後期ゲーム制作

僕の基本スタイルは

こういう事です。

ナメてるとフツーに単位が落ちると思ってください。

まどの道ここで武器を磨いておかないと『就職というモンスター』に踏み潰されてしまいますので『ここをセコく切り抜ければいいんでしょ?』みたいに思ってると、もっとひどい結果が貴様等に齎されることになるであろう。

目次

1 制作の流れについて

とにかくハイスピードでやっていきます。「ついていくこと」がある意味「課題」であり「試練」であり「ハンター試験」です。

流れにするとこんな感じです

1.授業の流れの説明と知っておくべき知識の概要と色々と下準備

2.前期のおさらい(C++,Direct3D)

3.DirectX11 への移行(初期化はコードを配布)

4.基本的なシェーダの解説＋実習

5.PMD モデルのロードとモデル表示

6.XAudio でサウンドを組み込む

7.シューティングゲーム①(スターフォックス的なやつ)

8.シューティングゲーム②

9.ボーンの解説とモデルの変形

10.VMD のロードとモーション反映

11.ベルトスクロールアクションゲーム①(ファイナルファイト的なやつ)

12.ベルトスクロールアクションゲーム②

13.ベルトスクロールアクションゲーム③

14.Effekseer 組み込み(エフェクトを作る)

15.影、その他ポストエフェクト

既に説明があったかもしれませんが…こんな感じです。とりあえず授業ではゲームはスターフォックスとファイナルファイト的なやつしか作りませんので、その他のジャンルが作りたい人は自主的に作っておいてください(質問は受け付けますが、わかんねー事もありますので、その場合は別のセンセーとかグーグル大先生に聞いてください)

ヒトコトでいうと「簡単じゃねえから覚悟しとけよ？」ってことです。

とりあえず最初に「なぜ C++と DirectX11」を題材にしたのかという話をしておこうか。

まず、なぜ C++かって話なんですけど、結局のところ未だに「C/C++」を募集要項にあげている会社が多いんですよね。コンシューマ系(家庭用とかアーケードね)はモチロンのこと、スマホゲームとかソーシャルゲーム系までもが C/C++やっとけって言ってるのが現状。

んじゃあ会社に入ったらホントに C/C++で開発するのかって言うと、実のところ Unity だの UnrealEngine だので開発することが多いんだ。では何故企業は C/C++なんて言っているのかというと、ゲームエンジンを使おうがメモリやハードウェア周りの事をよ～く知っておく必要があるわけだ。

その辺を C#や Java では自動でやっちゃうから Unity とかだけで制作されると、そのへんが見えてこない→募集要項に C/C++って書いてあったり、C/C++やってる？とか聞かれたりするわけ。

勿論コンシューマではガッツリ C/C++なので、それを掘り下げていくことになるけど…まぁ要は「やっとけ」ってことよ。

さて、次に DirectX11 だけれども、なぜわざわざ難しいこいつを選んだのかというと、流石にもう DirectX9 は古すぎるからだ。いや、確かに DirectX9 のゲームは今でも多数存在するんだけど、ゲームプログラマを目指すのならば「DirectX9 は古い」というのは知っておいたほうが良い。

そしてそれは企業の方もそう考えると言う事で、やっぱり新しいのを使用している方が「あ、コイツはアンテナ張ってるな」ってなるわけ。

あと…もう一つ付け加えておくと、DirectX11は最初からシェーダを書くことを要求してきます。つまり「嫌でもシェーダに親しむことになっちゃうよ」ってことやね。どうしても難しそうなことには人間目を背けちゃうからね。もう半強制的にシェーダを使わせたいわけ。

あと、DirectX11は相当にややこしい。グラフィックスパイプラインの構造がわかっていないと、妙な事で不具合が発生するし、バイナリの特性も知っておかなければ手に負えない。

…それくらいにややこしい。

例年、これをやると挫折者が多数出る。

1.1 欠課、居眠りは命(就活生命)にかかわるぞ

だから前期に言ったことをもっと強調して言っておく。

「授業を休んだら相当の努力をしない限り、まず置いて行かれると思え」

言っておくぞ!!**相当**だぞ!?

あと

「寝たら死ぬぞ」

ここに来ている人は、ゲーム会社を目指していると思います。しかもコンシューマ系のゲーム会社……でしょう？

寝たり休んだりした時点で……というかそういう甘い考えの時点で夢は絶たれ、課題は提出できず…時間と労力を無駄にして何も残らない。

中途半端な努力ってのが一番勿体無いことになる…意地でも寝るな…!

あと、これも前期に散々言ったことだけど、何度言っても分かってない奴ってのは出てくるもので……今一度言っておきます。

とりあえず自分がついていけてないなー、遅れてきたなーって思ったら…

「分からんかったらすぐに聞け！」

「遅れたらその日のうちに取り返せ！」

「周りの奴らの助けを借りろ！！！」

今回は自ら**助けを求めない奴をイチイチ助けません。**置いていきます。来年度は就職年次ということはあなた方もよくご存知のはず…

さて、ここでもう一度考えて欲しい……あなたは本気ですか？本気でプログラミングが好きなんですか？コンシューマのゲームプログラマになりたいですか？

とりあえず作るものを作らない人は本気とみなしませんので、そこは覚悟しておきましょう。

1.2 就職に向けて

「楽しく制作」していって欲しい。楽しく制作をしていって欲しいが、ゲーム会社志望の人で、現段階だと「実力が足りない」って思う人は

時間を投資

してほしい。プログラム組める奴と組めない奴の違いってのはつまるところ、プログラミングに「かけた時間」の違いです。

はっきり言っちゃうと「できる奴」ってのは圧倒的時間をプログラミングに充てているのです。現段階で「遅れている」って思ってる奴が、今までの習慣のまま中途半端にやっていっても結果は変わらないし、時間と労力の無駄ですよ？

就職活動まであと半年～1年くらいだし…その間くらいは色々なものを我慢して(我慢しなくとも時間を決めて(減らして))プログラミングに充ててください。

1.3 日報

仕事のシミュレーションということで日報を書いてもらいます。

http://goo.gl/forms/ZZYaQb5YmQ

毎回毎回その日にやったことを書いてもらいます。大したこと書かないので、授業の残りの3～5分で書いてください。

1.4 注意事項

ダメ人間の心理って奴ぁ…難儀なもんで、自分が分からなくなってくると「もっとがんばろう」って思えばいいのに「周りの奴らを引きずり下ろしたる」っていう情けねー人間がいるんだわ。

ここはゲームプログラマを本気で目指す人らが集まっているのでそんな人間混ざってないと思いますけどな？

一人で腐ってるだけなら一人が単位落ちで済みますが、授業を妨害したり周囲のやつの邪魔をし始めるなら「公共の場所での迷惑行為」ってことで退出してもらいますからな？あ、モチロンその場合は15分以上席を外してたら「欠課」にしますからな？よろしくな～？

2　制作の準備

Git とかそういうのを活用して計画的に作っていこう。

きちんとしたスピードで開発していけるよう、計画を立てて、またその計画に沿って開発が進んでいるのかを測定していき、遅れていれば放課後の時間などを使用して遅れを取り戻すことが大切です。

『早い段階で「遅れ」に気づく』というのが大切です。末期になればなるほど取り返しがつかなくなりますので注意しましょう。

今一度言います。

「末期になればなるほど取り返しがつかなくなりますので注意しましょう」

早期発見が大事です。

でも、人間、そう簡単に問題には気づかないものです。

そこで先人たちが色々とツールとか思考法を考えてくれているので活用しましょう。

「ガントチャート」

「トラッキングシステム」

「ソースコード管理システム」

さて、そのための便利なツールが

2.1 Git について

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

- 「変更したらなんかバグった」時に、原因を特定しやすくなるぞ!!

- ボタン一発で昔の状況に戻ることができるぞ!!
- チーム開発の時に「責任の所在」を明らかにできるぞ!!
- 「差分」を保存していくスタイルなのでハードディスク容量を節約できるぞぉ!!

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

一ヶ月前の自分のコードは他人のコード

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

まぁ大した手間がかかるわけでもないのでぜひ使いましょう。

2.1.1 基本的な使い方(ローカル)

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

とにかく使ってみよう

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

まず、みなさんの DropBox の中にフォルダを作って…まぁここでは DxGit という名前にデモしておきましょう。このフォルダで右クリック

「リポジトリを作成」

とします。

Bare がどうこう言われることがありますが、無視します。

これで準備ができました。

で、例えばここの中に test.txt とか置いて「追加」して、「コミット」します。

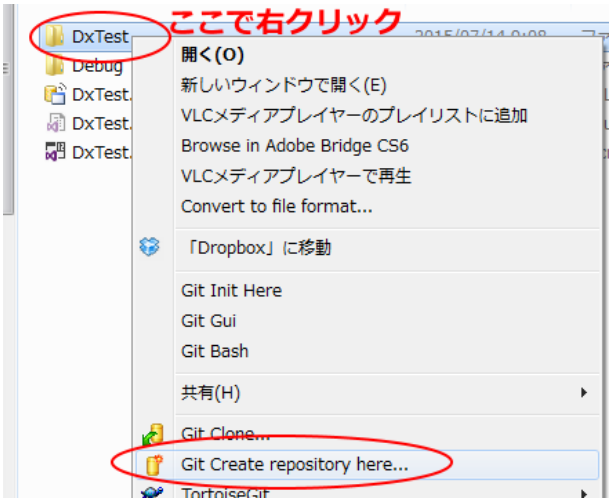
これで履歴ができました。

次に変更してください。そしたら、差分ってのがあると思います。

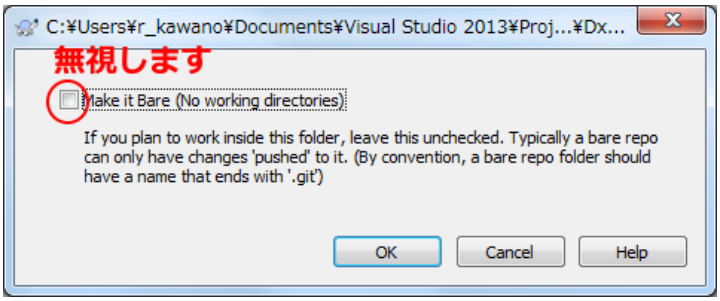
その場合どの行が変更されたのかとかの情報を見れます。

これを繰り返していくことにより、いつ、どこで、どんな修正がされたのかなどという情報を見ることができますので、いじった結果おかしくなった場合など、「どこを変更したのか」というのを行単位で知ることができます。

興味がある人は自分が今開発しているフォルダのcppとかのソースコードがあるフォルダを右クリックして、

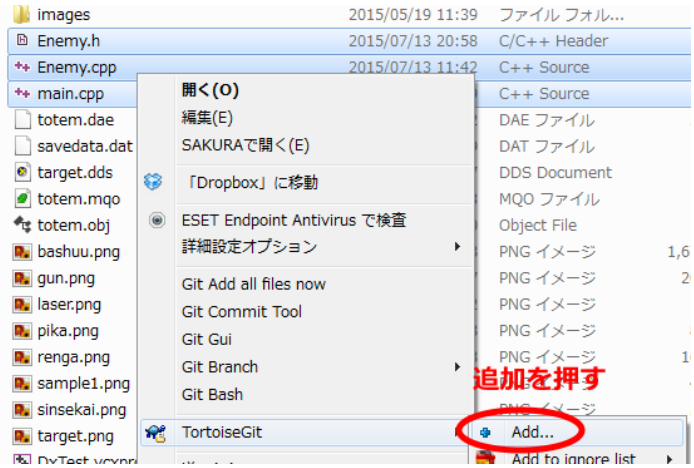


僕のところは日本語化パッチ当ててないから、みんなと表示が違うと思いますがだいたいわかるでしょ？



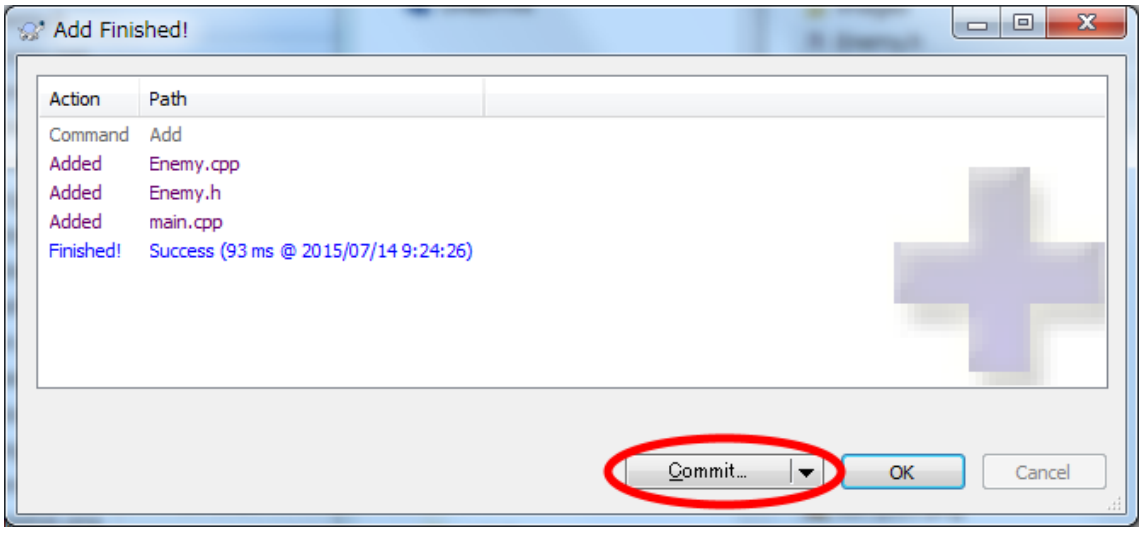
これでリポジトリが出来上がりました。

んで、基本はソースコードのみをバージョン管理していくので、



h と cpp ファイルを選択して追加を押してくだしあ。

で、コミット



おめでとう!!これで君のプログラムはバージョン管理されたのだ!!!

あとはここから

ソースコードを変更→動作確認→コミット

のループを繰り返す。

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

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

2.1.2 基本的な使い方(サーバー)

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

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

GitHubを使ってみよう

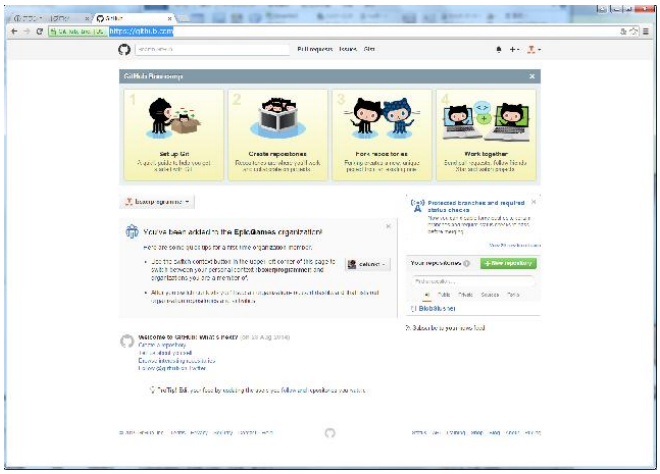
とりあえずGitHubっていうサイトがあるんだけど、そこは通常「リポジトリ」(チームで作るのに必要なGitのサーバ置き場)を作るのにお金がかかる。

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

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

https://github.com/

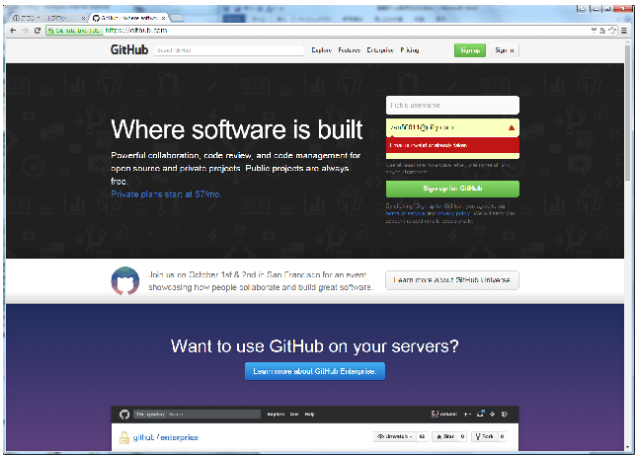
に行けば、



こういう可愛らしいサイトがあるとおもいます。

あ…

もしかしたらアカウントを作ってないから…



こんな感じになっていたりするかもね。

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

右側のメールアドレスに自分のメアドを入力して「Sign up」だ。

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

さぁ、サインアップし給え。

さて、そうやってサインアップして、自分の情報を入れたら準備完了だ。

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

UnrealEngine のアカウントは作っているよね？

作ってない!?

じゃあそっちも急いで作っちゃおう。

作ったら

https://github.com/EpicGames/UnrealEngine

からのクローンを作ってみよう。

どうだい？

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

そしたら次はBacklogを利用したいところだけど、まだチーム開発とかやってないのでそっちはちょっと後回しにしよう

Backlog を使ってみよう

<<そのうち書きます>>

3 前期のおさらい

とりあえず前期のおさらいとして、C++について…と、Direct3D の解説をしましょう。後期のこの授業においては以下の事はわかっている前提で進んでいきますので、分かっていない人は自主的にお勉強しておきましょう。

- C++の文法(構造体、クラスの作り方、ポインタ、参照など)
- ポインタ
- オブジェクト指向
- バイナリファイルについて
- 再帰とかのアルゴリズム
- Direct3D における行列とかベクトルの扱い
- グラフィクスパイプライン
- UV について、UV アニメーション
- ビルボードについて
- 参考になるもの、Web サイト

こっから先はあまりにも「度を越して分かんねー人」とか「真面目にやんねー人」には構ってる時間ないんで、よろしくな？邪魔したらクラスのみんなから呪われますよ？

3.1 C++文法

3.1.1 関数

C++では関数を単品で作ることができます。記法は

関数の宣言

戻り値の型 関数名(型名１ パラメータ１,型名２ パラメータ２ ....);

もし戻り値が必要ない場合は **void** を戻り値のところに書いてあげます。

また、パラメータが一つもない場合は省略できます。

つまり、戻り値もパラメータもない関数宣言は

```
void Func();
```

となります。

では、関数の定義ですが、

```
戻り値の型 関数名(型名１ パラメータ１,型名２ パラメータ２ ....){
        return 戻り値;
}
```

なお、戻り値がない場合は return 文を省略できます。

整数型の引数をそのまま返す関数を定義するのならば

```
int Func(int param){
        return param;
}
```

てな感じで定義します。

3.1.2 配列

C++では配列は

```
型名 変数名[配列要素数];
```

と宣言します。

基本的な利用方法としては

```
配列名[インデックス]=値;
```

とか

変数=配列名[インデックス];

とか

関数(配列名[インデックス]);

てな感じで角カッコの中に整数インデックス値を入れることで使用できます。

また、C++言語ではカッコつけずに配列名を書くと、それは配列の先頭要素のアドレスを指し示すことになっています。

3.1.3 アドレス演算子

アドレス演算子は&と書きます。

C/C++では変数とか関数の頭に&をつけることで、その変数のアドレスを指し示すことができます。

int a=10;

とかって宣言しておいて

&a

とやれば、変数aのアドレスを得られます。C/C++では変数のアドレスを利用することが多いので、覚えておきましょう。

3.1.4 間接演算子

間接演算子とは*と書きます。

C/C++言語ではこの後にお話する「ポインタ」に関連するものとして「間接演算子」ってのがあります。

「ポインタ」ってのは基本的にメモリのアドレスを指し示すものです。そのメモリアドレス先が指し示しているものを値に変換する時に使用します。

たとえば

int a=10;

int* p=&a;

ここでpはポインタなので、変数aのアドレスを指し示しています。

このｐそのものではｐの中身の値は分かりません。

値を知りたければｐの左にアスタリスクをつけて

cout<<*p<<endl;

とやれば、10 が出力されます。

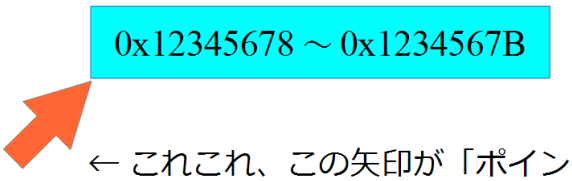
3.1.5　ポインタとアドレス

さて、ポインタですけれども、ここはいわゆる難関と言われているところなので、ここだけで1コマ使っちゃう勢いなのでさらっと行きます。

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？



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ょう。

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。何がちゃうかというと

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。

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。

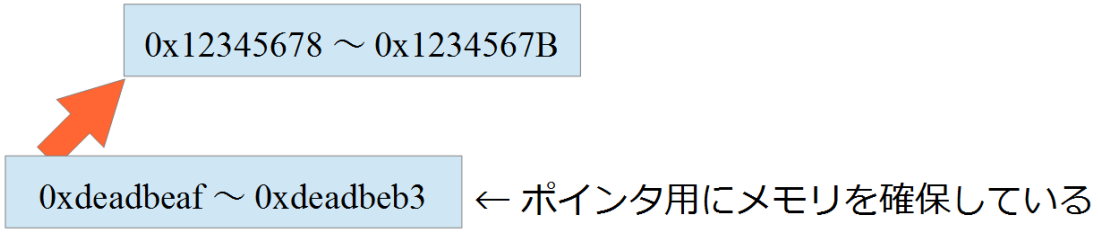
そうすると、当たり前の話なんだけどポインタを宣言した時点でまたメモリ上に4バイトは確保されているってことなのね。

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、そっから4バイト確保されることになる。



そう、ポインタであっても変数である以上はメモリをほんの少しだけ食いつぶしている意識は持って下さい。

例えば、アドレスがこういうものであるとする。

表みたいなもんやね

NULL									
					a=5				
							c=0x3F		
			b=0x0F						

この例では 0x12345678 とかっ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;

とかって書いたとする。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。

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かのアドレスに確保されているわけや。つまり b にもアドレスがあると…。今回の例だと 63 にある。16 進にすると 0x3F。

これも変数に入れるならどっかに入っていることになる。

int* c=&b;

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 b のこと。

DirectX9 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。

わかりにくいかな？

b=NULL にしていると仮定する。

で、下のような関数がある。最後の引数がポイントなのね。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、ポインタのポインタで渡された 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。

とりあえず「ポインタ」のおさらいについてはこんなものでいいでしょう。

3.1.6　参照

C++には「参照」ってのがありましたね？

「参照」というのはポインタのようでポインタでない、でもちょっとだけポインタぽい、でも見た目は普通の変数…

という一風変わったものであります。

まず使い方を言っておきます。

参照の宣言は…

型& 変数=参照先；

とします。型の横に「アンパサンド&」をつけます。これでこの変数は「参照型」となり、ポインタのように最初に割り当てたオブジェクトを指し示すようになります。

ただし、ポインタと違って2つのルールがあります

- 必ず参照先を指定してやる必要がある
- 一度参照先を指定すると参照先の変更はできない

このルールにはメリットデメリットが有ります。

メリット

「参照はさし示す先が存在することが保証されている」

「最初に設定した参照先であることが保証されている」

これは結構重要です。

ポインタなどの場合は、自由にアドレスを指定できるため、指し示す先が存在しないなど、結構困ったことを引き起こします。

デメリットはこれの裏返しですね

「融通がきかない」

です。

融通がきかないけど、関数の引数として使う文には非常に役にたちます。

とにかく使ってみましょう。

```
int b=90;
int& a=b;
```

とでも書いてやってあげると、a は b そのものを指します。ポインタと同様 a 自体には中身がなく、ただただ b の中身を見ているだけです。ですから上のコードを書いた後で

```
a=50;
```

とかって書くと b の中身が 50 に変更されます。

例えば、

```
void CheckHitEnemy(Player& player,Enemy& enemy){
        if(IsHit(player.Rect(),enemy.Rect()){
                player.Damage();
```

```
        }
}
```

などと書くと、ポインタを使わずに、この関数の中で player や enemy そのものの値を変化させることができます→つまり呼び出し元の変数が変更されるってことです。

今まで何度も関数内で値を変更したいがために「ポインタのポインタ」を使うことがありましたが、「参照」を知っているとそこはちったあマシに書けるようになるということです。

例えば

```
void func(int* p){
        *p=10;
}
int q=20;
func(&q);
cout << q << end;//qは10を出力します
```

こんなことにいちいちポインタを使いたくない。そういう場合はそっと*を&にします。そうすると

```
void func(int& p){
        p=10;
}
int q=20;
```

```
func(q);
cout << q << end;//qは10を出力します
```

3.1.7 構造体の作り方

文法は

```
struct  型名{
        型1  要素1;
        型2  要素2;
          :
          :
};
```

という風にして、幾つかの型をまとめます。

とりあえずこれは今はめんどくさいだけかもしれませんが、そのうちこの便利さが効いてくるので覚えておきましょう。

3.1.8 namespace と using namespace

namespaceは日本語では「名前空間」という。直訳やね。

namespaceに関してはC#のnamespaceとほぼおなじ意味。だけどC++の場合は「とにかくまとめる」「区別する」ってくらいの意味。

例えば自分がint Max(int a,int b)などという関数を作っていたとする。

で、こんなもんクラスのメンバにしたくもないしいちいちオブジェクト生成したくないし、staticメンバにするのも嫌)という場合があるわけだ。

そうしたら自然と、クラスのメンバにせずに関数単体で宣言するだろう？

その場合に問題が発生することがある。

何故かと言うと、他のライブラリとか使った場合に「名前がかぶる」ことがある。モチロンこれはエラーになる。

かといって OreMax なんてするのもバカバカしい。さてどうしようかといった時に、namespace を使用する。

namespace の使い方はいたって簡単。namespace と書いて、namespace 名を決めて、あとは囲むだけである。

```
namespace 名前{
	ここに他のライブラリとかと特別したい関数を宣言しとく
}
```

これだけ。ねっ？簡単でしょう？

さて、これと同じような事を DxLib や Effekseer や STL やらがやっている。

```
namespace DxLib
{
	//中略
}
```

や

```
namespace EffekseerRendererDx11
{
	//中略
}
```

という風にだ。

じゃあ例えば、Ore なんていう namespace 作って、その中で Max ていう関数を定義し、それを使いたいとする。

どうするのかというと

```
namespace Ore{
        int Max(int a,int b){
                return a>b?a:b;
        }
}
```

などという風に定義する。この関数を外部から使うには4つの方法がある。

- その①自分自身が Ore の namespace の一員となる
- その②スコープ解決演算子を使う
- その③using を使う
- その④using namespace を使う

のどちらかの方法でアクセスできるようになる。更に言うと「自分自身が Ore の namespace の一員となる」には4つの方法がある。

まずは Ore の一員になってみましょう。

```
#include<iostream>

namespace Ore{
        int Max(int a, int b){
                return a>b?a:b;
        }
}//end of namespace Ore

namespace Ore{

int main(){

        std::cout << Max(2,3) << std::endl;


        getchar();
```

```
        return 0;

}
}//End Of namespace Ore
```

一応…コンパイルは通ります。通るのですがたぶんこの例だとリンカで失敗します。

なんでか？

main 関数の鉄則として

「main 関数は必ず一つだけ存在しなければならない」

ってのがあります。

ところがこの例の場合は main 関数が Ore 名前空間内に入っているため、コンパイル後の名前は

Ore::main()

ってなってるわけで、main 関数が存在しないといってリンカエラーを吐きます。

次に、スコープ解決演算子の話ですが、これは C++でよく見かける::のことです。

詳しくは口述しますが、C++では名前空間やクラス名の入れ子によって、とある関数や変数が直接指定できないことがあります、

その場合に、入れ子をたどっていくための演算子が::なのです。慣れるまではちょっと難しいかもしれませんが…。

ともかく、先ほどの例の場合だと Ore 名前空間内の Max 関数なのだから

Ore::Max(1,2);

などのように記述すればいいわけです。

ねっ、簡単でしょ？

おそらくこの使い方がメインにはなってくることでしょう。

次に using を使います。

これは、特定の関数や変数の名前を呼んだ時には、とある名前空間のやつを使うという約束事を宣言するものです。

```
Int main(){
        Using Ore::Max;
        Max(1,2);
        Max(3,4);
}
```

とか記述しておくと、main 関数内で Max を使った時には Ore::Max を使っているよと暗黙に宣言しておくわけです。

いくつも使わなければならない時には有効かもしれません。

しかしこれは後述するようにちょっと注意が必要です。

ちょっと先に、using namespace について説明しておきましょう。

using namespace 名前空間名;

と書く事により、これを書いたのと同じスコープにおいては、無条件で名前空間以下の変数関数が使用されるようになります。つまり

```
namespace Ore{
        int Max(int a, int b){
                return a>b?a:b;
        }
        int Min(int a, int b){
                return a<b?a:b;
        }

}//end of namespace Ore
```

```
using namespace Ore;

int main(){

        std::cout << Max(1,2) << std::endl;//2 が出力

        std::cout << Min(3,4) << std::endl;//3 が出力

}
```

てな風に、特にスコープ解決演算子を使わなくとも Min や Max を使えるようになります。

ところが、この using のメリットはそのままデメリットになって跳ね返ってきます。

つまりいったん using や using namespace を使ってしまうと、それ以降のスコープで Max とか Min を使用すると問答無用で Ore のやつが使われることになります。

これが別の厄介なことを引き起こします。

using namespace 等が使われていることがわからないまま、それ以降の関数呼び出しを行っていると自分でも気づかずに別の挙動を書いている事があります。

この「自分でも気づかずに」って所がデメリットポイントで…まぁずっとプログラマやってると身にしみると思いますが、自分の思ってるのと違う挙動が混ざっているととんでもなくバグが見つけにくいってことになります。

以上の理由から、基本的には using や using namespace を使用するのはオススメしません。

特にぜッタイにやめて欲しいのが

ヘッダ側で using を使用する

これはダメ!!ぜッタイ!!!

まぁ、基本的には「::」を使用しとけってことです。あえてここまで書いたのは「C++の教科書」と呼ばれるものでは頻繁に using や using namespace が最初のサンプルで使われているからです。

気をつけてないと「実務で嫌われるプログラミング」を覚えてしまいますので、敢えて書きました。実務で嫌われるプログラミングはもちろん…就職活動の時にも嫌われるので気をつけてください。

まぁちなみにこういう「構造をぶっ壊す記述」ができちゃうのがC++の嫌われる理由でもあるんですよねぇ～。

3.1.9 C++における::の役割

前述した「名前空間」を解決するために出てきましたが、この::っていう演算子は

「スコープ解決演算子」と言って、入れ子になっている構造をドゥンドゥン降りていくために使用するものです。

「名前空間」だけでなく「クラス(構造体)」や「入れ子クラス(構造体)」に使用することができます。

つまり

```
namespace Ore{
	class A{
		class B{
			void func(){}
		}
	};
}
```

のfuncにアクセスしたいならば

```
Ore::A::B::func();
```

のように使用します。

しばらくはお目にかからないとは思いますが、クラスを多用し始めるとバンバン使うことになります。

3.1.10 クラスの作り方

クラスの作り方は構造体と同じです。C++ではホント構造体と同じです。

違いは…

構造体はデフォルトでぜんぶ public(公開)で

クラスはデフォルトでぜんぶ private(非公開)

ってことだけです。

書き方はこう。

```
class クラス名{
        関数とか変数とか書く
};
```

最後の;を忘れないようにな!!!

んで、ここで「アクセス指定子」ってのが大事なのだ。

「オブジェクト指向」の大事な要素の一つに「カプセル化」ってのがある。

これは「外に見せるものは見せて、見せるべきでないもんは見せない」っていう事を意味しているんだわ。

C#の時にもあったであろう private と public やな。C++の場合は protected なんてのもあるけどな！！！

とりあえず軽く説明しとくと

private で宣言された変数や関数はクラスの中だけで使用でき、public で宣言された変数や関数はクラスの外からも参照できる。

ついでに言うと protected ってのは、宣言されたクラス本人とその子供だけが参照でき、外側からは参照できないという中途半端なヤツや。

C#とは宣言の仕方がちょっとだけ違う。

C#では

```
public void func();
```

こんな使い方だったと思うがC++では

```
class A{
        public:
                int valp;//公開変数
                void funcA();//公開関数
                void funcB();//公開関数
        private:
                int valq;//非公開変数
                void funcC();//非公開関数
                void funcD();//非公開関数
};
```

のように書きます。class Aの中でpublic:を書いた行以降はすべて公開となり、private:を書いた行以降はすべて非公開となります。

そしてこの『クラス』も構造体同様に『型』の一つなので、フツーに型として使えます。

ですから

```
A a;
```

はい、これでもう実体ができました。

できましたので

```
a.funcB();
```

これでオッケーなんです。

えっ?aをnewしていないけど、いいの?って思うのはC#ユーザーですね。C++の場合は構造体とクラスの違いはデフォがprivateかpublicかって違いしかないと言ったはずです。

つまり通常の変数として使用する分にはnewは必要ないわけです。

逆にnewを使用することもできます。できますがその場合は左辺値はポインタ型変数である必要があります。

```
A a;//これはオッケー
A* b=new A();//これもオッケー
A c=new A();//ダメ
A* d;//通るけど、dの中身はありません。
```

ま、そんな感じ。

3.1.11 クラスメソッド

クラスの持っている関数のことを「クラスメソッド」と言います。C言語の場合は構造体は変数だけを中に持てるものの、関数の定義はできません。

オブジェクト指向ってのは「振る舞い」を大事にするので、クラスが関数を持つのは必須なのです。そうC++ならね。

というわけで、クラスのメンバとして関数を定義することができます。

まぁ、フツーに定義すればいいので、

```
class B{
	public:
		void funcB();//公開関数
	private:
		void funcD();//非公開関数
};
```

これで宣言は OK です。ところが実装はどう記述すればいいんでしょう？

2つやり方があります。

一つはクラス宣言時に実装まで書いちゃう事です。

```
class B{
        public:
                void funcP(){
                        funcQ();
                        std::cout << "funcP" << std::endl;
                }
        private:
                void funcQ(){
                        std::cout << "funcQ" << std::endl;
                }
};
```

この書き方も間違いでは…ないです。

とは言えあまり一般的な書き方ではありません。C++では**「宣言はヘッダー側で」「実装は cpp 側で」**っていうセオリーがあります。

で、この実装を書くのがちょっとヤヤコシイんですよ。

例えば B.h というヘッダーに

```
#pragma once
class B{
        public:
```

```
        void funcP();//公開関数
    private:
        void funcQ();//非公開関数
};
```

と書いて、ここでは実装を書きません。そしてB.cppとかの中で

```
#include"B.h"

void B::funcP(){
    funcQ();
    std::cout << "funcP" << std::endl;
}
void B::funcQ(){
    std::cout << "funcQ" << std::endl;
}
```

と書けばfuncPとfuncQの実体を書いたことになります。

なお、B::を書かないとどういう扱いになるのかというと、それはBのものではなく、グローバルなものって扱いになるので、Bのメンバ関数(クラスメソッド)という扱いにはなりませんので注意が必要です。

とりあえず実装側の文法は

```
戻り値 クラス名::関数名(パラメータ){
    //実装
}
```

とでも覚えておいてもらったら良いと思います。

これでクラスのメソッドの実装はできるようになりましたね？

3.1.12 メンバ関数ポインタ

はい、フツーにクラスのメンバ関数を作れるようになったところで、メンバ関数ポインタの話をします。

既に「関数ポインタ」についてはお勉強していると思いますので、特にそっちは説明しませんが、C++にはメンバ関数ってのがアリましたね？

アレのポインタって取れるんでしょうか？

通常の関数ポインタはこうでしたよね？

例えば

```
int funcA(int p,int q);

int funcB(int p,int q);
```

なんていう関数があったとすれば、これを代入できるポインタ宣言は

```
int (*pFunc)(int,int);
```

こういう宣言ができて、

```
(*pFunc)(5,7);
```

のような記述で関数呼び出しができましたね？

この関数ポインタには、実際の関数の名前を右辺値に置くことにより

```
pFunc=funcA;
```

と代入でき、

```
pFunc=funcB;
```

などのように関数を切り替えることができるため、状態遷移(ステートマシン)の実装が容易になるというものでしたね？

さて、では、クラスのメンバ関数のポインタもこのようにできるのでしょうか？

答えは NO です。

例えば

```
class A{
private:
	int Add(int p,int q){
		return p+q;
	}
	int Subtract(int p,int q){
		return p-q;
	}
	int (*pFunc)(int,int);
public:
	A(){
		pFunc=Add;//代入できません
	}
	void Change(){
		pFunc=Subtract;//代入できません
	}
	void Print(int p,int q){
		std::cout << (*pFunc)(p,q) << endl;//アカン
	}
};
```

こういう書き方はオッケーなのでしょうか？

実はメンバ関数ポインタは、フツーの関数ポインタと同じようには扱えません。面倒ですね。

関数ポインタに関数の実体を代入しようとした時にエラーが発生します。

何故か？

とりあえずエラーメッセージを見てみよう

'A::Add': 関数呼び出しには引数リストがありません。メンバーへのポインターを作成する

ために '&A::Add' を使用してください

'int (__thiscall A::*)(int,int)' から 'int (__cdecl *)(int,int)' に変換できません。

エラーメッセージを見てもわけがわからないよ。

ちょっとわかりづらいですけど、C++言語は、メンバ関数ポインタ(静的関数を除く)と通常の関数ポインタを区別しているのです。

つまり

通常の関数ポインタなら

int (*変数名)(パラメータ);

という型で済むのだが、これがメンバ関数の場合はそうはいかない。以下のように解釈される

int (クラス::*変数名)(パラメータ);

なんでこんなヤヤコシイ解釈になるのかというとC++言語のメンバ関数がコールされるとき、隠し引数として、そのクラスの実体自身(this ポインタ)が渡されるからである。

ということで、クラスメンバは暗黙のうちに「誰の持ち物か」という情報がくっついた状態になっており、これでは「型が違う」と判断されてしまうからなのだ。

さて、これに対処するにはどのように書く必要があるのかというと、メンバ関数ポインタが「誰の持ち物の関数を指し示すのか」を明記してやればいい。

つまり先程の例であれば

```
int (*pFunc)(int,int);
```

ではなく

```
int (A::*pFunc)(int,int);
```

である。クラス名を関数ポインタの前に持ってきてやればいい。ちょっとキモチワルイ記述の仕方になるが、まぁ、こういうもんだと思ってくれ。

さて、次にこの関数をコールする時だが、これもまた面倒である。

```
void Print(int p,int q){
        std::cout << (*pFunc)(p,q) << std::endl;//ダメ
}
```

この書き方だと、関数呼び出しの部分でコンパイルエラーが発生します。

ホントにメンバ関数ポインタってのは特別扱いなんだなぁ…。

何がアカンかというと、前にも書いたように、メンバ関数ってのは持ち主のポインタが隠しパラメータとして渡されているため、コールする時に、明示的に「誰の持ち物の関数ポインタを読んでいるのか」を伝えなければなりません。

メンドクサイ

この例だと

(this->*pFunc)(p,q);

なんていうダッセー書き方をしなければなりません。もう文法なので仕方ないのですが、やっぱりC++が嫌われるところってこういう所だよなーっては思う。

ちなみに持ち主がポインタでない場合は

(a.*pFunc)(p,q);

みたいな書き方になる。

で、これで終わりかというと、まだ問題があるようで、このままコンパイルを通そうとすると通らない。エラーメッセージを見るとこうである。

'A::Add': 関数呼び出しには引数リストがありません。メンバーへのポインターを作成するために '&A::Add' を使用してください

よく分かりません…なにこれ。

一応

https://msdn.microsoft.com/ja-jp/library/b0x1aatf.aspx

を見てみたが、古いコンパイラだとこの書き方をしなくてもいいらしい的な事が読み取れる…つまり、安全のために入った文法エラーということだろう。まぁ、これ以上これについて考えるのも無益なので「そういうもん」だと思っておこう

結果として

```
#include<iostream>


class A{
private:
	int Add(int p,int q){
		return p+q;
	}
	int Subtract(int p,int q){
		return p-q;
	}
	int (A::*pFunc)(int,int);
public:
	A(){
		pFunc=Add;
	}
	void Change(){
		pFunc=Subtract;
	}
	void Print(int p,int q){
		std::cout << (this->*pFunc)(p,q) << std::endl;
	}
};
```

```
int main(){
	A a;
	a.Print(1,2);
	a.Change();
	a.Print(1,2);
	getchar();
	return 0;

}
```

こういうコードだとコンパイルが通るメンバ関数ポインタの使い方ってことになります。

さらっと書く予定だったけど、長くなりました。

3.1.13 static

static ってのは静的変数、静的関数を作るものです。

静的って何なんでしょう？何をもって静的と言っているんでしょうか？

例えば関数の中で宣言する変数は、関数呼び出しとともに生成され、関数が終われば破棄されます。そういう意味でダイナミック(static の反対)というわけです。

static ってのはそういう枠にとらわれないやつです。

一度宣言されれば一生(アプリが終了するまで)メモリ上に存在する変数です。

つまり

```
void func(){
	int a=0;
	a++;
	cout << a << endl;
}
```

なんて関数を作って、

```
for(int i=0;i<66;++i){
```

```
        func();func();func();func();func();
}
```

等と書いても、出力される数値は毎回1ですね。これ分からない人は1年からやり直しましょう。真面目な話で。

毎回毎回 a という変数が 0 初期化され生成され、それに++されるからずーっと 1 が出力されます。

ところがこのローカル変数を static にすると話は別です。

```
void func(){
        static int a=0;
        a++;
        cout << a << endl;
}
```

ずっとメモリ上に残り続けるってことは関数を呼ぶたびに、インクリメントが繰り返されます。やってみればわかりますので、やっといてください。

ある意味危険なので、C++言語における static 変数の使用は慎重に…どうしても必要でないかぎりはまず使わないほうが良いでしょう。

さて、本題はここではなくて、static 関数の方です。結構お目にかかるので、知っておいたほうが良いと思います。

例えばメンバ関数を呼び出す場合

```
A a;
```

という風に型 A のオブジェクトを作ったうえで

```
a.func();
```

のように呼び出す必要があります。つまり**「実体」がないとメンバ関数ってのは原則的には呼び出すことができません。**

ところが、static を頭につけると実体がなくてもメンバ関数を呼び出すことができます。

例えばクラス A が

```
class A{
        public:
        void func(){cout << “call func” <<< endl;}
};
```

ならば

```
A a;
a.func();
```

でした。これを単に関数呼び出しだけしたい場合はこの func を static にします。

```
class A{
        public:
        static void func(){cout << “call func” <<< endl;}
};
```

そうすればこの func 単体で呼び出すことができます。ただし A の持ち物ではあるので…そう、「スコープ解決演算子::」を使います。

```
A::func();
```

こんな感じです。

3.1.14 const

const は簡単ですね。定数を作るものです。

```
const int STATUS_DEAD=16;
```

こんな感じです。簡単すぎです。

これが const 定数の使い方です。よく使いますので、きっちり使いこなしましょう。

さて、これを引数で使用すると当然のように引数の数値を変更することができなくなります。

```
func(const int p){
        p++;//エラー!!扱えないよ
}
```

融通がきかなくなります。

これが何の役に立つのか？融通がきかなくなるということは、引数の値が入力時と同一であることは保証されるわけです。

プログラムが複雑になってくればくるほどこの手の「あえて融通を利かなくする」っていうテクニックは重要になるので覚えておきましょう。

さて、こんなもん知ってると思うんで別に本題でもなんでもないです。

本題は const メンバ関数です。

作り方は

型 関数名(パラメータ) const;

です。

おしりに const をつけるって所が新しいですねぇ～。

さて、コイツは何が嬉しいのか？

それは「おしりに const をつけたメンバ関数は"オブジェクトの状態を変更できない"」という仕様があるからです。

オブジェクトの状態ってのは何なのかというと、その関数呼び出しによりメンバ変数の値が変化することはないってこと。

変化させようとするとコンパイルエラーになります。

例えば

```
class A{
        public:
                int a;
                void func()const{
                        int b=10;//OK
                        b++;//OK
                        a++;//NG!!メンバの変更はできません!!!
                }
};
```

なんでこんなもんを使うかというとさっきも言った「あえて制限をかけることで安全にしたい」という思想がC++言語…いやたいていの言語思想にはアリます…この辺に慣れてくると、よりプロっぽいプログラムができるようになるでしょう。

3.1.15 継承

継承も簡単やね。でもC#とかJavaとちょっとだけ書き方が違うので注意。

基本的には…こう

```
class 派生先クラス名 : public 派生元クラス名{
};
```

これが基本的な「継承」の文法です。

C#との違いは「public」とか書いてある部分です。面倒だなあ。C++の場合、これを書いていないとprivate継承として扱われます。

private 継承ってナンヤネン？

うんぃまぁぃそれはなぃ親の全てのメンバが private 扱いになるってことや…。

親の public メンバには子からはアクセスできるんやけどな？

つまり

```
#include<iostream>

using namespace std;

class Base{
public:
	void Baka(){

		cout << "バーカ!!" << endl;

	}
};


class Derived : Base{//public なし継承

	public :
		void Aho(){
			Baka();
		}
};


int main(){
	Derived d;

	d.Aho();//アクセス OK!!


	d.Baka();//アクセス不可！！

}
```

というわけ。

親の Baka にアクセスさせたいのならば

class Derived : public Base{～

と、public を追加します。

なんとも面倒な仕様です。更に言うと所謂「ポリモーフィズム」が働かなくなります。

public をつけていれば後述する「ポリモーフィズム」が働き

Derived d;

d.Aho();

Base& b=d;

こういう書き方は OK なはずなのですが、public が入っていない場合、これは許されません。仕様です。

public 継承していないと…

Base& b=d;//NG!!プライベートクラスはポリモーフィズム使えねーよｗｗｗ

となります。とりあえず皆さんは C++で継承を使いたい場合には「必ず」public をつけとくことを忘れないようにしてください。

3.1.16 ポリモーフィズム

はい、コイツは

「親のフリできる子」

とでも覚えておいてください。つまり継承先のクラスからできたオブジェクトは継承元のオブジェクトのように扱うことがデキるってことです。前述のとおり Derived クラスが Base クラスから継承されていれば Base のフリができるわけです。

例えば

Animal から Cat と Dog が継承されていたとします。

Animalには Bite(噛みつき)とか Scrach(ひっかき)などがあって、それで攻撃をするとシておきます。それぞれ Cat も Dog も同名のメソッドを持ってるとします。

そうすると

```
void Attack(Animal& animal){
        animal.Bite();
        animal.Scrach();
}
```

なんていう関数を作ってやると

Cat cat;

Dog dog;

Attack(cat);//ネコの攻撃(噛みつき&ひっかき)

Attack(dog);//イヌの攻撃(噛みつき&ひっかき)

のようにひとつの Attack 関数を Dog だろうが Cat だろうが区別なく使えるわけです。

これがポリモーフィズムです。

簡単でしょ?

3.1.17 仮想関数、純粋仮想関数

さて、こいつは継承とかポリモーフィズムと関連している話なのだが、Virtual 関数。つまり仮想関数の話をします。

例えばこんなコードを書いたとします。

```
#include<iostream>

using namespace std;

class Animal{
public:
```

```
        void Cry(){
                cout << "知るかよ" << endl;
        }
};


class Cat : public Animal{
        void Cry(){
                cout <<"ニャーン(゜Д゜)" << endl;
        }
};

class Dog : public Animal{
        void Cry(){
                cout <<"わんわんお（ ^ω^）" << endl;
        }
};


void CryAnimal(Animal& a){
        a.Cry();
}

int main(){
        Cat c;
        Dog d;
        CryAnimal(c);
        CryAnimal(d);
        getchar();
        return 0;

}
```

さて…結果はなんて出力されるんでしょう？

おそらく想定しているのは

ニャーン(゜Д゜)

わんわんお(^ω^)

と出力されるのを期待するでしょう。

だが結果は

知るかよ

知るかよ

である。現実は非情である。

C++の仕様として『ただ単に継承しただけでは「現在の型」が持っているメソッドが呼ばれる』ってやつです。

つまり、CryAnimal 関数における「現在の型」は Animal であり、Dog や Cat ではないんだわ。

あくまでも Animal 型のオブジェクトなんやね。

それでは継承した意味が…という部分を解消するための機能がこちら

「仮想関数」

これは親の関数に virtual キーワードをつけることで、継承したオブジェクトを親として扱っても、その際に呼び出される関数はそのオブジェクト自身の関数が呼び出されるってこと。

よくわかんねーと思うんで、専門的なことはともかく親の Cry 関数に virtual をつけてください。

きちんとそれぞれの関数が呼ばれるようになったともいます。

次に C++には純粋仮想関数などという仕様がある。

フツーに考えて、Animal ってのは Animal 単体では使いません。必ず継承したものが使われます。その場合、Animal に関数を実装しても無意味です。なので、これを満たす仕様として

「この関数は継承先で実装するはずだからここでは定義しないでいいよ」って仕組みがある

これのことを「純粋仮想関数」って言います。

文法は至って簡単。=0(イコールゼロ)をメソッドのおしりにつけるだけ。

virtual 戻り値 関数名(パラメータ)=0;

です。

こういうメソッドを持っているクラスを「純粋仮想クラス」と言い、純粋仮想クラスは単品でオブジェクト化することができなくなります。

どういう事かというと、メソッドの中身がないわけですから、誰かが継承して、その関数の実装をすることが求められているということです。

DirectX でいうと

IDirectX～

とか

ID3DX～

とかのクラスがそれに当たります。こいつらは単品で存在できないので new することができません。ですから create 系関数が呼ばれることになります。create 系関数の中では別のクラスがあって、そいつが関数内で new されてそれを create の戻り値にされていると考えられます。

3.1.18 typedef

これは便利機能です。覚えなくてもゲーム作れますが、覚えておくと便利なので教えます。

文法は

typedef 元の型名 新しい型名;

ってやつです。

え～？こんなもの何に使うのさぁ…めんどくせーだけじゃん。

そうですね。例えば vector とか使用してやたらと長い型名とかになった時や、パット見型名がわかりづらい時とかに使用します。

例えば vector ならば

typedef std::vector<int> IntArray_t;

とかってやれば宣言時には

IntArray_t intarray;

などと宣言できるしイテレータ使う時も

IntArray_t::iterator it=intarray.begin();

なんて使い方ができるわけです。

typedef 使わないと

std::vector<int>::iterator it=intarray.begin();

なんてくっそ長い文字列を書かなければならなくなります。それを軽減するために使用します。

他には例えば関数ポインタとか使うとき、関数ポインタ型変数の宣言は

int (*func)(int,int);

のように非常にわかりづらく、これが「変数(関数ポインタ型変数)」であることに気づきにくいです。初心者～中級者前半だとわからないです。

なのでこれを typedef 使ってもう少しわかりやすくします。

typedef int (*FunctionPointer_t)(int , int);

ここまで見ると「さっきと変わんねーじゃん」と思うでしょうが、さっきのカッコ内は「変数」であり、今回は「型」です。

つまり

FunctionPointer_t a;

FunctionPointer_t b;

とやれば a も b も同じ「関数ポインタ型の変数」として扱うことができます。

もうついでなのでメンバ関数ポインタもやっちまいます。

例えば

```
class Test{
        public:
                void Func(Test::*pFunc);
};
```

これが通常のメンバ関数ポインタの宣言。

typedef を使用すると

```
class Test{
        public:
                typedef void (Test::*MemberFunctionPointer)();
};
```

となり

MemberFunctionPointer が「型名」の役割をはたすので

MemberFunctionPointer p;

MemberFunctionPointer q;

は両方共メンバ関数ポインタとなります。

3.1.19 enum

最後にenumですけど、これは知っておいたほうが良い。いやもう知ってると思いますが、活用してる人が少ないと思いますので、ぜひ使用してください。

使い方はいたって簡単

```
enum 型名{
        要素1,
        要素2,
        :
        :
};
```

という風にぞろぞろーっと並べていくものです。

で、Enumは内部的にはint型でできてて、それをenum型と言っているに過ぎません。どうやりやすいのかというと、例えば敵のデータで

スライム=1

ドクロ=2

コウモリ=3

てな感じで番号によって表す敵が違うとします。これを1とか2とかの数値のまま運用すると、この番号を覚えておかなければいけないし、番号の変更があった場合の手間が半端ないわけです。

これがenumで扱うなら

```
enum EnemyType{//敵種別
        et_none,//種別なし(0)
        et_slime,//スライム(1)
        et_skelton,//ドクロ(2)
        et_bat//コウモリ(3)
```

};

と言った具合に表すことができます。

enum は内部的には 0 番から開始して、通しで番号が入るようになっているので、int にキャストして使用することも可能です。

こいつは「マジックナンバー」を防ぐためにあります。例えばなんかしらのフラグとか種別番号とかで189とか32って数値を書いているバカがいますが、そんな奴はプロのゲームプログラマーとしては認められません⇒ゲーム会社に受からない。

ということを覚えておいてください。

なお、enum はあくまでも、通しの種別等に使用するのに向いているものなので、ビットマスクとか画面の固定幅を表すには const int 等を使ったほうが良いでしょう。

ともかく「マジックナンバー」は極力避けていきましょう。

とりあえず「クソコード」と呼ばれるものには

- マジックナンバー(わけわかんねー数字)
- クッソ長い関数(100 行超えるな)
- ゾンビコード(消せっ…!)
- クソ深いネスト(深すぎるっ…!)
- コピペコード(関数化しろよクソがっ…!)
- バカコメント、嘘コメント(さようなら～、そんな害悪は消えてしまえ)
- 名前が不適切(もっと頭使えよ…)

などがあります。

プログラマーにとってソースコードは履歴書みたいなもんです。いや…ゲーム会社ならば履歴書よりもその人のパーソナリティを表現するものだと思われます。

とりあえず企業に提出するコードにこれらの欠点が含まれていたら、どんなに履歴書を綺麗に取り繕っても、面接でそつなく返答したとしても、クズコードを書く奴はクズなのだ扱いされますので、十分に気をつけてください。

これで C++言語的なおさらいは終了です。

4 ウィンドウと Direct3D の初期化

ここは皆さんに1から記述して欲しいところですが、ここで手こずられてもしょうもないので、とりあえずサーバーから

Dx11InitSample

ってフォルダごと落としてください。

動きますかね？

とりあえず自分のところに落として、また Git ローカルリポジトリを作って、そこに入れて作業していきましょう。

これが一番最初のプログラムです。

「いいや、俺は1から頑張るぞッ!!!(•皿•)」っていうマゾい人は、

¥¥132sv¥gakuseigamero¥rkawano¥2~3 年向け課題¥進撃の DX11L.pdf

でも見て1から書いてみてください。1から書いた方がプログラミングの力はつきます。

ただ……時間がかかっちゃいますので次年度就職年次のキミたちにはオススメできない…かな？

しょーもない課題

さて、サーバのやつを落としたらひとまずはタイトルバーの名前を

学籍番号_氏名

の状態にしてほしい。そこでそのまま「学籍番号_氏名」とか書く人は…さよなら。やる気ねーんだったらそう言ってくれ、な？ここは義務教育じゃねえんだ。幼稚園児くらいの思考しかできない奴はお呼びではない。

次に画面の色を赤とか青とかに変更してみてください。ノーヒント…と言いたいところだけど、メインループ(App 関数)の中のクリア系の関数の所がポイントです。

頑張って見つけてみてね。

4.1 解説

ちょっとここで初期化コードの解説をしておこうか。

解説をする前にヒトコト言っておくけど

「学ぶ気がなければ何を聞いても無駄」

です。

「勝手にセンセーが話して、それを聞いていればボクらはゲーム業界に行けるんでしょ？」

なんていうクッソ甘い考えならゲーム業界はアキラメロン。そりゃあ勘違いだ。

「馬を水辺に連れていくことはできるが水を飲ませることはできない」

の諺どおり、水を飲む(プログラミングを学ぶ)のは君らで、俺やないねん。そこんとこ勘違いされると時間と労力の無駄やからそのつもりでな。

あと、就職活動の時に頼りになるのはあくまでも『プログラマとしての自分の腕』ってことは忘れんといてね？

ちょっと注意したほうがいいのがラボメンバーなど『みんなでゲーム作ってる』ところね？みんなで壮大なゲーム作ったからって、それ自体は君自身の技術力の証明にはならないので、自分の面倒は自分で見なければならない。

何が言いたいかというと、就職活動の時にラボで作った作品だけ持って行っても夕メよ？ってことです。自分で『見せれるもの』を作ってないと評価されないからね？

ここでしっかり勉強して課題こなして、その上でさらに自分のゲームを作って就職に挑むんだからね？授業で音を上げている場合じゃないよ？

あと、もちろん授業外でも作品は作っておいてください。授業のやつを就職に使ってもいいけど、それじゃあ他人と差をつけることはできないよ？

差をつけるためには他人より少しでも勉強してプログラミングしてゲーム作るしかない。それ以外にはないわ…。

で、今までゲーム会社に就職できた奴らは「バイトガー!」とか「課題ガー!」とかの弱音とか言い訳した奴はいないからね。

逆にいうと……ね。わかるよね？そこで言い訳に逃げちゃうと、ガタガタガタガタと、目標は遠のいていくからね。

バイトに関しても深夜の飲み屋でバイトしながらも授業の課題をしっかりこなして、自主的にゲーム制作してゲーム会社に言った奴もいるからね？言い訳にならんよ？

言い訳するくらいだったら、どっちかをアキラメロン…マジでこっから遊ぶ暇とかほとんどないと思うくらいでいいよ。それくらいの覚悟がないとアカンすわ。

事情もあるだろうから多時間のバイトをしなければならない人もいるだろう。ただ、目標を達成したいのなら何かは我慢しないとね。時間は有限ってのは俺が言うまでもないよね。

うーん、ダラダラお説教書かせてもらいましたがぱっと見た感じ、どうやら次年度就職年次の自覚が足りてない人が結構な数いそうなので言わせてもらいました。

さて、初期化コードを見ていくわけだが、とりあえず。

InitDirect3D 関数を見てみよう。

やたらとコメントが多いけど、これは解説のためにやっているので、わざとです。

みなさんはこんなに沢山コメントを書く必要はありません。むしろうざいです。

さて、解説していきましょうか…。

とりあえず、Window 自体を作ってるのは WinMain の中にある CreateWindow なんやわぁ。

2年生は前期にやったと思うけど、3年生は初めてだろう。ともかくこいつでウィンドウを作るんや。

んで、キーワードは戻り値である **g_hWnd** なんDA。

まぁ、この名前自体は適当につけた変数名なので、名前に意味があるわけじゃなくて、コイツの中に「ウィンドウハンドル」ってのが入っていて、ウィンドウの操作や、ウィンドウ情報の取得はコイツを使って行われるわけだ。

DirectXの初期化にもこの「ウィンドウハンドル」が必要なので、非常に重要なのである。

それだけ覚えてればオッケー。

で、InitDirect3Dの中身はこうだ

4.1.1 初期化ソースコード

```
//DirectX11初期化関数
HRESULT InitDirect3D()
{
	//デバイスとスワップチェーンの作成
	DXGI_SWAP_CHAIN_DESC sd={};//構造体初期化
	//ここからスワップチェインに対する指定をひたすら書いていく
	sd.BufferCount = 1;//バックバッファの数=1
	sd.BufferDesc.Width=WINDOW_WIDTH;//バックバッファ幅
	sd.BufferDesc.Height=WINDOW_HEIGHT;//バックバッファ高
	sd.BufferDesc.Format=DXGI_FORMAT_R8G8B8A8_UNORM;//バックバッファのフォーマット
ARGB
	sd.BufferDesc.RefreshRate.Numerator=60;//分子
	sd.BufferDesc.RefreshRate.Denominator=1;//分母
	sd.BufferUsage=DXGI_USAGE_RENDER_TARGET_OUTPUT;//このサーフェスはレンダーターゲ
ットとして使用する
	sd.OutputWindow=g_hWnd;//ウィンドウハンドルを指定
	sd.SampleDesc.Count=1;//サンプリング数
	sd.Windowed=TRUE;//ウィンドウモード
```

```
	D3D_FEATURE_LEVEL pFeatureLevels = D3D_FEATURE_LEVEL_11_0;//使用したいフィーチャレベル(DirectX11モード)
	D3D_FEATURE_LEVEL* pFeatureLevel = NULL;//実際に使用可能なフィーチャレベル配列へのポインタ

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

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

	//デプスステンシルビュー(DSV)を作成
	//デプスステンシルビューってのは所謂「Zバッファ」を有効にするために必要
	//Zバッファってのは描画順によらず前後のオブジェクトを正確に描画するのに必要
	D3D11_TEXTURE2D_DESC descDepth;
	descDepth.Width = WINDOW_WIDTH;//画面幅
	descDepth.Height = WINDOW_HEIGHT;//画面高
	descDepth.MipLevels = 1;//ミップマップとかいらないので１でいい
	descDepth.ArraySize = 1;//テクスチャの数
	descDepth.Format = DXGI_FORMAT_D32_FLOAT;//深度バッファのビット数と型の指定(32ビットFloat型)
```

```
	descDepth.SampleDesc.Count = 1;//1でいいよ
	descDepth.SampleDesc.Quality = 0;//アンチェリしないのでいらない。
	descDepth.Usage = D3D11_USAGE_DEFAULT;//ここはこれ固定で
	descDepth.BindFlags = D3D11_BIND_DEPTH_STENCIL;//このテクスチャは「深度値」保存用に使うよと明記
	descDepth.CPUAccessFlags = 0;//とりあえず0でいい
	descDepth.MiscFlags = 0;//とりあえず0でいい
	g_pDevice->CreateTexture2D( &descDepth, NULL, &g_pDS);//深度値用テクスチャの生成

	g_pDevice->CreateDepthStencilView(g_pDS, NULL, &g_pDSV);//作ったテクスチャ元にデプスステンシルビューを生成

	//ビューポートの設定(これがないとモノを表示した時に表示されません)
	D3D11_VIEWPORT vp;
	vp.Width = WINDOW_WIDTH;
	vp.Height = WINDOW_HEIGHT;
	vp.MinDepth = 0.0f;
	vp.MaxDepth = 1.0f;
	vp.TopLeftX = 0;
	vp.TopLeftY = 0;
	g_pDeviceContext->RSSetViewports(1,&vp);

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

	return S_OK;
}
```

なるほどな、なるほどなるほどなるほどな！

DirectX11のハードルが高いのは、この初期化をするためだけにでも、コンピュータグラフィクスの知識やハードウェアの知識が多少必要になるってあたりなんだ。

逆に言うとDirectX11のハードルを超えられる奴はそうそういない。ガッツリわかってないとプログラムを組めないからだ。ということは…この授業を乗り越えられれば他校の奴らに差をつけられる…そういうことだ。

頑張ろう。な!

乗り越える気がないカスはもう知らんので、せいぜい単位を落とさない程度には頑張ってください。

4.1.2 ひとまず必要な用語の解説

初期化に必要な基本的な用語と知識

デバイス(ID3D11Device)

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

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 先生と深あい関わりを持っている。つまるところ GPU 先生にコマンド投げる役割を持っているのである。

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。

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。

意味がわかりづらいなら、ビューっていうのはコンピュータの中の「絵をバッファに描く職人さん」くらいに考えておいたら良いよ。

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。通常の出力先はウィンドウの中なわけだから、ウィンドウの中をレンダーターゲットとして設定しておけば良い。

RGB の絵を描く職人さんやね。ゲームを作るならこの人が最低一人は必要になります。

ダブルバッファリング

こいつを説明するには画面のチラツキについて知らなければならないのだが、そもそもウィンドウに絵を描画するってのは、ものっすごい高速で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(見えてない画面)に描画し、書き終わったら(適切なタイミングで)表と裏を入れ替えるという技法。

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

4.1.3 実際にソースコードリーディング

まず、初期化でやらなければならないことは、DirectX9 と同様に「デバイスの生成」です。
CreateDevice は覚えてるよな？

DirectX11 の場合はそれよりも少しだけ面倒で、デバイスとデバイスコンテキストとスワップチェーンを生成しなければならないようです。

まぁ関数ひとつで生成してくれるのでまだマシです。

D3DXCreateDeviceAndSwapchain

この関数が、デバイスとデバイスコンテキストとスワップチェーンを作ってくれます。

次にやらなければならないのは、レンダリング結果を書き込むための「レンダーターゲット」の生成です。

レンダリング結果を書き込むためのレンダーターゲットってのはなんていうのかな…？画家が「レンダー」で絵を描くことを「レンダリング」って言って、そうなると「レンダーターゲット」っていうのはキャンバスのことなんやな。

DirectX11 では D3DXCreateDeviceAndSwapchain した時点でメインのレンダーターゲットは作られていると思っていい。

なんだけど、このレンダーターゲットを画面に見える状態にするための「ビュー」を作る必要があるので、CreateRendertargetView を呼び出している。

で、ソースコードの方ではデプスステンシルビューまで作っちゃってるけど、とりあえず今は

「深度バッファを使用するための準備」

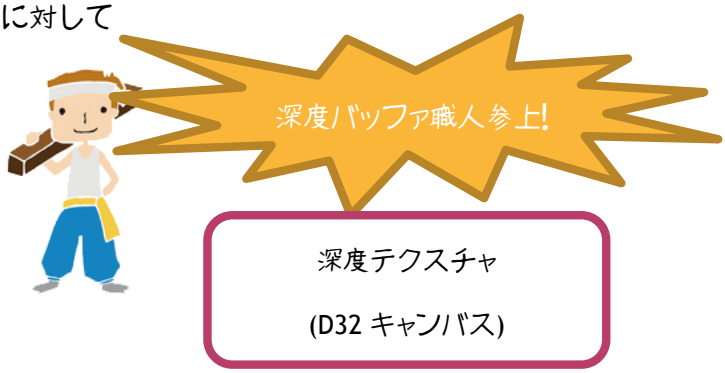
と理解しておいていい。

え？深度バッファを知らない？

そいつは参ったな…。もうちょっと後で説明するんで、とりあえず今は「物体の前後を効率的に判断して正しいレンダリングを行うモノ」と思っておいてください。

で、深度バッファってキャンバスに深度値を書き込む職人(デプスステンシルビュー)がいる。って理解しといてください。

に対して



がいて、それぞれが書き込むキャンバス(絵を描く紙)も違うし、役割も違う。職人さんも違う。そんなイメージだと思ってくだしあ。

プログラム文中のdescDepthってのは職人に指定するキャンバスの大きさとか色鉛筆の色数とかの注文です。

深度バッファは別に色鉛筆の必要はないので、D32_FLOATという指定になっているし、そのキャンバスを何に使用するかって説明はBIND_DEPTH_STENCILつまり深度とステンシルに使用するという指定になっています。職人さんが使用するキャンバスは基本的に「何に使用するか」を指定する必要があります。

キャンバスを生成しているのがCreateTexture2Dで、深度バッファ職人さんを雇っているのがCreateDepthStencilViewです。

細かい専門的なことはともかく、次行ってみよう。

次はビューポートの作成です。

ビューポートはなんかっつーと、出来上がった絵を何処に納品(展示？)するかってのを指定する奴です。

どういう風にどこに展示するのかってのを決めます。具体的に言うとウィンドウにどんなふうに描画するのかってことですね。

とりあえず

幅=ウィンドウ幅

高さ=ウィンドウ高さ

左上=(0,0)

MinDepthとMaxDepthとあるけど、今は0.0と1.0にしといてね。

意味はあるんだけどね…クリッピングボリュームのニアとファーの意味なんだけど、たぶんCGの勉強してないとFFの「ファルシのルシがコクーンでパージ」ってのと同じくらい意味不明だと思います。そういう理由で、この辺も後々説明します(現状だと頭に残らない)。

とりあえず「こういうもんだ」と思っておいてください。

4.1.4　深度バッファ(Z バッファ)について

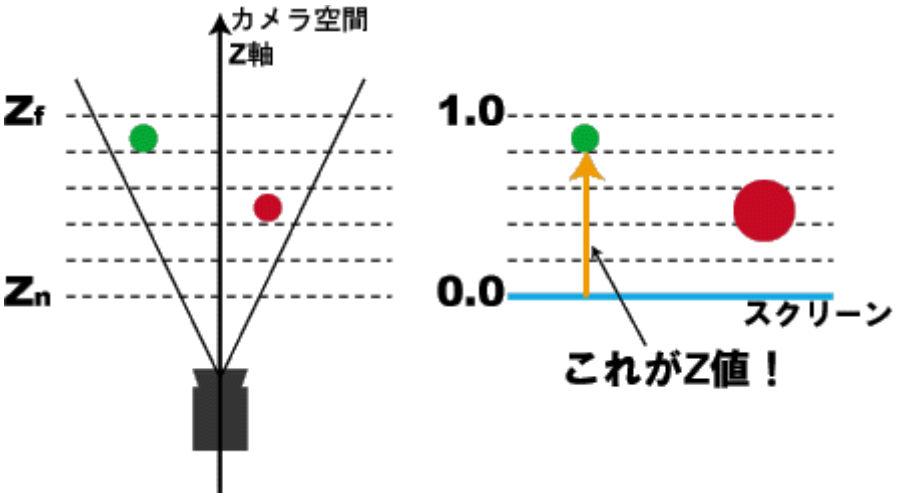
深度バッファってのはその昔 Z バッファと呼ばれていたもので、奥行き方向にいくつも重なって見えるオブジェクトを正確に描画するために考えられた手法なのだ。

DirectX というか CG ってのも割りと単純に考えられていて、油絵とか書いたことがある人はわかると思うけど、絵の具を上から重ねてしまったら、下がどんな色であろうとも、塗ったその色になってしまうよね？

それと同じことが起きてしまうんだ。

つまり手前の物体から描画をしていると、奥の物体の描画で塗りつぶされて、手前の物体が見えなくなるのだ。でも、手前の物体は絶対見えるはずだよね？その不自然さを解消するための技法なのだ。

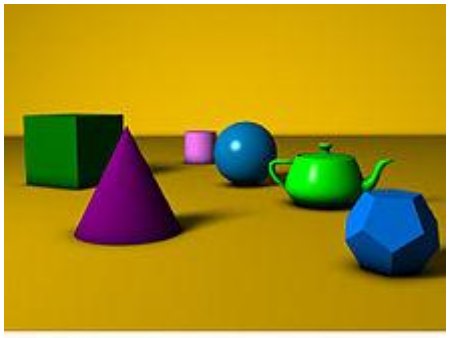
Z バッファがないと…向こうにあるものが手前に描画されることがあって、非常にキモチワルイ。ということで、カメラからの距離を最大1(クリッピングボリュームの ZFar 側)として、深度テクスチャ(レンダーターゲット)に深度値を書き込んでいくのだ。



(げーむつくろーの HP から図をパクってきました)

そうすることによってそれぞれの画素値として「深度値」が書き込まれていく。

例えば、こういう配置の物体の深度値をグレースケールで表すとこうだ。



奥に行けば行くほど白く描画される。

で、仕組みとしては、「今から書こうとするピクセルの深度値」と「元から書かれていたピクセルの深度値(深度バッファの値)」を比較して、今のが小さければ(より近ければ)色を塗り(かつ深度バッファの深度値を更新する)、そうでないなら何もしない。

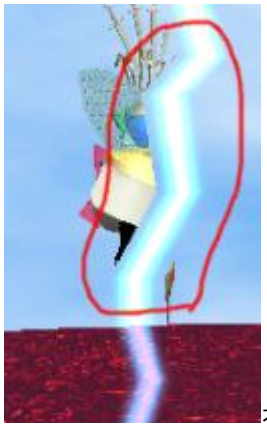
とすることによって、

手前の物体より奥にあるピクセルは描画されず、奥の物体より手前にあるピクセルは描画されるわけだ。この仕組により重なっている物体を正確に描画することができるわけ。

ただしこの技法は残念ながら万能ではない。「アルファブレンディング」との相性が頗る悪いのだ。

どういう事かというと、半透明物体ってのは向こうが透けて見えなければいけないよね？「半透明」なんだからさ。

ところがこの深度バッファの仕組みだと、「先に書かれている深度値がより手前ならば描画しない」というルールが裏目に出てしまい、透けるべき向こうのオブジェクトが透けなくなってしまうのだ!!!

 本来は雷の向こうのオブジェクトが見えるはずなのに…

見えてないよね？

後ろの絵が一部消えてるよね？

こういう事が発生してしまうわけ。これではイカンね。

んで、これの対処法は色いろあるんだけど、まぁまずは深度バッファを切る。これで全ての描画が行われるはずなのでうまくいく……わけねーだろ!!!

深度バッファを切るってことは「隠面消去」が行われないってことだから、不適切なオブジェクトまで描画されてしまうのだ。

 スカートの向こう側が描画されてしまっているの図

まぁそもそも不適切なオブジェクトを表示したくないから深度バッファなんて言う技法によってメモリを消費しているのに、それを切っちゃあなんにもならねえよな!!

ということで、深度バッファを切らずにやる方法の一つとして、

最初に透明オブジェクトと不透明オブジェクトを分割する。

予め透明なマテリアルにタグか何かをつけておき、区別できるようにしておく。

透明部分、不透明部分のそれぞれの描画をそれぞれ命令バッファに溜めておく。

で、その上で

① 不透明オブジェクトを全て描画
② 透明オブジェクトを描画

という風に不透明オブジェクトを全て描画したことを確認したうえで、透明オブジェクトを描画すればいい。

更に言うと②段階目の不透明オブジェクトを描画する際に、

- 深度バッファ(深度テスト)は有効にしておく
- 深度バッファ書き込みは行わない

という、ちょっと特殊な状態にしておくことによって、

不透明部分の向う側にある透明物体は表示されない

かつ

透明部分同士が重なっている部分はきちんとアルファブレンディングする

という期待していた動作にすることができます。

実際に実装するのは結構たいへん(そうでもないけど)だし、まだまだ知識が必要です。

理屈がわかってたら別に難しくはないんだけど、もうちょっと基礎知識は必要なのでとりあえずそれは実際にメッシュを表示する時に説明していきます。

5 三角形を表示しよう

三角形を表示する…たかだかそれだけのことが DirectX11 では……そうね。ものごっつ面倒なのよねー。

なにぶんシェーダを書かなければならないからねー。

だから二年生前期は時代遅れと知っていつつも DirectX9 にしてたんだよね。まぁ、Dx9 は未だにエロゲ業界とかではメジャーなのでそっち業界狙いの人は DirectX9 でも良いと思う(ただし illusion を除く…あそこは頭おかしいレベルの技術を使ってやがる…一度先入観抜きで、自社 HP の技術研究ブログを見てみると良い…そこらのコンシューマよりも研究してるぞ! 方向性はアレなんだけど…)

まぁ、ここで一段階ハードル上がるからね。覚悟してね。

DirectX9 の時の手順は

1. 三角系の頂点定義する
2. SetFVF でどういうデータかを DirectX に教えてやる
3. 表示の仕方を定義する
4. 三角形を表示する

くらいだったんですが、DirectX11 だと

1. 事前にシェーダを書いておく
2. 頂点シェーダをロードする(まだ使える状態じゃない)
3. ピクセルシェーダをロードする(まだ使える状態じゃない)
4. 頂点シェーダとピクセルシェーダを使用できる状態にする
5. 三角系の頂点を定義する
6. 三角形データをバーテックスバッファに変換し、Direct3D 側に渡す
7. 頂点データレイアウト(どういうデータかを DirecX に教えてやる)を作成
8. 頂点シェーダ、ピクセルシェーダをセットする
9. 三角形を表示する

倍くらい手間が増えてます。がんばろう。

5.1 ひとまずシェーダを書いてみよう

シェーダ書き始めると、まぁプロっぽい感じがするな。

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

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

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

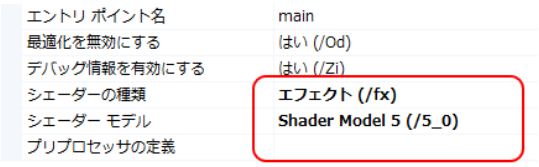
そうするとプロジェクトに追加されていますね。デフォルトの頂点シェーダが既に書かれています。今回は main 関数を使用しないので

関数名を VS とでもしておきましょう(VertexShader の略ね)

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

なんか main がないとダメとか言って怒られるので、黙らせます。

さっき作ったシェーダ(hlsl)を右クリック→プロパティ→全般



その部分をこう書き換えます。そうしないと main 関数ないって怒りますよ。

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

先ほど修正した頂点シェーダの下にでも入れておいてください。

もちろんコイツも Git の仲間に入れてあげてください。

ピクセルシェーダを書いたうえで再び実行。うまくいきましたか？とりあえず今は何も表示されません。

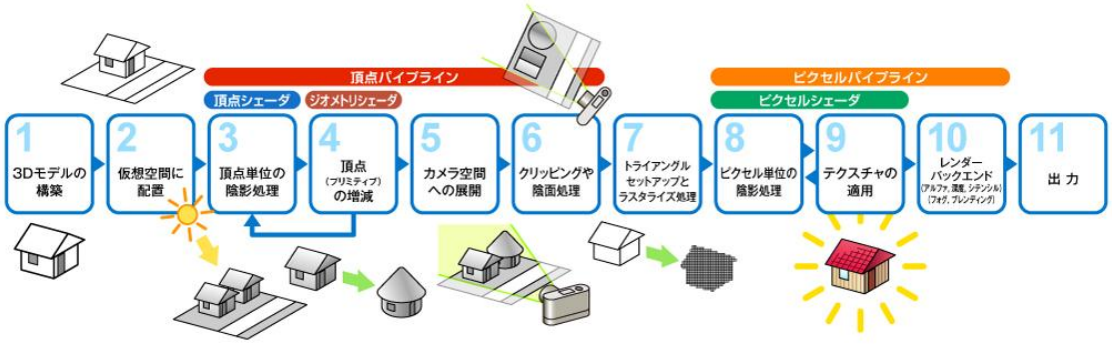
まずはシェーダのロードです。

シェーダには大きく分けて、頂点シェーダとピクセルシェーダってのがあります。ホントいうと DirectX11 には

- 頂点シェーダ
- ピクセルシェーダ
- ジオメトリシェーダ
- ハルシェーダ
- ドメインシェーダ
- コンピュートシェーダ

というシェーダがあります。でも基本になるのは頂点シェーダとピクセルシェーダです。その名の通り、頂点シェーダは頂点ごとに呼び出されるシェーダで、ピクセルシェーダはピクセルごとに呼び出されるシェーダです。

Direct3D などの仕組みにより、3DCG を表示するまでの過程をレンダリングパイプラインとか言います。



(この図が一番わかりやすい)

図のように頂点シェーダは頂点単位の陰影処理や、頂点自身の座標変換に使用されます。

まぁ、なんというか、みんなが数学の授業でやってきた「行列の乗算による座標変換」とか、「内積による陰影処理」をこの頂点シェーダで行います。

まぁ…その…なんだ……やっぱり数学が必要なんだ。

それわかって手この教室にいるんだよな？コンシューマゲーム業界狙ってるんだよね？そういう事だから仕方ない。

自分で決めたことなんだから今更こんなこと言うなよな？

んで、ピクセルシェーダのほうはというと、既にラスタライズ(頂点情報をピクセル情報に変換)された後のデータについて処理を行います。

簡単にいうと「表示する色を決定する」ってわけ。序盤は数学いらないです。だから、どっちかというとしばらくの間はピクセルシェーダのほうが簡単です。

さて、前置きはこれくらいにしてやっていきましょう。

5.2 頂点シェーダのロード

InitShader みたいな関数(とりあえず戻り値も引数もなしでいい)を作ってください。その中でロード処理を書いていきます。InitShader()のコール場所は InitDirect3D の直後くらいでいいでしょう。

ひとまずシェーダのロード&コンパイルです。

シェーダをロードするには

D3DX11CompileFromFile 関数を使用します

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

こいつは頂点シェーダおよびピクセルシェーダ両方で使えるやつなので、それぞれこの関数をコールします。

```
ID3DBlob* compiledShader=NULL;
ID3DBlob* error=NULL;

//頂点シェーダ読み込み
HRESULT result = D3DX11CompileFromFile(L"",//シェーダファイル名書いてね
        NULL,//10使わないし・・・NULLでいいや
        NULL,//10使わないし・・・NULLでいいや
        "",//最初にコールしたいシェーダ関数名
        "vs_5_0",//シェーダバージョン(なんでか文字列で指定なのよね)
        0,//コンパイルオプションは特にないので0(ホントは最適化のために色々と設定する)
        0,//実行時オプションも特にないので0
        NULL,//とりあえずここではスレッド使わないので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);//スレッドを使用した際の戻り値
```

あ、もちろんシェーダファイル名と、シェーダ関数名は自分で考えて書いてな？

全てを理解するのは無理だと思うけど、

必ず一度は公式リファレンスを読んでから理解できるところは理解して書いてください。

「写さなければプログラムできない」ならそれはプログラマではないのでね。今は分からなくて写してもいいけど、プロになったら自分で考えて「なんでこう書くのか」を確認しなければならない。

「どっかのブログにこう書いてあったから」とか言って、ほぼほぼどっかのブログで書いてあったコードまんま書いて「キチンと書いたのに動かないです」

なんて言う奴は単なるコピペ野郎で、プログラマではないです。

あと、基本的にどっかのブログにそう書いてあったんなら、妥当性を必ず検証して、API やライブラリの話であれば必ず**公式のドキュメントを読んで必ず裏をとってください。**

DirectX なら信用できるのは MSDN ドキュメントだけだと思っておいていい。あと、たまに MSDN でも翻訳のせいでおかしい場合があるので「怪しいな、納得出来ないな」って思ったら、手間だけど英語サイトまで見てください。

あと「どうして自分はこういうコードを書いたのか」を説明できないとプロプログラマとはいえないです。

とりあえず書けたら実行してください。エラーは起きてませんか？

5.2.1　エラー対応

コンパイルエラーの場合

まずエラー行を見ておかしな所がないか探す→分からないならメッセージを読む→分からなければエラーコードをグーグル先生で調べる→分からないなら隣の人に聞く→分からないなら先生に聞く

の順番で考えてください。先生は最後にしてください。

ここの教室には40人近くいて、センセーは1人やし…あんまり頻繁に質問されても分身でけへんねん…。

でも**「分からないまま終わる」は最低最悪の選択肢**なので、最後には聞いてください。

まぁ…今の貴重な時間をドブに捨てたいならどうぞお好きに。

あと、実行時のエラーなんですけど、「なんか分からへん」じゃなくて必ず関数のリザルトくらいは確認しよう、な！

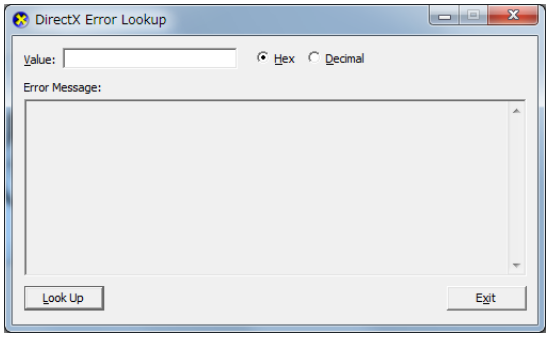
ちなみに DX 系の関数の戻り値の HRESULT result で戻ってきた値は、そのままでは単なる数値だ。だから調べる方法を教える。例えば D3DX11CompileFromFile から

result = 0x887c0002

なんて値が返ってきたとする。

このままではどうにも分からない。使いものにならない。

そこで DirectX Error Lookup というのを立ち上げてくれ。おそらく Windows キー＋Q ボタンで検査具画面になるので、Error と撃ちこんだら出てくるだろう。



こんなの。

Value に 0x887c0002 を入力して Lookup ボタンを押す。

すると

HRESULT: 0x887c0002 (2289827842)

Name: D3D11_ERROR_FILE_NOT_FOUND

Description: **File not found**

Severity code: Failed

Facility Code: FACILITY_D3D11_OR_AE (2172)

Error Code: 0x0002 (2)

などというメッセージが表示される。とりあえず英語が苦手な人でも FileNotFound くらいは分かるだろう？

ファイルが見つからねえってことよ。

どうしても英語がイヤ…虫酸が走るわって人は Google 先生にさっきのエラーコードを投げてください。色々と日本語解説が出てくるかもしれませんが、関係ないものも引っかかるので、そこは注意してください(英語…勉強しようっ)。

とりあえずきちんと S_OK が帰ってくるところまで確認してください。

ちなみに最後の引数の ID3DBlob ですが、これはでかいバイナリオブジェクトを表すと思っておいてください。

(BLOB ってのは Bynary Large OBject の略です…ですから GetBufferSize()や GetBufferPointer()なんていう関数を持っています。

…まぁ、つまるところ ID3DBlob というバイナリデータを扱うものがあって、コンパイル済みシェーダや、エラー情報はそれで受け取り、色々なことに利用します。

はい、そこまでで頂点シェーダプログラムのロードは完了しました。

で、これはあくまでもプログラム(シェーダ文字列をコンパイルしたバイナリ状の何か)でしかないので、この「バイナリ状の何か」をきちんと「頂点シェーダくん」にしてあげる必要がアリます。

だから次にやることは CreateVertexShader で「頂点シェーダくん」を作ることです。これは大したことしないです。

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

それなりに理解しながら進んでいる人や、勘のいい人はリファレンス見ればわかると思います。あなたがたどうですか？

このリファレンスの日本語は比較的わかりやすいです。ヒントをいうと、第三引数は nullptr で構いません。それ以外…ドキュメントを読んで、自分で考えて呼び出してください。

ちなみにヒントをひとつだけ

IDirect3D11Device::

とかいうのがついている場合は、これどういう意味でしたっけ？そう、CreateVertexShader は IDirect3D11Device の持ち物なんでしたよね？

いま、型が IDirect3D11Device なのは誰ですか？探して、そしてじっくり考えてみましょう。

最後の引数で頂点シェーダを受け取るわけだけど、それの変数名は vs とか vshader とか vertexShader とか、「頂点シェーダ」ってわかる名前にしておきましょう。僕は面倒なので、vs にしてます。

とりあえず戻り値である HRESULT が S_OK になるところまで自分で考えてそして試行錯誤してやってみましょう。

できたかなー？

よしできた!!!前提で進むよ

で、順番的に次はピクセルシェーダのロードまでやっておきたいんですけど、その前にやることがあります。

コンパイル済み頂点シェーダがあるじゃん？

コイツを頂点レイアウト定義のために使用しなければならないんですよ!!なのでピクセルシェーダ前にレイアウト定義します。

5.3 頂点レイアウト定義

そもそも「頂点レイアウト」ってなんやねん？

これはね DX9 で言うところの FVF の親戚みたいなもんです。FVF 覚えてる？「**頂点1つあたりにどういう情報が含まれているのか**」っていうのを決める奴だったよね？

ほぼアレと同じです。DirectX11 の場合はもっと具体的に指定する。めんどくさいけど、言い換えるともっと柔軟になったとも言えるわけ。

ともかく頂点レイアウトを定義していこう。

最も簡単なのは「座標のみ」だ。

X 座標 Y 座標 Z 座標…つまり float3 つ分だね。それを「座標」と定義する。

頂点レイアウトを定義するためには

device->CreateInputLayout(レイアウト情報,

情報数,

コンパイル済み頂点シェーダポインタ,

コンパイル済み頂点シェーダサイズ,

頂点レイアウトオブジェクト取得用);

https://msdn.microsoft.com/ja-jp/library/ee419795(v=vs.85).aspx

こんな感じの関数で定義します。

リファレンスは一回見てみましょう。で、第一引数の説明が超わかりづらいですよね。

D3D11_INPUT_ELEMENT_DESC

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

「入力アセンブラー ステージの単一の要素についての記述です。」

とりあえず何言ってるか分かりませんが、DirectX 系で○○DESC と来たら、なんか生成するときの細かい指定をしていくために使うものです。

初期化の際とかのスワップチェインとか、テクスチャ生成の時にも出てきてましたね。

色々と書いていく必要がありますが、

ヘルプ見ると

LPCSTR SemanticName;//セマンティクス名(POSITION)

UINT SemanticIndex;//セマンティクスインデックス(0 で)

DXGI_FORMAT Format;//フォーマット(DXGI_FORMAT_R32G32B32_FLOAT)

UINT InputSlot;//入力スロット(とりあえず今は 0 でいい)

UINT AlignedByteOffset;//オフセット(どれくらいのバイトからがそのデータか→0)

D3D11_INPUT_CLASSIFICATION InputSlotClass;//D3D11_INPUT_PER_VERTEX_DATA にしとこう

UINT InstanceDataStepRate;//0 でいい

こんな感じになっています。

ところで「セマンティクス」ってなんでしょう？ゲームプログラミングに限らず色んな所で出てくる用語ですがこれは「データの意味」みたいな感じです。

つまり(0,0,1)というデータがあったとして、それが座標情報なのか法線情報なのかで扱いは全然違いますよね？

そういう**意味**を付加するためのものです。

セマンティクスはここに書いてます

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

とりあえず"POSITION"でいいと思います。

ということで

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

こんな感じで定義してください。

構造体はこういった感じで一気に初期化できるのは知っているよね？たぶん C#でも同じだと思いますし…。

さらに最終的にコイツは構造体の配列となっていくので

```
//頂点インプットレイアウトを定義
D3D11_INPUT_ELEMENT_DESC layout[] =
{
        {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

と定義しましょう。

現状だと要素数1ですけどね。

現状 layout[0]しかないわけです。

それでいいです。

それを第一引数に放り込みます。

今回は POSITION 一個だけなので、第二引数は1

あとはさっきコンパイルしたコンパイル済みシェーダを放り込む。

そして最後の引数で、生成されたレイアウト情報を取得します。

ちなみに最後の引数の型は ID3D11InputLayout のポインタのポインタです…ということは…分かるな？

ポインタのポインタは解説済みだからね。

…二年生で分からない人は前期の授業を思い出そう。Create なんとかの最後のポインタのポインタですよ。

これを実行したうえで S_OK が返るまでやってください。

あとはセット(Direct3D にレイアウトを教えてやる)です。

IASetInputLayout 関数を使用します。

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

持ち主はコンテキストくんなので

context->IASetInputLayout(さっき作ったレイアウトオブジェクト);

ここまで来たらもはや頂点シェーダロードに使用した CompiledShader は必要ないので、捨てます。

DirectX11 では破棄するときは Release を使います。

compiledShader->Release();

compiledShader=nullptr;

とでも書いてあげます。この変数自体は、あとで再利用します。

5.4 ピクセルシェーダのロードと生成

さぁいよいよピクセルシェーダのロードと記述です。

ファイルは頂点シェーダと同じなのでちょっと無駄に感じますが、同じように

D3DX11CompileFromFile を使いましょう。

違いは、

- シェーダ関数名(“PS”)
- シェーダモデル名(“ps_5_0”)

くらいです。で、また頂点シェーダの時と同様に CompiledShader を作りましょう。

できましたかー？

もちろん次は CreatePixelShader ですね

https://msdn.microsoft.com/ja-jp/library/ee419796(v=vs.85).aspx

あとはわかるな？

最後の引数の変数名は僕は ps とでもしておきます。

さて、ここまで HRESULT が S_OK になるまでがんばりましょう。

5.5 頂点の定義

来ました。

やっとこさ頂点定義です。

5.5.1 頂点配列を用意する

三角形を表示したいので3頂点ですね。1頂点あたり X 座標(float)Y 座標(float)Z 座標(float)ですね。

ということは XYZ 座標を持つものが3つ…つまり float が 9 個あるので、どういうやり方でもいいので 3 頂点を表す 9 個分の float を用意してください。

やり方はいろいろありますよね？

① float9 個ぶんの配列を作る

```
float vertices[]={1.0,2.0,0.0,-1.0,5.0,0.0,2,2,0};
```

みたいな感じで(数値はテキトーです)

② xyz の構造体を定義して、それを三つ分用意する

```
struct Position{
                    float x,y,z;
};
Position vertices[] ={{1.0,2.0,0.0},{-1.0,5.0,0.0},{2,2,0}};
```

てな感じ。別にどっちでもいいです。

頂点を作る際の注意点

頂点は時計回りになるように定義してください。

OpenGLでもDirectXでもデフォルトで「背面カリング」という機能が搭載されていて「裏面を描画しない」ことで、計算量を減らし、更には陰面消去にも役立っているのです。

で、どっちが「裏」かってのを判別するために時計回りか反時計回りかってのが非常に重要なのです。デフォルトでは反時計回りが「裏」なので、表示したいなら時計回りに定義してください。

5.5.2 頂点バッファを作る

さて、めんどくさいんですけど、この頂点配列の状態では Direct3D に渡すことができません。渡すには「頂点バッファ」というものに変換してあげる必要があります。

CreateBuffer 関数を使用します。

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

で、最後の引数でバッファを受け取るのはわかるんだけど、第一、第二引数がよくわからんね。

まぁ、第二引数は初期化データってことなので、さっきの頂点データを放り込めばいいことはなんとなくわかるか。

問題は第一引数

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

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

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

Usage に関しては、D3D11_USAGE_DEFAULT

bindFlags は、頂点バッファなので D3D11_BIND_VERTEX_BUFFER

CPU アクセス必要ないので、CPUAccessFlags=0 で

MisgFlags も 0 でいいです。

ByteWidth はバッファの全サイズを…つまり float*3*3 な

そこまで書けたら次は SUBRESOURCEDATA ですが簡単です。

pSysMem にさっきの頂点情報のアドレスを入れれば終わりです。

D3D11_SUBRESOURCE_DATA initData={};

initData.pSysMem=vertices;

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

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

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

さぁ送りましょう。どうやって送るのかというと

IASetVertexBuffers

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

を使用します。

DeviceContext の持ち物なので…わかるな？

ところで頭に付いている”IA”ってなんだろうね？”OM”とかもそうなんだけど…一応

http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/DirectX%2010%20for%20techies.pdf

こんな資料があって、

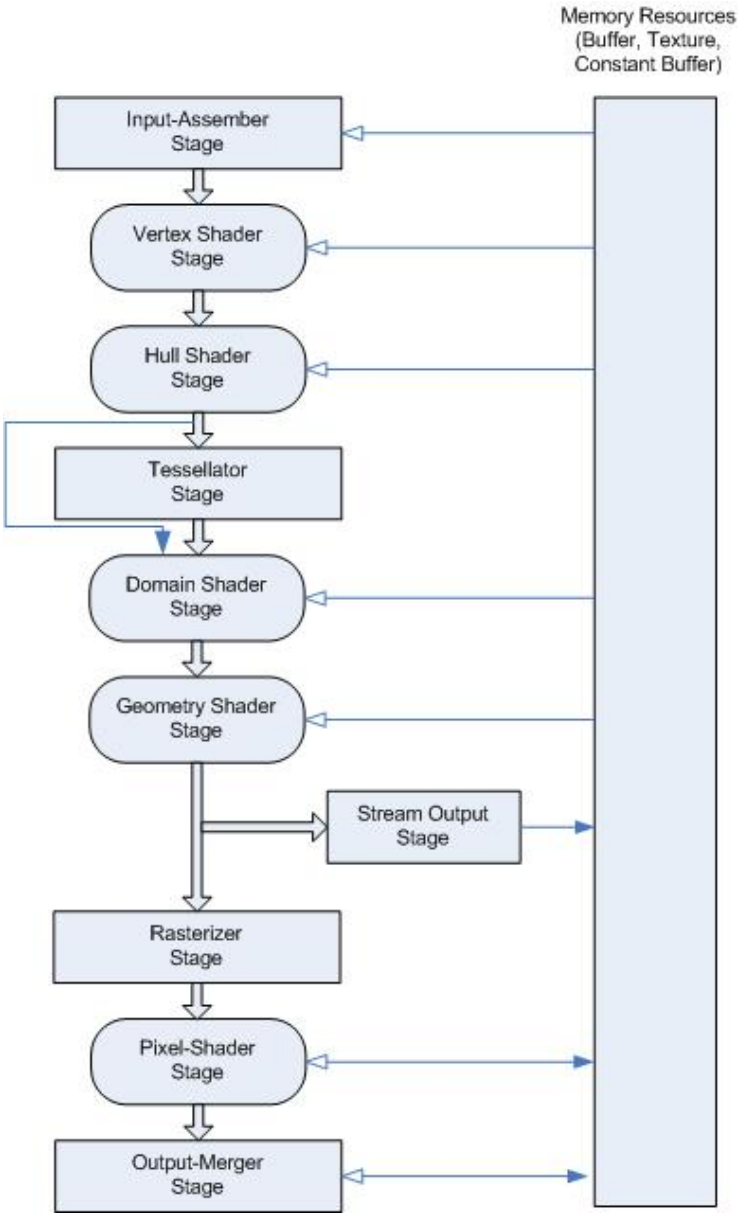
IA=InputAssembler

OM=OutputMerger

と書いています。

日本語の説明のドキュメントがほとんど見つからない…大抵の場合「こう書くんだ」で終わってる…ある意味この IA とか OM とか RS にこだわっても仕方ないので、それはそれでいいんだけど、やっぱ気になるじゃん？

https://msdn.microsoft.com/ja-jp/library/windows/desktop/ff476882(v=vs.85).aspx



なんかこうゞうことらしいけど、なるほどわからん。わからんが、InputAssembler は頂点シェーダの前にあって、OutputMerger は最後に来てるってことだけはわかる。

一応 IA を日本語に直すと

https://msdn.microsoft.com/ja-jp/library/bb205116(v=vs.85).aspx

らしい。

読んでも「ファルシのルシがコクーンでパージ」状態なので、とにかく float のデータの塊を点として扱うか、辺として扱うか三角形とかとして扱うか…みたいな意味だと思っておいてね。

アセンブルってのはまとめるとかそういう意味なのだ。

ちなみに OM は

https://msdn.microsoft.com/ja-jp/library/bb205120(v=vs.85).aspx

これもまた初心者には「ファルシのルシが(ry」状態。

とにかく最終出力の時に、色の塗りつぶしと深度の結合…つまり、ペンキ職人と測量職人が協力しあう場所とでも思っておいてください。

まぁ、別にこの辺はわからなくてもいいんだけど、気になる人がいると思いますので、一応書いておきました。

ともかく IASetVertexBuffers の定義はこんな感じ

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

さて…とりあえず第1～3引数は大丈夫だよね？

問題は残りのやつだけど、

ストライド値ってのは頂点1つあたりの消費バイト数。

だから、今回は1頂点あたり3つの float なので、sizeof(float)*3 もしくは sizeof(Position)とかで良いと思います。

で、なんでそんな値がわざわざポインタになっているのかというと…引数の説明のところの Stride って単語には s が付いていると思う。

変数の最後にｓがついたら複数形→配列を暗に表していると思っていい。

だから本来はこいつは配列であるべきだが、どうせ1種類しかストライドは存在しないので、フツーに変数作って、そのアドレスを渡してやればいい。

つまり

```
UINT stride=sizeof(Position);
```

```
&stride
```

みたいな指定で良い。とりあえずはオフセットしなくていいので、ここも０を入れる。こいつも配列を想定しているが一つでいいので

```
UINT offset=0;
```

```
&offset
```

としておく

つまり

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

みたいに書く。

こいつはHRESULTは返さないので、失敗したら…まぁクラッシュするか、何も起こらない…そんな感じです。

さて、頂点バッファもセットした。

あとは仕上げ…シェーダをセットして、ドローだ!!!

と行きたいところだけど、後一段階やらなければならないことがある。

プリミティブトポロジの設定である。

なに、大したことじゃなくて、頂点データをどのように組み合わせるのかって指定が必要になるわけ。どういうことかというと

https://msdn.microsoft.com/ja-jp/library/bb205124(v=vs.85).aspx

こういうことだ。

つまり前に説明した「点として扱うか、線として扱うか、三角形リストとして扱うか、連続三角形として扱うか、扇型として扱うか」を指定するための関数を指定子てあげないといけない。

5.5.3 プリミティブトポロジの設定

プリミティブトポロジを設定するには

IASetPrimitiveTopology 関数を使用します。

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

引数はひとつだけなので簡単です。

ここから一つ選ぶだけ

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

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

これでやっと下準備は終了です。

5.6 いよいよ描画命令

いよいよ描画命令ですよ。

描画命令は、まあそれなりに簡単なんだけど記述する部分はもちろん、メインループの中でお願いします。つまり App 関数やね。

やることは

- 頂点シェーダのセット
- ピクセルシェーダのセット
- ドロー命令

これだけです。

さぁ、やっていこう

名前がちょっとわかりづらくて頂点シェーダのセットが SetVertexShader かと思ったら、

VSSetShader と PSSetShader って名前になっています。

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

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

とりあえず両方目を通しておきましょう。

中を見ると分かりますが、第一引数はシェーダを入れて、

第二引数は NULL でいいです。

第三引数は、第二引数が NULL なので 0 でいいです。

そうなると

context->VSSetShader(vs,nullptr,0);

となります。

ピクセルシェーダに関しては自分で考えましょう。

次にドローですが簡単です。

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

第一引数が頂点の数、第二引数がオフセット。

今回オフセットは必要ないので 0 で…

そうすると

context->Draw(3,0);

みたいになると思います。

はい、今週の課題です。

この三角形表示を四角形(クアッドポリゴン)表示にしてみてください。

よろしく!!!

色々と調べなければいけないと思うけど…いや、そうでもないけど。

ひとまず、クアッドポリゴン表示の時のトポロジは TRIANGLE_LIST ではなく

TRIANGLE_STRIP のほうがラクに書けますよ。

どうですかー？書けましたかー？

まずは、トポロジを TRINANGLE_STRIP にします。

D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP

にした後は

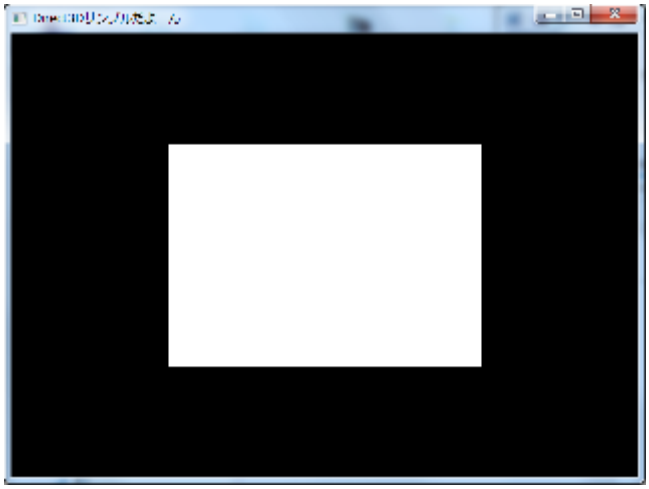
ひとまず頂点を4つにしましょう。この時の注意点は、**Z の字**になるように、もしくは **N の字**になるように頂点を設定してください(そうしないとカリングにより見えなくなります)。

つまり

```
Position vertices[] ={
                        {-0.5,-0.5,0.0},//左下
                        {-0.5, 0.5,0.0},//左上
                        {0.5,-0.5,0.0},//右下
                        {0.5,0.5,0.0}//右上
};
```

ちなみに、頂点バッファサイズも×4にしなければならないし、Draw の第一引数も4にシなければならないことは分かるよね？

それができたらこうなります。



あるぇー(•3•)？なんで縦横ともに 0.5 ずつなのにこんなに横長なの？

それは現在の画面のアスペクト比が 4:3 になっているからだよ？

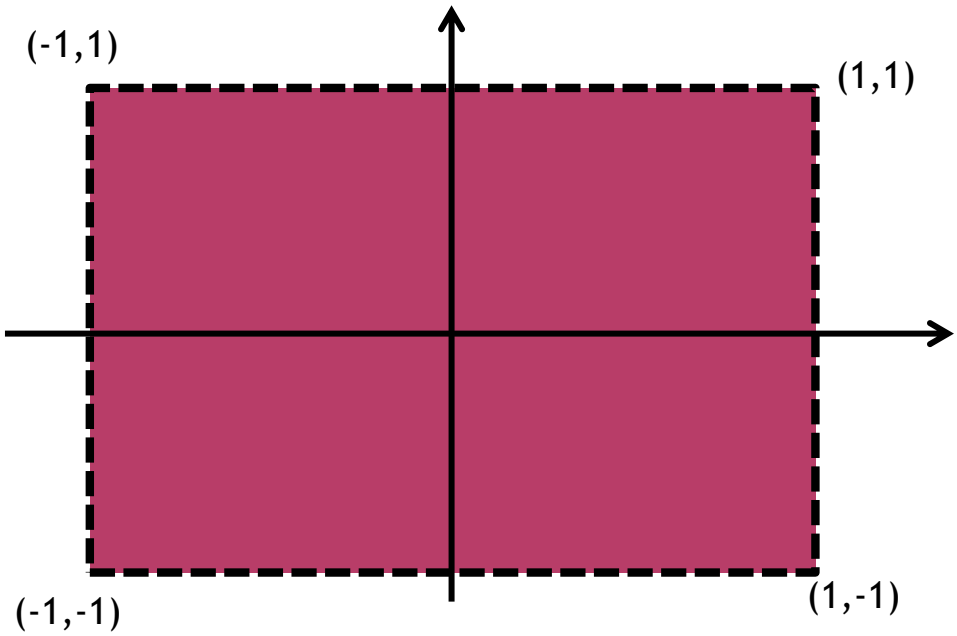
縦横が何ピクセルかとか、そういうのに関係なく画面の

左上を(-1,1)

右上を(1,1)

左下を(-1,-1)

右下を(1,-1)



として見ているからなのだ。

だから横長なのだ。

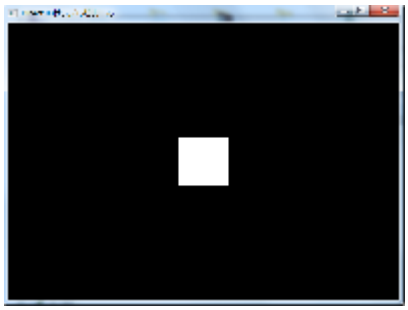
どの道 3D 表示をする際にパースペクティブ関数でアスペクト比を整えるから、3D 表示の時にはそれが解決されるのだ。

では 2D の時にはどのように解決すれば良いのか？3D の時のように変換関数を作って、それを頂点シェーダの時に計算してやればいい。

例えば…

pos.xy=float2(pos.x/4,pos.y/3);

のように書けば



のように表示される。アスペクト比は正しくなる。だが、2D であるならばやはりピクセル単位の指定がしたい…イワユル2D 座標系で扱いたい。

そうであるならば

シェーダを

pos.xy=float2(-1+pos.x/320.0,1-pos.y/240.0);

のようにしておいて

指定座標を

Position vertices[] ={

{100,200,0.0},//左下

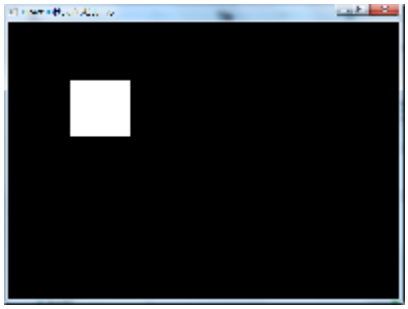
{100,100,0.0},//左上

{200,200,0.0},//右下

{200,100,0.0}//右上

};

のようにするとピクセル単位で指定でき



こうなります。

まぁここまでの話はあくまでも「2D 表示」のための豆知識なので読み飛ばしてもいいです。

がちょっとシェーダコードの説明をさせてもらうとシェーダプログラムってのは結構柔軟にできています。

まず一番良く使うのが float 系だと思いますが

- float
- float2
- float3
- float4

成分の数によって float～float4 までが使用できます。

予想つくとは思いますが、これをベクトルとして使用できます。

float3 とかの場合だと

float3 pos;

pos.x=1.0

pos.y=2.0

pos.z=3.0

などという風に、構造体の要素アクセスと同様に「．」ピリオドをつけることでそれぞれの要素にアクセスできます。

で、ここが C 言語の思考だと「？」ってなるところなのですが

https://msdn.microsoft.com/ja-jp/library/bb509634(v=vs.85).aspx

↑のドキュメントに書いてあるとおり

pos.xyz=float3(1,2,3)

的な指定もできるし

pos.xy

pos.yz

pos.zx

のように特定のベクトル成分の参照も可能だ。

さらに左辺が 3 要素、右辺が 1 要素ってのも可能だ。

pos.xyz=color.r;

この場合、右辺の r 成分が左辺の x と y と z にそれぞれ入る。

ちなみに

pos.xyz と pos.rgb は等価である(同じ意味)。

かなり柔軟すぎると思いますが、そのうち慣れます。「直感的でいいよね」とポジティブに考えましょう。

5.7 テクスチャを貼ってみよう

テクスチャを貼る手順

1. テクスチャロード
2. シェーダ側にテクスチャの定義を追加
3. シェーダ側にサンプラの定義を追加
4. シェーダ側に UV 情報を追加
5. 頂点レイアウトの変更
6. 頂点情報に UV 情報を追加
7. サンプラの作成
8. GPU 側にテクスチャとサンプラを渡す

また手順が多いですね。まぁこれも慣れです。きちんとやってりゃそのうち慣れます。

5.7.1 テクスチャロード

DX11 ではテクスチャロードは

CreateShaderResourceViewFromFile を使用します。

https://msdn.microsoft.com/ja-jp/library/ee416885(v=vs.85).aspx

```
HRESULT D3DX11CreateShaderResourceViewFromFile(
    ID3D11Device *pDevice,//デバイスポインタ
    LPCTSTR pSrcFile,//画像ファイル名
    D3DX11_IMAGE_LOAD_INFO *pLoadInfo,//nullptr でいい
    ID3DX11ThreadPump *pPump,//nullptr でいい
    ID3D11ShaderResourceView **ppShaderResourceView,//受け取り用
    HRESULT *pHResult//nullptr でいい
);
```

とりあえず……ここまで来たら特に説明しなくてもわかるな？

わかるな？

というわけで

ID3D11ShaderResourceView* shaderResourceView;

という変数をラストの引数に放り込んだら

shaderResourceView にはテクスチャデータが入っているという認識で良い。

とりあえずロードはこれでいいです。

5.7.2 シェーダ側にテクスチャの定義を追加

シェーダ側にテクスチャの定義を追加します。

簡単です。

シェーダ側の先頭に

Texture2D tex;

とでも書いてください。シェーダ内のグローバル変数みたいなもんです。

Texture2D ってえのは

https://msdn.microsoft.com/ja-jp/library/bb509700(v=vs.85).aspx

に書いてあるけど、cpp 側で書かれたテクスチャを受け取るためのものやと思っておけばいい。

で、↑のドキュメントの下の方に書いてあるように色々とメソッドはあるが、暫くの間は Sample 関数しか使用しない。「テクスチャをサンプリングします」とあるけど、まぁ要は uv 座標からその場所の色を取ってくるって関数や。

5.7.3 シェーダ側にサンプラの定義を追加

これが初心者には意味不明なもんなんだけど、シェーダプログラミングするには避けては通れないものだ。

なにかというと、ヒトコトでいうと、指定のピクセルをどう表示するかってことなのよね。
え？テクスチャの通りじゃないの？って思うでしょ？まぁ事実そのとおり何だけど…うーん、例えば、絵っていうのは 300*400 とかで出来てたりするやん？

でもUVの指定って、浮動小数点の0.0～1.0なわけね。もうね、この時点で正確に「横にnピクセル目の縦にmピクセル目」なんていう正確な指定は事実上不可能なわけ。

で、どういう風にして指定uv座標の色を取ってくるのか、もし101～102ピクセルの間を指し示していたらどうするのか？

そういうのを設定するもんです。それをサンプラーっていうんです。HLSLだとSamplerStateっていうやつや。

というわけで、Texture2Dと同様にSamplerStateも先頭に

SamplerState sampler;

とでも書いてくれ。

5.7.4 シェーダ側にUV情報を追加

さて、これはどういうことかというと、現在の状態だと普通に空間内の座標しか指定していない状態である。ていうか現状だとそれしか指定できない。

これにUV情報を付加するのは簡単である。

まず、頂点シェーダの引数を増やす。

これを追加してくれ

float2 uv:TEXCOORD

これがuv座標を指し示すようになる。

次にこれがちょっと面倒だが、シェーダ内で構造体を宣言してもらおう。

//頂点シェーダ出力用

struct VsOuput{

float4 pos:SV_POSITION;

float2 uv:TEXCOORD;

};

まぁ、なんとなく意味はわかるだろう？

で、頂点シェーダの出力をコイツに変えてしまうのだ。

つまり、この構造体で変数を宣言し、そこに頂点座標と、さっき追加した uv 座標を代入するように書く。

そして戻り値の型を、float4 から VsOutput に変更

実際に戻り値を、頂点座標と uv を入れた先ほどの変数にしてしまう。

次にピクセルシェーダ側である。出力データが変わったのだから、ピクセルシェーダに綿わされる値も変わるのだ。

float4 pos;

としている部分を

VsOutput input

と引数変更しておこう

5.7.5 頂点レイアウトの追加

頂点データに「UV」が追加されたわけだから、頂点レイアウトにも UV レイアウト情報を追加しなければならない。

このように複数のレイアウトを記述するために、レイアウト情報は「配列」だったのだ。

で、UV のセマンティクスは"TEXCOORD"である。ちなみに TEX はテクスチャのことというのはわかるだろう。ならば COORD とは？

これは、数学における「座標」を示す coordinate から来ている。つまり、テクスチャ座標=UV というわけだ。

というわけで、最初に書いた POSITION の行をコピペして、POSITION の部分を TEXCOORD に還ればいい。

また、テクスチャ座標は xyz 座標と違って、二次元である。つまり、float が二つあればいいので、

DXGI_FORMAT_R32G32_FLOAT にすればよい。

ただ、次の問題として、5 番目の要素、これは POSITION では 0 だったが、今回はこれでいいのだろうか？

駄目である。データの開始位置を示さなければならない。

Ｘ、Ｙ、Ｚ、Ｕ、Ｖ

と並んでて、ＵＶは 0 バイト目からではなく sizeof(float)*3 から始まるはずだ。

なので、5 番目は sizeof(float)*3 としたいところだけど、もうちょっと簡単な方法がある。
D3D11_APPEND_ALIGNED_ELEMENT

を使うのだ。これは、最後のデータの先頭を自動で計算してくれる便利なマクロなのだ。こいつを使おう。

あ、レイアウト数がふえたお

5.7.6 頂点情報に UV 情報を追加

さて、いよいよ頂点情報に uv までも追加していくわけだが。

ここまできたら構造体を使ったほうがいい。

```
struct Vertex{
                float x,y,z;
                float u,v;
};
```

こんな風に宣言しておいて

```
Vertex verts[]={{-1,-1,0,0,0},
                {-1,1,0,0,1},
```

```
                {1,-1,0,1,0},
                {1,1,0,1,1}
};
```

とでも配列を定義する。

バイトサイズが変更されたことに注意。ByteWidth に sieof(Vertex)って書いてる人はまだいいんだけど

Sizeof(float)*3*4

とか書いてる人は、書き直しが必要だ。

Sizeof(float)*5*4

もしくは

Sizeof(verts)って書いておけばいい。

あとストライドも変わってるので注意な～

5.7.7 サンプラの作成

ナンプラーじゃねえぞ

これは前に話しましたが、特定の uv 座標の場所の色をどのように決定するのかアルゴリズムの指定になります。

で、その指定をＧＰＵ側に教えるためには、IDevice::CreateSmaplerState を使用する。

https://msdn.microsoft.com/ja-jp/library/ee419801(v=vs.85).aspx

とりあえず、サンプラーステートを得るためには、ＤＥＳＣを渡して、サンプラーステート本体を得ます。

その直後にＧＰＵ側に渡してあげましょう。

https://msdn.microsoft.com/ja-jp/library/ee416271(v=vs.85).aspx

はい、多くて面倒ですが、今回指定しなければならないのはアドレスモードと、Filter の指定だけです。

フィルタってのはどうやって補間するのか、アドレスモードってのは絵を繰り返すのか繰り返さないのかを指定する。

とりあえずひとまず初期化しといて、あとはそれぞれに値を代入します。

D3D11_SAMPLER_DESC samplerdesc={};

samplerdesc.Filter=D3D11_FILTER_MIN_MAG_MIP_LINEAR;//とりあえずリニアフィルタにしとく

samplerdesc.AddressU=D3D11_TEXTURE_ADDRESS_WRAP;//U 方向は繰り返し

samplerdesc.AddressV=D3D11_TEXTURE_ADDRESS_WRAP;//V 方向も繰り返し

samplerdesc.AddressW=D3D11_TEXTURE_ADDRESS_WRAP;//W 方向も繰り返し(ごめん、W 方向は正直よくわかってない)

さて、フィルタの D3D11_FILTER_MIN_MAG_MIP_LINEAR ってのは、拡大時も縮小時も線形補完するよーってこと。

ＡＤＤＲＥＳＳ＿ＷＲＡＰってのは、ＵＶが０～１を超えたときに、絵を繰り返すかどうかです。今のところはＷＲＡＰでいいでしょう。

こんなもんです。あとはいつもの要領でサンプラーステートを作成してください。

5.7.8 ＧＰＵ側にテクスチャとサンプラーを渡す

そこまでできたら、テクスチャとサンプラー双方をＧＰＵ側に送ります。送る関数はそれぞれ

DeviceContext::PSSetSamplers(0,1,サンプラ);

https://msdn.microsoft.com/ja-jp/library/ee419717(v=vs.85).aspx

DeviceContext::PSSetShaderResources(0,1,テクスチャ);

https://msdn.microsoft.com/ja-jp/library/ee419731(v=vs.85).aspx

です。

5.7.9 最後にシェーダをもう少しいじる

これでＧＰＵに…つまりはシェーダ側から、テクスチャ情報を見ることができます。

で、ピクセルシェーダを見てください。

ピクセルシェーダには

Return float4(1,1,1,1);

が書いてあると思いますが、これをテクスチャの色に変更します。

使用するのは Texture2D の Sample 関数です。

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

現状、Texture2D オブジェクトは tex と宣言しているので

tex.Sample(sample,input.uv);

とでもすればいいでしょう。

6 3D 化しよう

さて、現状だとまったく 3D 感がありません。何故でしょうか？それはねぇ、頂点シェーダにおいて、頂点情報をそのままスルーで返しているからだよ。

ではどうしたらええのんか？

まぁ、キャメラ設定が必要やな…。

6.1 カメラ設定

6.1.1 XNAMath をインクルード

DirectX11 のライブラリには数学系の関数が含まれていません。

よくわからないかもしれないけど簡単に言うと DirectX9 の時に使用できていた

D3DXMatrixRotationY とか D3DXMatrixTranslation とか D3DXMatrixLookAtLH とかが使えなくなってますというか、そういう関数がありません。

ねーなら自分で作ればいいじゃない!!と言いたいところなんだけど、流石にめんどくさすぎる…ていうかそんな時間ねーよ。

ということで、代替ライブラリを探すとXNAMathというのを使用するのを推奨されているようだ(とりあえずWindows7～8の間は)。

というわけで、XNAMath.hをインクルードしといてくれ

6.1.2 ビュー行列を作成

D3DX系を使い慣れている人にとってはそれほど難しくないと思います。

D3DXの部分をXMにすればいいだけです。

例えば、D3DXMatrix～ってのはXMMatrix～って感じになるだけです。

そもそも「行列」を表すにはDX9の場合はD3DXMATRIXでしたが、ここではXMMATRIXを使用します。

というわけでビュー行列はXMMatrixLookAtLHを使用します。

https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixlookatlh(v=vs.85).aspx

説明の必要はないですよね？左手系のビュー行列を作るものです。

第一引数に視点の座標を、第二引数に注視点の座標を入れてください。最後にアッパーベクトルですが、これは(0,1,0)でいいでしょう。

引数はXMVECTOR型ですが、これは予め

XMVECTOR eye={0,0,-10};//視点、

XMVECTOR target={0,0,0};//注視点、

XMVECTOR upper={0,1,0};//アッパー

と設定しておきましょう。それをそのままそれぞれの引数に代入すれば、ビュー行列が帰ります。

6.1.3 プロジェクション行列を作成

これは3Dのモデルをスクリーンに投影するための行列です。必要な情報として、画角(視野角)とアスペクト比と、近いとこと遠いとこを指定すればオッケーです。

https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.matrix.xmmatrixperspectivefovlh(v=vs.85).aspx

視野角はとりあえず90°でいいですけど、第一引数に 90 なんて書かないでくださいね？
Unity とかに慣れすぎている人はやっちゃいそうですけど、必ず「ラジアン」にしてください。

アスペクト比は「画面幅」を分子に、「画面高さ」を分母にして計算してください。なお、割った結果が 0 とか 1 とかにならないよう注意してくださいね？

ここをしくじると、何も表示できなかったり、ふとましい事になりますよ？

6.1.4 コンスタントバッファに行列をセット

専門用語の塊で何を言っているのかわからないと思いますが、ともかく頂点でもテクスチャでもないものを GPU に渡す時に使うのが「コンスタントバッファ」です。

行列を渡してもいいし、単なる数値を渡しても良い、そういうものです。コンスタントと言っても別に定数ってわけでもないんですけどね。とりあえず、こちらから渡した値がそのまま GPU で使えるッて状態だと思ってもらって構いません。

まずはコンスタントバッファを作りましょう。頂点バッファの時と同じだと思ってもらって構いません。CreateBuffer で生成します。

頂点バッファの時との違いは、アクセスフラグと、Usage くらいです。

今回は行列を渡すので ByteWidth は sizeof(XMMATRIX)になります。

```
D3D11_BUFFER_DESC buffdesc={};
buffdesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;//コンスタントバッファとして生成
buffdesc.ByteWidth = sizeof(XMMATRIX);//サイズ
buffdesc.CPUAccessFlags=D3D11_CPU_ACCESS_WRITE;//CPU からの書き込み可
buffdesc.Usage=D3D11_USAGE_DYNAMIC;//CPU から書き込まれ、GPU から読み取られる
```

とりあえずこんな感じで

さっきの LookAt と PerspectiveFovLH で作った行列をそれぞれ view と proj とすると

XMMATRIX camera=view*proj;

とすることで、カメラ行列が作成されます。これを頂点バッファの時と同様に
SUBRESOURCE_DATA の pSysMem に代入します。

つまり

D3D11_SUBRESOURCE_DATA data={};

data.pSysMem=&mat;

ですね。

もうあとはバッファを作成して、VSSetConstantBuffers にて GPU 側に投げます。

context->VSSetConstantBuffers(0,1,&matbuff);

これでデータが GPU に渡されます。

さて、ここまででデータが GPU に渡っているのであとはシェーダの記述をするだけです。

6.1.5 シェーダ側で CPU 側からの値を受け取る

とりあえずは簡単です。

matrix mat;

と宣言しておいてください。

この段階で一度ビルドして、特にエラーが発生ないことを確認してください。

この mat には先程 SetConstantBuffers から持ってきたカメラ行列が入っています。

6.1.6 カメラ行列を頂点に乗算する

カメラ行列というのは3D 座標系を2D のスクリーンに投影するものです。ですから、頂点に対して掛け算すれば良いのですが、ちょっと注意が必要で、左側から掛け算してください。

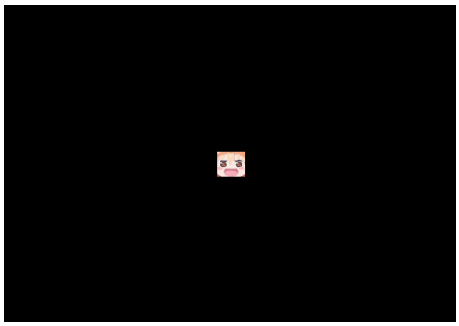
output.pos=mul(mat,pos);

をしてください。

なお、

output.pos=mat*pos;

ではだめなのか？ッて思われると思いますが、掛け算の場合は、型が違うとか言って怒られるので mul で記述してください。結果として



かなり小さいので、頂点を大きくするか、カメラを近づけてください。

カメラを近づけてこんな感じになってればいいでしょう。

ただ、これだと 3D ってのがわかりづらいので、してんをちょっと右に移動させてください。

こんな感じにあるかと思います。

ここまでできたら、もう次の段階に入っていきます。

7　3D データ(PMD)を読み込もう

ミクミクダンスのモデルデータは PMD と言って、バイナリファイルでできています。

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

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

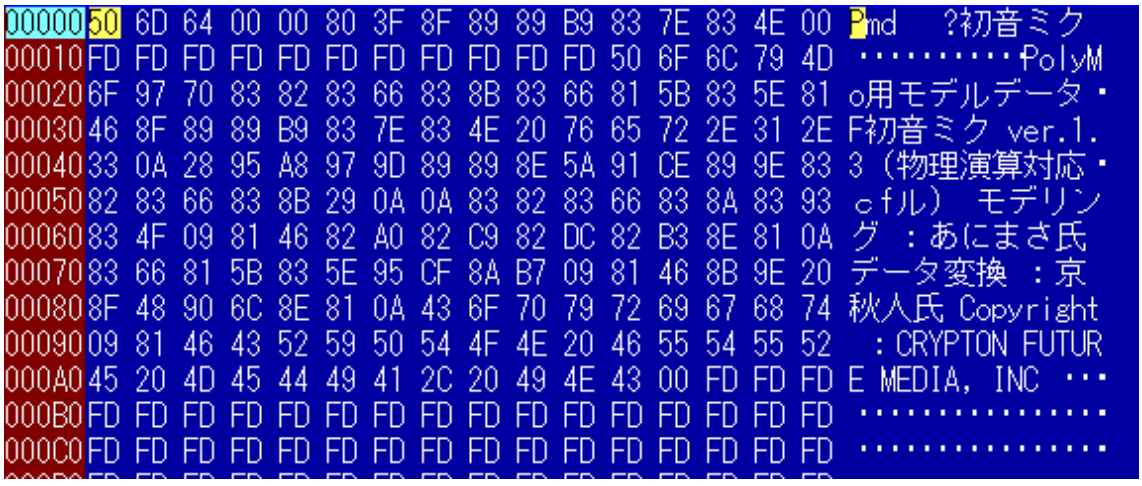
FBX はあくまでも「中間データ」くらいに思っておけばよいでしょう。

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

7.1 バイナリエディタで PMD の中身を見てみよう

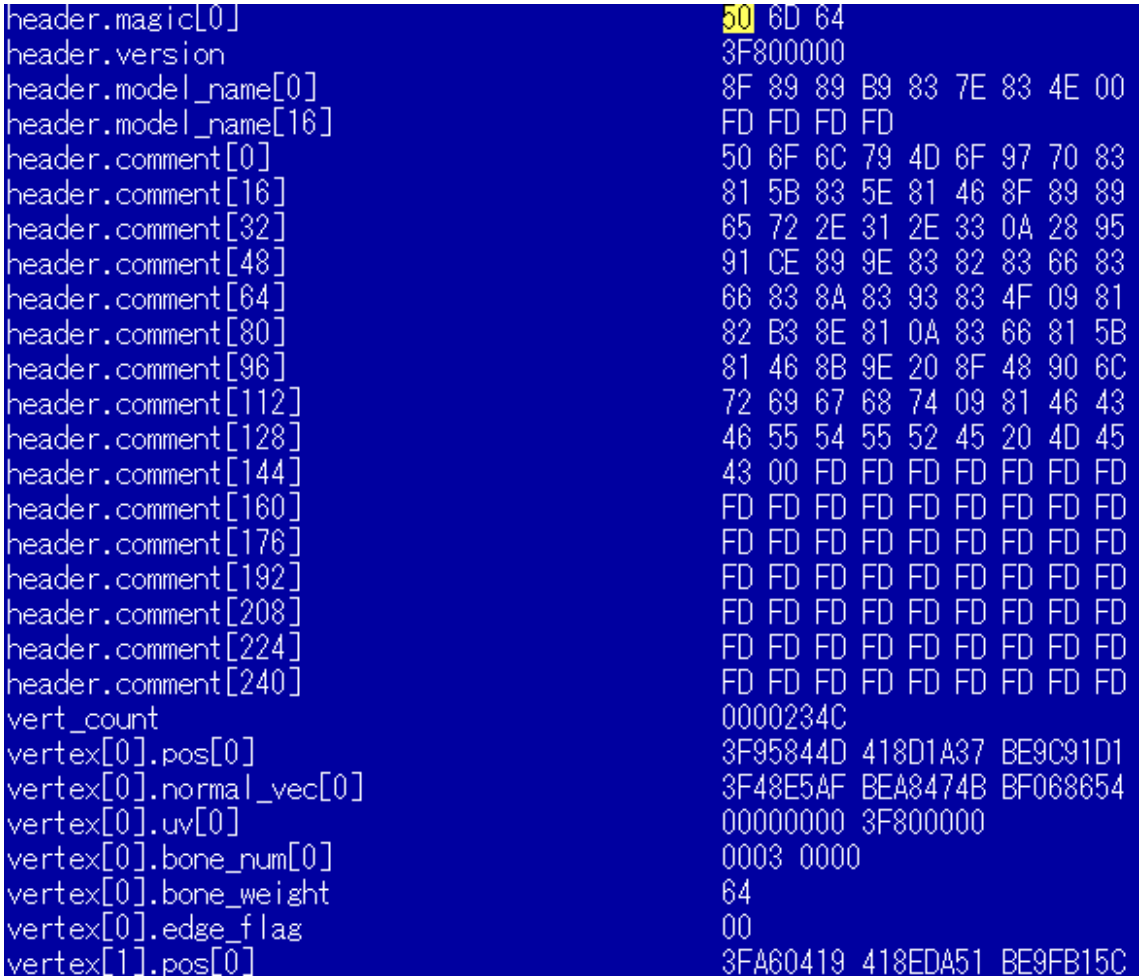
サーバの Tool のバイナリエディタから、tsxbn400 をダウンロードして、解凍してください。

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



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

で、再読み込みすると



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

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

こまけえ話は後で解説するので、

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

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

7.2 バイナリファイルを読み込もう

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

ちなみに fopen 系を使うときは

#include<cstdio>

をインクルードしておきましょう。

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

http://www.orchid.co.jp/computer/cschool/CREF/fopen.html

①ファイルオープン

基本的には、

FILE* fp=fopen("ファイル名","rb");

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

②ファイルリード

実際にファイルからデータを取得します。

ちなみに関数は fread です。

http://www.orchid.co.jp/computer/cschool/CREF/fread.html

ちょっと今回はいきなり「頂点数」にアクセスしたいので

fseek って関数を使用します。

http://www.orchid.co.jp/computer/cschool/CREF/fseek.html

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

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

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

ちなみに SEEK_SET はファイルの先頭の意味です。

さて、283 バイト目まで飛びました。

ここから fread 関数で 4 バイト読み込んで unsigned int 型変数に入れてください。ミクであれば 9036 くらいの数が入っていれば正解です。

```
unsigned int vsize=0;
```

どこかで頂点数を表す変数を宣言しておいて…

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


fread(&vsize,sizeof(unsigned int),1,fp);
```

こういう感じで記述して、vsize に頂点数っぽいのが入っていれば成功です。ミクモデルだと 9000 くらい。

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

なお、今回は頂点座標だけを見るので、一旦レイアウトの部分の **TEXCOOORD は一旦コメントアウト**してください。

ついでに**頂点シェーダの第二引数もコメントアウト**してください。

さて、ここからです。

7.2.1　頂点情報のロード

頂点情報は1頂点あたり 38 バイトです。

さて……どうしたらいいんですかね？やり方はお任せしたいところですが？

えー？

ループも使えないの～？

というわけで、そんなのはいないと思うんですけど、

「いや、ループはわかるけどさ…頂点データ数と頂点サイズがわかっても…どうするか分からへん」とかいう煽り文句を書きましたが…よく考えたらループ要らないやこれ。

つまり

38*頂点数

をリードしてください。でっ!!!まんま頂点バッファに放り込んでください!!!

もちろん、データサイズとストライドは書き換えてね。

その上で、今回は『頂点情報』だけなので、トライアングルだとうまくいかないのでトポロジを

D3D11_PRIMITIVE_TOPOLOGY_POINTLIST

にしてください。

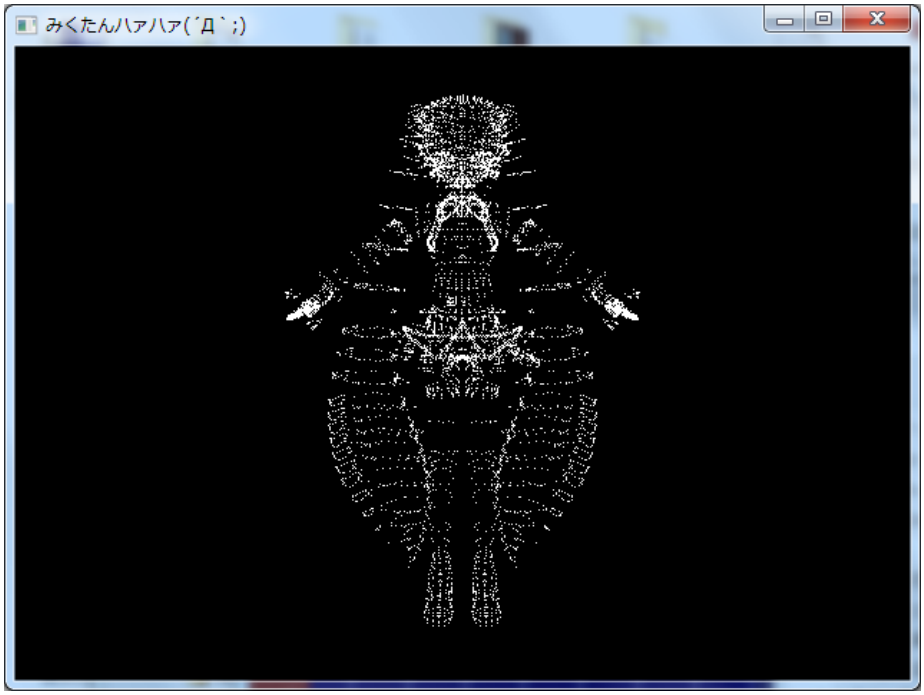
これだけです。はい、やって？

あ、突き放しすぎ？そっかー。でもこれだけでできちゃう人もいるでしょ？

しばらく時間あげるから考えてみて？ちょっとは自分の頭で考える練習しとかないと、この後はついてこれなくなるよ？

ちなみに**ピクセルシェーダの戻り値も一旦 float4(1,1,1,1)とかにしといたほうが分かりやすい**ですね。

ともかくうまく行けばこういう



なんか女の子っぽいシルエットができてますよね？出てないかな？

ちなみに、ミクモデルは結構大きいのでカメラを少々引かないと足の一部しか表示されません。

とりあえず僕のプログラムでは

XMVECTOR eye={0,15,-30};//ちょっと引いてかつカメラを上に上げておく

XMVECTOR target={0,10,0};//足元見てしまうので、注視点も少し上げておく

という設定にしています。

ちなみに

さて、さっそく解説してみましょう。ついてこれている人に限って、放課後残ってまでやってるんで…まぁ、残りはソースコード見せつつ解説します。

まずは、全頂点データの読み込みです。

頂点データと言っても xyz だけでなく、xyz,法線ベクトル,uv,ボーン番号,影響度,フラグがあり、1頂点あたりのデータサイズは 38 です。

全頂点のサイズがどれくらいか計算します。

38×頂点数なので、既に取得しておいた vsize と 38 をかければ総データサイズになります。

まるごと読み込みます。

fread(読み込み先メモリアドレス,サイズ,カウント,ファイルポインタ);

となるのですが、読み込み先メモリは時前に確保しておかねばなりませんね？

予め「確保すべきサイズ」が分かっているわけではないので「配列の宣言」として数を入れておくわけには行きません。

なので、4つほどやり方があります。

① malloc を使用する
② new[]を使用する
③ std::vector を使用する
④ どんなのが来ても大丈夫なようにくっそデカイ配列を用意しとく

ちなみに④はオススメしません。「くっそデカイ配列」を予め確保しとくという発想自体は良いんですが…少しややこしい(アロケータの作り方とか知らなければならない)ので、そういうメモリ周りは富永センセの授業でやっといてください。

ここは手っ取り早さと俺の好みの問題で std::vector を使用します。使用する際には

#include<vector>

しといてください。std::vector は C#の list みたいなもんで、配列みたいに扱えるコンテナです。

とりあえずメモリ塊として使用するので、char 型を頂点数×サイズぶん確保してみます。ここらへんは後で改変すると思いますが、一気にゴリッと読み込む気持ちよさ的なものを感じてもらいましょう。

std::vector<char> pmdverts(vsize*38);//メモリ確保

fread(&pmdverts[0],pmdverts.size(),1,fp);//全頂点データ読み込み

一つ言っておくと 38 はもちろんマジックナンバーなので後々修正します…が、まずはデータ読み込みの喜びを感じましょう。

ちなみに

&pmdverts[0]

に疑問を感じる人もいると思いますので、解説しておくと

std::vector というのは、配列のように確保したメモリが連続しているということが保証されているものなのよ。

なので要は先頭のアドレスを持ってこれればいいわけ。配列だったら先頭のアドレスは配列名で良いんだけど vector は配列のようで若干配列ではないので…大雑把に言うとこんな感じの構造です。

先頭にサイズがあるイメージでいいです。他にも色々とあるんですけど、ともかく先頭に実データはないです。で、配列のように扱うのならば実データへのアドレスが必要です。そこにどうやってアクセスするのかというと、配列の先頭要素 0 番のアドレスを貰えばいいです。

つまり

pmdverts[0]

にアンパサンド&をつけてデータ先頭アドレスを渡しているわけです。

さらに vector には便利な関数があって、size()メソッドは配列の要素数を返します。

というわけで

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

などという書き方になっています。ここで全頂点データの読み込みが完了です。メモリ上にミク全頂点データが乗っかってます。

とりあえずは「中がどんなデータ構造になってようが関係ない」です。

必要な情報は…

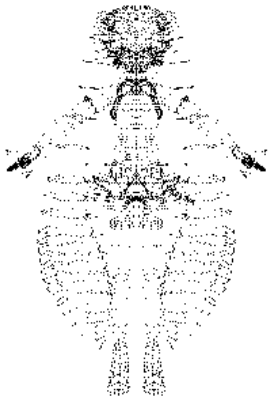
頂点バッファに必要な全バイト数＝38*頂点数

1頂点あたりのストライド=38

です。

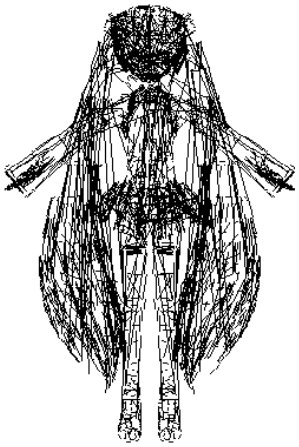
このふたつがわかれば…何処を変更したら良いか分かりますね？

あとは Draw 関数の「頂点数」をきちんとロードした頂点数に変更しとけば表示されると思いますが、いかがでしょうか？



うまくいかない人は必ず質問してね(ニコッ)

ちなみに LINELIST にするとこんな感じ



次にもう少し MMD のモデルっぽい表示にしていきましょう。そのためには『インデックス情報』が必要になってきます。

7.2.2　インデックス情報のロード

ここからちょっと難易度が上がります。

インデックスと言っても、別に株とかそういうのじゃないし、ましてや「インなんとかさん」でもない。

今までの「点」のデータを「面のデータ」にするのに必要なものです(超重要)。頂点データは頂点のデータであって、それ以外の何物でもない。

ところがひとつの「面」を構成するには、少なくとも頂点が3つ必要です。その3つの頂点の組み合わせを司るのが「インデックスデータ」なのです!!!

なのです!!

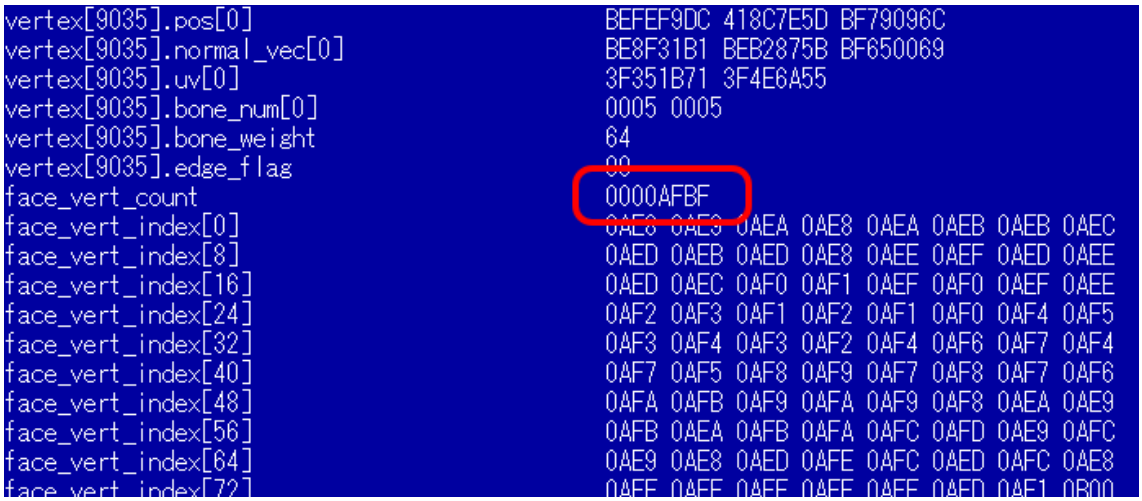
ほぼすべてのメジャーな3D フォーマットには必ずインデックス情報が含まれています。

データ構造は至って簡単。「何番目の頂点なのか」という「番号」がインデックスデータに入っているだけです。

つまり unsigned int 型かもしくは unsigned short 型ひとつで十分ということだ。

さて、PMD データの場合はインデックスデータは何処にあるのだろう。

ところでコイツを見てくれ。コイツをどう思う？



すごく…頂点データの後ろです。

そう、頂点データが尽きる所、そこにインデックスデータカウントが入っていて、そのあとにズラーっと頂点データが入っている。

ちなみに言うとインデックスデータのバイトサイズは 2 バイト(unsigned short)である。

これがどういうことか分かるかな？

2バイトで表現できる数は 0～65535 である。つまり PMD における頂点数は上限 65535 個であることがわかる。

ひとまず「インデックス数」を取得しよう。

簡単だ。特に何も考えず4バイト読み取り unsigned int に書き込んでください。

44991 個くらいである。大抵の場合、頂点数よりもインデックス数のほうが大きな数になる。

次に頂点データの時と同様にインデックスデータを読み込む

2 バイト×頂点数なので

std::vector<unsigned short> indices(頂点数);

fread(&indices[0],sizeof(unsigned short),頂点数,fp);

とでもすれば全ての「インデックスデータ」を取得できる。

あとはこれを頂点データと同じような流れで GPU 側に渡してやるだけだ。GPU 側に渡すには CreateBuffer でインデックスバッファを作らなければならない事はだいたい予想つくだろう？

CreateBuffer⇒IASetIndexBuffer

のコンボでインデックスデータが GPU 側に渡ります。

まずはバッファを作ります。

CreateBuffer は何を求めてるんだっけ？そう、DESC だ。D3D11_BUFFER_DESC である。

頂点データとの違いは BIND 先だけなので、BindFlags の VERTEX を INDEX に変更するだけでいいです。あ、もちろん ByteWidth も変わるからな？そこら辺は全くの額面通りには受け取るなよ？

これでできたバッファを今度は

コンテキスト->IASetIndexBuffer で CPU 側に投げます。

第一引数はさっき持ってきたバッファ。第二引数には…

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

2 バイトだから DXGI_FOR_R16_UINT を使用しましょう。

最後の引数はオフセットなので、今は不必要です。だから 0 でいいです。

あと、初期化データですが、これもインデックスデータの先頭アドレスを渡してあげれば終わりです。

そこまでできたら、POINTLIST から TRIANGLELIST 変更しておきましょう。今からは三角形を表示していきますからね。

次に Draw 関数を DrawIndexed 関数に変更します。インデックス情報ありの描画の場合はちょっとプログラムの変更が必要なのだ。

コンテキスト→DrawIndexed(インデックス数,0,0);

とでも指定してやれば

このように 3 角形として塗りつぶされ、BAD APPLE!の PV みたいな感じになるね。

ここまでできましたか？

7.2.3 法線情報を利用して 3D っぽくしてみよう

次に法線情報を利用して 3D っぽくします。法線というのは各頂点に設定されている「面に垂直な方向のベクトル」です。

ちなみに、頂点情報の直後が法線情報です。

やることは

- 頂点レイアウトに「法線」を追加
- シェーダ側に「法線」の記述を追加
- 「ランバートの余弦則」で明るさを計算

このみっつだけです。意外と簡単ですね。

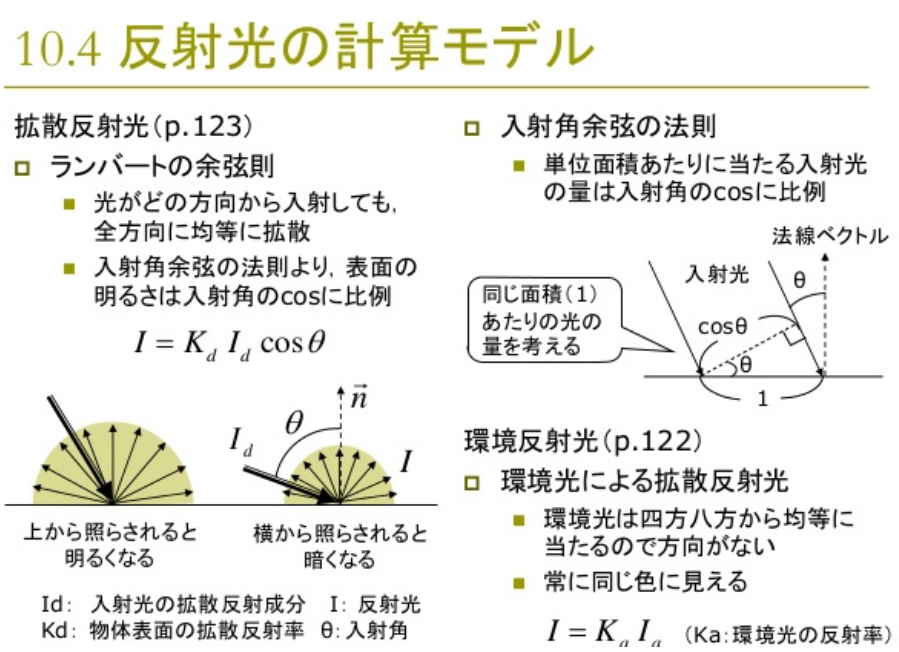
頂点レイアウトの方は簡単でしょ？とりあえず POSITION の行をコピペして、POSITION の記述の部分を NORMAL にしてください。ちなみに NORMAL ってのは「法線」を英語でいうと NORMAL なのです。正常って意味じゃないことに注意。

シェーダ側に法線の記述を追加します。引数に法線を追加してください。これも POSITION のをコピペすればいいです。セマンティクスは NORMAL でお願いします。

ついでに構造体側にも normal 追加しといてください。代入を忘れずに

そこまで出来たら、今度はピクセルシェーダに「ランバートの余弦則」による明るさの計算を記述します。

ランバートの余弦則ってのは



所謂こういうの。

簡単に言うと、「明るさは法線と入射光のなす角度の cosθ に比例する」って法則なのです。だからぶっちゃけ

明るさ=cosθ

でもとりあえず陰影がつくわけ。特定のベクトルの間の cosθ なんてどうやって求めたら良いんだろう？

こういう式を覚えていませんか？

$$a \cdot b = |a||b|\cos\theta$$

覚えていませんか(’•ω•`)

これくらいは覚えておこうよ…内積の性質だよ？ともかくこういう式が成り立ちます。そしてaとbの大きさが1であるならば

$$a \cdot b = 1 \times 1 \times \cos\theta$$

ですから、内積をとればそのまま明るさになるというわけです。

とりあえず手っ取り早くするために、光線ベクトル(反対向き)を定義しておきます。

例えば左上手前からの光ならば、ベクトルを左上手前に向かうようなベクトルにしておきます。

```
float4 light=(-1,1,-1);
```

とでもシておきます。

あとはピクセルシェーダにて、法線ベクトルとの内積を計算します。シェーダにおいて「内積」というのは dot という関数を使用します。

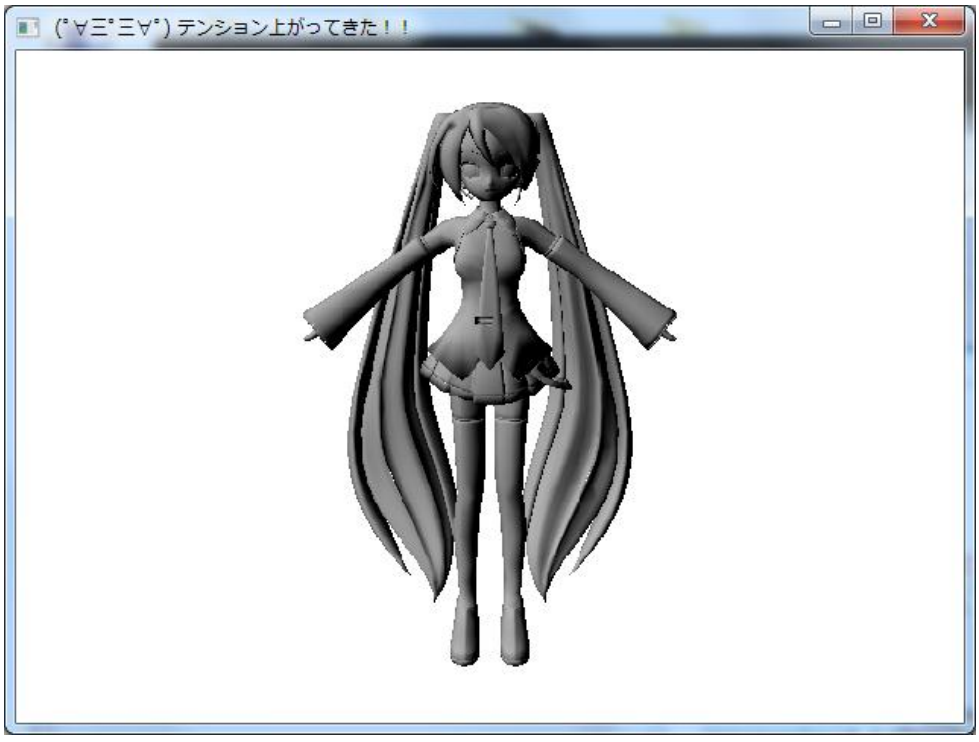
```
float brightness = dot(light,input.norm);
```

まぁぶっちゃけ内積なんざ関数使わずとも

$$\text{内積} = A_x \times B_x + A_y \times B_y + A_z \times B_z$$

でいいんですけどね。ともかくこれが明るさになります。ただし「スカラー値」でありベクトルではないことに注意してね？

うまいこと明るさに反映出来たならば、3D っぽくなります。



いやぁ、テンション上がってきますなあ。

7.2.4 色をつけよう

このままじゃモノクロですしミク感が薄いですね。

やっぱり髪の毛は緑色じゃないと!!!!

というわけで色をつけましょう。色のデータは何処にあるのか？

```
face_vert_index[44960]                  1F70 1F6E 1F70 1F71 1F69 1F6
face_vert_index[44968]                  1F6E 1F71 1F6E 1F6D 1F6A 1F6
face_vert_index[44976]                  1F70 1F6F 1F6E 1F70 1F6E 1F7
face_vert_index[44984]                  1F72 1F6A 1F69 1F71 1F6A 1F7
material_count                          00000011
material[0].diffuse_color[0]            3E083127 3F1EB852 3F36C8B4
material[0].alpha                       3F800000
material[0].specularity                 40A00000
material[0].specular_color[0]           00000000 00000000 00000000
material[0].mirror_color[0]             3D883127 3E9EB852 3EB6C8B4
material[0].toon_index                  00
material[0].edge_flag                   01
material[0].face_vert_count             00000C06
material[0].texture_file_name[0]        00 00 00 00 00 00 00 00 00 0
material[0].texture_file_name[16]       00 00 00 00
```

コイツを見てくれ

diffuse_color というやつがあるだろう？

これがベースの色を決定しているのだ。CG の表面材質モデルとして、ディフューズ(拡散反射)、アンビエント(環境光)、スペキュラー(鏡面反射)というのがあるが、今回は一番基本になるディフューズを取得し、色情報として反映してみましょう。

そして場所は何処かというと、インデックスデータが尽きる所にマテリアル数があり、マテリアルデータは 1 つあたり 46 バイトくらいですね。

で、注目して欲しいのは face_vert_count ですが、これは何を表しているのかというと、このマテリアルが反映される「インデックス数」を表しています。

ですから、各マテリアルの face_vert_count を合計すると「総インデックス数」と一致します。ミクデータならば総インデックス数が AFBF(44991)ですが、各マテリアルを見ると

C06+3EF1+2C64+13C8+D32+1E0+330+75+C+6+1C2+48F+D2+198+2A+840+AE

となっています。確かに 16 進数の足し算を行うと AFBF です。総インデックス数と一致しますね？

「それに何の関係があんの？」

今な、DrawIndex で全てのインデックスデータを表示してるやん？

context ->DrawIndexed(indexSize,0,0);

こんな感じでさ？

ひとまずこれをマテリアル毎に描き分ける。イメージ的には Direct3D9 の DrawSubset やね？DX9 のやつ…覚えてます？

```
for(int i=0;i<materialNum;++i){
                dev->SetMaterial(material[i]);
                dev->DrawSubset(i);
}
```

こんなん書いたでしょ？(3 年は書いてないかもしれへんけど…)まぁ、ともかくそういうことなんや…つまりはマテリアル毎に描き分ける必要が X ファイルだろうが PMD ファイルだろうが「ある」っちゅうことや。

まずマテリアル一番のみを表示させてみよう。

そのためにはマテリアル情報をとっておこう。まずはマテリアル数を取ってきましょう。

先程も言いましたがマテリアル情報はインデックスデータの直後です。ですからインデクスデータ読み込みの直後に 4 バイト読み取ればマテリアル数が取得できます。簡単でしょ？

というわけで取得な？

```
fread(&materialNum,sizeof(materialNum),1,fp);
```

materialNum に 16 くらいが入っていれば正解です。

さぁてマテリアルデータ取ってくるか!!!

今回のマテリアル処理はカラーデータを CPU 側で取得するので「塊」で持ってくるのはオススメしません。

とりあえず抜き出すための構造体を作りましょう。

```
struct Material{
		XMFLOAT3 diffuse;//ディフューズ
		float alpha;//アルファ値1
		float specularity;//スペキュラ強度
		XMFLOAT3 specular;//スペキュラ
		XMFLOAT3 mirror;//ミラー
		BYTE toonIdx;//トゥーンインデックス
		BYTE edgeFlg;//エッジフラグ
		unsigned int indexCount;//インデクス数
		char textureName[20];//テクスチャ名
};
```

こんなの作ってください。

で、本来ならばこれの sizeof(Material)をすると 70 と返るはずなんだが…実際にやってみてくれ…どうなった？

そう…72 なのよね。

なんでだと思いますか？これに関してはねー。知らない人もいるかもしれないけど

「アライメント問題」ってのがあるからなんだ。

これはどういう事かというと、メモリ的に効率よくするために 4 バイト未満のものは 4 バイトに丸めましょうって仕組みがあります。

これのせいなんですわ。

今回の構造体で言うと、トゥーンインデックスと、エッジフラグが1バイトのため、ここでアライメントが発生します。

でも、ファイルのデータとしては 70 バイトなので、それに合わせなければなりません。こういう時、どうしたら良いんでしょうか…。

C++言語には#pragma pack ってのがあるんですよ。アライメントの丸める バイト数を指定するディレクティブですね。こいつに一時的に1を設定します。

ですから

さっき作ったマテリアル構造体の宣言前に pack(1)として、終わったら規定値にするために pack()にします。つまり

#pragma pack(1)

構造体宣言

#pragma pack()

これでオッケーです。さぁ、マテリアル情報を取得しましょう。例によって vector でも使いましょうか。

std::vector<Material> material;

～中略～

material.resize(materialNum);

fread(&material[0],sizeof(Material),materialNum,fp);

あとはマテリアル数分ループしながらインデックス数を参照していきます。

なお、DrawIndexed 関数の第二引数がオフセット値になっているので、ループの中で加算しておきます。

context->DrawIndex(material[i].indexCount,indexOffset,0);

こんな感じでループして書いて、元のものと同じ画像が表示されるのを確認してください。

いよいよ着色です。

ここからの手順はこんな感じです

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

ひとまず材質情報を渡す用構造体を作成します。

ひとまずはディフューズデータだけなので、余り意味が無いですが

```
struct GMaterial{
        XMFLOAT3 diffuse;
};
```

こういうのを作りましょう。でいつもの様にコンスタントバッファを作ります。

ここに罠が張られています。僕も学生も引っかかりました……。なんか知らんけど ConstantBuffer の生成に失敗するんですよ。INVALID_ARG が返ってるとまず確実に「バイト数」問題です。まぁ何かというと

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

このページのね？下の方を見てください。

解説
この構造体は、バッファー リソースを作成するために ID3D11Device::CreateBuffer によって使用されます。

この構造体に加え、D3D11.h には派生構造体 (CD3D11_BUFFER_DESC) もあります。これは、継承されたクラスのように機能するので、バッファーの記述を作成する際に役に立ちます。

バインド フラグが D3D11_BIND_CONSTANT_BUFFER の場合、**ByteWidth** には、
D3D11_REQ_CONSTANT_BUFFER_ELEMENT_COUNT 以下で 16 の倍数となる値を指定する必要があります。

こういう事はさぁ…もっと上の方に書くとか、赤とか太文字で記述してくださいよぉマイクロソフトさ～ん(´；ω；`)ｳｯ…。

ちょくしょう…ちくしょおおおおおおおおおおおおおお!!!!

くっそｗｗｗこんなのでｗｗｗｗ。

とにかくコンスタントバッファを16の倍数サイズで作ってください。

構造体を

struct GMaterial{

XMFLOAT4 diffuse;

};

にするか

sizeof(material)+ (16-sizeof(material)%16)%16;

とでもしてあげます。意味は分かりますね？構造体のサイズが 16 の倍数ならば

16*a+(16-(16*a)%16)%16=16*a+(16-0)%16=16*a となりサイズ変更なし。

しかし 16 の倍数ではない場合…例えば今回のように 12 の場合だと

12+(16-12%16)%16=12+4%16=12+4=16 となります。なんかしらの数に切り上げたい時にこういう式を使うと良いでしょう。

ちなみに 17 の場合だと

17+(16-17%16)%16=17+(16-1)=17+15=32 となります。こういう式を自分で考え出せるようにしておきましょう。

さて、VSSetConstantBufferですが、今回はMatrixを乗っけるのと別で乗っけるので、乗っけるスロットを変える必要があります。

もともとカメラ行列は

context->VSSetConstantBuffers(0,1,&matbuff);

と書いてましたね？第一引数がスロット番号です。これが一致していると上書きされますので

context->VSSetConstantBuffers(**1**,1,&materialBuff);

とします。

もしくは「ピクセルシェーダ側」にカラー情報を送ってやればいい。

context->PSSetConstantBuffers(0,1,&materialBuff);

ひとまずこれでシェーダ側に送ります。この時頂点シェーダに送るかピクセルシェーダに送るかで、シェーダの書き方が変わってきますのでご注意ください。

…分かりますよね？

ちなみにピクセルシェーダ側に置いたほうがラクです。

さて、ではピクセルシェーダ側のプログラムを見てみましょう。

コンスタントバッファの受け取り先が増えましたので

cbuffer をまた新しく宣言します。

```
cbuffer Material{
                    float3 _diffuse : COLOR;
};
```

前の Matrix と混ぜないでくださいね？名前は似ていますが、これはあくまでも「マテリアル」です。

さて、これでピクセルシェーダ側で使える

_diffuse という変数ができました。

あとはこれをピクセルシェーダの明るさを決定している計算に掛け算してください。

return float4(_diffuse,1)*明るさ;

ですね。

このように_diffuse は float3 なので 1 をくっつけておいてください。

こうやっておいて、CPU 側の色を変えたりして確認してみてください。

でもこれで終わりではないです。

「マテリアルループ時にコンスタントバッファの内容を変更」ってのが意外とキます。そう簡単ではないのですよ。

ちょっ…と、面倒くさいんですよ。

既に GPU に投げてしまっているバッファの内容を書き換える必要があります。

まずマテリアルバッファから、書き換えるべきアドレスを取得します。関数は

DeviceContext::Map です。

https://msdn.microsoft.com/ja-jp/library/windows/desktop/ff476457(v=vs.85).aspx

```
HRESULT ID3D11DeviceContext::Map(
  ID3D11Resource *pResource,//バッファを入れる
  UINT Subresource,//0 でいいよ
  D3D11_MAP MapType,//D3D11_MAP_WRITE_DISCARD
  UINT MapFlags,//0 でいいよ
  D3D11_MAPPED_SUBRESOURCE *pMappedResource//弄る用のポインタ
);
```

D3D11_MAP_WRITE_DISCARD を詳しく知りたい人は

https://msdn.microsoft.com/ja-jp/library/ee416245(v=vs.85).aspx

を御覧ください。

ちなみに「サブリソース」について詳しく知りたい人は

https://msdn.microsoft.com/ja-jp/library/ee422118(v=vs.85).aspx

を見ておいてください。

見てもわからないと思いますが、そこに書いてある理由でサブリソース番号はひとまず 0 が適切なのです。

MapFlags は GPU がビジーな時の挙動なので、まぁここも0でいいでしょう。

となると…

D3D11_MAPPED_SUBRESOURCE map={};

context->Map(materialBuffer,0,D3D11_MAP_WRITE_DISCARD,0,&map);

とやることで materialBuffer の中身が map.pData が指し示している先になります。

もし中身を入れ替えたければここに新しいマテリアルデータを memcpy 等で書き換えます。

ちなみに memcpy は

http://www9.plala.or.jp/sgwr-t/lib/memcpy.html

に書いてあるように、あるメモリ領域をあるデータで上書きする関数です。これを使用して値を書き換えるならば

memcpy(map.pData,&material[i].diffuse,sizeof(material));

とします。

しますが正直ボクは memcpy とか memset とか ZeroMemory 等のように無防備なメモリを弄る関数はキライです。

なのでこういう書き方もあるということは言っておこう。

((XMFLOAT3)map.pData)=material[i].diffuse;

で、後始末ですが、これは

Map もルールがあって、内容を書き換えた後は Unmap しなければなりません。

context->Unmap(バッファ名,番号);

なので、もうここは関数化して「SetMaterial」とかいう関数作って、それでこのワンセットの処理を行ってください。

```
void SetMaterial(const GMaterial& material){

                    //～ここに色々と書いてください～

}
```

でぃあとはマテリアルループの中で

SetMaterial(materials[i]);

のように呼び出します。上手いこと実装できれば…

こんな感じのミクさんが表示されます。でもよく見てください…？

おわかりでしょうか？白目剥いてますね。

怖いです。目の画像データがないからこうなってるんです。

ひとまずぃ元になる画像データをサーバから落としてきましょう。

 eye2.bmp

こういうデータね。

7.2.5　モデルに絵を貼ろう

さて…頂点座標と、法線情報両方を反映することができました。

じゃあ残りは何か？

とりあえずは UV ざんしょ？絵を張り付けるためのアレですよ!!

そう、モデル読み込みの時に一度無効にしたあの TEXCOORD ですよ!!

さぁ…頂点データを見なおしてみよう

http://blog.goo.ne.jp/torisu_tetosuki/e/5a1b16e2f61067838dfc66d010389707

法線ベクトルの直後にありますね。だから「か～んたん」

既にメモリ上には法線ベクトルの直後にあるので、レイアウトとシェーダを変更するだけです。

レイアウトは既にコメントアウトしているであろう TEXCOORD を復活→シェーダ側も法線の直後に TEXCOORD を復活。

さて…アレ？目の画像のロードってどうやんの？

まぁ eye2.bmp をロードしてセットすれば簡単なんだけど…それだと汎用性がない…意味はわかるね？

ミクさん以外のモデルのロードに使えへんわけですわ!!!

ところでマテリアルのデータ情報を見てくれ

```
struct Material{
        XMFLOAT3 diffuse;//ディフューズ
        float alpha;//アルファ値1
        float specularity;//スペキュラ強度
        XMFLOAT3 specular;//スペキュラ
        XMFLOAT3 mirror;//ミラー
```

```
    BYTE toonIdx;//トゥーンインデックス
    BYTE edgeFlg;//エッジフラグ
    unsigned int indexCount;//インデクス数
    char textureName[20];//テクスチャ名
};
```

こいつをどう思う？

せやな……テクスチャ名(テクスチャパス)があるんやな。

う～～～ん。まぁぶっちゃけ此処から先は自分で考えて欲しいんやけどなー。やってやれんことはないはずやから…。

とりあえずマテリアルに全テクスチャ名があるわけやん？

これをテクスチャデータ ID3D11ShaderResourceView にしたいわけ。

どうしようか？

ひとまず

ID3D11ShaderResourceView *のベクタか配列を作りましょう。

ちなみに配列数はマテリアル数と同じでいいでしょう。理由はあとで説明します。

マテリアル数ぶんループしながら D3DX11CreateShaderResourceViewFromFileA でテクスチャをロードして、先ほど確保したベクタもしくは配列に代入しましょう。

ちなみに関数の最後に A がついているのは、データのテクスチャパスが char だからです。D3DX11CreateShaderResourceViewFromFile は wchar を受け取るようにできているので最後に A をつけとく必要があるのです。

この辺もめんどくせー説明が必要なので今は A のやつを呼び出すと思っておいてください。

さて、ロードできているはずなので、それぞれのシェーダリソースをループの中でセットします。

```
context->PSSetShaderResources(0,1,&_textures[i]);
```

こんな感じですね。

で、次にシェーダ側をいじくります。

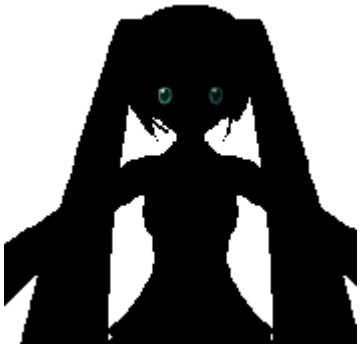
float4 rgba=tex.Sample(sam,input.uv);

みたいなコードが残っていると思いますが、これがテクスチャの色を返すんでしたよね？

ですからこれを乗算してやればテクスチャが表示されます。

return rgba*float4(_diffuse*brightness,1);

を実行すると…



ああん？なんで？

目のテクスチャは表示されているのですがほかがマックロです。これは目以外のテクスチャが設定されていないため、0 が乗算され、こんなことになっています。

もしくは目の画像の(0,0)が全てに貼られている状態です。

これを解決するには2つの方法があります。

① ファイル名が指定されていない時は真っ白テクスチャを貼る
② ファイル名が指定されていない時はテクスチャを反映しないシェーダを使う。

で、サーバに nulltex.bmp というテクスチャを置いたので、ロードテクスチャが nullptr のときはそいつを貼り付けるようにしてください。つまりＣＰＵ側のプログラムで if で場合分けしてください。

つまり

if(material[i].texturename==nullptr){

ヌルテクスチャをセット

}else{

textures[i]をセット

}

そこまでがうまくいけば

てな感じの表示になります。

今はテクスチャの切り替えで対応しましたが、シェーダの切り替えでも対応できます。

例えば現在の PS を BasePS(テクスチャなし)と TexturedPS(テクスチャあり)に分けて定義して ID3D11PixelShader*のピクセルシェーダも 2 つに分けておきます。

そこで D3DX11CompileFromFile と CreatePixelShader も 2 つにかけてロードしといてくださいね。

それでテクスチャが nullptr かどうかでシェーダを切り替える…そんな手もありです。

7.2.6 テクスチャファイル名について

PMD ファイルの中に入っている「テクスチャ名」には多少注意が必要です。

一部のモデルでは

こういうデータになっていると思います。

この文字列”2.bmp*a.sph”をそのまま使うと「そんなテクスチャねーよ!!」って思われます。そりゃそうだ。この〇〇.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。とりあえず今は必要ないので外しておきたいです。

どうしたら良いんですかね？

手っ取り早い方法として std::string を使うって方法があります。C#と同じように C++にも String 型ってのがあります。

std::string 型っていうんですけど使用するには、string をインクルードします。

#include<string>

http://www.cppll.jp/cppreference/cppstring_details.html

このうち、使用するのは find 関数と substr 関数なので確認しておきましょう。

ひとまずは string 型の変数を宣言します。

それとともに、マテリアル内のテクスチャ名をその変数に入れましょう。

std::string texturepath=materials[i].texturename;

さて、ここから*で区切りたいのだから、そこが『何文字目か』を測り、文字列をそこで切り抜きます。

まずは*を探す関数。これは find 関数を使用すれば、そのインデクスが返ります。

とりあえずそれは別変数にとっておきましょう

int idx=texturepath.find('*');

でこれで、見つかった場所の文字が返るので、先頭の位置からそこまでを抜き出すために substr 関数を使用します。

これは文字列の一部を抜き出す関数で

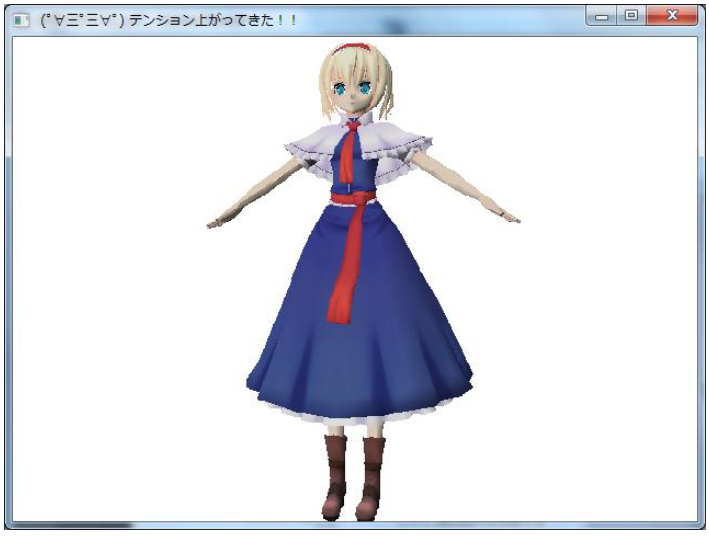
string.substr(スタートの場所,文字列の長さ);

こういう感じですから？スタートの場所は 0、長さを idx にすれば取れますよね。

というわけで

texturefilepath.substr(0,idx);

これで*2.sph 文字列が消えますので、きちんとロードできます。ここまでうまくいけば



こんな感じにアリスも表示できます。

ついでに言うと島風もほらこのとおり

と、言いたいところですが、実は島風の目は「透明度」が入っているためそう簡単ではありません。

正しい表示はこうです。

どうしても島風などの「透明度」が入っている物体を表示させたい人は…頑張ろう。

```
ID3D11BlendState* blendstate=NULL;
D3D11_BLEND_DESC blenddesc={};
blenddesc.AlphaToCoverageEnable=false;
blenddesc.IndependentBlendEnable=false;
D3D11_RENDER_TARGET_BLEND_DESC& blrtdesc=blenddesc.RenderTarget[0];
blrtdesc.BlendEnable = true;
blrtdesc.SrcBlend = D3D11_BLEND_SRC_ALPHA;
blrtdesc.DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
blrtdesc.BlendOp = D3D11_BLEND_OP_ADD;
blrtdesc.SrcBlendAlpha = D3D11_BLEND_ONE;
blrtdesc.DestBlendAlpha = D3D11_BLEND_ZERO;
blrtdesc.BlendOpAlpha = D3D11_BLEND_OP_ADD;
blrtdesc.RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;
float blendFactor[] = {0.0f, 0.0f, 0.0f, 0.0f};
device->CreateBlendState( &blenddesc, &blendstate );
context->OMSetBlendState( blendstate, blendFactor, 0xffffffff );
```

頑張ろう。

7.2.7　課題(2015/10/30)

さて、今週の課題です。

10/30(金)じゅうに、自分の好きなキャラクターの PMD ファイルを見つけ出して表示するところまでやってください。

好きなキャラクターの PMD がなければ別に好きじゃない奴でも構いません。

ただし、Lat 式ミクは余裕のある人以外はやめておいたほうが無難です。相当特殊な作り方をしています。これに対応するにはそれなりの知識が必要です。

これ一体のためだけにまた授業する時間はないので、そこは自分で徹夜してでも研究するか、諦めるかしてください。

あと、PMX もとりあえず諦めてください。資料はインターネットにいくらでもあるので、表示まではできるかもしれませんが、ボーンの構造がＰＭＤよりはるかに複雑なのでやめといたほうがいいですね。

あと、ヒトコト言っておくけど、ゲーム業界に行きたければ**他と差をつける**必要がある。

いいかい？

「他と差をつける」んだぜ？

隣のやつと同じことやってて差がつくとでも思っているのか？もしかしたら隣のやつは家でコソコソ勉強や制作しているかもしれないんだぜ？

聞いたことあると思うけどゲーム業界ってのは応募人数に比べて募集人数が極端に少ないのだ。だとすると他人を出し抜かなければならないだろう？

じゃあ何をすべきかはわかっているよね？

ちなみに GFF ゲームコンテストは1月末くらいです。

GFF AWARD 2016 とかね。

さて、ここまでできたらもうそろそろゲーム作り始めます。

なので、今のうちにコードを整理しておきましょう。コード整理の方法は冨永先生に習ってますよね？

来週までに各自やっておくようにね？

8 クラス設計的な何か

まぁ、ゆーてもコードを整理する心の余裕とかなかったかな？

で、現在の状況だと、特に気を利かせてない限り、自分のコードは完全に「クソコード」担っていると思います。

8.1 クラス設計以前に「クソコード」を避けよう

前期にも紹介しましたが自分のコードが「クソコード」になるのを避けましょう。

クソコードってのは

「俺様を怒らせてしまうようなコード」の事である。

1. 読めないコード
2. 要領の悪いコード
3. 意図がわからないコード

と、

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

ここのサイトに書いてました。ともかく俺様が怒るってことは、そういうソースコードを見る企業の人も激おこってことよ。つまり合格しない…まぁ実際はやんわりと落とすんだけどね。

まぁ、どんなコードがプログラマンに怒りを齎すのかはここも見ておいたほうがいいでしょう。

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

こーんなコードを書いていると

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

僕の前の会社で実際にあった話ですけど、ネットワーク周り担当してた奴のコードが酷くて、まぁメンテに困ってたわけですわ。そこで耐えかねた上司(テクニカルディレクター)がそいつが作ったコードを全消しして、「こんなものはゴミだ!!」とたった一行のコメントにしてしまったという…。もちろん、代替コードはその上司自ら書きました。

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

ちなみに「クソコードビンゴ」なるものがあり

識別子スペルミス	エラー揉み消し	乱調インデント	トートロジーコメント	マジックナンバー
巨大共通ライブラリ	最長不倒関数	マリアナ海溝ネスト	中括弧省略if	通らないコードブロック
多重継承	大富豪プログラミング	名実不一致関数	一人二役関数	ミックスイン
遺棄ライブラリ依存	複数言語混在実装	ステート依存	テスト困難	実行時警告洪水
グローバル変数	型キャスト	量産コピペ実装	デバッグコード残骸	永久不滅TODO

まぁ、ここに書いてあるものが必ずしも悪いわけじゃないんですけど、とにかく一つ一つ説明すんのめんどくせーけどこんな風に「禁じ手」みたいなのはあるわけ。

とりあえず俺の独自のコーディング規約を書くと

- 原則として1関数が100行を超えない
- 原則としてグローバル変数は使用しない(シングルトンも必要最小限)

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

こんな感じ

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

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

何故かと言うと、今のままでは1種類のキャラロードにしか耐えられないからです。

つまり

class PMDMesh

などというクラスを作っておいて

PMDMesh playerModel(ファイルパス);

でロードするか

PMDMeshLoader ってクラスが生成するようにしといて

PMDMesh* playerModel=pmdMeshLoader.create(ファイルパス);

でも構いません。

クラス設計の話じたいは冨永先生の授業でも細かくやると思いますが、キャラクターやモデルとして一つにまとめられるものはまとめておいたほうが良いでしょう。

で、なぜこういうローダーを作っといたほうが良いかというと、**後々の拡張に耐えられる**からです。

一番最初のバージョンで「プログラムが完成する」ってことはまずなくて、必ず「機能」の「拡張」的なものがガンガンやってきます。

場合によっては PMD だけではなく、PMX ファイルやその他 X ファイルなどをロードする必要も出てきます。もしくはロード時のアルゴリズムそのものを変えなければいけないかもしれません。

この時にスッ…と変更できるように設計しておくのです。

もう一つ、「コンストラクタでロード」しないほうがいい理由は、同一のメッシュを何度も使いまわす事があるからです。ていうかその方が多いですね。

例えば 1 ステージに 20 体のゾンビが出てくるとして、20 回のロードを行うのか？それはあまりにも非効率だ。ご存知のように「ロード処理」というのはすべての処理の中で桁違いに重いのだ。

どうやって解決したらいいかな？

① こいつはロードする、こいつはロードでなくクローンにすると自分で判断する
② std::map<>を使用し、ファイルパスとメッシュ情報を紐づけておき一括管理する

だいたいこの二つくらいが思いつくだろう。

何度も言うけど、プログラマの仕事は「現状の問題をアルゴリズムで解決する」だ。正しい答えなどない。どちらを選ぶべきかは状況によって変わるし、第三の選択もあるんだろう。すべて自分の頭で考えるのだ。それができなければプログラマではない。

さて、この場合僕ならば②を選択する。プログラマなら必ず「なぜ？」「なぜその方法をとったのか？」にこたえられるようにしておこう。どこぞのコードをコピーするその場しのぎのやり方をしていたら、答えられなくなる。それだけはやめよう。コピペしたらコピペしたで「なぜそれをコピーしたのか」を考えよう。

さて、なぜ②なのかというと、今回のはメッシュ…つまりリソースにかかわるものだ。リソース割り当てというのはメモリ周りの負担がでかいのである。このため、不要になれば破棄したいしそうすべきである。

で、その破棄のタイミングはこちらで管理しておきたいとする。もし①のやり方ならばどういう実装になるだろうか？クローンというのはまぁ、メモリ上のメッシュ情報のポインタをクローン先にも渡してあげて、一つのメッシュオブジェクトのクローンはみんな共通でそのデータを使用できるという使用である。

この場合、破棄のタイミングとして期待されるのはすべてのオブジェクトが消滅したときである。たとえばゾンビ 20 体を全破壊した瞬間に「メモリ上のメッシュ情報」は必要なくなるため破棄される。実装の仕方は「参照カウンタ」を管理しておき、クローンするたびに参照カウンタを増やし、オブジェクトのデストラクタで参照カウンタを減らす。

最終的にゼロになったときに実際のメモリ上のメッシュ情報まで消す…というわけだ。これはこれで間違いではないんだけど、一つ問題がある。

例えばとりあえずすべてのゾンビを倒したとしても、ゾンビの場合は後から補充されることも考えられる。

その場合、20 体倒した後にいったんメモリからメッシュ情報がなくなり、その後にまたゾンビを発生させる際にロード処理が発生する。ここで当然ながら半端ない処理落ちが発生する。これはどんなにＣＰＵスピードＧＰＵスピードが向上しても同じことだ。

何しろハードディスクに対するアクセスなのだからＣＰＵやＧＰＵは関係ない…。

というわけで①は使いたくない。そもそも参照カウンタの実装は思いのほかテクニカルなので、現状でひぎぃルルル言ってる人には実装できないだろう。

というわけで②だ。②はシンプルだ。

とりあえず std::map というＳＴＬの一種の力をちょっと借りる。

#include<map>で使用可能だ。

これはC#でいうところのDictionaryと同じようなもんだ。「キー」と「値」がペアになっており、キーを渡せば「値」を取得できる。意味が分からんかもしれん。たぶんそんな人は「連想配列」とか言っても聞いたこともないんだろうけど、要はこういう事だ。

例えば「ファイルパス」をキーとして、メッシュ情報を「値」とすると。

とりあえず説明の都合上メッシュ情報を MeshInfo とするとこういう宣言をする。

std::map<std::string,MeshInfo*>meshInfoMap;

ちなみに「ファイルパス」は当然「文字列」なので std::string を使用する。値はメモリ上のメッシュ情報なので MeshInfo へのポインタにしておく。

で、これをメッシュへのファイルパスで管理する。

例えば、ファイル名が zombie.pmd だとすると

“zombie.pmd”,ゾンビメッシュ情報

として入っている。これがすでに存在するかどうか調べるには

```
MeshInfo* PMDLoader::create(std::string filepath){
If(meshInfoMap.find(filepath)==meshInfoMap.end()){
                    //ロードされてないので新規ロードする
                    meshInfoMap[filepath] = LoadPMD(filepath);
                    return meshInfoMap[filepath];
}else{
                    //すでにロードされているのでポインタを返す。
                    return meshInfoMap[filepath];
}
```

ちなみにマップは『連装配列』なので、配列と同様に[]の中にキーを入れると、その情報を返してくれる。

マップ検索は若干のコストがかかるが、とりあえずそれほどメッシュの種類がないならば無視できるレベルである。

メッシュの種類が100とか超えてもたぶん大丈夫だろう。その辺は商用ゲームくらいじゃないと到達しないと思うので、今は特に気にしなくてもいいだろう。

とりあえず、MeshInfoの役割は頂点バッファ、インデックスバッファ、マテリアルを返す。

そういう考えで実装してみてください。

あと、先々の拡張を考えると、今回みたいにボーン考えないメッシュと、ボーン考慮したメッシュは分けて考えたほうが良いかもしれません。

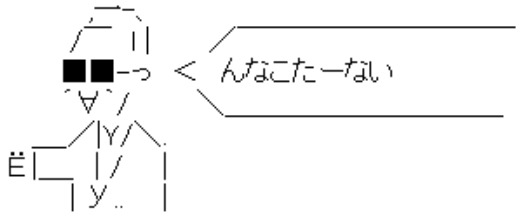
8.1.1 とりあえずシングルトンやってみっか？

「デザインパターン」という用語を聞いたことがあるだろうか？

別に知らなくても良いけど、こないだCC2のシャッチョさんは「知っとけ」って言ってた。

ただこのデザインパターンは真面目にやろうとすると23個もあって面倒なのでここではシングルトンパターンについてだけ話そう。

ちなみに一部の人達の間では未だにデザインパターンが「銀の弾丸」みたいに言われてるけど、



違います。十年以上前ならいざしらず、未だに無批判にデザインパターンを神のように扱うのは単なる「勉強不足」です。現場の人間がそんなんでどうするよ…みんなはそんな風にならないでくださいね。まぁ、デザインパターンの本がバイブルみたいに言われてますけど、あれ20年以上まえの本ですから……とっくに時代遅れなんだよね。

GoFの批判を見たければここを見ておこう

http://qiita.com/irxground/items/d1f9cc447bafa8db2388

ううん…また批判的になってしまってた。とりあえずシングルトンパターンの説明をしよう。

これは「一人っ子クラス」って意味で、プログラム全体の中で「ひとつだけ存在する事を保証するクラス」です。

これは言い換えると「**お行儀の良いグローバル変数**」です。なので無闇矢鱈に使うものではありません。この辺はご注意ください。

さて、デザインパターン全般に言えることですが、各パターンにおいて、

「正解などない！」

です。

なんでやり方が無数にあるんですけど、偉い人のを真似するってのはあります。その中でもシンプルかつ有名なのが

「Meyers の Singleton」

です。発想がオモシロイので見てください。例えば Device クラスをシングルトンにするとします。これを一人っ子クラスにするとすれば

```
class Device{
                public:
                        static Device& Instance(){
                                static Device instance;
                                return instance;
                        }
};
```

こんな感じ。

シンプルでしょ？シンプルかつオモシロイ。ここで注目して欲しいのはシングルトンの実装というよりも static の使い方ですね。本家の使い方はここポインタなんですけど、僕がポインタをキライなので勝手に「参照」にしています。

とりあえず言っとくと「static」なので関数を抜けたからと言って「オブジェクトが消える」事はありません。この辺は重要なので static の特性を忘れている人はきちんと思い出しましょう。

Device 型のオブジェクトが欲しければ Instance()を呼び出せばシングルトンオブジェクトが返るわけです。ところがこの Device 型。Instance()からオブジェクトを取得するのが義務付けられているんですけど、別にそれに強制力はないわけです。

ですから main 関数の中で

```
Device device;
```

とか

```
Device* device=new Device();
```

とか書いても別にお咎めがないわけです。

まぁ別にいいんですけど、強制する方法はないかな？ということでちょっとユニークなやり方を教えておきます。

まず

コンストラクタを private にする

です。

これにより、自分自身以外はインスタンスを生成できなくなります。

つまり

Device device;//ダメ

Device* device=new Device();//ダメ

となります。

もちろん Instance 関数は自分自身のインスタンス生成ですのでオッケーなのです。

これで生成はできなくなりましたが、まだ問題はあります。

それは「コピー可能」だということです。

コピーを抑制するには「コピーコンストラクタ」を private にすればいいのです。やり方はこうです。

private に

Device(const Device&);

および

Device& operator=(const Device&);

と書けば、代入が禁止され、更にはコピーコンストラクタも禁止されるため、本体のコピーは不可能になります。

ここまでやって初めて「プログラム中にオブジェクトが一つしか存在できないクラス」を作ることができます。

いやあ、奥が深いですなあ。

デザインパターンなんぞ覚える必要はないけど、とりあえず今回の「シングルトン」くらいは覚えておこう。

さて、まずはシングルトンでデバイスクラスを作ってください。そこに Init メソッドを作って、その中で Direct3D の最初の初期化を行うようにしてください 100 行超えた場合は関数を分けましょう。

あ、ちなみにシングルトンのやつの呼び出し方ですが、こういうことになります。

Device::Instance().Init();

こんな感じです。

こいつが GetDevice(), GetContext()を返すようにすれば。Device クラスで一括管理しながらも GetDevice や GetContext によるアクセスが可能です。

なお、main 関数が終了する際に

Device::Instance().Terminate();

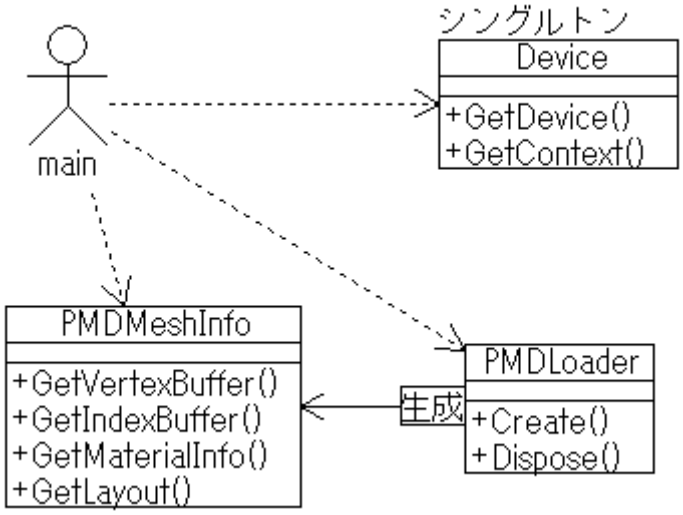
かなんか呼び出してあげて、それぞれの DirectX オブジェクトを Release するようにしてください。

さて、PMDLoader に関してですが、こいつは Create(パス)でデータロードしやがりますが、どうせ最終的に「破棄」しなければならないので、

Dispose(パス);

で破棄するようにしてください。

クラス図的な物を書くとこんな感じ



(あ、Device に Swapchain 忘れてた。みんな補間しといてね。)

だいたいこんな感じでどうだろうか？あ、クラス図の書き方はテキトーなので、みなさんはきちんと勉強しましょうね。

ちなみにしっかりクラス設計するのならばホントは、Draw 系の命令もクラスに入れるべきなんですけど、いきなりそこまでやると一部の人たちが死ぬので一旦は、一部のみクラス化することにしましょう。

今後もう少し整理していきますがね。理解を超え過ぎても仕方ないからね。でもみんなも追いつけるように勉強はしようぜ。あ、クラスを作る際はあれやで？プロジェクトで右クリックして追加→クラスで Device と PMDLoader と PMDMeshInfo を追加しといてね。

で、

PMDMeshInfo* PMDLoader::create(ファイルパス){

PMDMeshInfo を生成して返す

}

ちょっと前にも書きましたがPMDMeshInfoはメッシュの情報をまんま持っているものですがローダーから各プロパティ(メンバ変数)にアクセス出来ないのはちょっと面倒です。

ですのでfriendというのを使います。

さて……またC++クラス関係で新しいのが出てきましたよーごめんねー。でもしょうがないね。

ちなみに『プログラミング言語C++第四版P572』から説明されていますので、詳しくはそれ参照して欲しいんですけどヒトコトでいうと

「特定のクラスに全てをさらけ出す」

って指定をするやつだと思ってください。

『お友達にはボクの肉球から何からお見せするよ～』

てな感じでprivateとか関係なく、そのクラスに対しては全て丸見えになります。

というわけで、PMDMeshInfoの先頭にこう書いてください。

friend PMDLoader;

これだけで PMDLoader クラスに対して PMDMeshInfo の内容を丸見せできる状態になります。

もしこの時「PMDLoader なんてねえよ!!」的なエラーが出たら、PMDMeshInfo 定義前にクラスの前方宣言をしてください。

class PMDLoader;

なお、ちょっとかわいそうなのでヘッダ部分だけ見せてあげます。あくまでもヘッダ部分だけな？cpp 側は自分で推測してな？

PMDMeshInfo.h

```
#pragma once
#include <d3d11.h>
#include<xnamath.h>
#include<vector>

#pragma pack(1)
struct Material{
	XMFLOAT3 diffuse;//ディフューズ
	float alpha;//アルファ値１
	float specularity;//スペキュラ強度
	XMFLOAT3 specular;//スペキュラ
	XMFLOAT3 mirror;//ミラー
	BYTE toonIdx;//トゥーンインデックス
	BYTE edgeFlg;//エッジフラグ
	unsigned int indexCount;//インデクス数
	char texturename[20];//テクスチャ名
```

```
};
#pragma pack()

class PMDLoader;

//PMDファイルからのメッシュ情報が入っている
class PMDMeshInfo
{
	friend PMDLoader;
private:
	ID3D11Buffer* _vertexBuffer;
	ID3D11Buffer* _indexBuffer;
	std::vector<Material> _materials;
	std::vector<ID3D11ShaderResourceView*> _textures;
	unsigned int _vertices;//頂点数
	unsigned int _indices;//インデックス数
public:
	ID3D11Buffer* GetVertexBuffer();//頂点バッファを返す
	ID3D11Buffer* GetIndexBuffer();//インデックスバッファを返す
	const std::vector<Material>& GetMaterials() const;
	const std::vector<ID3D11ShaderResourceView*>& GetTextures()const;

	PMDMeshInfo();
	~PMDMeshInfo();
};
```

PMDLoader.h

```
#pragma once

#include<string>
#include<map>
#include"PMDMeshInfo.h"
```

```
class PMDLoader
{
private:

        PMDLoader();
        PMDLoader(const PMDLoader& );
        PMDLoader& operator=(const PMDLoader&);

        void LoadPMD(const std::string& ,PMDMeshInfo*  );

public:
        static PMDLoader& Instance(){
                static PMDLoader instance;
                return instance;
        }
        PMDMeshInfo* Create(const std::string& );
        void Dispose(std::string& );

        ~PMDLoader();
};
```

こんな感じです。あとは頑張ってね。

さて、いよいよ pmd ファイルの読み込みを PMDLoader に移行しますが、create がでかくなりすぎるのはちょっとアレなので、LoadPMD(プライベート)って関数を作ってそこに Main.cpp の PMD ロード処理をまるっと持ってきてください(最初はコピーでね)。

で、とにかくコンパイルを通しましょう。別のところでも話しましたが「コンパイルエラー」の原因が分からない場合は「言語の理解不足」が原因です。**スキマ時間に「プログラミング言語 C＋＋」でも読んで勉強**してください。それしかないです。

さてなんとかかんとか PMDLoader のコンパイルが通ったらいよいよ処理の移行です。ひとまず動いている状態であることを確認したら**必ず Git コミット**しておきましょう。

Git コミットもせずに「壊れました～」はあなたの責任です。必ず Git コミットしておいてください。

さていよいよ本体のロード処理を消して、この PMDLoader と PMDMeshInfo 側の処理を使うようにしていきましょう。

ちょっとハマるかもしれないところを予め書いておくと

```
device.GetContext()->IASetVertexBuffers(
                    0,
                    1,
                    &_mesh->GetVertexBuffer(),//ここでエラーが出ることがある
                    &strides,
                    &offsets);
```

上記の部分でエラーが出ることがあります。ダブルポインタを要求してくる部分ですね。

原因としては GetVertexBuffer はポインタを返しますが、こいつの更にポインタを返すとなると「戻り値」のアドレスを探すことになるため、ウマいことコンパイラが動作してくれない事があります。

これが出てきてしまったら、素直に一旦変数に代入してから使うと良いでしょう。

```
ID3D11Buffer* vertebuff=_mesh->GetVertexBuffer();
device.GetContext()->IASetVertexBuffers(
                    0,
                    1,
                    &vertebuff,
                    &strides,
                    &offsets);
```

ダサいけど、まぁ仕方ないです。

とりあえずこれでコンパイルが通るところまでやって、きちんと実行結果が出るところまでやってください。

ちなみにクラス設計とかコーディングの改善に関しては参考資料がありますのでご紹介しておきます。正直なところ、この短い授業時間のみで皆さんを良いプログラマに育てるには限界がありますので、こういう資料で自学自習してください。

他人を出し抜くにはこっそり勉強するしかないんですけど、もうね、一部の人は見てるとそれ以前の問題(文法がわかってない)なので、もう本当にスキマ時間でも何でも使って自分で勉強してください。

まず初心者用

http://www.pro.or.jp/~fuji/mybooks/cdiag/

少なくともここに書かれているクソコードを書くようにはならないでください。それはもうプログラマとは認められません。ここは最低限です。

次に C++の初歩を学びましょう

http://cpp-lang.sevendays-study.com/

あと、↓を読んで、「紙に書き出すこと」の重要性を感じておきましょう。

ぷよぷよ再現チャレンジ

あと C++におけるクラス設計に関しては

http://www.ogis-ri.co.jp/otc/hiroba/technical/CppDesignNote/

とか(ちょっと古いですけどね)

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

↑はグーグルのコーディング規約です。「汚いコードを書かないため」に規約を設けるのは重要な事です。

あと、C++初学者にとって Wiki は結構良いよ？

Wikipedia:C++

あとよく要望が上がるのは「書籍で勉強したいからオススメの書籍はありますか？」と来る。

そうだなあ…ひとまず初心者はそもそも文法が怪しいので

C++の絵本

とか

やさしい C++

とかでしょうか(ボクはもう初心者じゃないのでどれくらいが適切かわかりません)。古本屋に打ってる可能性もあるので博多駅近辺のブックオフに行ってみるのも良いと思います。

あと、本が分厚くてもいいという人は

ロベールの C++入門

がおすすめです。ただ本気で分厚いのでこれもブックオフで買って、100 ページごとに分割して持ち運ぶのをオススメします。ちなみに本の内容の一部は

http://www7b.biglobe.ne.jp/~robe/cpphtml/

に書いてます。

一応僕が初心者の時に見てた本は

独習 C++

(ちなみにリンク先は古いやつです)最新版はこれです。ただこの最新版も C++0x 以前の本なのでちょっと古い感はあります。

決して、決してオススメはしないけど、最新の C++を完璧にしたいなら

「プログラミング言語 C++第4版」

です。分厚いし重いし高額だしブックオフに置いてないしマジでオススメしませんが、完璧です。

で、とにかく授業についていけなくて「プログラミングに慣れたい」って目的なら、ひとまず目線を低く設定しましょう。自学自習の際は DxLib を使用してゲームを作るところからはじめましょう。

新•ゲームプログラミングの館

とか

龍神錬プログラミング

とかでとにかく「自力でゲーム作る」体験を積み重ねてください。授業は C++&DirectX11 で進みますが、DxLib&C++でも決して無駄じゃないです。でも就職活動の作品としてはパンチ弱いと思いますので DirectX も勉強しておきましょう。ただ行き詰まってプログラミング嫌いになるくらいなら放課後とかにシューティングゲームを作って「作れる」事を実感してください。

数学的な書籍のオススメは

「ゲームを動かす数学•物理」

です。これはホントに分かりやすいし、ゲームへの応用も書いてるし、必要なことは全て書いてると思います。お薦めです。

次に中級者用

C++やるのならばここは一度見ておきましょう。

MoreC++Idioms

色々と「へぇ～」となることがあると思います。

あと、学校とか企業では先輩が「まことしやかに嘘を教える」事があるのでそれに対する予防策として

http://www.kijineko.co.jp/tech/superstitions

ここを見ておきましょう。(俺も相当な頻度で先輩に騙されたし、今この瞬間にも俺がキミたちを騙しているかもしれない。センパイやセンセーの言うことは正しいのかもしれないが、鵜呑みにするのは危険である…覚えておこう)

あと、書籍だけど、わりと…いや意外と良い本だと思われるのが

ゲームプログラマのためのコーディング技術

です。これはマジでオススメです。ゲーム専門学校のセンセーが書いてるので、たぶんみんなにも読みやすいと思います。

なお、昨年のこの人の資料は

http://www.slideshare.net/MoriharuOhzu/ss-14083300

に上がっているので、これを読んだ上で購入を検討した方がいいでしょう。26～33 ページまでの内容を見るだけでもウヒョーってなることでしょう。

というわけでついでに言っておくと「SlideShare は良いよ？」

ものごっついい分かりやすいパワポの内容がばーんと太っ腹に公開されている。いろいろとあるんで気になったタイトルのを片っ端から読むのもいいでしょう。

あとこの手の論文というかパワポというか PDF が読みたかったら、最初の方でも言ったと思いますが、Cedil が良いと思います。思いますが、この内容…プロのゲームクリエータ向けの話がほとんどなので難しいかもしれません。

でも背伸びするのもまた力になるのでいいでしょう。

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

登録は完全無料なので、さっさと登録して資料を読みまくりましょう。まぁこれは「上級者用」かな？ゲームデザインについても結構書かれているので、自作ゲームの時の参考にするのも良いでしょう。

あと書籍の続きですが、C++のコーディングをもっと良くしていくなら

EffectiveC++

MoreEffectiveC++

EffectiveModernC++

の三冊ですね(一冊でもじゅうぶん)。

なお ModerC++Design はオススメしません。この本はテンプレート基地外すぎます。名前が似てるので間違えないようにしましょう。

個人的には「C++ Coding Standards」がオススメだけど、絶版なのよね…これ(’•ω•`)。

言語とか C++の設計に関してはこんなもんかな。

さて、中級者のゲームアルゴリズム本は

ゲームプログラマになる前に覚えておきたい技術

これも重いし 4000 円位するけど、たぶんブックオフにあると思います。

あと、数学•物理ですが、専攻科 3 年の教科書でも使ってる

動かして学ぶ3D ゲーム開発の数学•物理

これがありますが、分かりやすいんです。分かりやすいんですがこれは自学自習にはちょっとオススメできないかもしれません。ちょっと説明不足だったり、ソースコードですぐに答えが出てたりして、うーん(-_-;)って感じです。

「教科書」に使うにはこの不親切さがいいと思うんですけどね。

次は Unity の本になっちゃうんですけど

ゲームアプリの数学

これは一通り必要な数学の解説が書いてあって、Unity での実装も書いてるので、数学的なことを Unity 上ですぐに確認できるので良いと思います。

最後は上級者向けです

まずサイトのほうですけど

C++アルゴリズム

で基本を固めておいて

CEDIL

の論文とか見ておいたほうが良いでしょう。

また、英語さえわかればアメリカの CEDEC である GDC の資料も

Gamastra

にあるし、GDC の公演も GDC Vault から一部ですが聞くことができます。

あとはヘキサドライブの研究室もオススメです。

上級者はシェーダについてある程度知っておくべきなので

もんしょの巣穴

とか

Project ASURA

とか

まぁ、その他も皆さん自分で考えてください。

書籍はもはや定番でもある

ゲームエンジン•アーキテクチャ

です。分厚い、内容が濃い、難しい、高い。全てを兼ね揃えた大変よい本となっております。この内容で 7,344 円は非常にリーズナブル。良心的価格でございます。

C++のアルゴリズムを極めたいなら古くて高いですけどこの本がオススメです

アルゴリズム C++

これは図書館でさがした方がいいでしょうね。

シェーダの本は

Shader GURU

がオススメです。

さぁ、みなさんお勉強しましょう。うーん、まぁ書籍を買いはじめると金がいくらあっても足りないので出来る限り Web サイトで勉強しておくと良いと思いますが、言語の本だけは一冊くらい購入して行き帰りの電車内ででも読む習慣をつけてください(ループがわからんとか、関数の作り方や呼び出しがわからんとか、正直言って話にならんです。授業受けるレベルにすら到達してないです。さっさと独学でそこまで来てください)

9 音を鳴らそう

まぁ正確に言うと”XAudio2”なんですけどね。ちょっと不穏なのは Windows8 で XAudio2 がうまいこと動作しないという噂があるんですよね…

9.1 XAudio2 について

ひとまず XAudio2.h をインクルードし、XAudio2.lib をリンクしてください。ひとまず、XAudio2 を使用するには CoInitialEx を呼び出す必要がありますので、呼び出してください。

https://msdn.microsoft.com/ja-jp/library/cc308016(v=vs.85).aspx

https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms695279(v=vs.85).aspx

うん、とりあえず

CoInitializeEx(nullptr,COINIT_MULTITHREADED);

って書いておいて。これは XAudio2 を使う前に必要らしいね。

あとは

https://msdn.microsoft.com/ja-jp/library/cc308016(v=vs.85).aspx

に沿って書いていきましょう。

とりあえずここはまんま書いちゃって構いません。なにぶん公式ですから

```
IXAudio2*    pXAudio2;

if ( FAILED(hr = XAudio2Create( &xaudio, 0, XAUDIO2_DEFAULT_PROCESSOR ) ) )

    return;

IXAudio2MasteringVoice* pMasterVoice = NULL;

if ( FAILED(hr = xaudio->CreateMasteringVoice( &pMasterVoice,

XAUDIO2_DEFAULT_CHANNELS,

                                XAUDIO2_DEFAULT_SAMPLERATE, 0, 0, NULL ) ) )

    return;
```

ここまでの処理がきちんと通るか確認しましょう。

ちなみにちょっとだけ解説しておくと、XAudio2ってのはCOMコンポーネントの形式でできており、ComponentObjectModel の略称なのよね。

で、コイツの仕組みは大雑把に言うと DLL みたいなもんで、他のソフトウェアとの通信を行うものである。つまり Main プログラム側から COM を介して XAudio2 に対して通信を行い、サウンドドライバに命令を出して音を鳴らすってわけだ。

この COM を使用する際には一番最初に CoInitializeEx 関数を呼び、初期化する必要があるってことなのだ。Direct3D の create 関数みたいな感じで、これをやっとかないと始まらないって思っておいてください。

あと、アプリケーション終了時に

https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms688715(v=vs.85).aspx

CoUninitialize()

を呼び出すのを忘れないで下さい。

COM は別スレッドのやつを呼び出しますので、これを忘れると、ちょっと面倒なことになります。絶対忘れないで下さい。

次にマスタリングボイスに関してですが、この「ボイス」ってのは「声」って意味じゃないです。単なる「音」全般を操るオブジェクトと考えておいたほうが良いでしょう。

詳しくはここね

https://msdn.microsoft.com/ja-jp/library/cc677016(v=vs.85).aspx

全体的に知りたかったらここを見といてください

https://msdn.microsoft.com/ja-jp/library/bb694503(v=vs.85).aspx

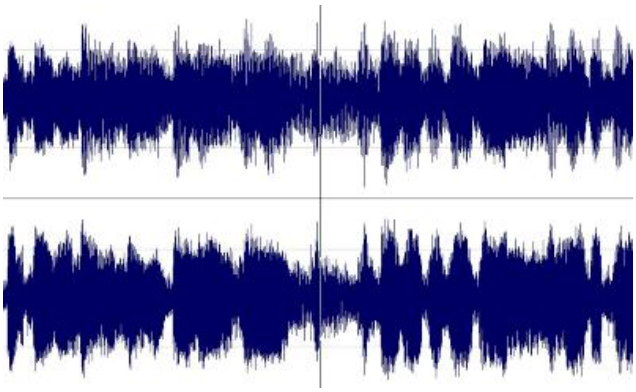
フルスクラッチで音を鳴らすのは思いの外面倒です。ただ…ここに時間をかけるわけにも行かないのよね…。

というわけで、前述のコードを書けばとりあえずは XAudio2 を使用する準備ができてる状態です。

とりあえず音を鳴らしてみますが残念ながら XAudio2 も DirectX11 と同様に…非常に様々なことを理解しておく必要があります。「音」ってのは「波」がスピーカーから発生し、その「波」が空気を伝わり鼓膜を振動させることにより「聞こえる」わけです。

で「波」って言いましたよね？

「波」ってことはヴィジュアル的には「波形」ですから



こういうの見たことありません？

これが「音」なのよ。これをものごっついい分解していくと

なんか見慣れたものになっていきますよね？sin 波です。試しにこの sin 波で音を鳴らしてみましょう(そこからかよ!!!ええ、そこからです。)

要はどれくらいスピーカーを振動させるかってデータになります。

手順としては

① ソースボイスを作成する
② サウンドデータバッファを作成する
③ 中身を波形データで埋める

となります。

9.1.1 ソースボイスを作成する

ソースボイスってのはマスタリングボイスとは違って、音を鳴らすためのソースそのものです。作成するには

audio->CreateSourceVoice

という関数を使用します。

https://msdn.microsoft.com/ja-jp/library/bb633468(v=vs.85).aspx

```
HRESULT CreateSourceVoice(
   IXAudio2SourceVoice **ppSourceVoice,//生成されるソースボイスへのダブルポインタ
   const WAVEFORMATEX *pSourceFormat,//フォーマット
   UINT32 Flags = 0,//デフォルトでオッケー
   float MaxFrequencyRatio = XAUDIO2_DEFAULT_FREQ_RATIO, //デフォルトでオッケー
   IXAudio2VoiceCallback *pCallback = NULL, //デフォルトでオッケー
   const XAUDIO2_VOICE_SENDS *pSendList = NULL, //デフォルトでオッケー
   const XAUDIO2_EFFECT_CHAIN *pEffectChain = NULL//デフォルトでオッケー
);
```

引数は 2 つだけでオッケー(他はデフォルトのままでいい)

どうせ第一引数は受け取るだけなので、問題は第二引数ってことになります。

はい、第二引数の型は WAVEFORMATEX ですね？

https://msdn.microsoft.com/ja-jp/library/cc371559.aspx

実はこれ、DirectX とも XAudio2 とも関係ない、Windows のマルチメディアの型やねんな。

```
typedef struct {
   WORD    wFormatTag; //フォーマットタグ(PCM とか ADPCM を指定する)
```

```
    WORD    nChannels; //チャンネル数(だいたい1～2ちゃんねる)
    DWORD nSamplesPerSec; //だいたい44100である…
    DWORD nAvgBytesPerSec; //平均データ転送速度(44100*ブロックアライメント)
    WORD    nBlockAlign; //ブロックアライメント(サンプリングビット数/8*チャンネル数)
    WORD    wBitsPerSample; //サンプリングあたりのビット数(16くらいでいい)
    WORD    cbSize; //フォーマット情報のサイズ(0でいいです)
} WAVEFORMATEX;
```

さて…

とりあえず一つ一つ解説すると、フォーマットタグってのは波形データのフォーマットです。とはいえ殆どの場合(特にWAVEデータの場合)はフォーマットは

```
WAVE_FORMAT_PCM
```

を指定します。

チャンネル数はとりあえず1(モノラル)で

ってなると、フォーマットの指定は

```
WAVEFORMATEX format = {};
format.wFormatTag = WAVE_FORMAT_PCM;
format.nChannels = 1;                       //チャンネル数
format.wBitsPerSample = 16;                 //1サンプルあたりのビット数
format.nSamplesPerSec = 44100;              //サンプリングレート
format.nBlockAlign = format.wBitsPerSample / 8 * format.nChannels;
format.nAvgBytesPerSec = format.nSamplesPerSec * format.nBlockAlign;
```

こんな感じでやってくらはい。

で、ひとまずコイツがS_OKを返すのをご確認ください。

では次に

9.1.2 サウンドデータバッファを作成する

サウンドデータバッファは

XAUDIO2_BUFFER 構造体で作ります。

https://msdn.microsoft.com/ja-jp/library/microsoft.directx_sdk.xaudio2.xaudio2_buffer(v=vs.85).aspx

パラメータが多くて面倒ですが、大抵の場合は上3つだけ使用すればいいので、他はゼロにしとけばいいです。

上3つとは

Flags

AudioBytes

pAudioData

です。「Xbox360」って記述があるあたり、「ゲーム作ってる～」って感じがしますなあ。

とりあえず

Flags は XAUDIO2_END_OF_STREAM を指定

AudioBytes は全波形データサイズを指定

pAudioData に全波形データのアドレスを渡します。

なので

XAUDIO_BUFFER audiobuffer={};

audiobuffer.Flags= XAUDIO2_END_OF_STREAM;

audiobuffer.AudioBytes=サイズ

audiobuffer.pAudioData=波形データ

となります。

これをボイスデータに対して渡してやれば良いのです

https://msdn.microsoft.com/ja-jp/library/bb694569(v=vs.85).aspx

sourcevoice->SubmitSourceBuffer(&audiobuffer);

でここで注意点ですけど、このオーディオバッファはあくまでも参照状態なので、**オーディオバッファおよび波形データを解放してしまわないように**スコープ等に注意しておきましょう。要らなくなったら破棄して良いんだけど、少なくとも使用する間は消さないように注意してください。

とりあえずここまでできたらまたRESULTがS_OKなのを確認し…たいところですが、まだ波形データが入ってないので、まず確実に成功しません。

というわけで、波形データを作ってみましょう。

9.1.3 波形データを作る

この部分は、通常はファイルから読み込んで作るものですが、今回は『音と波形の理解』のために敢えて自分で波形データを作ってみます。

ギターのチューニングって知ってますか？ラの音叉を使って行うのです。

音程は

https://www.youtube.com/watch?v=DycrcjPtw4M

こんな高さです

さて作ってみましょう。

とりあえず1秒のデータを作りたいと思います。どれくらいのバイト数が必要でしょうか？

現在44100Hzで作っていますので1秒間に44100のデータが必要…つまり44100バイト…なのですが、これが2ちゃんねる(ステレオ)だと倍になります。現在はBitPerSampleが16なので、1サンプルあたり16ビット→2バイト→short型

ということになります。

つまり 44100*2 ということになりますが、実はこれは既に

format.nAvgBytesPerSec

に渡していますので、これを使用します。

std::vector<char> data(format.nAvgBytesPerSec);

もしくは

std::vector<short> data(format.nAvgBytesPerSec/2);

どっちでもいいです。

とりあえず std::vector<short> data(format.nAvgBytesPerSec/2);のつもりでプログラム組みます。

short 型は-32767～32767 です。これが sin 波の振幅の上限と下限を表しています。

で、この値がスピーカーの飛び出し具合を制御します。

この値を増やしたり減らしたりすることで、スピーカーを振動させ、音を出します(空気を振動させます)

このスピーカーの変位を sin 関数もしくは cos 関数にすれば所謂 BEEP 音を鳴らすことができます。

例えば「ラ」音なら 440Hz です

http://acoutis.jimdo.com/acoustics/%E5%9F%BA%E6%BA%96%E5%91%A8%E6%B3%A2%E6%95%B0a-440hz-%E3%81%A3%E3%81%A6%E4%BD%95/

つまり波形をこの形にすればいい。

1秒間で 440 周(0～2πを 440 回)するようにすればいい。

もともとのサンプリング周波数が 44100 なので波長が 44100/440 になるようにすれば 440Hz となります。どういうことかというと、音の分解能が 1/44100 秒ってことですから1秒に 440 回(0～2π)を往復するようにすればいい。

プログラムにする場合、サンプリングレートを 440 で割ったものが一周の長さ(波長)を表しますので、それを元に「波長」を計算します(float な)。

float wavelength=44100.f/440.f;

という風にこのデータの「波長」を計算します。あくまでも 1 秒を 44100 としたうえでの「波長」です。

あとは全データをこの法則にあてはめて埋めるだけです。

```
for(size_t i=0;i<audiodata.size();++i){
        *p=SHRT_MAX*sin(i*XM_2PI/length);
}
```

とやれば 440Hz の音データの完成です。

あとはこのできたデータを audiobuffer に割り当てればいいので

```
audiobuffer.AudioBytes=audiodata.size()*sizeof(short);
audiobuffer.pAudioData=(BYTE*)&audiodata[0];
```

なお、pAudioData のデータ型は BYTE ポインタ型なのでキャストする必要がありますが、short にせよ char にせよキャストしとけばいいのです。

どうですか？音が出ましたか？

んじゃ、次の段階としてシンプルな wav ファイルを読み込んで鳴らしてみましょう。サーバに置いてます(bomb.wav)から読み込んでみてください。

とりあえず wav データはこんな感じになっています。

```
RIFF.ID[0]        52 49 46 46                                       RIFF
RIFF.Size         00010CEE
RIFF.FileType[0]  57 41 56 45                                       WAVE
Chanck.ID[0]      66 6D 74 20                                       fmt
Chanck.Size       00000010
ＰＣＭ            0001
モノラル          0001
標本化周波数      0000AC44
転送バイト数      00015888
サンプルサイズ    0002
量子化ビット数    0010
Chanck.ID[0]      64 61 74 61                                       data
Chanck.Size       00010CCA
WAVEデータ[0]     4F E1 E6 EF 66 E9 96 11 43 E4 23 EF 2A F5 AF F8 O礦・
WAVEデータ[16]    D2 F9 B3 F6 E6 09 BA 1D 68 17 24 1A 0C 04 CD 01 メ・・
```

とりあえずこのフォーマットに関して言うと、最初の 44 バイトまでがヘッダデータなので、データは 44 バイト目からということになります。

http://www.kk.iij4u.or.jp/~kondo/wave/#riff

ぶっちゃけメンドクサイ。

とりあえず読み込んでみましょうか…。とりあえずこのデータにしか使用できませんが直値でやってみます。

とりあえず fopen でオープンして標本化周波数と転送バイト数と量子化ビット数とサンプルサイズをリードします。

標本化周波数は 24 バイト目から 4 バイト、転送バイト数はその後の 4 バイト。サンプルサイズはその後の 2 バイト、量子化ビット数はその後の 2 バイトです。さぁ読み込んでみてください。

そうすると標本化周波数が 44100

転送バイト数が 88200

サンプルサイズが 2

量子化ビット数が 16 になることを確認してください。

そして、そこから 4 バイト飛ばします。4 バイト飛ばすに fseek(ファイルポインタ,4,SEEK_CUR)とします。

そこから 4 バイトがチャンクサイズ(データのサイズ)となります。それも読み込んでください。うまくいけば 68810 になるはずです。

あとは 68810 バイト読み込みましょう。

```
fread(&audiodata[0],sizeof(unsigned short),audiodata.size(),wavfile);
```

読み込んだデータを元にフォーマットを設定…

```
format.wBitsPerSample = bitpersample;     //1サンプルあたりのビット数

format.nSamplesPerSec = samplingrate;     //サンプリングレート

format.nBlockAlign = samplesize;

format.nAvgBytesPerSec = transferbytes;
```

それで…鳴らすッ…‼

さぁ、やってみよう。やれましたか？

さてここまでやっておいて何なのですが、ホントにサウンドを自前でやろうとするとファイルフォーマットやらチャンクやらスレッドやらをいじくりまわさなければならないので、ライブラリに頼ったほうが良いと思います。

9.1.4　CRI ADX2 LE について

…様々なフォーマットの様々なデータを扱おうとするとちょっと大変なんすわ。とりあえずは「仕組み」を知ってもらうために授業してみたってのが正直なところです(ホントは自前でOggVorbisとかをライブラリリンクして、その波形データを流し込んで再生とかやってみたかったけど…う～ん)

ともかくCRI ADX2 LEに関しては

http://www.slideshare.net/takaakiichijo/adx2-le-38962377

ここを見ると良いと思います。

こういう本もありますが…

プロ用に誘導している本なので、ぶっちゃけあまりオススメしません。ちょっと高い割にプログラムの書籍ではないですしね。(音楽に興味がある、音作りに興味がある人は買ってもいいと思います。)

まぁ、ひとまずはダウンロードしましょう。

http://www.adx2le.com/download/index.html

で、よくライセンスのところを読んでおきましょう。

ライセンス更新について

「ADX2 LE」のライセンスには下記の 2 種類があります。

- 短期ライセンス（初期ライセンス）
  ダウンロードから約 1 ヵ月使用可能なライセンスです。
  「ADX2 LE」SDK をダウンロードすると最初はこのライセンスが同梱されていいます。
- 長期ライセンス
  1 年間使用可能な長期ライセンスです。
  長期ライセンスの入手・更新には、申請手続き（無償）が必要です。
  長期ライセンスにつきましては[こちら]

こういうライセンスが書いてます。長期のやつは色々と面倒なので短期のやつを落としてください。

もちろん Windows 用を落としてね…？

マニュアルはここです

http://www.criware.jp/adx2le/docs/windows/index_man.html

よ～～～く読んでおきましょう。変なところで嵌らないようにね？

実際のゲームプログラムへの組み込みはここを読んでください。

http://www.criware.jp/adx2le/docs/windows/index_man.html

サーバにもライブラリ本体をアップロードしておきました。

gakuseigamero¥library¥CRI の中のファイルを落として解凍してください。

次にライブラリパスを通します。

プログラマならこれくらい知っとく必要がある。プログラムを組むだけがプログラマの仕事ではない…ああ、面倒くさい。

解凍フォルダの PC って書いてあるフォルダにパスを通す。中に include と Lib があることを確認してください。

パスの通し方は「コンピュータ」で右クリック→プロパティ→システムの詳細設定→環境変数→ユーザ環境変数の新規

変数名を CRI_LIB とでもしておきます。変数値に先ほどのフォルダ名を指定します。

確認のため、コマンドプロンプトで

echo %CRI_LIB%

と打ち込んで、パスが出るようなら成功です。

ひとまずこのパスをプロジェクトに組み込みます。

追加のインクルードパスに$(CRI_LIB)¥Include と指定し、$(CRI_LIB)¥Libs¥x86 と指定してください。(この環境変数操作の前から VS を立ち上げていたのなら一度 VS を落としてから開き直しましょう)

さらに「追加の依存ファイル」に

cri_ware_pcx86_LE_import.lib

も追加しておいてください。

これで準備第一段階は終了です。

ひとまずは実行して、何も変化無しということを確認しておいてください。

サンプルフレームワークを使っても良いんですけど、余計な機能(ウィンドウ出すとか)が付いている上に、内容がそれなりに難しい(オブジェクト指向分かってないとお手上げ)ので、必要な分だけ使用したほうが良いでしょう(若干書き方は古い部分はあるけど、読むのは勉強になると思います)。

というわけで、ちまちまとやっていきますが、そのためにはここを今一度読んでください。

http://www.criware.jp/adx2le/docs/windows/index_man.html

さて、最初は CRI プロジェクト自体はサンプルのものを使用させてもらいましょう。

samples¥criatom¥data¥Public

っての中を見ると ADX2 だの Basic だのがたくさんあると思います。これを自分のプログラムの直下に Sound ってフォルダを作って、そこにまるっとコピーしてください。

さて、いよいよ CRI ライブラリのプログラミングです。

ひとまず XAudio2 のコードは邪魔になりましたので、コメントアウトするなり消すなりしておきましょう。

ひとまずマニュアルの通りに進めましょう。でもメイン関数の中に書くとゴチャゴチャになるので、いったん SoundManager クラスを作りましょう。いつもの要領で作ってください！

SoundManager.cpp のトップで

#include<cri_adx2le.h>

と書いてください。

あくまでも SoundManager は CRI の処理をメインに置きたくないがためのクラスです。大したテクニックは使いません。とにかく Init()関数と Load()関数と Play()関数と Terminate()関数があればいいのでとりあえず、引数、戻り値無しで関数を用意してください。あ、あとは Update()関数もな？

よ～し、関数を定義した前提でコーディングしていくよー。こっからは「なんで」って考えても不毛なんでわりかしそのままやってください。

ひとまず中で共通で使用する変数を定義します。

```
CriAtomExAcbHn acbHandle;          //ACBハンドル
CriAtomExVoicePoolHn voicePool;    //ボイスプールハンドル
CriAtomDbasId dbas;                //D-BASハンドル
CriAtomExPlayerHn player;          //プレーヤハンドル
```

これ、関数の外で宣言してください。プレーヤはもちろん遊ぶ人って意味ではなく、CD プレーヤとかのプレーヤです。

更に、様々なコールバック関数を定義する必要があります。

まずはエラー時のコールバック…

```
static void CriErrorCallbackFunction(const CriChar8 *errid, CriUint32 p1, CriUint32 p2,
CriUint32 *parray)
{
    const CriChar8 *errmsg;
    //デバッグウィンドウにエラー出力
    errmsg = criErr_ConvertIdToMessage(errid, p1, p2);
    OutputDebugStringA(errmsg);
}
```

次にメモリ確保時のコールバック…

```
void* CriAllocatorFunction(void *obj, CriUint32 size)
{
    void *ptr = malloc(size);
    return ptr;
}
```

メモリ解放時のコールバック

```
void CriFreeFunc(void *obj, void *ptr)
{
    free(ptr);
}
```

で、Init 関数で初期化してください！

```
void
SoundManager::Init(){
        //エラー時のコールバック関数登録
        criErr_SetCallback(CriErrorCallbackFunction);

        //メモリアロケータの登録
        criAtomEx_SetUserAllocator(CriAllocatorFunction, CriFreeFunc, nullptr);

        //ライブラリ初期化
```

```
        criAtomEx_Initialize_PC(nullptr, nullptr, 0);

        //ストリーミング用バッファの作成
        dbas = criAtomDbas_Create(nullptr, nullptr, 0);

        //全体設定ファイルの登録
        criAtomEx_RegisterAcfFile(nullptr, acfFilePath, nullptr, 0);

        //DSPバス設定の登録
        criAtomEx_AttachDspBusSetting("DspBusSetting_0", nullptr, 0);

        //ボイスプールの作成
        CriAtomExStandardVoicePoolConfig vpconfig;
        criAtomExVoicePool_SetDefaultConfigForStandardVoicePool(&vpconfig);
        vpconfig.player_config.streaming_flag = CRI_TRUE;
        voicePool = criAtomExVoicePool_AllocateStandardVoicePool(&vpconfig, nullptr, 0);

        //ACBファイルのロード
        acbHandle = criAtomExAcb_LoadAcbFile(nullptr, acbFilePath, nullptr,awbFilePath,
nullptr,0);
        //プレーヤの作成
        player = criAtomExPlayer_Create(nullptr, nullptr, 0);

        //とりあえずBGM鳴らしましょう→あとでこの処理は別の場所に移します。
        criAtomExPlayer_SetCueId(player,acbHandle,0);
        criAtomExPlayer_Start(player);

}
```

次に Update 関数

```
void
SoundManager::Update(){
        criAtomEx_ExecuteMain();
}
```

最後に Terminate 関数

```
void
```

```
SoundManager::Terminate(){
	//後始末
	criAtomExPlayer_Destroy(player);
	criAtomExAcb_Release(acbHandle);

	criAtomExVoicePool_Free(voicePool);
	criAtomEx_UnregisterAcf();
	criAtomEx_Finalize_PC();

}
```

とりあえず説明書に書いてるとおりに、初期化→アップデート→後始末の流れを作りましょう。

初期化はメインループ直前で、後始末はメインループを抜けた後。メインループでは毎回 Update 関数を呼ぶようにしましょう。

とりあえず Init で BGM 鳴らすところまで記述しているので、Main 関数に呼び出しをきっちり記載すれば BGM がなると思います。ほぼほぼまんまソースコード載せたので「鳴らない」って人は「サウンドデータ」を Sound フォルダの下に置くことに失敗しているのではないでしょうか。

さて、後は鳴らしてみましょう。

Play 関数を引数アリにします。

```
void
SoundManager::Play(int cueId){
	criAtomExPlayer_SetCueId(player, acbHandle, cueId);//キューを選択
	criAtomExPlayer_Start(player);//現在のキューを再生
}
```

さて、ここまで書けたら宿題です。

スペースキーを押したら銃の音がなるようにしてみてください。

銃の音が何なのかってのは実は Basic.h に書いてあって

/* Cue List (Cue ID) */

#define CRI_BASIC_MUSIC1 (0) /* Crossfade and Ducking */

#define CRI_BASIC_MUSIC2 (1) /* Crossfade and Ducking */

#define CRI_BASIC_VOICE_RANDOM (2) /* Random Track and Ducking */

#define CRI_BASIC_VOICE_EFFECT (3) /* Bus Effect and Ducking */

#define CRI_BASIC_KALIMBA (4) /* Random Pitch */

#define CRI_BASIC_GUNSHOT (5) /* Timeline */

#define CRI_BASIC_BOMB_LIMIT (6) /* HCA-MX and Cue Limit */

#define CRI_BASIC_PITCH_UP_M2 (7) /* Action */

#define CRI_BASIC_PITCH_DEF_M2 (8) /* Action */

こんな感じで定義されています。

Main 関数で Basic.h をインクルードして

SoundManager::Instance().Play(CRI_BASIC_GUNSHOT);

で BGM 再生中にスペースキーで音がなるようにしてください。

スペースキー検知はどうするんでしたっけ？

GetKeyboardState を使用します。

使い方がわからない？うっそー？前期でやらなかったー？

まぁわからなくてもグーグルセンセーに聞けばわかるので解説しません。とりあえず宿題なのでやってください。

10シューティングゲームつくろー

さてもう新しいところです。

PMDモデルでシューティングゲーム作ります。

戦闘機のモデルは

http://www6.atwiki.jp/vpvpwiki/pages/483.html

このへんからとってくればいいよ。