クソザコナメクジに

は難しい

はい、スプラッターハウス終わったし、いよいよ 3D を作っていきましょう。

内容

概要

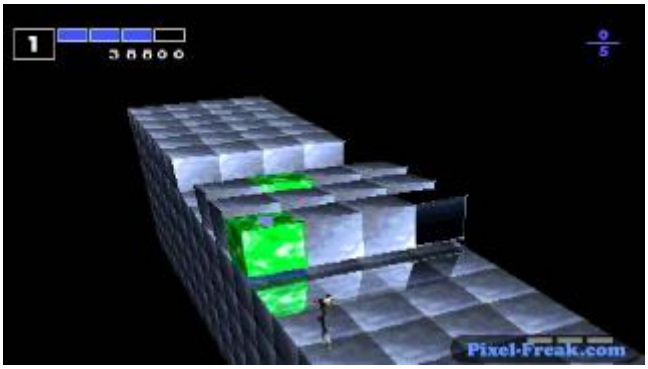
ぼくは正直者だから「簡単」とは言いません。言いませんが、DxLib を使っていきますので、それほど難しくもないと思います。

今回参考にするゲームは IntelligentCube…I.Q です。

知ってる人は少ないんじゃないかな…

動画を見てみましょう。
https://www.youtube.com/watch?v=BZM9kTGFeko
こういうやつです。

PS1 の時代のゲームのくせにしれっと生意気にも反射とか使ってたりしてなかなかの良ゲーなのですよね。



おそらく、作ること自体にそれほど時間はかからないし、スプラッターハウスが無事習得できているみんなにとっては…詰まることも少ないのではないでしょうか？

スケルトン

まずスケルトン的なのをさっさと作りましょう。

```
#include"Game.h"

int main() {
	Game& game = Game::Instance();
	game.Initialize();
	game.Run();
	game.Terminate();
}
```
エントリポイント作りーの

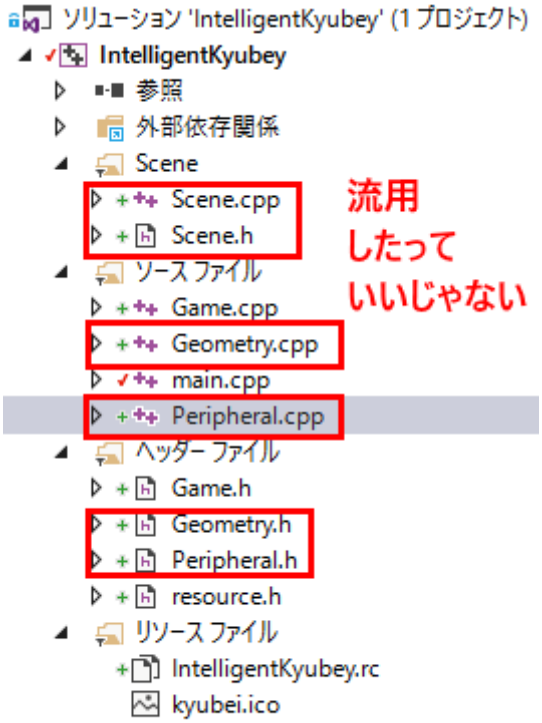
```
class Game {
private:
	Game();
	Game(const Game&);
	void operator=(const Game&);
public:
	~Game();

	//インスタンスを返す
	static Game& Instance() {
		static Game instance;
		return instance;
	}
	//初期化とかやるんやで
	void Initialize();

	//実際のゲームループを動かすんやで？
	void Run();

	//終了時の後処理をするんやで？
	void Terminate();
};
```
Game クラス作りーの

その他一部(Geometry クラス、Scene クラス)はスプラッターハウスから流用しーの



トツギーノ

ひとまずここまでやってみましょう

前のプログラムを一部流用してもいいし、見ながらやってもいいし、流用しなくてもいいし、作り方を変えてもいいので、まずはタイトル画面を出しましょう。

ちなみに今回の画面サイズは 1280,720 とします。

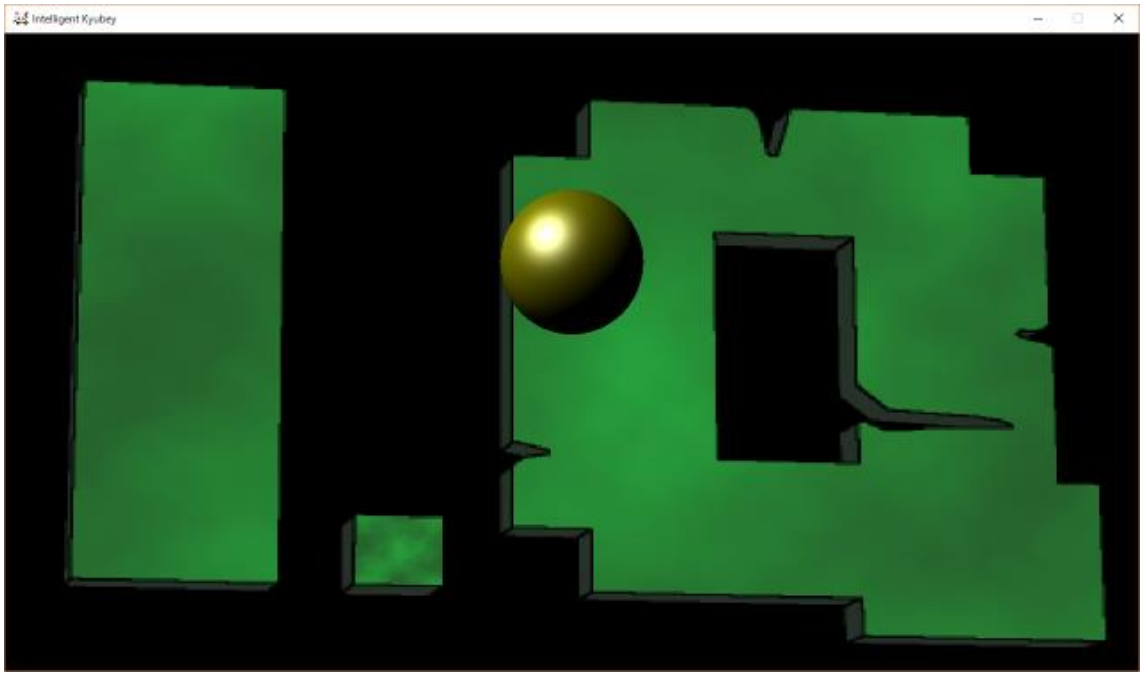
3D 事始め

とにかく表示だ

きんのたま

とりあえず金の玉を表示してみましょう。こんな感じで記述してください。

```
DxLib::DrawExtendGraph(0, 0, wsize.w, wsize.h, _titleH, false);
DxLib::DrawSphere3D(VGet(wsize.w / 2, wsize.h / 2+100, 0), 80.0f, 36, GetColor(128, 128,0), GetColor(255, 255, 255), true);
```



こんな感じになるかと思います。

おおー!!3D だー!!!

って、あんまり感動はしないか。大した話じゃないしね。最後の引数を true->false にすると球体がワイヤーフレーム表示になります。

ちなみに、原因は良く分からないのですが、この球体を画面中央に置くと表示が妙な事になります。

まぁ、恐らくは Z バッファが無効になってるんだけど…とりあえず球体の座標を

wsize.w / 2, wsize.h / 2,0
にしてみてから試すと

謎の現象…

こうなります。

球体の下半分がおかしなことになってるのに気づきますね？

3D というのは厄介な事が多いのです。厄介な物事に対処するには事前にある程度仕組みを知っておく必要があります。

という事で、3D に必須の知識をちょっと触れておきましょう。ほんの初歩をちょっとね？
あっ…(察し)ふーん。

事前知識

3D のプログラミングをするには、例えそれが DxLib を使っていたとしても、言語とは別に特有の基礎知識が必要となります。大変だろう？ワクワクするだろう？基礎知識がないと↑の原因すら全く分からないのだ!!!

基礎知識があってもピンとは来ないかもしれない。だが基礎知識がなければこれに対応するのはほぼ不可能です。

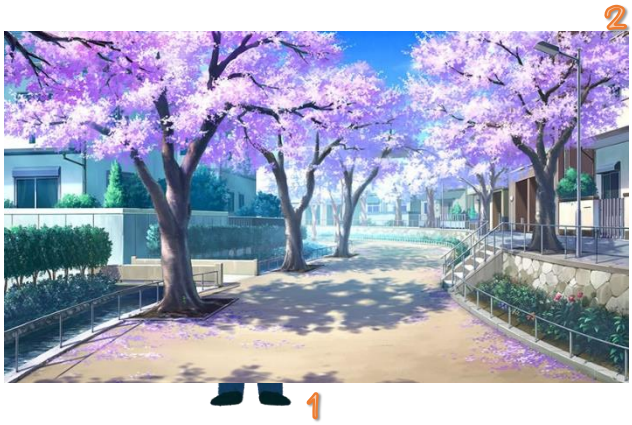
というわけでいくつかの用語を解説しておきます。なお、これは多分後期でも言いますがこのGoogle先生時代においては「用語」がかなりの「力」を持ちます。いわゆる「検索力」につながります。

Google先生のおかげで1から10までを完璧に頭に入れておく必要はなくなりましたが、迅速に答えを得るためには自分の技術に関連する「用語」はしっかり押さえとけって事だよ。

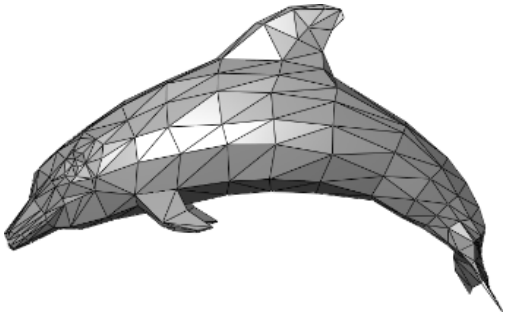
Zバッファ(深度バッファ(デプスバッファ))

ひとまず↑の現象を解消するために必要な概念です。これを知ってないとまずさっきのを治すことはできません。
2Dのゲームであっても「描画順序」が大事な事は分かりますね？
例えば

背景→警官の順番に描画すればこの場合、警官がちゃんと表示されますが、これが逆なら



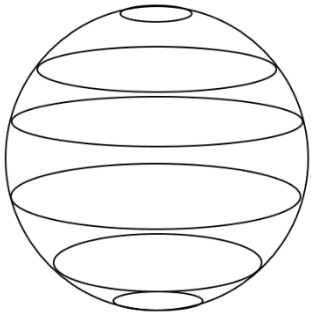
こうなる。これは分かりますよね？
だから描画の順序が大切なんです。で、3Dモデルと言うやつは無数の三角形の集合体でできている事は何となく知っていますね？



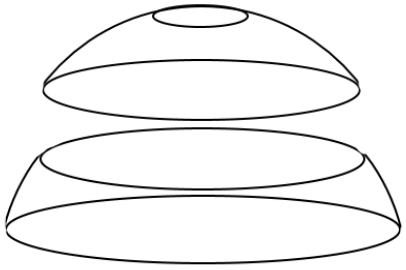
こういうやつね？
で、この三角形も前後の順序を間違えると向こう側が見えちゃったりするわけですよ。イルカくんの胃袋も心臓も肝臓も腸も見えちゃうわけですよ。

で、今回の例ではどうなってるのかと言うと、DrawSphere のポリゴンの描画順は恐らく上から下に(そして奥から手前に)書いているものと思われます。

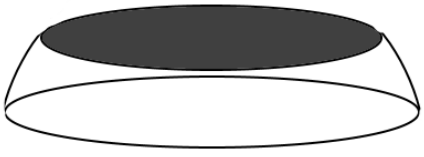
書き方としては、



このような球体を輪切りにしたものを積み重ねて球体にしていると考えられます。



こんな感じに積み重ねているわけですね？
で、上の段を描画した後に下の段を描画したとします。そうすると、下の段の裏側が上の段の表側の後に描画されることになり、先ほどのような変な模様が出るというわけです。

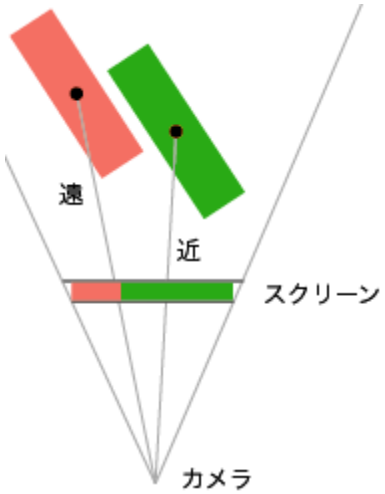


なんとなく原理は分かりましたね？3D というのは面倒な事だらけなのです。

そこで考え出されたのが「Ｚソート法」というやつです。ポリゴンそれぞれのＺ値(視点からの距離)を計算し、それによってポリゴンをソートするというものです。

それで向こう側にあるポリゴンから塗りつぶしを行って最後に一番手前にあるポリゴンを描画することで解決していたのです…

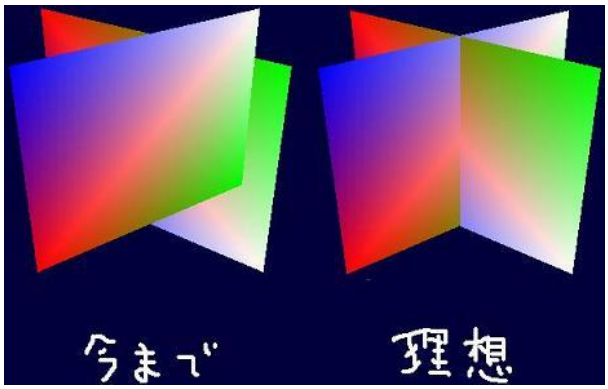
しばらくはその時代が続きました……。ただ、その当時から問題点は指摘されていて…



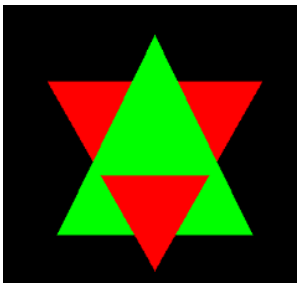
こういう位置関係だったとする。「比較されるポリゴンの座標」っていうのが、ポリゴンの頂点の座標の平均から割り出しているので、上の図のようなおかしなことが発生します。

また、それ以外にも「突き抜け」問題があって…

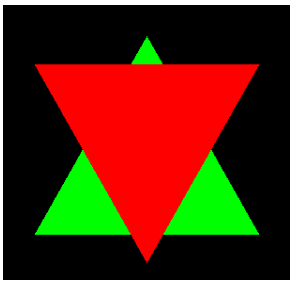
ポリゴン同士が奥行き方向にクロス、もしくは突き抜けていたりするともっとえげつない問題が発生します。



このように見た目の位置関係が全然違って見えるわけです。



理想



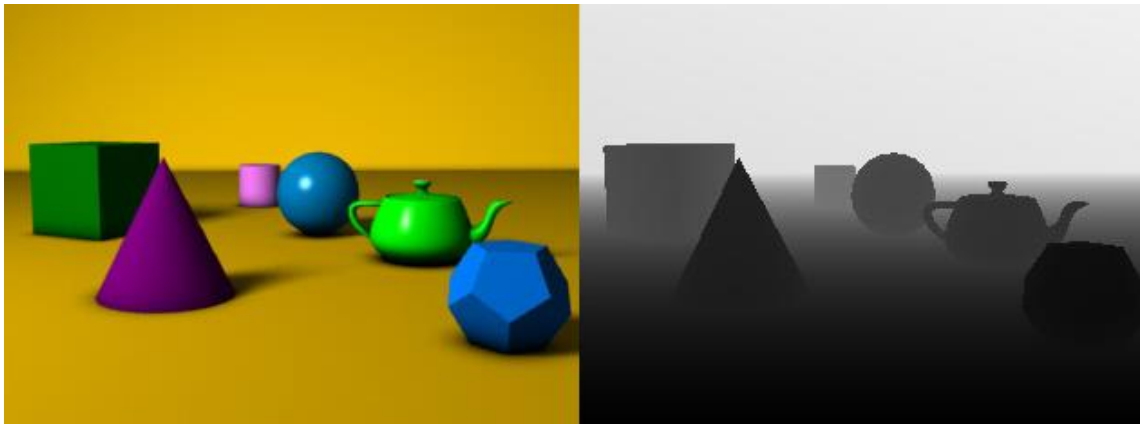
現実

こういう事ですね。

とりあえず、こういう問題がある事は知っておいてくださいね？で、出てきた解決法が

「全ピクセル判定対象にすれば全部解決じゃね？」

という豪快な解決法であり、現在も使用されている解決法です。
やり方は簡単。Zバッファ(深度バッファ)という、「カメラからの距離を記録するバッファ」を用意します。と言っても良く分からないだろうから図で見せると…



左が実際のオブジェクト配置(=そしてレンダリングされるであろう画像)

そして、右が「Z値(深度値)」を画像として表現したものです。正確に言うと後述する「クリッピングボリューム(視錐台)」と非常に密接に関わっているのですが、とりあえず今は

「カメラにクッソ近い所を0。
カメラからクッソ遠い所を1とする」

というルールでZ値が記録されていると思ってください(厳密にいうとこの理解だと不正確です)。基本的にはそれの事を深度値と言い、depthという値で表します。

この値を輝度としてレンダリングしたのが、さっきの画像の右側です。視点から近い方が暗く、遠い方が明るくなっているだろう？

こんなものが何の役に立つのかと言うと、ピクセルごとに「そのピクセルを今塗りつぶすべきか」を判断する手助けになります。

どういう事かと言うと、ピクセルをそのオブジェクトの色で塗りつぶすには、そのピクセルにおいて、そのオブジェクトが一番手前に(一番カメラに近く)ある必要があります。

これに関しては大丈夫ですよね？後ろにある奴の色で塗りつぶされても困りますからね。

で、どうやって行くのかと言うとピクセルごとのZ値を記録していくんですが、先に記録されている側が手前で、今から記録しようとする側が奥にある…つまりZ値が

今から塗りつぶそうとするZ値＞既に塗りつぶされたZ値

であれば、描画もしないし、Z値も更新しないわけです。

今から塗りつぶそうとするZ値＜既に塗りつぶされたZ値

であれば、新しい色に塗りつぶし、Z値も更新します。そうするとどうでしょう。座標的に手前にあるピクセルは描画され、奥にあるのは描画もされず無視されるしZ値も書き込まれない。

結果として、最も手前にあるピクセルだけが残る

という事になります。そういう形で描画順序によらずにオブジェクトを描画する方法がＺバッファ法(深度バッファ法)です。

ちなみにDxLibではこのＺバッファを有効にするフラグとＺバッファに書き込むフラグがデフォルトでOFFになっています。ONにするには

```
DxLib::SetUseZBuffer3D(true);
DxLib::SetWriteZBuffer3D(true);
```

http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N12
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N13

とします。基本的にはこれは両方ともtrueにすべきですが、場合によってはこれをOFFにしたい場合もあります。たぶん２年生の今の知識だけでは分からないと思いますので、現在の所は、両方trueにしておいてください。

カメラ(視点、注視点、視線ベクトル…そして上)

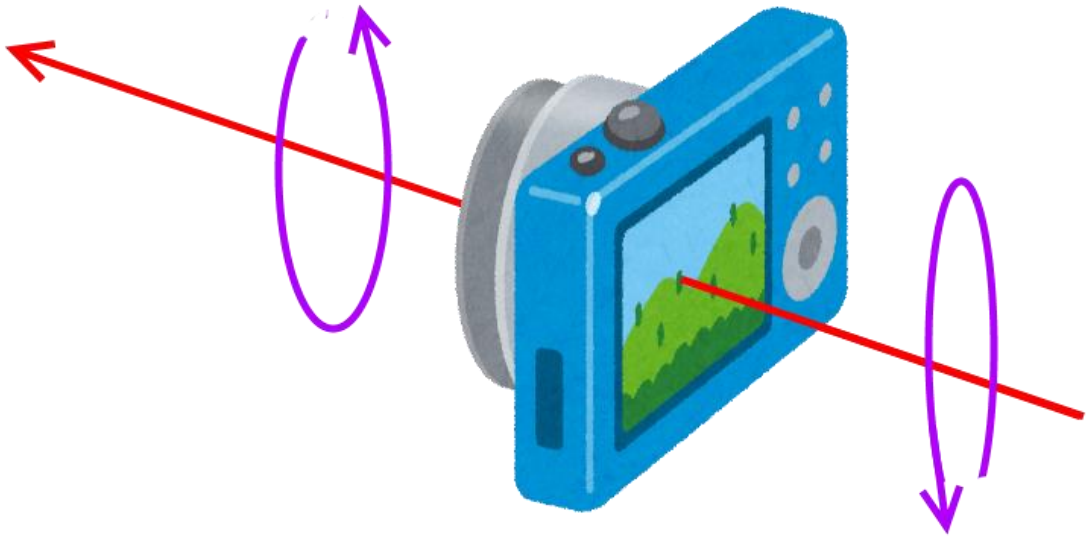
次にカメラの概念を話します。さっきも「カメラからの距離」なんて言い方をしましたね。カメラの座標と言うのは視点の座標であり、3D空間上のどこかに設定します。

ぶっちゃけた話、カメラがないと3Dでは物を3Dっぽく表示できません。ちょっとここで用語を言っておくと

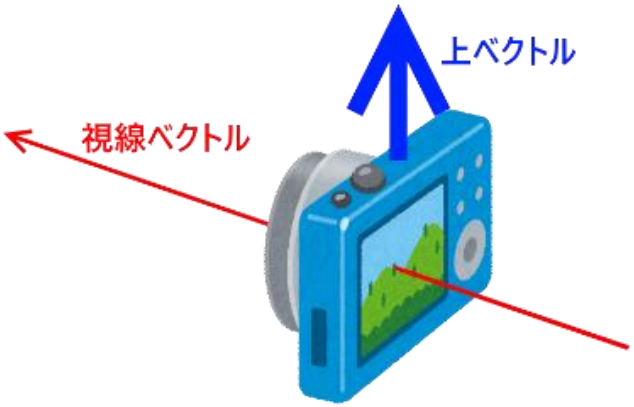
カメラは「視点」と言って、カメラが中心として写す被写体(点)を「注視点」と言います。そして、視点から注視点に向かうベクトルを「視線ベクトル」と言います。

さぁ、この三つの概念があればカメラはOKだと思いますか？思うかもしれません。しれませんがちょっと待ってください。視線ベクトルだけでは不十分です。何故なら軸が一つだからです。3D空間上で軸が一つではいけないのです。

何故なら…



軸が一つでは図のようにどちらが上だか下だか定まらずに、くるんくるんくるんくるん回転してしまうのです。重力があるとこういう事に気づきませんが、宇宙空間に放り出されたら、どちらが上も下もないでしょう？そんな状態になるのです。
これはまずいという事で必要なベクトルがアッパーベクトル…つまり「上」ベクトルです。



これでカメラの向きが初めて固定されます。

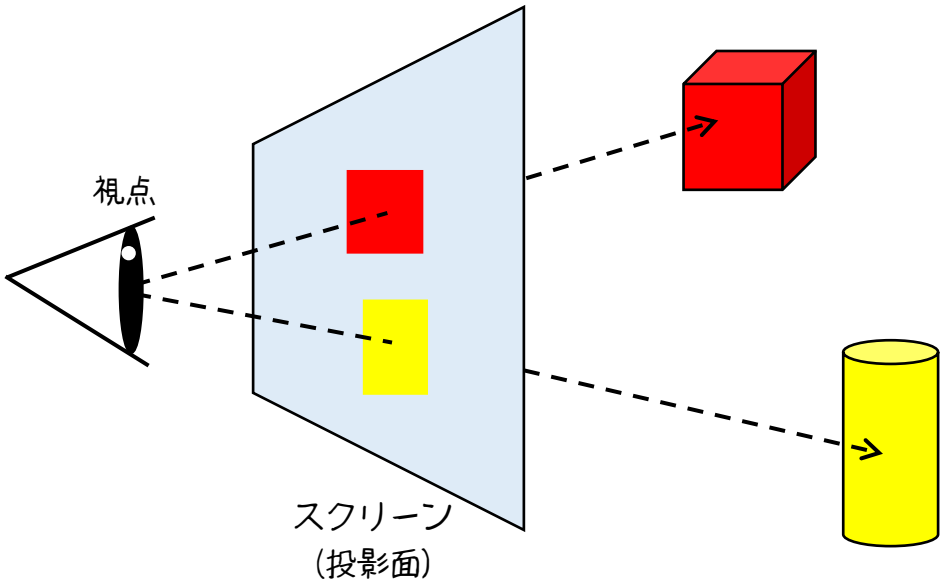
なお、この**視線ベクトルと上ベクトルを外積してさらに外積**して、カメラから見たXYZ座標系と言うのを作って、カメラ座標系と言ったりします。

ややこしいですねぇ…その話はとりあえず置いておいて次行きましょう。

スクリーンと画角と視錐台

ところでここで問題があります。
先ほども述べたようにカメラはあくまでも「視点」であるため、これを定義しただけでは目に映るすべての物は点に集約されてしまいます。これでは何も見えない。

というわけで**視点から適度に離れた場所に「投影」**したものを実際のゲームの画面に表示する必要があるのです。

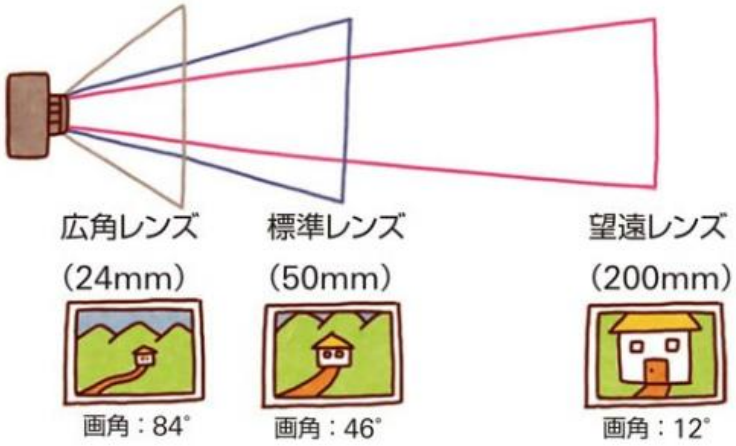


というわけで、視点とスクリーンの位置は一緒ではないのでちょっとややこしい…注意しましょう。

で、上の図で見て分かるように視線は遠くに行けば行くほど広がります。もっと見える範囲が広がっていきます。そして見える範囲が広がるという事はそれをスクリーンの範囲内に収めるためには、視界の中に入る物体の大きさは遠ければ遠いほど小さくなる必要がありますね？

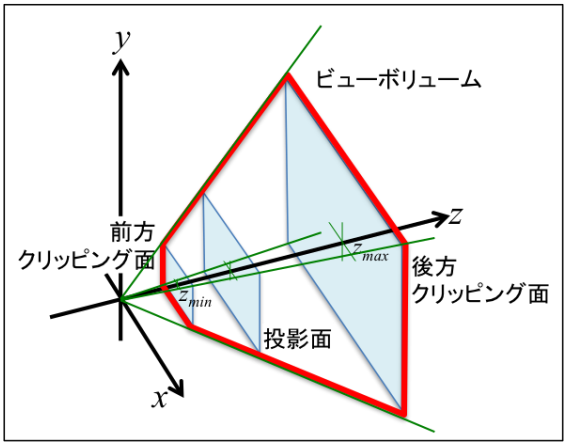
いわゆる遠近法と言うやつです。

カメラは「画角」と言って、見える範囲を広げるか狭めるかを角度で表します。これもまた専門用語かもしれませんね。**画角が狭いと望遠レンズに。画角が広いと広角レンズになります。**



広角は広い範囲を収めるために、遠くのものが極端に小さくなってしまうため見える距離は短くなります。逆に望遠レンズは見える範囲は狭まりますが、遠くまで見えるようになります。

さて、CG の話に戻しますが、いくら望遠レンズといえども永遠の距離まで計算していてはいつまで経っても計算が終わりません。という事で CG では近い方と遠い方にレンダリングする範囲を限定します。もちろん上下左右も限定しますのでレンダリングする範囲は



立体的に図の範囲内に限定されます

この範囲の事を「ビューボリューム」「クリッピングボリューム」「視錐台」と言ったりします。ちなみに前方クリッピング面をニアクリップ。広報クリッピング面をファークリップと言います。near(近い)と far(遠い)ですね。

一応 3D 的な台形の形をしているので『視錐台』と呼んでいます。

とりあえず、初歩の初歩の基本中の基本の知識をお話しいたしました。早速ゲーム作りに励んでいきましょう。

モデルを読み込んで表示

金玉なんて表示しても面白くもなんともないので、せっかくだから可愛いモデルを読み込んで表示したいですね？

とりあえず読み込んで表示

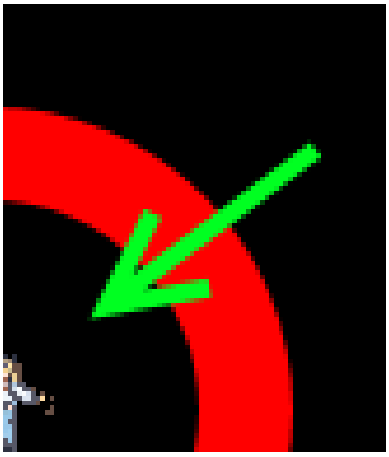
読み込むには MV1LoadModel という関数を使います。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R1N1

そして表示するには MV1DrawModel という関数を使います。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R2N1

これがバグなくできてればこうなります。

あれ？出てねーじゃん？
いやいや…よく見るがいいよ。

左下を見てください。



アッー!!

どうやら表示はされているようだ…
非常に見落としやすいので注意してください。

…そもそもどうしてデフォルトをこんなに使いづらい状況にしているのか、そこに対してはいまだに疑問が残るのだけれども…

ともかく先に原因を言っておこう。カメラだ。

カメラという概念について、一応解説はしたが、DxLib は優しすぎるため、カメラのために特別な設定しなくても 3D の表示はできるようになっています。

今回もそうでしょう？特にカメラの設定はしてませんよね？
ですがご覧のような結果になっており、ぶっちゃけ使い物にはなりません。デフォルト設定なんて所詮そんなもんです。

ちなみに本来であれば画面中央に出るべきものなのですが、何の気を遣いやがったのか左下が(0,0)になるように設定されています。と、いうわけで、3D の一般的なカメラ設定をしましょう。

視点が(0,15,-25)
注視点が(0,10,0)
画角は 60°
ニアとファーは…0.5～300.0 くらいにしましょうか。

カメラその他の設定は
SetCameraPositionAndTarget_UpVecY(VECTOR Position, VECTOR Target) ;
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R12N2

SetupCamera_Perspective(float Fov)
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R12N6

SetCameraNearFar(float Near, float Far) ;
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R12N1

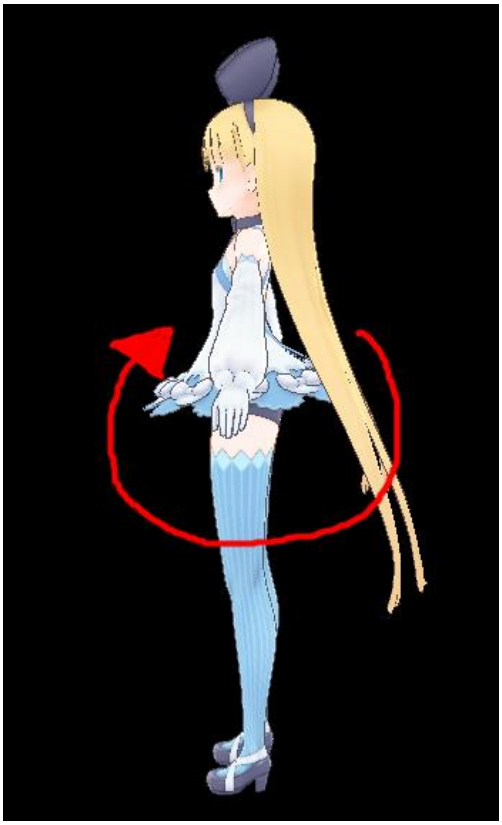
で行えます。うまいこと設定できてれば

モデルが表示されます。

回転します

舐めまわすように眺めるために回転させます。
MV1SetRotationXYZ
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R3N6
この関数は軸と回転量を指定することで回転します。

あ、ちなみに VGet 関数で DxLib::VECTOR 型を作ることができます。ご利用ください。



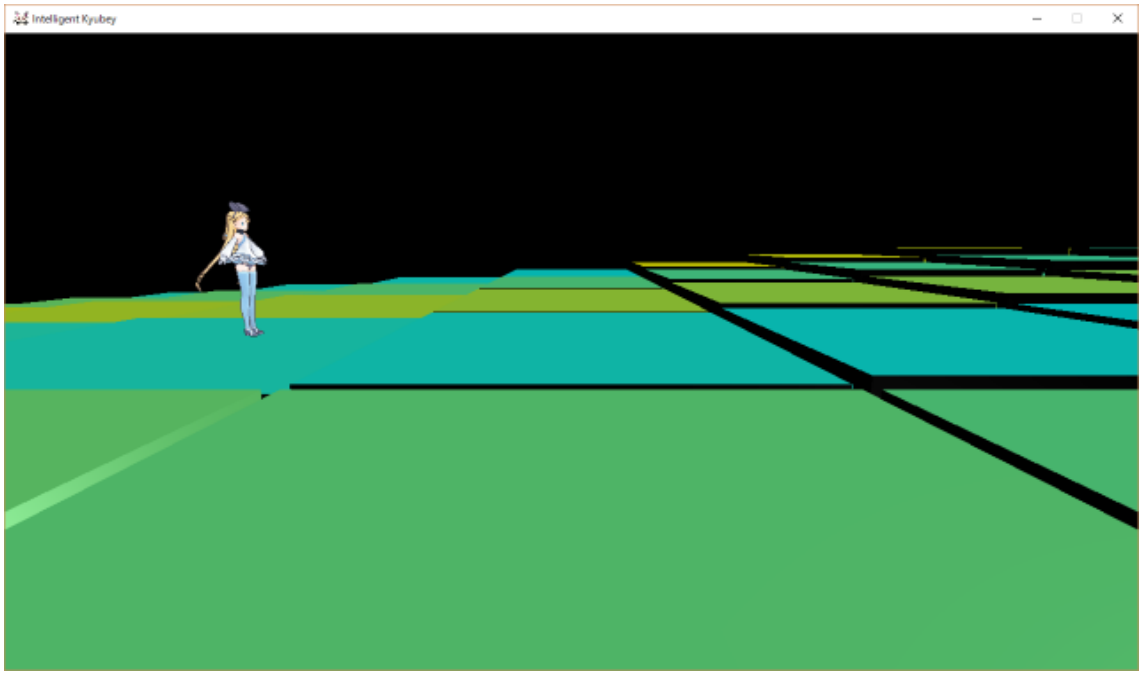
くるくるとね

移動します

パッドの上下左右で(左右奥前)と動くように。更に言うと動いている方向に顔が向くようにして動かしてください。

ちなみに平行移動というか、座標の設定は MV1SetPosition と言う関数を使用します。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R3N2

うまいことやれれば



このように 3D 空間上をあちこち移動できます。

ちなみに地面は DrawCube3D という関数で描画してます。

していますがちょっとだけ…言っておくと、この DrawCube3D は実際のゲームにおいてはほぼ使い物になりません。なぜかというと、**この立方体…回転ができません。**そういう仕様です。たぶんデバッグ用なんじゃないかと思います。

ともかく 3D 空間上でキャラを移動させてください。

アニメーションさせます

いつまでも同じポーズでは面白くありませんね。
アニメーションさせたいと思います。

アニメーションさせるにはまず MV1LoadModel を読み直しましょう？
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R1N1

「MikuMikuDance ではトゥーン用のテクスチャ(toon01.bmp 等)はモデルファイル(pmd or pmx)が存在するテクスチャとは別のフォルダにあっても問題なく読み込むことが出来ますが、ＤＸライブラリではトゥーン用のテクスチャもモデルファイル(pmd or pmx)と同じフォルダに格納しておく必要があります。(トゥーン用のデフォルトテクスチャは MikuMikuDance の Data フォルダの中にあります)

また、ＤＸライブラリでは MMD のモデルファイル形式(pmd or pmx)とモーションファイル形式(vmd)の読み込みに対応していますが、モーションファイル(vmd)はモデルファイル(pmd or pmx)を読み込む際に一緒に読み込まれるようになっています。
ただ、MV1LoadModel にはモーションファイルのファイル名を渡す引数はありませんので、次のようなルールでモデルファイル(pmd or pmx)用のモーションファイルを検索します。

1.モデルファイル名に3桁の番号がついたモーションファイルがあるか検索して、あったら読み込む
(検索する番号は 000 から)

例えば、Miku.pmd(若しくは Miku.pmx) というファイル名を FileName として渡した場合は、
最初に Miku000.vmd というモーションファイルが存在するか調べます。

2.検索する番号を 000 から順に1づつ増やしていき、存在しないファイル名になるまで読み込む

例えば、Miku000.vmd、Miku001.vmd、Miku002.vmd と数字の繋がった3つのモーションファイルが
あった場合は3つとも読み込まれます。
仮に Miku000.vmd,Miku001.vmd,Miku005.vmd のように、番号が途切れていたら、Miku000.vmd と
Miku001.vmd の二つだけ読み込まれ、Miku005.vmd は読み込まれません。

尚、読み込み時にＩＫ計算を行いますので、x ファイルや mv1 ファイルに比べて読み込み時間が非常に長くなっています。

<ループ再生するモーションについて>

モーションの中には歩きや走りといったループさせて再生を行う用途のモーションがあると思います。

そのようなモーションの vmd ファイルは、<読み込みについて>の解説にあったファイル名の付け方にある3桁のモーションの番号の最後に半角の Ｌ をつけてください。』

…ということです。

まとめて言うと

- PMD のモーションファイルは vmd と言う拡張子
- モデルが X.pmd であれば、モーションは X000.vmd～X999.vmd のような名前

- ループモーションは XOOOL.vmd のように最後に L を付ける事

このくらいです。
大したことはないです。

とりあえず今回は
OOOL をニュートラル、001L を歩きモーションとします。

あれ？

ニコニ立体ちゃん…あれ？

非常にかわいいニコニ立体ちゃんなのですが、アニメーションさせようとすると何故かベロが出ます。それはそれでかわいいのですが、ひっこめられないのでちょっと間抜けです。一応 DxLib の注意書きでも

読み込むことのできるモデルファイル形式は x,mqo,mv1,pmd(+vmd),pmx(+vmd) の4種類です。(但し、pmx は pmd 相当の機能だけを使用していた場合のみ正常に読み込める仮対応状態です)

という事は、こういう不具合が起きても仕方ないという事か…。ちなみに僕の一番使いたいモデル「キズナアイ」は、ロードすらできませんでした…OTL。

ちょっとクソムカついたわぁ……それなら、こうだ!!!

そしてこのキャラである

ともかく動かそう

動かすためにはDxLibではひとまず「アタッチ」が必要になります。アタッチとは再生したいアニメーションを指定することです。アニメーションに関しては既にモデルごと読み込んでおりますので、番号を指定するだけです。今回の「歩き」モーションは1番なので 1 番を指定しましょう。

```
_attach=DxLib::MV1AttachAnim(_playermodelH, 1);
```

これだけで、動かないんですが、ポーズが歩きモーション0フレーム目状態になっています。

さて、これを動かしていくためには
MV1SetAttachAnimTime
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R4N3
これの第 3 引数の「アタッチ時間」を少しずつ進めていけば動いてくれます。1は 1 秒…ではなく 1 フレームという意味なのですが、ここがちょっと曲者で、MMD の 1 フレームは 30 分の1秒なんですよね。

通常ゲームは 60fps ですが、MMD は 30fps なので、毎フレーム＋1すると早すぎるのです。なので

```
_frameTime+=0.5f;
```

などのように調整してあげる必要があります。とりあえずこれでアニメーションできていれ

ばオッケーです。

ちなみにループ再生についてですが

モーションにＬをつけたら勝手にループしてくれるとでも思った？
残念!!!こいつは物理演算用でしたーっ!!!

という事で、ループ対応は自前で行わなければなりません。ループ対応…ループ再生とはそもそも何ぞや？

自分でしばらく考えてみよう。

そう、最後まで逝ったら最初に戻ってくるようにすればいいんですね？そして最終フレームと言うのも自動で判断してくれるわけではないっぽいです。じゃあどうするのかというと「総再生時間」を得る関数

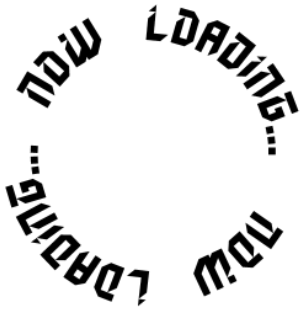
```
float MV1GetAttachAnimTotalTime( int MHandle, int AttachIndex ) ;
```

http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R4N5

を使用して「総再生時間」を取得。これを越えたら_frameTime を 0 に戻すような処理を行えばループになるというわけです。

クッソ重い…

モデル自体は大した重さではないのですが、IK のベイクがそれなりに時間かかってる…ので、読み込みが重いんですよね…。というわけで NOWLOADING を出しましょう。



ロード画面の例

ちなみにこの重さは単純に IK 計算の重さなので、Release にするとちょっとは改善されるとは思います。

また、この NOWLOADING はローディング中はくるくると回るようにしましょう。一応コンシュー

マゲームの制作のチェックリスト項目のひとつに

「処理は１フレたりとも止めてはいけない」

と言うのがあったりします。無茶言うな。流石にこいつは努力目標ではあるんですが、各プラットフォームでほぼ共通の明確なルールがあって

「1秒以上、画面更新が止まってはいけない」

というのがあります。これはゲームセンター時代からそうなのですが、1秒以上画面の更新が行われないと、ハードウェアと言うかドライバと言うか、そういうのが「あっ、これ故障だ」と見なして、シャットダウンしちゃうんですよね。これをウォッチドッグというのですが、面倒な仕様なのです。なので、画面を止めないようにしたいと思います。

非同期読み込み

DxLib ではこれは簡単で…

```
SetUseASyncLoadFlag(true);
```

http://dxlib.o.oo7.jp/function/dxfunc_other.html#R21N1

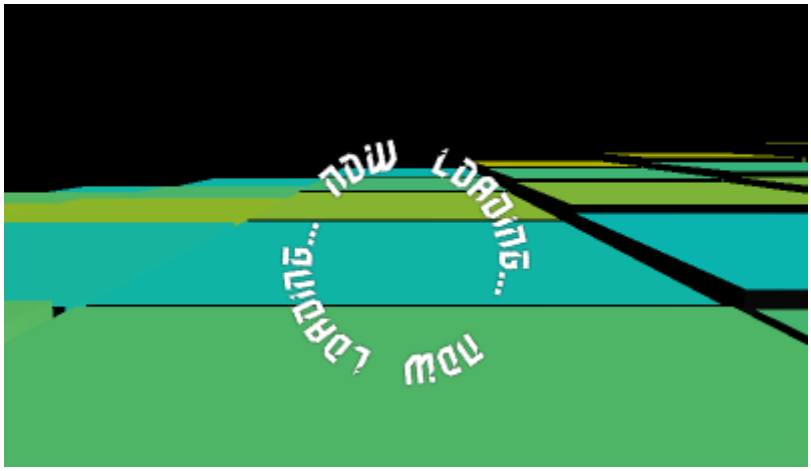
SetUseAsyncLoadFlag を true にすると非同期読み込みになります。3D でも同様です。関数自体は即時復帰ですが、読み込みが終わり次第描画されます。

ただし、読み込み中はアタッチも何も、メッシュに対する操作を受け付けないので、モデルロード処理が終わってからアニメーション適用をしなければならないわけです。

つまり終わるタイミングを知りたい。読み込み終了タイミングを知るには
CheckHandleASyncLoad
http://dxlib.o.oo7.jp/function/dxfunc_other.html#R21N2
を使用するといいでしょう。

引数にハンドルを渡して関数をコールすると、現在のロードスレッドの対象となるモデルが読み込み完了してたら true。まだなら読み続ける…そして読み続けている間は NOWLOADING を表示する。

という具合で良いと思います。うまい事行けば、ロードするまでローディング画面が表示されているという状態になります。



コツと言うか、手順としては、ゲームシーン開始の時点で非同期読み込みを開始して、ローディングアイコンを回転させつつフェードイン。読み込みが終わらなければローディング画面のまま。終われば メインのアップデータ(状態)に切り替えてモデル表示。

この切り替えるタイミングで、アタッチを行えばうまくいくと思います。

ちなみに床は…

Cube 形状を作るのは意外と大変だったりするので、実は床に関しては今の所 DrawCube3D という関数で描画しています。デフォだと

暗いんで、しれっとライティング設定

あまりここでやる気はなかったんですが、ちょっと床の色がアレすぎるので仮にライトの設定をしておきます。
ライトをオンにした状態で…

```
SetUseLighting(true);
SetLightEnable(true);
```

ライトの方向及び属性を設定

```
SetLightDirection(VGet(1.0f, 1.0f, 1.0f));
SetLightDifColor(GetColorF(0.8f, 0.8f, 0.8f, 1.0f));
SetLightAmbColor(GetColorF(0.4f, 0.4f, 0.4f, 1.0f));
SetLightSpcColor(GetColorF(1.0f, 1.0f, 1.0f, 1.0f));
```

ちょっと詳しくは書かないが、平行光線ベクトルと、ディフューズ、アンビエント、スペキュラーを設定している。実はアンビエントは「マテリアル」を設定しないとほぼ意味がないのだが、それはもう少し後で設定することにしよう

Cubeオブジェクトを作る

現在表示しているCubeはDrawCube3Dで描画しているのですが、こいつは回転もできないしIQに使うには力不足である。

で、外部から立方体を読み込んでもいいのだが色分けとかはプログラマブルにできた方がいいと思いますし、頂点情報を作るのも、まぁいい勉強だと思いますので、三角形の集合体としてCubeクラスを作っていきましょう。

```
#pragma once
class Cube
{
public:
	Cube();
	~Cube();
	void Update();
	void Draw();
};
```

まずは三角ポリゴン１枚を表示してみる

現在の所DirectXだろうがOpenGLだろうが3Dの物体は全て三角形の集合体として表現されます。
そういうわけで三角ポリゴンを１枚表示するところからやってみましょう。
当然ながら3つの頂点情報が必要です。描画を行うには
DrawPolygon3D
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N7
という関数を使用します。実はこれはのちにDrawPolygonIndexed3Dにする予定なのですが、ちょっと概念がややこしいので、まずは三角形を表示しましょう。

マニュアルによれば頂点情報はVERTEX3Dという構造体であり

```
// ３Ｄ描画に使用する頂点データ型
struct VERTEX3D
{
    // 座標
    VECTOR pos ;//12bytes
    // 法線
    VECTOR norm ; //12bytes
```

```
    // ディフューズカラー
    COLOR_U8 dif ; //4bytes
    // スペキュラカラー
    COLOR_U8 spc ; //4bytes
    // テクスチャ座標
    float u, v ; //8bytes
    // サブテクスチャ座標
    float su, sv ; //8bytes
} ;
```

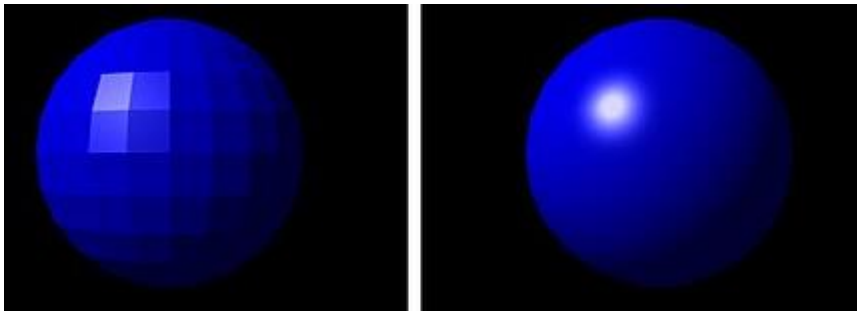
という定義がされています。非常に情報量が多いですね～。なのでぶっちゃけ頂点数は少ない方がいい。1 頂点当たり 48 バイトですね。たいていのフォーマットはこれ以上なのでしゃあないと思いますわ。

用語

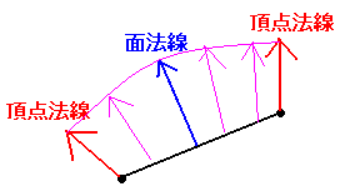
またけったいな用語が出てきたのに気づいておりますでしょうか？
まず法線…ここでは法線ベクトルの事ですが、聞いたことくらいはあると思いますが、

法線ベクトル⇔面に垂直なベクトル

の事です。
あれ？面に垂直なのになんで頂点情報なの？と思った人もいるかもしれませんが、基本的に法線情報ってのは頂点が持っています。本来は「面」であるのは確かに正しいのですが、スムーズシェーディングと言って、面にグラデーションをかけて滑らかにするには頂点にあった方が都合が良いのです。



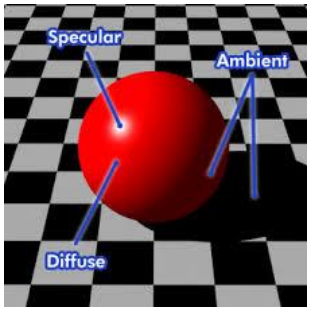
面法線　　頂点法線



ただ、今回のように立方体を作るときはちょっと不向きなのですが…まぁ、全モデルのうち角ばってる方が少ないんじゃないかなと思うので、現状では頂点法線が採用されてると思ってください。

あ、この法線ベクトルが何に使用されるのかと言うと、物体表面の明るさを決めるのに使用されています。おおざっぱに言うと、法線ベクトルと光線ベクトルの**内積**が明るさに対応していると思ってください。なので、何はなくとも内積は復習しておくように!!!

次にディフューズ(拡散反射成分)とスペキュラー(鏡面反射成分)ですが、こいつらは先ほどの明るさにかける係数みたいに思っておいてください。なお、ディフューズはぼやっとした明るさの変化で、スペキュラーはキュンとした明るさの変化に対応します。



なお、↑の図にはスペキュラとディフューズに加えて、アンビエント(環境光)というのがありますが、こいつはスペキュラ&ディフューズのみだと真っ暗(真っ黒)になってしまうので、下駄を履かせる意味でついていると思ってください。

古典的なシェーディングの式ではこのディフューズ、スペキュラー、アンビエントが物体の明るさを決める三大要素になっています。

次の「テクスチャ」はお判りでしょうが、UV って何でしょうね？**UV ってのはテクスチャをポリゴンに張り付けるための目印となる数値です。**頂点と絵のどこを対応させるのかと言うのを 2D 座標で設定する…そういうものです。DxLib の説明だと

「頂点のテクスチャ座標です。画像の左上端を u=0.0f v=0.0f 右下端を u=1.0f v=1.0f とした座標で指定します。」だそうです。

最後のサブテクスチャは、今の所使用しないそうですので、無視してください。

ちなみにテクスチャについて、DxLib における注意点を一つ…
「GrHandle で指定する画像は 8 以上の 2 の n 乗のピクセルサイズ(8,16,32,64,128,256,512,1024 …)である必要があり(使える画像サイズの限界はハードウエアが扱えるサイズの限界

ですので、2048 以上のピクセルサイズは避けた方が良いです)」
だそうです…マジか。3D めんどくせーな。

頂点情報を設定する

で、頂点情報を設定しようとして気づいたのですが、うーん。DxLib の「モデル以外の回転」とかって、面倒…というか、CPU 側でやるんだね…GPU 側にやらせたければシェーダ書かなきゃいけないんだね。シェーダは 2 年生にはまだ早い気もするし…うーん。

いやさ、DX9 とかだったらシェーダ使わなくても頂点に対して SetWorldMatrix 的な関数で回転行列投げてやれば GPU 側で回転してたんだけど、DxLib はモデルに対してしかそういう機能を公開していないっぽい…仕方ないなあ…CPU 側でやるか。

どうやら VTransform で各頂点を動かせという事らしい。掲示板における DxLib の管理人のサンプルコードを見ると…

```
// 行列を使ってワールド座標を算出
Vertex[ 0 ].pos = VTransform( VGet( -100.0f,  100.0f, 0.0f ), TransformMatrix ) ;
Vertex[ 1 ].pos = VTransform( VGet(  100.0f,  100.0f, 0.0f ), TransformMatrix ) ;
Vertex[ 2 ].pos = VTransform( VGet( -100.0f, -100.0f, 0.0f ), TransformMatrix ) ;
Vertex[ 3 ].pos = VTransform( VGet(  100.0f, -100.0f, 0.0f ), TransformMatrix ) ;
```

http://dxlib.o.oo7.jp/cgi/patiobbs/patio.cgi?mode=past&no=2749

マジか…クソめんどー。正直嫌すぎるとしか…。

まぁ、文句言っても仕方ないので…やろう!!!

頂点の情報は
頂点座標、法線、ディフューズ色、スペキュラ色、UV、ゴミ
となっておりますので、三角形を作るならば

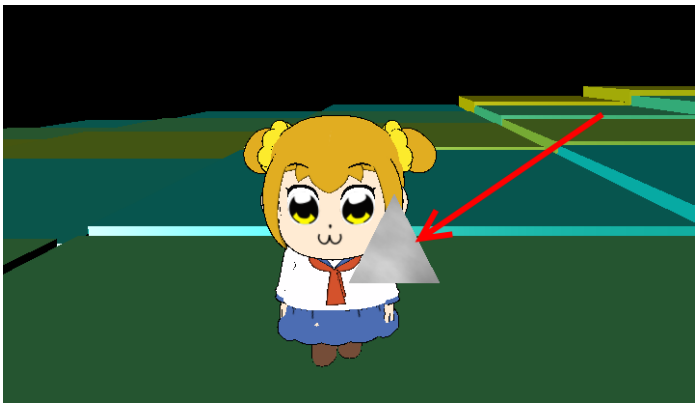
```
VERTEX3D v[3] = {
{ VGet(0,10,0),VGet(0,0,-1),GetColorU8(255,255,255,255),GetColorU8(255,255,255,255),0.0f,0.0f,0.0f,0.0f },
{ VGet(2,10-4,0),VGet(0,0,-1),GetColorU8(255,255,255,255),GetColorU8(255,255,255,255),1.0f,0.0f,0.0f,0.0f },
{ VGet(-2,10-4,0),VGet(0,0,-1),GetColorU8(255,255,255,255),GetColorU8(255,255,255,255),1.0f,1.0f,0.0f,0.0f } };
```

のように3頂点を定義して

```
void
```

```
Cube::Draw() {
        DxLib::DrawPolygon3D(v, 1, gH, false);
}
```

のようにドローします。うまくいけば…



うまいこと三角形が表示されます

三角形を回転しよう

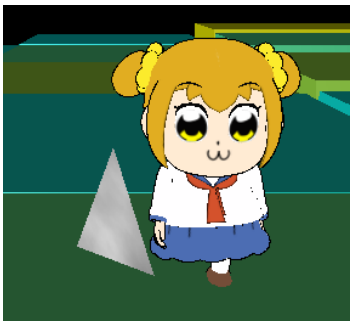
はい、三角形を回転させてみます。Y 軸回転させてみましょう。
ちなみに MV1SetRotationXYZ はモデルに対しての関数なので使えません(MV1 と先頭についている関数はモデルに対しての関数だと思ってください)

それなら、どうやって回転させるのか？
それはですね…すべての頂点の座標を回転させるのです。回転には行列を用いる必要があります。
回転行列を返す関数は色々とあるのですが、今回は Y 軸回転をさせましょう。関数は
MATRIX MGetRotY(float YAxisRotate) ;
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N18
です。
で、MATRIX ってのが行列です。これで取得した MATRIX 型の行列を VTransform で頂点に適用します。
VECTOR VTransform(VECTOR InV, MATRIX InM) ;
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N12
つまり

```
v=VTransform(inv,MGetRotY(angle));
```

これで回転後の頂点座標が得られますので、この回転後の頂点情報を DrawPolygon3D に放り込んであげればいいのです。

そうすれば



このように三角ポリゴンも回転します

平行移動もしてみよう

画面中心から少しずらした状態で平行移動をしてみましょう。回転はそのまま続けつつ。
平行移動の関数は MGetTranslate

さて、平行移動と回転と…2 つ組み合わせるにはどうすればいいのでしょうか？

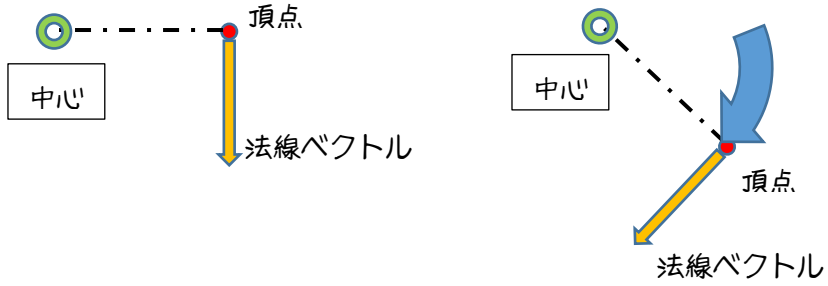
数学の時間にやったはずなのですが…分かりますか？
そう…行列同士を乗算するんですよね。ちなみに DxLib では行列の*オペレータは搭載されておりませんので MMult 関数を使用してください。行列同士の乗算を表します。
MGetTranslate
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N16

MMult
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N25

うまくいけば、中心から離れた場所で回転します。うまくいっている人は試しに MMult 関数の 2つの引数を入れ替えて挙動の違いを確認してください。数学の時間に「行列は乗算の順序を間違えないようにしようね」と言った意味が分かると思います。

法線も回転

実は 3D 的に陰影をつけるのであれば、頂点が回転した時に法線も一緒に回転しないと不自然な事になりますので、法線も回転する必要があります。



ただし、法線には平行移動成分をかけてはいけません。平行移動成分を抜くには 4x4 行列のうち、左上から 3x3 行列だけをかけてやればよく、そのための関数も用意されており

VTransformSR

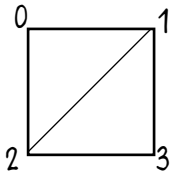
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N13

です。その結果

こんな風に明暗がついていれば OK です

三角形→四角形

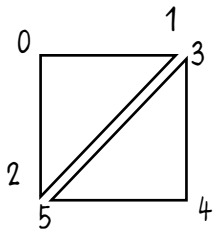
三角形を四角形にします。ただ、ここで DrawIndexedPolygon3D に切り替えます。何故かと言うと本来



このように4頂点で済むはずの情報が実際には6頂点情報となってしまい、データの無駄だからです。こうすると、4 頂点情報＋インデクス情報 6 つで、一見情報が増えているようですが、1頂点当たり 48 バイトなので 6*48>4*48+6*4 となります(288>216)。数が増えれば増える

ほどこれは顕著です。

ちなみにインデックスデータは配列のインデックス(添え字)情報で三角形ができていくよう



に並べていきます。↑の例で言うと

{0,1,2,1,3,2}というわけです。一応のルールとして周り方向(右回り、左回り)は統一しておいた方がいいので、こうしています。
例えば

```
const float ed_w = 5.0f;

VERTEX3D inv[4] = {
        { VGet(-ed_w,ed_w,-ed_w),VGet(0,0,-
1),GetColorU8(255,255,255,255),GetColorU8(255,255,255,255),0.0f,0.0f,0.0f,0.0f },
        { VGet(ed_w,ed_w,-ed_w),VGet(0,0,-
1),GetColorU8(255,255,255,255),GetColorU8(255,255,255,255),1.0f,0.0f,0.0f,0.0f },
        { VGet(-ed_w,-ed_w,-ed_w),VGet(0,0,-
1),GetColorU8(255,255,255,255),GetColorU8(255,255,255,255),0.0f,1.0f,0.0f,0.0f },
        { VGet(ed_w, -ed_w, -ed_w), VGet(0, 0, -1), GetColorU8(255, 255, 255, 255),
GetColorU8(255, 255, 255, 255), 1.0f, 1.0f, 0.0f, 0.0f } };

unsigned short indices[6] = { 0,1,2,1,3,2 };
```

こんな感じに頂点とインデックスを定義しておき

```
DxLib::DrawPolygonIndexed3D(v, 4, indices,2,gH, false);
```

こんな感じで描画すれば…

このように正方形が表示できることでしょう

ここまでができたらより 3D らしく、立方体を作っていきます。

四角形→立方体

さて、ここからは自分で考えて立方体を作ってください。なぁに…立方体は 6 面だから同じものを向きを変えて 6 面つくればいいだけですよ。

- 点の数は 24 個(6x4)
- インデックスの数は 36 個(6x6)

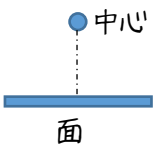
これがヒントです。頑張って作ってみてください。

ひとまずこうなれば…

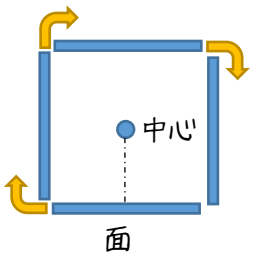
やり方は問いません

一例

さて、やり方の一例ですが、「頂点の回転」を利用します。まずオリジナル頂点を 1 面分だけつくっておいて、それを Y 軸中心回転して 4 面分つくります。このときに最初の 1 面は Y 軸から手前側にずらしておきます。



この状態で回転します。



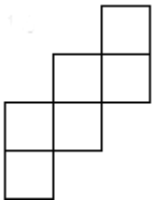

```
//側面4面
for (int j = 0; j < 4; ++j) {
	float angle = (float)j * DX_PI / 2.0f;
	auto m = MGetRotY(angle);
	for (int i = 0; i < vertnum_for_surface; ++i) {
		_vertices[i+ vertnum_for_surface*j] = inv[i];
		_vertices[i + vertnum_for_surface * j.pos = VTransform(inv[i].pos, m);
		_vertices[i + vertnum_for_surface * j].norm = VTransformSR(inv[i].norm,
m);
	}
	for (int i = 0; i < 6; ++i) {
		_indices[i + j * 6] = indices[i] + j * 4;
	}
}
```

こんな風に作ります。後は上蓋と底を作れば OK なんですなあ…

```
float angle =  DX_PI / 2.0f;
auto m = MGetRotY(angle);
//上蓋下蓋
for (int j = 4; j < surface_num; ++j) {
	auto m = MGetRotX(angle);
	for (int i = 0; i < vertnum_for_surface; ++i) {
		_vertices[i + vertnum_for_surface * j] = inv[i];
		_vertices[i + vertnum_for_surface * j].pos = VTransformSR(inv[i].pos,
m);
		_vertices[i + vertnum_for_surface * j].norm = VTransformSR(inv[i].norm,
m);
	}
	for (int i = 0; i < 6; ++i) {
		_indices[i + j * 6] = indices[i] + j * 4;
	}
	angle += DX_PI;
```

```
}
```

学生さんが展開図によっては一気に行ける的なヒントを言ってましたねー。



こういうやつ。↑→↑→↑という具合になっています。という事で X 回転 Y 回転を繰り返せばいいかと思ったらうまくいかない(’•ω•‘)

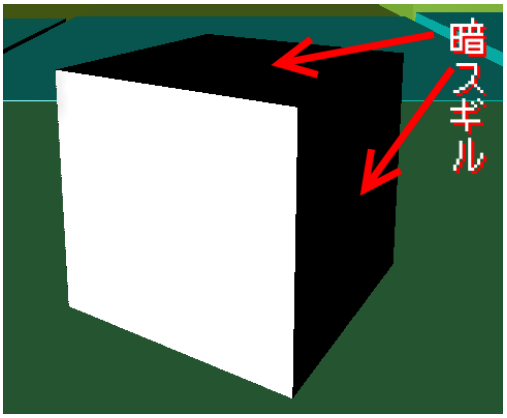
よくよく頭の中で展開するとわかりますが、X 回転 Z 回転 Y 回転 X 回転 Z 回転という流れになっています。

このやり方なら一度で一気に定義できます。

```
//6面設定
MATRIX rots[3] = { MGetRotY(DX_PI_F / 2.0f), MGetRotX(DX_PI_F / 2.0f) ,MGetRotZ(DX_PI_F / 2.0f) };
for (int i = 1; i < 6; ++i) {
	auto& rot = rots[i % 3];
	for (int j = 0; j < _countof(original_verts); ++j) {
		_vertices[i*_countof(original_verts) + j] = original_verts [ j];
		_vertices[i*_countof(original_verts) + j].pos = VTransform(_vertices[(i - 1)*_countof(original_verts) + j].pos,rot);
		_vertices[i*_countof(original_verts) + j].norm = VTransformSR(_vertices[(i - 1)*_countof(original_verts) + j].norm, rot);
	}
	for (int j = 0; j < _countof(original_indices); ++j) {
		_indices[i*_countof(original_indices) + j] = original_indices[j]+4*i;
	}
}
```

アンビエントとマテリアル

こ↑こ↓まで来てみるとある程度 3D の事が分かってきたかなと思いますが、ちょっと気に入らない部分があります。



そう、暗い部分が暗すぎるのです。マックロクロスケなのですわー。
こういう時に明るさの下駄を履かせるのがアンビエント(環境光)成分なのですが、

```
SetLightAmbColor(GetColorF(1.0f, 0.0f, 0.0f, 1.0f));
SetGlobalAmbientLight(GetColorF(1.0f, 1.0f, 0.0f,1.0f));
```

こんな風に書いても真っ暗なんです…何故？何故ッソヨー!!!モルゲッソヨー!!
と思ってリファレンスを見ました
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R13N45
こう書いてます。

「逆に、この値を何にしてもマテリアルのアンビエントカラーが真っ黒だとなにも見た目は変わりません。」

マジすか。初心者向けのような…そうでない…ような？

ということでマテリアルとは何でしょうか？これは日本語で言うと「表面材質」の事です。
「表面の色」「ざらざら」「つるつる」とかそういうやつ。

先ほどライティングの設定をやりましたが、本来、僕らの目に物の色やら物の形やらが見えるのは、僕らの目に「ライト」から放射された「光」が「物体表面」に「反射」して、その「反射した光」が「僕らの目」に入ることによって、例えば「赤い本」等と言うものが認識できるようになっ

ています。

で、古典的なマテリアルは、光の種類それぞれに対応するように設定ができています。ところでまだマテリアルってのを設定してないんですが、通常はモデルそのものに定義されています。

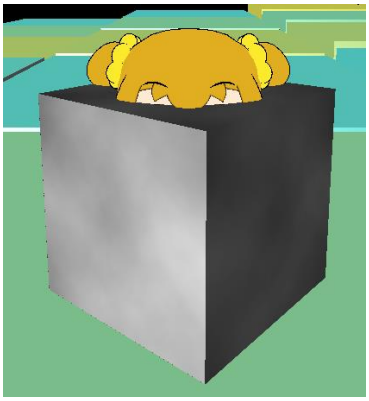
ですが今回はプロシージャルに作った立方体。確かにマテリアルは無いですねぇ…ということで適切な関数はないですかねえ…
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N11
その名も SetMaterialParam ですね。

アンビエントはそれほど強くなくていいと思います。大体スペキュラは真っ白、ディフューズは背景の色に合わせる(今回は背景黒だけど、まぁ、白灰色くらいで)。で、アンビエントはあくまでも「真っ暗にならない」ための措置なので、黒に近い灰色でいいと思います。

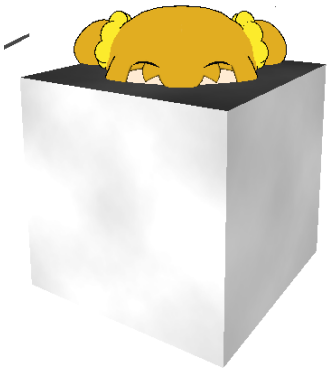
また、Power というのがありますが、これを 0 にしないでください。

```
MATERIALPARAM mp = MATERIALPARAM();
mp.Ambient = GetColorF(0.2f, 0.2f, 0.2f,1.f);
mp.Diffuse = GetColorF(0.75f, 0.75f, 0.75f, 1.f);
mp.Specular = GetColorF(1.f,1.f,1.f,1.f);
mp.Emissive = GetColorF(0.2f, 0.2f, 0.2f, 1.0f);
mp.Power = 10.0f;
```

こういう感じで設定すると



これで、なんとなくアンビエントがついたのが…わかるだろう？あと、さっき Power は 0 に写ちゃダメって言いましたが、何故か分かりますか？何故か？



DrawCube は真っ白に。自分で作った頂点は上部が真っ黒に…

良く分からないかもしれないので、スペキュラの計算方法について説明しよう。スペキュラと言うのは鏡面反射による明るさの決定の事。

通常であればきちんと反射ベクトルを計算するのだ。だが、DxLib では簡単にするために「ハーフベクトル」と言うのを用意します。

ちなみに反射ベクトルの計算方法は覚えてますか？覚えてますよね？

$$R = I - 2N(N \cdot L)$$

って感じで求めてました。覚えてますかぁ？で、スペキュラの明るさって奴は、これと視線のベクトルの内積→$\cos\theta$ を求めて、さらにさらに乗数を乗算してやることによって、スペキュラの明るさを決定しています。

$$\text{明るさ} = k(R \cdot E)^n$$

さて、これの簡易版が、ハーフベクトルによるスペキュラ計算だ。DX9 の時はよく使用されていたようだけど、今って使われてるのかな？

ともかく理屈はこうです。
ハーフベクトルを作る

$$H = (L + E)$$

Hを正規化する
法線ベクトルと内積

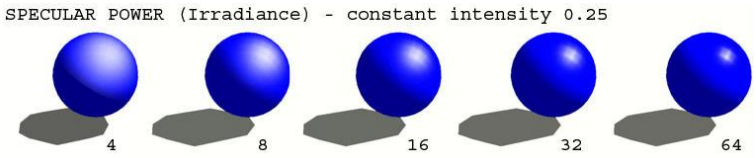
$$H \cdot N$$

合わせて書くとこう

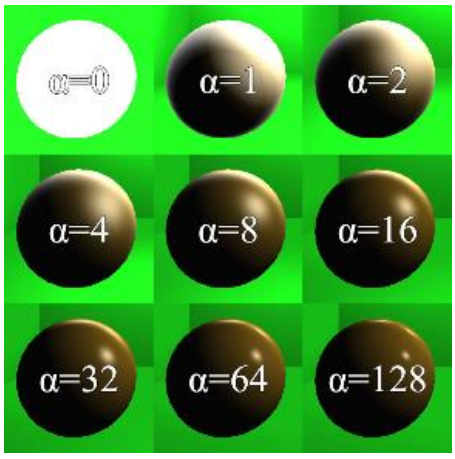
$$(L + E) \cdot N$$

おわり。確かにシンプルですね。ちなみに、ライトベクトルと視線ベクトルからハーフベクトルを作っていますが、理屈はお分かりですか？ベクトルの足し算は平行四辺形状態になるので、足し算結果ベクトルは二つのベクトルの真ん中にあります。どうせ正規化するんだから方向だけ真ん中であればいいという事。

で、恐らくこのハーフベクトルからのスペキュラ計算なんですが、スペキュラと言うのは、このベクトルの内積に対してpowerつまり○○乗したものが明るさとします。普通に考えたらむっちゃ明るくなりそうですよね？でもよく考えてください。正規化済みのベクトルの内積なんて、最大値が1.0fですよ？つまり、明るくなるどころか暗くなります。



ご覧のように値を上げれば上げるほど白い部分が小さくなります。
この値を上げれば上げるほど、金属感が出てきます。
さて、これを0にするとどうなるでしょうか…そう。常に1になります。つまり全てが真っ白になっちゃうんですね。気をつけましょう。



閑話①

コーディングは『アホ』と仕事するつもりで

IT 用語で『フールプルーフ』という言葉を知っているかな？これは『顧客』が『とんでもないアホ』であることを想定して UI を作れよという格言や。

で、これは『同僚』にも言える事でな？ゲーム業界は『チーム作業』と言うやろ？チームにもアホはおるんや…おるんやで…厳選なる選考の結果にも関わらずアホはおるんやで…。

例えば口を酸っぱくして言っているシングルトンのな？

- 『代入禁止』
- 『コピーコンストラクタ禁止』

を無視して、アホは代入するしコピーしようとする。

で、皆でバグる。

さて、この場合悪いのは、アホでしょうか？コピー可能にした設計でしょうか？

もちろん悪いのはアホです。でも 5 人チームならば少なくとも 1 人はほぼ必ず混じっているアホです。

アホは悪い。しかし被害を受けるのは全員です。アホを責めても仕方ありません。じゃあどうすべきかと言うと、『分かってる奴』が可能な限りアホコードを書かせないように配慮すべきなのです。面倒かもしれませんが、その結果、もっと面倒な事になると思っておいてください。

アホな事を許さないためのコーディングとは

- 可能な限り const を使用する
- private を有効活用する。
- 生ポインタを(絶対に)使用しない
- 定期的に簡易コードレビューをする

です。

俺的コーディング規約？

コーディング規約と言うか、何と言うか…俺的なポリシー？っス。

- 状態遷移に switch～case は使わない

- 可能な限り「分岐」を減らす。減らしたいなあ…。
- malloc～free は使わない！
- 配列 new を使わない！
- シングルトン時に「コピー禁止」「代入禁止」を忘れない！
- シングルトンは所詮「グローバル」…それ本当に必要ですか？
- ライブラリ関数、組み込み関数を使用する際は必ずドキュメントを確認
- 「分かってない！部分」は「分かってない！」と潔く認める
- コメントで誤魔化そうとするな
- マジックナンバーがないか帰る前に確認しよう
- ヘッダーで「ヘッダーをインクルード」は極力避ける
- 「警告」を軽んじない！
- メモリ的視点やコンパイラ的視点を身につけよう
- 「型」まわりの理解を深めよう
- コードは短く、シンプルに
- 長いコードは書かず、標準ライブラリを活用しよう
- 「調べる」ための「ひと手間」を惜しまない。自信を持って答えられるようにしよう
- フールプルーフコーディングを心がけよう
- バージョン管理システムを使おう
- コメントは「ヘッダ側」に書こう
- バグったら「きちんと」デバッガを使ってデバッグしよう。
- STL は十分に仕様を把握しておこう
- 自分のコードには責任を持って!!!自分で「何故そう書いたか」を説明して下さい。

という事で、今一度、今一度自分のコードを見直してみよう。

C++コーディングガイドライン

ちなみに C++製作者が書いてる C++CoreGuidelines ってのがあるので、英語だけど一度目を通しておこう

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines

あと、CoreGuidelineChecker なんてのもあって、

https://msdn.microsoft.com/ja-jp/library/mt762841.aspx

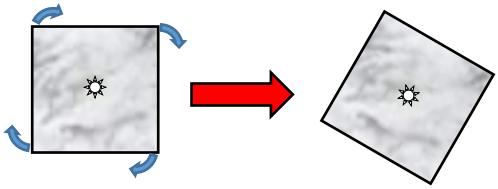
や

https://qiita.com/TomoyukiAota/items/28b39d77646daa74291a

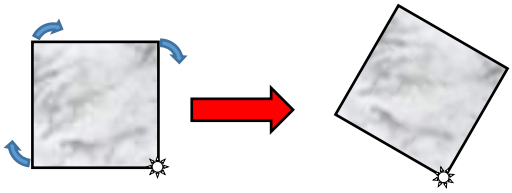
に情報が書いてあります。なお、後者の方は「このツールにはいくつか欠点があります」と明記されてありますので、ちょっと評価を待った方がいいのかもしれません。

転がしてみよう

転がすという事は回転するという事なのですが……今まで学んできた回転とは
こういう風に物体中心の回転でした。



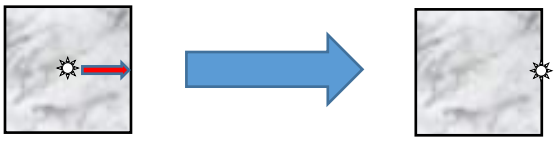
しかし、転がる場合と言うのは立方体を構成する辺のうち、地面に設置している２つの辺のどちらかを中心に回転します。
どちらかというと、転がる側に寄ってる辺を中心に回転します。つまり↑の例だと時計回り方



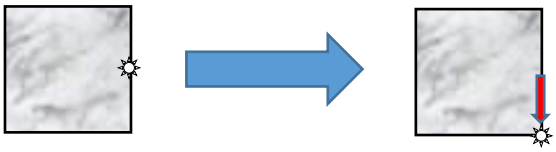
向なら右側。反時計回り方向なら左側へ回転します。
「辺が～」って言うと非常に難しく思えますが「点」と「軸」が分かってれば OK です。もーつ言うと、現在の中心さえ分かれば中心を辺長/2 ぶん移動すれば終わりなので、大した話でもないです。

回転方向はともかく、移動方向は意図的であるため分かっています。この辺の解決アイデアは自分で考えてほしいん(若い頭で…)ですが時間もないし天才頭脳の僕の答えを言います。

中心は転がす前には常にすべての頂点の中間点にあると仮定します。
そこから「進行方向ベクトル」×辺/2 を加算します



次に地面方向に移動させたいので「下ベクトル」×辺/2 を加算



これで求めたい辺と言うか点が出ます。…最終的に最適化はしますが、中間点は全点の平均

にしましょう。

ちなみに「転がる」は rollover ですので

void Rollover(float x,float z)

的な関数を作りましょう。x,z は-1,0,1 しかとらへんで？
もうチョイ細かく言うと
(x,z)=(-1,0),(1,0),(0,-1),(0,1)
にしか動きません。ぶっちゃけ enum Direction でも良かった気もするけどね。

で、一回当たり 90°しか回転しません。これを一回の回転ごとにゆっくりと動かします。
なので
「待機」
「回転開始時」
「回転中」
という3つの状態を遷移すると考えましょう。

ちなみにGeometryクラスのベクタは3Dバージョンを作っておくと楽ができるかも

```
struct Vector3D {
	Vector3D() : x(0), y(0),z(0) {}
	Vector3D(T inx, T iny,T inz) : x(inx), y(iny),z(inz) {}
	T x;
	T y;
	T z;
	void operator+=(const Vector3D<T>& in) {
		x += in.x;
		y += in.y;
		z += in.z;
	}
	void operator*=(float scale) {
		x *= scale;
		y *= scale;
		z *= scale;
	}
	void operator-=(const Vector3D<T>& in) {
```

```
            x -= in.x;
            y -= in.y;
            z -= in.z;
        }

        Vector3D<int> ToIntVec()const {
            Vector3D<int> v(x, y,z);
            return v;
        }
        Vector3D<float> ToFloatVec()const {
            Vector3D<float> v(x, y,z);
            return v;
        }
        float Length()const {
            return sqrtf(x*x+y*y+z*z);
        }
        Vector3D<float> Normalized()const {
            auto len = Length();
            return Vector3f((float)x / len, (float)y / len,(float)z/len);
        }
};
```

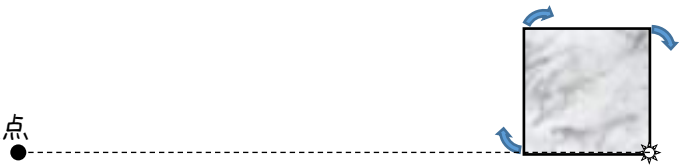
こういうのを作っておいてもいいかな。いや、要らないかも…ぶっちゃけ要らんな!!!

行列を考えよう

今回考慮しなければならない回転はX軸回転とZ軸回転のみ。そしてもちろん回転の前に辺を中心に移動せにゃらなんので、その分の移動行列をTとする。回転行列をそれぞれR_x, R_zとして、Tを元の座標に戻す移動行列をT^{-1}とする。

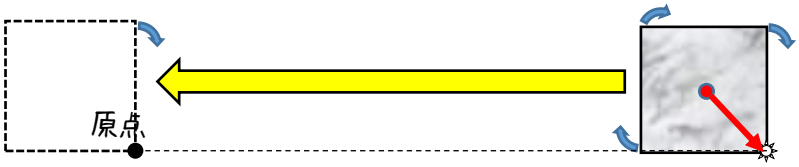
移動行列

まず移動を考えよう…。今回は中心からある程度移動した後の状態からの転がりを考えましょう。



図のように原点からある程度離れた立方体の回転であるならば、その分を考慮して原点に戻さなければなりません。回転後に「中心点再計算」を行っているという事にするのならば

↓の図を見てください。



まず中心を右下へ移動して、さらにそこを原点に持ってきたいわけです。
トータルでどれくらいずらせば(移動させれば)良いでしょうか？

ここまでヒントを言えば分かりますよね？分からんでも試行錯誤しろっ!!
図まで書いてやってんだからさっ!!

いや、もう、ホンマ!?12年生の7月なんだからよ!!!自分で考えようよ!!
とりあえず平行移動の関数と回転の関数教えとくからさ!!!

平行移動
MGetTranslate
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N16

X軸中心回転行列
MGetRotX
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N17

Z軸中心回転行列
MGetRotZ
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N19

行列乗算
MMult
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N25

ベクトルに行列を乗算
VTransform
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N12

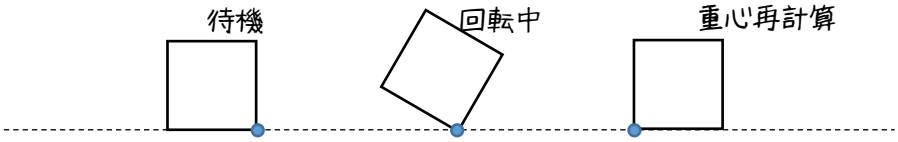
いやホンマ、ホンマはこの関数も自分で調べてほしい。でも何度言っても!!何度関数を確認しろと言っても調べないアホがいるみたいなんでリストにしてあげましたよ!!!ホンマムカつ

くわ!!!なんやねん!!!ホンマ!!!!

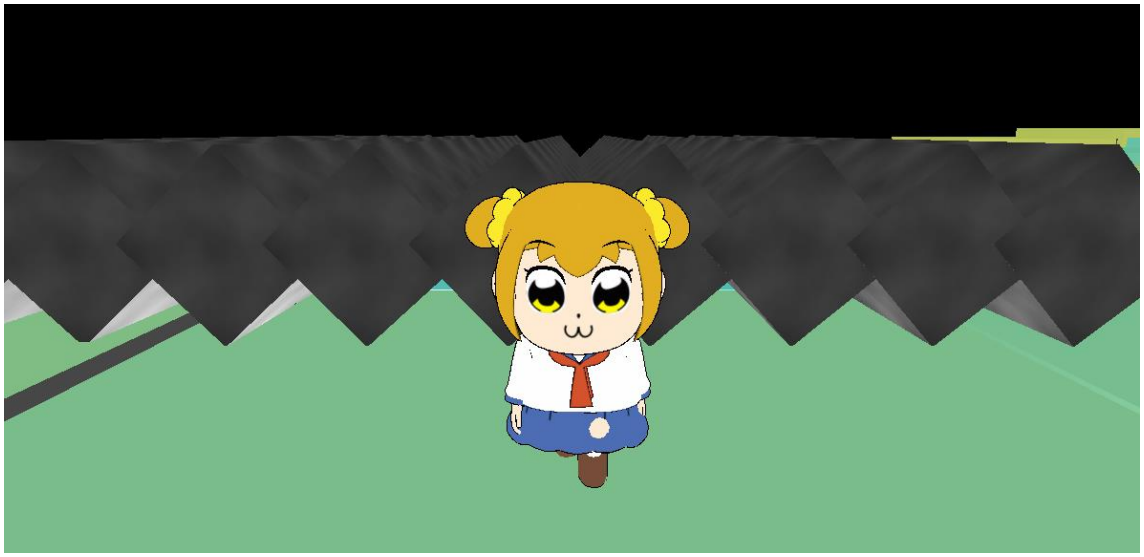
ひとまず、奥側にキューブを転がしていってください。ちなみに僕はCubeの中にいつものように状態を用意して状態遷移をさせています。

```
void WaitUpdate();//待ち状態
void RollingUpdate();//転がり中
void RolledUpdate();//終わった時点で重心再計算
```

この3つの状態を繰り返すわけ。



あ？重心再計算が分からない？
そんなんお前、全部の点(24)の平均を取ればええやんか。え？そんなん重そう？
大丈夫やって

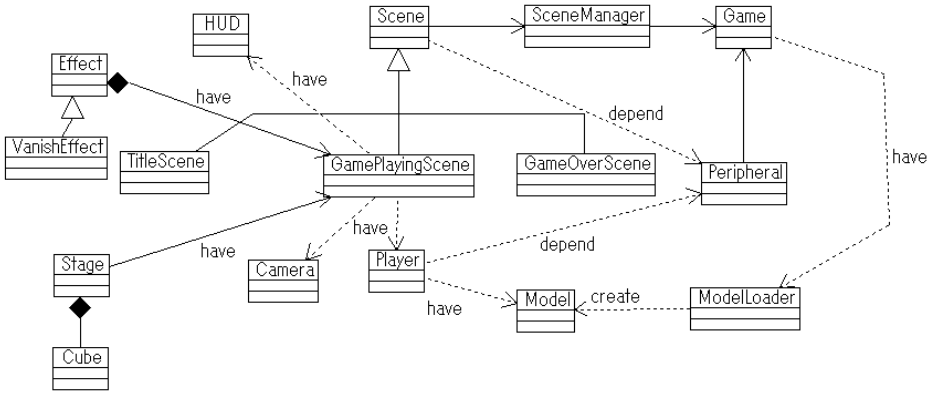


100個Debugモードで動かしても処理落ちせんかったんやから大丈夫やろ。気にすんな。
I.Qの実装だけを考えると前後左右への転がりは考えなくてもいいんやけど、余裕あったらXI(sai)にしようと思ってるしね。

リファクタリングしよう①

さて、ここで一旦リファクタリングしよう。

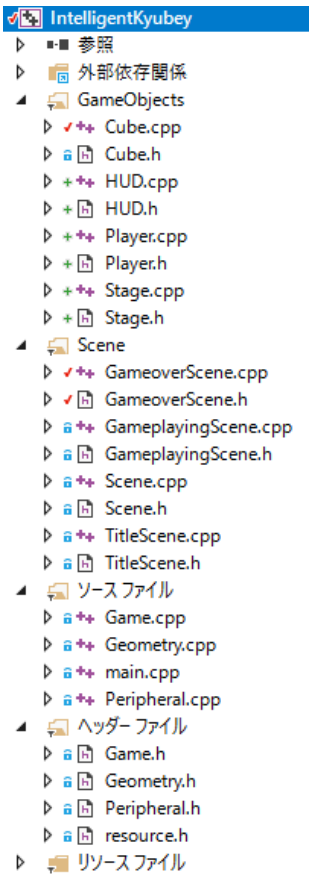
クラス設計



とりあえずこんな感じで(この通りにするとは言ってない…なんとなく流れだけ)。

ちなみに、このツールは前にも紹介したけど UMLMemo ってツールなので、クラスの構成とか考えるのに使ってもらえるといいかなと思います。

そしてプロジェクト構成もフォルダ構成もこんな感じにします。



色々と面倒ですが、必要な事なのでやっておきましょう。プログラム的にもファイル的にも整

理をする癖をつけておきましょう。

モデル読込ストレスを減らすModelLoaderの簡易版を作ろう

今、モデル読み込みめっちゃ重いよね？ストレスだよね？

正直に言うと「ModelLoader」は読み込み時間を減らしません。でも「ストレス」を減らす工夫は作れます。
どういう事かと言うと ModelLoader はシーンに関係なく動作するように作っておきます。Game クラスの持ち物として作ります。

そして、タイトル画面がフェードインしてる間にも裏で読み込みを走らせておきます。メインスレッドは止めないので、基本的には裏でロードしてる事に気づきません。こういう手法は結構使用されます。

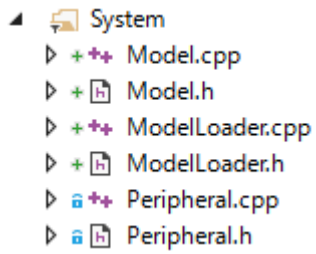
で、ModelLoader なんですが、何をするかと言うと例によってファイル名とハンドルのマップを作ります。二重読み込み防止の意味もあるんですが、ファイル名で指定出来れば裏読みすべきものを事前に登録しておくことが可能になります。

で、とりあえず Game クラスに持たせておくので

```
Game->GetModelLoader()->LoadModel("ファイル名");
```

でモデルを返せるようにします。で、一応モデルクラス Model を返して、モデルハンドルといくつかの変数関数をラップします。

そういう流れで行きましょうか。一応こういうファイル系は File フォルダにまとめておきたいですが、うーん。Peripheral と同じ流れで System ってフォルダを作ってそこに Peripheral と一緒に放り込みましょう。



Model は**ひとまずは**モデルハンドルを返すだけのクラスにしましょう。あんまり一気にやるのはよろしくない。でも余裕がある人はモデルの情報が扱いやすいようなクラスにしてくだ

さい。

ぼくはこういうインターフェイスにしました。

```
class Model
{
	int _handle;
	bool _loadcompleted;
	Position3f _pos;//現在の座標
	Vector3f _vel;//
	float _direction;//現在の向き(X軸を起点としたY軸中心回転角度)
public:
	Model(int h);
	~Model();

	///モデルのハンドルを返す
	///@return ハンドル
	int GetHandle()const;

	///ロード済みである
	///@retval true ロード済み
	///@retval false ロード中
	bool IsLoadCompleted();

	///モデルを描画する
	void Draw();

	///モデルの原点中心Y軸回転する
	///@param yrot 回転角(ラジアン)
	void RotateY(float yrot);

	///特定の座標に移動させる
	///@param pos いるべき座標
	void SetPosition(const Position3f& pos);

	///特定の座標から移動する
```

```
        ///@param vel 今の座標からのオフセット
        void Move(const Vector3f& vel);
};
```

と言った具合ですね。そうすれば呼び出し側(というか Model オブジェクトの持ち主側)のコードは

```
_playermodel->Move(vel);
_playermodel->RotateY(angle);
_playermodel->Draw();
```

と言った具合にシンプルになります。で、このモデル自体も Player クラスの持ち物になるのですが…

入力を受け取り、プレイヤーモデルを挙動定義通りに表示するだけですので大して難しくないと思います。基本的には Update と Draw があれば事足りると思います。

プレイヤークラス

```
#pragma once
#include<memory>///shared_ptr用
class Peripheral;
class Model;

///プレイヤークラス
class Player
{
private:
        std::shared_ptr<Model> _model;
public:
        Player();
        ~Player();
        ///プレイヤーの状態。プレイヤーモデルの状態を更新
        ///@param p 入力情報
        void Update(const Peripheral& p);

        ///プレイヤーモデルの表示
        void Draw();

        ///プレイヤーモデルがロード済みかどうか
        ///@retval true ロード済み
```

```
        ///@retval false ロード中
        bool ModelIsLoaded();
};
```

と言った具合ですね。GamePlayingScene からモデルを直接操作するのではなく、Player に操作させてみましょう。

で、ちょっと土日祝の間にぼっちぼちリファクタリングしてたらですね、ちょっと Model クラスとか Player クラスに変化がありましてね？

```
#include"../Geometry.h"

class Model
{
        int _handle;//モデルのハンドル
        bool _loadcompleted;//モデルのロードが完了している
        Position3f _pos;//現在の座標
        Vector3f _vel;//現在の速度ベクトル
        float _direction;//現在の向き(X軸を起点としたY軸中心回転角度)
        int _currentAttachedNo;//現在のメインのアタッチ番号
        float _totalTime;//現在アタッチ中のアニメーションの総時間
        float _time;//現在のアニメーション時間
public:
        Model(int h);
        ~Model();

        ///モデルのハンドルを返す
        ///@return ハンドル
        int GetHandle()const;

        ///ロード済みである
        ///@retval true ロード済み
        ///@retval false ロード中
        bool IsLoadCompleted();

        ///モデルのフレームを進める
```

```
        void Update();

        ///モデルを描画する
        void Draw();

        ///モデルの原点中心Y軸回転する
        ///@param yrot 回転角(ラジアン)
        void RotateY(float yrot);

        ///特定の座標に移動させる
        ///@param pos いるべき座標
        void SetPosition(const Position3f& pos);

        ///特定の座標から移動する
        ///@param vel 今の座標からのオフセット
        void Move(const Vector3f& vel);

        ///アニメーションをアタッチする
        ///@param animno アニメーション番号
        void AttachAnimaton(const int animno);
};
```

となりーの

```
#pragma once
#include<memory>
class Peripheral;
class Model;
class Player
{
private:
        std::shared_ptr<Model> _model;
public:
        Player();
        ~Player();
        void Update(const Peripheral& p);
        void Draw();
```

```
        bool ModelIsLoaded();
        std::weak_ptr<Model> GetModel() { return _model; }
};
```
と、なりーのトツギーノてな感じで修正しました。なお、GamePlayingScene 側からは Player の生成と Player の Update と Player の Draw しかしてない状態です。

閑話②

ドローコールは極力減らそう(でも極端にはならないように)

DirectX だろうが OpenGL だろうが描画をするうえで使用されるのが描画系の関数です。だいたい Draw なんとかっていう名前になっています。
もちろんこれは DxLib でも同じで MV1DrawModel や DrawPolygonIndexed などが、それにあたります。
CPU⇔GPU に関する命令には色々とありますが、このドロー系の命令を悪戯に増やすと速度低下の要因になります。

ですから、例えば、キューブを描画する際に
```
for(6 面ぶん){
        Draw(1 面)
}
```
などという書き方をするのではなく、6 面分の頂点を事前に用意しておき
```
Draw(6 面分)
```
としたほうが効率が良いのです。こういう大して労力が変わらない部分であれば、極力ドローコール回数が減るようにしてください。

ところでカッコで「極端にならないように」と書きましたが、プログラマと言う人は思い込みが激しくドローコールを減らすために全てを犠牲にしようとする人がいます。何が犠牲になるかと言うと可読性や時間などですね。

ドローコールが増えると確かに描画コストが増えてしまうのですが、そこはゲームプログラミング…「ボトルネック」がそこにあるわけではありません。例えばものごっつい時間をかけてドローコールを減らして頑張ったんだが、プログラムの処理落ちの原因が経路探索にあったならば、それは労力に見合わない頑張りという事になります。

例えば「インスタンシング」を利用するとドローコールを減らすことができますが、それは効率が格段に良くなることが期待されるもしくは、大量のオブジェクト描画がボトルネックになっていると確認できるときにしましょう。

あとさっきの例のようにドローコール減らしが簡単な場合な場合はどしどし減らしましょう。なんでもそうですが極端は良くないです。

また、マテリアルの変更などが行われると一緒にドローができないため、その場合にもMV1DrawModel の中で「隠れドローコール」が増えます。これはプログラムで対処するよりもモデルの作り方を変えたほうが賢明ですね(テクスチャをまとめてしまう…メッシュをまとめてしまう)。

ひとまずボトルネックになりそうってわかっている部分に対してはコメントで「ボトルネックの可能性あり」として一旦放置しよう。後でレベルが上がったら検索して潰していけばいい。

久々の文法①

キャストについて

C++では C 言語のあの、皆さんご存知の言わずもがなのキャストは推奨されていません(とはいえバグを引き起こすという程でもないので僕は正直使ってもいいんじゃないかと思っています。ZeroMemory を蔓延させる方がよっぽど重罪ではないでしょうかね…)

とはいえ、C++で推奨されてるものは使っていくべきだとは思っておりますので、ここでご紹介しておきましょう。

全部で4つあります。static_cast , const_cast , dynamic_cast , reinterpret_cast です。

static_cast

いつものキャスト。普通のキャスト
double⇒int⇒float⇒char⇒float⇒int などの変換に使用されます。

このように基本的なデータ型の変換に使用します。基本的には「暗黙の型変換」が行われる系の物を明示的に変換するのに使用します。また、void*⇒型* も static_cast でオッケーのようです。最も使用されるキャストですね

const_cast

これな、それな、あれな。const_cast な？これはさ、既に const がついてる奴から const 属性を消すというちょっと「ええんかそれ？」なキャストです。いわゆる「const はがし」に使用します。普通は使う事がほとんどないと思います。
というか、これを使用する場面は設計とか使用法とかそういう部分が間違ってないか今一度確かめてください。そういう意味で使用する場面が少ないキャストだと思います。

まぁ…基本使うな!!!

そして忘れろ…どうしようもない時に思い出せ

dynamic_cast

これは「ポリモーフィズム」とともに使用するキャストやね。例えば Dog クラスと Cat クラスがあって、そいつらの親クラスが Animal だとするやん？例えば Dog や Cat を Animal として使用するぶんには問題あらへん。

```
Animal* anim1=new Dog();
Animal* anim2=new Cat();
```

この場合はもはやキャストも必要ないんやで？ただ、問題は逆やで…。こうやって Animal にしてしまった型を再び Dog として使いたい場合…そういう場合はごくたまにあるんや…そんなにないけどな。その場合 Dog やと思ってた中身が Cat やったりして、それに気づかずにやっちまうとメモリぶっ壊したりすんねん。

やから基本的に anim1⇒Dog のキャストは**許されません**。ちなみに Animal にするほうをアップキャスト、Dog にする方をダウンキャストと言います。

で、さっきも言うたけど、ごくごくたまに Animal から Dog にしたい場合がある。まぁ、Virtual で Dog 側の関数が呼べる以上は普通に考えて元の型は知っとると思って良い。ぶっちゃけそれに名前がついてて RTTI(RunTime Type Identification)って言うんやけど、これは忘れてｲﾙﾖ。

とにかくキャストしようと思ったらキャストできるけど、さっき言ったような危険性がある。もしやらかした場合は直ちにクラッシュさせたい…そういう場合に使用するのが dynamic_cast なんや。

もし元の型が Cat だった場合、このキャスト結果は nullptr になるんや

ただ…僕に言わせれば dynamic_cast が必要な場合…それは設計が間違っていると思ってしまっていいと思います。

reinterpret_cast

絶対使うな!!!以上!!!

一応説明しておくと「何でもオッケー」なやつ。やめとけ!!!忘れろ!!!!この言葉を思い出したら電撃が走るようにしとけ

ゲームルール部分を作っていこう

さて、キューブを転がすだけではゲームになりません。「キューブが時間で自動で転がり、Z キーかなんかでプレイヤーが地面にマーキングして、再び Z キーか何かでマーキング解除(この時、上にキューブがあればそれを消す)。」などのルールを実装していきます。

で、ゲームの設計に関わるヒントの話ですが、こういうゲームのルール(自分がゲームにさせたいこと)を**まずは文章化**してください。
頭の中にある「ぼくがかんがえたさいきょうのげーむ」の時点ではほぼ完成不可能だし何から手を付けていいか分かりません。

アイデアの外部化(要は文章化することで他人の目に見える形にする)をすることで、少しは客観的に見ることができます。この時点で矛盾が見つかったり、もしくはゲームの概要がよりはっきりしてくるため、これを面倒がらずにやってください。

また、これを誰かほかの人に見せてください。この場合に「説明が伝わりづらいな」と思ったら図や絵を用いてください。その過程で、自分の中のアイデアもよりしっかりしていきます。
勘違いクリエータにありがちなのはアイデアを自分の頭の中のみに置いておいて全然実現させず、後に誰かが作った時に「俺なんてずっと前に考えてたよ」とか言うわけです。
惨めですね。ある意味その証拠を残すためにも外部化していってください。

で、文章にしたら箇条書きにまとめていってください(アイデアを細分化してください)

ルール

さて、今回の IQ の仕様に関してですが、仕様を箇条書きにすると

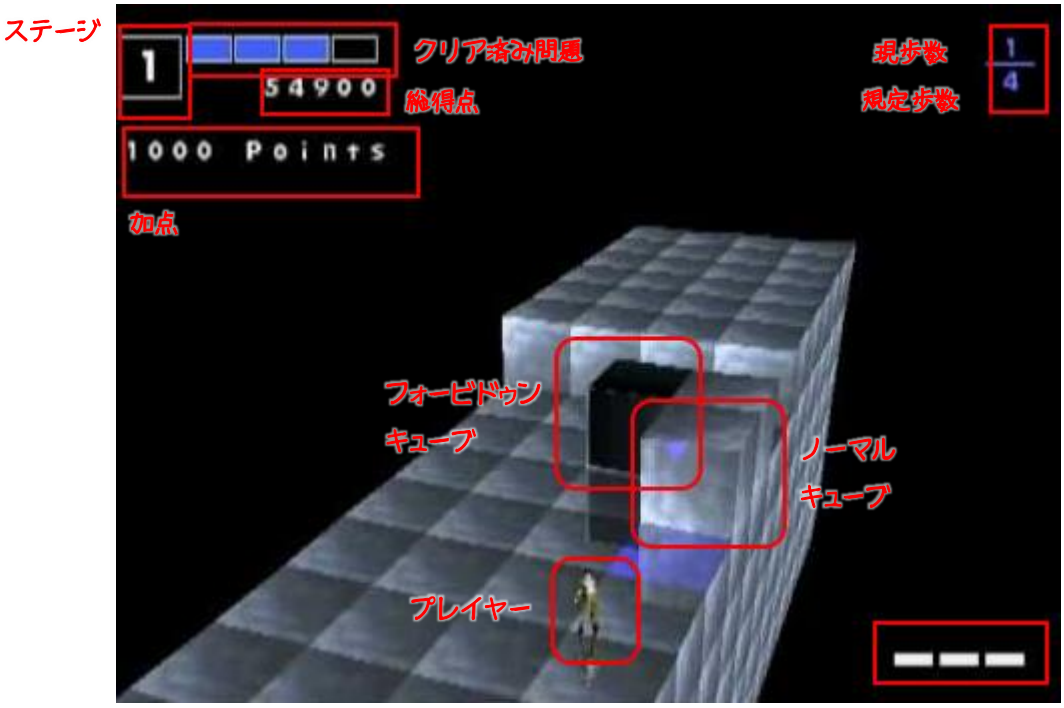
- プレイヤーはマス(キューブと同形状)の上を移動できる
- マスからキューブが一定列出現し、一定時間ごとにキューブが手前に転がる
- 手前に転がってステージの端にくるとそこからキューブは落下する
- 手前に転がるキューブに潰されても死なないがステージが１列ずつ崩壊する
- プレイヤーが潰された場合は残ったキューブは早送りでステージ端に落とされる
- ↑のとき後述するルールによりペナルティがある。さらにその問題はやりなおし。
- 崩壊は手前側から崩壊する
- プレイヤーは自分のいる位置の真下のマス(キューブと同形状)をマーキングできる
- ↑のときのマーキングエフェクトはマスの上部の色が青に変わる
- マーキングした部分はあとから「解除」することができる。この時赤に変わる
- 赤マーキングしたマス上にキューブが乗った状態で「解除」するとそのキューブが消える
- マーキングとマーキング解除は同じボタン(PS では○)で行う
- ↑の仕様により一度にマーキングできる箇所は１か所だけである
- キューブの種類は３種類あるが、全てに対して「消す」操作は可能である
- 「ノーマルキューブ」は消せるし全て消すことが目的である。
- 「フォービドゥンキューブ」を消してはいけない。
- フォービドゥンキューブを消すとステージが１列崩壊する
- ノーマルキューブやアドバンテージキューブが3つ落下するごとに１列崩壊する
- もちろん自分が潰された際に早送りで落とされるキューブはカウント対象である
- フォービドゥンキューブが落下しても通常ならステージ崩壊に影響は与えない！
- しかし、自分が潰された場合はフォービドゥンキューブもカウント対象である
- プレイヤーがステージの崩壊に巻き込まれるとゲームオーバー
- 「アドバンテージキューブ」を消すと緑色のマーキングが残る
- ↑の状態でボタン(PS では△)を押す事でそこを中心に９マス赤マーキングされる
- 既にマーク済みの部分は↑のアドバンテージマークの影響を受けない！
- □ボタンを押す事でキューブの転がりを早送りできる
- ノーマルキューブ消し 100 点、アドバンテージキューブ消し 200 点、
- ９マスマーキングで消すと↑の倍が加点される
- 問題は１ステージ当たり１定数(ここでは4とする)
- ステージクリア毎に残りの列数×1000 点加点される
- 各問題には「規定歩数(模範歩数)」があり、「歩数」とは転がり回数である
- 規定歩数を越えてクリアすると GREAT の称号がもらえる(2000 点)
- 規定歩数通りにクリアすると PERFECT の称号が貰える(5000 点)

- 規定歩数未満でクリアするとEXCELLENT!の称号が貰える(15000点)

と、箇条書きにすると、やたら多くなってしまう事が…わかるだろう？書くだけで疲れますな。
でも少なくともここまで細かくしないとゲームとして実現しないことも…わかるだろう？
ここまで書いても穴がある可能性が高い…これが仕様なのだ。

でも一旦書き出しておけば、あとはこれを実装するだけ…まぁ、うんざりするほど多いけど、
やらざるを得ない…。

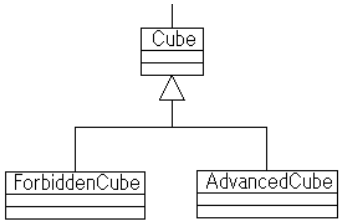
ひとまず既にあるDrawCubeは消してしまいましょう。一応ゲーム画面はこうなっています。



まずはこの体裁を整えていきましょう。ちなみに土台の厚みは5キューブ分に見えます。最初のステージでは4×6くらいの土台にキューブが盛り上がってきて4x2くらいの問題から出題されます。

ひとまず最初の問題はチュートリアル的に
□◇□■
□□□□
こんな感じです。□がノーマル。■がフォービドゥン。◇がアドバンテージという事にしています。
とりあえず今は効率の事など考えずに素直に配置してみましょう。

フォービドゥンキューブとアドバンスドキューブ



ひとまずは素直に継承させます。継承するので Cube のデストラクタは virtual にしてください。
とりあえず違いは色のみという事で…あと Cube の private を protected にします。つまりそれが

```
#include<vector>
#include<memory>
#include"../Geometry.h"

class Cube
{
protected:
	int _zushinH;
	VECTOR _center;
	VECTOR _direction;
	float _angle;
	float _diff;
	int _frame;
	std::shared_ptr<MATRIX> _mat;
	std::vector<VERTEX3D> _vertices;
	std::vector<unsigned short> _indices;

	void (Cube::*_updater)();

	void WaitUpdate();//待ち状態
	void RollingUpdate();//転がり中
	void RolledUpdate();//終わった時点で重心再計算

public:
	///キューブ一片の長さ(共通)を返す
```

```
        const static float GetLengthOneSide();
        Cube(const Position3f& pos);
        virtual ~Cube();
        void Update();
        void Draw();
        void RollOver(const VECTOR& direction);
};
```

である。そして

```
#pragma once
#include "Cube.h"
class AdvancedCube :
        public Cube
{
public:
        AdvancedCube();
        ~AdvancedCube();
};
```

および

```
#pragma once
#include "Cube.h"
class ForbiddenCube :
        public Cube
{
public:
        ForbiddenCube();
        ~ForbiddenCube();
};
```

というわけだ。ひとまずは色だけなので、それ以外の部分は継承して使用すればいいでしょう。

Cube 自身がこのようなコードになります。

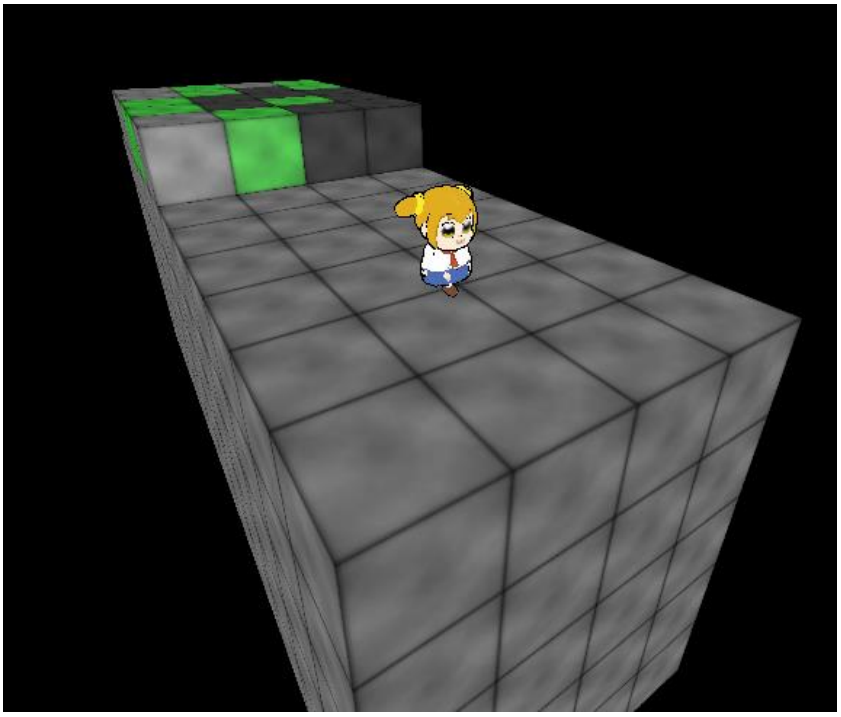
```
#pragma once
#include<vector>
#include<array>
#include<memory>
#include"../Geometry.h"
```

```
class Cube
{
protected:
	int _cubeTex;
	int _zushinH;
	VECTOR _centerpos;
	VECTOR _direction;
	float _angle;
	float _diff;
	int _frame;
	std::shared_ptr<MATRIX> _mat;
	std::vector<VERTEX3D> _vertices;
	std::vector<unsigned short> _indices;

	void (Cube::*_updater)();

	void WaitUpdate();//待ち状態
	void RollingUpdate();//転がり中
	void RolledUpdate();//終わった時点で重心再計算
	//中心座標を計算する(頂点集合)
	VECTOR CalculateCenterPos(const std::vector<VERTEX3D>& vertices);
	//全てのキューブに共通の初期化(頂点定義など)
	void CommonInitialize(const std::array<VERTEX3D,4>& orig_verts);
public:
	///キューブ一片の長さ(共通)を返す
	const static float GetLengthOneSide();
	Cube(const Position3f& pos);
	virtual ~Cube();
	void Update();
	void Draw();
	///転がす
	///@param direction
	///@note directionの取りうる値は
	///(1,0,0)(-1,0,0)(0,0,1)(0,0,-1)のみとする
	void RollOver(const VECTOR& direction);
};
```

まぁうまい事できれば、こうなる

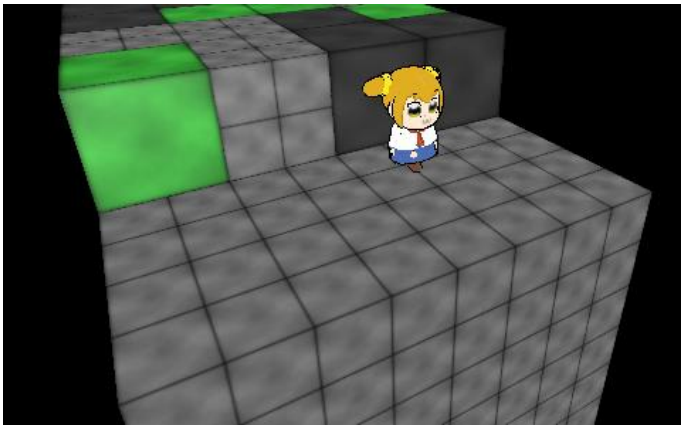


いいかがかな？

無駄をなくそう

ちなみに現段階では土台キューブは本当に4x5x12 個用意しているので非常にもったいない…。どうせ動かさないし全部同じ色なんだからどうにかまとめたい。ドローコール数も多くなりますしね…

ところでこいつを見てくれ…こいつをどう思う？



すごく…たくさんです。
実はキューブの数自体は変化していない。どうすればいいと思いますか？

そういえば、UV 値が 0～1 の範囲を超えた時の挙動についてご存知かな？

その時の挙動は設定によって決まるのだ。そしてその設定とは…

テクスチャアドレッシングモード

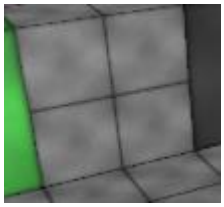
テクスチャアドレッシングモードと言う設定が DxLib にも DirectX にも OpenGL にも存在する。ただ、設定の方法は各ライブラリで違うのだが…とりあえず『テクスチャアドレッシングモード』というのがあると思ってほしい。

そしてそいつは何かというと、テクスチャ座標の割り当て方を規定できる。難しいかな。ひとことで言うとテクスチャ座標が 0～1 の範囲を超えた部分の表示をどうするかである。

で、結論から言うと大きく 4 種類くらいの中から選ぶことになる。

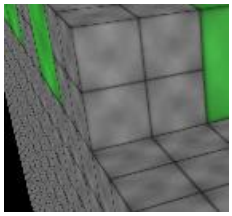
- WRAP:繰り返し
- MIRROR:繰り返し(ただし反転)
- CLAMP:端っこの色を引き延ばす
- BORDER:デフォルト色

さて、それぞれどのような挙動かイメージできるだろうか？WRAP だと繰り返しなので



同じ画像が並んでいるのが…わかるだろう？

次にミラーだが

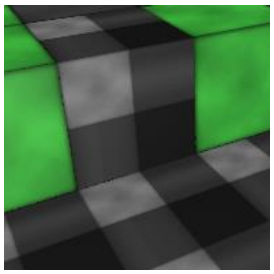


ご覧のように上下左右が対称形となる

大体使用するのはこの2つなんだが

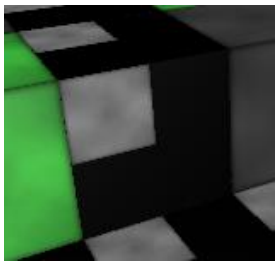
例えば、デフォルトであるクランプを適用すると

こうなる



端っこの色が出ているのが…分かるだろう？

最後にボーダーだが…



デフォルト値が黒なので真っ黒

さて、DxLib では便利な事に１行追加するだけでこの状態にすることができる…それが

SetTextureAddressModeUV

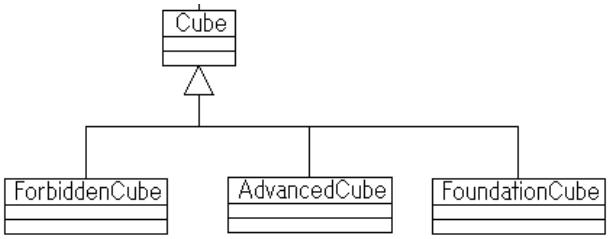
である。

http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N21

を参照すると使い方が記載されているので、GamePlayingScene の頭で、これが呼ばれるようにして、WRAP モードにしていればいいと思います。そのうえで

UV 値の大きい方を n 倍にしてやればよいのです。

という事で、新たにキューブの仲間を増やしましょう…土台は英語で Foundation というので FoundationCube とでもしておきましょう名詞が連続してるので良くないかもしれませんがなんとなく意味が伝わればいいのです。BaseCube とかでも良かったんですがちょっと微妙だった(意味が誤解されそうな)ので…。



こうしましょう

で、他同様 Cube からの派生でいいんですが、コンストラクタが微妙に違って、左奥上と右手前

下の座標を渡したら適切な(見た目たくさん)ブロックが配置されるようにしましょう。

ただしここで注意すべき点がある。今回のこれは立方体ではない。直方体である。つまり以前に用意したような展開図回転だけではうまくいかない。面倒だが仕方がない。

相手は直方体だ…どうする？

でいつもの通りアイデアの一つを公開するけど、これだって、最初のアイデアを使ってるわけでもなくあーでもないこーでもないっていう時間をかけとるんやで？俺がいくら天才やゆーてもな、一足飛びに最適解出しとるわけやないんやで？もちろん最適解でも何でもない。

話しだけ聞いたら「ふーん、そっかー」だけど、これを捻りだす思考過程が重要なのだ。正直展開図を書いたり頂点の共通点や法則を考えたり色々とやった…そして一つの考えを最適かなと判断した。

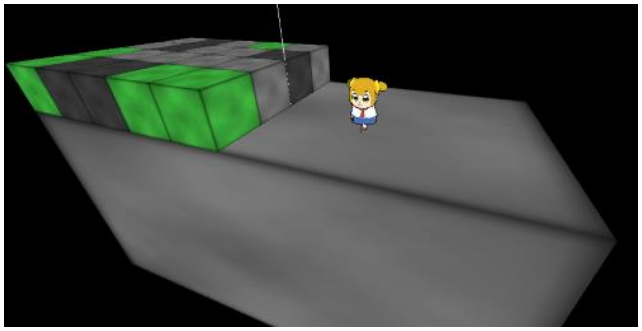
さて、それではやり方の一つを説明しよう。

まず、通常のキューブと同じように作る。出来上がったらそいつに対して縦横奥行き方向に引き延ばしを行うわけだがどうしようか？

意外と簡単なんだけど、X 方向、Y 方向、Z 方向それぞれの拡大率が分かっているとすればそれぞれの頂点の座標をそれぞれの拡大率で拡大すればいい。初めは MScale を使用しようかとも思ったが、こいつは均一拡大縮小しか対応しない無能なので自前でやることにした。なに…大したことじゃない。

```
///それぞれの拡大＋指定座標平行移動
for (auto& v : _vertices) {
        v.pos.x = v.pos.x*wnum+pos.x;
        v.pos.y = v.pos.y*hnum+pos.y;
        v.pos.z = v.pos.z*dnum+pos.z;
}
```

ちなみに wnum,hnum,dnum はそれぞれ X 方向 Y 方向 Z 方向のブロックの数だ。
やっている事はそう難しくもないだろう？それぞれあるべき大きさに拡大して、平行移動しているだけだ。これをやれば…



難しいのはここからでどうやって UV 値を設定しようか？既にテクスチャアドレッシングモードは WRAP(繰り返し)にしているならば、UV 値も wnum や hnum と同様に乗算してやればいい…ただしここで難しいのは U と V がどのように対応しているのかという事だ。

どういう事かと言うと、真正面と真奥に関しては Z 方向の干渉はないため dnum は関係ない。また同様に左右側面の場合は X 方向の干渉がないため wnum は関係ない。上下も hnum は関係ない…という事になる。

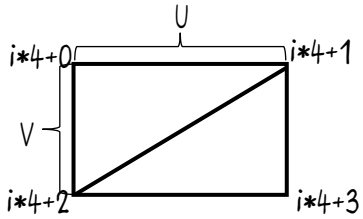
何が問題かと言うとそれぞれの U と V ががが w,h,d のどれに対応するのか分からないのである。これをどう解決したらいいんでしょうかね？

これも頭の体操だと思ってしばらく考えてくれ。ある意味パズルだよね？

そしてスッキリした解決法を思いつく。意外なやり方である。まず一旦すべての頂点座標を決定した後で…

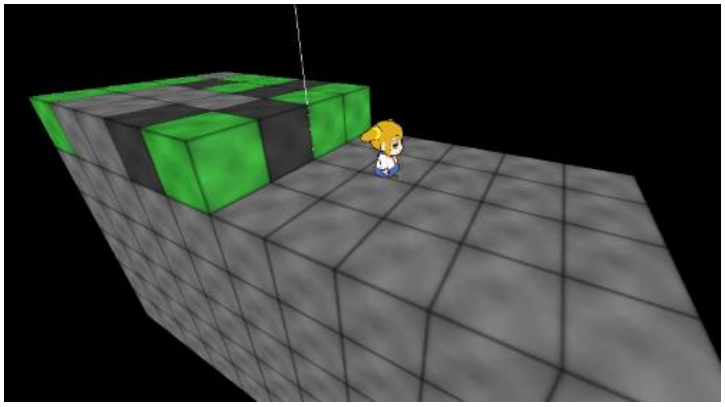
```
float uscale = DxLib::VSize(VSub(_vertices[i * 4 + 1].pos, _vertices[i * 4 + 0].pos)) / ed_w;
float vscale = DxLib::VSize(VSub(_vertices[i * 4 + 2].pos, _vertices[i * 4 + 0].pos)) / ed_w;
```

このような事をする。やっている意味は分かるだろうか？



全ての面において↑の図のような位置関係になっているはずであるため、0～1 間 0～2 間の大きさを求めてやれば U に当たる部分と V に当たる部分のスケールが分かるはずである。

更に言うと、0～1がどのような方向に延びているかも分からないためxyzと分解して計算するよりまとめて長さを測ったほうが良いという事にした。これによりUVに対して適切にスケーリングできるため



のように適切な繰り返しが見れるようになります。

自転

まぁ、色々とやってはきましたが、当然ながらキューブは自転します。しかも 1 方向へ自動で転がります。前後左右よりかは簡単ですよね。時間的なものは待機 2 秒転がり 1 秒くらいでまずは作りましょうか。

カメラ挙動とプレイヤー

カメラはプレイヤーを追いかけよう

基本的にこのゲームにおいてはカメラ座標はプレイヤーと一定の距離を保ちつつ動きます。
イメージとしてはプレイヤーとカメラが固定棒(自撮り棒)でくっついている感じですね。UE4のサードパーソンのカメラなんかがこれに当たりますね。

ということはプレイヤーが動くとカメラが動かなければいけませんね？一定の距離は保つ必要がありますね？

という事で、今現在

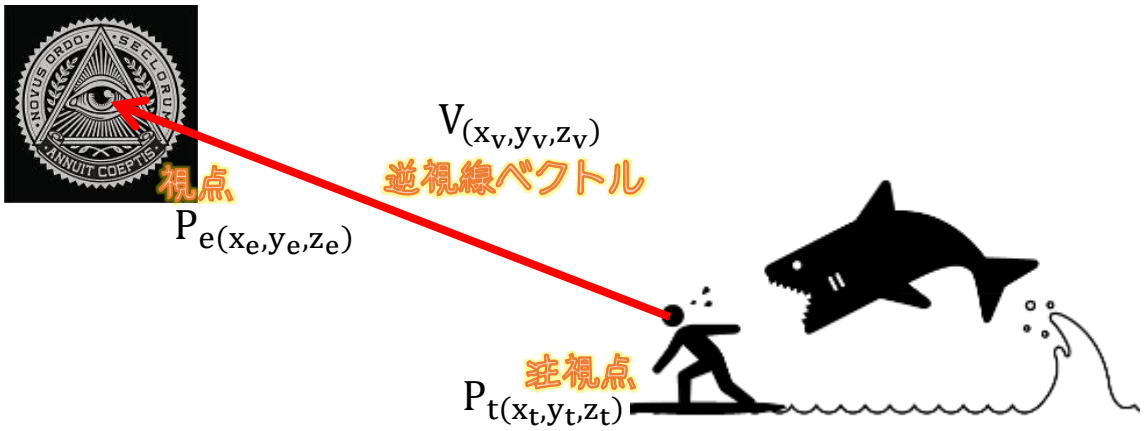
```
DxLib::SetCameraPositionAndTarget_UpVecY(
        VGet(0, 15, -25),//視点、
        VGet(0, 10, 0));//注視点、
```

のような設定をしていると思います。ご覧になれば分かる通り固定値です。通常であれば

_eyepos:視点、
と
_targetpos:注視点、
の 2 つの座標変数をメンバ変数として用意するところですが、先ほど言った動きにするならこれはちょっと違います。

重要なのは注視点で、視点の方はそれに追従します。

となると注視点から視点を逆算するベクトルがあればいいという事になります。
図に書くと



さて、ここで言ってる「逆視線ベクトル」というのは当然終点から始点を引いて作りますので

$$V\,(x_v, y_v, z_v) = P_e(x_e, y_e, z_e) - P_t(x_t, y_t, z_t)$$

ですね？つまり

$$V\,(x_v, y_v, z_v) = V(x_e - x_t\,, y_e - y_t\,, z_e - z_t)$$

なわけです。

ここまではご理解いただけますかね？さて、この V が分かっていさえすればあとは注視点が分かれば視点が求められますね？そう…注視点にこのVを足せばいいのです。

$$P_t(x_t, y_t, z_t) + V(x_e - x_t\,, y_e - y_t\,, z_e - z_t) = (x_e, y_e\,, z_e)$$

というわけで、初期状態の時点でまずは注視点と逆視線ベクトルをメンバ変数とします。注視点はプレイヤー座標なので、注視点も正直いらないです。

```
Position3f _targetpos;//注視点
Vector3f _toEyeVector;//注視点側から視点へのベクトル
```

既に

```
DxLib::SetCameraPositionAndTarget_UpVecY(
        VGet(0, 15, -25),//視点
        VGet(0, 10, 0));//注視点
```

と言う感じにカメラを定義しているのならば

```
_toEyeVector = VSub(VGet(0, 15, -25),VGet(0, 10, 0));//逆視線ベクトル
```

であるわけですが一旦これで位置関係を決めてしまえばあとは

```
auto eye=_player->GetPosition()+_toEyeVector;
```

てな感じで視点が求まりますので、これによって毎フレームカメラを更新すればいいでしょう。

```
DxLib::SetCameraPositionAndTarget_UpVecY(eye,playerpos);
```

はい、これでプレイヤーにカメラが付いて回ります。

カメラはプレイヤーを中心に回転しよう

とはいえこのゲームでは完全にカメラ固定だとプレイしにくいですね。キューブにプレイヤーが隠れてしまいますし、位置によっては見たいキューブも見えにくいですし。

さて、今固定している状態ですがどのようにして回転させればいいのでしょうか？

sin,cos で回転させてもいいのですが、はっきり言って今は勝手に視点と注視点を決めてるので、回転と言っても何処基準に回転すればいいのかよくわかりません。

そもそも**回転と言うと sin,cos しか思いつかないようでは芸がありません**ね。

おっ、そうだな。

プログラマと言うのは選択肢が多い方が腕がいいと言っても過言ではありません。ちょっと考えてみてください。

俺もやったんだからさ。

俺はどこぞの教科書を見たり、どこぞの答えを知ってるわけでもない。ただ「考える時間」をアホほど費やしたに過ぎない。スパーリングしてる時も自転車運転中でも考えているのだ。

さぁ…自分で考えよう。

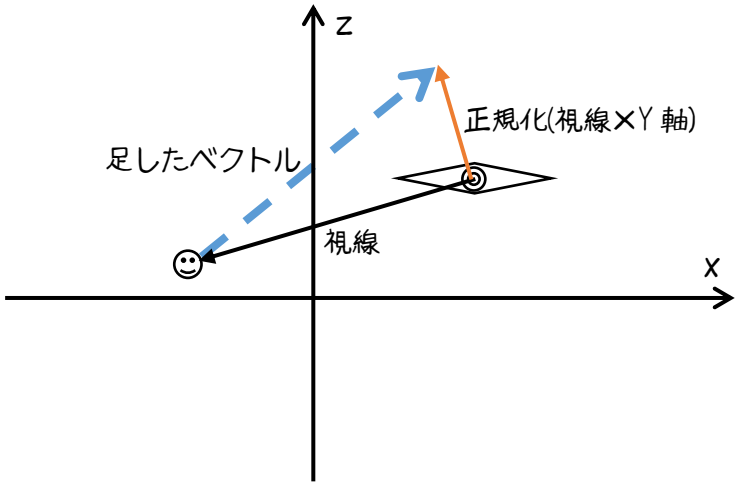
さて、ここで僕のやり方を説明しよう。二つの物を使用する。それは

- 外積
- 正規化(標準化)…大きさを１にすることやな

オンリーである。

さて、どのようなトリックを使うのだろうか？

3D 的な話なのでちょっと説明が難しいが現在の視線ベクトルは真上から見るとこうなっているはず。
ここに(0,1,0)ベクトルと視線ベクトルの外積をとって正規化すると直交するベクトルが得られる。



で、ここで得られたベクトルと逆視線ベクトルを足すやん？で、このままやと当然ながら

視線ベクトルの長さ＜足したベクトルの長さ

ですから、これもまた長さをそろえる必要があります。

つまり求めたい視線ベクトルは

正規化(足したベクトル)*元の視線ベクトルの長さ

になるわけです。これをやれば等角速度で回転していきます。

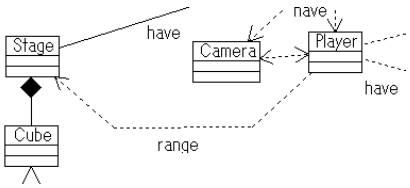
この考え方は飛行機の旋回や回転物体全てに応用できると思いますので知っておくと良いでしょう。

sin,cos で回したい人はそれはそれで構いません。

ゲームルール部分②

プレイヤーの動ける範囲

プレイヤーの動ける範囲はもちろん土台がある部分のみです。ただ、土台キューブからこの情報を取ってくるのはちょっと違う気がします。というわけで、



Stage クラスを作って、そいつがそのステージにおけるプレイヤーの行動可能範囲を知っている事としましょう。と、大げさに言うておりますが、こいつはただ単に足場が

$$n \times m$$

であることを知っていればいい。それだけである。足場の正確な幅と言うか、縦横のキューブ数を知ってればいい。もちろん将来的に崩壊した列は減らすという風に考えておく。

とりあえずひとまず Stage は内部に n×m を持っておいて、そこからプレイヤーの可動範囲を渡せるようにしよう。そして Stage 情報をプレイヤーが知っている状態にしたいので Stage への参照(シェアドポインタ)をプレイヤーが持つことにする。

ひとまず Stage クラスはこんな感じで定義しています。

```
///ステージクラス
///@brief ステージのデータロードやら可動範囲やらを制御します
class Stage
{
private:
	Size _cubesRange;//縦横キューブ数
	RectF _movableRange;//可動域
```

```
public:
	Stage();
	~Stage();

	///縦横キューブ数
	///@return 範囲Rect
	const Size& GetCubesRange()const;

	///プレイヤーの動ける範囲
	///@return 範囲Rect
	const RectF& GetPlayerMovableRange()const;

	///ステージのセットアップ
	///@param one_side_edge_f
	///@remarks この関数を呼ぶことでプレイヤーの可動範囲が計算されます
	///キューブの大きさをステージ側で定義したくはないので…
	void SetupStage(float one_side_edge_len);
};
```

まぁこんな感じ…SetupStageだけ見せておくと…

```
void
Stage::SetupStage(float oneside_len) {
	_movableRange.center.x = 0;
	_movableRange.center.y = 0;
	_movableRange.size.w = static_cast<float>(_cubesRange.w)*oneside_len;
	_movableRange.size.h = static_cast<float>(_cubesRange.h)*oneside_len;
}
```

こんな感じ。やってる事(意味)は分かりますよね？これがプレイヤーの可動範囲になります。
ちなみに_cubesRangeはステージごとに定義されるもので今は定数で

```
_cubesRange.w = 4;
_cubesRange.h = 10;
```

と定義しています。

で、これでプレイヤーの移動を制限すればいいのですが、プレイヤーをModelクラスのMove関数で動かしているので、制限させづらいので思い切って制限付きMoveを作りました。

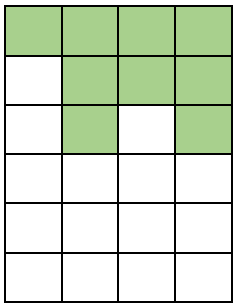
```
///特定の座標から移動する
///@param vel 今の座標からのオフセット
void Move(const Vector3f& vel);

///特定の座標から移動する(制限範囲付き)
///@param vel 今の座標からのオフセット
void Move(const Vector3f& vel,const RectF& range);
```

下の関数の、第二引数が制限範囲です。これでプレイヤーの可動範囲を制限できます。

ブロックがある部分に入れないようにしよう

これの判定ねえ…3D の立方体との OBB やっちゃってもいいんですけど面倒だしもう時間がないので、2D のマス目で判定します。例えば以下のような感じで定義します



図の緑色の部分がプレイヤーが入れない部分です。さて、このデータの緑の部分には入れないし、白い部分には入れるとします。この状態が配列になっていればいいわけです。とりあえず char 型で std::vector しとく

```
std::vector<char> _cells;
```

あれ？2 次元配列じゃないの？と思う人もいるかもしれないですが、こういう時は 2 次元を使うまでもないです。可変長×可変長ならば二次元ベクタですが、固定長×固定長なら 1 次元ベクタで十分です。

なお、2 次元固定配列および 2 次元動的配列は使用しません。理由は

二次元固定配列だとステージ構成が限定されてしまう(実際の IQ では幅が 4～8 に変化する)ため、固定配列は使えないし 2 次元動的の場合、列ごとにまた new しなければならず煩雑になるからです。そこで 1 次元ベクタ。

1次元ベクタはいいぞ。
何より扱いやすい。

ここで1次元ベクタを2次元のように扱う方法についてだけど、まぁお分かりのように

_cells.resize(n*m);
で幅mの奥nぶんの領域を確保し

_cells[j*m+i]

で(i,j)の位置のセルを参照できまぁす!!

例えば先頭1列消したければ
_cells.erase(_cells.begin(),_cells.begin()+m);
で消せまぁす!!

おケツの1列消したければ
_cells.erase(_cells.end()-m,_cells.end());
でいいんでぇす!!

逆にインデックスから場所を知りたければ
x=idx%m;
y=idx/m;
でいいんでぇす!!!

そもそも2次元配列なんて1次元配列のシンタックスシュガーでしかないんだから慣れた人間にとっては

2次元配列は不要!はっきり分かんだね!!

と2次元配列不要論を唱えたところで、enumも作ります。

```
///セル(マス目)タイプ
enum class CellType {
	none,//キューブなし
	natural,//通常キューブあり
```

```
        advantage,//アドバンテージキューブ
        forbidden//禁じられたキューブ(消したらあかんやつや)
};
```

しれっとenum class使っとるけど大丈夫やんな？

さて、こいつをStageが所持した状態でまずは全部noneで埋めて、現在ブロックがある部分をnaturalで埋めよう。そしてそこに入れないようにしよう。

ひとまず埋めるところですが、せっかくだからstd::fillを使ってみます。

#include<algorithm>
で使えるようになります。

使い方は各自調べてみてください。

```
std::fill(_cells.begin(), _cells.end(), static_cast<unsigned char>(CellType::none));
std::fill_n(_cells.begin(),_znum*_cubesRange.w,static_cast<unsigned char>(CellType::natural));
```

さて、これが何を意味しているのかは各自考えてください。

とりあえずこれでマス目上のブロックの状態ができました。次にやるべきことはプレイヤーの現在座標からブロックの何番目に当たるのかを計算し、位置を補正します。

まずはプレイヤーが動いていない状態の時の座標補正からやります。これはZのマイナス方向(キューブが進行する方向)に補正してやればいいです(もしキワキワにいたら一番近いところに逃がすべきですが…)

まずはプレイヤーの座標からマス目番号を返す関数です。

現在の動ける範囲は
```
const RectF& GetPlayerMovableRange()const;
```
で取得できるようにしています。

ちょっと自分の頭で考えて、これを利用してプレイヤーの居る位置のインデックスを取得する関数を作ってみてください。

次にそのインデックスを元にそこのマスの状態を返す関数を作ります。

```
CellType
Stage::GetCellType(int idx) {
	return static_cast<CellType>(_cells[idx]);
}
```

簡単ですね。ハイ次
今度はそのインデックスのレンジ(矩形)を返す関数を作ります。

```
RectF
Stage::GetCellRange(int idx)const {
	RectF rc;
	auto xidx = idx % _cubesRange.w;
	auto yidx = idx / _cubesRange.w;
	rc.center = Position2f(_movableRange.Left() +xidx*_oneside_len+_oneside_len/2,
_movableRange.Bottom()-yidx*_oneside_len - _oneside_len/2);
	rc.size.w = _oneside_len;
	rc.size.h = _oneside_len;
	return rc;
}
```

ちなみに RectF は Rect の Float 版です。
ついでにその部分も公開しておくと、こうです。

```
///矩形を表現する構造体
template<typename T>
struct RectBase {
	Vector2D<T> center;
	SizeBase<T> size;
	RectBase();
	RectBase(T x, T y, T w, T h);
	RectBase(Vector2D<T>& pos, SizeBase<T>& sz);
	const T Left()const;
	const T Top()const ;
```

```
        const T Right() const;
        const T Bottom()const;
        const T Width()const { return size.w; }
        const T Height()const { return size.h; }

        ///矩形を描画する
        ///@param color 色を0x00000000～0xffffffffで指定する(デフォルト白)
        void Draw(unsigned int color = 0xffffffff);

        ///矩形を描画する(オフセットつき)
        ///@param offset ずらした位置に表示したい場合にオフセット値を指定
        ///@param color 色を0x00000000～0xffffffffで指定する(デフォルト白)
        void Draw(const Vector2& offset,unsigned int color = 0xffffffff);
        void Draw(const Vector2f& offset, unsigned int color = 0xffffffff);

        ///左、右、上、下から矩形を作る
        ///@param left 左
        ///@param right 右
        ///@param top 上
        ///@param bottom 下
        static RectBase CreateRectFromLRTB(T left, T right, T top, T bottom);

        ///二つの矩形から重なり矩形を作る
        ///@param rcA 矩形A
        ///@param rcB 矩形B
        static RectBase CreateOverlappedRangeRect(const RectBase<T>& rcA, const
RectBase<T>& rcB);
};

typedef RectBase<int> Rect;
typedef RectBase<float> RectF;
```

微妙にテンプレートと typedef を駆使しているので戸惑うかもしれませんが、分からんかったら素直に RectF を1から作った方がいいです。

マルチスレッドの強い味方

ところで、俺の知らんうちに、マルチスレッドには強い味方が増えているようです。昔は _beginthread でひーひー言ってたんですが、もうそんなものに悩まされることも少なくなるね!(なくなるとは言っていない)

_beginthread とかで作ると、スレッドセーフに作っていくのがクッソ大変だったんよ…でも今はそうじゃない。そうじゃないんだ!!ありがたい…本当にありがたい。_beginthread と CreateSemaphore の組み合わせとか、考案者をブチ殺したくなるほど面倒だったんだよー。

マルチスレッドを使うために便利なテンプレートクラスが

- std::thread
- std::atomic
- std::mutex
- std::condition_variable

これらなんですけどね？昔はこれと同じ仕組みをする奴を自分で作ってたんだわー。なんであの…あの…月 600 時間煉獄に入る前に何故存在しなかったんですかねぇ…。

まぁ、ぶっちゃけ現代では、どっちみち GPU 側のマルチスレッド地獄が待っているのですがね？それへの布石として通常のマルチスレに足を踏み入れるのも良いかと思います。