ACTION GAME

LET'S ACTIONGAME PROGRAMMING 2016

次の話はアクションゲームだ…

シューティングは『最初の』実習だったので、かなりこちらのやることをなぞってやっていただいていたが、次は完全に自分で考えて、自分の頭で作ってもらう。僕もみんなと同じペースで作っていくけど、決してコードは見せません。

とはいえ、あのシューティングゲームをなんとか作ってきて、よくもここまでついてこれたものだ。腹立たしいまでに優秀である。だがもっとも望ましい形に進んできているのはとても愉快だ。

我がゲームプログラミング教育計画は君らの強い命を以ってついに完遂されることとなる。

内容

アクションゲームとシューティングゲームの違い

アクションゲームとシューティングゲームの違いってなんでしょう…。
思いつく限りリストアップしてみます。

①重力がある
②プレイヤーのアクションにジャンプがある
③しゃがみがある
④「足場」がある
⑤スクロールするしないを自分で決められる(強制スクロールではない)
⑥アクションパターンが多い(これが面倒)

という事です。
ぶっちゃけた話、それほどの違いはないです。なのでシューティングゲームをしっかり作れていれば、それほど難しくはないと思います。

クラス設計

最初に「これはクラス化しておいてまとめておいたほうがいいもの」を思いつく限り列挙しておきます。

- プレイヤー
- カメラ(スクローラー)
- 敵(派生あり)
- 敵飛び道具
- ステージ
- スコア
- UI
- シーン
- 当たり判定
- サウンド

くらいでしょうか…。うえーん多いデチー(>_<)とか思ってる人は、世の中のオープンソースのアクションゲームの中身でも見てみるといい…。まぁ最初から全部クラス化する必要もないけど、ある程度しておくと後が楽になるよ(料理の下ごしらえと一緒な…カレーができあがってからジャガイモの皮をむく奴あいねーだろ?)

まぁ難しければ、クラス化などという強力だが難しい道具は使わなくてもいい。使うにせよ使わないにせよ君の自由だが、強力な武器がなくなるという事はどういうことかわかるよな?

それでもよければどうぞ？

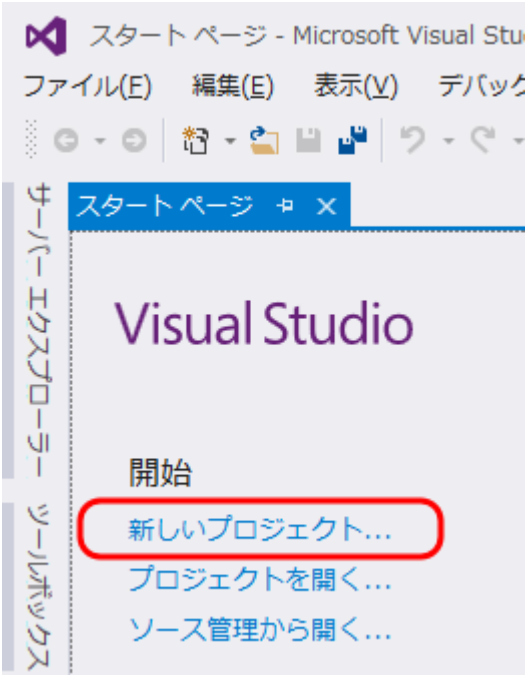
まぁそんな整理されてないクソコードは僕も見たくない。

プロジェクトを作る

シューティングの時と同じだけど、一応書いておきます。書いてることはシューと同じです。
現行の VisualStudio は 2015 が最新であるが、学校で使用している VisualStudio は一部が 2013 しかインストールされていなかったり、2015 がインストールされていなかったり、両方インストールされているのが現状です。
ちなみに言うと既に 2016 版のがテスト版として出ているという噂なのだが、まだ噂の段階であり、おそらくは 2016 年末か 2017 となるであろう。

それはともかく、こういった場合どちらのバージョンで作ればいいんでしょう…ぶっちゃけ「どっちでもいい」んだけど、自分ちで開発する場合は自分ちのバージョンと合わせておこう。
合わせなくても何とかなるけど、合わせておいたほうが良いだろう。

ちなみに Unity のデバッグは 2015 を要求されるので、2015 が入っている所は多いはず。なので説明は 2015 でやらせていただきます。担当する教室によっては 2013 かもしれないけど 2015 でやります。だけど 2013 と同じなのであまり気にしないでください。
とりあえず立ち上げたらスタートページにある「新しいプロジェクト」をクリックするか



もしくは、ファイル→新規作成→プロジェクトをクリックしてください

初回はここで時間がかかるかもしれませんので、慌てずに落ち着いていきましょう。

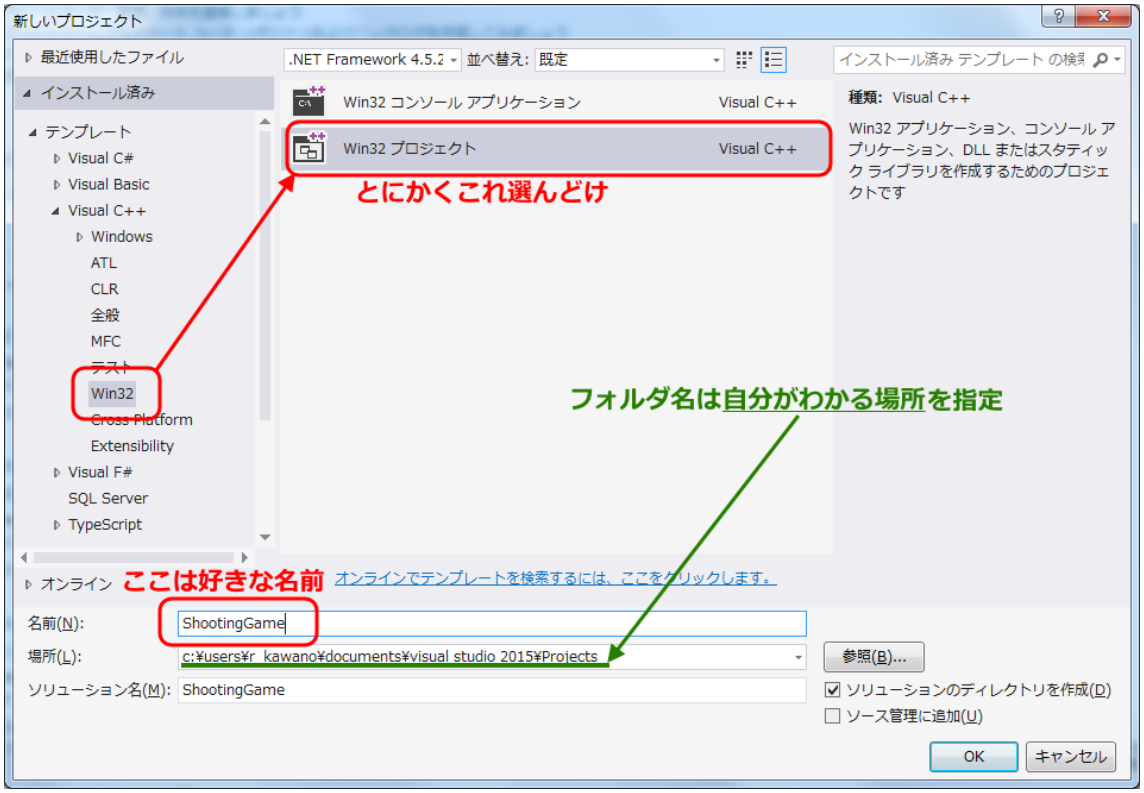
人の話は注意深くしっかりと聞いておきましょう。

説明書は隅から隅までキチンと読みましょう。

プログラムの言語仕様、API のマニュアルは必ず読みましょう。わからない部分があったら必ず調べましょう。

新しい外国語を学ぶ時と同じ気持ちで開発にとりかかってください。
なんでこう言うかというと、プロジェクトの作成時にしくじると、結局最初から作りなおしになるので落ち着いて、よく話を聞いてよく読んでやれってこと。パッパラパーって作って後から「動きませーん」って言われても困ります～。だって40人がそれ言い始めたら授業にならんし…と言う事でよ～く聞いとけ。

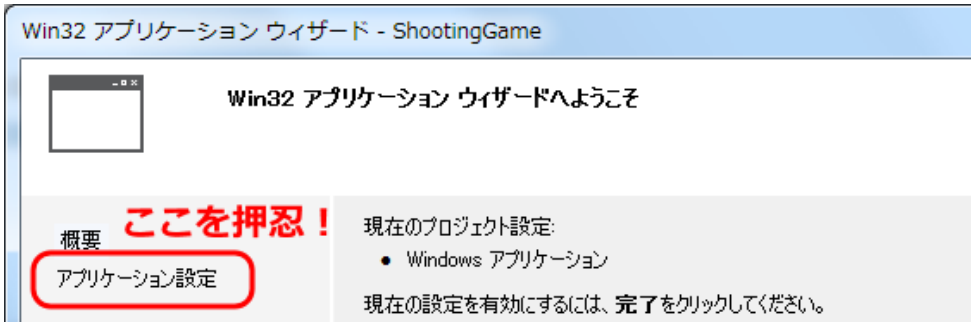
こういう画面が出てると思います。



「場所」には自分で何処かわかるパスを指定してください。
これでオッケー押してからが勝負だ。ここが間違えやすい。

VisualStudio の設定

VisualStudio の設定というよりプロジェクトの設定なんだけどな？とりあえず次の画面で



「アプリケーション設定」を押してください。
うっかり完了ボタンを押した慌てん坊さんはやり直しです。
アプリケーション設定を押した後の画面が大事です。ここ間違えたらやり直しですよ？

アプリケーションの種類:Windows アプリケーション

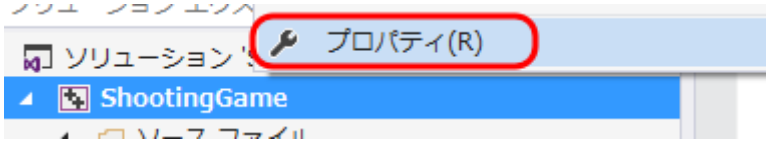
「空のプロジェクト」にチェックを入れる。ここまでやってから「完了」を押してください。

正しく出来てればソースコードの追加です。

「ソースファイル」で右クリック→「追加」→「新しい項目」で cpp ファイルを追加してください。

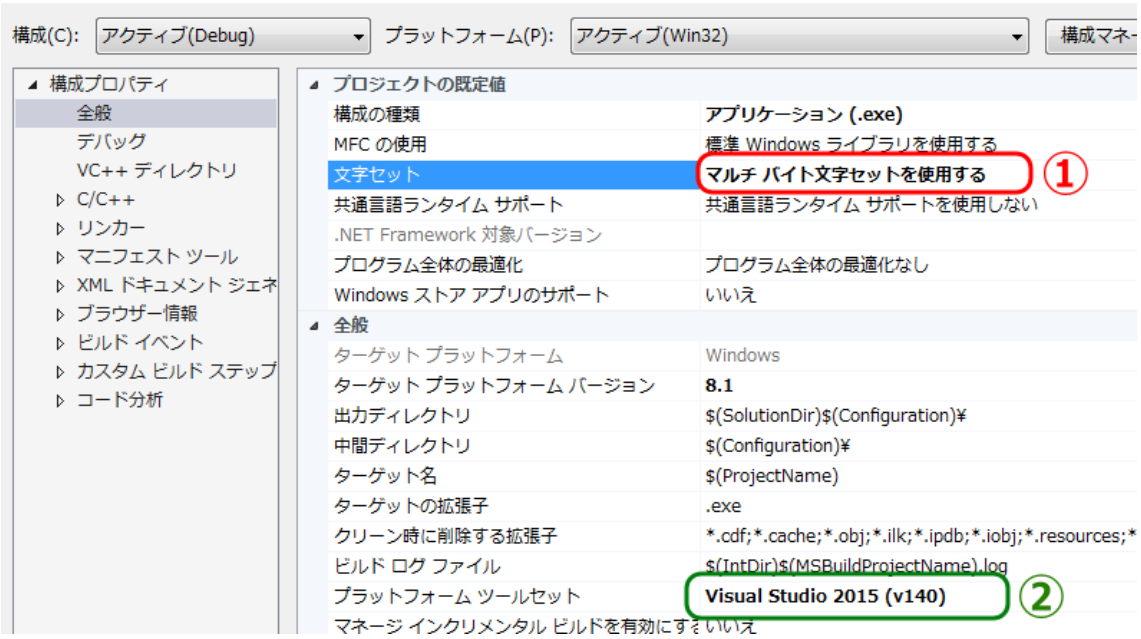
とりあえず main 関数を持つソースコードなので main.cpp ってな名前にしてください。わかってる人は別にどんな名前にしても構いません。

ソースコードを追加したらプロジェクト本体の設定に入ります。



プロジェクト名のところで右クリック→「プロパティ」を選択するとプロジェクト設定画面が出てきますので一気に設定していきます。

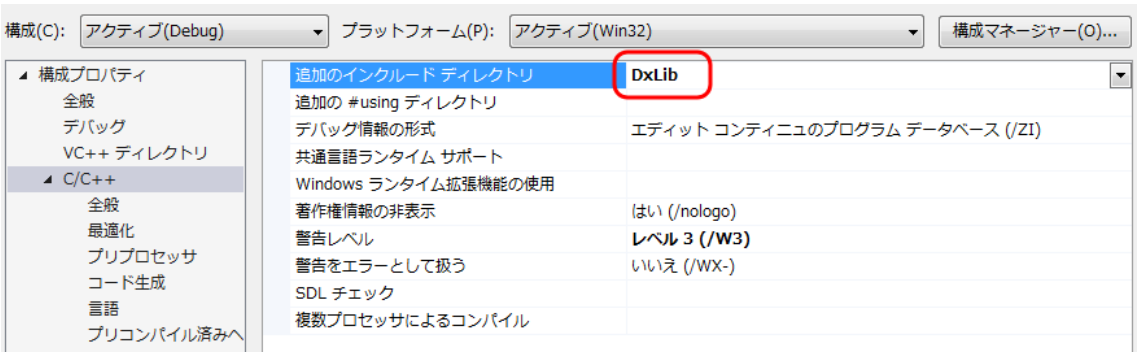
まずは「全般」を見ましょう。



変更すべきは「文字セット」。これを Unicode から「マルチバイト文字セット」にしてください。Unicode でも行けるんですけど、そのままだと文字列リテラルを使う際に文字列の先頭に"L"を付ける必要が出てきます。慣れれば分かるようになるんだけど、面倒なので「マルチバイト文字」にしておきます。

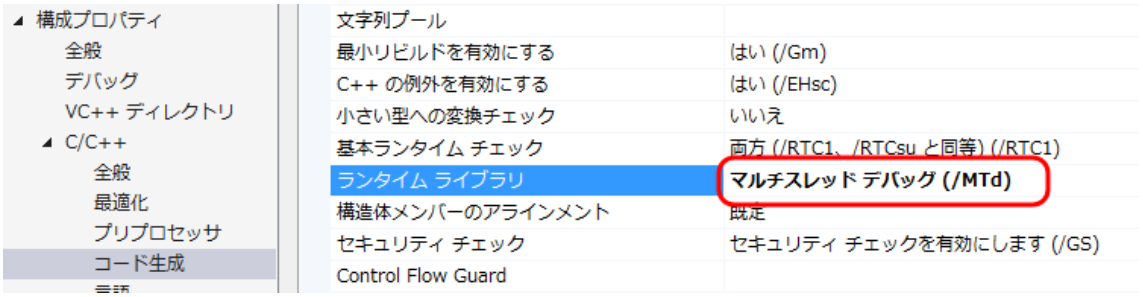
ついでにプラットフォームツールセットについて話しておきますが、もし VisualStudio2015 と VisualStudio2013 を共存させたい(つまり学校で 2015、家で 2013 もしくはその逆の状況)なら VisualStudio2013 としておきましょう。

ああ…次は「C/C++」の設定だ。とりあえず「全般」を押して



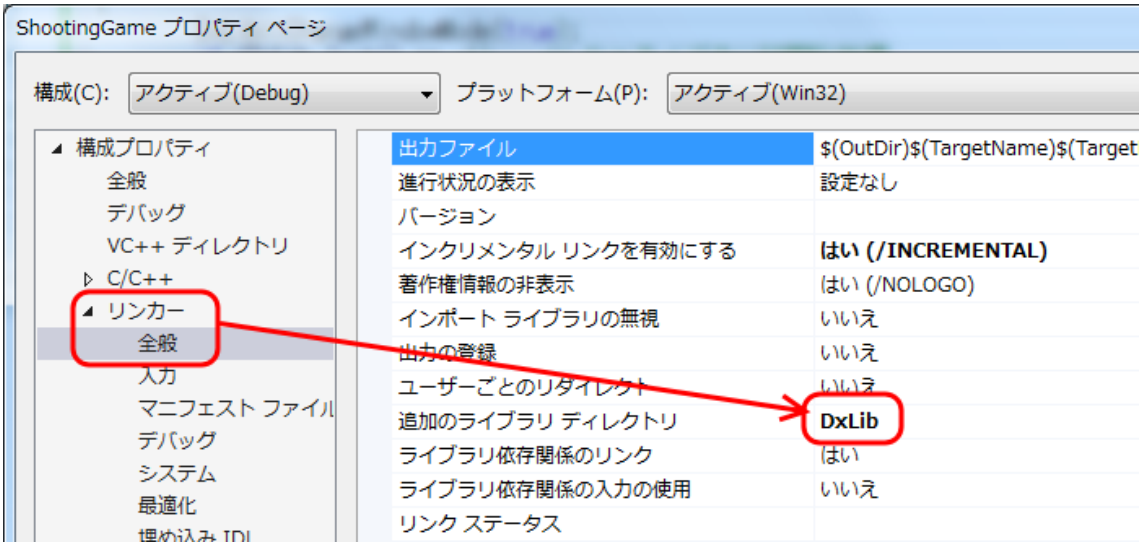
「追加のインクルードディレクトリ」に DxLib と書いたら「全般」の設定は終わりです。

次に「コード生成」を押してください。



ここで「ランタイムライブラリ」が「マルチスレッドデバッグ DLL」となっている部分を「マルチスレッドデバッグ」に変更。

これで「C/C++」の設定は終わり。次に「リンカ」の設定です。もうひと踏ん張りです。



簡単です。

「リンカー」→「全般」→「追加のライブラリディレクトリ」に DxLib と入力してください。

さていよいよ…

下準備(いつもの)

これもシューティングゲームの時にも書いたのと同じものですが、忘れてる人もいるので一応書いておきます。思い出しながらプロジェクトを作りましょう。
自分の記憶を過信しないようにね。ここで「センセーなんかおかしいです」っていう人の大半は、「一回作ったんだから読む必要ねーだろ」って言って、自分でテキトーに作った結果そうなったのです。
「なんかおかしく」なったらぜひとも**センセーを呼ぶ前にテキストを読みましょう**。

DxLib をダウンロードして自分のプロジェクトに

今一度言いますが「ゲームエンジン」じゃありません。「ライブラリ」です。
http://dxlib.o.oo7.jp/dxdload.html
にありますので、ここからダウンロード(VisualC++用)するか
¥¥132sv¥gakuseigamero¥library¥DxLib
から取ってきてください。自己解凍形式なので exe をダブルクリックすると解凍されます。
解凍したら DxLib_VC ってフォルダができていると思いますので、その中の
「プロジェクトに追加すべきファイル_VC 用」というフォルダ内の全てをコピーしてください。

次に自分のプロジェクトのフォルダに移動します。

なに？「自分のプロジェクトのフォルダが見つからない？」だから
「自分で何処か分かるパスにしろ」って言ったのに…もう知らんがな。まぁ、大抵の場合はマイドキュメントの中に VisualStudio2015 ってフォルダがあるから(人によっては VisualStudio2013)
その中を見てみろよ Projects ってあるだろ？
その中に入ってもそれっぽいものがなければ、検索かなんかで探してくれ。

さて、ここからは自分のフォルダが見つかった前提で話をしよう。

その自分のプロジェクトのフォルダを降りて行って、cpp ファイルがある所まで下りてみよう。
下りれたかな？

ではそこに

DxLib

という名前のフォルダを作って、先ほどコピーしたライブラリをそこに貼り付けてくれ

コーディング準備

おまたせいたしました。つまらなかったでしょう？やっとコーディングに入ることができるようになりました。

最低限スケルトン

さて、そんな僕でも一番最初の部分だけは「見ないと」できないと思います。とりあえずこのコードの通りに書いてくれ。

```
#include <DxLib.h>

int WINAPI WinMain(HINSTANCE , HINSTANCE ,LPSTR , int ){

	DxLib::ChangeWindowMode(true);//これを書いてないとフルスクリーンになっちゃう

	if (DxLib_Init() == -1){	// ＤＸライブラリ初期化
		return -1;	// アプリケーションの終了(以上終了時は0以外を返すのがルール)
	}

	DxLib_End();		// ＤＸライブラリの終了処理(初期化と終了処理は必ずペアで)

	return 0;		// アプリケーションの終了(正常終了時は0を返すのがルール)
}
```

あ、コピペろうとしても無駄ですよ？これ、画像ですから。
あと、僕の授業では基本的には「コードのコピペ禁止」です。コードの移動ならまだ許しますけどね…。とくに文法が自分の血肉になっていないうちのコードコピペは害悪です。

タイピングの練習だと思ってさっさと打ち込んでください。

実行すると一瞬ウィンドウが出てそして終了しましたね。

それで正解です。

あと、僕のコードの注意点としては「僕はとんでもなくC++er」だということです。
この学校の中で多分**いちばんC++やってます。**
ところが皆さんの大半は(2 年生は)、C 言語なんですよね？C++は全然やってませんよね？ちょいちょい僕は無意識に C++のコード書いちゃうので、意味がわからない部分は遠慮なく聞いてください。

ちなみに昔、C 言語やってた人用の C++のテキストも書いたので学習意欲の高い人は見といて

ください。

¥¥132sv¥gakuseigamero:¥rkawano¥3 年 C++¥C++.pdf

ってなところに原稿を上げているので、興味のある人は見といて。ただしこれ数年前の原稿なので今流行の C++のラムダ式だの、nullptr だの新しい STL の機能などは書いてません。そこら辺にまで興味が出たら Google センセーに聞くか、僕に直接聞いてください。

ちなみに、C++は奥が深すぎてネットの情報のいくつかは結構嘘が混じってるので気をつけよう。それくらい完璧に理解するのは難しい言語なのだ。
だから「完璧」は目指さないほうが良い。C++の生みの親のビョーンストロウストラップ氏ですら「もう手に負えない」って言ってるくらいだ。

でも初歩的なところだったら、ちょっと勉強すればできるし、とっても便利だ。50%の理解を目指して頑張ろう。

スケルトンの解説

#include<Dxlib.h>
これは DxLib.h ヘッダーをインクルードすることにより、DxLib 内の関数を参照可能にしている。
ちなみに#include には"DxLib.h"ってやる方法と<DxLib.h>ってやる方法があるが、これの違いは分かるかな？
ヒトコトでいうと""であれば現在のソースコードから見た相対パス。
<>であればプロジェクトで設定されてる場所からの相対パス
ってこと。
「プロジェクトの設定」で DxLib って入力したでしょ？あれはプロジェクトから見て DxLib ってフォルダにパスを通すってこと。
これにより
#include<DxLib.h>
という記載で済んでいるのだ。これを
#include"DxLib.h"
とするとエラーが出る。この場合は
#include"DxLib/DxLib.h"
にしなければならない。
よくわからないとおもう。今はそんなもん。しばらく使い慣れてくると分かるようになる。精

進しなさい。

さて次に WinMain だが、これはコンソールにおける main 関数だと思っておいてくれ。

```
#include<stdio.h>

int main(){
        printf( "Hello World¥n" );
        return 0;
}
```

とかってやったでしょ？これの「main」と同じよ？main はコンソール用で WinMain はウィンドウアプリ用って思っておいて？

さて、WinMain 関数にはルールがある。引数を4つと戻り値が1つ必要だ。
こういうルールは Microsoft などが管理している。こういうのに出会ったら必ず

「Google で検索して調べるクセをつけましょう」

調べてもないのに「わからん」「動かん」てのは阿呆の言うセリフです。
この調べる時の原則ですけど

「信頼できる情報かどうかを必ず確認しよう」

信頼できる情報とは
公式ヘルプ/公式リファレンスや「マイクロソフト」「nVidia」などのベンダーからの情報です。
要注意なのは、ブログ等でまことしやかに書かれてる情報で、80%くらい怪しいと思っといたほうが良いです。

さて、マイクロソフトの公式とは「MSDN」のことで、例えば WinMain ならば
https://msdn.microsoft.com/ja-jp/library/cc364870.aspx
ここを見ればいいわけです。

さて…

```
int WINAPI WinMain(
  HINSTANCE hInstance,      // 現在のインスタンスのハンドル
  HINSTANCE hPrevInstance,  // 以前のインスタンスのハンドル
  LPSTR lpCmdLine,          // コマンドライン
  int nCmdShow              // 表示状態
```

);
こんなこと書いてますね。
たまにいるんだけど、これをこのままコピペして「動きません」とか言うダイナマイト•バカ。

…だれがこれを写せと言ったよ？

ヘルプに書いてある「これ」はあくまでも関数の型と引数の説明用です。
これをこのまま貼り付けても動きません。

だから、ここで注目すべきは…

- 戻り値が int であること
- 引数が 4 つであること
- WINAPI ってなんじゃらほい？

戻り値については main と同じなんで深くは解説しないけど、予備知識はあるだろうから簡単に言うと main と同様にアプリケーションが正常終了すれば 0 を返し、異常終了すれば 0 以外のエラーコードを返す。そんだけ。

で、次に引数が4つあって、なんか色々と書かれてるけど、DxLib での開発においてはここは使用しません。だけど関数の「型」は合わせておかなければならないので、引数の数とそれぞれの型だけは指定しておきます。そのかわり**引数の名前は必要ない**です。
ですから、型のみを書いて

```
WinMain(HINSTANCE , HINSTANCE ,LPSTR , int )
```

と書いているのです。
さて、最後の疑問の WINAPI ですが、これは__stdcall を言い換えているだけのもので、こいつは何をするのかというと、呼出し後にスタックをクリーンしています。←これについては分からなくていいです。(通常はスタックは巻き戻されるのだが、サブシステムから呼び出される場合は明示的にクリーンしなきゃならんってこと)
ともかく Windows のサブシステムから WinMain が呼び出される際の規約として__stdcall をつけておく必要がある…ただそれだけの理由です。

つまり結論から言うと**「とにかく WinMain を書くときは WINAPI を書け」**ってことです。

こういう「まことしやかな理由はあるんだけど、正直よくわからん」規約もあるので、そこは割りきっちゃってもかまいません。

さて次は
ChangeWindowMode(true)
だが、これは DxLib の関数なので、そこから探してみよう
http://dxlib.o.oo7.jp/function/dxfunc_other.html#R11N1
とりあえずひと通り読んでください。今回の授業を通して、こういう「リファレンスの読み方」を学んでください。辞書の引き方みたいなもんですね。

ちょっと前にも書きましたがこのリファレンスを正しく読んで C 言語の文法を知ってたらそれだけでゲームが作れます。
作れないのは「正しく読んでないから」です。正しく読むとはどういう事かというと

- 隅から隅まで読む
- わからない用語があったらちゃんと調べる

です。まぁあんまり厳密にやってるとモチベーション下がっちゃうのでそこはそれぞれ各人のさじ加減でね。でもあまりにもリファレンス読まずに間違って使って「動きません」が多すぎるんでね…。

さて、この関数の作用を要約すると「ウィンドウモードとフルスクリーンモードを切り替えます」ってことです。true だとウィンドウモード。false だとフルスクリーンモードになります。

だから ChangeWindowMode(true);でウィンドウモードにしているわけです。

こういうところで「注意」とか書いてあったり、赤字で書いてある場合は実際やらかしやすい部分なので、良く読んでおきましょう。

ちなみに以下の部分
＜注意＞
　この関数を実行するとロードしたすべてのグラフィックハンドルと３Ｄモデルハンドル、作成したフォントハンドルは自動的に削除され、SetDrawArea, SetDrawScreen, SetDrawMode,SetDrawBlendMode, SetDrawBright 等の描画に関係

する設定を行う関数による設定も全て初期状態に戻りますので、 画面モード変更後 LoadGraph 関数や CreateFontToHandle 関数等で再度ハンドルを作成し直し、 描画可能領域、描画対象画面等の各種描画系の設定も再度行う必要があります。

読んで分かる人はどれくらいいますか？分からない人も多いでしょう。それでもいいからひと通り読むクセ自体はつけておいてください。これを要約すると

「ChangeWindowMode を呼ぶとハンドル系が初期化されちゃうから注意してね」ってこと。
これがどういう事を意味しているのかというと、フルスクリーンモード⇔ウィンドウモードの切り替えの際にハンドルの初期化が行われてしまうので、それを考慮しながら設計しないといけない。

ともかく今は ChangeWindowMode(true)にしといてくれ。興味がある人は false にしてみよう。

次に DxLib_Init だが…これは「DxLib の初期化」を行っている。初期化ってヒトコトで言ってるけど、DxLib を使用せずに DirectX の初期化を行おうとすると結構大変なんだよね。3年生のみんなは知ってると思うけど。
まぁ…いずれ通らなければならない道なので、覚悟はしておこう。
ともかくいろんなことをやってくれているのだ。
http://dxlib.o.oo7.jp/dxfunc.html#R1N1
とりあえずここでの説明は、詳細書いてないので「初期化してる」って思っとこう。
そして失敗すると-1 を返すと思っておこう。
敢えて注意点、を挙げるとすればこの一文

※ＤＸライブラリを使用するソフトウェアはまずこの 関数を呼び出す必要があります

これをちょっと説明しておくと…
幾つかの例外を除いて DxLib の関数は DxLib_Init が成功した後に呼び出すべき
である。ということ。
幾つかの例外(ChangeWindowMode など)ってのはおいおい説明していくけど、基本的には最初に DxLib_Init を呼び出しておくってのはおさえておこう。

最後に DxLib_End()だが、これは初期化の反対で、後処理ってやつだ。

http://dxlib.o.oo7.jp/dxfunc.html#R1N2

例えば図書館において「本棚から本を取り出したら、元の棚に返すべき」という風に、ソフトウェアにおいても、ソフトウェア終了時にはソフトウェア起動時の前の状態にすべしって原則があるのだ。

DxLib_Init は内部で色々とやっちゃってるので、それを元に戻す作業…それをやってるのが DxLib_End()なわけ。

つまり、
DxLib_Init に始まり、DxLib_End に終わると思ってたら良い。
さぁ、これでスケルトンコードの説明は終わりだ。

早速少しずつゲームを構築していくとしよう。

ともかくウィンドウをキープする(ゲームループを作る)

現段階ではゲームのためのウィンドウが出て…そしてすぐに消えてるので、ともかくこれをキープしよう。
そのために必要なのがゲームループだ

ループなんだから当然ループ文を書くんだが
for と while のどちらを使ったら良いんだろうね？

ちょっと自分でも考えてみよう。ぶっちゃけ正解など無いのだ。

ここで考えのヒントを言っておくと

「ループを作るときは「終了条件」もしくは「継続条件」を意識しよう」

ということだ。

for に適しているのは「回数がループ継続条件となっているもの」
while に適しているのは「true/false が継続条件になっているもの」

です。
true/false が継続条件になっているものがちょっとわかりづらいんですけど、if 文の中が真か偽かっていうのは既にやってますよね？

あれが真=true だったらループ継続し、偽=false だったらループ終了する
というわけです。

ちなみに C/C++では数値の 0 が false(偽)を表し、0 以外なら true(真)を表します。これも覚えとくと他の人のコードを見る時に役に立つと思います。

とりあえず無限ループにするとキープできます。
Init と End の間に

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

って書いてみよう。

うん、そうなんだ(*•ω•*)すまない。単なる無限ループだから固まってしまって×ボタンを押しても終了できないんだ。VisualStudio の「停止」ボタンを押すまで落とすことができない。

謝って許してもらおうなんて思ってない。
でも、ProcessMessage()を呼び出して、ウィンドウを閉じることができたとき、君は
「ときめき」みたいなものを感じてくれたと思う。
サツバツとした世の中で、そういう気持ちを忘れないで欲しい

というわけで ProcessMessage 関数を見てみよう
http://dxlib.o.oo7.jp/dxfunc.html#R1N3

この関数は Windows 環境でのソフトプログラムに付きまとう メッセージループ処理を肩代わりしてくれる関数です。

この関数がなにをしているのか、というのは特に気にする必要はありませんが、**とにかく定期的にこの関数を呼び出してやる必要があります**。

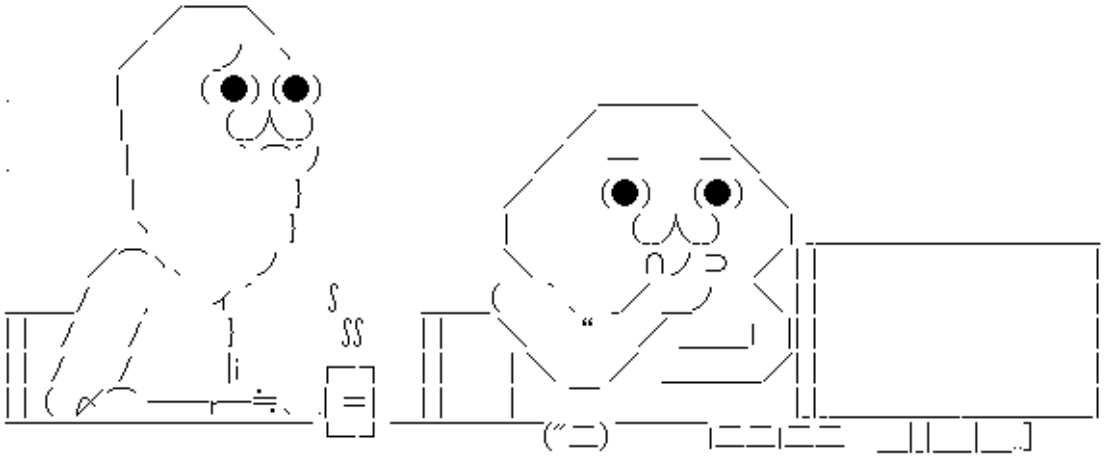
目安としては６０分の１秒に一回程度、用はゲームのループに 一回程度です。この関数を呼び出さないと結果としてシステムが 異常に重くなったり不安定になったりします。

あと、戻り値が-1(エラー発生・若しくはＤＸライブラリのウインドウが閉じられた)になったらなるべく早めに DxLib_End でライブラリ使用を終了し、同時にプログラムも終了する必要があります。(そうしないとウインドウを閉じてもプロセスが残るという事態になります)

メインループ内で毎回 ProcessMessage 関数を呼び出して、こいつの戻り値が-1 の時はすみやかにループを終了するようにコーディングすればいい。
さて、僕が答えを書くのを待っているようではダメだ。

しばらく自分で考えてみてください。

…どうしよう？



素直に書くならこうですね。

```
int ret=ProcessMessage();
if(ret==-1){
        break;
}
```

うんぁぁ…1年生のC言語の時間であればこれで良かったのだろう。

だが2年生以降の人は…「動けばいいってもんじゃない！」

そういう思考をしてください。

「プログラムは可能な限りシンプルに書くべきである」

このプログラムってこれ以上シンプルになりませんかね？不必要な物はありませんかね？

ちょっと考えてください。

そう…retは必要ないですね。ProcessMessageの結果をretに入れて、それを-1かどうか判定している。これは
ProcessMessage の戻り値をそのまま判定すればいいんじゃね？って思えばいい。そうすることで

```
if(ProcessMessage()==-1){
        break;
}
```

と書ける。さぁ一行減った!!

まだ減らせませんかね？もう少し広く見てみよう。

```
while(true){//無限ループ
        if(ProcessMessage()==-1){//←ループ終了条件
                break;
        }
}
```

while は終了条件を満たすまでループするんでしたね？
そして終了条件は今 if 文の中にある…つまり if 文の中身を while の条件式に入れればいいわけだ。

つまり

```
while(ProcessMessage()!=-1){
        ;//ここに毎フレームの処理を書く
}
```

と、ここまでシンプルにすることができるわけ。こういう風に「思いついたコード」を「落ち着いてシンプルにしていく」事を心がけてください。

今回はアクションゲームを作っていきますが、どんなアクションゲームを作りましょうか？
正直、何でもいいです。

スパルタンX

ひとまずは「スパルタン X」システムでいきましょう。たぶん、僕が好きなアクションゲームの中では最も簡単でシンプルな作りだからです。

- 床が平坦
- 穴とかがない
- 攻撃はパンチとキックのみ
- 1キャラのあたり判定が複数種類ある
- アイテムなし
- 敵のパターンがシンプル
- アニメーションが少ない

こんな特徴があるからです。中でも床が平坦ってのは相当に簡単です。スパルタンができたら次には穴あり、動く床あり、斜め床ありの話をしていきますが、まずはスパルタンシステムを作っちゃいましょう。あくまでもサンプルとしてなので、キャラの絵は変更してください。

https://www.youtube.com/watch?v=Xy6zQWes8gM

アクションの基本を作っていく

アクションゲーム特有で一番難しいのは足場です。これはマジでヤバい。これを突き詰めると恐ろしく難しくなって、シューティングとか簡単だったと思うようになってしまいます。

ですので今回はその前段階。
「重力」
と
「ジャンプ」
を実装しましょう。

まず重力です。ここでは「力」というよりも重力による「動き」を実装します。
重力などというと

$$P_t = V_0 t - \frac{1}{2} g t^2$$

などという忌まわしい式を思い出したりする人もいるでしょうが、ここではそれは必要ありません。
まず、用意してもらいますのは、現在位置を表す変数と速度ベクトルを表す変数です。

何度言ってもよくわからない人がいるので、言っておくと重「力」とは加速度を生み出します。
ですから、重力による加速度を「重力加速度」なんて言ったりします。

そこで考えましょう。「加速度」とは何なのでしょう？

加速度について

読んで字の如く、速度に加える値です。速度に加えるのです。オーケー？

例えば速度がｖだとして、加速度がαだとすると、速度ｖに加速度αを足すとｖ+αですよね？
ただ単にこれだけの話ですよ？難しいかい？足し算よ？

ただし、v+αは一回ぶんの話です。

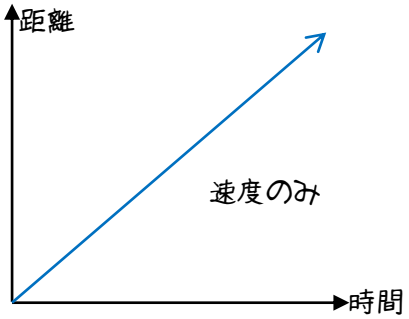
加速っていうのは一回でおわりゃしない。「止める力」がなきゃドゥンドゥン加速していくつまり初速がｖだとすると次はｖ+α 次はｖ+2α 次はｖ+3α…となっていくわけ。

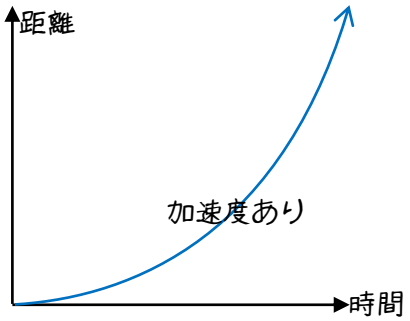
この一回ってのが１フレーム(つまりゲームループ１回につき)なので、毎フレーム

v=v+α;

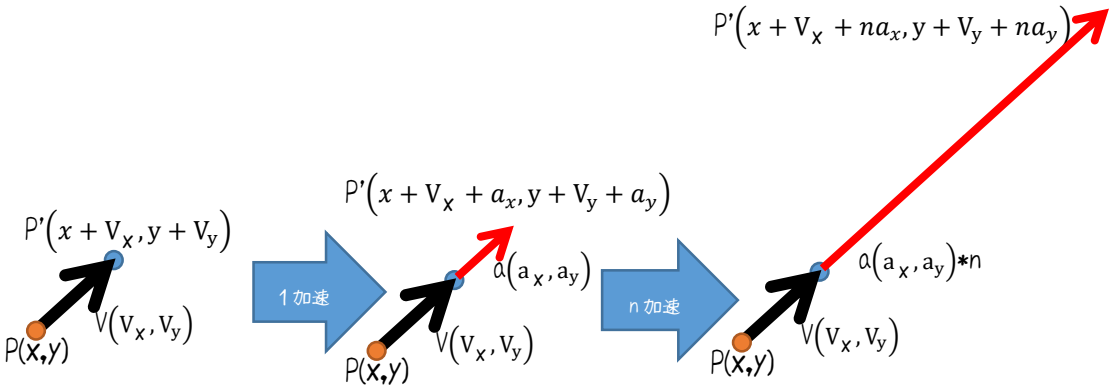
と書けばどんどん加速していくわけ。
さらに「速度」ってなんやったっけ？速度ってのは**現在の位置(座標)に足すことで、移動後の位置(座標)になるもの**やね？
つまりこういうこと

分かるよな？一回で進むスピードがドゥンドゥン早くなりながら、現在座標も変わっていくので、速度オンリーの時と加速度ありの時では





こういう感じで時間と距離との関係が変わってくるここがちょいとややこしいんだが、
プログラムにして書くと意外と大したことはない。
//速度のみ
p=p+v;//現在座標に速度を足している「だけ」



```
//加速度あり
v=v+a;//現在の速度に加速度を足している「だけ」
p=p+v;//現在座標に速度を足している「だけ」
```

のように一行しか増えてもないし、そもそも難しい数式も書いてない。足し算オンリー

2 行の足し算しか書いてないのに「分からない」とか「難しい」と感じるならばそれは君が目を開いていないからだ。小学生でもわかるだろう？

難しいという先入観で、見えるものも見えなくしているのだ。何故ならそれが楽だから。分からないほうが楽なのだ。

ただ、君の「可能性」は失われていく。本当に難しいものや分かんねーものも確かにある。あるが、分かるものすら見えなくしてどうするんだ…？

見えるものも、君が見ようとしなければ見えないよ？

さて、これで加速度というのが分かったと思う。わかんなくても、だいたいの実装はできるんじゃないかな？

じゃあ重力加速度はプログラミングにおいてどう実装するのか？

X 方向

重力は Y 方向にしか働きません。そこがミソです。X 方向と Y 方向を分けて考えましょう。いつも言っていることですが、ごっちゃにしちゃうとわかんなくなりますんで、1 軸ずつ考えましょう。

X 軸は通常通り右を押したら右に、左を押したら左になるようにしておいていいです。
このとき、シューティングと違うのは、左右に動いたときに画像を反転する必要があるってことです。
DrawGraph 系列の最後の引数が、turnFlg になっているので、リファレンスを読みながら向きを反転させてください。例えば DrawRotaGraph2 なら

```
DrawRotaGraph2(x,y,中心 X,中心 Y,拡大率,回転角,ハンドル,true,反転フラグ);
```

のように最後の引数が反転フラグになっています。

ここら辺が、強制スクロールシューティングとは違う所ですね。右を向いたり左を向いたりしますので、反転フラグを多用します。

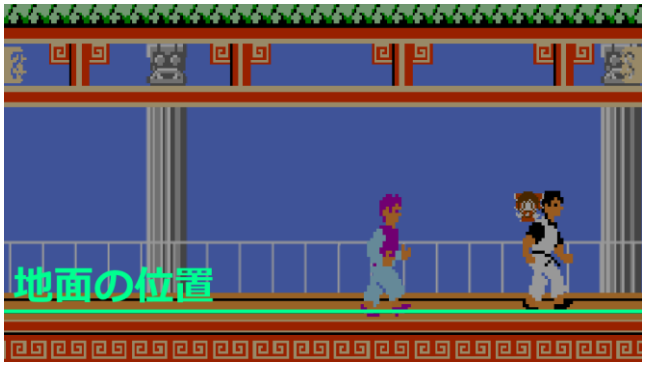
これ以上は説明しませんので、なんとか考えて、右おしたら右、左押したら左に向くようにしてください。

重力加速度の実装

今度はＹ方向ですが、まずはＹ方向の加速度をプログラムします。つまり

```
//重力による落下
vy+=g;
player.pos.y+=vy;
```

当然、これだけだと奈落の底なので地面の制限をつけましょう。



図のように適当な位置を地面として定め、それ以上落ちないようにします。

```
const int groundZero=360;
```

みたいに地面の高さを定め、画像がそれ以上下に行かないようにします。
ちなみに画像の「ボトム(下端)」がってのを忘れないようにね。

ちょっともうシューティングの流用で考えるならば

```
if(_player.GetRect().Bottom()>=groundZero){
        vy=0;//速度をリセット
        _player.SetPosition(x,groundZero);//位置を補正
}
_player.pos.y+=vy;
```

てな風にします。ほんとは速度リセットだけで十分なんですけど、下向き速度が大きすぎる場合はめり込んでしまいますので、groundZero に強制的に位置を補正します。

この辺の話が、地形が平坦じゃない場合はちょっと難しくなってきます。

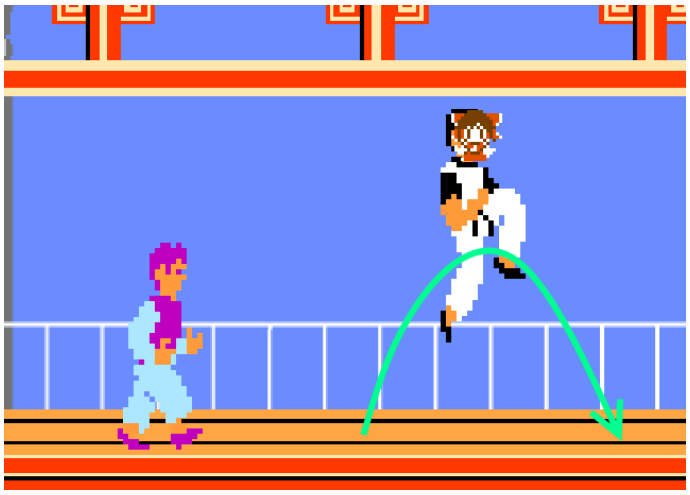
ジャンプの実装

ジャンプの実装も簡単です。
ジャンプの瞬間に強制的に「上向き速度」を代入すればいいのです。

つまり…まぁ、例えば上ボタンでジャンプするとすれば…

```
if(ジャンプ){
        vy=-jumpPower;
}
```

jumpPower は固定値です。自分で調整してみてください。調整は、最初は重力加速度 g をかなり小さめ(0.3 くらい)に、上向き速度も小さめに…。高さが足りなければ、上向き速度を大きくするのか、加速度を小さくするのか、そこは自分で判断しながらジャンプしてるっぽくしてください。



最初はそれで１コマ使っちゃうだろうね…

あたり判定とか、速度とか、幾何学系の武器

実はこの幾何学系の計算はすでにシューティングで使っている。シューティングの奴を応用できるのは

- あたり判定
- 自機に向かってくる処理
- 一定のスピードで進む処理
- 距離を測る処理

これらはシューティングの奴を流用しても応用してもいい。

時間はないのだから、一度作ったものは再利用しよう。
再利用しやすくするために、クラス化や、関数に分ける作業をしておくのだ。
関数化してなかったらほかのプロジェクトの時、また一から作らなければならないだろ？
それは面倒だろ？

で、動きとか、あたり判定はいいんだ。

アクションゲームの面倒くささは、「状態遷移の多さ」にある。

プレイヤーの状態遷移

まず思いつく限りの状態遷移表を書いてみる

入力 状態	上	下	左右	パンチ	キック	着地	無入力時間経過
ノーマル	ジャンプ	しゃがみ	歩き(向きの変更)	パンチ	キック		
歩き	斜めジャンプ	しゃがみ(停止)	歩き(向きの変更)	パンチ(停止)	キック(停止)		ノーマル
ジャンプ中					ジャンプキック	ノーマル	
しゃがみ中	ジャンプ		向きの変更	屈みパンチ	屈みキック		ノーマル
パンチ							ノーマル
キック							ノーマル
ジャンプキック						ノーマル	ジャンプ
屈みパンチ							ノーマル
屈みキック							ノーマル

状態が変わらないものは灰色で塗りつぶしています。
また「無入力時間経過」というのは入力がない状態で前のアクションが終わったことを示します。どういうことかというとパンチを押してパンチ状態が終わるまでに数フレームあるわけです(そうしないと無敵になる)。その終わるときが「時間経過」です。

まぁ、こういう表とか図は自分で書けるようになってほしい。ていうか、今からでも自分で書いてみて、センセーが作ったのと違ったら、どう違うのか考えてみよう。
センセーは必ずしも正しくないのだ。「こういう場合はどうなのか？」実際に考えてそれをプログラムに落とし込んでみよう。

とりあえず「ダメージ」「死」の状態はこの段階では入れてません。最終的にあたり判定とか入った段階で考えます。

ね？この辺がシューティングゲームに比べると面倒なんですよ。でもまだ「面倒」なだけで難しくはないです。ここで難しいって言ってる人は「メンドクサイからやりたくない」深層心理の裏返しだと思っておきましょう。

ここの実装は正直お任せします。僕は例によって状態関数を切り替えて作っていきますが、正直面倒です。
ですから、みなさんは空中フラグとかそういう「状態を表現する」フラグを使って処理を切り替えても構いません。
どちらにせよ面倒なのには変わりありません。ここに関しては。

ただ、どういう方法をとるにせよ「関数化」「構造化」は細かくやっておきましょう。100 行以上のコードとか、同じコードが散見するプログラムを書いていると、絶対後で泣きます。
何度も言いますけど「そんなコード僕も見たくない」です。ご理解ください。

パンチ、キックについての注意点

シューティングゲームの弾と違って、パンチ、キックはその場で留まり続けます。
このため、連打するとパンチキックが出っ放しの状態になります。分かりますよね？

これはまずいですね。あと、パンチとかキックも一瞬しか出ないとすぐひっこめられるんで 10 フレームくらいは継続したい。

つまり連打された場合は

キック 10 フレ→キック→キック→キック 10 フレとするのではなく

キック 10 フレ→ニュートラル→キック 10 フレ
とするようにしてください。

きちんとニュートラルに戻してから次の行動をとるようにしてください。やり方としてはキック中にキックを出せないように…つまりキック継続カウントでやってるなら

```
if (_kickingCountDown<=0 && input.state(KEY_INPUT_Z) && !_prevInput.state(KEY_INPUT_Z)){
```

こういう条件式にしとくといいよ。

パンチキックのあたり判定

この手のアクションゲームは、攻撃時にだけ発生する矩形があるということです。つまり抱えている矩形が複数あるという点がシューティングゲームとちょっと違いますが結局のところ、攻撃有効フレーム間だけ有効な「弾」が留まっていると考えればいいです。

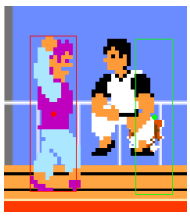
スパルタンXの場合は矩形が＋1しかないのでいつもの矩形＋「攻撃矩形」を用意し、isAvailableで「有効」「無効」を決めて、有効な状態の時に敵に当たればダメージを与えればいい。
相手がつかみ男なら耐久力1なので、当たった瞬間にKillしておk

だから、ゲームのメイン側からプレイヤーに「あなたのあたり矩形は有効ですか？」と問い合わせて、有効ならあたり判定をとればいい。

ちなみにパンチとキックがありますが、共用しても構いません。パンチ関数やキック関数(呼ばれた瞬間にパンチ状態、もしくはキック状態になる)が呼ばれたタイミングで矩形を定義しなおせばいいからです。

しゃがみの実装

しゃがみは特に何も考えずに実装してしまうと…



こういうことになってしまいます。これを防止するには「下に」合わせます。
つまりプレイヤーのDrawGraphをラップして、下に合わせるように書き換えるか、もしくはその都度計算するかして、下をとりましょう。

なお、言ってませんでしたが、DxLib にはグラフィックの大きさを測る関数があって、

GetGraphSize って関数でとってこれます。ちょっとポインタ絡んでるんで難しいですが
http://dxlib.o.oo7.jp/function/dxfunc_graph1.html#R3N13

```
int handle;
int w,h;
GetGraphSize(handle,&w,&h);
```

みたいにして、自動でとってきます。

これを自動でとってくれば

```
DrawRotaGraph2(center.x,あるべきボトム座標-scale*(w/2),cx,cy,scale,…);
```

といった具合で下端に合わせることができます。

また、あたり判定もこれに合わせて低くしなければなりませんので注意してください。
ちなみにナイフ男のナイフをぎりぎりよけれるくらいにしておけばいいです。

雑談①

川野センセーの授業は難しいとか、速いとか、厳しいとかよく言われます。結構聞こえてきますし、直接言われたりするので、知ってます。速いですし、難しいです。厳しいです。自覚してます。

わかる。わかるよー。

いっそのことこの教室から逃げ出してしまいたい…そう思っている人も多いのだろう。ぶっちゃけ僕も嫌われたくはないし、難しいのをする以上は俺も相当勉強しとかなきゃならないし。キツいだけやねん。でもな…でもね、でもでもでもでもね…

センセーのコードを写したり、センセーから言われた通りに作るだけではクリエイターは務まりません。そのために必要なことを今教えています。とくにクリエータ科には時間がないので、圧縮しています。悪いが、専攻科２年もそこと同じ教室で授業をやっている以上は付き合ってください。

さっさとゲーム作れるようになりたいでしょ？

だったら…

とぃぅわけです。頑張ってくださぃ。

つかみ男

こいつは「つかみ男」と言って、プレイヤーをつかんで体力を奪う嫌な奴です。で、こいつの実装。一番の雑魚のくせに難しいんだこれが。むしろナイフ男より面倒。頑張りましょう。
ちなみにはっきり言って俺はこいつの実装は自分で何回も「違う」「違うんじゃあ！」とか言いながら作ってますよ。皆さんも悩んでください。センセーがぜーんぶ答えを知ってると思ったら大間違いですよ？

さて、こいつのパターンはパッと見

- プレイヤーに向かって歩いてくる
- プレイヤーと当たるとプレイヤーは移動できなくなる
- ジャンプ中も当たると移動できなくなる
- レバガチャかボタン連打で外せる

です。一つ一つ細かく見ていきましょう。最初に言ったようにヒントしか言いません。実際のコードなんか見せませんので悪しからず。コードっぽいものがあっても疑似コードみたいなもんですので、そのまま書き写して動くはずがねえだろぅ…。

プレイヤーに向かって歩く

これ、簡単ですよね。生成されたときのプレイヤーとの位置関係からベクトル方向を決めます。
つまりコンストラクト時に

```
float distance = _player.GetCenter().x - _rc.Center().x;
Vx = distance / abs(distance) * speed;
```

とすればいいでしょう。この Vx が X 方向の移動量なのです。
なお、Center 位置で比較している理由はシューティングゲームで言ったので説明は割愛します。左上だとおかしな見え方になるからです。注意しましょう。

プレイヤーと当たるとプレイヤーは移動できなくなる

さて、これを実装するには「衝突による状態変更」を実装する必要がある。つまり「あたり相手」を知る必要がある。この辺もシューティングとは違いますよね。

というわけでよくよく設計を考えましょう。正解はありません。…さてどうしましょうか。この辺はプロでも迷う迷いどころです。
①Rect を内包、もしくは継承した Collider というクラスを作り、Player も Enemy もそっから継承する。Collider のメンバとして、「キャラタイプ」みたいなのを用意して

```
enum CharacterType{
        ct_none,
        ct_player,
        ct_grabman,
        ct_knifeman
                :
                :
};
```

を Collider は保持する。あたり判定の結果の OnCollided の引数として渡してやって、相手の情報を得る。
相手の情報を得るんやったら、dynamic_cast で派生先クラス情報を得る…うーん、そもそもプレイヤーのあたり判定は一つじゃない(やられ判定と攻撃判定)わけだし、ちょっと違うかなー。
このやり方だと、

```
enum CharacterType{
        ct_none,
        ct_player,
        ct_playerattack,
        ct_grabman,
        ct_knifeman
                :
                :
};
```

などとする必要があり、ちょっと違うかなぁ…ホントに自分で考えて？悩んで？
じゃあ僕は CharacterType と ColliderType を分けようかな…。

```
enum CharacterType{
        ct_none,
        ct_player,
        ct_grabman,
        ct_knifeman
                :
                :
};
enum ColliderType{
        col_none,
        col_default,
        col_attack,
};
```

これを Collider に持たせて、かつ持ち主へのポインタか参照でも持たせておきます。

ですから、うーん。

class GameObject ってクラス作っといて、Enemy も Player もそっから継承する。

```
class GameObject
{
public:
        GameObject();
        ~GameObject();
};
```

とくに中身は必要なイ

```
class Collider : public Rect{
private:
        GameObject* _gameObject;
        ColliderType _colType;
        CharacterType _characterType;
public:
        Collider(GameObject* go,ColliderType coltype,CharacterType charType):
                _gameObject(go),_colType(coltype),_charType(charType){}
};
```

あのさ…何度も言うけど、このコードそのままパクって「先生の言うとおりにしたけど動き

ません」言うても知らんからな？
君は君の作り方をしてくれ。このソースコードはあくまでも例だからな？考えるヒントに過ぎないからな？責任を俺に転嫁するなよ？

はい、俺が言うと「アカハラ」とか言われたりするので、アームストロング少将に言ってもらいました。

なんかしら工夫して当たった相手の情報を得られる仕組みを作っておきましょう。

まぁともかくこれで誰に当たったか、誰から当てられたかわかるわけです。

あとちなみにな？最初っから「正解」の設計なんてプロでも得られへんねん。まず動かしてな？そっからリファクタリングしたり、いろいろしながらクラスは完成に近づいていくねん。

プログラムに関しては 100 点を狙うな!!とりあえず動く 50～70 点くらい狙って作って、そっから 100 点に近づけていけ。

ともかく、相手が「つかみ男」だったら、プレイヤーの動きを止めてください(向きは変えて構いません)

ジャンプ中も当たると移動できなくなる…まぁ、フツーにプレイヤーのジャンプ高さをつかみ男を飛び越せない高さにしとけば大丈夫です

最後…

レバガチャかボタン連打で外せる

これ…どうやって実装しましょう。ちょっと遊んだ感じではボタン押したり、レバガチャをすると敵は振り落とされる仕様のようです。

じゃあこうしましょう

捕まれて動けない状態でトリガー発動(前フレ押してなくて今押してる)するとポイントが溜まっていき、溜まりきると敵を一人ふるい落とす。そういう仕様にしましょう。

うーんメンドクサイ。

例えば「つかまれている状態」のときに上下左右どれでもいいからキーを押すとなんかしらのポイントが溜まっていく…変数作りましょか

```
unsigned int _shakeCount;//レバガチャカウント
```

こういうのをメンバ変数として作る(初期値はゼロ)。

で…

```
if (input.state(KEY_INPUT_RIGHT)){
	if (!_prevInput.state(KEY_INPUT_RIGHT)){
		++_shakeCount;
	}
}
```

こんな感じでカウントアップさせる。もちろん掴まれているときしかカウントアップしちゃ駄目だぜ？

もしフラグで「掴まれた」にしてるなら

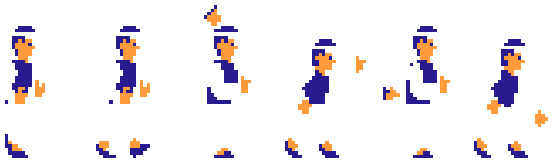
```
if (input.state(KEY_INPUT_RIGHT)){
	if (!_prevInput.state(KEY_INPUT_RIGHT)&&_isGrabbed){
		++_shakeCount;
```

```
        }
}
```
なんて書く。

で、すでにOnCollidedで「相手のこと」が分かる状態であるとすると

```
if (enemy->Type()==et_grabman){//相手がつかみ男だったら…
        _vel.x = 0;//プレイヤーを動けなくする
        if (_shakeCount>=5){//レバガチャカウントが５以上だったら
                _shakeCount = 0;//リセット
                enemy->Kill();//敵を振り払う(殺す)
        }
}
```

ナイフ男



次は「飛び道具使い」のナイフ男です。すでにシューティングゲームを作っている皆さんにとっては大したことがないとは思いますがいかがでしょうか？

動きはそれほど考えられてはいないと思われます。
①登場
②プレイヤーと一定の距離に近づく
③立ち止まる
④何フレーム(数秒？)かに１回ナイフを投げる(上か下かはランダム？)
※基本逃げない(逃げる前に倒してしまうからなのか分からないが、逃げたのを見たことがないし、それ以上は近づきもしない。

ざっと観察したところこんな動きですね。

投げるまで

ひとまずはつかみ男と同様に、登場させてから、プレイヤーに向かわせます。ですが、このとき

近づききるのではなく、プレイヤーとの距離が…200 くらい？の位置でとまってみましょう。
どうですか？
掴み男が自力で実装できてるならここは簡単でしょう。
結構つかみ男は状態から状態の切り替えが発生するので、できればメンバ関数ポインタによる状態の切り替えをお勧めします。

仮にこの関数を WalkUpdate もしくは ApproachUpdate とすると、この関数内でプレイヤーとの x 軸方向距離を測り 200 以内に入った時点で歩きを停止。

というわけで、次に停止の状態に切り替えます。フツーに考えたら 200 以内の時は VX=0 にすればそれで済むんですが、その考えだと後でちょっと苦労すると思います。

まぁとりあえずそこは自分の判断でお願いします。でも分かる努力は続けてください！？
逃げてばかりでは上達しません。

さて、では仮に停止状態を WaitUpdate という関数で表現するとします。

ここからがナイフ男らしいところになります。

投げる

いや、まぁ別に段落を分けるまでもないんですが、停止して一定時間ごとにナイフを投げます。
上段投げと下段投げがありますが、とりあえずは上段投げから実装しましょう。

で、この一定時間についてですが、実際のゲームの動画を見ても正確なところがよくわからないので、教育的観点から「ランダム」を用いましょう。

例えば、停止してから 1～2.5 秒の間のどこかで投げるとします。

君ならどう書く？1～2.5 秒という事は 60～150 フレーム間のどこかってことだね。これをランダムにするとするとどう実装しますか？

ちなみに C 言語標準のランダムは

#include<math.h>
で使える。

rand()

と呼べば勝手に 0～INT_MAX までのランダム値が返ってくる。

さて、どう実装しようか？

ちょっと自分で考えてみてくれ。
マジで「自分で」考えてみてくれ

そもそも 0～MAX では困るわけである。こういうのの範囲を限定するにはどうしたらいい？

そうだね。%を使うんだよね？

でもそれじゃ…例えば 2.5 までだったら 150 だから

rand()%150
なんてするかい？

でもこれじゃ 0～150 だから、後ろはあってるけど、前が 0 だから、ノーウェイトで攻撃してくるよ？

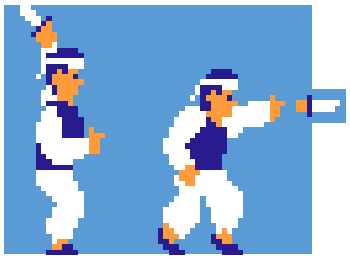
さて？

a<=r<=b

の範囲のランダムにしたい場合はどうしたらいいんだろうね？こんなの何処の教科書の書いてませんから自分で考えましょう。

ナイフ

うん、ナイフ投げ男がナイフを投げた瞬間にナイフを発生させるんだけど、



割と投げるタイミングが難しいです。相当調整しないとおかしな投げ方になります。ここは自分で調整しまくってください。
ちなみに早すぎるとまだ手に持ってる段階でナイフが出て二重ナイフになりますし、遅すぎるとなんだか間抜けになります。

ゲーム制作者は結構こういう所で苦しむのです。頑張りましょう。

でも、そこに時間をかけすぎてもいけないので、だいたいでいいです。

ナイフは「弾」つまり Bullet として扱いましょう。

シューティングゲームの時にもやった BulletFactory を使うのをお勧めしておきます。

BulletFactory は Bullet をまとめて扱い、update 時にすべての Bullet の Update をコールします。

また BulletFactory は不必要な弾の削除も行います。

ちなみに僕は不必要な弾の削除の関数の名前を CollectGarbage って名前にしています。
シューティングの時も似たような感じだったと思いますが

```
void
BulletFactory::CollectGarbage(){
	Bullets_t::iterator it = _bullets.begin();
	for (;it!=_bullets.end();){
		if ((*it)->IsAvailable()){
			++it;
		}
```

```
                else{
                        it = _bullets.erase(it);
                }
        }
}
```

ややこしいから珍しくもソースコード見せたるんやで？せやからよ～く意味を考えてな？要はこのソースコードを見てしまった以上はこのソースコードを理解しているという風に扱いますけぇな？

体力

ナイフ男はつかみ男と違って「体力」があります。一発で死んでも困るわけです。というわけで敵をライフ制にしなければなりません。

敵側の OnCollided 関数を呼び出すときに、こちらの矩形が「攻撃矩形」なのか「やられ矩形」なのかを区別する必要がありますが、ともかく「攻撃矩形」だったらライフをひとつ減らします。

そしてライフが 0 になったら「死亡状態」に移行すればいいのです。

…簡単でしょ？

1面ボス

棒男(バットマン)

たいしたこたあない。ただ、耐久力があるのと、ちょっと頭がいいだけですわー。

ご覧のように棒を持っているので、リーチがプレイヤーより長い。

ではナイフ男と同じじゃん、どこがボスなの？そういう事をクソまじめに考えるのがお仕事ですよ？

面倒なのは、その場にあたり判定が残り続けるのでジャンプで躱すことができないってことだね。
あと、ナイフ男みたいに後ろを向かないので常時攻撃の体制にあるってことかな。あとは攻撃に至るまでのインターバルがナイフ男より短い。

こいつは上攻撃と下攻撃があるけど、先ほど言ったようにジャンプがあまり効果的ではないため、だいたいしゃがみが戦略の中心になると思います。

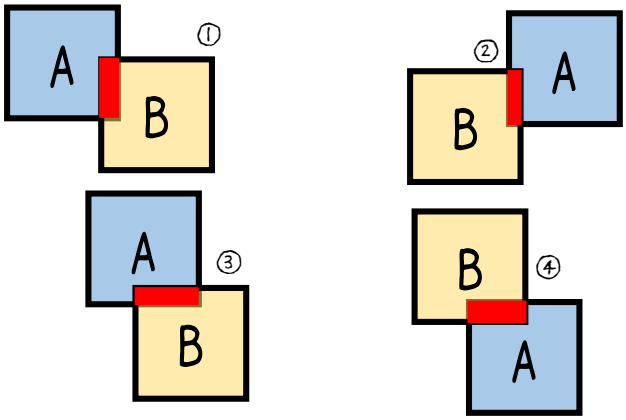
あ、あと知ってる人もいるかもしれませんが、ゼロ距離ではこいつの攻撃あたりはスカります。ですが、そんなところまで再現しなくていいです。

動き自体は
①こちらとの間合いをとる(棒がぎりぎりあたるくらい)
②一定時間ごとに攻撃する(上か下かはランダム)

うん、まぁこれくらい。
ナイフ男までを作れた人なら簡単ではないかと思います。

押しあたり

ほかのやつらと違う部分がもう一つありましたね。
ほかの奴らは、体が当たっても「掴まれる」か「すり抜ける」でしたが、この棒男の場合はすりぬけられません。掴んでも来ないので、自分自身は動けつつ、移動の制御を考えなければなりません。
押し当たりの処理は、シューティングでやった時と同様でいいです。今回はね。



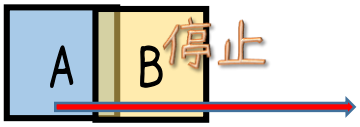
こんな図があって、小さいほうに押し返せばいいんでしたよね？まぁ今回までは①か②のパ

ターンしかないので右か左に押し返せばいい。とりあえずAかBか両方を押し返しましょう。格闘ゲームなんかではAとB両方を半分ずつ押し返します。

これによって何が便利かというと

- 押してる時
- 押してない時

でスピードが変わるようになるんですね。



どういうことかというと例えば上のような場合。「A が B によって押し返される」アルゴリズムだとAはあたかもBが「壁」であるかのように先に進めなくなる。これはこれで気持ち悪いだろう。

逆に「BがAによって押し返される」アルゴリズムの場合、あたかもBがいないかのようにAが進んでしまう。

これもまた気持ち悪い。

こういう場合は間をとってお互いを半分ずつ押し返してやるのがいい。もちろん相手が壁の場合は最初のアルゴリズムを用いるべきだが。こういう状態の時であればお互い押し返したほうがいいだろう。なお、場合によってはこのアルゴリズムを応用して

体格差があることを表現することも可能です。どういうことかというと、半分半分押し返しの場合、めり込んでいる押し返す量がdだとすると、位置関係がA<Bならば

A.x-=0.5f*d;

B.x+=0.5f*d;

とすればいいわけです。これを体格差まで考慮に入れるなら、例えば大きい奴(A)が 80Kg で小

さいi奴(B)が40Kgなら、それだけ押すときの抵抗も変わってくるわけです。
それを表現したいなら、「足して1になるような割合」を考えます。数学というより算数ですね？
どう計算しましょうか？

足して、割りやいiんですよ。足す→80+40=120。割る 80/120,40/120→(2/3,1/3)
2/3+1/3=1だから辻褄もあう。
そしてこれをお互いに押し返すつまり
A.x-=d*(1/3);
B.x+=d*(2/3);
小さいほうがより押し返されることに注意な。

わかんなかったら半分ずつ押し返せばいいよ。

その他の要素

はい、ここまで出来たらほぼほぼスパルタンXの実装は終わったようなものです。スパルタンXを5面まで作ると言っても、あとはほぼ同じような要素の繰り返しです。
敵は基本的に

- 掴む
- 殴る
- 投擲する
- 体当たりする

くらいしか攻撃手段を持ちませんので、ここまでがしっかりできてれば全面作るのもそれほどではないです。

とりあえずは、タイトルループがまだの人はそこからやってください。

画面Enter/画面Exit演出

もし、画面の入りと出で、フェードイン、フェードアウトの演出をしたければシーン内にメンバ関数ポインタを用意して

シーンEnter状態→シーン本編→シーンExit状態

という風にしてください。こうしたうえで、シーンExitは時間で次のシーンに切り替えるようにすればいいと思います。

例えばタイトルシーンならこういう風に書いています。

```
void
TitleScene::AppearUpdate(){
	int centerL = (_windowW - _titleW) / 2;
	int centerT = (_windowH - _titleH) / 2;
	DxLib::SetDrawBlendMode(DX_BLENDMODE_ALPHA,255*(60-_appearTimer)/60);
	DrawGraph(centerL, centerT-100+_appearTimer, _titleHandle, true);
	--_appearTimer;
	if(_appearTimer<=0){
		_func=&TitleScene::PressStartUpdate;
		SetDrawBlendMode(DX_BLENDMODE_NOBLEND,255);
	}
}

void
TitleScene::PressStartUpdate(){
	int centerL = (_windowW - _titleW) / 2;
	int centerT = (_windowH - _titleH) / 2;
	DrawGraph(centerL, centerT - 100, _titleHandle, true);
	if ((_blinkTimer / 40) % 2){
		DrawString(_windowW / 2 - 100, centerT - 50 + _titleH, "PRESS START
BUTTON", 0xffffffff);
	}
	_blinkTimer++;
	if (CheckHitKey(KEY_INPUT_RETURN)){
		if (!_isPressedEnter){
			_blinkTimer = 0;
			_func = &TitleScene::PostUpdate;
		}
		_isPressedEnter = true;
	}
	else{
```

```
                _isPressedEnter = false;
        }
}

void
TitleScene::PostUpdate(){
        int centerL = (_windowW - _titleW) / 2;
        int centerT = (_windowH - _titleH) / 2;

        DxLib::SetDrawBlendMode(DX_BLENDMODE_ALPHA,255*(60-_blinkTimer)/60);
        DrawRectGraph(centerL-_blinkTimer*15, centerT-100, 0,0,893,128,_titleHandle,
true,false);
        DrawRectGraph(centerL+_blinkTimer*15, centerT-100+128,
0,128,893,237,_titleHandle, true,false);

        if ((_blinkTimer / 8) % 2){
                DrawString(_windowW / 2 - 100, centerT - 50+_titleH, "PRESS START
BUTTON", 0xffffffff);
        }
        _blinkTimer++;
        if (_blinkTimer>60){
                SetDrawBlendMode(DX_BLENDMODE_NOBLEND,255);
                GameMain::Instance().ChangeScene(new PlayingScene());
                _isPressedEnter = true;
        }
}
```

ツールについて

ちょっとね、見てるとね。
アニメーション作るときに一所懸命座標を打ち込んでいる人がいたからね、これはもう「ツール作成」を教えるべき時が来たかな…と。
まぁ、これは必須ではなく作るうえでのヒントやな。ゲーム会社に入ったら必ず「ツーラー(ツール作成者)」は必要だし(そういう意味で「俺 IT 企業に行きたい」とか言ってる人もゲーム会社での仕事はあるわけや。事実、IT 企業からの転職者は最初にツール作成をさせられること多いで)
例えば、今回のような場合だと「アクション/アニメーションツール」な。

適当に配置だけやってみたけど、例えばこういうやつを作って、ツール上で切り抜いて、切り抜き情報をそれぞれのアクションにリンクさせる(紐づける)。そんなツールを作ればいちいち何ピクセル目が～とか言わなくて済むし、なんだったら中心点も保存できるようにすれば、中心のずれの修正も楽々である。
ただ、このツールも一日で作れるようなものでもなく、それなりに時間がかかります。ですが、最長でも 3～7 日以内で作るようにしましょう。ゲーム本体に比べればまだマシだと思います。
何度も言いますが、この「ツール作成」は強要はしないので、それぞれのペースで作っていってください。
煽るわけではありませんが、フロムソフトウェアとかレベルファイブや CC2 に就職した学生は自分でツールを作ってました(僕が教えたわけではなく、いや、寧ろ僕の進めた Platinum が使いにくいと言って、自分で作ってました(’•ω•’)

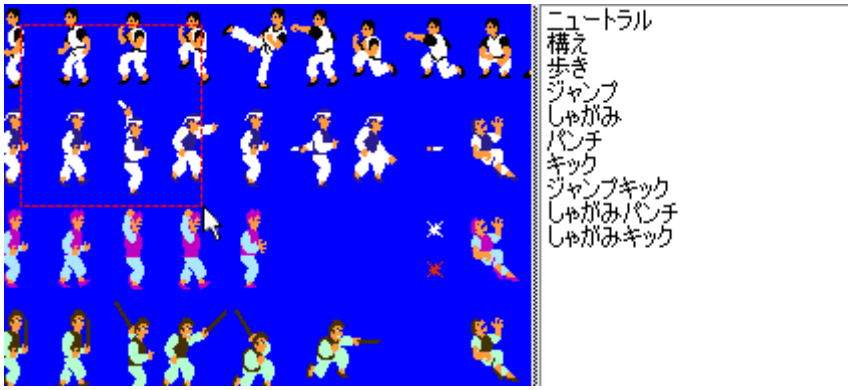
ここにきて C#言語です

ツールは「コンバータ」みたいなコンソールアプリケーションでない限り C#を使ったほうが

いいと思います。
(うえぇー、C++でも死にかけてるのにこれ以上言語覚えたくないズラ)という声が聞こえてきそうですが…
C++でツール作るのは相当しんどいですよ。だからC#を使いましょう。Unityの言語と同じだし、C++にも似ているからとっかかりやすいとは思うんですけどねえ…。

フォームアプリケーション

本当は僕もWPF使いたいんだけど、初心者にはちょっと難しいかもしれません。
https://cedil.cesa.or.jp/cedil_sessions/view/1374
WPF…興味あるんだけどなあ
でもフォームやります。



C++文法の話①

はい、久々にやってまいりました。
まだまだ文法の話は知っておくべきことは多いですよ？うぇーんもういややー(つД⊂)
と思ってる人もいるかもしれませんが、今回のこれはマジ便利だから頑張ろう

std::string

ちょうど今頃皆さんは、Javaの授業にてstring(文字列型)をお勉強したと思います。
string 超便利!!!
C++にはstringないのかなぁ…('•ω•`)ｹｲﾖﾈ…

あります。あるんですよこれが。

その名はstd::stringです。
使うには
#include<string>

として使います。

こいつが便利なのは文字列をつなげたりするのがC言語のchar*なんかより超便利ってことです。

例えば

std::string str;

などという風に宣言します。(std::stringまでが「型名」です。std::vectorとかと同じです)
中に値を入れる場合は
str="中身";

のようにすればstrの中身は"中身"という文字列になります。

文字列の長さが大きくなっても大丈夫。自動で長さを計算してくれてメモリを確保してくれます(vectorと一緒やね…まぁそれによるデメリットもあるんやけど、今は考えなくてもよかろう)

さらにこのstring型、文字列の連結も自動でやってくれます。つまり

str+="はカラだよ";

とすればstrの中身は"中身はカラだよ"という文字列になります。

便利でしょ?

あれ?でも例えばC言語の関数にこの文字列を入れようとするとどうなるの?例えば

DrawString(0,0,str,0xffffffff);

ってのは可能なの?

残念ながらそこまでは親切じゃないんだなあ…。Cの関数に使うにはやっぱりC言語用の文字列表現にしなければいけない。

そこで便利な関数。
ここは重要ですよ？だから目を開いて耳をすましてください。

その関数の名は、c_str()関数です。
c_str()
c アンダーバーstr()
まぁ、そういう事です。string 的文字列表現を C の表現に変えますので char*ポインタ型に変換するものです。つまり

```
DrawString(0,0,str.c_str(),0xffffffff);
```

とやれば通ります。

さらにこの string の偉いところは、文字列の比較も簡単だという事です。

C 言語の場合、文字列の比較には==は使えませんでした。

```
char name[]="abcde";
if(name=="abcde"){}
```

などという比較は不可能でしたね。

ところが string はそれが可能なのです。

例えば

```
std::string name="abcde";
if(name=="abcde"){}
```

は予想通りの結果となります。マジ便利でしょう？

ついでに便利な関数も紹介しておきます。
vector とほぼ同じ構造なので vector の関数はフツーに使えます。

つまり、size()は文字列の長さを表し、empty()で文字列が入っているかどうか判別できます。

それ以外の便利な関数は…

c_str()　C の文字列表現を返す
data　　C の文字配列表現を返す
copy　　文字列のコピー
find　　文字列の中の指定文字列を検索して、その場所(インデックス)を返す
substr　文字列の一部を取り出す

など、まぁ今のところはそれほどありがたくないかもしれないですが、知っておくのと知ってないのとでは結構プログラミングの戦略が変わってきます。

こいつと stream ってのを組み合わせると非常に強い効果を発揮しますが、アレは若干難しいので、またの機会にしましょう。もう少しスキルが上がってからね。

ストリームについてちょっとだけ

STL って組み合わせられるのよ？

思い込みというのは恐ろしいもので、map や vector というのは一度に一種類しか使えないと思っていたりします。

大丈夫、組み合わせて使えます。
ですから、

```
struct Nanika{省略};

std::map<int,vector<Nanika>> chaos;
```

などという使い方も可能です。
なお、
chaos[14].push_back({～初期化～});

という風に値を入れることもできます。

テンプレートについて

テンプレート、これはいいものだ。ただしダークサイドに陥る危険性もはらんでいる。チョット使うくらいならいい。
だが、
http://ja.wikibooks.org/wiki/More_C%2B%2B_Idioms
に載っているテクニックや
「ModernC++Design」って本のテクニックを読むと間違いなくダークサイド行きである。もはや頭がおかしいレベル。
http://f3.tiera.ru/other/DVD-009/Alexandrescu_A._Modern_C++_Design(c)_Generic_Programming_and_Design_Patterns_Applied_(2001)(en)(271s).pdf

boostの中身を見ても、頭がオカシイことが分かるだろう。
テンプレートは便利すぎるために、使用法を誤るとKittyGuyになっちゃう。くれぐれも注意した上でこれからの話は聞いて欲しい。
テンプレートというのは型を特定しないで様々な関数やクラスを定義できるというスグレモノなのです。たとえば、2つの数のうち大きい方を返すような機能をほしいとする。このような機能は整数型だろうが浮動小数点型だろうが使ってみたい。
こういう時に使えるのだ。後で説明する関数テンプレートを使用すると
Max(1,7)は整数型の7を返し、Max(1.6f,9.0f)は浮動小数点型の9.0を返すことになる。

最後まで聞けば感じる人もいるけど、テンプレートは型を特定しないとかそんなチャチなものじゃ断じてねぇ。もっと恐ろしい物の片鱗を味わうだろう。

関数テンプレート

先ほども言ったように関数テンプレートは型を特定せずにMax関数やらAdd関数やらを作れるものである。

で、ちなみに関数テンプレートの使い方は

template<class 仮の型名> 戻り値 関数名(パラメータ){

```
	ほにゃらら、ほにゃらら
}
template<typename 仮の型名> 戻り値 関数名(パラメータ){
	ほにゃらら、ほにゃらら
}
```

である。キーワードは template と class もしくは typename である。で、class と typename にはとりあえず違いはないと思っておいていいです。ですから好きな方を使いましょう。
たとえば先ほど例に出した Max 関数を作るのならば

```
template<typename T> T Max(T a, T b){
	return a>b?a:b;
}
```

などという定義をすることができる。
で、仮の型名をここでは T と置いてはいるが、何でもいい。で、数字や変数が入った時点で自動的に int 型の Max 関数や float 型の Max 関数が生成される。
つまり

```
cout << Max(1,9) << endl;
```

などと書いた瞬間に

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

が生成され、

```
cout << Max(1.0f,7.2f) << endl;
```

と書いた瞬間に

```
float Max(float a, float b){
	return a>b?a:b;
}
```

というコードが見えない所で生成されるわけです。ここまで読んだ人はもしかしたら「え？テンプレートって指定できる型名は1つだけなの？」と思うかもしれませんので複数設定してみましょう。

```
template<typename Ta,typename Tb> Tb Scaling(Ta value,Tb scale){
	return (Ta)value*scale;
}
```

なんて書いて

```
cout << Scaling(2,5) <<endl;
```

とでも書けば
```
int Scaling(int value,int scale){
	return (int)value*scale;
}
```
が生成されますし、
```
cout << Scaling(5,5.9f) <<endl;
```
とでも書けば
```
float Scaling(int value,float scale){
	return (float)value*scale;
}
```
というのが見えない部分で生成されます。ここに挙げた例だとあまりアリガタミがありませんが、ぼちぼち利用できそうな場面を見つけて活用…別にしなくてもいいですが、知識としては覚えておきましょう。C#でも「ジェネレータ」という名前で使用されますので、こういう「型がコンパイル時に決定される」系の話は頭の片隅に入れておきましょう。

クラステンプレート

関数テンプレートはまだわかりやすかったかもしれませんが、次は少しだけマニアックになってきます。なってくるのですが、実際はこいつが本番なんですねー。C++さんはホンマ初心者殺しやでえ…。
クラステンプレートってのは、要はクラスのメンバの型をコンパイル時に決定するというものです。
定義は
```
	template<typename  仮の型名> class  クラス名{
		仮の型名 メンバ変数;
		仮の型名 メンバ関数();
		int a;
		void func();
	};
```
と言った具合に定義します。ポイントは赤字で書いてある部分です。関数テンプレートの時と同様に typename または class というキーワードで仮の型名を定義するところまでは同じで、クラステンプレートの場合、メンバに対して仮の型名を使えるというところが強力なのです。

例えば、前回使ったベクトルクラスは int 型向けにつくりました。これを Float 型向けに作るとして、わざわざ
Vector2f なんてクラスを作りますか？作ってもいいですが、このクラステンプレートを使用

すると簡単で。

```
template <typename T>class Vector2{
        public:
                T x;
                T y;
                Vector2():x(0),y(0){}
                Vector2(T inx,T iny):x(inx),y(iny){}
                void operator+=(const Vector2<T>& v){
                        x+=v.x;
                        y+=v.y;
                }
                (略)
```

などという感じに書き換えられます。これで Vector2<T>というのは int 型だけではなく、float 型など様々な型に対して対応することができるようになります。
ということで本日の課題その1として、テンプレートによるベクトルクラスを完成させて下さい。つまり

配列オブジェクトを作ってみよう

C++において、実行時に生成される配列をつくろうと思うと
char*c=newchar(255);
などと書かねばならず、こいつにさよならするには
delete()c;
などと書かねばならない。これは malloc～free でも同じですね。
どうせ関数の中だけで使用するような場合に動的配列は煩雑だし、解放忘れもある。どうにかして、通常の配列変数のように自動的に消えてくれないものだろうか…

なければ作るのがプログラマのお仕事です。
ということで、配列として作れば自動で消えるようなオブジェクトを作ってみましょう。もちろん、いろんな型に対応できるようにテンプレートも使用します。
ここで使用するテクニックは

- 配列の確保(new)、解放(delete())
- 引数つきコンストラクタ
- デストラクタ
- クラステンプレート
- オペレータオーバーロード(()演算子)

です。今までやってきたことの集大成ですね。今回はこれを Array というクラスで作ってみましょう。
要求としては

- 配列のように使える
- スコープを抜けたら解放される
- 配列のサイズが分かる関数 Size()を作る
- 型を問わない

です。

```
template<typename T>class Array{
	private:
		T* _array;
		unsigned int _size;
	public:
		Array(unsigned int size):_size(size){
			_array=new T[size];//配列の確保
		}
		~Array(){
			delete[] _array;//配列の解放
		}
		（これ以降は自分で考えて書いて下さい）
};
```

という定義があれば

```
Array<int> ints(100);
for(int i=0;i<100;++i){
	ints[i]=i+1;
}
```

というように使われ、スコープを抜けた後は、きっちり全て開放されるようにしてください。
これは int や float だけでなく Vector2 でも使えるものです。

```
Array<Vector2<int> > vectors(100);
for(int i=0;i<99;++i){
	vectors[i].x=i+1;
	vectors[i].y=i+1;
}
```

という風に使えるわけです。
コード的なヒントはここまでです。さぁ、頑張って作りましょう。

オペレータは()演算子も使えますので、これを使って配列っぽく使えるようにして下さい。
戻り値& operator[](unsigned int idx)
などと定義すれば通常の配列の如く使えます。なお、戻り値に&が付いているのは、以前に勉強した「参照」を表しており、配列のように配列の中身を書き換えたいがためです。

とりあえずは、クライアント側で以下のように使えるようになってればオッケーです。

```
Array<int> ints(100);
for(int i=0;i<100;++i){
	ints[i]=i+1;
}
for(int i=0;i<100;++i){
	cout << ints[i] << endl;
}
cout << "size=" << ints.Size() << endl;

Array<Vector2<int> > vectors(100);
for(int i=0;i<100;++i){
	vectors[i].x=i+1;
	vectors[i].y=i+2;
}

for(int i=0;i<100;++i){
	cout << vectors[i].ToString() << endl;
}

cout << "size=" << vectors.Size() << endl;
```

このように使えるようになっていれば要求は満たします。これが今日の課題2だ。
頑張れ～。

スマートポインタ(初歩)

スマートポインタというのはその名の通り「スマートな(カシコイ)」ポインタの総称である。
どういうことか。
例えばポインタの厄介な所はたとえばクラス Enemy があったとして
Enemy* e=new Enemy();
とやると、当然 sizeof(Enemy)ぶんの領域が確保されてしまい、明示的に delete しない限り解放されることはありません。

これは結構マズいことなんですよね。特に複数人でプログラミングしている場合はどうしても解放を忘れる人が出てくるし、そもそも解放のタイミングをいつにするのか、誰(どの所有者(クラス))が解放の責任を持つのか、予め決めておくか、自動で消えるようにして欲しいところです。
C#やってる人ならわかると思いますが、GC(ガベージコレクタ)があるだけで、随分とプログラムがラクになっているでしょう。
C++が嫌われ C#が好かれるのはこういう所なんですよね？じゃあ何故未だに C++がゲームや組み込み、果ては Android にまで使用されているのかというと、ガベコレの解放タイミングがクライアント側で調整できない(ある程度の調整はできるが細かいのは無理)ため、意図しないタイミングで大量解放が行われてしまう。
コンピュータのメモリ解放はそれなりに時間的コストを食うのである。これをきっちり自分で把握したい要望が未だにゲーム業界や組み込みでは多いため C++が仕方なく使われているのである。まぁ、純粋に C++好きな奴も多い業界なんで…。

それはさておき、どちらにせよアホが居るチームで明示的な delete を期待するのは危険なことである。これをもうちょっとだけ安全に使う目的で作られるのがスマートポインタである。
スマートポインタは boost のが有名である。サーバーに boost からスマポライブラリ部分のソースコードだけ取ってきて置いていますので、興味がある人は見ておいて下さい。

試しに scoped_ptr を作ってみよう

後で紹介するスマートポインタを正しく理解するためには、とりあえず「いちばん簡単なやつ」くらい自分で実装しといたほうがいいです。
そこでスコープドポインタ…を、余裕のある人は実装してみましょう。それなりに便利っちゃ便利です。

スコープドポインタとは俺の造語…なのか？いやそんなことは無いと思うんだが……ほうぼうで「それ君の造語やん」って言われるのだ。
いやいや、だって boost の中にも"scoped_ptr.hpp"ってあるやん。まぁそれはさておき
配列オブジェクトまで書けていたらなんちゃないです。こいつは何かと言うと、通常はポインタで扱うべきものを通常の変数のように自動で解放されるものであるかのように扱えるようにするものです。
何を言っているのか分からねえと思うが俺にも何を言っているのか分かんねえじゃ仕方ないので、ぼちぼちと説明していきます。例えば

```
void function(){
        int a;//これは function()を抜けた時点でスコープから抜け、解放される
```

```
}
```
は問題なく変数 a が解放されるのだが、
```
void function(){
	int* a=new int();
}
```
この場合、int 型の 4 バイトがアプリ終了まで生き続ける…だれからも使われずにな!!更に言うと、これがメインループの中とかで呼び出されているのであれば非常にマズいことになる。
おお…ヤバイヤバイ。
「そんな馬鹿な事やんねーよ」と思ってるあなた…甘いですよ。5 人以上で開発してると、どこかにそういうコードが紛れ込むものだし、更に言うと 1 ヶ月まえの自分のコードは他人のコードである。
まぁ、とにかく少なくとも、関数内で確保したメモリは関数を抜ける時に解放したいもしくは所有者(所有クラス)が破棄される時に一緒に破棄されるようにしたい。
そんな場合に使用するのがスコープドポインタです。

ここで作るべきものは

- コピー、代入の禁止
- ->演算子オペレータオーバーロード
- *演算子オペレータオーバーロード
- デフォルト引数
- Reset()関数
- デストラクタで delete
- ナマポを返す Get()関数

といった感じです。今回は boost のものよりは機能を少なくしておきます。
このスマポはスコープから外れると消滅する(関数内で定義されれば関数を抜けると共に消滅。クラスのメンバとして定義されていればクラスオブジェクト消滅とともに消滅する)哀れなスマートポインタです。でも結構使うと思います。

class ScopedPtr で宣言しておきましょうか

で、このスマートポインタは色々な型に対して使えるようにしなければ意味が無いのでテンプレートを使用します。

```
template <typename T> class ScopedPtr{
		T* _ptr;//本当のポインタを内包しておく
		:
```

}
実際に確保したメモリのアドレスを指し示すポインタをこのテンプレートクラスの中に内包しておきます。こいつが裏の主役となります。こいつに自動変数(いい子ちゃん)の仮面をかぶせて、善行をさせようと思います。

で、このスコープドポインタ、コピーや代入を行われてしまってはスコープドポインタが崩壊してしまうため、予めコピー、代入を禁止しておきます。

コピー、代入を禁止するにはコピーコンストラクタ及び=演算子を private にするのでしたね？
とりあえずコピーコンストラクタはこうでしたね？
ScopedPtr(const ScopedPtr&);
代入の方は前にやったので、自分でやっておきましょう。

次にコンストラクタにて真のポインタを代入→保持できるようにします。ですから
ScopedPtr(T* ptr=0):_ptr(ptr){}

とでも書いてやってコンストラクト時に真のポインタを渡せるようにしましょう。なお
ScopedPtr(T* ptr=0):_ptr(ptr){}
この部分ですが、C++ではデフォルト引数というものが使えるようになっており、この引数が入力されてない場合は=の右辺値が採用されることになっています。つまりこの場合は引数を入力しなければヌルポインタ(0)が設定されるようになっています。
そう、コンストラクト時に真のポインタが渡せない事も有り得ます。なんかしらの Create 関数を他所から呼び出して作らなければならない時なんかがそうでしょうかね。
ここで、遅延して真ポインタをセットできる関数 Reset 関数を作ります。
Reset 関数の仕様は
何もポインタが渡されなくて、かつ内包するポインタがヌルなら何もしない。
何もポインタが渡されなくて、内包するポインタが生きてるなら元のポインタを破棄して下さい
なんかしらのポインタが渡されて、かつ内包するポインタがヌルなら新しいポインタを内包するポインタに代入して下さい。
なんかしらのポインタが渡されて、かつ内包するポインタがヌルでないなら、元のポインタを破棄した上で、新しいポインタを内包するポインタに代入して下さい。
この仕様に合うような Reset 関数を作って下さい。

さて、ポインタのようにこいつが挙動するのであれば->演算子でポインタ自身の関数をコー

ルできなければなりません。
どうすればいいのか？簡単です。
->オペレータに元のポインタを返させればいい。それだけです。簡単すぎるでしょ？
では->オペレータオーバーロードを作って下さい。

次に、ポインタであれば*演算子で、ポインタのもつ「値」を返さねばなりません。これも簡単です。
自分の持っているポインタの値を返しさえすればいい。
というわけで*オペレータオーバーロードを作って下さい。

さて、ここまで書けたらスコープドポインタは出来上がっているはずですので

```
class Test{
	private:
		const char* name;
		int value;
	public:
		Test(const char* inname) : name(inname){
			cout << inname << "が生成されました。" <<endl;
			value=0;
		}
		~Test(){
			cout << name << "が破棄されました。" << endl;
		}
		int Value() const{
			return value;
		}
		void Value(int v){
			value=v;
		}
};
```

こういうクラスを用意してやって、自分が作ったスコープドポインタを使用したサンプルコード

```
int main(){
	ScopedPtr<Test> a(new Test("a"));
```

```
        a->Value(6);
        cout << a->Value() << endl;
        cout << a.Get() << endl;
        a.Reset(new Test("b"));
        a->Value(2);
        cout << a->Value() << endl;
        cout << a.Get() << endl;
        return 0;
}
```

こんなのをつくってやって、

```
a が生成されました。
6
0x5b2e50
b が生成されました。
a が破棄されました。
2
0x5b3e78
b が破棄されました。
```

こういう出力が行われることを確認して下さい。

スマートポインタについて

C++はメモリまわりが難しい、難しいと言われて嫌われてますが、C++の人たちだって黙って放置しているわけではないのよ？
というわけで、それに対する便利な機能としてスマートポインタってのがあります。
シェアドポインタ(参照カウンタポインタ)とウィークポインタ(弱参照ポインタ)ってのがあります。
それについて説明しましょう。
使い方はどちらも
#include<memory>
とすれば使えるようになります

std::shared_ptr

shared_ptr は内部に参照カウンタを保持しており、誰かが参照すれば、このカウンタが上がっていき、参照してるやつがいなくなったら(スコープから外れる等したら)カウンタが減っていきます。
このカウンタがゼロになった時点で誰からも見られてないという事になるので、その時点でポインタを削除(delete)します。

今、敵を作って、ポインタをヴェクタに入れたりして、削除するときに苦労していると思いますが、そういう苦労からも解放され、メモリを削除する手間もなくなります。

余裕のある人は試しに今のエネミーを shared_ptr 化してみましょう。解放とかの処理がだいぶ楽になると思います。

ただ、これはテンプレート系全般に言えることなんですけど、やらかしたとき(コンパイルエラー時)に結構ややこしいメッセージが出てくるので、それにも慣れるまでは結構大変かもしれません。

std::weak_ptr

ウィークポインタってのは弱参照って意味です。具体的には「見てるけど、所有権は持ってないよ」という意味のスマートポインタです。

そんなものの何が役に立つのかというと、ダングリングポインタ(どっかを指し示してはいるけど、実はその中身が解放されてるポインタ)の防止に役立つわけです。

どういうことかというと、指示してたアドレスが死んだ(shared_ptr が解放された)タイミングで、weak_ptr の指示しているポインタが expired になります。
具体的には expired()関数が true を返したら、元のポインタが死んだってことです。

少なくとも死んだかどうかが分かるため、それを利用して危険を回避することができます。

スクロールしよっか

さてそろそろスクロールを実装しましょうか…。

とりあえず「スクローラー」もしくは「カメラ」というクラスを作って制御するようにしましょう。今回は「僕は」Camera というクラスで制御します。
あくまでも「僕は」なので、君たちは別のやり方をしていいですよ。

カメラの役割はとにかく「プレイヤーが何処にいたとしても、画面の真ん中に表示させる」わけで、そのやり方は問わないです。

さて…どうしましょうかね？

とりあえずプレイヤーの座標(への参照)を持っておいてそれを画面の真ん中にずらす値を返すようにしましょう。

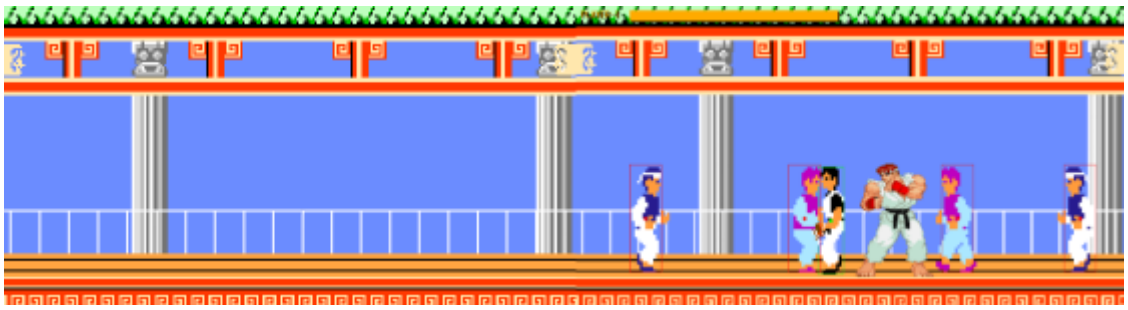
というわけでカメラはセットアップ時に画面の幅と高さを持っておくようにします。

あ、あとカメラはゲームシーンの持ち物ってことでよいでしょう。

で、プレイヤーへの参照持たせとく…セットアップ時に取得した幅を sw とすると

プレイヤーの座標を p とすると、px が画面の真ん中に来るようにしなければならない…。

このためには予め画面の大きさと「ステージの大きさ」を知っておく必要があります。とりあえず 2 画面分作ってみてスクロールする実験からやってみましょう。



ひとまずスクロール止めとかはない方針で。超えたら背景真っ黒でいいから。

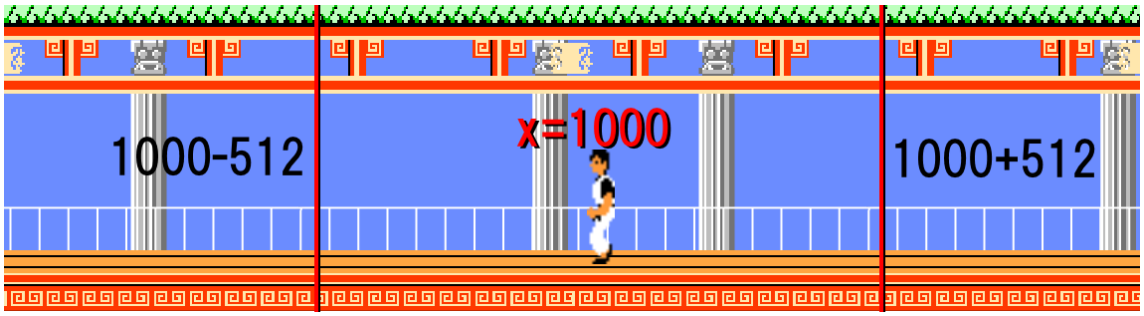
じゃあ2画面分だったら例えば一画面 1024 だったら 2048 なのでステージ幅 2048 だとします。

そうすると
左端が 0 右端が 2048 ですわな？そして画面が 0～1024 やね？で、プレイヤーを画面の 512 あたりに表示すればいいと考えます。
適当なプレイヤー座標(カメラ座標ではなくステージ座標)を px とする。この x は 0 からの座標だと考えてください。ちょっと面倒なんですけど、スパルタン X は右端からスタートします。ここだけ気を付けてください。
ですから、最初の座標は画面の真ん中ですけど、ステージ全体からいうと 1536 ピクセルあたりが初期座標になるわけです。

さて、その前提で px を真ん中にするにはどうしたらいいんでしょうか？

本来画面の左端は 0 であるべきです。そこからどれくらいずらすのか
英語ではこれを Offset といいますので、それ考えましょう。

やみくもに考えて分からない場合の考えるヒントとして、いくつかの
「実例」で試してみるってのがあります。数学でもそうですけど、一般解が分かりにくければ特殊解について考えるという…そういう思考のテクニックがあります。
どういうことかというと、x とか y とか a とか b で分かりにくければ実際に具体的な数として 5 とか 10 とかいくつか入れてみて、どういう結果になるのかという所から考えるわけです。
というわけで自機が例えば px=1000 の位置にいると仮定しましょう。



そうすると、ね？シンプルでしょう？

当然ながらカメラ中心位置はプレイヤーと同じ座標。カメラ左端はその中心から画面幅/2 ぶん左に、右端は中心から画面幅/2 ぶん右にあるわけです。

画面にはこの範囲に入っているものだけ表示すればいい。

そしてこの絵をよく見ると…根本的なところに気づきませんか？

要は「ずらす量」ってのはプレイヤーの X 座標をまるまる引いてやって、画面幅ぶん足しているだけです。

でしょ？
自分でも図を描いて考えてみてね。これ、授業のたびに言ってるのに、描きもしないで「わかりませ～ん」てお前な…って思うことがありますわー。
小学校の頃は素直に九九を唱えたから今でも九九が言えて、ちょっとくらい役に立っているんでしょうが…。とりあえず試せ。文句はそれからだ。

まぁそんなこんなでカメラクラスを書くとヘッダだけやけどこんな感じにかけました。

```
#include"Geometry.h"
class Player;
//カメラクラス
class Camera
{
private:
	Player& _player;//プレイヤーの座標は知っとかんとな
	int _stageWidth;
	Rect _rc;//画面枠
public:
	Camera(Player& player);
	~Camera();
	void Setup();//ここで画面の幅とか取得するやで
	Rect& GetRect(){ return _rc;}//今は使わないけど、まぁなんかに使うかも
	float OffsetX();//これが目的の品やで
};
```

どやぁ…？まぁヘッダだけしか見せへんけど……どうにかなるでしょ？

ちなみに画面の大きさを取得するには

GetScreenState を使用します。
http://dxlib.o.oo7.jp/function/dxfunc_graph3.html#R4N2

さらに一定時間ごとにこの数値を表示することで、よりスパルタン X に近づけていきましょう。

一二三四五六七八九十

おまけ

ヒットストップ

ヒットストップというのは、アクションゲームや格闘ゲームにてよく使われる手法で、攻撃判定が入った瞬間に画面が(キャラクターが)一定時間ストップするというものです。

まぁ例えば人殴ったとするじゃん？
殴ったと同時に吹っ飛んじゃったら、パンチの重みが表現できないのよね。
むしろパンチのダメージは殺されてるわけよ。

という事で、吹っ飛ぶ前に「ため」の時間を作る。

だいたいスト3rdとかだと8フレームくらい止めているらしい。

アーケード版イーアルカンフーとかだと相当に大げさである
https://www.youtube.com/watch?v=yxTzc0YfVT0
0.5秒(30フレーム)くらいは停止してないかこれ

ということで、攻撃もしくは食らった時の姿のまま停止する。

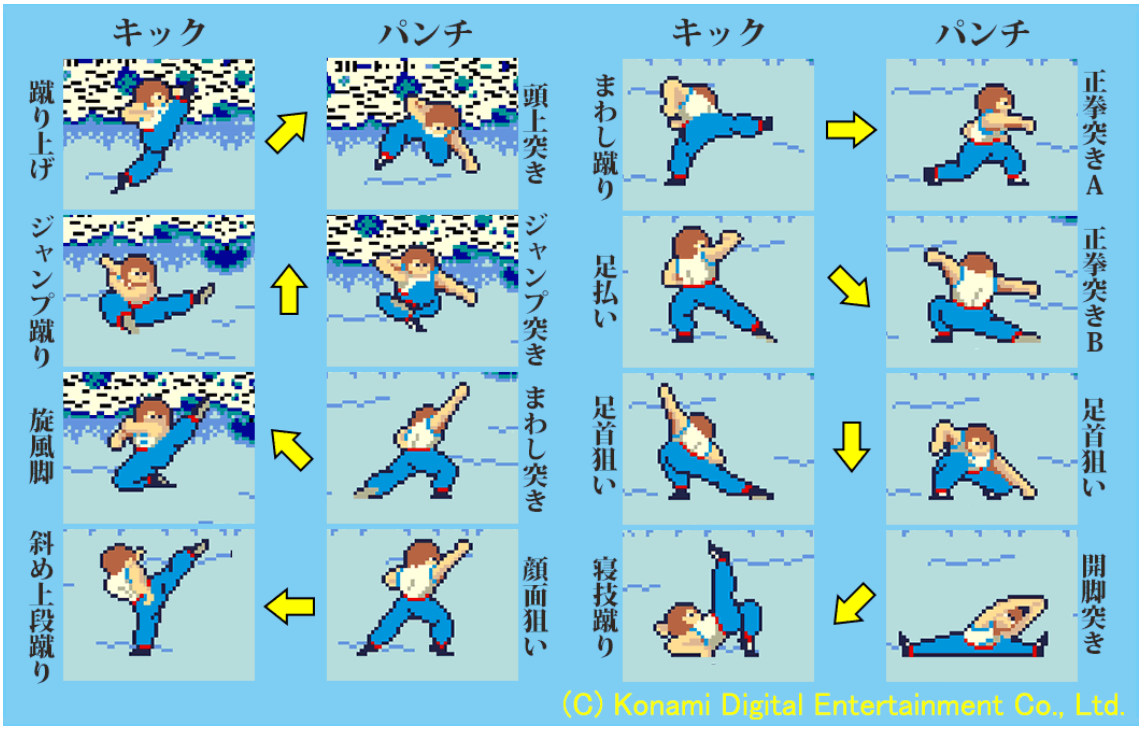
というのを、余裕があれば実装してみてください。

コマンド

以前にちょっとコマンド実装の話をしましたが、ストリートファイターなどは入力履歴をさかのぼる必要があるので結構難しいです。

そこでまだ初心者でも作れそうなコマンドを紹介します。
初心者でも作れるかもしれないコマンド

イーアルカンフーのコマンド表です



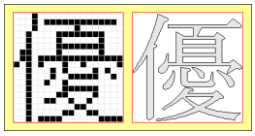
同時に押したレバーで、技が変わります。
これだったらなんとか実装できそうでしょ？
要は、攻撃ボタンを押した時点のレバーが何処が入っているかだけを見ているわけです。

文字関連

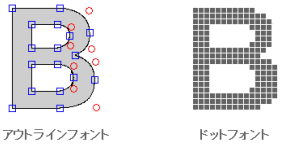
ゲームにおいて結構文字ってのは重要であります。何故ならユーザー側に対して説明をするための一番手っ取り早い方法だからです。ただ、フォントをラスタライズするってのはそれなりに処理を食うので(今回のような Windows 向けプログラムなら問題ないが、Android 向けのを作ろうとするならちょっと考えたほうがいい)。

フォントのデータ構造と文字ラスタライズのしくみ

とりあえず知ってるとは思いますが「基礎知識」としての、フォントの話をしようと思います。
フォントってどういう情報だと思う？知ってる人、どれくらいいるかなあ？
うん、もう面倒だから答え言っちゃうと「ベジェ曲線」やねん。
これ結構重要な情報やねんで？
あ…すまん、全部が全部「ベジェ曲線」で出来とるわけちゃうで。
大きく分けると「ビットマップフォント」と「アウトラインフォント」ってのがあるんや。
ビットマップってのは君らのご想像通り、「画像情報」やで。つまりピクセルごとに「点を打つか打たないか」情報が入っとるわけや。
そこは理解できるね？
でも、それだと問題が起きるんだ。例えば今みんなが見ているドキュメントなんかそうだよね？
例えば画面のサイズがでかくなった時に「ガタガタ」のフォントが見えてしまう。



それでは困るんですよね。
というわけでどういうデータになっているのかというと「ベジェ曲線」のコントロールポイントデータになっている。



上の図の○とか□が「コントロールポイント」である。まぁベジェは細かいこと言うと長くなっちゃうんだけど、曲線を作るための仕組みや思っておいて？
フォトショップの【パス】使ってる人ならわかると思いますけど、あれです。GIMP にもあるので、試すとわかりやすいです。

ともかく、いつもみんなが使っているフォントはそういうベジェ曲線のコントロールポイン

トってデータでできると思っててね。
それを元に曲線を作って表示すれば、フォントになってるわけです。
で、曲線をピクセルデータに変換するのを「ラスタライズ」って言います。ラスタライズってのは数学的なデータをピクセルデータに変換することを言います。

この「ラスタライズ」処理が Android は恐ろしく重い…予想以上である。もちろん、Android 端末でもブラウザの拡大縮小ができて、それなりに速い以上は高速化する方法はあるのだが、何の工夫もしないと恐ろしく重いのだ。

とはいえ、Windows でやる場合はそれほど気にすることはないので「豆知識」くらいに思っておいてください。

一文字ずつ表示

例えば文字を左から流れるように描画していきたい。そういう要望もあると思います。
ここでは string を使用して「STAGE　CLEAR」と表示する方法を提示したいと思います。

前にも話しましたけど、string には substr という「文字列の一部のみ」を抜き出す関数があります。これを使用します。

string 型の変数にあらかじめ表示したい文字列リテラルを代入します。

```
std::string clearString = "STAGE　CLEAR";
```

これはオッケーよな？
例えばこの中から最初の１文字のみを抜き出したければ

```
clearString.substr(0,1)
```

で抜き出せる。

これを DrawString すればいい。勿論、C 言語文字列に変換する必要があるので c_str()を忘れないように。

```
DrawString(0,0, clearString.substr(0,1).c_str(),0xffffffff);
```

これで一文字目が表示される。

ちなみに substr は

substr(開始位置インデックス,文字列長)
で指定した文字を抜き出せる。

これは分かるね？ということは 10 フレごとに 1 文字表示させたいのならば
```
DrawString(x,y,clearString.substr(0,_frame/10),0xffffffff);
_frame++;
```
のようにします。

文字に「影」をつける

えーと、これ、期待するとガッカリすると思いますが、コスト安の割には効果的なので教えときます。
さっきの
```
DrawString(x,y,clearString.substr(0,_frame/10),0xffffffff);
```
の前に一行追加するだけ。
右下にずらした状態で、色の黒いのを描画するだけですたい。
```
DrawString(x+2,y+2,clearString.substr(0,_frame/10),0xff000000);
```

このずらし具合で影の距離感が変わります。そこは何度も自分で見て調整してください。

カメラまわり

ちょっとスパルタン X のカメラ処理に色々と細工をしてあげることで、より迫力のある演出をしてみましょう。

画面揺れ(地震)

ここまで読まれている皆さんは既にカメラを実装していることだろうと思います。
それでしたら「画面揺れ」を実装するのはそう難しいことではありません。
画面揺れは実装のコスト(むずかしさ)の割に、かなりの演出的効果をもたらすものです。

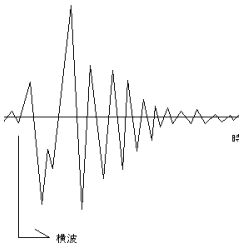
ではやってみましょう。

void Quake();
って関数でも作りましょうか。

パラメータとして

- 振幅(揺れる最大ピクセル数)

を与えます。



実際の「地震」の波の図がコレなんですけど、見て分かるように、最初にズガン!!と来てスゥーッと振幅が小さくなっていき、消える。そんな感じです。

そうなるとCameraにUpdate関数を用意してその中で最大振幅に0.9をかけていくとかそんな風にすればいいんじゃないかな。

まぁ、何も考えずに作るとずーっと振動するので工夫は必要ですね

例えば、こう

```
if(_quakeFrame>0){
        _quakeOffset.x=-_quakeOffset.x*0.9f;
        --_quakeFrame;
}else{
        _quakeOffset=Vector2(0,0);
}
```

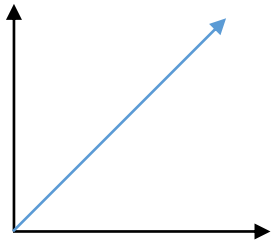
とでもしておきます。_quakeFrame が振動終了条件ですね。
ですから、パンチとかが入った瞬間に

```
void
Camera:: Quake(float power){
        _quakeFrame=60;
        _quakeOffset=Vector2(power,0);
}
```

とでもしておくわけやな。
これが画面揺れの実装や。

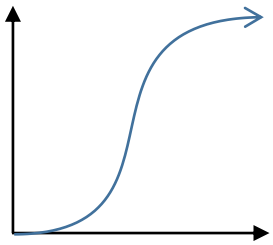
より人間臭いカメラの動き

例えばプレイヤーが動いたとして、即プレイヤーを追尾できるのはコンピュータくさい。



ロボロボしい動き

人間のキャメラマンやったら、こうは動けんのや。どうしても最初にラグが発生する。
というわけで、人間がキャメラマンならこういう動きをする。スパルタンＸとかならメカメカしくてもいいんだが、



人間ぽい動き

最近のゲームとかだとこういう動きをしたりする。

さて、こういう曲線には名前がついていて
ヒステリシス曲線、ロジスティクス曲線、シグモイド曲線とかいう種類がある。

この中でかろうじて実装が簡単なのがシグモイド曲線と言って、

$$t = \frac{1}{2}(1 + \tanh(\frac{1}{2}ax))$$

こんな数式で表されるものです。
まぁ、利用するのはそれほど簡単ではないので、ここで実装するのはお勧めしません。

「そういうの」があって「実際のゲーム」で利用されている。

ってことを知っておけば十分です。

さあ魔改造だ

ここからは改造して別物にしていきましょう。

足場を作ろう

え？足場あるじゃん？とか思ってるかもしれませんが、ところがどっこい現在は決め打ちの足場ですから拡張性がありません。

おそらく今は足場を固定で作っておられると思います。

スパルタンXのルールであれば全く問題はないですが、これから足場を改造して様々なことをやっていきたいと思います。

そのための準備として、ものごっついながーい足場を作って、そことのあたり判定と押し返しを考えればいいです。

この場合、必ず上に押し返せばいいのだから簡単でしょ？

とりあえず床のあたり判定を置いて、地面に落ちないようにできたら、床のあたり判定をステージ全体ではなく、少し小さくしたりして「穴」を用意しよう。

床から離れたら落ちるようにしてみてください。

なお、「穴」を表現するためにいちいち背景を切り取ると面倒なのでDrawBoxでその部分の床を真っ黒に塗りつぶしましょう。

もしくはトゲトゲ画像を置いても構いません。落ちたらティウンティウンするように(死亡処理)してください。

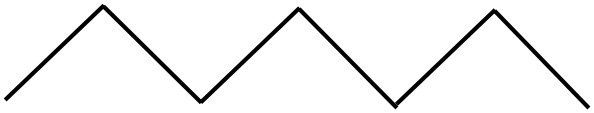
動く足場

これ結構みんな出来ないんだよねぇ～。

いや、表示と当たり判定がうまくいってるのならそこまで難しくないよ？
動かすだけならね？

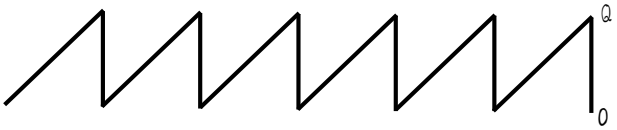
等速で行ったり来たり動かすときのヒント…

いや、これ前にもやったと思いますが、改めて話します。
行ったり来たりってことは動きが

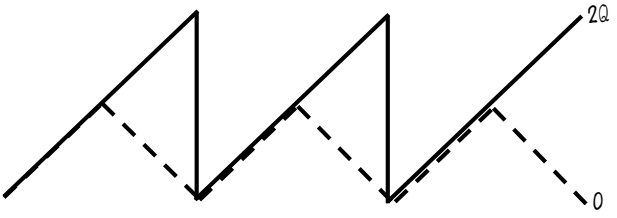


このようになっているわけですね？
どうやって実装したらいいんでしょう？これは僕がものごっついアルゴリズムを考えたので特別に教えてあげましょう。

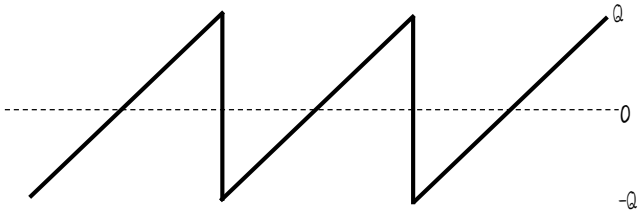
ひとまず%を使って、値が以下のようにノコギリ状に変化するようにします。



例えば変化する値がｎで、最大値がＱならば
n=(n+1)%Q;
です。でもこれノコギリなのよね～。
そこでまずは最大値を２倍にします。



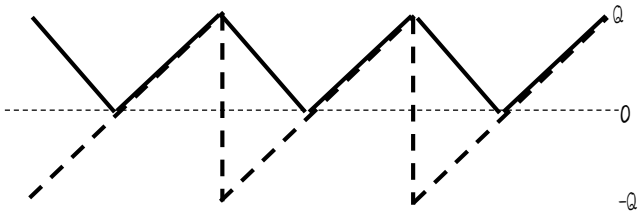
こういう状態やね？
n=(n+1)%(Q*2);
さて、これでもまだノコギリなので、全体的にＱ引きます。すると



n=(n+1)%(Q*2);

m=n-Q;

こうなるやん？あとは絶対値とればマイナスの部分がプラスに反転するので



n=(n+1)%(Q*2);

m=abs(n-Q);

こうなるわけ。これで山の形に行ったり来たりするコードが書けるわけだ。

だけどこのままじゃ動く足場としては不十分。

さて「動く足場に乗って移動」を作ってみましょう。

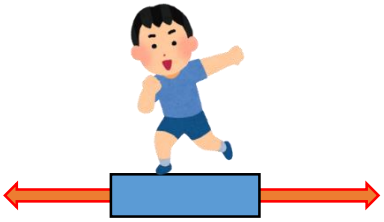
足場に合わせて動く

まずは自然のことを考えてみましょう。

スケボーに乗っているとき、足場(スケボー)だけが前に進むわけじゃなくて、乗っている人も前に進むよね？

一応物理的には摩擦であったり慣性の法則であったりとが作用して「スケボーに乗って移動する」が成り立っている。

が、そこは所詮は 2D ゲーム。もう少し簡単に楽に考えてみよう。



例えば君が電車に乗っているとする。電車の中で君が動かなければ、電車内世界での速度はゼロなんだが、外側から見ると電車の速度V_{train}で動いていることになる。

じゃあ君が電車の中で動くとどれくらいの速度になるだろう。当たり前の話だけど、しっかり考えなければならない。何度も言うがプログラミングが分からなくなる奴ってのはこういう「当たり前のこと」を軽視しすぎる傾向にある。

何度も言ってるのは、それだけ大事なことで、かつ、それを忘れちまう人がたくさんいるってことだよ。俺は大丈夫だなんて思わずに気を付けよう。

自分の速度をV_{char}とすると、外から見た速度は$V_{char} + V_{train}$とりますよね？

…つまり簡単に言うと、乗り物の速度を乗っている奴に足してあげればいい。単純でしょ？

さて実装はというと、そこがそう単純でもない。

「乗り物の速度を**乗っている奴**に足してあげればいい」

この「乗っている奴」はもちろんプレイヤーなんだ。そこに足場の「速度」を渡してあげる。
単なる矩形では「速度」を持っていないので、また例によってクラス作りましょうか…。

例えば Block ってクラスを作ります。画面内のブロック系はすべてここから派生させます。

あとは例によって例の如く BlockFactory とか作って、そこに描画と位置更新させる(Update 関数)。

僕はこの規模のプログラムをするときはたいていこのパターンになる。巷で言われている「デザインパターン」というやつは、本来はこういう「同じパターン」を体系化したものに過ぎない。
教科書とかでは「23 個のデザインパターン」とかってあるけど、そうじゃないんだ。自分で作っていいんだ。それが本来の「デザインパターン」だと俺は思うのだ。

さてちょっと話は逸れたけど、もうパターン化してるからやり方は分かるでしょ？って言えるのがデザインパターンのカやね。

逆に言うとデザインパターンの利点なんてその程度のものです。
変なデザパタ教に入らないようにね。

ちなみに、OnCollided 関数と、ファクトリを適切に使ってれば、MovableBlock::OnCollided は

```
void
MovableBlock::OnCollided(Player& player){
        player.Move(_vel);
}
```

これで済むんやで？

うん、最初は大変だったかもしれないけど C++とかオブジェクト指向とか理解して、きちんとプログラミングして、アルゴリズムをできるだけシンプルに考えれば、たったこれだけで「動く床」を実装できるのだ。

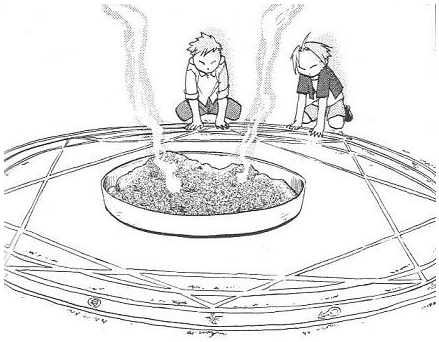
動く床は結構躓く人も多い部分だけど、難しく考えすぎずに、要素を理解、分解、再構築の手順で考えればええんやで？

まぁ、理解分解再構築しても分からんかったら、先に分解しよう。わかんなくてもとにかくバラバラにしてみる。

分解→わかるとこまで分解→理解→ちょっとだけ再構築→再構築

僕らアホなんで、錬金術師みたいに一足飛びには分からんのよ。自分をのび太だと思って徹底的に分解しよう。

分かるか？分解やで、あと図にかくんや。



言葉とか概念とかイメージなんてのは頭の中にある間は曖昧すぎるものなんや。
文章として書き起こすこと、絵や図を描くこと、実際に小規模のモノを作ること。この過程を経て「モノ」ができるんやで。
プロだってそうなのに…図を描かない人は両手をパンって合わせるだけで無から有が生み出せるとか思ってるやろ？自分をエドワードニエルリックみたいに思うとんのやろ？実際そんな奴は周りから見ればこうやで？

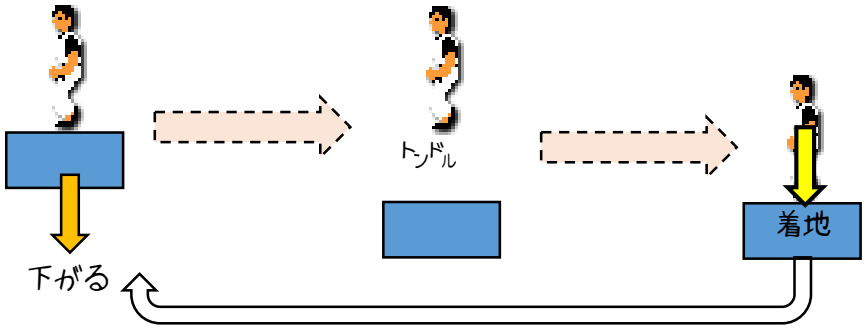
上下とかもやってみよう

一旦ここまでできたらあとは量産などたやすいものよ。

とりあえず左右のやつをコピーして名前を変えて移動方向を上下にしてみよう。
ひとまずは正常に動くはずだ。

ただし、上下の時にはちょっとだけ注意が必要だ。何かというと、床が下に動いたとき、飛んでる状態→着地状態→飛んでる状態になるのでガタガタしてかっこ悪い。

これをどうするかだが、そもそも原因が何なのかを考えてみようか。



繰り返す

うーん。これは仕方がない。というわけでとにかく辻褄を合わす方法を考えてみよう。ホントこれあと数年後には自分だけで考えるんだぜ？マジで。センセーだって明確な教科書的な答えを知っているわけじゃない。それがゲームプログラミング。

一番手っ取り早い方法としては先にプレイヤーの更新をかけてやる。まぁ、これだけでよかったりする。

重力で常に下向きの力がかかっているのでプレイヤーの更新を先にやれば確実に足場にめり込むのだ。で、押し戻されて地上フラグが立つ。

これでも上昇と下降の切り替わりタイミング(最高点)にて、1フレームだけ空中状態が発生するが毎回起きてしまうよりかはマシであろう。

疑似コードを描くとこうなる。

```
足場座標の更新
プレイヤー座標の更新
for(全ブロック){//あたり判定
	if (プレイヤー矩形とブロック矩形が衝突){
		Rect rc=重なり矩形を計算
		プレイヤーの位置を補正
		if(rc.w>rc.h){//上からのめり込みにのみ反応。
			プレイヤー着地フラグON
		}
	}
}
```

こんな感じである。

乗ったら落ちる足場

これもスーパーマリオ、ロックマン、魔界村などでよくある床である。乗って 10 フレームくらいで重力で落ちてしまう床である。

これは上下床がきっちり作れた諸君ならそれほど難しくもあるまい。

まぁ「せっかくだから、俺はメンバ関数ポインタを使うぜ!!」とやれば「停止状態」「落ち状態」をきちんと区別できる。

上から乗ったときにのみ「落ち」状態に移行すればいいです。

でいきなり落ちると「床」感がないので、「停止状態」「落ち状態」の間に落ちるまでのインターバル状態を入れておいてあげます。

```
//固定してる状態
void FixedUpdate();

//落ちる前インターバル状態
void PreFallUpdate();

//落ち状態
void FallingUpdate();
```

でいきなり固定状態→落ち状態に移行するのではなく、「固定」→「落ちる前」→「落ちる」にします。
つまり

```
void
FallBlock::OnCollided(Player& player){
	if (_vel.y == 0.0f&&_frame == 0){//上から触れたら落ち前状態へ
		_frame = 1;
		_func = &FallBlock::PreFallUpdate;
	}
	else{//実際落ち始めたらプレイヤーに速度を伝播
		player.Move(_vel);
	}
}
```

で、PreFallUpdate でカウントダウン

```
void FallBlock::PreFallUpdate(){
	DrawRotaGraph3(_rc.pos.x, _rc.pos.y, 0, 0, 2, 1, 0, _handle, true);
	_frame++;
	if (_frame == 15){
		_vel.y = 1;
		_func = &FallBlock::FallingUpdate;
	}
}
```

こういう感じやね。

3D化していこう

うん、そろそろ2Dばかりで飽きてきたと思いますので、3Dの話を始めましょう。

3Dの時も表示までの手順はほぼ同じで

モデルのロード　：　　MV1LoadModel
モデルの表示　　：　　MV1DrawModel
モデルの削除　　：　　MV1DeleteModel

の流れになります。

で、そのまま表示すると…「ちっちゃ!!」ってなると思います。手っ取り早い方法は拡大を使います。それには

```
MV1SetScale( _model, VGet( 25.0f, 25.0f, 25.0f ) ) ;
```

くらいに拡大しておきます。細かくはあとで解説します(時間ない…すまん)。

で、輪郭線がくっそでかくなってると思いますので、以下のコードをロードの直後くらいに書いてください。

```
int materialNum = MV1GetMaterialNum(_model);
for(int i=0;i<materialNum;++i){
        MV1SetMaterialOutLineDotWidth(_model, i, 0.02);
}
```

これで輪郭線がマシになります。

あとは2Dの表示に合わせるために平行投影にします。

```
SetupCamera_Ortho(1000);
```

これが平行投影をするコードです。