

ACTION GAME

LET'S ACTION GAME 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.....	6
クラス設計	6
プロジェクトを作る	8
VisualStudio の設定.....	10
下準備(いつもの)	14
DxLib をダウンロードして自分のプロジェクトに.....	14
コーディング準備	15
最低限スケルトン.....	15
スケルトンの解説.....	16
ともかくウィンドウをキープする(ゲームループを作る).....	21
スパルタンX	25
アクションの基本を作っていく	27
加速度について	27
X 方向	29
重力加速度の実装	30
ジャンプの実装	31
あたり判定とか、速度とか、幾何学系の武器.....	31
プレイヤーの状態遷移.....	32
パンチ、キックについての注意点.....	33
パンチキックのあたり判定.....	34
しゃがみの実装	34
雑談①	36
つかみ男	38
プレイヤーに向かって歩く.....	38
プレイヤーと当たるとプレイヤーは移動できなくなる.....	39
レバガチャかボタン連打で外せる.....	42
ナイフ男	43
投げるまで	43
投げる	44
ナイフ	46
体力	47
1 面ボス	47

棒男(バットマン)	47
押しあたり	48
その他の要素	50
画面 Enter/画面 Exit 演出.....	50
ツールについて	53
ここにきて C#言語です	53
フォームアプリケーション	54
C++文法の話①	54
std::string	54
ストリームについてちょっとだけ.....	57
STL って組み合わせられるのよ?	57
テンプレートについて.....	58
関数テンプレート.....	58
クラステンプレート.....	60
配列オブジェクトを作ってみよう.....	61
スマートポインタ(初歩).....	63
試しに scoped_ptr を作ってみよう.....	64
スマートポインタについて.....	68
std::shared_ptr	69
std::weak_ptr	69
スクロールしよっか'	70
おまけ	73
ヒットストップ	73
コマンド	73
文字関連	75
フォントのデータ構造と文字ラスタライズのしくみ.....	75
一文字ずつ表示	76
文字に『影』をつける.....	77
カメラまわり	78
画面揺れ(地震).....	78
より人間臭いカメラの動き	79
課題の提出	80
さあ魔改造だ	81
足場を作ろう	81
動く足場	82
等速で行ったり来たり動かすときのヒント...	82

足場に合わせて動く.....	83
上下とかもやってみよう.....	87
乗ったら落ちる足場.....	88
消える足場	89
斜めの足場	92
45°足場	92
任意の傾きの足場.....	96
下りの時のガタガタ(ジャンプできなくなる現象)について	99
3D 化していこう	104
2D に合わせていこう.....	105
『行列』とは?	105
Orthogonal(平行投影).....	108
アニメーションの基礎.....	112
アニメーションのアタッチ	112
アニメーションの切り替え.....	114
ループアニメーション.....	114
アニメーションブレンド.....	115
Effekseer を組み込む.....	119
準備	119
プロジェクト設定.....	120
なんかうまく表示されない!... ..	121
コイツも 2D 表示に合わせてみる.....	123
おまけ	126
モデルの反転	126
輪郭線の仕組み	127
文字コードと文字セットについて.....	130
文字コードと文字セットの違い.....	130
プログラミングにおける文字セット.....	133
便利なマクロ	136
マルチバイトとワイド文字(Unicode)の変換.....	137
TCHAR 型 string	141
その他文字に関すること.....	142
改行コードについて.....	142
文字おまけ	144
DxLib を使用したさらに本格的な 3D ゲーム.....	146
カメラを見してみる.....	148

あたり判定をしてみる.....	149
その他おまけ情報	150
ドット絵素材について.....	150
最終課題	152

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しやがみがある
- 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…。まあ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か後が楽になるよ(料理の下ごしらえと一緒に…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?)

まあ難しければ、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かな？

それでもよければどうぞ？

まあ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

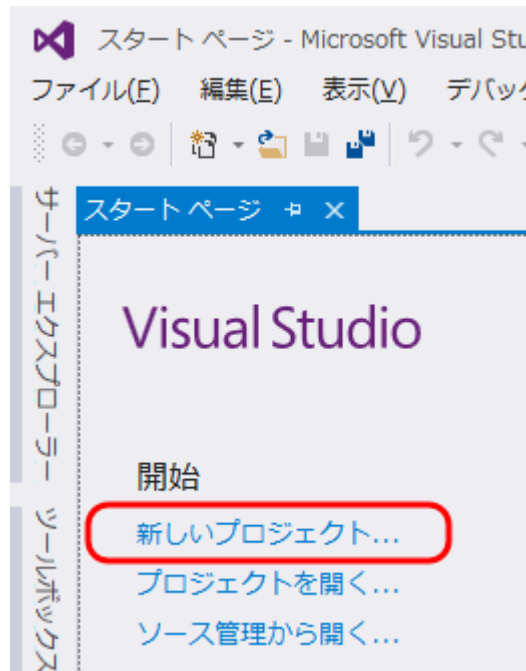
シューティングの時と同じだけど、一応書いておきます。書いてることはシューと同じです。現行の VisualStudio は 2015 が最新であるが、学校で使用している VisualStudio は一部が 2013 しかインストールされていなかったり、2015 がインストールされていなかったり、両方インストールされているのが現状です。

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。

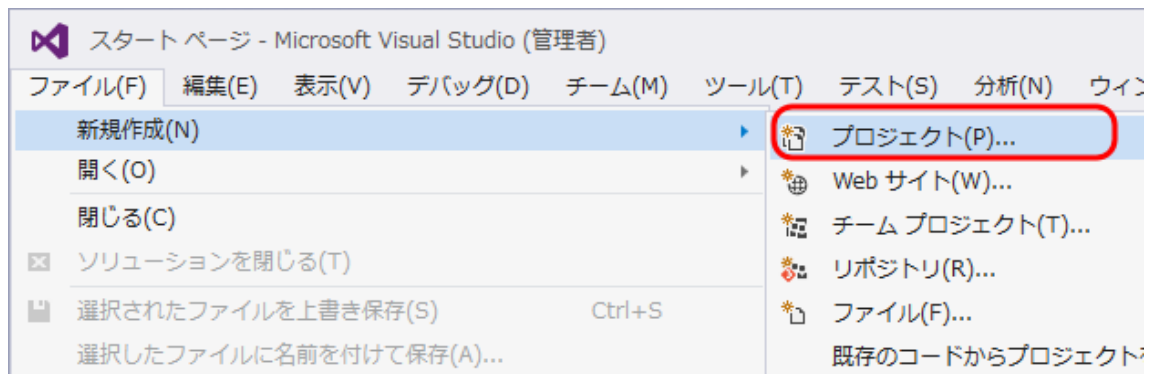
それはともかく、こういった場合どちらのバージョンで作ればいいんでしょう…ぶっちゃけ『どっちでもいい』んだけど、自分ちで開発する場合は自分ちのバージョンと合わせておこう。合わせなくても何とかなるけど、合わせておいたほうが良いだろう。

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が、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。

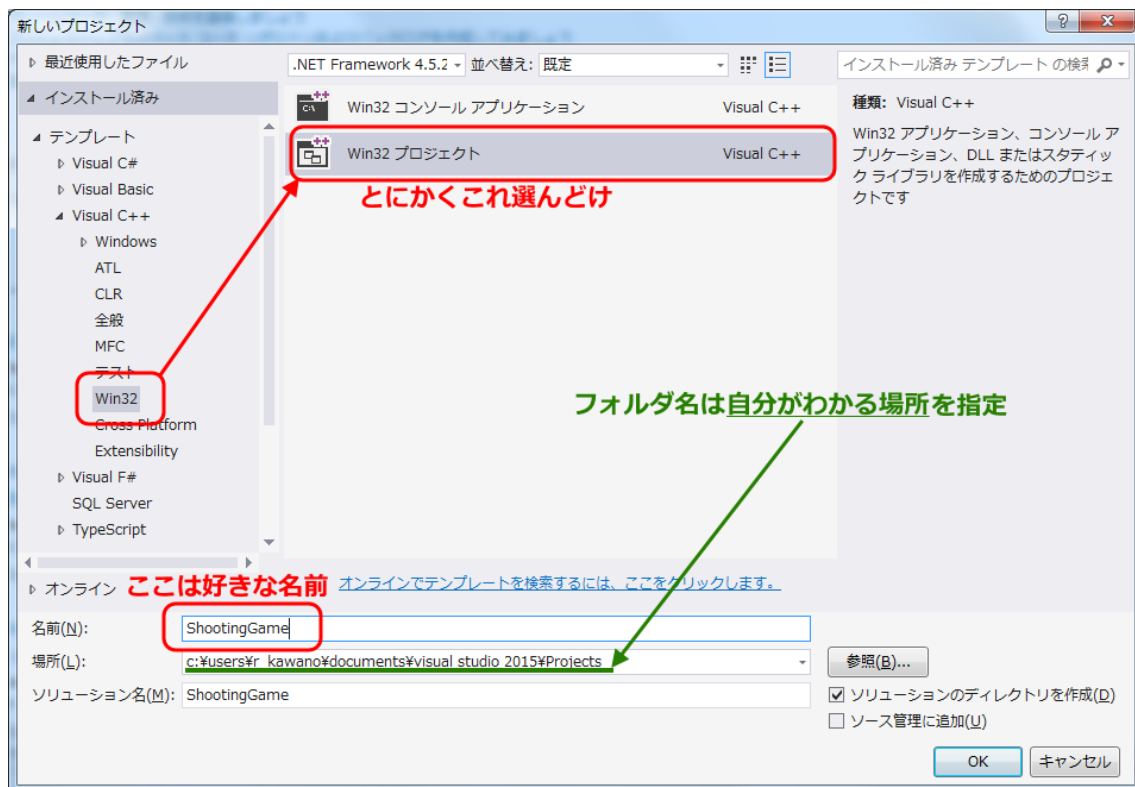
説明書は隅から隅までキチンと読みましょう。

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、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。パツパツ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。

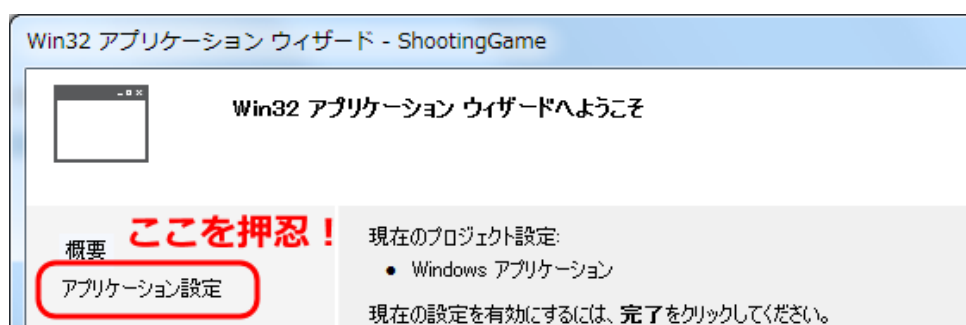
こういう画面が出てると思います。



「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？

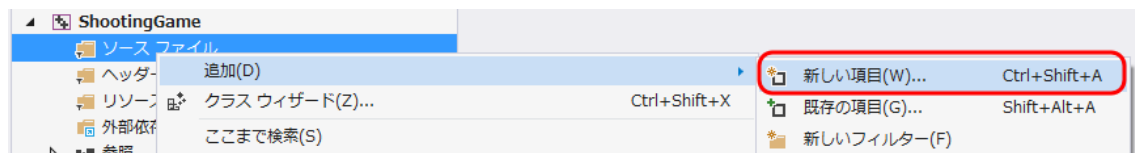


アプリケーションの種類: Windows アプリケーション

「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。

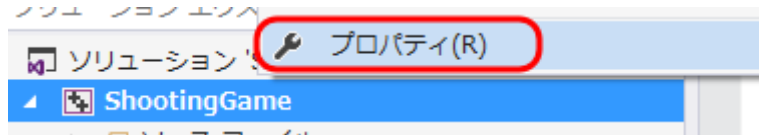
正しく出来てればソースコードの追加です。

「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。



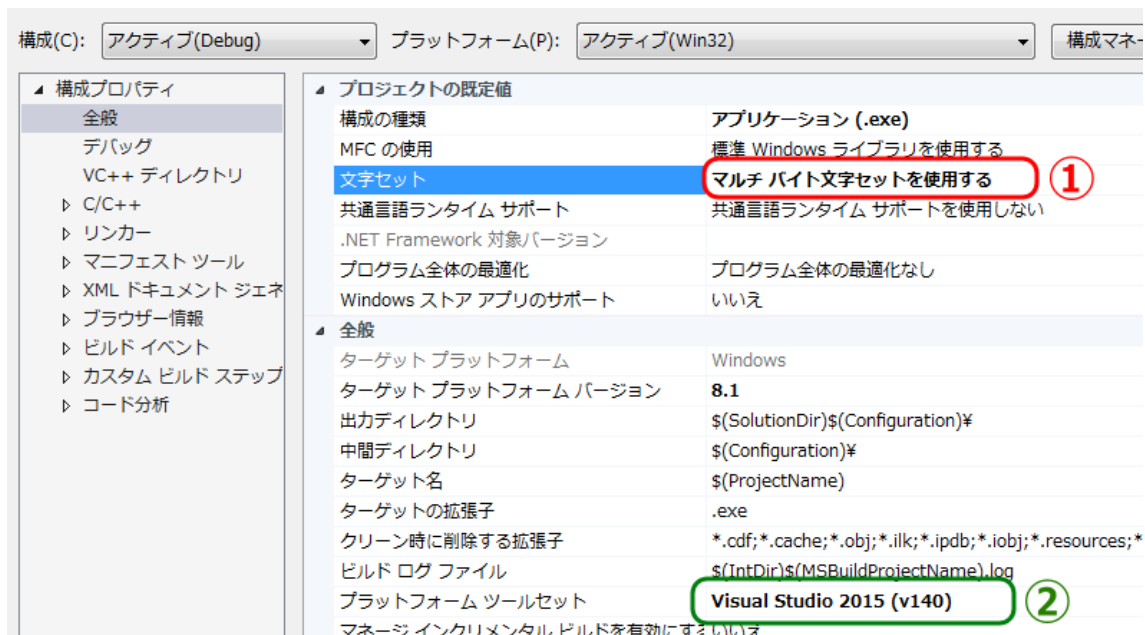
とりあえず main 関数を持つソースコードなので main.cpp ってな名前にしてください。わかってる人は別にどんな名前にしても構いません。

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。

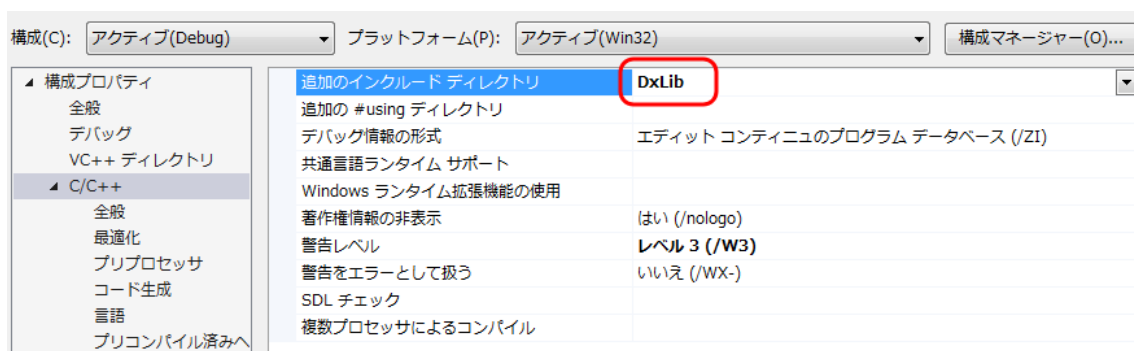
まずは「全般」を見ましょう。



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。

▲ 構成プロパティ	文字列プール	
全般	最小リビルドを有効にする	はい (/Gm)
デバッグ	C++ の例外を有効にする	はい (/EHsc)
VC++ ディレクトリ	小さい型への変換チェック	いいえ
▲ C/C++	基本ランタイム チェック	両方 (/RTC1、/RTCsu と同等) (/RTC1)
全般	ランタイム ライブラリ	マルチスレッド デバッグ (/MTd)
最適化	構造体メンバーのアラインメント	既定
プリプロセッサ	セキュリティ チェック	セキュリティ チェックを有効にします (/GS)
コード生成	Control Flow Guard	

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。

ShootingGame プロパティ ページ		
構成(C):	アクティブ(Debug)	プラットフォーム(P):
		アクティブ(Win32)
▲ 構成プロパティ	出力ファイル	\$(OutDir)\$(TargetName)\$(Target
全般	進行状況の表示	設定なし
デバッグ	バージョン	
VC++ ディレクトリ	インクリメンタル リンクを有効にする	はい (/INCREMENTAL)
▷ C/C++	著作権情報の非表示	はい (/NOLOGO)
▲ リンカー	インポート ライブラリの無視	いいえ
全般	出力の登録	いいえ
入力	ユーザーごとのリダイレクト	いいえ
マニフェスト ファイル	追加のライブラリ ディレクトリ	DxLib
デバッグ	ライブラリ依存関係のリンク	はい
システム	ライブラリ依存関係の入力の使用	いいえ
最適化	リンク ステータス	
埋め込み ID		

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『センサ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だったのです。

『なんかおかしく』なったらぜひともセンサ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/dxdownload.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。まあ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) その中を見てもよ 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、検索が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が、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？ C++ は全然やってませんよね？ ちよいちよい僕は無意識に 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 センサーに聞くか、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++ の生みの親のビヨ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

進みなさい。

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;
}
```

とやってやったでしょ？この『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

「信頼できる情報が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
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。

たまにいますけど、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』は合わせておかねばならないので、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いらないです。

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こと

- 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。まああんまり厳密にやってい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って使っって『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ワ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。

まあ…いずれ通らなければならぬ道なので、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』って思っとう。

そして失敗すると-1を返すと思っとう。

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 は内部で色々を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。
さあ、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いない。

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://dxtlib.o.o07.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。

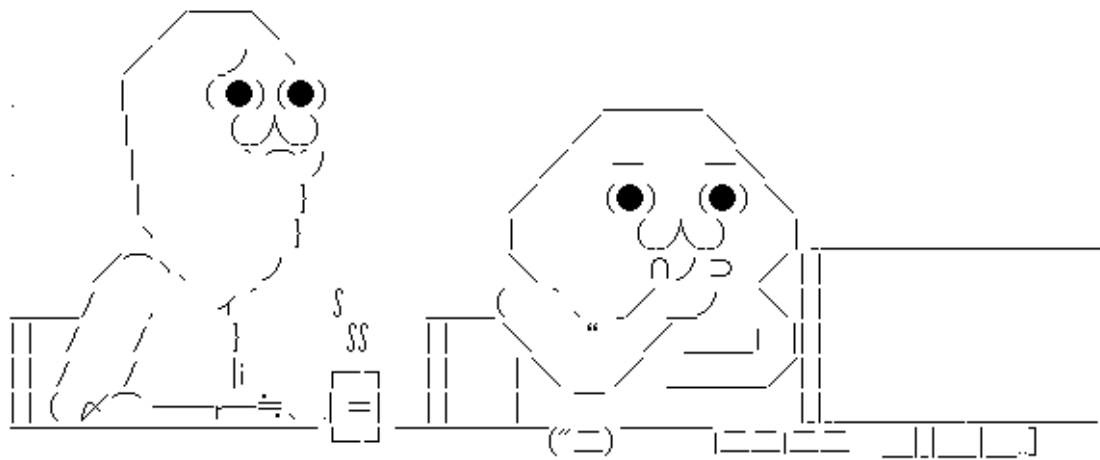
目安としては 60 分の 1 秒に一回程度、用はゲームのループに 一回程度です。この関数を呼び出さない
と結果としてシステムが 異常に重くなったり不安定になったりします。

あと、戻り値が -1 (エラー発生・若しくは DX ライブラリのウィ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
うしないとウィ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、まあ…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？ unnecessary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ないかね？って思えばいい。そうすることで

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

と書ける。さあ一行減った!!

まだ減らせませんか？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すね？

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たいでしょうか？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。これはマジでヤバイ。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 a だとすると、速度 v に加速度 a を足すと $v+a$ ですよ？

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、 $v+a$ は一回ぶんの話です。

加速っていうのは一回でおわりやしない。「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 $v+a$ 次は $v+2a$ 次は $v+3a$... 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

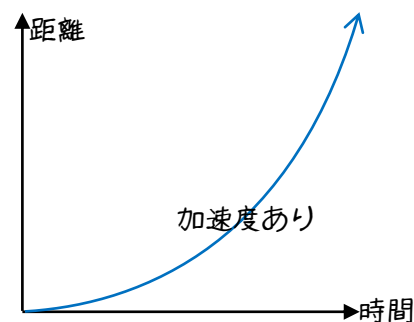
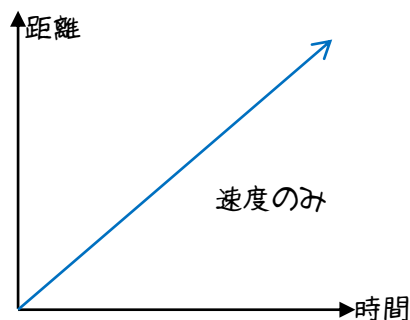
$$v = v + a;$$

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

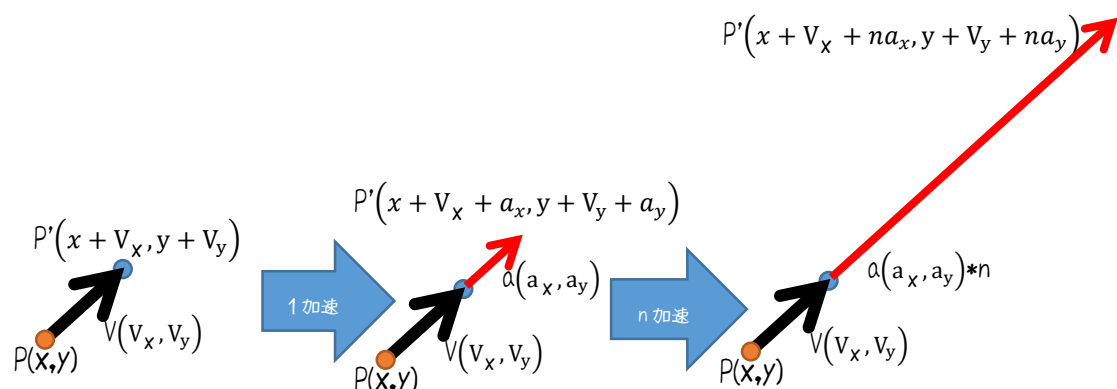
分かるよな？一回で進むスピードがドゥンドゥン早くなりながら、現在座標も変わっていくので、速度オンリーの時と加速度ありの時では



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よ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。

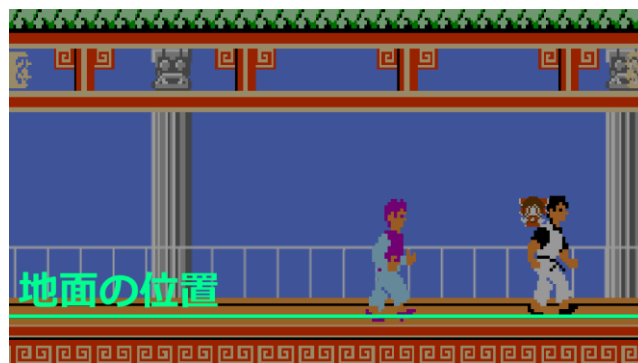
これ以上は説明しませんので、なんとか考えて、右おしたら右、左押したら左に向くようにしてください。

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。ほんとに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

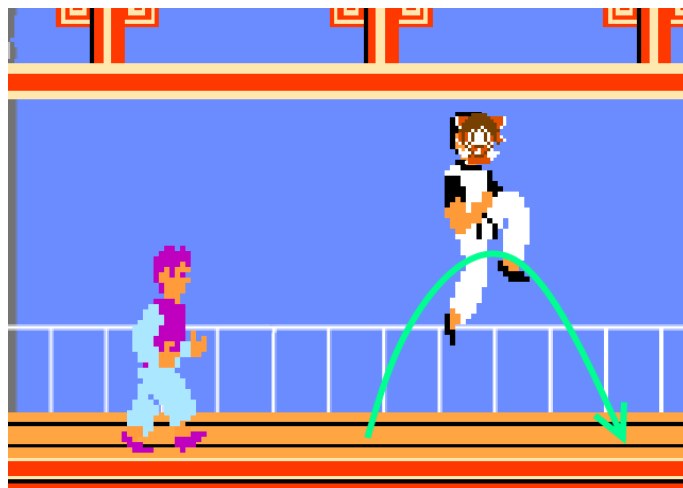
ジャンプの実装も簡単です。

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…まあ、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、またー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。

まあ、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、センサーが作ったのと違ったら、どう違うのか考えてみよう。

センサ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。

何度も言いますが、『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よつ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。

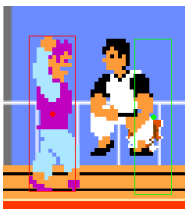
相手がつかみ男なら耐久力1なので、当たった瞬間にKillしておk

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、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ておけばいいからです。

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けど、DXLibにはグラフィックの大きさを測る関数があって、

GetGraphSize っていう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みたい…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ぬ。キツいだけやねん。でもな…でもね、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。とくにクリエイター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…



というわけです。頑張ってください。

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。頑張らしましょう。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』『違うんじゃない!』とか言いながら作ってますよ。皆さんも悩んでください。センサーがゼーンが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たても疑似コードみたいなもんです。そのまますき写して動くはずがねえだろう…。

プレイヤーに向かって歩く

これ、簡単ですよ。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。左上だとか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 もそこから継承する。Collider のメンバとして、『キャラタイプ』みたいなものを用意して

```
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、ちよつと違うかなー。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、ちよつと違うかなあ…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が参照でも持たせておきます。

ですから、うーん。

class GameObject ってクラス作つといて、Enemy も Player もそこから継承する。

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

とくに中身は必要ない

```
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ッて『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。

まあ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？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…まあ、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り落とす。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』にしているなら

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

```

    }
}
なんて書く。

```

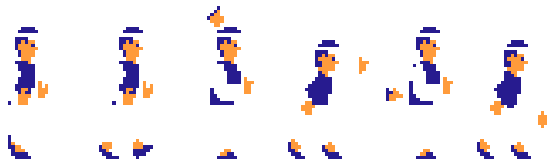
で、すでに OnCollided で「相手のこと」が分かる状態であるとする

```

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な)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。

まあ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、まあ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

$\text{rand()} \% 150$

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？

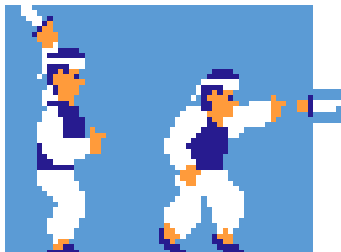
さて？

$a \leq r \leq b$

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い投げ方になります。ここは自分で調整してみてください。

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、だいたいこれでいいです。

ナイフは『弾』つまり Bullet として扱います。

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 unnecessary 弾の削除も行います。

ちなみに僕は unnecessary 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
 扱いますけえな？

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、とにかく『攻撃矩形』だったらライフをひとつ減らします。

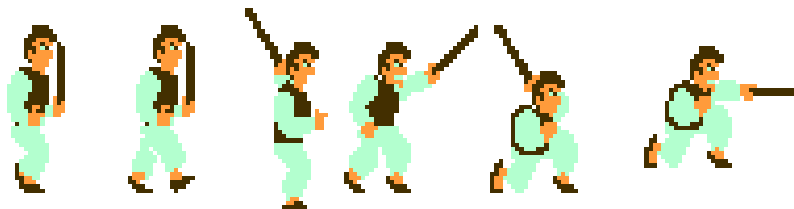
そしてライフが0になったら『死亡状態』に移行すればいいのです。

…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、その場に当たり判定が残り続けるのでジャンプで躲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、まあこれくらい。

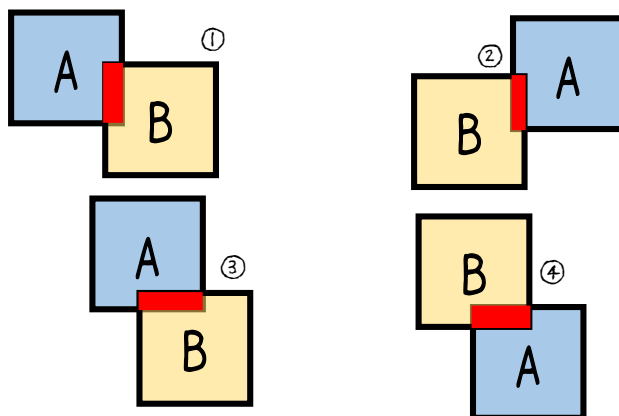
ナイフ男までを作れた人なら簡単ではないかと思います。

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。



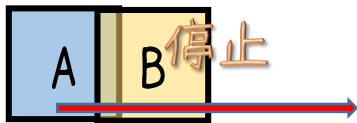
こんな図があって、小さいほうに押し返せばいいんじゃないかな？まあ今回までは①か②のパ

ターンしかないなので右か左に押し返せばいい。とりあえず A が B を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こ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こと、半分半分押し返した場合、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、割りゃいいんですよ。足す→ $80+40=120$ 。割る $80/120, 40/120 \rightarrow (2/3, 1/3)$

$2/3+1/3=1$ 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、ここまできちん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ー関数ポイントを用意して

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あります。

```
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。

まあ、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『ツララー(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が多いで)

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いました(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、自分で作っていました(°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。

(うええー、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いますがねえ…。

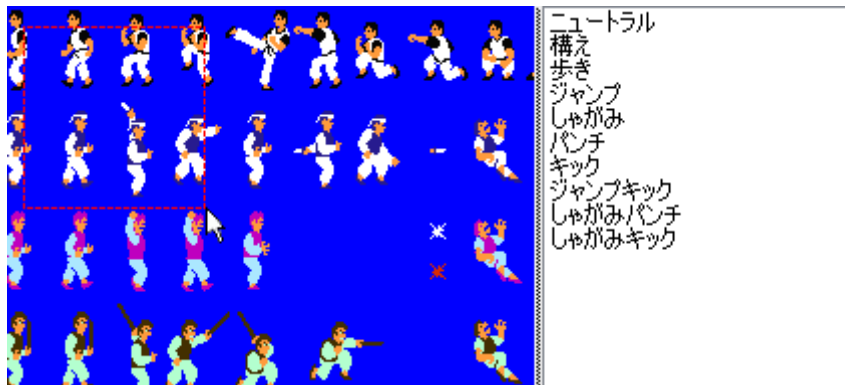
フォームアプリケーション

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？うえーんもういややー(つム)

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ないのかなあ…(*・ω・*)ナゲネ…

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…まあ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らう)

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+="はかうだよ";
```

とすれば str の中身は"中身はかうだよ"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()`

まあそういう事です。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ことです。

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、まあ今のところはそれほどありがたくないかもしれませんが、知っておくのと知ってい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、組み合わせで使えます。
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-](http://f3.tiera.ru/other/DVD-009/Alexandrescu_A._Modern_C++_Design(C).Generic_Programming_and_Design_Patterns_Applied_(2001)(en)(271s).pdf)

[009/Alexandrescu_A._Modern_C++_Design\(C\).Generic_Programming_and_Design_Patterns_Applied_\(2001\)\(en\)\(271s\).pdf](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え。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。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が、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ありがたみがありませんが、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らうと思うと

```
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。さあ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よ。

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。まあ、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ありますので、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たいほうがいいです。

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よ。

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……ほうほうで『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。まあ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ら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
```

```
    }
```



```
}
```

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 人以上で開発してい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。

まあ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
- \rightarrow 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。こいつが裏の主演となります。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つ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` 関数を作して下さい。

さて、ポインタのようにこいつが拳動するのであれば \rightarrow 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？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ください。

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ください。

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(b);  
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。

b

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、ポインタをベ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。

スクロールしよつか

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るですよ。

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(ハの参照)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、プレイヤーハの参照持たせとく…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ぬ…。

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なんですけど、ステージ全体から言うと 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が英語ではこれを Offset といいますので、それ考えましょう。

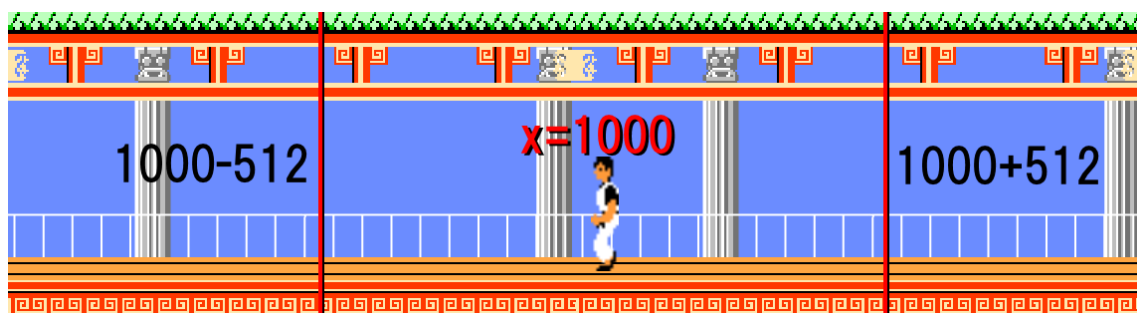


やみくもに考えて分からない場合の考えるヒントとしていくつかの

『実例』で試してみるってのがあります。数学でもそうなんですけど、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クがあります。

どういうことかということ、 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、ちよつ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。

まあそんなこんなでカメラクラスを書くとヘツタ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
    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、まあなんか使うかも
    float OffsetX(); //これが目的の品やで
};
```

どやあ…？まあヘツタ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://dilib.o.o07.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。

まあ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=yxTzcOYfVT0>

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。

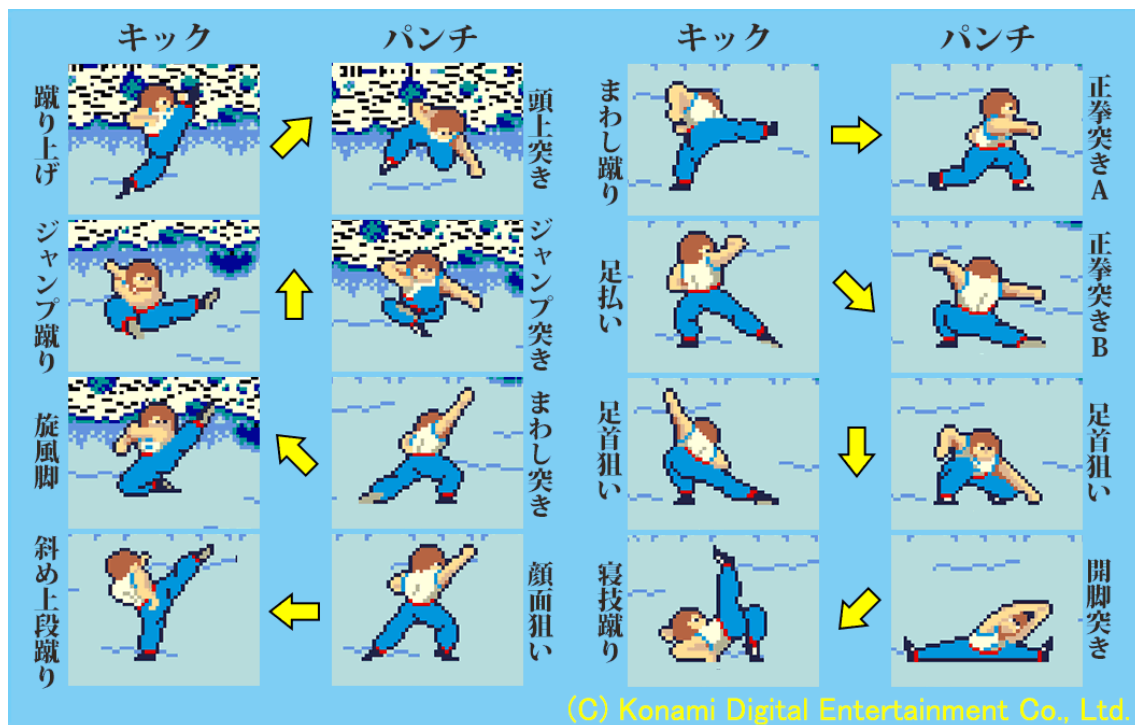
コマンド

以前にちょっとコマンド実装の話をしたましたが、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に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おうと『ベジエ曲線』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『ベジエ曲線』で出来とるわけちゃうで。

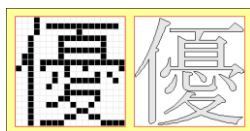
大きく分けると『ビットマップフォント』と『アウトラインフォント』ってのがああるんや。

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？

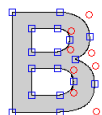
でも、それだと問題が起きるんだ。例えば今みんなが見ているドキュメントなんかそうだよな？

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。

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『ベジエ曲線』のコントロールポイントデータになっている。



アウトラインフォント



ドットフォント

上の図の○とか□が『コントロールポイント』である。まあベジエ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エ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たね。

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が『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。

前にも話しましたが、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";
```

これはオッケーかな？

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 フレームごとに 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により、より迫力のある演出を試みましょう。

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。

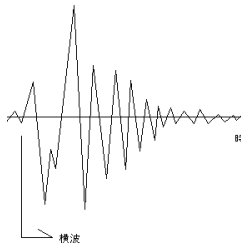
ではやってみましょう。

```
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。

そうすると 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。

まあ、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

```

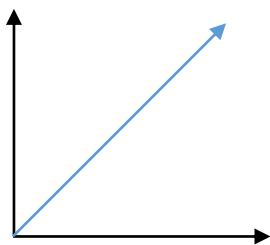
Camera::Quake(float power){
    _quakeFrame=60;
    _quakeOffset=Vector2(power,0);
}

```

とでもしておくわけやな。
 これが画面揺れの実装や。

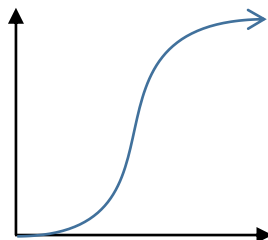
より人間臭いカメラの動き

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

人間のカメラマンやったら、こうは動けんのか。どうしても最初にラグが発生する。
 というわけで、人間がカ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、



人間ばい動き

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。

まあ、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。

課題の提出

2016/07/22(金)の 18:00 までに課題を提出してください。提出フォルダは

¥¥132sv¥gakuseigame¥game¥ゲーム PG131¥SpaltanX

の下に、R-Type の時と同様に、タイトルバーに自分の学籍番号と名前が出る exe ファイルを自分の学籍番号と名前が分かるフォルダにつめて提出してください。

とりあえずできているところまでで採点します。

採点ポイントは

- 状態遷移ができているか
- あたり判定ができているか
- 着地等がきちんとできているか

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いいかもしれません。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

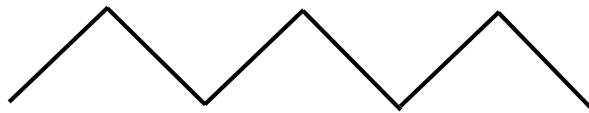
これ結構みんな出来ないんだよねえ〜。

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。

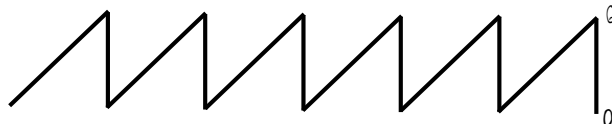
行ったり来たりってことは動きが



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。

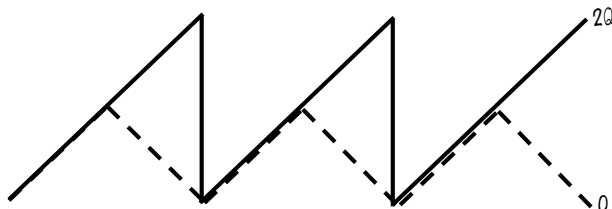


例えば変化する値が n で、最大値が Q ならば

$$n = (n+1) \% Q;$$

です。でもこれノコギリなのよね〜。

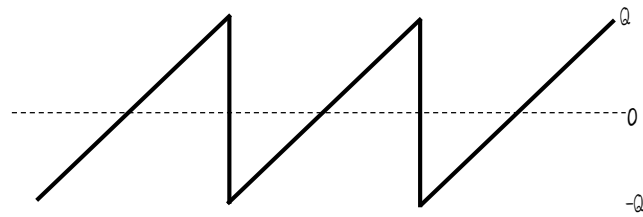
そこでまずは最大値を2倍にします。



こういう状態やね？

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

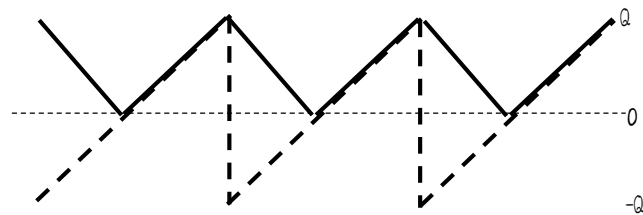
さて、これでもまだノコギリなので、全体的に Q 引きします。すると



```
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

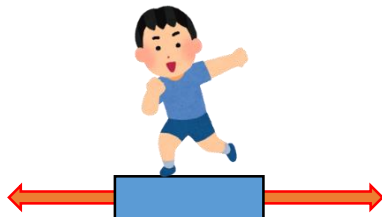
まずは自然のことを考えてみましょう。



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か作用して『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き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、かつそれを忘れちゃ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 $V_{\text{char}} + V_{\text{train}}$ 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んです。
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。

動く床は結構蹴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？



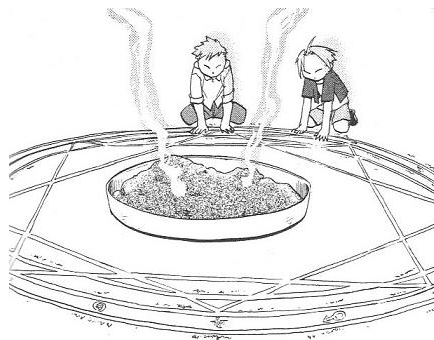
まあ、理解分解再構築しても分からなかったら、先に分解しよう。わかんなくてもとにかくバラバラにしてみる。

分解→わかるそこまで分解→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？



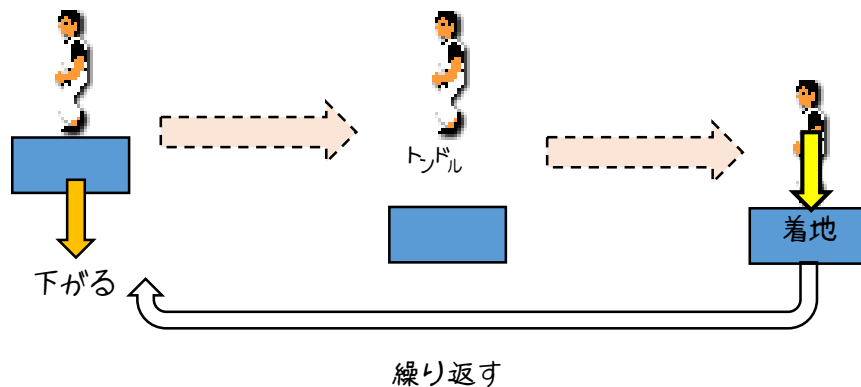
上下とかもやってみよう

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つこ悪い。

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。



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せ方を考えてみよう。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。センサ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。まあこれだけでよかったです。

重力で常に下向きのかが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。

そ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はマシであろう。

疑似コードを描くところなる。

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。

まあ『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。

消える足場

これはここまでついてきたみんななら簡単だと思います。簡単だと思います。そう思います。

ヒントを言っておきますと

出現前待ち状態→出現途中状態→出現中状態→消滅途中状態→消滅中→最初へ

という状態遷移をループすればいい。状態遷移は例によって例の如くメンバ関数ポインタを使えばいい。関数をこう作って…

//見えてない状態

void WaitUpdate();

//出現途中状態

void AppearUpdate();

//出現中状態

void NormalUpdate();

//消滅途中状態

void DisappearUpdate();

これを宣言して…

void(VanishBlock::*_func)();

こうやな

_func = &VanishBlock::WaitUpdate;

で時間ごとに切り替える…と

++_frame;

if (_frame == vanish_interval){

 _func = &VanishBlock::AppearUpdate;

 _frame = 0;

}

で、出現途中はαブレンドを使って半透明にしていく

++_frame;

SetDrawBlendMode(DX_BLENDMODE_ALPHA, min(vanish_interval_alpha, _frame) * 255 /
vanish_interval_alpha);

```
DrawRotaGraph3(_rc.pos.x + _player.GetCamera().OffsetX(), _rc.pos.y, 0, 0, 2, 1, 0,  
_handle, true);  
SetDrawBlendMode(DX_BLENDMODE_NOBLEND, 0);
```

だが、これは見た目だけの話。あたり判定にも気を使わねばならない。

あたり判定に関しては『半透明あたり判定』など存在しないので、当たるか当たらないかだ。とりあえず、半透明の時に当たるか当たらないかは各自の判断にお任せしよう。僕は『出現中』しかあたり判定を付けていません。

で、あたり判定がついたり消えたりするので、ブロックの大元にも『isAvailable フラグ』を付けておこう。

そして状態が遷移するときにこのフラグをいじって切り替える。

後はあたり判定部分で IsAvailable を確認する処理をつけ足せばよい。

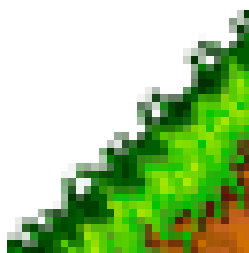
```
//プレイヤーと地形のあたり判定(ブロック)  
for(auto& block:_blockFactory.GetBlocks()){  
    if (!_player.IsActive()) break;  
    if (!block->IsAvailable())continue;//ここやで  
    if (CollisionDetector::IsHit(_player.GetRect(), block->GetRect())){  
        Rect rc=CollisionDetector::RectOfOverlapped(_player.GetRect(), block->  
GetRect());  
        CollisionDetector::AdjustOverlap(_player.GetRect(),block->GetRect());  
        if(rc.w>rc.h){  
            block->OnCollided(_player);  
            _player.OnCollidedStage();  
        }  
    }  
}
```

これで消える足場は完成です。

斜めの足場

次に斜めの足場について語っておこう。簡単に実装できるのは角度が固定の足場だろう。

45°足場



悩ましいのが画像だが、角度が決まっているのならあらかじめ画像を作っておいてもいいだろうし、DrawRotatedあたりで45°回転させてやってもいいだろう。

45°足場の何が簡単ってそれは君イ……あれだよあれ。

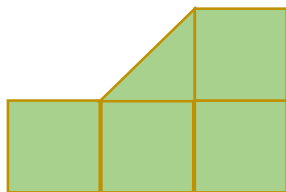
ここを歩くときは水平方向と同じだけ垂直方向に動かせばいいんだよ。簡単じゃろ？つまらんこと聞くなよ。

ええっ!?あたり判定どうするのかって？それは君イ……えーっと、その…。

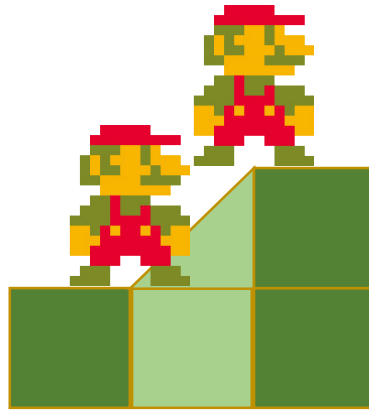
というわけで斜めのあたり判定は結構悩ましいものです。だけどそこまで難しくもない。こんな2Dの45°斜め床ごときにOBBを持ち出すものではありません。なんでも『シンプルに』考えろよ。

通常の矩形のあたり判定はどうやってましたっけ？考え方的にはX方向を確認してY方向を確認するという二段構えやったはずや。

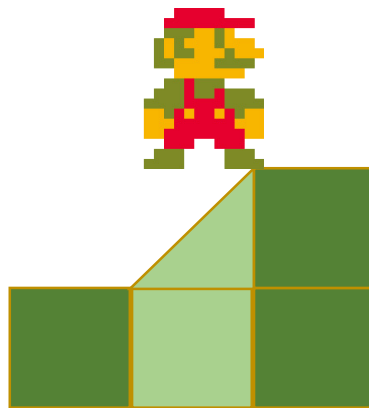
いきなりOBBなんちゅう難しいところに飛びついて『やっぱゲーム開発は難しい』とか言わんと、分かるところを武器に攻めていけばいい。



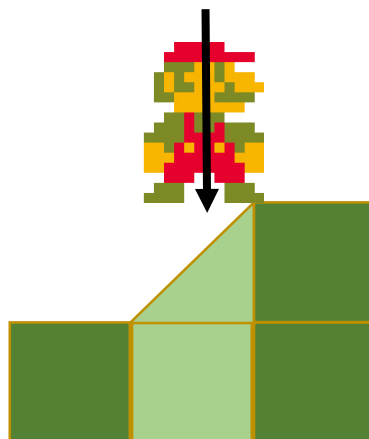
例えば床がこうなっていると。真ん中の部分が45°床やな。まずは普通にあたり判定する(まだ補正はしたらあかんぞ?)。



さあて、斜め床さえなければマリオくんは通常どおりやればええんやが…ここで通常通り補正をかけちゃうと…

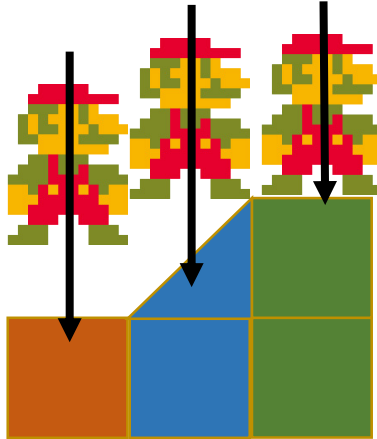


こういう事にもなってちょっと気持ち悪いんやで。というわけでマリオ君にぶっとい杭を刺す。



何にもしてないのに、ベクトル杭を刺されたマリオ君がかわいそう。このぶっとい杭はマリオ君のちょうど真ん中に刺さってると思ってくれ(キャラクターによっては真ん中じゃなくてもいい。つまりところ、何処を基準にするかという事が重要なのだ。)

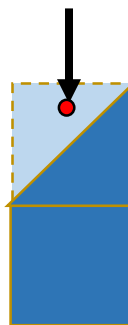
なぜこの真ん中のことを言っているかというと、この真ん中の杭が何処とのあたり判定を採用するのかということに関わるからだ



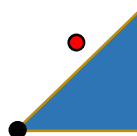
真ん中が左の矩形領域にあるなら左の矩形を採用。右の矩形領域にあるなら右の矩形を採用。そして真ん中の領域にあるなら真ん中のあたり判定を採用する。そうすれば比較的に自然なあたり判定になるはずだ。

さてここで問題は真ん中の場合ですね。

まず、あたり判定自体は



こういう感じでとってもらっていい。もしこれで当たっていたら次の話だ。今度は垂直方向を見る。そして現在のプレイヤーの下端Y座標を見る。この下端Y座標を何と比べるかということ...



左下の点と比較する。カンがいい人はわかったんじゃないかな？

せやで。

赤い点と黒い点のX方向の差をまずは求めるんやで。

```
diff=赤.x-黒.x;
```

左下から求めるのだから、この結果は必ずプラスやな？となるとこれ45°だから

```
if(赤.y < 黒.y-diff ){//当たってない
```

```
}
```

もしくは

```
if(赤.y >= 黒.y-diff ){//当たってる
```

```
}
```

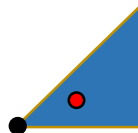
というわけ。ごめんけどこれが分からんという人には斜め床の実装はまだ早いと思うねん。
おとなしく縦横床で我慢しといてな。

さて『当たってる』は何となく分かったと仮定して、じゃあ『押し戻し』つまり『補正』はどうしようか？って話ですが、これもなんてことはない。

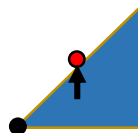
当たってるってことは

こういう状態なわけ。

分かりますよね？



どう補正したらいいかというと



こうだよな？Y軸方向の『あるべき場所』にまで押し返せばいいだけです。
となると

```
赤.y -= 赤.y-(黒.y-diff);
```

だよな？

この赤.y がプレイヤーの下端Y座標と思ってもらえばいい。

そういう話です。

どーですかねー？

これが 45° 足場です。

45° 足場なら横 diff と縦 diff は同じなので楽なのですが傾きが変わるとどうなりますかねー。

任意の傾きの足場

45° なのは $y=x$ です。ですから数学的に書くと $y=f(x)$ 。そして 45° は $f(x)=x$ ですから x が $x+diff$ になると $f(x+diff)=x+diff$ ですから y も $y+diff$ でした。

例えば傾きが $a=\frac{1}{2}$ の場合 $f(x) = \frac{1}{2}x$ ですから $f(x + diff) = \frac{1}{2}(x + diff)$ です。ちょっと数学チックに

なってますまないが、これくらいは理解してくれ。

つまり、傾きが $\frac{1}{2}$ の場合、 Y 方向の変化値は $\frac{1}{2}diff$ だけ変化するというわけ。なんとなくわかるかな？

斜面は英語で「Tilt」か「Gradient」です。まあどっちでもいいですけど、字数が少ない Tilt にしましょうか。

// 任意の斜め足場

// 情報は左(始点)と右(終点)

// つまり「線分」のデータを持っている

// 左右の座標を設定した時点で傾きを求めるようにする

class Tilt

{

private:

Position _leftP; // 左の点

Position _rightP; // 右の点

float _a; // 傾き

public:

Tilt();

~Tilt();

///
// @brief 座標を設定する

///
// @param leftP 左の点

///
// @param rightP 右の点

void SetPositions(Position& leftP, Position& rightP);


```

    bool IsCollided(Rect& rc); //当たったらtrue返す
    void AdjustY(Rect& rc); //押し返す
    void Draw(); //表示処理を行う
};

```

クラス設計としてはこんな感じで考えます。

で、SetPositions にて座標設定するときに傾きも計算しておきます。

```

void
Tilt::SetPositions(Position& leftP, Position& rightP){
    _leftP = leftP;
    _rightP = rightP;
    _a = (_rightP.y - _leftP.y) / (_rightP.x - _leftP.x);
}

```

フツーに中学数学の『傾き』の計算やねんで？分かりますやろ？さて、傾きの事前計算ができます。

あとは X 座標から Y 座標を求める計算式を考えましょう。

求めたい Y 座標を ypos とし、入力の X 座標を xpos とすると、

```
ypos=xpos*_a;
```

という答えに飛びつきたくなる。が、プログラマってのはいきなり答えに飛びついてはいけな
い。ちょっと慎重になろう。パッと見の答えに飛びつくってのはマインスイーパで『考えずにマ
スを選ぶ』ようなもんですよ？

直線の方程式には『切片』が必要ですよ？ $y=ax+b$ の b ですよ。

そのままちょっと考えようね？この線分の始点ってどこでしたっけ？左端を始点とするとあ
るべき Y 座標は

```
ypos = _left.y + _a*(xpos-_left.x);
```

となります。で、重力は下に向くのでこの ypos がプレイヤーの Top() と Bottom() の間にあれば
『当たっている』ってことです。

とりあえずはそれで『当たってる』が判定できますね。

ですから X から Y を求める private 関数を作っておいたほうがいいですね。

う〜ん。

```
float GetYFromX(float xpos);
```

という関数を作りましょう。中身は先ほどの説明を見て自分で考えましょう。

SetPositions 前にこの関数を呼んでも意味がないことに注意しましょう…理由は分かりますね？

で、こんな感じ。

```
IsCollided(Rect& rc){  
    //まずはX方向判定  
    if (!(rc.Right() < _leftP.x ||  
          _rightP.x < rc.Left())){  
        return false;  
    }  
    //次にY方向判定  
    float xpos = rc.Center().x;  
    float ypos = GetYFromX(xpos);  
    if (rc.Top() < ypos && ypos < rc.Bottom()){  
        return true;  
    }  
    else{  
        return false;  
    }  
}
```

説明のためにまどろっこしい書き方してますが、もっと短くできますので、各自行数は減らしておいてください。

あえてソースコードを書いてますが、丸コピペすんなよ？理解できないなら使うんじゃないぞ？これはあくまでも参考だ。理解して使えな!!

さて、ここまで来たら補正なんですけど

```
赤.y -= 赤.y-(黒.y-diff);
```

を思い出してください。これは傾きが1の時でしたね？ですから

```
赤.y -=赤.y-(黒.y-diff*a);
```

ってことになりますね？もしくはもっと手っ取り早く考えると

「要は斜め線の上に乗っけりゃいいんだから」

赤.SetBottom(左端.y+a*(xpos-左端.x));

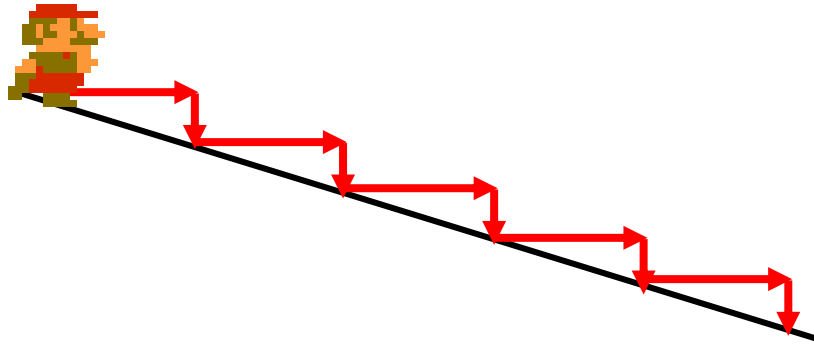
としても構わない。自分が思うシンプルで考えよう。

下りの時のガタガタ(ジャンプできなくなる現象)について

45°だろうが任意角度だろうが下りの時にはガタガタとします。これはどうやって対処しましょう。

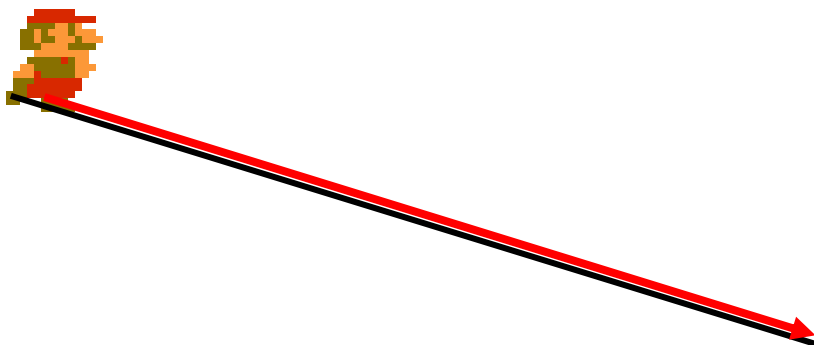
とりあえず原因を考えてみます。

プレイヤーが左右に動くときというのは文字通り左右に動きます。ですから道に沿ってというより…。



ごらんのように右に進んでは重力で落ちていきます。そらがタガタもしますわな…。そもそも自分が坂道を降りる時を考えてみよう。こんな動きはできないでしょう。

自分が坂道を降りるときには道に沿って



このように下っていくでしょう？

つまり斜め足場に着地している間は、左右の移動時はこのベクトルに沿って動かなければならないってことです。

さて、どうしましょう？

数学的には簡単なんですけど、プログラムの的にはどうするのが適切でしょうか。
何度も言いますが『正解などない』んです。ぶっちゃけ俺も『思いつき』です。でもゲームなんて『思いつき』の集大成で成り立ってるんですわ。
まあ、とりあえずどうしたらいいかわからない時は、3つ『参考になるものはないか?』『妥当性の検証はどうするのか?』『ほかにいい方法はないのか?』を考えましょう。

参考になるものというとすぐ他人のコードとか、解説ページを見たがるだろうけど、そうじゃない。それでもいいけど、ただ『探したけど見つかりませんでした』とか言わないようにね。他人のコードを採すだけの癖がついちゃうと、自分の考えの幅を狭めるからね。

参考ってのは本当に『どっからでも』探してください。よそのゲームの動きをよ〜く観察してみる。あとは数学的なもの、物理的なもの、生物的なもの、日常的なものをよ〜く観察してみる。それがコツです。

『妥当性の検証』ってのは、それが正しいかどうかってこと。他人のコードを引っ張ってくるのはいいけど、それそのものが正しいか。正しくても自分のソースコードに適用できるか。そして、自分で思いついたアルゴリズムに関する妥当性の検証の場合は、まず紙に書いてください。何度も言ってますが、これは重要です。数式でも図でも書いて検証してください。

検証しづらいものなら実際の数値を入れて計算してください。それができたら一般的入力としてXとYの関係を検証してください。

そしてテストプログラムを書いて検証してください。ここまでが妥当性の検証です。

あとは『ほかにいい方法がないのか?』ですが、これもプログラマの悪い癖でよくあるのですが、最初に思いついたアルゴリズムに飛びついちゃう。そして固執しちゃう。よ〜〜く考えたり、探したりすればもっといいアルゴリズムがあるかもしれないのに…。アルゴリズムを考えたら『1度だけ』疑ってかかってみよう。もっといい方法はないのだろうか。

さて、話が逸れたが大事なことでしたので話しました。
一応ここからは僕の『思いつきアルゴリズム』を書いていきます。あくまで参考程度に考えてください。

1. プレイヤーが傾きを持つようにしておく(メンバ変数に傾きを)
2. とりあえず斜め坂に衝突している間は傾きを更新する
3. 左右が押されたらその傾きを元に移動する(傾き0ならただの左右移動だが、それ以外の場合は斜めに移動するように)
4. 斜め足場から離れた時点(通常足場に乘っかる、ジャンプする等)で、傾きを0に戻す
5. 傾きの違う斜め足場に移った時点で傾きを更新する

如何だろうか…？まあ『思いつき』なので、参考程度に考えてください。

う〜ん。ガタガタはしなくはなったんですが…接地判定出力をしたところ…

```
IsGround  
NotGround  
NotGround  
IsGround  
NotGround  
IsGround
```

と、地面と空中がまだ混在してるんですよ。すべて接地してるはずなんですけどね。

これだとジャンプできないタイミングが出てしまうので、正直使い物にならんのですよね…

ちよつと渇まんが時間が無いので、もう少し待ってほしい。

はい、1時間かけてもよくわかりませんでした。

ここで諦めるのか？

そうではなくこだわりを一旦捨てよう。プログラマは完全なプログラムを目指すことが理想だが時間が無いとき(納期が迫っているとき)はその限りではない。

あまりやりたくはないのだが『斜め床専用の処理』を追加することにしよう。

- ①『斜め床』に着地したら斜め床用のフラグを立てる。
- ②ジャンプやX軸離れが発生すればフラグを落とす。
- ③そして歩いている間は床面に強制吸着させる(Y座標を床面に固定し、着地フラグを立てる)。

こんな感じのプログラムでなんとかお茶を濁そう

ホンマはこんなプログラム良くないんだけど、1時間一つのバグに取り組んで答えが出ないなら、そのまま時間を延長しても無駄なことが多い。とりあえず辻褄合わせといて別の作業に取り掛かるとか、一晩寝るとかすると解決したりする。

そういう意味で『あきらめる』のとはちょっと違う。

一つ言っておくと、こういうフラグがいたずらに増えてしまうのはよくないプログラミングです。最終的には管理できないレベルのフラグが乱立することになりますので、今回は仕方ない措置だと思ってください(´;ω;`)

すまぬ…ワシが不甲斐ないばかりに。

```
bool _isTiltGrounded; //斜め床着地フラグ
```

こんなフラグを作っておいて…

```
void  
Player::OnCollidedTilt(Tilt* tilt){  
    _a = tilt->Gradient();  
    _isTiltGrounded = true;  
    _currentTilt = tilt;  
}
```

こんな感じで着地イベントで渡してあげて(ポイントのぬるぽで判断すればフラグ自体はいらないかも)

で、現在の斜め床オブジェクトを持っているので左右に歩くときにもう補正(吸着)しちゃいます。

```
if (_currentTilt != nullptr)  
    _currentTilt->AdjustY(_rc);
```

うーん。気に入らないなあ……まあ仕方ないな。

とりあえずこれでなんぼかマシになるかな。少なくとも下りでジャンプできなくなることはなくなります。

あと、坂道の角度が切り替わるころとかジャンプした直後とかに、フワーツってなる人はたぶん傾きをきちんと更新できてないです。

傾きは斜め床に着地した時以外は0にしましょう。

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、輪郭線がくっさ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。これも解説はもう少し待ってください。

ひとまず 2D に合わせるのを優先していきましょう。

2D に合わせていこう

これをやっていくためにはさっき話した

```
SetupCamera_Ortho(1000);
```

についてもうちょっと理解する必要があります。これは今は仮に 1000 と書いてますが、適切ではありません。

たぶん、読んでも今は分からないとは思いますが、以下のリファレンスを読んでください

<https://msdn.microsoft.com/ja-jp/library/cc372881.aspx>

「正射影行列」というのを作っています。

はい出ました「行列」

まだ数学はそこまで行ってないんですけどね。いやホントはとっくにやっとなきゃイケないんですけど…ねえ。

まあクラスの半分くらいがピタゴラスもよー理解できん状況で半年で行列まで行くってやっぱり…ねえ。まあ後期はガツチガチにやりますよ。

行列ってのはものごつーい便利な数学的武器なんです。2D でも 3D でも強力な力を発揮するし、そもそも 3D ゲームは行列を理解してないと作れないと言っていいです。

DxLib の力を借りれば理解しなくても何とかかなりますけど、いろいろとやろうとするとやっぱり避けて通れません。

ここで軽く予習しておきましょう。

「行列」とは？

とりあえず簡単に言うと、行列とは数字もしくは文字式が縦と横に配置されている「値」もしくは「式」のことです。

簡単じゃないですか、そうですか。

実際に見てもらいましょう。

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

こういうのが『行列』です。縦と横に数値が並んでいますよね？これって実はベクトルの拡張ともいえるんですよね…いや、行列を1行に限定したものが『ベクトル』ともいえるのですが、とにかくこういうものです。

ここまではいいかな？

うん、まあ並べてるだけ。小学生みたいな素直な気持ちで見ればなんてことはない。

行列は通常は『乗算』することにより力を発揮する。この『乗算』がちょっとばかりややこしく、そしてトラブルの元になる。

行列の掛け算ってのは行列の行と列について区別しておく必要がある。

行はフツーに1行2行の『行』なので縦方向に増えていく。列はそれと直交する概念なので横方向に増えていく。さっきの例なら

	1列目	2列目
1行目	1	2
2行目	3	4

こういう感じです。

で、掛け算なんですけど、行と列をかけて足します。1行目と1列目をかけて足したものが乗算結果の1行1列目です。じゃあ1行2列目は1行目と2列目をかけて足したものになります。

例えば以下のような行列同士をかけるのならば

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{pmatrix} = \begin{pmatrix} 5+14 & 6+16 \\ 15+28 & 18+32 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

という結果になります。ルールさえ分かればかけて足すだけですから簡単ですね。

この注意点は乗算の左右を入れ替えると結果が変わっちゃうってことなのよね。

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 5*1+6*3 & 5*2+6*4 \\ 7*1+8*3 & 7*2+8*4 \end{pmatrix} = \begin{pmatrix} 5+18 & 10+24 \\ 7+24 & 14+32 \end{pmatrix} = \begin{pmatrix} 23 & 34 \\ 31 & 46 \end{pmatrix}$$

結果が変わるぞお!!気を付けろお!!

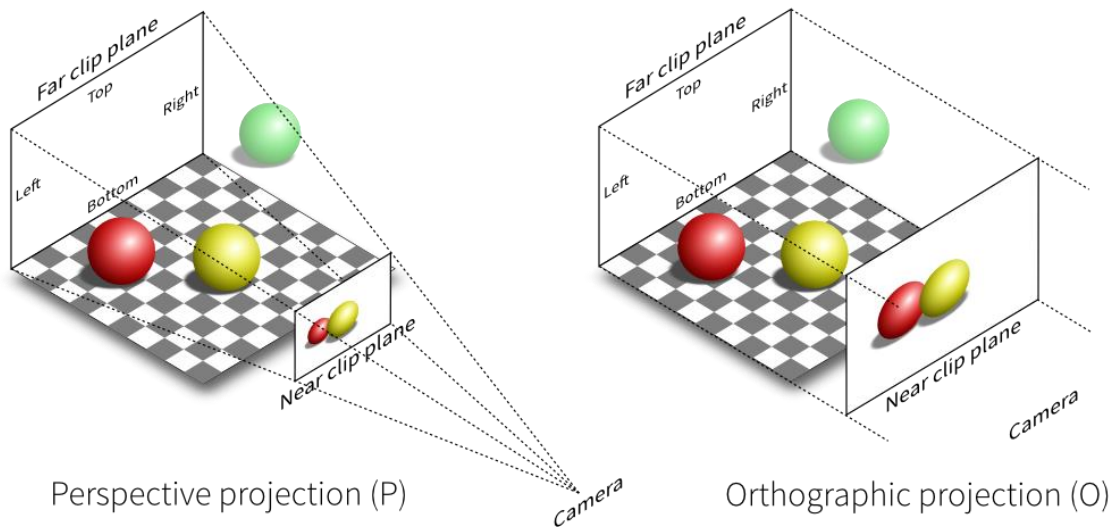
何となくわかりましたか？…まあもったいぶった割にはこの程度の話なんですよ。
ここでは 2×2 の行列を考えましたが、実際の3Dでは 4×4 の行列を使います。こいつの便利
なところは回転だの投影だのという変換が行列イッパツで完了します。

Orthogonal(平行投影)

さて、3DCG の話をちょっと書きますが、皆さんは遠近法を知ってますよね？遠くに行けば行くほど小さくなるアレです。通常のカメラはこれですよね？

ただこの平行投影はどこまで行っても同じ大きさのカメラです。

図で書くと



こういう事なんだぜ？で、3D プログラミングにおいてはパースかけるときもパースかからない時もどちらも『行列』で表すわけ。

どういう行列かというと

$$\begin{pmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{z_f - z_n} & 0 \\ 0 & 0 & \frac{z_n}{(z_n - z_f)} & 1 \end{pmatrix}$$

こんな行列です。あっ…うそうそ、気分悪くした？ね？どうせこういう難しいのはコンピュータがやってくれるからビビらないでね？

まあともかく必要なのは画面の幅(w)と画面の高さ(h)と z 方向の近い限界(zn)と z 方向の遠

い限界(zf)なわけ。

うん…わけわかんないでしょ？とりあえずここまでの話はもう少し先でまた話します。ここで大事な点は平行投影には画面の幅と高さが必要ってことなんですよね。

で、DxLib は初期化の時点でアスペクト比(縦横比)を最初に計算してるので高さが分かれば幅はわかるんです。

だから DxLib::SetupOrtho において分からないのは画面の『高さ』だけです。

なので、
SetupCamera_Ortho(画面の高さ);
となります。

さて、画面の高さは人によって違うと思いますので、自分の画面をどう設定しているのかを改めて確認してください。

僕は 576 ピクセルにしてるのでそれならば

```
DxLib::SetupCamera_Ortho(576);
```

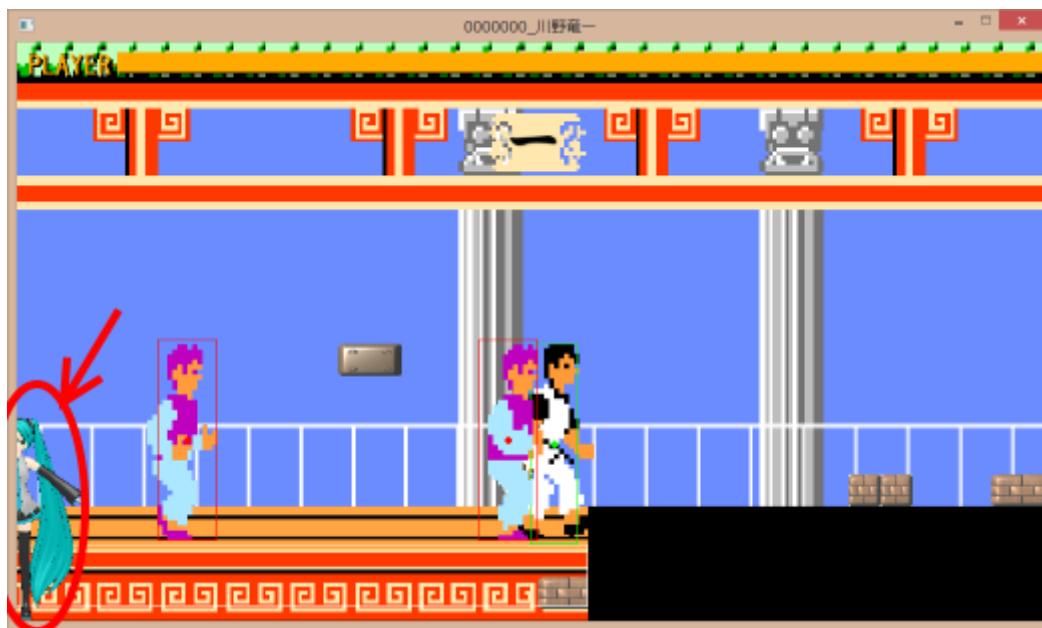
が最適なパラメータになります。
直値入れるのが嫌な人は

```
int w,h,depth;  
GetScreenState(&w,&h,&depth);  
DxLib::SetupCamera_Ortho(h);
```

でいいでしょう。

ここまでやると 2D として扱いやすい状況になります。

実行してみましょう。



こんな感じになります。

スケーリングが不適切だともう少し大きくなってしまいますので、キャラクターの大きさに合わせて調整してください。

ところで何が『扱いやすい』んでしょうか？

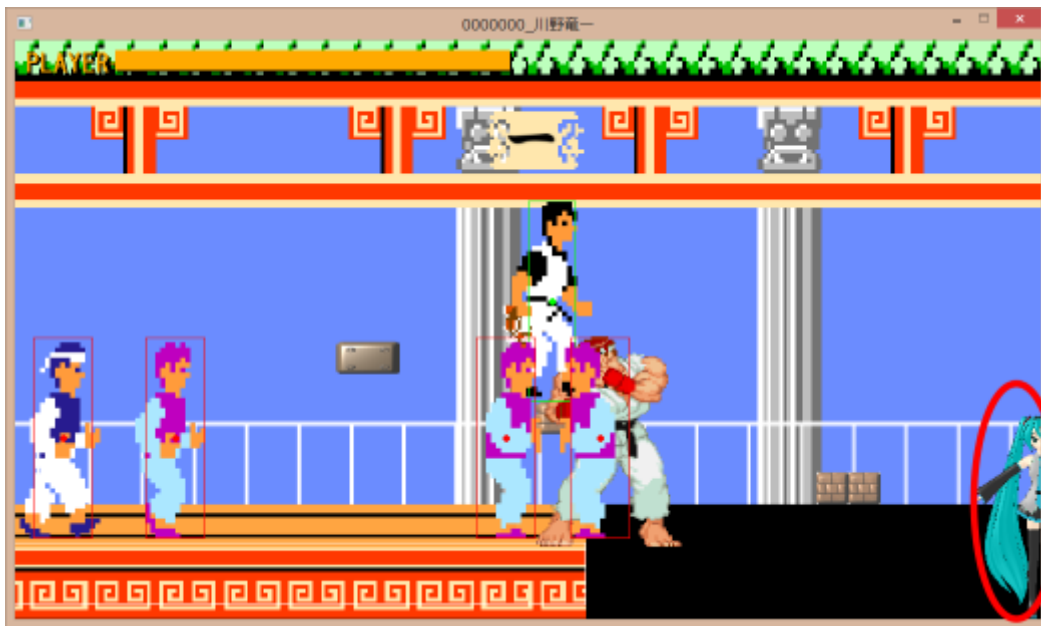
それはですね、ピクセルの単位が 2D の時と同じになるんですよ。試しに画面幅ぶん座標を右にずらしてみてください。

座標を移動するには MV1SetPosition 関数を使用します。

今の僕の画面は画面幅が 1024 なので
例えば表示の前に

```
MV1SetPosition(model,VGet(1024,0,0));
```

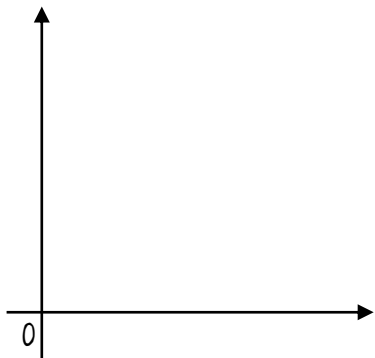
と指定すると画面幅分右に移動するので



こうなります。

ね？行けそうでしょ？(なんとかなりそうでしょ？)

で、この場合の座標系ですが



となっています。

アニメーションの基礎

アニメーションというか、ポーズみたいなもんなんだけどね。

MMD のアニメーションデータのことを VMD って言います。拡張子は*.vmd です。

で、名前付けのルールがあって『PMD 名称 000 連番.vmd』という名前で保存しておけば勝手にアニメーションデータも読み込んでくれます。

拡張子に関しては

- MMD のモデルデータは～.pmd です。
- MMD のモーションデータは～.vmd です。

だが DxLib でモーションデータを用意するときには注意点があります。

vmd の名前は『PMD000 連番.vmd』という名前にしてください。

例えば

『初音ミク.pmd』に対応する vmd は

『初音ミク 000.vmd』から始まります。

つまり

初音ミク 000.vmd

初音ミク 001.vmd

初音ミク 002.vmd

:

初音ミク 999.vmd

という形で必ず連番にしてください(番号抜けがないようにしてください)

この名前にしておくと DxLib が自動でアニメーションデータもロードしてくれます。

これができていれば、自動でロードはできてるんでアニメーションデータのロードについて気にする必要はありません。

あとは何をすればいいのか？

アニメーションのアタッチ

『アタッチ』って何？

どのアニメーションを再生したいのかの指定です。

番号で指定します。その番号は vmd の連番番号を指定します。後述する『アタッチ番号』とは違うことに注意しましょう。結構混同します。もし 000 番を再生したいなら 0 と書きます。

```
_anim=MV1AttachAnim(_model,0);
```

MV1AttachAnim という関数は、アタッチしたアタッチ番号を返します。これも連番で帰ってくる番号ではあるのですが、VMD の番号とは違います。単なるアタッチ済みのアニメーションの識別番号です。アタッチするたびに内部でアニメーションに必要なオブジェクトが生成されます。

```
アタッチアニメーション番号 = MV1AttachAnim(モデルハンドル,vmd 番号);
```

あくまでも『アタッチ』なので、まだ再生しません。まだアニメーションしませんよ？

何故かという、アニメーションフレームが 0 のままになっています。アニメーションである以上、時間を進ませてあげないとアニメーションしません。

アニメーション時間を進ませるには

MV1SetAttachAnimTime という関数を使います。

```
MV1SetAttachAnimTime(モデル,アタッチアニメーション番号,進んだ時間);
```

ですから

```
MV1SetAttachAnimTime(model, _anim, animTime);  
++animTime;
```

のようにすれば、アニメーションを動かすことができます。なお ++animTime しておりますが、この単位はよくわかりません。正直によくわからないのです。

DxLib のマニュアルには『再生時間』とありますが、そもそも単位が明記されていないんですよ。秒なのかフレームなのか…とりあえず数値を入れるのは分かるんだけど…。ひとまずフレームという扱いで考えてみましょう。サンプルによると 0.2f だったり 10f だったりと統一されていない。

とりあえず1くらいで進めておこう。そうしたらアニメーションするようになったらどう？

アニメーションの切り替え

アニメーションの切り替えをしたい場合、必ず前のアニメーションを殺してください。

つまり後始末してください。アニメーションした結果にアニメーションされるのでおかしくなります。

前のアニメーションを殺すには

MV1DetachAnim という関数を使用します。

- MV1DetachAnim(モデル, 前のアニメーション番号);

なのでパンチとか切り替えたいときは

```
if(key.state(KEY_INPUT_X)){
    _animTime=0;//フレームを0に戻す
    MV1DetachAnim(_model,_anim);//前のアニメーションを殺す
    _anim=MV1AttachAnim(_model,1);//新しいアニメーションにする。
}
```

こんな感じにします。

ループアニメーション

前のアニメーション(今のアニメーション)が終わったかどうかを判定し、終わってたら0からやり直せばいい。人生といっしょやな。

Q: 終わったかどうかどうやって判定するの？

A: 最初にアニメーションの長さを取得しておく。

モデルロードの時点で各アニメーションの長さを記録しておこう

`_duration = MV1GetAttachAnimTotalTime(モデル, アタッチ番号);`

で、アニメーションの時間がこれを超えたらまた0に戻せばいい

```
if(_animTime>=_duration){  
    _animTime=0;  
}
```

ちなみに物理が入っている場合は動きが時になんかしらアクションをとるんだけど、このせいでループするとき不自然になることがある。

DxLibはその点も考慮して対応策を入れてくれている。ありがたい!!!自分でやろうとすると大変なんだよね…

対応策は『vmd ファイルの命名の最後にLをつければループ扱いにして物理演算をいい感じにしてくれるよ!』だ。

つまり

- ニュートラル
- 歩き
- ジャンプ中
- しやがみ

のようなループ動作の場合にいちいちループ戻りの影響をつけないという事だ。これはすごい楽だ。

つまり

初音ミク 000.vmd がループアニメーションだったら

初音ミク 000L.vmd とすればいい。

詳しくは

http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R1N1

MV1LoadModel の項目の<ループ再生するモーションについて>が参考になるだろう。

アニメーションブレンド

『アニメーションの切り替え』を読んで疑問に思った人はいないだろうか?なんでわざわざデタッチなんかしなければならぬのか?と

それには理由があって、

『2D ならともかく、3D の時にはあるポーズからあるポーズへ、一瞬で遷移してしまうと非常に不自然に見えてしまう。』

2D の場合は例えば、歩き状態からパンチ状態に移行するのは一瞬でいいのだ。だって別の絵なんだから。

でも 3D の場合ってというのはそういうのは基本的には許されない。

人間の脳みそがそんな風に知覚しちゃうんだよね。ゲームってのはそういう事(人間の知覚に振り回される。逆にその知覚を利用して『気持ちいい』体験を与える)に直面することが多い。『感覚/知覚』や『感情/心理』をちょっとは勉強しておくのと役に立つだろう。そこは授業ではやらんけどね。いろんな本とか読んで家とかで勉強してね。個人的に質問してきたらわかる範囲で答えるけどね。授業ではやらんよ。

前述のようにあるモーションから別のモーションに一瞬で切り替わるなんて、人としてありえない動きです。自然に切り替わるには『モーションブレンド』が必要になります。

ゲームプログラミングにおいて、『ブレンド』って言葉がよく使われますが、

<http://eow.alc.co.jp/search?q=blend&ref=sa>

英語の意味は『混ぜること、混合』ですね。コーヒーのブレンドとかいうのはモカとキリマンジャロを何対何の比率で混ぜ合わせると旨いとかそういうやつですね。

モーションのブレンドも同じ。A っていうモーションと B っていうモーションを何対何で混ぜ合わせるわけ。

混ぜ合わせるときのルールは『足して 1 になるようにする』

つまり A が 0.5 なら B も 0.5 で、A が 1 なら B は 0、A が 0 なら B は 1、A が 0.3 なら B は 0.7 とかそういう感じ。

で、ブレンドブレンドモーションブレンド言うとりますけれども、実際には何と何を混ぜ合わせているのかというと

ボーンの状態をブレンドしているのである。

ボーンは分かるよね？

基本的には人間のモデルとかだと回転しかできません。で、ボーンの状態ってのは 4 次元行列

とかフォータニオンで表されるんだけどまあ4次元行列の話をする

$$B = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

こういうやつ。この行列の中に $\sin \theta$ だの $\cos \theta$ だのが入っているわけ。これがすべてのボーンについて情報が入っている。そう思ってください。ちなみにボーン行列だから B で表しています。それが全身に入っていると思ってください。

とりあえず体のひとつのボーンについて考えてみよう(例えば膝のボーンとか)
歩きの時のボーンを B_{walk} とし、パンチの時のボーンを B_{punch} として、移行中のボーンの状態を B_{current} とすると

$$B_{\text{current}} = (1 - t)B_{\text{walk}} + tB_{\text{punch}}$$

で、この t を $0 \sim 1$ まで動かしてあげると『歩き』→『パンチ』までがスムーズに移行できるわけです。こういう $(1-t)$ と t の合成ってのはゲームプログラムでよく使用するので覚えておこう。

じゃあ DxLib では実際にはどう実装するのかというと

MV1SetAttachAnimBlendRate(モデルハンドル, アタッチ番号, 割合(0~1))

http://dxlib.o.oo7.jp/function/dxfunc_3d.html#R4N6

という関数を使用します

予めアタッチ番号を

ニュートラルは `_neutral`

パンチは `_punch`

などとしてアタッチしておいて、

例えば毎フレーム通る場所に

```
MV1SetAttachAnimBlendRate(_model, _neutral, 1-t);
```

```
MV1SetAttachAnimBlendRate(_model, punch, t);
```

```
if(t<1.0f){
```

```
    t+=0.02f;
```

```
}
```

などと書けば、ニュートラルモーションからパンチモーションへスムーズに移行できるわけ

です。このとき、アニメーションフレームは一旦停止させておいたほうがいいでしょう。

ただし、『現在のモーション』ってのは状況によって変わるものです。

ですから上の `_neutral` とか `_punch` とかみたいに直接のアタッチ名を使うのではなく `_start`, `_end` みたいな変数に代入して

```
MV1SetAttachAnimBlendRate(_model, _start, 1-t);
MV1SetAttachAnimBlendRate(_model, _end, t);
if(t<1.0f){
    t+=0.02f;
}
```

としたほうがいいでしょう。

で、『何フレームで移行完了するか』なども指定出来たほうがいいので、まあここも状態遷移で管理できればいいかなと思います。

つまり

```
void (Model::_animUpdate)();
とかって宣言しておいて
```

```
void NormalAnimationUpdate();
void BlendingUpdate();
で切り替えるのがよいと思います。
```

Effekseer を組み込む

…そろそろついてこれなくなる人が出てきそうだが、頑張ってやっていきましょう。

Effekseer とはフリーの 3D エフェクトツールである。

<http://effekseer.github.io/jp/>

はっきり言ってカッコいい。

<http://effekseer.github.io/jp/contribute.html>

ここにあるやつはホントカッコいい。とりあえず全部見てみよう…な!!

さて、こいつは DXLib に組み込むことも可能なのである。

それなりにテクニカルではあるけどね。ひとまずダウンロードしよう。

準備

ダウンロードするものは

- Effekseer1.20 本体
- Effekseer1.20 For Runtime

です。

ちなみに Effekseer は Github でオープンソース開発されているので、興味がある人は見てみるといいですよ。かなり良い設計されてるので、プログラム設計に興味がある人は見てみましょう。人によってはツールそのもの以上に有用ではないでしょうか。

ダウンロードしたら解凍してください。

ひとまずは Effekseer 本体でちょっと遊んでみて、そのあとで本体で再生してみましよう。

本体はともかくランタイム側を解凍したら

解凍フォルダの Compiled の中に include と lib ってフォルダがあると思います。

自分のプロジェクトに Effekseer 作って include フォルダを放り込んでやってください。次に lib フォルダですが、これは使用 VisualStudio ごとに分かれていますので、自分が現在使用している VS に合わせてコピーしてください。

後で説明しますが Debug のほうを使うことになるでしょうね。

こういう外部のライブラリを使うのは慣れておいたほうが良いと思います。

プロジェクト設定

持って来たら DxLib でやったようにパスを通します。

追加の依存ライブラリに、Effekseer.lib と EffekseerRendererDX9.lib を入れてください。

うーん、久しぶりに扱ったらなんかいろいろと変わっててうまくいかなかったなー。

という事で注意点です。

1. DxLib_Init()の前に SetUseDirect3D9Ex(FALSE);を書いておく。
2. パスの指定が Unicode 文字列なのでエフェクトファイルの文字列リテラルの前に L をつける。
3. ひとつおりのエフェクトを出した後は RefreshDxLibDirect3DSetting();を呼び出さないと 3D 表示がふっとので、エフェクトを出した後に呼び出してください。

ちなみによくわからない人は RuntimeSample を動かして確認してほしいんですけど、DxLib のサンプル中にあるので、それを参考にしてください。

ただし中には DxLib 本体が入ってないので、動かすためには DxLib からライブラリを持ってきて、パスを通してください。

さらに DirectX 本体へのパスも通ってないので、パスを通します。

追加のインクルードディレクトリには

```
$(DXSDK_DIR)Include
```

を追加して

追加のライブラリディレクトリには

```
$(DXSDK_DIR)Lib\x86
```

を追加してください。

で、サンプルのコードに

Effekseer.Debug.lib と記述している部分がありますが、そんなものは存在しないので、その部分は Effekseer.lib に変更してください

あ、ちなみに自前で Effekseer とレンダラをコンパイルすると Effekseer.Debug.lib みたいな名前で出力されます。

なんかうまく表示されない…

サンプル通りにやったのになんかうまくいかねーって人はいろいろと間違ってるんだけど、気にしないでいい。

だって僕もやられましたから…ね。

とりあえず今の 131 教室では表示されていると思いますが家に帰ると表示されない。ソースコードはそのままんだけど、持って帰ると表示されなくなる。そんなことはありませんでしたか？

色々と原因は考えられますけど…。

- ロードに失敗している
- カメラとかプロジェクションの設定に失敗している
- ライブラリの不整合でやらかしている
- Effekseer 特有の問題

などがあります。

まずはロードに失敗しているはよく確認しろよ。

```
_effect=Effekseer::Effect::Create( _mgr, (const EFK_CHAR*)"パス" );
```

この戻り値が NULL であればロードに失敗しています。この場合はパスを間違えているのが原因です。よく確認してください。

次にカメラの設定なんですけど、当然ながらエフェクトの発生が成功しててもカメラが明後日の方向を向いていると見えなくなります。

カメラの方向を確認しましょう。

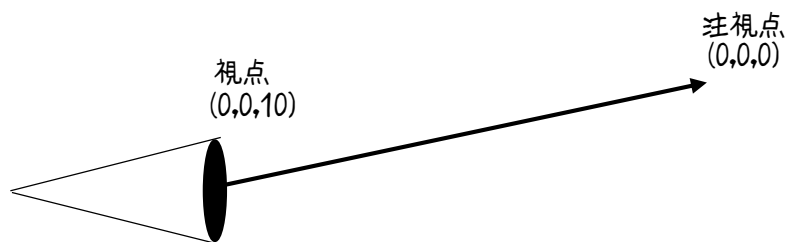
まずはエディタでエフェクトを確認してみて、どこから発生しているかを確認。例えば 0,0,0 から発生しているのならカメラが 0,0,0 を向いているかどうかを確認します。

例えば

```
_er->SetCameraMatrix(  
    ::Effekseer::Matrix44().LookAtRH(  
        ::Effekseer::Vector3D( 0.0f, 0.0f, 10.0f ),//視点  
        ::Effekseer::Vector3D( 0.0f, 0.0f, 0.0f ), //注視点  
        ::Effekseer::Vector3D( 0.0f, 1.0f, 0.0f ) //アップパー  
    )  
);
```

なんていう風に設定しているとします。

これはどういう状況かというと



こういう状況です。視点から注視点をみるベクトルを作るのが LookAt 関数です。最後に RH がついているのは RightHand(右手座標系)のことです。

ただ…通常 DirectX は LH…LeftHand(左手座標系)なんですよね…なんで DxLib のサンプルはこうなってるんでしょうね。もうちょっと調べさせてください。

ちなみに僕が DirectX でやる時は LeftHand にしています。

最後の『アップパー』はとりあえずこう書いておいてください。後期の 3D の話に入った時に改めてお話しします。

次にライブラリの不整合に関してですが、まあほとんど発生しないはずなんですけど、起こらないこともないので、怪しいと思ったらライブラリをリコンパイルしてください。それで治ることもあります。

最後の Effekseer 特有の問題に関してですが、これはライブラリ内部の話なので詳しく調べないとよくわからないのですが、サンプルを見ると…

```
// 何でもいいのでTransFragを有効にして画像を描画する。  
// こうして描画した後でないと、Effekseerは描画できない。  
DrawGraph( 0, 0, grHandle, TRUE );
```

とか書いてやがります。

なんだかよくわからないのですが、こういう仕様みたいです。131 教室でやっている分にはこの仕様を無視して描画されるみたいですが…。

ついでに言うと DxLib の 3D 表示をした後にエフェクト表示をしようとするこの問題で表示されないっぽいです。

つまり

2D 描画→エフェクト表示→3D 描画

の順に描画命令を出せば表示されることになります。

順番が影響するってのは面倒ですね。何故かの理由が分かれば解説するし、解消できそうなら解消してみますので、ちょっと待っておいってください。

コイツも 2D 表示に合わせてみる

一応こういう記事はあるんですけど、

<http://kage3.cocolog-nifty.com/blog/2015/06/effekseer2d-0c6.html>

```
DrawEffekseer2D_Begin();
```

```
DrawEffekseer2D_Draw(effectHandle);
```

```
DrawEffekseer2D_End();
```

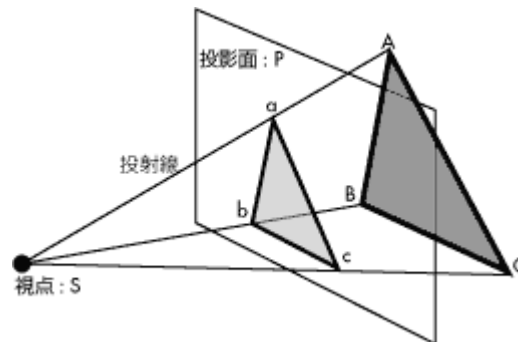
の流れは本線じゃないっぽいので、そういうんじゃなくて別の方法(正攻法)を試してみよう。

そもそも DxDLib のミクさんを 2D に合わせて表示したときのことを覚えていますか？

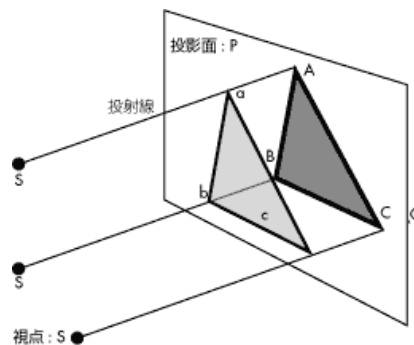
パースペクティブとオルソーって話をしたのを覚えていますか？

透視投影と、平行投影の話ね。

これがパースペクティブ(透視投影)



で、以下がオルソー(平行投影)



さて、もうそろそろ感覚がつかめてきたのではないのでしょうか？

で、じゃ Effekseer って Orthogonal にする関数とかってあるんですかねえ…(期待)
あるんですけどね。

::Effekseer::Matrix44().OrthographicRH(幅, 高さ, 近, 遠)

これがどうも平行投影行列を返す関数のようです。DxDLib の Ortho を使えばいいと思われるかもしれませんが、そもそも行列の型宣言が別物なのでそれはできません。

まあ

```

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];
    }
}

```

こうやってしまうのもアリですがここまでやるくらいならもう Ortho 関数を使っちゃったほうがいいんじゃないかなと思います。

こういう風に設定します。

```

_er->SetProjectionMatrix(
    ::Effekseer::Matrix44().OrthographicRH(1024,576,1,100)
);

```

プロジェクションマトリックスにパースペクティブの代わりにオルソーをセットします。なんかルー大柴みたいな説明になっちゃったけど言わんとすることは分かりますよね？

ただ、これは奥行きを出したい場合には向いていけませんので、その場合はちよいと工夫しなければなりませんが、それはもうちょっと面倒なので、後期にしたいと思います。

おまけ

モデルの反転

モデルの反転自体はスケーリングを反転したい方向に-1 をかければいいんですが、残念ながらこうなります。



マックロクロスケになってしまいます。これには二つくらい原因があるんですけど

- 頂点情報が反転してしまうためカリングが不適切になる
- 輪郭表示が悪さをして、全身真っ黒になってしまう

正直面倒なんだよね…

このうち、上の問題は

// モデル中のバックカリングを行うメッシュを、左回りバックカリングから右回りバックカリングに変更する

```
int meshNum = MV1GetMeshNum(_model);  
for (int i = 0; i < meshNum; i++){  
    // 背面カリングを元々行わない設定のメッシュは設定を変更しない  
    if (MV1GetMeshBackCulling(_model, i) != DX_CULLING_NONE){  
        // 背面カリングの設定をデフォルトの  
        //DX_CULLING_LEFT とは逆の DX_CULLING_RIGHT を設定  
        MV1SetMeshBackCulling(_model, i, DX_CULLING_RIGHT);  
    }  
}
```

てな感じで、それぞれのメッシュに対するカリングを指定できるんですけど、今度は下の『輪郭線』問題が出るんですよ…

輪郭線を消す方法を探したんですけど、ちょっと見つからないので、まず輪郭線の描画を自前シェータにする必要がありますが、そこまでの話は後期でやっていきたいと思います。

ちなみに完全に輪郭線を消してしまえば



このように反転してもまともに表示されます。

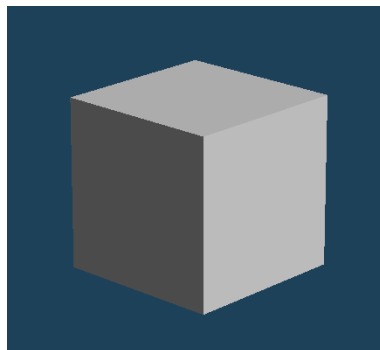
この状態に対してシェータで輪郭線をつけてやれば OK。

まあ『OK』って言ってもそれがそれなりに難しいんですけどね…。

輪郭線の仕組み

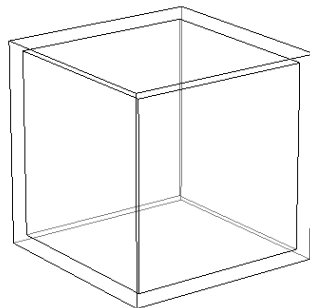
輪郭線の話が出たので軽く輪郭線を描画する仕組みについて話していきたいと思います。

一番簡単な方法はこうです。

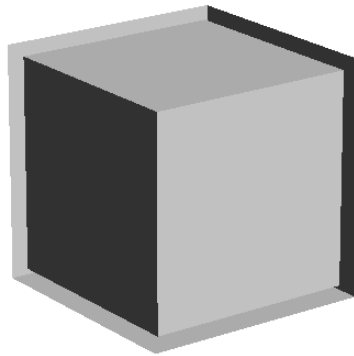


こういう立方体があって、輪郭線を作るとすると…

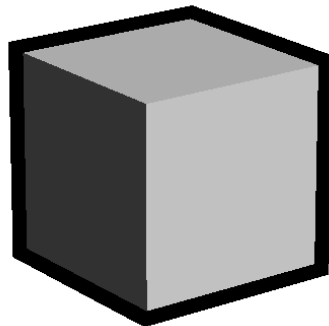
まず拡大コピーします。



そのあとで面を反転…



そして最後にその反転した面を輪郭線の色(黒)で塗りつぶすと



仕組みを見ちゃうと、輪郭線に見えないんですけど、何も知らない人が見たら『アッ!輪郭線だ』と思います。

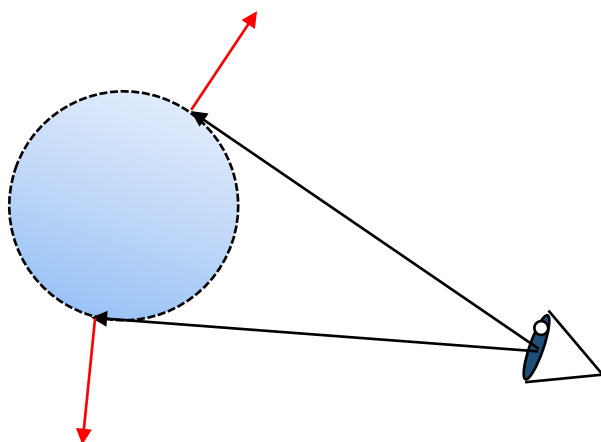
もっとも簡単な方法の場合はこんな感じで輪郭線を表現してます。DxLib は自動でやってくれますが、本当は(DirectX や OpenGL では)自分でやらなければならないんですよ。

んで、輪郭線作る方法はこれだけではないんです。いろいろあるんですけど…まあそれやろうとすると『シェータ』を使いこなさなければならないので、現段階ではそこまではできないかなって思います。

シェータを簡単に説明すると、ピクセルごとの情報を見てそれを元に色々な処理をすることが出来ます。

仕組みの一つを言っておくと『法線』を見ます。ピクセルごとに『法線ベクトル』を見て、この法線ベクトルと視線ベクトルとの間の角度が 90° 近辺の時に黒く塗りつぶせば勝手に輪郭線になるわけです(完全ではないよ?あくまでも簡単に概念を言うとね?こんな感じってことよ?)。

例えばこういう球体を見ているとする。



そうした場合、視点から見て球体の端っこのところで輪郭線を入れるべきなんだけど、そこにおける法線はどんなのかというと図のように直交状態(90°)なわけです。

で、後々お勉強する『内積』が強い味方になるんですけど、計算は

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$$

こんな感じに『かけて、足す』これだけです。計算上も大したコストはかかりません。
さらに内積には面白い性質があって

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

こういう性質があります。

直交するってことは $\cos 90^\circ = 0$ ですから、内積の結果が0になったら輪郭線を入れる。これで輪郭線実装できたりするわけです。

んで、実際にはこれをピクセルシェータにて計算することによって輪郭線を出します。
ちなみにこのやり方だと角ばってる図形の輪郭線が正確には表現できません。丸っこい物体にしか有効ではないです。

もう一つのやり方は…これちょっと難しいんですけど

あとはすべてのピクセルにおける『深度値』をあらかじめ計算しておいて、その値が急激に変わる部分で色を濃くしてやれば、それが輪郭線になるってわけです。

そういうやり方もあります(割とこれが主流だったりします)

ともかく知っておいてほしいのは、輪郭線を作るには色々な方法があって、割と数学を使う(特にベクトル)ってことを頭の中に入れておきましょう。

文字コードと文字セットについて

文字コード…それは軽視すると非常に痛い目に遭うものです。

とくに Web 系とかの開発をするとき、あとはチャットシステムとかを使うときにもこの手の問題が発生するかもしれません。

もっと言うと実際のゲーム開発では『ローカライズ』と言って、様々な言語への対応が求められるんですけど、アラビアとかアジアの文字コードの多さはもう滅茶苦茶なレベルで多いし、ルールも多岐に渡って頭が爆発します。

まずは皆さんが直面するであろうものが文字コード…ではなく『文字セット』について。

えっ？文字コードの話じゃねーの？と思った人は、そりゃそうだと思います。文字セットなんて聞いたこともないでしょうから…。

文字コードと文字セットの違い

ホントにこの辺は超ややこしいので、さらーっと概念だけ話して、変換の方法くらい話して終わりにしたいと思います。

文字エンコードとかそういう話…聞くでしょ？

あそこで問題になるのが文字コードで、その前に話しておかなければならないのが『文字セット』

何でかという、例えば皆さんが最初に C 言語で習うのはシングルバイト文字セットのことです。

つまり『文字』は 0~255 までで表現しようってことです。英語圏だけの話ならこれでもよかった。僕が少年のころはまだこの時代でした。だから友人の家で『パーソナルコンピュータ』を見たときは英語だらけだったし、日本語があったとしても日本語はすべてイッテ王、ゴルァなどの半角文字列でした。

もちろんこれは日本特有の話で英語が 41~7A ままで隙間が空いてたので、そこに日本語を入れてたんだよね。これがいわゆる半角文字ってやつね。

じゃあ、そんな時代の『ドラゴンクエスト』ってどうやってたの？という疑問があるかもしれない。

<https://ja.wikipedia.org/wiki/%E3%83%89%E3%83%A9%E3%82%B4%E3%83%B3%E3%82%AF%E3%82%A8%E3%82%B9%E3%83%88%E5%AE%B9.E9.87.8F.E5.89.8A.E6.B8.9B>

そうだよ。今みんなは DrawString とかで文字が表示されてるから文字リテラルがそのままゲーム本編に出ると思ってるんだよね？

違うんですよ。

ドラゴンクエストの総容量は 64KB で、文字に割けるビット数は限られていたんです。という所で、興味深い記述が…

初代ドラゴンクエストはひらがな 50 音と、数字と 10 と記号が 10 くらいなものです。

資料を眺むと

『具体的には『イ・カ・キ・コ・シ・ス・タ・ト・ヘ・ホ・マ・ミ・ム・メ・ラ・リ・ル・レ・ロ・ン』の 20 文字に濁点と長音(音引き)を加えたものでした。』

らしいです。魔法とかもこの文字の制限の中から考えられたとのこと。

さて、となると、パソコンで使用している文字をそのまま対応させるわけにはいかんのですよ。

つまり先ほどのカタカナをそのまま当てはめるならば

0	1	2	3	4	5	6	7	8	9
イ	カ	キ	コ	シ	ス	タ	ト	ヘ	ホ

みたいに対応しているわけ。

実際の ASCII 文字もそうなっているんですけどね。要は文字なんてもんは、コンピュータにとっては数値⇔画面に表示する文字の対応表でしかないわけ。

でその対応が文字コードなわけだけど、文字セットの概念はもう少し上位にあって、どれくらいのビット数の中にどんな文字集合が入っているのかって概念が文字セットなのね。

ホンマややこしいです。

『文字セット』はエンコード(つまり数値と実際に表示される文字対応)は気にしてない概念なわけね。

さて、ここで文字セットにはいくつか概念があるんだけど、

0~256 の1バイトで全文字を表現しようというのが『シングルバイト文字セット』これが僕の少年時代。

そこから時代は進んで、うん、日本語とかある場合、漢字もあるし1バイトとか無理やろ…となって出てきたのが2バイトで表現する文字セットつまり『ダブルバイト文字セット』なわけ。そしてシングルバイトとダブルバイトをごっちゃにしたのがマルチバイト文字セット。

ああ〜、ややこしいんじゃああ!!!

ちなみにマルチバイト文字セット(SJIS)

1バイト目	2バイト目	文字種別
20-7E		アルファベット
A1-DF		半角カナ
81-9F E0-EF	40-7E 80-FC	漢字

でも、僕が社会人になったころに主流になってきた(Windows95 くらいから)のがこのマルチバイト文字セットと SJIS 文字コードだったわけや。潔く Unicode みたいなのが出てればよかったんだが、まだまだ要領に余裕がそれほどなかった時代だったため、マルチバイトがそのまま主流になって現在に至っている。

ちなみにここがまたややこしいのだが、マルチバイト文字セットは SJIS だけではなく EUC 文字コードなんかもマルチバイト文字セットである。

EUC 文字コード

1バイト目	2バイト目	3バイト目	文字種別
20-7E			アルファベット
8E	A1-DF		半角カナ
A1-FE	A1-FE		漢字
8F	A1-FE	A1-FE	補助漢字(拡張漢字)

さて、なんだよ3バイト目って…もうやめろよ。こんなのが混在してたから Web はもう大変なことになってきた。

だけど、この時代まではビジネスシーンでしか PC が使われなかったんで、みんな何とか我慢して、これでよかったのだから、物書きや、専門職までが PC で記述するようになってきたからさあ大変。多言語対応やら、となってくるともうマルチバイト文字セットじゃ無理が出てきた。

そこで出てきたのが Unicode である。この Unicode も潔く 4 バイトにすりゃいいのに 2 バイト Unicode と 4 バイト Unicode なんてのがあったりして。まだまだ文字の戦いは長そうだな。ホントにやめていただきたい。4 バイトくらいで終わりにしてほしいなあ…。

そのせいで Unicode 文字セットの中には UTF-8 と UTF-16 があるというありさまなのよ。さらに UTF-8 のバイト範囲は 1~4 バイトである。えっ？可変なの？ってそうなのよね。可変なのよ。

<https://ja.wikipedia.org/wiki/UTF-8>

中を見てもわけわかんなくもていいです。まあ 7 バイトまでが書いてますが、実質使われているのは 1~4 バイトまでです。

ともかく

シングルバイト文字セット→マルチバイト文字セット→Unicode 文字セット

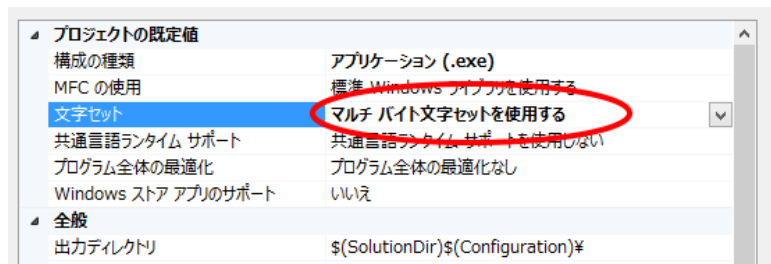
というわけで、これらは文字コードではなく『文字セット』。この違いはわかりましたね？そもそも文字を表す『体系』が違います。

そして『文字コード』ってのは文字を表す数値と、文字そのものの『対応表』だと思っておいてください。そして『エンコード』ってのは、対応表を選んで、この数値から『表示する文字を探して返す』ことだと思ってください。

ホンマにややこしいけどそういう事。さっきのドラクエの話は文字セットも文字コードもゲーム用に作ってたってことやね。

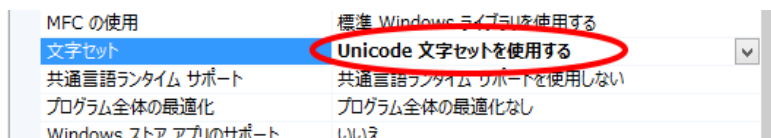
プログラミングにおける文字セット

これは既に皆さんは触れているんです。



ここね。

ここをこうすると



DrawString 等でエラーが発生するようになる。

```
:DrawString(12, 12, "PLAYER-1", ...);
:DrawString(10, 10, "PLAYER-1", ...);
:DrawBox(100+2, 10+2, ...);
:DrawBox(100, 10, 10, ...);
```

エラー: 型 "const char *" の引数は型 "const TCHAR *" の

見ての通りよね。

原因を見てみましょう。

const char* と const TCHAR* の互換性がなくて怒られます。

TCHAR の宣言部分を見に行きましょう。

```
typedef WCHAR TCHAR
```

と書いています。どうやら正体は WCHAR のようです。WCHAR を見に行きます。

```
typedef wchar_t WCHAR;    // wc,   16-bit UNICODE character
```

16ビットの unicode 文字の事みたいですね。wchar_t は組み込み文字なのでここで終わりです。つまり TCHAR は 16ビット文字というわけです。

ではマルチバイトの時は何故エラーが出なかったのでしょうか？一旦マルチバイト文字セットに戻したうえでもう一度 TCHAR の定義を見てください。

```
typedef char TCHAR
```

ん？

今確かに WCHAR だったよね？

ここにプリプロセッサマジックがあるんだけど、マルチバイトの定義の部分をもう少し離れてみてみよう。ちょっと長いけど我慢せえよ。

```
#ifdef UNICODE                                // r_winnt

#ifdef _TCHAR_DEFINED
typedef WCHAR TCHAR, *PTCHAR;
typedef WCHAR TBYTE , *PTBYTE ;
#define _TCHAR_DEFINED
#endif /* !_TCHAR_DEFINED */
(中略)
#else /* UNICODE */
#ifdef _TCHAR_DEFINED
typedef char TCHAR, *PTCHAR;
typedef unsigned char TBYTE , *PTBYTE ;
#define _TCHAR_DEFINED
#endif /* !_TCHAR_DEFINED */
(中略)
#endif /* UNICODE */                                // r_winnt
```

ええ…(困惑)

申し訳ないがこれは NG やで…

ではなくて、ポイントは先頭の **#ifdef UNICODE** やで？

プロジェクト設定で UNICODE を選ぶと UNICODE が define されたことになってて、マルチバイトだと UNICODE が define されてないことになってるわけ。

分からない人は『独習 C 言語』の『マクロ』を読んでおこう。

軽く説明すると

#ifdef ってのは『もしこれが#define されていたら』ってことで

#else は上の#ifdef に対応する奴。それ以下の部分は#define されてない時に見えるわけ。

で最後の#endif は if 文のカッコ終わりと同じです。

まあこれくらいは常識だよなあ…。

という事で、UNICODE をオンにすると char とは型が違うと言って怒られるわけです。

なんとなく理解できましたか？



ちなみに“**L**”は『この文字列リテラルは Unicode だよ〜』ってのを表す『マクロ』らしいんだ

けど、明記しているドキュメントがないんですよ…(・ω・)

<https://msdn.microsoft.com/ja-jp/library/baw8xdf2.aspx>

ここを読むと『マクロ』じゃないんじゃないかな？って気がしますが…まあマクロで対処している処理系もあるってことでしょうね。ともかく頭に L をつければ wchar_t の文字型として扱われます。

さて、wchar_t に関してですが、最後の t はなんやねんと思うかもしれませんがたぶんこれは typedef の名残の t だと思います。

昔の話だからね。組み込み型で wchar なんてなかった時代の時におそらく

```
typedef unsigned short wchar_t;
```

とでも定義していたんじゃないでしょうか。_t は『これは typedef だよ〜』ってのを表すのによく使います。

便利なマクロ

とりあえず便利なマクロがあるんで紹介しときます。L で Unicode 文字列であることを示せるんですが、ここで L つけちゃうと、逆にマルチバイトの環境に戻した時にまたエラーになっ

ちやうので、どちらでもオッケーなマクロを紹介します。

それは「**_T マクロ**」です。これは UNICODE マクロと連動していて

```
#ifdef UNICODE
#define _T(x) x
#else
#define _T(x) L ## x
#endif
```

みたいに定義されてます。

つまり `_T("文字列")` とやれば非 UNICODE の場合はそのまま“文字列”として扱われ、UNICODE の場合は `L"文字列"` として扱われます。便利ですね。

ちなみに `_TEXT マクロ` ってのもありますが、`_T` と同じです。

とりあえずこれからは相手が TCHAR の文字列を見たら `_T` を付けておきましょう。

マルチバイトとワイド文字(Unicode)の変換

今までのリテラルの変換の話をしてきましたが、例えばファイルから読み込んだのがマルチバイトの SJIS なのを Unicode の UTF8 に変換したいとします。

C 言語標準関数には `mbstowcs`

<https://msdn.microsoft.com/ja-jp/library/k1f9b8cy.aspx>

こういう関数があるんですけどね。

使い方は 2 回に分けて 使う必要があります。

- ① 最初の呼び出しで要求される文字列の長さ
- ② 2 回目の呼び出しで変換後の文字列が得られる

これを知らないとサンプルコードを見てもちんぷんかんぷんだと思います。

例えば

```
std::string str = "テスト";
```

という文字列があるとして。

```
int requiredSize=mbstowcs(nullptr,str.c_str(),0);
```

このrequiredSizeに変換後の文字列の長さが入っています。変換するまでは文字列長がわからないのです。

というわけでワイド対応の文字列型 `wstring` を宣言します。

```
std::wstring wstr;
```

```
wstr.resize(requiredSize);
```

これで必要な分の文字サイズを確保できてますので、メモリに書き込みます。

もう一度 `mbstowcs` を呼び出します。

```
mbstowcs(&wstr(0),str.c_str(),requiredSize);
```

というわけです。

この逆は `wcstombs` を使用します。

```
requiredSize=wcstombs(nullptr, wstr.c_str(), 0);
```

```
str.resize(requiredSize);
```

```
wcstombs(&str(0), wstr.c_str(), requiredSize);
```

これでよし。

関数化するならこんな感じ

```
wstring ConvertString(string& str){  
    wstring wstr;  
    int requiredSize = mbstowcs(nullptr, str.c_str(), 0);  
    wstr.resize(requiredSize);  
    mbstowcs(&wstr(0), str.c_str(), requiredSize);  
    return wstr;  
}
```

```
}
```

```
string ConvertString(wstring& wstr){  
    string str;  
    int requiredSize = wcstombs(nullptr, wstr.c_str(), 0);  
    str.resize(requiredSize);  
    wcstombs(&str[0], wstr.c_str(), requiredSize);  
    return str;  
}
```

で、これにはちょっとだけ面倒な部分があって、現在のロケール(国とかそういうやつ)に依存するっていう仕様なんですよ。

実行時にロケールを変更するのはちょっと嫌だよね。

というわけで Windows 標準関数にはその点を改善したのがあります。

<https://msdn.microsoft.com/ja-jp/library/cc448053.aspx>

MultiByteToWideChar です。

あ、ちなみに逆変換はこちら

<https://msdn.microsoft.com/ja-jp/library/cc448089.aspx>

WideCharToMultiByte

これも使い方はだいたい一緒です。最初にコードページを指定できる点が違います。

ちなみにコードページは

定数	意味
CP_ACP	ANSI コードページ(SJIS)
CP_MACCP	MAC コードページ
CP_OEMCP	OEM コードページ
CP_SYMBOL	シンボルコードページ(?)
CP_UTF7	CP_UTF7(使わんでしょ)
CP_UTF8	CP_UTF8(これは使うわなあ)

ちなみに OEM ってのは

「米国を含む多くの 1 バイトコードを常用している国では、歴史的な理由から Windows 環境下の文字セットとファイルシステム上で使用される文字セットが異なる場合があります。Windows 上の文字セットを ANSI 文字セットと呼び、動作環境に固有の文字セットを OEM 文字セットと呼ぶ場合があります。」

だとさ。まあ Windows 以降の環境においては気にしなくていいです。

シンボルとか UTF7 もたぶん使わない。という事で、実験してみましょう

例えば SJIS と UTF-8 の変換であれば

SJIS は CP_ACP です。UTF8 は CP_UTF8 です。

で、実験してなんとか成功っぽい結果は出たんですが…

ごめん。以下間違い!!(※以下のコードはテストコードで、一応変換の確認はできたけど、確信が持てないので保留。コード自体は参考程度に留めておいてくれ)

```
str = "テスト2";  
int requiredSize = MultiByteToWideChar(  
    CP_ACP,  
    0,  
    str.c_str(),  
    str.size(),  
    nullptr,  
    0);  
wstr.resize(requiredSize);  
MultiByteToWideChar(  
    CP_ACP,  
    0,  
    str.c_str(),  
    str.size(),  
    &wstr[0],  
    requiredSize);  
requiredSize = WideCharToMultiByte(  
    CP_ACP,  
    0,  
    wstr.c_str(),  
    wstr.size(),  
    nullptr,  
    0, nullptr, nullptr);  
str = "";  
str.resize(requiredSize);  
WideCharToMultiByte(  
    CP_ACP,  
    0,  
    wstr.c_str(),  
    wstr.size(),
```

```
&str[0],  
requiredSize, nullptr, nullptr);
```

…う～ん。関数の仕様の理解はしてるんだと思うし、一応正しく変換できてるんだけど…
なんだろうこの気持ち悪さ。

ちなみに二番目の引数を 0 にしているのはそっちのほう呼び出しが高速になるとか MSDN
に書いてたからそうしてます。いろいろとパラメータ変えてみましたが変化なし。

もう一度関数を読んでみよう……よくわからない。

あ、ちなみに皆さんが Web で検索すると `requiredSize+1` って書いてるサンプルコードがほと
んどだと思います。

それは '¥0' 用なんですけど、string 型の場合は別に最後に '¥0' なんてついてないの、+1 の必
要はありません。逆にあるとおかしくなるので注意。char や wchar_t の配列を使うなら +1 し
ないとだめだけどな。

情報を鵜呑みにすんなってというのはこういう事よね。自分で考えよう。

あ、ちなみにワイド文字の ¥0 はどういうものなのかというと L'¥0' です。

さて、長々と文字セットの話をしたがこれくらいで終わりにしよう。あ…UTF-8 は『8-bit UCS
Transformation Format』の略なんだけど、だからと言って文字を 8bit で表現するって意味では
ないからね。僕は昔そう思ってハマっちゃったことがあるので注意してください。

最後の最後に…TCHAR 型 string 作るで一。

TCHAR 型 string

`std::string` はマルチバイトの文字列型です。

`std::wstring` は Unicode の文字列型です。

ところで TCHAR 型(unicode を define してるかどうかで切り替えられる)の string はない
の？

というご要望もあるかと思います。

うん(・ω・)ないっばいんだ。すまない。

でも『なけりゃ作ればいいじゃない!!!!』がプログラマ精神です。

作り方は簡単。とっても簡単。

でも『なければ作ればいいじゃない』と思わないと見つける事すらできないであろう。プログ

ラマ精神は大事です。

まず、string と wstring の定義を見てみましょう。

- `typedef basic_string<char, char_traits<char>, allocator<char> > string;`
- `typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> > wstring;`

なん…だと？

これは…

…たまげたなあ。

うん。頭のいい皆さんならわかったでしょう？アホみたいに簡単だという事が。

つまりこう

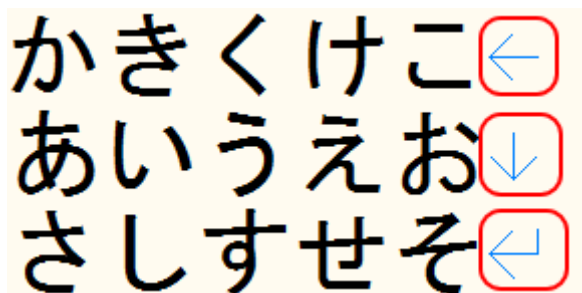
```
typedef basic_string<TCHAR, char_traits<TCHAR>, allocator<TCHAR> > tstring;
```

すればいい事がわかります。これで TCHAR 用の string である tstring ができました。
あとはこれを string や wstring と同じように使用すればヨイノデス。

その他文字に関すること

改行コードについて

え？何？今更？って思うかもしれませんが、はまるかもしれない部分を言っておきます。
ちょっとサクラエディタでとある処理をすると3種類の改行コードにお目にかかれます。



か	き	く	け	こ
あ	い	う	え	お
さ	し	す	せ	そ

3つとも改行コードです。

さて、違いはなんなんでしょうね……。まあね。通常は改行コードなんて目に見えないもんね。でもこれ混在していると VisualStudio から『統一しろ』って怒られるんだよね。もったいぶっても仕方ないので言っておくとこれは

CR(キャリッジリターン)→'¥r' 復帰

LF(ラインフィード)→'¥n' 狭義の改行

これらの組み合わせの違いやねんで

なんでこんなややこしい話になっとなるかという時代はタイプライターの時代にさかのぼるねん。



たまげたなあ…。

ラインフィードってのは、紙を1行送る(ずらす)って意味で、キャリッジリターンってのは打ち込んで進んだ列を元の1列目まで戻すって意味です。

で、これをコンピュータのテキストに対応させるときにいくつか派閥があったわけね。

- タイプライター同様に『CR+LF』で表現すべきだろう派(Windows デフォルト)
- ラインフィードだけで十分じゃね派(Linux, MacOSX とか)
- キャリッジリターンだけで十分じゃね派(Apple II とか)

こういう風なのが現在も続いている。

元が昔のことなので、今バカバカしくてもあまり馬鹿にしたり文句言わないように。

歴史的なのが原因で変なルールが残ってるってのは多いよ。

と、まあ色々あるわけ。実質 CR だけ派は駆逐されてるので考えなくてもいいけど CR と CR+LF が分かれてるのは厄介。

ネットワーク通信とか Web とかテキストエディタとかでトラブったりする。だから Web ページってのは改行コードではなく
で改行するルールにしているんだ。

みんなが直面するであろう問題は sakura エディタなどで『正規表現』を使って置換とかする場合、改行コードが CR+LF つまり“`\r\n`”でできていることを知らないで痛い目を見るからだ。

また、`getchar` とかで一文字ずつテキストから読み込む場合も改行の部分が 2 キャラ(CRLF)になっている場合は注意したい。

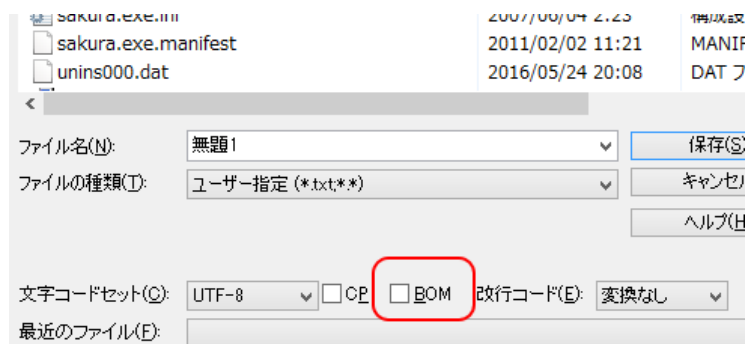
文字おまけ

BOM(Byte Order Mark)について

最後の最後にこれももしかしたら将来ハマるかもしれないのでヒトコトだけ言っておくと、UTF 系列にはファイルの先頭に『BOM』という厄介なものがくっついている事がある。

<https://ja.wikipedia.org/wiki/%E3%83%90%E3%82%A4%E3%83%88%E3%82%AA%E3%83%BC%E3%83%80%E3%83%BC%E3%83%9E%E3%83%BC%E3%82%AF>

こいつはバイトオーダーマークと言って、Unicode とエンコーディングを判別するための『見えない文字(数値)』である。これがめんどくさいのは『ついているとき』と『ないとき』があるためだ。



サクラエディタでもつけたり外したりの指定ができる。自分でルール付けしてるファイルならともかく、他人に作らせたテキストファイルにこれが含まれているかどうかで処理が変わ

ってくる。

面倒。面倒だけど UTF-8 の BOM は

0xEF 0xBB 0xBF

という固定値なので、最初の3バイトを見てこれがあるかないかで判断すればいい。

これでいよいよ文字列の話は終わり...だと思います。寧ろ俺が終わってほしい。文字の話はつまないし。

DxLib を使用したさらに本格的な 3D ゲーム

とりあえず参考にすべきはこれかな

<http://dxlib.o.oo7.jp/SwordBout/>

最近はこの付録が付録 CD とかじゃなくて Web サイトから落とせるからありがたいね。

SwordBout っていうゲームが DxLib の HP から落とせるようになっています。それなりに大きなファイルですけど興味がある人は落としておいたほうがいいでしょう。

なかなかしっかりしたゲームです。Unity 使ってもここまで作れる学生はどれくらいいるんでしょうかねえ…



まあそんな事は置いておいて、このゲームの目的はゲームをプレイすることではなくソースコードの中身をのぞくことですね。ちなみに DxLib 教育用ゲームなのでほとんどが C 言語ソースで書かれています(クラスとか全然使用されていません)。

本当に『昔の』作り方ですね。

『タスクシステム』とかも相当懐かしい作りになっています。タスクシステムは『知ってないと何やってるか分からない』のでタスクシステムを知らない人はとりあえず読み飛ばしましょう。

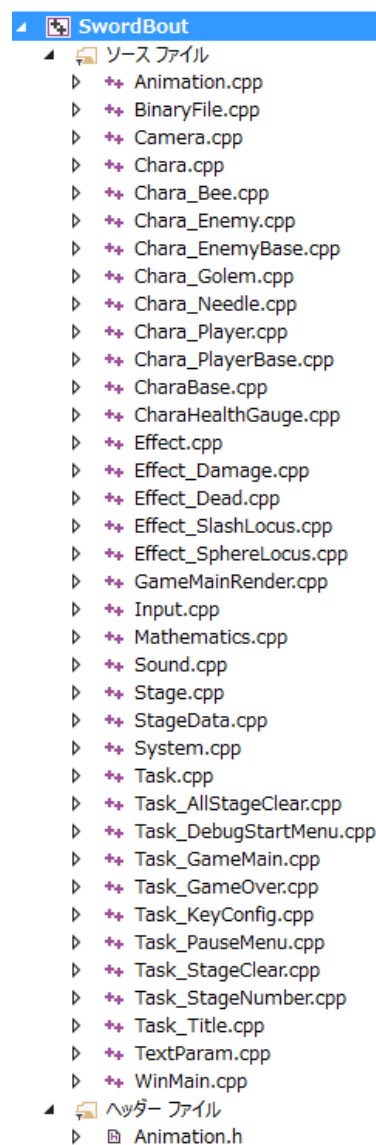
古い書き方ですが『C 言語』を再勉強したい人にはちょうどいいソースコードとなっています。

大きく分けて

- ゲーム本体
- ステージエディタ
- アニメーションエディタ

という構成みたいです。

ゲーム本体を見てみましょうか。



CPP ファイルだけを見てますが、思ったよりかはファイルが少ないですね。

行数的には Chara.cpp が一番大きくて 4000 行くらい。平均的には 1 ファイル 300 行くらいです。Chara.cpp だけがずば抜けて大きいです。

教育用ソースコードなだけあって、コメントも充実しています。

が、すべてのソースコードを見る時間はないし、そんなことでも自分で作ったのではない限りどうせ忘れます。

教科書読んで丸ごと覚えてる人っている？そんなにいないと思います。

というわけで、こういう他人のソースコードを読むときはポイントを絞って読みます。

とりあえず 2D の基本的なところはやっているんで 3D 特有の部分に絞ります。

(※自信がない人は CharaHealthGauge.cpp あたりを読んで自信をつけましょう。これくらいなら君たちならわかるはずです。)

カメラを試してみる

まずは Camera.cpp を見てみましょう。

カメラそのものの処理は大したことないんですけど、『カメラ揺らし』の処理が結構ソースコード量を食べていますね。

TPS 状態なのでカメラは主人公のちょっと後ろにある状態です。このテクニックを見てみましょう。

まずは 121 行目のコメントを見てみましょう。『プレイヤーが注視点となる』と書いてますよね？TPS 作ったことがない人はこれもたぶん思いつかないでしょう。

あ、『注視点』の言葉の意味…分かりますかね？3D ゲームプログラミングする場合はこの言葉はやたらと出てくるので覚えておきましょう。



視点から注視点を見るってのこういう事で、これは何のためにあるのかというと最終的には『視線ベクトル』を算出するためです。

213 行目を見れば分かるように『視線ベクトル』を算出しています。

あたり判定を試みる

DxLib には 3D あたり判定系の関数がそれなりに充実しています。

3D あたり判定は MV1Coll~ で始まる系の関数です。

その他おまけ情報

ドット絵素材について

某ゲームで使用されているドット素材が欲しい場合は Google 先生に

「ゲーム名(日本語)」『ドット絵』

などと検索しても出てきません。何故ならドット絵の抜き出しは日本では一応著作権法違反ですから。

ところがこれが日本を超えてしまうと割とうひょーです。検索ワードは

「ゲーム名(英語名)」『sprite』

と検索します。英語ではドット絵のことを sprite art というので、sprite でいいです。くれぐれも dot などと検索しないように。英語で dot ってのは内積のことです。

ちなみにゲーム名の英語名なんてわからないよ〜って思う人もいます。

例えば『魔界村』などというゲームは makaimura ではだめです。

こういう時はウィキペディア大先生を使います。

まず魔界村の Wikipedia を見ます。

<https://ja.wikipedia.org/wiki/%E9%AD%94%E7%95%8C%E6%9D%91>

魔界村の場合は丁寧に英名：Ghost'n Goblins と書いてますが、これを書いてない場合は左下の『多言語版』を見ます。ここで English をクリックすると

https://en.wikipedia.org/wiki/Ghosts_%27n_Goblins

このタイトルを入力すればいいです。

もしくは…うーん、これは明らかに法に抵触してるサイトですが

<http://www.sprisers-resource.com/>

とか

<http://spritedatabase.net/>

とかを利用するといいいと思います。

あと結構困るのは背景データだと思いますが、

「background」『sprite』で結構出てきます。

まあ…授業で使う分にはいいと思います。使用は自己責任でお願いします。

最終課題

最後の試練です。

8/5の18:00までに今まで学んだことを応用してC++で作ったゲームを提出してください。シューティングでもアクションでも、その他なんでも構いませんが君の技術力を見せてください。

提出場所は

[¥¥132sv¥gakuseigame¥game¥ゲーム PG131¥前期最終課題¥](#)

フォルダの中に入れてください。他人にできないものを入れると評価が高いです。いつものように自分の『学籍番号_氏名』のフォルダを作って、その中に実行可能状態のファイルその他を入れておいてください。

なお、評価のポイントとしては

今まで勉強したこと、もしくは今まで勉強したことが応用されているのが高ポイントのコツです。

2年生の目標

- 状態遷移ができています
- 衝突判定とそのリアクションがきちんとできています(アクションの場合)
- 地形とのあたり判定ができています(アクションの場合)
- 敵の動きが状態遷移に基づいており、生きてるように動く(アクションの場合)
- 画面遷移がきちんとできています
- 画面遷移の切り替え時になんかしら演出(タイトルが下からせりあがってくる、やフェードを利用している)
- なんかしらの3Dオブジェクトを表示できています

3年生の目標

- 数学の授業等で学んだ知識を応用している事
- 斜めの衝突判定に対応している事(アクション)
- 動く床に対応している事(アクション)
- 可能な人は DirectX や OpenGL を直でいじっている事
- その他チャレンジングな技術を使用している事(その場合 ReadMe.txt あたりに書いておいてください)
- シェーダを使っている

作品はかけた時間がそのまま評価に跳ね返ってくると思ってください。才能とかそういうも

んが重要だと思ってるでしょ？

自分が投入した時間がそのまま作ったモノに還元されるだけです。
プログラミングなんて、ほとんどの人が学校に入って初めて学んでるんだ。そこからの時間の差に過ぎないですね。

たまに例外もいるけど、君らが『すげえ』って思ってるやつって、だいたい寝る間を惜しんで(文字通り徹夜して)作ってるんだぜ？

体調が超悪くてもプログラミングしてるんだぜ？バイトを17時から22時までやってから午前3時まで毎日プログラミングやってる奴もいるんだぜ？

違いはなんだ…？



根性だろう。『頑張ってもできなかったらどうしよう』なんて考えてる暇は君たちにはあらへんねん。頑張るしかないのだ。