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

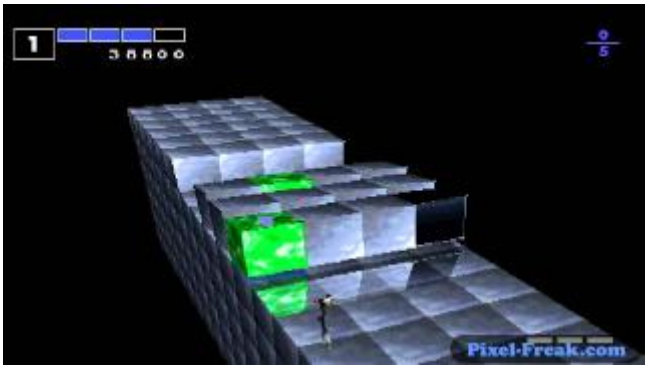
ぼくは正直者だから「簡単」とは言いません。言いませんが、DxLib を使っていきますので、それほど難しくもないと思います。

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

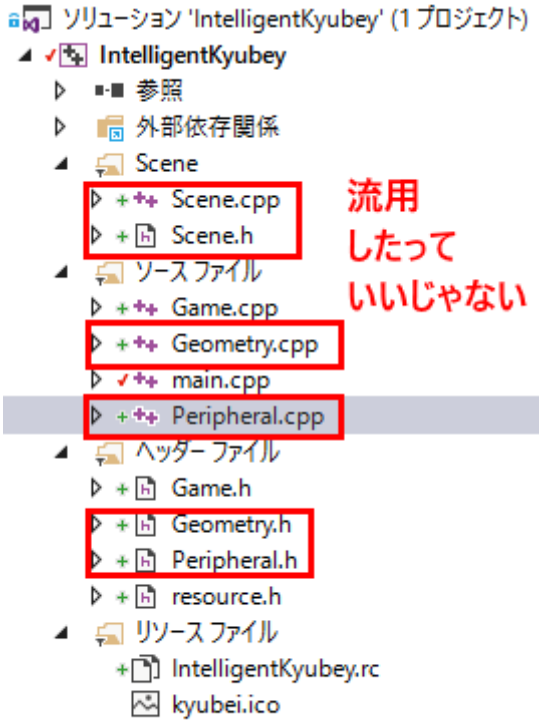
```
class Game {
private:
	Game();
	Game(const Game&);
	void operator=(const Game&);
public:
	~Game();

	//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。

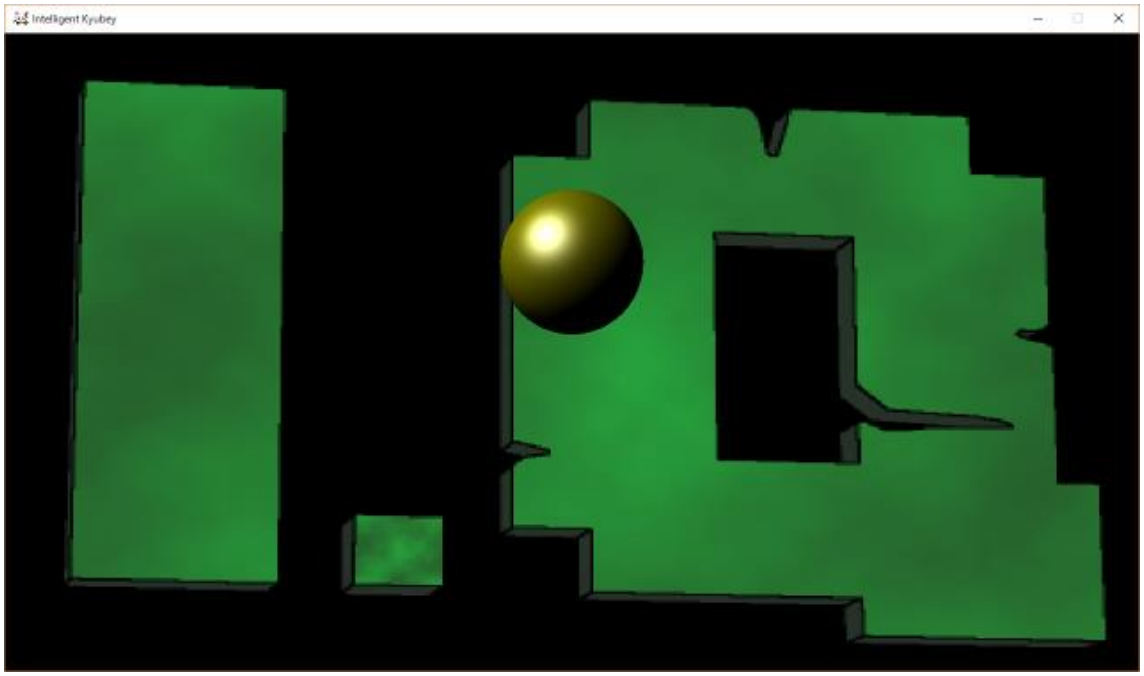
3D 事始め

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。

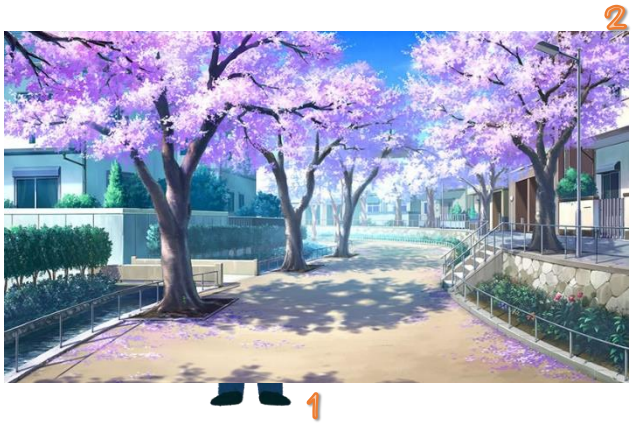
というわけでいくつかの用語を解説しておきます。なお、これは多分後期でも言いますがこのGoogle先生時代においては「用語」がかなりの「力」を持ちます。いわゆる「検索力」につながります。

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。

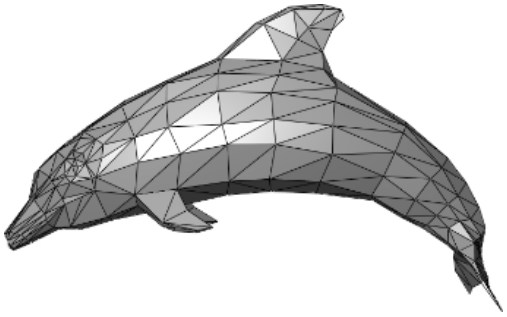
Zバッファ(深度バッファ(デプスバッファ))

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



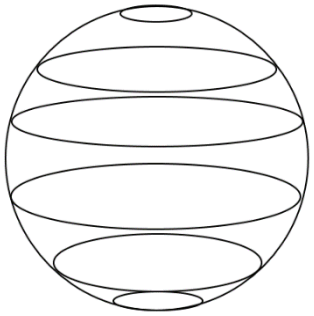
こうなる。これは分かりますよね？
だから描画の順序が大切なんです。で、3Dモデルと言うやつは無数の三角形の集合体でできている事は何となく知っていますね？



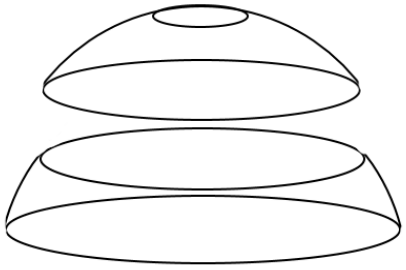
こういうやつね？
で、この三角形も前後の順序を間違えると向こう側が見えちゃったりするわけですよ。イルカくんの胃袋も心臓も肝臓も腸も見えちゃうわけですよ。

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。

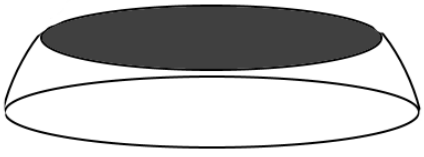
書き方としては、



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。

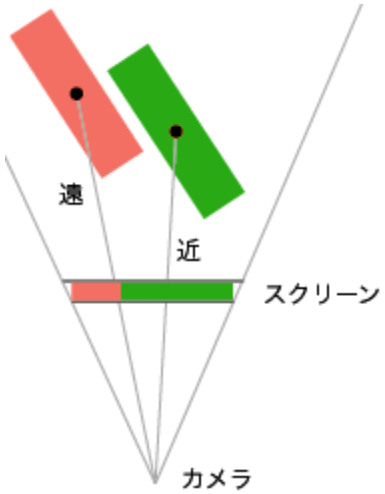


なんとなく原理は分かりましたね？3D というのは面倒な事だらけなのです。

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…

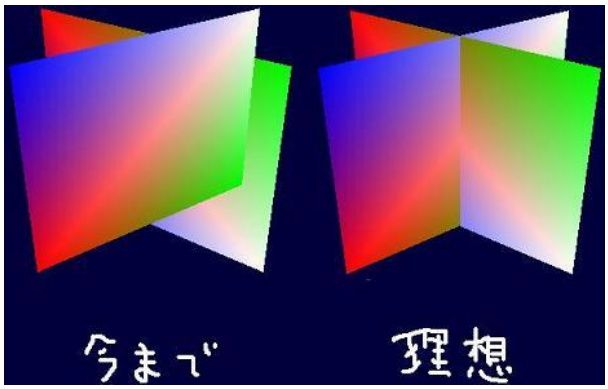
しばらくはその時代が続きました……。ただ、その当時から問題点は指摘されていて…



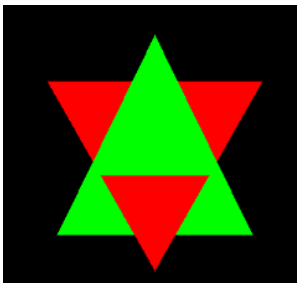
こういう位置関係だったとする。「比較されるポリゴンの座標」っていうのが、ポリゴンの頂点の座標の平均から割り出しているので、上の図のようなおかしなことが発生します。

また、それ以外にも「突き抜け」問題があって…

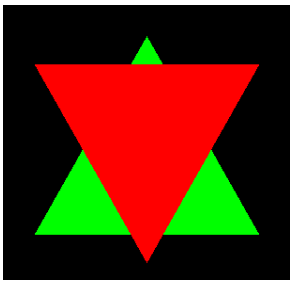
ポリゴン同士が奥行き方向にクロス、もしくは突き抜けていたりするともっとえげつない問題が発生します。



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



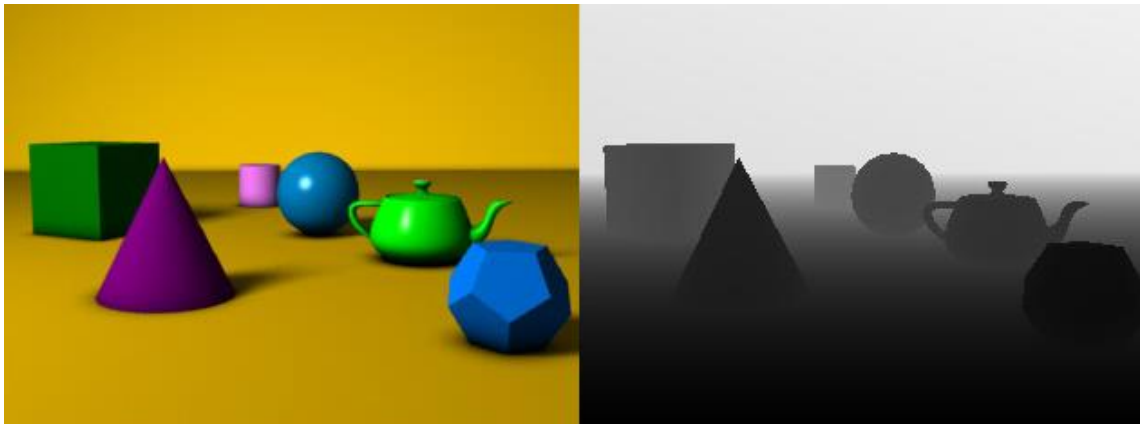
現実

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)

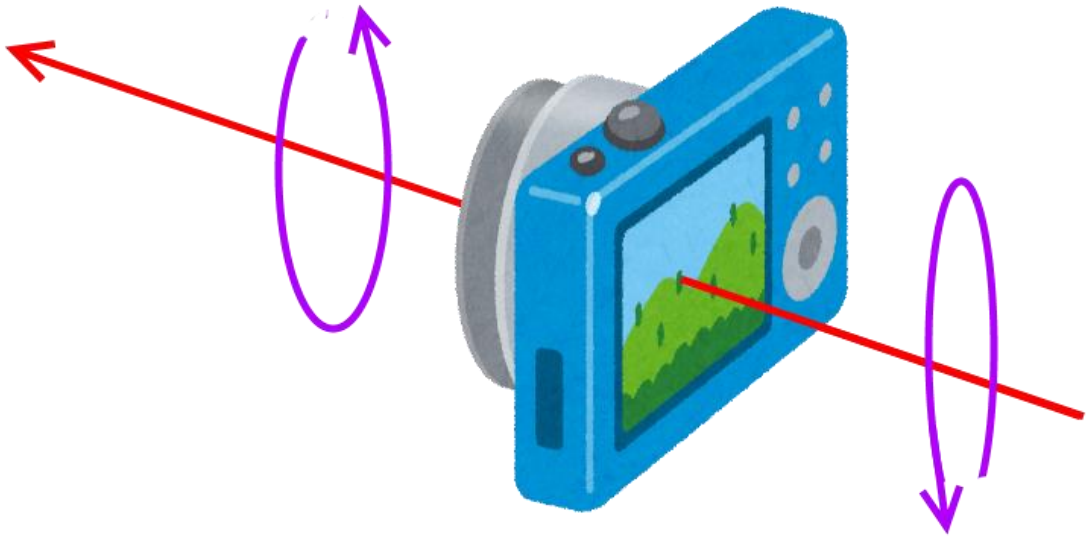
次にカメラの概念を話します。さっきも「カメラからの距離」なんて言い方をしましたね。カメラの座標と言うのは視点の座標であり、3D空間上のどこかに設定します。

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

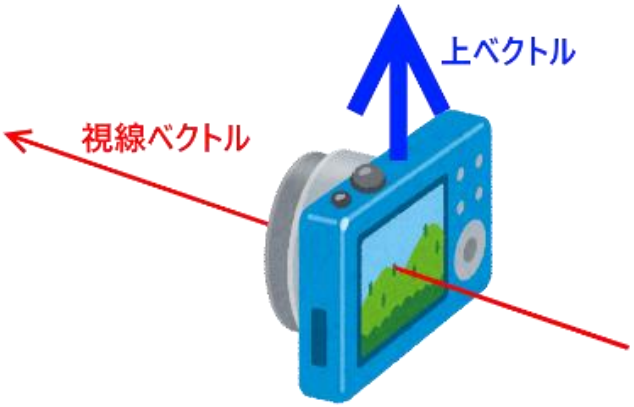
カメラは「視点」と言って、カメラが中心として写す被写体(点)を「注視点」と言います。そして、視点から注視点に向かうベクトルを「視線ベクトル」と言います。

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。

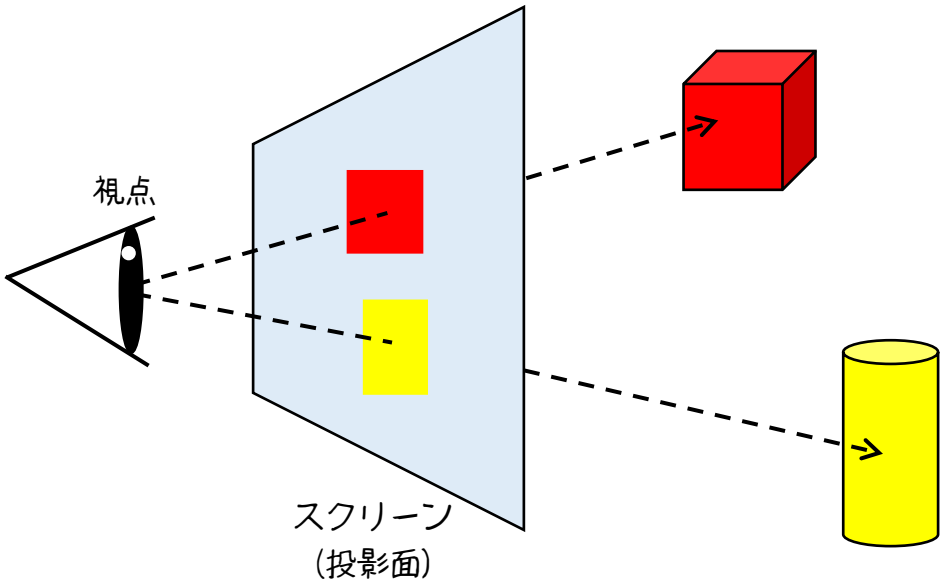
なお、この**視線ベクトルと上ベクトルを外積してさらに外積**して、カメラから見たXYZ座標系と言うのを作って、カメラ座標系と言ったりします。

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。

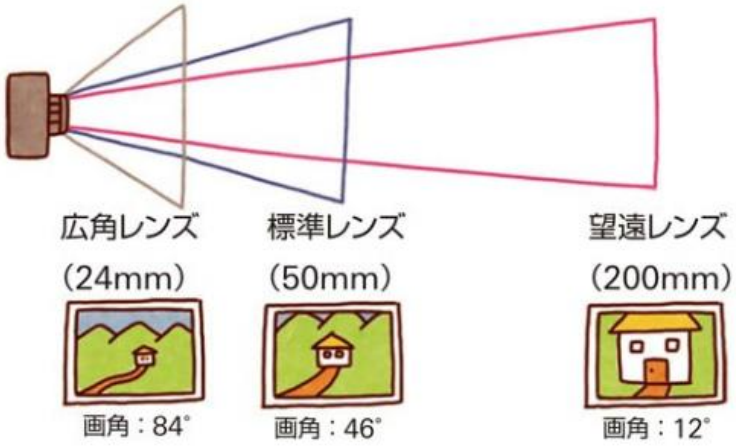


というわけで、視点とスクリーンの位置は一緒ではないのでちょっとややこしい…注意しましょう。

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？

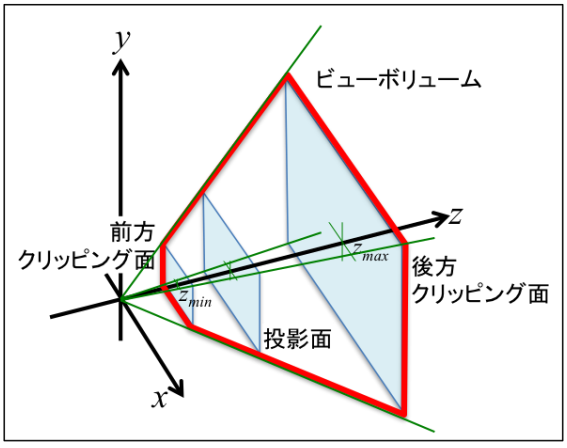
いわゆる遠近法と言うやつです。

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

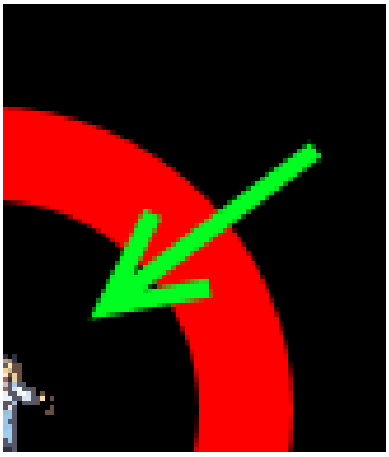
読み込むには MV1LoadModel という関数を使います。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R1N1

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

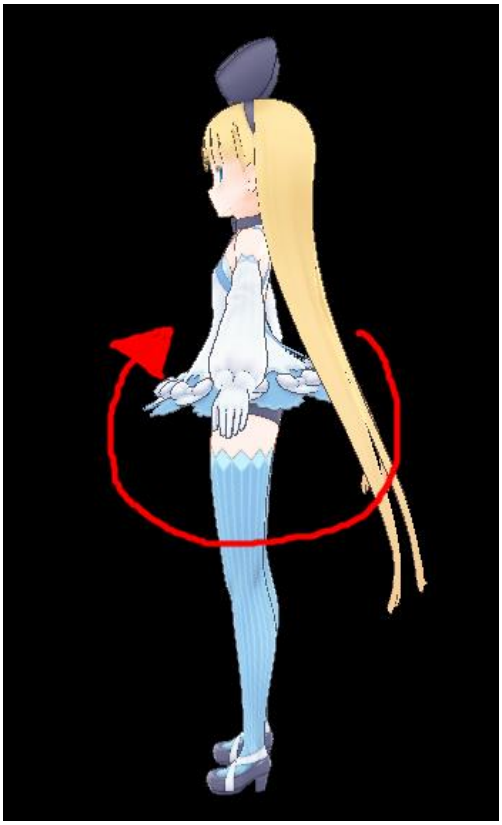
で行えます。うまいこと設定できてれば

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。



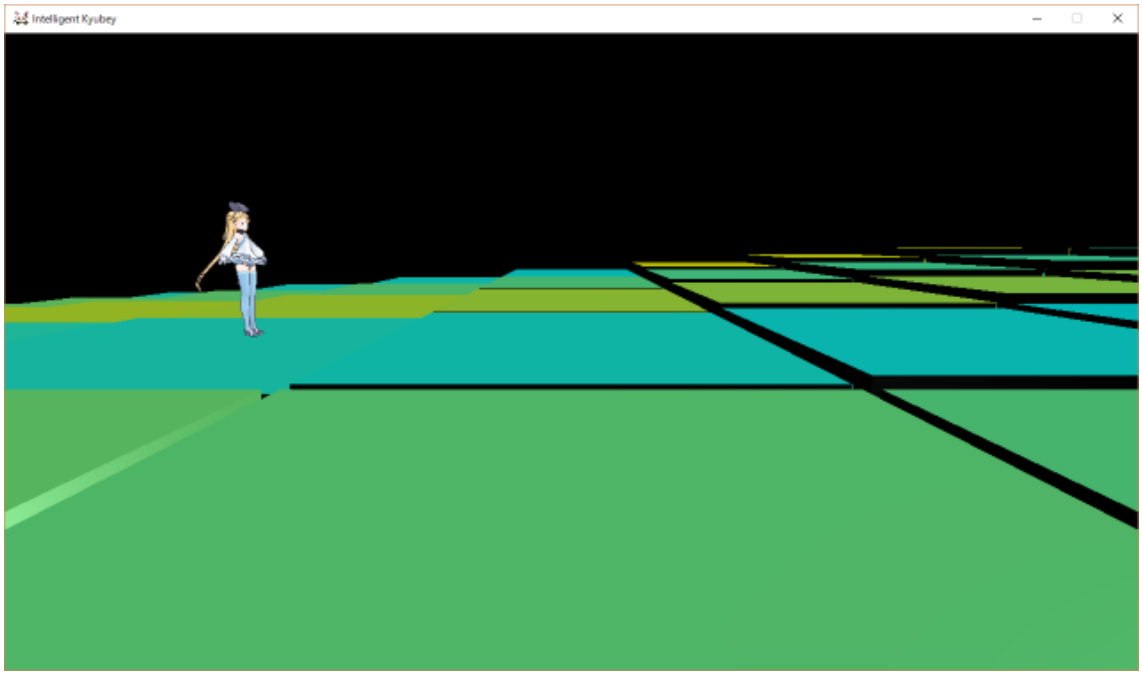
くるくるとね

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

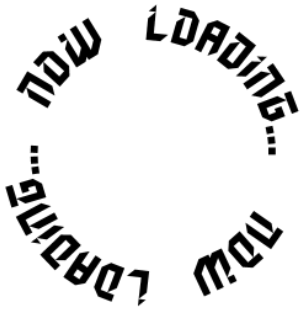
```
float MV1GetAttachAnimTotalTime( int MHandle, int AttachIndex ) ;
```

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

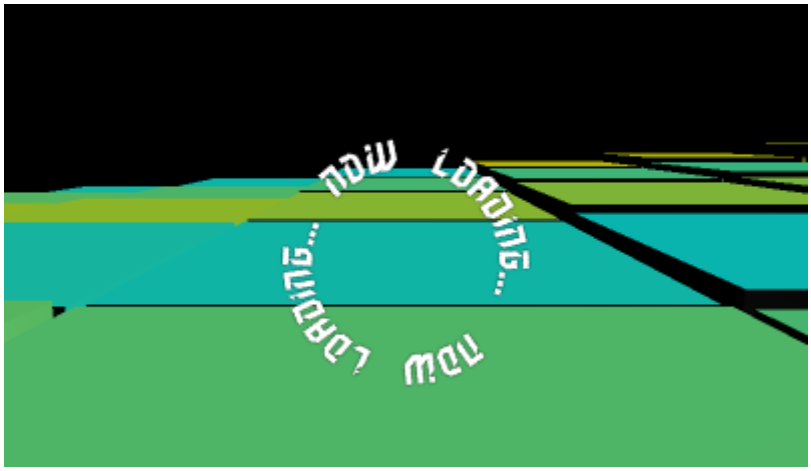
SetUseAsyncLoadFlag を true にすると非同期読み込みになります。3D でも同様です。関数自体は即時復帰ですが、読み込みが終わり次第描画されます。

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。

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
} ;
```

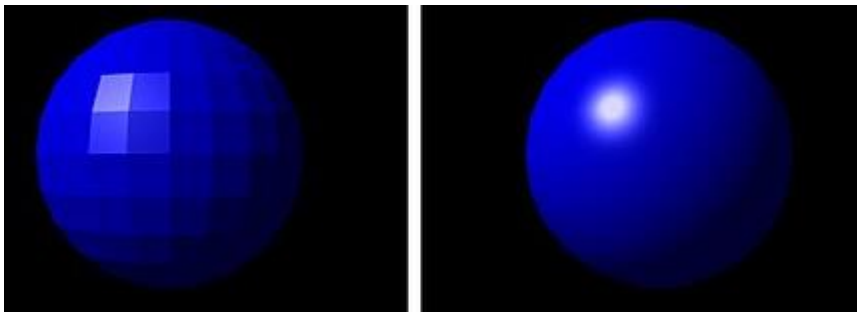
という定義がされています。非常に情報量が多いですね～。なのでぶっちゃけ頂点数は少ない方がいい。1 頂点当たり 48 バイトですね。たいていのフォーマットはこれ以上なのでしゃあないと思いますわ。

用語

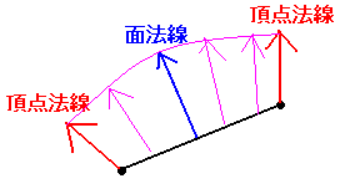
またけったいな用語が出てきたのに気づいておりますでしょうか？
まず法線…ここでは法線ベクトルの事ですが、聞いたことくらいはあると思いますが、

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。



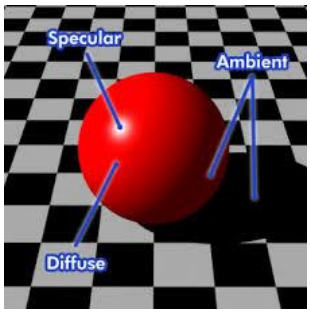
面法線　　頂点法線



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!!!

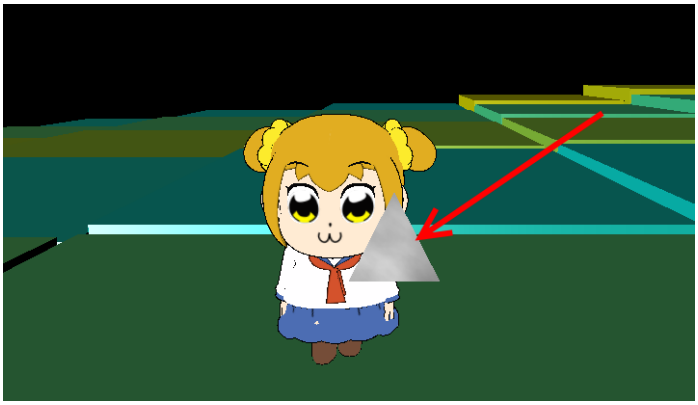
頂点の情報は
頂点座標、法線、ディフューズ色、スペキュラ色、UV、ゴミ
となっておりますので、三角形を作るならば

```
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？

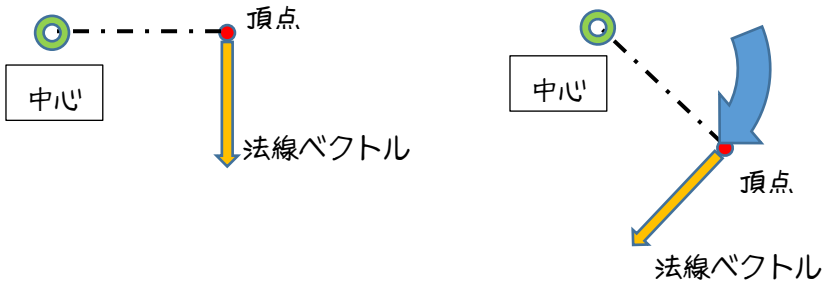
そう…行列同士を乗算するんですよね。ちなみに DxLib では行列の*オペレータは搭載されておりませんので MMult 関数を使用してください。行列同士の乗算を表します。
MGetTranslate
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N16

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。

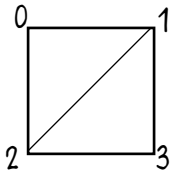


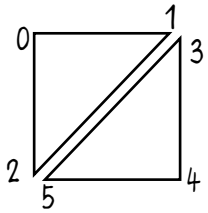
ただし、法線には平行移動成分をかけてはいけません。平行移動成分を抜くには 4x4 行列のうち、左上から 3x3 行列だけをかけてやればよく、そのための関数も用意されており
VTransformSR
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N13
です。その結果

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。

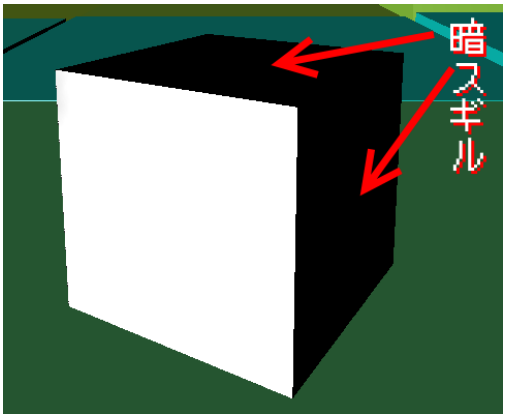
- 点の数は 24 個(6x4)
- インデックスの数は 36 個(6x6)

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

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。

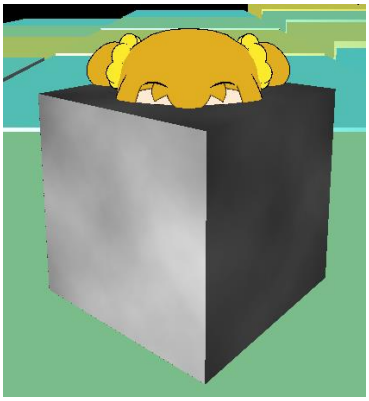
ですが今回はプロシージャルに作った立方体。確かにマテリアルは無いですねぇ…ということで適切な関数はないですかねえ…
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N11
その名も SetMaterialParam ですね。

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。

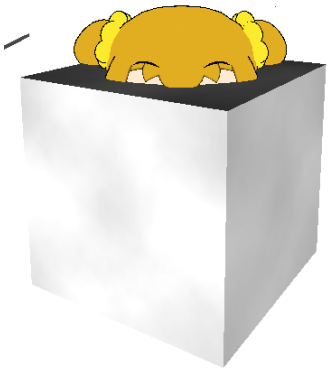
また、Power というのがありますが、これを 0 にしないでください。

```
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

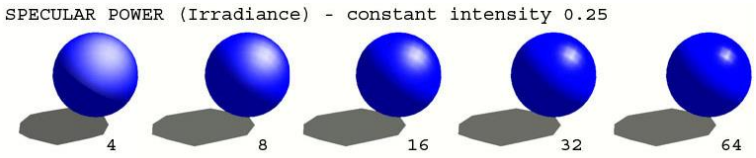
$$H \cdot N$$

合わせて書くとこう

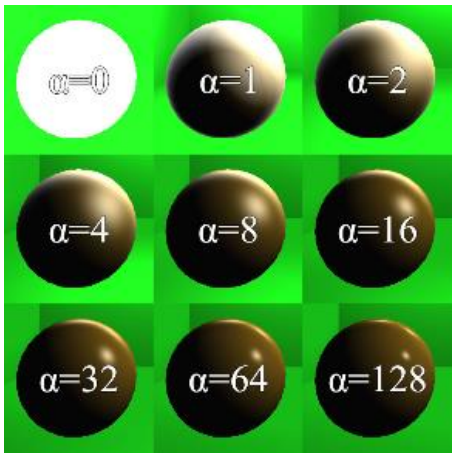
$$(L + E) \cdot N$$

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…

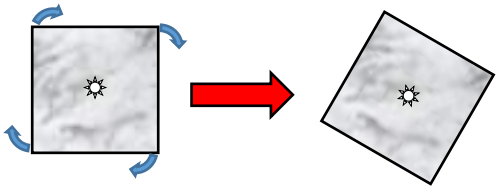
- 状態遷移に switch～case は使わない

- 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
- 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。

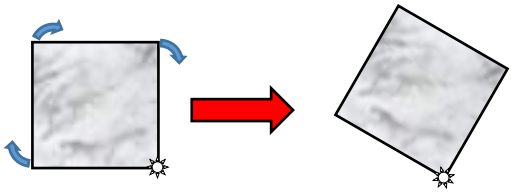
という事で、今一度、今一度自分のコードを見直してみよう

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

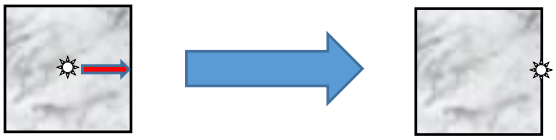


向なら右側。反時計回り方向なら左側へ回転します。
「辺が～」って言うと非常に難しく思えますが「点」と「軸」が分かってれば OK です。もーつ言うと、現在の中心さえ分かれば中心を辺長/2 ぶん移動すれば終わりなので、大した話でもない

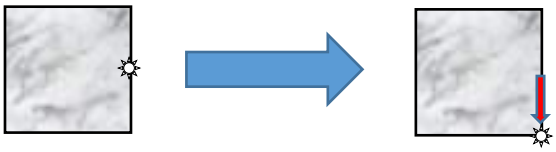
です。

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いてください。

```
template<typename T>
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);
        }
};
```