

進撃のDX11

attack on DirectX11

申し訳ございません!!

私が無能なばっかりに、ただいたずらにゆとり教育を行い…!!

貴方の能力を…!!最大限向上させることができませんでした!!

というわけで、もうC#やDxLibとはお別れである。お別れはしたか?

よろしい。

それではこれから地獄の旅路へと足を踏み入れることにしよう。ただしこれからやるのはDirectX11である。DirectX9よりも少し難しいのである。

なぜわざわざこれを選択したのかというと、君たちが卒業する頃にはXbox OneやPS4用のゲームが現実に開発される状態になっているだろう。そういう状態になってしまっていればDX9時代の技術だけではだんだん通用しなくなってくるのが正直なところだ。

作品作りはDirectX9でいいかもしれない。だが面接の時に確実に次世代の技術について聞かれるであろう。その時にDirectX9の知識だけでは不十分なのだツツツ。

どうせそういうことならば今のうちからDirectX11を弄り倒そうではないかというのが本講座の趣旨である。

とはいえやはりいきなりDX11の話は難しいかもしれないため、付いてこれないと思う人は『地獄の3D.pdf』でも見ておいてくれ。

で、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を理解した後でさあゲーム開発ってやってると、次年度就職年次的人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お勧めする。というか、ホンマの話、授業の課題だけやってたら確実にゲーム業界とか行けへんからな!

オススメは授業は授業としてDirectX11の技術を身につけ、授業外、放課後にて自分からかじめDxLibやDirectX9でもゲームを制作しておき、DirectX11が理解できたらDirectX11に移植するってやり方かな。

ここで注意点はDxLibを使用したゲームはゲーム外者に提出しても評価してもらえない可能性が高い(というか評価されない)という事である。またDirectX9でも、相当のものを作っておかないと武器にはならない。

と書くと、DirectX11ならなんでも評価されると勘違いする人が出ると思うけど、それも

意味が無いからな。とにかく何かしらゲームを作つておくこと、DirectX11で作つておくとなお望ましいという程度だ。

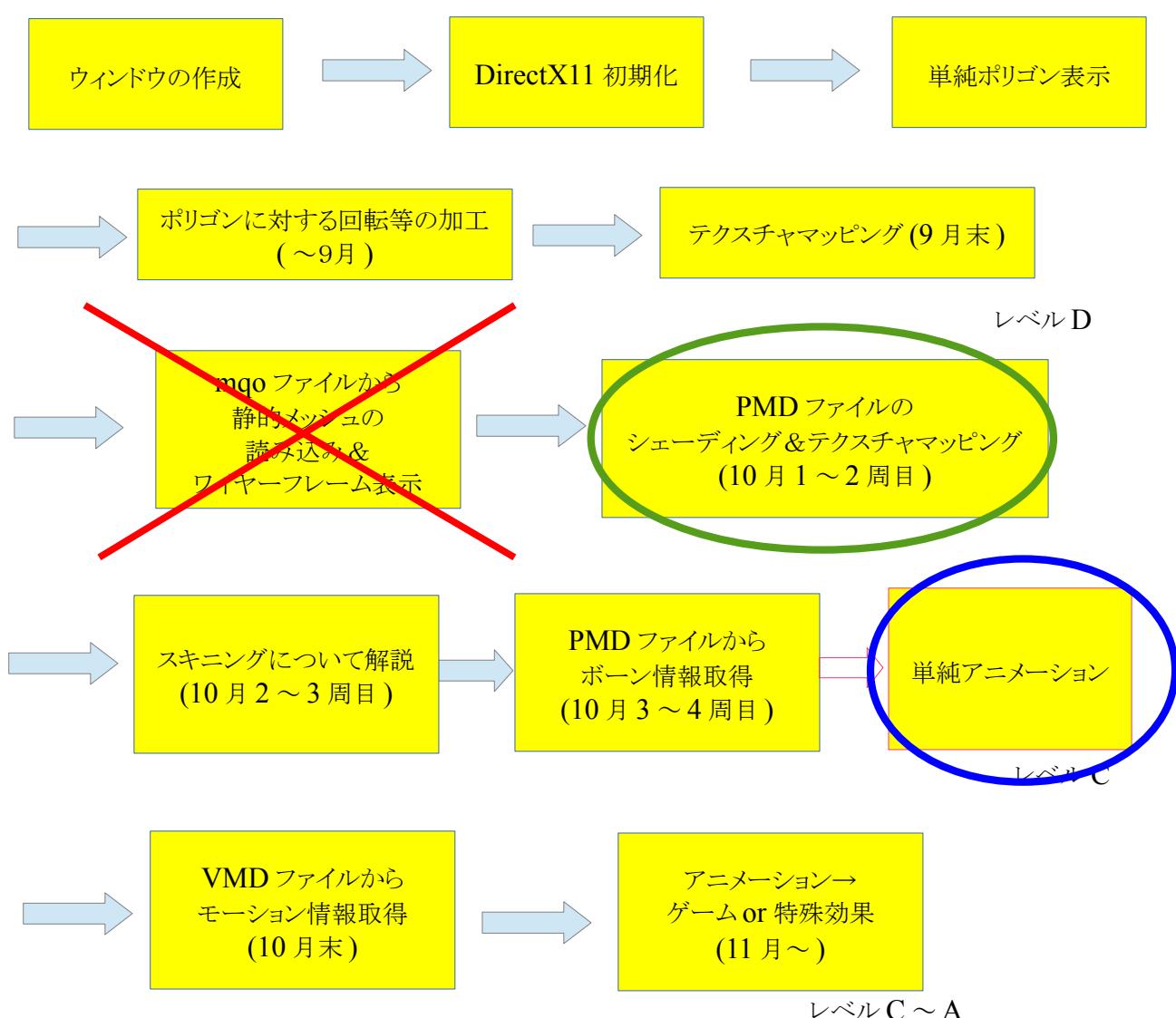
何度も言うようだけど、ゲーム業界に通用する作品つくりたいなら、授業外でも頑張らないと無理っ…無理っ…!

ちょっとくらい血反吐はかないダメッ…ということは分かつて下さい。ああ、課題できない人は、もう、業界受ける資格さえ無いと思ひますから、自覺しといてね。

課題提出できない人は落第させますんであしからず。

独学で DirectX11 やつて、アツ！無理！と思ったら、もう DX9 で自主制作しといたほうがいいと思います。無理して何もできなかつたじや悲しいので。

授業全体の流れ



合間合間にC++やらハードウェアに関する話、数学に関する話を挟んでいく。このへんの話を聞き逃すと全くわからなくなるので、授業を欠課しないことをお勧めします。

目次

プロジェクトの下準備	9
ウインドウ表示するバー	12
C++概論	17
ポインタ	17
アロー演算子	20
ポインタのポインタ	20
参照	23
C++のクラス	23
その他	24
include文	24
テンプレート	24
STL	25
sizeof(型もしくは変数)	25
DirectX 11の初期化	26
初期化に必要な基本的な用語と知識	26
①デバイス&デバイスコンテキスト&スマップチェーンの作成	27
②レンダーターゲットビューの作成	31
画面クリアすつか	33
ポリゴンを表示しよう	34
三角ポリゴン表示	36
ビューポート設定	37
頂点定義	38
Direct3D 頂点用のバッファを生成	39
シェーダ初步	43
スリーディーっぽく…していこう	46
行列について	47
四角にしてみよう	56
テクスチャマッピング	58
絵を貼り付けるための基礎知識	58
用語	58
テクスチャマッピングする手順概要	61
注意点といふか	62
テクスチャ読み込み	62
シェーダ側の準備	63
頂点レイアウトの変更	64
頂点情報にUV情報を追加	66
サンプラーの作成	67

シェーダにテクスチャとサンプラーをセット	68
シェーダ側にUV情報を追加する	69
もう最後!ピクセルシェーダ側だよ。	69
PMDからワイヤーフレーム表示	73
バイナリファイルとは…	76
PMDファイルを読もう	78
PMDのヘッタ部分	79
頂点データを読んでいこう	85
①頂点構造体を定義	86
②頂点レイアウトの変更	86
③頂点データ読み込み	87
④頂点からPMDを表示する	88
ところでマニュアル(MSリファレンス等)はきちんと読んでる?	91
インデックスバッファとは	92
インデックスバッファ作成	93
インデックスデータ読み込み	94
CreateBufferでインデックスバッファ作成	94
インデックスバッファをセット	96
インデックスバッファを使って描画	96
トライアングルリストで表示してみよう	97
早速シェーディング行ってみよう!	98
基本用語	98
シェーディング	99
輝度	99
平行光源(平行光線)	100
法線ベクトル	100
線形補間(リニア補間)	102
内積	103
ベクトルの正規化	104
ディフューズ・スペキュラー・アンビエント	105
コサインシェーディング	106
法線ベクトルも回転させよう	108
Zバッファ法(深度バッファ)	109
深度バッファ(深度レンダーターゲットビュー)の作成	111
深度のクリア	112
マテリアルを反映	114
マテリアルデータを見てみよう	114
マテリアルフォーマット	114

マテリアル読み込み	115
マテリアル適用(ディフューズのみ)	116
マテリアル毎に面を描画	117
ディフューズ反映の概要説明	118
材質情報を渡す用の構造体を作成	119
コンスタントバッファーの作成	119
マテリアルループ時にコンスタントバッファーの内容を変更	119
頂点シェーダにマテリアルバッファをセット	120
シェーダ側のコード変更	120
モデルにテクスチャを反映させます	121
複数のテクスチャオブジェクトの準備	122
テクスチャ名取得の際の注意点	122
余談(スフィアマップについて)	124
マテリアルループ内でテクスチャの切り替え	124
シェーダ側の処理	125
テクスチャがあるときないとき	125
トラブルシューティングゲーム	126
特定のポリゴンが特定の向きでは表示されない	126
背面カリングとは?	126
背面カリングOFFはどうやんの?	127
透明色への対応ツ…!	128
そもそもアルファが有効になっていなレ!!	129
Zバッファ法と透過色の気色悪い関係	130
地獄のスキンメッシュ	131
基礎知識	131
基本用語	131
ボーン	131
スキニング	132
FK(フォワードキネマティクス)とIK(インバースキネマティクス)	133
クオオ…タニオン!!	135
ツリー構造	137
メインになる技術	138
ボーン情報の読み取り	138
モデルを動かすこと	140
オフセット回転	141
ボーンを全部投げる	144
コンスタントバッファに対する番号付け	144
ボーン行列をゼーんぶ渡す	146

<u>そうだウェートもかけておこう</u>	148
<u>ツリー構造を理解する</u>	149
<u>そもそもツリー構造とは</u>	149
<u>ツリー実装</u>	151
<u>行列について再考</u>	158
<u>行列の×の順番と、転置についての話</u>	158
<u>行列と頂点とボーンの関係</u>	160
<u>2つのボーンと影響度</u>	164
<u>VMDを再生しよう</u>	170
<u>ポーズ情報を見てみよう</u>	170
<u>VMDファイルを見てみよう</u>	173
<u>std::mapを使ってみる</u>	176
<u>フレーム間補間</u>	177
<u>線形補間(フレーム補間)</u>	178
<u>XMQuaternionSlerp</u>	180
<u>ベイジえへの話</u>	181
<u>数学的なベジエ解説</u>	182
<u>MMDのベジエデータとその応用</u>	184
<u>CCD-IKについて考えよう</u>	186
<u>シャドウについて考えよう</u>	186
<u>シャドウマップ</u>	190
<u>シャドウボリューム(ステンシルシャドウ)</u>	194
<u>ステンシルバッファとは</u>	203
<u>隣接頂点プリミティブ</u>	211
<u>フェイシャルモーション</u>	217
<u>モーフと表情</u>	218
<u>みんな…だいすき…IKっ…!</u>	223
<u>基本的な数学</u>	224
<u>2次元CCD-IKから学ぼう</u>	225
<u>PMDのIKを動かしましょう</u>	229
<u>PMDのIK情報を見る</u>	229
<u>いざCCD-IKを実践</u>	231
<u>セルシェーダについて</u>	238
<u>輪郭線</u>	238
<u>反転法</u>	238
<u>深度から輪郭線</u>	239
<u>隣接頂点の法線情報を見る</u>	239
<u>PMXについて</u>	240

頂点データ表示.....	241
PMX テクスチャテーブルとマテリアル.....	242
PMX のマテリアル.....	244
透過とミッファについて.....	246
2パスレンダリング.....	249
真・2パスレンダリング.....	252
スペキュラを実装しよう.....	255
<u>豆ちしきー</u>	256
エラーへの対処.....	256
リザルトの活用.....	256
シェーダコンパイラ.....	257
error オブジェクト.....	258
VisualStudio のお便利なツール.....	258
PIX for Windows.....	260
SVってなんや?.....	261
凸凹の進化.....	262
ディファードレンダリング.....	264
設計なんかクソつ喰らえ.....	265
Doxxygenとか.....	266
<u>最終課題</u>	267

プロジェクトの下準備

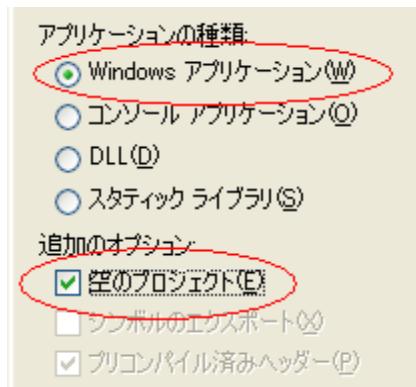
さて…Visual Studio の C++ を使用してプログラムを書いていくわけだが、C# しかいじつたことのない人もそれなりに混じっていると重いますので、ひとまずはプロジェクトの設定からお話ししましよう。前期に C++ やってた人はともかく、C# でやってる人はまずプロジェクトの設定からやらなければならぬ。

なので、C++ やってた人はしばらく退屈だろうから、ここは飛ばしてホイホイ先に進んでいいともういたい。

まずは Visual Studio を起動して下さい。

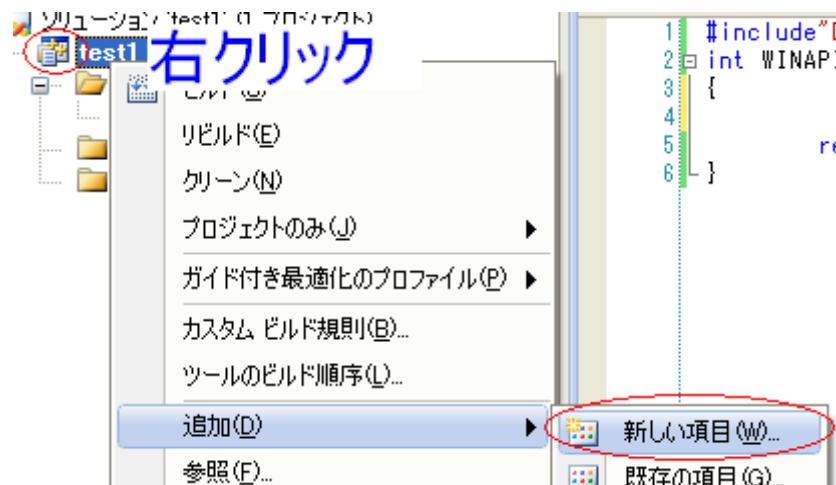
立ち上げたら「プロジェクトの新規作成」→「Visual C++」→「Win32 アプリケーション」を選択（「コンソールアプリケーション」ではない）。名前は好きに付けてください。

すると、「Win32 アプリケーションウィザード」というウインドウが出てくるので「アプリケーションの設定」ボタンを押します。ここでアプリケーションの種類は「Windows アプリケーション」です。追加のオプションの「空のプロジェクト」にチェックします。



できましたか～？

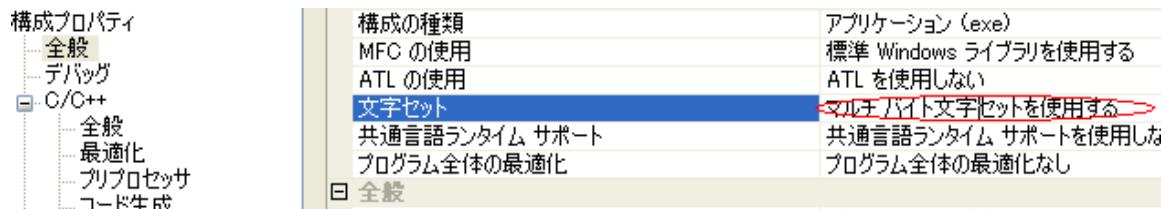
それができたらメインとなるコードをプロジェクトに追加します。ソリューションの下にプロジェクトがある状態になっていると思いますので、プロジェクトで右クリック→追加→新しい項目→C++ ファイル



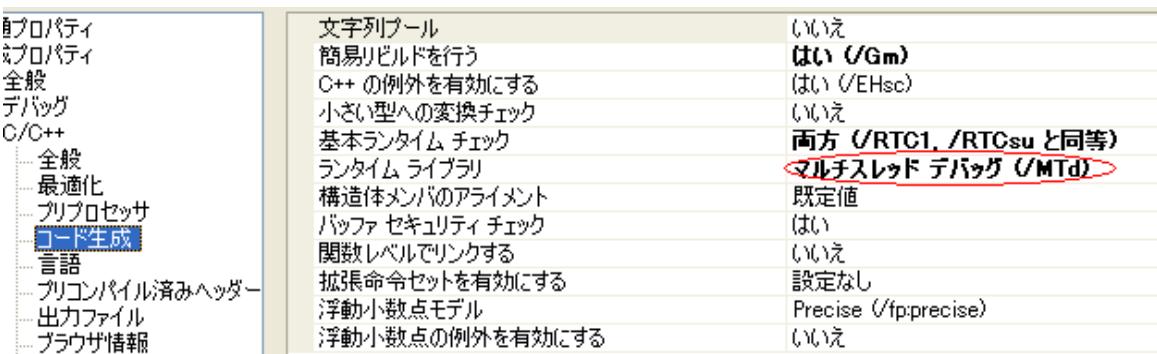
と選択し、メインとなるソースコードを追加します。この時の名前は適当でいいですが、main関数があるので main.cpp とでもしておきます。

はい、これでプログラムを書く準備はできましたが、このままでは書いたプログラムがうまいこと動かないのでもう少し設定を付け足してあげます。

メニューのプロジェクト→プロジェクトをクリックするとウインドウが立ち上がりますので(全般)を選択し、文字セットの設定を“マルチバイト文字列を使用する”に変更します。



また、C/C++のコード生成の“ランタイムライブラリ”的設定を「マルチスレッドデバッグ DLL」から「マルチスレッドデバッグ」に変更します。



まだまだです。

次にもう一回「全般」に戻って、「追加のインクルードディレクトリ」を選択します。そうすると、C言語におけるディレクトリサーチングパスを指定できるので、ここで DirectX11 のインストールディレクトリの SDK の Include を指定してあげます。

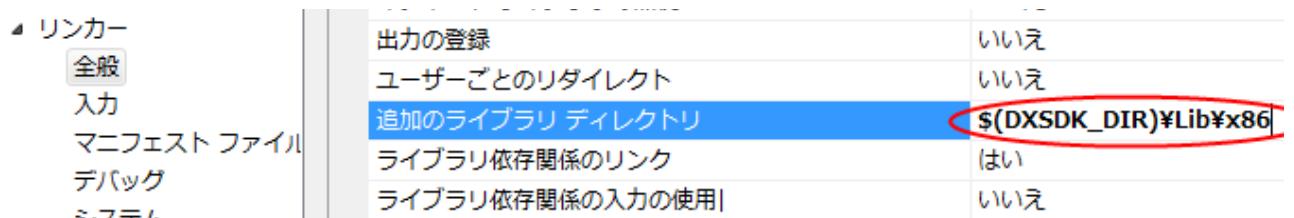
通常はDXSDK_DIRがSDKのフォルダのエイリアス(別名)になっているんですけど、うまくいかない人はコマンドプロンプトでecho %DXSDK_DIR%と入力し、パスが通っているか確認しておいて下さい。

上手く行ってる人は、追加のインクルードディレクトリを以下のように指定します。



さて、さてさてさてさて…まだだよ。まだまだアルよ。Direct3Dのプログラムはプログラム単体で動くわけじゃなくて、DirectX11の力を借りなきゃいけない。このためにC++ではexe生成時にライブラリをリンクすることにより大いなるDX11の力を使うことができるのだッ!プレイブリーデフォルトの魔界幻士みたいなもんですね。

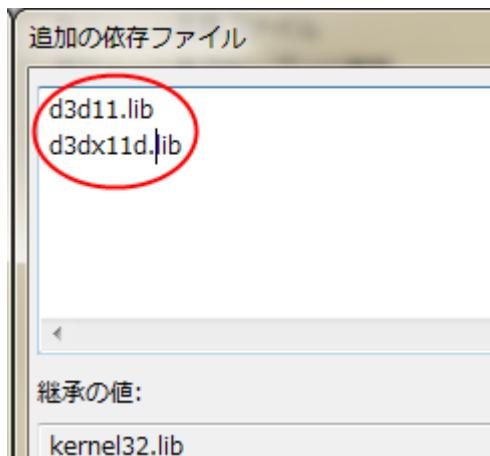
で、こいつもインクルードと同じように、色々とリンク先を探しに行くのだけれど、こちらから指定してあげないと見つけきれないのさっきと同じように教えてやる。



はいこのとおり。

最後にリンクするファイルの指定が必要だ。

リンク→入力で→追加の依存ファイルの右の…を押すとウインドウが出てくるので、そいつに以下のように入力して下さい。



さて、やっとこれでプログラムを書く準備ができましたので、一番最初に追加した main.cpp の先頭に

```
#include<windows.h>  
  
#include<D3D11.h>  
#include<D3DX11.h>
```

と書いておいて下さい。下準備はここまでです。

ウィンドウ表示するべー

まずは何よりも先に、ウィンドウを表示しなければならない。しかし DxLib を使わない
ウィンドウ表示は思った以上に面倒である。だがやらねばならない…面倒くせえとかじやね
えんだよ。やらにゃならんのよ。

では DxLib の時にもやったように WinMain 関数を作ろう。これはもう説明する必要はある
まい。

```
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPCSTR, int)
```

のアレである。実は int main()でもいいんだよ？ 実はね。知ってた？ 結局アプリケーション
(exe)にマッピングされたハンドルさえあればイ茵タヨ。

ちなみに、int main()で作っても exe はできるので、そのハンドルを返すには、
GetModuleHandle 関数を使用する。

```
int handle=GetModuleHandle(NULL);
```

で、exe 自身のハンドルを使用出来る。で、このハンドルが何に当たるかっちゅーと、WinMain
の第一引数のハンドルである。ただしプロジェクトの設定で、エントリーポイントを WinMain
にするか、main にするかを設定しておかないと多分リンクエラーは起きるね。そこは各自設

定しておいてくれ。

余談だが、このmainでアプリケーション作っておくと何が便利かというと、printfやcoutの結果をコンソールウインドウで確認できるので便利である。

さらに余談だが、Visual Studio のデバッグの「出力」タブに文字列を出力したい場合は::OutputDebugString(出力したい文字列);と書いておけばいい。

ああ…このいわゆる「printf デバッグ」が必要ないもしくはどうして必要があるのかわからないという人は、相当のレベルのプログラマかもしくは全然自分の頭で考えてないエンジニアだろうね。

とりあえず、余談はここまででさっそく次行ってみよう。
何度も言うようだけど、ウインドウ表示するだけで面倒なのだ。

ウインドウの作成部分は Direct3D と関係しているわけではないので、まだ Direct3D らしい所は出てこないが、しばらく辛抱してくれ。

何が面倒なのかというと、表示するウインドウの指定は細かく指定できるが、細かく指定出来るだけに全てを書かなければならない。

ウインドウの設定に使用する構造体にそれぞれ値を設定することにより、表示するウインドウのスタイルが変化する。

その構造体は WNDCLASS というもので、CLASS と書いてあるが実際は構造体である。とりあえず実際に表示するまでの手順を書いておく。

ウインドウ表示の手順

- ① WNDCLASS 型のオブジェクトを作成し、各パラメータを設定
- ② RegisterClass で WNDCLASS の内容を OS に教えてやる
- ③ AdjustWindowRect でウインドウのサイズを補正
- ④ CreateWindow でウインドウオブジェクトを作成します（まだ画面に表示はされていない）
- ⑤ ShowWindow で手順④で作成したウインドウを表示
- ⑥ メインループを作成する

ここからたったの 6 ステップでウインドウを表示できる。あとは Direct3D の初期化となる。

まずは WNDCLASS 型の変数を宣言、初期化しよう。

```
WNDCLASS w=WNDCLASS();
```

これで、変数 w の中身は 0 で初期化された。C++ の仕様で構造体はデフォルトコンストラクタを呼ぶと 0 で初期化されるため、それを利用している。

あとは必要なパラメータを設定していくだけだ。この構造体の中で設定する必要があるのは `lpfnWndProc`, `hInstance`, `lpszClassName` の3つであり、`hInstance` は前述したように `WinMain` の引数を使用するか、`GetModuleHandle(0)` の戻り値を使用する。

`w.hInstance=hInstance;`
もしくは
`w.hInstance=GetModuleHandle(0);`

とでも書いてやる。

次に `lpszClassName` だが、これは適当に設定すればいい。

`w.lpszClassName="適当";`
とでもしてやる。

最後が若干ややこしいのが、前期までで関数ポインタをやったと思うので、特に説明はしないが、`lpfnWndProc` には、後述するウインドウプロシージャ関数のポインタを代入する。

これは Window アプリがコールバックシステムで成り立っているため、ウインドウズ OS からのイベントを取得するための取得先を設定する必要があるからである。

なので、予めコールバック用の関数を用意して、そいつを指定してあげればいい。コールバック関数の定義は

```
LRESULT CALLBACK 関数名(HWND hwnd,UINT msg,WPARAM wparam,LPARAM lparam){  
    //  
}
```

と書く。あとは、この関数名を指定してやればいい。

`w.lpfnWndProc=関数名;`

ここまでかけたら、OS に対して、アプリケーションクラスを登録する。ここでいう「クラス」ってのはただ単に WinAPI の用語であり、オブジェクト指向言語における「クラス」とはまったく意味が違うことは注意しておこう。紛らわしい。

`RegisterClass(&w); // アプリケーションクラスの登録`

ここまできてやっとウインドウの生成ができます。

```
HWND hwnd=CreateWindow("適当">//クラス名  
,"適当ですよ");//ウインドウ名  
,WS_OVERLAPPEDWINDOW//タイトル枠、ウインドウタイトル、ウインドウメニュー
```

```
ボックス、サイズ変更有り、最小化最大化あり  
,CW_USEDEFAULT//出現X座標はデフォルトを使用するよ  
,CW_USEDEFAULT//出現Y座標はデフォルトを使用するよ  
,wrc.right-wrc.left//幅指定  
,wrc.bottom-wrc.top//高さ指定  
,NULL  
,NULL  
,hInst//アプリケーションインスタンスを指定  
,NULL);
```

CreateWindowはウィンドウ生成をするのみで、表示そのものには全く関与していない。つまりこのまま実装すると、ウィンドウは生成されるが表示がされないという、結構けつたけな状態になりますのでShowWindow関数で表示をさせてあげます。

```
ShowWindow(hwnd,SW_SHOW);
```

このまま実行するとどうなるのでしょうか？そう、一瞬でウィンドウが消えてしまいますね！どういう事がというと、ShowWindowはウィンドウを表示するだけの関数ですから、表示後にWinMain関数を抜けてしまい、アプリケーションが終了しちゃうんですねー。

アプリケーション起動→ウィンドウ生成→ウィンドウ表示→アプリケーション終了

という流れになってるんですね。ということは、ウィンドウ表示の後で処理を留めておかなければならぬ。無限ループを使用してアプリケーションが終わらないようにしてあげましょう。

無限ループはwhile(true)でしたね。

こいつをShowWindowの後に置けば、アプリケーションが終わることはありません…とりあえずは、

```
while(true){  
}
```

とでも書いておきましょう。

ところがこいつは無限ループなので、完全にアプリケーションがフリーズします。なので色々と書いてやります。

PeekMessage等を使って、フリーズ状態にならないようにし、ウィンドウ右上のXボタンを押せば、ウィンドウが落ちるようにするには

PeekMessage,TransferMessage,DIspatchMessageを使用します。

これらへん自分で調べたり考えたりして書いて欲しいのですが、全体的にそんな時間もないんで、ウィンドウ表示サンプルとして公開します。

```

//ウィンドウ表示(のみ)サンプル
#include<windows.h>
#include<D3D11.h>
#include<D3DX11.h>

///Windowsからのメッセージを受け取る関数
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam){
    if(msg==WM_DESTROY){//ウィンドウ破棄された
        PostQuitMessage(0);
        return 0;
    }
    //それ以外の時はデフォルト処理を行う
    return DefWindowProc(hwnd,
        msg,
        wparam,
        lparam);
}

///Windowsにおけるmain関数
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE hPrev,LPSTR command,int cmdShow){
    WNDCLASS w=WNDCLASS();//このクラスはC#やC++でいうところの「クラス」ではないことに注意
    w.lpfnWndProc=(WNDPROC)WindowProcedure;//メッセージを受け取る関数を設定
    w.lpszClassName="適当";//クラス名
    w.hInstance=hInst;//アプリケーションインスタンスハンドル
    ::RegisterClass(&w);//クラス情報をOSに登録
    RECT wrc={0,0,800,600};//画面の幅と高さを決めておく
    ::AdjustWindowRect(&wrc,WS_OVERLAPPEDWINDOW,false);//指定した幅と高さになるように補正をかける
    //ウィンドウを作成し、ハンドルを取得
    HWND hwnd = CreateWindow("適当");//クラス名
    ,"適当ですよ");//ウィンドウ名
    ,WS_OVERLAPPEDWINDOW//タイトル、枠、ウィンドウタイトル、ウィンドウメニュー ボックス、サイズ変更有り、最小化最大化あり
    ,CW_USEDEFAULT//出現X座標はデフォルトを使用するよ
    ,CW_USEDEFAULT//出現Y座標はデフォルトを使用するよ
    ,wrc.right-wrc.left//幅指定
    ,wrc.bottom-wrc.top//高さ指定
    ,NULL
    ,NULL
    ,hInst//アプリケーションインスタンスを指定
    ,NULL);
    ShowWindow(hwnd,SW_SHOW);//ウィンドウを表示
    while(true){
        MSG msg;
        //メッセージを受け取り、受け取ったメッセージは削除
        if(::PeekMessage(&msg,NULL,0,0,PM_REMOVE)){
            ::TranslateMessage(&msg);
            ::DispatchMessage(&msg); //メッセージをディスパッチ
        }
        if(msg.message==WM_QUIT){
            break;
        }
    }
}

```

実際、次世代OS以降はウィンドウの作り方も変わっちゃいますし、これを一所懸命覚えて
もしやーないないので、まー写しちゃってもいいよ。
この後、どうせ血反吐はくしね。
さあ、地獄はここからだ

『もっと先へ『加速』したくはないか、少年。』



というわけで難しさが加速します。もうゆとりの時代については忘れましょ

C++概論

C++やるのが初めての人もいるみたいなので、サラッとおさらいも兼ねてC++の話をします。ガツソリC++勉強したい人は授業外でお願いよ。

ポインタ

特定の型の後に*(アスタリスク)を付けることによって宣言される型をポインタ型といい、それによって宣言される変数をポインタまたはポインタ型変数といいます。

C#しかやってない人にとってはC/C++で開発するときはこの「**ポインタ**」がキモになってくるでしょうね。ポインタってのは特定の**アドレス**を指し示す変数のことで、ポインタ型ってのがあるんだけどint変数を指し示すポインタとchar変数を指し示すポインタは違うわけ。この辺は慣れてくると思うけど、アドレスの意味がわからないと言にならないので説明しておく。

アドレスってのはメモリ上の特定の場所のことなわけね。

で、ポインタってのはアドレスを指し示す。

ひとことで言うとそんな感じ。

アドレスってのは数値で表されるものなんでポインタって言っても、中身だけ見るとint型変数のように見えるんだけど、コイツに型情報も含まれているところが通常のint型とポインタ型の違いなわけ。

ポインタ型でなにか宣言したければ、例えばchar型として特定のアドレスの内容を指し示す変数を作るのならば、

char* p;

という宣言になる。つまり一般化した書き方だと

型* 変数名;

という宣言になるわけ。で、何度も言ってるけど、こいつが指し示しているのはアドレスな

ので、いくらこの変数を見てもアドレス情報しか見えてこない。

どういうことかというと例えば

```
int a=18;
```

`int* p=&a; //変数の先頭に&をつけるとその変数のアドレスを返す。`

と書いた場合、`p`の内容はアドレスなんで例えば`0x0013dead`とかいうわけのわからん数值になっていたりする。(ちなみにC言語では先頭に`0x`つけると16進を表す)

`p`は既にメモリ上に確保された変数`a`の場所を指し示しているのであって、`a`の内容を指し示しているのではないことに注意して下さい。

とはいっても、結局最終的に欲しい情報はアドレスに書き込まれた中身です。これを知るためには、変数のアタマに`*`をつければ取り出せます。

例えばこの場合 `printf("%d", *p);`と書けば、18がコマンドプロンプトに出力されるわけです。

もうちょい詳しく説明しよう。

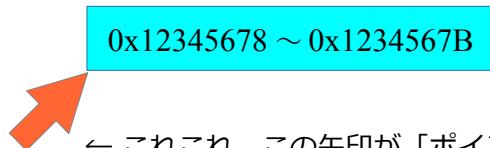
ポインタってのは日本語に訳すると、指し棒みたいな意味なのね。

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

ほら、マウスポインタっていうでしょツツツ!?ああいう意味。実際、ポインタってのはメモリ上に確保された何らかの情報を指し示すモノなわけよ。

```
int a=5;
```

とかであっても、実際にはコンピュータのメモリの`0x12345678`番地やらに配置されているわけやん?



← これこれ、この矢印が「ポインタ」なわけ

そして上の例のように `int` 型なら4バイト食いつぶしてるわけじゃん? そしたらポインタはどれくらい食いつぶしてるのかも知ってる必要があるわけ。

更に言うとポインタを利用するためには元の型を知っておかなければ使えねーじゃん? なのでポインタは元の型を覚えている作りになってるわけ。だからポインタの宣言は

```
int* p;
```

みたいに、型名の後にアスタリスクがついていたりするわけ。要は後で利用しやすいように『これは `int` 型を指し示すポインタですよ』って強調してるわけ。

もしそんな必要がなく、ただただアドレスだけを指し示したいのなら

```
void* p;
```

でいいわけだし。ね？ポインタの宣言が“型名*”の形式になっている理由とか意識したことある？別になくてもいいけど、なかなか理解できない人はこの理由を意識しておくといいでしよう。

以上のことから、例えば

```
int* a=0x12345678;  
char* b=0x12345678;
```

といったように、全く同じアドレスを指し示している2つのポインタ、これは意味が違う。ぜんぜん違う。ということを理屈的にも直感的にも理解していただきたい。何がちやうかというと

だから*aと*bは同じアドレスですが、全く違う結果になることを知っておきましょう。

また、

```
++a;  
++b;
```

この時のポインタの進み具合も変わります。

aはint型のポインタなので、++aで4バイト進み、++bはchar型なので1バイトしか進みません。つまり結果的には

aは0x1234567Bになり、

bは0x12345679になります。

分からん奴はわかるまで勉強しましょ。C/C++プログラマになりたければね。

で、ここできちんと認識しておいて欲しいのは、ポインタが指し示しているものはアドレスであるので、アドレス自体は32bitの場合は4バイトのunsigned intと等価な型になっているわけよ。

とにかく数値で表されているわけ。

そうすると、当たり前の話なんだけどポインタを宣言した時点で、またメモリ上に4バイトは確保されているってことなのね。

だから例えば

```
int a;  
int* b=&a;
```

なんてやって

int型の変数aが0x12345678だったとして、ポインタbがaのアドレスを指し示しているとすると、例えば、b自体のアドレスは0xdeadbeafであり、そつから4バイト確保されることになる。

アロー演算子

ちなみに、ポインタ変数が構造体型やクラス型だった場合、そのメンバ変数(属性)やメンバ関数(操作)にアクセスするには->アロー演算子を用います。->記号を組み合わせます。

ですから、例えば以下の様なクラスまたは構造体があるとします。

```
class A{  
public:  
    int value;  
    void Commit();  
};
```

で、通常はC#と同様に

```
A a;  
a.value=5;  
a.Commit();
```

などのようにドット演算子(.)で組み合わせることができます。A型のaがポインタ出会った場合は表記が変わってしまうのです。例えば

```
A a=new A();  
a->value=5;  
a->Commit();
```

のようにアロー演算子->でメンバにアクセスする必要が出てきます。こういうところがC++の厄介なところですね。

Direct3Dから取得するオブジェクトの殆どがポインタ状態であるため、この使い方を覚えておかないと分からぬことになりますので注意しましょう。アロー演算子ですよ!

ポインタのポインタ

はい、次にそのDirect3Dから取得するオブジェクトという部分と微妙に関わってくるのですが、Direct3DのCreate系の関数の殆どがポインタのポインタを渡すことでの出来上がったDirect3D系オブジェクトを受け取れる仕組みになっています。

ポインタの理解できていると思っている人でもたまに
「ポインタのポインタがわからん」という。

これはね、はっきり言っちゃうと理解してへんねん。え？なんでかって？

ポインタもまた変数であり、そいつも変数である以上はどこかしらにメモリが確保されているという重大な事実がわかつてへんねん。

つまり、やぱりメモリとかアドレスとかポインタが理解できていない。もうひと頑張りが足りていなければ。

例えば、アドレスがこういうものであるとする。この例では 0x12345678 とかつて書くとヤヤコシイのでメモリ上に 0~100 までアドレスが並んでいるとする。

メモリは一次元なので、一行目は 0~9 番目、二行目は 10~19 番目が並んでいて後は同じようにって感じだと思ってくれ。

さて、まず例えば `int a=5;` とかつて書いたとする。それが実行されるとアドレス上の何処かに 5 が書き込まれるわけや。自分でも適当に配置してちょ？

で、下の図だと、そのアドレス値は 15 であることがわかる。アドレスであることを分かりやすくするために 16 進数で書いとくと 0x0F ね。

ここでこの 0x0F というアドレス値を何かの変数に保存したとする。

`int* b=&a;`

そうするとこれもどこのアドレスに確保されているわけや。つまりにもアドレスがあると…。今回の例だと b にある。16 進数になると 0x3F。

これも変数に入れるならどつかに入っていることになる。

`int*c=&b;`

NULL									
					a=5				
				b=0x0F					

で、ポインタのポインタの話だが、ポインタのポインタを扱う時の主役はアドレス値なのね。今回で言うとりのこと。

DirectX11のCreate系の関数とかでよくある、あの最後の引数にポインタのポインタを渡すってのは

ポインタ型をbとすると

Createなんとか(1,2,3,&b);

ってなるわけ。この例であれば最後の部分にbのアドレス(この例では0x0F)が入っている。で、例えばCreateなんとかの場合はb=NULLだったりするわけ。

うん、でそのアドレスを渡すということはCreateなんとか関数内で適当にメモリを確保して作られたオブジェクトのアドレスを変数bに放り込めるってことなんだ。

わかりにくいけな?

b=NULLにしてると仮定する。

で、下のような関数がある。最後の引数がポインツなのね。外側からポインタ型の変数のアドレスが渡される…。関数内でこのアドレスの指す内容を書き換える。Createの場合であれば大抵は内部でメモリが確保され、確保したアドレスを以下のように値として渡してやる。

```
void Createなんとか(int a, int b, int c, int** d){  
    *d=new int;  
}
```

newは渡された型に応じて適当にメモリを確保し、そのアドレスを返すものだ。そのアドレスが、ポインタのポインツで渡されたdに入っている。いや、*dなので、アドレスdが指し示す先(ここもアドレス)に、新しく確保したメモリアドレスを渡しているのだ。

結果として

```
int* q=NULL;  
Createなんとか(1,2,3,&q);
```

とやれば、本来NULLのアドレスに、きちんと確保されたアドレスが入って来るってことです。

参照

まあ、本来はこんなもの知らなくてもプログラム組めるわけですが、僕が結構この機能をしれっと使っちゃうので、書いとかんと混乱すると思い、書いておきます。別にこの機能を使えつてわけじゃない。とりあえず紹介だけしておこう。

基本的には参照というのはポインタみたいなもんで、本来的にはアドレスを指し示してはいるんだが、若干使いやすくなってるし、アドレスを意識しない作りになっている。

使い方はこうだ。

型 & 変数名 = 参照先；

である。で、変数は変数として使える。&(アンパサンド)がポイントやね。どういうことがどういうこと…

```
int a=10;
```

```
int& b=a;
```

これで、bを見ると10と書いてありb++とすると、当然bの中身は11になるのだが、aの中身も11になる。どういう事がというと、通常であればb=aというコードはaの内容をbにコピーするという意味になるが、この参照というやつは実はb=aと書いた時点では、bはaと同じになっているわけである。

そんなん何に使うの？って思うかもしれません、例えば関数の中身で、引数の値をいじつてしまいたい時等に使んだよ。

とりあえず参照の説明はここまで、詳しくは実際に使う際にでもまたお話しします。

C++のクラス

C++のクラスも基本的にはC#のと似たようなもんです。…が！

```
class A{  
    private:  
        int life;  
    public:  
        void Punch(int damage);  
}; //←セミコロン忘れないように
```

こんな感じでクラスを作ります。まず、しょーもない所で「マらない」ように細かいことを言っておくと **C++の場合はクラスの宣言の最後にはセミコロンがつきます**。これを忘ると30分くらいを無駄に消費するので気をつけましょう。

もうひとつC#と違う部分は、`private`と`public`指定子の書き方ですね。確かC#では

```
public void Punch(int damage);
```

って書いてあったと思いますがC++の場合は上のクラスを見て分かるように`private:`と書くと、次に`public:`と書くまでは`private`扱いであり、逆に`public:`と書くと、次に`private:`と来るまでは`public`扱いになります。もちろん`protected:`もあります。

ちなみにデフォルトは`private`です。

慣れるまでは結構間違えると思うのですが、慣れるとこっちのほうがラクですよ。

その他

include ★

C++では外部のファイルを使用する時は`#include`で呼び出すことで使えるようになります。

基本的には

```
#include "ファイル名"
```

と

```
#include<ファイル名>
```

で必要なファイルをインクルードしますが、通常は「ヘッダファイル」と呼ばれる`.h`で表されるファイルをインクルードします。例外としてSTL系には`.h`の部分がありませんが殆どが`.h`です。

C#における`import`みたいなもんです。ヘッダファイルの先頭には

```
#pragma once
```

と書くことが多いです。これは「一度しか読み込まない」という指定になります。

テンプレート

C#でジェネリクス使ったことのある人はC++で戸惑うかもしれません。用語が違うだけです。C++では「テンプレート」ってのがジェネリクスに相当します。

これはおおい活用していくので今は用語が違うと思って下さい。

STL

C#で「list<型>」だの「dictionary<型,型>」だの使ってた人も多いと思いますが、これをC++でやろうとすると、STL(Standard Template Library)というライブラリセットを使用することになります。

が、これもここでガツツリやらずにおいおい必要なときにしれっと使っていきます。

sizeof(型もしくは変数)

これ単品で説明するのは冗長な感じもするけど、やたら出てくるし結構大事なやつなので軽く説明しておくと、このsizeofは演算子の一つで、sizeof(何か)の「何か」のサイズをバイト数で返すものだ。

演算子って言うともしかしたら変に感じる人もいるかもしれないが、変に感じるのならば「型のサイズを返す関数」とでも思っておけばいい。とにかくC++ってのはハードウェアに「メモリの塊」を投げることが多く、そいつのサイズがどれくらいなのかっていうのが重要なのだ。真剣に勉強してれば、そのうち分かるようになると思うが、今はそのくらいの理解で十分だろう。

ここでのC++の解説はここまでです。しっかりとC++について知りたい人は友達に聞くか、放課後に僕に聞きに来るか、自分で調べるかしてください。授業中でやるのはここまでです。さて、いよいよ DirectX11 を使えるようにしていきます。

DirectX 11 の初期化

正直な話 DirectX11 は初期化だけでめんどくさい。色々と段階を踏んでから初期化が完了するわけだが、DirectX9 をやってる人にとっても新しい概念とか出てくるので、それなりにハードであることは理解しておいてくれ。

そのために、事前にさらっとだけ、用語と流れの解説をやっておく。細かくはコードを書きながらにはなるが、全体の概要が見えておいたほうがわかりやすいとは思う。

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、おいおい分かつくるけど、とにかく全体的な管理をやってくれるクラスである。

- 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。つまりところ GPU 先生にコマンド投げる役割を持っている。

- 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。ややこしいので詳しくは後述する。

- 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。

- 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だ。通常の出力先はウインドウの中なわけだから、ウイ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。

- 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ぬのだが、そもそもウイ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。

※ID3DとかIDXGIの先頭のIはインターフェイスクラスを意味している。C++言語はC#のようにインターフェイスクラスなど存在しないため、このように先頭にIを付けることによってインターフェイスクラスであることを示すことが多い。

※GIは何の略かというと Graphics Infrastructure の略でグラフィックスのハードウェアに近い部分を担当しているという意味になっている。



ね？かなりハードウェア寄りでしょう？

概略はここまで、早速初期化コードを書いていこう

①デバイス&デバイスコンテキスト&スワップチェーンの作成

いろいろとややこしいんだけど、DX11はまだ優しさが残っているようで、とりあえずデバイスの作成、デバイスコンテキストの作成、スワップチェーンを一度に作ってくれる関数があるんだ。正直これがないとマジやってらんねえのね。

で、定義はこうなっているのよ

[http://msdn.microsoft.com/ja-jp/library/ee416033\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416033(v=vs.85).aspx)

HRESULT D3D11CreateDeviceAndSwapChain(

IDXGIAdapter *pAdapter, //アダプターを渡す…とりあえずNULLでいいよ。

```

D3D_DRIVER_TYPE DriverType, //とりあえずD3D_DRIVER_TYPE_HARDWAREにしつく。マシンによってはうまくいかないので、その場合はhardwareをREFERENCEにしてくれ
HMODULE Software, //NULLしかダメ(D3D_DRIVER_TYPE_SOFTWAREの時にのみ非NULL)
UINT Flags, //とりあえず0で
CONST D3D_FEATURE_LEVEL *pFeatureLevels, //D3D_FEATURE_LEVEL_11_0がはいつたポインタ
UINT FeatureLevels, //今回フィーチャレベルは11オンリーなので1にしつく
UINT SDKVersion, //D3D11_SDK_VERSIONと書いてくれ
CONST DXGI_SWAP_CHAIN_DESC *pSwapChainDesc, //詳しくは後述する
IDXGISwapChain **ppSwapChain, //スワップチェインの実体を受け取るためのポインタへのポインタ(後述しる)
ID3D11Device **ppDevice, //デバイスの実体を受け取るためのポインタへのポインタ
D3D_FEATURE_LEVEL *pFeatureLevel, //NULLでよし
ID3D11DeviceContext **ppImmediateContext //コンテキストの実体を(r
);

```

うんざりだね。ところで毎年いるんですけど、これをコピペして「センサー動きません」とか言うのがいるのね。もうね、アホかとバカかと。お前ら何年生かと。もっかい入学しなおしてこいと。定義ってゆーてるやん。「定義」の言葉の意味がわからんのかと。中学校からやり直せと…ハアハア。そういうこと言う人はDX11向いてないんで進路を変えたほうがイイですよ。

専門的なことはともかく説明の必要があるのは「ポインタへのポインタ」かな? C/C++において、API側から情報を取得したいときには、変数のポインタを渡して、そこに書き込んでもらうのが多い。C#でいうところの out とか ref みたいなもんだよね。

で、ポインタへのポインタを使うパターンってのは、大体インターフェイス系(頭にIがついてるクラス系ね)の実体を API に作ってもらってそれを取得するときに使うんだ。

だからこいつが出てきた時は特に悩む必要はないんだ。むしろ API 側で作ってくれるんだから「APIさんありがとさん」と言って、ポインタへのポインタを投げ込めばいいわけ。

C++やったことない人にとってこのやり方は、慣れないと思いませんが、こういうやり方でポインタ受け取るのは DirectX 系ではほぼジョーシキになってるので覚えておきましょうね。

で、上の説明で「とりあえず~でいいよ」ってのは、リファレンス見ながら判断して書いてます。マシンによってはうまくいかないこともありますので、上の説明通りにやればかならずうまくいくとは限りませんのでアシカラズ。

ただし、いくつかはきちんと設定しなければならない。DXGI_SWAP_CHAIN_DESC もそのひとつだ。まずはリファレンスを見ておこう。

[http://msdn.microsoft.com/ja-jp/library/bb173075\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb173075(v=vs.85).aspx)

はい、いつもどおり定義を見て行きましょうか

```
DXGI_SWAP_CHAIN_DESC {  
    DXGI_MODE_DESC BufferDesc; // 後述  
    DXGI_SAMPLE_DESC SampleDesc; // 後述  
    DXGI_USAGE BufferUsage; // DXGI_USAGE_RENDER_TARGET_OUTPUT にしとく  
    UINT BufferCount; // バックバッファ数なんて 1 でいいよ  
    HWND OutputWindow; // ウィンドウのハンドル  
    BOOL Windowed; // いきなりフルスクは困るんで true にしとこう  
    DXGI_SWAP_EFFECT SwapEffect; // 放置で可  
    UINT Flags; // 放置で可  
}
```

まあ…こんな感じ。いつもどおりで、

DXGI_MODE_DESC を見てみると

[http://msdn.microsoft.com/ja-jp/library/bb173064\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb173064(v=vs.85).aspx)

まあ、色々とありますね、Width と Height はわかるでしょ？…君たちなら分かるでしょう…ねえ。

で、RefreshRate も構造体になってるのが、結構面食らうね。これは最大値と最小値を指定できるんだ。マニュアルはものごつい分かりにくいけど、とにかくフレッシュレートの最大値が Numerator なので、Numerator=60, Denominator=1 にしとけばいいよ。

次に Format なんだけど、こいつは DXGI_FORMAT_R8G8B8A8_UNORM にでもしとこう。実は UNORM の意味がよくわからへんねんけど、U は符号なし(Unsigned)の U なんだろ…でも NORM って何？法線かと思ったけど、多分これは正規化整数って意味なんだろうな。まあ、深く考えずにこいつにしとこう。

次に SampleDesc の話だけど、こいつはカウントとクオリティを指定するだけ

Count はマルチサンプリング数の話で、Quality は品質です。アンチエリの必要がないなら、Count=1, Quality=0 になりますが、そのうち汚く思えてくると思うんで、ここで品質調整するってことは覚えておいてくれ。

と、まあ、ここまで長々と説明を書いてきたんだけど、このツソ面倒な設定を見かねたのがマイクロソフトがチュートリアルを用意してくれている。

[http://msdn.microsoft.com/ja-jp/library/bb172485\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb172485(v=vs.85).aspx)

はい、

くれぐれも言っておくと、ネット上のサンプルなんか信用するべきではない。どんなに素晴らしい人のものでも丸コピペなど許されない。世のエンジニアには、どつかのソースを丸コピして使った挙句、「グダグダしたら『サンプルがそうなっていたから』と苦しい言い逃れするんだけど、もうね、お前プロかと、(•▽•)カレ!!」と。

で、そういうサンプルの中でまだ、信用できるのはベンダーが発行しているものである。こ

いつも、サンプルというより、ホワイトペーパーやね。今回のDX11のベンダーはマイクロソフトである。そしてこの資料はマイクロソフトの資料である。

これを参考にして、面倒な初期化処理にお役立て下さいって程度。

ただ、これでうまくいかないこともあるし、用途によって変更しなきゃいけないところも多い。

丸コピだけはやめておいたほうがいい。理解せずにコードだけ写経すると、あとで手痛いしっぺ返しを食らうであろう。実際、このコードはDX10用ですしつ。

個人的には「パラメータ多すぎだろ!!」って思っている。別に走査線の順序とか、リフレッシュレートの最小値とか正直どうでもいい。もう少しシンプルにならんもんかなとは思うんだわあ…。

まあ、こんなところで詰まってたら先に進めないので、スクショだけ見せてあげるんだからねつ

```
DXGI_SWAP_CHAIN_DESC scdesc={};  
scdesc.BufferCount=1;//バックバッファいくつもいらん  
scdesc.BufferDesc.Format=DXGI_FORMAT_R8G8B8A8_UNORM;//?  
scdesc.BufferDesc.Width=800;//暫定的幅  
scdesc.BufferDesc.Height=600;//暫定的高さ  
scdesc.BufferDesc.RefreshRate.Numerator=60;//(↑^o^)↑  
scdesc.BufferDesc.RefreshRate.Denominator=1;//(↑^o^)↑  
scdesc.OutputWindow=hwnd;//ウィンドウハンドル  
scdesc.SampleDesc.Count=1;//アンチエリなし  
scdesc.SampleDesc.Quality=0;//アンチエリなし  
scdesc.Windowed=true;  
  
IDXGISwapChain* swapchain;  
ID3D11Device* device;  
ID3D11DeviceContext* context;  
D3D_FEATURE_LEVEL f1=D3D_FEATURE_LEVEL_11_0;  
  
HRESULT result = ::D3D11CreateDeviceAndSwapChain(NULL,  
    D3D_DRIVER_TYPE_HARDWARE,//HARDWAREで通った  
    NULL,//とりあえずNULLかな  
    0,//とりあえず0やね  
    &f1,//フィーチャレベルは11オンリー  
    1,//だから1  
    D3D11_SDK_VERSION,//これ固定  
    &scdesc,//これがいいんだろ?  
    &swapchain,//スワップチェインありがとさん  
    &device,//デバイスありがとさん  
    NULL,//いらん  
    &context); //コンテキストありがとさん
```

べつべつにアンタのためじゃないんだからねつ!授業が進まないから仕方なくスクショ置いておくんだからねつ!!

めんどくさかったー。あとやっぱりグラボがショボイとD3D_DRIVER_TYPE_HARDWAREじゃ動かないみたいなんで、初期化失敗したらD3D_DRIVER_TYPE_REFERENCEにしてちょ。で、実行結果のresultにS_OKが入っていたら成功であります。

②レンダーターゲットビューの作成

残念だが、初期化はまだ終わりじゃない。もうちょっとだけ残っているのだ。あーもーめんどくせー。

何が残っているのかというとレンダーターゲットビューってのを作らなきゃいけんのだ。以前にも話した、レンダリング結果の出力先やね。

CreateRenderTargetView を使うわけだが、定義はこうなっている。

[http://msdn.microsoft.com/ja-jp/library/ee419800\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419800(v=vs.85).aspx)

```
HRESULT CreateRenderTargetView(
```

```
    ID3D11Resource *pResource, // レンダーターゲット(バックバッファの塊)
```

```
    const D3D11_RENDER_TARGET_VIEW_DESC *pDesc, // NULL でええねん
```

```
    ID3D11RenderTargetView **ppRTView // レンダーターゲットビュー
```

```
);
```

こうなっておりやす。へえ。

で、レンダーターゲットビューをつくりたいんだが、コイツの説明を読むと、

「レンダーターゲットを表す ID3D11Resource へのポインターです。このリソースは、
D3D11_BIND_RENDER_TARGET フラグを使用してあらかじめ作成しておく必要があります。」って書いてあるんで、ちょっと一旦戻って SWAP_CHAIN_DESC 指定の部分に

```
scdesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
```

と書いておいてくれ。

そうすりやあとは割と簡単だ。で、第一引数にレンダーターゲットを指定するのだが、上の設定によりこいつはスワップチェインから持ってこれる。

```
ID3D11Texture2D *buffer;
```

```
swapchain->GetBuffer(0, __uuidof(ID3D11Texture2D), (void**)&buffer);
```

バックバッファつても結局は「絵」なので、方は ID3D11Texture2D となる。こいつをバックバッファリソースとしてスワップチェインから取得するのだ。

こういうバイナリの塊を取得するときによく使われるのが GetBuffer メソッドがある。

[http://msdn.microsoft.com/ja-jp/library/ee421954\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee421954(v=vs.85).aspx)

要はこのスワップチェーンに関連付けられてるサーフェスのバイト列を表すポインタをとってくるわけなのだが、わかりづらいのは2番目の引数だ。こいつは Direct3D 特有なのだが、どうも ID を指定しなければいけないんだが、その ID を知るには __uuidof を使う。

__uuidof 演算子は敷に割り当てられた GUID を取得するんだそうだ。これってこういう時にしか使わないんだが…。まあ __uuidof(GUID を知りたい型) でいい。

で、最後の引数にバイトの塊が飛んでくるのでこれを受け取る。また、void*ってのに面食らうと思う。voidって型がないのに…そのポインタ?って思うかもしれないが、これは型を特定しないただのバイト列を指すためのポインタってわけ、言い方を変えると何にでもなるってこと。つまりキャストしさえすればいい。

あ…キャストわかりません?おかしいですね。C#でもjavaでもやってるはずなんですかね?同じなんですかね。

(変換先の型)型を変換したい変数

これだけです。変換したい変数の前に、変換したい型をカッコでくくるだけです。

だから今回の場合、バッファ型を void**型にして渡してあげたいのですから

(void**)&buffer

というふうに渡してあげることができます。

なお、C++においてはこのようなC言語的なキャストではなく、ちょっとヤヤコシイキャストが好まれますが、それはまた別の機会にお話いたします。

で、スワップチェインからバッファリソース(buffer)を持ってきたと。あとはこれを CreateRenderTargetView の引数にぶち込めばレンダーターゲットビューの出来上がりだ。

```
ID3D11RenderTargetView* rtv;  
result=device->CreateRenderTargetView(buffer,NULL,&rtv);
```

はい、上手にできました~かな?これで一応ここは終わりなんだけど、最後に buffer はもう用済みなので、Release してやっとく。

```
buffer->Release();  
Releaseしても参照カウンタが減ってるだけなので、実態はなくなつてないので、安心していい。
```

あとは、作成したレンダーターゲットビューをパイプラインにバインド(要はレンダリング結果をきちんとバッファに出力)するために、

context->OMSetRenderTargets 関数を使う。これがんたん

三つ引数があるが、1つめはレンダーターゲット数なので、1を指定する。2つめにはレン

データーゲットビューへのポインタ。最後にはNULL入れときゃいい。

おめでとう。これで初期化は終了だ。

画面クリアすつか

ひとまず、きちんとレンダリング結果がバインドされるのか、レンダーターゲットビューに
対しての処理が反映されるか確認してみよう。

画面クリアするにはcontextのClearRenderTargetView 関数を使う。これは簡単。

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

見ればわかるね。

これで、確かに画面…つまりレンダリングイメージはクリアされるんだけど、あくまで
バック/ティッファなので、こちらにはみえない。

画面をフリップするにはPresent 関数を呼ばなければならぬ。

ここらへんはDX9と同じやね。

[http://msdn.microsoft.com/ja-jp/library/bb17457b\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb17457b(v=vs.85).aspx)

とりあえず1枚しか使ってないので、即座に表示したいので、第一引数は0、第二引数も、特
に指定無しなので0にします。

つまり、

swapchain->Present(0,0);

はい、どうでしょうか？これで画面の色が変わったのではないでしょ？

やっと…やっとスタートラインです。こつからです。本番はこつからです。頑張っていきま
しょう。

ポリゴンを表示しよう

最初に言っておくと、DX11はポリゴンを表示するのさえ面倒なのだ。何が面倒なのかといふと固定パイプライン廃止したためにいちいちシェーダ書かなきゃいけない。

シェーダー…だと？

説明しよう。シェーダーとは(基本的には)ライティング・シェーディング・レンダリングを実行するためにグラフィックリソースに対して使用するソフトウェア命令である。

<http://ja.wikipedia.org/wiki/%E3%82%B7%E3%82%A7%E3%83%BC%E3%83%80>

よくわからんね。まあつまりですね。DXLibだのDX9だのでは、特に何もしなくても、例えば三角系ポリゴンを描画する際には、頂点情報投げさえすればポリゴンは描画されるし、ライティング設定をしさえすれば陰影がつくし、カメラとビューポートを設定しさえすれば画面にモノが表示されていた。←これを固定パイplineという。

しかし時代は進み、3DCGの表示についての開発者の要望が非常に多岐になり始めたため「んじゃあもういいよ！自分で全部設定計算やってや！もう手伝わへん！」となり、プログラマが色々と楽しいことをやれるようになつたのだが、面倒くさいことまでやらなきゃいけなくなった。

その色々と楽しい部分をプログラミングするためにシェーダに対して命令を出せるようにした。シェーダに対してC言語みたいなプログラミングで命令できるようにしたのが、現在のHLSL(High Level Shader Language)である。OpenGLのGLSLも同様である。

プログラマがこれまでブラックボックスだった表示部分をいじれるということで、プログラマブルシェーダということもある。

シェーダはDirectX8からあり、DirectX11ではシェーダーのバージョンが5.0になっている。かなりいろいろな表現ができるようになっている。

さて、さきほどの説明で「基本的には」ってこっそり書いたのにはワケがある。それはもう少し先に書く事になるが、

GPGPU(General Purpose Graphic Processing Unit)

として使用するためだ。かなり最後の方にしか出てこないとと思うし、かなり難しいので、今は基本的な頂点シェーダ、ピクセルシェーダからはじめて行くことにする。

まずは、実際どういった現場で使われているのかという話をしようか。アニメ業界でも使用されている。現在のアニメはキャラクターが3Dで作られており、最終レンダリング時にアニメっぽくし、手書き感を追加することも珍しくない。そういう時代になってきた。例えば、CG WORLD(2012/06)の特集だが

Character Anatomy

03 Facial Rigging / リグ(フェイシャル)解剖

フェイシャルリグも同様に『ONEPIECE 3D 変わらチエス』時からさらにバージョンアップされている。なお、『スイートブリキュー』まではアニメのキャラクターのテイストを損なわないようなるべく正面からのカメラワークが用いられたが、本作では積極的に多様なアングルが用いられた。そのため、これまで以上に目・口・鼻の位置調整、シルエットの変形などの自由なコントロールが求められた。

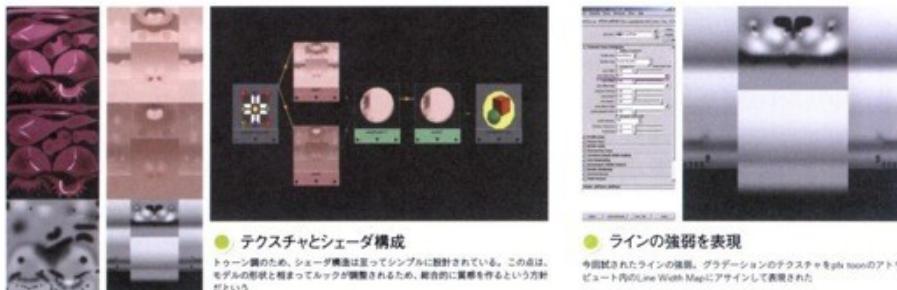
● フェイシャルリグとコントロールパネル
左がフェイシャルリグの構造。右がFacial Selector(コントロールパネル)。Slider, Area mode, Soft mode, Selectorと直感的にコントロールできるのがポイントだ。

● ターゲットシェイプ
ターゲットモデルもこれまで以上に多く作成された。頭部の各部位を直接操作して上げ下げなどが新たに追加された要素だというが、BDバーン上上のシェイプが準備されている

● アニメへの対応
目・鼻・口の形状を変化させた例。本画像を見るだけで自由度の高さが察せられる。アニメーターはモーリングをする慣習で作業を行うというが、実際アニメーションにはアニメ制作会社ならではの効率チェックがあるため、必須の仕組みだと答える

04-1 Texture / テクスチャ/質感解剖

キャラクターの質感はMayaのトゥーンシェーディングをベースにしている。アニメでのCGを活用する上で質感関連の技術は大きな課題の一つであるが、トゥーンの表現を突き詰めようと試行錯誤を繰り返し、徐々にバージョンアップしている途中だとか。本作ではグラデーションのテクスチャを活用してラインの強弱を調整することが試された。



095

左上がそのままレンダリングしたものだが、まあヘタをすると子供が泣き出しそうなレンダリング結果である。僕らは3DCGに慣れてるからいいが、このままではお子ちゃまアニメとしては使えない。なんというか、ミクダヨーとまでは言わないが、やっぱりちょっと残念なのだ。

これをアニメっぽく見せるためにセルシェーダ等を使用しお子ちゃん(大きなお友達)が喜ぶ作りにしていくのだ。

ちなみに、ゲームクリエータになりたいのならばCGWORLDの特集は定期的にチェックしておいたほうがいい。現在業界で使用されている技術がわりかしマニアックなレベルで特集されている。

余談だが2013年10月号は「リアルタイムシェーダ入門」だ。1470円(税込)とそれなりにお値段がアレだが本気でゲーム会社を目指しているのならば面接時のネタのためにも読んでおいたほうがいいとは思う(別に立ち読みでもいいし、実は教務室にバックナンバーがいくつもある…が、プリキュア回はおつきなお友達が買い占めちゃったせいで入手困難である)

こういうおつきなお友達のせいだね!!!!↓

<http://purisoku.com/archives/6596482.html>

まあ、こういうアニメの現場でなくとも、もちろんゲームでも使用されているし、MMDでも使用されている。ちなみにMMDではデフォルトでトーンシェーダがかかっているのと、スフィアマップ等にも対応しているため、割と見た目が違う。



左がそのままレンダリング(メタセコ)で、右がMMDである。全然見た目が違うだろう?右は輪郭線を表示し階調を調節している。ライティングの違いはシェーダのせいではない。デフォルト設定のせいだ。

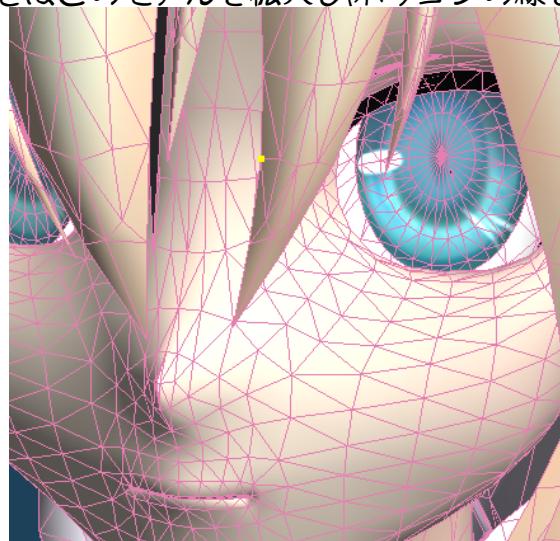
これだけでもそれなりに頑使うんだけど、世の中の最先端はそれどころじゃないので、そこに少しでも近づいていくと勉強していくのがこの授業の目的なんですね。

三角ポリゴン表示

んじゃあ早速最初の第一歩、三角ポリゴンを表示してみましょうか?えー、そんなのくそつまんねー。確かにつまらん。が、今のところ世のポリゴンモデルってやつは、結局は無数の三

角形の集合で出来ているのだ(将来的には変わるかもしれないがね)。

ちょっと見てみよう。さきほどのモデルを拡大し、ポリゴンの線をピンク色にしてみよう。



どうだろうか? ありとあらゆる部分が三角で分割されていることがわかるだろう? 何はなくともこの三角ポリゴンからなのだ。

三角形はなにから構成されているのだろうか? そう、三つの頂点、三つの辺から構成されている。三角形であることがわかつていさえすれば、三頂点さえ与えれば辺はわかるよね。だからまず三頂点を作ろう。

ビューポート設定

画面に物を表示するためには、最終的な画面の情報が必要です。つまり、画面の解像度だのなんだの、えっ? レンダーターゲットちゃうのん? いやいや、レンダーターゲットはあくまで仮想画面なんや。

実際の画面の情報が必要なんや、ほぼ「ウインドウ」の情報と一緒になんやけどな、まあ「ウインドウ」とは独立していると思っていい。何でかっちゅうと、一つのウインドウ内に複数のビューポートを設定することだってあるわけ。ほら、二人対戦とか四人対戦で画面割りたい時とかさ。

今回はひとつの画面にのみレンダリングする予定なので、ビューポートは一つでいい。

ちなみに関数は `RSSetViewport` って関数。描画に関する部分なんで `Context` の持ち物やね。

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

こういうやつね。

パラメータ見たらわかると思うけど、バインドするビューポートの数とか指定できる。そして第二引数はポインタ渡し & 複数形の `S` がついてる定義だ。つまり、配列で渡されることを想定している。

が、今回もひとつなので、`NumViewport` については1でいいし、2番目の引数に関しては

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

これ見ながら考えてくれ。

通常は、左上から幅、高さ 1つだけに使うので、0,0,幅(今回は 800 かな),高さ(今回は 600 かな)って感じになっている。最後のやつの `MinDepth` と `MaxDepth` は `Min=0.0f, Max=1.0f` でいい

いんじやないでどうか。深度に関してはもう少し進んだら、詳しくお話ししますので、このように設定して下さい。

一つをさっきの関数に放り込めばオッケー。

(ちなみに、RSってのはラスタライザーステージの略。格好いい!名前の「ラスタライザー」については後述します。)

頂点定義

頂点を指定するのに便利な構造体はD3DXVECTOR3構造体である…がDirectX11では廃止されている(DX10まではあった)ので、もし使いたければd3dx10.hをインクルードしておこう。

この構造体は名前を見てわかるように三次元ベクトルを表している。三次元ベクトルと言うと、ベクトルにしか使っちゃいけないとと思うかもしれないが、座標にも使える。「位置ベクトル」だと思っておけばいい。

*なんでDirectX11からD3DXVECTOR3が消えたのかというと、

[http://msdn.microsoft.com/en-us/library/windows/desktop/ff729728\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff729728(v=vs.85).aspx)

英語で申し訳ないが、どうやらかつてのD3DXMath(D3DX9,D3DX10に含まれてた)は推奨されず、代わりにDirectXMathを使えということらしい。割とDirectXまわりはバージョン進むごとにライブラリの機能の統廃合や移動が行われてしまうため、色々と見ておいたほうがいい。…っていうかホントにDirectX9やってた人間が面食らう仕様にしてるな。

で、このDirectXMathはどうから取ってくるかというと、

[http://msdn.microsoft.com/en-us/library/windows/desktop/ee418725\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ee418725(v=vs.85).aspx)

を見ればわかるが、WindowsSDKに同梱されているらしいため、こいつをWindowsStoreからダウンロードしてインストールしなければならない…流石にそこまでやる必要があるとも思えないないので、D3DX9かD3DX10をインクルードしておこう。もし、**使ってみたい人は各自WindowsSDKをインストールしよう。**インストールの際に権限で困ったら呼んでくれ。

WindowsSDKはココ

<http://www.microsoft.com/en-us/download/details.aspx?id=8279>

WindowsSDKをインストールしたいのは山々なんだが、このインストーラがWebインストーラな上、色々とアホ挙動を示すために妙に気を遣う。ので、心に余裕がない間はあまりオススメしない。

ここからは仕方ないので、D3DX10をインクルードして、D3DXVECTOR3を使ってる状態の話をする。

この構造体1個につき(x,y,z)の情報を持っているため、この構造体1個あたりの大きさはfloat三個分である。ちなみにこいつはコンストラクタで三つ引数をとってXYZ設定してくれる。

```
D3DXVECTOR3(X 座標,Y 座標,Z 座標)でいい。つまり、  
D3DXVECTOR3 V()={  
    D3DXVECTOR3(1.f,1.f,0.1f),  
    D3DXVECTOR3(1.f,-1.f,0.1f),  
    D3DXVECTOR3(-1.f,-1.f,0.1f),  
};
```

あ、現状のまま新しい構造体を使用する方法があります。

Xnamath.h をインクルードしてください。

XMFLOAT3ってのがありますんで、それ使ってください。

つまり、

```
XMFLOAT3 V()={  
    XMFLOAT3(1.f,1.f,0.1f),  
    XMFLOAT3(1.f,-1.f,0.1f),  
    XMFLOAT3(-1.f,-1.f,0.1f),  
};
```

でいいんじゃねーかと。もちろん、こんなのは結局は float の塊なので

float V()={

```
{1.f,1.f,0.1f},  
{1.f,-1.f,0.1f},  
{-1.f,-1.f,0.1f}
```

};

でも構わないで、この float の塊をどうするのかというと、Direct3D 用のバッファに食わせて、そのバッファを頂点バッファとして登録してしまうっちゅうわけよ。

どういうことがわかるかな？ float の固まりじゃどう使っていいのかわからない。それを「頂点バッファ」として登録することで Direct3D はさっきの float 情報を頂点として使用することができるようになるというわけ。

Direct3D 頂点用のバッファを生成

で、まずは食わせるためのバッファを作成するんだが、その関数が

device->CreateBuffer()

[http://msdn.microsoft.com/en-us/library/windows/desktop/ff476501\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff476501(v=vs.85).aspx)

英語だけど、最後の引数に確保したバッファへのポインタが渡されることがわかるね。問題は最初の二つだ。

最初のは、D3D11_BUFFER_DESC 型だ。こいつは入力されるバッファについて説明するものなのだ。

```
struct D3D11_BUFFER_DESC {  
    UINT ByteWidth; // バッファサイズ(3*3*sizeof(float))  
    D3D11_USAGE Usage; // D3D11_USAGE_DEFAULT でいいよ
```

```
UINT BindFlags;//http://goo.gl/ix9Vidでも見とけ!(VERTEX_BUFFER 選択しとけ)
UINT CPUAccessFlags;//CPU アクセス必要ないから 0 で
UINT MiscFlags;//0 でよくね?
UINT StructureByteStride;//0 でいい…かな?
};
```

これを参考に設定してね。あくまでも自己責任で選択してね。自己責任ゆーてるやろ?

さて、ともかく設定していこう。上に書いてるように設定してみればいい。もし分からなければ

[http://msdn.microsoft.com/ja-jp/library/ee416041\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416041(v=vs.85).aspx)でも見ながら考えてくれたまえ。

バッファサイズが $3 \times 3 \times \text{sizeof}(\text{float})$ ののは、頂点自体が XYZ の三要素なのと、これが三角形なので 3 倍している。 $\text{sizeof}(\text{float})$ は float 型のメモリ上におけるサイズを表す。

頭の良い少女は気づいていると思うけど、これは $3 \times \text{sizeof(XMFLOAT3)}$ でもかまわない。

で、`CreateBuffer` の引数にはもうひとつワケがあるのがありますよね。それは…

[http://msdn.microsoft.com/ja-jp/library/ee416284\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416284(v=vs.85).aspx)

`D3D11_SUBRESOURCE_DATA` って構造体を使う。実際こいつの中に頂点情報を放り込むことになるのだが、そいつは

```
D3D11_SUBRESOURCE_DATA {
    const void* pSysMem; // 対象となるメモリアドレス(今回は頂点)
    UINT SysMemPitch; // ピッチなんて float 3 個ぶんかな
    UINT SysMemSlicePitch; // これも今は意味ないっぽい? 0 にしちゃか
}
```

こういう定義になっている。

これも似たような感じで、まずは変数作って…今回は `pSysMem` にのみ値を入れてみればいいよー。

とにかく、こいつらを使って Buffer を生成する。

```
result = device->CreateBuffer(&bufdesc,
    &subresource,
    &buff
); // ここで頂点情報のバッファを得られたわけだけど…
```

で、Buffer を生成する。このままでは単なるデータの塊です。これを使えるようにしなければならない。そのためには、入力アセンブラーステージに頂点の塊をバインドする必要がある。

そのための関数が `IASetVertexBuffers` なんだが、こいつの定義は

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

となっていて

```

void IASetVertexBuffers(
    UINT StartSlot,//開始スロット(今回は0でいい)
    UINT NumBuffers,//対象バッファの数(今回はひとつなので1)
    ID3D11Buffer *const *ppVertexBuffers,//さっきのバッファを放り込む
    const UINT *pStrides,//頂点ひとつ当たりの大きさを表すポインタ
    const UINT *pOffsets//オフセットしないので0を示す変数へのポインタ
);

```

だいたいこんな感じですわ。あ、これ、定義だからな。コピペすんなよ。動かなくなつても知らんよ。

ちなみにIAってのは入力(Input)アセンブラステージ(Assemblerstage)の略やね。どうでもいいです。こんなとここだわらぬreiに限りますが、ちなみにアセンブラステージに関しては

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

に色々書いてますが、うん、まあまだ気にしなくていいです。気にすると色々と終わりません。さらっと見ておいて、なんとなくのイメージを頭に入れておいて下さい。イメージ出来ない人は、こだわらずに次に進みましょう。IAほにやららはコンテキストの持ち物なので

```
context->IASetVertexBuffers(0,1,&buff,&strides,&offsets);
```

こんな感じにしておきます。ちなみに、UINT*stridesのように、わざわざポインタで指定する事を疑問に思う人も多いと思いますが、この変数名の例をみると最後に複数形のSが付いていることがわかると思います。

ここから推測できるようになって欲しいのですが、こいつらは本来複数のものを(配列)指定できるようになっていると考えましょう。このような部分に注目できるようになります。

大体C/C++言語において、わざわざポインタ渡しをする理由とは

- ①ライブラリ側から値を受け取る用
- ②配列を渡す用
- ③でけえ構造体を渡す用

なので、UINTのような、ポインタ自体と変数サイズが変わらないような変数がポインタ型になっている場合は①か②になるでしょう。しかしここでは、strideおよびoffsetはこちらで指定するようになっているようです。ここから考えられることは、このUINTは配列を想定しているであろうということです。

まことにかくuint型変数に値はりこんでおいて、そのポインタを突っ込めばいいんです。

ということで今回は両方共ひとつ分しかいらんので、

```
UINT strides=sizeof(XMFLOAT3); //もしくは3*sizeof(float)
```

```
UINT offsets=0;
```

```
context->IASetVertexBuffers(0,1,&buff,&strides,&offsets);
```

でもいいし、まあ、本来の要求を考えるのならば

```

UINT strides[1] = { sizeof(XMFLOAT3) };
UINT offsets[1] = { 0 };
context->IASetVertexBuffers(0, 1, &buff, strides, offsets);

```

と書くべきかも知れん。まあどっちでもいいよ。ともかくこれでさっき作ったバッファをDirect3D側が分かること態にすることができたわけだ(正確に言うとアセンブラーステージとつながったってこと)



もうちょっとだけ、もうちょっとだけ我慢してけれ。

で、既にD3Dにバッファを渡しているんだが、この頂点情報をどのように読み取ってポリゴン化していくのかの指定をしなければならない。今回は三角形ひとつだけなので、この中から

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

三角形リストを選択します。つまり

D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST

こいつを使う。この指定をD3Dに教えてあげるにはデバイスのIASetPrimitiveTopologyって関数を使う。トポロジーってのは、まあ、頂点同士をどういうつなぎ方するかってことだと思つといいでいい。数学的にトポロジーっていうと、円も三角形も同じとみなすような学問なんだけど、イメージと若干異なるので、数学的なトポロジーは考えないほうがいいだろう。

ということで、

context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

てな感じで指定する。

こ、これで終わり!?

もうゴールしてもいいよね…!

あかん!! ゴールしたらあかん!! まだシェーダ書いてないやろ!!
始まったばかりやんか… やっとスタートされたんやないか…!

さて、ちゅーことで、シェーダ書きましょうか。

シェーダ初歩

とりあえずこう書け

```
//頂点シェーダなんだぜ
//SVが何の略か知らないけど、とにかく座標を返すんだぜ
float4 BasicVS(float4 pos:POSITION) : SV_POSITION{
    ^ return pos;
}←
←
//ピクセルシェーダなんだぜ
//SV_TargetとまたSVが出てきてるけど無視。Targetは出力先の色なんだぜ。
float4 BasicPS( float4 Pos : SV_POSITION ) : SV_Target{
    ^ return float4(1.0f,0.5f,1.0f,1.0f); //RGBA
} [EOF]
```

かけたら basic_shader.fx とかなんとか適当に名前つけて、保存

はい読み込み

```
ID3DBlob* compiledShader=NULL;
ID3DBlob* error=NULL;
//シェーダ読み込み
result = D3DX11CompileFromFile("basic_shader.fx", //シェーダファイル名
    NULL, //10使わないし・・・NULLでいいや
    NULL, //10使わないし・・・NULLでいいや
    "BasicVS",
    "vs_5_0", //シェーダバージョン
    0,
    0,
    NULL,
    &compiledShader, //コンパイルシェーダオブジェクトを受け取るための変数
    &error, //エラーオブジェクトを受け取るための変数
    NULL);
```

はいはい。頂点シェーダしか読み込んでないワロスワロス。ピクセルシェーダも同様に読み込みましょうね。で、わけわからん型がある。ID3DBlobってのが、プロブってのはブログではないぞ。

どういう意味なんでしょう。ウイキペ大先生に聞いてみよう。

<http://ja.wikipedia.org/wiki/%E3%83%96%E3%83%AD%E3%83%96>

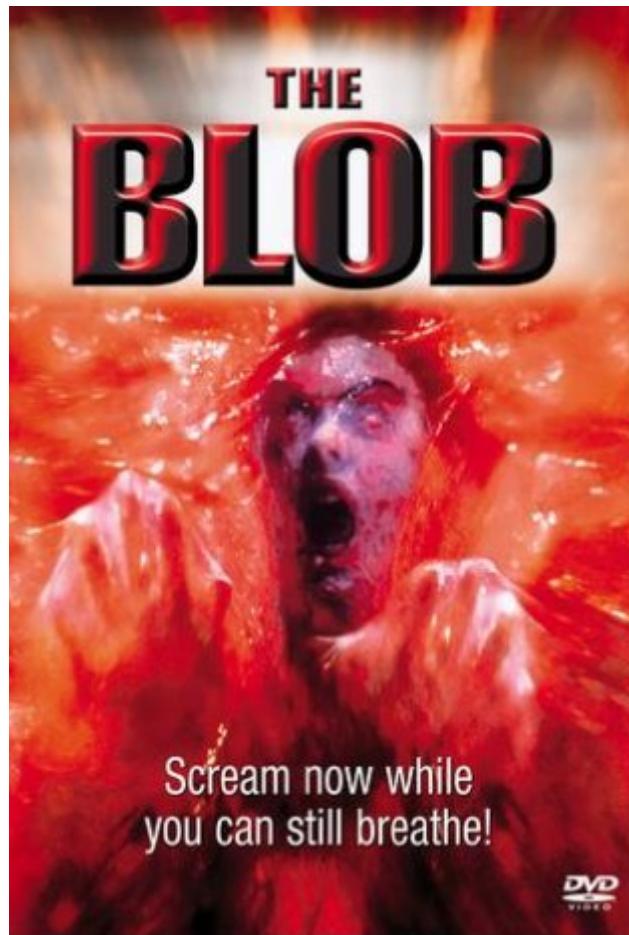
なになに「海岸に漂着する謎の肉塊」だと…? オレ…ニクカイ、ダイスキ!

ではなくて、

<http://ja.wikipedia.org/wiki/%E3%83%90%E3%82%A4%E3%83%8A%E3%83%AA%E3%83%BB%E3%83%96%E3%83%AD%E3%83%96>

これのことなんだろう。

つまり CompileFromFile によって、ベンダーしかわかんねー謎の肉塊になっちまうわけよ。まあいい。とにかくハードウェアがわかる「プロブ」ができた、と。



さて、デバイスにこの肉塊を食わせて頂点シェーダを作るわけよ。デバイスは悪趣味やな。
もちろんCreateVertexShaderって関数を使う

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

へえへえ、

で、当然第一引数はさっき取得した肉塊のポインタ。第二引数はさっき取得した肉塊のサイズ。次はNULLでいいわな。

ちなみに肉塊のポインタはGetBufferPointer()で取得できるし、サイズはGetBufferSizeで取得できる。簡単だろ？

最後はバーテックスシェーダを受け取るためのポインタポインタを放り込む。で、頂点シェーダの型は

ID3D11VertexShader となっておりますので、そのポインタポインタを放り込みましょう。

さて、これで頂点シェーダを得られたので、同様にピクセルシェーダも作って下さい。
できましたか？あ、もちろんピクセルシェーダはID3D11PixelShaderね。関数はCreatePixelShaderね。

さて、コンパイル済み頂点シェーダ肉塊に関してですが、実はもう少し使わせていただく。

MSのマニュアル見ると、「バーテックスバッファー生成する前に、レイアウトを決定しなければならない」とある。

レイアウトってなんかつちゅーと、Direct3D9の時にあったFVFみたいなもんだと思ってくれたまえ、実際その頃よりも柔軟に扱えるようになっている。

柔軟に扱えるようになったってことは、予想されるデータがさらに色々と増えてしまっているってことだ。ということは「今回のデータはこういう感じですよ」とことを細かく定義しなければいけなくなってしまったのだ。

この「レイアウト」を決定するための関数がCreateInputLayoutと言うやつなのだが、

[http://msdn.microsoft.com/ja-jp/library/ee419795\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419795(v=vs.85).aspx)

こういう定義になっている。割と困るのが第一引数。

D3D11_INPUT_ELEMENT_DESC

という、また細かい事を書かなければいけないヨカーン。しかも…配列だと？

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

しかもセマンティクスだのなんだのよく分からん。こんなことまで解説してたら終わらないので、



どうせ頂点しか書いてないし、もう、こう書いちゃえ。

```
D3D11_INPUT_ELEMENT_DESC layout={ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 };
```

さて、このポインタを第一引数に渡せばいい。

こうすれば、CreateInputLayout の第二引数は、レイアウト数1なので、1にしつければいいし、第三、第四引数には、頂点シェーダープロブのポインタ、サイズを指定してやろう。そうすれば、最後の引数でレイアウト情報をくれるので、そいつを IASetInputLayout に放り込んでやる。

```
context->IASetInputLayout(ilayout);
```

たまに、ここ Setって書いてるのに、Getって書く奴が出てきますが、Setですからね。間違えないでね。ここまでやれば、DirectX11側にレイアウト情報も伝えました。

ここまでがエラー無くできたら、もはやさっきの肉塊はポイしていいので、捨てちゃいましょう。

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

こんな感じのコードを頂点、肉塊、ピクセル、肉塊についてやっておいてください。

はい、上手にできましたら、デバイスコンテキスト側にシェーダを教えてあげましょう。
context->VSSetShader(vs,NULL,0);
ピクセルシェーダの方も同様にね。

で、やっと、やっと最後に既に登録してある三角形の表示を行いませう。
context->Draw(3,0);

あ、こいつらはな、ClearとPresentの間に入れるんやで。わかったな？変なところに置くなよ？

最終的にはこ一なるわけや

```
while(true){
    MSG msg;

    //メッセージを受け取り、受け取ったメッセージは削除
    if(::PeekMessage(&msg,NULL,0,0,PM_REMOVE)){
        ::TranslateMessage(&msg);//
        ::DispatchMessage(&msg);//メッセージをディスパッチ
    }
    if(msg.message==WM_QUIT){
        break;
    }
    context->ClearRenderTargetView(rtv,col);

    context->VSSetShader(vs,NULL,0);
    context->PSSetShader(ps,NULL,0);
    context->Draw(3,0);
    swapchain->Present(0,0);
}
```

はい、ゆーとーりに書いたら、画面上でつけえ直角三角形が出たんじやないかと。出てない人はどつか間違ってるか、このテキストがどつか間違ってる可能性があります。とりあえず、隣の友達と見比べながらやってみてください。

ここまでできたら、もうちょっと3Dっぽくしていきます。

スリーディーっぽく…していこう

さて、今の状態だとまったく3Dっぽくないと思う。事実3Dではなあい！

3D空間なんかなかったんや。

なんでかというと…Direct3D9をやってる人は疑問に思ったかもしれない。
疑問に思わなかつたか？どうしてカメラもパースも設定しないのかと…。

そうなんや。三次元座標で設定はしたんだが、今の状態ってな？がまったく効いてない状態なわけよ。いや、有効ではあるんやけど、カメラもパースもないために、3Dらしさがないわ

け。

というわけで3Dらしくするためにカメラ行列、プロジェクション行列について勉強しよう。

行列について

残念ながら、この辺から、数学的な話がでんりでん出てくる。3Dにおいて基本となるのがこの「**行列**」なのだ。英語でいふと **Matrix**。この後やたらと出てくるので、この英語も覚えておこう。とにかく何かにつけてこの「**行列**」である。

行列、行列言つてもイメージわかないだろうから、とにかくこいつを見てくれ。こいつをどう思つ?

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

凄く…ワケワカです。

大雑把に言うとベクトルと似たようなもんなんだ。あくまでも「大雑把」ではあるけど。

まあいきなり四元は早かつたかな。これならどうだろうか。

$$A = \begin{pmatrix} 2 & -1 \\ 2 & 3 \end{pmatrix}$$

どうだぐっと親しみやすくなつただろう?さて、この行列に対する演算には、足したり引いたり乗算したりっていういくつかの演算があるんだが、基本的には乗算ばかり使うし、加算減算は簡単すぎるのでここでは書かない。

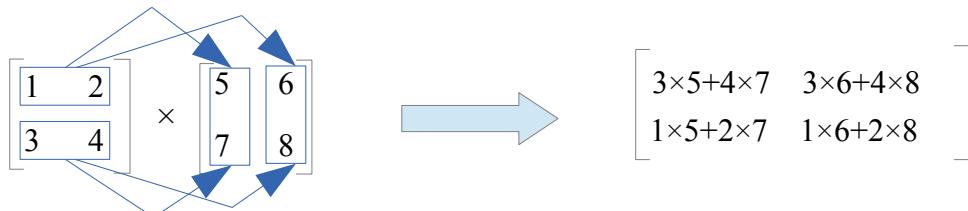
とにかく乗算について説明してみよう。数学における乗算記号は×だし、たいていのCG系ライブラリでは行列同士の乗算は*で表される。

専門的なことはともかくこんな変な「**行列**」とやらを乗算したらどうなるのか見てもらおう。乗算のルールは

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

はい、乗算結果の左上は、乗算左辺の左上×乗算右辺の左上+乗算左辺の右上×乗算右辺の左下となっている。

なんか内積の計算みたいな感じと思つたらいい。



この計算方法は3元になろうが、4元になろうが変わらない。ちなみに3次元空間内の変換

を表すには4元の行列が必要となる。このためDirect3D上でMatrixといえば、 4×4 行列を表すのである。

こんなものが何の役に立つのかと思っているのもかもしれない。実はこれを用いれば、ベクトル…つまり座標値に対してありとあらゆる変換を行えるのだ。どういうことなのか。

説明のためにまた 2×2 行列に戻って話をするが、変換のための行列にはこういうものがある。

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

なにか思い出さないだろうか…？コブラのマシンはサイコガン

そう「回転」なんだわ。だからこういうやつを特別に「回転行列」と言ったりする。ミクの手を曲げるのも、つまるところ、この回転なんだね。

例えばこいつとベクトルを乗算するとどうなるのか。

ちなみにベクトルと行列の乗算は

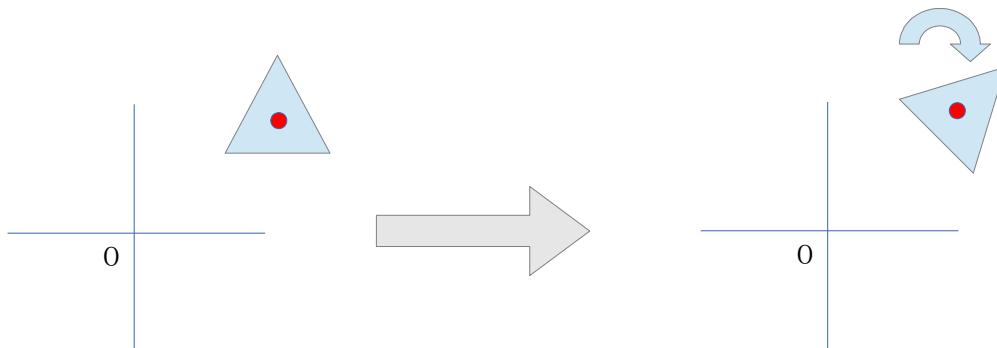
$$\vec{v}' = M \times \vec{v} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} m_{11}v_x + m_{21}v_y \\ m_{12}v_x + m_{22}v_y \end{pmatrix} = \vec{v}',$$

と言った感じだ。ベクトル自身を行列の仲間と捉えてればいい。事実そうなんだし。さて、では回転行列をかけてみよう。

$$M \times \vec{v} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} v_x \cos \theta - v_y \sin \theta \\ v_x \sin \theta + v_y \cos \theta \end{pmatrix} = \vec{v}',$$

と、このように元のベクトルを回転させたベクトルを算出できるわけだ。ベクトルを算出できるということは言い換えると現在座標を原点を中心回転することができるということなのだ。

で、実用の場合にはこの「原点を中心」ってのがネックになり、回転だけじゃ事足りない。なので、これに平行移動を付け加えて…要は要素数を一個ふやして、回転&平行移動を扱えるようにしている。実際はこれに拡大縮小も合わせてアフィン変換とか言ったりするんだが、とりあえず使うのは回転と平行移動である。

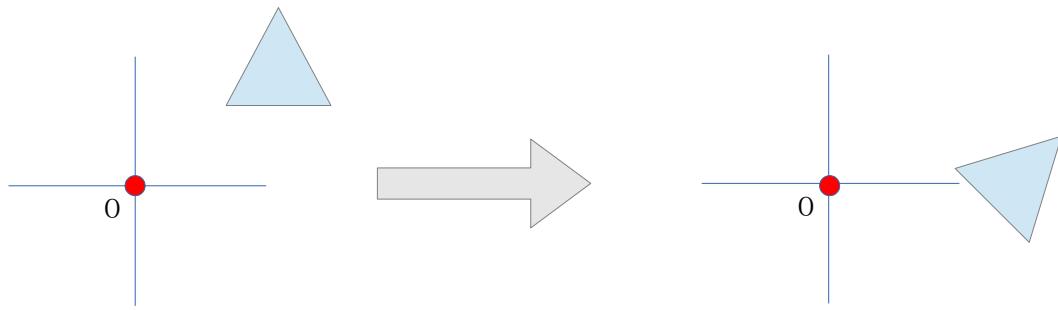


で、さっきの原点を中心ってのが厄介ちゅう話をしよう。

例えば、どこぞのポリゴンを回転させるには回転行列をかけることによって回転させるわけなんだが、

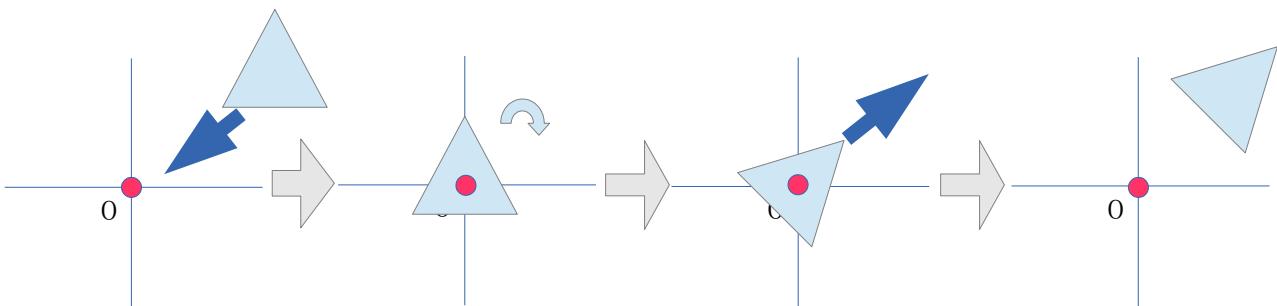
通常、回転ってのは上のように「モデルの中心」で回転することを期待するのだろう。だが、

そうはいけないのである。「原点を中心にしてことは、何も考えずに回転させると



こうなるわけだ。これではよくないね。そりや、どつか中心に旋回するって動きならいいんだろうけど、腕を回転させるには向かないわけだ。

じゃあどうするのかというと、一度原点に移して回転して、またもとの位置にそっと戻すって手順をとるわけ。回転1回ですむところを3回も変換かかって面倒なんだが、これは仕方がない。



見て分かるように回転の中心は常に原点なのだ。はい、こんなことを全ての頂点にやるのか、遅くねーのかと思うかもしれないが、心配ご無用。

行列というのは、ドゥンドゥン合成できるのだ。
これから毎日合成しようぜ。

つまり、上のようなモデル中心回転行列を予め計算しておく。例えば、原点に移動する行列をT、回転行列をR、もとの位置に戻す行列をT'すると

モデル回転行列 = T × R × T'

となる。つまりこいつはアプリ側でちゃつちゃと計算しちゃって、そいつをシェーダ側に流し込めばそれで終了なのだ。

わかったかね？

はい、ものごつい長く遠回りしたんだが、3DCGってやつは、仮想のカメラで仮想の世界を写すことによって、ディスプレイに仮想世界を見せるっていう流れで、僕らは3Dゲームを遊ぶわけだ。

だが、実際にカメラとやらがあるわけではなく、数々の数学的な処理を経て「あたかもカメラで撮影したかのような」しきみを提供する。

CG検定の教科書的に言うと、

モデリング変換(ワールド変換)→視野変換→投影変換→ビューポート変換

これらの変換を行うことにより、仮想世界の3D物体が僕らの目に見えるようにする。この流れをビューラインパイプラインというが、この名前はCG検定を受けない限りは覚えてないでいい。

ただ、流れはこの流れなので、覚えておいてくれ。

まず、ワールド変換なんだが、これは物体が、仮想世界の標準からどれくらいズレているのかつまり、どれくらい中心から離れているか、どれくらい回転しているかって変換を行うわけ。これは一番わかりやすい変換。

プログラム上では、単なる回転行列や、平行移動行列を使用する。

次に視野変換なんだが、こいつがカメラを表している。これはどういうことかっていうと、モデルは世界のあちこちに配置されているわけなんだが、例えばゲームの時にはカメラが固定なわけではないし、視点は動くものだと考える。そうすると、視点がただ原点からZ軸方向を見ているというところから、カメラが向いている方向をZ軸とするような変換を行わなければならない。これが視野変換です。

プログラム上ではビューベクトルを使用する。

次に投影変換。これは僕が「パース」、パースって呼んでいるもので、みんな漫画のパースって聞いたことがあるかな？

あれって、要は遠近法のことなんだよね？遠くにあるものが小さく見えて、近くにあるものが比較的大きく見えてしまうアレだよ。で、これはカメラの特性にも依存していて、画角というものの、画角が大きいとパースがきつくかかたり（遠くのものがより小さく）、画角が小さいとパースが緩くなる。

これはプログラム上ではプロジェクション行列にて表現する。

最後のビューポートはすでにやってるんで気にしなくていい。

今回は、これらの行列を駆使して、ちょっとだけ3D的にしてやろうってのが目的なのだ。

で、行列がやたらと必要なんだが、怖がらなくていい。

今回のDirectX11で行列を表すには
XMMatrixを使用する。

中身はだいたいこんな感じ

```
FLOAT _11, _12, _13, _14;  
FLOAT _21, _22, _23, _24;  
FLOAT _31, _32, _33, _34;  
FLOAT _41, _42, _43, _44;
```

はい、もうこれだけで、 4×4 行列ってことがわかるだろう。はいはい。まあだけどね。僕らが計算するわけではないんだから、コマケ工事気にする必要はない。だが、これだけは知っておかねばってのがある。

行列ってのは、かける方向によって答えが変わっちゃう

どういうことが、下を見ていただこう

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

左と右を入れ替えてみる

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

うぎゃー!!ぜんぜん違う!!!!

はい、 $A \times B \neq B \times A$ って事がわかりますね。これだけは頭に入れておいて下さい。

で、さっき自分で計算する必要が無いとか言った。それは、例えば単位行列(全てを0にして左上から左下までの斜め線が1になる行列)

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

こいつを作りたければ一個ずつ入力しろ!

嘘だつ。

あるのだ。

XMMatrixIdentity();

という関数がな。これは単位行列を返す。単位行列はなにもしない行列です。

次に回転だが、こいつも例えればY軸中心に回転したけりや

XMMatrixRotationY(ラヂアン値);

ってのがある。ラジアン値ってのがポイント。ラジアン値がわからんね一人はここで脱落願います。

次に平行移動

XMMatrixTranslation(X 移動値, Y 移動値, Z 移動値);

さて、こいつらはいい。まだいい。んじゃ、さっきビュー行列だのプロジェクション行列だの言ったが、そんなんあるのか。

あるのだ。

カメラの行列は

XMMatrixLookAtLH(視点, 座標, 注視点, 座標, 上ベクトル);

はい、上ベクトルってのは、謎だと思ってるかもしれないが、後々解説する。今は(0,1,0)にでもしといてくれ。

あ、視点、座標も注視点、座標も上ベクトルも XMFLOAT3 ではなく、XMVECTOR であるため、ここはちょっとだけ工夫が必要だね。

XMFLOAT3 を XMVECTOR に変換するには XMLoadFloat3 という関数を使用する。

つまり視点が(0,0,-100)ならば

XMLoadFloat3(XMFLOAT3(0,0,-100)) が視点の XMVECTOR となる。

それから、XMMatrixLookAtLH はあくまでも行列を作って返すだけの関数なので、こいつを呼び出すだけではなんにもならない。とりあえず XMMATRIX 型の変数にでも入れておこう。

XMMATRIX viewmat=XMMatrixLookAtLH(XMLoadFloat3(視点), XMLoadFloat3(注視点), XMLoadFloat3(上));

となる。

視点はわかるな。注視点は、自分が何に注目しているか。たとえば、向こうの戦車を見ているのなら、戦車の座標が注視点座標となる。

最後にプロジェクション行列なんだが、

XMMatrixPerspectiveFovLH(画角, アスペクト比, 近い方, 遠い方);

こういうやつだ。こいつでクリッピングボリュームも設定する。ともかく画角はさっき言ったパースのきつさだ。大体 45° になるようにラジアン値で書いてくれ。ラジアン値分からんような奴は知らんが、 45° はラジアンで表すと $\pi/4$ であるから、 $3.141592\cdots$ を 4 で割ってやればいいことになる。

アスペクト比は高さ ÷ 幅なんだわ。だが、この逆数になるようにしておいてくれ、理由は後で話す。つまり幅 / 高さにしてくれ。この時に、整数型 / 整数型の状態にしていると、800 / 600 なら、 $1.333333 \rightarrow 1$ になってしまい、600 / 800 とかにしてしまうと、 $0.75 \rightarrow 0$ になるため注意す

る必要がある。だから、 $800.0f / 600.0f$ とする。

```
XMMATRIX proj=XMMatrixPerspectiveFovLH(PI/4, 800.f/600.f, 0.1f, 100.f);
```

近い方は大体 $0.1f$ 遠い方は $100 \sim 500.f$ くらいでええんちゃうんか。
この範囲を超えると見えなくなると思っておいてくれ。

で、これでもらってきた行列をかけてやる。例えばワールドを world、ビュー行列を view、プロジェクション行列を projection にした場合、
そのままかけねばいい。

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

さて、この matrix をシェーダ側に渡したらんとアカンわけや。

とりあえず、こういう決まりきったようなパラメータはコンスタントバッファ(定数レジスター)に送るのがお決まりや。

今回は頂点に対する加工をしたいので、頂点シェーダ側にコンスタントバッファを送信する。ことにする。

で、こいつもそれなりに面倒なのだ。どうしてこんなに DX11 は面倒なのか…ホイホイ渡してくれよ…バッファというからには、例によって例のごとく CreateBuffer を使わにゃならんわけです。まあええよ。

```
D3D11_BUFFER_DESC matBuffDesc={};  
matBuffDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER; //これが今回のポイントやね  
matBuffDesc.ByteWidth = sizeof(XMMATRIX);  
matBuffDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE; //バッファの中身は CPU で書き換えます。  
matBuffDesc.Usage = D3D11_USAGE_DYNAMIC; //CPUによる書き込み、GPUによる読み込みが行われるって意味。
```

はいはいいつもどおり CreateBuffer

できたバッファの名前は matrixBuffer とでもしておこう。

で、このバッファに対して値を書き換えるのがちと厄介だ。こいつの値を書き換えるためにロックかけて書き換えてアンロックするように、DX11 では、マップとアンマップの間で書き換える。

パラメータに関してはここをとりあえず見ておこう

[http://msdn.microsoft.com/ja-jp/library/ee416245\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416245(v=vs.85).aspx)

つまり、

```
D3D11_MAPPED_SUBRESOURCE mem; //マップして取ってくるメモリの塊
```

```
context->Map(matrixBuffer,0,D3D11_MAP_WRITE_DISCARD ,0,&mem);
//ここでこのメモリの塊に、マトリックスの値をコピーしてやる。
memcpy(mem.pData,(void*)(&matrix),sizeof(matrix));
context->Unmap(matrixBuffer,0);
```

はいこれで matrixBuffer の中身が書き換わったでござる。

あとはセットしてやりや終わり。

セットするには

```
context->VSSetConstantBuffers(0,1,&matrixBuffer );
```

今回は変数1つだけだし、一番最初の変数だしで、スロットは0。バッファ数は1にしつければいい。

で、これだけじゃダメなんだわ。

これを受け取るシェーダ側の準備をしなければならない。

シェーダ側のコンスタント/バッファーは

cbuffer

で表現します。

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

今回はレジスタゼロなので

シェーダ側に

```
cbuffer global{
    matrix mat;
};
```

って書いとけばいい。

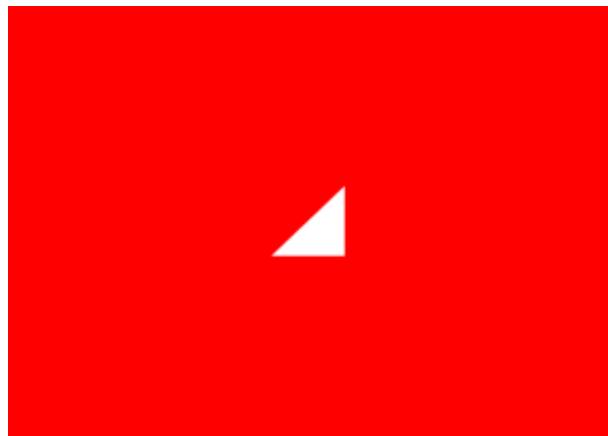
で、あとは入力posに対して、mul(乗算)を書いてやればいい。mulはシェーダにおいて、行列やベクトルを乗算するのに使います。なお、内積、外積は専用の関数があるので、そっちを使いましょう。

[http://msdn.microsoft.com/ja-jp/library/bb509628\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb509628(v=vs.85).aspx)

ということで、変換後の座標が欲しければ

```
return mul(pos,mat);
```

とでもすればいい。そうすれば、こんな感じで出ると思いますが、如何でしょうか？



カメラの設定ができてればこんな感じで表示されるはずであります。

さて、ここからこの三角形をくるくる回転させてみましょう。既にワールド変換行列の準備はできておりますので、さっそく定義したワールド行列を少しずつ回転させてみます。

おっと、その前に、説明をちょっと忘れていた。

とりあえず今の実装でやるのならば、シェーダに渡す前に、渡す行列を転置してくれたまえ。転置行列を作るには、`XMMatrixTranspose` 関数を使用する。

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

ところで、転置行列ってどういう行列なのか？

ハリハツワロス。

そんなに怯えることもながろう。

縦と横がひっくり返ってるってだけの行列だよ。

これは、行列のタテヨコ（行と列）をひっくり返すという意味になります。プログラムの二次元配列の $\alpha(y)(x)$ というのの扱いを $\alpha(x)(y)$ にしてしまうというのと同じです。

例えば、 (x, y) を転置すれば $\begin{pmatrix} x \\ y \end{pmatrix}$ になるし、その逆もまたしかりです。回転行列であれば
 $\begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$ を転置すると $\begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix}$ になりますよね？

そんだけ。`XMMatrix` の出す結果と、シェーダ側の `mul` 演算に食い違いがあるため、こういうことになってる。

シェーダ側の

`mul(pos,matrix);`

ってコード。この順番でかけるとすると、ならば転置しておかなければ正しい演算結果にはならないのだ。

もし Transpose 使いたくないなら
mul(matrix, pos);
にすべきでしょう。

ただ、これだと、pos に対して matrix 行列で変換かけるってイメージじゃないから、この順番にしてるだけ。

あと、噂によると、GPU 側で転置するのって、なんかコストかかるから CPU でやつとけって話があるんだけど、真偽のほどが定かじゃないので、とりあえず無視しとく。

できたかな？ 回転したかな？ で、向こう側（反対側の面）が見えなくなってると思うけど、これはデフォルトで「背面カリング」ってのがかかってて、見えなくなってるんだ。

これいじるには、ラスタライザに対して働きかけなきゃいけないので非常に面倒なので、今回は割愛しておく。どうせメッシュモデル読むようになつたら不要だしね。

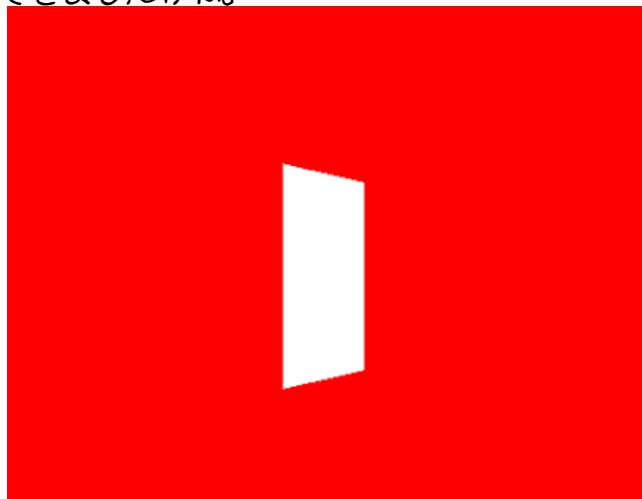
四角にしてみよう

さて、いつまでも三角じゃおもんないな。じゃあ三角を四角にしてみよう。
できるかな？

現在3点定義している部分を、4点に増やして ByteWidth を 4*sizeof(XMFLOAT3) もしくは sizeof(配列名) にする。

あとは TRIANGLE_LIST を TRIANGLE_STRIP にするだけ…じゃなくて、Draw の頂点部分も書き換えて、Draw(4, 0) にすればオッケー

はい、どうでしょうか？ できましたかね。



できましたか。ではさっさとテクスチャマッピングへと洒落こもうか!!

テクスチャマッピング

そもそもテクスチャマッピングってのを知らない人のためにお話ししておくと、今回の四角のようなペラペラの板に絵を貼り付けることをテクスチャマッピングという。

まあ、実際はペラペラではなく、人間の顔を作ったり、壁の模様つける時等に使用する技術なのだ。最初は簡単にするためにこのペラペラ四角に貼り付けます。

貼り付けるためには、今の三次元座標情報だけではダメで、UVというこのペラペラに貼り付けるために、画像を配置する二次元座標情報が必要になります。

というように、テクスチャを貼るためには少々また新しい用語、知識を知っておく必要があります。

でも、君らもう二十歳超えてたりするんやから、自分に必要な知識くらいは自分で発掘して身につけや。社会に出たら「習ってないから知りません。教えてもらってません」とか通用しまへんで、ホンマの話。習うまで手を動かさずにアツバーンと口開けて待ってるんて恥ずかしいで。



少なくとも大人の…二十歳にもなった人間の態度じゃないわな…バカッターでバカなことしてるとおんなじや。

まあ、CGに関しては、色んな本が出とるんやから、CGWORLDとか読むんでもいいし、ゲームプログラマになりたければ、最低限「CG検定エキスパート」の知識くらい身につけとくんやで。最低限やで。当たり前やん？

はい、気を取り直して、知つて当たり前のことやけどいいで～。

絵を貼り付けるための基礎知識

用語

テクスチャ

いわゆる画像のこと。bmpだのpngだのddsだの…があるよね？こいつらから情報を

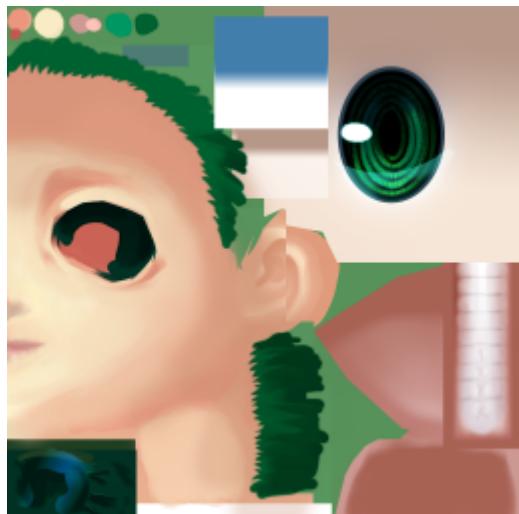
よつ見込んで画像を表示することになるのだが、その画像のことを「テクスチャ」という。そのテクスチャをモデルの表面に貼り付けることをテクスチャマッピングといいします。

マッピングって言葉が、元々物体表面に何か貼り付けるって意味になってると思ってくれ。なので、ここではただ単に絵を貼り付けるだけですが、そのうち凸凹や光沢を貼り付けたりすることになると思います。

UV 座標

UV 座標ってのはテクスチャを貼り付ける時の基準となるテクスチャ上の2次元座標のこと。うん、まあよく分からんと思うので、詳しく説明する。

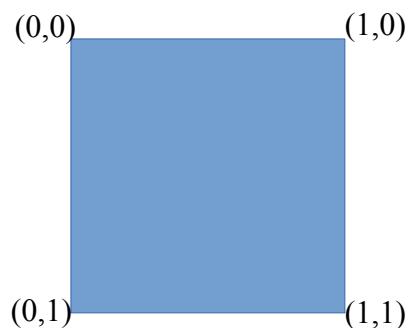
例えば MMD 等において前述のテクスチャはどのような画像となっているのかというと



ああ…かわいいぜ…。

はい怖いですね。

まあ、これをそのまま貼り付けたんでは、怖い結果にしかならないわけ。なので、対象となる画像の左上を(0,0)、右下を(1,1)とする座標系を作るわけ。



このようになっている。3D 座標系と違い、下方向がプラスになっていたりするのが、ちょっとヤヤコシイかもしれない。

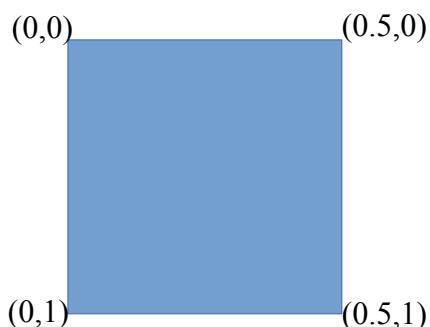
で、これがどのように役に立つのかというと、貼り付ける際の貼付け方が柔軟に設定できる。どういうことがというと、例えば、物体にびっしり貼り付けなければいけないというルールだった場合非常にテクスチャが沢山できてしまい、また絵作りも大変になってしまふのである。

このため、画像の座標系というものを定義し、その数値でどのように貼り付けるのかを設定します。

例えば、板ポリに絵を貼り付けた場合、通常ならばテクスチャ的には板の端が(0,0),(1,0),(0,1),(1,1)になっているわけだ。絵の端から端までの大きさを1としてるわけや。



通常通り貼り付ければ、板ポリがこのようになる。だが、



このように指定されている場合、貼り付けた結果は



ご覧のとおり、右端が0.5に指定されているため、絵の0.5つまり半分の地点までが採用される。結果として半分までの画像しかポリゴンには張り付かない。

サンプラー

サンプラーっていうのは、画像をどんな感じに貼り付けるのかって指定するもの。ついでに言うと、どういう風にピクセル値を取ってくるのかにも関わっている

通常は紙一枚貼り付けるイメージでいいんだけど、そこはコンピュータ。設定によっては繰り返し指定にすれば、小さいテクスチャでもタイル状に並べて模様にすることもできる。その他、拡大縮小時の補間法など、割と細かい指定ができる。それがサンプラー。

専門的なことはともかくテクスチャと対になっているオブジェクトなんDA!

DX11におけるテクスチャマッピングはこの3つ(テクスチャ、UV座標、サンプラー)がわかつていればとりあえずはなんとかなる。

テクスチャマッピング手順概要

手順としては

- ①テクスチャをファイルから読み込む
- ②シェーダ側にテクスチャ用変数、サンプラーステート用変数を用意しておく
- ③頂点レイアウトの変更(UV情報の追加)
- ④頂点情報の追加(それぞれの頂点にUV値を設定)
- ⑤サンプラーの作成
- ⑥シェーダにテクスチャとサンプラーをセット
- ⑦頂点シェーダの出力にUVを追加
- ⑧ピクセルシェーダの出力色をテクスチャからサンプリングした点に変更

(たった?)これだけの手順でテクスチャをペラ紙に貼り付けることができるわけだよ!!!
やったー!!!



というわけで、さっさと作っていこう。

注意点といふか

注意点といふか、ちょっと混乱しないように先に行っておくと、Direct3D9まで「テクスチャ」と言っていたアレはDirect3D11では「**シェーダリソース**」って呼ばれるようになります。

恐らく、いわゆる画像ファイルからロードされる「画像」って奴の使い道が「絵」としての用途ばかりでなくなつたからだろう。

後々出てくると思いますが、ベクトル情報を「画像ファイル」として保存して、それを実行時に使用するって技法があつたりして、使い道が異様に多岐に渡るようになつたため。もはや用途が「絵」としてではなくつてしまつたと言う事が言えます。

テクスチャ読み込み

これは絵を用意してそれをロードするだけ。ロードする関数は D3DX11CreateShaderResourceViewFromFile って関数だ。各自リファレンスを読んでいただきたい。

[http://msdn.microsoft.com/ja-jp/library/ee416885\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416885(v=vs.85).aspx)

そして悲しいことにまだいるのだが、このリファレンスのテキストをコピペして「動きません」とか言ってる奴がちらほらいる。

最初に言ったでしょ？ これはあくまでも「定義」であつて、コピペしても動きませんよって。人の話も聞いてない、テキストもきちんと読んでないくせに「できない」じゃねーよ。まず人の話を聞こうぜ。

あと、分からんでもいいからざつとマニュアルも読んでおこう。このテキストはリファレンスをオレの独断と偏見で読んだ結果を書いているテキストなので、間違ってるかもしれない。きちんと読んでから先に進もう。

ほい、苦言はここまでにしてとりあえずロードしてみようか。リファレンス見てわかるように、こいつは誰の持ち物でもないので、そのまま呼び出せばいい。で、定義は

```
HRESULT D3DX11CreateShaderResourceViewFromFile(  
    ID3D11Device *pDevice, // デバイス(CreateDeviceAndSwapchainで作ったアレな)  
    LPCTSTR pSrcFile, // テクスチャファイル名  
    D3DX11_IMAGE_LOAD_INFO *pLoadInfo, // NULLでいい  
    ID3DX11ThreadPump *pPump, // NULLで  
    ID3D11ShaderResourceView **ppShaderResourceView, // ポインタのポインタ
```

```
HRESULT *pHResult //非同期実行時には必要だが今はNULLでいい  
);
```

何度も言うようだけど、これは定義だ。これをコピペすんな、書き写すな。しつこいようだけど、何回言ってもやるやつがいるのでしつこく言う。特にゲーム開発者になろうと爪の先ほども考えている奴はこんな間違いしちゃダメだ。…命が危ないぞ。

テクスチャファイルはサーバー(gakuseigamero.yukawa.no)2~3年向け課題サンプル素材に色々と置いてあるので使ってもいいのよ。

```
ID3D11ShaderResourceView* texture;  
result = D3DX11CreateShaderResourceViewFromfile(device, "soniko.jpg", NULL, NULL, &texture);
```

ねつ、簡単でしょ？

シェーダ側の準備

次に、最終的にシェーダ側にテクスチャ情報、サンプラー情報を送るための準備をしておこう。

ありがたい事にテクスチャ情報を受け取る仕組み、サンプラー情報を受け取るための仕組み自体は既にできている。これも自作でやろうとしたら死ねるって

で、テクスチャなんだが、シェーダ上でテクスチャオブジェクトを表すには Texture2D 型で表す。

[http://msdn.microsoft.com/ja-jp/library/bb509700\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb509700(v=vs.85).aspx)

見て分かるように、色々とテクスチャの種類があるが、とりあえず一番直感的に使えそうな Texture2D を使用する。

とりあえずアプリケーション側から、テクスチャ情報を渡してそれを受け取るためだけの変数をシェーダ上に書く。

```
Texture2D tex;
```

これでいい。簡単でしょ？

次にサンプラーについてなんだけど、別にサンプラーが必須ってわけじゃない。だけどサンプラー使わないとシェーダのコードが結構面倒になる。どう面倒になるのかというと、いちいちピクセルシェーダで画像サイズやミップマップレベル(後述)までシェーダに取得関数を書かなければいけなくなるからだ。

そんなんするくらいなら、素直にサンプラー作った方が良いのである。

で、こいつも既に用意されているので、後は宣言するだけとなります。ちなみにここで

出てくるSamplerStateはDX10と同じものなので、ヘルプはそっちを見ておこうね。

[http://msdn.microsoft.com/ja-jp/library/bb509644\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb509644(v=vs.85).aspx)

とりあえずシェーダ側に

```
SamplerState samplerstate;
```

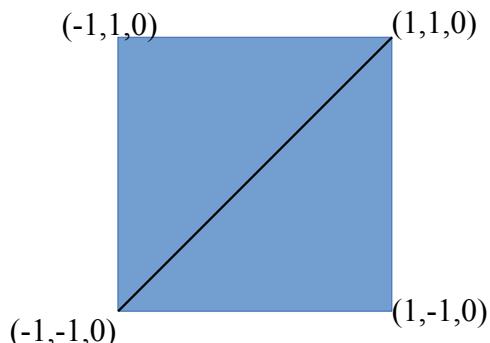
とでも書いておこう。

頂点レイアウトの変更

はい、んで、次に頂点レイアウトの変更を行います。なんでそんなのが必要なのか？CGの初步的な知識がある人なら知ってると思うけど、テクスチャの貼られているモデルには通常ではUV座標情報が付加されている。

つまり、頂点ごとにUV座標を指定することによって、面のテクスチャの貼り具合を調整しているのだ。

例えば、既にトライアングルストリップで四角形を作っているのだけれど現状ではこのように



4つの頂点があり、それぞれに(x,y,z)のfloat*3が割り当てられている。これはいいね？

しかしテクスチャを貼るにはこの情報だけでは足りない。UV情報も必要なのだ。UVは前にも言ったように2次元座標系なので、あと2つ分floatが必要となる。つまり



このような表示にしたいのならば、それぞれの頂点に(x,y,z,u,v)の5情報が必要となる。

かなり煩雑になってきますね。

(-1,1,0,0,0)

(1,1,0,1,0)

(-1,-1,0,0,1)

(1,-1,0,1,1)



それだけにレイアウトをきっちり規定しないとコンピュータ側はわからなくなってしまう。

まあまあ、既にレイアウトの仕組みは作ってんだから追加するだけ、簡単なもんよ。ちなみにUV座標を表すセマンティクス文字列はTEXCOORDである。ご想像通りtexture coordinateの略である。coordinateは座標って意味。テクスチャ座標ね。

で、(u,v)という2次元座標なので、float2つぶん。

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

この中からfloat2つのやつを探す。floatは32ビット…！

uとvだからポイントは2成分、さらにFLOATというだけで特定できる。

DXGI_FORMAT_R32G32_FLOATしか合致しないことが分かる。…だろ？こうやって選ぶんだ。

他はPOSITIONの時と同じ…って言いたい所なんだがちょっと違う。

5番目の要素。こいつは0ではない。先頭から数えて何バイト目に当たるのが、それをおしゃてやらにやならん。POSITIONがR32G32B32ってことは、何バイトかな？

ちょっとしたクイズ問題だと思って、じぶんで考えて書いてみよう。

あー、あと2~3年にもなってこんな阿呆なことはやらないかとは思うんだけど、元のレイアウトは消すんじゃねーぞ、追加だぞ、追加。

で、下手すると、追加の仕方がわからんとかゆーやつが出てくるかも知れへんけど、そんなことは知らん。どうにかしなさいGoogle先生に聞きなさい。どうにかできない子はプログラマとか



注意して欲しいのは、前回のレイアウト宣言を「配列」で宣言していない場合、コンパ

イラから怒られるから配列で宣言しこう。

```
D3D11_INPUT_ELEMENT_DESC layout[] = {  
    {"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 },
```

たとえばこんな感じですかね。

頂点情報にUV情報を追加

レイアウトの準備ができたので、今の頂点情報(座標のみ)にUV情報を追加しよう。当たり前なんだけど、これもただ1.0,1.0,0.0,0.0,0.0って書くだけじゃダメなんだぜ?

さあ、これくらい自分で考えよう。

さらに言うと、ここでUV座標を追加した上で実行しようとすると、四角形が表示されなかつたりする。

なぜか?自分で考えて欲しいのだけど、あえて書くと、各所で設定したサイズと食い違ってきてているからだ。これも自分で考えて修正してくれ。

はい、まずはこの状態で四角形が表示できるようにしてみよう。こっからやね。ちょっとヒント少ないかもしれないけど、このくらい自分で考えられるようにならんとちょっとDX11は話にならんのでね…

はい、授業で追い付いていない人に課題です。一週間後9月24日までに、3D空間上の四角形表示ができるようになっておいて下さい。

できなかった人は、たぶん、単位落とします。

頑張ってください。

サンプラーの作成

ええ、まあ、サンプラーは最終的には

[http://msdn.microsoft.com/ja-jp/library/ee419717\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419717(v=vs.85).aspx)

PSSetSamplersでセットするんだけど、そのために必要なID3D11SamplerState型の変数を作成する必要がある。

[http://msdn.microsoft.com/ja-jp/library/ee419870\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419870(v=vs.85).aspx)

作成するにはCreateSamplerStateを使用する。

[http://msdn.microsoft.com/ja-jp/library/ee419801\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419801(v=vs.85).aspx)

どういうサンプラーを作っていくかの設定はD3D11_SAMPLER_DESCを使用する。

[http://msdn.microsoft.com/ja-jp/library/ee416271\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416271(v=vs.85).aspx)

…またパラメータ多いなあ。とりあえず指定が必須なのは最初の4つのパラメータなので、ちまちま設定してきます。

Filterに関してですが、これはD3D11_FILTERのどれかを選びます。

[http://msdn.microsoft.com/ja-jp/library/ee416129\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416129(v=vs.85).aspx)

…まあ、とりあえず色々と試して欲しいんだけど、一般的には線形補間をかけますので、

D3D11_FILTER_MIN_MAG_POINT_MIP_LINEAR

を使用します。なお、ポイントサンプリング(D3D11_FILTER_MIN_MAG_MIP_POINT)にすると、大体何がおこるか分かるんじゃないかな…？

最初はリニアにしといて、テクスチャ表示までできたらポイントサンプリングにしてみましょう。

で、次の3つ。AddressU,AddressV,AddressWですが、これも

[http://msdn.microsoft.com/ja-jp/library/ee416346\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416346(v=vs.85).aspx)

から選びます。基本的にはなんでもいいです。今回は0~1縛りにしてるんで意味ないです。好きなのにしておきましょう。

まあ、説明しないと「お前分かってへんねやろ?」って思われるのもムカつくんで一応解説だけやっとくと、こいつはuv値が0~1の範囲を超えた時に、テクスチャをどのように表現するかを決めるためのものです。

ちなみにWRAPは、おんなじものがずっと繰り返し出します。つまり0.5の部分の色と、1.5の部分の色は同じになるわけです。

次にMIRRORですが、こいつは繰り返すところはWRAPと一緒になんですが、繰り返すたびに画像が反転します。これ、何がつええかというと反転してるんで、画像の切れ目が見えなくなるんですね。

次にCLAMPですが、これ、端っここの色をガーッと引き伸ばします。…あんまし使わない

んじやないかな。

次にBORDERですが、これは予め指定した色で塗りつぶされます。特に指定しなければデフォルトの色(黒じゃねーかな)が使われます。

次にMIRROR_ONCEは、一回だけミラーして、あとはそのままです。終わり。

まあ、なんでもいいけど、基本はWRAPかMIRRORですね。

はい、んじゃ、そろそろ何をすべきか分かってるとと思うので、サンプラーつくってみようか。

ああ?わからん…だと?

```
ID3D11Sampler* pSampler;
D3D11_SAMPLER_DESC sampdesc = {};
sampdesc.Filter = D3D11_FILTER::D3D11_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR;
sampdesc.AddressU = D3D11_TEXTURE_ADDRESS_MODE::D3D11_TEXTURE_ADDRESS_WRAP;
sampdesc.AddressV = D3D11_TEXTURE_ADDRESS_MODE::D3D11_TEXTURE_ADDRESS_WRAP;
sampdesc.AddressW = D3D11_TEXTURE_ADDRESS_MODE::D3D11_TEXTURE_ADDRESS_WRAP;
device->CreateSamplerState(&sampdesc, &pSampler);
```

とかって書くんだよ。いい加減自分で書けるようになりますねオホホホ。

やった!やったよ!!テクスチャとサンプラーの用意ができました!!

さあ、シェーダにセットだ。

シェーダにテクスチャとサンプラーをセット

関数はPSSetSamplers

[http://msdn.microsoft.com/ja-jp/library/ee419717\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419717(v=vs.85).aspx)

とPSSetShaderResources

[http://msdn.microsoft.com/ja-jp/library/ee419731\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419731(v=vs.85).aspx)

を使います。

もー、ごめん。これくらいは自分でやってちょ。両方ともに、第一引数は0で第二引数は1でいいから!!いいから!!



ここまできたんだから後はわかるんだよね?がんばろう。

さあ、盛り上がってまいりました。

お次はシェーダ側のコードだ。

シェーダ側にUV情報を追加する

結構ここで変更が発生します。なぜかっちゅーと頂点シェーダおよびピクセルシェーダに細工を加える必要があります。

既に渡す情報が「頂点座標のみ」なんだけど、こいつにUVを付け足す。

シェーダ側でもstructは使えるので、

```
struct V{  
    float4 pos:SV_POSITION;  
    float2 uv:TEXCOORD;  
};
```

って型を作る。頂点シェーダはこいつを返すようにしてやる。現在のところ頂点シェーダの戻り値がfloat4とかになっていると思うが、こいつをこのVにしてやる。

当然だが、戻り値がSV_POSITIONではなく、Vになつっているため、関数に書いてあるセマンティクスは消してしまっていい。

さらに、頂点シェーダへの入力にUV情報が追加されている(レイアウトを変更したので)わけだから、こういう場合は引数情報が増えるので、増やしてやればいい。

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

こんな感じですね。uvのセマンティクスがTEXCOORDだったから、第二引数のセマンティクスもTEXCOORDになっています。float2つ分なので、float2ですね。

ただし、このuv値は今のところは流すだけです。要はピクセルシェーダ側に情報を渡すためだけに使われます。

既に構造体を作っているので、その各要素に値を突っ込んで、返してやればいいだけですね。

```
v.pos=mul(mat,pos);  
v.uv=uv;  
return v;
```

とでもしてやりやいいわけです。

もう最後!ピクセルシェーダ側だよ。

さて、もう最期だ。

既にテクスチャオブジェクトができているとは思うが、こいつの仕様がこうなっている。

[http://msdn.microsoft.com/ja-jp/library/bb509700\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb509700(v=vs.85).aspx)

そしてざーっと見てもらえば分かるようにいくつかのメソッドを持っている。今回はこの中の Sample というメソッドを使用する。

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

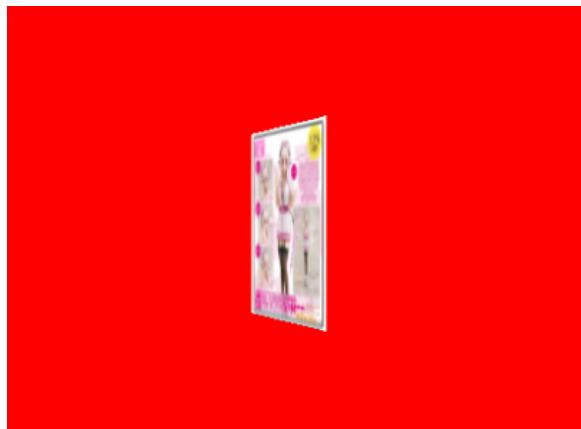
見て分かるように、こいつは2つの引数を受けとり、テクスチャ上のロケーション(場所)を指定することで、その場所の画素値を返す関数なのだ。

さて、1つ目の引数は、すでに渡しているサンプラーステートを渡してやればいい。

2つ目の引数は、頂点シェーダからもらってきたUV値(この値はピクセルシェーダに渡ってきた時点で、頂点間のUV値を補間しラスタライズした後の値になっている…よく分からんかも知れんが、とにかく、その画素に当たる場所のUV値が入っている)である。

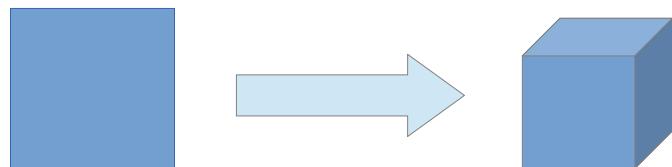
このUV値が結局は2DのUV座標値になっているため、第二引数にはこいつを渡してやればいい。

ここまでできれば、



このように絵を貼り付けることに成功するだろう。

さて、この後は無数の頂点を扱っていく事になるので、その前に練習をしてもらう。実際、このペラポリゴンだけじゃ3D扱っているって感じしないだろうからね。そこで、ひとまずは、このペラペラを立方体にしてもらう。



なお、今回はまだシェーディング(陰影つけ)まではやらない。あくまでも立方体の形にしてしまうだけだ。

この程度なら、たぶんこの時点で遅れている三分の1くらいの皆さんにもできるだろうからやってみてください。そうだね、今週末までにこれができてたらよしとするよ。

今週末(9/27)までの課題

課題①

立方体に画像を貼り付けて、くるくる回して下さい。

ヒント：1枚のペラペラを6枚用意すればいい。立方体は六面体だから、それで十分である。

基本的には、頂点を6倍に増やして、それぞれの頂点を3次元的に配置していけばそれで済む話なんだけど、Draw関数をもう一度読みなおしておこうか。

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

はい、第一引数が頂点数。次が最初の頂点のインデックス。となっている。インデックスってのは「何番目の～」の「何」に当たる部分ね。

で、現在ペラペラ1枚あたりDraw関数を一個コールしているので、…この先インデックスバッファだの何だのやっていけば一回のDrawでひとつのモデルを描画できるようになるんだけど、まあ、たぶん今いきなりやっても意義がわからないうちだろうから、このDraw関数を一つの四角面あたり1回。合計6回読んで立方体を表示する。

で、この場合の第二引数がポイントで、四角平面一個目は確かに0だが、次は4からになる。当然だよね。一枚目で0~3までの頂点使っちゃってるんだから。

context->Draw(4,0);

context->Draw(4,4);

:

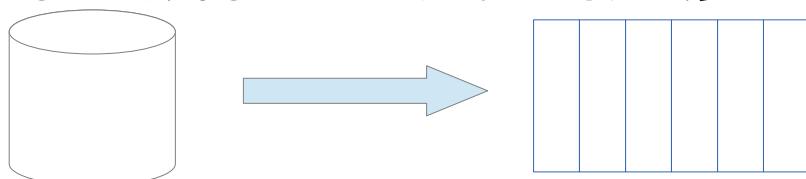
:

あとはわかるな？

課題②

円柱(上下のフタはしなくてもいい)に画像を貼り付けて、くるくる回して下さい。

ヒント：フタをしなくてもいいので、トライアングルストリップのみで実現できる。円柱ということは展開すると下の図のように、短冊を細かく並べたものになるので、これの



座標を、三角関数を用いて並べていけばいい。これはひとつのトライアングルストリップができるので、Draw関数はひとつでも済む。うへんたぶん36角形くらいでいいんじゃない

いいかな。

とりあえず課題1はこんな感じで。…まあ、分かる人は10分もあればスクできるでしょ。



課題2はこんな感じ。三角関数苦手な人はたぶん絵とか図とか書かないとできないと思うので横着せずに図を書いて考えよう。

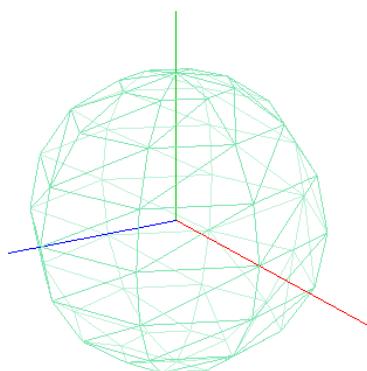


ちなみにカーリングが自動でかかるってるので、円柱の裏側は見えなくてもいいです。なお、余裕のある人はフタをしてみてください。そして先程も言いましたがDrawは一回ですみますね。

```
context->Draw(sizeof(V)/sizeof(TEXVERT), 0);
```

あ、くれぐれも言っておくと、これ、意味もわからないままに写しても動かんからね？ sizeof()が何だったのか、そう、sizeof(変数)ならその変数が食いつぶしているメモリの大きさだよね。sizeof(型)なら、その型のサイズが返る…と。

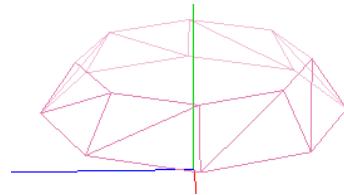
さらに余裕がある人は、球体作ってみて下さい。結構難しいですよ？



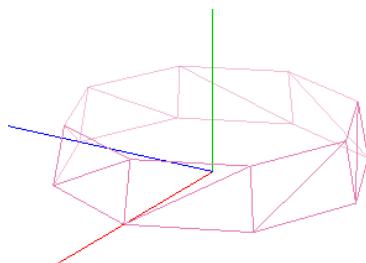
でも、円柱の応用です。頑張って球体考えてみて下さい。

円柱の場合は、Y方向が変化しても、X,Zに影響はありませんでしたが、それをYによって大きくなったり小さくなったりするわけです…うーん。全体で考えるとどうやるのがわ

がらないかもしませんが、結局は、



と、



が位置的につながっているだけの話ですよ？微分の話じゃないですが、球体なんて円柱が沢山重なっていると考えれば、そこまでむずかしくはありますまい。

はい、まあここまでやれば、恐らくは頂点とか、テクスチャとか、UVに関してちょっとくらいは慣れて大体の仕組みが分かってきたのではないかと思う。

では、早速ではあるが、PMDファイルを読み込み、頂点同士をつないでワイヤーフレーム表示をやってみよう(m90からの表示は、C++の文字列処理が面倒なので飛ばします)

PMDからワイヤーフレーム表示

ついに…やってまいりました。ミクちゃんを表示するための第一歩がついに…。そして皆さん、m90飛ばしちゃったので、バイナリファイルです。C#ばかりやりやってきた人にはハードルがちと高いかもしませんが、バイナリです。

どのみちこれ分からんとDX11とかまともにいじれないし、プログラマやってる以上は避けては通れない世界なんでこれを機会に慣れていく。

で、このへんから、「前を写さないとプログラミングできない」ような似非プログラマは徐々に＼(^o^)／オ外になりますので、気を抜かないようにしましょう。

ただし、ここに来た時点では既にPMDから読み込み、モデル表示まで終わっているツワモノたちは、ここらへんから別行動をとってもらう。おもんないやろ？授業はもう君たちができているここまでしかやらないのだ。

新たな課題だ。はっきり言ってしまうと、この授業の最終到達地点はあくまでPMDをアニメーションさせることだ。

当然、そんなものは企業に対する売りにするには不十分である。あえて「そんなもの」

と言ってしまおう。

だから、もうガチでゲーム会社に行きたくてかつPMD表示まである程度できている人は、そんなレベルの低いことで満足してはいけない。次の表現を目指すのだ。別にそいつらの名前言わなくても自分で分かってるだろ？言わせんなよ恥ずかしい。

例えば

- 視差マッピング
- 被写界深度の表現
- ディスプレイスメントマッピング
- サブサーフェスキャッタリング
- ソフトシャドウ
- ディファードシェーディング
- ファー表現
- アンビエントオクルージョン
- ライトブルーム
- 物理ベースレンダリング

などなど、ここに書いてある程度の話であれば、インターネットを調べればいくらでも出てくるし、これで満足できない最強の人間は、既にヘキサドライブにて公開されている資料や、CEDILの資料を見て、自分なりに研究して、新たな表現を目指して欲しい。

君等のライバルたちはこの学校ではないのだ。どこぞの東京大学生とかなのだ。覚悟して、さらなる高みを目指して欲しい。

ちなみに、技術系のPDFやPowerPointがあるサイトを紹介しておこう。

Cedilは

<http://cedil.cesa.or.jp/search>

初回に登録とかが必要だが、一応「完全無料」なのでさっさと登録しちゃおう。別にエロサイトに同時登録とかされないので安心して登録→閲覧してほしい。

あと、ヘキサドライブ研究室

<http://hexadrive.jp/index.php?id=12>

ぬこイラストがかわいいですね。

3Dグラフィックスマニアックス

<http://news.mynavi.jp/column/graphics/001/index.html>

トライエース

<http://research.tri-ace.com/>

いかついおにーちゃんのページ

<https://sites.google.com/site/takahiroharada/>

で、ここからは、そんなものエンもユカリもなさそうなゆとりな人のための初步的な説明を始めていきます。

…あと、最後に、どこぞのコードを良く調べもせずに取ってきて変な不具合が起きてても俺あ知らんからな?別に、俺のコードを参考にしてないから気を悪くしてるとかじゃなくて、俺もDX11の仕様を全部網羅してるわけじゃないんで、別のやり方されて…更にそれで不具合出されると、ぱっと見て分からへんねん…未熟者ですんまへん。

そこは悪いけど自分で何とかしてや、ホンマ。

バイナリファイルとは…

そうビビるな。大したものじゃないよ。テキストファイルと一緒に。ホントはね。なんだ
けど、バイナリファイルは大抵の場合は、例えば整数を整数のまま保存している。

え…？当たり前やん？整数型を整数型のままファイルに保存するのアッタリマエでしょ～？

そう…思うか?違うんだなあ…違うんだわ。うん、あのな?君等がよく見ているいわゆるテキストファイルの中の数字、これのほうが不自然なんだわ(少なくともコンピュータにとっては…)

この数字、テキストファイル上ではあくまでも数'字'なわけなのよ。例えば255って数をテキストとして保存すると'2','5','5'という文字が並んでいるだけなのね?

こいつをファイルから読み取る時、いちいち'2','5','5'をatoi等の関数で整数に変換してるので、ごつついロスなわけですよ。

例えば、255ってのはバイナリで保存すると $0xff=255$ で保存されている。これはコンピュータにとっては、「そのまま」保存してることになるわけ。

事が浮動小数点ともなると、文字保存だとかなりのロスになる…だからこんなデータで保存されることになる。はい、最初の方は辛うじて読みますが、「モデリング&データ変換：Lat」の下くらいからもう読みませんね…。

なんだよ”_Aキーゼン”って思うよね？

まあ、なんでこういう状態になっているのかって話をしよう。例えば、数字で123456789って数字があったとする。1億2345万6789ってわけだ。これを数字で保存すると、'123456789'という9文字になる。当たり前だね。

ところがこれを数値のまま保存して、例えば16進で表すと

75BCD15

となる。ちなみに、文字っていうのは、この数字で表現されていて、アスキーコードって言うんだけど、文字は基本的に0~255の数値で表されている。これを文字にすると、AだのBだのになる。

参考

http://www9.plala.or.jp/sgwr-t/c_sub/ascii.html

で、これ、127超えると、2つ組み合わせて漢字とかを表現するものになったりする。結果として、わけわからん文字列だらけになるわけだ。

ちなみに綺麗にまとめると、こう

header.magic[0]	50 6D 64
header.version	3F800000
header.model_name[0]	4C 61 74 83 7E 83 4E 56 65 72 32 2E 33 4E 00
header.model_name[16]	FD FD FD FD
header.comment[0]	4C 61 74 8E AE 83 7E 83 4E 56 65 72 32 2E 33
header.comment[16]	4E 6F 72 6D 61 6C 83 82 83 66 83 8B 0A 56 65
header.comment[32]	2E 32 30 31 30 30 36 32 37 0A 83 47 83 62 83
header.comment[48]	82 CD 81 75 30 2E 33 81 60 30 2E 36 81 76 82
header.comment[64]	82 E7 82 A2 82 AA 83 49 83 58 83 58 83 81 82
header.comment[80]	82 B7 0A 0A 83 82 83 66 83 8A 83 93 83 4F 81
header.comment[96]	83 66 81 5B 83 5E 95 CF 8A B7 20 81 46 4C 61
header.comment[112]	0A 43 6F 70 79 72 69 67 68 74 09 81 46 43 52
header.comment[128]	50 54 4F 4E 20 46 55 54 55 52 45 20 4D 45 44
header.comment[144]	41 2C 20 49 4E 43 00 FD FD FD FD FD FD FD FD FD
header.comment[160]	FD
header.comment[176]	FD
header.comment[192]	FD
header.comment[208]	FD
header.comment[224]	FD
header.comment[240]	FD
vert_count	00003DE7
vertex[0].pos[0]	BDD3B3A7 4181C4FF BF31D29E
vertex[0].normal_vec[0]	BF7FDCE0 3CD08435 3CA8B0AD
vertex[0].uv[0]	3F5779A7 3E3D9A95
vertex[0].bone_num[0]	0021 0021
vertex[0].bone_weight	64
vertex[0].edge_flag	00
vertex[1].pos[0]	3DD52D23 4181C4FF BF31D29E
vertex[1].normal_vec[0]	3F7FDCEA 3CD05E1B 3CA891DA
vertex[1].uv[0]	3F604428 3E3F30E8
vertex[1].bone_num[0]	0021 0021
vertex[1].bone_weight	64
vertex[1].edge_flag	00
vertex[2].pos[0]	BDD3B3A7 4181F7E4 BF394707
vertex[2].normal_vec[0]	BF7FDCE0 3CD08435 3CA8B0AD

よく見ればわかるように、座標情報、法線情報、UV情報がズラーッと並んでいる。このシンボルファイルはサーバに上げてるんで、見ておこう。

あと、ちなみにこのシンボルファイルを読み込んでPMDファイルを解析するにはTXNBNってバイナリエディタを使用します。

<http://www.vector.co.jp/soft/dl/win95/util/se094825.html>

からダウンロードしましよう。

で、exeと同じ場所にシンボルファイルを置けば、PMD読み込んだ際にこのようなデータになります。

で、読みにくいファイルと思うかもしれない。だけど実はコンピュータにとっては非常に読み取りやすい形式なのである。

嘘だと思うのなら、これ以降の話をmqoやxファイルでやったって構わないよ?…死ぬだけだから。

整数型だの浮動小数点型だのが、そのまま保存されているので、読み込みさえすれば、頂点データが全て自動的に出来上がっているわけだ。

じゃあ!さっそくPMDファイルを読んでいこう!!

PMDファイルを読む

ここからはさっきまでやってきた事とかなり違う事をやっていきますので、新しくプロジェクトを作り、初期化とメインループ部分だけをコピペして全く新しい気持ちでコーディングしていくことをお勧めします。

もしくは、さっきまで作っていたプロジェクトから何からを丸々コピーして、そこから今回の件には必要ない、立方体データ作成部分やら円柱データ作成部分を捨てておきましょう。

プログラミング苦手な人に限って「捨てる」と嫌いますが、プログラミング苦手だからこそ、捨てるべき部分は捨てたほうがいいのです。そこをケチっても頭が混乱してしまうのなら意味が無いどころか害悪ですからっ…!

新しくプロジェクト作って、初期化がS_OK出るの確認して、メインループ書けた?じゃあ行ってみよう。

まずは文字通り読んでみましょう。ヘッダ部分から読んでみます。

PMO のヘッダ部分

①シグネチャ

header.magic(0) 50 6D 64

はい、magicとか書いてますが、一般的に言う「シグネチャ」ってやつです。このファイルがどういうファイルなのかってのを表しています。50 6D 64ってのはPmdって文字列を表しており、読み手はこれを見て、このファイルがPmdだって判断するわけ。

実際、拡張子だけで判断していると、たまに阿呆がbmpの拡張子をpngに書き換えればpngファイルになると思ってるみたいなんですね。こういうバカにはいじれない場所にファイルの種類を埋め込んでいるわけ。

そういうのと同じように最初の3バイトを読めばこれがpmdってわかるわけ。ここでちょっとこのフォーマットに文句言いたいのは、なんで3バイトって言う中途半端な数のかってことなんや。大体、32ビットマシンなら、キリの良いバイト数は4の倍数のはずなんだけど、ここが3バイトなばっかりに、色々と処理しづらいのだ…。

②バージョン

00003 header.version 3F800000

はい、バージョンです。エエーッ！！いやいや、バージョンってのは1.0.5とかそういうやつじゃないですかアッ！なんですか3F800000って!!!!

まあ、慌てるな。まだ慌てるような時間じゃない。

こいつは実はfloat型を保存した形なのだ。例えば

```
unsigned int i=0x3f8000;
```

```
printf("%f",*((float*)(&i))); //メモリ上のuintをfloatとして扱う
```

とでも書けばわかるからやってみよう。そうすると、1.0000が出ると思います。バージョンを浮動小数で表しているだけなんですね。

結局、整数だろうが浮動小数だろうがメモリ上のビットで表現するしか無いので、バイナリエディタで読むとバイト表現になっているというだけの話。

③モデル名

20文字でモデル名を表現しています

④コメント

256文字でコメントを表現しています。

⑤頂点数

4バイト整数で頂点数を表現しています

まずはこのヘッダを読み出すところから話が始まる。つまり、ファイル入出力だ。C++のファイル入出力…しかもバイナリデータと来たもんだ。

とはいってほど難しいわけでもないので、一緒に頑張ろう。ファイル読み込み(ヘッダ部分)

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。プ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 側は引数が多いくらい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。

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。

あ、3年生で fopen の使い方が分からないという人は、C 言語の先生に謝ってきて下さい。

<http://www.orchid.co.jp/computer/cschoo1/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。

最後のrbはバイナリモードで読み込むって意味です。これがテキストモードになってると、¥rがすっとばされますので、バイナリファイルには適切ではない(^ω^)おっ。

これでオープンします。

FILE* fpってのはファイルポインタというやつです。実際はポインタというより、ファイルハンドルとして扱われます。

専門的なことはともかく、ファイルのオープンができました。fpの値にNULLが入っていないことをご確認下さい(NULLだとファイルが見つかってないなんかでバグってる)。

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。とりあえず、まずはこういう構造体を作りましょう。

```
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  
};
```

C#と違ってC言語のありがたいことは、freadを使用すればこれらの情報が1命令で取得できるということだ。

fread

<http://www9.plala.or.jp/sgwr-t/lib/fread.html>

定義はこうだ。

```
size_t fread(void *buf, size_t size, size_t n, FILE *fp);
```

ファイルから、sizeバイトのデータをn個読み込み、その内容でbufの指すアドレスに書き込んでいく。

で、さっきも言ったがC言語は、構造体を作ってしまえば、その構造体ぶん一気に読み込めるという話をしたが

どういうことがという

```
PMDHeader header;
```

```
FILE* fp = fopen("cube.pmd", "rb");
fread(&header, sizeof(header), 1, fp);
```

とか書けば一気に読み込むことができます。でけるんですけど…ここで問題があるんですよ～。ああ、これがまたC言語が嫌われる理由なんだろうと思うんだけどオ

先頭に3バイト文字列あるやん？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。これにより、データがガンガンぶつ壊れてしまう。例えば、今回の場合はパージョンがトンでもない数になったり、頂点数が0になったりする。もうね、こうなったらデータは使い物にならん。

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

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

で、とある変数がこのメモリにアクセスしようとした時に、

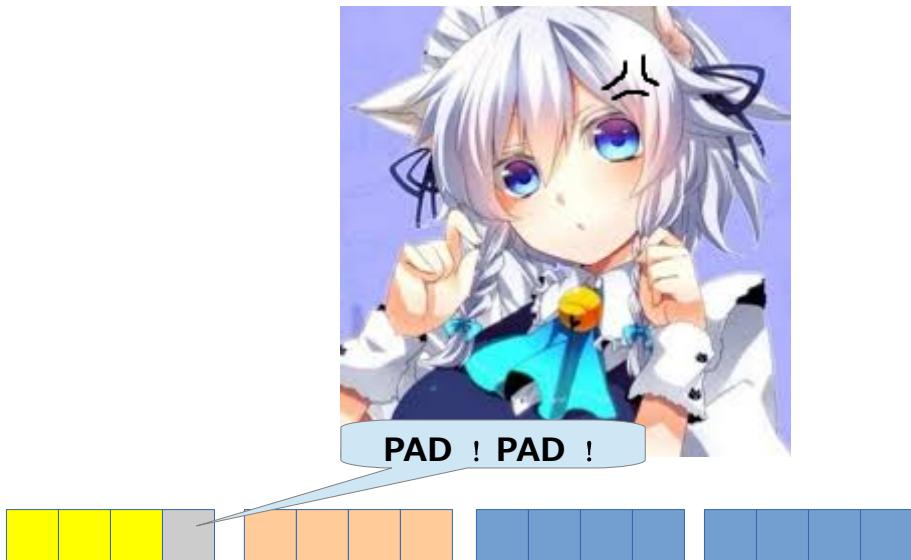


結局1バイト毎に、こういう区切りで並んでいるわけ。でね、このメモリをまたぐようなアクセスが多くなると、処理を食われるわけ。なぜなら32bitコンピュータは4バイトごとにメモリにアクセスするからだ。これがまたがった状態になると前のメモリのここまで、次のメモリのここまで…みたいに少しずつ処理がかかるのである。例えば今回の場合であれば、アライメントがなれの場合



のように、versionの部分が、前の4バイトのケツと、次の4バイトの頭3バイトにまたがるわけ、これでは処理が重くなるのだ。

なので、コンパイラがご親切にも(おせっかいにも?)パディング(詰め物)してやって、あなたのプログラムを速くしてあげましょうって事なのだ。



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みつつの選択肢がある。

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昔はそうだったのだろう。ということで、言語仕様として既に対処されてたりする。

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。

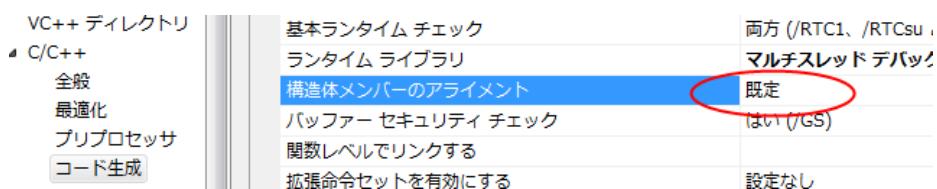
```
#pragma pack(1)
```

と書けば、これ以降に書いた構造体のまとめバイト数が強制的に1になる。もちろんそのままにしてたら処理速度に影響が出るため、構造体の定義後に

```
#pragma pack()
```

と書いて規定値に戻す。

最後に、もう、全てのまとめバイト数を1にしてしまう方法だ。あまりおすすめしないが。



ここをいじって、パッキングバイト数を1にしてしまう。

ともかく、これらの問題に以上の3つのどれかで対応すれば、頂点数までわかるので、今日の所は、モデルの頂点数を確認しておこう。…まあ、フツーにシグネチャを4バイトにしてくれてればこんなのが説明する必要ないのに…1バイトケチってどーすんだって問い合わせたい問い合わせたい小一時間問い合わせたい。

ひとまず、サーバにあるcube.pmdの頂点数を確認してくれ。たぶん24頂点だと思う。それができたら、君の愛するモデルをダウンロードして頂点数を確認してくれ。

ちなみにアリスたんは12467頂点でした(*'ゞ')ハハハ



頂点データを読んでいこう

それではイヨイヨ頂点データを読んでいきましょう。既に頂点数が分かっているので大したこたあねえんだよ。

とりあえず頂点データは

- 座標情報：float3
- 法線情報：float3
- UV:float2
- 影響ボーン番号:WORD(unsigned short=2バイト)ふたつぶん。つまり4バイト
- ボーン影響度(ウェイト):1バイト
- エッジフラグ(輪郭線): 1バイト

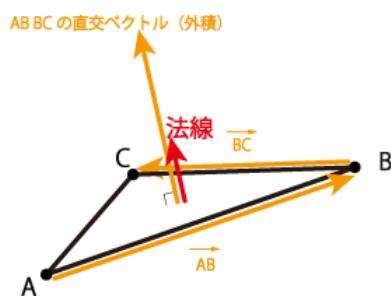
になっている。…はい、気をつけよう。こいつも38バイト(つまり4バイトの倍数になつてはいけない)

これ…わかつてないと結構死ぬ。…それなりに死ぬ。だからなんで…2バイトをケチるのさ…樋口さんよオ…OTL。(樋口さんってのはMMDを作った人)

ところで「法線情報」ってのが入っているけど、この意味はわかるかな?今まで立方体だの球体だの作ってきたけど、それを構成するのは…そう、三角形の集合だったよね?その三角形は「面」でもあるわけ。それはわかるよね?

で、その「面」に垂直なベクトルのことを「法線ベクトル」っていうんだ。この用語、かなり後までずーっと出てくるので覚えておこう。

一つの三角形に注目すると



こういう感じになる。ね?面に垂直になっているでしょ?まあこの法線の求め方も後々教えるけど、今はこういうものだと思っておけばいい。面に対してつくのに何故頂点一つ一つに入っているのかという理由もその時にお話しよう。

次のUVはいいよね?その後の影響ボーン番号、ボーン影響度、エッジフラグはとりあえずボーンの勉強をしてからにしよう。とにかく今は頂点座標をもらってきててしまおう。

んー、まあ、フツーにガバッと頂点数ぶんのデータを一気に読むのが正解なんだけど、ゴチャゴチャするのも嫌いなんで、1頂点ずつ持ってきます。ループを使います。ループくらい分かるでしょ？

じゃあ実際に頂点データを取ってきて、画面に表示してみましょう。

手順は

- ①頂点データ構造体を定義する
- ②頂点レイアウトを変更する
- ③ループしながら頂点情報を読み取る
- ④頂点情報は分かっているので、画面上に描画

結局④に関しては、立方体やら円柱やらを表示した時とほとんど一緒なので、気をつけるべきは①～③の話やね。とにかく PMD のフォーマットに合わせて頂点データ構造体を定義しよう。

①頂点構造体を定義

```
//PMD 頂点構造体
struct PMDVERT {
    XMFBLOCK pos://頂点座標
    XMFBLOCK norm://法線ベクトル
    XMFBLOCK uv://UV
    WORD boneId[2];//影響ボーン番号
    BYTE boneW;//ボーン影響度
    BYTE edgeFlg;//輪郭線フラグ
};
```

まあ、こんな構造体でしょうか。みんなの好きにしてくらさい。

②頂点レイアウトの変更

これはちょっと…考えが必要ですよ。

頂点座標と UV は変わらないのですが、まず法線ベクトルを、その間にれる必要があります。で、フォーマットとセマンティクスですが…

法線ベクトルは float 型です。ですから_FLOAT と書いてあり、32って書いてるのを選ぶのが正解…そして3つの要素を持っているんだから、頂点座標と同じフォーマットでいい。つまり DXGI_FORMAT_R32G32B32_FLOAT となる。

次にセマンティクスですが、法線のことを英語で normal といいますので、“NORMAL”とかいておいて下さい。

次の影響ボーン番号ですが、これは…ちょっと微妙なのですが適當なのを割り当てておきましょう。16bit2つ分で、しかも整数なのでDXGI_FORMAT_R16G16_UINTってところでどうか。セマンティクスは…これまた適当に“BONE_ID”とでもしておきましょう。

最後にボーン影響度と輪郭線フラグですが、もう面倒なのでまとめておきたいのですが、輪郭線フラグと影響度ってまとめると後が面倒っぽいんで一応分けておきます。8ビットです。それが1つなので、フォーマットはDXGI_FORMAT_R8_UINTかな～。それぞれ、“WEIGHT”と“EDGE”とでもしておきましょうか。

さて、これでレイアウトと、それを放り込むための構造体ができました。では、早速頂点データを取得していきましょう。

③頂点データ読み込み

はい、前にも話したとおり、`fopen`→`fread`→`fclose`を使用します。あ、くれぐれもpmdファイルを自分のプログラミングしてローカルに置くようにね。

前にも言ったように

```
FILE* fp=fopen("cube.pmd","rb");
```

でファイルを開き、

```
fread(&pmdvertices[i],38,fp);
```

で読み込んでいきます。ここで注意点は、`sizeof(PMDVERT)`などとはしていないこと。もちろん`pragma pack(1)`すればいいんだけど、個人的には使いたくない。なので直値を入れている。

モチロン、プログラミングにおいてこのようなマジックナンバー(ヨクワカラナイ直値)を入れることはタブーとされてはいるが、しゃーない。常識にとらわれてはいけないのでよ(•`ワ•')

はい、さて、C++で動的配列を確保するには、いくつかやり方があって、C言語的に行くなら`malloc`で確保して`free`で解放してもいい。

`new()`で確保して、`delete()`で解放してもいい。

だが、個人的には、`vector`というやつを使いたい。ということで使う。C#のListみたいなものだと思ってくれればいい。

まずはインクルード

```
#include<vector>
```

今回も.hを付けないのがポイント。List分かってる人はすぐわかると思うけど、こい

つは配列みたいにして使えるわけ。

まずは

```
std::vector<型名> 変数名;
```

で宣言すれば、この変数名が、ほぼ配列的に使えるようになる。

既に頂点数は分かっているんだから、

```
std::vector<PMDVERT> pmdvert(頂点数);
```

とでも書いておけばいい。もし、宣言後に頂点数が分かるような場合も簡単である。

```
pmdvert.resize(頂点数);
```

とやれば、サイズ変更も簡単にできる。

あ、ちなみに fopen は stdio.h もしくは iostreamあたりをインクルードしなければ使えないかもしないので、fopen(と書いて、何のヒントも出てこなければ、インクルードしておいてくれ。

で、あとは、これで取ってきた頂点データを使って、PMD を表示するだけだ。

④頂点から PMD を表示する

とは書いたものの、自分でなんとかできない子も多いだろうから、一応間違えないように書いておく

```
bufdesc.ByteWidth=sizeof(PMDVERT)*vertices.size();
```

当たり前だね。頂点バッファーのサイズが変わってるんだから、ここは書き換えなきゃいけない。

```
subresource.SysMemPitch=sizeof(PMDVERT);
```

これもやね。

```
UINT strides=sizeof(PMDVERT); //←ここも変更
```

```
UINT offsets=0;
```

```
dcs.Context()->IASetVertexBuffers(0, 1, &buff, &strides, &offsets);
```

あとは基本変わらないので、もしエラーが出てきたら、エラーメッセージと格闘しよう。

で最後は

```
Draw(vertices.size(), 0);
```

ですね。vector は .size() というメソッドでその配列要素数を知ることができます。

で、vector はかなり人を選びますので、分かんない人は、…まあ既に TXNBIN 上で、要素数が分かってるんだから、もう最悪直値を入れて配列作っちゃっても構いません。お任

せします。

で、どうなったでしょうか？

たぶん、壊れたのではないでしょ？頂点は間違いないはずなのに…まあ、それはな、実はまだお話ししていない「インデックスバッファ」ってのを設定していないからなんや。ただ今は、とにかく頂点を読み込むのが目的なので、その話はもう少し後にしよう。

今の状態を気にしたくないのならば

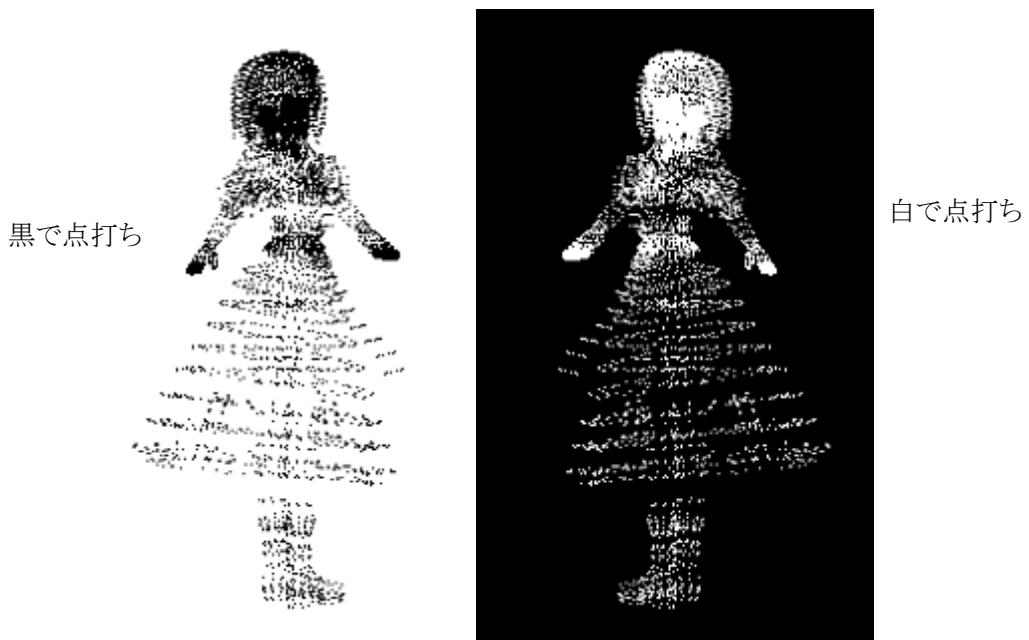
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP

って書いてある部分を

D3D11_PRIMITIVE_TOPOLOGY_POINTLIST

にしておこう。そうすれば頂点しか表示されないので、悩むことは無いだろう。ただし、そうなると立方体の場合、頂点数が少なくて分かりづらいので、もうちょっと頂点数の多いものをとてこよう。

ちなみに俺の愛するアリスたんだと、こうなる



ギギギ…くやしいのうくやしいのう…。

おんどりや！わしゃ絶対にアリスたんを表示してみせるけえ！！

ハイ、というわけで、今週末までの課題ね。君の愛するキャラクターのPMDモデルの頂点を表示して下さい。

課題提出期限:10/4(金)

提出場所: gakuseigame¥ゲーム¥DirectX11¥PMD 頂点課題

いつもどおり、学籍番号_名前のフォルダにexeとpmdモデルを放り込んで提出して下さい。

ということでさっきから出し惜しみしていた「インデックスパッファ」の解説を行います。



ところでマニュアル(MSリファレンス等)はきちんと読んでる?

例えばこのページ

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

ラクするために非常に大事なことが書いてるんだ。だが、MSのマニュアルは俺様のように親切ではないのだ。だから見落としやすいが、お気づきだろうか?

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、 4×4 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値です(「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

とっても楽ができるヒントが隠されていることに気がついただろうか…

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。

日本語なんだから、きちんと読みましょうね。

色々と、役立つことが密かに書いています。なんかうまくいきないとか、面倒な部分はこういうのをよくよく読みなおすことをオススメいたします。

手を抜くための努力を惜しまないようにしましょう。

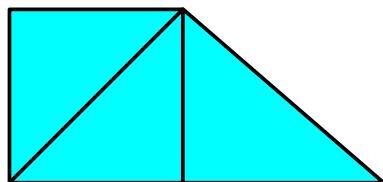
インデックスバッファとは

ギギギ…ここまででも死にかけてるのに、まだ新しい概念が出てくるとは…わりや殺す気が！

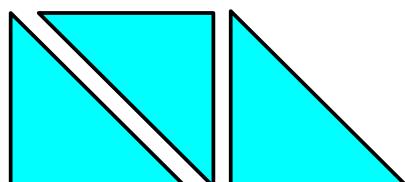
まあ待て、まだ慌てるような時間じゃない。…大したことじゃねえよ。今さ、頂点データって、38バイトじゃん？

で、例えばね、これだと5頂点じゃん？

でもさ、結局三角形の集合にするなら



これを3つの三角形に分割して…

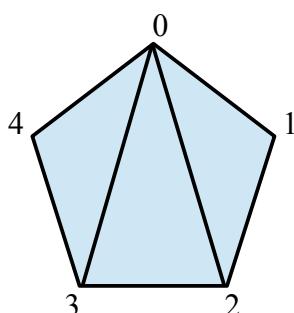


ってやると頂点が増えてまうやろ？増えてまうやろ？3頂点が3つぶんで9頂点や。そのままやと、頂点データだけやといくらでも38バイトって感じで増えてまうんや。

この例だと2Dなんで、そこまで増えてないけど、3Dだと立体的に頂点が増えてしまい、共有頂点が多くて大変なことになってしまふ。

なのでどうするのかというと、頂点ひとつひとつにインデックス(番号)をふるわけ。番号ってのはunsigned intで済むから、4バイトで済むわけ。38バイトとえらい違いでしょ？

だから、頂点のそれぞれに見えない番号が付いていると教えてくれ。ていうか、頂点ロードしたでしょ？あのインデックスを並べて三角形にしてしまうのよ。



たとえば、こんなふうな順番に頂点が定義されていたとする。そうすると、三角形は $\{0, 1, 2\}, \{0, 2, 3\}, \{0, 3, 4\}$ となる。ちょっとこの例だとTRIANGLE_FANみたいになってしまふ

まっていますが、勘違いしないで下さいね。とにかく配列の番号を3つ用意することで三角形を表現する。それだけです。

で、ここで、最初に立方体を作って、あれが壊れていたとは思います。…場合によっては壊れてなかったかもしれません…。PMDはインデックス情報ありきで作られますんで、順番めちゃくちゃだったりするんですわー。

で、手順としてはインデックスバッファ作ってGPUに流し込むって流れになるんだわー。

インデックスバッファ作成

…さあ、インデックスバッファを作成しようか。作成の仕方は頂点バッファの時と似たようなもんです。

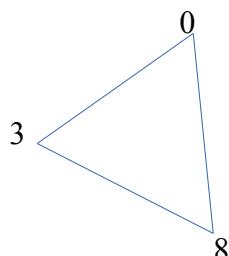
D3D11_BIND_VERTEX_BUFFER

って指定してた所あったでしょ？それを

D3D11_BIND_INDEX_BUFFER

にするんですわー。詳しくはこの後やってみます。で、インデックスの役割は結局のところ、「**三角形(面)をつくるためのもの**」とでも考えて下さい。基本的に頂点だけでははばらんばらんの点データでしかない…と。

だから、インデックスデータって考え方がしっくり来ない人は、「面データ」って捉えておいて下さい。



どの頂点とどの頂点とどの頂点で面を構成するのかっていうデータってことね。なので、インデックスデータは、3つの整数值を持てばいいわけ。

で、早速インデックスバッファを作っていきたいんですが、まずその前にPMDのデータのどこにインデックス情報が入っているのかを確認してみましょう。

.ADvertex[23].bone_weight	00
.AEvertex[23].edge_flag	00
.AFface_vert_count	00000024
.B3face_vert_index[0]	0003 0001 0000 0000 0002 0003 0005 0008
.C3face_vert_index[8]	0009 0009 0004 0005 0007 000A 000B 000B
.D3face_vert_index[16]	0006 0007 000C 000D 000E 000E 000F 000C
.E3face_vert_index[24]	0010 0011 0014 0014 0015 0010 0016 0017
.F3face_vert_index[32]	0012 0012 0013 0016
.FBmaterial_count	00000001

ちなみにPMDは頂点データのすぐ後に、faceという名前でインデックスデータが入っています。

で、もしかすると、でかいデータを扱っている場合は、頂点が多すぎてface情報を見るまでが大変だと思いますので、こうして下さい。アリスを例にお話します。

アリスは頂点数が12467であるということが前回の調査で分かっております。そして、頂点1つあたりのバイト数が38です。 $12467 \times 38 = 473746$ 。これを16進にすると73A92

ここまで判明したらTXNBBIN上でまず、頂点データの先頭にカーソルを合わせた状態でCtrl+Jを押す。そうするとどれくらいジャンプするかを聞いてきますので、そこにさつき書いた73A92を入力。

すると、face_vert_countってのが見えると思う。確認しておいてくれ。ともかくこれでどこにインデックス情報があるのかわかったんだろうと思う。

インデックスデータ読み込み

各自、プログラム上で、このインデックス数を取得しておいてくれ。

ちなみにインデックス数は4バイトだ。たのむぞ。で、面は必ず3点一組で構成されるため、インデックス数は必ず3の倍数になる。そうなっていなければなにか間違ってる。

で、今度はインデックスデータなんですが、こいつ、PMD上で2バイトでできているようござる…これだと頂点数が65535を超えた時点で死にますけど大丈夫でしょうか樋口さん…OTL。

まあ、今は幕末ゆえ致し方なしでござる。

しかたないので、いつものようにループでくつぱあーって読み込むか、今回は構造体使ってないし、いっぺんに読み込んでええんちゃいますやろか。

`fread(&indices[0], sizeof(WORD) もしくは unsigned short), indices.size(), fp);`とかで
`CreateBuffer`でインデックスバッファ作成

あれ? `CreateBuffer`って頂点バッファのためのものじゃないの?

いやいや、とにかくバッファの塊を作るために使うものなんです。そして今回はインデックスバッファを作るために使うものになるんです。

最終的にID3D11Buffer型のをもらうんでいつもどおり、ポインタのポインタで受け取ることになります。そろそろポインタのポインタは覚えてね。

- ① ID3D11Buffer* buffer; てな感じでます、ポインタとして宣言して…
- ② &buffer てな感じで、ポインタにアンパサンドをつけてポインタのポインタ扱いで `CreateBuffer` に渡すんだ。

インデックスバッファ作成の手順

うーん、これはほとんど頂点バッファの時と同じやね

① ID3D11Buffer* 型の変数を宣言

② D3D11_BUFFER_DESC 型の変数を宣言

③④で宣言した変数の中身を入れていく

④初期化データとしてD3D_SUBRESOURCE_DATA 型の変数を宣言

⑤⑥に初期化データのポインタを渡しとく

⑥ CreateBuffer の最後の変数に①で宣言した変数のアドレス(&)を放り込む

\(^o^)/終わり。

です。簡単ですね。

まあ、①②は問題無いだろうから、③から説明していく。

③ D3D11_BUFFER_DESC の中身を入れていく。

うん、ヨクワカラナイ人は、頂点バッファ作った時のやつをコピって持ってこよう。で、そこから色々書き換えていく。当たり前だけど、コピったまま頂点バッファの変数と同じ名前とかにするなよ…マジでするなよ。

それで「エラー出るんですけど…」とかホンマ、地獄の業火に投げ入れられるレベル。天からお塩！！

あっ、コピペして、名前変える。

```
D3D11_BUFFER_DESC idxBuffDesc={};
```

そして、変更すべきはByteWidthとBindFlagsのみ。まあ、BindWidthは予想ついてると思いますが、…PMDのフォーマットに合わせるのならば16ビットつまり2バイトですね。うーん。このケチリ方…キモチワルイけど、合わせましょうか。

で、既に取得していると思いますが、インデックス数も fread で読むんですよ。そろそろそれくらいできるようになっておきましょう。

で、cubeなら、36インデックス。アリスたんなら59724でした。はい。で、そうなると ByteWidth は

sizeof(WORD もしくは unsigned short)*indexCount

ということになります。

次にBindFlagsですが、今回はインデックスバッファとして使いますので、

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

こつから、インデックスっぽいやつを探しましょう。ホイ、あった。

D3D11_BIND_INDEX_BUFFER やね。

④はいいけど…

ほい、次に⑤ですねー。インデックスバッファの初期化データを放り込んでいきます。

バーテックスからの変更点はまずインデックスデータのアドレスの部分を放り込む

idxSubResource.pSysmem=&indices[0];

しつこいようですが、これをそのまま写しても動かなかつたりしますよ？

で、あとはこれらを⑥…つまり CreateBuffer の各変数に放り込むだけです。

はい、これで S_OK が返ってきたら、インデックスバッファのできあがり～。

インデックスバッファをセット

どう？バッファできた？

んじゃ、インデックスバッファ設定するだけですよ。

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

IASetIndexBuffer

ですねー。で、こいつは簡単なんだけど、第一引数は、インデックスバッファポインタ。
第二引数がなんにかな？

サイズいくらにした？unsigned short だったよね？2バイトだよね？

DXGI_FORMAT_〇〇〇_UINT

あとはわかるな？

オフセットはいつもどおり0でおながいします。

で、これでも終わりちゃう。

インデックスバッファを使って描画

Draw を DrawIndexed に変える必要が…必要があんねん。

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

はい、このマニュアル見ると、だいたい

DrawIndexed(インデックス数, 0, 0);

こう書けばいいことが分かりますね。まずはこう書いた上できちんと頂点が描画される
ということをご確認下さい。

トライアングルリストで表示してみよう

さて、このインデックス。以前にも話しましたとおり、三角形面を構成するためのものです。

a b cという番号が並んでおり、a番目の頂点、b番目の頂点、c番目の頂点から三角形を構成するというルールになっております。

チョット前にも言いましたが、その性質からこのインデックス数は3の倍数になっているという話もしました。つまり、逆に言うと無数の三角形の構成情報は既に持っていると



いうことになります。つまり、今は頂点集合であるという指定なんだけど、これを三角形集合であるという指定に変えれば、ほらこのとおり…

ちなみにテクスチャ表示をしてしまうと、ちょっとキモくなる。



うん、まあ、ここはPMDからテクスチャを読み込んだり、**マテリアル**の設定ができるようになってからにしましょう。

きちんとしたアリスたんを見るのはもう少し先になりそうです。

早速シェーディング行ってみようか

うん、なんか3Dらしくするにはもうひと頑張りって感じがするなあ。もうほんのチョット。ベクトルとか三角関数が出てくるけど、プログラミング的にはもうちょっとだから頑張ろう。そうすれば



こんな感じに恐ろしいレンダリングができるようになる。

見て分かるように明るい部分と暗い部分ができている。これをCG用語ではシェーディング(陰付け)という。

で、このへんから色々とCG系の用語、数学系の理論が出てくるので、…まあ、心してかかってきててくれ。

基本用語

とりあえず、今回使いそうなものを羅列だけすると…

- シェーディング(グローやラッポンやら)
- 輝度
- 平行光源(本当は光源の種類は沢山あるんだけど今回使うのはこれ)ベクトル
- 法線ベクトル
- 補間
- 内積
- ベクトルの正規化
- ディフューズ、アンビエント、スペキュラー

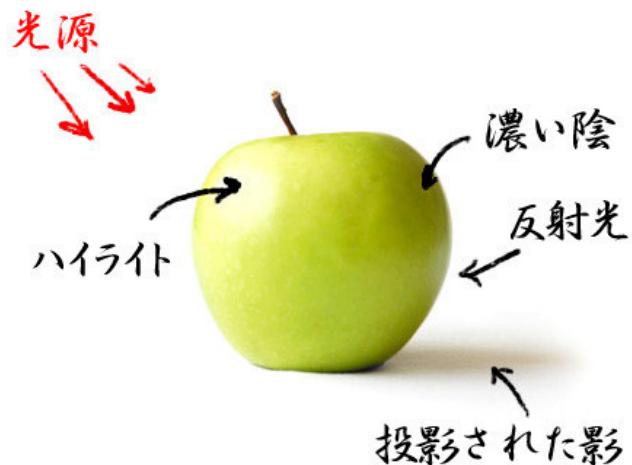
こんだけかな…いっぺんにやると挫折するしね。新しく学ぶ用語は10個以内に収めておきましょう。

最初に用語きっちりやつとかないと、たぶん、僕が何ゆーてるか分からんようになりますから。

シェーディング

シェーディング(shading)とは、「陰影をつける」って意味の shade に ing が付いているだけのもの。だから意味としては「陰影付け」って意味になる。

ちなみにヒトコトで「陰影」って言いますが、「陰」と「影」ってあるんですわー。美術部とかで活動したことある人は聞いたことがあると思いますが、こういう…



この図のようにね、物体に光が当たると、明るい部分と暗い部分、さらには光が物体に遮られて影が落ちます。この明るい部分と暗い部分を表現するのが「陰」、遮られて落ちるのが「影」ということになります。大雑把に言うとそういうことです。

で、CG用語でシェーディングって言うと、この「陰」の部分を指します。「影」の部分は「シャドウイング」とか言ったりします。

で、CGの世界には、このシェーディングのやり方にも色々あって、考案者の名前がついてたりします。古典的なものだと、グローブシェーディングだの、フォンシェーディングだの、ランバートシェーディングだの、ブリンクシェーディングだの言ったりします。

輝度

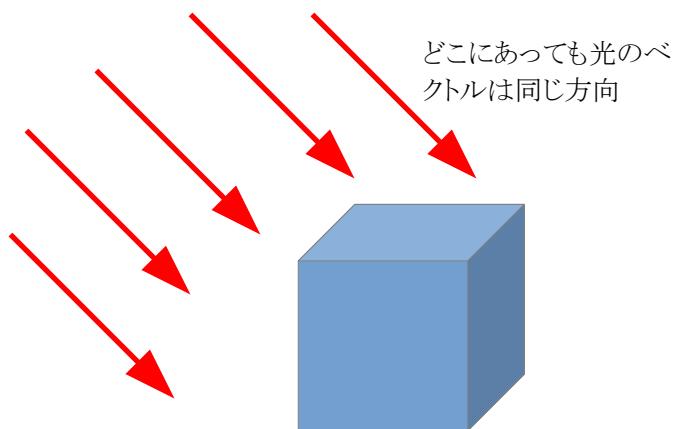
明るさを表す値で、輝度値とか言ったりします。例えば、この輝度値が0~255の範囲内にあるのならば、0が真っ暗で、255がめっちゃ明るいということになります。機転の利く人はわかると思うが、シェーダ上では輝度値が0.0~1.0になっており、0.0が一番暗く、1.0が一番明るいということになります。



白い石膏とかであれば、これだけで十分表現できます。表面に色がついてる(テクスチャが貼られていたり)奴は、その色にこの輝度を乗算(掛け算)することによって最終的に出力する色を決定します。

平行光源(平行光線)

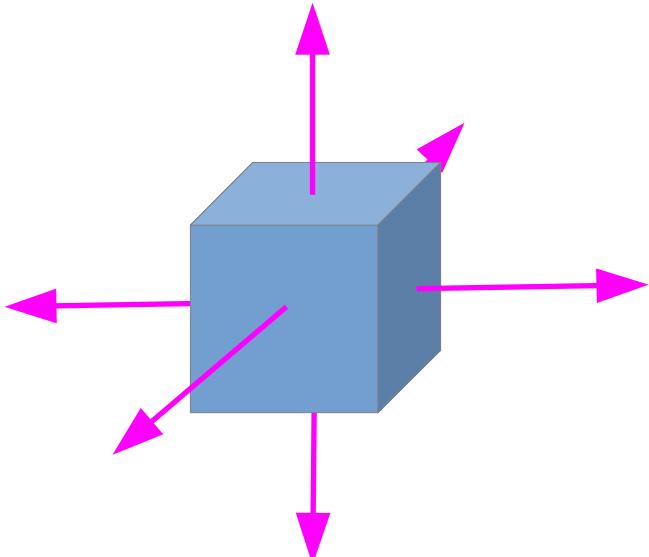
読んで字のごとく、どこに居ようと飛んでくる平行な光源(光線)です。この光線は常に同じ方向を向き、減衰もしないため、現実的には「ありえへん」光源です。なんとかというと、平行光線を出す光源(ライト…蛍光灯?)はどこにもないからです。そういう光線なので、あくまでも想像上のライトという扱いでリアルを追求するときには向きませんね。



後に説明する「点光源」や「スポットライト」などは、光源と物体の位置関係でベクトルが変わったり、光が減衰したりします。それだと面倒なので、ありえへん「平行光線」を設定することによりお手軽にシェーディングしようっていう、初心者向きのライトがあります。

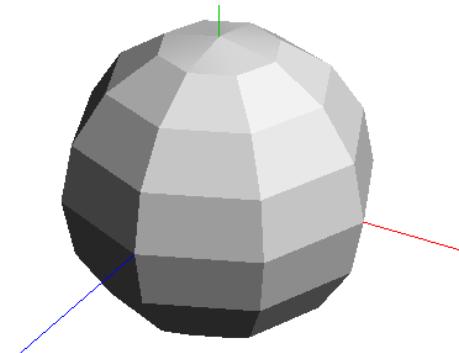
法線ベクトル

法線ベクトルとは、以前にも話しましたとおり面に垂直なベクトルです。



基本の考え方は確かに「面に垂直なベクトル」なので、図のように各面にベクトルがくっついているイメージなんですが、実際のPMDデータとしては頂点にベクトルがくっついています。PMDだけでなくほとんどの3Dデータの法線ベクトルは頂点にくっついています。

これには理由があって、スムーズシェーディングの中のフォンシェーディングってのが関わっています。スムーズシェーディングなしのレンダリング(フラットシェーディング)で、球体を表現するとこうなります。



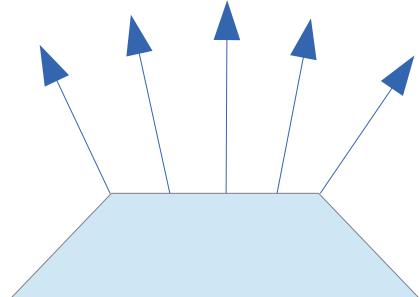
なんかスターフォックスの時代を思い出しますね。



まあ、正直言って現代のPCで表現するものじゃないですよね。ということで考えだされたのが、それぞれの面のベクトルを滑らかに補完して、表面を滑らかに見せようというものです。これをフォンシェーディングって言うんですが、これをやるのには頂点に法線くっついてると都合がいいんですよ。



実際には数学の性質上(外積から法線を作る)面ごとにしか法線を設定できないのだが、後々の便利さから、頂点ごとに法線を設定している(面の平均値など)。で、この頂点ごとの法線を、実行時にラスタライザが下図のように法線を補完することにより表面がなめらかに見えるようになるというわけです。



おわかりいただけましたか？

線形補間(リニア補間)

まあ、これは実は小学生でもわかるかも知れないような話ではある。たとえば



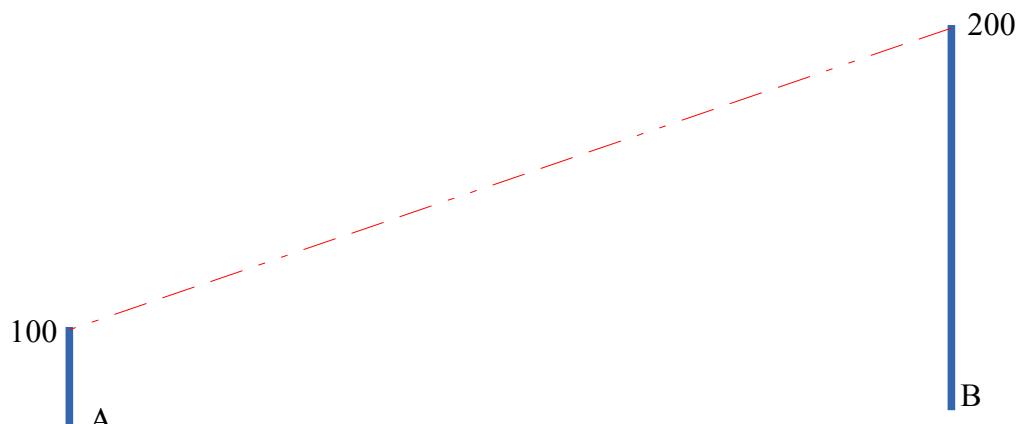
このように離れた位置に2つの値があったとする。で、この間の位置における値を推測する必要があるとする。例えば、グラデーションをかけたい時とかやね。1ピクセルずつ推測値を求める必要がある。

そうだね、小学生に戻った気分で、この2点間の距離が200であるとして間の値を推測してみよう。

とりあえず中間地点の値(Aから100進んだ位置の値)を考えてみようか。こんな小学生でも答えられるよね。そう、150だね。

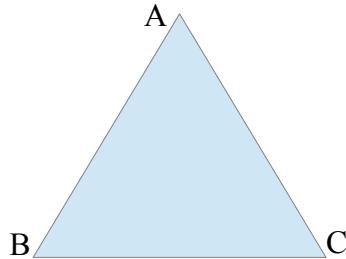
じゃあ、ちょっと難しくして、Aから50進んだ位置の値を考えてみようか?どうなるかな?そう、25になるね?できない子はいねえか?そうだよね。みんなもう成人大もんね!!じゃあ、これをめっさ細かくやってたらどんな感じになるかな?

ちょっと気の利く小学生とか、中学生ならわかるよね。

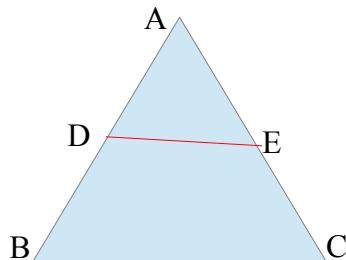


はい、図のように二点間を線でつないで間を推測する(補間する)から「線形補間(リニア補間)」なんだよ。簡単でしょ?で、これって一次元的な補間じゃないか?画像処理的

には二次元的に補間する必要があるんだ。ポリゴンの場合だと。



3点あるじゃん？イメージ的にはまずAとB、AとCの間を補間する。そうすると、例えばABの間の地点および、ACの間の地点の画素は推測できるよね？そいつらを例えればD,Eとする。

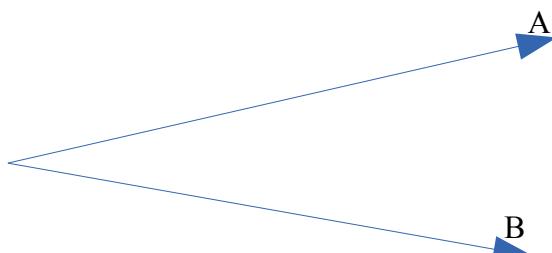


そうなると、DとEの間の画素は、DEを線形補間すれば推測できるよね？こんな風に2方向の線形補間から、中間の値を推測することを「**ハイドリニア補間**」っていうわけ。

内積

内積は1年生の時にやってるので、いまさらやるものでもないし、こんなんちょっと気の利いた中学生なら知ってる(いや…どうだろう…まあ、あくまで気の利いたやつな)。

とりあえず、証明その他は1年次でやってんので、ここでは公式のみ思い出させようか。まず、確認だがベクトルは覚えてるな？「方向」と「長さ」を持った値のことだ。



空間上における速度や加速度もベクトルで表される。とりあえずCGに関して言うと法線ベクトルやら何やらで表される数学的な値である。

で、こいつは次元が増えると要素数が増えるというアレである。二次元なら $A=(X_a, Y_a)$ ってな具合で、三次元なら $A=(X_a, Y_a, Z_a)$ ってな具合である。

で、内積だが、こういうやつである。

$$\vec{A} \cdot \vec{B} = |\vec{A}| |\vec{B}| \cos \theta$$

覚えてるかな？覚えてるね。僕がひとつこく教えたもんね。そもそも自分からこの教室に

来ようなんて人は覚えてないとダメ。何考えてんの？

…まあ、そういうやつです。ちなみにこのAとBの大きさが1のときは内積= $\cos\theta$ になるので便利です。ちなみにシェーダ上ではこの内積を出すのにdotって関数を使います。

dot(ベクトルA,ベクトルB)

これで内積値が出せます。1年の時にしつこく言ったと思いませんが、内積値はあくまでスカラーバリューであり、ベクトルではありません。絶対間違えないように。

ベクトルの正規化

内積のところでも話しましたが、AとBの大きさが1のときは内積= $\cos\theta$ になるという性質を利用しやすいように、予めAとBのベクトルの大きさを1にしておくと便利です。

で、この「ベクトルの大きさを1にする」というのを「ベクトルの正規化」と言います。

ということで、ちょっと頭の体操。

$$A=(1,1)$$

というベクトルを正規化するとしたら、どんな値になるかなあ～？ねえねえ、どんな値？ねえ、どんな値？うん、考えた？ほんとに考えてたら5秒できるんだけどね。

$$\vec{A}=\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right) \text{ だね。}$$

まあ、これは三次元でも同じで、三次元のA=(1,1,1)なら

$$\vec{A}=\left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$$

になるんだね。はい、これもhlsl(シェーダ言語)の中には関数が搭載されていて、normalizeって関数を使います。

正規化済みベクトル=normalize(ベクトル);

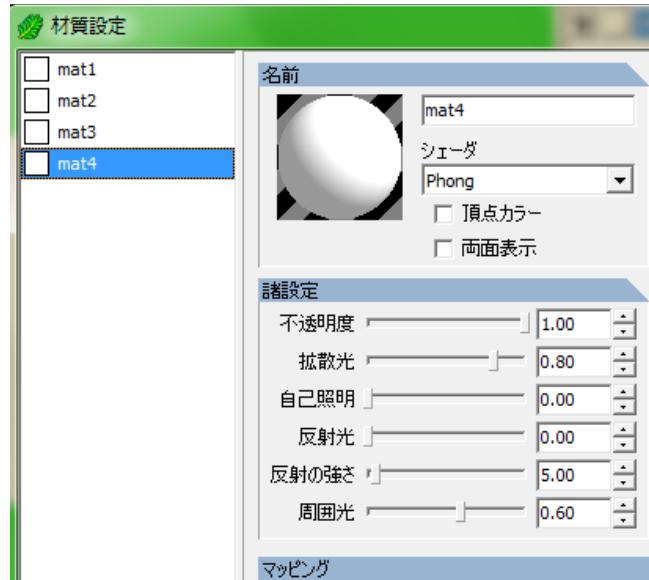
というわけです。

ディフューズ・スペキュラー・アンビエント

はい、聞いたことがあるのかな？ああ、ないんだね。あー、でも、これ、正確に答えられないようじゃやたぶん、CG検定ベーシックすら合格できないわ。

じゃあ、説明しよう。

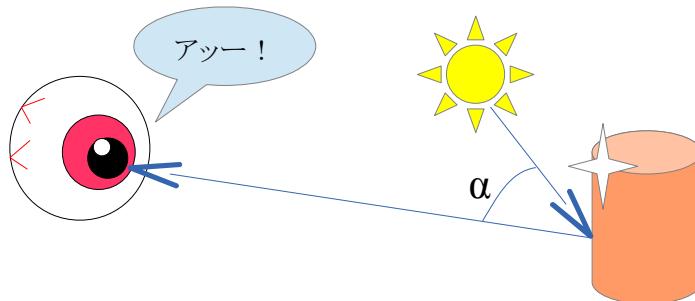
こいつらは、物体の材質を表す代表的な簡易モデルである。メタセコ立ち上げて物体作って「材質」の部分を見ると、



このように、スライドバーをいじって材質をいじることができる。この中に見える「拡散光」、「反射光」、「周囲光」ってのが、ここで説明する diffuse, specular, ambient に対応している。

ディフューズ(diffuse)ってのは、**拡散反射光成分**と言って、石膏みたいな質感の物体を表現するのに向いているのだ!!で、なんで**反射光**なのかというと、僕らが物体を見れているのは、光が目に入っているから。これはわかるね？

たとえば缶ジュースが見えるとすれば、部屋の明かりが缶ジュースに反射してその光が目の玉にズシリと突き刺さって僕らは「アッ！コーヒーッ…！」と認識するわけだ。



だから、反射光も光って扱いなわけ。なので、この反射光の計算式は

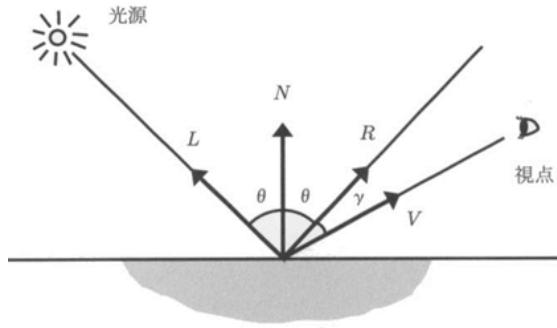
$$I_d = K_d I_i \cos \alpha$$

となる。ちなみにこのモデルは、一番簡単なモデルで「ランバート反射」という。これはこの後、プログラミングでまたお話ししていくが、お手軽に立体感を出すことができる計算式であ

る。

次にスペキュラーだが、これは「**鏡面反射**光成分」というものだ。金属的なもののハイライト表現するのに使う。PMDの場合だと、髪とか目とか、一部の衣装に使われているのではないかでしょうか。

今回はまずは拡散反射(Diffuse)のみを扱うので、ここではさらっと、計算式だけ見せておこうか



$I_s = K_s I_i \cos^n \gamma$ という。拡散反射に似ていますが、 \cos に乗数がついてますね。あと、図と見比べて欲しいんですが、 γ は、光源と法線の角度ではなく、光の反射ベクトルと、視線ベクトルのなす角度を表しています。

ホントは反射ベクトル計算するのもそれなりに頭使うんで、そのへんの話は実際にスペキュラの段階に入った時に話しましょ。

アンビエントとは**環境光のことである**。環境光ってのは空間の至る所に漂ってる明るさを表す。だから本来は非常に多くの要素を Σ する必要があるし、計算も面倒なんだが、もうそういうの全部含めてシンプルに

環境光イコール定数(下駄)

って考える。まあ、要は Diffuse とかだけでは、光の反対側にある面が暗くなりすぎるため、0.2~0.5くらいの値を全体にプラスして、真っ黒にならないようにするって思つておけばいい。

しかし勘違いはしないで欲しいのは、これはあくまでも近似、その中でもしょーもない近似であるため、最先端のゲーム技術では不十分なものであることは認識しておいて欲しい。

さて、実践に入っていこうか。まずは最もお手軽に立体感を出せるコサインシェーディング（俺が命名…なので、他所でこんな呼び方は使わないで欲しい）をやってみよう。

コサインシェーディング

何度も言いますけど、この呼び方は外で使うのはやめましょう。恥ずかしいことになるので。せめて簡易版ランダート反射モデルとでも言っておきましょう。

先程のディフューズの話でもアツたように、初步的なシェーディングの輝度値は

$$I_{out} = K * I_{in} * \cos \theta$$

で求められます。が、色々と面倒なので、Kと I_{in} を 1.0 として扱います。すると

$$I_{out} = \cos \theta$$

シンプルでしょ？

で、内積の話でもありましたか？

$$\text{輝度} = \cos \theta = A \bullet B$$

です。

今回は平行光線を使いますので、決め打ちの光線ベクトルを使用します。あとは法線がわかれればいいんですけど、既にシェーダ側では NORMAL で入ってきてますので、

$$\text{輝度} = \cos \theta = \text{dot}(\text{LIGHT}, \text{NORMAL})$$

前にも話しましたが、`dot` は内積値を返す関数です。

…シンプルすぎるなあ。

でもやってみましょう。

シェーダ側のコードを書き換えます。シェーダ側だよ!!!人の話を聞け!!

まずは頂点シェーダ側の関数を書き換えます。

現在の頂点シェーダは、頂点の座標と、UV 値のみを受け取る状態になっていますが、そこに NORMAL つまり法線情報を挿入します。

引数が

`float4 pos:POSITION, float4 norm:NORMAL, float2 uv:TEXCOORD`

になるわけですね。

で、モウヒトツです。この NORMAL 値をピクセルシェーダ側まで持って行きたいので、それ用の構造体にも NORMAL を追加しておきます。

```
float4 pos:SV_POSITION;
```

```
float4 norm:NORMAL;
```

```
float2 uv:TEXCOORD;
```

はい、これで準備完了です。

まずはピクセルシェーダ側に渡すために、

```
v.norm=norm;
```

と言った具合に、法線情報をそのまま渡してあげましょう。

次にピクセルシェーダ側です。

まずはライト定義しておきます。ホントはこいつも外側から与えてあげるものなんでしょうけど、今はテスト的なものだし面倒なんでピクセルシェーダ側に

```
float4 light = float4(-1,1,-1,0);
```

とでもしてあげます。このときのライトベクトルは、**対称点から光源に向かうベクトル**であることに注意して下さい。ですから上のほう向いてるんです。

さていいよ内積です。

```
dot(light,v.norm)
```

で内積が出来ます。ここで注意点は、あくまでも**内積値はスカラー値**であるため、戻り値として使う時はベクトルにする必要があるということです。そこは自分で考えてみましょう。

どうですか？ちょっと立体っぽくなつたでしょうか？

でも、回転させてみればどうでしょうか？なんか影まで回転してませんか？

そうです。ダメなんです。

当たり前の話ですが、モデルが回転すれば法線ベクトルも回転します。当然のことですね。これが足りてないのです。

それでもまずは立体感ぽいものが出来ることを体感してください。

法線ベクトルも回転させよう

はい、法線ベクトルも回転させたいと思います。しかしそのためにはワールド行列の回転成分を法線ベクトルに乗算してやる必要があります。

ところがここで面倒なことになりますね。なにかつちゅーと、既に計算済み行列をシェーダに渡せるようになってるんだけど、そこにワールド行列を追加したかったらどうするんだろう？

もし、計算済み行列の後に追加するなら、連續したメモリにしておく必要があります。連續したメモリに配置するには2つの方法があります。

構造体にまとめるか、配列にするかです。

今回は構造体にでもまとめておきましょう。

以下からC++言語側のコードです。

```
struct MATRIXIES{
```

```
    XMATRIX matrix; // 色々計算済み行列
```

```
    XMATRIX world; // ワールド変換行列
```

};

当たり前だけど、行列二個ぶんになったんで、バッファのサイズは2倍です。で、それぞれの値を設定…

```
MATRIXES m;
```

```
m.matrix=world*view*proj;
```

```
m.world=world;
```

とでもしてやれば、そのままコンスタントバッファーとしてシェーダ側に流れますので、あとはシェーダ側の

```
cbuffer global{
```

```
    matrix mat;//合成済み行列
```

```
    //ここにworld行列を追加
```

```
}
```

合成済み行列の下にいま追加したワールド行列を追加しましょう。

どうです？ある程度それっぽくなりましたが？でもまだ何かおかしいですよね？うん、実はこれには海よりも深あへいワケがあるのだ。

Zバッファ法(深度バッファ)

あー、はいはい、そうね。

シェーディングまでできても、なんかおかしいある。

どういうことか。

それは、Zバッファとかやってないからよ。

Zバッファ…これは基本的なものよ。まあ、その前にどうして今のような状況が起こっているか説明するある。

基本的には、描画の順番というのは、登録した三角形の順番に描画されるある。

で、カリングされていない三角形というのは、問答無用で描画される。

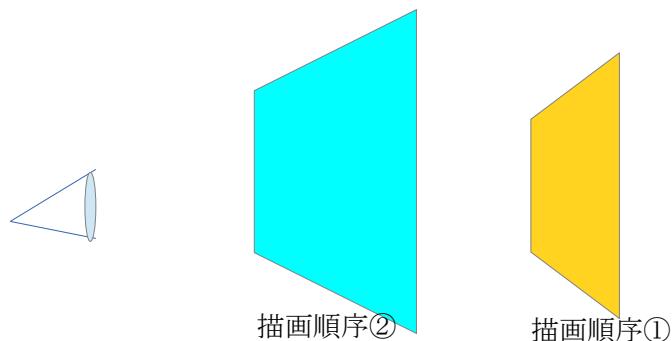
これが単純で少ないポリゴンモデル（たとえば立方体など）の場合は問題ないけど、複雑なモデルの場合、結構大変。

どういう時に問題になるか。それは、奥にあるポリゴンが、後のほうに描画されるとき。

さっきも言ったように、とにかく登録した順番に描画されるわけよ。

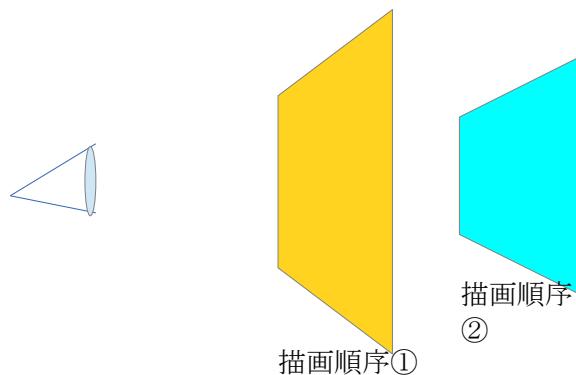
そうすると何が起こるかというと、

奥にあるポリゴンで、手前のポリゴンが隠されてしまうのだ!!!



こういう状態なら、描画順序①のものが先に描画され、上から手前のポリゴン（描画順序②）によって塗りつぶされるため、「奥のポリゴンが手前のポリゴンによって隠れて、一部もしくは全体が見えなくなる」が、これはアタリマエのことなので問題ない。

ところが、この順序が、手前の物が先に描画されたとすると気持ち悪いことになる。



これは気持ち悪いよー。

普通は手前の物体に隠れて見えないものが、見えててしまうのだから!!!

内臓まで見えててしまうよー!!怖いねー!!!

ここで考え出されたのがゾノダッファ法という概念ね。

最終的にラスタライザが、描画するピクセルごとにいろいろなものをピクセルシェーダに送り込むんだけど、

この時に「カメラからの距離」を、float型でピクセルごとに登録しておくんだ。

…ピクセル毎に登録ってことはどういうことかな？

そうだね…絵を描画するのに似ているね。結局のところ絵を描画するってことはどういうことでしょうか？

うん、そう、RGBAの値をそれぞれのピクセルに登録してるんだ。

ということは逆に言えば、カメラから見た Z 値を、一枚の絵を書くように登録していいことになる。

で、 Z 値が小さければ小さいほどカメラに近いわけだから、 Z 値を登録しておいて、同じ場所に別の Z 値を登録しようとした時に、

- 新しい Z 値の方が小さければ描画＆新しい値を登録

- 新しい Z 値の方が大きい場合はナニモシナイ

これだけで奥の物体が描画される心配はなくなります。

で、 Z 値、 Z 値と、単純な Z バッファとして使っていた昔の名残で読んでますけど、 Z って言うと誤解を招くので、Direct3D10くらいから Z バッファって言わなくなっていますね。

深度バッファって言うようになります。

Direct3D9の時代でも「 Z バッファまたは W バッファと呼ばれる深度バッファ」とか書いてますね。

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

それはともかく、DX11になると Z バッファってあんまし言わなくなっていますので、注意しましょう。

深度バッファ（深度レンダーターゲットビュー）の作成

さて、深度バッファを作成していくわけですが、さっきも書いたとおり、深度値登録ってのはある意味絵を書く作業に相当するって話をしました。

というわけで、DX11では、それ用のレンダーターゲットビュー（深度値書き込み先）を用意してやつて、そこに深度値を書き込んでやればオッケーってことです。

…DX9に比べると面倒になったもんだ。

愚痴はともかく作っていこう。

レンダーターゲットビューを作った時と同じようにやっていきます。まずは書き込む紙が必要だよね。ということで、画面のサイズと同じ大きさの紙を用意します。

今回は深度値(32ビット float)のみが必要なので、グレースケール用の紙でOKですね。レンダーターゲットビューを作った下あたりからコードを書いていけばやりやすいでしょう。

まずどうするんでしたつけ？紙を用意する CreateTexture2Dですね。

[http://msdn.microsoft.com/ja-jp/library/ee419804\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419804(v=vs.85).aspx)

まあほとんどかつてのレンダーターゲットビューと同じでいいです。違う部分は、32bit 浮動小数点のみでいいので、

D3D11_TEXTURE2D_DESC の Format が DXGI_FORMAT_D32_FLOAT…になります。R32でもいいんじゃない？とか思ったんだけど、Rだと深度テストがうまいこと働かないでござる。

どうもDにしどかないと深度テストしないようです。リファレンスにははっきり書いてないのによく分かってません。ごめん勉強不足です。

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

不親切ってレベルじゃねーぞ!!外人さんも困ってはるやん!!

<http://social.msdn.microsoft.com/Forums/ja-JP/716dfc08-183f-4133-b72e-7bb432d4e8b2/differences-between-dxgiformatd32float-and-dxgiformatr32float?forum=wpappszh>

どないなってんねん!!!質問に答えんかレ!!!

<http://developer.download.nvidia.com/presentations/2008/GDC/GDC08-D3DDay-Performance.pdf>

ここを見てもはつきりしたこと分からんし…なあ。

まあ、きちんとしたリソースわかつたら追記しとくから、それまで待っておいてくれセリフ
ンティウス!!!

さて、D32にしたら、あとはBindFlagsです。こいつはD3D11_BIND_DEPTH_STENCILですね。あ
とはレンダーターゲットと似たようなもんです。

ある程度かけたらCreateTexture2Dでテクスチャ作っちゃいます。

2番めの引数は初期化データですので、今回は初期化データが必要ない(IMMUTABLEではな
い)ためNULLにでもしておきます。

で、そのテクスチャができたら、深度Stencilビューを作成し、最終的にレンダーターゲッ
トにセットします。

はい、いよいよ次に深度Stencilビューを作っちゃいます。

CreateDepthStencilViewですね。

[http://msdn.microsoft.com/ja-jp/library/ee419790\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419790(v=vs.85).aspx)

で見たら大体把握できると思いますが、第二引数はNULLにしていいです。

第一引数は、さっき作ったテクスチャです。

第三引数はID3DDepthStencilViewへのポインタへのポインタです。

いつものやつなので、もうそろそろ慣れてください。お願いします。で、それができたら
OMSetRenderTargetsでこの深度Stencilとレンダーターゲットを紐付けて登録します。

[http://msdn.microsoft.com/ja-jp/library/ee419706\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419706(v=vs.85).aspx)

今回はバインドするレンダーターゲットは深度のみなので1でいいです。

つぎのがレンダーターゲットビューへのポインタへのポインタ。

最後がさっきつくった深度Stencilビューを置きます。

深度のクリア

あ、クリアしないと、たぶんにも映らなくなりますのでね。

ClearDepthStencilViewを呼び出して下さい。

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

第一引数は既に作っている深度Stencilビュー

第二引数はクリアフラグだけど、

[http://msdn.microsoft.com/ja-jp/library/ee416060\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416060(v=vs.85).aspx)

今回は深度のみを使ってるので,D3D11_CLEAR_DEPTHでいい。

第三引数はこれは1.0でいいです。0.0だと、それより近いものがないので、なにも描画されなくなります。1.0にしてください。

第四引数はとりあえず0でいいです。Stencilはまだクリアしないんで。

さつこれで、可愛くなつたろう？



白黒だ…色…色をつけようか。

マテリアルを反映

なんで白黒なのかというと、何も色をつけていないからです。で、どこに色が隠されているのかというと、マテリアル(材質)というところに隠されています。別に隠しているわけではないんですけどね。

PMDの場合それは、インデックスデータの直後に書かれてます。

マテリアルデータを見てみよう

15 face_vert_index[59720]	2FFC 2FF9 2FFC 2FFB
1D material_count	0000001A
61 material[0].diffuse_color[0]	3F33D07D 3DCCCCCD 3DFAACDA
6D material[0].alpha	3F7FBE77
61 material[0].specularity	40A00000
65 material[0].specular_color[0]	00000000 00000000 00000000
71 material[0].mirror_color[0]	3EE0C49C 3D800000 3D9CAC08
7D material[0].toon_index	00
7E material[0].edge_flag	00
7F material[0].face_vert_count	00000384
83 material[0].texture_file_name[0]	00 FD FD FD FD FD FD FD
83 material[0].texture_file_name[16]	FD FD FD FD
87 material[1].diffuse_color[0]	3DAAE1BA 3DAAE1BA 3DAAE1BA
93 material[1].alpha	3F800000
97 material[1].specularity	4299999A
A3 material[1].specular_color[0]	00000000 00000000 00000000
B7 material[1].mirror_color[0]	00000000 00000000 00000000
C3 material[1].toon_index	00

見て何となく分かるようになってきたと思いますが、インデックスデータの直後にあるのが、マテリアル数です。この例だと1Aと書かれます。10進数だと26なので、26種類のマテリアルが存在することになります。あ、見て分かるようにマテリアル数は4バイトのunsigned intです。

マテリアルフォーマット

ちなみに、ひとつひとつのマテリアルのフォーマットは

ディフューズカラー RGBA : $4 \times 4 = 16$ バイト

スペキュラ強度 : $4 \times 1 = 4$ バイト (累計 20 バイト)

スペキュラカラー : $4 \times 3 = 12$ バイト (累計 32 バイト)

アンビエントカラー : $4 \times 3 = 12$ バイト (累計 44 バイト)

トゥーン番号: 1 バイト (累計 45 バイト)

輪郭フラグ: 1 バイト (累計 46 バイト)

このマテリアルが適用されるインデックス数: 4 バイト (累計 50 バイト)

アライメントに注意！！

テクスチャファイル名: 1*20=20バイト(累計70バイト)

とにかく最初にそれっぽくするためにディフューズ色から反映させていきます。PMDモデルによっては色付けが全てテクスチャのみで行われている事があるため、思ったような感じにならないこともあると思うが、そういう場合はテクスチャマッピングを行えるようになるまで楽しみに待っておこう。

余談ですが、テクスチャマッピングなんですが、たまにtgaファイルってのがあるんですけどCreateShaderResourceFromFileってな関数あったじゃない?あれってtgaファイルをサポートしてないんだよねー。このことは後々話しますが、結構ややこしいことになってんのよねー。だから、tgaはとりあえず捨てといたほうがいいと思います。後で時間がアレばtgaをロードする方法も教えますけど、たぶん自分の首がしまるだけですよ。こんなもんロードできた所で、企業へのウリになんかならないから(但しペガサスジャパンだと売りになるかも知れへんけど…)。

マテリアル読み込み

それはさておきマテリアルです。早速読んでみましょうか。まずマテリアル数。これはそんなに多くない(20~30)んですけど、とりあえずunsigned intで定義されてるので、

```
fread(&materialCount,sizeof(unsigned int),1,fp);
```

とかで取得しちゃいましょう。で、例によって例のごとくマテリアルをひとつひとつ取得していくんですけど、構造体は例えば、こんな感じで。

```
//マテリアル読み込み
struct MATERIAL{
    XMFLLOAT4 diffuse;//ディフューズカラー成分
    float specularity;//スペキュラ強さ
    XMFLLOAT3 specular;//スペキュラカラー成分
    XMFLLOAT3 ambient;//環境光成分
    BYTE toneIdx;//トーンマッピングインデックス
    BYTE edgeFlg;//輪郭線フラグ(ここでアライメントに気をつけてね)
    unsigned int indicesNum;//対象インデックス数
    char textureName[20];//テクスチャファイル名
};
```

ここもさっき書いたようにアライメントに気をつけながら読んでいきます freadとかで。
アライメントに気をつけて下さい。大事なことなので二回言いました。

マテリアル数がわかっているので、for文なりなんなりで繰り返しながら読んでいきます。ここはもう三回目のお決まりのパターンなんでそろそろ自分でできるようになってくださいね。

さて、これで材質情報をロードできたわけなんですが、こいつはロードよりも活用のほうが面倒なのだ。あ、ロードがわからんて人は周囲の分かってそうな人に聞いて下さい。

分からん人に分かるように教えるのも勉強になるものです。あ、かつての授業でやつてゐるはずの所を丸っきり忘れているような不届き者に対してはキレついでです。今までどれだけユトってきたのか思い知らせましょう。

マテリアル適用(ディフューズのみ)

あ、やる前に言っておくと、MMDのいくつかのモデルの場合、テクスチャ有りきでモデルが作られているため、今からやるディフューズをやったとしても、見た目変わらないことがあります。

例えばミクならこんな感じになるんですけど



アリスだとこうなっちゃいます。



残念でならないッ！なので、ディフューズ反映の部分はちゃつちゃと済ませちゃって、はよテクスチャを貼っていきませう。

はい、マテリアルの反映です。既にマテリアル情報は読み込んでいますが、これをどうやって反映させるのでしょうか？うん、まあ、頂点データに混ぜでもいい。別にそれはありなんだが、よく考えてみてくれ、MMDの材質のデータは頂点にくっついしているのだろうか…？

データの上でもマテリアルデータとして

090E7F material(0).face_vert_count 00000384

とか書いてある。つまりマテリアルがインデックス情報を持っている。そしてインデックスとは面を構成する。故にマテリアルは面にくっついしているものなのだ。

というかデータ的にはマテリアルに面がくっついていると考えたほうがわかりやすいかもしだれない。

まあ、まずはそれぞれのマテリアルが持っている面の数の合計イコール全ての面の数であることを確認しよう。

マテリアル毎に面を描画

なに、難しいことはない。マテリアルは複数あり、君はマテリアル数を知っている。そしてそれぞれのマテリアルには面数が記載されている。どうする？

ちょっとはさー、自分で考えましょうねー。会社に努めたらセンセーいなーし、いちいちこんな事教えてくんないよー？

こんなのすら自分で考えられない人がプログラマとして入ると、上司とか先輩から疎まれますよ？

はい、一応学校だから教えたげるよ for 文を使うんだ。0番目からマテリアル数まで回すんだ。

で、描画は相変わらず DrawIndexed なんだけど、ここで今一度マニュアルを見直そう。

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

普通の脳味噌持ってたら、第一引数は予想がつくね。うん、そのとおりだ。いいよ。

第三引数はそのままいい。

問題は第二引数だ。

恐らく、第一引数にのみマテリアル数を設定すると、なんとも悲しい表示にならないだろうか？

そう、だからマニュアルの第二引数の部分を読んでみよう。

「インデックス バッファーから GPU で読み取る最初のインデックスの場所です。」

わっかりにくっ！！なんだこれ、「インデックスのオフセット」でいいじゃん。

つまり、既にインデックスバッファには沢山のインデックスが登録されてるんだけど、そのどこから始めるのかってことなんだ。

ということは、例えば最初のマテリアルであれば

context->DrawIndexed(material[i].indicesNum, 0, 0);

でもいいんだが、2つ目からはそろは行かない。

context->DrawIndexed(material[i].indicesNum, インデックスのオフセット, 0);

さあ、ここまで書いたらプログラマを志している諸君ならば“わかるだろう。健闘を祈る。分からぬ人は、例によって周囲の分かってそうな人に聞いて罵倒されて下さい。そのうち気持ちよくなります。…このドMがツツツツ！

とにかくループしつつ、マテリアル毎にDrawIndexedしていこう。

ここが一応正常っぽく表示できたらイヨイヨ色を付けていこう。前にも言いましたが、残念ながら、ここまでやっても色がつかないモデルがおおいで。いちどプレーンなミクで確認する事をお勧めします。

ディフェューズ反映の概要説明

チョット前に話したように、マテリアルは面につきます。で、今マテリアル毎に描画しています。

ということはどうやってディフェューズを反映させるのか、ちょっと予想してみて欲しい。難しいかな。これDirectX9経験者なら結構ピンと来ると思うんですけど…やってないし仕方ないね。もし予想ついた君は賢い、天才だ。もう教えることはないッ！

結論から言うと、マテリアル毎にコンスタントバッファーとしてシェーダに登録します。コンスタントバッファーって忘れてる人も多いと思いますが、行列を渡しているアレです。あれって行列以外も渡せるんですよ。

で、行列とは別に渡すので、スロットは別番号にしちきましょう。どこぞのサンプルでは行列もマテリアルも一緒にたのメモリにして渡していますが、僕はそれはどうかと思います。何故ならマテリアルループでいちいち変換行列まで渡すのはタッセーと思うからです。

で、シェーダ側はどういうコードになるかというと、cbufferを一個増やします。当然ですね。

で、このへんも好みが分かれるトコロですので、各自好きな方法でやってほしいんですが、僕はdiffuseをいったん頂点シェーダの戻り値に渡してあげます。つまり、戻り値構造体にひとつdiffuseって項目を追加してあげます。

このとき

```
float4 diffuse:COLOR;
```

とでもしてやればいいでしょう。要は描画する毎にマテリアルを変更してやることによって、面の色を変えていくわけです。

さて、最後にピクセルシェーダなのですが、現在輝度値を返していると思いますが、こ

の輝度にディフューズカラーを乗算することになります。

なので、手順としては

- ①材質情報を渡す用の構造体を作成
- ②コンスタントバッファーの作成
- ③マテリアルループ時にコンスタントバッファーの内容を変更
- ④バーテックスシェーダ(ピクセルシェーダでもいいよ)に②のバッファをセット
- ⑤シェーダ側のコードを変更

以上です。

材質情報を渡す用の構造体を作成

とりあえずディフューズだけでいいので、

```
struct Material{
```

```
    XMFBLOCK4 diffuse;
```

```
};
```

こんな感じにしつく。現状だと構造体なんぞいらんのですが、後々の拡張を考えてこうします。

コンスタントバッファーの作成

行列で作ったコンスタントバッファー作成の部分をパクって下さい。変更箇所は
ByteWidthくらいしかないですね。あ、こんな阿呆はいないと思いませんが、行列で作った
部分をコピペしてもいいんですけど、matrixBufferのまま作ろうとしないでくださいね。

ByteWidthはモチロン sizeof(Material); とでもしておいてください。

で、CreateBufferあたりで、materialBuffer って名前の材質バッファを作りましょう。

マテリアルループ時にコンスタントバッファーの内容を変更

はい、既にループしつつマテリアルごとに DrawIndex を行っていますので、次は
DrawIndex 前にコンスタントバッファーの内容を更新します。更新には例によって例のご
とく Map～Unmap で変更します。

あらかじめ

```
D3D11_MAPPED_SUBRESOURCE mappedMaterial;
```

てな感じで宣言しておきます。

で、ループ内で

```
context->Map(materialBuffer,0,D3D11_MAP_WRITE_DISCARD,0,&mappedMaterial);
//ここでこのメモリの塊に、マトリックスの値をコピーしてやる。
memcpy(mappedMaterial.pData,(void*)(&material),sizeof(material));
context->Unmap(materialBuffer,0);
```

こんな感じね。ここでも、似てるからって matrixBuffer と間違えちゃダメだよ。

頂点シェーダにマテリアルバッファをセット

頂点シェーダにコンスタントバッファをセットするということは、当然呼ばれる関数は VSSetConstantBuffersですね。

[http://msdn.microsoft.com/ja-jp/library/ee419762\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419762(v=vs.85).aspx)

はい、ここで注意。コンスタントバッファとしては二個目なので、startSlotは1です。それ以外はご想像の通り。バッファ数は1。最後の引数にはマテリアルバッファを放り込みます。

シェーダ側のコード変更

まず、コンスタントバッファが増えていますので、受け取り用の変数を作ります。

```
cbuffer マテリアル用名前{
    float4 _diffuse : COLOR;
};
```

ホントはそろそろ register とか入れなきゃいけないんだけど、トラブルが起きるまでは気にしないことにします。

で、次に構造体に diffuse を追加します。float4 diffuse : COLOR とでも書けばいいです。

で

```
vout.diffuse=_diffuse;
```

とでもしてやれば、ディフューズ成分を渡せます。

最後にピクセルシェーダですが、これは簡単です。既に輝度情報を持っているでしょう？例えば`r`という変数に入れているのならば、

```
return vin.diffuse*b;
```

これで終了。

ここまでできれば、マテリアルを「ある程度」反映した上でモデルを表示できるのではないでしょうか？

モデルにテクスチャを反映させます

これ、結構ややこしいので、部分的にできてれば、まあよしとします。全員が全員できるような代物とはいえません。どうしてもやりたい…でもやる実力がないって人は、プライドにこだわらず周囲の人には聞きましょう。

頑張れば



この程度はできますので、キャラに対する「愛」があるのならば頑張りましょう。挫折すれば、君の愛はそんなものだったということだ。何が愛宕ちゃんだ島風ちゃんだこのやろう。可愛く表示してこそ愛だろツツツツ！！！

で、マテリアル反映モデルの表示の期限を今週末(10/11)とします。

課題：マテリアルを反映させたモデルを10/11(金)までに表示させて下さい

はい、しょーもない事言いましたがやっていきましょう。

テクスチャ情報はマテリアルの最後に文字列として指定されております。最後に

```
char textureName[20];
```

ってあると思いますが、ここに入ります。で、ここに「a.bmp」だの「1.png」だの「g.jpg」とかっていう文字列が入っています。もちろん文字列の最後には¥0が入っておりますので、そいつを元に画像ファイルを探します。

で、ほとんどのpmdファイルがそうだと思うんですが、pmdファイルが有るところにpngだのjpgだの画像ファイルがあると思いますので、あとはこの取得した文字列を

D3DX11CreateShaderResourceViewFromFile

[http://msdn.microsoft.com/ja-jp/library/ee416885\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416885(v=vs.85).aspx)

に放り込んでやればいい。もう前に1回テクスチャロードしてるから思い出してください。ハイ、第一引数にはデバイスですね、device。もうわかるでしょ？

第二引数に読み込んだ文字列を入れてあげます。

面倒なので第三引数はNULLでいいです。ファイルから勝手にAPIが判断してくれます。

第四引数もNULLでいいです。

第五引数にはID3D11ShaderResourceViewのポインタのポインタを放り込みます。

これでテクスチャのデータをロードできますよ。

で、おわかりかとは思いますが、使用されているテクスチャは一つとは限りません。最大の数としてはマテリアルの数と同数あることになります。

ということはどうすべきかな？

複数のテクスチャオブジェクトの準備

そう、マテリアル数ぶんループしつつロードするんですよ。ロード。ということはロードしたものを確保しておくためにID3D11ShaderResourceView*型の配列も必要ですよね、例えばベクターを使用するのであれば

```
std::vector<ID3D11ShaderResourceView*> textures(マテリアル数);
```

とでも宣言しておきループしつつ、こいつを最後の引数に当てていけばいいんです。

テクスチャ名取得の際の注意点

で、このときに注意点があります。

PMDのテクスチャ名の部分が「2.bmp*a.sph」という風になっているものがあるとおりもいます。ほんやらら.sphってのは、「**スフィアマップ**」用の画像です。これはツヤツヤ感とかメタリック感を出すための補助的なテクスチャです。

とりあえずコピペしてみて、画像の拡張子の sph の部分を bmp に変えてみて下さい。そうすると、スフィアマップ用っぽい画像が出てきます。アリスたんなら



こうなっており、アリスたんの美しい髪の毛の艶を演出するためのテクスチャという事がわかりますね。ちなみにメタルミクさんなら



こういうのが設定されています。

まあ、大体どういうものが分かってきたとは思いますが、今回はこれ抜きでまずはそれっぽく表示するのを目標にしましょう。

なので、*a.sphを見つけたらその部分を削除するコードを書きます。

この時に便利なのが std::stringですね。

使いたいなら #include<string> しておきましょう。

```
std::string str=materials[i].textureName;
```

とでもしてやってコピーします。こいつの優れている所は

```
str.find('*');
```

の戻り値が、文字*がある部分を示しますので、先頭から、この手前まで読み取ればいいことになります。文字列の一部を取得するには str.substr 関数を使います。つまり

```
int idx=str.find('*');
```

```
str=s.substr(0,idx);
```

でおおもとのテクスチャ名を取得できます。

最後に、こいつを D3DX11CreateShaderResourceViewFromFile に渡したい場合は C の文字列にする必要がありますが、その場合は C_str() 関数を使えばいいので、str.C_str() を渡してあげます。

余談(スフィアマップについて)

なお、MMD の挙動から察するにスフィアマップはスペキュラに対する補助的な効果のために使われているっぽいです。なので、詳細はスペキュラの実装のあとで話す予定ですが、もし、手っ取り早くその効果が見たい人は

①ピクセルシェーダの法線ベクトルの xy 成分のみをとってきて正規化する

②その時点で円形の -1.0 ~ 1.0 になっているはずなんでそれを 0.0 ~ 1.0 の範囲に補正する。

③そのまま uv として扱い、スフィアテクスチャから画素値を取得

④③の画素値に適当な重み(係数乗算)をつけて加算または乗算する。

で大体の効果はわかると思う。まあこのやり方は非効率かもしれんので各自改善できる人は改善してくれ。

わがんね一人はスペキュラ後の詳細解説を待て。

マテリアルループ内でテクスチャの切り替え

さて、話はそれたけど、これでテクスチャ自体をすべてロードできているはずなので、あとはマテリアルループに組み込むだけだ。

すでにどこかに書いてあるであろう PSSetShaderResources をこのループ内に持ってきます。で、ループ時にループ変数は変わるでしょ？当然。そのループ変数に合わせてテクスチャを切り替えていけば簡単に変わります。

さて、ここでちょっとだけ大事なことを言っておきます。

☆テクスチャがロードできなかった(ファイルがないもしくは“**ファイル名が空白だった**”)場合は、**テクスチャオブジェクトにはNULLが入っているはずです☆**

さて、このことが、後にシェーダをいじる際に重要な話になっているので、後で困りたくない人は頭のなかにメモっておいてください。

シェーダ側の処理

それはさておき、シェーダ側ですが、以前に texture の Sample 関数を使ってテクスチャを反映させましたよね？それを一回復活させてください。

で、テクスチャ反映後のピクセル値をそのまま return してたと思いますが、それに対して、前回計算した輝度値(b?)を乗算してください。

```
return texture.Sample(略)*b;
```

こんな感じ？

…さて、実行してみてどうだろうか？同じソースコードでも、うまくいくモデルとうまくいかないモデルが有ると思います。

テクスチャがあるときないとき

PMD のマテリアルデータにはテクスチャがある場合とない場合があります。以前にも書きましたが、ない場合はテクスチャオブジェクトには NULL が入っています。

こいつが悪さをしゃがります。

どういうことがというと、PSSetShaderResource にて、NULL を設定すると、シェーダ上のテクスチャオブジェクトは、言ってしまえば float4(0,0,0,0) 状態になります。正確に言うと不定ですが、まあ、ろくでもない状態になります。

で、こいつとディフェーズを乗算すると、当然輝度値は 0 になるので、真っ黒ではいけない部分が真っ黒になってしまいます。

これをどうにかする方法としてはいくつありますが。よくやる方法としては

- テクスチャがある時と NULL のときとで、使用するピクセルシェーダを切り替える
- 真っ白なテクスチャを用意し、テクスチャ名がない場合は全て白テクスチャにする
- シェーダ上で if 文で分岐する

のどれかになると思いますが、最後のやつは正直おすすめしません。最初のやつか二番目のやつをお勧めします。

ちなみに、スフィアマップ(メタルミク用)を無理やりアリスに適用するとこうなります。



即興(10分)で適当に作ったコードだとこんな感じです。

```
return rgba*b*sph.Sample(sample,(normalize(v.norm.xy)+1.0)/2)*1.5;
```

まねしなくていいです。MANE SINAKUTE IIDESU.

これは動いているところを見たほうが面白いと思いますので、お見せしようと思っています。バーチャライターのデュアルみたいになります。

とりあえず、ディフェーズに関してはこんなもんです。お疲れ様でした。

トラブルシューティングゲーム

ここ以下に書くことは、ここまで表示の際に妙なトラブルが起きた人用の話なので、それを解決する気がない人は読まなくてもいいです。

特定のポリゴンが特定の向きでは表示されない！

これは恐らく、隠面消去(背面カリング)という仕組みがフルサーをしているのだ。PMDは基本的に背面カリングを行なっていいようだ。

背面カリングとは？

英語で Culling と書き、「選抜除去」と訳されている。**背面カリングとは「後向している面はどうせ描画されないので(△)イテネ」という仕組みにより、余計なポリゴンを処理するコストを省いているのだ。**

ところがMMDの場合「別にゲームじゃないし(ムービー作るソフトだし)コスト削減は別に気にしなくていいんじゃね？」という思想の元に作られている。このため背面カリング等

は行われていない。

たいていのモデルはソフトでカリング考慮されて作られているため問題ない。

しかし、特定のモデル(Lat ミクとかしまかぜ等)ではカリングが考慮されていないようで大変なことになっていたりする。

それでも表示したい。そんなあなたのためにカリングを OFF にする方法をお教えいたしましょう。

背面カリング OFF はどうやんの？

そもそも背面をカリングするってのは…してくれているっていうのは誰がやってんのかって言うと DX11 ではラスタライザというやつがやってくれている。こいつがデフォルトで用意したラスタライザがやってくれている。

カリングを OFF にしたければ、自分で適当なパラメータでラスタライザを生成し、それを DX11 に登録すればいい。

使うのは

CreateRasterizerState

[http://msdn.microsoft.com/ja-jp/library/ee419799\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419799(v=vs.85).aspx)

と

RSSetState

[http://msdn.microsoft.com/ja-jp/library/ee419742\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419742(v=vs.85).aspx)

です。

勘の良い人はもう使い方わかりましたね？

問題はこいつ

D3D11_RASTERIZER_DESC

[http://msdn.microsoft.com/ja-jp/library/ee416262\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416262(v=vs.85).aspx)

ですね。どこに何を入れていくのか。

とにかく今回の肝になるのは2番めの要素。

D3D11_CULL_MODE CullMode;

これ。これがカリングモードなわけね。

[http://msdn.microsoft.com/ja-jp/library/ee416078\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416078(v=vs.85).aspx)

ここ見れば分かるように、カリング切りたければ D3D11_CULL_NONE にする。他は…まあありがたい事に規定値が D3D11_RASTERIZER_DESC の説明の下の方に書いてあるので、それに従って設定すればいいよ。

手順は

- ① D3D11_RASTERIZER_DESC 型の変数を作る
- ② ①のそれぞれにパラメータを設定
- ③ ID3D11RasterizerState のポインタ型の変数を作る
- ④ device->CreateRasterizerState して③のアドレスに値を入れる
- ⑤ ④で受け取ったモノを context->RSSetState する。

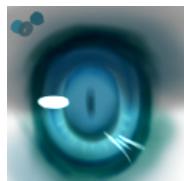
終わり。

透明色への対応ツ…!

本当はスキンメッシュに入る予定だったけど、なんかしまかぜちゃんが上手く表示できない系のバグがあるっぽいので、そっちから先に解決することにしよう。

というか、そもそも何故正しく表示できないのかを考えましょう。島風ちゃんの眼は透明色で出来ています。

どういうことが分からぬい? 例えばアリスたんの眼球なら



このように透明成分が無いのですが、



このように黒目の周囲が透明になっております。結果として、何がおこるのか。



こういう事がおこります。本来の画像と比べてみましょ。



はい、全然違いますね。何故、こういうことが起きているのかをまず考えましょう。

そもそもアルファが「有效」にならないでしょ!!!

面倒なんですが、デフォルトですとアルファが無効になっています。ああ…ごめん俺もハマったわ。ゞバッファのせいだとばかり思っていた…そんな時代が僕にもありました。ゞバッファとアルファ関連の処理をしつかりやってもやってもしまかぜちゃんの目玉が治らない…こういう時はDX9を思い出すんだ…ハツ!!俺…アルファブレンディング有効にしてるか?

し…し…し…してねエーッ!!



仕方ない。これは俺の凡ミスである。もうね…仕方ないのでアルファ有効のためのスクショ見せるです。こいつをDX11初期化の後にでも置いておいてくれ。

```
ID3D11BlendState* blendstate=NULL;
D3D11_BLEND_DESC blenddesc={};
blenddesc.AlphaToCoverageEnable=false;
blenddesc.IndependentBlendEnable=false;
D3D11_RENDER_TARGET_BLEND_DESC& birtdesc=blenddesc.RenderTarget[0];
birtdesc.BlendEnable = true;
birtdesc.SrcBlend = D3D11_BLEND_SRC_ALPHA;
birtdesc.DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
birtdesc.BlendOp = D3D11_BLEND_OP_ADD;
birtdesc.SrcBlendAlpha = D3D11_BLEND_ONE;
birtdesc.DestBlendAlpha = D3D11_BLEND_ZERO;
birtdesc.BlendOpAlpha = D3D11_BLEND_OP_ADD;
birtdesc.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 );
```

すまんが、「しまかぜ問題」は解決してしまったので以下のゞバッファの話はビルボードの話に至ってからやっています。しばらく保留です。

Zバッファ法と透過色の気色悪い関係

以前に深度バッファ(Zバッファ法)を使用して向こう側が描画されないようにしましたよね?今回の現象はこのZバッファ法が悪さをしているのです。悪さというか、結果的にそうなったというか、あちらを立てればこちらが立たず的なところです。

Zバッファ法をおさらいしましょう。

Zバッファ法は、各ピクセルにZ値を保存しておき、今から書き込もうとするピクセルのZ値と比較して、小さいならば塗りつぶし&Z値保存。そうでなければ「ナニモシナイ」でした。

これが良くなかった…何が起きているのか?

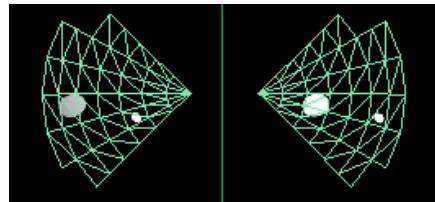
Zバッファ法ってのは、不透明度が1.0未満(半透明)のものにも適用される。これによって何が起きるのだろうか…?

そう、半透明のモノが手前にあり、かつ描画順として半透明が先であるのならば、半透明の向こうのポリゴンが表示されない。

半透明というのならば、向こうにある画像のピクセル値と合成しなければならないにも関わらず…だ。

これにより何が起きているのか?たとえば島風なら正面から見た場合瞳孔の部分が最も手前に表示される…と思ったが、もっと言うと眼の光がまた悪さしてるな。

眼の光は



このようになっており、殆どが透明部分である。ということは、この向こうにあるものは表示されない。つまり真っ黒…もしくは背景色になってしまふのである。

さてこいつを解決するためにもいくつかの方法があります。

①そもそも半透明をなくす…元々のテクスチャから半透明を排除する。今回のしまかぜくらいであれば半透明部分を白で塗りつぶしても問題ない。根本的な解決にはならんが、簡単だし誰にでもできるしコストは安いし…つまりある意味正解かも知れん。

②半透明と不透明を分けといて、半透明だけ後で描画する…これ、マテリアル全体が半透明なら使えるんだけどテクスチャの一部が半透明な場合は、テクスチャのピクセル値をイチイチ見ていかにやらん…使えないわけではないんだけど、今回の例では使わない。

さあ、そろそろ…恐怖の…恐怖のキンメツシュ!!!キンメツシュ地獄ッ!!!ここに入れば、今までが如何に、如何にゆとっていたのかを実感するであろうッ!!!

地獄のスキンメッシュ

スキンメッシュアニメーションってのは要はMMDのようにモデルの関節を曲げたりすることです。で、スキン(肌)になんの関連があるのかというと、フツーに関節を曲げると、肘とかぶつ壊れてしまうからなんです。

曲げてしまった関節を柔らかい皮膚(スキン)で覆うような感じで頂点を配置するため「スキニング・スキンメッシュアニメーション」と呼ばれます。

基礎知識

スキンメッシュを実装するには、それなりの基礎知識がないと結構厳しいと思います。インターネットからソースコードをコピーして持ってくればスキンメッシュとかはイケるかもしれません、意味分かってないと応用が効きません。

とはいってもさっさと動かしたい人のためのHPを紹介しましょう。

『ゲームつくろー』

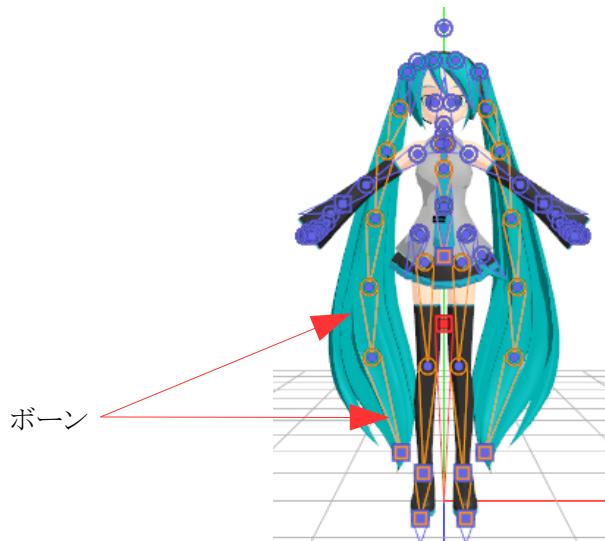
http://marupeke296.com/DXG_Nob1_WhiteBoxSkinMeshAnimation.html

予習する気があるのならばここ『完全ホワイトボックスなスキンメッシュアニメーション』をまず読んでおきましょう。

基本用語

ボーン

ボーンってのはbone(骨)の意味です。まあホントに「骨」っていうわけではございませんが、骨のような役割を果たすものということですね。



図のように、骨っぽいの部分に長い三角のが入っているでしょう？こいつが『ボーン』です。ただし、ボーンにおいて重要なのはこの『骨』ではなくて『関節』のほうなんですよね。だから、青い丸がありますよね？こいつが重要なのです。

この関節(青丸)を回転させることによってボーンの向きが変わります。この向きに合わせて、各頂点の座標を変換することにより関節を曲げたり伸ばしたり捻ったりすることができます。

ただし、関節の回転行列というやつはそれ単体で動作するわけではなく、例えば手首の角度は、腰のひねりから肩の角度、肘の角度が影響しているわけなので、いちいちたどって再帰しながら計算する必要があります。

スキニング

上で話したボーン処理だけだと、各頂点がめり込んだり乖離したりするため、各頂点に対してボーン影響度を適切に設定することによって、曲げた関節を皮膚が覆うような感じに見せることができます。

このため、スキニングを行うには、影響度をそれぞれの影響度を k_n 、それぞれの行列を M_n とすると、 $\sum k_n M_n$ 手な感じになります。簡単すぎますがあくまで簡略数式です。



専門的なことはともかく、自然に手足を曲げるために必要な技術だとでも思っておいて下さい。MMDはスキンメッシュの技術で作られているため、ねつ？自然でしょ？

FK(フォワードキネマティクス)とIK(インバースキネマティクス)

フォワードキネマティクスとは「順運動学」の日本語訳で、体の中心から回転させて特定のポーズを取らせるための手順です。

MMD 技能者たちは MMD を立ち上げてみて、足とかつま先に合わせると、IK の on/off が選べますので、off にしてみてください。これで IK が無効になりますので、そのままセンターを持って下におろしてみて下さい。

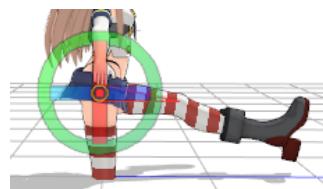
MMD やってる人ならわかると思いますが、IK が on の状態(デフォルト)であれば図のようにきちんとしゃがみますが、



IK を off にしてしまうと足が地面にめり込んでしまい



のように哀れむ然、図のように情けない状態になってしまいます。これを IK off でなんとかするには、順運動学の考え方でなんとかするしかありません。順運動学は根本からの回転によってのみ現在の状況を作り出すものです。ですから、このまま蹲踞ポーズをとらせたければ、まず太もも(*'Д')アリヤを前に回転させます。



次に膝を太ももの回転方向と反対側に回転させます。



はい、なんとかしゃがんでいるようなポーズになってきました。実際はここからまた足首を曲げて~なんていうふうにして、自然な形に調整する必要があります。このように、体の中心側から末端に向かって回転を行うことにより、ポーズを作るのが「フォワードキネマティクス」です。

面倒ですが、実装は比較的簡単なものです。基本的な数学と

次にこの面倒な作業をなんとかしたいとして考えだされたものが「インバースキネマティクス」です。AIKE(IK)って呼ばれることも多く、順運動学に対して逆運動学を表します。

こいつがかなり強力で、要は「末端の座標を決めれば途中の関節の回転が決まる」ってやつです。便利です。こいつのお陰でMMDのモデルは「センター」を下に移動するだけで



図のようにしゃがみポーズをとってくれます。

頭の悪い人は「いやおまえ末端ゆーたら、センター移動させたら話がちやうやろ?」

と思うかもしれません、センターは中心なんです。英語勉強しまちようね? 中心から見たら足首は膝に近づいているわけやん?

そこに合わせるように太ももが回転し、膝が回転してんねん。わかる?

5+□ニ7

とかって書いてあたら、□の部分の数字は2ってわかりますよね? そういう感じです。え? この喻えの計算自体わからない! ? …すまんないわ。ゆっくりニートしていくってね!!

という風にIKってのはかなり強力なんだけれど、実装がそれなりに面倒なのだ。で、IKはホント色々種類があって、

ウォータニオンIKだとか、ヤコビIKだとか、パーティクルIKだとか、色々あんねんけど、実装が比較的ラクなのはCCD-IK(Cyclic Coordinate Descent-IK)であるため、MMDではCCD-IKが利用されているっぽいです。

直訳すると巡回座標下降逆運動学ってわけがわからない言葉になりますが、まあ要は何回か反復計算することで近似解を求めるやり方です。

詳しく知りたい人は

<http://www.tmps.org/index.php?CCD-IK%20and%20Particle-IK>

とか

<http://d.hatena.ne.jp/edvakf/20111102/1320268602>

とか、2Dからイメージしてみたい人は

<http://gamehell.g.hatena.ne.jp/kenmo/20080123/1201102997>

とかを見るよろし!!

クオオオ…タニオン!!

失礼、「クオータニオン」もしくは「四元数(しげんすう)」というものです。こいつは現代のCGには無くてはならないものとなっております。

概念的には簡単であります。クオータニオン自体は虚数の次元を拡張したものであります。

<http://ja.wikipedia.org/wiki/%E5%9B%9B%E5%85%83%E6%95%B0>

そもそも虚数って知っています？中二病ラノベにありそうな数ですねー。ですが、実用的なものです。色んな所で（電気工学とかフーリエ変換とか）使用されて、我々の生活を支えているのであります。時間もないし、虚数すら聞いたことがない人間を相手にしてらんないんだけど、俺様は底辺に優しいので、奈落の底の人たちにも特別にサラッと教えてやろう。蜘蛛の糸だ。

虚数ってのは、二乗するとマイナスになる数のことです。数学では通常は虚数単位*i*を使って表します。*i*は二乗すると-1になる数のことです。本来 Imaginary Number の意味で*i*が使われているっぽいです。

ほらほら、 $\sqrt{2}$ は2乗したら2になるでしょ？で、フツーはさ、マイナスを二乗するとさ-1*-1=1だとか、-3*-3=9だとかで、必ずプラスになるし、必ずプラスになる性質が色んな所で利用されてもいる。

が、ここでとある中二病イタリア人が

「ちょwwwこれ、ルートの中にマイナス入れたら面白くね？」とか言い出したのだ。

$\sqrt{-1}$ 「俺天才www」

と得意げになっていたのだが、

「数学の基礎も分かってない中二乙www」

「m9(^Д^)フギヤー」

「イタリア人め…この程度か。それもまた良し」

「天からお塩!!」

とかさんざん言われてしまう。

が、一応イタリア人も悔しいので、その妥当性を証明するのよ。そこでデカルトが

「お前がそう思うんなら、そうなんだろう。お前の中ではな。」

「こんな数の仲間に入れるのは恥ずかしいから Imaginary Number といいなさい。」

てな感じで定義された。この Imaginary が日本語で「虚数」って翻訳されているだけの話。

じゃあ例えばこの表記(虚数を i とする)にしたがって $\sqrt{-5}$ を表記すると、このマイナス部分だけが外に出て

$$\sqrt{5}i$$

という表記になる。さらにこの虚数と実数(フツーの数)を組み合わせたものを「複素数」という。アンサイクロペディアによると「複雑で素敵な数」らしいですが、アンサイクロなん…ねえ。フツーに考えると、実数と虚数というふたつの「素」で成り立っているので複素数言うんちゃうんかと…。

複素数の一般式

$$z=a+bi$$

こんな感じです。教科書はなぜか複素数に z を使うことが多いですね。で、更にトンデモナイ奴が現れた。

奴の名はウイリアム・ローワン・ハミルトン。そもそも複素数をきっちり定義したのがこの人なんだけど、この人は更に重度の数学病患者だったのだ。

「(^ω^) おっ、複素数すげえ! これは世界を変える WWW!」

「世界は三次元だし、この複素数とやらも3次元に拡張できるんじゃね? 僕天才 WWW!」

とか考えて、生涯の殆どをこの研究に捧げていた。…がうまくいかなかつた。

ある日、

「キタ━(￣▽￣)━! 次元をさらに1個増やして4つにしたらうまくいくよ!!」

と、そこら辺にあった橋に公理を刻みつけた。それがこれだ。

虚数の単位を i, j, k の三種類とする。そして、それぞれの関係が

$$i^2 = j^2 = k^2 = ijk = -1$$

となる数を定義した。これが S I G E N N 数である。ちなみに

$$ij=k, ji=-k$$

$$jk=i, kj=-i$$

$$ki=j, ik=-j$$

という順繰りに成り立つ公理を持っている。

見て分かるように、この四元数とやらも行列と同様に乗算は非可換である。つまり乗算の左右を入れ替えると別物になるのである。注意が必要である。

ガウス「(´_ゝ｀)フーン、ま、俺はとっくにそんなの考えついてたけどね」

ともかく、こいつが後々出てくることは今のうちに言っておくことにしよう。

ツリー構造

よく、計算機科学上では木構造と呼ばれるものである。PMDのボーンのような構造もこの木構造のようになっている。

どういうのが木構造と言うのかというと、大雑把に言うと、誰かが誰かに対するリンクを持っている。このリンクが階層構造になつていければ「木」と思ってくれていいい。

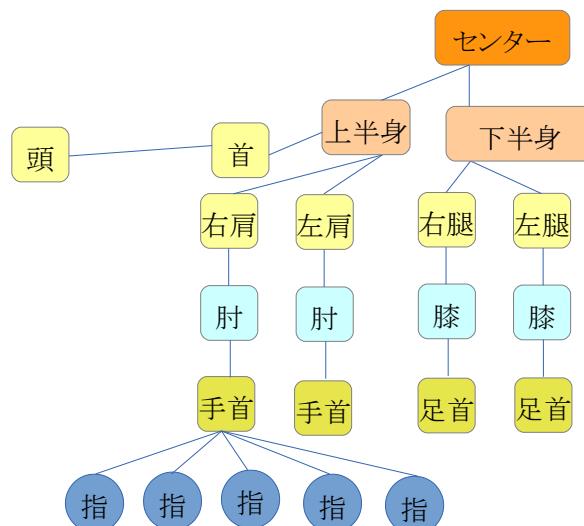
これは単なるリストのように、自分の次の場所がどこ?って決まっているものではないところが特徴である。

ややこしいのだがこういるのは僕らの体(もちろんモデルの体も)を表現するのに都合がいい。例えばヘソあたり(丹田とやら)から始まるとする。背骨はそうすると、上半身と下半身というように大きな分かれ方をする。

小さい部分で言うと、手首からは5本の枝がわかっている。そう、指だ。

PMDの場合も同じような感じで、それぞれが階層構造くなっている。例えば、ミクの指部分のボーンを見ると、全ての指のparent(親)が手首のインデックスを指していることがわかる。

ちなみに肩は上半身ボーンを親としている。



と、まあ、大雑把に言うとこういった木構造くなっているのである。データ的には親が子を管理するのは大変なので、子が親を知っているという状態である。

単純木構造の場合は複数の親を持つことはないため、子：親ニタ：1といった状況なのである。

ちなみにPMDではセンターがみんなのお母さん(鳳翔)ポジションなのである。それはともかく、おおまかに木構造というやつが分かっていただいたのではないだろうか。

メインになる技術

メインになるアルゴリズムとしては、重み付け行列の合成およびツリー構造、再帰等です。あとはまあ、適当にチョイチョイ必要な技術をお話していくことになるでヤンス。

勿論、データ読み込みの技術もあるし、アライメントに注意する必要があるし、C++側とシェーダ側のやりとりもまた増えますし、色々あるんですけどね。

まあまずはボーン情報を読み取ってドーウーふーに配置されているのかを見て行きましょう。

ボーン情報の読み取り

はい、例によって例のごとくデータがどういう風になっているのかを確認しましょう。

270 material[0].texture_file_name[0]	00 00 00 00 00 00 00 00
28F material[16].texture_file_name[16]	00 00 00 00
293 bone_count	007A
295 bone[0].bone_name[0]	83 5A 83 93 83 5E 81 5B
2A5 bone[0].bone_name[16]	FD FD FD FD
2A9 bone[0].parent_bone_index	FFFF
2AB bone[0].tail_pos_bone_index	006E
2AD bone[0].bone_type	01
2AE bone[0].ik_parent_bone_index	0000
2B0 bone[0].bone_head_pos[0]	00000000 41000000 00000
2BC bone[1].bone_name[0]	8F E3 94 BC 90 67 00 FD
2CC bone[1].bone_name[16]	FD FD FD FD
2D0 bone[1].parent_bone_index	0000
2D2 bone[1].tail_pos_bone_index	0002
2D4 bone[1].bone_type	00

ボーン情報めは最後のマテリアルのケツの部分にございまして、へえ。

最初の2バイトがボーン数である。…つづ一かマテリアル数が4バイトでボーン数が2バイトって絶対おかしいだろ!!ホンマに練習で作ったんやな…MMDって。

はい、例によって例のごとく

ボーン名 20 文字 : $1 * 20 = 20$ バイト

親ボーン番号 : 2 バイト

先っちょインデックス : 2 バイト

ボーン種別 : 1 バイト

IK 親ボーンインデックス(なにこれ?) : 2 バイト

ボーン座標 : 4 バイト * 3 = 12 バイト

だからなんでここに
1バイト狭むのよッ!!

ということで合計 $20 + 2 + 2 + 1 + 2 + 12 = 39$ バイト…お前マジでぬめてんのか。なのにその1バイト!つざつけんじやねーよ!!そんなもんケチったって面倒になるだけだつーの!!

…ちがうな、逆に考えるんだ。樋口氏は僕らにおべんきようさせようとあえてこのようなフォーマットにしたのだろう…そうに違いないや。

「なに？読み込みでずれる？JOJOそれはムリヤリ読み込もうとするからだよ。

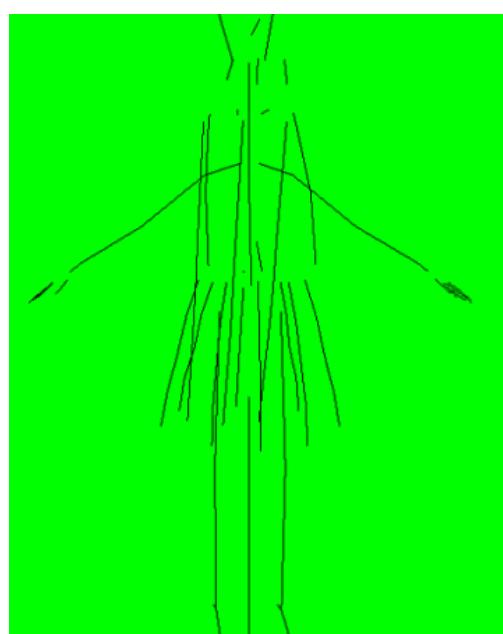
逆に考えるんだ。「読み込まなくてもいいさ」ってね。考えるんだ。」

んなこたーないのでアライメントに泣きながら読み込みます。

今回ちょっとやってほしいことはまず、ボーンの始点と終点の座標を見つけ出し、そいつらをラインリストあたりでつなげて表示していただきたい。

手順は

- ① HeadPos(ボーンの座標)と TailIdx を取得
- ② TailIdx から TailPos(TailIdx の番号に対応するボーンの座標)を取得
- ③ HeadPos, TailPos を頂点配列(頂点ベクタ)に溜めていく
- ④ Head と Tail がこの時点でワンペアになるようにしておく
- ⑤ ④の頂点配列から頂点バッファを生成
- ⑥ ⑤の頂点バッファをグラボに送れる状態に IASetVertexBuffer でセットする
- ⑦ トポロジーをラインリストにしておく
- ⑧ 真っ黒表示する(背景が真っ黒なら真っ白表示にする)シェーダを作っとく
- ⑨ ボーン表示にしたいときに⑦のシェーダを使うようにする。
- ⑩ ラインリストのインデックスは作ってないので Draw で描画する



ラインリストにするには、まずトポロジーを

```
IASETPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_LINELIST);
```

にします。それだけでライヤーフレームになります。

で、そもそも、頂点がポリゴンの頂点ではなく、ボーンのための頂点を定義しますので、新しくCreateBufferで頂点バッファを作成する必要があります。

作って下さい。

で、ラインリストでなくて三角形表示したい場合なんんですけど、三頂点目が問題ですよね？

これは**外積を使う**のです。基本的に人型ボーンってゆーのは \vec{z} 方向の厚みはあまりありません。せいぜい衣服(スカート等)のボーンが放射状になってるくらいです。

ですから、真正面から見ると仮定すると、ボーン方向に垂直(\vec{z} ではない)な方向に三頂点目を配置すればうまくいくことが想定されます。

ちなみにXMVector3Crossあたりを使えば、面倒な計算を関数がやってくれます。

[http://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3cross\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.geometric.xmvector3cross(v=vs.85).aspx)

やってくれますが面倒なことがひとつ。こいつはXMVECTORなので値のセットが面倒なのだ。

初期化の段階なら

```
XMVECTOR tekitou={0.1,0.1,0.1};
```

とかでいいのですが、初期化でない時に値をセットしたいならば

```
tekitou = XMLoadFloat3(&(XMFLOAT3(1,1,1)));
```

のようにXMLoadFloat3を使用します。

[http://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.loading.xmlloadfloat3\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.loading.xmlloadfloat3(v=vs.85).aspx)

ENTERキーだからなんだから表示が切り替わるように(モデル表示 or ボーン表示)。もし提出する際は何キーで切り替わるかreadmeあたりに明記するようにしてください。

次からの課題には不正コピーを防ぐためにタイトルバーにあなたの名前を書いておいて下さい。

提出期限：10/18(金)

要件：ボーンを、ラインもしくは三角形その他で表現して下さい。

モデルを動かすこと

まずはおさらいとして、モデルを簡単に動かすことからやってみましょう。

オフセット回転

原点からオフセットした状態(今まででやっているような原点回転ではなく、ちょっとずらした状態)で回転してみましょう。

例えば $x=2.0, z=2.0$ で Y 軸回転して下さい。

それでませんか? そう。中心がずれるのです。以前に…今となってはすーっと以前にやったと思いますが、オフセットがかかった状態で回転をかけると、回転行列をかけた場合の中心は **問答無用で原点を中心になる**ため中心からそれてしまします。

はいここで、どうすればいいか考えて下さい。

原点を中心に回転させればずれないのです。

つまり

回転→移動

にすればいいのです。移動には XMMatrixTranslation 関数を使います。

さらに、**「モデル座標系で予めズレている場合」**は、一旦原点に戻してやる必要がありますので

原点に戻す移動→回転→もとの位置に戻すもしくは所定の場所に置く

そう、

変換行列は

平行移動に始まり平行移動に終わるのです。

さて、ここまでできた諸君。退屈だろう?

んじゃあ

ワイヤーフレームの左肘から先を(Z 軸中心で上向きになるように)曲げてくれたまえ… 曲がりますか?

データとしてはたいていのモデルはこうなっているはずですがね?

```

d_pos[0]          3F935076 417B0683 3EA2BA9D
e[0]              8D B6 82 D0 82 B6 00 FD FD FD FD FD FD FD FD 左ひじ ...
e[16]
one_index        0012
_bone_index      0014
e                00
t_bone_index    0000
d_pos[0]          405D4791 41647F41 3EF5C28F
e[0]              8D B6 8E E8 8E F1 00 FD FD FD FD FD FD FD FD 左手首 ...
e[16]
one_index        0013
_bone_index      005D
e                00
t_bone_index    0000
d_pos[0]          40A09408 414F6C27 3E8108A2
e[0]              8D B6 91 B3 00 FD FD FD FD FD FD FD FD FD 左袖 ...
e[16]
one_index        0013
_bone_index      005C
e                01
t_bone_index    0000
d_pos[0]          40906569 41560B3A 3E7F3B21
e[0]              8D B6 90 65 8E 77 82 50 00 FD FD FD FD FD FD FD 左親指1 ...
e[16]
one_index        0014
_bone_index      0017

```

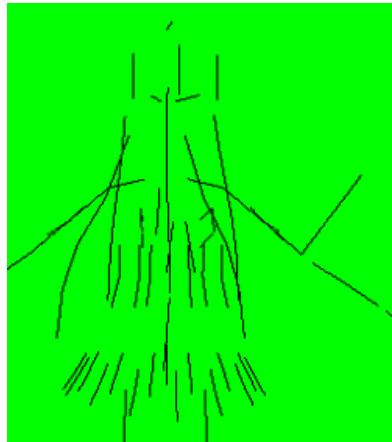
左肘→左手首→左袖ってな感じになっています。まあ、最初は先っちょまで考えなくともいいので、肘を肘支点で回転させてみましょう。

手順としては

- ① HeadPos(ボーンの座標)と TailIdx を取得
- ② TailIdx から TailPos(TailIdx の番号に対応するボーンの座標)を取得
- ③ HeadPos, TailPos ともども、HeadPos が原点に来るよう平行移動する
- ④ 原点に来た時点で回転しておく
- ⑤ もとの位置に戻す。



ちょっとこの例ではモデルとの対応関係を分かりやすくするためにボーンとモデルを同時に表示させてますが、ボーンだけだとこんな感じです。



ボーンが肘部分から曲がってしまっているのが、わかるだろう？

さて、このままでは面白くないね。何しろ針金を曲げただけで、本体にはちっとも変化が無いのだから…これでは(・A・)イケナイ!!

というあなた！

ひとまずこのボーンに影響している頂点をぐりっと曲げてみましょう。そうすると…



ああああああああああああああ！！！折れとる！！絶対折れてるよこれ！！！

うん、そうだね。やっぱり肘から先も曲げてあげないとね、これは。で、ここもモデル構造(ボーンの数とか)が単純でないと効果がわかりにくいので、最初はプレーンなミクから始めたほうがいいと思います。

ちなみに、こんな状態でも曲げてみたいというSなあなたに、折り方をお教えします。
まず曲げたいボーンを選びます。色々やると面倒なので曲げるのは一個だけにしましょう。

```
v.pos = boneInfoes[boneIndex].transform.vPos;
//肘で曲げる処理
if(strcmp(_boneInfoes[i].boneName,"左ひじ")==0){
    XMATRIX mat = XMMatrixTranslation(-vec.x,-vec.y,-vec.z);
    XMVECTOR xv=XMLoadFloat3(&(v.pos));
    xv=XMVector3Transform(xv,mat);
    v.pos.x=XMVectorGetX(xv);
    v.pos.y=XMVectorGetY(xv);
    v.pos.z=XMVectorGetZ(xv);
    elbowMatrix=mat;
    elbowNum=i;
}
```

で、例えばこんな感じで曲げたい部分のマトリクスとインデックスを取得しておきます。

これをコンスタントバッファあたりでシェーダに流し込んであげます。

で、

```
matrix tmp=mat;  
if(boneid&0xffff==elbownum || (boneid&0xffff0000)/0x10000==elbownum){  
    ^ tmp=mul(tmp,elbow);  
}  
v.pos=mul(tmp,pos);
```

こんな感じで、既に頂点情報の中にボーンIDが入ってるんだから、それに対応する奴らに「肘行列」をかけてやる。そうすると、ほら。



まあ、ヒドイことになりますね。

次はこれを何とかしてあげましょう。

ボーンを全部投げる

ボーン情報を全部(ダミー含め512 固定くらい)投げます。

DX9でやってきた人の場合、逆にこの発想が出てこないかもしれない。

とか思ったが、確かに最終的に結局は頂点からボーンが見れない意味ないし、処理量としては妥当なところかなってところだ。

寧ろ考え方としては単純になっているわけで。問題ない。

とはいって、 $512 \times 4 \times 4 \times 4 = 32\text{KB}$ ってのは少ない数じゃないので、**そのうち工夫が必要になるのかもしれない。だけど細々と事は考えず今はそのまま投げてみましょう。**

定数バッファです。DX9の感覚から言うとはっきり言って怖いですね。まともに渡せるんかこれ。

で、そろそろコンスタントバッファの数が煩雑になってきましたので、きちんと番号付けを行ってイミフナバグを防いでおきましょう。

コンスタントバッファに対する番号付け

ここまで来た時点で殆どの人は複数のコンスタントバッファを作っているのではないかと思います。一つにしている人もいると思いますが、まあ聞いて下さい。

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

を見ると

```
BufferType (Name) (: register(b#))
{
    VariableDeclaration (: packoffset(c#.xyzw));
    ...
};
```

こんな感じのことが書いてると思います。わけわからませんねー。

ここで注目すべきは:register(b#)です。リファレンスに#って出てきたら、なんかしらの番
値が入ると思って下さい。このb#が番号付けに関わってきます。シの半音上げじゃないん
ですよ。

:registerってやつは、レジスタなどの部分を使うかってのを指定するオプションです。レ
ジスタってのは、一次データの保存領域で、メモリみたいなものなんだけれど、もうちょっと高速
なのね。

で、そこは番号で領域分けてできるようになってたりするわけ。この番号が、アプリからバッ
ファ指定する際の「スロット」に対応してると思ってればいい。

専門的なことはともかく

レジスタ番号=スロット番号

とでも思っておけばいい。とりあえず今動いている状態でレジスタ番号を設定してみま
しょう。

なお、レジスタ番号は0から始まります。スロット番号と合わせて下さい。例えば

```
VSSetConstantBuffers(0,1,&matrixBuffer);
```

と書いていれば、スロット番号は0番だからこいつをシェーダ側で使うなら

```
cbuffer mainmatrix : register(b0){(chu-ryaku)};
```

とでも書けばいいし、

```
VSSetConstantBuffers(1,1,&materialBuffer);
```

と言う風になつてるのであれば、スロット番号は1であるのでシェーダ側は

```
cbuffer material : register(b1){(chu-ryaku)};
```

と書けばいいと思うよ。

できましたー？じゃあ、ボーン行列を渡す準備の第一段階が整つたわけだ。

じゃあ、ボーン行列を渡して肘を曲げてみましょう。ゼーんぶ渡します。

ボーン行列をゼーんぶ渡す

モデルによっては256で十分なんですけど、アリス大先生のボーン数が262だったので、大事をとて512にしています。

で、シェーダの配列はドウ的確保ってわけに行かないっぽいので、512固定の行列を渡します。
かなりムダですけどね。

手順

- ① XMMATRIX 配列512 宣言
- ② 全てに XMMatrixIdentity()入れとく
- ③ 肘曲げる
- ④ いつもどおりコンスタントバッファ作つとく
- ⑤ コンスタントバッファを GPU 側に流す
- ⑥ シェーダ側で曲げ対処とく

①は簡単ですね。

```
XMMATRIX boneMatrices(512);
```

②は XMMatrixIdentity()を入れる。

XMMatrixIdentity()ってのは**単位行列**のこと

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

ってな感じの行列である。こいつはいくら書けても変化ない数字の1

のような行列なのである。予め全部にこれを入れておけばゴミもなく無害な行列になる。

```
for(うおおおおお!!!){
```

```
    boneMatrices(i)=XMMatrixIdentity();  
}
```

③ 肘曲げる

肘は前々回に曲げているはずなので曲げれると思いますが、曲げきれない人に言っておくと、まず、肘ボーン自体は、テールとヘッドをつなげばベクトルになりますわな？

で、ヘッドをまず原点に置きますわな？回転しますわな？もとに戻しますわな？

```
elbowmat=XMMatrixTransform(-ヘッドのx,-ヘッドのy,-ヘッドのz)*  
XMMatrixRotationZ(90°)*XMMatrixTransform(ヘッドのx,ヘッドのy,ヘッドのz);  
ってかんじですね。
```

④レリーフ加減覚えるコンスタントバッファの作り方

CreateBufferで作ります。

今回は肘曲げるだけなので、subresourceDataで作つとけばいいかな。

…マジで覚えてよ。ここまで来てイチから説明させんって。

```
CreateBuffer(&desc, &subresource, &bonesbuff);
```

ただし、descのByteWidthは
sizeof(XMMATRIX)*512ですね。

で、これでコンスタントバッファbonesbuffができましたと。

⑤コンスタントバッファをGPUに流す

これももう何度もやってるよね。やつてくれ。今回は頂点シェーダだけに必要なので

```
VSSetConstantBuffers(2, 1, &bonesbuff);
```

はい、前も言いましたが、スロット番号を2にするのを忘れないで下さい。

⑥シェーダ側コードに512個の行列を用意

```
cbuffer bones : register(b2){  
    matrix boneMatArray[512];  
};
```

スロット番号とレジスタ番号とが一致するよう気をつけてください。

で、最後に肘ボーン適用なんすけど、ここでチョイと問題があってですなあ。シェーダには16ビット整数(unsigned short)なんて無いんですわー。

誤算つすわー。なのでムリヤリなことするつすわー。

レイアウト部分を見て下さい。

```
{"BONE_ID", 0, DXGI_FORMAT_R16G16_UINT, ...
```

ってあるでしょ？こいつを

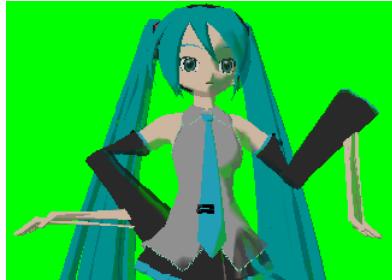
```
{"BONE_ID", 0, DXGI_FORMAT_R32_UINT, ...}
```

にしてください。

うん、そうすると、short 2つ分でひとつの uint 変数になってるわけや。これをドゥ分けるのがと言うと、

まさかのビット演算。できるんやあ…

```
tmp=mul(tmp, boneMatArray(boneid&0xffff));  
tmp=mul(tmp, boneMatArray((boneid&0xffff0000)/0x10000));
```



おお、怖い怖い。

さて、ここまであくまでも準備段階。次にウェイトをかけてやります。

そうだウェートもかけておこう

ここからは見た目あんまし変わらないですが、やつとかないとイケないのやつときます。

そもそもウェイトって何かというと、PMD の場合は各頂点が最大2つのボーンからの影響を受けるわけです。

で、2つあるんですけど、どっちがどのくらい影響を受けるのかっていうのが、各頂点ごとに設定されています。

そして、その影響度は一方が0~100であり、もう片方は100からそれを引いた数になります。ということはどうしたらいいのかというと、

①まず頂点からその情報を受け取る

②足して100になるよう分配する。ひとつめは weight, ふたつめは 100-weight

③ 100が1.0になるようにしなければならないため100でわる。

④ふたつの③を足す

つまりさっきの例でウェイトを100で割ったのを wgt とすると

```
tmp=mul(tmp, boneMatArray(boneid&0xffff)*wgt)+
```

```
mul(tmp, boneMatArray((boneid&0xffff0000)/0x10000)*(1.0-wgt));
```

となる。プラスがポイントやね。

つぎは、肘の曲げを手先まで伝播させます。

ツリー構造を理解する

結局、肘の行列がその先の行列に影響を与えてるわけです。最初は肘の子孫すべてに肘行列を適用するだけでそれっぽくなるので、そこからやってみましょう。

そもそもツリー構造とは

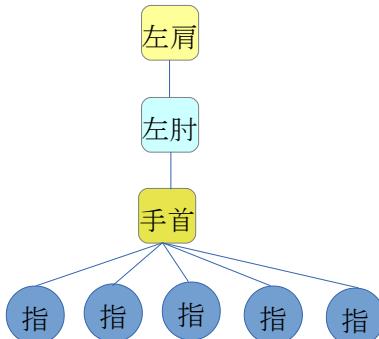
ツリー構造とはなんぞよ？

ヒトコトで言うと親子関係があるっちゅーことだ。ツリーには色々と種類があります。

[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))

よくやるのは二分木とか赤黒木とか2-3-4木とかAVL木とかですかね。

まあ、今回はそういうくだらない専門的な話はともかく、



これを表すデータ構造を作りたいんっすわー。

どうするっすかねー。ていうか、何に対してツリーを構築したいのかを明確にしこなにいかんよねー。これ忘れて「ツリー」しか頭に無かったらグダっグタになるからね。

ちょっと改めて考えてみてよ。

何のために階層構造が必要なん?

そもそもの目的は何よ?

…そうね。肘の回転手先まで伝播させたいがためやんね?

肘の回転てなんやつけ?

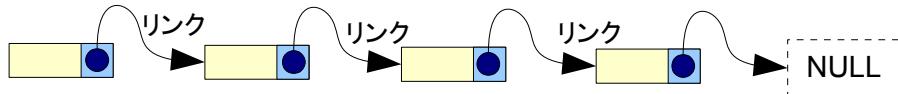
そう、行列なんやね。行列は今とりあえず XMMATRIX boneMatrixes(512)なわけだから。こいつらの親子関係を構築すりゃいいんや。

で、前に言ったかも知れんけど、ツリーってのはリストのもうひとつ複雑なやつなんよ。残念ながらリスト構造が分からん人はこっから先は苦労すると思う。

まず、リスト構造を思い出しましょ。あれは次のやつがどうなんかってのをそれぞれのオブジェクトが持ってるものですね?

自分のメンバにもう一つ余分に自分の「次」を指示する変数を持つとけばオッケーって

やつね。



こういうやつ。

さて、ではツリーリーってどういうものなんかー?っていうと「次」ってのを拡張して「子どもたち」にしているわけね。

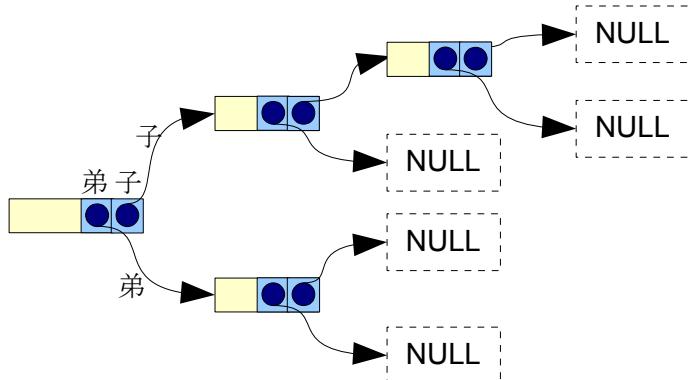
で、大体そんたくさん管理すんの面倒なんで「子」と「弟」で考えることが多い。まあ、「子」はわかるだろう。

子は親の影響を受けるものだ。

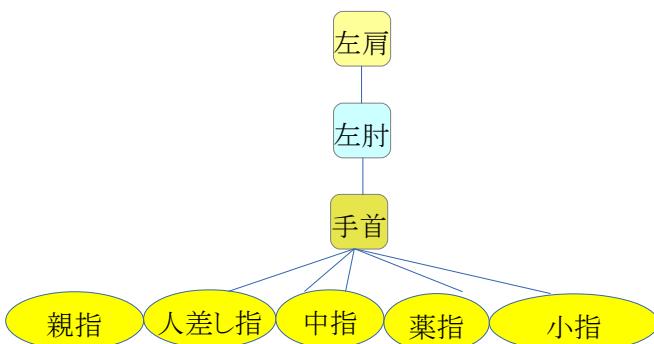
ただし、データ構造上の弟は兄の影響は受けない。つまり「弟」等と言ってはいるが、全然上下関係や順序などはない。

同じ親の「子」ってだけだ。

作り型はリンクを拡張して、もうひとつふやして「子」と「弟」を指示示す。



つまりこの例でいようと



左肩の子は左肘、左肘の子は手首、手首の子は親指なんだけど、人差し指は手首の子なので、親指の弟に当たる、で弟…弟で小指まで行く。

ややこしいけど「子」と「弟」って概念が必要なのはわかるだろう?まあ萌えたい人は「娘」と「妹」でもいい。正直呼び方などはどうでもいい。

ツリー実装

さて、実装に入っていくがとりあえずは子と弟さえあればいいので、こういう構造体を考える。PMDに型を合わせてケチるならこうだろう。

今回はポインタが苦手な人用にも理屈がわかるように作ってみます。ポインタの理解に自信がある子はポインタでやってみてくれ

```
///ボーンの木の節
struct BoneNode{
    unsigned short child;//娘インデックス
    unsigned short sibling;//妹インデックス
};
```

あくまでも娘はdaughterで妹はsisterなんだけど。女性差別にならんように無性別で表しております。daughterとsisterがいい人はそっちで書いて下さい。

で、こいつをボーン数ぶん用意する。

```
std::vector<BoneNode> boneTree(boneNum);
[20].bone_name[0]          8D B6 8E E8 8E F1 00 FD 左手首
[20].bone_name[16]         FD FD FD FD
[20].parent_bone_index    0013
[20].tail_pos_bone_index  005D
[20].bone_type             00
[20].ik_parent_bone_index 0000
[20].bone_head_pos[0]      40A09408 414F6C27 3E8108A2
[21].bone_name[0]          8D B6 91 B3 00 FD 左袖 ...
[21].bone_name[16]         FD FD FD FD
[21].parent_bone_index    0013
[21].tail_pos_bone_index  005C
[21].bone_type             01
[21].ik_parent_bone_index 0000
[21].bone_head_pos[0]      40906569 41560B3A 3E7F3B21
[22].bone_name[0]          8D B6 90 65 8E 77 82 50 00 FD FD FD FD FD FD 左親指1
[22].bone_name[16]         FD FD FD FD
[22].parent_bone_index    0014
[22].tail_pos_bone_index  0017
[22].bone_type             00
[22].ik_parent_bone_index 0000
[22].bone_head_pos[0]      40A669E8 4147F0C4 BE522002
[23].bone_name[0]          8D B6 90 65 8E 77 82 51 00 FD FD FD FD FD FD 左親指2
[23].bone_name[16]         FD FD FD FD
[23].parent_bone_index    0016
[23].tail_pos_bone_index  005E
[23].bone_type             00
[23].ik_parent_bone_index 0000
[23].bone_head_pos[0]      40ADEA51 41435BDF BEBBBF56
[24].bone_name[0]          8D B6 90 6C 8E 77 82 50 00 FD FD FD FD FD FD 左人指1
[24].bone_name[16]         FD FD FD FD
[24].parent_bone_index    0014
[24].tail_pos_bone_index  0019
```

ちょっとこのデータを見てくれ、こいつをどう思う？末端から見てみますか？

`parent_bone_index`が親のインデックスを表していると思って下さい。

左人差し指1の親は $0 \times 14 = 20$ 番目なので → 左手首

左親指2の親は $0 \times 16 = 22$ 番目なので → 左親指1

左親指1の親は $0 \times 14 = 20$ 番目なので → 左手首

左袖の親は $0 \times 13 = 19$ 番目なので → (スクショでは見えてないけど) 左ひじ

左手首の親は $0 \times 13 = 19$ 番目なので → 左ひじ

ちなみに左ひじは $0 \times 12 = 16$ → 左うで

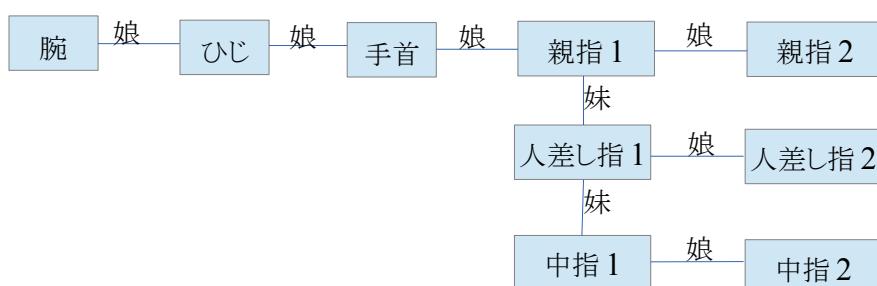
はい、でも思い出して下さい。ここで欲しいのは親インデックスですか？ そうじゃないですね？ 「娘」と「妹」ですよね？

この子が「親」を知っている状況から「娘」「妹」のツリーデータに変換していくわけです。実際現状でもツリー 자체はできているのだから「さかのぼり」だけでも十分だとは思います。要は最終的に関連行列の合成が行えればオッケーなわけですから。

ただ、ツリーだと再帰と組み合わせると割りと簡単にイケるんでそっちでいきます。

それはさておきツリー作りましょう。

上の例だと殆どが親子関係なんだけど、親指1と人差し指1が姉妹関係にあることがわかる。というわけで、図に書くと



…こうなる。

で、どうやってこれを構築していくのかというと、ループしながら親を探し、親があるのならその親の娘を現在のものにします。ただし、娘のインデックスが有効(0xFFFF 以外)であるのならば、その娘の妹に託します。ただしこの妹がまた有効(0xFFFF 以外)ならばそのまた妹に託すという感じに構築します。

```
//ツリー情報初期化
```

```
memsetで全部に 0xFFFF 入れとく
```

```
//ツリー構築
```

```

for( int i=0;i<boneNum;++i){
    親インデクス=_boneInfoes(i).parentBoneIdx;
    //親いないなら処理しない
    if(親インデクス==0xFFFF){
        continue;
    }
    BoneNode& pnode=_bonetree(親インデクス);
    //娘いないなら娘作る
    if(pnode.child==0xFFFF){
        pnode.child=i;
        continue;
    }
    int idx=pnode.child;
    //娘が1人以上居るのなら、妹に割り当てる
    while(_bonetree(idx).sibling!=0xFFFF){
        idx=_bonetree(idx).sibling;
    }
    _bonetree(idx).sibling=i;
}

```

終わり＼(^o^)／簡単でしょ？むずい？そうですか…ここまでみんなの恨みレベルがわかってきたので、手抜きして擬似コードを書いたんですけどね。え？擬似コードが二番目に手抜きなんですよ？

一番手抜きはソースコードをまるまるここに書くことなんですけどね。みんなもソッチの方がいいのかもしれないけど、それじゃ何にも学ばない人が出てくるからね。まあここまでついてこれてる人は意図を理解してくれていると思いますので、敢えて擬似コードで書いてるんです。

はい、それはともかくこれでツリーができました。あとは肘曲げを末端にまで伝播させていきます。

再帰(サイキ)呼び出し

こういうの(ツリー)を伝播させるのには再帰が便利です。

再帰とはなんぞや?

これも重要な技術だから覚えておいて欲しいのですが、再帰呼び出しとは、関数が自分の関数の処理内で自分自身を呼び出すことです。

どういう事がというと

```
void func(){//俺の名前はfuncだぜ  
    func();//俺自身(func)を呼び出しそ  
}
```

こういう感じです。当然、上の例だと「無限ループ」です。結構気をつけて使わないと簡単に無限ループするので気をつけて使おう。

ツリー構造に再帰処理、これ最強。しかしこれをやると無限ループの危険性があるという諸刃の剣。素人にはお勧めできない。まあお前らは長いこと「専門教育」を受けてきているので再帰くらい使いこなしひとけってこった。

というわけで、再帰処理にとって重要なのは「終了条件」である。ここがグタグタだといとも簡単に無限ループする。

基本的にツリー構造ってのはツリーの根っこ(root)から始まります。ツリーに考えると0番の“センター”ですね。で、こつから木のように娘へのリンク、娘またその娘や妹へリンクしていくきます。

で、末端は0xFFFFですよね?なんかというと最初に0xFFFFで初期化して、娘が妹がいればそのインデックスになっているはず。そうじゃなければ0xFFFFのまま。

つまり自分の娘が0xFFFFって時点で終了～ってわけ。

で、子に情報を伝播させるんだから

```
void 行列伝播(インデックス, 親行列){  
    boneMatrices[インデックス] = 親行列 * boneMatrices[インデックス];  
    while(自分の娘さん全員ぐるぐる){  
        行列伝播(娘さんインデックス, boneMatrices[インデックス])  
    }  
}
```

です。

簡単ですね。娘さんいなければもう「行列伝播」関数は呼び出されないんですからー。

ここまでやれば…うおーーーー!!!



曲がりました!!曲がりました!!!

さてうまくいくモデルはうまくいくもんですが、「捻り」が入っているボーンがある場合どうも上手く行っておりません。

「捻りボーン」だが「IK影響下ボーン」がある場合、事はそんな単純ではないみたいでアリスちゃんも島風ちゃんも一筋縄では動きません。

現状、アリスの場合はIK影響下ボーン(ボーン種別4)が、肘ボーンの子についてるのですが、その1つの影響範囲が二の腕まで到達しているのです。

ですから、肘を動かそうとすると二の腕の半分くらいが同時に移動し、図のように腕がすっぽぬけたような現象が発生します。



これではイクナイので4番の時は直前のボーンの影響をなくすようにしてみる。



そうすると、いちおう肘にはくっつくのですが、これを「肩」と「肘」を同時に動かしてしまうと



非常にマズい事になります。

一応、色々と処理を行うと



このように、きっちり曲がってくれます。ちなみに色々とやってもおかしかった原因のひとつは俺が「左ひじ」の上は「左腕」と決めつけていたからです。捻りがあるやつはそうとは限らないため、名前をきっちりまずは確認しましょう。

で、ボーン種別4と8に気をつけていればアリスは表示できます。まだテキストに纏める時間はないため、もうちょっと待って下さい。

更に言うと、「島風」はデータがこのようになっています。

[0].bone_name[0]	91 80 8D EC 92 8B 90 53 00 FD FD FD FD FD FD FD 操作中心
[0].bone_name[16]	FD FD FD FD
[0].parent_bone_index	FFFF
[0].tail_pos_bone_index	0000
[0].bone_type	01
[0].ik_parent_bone_index	0000
[0].bone_head_pos[0]	00000000 00000000 00000000
[1].bone_name[0]	91 53 82 C4 82 CC 90 65 00 FD FD FD FD FD FD 全ての親
[1].bone_name[16]	FD FD FD FD
[1].parent_bone_index	FFFF
[1].tail_pos_bone_index	0002
[1].bone_type	01
[1].ik_parent_bone_index	0000
[1].bone_head_pos[0]	00000000 00000000 00000000
[2].bone_name[0]	83 5A 83 93 83 5E 81 5B 00 FD FD FD FD FD FD FD センター
[2].bone_name[16]	FD FD FD FD
[2].parent_bone_index	0001
[2].tail_pos_bone_index	0041
[2].bone_type	01
[2].ik_parent_bone_index	0000
[2].bone_head_pos[0]	00000000 41000000 00000000
[3].bone_name[0]	83 4F 83 8B 81 5B 83 75 00 FD FD FD FD FD FD FD グループ
[3].bone_name[16]	FD FD FD FD
[3].parent_bone_index	0002
[3].tail_pos_bone_index	0004
[3].bone_type	01
[3].ik_parent_bone_index	0000
[3].bone_head_pos[0]	00000000 41205B69 BE26DE00
[4].bone_name[0]	83 4F 83 8B 81 5B 83 75 90 E6 00 FD FD FD FD FD グループ
[4].bone_name[16]	FD FD FD FD

「『このように曲ってなんやねん』と思うかもしれません、フリーにツリーを作るセオリーが頭に入っていると失敗する箇所があります。0番と1番のデータを見てみると

0番の親インデックスは0xFFFFです。まあ、予想通りですね。親居ないんですから。では1番

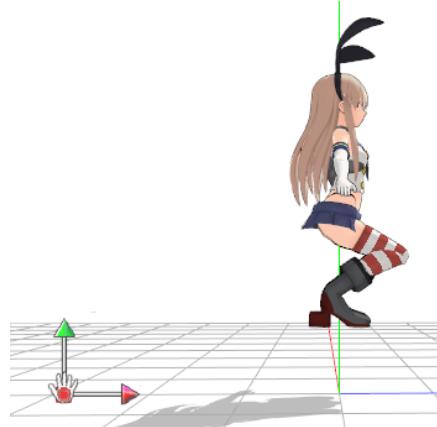
見てみましょう。

1番の親インデックスも0xFFFFです。

つまり、予想に反して最上位の親が2つあることになります…OTL。

原則的には親は一つなんでしょうけど…

ちなみに言うと、ボーン番号としては、0番はだれからもつながっていません。MMD上で見ても、関連性を見いだせません。



左下にあるのが「操作中心」です。こいつを動かしても変化ないです。…わからん。どうもこいつはツリーにおいては仲間はずれっぽいので、ツリー構築の際に

- 親が居ない
- 子もない

これらを満たしている場合はツリーのルートとして認めなければとりあえずはオッケーだと思います。

つまり、このようなデータ…「子が居ない、かつテールインデックスが0番の場合はルートにしない」の場合ルートノードは0番ではなく、1番とします。もちろん1番も同じようなボーンなら2番…3番としていきます。

そうすれば…!?



そしてこの欠損少女である。

どうしてこうなった。どうしてこうなった。(^ω^)

ぐぬぬ…わからん。

まあこのままここで詰まってしましがたがないので、別の話をしよう。各個別のモデルの不具合については全体的な解説ができたら話していきましょう。

行列について再考

行列の×の順番と、転置についての話

何度も言っていますが、行列はかける順序で結果が変わってしまいます。BLモノも同じですが、順番が変わるとエライことになります。

で、この行列の乗算の順序と、行列の転置にはちょっとした関係性があります。

数学的な行列の掛け算の結果とプログラムの結果が若干違ったりするので、そのへんがヤコシインですが…

今まで、`XMMartrixTranspose`を使って転置をしていただいておりましたが、アレは何故かというとシーダ側のコードが

```
out.pos=mul(pos,mul);
```

という順序になっていたからで、本来は、直感には反するのですが

```
out.pos=mul(mul,pos);
```

が正しいわけです。で、ボーンも入ってきて段々と行列関係がややこしくなってきましたので、Transposeせずに

```
out.pos=mul(mul,pos);
```

こっちの順序に直してしまったほうがいいと思いつますので、直しましょう。

さて、<転置をする→順序をひっくり返す>の対応関係をきちんと見てみましょう。例えばある点を変換するのに

$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ という行列で、点 (x, y) を変換してみよう。この場合、行列は左からかけます。

$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$ となります。試しに回転行列で考えてみるとわかりやすいかなーとは思います。 90° 回転ならば

$A = \begin{pmatrix} \cos \frac{\pi}{2} & -\sin \frac{\pi}{2} \\ \sin \frac{\pi}{2} & \cos \frac{\pi}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & -1 & y \\ 1 & 0 & y \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}$ となり、見事に 90° 回転している事がわかる

と思います。しかしながら、こいつを逆からかけるとどうでしょう…そうですねー、以前にもお話ししたとおり、全くの別物になってしまいます。

$A = \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} \cos \frac{\pi}{2} & -\sin \frac{\pi}{2} \\ \sin \frac{\pi}{2} & \cos \frac{\pi}{2} \end{pmatrix} = \begin{pmatrix} 0 & x+1 & y \\ 1 & x+0 & y \end{pmatrix} = \begin{pmatrix} y & -x \end{pmatrix}$ ね。これは回転行列だし、 90°

だとそれほど目立ちませんが、別物であることはわかると思います。

$A = \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} ax + cy & bx + dy \end{pmatrix}$

はい、別物ですね。

かつて `XMMatrixTranspose` を何故使ったのかというと、まあ慣れるまでは行列は右から書けたほうが無難かなと思ってかけていただけなのです。というか事実 C++ で Direct3D9 のみを使用している場合は、この順序でオッケーだったのです。

ところがシェーダといふのは数学的な順序での行列乗算となっておりますので、掛け算順序が逆になるというわけです。

これを序盤のうちに手取り早く解決するには転置を使うのが簡単でした。

転置といふのは以前にもお話ししましたが、縦と横を入れ替えるといふものです。縦と横を入れ替えれば

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

のように、かける順序を入れ替えてもうまくいくということです。これによりうまく行っていただけの話。モチロン行列同士の乗算も同じ

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

反対からかけても転置してさえいれば…

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

と、答えが「ラバラにならず」(転置はされた状態だが)、同じ答えが出てくる。

よくわからない人は実際の数値を入れてみればいいだろう。

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

と

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

と言った具合に、答えが全然違うものだが、これも転置すると…

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

元の答えを転置したものになってはいるが、答えは一致しているということがわかると思います。つまり、シェーダ側のかけ算の順序を本来のもの「**かけたい変換行列を左側から乗算していく**」のであれば転置の必要はありません。

じゃあ転置なんてもう必要ないのかというと、そうではなくビルボード行列作成時の計算簡略化のために転置を使うことになります。

C++ 側は右に右にかけていくので、イメージが非常に面倒であるが仕方なし。

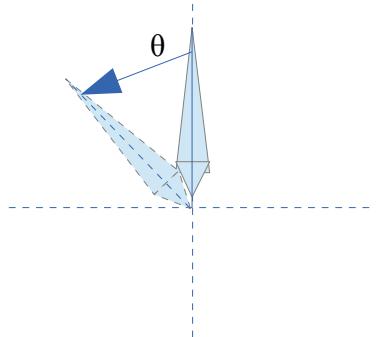
行列と頂点とボーンの関係

ボーンを曲げると何故、実際の腕とかのポリゴンがそれに合わせて曲がるのか、何故それでもともに動くのかってのが納得いっていない人もいるかもしれません。

まずは、納得してほしいのは、3次元空間上の点に対してなんかしらの<行列>をかけると、どんな点であっても問答無用で変換します。

<回転行列>であれば、どんな点であっても問答無用で「原点中心に回転」させます。しつこいようですが、ここはしつこく言わないといけないところです。

で、単純で初步的なボーンは基本的には「回転行列をつくるもの」です。つまり、原点にボーンの根っこ(頭)があるのならば



このボーンを回転させることによって「回転行列」が発生します。言い換えると、ボーンを回転させるのに必要な行列が作られます(前回のでやらせた「原点移動→回転→もとの位置」にするために作った行列)。

で、数学の世界では「回転」は \sin と \cos でできている「回転行列」によって作られます。2次元の回転ならばこういうものです。

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

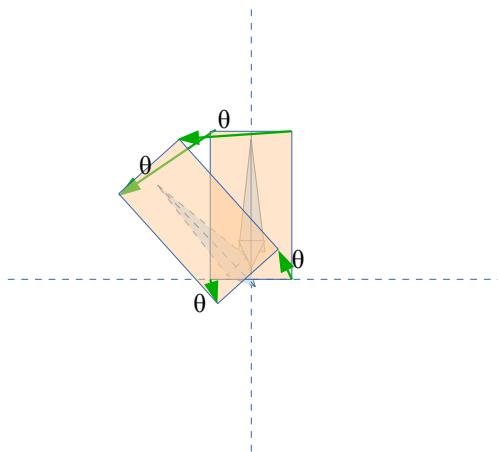
どこかで見たことがありますよね? そう、コブラのマシンはサイコガンのアレです。ともかく行列によって回転、平行移動を行えます。

で、上の数式を見てもらえれば、座標上の点 (x, y) に対する回転であることがわかりますね? ということは、無数の頂点に対しても適用可能なわけです。

どういうことなのかというと、空間上の点であれば、どのような点でも原点中心に回転させてしまう、そんな行列なのです。

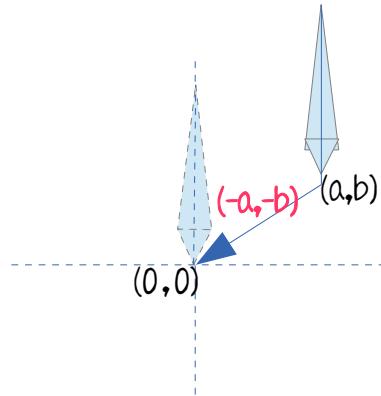
例えば長方形の場合、頂点が4つですよね? ボーンがこの4点に影響するのならば、この4点全てが原点中心に回転します。

つまり…



この理屈はわかるかな？それぞれの頂点が原点中心に回転すると、長方形の形を保ったまま回転することになるんだね。

じゃあ、原点中心でない場合はどうなるのかというと、一回原点に戻す必要があるんだったね。つまり例えばボーンの根っここの座標が (a, b) だったとしよう。



そうすると、原点 $(0, 0)$ に戻すためには $(-a, -b)$ 移動させる必要がある。…まあ、移動させること自体は簡単ですわな。足し算と引き算しか無いんだから、小学生だって計算できる。

しかし、これでは回転と組み合わせて一律に計算することができない。何とかして平行移動も行列で表現できないだろうか…。うーん、悩ましいですね。

ということで考えだされたのが「現在の次元で表現不可能なんだったら、次元を増やせばいいじゃない!!!」って発想。これを**同次座標系**と言います。英語で言うと homogeneous coordinate systemです。わざわざ英語の訳を言うのは、たまたまにホモジニアスのHが関数名として使われるからです。

はい、この同次座標系、3次元にするとややこしいので2次元から話していきますが、例えば二次元座標系ならば座標は (x, y) とするところですが、行列による平行移動演算をするためにムリヤリ $\rightarrow (x, y, 1)$ と表現します。

え？ なんで？ うん、その疑問は正しいが、もう少し待つがいい。

例えば、この同次座標系にて原点中心の回転を表現するところだ。

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

なんや!? ややこしなつただけやないけ!! そうだねプロテインだね。

ところがこいつに平行移動をさせようとすると話が変わる。例えば、とりあえず θ を 0 として何も変化しない状態で、 (a, b) だけ平行移動すると考えてくれ。そうすると

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

となり、回転後に平行移動するという表現ができる。

実はこの状態、行列の乗算合成で作ることができるのである。回転 → 平行移動なので回転の左から平行移動行列をかける。

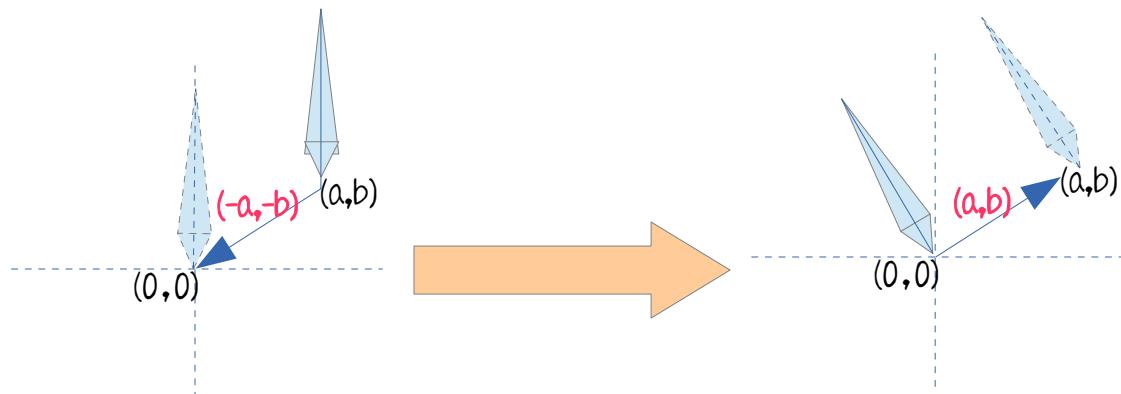
$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta & a \\ \sin \theta & \cos \theta & b \\ 0 & 0 & 1 \end{pmatrix}$$

疑り深い人は自分で紙で計算して欲しい。

ということは原点移動 → 回転 → 平行移動は（もう書きたくないが）

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

こんな感じになる。



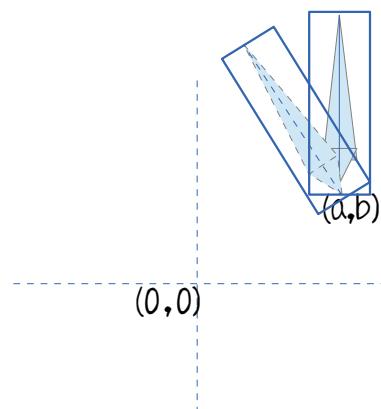
で、当然ボーンが (a, b) 平行移動するような変換であれば、それに影響する頂点もそれぞれが (a, b) 移動する。

これは大丈夫？ ボーンが $+a, +b$ 動くってことは、全ての頂点の座標に対して $+a, +b$ されるんだから、ボーンが動いた量と同じだけ動くよね？

例えば長方形であれば、長方形の形を保ったまま平行移動します。長方形でなくとも同様で

すね。

ですから、



このようにボーンをボーンヘッド"中心"で動かせば影響ボーンもそこ中心に回転することは理屈としては分かっていただけたのではないでしょうか?

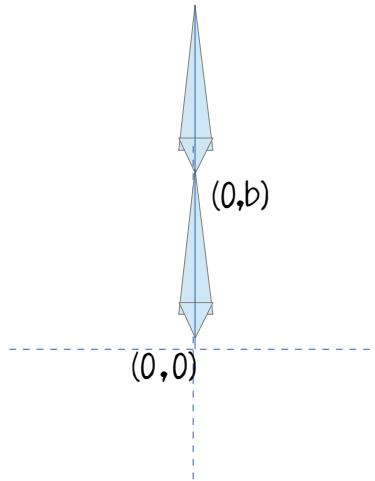
理屈としては分かる。分かるんだが感覚として分かんないって人は自分で図を書いて、実際に行列の計算をやってみてからにした方がいいと思います。「感覚」ってのは自らの体と頭を動かさない限り身につかないものだ。

自転車だか泳ぎだかギターだか、何でもそうなんよ…。

さて、ここまで分かっていただけたとして、2つのボーンが影響しているってどういうことなん?と思う人も居るかもしれませんのでその話をします。

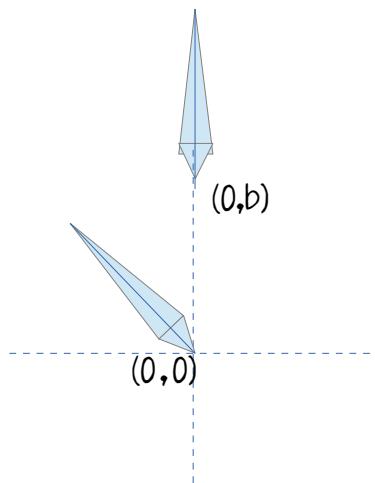
2つのボーンと影響度

そう、2つ以上のボーンが影響しあって初めてスキンメッシュは完成するツツツツ!!!
というわけで、2つのボーンについてお話しします。



簡単にするために原点からY軸に伸びている2つのボーンについて考えます。この2つには親子関係ができているとします。ツリーにおける親子関係です。モチロン原点に近いほうが親とします。

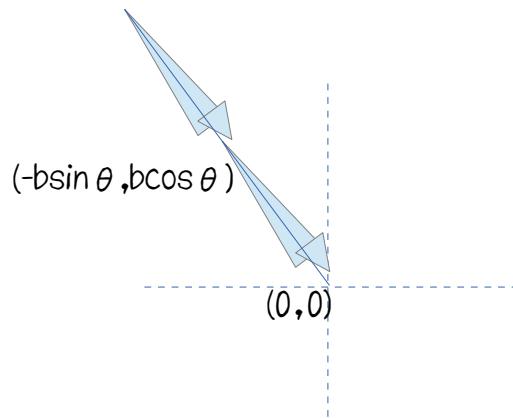
もし親子関係ができていなければ、親ボーンを回転させると



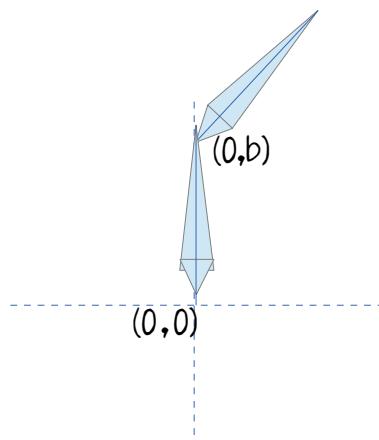
このように親子が泣き別れ状態となります。ツリーが上手く構築できていなかったり、ツリーができても行列をきっちり伝播できてないところになります。

ツリーを構築し、行列を伝播するってことはどういうことなのかというと、親の回転により子のボーンの座標が変わることを意味します。

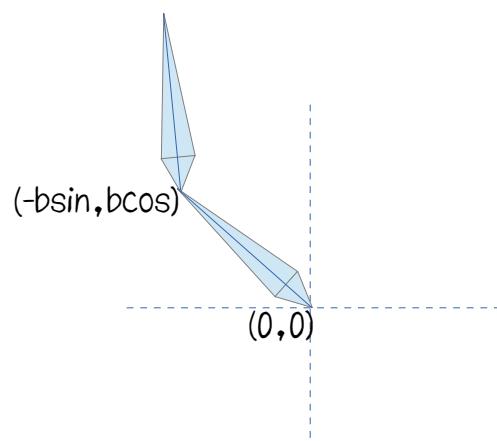
本来ならばこうなってほしい。



これは分かりますね？こうならないと肩を回しても肘からぶつちぎれますから。じゃあ子が回るのはどうでしょうか？



問題ないですね。きちんと平行移動→回転→戻しのステップができていれば簡単だと思います。では親も子も回転している場合はどうでしょう？



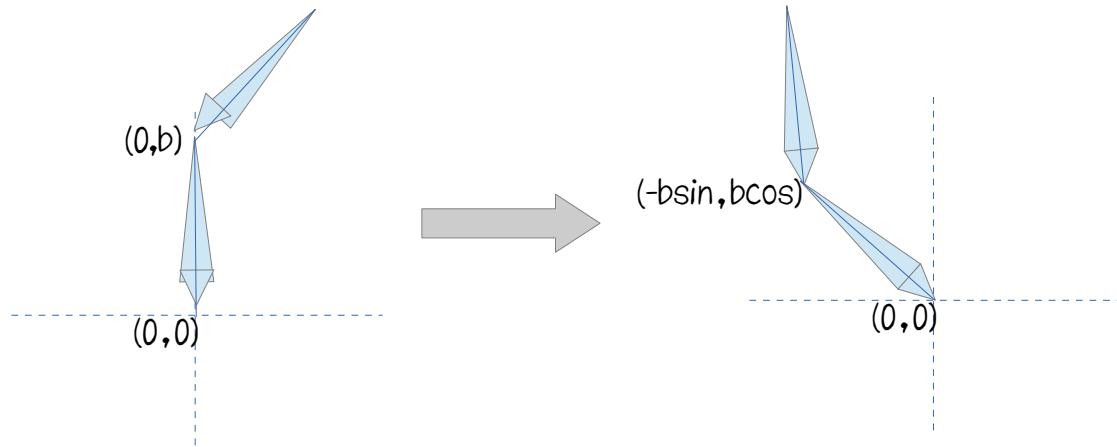
なに、恐れることはない。回転後だろうと平行移動後だろうと座標は座標である。つまり「原点、戻し→回転→もとの位置」は変わらないわけだ。

きちんと計算しさえすれば親回転が先でも、子回転が先でも構わない。別にセオリーがあるわけじゃない。

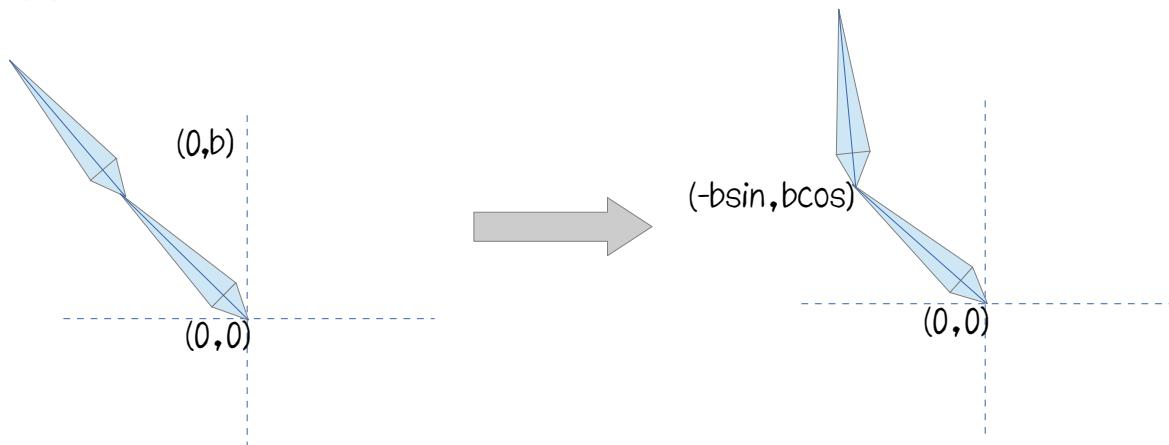
あとは「どちらを先に回転させるか」だ。言い換えると**行列乗算の順番**である。

仕組みがわかつていればどちらが先でも対応はできる。が、**どっちがラクか**ってのは大事な問いただ。

さて…どちらが楽だろうか？



だろうか？



それともこっちだろうか？

ちょっと考えてみて欲しい。そして今から言うやり方はあくまでも僕がラクだと思ってい
るに過ぎないことは理解しておいてくれ。

結論から言うと末端から回したほうがラクである。後々 VMD 適用時にどうなってしまうの
かはわからないが…理由は「いちいちオフセット回転まで考えるのは面倒」ってことなのだ。

どの道「原点戻し」が発生する以上は「ボーンオフセット行列(ボーンヘッド座標)」は持つ
ていなければならないのですが、今回そのボーンオフセット行列は

XMMatrixTranslation(offsetx, offsety, offsetz)で作っているため、意識はしていないかも
もしそれませんが、要はボーンの座標(ボーンオフセット)は知っている必要があるわけです。

で、例えば根っこから掛け算していくということは、そのボーンオフセットが次から次に変
換されてしまい、イザ末端を座標変換する際の最終的なボーンオフセット座標が考え方とし
てわかりづらくなる。

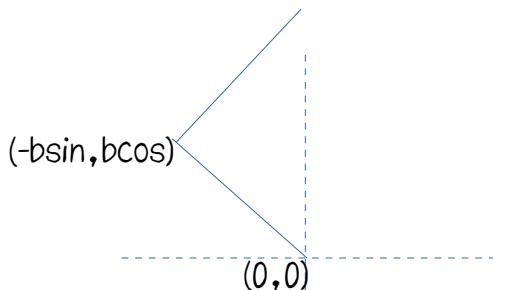
このためボーンの掛け算は末端からかけて行ったほうが楽であろう。というわけで今紹介しているコードは末端からかけていくコードとなっています。

おっと…重み(ウェイト)の話を忘れていたぜ…。

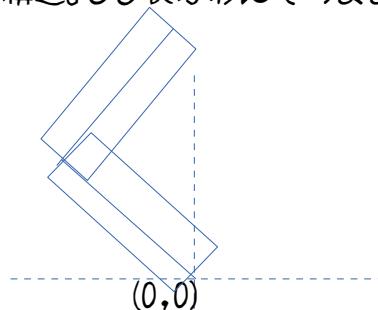
親子が曲がるのはいい、分かった理解した。

しかし「影響度」ってのを忘れていた。こいつが重み(ウェイト)ってやつだ。

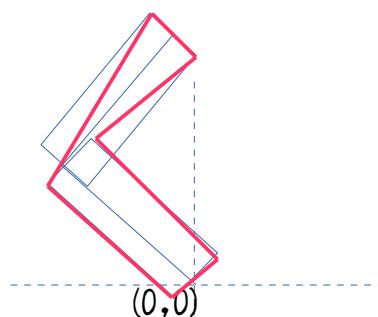
WEIGHTだか BONE_WEIGHTでセマンティクス定義しているとは思う。



ところで、このボーン構造。もし長方形にそのまま当てはめるのならば



こうなるだろう。切れ目が入っているし、めり込んでいる。もちろん連続する部分ならば頂



このようへしゃげた形となってしまい、よくない。

もちろん、子に 100% のウェイトを書けた状態でも同様だ。

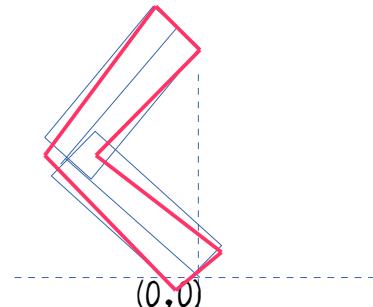
きちんとイメージしたい人は自分で紙に書いて下さい。これでは使えない。実際昔のゲームとかだとこういうへしゃげたモデルも多かった。

頂点ごとにどうこうできる仕組みがなかったのだ。

だが今はあるため、ここで、2つのボーンが影響する部分に影響度(重み/ウェイト)をつけてやることによって、もうちょっとマシにしてしまおうって考えた。

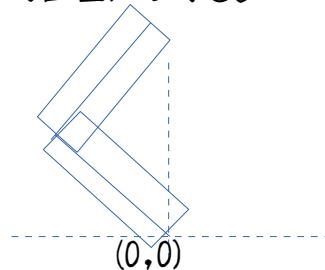
例えばさっきの共有部分のウェイトを親0.5で子0.5にしたとする。そうすると

図のようにモデルがへしゃげることなく形を保つことができるというわけだ。

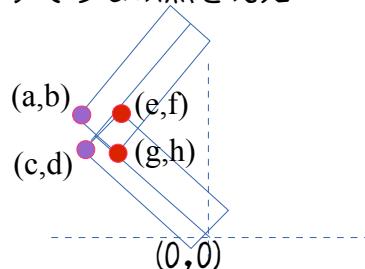


理屈がわからない人に言っておくと、この例で言うと

①2つの長方形をヴァカ正直につくる。



②両方の影響を受けそうな頂点を発見



③予め設定しておいたこの両方の影響度を見て頂点の座標を混ぜる

例えばボーンのままやると出てくる $(a,b)(c,d)$ の座標を混ぜるためににはそれぞれがどれくらい分配されるのかを $0.0 \sim 1.0$ の範囲で設定しておく。それを w とすると、最終座標 (x,y) は $(aw+c(1-w), bw+d(1-w))$ となる。

④それだけで終わり。ジ・エンド

実際は②のように「両方の影響度を受けるところを発見」なんてやってたら処理を食うので、片方の影響しか受けない部分の影響度は最初から1.0に設定されている。

ただし頂点の数や、曲げる角度によってはどうしても不自然になったりもするが、MMDは主に人間モデルを扱うため、ニンゲンの可動範囲のみ考えればいいため、2つのボーン影響度

しか持っていないのだろうと考えられる。

で、今回紹介したスキンメッシュアニメーションは、その仕組みを知ることが後期の授業のボーダーラインなので、ここまでできれば単位を落とすことはないでしょう。

自分の嫁の肘、腕がきちんと動けば単位C判定の最低ボーダーはクリアしたと思っていいでしょう。ただしパクリは認めない。

さて……ここからが本番だ。ここまで来た人は頑張ったね。此処から先はもっとマニアックで大変な事になってくる。おとなしく〇判定で満足しどとか、共に先に進むか、自分の研究に入っていくか…。

で、今一度言っておくが、既に10月末。最初にも言ったことだが、この授業のペースに合わせていたらゲームなど作れない。授業外でゲーム作っているか？中途半端は苦しいだけだぞ。

VMDを再生しよう

まあとずはVMDの元情報になってるポーズ情報から見てみましょう。VMDに入る前にVMDより見やすいポーズ情報VPDから見てみましょう。

ポーズ情報を見てみよう

MMDを立ち上げて



左腕と左ひじ(別にどこでもいいけど)を適当に曲げた状態で、左のタイムラインの左腕と左ひじを選択した状態で、「ファイル」→「ポーズデータ保存」してみよう。

で、テキストエディタで開いて中を見ると

Vocaloid Pose Data file

```
初音ミク.osm;           // 親ファイル名
2;                      // 総ポーズボーン数
```

```
Bone0{左腕
0.000000,0.000000,0.000000;          // trans x,y,z
0.000000,0.000000,-0.337818,0.941212; // Quaternion x,y,z,w
}
```

```
Bone1{左ひじ
0.000000,0.000000,0.000000;          // trans x,y,z
0.237900,0.618988,0.663254,0.346914; // Quaternion x,y,z,w
}
```

となっている。

はい、何の数字なんでしょう？

transはまあ移動ってのは分かるよねえ…。で、Quatationって何？

うん、概念的には前に話したんだ……覚えてないでしよううう

カタカナで書くと「オータニオン」です。覚えてない？四元数です。覚えてないか…いいよ…忘れて。

とにかく、この数値を **XMMatrixRotationQuaternion 関数** の引数に放り込めばそれでいい。モチロン、こいつは回転しかないのでボーンオフセットは各自やっていただきたいが、これを投入してやればいとも簡単にポーズをとります。



あ、なんかこの「お試しでやってみよう」的な部分だけ読んで、なんか

0.237900, 0.618988, 0.663254, 0.346914; // Quaternion x, y, z, w

のこの各数値を(X回転量, Y回転量, Z回転量)とか思って

「まともに動かないんッスけど～？センセの授業腐ってますね WWW」みたいに思ってる人も居るかもしれない。

よく見ろ…落ち着けッ

こいつはクオータニオンだツツツツ!!! XYZ ではないつ!!!

まあ学生のうちなら許されるだろう…だが、もうすぐ君たちは就職活動、そんな慌てんぼさんは



である。

まあ、分かろうと分かるまいと「先に進もうとする姿勢」そのエネルギー、その向こう見ずは学生らしくて



うん、大好きさ!!

なんだが、きちんと書いてあることは読もう。俺様は優しいので大事な部分は赤とか太字で書いてるだろ？

さて、とにかくさつきの数値を

```
XMMatrixTranslation(-elbowpos.x,-elbowpos.y,-  
elbowpos.z)*XMMatrixRotationQuaternion(XMLoadFloat4(&XMFLOAT4(0.237900f,0.618988f,0.663254f,0.346914f)))*XM  
MatrixTranslation(elbowpos.x,elbowpos.y,elbowpos.z);
```

てな感じで放り込めばいい。ほら、下線引いてるだろ？ポーズとるからさ。

じゃあ全身ポーズとれる準備から入っていこう。今回はelbowPosなどの個別のポジションのオフセットを保管していたが今からはそうではない。

全身のボーン座標のオフセットを保管しておくのだ。ちなみに今はやテールはいいらん。ヘッドのボーン座標だけあればいい。

この時点で、初音ミクだとXMFLOAT3を122個で十分であるし、さらに減らそうと思うのならば、まずはボーン種別が0番以外は無視する。そうすればまだいいが減らせますな。

…まあそこまでやるんだったら元のボーン情報そのまま使ったほうが早いか。

問題は

```
↑ 初音ミク.osm;^ ^ // 親ファイル名  
2;^ ^ ^ // 総ポーズボーン数  
↑  
Bone0{右腕^  
0.000000,0.000000,0.000000;^ ^ ^  
0.479111,-0.380736,0.012401,0.800106;^  
}  
↑  
Bone1{右ひじ^  
0.000000,0.000000,0.000000;^ ^ ^  
0.638612,-0.033037,0.169747,0.749846;^  
}  
↑
```

この文字列情報をどう扱うかやね…ね？この文字列を解析することを考えたらバイナリの方がラクでしょ？

そうでもない？

まあ、言うほど難しくないんだけど、面倒なんだわー。つべーわー。

ということで、1フレーム(vmd)モーションデータ)作って、それでポーズを取らせようそうしよう。

VMDファイルを見てみよう

とりあえずvmdのシンボルファイルは例によって例のごとくサーバに上げてるんでpmdン時と同様にやってくれ。ちなみにvmdの解説はここ

http://blog.goo.ne.jp/torisu_tetosuki/e/bc9f1c4d597341b394bd02b64597499d

である。

そしてこのデータである

header.VmdHeader[0]	56 6F 63 61 6C 6F 69 64 20 4D 6F 74 69 6F 6E 20	Vocalo
header.VmdHeader[16]	44 61 74 61 20 30 30 30 32 00 00 00 00 00 00	Data 0
header.VmdmodelName[0]	8F 89 89 B9 83 7E 83 4E 00 FD FD FD FD FD FD FD	初音ミ
header.VmdmodelName[16]	FD FD FD FD
motion_count	00000004	
motion[0].BoneName[0]	8D B6 98 72 00 FD FD FD FD FD FD FD FD FD	左腕
motion[0].FlameNo	00000000	
motion[0].Location[0]	00000000 00000000 00000000	
motion[0].Rotatation[0]	00000000 00000000 BEACF67B 3F70F345	
motion[0].Interpolation[0]	14 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B	
motion[0].Interpolation[16]	14 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01	
motion[0].Interpolation[32]	14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01 00	
motion[0].Interpolation[48]	14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01 00 00	k
motion[1].BoneName[0]	8D B6 82 D0 82 B6 00 FD FD FD FD FD FD FD	左ひじ
motion[1].FlameNo	00000000	
motion[1].Location[0]	00000000 00000000 00000000	
motion[1].Rotatation[0]	3E739C0F 3F1E75FF 3F29CB04 3EB19EB6	
motion[1].Interpolation[0]	14 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B	
motion[1].Interpolation[16]	14 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01	
motion[1].Interpolation[32]	14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01 00	
motion[1].Interpolation[48]	14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01 00 00	k
motion[2].BoneName[0]	89 45 98 72 00 FD FD FD FD FD FD FD FD FD	右腕
motion[2].FlameNo	00000000	
motion[2].Location[0]	00000000 00000000 00000000	
motion[2].Rotatation[0]	3EF54E09 BEBB8B264 3C4B2D90 3F4CD3BF	
motion[2].Interpolation[0]	14 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B	
motion[2].Interpolation[16]	14 14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01	
motion[2].Interpolation[32]	14 14 14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01 00	
motion[2].Interpolation[48]	14 14 14 14 14 14 6B 6B 6B 6B 6B 6B 6B 6B 01 00 00	k
motion[3].BoneName[0]	89 45 82 D0 82 B6 00 FD FD FD FD FD FD FD	右ひじ
motion[3].FlameNo	00000000	

ではぱっと見てわかることが、「こいつ…ボーン名で管理してやがるぜッ!!」

うん、これ一瞬「ええー」とか思うんですけど、ミクのモーションを他のモデルにも適用できるように「名前で管理」しているのだと思います。気に入らない人は「俺フォーマット」を作りましょう。ここまでついてこれている人なら造作も無いことでしょう。

ちなみにRotationはXYZ回転ではなく、しつこいようですがクオータニオンです。

こいつの詳細な説明はそのうちやろうとは思っているので、今は黙っておとなしくXMMatrixRotationQuatnion使つとこう。

さて、vmdを読み込んで、ボーンをvmdに合わせて回転させる事が今回のメインの話となる

であろう。

なのでしばらく重要になってくるのはボーン名(BoneName)とクオータニオン(Rotation)です。

その後にFlameNoと補間の話(最初は線形補間でええやろ)。

その次にベジエ曲線→ベジエによるイーズインイーズアウト

最後にIK…いけるかなあ…ってところかな。

で、何度も言いますけどこの授業の進度に合わせてたらゲーム作れませんからね。

ゲーム会社に就職するつもりがあるのならば授業外でゲーム作って下さい。企業は課題で作ったものとか求めてません!!!

艦隊これくしょんで遊ぶのもいいよ。うん、いいんですよ。でもそれはきっちり作るもの作った後でやることだと思います。艦隊これくしょんばっかりやってゲーム完成しませんでした。ゲーム業界行けませんでした…そう、そのたった数ヶ月の我慢ができないればっかりにゲームプログラマになることを諦める…今の愛宕ちゃんを愛で続けて、好きでもない職につくのかちょっとだけ我慢して毎日授業外で数時間プログラミングする…どっちだ!!どっちなんだ!!!今すぐ決める!!!!!!いや、そこまでやっても確かにゲームプログラマになれるか俺には分からへんのやけど



苦言はここまでにいたしましょう。時間が勿体無い。

とりあえずフォーマットの確認ですね。

VMD モーション名: char(30) 1*30BYTE;

VMD モデル名: char(20) 1*20BYTE;

モーションデータ数: uint 4BYTE;

これが最初に来ているので、まずは54バイトをガーッと読み込みましょう。

次に一つ一つのデータを読んでいきますが、

ボーン名:char(15) 1*15BYTE;//なにこの中途半端さ…PMXではもっとマシだと信じたい

フレーム番号:unsigned int 4BYTE;

移動量:float(3) 4*3=12BYTE;

クオータニオン: float(4) 4*4=16BYTE;

ベジエ値:char(64) 1*64BYTE;

…ぐらいでしょうか?ではいつものようにアライメントに気をつけつつモーションデータをロードしましょう。正直ヘッタにある『名前』はどーでもいいデータなので、捨てちゃっても構いません。

ロードできましたか?もうできますよね~?

構造体作ってもいいし

```
struct VMDHeader{
    char name[30];//モーションデータ
    char modelName[20];//モデル名
};

struct MotionData{
    char bonename[15];//ボーン名
    unsigned int frameNo;//フレーム番号
    XMFLOAT3 translation;//移動量
    XMFLOAT4 quaternion;//クオータニオン
    BYTE bezier[4][4][4];//ベジエ
};
```

さて…どうしましようかねえ…さっきも言いましたが、まずはボーン名とRotationですね。せつかくここまで来ている人たちですから…思い切ってmap使っちゃいましょうか。あー、難

しいかなー。

std::map 使ってみる

C#で Dictionary ってあるんですが、アレみたいなもんです。どういうものかというと、vector や list や set みたいに一つの値をドゥンドゥン蓄積していくためのものではなく、map は「キー」と「値」のペアを蓄積していくものです。



英語で言うと、Key and Value ね。

要は「配列の添字にインデックス以外を用いる」とでも思ってればいいんじゃない。厳密に言うとちょっと違うけど、初心者はそんなもんではよがんべ。

使い方はいたって簡単…でもないかもしれない。俺にはもう判断つかない。

```
#include<map>
```

で

```
std::map<キーの型, 値の型> 変数名;
```

で宣言する。

で、今回は名前とボーン配列番号の対応関係なので

```
std::map<std::string,unsigned short> boneIndicesMap;
```

こうとでも登録しておく。まあ正直別にこんなもん使わなくてもいいんだけど、こいつの面白いのは…

boneIndicesMap(ボーン名)=インデックス;

って書き方ができるわけ。こういう「文字列を配列添え字として使う」っていうのは「連想配列」って言うんだけど、これで VMD 読んだ時点でボーン名とインデックスペアを登録しておけば

VMD 読んでボーン名を見た時点でインデックス分かる→クオータニオンとインデックスが対応する→クオータニオンで曲げる→(° △ °)ウマ-

である。

これやっとくと、フレーム数が増えた場合でも割と対応しやすいと思いますので、興味アル人はお試し下さい。

この後の話は map してる前提で話していきますが、要は名前(PMD のボーン名)と VMD のボーン

ン名)を合わせて処理をしていけばいいって話なので、map使いたくないって人は、そこは読み替えて考えて下さい。

さて、std::mapするには

```
for (int i=0; i<boneNum; ++i) {  
    boneIndicesMap[_boneInfoes[i].boneName]=i;  
    :  
    中略  
    :  
}
```

でいいわけなんんですけどね。今度はVMDの方を読み込んだ後の対処ですね。

はい、ヘッタ部分は読み込んだ前提でフルクル

```
for (int i=0; i<motiondataCount; ++i) {  
    fread(&(_motiondata[i].bonename), sizeof(char), 15, fp);  
    fread(&(_motiondata[i].frameNo), sizeof(unsigned int)  
+sizeof(XMFLOAT3)+sizeof(XMFLOAT4)+64, 1, fp);  
}
```

ですかね。モーションデータ部分は前に提示したMotionData型のベクタを作っている前提で書きました。std::vector<MotionData> _motiondata(motiondataCount);ってところです。

ここまで書いて実行したらモーションデータが入ってるはず、そしてその中にボーン名が入っていると思いますが、それ一つを使うのですな。そつからインデックスとてくれれば終了

//ある点を中心にクオータニオン回転

```
void PointRotation(std::vector<XMFLOAT3>& boneoffsets, XMFLOAT4& quatanion, int idx) {  
    boneMatrixes[idx] = XMMatrixTranslation(-boneoffsets[idx].x, -boneoffsets[idx].y, -  
    boneoffsets[idx].z)*  
        XMMatrixRotationQuaternion(XMLoadFloat4(&quatanion))*  
        XMMatrixTranslation(boneoffsets[idx].x, boneoffsets[idx].y, boneoffsets[idx].z);  
}
```

こんな関数でも作っておいて、

```
for (int i=0; i<motiondataCount; ++i) {  
    PointRotation(_boneoffsets, _motiondata[i].quatanion, boneIndicesMap[_motiondata[i].bonename]);  
}
```

こんな感じで呼び出してやればオッケ。ただし、足のポーズは保証しない。IK入ってると難しいですな。

まあ、でもミクモデルやそれに準ずる単純なモデルならIK切ってやればきちんとポーズを取るので問題ないでしょう。

ミク系の単純なモデルなら、FKでなんとかなります。あとは時間による補間の話をていきましょうか。

フレーム間補間

はい、ポーズはとれたと思いますが、例えば歩かせたりダンスさせたりしたいのならば、ポーズとポーズの間をつなげてやる必要があります。

本来MMDはベジエ補間なのですが、難しいので線形補間からやっていきましょう。

線形補間(フレーム補間)

補間自体はボーンウェイトの時にやったと思うのですが、あれを時間で行います。で、時間で補間を行うというのはどういう事がというと、FKのみの場合、あるポーズというものは行列でできているのはわかると思います。

つまり、AというポーズからBというポーズに移行するための補間を書くならば、AとBの間の状態は

$$A(1-t) + Bt \quad (t=0.0 \rightarrow 1.0)$$

ということになります。ウェイトの時と同じですね。さて、この行列は各関節ごとに持っているのですが、どう扱いましょうか？無数のボーンがあり、そのデータもフレームごとにあります…と。それに必要なデータはVMDから読み取ってくる…と。

ちょっと今思いついたやり方は、今ボーン名をマップでとっているでしょう？で、VMDはそのまま持ってきていると思うんですが、VMDのデータを

ボーン名①—キーframe Aでの情報—frame Bでの情報—frame Cでの情報

ボーン名②—キーframe Aでの情報—キーframe Cでの情報

ボーン名③—キーframe Aでの情報—frame Dでの情報—frame Eでの情報

:

のように扱ってみてはどうだろうか？これは単なるアイデアである。**特に正解とかそういうわけではない**、セオリーなどない。

「設計」とか気にする人もそうなんだけど、設計に正解など無い！その時のチームの状況（メンバーやのスキル等）などによって変わるので臨機応変にしなければならない。

ひとつだけ、真理があるとすれば**「他人が読みやすいコード」を書くこと。**

変数名にαとかβとかγとか使わないようにしてください。テンプレート地獄にしないで下さい。

modelとかcameraとか、わかり易い名前にしましょう。

だが、これもバカ（英語が読めないバカ）には読めなかつたりするので、ホンマ臨機応変やでえ……バカが多い職場に行ったらバカに合わせないといけないんだねー。

だから、「自分のレベルの高さを自覚してる人」は、レベル高いところに行けるよう頑張りましょう。**キツイと思いますけど**自分のレベルは上がります。

レベルの低い職場に行ったら苦労するでえ…なんだよ int α4;って…OTL。でも自分のレベルが「高くなないとそういう職場に行くことになります。イライラします。だから自分のレベル + αくらいを目指して頑張りましょう。今のところに留まっては駄目だ。

ちょっとだけ横道にそれたが、先程のデータ構造にしたのはわけがある。何かと言うとスタートラインは基本的に全てのポーズ情報が入っている（動かさない状態かもしれないし、最初のポーズ情報かもしれない）

要は、次のキーフレームの情報を見て、線形補間してやればそれっぽくなるのである。だから最初にそれぞれのボーンの情報を読み取った際に、次のボーン情報にアクセスしやすければ制御がラクになるのである。

つまり、現在の「右足」ボーンを補完したい場合は、イメージとしては

```
int i = vmdata(右足).currentIndex;  
if(i+1 < vmdata(右足).size()) {  
    float div=(float)(vmdata(右足)(i+1).frameNo-vmdata(右足)(i).frameNo)  
    float t=(float)(currentFrame - frameNo-vmdata(右足)(i).frameNo)/div  
    bonematrixes(右足)=vmdata(右足)(i).mat*(1.f-t)+ vmdata(右足)(i+1).mat*t;  
}
```

はい、何度も言うようですが、これを写しても動きませんからね。言わんとすることは自分で汲み取って下さい。

なんかうまくいかない人もいるみたいなので、スクショ載せときます。

```
//登録されているボーンをなめる  
unsigned int endbonecount=0;  
for(;it!=keyframedata.end();++it){  
    //現在のインデックスを検索(だせえ)  
    int currentIdx=0;  
    float div=1.0;  
    for(int i=0;i<it->second.size();i++){  
        if(i+1==it->second.size()){  
            PointRotation(_boneoffsets,it->second[i].quatanion,boneIndicesMap[it->first]);  
            ++endbonecount;  
            currentIdx=i;  
            break;  
        }else{  
            if(it->second[i].frameNo<currentFrame&&currentFrame<it->second[i+1].frameNo){  
                div=(float)(it->second[i+1].frameNo-it->second[i].frameNo);  
                float t=(float)(currentFrame unsigned int KeyData::frameNo  
                XMVECTOR lapsedq=XMLoadFloat4(&it->second[i].quatanion)*(1.f-t)+  
                XMLoadFloat4(&it->second[i+1].quatanion)*t;  
                PointRotation(_boneoffsets,lapsedq,boneIndicesMap[it->first]);  
                currentIdx=i;  
                break;  
            }  
        }  
    }  
}
```

あと、余計なことかもしれませんのが、歩くモーションとか作って、一周して戻る際には最終フレームと最初のフレームをいい感じに「補間」しどうかないと「カクッ…カクッ」ってなりますので、できれば「補間」は入れることを考えておきましょう。

※訂正です。

クオータニオンを補間する場合は「線形補間」だと上手く行きません。おかしなことになります。あやつらは虚数の世界の住人なので実数世界の常識は通用せんのですよ。

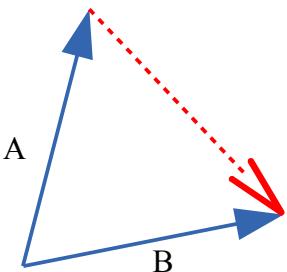
残念ながらクォータニオン用の補間関数を用意する必要があるのです。それが

XQuaternionSlerp

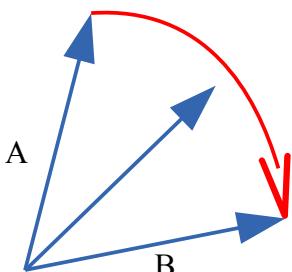
という関数。こいつは、クォータニオンとクォータニオンの補間結果をクォータニオンで返してくれるスグレモノなのが、単純な

$$A(1-t) + Bt \quad (t=0.0 \rightarrow 1.0)$$

と何が違うのかといふと…まあ、クォータニオンってのはベクトルみたいなものなんですね。その場合、線形補間をしちまうとちょっと良くない状態になるのです。どういうことがといふと…クォータニオンはイメージしづらいのでベクトルで見てみるとこうです。ベクトルAか



らベクトルBに補間すると…図のような状態となります。この何が問題なのかといふと、間のベクトルは、実際想定されるベクトルより縮まってしまうということです。ベクトルAからベクトルBに補間するという時、実際には



このようにあるべきなんですが、ただの線形補間だとそうはならないんですね。これをどうこうしてくれるのが XQuaternionSlerp なわけです。Slerp の S は Spherical(球の)って意味ですね。球面とかいうと難しく思うかもしれません、要はベクトルの長さを一定に保つような補間のことです。

そういうわけで補間プログラムを修正すると

補間後 $Q = \text{XQuaternionSlerp}(\text{Normalize}(\text{補間前 } Q_A), \text{Normalize}(\text{補間前 } Q_B, t))$;

といった具合になるでしょう。

この時の注意点は渡すクォータニオンを正規化しておくということですね。角度補間の形になるんですから、クォータニオン正規化しないと大変ってことだよね。ちなみにクォータニオン正規化は

$\text{XQuaternionNormalize}$ 関数を使用します。

この関数の中身についてですが、A と B の角度(内積)を求めて、角速度 ω が一定になるように回転させればいいと…その場合の回転中心は A と B の外積から求められるからいいんでないかと…とか思ってたら、

http://marupeke296.com/DXG_No57_SheareLinearInterWithoutQu.html

に詳しい説明がありますね。

個人的には角度の線形補間でもいいんじゃない?とか思ってたりするんですが…こんどやってみましょう。

※「球面」で思い出した余談ですがCG表現の世界でよく出てくる関数として「球面調和関数」なるものがあります。

詳しい説明は

<http://news.mynavi.jp/column/graphics/068/>

とか

<http://ja.wikipedia.org/wiki/%E7%90%83%E9%9D%A2%E8%AA%BF%E5%92%8C%E9%96%A2%E6%95%B0>

とか

<http://www.wakayama-u.ac.jp/~tokoi/lecture/gg/gnote10.pdf>

にあるんだが、わけわかんないですね。ただ、様々な所で使われているってのはわかるので、この辺を頑張って研究すればそれだけで発表できてるぶんみんな「オオオオオオわからん!」となると思いますし、企業にもアピールできると思いますので、調子にのってる人は頑張りましょう。

ちなみに

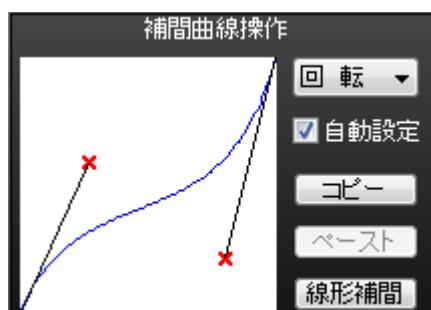
http://www.t-pot.com/program/109_RealTimeGI_SelfShadow/

http://www.t-pot.com/program/108_RealTimeGI/

で実際の活用例があるみたいなので、よく見ておきましょう。ありがとうございますt-potさんは。

ベジエへの話

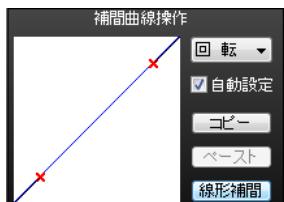
MMDのフレーム間補間は線形補間がデフォルトではあるんですが、きっちりMMDを動かしている人はきっちりベジエで設定している。「ベジエ、ベジエって何のことだよ」とおっしゃる人もいると思いますので、解説しておくとMMDを開いて適当なボーンを選択すると、左下の部分に



こういう画面が出てくると思います。こいつが「ベジエ曲線」です。で、線形補間の時に

$$C=At+B(1-t)$$

といった話をしました。その時の t はただただ時間の変化量を表しています。ただしそれだけだと、細かい動きの場合は不自然に見えることがあります。ものの動きは線形ではないからですね。



ちなみに「線形」ってのはこういうやつで、 $y=ax+b$ で表せる(つまり一次方程式で表せる…一次のこと)を「線形」って言ったりするのだ)

で、これにもうすこしなめらかに変化をもたせようというのが「ベジエ曲線」なわけだ。曲線には大雑把に言うと

- ベジエ曲線
- スプライン曲線
- エルミート曲線

などがあるが、ベジエ曲線が一番簡単ではある。ツールとしての使いやすさはちょっと劣るんだけどね。

ベジエはコントロールポイント(制御点)によって様々な曲線を表現できるものである。さて、ベジエについてだが、

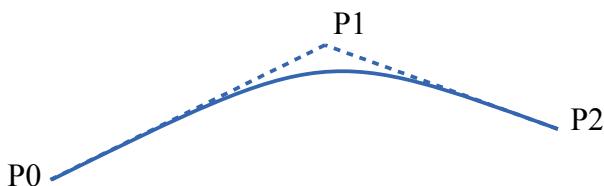
<http://ruiueyama.tumblr.com/post/11197882224>

の説明がわかりやすい。さすが「中学生でも分かるベジエ曲線」と言ってるだけのことはある。ここで私がゴチャゴチャ解説するより、これを見たほうがいいのではないか…。

数学的なベジエ解説

まず、線形の場合は制御点が2つ。この時点での見た目はは曲線として現れない。あくまで点と点を結ぶ先にすぎない。

この間に制御点を1つ追加することで曲線が現れ始める。



曲線が現れる様子は上のページの真ん中くらいのアニメーションを見ていただきたいのだが、時間 t の間に P_0 から P_1 まで移動する点があり、 t までの間で P_1 から P_2 まで移動する点がある。このそれぞれの「移動する点」を M_0, M_1 とすると

$$M_0 = P_0(1-t) + P_1 t$$

$$M_1 = P_1(1-t) + P_2 t$$

となる。そこはわかるかな?で、さっきのアニメーションをもう一度見て欲しいのだが、このM0,M1からできる線分の上を走る点の軌跡Bがベジエ曲線を表しているのだ。

で、これもアニメーションを見ながらイメージして欲しいのだが、時間tの間にM0→M1にBが移動しているわけ。つまり

$$B = M_0(1-t) + M_1 t$$

である。これを展開すると

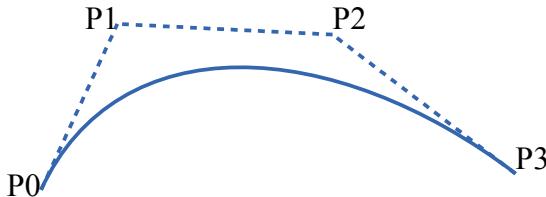
$$B = (P_0(1-t) + P_1 t)(1-t) + (P_1(1-t) + P_2 t)t$$

となる。さらに展開すると

$$B = P_0(1-t)^2 + 2P_1(1-t)t + P_2 t^2$$

となる。かたち的にはよく見る二次方程式の状態になってるよね?これを二次ベジエ曲線といいします。で、よく見れば分かる通り、制御点は3つですが、曲線が通るのは両端のコントロールポイントのみで、真ん中のコントロールポイントは通りません。

で、MMDのベジエはコントロールポイントが4つであるため二次ベジエでなく、惨事ベジエになります。つまり



このような形になる曲線というわけです。ただしルールは二次のときと変わりませんので、結局、間を移動する点は

$$M_0 = P_0(1-t) + P_1 t$$

$$M_1 = P_1(1-t) + P_2 t$$

$$M_2 = P_2(1-t) + P_3 t$$

と定義されることになります。さらに+という時間でM0→M1とM1→M2の間をそれぞれ移動する点をB0,B1とすると

$$B_0 = M_0(1-t) + M_1 t$$

$$B_1 = M_1(1-t) + M_2 t$$

となる。さらにさらにこのB0,B1という点の間を+時間で移動する点を最終ベジエBとすると、

$$B = B_0(1-t) + B_1 t \text{ となる。さて、面倒な展開の時間だ。まず第一段階}$$

$$B = M_0(1-t)^2 + 2M_1(1-t)t + M_2 t^2$$

こうですね。さて、第二段階やりましょうか。俺にはつかりクソ面倒な計算させずに君たち

も紙に書いて考えましょう。ここを面倒臭がる子に限ってアニメーションできません。

$$B = P_0(1-t)^3 + 3P_1(1-t)^2 t + 3P_2(1-t) t^2 + P_3 t^3$$

はい、中学生の頃にやった三次方程式みたいな状態になっていますよね？これが3次ベジエです。

はい。では、イヨイヨあのデータについて解説していきます。

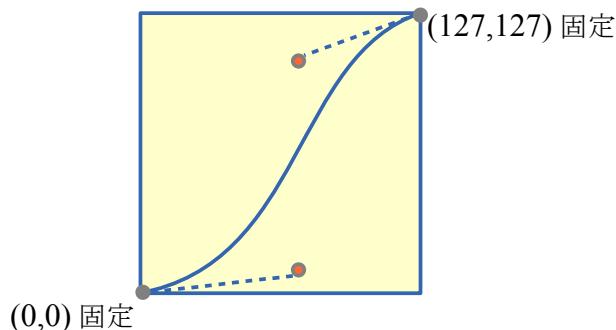
MMDのベジエデータとその応用

はい、どういうデータになってましたっけ？

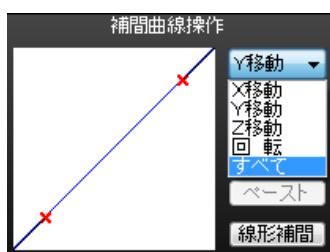
20	20	20	20	20	20	20	20	107	107	107	107	107	107	107	107
20	20	20	20	20	20	20	107	107	107	107	107	107	107	107	1
20	20	20	20	20	20	107	107	107	107	107	107	107	107	1	0
20	20	20	20	20	107	107	107	107	107	107	107	107	1	0	0

こんな感じのデータでした。

ちなみに右上と左下は固定のためデータは入っていません。



こういう状態です。じゃあ上のデータはどういうことなの…と思うでしょうが、曲線補間は見ての通り



X移動 Y移動 Z移動、回転という並びになっています。しかし…移動っておまえIKボーンしか「移動」操作できないじゃん…ムダじゃないのか…とか思うんだが。

まあともかく入っているデータが上の赤Xの二元的座標を1バイトで表しているのはわかった。ということは一行目の

20	20	20	20	20	20	20	20	107	107	107	107	107	107	107	107
----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

というのは左側半分が左下赤×で、右側半分が右上赤×の座標を表しているようです。

なるほどここまではいい。こういうデータだと線形補間状態だなってのもわかる。なんちゃない。

だがそれ以降のデータってどうなってんの?と思いつものページ

http://blog.goo.ne.jp/torisu_tetrosuki/e/bc9f1c4d597341b394bd02bb4597499d

を見てみると…

// 回転は4軸(クオータニオン)だが、1個にまとめられているので注意

// X軸 Y軸 Z軸 回転

// A(Xax,Xay) (Yax,Yay) (Zax,Zay) (Rax,Ray)

// B(Xbx,Xby) (Ybx,Yby) (Zbx,Zby) (Rbx,Rby)

// とすると、

// Xax,Yax,Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,

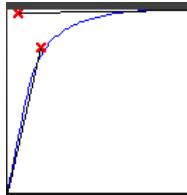
// Yax,Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,

// Zax,Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,00,

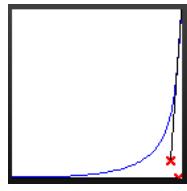
// Rax,Xay,Yay,Zay,Ray,Xbx,Ybx,Zbx,Rbx,Xby,Yby,Zby,Rby,01,00,00

などと書いてある。一行目は確かに予想通りかなとは思う。だけど、2行目以降が「ごめん、意味分かんない」状態である。これって一行目だけで良くない?とか思いますね。こういう「フォーマットに疑問」が出てきたら、まあ自分のやりたいようにやってみましょう。

で、縦軸と横軸の関係なんですけど、横軸は経過時間で、縦軸は実際の変化量を表しているようです。つまり、



こういう形だと、すぐに最終の形になりますし、



だと、なかなか最終形態にならない。そういうものです。

はい、ということで1つ目だけを使うとすれば、元々

Matrix=MatrixA*(1-t)+MatrixB*t;

こういう単純な線形補間だったものをベジエ補間にするわけです。

で、横軸を時間。縦軸を変化量だとすると

$\vec{P}_o = (t_0, v_0)$ のように時間と量のポイントで表すことができます。

もちろん、この t は上の式における t_0 と混同しないよう s として表しておきましょう。そうすると

$B = P_0(1-s)^3 + 3P_1(1-s)^2s + 3P_2(1-s)s^2 + P_3s^3$ という式になります。

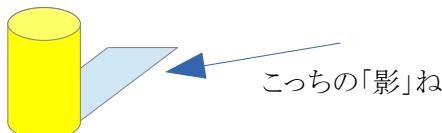
ベクトルの要素に分解すると

$$t = t_0(1-s)^3 + 3t_1(1-s)^2s + 3(1-s)s^2 + t_3s^3$$

CCD-IKについて考えよう

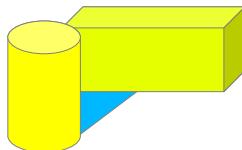
シャドウについて考えよう

この話は一度「2Dスレンダリング」以降を見てから勉強したほうがいいかもしれません。シャドウはシェーディングとは違い、「落ちる影」の方のお話です。

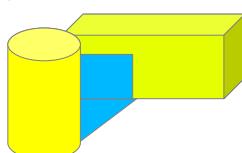


平行光線相手であれば、DX9の時代には D3DXMatrixShadow なんていう関数もあったんですが、XMでも一応あります。XMMatrixShadow という関数です。この関数はポリゴンを特定の平面に押しつぶすような変換をしてくれる関数です。1年の時に習った投影変換の3Dバージョンみたいなもんです。

まあこいつは確かに「影みたいなもの」を表示するお手伝いをしてくれますのでそれはそれでいいんですが…けっきょくこれができる影は、平面をベクトルで定義（要は法線ベクトルが a, b, c で定義され、オフセットが d となる）して、そこに対する投影行列を返すものなんだけどさ…



結局押しつぶしただけだから、影の位置に物体があっても上の図のようになってしまう…。本来は以下の様な感じで



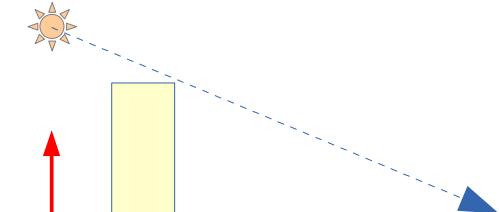
後の物体にも影が落ちるようにすべきなんですね。

これができない以上「中途半端」と言わざるを得ない。他に使い道はあるのかかもしれません
が、「影」として使うなら「丸影」でいいじゃんって気がします。

とりあえず、`XMMatrixShadow`について書いておくと

`XMMatrixShadow(平面のベクトル情報, 光線情報);`

ここから射影行列を返します。ちなみに平面のベクトル情報は法線のx,y,z成分とオフセット値をwとして書き込めばオッケ。



単純に図のように、光源と直線を結ぶベクトルが、指定の平面とぶつかる場所を変換先として返す行列ですね。

要はこの行列を頂点シェーダに渡してやって変換かけてやれば「影っぽいのは」は出ますのでそれで満足する人はこれでもいいでしょう。



潰しているだけなので、地面のポリゴンが無くても影が出ますが障害物には反応しませんし、当然セルフシャドウも発生しません。

丸影はなんというか、スポットライトみたいなもんですね。スポットライトは当たった場所（光の範囲内だけ）が明るくなりますが、丸影は光の範囲内だけが暗くなります。ここからの話は「落ちる影」相手なので、「落ちる先」を用意してやる必要がありますので、テキトーに「地面」を用意してみましょう。できれば障害物等あったほうがいいでしょう。

既にみなさんペラペラポリゴンと立方体を作っていますから、そつから作ってもいいですし、グラフィックツールからロードしても構いません。



とりあえず、こういう風に単純な地面を作りましょうか。ところで『影』ってなんでしょう？ そう、光が遮られた範囲です。光が遮られた範囲はどうなるのでしょうか？ 暗くなります。このアタリマエのことを実装してみましょう。

『光が遮られる』これがポイントです。光源から見て障害物があれば、そいつに光が遮られてしまいます。

『遮られているのかを知るためにには、光の方向から物体を見ればイイノデス!!』

これ、すげー頭やわらけー考え方ですわ。こういうのがクリエイターって奴ですよ。

ともかくライトから見た映像を作ります。ライトから見た映像ではカラー情報は不要なので、使用するレンダーターゲットのテクスチャはステンシルと同じやつで結構です。

まず、ライト方向から見たらどんな風に見えるのか確認しましょう。

現在、ライトベクトルを

```
float4 light= float4(-1, 1, -1, 0);
```

で作っている人は、ここからある程度引き伸ばした位置をライト座標としましょうか。どのくらい引き伸ばすかは人によって違うと思いますので、あえて言いません。

ぼくは20くらい離しておくことにします。で、ライト方向ビューを作ってライト方向からレンダリングしてみるとこうなります。



つまり、この目線から見えない部分には影が落ちるということですね？ ライトから見えないということはライトから隠れているわけですから…。

では、例によってライトビュー用のレンダーターゲットビューを作りましょう。で、その作ったレンダーターゲットに

`pos.z / pos.w` を返してあげましょう。

そうすれば、こういうテクスチャが出力されます。



この値が実はデプス値になっているのです。

```
depth = pos.z/pos.w;
```

最終的にこれを利用スルノデス。…とか思ってたら、やつぱりなんかおかしい。あれれ？こんなふうになるはずが無いんだが…

色々と試行錯誤した結果、どうも **SV_POSITION** で渡した **z** と **w** 値ではうまく計算されてないっぽいんです。じゃあどうすればいいのかというと、POSITION セマンティクスならオッケーみたいで。つまりどうすればいいのかというと、

```
float4 depos:POSITION;//デプス計算用 Position
```

とか作っておいて、頂点シェーダにて

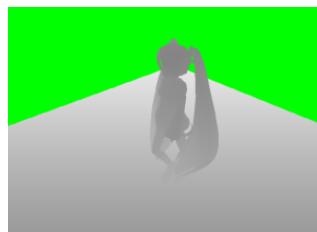
```
v.depos=v.pos;//座標情報をコピー
```

こいつに座標情報を置いておく。

そうした上で、ピクセルシェーダ側にて

```
return v.depos.z/v.depos.w;
```

としてやればいい。そうすれば図のように **z** 値が出ます。



例によって上の図は乗算かけて大きさにしているものです。さて、これで **z** 値がとれていることがわかりましたので、いよいよシャドウマップを行っていきましょうか。

*なんで **SV_POSITION** だとうまくいかないのかは調査中です。同様の質問に対して海外の質問サイトではこういう返答がされているようです。

<http://gamedev.stackexchange.com/questions/44403/in-hlsl-pixel-shader-why-is-sv-position-different-to-other-semantics>

どうも SV_POSITION ではピクセルシェーダに飛んだ時点で VIEWPORT 変換が行われ、0~1 の範囲ではなくなっているようです。流石に System Value は予想外の挙動を示しやがる…。

それは置いておいて、せっかく深度値を取得出来ましたので、シャドウマップです。

シャドウマップ

さて、この取得した Z 値テクスチャ…どのように使えばいいんでしょうか。この「一度ライトからの距離(Z 値)を計算した」のがポイントなんです。

今度(2/4 回目)は、カメラ座標からフリートレースにレンダリングしてみます。で、このレンダリングする際に、光源からの距離も一緒に計算しておきます。

え？ また？ そう。またなんです。さっきやったじゃん？ そう、さっきのはライトから一番近い部分が残りましたよね？

要は、この後に計算する「**ライトからの距離**」がシャドウマップの値より大きい(ライトから遠い)なら影の範囲内だということです。

単純でしょ？ 考えた人頭いいよね。



まあ… 単純に実装すると図のようになんか影が離れてしまいます。が、まずはここまでやってみてください。ご覧のようにセルフシャドウも入っています。これがシャドウマップによる影付けです。XMMatrixShadow を使ったやり方と違って物体がない部分には影が落ちません。

上の例ではなんで影が離れてしまっているのかというと、ピクセル値に直した時点で、一度 0~256 という 8bit に丸められているからです。まあ、そういうのはともかくやってみましょう。

前回まででライトビューからの眺めは出せていると思いますので、その後の実装から話していきます。手順としては

- ① ライト用レンダーターゲット作成
- ② ライト用レンダーターゲットに切り替え
- ③ ライトビューからのシーンをレンダリング
- ④ ①のレンダーターゲットに Z/W 値を書き込む
- ⑤ 表示用レンダーターゲットに切り替え
- ⑥ シーンの再レンダリング

- ⑦頂点シェーダにてライトビュープロジェクションを計算
- ⑧⑨のテクスチャ情報をシェーダリソースとしてセット
- ⑩⑪のシェーダリソースの Z/W 値と⑦の計算結果を比較。⑦の方が大きければ暗くする。

ちょっと手順多くなってきましたね仕方ないね。でもこの手順はここまでついてきた人なら特に難しくもなくソースコードを書けると思いますので、嵌りそうなところだけ言っておきます。

*まずは先程も言いましたが、SV_POSITIONを使用して Z/W を計算しようとするとうまいこといきません。

* Z/W > シャドウマップピクセル値

の計算ではそのポリゴンが一部含まれてしまうことがありますので、0.005だかなんだかを加算することによって回避して下さい。

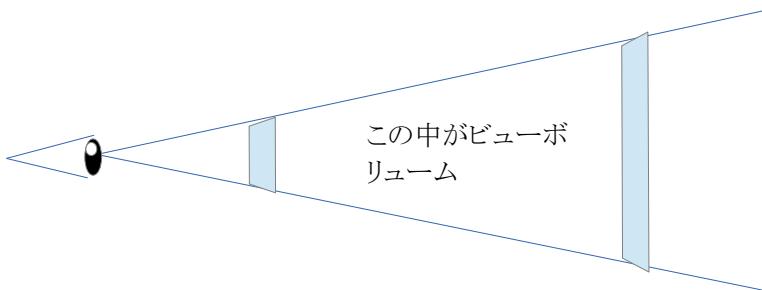
でここまで不明な点といえばテクスチャのUV値ではないでしょうか？

シャドウマップに Z/W 情報が書き込まれているのはわかった。では、そのテクスチャ情報のどれを見ればいいのでしょうか？

これもそんな難しいわけではありません。頂点シェーダで計算したライトビュープロジェクション変換後はスクリーン座標系になっているわけですね？

でそもそも W ってなんやねんっつゆ一話なんですが、通常のアフィン変換においては大した意味を持ちません。通常は $W=1$ にしつければ平行移動とかに使用できるとかそういうものです。

ただし、射影変換の W ってのはそれなりに重要な意味を持ちます。何かと言うと、プロジェクション行列は画角によって、遠くのものを小さくしたり、近くのものを大きくしたりします。で、眼球の近くから向こうのはしまで…今のプロジェクトでは0.1~100.fとかにしてる人が多いと思いますが、CGで表示出来る範囲は視点から始まる四角錐をこのnearとfarの位置でぶった切ったような形になっています。このぶった切った三次元台形を**視錐台**と言います。CG検定とかでは**ビューポリューム**とか言います。



パースペクティブの場合、 W ってのは通常遠くに行けば行くほど大きくなります。なんででかくなってるのかというと、遠くのものほど小さくしなければいけないからです。つまり、座標変換後の値を W で割ってやれば、 $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $0 \leq z \leq 1$ といった範囲内に収ま

ることになります。

ということは、ライトビューのスクリーンとして使用しているuv値は、ライトビュープロジェクション後のx,y座標をwで割ってやれば-1~1の範囲内に収まりますつまりシャドウマップのuvは

```
uv.x=v.depos.x/v.depos.w;
```

```
uv.y=v.depos.y/v.depos.w;
```

てな感じになります。こつから

```
float zdivw=shadow.Sample(samplerstate,uv);
```

とでもしてやれば、その座標のシャドウマップの深度値がわかりますので、あとは

```
if(z/w>zdivw){
```

暗く描画

```
}else{
```

通常描画

```
}
```

という感じにすれば影が落ちているように見えます。

※最後にひとつ注意点…

```
uv.x=v.depos.x/v.depos.w;
```

```
uv.y=v.depos.y/v.depos.w;
```

とは書きましたが、座標系の理がxyzとuvとでは違います。これはわかりますね？



xyzは画面中心から上がYプラスで、uvは左上中心で下がYプラスです。

つまりこれは考慮に入れなければいけません。さらに範囲も違います。

-1.0 <= x <= 1.0 , -1.0 <= y <= 1.0 を 0 <= u <= 1 , 0 <= v <= 1 にしなければいけません。更に言うとYが逆転します。どうしましようか…とりあえずuvは0開始なので1.0を足してみましょう。そうすると

```
0 <= x <= 2.0 , 0 <= y <= 2.0
```

になりますので、これを半分にすればいいので0.5をかけます。

```
0 <= x <= 1.0 , 0 <= y <= 1.0
```

さらにここからYを逆転させるには1-yにすればいいので結果的には

```
uv.x=(1.0+v.depos.x/v.depos.w)/2;
```

$uv.y = (1.0 - v.depos.y) / v.depos.w) / 2;$

こんな感じですね。

ここまでをやればとにかく影らしきものは出るでしょう。

で、最初の例で言いましたが、現状ではRGB各8ビットに出していますので、精度が悪いです。どうせデプス値しか記録しないのだから、R8G8B8_NORMにする必要はなく、R32でレンダーランゲットを作りましょう。

それだけで幾分ましになるでしょう。



今は平行光線を扱ってますので、影が広がるのはおかしいと思う人も居るでしょう。そういう場合はライトビュープロジェクションを作る際に、射影行列を

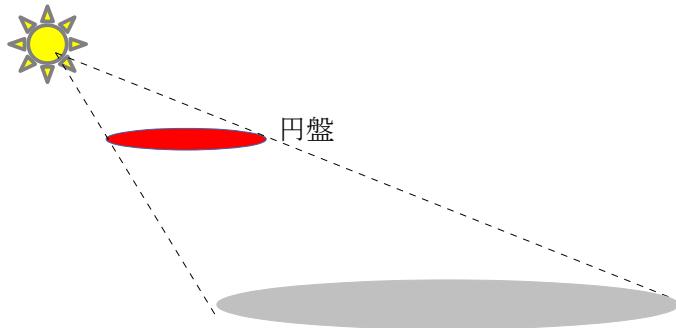
`XMMatrixOrthographicLH`

で取得すれば平行投影になりますので、そちらを使って下さい。うーん平行投影の場合めっさかっこわるくなるのでおすすめしませんが…

次にまた違った影の表現をお話いたしましょ。これも「考えたやつ頭いいなあ」って思えるものです。それは…シャドウボリュームもしくはステンシルシャドウと呼ばれるものです。

シャドウボリューム(ステンシルシャドウ)

はい、最初に原理を言っておきましょうか……これはスゴイ。頭いいよ。例えば円盤が浮かんでいると思って図を見てくれ…



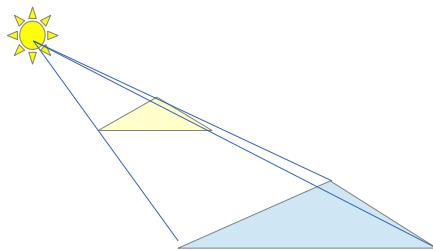
そうすると影は図のような落ち方をしますよね？で、今回紹介するステンシルシャドウってやつは、前回のシャドウマップよりもうちょっと直接的なことをする。つまりどういうことなのかというと…

まず

光源から見た物体の面を引き伸ばします。つまり

三角形を引き伸ばせば錐台の形（三角すいのてっぺんをぶった切ったようなやつたち）になりますが、こいつを利用します。

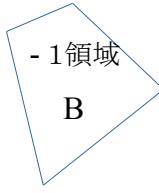
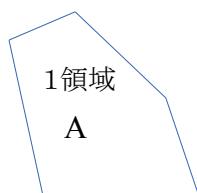
なお、この「引き伸ばした立体」をシャドウボリュームと云います。



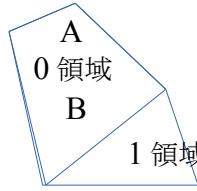
で、とりあえず「どうやって引き伸ばすか」はいったん脇に置いておいて、原理を先に説明しいましょう。上の図では錐台のシャドウボリュームができていますが、実はこれポリゴンで作成します。（えっ？そんなことしたら重いでしょって疑問はちょっと脇においとして）

で、ここで重要な話です。こつからどうやったら影になるのか？

このボリュームの、カメラから見た表側の面を+1とし、このボリュームの裏側の面を-1としてそれぞれレンダリングする。そうすると



表面に映るのはA。裏面に映るのはBであります。そしてこの2つを加算合成すると…



図のように「0領域」と「1領域」に分かれます。0は無視すればいいので、1の部分だけが影になります。これにより影が落ちることになります。

原理はなんとなくわかったんじゃないかな?頑張りなあって思いますわ。まさか表を1、裏を-1として足しちゃおうなんて…思いつかねえよ。

で、肝心の「どうやって引き伸ばすか」なんですが、こんなの頂点シェーダにもピクセルシェーダにもやらせられない…。どうしましようか。

…かと言ってフリーに引き伸ばし計算をするととてもじゃねーけど重そうだ。

※実際重い…このため、現在のゲーム業界ではシャドウマップのほうが優勢ではあると思われる……が!!ジオメトリシェーダだのテッセレータだのをいじれるようになってきたため、この状況も変わりつつあります。

というわけで来たよ…ジオメトリシェーダ。

これはDX9だけやってた人にとっては「なにそれおいしいの?」ってところなんだけど…というか、逆に考えるとMMDのシャドウ技法はシャドウマップであることが分かりますね。なんでかつて、普及してるMMDはDX9版ですし、シャドウに関してはセルフシャドウをONにできることから、シャドウマップであることが予想されます。ステンシルシャドウ人気無いですね。

まあ、人気がない理由のモウヒツは、ポリゴンごとの演算であるため、テクスチャにアルファがある場合、影が正しく出ないってのがあるんだよね。つまり樹木とかのモデルを抜きありビルボードで作っている場合は~~抜きは無視されるため~~なんか樹が…特に葉っぱの影がショッボイことになるんやね…。使えないなあ。考えれば考える程使えねえ…。

※ちなみに余談ですがMMDの影に関して、恐らくD3DXMatrixShadowとシャドウマップの合わせ技でやってるようですね。床がない状態でもシャドウは落ちるし「セルフ影」をオフにすると背景に影が落ちませんしね。

閑話休題、シャドウボリュームですが、きちんと自学しての人なら見たことがあると思うが、こういう画像。



この緑の部分がシャドウボリュームってわけです。昔はこいつが重くて使い物にならなかつたっぽいですね。こいつはDirectXサンプルブラウザのShadowVolumeにあります。なお、こいつはDX10時代のサンプルなんで、DX11とはコードが変わってしまうとは思います。

なお、DX11時代のシャドウはより高度でより綺麗なので『できる皆さん』はそっちに挑戦して下さい。『できる皆さん』はVarianceShadowに挑戦してね。

こいつもサンプルのVarianceShadows11にサンプルが有ります。なお、こいつはカスケードシャドウマップのハッテン系なので先にカスケードシャドウマップからやったほうがいいかもしれませんね。

ちなみにこいつはヘキサのHPでも公開されているテクニックです。

<http://hexadrive.jp/index.php?id=67>

やってやれないことはない。頑張りましょう。ジオメトリシェーダとは

ところで、ステンシルシャドウに入る前に説明すべきジオメトリシェーダについてですが、こいつはDX10のときはフィーバーしてたんだけどDX11に入ると…廃れつつあるシェーダなんだよねえ…まあ知つとかなイカン知識ではあると思うんで解説すると…

ヒトコトで言うと『プリミティブ(三角ポリゴン)単位で色々とイジれるシェーダ』、ちゅうことやね。だからOpenGL界隈では『プリミティブシェーダ』とも呼ばれます。

で位置的には

今まで

頂点シェーダ→ラスタライザ→ピクセルシェーダ

というイメージだったと思いますが、こここの間、頂点シェーダの直後くらいに入り込むイメージです。

つまり

頂点シェーダ→ジオメトリシェーダ→ラスタライザ→ピクセルシェーダ

という感じです。ドゥンドゥン多くなってきてうんざりしてきますか?まあしゃーないです。これから君たちに求められるモノはドゥンドゥン跳ね上がってきます。

まあ、増えたとはいっても要はですね今までCPUでさせてた演算をGPUで行うことによつてグラフィックス処理を速くしていくっていう流れなわけね。最近はOpenCL(OpenGLじゃないよ)なんていう便利なものも出てきてとにかく、CPU(もちろんマルチスレッド)とGPUを

うんま~いこと使って早くしようっていうご時世なのよ。

いくらPS4やらXBoxOneの性能が上がってマシンそのものが速くなつたように見えて、それ以上に「ソフトウェアを速くする努力」が必要になつてる時代なんだねー。

ともかくジオメトリシェーダってのはそういう流れの中から生まれたもので、先程も言ったようにプリミティブ単位での処理(三角形をいっぺんに処理)が可能になってくるわけ。

で、このジオメトリシェーダ、何がウレシイのかというと、シェーダ上でプリミティブ(ポリゴン)を増やせちゃうわけ。今までCPU側でしか増やせなかつたものが、GPUで増やせちゃうということは、処理速度をある程度保つたまま…いろんな処理ができちゃうわけよ。

で、早速ジオメトリシェーダを組み込んでいこうぜって話なんですが、あ、悪いけどこのシェーダDX10以降用なんだ。



というわけで、ここ以降の話ってのはDX9でゲーム作ってる人は実現できない。スマナイ。

ちなみにDX11でジオメトリシェーダを今までのようく使うには

ジオメトリシェーダ関数作つとして

D3DX11CompileFromFileでコンパイルして

CreateGeometryShaderでシェーダ作つて

GSSetShaderでセットする

このいつもの流れですたい。

で、肝心のHLSLにおけるジオメトリシェーダの作り方ですが、基本的な関数の作り方はピクセルシェーダとか頂点シェーダと変わりません。…なんですが、やっぱ違う部分があります。何かと言うと、前にも話しましたが、ジオメトリシェーダによって、ポリゴン数を増やしたりすることができます。

できるのですが、予めそれ(頂点数)の最大数を指定しておく必要があります。

これを指定するのがmaxvertexcountというキーワードで

(**maxvertexcount**(最大長点数))

```
void ジオメトリシェーダ名(引数/パラメータ, inout 出力ストリーム){  
}
```

てな感じで使用します。

[http://msdn.microsoft.com/ja-jp/library/ee418313\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee418313(v=vs.85).aspx)

ちなみにレジスタはこんな感じ…

[http://msdn.microsoft.com/ja-jp/library/ee418367\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee418367(v=vs.85).aspx)

ひとまずはナニモシナイジオメトリシェーダをつくりましょう。で、上の例を見てわかると思いますが、今までの頂点シェーダやピクセルシェーダの時とは違い、ジオメトリシェーダには戻り値の型がございません。

voidなんつすよ…じつはんじやあジオメトリシェーダで作った情報はどうやって返すのか？それはですね…定義をよく見て下さい。**inout**というのがあるでしょう？

この「出力ストリーム」ってのは入力であり、出力でもあるわけです。概念としては難しく感じる人もいると思いますが…。

まあ、とりあえず何もしないってのを作りましょう。

ナニモシナイので、maxvertexcountは3つにしておきましょう。では…元々作っていたBasicVSに対応するような形で作ってみましょう。頂点はV型で定義してたので…

```
void GTest(triangle V vin[3], inout TriangleStream<V> outstream){
```

```
}
```

こんな感じで。すごく…なにもしません。ちなみにtriangleはプリミティブ型を表しています。

[http://msdn.microsoft.com/ja-jp/library/ee418313\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee418313(v=vs.85).aspx)

さらにTriangleStreamとは、ストリーム出力オブジェクトの型を表します。

[http://msdn.microsoft.com/ja-jp/library/ee418375\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee418375(v=vs.85).aspx)

本気でなにもしないものを作りましたので、C++側でD3DX11CompileFromFileがうまくいくかどうか試しましょう。S_OKが返ってきましたか？

あ…いい忘れてましたが、シェーダリージョンの文字列は“gs_4_0”もしくは“gs_5_0”にしておいてください。

おっと、もうひとつ…出力ストリームを再起動する必要がありますのでジオメトリシェーダの最後に

```
outstream.RestartStrip();
```

と一行追加しておいてください。

[http://msdn.microsoft.com/ja-jp/library/ee418374\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee418374(v=vs.85).aspx)

S_OK返つきましたか?

ではCreateGeometryShader しましょう。

CreateGeometryShader(コンパイル済みポインタ、サイズ、NULL、NULL、ジオメトリシェーダ)
と言った感じで、頂点シェーダの時やピクセルシェーダの時と同じです。

作れましたか?

ではいよいよジオメトリシェーダのセットです。ちなみに **BasicVS** に対応するところにだけ適用して、それ以外の部分はNULLをセットするようにして下さい。

さて…どうなるでしょうか?見てください。



はい、ものの見事にミクちゃん消えました。

なんですかというと、**ガチで何もしていない**からです。というわけで、何かしてあげましょう。
RestartStrip()だけでは足りないということですね。なんですかというと outstream に何も入っていないからです。きちんと**追加してあげましょう**。

もう一度ここを見てみましょう。

[http://msdn.microsoft.com/ja-jp/library/ee418375\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee418375(v=vs.85).aspx)

そうすると、それっぽいのがありますよね? Append 関数

[http://msdn.microsoft.com/ja-jp/library/ee418373\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee418373(v=vs.85).aspx)

これを使いましょう。では、パラメータで渡された3頂点をそのまま追加してみましょう。

outstream.Append(v[0]);

outstream.Append(v[1]);

outstream.Append(v[2]);

てな感じで…どうでしょうか?ミクちゃん復活したんじゃないでしょうか?

おめでとう! レベルが一個上がつて君はジオメトリシェーダを使えるようになったよ!!!

とはいってこのままで面白くないのでチョイとポリゴンを増やしてみましょうか…ポリゴン数を2倍にしてみましょう。

```

//元のポリゴン
outstream.Append(vin(0));
outstream.Append(vin(1));
outstream.Append(vin(2));
//横にコピー
for(int i=0;i<3;++i){
    vin(i).pos.x=vin(i).pos.x+1;
}
outstream.Append(vin(0));
outstream.Append(vin(1));
outstream.Append(vin(2));

```

それぞれのx座標に1を足してみました。



すごく…ふとましいです。

ちなみに射影変換済みのポリゴンに対して処理を行っているので、どの方向からみてもふとましくなります。

さて、しかしシャドウボリュームを作るためには射影変換後の状態では困ります。モデル自身の回転やボーンの状況は更新して欲しい所ですが、それ以外の変換はちょっとまっていただきたいですね。

ということで、現在一緒にしているかもしれないWVPはWorldとViewProjに分けておいて下さい。

で、Worldおよびボーンの行列はVSで計算してもらって構いませんが、ViewProjはジオメトリシェーダの中で行って下さい。

で、元ポリゴンの部分はこんな感じになるでしょう。

```

//元ポリゴン
for(int i=0;i<3;++i){
    V v=vin(i);
    v.pos=mul(viewproj,vin(i).pos);
    outstream.Append(v);
}

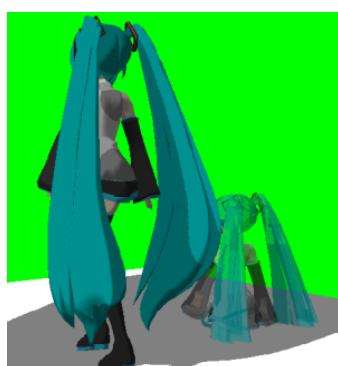
```

}

で、この後でそのまま

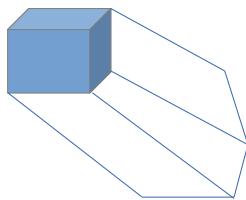
```
//ズラし描画
for(int i=0;i<3;++i){
    V v=vin(i);
    v.pos.z+=10;
    v.diffuse.a=0.5;
    v.pos=mul(viewproj,v.pos);
    outstream.Append(v);
}
```

なんて書くとポリゴン同士がつながってカッコ悪いので、元のポリゴンを描画した後で RestartStrip しといてください。で、適当にミクちゃんズらしておくと



ミクちゃんのコピーが影方向に描画されます。あとは、元のミクちゃんからコピーミクちゃんの位置に引き伸ばしてあげましょう。

さて…本番はここからです。元の位置からそれぞれ引き伸ばすのが目的です。そして引き伸ばした結果をポリゴンになるようにする必要があります。つまり…

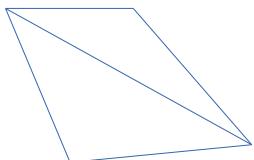


図のように影が落ちるとすると、それぞれの辺から四角形ポリゴンが伸びていくことがわかります。四角形ポリゴンは三角形ポリゴンふたつなので、それなりにポリゴンが増えますが、まあせいぜいが7倍程度です。

☆ポリゴンは7倍だけど、頂点数は $18+3$ で 21 倍になるので注意。

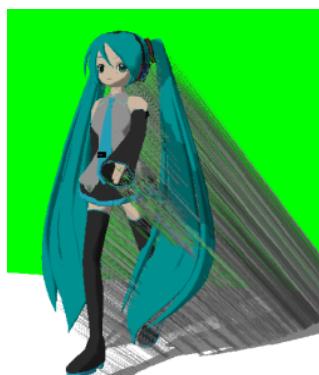
最終的にはこつから要らないものを削っていくんですが、とにかく引き伸ばしてみましょう。

引き伸ばすには、それぞれのポリゴンを構成する辺をそのまま…そうですね。光源のベクトル方向に30伸ばしてみましょうか。



というわけで三角形であれば0-1から伸ばす、1-2から伸ばす、2-3から伸ばしたものを作ってやります。

そうやって単純に引き伸ばしてやれば



こんな具合にシャドウボリュームを発生させられます。この「シャドウボリューム」を元にステンシル値によって影を描画すればいいのです。

その前に、不必要なポリゴンを削減しておきます。どこが不必要でしょうか？これは普通にシェーディングする際に、法線ベクトルとライトベクトルから描画すべきか、そうでないか決めてましたよね？

法線方向と、ポリゴンからライトへ向かうベクトルの内積で計算してました。ということは

その逆を行えばいい。とにかく光の方向と法線方向の内積がプラスであれば引き伸ばすようにします。これでだいぶ減らせるでしょう。

ここまででやっとステンシル値を使用して影を落とすのが落とさないのかを決定する準備ができました。

Stencil/バッファとは

深度バッファの時にも名前が出てきましたStencil/バッファ。…そもそも stencilって英語でどういう意味なのかというと

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

型板ってな意味ですね。

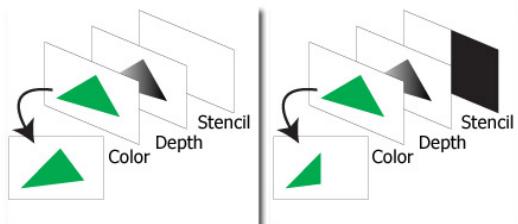


こんな感じのアートに使われている…美術部経験者とかだと知ってると思いますが、絵の形に切り抜いた紙の上からブツシューっとエアブラシかけたり、あとはインク流しこんだりして独特の表現をするものです。

で、その切り抜いた型紙やら鉄板やらを英語で stencilって言うんだわ。



こういうやつね。小学校の時見たことがある人もいるでしょう？はい、まさに今回使用する「Stencil」はこういう「型」を表現するためのバッファなんDA!!



図のように、Stencil値が0の部分は完璧に抜かれてしまいます。前述したようにこれを利用するわけです。

既に分かっているとは思いますが、例によってまたテクスチャ作成→ビュー作成ってな流れになっていくんですが、今回活躍するのは「**Stencilステート**」です。細かく書き込む情報のルールやら演算を弄くり回しますので、こいつが役に立つのさ。

で、早速例によってStencil用テクスチャを作っていく…と言いたいトコロですが、その必要はないわ。

既に使っているであろう「**デプスStencilバッファ**」名前をよく見てみよう。デプス…Stencilである。つまりデプスにもStencilにも使えるわけである。ただし現状ではデプ

スに絶力をつぎ込んでいるため、

`DXGI_FORMAT_D32_FLOAT`

という指定になっているであろう…。これをチョットだけ…ほんのチョットだけステンシル演算に使わせてもらうのだ…ほんの8ビット…。

というわけでまずは

`DXGI_FORMAT_D24_UNORM_S8_UINT`

にします。見てなんとなく想像就くと思いますが、デプスに24ビット、ステンシルに8ビット使用します。どのみちステンシルは型紙データなので、8ビット(1バイト)で十分ですね。ただこの精度は少しだけ落ちます。落ちますが…まあカメラから見たエッジ用分解能なんてそんなにいらんでしょう…。

あと、UNORMについてね、前はなんかごまかしてましたが、

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

の一番下に書いてますね。まあ…そういうことですわ。わがんねーひとは個人的に聞いてください…わがんね一人が多かつたら別枠でまたお話ししますが…。

じゃあ、手順です。

- ①まずはD32→D24S8
- ②ステンシルステートを新しく2つ作成する(書き込み用と読み込み用)
- ③普通に描画(本体を書く)
- ④実際の描画時に元のステンシルステートをバックアップ
- ⑤書き込み用ステンシルステートにエンジ
- ⑥シャドウボリューム描画(ステンシル計算のためだけの描画パスなんで、全部透明にして)
- ⑦読み込み用ステンシルステートにエンジ
- ⑧R=0,G=0,B=0,A=0.5くらいのポリゴンを描画(切り抜かれている…ペロッこれは影!)
- ⑨ステートを④のバックアップに戻す

とりあえず①から…

`DXGI_FORMAT_D24_UNORM_S8_UINT`

にした上で挙動が変わらないことを確認して下さい。…変わりますか?

②ですが、…もうそろそろ直面している問題に対するググール先生の利用法も身についてきたのではないでしょうか…?

恐らくステンシルステート作れってことは“Create” “Stencil” “State” “DX11”で検索す

ればなんとかなるんじゃね?と

[https://www.google.co.jp/search?q/Create+Stencil+State+DX11&oq/Create+Stencil+State+DX11&sourceid=chrome&espv=210&es_sm=122&ie=UTF-8](https://www.google.co.jp/search?q>Create+Stencil+State+DX11&oq/Create+Stencil+State+DX11&sourceid=chrome&espv=210&es_sm=122&ie=UTF-8)

あー、英語ばっかし…検索ワード間違えたと思ったら、あとはカンで色々と組み合わせてみる。この労力を怠ればカンは育たない。

D3D11 DepthStencilStateで検索しよう

[https://www.google.co.jp/search?q/Create+Stencil+State+DX11&oq/Create+Stencil+State+DX11&sourceid=chrome&espv=210&es_sm=122&ie=UTF-8#es_sm=122&espv=210&q=d3d11%20depthstencilstate&safe=off](https://www.google.co.jp/search?q>Create+Stencil+State+DX11&oq/Create+Stencil+State+DX11&sourceid=chrome&espv=210&es_sm=122&ie=UTF-8#es_sm=122&espv=210&q=d3d11%20depthstencilstate&safe=off)

きた…ID3D11Device::CreateDepthStencilStateだ…

[http://msdn.microsoft.com/ja-jp/library/ee419789\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419789(v=vs.85).aspx)

悪いが俺だって一発で正解を見つけてるわけじゃない。何度も何度も失敗してるんだ…。でこいつも別で開いておいたほうがいいね。

[http://msdn.microsoft.com/ja-jp/library/ee416082\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416082(v=vs.85).aspx)

さて、どう設定しましようかね…。

とりあえずは

DepthEnable=true;

DepthFunc=D3D11_COMPARISON_LESS;

DepthWriteMask=D3D11_DEPTH_WRITE_MASK_ALL;

StencilEnable=true;

StencilWriteMask=D3D11_DEFAULT_STENCIL_WRITE_MASK;

StencilReadMask=D3D11_DEFAULT_STENCIL_READ_MASK;

ステンシル値は表面、裏面共に見なければいけないのですが、BackとFrontのそれぞれの設定ができます。

で、なんとかOpってな部分の指定はここを見て

[http://msdn.microsoft.com/ja-jp/library/ee416283\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416283(v=vs.85).aspx)

なんとかFuncの指定は

[http://msdn.microsoft.com/ja-jp/library/ee416063\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416063(v=vs.85).aspx)

を見ましょう。

で、表面、裏面双方カンケーなく書き込みは行うので、Funcの値は

D3D11_COMPARISON_ALWAYS

にしましょう。これは一つでもテスト合格…言い換えると素通りでステンシル値の書き込

みが行われます。

で、次に他のパラメータなんですがそれぞれ

StencilFailOp

ステンシルテストで不合格となったときに実行するステンシル処理です。

StencilDepthFailOp

ステンシルテストに合格し、深度テストで不合格となったときに実行するステンシル処理です。

StencilPassOp

ステンシルテストと深度テストの両方に合格したときに実行するステンシル処理です。

となっています。で、それぞれ何を指定するのかというと、

D3D11_STENCIL_OP_KEEP

既存のステンシルデータを保持します。

D3D11_STENCIL_OP_ZERO

ステンシルデータを0に設定します。

D3D11_STENCIL_OP_REPLACE

ステンシルデータを、ID3D11DeviceContext::OMSetDepthStencilState を呼び出して設定された参照値に設定します。

D3D11_STENCIL_OP_INCR_SAT

ステンシルの値を1増やし、その結果をクランプします。

D3D11_STENCIL_OP_DECR_SAT

ステンシルの値を1減らし、その結果をクランプします。

D3D11_STENCIL_OP_INVERT

ステンシルデータを反転します。

D3D11_STENCIL_OP_INCR

ステンシルの値を1増やし、必要に応じて結果をラップします。

D3D11_STENCIL_OP_DECR

ステンシルの値を1減らし、必要に応じて結果をラップします。(くっそ WWW 誤字 WWW)

へえへえ、それではひとつずつ考えていきましょうかね。上のみつつの指定からいって、ステンシルテスト→深度テストってな具合で流れがあるっぽいですね。

では、StencilFailOp から考えましょう。不合格だったらどうすんの？ うん、ナニモシナイ。なにもしないのは D3D11_STENCIL_OP_KEEP です。現状維持です。ていうか今回って不合格ないしね。

次に StencilDepthState ですが、これデブスで不合格な場合…どうするんでしようか。こいつは通してはいけません。通すとシャドウボリュームの意味がなくなります。なので、これも D3D11_STENCIL_OP_KEEP にしておきましょう。

最後にStencilPassOpですが、両方に合格した場合。裏面(Back)であれば"-1"にして、表面(Front)であれば"+1"になるように考えましょう。で、それっぽいのないかな~。

あつ、これが。

D3D11_STENCIL_OP_INCR

ステンシルの値を1増やし、必要に応じて結果をラップします。

D3D11_STENCIL_OP_DECR

ステンシルの値を1減らし、必要に応じて結果をラップします。(ひでえ誤字)

ということで

Back.StencilPassOp=D3D11_STENCIL_OP_DECR;

だし、

Front.StencilPassOp=D3D11_STENCIL_OP_INCR;

…余談だけど、ここ、ヒドイ誤字がありますね。Decrementをなぜ「増やす」と訳するのか…。

[http://msdn.microsoft.com/en-us/library/windows/desktop/ff476219\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff476219(v=vs.85).aspx)

というわけで、それぞれ設定して下さい。

これで、ステンシル値書き込みの時のステートは設定出来ました。

次にモウヒトツ作るべきものがあります。それは上の計算結果によって切り抜くためのステートです。

最初に作ったのとほぼ一緒ですが、計算が違います。今回はステンシル値のみを見て、それが0なのかどうかで判断します。なので、次のパラメータは

```
dc.FrontFace.StencilFunc=D3D11_COMPARISON_NOT_EQUAL;//ステンシル値が0以外を通す  
FrontFace.StencilPassOp=D3D11_STENCIL_OP_KEEP;//通った時はそのまま書き込み  
FrontFace.StencilDepthFailOp=D3D11_STENCIL_OP_KEEP;//ステンシルに通ったらそのまま書き込む  
FrontFace.StencilFailOp=D3D11_STENCIL_OP_ZERO;//ステンシル失敗したらとにかくゼロ
```

これ、背面も同じね。と言った具合に2つ作っておく。とりあえず"言っておくと、今回設定したこのステート、正解ではない。"が、とりあえずどういう効果が出るか見てみましょう。

まあ、2番めのやつは逆さまの意味にしてもいいけどね? EQUALにして、通ったらゼロ。通らなかつたらキープね。

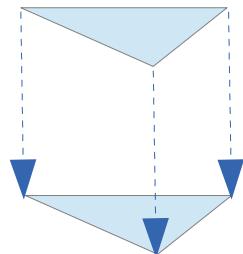
これで切り抜き型ができるので、あとは大きな半透明ペラ紙を持ってきてペロッとはりつけねばオッケー(^ω^)

で、ここまで注意点なんですが…あのジオメトリシェーダで引き伸ばす部分なんですけ

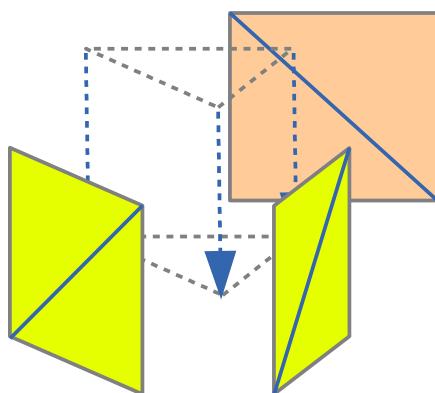


どね？

こういうやつを引き伸ばすとするじゃん？



こうやん？



で、実際はこういうこと（ポリゴンが6つでボリュームになってる）やん？結局はさ。で、この時、初歩的なことなんだけど気をつけて欲しいのがオモテとウラね。もし、全ての面の作成時にこの字になるように作ってしまうとおかしなことになる。くれぐれも気をつけてください。

さて、ここまでがマトモにできてたとしても…たぶん、



こうなるのではないかね？これは罠だ!!これはMSが僕を陥れるために仕組んだ罠だ!!どういう事が」というと、ソリッドファを有效地にしているため、背面ポリゴンのステンシルが描画されずに、残っているというわけだ。

これをどうにかするにはもう少しだけデプスソリッドファについて理解を深めねばならない。どういう事が」というと、これを見た瞬間「じゃあデプス無効にすりゃいいんじゃない？」って思う人はちょっと短絡だと思います。

例えばシャドウボリューム書き込みの際にデプスを無効にしたとします。そうすると地面よりボリュームは後に書かれるため、地面に落ちるはずの影が地面に落ちなくなります。つまり地面だの障害物だのかんけーねー状態になります。

…つまり、影が落ちません。全部〇になりますからね。

もし、デプスが有効か無効かという2択なら、この解決法はしんどいことになるんだろうけど…そうではなく、**デプスは有効…かつ書き込みはしない!**…というものを作りさえすればいいわけ。そしてそれは作れるわけ。とりあえずデプス有効のまま

```
DepthWriteMask=D3D11_DEPTH_WRITE_MASK_ZERO;
```

としてください。

[http://msdn.microsoft.com/ja-jp/library/ee416087\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416087(v=vs.85).aspx)

を見ればわかると思いますが、

D3D11_DEPTH_WRITE_MASK_ZERO

深度ステンシル/ソリッドファへの書き込みをオフにします。

てな記述がされていると思います。そう、これこそが求めていたもの。デプスは有効かつ書

き込みはしない」という操作が可能になります。

結果として、シャドウボリュームを描画する際は地面の向こうのシャドウボリュームは地面の上に遮られ描画非対象となるが、シャドウボリューム自身は書き込みを行わないため、「前面によって背面が描画されない」などという現象はなくなります。

こうすれば結果として



このように「影?」らしものがでます。なんでこんな状態になっているのかというと、ミクちゃんの9000ポリゴンからそのまま引き延ばしていますので、オモテウラの状態が非常にアレになっているためです。

まあそもそも9000ポリゴン×18なんて、トンでもない数になっちゃってるんですよ。これは使えないわけ。なので、ステンシルシャドウに必要な分だけシャドウボリュームを削る必要があるわけです。

で、そのためのヒントが

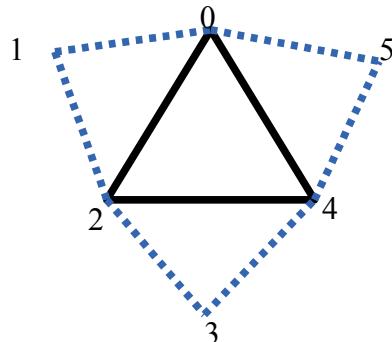
[http://msdn.microsoft.com/ja-jp/library/ee416427\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416427(v=vs.85).aspx)

に書いてあります。**隣接頂点プリミティブ**がどうとか書いていますね。次回はこの辺をいじってマトモなスピードでマトモな影を表示できるようにしてみましょう。

まず、この隣接頂点プリミティブってのがなんなのかお話ししましょう。

隣接頂点プリミティブ

これはなんなのか？簡単に言うと、あるポリゴンを構成する三角形のお隣さんの頂点の情報まで含むプリミティブってこと。通常だと3点の三角形なんだけれど、これにさらに隣接する3点を加えて6点になっているのが「隣接頂点プリミティブ」なのだ。



この状態にするためにはジオメトリシェーダのtriangleって書いてある部分がありますよね？それをtriangleadjにするだけです。そうした場合6頂点になります。

例のマニュアルにはこの順序で書かれてるんですが…なんか違う気がします。

とはいっても、色々と資料を見ても、元の三角形が偶数で0, 2, 4で、隣接頂点が1, 3, 5と書いてあります。…なんだかなー。ほんとかなーと思いつつ…

[http://msdn.microsoft.com/en-us/library/windows/desktop/bb509609\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb509609(v=vs.85).aspx)

ここを見ても、偶数が元のポリゴン頂点…奇数が隣接頂点なんだよね。

…どうやっても0, 1, 2で正しいポリゴンが出ますね。

とか思ってたら…あれか、隣接頂点情報って自分で入れてあげなきゃいけなかったのか。

くっそ WWW やってらんねえ WWW

すみませんね。

じゃあ、どうやって、通常のポリゴン情報を隣接アリのポリゴン情報に変更するのか。
自分でやんだよ…どうやって？

隣接頂点ってのは、隣のポリゴンなわけ。隣のポリゴンとは何を共有している？

1辺を共有しているわけ。ということは三頂点のうち二頂点を共有しているポリゴンが隣接ポリゴン。

で、共有していない頂点が隣接頂点なわけ。元々インデックスは3つ一組です。なので3つおきに調査して2頂点を共有しているのを見つけたら、残りの頂点が隣接でオッケー。

まずは愚直に探しましょうか…ところでミクちゃんデータはインデックス数が44991です。

…多いですね。仕方ないですね。そして隣接データを付加すると二倍になって…89982です。
泣きそうですね。

でもやるのだ。仕方ないのだ。

で、インデックスデータは2byte整数で並んでいますから、こいつを3つ一組で見ていくわけです。

```
(0AE8 0AE9 0AEA) (0AE8 0AEA 0AEB) (0AEB 0AEC 0AED)  
(0AEB 0AED 0AE8) (0AEE 0AEF 0AED) (0AEE 0AED 0AEC)
```

で、最初の2つを見て下さい。0AE8と0AEAを共有しています。つまり、1番目のポリゴンにおける隣接頂点インデックスは0AE9だということがわかるわけです。ちなみに4ポリゴン目は0AE8を共有しては居ますが、1点ですので、隣接ポリゴンではありません。

こういったルールで隣接頂点を探していきます。で、これを隣接インデックスデータとして作っていくわけですが、例えば

```
std::vector<unsigned short> adjindices;  
for(中略){  
    adjindices.push_back(元ポリ頂点0);  
    adjindices.push_back(隣接頂点0);  
    adjindices.push_back(元ポリ頂点1);  
    adjindices.push_back(隣接頂点1);  
    adjindices.push_back(元ポリ頂点2);  
    adjindices.push_back(隣接頂点2);  
}
```

こんな感じで頂点増やしていくわけですよ。うん、で、このコードのvectorの一番コストかかるのって何だと思いますか？ここまで読めている人にはわかると思いますが、push_backなんですね。

push_backの中で何をやっているのかというと、push_backするたびに確保しているメモリをチェックして、確保領域を超えそうならもっと大きい領域を確保してるわけ。

ということはそこで新たにmalloc(もしくはnew)とfree(もしくはdelete)が発生してるわけですよ。メモリの確保と解放はC/C++においてかなりコストがかかる。この辺のこと理解できるかなあ…？

例えばpush_backで1バイト確保して、次のpush_backで1バイト解放して2バイト確保するってな自体になるわけですよ(実際は10個分ずつ確保するような最適化があるらしいで

すが…)

どのみち最終的に89982確保するわけですから……最初つから確保しておいたほうがいいですよね。

となると、みんな知ってるresize関数で最初から配列状態にしておくってのもひとつの手です。

だけどpush_backを使いたいって場合もあると思います。そういう場合に使うのが、**reserve**関数です。

こいつは、将来使用するかもしれないメモリを予め確保しておくものです。確保はしていますが、たんなるメモリの塊にしかなっていないので、配列状態にはなっていません。これをやっておけばpush_backの時のコストは変数のコピーコストだけになります。ですから先程の式を書き換えるならば

```
std::vector<unsigned short> adjindices;  
adjindices.reserve(indicies.size()*2);  
for(中略){  
    adjindices.push_back(元ポリ頂点0);  
    adjindices.push_back(隣接頂点0);  
    adjindices.push_back(元ポリ頂点1);  
    adjindices.push_back(隣接頂点1);  
    adjindices.push_back(元ポリ頂点2);  
    adjindices.push_back(隣接頂点2);  
}
```

みたいになるわけです。まあどっち使っても結果は一緒なんで好きな方を使いましょう。
reserve→push_backのほうが添字のことを考えなくていいので、僕は好きですね。

話がそれましたが、そんな感じです。あ、くれぐれも言っておきますが、元のインデックスデータはマテリアル反映の時に重要な役割を持っているので、捨てないでくださいね。ですから、これをやろうとすると、インデックスデータは合計3倍になるわけで270KBくらいになります。まあ…大したことないとか。

さて、最初に言っておきますが、最初は最適化なんてしません。なので、相當に重くなると思いますが、ま…いいでしょ。

で、検索方法ですが、もうね、自分以外の3組みーんな調べましょう。うん、きつついけど頑張りましょう。

```
for(インデックス数*2 i+=3){  
    for(インデックス数*2 j+=3){
```

```

if(i!=j){
    if(2頂点が一致){
        隣接追加
    }
}
}

```

こんな感じですね。まあ…45000くらいのインデックス2重ループなんてクソ重いです。
あとこいつはトポロジがTRIANGLE_LISTでなくTRIANGLEADJ_LISTなので、それも注意しておきましょう。

具体的に言うとこういう関数作ります。

```

///共有頂点情報をビットで返す
BYTE VertexesIsDuplicated(const unsigned short* left,const unsigned short* right){
    int dupCnt=0;
    BYTE ret=0;
    for(int i=0;i<3;++i){
        for(int j=0;j<3;++j){
            if(left(i)==right(j)){
                ++dupCnt;
                ret|=(1<<(i+3)|(1<<j));
            }
        }
        if(dupCnt==2){
            return ret;
        }
    }
    return ret;
}

///隣接頂点情報作成
void CreateAdjacentIndices(const std::vector<unsigned short>& original,std::vector<unsigned short>& adj){
    size_t sz=original.size();
    adj.resize(sz*2);
    for(int i=0;i<sz;i+=3){
        adj(i*2)=original(i);
        adj(i*2+2)=original(i+1);
        adj(i*2+4)=original(i+2);
    }
}

```

```

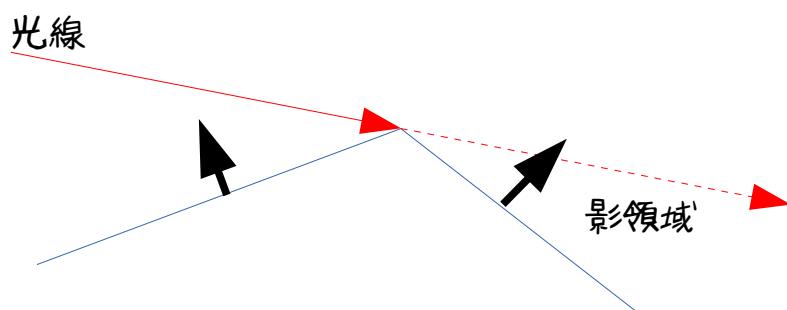
for(int j=0;j<sz;j+=3){
    if(i==j){
        continue;
    }
    BYTE dup=VertexesIsDuplicated(&original(i),&original(j));
    if(dup>24){ //24ってのは2^3+2^4のことね。
        int idx=(dup&1)==0?0:((dup&2)==0?1:2);
        int adj_idx=(dup&8)==0?0:((dup&16)==0?1:2);
        adj(i*2+(2*adj_idx+3)%6)=original(j+idx);
    }
}
}

```

コード見て分かるように馬鹿正直なコード書いてるんでクソ重いです。まあ、最適化なんてものは後回しでいいです。

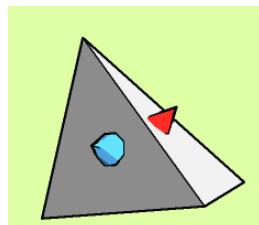
いよいよこれで隣接頂点情報が渡りましたので、ここから如何にしてこいつを使っていくのかという話をしましょう。

いよいよ隣接頂点を使うわけなんだが、どうやってつかうのだろうか？それは



こういうイメージを持っておくわけ。そうするとさ、光源から見たエッジ部分ってのはどうなってるのか？どの情報から「エッジ」ってわかるのか？そうですね。法線見てください。

法線が光線と相対する向きを向いている部分→光線に沿う部分と変化する時に影が発生するわけです。



ということは…もうわかりますね。

①対象となる三角形が光源側を向いているか判定

- ②隣接頂点の法線が光源方向を向いていないか判定
- ③①かつ②が成り立つのであればその辺は光源から見たエッジ(輪郭線)と言っていい。

うおーーーーー!!!準備はああああああああ!!!できいれれれれれいたあああああああ!!!!いざあああああ!!!シャドオオオオオオオオ!!!!ヴォリュウウウウウムゥウウウウウウ!!!!

で、まあ、頂点法線でなく面法線でやると思いますんで、とりあえずは三頂点足しちゃって正規化しちましょ。それで面法線になります。

```
面法線=normalize(頂点1.norm+頂点2.norm+頂点3.norm);
```

こんな感じですね。これで、中心ポリゴンと他の3ポリゴンを調査していきます。



ここまでやれば、シャドウボリュームにより影が落ちたように見えることでしょう。ちなみにシャドウボリュームはこんな感じです。影に関しての基礎的な部分はこのくらいです。これ以降の部分はMSのサンプル見るなりして進めていきましょう。

最後に、僕の体験からお話をすると、この手の面倒な処理は一度**プリミティブな立方体だの円柱だの、なんかしら分かりやすい図形でテストしてみることをお勧めします。**何故かというと9000頂点のようなデータになるとどこに間違いがあるのかを推測しづらいからです。

まさに「**急がばまわれ**」です。色々と焦る時期かもしれません、落ち着いていきましょう(と自分に言い聞かせる)。

フェイシャルモーション

さて…せっかくみなさん「俺の嫁」を表示してもらいましたが、無表情ですね。これはいいただけない。無表情なのはいいただけませんね。美しくない。



MMD モデルはやはり表情あってのものだと思います。

```
header.VmdmodelName[0] 94 8E 97 ED 97 EC 96 B2 00 FD FD FD FD FD FD FD FD 博麗靈夢 .....
header.VmdmodelName[16] FD FD FD FD .....  

motion_count 00000000  

vmd_skin_count 00000011  

skin[0].SkinName[0] 82 DC 82 CE 82 BD 82 AB 00 FD FD FD FD FD FD まばたき .....
skin[0].FlameNo 00000000  

skin[0].Weight 00000000  

skin[1].SkinName[0] 82 ED 82 E7 82 A2 00 FD FD FD FD FD FD FD わらい .....
skin[1].FlameNo 00000000  

skin[1].Weight 00000000  

skin[2].SkinName[0] 94 BC 96 DA 00 FD FD FD FD FD FD FD FD 半目 .....
skin[2].FlameNo 00000000  

skin[2].Weight 00000000
```

VMD が吐き出すデータはこんな感じですね。上の靈夢さんの表情だとどこが変わっているのかというと

```
skin[9].SkinName[0] 82 C9 82 E2 82 E8 32 00 FD FD FD FD FD FD FD にやり2 .....
skin[9].FlameNo 00000000  

skin[9].Weight 3F800000
```

3F800000 ってところですね。これがどこと対応しているのかというと…



はい、「にやり」が思いつきり右にふれてますね？左が 0.0 で右が 1.0 っていう例のアレです。ちなみに 3F800000 は浮動小数点の 1.0 を表しています。

はい、これは VMD データです。これが PMD のどこにつながっているのかというと…。ボーンがあつたと思いますが、ボーンの後に IK がきて、その後にスキンデータってのがあると思います。

こいつが今回の表情を動かすためのデータです。

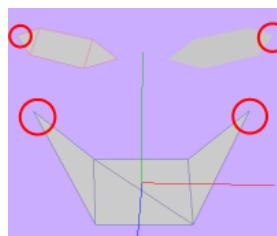
で、MMD の表情はボーンを動かして表情を変えるのではなく頂点そのものを移動して変形

させることによる「モーフィング」で対応している。MMD 界隈では単純に「モーフ」と言っていますが、某ゲームの「ネクロモーフ」とは関係ございません。あと、モーフィングは 3DCG では頂点ブレンディングやブレンドシェイプなどと言われています。

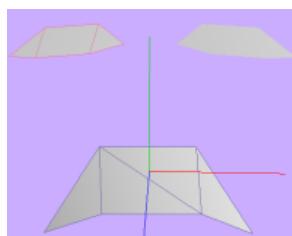
<http://ja.wikipedia.org/wiki/%E3%83%A2%E3%83%BC%E3%83%95%E3%82%A3%E3%83%B3%E3%82%B0>

モーフと表情

この効果はホントは「変形」をすることにより、変身や変形（ターミネータ2のアレ等）に利用されていたのですが、MMD 界隈では表情をつけるのに役立っています。もう一度言いますが、頂点を動かして表情を作ります。



例えば、このウーズマンの笑い表情の〇で囲んだ頂点を下にずらして下さい。そうすると



「笑い」が「悲しみ」になりました。つまりそういうことですわ。頂点を動かすことにより表情が変わるので、実際のデータは先ほども言ったように IK のあとにあります。

```
skin_data.skin_name[0]          B2 B1 73 B5 00 FD base
skin_data.skin_name[16]         FD FD FD FD ....
skin_data.skin_vert_count      00000956
skin_data.skin_type            00
skin_data_base[0].skin_vert_index 00001749
skin_data_base[0].skin_vert_pos[0] 3F6948FF 4197A613 BF7FDE93
skin_data_base[1].skin_vert_index 0000174A
skin_data_base[1].skin_vert_pos[0] 3EA638D6 4197DB03 BF465D8E
skin_data_base[2].skin_vert_index 0000174B
skin_data_base[2].skin_vert_pos[0] 3F3FE7C1 41984F28 BF8F53F8
skin_data_base[3].skin_vert_index 0000174C
```

こういうデータです。

おわかりだろうか？ とりあえず言っておくと、スキン名（要は表情名な）が 20 バイト。ただ、base ってのは基本情報（無表情）なので、MMD 上には現れない。ただ、こいつが重要なのは無表情時の頂点座標と、そのインデックスを定義しているということなのだ。

で、とりあえず言っておくとこのインデックスと座標は既に最初の頂点定義データと同じものだ。ではなぜそんな二度手間データをこんなところに置いているのか？ それは表情を

つけた時にわかります。がヒトコトで言うと**表情データはPMDにおいては「オフセットデータ」ということです**。今わからなかつても後々ぼちぼちわかると思います。

あと、ヒトコト言っておくと、もしかしたらボーン制御よりこっちのが簡単かもしれませんので、ボーンで挫折した方も挑戦してみてはどうでしょうか？

そうですね。僕のほうでは今「わらい」「にやり」「わらい(口)」を調整していますので、ここを見てみます。もし皆さんの方で別の表情を作っているのならそれに合わせて考えなおして下さい。

ちなみに今「わらい」のウェイト値を0.480にしています。ココらへんはボーンと一緒にbaseが0.520となるわけです。まあ、それは置いておいてPMD側の「わらい」を見ると

```
82 ED 82 E7 82 A2 00 FD わらい
FD FD FD FD .....  
0000024B  
02  
00000609  
00000000 B97B3333 80000000  
0000064  
00000000 B97B3333 80000000  
00000065
```

こんな感じです。Baseと同じく表情名20バイト、対応頂点数4バイト、スキントタイプ1バイト、あとはインデックス情報とオフセット情報です。先にインデックスが来て、そのインデックスをbaseからどれくらいオフセットする(ずらす)のかを指定する情報になっています。

簡単に言うと「頂点ごとに、新しく定義した別の頂点と補間をかける」ってことです。わかんねーなら手を出すな。ドツボるぞ。

前にも説明したように、デフォルトのモデルがあつて、それの一部の頂点をいじった頂点情報がある…と。つまり単純に考えて頂点情報がふたつあるというわけだ。もう少し言うとインデックス情報も2つある。

モチロンIASetVertexBuffers関数には…あー、やめよう…だめだやっぱり難しい。うん、無理に難しくするこたあねーわな…とりあえずCPUでモーフすつか。うーん、うん、とりあえずCPUで出来る部分はまずCPUでやろうや。そつからGPUにどう移行するか考えよう。

それなりに膨大ではあるんだが、ともかくデータを読み取ろう。今回もmapとvectorの二段構えでやっていこうかと思います。まず名前をロードしてそれをmapのキーとし、インデックスと頂点をmapの値として内部データベースを作っちゃいます。

まずはとにかくここまでやってみましょう。今回はIKすっ飛ばすので(後で使いますが)、シークですっ飛ばすか、適当なバッファにデータを入れて表情データを読んでください。

ちなみにIKデータはひとつあたり $2+2+1+2+4+2*n=11+2n$ バイトです。で、最初の2バイト

はデータ数(unsigned short)を表しているので…あとはわかるな?

要は ik_data_count*11+2n バイトだけすっ飛ばせばいい。ちなみにシークでぶつ飛ばすには fseek という関数を使用する。

```
fseek(fp, ik_data_count*(11+2*n), SEEK_SET);
```

とでも言えばいい。くれぐれも言っておきますが、これをそのまま使って「動かねー」とか言わないでくださいね。ともかくすっ飛ばして下さい。で、しれっと fseek を使いましたが、この手のシーク系関数はゲーム作ってると結構使うことになりますので、ゲームの仕事したい人は覚えておいて下さい。それ以外の職業だと…あまり使わないと思います。

あ、2n の部分ですが、これらバイト目の ik_chain_length あたりにあたりますので、この IK データってのは可変長みたいですね。ここに来て可変長ですよ…。

あー、やっぱりこの状態ならシークより読みじまった方が早いですね。というわけで、もう構造体を作っちゃいましょう。

```
struct IKList{
    unsigned short boneIdx;
    unsigned short tboneIdx;
    unsigned char chainLen;
    unsigned short iterationNum;
    float weight;
    std::vector<unsigned short> boneIndices;
};
```

で、chainLen で boneIndices の数をふやしながらロードしましょう。…もうわかるよね。



最終的にこんなにこやかな靈夢さんが表示されればオッケーです。ここまでヒントでできますかね? きつついですかね?

このへんまで来たらあまり優しく教える必要はないと思うんですが、いよいよ表情データ

です。

表情データは、前にも書きましたが、一番最初の4バイトが表情の数です。その後からデータが始まります。

- 表情名が20文字=20バイト
- 頂点数4バイト整数=4バイト
- 表情タイプ1バイト(無視していい)=1バイト(アライメント発生)

ときて、この後に頂点データが入ります。頂点データは

- 頂点インデックス4バイト整数=4バイト
- 3頂点座標(オフセット)= $3 \times 4 = 12$ バイト

インデックスと座標情報が固まっているので、構造体作つといふのがいいですね。

```
struct IndexAndOffset{  
    unsigned int index;  
    XMFLOAT3 offset;  
};
```

そしてデータの構造としてはそれぞれの表情に対して1000～2000くらいの頂点情報があります。

そしてひとつひとつの表情は名前で管理されています。ということはボーンとモーションの関係に似ていますよね？ですからここではmapとvectorで管理します。

つまり

```
map<string, vector<IndexAndOffset>> skindata;
```

てな感じで表情のデータが構成されます。

あとはロードしながらボーンの時と同様にデータを放り込んでいきます。今回はひとつの表情に頂点が複数あるため、ダブルループで読み込む必要があります。

```
for(表情ループ){  
    表情名読み込み;
```

```

頂点数読み込み
表情タイプ無視
頂点数リサイズ
for(頂点ループ){
    頂点データ読み込み
    skindata[表情名][ループ変数]=頂点データ;
}
}

```

とやれば、表情名がわかれれば頂点データを取得することができます。上の擬似コードの「読み込み」の部分はもちろん fread を使います。

あと「無視」の所で fseek 使います。

さて、これで表情データがロードできました。できたんですが、ここで注意点を一つ。

表情データは「基本データ」と、「個別表情データ」からできています。

基本データは表情データの最初に配置され、デフォルトの表情（無表情）を定義します。それ以外の表情データが個別表情データになります。

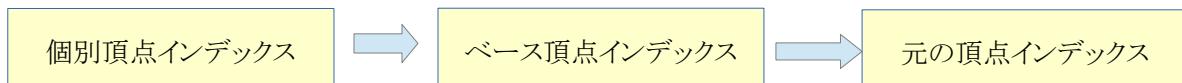
ところで個別表情データは元の位置からのオフセットを持つことによって、表情変化に必要なデータとなっていますが、この中にある「インデックス」…注意が必要です。

このインデックスがそのまま頂点データのインデックスを表していると思ったら大間違いです。

違います。

違うんですよ。個別のやつは「ベース」の中でのインデックスを表しています。つまり個別データのインデックスから元の頂点データにアクセスできるわけではなく、一度ベースデータを介してじゃないとアクセス出来ないんですよ。

つまり



てな感じで面倒です。面倒ですが仕方ないです。

すでに表情データベースを作っていますので、

```
data=skindata("笑い");
```

```
for(頂点まわし){
```

```

index = skindata("base")[data(i).個別インデックス].index;
頂点(index).pos+=data(i).pos;
}

```

てな感じですね。これで頂点が動きます。まあ、これを頂点ノーマル生成前にやってみるのが一番手っ取り早いですね。ちなみにこの例ではウェイトもなーんもかけていませんので、微妙な表情を作りたい人はウェイトもきちんと作りましょう。微妙に変えれば



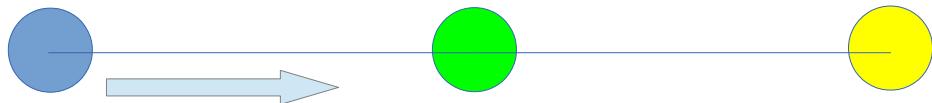
このような、なんといふか、アルカイックスマイルってやつも表示することができます。
意外と簡単でしょ？（ここまでついてこれた人にとっては…ね）

みんな…だいすき…IKつ…！

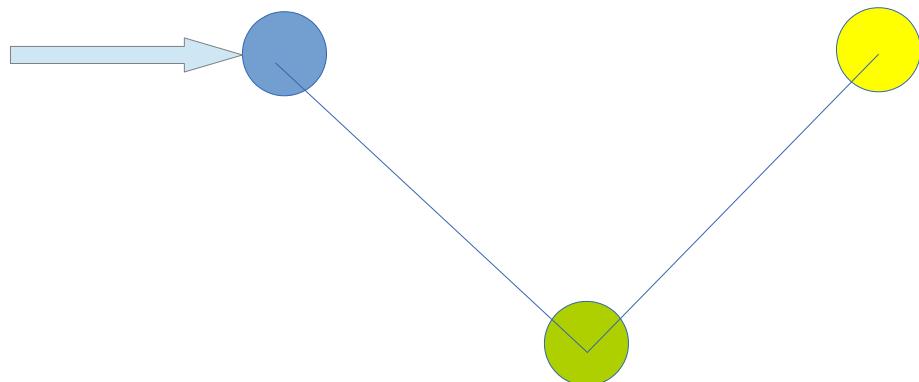
インパアアアアアアアアス!! キィイイイイイネマティクスゥウウウウウッ!!

お待たせいたしました。ついにキました。きちゃいましたーっ!!

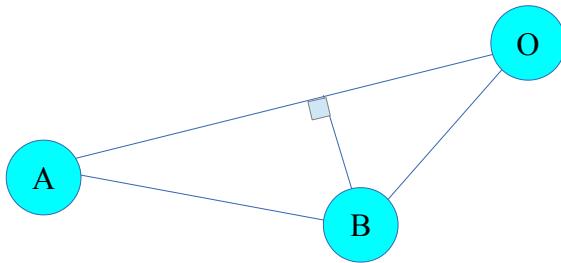
簡単に説明すると



こういう3つのノードがあって、ボーンが2つの状態であるとします。そうすると、一番左のノードを右側に平行移動した場合、図のように真ん中のノードが押し出される形で移動することになります。



これは分かりますか?で、単純に考えると実装も…この状況で2Dならばそう難しく無いことがわかりますね?



基本的な数学

できた三角形のAとOの間に直線を引き、その間のBから垂線を引けば、AOの距離が $AO = AB\cos\alpha + OB\cos\beta$ になるようにBの座標を決定すればいいわけです。

今まで僕らが勉強してきた武器の中で使えそうなのは余弦定理ですね。ボーン自体の長さが変わらないのですから、

$\angle AOB = \theta$ とすると

$$AB^2 = OA^2 + OB^2 - 2|OA||OB|\cos\theta \text{ が成り立ちます。}$$

AB, OA, OBは固定値ですから $\cos\theta$ が一意に成り立ちます。一応はね。まあ $\cos\theta$ とかが目的じゃなくて、Bの座標が目的なんですよ。となると垂線が降ろされた地点は OA の直線上の何処かなわけです。ということは垂線地点を M とすると $M = O + k\vec{OA}$ になるわけ。じゃあこの k はどうやって計算するかってーと OA の長さと OB 射影の割合で出せるわけ。つまり

$$k = \frac{\vec{OB} \cdot \vec{OA}}{|OA||OA|} = \frac{\vec{OB} \cdot \vec{OA}}{|OA|^2} \text{ ですから垂線地点は } M = O + \vec{OA} \frac{\vec{OB} \cdot \vec{OA}}{|OA|^2} \text{ になりますね。}$$

ところで上の式の AB, OA, OB はわかっている。そして知りたいのは $|\vec{OB}|\cos\theta$ である。

$$|OM| = |\vec{OB}|\cos\theta = \frac{OA^2 + OB^2 - AB^2}{2|OA|} \text{ である。右辺は分かっている値で更に言うと、これは } OM \text{ の大きさである。}$$

$$\text{つまり } M = O + \vec{OA} \frac{|OM|}{|OA|} \text{ これでMの座標が出ましたね。}$$

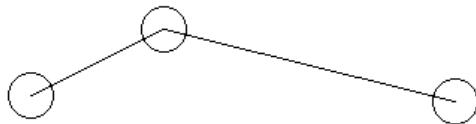
$OM = |\vec{OB}|\cos\theta$ が分かっているということは、あとは三平方の定理にて垂線の長さがわかりますね。OBの長さは分かっているわけですから、 $|MB| = \sqrt{OB^2 - OM^2}$ です。

さらに OA から 90° の角度のベクトルは…まあ2Dの場合であれば x と y ひっくり返して正規化すればいいわけ。

M 地点から OA 90° 回転の正規化ベクトル * $\sqrt{OB^2 - OM^2}$ をすりゃいいわけ。となると

B の座標は $B = M + \vec{OA}^R \frac{\sqrt{OB^2 - OM^2}}{|\vec{OA}|}$ となります。うーん。まあ簡単でしょ?簡単じゃない?ごめんね。でも分からんレベルでもないでしょ?

簡単なIKならここに書いてある式だけでなんとかなります。わりかし簡単です。



上の式まんまプログラムで書くとこんなところです。

```
float OM=(float) ((oa+ob-ab)/(2*Math.Sqrt(oa)));
Vector2 M=(new Vector2(o))+voa*(OM/(float)Math.Sqrt(oa));

float MB=(float)Math.Sqrt(ob-OM*OM);

b=(M+
    voa.RightAngle()
)
*
(
    MB/(float)Math.Sqrt(oa)
))
```

さて、ここからが本番です。コントロールポイントが一個増えるだけで途端にヤヤコシイ話になってしまいます。解が無数になってしまふこと

さて、上の例であれば軸からOBAでした。これがOCBAになつただけで余弦定理が使えなくなります。こいつをなんとかするためにヤコビIKだのCCD-IKだのがあります。

ヤコビアンIKはヤコビアン(ヤコビ行列式)を用いる解法ですが

<http://ja.wikipedia.org/wiki/%E3%83%A4%E3%82%B3%E3%83%93%E8%A1%8C%E5%88%97>

わけわからんです。こうなってくると現状の武器だけでは太刀打ちできません。

ということでMMDがCCD-IKを採用しているってのも有りますが、時間的な問題も有りCCD-IKの話からやつていきます。

CCD-IKってのは、前にも話したと思いますが、Cyclic-Coordinate-Descentは巡回下降座標系アルゴリズムって訳するのかな？まあとにかく繰り返して巡回することにより解を求めるアルゴリズムです。

このため最終回数は決めておく必要がありますし、完全解ではありません。実際完全解なんてIKにはありえないことは3つのコントロールポイントでわかつたと思います。

2次元 CCD-IK から学ぼう

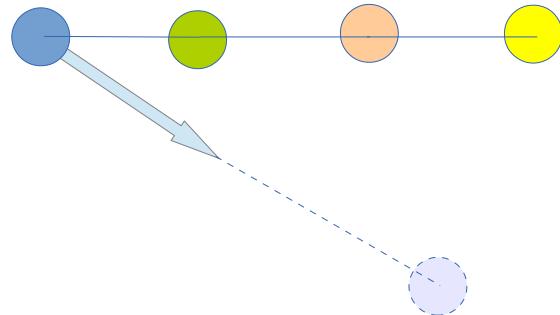
2次元から行くのは決して僕が3Dが苦手だとかそういうわけじゃないんだからね。あんたたちが分かりやすいと思って2次元からやってるんだからね。勘違いしないでよね。

じゃあ具体的に手法を説明しますが、IKのボーンから遡るようにして回転を行い、座標的に一致した時点で解とするのです。

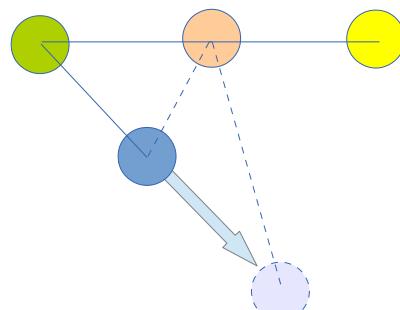
たまにMMDいじっていてもおかしな姿勢になるのは回数末端に達したか、座標的に一致してしまったばかりに解とされてしまったためだと思われます。

図で示すとこう。

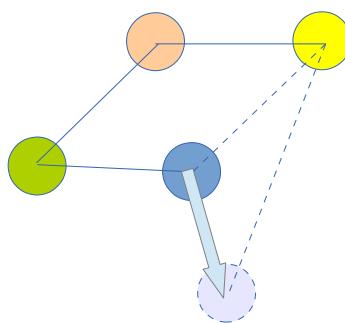
末端を図のように動かしたいとします。



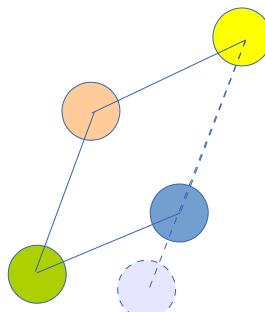
こういった場合、まず末端から、最も目的の座標に近づくように回転させます。とは言えまだ座標的な一致には程遠いです。



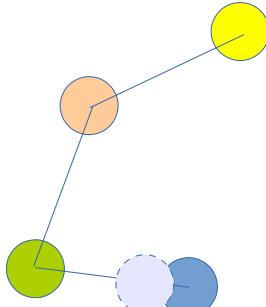
次はひとつ上(親)のボーンを回転させて最も近づくように回転させます。だいぶ近づいてきましたが、もうひと頑張りです。



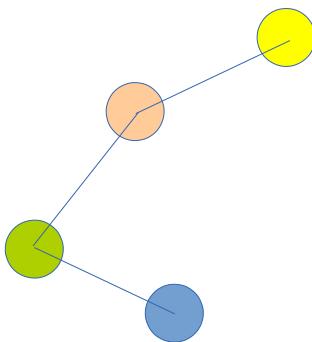
最後にさらに親のボーンから回転させて近づけます。もう限界です。



ここまで来たらまた最初に戻って回転します。



ズレていますが、最初に比べればだいぶ近づきましたね。後一回だけやつときましょう。こんなのはコンピュータに任せるべきです。



と言った風に近似解に近づけていきます。…まあ考え方自体は思ったより簡単でしょう？単純なものであれば繰り返し回数は2~4回くらいでいいんじゃないでしょうか？

繰り返し回数が増えれば増えるほど精度は上がりますが計算量が爆発的に増えますんで、そこはPMDデータの中の IterationCount を参照すればいいでしょう。

…だいたい13~15回になっていますね。15回も必要なのかなあ。

アルゴリズムは単純です。末端のボーンの根っこから仮のベクトルを目的座標へ向けて、ボーンのベクトルがそれと一致するように回転させます。

…これ、余弦定理より簡単ですね。

ループ回数が多くなってしまうのだけが難点ですが、演算自体が単純なため速度的にはそこそこ出るでしょう。ちなみに2Dの例で言うとこんな感じです。C#のフォームで例を作っているので、変な書き方になっていますが要はループですね。

```
for (int c=0;c<cycle;++) {
```

```

//末端から根本に向かって曲げていく
for (int i=nodecount; i>=1;--i) {
    //回転角度計算
    Vector2 vec0 = new Vector2(pos[nodecount-1])-new Vector2(pos[i-1]);
    Vector2 vect = new Vector2(target)-new Vector2(pos[i-1]);
    double angle0=Math.Atan2(vec0.y, vec0.x);
    double anglet=Math.Atan2(vect.y, vect.x);
    float rot=(float)RadianToDegree(anglet-angle0);
    //角度制限
    rot=Math.Min(rot, 114);
    rot=Math.Max(rot, 0);
    Vector2 center=new Vector2(pos[i-1]);
    Matrix mat=new Matrix();
    mat.RotateAt(rot, center.AsPointF());
    //根本を回転させる際は末端も一緒に回転させる
    for (int j=i; j<nodecount; ++j) {
        Vector2 vec=new Vector2(pos[j]);
        pos[j]=vec.Transform(mat).AsPoint();
    }
}
}

```

てな具合です。

ではここからがつ…まさに本番

PMDのIKを動かしましょう

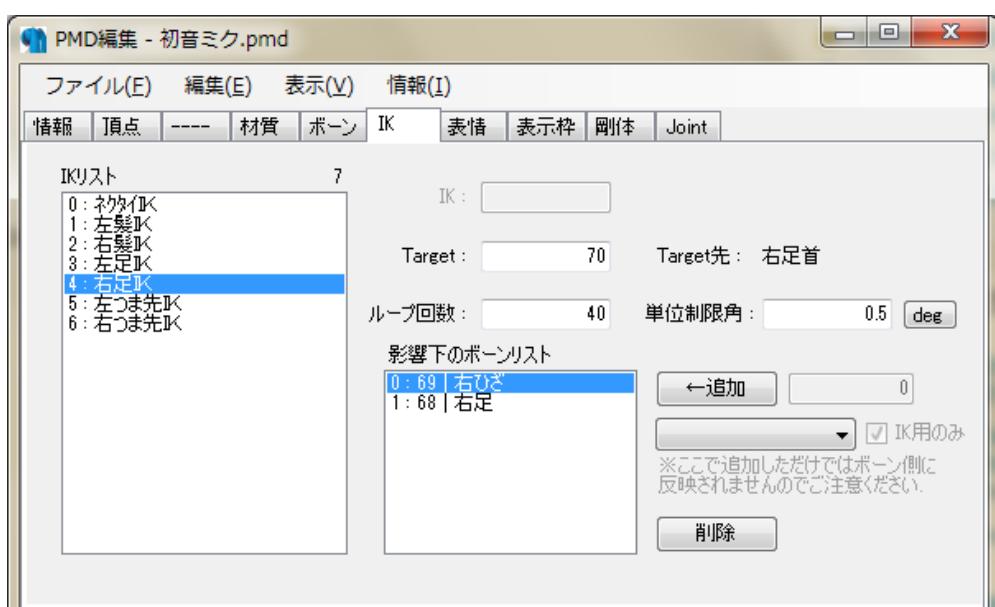
PMDのIK情報を見る

まず、ボーン情報を見て下さい。

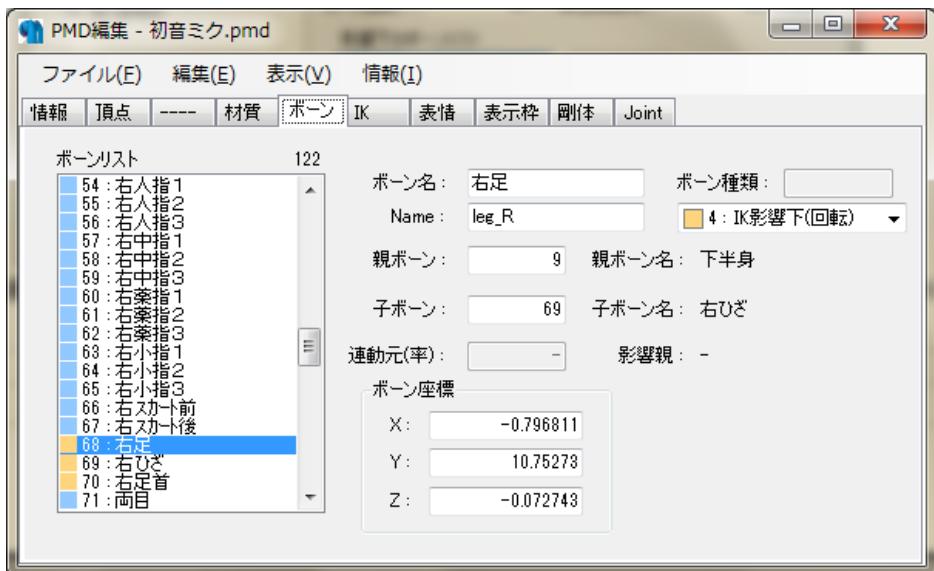
ボーンの中にボーン種別が2であるものがあると思います。こいつがIKボーンを表しています。ボーンの名前を見れば一目瞭然ですね。

bone[84].bone_head_pos[0]	00000000 00000000 00000000
bone[85].bone_name[0]	8D B6 82 C2 82 DC 91 左つま先 IK ...
bone[85].bone_name[16]	FD FD FD FD
bone[85].parent_bone_index	0053
bone[85].tail_pos_bone_index	0075
bone[85].bone_type	02
bone[85].ik_parent_bone_index	0000
bone[85].bone_head_pos[0]	3F6DBF38 00000000 C1
bone[86].bone_name[0]	89 45 82 C2 82 DC 91 右つま先 IK ...
bone[86].bone_name[16]	FD FD FD FD
bone[86].parent_bone_index	0054
bone[86].tail_pos_bone_index	0076
bone[86].bone_type	02
bone[86].ik_parent_bone_index	0000
bone[86].bone_head_pos[0]	BF6DBF38 00000000 C1
bone[87].bone_name[0]	89 BA 94 BC 90 67 91 下半身先
bone[87].bone_name[16]	FD FD FD FD
bone[87].parent_bone_index	0009
bone[87].tail_pos_bone_index	0000
bone[87].bone_type	07
bone[87].ik_parent_bone_index	0000

できればPMDEditorを見て欲しいんですが…まあそこは強制しませんが、PMDエディタで右足IKを見ると…



右足、右膝が影響下に入っています。ではボーン情報を見るとどうなっているでしょうか？



右足、右ひざは4番のIK影響下(回転)というボーン区分になっています。つまり、回転効くっちゃ効くけど、IKに影響されるってわけ。

このへんのデータはPMDではどこにあるのかというと、前に無視した部分…ボーン情報の直後ですね。

- データとしては以前にも書きましたが
- IK ボーンインデックス
- ターゲットボーンインデックス
- 子ボーンの数
- 再計算回数
- 単位制限角(1回あたりに曲げられる角度)
- 子ボーンインデックス(子ボーン数)(子ってよりは遡れるボーンインデックス)

という具合です。で、単位制限角なんですが、これはIK巡回において曲げられる角度(ラジアン)のようです。

自分でMMD動かしたらわかりますが、右足と右ひざはIKをオンにしていると回転させてもほとんど動きません。IK影響下にあるため、IKによってボーンの状態(角度)が変わります。

手順はさっきの2Dとそう変わらないとは思います。変わるのは、軸が2Dの時のように一定(Z軸)ではないということです。

じゃあどうやってその「軸」を決めればいいのか?

- ①元のボーンの向きベクトルを出す
- ②移動(したい)先とボーン起点を結ぶベクトルを出す

- ③①と②の外積から直交ベクトル(これが『軸』)を出す
- ④①と②の間の角度を計算
- ⑤③の軸と④の角度から回転クオータニオンを計算する
- ⑥親に移動してくりかえし(指定回数分)

いざ CCD-IK を実践

元のボーンの向きベクトルは IK ボーン情報のヘッドとテイルから算出することができます。さらにターゲットベクトルはヘッドと、移動先 IK 座標からわかるわけです。

さて次に回転角度を計算するのですが、どこが『支点』になるのでしょうか？『支点』とは、先程の 2D で説明した



赤丸のように「動かない部分」のことです。

もちろん「動かない」と言っても他の操作では動きますし、調査する IK の範囲の中では動かないだけの話です。例えば単純にサッカーボールを蹴つ飛ばす時には膝とつま先の座標はむっちゃ移動しますが、股間が移動しないようなものです。

なんか回転させる時って、支点が重要でしょ？ その支点に当たる部分です。

さて、「動かない部分」は基本的にはさかのぼりボーンの最後にあたります。さかのぼりきった部分ですね。なお、IK は自分自身は数えてないので、さかのぼりボーンの最低数は 1 になります（つま先 IK とか）。その場合はつま先と支点だけになります。他には影響しないわけです。

ともかくこれで元のベクトル（ボーン座標 - IK 親ボーン座標）が出るわけです。さらにターゲット座標ですが、これは VMD から取得します。VMD の中の Location(Float3) を使用します。今まで使ってなかったですが、やっと役に立ちますね。

ただしここに書いてある座標。これは元の座標（ボーン座標）からのオフセットです。つまり動かさなければ $x=0, y=0, z=0$ です。

さて、とにかく元ベクトルと目標ベクトルがでました。元ベクトルから目標ベクトルへの回転を出すにはまず軸が必要です。軸は元ベクトルと目標ベクトルの外積から出せます。

外積は

`XMFLOAT3 XMVector3Cross(元, 目標)`

で計算できます。外積くらいこんな関数なくても自分で計算したいトコロですが、最適化等の関係で関数使つといいたほうがいいでしょうね。

次に元と目標の角度を出します。

`XMVector3AngleBetweenNormals` という関数で間のラジアン角がわかるっぽいので、そいつで計算しましょ。なお、突っ込むベクトルは単位ベクトルである必要があるっぽいので、`XMVector3Normalize` で正規化しておきましょ。

角度と軸がわかつたら回転だ。

`XMQuatFromRotationAxis` で回転させましょ。

あとは基本的には 2D と同じです。なかなかうまくいかないと思いますが頑張ってみましょうか。

実際に IK ボーンを動かしていきますが、その前に PMDEditor の ReadMe を見ておいたほうがいいです。

気になる箇所は

●(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 处理が行われる

IK ループ回数(整数) : IK 处理での計算回数(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

※IK 处理においては、特定の制限角度がボーン名に従って MMD 側で調整されるようです(PMD)

例: **ひざ→縦(X 軸方向)にしか稼動しない**

赤字の部分、これはハマリそうな部分なので、見落とさないようにしましょ。そのままラジアン値を使うと、膝とか全然曲がらないと思います。

もつと言ふとこれが書いてある前後の部分に「振りボーン」「回転連動ボーン」の説明が書かれています。興味がある人は個々にも対応すれば一部のモデルでの不具合が解消されるでしょう。

で… 実際のプログラミングですが、IK を動かした情報は VMD ではやはり名前と Location で記載されています。

なので… IK リストとの名前マップを作つておきましょ… 名前マップ多いでですね。流石にやんなるなあ…。

つまり IK リストを読み込んだタイミングで、ボーンインデックスを見に行って名前を確認。名前をキー、IK リストを値とした名前マップをつくります。

例によって

```
map<string,IKList> _ikmap;
```

と言った感じで作りましょうか。IKは頂点だのインデックスだのボーンだのデータに比べるとはるかに少ないので大したことはありません。

```
_ikmap("右足 IK")=iklist;
```

ですね定番ですね。

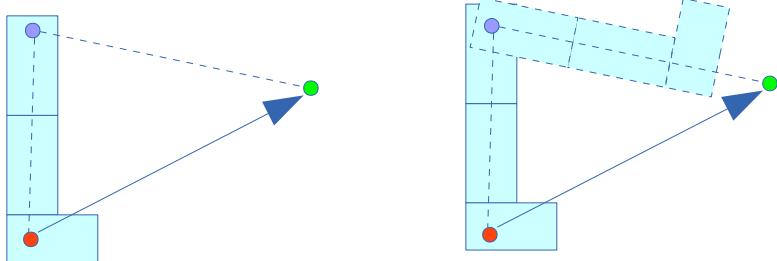
さて、ここからやっていくわけですが、一旦ポーズ状態で出力してみます。イキナリ動いているところに行って失敗すると原因を掴みづらいので、最初から足を一步踏み出しているポーズを使います。



このポーズはIKで足を踏み出しているわけであって、FK回転でやっているわけではありません。まずはIKはともかく「足」が目的の座標を向くということをやってみてください。やり方はかんたんです。

IKボーンから最大さかのぼって、その位置に回転をかけます。

回転は例えば右足IKであれば2つさかのぼって右足を回転させます。なお、回転軸は前にも言いましたが外積でだせます。まずはここまでやってみましょう。



点線が2つあると思ういますが、この点線をなすベクトルから外積を出して、点、線間の間の角度を測るだけです。で、右足からさきを回すだけ。

まずはここまでやってみてください。これがやれなきやIKがウンヌンとか言っている場合

じゃないです。悪いけど諦めてFKでやつといてください。難しいんです。理解できなきゃしないでいいです。

ここができたら最初の要領で足を動かしてみましょう。

でもあ角度制限をつけてないところになります。



おお…痛い痛い。

なんでこうなるのかというと、膝先を曲げた時点で解に到達しちゃうからです。これではいかんのですよ。

さて、PMDには「ひざ」のとる角度が特殊だといろんな資料に書いてあったので、信じることにしましょう。

ということでIK名が「右ひざ」もしくは「左ひざ」ならばX軸方向にしか稼働しないらしいので、強制的に軸をX軸にしちゃいましょう。

簡単です。

```
if(対象 IK が“右ひざ”もしくは“左ひざ”){
```

```
    軸.X=-1;
```

```
    軸.Y=0;
```

```
    軸.Z=0;
```

```
}
```

それ以外の場合は外積で軸を作りましょう。

…後述しますが外積で軸を作れないパターンがありますので、それには注意しましょう。それを考慮しないとしゃがんだ時とかに膝周りが「あらぶる」事になります。

「荒ぶる膝神さま」を鎮めるにはチョイと工夫が必要になります。



2つのベクトルが一致に近づけば近づくほど外積のベクトルがちっちゃくなっちゃうので、計算上、軸を維持できなくなります。なので膝を鎮めるには

内積を計算し、内積の結果の絶対値が1に近づく、もしくは内積からACOSで角度を算出して角度の絶対値が0に近い場合に処理を打ち切る必要が出てきます（うち切っていい）。

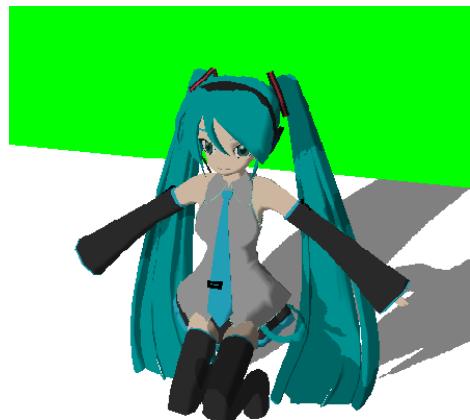
ちなみにベクトルを2つ持ってきて内積出して、その内積結果がcos値になっているので、
角度=acos(内積);

で出せます。

※外積を求めるときも内積を求めるときも注意して欲しいのが**2つのベクトルは必ず正規化してから計算**して下さい。

分かっているようでもやらかしますので注意しましょう。

以上の事を考慮に入れた上でIK計算を行えばとりあえずはキックしたりしゃがんだりしてくれます。



とりあえず、CCD_IK 第一段階のスクショ置いときます。

```

///だが、CCDであるため途中経過の座標も重要なので、関節内部にテンポラリとして保持しておく
///関節を抜いたら用済みである
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-ikpos;//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);
            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;
            }
            //角度計算
            XMVECTOR rot=XMVector3AngleBetweenNormals(XMLoadFloat3(&Normalize(originVec)),XMLoadFloat3(&Normalize(transVec)));
            //角度が小さすぎる場合は処理を打ち切る
            if(abs(rot.m128_f32[0])<0.0001f){
                return;
            }
            if(rot.m128_f32[0]>2.0f){
                rot.m128_f32[0]=2.0f;
            }

            //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);
            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;
            }
            //角度計算
            XMVECTOR rot=XMVector3AngleBetweenNormals(XMLoadFloat3(&Normalize(originVec)),XMLoadFloat3(&Normalize(transVec)));
            //角度が小さすぎる場合は処理を打ち切る
            if(abs(rot.m128_f32[0])<0.0001f){
                return;
            }
            if(rot.m128_f32[0]>2.0f){
                rot.m128_f32[0]=2.0f;
            }

            XMVECTOR q=XMQuaternionRotationAxis(XMLoadFloat3(&norm),rot.m128_f32[0]);

            //ボーンの変換行列を計算
            XMFLOAT3 offset=_boneoffsets[itn[i]];
            XMATRIX RotAt=XMMatrixTranslation(-offset.x,-offset.y,-offset.z)*
                XMMatrixRotationQuaternion(q)*
                XMMatrixTranslation(offset.x,offset.y,offset.z);
            _boneMatrixes[itn[i]]=_boneMatrixes[itn[i]]*RotAt;

            offset=tmpLocation[i];

            //理論上の変換行列を計算
            RotAt=XMMatrixTranslation(-offset.x,-offset.y,-offset.z)*
                XMMatrixRotationQuaternion(q)*
                XMMatrixTranslation(offset.x,offset.y,offset.z);

            //理論値を更新
            ikpos+=RotAt;
            for(int j=i-1;j>=0;--j){
                tmpLocation[j]=RotAt;
            }
        }
    }
}

```

さてIK関連ですが…実は「ひざ」関係以外にも色々と特殊な部分があるみたいなので、マニュアルを見てみるとこう書いてあります。

「単位制限角による制限はIK影響下ボーンが下位に進むにつれ(ボーン列的には根元に行くくなるに従つて)、制限角度が拡がっていく仕様なので、多関節の場合、先端付近より根元の方が比較して大きく曲がるようになる。」

…らしいです。つまり、右足IKの場合、「右ひざ」より根っこにある「右足」の角度のほうがよくまがるってことです。

つまり制限角が0.5であれば、最も根っこの「右足」の制限角は2.0なのですが、そうすると右ひざの制限角は1.0にするとかそういう感じですね。

```
float strict=(2.0f/(float)nodecount)*(float)(i+1);
```

制限角度がこのように変わりますから根本に向かうにつれ1回で回転する角度がキツくなっています。

なお、それでも不自然に荒ぶる場合は、IK角度算出の際に、角度を半分にすればそれなりに自然に見えます。

あふう…とりあえずCCD-IKのお話はこれにて終了です。流石に疲れましたわあ…。こうやつて書くと原稿の量 자체は大した事はないんですが、これをテストする段階で死ねましたわ。

セルシェーダについて

ここまでレンダリングてきて、物足りない人もいると思います。何が物足りないんでしょうか…そう。輪郭線がないんですよ。

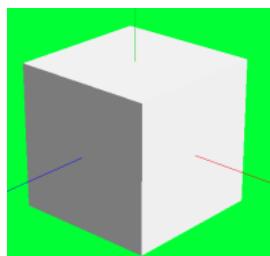
輪郭線

輪郭線を表現する方法はいくつあるんですが、一番簡単なのは、モデルをちょっと拡大して、面を全部反転するって方法ですね。

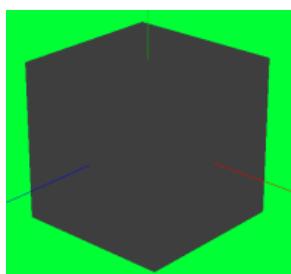
反転法

プログラムで言うとカーリング方向を変更することなんですが…順を追って説明しましょう。

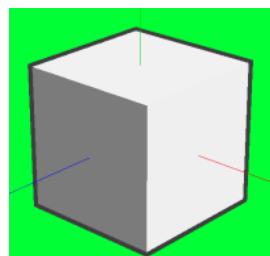
例えば図のような立方体があるとします。



そしたらこれをちょっと拡大して黒く塗りつぶします。



で、外側の立方体を反転します。



なんちやないでしょ？

じゃあ、これでPMDもやれるかなー？って思うのかもしれません、恐らく上手く行きません。立方体みたいに単純にはいけないと思います。

ともかく立方体を拡大ってのは言い換えると**法線方向に面をずらしている**とかんがえることができます。

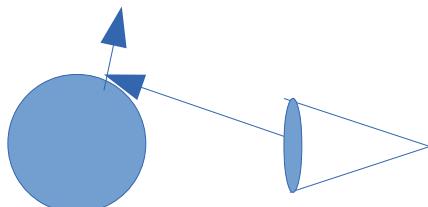
もしこの方式をとるのであれば、通常レンダリングする前に、頂点シェーダで法線方向に頂

点を少し移動させたものを黒で描画(このときのカーリングはFRONTにしておくこと)その後で通常描画すれば、まあ輪郭線っぽいものは出るでしょう。

これが一番単純であろうやり方かな。ただしあんまし綺麗にはならないであろうし、透明が絡んでくるとまたヤヤコシイことになるでしょう。

視線と法線ベクトルから…

視線ベクトルと法線ベクトルの内積からエッジを判断することもまあ可能です。どういう事がというと、視線ベクトルと法線ベクトルの内積が0に近いならば黒く塗りつぶす



とすればある程度輪郭線っぽく見えます。こいつだと1パスで終わりますし使うのは無い席なので非常に簡単に実装できるのですが、

```
float4 eyevec=normalize(eyepos-v.ppos);  
if(dot(eyevec, v.norm)<0.0005){  
    return float4(float3(0,0,0),alpha);  
}else{  
    return float4((rgba*v.diffuse).rgb, alpha);  
}
```

いかんせん仕上がりがきちゃない。



あと、輪郭線にムラが出ます。更に言うと、靈夢の袖の端のようにポリゴンがいきなり切れている部分ではエッジが出ません。ちなみに言うと裾のフリルの部分でも出ません。1パスではこれが限界みたいですね。

次によくやるのが、情報(深度情報)からエッジ抽出する方法です。輪郭線抽出では画像処理的な技術が必要になりますので、ちょっと覚悟しましょう。

深度から輪郭線

深度情報から輪郭線を抽出ってのはどういうことなのかというと、自分で絵を描く時を思い浮かべて欲しいのですが、輪郭ってどういうところに書きますか? そうですね。端っこ(エッジ)です。では端っこってのはどのように判断するのでしょうか?

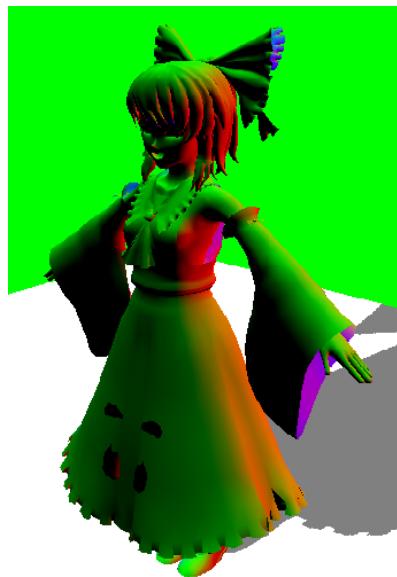
ここで目(カメラ)からの距離(Z)に注目すると、輪郭線を書こうと思う部分では Z 値が極端に変化していると思います。 Z 値に変化がなければそもそも輪郭を書く必要などないのですから。

ということでシャドウを作った時のように、1パス目で Z 情報を抽出してそのテクスチャを解析して Z 値が極端に変化する部分を黒く塗りつぶすということで輪郭線を描画することができます。

法線マップから輪郭線

法線マップから輪郭線!?そもそも法線マップって何さ?

法線マップってのは法線ベクトルXYZを色情報RGBに変換して画像にしたものです。通常描画してピクセルシェーダに渡った時点で法線ベクトルがラスタライズされていますので、そのXYZをRGBにするだけで、ほらこのとおり。



X成分がRに、Y成分がGに、Z成分がBになっているってわけです。

return float4(norm.xyz, 1);

ってやってるだけですね。なんなんなんなんちゃない。

隣接頂点の法線情報を見る

これはステンシルシャドウの時に輪郭をとったのと同様の方法ですね。法線ベクトルを比較して、カメラ側から見た時にポリゴンの法線の方向がカメラ側に、隣接ポリゴンがカメラから遠ざかる方に向いていれば、そこを輪郭線としてみなしてもよいということです。

難しいですね。

この中で一番うまいこと輪郭が出そうなのは深度情報から輪郭線抽出なので、それ行ってみましょう。

深度から輪郭線抽出をやってみよう

さて、実際に深度情報から輪郭線を抽出してみましょう。今回はライト方向から見た深度ではなく、カメラ方向から見た深度が必要になりますので、まずはカメラ方向から深度レンダリング。

ここまででは「影」ができた人にとってはそこまで難しいものではないでしょう。さて、ここからどうやってエッジを抽出するのでしょうか？

やり方は色々とあります。

-1	-1	-1
-1	8	-1
-1	-1	-1

このようなラプラシアンフィルタを使うのもあります。ラプラシアンフィルタとかご「大層な名前」がついていますが、周囲のピクセルを-1倍。自分のピクセルを8倍、それらをすべて足すことによりエッジがくつきりするというものです。

このフィルタを深度(必要とあらば法線マップ)に行うことによってエッジを抽出することができます。

言うと互い違いに差をとり、その絶対値

PMXについて

PMDの次のバージョンのフォーマットにPMXというのがあります。



TDAミクとかにがもん靈夢とかがそうですわな。

色々とPMDから改良されているものです。で、結論から言うと、これはPMDマクロを作るよりもさっさとC++言語でロードしてしまったほうがよさそうだ。

何故かというと、TXSBINはループをすると非常に遅くなってしまう。だからループしたくないのだが、各頂点の頂点サイズが、ボーン影響度種別によって可変長になってしまったため、ループせざるを得ない。

だが残念なことにTXSBINでループすると非常に重いのである。TDA式ミクは26000程度の頂点だが、頂点を解析するだけで5分かかるてしまうため使い物にならない(ぱっと見フリーズしたようにしか見えない)。あとループとか、ネストが深いとバグっぽい挙動を示すので注意。

もうここはアレですよ君。**鬼に切ってC++でやつちゃいなさいで、なんかしらバグが起きた時にもバックグラウンドで読みさせながらC++で悩んで、データ解析が終わった後でもみやかにデータをチェックすればいい。**

いやあPMXいいですよ。

ヘッダーはまともだし、ウェイトは0.0~1.0になっているしで使いやすいですね。とはいって、pmdの時よりもさらなる知識が求められているのは間違いない。たぶんここにかかずりあつてたら、ゲームの方は作れなくなりそうな気がしますが、そのイバラの道を行きますか。行かれますか。

とりあえずデータの持ち方がだいぶ違うので、これを表示するところからやってみましょう。



頂点データ表示

基本はPMDエディタの中のPMX仕様.txtを見ればわかるので、少なくとも頂点情報だけは取ってこれると思います。頂点情報が正確に取れたらインデックス部分はPMDと全く一緒なので、ここまででは行けるのではないか?でしょうか?行けませんかね。とりあえずPMDのデータにムリヤリブチ込むところからやってみたらどうでしょうか?仕様を読み違えなきや30分もあればできるでしょう。

頂点一つ一つが可変長なんでヤヤコシイのは分かりますが、難しいわけではありません。面倒なだけですね。ちょっと上手く行かないとかヤヤコシイので諦めるのは**愛が足りない**と思います。

```

//頂点読み込み
std::vector<CPMDVERT> vertices(vertexcount);
for(int i=0;i<vertexcount;i++){
    fread(&vertices[i],sizeof(XMFLOAT3)*2+sizeof(XMFLOAT2),1,fp); //座標、法線
    if(header.byteInfo[1]>0){ //追加UVあるなら
        std::vector<XMFLOAT4> adduv(header.byteInfo[1]);
        fread(&adduv[0],sizeof(XMFLOAT4),adduv.size(),fp);
    }
}
BYTE weighttype=0;
fread(&weighttype,sizeof(weighttype),1,fp);

WORD boneIndex[4];
float boneWeight[4];
XMFLOAT3 sdef_c;
XMFLOAT3 sdef_R0;
XMFLOAT3 sdef_R1;
switch(weighttype){
    case 0://BRDF1
        fread(&boneIndex[0],sizeof(WORD),1,fp);
        break;
    case 1://BRDF2
        fread(&boneIndex[0],sizeof(WORD),2,fp);
        fread(&boneWeight[0],sizeof(float),1,fp);
        break;
    case 2://BRDF3
        fread(&boneIndex[0],sizeof(WORD),4,fp);
        fread(&boneWeight[0],sizeof(float),4,fp);
        break;
    case 3://SDEF
        fread(&boneIndex[0],sizeof(WORD),2,fp);
        fread(&boneWeight[0],sizeof(float),1,fp);
        fread(&sdef_c,sizeof(XMFLOAT3),1,fp);
        fread(&sdef_R0,sizeof(XMFLOAT3),1,fp);
        fread(&sdef_R1,sizeof(XMFLOAT3),1,fp);
        break;
}
float edgeScale=1.0f;
fread(&edgeScale,sizeof(float),1,fp);
}

```

開発時間ないんでこんなコードだよチキショウ。参考になんかしない!もうがれい!と思いますが、とりあえずスクショ置いときます。

そして、PMDとの違いはこの後のテクスチャ情報の持ち方でしようか。

PMX テクスチャテーブルとマテリアル

この持ち方はいいと思います。初心者にありがちな『同じテクスチャファイルを何回もロード』って事を防いでくれるよう最初からサポートしてくれています。

最初にテクスチャテーブルを作つておいて、材質からはそれを参照する番号のみを指定するといった状態ですね。

ただ…ただしですね、このPMX仕様.txtは『どうやってテクスチャパスを区切っているのか』ってのを明記していないんですね…。まあ大体の予想はつくんですけどね…

プログラマってのは大体の予想をして実験して失敗して次の予想を立てて繰り返しながら、1から10までの答えを知ってる人などおらんのですよ。偉い人にはそれがわからんのです。

とりあえず、テクスチャ情報は

- テクスチャ数(int)
- テクスチャパス * テクスチャ数

らしいので、`unsigned int texturecount +char texnametest(256)`読み込んでみましょう。あくまでも最初は実験です。

さて、このデータからアペニクさんのデータを読んでみると

....f.a.c.e._M.i.k.u.A.p...t.g.a.....s.k.i.n._s...b.m.p."...b.o.d.y.0.0._M.
i.k.u.A.p...t.g.a.....t.o.o.n._s.k.i.n...b.m.p.....b.o.d.y.0.0._s...b.m.p.....
t.o.o.n._d.e.f.o...b.m.p."...b.o.d.y.0.1._M.i.k.u.A.p...t.g.a.....b.o.d.y.0.1.
_s...b.m.p

といった具合になっていました。で、PMX仕様.txtを読むと、UTF16がデフォルトっぽいですね。ということは僕らが使っている1文字は2バイトなわけです。多国語対応なんですねー、さすがは世界のミクさん。

あ、ちなみにこの手のデータはデバッグ中に「デバッグ」→「ウインドウ」→「メモリ」で開いたところに「アドレス」って書いてあると思いますので、そこにtexnametestを放り込んでやれば、その周辺のメモリの状態として見ることができます。

アドレス: 0x0015F3C8				列:	自動									
0x0015F3C8	30	0	0	0 102	0 97	0 99	0 101	0 95	0 77	0 105	0 107	0 117	0 85	0f.a.c.e._M.i.k.u.A.
0x0015F3E0	112	0	46	0 116	0 103	0 97	0 20	0 0	0 115	0 107	0 105	0 110	0 95	0 p...t.g.a....s.k.i.n..
0x0015F3F8	115	0	46	0 98	0 109	0 112	0 34	0 0	0 98	0 111	0 100	0 121	0 48	0 s...b.m.p."...b.o.d.y.0.
0x0015F410	48	0	95	0 77	0 105	0 107	0 117	0 65	0 112	0 46	0 116	0 103	0 97	0 0_..M.i.k.u.A.p...t.g.a.
0x0015F428	26	0	0	0 116	0 111	0 110	0 95	0 115	0 107	0 105	0 110	0 46	0t.o.o.n...s.k.i.n...	
0x0015F440	98	0	109	0 112	0 24	0 0	0 98	0 111	0 100	0 121	0 48	0 48	0 95	0 b.m.p....b.o.d.y.0.0_.
0x0015F458	115	0	46	0 98	0 109	0 112	0 26	0 0	0 116	0 111	0 111	0 110	0 95	0 s...b.m.p....t.o.o.n..
0x0015F470	100	0	101	0 102	0 111	0 46	0 98	0 109	0 112	0 34	0 0	0 98	0 111	0 d.e.f.o...b.m.p."...b.o.
0x0015F488	100	0	121	0 48	0 49	0 95	0 77	0 105	0 107	0 117	0 65	0 112	0 46	0 d.y.0.1_..M.i.k.u.A.p...

こういう奴ね。

で、UTF16ってやつは2文字で1文字を表している。よく見ると複数のファイルパスが256の中に入っているので、恐らくは可変長文字列であろう。

では区切り文字はなんなのか?これがわからないと文字列の長さがわかりませんね。C言語使って大抵の場合は¥0というヌル文字で0x00で区切っている。ただし、よく見て欲しい。ここには0x00は無数にある。これはどういうことなのか?

C言語で「0x00は終端」とて習ってる人はここは面食らうだろうね…。何しろUTF16は2文字で1つを表しているつまり0xabcdで一文字なのだ。つまり、b600とかでひとつの文字を表してしたりもするわけで、0x00が終端文字という風には断定できないし、事実ここではそうではないのである。

さて、ここでデータを観察し、作者さんの気持ちになって考えましょうか…。うん、2文字でひとつならもしかしたら0x0000が終端文字なのかもしれない…うん、それは考えうる答えだよね?

だが、そうした場合疑問が残ってしまうのである。

1e 00 00 00 b6 00 b1 00 b3 00 b5 00f.a.c.e

最初のこの部分。そういうことならば最初の4バイトは何なのか?って話やね。ここでもう一度PMX仕様.txtを見てみよう。

材質内で参照されるテクスチャパスのテーブル

4 + n : TextBuf | テクスチャパス

明記はされてはいけないが、ここまで流れから言って、最初の4バイトがバイト数をあらわしているようだと推測してロードしてみる。

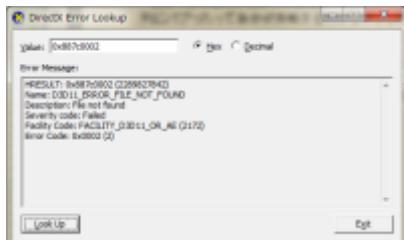
はい!どうやらそれっぽいデータのようですね。カンガレル!人はいきなり4を見てバイト数だと思うでしょうが、たまにこういう寄り道もレル!でしょう。

0000を見てすぐに飛びつかないようにしてくださいね。

さて、ここからテクスチャを読み込むわけなんですが、相手はUTF16…こんなんで本当に読みめるのか?

論より証拠。読み込んでみましょう。答えを聞くまで動けない臆病な人がいるようですが、別にバグったって自分が怪我するわけじゃないし、鬼に切って間違えましょう。

はい、やった結果 0x887c0002 が返ってきました。こいつは FILE_NOT_FOUND です。ちなみにどうやったら分かるかというと、エラー表示プログラム書いてもいいんですが、説明するの面倒なんで、DirectX SDK の Utilities の DirectX Error Lookup に放り込んで下さい。



さて…どうするよ…これ。

で、これは知らないややれないと思うんですけど、Windows 系の API にはほぼ必ず 2 バイト文字に対応するような API が用意されているんですねー。

これは Direct3D であっても例外でない!!!

どうやって探すのでしょうか？簡単あります。文字列を渡す系の API の名前の最後に W を入れてみましょう。D3DXCreateShaderResourceViewFromFile であれば、この最後に W をつけて

D3DXCreateShaderResourceViewFromFileW とスルノデス!!

これで読み込めるようになります。これで失敗する場合は TGA に失敗してるんでしょうねー。

あと、C 言語で 2 バイト文字を扱う際は

```
char ⇌ wchar_t
std::string ⇌ std::wstring
"abc" ⇌ L"abc"
```

という対応関係になっています。はい、ロードは出来ると思いますので、やってみましょう。次にマテリアルです。

PMX のマテリアル

- 材質数 (int)
- 材質 * 材質数

こんなふうになっていて、一つ一つの部分は

4 + n : TextBuf | 材質名
4 + n : TextBuf | 材質名英
16 : float4 | Diffuse (R,G,B,A)
12 : float3 | Specular (R,G,B)
4 : float | Specular 係数
12 : float3 | Ambient (R,G,B)
1 : bitFlag | 描画フラグ(8bit) - 各bit 0:OFF 1:ON
 0x01:両面描画, 0x02:地面影,
 0x04:セルフシャドウマップへの描画,
 0x08:セルフシャドウの描画,
 0x10:エッジ描画
16 : float4 | エッジ色(R,G,B,A)

4 : float | エッジサイズ

で、ここまでとrogenk、

n : テクスチャIndex サイズ | 通常テクスチャ, テクスチャテーブルの参照 Index

ここね。このnに注意してね。これはヘッダの1バイト列3番目のテクスチャIndex サイズである。

大抵は1バイトか2バイトなんだけど、ここは臨機応変に対応してくれ。プログラム上では unsigned int で用意して1,2,4でロード/バイト分岐するだけでいいんじゃないかなと思います。

注意するのはここくらいですね。ここができるれば。



はい、TDA ミクさん出力出来ました。必要な物は知識？根性？違います…愛ですよ、愛!!!まあ
こいつのボーンはものごつい面倒なんだけどね(・•ω•`)



透過とゾノツファについて

既にちょいちょい話していると思いますが、ゾノツファと透過処理については嫌な関係があります。(たいていのMMDのモデルはその辺を考慮してか半透明テクスチャの部分の描画が後ろに来るよう設定されているみたいなのでモデル単体だったなら問題ないんですが…)

どういうことなのかというと、テクスチャやサーフェス(面)のマテリアルの半透明が有効である場合、見た目が不自然になることがあります。なぜ、これが起きるのか?

半透明の処理の場合、既に描画された情報と今から描画しようとする情報の合成を表示することになりますが、半透明以前にゾテスト(深度テスト)といって、既に書かれているものより向こう(ゾが遠い)にあるピクセルは描画しないというルールが、ステンシルゾノツファを設

定した時点で決められています。

これが厄介な事態を引き起こしています。

何故かというと、こいつは半透明物体であっても同様に適用されるからです。

半透明物体だったら何がいけないのか？

よくイメージしましょう。

半透明物体は、向う側(Zが遠い)にあるピクセルを描画しないんでしたよね？

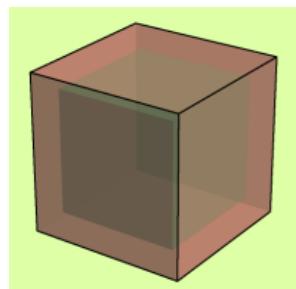
ということは既に最前面に半透明物体がある場合、Zテスト(深度テスト)によって、はなつから描画されないということが起こります。

これはよろしくないです。

いくつか回避方法はあるので、簡単なものから紹介していきます。まずは半透明物体がビルボードや2D(つまりプロジェクション変換していないもの)である場合、半透明のものは後回しに後回しにして書いていけばいいのです。また、その際にはC++側で中心座標をとっておいて(大体でいいです)、その座標とカメラ座標の距離でソートしてその順番で描画するようにしてやればいいです。ビルボードですから基本的に重なることは無いと思います。

まあビルボードや2Dであれば自分でそのへんは制御できるので問題ありません。また、ポリゴン全体が半透明であれば、不透明のものの後に描画すれば問題ないでしょう。

ただしこういう半透明の中に半透明があるような物体はどうするでしょう？まあこういう場合はいっそのこと一時的にZバッファ向こうにしてやるものもありでしょうね。別に向こう側が見えたって構わないわけですし。



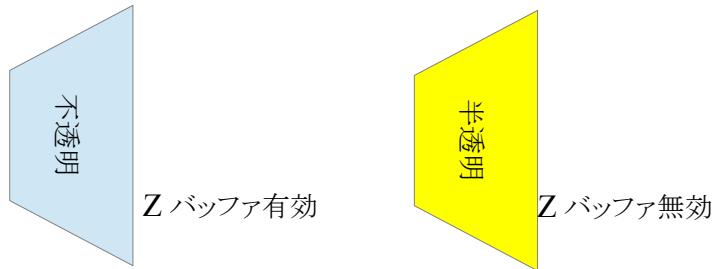
ところが、透過テクスチャを内包して、ピクセル単位で透過だのの状態になればもうお手上げになります。

ピクセル単位でどうこうするには…最終問題の解決とまでは行きませんが“2パスレンダリング”という手があります(これが正解というわけではなく、DX11の機能を知っていくともっと便利で確実なやり方もあるはず)。

ともかく半透明のものが不透明のものより後に描画されるようにしてしまうのです。つまり、2枚描画用の画面(レンダーターゲットを2つ)を用意しておき、片方には不透明なものをレンダリング、もう片方には半透明のものをレンダリングしておきます。

で、最終的に描画する際に、先に不透明、そして後から半透明のものを描画することにより、半透明のものがそれなりにそれっぽく描画されているように見せることができます。

で、このとき、不透明のものを描画するレンダーターゲットでは深度バッファを有効にして



おり、半透明のものを描画するレンダーターゲットでは Z テストを無効にしておきます。これの手順を一応書いておくと

1. レンダーターゲットを最終表示用以外に 2つ用意
2. シェーダ側の出力(ピクセルシェーダ)が 2つの情報を出力するよう変更
3. 合成する部分をコーディング

こんな感じでしょうか?

ちなみに Z テスト設定をいじるには

D3D11_DEPTH_STENCIL_DESC でこまけこと書いて

[http://msdn.microsoft.com/ja-jp/library/ee416082\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee416082(v=vs.85).aspx)

CreateDepthStencilState で ID3D11StencilState を作って

[http://msdn.microsoft.com/ja-jp/library/ee419789\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/ee419789(v=vs.85).aspx)

OMSetDepthStencilState でセットする。いつものパターンですね。

まずはこいつを使って今表示されている Z テストを無効にしてみましょう。

で、あなた方のご想像通り

depthstencildesc.StencilEnable

をいじれば Z テストは無効化されます。この辺を動的にいじつといて切り替えるように書いておけばいいと思います。

さてでは 2つのレンダーターゲットを作りましょうか。もう言わずとも作れるとは思いますが、ついでに言うと、こういういくつかのテクスチャをレンダーターゲットとして使った上で、そのテクスチャをシェーダリソースとして使うことにより様々な合成をすることを **マルチパスレンダリング** と言います。

今回は 2 パスレンダリングですね。

で、今回作成するテクスチャはレンダーターゲットとしても使うし、シェーダリソースビューとしても使いますので、テクスチャを作る際には 2 つを組み合わせた指定を行います。

BindFlags = D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE;

のように縦棒 | で区切って指定することになります。もしデプスステンシルで使っているやつをシェーダリソースとして使いたいなら

```
BindFlags = D3D11_BIND_DEPTH_STENCIL | D3D11_BIND_SHADER_RESOURCE;
```

とします。

専門的なことはともかくこのようにして作ったテクスチャからはレンダーターゲットビューが生成できるし、シェーダリソースビューとしても使えるってわけです。

ここでビューについてひとこと言っておくと、特定のリソースをどのように見るのかっていう…うーん、顕微鏡のレンズみたいなもんかな？まあ、そういうイメージである。

で、ここからがちょっと面白い…と感じるかめんどいと感じるか(´_ゝ`)ワーンかわかりませんが、結構これは面白い考え方ですね。とにかくデプス値をそのままテクスチャにするとこう…



わかりづらいかも知れませんが、手前にある方が濃く、遠くにある方が薄い色になっています。ちなみに本来はもっとわかりづらく、この画像は

```
float b= pow(dep.Sample(sample, v.uv), 256);
```

のように 256 乗に引き伸ばしてやっとこの輝度差が出ています。この後でこれを行う(レンダーターゲットを分ける、深度値をテクスチャ化する)方法を教えますのでついてきて下さい。

2/パスレンダリング

さっきも言いましたが、最終的にデプス/ドッファをテクスチャとして表現するために、まずはデプス/ドッファの BindFlags に D3D11_BIND_SHADER_RESOURCE を追加しておきましょう。で、他にもイジるところがありまして…Format のところを TYPELESS つまり、

```
DXGI_FORMAT_R32_TYPELESS
```

にしておきましょう。これは何故かというと、今回使用するフォーマットがデプス/ドッファのためにも使用でき、シェーダリソースとしても使用できるようにするためです。

ただし、このままでは今まで使用してきたデプスステンシルビューが失敗してしまうため、
device->CreateDepthStencilView(dtex, NULL, &_dsv);

としていた部分も変更する必要があります。NULL の部分には D3D11_DEPTH_STENCIL_VIEW_DESC の設定値を入力してやります。

設定値は大したことありませんが…とりあえず、いまさら説明してもアレなんで書いておきます。

```
D3D11_DEPTH_STENCIL_VIEW_DESC dsvdesc={};  
dsvdesc.Format=DXGI_FORMAT_D32_FLOAT;  
dsvdesc.ViewDimension=D3D11_DSV_DIMENSION_TEXTURE2D;  
dsvdesc.Texture2D.MipSlice=0;
```

ディメンジョンとフォーマットはきっちり設定しないといち、デプスステンシルはこけます。要は単なるバイト列の塊の扱い方をきっちり教えてく必要があるってことですね。ここまで書けたら、この設定を CreateDepthStencilView に割り当ててあげます。

その次はこのデプスに使ったテクスチャからシェーダリソースビューを作っていくのですが、CreateShaderResourceView の第一引数であるテクスチャリソース。これには二通りの取得方法があります。ひとつはデプス作った時のテクスチャ情報を覚えておいて

```
device->CreateShaderResourceView(dtex, &srvdesc, &_depthSRV);
```

なんていう風に設定してやる。モウヒトツは既に作っておいたデプスステンシルビューから取得する方法です。

```
ID3D11Resource* dres;  
_ds->GetResource(&dres);
```

この例では _ds がデプスステンシルビューになっています。ここで取得したリソース情報から

```
device->CreateShaderResourceView(dres, &srvdesc, &_depthSRV);
```

とやって上げればステンシルとして使用しているテクスチャをシェーダリソースとしていつでも見ることができます。

で、ここからが本題です。まだまだレンダーターゲットは一つであり、1パスレンダリングです。次に行うべきことはレンダーターゲットを増やすことです。

デプスステンシルビューを作った要領でテクスチャを作ります。ここで作るのが1/1パス目のレンダーターゲットとなります。今までのが2/1パス目のレンダーターゲットとなるのです。

デプスステンシルで作ったのとの違いは RGB ありますので、Format が DXGI_FORMAT_R8G8B8A8_UNORM になりますし、BindFlags は D3D11_BIND_RENDER_TARGET に加えて D3D11_BIND_SHADER_RESOURCE を指定します。当たり前ですが、指定の追加は演算子を使用します。わかんない人はマルチパスレンダリングは諦めましょう。

で、テクスチャができたら、例によってレンダーターゲットビューとシェーダリソースビューを作りましょう。

```

//レンダーターゲット作成
D3D11_TEXTURE2D_DESC rendertexdesc;
rendertexdesc.Width = 800;
rendertexdesc.Height = 600;
rendertexdesc.MipLevels = 1;
rendertexdesc.ArraySize = 1;
rendertexdesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
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;
device->CreateTexture2D( &rendertexdesc, NULL, &rtex );
D3D11_RENDER_TARGET_VIEW_DESC rtvdesc={};
rtvdesc.Format=rendertexdesc.Format;
rtvdesc.ViewDimension = D3D11_RT_V_DIMENSION_TEXTURE2D;
rtvdesc->CreateRenderTargetView(rtex,&rtvdesc,&rtv2);

//シェーダリソースビュー作成
D3D11_SHADER_RESOURCE_VIEW_DESC srvdesc={};
srvdesc.Format=rendertexdesc.Format;
srvdesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
srvdesc.Texture2D.MostDetailedMip = 0;
srvdesc.Texture2D.MipLevels = rendertexdesc.MipLevels;
device->CreateShaderResourceView(rtex,&srvdesc,&path1);

```

はいはい、ここまで書けたらつまり、1/パス目のレンダーターゲットを手に入れて、そいつらをテクスチャとして読み取る準備ができたということです。次にやるべきことは現状のレンダーターゲットを変更するところです。

現在はまだスワップチェーンから持ってきたレンダーターゲットを OMSetRenderTarget で書き込み先として使用していると思いますが、一時的にこいつを先ほど作った新しいレンダーターゲットに変更します。つまり

`OMSetRenderTargets(1,&newRTV,DSV);`

と言った具合になります。こう変えただけだと画面には何も映らなくなります。当然ですね…スワップチェーンとつながってないやつをレンダーターゲットにしたんですから…。

だから潔く、2/パスレンダリングをやらないといカンのですよ。

2/パスレンダリングをやるのはいいんですけど、これは要は1/パス目のレンダリングをテクスチャとして使うってだけの話なんで、そいつを貼り付けるペラ紙が必要です。

昔つくったでしょ？ペラ紙。そいつを復活させる時が来たのです。とりあえず pos と uv があれば事足りるので

`struct TEXVERT {`

```

    XMFLOAT3 pos;
    XMFLOAT2 uv;
}
```

的な構造体型の頂点を4つ作っておきます。もちろんこれ用の頂点レイアウトは定義しなければいけませんよ？

で、4頂点作ったらいつものように頂点バッファを作つておきます。さらにこいつを表示するためのシェーダも作つておきましょう。

`struct PeraV {`

```

    float4 pos:SV_POSITION;
    float2 uv:TEXCOORD;
};
```

こんな感じの構造体を作つておいて

`PeraV PeraVS(float4 pos:POSITION, float2 uv:TEXCOORD) {`

```

    PeraV v;
    v.pos=pos;
```

:

てな感じの頂点シェーダを作つておきます。とりあえずは頂点変換はしなくていいです。同様にピクセルシェーダは

```
return float4(tex.Sample(sample, v.uv));
```

書いておけばいいと思います。あ、勿論、こいつらもシェーダコンパイルして頂点シェーダオブジェクト、ピクセルシェーダオブジェクト作つておいてくださいね。できない人はマルチパスレンダリング諦めましょう。

さて、ここまでできていれば準備ができます。

ということでメインループの最初で

```
OMSetRenderTargets(1,&newRTV,DSV); //新しく作った方
```

のコードを書いておいて下さい。これは後々意味がわかります。あ、この後で ClearRenderTarget があると思いますが、モチロン新しい方をクリアしておきましょう。

で、いつもどおり DrawIndex をやつていると思いますが、その描画コードと Present 関数の間でレンダーターゲットを入れ替えます。

```
OMSetRenderTargets(1,&originalRTV,DSV); //元々のレンダーターゲット
```

さて…ここからが2パスになります。

とりあえずはシェーダリソースを設定しますが、1パス目に使つたレンダーターゲットと関連があるシェーダリソースビューを

```
PSSetShaderResources(0, 1, &pass1ShaderResourceView);
```

と言つた具合に設定します。ピクセルシェーダと頂点シェーダは先ほど作ったやつを割り当てますので、

```
dcs.Context() -> VSSetShader(&peraVS, NULL, 0);
```

```
dcs.Context() -> PSSetShader(&peraPS, NULL, 0);
```

こんな感じになるでしょう。あとは既に作つておいた頂点ノッタフアを割り当ててあげればペラ紙ポリゴンが表示されますので、元々の絵と同じものが表示されると思います。

変わらないようですが、実は2パスレンダリングになつています。疑う人はペラ紙頂点シェーダに対してWVP 変換を有効にしてやればいい

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

ですね。これでわかると思います。ただし小さくなりすぎる可能性がありますので、これをやるときにはペラ紙自体を大きくしておいたほうがいいでしょう。

さらに言うと、今回は元々のレンダーターゲットをそのまま表示しましたが、デプス値も取つてこれまで、こいつをテクスチャとして使用すると深度値が色の濃さとなって現れます。但しそのままだと色の差が薄いため、POW 関数などである程度大きさに表現することをお勧めします。

```
float b = pow(dep.Sample(sample, v.uv), 256);  
return float4(b, b, b, 1);
```

ここまでやれば最初に見せたようなアレが表示されると思います。

さて、ここまで単なる前置きであります。2/パス2/パスと言っておりますが、実際は経路が1本道(IPPONDOU)なので2/パスって言っていいのかっていうとビミョー…なのよねー。

というわけで、今からが本当の2/パスレンダリングです。皆さん心してかかってきなさい。

真・2/パスレンダリング

さて、皆様……疑問に思ったことはないでしょうか?どうして

OMSetRenderTargetSなのか…と OMSetRenderTargetSなのかと…どうして

OMSetRenderTargetSなのかと…ねえ、レンダーターゲットが複数形なんてありえないって思います(^ω^)おつ。

そうね、ここまでついてきてる人ならもう、なんとなくわかってるんだよね。そう、そういうことなんだ。ピクセルシェーダから複数のレンダーターゲットに出力することが可能なんだよ実はね。

仕方ないのでレンダーターゲットを2つ作りましょう。仕方ないんです。どうせ配列で渡さなければならないので、配列でもしくはベクターで宣言しましょう。

で、現在のレンダーターゲットのための CreateTexture をモウヒトツ作ってあげて、そいつを2つのレンダーターゲットとシェーダリソースビューに割り当ててあげます。

これで2つのレンダーターゲットができました。あとは OMSetRenderTargetS に2つわりあってあげればいい。

デブスの方は1個でいいのでそのままいいですが…

dcs.Context() -> OMSetRenderTargetS(2, &dcs.PassRTVs()[0], dcs.DepthStencil());

で、な感じになるでしょう。

このままだと何にも変わりません。これでは面白くありません。一回 SV_Target を見直しましょう。

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

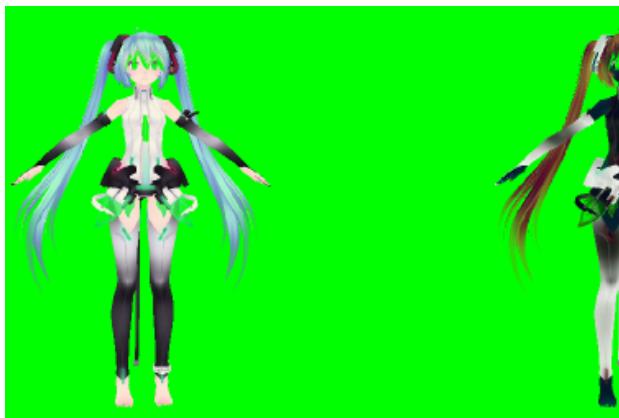
ここに書いてますが、SV_Target0～SV_Target7まで使えます。今回は0と1だけ使いますので、2種類出力します。つまり

出力は今は SV_Target だけですが、複数にするためにこういう構造体を作ります。

```
struct P_Out {
    float4 t0:SV_Target0;
    float4 t1:SV_Target1;
};
```

で、それぞれのピクセルシェーダの出力を2種類出力するようにします。

例えば、反転したのをそれぞれに出力し、それぞれのテクスチャを貼り付けたペラ紙を表示すると以下の様な表示になります。

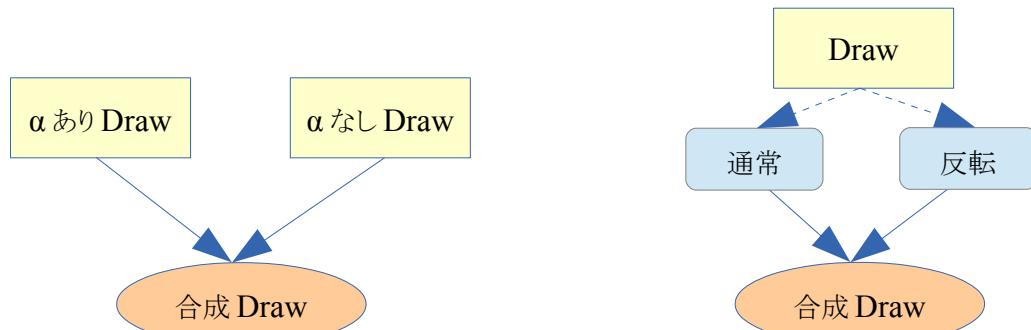


これがマルチパスレンダリングの第一歩です。これが色々なこと(例えば影とかの表現)のために使えるようになります。ちなみに右の「反転ミクさん」が見切れているのは、画面の大きさのペラ紙しか用意していないので、2つ重ねようとするのですが…まあ表示がキレてしまうわけですよ…。ちなみに出力コードの一部を示すと、こう…

```
p.t0=float4((rgba*v.diffuse).rgb,alpha);  
p.t1=float4(((float4(1,1,1,1)-rgba)*v.diffuse).rgb,alpha);  
return p;
```

ともかくこれで2つの演算結果が2つのレンダーターゲットに帰るわけです。この時点(Draw系関数呼び出し後)で、シェーダリソースビューの内容は変わっていますので、そいつを使って色々やっていくというわけです。

また、ちょっと置いてけぼりだったアルファブレンディングに関してですが、ここまでやつてればなんとか分かるんじゃないでしょうか?



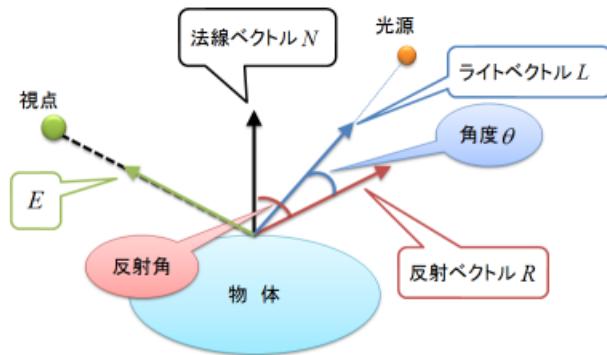
図のような感じです。

まあ、ホントしばらくはこういうのを使う場面には遭遇しないとは思いますが、知っとかないと使えないテクニックも多いので、使えるようになっておきましょう。このハッテン系がデファードレンダリングになってくるわけです。

スペキュラを実装しよう

ここまでディフューズ(拡散反射)とアンビエント(環境光)をやってキマシタワーですが、これにスペキュラ光を追加してみましょう。

スペキュラは前にも見てもらったようにキュピーンって感じの光を表すもので



視線ベクトルと光線ベクトルによって計算されるものです。基本的には視線ベクトルの反射ベクトルと光線ベクトルの内積(つまり $\cos \theta$)をホニヤララ乗することによって表わされるものです。

もしくは光線の反射ベクトルと視線の内積のほにやらら乗比例ですね。

つまり…視線の反射ベクトルを R 、光線ベクトルを L_i としてスペキュラの強さを s とする
とスペキュラ光 L_s は

$$L_s = (\vec{R} \cdot \vec{L}_i)^s$$
 と言った具合になります。実際にはデータから持ってきたスペキュラ成分 K_s をかけて

$$L_s = K_s (\vec{R} \cdot \vec{L}_i)^s$$

てな具合ですね。

なお、HLSLにて反射ベクトルを計算するには reflect 関数を使用すればいいです。反射ベクトルくらいは法線ベクトルさえ分かってれば自分で計算できるんですけど、とりあえず楽しましょう。

[http://msdn.microsoft.com/ja-jp/library/bb509639\(v=vs.85\).aspx](http://msdn.microsoft.com/ja-jp/library/bb509639(v=vs.85).aspx)

で、スペキュラの解説をしたわけだけど、車等のモデルならいざしらず、ニンゲンモデルにはあまり適用されていねのがこのスペキュラだったりもします。なぜかというと金属っぽくなりますからね。

あと、適用されているとすれば「瞳」なんですが、これも最近はテクスチャ貼り付けることが多かったりしますのでね…

結構な量のスペキュラ成分が0,0,0に設定されていますからねー、まあ、あまり使われていなければつがわれていませんね。あと、大体スペキュラ成分もスフィアマップで表現されていくことが多いことMMDにおいてはスペキュラの恩恵は今ひとつって感じです。

豆ちしきー

エラーへの対処

エラーが発生するたびに手が止まる人が居ますが、エラーは発生するものです。「エラーが出たからお手上げ」みたいに思っちゃう人はプログラマに向いてないでしょ。

ここで言っているのはコンパイル時エラーではなく、実行時のエラーのことです。たぶん、慣れてくると「エラーが出るような不具合であるだけマジ」とか思うようになるでしょう。

とりあえず、エラーが出た箇所から「原因を推測」しなければいけません。判断能力、推測能力が問われますので、頭をフル回転させて臨みましょう。

で、今回の話はエラーに対処するためのオタスケとなる下準備の話です。

リザルトの活用

まず「HRESULT」を返すものはこまめにリザルト値をとっておきましょう。オッケーの時は S_OK が返されますが、そうでない時は色々と出てきます。

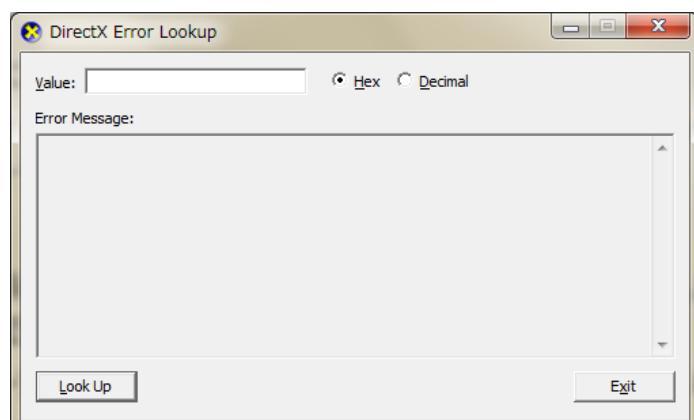
で、この戻り値が E_INVALIDARGS(引数が不正ですって意味)とかわかりやすい値だったらいいんですけど -216542398 みたいな値が返ることがあります。

あと、この辺のメッセージはたいてい英語なので、エラーで返される英語くらいはよめるようになってください。

なんだったら Google 翻訳にかけても構いませんし、E_INVALIDARG で Google 検索かければいくらでも出てきますし、その他のエラーも同様なので、それくらいの対処を自分でした上で質問して下さい。

で、ここでお助けツールを紹介します。

スタートメニュー→DirectXSDK→DirectXUtilities→DirectXErrorLookup をクリックすると



こういうのが出ますので、Value のところに戻り値の数値を入れておいて下さい。そ

うすれば英語ですが説明が出てきます。このときHexがデフォルトになっていますが、
-205434636みたいな数値の場合はDecimalの方に合わせておいて下さい。

これで大抵のエラーの意味はわかるでしょう。

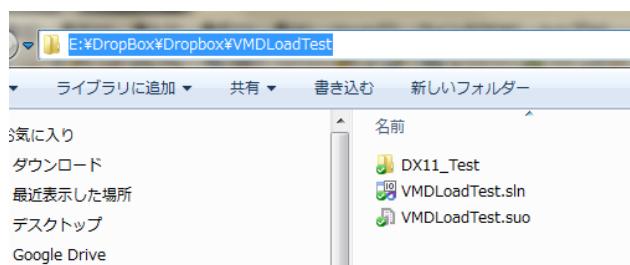
シェーダコンパイラ

次によくあるのがCompiledShaderの値が0x00000000であるために発生するクラッシュですが、この場合は大抵「シェーダー」にバグがあります。というか現段階ではこれしか考えられないでしょう。

で、シェーダ自体のエラーを特定するにはどうしたらいいのでしょうか？以前に紹介したかもしれませんのがfxc.exeを使う方法があります。

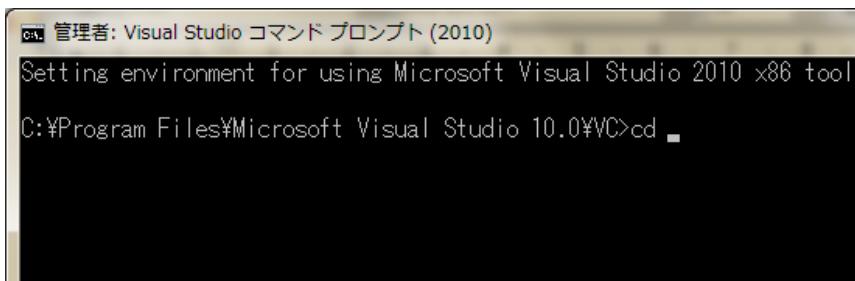
C:\Program Files\Microsoft DirectX SDK (June 2010)\Utilities\bin\x86

とかの中にfxc.exeってのがありますので、そいつを取ってきて自分のシェーダスクリプトがあるフォルダに放り込んで下さい。



で、コマンドプロンプトで自分の作業フォルダに移動して……。移動の仕方がわからないというITリテラシが非常に著しく低い人がいるみたいなので一応言っておきます。エクスプローラの作業フォルダの場所を全選択した状態でコピーします。

コマンドプロンプト上でcdと書きます。その後にスペースを開けておいて下さい



で、その状態で右クリック→貼付けを選択してEnterを押してください。

その状態で、

```
fxc.exe basic_shader.fx
```

とかいてEnter押してくれればシェーダ側のエラー内容がわかります。Warningの方は無視しても構いませんが、Errorと書いてる所は明らかにエラーなので、そこを修正しましょう。中学生程度の英語がわからなくても行番号はわかりますよね？行番号は出でますからその辺におかしいところがないかチェックしましょう。

errorオブジェクト

もしくは、CompiledShader作成の際にerrorオブジェクトを持っているのであれば、error->GetBufferPointer()関数から取得したポインタのアドレスにメッセージが書き込まれているので、それを参照して判断して下さい。

```
char* errorormsg=(char*)error->GetBufferPointer();
```

とでもしてやって、errorormsgを見ればエラー内容書いてあるので、それを確認しておきましょう。

あなたが中学生以上の脳髄を持ち合わせているというのなら、この程度の労力もかけず聞いてこないで下さい。お願いします。とは言え、一番イケナイのは、八方にも尽くさず、聞くこともせず、**ただただ手を止めるというのが人として最低**なので、それだけはやめましょう。

そういう意識で生きてるとすぐニート等の「他人に迷惑をかける存在」になることでしょうから今のうちから目の前の問題を意地でも解決する意識をつけておきましょう。

あと、ステップ実行の件ですが、F9でブレークポイントを置いて、もつかい同じ場所でF9押すと解除されます。

で、F10で1ステップずつ実行されます。F11を押すと呼び出している関数の中に入っています。

とりあえず自分である程度調査した上で人に聞きましょう。チョット動かなくなったらスグ人に聞くクレクレくんは2ch上だけでなく、色んな所で嫌われます。

VisualStudio のお便利なツール

VisualStudio にはお便利な機能が満載されています。これを使わずにクリエイティブなデバッグが出来るとと思うながれ。前回までの話でステップ実行(ステップオーバー、ステップイン)はできるようになったと思います。

それだけではないのですよ便利な機能は…。

まずご紹介するのは「出力」ウィンドウ。こいつはデバッグ→ウィンドウ→出力で見ることができます。

「知ってるよそんなもん。」

ええ、まあ、そうでしょうね。

このウィンドウに情報(つまり文字列)を出力できたらデバッグがしやすくなるとは思わないかい? 例えばなんかしらの変数の変化を観察したいときなど…ねえ? 昔の人はこういうのを「printf デバッグ」と言ったものなんだけど、残念ながら printf で出力すべきコマンドプロンプトはない!…

するにはどうすればいいのでしょうか? うん、こいつは知らないと見つからないと思いますが、OutputDebugString 関数を使います。使い方はいたって簡単

OutputDebugString と書いて、文字列をセットするだけ!! その上お値段はタダ!! ご利用はお早めに。

まあ、文字列出力は結構コスト食うので使えば使うほど重くなりますけどね…モードで切り替えられるようにしておくとか、バグった時に一次的に使うのであれば大いに役に立つと思います。

また、実行時のメモリリークなど有用な情報もこの「出力」ウィンドウに出力されますので、「どうも動作がおかしい」などという人は是非ご利用ください。

さて、次にご紹介いたしますのはこの「呼び出し履歴」。これは読んで字の如し、呼び出しの履歴でござる。ワケワカラ…という人も多いと思います。昔はこれ「コールスタック」とか読んでたんですが…基本情報とかで聞いたことないかな?

使い方はいたって簡単。とりあえずクラッシュしたり、ブレークポイントでストップした時点で「デバッグ」→「ウィンドウ」→「呼出履歴」を開きます。

例えばクラッシュしたポイントが STL の中だった!! よくありますよね~。こんな時に「呼出履歴」!!

フレーム	行	言語
VMDLoadTest.exe!CreateBoneTree(int boneNum)	行 1	C++
VMDLoadTest.exe!WinMain(HINSTANCE__ * hInst, H	行 2	C++
VMDLoadTest.exe!_tmainCRTStartup()	行 547 + 0x2	C
VMDLoadTest.exe!WinMainCRTStartup()	行 371	C
kernel32.dll!75c611940		
[下のフレームは間違っているか、または見つかりま		

こういうウィンドウが出てきたら、自分の見覚えのあるコードが見えるようになるまで、それぞれの行をダブルクリック!! お目当ての画面が見つかったら早速調査!!

これであなたのデバッグルイフは快適なものに!!一家に一台「呼出履歴」。

次に紹介するのは「条件付きブレークポイント」

恐らくブレークポイントってのは一番お世話になる機能だと思いますが…ループの中に仕掛けるとなると面倒ですよね。128回ループするとして、128×ループ内処理行数の回数だけF10を押さなければなりません。これでは日が暮れちゃいますね~。

そんなあなたに「条件付きブレークポイント」

使い方は至って簡単。既に仕掛けているブレークポイントにマウスポインタを合わせて右クリック→条件を押せば



こんな画面が出てきますので、ここにストップしたい条件を入れます。条件ってのはif文の中身みたいなのでオーケー。これだけで、F10を押す回数は激減!!

ご利用はお早めに!!

次に紹介いたしますのは「データブレークポイント」です。こいつは実行時にしか使えません。なぜかはスグわかると思います。

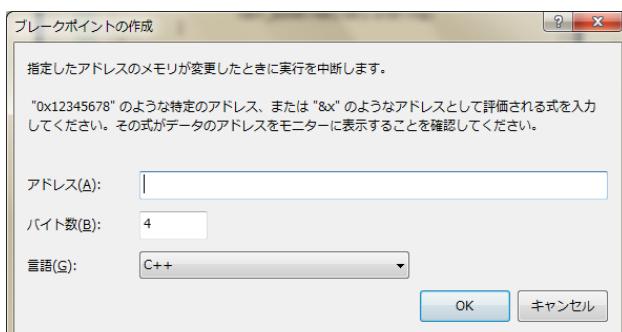
データブレークポイントっていうのはヒトコトで言うと、**特定のアドレスの値が変更された時にプログラムが中断する**というものです。

というわけで、この機能の性質上、実行時にしか設定できません(アプリケーションは起動時にそれぞれの変数等のアドレスが決定されるため)。

使い方はいたって簡単?

まず、デバッグ→ウィンドウ→ブレークポイントを開いて下さい。

んで、「新規作成」→「新しいデータブレークポイント」を選択してください。



そうしたらこういう画面が出ますので、ここにアドレスを入力しておきます。でOKを押して、中断しているプログラムを動かします。

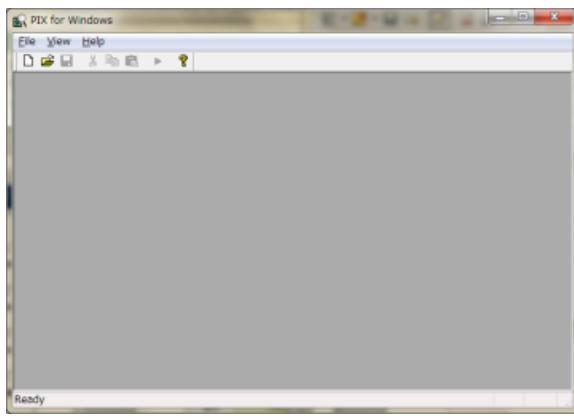
そうすると、入力したアドレスが変更されたタイミングで処理が中断されるので「あれ？ なんでこんな状態なんだろう？ どこで壊しちゃったんだろう？」ってなった時に効果を發揮します。

PIX for Windows

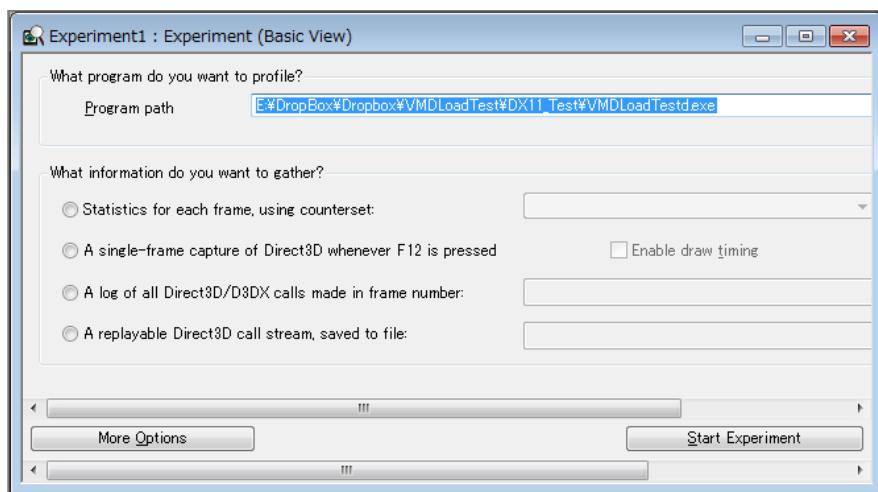
Direct3Dには「**PIX for Windows**」などという便利な…ある意味便利すぎるツールがあります。どちらかと言うと **GPU の挙動を見る**ためのものだと言えます。

場所はスタートメニュー→DirectXSDK→Utilities→Pix for Windowsです。

これをクリックすると今回ご紹介する PIX for Windows が立ち上がります。



こういうのが立ち上がると思います。わけわかんないかもしませんが、これが強力なのです。まずはFile->New Experimentを選択して下さい。



こんなのが立ち上がると思いますので、Program path の部分にチェックしたい exe のパスを入れて下さい。

で、A single-frame なんとかを選択。Enable draw timing にチェックを入れる。

これで準備完了です。

(Start Experiment)

を押してください。立ち上がるとレンダーの状態とか見れますので、非常に役に立ちますし、パフォーマンスレポートも出ますので、後々シビれると思います。

SVってなんや？

SV_POSITION やら SV_Target やらは

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

によれば、…たぶん System Value の略やね。

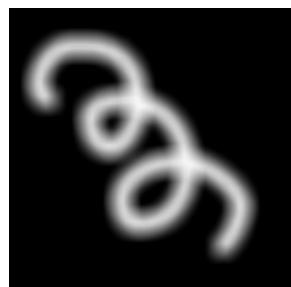
凸凹の進化

コンピュータグラフィクスにおいて微細面の表現は日々進化している。まずはパンプマッピングから始まる。パンプマッピングとは高さ情報(ハイトマップ)をグレースケールテクスチャ(0~255)として保存しておいて、そいつを物体表面に貼り付けることにより相対的な高さを表現できる。

ただし、これで表現できる範囲には限界があり、その後もう少しカシコイ方法が考案された。それが法線マッピングである。

法線マッピングとは法線ベクトル(x, y, z)を無理やり(r, g, b)に変換することにより法線の「揺れ…要は表面の凸凹による法線のはらつき」をテクスチャとして保存するのである。

なので、ハイトマップがこういう



白黒の256階調データであるのに対して、法線マップは以下の図のように



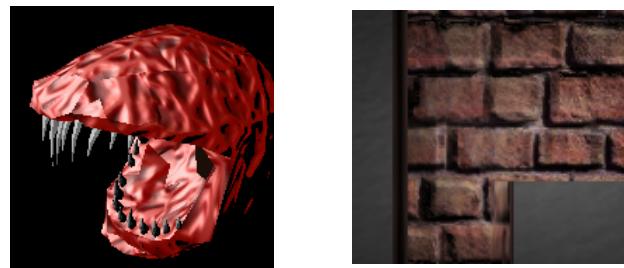
いかにもRGBで法線を表現しましたアーツ!!って感じのテクスチャになっている。要はこれを物体表面に貼り付けて、シェーダで適切に計算してやることにより物体表面が凸凹しているような見た目になるのである。

ここまでがCG検定の教科書レベルだ。ゲームプログラマとしては「常識」レベルである。だが技術者はこの先に行く。

「結局凸凹のない表面に貼り付けてるだけだから、近くで見るとモロわかりだよね。」

そう、人間の目というものは大したもので、ぱっと見騙されたようでも無意識の何処かに不自然さを感じるものなのである。

というか、自分で法線マップしてみればわかるが、ある程度以上の深さを持つ凹凸の表現はやはりしょぼいのである。



このような爬虫類系生物表面の凹凸や掘りの薄いリリーフ紋様等を正面から写したものなら十分に表現できるのであるが…

石畳のように、ガツツリ凹凸が入っているものなら、そのしよぼさが露呈する。なぜならば、こういう凸凹は法線の差異による濃淡の差によってのみ表現されているため、ちょっとキツイ角度から凸凹を眺めると不自然に見えてしまう。もうチョット言うと、法線の濃淡によってのみふくらみへこみを表現するために、やはり何処か平坦なのである。

これをなんとかして改善しようというので考えだされたのが、**視差マッピング**(パララックマッピング)である。

こいつは法線マッピング的アプローチではなく、バンプマッピング(ハイトマップ)的なアプローチであるため、法線マップと組み合わせればよりよい凸凹の表現が期待できる。

よく引き合いに出されるのがこういう例え



斜めからこういう模様を見た場合、視線の角度と距離の関係上、

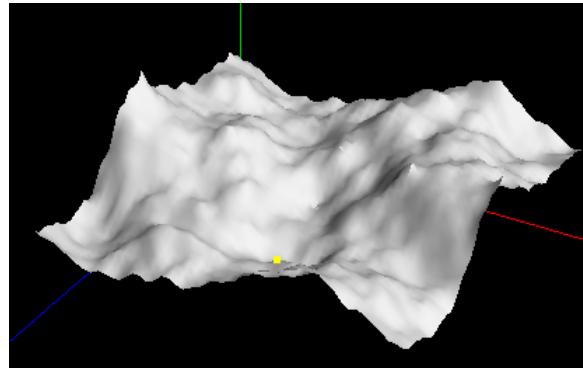


図のIよりもIIの方が自然に見えるのである。これを実現するのが視差マッピングである。

だが、CGの高速化とハードの進化は年々スゴイことになっており、PS4やらXBoxOneの時代(現代)になると「もう実際に凸凹させちゃおうぜ」ということになり、**テセレーション**の技術と**ディスプレースメントマッピング**を用いてこのような凸凹を実際のポリゴンとして自動生成しちゃおうという…そういう力技が可能な時代になってきている。

ちなみにディスプレースメントマッピングとは、高さ情報を0~255のグレースケールで表されている…バンプと同じ画像から、実際の高さを変化させちゃおうというものだ。わかり易い例はメタセコの「オブジェクト」→「生成」→「凸凹地形」を見てみれば分かるだろう。実

際あれを用すれば…



フルポリゴンでこのような地形を表現することも可能なのである。だが流石に現段階で全てのシーンの細かい凸凹をこれでやってのけるのは限界があるため、ディスプレースメントマッピングと視差マッピング+法線マッピング等のハイブリッドになっていくのだろう。

「ディファードレンダリング」

英語で書くと Deferred Rendering なんて、本当は「ディファードレンダリング」の方がいいと思うんですが、一般的には「ディファード」で表記されますね。

Deferredってのは「延期された」とか「遅延した」とかいう意味で

Differedだと「異なる」って意味で全然違う意味になるんで…まあ、納得行かないんだけど仕方ない。

日本語で言うと「遅延レンダリング」ってやつです。高速化テクニックのひとつですね。とりあえず2パス以上のレンダリングになってしまったため、その話をしてからのはうがれりいかもしれないけど、

大雑把に言うと「時間かかる系の処理を省いた状態」のものを先に別のテクスチャにレンダリングしておきます。

で、時間のかかるライティングなどの処理を2パス目以降に回し(遅延させ)ます。で、この間に様々な処理を別々のテクスチャに書き込んでおき、最終的に合わせてレンダリングするって手法です。

ですから「ディファードレンダリング」で検索すると



Wikipedia 見ると、

<http://ja.wikipedia.org/wiki/%E9%81%85%E5%BB>

[%B6%E3%82%B7%E3%82%A7%E3%83%BC%E3%83%87%E3%82%A3%E3%83%B3%E3%82%B0](#)

色々と書かれておりますが、専門的なことはともかく、ライティングなしで一回レンダリングしといて、後からライティングを合成するといった手法が一般的な感じですかね。

ちなみに、シェーダ系のオススメの書籍はこういうやつです。

<http://goo.gl/uynhkG>



薄い本ですが6200円と高いので、万人にオススメはできませんが、初歩からはじまってテクニカルな説明、ディファードレンダリングまで網羅しているので、これ一冊で事が終わります。まあ僕が持ってるのここでは貸しますけど家で読みたい人はお金を溜めよう。

設計なんかクソつ喰らえ

設計について聞きたい人がいるっぽいのですが、設計とか、変に考えない! もうがイイですよ。とりあえず、

- 設計に時間をかけ過ぎない!(考え過ぎない!)
- いつでも自分のプログラムを捨てる(作ったものに執着しない!)
- 暇な時にSourceForgeなどで評判の高いプログラムを眺める
- とにかく(手当たり次第)色々作る
- 読みやすいプログラムを心がける(…1年後の自分が読んでもすぐ分かるような)
- 無駄に難しいテクニックを使わない!
- できるだけ拡張可能、変更可能、削除可能な状態にしておく
- シンプルさ>堅牢さ>実行速度
- わざわざ遅いコードを書かない!
- どつかのサンプルを鵜呑みにしない!
- 日頃からいろいろな言語に触れておく

だけは鉄則として守っておけばオッケー。

Doxygenとか

難しい設計とかウンヌン考えるくらいなら、完成させるほうが最優先事項だし、どつかの会社で作業するつもりならば最低限度のわかりやすさだけ考えておきましょう。

で、ここで紹介するツールは Doxygen…です。

<http://www.doxygen.jp/>

こいつはソースコードの中に「**特定のルール**」で書いたコメントと、関数およびクラスを抽出してドキュメントを作ってくれるというスグレモノのツールなのです。

わりかしプロのゲーム開発の現場でも使用されているツールですね。もっときちんとした現場であれば git や subversion 等とこの Doxygen とバグトラッカーを組み合わせたようなサーバツールの redmine とか trac などが使われています。

SourceForge のようなフレームワークですね〜いです。

ちなみに「特定のルール」ってのも簡単です。関数の頭にコメントを書くとして、通常は// や/*～*/でコメントを書いています

//や/*～*/でコメントを書くと Doxygen というツールが、この部分を抽出して関数に紐付いたドキュメントを書いてくれます。

こいつのスグレモノな所は関数の呼び出し依存関係や、クラスの親子関係等を記述したクラス図などを残してくれ、プログラムの規模が大きくなればなるほど役に立ちます。

例えばこういう風に記述します。

```
///@brief IKリストデータを読み込みます。  
///@param fp ファイルポインタ  
///@param ikmap IKマップデータへの参照  
///@attention fpのカーソルはIKリストデータ先頭であること  
void ReadIKLists(FILE* fp, std::map<std::string, IKList>& ikmap, std::vector<BoneInfo>& boneInfo)  
{  
}
```

@brief やらはそれぞれ意味がありますが、そこは各自マニュアルを参照してください。
@brief は概要、@param は引数説明、@attention は注意すべきこと

てな意味です。

さらに構造体やクラスにも適用できます。

```
///@brief IKリスト  
struct IKList{  
    unsigned short boneIdx; ///  
    unsigned short tboneIdx; ///  
    unsigned char chainLen; ///  
    unsigned short iterationNum;  
    float restriction;  
    std::vector<unsigned short> boneIndices;  
};
```

//**K**って書くとその行の変数の説明を記述できます。

…このくらい覚えてたら十分でしょ。

最終課題

最終課題です。

1/16(木)放課後までに、ここまで

¥¥132sv¥gakuseigame¥ゲーム¥DirectX11¥最終成果物
に提出して下さい。

いつもどおり

学籍番号_氏名

というフォルダを作つて、そこに自分の成果物をアップロードして下さい。

なお、アップロードする際の注意点ですが、sdf ファイルやilk フォルダなどの無駄なファイルは削除した上でアップロードして下さい。

*アップロード時の注意

他人のファイルを消さないように注意しましょう。

どうしてもプログラムが苦手で、単純なモデルも表示できない!可哀想なオトモダチのためにプログラムコードのスクリーンショットを

¥¥132sv¥gakuseigamero¥rkawano¥2~3年向け課題¥どうしてもできない人のためのスクショ

においていますので、どうしてもできないって人はそれを丸写ししてください。それだけで落第は免れます。どうぞどうぞお使いになつてくらはい。