスチャファイルパス  
③D3DX11_IMAGE_LOAD_INFO *pLoadInfo,//nullptrでいい  
④ID3DX11ThreadPump *pPump,//nullptrでいい  
⑤ID3D11ShaderResourceView **ppShaderResourceView,//欲しいやつ  
⑥HRESULT *phResult//nullptrでいい  
);
```

ちなみに3番目の引数はテクスチャをそのまま読み取る時には nullptr でいい。nullptr でない場合は色々と指定できるが、今その必要はない。

④と⑥は null 以外の実体を入れると「即時復帰関数(同期)」として動作し、ロードが完了しなくても復帰します。でもスレッドとかまだやってないので、ここも nullptr です。

じゃあロードして result 確かめてください。

ロードしたテクスチャを GPU に伝えるのは簡単です。

PSSetShaderResources

[https://msdn.microsoft.com/ja-jp/library/ee419731\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419731(v=vs.85).aspx)

を使用します。簡単ですね。安定の突破口番とビューの数 1 つ。最後の引数は話の流れからしてもちろん…。

VSSetShaderResources もありますが、これは使いみちがよくわかりません。そもそも VS のほうでグーグルセンターに聞いても「もしかして PSSetShaderResources?」と言われる有様で、MSDN にすらたどり着かせてくれません。

[https://msdn.microsoft.com/ja-jp/library/ee419768\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419768(v=vs.85).aspx)

↑あるのはあるんだけどね…

まあともかく GPU 側にテクスチャ情報を送ります。GPU 側ではこう書いてください

Texture2D tex;

[https://msdn.microsoft.com/ja-jp/library/bb509700\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509700(v=vs.85).aspx)

に書いてあるとおり 2D テクスチャを表します。それ以外の意味はとくにないです。先程送り込まれたテクスチャの情報が入っています。

さらにその下にでもこう書いてください

Sampler sample;

サンプラーといふやつです

[https://msdn.microsoft.com/ja-jp/library/bb509644\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509644(v=vs.85).aspx)

まあ、またあとで話します。

ともかくこの 2 つを置いといてください。

さあ、ロードはできた。GPU にセットもした。受け取る準備もできた。

## 11.2 レイアウト変更

次の仕事は頂点レイアウトの追加だ。今回追加するものは UV 値だ。

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

さて、今一度ここを見てみよう。追加と言っても、今回はどう書けば良いのだろうか？UV 座標に当たるものってなんだろうな。

よく見ると近いものはあるようだ。

TEXCOORD(n)	テクスチャ座標
-------------	---------

そう、この「テクスチャ座標」が今回欲しいUV 座標に当たるのだ。つまり…あとは分かるね？

いつもどおりだ。ああ、いつもどおりではないけな。今回は float2 つ分だから

R32G32\_FLOAT

とでもしておいてくれ。

さて、レイアウトもできたら uv が GPU 側で受け取れるようになるぞ!!!

これもまた受け取る準備をしよう。

例によって、頂点シェーダが受け取れる引数を増やしてくれ

float2uv:TEXCOORD

を引数に追加だ。

ついでにアウトプット構造体にも追加だ。これで uv 情報がピクセルシェーダまで流れるようになる。試しに uv をそのまま色として出力するとこんな感じ。



なお `return float4(0.5, outputp.uv, 1);` などで表示されるもよう。

なるほど、UVはきっちり設定されているようだ。

ではこれをテクスチャとして使うにはどうしたらいいんでしょうか…

### 11.3 UV値からテクスチャの「色」を抜き出す

UV座標からテクスチャの「色」を抜き出すには、先程 Texture2D型として取ってきたテクスチャ(シェーダリソース)の Sample 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)

そして個々の説明が非常にわかりづらい…

Object.Sample(サンプラー, UV座標);

でいいじゃんと思うんですけど…。で、戻り値も「テクスチャフォーマット」などというわからない事を書いています。これ日本語誤訳ちゃうんかい…。

[https://msdn.microsoft.com/en-us/library/windows/desktop/bb509695\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509695(v=vs.85).aspx)

と思って英語見ましたが、同じような感じで…。これほんとユーザビリティ悪いよな。まったく DirectX11まわりはこんなばつかしや。

結局アレやろ？意訳するとやな、突っ込んだテクスチャのフォーマットによって戻り値も変わる……つまるところ元の画像データの

[https://msdn.microsoft.com/ja-jp/library/ee416920\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416920(v=vs.85).aspx)

の

[https://msdn.microsoft.com/ja-jp/library/ee416919\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416919(v=vs.85).aspx)

の

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

フォーマット通りに色(とは限らんか?)データが入るというわけだ…分かるかんなもん!!!

こいつら分からず気ないやろホンマに。

…まあともかく画像の色データが返ってくるんや。とりあえずは float4 と思っておいたら間違いないんちゃいますやろか？

```
float4 col = tex.Sample(sample, outputp.uv);
```

この中のRGBだけ欲しければcol.rgbと書けばいい。

そして既に陰影をつけている濃淡データに対して乗算をかけるとあら不思議…。



なんだが…これでは流石に可哀想である。きちんとしたデータを貼ってやろう。あ、この市松模様に関してはミクさんは不適切な表示になります。

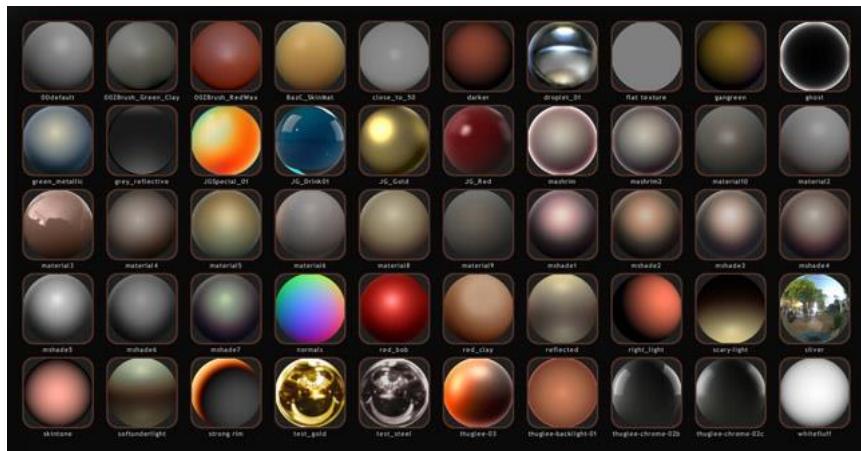
何故かと言うと、ミクさんのマテリアルデータはテクスチャがもともと貼られておらず頂点データだけでミクさんが表現されています。

つまりここに市松テクスチャを貼ると、白、もしくは黒としか表現されなくなってしまう。正しいか間違っているかの判別が難しいからです。

ちょっとここだけはミクさん以外のデータを取ってきて実験してください。

ちょっと一旦ここで僕もミクさんに戻すこととする。何故かと言うとミクさん(やハクさんなどのデフォルトの人たち)以外はちょっとめんどくさい場合があるからだ。

## 12 マテリアルで分けよう



そもそも「マテリアル」とは何でしょうか?

日本語にすると「材質」「材料」「素材」とかそういう意味があります。正確に言うと「表面材質」と捉えたほうが良いです。

「中身の材質」じゃなくて、あくまでも「表面」の材質ね？

表面の色や、テクスチャ…光沢が変わればそれすなわち「マテリアルが違う」

ということになります。

PMDでは基本的に

表面の色(ディフューズ)、光沢色、環境色、およびテクスチャファイル名

として定義されています。

ここにも色々と書いてますが、うーん。いったんテクスチャは置いておいて、表面色のみでやってみましょう。

で、マテリアルデータは↑に書かれているようにまずマテリアルの数が書かれており、その後にずら～っとマテリアルのデータ本体が並んでいます。

69E0Bface_vert_count	マテリアル数	00000011
69E0Cmaterial_count		3E083127 3F1EB852 3F36C8B4
69E0Dmaterial[0].diffuse_color[0]		3F800000
69E0Fmaterial[0].alpha		40A00000
69E0Fmaterial[0].specularity		00000000 00000000 00000000
69E01material[0].specular_color[0]		3D883127 3E9EB852 3EB6C8B4
69E0Dmaterial[0].mirror_color[0]		00
69E19material[0].toon_index	マテリアルデータ	01
69E1Amaterial[0].edge_flag		00000006
69E1Bmaterial[0].face_vert_count		00 00
69E1Fmaterial[0].texture_file_name[0]		00 00 00 00
69E2Fmaterial[0].texture_file_name[16]		3E083127 3E9EB852 3EB6C8B4
69E33material[1].diffuse_color[0]		

こんな感じですね。

さて、この構造。また面倒なことになっています。どこが面倒かというと

```
struct PMDMaterial{
```

```
    float diffuse_color(3); // dr, dg, db // 減衰色
    float alpha; // 減衰色の不透明度
    float specularity;
    float specular_color(3); // sr, sg, sb // 光沢色
    float mirror_color(3); // mr, mg, mb // 環境色(ambient)
    BYTE toon_index; // toon???.bmp // 0.bmp:0xFF, 1(01).bmp:0x00 10.bmp:0x09
    BYTE edge_flag; // 輪郭、影
    DWORD face_vert_count; // 面頂点数
    char texture_file_name(20); // テクスチャファイル or スフィアファイル名
};
```

赤字の部分です。何でかは…分かりますよね？そう、アライメントが発生するのでちょっとコード時に工夫しなければいけないです。

あと、最後の「面頂点数」はすごく重要なので、ここに入る数値はきちんとチェックしておきましょう。

とにかく『初音ミク.pmd』で見ていきましょう。

## 12.1 マテリアル数ロード

これは簡単ですね。

さっきの material\_count を 4 バイト取ってきます。ちなみに 4 バイトもイラネーだろって思いますが、そういうツッコミはなしで。

インデックスを全読み込みした直後にマテリアルがあるので、それを取得してください。

初音ミクの場合だと 17 個のマテリアルがあります。数を確認してください。

## 12.2 マテリアルデータロード

17 個マテリアルが有るので、ロード処理を 17 回読み込み処理を行います。17 の決め打ちはダメですよ？きちんとロードした結果を使ってください。

で、今回使用するのは「色」と「面頂点数」を使って、今の石膏像を塗り分けていきます。

さっさとさっきの構造体を使ってロードしましょう。うん…もちろんアライメント対処します。めんどくせえ…まあここでしか使ないので#pragma pack(1)使っても良いんだけど…みんなが変なふうに覚えてても困るので、敢えてかつこ悪く書きます。

```
std::vector<PMDMaterial> materials(materialCount);
for (auto& m : materials){
    fread(&m, sizeof(float), 11, fp);
    fread(&m.toon_index, sizeof(BYTE), 1, fp);
    fread(&m.edge_flag, sizeof(BYTE), 1, fp);
    fread(&m.face_vert_count, sizeof(BYTE), 24, fp);
}
```

まあ、大したことにしてないんですけど。

ひとまず、↑の face\_vert\_count の合計が、ミクさんのインデックス数と一致しているのを確認してください。

確認できましたか？

では次にカラーデータをピクセルシェーダのコンスタントバッファにセットできるようにしたいと思います。

### 12.3 マテリアルデータをGPUに渡す準備

思うのですが、少々面倒なことがあります。

現在、ピクセルシェーダと頂点シェーダと一緒にして使っているため(後々分離しますけど)このままコンスタントバッファにセットしてしまうと使い分けがちょっと面倒なことになります。あくまでもカラーはピクセルシェーダ側で使用したいので、マテリアルデータ(今はカラーデータのみ)用のバッファ作って送ってやれば良いのですが、そのままだとごっちゃになってしまいます。

なのでごっちゃにならないよう、スロットを分けます。

やっとスロット番号の意味が出てきましたね。

現在の

```
cbuffer cbuff{
    matrix world; //ワールド座標
    matrix camera; //カメラ(ビュー*/ペースペクティブ)
}
```

を、

```
cbuffer cbuff : register(b0){
    matrix world; //ワールド座標
    matrix camera; //カメラ(ビュー*/ペースペクティブ)
}
```

と書き換えてください。↑のb0ってのがレジスタ番号…つまりスロット番号だということです。

ここにカラーデータを入れられるように次のスロット用の記述をします。

```
cbuffer cbuff : register(b1){  
    float3 color;  
}
```

これで行列とカラーの住み分けができました。あとは…CPP 側です。

今回、例の構造体で使用するのは `diffuse_color(3)` と `face_vert_count` です。これをクライアント側から見えるようにしたいと思います。

更に言うと、こいつらはマテリアル数分ありますので、例えば PMDMesh の中でクラス無いクラスとして

```
struct Material{  
  
    float diffuse(3); // 色情報  
  
    unsigned int count; // カウント  
  
    unsigned int offset; // オフセット  
};
```

などと定義しておきます。そして配列(ベクタ)化して PMDMesh 内に持つておきます。

その上で先程ロードした情報をそれぞれ代入していきます。

さて、ここまでできたら、あとは GPU に渡すだけです。

コンスタントバッファをマテリアルデータ用に用意してください。`materialBuffer` とかでいいです。

この際にちょっと注意点。

カラーデータだけなので、

```
XMFLOAT3 color = { 0.5, 1.0, 0.5 };
D3D11_BUFFER_DESC desc = {};
desc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
desc.ByteWidth = sizeof(color);
desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
desc.Usage = D3D11_USAGE_DYNAMIC;
D3D11_SUBRESOURCE_DATA initdata = {};
initdata.pSysMem = &color;
ID3D11Buffer* materialBuffer = nullptr;
result = device.Device()->CreateBuffer(&desc, &initdata, &materialBuffer);
device.Context()->PSSetConstantBuffers(1, 1, &materialBuffer);
```

と、書いても構わんように思えるんですが、ここでマイクロソフトさんは罠を張ってます。スキを生じぬ二段構えやな

このままやってみ?

なんとかエラー返ってくるから。

というわけで、そんな時はもう一度 CreateBuffer のヘルプを見ましょう。

[https://msdn.microsoft.com/ja-jp/library/ee416048\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416048(v=vs.85).aspx)

さて…

何か興味深い記述はないかい?

そう!!!

#### 解説

この構造体は、バッファー リソースを作成するために `ID3D11Device::CreateBuffer` によって使用されます。

この構造体に加え、`D3D11.h` には派生構造体 (`CD3D11_BUFFER_DESC`) もあります。これは、継承されたクラスのように機能するので、バッファーの記述を作成する際に役に立ちます。

バインド フラグが `D3D11_BIND_CONSTANT_BUFFER` の場合、`ByteWidth` には、`D3D11_REQ_CONSTANT_BUFFER_ELEMENT_COUNT` 以下で 16 の倍数となる値を指定する必要があります。

これ!!!

イラッ…!!



知らないとハマります。

解消するためには float4つぶん渡せば良いんですが、それだと汎用性に欠けるのでこうします。

```
desc.ByteWidth = sizeof(color) + (16 - sizeof(color) % 16) % 16;
```

何をやってるか分かりますかね…そう、手動アライメントというか、そういう事をやってるんですよ!!!!くっそー!!!!ちくしょつおおおお!!!!めんどくせー!!!!

まあ、何はともあれ、これでカラーデータを渡すことができました。

↑のカラー設定通りにするならば、恐らく緑色になっていると思います。

どうですか？

緑色ですか？ それなら正解です。

さて…あとはこれをマテリアル毎に分けるだけです。

## 12.4 マテリアル毎に色を変える

さて、先程の構造体…覚えていませんでしょうか？

```
struct Material{
```

```
    float diffuse[3]; //色情報
```

```
    unsigned int count; //カウント
```

```
    unsigned int offset; //オフセット
```

};

この、カウントとオフセット…これが重要です。

実は DrawIndexed 関数は、インデックス数と、インデックスのオフセットを指定できるのですよ。覚えてますか？

[https://msdn.microsoft.com/ja-jp/library/ee419591\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419591(v=vs.85).aspx)

ここにそれぞれの値をループしながら放り込みながら、ドッファの色を変えていけばいいのです。…ところで、カウントは PMD マテリアルデータに有るからいいけどオフセットはどうするの？って思ってるひといますか？

ちょっと自分で考えてみましょう。

しばらく考えてください。うまいこと行けば



こういう感じに表示されるはずです。

…よ～く考えれば分かるはずです。

ヒントはそれぞれのインデックスを足すと、全てのインデックスに等しくなる…これがヒントです。

…そろそろ地獄の始まりかもしれません。ここまで割とソースコード見せてきましたからね。最初のスタートダッシュの加速はついていると思いますのでそろそろハシゴを外していくますよ～。

## 12.5 マテリアル毎にテクスチャを…

マテリアル毎にテクスチャを変えていきます。

ここがちょっと面倒なんんですけどね。

ひとまずちゃつちゃと作りましょう。

マテリアルロードループ時にテクスチャパスを取ってきます。ロードした物を入れる容器を定義する必要がありますので、追加しておきます。

```
struct PMDMaterial{
    float r, g, b;
    unsigned int offset;
    unsigned int count;
    ID3D11ShaderResourceView* texture;
};
```

そしてループしながらロードします。

```
for (int i = 0; i < materialCount; ++i){
    result = D3DX11CreateShaderResourceViewFromFile(dev.Device(),
        materials[i].texture_file_name,
        nullptr,
        nullptr,
        &ret->.materials(i).texture,
        nullptr);
}
```

このループが終わったら、きちんとテクスチャが入っているのをご確認ください。

初音ミクの場合はこのテクスチャは『eye2.bmp』しかないので(しかも5番目のみ)そこにnullptr以外が入っていればいいです。

あとはこれをマテリアルデータループのときにテクスチャをセットしてあげればいいんです。

```

for (auto& m : mesh->Materials()){
    device.Context()->Map(materialBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedSubresource);
    memcpy(mappedSubresource.pData , &m, sizeof(float)*3);
    device.Context()->Unmap(materialBuffer, 0);
    device.Context()->PSSetShaderResources(0, 1, &m.texture);
    device.Context()->DrawIndexed(m.count, m.offset, 0);
}

```

さて…

うまいこと行くと…こうなってしまうはずです。



何でこうなってしまうのでしょうか?それはですね『目』以外のテクスチャがない&UVが設定されていないためにこのような状況になっているのです。

色々と対処法は有るんですが、手っ取り早い対処法として『null テクスチャ』を用意します。

簡単です。

サーバーに null.bmp か nulltexture.png がありますので、PMDLoader のコンストラクタあたりでロードしてメンバ変数として保管しておきます。

そしてそれを先程のロード時マテリアルループのときに

```

if (ret->_materials(i).texture == nullptr){
    ret->_materials(i).texture = _nulltexture;
}

```

こんな感じ。

あとはうまく行けば



このように完成品のミクさんが表示されるはずです。



靈夢さんもこの通り。ところが…いくつかの PMD モデルでは表示されません。理由は結構な確率でテクスチャが「tga ファイル」になっているからです。

TGA ファイルは過去の遺物である。

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

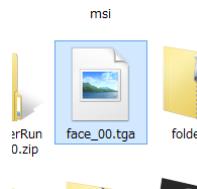
<http://3dtech.jp/wiki/index.php?TGA%E7%94%BB%E5%83%8F%E3%83%95%E3%82%A9%E3%83%BC%E3%83%9E%E3%83%83%E3%83%88%E8%A9%B3%E7%B4%BD>

正直このような時代遅れのフォーマットが使われている理由もよくわからないし、教育的意義も感じない。だから解説しない。(CG のセンターに聞いてもゲームの現場では使用されていないそうだ)

何故こんなフォーマットなのがはよくわからないが、つまりそういうことだ。一番手っ取り早い方法としてはグラフィクスツールで png に変換し、変換するが拡張子はそのままにしておくという方法がある。ぶっちゃけこれでいい。

TGA ファイルは例えば有名どころでは unity-chan などで使用されている…けどなあ…まあ暇がでてからでいいよ。

ちなみに Unity-chan の例だと

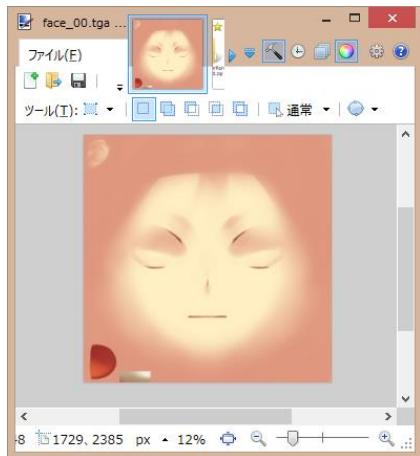


ご覧のように Windows では「画像」と認識はされるもののプレビューできる形式ではありません。このため Photoshop やその他画像ソフトで開くのですが…Windows「ペイント」では



こうなります。フリーのツール(GIMP など)で開けるので、それで開きます(僕はいつも GIMP ではなく Paint.net ってのを使ってます)

開くといつもどおり



こういう画面が出てきますので、保存するときに形式を png 形式に変更します。

そうすると PNG 画像として出力されますが、PMD ファイルの中身が TGA 指定なので、ここでリネームします。



はい。ここまでやれば画像を読み込むことができます。しかしこのやり方もテクスチャ一個一個に適用しなければならないので、いいやり方ではないですね。

授業で暇ができるたら TGA 読み込みも実装したいと思います(暇なんてないんですけど)

ちなみに TGA ローダーだが…どうしても必要ならば

<https://github.com/serge-rgb/tgaLoader>

こいつを使うといいだろう。

## 13 モデルをフォルダ分けしたい

ちょっと DirectX からは離れますか、現在はモデル一体だけだからモデルデータとテクスチャデータを直接プロジェクトと同じフォルダに置いていくと思います。

ところがこれではモデルが増えてきた時などに、プログラムのフォルダまでゴチャゴチャになってしまいます。

『じゃあモデルごとにフォルダで分ければいいじゃない!』と思うかもしれません

例えば…

名前	更新日時	種類
Debug	2016/10/13 13:43	ファイル フォル
博麗靈夢	2016/10/13 9:55	ファイル フォル
App.cpp	2016/09/20 14:07	CPP ファイル
App.h	2016/09/20 14:12	H ファイル

のようにフォルダに全て入れてしまってから、パスを

CreatePMDMesh("博麗靈夢/reimu\_F01.pmd");

のように指定すれば済む話かと思いますが、そうはいきません。



まさかの石膏像に逆戻りです。

```
material[7].texture_file_name[0]    72 5F 68 61 69 72 2E 62 6D 70 00 FD FD FD FD r_hair.bmp ...
material[7].texture_file_name[16]   FD FD FD FD ...
material[8].diffuse_color[0]       3F400000 3F400000 3F400000
material[8].alpha                3F800000
material[8].specularity         40A00000
material[8].specular_color[0]    3DC00000 3DC00000 3DC00000
material[8].mirror_color[0]      3F000000 3F000000 3F000000
material[8].toon_index           04
material[8].edge_flag            01
material[8].face_vert_count     00006DFE
material[8].texture_file_name[0]   72 5F 62 6F 64 79 2E 62 6D 70 00 FD FD FD FD r_body.bmp ...
material[8].texture_file_name[16]  FD FD FD FD ...
material[9].diffuse_color[0]       2F400000 2F400000 2F400000
```

画像ファイルが指定されてはいますが、これは「モデルからの相対パス」なんですよ。そこはわかりますか？

これを抜き出してテクスチャファイルを読み込もうとするんですけど、プログラムで指定するときにこれをそのまま D3DXShaderResourceViewFromFile に渡すと、それは「プログラムからの相対パス」になるわけです。

どうやって解決しましようか？

何か明確な答えがあるわけでもありません。そういうものを何とかするのがプログラマの仕事です。

C++の文法や Windows API ってのはそういう「困難な状況に立ち向かうための武器」だと思ってください。武器がなければタレリほど解決の選択肢は増えます。

ひとまず僕のやり方を紹介すると…

①モデルを読み込む際に、指定パスから「フォルダ」だけ抜き出す(ファイル名を消去する)

②①で抜き出したフォルダ名と、テクスチャパスを連結する

書いてみると手順すくないですねえ…

でこれをやるために一つの武器を教えないといけないのだ。それは…

### 13.1 std::string(文字列型)

実は今まで何度か出てきてはいるのだが、大まかにどういうものか分かってますか？C 言語の時は文字列といえば

```
char* str="abcde";
```

こんなんですよね？ 文字列終端を'¥0'というルールにして文字列という風に扱っていました。

ただ、これ文字列を連結したりしようとすると結構面倒なんですよね。増える文の文字列を確保して、連結結果を入れて…だのなんだの。文字列長を見るためにわざわざ strlen とか使わなければいけないし、文字列同士の比較も strcmp なんて使わなければいけない。

…そういう面倒臭さをある程度解消してくれるのが std::string です。

`std::string` はいわば `std::vector` の亜種です。

要は

```
std::vector<char> str;
```

みたいなもんなんです。となると色々と使い方は見えてくるでしょう？まあ `vector` は前に説明していますから、ここでは `std::string` ならではの機能を紹介しましょう。

<http://vivi.dyndns.org/tech/cpp/string.html>

殆どは `vector` からの継承なのでそれ以外…

の中でも今回使うものと言います。

- 文字列の検索(`find/rfind`)
- 部分抜き出し(`substr`)
- 文字列連結(`+, +=`)

の3つです。

何故この3つを使うのか？言語の問題ではなく「問題解決」の観点から考えてください。自分で考えてみてください。

…自分の頭で考えてください。あれ？今の「問題」が何か忘れた？そりや重症だ。

「テクスチャのファイルパスが分からぬ！」→「モデルのファイルパスを利用しよう」

これでした。ファイルパスといえども文字列…!!



あとは自分の頭で考えましょう。

言われるがままに書いていくのはプログラマではなく「キーボードパンチャー」だぜ？

考える…それが仕事だ。

## 手順

1. PMDLoader に渡された文字列の中から最後の'/'もしくは'¥'を検索  
(検索結果はインデックスで保存されます)
2. 文字列の先頭から 1 で取得したインデックスまでを抜き出す
3. 2 で抜き出した文字列をテクスチャのファイルパスに連結します

以上です。今回使用する関数は

`rfind,substr,+,+=`です。

- `find` は指定文字列を先頭から検索し、最初に見つけた場所を返します。
- `rfind` は指定文字列を末尾から検索し、最初に見つけた場所を返します(最後の場所)
- `substr` は指定されたインデックスから指定された長さの文字列を返します。
- `string` 型 A,B があり、`A+B` は連結文字列を表します。また、`A+=B` は A に B を連結します

考える…考えるよ？

なお、`string` を使用するには

```
#include<string>
```

が必要です。

さて…ここで参考のため僕のコードも見せておこう。これを丸写しするようじゃプログラマとしての資質が問われますよ？技能的な意味じゃなくて、プログラマになろうという意識的な意味で。

```
//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¥か/を得られればいいので、双方のrfindをとり比較する  
//int型に代入しているのは見つからなかった場合はrfindがepos(-1→0xffffffff)を返すため
```

```
int pathIndex1=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)
```

まあ…こういうことだ。

これで靈夢さんもこの通り



## 14 背面カーリングについて

さて…靈夢さん正面を向いている時は良いんですけど後ろを向くと…



レリうなじですね…フヒヒ。

ではなく、リボンが消えているのが分かると思います。これは「背面カリング」といって、「不要な裏側を描画しない機能」によるものです。

例えば箱を作った時、箱の裏側は「存在はしていても見える必要のないもの」ですよね？

そういう無駄なものは省くのがプログラマの世界ですが、MMDは背面カリングを行わないため、PMDデータの場合はモデルの制作ツールによってはこのように面が貼られていないことがあります。

これをどうにかする方法はないのか？というとあります。あるんですが、通常はゲームの効率を下げる事になるため、通常はカリングONだということを知っておいてください。それを知った上でカリングをOFFにしてみましょう。

## 14.1 ラスタライザステートを生成して、セット

カリングはレンダリングパイプラインの中で「ラスタライザ」…つまり「塗りつぶすか、塗りつぶさないか」を決める部分に対して、カリングを設定していきます。

いつもの流れですが、CreateRasterizerStateで生成し、

[https://msdn.microsoft.com/ja-jp/library/ee419799\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419799(v=vs.85).aspx)

RSSetStateでセットします。

[https://msdn.microsoft.com/ja-jp/library/ee419742\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419742(v=vs.85).aspx)

「RS」自体が「ラスタライザーステージ」を表しているため SetStateで十分なのです。

RSSetStateはセットするだけなので簡単なので、キモは CreateRasterizerState…その中でも第一引数の D3D11\_RASTERIZER\_DESC

[https://msdn.microsoft.com/ja-jp/library/ee416262\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416262(v=vs.85).aspx)

です。

この中でカリングを表すのが二番目の要素…D3D11\_CULL\_MODEです。

[https://msdn.microsoft.com/ja-jp/library/ee416078\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416078(v=vs.85).aspx)

これを

D3D11\_CULL\_BACK にすればいつもの「背面カリング」になります。

カリングを OFF にするには D3D11\_CULL\_NONE にします。

なお、D3D11\_CULL\_FRONT にすると、全部ひっくり返ります。



嬉しい! なんか嬉しい! なんかよくわからない! 画像になりますね(SCPみたいですね)

あと、この構造体。設定するのは CULL だけで良いんですけど、FillMode…つまり塗りつぶしモードも設定してあげないとダメっぽいです。

D3D11\_FILL\_SOLID

を指定してください。日本語にすると固体をいっぱいにするって意味ですが、こういうのは意訳しないとわけわかんないです。まあ、ぶっちゃけ「ベタ塗り」って思ってください。

じゃあ SOLID 以外になんがあるんか? WIREFRAME があります。それを使うとこの通り



ねつ

ともかくカリングオフにすれば



こうなります。では次に行ってみましょう。

## 15 うまく表示されない

皆さん「俺の嫁」を表示して満足かなと思いますが、嫁によっては正常に表示されないことがあります。

### 15.1 PMD 特有のテクスチャルール

それにはいくつもの理由があります。大抵はテクスチャ周りの指定に問題があります。PMDはテクスチャの指定にちょっとルールがあるんです。



これを見てください。ちょっと目がおかしいのは後で解説しますが、そもそも服とか、体のテクスチャがおかしい。

```
0].face_vert_count          00004032
0].texture_file_name[0]      91 CC 2E 70 6E 67 2A 66 6B 2E 73 70 61 00 FD FD 体.png*fk.spa ...
0].texture_file_name[16]     FD FD FD FD
1].diffuse_color[0]          3F800000 3F800000 3F800000
1].alpha                     3F7FBE77
1].specularity               40A00000
1].specular_color[0]         00000000 00000000 00000000
1].mirror_color[0]           3F000000 3F000000 3F000000
1].toon_index                 00
1].edge_flag                   01
1].face_vert_count          00000F0C
1].texture_file_name[0]      95 9E 2E 70 6E 67 2A 66 6B 2E 73 70 61 00 FD FD 反.png*fk.spa ...
1].texture_file_name[16]     FD FD FD FD
...
```

実はテクスチャ名が、ご覧のように妙な指定になっているのです。

OO.spaって書いてますね？

これは何なのでしょう？

<https://wwwb.atwiki.jp/vpvpwiki/pages/216.html>

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32](http://blog.goo.ne.jp/torisu_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32)

うへん。まあつまるところ『スフィアマップ』とやらを指定すると

ファイル名\*スフィアマップ名

のようなテクスチャファイル名になるようですね。

というわけで、ひとまずスフィアマップ自体は置いといて、正しくテクスチャ名を取得しましょ。

以前に紹介した `string` を使用します。`find` で \* を検索してもしあつたら分割処理を行いましょ。

① `find` で検索

② `substr` で 0 ~ `find` で見つかったところまでを『ファイル名』とする。ですね。

うまいこと行けば



なのですが、目がおかしい。これがアルファブレンディングです。

## 15.2 アルファブレンディングの話をしよう

```
ID3D11BlendState* blendstate=NULL;
D3D11_BLEND_DESC blenddesc={};
blenddesc.AlphaToCoverageEnable=false;
blenddesc.IndependentBlendEnable=false;
D3D11_RENDER_TARGET_BLEND_DESC& blrtdesc=blenddesc.RenderTarget[0];
blrtdesc.BlendEnable = true;
blrtdesc.SrcBlend = D3D11_BLEND_SRC_ALPHA;
blrtdesc.DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
blrtdesc.BlendOp = D3D11_BLEND_OP_ADD;
blrtdesc.SrcBlendAlpha = D3D11_BLEND_ONE;
blrtdesc.DestBlendAlpha = D3D11_BLEND_ZERO;
blrtdesc.BlendOpAlpha = D3D11_BLEND_OP_ADD;
blrtdesc.RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;
float blendFactor[] = {0.0f, 0.0f, 0.0f, 0.0f};
device->CreateBlendState( &blenddesc, &blendstate );
context->OMSetBlendState( blendstate, blendFactor, 0xffffffff );
```

これを初期化のときにやってください。解説は後でやります。

これも DirectX9 に比べるとかなり複雑です。1つの流れではありますけどブレンドステートって奴を作つておいて、これを OMSetBlendState に設定する。

ちなみに OM ってのは OutputMerger のことで、「今から描画する画像を、既に書かれている画像とどう合成するか」を表しています。

ともかく CreateBlendState でステートを作成して

[https://msdn.microsoft.com/ja-jp/library/ee419779\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419779(v=vs.85).aspx)

[https://msdn.microsoft.com/ja-jp/library/ee416043\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416043(v=vs.85).aspx)

OMSetBlendState で設定すれば良いのです。

[https://msdn.microsoft.com/ja-jp/library/ee419702\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419702(v=vs.85).aspx)

ちなみに、この第二引数の BlendFactor は DESC の中のブレンドで BLEND\_FACTOR 系を指定したときにのみ有効なので、アルファを有效地にする程度の処理では意味を持ちません。

まず、AlphaToCoverageEnable ですが、これ… 説明を読むと

「ピクセルをレンダーティーゲットに設定するときに、アルファトウカバレッジをマルチサンプリング テクニックとして使用するかどうかを決定します。」

もはや「アルファのルシガコクーンをページ」状態ですね。

英語で書くと Alpha to Coverage です。アルファは分かりますが Coverage ってなんでしょう?<http://eow.alc.co.jp/search?q=coverage>

「被覆率」とか書いてますが、「遮蔽率」と考えたほうがわかりやすいでしょう。

[https://msdn.microsoft.com/ja-jp/library/bb205072\(v=vs.85\).aspx#Alpha\\_To\\_Coverage](https://msdn.microsoft.com/ja-jp/library/bb205072(v=vs.85).aspx#Alpha_To_Coverage)

どうも MSAA(マルチサンプリングアンチエリアシング)と関係しているようです。

[http://maverickproj.web.fc2.com/d3d11\\_19.html](http://maverickproj.web.fc2.com/d3d11_19.html)

ん?

「 $\alpha$  テストを有効にした場合と同じ」…だと?

まあ…良くわからない。もちろん信用すべきは MSDN だが…。まあ今回はアンチエリアシングしてないので, false でいいでしょう。将来アンチエリ有効にしてからにしましょう。

次に IndependentBlendEnable ですが、これは

「同時処理のレンダー ターゲットで独立したブレンディングを有効にするには、TRUE に設定します。FALSE に設定すると、RenderTarget[0] のメンバーのみが使用されます。RenderTarget[1..7] は無視されます。」のことです。まあ、わからんでもない。

今回はレンダーターゲットは1つだけなので, false でいいでしょう。

最後の RenderTarget(8)ですが…これは…

[https://msdn.microsoft.com/ja-jp/library/ee416264\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee416264(v=vs.85).aspx)

ブレンドの仕方を指定します。

```
typedef struct D3D11_RENDER_TARGET_BLEND_DESC {  
    BOOL BlendEnable; // ブレンドするかどうか  
    D3D11_BLEND SrcBlend; // 今から描画する画像のブレンド具合  
    D3D11_BLEND DestBlend; // すでにある画像のブレンド具合  
    D3D11_BLEND_OP BlendOp; // どのようにブレンドするか  
    D3D11_BLEND SrcBlendAlpha; // 描画する画像のアルファ値のブレンド具合
```

```

D3D11_BLEND DestBlendAlpha; //すでにある画像のアルファ値のブレンド具合
D3D11_BLEND_OP BlendOpAlpha; //どのようにアルファをブレンドするか
UINT8 RenderTargetWriteMask; //ブレンド対象をビットで指定できる
} D3D11_RENDER_TARGET_BLEND_DESC;

```

Op はオペレーション(操作)の略やね…まあええねん。ブレンドさせたいんやつたら当然ながら BlendEnable は true やな?

```

ID3D11BlendState* blendstate = nullptr;
D3D11_BLEND_DESC blenddesc = {};
float blendFactor() = { 0, 0, 0, 0 };
blenddesc.RenderTarget[0].BlendEnable = true;
result = _device->CreateBlendState(&blenddesc, &blendstate);
_context->OMSetBlendState(blendstate, blendFactor, 0xffffffff);

```

試しにこれで作ってみまひよか? そうすると result は S\_OK やで? でもな?



当然っちゃん当然やな。やっぱ設定する必要がありますわ。RenderTarget[0]をバンバン設定していくやで?

ひとまずブレンド具合を設定しなければならんのやけど、

```

blenddesc.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
blenddesc.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_ALPHA;
blenddesc.RenderTarget[0].DestBlend = D3D11_BLEND_INV_SRC_ALPHA;

```

それぞれ解説する。

BlendOp を OP\_ADD にしていますが、これはどういうことを意味しているのかといふと元の画像のピクセル値を D とし、描画しようとする画像のピクセル値を S とすると

結果のピクセル値 C は

$$C=D+S$$

となるわけです。通常はブレンドそのものをしないので

でもこれだとただ単に明るくなるだけで、全然アルファブレンディングではありません。これにアルファ値をかけることでアルファブレンディングとなります。アルファ値を  $\alpha$  とおくと

$$C=D(1-\alpha)+S\alpha$$

これが「アルファブレンディング」です。

ちなみに SrcBlend に BLEND\_SRC\_ALPHA と書いていますが、これは  $S\alpha$  を表しています。

DestBlend に BLEND\_INV\_SRC\_ALPHA は  $1-\alpha$  を表しています。INV は inverse つまり「逆」を表しています。

$\alpha$  の範囲は 0~1 なため  $(1-\alpha)$  の範囲は 1~0 になるわけです。

このため、SrcBlend に SRC\_ALPHA( $\alpha$ ) を DestBlend に INV\_SRC\_ALPHA( $1-\alpha$ ) を指定しています。

ひとまずここまでやれば島風ちゃんが再び表示されます。されますが



まだ昆虫顔(目が真っ黒)ですね。

ということで、さらにやらなければいけないことが、アルファ値そのものも計算しないというまいこと行きません。

それを計算しているのが BlendOpAlpha です。

```
blenddesc.RenderTarget(0).BlendOpAlpha = D3D11_BLEND_OP_ADD;  
blenddesc.RenderTarget(0).SrcBlendAlpha = D3D11_BLEND_ONE;  
blenddesc.RenderTarget(0).DestBlendAlpha = D3D11_BLEND_ZERO;
```

と、書かれていますが、BlendOpAlpha はアルファ値はどう計算するのかってのを表しています。  
現状だと、1 パスの描き切りなのでここは「指定してさえいれば、ぶっちゃけなんでも良い」ん  
ですけど、まあ D3D11\_BLEND\_OP\_ADD にしておきました。

SrcBlendAlpha と DestBlendAlpha も同様です。指定してさえいればなんでも良いです。ですが  
ら…ぶっちゃけこれでもいいのです。

```
blenddesc.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_MAX;  
blenddesc.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ZERO;  
blenddesc.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
```

…なんでも良いってことが、わかるだろう？

でもこれだけでは足りないんですね～。アルファを使うなら

```
blenddesc.RenderTarget[0].RenderTargetWriteMask
```

もしっかり指定する必要があります。これはどの成分をブレンドするかを表しており、

```
D3D11_COLOR_WRITE_ENABLE_ALL
```

を指定すれば色の全要素のブレンドが発生しこうなりますが、



例えば

D3D11\_COLOR\_WRITE\_ENABLE\_RED などと指定すると RED 部分のみがブレンドされ



こうなります。

ところで、この段階になってもまだ昆虫顔の人はシェーダに問題がある可能性があります。もしかして…。

```
return float4(float3(1,1,1)*bright*diffuse*col.rgb, 1);
```

こんな戻り値にしてないでどうか？これでは必ず $\alpha=1$ となりアルファブレンディングの意味がありません。

じゃあこうしましょう。

```
return float4(float3(1,1,1)*bright*diffuse*col.rgb, col.a);
```

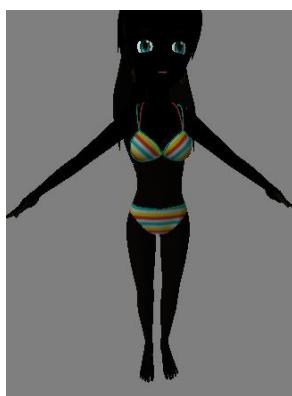
これできちんと表示されるはずです。

### 15.3 リクエストにお答え

それ以外にも上手くいかないパターンがあるみたいなので対応します。

例えば『我那覇響 v1\_グラビアミズギ.pmd』ですが…これを現状のまま表示させると…。

なるほど、さすがは沖縄出身



よく焼けてらっしゃる…ってレベルじゃねーぞ!!

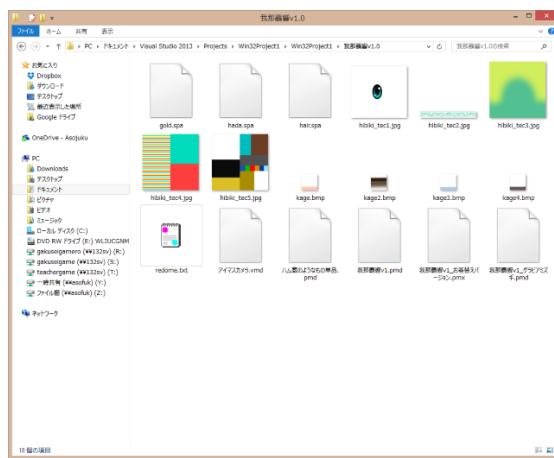
これは沖縄とかそんなレベルじゃなくてセネガルとかザイールだろ!!!

いや…そんな程度じゃ断じてねえ…

これは…まつざきしげるいろだツツ…!!



さて、何故このような状態になっているのか…ファイルを見てみましょう。



おや? これは…

肌のファイルが無くな?



PMD ファイルを見てみましょう…

```
trace_vert_count          00001100
texture_file_name[0]       68 61 64 61 2E 73 70 61 00 FD FD FD FD FD FD hada.spa
texture_file_name[16]      FD FD FD FD
diffuse_color[0]           3F000000 3ED3B646 3E96872B
alpha                   3F800000
specularity             40A00000
specular_color[0]         00000000 00000000 00000000
mirror_color[0]           3F333333 3F1432CA 3ED2BD3C
toon_index                00
edge_flag                  01
face_vert_count          00000B04
texture_file_name[0]       68 61 64 61 2E 73 70 61 00 FD FD FD FD FD FI hada.spa
texture_file_name[16]      FD FD FD FD
...
```

あっ…(察し)

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32](http://blog.goo.ne.jp/torisu_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32)

をちょっと見てください。

- “テクスチャ名.bmp\*スフィア名.sph”で乗算
- “テクスチャ名.bmp\*スフィア名.spa”で加算

という記載がありますね? spa は「加算」なんですね。何に対する加算なんでしょうね…? ココらへんも重要なんですけど…まあディフューズとミラーにそれなりの値が入っていることを考えると、それに対する加算だと考えられますね。

というわけで更に面倒な話ですが、今カラー値を輝度値にかけていますが、ファイル拡張子が spa だった場合は加算しなければいけません。

というわけで拡張子をチェックします。さらに std::string の出番です。後ろから検索してピリオドで分割して、それが“spa”だったらテクスチャは“ない”ものとして扱ってみましょう。

```
int extIdx = texturefilename.rfind('.');
std::string ext = texturefilename.substr(extIdx + 1, texturefilename.length() - extIdx);
if (ext != "spa"){
    result = D3DX11CreateShaderResourceViewFromFile(dev.Device(),
        (FolderPath + texturefilename).c_str(),
        nullptr,
        nullptr,
        &ret->_materials[i].texture,
        nullptr);
}
```



いや…それでも黒くね? アイドルマスターでこれくらい黒いってナターリアくらいだろ!?

というわけで、今使っていないマテリアルデータをさらに使います。どういう事がというと

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32](http://blog.goo.ne.jp/torisu_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32)

ここ見て、mirror\_colorまでを取ってきましょう。

ちなみにスペキュラはちょっとややこしいので、後回しで、環境光だけ影響させましょう。

既に作っている PMDMaterial 構造体に環境光まで追加します。

```
struct PMDMaterial{
    RGBColor diffuse;
    float alpha;//追加
    float specularity;//追加
    RGBColor specular;//追加
    RGBColor ambient;//追加
    unsigned int offset;
    unsigned int count;
    ID3D11ShaderResourceView* texture;
};
```

まあ、この中で追加で使用するのは ambientだけなんですね。

中身は読み込み時に(PMDLoader 内にて)代入してください。

そしてマテリアルバッファの内容も増えますので、そこも変更します。

```
//マテリアルバッファのための構造体
struct MaterialForBuffer{
    XMFLOAT3 diffuse;
    float alpha;//追加
    float specularity;//追加
    XMFLOAT3 specular;//追加
    XMFLOAT3 ambient;//追加
};
```

じゃああとはマテリアルループにて代入すればいいだけです。ここまで来ると流石に memcpy で

```
memcpy(mappedSubresource.pData , &m,sizeof(MaterialForBuffer));
```

で良いと思います。これを memcpy 使わないとしても結局

```
*(MaterialForBuffer*)mappedSubresource.pData= *(MaterialForBuffer*)&m;
```

こんなケツタイなコードになってしまいますからね…

あとはシェーダですけど

```
cbuffer cbuf : register(b1){  
    float3 diffuse;  
    float alpha;//追加  
    float specularity;//追加  
    float3 specular;//追加  
    float3 ambient;//追加  
}
```

合わせてそれぞれ追加していきます。

この中で今回使用するのは diffuse と ambient だけでいいでしょう。

(輝度\*ディフューズ成分+環境光成分)\*テクスチャ成分

が色情報になります。

さらにアルファも有效にしておりますので、

アルファ情報\*テクスチャアルファ

をアルファ情報として使います。

```
return float4((bright*diffuse + ambient.rgb)*col.rgb, alpha*col.a);
```

結果…



だいいぶそれっぽいでしょ? ホントはこれにスフィアマップを加算していくんですけど、そもそもスフィアマップって何なんでしょうか?

## 15.4 スフィアマップ(嘘)

面倒なので真面目にやりませんが、軽く解説しておきます。

スフィアマップとは、環境マッピングの手法の一つで、映り込みとかを表現するものです。

<http://marina.sys.wakayama-u.ac.jp/~tokoi/?date=20050107>

スフィアマップで画像検索するとそれっぽい画像が出てきます。

[https://www.google.co.jp/search?q=sufiamappu&safe=off&source=lms&tbo=isch&sa=X&ved=0ahUKEwi5kJ\\_n-TPAhUl1QKHQiDC3cQ\\_AUICSG&biw=1507&ampbih=848#safe=off&tbo=isch&q=%E3%82%B9%E3%83%95%E3%82%A3%E3%82%A2%E3%83%9E%E3%83%83%E3%83%97](https://www.google.co.jp/search?q=sufiamappu&safe=off&source=lms&tbo=isch&sa=X&ved=0ahUKEwi5kJ_n-TPAhUl1QKHQiDC3cQ_AUICSG&biw=1507&bih=848#safe=off&tbo=isch&q=%E3%82%B9%E3%83%95%E3%82%A3%E3%82%A2%E3%83%9E%E3%83%83%E3%83%97)

DirectX9 の時代はシェーダとか特に書かなくても

<https://msdn.microsoft.com/ja-jp/library/cc324364.aspx>

D3DTSS\_TCI\_SPHEREMAP

を指定すればいい感じに表示してくれました。MMD 自体は DX9 が最初なので、たぶんそれ使ってたんだでしょう。

基本的には肌ツヤ、金属、髪の毛に使用されています。MMD のモデルフォルダに mmd.sph が入ってると思いますので、こいつをコピーして拡張子を bmp にしてみてください。



こんな状態になります。中はこんな感じでツヤツヤしています。テクスチャは別スロットで2つ以上貼れますので、例えば metal.sph をロードします。んで、スロット1にセットします。この場合、GPU 側で区別しなければいけません。コンスタントバッファの場合は b0,b1 でしたがテクスチャは t0 や t1 で表します。



とりあえず、嘘スフィアマップを教えます。そしたらごらんのように金属感が出てきますのでヒントにしてください。

それっぽく見えれば良いんですよ、結局。

ほんまもんのスフィアマップは視線が物体表面で反射して、反射した先の画像を表示するつてのがスフィアマップです。

つまり、視線とピクセルを結ぶベクトルを出して、その反射ベクトルを計算し、その反射ベクトルの方向を元に uv 座標を決定すればいいことになります。これがホントのスフィアマップです。

でも僕はめんどくさい。それっぽく見えればいい。この画像を見てくれ。こいつをどう思う？



すごく…球形…いや、円形です。

これがポイント。この画像……円ですよね？

そこでワシは思いついたのだ…。

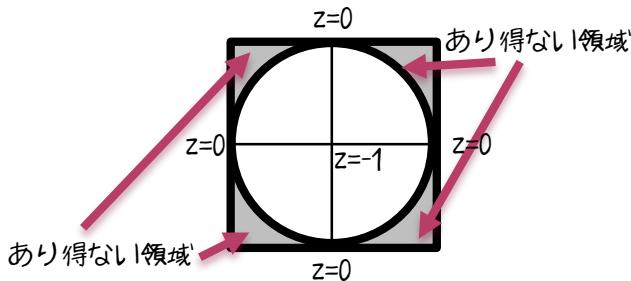
おや…これは法線ベクトルの XY を UV に対して適用すれば「それっぽく」なるんじゃね？

えっ？でもそんなデタラメなことをやつたら、テクスチャの切れ目が見えて見苦しくないの？

大丈夫なのです。

根拠をお話します。

このテクスチャは図のように、正方形の領域に内接するように書かれた円の形をしています。



そしてモデルにおける法線ベクトルを $(nx, ny, nz)$ とすると、正規化済みのため

$nx^2 + ny^2 + nz^2 = 1$ が成り立っています。もし、 $z=0$ で法線が完全にそっぽを向いていたとしても $nx^2 + ny^2 = 1$ が成り立つはずです。つまり円です。更に言うと $z$ が0以外の実数であれば $xy$ は円の内部になるはずです。

つまり、法線の $xy$ を $uv$ に使用しても円の外側に出ることはないと、見た目おかしくはないのです。ただし

sph.Sample([sample](#), outputp.normal.xy);

で良いかというと、そうではありません。よく考えてください。

$$-1 \leq x \leq 1, \quad -1 \leq y \leq 1$$

ですが

$$0 \leq x \leq 1, \quad 0 \leq y \leq 1$$

ですね？まず、範囲が違います。領域をUVに合わせて加工する必要があります。ひとまず言えるのは-1~1は2であり、0~1は1であるということです。合わせてあげるならば得られた $xy$ をそれぞれ2で割りましょう。

$$-0.5 \leq x \leq 0.5, \quad -0.5 \leq y \leq 0.5$$

さらにそれぞれ0.5を足す

$$0 \leq x \leq 1, \quad 0 \leq y \leq 1$$

ねっ？簡単でしょう？

今まで話したことを総合すると sph に出会ったら、テクスチャはスロット 1 番がなんかに入れておいて、それを +1 として受け取る。

そしてテクスチャのカラー値と sph とで乗算する。

```
float4 col = tex.Sample(sample, outputp.uv);
float3 spcol = sph.Sample(sample, outputp.normal.xy/2*float2(1,-1)+float2(0.5,0.5));
return float4((bright*diffuse + ambient.rgb)*col.rgb*spcol, alpha*col.a);
```

あ、Y 方向がひっくり返っているので float2(1,-1) もかけていますね。

というわけでちょっとここまで試してみてください。



さて、そもそもスフィアマップはツヤ系に使用するためのものですが、今回のこの計算の場合は乗算になってしまっているため、黒髪の艶には使用しづらいです。このため後のバージョンから追加されたのが spa です。

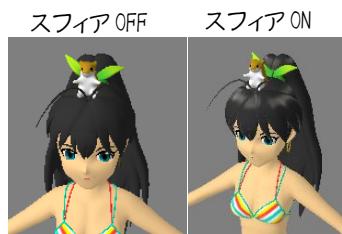
sph は乗算、spa は加算と書いてありましたよね？ それです。

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32](http://blog.goo.ne.jp/torisu_tetosuki/e/ea0bb1b1d4c6ad98a93edbfe359dac32)

きちんと sph、spa を設定できれば

```
return float4((bright*diffuse + ambient.rgb)*col.rgb*spcol+spacol, alpha*col.a);
```

こんな感じで計算すれば



ご覧のようにキューティクルの素晴らしい響ちゃんが表示されます。

## 15.5 Lat式ミクさん奮闘記

ちなみに、リクエストには上がらなかつたが、Lat式ミクについて…。通常通り表示しようとするとこうなります。



本当はかわいいモデルのはずなんですが、相当悲しい状況です。さて…どうしてこうなるのでしょうか。大体予想はつくのですが、ポリゴンが反対向いてますね。ただ、この実験の時はカリング OFFにしてるので反対向きでも表示されるはずではあるので、おかしいです。

<http://dic.nicovideo.jp/a/lat%E5%BC%8F%E3%83%9F%E3%82%AF>

に興味深い記述があります。

### ▣ 構造

このモデルデータは、画面を通して鑑賞(2D出力)されることを考慮し、顔などポリゴンを裏返しに貼るという構造になっている。

これにより輪郭出しがきれいになり、3Dシステムのトゥーンレンダリング機能だけに頼らず自らシェーディングのような効果が得られる。

3Dの写実的で硬質な要素を抑え、漫画的な輪郭黒線や横口の表現を実現している。

なるほど…。

一時期前に「夜明け前のレゾナンス」で話題になった方法と同じだな

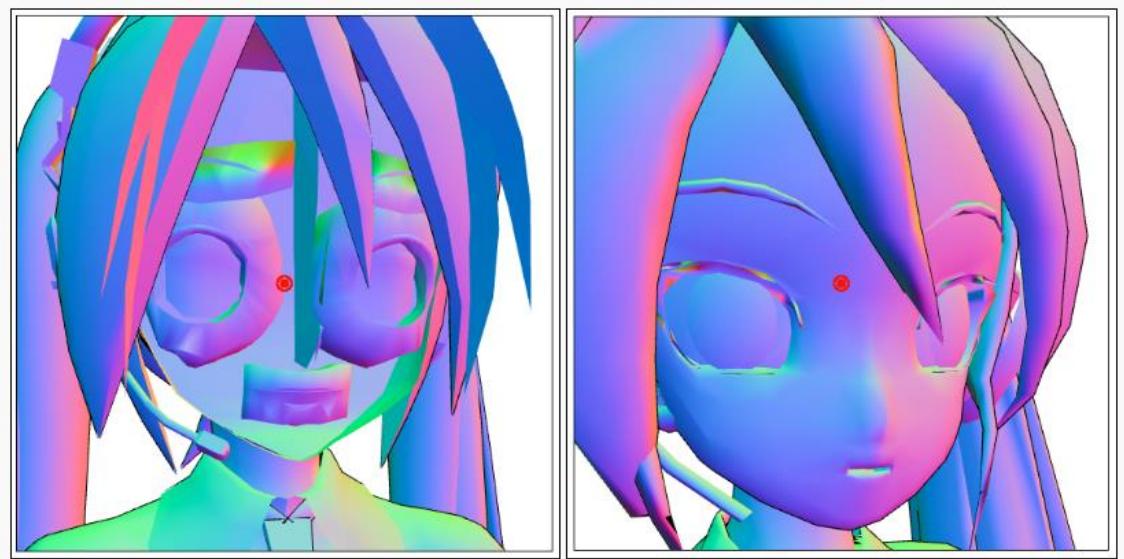
<http://3dnchu.com/archives/yoake-maeno-resonance/>

ていうか、Lat式のほうが早いんだな。

さて、構造は何となくわかった。ではどうすればいい?

<http://d.hatena.ne.jp/edvakf/20111021/1319213629>

いろいろとみてみよう…



まあこういう事だよなあ…つまり顔面表面にはメッシュが張られていない?

裏返しに張かれていると仮定すると、じゃあ試しに法線による明るさ計算を

```
saturate( abs(dot(light, output.normal.xyz)) );
```

このように変更してみた。すると



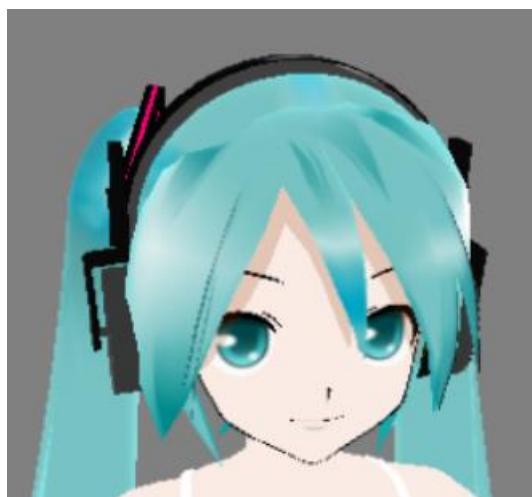
さっきよりかはマシにはなったみたいだが…何だこの黒いの…まだ残ってやがる。

よくわからないので、カーリングをいったんONにしてみよう。



うーん…正解に近づいてる…のか?

そうだ!!試しにシェーディング外してみよう…。



( ^ω^)おっ!!!それっぽい。

とりあえずここまでやってみたことで有効そうなのは

- カーリング ON
- シェーディング OFF

である。全身も見てみよう。



なんか違う…。同じものを MMD で見てみるとこうだ。



なるほど、顔の部分はこれでいいが、体のシェーディングが消えてしまっている…つまり Lat  
式とは

- 顔にシェーディングはかけない
- 体にはシェーディングをかける

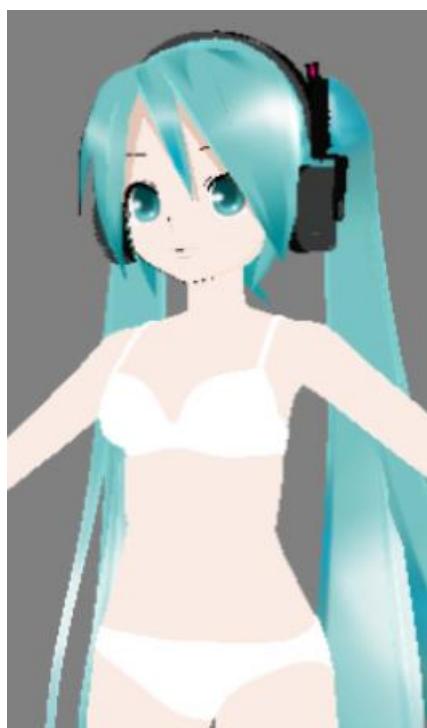
という仕様になっているようだが、特に Lat 式用シェーダがあるわけではない。ということは  
共通仕様のはずだ。

いくら考えても分からなかつたため Google センサーに頼み込むところというサイトが出てきた。

<http://d.hatena.ne.jp/edvakf/20111020>

えっと…表面と裏面で設定変えて二回書くの？正直それはゲームで使うには…というか  
MMDってそんなややこしいことをしてるの？何してんの樋口さん!!

今の僕にはここがええとこですね



すまんが今の僕には技術的にも時間的にもここがいつけないつぱいだ。

課題…10/28(金)までに PMD メッシュモデル(君の嫁)を表示してください。

提出場所は

[¥¥132sv¥gakuseigame¥game¥b43PG](#)

です。

ああ……早くもこの時が来てしましました来てしましたね…皆さんスピードの速さに僕も戦慄しています…。



ついにきました。

そう…「ボーンをいじる時」が…テンポ早すぎイ!!

トゥーンを先にしたくもあるんですけど、必須具合で言うとこっちが重要。さあ時は来た。

さあ…

地獄入ようニマ

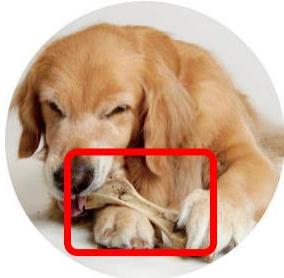
早かったなあ…ホント。最高記録ちゃうやろか。君らマジ優秀。

## 16 ボーン

ここからはモロ数学です。行列です。そういうことです。よろしく。さらなる概念であるクォータニオンもあるんやで

### 16.1 そもそもボーンてなんだ？

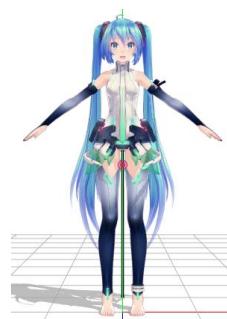
ボーン(bone)てのは、骨やで？



いやマジで。フツーに骨。これがMMD上ではこのように表現されたります。

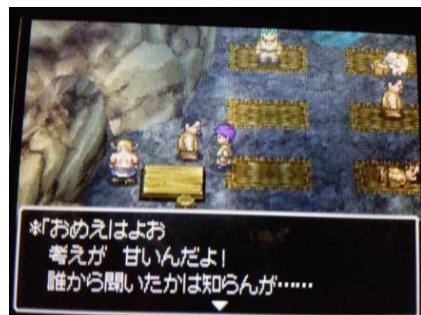


ミクさんの表面に◎▶がたくさんあるやろ？これが「ボーン」なんよ。モデルデータというのはボーンを入れて、そしてそのボーンを動かさない限りポーズは変わらへんね。



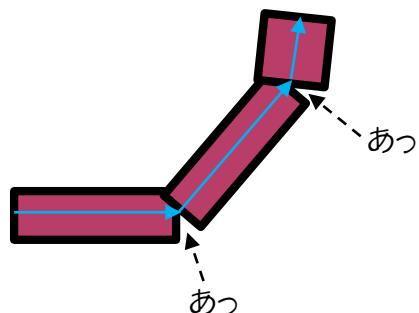
こつからはその「ボーン」をいじることでポーズを付けて行くことを目指してみましょう。

## 16.2 ボーンをいじるだけでいいのか？



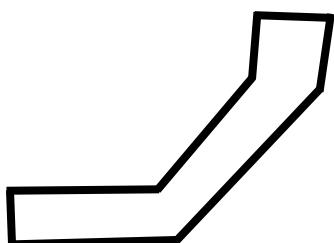
実はそうは行かない。ボーンを動かしたところで、それに連動して「頂点」が動かないことに話にならない。

更に言うと、きれいに「関節らしく」動かすためには1つの頂点に1つのボーン対応…なんてやつると、つまるところ



このように隙間が空いたり無意味に重なったりする(重なるとZファイティング等が発生して望ましくない)。

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、自然な感じでポージングできるようになるわけです。

これらの機能を「スキニング」と言い、これをもとにアニメーションすることを「スキンメッシュアニメーション」と言います。

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。

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/5a1b16e2f61067838dfc6bd010389707](http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b16e2f61067838dfc6bd010389707)

で、頂点のことを思い出してほしいんですけど、1頂点が複数のボーンの影響を受ける場合に、それぞれの「影響度」を設定しておく必要があります。

つまり、とある頂点  $V(x, y, z)$  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、それは言っても、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。

## 16.3 ボーン情報の取得

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/b38463f52d0adbca1c46fd315a9b17d0](http://blog.goo.ne.jp/torisu_tetosuki/e/b38463f52d0adbca1c46fd315a9b17d0)

う～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。

## 16.4 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んしません。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？

懸命な皆さんなら、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れないなあ。

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).tail\_pos\_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).tail_pos_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つ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。

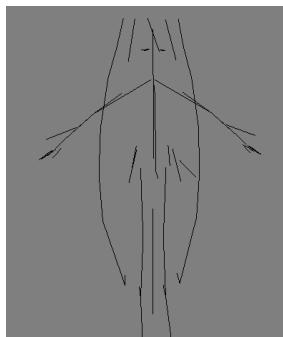
つまり

```
//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
};
```

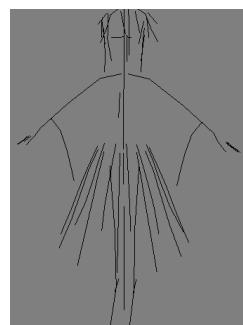
と、そして表示したい情報はとりあえず座標だけなのでひとまず

```
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ろなります。



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0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(XMFLOAT3) でいいでしょう)

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ないため、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。

## 16.5 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

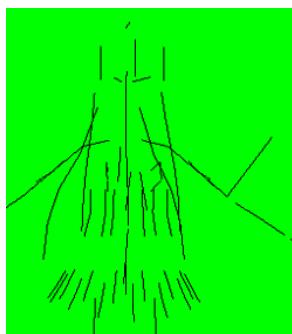
②「左ひじ」のテール位置を、ヘッド位置を中心に回転させる

③頂点再セット

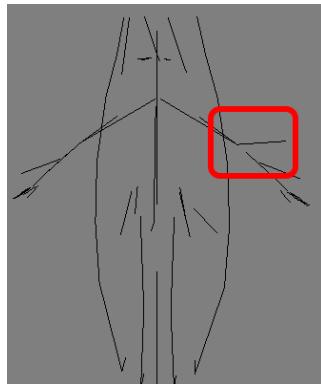
このようにするためにボーン頂点バッファを変更可能にしておきましょう。

```
bdesc.Usage = D3D11_USAGE_DYNAMIC;  
bdesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;  
bdesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
```

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;
```

これで名前からインデックスを検索することができます。

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これます。

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すが、ロードとストアの対応を取っているためです。(アセンブラーで `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、定義へジャンプすると、行列みたいな扱いであることわかれります。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、関数を解することで XMFLOAT3 とのやりとりができますよってこと。

面倒ですね。まあ仕方ありません。

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、

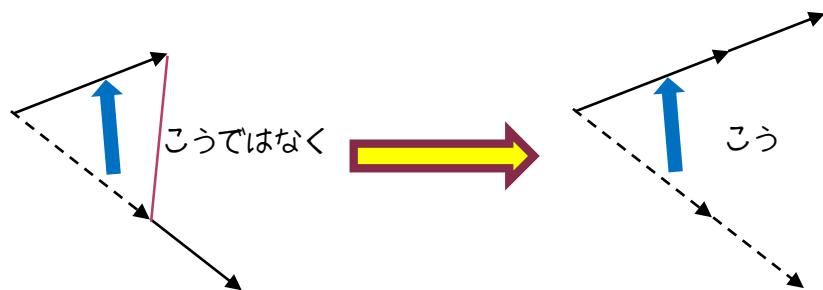


こんな感じだったんですけど…まあどっちにしても痛ましいですね。

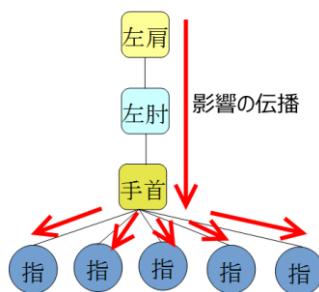
何故かわかりますか？それはこの構造がまだツリー構造になっていないからです。

## 16.6 ボーンツリー

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

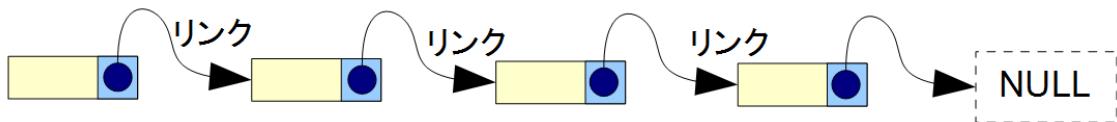
こういう感じですね。

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…

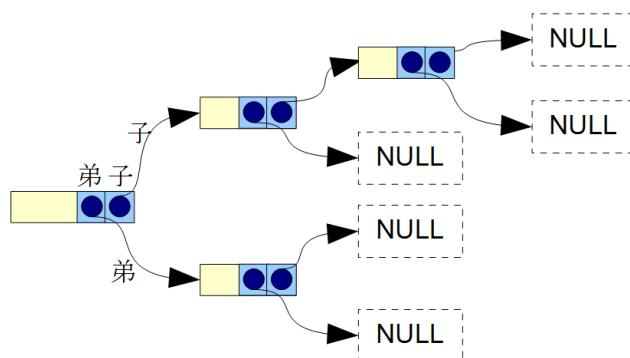
[http://ja.wikipedia.org/wiki/%E6%9C%A8%E6%A7%8B%E9%80%A0\\_\(%E3%83%87%E3%83%BC%E3%82%BF%E6%A7%8B%E9%80%A0\)](http://ja.wikipedia.org/wiki/%E6%9C%A8%E6%A7%8B%E9%80%A0_(%E3%83%87%E3%83%BC%E3%82%BF%E6%A7%8B%E9%80%A0))

リンクとだいたい同じなんですが、リンクは一つのノードから一つのノードへ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 とかあるので

```
struct BoneNode{  
    int index;//インデックス  
    std::vector<BoneNode> node;  
};
```

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。つまり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。

### 16.6.1 ツリー反映の準備

なので、先程 head と tail を作りましたが、同じ数だけの matrix を入れられるようにします。

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。

あと、余談ですが std::には algorithm ってのがあってですね。その中の fill って関数を使うと

```
std::fill(_bonematrixes.begin(),_bonematrixes.end(), XMMatrixIdentity());
```

この一行で全てのボーンマトリクスを XMMatrixIdentity()にしちゃえるんですよ。こういう便利なものをまとめた #include<algorithm> ってのがあります。この手のやつ全部覚えるのは大変なので、今は「そういうのもあるのか」くらいに思っておきましょう。



こんなノリで

アルゴリズムに関しては機会があつたらまた解説します。

これでボーン情報には全て単位行列が入っている状態です。前にも言ったようにここに目的の行列を入れます。

何も加工しなければ XMMatrixIdentity()のままでいいのですが、ここに「左ひじを曲げる」という操作を追加しようとするならば当然左ひじ以降は全て「左ひじを曲げる」の影響を受けます。

### 16.6.2 ツリー構造の構築

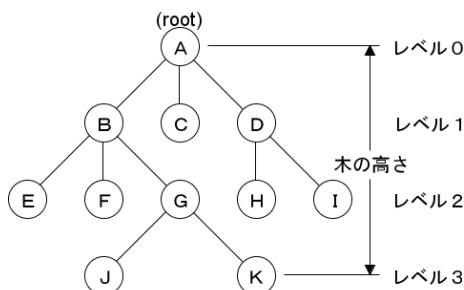
「左ひじ以降」というのはツリーにおける左肘から先のノードのことです。

おっと、まだツリー構造を構築してなかったですね。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C:¥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<BoneNode> child; //自分の子どもたち
};
```

```
std::vector<BoneNode> _bonenodes;
```

ぶっちゃけた話、`std::vector<BoneNode>`の中に `std::vector<BoneNode>` があつたりして、もう混乱の元になりそうですが、今回の `BoneNode` はただ単に「自分の子供達を管理する」構造体だと思ってください。そしてそれが **全てのボーンに割り当たっている**。つまり全てのボーンがこれを持っている…そう思ってください。

きれいなツリー構造ならば実は `_bonenodes[0]` 以外いらなくなるんですけどね。

(※きれいなツリー構造ってのは、必ずルートから末端までが一つの道で示される…が PMD の構造はモデルによってはちょっと怪しい…ように見える。検証はしていない。)

はい、`_bonenodes` はボーン数分確保してればいいです。その上でロードの際に

```
if (b.parent_bone_index != 0xffff){  
    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「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);  
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。

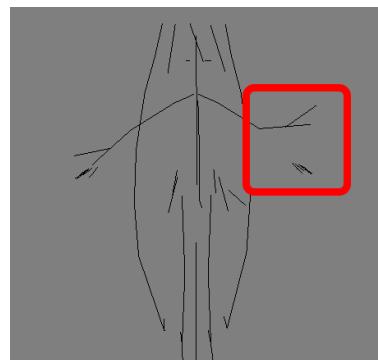
実際に変形させる時には

```
//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



手首までは運動してますが、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。

### 16.6.3 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。

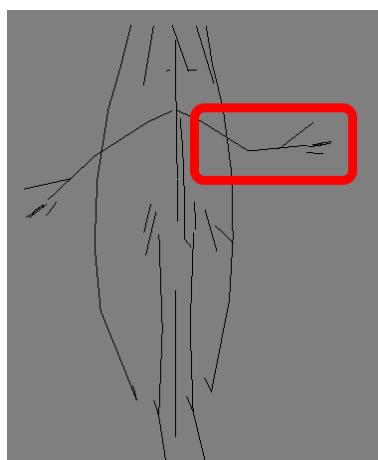
今回の場合次を呼び出す条件は「子がいる」ってことですね。

つまり

```
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ありませんからね。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。

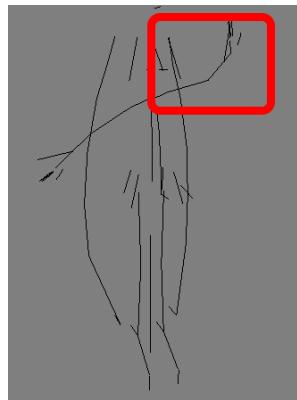


自分で考えて、末端までボーンを回転させてください。

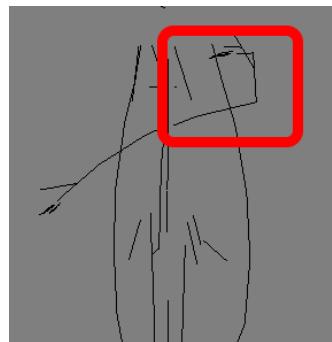
できましたか？ここでちょっと待つよ？

とは言え、ここでは終わらんのだ。

例えば…そうさのう…肩を  $45^{\circ}$  回転させて、肘を  $45^{\circ}$  回転させて、手首を  $45^{\circ}$  回転させてみましょ。

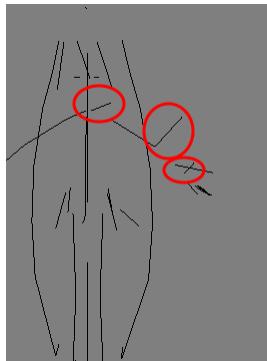


ちょっとわかりづらいですね…。肘と手首を  $90^{\circ}$  にしてみましょ。



それぞれ適切に曲がっているのが分かると思います。

ところが、これを適切にプログラムしないと…。



こんな感じになります。

どういう事がというと、子の座標変換は、それ自体が親行列の影響を受ける必要があるんです。分かりますかねえ…肘も、手首も、肩から出てるわけですよね？つまり型の影響を受ける…それは座標自体もそうなんですが、「座標変換」つまり行列自身が親の影響を受けるのです。

実はここまで来るとまたちょっとややこしい話になってくると思いますが、行列の乗算を行う必要があります。

そもそも左ひじの時はどうしてましたつけ？

そう、`head_pos` を原点に戻し→回転→元の座標に戻す行列をそれぞれにかけてあげるんでしたね？

で、これは…親ボーンの行列を子ボーンに乗算してやることで、肩を回転→肘を回転→手首を回転…となるわけです。

以上のこととは再帰の時にやってあげれば以外に簡単に実装できます。

```
bonematrixes(idx) = bonematrixes(idx) * bonematrixes(index);
```

と、このように乗算してやればいいだけです。それを再帰しながら…つまり

```

//ヘッドとテールの座標の変換
XMVECTOR headpos = XMLoadFloat3(&bones(index).headpos);
XMVECTOR tailpos = XMLoadFloat3(&bones(index).tailpos);
headpos = XMVector3Transform(headpos, bonematrixes(index));
tailpos = XMVector3Transform(tailpos, bonematrixes(index));
XMStoreFloat3(&bones(index).headpos, headpos);
XMStoreFloat3(&bones(index).tailpos, tailpos);

//子へ伝播
for (auto& idx : nodes(index).childIndices){
    XMVECTOR hpos = XMLoadFloat3(&bones(idx).headpos);
    XMVECTOR tpos = XMLoadFloat3(&bones(idx).tailpos);
    bonematrixes(idx) = bonematrixes(idx) * bonematrixes(index);
    RecursiveApplyBone(nodes, idx, bones, bonematrixes);
}

```

ここまでうまくやれば、適切にボーンが変形されるでしょう。再帰なので確実に言えることは、処理を「関数化」しなければならないです。ここにいる皆さんなら大丈夫だとは思いますが、「ラクラクカンタン」ではありませんので、それなりに頑張ってください。

## 16.7 アッター！！

すみません。忘れてました。なんか実行時に変なバグが起きてる人が多いようです。それは

<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いといけない**のですが、new で動的に作ったものだと問題が起こります。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。」

### 16.7.1 どういうことなの？

うーん。

[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「クラッショウ」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 XMATRIX 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」ではないため、メンバ変数に配置した時点で規約違反してるんですよね。

### 16.7.2 対処法

つまるところ、今回の件で言うと

mesh->BoneMatrixes(0)～

を演算で使用しちゃダメってこと。

というわけで RecursiveApplyBone 内の

```
bonematrixes(idx) = bonematrixes(idx)*bonematrixes(index);
```

これがマズかったんです。

まどろっこしいですが

```
XMMATRIX bone = bonematrixes(index);
```

```
XMMATRIX mat = bonematrixes(idx);
```

```
mat=mat*bone;
```

```
bonematrixes(idx) = mat;
```

とする必要があるというわけです。非常に面倒ですけど仕方ないですね。

### 16.7.3 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](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、「一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)一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…

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

このレベルで計算されます。

ともかく 16 バイト分いつぱ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。

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 XMMATRIX 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だけでどうか?

さて、終わってないですよ?

で、このボーン情報…何に使うと思しますか?

## 17スキニング

### 17.1 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/5a1b1be2fb10b7838dfcb6bd010389707](http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b1be2fb10b7838dfcb6bd010389707)

bone\_weightって言うのが影響度ではあるんですが、今回使用するのはそれではなく、  
bone\_numの方を使用します。

WORD bone\_num(2); // ボーン番号 1、番号 2

うん、分かりますね。

### 17.2 最初の困難

そういうことなのでレイアウトには

```
{ "BONENO", 0, DXGI_FORMAT_R16G16_UINT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_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。

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?

かといって、shortなどという型は HLSL にはまだないんだ…どうしよう…。ということでここでいきなり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、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。

### 17.3 影響度

次に影響度だ。再び

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/5a1b1be2fb10b7838dfcb6d010389707](http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b1be2fb10b7838dfcb6d010389707)

を見てくれ。次の1バイトが影響度だ。

これは割と素直に渡してしまっていい。

{ "WEIGHT", 0, DXGI\_FORMAT\_R8\_UINT, 0, D3D11\_APPEND\_ALIGNED\_ELEMENT, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

ちなみに言うと、俺もここまで試行錯誤の結果だ。答えがどうかにあったわけじゃない。

で、このウェイトもちょっとやつかれだ。

『min:0 max:100 // ボーン 2 への影響度は、(100 - bone\_weight)』

というわけだ。

うーん。そこまで厄介でもないか。簡単だよね？ 100で割れば良いんだ。

```
float wgt1=float(weight)/100.0;
```

```
float wgt2=float(100-weight)/100.0;
```

これだけの話。大した話じゃない。あとはこれをそれぞれ頂点に対して乗算してやればいい。

さて、ここで問題になってくることがある。

それは

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。どうしたらいいんでしょうか！？



うん、もうね考えるのめんどくさいから全部投げちゃえ YO!!!

こうなつたらそれしかないんだぜベイベー。全部…全部なんだ。つまり

ボーン 256 個(512 くらい)の必要もあるがひとまず決め打ち)でコンスタントバッファを作成  
→GPUに投げる→スキニングなんだ!!!

## 17.4 ボーン配列(256 個)用コンスタントバッファ作成

…まあGPUに送りつける情報の数がごつつい増えただけの話です。コンスタントバッファは何のかんの言っても「コンスタント」バッファなので、可変長にすることはできません。だから今回は256個に固定してボーン情報を送りつけます。

`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？おもつ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…。`

取り越し苦労はしても仕方ないので、ちゃっちゃと作っちゃいましょう。

コンスタントバッファ自体の作り方は、もう分かりますよね？忘れちゃった人はワールドだのビューだののバッファを参考にしてください。そして 256 個ぶんバッファ作るのです。

で、セットする時には 2 番にセットするようにしましょう（0 番はワールドビュープロジェクション、1 番はマテリアルに使用されているため）

なお、boneinitdata のところだけヒントを書くと

```
boneinitdata.pSysMem = &mesh->BoneMatrixes()(0);
```

こんな感じです。これを 2 番スロットに送ってください。

## 17.5 頂点シェーダ側での処理

もちろん、頂点シェーダ側ではこれを受け取る用意をしなければなりません。賢明な皆様ならもうお分かりかと思いますが `register(b2)` を使います。

ところで今回のように大量のマトリクスを渡す場合はどうすれば良いのでしょうか？

```
matrix boneMats(256);
```

とかで良いのでしょうか? Bingo!! その通りでございます。予想通りです。

といふか、考る前にやってみましょ… こういふのは。

さてこれでオッケーなことがわかつたんですが、これはボーン行列です。つまり頂点への座標変換を行うためになります。

つまり頂点に乗算をする必要があります。

順序としては

ビュープロジェクション×ワールド×ボーン行列の順序で乗算していきます。

あ、ここで一つ言っておくことがあります。

数学では行列は、変換の順序を左から左からかけていくのでした。つまり

AしてBしてCしてDという場合ならば

D×C×B×Aでした。

しかし DirectX の場合だと

A×B×C×Dとなります… では HLSL の場合はどうなるんでしょうか? そう

D×C×B×Aとなります。

ややこしいですね。

一応理由は MSDN にも書いてますが、

[https://msdn.microsoft.com/ja-jp/library/bb509634\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509634(v=vs.85).aspx)

この「行列の順序」ってところで書かれていますが、要は XMATRIX 側の「パッキング」はデフォルトでは行優先で、HLSL 側の「パッキング」は列優先なわけです。

<https://msdn.microsoft.com/ja-jp/magazine/dn781365.aspx>

を見ると思はわかります。XMATRIX 側は初心者にも優しく行優先で格納しているのですが、HLSL は硬派に列優先になっています。

見ると

「いずれにしても DirectX とシェーダコードには互換性がありません」と明記されているので仕方ないです。

ちなみに、ディレクティブを使用すれば行優先一列優先も変更できますが、寧ろわけわからなくなるので

DirectX は「右にかけていく」

hlsl は「左にかけていく」

と思っておきましょう。

さて、そうなると先程も言ったようにボーン行列をかけるのは最後です。何故だかわかりますか？想像してください。ワールドだのの影響を受けたあとでボーンを動かすと体中ばらばらになるからです。やってみてもいいですけど…



こうなります(ニッコリ)

といふわけで、ボーン行列が最後に書けられるようにします。ところでボーンは 256 個あるわけですが、どれを使用したら良いんでしょうか？

前にも言いましたね？それはボーン ID に入っていて、そして 2つ分入っていると…。ですから 0xffff と論理積(&)をとります。

```
m = mul(m, boneMat(boneid & 0xffff));
```

さて…実行してみてくれ。どうなるかな？



うむ。曲がり方は間違ってない。間違ってないんだが…なんかカクカクしてるよね？これは前にも言った「影響度」による「重み付け」が必要なんです。

影響する「ボーン」は2つありますよね？そう…もう一つを見つけなきゃいけない。これも前に言ったと思いますが、

```
boneMat((boneid & 0xffff0000) >> 16)
```

を使用する必要があります。

## 17.6 重み付け加算

さて、ここで漸く「影響度」が関わってきます。

```
float wgt1=float(weight)/100.0;  
float wgt2=float(100-weight)/100.0;
```

これ。以前に言いましたよね？これを使用します。どうやるのかというと、これはよくあるアレ…アルファ乗算のときと同じだと思ってください。

$$C = \alpha S + (1 - \alpha) D$$

そう…ここでめったに使用されない演算「行列の和」と「行列のスカラー倍」が使用されます。

つまり先程まで使用していたボーンを

```
bone1*wgt1+bone2*wgt2
```

すればいいというわけです。まあこんな単純というわけには行かないですがやってみてください。



こうなれば正解です。

え？袖が飛び出してるじゃん。と思われると思いますが、これは本家でも



こうなっているので仕方ないのです。今回は「本家再現」が目標です。

ちなみに今回は  $wgt+1$  と  $wgt+2$  という風に分けていますが、変数がもつたいないという人は別に  $wgt$  と  $(1-wgt)$  で構いません。

ちなみに靈夢さんでもこの通り



物理が入っていないので、袖はそのまま回転します。

ちなみに我那覇ちゃんもこの通り、



## 17.7 後から思ったんだけど…

後から思ったんだけどレイアウトを

```
{ "BONENO1", 0, DXGI_FORMAT_R16_UINT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },  
{ "BONENO2", 0, DXGI_FORMAT_R16_UINT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
```

として頂点シェーダ側を

```
uint boneid1 : BONENO1, uint boneid2 : BONENO2,
```

にしたらどうかな?うまくいくかな?と思った。

結果…ダメでした。

やっぱりダメなんだなあ…最初に思いついたのがベストってこともあるんだなあ…。とはい  
えプログラマというのは「検証」は大事です。センサーである僕もいつも「検証」してますの  
で、ガクサーの皆さんにはセンサー以上に「検証」しておきましょう。

## 18 そろそろまたコード整理

そろそろコーディングがひどいことになってきたと思いますので、整理しましょう。

久しぶりですねえ。

多分、現在のところだと main 関数の行数が 300 行を超えるくらいになっているのではないかと思うか？

### 18.1 リファクタリング

今回は「説明のしやすさ」「理解のしやすさ」に重点を置いてるので、なかなかコーディングにまでは話が来なかつたと思います。結果としての今の状況なのでしょう。

本来はもう少し最初からクラス設計してやっていくべきかもしれません。それだと DirectX について…3D についての理論の理解と技術力を阻害します。ある程度できたら整えていくほうがこの場合は良かったと思います。

などと、言い訳じみてますが「リファクタリング」すれば、後からキレイにすることも可能です。もちろん最初からできるだけキレイに書きたいものです。



現状、この有様です。

「コードの長さ」が問題である場合、最初にやるべきことは「関数化」ですね。可能な限り「機能」で分割していくと良いのですが、今回はもう機械的にぶつた切っちゃいましょう。



このように「文字が認識できないレベル」まで小さくし、全体を把握できるようにして考えるのもコードを整理するためのヒントです。印刷してボールペンで印をつけるのもアリですね。

とりあえず機械的に関数化します。このとき、ローカル変数にすべきものとグローバル変数にすべきもの、引数として渡すものを考えましょう。頑張れば…

```

int WINAPI WinMain(HINSTANCE instance, HINSTANCE, LPSTR cmdline, int){
    HRESULT result = S_OK;
    HWND hwnd = InitWindow();
    deviceDx11* device = deviceDx11::Instance();
    device.Init(hwnd);

    if (FAILED(InitDirect3D(hwnd))){
        return 0;
    }

    PMDLoader loader;
    //PMDmesh* mesh=loader.CreatePMDMesh("reimu_F01.pmd");
    //PMDmesh* mesh = loader.CreatePMDMesh("博麗靈夢/reimu_F01.pmd");
    PMDmesh* mesh = loader.CreatePMDMesh("我那歌姬V1.0/我那歌姬V1_グラビアミズギ.pmd");
    //PMDmesh* mesh = loader.CreatePMDMesh("miku/初音ミク.pmd");

    unsigned int offset = mesh->VertexOffset();
    unsigned int stride = mesh->VertexStride();
    ID3D11Buffer* vb = mesh->GetVertexBuffer();
    ID3D11Buffer* ib = mesh->GetIndexBuffer();
    device.Context()->IASetVertexBuffers(0, 1, &vb, &stride, &offset);
    device.Context()->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    device.Context()->IASetIndexBuffer(ib, DXGI_FORMAT_R16_UINT, 0);

    //シェーラード&コンパイル
    ID3DBlob* compiledShader = nullptr;
    ID3DBlob* shaderError = nullptr;

    //頂点シェーダーの作成
    ID3D11VertexShader* vs = nullptr;
    ID3D11InputLayout* inputLayout = nullptr;
    result = VertexShaderCompile(&vs, &inputLayout);
    device.Context()->IASetInputLayout(inputLayout);

    //ボーン頂点シェーダーの作成
    ID3D11VertexShader* bonevs = nullptr;
    ID3D11InputLayout* boneLayout = nullptr;
    result = BoneVertexShaderCompile(&bonevs, &boneLayout);

    //ピクセルシェーダーの作成
    ID3D11PixelShader* ps = nullptr;
    PixelShaderCompile(&ps);

    WorldAndCamera woc = {};
    //変換行列の作成
    ID3D11Buffer* wvpbuffer = nullptr;
    result = CreateWVPConstantBuffer(&wvpbuffer, woc);
    device.Context()->VSSetConstantBuffers(0, 1, &wvpbuffer);

    ID3D11Buffer* materialBuffer = nullptr;
    result = CreateMaterialBuffer(&materialBuffer);
    device.Context()->PSSetConstantBuffers(1, 1, &materialBuffer);

    device.SetupRasterizer();
    device.SetCulling(false);

    //左ひじ変形
    DeformBones(mesh);

    D3D11_MAPPED_SUBRESOURCE bonemappedSubresource = {};
    device.Context()->Map(mesh->GetBoneBuffer(), 0, D3D11_MAP_WRITE_DISCARD, 0, &bonemappedSubresource);
    memcpy(bonemappedSubresource.pData, &mesh->Bones()[0], mesh->Bones().size()*sizeof(PMDmesh::Bone));
    device.Context()->Unmap(mesh->GetBoneBuffer(), 0);

    ID3D11Buffer* boneMatrixBuffer = nullptr;
    CreateBoneMatrixesBuffer(mesh, &boneMatrixBuffer);
    device.Context()->VSSetConstantBuffers(2, 1, &boneMatrixBuffer);

    SetBoneBuffer(mesh, bonevs, boneLayout);
    SetVertexBuffer(mesh, vs, inputLayout);

    bool resultShowWindow = ShowWindow(hwnd, SW_SHOW);
    MainLoop(mesh, woc, vs, bonevs, ps, inputLayout, boneLayout, wvpbuffer, materialBuffer);
}

```

メインが100行以内になりました。なお、MainLoopの引数とかがとんでもないことになっているので、いざれどうにかしなければなりませんが…

## 19 ポージング

ついにやってきましたよ。ポージングです。アニメーションの前段階です。

で、MMDにはポーズデータという「\*.vpd」ってデータがあるんですが、こいつはテキストファイルなんですよね。正直面倒なので、VMDファイルでポーズを取らせようと思います。

¥¥132sv¥gakuseigamero¥rkawano¥MMD

に VMD.SYM があります。これを自分の TSXBIN.exe のフォルダに入れて、VMD ファイルを開いてみましょう。

eader.VmdHeader[0]	56 6F 63 61 6C 6F 69 64 20 4D 6F 74 69 6F 6E 20 Vocaloid Motion
eader.VmdHeader[16]	44 61 74 61 20 30 30 30 32 00 00 00 00 00 Data 0002
eader.VmdModelName[0]	8F 89 89 B9 83 7E 83 4E 00 FD FD FD FD FD FD 初音ミク .....
eader.VmdModelName[16]	FD FD FD FD .....
otion_count	0000008C
otion[0].BoneName[0]	83 5A 83 93 83 5E 81 5B 00 FD FD FD FD FD センター .....
otion[0].FlameNo	00000000
otion[0].Location[0]	00000000 00000000 00000000
otion[0].Rotatation[0]	00000000 00000000 00000000 3F800000
otion[0].Interpolation[0]	14 14 00 00 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B .. ..kkkkkkkk
otion[0].Interpolation[16]	14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 00 .. ..kkkkkkkk
otion[0].Interpolation[32]	14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 00 00 .. ..kkkkkkkk
otion[0].Interpolation[48]	14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 6B 00 00 00 .. ..kkkkkkkk
otion[1].BoneName[0]	8F E3 94 BC 90 67 00 FD FD FD FD FD FD 上半身 .....
otion[1].FlameNo	00000000
otion[1].Location[0]	00000000 00000000 00000000
otion[1].Rotatation[0]	00000000 00000000 00000000 3F800000

ご覧のとおりです。今回重要なのは「ボーン名」と「Rotation」です。

ところでこの Rotation…何に対する回転なんでしょうね？

ここで今までの Rotation に関する考え方を変えなきゃいけないんですが、準備はいいかな？

今までみなさんは X 軸回転、Y 軸回転、Z 軸回転のような回転しかしていませんでした。

その回転だけで、全てがまかねると思ってますか？

残念ながらそうは行かないんですね…。理由は後からぼちぼち言いますけど、「任意軸回りの回転」というものを使用します。

で、これも通常通りの回転でやってもいいんですが、それだと

### ( $n_x, n_y, n_z$ ) を軸に $\theta$ だけ回転する場合

$$\begin{pmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_xn_y(1-\cos\theta)-n_z\sin\theta & n_zn_x(1-\cos\theta)+n_y\sin\theta \\ n_xn_y(1-\cos\theta)+n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_yn_z(1-\cos\theta)-n_x\sin\theta \\ n_zn_x(1-\cos\theta)-n_y\sin\theta & n_yn_z(1-\cos\theta)+n_x\sin\theta & n_z^2(1-\cos\theta)+\cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

まあこんな式があるんですけど残念ながらシンプルじゃないんですね。

ということで「クオータニオン」という数学的な概念が利用されます。

## 19.1 クオータニオンとは

<https://ja.wikipedia.org/wiki/%E5%9B%9B%E5%85%83%E6%95%B0>

[http://marupeke296.com/DXG\\_No10\\_Quaternion.html](http://marupeke296.com/DXG_No10_Quaternion.html)

クオータニオンの話をする前にちょっと聞いておきましょう。皆さんには「虚数」というのを知っていますでしょうか？

### 19.1.1 複素数のおさらい

そう、2乗するとマイナスになる数です。大抵の場合は虚数を Imaginary Number の略で  $i$  とおきます。聞いたことはありますね？

そしてこの「虚数」と「実数」を組み合わせたものを「複素数」と言います。

複素数は

$$z=a+ib$$

と表されます。複素数のことを英語で Complex Number と言います。ちなみに「実数」は Real Number です。 $a$  の事を「実部」と言い、 $b$  の事を「虚部」と言います。

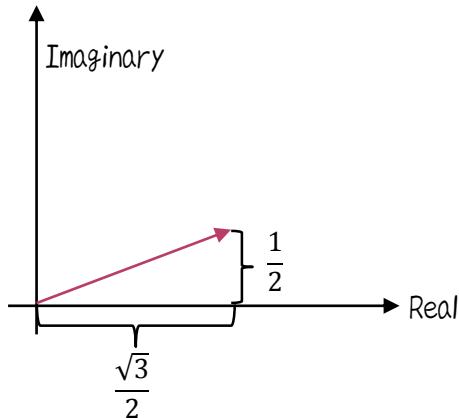
で、こいつですね。面白いことに「回転」を表すことができるんですよ。

Real を横軸に、Imaginary を縦軸に考えてください。

そうすると  $30^\circ$  回転とは

$$\frac{\sqrt{3} + i}{2}$$

となります。

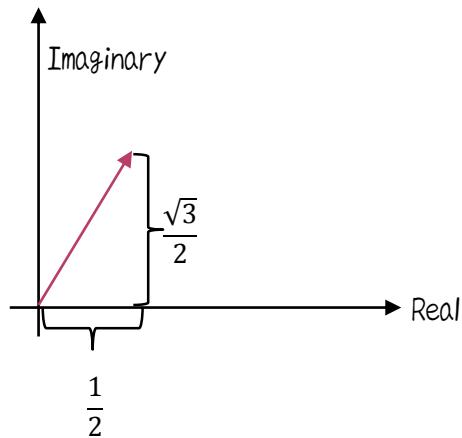


この面白いところは「乗算」すると回転が進むことがあります。

例えば、 $30^\circ + 30^\circ = 60^\circ$  だが、これは複素数を使うと

$$\left(\frac{\sqrt{3}+i}{2}\right)\left(\frac{\sqrt{3}+i}{2}\right) = \left(\frac{\sqrt{3}^2 + 2\sqrt{3}i + i^2}{2^2}\right) = \frac{(3-1)+2\sqrt{3}i}{4} = \frac{1+\sqrt{3}i}{2}$$

つまり  $60^\circ$  ということになります。



お分かりいただけただろうか？これが複素数である。一般化して言うと

$$Z = \cos \theta + i \sin \theta$$

なわけだ。なんで急にこういう話をしたのかといふと、クオータニオンってのが複素数を 3D に拡張したものだからだ。

ともかく、複素数を用いて回転を表すことができることはわかりましたね？

### 19.1.2 四元数

クオータニオンは3Dに拡張された複素数なので、以下のように表します。

$$Q = iX + jY + kZ + w$$

です。なお、 $i, j, k$  はどれも次元の異なる虚数記号で

$$i^2 = j^2 = k^2 = -1$$

$$ij = k, jk = i, ki = j$$

$$ijk = -1$$

が成り立つような「虚数？」と定義されます。クオータニオンの場合  $w$  を実部といい、 $i, j, k$  を虚部と言います。Unityにおける「回転」を表す Quaternion

こいつがモノごつい便利なのは「任意軸回りの回転」をさっきの式よりかはシンプルに記述することができる点です。

## 19.2 クオータニオンを使用して任意軸回転する

まず任意ベクトル  $V$  周りの  $\theta$  回転のクオータニオンは

$$Q = \left( \frac{V_x \sin \theta}{2}, \frac{V_y \sin \theta}{2}, \frac{V_z \sin \theta}{2}, \frac{\cos \theta}{2} \right)$$

と表されます。VRM データの場合、4つの数値が指定されていますが、それはこういう並びだと思ってください。ともかくクオータニオンで格納されているのはありがたいことです。

このクオータニオンを回転行列にするには

$$\begin{pmatrix} Q_x^2 + Q_y^2 - Q_z^2 - Q_w^2 & 2(Q_y Q_z - Q_x Q_w) & 2(Q_y Q_w + Q_x Q_z) \\ 2(Q_y Q_z - Q_x Q_w) & Q_x^2 - Q_y^2 + Q_z^2 - Q_w^2 & 2(Q_y Q_w - Q_x Q_y) \\ 2(Q_y Q_w - Q_x Q_z) & 2(Q_z Q_w + Q_x Q_y) & Q_x^2 - Q_y^2 + Q_z^2 - Q_w^2 \end{pmatrix}$$

こういう回転行列を作ります。別にシンプルではないですね。それでもクオータニオン使う理由はジンバルロック防止と球面線形補間のためです。

クォータニオンに関しては「道具」と割り切ってください。いつもは「理屈」をしつかり教える僕ですが、こればっかりは「理屈」を知るのはコストが悪いです。

何故かと言うと、「テンソル」「スピノール」などの概念を知る必要が出てくるからです。  
ちょっとやめておきましょう。

さて、今回のMMDのデータに関しては既にクォータニオンで入っているので、これを利用するには

XMMatrixRotationQuaternion関数を利用します。

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.matrix.xmmatrixrotationquaternion.aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixrotationquaternion.aspx)

ま、それはともかくひとまずはロードしましよう。

### 19.3 ポーズデータのロード

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/bc9f1c4d597341b394bd02b64597499d](http://blog.goo.ne.jp/torisu_tetosuki/e/bc9f1c4d597341b394bd02b64597499d)

を見ましょう。

```
struct VMD_HEADER {  
  
    char VmdHeader[30]; // "Vocaloid Motion Data 0002"  
  
    char VmdModelName[20]; // カメラの場合:"カメラ・照明"  
  
};
```

ヘッダを読み込んだら次の8バイトがモーションデータ数です(ヘッダ32バイトでいいじゃん…)

そして次にモーションデータですが、以下のようになっています。

```
struct VMD_MOTION { // 111 Bytes // モーション  
  
    char BoneName[15]; // ボーン名(くそが!!!!)  
  
    DWORD FrameNo; // フレーム番号(読み込み時は現在のフレーム位置を0とした相対位置)  
  
    float Location[3]; // 位置  
  
    float Rotation[4]; // Quaternion // 回転
```

```

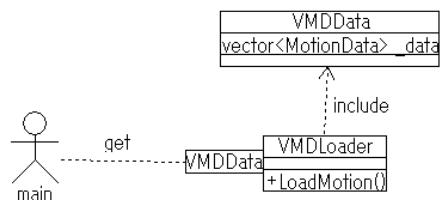
    BYTE Interpolation(64); // [4][4][4] // 補完(ベジエーデータ)
};

…またツツツツ!!!

```

なんなんだよ15バイトってよ!!!おどりやくそ!!!

仕方ない…PMDMesh&PMDLoaderと同様にVMDDataというクラスとVMDLoaderを作りましょう。



今回はテクスチャもコンスタントバッファも使わないでとにかくロードしよう。話はそれからだ。ここは特にヒントなしでもできるでしょ？

あでもVMD\_MOTIONはこう書き換えておいたほうが便利かな？

```

struct VmdMotion { // 111 Bytes // モーション
    char boneName(15); // ボーン名(くそが!!!!)
    unsigned int frameNo; // フレーム番号(読み込み時は現在のフレーム位置を0とした相対位置)
    XMFLOAT3 location; // 位置
    XMFLOAT4 rotation; // Quaternion // 回転
    unsigned char interpolation(64); // [4][4][4] // 補完(ベジエーデータ)
};

```

違いわかりますかね？

で、とりあえずメイン側に対して今教える必要があるのは「ボーン名」と「クォータニオン」ですね。

```

//モーション情報(1個あたり)
struct MotionData{
    std::string boneName;//ボーン名
    XMVECTOR quaternion;//クォータニオン
}

```

};

ですから先程の構造体を pack(1)でも何でも良いので、とにかく読み取って、その後で VMDData クラスに必要なものを入れてください。今は必要なものだけいいです。そっちのほうが考え方やすいから。でももちろんこれも複数あるので std::vector で配列しといてください。

分かりませんか？

構造的にはですね。

①VMDData の中に vector<MotionData> が入ってて、それをクライアント側に提供します

②VMDLoader は VMDData を new 生成して、それをクライアント側に返します

③VMDMotion はロード時に使用するだけ

というわけです。

ここまで頑張つてついてこれたんだから、ここは僕のソースコードなんか見ずに考えられるはず。

あ、最後に fclose は忘れないようにね。忘れやすいからね。

## 19.4一方、クライアント側では…

ひとまずクライアント側は VMDLoader をインクルードしてください。そして LoadMotions 関数を呼び出してください。いや、ロード関数名はなんでもいいんですけど、ともかくロードしてください。

```
VMDData* vmddata = vmdloader.LoadMotions("pose.vmd");
```

そしてこの vmddata にモーションデータ(ボーン名とクォータニオン)が入りまくっています。次にやるべきことは、リファクタリング時に

DeformBones 関数を作ったと思いますが、こいつを改造します。リファクタリングしないところが面倒になってたんですよ。やっててよかったです。

もともと DeformBones は自分でボーン名と回転を指定していたので

```
void DeformBones(PMDMesh* mesh, VMDData* vmddata)
{
    //左ひじ変形
    int elbowIdx = mesh->BoneMap()["左ひじ"];
    XMVECTOR offsetVec = XMLoadFloat3(&mesh->Bones()[elbowIdx].headPos);
    XMVECTOR tailPos = XMLoadFloat3(&mesh->Bones()[elbowIdx].tailPos);
    XMATRIX bone = XMMatrixTranslationFromVector(-offsetVec);
    bone *= XMMatrixRotationZ(XM_PIDIV2);
    bone *= XMMatrixTranslationFromVector(offsetVec);
    mesh->BoneMatrices()[elbowIdx] = bone;

    int shoulderIdx = mesh->BoneMap()["左手首"];
    offsetVec = XMLoadFloat3(&mesh->Bones()[shoulderIdx].headPos);
    tailPos = XMLoadFloat3(&mesh->Bones()[shoulderIdx].tailPos);
    bone = XMMatrixTranslationFromVector(-offsetVec);
    bone *= XMMatrixRotationZ(XM_RIDIV4);
    bone *= XMMatrixTranslationFromVector(offsetVec);
    mesh->BoneMatrices()[shoulderIdx] = bone;

    int wristIdx = mesh->BoneMap()["左手首"];
    offsetVec = XMLoadFloat3(&mesh->Bones()[wristIdx].headPos);
    tailPos = XMLoadFloat3(&mesh->Bones()[wristIdx].tailPos);
    bone = XMMatrixTranslationFromVector(-offsetVec);
    bone *= XMMatrixRotationZ(XM_PIDIV2);
    bone *= XMMatrixTranslationFromVector(offsetVec);
    mesh->BoneMatrices()[wristIdx] = bone;
}
```

こんな感じのソースコードだったと思いますが、もう自分でやる必要はないのです。何故ならボーン名と回転情報は今作った vmddata に入っているんですから。

というわけで元の直指定ソースコードは潔く消しちゃいます。あとは全てのボーンモーションデータに対して処理を行いたいので当然 for 文を使います。vmddata の持つデータで回します。

あと、やることは↑の肘曲げとほぼ同じです。回転がウォータニオンになるだけです。

1. それぞれのボーン起点をとってくる
2. 中心移動行列を作る
3. ウォータニオン回転行列をかける
4. 元の位置移動行列をかける

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.matrix.xmmatrixrotationquaternion.aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixrotationquaternion.aspx)

を見ると難しそうですが、とにかくロードしたウォータニオン値をいれて見れ。

うまくいけば



こうなるはずじゃけえ。ちなみに肩、肘、上半身、首をまわしています。なお、IK切ってモーションを作れば下半身もMMDで設定したとおりに動きます。



おめでとう!!君はついに2つ目の山を超えたのだ。

だが…これで終わりではない。地獄というのはいくつもあるのだ。



## 20 アニメーション

ついに一つの山場が終わりましたが、このままではまだ命が入っていません。ミクさんに命を与えるにはアニメーションしなければなりません。

まあでも、ついにここまで来たかって感じですね。長かった。

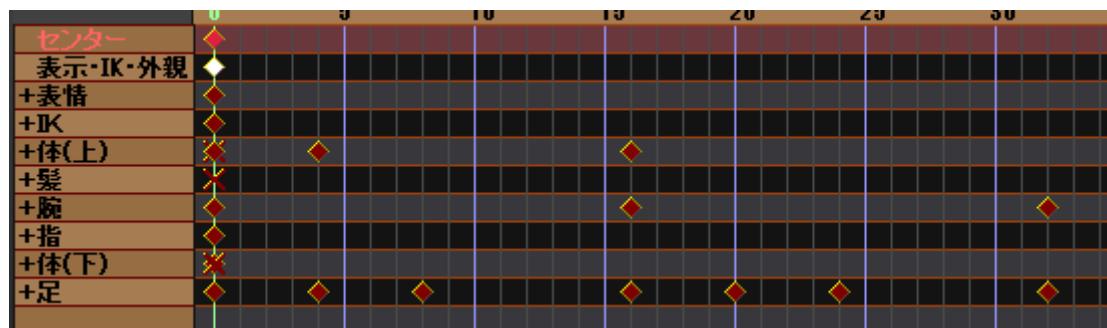
さて、ここでまた悩むことになります。アニメーションは、ポージングとは違うのです。

どう違うのか?というと…

1. 時間について考えなければならない
2. キーフレームについて考えなければならない
3. 一つのボーンに対してのポーズが一つではない
4. クォータニオン値補間を考える必要がある

などです。

今までの次元に更にもう一つ…時間の概念が入ってくるのです。先に言っておくと、残念ながらカンタンではないですね。



ご覧のように

「各部位」と「キーフレーム」で行列みた感じになっています。

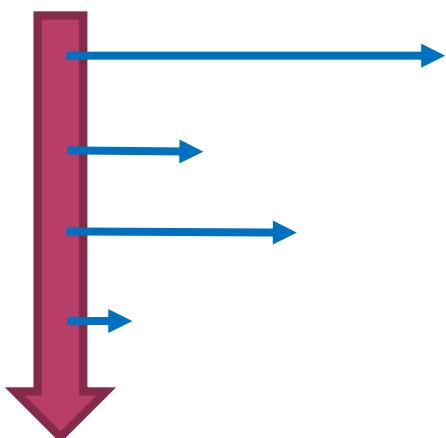
そして「キーフレーム」の数は部位によって違いがあるのです。こういう時はシンプルに考えればいいです。

…答えなんかないんです。こういう場合にどうしたら良いんだろう?「それを教えるのがセンセーだろうが!!」っていう意見もあるんでしようけど、教えるのはカンタンのよ?

でもワシはそれ以上に「自分で考える力」をつけるのが大事だと思つるんじゃ。だってあと1年半くらいでもう働くんだぜ? 現場じゃ「そんなもん自分で考える」が落ちだぜ? そろそろ自分で考える練習をしてみようなんだぜ。

## 20.1 構造を考える

とにかく図を書けメモ書け、頭フル回転じゃ!!



とまあ、図のように二次元配列のようだが、フレーム方向は可変長である。

右方向は次から次へ追加できるように vector ガレル! だろう。じゃあ縦方向はどうしようか…ここは自分で考えてほしいんだけど、map だろうが vector だろうがあんまりカンケーないわなあ…。

まあ…入れやすさを考えるとやっぱりこうかなあ…

```
map< string, vector<MotionData> > _keyframes;
```

結局のところ「ボーン名」と「モーションデータ」のペアなわけよね。というわけで MotionData から「ボーン名」を削除し、代わりに「フレームカウント」を追加します。

```
struct MotionData{  
    unsigned int frameNo;//キー フレームのフレーム  
    XMVECTOR quaternion;//クオータニオン
```

};

だね。

となると

```
std::map<std::string, std::vector<MotionData>> _keyframes;
```

と書くことになるんだが、如何せん長い。

## 20.2 `typedef`

こういう時に使うのが、`typedef`だ。C 言語からある文法なので、「知らない」とは言わせない。言わせない方が、中々使う機会もないだろう。

# 今がその時だ!

`typedef` そのものは「何かする」ものではないからね。要は「コードを読みやすくする」ためのものだ。C/C++に限らずいろいろな言語には「読みやすくする」文法があるが、必ずしも必須ではない。でもプロとしてプログラムする気があるなら使いこなせるようになっておこう。

さてなぜ「今でしょ」なのかというと、こういう STL コンテナの入れ子により「型名」が非常に長くなってしまった時に効果的なのだ。

今一度 `typedef` について思い出してほしい。どういうものだった？

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

そう。データ型に対して新しい名前を付けるものだったね？ うん、特に意識せずにプログラムしてると存在すら忘れてしまいそうだね。忘れちゃってる人はもう一回勉強してくれ。

だが今回のような場合は非常に役に立つ。

```
typedef std::map<std::string, std::vector<MotionData>> KeyFrames_t;
```

と書けば変数宣言は

```
KeyFrame_t _keyframes;
```

で済むのだ。ちなみに `typedef` の最後に `_t` をつけるのは「僕の」コーディングルールなので、特に真似する必要はないよ。

さて、ここを `typedef` にした事にはもう一つの便利さが隠されているんだけど、それはまた後で話しましょう。

というわけで読み込んだボーン名とボーン情報から

```
MotionData motion={frameNo,rotation};  
_keyframes(boneName).push_back(motion);
```

などと書けるわけだね。もうこの辺は問題ないかな？

### 20.3 std::map についてもう少し詳しく

ちょっとこの後の話に入る前に `std::map` についてまだ話していないことを解説せねばなるまい。

`map` が連想配列のように働くのはもう大丈夫だね？ 散々使ってるもんね。ところで今の状況は

「ボーン名」がキーで、「MotionData のベクタ」が値なわけだ。

さてここで質問です。

例えば連想配列の先頭の「ボーン名」が欲しいとします。っていうか全ボーン名が必要だよね？ そういう場合ってどうやって取ってきたらいいんでしょうか？

ほい、ヘルプ

グーグルトップ

<http://cppref.jp.github.io/reference/map/map.html>

グーグル二位(ちなみにこれ書いてる人は早良区に住んでるらしい…ワイのライバルやで)

<http://vivi.dyndns.org/tech/cpp/map.html>

グーグル三位

<http://ppp-lab.sakura.ne.jp/cpp/library/010.html>

読んでわかりますか？

うん、まあ初学者にとっては割と難易度が高いんだよね、これ。上位三つ読んでわからなかつたら普通諦めるわなあ…というか多分「あ、これ無理なんだ」と思って MotionData 側にボーン名入れちゃつたりするんだよね。

## んなこたあなれ！

<http://cppref.jp.github.io/reference/map/map.html>

の最初の方を読んでみよう。

`map` は一般的に二分木として実装される。従って、連想コンテナである `map` の主な特性は以下の通りである。

- ユニークな要素のキー:互いに等しい二つのキーを持つ要素が `map` に格納されることは無い。複数の等しいキーを許す同様の連想コンテナは `multimap` を参照のこと。
- **要素の値はキーと値の `pair` 型である。**
- 要素は常に 狭義の弱順序 に従う。
- 挿入操作はイテレータや要素の参照に影響を与えない。

`map` を連想配列のように使うというのは使い方の一つでしかないんだよねえ。というか `map` 自身のオペレータオーバーロードによって連想配列のように使えるに過ぎないんだよね。

とりあえず構造は「木構造」である。とはいえ、木の並びの法則性がボーン構造と違いすぎる  
ので、ボーンツリーには使えない。単純に検索しやすくするためにだけの「木」だ。

興味がある人はハッシュツリーとか二分木とか2-3-4木とか赤黒木とか調べるといいよ…そ  
れなりに死ぬけど。

さてここで注目してほしいのは

pair型

についてだ。こいつは二つの値をペアにして持つという…ただそれだけの型だ。

<http://cppref.jp.github.io/reference/utility/pair.html>

要はmapというのはこのpair型が二分木構造で並んでいるというデータ構造なのだ。単なる  
ペア型に過ぎないものをmapは片方をキーとして、もう片方を値として利用しているわけ  
だ。

つまりこのコンテナから一つの要素を取り出してキーを指示する側…それが今回の場合は  
ボーン名にあたるというわけ。

で、pair型ってのはfirstとsecondでペアを作っているわけで、mapは

- first:キー
- second:値

として扱っているわけだ。あとは分かるね？ボーン名が欲しけりや

要素.first

もしくは

要素->first

とするわけだ。

例えば、VMDDataのData()関数がKeyFrames\_tを返すのならばこうなるだろう

```
for (auto& framebone : vmddata->Data()){
    int idx = mesh->BoneMap()(framebone.first);
```

あとはやる事いつしょやね。

ひとまずは、返すべき情報を

```
std::vector<MotionData>
```

から

```
std::map<std::string, std::vector<MotionData>>
```

に変更したうえで、元と同じようにポージングできるところまでやってみてくれ。何度も言うようだが、動きなり動かそうとするのはまだ早い。おちつけ。

## 20.4 フレームレートを安定させる

漸くアニメーションだ。とりあえず腕を振ってみよう。最初なのでひとつボーンしか動かさなくていいんだろう。

ちなみに MMD は秒間 30 フレームの計算であるが…。まあやってみりや分かるが、30 フレームどころか秒間 1000~6000 フレームくらいで動作してるんじゃないだろうか？

そもそもそこを解決しないと MMD を再現どころじゃない。

えっ？ だって最初にリフレッシュレート決めたじゃん…って思うでしょ？

```
sd.BufferDesc.RefreshRate.Numerator = 60; // 分子
```

```
sd.BufferDesc.RefreshRate.Denominator = 1; // 分母
```

これはあくまでも「リフレッシュレート」の指定であって「フレームレート」ではないんや。この一つはフルスクリーンにしたときに、画面更新周波数をどれくらいにするのかっていう、ハードウェアに対する指定に過ぎないんや…だからプログラムの処理自体…メインループの更新回数には影響ないねんで。

ちなみに言うと、画面が 60Hz で更新されている場合、MainLoop 関数が 60 フレーム以上で更新されても意味がないぜ。なんでかというとディスプレイには反映されないからだ。

ついで CPU 的にももったいねいし、そもそも MMD 再生がおかしなことになるわけなんだぜ。

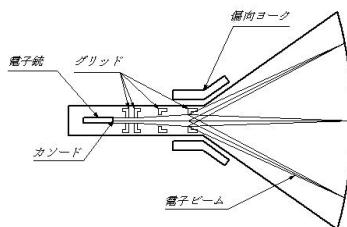
まあ一応垂直同期(仮想 60Hz)と合わせるには

Present 関数の第一引数を 1 にするといいぜ。

これで一応垂直同期に合わせることになるんだ。ちなみにこの「垂直同期」って言葉に少年たちは馴染みがないのかかもしれないが所謂「ブラウン管」と呼ばれるものの仕組みに由来するのだ。

<https://ja.wikipedia.org/wiki/%E3%83%AA%E3%83%95%E3%83%AC%E3%83%83%E3%82%B7%E3%83%A5%E3%83%AC%E3%83%BC%E3%83%88>

昔のブラウン管というのは、電子銃から電子(光の粒)を発射し、それを画面にぶつけて発光させていたのだ。で、その位置制御には電磁石を使っていたのだ。



まあそんな昔ばなしはどうでもいいんだけど、この電子銃が左上に発射されて、右下に発射し終わるまでをおよそ 1/60 秒としていた。そういう意味で「アナログディスプレイ」とか呼ばれたりするんだが…。ちなみに液晶の場合はどうかというと電子銃方式ではないため、液晶がその気になれば画面すべてを一度に更新することもできる。

…とはいっても、どっちみち今度は GPU 側のフレームレート(画面の色を(情報として)更新する時間)があるのと、過去との互換性のためだいたい 160Hz とか 30Hz が採用されることとなる。

ちなみに先ほどの Present 関数だが

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

第一引数を 1 にすると垂直同期信号(60Hz なら 1/60 秒)を 1 回待つ…つまりフレームレートが 60 になる。2 にするとフレームレートが 30 になるため、一応 MMD と同じにはなる。

しかしゲームとして考えた場合はやはりフレームレート 60 が理想であるため、ここでは 1 としておく。

## 20.5 フレームと連動させる

勿論、フレームと連動させるにはボーン更新処理を毎フレーム行う必要があります。つまりボーン情報の更新をメインループ側に入れてやります。処理スピードが心配ですが…まあ仕方ないのです。

ちなみに僕のソースコードの DeformBones の中身はこんな感じです。

```
for (auto& frames : vmddata->Data()){
    int idx = mesh->BoneMap()(frames.first);
    for (auto& frame : frames.second){
        XMVECTOR offsetVec = XMLoadFloat3(&mesh->Bones()(idx).headpos);
        XMVECTOR tailpos = XMLoadFloat3(&mesh->Bones()(idx).tailpos);
        XMMATRIX bone = XMMatrixTranslationFromVector(-offsetVec);
        bone *= XMMatrixRotationQuaternion(frame.quaternion);
        bone *= XMMatrixTranslationFromVector(offsetVec);
        mesh->BoneMatrixes()(idx) = bone;
    }
}
```

今後はここからさらに時間の要素まで絡んでくるので相当めんどくさいよなあ。ともかくボーン変形更新関数をメインループにも設置してくれ。

そして、引数の最後に frame を入れてくれ。これが今回のメインだ。

現在のフレームを元に、現在あるべきボーンを取得するわけだが、このフレームと vmddata 側の frameNo を比較する。とあるボーンには複数のキーフレームデータがあるので前方から検索していく。

とりあえずその frameNo と一致したらボーン情報をそっちに変更する。ひとまずここまでやってみましょう。

さて…「検索」と言いましたが、どうします？まあ、横向き(ベクタ方向)の数は知れてるので、一個一個チェックしてもいいんだけど、せつかくだから俺は C++ の機能を使うぜ!!

## 20.6 algorithm : : find\_if を使ってみよう

ちょっと前に予告していた「algorithm」をちょっとだけ使うよ。

algorithmにはお便利な関数がたくさんあって、find\_ifという関数もそのうちのひとつ。…なんだけど、昔はこのalgorithmってのを利用するためにはファンクタっていうややこしいクラスを作らなければならなかつたんだけど、今は「ラムダ式」という超便利な機能があるから割とシンプルに書けるよ。

[http://cppref.jp.github.io/reference/algorithm/find\\_if.html](http://cppref.jp.github.io/reference/algorithm/find_if.html)

皆にとっては「C++って仕様多すぎだろ」とて事で正直 C++嫌いになりかけてるかもしれないが、同じことを自分で実装することを考えてみてくれ…相当面倒くさいぞ!?

ちなみにみんなは「ファンクタ」の事を知らなくてもいい世代だ。それはうらやましいのだ。

専門的なことはともかく…男は度胸…なんでもやってみるのだ。

ちなみにラムダ式は以前にも話したかもしれません

(クロージャ)(引数){処理;}

という文法です。うーん、要は「無名関数」ってところかな。関数自体の存在はあるんですけど。宣言の仕方がフツーと違うってことと、別に関数内で宣言してもいい(フツーの C 言語だと関数内関数は許されてないよね?)

ところで…以前に話した時には多分言ってなかつたと思うんだけどこの「クロージャ」ってなんでしょうね。

ラムダ式の説明↓

[http://cppref.jp.github.io/lang/cpp11/lambda\\_expressions.html](http://cppref.jp.github.io/lang/cpp11/lambda_expressions.html)

()内は「キャプチャ」とか書いてますけど、要はラムダ式定義時に渡すべき変数というわけです。つまり先程の find\_if と組み合わせると

```
std::find_if(frames.second.begin(),
```

```
frames.second.end(),
```

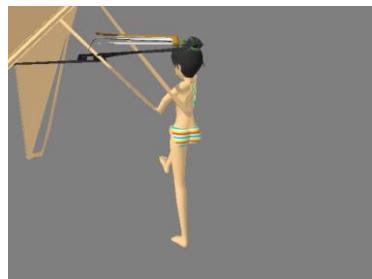
```
(frameNo)(const VMDData::MotionData& md){return md.frameNo == frameNo;});
```

こんな感じに書けます。

で…見つかった場合に何が返るんでしょう？これが今まであまり意識してなかったけどイテレータと言うやつで、ポインタみたいなもんです。見つからなかった場合は end()と一致します。

もちろんその時に DeformBones のあとにはボーン情報の Map～Unmap をするわけだが、特に説明はいらないよね…？

ところで特に言ってなかつたが、うまいこと行くと恐らく…



こうなる。何を言っているのかよく分からねーと思うが俺にも何が起きたのか…あつ!!

DeformBones を毎フレーム実行するということはつまり…毎回乗算がかかるということであり…つまりそういうことなのだ。パンパンあるべき場所からずれていく。

というわけで、ボーン行列は毎フレーム全初期化してあげないといけません。というわけで全初期化してあげましょう。

全初期化をする時は algorithm の fill を使うと良いです。

<http://cppref.jp.github.io/reference/algorithm/fill.html>

```
fill(mesh->BoneMatrixes().begin(), mesh->BoneMatrixes().end(), XMMatrixIdentity());
```

こんな感じで呼び出せば OK

あ、この時 HeadPos をずらしてたりするとまたおかしなことになりますので、HeadPos をずらしてある部分は削除しましょう。

これでうまいこといくとおもいます。

思いいますが多分腕が上がるの一瞬だけだと思います。そりやそうだ…合致するときだけボーン適用だからね。

さて…どうしましよう？

frameNo は時間経過で増えていきます。それだったら find\_if の評価関数を m.frameNo <= frameNo にすれば、最後のものが選択されるんじゃないの？と思うかもしれません…今一度 find\_if を見てください。

[http://cpprefjp.github.io/reference/algorithm/find\\_if.html](http://cpprefjp.github.io/reference/algorithm/find_if.html)

「[first, last) 内のイテレータ i について pred(\*i) != false である最初のイテレータ」を返す。そのようなイテレータが見つからなかった場合は last を返す。』

イテレータってのは「要素を指し示すもの」って意味ね。

そうなのよ。合致する最初のイテレータしか返さないのよこの関数。どうしましよう…

<http://cpprefjp.github.io/reference/algorithm.html>

を見ても適切に使えそうな関数はないですね…

というわけで、また更に更に皆さんには申し訳ないことを教えます。それが逆イテレータ

## 20.7 std::reverse\_iterator(逆イテレータ)

今までしつつ使ってきましたが、コンテナの「イテレータ」は順方向イテレータと言って、

```
std::vector<MotionData>::iterator it = find_if(frames.begin(), frames.end(), 評価関数);
```

で取得し、例えば `it++` なんて書くと、「次の」要素を指し示すものでした。

でも逆からアクセスしたいときってあるよね？今回みたいに。

そういうのもきちんと用意されています。それが `reverse_iterator` です。

名前からピーンと来てる人はいると思いますが、通常のイテレータの進み方と逆方向に進みます。

こういうのがあるってのを知つてると…なんとかなりそうでしょ？ちなみに逆方向の begin() ~end() は rbegin() ~rend() となります。

つまり

```
auto revit=std::find_if(frames.second.rbegin(),
    frames.second.rend(),
    (frameNo)(VMDData::MotionData& md){
        return md.frameNo <= frameNo;
    });

```

分かるか？こういう事だ。

で、この辺は frameNo 以下のフレームナンバーと一致する最後の要素を指すイテレータが返ります。

こいつを使って座標変換を行うと…動画を見せられなくて申し訳ないけど、腕を上げたり下げたりしますが……そうね、こんなんじゃないよね。

## 20.8 ボーン間の補間

そうだね。ボーンを補間しなきゃならないよね？今現在のフレーム～次のフレームの間を補間しなければいけナイわけですね。

まずは現在のフレームがある部分をなんとかして  $0 \leq t \leq 1$  の範囲内に変換して

最終行列 = 変換行列 A \* t + 変換行列 B \* (1 - t)

となるようにすればいいですかね？

では、t をどのように  $0 \leq t \leq 1$  とするかですが、うーん。現在の行列と次のフレームがわかれば良いわけです。

数式で書くと

$$t = \frac{\text{frameNo} - \text{now.frameNo}}{\text{next.frameNo} - \text{now.frameNo}}$$

こんな感じですね。分かりますか？いわゆる内分比ってやつですね。

こんな感じで $\dagger$ を計算してください。

それを用いて

```
nowMatrix*(1-t)+nextMatrix*t
```

とします。

ところで nextはどうやって取ってきたら良いんでしょうか?

```
revit++
```

ですか?

そうは行かないんですね。こいつはリバースイテレータ…ですから $++$ すると逆方向に向かうので、「次」ではなく「前」になります。

じゃあ、こう?

```
revit--
```

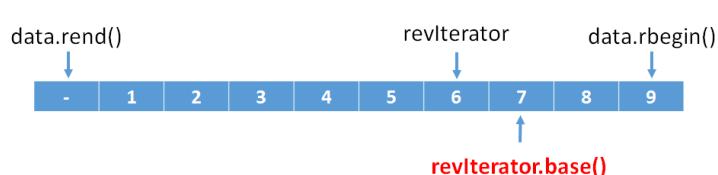
まあ、悪くないね。でも… $revit==rbegin()$ である場合にはエラーが起きます。

```
if(revit!=rbegin()){  
    revit--  
}
```

でもいいんですけど…`reverse_iterator`には`base()`って関数があります。

[http://en.cppreference.com/w/cpp/iterator/reverse\\_iterator/base](http://en.cppreference.com/w/cpp/iterator/reverse_iterator/base)

こいつでイテレータにすることができるんですが、実はこの`base()`結果のけが指し示すものは



そう…一つ後なんですね。これについては後々詳しく意味を話しますが、とりあえず「そうなってる」と思ってください。

最後に…ついてこれでないならついてこれでないと言ってくれ…ある程度待ってやってるじゃろ？

あと、どこかの項目を再勉強したいならそれもリクエストしてくれ…1人で遅れて行かないでくれ…マジで。特に次年度就職年次はさ…ただ、就職のためには自分で考える力が必要なので、自分で考えて考えた上で質問してくれ。

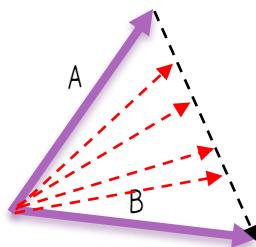
というわけで合成しましょう…あれ？



いや…これアカンやつでしょ……理由はこれからお話ししましょう。

## 20.9 Slerp(球面線形補間)

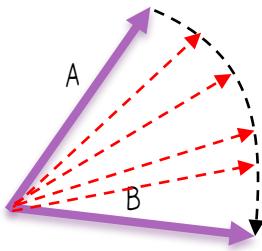
どうして響ちゃんの腕が大変なことになっているのか…？以下のベクトルの補間について考えてみよう。ベクトルAからベクトルBへ補間するとする。すると…



このような変化をすることになる。 $A \rightarrow B$ に移行する間、図の赤い矢印のように長さが半減してしまうわけだ。これじゃあイカンわけよ。

というわけでいい感じに補間する方式として「球面線形補間」というものがあります。

球面線形補間で補間すると…



こうなります。ベクトル補間の場合はこうしないと妙な感じに変形されます。

球面線形補間は slerp と言います。SphereLinearInterpolation の略です。変な略し方ですが、なんかこれが通称になってます。

で、クオータニオンとして slerp するには XMQuaternionSlerp を使用します。

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk quaternion.xmquaternionslerp\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk quaternion.xmquaternionslerp(v=vs.85).aspx)

あとはカンタンですね。

これを利用すれば



適切な補間が行われます。

## 20.10 VMD ファイルの罠…

VMD ファイルにはちょっとした罠が仕掛けられている。大したことじゃないんだが、いや…重大な罠だな。これを放置するとチャージマン研の変身シーン以上にカッコ悪くなる。

それはだな…VMD のキーフレーム情報は「登録」した順に並んでいるのであって、決してフレーム番号ごとに並んでいるわけではないということだ。

つまり、例えば

0→30→60

と並んでいるところに後から 50 フレームを追加するとデータの並び的には

0→30→60→50

となり、50 フレーム時点での挙動が無視されてしまうわけだ。

ところが本来は

0→30→50→60

にならないとうまくいかないわけだ。

じゃあどうすればいいのかというと、…どうすればいい?

#### 20.10.1 ソート (std::sort)

まあ順当に考えて「ソート」ですね。これも algorithm に頼ります。

<http://cpprefjp.github.io/reference/algorithm/sort.html>

といふわけなんだが、この例のやり方ではうまくいかない。何故なら、どの数値を基準に並べ替えをするのが明記されていないからだ。

つまり…

```
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

こちら側…ソート関数まで明記されているソートを使う必要がある。でも既に find\_if とかを使ったみんなには分かっていると思うがラムダ式を使えばカンタンなのである。

```
std::sort(begin(), end(), [](MotionData& a, MotionData& b){return a.frameNo < b.frameNo;});
```

といふわけだな。分かるかな?

左側が小さければ true。そうでなければ false…つまり小さい順に並ぶということだ。

### 20.10.2 そもそも挿入時に並び替えすればよくね？

ご名答…かどうかはよくわからんが、いちいちソートしなくても済む方法はある。

std::set

を使うのだ。

<http://cppref.jp.github.io/reference/set/set.html>

こいつは勝手に挿入時にソートを行ってくれるコンテナなのだが…

こいつを使う時は……要素の operator< を定義しなければならない。

何故なら並び替えを行うために「どちらが大きいか」という定義をしてあげなければならぬ  
いからだ。それはそれで面倒くさいんだけどね。

というわけで MotionData の < オペレータを作ります。

```
bool operator<(const MotionData& l, const MotionData& r)
{
    return l.frameNo < r.frameNo;
}
```

こんな感じですね。

そうすると挿入の時点でソートされているため、特にソートの必要がなくなります。

あ、ただし挿入のときのメソッドは push\_back とかではなく insert 関数になります。

push\_back ってのは末尾に追加って意味なので、set の意味にそぐわないんですね。末尾に入らないですから…というわけで vector から set に変更したら挿入関数は insert にしてください。

## 20.11 最後の仕上げ

さて、頂点座標をボーンに適用したのは良いけど、問題が一つだけ残っているかもしれません。  
とりあえず僕の言うとおりに作ってると一つだけ問題が残ります。

それは…画像を見てもらったほうが早いと思います。

違いが分かりでしょうか?



よく見てください。よく見れば分かることです。

全然わからない人はもう少し観察力をつけましょう。じゃあ聞きます。どちらが自然な感じですか?

これは敢えてここには書かないでおきましょう。僕としてはゲームプログラマたるものそういうセンスも身につけてほしいと思っています。少なくとも「自分の嫁」の不自然さには気が付きましょう。

さて、おかしいのは分かるんですけど「どうおかしいのか」を説明できる人はいますか?その上で「何が原因なのか」を説明できる人は?この先を読み進める前に今一度観察してください。大事なことです。

…それがゲームプログラマの仕事だと思ってください。システム系のプログラマならやってきた仕様を仕様どおり動作するように作ればそれでいいのでしょうか。

でもゲームプログラマと言うやつはエンターテインメントを作っています。美的感覚、観察眼も技術力と一緒に鍛えましょう。

さて…もう殆どの人気がお分かりかと思いますが…

そうですね。陰が不自然ですよね？左手前上から光が当たっているにも関わらず手前に来る手、上げている手が暗すぎですよね？

何ででしょう？想像してみて、思い当たるところを考えてみてください。いつもデバッグをセンサーに頼ってばかりでは技術力は上がりません。自分で調査、そして推理するのがゲームプログラマのお仕事です。

- 明るさが不自然→
- 明るさを決めているのは誰だっけ?→
- ランパートの余弦則だ→
- 法線が光線のどちらかがおかしい→光線はそろそろ変わらないはず→
- 法線がおかしい!!!!犯人はお前だ!!!



という思考の経路を辿って、私はどうやら法線がおかしいっぽいことに気が付きました。  
どうおかしいんでしょうか？

今一度頂点シェーダを見てみましょう。

```

//頂点シェーダ
OutputP BaseVS(float4 pos : POSITION,float4 normal : NORMAL,float2 uv:TEXCOORD,uint boneid : BONENO ,uint
weight : WEIGHT)
{
    OutputP outputp;
    float wgt1 = float(weight) / 100.0;
    float wgt2 = 1.0-wgt1;

    matrix m = mul(camera,world);

    m = mul(m, boneMat(boneid & 0xffff) * wgt1 + boneMat((boneid & 0xffff0000)>>16) * wgt2 );

    outputp.pos = mul(m, pos); //座標情報をまるごと代入
    outputp.normal = mul(world,normalize(normal)); //法線にもワールド行列をかけないと…
    outputp.uv = uv;
    return outputp;
}

```

一体誰が犯人なんだ!?犯行現場はどこなんだ!?

はい、聰明な皆様ならお気づきになられたかと思います。思い出してください…なぜ法線にもワールド行列をかけないといけなかったのかを…。

そうです。ライト固定で回転してるとおり、お尻に光が当たっているにも関わらず暗いままでしたね？

法線方向は『回転による影響を受けるべき』なのです。えっ?だってワールド行列さえかけば回転は反映させているはずじゃ…はつ!!!

そう、このスキンメッシュアニメーションでは他にも回転させているところがありましたよね？

# それはボーン行列…お前だ!!!!

オマエノシワザタタノカ…

というわけで、今までワールド行列だけをかけていた部分にボーン行列もかけてみましょう。



あ…あれ?

「お~ゴミ野郎!!!これじゃ前のほうがマシじゃねえかよ!!!なんだとこの陰影は!!!」

まあ待て、慌てるな。「おかしな現象」には何らかの原因が必ずあるものだ。よ~く考えろ。自分の頭で考えろ。今まで学んだことを思い出せ…思い出せ…。

ボーン行列とはいったい…うごごごご。思い出せエーッ!!!!

ボーン行列=原点に移動する行列×回転行列×元の座標に戻す行列

だったはずです。

法線は「回転による影響を受けるべき」であって、それ以外の影響を受けるべきではありません…。

そろそろ分かってきたかな？ボーン行列には余計な変換が含まれてしまっているのです。さて…ここから「回転以外の余計な操作」…ぶっちゃけて言うと平行移動成分をなくすにはどうしたらいいんでしょうか？

これは数学の時間にもやりましたので思い出してください。行列は

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

こういうものでしたね？これは二次元における変換行列ですが、仕組みは同じことです。平行移動成分を消したければ平行移動成分を0にすればいいじゃない!!!!

[https://msdn.microsoft.com/ja-jp/library/bb509634\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509634(v=vs.85).aspx)

ここを見れば分かるように行列のそれぞれの成分をいじるには

\_11メンバや\_m00メンバを使用します。

今後もこういう機能は使うかもしれない（）のでこの変換はhlslの中で関数化しておきましょう。通常のC言語の関数みたいにすればいいです。なのに？行列のどこをいじればいいかわからぬ？

いやいや…。分かるはずですよ? \_14 と \_24 と \_34 をいじれば終わります。がんばってください。

(0 君から指摘があったのだけど、NORMAL の w を 0 にしたら平行移動成分が 0 になるから 1 行で済みますよ…とのこと。せやな…気が付かんかった。



長く苦しい戦いだった…。

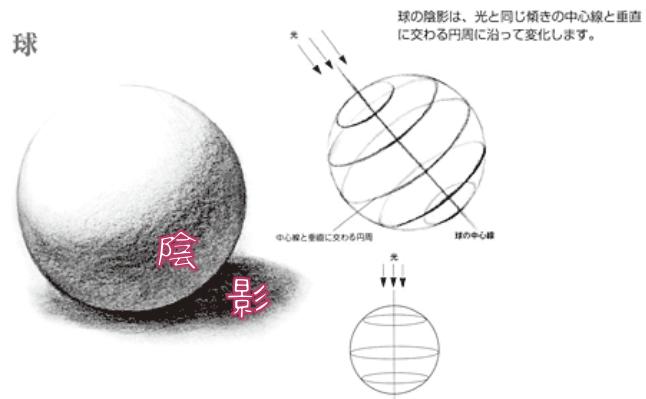
おめでとう!!ここまでできれば君は(IK 抜きの)アニメーションをマスターした!!!

本当におめでとう!!!

新たなる地獄へようこそ…ククク。

## 21 シャドウマップ

さあお待ちかね…。物体に「陰」を落としたときのことを思い出してください。今一度あの資料を見せますが



こういうことでしたね？

今回はこの「影」のほうのお話をします。少しだけ考え方がややこしいのでそれなりの覚悟はしてください。「影」の落とし方は色々なのがありますので、簡単なものから説明していきまですが、どれも割と興味深いものです。

影に関してかなりの情報を提供してくれるのがこの書籍です。



とはいって別に買わなくてもいいです。高いし。

[http://www.project-asura.com/program/d3d11/d3d11\\_008.html](http://www.project-asura.com/program/d3d11/d3d11_008.html)

とか

<http://marina.sys.wakayama-u.ac.jp/~tokoi/?date=20050926>

とか

<http://d.hatena.ne.jp/setuna-kanata/20090131/1233413629>

が参考になるとは思います。最後のサイトの人はもう亡くなってるんですね(´;ω;`)

もうぶつちぎりでシャドウを終わらせたい人はサイトを見たり書籍を見てください。

で、シャドウマップをいきなりやりたいところですが、挫折者が出てそうな部分なので簡単に影が出る方法を教えておきます。

## 21.1 XMMatrixShadow を使う

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

これを使う前に知っておいてもらわなければならぬ数学の仕様があります。

「三次元平面の方程式」です。

### 21.1.1 三次元平面の方程式

二次元における直線の方程式を三次元に拡張しただけの話だと思ってください。

二次元の直線の方程式は

$$y = ax + b$$

でしたよね？ これって見方によっては

$$ax + by = c$$

$$ax + by - c = 0$$

と表記しても構いませんよね？ この辺がわからない人は正直に言ってね？

これを三次元に拡張するということはつまり

$$ax + by + cz = d$$

$$ax + by + cz - d = 0$$

というわけよね？これが平面の方程式です。教科書によつては

$$ax + by + cz + d = 0$$

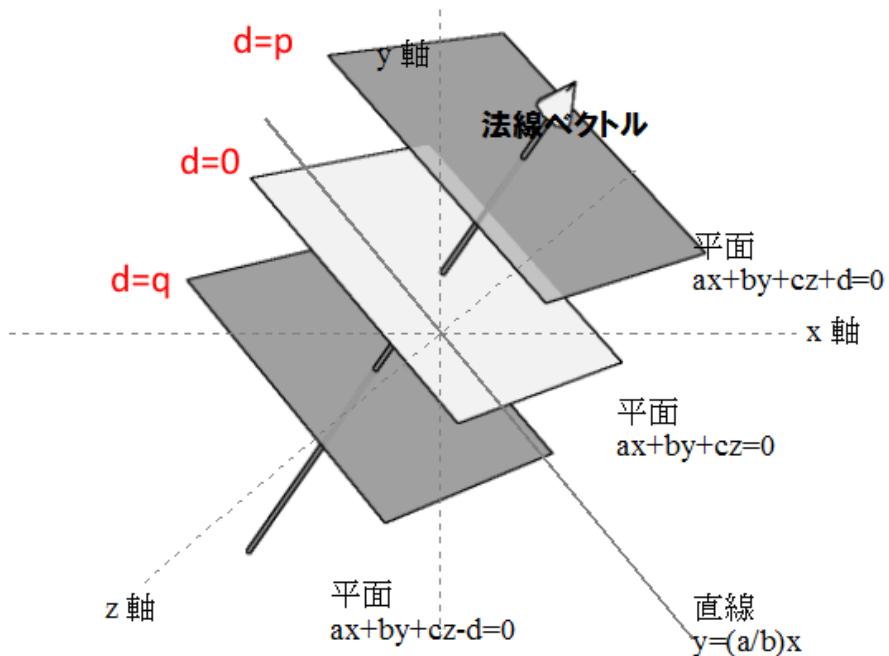
と書いてるけど、僕は  $d$  はマイナスにしたほうが分かりやすいと思います。

ここまででは良いかな？難しいかも知れないけどついてきてね？

実はこの平面の方程式ってのは法線ベクトルに対応していて、この平面の法線ベクトルは

$$\vec{N}(a, b, c)$$

というわけだ。



よくわかんない人は、例えば  $XZ$  平面に平行な平面を考えて欲しい。 $XZ$  平面の場合は法線ベクトルは  $(0, 1, 0)$  であるため、それに平行ならば法線ベクトルも一致し、

$$by = d$$

となります。つまりこの場合、 $x$  および  $z$  がどんな値を取ったとしても  $y$  の値は

$$y = \frac{d}{b}$$

で固定である。つまり  $XZ$  平面上に平行な平面というわけ。ちなみにこの時の

$$\frac{d}{b}$$

が  $XZ$  平面(もっといと原点)からのオフセット(ズレ)を表します。 $d=0$  のとき、この平面は  $XZ$  平面そのものとなります。ちなみに僕が

$$ax + by + cz - d = 0$$

のほうが

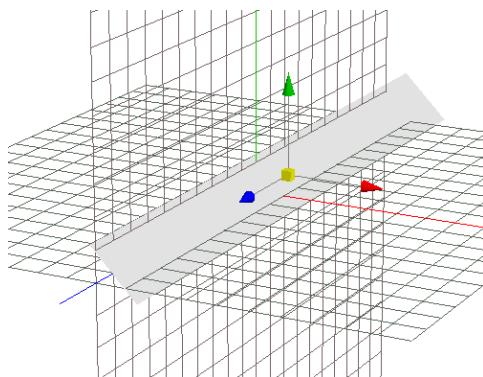
$$ax + by + cz + d = 0$$

よりも分かりやすいと言っている理由もそれです。ズレがベクトル方向へのずれになるからです。ちなみに「ズレ」を計測するときの法線ベクトルはもちろん「正規化」されてる必要があります。

例えば法線ベクトルが  $(1,1,0)$  のベクトルならば  $\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0\right)$  とすべきです。そうすると

$$\frac{x}{\sqrt{2}} + \frac{y}{\sqrt{2}} = d$$

という式になり、凡そ



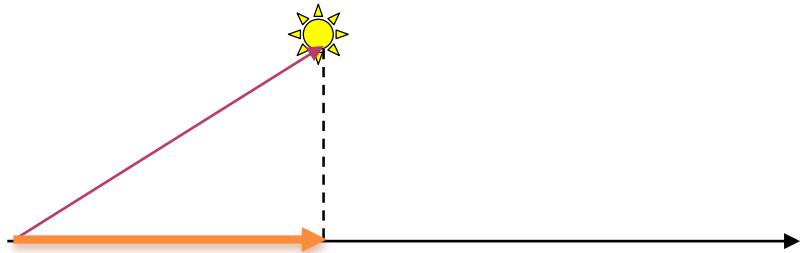
このような平面となるでしょう。ちなみにみやすさ重視で  $X-Y$  方向の広がりを狭く表示していますが、 $X-Y$  方向の広がりは無限です。

このとき、この平面は、原点から法線方向に  $d$  離れています。

この「正規化したベクトル」における  $d$  が原点からの距離…って意味は結構重要なのにもかかわらず教科書や参考書、高校数学ですら出てこないんだなあ…何ででしょうね。

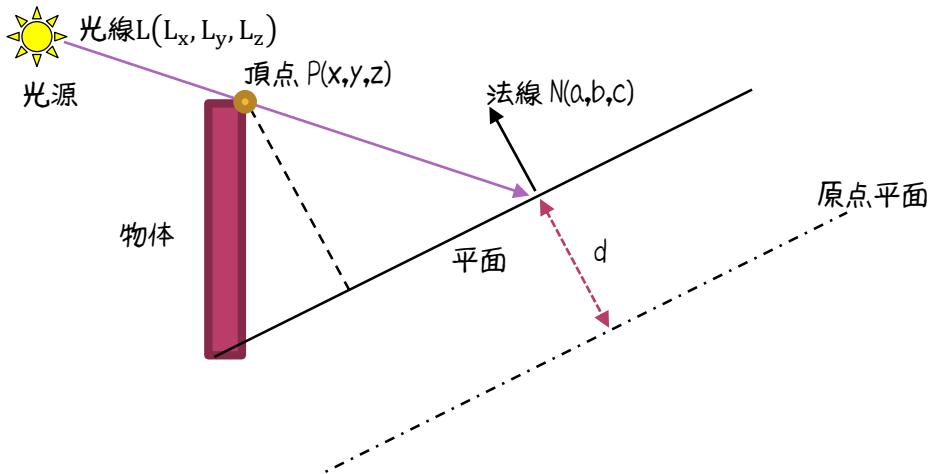
まあ、専門的なことはともかく「平面の方程式」が、「法線ベクトル」と「原点からのオフセット」でできていることはわかつただろう？

そこで XMMatrixShadow だが、これはとある平面に対する射影を作る行列だ。射影ってのは数学の時間にも習ったと思うが



こういうことだったと思う。ただ、今回は太陽が真上から当たっているわけではなく、斜めから当たっている。少しあやこしい。今回は平面に射影するので射影平面は分かっている。

※以下の図は、誤解を防止するために敢えて平面(直線)を斜めで書いている



大体こんな感じなのは分かるだろうか？

物体の特定の頂点  $P$  に対して法線ベクトルの内積を取ると、原点平面までの距離を求めることができる。実際の平面が原点から  $d$ だけずれていれば、頂点  $P$  の平面までの直線距離…つまり求めたい平面に対する垂線の長さは

$$P \cdot N - d$$

となる。…大丈夫かな？

さらに光線ベクトルと法線ベクトルの内積は「光線ベクトルの法線方向の長さ(単位)」を表します。

さて…ここで今度は3Dにおける直線(線分)の表現についても勉強しようか。直線の方は簡単です。

直線の方向ベクトルを $\vec{V}(v_x, v_y, v_z)$ とします(正規化済みベクトルとします)。そうすると直線の方程式ってのは

$$P(x, y, z) = \vec{V}t \text{ (ただし } t \text{ は任意の実数)}$$

なんんですけど、これは $y=ax+b$ における $y=ax$ というのと同様で、これでは原点しか通れないわけです。切片が必要なんですね。切片ってのも大したものではなくて「何処を通るのか」という情報なので、通る点がわかつていればそれで終わるわけです。

通る点を $P_0$ と置くと

$$P(x, y, z) = P_0 + \vec{V}t$$

これが直線の方程式です。 $P_0$ の座標から特定の方向に向かったベクトルに任意の数値をかけるということはそれを成り立つ全ての実数 $t$ を結ぶと直線になる。これが直線の方程式というわけです。

ていうか同時に「線分の方程式」なんんですけど…。線分と直線の違いは

- 直線の場合:  $t \in \mathbb{R}$   $t$  は任意の実数
- 線分の場合:  $a \leq t \leq b$   $t$  の範囲が明記されている

これだけです。

さてゴメンな?話が少しそれたかもしれないが、直線もベクトルで表せるわけだ。ここまではいいだろうか?

本題に戻ろう。

目的は特定の頂点の座標を光線方向に伸ばしていく、それが平面と衝突する部分を求めたいというわけだ。光線の直線はさつきも言ったように

$$P(x, y, z) = P_0 + \vec{V} t$$

とすると、 $P(x, y, z)$  は平面と衝突する座標。 $P_0$  は頂点の座標。 $\vec{V}$  は光線ベクトル。で、交点の  $t$  が求まりやれりんんですけど、連立方程式で解く？

いやいや……最初の式で  $P_0$  ～ 平面の距離が出てますよね？光線ベクトルの法線方向成分が分かってますよね？つまりライトベクトルを  $\vec{L}$  とし、法線ベクトルを  $\vec{N}$  とすると  $t$  は

$$t = \frac{\vec{P}_0 \cdot \vec{N} - d}{\vec{L} \cdot \vec{N}}$$

というわけです。つまり求めたい座標は

$$P(x, y, z) = P_0 + \vec{L} \frac{\vec{P}_0 \cdot \vec{N} - d}{\vec{L} \cdot \vec{N}}$$

となるわけです。

でも面倒なので `XMMatrixShadow` を使います。じゃあなぜ長々とこんな話をしたかというと後々この考え方は当たり判定で重要なからです。

ちなみに XNA Math 的には平面の方程式はやっぱり

$$ax + by + cz + d = 0$$

であるので、それに従ってパラメータを入力する必要があります。面倒だね。

ちなみに `XMMatrixShadow` 関数の中身は

```
N = normalize(Plane);
```

```
L = Light;
```

```
dotLN = dot(P, L)
```

$$\begin{pmatrix} N_a * L_x + dot_{LN} & N_a * L_y & N_a * L_z & N_a * L_w \\ N_b * L_x & N_b * L_y + dot_{LN} & P.b * L_z & N_b * L_w \\ N_c * L_x & N_c * L_y & N_c * L_z + dot_{LN} & N_c * L_w \\ N_d * L_x & N_d * L_y & N_d * L_z & N_d * L_w + dot_{LN} \end{pmatrix}$$

ちなみに平行光線の場合は  $L_w = 0$  になるらしいので

$$\begin{pmatrix} N_a * L_x + \text{dot}_{LN} & N_a * L_y & N_a * L_z & 0 \\ N_b * L_x & N_b * L_y + \text{dot}_{LN} & P.b * L_z & 0 \\ N_c * L_x & N_c * L_y & N_c * L_z + \text{dot}_{LN} & 0 \\ N_d * L_x & N_d * L_y & N_d * L_z & \text{dot}_{LN} \end{pmatrix}$$

となります。まあ忘れていいです。

正直言って、さっきの式からどうしてこれが出てくるのか…。

$$x' = x * N_a * L_x + x * \text{dot}_{LN} + y * N_b * L_x + z * N_c * L_x + 1 * N_d * L_x$$

→

$$x' = L_x (x * N_a + y * N_b + z * N_c + N_d) + x * \text{dot}_{LN}$$

→

$$x' = L_x * P_0 \cdot N_{abc} + L_x N_d + x * \text{dot}_{LN}$$

$$y' = L_y * P_0 \cdot N_{abc} + L_y N_d + y * \text{dot}_{LN}$$

$$z' = L_z * P_0 \cdot N_{abc} + L_z N_d + z * \text{dot}_{LN}$$

→

$$P' = \vec{L} * (P_0 \cdot \vec{N}_{abc}) + \vec{L} * N_d + P_0 * \text{dot}_{LN}$$

で、ぱっと見、似ても似つかないような感じなんですが、4行4列目を見てください。ここが同次座標系のwと同じ値になるんですよね。

同次座標系はx,y,zを全てそれで割ると辺縁が合うというものなので、

$$P' = \vec{L} * \frac{(P_0 \cdot \vec{N}_{abc} + N_d)}{\text{dot}_{LN}} + P_0$$

ここで $N_{abc}$ は法線ベクトルであり、 $N_d$ は原点からのズレであるから

$$P' = \vec{L} * \frac{(P_0 \cdot \vec{N} + d)}{\vec{L} \cdot \vec{N}} + P_0$$

まあ…辺縁は合うわけです。なるほど、分かりませんね!!!

おとなしく XMMatrixShadow 使いましょう。

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

*ShadowPlane*

(in) 基準面

*LightPosition*

(in) ライトの位置を記述する 4D ベクトル。ライトの w 要素が 0.0f の場合、原点からライトまでの光線はディレクショナル ライトを表します。ライトの w 要素が 1.0f の場合、ライトはポイントライトです。

基準面には平面の方程式における a,b,c,d を入れてやればいいです。普通は平面の法線ベクトルを正規化したものを a,b,c に入れればいいでしょう。

次の LightPosition ですが、これは文字通りライトの場所です。ところが平行光源の場合、ライトの場所って何なん?って思いますけど、それは各頂点から見たライトの場所ベクトルだと思つてればいいです。

つまり平面が原点を通り法線が(0,1,0)ならば

```
XMMatrixShadow( XMVectorSet(0, 1, 0, -0.1), XMVector4Normalize( XMVectorSet(-1, 1, -1, 0)));
```

なんて感じになります。

うまくいけば…



このように自分を平面につぶしたモデルが足元に表示されています。でもなにやら変ですね…というか気持ち悪いです。

さっきも言ったけど、ただ単に平面に潰しているだけなのでマテリアルとシャドウが見えちゃってるんですねえ…。

というわけで「影用ピクセルシェーダ」を用意します。いや、全然難しくないんですよ?

元のピクセルシェーダの関数とほぼおなじソースコードで「黒」を返せばいいでしょう。

例えば、そのピクセルシェーダの名前を EasyShadowPS とでもします。

そして影描画の時に切り替えれば…



ねっ？簡単でしょ？

これがMMDの中で使用されている簡易版の影描画です。なおMMDには「セルフシャドウ」というトグルがあるのですが、これがこの後にお話する「シャドウマップ」のことです。



これのことね。

今回作った影には弱点があって、所詮は本体をつぶして作っただけなので、常に足元にこの形で表示されます。つまり

- 凸凹した地面には使用できない
- 地面のない部分に影ができる
- 障害物の下に影が潜り込む
- キャラクターが地面の下に潜った場合も、全身の影が出てしまう

などの問題が発生します。なのでこれが使える範囲は限られていて、これが通用するのはバーチャルファイター2まででしょうね。



格闘ゲームはその性質上、地面が平坦なものが多いため、この手法が使われることが多かつたようです。『ストーク3』では地形が色々出てくるので、それに対応するようになりますが、それに対応するには次にお話するシャドウマップ(もしくはシャドウボリューム)の概念が必要だったわけです。



## 21.2 シャドウマップの概念

シャドウマップに必要な技術を最初に書いておきます

- ライトビュー
- テクスチャへの深度値の書き込み
- 深度値の比較

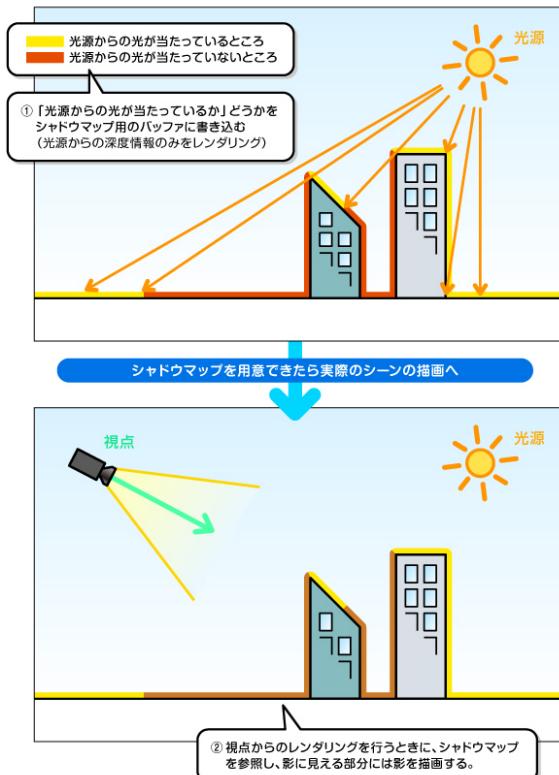
大雑把に言うとこんなもんです。「テクスチャへの深度値の書き込み」が一番の難関だと思しますがやるしかねえ。

でも割とイメージもしやすいため、そこまで苦労はしないと思いますが…まあ頑張ってやっていきましょう。

そもそも影が落ちるということはどういうことなのかというのを今一度考えてみましょう。

「影ができる」ということは「光源からの光線が遮られている」事に他ならないわけですね。

それについて <http://news.mynavi.jp/column/graphics/024/> の解説が分かりやすいとは思い  
ますが、この図が超分かりやすいので使いましょう。



この図に全てが書かれています。この図よりも分かりやすい図ってないんじゃないかな…。

まあ当たり前の話ですよね？図を見れば。

この「当たり前」の事を塔やって実装するのか…これ最初に考えたやつは天才だと思います。

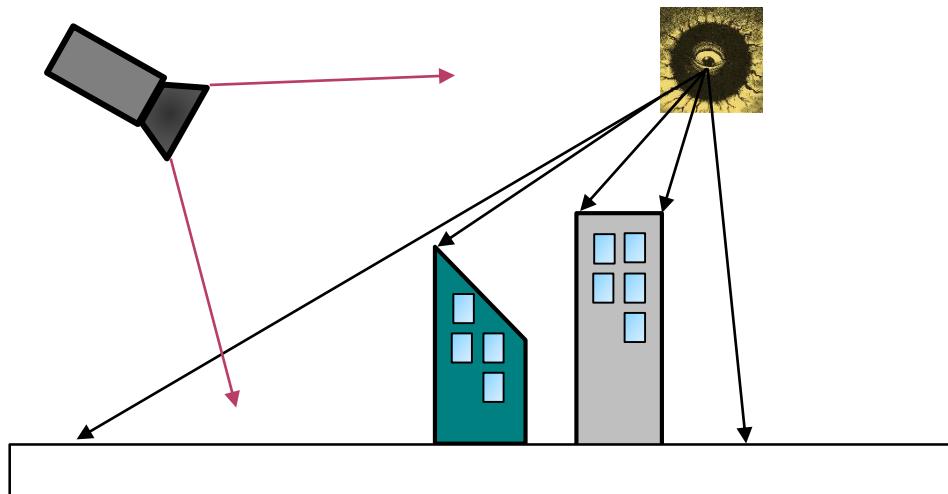
上の図のオレンジの部分は手前の物体に光が遮蔽されて光が届いてないわけです。

それはどういうことなのか？

そうですね。

光源を光の発生源であると同時に視点だと思ってください。

と、言うことは光源に視点を置いて、そこから見えているものにのみ光が当たるというわけだ。



見えないものには光が当たらない…そういうことだ。

なるほど、理屈はわかった。

- 光源から見えている&&カメラから見えている→明るく見える
- 光源から見えていない&&カメラから見えている→暗く見える
- カメラから見えない→問題外

こういうわけか。

さて、んじゃあ特定の点に対して「遮蔽されているかどうか」をどうやって判断したら良いんだろう？

二段階に分けて処理をすることになります。

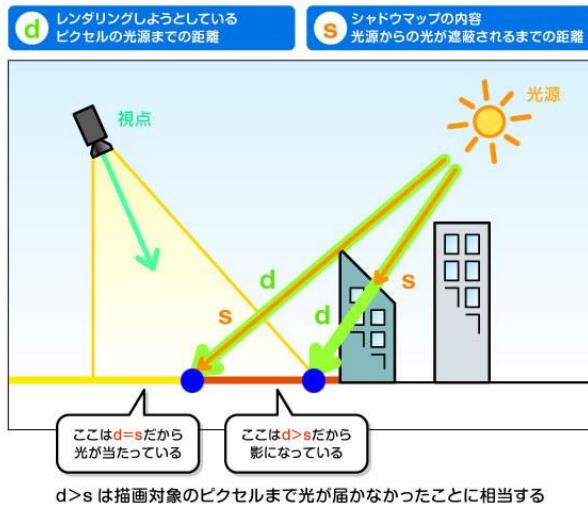
第①段階：ライトからのレンダリング（深度値）をテクスチャに書き込む

第②段階：通常のカメラレンダリングを描画する

この時の第②段階が重要。

描画の際に、描画しようとしている点が「ライトから見えているかどうか」をチェックしなければならない。この「チェック」とは何をチェックするのか？

そこでまたさっきのページの別の図を見てみましょう。



ホンマにこの図は秀逸ですわあ…。パーフェクト!!

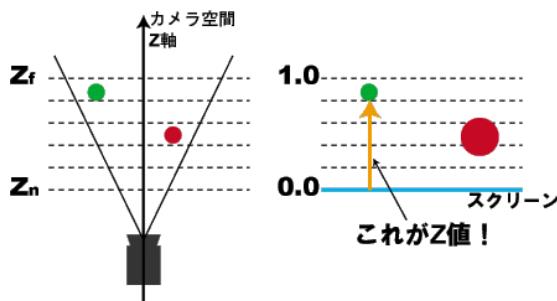
チェックとは…特定の点の座標の「光源からの距離」が「光線遮蔽物より遠くにあるのかどうか」である。

$s$  の内容は第①段階で計算されます。ただし  $0.0 \leq s \leq 1.0$  である。こうなる理由は後で話します。

じゃあ  $d$  はどうやって計算すれば良いんでしょう? 光源位置を  $L$  とし、対称座標を  $P$  として

$$d = \text{length}(P - L);$$

とてもするか…だがここで問題があるのだ。先程も言ったように  $0.0 \leq s \leq 1.0$  である。つまりこの単位系がぜんぜん違うのと似たような状態になっているのでこれに合わせなければならぬ。



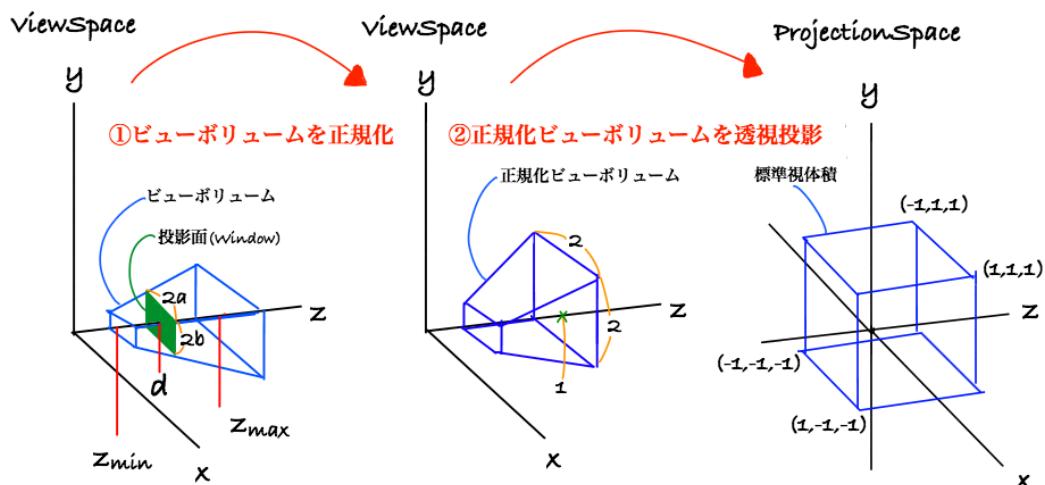
ちなみにそもそも何故  $0.0 \leq s \leq 1.0$  なのか?

<http://www.slideshare.net/todoroki/2-28983824>

よくわからんね…

<http://esprog.hatenablog.com/entry/2016/04/09/172925>

こっちのほうが分かりやすいかな…



そもそも射影行列ってのが、カメラから見た視錐台を「正規化デバイス座標系」に変換するものです。上の図のようにカメラから見たバーチャル空間を正規化デバイス座標系に変換し、それをスクリーンに書き出します。スクリーンは二次元平面なので $Z$ 方向は一番手前のもののみ描画します。

ところで画面を見ればわかりますが、どれも $-1 \sim 1$ の範囲になってますが、結局これって深度バッファに書き込む時に最終的に「テクスチャ」に書き込む必要があるので $0 \sim 1$ の範囲にしなければいけません。つまり $1$ を足して $2$ で割っています。

はい…ちょっと長々と書いたけどつまるところ、結果が $0 \sim 1$ になるようになっています。

もちろん比較関数もこれが必要なので、何をする必要があるのかというと、カメラから見たデプス値( $0 \sim 1$ に正規化された「光源からの距離」と、最初に光源から取ったデプス値を比較するわけです。

さて、比較というわけですが、ライトから見た「カメラ行列」で変換した値と、第①で取った深度値を比較すれば良いわけです。

## 21.3 演習準備

今回は「ガチの影」を作っていくので、影を落とす先が必要になります…つまりまずは床が必要になります。

また、遮蔽物があっても適切に影が落ちることを確認するために障害物も置きましょうか。

うーん。

メッシュと同じ頂点情報でやっちゃうと過剰情報な気がしますので、PMD表示用とは異なるものを作りましょう。

つまり…

Plane クラス

Cylinder クラス

を作りましょう。うへん。順調に進みすぎてるし、ひとまず仕様だけ言います。ちょっとクラス設計的な意味でね。

Plane は「平面」って意味で、Cylinder は円柱、円筒って意味です。

Plane はコンストラクタに幅と奥行を与えると法線(0,1,0)の平面の頂点バッファを生成します。

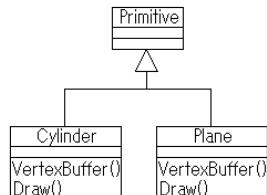
Cylinder はコンストラクタに半径と高さと分割数を入れると円柱の頂点バッファを生成します。

どちらも VertexBuffer() 関数で頂点バッファを返します。

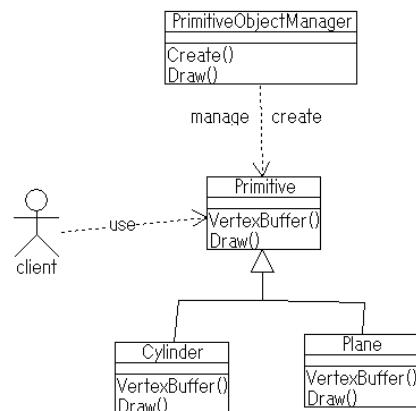
頂点バッファは返しますが、今回のこいつらはどうせプリミティブ(原始的な)形状なので Draw() 関数もこいつらに持たせてあげていいでしょう。

基本的にはこいつらは内部で全て完結するようにしてあげましょう。

また、内部で使用するシェーダはプリミティブ共通のものを使うようにすればいいでしょう。  
つまり…



う～ん。そうなってくると結局は PrimitiveObjectManager なんて作って… Create 関数と Draw  
関数作って…作ったプリミティブを操作する…って感じにするなら



こうでしょうかね。うーん、やっぱりこんな構造にするんじゃねえかよお前ふざけんな。

まあまあ、最初から完成品にする必要はないのだ。

ひとまずは最初に言った仕様を満たせば良い。

コンストラクタに必要なものを入れたら内部的に頂点バッファを作りましょう。

ひとまずコレの実装をお願いします。

```
///円柱
class Cylinder : public Primitive
{
public:
```

```

    Cylinder(float radius,float height,unsigned int div);
    ~Cylinder();
    ID3D11Buffer* VertexBuffer();
};

///平面
class Plane : public Primitive
{
public:
    Plane(float width,float depth,float nx,float ny,float nz);
    ~Plane();
    ID3D11Buffer* VertexBuffer();
};

```

とりあえず頂点情報のみで床と遮蔽物を表示できるようなシェーダとレイアウトを用意しましょう。

入れる情報は座標情報と法線情報と…uvだけでいいですね。頂点カラーのつけたい人は各自設定してください。

あとあと無駄なところは省くとして、ひとまずレイアウトと頂点バッファを内部で作りましょう。簡単な床から始めましょう。

で、頂点1つあたりの情報を格納するための構造体を作るんですが、コンストラクタで初期値を設定できるようにしといてあげると楽ですよ。

こんな感じで

```

///プリミティブ頂点型
struct PrimitiveVertex{

```

```

XMFFLOAT3 pos;
XMFFLOAT3 normal;
XMFFLOAT2 uv;

PrimitiveVertex(){
    pos.x = 0;
    pos.y = 0;
    pos.z = 0;
    normal.x = 0;
    normal.y = 0;
    normal.z = 0;
    uv.x = 0;
    uv.y = 0;
}

PrimitiveVertex(XMFFLOAT3& p, XMFFLOAT3& norm, XMFFLOAT2& coord){
    //入力変数名は、自分のメンバと重ならないようにするためにこんな名前にしている。
    pos = p;
    normal = norm;
    uv = coord;
}

PrimitiveVertex(float x, float y, float z, float nx, float ny, float nz, float u, float v){
    pos.x = x;
    pos.y = y;
    pos.z = z;
    normal.x = nx;
    normal.y = ny;
    normal.z = nz;
    uv.x = u;
    uv.y = v;
}

};

あとはそれぞれの(Plane や Cylinder)クラスのコンストラクタで頂点情報を作っていってやる。

```

まあ Plane は4頂点しか指定する必要が無いので楽だわな。

先程も言ったけど、この手の構造体の初期化を作つておくと便利な理由は

例えばプリミティブ頂点を宣言する時に

```
PrimitiveVertex p(-10, -0.2, 10.f, 0, 1, 0, 0, 0);
```

などと書けます。

更に言うと、vectorへのpush\_backなどでは

```
std::vector<PrimitiveVertex> vertices;  
  
vertices.push_back(PrimitiveVertex(-10, -0.2, 10.f, 0, 1, 0, 0, 0));  
vertices.push_back(PrimitiveVertex(10, -0.2, 10.f, 0, 1, 0, 1, 0));  
vertices.push_back(PrimitiveVertex(-10, -0.2, -10.f, 0, 1, 0, 0, 1));  
vertices.push_back(PrimitiveVertex(10, -0.2, -10.f, 0, 1, 0, 1, 1));
```

とでもすればいい。ここで頂点をつくって頂点バッファを返せばいい。あとはレイアウトと、シーケンスをセットすれば床も表示されるようになります。

あとは円柱はこの応用で作ってくれ…とはいいうものの、円柱の頂点を自動で作る方法とか知らないかもしないので、ヒントを言おうか。

- 円柱と言っても結局は正多角形柱
- ループを使う
- ループ回数は引数で渡した分割数
- 頂点位置には sin, cos を利用
- ループの要素が++されるたびに角度は  $2\pi / \text{分割数}$  進む
- 円柱側面は長方形
- 長方形は三角形2つぶん
- プリミティブトポロジが TRIANGLESTRIP であれば N 字を並べればよい
- プリミティブトポロジが TRIANGLELIST ならば長方形一つに 6 頂点…頂点の並びに注意
- 法線は中心から外側に向かうように伸びている…つまり円柱の中心から側面頂点まで
- UV は…まあよきにはからえ(考えるのが面倒なら貼らなくても良い)

```

std::vector<PrimitiveVertex> vertices(div*2+2);
for (int i = 0; i <= div; ++i){
    vertices[i * 2].pos.x = r*cos((XM_2PI / float(div))*(float)i);
    vertices[i * 2].pos.z = r*sin((XM_2PI / float(div))*(float)i);
    vertices[i * 2].pos.y = 0;

    XMFLOAT3 norm = vertices[i * 2].pos;
    XMStoreFloat3(&vertices[i * 2].normal,XMVector3Normalize(XMLoadFloat3(&norm)));

    vertices[i * 2].uv.x = (1.f / float(div))*float(i);
    vertices[i * 2].uv.y = 1.0f;

    vertices[i * 2 + 1].pos.x = r*cos((XM_2PI / float(div))*(float)i);
    vertices[i * 2 + 1].pos.z = r*sin((XM_2PI / float(div))*(float)i);
    vertices[i * 2 + 1].pos.y = height;

    vertices[i * 2 + 1].normal = vertices[i * 2].normal;

    vertices[i * 2 + 1].uv.x = (1.f / float(div))*float(i);
    vertices[i * 2 + 1].uv.y = 0.0f;
}

```

こんな感じかな?

とりあえず画面上に床と円柱を表示させてください。

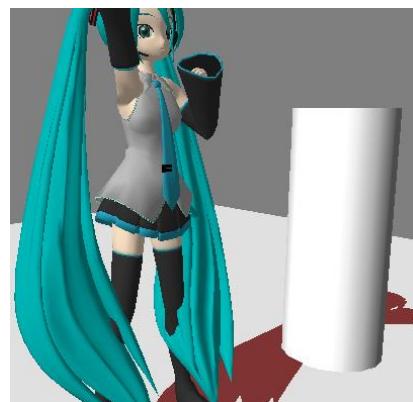
うまいこと行けば



こんな感じで床と円柱が表示されるでしょう。

予想したことですが、影が円柱の下に潜り込んで、不自然な感じになってしまいますよね？

それ以前に円柱が円柱に見えないので、余裕がある人は陰影もつけてください。



## 21.4 ファイティング

ちなみに床を描画した際に



このように「疑似影」がガサガサっとなることがありますこれが「Zファイティング」です。

<https://en.wikipedia.org/wiki/Z-fighting>

これに関してはなぜか Wikipedia に日本語の記事がないんですね。ちなみに Z バッファのことを「深度バッファ」というようになって久しいにも関わらず Z ファイティングだけは Z ファイティングのままなんですね。

ちなみに一部を Google 翻訳すると(最近の Google 翻訳はホンマ有能やでえ…)

ステッチングとも呼ばれる Z 戰闘は、2つ以上のプリミティブが Z バッファ内で類似または同一の値を有するときに生じる 3D レンダリングの現象である。2つの面が本質的に同じ空間を占める同一平面ポリゴンで特に広く使用されています。影響を受けたピクセルは、Z バッファの精度で決まる方法であるポリゴンまたは他のポリゴンのフラグメントでレンダリングされます。また、シーンやカメラが変更されると変化し、あるポリゴンが Z テストに“勝ち”、次に“別のもの”などになることもあります。全体的な効果は、スクリーンピクセルを着色するために「戦う」2つのポリゴンのちらつき、ノイズの多いラスタライズです。この問題は通常、限られたサブピクセル精度と浮動小数点数および固定小数点丸め誤差によって引き起こされます。

Z バッファの精度が高いほど、Z ファイティングに遭遇する可能性は低くなります。しかし、同一平面ポリゴンについては、是正措置がとられない限り、問題は避けられません。

近接クリッププレーンと遠距離クリッププレーンとの間の距離が増加し、特に近方平面が眼の近くで選択されると、プリミティブ間の Z ファイティングが起こる可能性が大きくなる。大規模な仮想環境では不可避的に遠方と前景の可視性を解決する必要性の間には固有の矛盾があります。例えば、宇宙飛行シミュレータで遠方の銀河を描くと、視界を解消する精度はありません。フォアグラウンドの任意のコックピットジオメトリ上に表示されます(数値表示であっても、Z バッファリングされたレンダリングの前に問題が発生します)。これらの問題を軽減するために、Z バッファの精度がニアクリップ面に向かって重み付けされますが、これはすべての可視性スキームでは当てはまらず、すべての Z ファイティングの問題を解消するには不十分です。

それでもなんとかマシにする方法…

Z-ファイトは、より高い解像度の深度バッファの使用、いくつかのシナリオでの Z バッファリング、または単にポリゴンをさらに離して移動させることで軽減できます。このように完全に

除去することができない $Z$ ファイティングは、ステンシル/バッファの使用、またはスクリーン上の投影形状に影響を与えない $Z$ が、影響を与える1つのポリゴンに変換後スクリーン空間 $Z$ /バッファオフセットを適用することによって解決されることが多い $Z$ ピクセルの補間と比較の際のオーバーラップを排除するための $Z$ バッファ値。ハードウェアで同じジオメトリ(たとえばマルチパス・レンダリング・スキーム)の異なる変換パスによって $Z$ -fightingが発生する場合、ハードウェアが不变の頂点変換を使用することを要求することによって解決することがあります。

深度バッファの不十分な精度によって引き起こされる $Z$ ファイティングは、単に世界の可視距離を減らすことによって解決することができます。これにより、近平面と遠平面の間の距離が短縮され、精度の問題が解決されます。しかし、スペースシミュレーターやライトシミュレーターなどの特定の仮想環境では、これは不可能です。これらの場合には代替技術が存在する。これらの技術の1つは、実際に位置を変えずに、ユーザーから離れたオブジェクトの距離を「シミュレート」することです。たとえば、最大安全表示距離(それを超えて $Z$ 戦闘が発生する)が10,000単位であり、レンダリングされるオブジェクトが15,000単位離れた場合、そのオブジェクトは代わりに10,000単位でレンダリングされますが、それに比例して縮小できますそれが動かされた距離。したがって、半分に縮小されたオブジェクトは、実際には2倍のように見えます。これがすでに近くにある、または最大視距離にあるオブジェクトに対してのみ行われ、ユーザーに近いオブジェクトが通常にレンダリングされる場合、この手法は目立たないはずです。 $Z$ ファイティングを減らすか完全に排除するために使用されるもう1つのテクニック

は、 $Z$ を逆にして対数 $Z$ バッファに切り替えることです。このテクニックは Grand Theft Auto V ゲームで見られます。コード化されているため、浮動小数点数はここで $Z$ を逆にすると、非常に遠いオブジェクトの深度を格納するときに精度が向上し、 $Z$ 戦闘が大幅に減少します。

といふわけや。

ここで「対数」という言葉が出てきたが、知ってるかな?  $\log_a b$  の事だよ。あまりにも数値の変化が激しい場合は「対数」を使うことがよくあります。計算オーダーで  $\log$  って出てくるんだけど、それ以外の部分で出てくるとはね。

○○デシベルって知ってるでしょ? あれも対数なんだ。

<http://macasakr.sakura.ne.jp/decibel.html>

…まあ、学生のうちはそこまでやることはないと思いますが、そういう世界もあるということでの、この職業を選ぶなら数学のあらゆる方面が仕事に関わってくると思っておいてください。

## 21.5 2つのレンダーターゲット

通常のレンダーターゲット以外にシャドウ用のレンダーターゲットが必要になります。なので作ります。レンダーターゲットに入れるのは影の深さ情報だけなのでこんな感じで作っておいてください。CreateShadowRenderTarget 関数とかで。

```
D3D11_TEXTURE2D_DESC rendertexdesc={};  
rendertexdesc.Width = 幅;  
rendertexdesc.Height = 高さ;  
rendertexdesc.MipLevels = 1;  
rendertexdesc.ArraySize = 1;  
rendertexdesc.Format = DXGI_FORMAT_R32_FLOAT;  
rendertexdesc.SampleDesc.Count = 1;  
rendertexdesc.SampleDesc.Quality = 0;  
rendertexdesc.Usage = D3D11_USAGE_DEFAULT;  
rendertexdesc.BindFlags = D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE;  
rendertexdesc.CPUAccessFlags = 0;  
rendertexdesc.MiscFlags = 0;  
ID3D11Texture2D* rtex=nullptr;  
dev->Device()->CreateTexture2D( &rendertexdesc, NULL, &rtex );  
D3D11_RENDER_TARGET_VIEW_DESC rtvdesc={};  
rtvdesc.Format=DXGI_FORMAT_R32_FLOAT;//深さ情報だけでいいのでこれで  
rtvdesc.ViewDimension = D3D11_RTV_DIMENSION_TEXTURE2D;  
result = dev.Device()->CreateRenderTargetView(rtex,&rtvdesc,&_rtvlight);
```

当然、文中の \_rtvlight はローカルじゃない変数で宣言してください。

このテクスチャにライトから見た世界を書き込みます。ただ、現在は平行光源であるためライトの「座標」なんてものはないのです。

## 21.6 ライトからの見た目

このため仮想的なライト位置を設定してあげます。距離的にはカメラと同じくらいの距離に置けばいいでしょう。

現在のカメラ位置を僕は

```
XMVECTOR eye = { 0, 15, -25 };
XMVECTOR target = { 0, 10, 1 };
XMVECTOR up = { 0, 1, 0 };
```

としているので…現在の視点と注視点との距離は

$$\sqrt{(0-0)^2 + (10-15)^2 + (1+25)^2} = \sqrt{701} \approx 26.5$$

というわけだから、これをライト方向に当てはめるとおよそ

(-15.3, 15.3, -15.3)

ただしこれは注視点からの距離なので注視点座標を足してやって…

(-15.3, 25.3, -14.3)

くらいに配置しましょう。ちなみにカメラが動く場合は注視点を移動させて本当は

(Target<sub>x</sub> - 15.3, Target<sub>y</sub> 15.3, Target<sub>z</sub> - 15.3)

としたほうがいいでしょうけど、ひとまず実験ということで固定値でやってみます。

CreateLightViewなどという関数でも作って…視点以外はカメラと同じ。ライトビュー行列を返す関数を作りましょう。

返す行列は

ライトカメラ行列 \* プロジェクション行列でいいです。

後はこいつを入れるためのメンバ変数 lightview を作ってやって

```
struct WorldAndCamera{
    XMATRIX world;
    XMATRIX camera;
    XMATRIX lightview; // 追加
```

```
};
```

そこにさっそく作ったライトビューを入れてください。

もちろんシェーダ側にも追加してください。

```
matrix lightview;
```

試しにカメラビューをライトビューに切り替えてみてください(今までのレンダーターゲットのままでいいです)

こんな感じになります。



## 21.7 レンダーターゲットの切り替え

うーんこのレンダーターゲット周りもシャドウマップが完成し次第クラス化したほうがいいですね。

まあそれは置いておいてレンダーターゲットを作つて、そして切り替えましょう。マルチレンダーターゲットだ!!!

ライトビュー用レンダーターゲットは作つてるので、そこに書き込むようにします。

どうやってやつたらいいのかといふと…簡単です。

```
OMSetRenderTargets
```

[https://msdn.microsoft.com/ja-jp/library/ee419706\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419706(v=vs.85).aspx)

を使うのです。これが切り替えになります。予めレンダーターゲットを2つ用意しておいて切り替えるだけなのです。

説明のために先に書かせた影用レンダーターゲットをちょっと改変します。いきなり影用の設定にしましたが、最初は通常のテクスチャとして使いましょう。そこでフォーマットを変更します。

テクスチャのフォーマットと、レンダーターゲットビューのフォーマットを通常のRGBAに変更します。

```
rendertexdesc.Format = DXGI_FORMAT_R8G8B8A8_TYPELESS;  
rtvdesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
```

これで通常のRGBレンダーターゲットとなります。あとはこれと連動する「シェーダーリソースビュー」を作ります。

## 21.8 影用シェーダリソースビュー

既にライトビュー用テクスチャデータはあるので、それを再利用します。

```
D3D11_SHADER_RESOURCE_VIEW_DESC srvdesc = {};  
srvdesc.Format = rtvdesc.Format;  
srvdesc.Texture2D.MipLevels = 1;  
srvdesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;  
result = device.Device()->CreateShaderResourceView(rtex, &srvdesc,  
&_shaderResourceViewForShadow);
```

見ての通りフォーマットはレンダーターゲットビューのものを流用すればいいのでそれほど手間はかかりません。

↑の\_shaderResourceViewForShadowがそのままテクスチャとして使用できます。

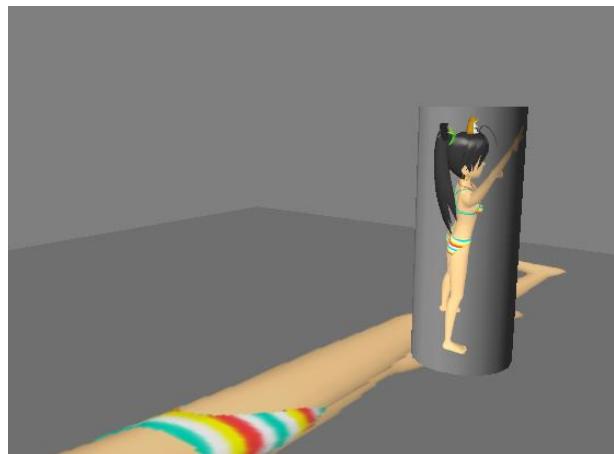
それでは今から次のことを試してみてください。

1. 影用レンダーターゲット&デプスをクリア
2. 影用レンダーターゲットをOMSetRenderTargetでセット
3. キャラクターを描画
4. カメラ用レンダーターゲット&デプスをクリア
5. カメラ用レンダーターゲットをセット

6. シェーダリソースとして『影シェーダリソース』をセット
7. 床＆円柱を描画

如何でしょうか？

うまくいけば



このようにレンダリング結果が『テクスチャとして』描画されます。

さあいよいよ大詰めなのです。

ここまで…理屈は分かったと思います。

今は手っ取り早く体験してもらうためにキャラの描画と床＆円柱を分けましたが本当は全オブジェクトを『2回描画』しなければなりません。

更に言うと影用シェーダも作らなければいけなくなります。細かく言うと検証のためのサフェイスの描画の必要性もあるでしょう…シェーダが増えますね。

ここからが結構大変なのです。道のりは長いのです。

## 21.9 ライトビュー用シェーダ

ライトビューシェーダを作ります。でも通常描画よりもシンプルなプログラムになるとは思います。基本は頂点情報だけでいいでしょう。最終的には深度値を入れることになりますがひとまずは普通に作りましょう。

最終的には深度値だけを取りたいので色情報も必要ないです。法線も必要ないです。

座標情報だけが正確ならばいいです。

このへんから通常レンダリングのシェーダから離れてしまい、管理しづらくなるため新しくシェーダを作りましょう。ライトビュー用シェーダなので `lightview.hlsl` とでもしてください。コンパイル時に怒られる人は、シェーダーディジョンを 5 にして、エフェクトファイルとしておきましょう。

確認のために座標情報のみで、白塗りで表示するシェーダを作ってください。

シェーダ名は `LightViewVS` とか `LightViewPS` とかにしつければいいでしょう。



そうするとこのような感じになるでしょう。これは `brightness=1` で

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

んで、この `brightness` をデプス値と関係のあるものにしてみたいのです。

```
brightness = o.pos.z / o.pos.w;
```

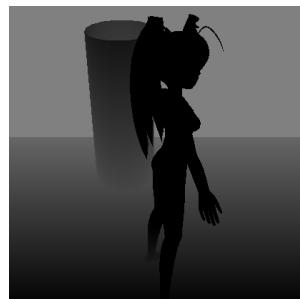
のですが、ちょっと問題があって…まあ見てもらったほうが良いと思います。



う~ん、くらい。試しに30倍くらいにします。



うん。おかしいですね。正規化済みの $\varepsilon$ 値ならば遠くに行けば行くほど1に近づくはずですが  
からホンマは



こんな感じやと思うんですよ。僕は。

CGソフトなんかもなんか、こういう $\varepsilon$ 値になるんですよね…(•ω•)…

納得行かない。

何でだろう…。これをほつといて良いものか…。

というわけで実験。

```
struct Output{  
    float4 pos:SV_POSITION;  
    float4 depos:POSITION;  
};
```

としておいて、posと同じものを deposに入れます。

そうすると

```
brightness = o.depos.z / o.depos.w;
```



なんですが、差がわかりづらいです。さっきより難しいのは、大きい値の方に差がないのです。  
こういう場合は乗数を使います。ほんにやらら乗です。

hslでほんにやらら乗ってどうやるのかってのは pow です。

[https://msdn.microsoft.com/ja-jp/library/bb509636\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb509636(v=vs.85).aspx)

100乗くらいしてみましょう。



こんな感じになります。

うーん。「違い」はあるようですがねえ…。まあ違いがあることがわかれれば良いので、とりあえず`z`値がこのように入っていることをご確認ください。

ところで、`POSITION` と `SV_POSITION` の違いなのですが、`SV` が `SystemValue` であること以外に明確な違いは見つけられませんでした。

[https://www.google.co.jp/search?hl=ja&q=SV\\_POSITION+POSITION+hlsl+%E9%81%95%E3%81%84&lr=lang\\_ja&gws\\_rd=ssl](https://www.google.co.jp/search?hl=ja&q=SV_POSITION+POSITION+hlsl+%E9%81%95%E3%81%84&lr=lang_ja&gws_rd=ssl)

そうは言っても、このままヨクワカラナイで済ませては教育者の名折れ。

公式リファレンスを熟読するのだ。

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

むむ…わからん。

無駄なのかもしれないがよく読んでみよう

POSITION[n]	均質空間内の頂点の位置。 $(x,y,z)$ を $w$ で除算することにより、スクリーン空間内の位置を計算します。各頂点シェーダーは、このセマンティクスを使用してパラメーターを記述する必要があります。
-------------	---

ちなみにスクリーン空間というのは以前に話した-1~1の空間の事だ。

SV_Position	均質空間内の頂点位置	ラスタライザー	float4
-------------	------------	---------	--------

どちらも「均質空間内の頂点位置」である。これは困った。

意味するところは同じであるにもかかわらず結果がまるで違うのである。これでは安心して使用できない。

そこでこのようなことを試してみた。

```
float4 pos:SV_POSITION;  
float4 lpos:POSITION;
```

とし、

頂点シェーダ側にてこう書く

```
o.pos = pos;  
o.ltpos = pos;
```

そして、こんな出力にする。

```
return float4(saturate(o.pos.z), saturate(o.ltpos.z), 1, 1);
```

両社に違いが“あれば”色に偏移(水色か紫色に偏る)が現れるはず。

結果…



目視不可能なレベル。では次に w を試してみる。

```
return float4(saturate(o.pos.w), saturate(o.ltpos.w), 1, 1);
```



同じく識別不能…といったい何がどうなってんの?と、思われるがよくよく考えてみるとし  
w や z の値の大半が 1 以上なら…。まあそういう事なんだな。だから z を w で割るわけなん  
だが、どちらにせよこの結果からわかるのは SV\_POSITION と POSITION 双方の z と w は 1 に  
近い値か 1 より大きな値であることは確実である。

うーん、こうなりや片っ端から調べてみる。一応 w は逆数を使うべきなのでこう書いてみる。

```
return float4(saturate(100/o.pos.w), saturate(100/o.ltpos.w), 1, 1);
```

100 は far 値を入れている。原理からすればこれで真っ白色になるだろう。ここで偏りが現れ  
れば…



残念…。ちなみに、 $w/100$  をするとこうなる。



言っておくが、これは色の偏りはないが一つ言えることは  $w$  がどうやら SV も無印も  $0 \sim 100$  の範囲内にありそうだということ。 $B$  成分が 1 だからだ。だが、ここからわかることが一つある。 $w$  値は遠い方が大きいという事だ。確かに視錐台を立方体状態にするのだから、そういうのだろう。しかし偏りはない…なんなのだ。では次の実験。 $z/100$  でやってみる。

```
return float4(saturate(o.pos.z/100), saturate(o.tpos.z/100), 1, 1);
```



おや…？なんか偏ってないか？試しに  $B=0$  にしてみよう。



こWWWWれWWWWはWWWW。見つけたぞ偏りを!!!R値がぼぼトドルやん!!!

というわけで、試しにこう書いてみる。

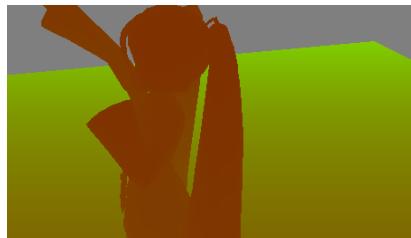
```
return float4(saturate(o.pos.z), saturate(o.ltpos.z/100),0, 1);
```

すると…



あつ…つまり pos.z の値は 1 近辺に集中している。これでもわかりにくく人のために、こう書いてみる。

```
return float4(o.pos.z/2, saturate(o.ltpos.z/100),0, 1);
```



もし双方が線形な違いしか持てない(比例関係)なら、こうはならないはずだ。近いときは赤が勝って、遠い方は緑が勝っているのである。

大雑把に言うと SV\_POSITION の z は 0~1 であるのに対して、POSITION の z は 0~100 であることがうかがえる。

ちなみに前の実験により、w の範囲は双方ともに 0~100 である。そりや結果が全然違うことになるわあ…コワイ…こわいわあ。

しかし MSDN の説明はどちらも「均質空間」と書いてある。MSDN は誤訳でない限りは嘘をつかないはずだ…つまり行間を読むしかないのだ。

ちなみに僕の勤労感謝の日(11/23)のツイートです。

つちのこ @tsuchinokoboxer 3分  
もう・・・それしかない・・・学生への説明は

つちのこ @tsuchinokoboxer 8分  
むむっ！わかったぞ！！MSDNを読むと、POSITIONはwで割るとスクリーン空間内の位置を計算するという事だ。そしてSV\_POSITIONはそのことが書いてない・・・どちらも「均質空間」とは書いているもののそこに違いがある。SV\_POSITIONは既にwで割られた結果である？

つちのこ @tsuchinokoboxer 8分  
MSDN熟読中・・・行間まで読まんとわかりそうにない

つちのこ @tsuchinokoboxer 10分  
つまり、  
pos=WVP\*vpos  
svpos=WVP\*vpos  
とやってピクセルシェーダで  
svpos.z\*svpos.wとpos.z/pos.wを見ると、結果が全然ちゃうよね。  
ラスタライザが関係してるのかしら。

おわかりいただけるだろうか…土日もろくに休めないこのワシがたまの祝日の合間に何をやっているのがツイッターしてんじゃねえかよ)。

くっそ…次年度就職年次のくせに休みとあらばゲームしやがって…プレイの喜びを知りやがって…お前ら許さんぞ!!!

祝日における人の自由を剥奪しやがって、プレイの喜びを剥奪しやがって…許さんぞ。

『あんなゲーム作りたいな、こんなゲーム作りたいな』って言いながら作ってへんのやろ!んで、年度末の就職活動でそ、そ、そういうのやってないから作品がないんやろ。

まあつまるところ…だ。POSITIONのxyzはwで割ると、スクリーン空間内座標になるが、SV\_POSITIONはそうではない…そういうことだ。MSDNも明記しろよそのへん!!



ということは、SV\_POSITION.zはwで割らなくていいのだろうか？

じゃあ SV\_POSITION における w はなんなんだよ…色々聞いて回ったが分からんかった。これはもう MS に直接聞いてみてもいいレベル。

ちなみに、そもそもなぜ z/w なのかというと、

[http://marupeke296.com/DXG\\_No55\\_WhatIsW.html](http://marupeke296.com/DXG_No55_WhatIsW.html)

を見てください。

いつもの状況であれば座標というのは  $(x, y, z, 1)$  となっているものですが、

最終的に

遠い面の左下座標が

$$(-Z_f, -Z_f, Z_f, w = Z_f)$$

となっているんです。これであるため z/w にする必要があるわけです。

というわけで全成分を  $w = Z_f$  で割るとちょうどいい座標になるわけです。結局のところ、  
SV\_POSITION はラスタライザを経由した時に w で割られてしまっていると考えるのが妥当  
なようです。

とはいっても、拳銃の確認が得られていないので、今回は POSITION の方を返すようにしましょう。

つまり

```
return shadowpos.z/shadowpos.w;
```

というわけだ。

このライトシェーダで一回描画…あとは本体側シェーダに切り替えて(同時にレンダーターゲットも切り替え)描画…この時に、ライトシェーダが描いたテクスチャを利用するのである。

## 21.10 ライトビューテクスチャの利用

ライトビューのレンダーターゲットは同時に shader リソースビューに描画されているため、  
既にそのテクスチャは用意されている。

…とりあえず 3 番レジスタにでもこれを割り当ててくれ。

```
dev.Context()->PSSetShaderResources(3, 1, &_shaderResourceViewForShadow);
```

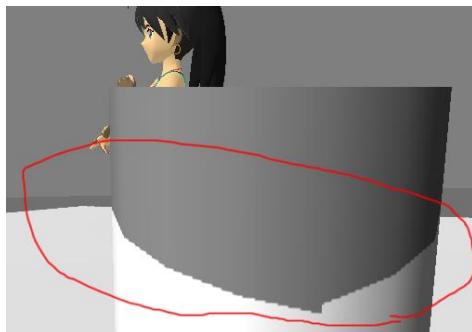
3番に割り当たっているので、シェーダ側では3番を見ればいい。簡単なことだね。

シェーダ側には

```
Texture2D shadowtex : register(t3);
```

とでも書いておけばいい

あとはこのテクスチャを利用するわけだが、まずはきちんとした値が入っているのかどうか確認してくれ。



なんかこんな感じでデプス値が出てればOKだ。利用できるという事だ。

## 21.11 ライトビューテクスチャのUV値は？

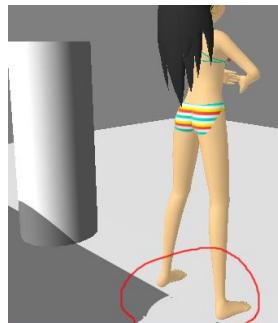
ひとまずちょっと考えるのが難しいのがライトビューテクスチャのUV値だ。

とりあえず、カメラ行列の中でもライトビュー行列を持っておいて、それをposに欠けたものをshadowposだかlightviewposだかに代入しておいてほしい。

とりあえず POSITION のxyをwで割ることでスクリーン空間内の座標になるわけだが、



これは-1~1であり、uvとして使うには不向きだ。よって例によつて、1を足して2で割ろう。そうすれば0~1の範囲になる。



あ、そうか…スクリーン空間はまだ空間なので上がプラス…しかしUVのVは下がプラス…つまりY方向はマイナスなのである。

つまるところ、こう書くべきなのだ。

```
uv=(float2(1,1)+(shadowpos.xy/shadowpos.w)*float2(1,-1))*0.5;
```

ここまでではいいだろうか？

このUVを使えば対象地点でのライトビュー $z/w$ が得られるというわけだ。

## 21.12 いよいよ比較だ

ちなみにカメラからのビューでもライトビュー行列を使ってあらかじめ計算した $z$ と $w$ を使って、これと比較する。

```
float Id = outputp.shadowpos.z / outputp.shadowpos.w;
```

この値と、テクスチャの値を比較する。 $Id$ の方が $\neq$ かければ影であるから、何かしら暗くする処理を書いてくれ。

すると…



何このノイズ…

それはだね。

ソファイティングがまたもや起きているのだ。

何故かといふと、物体表面は、影を落とす頂点そのものもあるので、ちょっと細工しないと表面が影になってしまふのである。

よって…

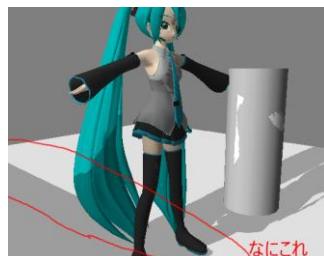
```
if ( |d>f.r + 0.0001f) {
```

このようなショボレ対処ですまないが、下駄をちょっとはかせる。



これでオッケー。

ちなみに



こういう謎の影が出ることがあるが、これは精度の問題であることが多い。なので **R32\_FLOAT** のテクスチャを使うようになっているが確認してくれ。

ちなみに、まだ誤魔化しているところがあって、頭の部分にもなんか変な影が入っているんだよね。

これ(変な影)原因がよく分かってない。どつかでしくじっているんだろうけど…

ちょっとすまんが、原因究明はもう少し待って欲しい。

ともかく、ここまで『シャドウマップ』というものの理解と実装ができたことだろうと思う。

おめでとう！！

ついに君はスキニングだけでなく  
影まで制御できるようになったのだ！



## 22 デバッグ用 HUD を作る

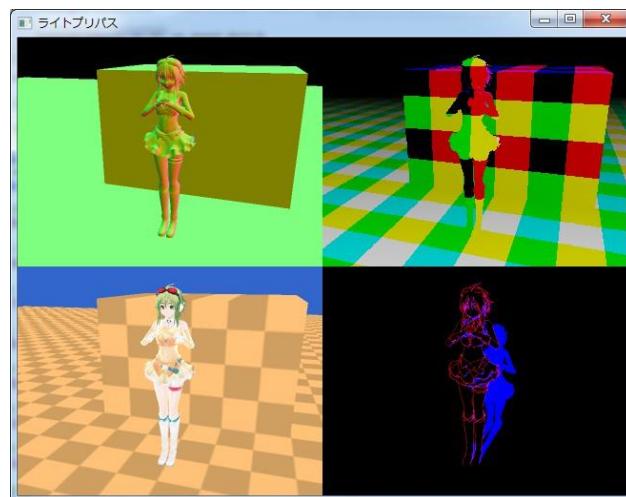
さて、ここまで基本の部分は解説したので一旦暫休めとして、デバッグ用 HUD を作りましょう。

<https://ja.wikipedia.org/wiki/%E3%83%98%E3%83%83%E3%83%89%E3%82%A2%E3%83%83%E3%83%97%E3%83%87%E3%82%A3%E3%82%B9%E3%83%97%E3%83%AC%E3%82%A4>

HUD というのは UI みたいなもんで、画面上に表示されるものだと思ってればいいです。格闘ゲームにおけるゲージ類もそのように呼ばれます。

本来は戦闘機のコックピットに表示する情報表示のことを HUD と言うので、今回のこれを HUD といって良いのかどうか分かりませんが…。

ちょっと直近で必要だと思うのは今回のような「マルチパスレンダリング」をやっている場合、それぞれの出力がどうなってるのか確認できます。っていうか必要です。



このようにね。

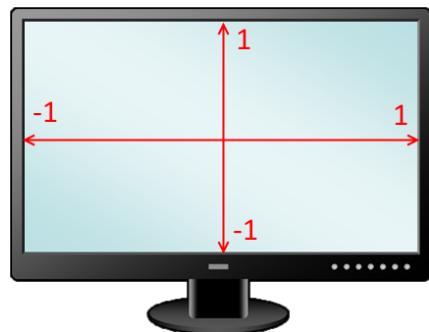
さて、手始めに Z 値を出力する事にしましょう。

どんな風に実装しましょうか…？

2D に対する配置です。まだ話していませんが「ビルボード」として配置するのも手です。ですが今回の HUD に関してはそこまでするまでもないでしょう。

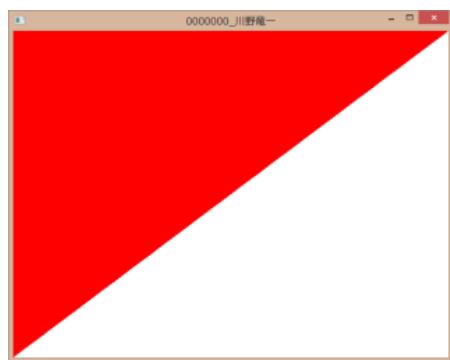
2D は 2D として配置すれば良いのです。

配置をピクセル単位で指定できるようにしましょう。一番最初の描画を思い出してください。

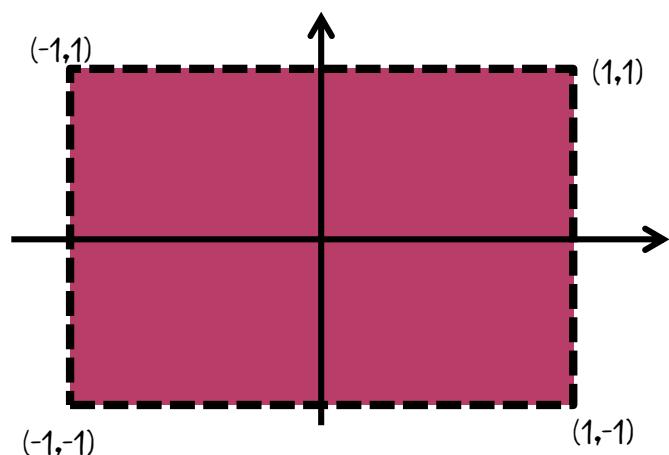


こんな感じでしたね？

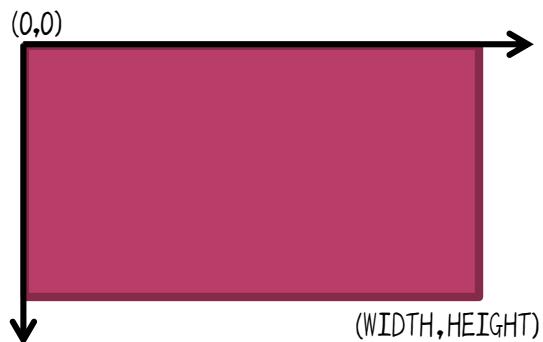
左端が-1、右端が+1、上端が+1、下端が-1で…



こんな感じのやつたでしょ？これを DxLib で 2D 作ったときのようにピクセル指定できたらよくないですか？できるんです。っていうかここまで来たら分かるよね…。



この状態から…



にしたいわけだ。

$$(0,0) \rightarrow (-1, 1)$$

$$(\text{WIDTH}, \text{HEIGHT}) \rightarrow (1, -1)$$

になるような…変換はどうしたらいいんだろう？

ひとまず幅と高さを正規化する必要がありますので

$$\div (\text{WIDTH}, \text{HEIGHT})$$

とすると

$$(0,0) \rightarrow (0,0)$$

$$(\text{WIDTH}, \text{HEIGHT}) \rightarrow (1, 1)$$

となる。

幅が足りないので2倍します。

$$*(2,2)$$

とすると

$$(0,0) \rightarrow (0,0)$$

$$(\text{WIDTH}, \text{HEIGHT}) \rightarrow (2, 2)$$

正規化に逆づけるために-1します。

$$-(1,1)$$

$$(0,0) \rightarrow (-1, -1)$$

$$(\text{WIDTH}, \text{HEIGHT}) \rightarrow (1, 1)$$

垂直方向反転のためYだけ-1します。

\*(1,-1)

とすると

(0,0)→(-1,1)

(WIDTH,HEIGHT)→(1,-1)

1. 幅と高さで割る
2. 2倍する
3. -1 する
4. Yだけ反転する

となります。

これを一つの行列で書くならば…どうなるでしょうか？

## 22.1 HUD 行列を作る

ちょっと自分で考えてみてください。考えろよちくしょう!!!

こんな感じ。

$$(x \ y \ z \ 1) \begin{pmatrix} \frac{2}{WIDTH} & 0 & 0 & 0 \\ 0 & \frac{-2}{HEIGHT} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 1 & 0 & 1 \end{pmatrix} = \left( \frac{2x}{WIDTH} - 1, 1 - \frac{2y}{HEIGHT}, z, 1 \right)$$

こういう行列を作ってください。そしてこういう行列を返す関数を作ってください。

XMMATRIX CreateHudMatrix(画面幅,画面高さ)

でいいからよおおおお!!!

## 22.2 HUD シェーダ作る

VSもPSもレイアウトもゼーんぶまとめて作りましょう。可能な限りシンプルに作るのが良いね。

今まで作ってきたから簡単でしょ？今回は座標情報とUV情報さえあればいいので…。

あと、world にさっきの CreateHUDMatrix でも放り込んで座標と乗算。uv はそのままね。

HUD 用のアウトプット構造体も用意してあげましょう。SV\_POSITION と TEXCOORD 成分さえあれば十分でしょう。ひとまず HUDVS と HUDPS を作って実行確認してください。できたら実際にシェーダオブジェクトを作りましょう。

いっぺんに頂点シェーダとピクセルシェーダとレイアウトを作るの、戻り値は HRESULT の…引数参照にしてあげるのがいいでしょう。

```
HRESULT CreateHUDShader(ID3D11VertexShader*& vs, ID3D11InputLayout*& layout, ID3D11PixelShader*& ps){
```

こんな感じね。レイアウトは POSITION と TEXCOORD のみで OK

```
D3D11_INPUT_ELEMENT_DESC desc[] = {  
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },  
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },  
};
```

## 22.3 HUD 頂点データ(頂点バッファ)作る

はい、CreateHUDVertexBuffer ね。

あとはやること一緒だけど、この関数の指定は左上座標と幅、高さになると良いね。つまり

```
ID3D11Buffer* CreateHUDVertex(float left, float top, float width, float height){
```

こんな感じです。

## 22.4 ループの前で全部そろえとく

```
XMMATRIX hudmatrix = CreateHUDMatrix();  
hudbuffer = CreateHUDVertex(0, 0, 320, 240);  
ID3D11VertexShader* hudvs = nullptr;  
ID3D11InputLayout* hudlayout = nullptr;  
ID3D11PixelShader* hudps = nullptr;
```

```

unsigned int hudstride = sizeof(float) * 5;
unsigned int hudoffset = 0;
HRESULT r = CreateHUDShader(hudvs, hudlayout, hudps);

```

## 22.5 描画

で、全部の描画が終わった後で…HUD の描画をするんだよ!!!

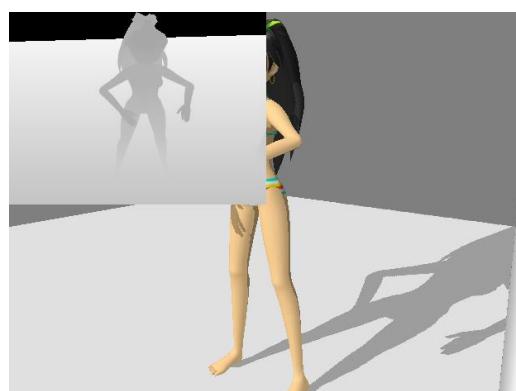
```

dev.Context()->VSSetShader(hudvs, nullptr, 0);
dev.Context()->PSSetShader(hudps, nullptr, 0);
dev.Context()->IASetInputLayout(hudlayout);
dev.Context()->IASetVertexBuffers(0, 1, &hudbuffer, &hudstride, &hudoffset);
dev.Context()->Draw(4, 0);

```

こういうことやな。描画前に world への行列反映も忘れずに…。

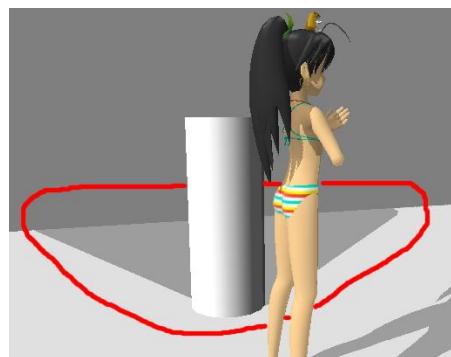
うまくいけば…



こうなります。

正直邪魔なんで「D キー」とかでトグルできるようにしましょう。

で、問題の部分…



円柱の影がおかしい部分について…。

デバッグ画面を見てみましょう。



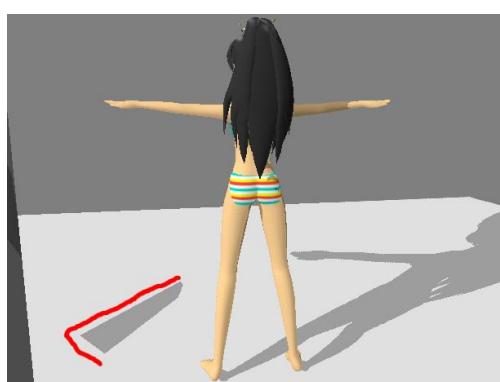
おー、

といふわけでこの画像から、予測できる原因を考えてみましょう。

円柱がライトビューではほとんど「写っていませんね」。試しにカメラビューの比較のところにこう書いてみてください。

```
float2 satuv = saturate(uv);
if (uv.x==satuv.x && uv.y==satuv.y && Id>f.r + 0.0001f){
    shadowWgt = 0.7f;
}
```

これで0~1からはみ出た部分に影が落ちなくなります。結果として…



ライトビューに「写ってない」部分から影が消えます。

結局はそういうことだったのです。

さて、対処はどうしましょうか？

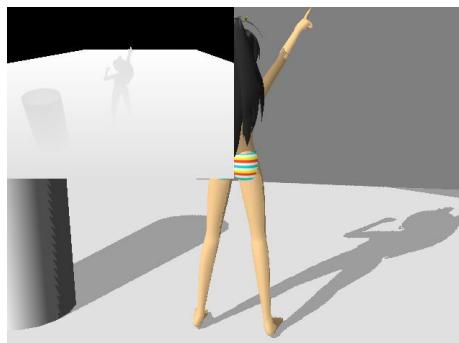
## 22.6 おかしなシャドウへの対処

- ライトビューを引く
- 画角を変更する
- Orthogonal にする
- 平行光源をやめる(点光源にして減衰させて誤魔化す)

などがあります。

## 22.7 ライトビューを引く

ライトビューを対象物から離します。そうすることでカメラに入っているビューボリューム全体がライトビューの範囲に入ります。



ご覧のようにカメラに映るべきオブジェクトがライトビュー内に入っているため影も正常に表示されます。デメリットとしては右上に見えている影…つまり離したことにより一部がライトビュー外に出てしまい、変な影が見えててしまうことです。

じゃあ、クリッピングボリュームを広げりや良いじゃん。確かにその通り。その通りなんだが、そうすると今度はボリュームが広がってしまった分、 $\epsilon$  値の制度は下がってしまうのだ。これはわかるね？

near～far が近がろうと、遠がろうと、結局は 0～1 に正規化されてしまうため、当然 near～far が近いほうが精度が高く、遠いと精度が下がるわけだ。

## 22.8 ライトビューの画角を広げる

次に画角をいじる方法だが…画角を広げれば広い範囲が見えるようになる。ライトビューにもこれを適用してやれば良いのではないか?



ライト～注視点までの距離はそのままにライトビューの画角を広げました。きちんと出ていますね?これだと $\alpha$ 値の精度は変わらないため良さそうに見えますね?

ただしこれも $\alpha$ 方向の精度はいいんですが、ビューボリュームの台形が広がっているため、今度はXY方向の精度が下がります。ちょっとスクショじゃ分かりにくいですが、ジャギが出やすくなるのがデメリットです。

最後に、もっと思い切ってorthogonal(平行投影)にしてみましょう。

## 22.9 Orthogonal(平行投影)

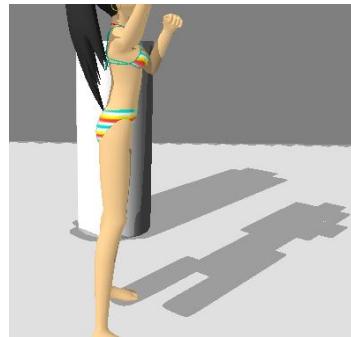
平行投影とはなんでしょうか?

ここまでやってきた影は、光源から離れれば離れるほど広がるものでした。

しかし、平行光線の場合、よくよく考えるとおかしなことでもあります。

というわけで平行投影(遠くに行っても大きさが変わらない)でライトビューを作ってみましょう。

どうなるでしょうか…?



アヒヤヒヤヒヤ(°A°)ヒヤヒヤヒヤヒヤ

あれー？

ちなみに正射影(平行投影)の関数は

XMMatrixOrthographicLH

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

なんですかね？

ちなみに

XMMATRIX proj = XMMatrixOrthographicLH(640,480,1,100.0f);

という具合に呼び出しています…。

＆値見てみましょか…。



なるほどね…。ちなみに縦横に関しては、光線なので同一比率の方がいいです。50,50くらいにしてみましょう。



( ^ω^)おつつつつつつつつ

これはほれい! 感じ!! イイ感じすぎる!!!! これやな!!!!

## 22.10 点光源もどき

ちなみに光源から離れると明るさが減衰するようにつくるとこんな感じです。



う~ん。これ以上減衰させると暗くなりすぎるしなあ… 問題の箇所はどうだろ…

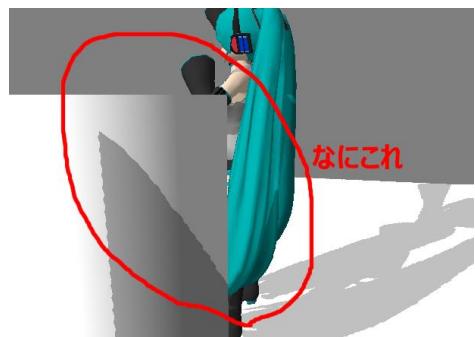


うん。やっぱり誤魔化せませんね。カメラを引くか画角を広げるか平行投影にしたほうがよさそうです。

まあ、それで解決としましょう、一旦。

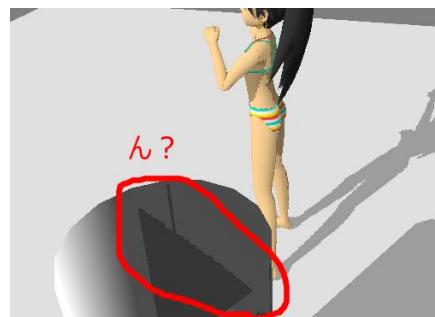
## 22.11 まだまだおかしい円柱の影

しかしまだまだこういう意味不明な影が…残ってます。



…これは…なんでしょうね？

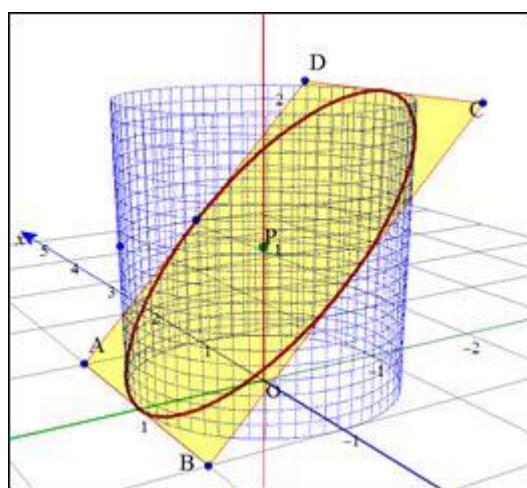
んで、これカメラYを上げるとわかるんですが…



何だろう…これ？

これ気づくのに一時間半かかりました。

こういうことです。



数学のテストでよくある「アレ」です。

フツーに考えれば円柱であればさっきのような影の落ち方しないんだけど、今のこの円柱は「蓋がない」わけです。

ということは、ライト手前に見える部分からスパッと切ったように光が当たって、また、影が落ちるわけです。

結果的に、取得した $\alpha$ 値の向こう側を暗くしている以上、↑の図のような影が入るわけです。ちなみに、円柱に蓋をすると…



のように、より適切に影らしく見えます。うーん、セルフシャドウと陰が乗算されると暗くなりすぎるので調整すると…



いい感じですね。

さて、今度こそシャドウマップはOKでしょう。で、またまたコーディングが大変な状況になっていると思いますので、リファクタリングしましょう。

## 23 何度目のリファクタリングだ？

もう把握できていませんが…何度目でしょうか。

相当に汚らわしいコードになっているでしょう？キレイキレイしてあげましょう。

さて、ここまで来ると色々と複雑なことになっていると思います。そろそろ全体的なことを考えつつリファクタリングしましょう。

### 23.1 ちょっと突っ込んだ基本の話

で、設計の話とかしたい所なんですが、

C++の場合はですね。設計どうこうする前に、テンプレートなどの挙動について知っておく必要がある必要があるわけです。それを勉強するのに都合いいサイトが

[https://ja.wikibooks.org/wiki/More\\_C%2B%2B\\_Idioms](https://ja.wikibooks.org/wiki/More_C%2B%2B_Idioms)

英語で良ければ

<http://www.dre.vanderbilt.edu/~sutambe/documents/More%20C%2B%20Idioms.pdf>

のドキュメントを落とすがいいよ。

あと、↑はハードルが高いかかもしれないんで

[http://marupeke296.com/CPP\\_main.html](http://marupeke296.com/CPP_main.html)

でも見ておくと良いよ。設計手法よりも

<http://qiita.com/kenjhiranabe/items/9eddc70e2798b1992274>

とかこういう細かい原則を心がけたほうがいいんじゃないかな。あとはこれね

<http://www.slideshare.net/MoriharuOhzu/ss-14083300>

あと、毎年『デザインパターン信者』が出てくるんですが、アレはあまり気にしない方がいいです。

別にデザインパターンが悪いとかじゃないんですが、正直ネットに蔓延してアレ(GoF)はもう古いんですね…



きしда<sup>昌</sup> @kis · 24時間

GoFの本は、歴史的価値しかなくなってると思うなー。基本的には1990年ごろのC++で関数的なことをどう実装するかというもので、いまだとだいたい「関数オブジェクト渡せばok」みたいなものが多いし、使う機会がないパターンも多いし。なるべく薄いダイジェスト本を読めばいいのではないかと

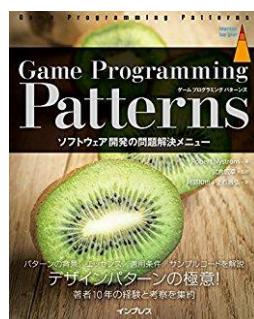
Java の申し子のキッサーもこう言ってることですしそ…



こういう本ね。昔は価値があったかもしれないけど…今はインターネットで全く同じ内容を見るし、思いし時代を懐かしく、古いし…若いみなさんが読むものではないです。

だいたい23個も覚えてられつかよ。

ちなみに



こういう本も出でていて、この中で紹介されている GoF のパターンは

- コマンドパターン
- フライウェイトパターン

- オブザーバーパターン
- プロトタイプパターン
- シングルトンパターン
- ステートパターン

です。確かにこれらは使った覚えがあります。ただ

## デザインパターンは何かを解決するものではありません。

これは覚えておいてください。フリーにオブジェクト指向っぽく、読みやすく作ったら勝手にそういうパターンになっていた…それだけです。くれぐれも信者にならないように。

ちなみにこの本は今は「Kindle Unlimited 対象本」なので、もし Kindle Unlimited にしてる人は DL して読んでみましょう。

あと、若い君達が冬休み中に読んでおくべき本があるとすれば…



これだろう。3800 円(税込で 4104 円)とお高いが、

<https://matome.naver.jp/odai/2142304549947328601>

こんなキチガイどもが課金している額に比べれば微々たるものです。キチガイ課金もやってる間は楽しいが、サービス終了すれば無価値です。キチガイ課金者も無価値な人間です。粉碎しましよう。

でも、EffectiveC++にかけたお金は自分の血肉になります。自分の価値が上がります。

次年度就職年次の人们は、今の自分が置かれた状況を考えれば…分かるだろう?



さて…話がそれ気味なので戻しましょう。

今回 C++ の設計の話ですが他の言語実装系と違い、メモリのことや DirectX の細かい話が絡んできます。更に言うとそもそも C++ の仕様についても話していないことはまだまだあります。

テンプレートとかコピー構造体とか仮想関数のメモリ消費とかその他色々なことについて…とはいえあんまし細かいこと考えすぎててもゲームは完成しないので、完成を第一目標に掲げて頑張りましょう。

今日は「テンプレート」と「コピー構造体」について簡単にお話します。

### 23.1.1 テンプレート

これは Java でもジェネリクスなどと言われている機能だが、C++ では「引数」や「メンバ変数」の型を特定せずに関数やクラスを作ることができる。これが「テンプレート」である。

[https://ja.wikipedia.org/wiki/%E3%83%86%E3%83%B3%E3%83%97%E3%83%AC%E3%83%BC%E3%83%88\\_\(%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9F%E3%83%B3%E3%82%B0\)](https://ja.wikipedia.org/wiki/%E3%83%86%E3%83%B3%E3%83%97%E3%83%AC%E3%83%BC%E3%83%88_(%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9F%E3%83%B3%E3%82%B0))

この何がおいしー( ^q^ )のかというと、何ていうかなあ…アルゴリズムは同じだけど、型ごとにいちいち関数を作りたくないときってあるやん？

たとえば A+B を表す Add 関数を作るとする。

そうすると

```
int Add(int a, int b){  
    return a+b;  
}  
  
float Add(float a, float b){  
    return a+b;  
}  
  
double Add(double a, double b){  
    return a+b;  
}  
  
string Add(string a, string b){  
    return a+b;  
}
```

なんて書かなければならなくなるわけやん？でもそんなのもったいねいやん？やってることは同じやし。

そういう要望にお答えして出てきた文法が『テンプレート』である。実は既にテンプレートの恩恵は得ていて、それは STL の vector や map なのね。あれって色々な型に使えるでしょう？

まあというわけでさっさと文法いくやで。

**template<typename T> T 関数名(T 引数, T 引数);**

ポイントは“template”と“typename”や。あとで話すがこの typename は class と書いちやつても良い。

`template` は「ここからテンプレート関数もしくはクラスを記述しまっせ」とて意味。

`typename` は「俺の後に続く文字をテンプレート仮引数として使うやで」とて意味。

この `T` という文字は「なんでもいい」。別に「`T`」じゃなきゃいけないわけじゃないので

`template<typename R> R 関数名(R 引数, R 引数);`

でも構わないし、1 文字制約もないないので

`template<typename SONIKO> SONIKO 関数名(SONIKO 引数, SONIKO 引数);`

でも構わない。ただ、意味が無いので大抵は 1 文字だし大抵は `T` である。理由は「テンプレート仮引数」であることを示すためにやっぱり `T` にしこうってなるわけ。

そんだけ。

このテンプレートを用いればさっきの多数の関数が

`template <typename T> T Add(T a, T b){`

`return a+b;`

`}`

と書ける。つまり

`template <typename T>`

と書いたら、ここ以降の文節は `T` の型を「今は決めないよ。誰かに使われる時に決めるよ」という意味になる。

ここでちょっと注意点、といふか、中身の話だが…「今は決めないよ。誰かに使われる時に決めるよ」とはどういうことだろうか？

これはコンパイル時に

```
int ret=Add(4,5);
```

という文があったとすると、こう書いた時点でのコンパイル時に

```
int ret=4+5;
```

と展開されているようなもので、C言語のマクロ関数に挙動が近いのである（マクロよりは安全ではあるが）。

つまり、コンパイル前ではこいつは関数のような振る舞いを見せるが、コンパイル後ではそこに関数の中身が生成されているといえる。違いがわからないかもしれないが、通常の関数だと、別の番地に関数を作っておいて、呼び出し時にそのアドレスにジャンプしているため、いくつ呼び出しても呼び出しコード以外は増えないのだがテンプレートの場合は、この関数がコンパイル時に実体化されてしまうため使いようによってはコンパイル後コードが肥大化してしまうということがある。

もしかしたら最新のコンパイラでは改善されているかもしれないが、割と諸刃の剣的な機能なのである。

で、テンプレートの話はまだ終わってなくて、テンプレートには関数の引数としてだけではなく、クラスのメンバ変数の型としても指定できるわけ。これをテンプレートクラスという。

例えば

```
template<typename T> class A{  
    private:  
        T _member;  
    public:  
        T Function(){ return _member};  
};
```

という風に使用できる。

template<typename T> class または struct クラス名もしくは構造体名{

テンプレート仮型 T を使った変数や関数を記述

};

使いでがなさそうだが、実は非常に使えるのだ。vector や map もこの機能を使って生成されている。

例えば、僕が実装してよく使ってたのが『スコープドポインタ』である。

例えば、new や delete で動的確保せざるを得ない変数があったとする(もしくは特定の関数から動的確保されて返され、その解放責任はこちら側にある場合など)… そういう場合に使用する。

どういう事がいうと…

```
template <class T> class ScopedPtr{  
  
private:  
    T* _ptr;  
  
public:  
    ScopedPtr(T* ptr=nullptr):_ptr(ptr){}  
  
    ~ScopedPtr(){  
        if(_ptr!=nullptr){  
            delete _ptr;  
        }  
    }  
};
```

などとする。そしておくと例えばクラス A ってのがあったとする。で

```
ScopedPtr<A> a(new A());
```

などとスコープドポインタとして宣言してやれば、この `a` 自体は動的確保ではないためスコープを抜けた時点で自動で `delete` がかかり、解放忘れを防ぐのだ。

わかりづらいかもしだれなしが

`ScopedPtr<A>`

はこの時点で `ScopedPtr<A>` という「型」であり、\*も何もついてないので、ポインタではなく通常の変数として扱われる。通常の変数であればスコープ外に行く時に自動で解放されるためデストラクタが呼ばれる。

→そこで実体の `delete`

となるわけだ。

例えば

```
class A{
public:
    A(){
        printf("A 生成");
    }
    ~A(){
        printf("A 解放");
    }
};
```

とでも書いてやって

`A* a=new A();`

の挙動と

`ScopedPtr<A> b(new A());`

の挙動を比較してみると良い。

ただしこれで終わりではない…色々とまだまだやらねばならないことがあるのだ。

ScopedPtr が『ポインタのように』動作するには例えば型 A に Func() という関数があつた場合、

```
b->Func();
```

とやって呼び出せるようにする必要があるわけです。今のままで呼び出せません。どのようにすればよいのでしょうか？

実は大した話ではないのです。オペレータオーバーロードを使えば良いのです。ああ、オペレータオーバーロード自体知らなかつたっけ？どつかで話した覚えはあるんだけどなあ。

オペレータってのは演算子のことです。このオーバーロード…つまり +-\*=/<><<>>などの意味を変更できるのがオペレータオーバーロードです。set の話をした時にちょっとやつたよね？

ちなみに Java はこのオペレータオーバーロードを否定してるので好きになれません。

で、実は -> もオペレータの仲間なのです。で、この場合どうすれば良いのかというと…。

ScopedPtr 内部に

```
T* operator->()const{
    return this;
}
```

など…自分自身を返す関数を作ります。で、これは後々解説しますが、二重の意味を持つ事になります。

例えば

```
b->Func();
```

というコードはプリコンパイル時に

```
(b.operator->())->Func();
```

と解釈されます。もうわからんねえな、これ。つまり

```
b->Func();
```

は

```
b._ptr->Func();
```

と同一というわけで、通常のポインタのように動作させることができです。他にポインタに必要な演算子はありましたつけ？

そう\*演算子ですね。これはポインタの実体の「値」を返すものです。乗算の演算子と違うところがポイントです。この辺の使い分けもオペレータオーバーロード使いにとっては悩みどころかもしれませんのが、慣れるしかないです。

これも簡単です。

```
T& operator*() {  
    return *_ptr;  
}
```

とします。参照を返しているのはメモリの節約と、返した先で値が変更されても反映されなくなるのを防ぐためです。

ここまでやれば「ポインタとして振る舞うようになるわけです。しかも自動解放付きでね!!

ええやろ？

あ、実体の生ポインタを返したいこともあるだろうから Get 関数も作つとくな。

```
T* Get() const{  
    return _ptr;  
}
```

一応ソースコードのつけとくんで参考にしたい人は参考にしてや。

```

template <class T>
class ScopedPtr{
private:
    T* _ptr;
    ScopedPtr(const ScopedPtr& );
    ScopedPtr& operator=(const ScopedPtr&);

public:
    ScopedPtr(T* ptr=0):_ptr(ptr){
    }
    ~ScopedPtr(){
        if(_ptr){
            delete _ptr;
        }
        _ptr=0;
    }
    ///ポインタのリセット
    ///@param ptr スコープ管理したいポインタ
    void Reset(T* ptr=0){
        if(_ptr || _ptr!=ptr){
            delete _ptr;
        }
        _ptr=ptr;
    }
    ///生ポインタを返す
    ///@retval 管理しているポインタ
    ///@note 他のライブラリとのやり取り等のために必要
    T* Get() const{
        return _ptr;
    }
    ///メソッドを求められたとき
    T* operator->() const{
        return _ptr;
    }

    ///実体もしくは参照を返す
    T& operator*() const{

```

```

        return *_ptr;
    }

///このオブジェクトの実体はNULL(0)か?
bool IsNull()const{
    return _ptr==0;
}

///ピクリマークが来たらNULLかどうか調べてるってことだから…
bool operator!()const{
    return IsNull();
}

///持っているポインタを交換する
void swap(ScopedPtr<T>& ptr){
    T* tmp=ptr._ptr;
    ptr._ptr=_ptr;
    _ptr=tmp;
}

};

ちなみに、typenameとclassの違いは僕の知る限り
```

## 「とくにないです」

寧ろ知つたら教えて欲しい。

一応慣例として、classは対象の「型」が構造体やクラスである事が期待されるときに、typenameはそうとは限らない時に…って事のようです。

覚えるの面倒なので typenameだけ使っておけばいいよとも思います。

できれば↑のコードでコピー・コンストラクタと代入演算子を禁止していることも注意して欲しい。

スコープドポインタをコピーしたり代入したりするのはスコープがめちゃくちゃになる恐れがあるのと、二重解放の危険性があるからだ。

ちなみに今回お話ししたポインタは「スマートポインタ」の一種です。

今回のよりもずっと便利で堅牢な「スマートポインター」は C++ の memory に入っているので、時間があればご紹介したいと思います。

多分、一番使うのは `shared_ptr` です。これは特定のポインタを参照している人間が全滅したら初めて解放するというものです。

仕組みを簡単に言うと、構造はさつきの `ScopedPtr` とほぼおなじなんだが、内部に「参照カウンタ」なるものを持っており、他の `shared_ptr` に代入されるたびに参照カウンタが上がっていきます。

で、そいつがスコープを離れたり解放されたりするたびに参照カウンタが減っていき、ゼロになつた瞬間に解放されるというものです。

<http://qiita.com/hmito/items/db3b14917120b285112f>

の例を見てみると良いです。今後詳しく話したいとは思っています。

次にコピーコンストラクタの話をしよう

### 23.1.2 コピーコンストラクタ

そもそも「コピー」とは何で、「コピーコンストラクト」とはどういうことか?を話しましょう。

コピーコンストラクトが発生するタイミングは

A a;

(中略)

A b=a; // この時にコピー(コピーコンストラクタ)が発生

b=a; // この時はコピーコンストラクタではなく代入が走っている。

うーん、正直言うとコピーコンストラクタは自動で実装されるんですが、自動で実装されるやつは

```
A(const A& in){
```

```
    *this=in;
```

```
}
```

なので代入と変わりません。

だからコピー構造関数自体は、どうかの内容を元にオブジェクトを作る時になんかしらの加工をしたい時に使うことになるでしょう。

とりあえずは「代入とは違うのだよ」と思っておけばいいです。

## 23.2 コーディング規約

いよいよ僕が『コーディング規約』の話をやる時期に入りました。嫌なんですよ、コーディング規約を学生に言うの。なんか固まっちゃうから…『正解』だと思ってほしくないから。

あまりコーディング規約を設けたくないのだが、最低限のガイドラインは言っておく。

- 関数の行数は 100 行以内
- Warning を無視しない。可能な限り黙らせる努力をしよう
- インデントはタブにすること
- マジックナンバーを使わないこと(コメントでごまかさない)
- public 関数には必ず関数コメントを入れること
- 5 行以上の同じコードのコピペが発生する場合は必ず関数化する
- ヌルは nullptr を使うこと
- ベクトル/行列に関しては必要に応じてオペレータオーバーロードを定義していい
- 理解の及ぶ限り STL などの便利な機能を使おう
- キャストは interpret\_cast 以外は積極的に C++ キャストを使おう
- 何処か解説サイトや本のコピペコードは決して使用しない
- 変数名、関数名はローマ字不可。調べても英語を使用すること
- クラス名の頭に "C" とかつけないこと
- ハングarian 記法なんか使用しないこと
- 変数は名詞句の名前にすること(ただしプール型の is〇〇 はあり)
- 関数は動詞句の名前にすること(Draw なんとかだの Update なんとかだの)
- クラス/構造体名は Actor や Monster などの名詞を使用
- for 文は可能な限り『範囲ベース』の for 文にする
- 名前の付け方やコーディングスタイルは一貫させよう
- auto を使って問題ない部分は auto を使おう
- 関数の引数や関数に const をつけられる場合はつけよう

- 構造体を引数や戻り値として使用する場合は可能な限り参照を使おう
- クラス名について考える時に Manager や Wrapper や Helper に飛びつかない

逆に、

- インデントの空白数
- 中括弧{}の位置
- その他こまけーこと

に関しては言及しない

## 23.3 設計

次に設計の話だ。これも当然のようにする気はなかったが、しよう。

ただし

- MVC(ModelViewController)などを考えない←初心者にはちょっとハードル高い
- 結果的にどつかのパターンになることもあるが、パターン前提では考えない
- 「現状の問題」と「ほんのちょっと先」だけを見据えてクラスを設計する

### 23.3.1 典型的なやつ

さて、クラス設計となるとどうやつたらいいのか分からぬものなんだが、そこで手が止まるのはよろしくない。クラス設計の代表的なものとして、シナリオ法ってのがある。

[https://www.ogis-ri.co.jp/otc/hiroba/UMLTutorial/frm\\_cls.html](https://www.ogis-ri.co.jp/otc/hiroba/UMLTutorial/frm_cls.html)

<http://www.oki-osk.jp/esc/go/class.html>

クラス設計のシンプルな手法としてまず、シナリオ(アプリの挙動を日本語で書いたもの)を書いてその中の「動詞」を関数に、その関数の主語を「クラス」に割り当てます。

あんまし細かいものは「変数」で良いと思います。

で、シナリオを書くと

「予めロードしておいた PMD データから GPU 用のデータ(頂点バッファインデックス/バッファ)へ変換し、表示用ビュー変換をかけて画面上へモデルを表示する。」

これが基本で、アニメーションや座標変換を用いるならば

「予めロードしておいた PMD データから GPU 用のデータ(頂点バッファインデックス/バッファ)へ変換し、ロードしておいて VMD データからフレームごとのクオータニオンを抽出し、ボーン変換、ワールド変換を行い、表示用ビュー変換をかけて画面上へモデルを表示する。」

これがスキニングまで。シャドウまで入ると

「予めロードしておいた PMD データ(PMDData)から GPU 用のデータ(頂点バッファインデックス/バッファ)へ変換し、ロード(Load)しておいて VMD データからフレーム(Frame)ごとのクオータニオンを抽出し、ボーン変換、ワールド変換(Transform)を行い、ライトビュー(LightView)変換をかけてシャドウ用 RT に描画(Draw)し、それをもとに影(Shadow)を画面上へモデル(Model)を表示(Draw)する。」

なんですが、流石に細かすぎなんですよね。細かすぎて伝わらない！

「PMD データからメッシュデータを取り出し、そのメッシュデータを VMD によりスキニングし、平行移動や回転などの処理をそれに対して行った上で本体と影を描画する。」

くらいでいいと思います。ただ、「本体と影を描画」って部分は実際本体描画の際に必ず描画されるため分けられるものではない。

つまり、この辺に関しては実装しなきゃ分からなかったわけよね？通常の考え方で行くなら分けちゃうでしょ？

だから最初にあんまりガチガチのクラス設計するのはオススメしないわけ。だからプロがクラス設計する場合は、よっぽど何度も作っている構造でない限りは一旦 ある程度作ったらぶつ壊してもう一回作り直したりします。

ある程度実装系の特性が分かってないと不適切な設計になっちゃうわけです。

この辺がそのへんのSEとは思想が違うわけですよね。だから最初から設計って言いたくないんですよ。

さて…流石にここまで作ってくるとなんとなく DirectX の思想は…思想のかけらくらいは分かってきたんじゃないだろうか？

ちょっとまとめてみよう。まずは名詞

- PMDData
- VMDData
- Frame
- Camera(CameraView)
- Light(LightView)
- RenderTarget
- ~~Shadow~~

次に動詞

- Load
- Transform
- Draw

…うーん。正直この「名詞」「動詞」法は考えるための取っ掛かりにはなるけど、実際のクラス設計はこれにとらわれないほうが良いんですね。

ちなみに実装的な部分で留意しなければならない点が

- マテリアル切り替え
- シエーダ切り替え
- レンダーターゲット切り替え
- プリミティブトポロジ切り替え
- バッファーカッティング

など、切り替え系には留意する必要があります。ここをしくじると全然違う結果になるからね。

となってくると

キャラクターやオブジェクトごとには Draw 関数を持っておきたい→でもレンダーターゲットが違う…ということは Draw 関数の引数にレンダーターゲットを渡してはどうか…ちなみに、切り替え系はそれなりにコストがかかるものです。

なるべく減らしたいので、Draw に切り替え系を入れるのはオススメしないかも…ということで Renderer なんていうクラスを作つて、そいつに切り替え命令を出すというのが良いでしよう。

あ、ひとつここで誤解のないように言っておくと、今回の設計はあくまでも「**整理するため**」のものです。**エンジンを作るためでも、簡単にするラッパーを作るわけでもない**です。

設計をする時は目的をはっきりさせておくことが大事です。

何でもかんでもやりたくなっちゃうのがプログラマだからね。自制しないと…。

うーん。とりあえずは切り替えるべきレンダーターゲットとかシェーダは Renderer にもたせておきましょう。Renderer が肥大しすぎるようならまたその時に考えましょう。

また、PMD は Model クラスでラップして Draw で描画するようにします。

あ、描画の際にマテリアルを切り替えない SimpleDraw とかも作つておきましょう。

で…、今は WinMain にたくさん書いてもらってますが、Application ってクラスを作つて、そこの中に Run って作つて、その中にメインループを作りましょう。

基本的にメイン関数にあまりコードを残しません。結構大規模なリファクタリングになります。

動がなくなることも十分に考えられますので慌てずに行きましょう。

ちなみに、エンジンやライブラリを作りたい人は、他の人が公開しているエンジンやライブラリの中身を見て参考しましょう。

なお、オープンソースじゃなくても、ヘッダーファイルのインターフェイス部分だけを見ても相當に参考になると思いますので、どしどし参考にしましょう。ここではやらないけどもね…。

ただ、一流のプログラミングスタイルを知りたい人は見ておいたほうが良い。下手な本よりも参考になる。

ちなみにご存知のように Unreal Engine 4 はソースコード公開している。

また、KLab も自社エンジン「PlayGround」を公開している。



Silicon Studio 社も C# ではあるが、オープンソースのエンジンを出してて「Mizuchi」と言います。

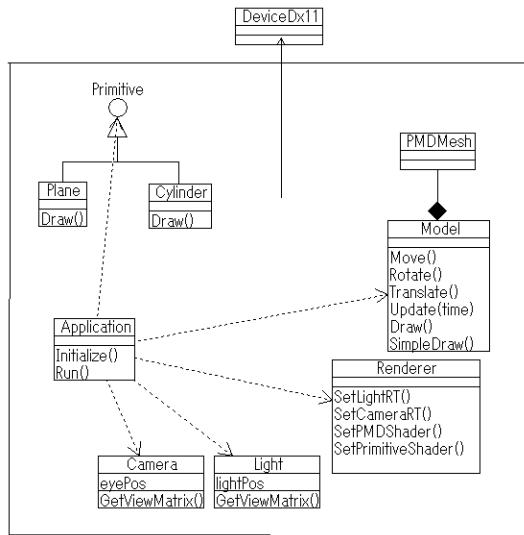
海外のことでよければ Godot などというエンジンもある。これもオープンソースだ。

あとは Irrlicht Engine などというものもある。

このあたりの話は Wikipedia の情報は古いので、自ら情報を収集しておくことをおすすめする。

それはさておき

今回の整理のためのクラス図は大雑把に書くとこんな感じ。



このようにクラス図自体は書いたけど、この通りになるとも限らないので注意ね。

あー、ちなみに自作 ScopedPtr をサーバに置いといたので必要に応じて使ってくれ。何でこれを使いたいかというと [Pimpl イディオム](#) を使う時に便利になるからだ。基本的にヘッダに実体を置きたくないのだ…マジで。

別に SharedPtr でも良いんだが、ちょっと大げさすぎるんだよなあ…あれ。

ちなみにクラス化していく順番は、末端からのほうが良いと思います。影響度合いがゆるい方向から行ったほうが良いです。全体に影響を与えそうな Application は最後にした方がいいでしょう。

最初は Model からやったほうが良いでしょうね。

とりあえずヒントとして、Model.h を書いておきます。

```

#pragma once
#include "ScopedPtr.h"

class PMDMesh;
struct ID3D11Buffer;
  
```

```

struct _XMFLOAT3;
typedef _XMFLOAT3 XMFLOAT3;

///@brief PMD等のモデルの操作
///Draw,座標などを制御するクラスです
class Model
{
private:
    ScopedPtr<XMFLOAT3> _pos;
    ScopedPtr<XMFLOAT3> _rotation;
    ScopedPtr<PMDMesh> _mesh;
    ID3D11Buffer* _materialBuffer;
public:
    Model(const char* filePath, ID3D11Buffer* materialBuffer);
    ~Model();

    ///@brief 場所を設定する
    ///@param x X座標
    ///@param y Y座標
    ///@param z Z座標
    void SetPosition(float x, float y, float z);

    ///@brief 回転を設定する
    ///@param x X軸回転
    ///@param y Y軸回転
    ///@param z Z軸回転
    void SetRotation(float x, float y, float z);

    ///@brief 今いる場所から移動する
    ///@param x X移動量
    ///@param y Y移動量
    ///@param z Z移動量
    void Move(float x, float y, float z);

    ///@brief 現状から回転する

```

```

///@param x X軸回転
///@param y Y軸回転
///@param z Z軸回転
void Rotate(float x, float y, float z);

///モデル内時間を1フレーム進める
void Update();

///@brief モデルの通常描画
///@note マテリアルごとにインデックスを分け
///マテリアルを反映させつつ描画します
void Draw();

///@brief モデルの簡易描画
///@note 全インデックスを同一マテリアルで描画します
///@remarks マテリアル切り替えしない分ドローコールが減らせる
void SimpleDraw();
};


```

ちなみに`_materialBuffer`としてマテリアルバッファを内包してはいるが、あくまでも参照用です。`PMDMesh`が複数種あったとしてもマテリアルバッファは一つでいいので…、こいつは最終的には`Renderer`にでも持たせておきましょう。今ん所は`main`のどつかにおいといて構いません。

あと「`Getter,Setter`を使うな」という思想に反していますが、

<http://www.slideshare.net/MoriharuOhzu/ss-14083300>

の66ページ目ね

この部分はその次のリファクタリングにとっておきましょう。このルールを貫くのは少しむずかしいので、もう一いちよレベルアップしてからのほうがいいでしょう。

で、この過程で`PMDLoader`をシングルトンにしたくなると思います。していいです。後から外せば良いんです。柔軟に行きましょう。目的は「整理すること」です。

移行した上で動作チェックしてください。

迷いどころは VMD との組み合わせかなーと思うんですが、ほんのちょっと先の事を考えると、アニメーションはひとつだけではないので

「アニメーション名」と「VMDData」の組み合わせをいくつか持つておいて、いつでも参照できるようにしておきたい。そして「いくつか」なので、うーん…map かな。

```
void RegisterAnimation(アニメーション名, アニメーション);
```

を作りましょう。で、中でそのマップに追加してください。そうするとアニメーション切り替え可能にすることが楽になるでしょう。

また、Model の Draw の中身は描画するだけではなく、頂点ノーファのセット、インデックスノーファのセットを入れておいたほうが良いでしょう。同一頂点ノーファを連続して複数回描画することは今のところないです(同一キャラ複数描画が発生した時点でその時に考えればいいでしょう)

さて、そんな感じで、Camera→Light→Renderer と言った具合にクラス化していきましょう。

で、カメラを作ろうとしてふと…思ったことが…やっぱり xnamath 系は難しいなあと

### 23.3.2 XMATRIX に悩む

XMMATRIX を使うときにも

```
ScopedPtr<XMATRIX> _projection;
```

などとしてコンストラクタ時に

```
_projection.Reset(new XMATRIX);
```

としたかったのだが、皆さんご存知の通り

これには問題がありましたね？覚えてますか？

そう。

16バイト境界にしないとアカンやつでした。

```
typedef _declspec_align_16_ struct _XMMATRIX
```

こういうやつです。

では…どうすればいいのでしょうか?

<https://msdn.microsoft.com/ja-jp/library/83ythb65.aspx>

に書いてあるように\_aligned\_mallocを使えばよいようです。

<https://msdn.microsoft.com/ja-jp/library/8z34s9cb.aspx>

コレいつは\_aligned\_freeとペアで使用します。

```
p=_aligned_malloc(1バイト数, アライメント);  
_aligned_free(p);
```

のように使用するらしい。ちなみに使用にはmemoryをインクルードする必要がある。

ただし、この場合は ScopedPtrも使えないし、なんかヤダって思う。malloc

まあ、XMMATRIXの正体はstructだし malloc&free系でいいかなあと。

手っ取り早い方法は…XMMATRIXはもう諦めてxnamath.hインクルードしちゃうのがいいと  
思います。

もう一つ考えられることは、プロジェクト行列は「保持しておく」のではなく  
カメラから参照されるたびにローカル計算して返してもいいかなとこれだったらアライメ  
ントの問題は発生しません。…そんなに参照頻度があるわけでもない(1フレ1回)ですし  
画角を動的に変更することもあることを考えると、特にメンバ変数で保持しておくメリット  
はそれ程高くなないのでしょう。

ローカル変数としての XMMATRIX は必ず 16 バイトアライメントされますので問題ないです  
…あとはちょっとだけ思うこと。

基本的には

```
XMMATRIX  
Camera::ViewFunc(){  
    ~ゴチャゴチャ処理~  
    return ローカル行列  
}
```

これはありだと思います。

ここでよく考えちゃう人は XMMATRIX のコストについてね？

const を付けた参照変数で一時オブジェクトを受け取ると、その一時オブジェクトの寿命が、  
その参照変数と同じ長さにまで延長されます。この行為は、「一時オブジェクトの参照による  
束縛」と呼ばれます。

例えば

```
#include<iostream>  
  
using namespace std;  
  
int func(){
```

```
int a;  
a=10;  
++a;  
return a;  
}
```

```
int main(){  
    int b = func();  
    cout << b << endl;  
    getchar();  
}
```

というコードがあったとして、このコードは問題ないわけです。ところが参照を使うと

```
#include<iostream>
```

```
using namespace std;
```

```
int& func(){  
    int a;  
    a=10;  
    ++a;  
    return a;  
}
```

```
int main(){  
    int& b = func();  
    cout << b << endl;  
    getchar();  
}
```

これがマズいのは分かりますよね？`a`はローカル変数なので、当然開放されたものに対して参照することになるためダメです。

VSだとコンパイル通っちゃうし、値も出ちゃうのが問題なんですが、まずバグコードです。

ああごめん。一部間違えてた。

やっぱり関数の方を

```
int& func()
```

みたいにするとダメっぽいです。要は束縛は

```
const int& b = func();
```

この部分にだけ働くっぽいです。嘘言ってしまった。すみません。XMMATRIXで検証したら戻り値はアカンことになってました。

## 23.4 Warning を黙らせる…

Warning を黙らせるには、warning で怒られている部分をちまちまつぶしていくことになります。warning メッセージを読めばやり方はわかると思いますが、たまに同じような Warning があります。

どういう Warning かというと、DirectX やら XNAMath が発生させている warning ですね。ライブラリ発生なのでどうしようもないのです。ただ、これが頻発すると取るべきエラーが取れなくなるので、あまり良くはないんですけど、特定の warning だけはコンパイルオプションで黙らせます。

今僕のところに出てるエラーの一つは

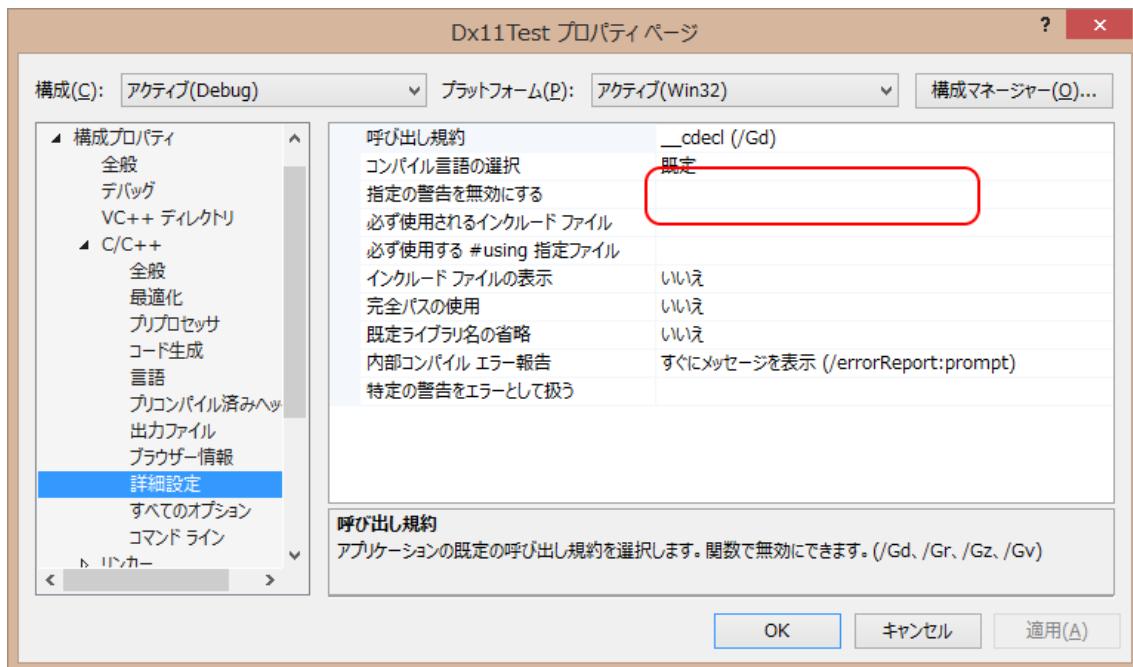
```
C:\Program Files(x86)\Microsoft DirectX SDK(June 2010)\Include\dxgitype.h(12): warning  
C4005: 'DXGI_STATUS_OCCLUDED': マクロが再定義されました。
```

```
C:\Program Files(x86)\Windows Kits\8.1\Include\shared\winerror.h(50092):  
'DXGI_STATUS_OCCLUDED' の前の定義を確認してください。』
```

というやつです。ひどい話ですが、winerror.h と dxgitype.h で定義されているマクロが同じ名前のようにです。

で、これまたひどいんですが、指示している値が同じなんです。意味ね～じゃん!!!

というわけで、黙らせます。どうせ自分でマクロ使うこともないと思いますので、この C4005 というエラーをつぶしましょ。



プロジェクトのプロパティ→C/C++→詳細設定→指定の警告を無効にする。

ここにさっきの Warning を入れてください。なお、C4005 の C は必要ないです。4005 という数値部分だけを入力してください。

あとは

C4996:'fopen':This function or variable may be unsafe. Consider using fopen\_s instead. To disable deprecation, use \_CRT\_SECURE\_NO\_WARNINGS. See online help for details.

これ。これは MS が勝手につけてくる Warning で、とにかく「俺が作った最強のセキュアな fopen\_s を使ってくれよ」という警告です。正直ウザいです。標準じゃないやつはいらねーんだよ!!!

場所によっては「標準」って書いてるけど、それは MS における標準だろうが!!!GCC にはそんなもんねーんだよ!!

とか思ってたら C11 から標準はなっているらしい…業界のトップが決めたルールがそのまま標準になるのはよくあることよな。でもまだまだ使えない処理系が多ish(GCC はまだダメだったような気がする)のと、そもそも\_s は「バッファオーバーラン防止」等のためのものだがこの対処の仕方が気に入らないんだよね…正直。

ちなみに fopen→fopen\_s にすると、エラーコードを返すようになるんだが、このエラーコード…なにを返すんだ?

まあ`strerror`にこのコードを入れれば良いんだけど…。まあここを黙らせるかどうかは皆さんにおまかせします。ちなみに C4996 です。

ただ、このセキュリティ警告は事ある毎に出てくるので正直ウザいので 4996 も黙らせておきます…僕はね。みんなはそれぞれの好きにしてください。

## 23.5 僕の Model.h を公開します

```
#pragma once
#include<vector>
#include<map>
#include<string>

#include"ScopedPtr.h"

class PMDMesh;
class VMDData;
struct ID3D11Buffer;
struct _XMFLOAT3;
typedef _XMFLOAT3 XMFLOAT3;

///@brief PMD等のモデルの操作
///Draw,座標などを制御するクラスです
class Model
{
private:
    ScopedPtr<XMFLOAT3> _pos;
    ScopedPtr<XMFLOAT3> _rotation;
    ScopedPtr<PMDMesh> _mesh;
    VMDData* _currentVMD;
    unsigned int _frameNo;
    ID3D11Buffer* _materialBuffer;
    ID3D11Buffer* _boneMatrixBuffer;
    std::map<std::string, VMDData*> _animations;
    void DeformBones(VMDData* vmddata, unsigned int frameNo);
```

```

public:
    Model(const char* filePath, ID3D11Buffer* materialBuffer, ID3D11Buffer*
boneMatrixBuffer);
    ~Model();

    ///@brief アニメーションの追加
    ///@param animName アニメーション名
    ///@param vmddata アニメーション
    void RegisterAnimation(const char* animName, VMDData* vmddata);

    ///@brief 場所を設定する
    ///@param x X座標
    ///@param y Y座標
    ///@param z Z座標
    void SetPosition(float x, float y, float z);

    ///@brief 回転を設定する
    ///@param x X軸回転
    ///@param y Y軸回転
    ///@param z Z軸回転
    void SetRotation(float x, float y, float z);

    ///@brief 今いる場所から移動する
    ///@param x X移動量
    ///@param y Y移動量
    ///@param z Z移動量
    void Move(float x, float y, float z);

    ///@brief 現状から回転する
    ///@param x X軸回転
    ///@param y Y軸回転
    ///@param z Z軸回転
    void Rotate(float x, float y, float z);

    ///モデル内時間を1フレーム進める

```

```

void Update();

///@brief モデルの通常描画
///@note マテリアルごとにインデックスを分け
///マテリアルを反映させつつ描画します
void Draw();

///@brief モデルの簡易描画
///@note 全インデックスを同一マテリアルで描画します
///@remarks マテリアル切り替えしない分ドローコールが減らせる
void SimpleDraw();
};


```

これはこんなもんでもいいかなーっと思うのですが、ちょっと Renderer には問題ありかなと思います。

## 23.6 僕の Renderer.h を公開します

```

#pragma once

#include<D3D11.h>

///レンダリングに関連する切り替えを制御するクラス
class Renderer
{
private:
    HRESULT _result;
    ID3D11RenderTargetView *_mainRTV;
    ID3D11RenderTargetView *_lightRTV;
    ID3D11DepthStencilView *_dsv;

    ID3D11Texture2D* _depthTexture;
    ID3D11ShaderResourceView* _shadowSRV;

    ID3D11VertexShader* _vsLvPmd;
    ID3D11VertexShader* _vsLvPrimitive;

```

```

ID3D11VertexShader* _vsPmd;
ID3D11VertexShader* _vsPrimitive;
ID3D11InputLayout* _pmdLayout;
ID3D11InputLayout* _primitiveLayout;

void SetViewport();

ID3D11RenderTargetView* CreateDisplayRTV();
ID3D11RenderTargetView* CreateLightRTV();
ID3D11DepthStencilView* CreateDepthStencilView();

HRESULT CreateLightViewVSPmd();
HRESULT CreateLightViewVSPrimitive();
HRESULT CreateLightViewPS();

HRESULT CreateCameraViewVSPmd();
HRESULT CreateCameraViewVSPrimitive();
HRESULT CreateCameraViewPS();

bool CheckShaderError(ID3D10Blob* errptr);

public:
    Renderer();
    ~Renderer();

    ///最新の結果を返す
    HRESULT GetResult(){ return _result; }

    ///@brief ライトビューのZ値が書き込まれている画像を返す
    ///@return シェーダリソースビューのポインタ
    ID3D11ShaderResourceView* GetSRVForShadow();

    ///@brief 通常レンダーターゲットのクリア
    void ClearRTV();

```

```
///@brief ライトビューレンダーターゲットのクリア
void ClearLightRTV();

///@brief 深度バッファのクリア
void ClearDSV();

///レンダーターゲットをライトビューに切り替え
void ChangeRTLightView();

///ライトビュー用のピクセルシェーダに切り替え
void ChangePSForLightView();

///ライトビューかつPMD用の頂点シェーダに切り替え
void ChangeVSForLightViewPMD();

///ライトビューかつ基本図形用の頂点シェーダに切り替え
void ChangeVSForLightViewPrimitive();

///レンダーターゲットをメインカメラに切り替え
void ChangeRTCameraView();

///メインカメラ用のピクセルシェーダに切り替え
void ChangePSForCameraView();

///メインカメラかつPMD用の頂点シェーダに切り替え
void ChangeVSForCameraViewPMD();

///メインカメラかつ基本図形用の頂点シェーダに切り替え
void ChangeVSForCameraViewPrimitive();

///プリミティブトポロジをPMD用に
void ChangePTForPMD();
```

```
//プリミティブポロジを基本図形用に
void ChangePTForPrimitive();
};
```

長いですね。

で、リファクタリング的に言うとここでの問題はこの「長い」事よりも private メンバが多すぎる事にあります。

そもそも private は private なのですから、公開したくないのです。

その割には多くの private メンバが可視状態にあり、情報のノイズにもなりますし何よりも隐蔽性の観点からちょっと問題あります。まあ学生作品レベルなら許されるので放って置いても良い。

ですが、ちょっとよくない。

そこで pimpl というイディオムを使用します。

そうなのです。ここからの話はちょっとアドバイスなので、真似をする必要はありません。

pimpl というのは p-impl で、つまり pointer-implement の略です。

ただ…よくインターネットで見かける pimpl も気に入らないんですね…。どういう事が」というとよくある実装は

```
class Foo {
public:
    // 公開部分
private:
    class Impl;
    std::unique_ptr<Impl> pimpl_;
};
```

(<http://torini.hateblo.jp/entry/2014/08/26/234812>)より引用

としておいて

```
// .cpp
class Foo::Impl {
public:
    Impl() : calculator_( Calculator::Add ) {
```

```

    }

~Impl(){}
// 加えて

void Add( int a ) {
    number_.emplace_back( a );
}

// 計算する

int Calculator() {
    return calculator_.Execute( number_ );
}

private:
    Calculator calculator_;
    std::vector<Number> number_;
};

Foo::Foo() : pimpl_( new Impl ) {

}

Foo::~Foo(){}

// Pimpl イディオムのデメリットである冗長な処理

void Foo::Add( int a ) {
    pimpl_->Add( a );
}

int Foo::Calculator() {
    return pimpl_->Calculator();
}

```

(<http://torini.hateblo.jp/entry/2014/08/26/234812>)より引用

なんですがね？僕はこれ…ヤなんですよ

だって、Calculator の中でまた、Calculator を呼び出すなんて…冗長でしょ？

だから僕の場合はこうします。

```

class Foo {
public:

```

```
// 公開部分
static Foo* Create();
};
```

```
// .cpp
class Foo::Impl : public Foo{
public:
    Impl() : calculator_( Calculator::Add ) {
    }
    ~Impl(){}
    // 加えて
    void Add( int a ) {
        number_.emplace_back( a );
    }
    // 計算する
    int Calculator() {
        return calculator_.Execute( number_ );
    }
private:
    Calculator calculator_;
    std::vector<Number> number_;
};
```

```
Foo::Create()
{
    return new Foo::Impl();
}
```

お分かりでしょうか? もはや pimpl イディオムではないですが、Impl そのものを返すため元の  
クラスに Calculator->Calculator()なんていうコードの仕方はしなくていいわけです。

## 23.7 シェーダエラー時にとっ捕まえやすくする

シェーダエラー時にきちんとキャッチできるようにしているでしょうか?キャッチと言ってもC++のtry~catchではないですよ?

エラーの原因がすぐ分かるようになりますか?今のところ、シェーダコンパイル時に失敗してもそのまま進んで、何が間違っていたのかわからない状況になっていると思います。これでは、時間の無駄が発生します。

そこでassertというのを使用しておきます。

C言語の頃からある機能なので知っていると思いますが、こいつは特定の場所でプログラムを停止させる役割を持っています。

### asset(評価式)

として、トラップをしかけておけば assert の評価式の結果が偽(false)である時に、プログラムが停止するようになります。

つまり、例えば

```
HRESULT result = D3DX11CompileFromFile("vs.hlsl", //シェーダファイル名
                                         nullptr, //DX10を併用するときに必要。使わないならヌルポでいい
                                         nullptr, //DX10を併用するときに必要。使わないならヌルポでいい
                                         "BaseVS", //関数名
                                         "vs_5_0", //プロファイル名(シェーダバージョンやね)
                                         0, //コンパイルオプション特に指定なし
                                         0, //実行時オプション特に指定なし
                                         nullptr, //スレッド使わないし指定なし
                                         &compiledShader,
                                         &shaderError,
                                         nullptr //即時復帰関数時に使うが今回は完了復帰なのでnullptrでいい
                                         );
```

だとして、

```
assert(result==S_OK);
```

と書けばそこで停止するため、シェーダコンパイルエラー時に引つかるので見つけやすくなります。SUCCEEDED使ってもいいけどね。

ところがこれでは不十分。これくらいのことなら誰でも思いつきます。

これでは「どういうエラーなのか」が分かりません。

というわけで、それをなんとか表示できるようにしてみましょう。

いや、それ程難しくもないんですが…ちょっと今のところ気に入らない実装になってるんですね。

シェーダエラーが発生した場合はコンパイル時の shaderError に入るようになっているんですね。

ということで、OutputDebugString(標準出力に出力する)

<https://msdn.microsoft.com/ja-jp/library/cc428973.aspx>

を用います。というわけでこういう関数を作りました。

```
bool  
Renderer::CheckShaderError(ID3D10Blob* errptr){  
    if (errptr == nullptr){  
        return false;  
    }  
    OutputDebugString(static_cast<LPCSTR>(errptr->GetBufferPointer()));  
    errptr->Release();  
    assert(errptr == nullptr);  
    return true;  
}
```

あまり良くないですねえ。ちなみにメッセージは警告に付け足したいんですがじゃあこうするのか…

assert(errptr == nullptr && errptr->GetBufferpointer());

これはダメですよね？わかりますよね？

というわけでちょっと悩みどころなんですよねえ…。

## 23.8 const と constexpr について

現代の C++ には `const` に加えて `constexpr` などという指定子が増えました。

(※申し訳ないが、この機能は VS2015 からのようだ)

<https://cpprefjp.github.io/lang/cpp11/constexpr.html>

まあ学生のうちには `const` だけ知つてれば十分だと思いますが、コードリーディング時に混乱してもあれなので解説しておきます。

簡潔に言うと

`const` は実行時に定数になるものであり

`constexpr` はコンパイル時に定数になるものである。

僕は `const` がコンパイル時定数かと思ってましたか…違ったんですねえ。

`const`: 実行時定数

`constexpr`: コンパイル時定数

何の違いがあるんだろう…とあるスレで両者の違いを解説している一文があつたので、そのまま引用します。

『`constexpr` はコンパイル時定数として評価されるもの、`const` は実行時に読み取り専用として扱われるもの（なので、`const` 宣言されても定数であるとは限らない）。そのスコープ

で書き込みされない)というだけ。ついでに言うと、`mutable` 指定されたメンバに対するアクセスは `const` 性の担保の対象外になります)。

うーん。

分かりづらい。

ともかく、`#define` の代わりに使われるような『定数』は今後は `constexpr` を使うのが主流になりそうだね。

ちなみに、上で“`mutable`”っていう耳慣れない言葉が出てきたんだが、こいつは何なのかというと `const` を殺すためのものだ。`const` 絶対殺すマンなのだ。

どういうことがというと、例えば

```
class A{
    private:
        int a;
        B b;
    public:
        A(){}
        void Func()const{
            a=10;//コンパイルエラー！！const 関数は自分のメンバに影響を与えることはでき
            //ないんやで？
            b.Function();//エラー！！この関数がconst 関数でなければ呼び出せないんや
        }
}
```

```
    }  
};
```

これは分かるかな?

このメンバ変数の a とか b は Func 関数においては const 修飾子に守られてるんやで。逆に言うと変更の事由がなくなってるんやで。Mother3 のポーキーみたいな状態なんやで。

ところがこの変数に mutable 修飾子をつけると…

```
class A{  
  
private:  
  
    mutable int a;  
  
    mutable B b;  
  
public:  
  
    A(){  
  
        void Func()const{  
  
            a=10;//mutable がついてるからええんやで?  
  
            b.Function();//ええんやで?  
  
        }  
  
    };
```

という風に const の意味を失うというわけや。使い方に注意なんや。ところでこの const を constexpr にすると mutable も死ぬのはず。

なぜ検証できていないのかというと、ここまで書いて気が付いたんだが `constexpr` は VS2015 からの実装で VS2013 にはついてないんだった…(; 'Д `)トホホ

<https://msdn.microsoft.com/ja-jp/library/hh567368.aspx>

`constexpr` って C++11 の機能じゃねーか!!なんで VS2013 でダメなんだよ!!!ふざけんな!!!

まあ…こんなアホな目にも合うが、最新情報にはアンテナを立てとけってことだ。

## 24 Effekseer 組み込み

Effekseer とは、エフェクトを 3D 空間に表示できるようにするためのものです。

<https://effekseer.github.io/jp/>

オープンソース・ソフトウェアです。

最新版(現在 1.22)の Effekseer 本体と、Effekseer1.22 for Runtime を落としてきてください。  
一部の人は前期に DxLib 用のを使いましたよね? それじゃなくて(DirectX or OpenGL)って書いて  
てるのを落としてきてください。

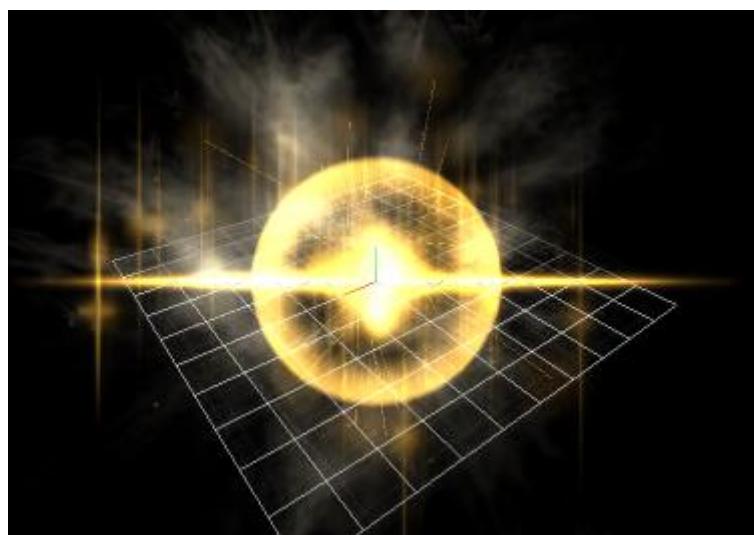
### 24.1 とにかくダウンロードして遊ぼう

ひとまず本体を落として展開しましょう。Tool ってフォルダの中に Effekseer.exe 本体があります。

起動してサンプルのエフェクトを見てみます。せつかくだからかっこいいのにしましょう。例  
えばこういうのです。

Sample¥01\_Pierre¥02¥Benediction.efkproj

なら



こういう感じです。

今回はこれを画面上に表示させてみましょう。

今度はランタイムの方を解凍してください。

## 24.2 ひとまずサンプルを動かそう

EffekseerRuntime122\RuntimeSample

の中にDirectX11.slnがありますので起動しましょう。

ちょっと実行してみてください。

そう、動きませんね。

ちょっと自分で解決しようとしてみてください。

ヒントは…

一つ上のフォルダにCompiledってフォルダがあって、その中にVS2013って書いてるので…まあそこに全ての答えがあります。

しばらくがんばってください。

で、しばらく頑張ると

エラー 1 error LNK1104: ファイル 'X3DAudio.lib' を開くことができません。

C:\Users\r\_kawano\Downloads\EffekseerRuntime122\EffekseerRuntime122\Runtime  
Sample\DirectX11\LINK DirectX11

などというエラーのみが残ります。

で、オーディオ関連のリンクエラーなんですけど今回はこれを放置します。オーディオ関連は真面目にやるとちょっと大変です。

ちなみにオーディオ関連はCRIADX2LEを使用するのが現状の問題解決としては適切かなーって思ってはいます。XAudio2をそのまま使うのはチョイとばかりハードルが高いので…。

で、今それをやっちゃうとかなり遠回りなので、とにかくサウンドをここでは使わないでの、サンプルの

```
#include <EffekseerSoundXAudio2.h>
```

```
#pragma comment(lib, "VS2013/Debug/EffekseerSoundXAudio2.lib" )
```

```
#include <XAudio2.h>
```

をまるごと消しちゃって、もちろんエラーが出まくるとは思いますが、片つ端からエラーの行を削除していってください。

そこまで行くと



こういう画面が出来ます。

ちなみに、このサンプルは「コマンドプロンプト」も出ているんですが気が付きました？

そう、こいつは WinMain ではなく、main 関数から動いているんですよね。

え？ だってウィンドウ作るのにインスタンスハンドルが必要じゃないの？ main じゃそれを取ってこれないんじゃないの？

そこで使用するのが GetModuleHandle です。

<https://msdn.microsoft.com/ja-jp/library/cc429129.aspx>

第一引数がモジュール名なんですが、ここを nullptr にすることによって、「呼び出し側プロセスの作成に使われたファイルのハンドルが返ります」

意味わからぬかも知れませんが、この呼び出し側プロセスのハンドルってのが、このアプリ自身を指します。

つまり GetModuleHandle(nullptr)で現プロセスのインスタンスハンドルを取得することができます。

ちょっと話がそれましたが、何も表示されません。

310行目くらいに

// エフェクトの読み込み

```
g_effect = Effekseer::Effect::Create( g_manager, (const EFK_CHAR*)L"test.efk" );
```

と書いてあると思いますが、この test.efk がないため怒られています。なお、Effekseer のリファレンスは Runtime のフォルダの中の Help 中の index.htm を開いて、中の「リファレンス」を見てください。

その中の Effect::Create を見ましょう。こういう仕様になっています。

```
static Effect* Effekseer::Effect::Create ( Manager* manager,
                                            const EFK_CHAR* path,
                                            float magnification = 1.0f,
                                            const EFK_CHAR* materialPath = NULL
)
```

エフェクトを生成する。

#### 引数

**manager** [in] 管理クラス  
**path** [in] 読み込み元のパス  
**magnification** [in] 読み込み時の拡大率  
**materialPath** [in] 素材ロード時の基準パス

#### 戻り値

エフェクト。失敗した場合はNULLを返す。

…とりあえず「戻り値が nullptr かどうかで assert を入れてみましょう。

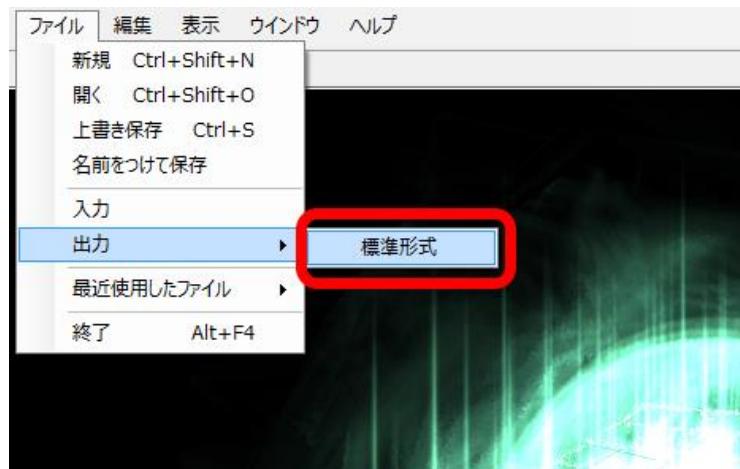
```
assert(g_effect);
```

を Create の下の行に書いてください。実行するとクラッシュすると思います。

とにかくエフェクトファイルがないんです。

エフェクトファイルを作りましょう。今度はもういちど Effekseer 本体を起動して、自分の好きなエフェクトを読み込みます。

そして



出力→標準形式

で出力するとOO.efkというファイルが出力されます。あ、倍率は1でいいです。

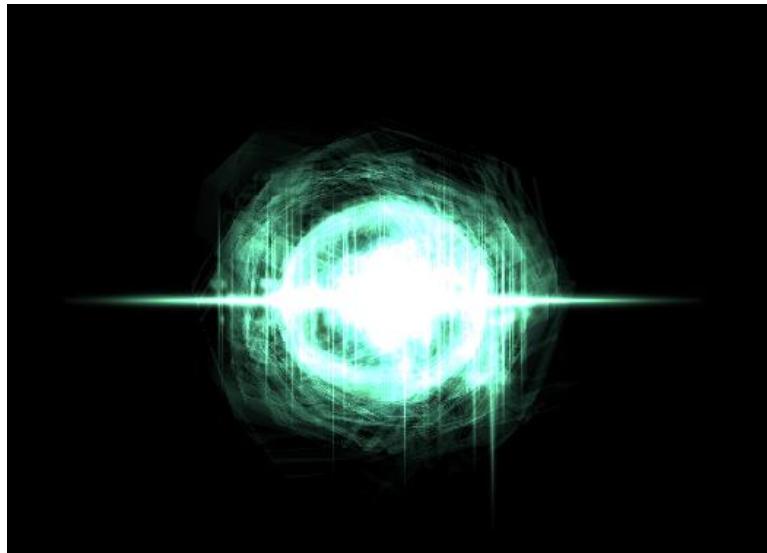
特に気をつけずに実行するとこうなるでしょう…



で、こうなってしまう場合はどういう事になっているのかというとエフェクトに使用している「画像」の読み込みができないんですね。

というわけで、エフェクトフォルダの Model や Texture も一緒にコピーしてください。

そうすれば



やったー＼(^o^)／

さて、ここまでができれば DirectX11 に組み込むことができるよう気がわかつただろう？

### 24.3 「俺の」プロジェクトに組み込んでみよう

さて…ここからが本番だ。ちょっと色々と面倒だけど頑張ろう。

まず、さっきもやったことですがランタイムを使うためには「パス」を通さないといけないんですよね。

#### 24.3.1 インクルードパスとライブラリパスを解決しよう

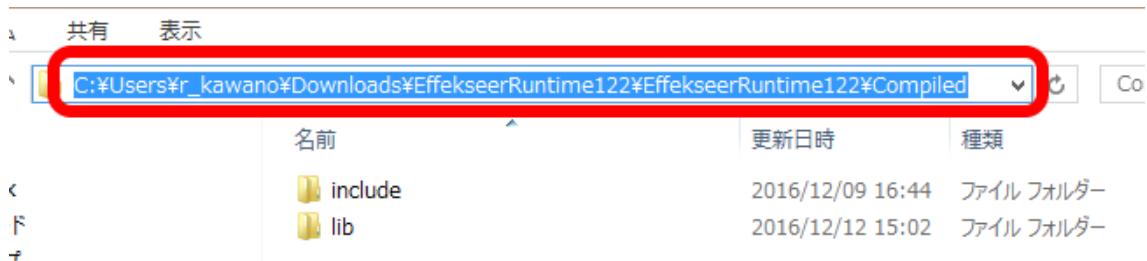
最初に言っておくと、この手の「外部ライブラリ」はサイズがでかいので学校の PC や家の PC でそれぞれの「インストールフォルダ」を設定しておいて、そこへのパスを通します。ただし Effekseer はインストールではなく単なる解凍なので、解凍フォルダを「インストールフォルダ」と見立てて作業します。

手順としては

- 適当なフォルダを「インストールパス」とする(今回はダウンロードフォルダ)
- そのフォルダに対する「環境変数」を作る(名前は EFK\_DIR とでもする)
- 環境変数を利用してプロジェクトからパスを通す

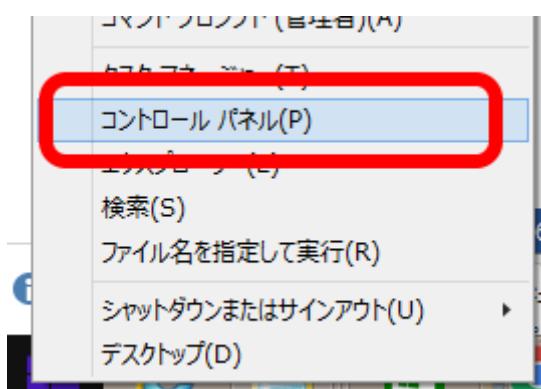
こんな感じです。

既にダウンロードして解凍しているでしょうからそこのパスを確認してください。



エクスプローラで件のフォルダに移動して、そのフォルダのパスをコピーしておきましょう。  
次に環境変数設定ですね。

Windowsマークを右クリクして「コントロールパネル」を開いて、

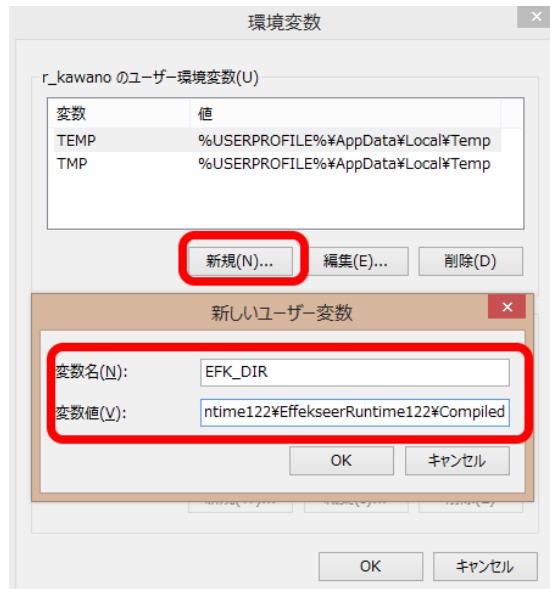


その中のユーザー アカウントをクリックします。

A screenshot of the 'User Accounts' section of the Control Panel. On the left, there are links: 'Control Panel Home', 'User Accounts and Family Safety', 'User Accounts', 'Change User Account Settings', 'User Accounts Settings', and 'Environment Variables' (highlighted with a red box). On the right, there are links: 'User Accounts Home', 'Change User Account Settings', 'User Accounts Settings', and 'User Accounts'. A large blue decorative element is on the right side.

こんな感じの画面が出てくるので「環境変数の変更」をクリックします。

そうすると



こういう画面が出てきますので、新規って押して、新規環境変数。今回は名前が EFK\_DIR で値がさっきコピーしたファイルパスです。

よろしいですか？ 設定したらオッケーを押して閉じてください。

では確認のためにコマンドプロンプトを立ち上げて

```
echo %EFK_DIR%
```

と書いてエンターを押してください。さっき設定したパスが表示されれば正解です。

行けましたか？

じゃあこの環境変数を使って、エフェクシアのランタイムへのパスを通しましょう。

先程の EFK\_DIR をプロジェクト設定で使用する場合は \$(EFK\_DIR) という具合に使用します。

今回のプロジェクト設定であれば、

インクルードパスは

`$(EFK_DIR)\Include`

となり、ライブラリパスは

`$(EFK_DIR)\Lib`

となるでしょう。

ここまで設定できたら、組み込みを実験してみましょう。

ひとまず自分で Effect クラスを作って Effect.h および Effect.cpp を作って Effect.cpp の先頭で `#include<Effekseer.h>` してみてください。

エラーが出ませんか？それなら成功です。

#### 24.3.2 ひとまずコンパイルおよびビルドを通す

今作っている Effect クラス(CPP)内で Effekseer 組み込みのための下準備ができているのかどうかを確認するために、以下のコードを書いてみてください。いつもはまんま書かせないんですが、ここは「動作テスト」ということで手早く行きましょう。

```
#include <Effekseer.h>
#include <EffekseerRendererDX11.h>

#if _DEBUG
#pragma comment(lib, "VS2013/Debug/Effekseer.lib")
#pragma comment(lib, "VS2013/Debug/EffekseerRendererDX11.lib")

#else
#pragma comment(lib, "VS2013/Release/Effekseer.lib")
#pragma comment(lib, "VS2013/Release/EffekseerRendererDX11.lib")
#pragma comment(lib, "VS2013/Release/EffekseerSoundXAudio2.lib")
#endif

#include "Effect.h"

::Effekseer::Manager* manager = nullptr;
::EffekseerRenderer::Renderer* renderer = nullptr;
::Effekseer::Effect* effect = nullptr;
```

```
Effect::Effect()
{
    Effekseer::Effect::Create(manager, (const EFK_CHAR*)"test.efk"));
}
```

とりあえずこんな感じで…書けたらビルドしてください。

インクルードやリンクがうまく行ってない時にコンパイルエラーもしくはリンクエラーが発生します。

大丈夫でしょうか?エラー起きてませんか?

それではひとまずは組み込みそのものは完了です。

## 24.4 さあ本格的インテグレーションだ

自分の

EffekseerRuntime122/EffekseerRuntime122/Help/Contents/execution.html

のヘルプを見てください。DirectX9 のを参考にすると良いでしょう。

手順としては

1. Effekseer 初期化処理
2. 必要な設定
3. 再生したいエフェクトのロード
4. エフェクトの再生処理
5. エフェクトの更新処理
6. エフェクトの描画処理
7. Effekseer 後処理

だいたいこんな感じで、5~6は毎フレームループで行います。

4の再生処理は何かしらのイベントで起こすようにしましょう。

とりあえず現在のエフェクシアサンプルと同様に起動時に1~3を行い、5~6を毎フレーム行います。

で、サンプルと違う部分としては4番を、何かしらのキーイベントで発生させましょう。

#### 24.4.1 初期化処理

ひとまずは初期化を行いましょう。

初期化時にはまずエフェクシアのマネージャとレンダラを最初に作る必要があります。簡単なんんですけどまずは

```
_manager=Effekseer::Manager::Create( 最大インスタンス数 );
```

というものでマネージャの初期化を行います。

一応サンプルでは 2000 となっています。これが適切かどうかを判断する方法が今のところないので 2000 を入れておきましょう。ただし 2000 と直に書くのではなく

```
const int effect_instance_max=2000;
```

とでもしておいて

```
_manager=Effekseer::Manager::Create(effect_instance_max);
```

次にレンダラーを作りましょう。サンプルではまた 2000 になってますが

```
_renderer=EffekseerRendererDX11::Renderer::Create( dev.Device(), dev.Context(), 2000 );
```

この 2000 って数値、関数定義を見ると「インスタンス数」ではなく「最大描画スプライト数」って書かれていますが、ヘルプを見るとどちらも「最大描画スプライト数」らしいです。

…同じ数値を使って良いようです。単なるヘルプのミスなのかどうかはわからないんですけどとりあえず同じ定数入れておきましょう。

```
renderer=EffekseerRendererDX11::Renderer::Create( dev.Device(), dev.Context(), effect_instance_max );
```

こんな感じでお願いします。

#### 24.4.2 基本設定

ヘルプに「独自に拡張しない限り定型文です」と書かれてるので、無用な追求はしません。サンプルをそんまま使っていいです。

```
// 描画用インスタンスから描画機能を設定
```

```
_manager->SetSpriteRenderer(_renderer->CreateSpriteRenderer());  
_manager->SetRibbonRenderer(_renderer->CreateRibbonRenderer());  
_manager->SetRingRenderer(_renderer->CreateRingRenderer());  
_manager->SetTrackRenderer(_renderer->CreateTrackRenderer());  
_manager->SetModelRenderer(_renderer->CreateModelRenderer());
```

```
// 描画用インスタンスからテクスチャの読み込み機能を設定  
// 独自拡張可能、現在はファイルから読み込んでいる。  
_manager->SetTextureLoader(_renderer->CreateTextureLoader());  
_manager->SetModelLoader(_renderer->CreateModelLoader());
```

#### 24.4.3 カメラとかの設定

Effekseer はカメラの設定が必要です。カメラの設定を行います。

DirectX と同様にカメラ行列とプロジェクション行列の設定を行わなければなりません。現行の DirectX と合わせる方法は2つあって

サンプルのように Effekseer が持っている関数を使って同じように設定するのが一つの方法…。

ひとまずサンプルをそのまま移行してみましょうか。

```
// 視点位置を確定
```

```
Effekseer::Vector3D eye = Effekseer::Vector3D(0.0f, 15.0f, -25.0f);
```

```
// こつからは左手系ですよー
```

```
_manager->SetCoordinateSystem(Effekseer::CoordinateSystem::LH);
```

```
// 投影行列を設定
```

```
_renderer->SetProjectionMatrix(  
    Effekseer::Matrix44().PerspectiveFovLH(XM_PIDIV2,  
    static_cast<float>(_windowWidth) / static_cast<float>(_ windowHeight), 0.1f, 100.0f));
```

```
// カメラ行列を設定
```

```
_renderer->SetCameraMatrix(  
    Effekseer::Matrix44().LookAtLH(eye, Effekseer::Vector3D(0.0f, 0.0f, 0.0f),  
    Effekseer::Vector3D(0.0f, 1.0f, 0.0f)));
```

```
// エフェクトの読み込み
```

```
_effect = Effekseer::Effect::Create(_manager, (const EFK_CHAR*)(L"test.efk"));
```

ところどころ「そのまま」ではないけどね。

気をつけるべきところは、ファイル名指定のところで、しが必要なことと、

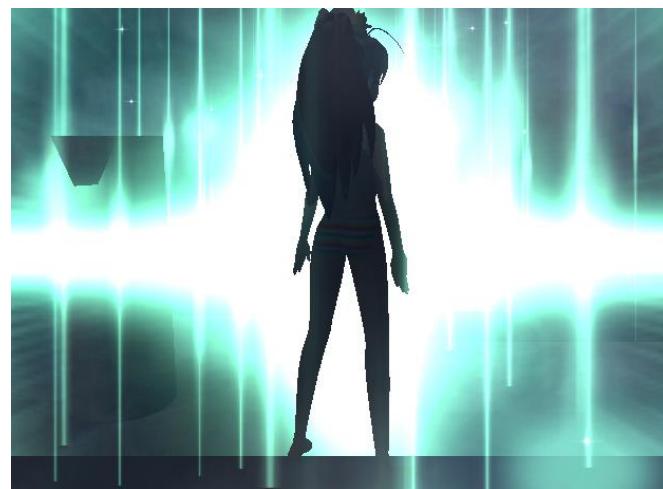


右手系、左手系を調整するところです。

「しはなんかつちゅーと、前期にも話した文字列リテラルを Unicode に変換するものです。変数の変換はそのうち話す予定です。今はしつけといしてください。

右手系、左手系のデフォルトが Effekseer は OpenGL に合わせて右手系になっているので、左手系に強制的にやつちやいます。

うまくいけば



こんな風にかっこよくなります。

ただ、この状況、やってみればわかりますが



このように地面に線が入ったりします。これはエフェクトの仕組みを考えると仕方ないことで、後々説明する「ビルボード」という仕組みでひとつひとつの絵が作られています。

で、それを多重に重ねることによって煙っぽいことしてるんですが、所詮ボリュームレンダリングなんていう強力なことをやっているのではなく、半透明セロハンを色々な所に重ねているだけなのね？

というわけで↑のような状況になってしまいます。所詮そういう仕組なので、半透明透明な物体ばかりなら目立たないかもしれないですが、このように不透明のものが地面に敷かれているとかなり目立ちます。

これへの対処は結構難しいので、今は放っておきます。

さらに…カメラとかの設定は既にやってんだから、どうにかして行列渡せないかな…と思うたりするんじゃないですか？

その場合はそれぞれのエフェクシア側の行列に対してこちらから直接に行列値を指定することができます。

…そもそも Effekseer::Matrix44().PerspectiveFovLH は向こう側の関数を使用してこっちのカメラと同じことやってるだけなので、行列を渡したほうが早い。ただし、行列の型が違うためコピーをしてやらねばならない。

そこで

```
void  
Effect::SetCamera(const XMATRIX& view, const XMATRIX& proj){  
    Effekseer::Matrix44 effproj, effview;  
    for (int i = 0; i<4; ++i){  
        for (int j = 0; j<4; ++j){  
            effproj.Values(j)(i) = proj.m(j)(i);  
            effview.Values(j)(i) = view.m(j)(i);  
        }  
    }  
}
```

という風に、一つ一つの値をそれぞれコピーしても良い。どちらのアプローチでもいいが、カメラの方向は、PMD 側と同じにしといていただきたい。

## 24.5 特定の場所にエフェクトを出そう

せつかくだから、地面をマウスでクリックしたら、爆発するようにしてみましょう。

これに使う理論は「レイと平面の交点」です。

あ、難しそうだと思ったでしょ？ 難しいんですよ。数学なんですよ。影の落とし方を覚えていたら何となくわかるとは思います。

レイはどうやって定義するかというと、「視点」から「スクリーン」の特定の点に向かうベクトルです。定義法は後で言います。

平面に関しては既に Plane クラスに内包しているんでそんなに難しくもないです。

視線ベクトルを  $V$  として、平面の法線ベクトルを  $N$  とし、平面の原点からのオフセットを  $d$  とすると

直線

$$V(V_x, V_y, V_z) \times t$$

と、平面

$$N_a x + N_b y + N_c z - d = 0$$

これが交われば良いわけだ。何となく思い出してきたかな？この  $t$  を求めれば良いわけだよね？

現在の点が視点  $E$  であるとすると交点  $P$  は

$$P = E + V(V_x, V_y, V_z) \times t$$

床を狙つていれば  $t$  が増えるほど床側に近づきますよね？で、 $t$  が 1 増えるたびにどれくらい床に近づくかというと、 $V$  と  $N$  の内積  $V \cdot N$  であるわけだ。

あとは  $P$  から平面に対しての垂線の長さがわかればいい。どうやって求める？

実はアホみたいに簡単で、 $P$  そのものが原点からの位置ベクトルになっているので、そいつと法線ベクトルの内積がそのまま法線方向の距離になる…つまり

$$t = \frac{P \cdot N}{-V \cdot N}$$

となる。 $d$  のオフセットを加えるならば

$$t = \frac{P \cdot N + d}{-V \cdot N}$$

であろう。

$t$  が求まれば、交点が分かるので、あとは必要な要素をさっさと計算すればいい。

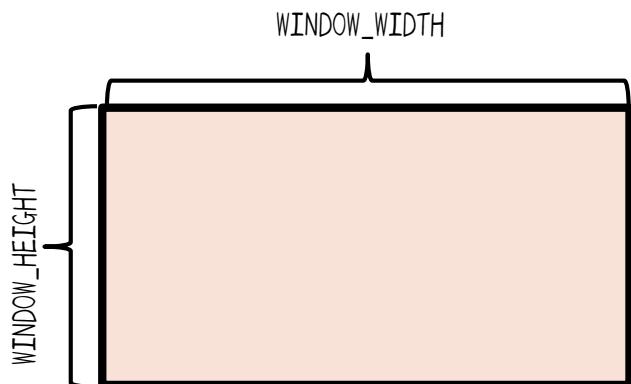
必要な要素とは「視点」、「法線ベクトル」、「視線ベクトル」、「オフセット」だが…視点は既にカメラ位置として定義しているだろう。視線ベクトルは「スクリーンの特定の点から視点を引けばいい」。法線は平面が持っているはず。オフセットも平面が持っている。

### 24.5.1 カメラクラスの改造

とにかくカメラベクトルから視点を返せるようにしよう。Position()という名前の関数でも作つとけばいいでしょう。

一番の問題は「スクリーンの特定の点から視点を引けばいい」とは言うものの、スクリーンの特定の点とはどうやって考えれば良いんでしょうか…。

画面は以下のようになっていますね？



いつもの逆に考えるのです。

<http://marina.sys.wakayama-u.ac.jp/~tokoi/?date=20090907>

最初に言えることはこのスクリーンそのものは3D空間におけるnearにあたり、縦横の範囲が-1~1になるようになります。

実際にはここから遠くに行くにつれて、これが広がっていくわけですが…

ひとまず“ウインドウにおける座標をスクリーンの座標…つまり逆ビューポート変換をかけましょ。

[https://msdn.microsoft.com/ja-jp/library/ee422543\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee422543(v=vs.85).aspx)

の逆ですやん。なに、簡単ですやん。

指定されたX座標を幅で割るやん？2倍するやん？1引くやん？Y方向にも同じことするやん？これがひとまずビューポートを無効にした状態やねん。

そこまでばいい。ええんよ。

じゃあ、視点からその変換スクリーンへのベクトル作れば良いのかって思うけど、そう単純でない。

何でかというとプロジェクション行列により空間が歪んでしまつとるから正確な3D座標にはならへんのや。

で、プロジェクション行列の逆行列を考えなければならぬんだけど、色々シンプルにならんもんか考えたんだけど、ちょっと思い一つがなかつた…。

というわけで素直に逆行列を使っちゃいましょう。つまり XMMatrixInverse の出番なわけです。(本当は逆行列は複雑だから使いたくないんだけど仕方ない)

ちなみにさつき言葉で言ってた逆ビューポート行列は

$$V_p^{-1} = \begin{pmatrix} \frac{2}{\text{WIDTH}} & 0 & 0 & 0 \\ 0 & -\frac{2}{\text{HEIGHT}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 1 & 0 & 1 \end{pmatrix}$$

こんなやつです。これは多分自分で作ったほうが早い(HUD行列とほぼ同じだしね)。疑わしい人はビューポート行列と乗算してみてください。ちなみにビューポート行列は

$$V_p = \begin{pmatrix} \frac{\text{WIDTH}}{2} & 0 & 0 & 0 \\ 0 & -\frac{\text{HEIGHT}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{\text{WIDTH}}{2} & \frac{\text{HEIGHT}}{2} & 0 & 1 \end{pmatrix}$$

こんなやつです。

最後にビュー行列の逆行列を考えなければならぬんだけど、こいつは簡単というかですね…思い出してほしいんですけど、ビューそのものはカメラの向きと起点からできるわけです。

つまりカメラの回転と平行移動のみでできています。

そして逆行列は使いたくない…なんかレリ方法はないでしょうか。

思い出してください

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

こんなやつ。

カメラは拡大縮小しません。つまりビュー行列の左上  $3 \times 3$  は純粹に「回転のみ」なのです。そして回転のみ行列の特徴としては

「転置すると逆行列の性質を示す」

ということがあります。これは面白い性質です。ちなみに「転置」というのは行列の行と列を入れ替えるというもので、コストはほぼゼロに近いものです。

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

に転置行列の

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

をかけてみます。

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \times \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} \cos^2\theta & \sin^2\theta \\ \sin^2\theta & \cos^2\theta \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

と言った具合に、単位行列になりますので、純粋な回転行列においては転置行列＝逆行列といえます。

とりあえず言っておくとビュー行列の逆行列を作る手順は…

1. ビュー行列の平行移動成分(4行目)を 0001 にする
2. 転置する
3. 視点座標をマイナスしたものを 4 行目に割り当てる

という感じです。

というわけで3つの行列の逆行列の作り方を書きました。スクリーン座標を3D空間へ変形するためには3D空間をスクリーン座標へ変換する逆を行えば良いわけです。

3D→スクリーンは

ビュー行列×プロジェクション行列×ビューポート行列

です。

スクリーン→3Dは

逆ビューポート行列×逆プロジェクション行列×逆ビュー行列

もしかしたらビュー行列とビューポート行列の区別がついてない人がいるかもしれないの  
で言っておきます。

- ビュー行列:カメラの向きと位置から計算される行列…カメラそのものを表す
- ビューポート行列:-1~1の範囲を0~WIDTH&0~HEIGHTにする行列

難しいけど区別してね。

で、ちょっとむずかしいんで、関数書いてみるけど、今のところうまくいってないんでどつか問  
違っていると思います。

```
void
Camera::CalculateViewportInverseMatrix(){

    *_viewportMatInv = XMMatrixIdentity();
    _viewportMatInv->_11 = 2.0f / _viewport->Width;
```

```

        _viewportMatInv->_22 = -2.0f / _viewport->Height;
        _viewportMatInv->_41 = -1;
        _viewportMatInv->_42 = 1;

    }

void
Camera::CalculateProjectionInverseMatrix(){
    XMVECTOR det;
    *_projectionMatInv = XMMatrixInverse(&det, *_proj);
}

XMMATRIX
CalculateViewInverseMatrix(XMMATRIX* camera,XMFLOAT3& eye){
    XMMATRIX m = *camera;

    m._41 = 0;
    m._42 = 0;
    m._43 = 0;
    m=XMMatrixTranspose(m);
    m._41 = eye.x;
    m._42 = eye.y;
    m._43 = eye.z;
    return m;
}

```

で、変換関数とか作つといて

```

ret =
ret*(*_viewportMatInv)*(*_projectionMatInv)*CalculateViewInverseMatrix(_view,*_pos);
掛け算します。

```

で、この結果を使って、スクリーン上の座標から画面上の座標を出してみたんですけど、今のところアリエナイ値が出てるので、修正したらまた教えます。

…なんでやろ(•ω•)

ちなみに所定の場所に表示する方法は

```
_manager->SetLocation(_currentPlayingHandle, pos);
```

のように、表示したい座標を指定します。面白いのは

```
effect->SetLocation(pos);
```

じゃあないんですね。エフェクト自体が動くって言うよりマネージャさんに「配置してもらう」ってイメージなんでしょうか。

もしくは発生場所自体を変更したい場合はエフェクトをエミットする際に

```
_currentPlayingHandle = _manager->Play(_effect, pos.x, pos.y, pos.z);
```

のように指定します。

で、クリック箇所を爆発させるという試みは…最終的な結論を言うと



うまくいきました。が、この結論に至るまでに**通算6時間**位かかりました。寝てません。



うまくいかない…何でだろう(•ω•)。からはじまり

理論的には正しいはずなんだけ  
どなあ…わからん



いじっていくうちにドウンドゥンとカオスになっていくコード…なにをやってもうまくいかない…

絶望

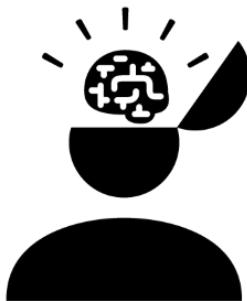


もうこれダメなんじゃね?

流石に諦めかけましたよ、もう。

何やってもうまくいかないんですもん…。

そこに天啓が!!



見つけてしまいましたよ…見つけてしました

XMFLOAT3 Unproject

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.transformation.xmvector3unproject\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.transformation.xmvector3unproject(v=vs.85).aspx)

キタ―――(°▽°)―――!!

読んでみましょう。

『スクリーン空間からオブジェクト空間に 3D ベクトルを射影します。』

これよ!!これやろ!!!

XMFLOAT3 XMVector3Unproject(

    XMFLOAT3 V, //元座標

    FLOAT ViewportX, //ビューポート左

    FLOAT ViewportY, //ビューポート上

    FLOAT ViewportWidth, //ビューポート幅

    FLOAT ViewportHeight, //ビューポート高さ

    FLOAT ViewportMinZ, //ビューポート Z 値(最小)

    FLOAT ViewportMaxZ, //ビューポート Z 値(最大)

    XMMATRIX Projection, //プロジェクション行列

    XMMATRIX View, //ビュー行列

```
XMMATRIX World//ワールド行列  
)
```

やたらとビューポートの値をとってきているので、システムのビューポートを取得しちゃいましょう。

[https://msdn.microsoft.com/ja-jp/library/ee419738\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/ee419738(v=vs.85).aspx)

RSGetViewports を使用します。

ワールド行列は必要ないので XMMatrixIdentity()でも入れておきましょう。

ひとまず RSGetViewports でビューポートを取得して、Unproject でスクリーン上の座標から 3D 座標へ変換します。

```
UINT num = 1;  
dev.Context()->RSGetViewports(&num, _viewport.Get());  
  
retvec = XMVector3Unproject(vnear, 0, 0, _viewport->Width, _viewport->Height, _viewport->MinDepth, _viewport->MaxDepth, *_proj, *_view, XMMatrixIdentity());
```

さて、これが“near”スクリーンの 3D 座標である。

次に “far” スクリーンの 3D 座標を同様にして求めます。

何故、それを求めるのかというと、その点における視線ベクトルを計算するためです。視線ベクトルは、near の点と far の点を結ぶベクトルが視線ベクトルとなります。

カメラ座標を使っても良いんですが、それだとカメラが 0 の座標にいなければおかしなことになるため、far 点 - near 点 からベクトルを生成します。

このベクトルをカメラ位置にくっつけます。

あとは平面に届くまでの距離を計算すれば良いわけです。

```
XMFLOAT3 vec = _camera->CalculateCursorPosition(p.x,p.y);
XMVECTOR ray = XMVector3Load(&vec);

//レイを正規化
ray=XMVector3Normalize(ray);
XMVECTOR camerapos=XMVector3Load(&_camera->CameraPosition());
//床までの距離を算出
float d;
XMStoreFloat(&d, XMVector3Dot(camerapos, plane.ABCD()));

float t;
XMStoreFloat(&t, XMVector3Dot(-ray, plane.ABCD()));

ray=XMVectorScale(ray, d / t);

XMFLOAT3 pos;
XMStoreFloat3(&pos,camerapos + ray);

effect.Emit(pos);
```

という感じで計算すれば、所定の場所に爆発を発生させることができます。

話だけ聞けばここまで30分くらいだと思うんですけど、最初にも書いたとおり6時間かかるってるんで…。

その重みは感じてもらうと嬉しいなあ…

## 25 アニメーション切り替え

今回はアニメーションの切り替えについて話していきたいと思います。

とはいっても、単純な切り替えのための準備は実は整っていて、

```
Model::RegisterAnimation(const char* animName, VMDData* vmddata){  
    _animations(animName) = vmddata;  
    _currentVMD = vmddata;  
}
```

っていうのを指定していれば、後からは名前を指定すれば vmddata を取得できるので

```
void  
Model::SetAnimation(const char* animName){  
    _currentVMD = _animations(animName);  
    _frameNo = 0;  
}
```

これで切り替わるようになります。

簡単でしょ？

ここまでではな…。

## 26 アンチエイリアシング

さて、ここまで作ってきて、思う事はないだろうか？



なんかエッジがジャギジャギしてるなあ。

と思わないだろうか？

そう。現在のラスタライズでは「スキャンライン法」というやり方でやってるんだが、そのやり方の場合、ピクセルごとに色を決定するため「塗るか塗らないか」になる。

その結果、オブジェクトがあると判断されるピクセルは塗りつぶされるが、そうでないところは塗りつぶされない。

これで出てきたガタガタを「ジャギー」と言ったり、広義の意味で「アーティファクト」と呼んだりする。一般的には「ジャギー」が使われる。

さて、これを解消するための方策として「アンチエイリアシング」ということを行ったり行わなかつたりするのだが…。

最初に言っておこう。フルスクリーンのアンチエイリアシングは「重い」。これだけは知つておいてほしい。

なぜ重いのかはあとで解説するとして、とりあえず DirectX の機能でアンチエリをかけるようにしてみましょう。

と思つてちょっとスワップチェインとか、いろいろいじつたら画面上に何も表示されなくなつたため、もう少し研究してまた違う機会にしたいと思います。単純に表示してた時はすぐにアンチエリがかかるたんですが、多分、レンダーターゲット切り替えとか入れてるから、その辺でうまくいかなくなつてるんでしょう。

一応ソースコードを記載しておきますが、現状だとうまくいかなくなることを覚えておいてください。

まず、デバイスとスワップチェインの作り方が変わります。

```
HRESULT hr = D3D11CreateDevice(nullptr,
    D3D_DRIVER_TYPE_HARDWARE,
    nullptr,
    0,
    &featureLevels,
    1,
    D3D11_SDK_VERSION,
    &_device,
    pFeatureLevels,
    &_context);

//インターフェースをクエリ
IDXGIDevice1* dxgi = nullptr;
hr = _device->QueryInterface(__uuidof(IDXGIDevice1), (void**)&dxgi);
```

```

if (FAILED(hr))
{
    return 0;
}

// Dxgiからアダプタを取得
IDXGIAdapter* pAdapter = nullptr;
hr = dxgi->GetAdapter(&pAdapter);
if (FAILED(hr))
{
    return 0;
}

// アダプタからファクトリを取得
IDXGIFactory* dxgiFactory = nullptr;
pAdapter->GetParent(__uuidof(IDXGIFactory), (void**)&dxgiFactory);
if (dxgiFactory == nullptr)
{
    return 0;
}

DXGI_SAMPLE_DESC sampleDesc;
for (int cnt = 0; cnt <= D3D11_MAX_MULTISAMPLE_SAMPLE_COUNT; ++cnt)
{
    UINT quality;

    hr = _device->CheckMultisampleQualityLevels(DXGI_FORMAT_R8G8B8A8_UNORM, cnt,
&quality);

    if (SUCCEEDED(hr))
    {
        if (0 < quality)
        {
            sampleDesc.Count = cnt;
            sampleDesc.Quality = quality - 1;
        }
    }
}

```

```
        }  
    }  
}
```

で、やっとスワップチェインに、ここで処理して出てきた、サンプラーのカウントとクオリティを設定して、スワップチェインを作ります。

```
//デバイスとスワップチェーンの作成  
DXGI_SWAP_CHAIN_DESC sd = {};//構造体初期化  
//ここからスワップチェインに対する指定をひたすら書いていく  
sd.BufferCount = 1;//バックバッファの数=1  
sd.BufferDesc.Width = WINDOW_WIDTH;//バックバッファ幅  
sd.BufferDesc.Height = WINDOW_HEIGHT;//バックバッファ高  
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;//バックバッファのフォーマットARGB  
sd.BufferDesc.RefreshRate.Numerator = 60;//分子  
sd.BufferDesc.RefreshRate.Denominator = 1;//分母  
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;//このサーフェスはレンダーターゲットとして使用する  
sd.OutputWindow = hwnd;//ウィンドウハンドルを指定  
//sd.SampleDesc.Count = 1;//サンプリング数  
sd.SampleDesc = sampleDesc;  
sd.Windowed = TRUE;//ウィンドウモード  
  
result = dxgiFactory->CreateSwapChain(_device, &sd, &_swapchain);
```

で、ラスタライザーステートにも

```
culloffdesc.FillMode = D3D11_FILL_SOLID;  
culloffdesc.CullMode = D3D11_CULL_NONE;  
culloffdesc.MultisampleEnable = true;  
_device->CreateRasterizerState(&culloffdesc, &_cullOffState);
```

と、設定します。

うまくいってない理由がわかつたら、また解説します。

## 27 インバースキネマティクス(IK)

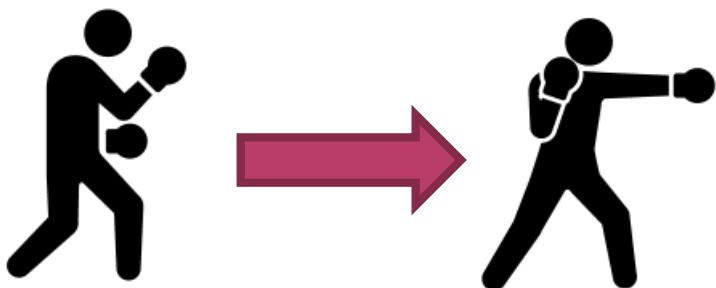
ついにきました。インバースキネマティクス。

もう前回まで影もエフェクトもやっちゃったし、これに入ってもいいでしょう。

このテキスト中最難関なのではないかなーと思います。

そもそもインバースキネマティクスとは何でしょう。知ってる人は知ってるかもしれません、なんていうかな~。日本語で言うと「逆運動学」っていうんですけど、日本語で言われても分かりませんよね。

例えばですね



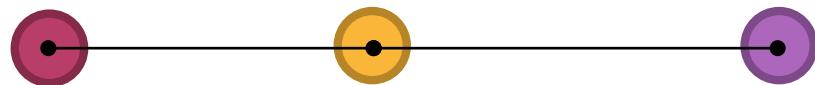
こう…パンチするわけです。

で、フツーにこのポーズを取らせようとするならば、肩を  $45^\circ$  回転させて、肘を  $90^\circ$  回転させてパンチポーズにするわけですね？

で、そういうやり方のことをフォワードキネマティクス(FK)と言います。

これを逆から考えるのが逆運動学で、拳の位置から肘などの位置を逆算するわけです。これをインバースキネマティクスと言います。

例えば



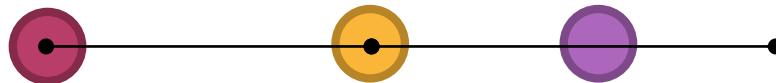
図のように3つの要素があると考えてください。赤が肩で、黄色が肘で、紫が拳にあたると見てください。

そうなると一つの制約が出てきます。

「骨の長さは変わらない」

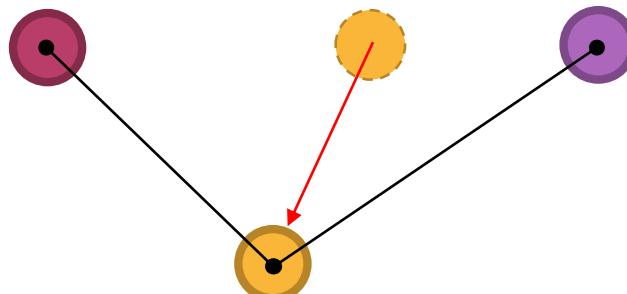
です。関節の角度や位置は変わっても、骨の長さだけは変わりません。変わったら大変なことになりますよね？

ここで拳を肩に近づけていきます。



さて、ただ近づけただけでは、拳から骨が飛び出てしまい、使い物になりません。でも骨の長さは変えられません…どうしたらいいのでしょうか？

この場合、肘の位置を変更します。つまり



このように肘の位置を「骨の長さが一定になり、拳の位置が所定の位置に来るよう」場所を変更します。

で、今回は CCD-IK と言うのを使ってやっていきます。色々と IK の種類はありますが、MMD では CCD-IK を使ってるらしいので、それを使います。

## 27.1 CCD-IK とは…

さっきから当たり前のように CCD-IK と言ってますが、コイツは Cyclic Coordinate Inverse Kinematics の略です。長いので CCD-IK と呼ばれています。

様々な Inverse Kinematics の手法の中でこれが MMD に使用されているのは

- 理論が簡単
- 計算が速い(比較的)

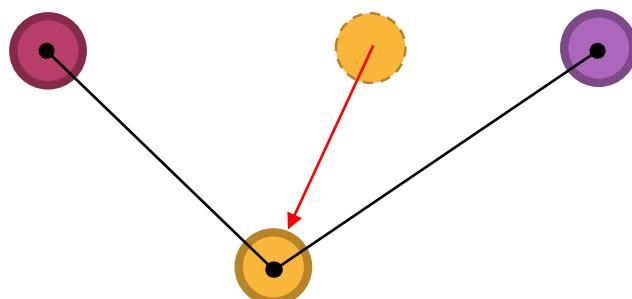
などの理由です。その他の IK を見ましたがよく分かりませんでした。ヤコビアン IK とかワオータニオン IK は直感的に分かりづらかったです。

## 27.2 高校数学だけで IK っぽくしてみよう

えっ!? 高校数学で IK を!?



難しいかもしれません、理屈を分かる上で必要なのでやっていきます。



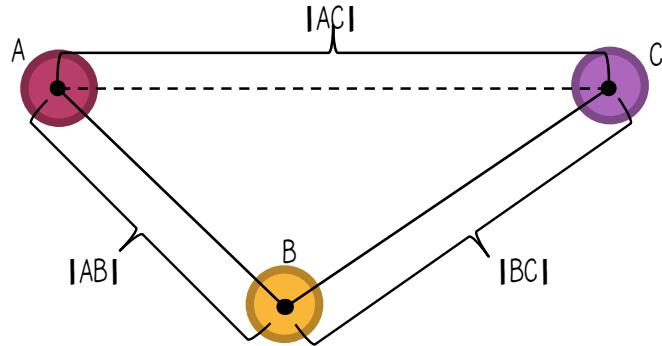
このこの、黄色の位置が知りたい。赤と紫の位置はわかつており、その間の「長さ」は既知であると、そういうわけだ。

仮に赤を A とし、黄色を B とし、紫を C とする。

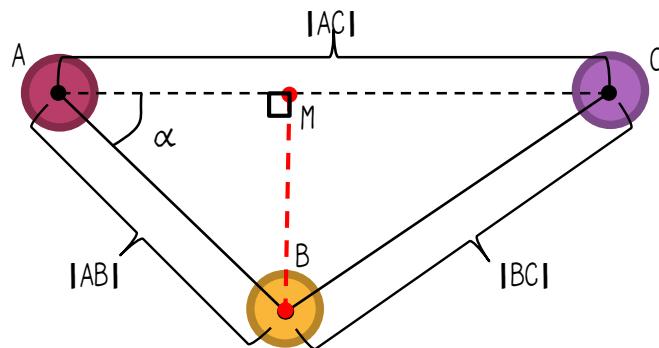
そうすると、既知のものは

座標 A, C および  $|AB|, |BC|$  である。

また、AとCが既知であるため $|AC|$ は $|C-A|$ で求まりますよね？つまりこれも既知。最終的に知りたい $|$ の値はBの座標。



さて、これだけの材料からBの座標を求めましょう。まず、Bから辺ACに対して垂線を引きましょう。ACと垂線の交点をMとします。



ここまで良いですかね？

直角三角形となりますから、 $|\overrightarrow{AM}| = |\overrightarrow{AB}| \cos \alpha = \frac{\overrightarrow{AB} \cdot \overrightarrow{AC}}{|\overrightarrow{AC}|}$ となりますね？ただしベクトル $\overrightarrow{AB}$ がわからなければ、まだ使いのにはなりません。早急点としてはダメですよ？

ここで余弦定理を使います。 $|BC|$ を求めるという体で使います。 $BC$ は既知ですが、これは後で使うからです。余弦定理は

$$c^2 = a^2 + b^2 - 2ab \cos \theta$$

こんなやつでしたね？今回の三角形に当てはめると

$$|BC|^2 = |AC|^2 + |AB|^2 - 2|AB||AC|\cos\alpha$$

です。

この式の中で既知ではないのは  $\cos\alpha$ だけです。ここで先程の

$$|\overrightarrow{AB}| \cos\alpha = \frac{\overrightarrow{AB} \cdot \overrightarrow{AC}}{|\overrightarrow{AC}|}$$

と連立させられないかどうか考えます。おやおや、 $|\overrightarrow{AB}| \cos\alpha$ が使えそうですね？

$$|BC^2| = |AC^2| + |AB^2| - 2|AB||AC|\cos\alpha$$

を変形すると

$$|AM| = |AB|\cos\alpha = \frac{|AC^2| + |AB^2| - |BC^2|}{2|AC|}$$

で、この $|AB|\cos\alpha$ ってのは A から M までの距離ですから座標 M は、A からベクトル AC 方向に  $|AB|\cos\alpha$ だけ進んだ部分ということになります。ですから

$$M = A + \frac{\overrightarrow{AC}(|AB|\cos\alpha)}{|AC|}$$

さらにこの $|AB|\cos\alpha$ は  $\frac{|AC^2| + |AB^2| - |BC^2|}{2|AC|}$ と書けるので、

$$M = A + \left( \frac{\overrightarrow{AC}}{|AC|} \right) \left( \frac{|AC^2| + |AB^2| - |BC^2|}{2|AC|} \right) = A + \overrightarrow{AC} \left( \frac{|AC^2| + |AB^2| - |BC^2|}{2|AC|^2} \right)$$

となるので、既知の情報だけで M が表せます。そうすると座標 M が求まりますね？

ただ、求めたいのは座標 M ではなく、B です。どうすれば良いのでしょうか？

$$B = M + \overrightarrow{MB}$$

ですから、ベクトル  $\overrightarrow{MB}$  を求めれば終わりです。なお、 $|\overrightarrow{MB}|$  はピタゴラスの定理から

$$|\overrightarrow{MB}| = \sqrt{|AB^2| - |AM^2|}$$

ですね。あとは向きがわかれれば良いのですが、向きはベクトル  $\overrightarrow{AC}$  と直行しているため

$$\overrightarrow{AC}^R = \begin{pmatrix} \cos 90^\circ & -\sin 90^\circ \\ \sin 90^\circ & \cos 90^\circ \end{pmatrix} \overrightarrow{AC}$$

で求められます。 $\overrightarrow{AC}^R$  が大きさ情報を持っていると使いにくいためこれを正規化したのと  $|\overrightarrow{MB}|$  を乗算します。そうすると座標 B は

$$B = M + \frac{\overrightarrow{AC}^R |\overrightarrow{MB}|}{|\overrightarrow{AC}^R|}$$

と表せます。これで  $B$  を求めりやれいなので、反復するまでもなく、 $B$  の座標は確定的に明らか。ちなみに回転方向が正の方向と負の方向が考えられますが、どちらでも正しいです。通常は関節に使うので、どちらかはユーザーが決めることがあります。

まあここまで話ならベクトルと内積と、サインコサインしか使ってないので、頭のいい高校生にもなんとか分かる話です。

さて、ここからが本番ですよ。間の関節が複数あるパターンです。それだけで解を求めるのが複雑になってしまいますが、頑張りましょう。

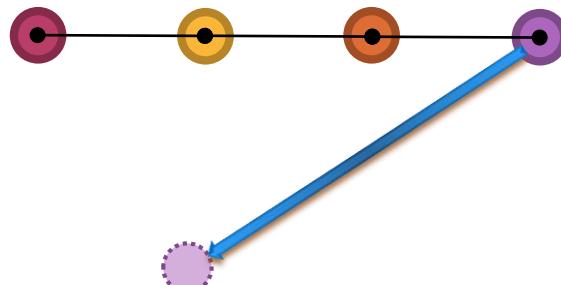
### 27.3 二次元における CCD-IK

さて、実際に CCD-IK をやっていくわけですが、いきなり 3D でやっていくと大変なので、2D における解説からして行こうかと思います。

一文で説明すると

「コントロールポイントを特定の場所に移動させるために、掴んだコントロールポイントからロート方向へ進るように回転処理(最終地点に近づくように)を行い、それを繰り返し近似座標を得る」

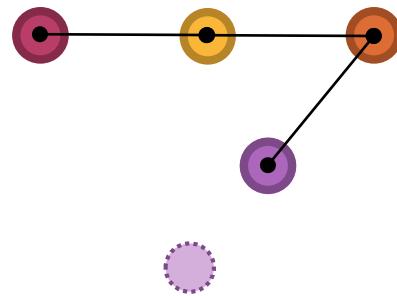
って事です。



図のように紫を動かしたいとします。なお、赤は IK の影響を受けない端点とします。そうなると間の移動対象は黄色と橙ですよね？

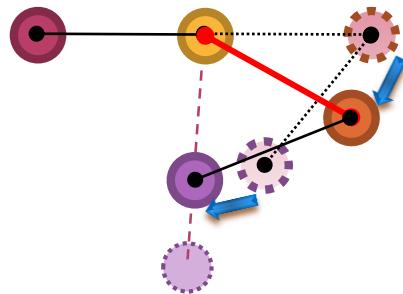
で、先ほどと一緒に各コントロールポイント間の距離は変化しない…と。

まずは対象コントロールポイントを『目的地に最も近づくように』回転させます。

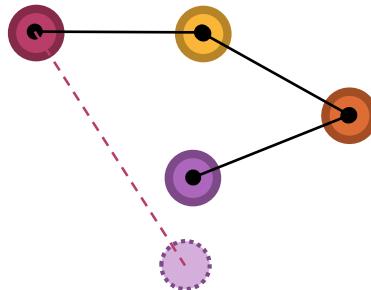


当然ながら届いてません。かつ、間のコントロールポイントを回転させれば更に近づきそうですよね？

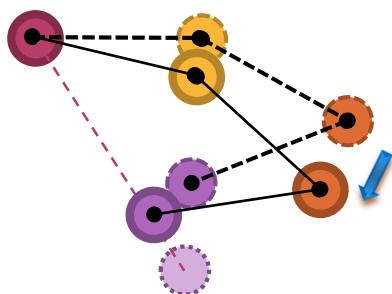
では一つ遡って黄色→橙の線を回転させます。



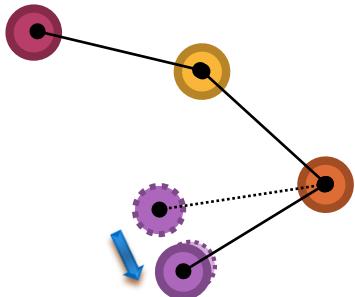
結果としてかなり近づいていますが、もうひと頑張りです。



今度は赤→黄の線を回転させます。



これで一周したわけですが、まだ離れてますので、この状態でまた末端からやり直します。



これを一致範囲内に入るまで、もしくは繰り返し制限回数を超えるまで繰り返します。

これが CCD-IK(2D)です。

## 27.4 三次元における CCD-IK

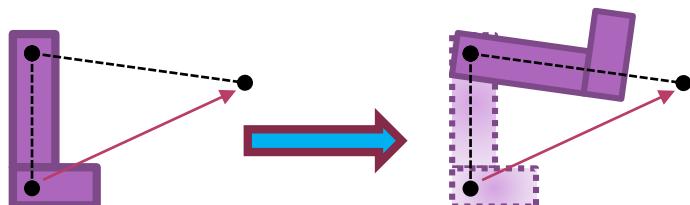
原理は 2D の CCD-IK と同じでいいです。面倒(というかトラブルメーカー)なのは回転の「軸」を決定することです。

で、どうやって決定するのかというと「外積」を使います。「外積」が「2つのベクトルに直交する」事を利用します。

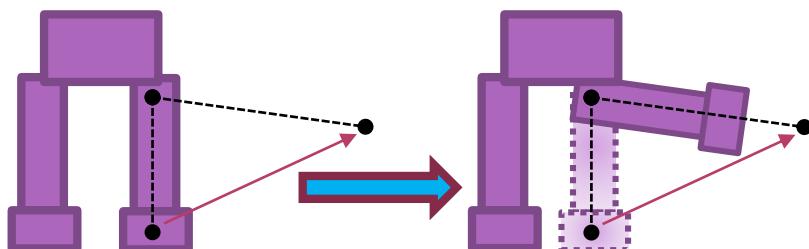
では、何ベクトルと何ベクトルで外積をとるのか?を考えましょう。

どの軸を中心回転すべきなのかについて考えてみれば分かるのではないか…

例えば真横から足の IK を見た場合



図のように足 IK を前にキック的に出そうとするとき、当然軸は X 方向といふか横方向の軸になります。

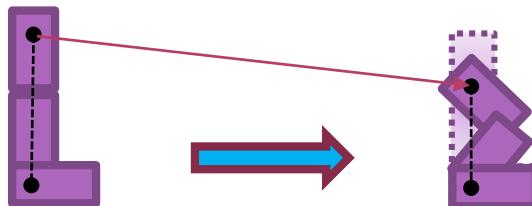


また、例えば図のように、開脚気味に足を動かそうとするなら軸は前向きというかZ方向になりますね？

あ、まだ正解じゃないですよ？でも、ちょっと自分で考えて、あーそーなんだなーって思ってください。

ひとまずこの考えをもとに実装してみてください。足を上げるところくらいまで。

これには穴があります。たとえばしゃがみを考えてください



図のように、しゃがみであれば足のIKとセンターとの距離が短くなるため、そのつじつまを合わせるために膝が曲がるわけですが、先ほどの考え方(だけ)だとうまくいかないことがわかりますね？

何故かって？

ターゲットへの「角度」が変わってないからだよ!! そうですね？

ベクトルの方向が同じままなので、これでは外積が出せません。意味は分かりますかね？

分からない人のためにちょっとだけ解説すると

ベクトルA( $x_a, y_a$ )とベクトルB( $x_b, y_b$ )とします。簡単のため二次元でやりますね？

この外積は

$$A \times B = x_a y_b - x_b y_a$$

ですね？ここでベクトルAとベクトルBが同じ方向を向いていなければ

$$B = kA = (kx_a, ky_a)$$

と書けますね？BはAのk倍ってことです。

さて、その前提で外積を出してみましょう。どうなりますか？

$$A \times B = x_a y_b - x_b y_a = x_a k y_a - k x_a y_a = k x_a y_a - k x_a y_a = 0$$

という感じでゼロになります。外積は同じ方向を向くとゼロになります。

これは三次元でも同様で

$$A \times B = (y_a k z_a - z_a k y_a, z_a k x_a - x_a k z_a, x_a k y_a - k x_a y_a) = (0,0,0)$$

つまりところ、同じ方向を向いたベクトルは外積をとっても意味がないわけです。

じゃあ、しゃがみ時はどうしてるのがかな?と考えよう。

実は3年前のテキストでは、分からぬあまりにこう考えました。

『さて、PMDには「ひざ」のとる角度が特殊だといろんな資料に書いてあったので、信じることにしましよう。』

ということでIK名が「右ひざ」もしくは「左ひざ」ならばX軸方向にしか稼働しないらしいので、強制的に軸をX軸にしちゃいましょう。

簡単です。

```
if(対象IKが"右ひざ"もしくは"左ひざ"){
```

```
    軸.x=-1;
```

```
    軸.y=0;
```

```
    軸.z=0;
```

```
}
```

それ以外の場合は外積で軸を作りましょう。』と言ってました。参考ソースコードをお見せしますが、

```
///CCD_IKを行い、結果を返す。
```

```
///@param ikmap IKデータから取ってきたIKボーン名とIKリストデータのマップ情報
```

```
///@param boneInfo ボーン名によっては特殊な動きをする必要があるため『名前』取得のためボーン情報への参照を受け取る
```

```
///@param ikname 対象IKボーン名
```

```
///@param location 動かしたい先の座標
```

```
///@note 重要なのは…というか必要なのはそれぞれのボーンをどれだけ回転するかの情報だけ
```

```
///だが、CCDであるため途中経過の座標も重要なので、関数内部にテンポラリとして保持しておく
```

```
///関数を抜けたら用済みである
```

```
void CCD_IK(std::map<std::string, IKList>& ikmap, std::vector<BoneInfo>& boneInfo, const char* ikname, XMFLOAT3& location){
```

```

int ikIdx=ikmap(ikname).boneIdx;//IKボーンマトリクス番号を取得
std::vector<unsigned short>& itn=ikmap(ikname).boneIndices;//IKに対応するボーンのインデックス
std::vector<unsigned short>::iterator it=itn.begin();

std::vector<XMFLOAT3> tmpLocation(itn.size());//IKから支点までのボーン座標(IKは含まない)
XMFLOAT3 ikpos=_boneoffsets(ikIdx);//IKの座標
XMFLOAT3 targetpos=ikpos+location;//IKの座標(目的地やから変わらへん)

int nodecount=itn.size();//チェーンノード数

for(int i=0;i<nodecount;++i){
    tmpLocation(i)=_boneoffsets(itn(i));
}

for(int c=0;c<40;++c){
    for(int i=0;i<nodecount;++i){
        if(ikpos==targetpos){
            break;
        }
        XMFLOAT3 originVec=ikpos-tmpLocation(i);//もとの先っちょIKとさかのぼりノードでベクトル作成
        XMFLOAT3 transVec=targetpos-tmpLocation(i);//目標地点とさかのぼりノードでベクトルを作成
        //ベクトル長が小さすぎる場合は処理を打ち切る
        if(abs(Length(transVec))<0.0001 || abs(Length(originVec))<0.0001){
            return;
        }

        //軸作成
        XMFLOAT3 norm=Normalize(originVec)^Normalize(transVec);//ちなみに^は外積として扱っている
        const char* name=boneInfo(itn(i)).boneName;
        if(strcmp(boneInfo(itn(i)).boneName,"右ひざ")==0 || strcmp(boneInfo(itn(i)).boneName,"左ひざ")==0{
            norm.x=-1;//norm;
            norm.y=0;
            norm.z=0;
        }
    }
}

```

```

}else{
    if(Length(norm)==0){
        return;
    }
}

//角度計算
XMVECTOR
rot=XMVector3AngleBetweenNormals(XMLoadFloat3( &Normalize(originVec)),XMLoadFloat3(&Normalize(transVec)));
rot*=0.5;
//角度が小さすぎる場合は処理を打ち切る
if(abs(rot.m128_f32(0))==0.000f){
    return;
}
float strict=(2.0f/(float)nodecount)*(float)(i+1);
if(rot.m128_f32(0) > strict){
    rot.m128_f32(0)=strict;
}
XMVECTOR q=XMQuaternionRotationAxis(XMLoadFloat3(&norm),rot.m128_f32(0));

//ボーンの変換行列を計算
XMFLOAT3 offset=_boneoffsets(ihn(i));
XMMATRIX RotAt=XMMatrixTranslation(-offset.x,-offset.y,-offset.z)*
    XMMatrixRotationQuaternion(q)*
    XMMatrixTranslation(offset.x,offset.y,offset.z);
_boneMatrices(ihn(i))=_boneMatrices(ihn(i))*RotAt;

offset=tmpLocation(i);

//理論上の変換行列を計算
RotAt=XMMatrixTranslation(-offset.x,-offset.y,-offset.z)*
    XMMatrixRotationQuaternion(q)*
    XMMatrixTranslation(offset.x,offset.y,offset.z);

//理論値を更新
ikpos*=RotAt;
for(int j=i-1;j>=0;--j){

```

```

tmpLocation(j)*=RotAt;
}

}

}

}

と。

```



当時はこれで上の画像のように、きちんと動いたものだから「これが正解じゃ!!」と思って当時の学生にドヤ顔で教えていたんですが、よくよく考えると、足を開いた状態でしゃがむと、重心軸もX軸に平行じゃないわけだから、おかしなことにならぬいかな?

ちなみに当時のコードでやってみたらこうなりました。



まともそうに見えますが、やっぱり膝が不自然なんですよね。ていうか、特定の角度の時にやっぱりおかしなことになります。一瞬なのでスクショ取れてませんけど。

3年前より僕は知識と技術は上がっているはず…ならば、何かあの頃に思いつかなかつたことも思いつけるはず…そうか!!軸のことばかり考えるからこうなるんだ!!全く違うことを考えなければならぬ! そう思うのだ!!!

三年前の俺になくて、今の俺にあるもの…。

そう…それは、

## LookAt 行列

LookAt 行列というのは、特定のベクトルを特定の方向に向かせるための行列です。ちなみに「LookAt 行列」って名前は僕が勝手に名前付けてるだけなので、インターネットで探しても出てきません。

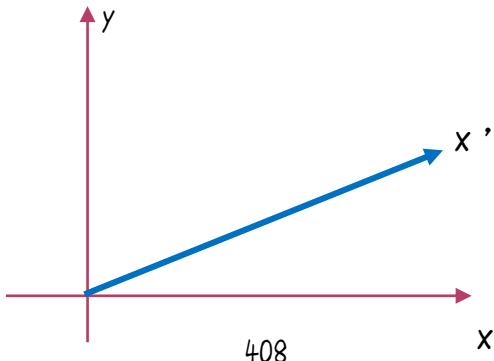
いや、出てきた。`gluLookAt`ですね。これは OpenGL の関数で、でもこれはカメラに使うやつなので、今回の用途には合いませんし、XMMatrix 系にも LookAt は存在しません。

僕が LookAt, LookAt 言ってるのは、Unity の Transform の関数に LookAt ってあるからです。そいつは、オブジェクトの  $\text{z}$  方向を任意のベクトル方向に向かわせるって関数です。

この関数はたぶん今回の IK だけではなくいろんな所で使うことになると思います。例えば敵がこちらを発見して顔をこちらに向けるとか、そういう事をするために必要なので、作っておきましょう。

### 27.4.1 LookAt 行列の作成

今回も最初は 2D で考えてみよう。結果的には回転を表すが回転角は分からず、ベクトルによる方向指示のみとする。で、向かせたいベクトルを  $x'$  とする。



この $x'$ は正規化されているとします。

で、角度を用いずに $x'$ を表すならば、 $x$ 方向の単位ベクトルと $y$ 方向の単位ベクトル…つまりどちらも長さが1のベクトルを用いて表すならば

$$x' = m\vec{x}_i + n\vec{y}_i$$

と表せます。

$m, n$ は任意の実数で、 $x_i, y_i$ はそれぞれ $x$ 方向、 $y$ 方向の単位ベクトル(長さ1)です。だから別に $x_i, y_i$ ってのはいらないっちゃいらないけどね。意味的にそう書いてるだけです。なんでわざわざこれを書いているのかというと、「ベクトル」ということにしていれば、目的地をベクトルの足し算で表すことができますね？

通常であれば座標 $(a, b)$ というのは

また、 $x'$ は正規化済みですから。

$$\sqrt{m^2 + n^2} = 1$$

とします。

ここまでよろしいでしょうか？

で、やりたいことは元の $X$ ベクトルに行列 $M$ をかけて $X'$ になる。そういう行列を考えたいとします。

元の $X$ は $(1, 0)$ なので

$$(1, 0) \times \begin{pmatrix} a & b \\ c & d \end{pmatrix} = (m, n)$$

こうしたいわけ。そうなると当然

$$(1, 0) \times \begin{pmatrix} m & n \\ ? & ? \end{pmatrix} = (m, n)$$

となるだろう。ただしこのままで

$$(0,1) \times \begin{pmatrix} m & n \\ ? & ? \end{pmatrix} = (? , ?)$$

となってしまい、X 軸方向のことしか考えてない。Y' は X' から 90 度ずれてる部分なので本当  
は

$$(0,1) \times \begin{pmatrix} m & n \\ c & d \end{pmatrix} = (-n, m)$$

となってほしい。

そうなると

$$(0,1) \times \begin{pmatrix} m & n \\ -n & m \end{pmatrix} = (-n, m)$$

となる。これが新しい Y 軸のベクトルだ。

どこかで見た形だなーって思った人は感がいい。コブラのマシンはサイコガンのあいだ。回転  
行列と同じような形になっている。さらに

$$\sqrt{m^2 + n^2} = 1$$

であるから、ますますもって m, n は sin, cos と対応しているのである。

回転行列に対応しているということは、特定の点を回転させるためには、それぞれの新しい  
軸との内積になっていることがわかるだろうか？

つまり、意味合いからすると、それぞれの軸に対する内積が、それぞれの軸の成分だったわけ  
で、それを 3D に拡張すると、どういう話になるのでしょうか？

結局は 3D の時もそれぞれの軸に対する内積をとり、それを全部合成したものが新しい座標  
となります。それができるような行列を作ればいい。

まとめいうと、

- X 軸との内積が X 座標
- Y 軸との内積が Y 座標

- $\vec{z}$  軸との内積が  $\vec{z}$  座標

これは分かると思いますが、これがなんかしら変換された後の  $XY\vec{z}$  に対しても同様である。  
つまり、特定の方向を向かせたければそれぞれの  $X, Y, \vec{z}$  に当たるベクトルをそれぞれ計算し、  
それぞれに対して「内積」をとってしまえばいいわけです。

じゃあ「どこか向く」と言った場合、3軸必要かつていうと、そうじゃなくて、カメラ行列の時も  
そうでしたが、二つあれば外積で直交ベクトル出せるので十分です。

つまり

向かせたい方向ベクトル:  $\vec{z}$  軸

うーん。ここでアツ! パーベクトルとか言って安易に  $(0, 1, 0)$  ってやっても良いんですが、センター  
から末端 IK が「真下」を向いてると結構大変なことになります。

くっそー!! 結局ここで軸を決めないといけないのか…。

うーん。重力方向は避けたいので  $X$  軸を決めよう。右ベクトルとして  $(1, 0, 0)$  を使おう。

ここで二次元の時を思い出してほしいんですが

$$\begin{pmatrix} m & n \\ -n & m \end{pmatrix} = \begin{pmatrix} \text{回転後のX軸} \\ \text{回転後のY軸} \end{pmatrix}$$

になっていますから、これを 3D に拡張すると

$$\begin{pmatrix} \text{回転後のX軸} \\ \text{回転後のY軸} \\ \text{回転後のZ軸} \end{pmatrix}$$

になります。これは言い換えると

$$\begin{pmatrix} x'_x & x'_y & x'_z \\ y'_x & y'_y & y'_z \\ z'_x & z'_y & z'_z \end{pmatrix}$$

となります。

です。この確認は $(0,0,1)$ が $(z'_x, z'_y, z'_z)$ となればオッケーなんですが、計算してみてください。  
なりますよね？

そうすると

$$(1,0,0) \rightarrow (x'_x, x'_y, x'_z)$$

$$(0,1,0) \rightarrow (y'_x, y'_y, y'_z)$$

$$(0,0,1) \rightarrow (z'_x, z'_y, z'_z)$$

という変換になるのが分かると思います。こういう行列を返す関数を作ればいいです。

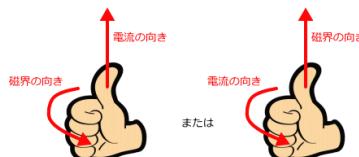
ここで外積に関して注意点ですが、外積は掛け算の順序によって符号が入れ替わります。例えば

$$(x_a, y_a, z_a) \times (x_b, y_b, z_b) = - (x_b, y_b, z_b) \times (x_a, y_a, z_a)$$

というわけです。ベクトルで向きが入れ替わるってことは、真反対に向くわけです。ここまで  
は大丈夫でしょうか？

ベクトルの向きをあまり考えなしに、適当な順序で外積とっちゃうと大変なことになります。

というわけで順序を考えなければならんのですが、どういう法則で外積後のベクトルの向き  
が決まるのかというと「右ねじの法則」によって決まっています。高校の電磁気学で出てきた  
あれですわ。



忘れたとか知らないって人までサポートしてたら、ページ数が膨大になるので、詳しくは

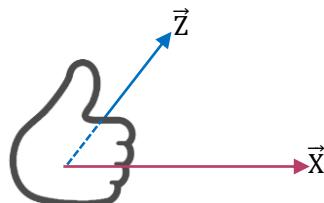
<https://ja.wikipedia.org/wiki/%E5%8F%B3%E6%89%8B%E3%81%AE%E6%B3%95%E5%89%87>

見てくれ。

具体的には、右手系の場合、二つのベクトルから直行するベクトルを出すときには、まず手のひらをシュッとさせて外積の左辺値の向きに右手の指先が向くようにする(図は $\vec{x} \times \vec{z}$ のとき)



次に右辺値ベクトル( $\vec{z}$ )の向きに指先だけカクっと向ける。その状態でグッジョブする。



この時の親指の方向が $\vec{y}$ になるというわけだ。

これが高校の理科の時間に習う「右ねじの法則」である。あるが、これは右手座標系の話である。確かに言われてみればそうなのだ。

つまり系が変われば、それに合わせるには外積の順序も変えなければならない…これは悩ましいことだが、最初に系を確定させて、それに全てを合わせるということが…わかるだろう?つまり DirectX の座標系を扱うときは「左ねじの法則」だ。ややこしいね。

ということを考えて書いてみた関数が以下の通りだ。

```
//特定の方向を向かす行列を返す関数
XMMATRIX LookAtMatrix(XMFLOAT3& lookat, XMFLOAT3& right){
    XMVECTOR vz = XMVector3Normalize(XMLoadFloat3(&lookat));
    XMVECTOR vx = XMVector3Normalize(XMLoadFloat3(&right));
    XMVECTOR vy = XMVector3Normalize(XMVector3Cross(vx, vz));
    vx = XMVector3Normalize(XMVector3Cross(vz, vy));
    XMVECTOR vy = XMVector3Normalize(XMVector3Cross(vz, vx));
```

```

vx = XMVector3Normalize(XMVector3Cross(vy, vz));

XMMATRIX ret = XMMatrixIdentity();
XMFLOAT3 fvx, fvy, fvz;
XMStoreFloat3(&fvx, vx);
XMStoreFloat3(&fvy, vy);
XMStoreFloat3(&fvz, vz);

ret._11 = fvx.x; ret._12 = fvx.y; ret._13 = fvx.z;
ret._21 = fvy.x; ret._22 = fvy.y; ret._23 = fvy.z;
ret._31 = fvz.x; ret._32 = fvz.y; ret._33 = fvz.z;
return ret;
}

```

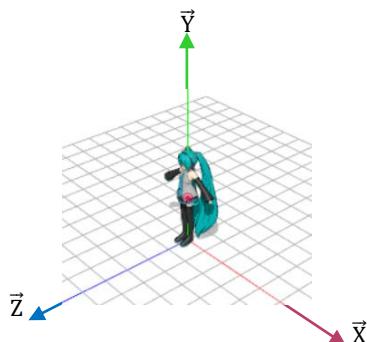
確認のために、キャラクターの $\vec{z}$ 軸を特定の方向に向けるのをやってみてください。

と書いたところで、いろいろと問題が発生した。

僕のプログラムで起きた既知(キチ)の問題は以下の通りだ

- 真正面のはずなのに尻を向ける
- モデルが縮む
- モデルが消える

お尻を向ける…といふか $\vec{z}$ 軸が反転してしまう原因は、座標系にある。MMD を起動し、エディタを見てもらえばわかるが、



右手系だよこれ!!どうして DirectX なのに左手系にしなかった!!まあたぶん他の CG ソフトに合わせたんでしょう。

ともかく、つまるところ $\vec{z}$ 方向が最初からお尻向けてるんですね。これは仕方ない。

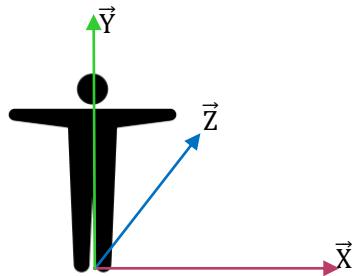
対応策がいくつか考えられますが、

- 表示の際にY軸180°回転させとくのとLookAtはお尻向きで設定する
- 読み込みの際に左手系にしてしまう(Ｚ値反転させてインデックス逆にするかカーリング逆にする)

こんな感じでしょう。どちらにするのかは皆さんにお任せします。正解はないんです。

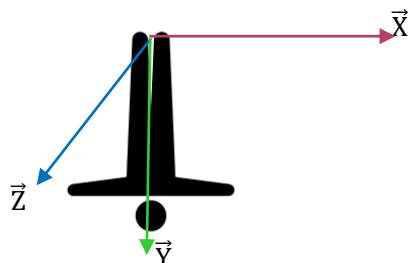
次に「モデルが縮む」ですが、これは単純に出来上がったそれぞれのベクトルをノーマライズし忘れていただけでした。

最後の「モデルが消える」ですが、これはＺを反対に向けた時に地面の反対側に行ってました。どういう事がというと、



このＺをこっちに向けてみましょう…ただし「右手系」と「右方向」は変えずに…。

そうすると、



こうなるわけです。「吊られた男」かよ!!

さて、ここにきて思い当たるのが「アツパー(Y)ベクトル」です。カメラやモデルにおいて向きを決めるときには、たいていの場合はアツパー(上方向)ベクトルです。つまりここを固定しておかないと、逆さになるから、rightでもleftでもなく上ベクトルを基準にしたのですね。

つまり、X 軸固定にしたときは、Y 反転の際に X 軸周りに 180° 回転したのに対して、アツリペーベクトルを使用する場合は Y 反転の時に Y 軸周りに 180° 回転しているというわけです。

通常、人間が後ろを向くときは Y 軸中心に回転するからで、Y 軸を基準にするのが通常は正しいというか、都合がいいことになります。

ということで、先ほどの関数を改変するならば

- 引数には Upper と Right の両方のベクトルを渡しておく
- 通常は Upper を基準として LookAt 行列を作成
- 真上を向くときなど Upper が向かせたい向きと同じになってしまっていれば、その時は Right のほうを使用する

といった感じだろうか。

まだノック混入しそうだが、今のところつじつま合わせるためにはこれが簡単だろう。

さて、ここでまた CCD-IK の話に戻る。

#### 27.4.2 納得いかないんですが…

まあ、CCD-IK の正式な記事は Computer Graphics Gems JP 2012 の記事の作者である

<http://mukai-lab.org/library/ik-legacy.html>

氏のやり方を参考にさせてもらったのですが…このやり方でうまくいくんです。うまくいくんですけど…今冷静に考えると納得行かないんですね。

うん、まあもう時間ないんで「論文に書いてるんだからその通りで正しいんだ」でもいいんですけど、そんなもんエンジニアの姿でも教育者の姿でもないんですね。



### 27.4.3 夢はひろがりんぐ

んで、そこまでしてなんでIKをやりたいのかというと、FKでもエエんちゃうのか?というと

<https://ja.osdn.net/projects/mmdmotion-java/wiki/MMDIKSolver>

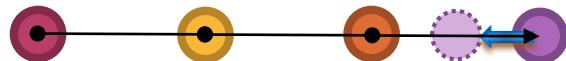


こんな風に特定の点を追いかけて体全体を動かすとか、バランスをとるとかやりたいわけです。これができれば階段とかで



手付けアニメーションだと階段の高さによってアニメーションを変えなければならないのですが、足の位置との当たり判定で全身のポーズを決めることが可能になるわけです。

### 27.4.4 何で納得いってないのか & 実験



例えば図のように平行に視点に近づけるように端点を動かそうとするときです。当然ながら「長さを変えてはいけない原則」があるため、関節を曲げなければいけません。

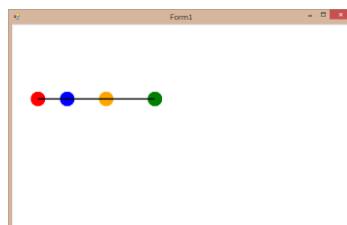
ところで CCD-IK ってどういう法則で曲げていきましたか? そう、ボーンを現在のエフェクタ(コントロールポイント…つまりそこへ動かしたい先の座標)に向くように(最も近づくように)曲げてあげるんですが…上の図を良く見てください。

エフェクタの位置は何処にありますか?

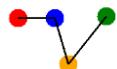
そう。最初に配置されていたボーン直線の真上です。そしてそこから全てのボーンが一直線上に並んでいます。

既にボーンの向きはベクトル的には目的地を向いています。これでは曲げようがありませんね？

試しにこういうアプリケーションを作つて実験してみました。



で、端点を左に押し込むと…



あれ？ 曲がつてしましましたね？ 予想外です…があることに気が付きました…人間だもの。押し込む際にまっすぐやつてゐるつもりがちょっと上下に行くわけよね。

そう考えて Shift キーを押しながら移動させると Y 座標が変わらないようにしてみます。そうすると…



詰まりました。なるほどそういう単純なことだったのか…OTL。

ちなみに ほんのちょっとでも曲げた状態で Shift 移動させると、正しく IK を行います。つまり ピーンとまっすぐになっている時は CCD-IK であったとしても詰まっちゃうってことです。良かっただ。安心した。まあ、ニンゲンでも関節伸ばした状態で関節方向に骨を押し込むと関節外したり関節破壊したりできますもんね…。

#### 27.4.5 まっすぐに対する解決案

といふわけでまずは今回の実験結果を大前提に考えましょう。そう考えると膝を曲げる動きの場合には2つの可能性が考えられます。

- 予め曲げられている
- まっすぐ押し込もうとされた場合には適当な方向にちょっとだけ曲げてやる

3年前のスクワットがうまく行っているのは前者の状況だったからだと推測できます。後者の対処をしてるならプログラムに何らかの痕跡があるはずですが、ありませんでしたし…。

では、何はともあれ、IK を実装していきたいところですが、まだ解決というか決めていかなければならぬことがあります。

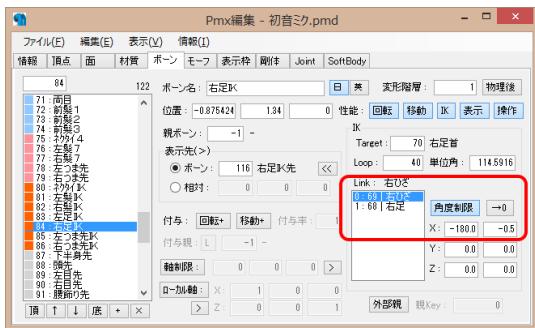
#### 27.4.6 曲げる軸について考える

二次元ならば軸は完全に「**Z 軸**」なので良いんですが、3D の場合、軸をどう取るのかっていうのがかなり悩ましいという話は前にしました。それを解消するために LookAt を作ったはずなんですが、結局「**制限角度**」を正しく設定するためには必要なんだよね。

なんとかというと、軸ベクトルが反対側向いてると角度の符号が変わっちゃうんだよね…。

角度制限がないなら基本的には LookAt のみでやっちゃってオッケー（結局固定すべき軸は決めておく必要はあるんだけど）。そうではあるんだけど人間の骨は大抵制限がついてるからね。仕方ないね。

さて、では手始めに「**ひざ**」について考えよう。一応 PMXEditor でミクさんを開いてみると



こんな感じです。こんな感じ言われてもよーわからへんといった具合なんでしょうね。右側のパラメータを見てもらうと分かるように、角度制限がついてますね。-0.5°～180.0°ということで、予めちょっとだけ曲げられているのが分かると思います。

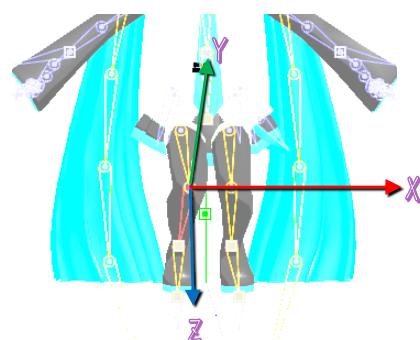
まあ、モデル自体はそういう風に作られていることもあり、3年前はうまく行ったんでしょうね。

ただ、前にもお話ししたように「**ひざ**」だからといって、回転軸を強制的に X 軸にするようでは

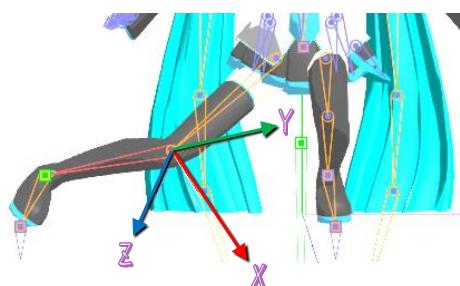


こんなふうになつた時にうまく行きません。

原因を言うと、普通にしゃがんだ時にはこうなんです。



Xが右方向を向いています(画像は右手系なので注意してね)。この時はワールドX軸中心に回しても問題ないんですが



まあ、当たり前の話なんんですけど、↑の図の座標系のX軸中心って意味なのよね。

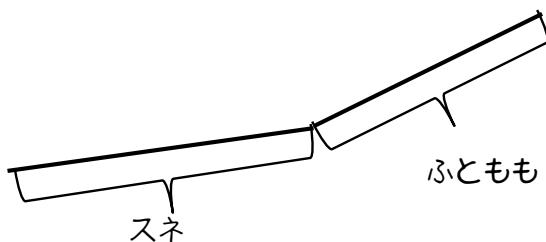
ということは、何かと何かの外積をとって、このX軸を計算しなければならないわけだ。



この図を見ると、右足 IK は右ひざと右足に対してリンクを持っているのがわかりますね？2つのリンクを持っているわけです。この数は可変ではあるんですが、大抵の場合は1~2です。

で、何度か試してみてわかるのは、太ももとヒザ(スネ)の座標系(軸)は一致させておかないと、おかしなことになるということです。

例えばこの「位置関係」が正しかったとしても、それぞれの座標系「軸」がずれていれば、膝関節でねじれが発生するわけです。



経験上わかると思いますが、「ありえない」ですよね？ヒザとスネの向きが違うということは、それはもう関節外れてますから。

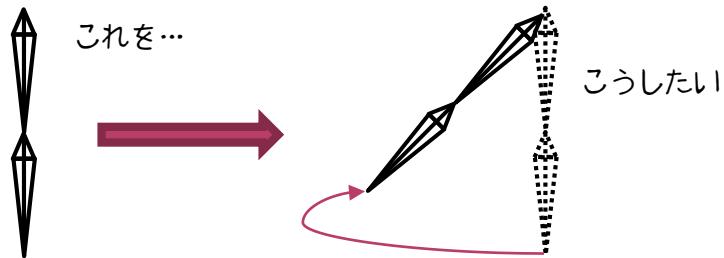
ただ、別の生き物や機械とかの場合はその限りじゃないです。ですが、多分、向きは同じにしておかないと IK を使う時には「解」がたくさん出てきてやりづらいと思いまして、統一しておいたほうが良いでしょう。

で、膝を使用する場合、X 軸を中心に曲げるという事だが、これは股関節を開いているときに、不具合を起こす。

そうだよね。さっきも書いたけど、股関節の開きに合わせて膝の X 軸方向を回転させなきゃならない。

どれくらい回転させたらいいんだろう。で、やっぱりここで「度」とか使いたくなかった。ベクトルだけでなんとかならないものだろうか。

ということで考えました。



2Dなら簡単に求められるところだが、3Dの場合だと、内積出して角度計算して外積で軸出して、その軸周りの回転行列を求める…となる。

もう少し簡単にいいかないものか？

『いや、特定のベクトル A を特定のベクトル B に向かせたいんでしょ？じゃあ LookAt で作った行列でいいんじやね？』

そうだわな。ただ、あれは  $\vec{Z}$  ベクトルが特定のベクトルに向くように作っているので、

$$R_{ZB} : \vec{Z} \Rightarrow \vec{B}$$

あって、

$$R_{AB} : \vec{A} \Rightarrow \vec{B}$$

ではない。

そういうことだ。俺は  $A \Rightarrow B$  が欲しいんだよ!!! さて、行列のことを考えまくって、なんとか  $R_{AB}$  を出せないか考えた。

みんなもしばらくは自分で考えてみてくれ。

うーん、武器が足りない。ほかに LookAt で分かるものは…ああ!! そうだ!!

$$R_{ZA} : \vec{Z} \Rightarrow \vec{A}$$

というわけだ。さらにこれを逆に考えると

$$R_{ZA} = R_{AZ}^{-1}$$

つまり、LookAt( $\vec{Z} \Rightarrow \vec{A}$ )

ちなみに今回のベクトルは、長さを変えない、移動しないベクトル→回転のみということだ。

$$\vec{A} \Rightarrow \vec{B}$$

はちょっと変形すると

$$\vec{A} \Rightarrow \vec{Z} \Rightarrow \vec{B}$$

となることはわかるよね？ $A$ から $B$ まで回転するには、 $Z \Rightarrow B$ と  $A \Rightarrow Z$ を組み合わせれば

$$A \Rightarrow B = A \Rightarrow Z \times Z \Rightarrow B$$

だから

$$A \Rightarrow B = (Z \Rightarrow A)^{-1} \times Z \Rightarrow B$$

というわけ。つまり

XMMatrixTranspose(LookAt(A,up,right))\*LookAt(B,up,right);

とでもしてやればいい。

#### 27.4.7 LookAt 行列関数の改造

これをを利用して、任意ベクトル  $A$  を任意ベクトル  $B$  へ向かわせる LookAt 関数のオーバーロード関数を作つてみよう。元の LookAt は 27.4.1 を見返してくれ。

大雑把に描くとこんな感じ。

```
XMMATRIX LookAt(XMFLOAT3& origin, XMFLOAT3& lookat, XMFLOAT3& up, XMFLOAT3& right){  
    XMMATRIX tmp=LookAt(origin,up,right);  
    tmp=XMMatrixTranspose(tmp)*LookAt(lookat,up,right);  
    return tmp;  
}
```

これを使用すれば、元の IK ベクトルを、IK ターゲット移動後 IK ベクトルにする行列ができるります。

## 27.5 いよいよ実装である

長かった…ホンマに長かった。そして苦しめられた…。某クラスのインフルエンザアウトブレイクによりカミサマから猶予期間を与えてもらったにも関わらず、最終的にはまだ不具合が残っている状況にある…申し訳ない。CCD-IK の論文を理解して 2D の IK を実装するのと、MMD の IK を実装することの間にはまだまだ深い深あへい谷があつたのだ。



ただただ申し訳ないが、それ言っても前に進まないので、実装しよう。最初に言っておくけど、まだ不具合が残っていることは認識しておいてほしい。やっぱりヒザ周りが難しいのだ。

もし、完璧にしたい人がいるなら、オープンソースの MMD エディタや MMD 再生機や、Java や Javascript でやっている強者もいるので、そちらを見ながら研究してほしい。

[http://dxlib.0.007.jp/DxLib/DxLibMake3\\_17a.zip](http://dxlib.0.007.jp/DxLib/DxLibMake3_17a.zip)

<https://ja.osdn.net/projects/mmdmotion-java/wiki/MMDIKSolver>

<http://www.nicovideo.jp/mylist/49125767>

<https://jthird.net/>

<https://github.com/edvakf/MMD.js>

<https://code.google.com/archive/p/nymmd/>

※ちなみに現在 jThreee は開発停止&サイト閉鎖されています。

投げっぱなしに思われても仕方ないのだが、うーん。まあキチンとやろうとすると、集中した潤沢な時間が必要だってことです。

それはともかく、実装してみましょうってのがこの章の趣旨である。

だが、ここに入る前にも最初に言ったように「IKは十分条件であって必要条件ではない」のでそこまでしなくてもいい。余裕があったらいい。特に次年度就職年次はそんな時間があるなら就活用の作品を作つたほうがいい。

さて、言い訳は済ませたので実装しよう。

### 27.5.1 PMD から IK 情報を取得する

久々に PMD のデータを取得するぞ。IK の情報だ。

[http://blog.goo.ne.jp/torisu\\_tetosuki/e/445cbbbe75c4b2622c22b473a27aaae9](http://blog.goo.ne.jp/torisu_tetosuki/e/445cbbbe75c4b2622c22b473a27aaae9)

さて、見てもらえばわかるが、相当アレだ。データの場所はボーン情報を読み切った後のところなので、ボーンを全読み込みしてるのでみんなは素直に読み込んでもらえばいい。

まず、最初の2バイトがIKデータの数だ。2バイト読み込んで数を確認しよう。デフォルトの初音ミクなら、7個になっているはずだ。

6B51Fbone[121].bone_head_pos[0]	3FA499A6	418A43E9	BFA54
6B52Bik_data_count	0007		
6B52Dik_data.ik_bone_index	0050		
6B52Fik_data.ik_target_bone_index	004B		
6B531ik_data.ik_chain_length	03		
6B532ik_data.iterations	000F		
6B534ik_data.control_weight	3CF5C28F		
6B538ik_child_bone_index[0]	0008	0007	0006
6B53Eik_data.ik_bone_index	0051		
6B540ik_data.ik_target_bone_index	004C		
6B542ik_data.ik_chain_length	05		
6B543ik_data.iterations	0008		

確認してくれ。

で、結構クセモノなデータである。IKデータひとつあたり

2+2+1+2+4+可変長\*2

という状況だ。もうアライメントを意識していないのは分かった。そこは諦めてる。

だが、最後のIK影響ボーン番号が可変長だから、必要なデータを「ガ-っと」取得することは不可能なのだ。

IKデータを一つ一つ丁寧に取得しなければならない。面倒だが、データの総量はそれほど多くもないはずなので、コストはそこまでかかりないんじゃないかな…とは思います。ちなみにこのIKの後にスキンデータ(表情データ)が控えているわけだが、IKを実装しなくても、表情を出したい場合はIKを丁寧に読み込まなくてはスキンデータの先頭すらわからないのだ!!!

もはや文句言う気にもならない(いや、IK再生を完成させられない僕にそんな資格など最初からないのだ)

構造体こんな感じかな

```
///@brief IKリスト
///IKボーンとターゲットボーンの二つがあるのは、
///それでベクトルを作るためであります。
struct IKList{
    unsigned short boneIdx;///<IKボーンインデックス
    unsigned short tboneIdx;///<ターゲットボーンインデックス
    unsigned char chainLen;///<さかのぼりボーン数
    unsigned short iterationNum;///<巡回回数
    float restriction;///<制限角度
    std::vector<unsigned short> boneIndices;///<さかのぼりボーン番号
};
```

で、アライメントの関係があるのでメンバを一つ一つ読み込むか#pragma pack(1)を使ってください。

僕は最後のメンバが可変長である以上どのみちガーッととってこれないので、一つ一つ丁寧に読み込みします。読み込んでください。

そんなに数はないので、おかしなデータになってないかどうかご確認ください。

もちろんこのIKListは配列状態になっていますのでいつも通り

```
std::vector<IKList> _iklist;
```

として読み込んでいきます。

この時にちょっとだけ注意点ですが、VMD データとしては IK であるとかの情報があるのではなく、ボーンの状態だけが格納されています。

このため、後から IK と辺縁を合わせられるように、IKList はベクタでなく、マップで持ったほうが良いかもしれません。やりようは色々あるので、各自ご研究あらんことを。

### 27.5.2 CCD\_IK 関数について

作りましょう。そうしましょう。ややこしいんですけどね。ちなみに通常の回転と IK とどっちを優先すべきかというと、IK の方です。

MMD のエディタ見れば分かりますが、IK 動かさない限り IK 周りの回転は反映されません。IK を切ると反映されますがね？

まあ…ぶっちゃけた話をすると…股関節→膝→足首であれば、三点なので、この場合に関しては CCD-IK を使わずとも軸さえ決まれば余弦定理で行けるとは思うんですけどね…うーん制御点が 3 点以下とそれ以上で場合分けしてもいいかなー。

2つのベクトル間の角度を知るための関数が用意されています。

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.geometric.xmvector3anglebetweennormals\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweennormals(v=vs.85).aspx)

XMFLOAT3AngleBetweenNormals はベクトル間の角度を返すものだ。

ちょっと疑問なのが「ベクトルを返します。各要素に N1 と N2 の間のラジアン角度が複数されます。」

の下線の部分…。確かにラジアン値がすべての要素に同じ値が入っている…これ、意味あんのか？ float 一個返したほうが速くね？

って思うんだけど、定義まで見ると、どうも SIMD 拡張命令を使用しているらしい。申し訳ないがあまり検証する気にはなれない。ともかく 0 番要素を取ってくればいいだろう。

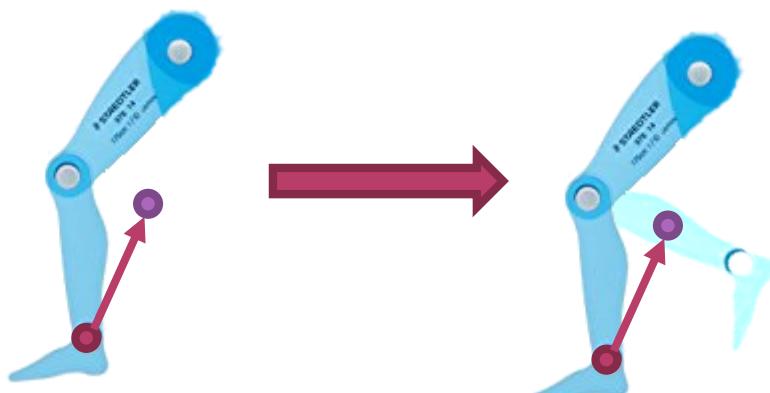
ベクトル値を受け取るが、ベクトル値ではないことに注意してくれ。

んで、最初は 2D のつもりで作っていきましょう。今は膝のことだけ考えて



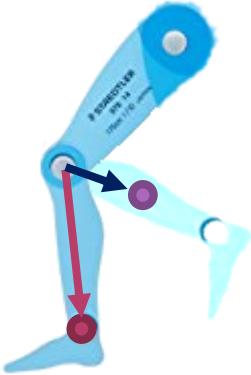
こんなイメージで作っていきます。つまり回転は全て X 軸を中心とします。2D の解説は既にやっていますので、それを元に解説していきます。

まず、膝を曲げることを想定します。



例えば足首の IK を図のように移動した場合、膝を中心にスネを回すわけですが、間の角度を計算して、その差分だけ回転させてあげれば良いわけです。

まずそこを考えましょう。先程の XMVector3AngleBetweenNormals には 2 つのベクトルがあればいいので、



図の2つのベクトルがわかれればいいですね?

そういうことで

- ひざから現在のIK座標ベクトル
- ひざから目的のIK座標ベクトル

を作ります。

### 27.5.3 CCD-IK の実装

```
XMFFLOAT3 originVec = ikOriginPos - bone.headpos;//IKとさかのぼりノードでベクトル作成
XMFFLOAT3 targetVec = ikTargetPos - bone.headpos;//目標とさかのぼりノードでベクトル作成
```

こんな感じで。

ちなみに PMD の IKList が持っている ik インデックスは末端から根っこに向かってインデックスが配置されてるので、別に逆からとかにしなくていい。

つまり

```
for (int i = 0; i < iklist.ikchainLen; ++i)
```

でいいわけです。さらに前述の2つのベクトルから軸をつくりましょう。外積を取るわけですが、事前に正規化しておきましょう(正規化する必要はないですが、一応)

//正規化します

```
originVec = Normalize(originVec);
targetVec = Normalize(targetVec);
```

//外積から軸を作成します

```

XMFLOAT3 axis = Normalize(Cross(originVec, targetVec));

はい、これで軸が出来上がるわけですが、

//もしひざ系なら、X軸を回転軸とする
if (bone.name.find("ひざ") != std::string::npos){
    axis.x = -1;
    axis.y = 0;
    axis.z = 0;
} else{
    if (Length(axis) == 0.0f){
        return; //外積結果が0になってるなら使えません
    }
}

```

とりあえず「ひざ」だけは強制的に X 軸なので、上のコードのようにボーン名を確認して「ひざ」という文字列が含まれていれば、強制的に X 軸の回転とします。

文字列に特定の文字列が含まれているかどうかを確認するためには C 言語では strstr という関数を使いましたが、C++では find で検索できます。

find の戻り値は「見つかった場所」を数値で返してきます。

なお、見つからなかった場合は std::string::npos を返しますので、これ以外なら「ひざ」という文字列があったという証拠になります。

ただし上のコードでは、axis.x=-1 としていますが、これは PMD が右手系で、デフォルトでお尻向けているため、左手系に合わせるために-1 にしています。

さて、ここまでいれりでしようか？

ここまでできたら、「軸」が決定していますので、あとは間の角度です。間の角度は

XMVector3AngleBetweenNormals

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.geometric.xmvector3anglebetweennormals\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweennormals(v=vs.85).aspx)

か

XMVector3AngleBetweenVectors

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.geometric.xmvector3anglebetweenvectors\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3anglebetweenvectors(v=vs.85).aspx)

もしくは自作の角度計算関数を使いましょう。

二つのベクトルの内積= $\cos\theta$ なので、 $\arccos(\text{内積})$ で間の角度は求められます。

でも面倒なので出来合いの関数を使ったほうが楽だし、多分高速だし、今回は  
XMVectorBetweenNormals を使用しましょう。

以前にも書きましたが、この関数は無駄に XMVECTOR 型で返してきますので、こう書いてもいい  
いですね。

```
//ふたつのベクトルの間の角度を計算(制限角度演算のため)  
float angle = XMVector3AngleBetweenNormals(XMLoadFloat3(&originVec), XMLoadFloat3(&targetVec)).m128_f32(0);
```

で、角度が出ましたので『そのための回転行列』を作ります。



そのための回転行列を作るためには、軸と回転角度が必要ですが、それはすでに分かっています。これのために使用できる関数は2種類です。

XMMatrixRotationAxis

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

か、もしくは

XMMatrixRotationQuaternion

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

です。クオータニオンはよくわかつてないし、ひとまずはまだわかる XMMatrixRotationAxis を使います。

これは特定の軸中心に回転する行列を作ります。

数式にすると

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} n_x^2(1-\cos\theta) + \cos\theta & n_x n_y(1-\cos\theta) - n_z \sin\theta & n_x n_z(1-\cos\theta) + n_y \sin\theta & 0 \\ n_x n_y(1-\cos\theta) + n_z \sin\theta & n_y^2(1-\cos\theta) + \cos\theta & n_y n_z(1-\cos\theta) - n_x \sin\theta & 0 \\ n_x n_z(1-\cos\theta) - n_y \sin\theta & n_y n_z(1-\cos\theta) + n_x \sin\theta & n_z^2(1-\cos\theta) + \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

こういう感じで既にわけわからぬですが、ともかくこれで  $(n_x, n_y, n_z)$  ベクトル中心に回る回転行列が作れるわけです。

ついでに言うと、クオータニオンのほうは

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) & 0 \\ 2(q_1 q_2 + q_0 q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2 q_3 - q_0 q_1) & 0 \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

こういう式です。q がクオータニオンなので、わけわからぬ! ですね!!

ともかく回転させます。

```
XMMATRIX rotMat = XMMatrixRotationAxis(XMLoadFloat3(&axis), angle);
```

で、ここで得られた rotMat という行列が「回転そのもの」を表していますが、これをボーン行列に乗算すればいい! … そう考えていませんか? そう考えていた時期が僕にもありました。

当然のことですが、回転はそのままだと「原点中心回転」になります。というわけで例によって  
 「回転中心を原点に移動する行列×原点中心回転行列×元の座標に戻す行列」  
 を作る必要がありますね？そういうことです。作ります。  
 どこを中心にしていいのかというと、ボーンの起点ですよね？今回はボーンの headpos がそれに当たりますから

//オフセットを考慮した行列を作る(原点に移動→回転→元の座標)

```
XMMATRIX mat = XMMatrixTranslation(-bone.headpos.x, -bone.headpos.y, -bone.headpos.z) *  

    rotMat *  

    XMMatrixTranslation(bone.headpos.x, bone.headpos.y, bone.headpos.z);
```

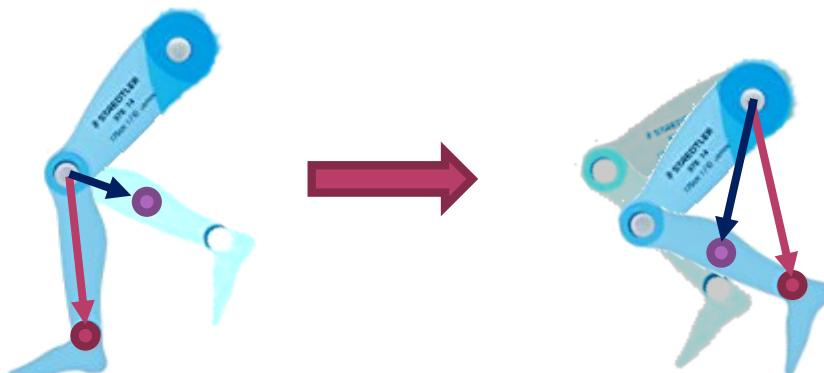
こういうことですね。横着して rotMat=ってやるとあとで痛い目を見ますので、もったいなし  
 ようですが、新規でマトリクスを作つてください。

あとはそれをもとのボーン行列にかけてあげればよい

//変換行列を計算(オフセットを考慮)

```
mesh.BoneMatrixes()(ikboneIdx) = mesh.BoneMatrixes()(ikboneIdx) * mat;
```

まあ、サイクリック(繰り返し)をしないなら、ここまで話で終わりなのですが、CCD-IK なので、  
 当然繰り返しします。



つまりサイクリック中のコントロールポイント(ボーン起点)を CCD-IK の間保持しておく必  
 要があります。

だったら bone.headpos を動かせばいいと思うかもしれません、最終的には IK→FK にしてポーズを決めるため、bone の場所は動かしたくないです。

なので一時変数を使用します。

```
//まずはIKの間にあるボーンの座標の一時変数配列を作つて、値をコピーする
//理由はIK再帰する毎にボーン座標が変更されるからです(元の座標は必要なので一次変数に格納)
std::vector<XMFLOAT3> tmpBonePositions(iklist.ikchainLen);
for (int i = 0; i < iklist.ikchainLen; ++i){
    tmpBonePositions[i]=mesh.Bones()( iklist.ikboneIndices[i]).headpos;
}
```

さて、そうなるといちいちベクトルも更新しなければならないため

```
XMFLOAT3 originVec = ikOriginPos - bone.headpos;//IKとさかのぼりノードでベクトル作成
XMFLOAT3 targetVec = ikTargetPos - bone.headpos;//目標とさかのぼりノードでベクトル作成
```

の箇所は

```
XMFLOAT3 originVec = ikOriginPos - tmpBonePositions(i);//もとの先っちょIKとさかのぼりノードでベクトル作成
XMFLOAT3 targetVec = ikTargetPos - tmpBonePositions(i);//目標地点とさかのぼりノードでベクトルを作成
```

となるでしょう。

あとはこれを繰り返せば良いわけです。

#### 27.5.4呼び出し側

また、忘れてならないのが呼び出し側ですが、

```
std::string name = "右足IK";
CcdIkSolve(*_mesh, name, XMFLOAT3(-0.0, _iky, 0));
RecursiveApplyBones(_mesh->BoneNodes(), 0, _mesh->Bones(), _mesh->BoneMatrixes());
```

CCDIK やったあとに Recursive 処理は忘れないようにしてください。

## 27.5.5 角度制限

角度制限をつけなければ、足の関節があらぬ方向へぶつ飛びます。

というわけで、角度制限をつけたいのですが、仕様が良くわからぬので PMDEditor の `readme.txt` を見てみます。すると…

### ●[IK]

IK リスト : IK ボーンとして機能するボーンの一覧。

→

IK(数値) : 対応する IK ボーン Index

Target(数値) : IK ボーンの位置にこのボーンを一致させるように IK 処理が行われる

IK ループ回数(整数) : IK 処理での計算回数(最大 255)

**単位制限角(実数)** : 一回の IK 計算での制限角度(数値は rad 値の模様? |  $1.0=4[\text{rad}]$ (230 度程度) 180 度:  $0.7854 = 3.141592/4$ )

影響下ボーンリスト : IK の影響下にあるボーン一覧 | IK 接続先に近い方からリスト順にする必要がある

と、書かれてました。なんだこの…うーんこの…。数値が rad 値とは書いてるが、どうやらそうではないらしい。

つまり制限角度に書かれている数値を 4 倍したものがラジアン(弧度法)になるようだ…。ややこしい。

ちなみに足 IK の制限角度は 0.5となっていたので、ラジアンは 2 くらい…だいたい  $144^\circ$  くらい。PMDEditor で見てもそうなのだからそうなのだろう。

そして疑問に思うのが、なぜ 4 分の 1 にまでしたんだろうってこと。別に容量的にも同じだし実際に使用する際も角度ラジアンをそのまま渡したほうがいいはずだ。

いいはずだが…何でなんですかね?まあいいや、ともかく進みましょう。

```
float strict = iklist.limitAngle * 4; // 制限角度は持ってきた角度の4倍
```

```
//それ以上に曲げられないようにしつく  
angle = min(angle, strict);  
angle = max(angle, -strict);
```

それでもなんか荒ぶるなあ…PMDEditor の readme.txt をもっと読んでみる。

○deg ボタン

単位制限角を角度で入力／**有効範囲は 0-180 度程度**(あくまで解析情報からの推測値となります)

などと書いてある。

…これも適用してみたが、結局荒ぶるため、各実装系を参考にしてみた所いくつか共通点がありました。

制限角度は一つ一つの制限ではなく、合計の回転角度制限(サイクリック1回あたり)ということのようです。

例えば

```
//で?なんかcosから角度を出して2で割って...  
Rot = 0.5f * _ACOS( Cos );  
//コントロールウェイト(演算一回の制限角度)*(j+1)*2よりもRotが大きいのなら  
//その制限角度にしてしまう。  
if( Rot > IKInfo->ControlWeight * ( j + 1 ) * 2 )  
    Rot = IKInfo->ControlWeight * ( j + 1 ) * 2 ;
```

というコードや

```
maxangle = ( i + 1 ) * ik.control_weight * 4;  
  
theta = Math.asin(sinTheta);  
  
if ( vec3.dot(targetVec, ikboneVec) < 0) {  
  
    theta = 3.141592653589793 - theta;  
}
```

```

if (theta > maxangle) {

    theta = maxangle;

}

q = quat4.set(vec3.scale(axis, Math.sin(theta / 2) / axisLen), tmpQ);

q[3] = Math.cos(theta / 2);

```

というコードを見つけました。そもそも本家が実装系の説明全くしてないもんなあ…。手探りなんだよなあ。それでも共通点は

- 「2で割っていること」
- 「制限角度を出てきた角度の4倍していること」

となっています。

そして以下のHPの説明を読んでみる…相当難しいけど

<http://d.hatena.ne.jp/edvakf/20111102/1320268602>

「極北Pさんの PMDEditor の Readme (一番下に引用した)の説明によると、指定された値×4 radian までしか曲がれないらしい。なので、上で $\theta$ を求めたときに、この単位制限角との max を取る必要がある。」

もう一つのポイントは、「ひざ」という名の付くボーンは X 軸方向にしか動かないというもの。(PMX だと好きなボーンに角度制限が付けられるらしいが)

これをやるにはちょっと面倒なことをしないといけない。なぜなら、先ほど求め

た  $\tilde{R}_0'$  などは、いわば「修正『合成』回転」になるため、そこから「修正『個別』回

転」 $R_0'$  を求め、これに角度制限をかけ、さらにそれを合成回転に直さないといけ  
ない。(回転軸制限は個別回転に適用されるもので、合成回転にではない)」

原文ママ

何言ってるか良くわからない。

でも、個別角度は求まっているから、ここに書かれているようなややこしいことはしなくていいんじゃないかな。

個別が求まっているから(i+1)をかけるのはちょっと置いておいて、注目してみたのは「2で割っている」という部分。

半分にしてみよう…。

```
angle *= 0.5;
```

さて…どうなるかな。



…なん……だと？

うーーーーーーーーーーーん!!!!この!!!何なんですかねー!!なんでうまくいくんですかねー!?

まあ喜ぶのはまだ早い。

足首が本当にIKの解の座標になっているのかを確認しましょう。普通に考えると、サイクリックしなければ本来の座標の半分くらいの位置にしか来ないはずです。

#### 27.5.6 確認コード

現在のIKの座標を画面上で確認できるようにしてみましょう。



画像のようなマーカーを表示させましょう。

指定された足のIK座標と、実際の足首が乖離していないか確認しましょう。

「ビルボード」でやっても良いんですが、それだとオブジェクトに隠れてしまいます。最前面に表示させたい時はHUDと同様に、2Dとして表示したいと思います。MMDのマーカーもそうですが、視点からの距離に関係なく同じ大きさで表示されています。

このことから、あのマーカーはビルボードというよりHUDの一種だと思います。

じゃあどうやって実装しますか？

そろそろ自分で考えていただきたい。大抵の教えられることは今まで教えてきました。この程度の事であれば自分で考えて、もうそろそろセンターの手を離れるべき時です。

ヒント…

- シェーダはHUD用のを使用すると楽
- 4点クアッドポリゴンを使用
- 点(座標)に対してモデルのWVPを乗算した座標を中心に2Dポリゴンを表示
- 最後に深度を無効にして表示

たまには自分で考えて実装しようぜ。

ちなみに



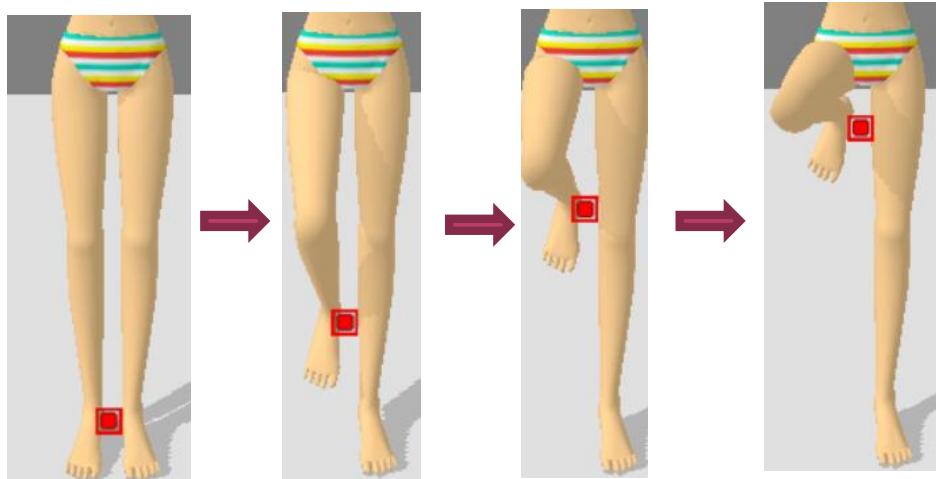
こういう画像を.ddsとしてサーバーにアップしているのでよければ使ってみてください。

"dds"という拡張子は駒染みがないかもしれません、DirectX用の画像フォーマットです。

ddsが分からなければ自分でマーカーを作って構いません。フツーにテクスチャとしてロードすれば良いんですけどね？

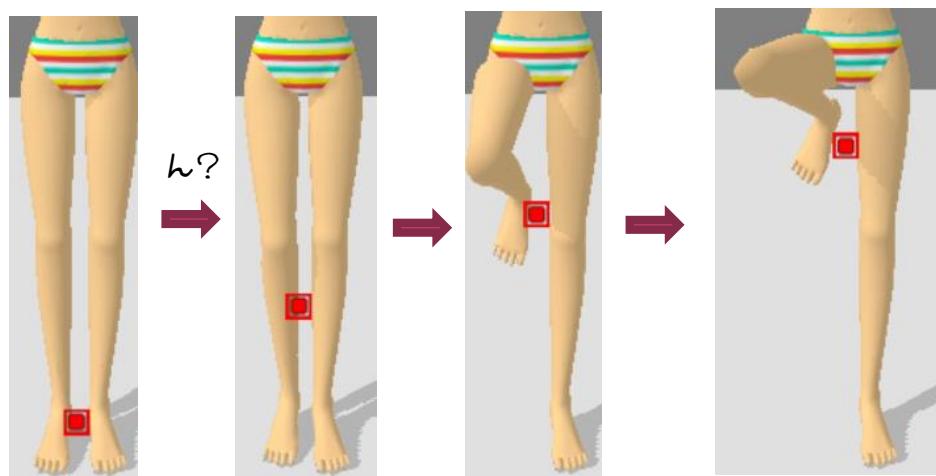
ともかく、実装してみて確認してみました。

右足のIKにマーカーを合わせています。



一応、マーカーに合わせて足が“上が”っていっているのが分かると思います。

それではサイクリックなしと比較してみましょう。



一目瞭然ですが、特に二番目を見るとおかしなことが分かりますね？マーカーが膝に来るまでは全然動かさずに、そこを超えるとIKとして追従します。

というわけで、ソリューションとしては、制限角度をつけたうえで2で割るということのようです。

どこかに明確な使用があればいいんですけどね。ホント手探りですわー。

今回のマーカーみたいに、拳動を一目で比較できる「デバッグツール」自分で考えておいたほうがいいでしょう。

### 27.5.7 仕上げ(軸の補正)

足を斜めにした時に応するコードを書きましょう。

以前も書きましたが、このままでは足を開いた時に、足の角度が不適切になります。IKについていきません。



これに対応するためにはループに入る前に最根っこボーンと IK(元の場所、移動先)のベクトルについて外積を取り、回転を計算します。

```
//ボーンの根っこ部分(IKから最も遠いボーン)からIKの元の座標へのベクトルを作つておく(軸作成用)  
XMFLOAT3 ikOriginRootVec = ikOriginPos - tmpBonePositions(iklist.ikchainLen - 1);
```

```
//ボーンの根っこ部分(IKから最も遠いボーン)から移動後IK座標へのベクトルを作つておく(軸作成用)  
XMFLOAT3 ikTargetRootVec = ikTargetPos - tmpBonePositions(iklist.ikchainLen - 1);
```

この2つのベクトルから回転行列を計算しましょう。

```
XMMATRIX matIkRot = LookAtMatrix(Normalize(ikOriginRootVec), Normalize(ikTargetRootVec),  
XMFLOAT3(0, 1, 0), XMFLOAT3(1, 0, 0));
```

とやってもいいし、

2つのベクトルの軸と回転を取得した上でその回転行列を求めて構いません。

```
rootAxis=Cross(Normalize(ikOriginRootVec), Normalize(ikTargetRootVec));  
rootAngle=XMVector3BetweenNormals(ikoriginrootvec, iktargetrootvec);
```

```
matIkRot=XMMatrixAxisAngle(rootAxis,rootAngle);
```

こんな感じ。

いかがですやろ？

これで得られた軸と角度で、『ひざ X 軸』に補正を行ってやる。

そうすると



ごらんのように、完ぺきではないですが、きちんと IK に追従するようになります。

#### 27.5.8 仕上げ(ベクトル最大長による IK 位置の補正)

足の長さは変わらないので、IK の場所を足の長さを超えるほど長くならないようにしましょう。

大した話ではないです。所謂『CLAMP』すればいいのです。



こうですか？

ちがいます。

これですか？



ちがいます。

値に上限と下限を設けてやればいいのです。XNAMathには便利な関数があって

XMVector3ClampLength(ベクトル, 最小長, 最大長)

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.geometric.xmvector3clamplength\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3clamplength(v=vs.85).aspx)

という関数があります。

その名の通り、ベクトルをクランプした結果を返してくれます。

ターゲットを動かすときに、元のオリジナルの長さ(膝が曲がってない)を上限としていれば、それより遠くに行くことはありません。

例えば

```
float ikmaxLen = Length(ikOriginRootVec);
```

とでも設定しておいて

```
vec=XMVector3ClampLength(vec, 0.1, ikmaxLen);
```

とでも書いてあげればいいのです。

もちろん、こいつはベクトルなので、

```
XMStoreFloat3(&ikTargetRootVec, vec);
```

と書いてやれば、元の float3 型変数になります。ただしこいつはベクトルであって、座標でないため、このベクトルを作ったもとの座標を足してあげる必要がありますので、注意してください。

```
ikTargetPos = ikTargetRootVec + tmpBonePositions(iklist.ikchainLen - 1);
```

とでもやれば、補正後の IK ターゲット座標が得られます。

まだまだ問題が残っていますが、結構きりがないし、僕にもなんとかわからぬリバグが多いので、とりあえず進行を優先して、いったん IK は切り上げます。力不足ですみません。

## 28 ビルボード

解説したと思ってたら解説してなかつたでござる。

「ビルボード」ってのは「常にこちらを向いている板」のことである。ボードってのは板のことだからね。

さて、どのようなときに使われるかというと幾つかの用途があつて

- 3D 空間上に 2D のキャラクターなどを表示したい(2.5D 的な表示)
- パーティクル的なやつを表示したい(爆発エフェクトなど)
- 樹木など「真面目にモデリングするとポリゴン大量地獄」になりそうなもの
- HUD の代わり

などがあります。

3D の世界の中で 2D のキャラクターが動いているゲームも見たことはあると思う。2D でアニメーションパターンを作っておいて、3D のゲーム中に表示する。しかし、この場合、カメラの向きによってはペラペラの部分が見えてしまう。

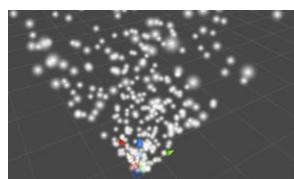


この画面上のキノコ野郎が 2D でできていることが分かるだろう?

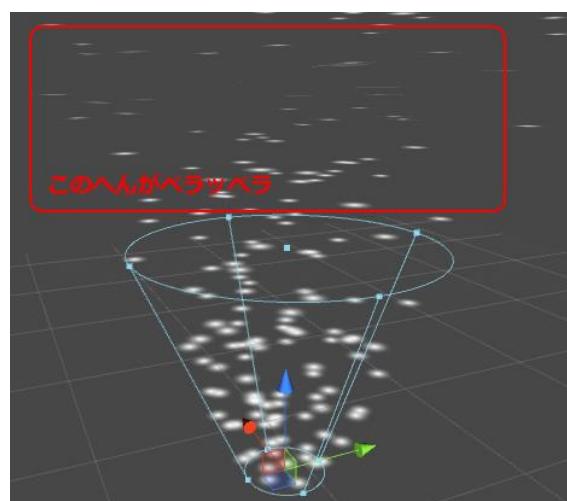
このゲームの場合は、これでいいのだ。何しろ「ペーパーマリオ」なのだから。コレデヨイ。

ただし、たいていの場合であれば、2D であっても 3D 空間内にいる以上は 3D に同化してほしい。

例えば Unity のパーティクルはデフォルトで「ビルボード」になっているため



このように同じ 2D 画像が大量に発生するが常にこちらを向いているため、特に違和感がないが、これの「ビルボード」を敢えて無効にしてしまうとこうなる。



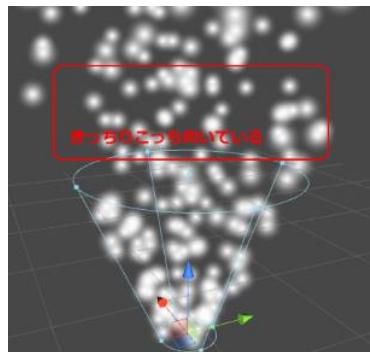
これではペラッペラな画像ってことがバレてしまいます。

どうにかしてこれを 2D オブジェクトとばれない方法はないだろうか?

そうですね。画像の法線方向が常にこちらを向いていればいいんです。

そうなると LookAt が使えそうですが…。

常にこちらを向くようすれば…



そもそもどうしてこちらを向かないことになってるんでしょうか?

それはカメラが $\vec{z}$ 方向を向いてないときにこうなるのです。

考えてください。

カメラが必ず $(0,0,1)$ の方向を向いていれば、法線ベクトル $(0,0,-1)$ の板ポリゴンは必ず真正面を向いているはずです。

つまり板ポリにとてのカメラの「回転」を無効にしてやればいいんです。

どうすればいいんでしょうか?

今、カメラ行列を持っていますよね?

ビューと、プロジェクション。

このビューのほうを使います。ビュー行列はどうやって作ってましたつけ?

`XMMatrixLookAtLH`

が何かで作ってました。こいつはカメラの位置と、ターゲットの位置から「カメラの回転とカメラの平行移動を合成した行列」を作っています。

で、ペラペラの物体がペラペラに見える原因を作っているのは「回転成分」です。

ちなみに、「カメラの回転」というのは、実際にカメラが回転しているのではなく、カメラに合わせて世界を回転させています。

ということはビルボードのカメラ回転を無効にしてやればいい。そうなります。

ここで数学としての行列のおさらいですけど、思い出して欲しい法則が3つあります

- 単位行列は乗算してもなにも変化しません
- 行列に、その行列の逆行列をかけると単位行列になります
- 回転行列は転置させると逆行列になります

賢明な学生さんは既に気がついていると思いますが、結論から言うとモデルに対してカメラの行列の回転成分だけを転置させた「カメラ回転逆行列」を予め乗算しておけば、カメラ回転を無効にできます。

これが一般的なビルボードです。

## 28.1 数学的な話

ああー、数学の話は必要なんじゃあああ!!!

すまんねえー!!!

カメラ回転を無効にするために必要なんじゃよ。

で、今回は行列をいちいち

$$M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

なんて書き方してたら逆に分かりにくいと思いまして、↑のMのようにアルファベットの大文字で行列を表そうと思います。

ここで行列の性質と、アルファベットの意味について最初に定義しておきます。

- 単位行列は  $I$  とし、行列  $I$  は乗算をしても結果が変化しない ( $IM = MI = M$ )
- 行列は逆行列を持っており(ないこともある)行列を $-1$ 乗したものとして表す(例: $M^{-1}$ )
- 逆行列ともとの行列を乗算すると単位行列になる ( $M \times M^{-1} = M^{-1} \times M = I$ )
- 今回は回転行列を  $R$  とし、平行移動を  $T$ 、ビュー(カメラ)を  $V$ 、投影行列を  $P$  とします
- カメラ回転を  $R_v$  とし、カメラ平行移動を  $T_v$  とします。
- 転置行列(もとの行列の行と列を入れ替えたもの)は  $T$  乗として表します(例: $M^T$ )
- 回転だけの場合、転置(行列の行と列を入れ替える)すると、逆行列になる
- モデル自身の平行移動や回転は  $W$  として表します

以上のことを見てカメラの「回転」を無効にしてみましょう。

通常であればモデル自身の頂点には  $WVP$  がかけられることで、画面に表示されます。つまり

$$(x, y, z) \times W \times V \times P = (x', y', z')$$

こういうことですね?

さて、ここから何かしらの細工を施したいと思います。 $V$  と  $P$  は既に決定されていて扱えない  
と仮定します。細工できるのは  $W$  だけだとします。

ここで

$$V = R_v \times T_v$$

というふうに考えます。そうすると、左から  $R_v^{-1}$  をかければ良いことは分かるでしょうか?

$$R_v^{-1} \times V = R_v^{-1} R_v T_v = T_v$$

となり、平行移動成分だけが残ります。

つまり、 $W$  と  $VP$  を乗算する前に  $W$  に対して  $R_v^{-1}$  を予め乗算しておけば、カメラによる回転だけ  
が無効になった合成行列を作ることができるわけです。

## 28.2 実装

実装の前にテストしやすい環境を作りましょう。

### 28.2.1 カメラを動かせるようにしよう

現在、実はカメラを動かしていません。モデル表示に躍起になって基本的な所をすっ飛ばしていましたね。

今の僕のプログラムだとカメラクラスがあって、それを移動したり回転させたりすれば自動的にカメラの動きが反映されるようになっています。そうなつていねい人は少なくともカメラの位置と向きがわかるような変数を作ってください。

多分、座標(視点)をいじるのはそれほど難しくないと思います。つまり平行移動は楽です。  
WASDで動かしてみましょう。

動かしてみればわかりますが、なんかオカシイですよね？

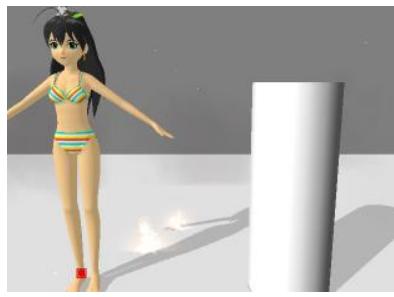


ん？

ああ、そうか、注視点がそのままだからな。じゃあ平行移動時に注視点も同じくらい動かしてみよう。

```
if(↑押した){  
    eye+=front;  
    target+=front;  
}
```

ひとまずノーヒントで。そんなに難しくないので、分からねーって人は自分でノート等に図を書いてたりして考えてください。



カメラを右に動かした時にこういう感じに動いてれば概ねオッケー

でも回転は?

そう、そこが悩みどころなのだ。

回転した場合は target も『視点から見て』回転させねばならない。

手順としては

1. 初期化の段階で target-eye で『視線ベクトル』を作る
2. 視線ベクトルを回転させる
3. target=eye+視線ベクトル

とする

これで作れるはずです。

ちょっと僕は都合により CTRL とマウス左右で視点が変わるようにします。左右だから Y 軸回転ですね。

```
//カメラを回転する
///@param pitch Z軸回り回転
///@param yaw Y軸回り回転
///@param roll Z軸回り回転
void Rotate(float pitch, float yaw, float roll);
```

こういう関数作ります。ヨーピッチロールはゲームではよく使用される用語なのでこの機会に触れておきましょう。

<https://ja.wikipedia.org/wiki/%E3%83%AD%E3%83%BC%E3%83%AA%E3%83%B3%E3%82%B0>

面倒な概念ですね。航空のところから来たんでしょうけど、結構関数名とかに使われてるんでホント面倒です。

頑張りましょう。

ここで使用するであろう関数は

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

XMVector3Transform [https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.transformation.xmvector3transform\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.transformation.xmvector3transform(v=vs.85).aspx)

ですね。

作っていくときに、前に作ったベクトル関連のオペレータを使いたいので

Geometry.h

というファイルを作って、その中に前に作った

```
///足し算引き算
XMFLOAT3 operator+(XMFLOAT3& lval, XMFLOAT3& rval);
XMFLOAT3 operator-(XMFLOAT3& lval, XMFLOAT3& rval);
```

///ベクトル比較

```
bool operator==(XMFLOAT3& lval, XMFLOAT3& rval);
```

///ベクトル長

```
float Length(XMFLOAT3& vec);
```

///内積

```
float Dot(XMFLOAT3& lval, XMFLOAT3& rval);
```

///外積

```
XMFLOAT3 Cross(XMFLOAT3& lval, XMFLOAT3& rval);
```

を宣言しておきます。

そこまでできれば、多分楽に TPS 的に動かすことができるでしょう。がんばってください。

### 28.2.2 カメラを動かせるようになってついでに

いまの段階では、カメラが何処を向いていようと、ワールド空間における XZ 方向を動く状態です。

ですが、TPS 的には「今向いている方向」に進んでほしいですよね？

つまり…X 方向にいくつ、Z 方向にいくつっていうより、front のプラスマイナスであり、right のプラスマイナスと考えたほうが良いでしょう。

さて、front ってのは「前」ですね。向いている方向の「前」ってどっち？

そう、視線(の Y 成分を削除した部分)ベクトル(つまり target マイナス eye)が front 方向ですね？

ひとまず前後に進めるようにしてみましょう。Move ではなく MoveTPS って関数を作りましょう。

前後に進めるようになったら、左右なんですが、簡単です。front を右に 90° 回したやつが「右」ですから。

「90° 回す」ってのはサインコサインを使うまでもなく簡単で、

- X 成分と Z 成分を入れ替える
- X 成分か Z 成分の符号を入れ替える

の 2 つで実現できます。で、二番目のやつが悩ましい所だとは思いますが、頭のなかで図を描いてください(実際にノートに図を書いてもいいですが)

Z を右にするには Z=X であるため、

$$(x, z) \rightarrow (z, ?)$$

です。普通に考えて右は後ろになるため、右にはマイナスがかかります。つまり

$$(x, 0, z) \rightarrow (z, 0, -x)$$

となります。

あとは自分が向いている方向の方向ベクトルを正規化したものを作ります。

rayfront

これを右に 90° 回転させたものを

rayright

もちろん正規化しといてください。

あとは rayfront ベクトルを「前方」倍して、rayright ベクトルを「右」倍して足せばいいわけです。

これで TPS みたいに動くようになったと思います。

### 28.2.3 ビルボード

さて、かなり引き伸ばしましたが、ビルボードです。忘れてたでしょ？

ひとまずはクアッドポリゴンを作ってください。テクスチャを貼り付けられる状態ですね。

んで、いつもどおりな感じで適当な場所にそのクアッドポリゴンを置いてください。

で、最後に WVP 行列を乗算する時に W に先程の「回転逆行列」を乗算してください。

あと、ここで注意点ですがビルボードの頂点は原点を中心に行っておいてください。何故かと言うと、ビルボード頂点が原点からズレていると、回転の軸も当然ずれるからです。

対象のビルボードを原点から違う位置に起きたければ頂点生成後に平行移動行列を乗算することで、移動させておいてください。

ひとまずはテクスチャと頂点バッファを作つて…

```
//ビルボードテストのためのテクスチャと頂点バッファ  
ID3D11ShaderResourceView* _billboardSRV = nullptr;  
ID3D11Buffer* _billboardBuffer = nullptr;
```

中身作つて

```
//ビルボードのテクスチャと表示用頂点を作る  
result = D3DX11CreateShaderResourceViewFromFile(device.Device(), "eyeball.png", nullptr,  
nullptr, &_billboardSRV, nullptr);  
_billboardBuffer = CreateBillboardVertex(0,0,0,5,5);
```

ちょっとそれ用頂点シェーダ作つて

```
CreateBillboardShader(billboardvs, billboardlayout);
```

あとは表示前に回転打ち消し行列を作るだけです。

```
XMMATRIX view=_camera->ViewMatrix();
```

ビュー行列を持ってきて

平行移動成分が邪魔なので消しといて

```
view._41 = view._42 = view._43 = 0;
```

転置して

```
view = XMMatrixTranspose(view);
```

ワールドに乗算することでカメラ行列の回転成分が打ち消されます。

```
world = world*view;
```

はい、ここまでやつたワールド行列を用いて座標変換を行えば、物体はかならずこっちを向いている。

向いているのだが…

なんかおかしい。

動かすと分かるんだが…

ちょっと離れたところで、ビルボードとキャラが見える状態でカメラを左右に振ってみて欲しい。



なんか位置関係がおかしくならないだろうか？ビルボードが変な動きをしないだろうか？

理由を言うと…

```
world = world*view;
```

この部分。

今回の目的は何だった？

「カメラの回転成分“のみ”」を打ち消すんだったよね？つまり座標が回転の影響を受けてはいけないんだ。

しかし平行移動していれば World の中に「平行移動成分」があり、それがカメラ回転逆行列の影響を受けて、カメラを振るとおかしな動きをするようになってしまう。

手順としては

1. ワールドの平行移動成分を退避
2. ワールドの平行移動成分を0にする
3. ワールドとカメラ逆回転行列を乗算(向きの回転だけが行われる)
4. ワールドに平行移動成分をそっと戻す

つまり、カメラ行列に対して平行移動成分を打ち消すため 41～43 を 0 にしたのと同様にワールドの平行移動成分も 0 にする。

```
world._41 = world._42 = world._43 = 0;
```

ただし、最終的に平行移動は有効にして置かなければならぬため、事前に変数等に退避させておく。

```
XMFLOAT3 trans = { wac.world._41, wac.world._42, wac.world._43 };
```

で、いつもどおり逆回転行列をかけたら、41～43に平行移動の数値を返してやる。

```
wac.world = wac.world*view;  
wac.world._41 = trans.x;  
wac.world._42 = trans.y;  
wac.world._43 = trans.z;
```

これにより、よりビルボードらしいビルボード（座標情報はそのままで常にこちらを向いている）を実装できます。



#### 28.2.4 ビルボードクラス

うん、まあ、クラス化していきまひよか？

量産できるよう意識しつつな。

こういうなあ、クラスを作るときってのは本来はじっくり考えたほうが良い。

どういう役割と責任を持つクラスにしたいのか、このクラスに何をさせたいのか、どこからどこまでをサポートするのか…

使う側が何を考えて使うのか、ふさわしくない使い方まで考えて…本来はなあ



こういう気持ちで作るのが理想である。

自分で考えて、時と場合にふさわしいクラスを作れるようになって欲しい。

今は荒削りでも良い、自分で考えられるようタクマシク育って欲しい。C++の授業も受けてるし、自分で設計できる知識はあるはずなんやで!!!

こういうクラス設計も含めたプログラミングで言えるのは、何度も言つてることだけだ、

**正解なんかない！**

だから…

**自分の頭で考えろ!!**

正解なんかないので、作った結果「あれー？」ってなつたら、修正するとか作り直せばいい。それだけだ。

知識は大事です。知ることは大事です。

でもそれだけではプログラミングはできないのです。センスがあればいいけど、ない場合はクソほど失敗して身に付けなければアカンのです。



さて、ビルボードクラスを作っていくための僕の場合のロードマップを例として書きます

1. まずプロトタイプとしてクラス化する前に簡単にビルボードを実装する
2. 必要なものがだいたい分かるので、それをメンバに持つクラスの形を考える
3. ビルボードに必要な変数や初期化関数をクラス側に移動
4. オブジェクト生成→基本設定すれば Draw 関数でビルボードが表示されるようにする
5. 量産のことを考える
6. リファクタリングする

まあ、①は終わってるんでひとまずは④までちゃっちゃと作ります。

必要なものは…

- テクスチャ
- 頂点バッファ
- 配置のための『座標』
- カメラクラスへの参照
- 頂点シェーダ
- ピクセルシェーダ
- 頂点レイアウト

である。『量産』の事を考えると頂点シェーダやレイアウトやピクセルシェーダに関しては他オブジェクトと共に用する必要があるがこの段階では考えない。

次にオブジェクト初期化時もしくは Init 関数等でやっておくべきことは

- トポロジの設定
- 頂点バッファの生成
- テクスチャのロード
- 各シェーダのロード
- 設置座標の設定

Draw 関数でやるべきことは

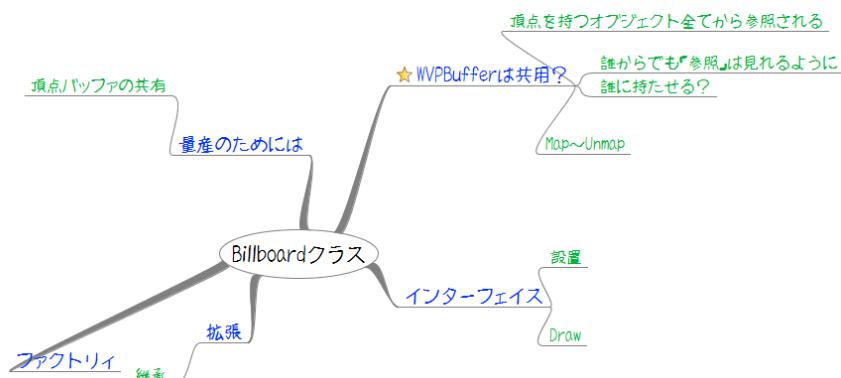
- セットすべき行列の更新
- シェーダのセット
- テクスチャのセット
- 頂点のセット
- レイアウトのセット
- トポロジのセット
- 描画

になりますかなあ…。

頑張りましょう。

がんばってください。ここまで提示して作れないはずはないと思います。

ていうか、僕自身も「正解」なんて知らんから、こう…



こういうのを書きながら行ったり来たりしてるわけですよ。

「人狼をチ殺す銀の弾丸などない!」

覚えとけ

これはその昔、フレデリック・P・ブルックスって人の論文で書かれた言葉であり、まあ一言で言うと「ソフトウェア開発に正解などない!」「オブジェクト指向だのスクラムだのあるけど、それで全てが解決することなんてアリエナイんだぜ!!!」っていう論文です。

ダイエットと一緒にですな。

体重測るのが良いとか、カロリー計算が良いとか、炭水化物がアカンとか色々あるけど、体质によっても対処は違うのと一緒に、プロジェクトによって最適な答えは違う。答えは現場の中で頑張って生み出さなければならないのだ。

その論文に興味ある人は

<https://ja.wikipedia.org/wiki/%E9%8A%80%E3%81%AE%E5%BC%BE%E3%81%AA%E3%81%A9%E3%81%AA%E3%81%84>

を見るか、図書館で

### 「人月の神話」

を探して読んで下さい。一度ウィキペくらいは読んだほうが良いと思いますよ。ゲーム業界を目指すにせよ、目指さないにせよ…

ああ、いかんいかん。また脱線仕掛けた。まずは共用のことは考えずに単品で動くようにしてみるのだから、

ちなみにさつき見せた変な図は「マインドマップ」といって、アイディア出しの支援ツールです。

<https://ja.wikipedia.org/wiki/%E3%83%9E%E3%82%A4%E3%83%B3%E3%83%89%E3%83%9E%E3%83%83%E3%83%97>

もし「求職票」とか「履歴書」を書く時に何もネタが見つからない時は、ここに手当たり次第にノードを追加していきます。

ホントに手当たり次第です。

ちなみにPC上で使えるツールとして「FreeMind」というフリーソフトがありますので、くわしくは

<https://ja.osdn.net/projects/freemind/>

からダウンロードして使ってみてください。

ちなみに一つ言うと

```
struct BillboardVertex{
    XMFFLOAT3 pos;//3D座標
    XMFFLOAT2 uv;//UV座標
};

struct WorldAndCamera{
    XMMATRIX world;
    XMMATRIX camera;
    XMMATRIX lightview;
};

ID3D11Buffer* _wvpbuffer;//コンスタントバッファ
ID3D11Buffer* _vbuffer;//頂点バッファ
ID3D11InputLayout* _layout;//頂点レイアウト
ID3D11VertexShader* _vs;//頂点シェーダ
ID3D11PixelShader* _ps;//ピクセルシェーダ
ID3D11ShaderResourceView* _texture;//テクスチャ
Camera& _camera;//カメラクラスへの参照(必須)
XMFFLOAT3 _pos;//現在のビルボードの座標
```

を内部に持たせました。最初に考えていたのとの違いはコンスタントバッファまで、内部に持たせているということです。これによりちょっと面倒なことが発生してますので、いずれ修正しましょう。

wvpbuffer をクラスメンバに持たせてしまった関係で、Drawにおいて、コンスタントバッファの再セットが必要になります…そう考えると現段階では Map~Unmap 意味ないよね。まあ元の通り置いとこう。

```
dev.Context()->VSSetConstantBuffers(0, 1, &_wvpbuffer);
```

Draw 関数内で WVP バッファのセットをしなおしてしまうと、呼び出し側でも再セットしなければならないので、呼び出し側としては

```
billboard.Draw();
```

のあとで、

```
dev.Context()->VSSetConstantBuffers(0, 1, &wvpbuffer);
```

という感じですね。クラス化して動作したら一旦そこまでにしておきましょう。

## 29 音関連

やっぱりと言うか、なんというか「音」を後回しにしてしまいました。しかしながらゲームにおいては「SE」「BGM」は超重要な要素です。

スーパーマリオでも、ファイアーボールを当てるとき「ピュウ」と音がするし、クリボーを踏めば「ブモツ」と音がしますね？あの音が気持ちよさとなっているのですから、舐めてはいけませんね？

ということで音を鳴らせるようにしてみましょう。自前で実装するのは大変です。Windows API のライブラリは処理速度等の問題でゲームには向いていないでしょう。

で、「音」ってのは多分みなさんが思っている以上に難しいし、面倒です。スピーカーがどうのうな原理で音を鳴らしているのが分かりますか？フーリエ変換って聞いたことがありますか？

そもそも音をストリーミング再生しているときって、どうなってるの？

そこんところ、非常に面倒で難しい…それだけで15コマくらい潰れる自信あります。ということで、ライブラリを使いましょう。

色々とライブラリ等あってどれを使用するか迷うところですが、代表的な使いやすいのは CRIADX2LE です。

### 29.1 CRIADX2LE

ひとまず、ここからダウンロードしましょう。

<http://www.adx2le.com/download/index.html>

簡単なツールの使い方から覚えなければいけないんですが、それでも他のライブラリよりかは楽だと思います。

使い方はここを読んでけば大体OK

<http://www.slideshare.net/takaakiichiyo/adx2-le-38962377>

あと、忘れちゃいけないのがライセンスなんですが、

## 29.1.1ライセンスについて

「ADX2 LE」のライセンスには下記の 2 種類があります。

- 短期ライセンス(初期ライセンス)

ダウンロードから約 1 カ月使用可能なライセンスです。

「ADX2 LE」SDK をダウンロードすると最初はこのライセンスが同梱されていいます。

- 長期ライセンス

1 年間使用可能な長期ライセンスです。

長期ライセンスの入手・更新には、申請手続き(無償)が必要です。

長期ライセンスにつきましては[こちら](#)

と書かれています。一応短期ライセンスでも、使用できますのでここでは短期ライセンスで説明します。

ライセンスが切れるとツールが使えなくなりますが、ライブラリは有効なのでそこまでにゲームが完成していれば問題ないでしょう。

就職活動にわたって使用するつもりなら長期のライセンス申請をしておいたほうがいいでしょう。

ただ、ツールとライブラリのバージョンが違つてると、実行時にクラッシュ(一応エラー関数にはコールバック飛んでくる)しますのでお気をつけください。

マニュアルはここです

[http://www.criware.jp/adx2le/docs/windows/index\\_man.html](http://www.criware.jp/adx2le/docs/windows/index_man.html)

よ～～～く読んでおきましょう。変なところで嵌らないようにね？

実際のゲームプログラムへの組み込みはここを読んでください。

[http://www.criware.jp/adx2le/docs/windows/index\\_man.html](http://www.criware.jp/adx2le/docs/windows/index_man.html)

インストーラでどうこうするものではないため、自分で解凍して、自分でライブラリパスを通してあげなければいけません。

パスの通し方は「コンピュータ」で右クリック→プロパティ→システムの詳細設定→環境変数→ユーザ環境変数の新規

変数名を CRI\_LIB とでもしておきます。変数値に解凍したフォルダ名を指定します。

確認のため、コマンドプロンプトで

```
echo %CRI_LIB%
```

と打ち込んで、パスが出るようなら成功です。

ひとまずこのパスをプロジェクトに組み込みます。

追加のインクルードパスに \$(CRI\_LIB)\Include と指定し、\$(CRI\_LIB)\Libs\x86 と指定してください。(この環境変数操作の前から VS を立ち上げていたのなら一度 VS を落としてから開き直しましょう)

さらに「追加の依存ファイル」に

```
cri_ware_pcx86_LE_import.lib
```

も追加しておいてください。

これで準備第一段階は終了です。

ひとまずは実行して、何も変化無しということを確認しておいてください。

サンプルフレームワークを使っても良いんですけど、余計な機能(ウィンドウ出すとか)が付いている上に、内容がそれなりに難しい(オブジェクト指向分かってないと手上げ)ので、必要な分だけ使用したほうが良いでしょう(若干書き方は古い部分はあるけど、読むのは勉強になると思います)。

というわけで、ちまちまとやっていきますが、そのためにはここを今一度読んでください。

[http://www.criware.jp/adx2le/docs/windows/index\\_man.html](http://www.criware.jp/adx2le/docs/windows/index_man.html)

さて、最初は CRI プロジェクト(プログラムじゃなくてサウンドツールの方)自体はサンプルのものを使用させてもらいましょう。

samples¥criatom¥data¥Public

ってのの中を見ると ADX2 だの Basic だのがたくさんあると思います。これを自分のプログラムの直下に Sound ってフォルダを作って、そこにあるとコピーしてください。

さて、いよいよ CRI ライブラリのプログラミングです。

### 29.1.2 組み込み

ひとまずマニュアルの通りに進めましょう。でもメイン関数の中に書くとゴチャゴチャになるので、いったん SoundManager クラス(シングルトンクラス)を作りましょう。一つもの要領で作ってください。

SoundManager.cpp のトップで

```
#include<cri_adx2le.h>
```

と書いてください。

あくまでも SoundManager は CRI の処理をメインに置きたくないがためのクラスです。大したテクニックは使いません。とにかく Init()関数と Load()関数と Play()関数と Terminate()関数があればいいのでとりあえず、引数、戻り値無しで関数を用意してください。あ、あとは Update()関数もな?

よ～し、関数を定義した前提でコーディングしていくよー。これからは「なんで」って考えても不毛なんでわりかしそのままやってください。

ひとまず中で共通で使用する変数を定義します。

```
CriAtomExAcbHn acbHandle;           //ACBハンドル  
CriAtomExVoicePoolHn voicePool;     //ボイスプールハンドル  
CriAtomDbasId dbas;                //D-BAS/ハンドル  
CriAtomExPlayerHn player;          //プレーヤハンドル
```

これ、関数の外で宣言してください。プレーヤはもちろん遊ぶ人って意味ではなく、CD プレーヤとかのプレーヤです。

更に、様々なコールバック関数を定義する必要があります。

まずはエラー時のコールバック…

```
static void CriErrorCallbackFunction(const CriChar8 *errid, CriUInt32 p1, CriUInt32 p2,
CriUInt32 *parray)
{
    const CriChar8 *errmsg;
    //デバッグウィンドウにエラー出力
    errmsg = criErr_ConvertIdToMessage(errid, p1, p2);
    OutputDebugStringA(errmsg);
}
```

次にメモリ確保時のコールバック…

```
void* CriAllocatorFunction(void *obj, CriUInt32 size)
{
    void *ptr = malloc(size);
    return ptr;
}
```

メモリ解放時のコールバック

```
void CriFreeFunc(void *obj, void *ptr)
{
    free(ptr);
}
```

で、Init 関数で初期化してください

```
void
SoundManager::Init()
{
    //エラー時のコールバック関数登録
    criErr_SetCallback(CriErrorCallbackFunction);
```

```

//メモリアロケータの登録
criAtomEx_SetUserAllocator(CriAllocatorFunction, CriFreeFunc, nullptr);

//ライブラリ初期化
criAtomEx_Initialize_PC(nullptr, nullptr, 0);

//ストリーミング用バッファの作成
dbas = criAtomDbas_Create(nullptr, nullptr, 0);

//全体設定ファイルの登録
criAtomEx_RegisterAcfFile(nullptr, acfFilePath, nullptr, 0);

//DSPバス設定の登録
criAtomEx_AttachDspBusSetting("DspBusSetting_0", nullptr, 0);

//ボイスプールの作成
CriAtomExStandardVoicePoolConfig vpconfig;
criAtomExVoicePool_SetDefaultConfigForStandardVoicePool(&vpconfig);
vpconfig.player_config.streaming_flag = CRI_TRUE;
voicePool = criAtomExVoicePool_AllocateStandardVoicePool(&vpconfig, nullptr, 0);

//ACBファイルのロード
acbHandle = criAtomExAcb_LoadAcbFile(nullptr, acbFilePath, nullptr, awbFilePath,
nullptr, 0);

//プレーヤの作成
player = criAtomExPlayer_Create(nullptr, nullptr, 0);

//とりあえずBGM鳴らしましょう→あとでこの処理は別の場所に移します。
criAtomExPlayer_SetCueId(player, acbHandle, 0);
criAtomExPlayer_Start(player);

}

```

次に Update 関数

```
void
```

```
SoundManager::Update(){
    criAtomEx_ExecuteMain();
}
```

最後に Terminate 関数

```
void
SoundManager::Terminate(){
    //後始末
    criAtomExPlayer_Destroy(player);
    criAtomExAcb_Release(acbHandle);

    criAtomExVoicePool_Free(voicePool);
    criAtomEx_UnregisterAcf();
    criAtomEx_Finalize_PC();

}
```

とりあえず説明書に書いてるとおりに、初期化→アップデート→後始末の流れを作りましょう。

初期化はメインループ直前で、後始末はメインループを抜けた後。メインループでは毎回 Update 関数を呼ぶようにしましょう。

とりあえず Init で BGM 鳴らすところまで記述しているので、Main 関数に呼び出しをきっちり記載すれば BGM がなると思います。ほぼほぼまんまソースコード載せたので「鳴らない」って人は「サウンドデータ」を Sound フォルダの下に置くことに失敗しているのではないでしょうか。

で、実行時に DLL がないよ～って怒られると思いますので、自分のプロジェクトのフォルダに DLL をコピペして実行してください。

\*ちなみに DLL(Dynamic Link Library)ってのは実行時にリンクされるものなので、配布時にもつけておかなければゲームが実行されません。ご注意ください。

さて、後は鳴らしてみましょう。

Play 関数を引数アリにします。

```
void  
SoundManager::Play(int cueId){  
    criAtomExPlayer_SetCueId(player,acbHandle,cueId); // キューを選択  
    criAtomExPlayer_Start(player); // 現在のキューを再生  
}
```

## 29.2 XAudio2について

ひとまず XAudio2.h をインクルードし、XAudio2.lib をリンクしてください。ひとまず XAudio2 を使用するには CoInitializeEx を呼び出す必要がありますので、呼び出してください。

[https://msdn.microsoft.com/ja-jp/library/cc308016\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/cc308016(v=vs.85).aspx)

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms695279\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms695279(v=vs.85).aspx)

うん、とりあえず

CoInitializeEx(nullptr, COINIT\_MULTITHREADED);

って書いておいて。これは XAudio2 を使う前に必要らしいね。

あとは

[https://msdn.microsoft.com/ja-jp/library/cc308016\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/cc308016(v=vs.85).aspx)

に沿って書いていきましょう。

とりあえずここはまんま書いちやつて構いません。なにぶん公式ですから

```
IXAudio2* pXAudio2;  
if ( FAILED(hr = XAudio2Create( &xaudio, 0, XAUDIO2_DEFAULT_PROCESSOR ) ) )  
    return;  
  
IXAudio2MasteringVoice* pMasterVoice = NULL;  
if ( FAILED(hr = xaudio->CreateMasteringVoice( &pMasterVoice, XAUDIO2_DEFAULT_CHANNELS,
```

```
XAUDIO2_DEFAULT_SAMPLERATE, 0, 0, NULL ) ) )  
return;
```

ここまで処理がきちんと通るか確認しましょう。

ちなみにちょっとだけ解説しておくと、XAudio2ってのは COM コンポーネントの形式でできており、ComponentObjectModel の略称なのよね。

で、コイツの仕組みは大雑把に言うと DLL みたいなもんで、他のソフトウェアとの通信を行うものである。つまり Main プログラム側から COM を介して XAudio2 に対して通信を行い、サウンドドライバに命令を出して音を鳴らすってわけだ。

この COM を使用する際には一番最初に CoInitializeEx 関数を呼び、初期化する必要があるってことなのだ。Direct3D の create 関数みたいな感じで、これをやつとかないと始まらないって思っておいてください。

あと、アプリケーション終了時に

[https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms688715\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms688715(v=vs.85).aspx)

## CoUninitialize()

**を呼び出すのを忘れないで下さい。** COM は別スレッドのやつを呼び出しますので、これを忘れるとなかなか面倒なことになります。絶対忘れないで下さい。

次にマスタリングボイスに関してですが、この「ボイス」ってのは「声」って意味じゃないです。単なる「音」全般を操るオブジェクトと考えておいたほうが良いでしょう。

詳しくはここね

[https://msdn.microsoft.com/ja-jp/library/cc677016\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/cc677016(v=vs.85).aspx)

全体的に知りたかったらここを見といてください

[https://msdn.microsoft.com/ja-jp/library/bb694503\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb694503(v=vs.85).aspx)

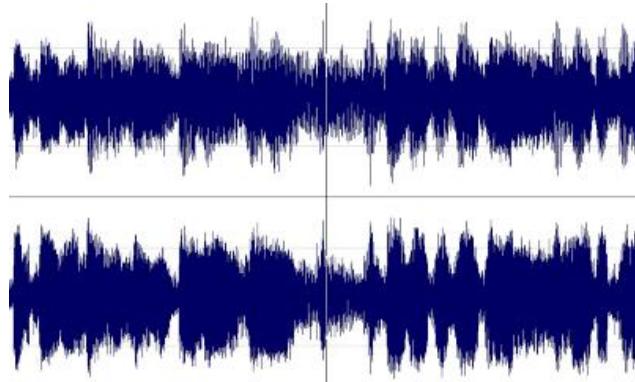
フルスクラッチで音を鳴らすのは思いの外面倒です。ただ…ここに時間をかけるわけにも行かないのよね…。

というわけで、前述のコードを書けばとりあえずは XAudio2 を使用する準備ができる状態です。

とりあえず音を鳴らしてみますが、残念ながら XAudio2 も DirectX11 と同様に…非常に様々なことを理解しておく必要があります。「音」ってのは「波」がスピーカーから発生し、その「波」が空気を伝わり鼓膜を振動させることにより「聞こえる」わけです。

で「波」って言いましたよね？

「波」ってことは、ヴィジュアル的には「波形」ですから



こういうの見たことありません？

これが「音」なのよ。これをものごつい分解していくと



なんか見慣れたものになっていきますよね？sin 波です。試しにこの sin 波で音を鳴らしてみましょう（そこからかよ!!!ええ、そこからです。）

要はどれくらいスピーカーを振動させるかってデータになります。

手順としては

- ① ソースボイスを作成する
- ② サウンドデータバッファを作成する
- ③ 中身を波形データで埋める

となります。

### 29.2.1 ソースボイスを作成する

ソースボイスってのはマスタリングボイスとは違って、音を鳴らすためのソースそのもので  
す。作成するには

audio->CreateSourceVoice

という関数を使用します。

[https://msdn.microsoft.com/ja-jp/library/bb633468\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb633468(v=vs.85).aspx)

```
HRESULT CreateSourceVoice(
```

```
IXAudio2SourceVoice**ppSourceVoice, //生成されるソースボイスへのタブリポインタ
```

```
const WAVEFORMATEX*pSourceFormat, //フォーマット
```

```
UINT32 Flags=0, //デフォルトでオッケー
```

```
float MaxFrequencyRatio=XAUDIO2_DEFAULT_FREQ_RATIO, //デフォルトでオッケー
```

```
IXAudio2VoiceCallback*pCallback=NULL, //デフォルトでオッケー
```

```
const XAUDIO2_VOICE_SENDS*pSendList=NULL, //デフォルトでオッケー
```

```
const XAUDIO2_EFFECT_CHAIN*pEffectChain=NULL //デフォルトでオッケー
```

```
);
```

引数は2つだけでオッケー(他はデフォルトのままでいい)

どうせ第一引数は受け取るだけなので、問題は第二引数ってことになります。

はい、第二引数の型はWAVEFORMATEXですね？

<https://msdn.microsoft.com/ja-jp/library/cc371559.aspx>

実はこれ、DirectX とも XAudio2 とも関係ない、Windows のマルチメディアの型やねんな。

```
typedef struct{

    WORD wFormatTag; // フォーマットタグ(PCM とか ADPCM を指定する)

    WORD nChannels; // チャンネル数(だいたい 1~2 ちゃんねる)

    DWORD nSamplesPerSec; // だいたい 44100 である…

    DWORD nAvgBytesPerSec; // 平均データ転送速度(44100*ブロックアライメント)

    WORD nBlockAlign; // ブロックアライメント(サンプリングビット数/8*チャンネル数)

    WORD wBitsPerSample; // サンプリングあたりのビット数(16 くらいでいい)

    WORD cbSize; // フォーマット情報のサイズ(0 でいいです)

} WAVEFORMATEX;
```

さて…

とりあえず一つ一つ解説すると、フォーマットタグってのは波形データのフォーマットです。とはいえる殆どの場合(特に WAVE データの場合)はフォーマットは

WAVE\_FORMAT\_PCM

を指定します。

チャンネル数はとりあえず 1(モノラル)で

ってなると、フォーマットの指定は

```
WAVEFORMATEX format={};

format.wFormatTag=WAVE_FORMAT_PCM;

format.nChannels=1; // チャンネル数

format.wBitsPerSample=16; // 1サンプルあたりのビット数

format.nSamplesPerSec=44100; // サンプリングレート

format.nBlockAlign=format.wBitsPerSample/8*format.nChannels;

format.nAvgBytesPerSec=format.nSamplesPerSec*format.nBlockAlign;
```

こんな感じでやってくらはい。

で、ひとまずコイツが S\_OK を返すのをご確認ください。

では次に

**29.2.2 サウンドデータバッファを作成する**  
サウンドデータバッファは

XAUDIO2\_BUFFER 構造体で作ります。

[https://msdn.microsoft.com/ja-jp/library/microsoft.directx\\_sdk.xaudio2.xaudio2\\_buffer\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.xaudio2.xaudio2_buffer(v=vs.85).aspx)

パラメータが多くて面倒ですが、大抵の場合は上3つだけ使用すればいいので、他はゼロにしつければいいです。

上3つとは

Flags

AudioBytes

pAudioData

です。「Xbox360」って記述があるあたり、「ゲーム作ってる～」って感じがしますなあ。

とりあえず

Flags は XAUDIO2\_END\_OF\_STREAM を指定

AudioBytes は全波形データサイズを指定

pAudioData に全波形データのアドレスを渡します。

なので

XAUDIO\_BUFFER audiobuffer={};

audiobuffer.Flags=XAUDIO2\_END\_OF\_STREAM;

audiobuffer.AudioBytes=サイズ

audiobuffer.pAudioData=波形データ

となります。

これをボイスデータに対して渡してやれば良いのです

[https://msdn.microsoft.com/ja-jp/library/bb694569\(v=vs.85\).aspx](https://msdn.microsoft.com/ja-jp/library/bb694569(v=vs.85).aspx)

```
sourcevoice->SubmitSourceBuffer(&audiobuffer);
```

でここで注意点ですけど、このオーディオバッファはあくまでも参照状態なので、**オーディオバッファおよび波形データを解放してしまわないようにスコープ等に注意しておきましょう。**要らなくなったら破棄して良いんだけど、少なくとも使用する間は消さないように注意してください。

とりあえずここまでできたらまたRESULTがS\_OKなどを確認し…たいところですが、まだ波形データが入っていないので、まず確実に成功しません。

というわけで、波形データを作つてみましょう。

### 29.2.3 波形データを作る

この部分は、通常はファイルから読み込んで作るものですが、今回は「音と波形の理解」のために敢えて自分で波形データを作つてみます。

ギターのチューニングって知つてますか？ラの音叉を使って行つうのです。



音程は

<https://www.youtube.com/watch?v=DycrcjPtW4M>

こんな高さです

さて作つてみましょう。

とりあえず1秒のデータを作りたいと思つます。どれくらいのバイト数が必要でしようか？

現在 44100Hz で作っていますので 1 秒間に 44100 のデータが必要…つまり 44100 バイト…  
なのですが、これが 2 ちゃんねる(ステレオ)だと倍になります。現在は BitPerSample が 16 なので、1 サンプルあたり 16 ビット → 2 バイト → short 型  
ということになります。

つまり  $44100 \times 2$  ということになりますが、実はこれは既に  
`format.nAvgBytesPerSec`

に渡していますので、これを使用します。

`std::vector<char> data(format.nAvgBytesPerSec);`

もしくは

`std::vector<short> data(format.nAvgBytesPerSec / 2);`

どっちでもいいです。

とりあえず `std::vector<short> data(format.nAvgBytesPerSec / 2);` のつもりでプログラム組みます。

short 型は -32767 ~ 32767 です。これが sin 波の振幅の上限と下限を表しています。

で、この値がスピーカーの飛び出し具合を制御します。

この値を増やしたり減らしたりすることで、スピーカーを振動させ、音を出します(空気を振動させます)

このスピーカーの変位を sin 関数もしくは cos 関数にすれば所謂 BEEP 音を鳴らすことができます。

例えば「ラ」音なら 440Hz です

<http://acoutis.jimdo.com/acoustics/%E5%9F%BA%E6%BA%96%E5%91%A8%E6%B3%A2%E6%95%B0a-440hz-%E3%81%A3%E3%81%A6%E4%BD%95/>

つまり波形をこの形にすればいい。

1 秒間で 440 周( $0 \sim 2\pi$  を 440 回)するようにすればいい。

もともとのサンプリング周波数が 44100 なので波長が  $44100/440$  になるようにすれば 440Hz となります。どういうことがというと、音の分解能が  $1/44100$  秒ってことですから 1 秒に 440 回( $0 \sim 2\pi$ )を往復するようにすればいい。

プログラムにする場合、サンプリングレートを 440 で割ったものが一周の長さ(波長)を表しますので、それを元に「波長」を計算します(float な)。

```
float wavelength=44100.f/440.f;
```

という風にこのデータの「波長」を計算します。あくまでも 1 秒を 44100 としたうえでの「波長」です。

あとは全データをこの法則にあてはめて埋めるだけです。

```
for(size_t i=0;i<audiodata.size();++i){  
    *p=SHRT_MAX*sin(i*XM_2PI/length);  
}
```

とやれば 440Hz の音データの完成です。

あとはこのできたデータを audiobuffer に割り当てねばいいので

```
audiobuffer.AudioBytes=audiodata.size()*sizeof(short);  
audiobuffer.pAudioData=(BYTE*)&audiodata[0];
```

なお、pAudioData のデータ型は BYTE ポインタ型なのでキャストする必要がありますが、short にせよ char にせよキャストしとけばいいのです。

どうですか？ 音が出ましたか？

んじゃ、次の段階としてシンプルな wav ファイルを読み込んで鳴らしてみましょう。サーバに置いてます(bomb.wav)から読み込んでみてください。

とりあえず wav データはこんな感じになっています。

RIFF.ID[0]	52 49 46 46	RIFF
RIFF.Size	00010CEE	
RIFF.FileType[0]	57 41 56 45	
Chanck.ID[0]	66 6D 74 20	WAVE
Chanck.Size	00000010	fmt
P CM	0001	
モノラル	0001	
標準化周波数	0000AC44	
転送バイト数	00015888	
サンプルサイズ	0002	
量子化ビット数	0010	
Chanck.ID[0]	64 61 74 61	
Chanck.Size	00010CCA	
WAVEデータ[0]	4F E1 E6 EF 66 E9 96 11 43 E4 23 EF 2A F5 AF F8 0磧・	data
WAVEデータ[16]	D2 F9 B3 F6 E6 09 BA 1D 68 17 24 1A 0C 04 CD 01 メ・	・

とりあえずこのフォーマットに関して言うと、最初の44バイトまでがヘッダデータなので、データは44バイト目からということになります。

<http://www.kk.ij4u.or.jp/~kondo/wave/#riff>

ぶつちやけメントクサイ。

とりあえず読み込んでみましょうか…。とりあえずこのデータにしか使用できませんが直値でやってみます。

とりあえず fopen でオープンして標準化周波数と転送バイト数と量子化ビット数とサンプルサイズをリードします。

標準化周波数は 24 バイト目から 4 バイト、転送バイト数はその後の 4 バイト。サンプルサイズはその後の 2 バイト、量子化ビット数はその後の 2 バイトです。さあ読み込んでみてください。

そうすると標準化周波数が 44100

転送バイト数が 88200

サンプルサイズが 2

量子化ビット数が 16 になることを確認してください。

そして、そこから 4 バイト飛ばします。4 バイト飛ばすに fseek(ファイルポインタ, 4, SEEK\_CUR) とします。

そこから 4 バイトが「チャンクサイズ(データのサイズ)」となります。それも読み込んでください。うまくいけば 68810 になるはずです。

あとは68810バイト読み込みましょう。

```
    fread(&audiodata[0],sizeof(unsigned short),audiodata.size(),wavfile);
```

読み込んだデータを元にフォーマットを設定…

```
format.wBitsPerSample=bitpersample; //1サンプルあたりのビット数
```

```
format.nSamplesPerSec=samplingrate; //サンプリングレート
```

```
format.nBlockAlign=samplesize;
```

```
format.nAvgBytesPerSec=transferbytes;
```

それで…鳴らすッ…!!

さあ、やってみよう。やれましたか？

さてここまでやっておいて何なのですが、ホントにサウンドを自分でやろうとするとファイルフォーマットやらチャンクやらスレッドやらをいじくりまわさなければならぬので、ライブラリに頼ったほうが良いと思います。

## 30 おすすめの書籍、サイト

最後の授業なので、これから自学自習ができるようにおすすめの書籍やサイトを紹介します。

DirectX編

書籍:

[SHADERGURU](#)

書籍に関してはこの本くらいです。ちなみにプログラム書法的にはあまり参考にならないのでテクニック的な部分だけ参考にしておいてください。最近はUnityやUE4の本ばかりで、こういうストイックな本は売れないというより技術者のブログを見たほうが早いですね。

Webサイト:

ZeroGram <http://zerogram.info/>

この人はもう DirectX12 にまで行ってますね。わりとすごい感じです

Project ASURA <http://asura.iaigiri.com/top.html>

ちょっと古いけど丁寧に理論の解説をしています

MarvericProject <http://maverickproj.fc2.com/pg00.html>

ピンポイントに様々なシェーダエフェクトを公開してくれています。

ゲームつくろー <http://marupeke296.com/GameMain.html>

言わずと知れた参考サイト。丁寧すぎるほどです。基礎的なことを書いているので、全て見ておくことをお勧めします。

## C++編

書籍:

[ゲームプログラマのためのコーディング技術](#)

必読。これくらいは自腹を切っておこう。

[GameProgrammingPatterns](#)

これも読んでおいたほうが良いかもです。

プログラムの本はそれなりに値が張るので、友人たちで出し合って輪講というか、読書会的な事をやったほうが良いかもしれません。

最後に…

[EffectiveC++](#)

[More Effective C++](#)

[Effective Modern C++](#)

基本ですね。高いので図書館やブックオフをお勧めします。

ちなみにこのへんの有名どころだと、インターネットの誰かがまとめてくれたりするので、本をよむ前にそちらを読んでおくと読みやすいと思います。

Effective++

<http://qiita.com/MoriokaReimen/items/58f183d421bb932cbbda>

ModernEffectiveC++

<http://qiita.com/ktsujino/items/39c8bc23dc1f7b17ab#1-%E3%83%86%E3%83%B3%E3%83%97%E3%83%AC%E3%83%BC%E3%83%88%E5%9E%8B%E6%8E%A8%E8%AB%96%E3%82%92%E7%90%86%E8%A7%A3%E3%81%97%E3%82%8D>

## 31 最終課題

提出期限 2/8(水)

テーマ: DirectX11 を使用してゲームを作りなさい

パクリOK。モデルやピクチャのデータもどつから持ってきててもいい(ただし就職活動に使うときは気をつけましょう)

音関係はどのライブラリ使っても可(なくともいい)

- CRIADXLite(ツール付き)
- XAudio2+ogg&voribis
- AudioKinetiic Wwise(評価版)

全体的に DirectX11 を使ってるなら、ライブラリは何を使ってもOK。

とにかく就職活動を意識してください。

ASOGAMESHOWとか、何かコンテスト出していると、その点は加算します。

## 32 ベジエ曲線について

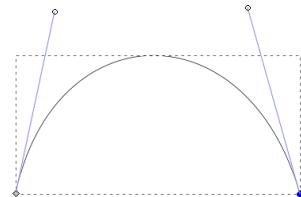
CG検定でベジエ曲面について出てきたので、ベジエ曲線について調べたことを解説します。

- 局所性
- 端点通過性
- 凸包性
- 平面再現性(直線再現性)

の中で「**当てはまらないもの**」を答えるというものです。う~ん。「用語なんかどうでもいいんだよ」といつも言ってる僕ですが、こと検定試験においては用語は重要ですね。

特に「局所性」が何を指し示しているのか、今とはわかりませんから。そこでひとつひとつ調べてみました。

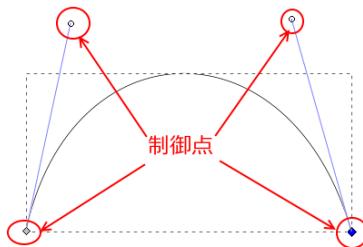
そもそもベジエ曲線とはどういうものでしょうか?これはパラメトリック曲線の一種です。



こういうやつです。PhotoshopやGIMPでも使うことがあるので知っていますよね。ちなみにMMDの中でも使用されています。この曲線にとって重要なのは

**制御点**

ちなみにこれはベジエ曲面でも同様ですが、曲線、曲面のかたちを決めるためのものです。



端点通過性は日本語として分かりやすいですね。端つこの点を通るかどうかですが、こいつは図を見てもわかるように「通ります」。ただし間の制御点は基本的には「通りません」。これが制御点通過性なら×ですが、端点通過性なら○となるわけです。

さて、次に平面再現性についてですが、これが二次元の場合は直線再現性となります。簡単に言うと、パラメトリック…つまりパラメータを変更すれば直線になれるかどうかということです。これはほとんどの曲線、曲面に当てはまります。

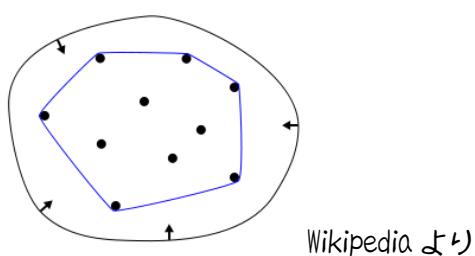
↑の例で言うと、制御点を端点と同じ直線状に置けばいいのです。



ね？というわけで平面再現性も○ですね。

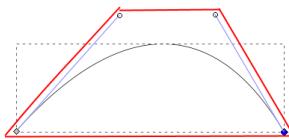
次に凸包性についてですが、これはベジエ曲線が制御点、凸包内部に収まるかどうかって話です。

凸包とはなんなのか？これは全ての点をすっぽり包むような閉じた形状のことです。

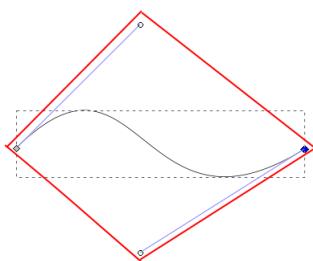


ベジエにおいては制御点をすっぽり包み込んでその中に曲線が収まってれば良いわけです。

つまり



これは凸包性を満たしていますね。さらに



こういう形でも満たします。

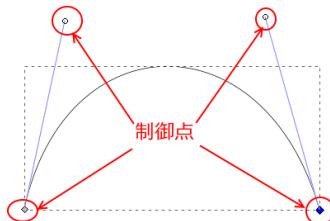
ベジエ曲線はその性質上、凸包の内部に収まるようになっています。

では最後に…局所性とはどういったものなのでしょうか？

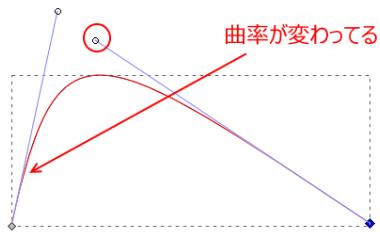
これは結構難しかったです。簡単に言うと、制御点の移動が、制御点の部分以外の曲線全体に影響を与えないという事を意味しています。

意味がわかりづらいですね。

つまり



この制御点のどれかを動かすと形を保てなくなってしまうと、局所性を満たさない。…そういうことです。ベジエ曲線は特定の制御点を動かすと…



のように制御点以外の曲率も変えてしまうため、局所性は満たしません。

ちなみに局所性を満たす曲線としては B-Spline があります。

<https://ja.wikipedia.org/wiki/B-%E3%82%B9%E3%83%97%E3%83%A9%E3%82%A4%E3%83%B3%E6%9B%BF%E7%BF%9A>

とはいって、B-Spline 曲線描画が可能なフリー・ソフトはなかなかないので、ここで図は描きませんが、制御点近傍にしか影響を与えないという特性があります。

ツールとしてはどうかというと、B-Spline は基本的に「制御点を通らない。端点も通らない」という性質であり、計算式もベジエと比べると難しいため、高度(高額)なソフト以外では使用されていないようです。

まあ…結局何が言いたいのかというと、今回のこの問題は「局所性」が×ってことです。

### 32.1 センサーも悩むやで

さて、これは2年前のテキストではテキトーに誤魔化して流した部分なんだが、

MMD の左下のこの部分…



これはベジエ曲線や。そこまではええで。そしてこの情報は VMD の中に入っとるやで。

ええねん…それはええねん。

ちなみに VMD の説明では

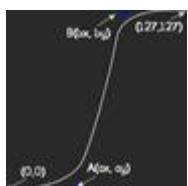
//// 補間の補足 ////

BYTE Interpolation(b4); // (4)(4)(4) // 補完

// 補間用の曲線

// (0,0), A(ax,ay), B(bx,by), (127,127) の 3 次(4 点)ベジエ

// A:左下の+, B:右上の+



// モーションの補間パラメータの並び順(MMD 板の情報)

// 回転は 4 軸(ウォータニオン)だが、1 個にまとめられているので注意

// X 軸 Y 軸 Z 軸 回転

// A(Xax,Xay)(Yax,Yay)(Zax,Zay)(Rax,Ray)

// B(Xbx,Xby)(Ybx,Yby)(Zbx,Zby)(Rbx,Rby)

// とすると、

// Xax,Yax,Zax,Rax, Xay,Yay,Zay,Ray, Xbx,Ybx,Zbx,Rbx, Xby,Yby,Zby,Rby,

// Yax,Zax,Rax, Xay,Yay,Zay,Ray, Xbx,Ybx,Zbx,Rbx, Xby,Yby,Zby,Rby, 01,

// Zax,Rax, Xay,Yay,Zay,Ray, Xbx,Ybx,Zbx,Rbx, Xby,Yby,Zby,Rby, 01,00,

// Rax, Xay,Yay,Zay,Ray, Xbx,Ybx,Zbx,Rbx, Xby,Yby,Zby,Rby, 01,00,00

こんな感じで書いてます。正直面倒くさいやで。

さて、何故このような記録のされ方をしているのか分からぬ。結局データは重複しているし、何をしたいのかさっぱりわからぬ。unsigned char\*b4 個のデータなんだが、どう見ても Xay や Xbx などが 4 つもかぶっているのがわかると思う。こんな感じで 64 バイト使うのならば一つ一つの数値を float(4 バイト)にして、それを 16 個使ったほうがよっぽど良いと思うんだが…。

…まあ、言っても仕方ないか。

と、それは置いておいて、最大の悩みはこれだ。

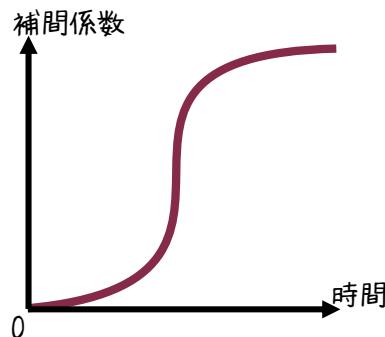


はい、これはパラメトリック曲線の一種であるため

$$y = y(t)$$

$$x = x(t)$$

という具合に、それぞれ $t$ によって値が決まる関数の組み合わせなのである。これを  
 $y=f(x)$ の形にしたい。



何故ならば図のように、補間係数が時間の関数になっているからである。…といふか、縦軸が横軸の関数にならなければならぬのだ。実用として。

数式で書くと

$$t = t(x)$$

が必要なのだ…これさえ求めれば

$$y = y(t(x))$$

とすることができる。ところでこの場合の $t(x)$ とはなんだろうか？

所謂「逆関数」である。高校でやったことがある人もいると思うが

$y = f(x)$ ならば

$x = f^{-1}(y)$ が存在するというわけだ。もうわからんねえな、これ。

ところでベジエ曲線というのは三次の場合

$$x = x(t) = t^3 x_4 + 3t^2(1-t)x_3 + 3t(1-t)^2x_2 + (1-t)^3x_1$$

および

$$y = y(t) = t^3 y_4 + 3t^2(1-t)y_3 + 3t(1-t)^2y_2 + (1-t)^3y_1$$

だが、こういった三次式の逆関数を求めるのは困難を極めるのだ。

<https://ja.wikipedia.org/wiki/%E4%B8%89%E6%AC%A1%E6%96%B9%E7%A8%8B%E5%BC%8F#.E3.82.AB.E3.83.AB.E3.83.80.E3.83.8E.E3.81.AE.E5.85.AC.E5.BC.8F>



## 逆関数を求める

- 3次方程式の解

$$x = 3t^2 - 2t^3$$

```
rules = Solve[3 t^2 - 2 t^3 == x, t]
{{t -> 1/2 \left(1 - \frac{1}{\left(-1 + 2 x + 2 \sqrt{-x + x^2}\right)^{1/3}} - \left(-1 + 2 x + 2 \sqrt{-x + x^2}\right)^{1/3}\right)}, 
 {t -> 1/2 + \frac{1 + I \sqrt{3}}{4 \left(-1 + 2 x + 2 \sqrt{-x + x^2}\right)^{1/3}} + \frac{1}{4} \left(1 - I \sqrt{3}\right) \left(-1 + 2 x + 2 \sqrt{-x + x^2}\right)^{1/3}, 
 {t -> 1/2 + \frac{1 - I \sqrt{3}}{4 \left(-1 + 2 x + 2 \sqrt{-x + x^2}\right)^{1/3}} + \frac{1}{4} \left(1 + I \sqrt{3}\right) \left(-1 + 2 x + 2 \sqrt{-x + x^2}\right)^{1/3}}}
```

今回の場合はもうすこしだけ簡単することは可能である。何故かと言うと  $x_1=0$  で、  
 $x_4=\max$  であることが分かっているからだ。少なくとも  $x_1$  の項は消えるんやで。

…ところが結局三次の項が残っている以上はろくでもない計算が残っているのだなあ。

でも、MMD の作者は何とかしているのだ。数学の専門家でもないはずだから、なんか抜け道はあるはず…あるはず。

ということで、

<https://ja.wikipedia.org/wiki/%E3%83%8B%E3%83%A5%E3%83%BC%E3%83%88%E3%83%B3%E6%B3%95>

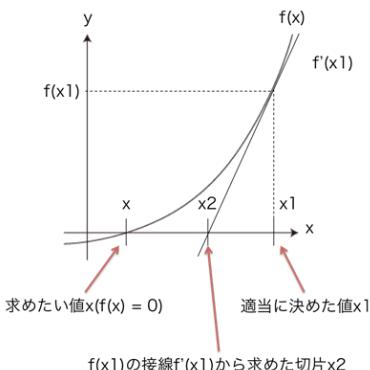
<http://qiita.com/PlanetMeron/items/09d7eb204868e1a49f49>

ニュートン・ラフソン法って奴を見つけました。

まず、この方法の目的は

$$t(x) = 0$$

となる  $x$  を見つけることとすると



微分法によって、求めたい答え( $x$ )に近づけていくというものです。試行回数を増やせば限りなく  $x$  に近づきます。

…IK が CCD-IK を使ってゐるあたり、MMD はこの手を用いている可能性が高い!!!!

まあ、繰り返し処理に強いのはコンピュータの特権やしな。

最終的に求めたいのは特定の  $X$  のときの  $Y$ 。そのためには特定の  $X$  のときの  $t$  を求めたい。