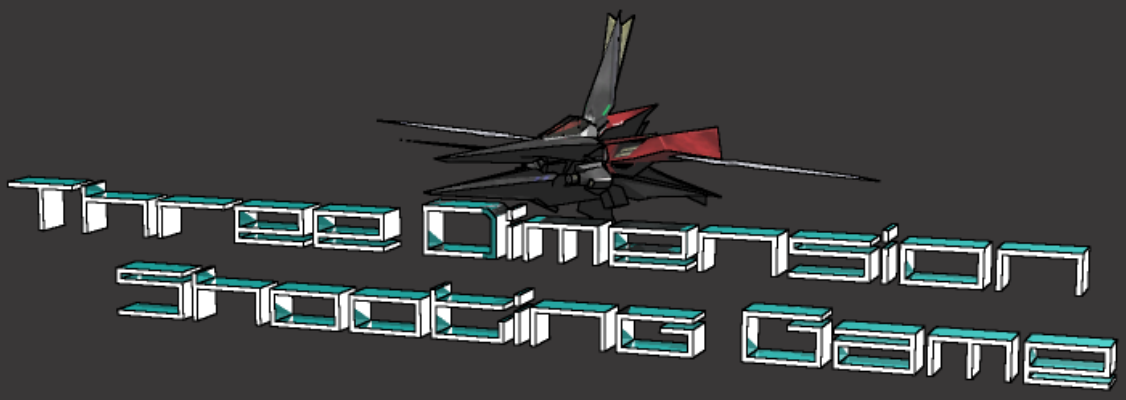



ここここからは3Dシューティングをやっていきます。
おいおまいら!!格闘ゲームが終わってホッとしてるかもしれんがー!!!

悪いがめんどくせーのはここからだ!!とはいえ、後期の地獄の片鱗を見せるだけになると思うんだけどな!!

# 目次

概要 ..... 2
準備 ..... 3
テスト ..... 3
はじめに ..... 11
　今後のスケジュール ..... 11
　数学マジ勉強して ..... 11
　制作と勉強とアイデアと ..... 11
必要なモデルを空間に配置してみる ..... 13
　Model クラスを作ろう ..... 14
　Peripheral クラスを作ろう ..... 18
　Object クラスを作ろう ..... 20
　Player クラスを作ろう ..... 20
　Camera クラスを作ろう ..... 21
　Enemy クラスを作ろう ..... 23
　EnemyFactory クラスを作ろう ..... 24
　　friend ..... 24
　Weapon クラスを作ろう ..... 27
　　下準備①曲げましょう ..... 27
　　下準備②絵を曲げよう ..... 34

見た目改善①真横に来ると潰れる……39
武器らしく動作させる……42
インデックスを使用する……45
段々と消えるようにしてみる……46
スピードを調整する……47
Effect クラスを作ろう……47
下準備(ライブラリの用意:64bit の場合)……47
試しにエフェクト出してみよう……49
ハマる箇所……51
指定の場所にエフェクトを出してみよう……52
ステージクラスを作ろう……56
より「らしく」していこう……67
そろそろ「ロックオン」をきちんと実装しよう……68
照準の表示……69
ロックオン……70
ロックオンしたらロックオンしたと分かるようにしよう……75
ショットも出せるようにしよう……76
ショットとの当たり判定も作っておこう……78
ザコ敵の挙動……79
ベジェーを利用しよう……79
ボス戦……82
中ボス戦について……83
中ボス登場時にスクロールを止める……84
戦車の砲塔等の回転アニメーション……85
敵マップデータ読み込み時の注意点……85
Background クラスを作ろう……86
ブラッシュアップ……86
UI……86
課題提出……89

# 概要

ひとまず今回は、3D でやっていきます。画面の解像度も 1280×768 です。3D ですから。
それっぽいモデルは見つけてきましたが、何しろ3D モデルはスプライトに比べて素材が探しづらいため、色々なもので代用していきます。

# 準備

ひとまずは、プロジェクトを作って、名前を shooting3d とかにして、ソースコードを追加して、ウィンドウを表示させてみましょう。

最初の部分などが分からなくなったら、前のテキストを参照しながら作ってみてください。もし、設計がキレイにできていれば、前のコードはそれなりに流用できるので楽に移行できるとは思います。

ひとまず最初の目標は可能な限り早くウィンドウを表示することです。


この時間の目標はこの画面を表示することですね。今回は解像度を高めにしておりますので、このようなサイズになっております(1280×720)。ですが、作業しづらいという人…そもそも画面に入りきれていないという人は、比率はそのままで解像度を下げていただいて構いません。

今回お世話になるのは DxLib の関数の中でも…3D の関数となります。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html
もちろん 2D 時代に使った関数も使っていきますので、そこはお世話になると思います。

# テスト

ひとまずは 3D 表示のテストを行います。

DxLib の 2D では LoadGraph～DrawGraph～DeleteGraph の流れがあったように、3D では

MV1LoadModel～MV1DrawModel～MV1DeleteModel の流れがあります。

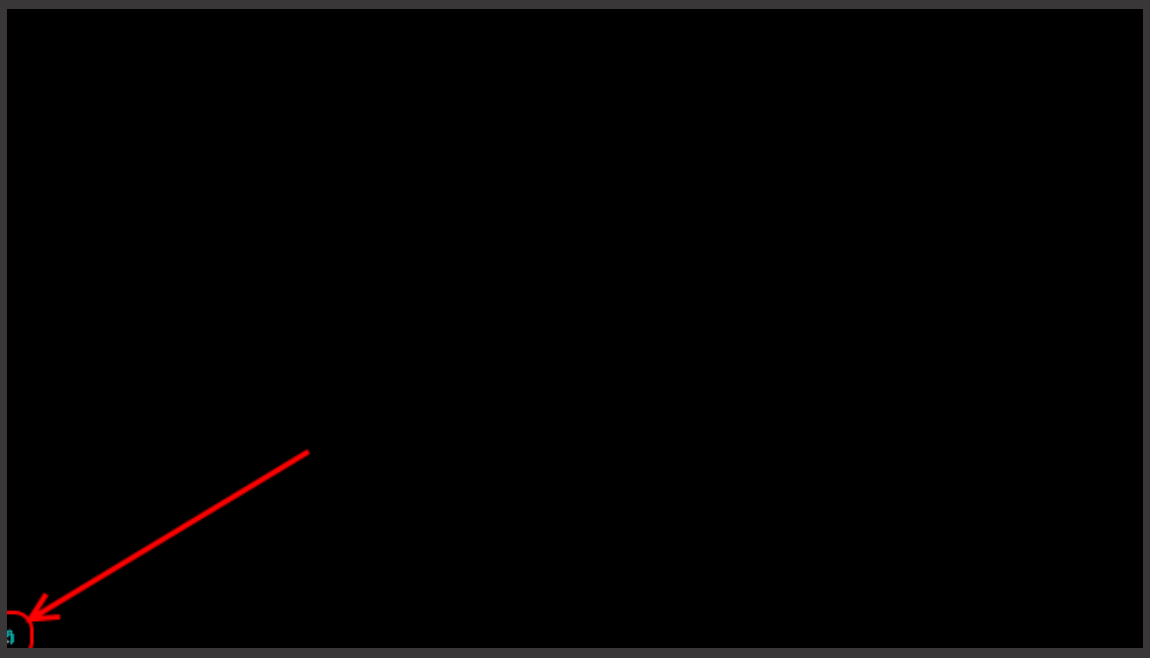
ちょっとテストのために PMX モデルを読み込んでみましょう。

gakuseigamero¥rkawano¥3D シューティング(DxLib)¥test¥mikudayo

のファイルを自分の所に持ってきて、

```
_testModel=MV1LoadModel(_T("model/mikudayo/mikudayo-3_6.pmx"));
(中略)
DxLib::MV1DrawModel(_testModel);
```

の流れで表示させてみてください。え？表示されない!?オカシイなあ…？
よく見てくださいよ。

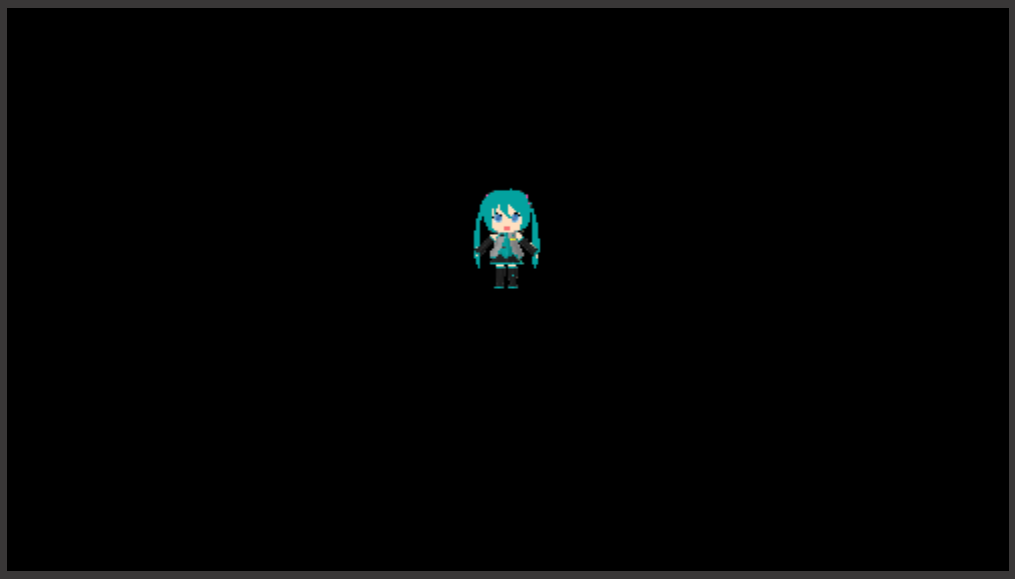


ん!?

何でこんなにちっちゃいの!?
キャラクターはデフォルトでは 0,0,0 の位置にいるはずなので、画面の中心に来るはずです。
思想がよくわからんのですが、どうも DxLib のカメラデフォルトは随分右上に位置している

ようです。
ということで、カメラの座標をずらしてみましょう。カメラの座標を変えるには
SetCameraPositionAndTarget_UpVecY
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R12N2
という関数を使用します。
VECTOR CameraPos = { 0,0,-100.0f};
SetCameraPositionAndTarget_UpVecY(CameraPos, VGet(0.0f, 0.0f, 0.0f));
こんな風に記述してください。

そうすると



だいぶ見えるレベルになってきました。

ところがまだまだ遠い気がしますので、カメラをキーボード操作で動かせるようにしてみましょう。

え？
それは自分で考えてね？ヒントはわざわざカメラの座標を CameraPos なんていう設定にしてることだよ？
上下左右前後に動くようにしてみよう。前後は1とか2を割り当てておいてください。もしくはパッド持ってる人はトリガーとかそういうの使いましょう。

ところがある程度近づくと…

割と大変なことになってしまいます。
これはですね、クリッピングというしくみのせいで、3D をやっていく時は、2D の時とは比べ物にならない知識が必要で、それを知らないとこういう ToLove るに見舞われます。

こまけぇ事は後で説明しますが、ニアファー設定をもっと近くまで有効にしてあげます。
SetCameraNearFar(5.0f, 1000.0f);
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R12N1

とでもしてやれば、


と、かなり近づいてもミクさんを拝めるわけです。

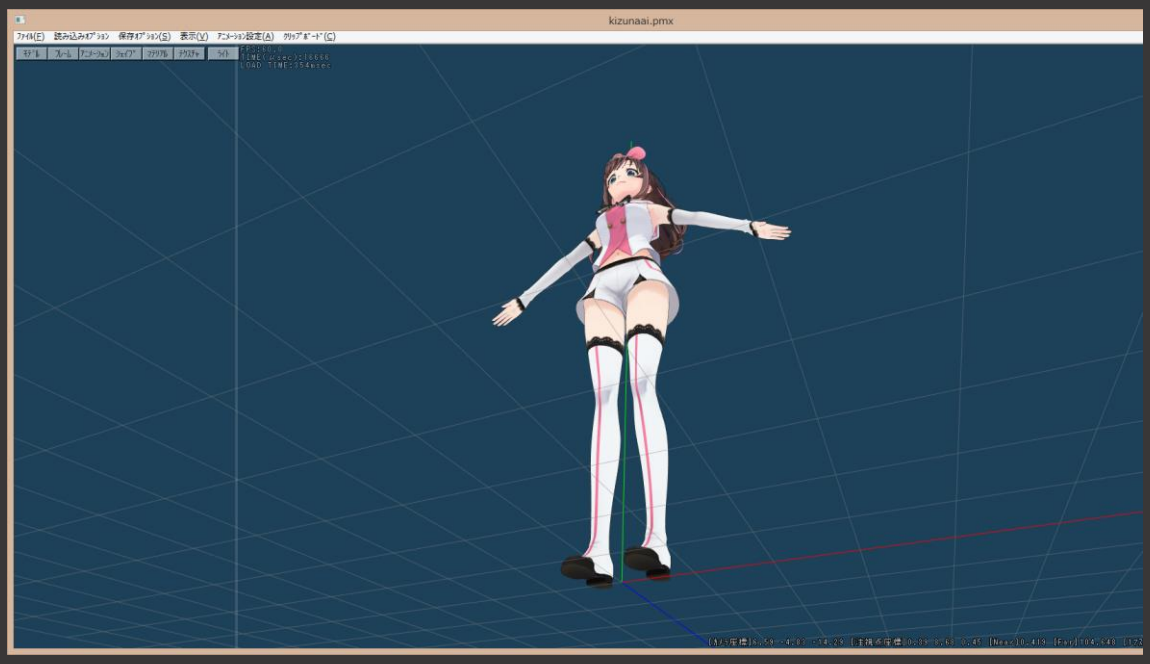
ちなみに

「そういうことならニアファーをもっと広げれば良いんじゃねーの？」って思うかもしれませんが、これは奥行方向の解像度みたいなもんで、広げれば広げるほど奥行方向の精度が悪くなるため、ゲームによってここは調整すべきところなわけです。

ちなみにこういう画像をやると、「俺はミクダヨーなんて嫌だ!!キズナアイを表示するぞ」と思うかもしれませんが、やめなされ……クラッシュします。

http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R1N1
に書いてますが…
「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 )」
らしく、PMX はぶっちゃけ保証対象外のようです(';ω;')

まぁ、ミクダヨーさんも PMX なんですが、大丈夫みたいなんですよね。
ちなみに DxLib を解凍したフォルダに
Tools ってのがあって、その中に DxLibModelViewer ってのがあります。コイツに読み込むと何故かキズナアイちゃんは読み込めます。




くそっ……スカートじゃねーのか!!!

よく分からんけど、DxLib だと失敗するんですよね…何ででしょ。ということでミクダヨーさんでやっておきましょう。

んで、今回のこのミクダヨーモデルは陰影がついてないっぽい(DxLib が対応してないっぽい)ので、陰影がつくミクダヨーを mikudayo2 にあげています。

まぁ、今日の所はミクダヨーさんを回転させたりして遊んであげてください。

いや、ミクダヨーさんじゃなくてもいいんですけどね探せばけもフレとかもありますし。ただ今回のキズナアイのように、対応していないものもあるみたいですので、ご注意ください。

ちょっと背景が無いと分かりづらいかもしれませんね。

もちろん、今まで使用していた LoadGraph～DrawGraph は使用できますので、それを使って背景を表示し、その後に 3D モデルを描画します。そうすると…

こんな感じですね。
ちょっと浮かんじゃってるんで、サーバルちゃんを下におろしてあげましょう。

キャラクターはデフォルトでは、空間内の 0,0,0 の座標に位置しています。これを下ろすには Y 軸マイナス方向に動かして上げる必要があります。

キャラクターの座標を決定するには…MV1SetPosition を使用します。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R3N2
MV1 ってついたらモデルに対する操作だと思ってください。

なお、わかると思いますが、DxLib3D のベクトルは VECTOR という構造体がありますので、それを使用します。

また VGet(X,Y,Z)で、直接入力ができます。

その関数を使用してだいたい地面的なところまでおろしてください。

ちなみに回転もありまして、
MV1SetRotationXYZ で回転します。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R3N6
Y 軸中心に回転させてみてください。

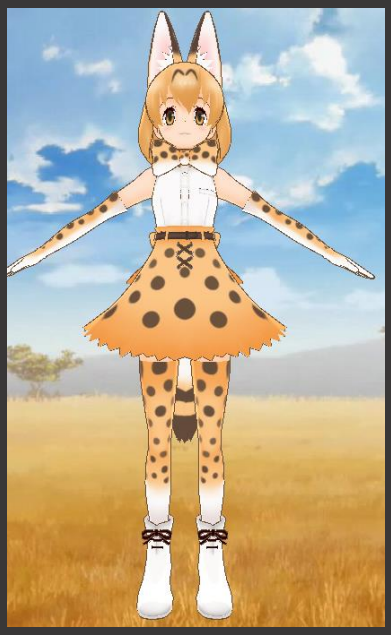
で、ここで勘の良い人は MV1SetScale で拡大縮小スルことがわかると思いますが、この時に注意しないと

こうなります。拡大縮小のアルゴリズムによっては、拡大時に輪郭線まで太くなる現象が起きます。

そういった場合は

```
int materialNum=MV1GetMaterialNum(ハンドル)
for(int i=0i<materialNum;++i){
        MV1SetMaterialOutLineDotWIdth(ハンドル,i,ドットサイズ);
}
```

で調整してください。そうすれば……

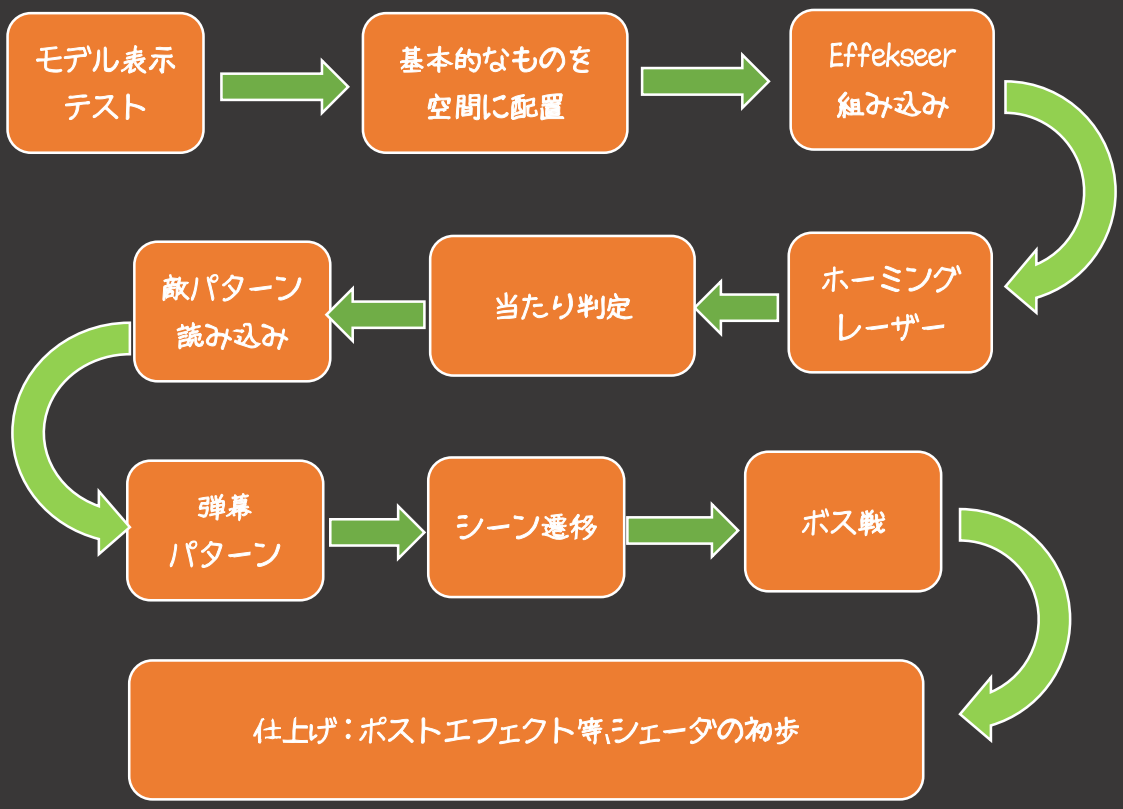


かわいいサーバルちゃんだ!やったね。

# はじめに

## 今後のスケジュール

さて、今回のこれはあくまでも導入部分。こっから7月中旬を目処にレイストーム的なものを作っていきましょう。大雑把なスケジュールはこんな感じ。既にモデル表示テストは終わっていますね。




で、

## 数学マジ勉強して

3D は数学だらけで、更に言うとシューティングゲームも数学だらけです。「数学が苦手だー」って人は今のうちに数学をおさらいしておいてください。特に、ベクトルと行列。3D の時はマジで必要だから。

ゲームの数学を勉強するときにオススメなのが「ゲームつくろー」ってサイトと「ゲームを動かす数学•物理」って本ですね。そして勉強する時は取捨選択してもいいんですが、ちょっと話しておきたいことがあります。

## 制作と勉強とアイデアと

例えば僕なんかは実装のアイディアをぽんぽん言ってますが、正直な話「その場の思いつき」

です。これって頭が固い人なんかは「学生に思いつきを教えるなんてけしからーん!!」とか言うんですけど、「バカか、ゲームづくりなんて大半が思いつきじゃい。正解しか追っかけなかったら全てのゲームがスーパーマリオにならあ!!」

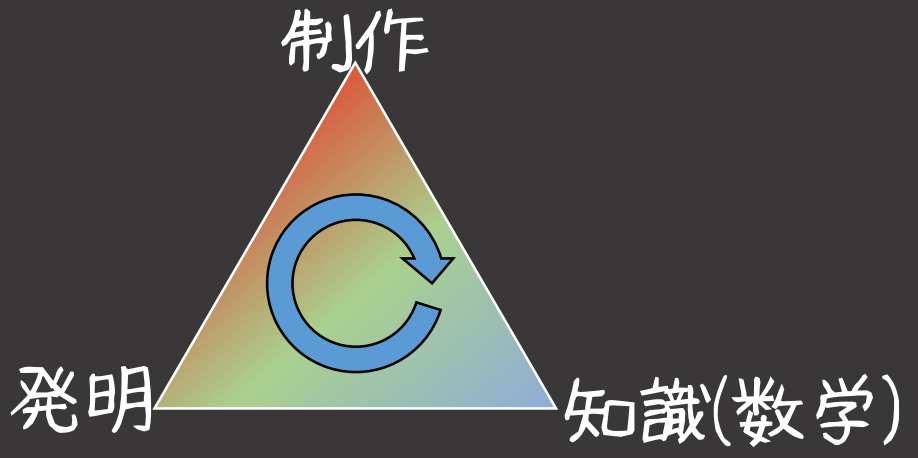
って事です。君たちがプロのゲームプログラマになった時は誰も正解なんて知らないことだらけです。

で、ぽんぽんアイデアを思いつくコツとしては、アホみたいに知識を頭のなかに入れとくことです。とは言え優先順位は

## 作ること＞知識を得ること

です。これだけは忘れないでほしいのですが、前にも言ったように、知識があれば、欲しい情報に最短でアクセスできることもあるし、知識を組み合わせて新しい発想を得ることができるからです。

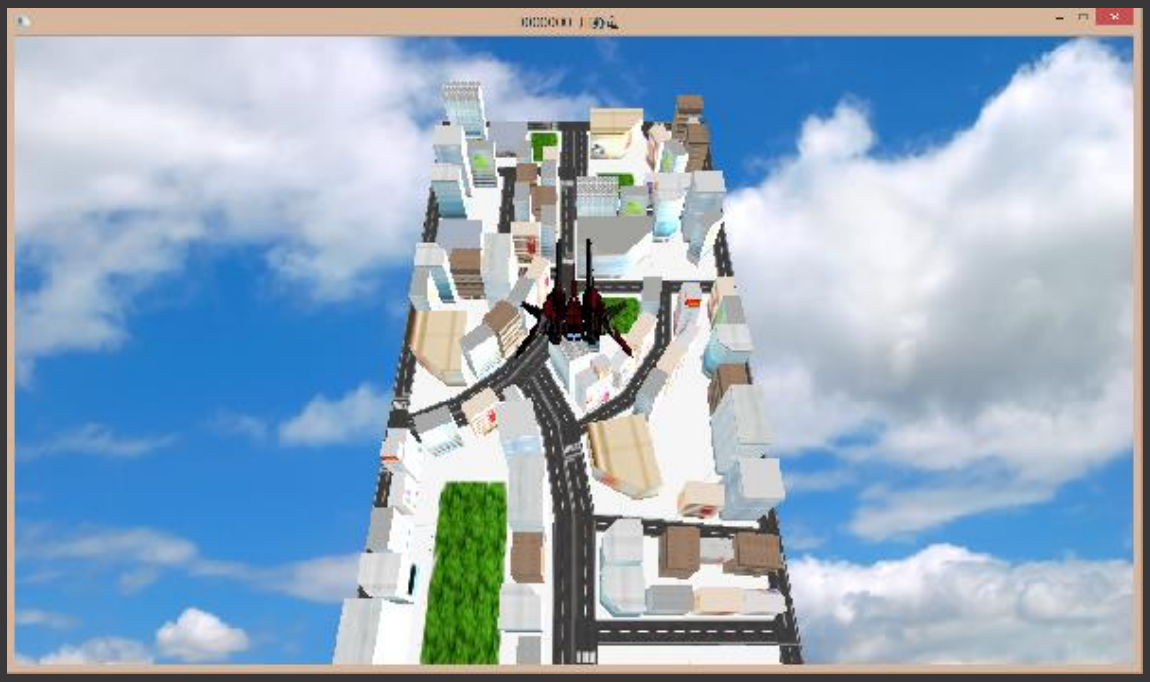
というか正確には、知識を詰め込むだけでは新しい発想は出てきません。頭にそれなりの知識が入っている状態でモノを作る時に発想が出てきます。実際に作っていかないと、実装に必要なものは分かりません、そして「必要は発明の母」であり、それによって実装が加速していくわけです。



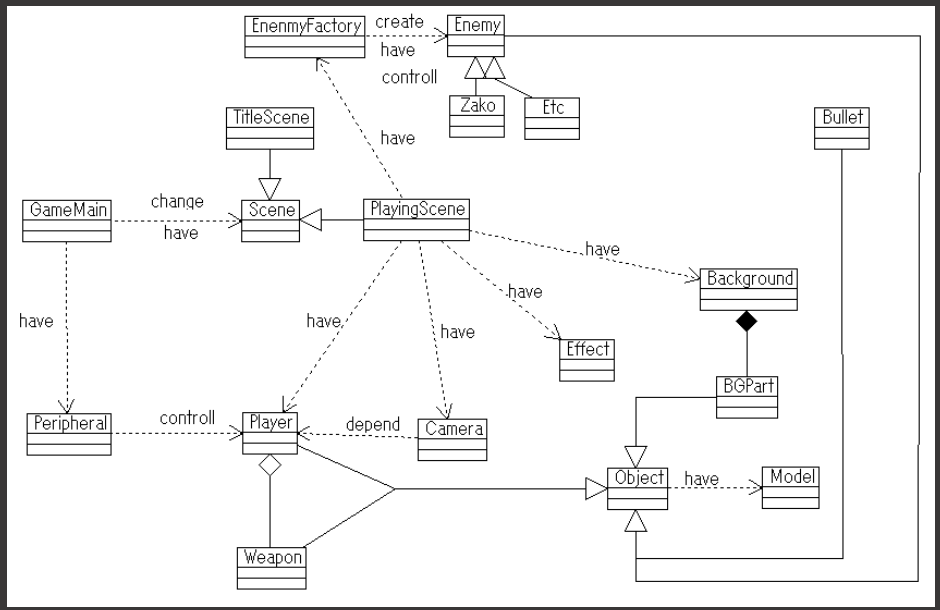

もうホント、こっから先、サイコーに楽しいけど、サイコーに死ねる時間が君たちを待っているぞ。
あと、一回聞いてわからんかった人もたぶん、このクラスに居る以上はポテンシャルはあるはずなので、時間を置いてもう一度トライしてみよう。

# 必要なモデルを空間に配置してみる

とりあえず、既にテスト段階でやりましたが、3D のモデルを空間上に配置しましょう。
あ、ちょっとその前に、どの道後でシーン分割やっていくので、今の内に、画面配置のうちに
PlayingScene か何かに入れておきましょう。
Scene クラスは流用して改変して構いません。ひとまずそこで自機と背景を表示しましょう。



ちょっと見えづらいですが、そこは今後調整していきましょう。
後々使いやすくするために、ちょっとクラス設計やっときましょう。
自機は一体だけなので、前と同じく Player で良いですが、全体的に大雑把に



こんな感じで計画しています。…もうこの時点で多くね？と感じるかもしれませんが、心配しなくても、もっと増えると思います。
第一の目標は
GameMain や PlayingScene から直接に DxLib の 3D 系 API を呼ばないようにしよう。ということを第一の目標とします。

## Model クラスを作ろう

現在の座標、向き、サイズを内部に保持し、モデルに対して必要な、便利なメソッドを搭載する。と、最終目標はこんな感じだが、とりあえず現在のモデルデータ(int 型のハンドル)を内包した形で作っていこう。
内部に持っておくのは

- ハンドル
- 現在の座標
- 現在の向き(ベクトル)
- 今の向き中心の回転量
- 現在の移動速度
- 拡大率(拡大縮小がある場合)

で、これをどうこうする関数は

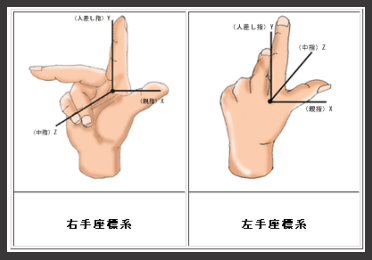
- Draw:モデルを表示する
- Update:モデルの座標を更新する
- SetPosition:場所を設定する
- LookAt:モデルの Z 軸を特定のベクトル方向に向かせる
- SetAngle:回転量を設定する
- SetVelocity:移動速度を設定する
- SetScale:拡大率を設定する

とでもしておきましょう。ひとまず今のプレイヤーの表示を Model クラスにどんどん置き換えてみましょう。

```cpp
#include"Geometry.h"

class Model
{
private:
	Vector3 _position;//現在座標
	Vector3 _direction;//現在の向きベクトル
	Vector3 _velocity;//現在の速度
	float _angle;//現在の、向きベクトルを中心とした回転量
	float _scale;//現在の拡大縮小率
public:
	Model();
	~Model();
	///描画する
	void Draw();

	///毎フレーム処理する
	void Update();

	///場所を設定
	void SetPosition(Position3& pos);

	///適当なところ向く
	void LookAt(Position3& target);

	///任意軸回転
	void SetAngle(float angle);

	///速度設定
	void SetVelocity(Vector3 vel);

	///スケーリング設定
	void SetScale(float scale);
};
```

ヘッダ部はこんな感じで作りました。実装部は、できるところまででいいんで自分で実装してみてください。
元のようにプレイヤーが表示されればオッケーです。
あと、ちなみに言っておくと、MMD のモデルを持ってくると、前後が入れ替わります。これは何でかというと「系」が違うからです。ところで右手系と左手系って知ってますか？

これは 3D 世界の派閥みたいなもんなんですけど、フレミングの法則みたいな感じで、親指を X とし、人差し指を Y としたときの中指の向きを Z とする。そうした場合に、右手と左手で、Z




の向きが逆方向に向いちゃうんですよね？
自分の指で試してみて？

で、残念ながら MMD は右手系。DxLib は左手系なんですよね。なので…何の細工もせずに表示

をしてしまうと前後がひっくり返ります。
つまり、お尻を向けちゃうわけね。なので、元のデータが右手系だった場合は(大抵のモデリングソフトは右手系なんよねぇ…)180°回転させておこう。
ここで、(1,1,-1)倍すれば良いんじゃね？と思った君。鋭いな。でもそれは問題があるのだ。
頂点の座標を反転するということは、「裏返す」ということを意味し…幸いMMDは両側に面が貼られて入るのだが、結局法線が逆を向いてしまい、正しくシェーディングされなくなってしまうのである。


この例だと真っ黒になります。
とはいえ系を変更する関数もないため、現状はY軸方向に180°回転させておくのが良いでしょう。このモデルの場合、違和感はないと思います。
この辺は結構トラブルの元にもなりますので、今回は問題ないですが、3Dにはそういうことがつきものであるということは認識しておいてください。

…まぁ、最初は後ろ向いといてもいいです。

ちなみにこのように

```
Model::Model(const TCHAR* path) : _velocity(), _position(), _handle(0)
{
	_direction = Vector3(0, 0, 1);
	_handle=MV1LoadModel(path);
}
```

```
Model::~Model()
{
	MV1DeleteModel(_handle);
}
```
デストラクタで DeleteModel しとけば、リソース解放し忘れがなくなるというスグレモノなのよ。というほどでもないけどね。

あと、たぶん疑問に思っているのが LookAt 関数なんですが、これはモデルの向きを特定の方向に向けたい時に使います。いちいち角度測って～なんて、3D のときにやってたら効率が悪くて仕方ない(ただし「角度制限」の時は角度はかるよ)

やり方は2つくらいあって
MGetRotVec2 関数
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N21
を使用して、回転行列を取得し、それをモデルの行列に適用する。

これもありだ。

MV1SetRotationZYAxis
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R3N8
だが、これを使用すれば任意の軸にモデルを向かせることができる。
これを使用すれば敵をこっちに突っ込ませることも可能である。

```
LookAt(float x,float y ,float z) {
	VECTOR vector = { x - _position.x,y - _position.y,z - _position.z };
	MV1SetRotationZYAxis(_handle, vector, VGet(0, 1, 0), 0.0f);
}
```
それでは 180°反転した上で LookAt したい場合はどうすれば良いのだろうか…？そこまでやりたかったらやっぱり行列を使わざるを得ない。


3D プログラミングである以上、避けては通れない…それが行列。そこは覚悟していたと思いますので、容赦なくやっていきます。

とはいえ、行列の中身はコンピュータにさせるので、「行列がどういうものか」分かってたらなんとかなるでしょう。

とりあえず、LookAt する前の行列を MV1GetMatrix で取得しておきます。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R3N12
次にダミーとして Z 軸方向に向いているベクトルを用意します。

これで出来たベクトルに対して MGetRotVec2 を使って
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N21
LookAt 行列を計算します。ここで行列の乗算を行います。行列は乗算することで、座標変換を重ねがけできるのです。とはいえ、DxLib の MATRIX は乗算演算子がオーバーロードされていないため MMult 関数を使用して乗算します。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N25

それを MV1SetMatrix します。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R3N11

とりあえずここまでやれば、回転した上で特定の方向を向かせることも可能です。

```
void
Model::LookAt(float x,float y ,float z) {
	VECTOR vector = { x - _position.x,y - _position.y,z - _position.z };
	MATRIX matOrig = MV1GetMatrix(_handle);
	VECTOR zvec = { 0,0,1 };
	MATRIX lookatmat = MGetRotVec2(zvec, vector);
	MV1SetMatrix(_handle, MMult(matOrig,lookatmat));
}
```

## Peripheral クラスを作ろう

次に入力周りを制御する Peripheral クラスを作ります。これは2D ゲームのときにもある程度作りましたが、GetJoyPadInputState を使用します。
http://dxlib.o.oo7.jp/function/dxfunc_input.html#R5N4
とにかく、キーの状況を独自の情報に変換して持っておくクラスだと思っておきましょう。

シューティングなので反転などは存在しませんが、方向キー情報は8方向情報としてまとめ

て持っておくと良いと思います。
アナログインプットに対応しなければ、8方向デジタルでいいでしょう。
これを現フレームと前フレーム保持しておき、トリガーも実装すればよいでしょう。

で、Player クラスはこのクラスへの参照を持っておきます。でも真の持ち主は PlayingScene です。これは、将来的に「ポーズ」を実装できるようにしたいからです。ポーズ時は状態を「ポーズ状態」にし、Player の入力を受け付けなくするため、Player の持ち物にするのはちょっと良くないからです。

ちなみに、レイストーム、レイフォースは
- 8方向移動
- ショット
- ロックオンレーザー

の入力です。入力に関しては格闘ゲームよりも簡単だと思います。今回それに加えて
- ポーズ

を追加しますので、その入力も書いておきます。

なので InputState 構造体は
```
struct InputState {
	DirectionState direction;//方向キー
	bool shot;//ショットボタン
	bool lockon;//ロックオン
	bool pause;//ポーズ
};
```
こんな感じになるかなあーと思います。

で、これを取得する関数を Peripheral 内に作ります。
まず、Update 関数でパッドの状態を取得し、その時点でのキー入力を
GetCurrentInput();
直前のキー入力を
GetLastInput();
で取得するようにします。
なお、トリガー状態を
GetTrigger()
で取得します。
Trigger はショットを撃つまでは必要ないと思いますがね。

## Object クラスを作ろう

これは様々なキャラクタークラスの大元になるクラスです。プレイヤー、敵、弾などなど。当たり判定を持ち、モデルを内包する。そういうクラスです。なお、抽象クラスとしたいため Update 関数と Draw 関数は純粋仮想関数としておきましょう。

```
#include<memory>
#include<windows.h>
class Model;
class Object
{
protected:
	std::shared_ptr<Model> _model;
public:
	Object(const TCHAR* filepath);
	Object(int handle);
	virtual ~Object();
	virtual void Update() = 0;
	virtual void Draw() = 0;
	const Model& GetModel() const;
};
```

こんな感じですかね～。
これに関しては実装も大したことしなくていいでしょう。コンストラクタでモデルを生成し、GetModel でモデルオブジェクトを返すだけです。こいつは純粋仮想関数持ちなので、さっさと継承先の Player クラスを実装しましょう。

## Player クラスを作ろう

さて、いよいよ Player クラスです。
Player クラスの特徴は…
モデル本体
当たり判定
を持ち、入力系への反応を示すものです。入力に反応し、モデルの表示を行うため
Update 関数
と
Draw 関数

は必須ですね。
ひとまず、Modelに対する操作をPlayerクラスに移し替えましょう。
もちろんObjectから継承してください。
僕はこんな感じのクラスにしました。

```
class Peripheral;
class Player :
	public Object
{
private:
	std::weak_ptr<Peripheral> _peripheral;
public:
	Player(const TCHAR* filename,std::weak_ptr<Peripheral> per);
	~Player();
	void Update();
	void Draw();
	void SetRotation(Vector3& rot);
	void SetPosition(Vector3& vec);
	void SetPosition(float x,float y,float z);
	void Move(float x,float y,float z);
};
```

こんな感じで作っています。だいたい想像つくと思いますので、作っていきましょう。

## Cameraクラスを作ろう

とりあえずまだプレイヤーとは関連付けることは考えずにカメラ周りをカプセル化することを目的にプログラムしていきましょう。
最初に、カメラのタイプを色々と種類考えておきます。

```
enum class CameraType {
	fixed_point,//定点カメラ(シューティング系)
	tracking_target,//ターゲットを決めて追尾(TPSとかこれよな)
	program,//最初からプログラムされたとおりに(3Dアニメツールとか)
	autonomous,//自律
};
```

今回はたぶん定点と追尾になるかと思います。とりあえずは定点で。
で、カメラじしんの座標がまずは必要なので_positionと、あと_targetが必要なのでそれを設定できるようにします。

また、ゲームオブジェクトを追尾するということを考え、_gameobject への弱参照も持っておきます。これは fixed_point の場合は null のままです。

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

///カメラ種別
enum class CameraType {
	fixed_point,//定点カメラ(ターゲットもカメラも固定)
	tracking_target_fix,//ターゲットを決めて追尾(カメラ固定)
	tracking_target_move,//ターゲットを決めて追尾(カメラ移動)
	program,//最初からプログラムされたとおりに
	autonomous,//自動
	last,//最終(使わない。enum 最後判定用)
};
class Object;
class Camera
{
private:
	CameraType _type;
	Position3 _position;
	Position3 _target;
	std::weak_ptr<Object> _gameobject;
public:
	Camera();
	~Camera();
	void SetCameraType(CameraType type);
	CameraType GetCameraType();
	void SetGameObject(std::shared_ptr<Object> object);
	void SetTargetPosition(Vector3& v) { _target = v; }
	void SetPosition(Vector3& v);
	void SetPosition(float x, float y, float z);
	void Update();
};
```

こんな感じで作っておいて…PlayingScene から毎フレーム Update()関数を呼び出します。

Update()関数の中身はこんな感じです。

```
SetCameraPositionAndTarget_UpVecY(VGet(pos.x, pos.y, pos.z), VGet(tgt.x, tgt.y, tgt.z));
```

今のところ、Update にはこれくらいでいいでしょう。

## Enemy クラスを作ろう

敵を出します。
Enemy も Object から継承しましょう。で、Enemy は量産型なので、最初の一匹以外は MV1DuplicateModel で増やしていきます。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R1N2
ですので、後述する EnemyFactory を使用して量産します。EnemyFactory はシングルトン…いや、PlayingScene にひとつだけにしましょう。

ひとまずは一体だけ出現させて、パターン的な動きをさせましょう。

ひとまず画面上方から下方に向かって動くようなのを作りましょう。

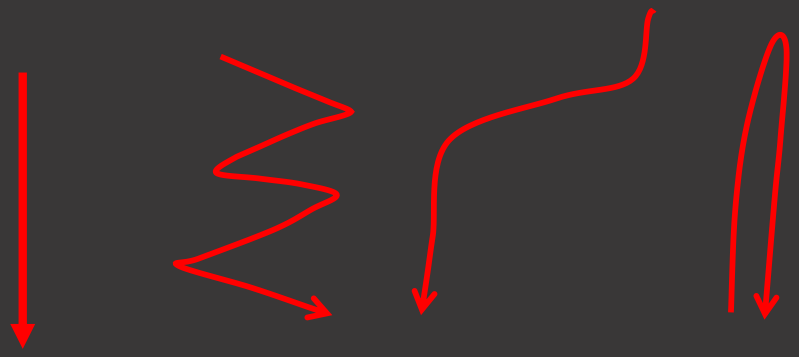
あ、Enemy も継承する予定なのでデストラクタは virtual にしておきましょう。

- ザコ敵は基本的には画面外から襲ってきます
- ボスは定位置です
- 一通りプレイヤーを攻撃する動きをすると基本的には画面外にはけます
- シューティングのザコは完全パターンです
- 中ボス、面ボスなどは AI(プレイヤーの動きを見る)で動くようにします

まぁ、ともかくまずは画面上に表示させないとね。

Enemy クラスにして表示が出来たら動かしましょう。
最初は本当にこの動きでいいです。

とにかくまずは「表示して」「動かして」ください。
慣れてきたら、色々な動きをやってみましょう。状態遷移でやってもいいし、ひたすらフレーム見て、時間で動きの変更をやってもいいです。
余裕があったら



みたいな動きもやってみましょう。一番左は簡単ですし。ジグザグならばサインカーブでもいいし、%演算を利用してもいいでしょう。
その次のような if 文が必要っぽいものは if 文とか状態遷移を使用しても構いません。
最後のは二次曲線か、もしくはこれも%演算でできるでしょう。色々と試して遊んでみましょう。

## EnemyFactory クラスを作ろう

色々な敵を複数生成、操作していくためには、それをまとめるクラスが必要です。これをシングルトンで～みたいに考えるの早いですね。そこでまたワンポイント C＋＋言語文法授業やります。

### friend

C++にしか無い文法の一つで、「friend」ってのがあります。これがかなり特殊なものです。
何に関わるものかというと、アクセス指定に関わるものです。
ある意味恐るべき文法なので、他の言語には使用されていない…そういうキーワードです。

簡単に言うと
「**あなたはお友達だから何を使っても、何を見てもええんやで**」っていうやつです。
正直これは、いやお前それって、友達どころの話ちゃうやろ？っていつも思いますが、そういうやつです。
つまり、誰か他のクラスに対して「フレンド指定」をやっちゃうと、そいつに対しては本人と同じレベルのアクセス権限を与えてしまおうというやつです。

ある意味、private も protected も public も関係なくなるわけで、恐るべき指定子なのです。ですからこの指定子の使用には十分に気をつけてください。濫用はやめましょう。

使い方。

```
class B;
class A{
        friend B;//Bはおともだちだから、隅から隅までイジってね？
        private:
                (中略)
};
```

だいたいワカッタかな？

で、今回はEnemyFactoryの仕様として、newできないようにしておきます。つまりコンストラクタをprivate…にしておきます。

もう一つ言うと、シングルトンの時に使用したstatic Instance()関数も使用しません。さてこれで誰にもインスタンス(実体)を持てなくなりました。どうしましょう…


勘のいいガキにはもうお分かりかと思いますが…そう、EnemyFactoryのコンストラクタをprivateにし、生成不可にした上で、PlayingSceneをfriendとするわけです。

で、

```
class Enemy;
class PlayingScene;

enum class EnemyType{
        insect,//昆虫型
        blogg,//水棲生物(メトロイド)
        manta//マンタ
```

```
};

///敵工場
class EnemeyFactory
{
	//一見シングルトンに見えるが…Instance()関数すら無い
	friend PlayingScene;
private:
	typedef list<shared_ptr<Enemy>> EnemyList_t;
	EnemyList_t _enemyList;
	EnemeyFactory(weak_ptr<Player> p);
public:
	~EnemeyFactory();
	weak_ptr<Enemy> Create(EnemyType et);
	void Update();
	EnemyList_t& EnemyList();
};
```

このように作ります。内部に std::list を持つつもりです(生成と破棄を管理するため)。
つまり、Create して、Enemy のポインタを返すと同時に、じしんのリストにも保持しておくというわけです。

これで「全ての敵のアップデート」を行うことができ、管理を全てこの EnemyFactory に任せることができます。

で、敵のモデルについては予めこのクラスのコンストラクタでロードしておきます。
そして Create の段階で MV1DuplicateModel を使用します。

で、一旦ハンドルを敵タイプの配列に放り込むことになりますが、後々の事を考えて、僕は連想配列(つまり map)にしておきます。

EnemyFactory のコンストラクタにて想定される敵キャラは全部ロードして連想配列に入れておきます。

```
_handleTable[EnemyType::insect] = MV1LoadModel(_T("model/enemy/insect/insect.mv1"));
_handleTable[EnemyType::blogg] = MV1LoadModel(_T("model/enemy/blogg/blogg.mv1"));
_handleTable[EnemyType::manta] = MV1LoadModel(_T("model/enemy/manta/manta2.mv1"));
```

で、実際に呼び出す時は

```
std::weak_ptr<Enemy>
EnemyFactory::Create(EnemyType et) {
	_enemyList.push_back(shared_ptr<Enemy>(new Enemy(_handleTable[et], _player)));
}
```
こんな感じで呼び出します。
なお、Enemy から継承した別のクラスなどにしたければ
```
std::weak_ptr<Enemy>
EnemyFactory::Create(EnemyType et) {
	switch(et){
	case EnemyType::blogg:
		_enemyList.push_back(shared_ptr<Enemy>(new Blogg(_handleTable[et], _player)));
		break;
	default:
		_enemyList.push_back(shared_ptr<Enemy>(new Enemy(_handleTable[et], _player)));
	}

	return _enemyList.back();
}
```
などとします。

あとはファクトリの Update と Draw をそれぞれ作って、
```
EnemyFactory::Update() {
	for (auto& enemy : _enemyList) {
		enemy->Update();
	}
}
```
みたいにしとけば良いでしょう。

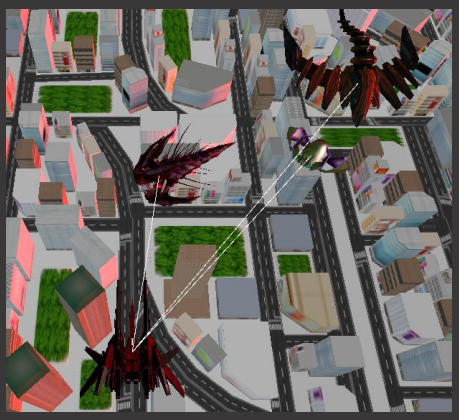
# Weapon クラスを作ろう

さて、いよいよ Weapon です。

## 下準備①曲げましょう

レイストームの醍醐味はレーザーですから、それの準備をしましょう。


これな。
まず軌道を実験的に考えてみましょう。実験なので、見た目は問いません。DrawLine3D を使用して画面上に線を描画します。
例えば
DrawLine3D(VGet(ppos.x,ppos.y,ppos.z),VGet(epos.x,epos.y,epos.z),0xffffffff);
のように書けば、自機から敵に向かって線が描画されますので



このように描画されます。しかし当然ながら、一直線なので…レーザーとしてはある意味正しいのですが、格好良くはありません。

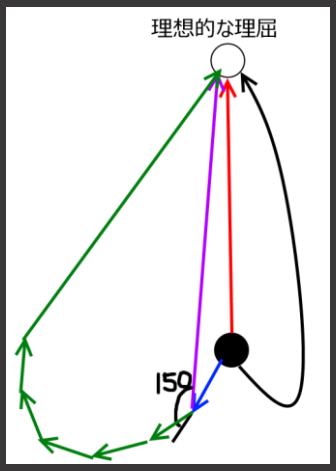
レイストームをよ～～く観察すると、後方にランダムな角度でぶわーっとレーザーを飛ばして、そこから少しずつターゲットの向きに補正をかけていっているのが、わかるだろう？

まずは最も簡単なやり方からやってみましょう。

基本的には後方に飛ばすという時点で、敵の方向を向いていません。手順としては
①レーザーの向きベクトルと、レーザーの先頭から敵機に向かうベクトルとの内積をとる
②acos で①の内積から角度を計算
③角度②が「制限角」を超えていれば、ベクトル補正をその角度にする

こういう手順を取ります。

図にするとこんな感じ




あと

$$\vec{a}\cdot\vec{b}=|\vec{a}||\vec{b}|\cos\theta$$

を思い出してくれ
ここから

$$\frac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|}=\cos\theta$$

cosθが求まるため、それにacosを使用することで、角度を得ることができる。そして次ベクトルの補正だが、これはひとまずはY軸中心として考えよう。実際は2つのベクトルの外積によって作られる平面における補正をするべきだが、今はあくまでもお試しなので、XZ平面上でY軸中心回転補正で良い。
つまり
float angle=acos(dot(a,b)/normal(a)*normal(b));
angle=min(angle,制限角);
こんな感じで良い。

で、この制限角度によりちょっと進んだ座標からまた同じことを行う…こういう時にはテクニック的には何をやったら良いのだろう？

そう…再帰である。

再帰関数を利用するのである。再帰関数は下手をするとスタックオーバーフローを引き起こすため、計算上限はつけておこう。

```
void
PlayingScene::DrawHormingLaser(VECTOR pos, VECTOR initvec, VECTOR epos, int limit) {
	VECTOR pos2 = VAdd(pos, initvec);
	VECTOR vec2 = VSub(epos, pos2);
	DrawLine3D(pos, pos2, 0xffffffff);//ライン描画
	if (--limit <= 0) {//スタックオーバーフローよけ
		return;
	}
	float cosine = VDot(VNorm(initvec), VNorm(vec2));
	float angle = acos(cosine);
	angle = min(angle, DX_PI / 6.0f);

	auto m = MGetRotY(angle);
	vec2 = VTransformSR(initvec, m);
	DrawHormingLaser(pos2, vec2,  epos, limit);
}
```

さぁこれでホーミングになるかな？

!?

あれれー？オカシイなあ？

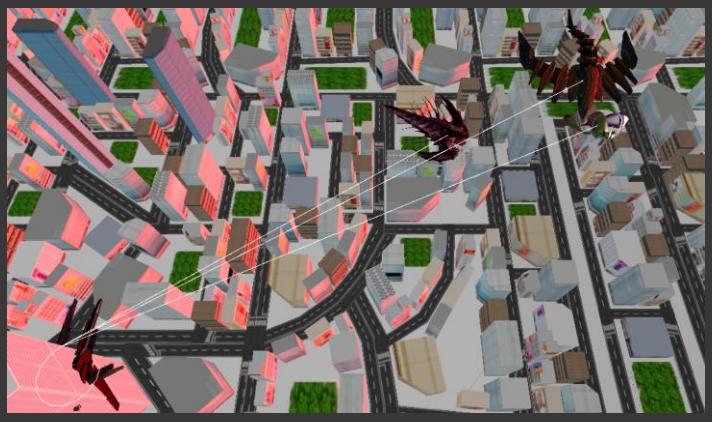
はい、理由としては角度が制限角度以内になったら計算を打ち切らないといけませんね？と

いうわけで打ち切るように変更すると…

```
if (angle < DX_PI / 8.0f) {
        DrawLine3D(pos2, epos, 0xffffffff);
        return;
}
angle = DX_PI / 8.0f;
```

とすると

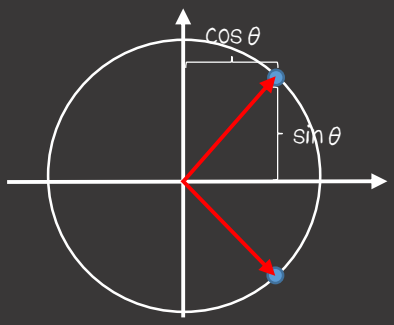
このように正解に近そうな感じにはなる。ところが、この組み方だと



左回りにしかならない。

右回りのほうが近いにも関わらず、だ。

これを改善するには内積とともに外積を活用する必要があります。sin と cos の定義を思い出してほしいのですが




図のように、元の軸(上の例だとX軸)から見て右側にあるのか左側にあるのかの区別がcosではつかないんですよね。

cos(-θ) = cosθ

sin(-θ) = -sinθ

こういうことですから

それではsinを…せめて左右判定を出すには…どうしたら良いのでしょうか？


「外積」を使います。外積の式は

a(ax,ay,az)×b(bx,by,bz)=(ay*bz-az*by, az*bx-ax*bz, ax*by-ay*bx)

こんな式です。当然DxLibには関数が用意されているのでそれを使用します。

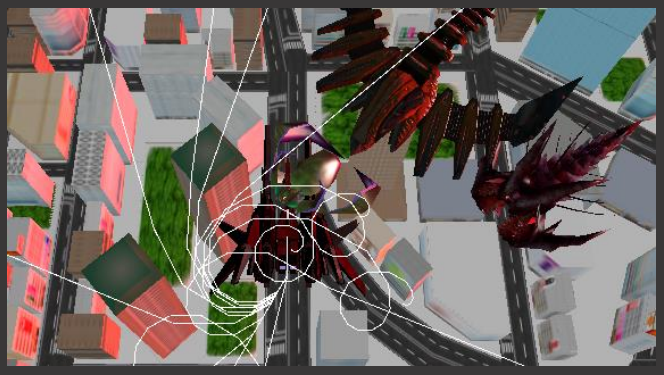
cross = VCross( vec1 , vec2 );

これが外積なのですが、上の式をみてお分かりのように、戻ってくる答えは「ベクトル」であって「スカラー値」ではありません。

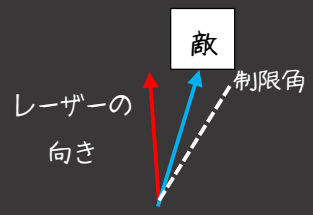
なので

cross.yの符号を見て右回り左回りを判定しようとしたのですが……かなり深い問題が発覚しました。符号がどうこうではなく、Y軸中心に回転しようとするのがそもそもの間違いだったようです。

どういう事かというと、2D のゲームではなく 3D のゲームですので、もし方向角度の補正が必要なくても、縦方向で既に角度がついているため、補正をしようとします。その結果として



近くに来た敵に当たるべき軌道が、さらに回転し、図のようにクセのついたバネみたいな感じになります。つまり




図のように真上から見たら制限角以内にあったとしても、上下方向を考慮に入れると制限角度を超えていたわけです。

これをきちんと判断するには、Y 軸とかではなく、2つのベクトルがキレイに乗っかる平面上で角度判定をする必要があります。

ここまではいいかな？ つまり2つのベクトルの内積から分かる角度は2つのベクトル双方が乗っかる平面における角度を測っていたのに、実際に回転する平面は x-z 平面だったわけ。

ところで平面と法線ベクトルってのは密接に関係していて
平面の方程式は一般的に
ax+by+cz+d=0
で表されるのですが、この式は
ベクトル(a,b,c)を法線ベクトルとする平面で d は 0 からのオフセットを表す。と思っておいてください。

まぁ、厳密なことはともかく、そういうもんだと思ってください。なお、この場合の法線ベクトルは正規化されているものとします。

ともかく回転させるならば、2つのベクトルから外積を計算し、その軸を中心に回転補正をか

けつつレーザーを描画していきたいと思います。

で、DxLib には任意軸回りの回転をする関数がありますので、それを使用したらいいわけです。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R11N20
その名も MGetRotAxis
ここに軸を入れて、内積で取得した回転角度を適用すれば良いわけです。

つまり
```
VECTOR cross=VCross(VNorm(initvec), VNorm(vec2));
cross=VNorm(cross);//外積取る→軸である

//その軸に対しての回転を行う
float nextangle=DX_PI / 6.f;
auto m=MGetRotAxis(cross,nextangle);//外積から取った軸での回転行列
vec2 = VTransformSR(initvec,m);//回転後ベクトルを取得
vec2 = VScale(VNorm(vec2),伸び率);//ベクトル伸縮

//再帰
DrawHormingLaser(pos2,vec2,epos,limit);
```

こんな感じですね。

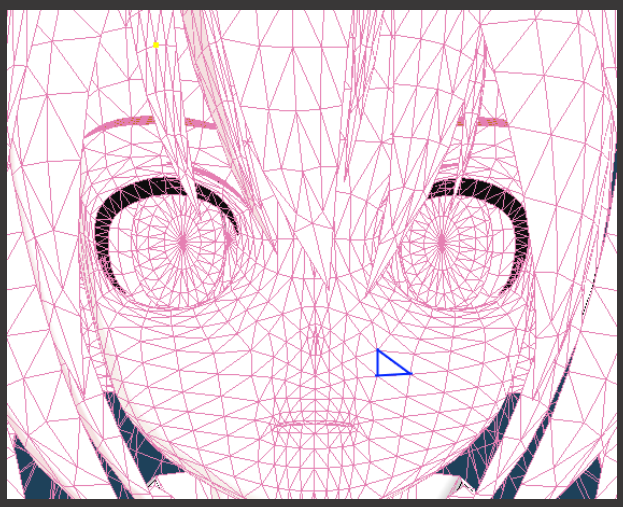
## 下準備②絵を曲げよう

この辺からチョイと難しくなってきます…っていうか、3D の話になってきます。
色々と知っておく必要はあります。

- ポリゴン(三角形)
- 表面材質
- UV
- 非ライティング
- 頂点配列

これらは 2D のときには考慮しなくて良いものでしたが、3D となると無視できなくなってきます。ていうか必須です。

3D の構成要素は最もプリミティブなものが三角形です。どんなに可愛らしいモデルでも
三角形の集合体です。



そこはお分かりいただけるだろうか？夢を壊すようで申し訳ないが、サーバルちゃんもキズナアイちゃんも、三角形の集合体にすぎないのだ。本当に申し訳ない。

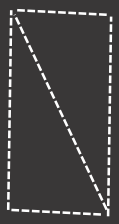
で、今回はじめにレーザー線を DrawLine3D で書いてもらいましたが、あれは無駄じゃなくて、こういうことを考えていました。

こういう線に対して、終点と始点から真横に触手(頂点)を伸ばします。

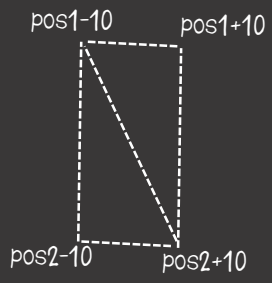
で、その伸びた触手の4点から四角形を構成します。

で、それで終わりじゃなくて、三角形に分割します。そうすると

このような三角形2つを構成すれば良いことが分かります。

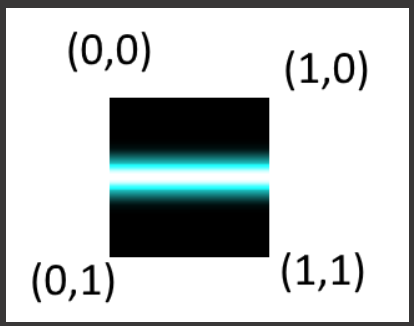


で、今回はちょっと簡単に X のプラスマイナス方向に 10 くらい伸ばしてこの四角形を作ります。つまり

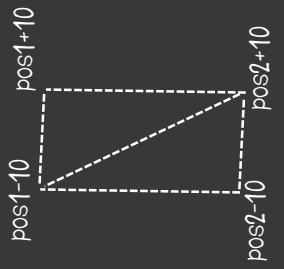



こんな感じで構成します。

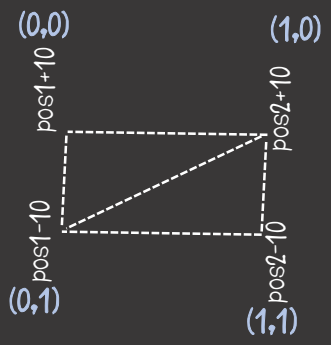
ここまではまだマシな話なのですが、もう一つややこしいのは UV の話です。
UV は




このように、画像の左上を 0,0 とし、画像の右下を 1,1 とし、その数値を各頂点に設定することで、ポリゴンに絵を貼り付けることができる。そういうものです。この場合の UV ですと、先程の図をちょっと回転させて考えたほうがわかりやすいでしょう。




ここに UV を割り当てる。




こんな感じになることは分かりますね？ただ、UV を貼り付けることができても表面材質が不適切ならばきちんと表示されません。注意しましょう。
なお、ここで言う表面材質とは「ディフューズ」「アンビエント」「スペキュラ」「エミッシブ」のことで、DxLib においては「ディフューズ」と「スペキュラ」しか使いません。

ちなみに今回のこのポリゴンに陰影処理は必要ないので、スペキュラも必要ないです。
つまり

```
DxLib::VERTEX3D v[6]={};
for(auto& vertex:v){
	vertex.dif=GetColorU8(255,255,255,255);//色を真っ白にしておく
}
```

これ、色を真っ白にしてるのは意味があって、真っ白以外の色にするとテクスチャの色がそれに乗算されるため、白じゃないと持ってきたテクスチャそのままの色が出ないからです。

ちなみに VERTEX3D の定義に行くと

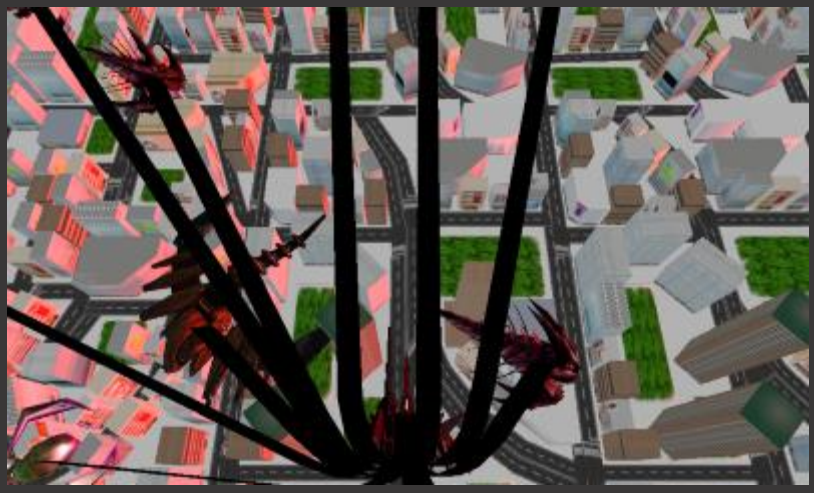
```
// ３Ｄ描画に使用する頂点データ型
typedef struct tagVERTEX3D
{
	VECTOR		pos ; // 座標
	VECTOR		norm ; // 法線
	COLOR_U8	dif ; // ディフューズカラー
	COLOR_U8	spc ; // スペキュラカラー
	float		u, v ; // テクスチャ座標
	float		su, sv ; // 補助テクスチャ座標
} VERTEX3D
```

と、なっています。

その中で今回いじって行きたいのはこの２つのパラメータです。既に dif は真っ白で設定しておりますので。

それでは2つの三角形を構成する三角形６個の頂点のパラメータを設定し DrawPolygon3D を使用して画面上に表示してみます。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N7

DrawPolygon3D(v, 2, ハンドル , false);
で画面上に表示してみてください。



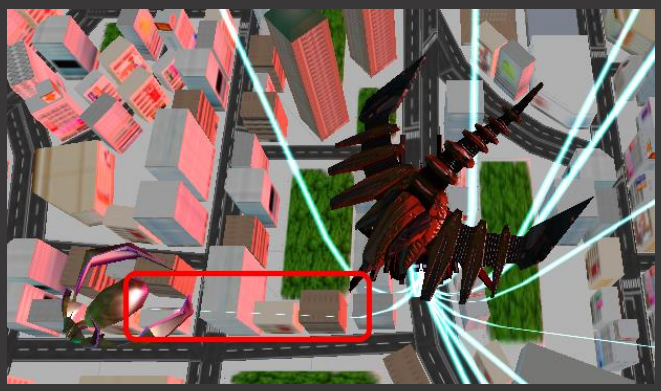
ところがこのように、真っ黒なトイレットペーパーになるかもしれません。これは原因があって、法線ベクトルを設定していないため、光源の影響を受けられないためですが、そもそもレーザーのようなエフェクト系に陰影があるのもおかしな話です。
というわけで、コイツを表示する間は一時的に光の影響を無効化します。
SetUseLighting で光の影響を消します。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R13N44
そうすると

のようにテクスチャが適切に表示されますが、まだまだレーザーっぽくないです。
レーザーは光ですから、「加算合成」にしたほうが良いでしょう。ここは 2D のときと同じく BlendMode の変更でどうにかなります。
http://dxlib.o.oo7.jp/function/dxfunc_graph1.html#R3N17
ブレンドモードを ADD にします。これで描画するものがこれ以降「加算」になります。
で、描画後はかならず NOBLEND に戻しておきましょう。これを忘れるとブレンドモードが加算のままになりますので、お気をつけください。
うまくいけば

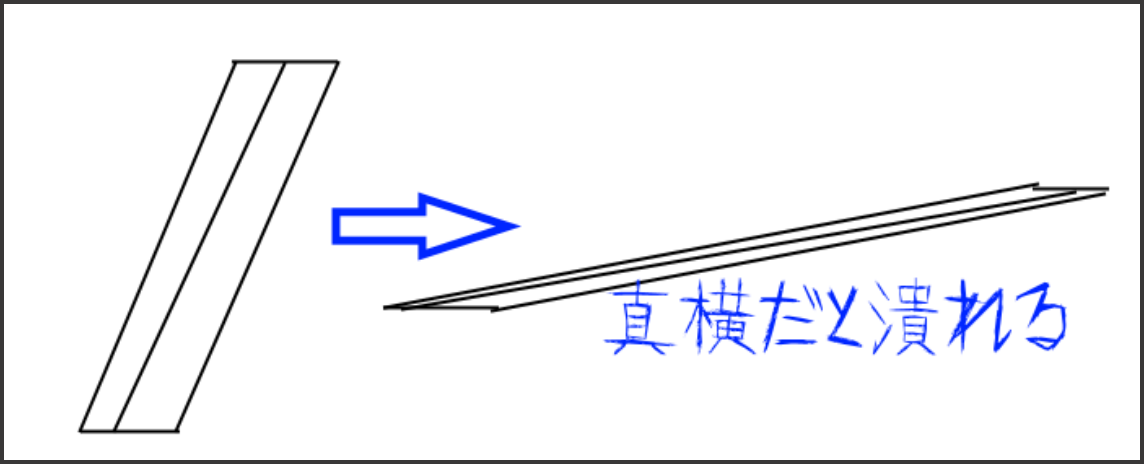
のようにかなりレーザーっぽい見た目になります。なお、描画順序のせいで、レーザーが常に手前に出てきて鬱陶しい場合は PlayingScene の先頭で
SetZBuffer3D を有効にしてみてください。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N12

## 見た目改善①真横に来ると潰れる

今はレーザーの幅を X 方向に伸ばしているだけなのでレーザーが真横に伸びると潰れた表示となります



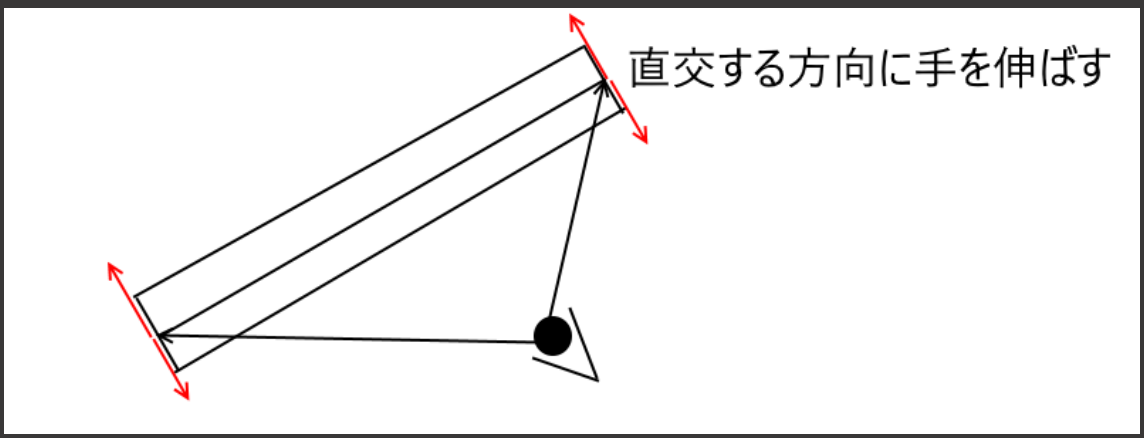
これは格好悪いのでどうにかしたいところです。可能な限り太さは一定に保ちたいですね。




これをどうにかするには、2D なら簡単ですよね？そう。レーザー方向から 90°回転させれば伸ばすべき方向は分かるはずです。ところがここは 3D です。
レーザーに直行するベクトルは

# 「無数に存在する」

のです。さぁ、困った…困ったなあ。とはいえよく考えてください。レーザーは「見た目上として幅があればいい」わけですから、依存しているのはレーザーベクトルだけではありません。
そう…視線にも依存しているわけです。




視線ベクトルとレーザーのベクトルから外積を取れば双方に直行するベクトルができるので、その方向に手を伸ばせばよいでしょう。

「視線ベクトル」の作り方はいたって簡単。レーザーのパーツの始点だか終点だかに対して始点からのベクトルを作ります。ベクトルの作り方は覚えてますよね？

そう…終点から始点を引く…のです。
で、一つ言っておくと、終点と始点で、伸ばす方向は同じで良いので…例えばレーザーパーツ

始点と視点で「視線ベクトル」を作り、レーザーパーツベクトルとの外積を取る。当然これは常にレーザーと視線に対して直交するベクトルだ。

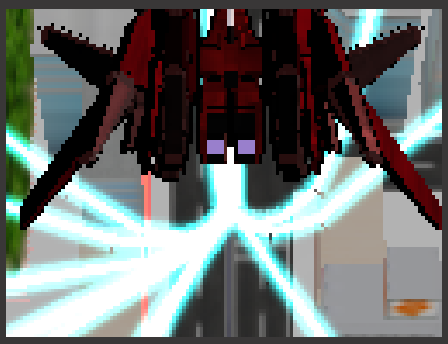
でこの計算で出てきた直交ベクトルは始点終点共用でかまわない。よく考えれば問題ないのが分かるはず。

うまく行けば図のように

真横でもきちんとしたレーザーが出ることでしょう。がんばってください。なお、この場合も外積が荒ぶらないように、視線ベクトルとレーザーベクトルの位置関係に注意しておこう。2つのベクトルが一致するようなら…if 文で差が 0.001 未満になるようなら強制的に x 方向に伸ばすようにしてあげればいい。

こういう「常にこっちを見せる」という仕組みは割と重要でよく使用されるので頭のなかに入れておいてください。

ちなみにうまいこと行っているようだが、拡大すると



ドット単位の切れ目が入っている。動いてると分かりにくいんだが…。ちょっと細かい話なので、それはあとに回して Weapon クラス化していこう。

では Weapon クラスを作ります。この光線の一本一本で new されると思っておきましょう。持ち主は Player とします。
これも量産できるように作る必要がありますので、レーザー画像は Player にロードさせましょう。コンストラクタで画像ハンドルを渡せる仕様にします。
また、発射時の座標と敵オブジェクトも知っておく必要がありますので、それも渡せるようにしておきます。

```
///@param handle レーザー画像ハンドル
///@param startpos 開始座標
///@param camera カメラオブジェクト
///@param enemy 敵オブジェクト
Weapon(int handle,Position3 startpos, weak_ptr<Camera> camera, weak_ptr<Enemy> enemy);
```

あと、先程の話でやりましたが、視点を知っておく必要があるため、カメラオブジェクトも渡しておきます。

これだけの情報があればなんとかホーミングレーザー表示はできるので、既に作っているホーミングを Weapon へ移植してください。

「弾」の扱いになりますので、敵に到達し次第オブジェクトを破壊することになりますが、現状ですと、弾が敵にずっとくっついている状況なので「破壊」まで行かなくても良いです。

まずは Weapon に移植しましょう。
大雑把に言うと、こういう感じで生成できるようになってれば OK
だいぶ書き方端折ったんですけど大丈夫ですよね？

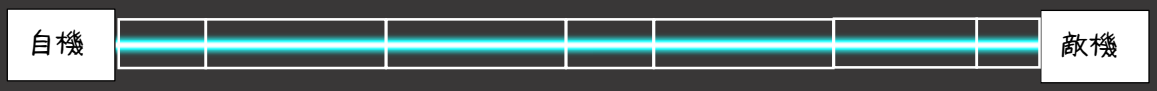
## 武器らしく動作させる

さて、それでは次にレーザーの長さを一定にしましょう。今は敵まで一気に伸びているものを、自機から少しずつ敵に近づけて行く感じにしましょう。


ではなく…本当は


こうすべきです。

ところが面倒なんだけど、曲がってることもあって




こんな感じになっとるわけですわ。
さて、どうしましょう。

実はそれほど難しい話じゃないです。軌道こそ同じ考え方で進むのですが、「尾を引く」という考え方にすれば難しくはないでしょう。

手順は
①尾の長さを最初に設定しておきます
②「尾」って結局は過去の座標の集合体なので vector かなんかでとっておきます
③うまいことつなげます

つまり今回の Weapon クラスは

- 今まで同様に敵に向かって座標とベクトルを更新する
- 内部に過去の座標を持っている

ですから
std::vector<Position3> _history;
は必要です。

で、このままの思想でやっていくと、曲率の高い所は遅く、曲率の高い所は速くなってしまうため、ちょっと…工夫が必要になってきます。

一度に進む距離を決めておく話はしました。
つまり、1ゲームループで1ベクトルぶん動くというより、ベクトルの距離の合計が予め決めていた距離に到達するまでベクトル演算を行うわけです。

今回まで使用してきた再帰の話も無駄じゃないわけです。で、ヒストリに関してはベクタでいいでしょう。

最初っから難しい構造にすると死ぬので、最初はスピードがまちまちに見えても一旦は単純にベクトルを積み重ねる形でいきましょう。

何度も試せるように、ロックオンボタンで出せるようにしておきましょう。

まずはヒストリ関係なく、「今」のぶんだけ表示しましょう。そうすると、ホーミングはホーミングですが、レーザーという形ではなく、ちっちゃな弾みたいになると思います。

そこまできたら、ヒストリを記録します。

```
void
Weapon::Update() {
	if (!_isActive)return;
	_history.push_back(_startpos);
	_startpos += _currentvec;
	_currentvec = NextVector();
}
```

とりあえずヒストリを5個位持っておくようにしてやってみましょう。
で、ここに来て久々のリバースイテレータを使用します。

```
auto it = _history.rbegin();
int count = 0;
for (; it != _history.rend(); ++it) {
	if ((it + 1) != _history.rend()) {
		DrawQuadPolygon(*it, *(it+1), 20);
	}
	if ((++count) > 5)break;
}
```

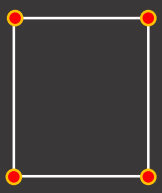
ひとまずはこんな感じで。

敵を追いかけていくようにしてみてください。まずは先頭のみ。その次は尾を引くようにしてみてください。

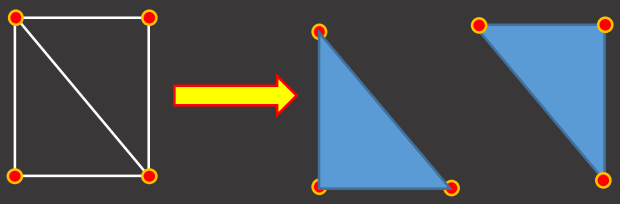
## インデックスを使用する

例えば、クアッドポリゴン(四角形)を作る時に、三角形はいくつ必要でしょうか？2つですね？では頂点はいくつ必要でしょうか…えーと、三角形が2つだから 3*2=6 で6個ですね。

そうなのですが、例えば下の図を見てみると4角形は4頂点あれば事足りるわけです。ところ



が、CG ポリゴンの最小単位は三角形であるため、以下の図のように、6頂点必要になる。



ここまではいいかな。ただし無駄な頂点2つ分は勿体無い。実はこの頂点情報は VERTEX3D であり、そのバイト数は 40 バイトである。
そうすると、本来 4*40=160 バイトで済んでいたものが 4*60=240 バイトと、高々クアッドポリゴンで随分な頂点が無駄になることになる。80 バイトのロスである。

もうちょっというと、モデルを移動させたり回転したり、スキニングさせたりする時に無数の頂点が動くわけだ。これが先程のように頂点が同じ場所で重なっている場合、この無駄な変換も頂点ごとにかましてあげなければならない。

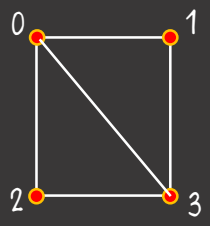
これはバイト数的にも計算量的にも無駄。

そこで考え出されたのが「インデックス」という考え方だ。これを使用すれば今回のような場合は4頂点で済む。だが、どの頂点とどの頂点とで面を構成するのかというデータが必要。それがインデックス配列なのだ。

配列には添字がつきものです。つまり頂点配列の頂点が持っているインデックスを3つずつ並べたもの…それがインデックス配列です。

インデックスは通常 unsigned short で表現される。ただし unsigned short は 65535 までしか

数えられないので、もし頂点数が7万を超えてしまえば使えなくなるだろう。




まぁ、自分で作っていく程度の頂点数なら大丈夫だろう。

で、インデックスデータとして、indices ってのを作る。

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

なんて書いてやってインデックスを登録しさえすればオッケー。で、インデックスを使用する場合は DrawPolygon3D ではなく、DrawPolygonIndexed3D を使用します。
http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N8
よくマニュアルの説明を読んで使ってね。

## 段々と消えるようにしてみる

同じ明るさだと尾の見栄えがちょっと微妙なので、段々と消えるようにしてみましょう。

難しいと思われるかもしれませんが至って簡単。

```
void DrawQuadPolygon(Vector3& startpos, const Vector3& endpos, float width,float brightness=1.0f);
```

まず、DrawQuadPolygon に brightness パラメータを追加します。brightness ってのは明るさって意味です。
で、
頂点のディフューズカラーに brightness をかける

```
vertex.dif = GetColorU8(255, 255, 255, 255 * brightness);
```

もしくは

```
vertex.dif = GetColorU8(255 * brightness, 255 * brightness, 255 * brightness, 255);
```

どっちでもたいして変わらないのでお好きにどうぞ

で、ヒストリを描画する際にだんだんと brightness を小さくしていく。
DrawQuadPolygon(*it, *(it + 1), 20, 1.0f / static_cast<float>(count / 3 + 1));

### スピードを調整する

次に「スピード」と「尾の長さ」を設定できるように作ってみて、もう少し見栄えを良くしましょう。

## Effect クラスを作ろう

ちょっと今は敵に当たったかどうか分からないので、当たったら爆発エフェクトを出すようにしたいと思います。今回は Effekseer というライブラリを使ってみます。

https://effekseer.github.io/jp/download.html#dxlib

ちょっと 135MB と、デカいファイルなので、ぼっちぼちダウンロードしておいてください。アクセスが集中しちゃうと多分ものごっつ重くなるので休み時間とかに時間かけてダウンロードしてください。
ああ、32bit で開発してる人はそこまで苦労しないとは思いますが…64bit で開発してる人の場合は結構たいへんです。

### 下準備(ライブラリの用意:64bit の場合)

で、昔は Effekseer 別リンクするとかそういう仕様だったのに、今は DxLib と一緒にしてるのね…逆に面倒だと思うんだけどなあ…と、検証してみたところ、どうも DxLib バージョンは 32bit には対応していないっぽいですね。コンパイルエラーが出ました。
もちろん分けわかんないコンパイルエラーではなく、明確に「32bit 以外無理」というエラーでした。

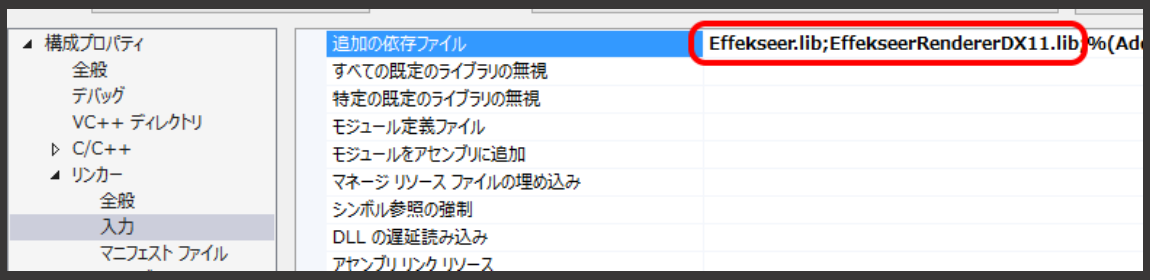
これはいかん…手も足も出ないのか？
んなこたぁない。と思って色々と頑張ってみました。よし、表示されました。ただし簡単じゃなかった。2時間位格闘した。

君らな～。ここではセンセーがこうやって人柱になってるから、ぶっちゃけスグにでも使用することができるんやで～？と恩を売ってみるんだけど、真面目な話、自らが人柱になろうとするなら思いの外大変なんやでという事は覚えておきましょう。

ドコらへんで詰まったかというと、まずEffekseerForDxLibが64bitダメ
DirectX9バージョンを使用しました→64bitでコンパイルしないとダメ
64bitでコンパイルしました→VSのツールセットが違うからダメ
VSのツールセットを2015に変更してコンパイルしました→実行時にクラッシュ
→DirectX9オブジェクトが生成されていない。
理由:既にDxLib自体がDirectX11で作られています。

おっとぉ…久々にDxLibを使ってみたらそういうことか…。そういうことだったよ。

というわけで、Effekseerを使えるようにする手順としては
①EffekseerRuntime最新版を落とす
②srcの中に入ってEffekseerRendererDX11.slnを開く
③x64でコンパイル
④出来上がったライブラリがsrc¥libに入っているのでこれをコピー
⑤Compiled¥Lib¥VS2015¥Debugの中にペースト
⑥環境変数を開いてEffekseerRuntime〇〇¥Compiledにパスを通す。EFK_DIRとかにする。
⑦プロジェクトのインクルードパスに$(EFK_DIR)¥includeを追加する
⑧ライブラリのパスに$(EFK_DIR)¥lib¥VS2015¥Debugを追加する
⑨Effekseer.libとEffekseerRendererDX11.libをリンクする




⑩プロジェクトをビルドしてみる
ここまでが"無理～(＞Д＜)"って人は、64bit用のlibファイルをサーバーの
¥¥132sv¥gakuseigamero¥rkawano¥Effekseer¥64bitDxLib用lib
にあげています。
上げていますが、自分で対応できるようになっておいたほうが良いと思いますよ？

ともかく、思いのほかクソほど大変だったよ。さて、リンクすることまでできたら
いよいよエフェクトの表示だ。

## 試しにエフェクト出してみよう

まずインクルードとして
#include<Effekseer.h>
#include<EffekseerRendererDX11.h>
を入れておいてください。

次に初期化を行う必要があります。レンダラーとマネジャーが少なくとも必要ですので
EffekseerRenderer::Renderer* _renderer;
Effekseer::Manager* _manager;
をヘッダに置いておきます。
ちょっとこいつらの破棄は「ただメモリを開放すればいい」のとは違いますので、しばらく生ポインタとして取り扱います。

いよいよ初期化処理ですね。
レンダラーとエフェクト管理のインスタンスを生成する必要がありますので、PlayingSceneのコンストラクタあたりで作っておきます。

ちなみにマニュアルの初期化部分の解説は当てにならないので注意しましょう。

まずは DxLib から DirectX11 のデバイス及びデバイスコンテキストを取得しましょう。
DxLib には隠し関数として GetUseDirect3D11Device と GetUseDirect3D11DeviceContexst があります。
GetUseDirect3D11Device で、DxLib で既に初期化されている DirectX を取得しましょう。
また
GetUseDirect3D11DeviceContexst で、DxLib で既に初期化している Direct3DDeviceContext を取得します。

この2つ必要なオブジェクトが得られたので、予め宣言しておいた_renderer にデバイスとデバイスコンテキストを使ってレンダラを作ります。
マネジャーは単独でも作れます。
```
auto dxdev = (ID3D11Device*)DxLib::GetUseDirect3D11Device();
auto devcon = (ID3D11DeviceContext*)DxLib::GetUseDirect3D11DeviceContext();

_renderer = EffekseerRendererDX11::Renderer::Create(dxdev,devcon, MAX_PARTICLE);
_manager = Effekseer::Manager::Create(MAX_PARTICLE);
```

次にEffekseerのカメラ行列とかプロジェクション行列を、DxLibの行列と合わせます。

```
DxLib::MATRIX proj = DxLib::GetCameraProjectionMatrix();
DxLib::MATRIX view = DxLib::GetCameraViewMatrix();
Effekseer::Matrix44 effproj, effview;
for (int i = 0; i<4; ++i) {
	for (int j = 0; j<4; ++j) {
		effproj.Values[j][i] = proj.m[j][i];
		effview.Values[j][i] = view.m[j][i];
	}
}
_renderer->SetCameraMatrix(effview);
_renderer->SetProjectionMatrix(effproj);
```

最初の2行でDxLibのビュー行列と射影行列を取ってきています。それをEffekseerの行列に突っ込みます。
構造が違うので面倒ですけど、二重ループでコピーします。
そして最後の2行でその行列をEffekseerにセットしています。

```
_mgr->SetCoordinateSystem(Effekseer::CoordinateSystem::LH);
```

その次に系を合わせときます。こいつはデフォルト右手系なんで、DxLibに合わせて左手系にしとくんです。

あと、カメラ行列が更新するたびにこのコピー⇒行列セットを行わなければ正しい場所に表示されませんので、カメラ座標とかいじったらその後でコピー&セット処理をする必要があります。

ちなみにMAX_PARTICLEは2000くらいにしとくと良いと思います。5000にしたら処理落ちし始めました。

次にエフェクトのロードですね。

```
_effect = Effekseer::Effect::Create(_mgr, (const EFK_CHAR*)_T("エフェクトパス"));
```

で、エフェクトオブジェクトを作ります。

そして再生。

```
_mgr->Play(_effect, 0, 100, 0);
```

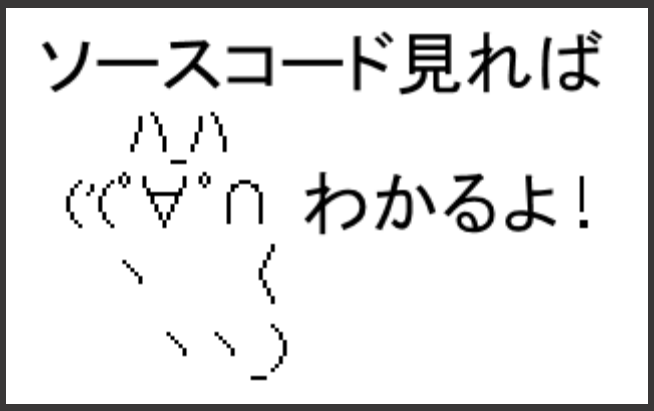
で、毎フレーム

```
_mgr->Update();
_renderer->BeginRendering();
_mgr->Draw();
_renderer->EndRendering();
```

を呼び出します。一応、この辺の話については解凍したフォルダのヘルプファイルをよく読んでください。

まずは「組み込み」→「実行」の項目を見ておいてください。サウンドに関しては今は無視しておいても構いません。

ちなみに「リファレンス」は意外と頼りになりません。「そんなに難しい仕様じゃないし…ソースコード見れば分かるよ」ってか!!




いや、初心者にとっては相当大変だと思います。そうは言ってもいつか通る道…今はその練習だと思って、頑張ろう。

## ハマる箇所

なんかみんな苦戦しているみたいなので、ハマる箇所を書いておきます。

**うまくいってなかった事例**

efk ファイルではなく efkproj を読み込んでた。

→efk を出力して、それを読み込む(efkproj は読み込むファイルではない)

Effekseer は文字が 16bit 扱いなのに 8bit でやろうとしてた

Unicode ではなくマルチバイト文字検索してファイルが見つからなかった

→とにかく Effekseer に渡すべきは 16bit 文字列である

対処法1→プロジェクトの設定で Unicode にする
対処法2→渡すときに2バイト文字に変換する。

カメラを追いかけきれてなくて、画面の左下に小さく表示されてた。
→毎フレーム、view 行列を更新する

なお
manager->Play は CD の再生ボタンみたいなもんです。
これは毎フレーム呼ぶ必要なし

毎フレーム呼び出す必要があるものは…
Update でタイムラインを進めて
Draw で表示する
エフェクトの Draw の前後で Manager の BeginRendering っと
EndRendering を呼び出す必要がある。

ここまでを把握しとけば画面上にエフェクトを出すことができるでしょう。

## 指定の場所にエフェクトを出してみよう

簡単です。
既にエフェクトの Play を呼び出していると思いますが、その要領で再生させてみましょう。
今回は僕はピエール氏のエフェクトを使用します。01Pierre02 の Benediction と FeatherBomb を使用します。出力の際の倍率は5くらいにしておくといいでしょう。

出力したら efk というファイルができますので、それをコンストラクタでロードしといて、衝突判定のタイミングで表示させてみましょう。

今は実験なので、そんなに厳密な衝突判定を行わなくてもいいので、とにかく当たった場所に表示させてみましょう。

ホーミングと敵の距離が 10 以内になったら、エフェクトを Play とかで構いません。

ちょっと贅沢なエフェクトのせいか、たくさん出すと一部表示されなくなります。

なので、自前でもっと軽いエフェクトを作る必要があるかもしれません。ともかく特定の座標に出してください。

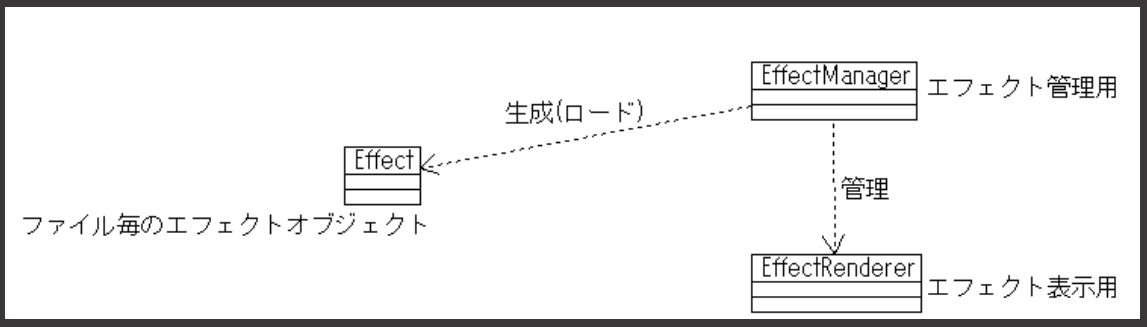
_manager->Play の第二引数以降が表示座標です。
h=_manager->Play(_enemyexplosion,epos.x,epos.y,epos.z);
こんな感じで。

今回はきちんとした当たり判定でなくても良いので、やってみてください。たまに変な場所に出てきますが、恐らくそれは当たり判定のタイミングによって敵がワープしてしまって変な所に行くためだと思います。

あとホンマにエフェクト消えるのどうにかせんとなあ……まぁどうもやっぱり発生パーティクル数が多すぎるのが原因ぽいのと、敵の当たり判定をきっちりやれてないのが原因のようですね。

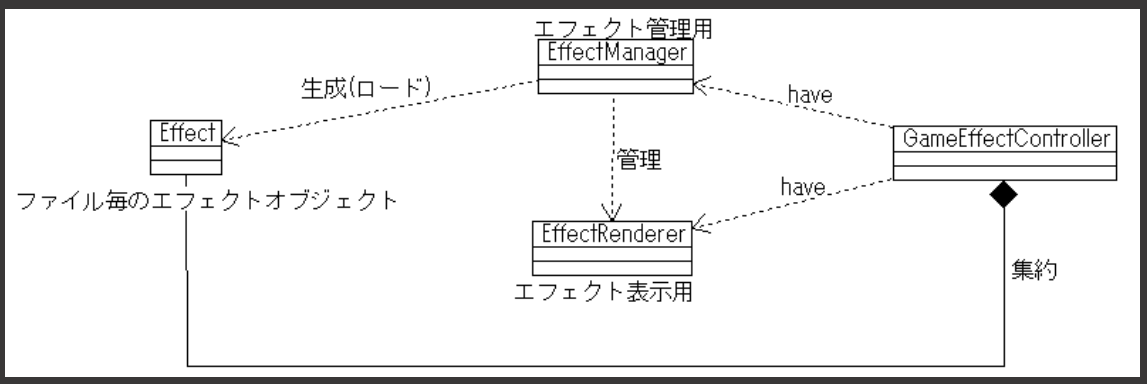
まぁちょっとあらかたできたら、これをエフェクトクラスとして作りましょう。
まぁ元々のクラスがきちんとカプセル化された構造なので、これをクラス化するのもどうかとは思うのですが、かと言って PlayingScene 内でそのまま使用するというのもちょっと良くない気がします。

現在は Effekseer の3つのクラスを使用しています。大雑把に書くとこういう関係




少なくとも、EffectManager と EffectRenderer はひとつずつあれば問題ないので、GameEffectController というクラスを作って、それのメンバとして EffectManager と EffectRenderer をもたせましょう。

んで、内部にエフェクトを持つわけですが、Effect そのものも内部に持っておき、必要な文を予めロードして map に持っておいて、エフェクトファイル名で Play していこうってわけです。




図にするとこんな感じね

マップのキーは何にしようかな…ファイル名でいいかな…

Effect/"ファイル名".efk
でロード

で、マップにはこの"ファイル名"で登録していこうかな…と。
ちなみに、C++の string 文字列と C 言語の char*文字列についてですが、例えば
benediction という文字列の場合
string s="benediction";

s(0)='b'
s(1)='e'
　:
s(11)='n'

となっています。

データとしてはstringってのは
std::vector<char>に毛が生えただけ
ですので
string s="benediction"のサイズは12になり
¥0が入っていません。

これをC言語の文字列char*として扱うときはこのままだと
ロクでもないことになります。

ということで、char*として使いたいときには、それ用の関数
c_str()を使用します。これは最後に¥0を付加したchar*文字列を返すものです。

# 解放処理

解放処理を行います。DirextX、OpenGL自体を解放する前に行う必要があります。

```
// エフェクトを解放します。再生中の場合は、再生が終了した後、自動的に解放されます。
ES_SAFE_RELEASE(effect);
// エフェクト管理用インスタンスを破棄
manager->Destroy();
// サウンド用インスタンスを破棄
sound->Destroy();
// 描画用インスタンスを破棄
renderer->Destory();
```

こういう事をやらなければならないので、下手にスマートポインタにしないほうが良いでしょう。ただ、それだけに解放処理は一箇所にまとめたほうが運用しやすいでしょう。

```
GameEffectController::~GameEffectController()
{
	for (auto& effect : _effectmap) {
		ES_SAFE_RELEASE(effect.second);
	}
	_manager->Destroy();
	_renderer->Destory();
```

}
こんな感じで。

あとは
コンストラクタで必要なもの
　　レンダラーとマネジャーの初期設定
　　必要なエフェクトをロード
　　系(右手、左手)の設定

毎フレーム必要なもの
　　アップデート
　　描画

デストラクタで必要なもの
　　レンダラーとマネジャーの解放

です。
あと、ちなみにデストラクターでエフェクトを解放するとクラッシュする原因は今のところ分かっていません。

# ステージクラスを作ろう

ステージクラスって言っても、要は敵の出現パターンデータです。
よく、ゲームづくりのサイトを見ると、エクセルで敵の出現パターンデータを用意してそれを読み込むという形で作っていますが、正直ちょっと分かりにくいと思います。

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | /カウンタ | 移動パター | 敵の種類 | x座標 | y座標 | スピード | 発射時間 | 弾幕種類 | 弾の色 | 体力 | 弾種類 | 待機時間 | アイテム1 | 2 | 3 | 4 | 5 | 6 |
| 2 | /cnt | pattern | knd | x座標 | y座標 | sp | bltime | blknd | col | hp | blknd2 | wait | item_n[0] | | | | | |
| 3 | 100 | 0 | 0 | 50 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 4 | 110 | 0 | 0 | 80 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 5 | 120 | 0 | 0 | 110 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 6 | 130 | 0 | 0 | 140 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 7 | 140 | 0 | 0 | 170 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 8 | 150 | 0 | 0 | 200 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 9 | 160 | 0 | 0 | 230 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 10 | 170 | 0 | 0 | 260 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 11 | 180 | 0 | 0 | 290 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 12 | 190 | 0 | 0 | 320 | -20 | 0 | 150 | 0 | 0 | 100 | 0 | 120 | 0 | -1 | -1 | -1 | -1 | -1 |
| 13 | | | | | | | | | | | | | | | | | | |

**「分かりにくい」というと誤解があるのですが、このデータ自体は正しい**と思います。ただ、皆さんがステージを作ろうとした時に、このデータでどの辺に敵が出てくるのかを想像しながら作れるでしょうか？

初心者だとなかなか難しいと思います。結局これらデータを見てると「出現位置」を表していると思います。「出現時刻」でも、結局は同じことですね。スクロール値がそこに到達したら出現するという意味ではね。

ということで、もし既にエクセルでシューティングゲームの出現データを作れるという方はエクセルで作っていってもらって構いません。

そこで RPG 等ではよく使用されているであろう「マップツール」を使用します。

もし既に特定のマップツールを使っていて、それのほうが使い慣れているのであれば、そちらを使用しても構いません。ただし、それの出力仕様はきちんと把握しておいてください。

今回は Platinum というソフトを使用してマップデータを作ろうと思います。
他にも色々と便利なエディタがあるとは思いますが、どれかに決めないと授業にならんのでひとまずこれで解説していきます。


多分、あれがいいだのこれがいいだのあると思いますが、そこは各人でご自由にお願いします。参考までに Platinum 以外のもので有望なのは

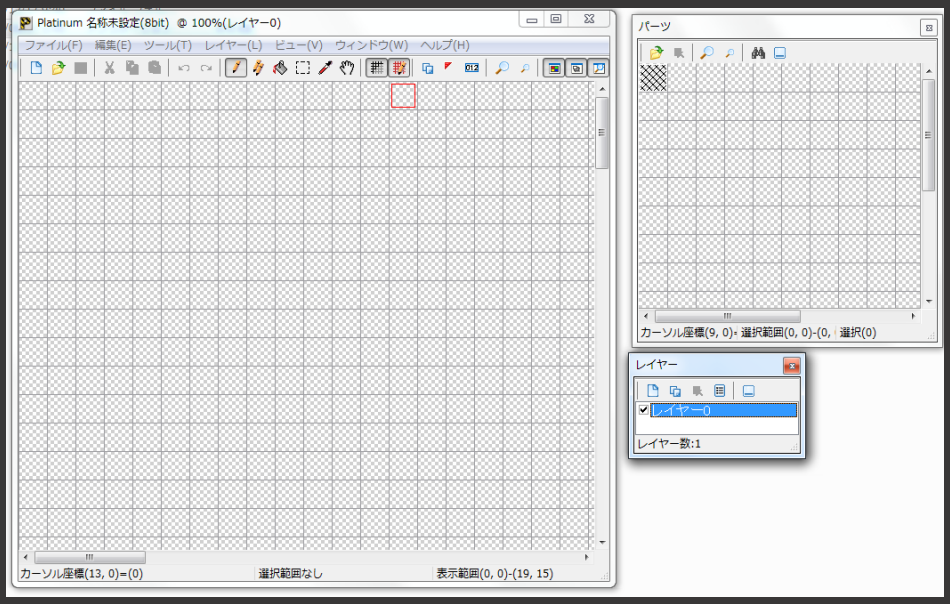
- Tiled
- Tiled Map Editor

とかそういうのですが、英語なので…僕は構いませんが、みんなが死ぬでしょ？
Quoyle ってツールもあったけど、作者 HP リンク切れてるし…platinum 使います(こっちも作者 HP リンク切れしてますけど)
最近だと、Unity でマップエディタがついてますしね。

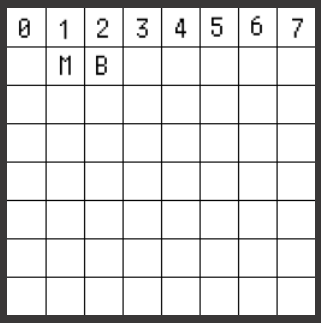
ともかく platinum 使います(しつこいようですがここまで言わないと、あれがいいだのこれがいいだの言われるので)

Platinum はサーバの gakuseigamero¥tools¥マップエディタ

にあります。解凍して起動してください。

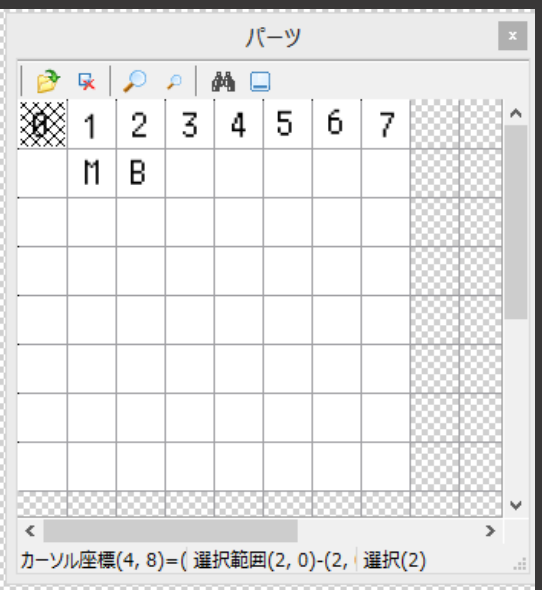



こんな画面が出るとおもいます。で、適当にペイントツールかなんかで




こういうのを作ります。

それをパーツツールで読み込みます。何故か png ファイルの読み込みができないので、画像は bmp 等で作っといてください。透明パーツがダメなんですかね。ともかくこれでペイントツールの要領でパーツで塗りつぶしていきます。



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 3 | 3 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

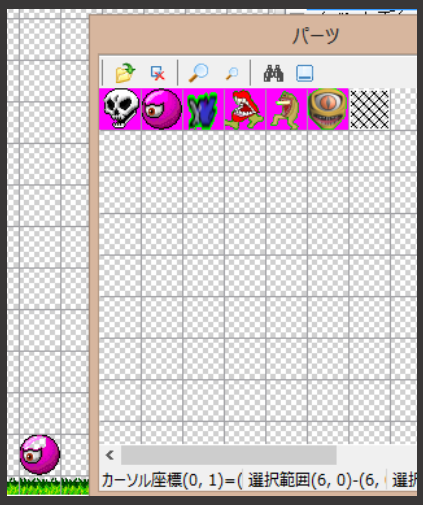
カーソル座標(4, 8)=( 選択範囲(2, 0)-(2, 選択(2)

わかりにくかったりするんで、それぞれの要所要所で色をつけておいたほうが分かりやすいかも。色を付ければ…

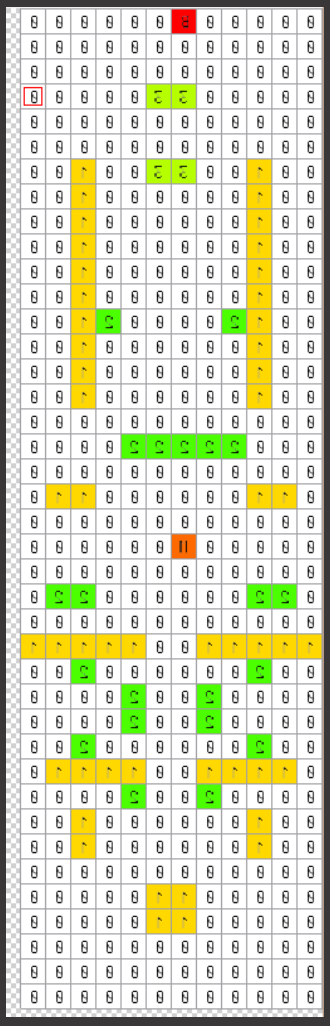
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | B | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 3 | 3 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 3 | 3 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

この通り。

分かりやすくなりました。番号ではなく、アイコンとかにしたほうが分かりやすいかもしれません。



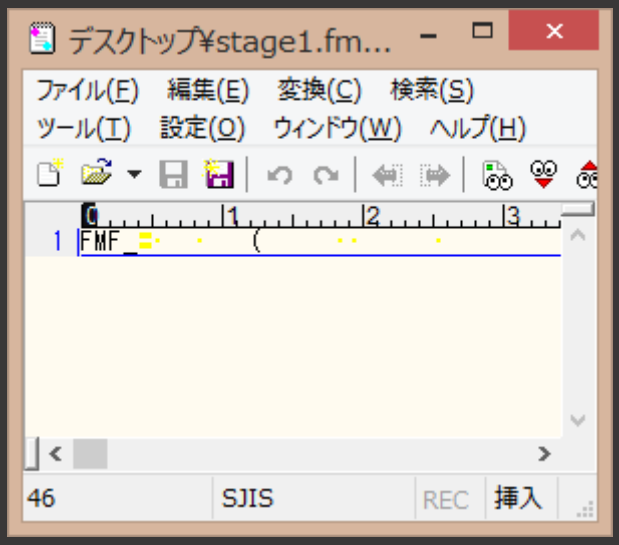

某オレオレアクションゲームのデータに使ってるやつです。ともかく、今回は適当にこういうデータを作ります。



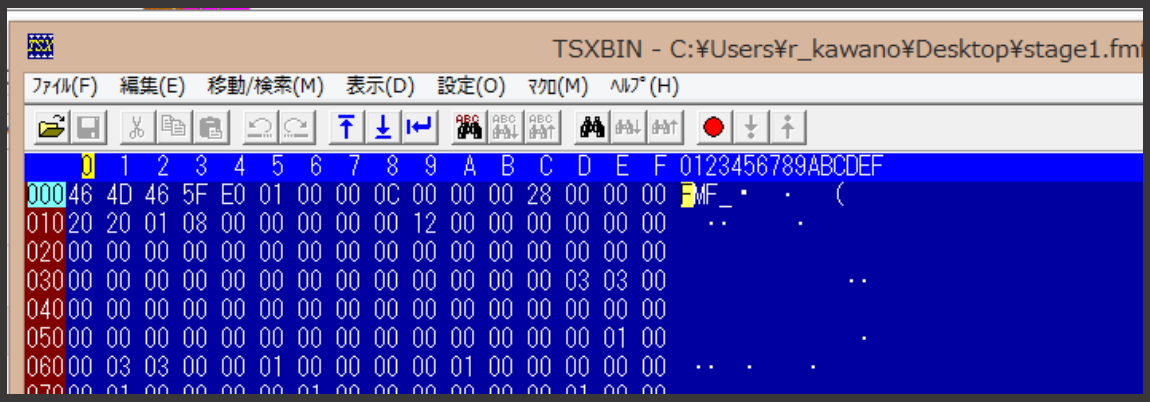
で、適当にステージデータを作ったら

敵をこの通りに出現させていきます。まず、ロードの部分から考えましょう。このツールの「書

き出し」でデータを出力します。「fmf で出力」を選択します。
このデータ、テキストファイルで中身を見ると




わけがわからないよ。なので、バイナリエディタで中身を見ます。




わからん!!まったくわからん!!
fmf ファイルについて.txt というテキストの中身を見てみましょう。
•データサイズの表記について
(U1)unsigned char(1byte)
(U4)unsigned integer(4byte)
バイトオーダーは最下位バイトから記録されるリトルエンディアンです。

【FMF ファイルヘッダ (20byte)】

| | | | |
|---|---|---|---|
| +0 | identifier | (U4) | // ファイル識別子 'FMF_'(0x5F464D46) |
| +4 | size | (U4) | // ヘッダを除いたデータサイズ |
| +8 | mapWidth(U4) | // マップの横幅 | |
| +12 | mapHeight | (U4) | // マップの縦幅 |
| +16 | chipHeight | (U1) | // パーツの横幅 |

+17　　chipHeight　　(U1)　　// パーツの縦幅
+18　　layerCount　　(U1)　　// レイヤー数
+19　　bitCount(U1)　　// レイヤーデータのビットカウント(8/16)

【レイヤーデータ】
FMF ヘッダの直後からマップパーツの値が mapWidth*mapHeight 個、ベタに並んでいます。

bitCount が 8 の場合、1 パーツ 8 ビット(1byte)
bitCount が 16 の場合、1 パーツが 16 ビット(2byte)
と、書かれています。
んで、実は TSXBIN では『シンボルファイル』というのを作れば、もうちょっとデータが分かりやすくなるという仕様があります。

このデータ仕様を元にテキストエディタでこういうファイルを作ります。

```
#include "typedef.h"

$BYTE identifier[4];// ファイル識別子 'FMF_' (0x5F464D46)
$DWORD size;//ヘッダをのぞいたデータサイズ
$DWORD mapWidth;//マップの横幅
$DWORD mapHeight;//マップの縦幅
$BYTE chipWidth;//パーツの横幅
$BYTE chipHeight;//パーツの縦幅
$BYTE layercount;//レイヤー数
$BYTE bitCount;//レイヤーデータのビットカウント(8/16)
```

これを FMF.SYM という名前で保存して、そして、閉じてください。

TSXBIN のシンボル再読込を使うと…

```
000 identifier[0] 46 4D 46 5F                                     FMF_
004 size          000001E0
008 mapWidth      0000000C
00C mapHeight     00000028
010 chipWidth     20
011 chipHeight    20
012 layercount    01                                              .
013 bitCount      08                                              .
014               00 00 00 00 00 12 00 00 00 00 00 00 00 00 00 00      .
024               00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

このようにまぁ比較的分かりやすいデータとなります。

あ、16 進数だと分かりにくいですかね？
その場合は下の SIGN だか UNSIGN を押してください。

```
identifier[0]  70  77  70  95
size          480
mapWidth       12
mapHeight      40
chipWidth      32
chipHeight     32
layercount     1
bitCount       8
                0   0   0   0   0  18   0   0   0   0   0   0   0   0   0   0
                0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
                0   0   0   0   0   0   0   0   0   3   3   0   0   0   0   0
```

ご覧のようになります。ここまでくればそれなりに意味が分かるのではないでしょうか？
ていうか「FMF ファイルについて」に書いてあったことまんまですね。

データのサイズが 480 バイト、ステージの幅がパーツ 12 個分で、縦がパーツ 40 個分です。今回はチップの幅と高さは関係ないので読み飛ばします。

layercount はレイヤー数です。今回は敵のデータしか無いのですが、いくつも重ねたりする時にはこれが増えます。

次もビットカウンタは読み飛ばしでいいでしょう。
ここからがデータ部分ですね。

この手のデータはだいたい大きく

- ヘッダ部
- データ部

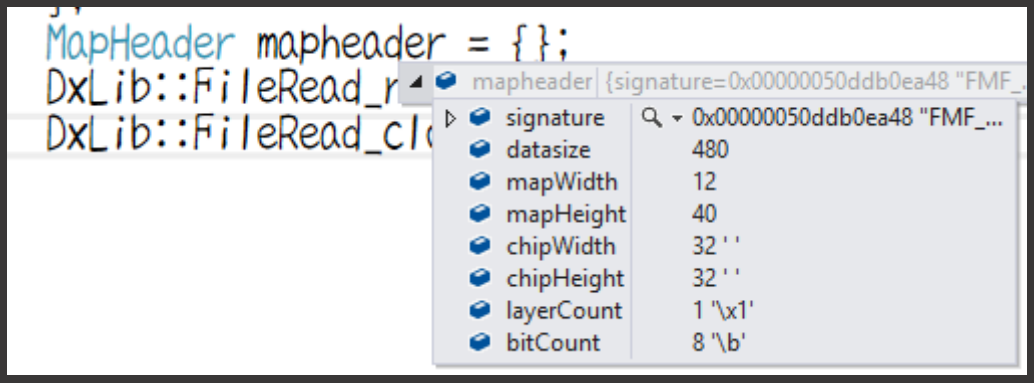
に分かれます。だいたいヘッダ部はどこかしらに仕様が書いてありますので、それを元に構造体を使って一発読み込みすればオッケーです。

ヘッダの構造体は例えば

```
struct MapHeader{
        char signature[4];
        unsigned int datasize;
        unsigned int mapWidth;
        unsigned int mapHeight;
        BYTE chipWidth;
        BYTE chipHeight;
        BYTE layerCount;
```

```
	BYTE bitCount;
};
```

このように定義します。
この定義したとおりにヘッダファイルを読めば、中にデータ値が入っているのが分かります。
まぁ、fopen でも DxLib::FileRead_open でもどっちでもいいので読み込んでみて、中のデータを確認してください。




設定したどおりのデータを取得できていれば上のスクショのような状況になるでしょう。
バイナリデータのロードは…できるよね？
fopen と FileRead_open の違いは「非同期読み込み」が可能かどうかです。「非同期読み込み」というのは、ロード中に「Now　Loading」を動かせるかどうかに関係しています。つまり通常の fopen をそのまま使用すると、アプリ全体の処理が止まってしまうため NowLoading が停止するのですが、非同期読み込みにすると、NowLoading を動かし続けながらロードすることができます。
製品版のゲームは同人も含めて、大抵のものに実装されていますので、実は重要なものです。
ただし、扱いがちょっとややこしいので、今回は非同期読み込みにはしません。

流石にそこまで解説したくないんですけど…。まぁ、もし fopen でやるなら

①ファイルオープン
FILE*fp=fopen("ファイル名","rb");
で

②ファイルリード
実際にファイルからデータを取得します。
ちなみに関数は fread です。
http://www.orchid.co.jp/computer/cschool/CREF/fread.html
fread(読み込み先アドレス,読み込みデータサイズ,1,fp);

で、最後にクローズとくるわけだが、まだ早い。

データ部分の読み込みが終わってないからだ。

今回、データは全て1バイトだ。
予め縦横幅が分かってればデータを作ることができる。横12の、縦40だ。

まず全データを取得しようか。
こういう場合どういう手順を踏むか分かるかい？

1. 最初に読み込みに必要なデータ領域を確保する
2. 実際に読み込む
3. 使いやすいように加工する

だね？

まず、データ領域の確保だけど、こいつは大した話じゃないと思う。1バイトデータなので
std::vector<BYTE>もしくは std::vector<unsigned char>でベクタを宣言しておく。

次に、データサイズで、ベクタを広げる
_data.resize(datasize);

読み込みだが、以前にも何度か言ったように、ベクタは連続したメモリ上に配置されることを保証しているのが特徴だ。つまり、確保師さえすればそこにデータを読み込める。ただし注意点は
fread(_data,～
や
fread(&_data,～
などとは書かないことだ。何度も言うようにベクタは配列のように扱えるが配列ではない。
メモリも連続しているが、先頭アドレスは&_data ではない。

先頭の「値」は_data(0)で取ってこれるのだから、そこに&をつければよい。

fread(&_data(0),～

というわけだ。ともかくこれでデータを取得できる。データの塊は得られるわけだ。あとはこ

れをどう解釈するか…だ。

データをどういう風に作っているかにもよるのだが、今回僕はこのデータを下から上に読むように読み込んでいる。

ということはリバースイテレータを使用したほうがよいかなーって思います。

で、このイテレータをそのまま返すというよりは一定時間ごとに1段ずつ遡っていき、その段にある敵データを敵オブジェクトとして生成させるのが良いだろう。一応、敵を生成する役目はPlayingSceneにもたせているため、このステージデータもPlayingSceneに持たせておき、その都度1列データとして返させ、それを元にしてEnemyFactoryに敵を生成させる。

何フレーム毎に列を遡っていくかルールは、実行しながら調整していくことになるでしょう。

```
struct EnemyPositionData {
	char enemyType;
	float posX;
};
```

こういうデータを作っておいて、これの1列データを返せるようにしておきましょう。

```
std::vector<EnemyPositionData> ReadLine(){
	std::vector<EnemyPositionData> ret;
	ret.clear();
	if(_currentRit==_data.rend()){
		return ret;
	}
	int count=0;
	for(;_currentRit!= _data.rend();++_currentRit){
		EnemyPositionData e={*_currentRit,count*100-600 };
		if(e.enemyType==0|| e.enemyType == 255){
			if (++count == _mapWidth) {
				break;
			}
			continue;
		}
		ret.push_back(e);
```

```
                if(++count==_mapWidth){
                        break;
                }
        }
        return ret;
}
```
こんな感じで。
で、以下のような関数を作って、データの終わりに来たのを PlayingScene が分かるように
```
bool
StageController::IsEnd()const{
        return _currentRit == _data.rend();
}
```
こんな感じで作っておきます。

この上で、PlayingScene 側で
```
if (!_stage->IsEnd()) {
        scrollZ += scroll_speed;
}
```
とやれば、採集データに来た時点でスクロールを止められます。

# より「らしく」していこう

ここまでで、基本的な部分は出来てきました。ここからはより「らしく」していきます。
ところで、ここまで作った時点で X ボタンがちっとも反応してないなーって思ってたんですが

- PAD_INPUT_X は xbox コンの Y
- PAD_INPUT_Y は xbox コンの LB
- PAD_INPUT_C が xbox コンの X

ってな感じになってました。


何でなんでしょう…マジで。良く分かりませんが、定義見ると

```
#define PAD_INPUT_A              (0x00000010)    // Aボタンチェックマスク
#define PAD_INPUT_B              (0x00000020)    // Bボタンチェックマスク
#define PAD_INPUT_C              (0x00000040)    // Cボタンチェックマスク
#define PAD_INPUT_X              (0x00000080)    // Xボタンチェックマスク
#define PAD_INPUT_Y              (0x00000100)    // Yボタンチェックマスク
#define PAD_INPUT_Z              (0x00000200)    // Zボタンチェックマスク
#define PAD_INPUT_L              (0x00000400)    // Lボタンチェックマスク
#define PAD_INPUT_R              (0x00000800)    // Rボタンチェックマスク
```

うーん。余計なやつが入ってますね。このせいで、箱コンとの整合性がとれてないようです。

恐らく箱コンであれば

PAD_INPUT_A→A

PAD_INPUT_B→B

PAD_INPUT_C→X

PAD_INPUT_X→Y

PAD_INPUT_Y→LB

PAD_INPUT_Z→RB

という割当になってるでしょう。キーボードも含めれば

PAD_INPUT_A→A→Z キー

PAD_INPUT_B→B→X キー

PAD_INPUT_C→X→C キー

PAD_INPUT_X→Y→V キー

のようです。面倒ですね。

## そろそろ「ロックオン」をきちんと実装しよう

まずレイストームのロックオンの仕組みですが…


こんな感じでロックオンのための照準を持っています。コイツの場所は決まってて、自機から

前方にちょっと離れた場所にあります。
自機からの位置関係に変化はありません。

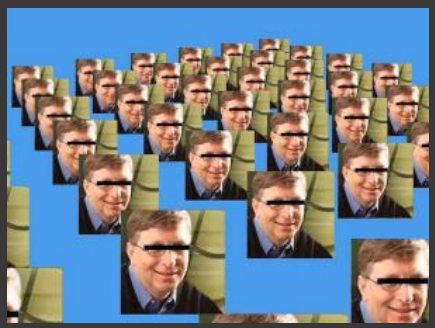
## 照準の表示

今回、こういう画像を作りました。
これを画面上の任意の座標にいい感じに表示するには DrawBillboard3D 関数を使用します。

http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R14N9

ビルボードって仕組みがあって、これは 3D 空間上に 2D の板を表示する技術です。この板は常にこちらを向いているのが特徴です。

通常板ポリに画像を貼り付けただけだと上の画像のようにどんどんペラペラになっていきます。しかしビルボードという技術を用いると



全員がこちらを向いてくれます。

これはパーティクルや、ステータス表示などに使用される技術です。細かい数式的な話は後期に演る予定ですが、ここではこういうのを「ビルボード」ということを覚えておいてください。

（まぁ仕組みを簡単に言うとカメラの回転逆行列を使用して「回転しなかったこと」にするわけよね…意味分からんと思うけど、一言で言うとそういうこと）

ともかくコイツを使って照準をじきの少し前に表示させましょう。

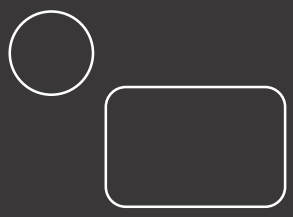
こんな感じ。
もし、背景のビルに隠されるのが嫌だったら、描画の順序を変えちゃいましょう。表示直前に Z バッファを無効にしていると思いますが、
無効直後に照準を表示すればいいと思います。
そして、Z バッファを有効に戻す。

```
SetUseZBuffer3D(false);
_player->DrawFocus();//照準の描画
SetUseZBuffer3D(true);
```

こういうことね。

## ロックオン

これは結構ややこしい。何故なら照準の範囲内に入ったとどのようにして判断するのだろうか？これが2D なら大した話じゃない。



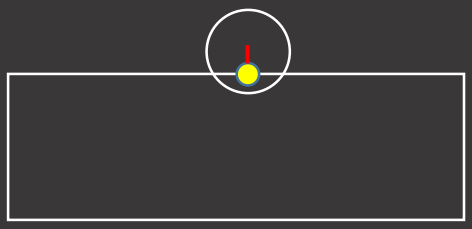
こういう当たり判定で十分だからだ。
あー、もしかして 2D における矩形と円の当たり判定ってやってない？参考までに話しておくと、3つのパターンで当たってるか判定して、どれも不可なら当たってないと判断します。

1. 矩形を構成する4つの点のどれかが円の中に入っている

2. 矩形の辺のどれかと円が重なっている
3. 矩形の中に円が入っている

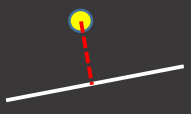
①は、説明するまでもないよね？各点が半径以内に入っていたら「当たり」。特に解説しない。

②は、以下の図のようなパターン



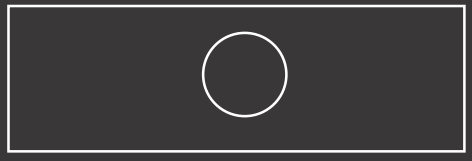
これはちょっとだけ難しいんだけど、まぁ簡単に言うと辺に垂線を下ろすわけ。その長さが円の半径よりデカイか小さいかで判断する。2D で垂線の長さを測るのはそれほど難しくない。外積を用いればいい。ちなみに矩形のそれぞれの辺の長さが分かっているため、垂線は(点～円の中心ベクトル)と辺ベクトルで求められる。もちろん矩形が AABB であれば中心の座標と、辺の座標はわかってるんだから外積を取るまでもない。
今回の話はあくまでも OBB だった時の話だ(3D の話につながっていく)。OBB だった場合は横方向(辺に水平な方向)と縦方向(辺に垂直な方向)に分けて考える。

水平な方向は内積を使えば求まります。まずこれで辺の長さ以内にあるかどうか確認します。垂直な方向は外積で求まる(2D の場合はスカラーになる)ので上の図の赤線の長さと円の半



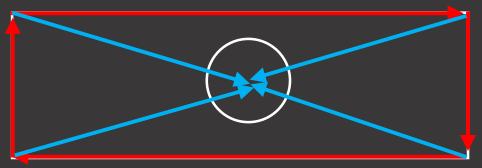
径を比較します。これを4辺全てに行います。どの辺にも触れてなければ③を確認します。

③は矩形の中にあるパターン



これはどうやって判断しましょうか。これは外積4回使えば終わります。何故かと言うと外積

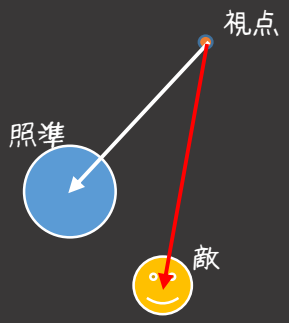
はベクトルの左にあるのか右にあるのかを判断するため…



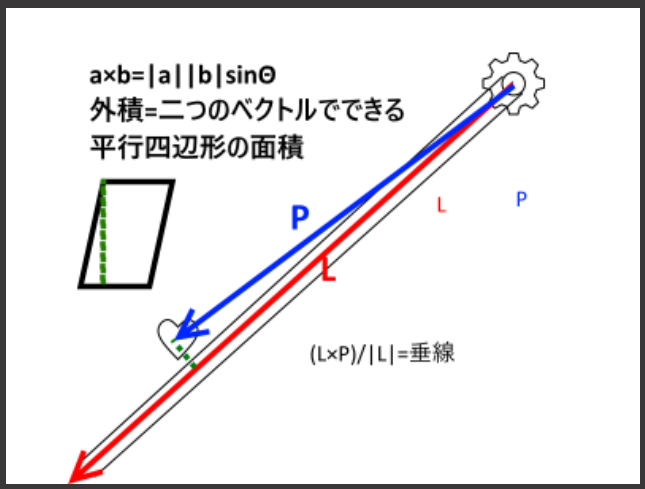
辺を構成するベクトルと、端点から中心に向かうベクトルが全て同じ符号であれば矩形の中に円の中心が入っているため、内部にあることがわかります。

…とまぁ、ここまでは2D の話。3D ならどうするべきか？かなりややこしいのでまずは、同心円に入っているかどうかを考えよう。

この辺からホンマに数学なので我慢してね。

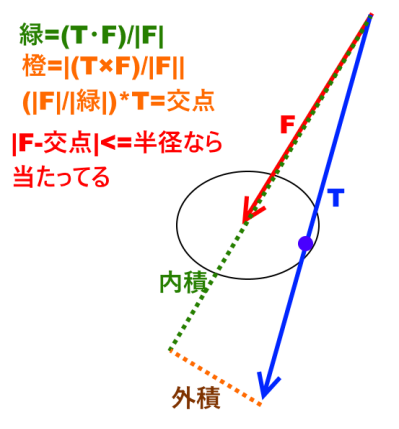



こういう位置関係を考える。垂線の長さは外積の長さと一致し、視点から垂線までの距離は内積に一致する。
ちょっと数学マニアックな話なんですが、外積の大きさは平行四辺形の大きさに等しいというのがあります。証明は
http://physmath.main.jp/src/cross-product-parallelogram-area.html
におまかせしておいて、詰まるところ




というわけなので、外積の大きさを求める→垂線の長さを求めるに等しくなる。

翻って、3D の照準範囲内に入ったかどうかの判定についてですが、




視点から敵へのベクトルを$\vec{T}$とし、視点から照準の中心へのベクトルを$\vec{F}$とすると
T の F への射影を P とすると

$$P = \frac{\vec{T} \cdot \vec{F}}{|\vec{F}|}$$

である。図のようにベクトル$\vec{T}$と、$\vec{T}$の$\vec{F}$への射影とで直角三角形ができる。
さて、元の F の長さと、この三角形の底辺の比は

$$P : |F|$$

である。で、この大きな三角形は照準平面で分断できる。このとき三角形の比も

$$P : |F|$$

である。つまり照準平面における部分ベクトル$\vec{T'}$は

$$\vec{T'} = \left(\frac{|\vec{F}|}{P}\right)\vec{T}$$

そして P を展開すると

$$\vec{T'} = \left(|\vec{F}|\frac{|\vec{F}|}{\vec{T} \cdot \vec{F}}\right)\vec{T}$$

と、まぁ割とごめんなさい展開になってきましたが、我慢しましょう。

$$\vec{T'} = \frac{|\vec{F}|^2}{\vec{T} \cdot \vec{F}}\vec{T}$$

という式ができました。これが「プログラマの仕事」です。何度も言ってると思いますが「式を作る」のがプログラマの仕事です。
で、照準中心点との距離は

$$|\vec{T'} - \vec{F}|$$

で求まりますのでそれと半径を比較して範囲内に入っているかどうかを判定します。

$$|\vec{T'} - \vec{F}| < R_f + R_e$$

でも良いのですが、これだと sqrt 処理が入って面倒なので

$$\left|\overrightarrow{T'} - \vec{F}\right|^2 < (R_f + R_e)^2$$

で良いと思います。
長々と書きましたが、ある程度数式で考えられる訓練をしておいたほうが良いと思います。
いきなりプログラミングするのではなく、図と数式そしてプログラミングくらいに思っておいたほうが良いです。

ゲームのプログラミングは

1. 図を書いて考える
2. 数式で考える
3. プログラムを書いて考える

のようになっていると思ってください。たまに天才が1も2もすっ飛ばしていきなり3で行けたりしますが、それは一部の天才です。変なところだけ真似しようと思わないように。

さて、関数の設計を考えましょう。

必要なものは

- 照準の座標
- 敵の座標
- カメラの座標
- 照準の半径
- 敵の半径

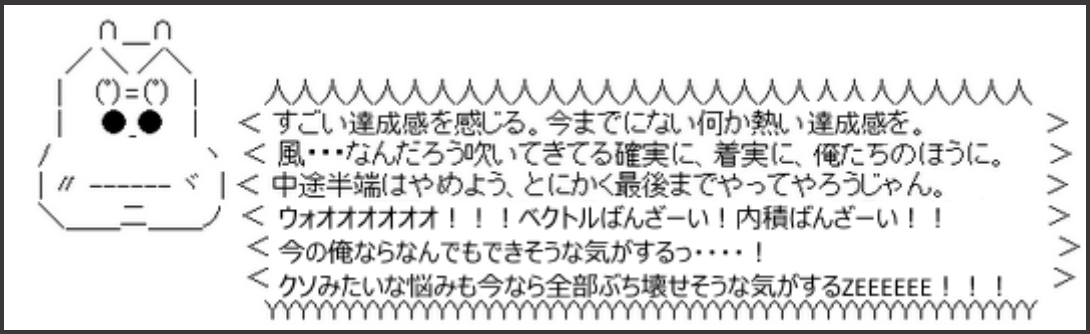
です。5つの引数はちょっと辛いんですが、まぁ、仕方ないっすわな。

プログラムにするとこんな感じ

```
Vector3 focusV = focusPos - cameraPos;
Vector3 targetV = enemyPos - cameraPos;
targetV *= ((focusV*focusV) / (focusV*targetV));
Vector3 retV = focusV - targetV;
return (retV*retV) <= (focusRadius + enemyRadius)*(focusRadius + enemyRadius);
```

補足しとくと*は内積を表しています。
ただし３行目の*=は内積ではなくスカラー倍です。

ちなみにいうと、今回の理論も「テキトー」です。ですが、思いの外ぴったりハマりました。こういう時はものごっつい気持ちいい。




ロックオンしたらロックオンしたと分かるようにしよう

サーバにこういう画像を用意しておりますので、ロックオン済みの敵にこういうのがついてまわるようにしてみましょう。

敵に locked フラグを作っておいて、
LockOn でフラグが立って、IsLocked でフラグが立っているのを確認できるようにします。そうした上で

```
//ロックオン
for (auto& enemy : _enemyfactory->EnemyList()) {
		if(enemy->Locked())continue;
		if (_player->CheckLockable(enemy->GetPosition(), 20)) {
			_player->AddLockonEnemy(enemy);
			enemy->LockOn();
		}
}
```

こんな感じでロックオンできるようにします。

## ショットも出せるようにしよう

今回はホーミングレーザーを先に描画したのですが、本来は基本攻撃は通常弾です。ちなみに Wikipedia にも書いてありますが、ショットは押し続けで連射です。東方系と同じですね。

ここのみなさんだったら説明するまでもなく作り方はご承知のことと思いますが、軽く説明しておきます。

トリガー処理はわかってると思いますが、おしっぱ連射は、どうやるかというと押しっぱなしにしている時間(フレーム数)をはかります。変数…例えば _pushingShotFrame とかで押してる間だけ毎フレームインクリメントしていきます。押してないときには0リセットしておきます。

その上で剰余(%)を使用して発射タイミングを決めます。

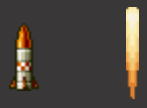
```
if(_pushingShotFrame%インターバル==0){
        発射
}
```

というわけです。

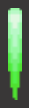
ちなみにショットは Shot クラスを作って Projectile 状態(びゅーんと飛ぶ状態)にしてればいいでしょう。敵に当たるか画面外に行ったら Kill するように。

そしてこれも Player 管理(ベクタ配列か、ただの配列で十分)で良いのではないでしょうか。

ちなみにショット画像はサーバーに上げています。なかなかいいのが見つからなかったため、メタルスラッグの画像を引っ張ってきました。

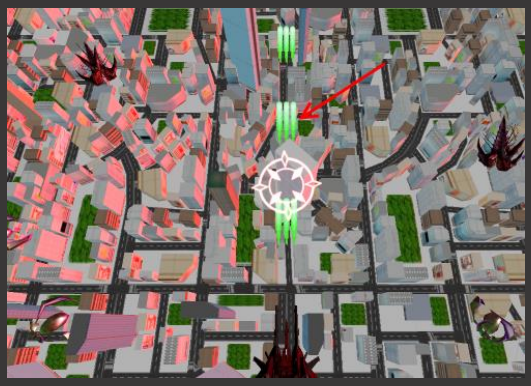


こいつを DrawBillboard3D かなんかで画面に表示します。
用意してますが、自分で何かいい画像見つけたらそっちを使ってほしいし、これを改変しても良いと思います。

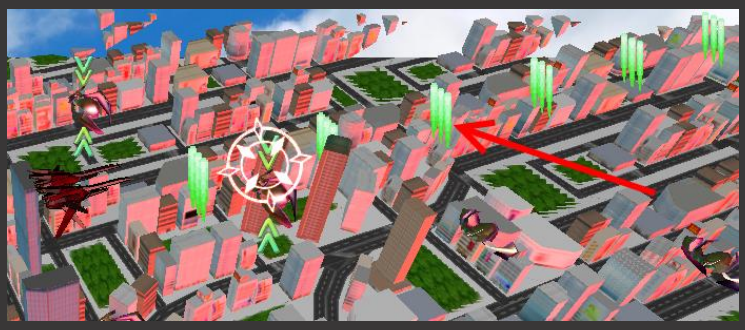


と、ここまで書いて、自分で実装してみたらあることに気づきました。DrawBillboard3D は、

「カメラ向きにかかわらず、その絵をそのまま表示」するものです。
真上からカメラを向けている時は問題なかったんです…ところがカメラを真横に向けると…悲Ｃ事になります。
まず、真上から見た時



うん、まぁ問題なさそう
ところが、カメラを横方向に持ってきた時は…



あれ…？
なんという事だぁ…
これはマズいですね。面白くはあるんですけど、流石に不自然です。

前にも書きましたが DrawBillboard は
「カメラ向きにかかわらず、その絵をそのまま表示するもの」なので所定の位置に「そのまま」表示しちまってるんですよね。

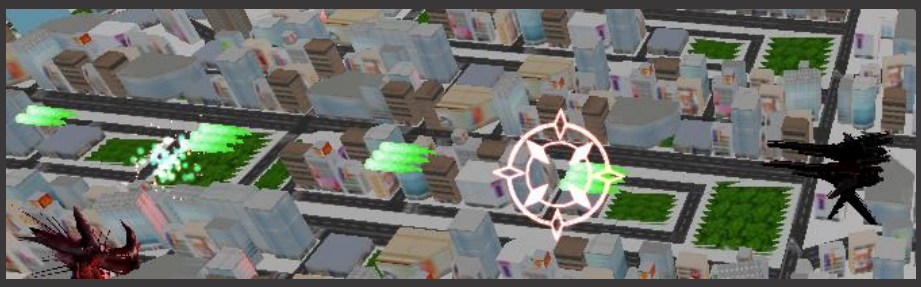
しかしこれは想定していた状態とは、まるで違うと言わざるを得ない。そこでレーザーの時に作った DrawQuadPolygon を流用しましょう。

つまり「絵は進行方向に向く」かつ「画像が常にカメラの方向に向いている」を同時に実現させれば良いわけです。

ホーミングレーザーの時にきちんと理解していれば、あれの応用で作れると思います。

↑真上から見た図

↓横から見た図



## ショットとの当たり判定も作っておこう

PlayingScene 内で、ショットと敵との当たり判定をするには

```
for(敵ループ){
        if(敵死んでる){
                continue;
        }
        for(ショットループ){
                if(弾死んでる) {
                        cotinue;
                }
                if(当たり判定){
                        当たり処理
                }
        }
}
```

こんな感じでやればいいですね。正直なところ、ネスト数が気になりますが…シューティングゲームの当たり判定の時はこんなもんだと思います。

で、ショットの場合は高さがプレイヤーの高さ固定なので、高さがあってないヤツには当たり

ません(レイストームはそれでいい)


さて、ショットができましたので、敵の挙動をもう少しマシにしていきましょうか。

## ザコ敵の挙動

### ベジェーを利用しよう

まず下のザコ敵を見てください。こいつの動きを再現しましょう。
こいつは下からやってきて宙返りして上から突っ込んできます。なお、宙返り後に体制立て直してるかというとそうじゃなくて、


裏返しのまま突っ込んできます。動きとしてはつの字の動きをしているようです。

さて、これをどうやって再現しましょうか…。状態遷移でやっても良いんですが、その場合動きが不連続になってちょっと美しくないです。

そこでベジェ曲線(3 次)を使います。

ベジェ曲線というのは…
http://blog.sigbus.info/2011/10/bezier.html

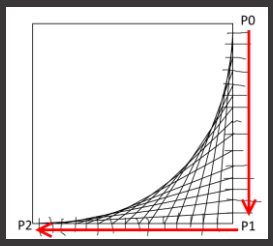
こういうやつで、ぶっちゃけると一番簡単な「パラメトリック曲線」です。スプラインは色々と種類があって、それぞれ数式と挙動がまるで違うので、説明には向いてないと思いました。

(B-スプライン,cutmull スプライン等…)スプラインに興味がある人は
http://hexadrive.jp/lab/demo/597/
でも見とくと良いと思います。みんなも今後はこういう資料を自分で漁るようになればいっぱしのプログラマですね。

ベジェってのは所謂線形補間を応用したものです。線形補間ってのは A 点と B 点があったとして、それぞれの座標を$P_A, P_B$とすると、A と B を結ぶ直線上の、AB 間の点は全て

$$M = (1 - t)P_A + tP_B$$

で表せることは分かりますよね？所謂内分比というやつです。t を 0～1 へと変化させていけばはじめは$P_A$で、中間地点では$0.5P_A + 0.5P_B$で、最後には$P_B$になりますよね？ベジェってのはこれを二点間ではなくて、三点間以上にしたときの線形補間みたいな感じです。




こんな感じに P0,P1,P2 という点を用意して、P0 は P1 へ移動し、P1 は P2 へ移動する。で、移動しながら P0→P1 の途中の点と、P1→P2 の途中の点をさらに補間しながら結んでいくわけ。結果として

$$P = P_0(1 - t)^2 + 2(1 - t)t\,P_1 + P_2t^2$$

という座標が求められるわけ。

まあそれはともかく3次ベジェなんだけど、

を作るための制御点は

こういう箱型になってればそれっぽくなります。さて、実際の数式というかプログラムですがおもったよりもとっつきやすいと思います。ちょっとだけ思い出してほしいのですが

$$(a + b)^3$$

って展開するとどうなってましたっけ？

$$a^3 + 3a^2b + 3ab^2 + b^3$$

こんな感じだったんではないでしょうか？3次のベジェはこれにそっくりな形になります。

制御点が4つありますので、$P_0, P_1, P_2, P_3$ の4点を想定します。なお、間にある$P_1$と$P_2$を曲線が通ることはありません。基本的には始点と終点だけを通過します。
で、ここでパラメータとしてtが登場します。a=(1-t)とし b=t とするようなものだと思ってください。そうすると

$$P = P_0(1-t)^3 + 3P_1(1-t)^2t + 3P_2(1-t)t^2 + P_3t^3$$

となります。
意味は、分かりますか？これをプログラムで書くのは意外と簡単だと思いませんか？

ベクトルクラスを作ってればこの制御点情報を4つ持たせておいて、あとは t を 0→1 に変化させていけば良いわけです。

で、今回はもう完全に敵が YZ 平面上を動くように定義しておいて、制御点は固定とします。
で、X 座標はマップ情報から生成した値とします。

最初～の方に作った Geometory.h+.cpp に Calculate3BezierPosition を作りましょう。
4点のコントロールポイントとパラメータ t を与えて、結果として該当する座標を返すという関数です。

```
///ベジェ補間された座標を計算する(３次版)
///@param pos0 制御点0
///@param pos1 制御点1
///@param pos2 制御点2
///@param pos3 制御点3
///@param t 補間パラメータ(内分比)
///@return 計算(補間)された座標
///@attention tの範囲は0～1に設定してください。
Position3 CalculateBezier3Position(const Position3& pos0, const Position3& pos1, const Position3& pos2, const Position3& pos3, float t);

///ベジェ補間された座標を計算する(３次版)(配列版)
///@param poses 制御点の配列(要素4)
///@param t 補間パラメータ(内分比)
///@return 計算(補間)された座標
///@attention tの範囲は0～1に設定してください。
Position3 CalculateBezier3Position(const std::array<Position3,4>& poses, float t);
```

ちなみに array ってのは固定長配列に使うもので、上のように使うと、ホントに固定長配列のように使えるので便利です。(余裕がある人は2次ベジェバージョンも作っておきましょう)

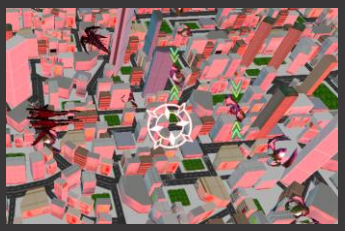
ちなみにベジェ実装は、数式通りに実装すれば問題ありません。

```
//tが0～1の範囲外ならクラッシュさせる
assert( 0.0f<=t && t<=1.0f );
float r = 1.0f - t;
return pos0*r*r*r+
       pos1*3*r*r*t+
       pos2*3*r*t*t+
       pos3*t*t*t;
```

このように、割と簡単に実装できます。余裕のある人は2次ベジェも作っておきましょう。どっかで使うと思います。

ともかく敵の動きとして、最初に t=0 の状態から始まります。つまり先の敵の例で言うと、後方-600 くらいから発生して、前方 600 くらいに行ってさらに-600 へ帰っていく…というふうにするとします。つまり…


4 点をこのように設定します。
X の部分がアスタリスクなのは、今回は YZ 平面でのみ考えようと思うからです。うまく行けば



スクショじゃ伝わりにくいですが、この時の動きを見せてくれます。

## ボス戦

- さて、ボス戦を作っていきましょう。
- ボス戦では弾を出していきましょう。

- 簡単なアニメーションもさせてみましょう。
- ちょっとアタマも使わせてみせましょう。
- ボス戦ではスクロールや、敵の読み込みを止めましょう

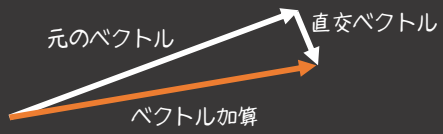
## 中ボス戦について

さて、まずはヘリコプターを中ボスとして登場させましょう。モデル自体はサーバの Enemy のフォルダに入れています。

動きとしては左上から登場→ぐるぐる回りながら弾をばら撒く→自機がいる所に突っ込んでくる→回りながら弾をばら撒く→突っ込み…を体力がなくなるまで繰り返す。

このくるくる回るってやつ。必ずしも sin 関数や cos 関数が必要ではない。


回るのに？三角関数要らないの？マジで？マジよ。予め必要なのは直交するベクトルのみ。直交するベクトルなんて外積使えば求められます。




そして、この加算ベクトルを正規化し、毎フレームこれを行ってあげれば回転していくベクトルが生成できることはわかるだろうか？進む方向がすこしずつ右に偏っていくんですね。これで回転状態になるわけです。

ちなみに正規化のコストは sin と cos 使うのとどっちが早いんですかね…。ネットで調べてみたところ
http://nonbiri-tereka.hatenablog.com/entry/2014/02/11/120634
あら、
sqrt>=sin
だわ。sqrt どんだけ遅いねん。
http://d.hatena.ne.jp/yatt/20090619/1245407569
にも書いてあるけど、sqrt 処理はすっごい重いみたい。sincos でも良いんじゃないかな。

まぁ多分、今回の例程度ではそこまで気を使う必要はないとは思うんですが、一応 sqrt はそ

こまで重いというふうに思っておいてください。
そうは言っても、
http://taustation.com/math-class-method-execution-time/
このサイトだとsqrtの方がかなり処理コストが安い。実装系によるのかな。この辺に関しては一概に正解が云々いえないので、今は好きに作って良いと思います。

今回はとりあえずは「回転運動をするのにsin,cosは必ずしも必要ない」という事を知っておいてもらえれば良いんじゃないかなと思います。

ぶっちゃけ今回のやり方の場合、Y軸中心回転であればXZ入れ替え法だけで対応できるし、面白いんじゃないかな。

このやり方のデメリットとしては、回転中心を特定しづらいということですかね。

## 中ボス登場時にスクロールを止める

これは簡単ですよね。マップを1列1列読み込む時に、そのデータに中ボスがいたらそれ以降の敵データの読み込みを止める。
あと、背景のスクロールも止めましょうか。東方とかだと背景スクロールは止まらないんですが、レイストームだと止まってるので、そこは再現したいと思います。

```
if(!_stage->IsEnd() && _bossEnemy.expired() && ++_frameCount%30==0){
		auto linedata=_stage->ReadLine();
		for(auto& data:linedata){
			if (data.enemyType == (unsigned char)EnemyType::helicopter || data.enemyType == (unsigned char)EnemyType::tank) {
				_enemyfactory->Create(static_cast<EnemyType>(data.enemyType)).lock()->SetPosition(Vector3(data.posX, 100, 500));
				_bossEnemy=_enemyfactory->EnemyList().back();
			}
			else {
				_enemyfactory->Create(static_cast<EnemyType>(data.enemyType)).lock()->SetPosition(Vector3(data.posX, 100, 500));
			}
		}
}
```

分かりづらいけど、ぼくはこんな風に管理しています。

## 戦車の砲塔等の回転アニメーション

戦車などのモデルはフレームと呼ばれる階層構造で出来ています。
例えば砲塔を回転させたいなら

```
int no = MV1SearchFrame(_model->Handle(), _T("砲塔"));
MV1SetFrameUserLocalMatrix(_model->Handle(), no, MGetRotY(angle));
```

このようにまず、フレーム番号を検索し、その番号に対して回転命令を出すことで、砲塔だけ回転させることができます。

名前の判別方法に関してですが、DXLibModelViewer にて確認できます。

んで、上の方法だけだと、砲塔の座標がおかしなことになります。これは階層構造のことを理解していないと少しむずかしいのですが、ともかく親からのオフセットがリセットされてしまうことによって発生します。

これに対処するにはもとの行列を取得し、それに対して回転行列を適用してやるということが必要になります。ですので

```
int no = MV1SearchFrame(_model->Handle(), _T("砲塔"));
auto rot=MV1GetFrameBaseLocalMatrix(_model->Handle(), no);
MV1SetFrameUserLocalMatrix(_model->Handle(), no, MMult(rot,MGetRotY(angle)));
```

とでも書くのが良いと思います。

## 敵マップデータ読み込み時の注意点

ちょっと敵マップデータ読み込み時にアホな事をやってたので共有しておきます。

```
for(;_currentRit!= _data.rend();++_currentRit){
	EnemyPositionData e={*_currentRit,count*100-600 };
	if(e.enemyType==0|| e.enemyType == 255){
		if (++count == _mapWidth) {
			++_currentRit;
			break;
		}
		continue;
	}
	ret.push_back(e);
	if(++count==_mapWidth){
```

```
                ++_currentRit;
                break;
        }
}
```

このような感じで読み込みの部分を作っていたのですが、赤字の++_currentRit を書き忘れていたんですよね。
そうするとどうなると思いますか？

次に読み込んだ時にイテレータが進んでないため二重読み込みとなり、どんどんデータがズレていくのです。

## Background クラスを作ろう

```
//背景描画
for (int j = 0; j < cityparts_y_num; ++j) {
        if (_partsZ[j]-scrollZ<cityparts_z_limit) {
                _partsZ[j] += 500 * 4;
                for (int i = 0; i < cityparts_x_num; ++i) {
                        _cityParts[j][i] = _cityModel[rand() % 4];
                }
        }
        for (int i = -cityparts_x_num/2; i < cityparts_x_num/2; ++i) {
                MV1SetPosition(_cityParts[j][i+3], VGet(i*250.0f, 0, _partsZ[j] - scrollZ));
                MV1DrawModel(_cityParts[j][i + 3]);
        }
}
```

## ブラッシュアップ

さぁ、ここまでで基本はできたと思うので、ここからはブラッシュアップ的なところだ。

## UI

まず、今は全然画面に出ていないUIを整備していこう。UI画面は奥行き関係ないので、深く考えずに2Dとして表示しましょう。
PlayingScene 内に DrawUI とか言う関数を作って全てそこに突っ込みましょう。
必要と思われるものは

- スコア
- 自機の体力
- ボスの体力

は必要かなーと思います。まず、スコアですが
このようなものを用意しました。言っておくけどMSゴシックなんか画面上に表示するなよ。
自分で「これがカッコイイ」ってフォントがあったらそれを使ってもらってかまわないけど

とにかく「MSゴシック」「MS明朝」は使うな。
自分がゲーム制作初心者だって言ってるようなもんだ。というわけで大抵の場合はそのゲームのために作ったフォント(画像)を用いる。コンシューマ機の場合、フォントのインストールなんて不可能だから画像を用います。

それをこんな感じで表示したいとします。
…どうされますか？

まず、桁数を求める必要がありますね。どうやって求めましょうか？ここで数学的な知識が必要になります。

対数って知ってますか？
https://ja.wikipedia.org/wiki/%E5%AF%BE%E6%95%B0
この中でも「底」が10の所謂常用対数というのを使って桁を出すことができます。常用対数ってのは簡単に言うと…

$$\log_{10} 10 = 1, \qquad \log_{10} 100 = 2, \qquad \log_{10} 1000 = 3$$

こういう感じで「ちっちゃい数字」を何乗したら「でっかい数字」になるのかを計算する関数です。このちっちゃい数の事を「底」と言います。この底が10の時を常用対数と言い、eの時を自然対数と言います。ゲームでは10か2しか使わないと思いますが…。

まぁ、ともかくこのlogを使えば桁数は簡単に求められそうです。10は2桁、100は3桁なので、スコアの桁数を求めるならば

int keta=log10(score)+1;//ルいや、桁数ってのはdigitって英語があるんですけどね。
とりあえずここはketaで。

さて、桁数がわかりました。実際に文字画像で置き換えていきます。今回の画像は幅41で高さが320です。正直今回幅はどうでもいいし。問題は高さだ。

高さが320で数値の種類は10個あるわけだ。

ということは1つあたり41×32になる。

そこまではいいかな？そうしたらスコアの数値のとおりに表示するにはどうしたら良いだろうか？自分で考えてくれよ？

簡単だと思うんだが、桁が上がれば上がるほど左に行くんだから…for文で桁数回しながら、桁を上げていけばいいのです。

```
int keta=log10(score) + 1;
for (int i = 0; i < keta; ++i) {
        ここで数字表示(Draw命令)
        score /= 10;
}
```

## 円形ゲージ

東方の敵とかが円形のゲージを使ってたりすると思いますが、本来はこれ相当めんどうなんですが、DXLIBでは、
DrawCircleGauge(spos.x,spos.y,per, _circleGaugeImgH);
を使用します。公式のヘルプには載ってないので注意しましょう。恐らく付け足しのライブラリとは思われますが、

# 課題提出

課題の提出をしてもらいます。
提出期限 8/9(水)AS0 ゲームショウの翌日１８：００厳守(これを超えると一切受け付けませんので早めに提出しておいてください…今日にでも)
提出場所：¥¥132sv¥game¥644 教室¥川野¥3D シューティングゲーム
課題要件：
①3D の表示ができていること
②入力によって自機が移動すること
③ステージ(敵の配置)データを外部から読み込み、時間ごとにそれを出現させていること
④ショット(できればホーミングレーザー)を実装していること
⑤画面上にエフェクシアを用いたエフェクトを表示していること

言っておきますが、これ、最低ラインです。これが出来ていなければ採点しません。出来が良ければ点数を上げていきます。