2年生の数学(数学応用)2020

2年生で学ぶ数学は、基本的にゲームプログラミングに数学を役立てるってな授業です。
前期は一般的な 2D ゲームで使う感じの数学をやります。3D も軽くは触れると思いますが、まぁどうせ後期もあるしええんちゃうかな(後期はレイトレーシングです)

うん、でも一部の人はもうさっそく 3D やると思うので、もしそれについて知りたい人は尋ねてください。

2D ゲームで必要な中高の数学は大雑把に言うと「ベクトル」以上!終わり!閉廷!とは行かない。

- ピタゴラスの定理(当たり判定かな。実は便利な関数 hypot があるんだよなぁ…)
- ラヂアン(弧度法やな。度数法と弧度法の区別はつけましょ。)
- 三角関数(どっちが cos でどっちが sin か間違えなければ OK。あと atan2 は神)
- 剰余(プログラムでいうところの%っすね。割った余りが重要になることが多いですね)
- **ベクトル(これだけはクッソ大事なのでしっかりやろう★★★★★)**
  - ベクトルの定義(x,y とか x,y,z をまとめただけだよ)
  - 加減算(x どうしを足す引く、y どうしを足す引く。それだけだけど?)
  - ベクトルの大きさ(ピタゴラスがわかれば簡単だな)
  - 内積(なぜか此奴が角度や射影長と深いかかわりを持つんだ。重要★)
  - 外積(これは 3D かな…?まぁ 2D でも内外判定に使うし…多少はね!)
- 行列(また…高校のカリキュラムに復活するらしいっすよ。機械学習でも重要)
- 物理的なやつ
  - 加速度と重力(ジャンプとか)
  - 斜面(張力も)
  - 運動量保存の法則

あとゲームならではのやつが
- 線形補間(まぁ、A 地点と B 地点の中間地点がどこかってやつね)
- ベジェ曲線(曲がるレーザーとかに使える)

これくらいかなあ…本当は微分積分とか確率統計とかやりたいんだけど…時間的に難しーんじゃねーかなー。まぁ別に確率統計やらんでもガチャの確率とかわかるじゃろ?時間余ったらやろうかなぁ…。だいたいこんな感じのを使って、プログラム組んでいくのがこの授業です。最初だし軽くテスト問題で…おさらいしようか。

確認小テスト中…

おわりましたか？

内容

実践編

こっからは実践編です。実際のゲーム的なとこで役に立つものをやっていきます。
数学の知識と経験だけではプログラムに活かせませんのでね。授業外でもしっかり C++を勉強しておいてください。
僕はしれっと C++で書いちゃいますが、もしかしたら一部の人にはコードの意味が分からないかもしれません。そういう事にならないように C++はきちんと理解しておきましょう。

環境設定

学校の環境ならともかくも、現在のような状況(リモート授業)になっているため、ここに環境設定の話も書いておくことにする。

まずは環境設定をしていきたいと思います。3 年生はすでに分かっていると思うのですが、2 年生にとってはこれからお話しすることは初耳かもしれません。

「DxLib のフォルダは C:¥DxLib や D:¥DxLib とは限りません」

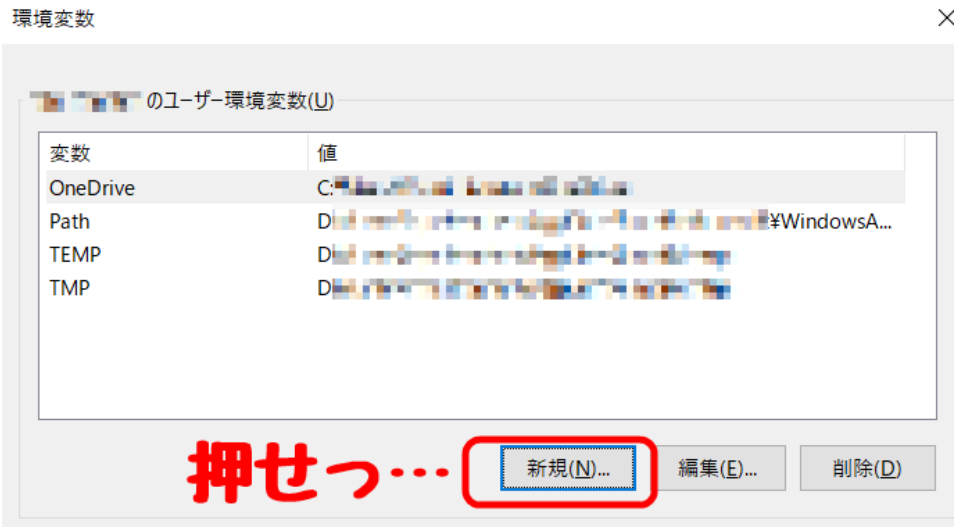
というか自宅で作業したことのある人ならわかると思いますが、DxLib のライブラリも自分でダウロードすんだよ当然だろ。というのが2年生以降の話です。一応 PC をセットアップした時点で僕が学校内の全 PC の C か D の直下に置きましたが、それができない環境もあったりしますので、
「DxLib をどのフォルダに置いていても環境設定さえすればどの PC でも同じように使える」ということをお教えします。めんどうですよね。
なぜこういうのが必要かというと、就職活動などをする際に、相手方は C や D の直下に DxLib のフォルダなんてまず置きません(もっと言うとダウンロードもしないと思います)。
あと、新2年生にとってはライブラリのリンク経験は DxLib くらいしかないのかもしれませんが、今後のプログラミングをやって行くと分かると思いますが他にもいくらでもあります。

というわけで、今後はライブラリのフォルダを直で書くのはやめましょう。1年生の頃はね、覚えることが多かったからこういう細かいことは教わらなかったと思います。頭がフットーしちゃいますからね。

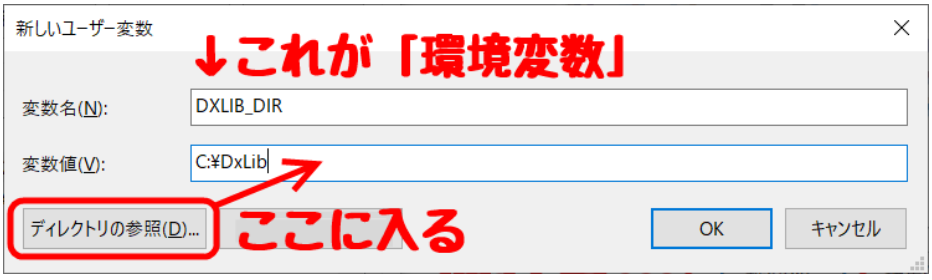
① まず自分の PC の DxLib のフォルダの場所を確認します。
② 次に Windows 左下の検索バーに「環境変数」と日本語で書き込みます

③ そうすると2つくらい候補が出てきますが「システムではないほう」を選択します。

④ そうすると「環境変数」のウィンドウが立ち上がりますので、上下に分かれている画面の上側で「新規」というボタンを押します。押せよ!絶対押せよ!

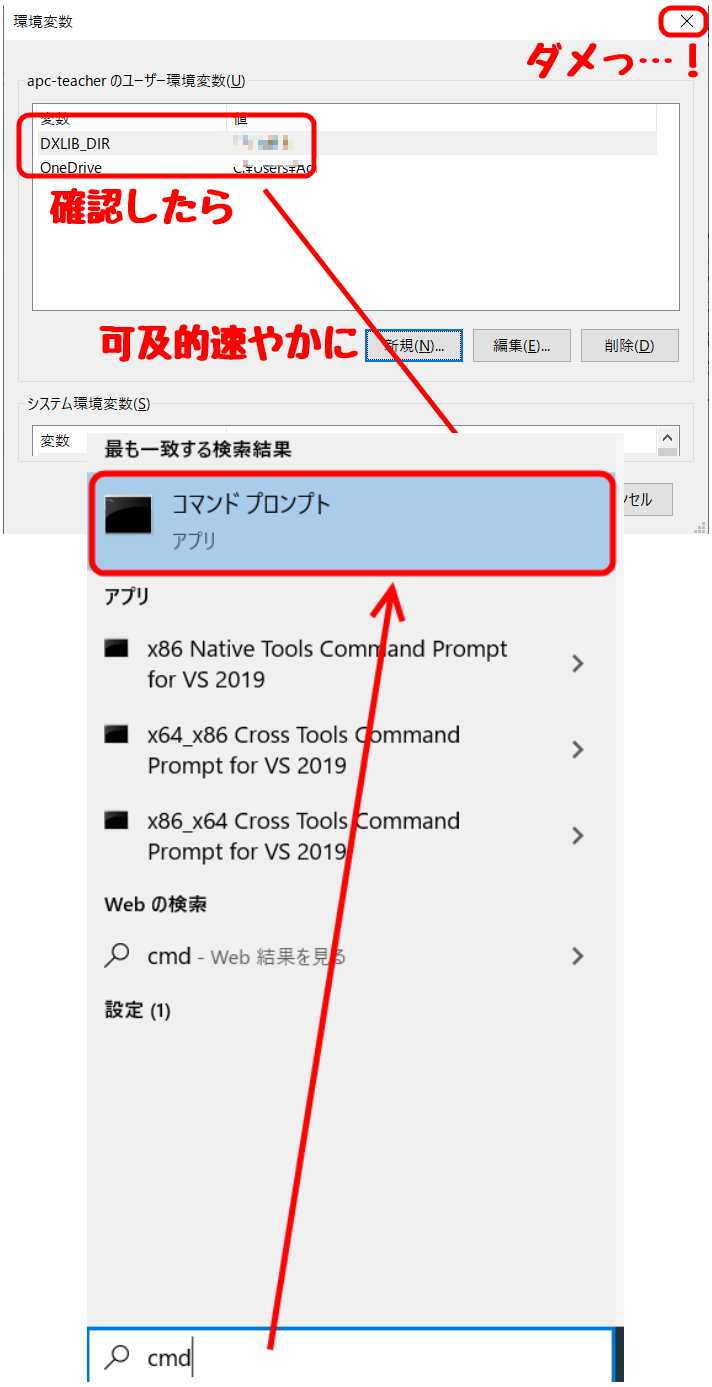


⑤ そうすると新規で「環境変数」を作るためのウィンドウが出てきます。今回の環境変数はディレクトリのためのものなので「ディレクトリ」ボタンを押して、DxLib が置いてあるディレクトリを選択してください。
また、上の段はそのディレクトリを表すマクロ変数みたいな感じになるので、そこに DXLIB_DIR と書いておく。終わったら OK を押す(☆右上の×を押すなよ!絶対押すなよ!)



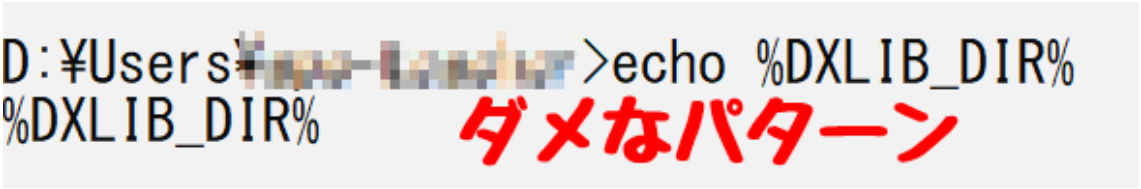
⑥ OK 押すと元のウィンドウに戻る。この時点で環境変数が出来上がっていると思うだろう？素人はまんまと引っかかる。
ところが大間違いなんだ。よくやるやつが右上の×を押しちゃうやつ。違う…っ!!
ウィンドウを閉じるときに反射的に×を押したくなる気持ちはわかる…っ!!しかし…それが罠っ…!!!巧妙に仕掛けられた悪魔的…罠っ!!今までの作業が台無しになってしまう。必ず OK を押すようにしよう。

⑦ さて、これで話は終わりではない。コマンドプロンプトを立ち上げてくれ。何ィ？コマンド



プロンプトを知らないだぁ!?お前ここをカルチャーセンターと間違えてんじゃねえのかァ?
コマンドプロンプトというやつは、システムコマンドを打ち込むためのものです。Linuxとか使ったことのある人ならLinuxコマンドは知っていると思うけど、lsだのcpだのcdだのというコマンドでOSに命令を出したり情報を引き出したりするためのものだ。
出し方はいたって簡単。ご注目!いきますよ～よく見といてくださ～い。
先ほど環境設定の時にやったように左下の検索バーに"cmd"と3文字打ち込んでください。
いいですかー、しー、えむ、でぃー、ですよ～。
はいこれで、立ち上がります。見覚えのある人もいるかと思います。デフォルトが黒背景の白文字なので、説明では見づらさ軽減のために設定で白背景の黒文字で表記しますね。

⑧ コマンドプロンプト上で"echo %DXLIB_DIR%と打ってエンター。まともに設定できていれば、完全なパスが表示される。表示されなければ設定に失敗してんだよもっかいやんだよあくしろよ。



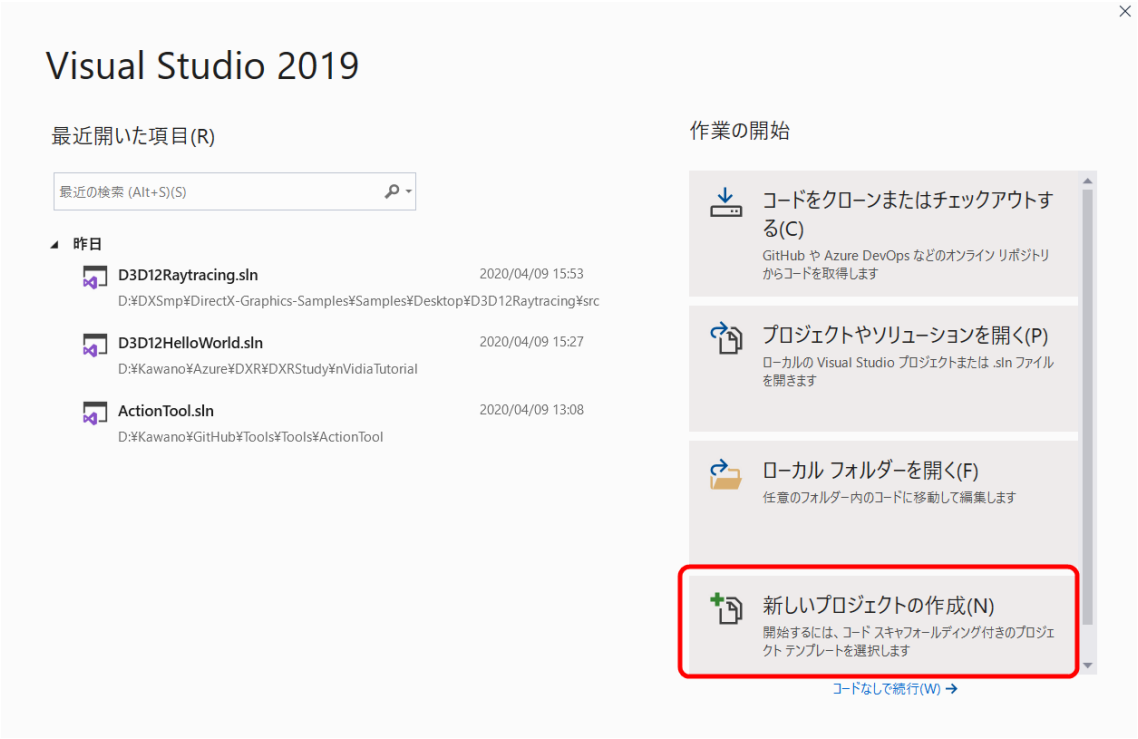
うまくいっていればフルパスが表示されます。なお、やり直す際はいったん全てのコマンドプロンプトを落としてもう一度立ち上げなおしてください。OKなら

このようにフルパスになります。これで設定できていることが確認できました。

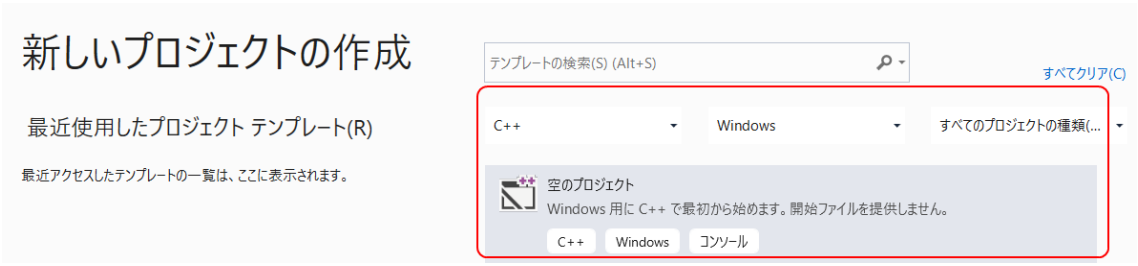
⑨ 次にもしVisualStudioを立ち上げているのであればすべて終了してください。一つでも立ち上がっている状態だと、この環境変数の変更の影響を受けないからです。

⑩ そしてVisualStudioを再び立ち上げて

新しいプロジェクトを作成します。



ここで作るのは「空のプロジェクト」(C++ Windows Console)です。



ソースコードとして main.cpp を追加してください。中身はこんな感じ

```
#include<DxLib.h>


int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int) {
	DxLib::ChangeWindowMode(true);
	if (DxLib_Init()) {
		return -1;
	}
	DxLib::SetDrawScreen(DX_SCREEN_BACK);
	while (DxLib::ProcessMessage() == 0) {
		DxLib::ScreenFlip();
	}
	return 0;
```

}

スケルトンプログラム

流石に DxLib のスケルトンプログラムを作れない人がこの中にいるとは思いたくない。いや 4～5月なんか作ってたはずなので、ちゃっちゃとプロジェクト作って、DxLib のウィンドウが出る状態にしてください。

これは数学の授業なので、プロジェクトの作り方なんて面倒見てられませんので、自分で作ってください。まともに１年間プログラムやってきた人間なら 10 分もあればできるはずです。

…ま、まぁ、そうは言ってもアンタみたいなクソザコナメクジじゃ何百時間かかるかわからないわね!!

https://github.com/RYUICHIKAWANO/MathLesson.git

あっ!!落とすんじゃねえぞ!!ゼッタイ落とすんじゃねえぞ!!自分でスケルトンを作るんだぞ!!わかったな!!

まずは手始めに簡単そうなとこから…

初歩的な当たり判定(矩形の当たり判定、円の当たり判定)

まずはすでにやってると思うんですが、当たり判定を作っていきます。簡単ですね。1 年生の時に使った当たり判定といえば、

- XY 範囲の当たり判定(AABB って言われてっからそれ)
- 円の当たり判定(距離の当たり判定)

AABB の判定は
bool IsHitAABB(const Rect& lval , const Rect& rval);

の定義で円の当たり判定は
bool IsHitCircle(const Circle& lval,const Circle& rval);

てな名前で作って。中身はまずは自分で考えてみてね。C++がわかる人は引数オーバーロードを用いてIsHit関数一本化しても構いません。

あっ…構造体が使えない(わからない)人は別に…
bool IsHitAABB(int ax,int ay,int aw,int ah,int bx,int by,int bw,int bh);

でも構いませんし

bool IsHitCircle(int ax,int ay,float ar , int bx, int by, float br);

でも構いません。数学を用いて当たり判定できていればいいです。
なお、ar,brはAやBのradius(半径)の略です。

ちなみに構造体に関してですが

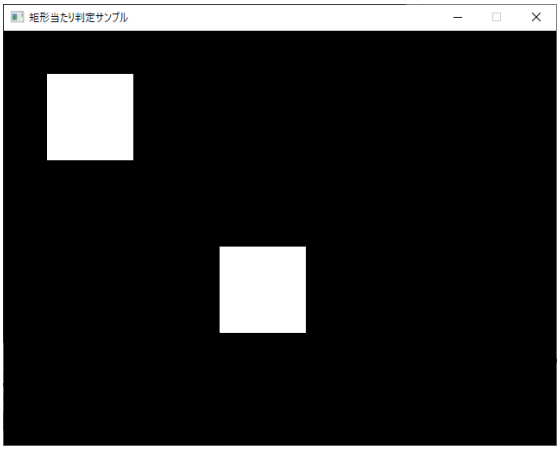
```
struct Rect{
	int x, y, w, h;//x座標 y座標、幅、高さ
};

struct Circle{
	int x,y;//中心座標 xy
	float r;//半径
}
```

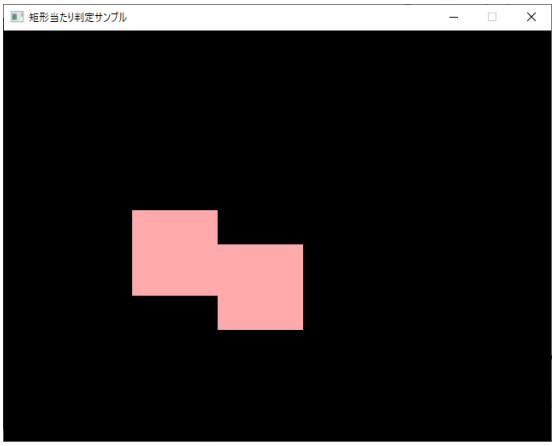
という風に定義します。もし余裕があるならこれらの構造体はGeometry.hというヘッダを作って、そこに置きましょう。

正直な話、このGeometry.hおよび今後作っていくであろうGeometry.cppが数学応用授業の生命線となります…なので、できれば…じゃなくて作っとこう。

そんで例えば矩形の当たり判定なら



当たってないとき～



当たってるとき～

てな感じで色を変えたりして、当たっている状況がわかるようなコードを書いてみよう。たとえばこんな感じかなあ。

```
while (!ProcessMessage()) {
	ClearDrawScreen();
	GetHitKeyStateAll(keystate);
	if (keystate[KEY_INPUT_LEFT]) {
		vx-=10;
	}
	if (keystate[KEY_INPUT_RIGHT]) {
		vx+=10;
	}
	if (keystate[KEY_INPUT_UP]) {
		vy-=10;
	}
```

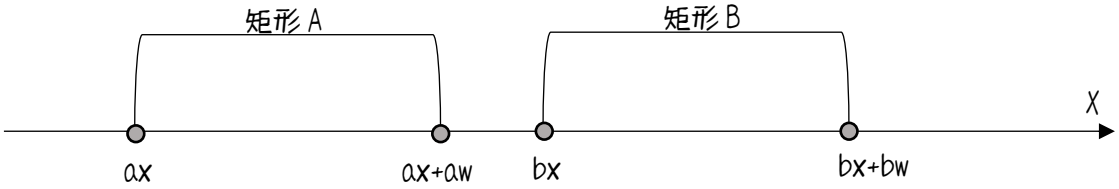
```
		if (keystate[KEY_INPUT_DOWN]) {
			vy+=10;
		}
		rcA.x += vx;

		int color = 0xffffff;//すっげぇ白くなってる。はっきりわかんだね
		if (IsHit(rcA, rcB)) {
			color = 0xffaaaa;//赤くなってんぜ…
		}
		rcA.Draw(color);
		rcB.Draw(color);
		ScreenFlip();
}
```

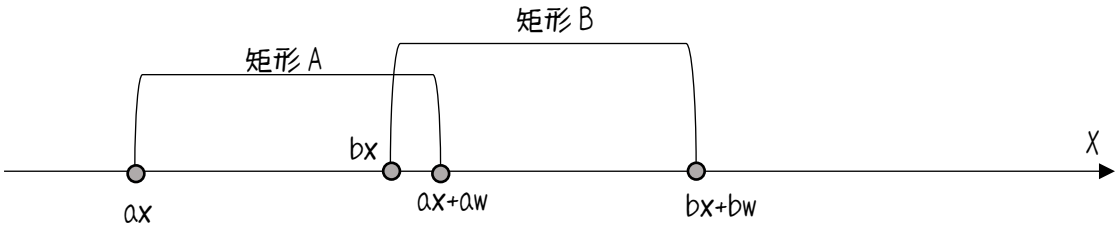
まま、矩形のほうはこれくらいでええやろ…あ、これそのままただ写経しても動かへんからね？必要な部分は自分で書いてネ。もしかして…？分かりませぇん？

矩形の当たり判定…というかAABB(AxisAlignedBoundingBox)はまず「軸ごとに分けて考える」が鉄則でございます。軸ごとに分けられればあとは数直線…算数の世界。非常に簡単でリーズナブル。良心的アルゴリズムでございます。

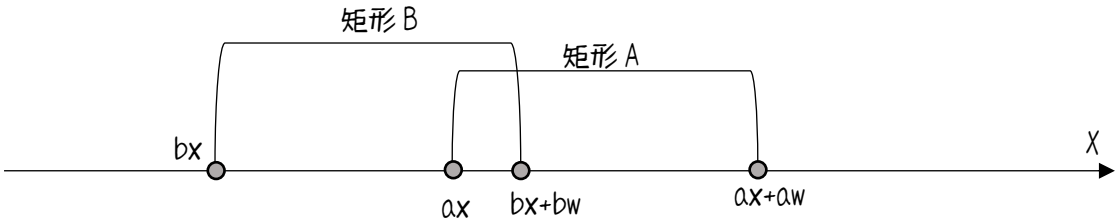
(※大量のものと当たり判定する場合はクアッドツリーなどを用いて最適化します)
まずX方向だけ考えてみよう。例えば以下のような位置関係だとすると



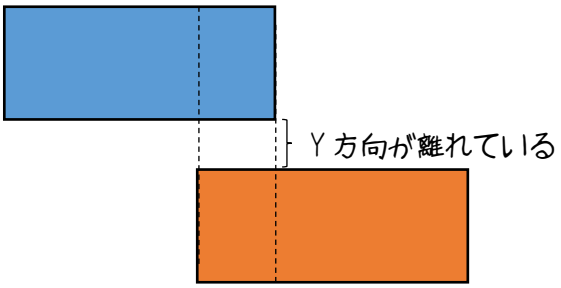
これは…当たってないよね？
という事は「当たっている」とは



この図のようにbx≦ax+awになっている①
それか、もしくは

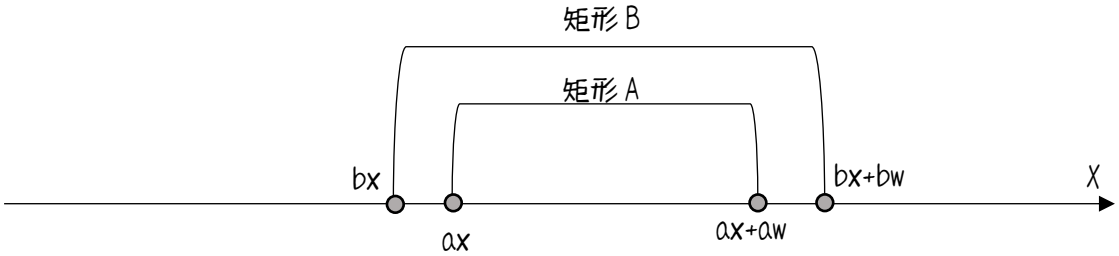


この図のように ax≦bx+bw になっている②
…と言いたいところだが、これは X 方向だけなので、これだけ成り立っても当たらない場合がある。どういうことかというと、Y 方向が離れているときですね。



この時は当然の権利のように当たりませんので、X 方向と Y 方向双方の範囲が重なっているときにのみ当たると思ってください。

うーん、しかし、これでは A が右にある時と B が右にある時で場合分けしないといけないし、もっと言うと



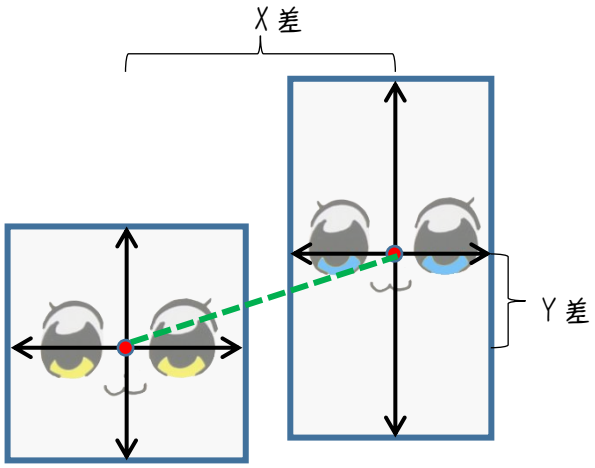
こういう場合もあったりしてこれもうわかんねえなってなりそうなんだけど、
よく考えて**「当たってない」条件に注目するともうちょっとシンプルに考えられますね。当たってない当然の条件とは、片方の右よりもう片方の左が右にある時**つまり

```
max(ax+aw,bx+bw)<min(ax,bx)
```

であれば確定で当たってないです。つまり

```
return !(max(ax+aw,bx+bw)<min(ax,bx) || max(ay+ah,by+bh)<min(ay,by))
```

これだけで矩形の当たり判定になります。また、ちょっと考え方を変えて「矩形の座標」を「矩形の中心」として考えれば

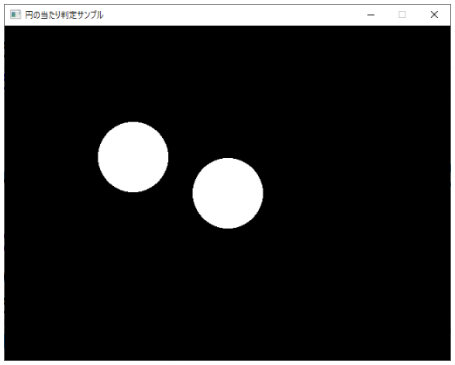


このような位置関係になるため、円の当たり判定の時のように「中心点間の距離」より幅および高さが離れているかどうかで判定でき、より概念的にわかりやすいと思います。

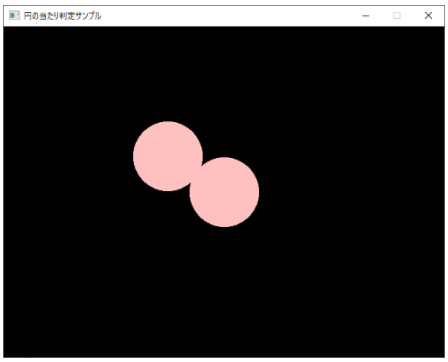
```
return abs(bx-ax)<aw+bw && abs(by-ay)<ah+bh;
```

てな感じに考えられます。

さて次に円の当たり判定ですが、これは簡単ですね。さっきの矩形のコードをちょっと変更して、頑張って

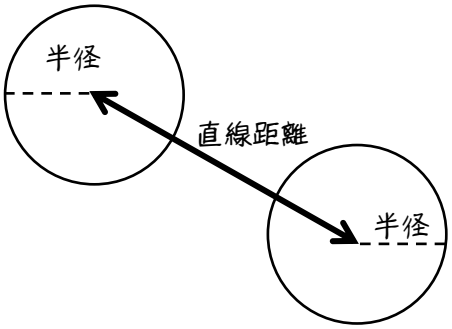


あたってな～い



当たってる

が判別できるようにしてください。



これはノーヒント…と言いたいところですが、ちょっとだけヒントを…三平方の定理はわかっているとは思いますが、C 言語には便利な関数があります。それが **hypot 関数**です。

#include<math.h>もしくは#include<cmath>をする必要がありますが、**hypot 関数というのは斜辺の長さを求めるもの**です。定義はこう

```
#include<math.h>
double hypot (
    double x,
```

```
    double y
);
```

hypot 関数は x の 2 乗と y の 2 乗の和の平方根を計算し,結果を double 型で返します。厳密さを考えるなら float 用に hypotf を使うべきですが、特に気にしなくていいでしょう。ピタゴラスがわからん人にとっては hypot は神関数。

つまり

```
float distance = hypot(xdiff,ydiff);
```

とやれば中心点間の直線距離がわかるので、それ知ってさえいればすぐに終わると思います。

※ただし hypot 関数の中身は結局

```
float hypot(float x, float y){
    return sqrt(x*x+y*y);
}
```

で、できているため、効率厨には好まれません。どういう事かというと sqrt 関数の処理がやたらと重たいのです。重たい理由は君たちの脳みそと同じです。どういうことかというと、2 乗するのと√するの、自分の頭でやった場合どっちが時間がかかりますかね…？

例えば32×32を頭の中で計算するのと$\sqrt{1296}$を頭の中で計算するの、どちらが時間がかかりますか？ちなみに1296ならまだ整数^{2}なのでいいですが、結果が無理数の場合は？

簡単に言うとニンゲンサマに時間がかかるものは結局コンピュータにも時間がかかるというわけ(コンピュータはあまり疲れないので計算間違いもないし、スピードも速いのは確かだけど、2 乗よりルート(平方根)のほうがはるかに時間がかかるわけ)

という事でここではお勉強の為に hypot を教えましたが、大量の当たり判定を行ったりする際には 2 乗のまま判定をしたほうがいいということです。

あと、知らない人もいると思うので、円を描く関数 DrawCircle を教えておきます。

```
DrawCircle(中心 x,中心 y,半径,色,内部を塗りつぶすかどうか);
```

なので

```
DrawCircle(ax, ay, radiusA, col, true);
```
って書けばいいだけです。

っさぁ～頑張ろう!!!

ああ、くっっっそ余裕がある人は、矩形や円が「重ならない」ように「押し戻し」まで実装してみてください。矩形はともかく、円の押し戻しがヤバいレベルでめんどうですが、がんばったらそれなりに見返りはある(就職活動的な意味で)と思います。

はい、当たり判定(初歩)はこんなもんでーす。

次は三角関数の初歩をやってみましょう～

三角関数

神関数の atan2

三角関数って、たぶん紙でやるときはクッソ難しかったと思いますが、プログラムでやるときは多分たいして難しくない…んじゃないかなって思います。

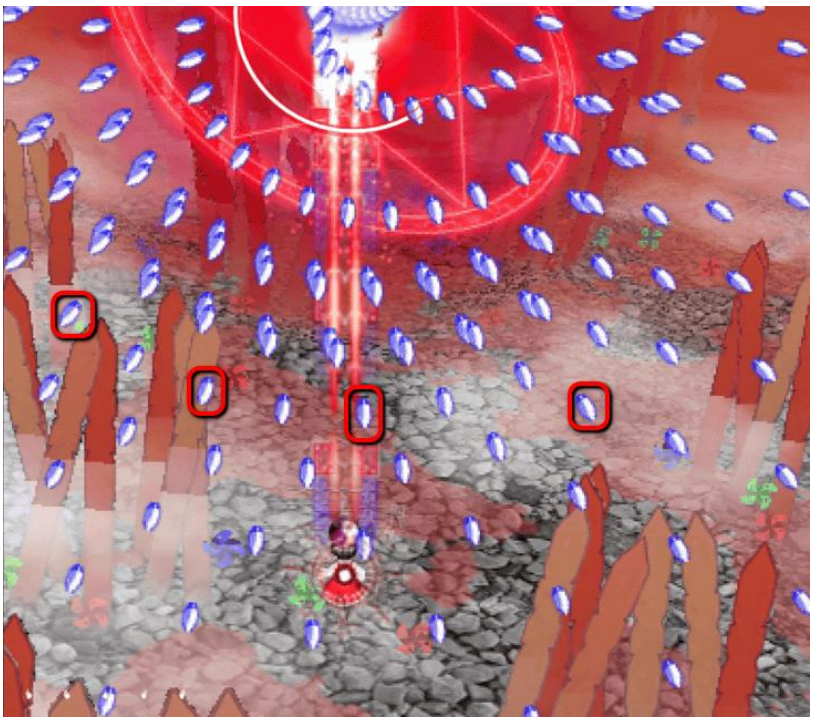
どっちが cos でどっちが sin かだけ区別がつけば十分です。逆に言うとこれの区別だけはつけてください…ほんとに困るんで…どっちか覚えれば他は自動的にわかるレベルなんだからさ～覚えてくれよな～頼むよ～。

あれ？tan は？って思うかもしれませんが、実は tan じたいにそれほど出番はありません。

だけど C 言語には素晴らしい神関数があります。この神関数をねっとりと使って…どうぞ。

神関数の名は atan2。なぜ 2 なのか 1 はあるのか…それは後でお話ししますが、とにかく神。

例えばシューティングなどで

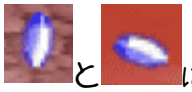


このように放射状に弾幕を張ることがあります

で、ひとつひとつの弾をよく見てみると…わかるだろう？

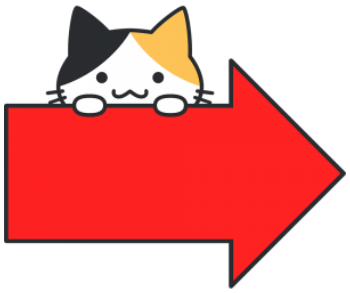
専門的なことはともかく…回転しているんDA

分かりやすい例だと以下の例ですが

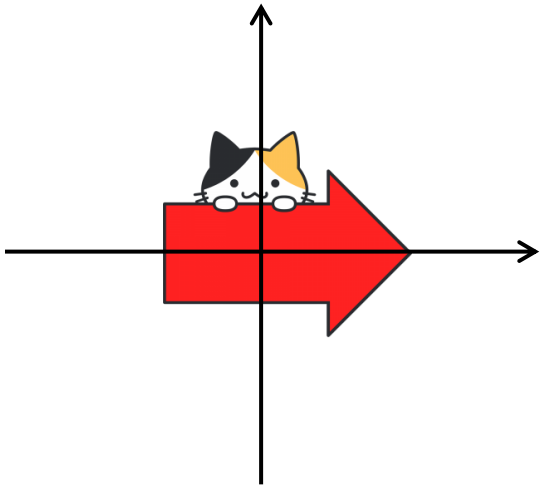


は明らかに角度が違いますね？つまり画像が回転している。DxLibにおいて画像を回転させるにはDrawRotaGraphを使用するのですが、どれくらい回転させればいいのでしょう…。

そういったときに神関数をねっとりと使ってやればいいのです。まずは元になる画像を用意します。この時に元画像の向きが右向きになるような画像を用意してください。



画像は好みで選んで、どうぞ。なぜ右向きなのかというと、座標平面上の角度0°が右向きを表すからです。



で、ここからどのくらい角度を変えたいのか、例えばこの矢印を特定の方向…例えば原点(0,0)から見て(3,4)の方向に向けたいとします。

そんな時に**神魔法 atan2 をねっとりとかけて**やって、戻ってきた値を DrawRotaGraph に入れてやれば思った向きに回転します。
関数 atan2 の定義はこうです。

```
#include <math.h>
double atan2(
    double y,
    double x
);
```

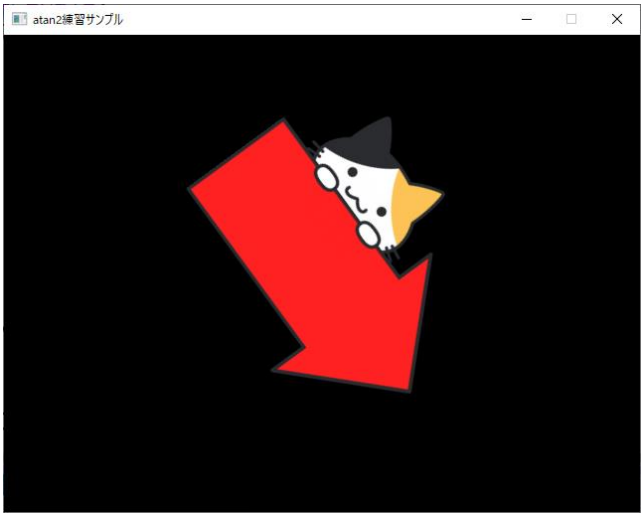
「atan2 関数は y/x の逆正接 (arctan) を計算し、結果を double 型で返します。」

注意するのは第一引数が y で第二引数が x ってとこですね。これよく逆にしちゃうんだよね。

プログラムで書くとこうなります。

```
auto angle = atan2(4,3);
DrawRotaGraph(250, 250, 1.0f, angle, handle,true);
```

結果はこうなります。



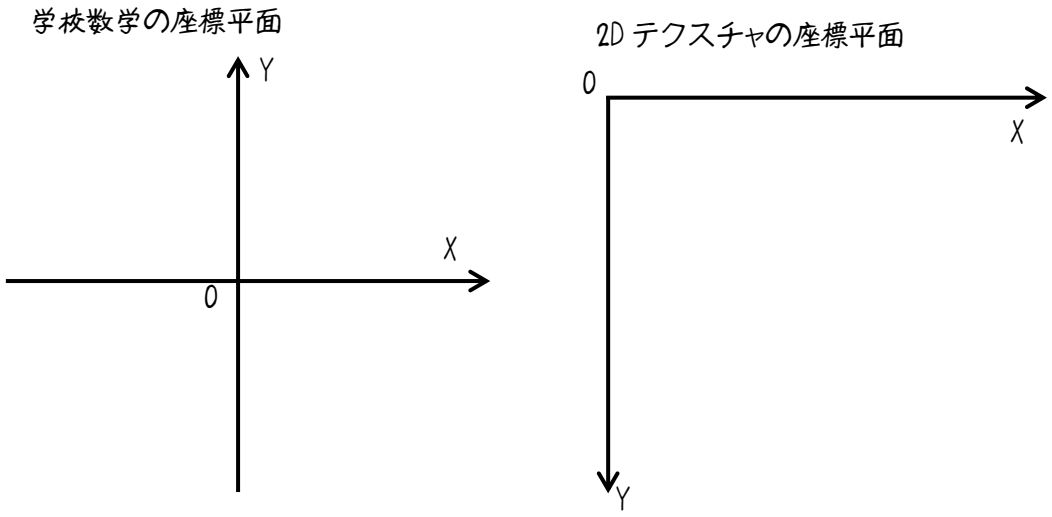
あれ？(3,4)って上向きじゃないの？

うーん…

これね、ゲームにおける 2D 座標系と、数学の座標平面の表現の違いが出てるんですわ。

どういうことかというと、数学の座標平面は上がプラスだったんだけど、コンピュータグラフィクスにおける 2D のテクスチャに対する座標系は下がプラスなんですよね。

ホントに面倒なんですが、「そういうルール」故…致し方なし



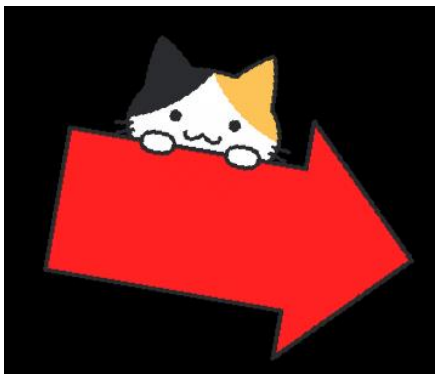
Y 軸方向だけ、向きが違う。もうちょっというと原点の座標も違う…。というわけで向きが反対になるのです。しかし、安心するがいい。どの道、atan2 に代入する数値がそもそも 2D テクスチャ座標平面であるから、その値をそのまま使用して RotaGraph で回転させてもきちんと辻褄は合うのだ。

例えば「**矢印の向きを、今マウスがあるところを指し示すようにしたい**」という仕様だったとします。そうなると向きは「マウス座標」マイナス「中心となる座標つまり画像が配置されている座標」という事になるためプログラムは

```
auto angle = atan2(my - posy, mx - posx);
DrawRotaGraph(posx, posy, 1.0f, angle, arrowcatH,true);
```

このようになります。mx,my がマウスの座標。posx,posy が中心座標として、プログラムを書いてみてください。うまくいけば矢印が自分のマウス座標を常に向くことがわかります。

で、もしかしたら矢印の画像が



このように辺にガタついた絵になるときがあります。これは仕方ない事で、元々回転していない画像(ピクセル毎に色が配置されている)を回転させて、その結果はまたピクセル毎に配置されるわけですから、当然ながらガタつきます。直線とかが分かりやすいので、例えば

ね？

水平もしくは垂直のときはきれいな直線になるんだけど、ちょっと角度が付くとガタガタでしょう？それは普通に画像の回転も同じなわけ。まま、しゃあないⅠ…けど、実は「バイリニア補間」というのを採用すると

若干マシになります。これは DxLib_Init の直後にでも

```
SetDrawMode(DX_DRAWMODE_BILINEAR);
```

とでも記述してあげれば、画像がきれいに表示されます。ただしこれも注意点があります。このバイリニア補間…ドット絵とは相性が非常に悪く。ドット絵のパキッとした感じが失われますので、使いどころさんには気をつけましょう。

sin と cos は区別つく？

もうハッキリ言ってしまおう。sin は縦で、cos は横である。以上。

さて、前の実習で回転させた矢印を矢印の向く方向に動かしてみたいと思います。どうすればいいのでしょうか？

そう、x 方向の速度 vx は横方向の速度だから cos に比例し、vy 縦方向の速度だからは sin に比例する値を入れればいいのです。

そしてそれを現在座標 x および y に加算すればいいのです。簡単ですね x+=vx とかでいいんです。

sin と cos の使い方は至って簡単。sin(角度)、cos(角度)。さぁ、さっきのプログラムを改造してさっそくやってみましょう。

※注意点…座標情報は int で宣言しているかもしれないが、今回のように角度が絡んでくると整数型だけで扱うのは難しくなってきます。なので現在座標の x,y および速度の vx,vy に関しては、float 型で宣言しておいてください。うまくいけば矢印がマウスの方向に向かって動きます。

ベクトル

そろそろX方向とY方向を別々に扱うんじゃなくて、いっぺんに扱いたいかなーって思います。いわゆるベクトルの話なんですが、ベクトルとは一言で言うと「同一種の値をまとめたもの」だと思ってください。実際std::vectorを使ったものも「同一種の値をまとめている」のだから、意味的には大差ないです。

作り方は簡単で、まずGeometryクラスをクラスウィザードで作っておいて、その中にVector2構造体を作りましょう。

要件は
1. float型のxとyをもつ
2. ベクトルの大きさを返す関数Magnitudeを持つ
3. 正規化関数(大きさを1にする関数Normalize)を持つ
4. ベクトルの加減算機能を持つ

このくらいです。回が進むごとにここに内積関数や外積関数が追加されることになりますが、今は必要なものだけでいいでしょう。

ここはどちらかというと数学と言うよりもプログラムですが作ってみましょう。

基本と言うか初歩

```
//2Dベクトル構造体
struct Vector2{
	//要件に従って自分で書いてみて下さい
};
```

ひとまず思った通り作ってみましょう。あまり深く考えすぎずに…。

いかが？

まずは基本部分ですが、floatのxとfloatのy。これをstructの要素にすればいい。ちなみにtypedefもしくはusingを使用して、Vector2の別名型のPosition2も宣言しましょう。

```
typedef Vector2f Position2f;
```

とでもやって、座標的なものもベクターとして扱えるようにします。

ベクトルの基本演算

次にベクトルの基本演算ですね。加算減算、スケーリングくらいまで実装すればいいでしょう。
まだオペレータオーバーロードは使いませんが、使える人はそれで実装しておいてください。

```
//加算
Vector2 Add(const Vector2& lval,const Vector2& rval){
	return Vector2(lval.x+rval.x,lval.y+rval.y);
}
```

加算は↑のコード見せたるので、減算(Subtract)を自分で作ってください。

```
//減算
Vector2 Subtract(const Vector2& lval,const Vector2& rval){
	ここは自分で書いてね
}
```

では次に Scale ですが、これも簡単ですね。引数にスケーリング値を入れて、x も y もそれ倍すればいいわけです。

```
void Scale(float val){
	ここも自分で書いてね
}
```

あと、自分自身を n 倍したものを返し、自分自身は返さない Scaled 関数も作りましょう。

あ、ここまでの話は C++をよく分かってない人にも伝わるように書きましたが、C++を使いこなせてる人は演算子オーバーロードを使って、＋演算子や＊演算子を定義しておきましょう。

大きさを測る

次に、大きさに関してですが、これは Magnitude(大きさ)と言う関数を構造体内に書いちゃいます。戻り値は float で

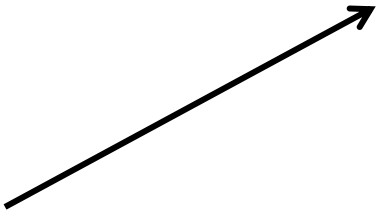
```
struct Vector2{
	//x,y の基本要素を書いてください;
```

```
        //ベクトルの大きさを返す関数
        float Magnitude()const;
};
```

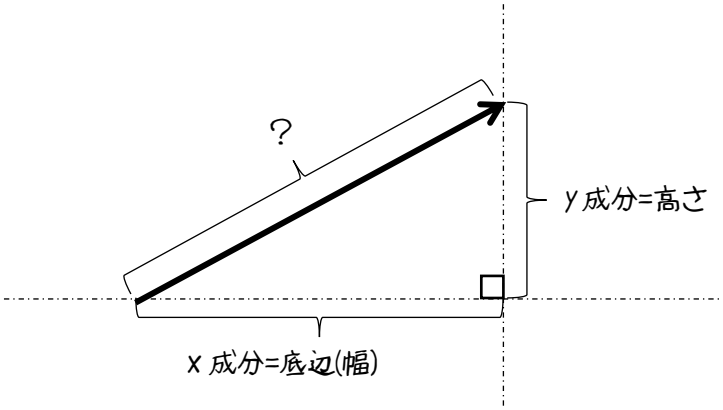
あ、どうしても構造体内に関数を書くのが気持ち悪い人は別に構造体外に書いても構いません…が、その場合は引数に Vector2 への参照を書くのをお忘れなく。

```
float Magnitude(const Vector2& v){
        //ベクトルｖの大きさを返してください
}
```

いや、ちょっとお前さ、ベクトルの大きさを返すとかしれっと言ってるけど、どうやって返すか知らねえんだけど？
え？習ってない？習ってない？本当に？それホントに M センセーの前で言えんの？
まま、ええわ、もっかい説明したろ。
ベクトルってのはさ、まぁご存知だと思いますが以下の矢印のようなもんです。



2D 直交座標系において、ベクトルと言うのは x 方向にいくつ、y 方向にいくつ進んだかで表されます。つまり



この直角三角形の斜辺の長さが分かればいいわけです。で、そもそも x 成分 y 成分がそれぞれ底辺と高さに対応しているわけなので…あとは分かりますね？なんだったら前に説明した hypot 関数を使えば即終わります。やってみましょう。

さて、長さが分かったところで正規化関数(大きさを 1 にする)も作りましょう。

ベクトルの正規化(標準化)

既に大きさを求めてるんで、簡単っすねぇ～。ベクトルを 1 にするっていうことは、ベクトルの大きさで元のベクトルを割ってあげればいいんじゃないかなーって思います。

```
auto len=Magnitude();
x/=len;
y/=len;
```

ひとまず関数名を Normalize としてやる。Magnitude と同様に構造体に Normalize 関数を追加しても良く、そんな感じなら

```
struct Vector2{
	//x,y の基本要素を書いてください;

	//ベクトルの大きさを返す関数
	float Magnitude()const;

	//正規化関数
	void Normalize();
};
```

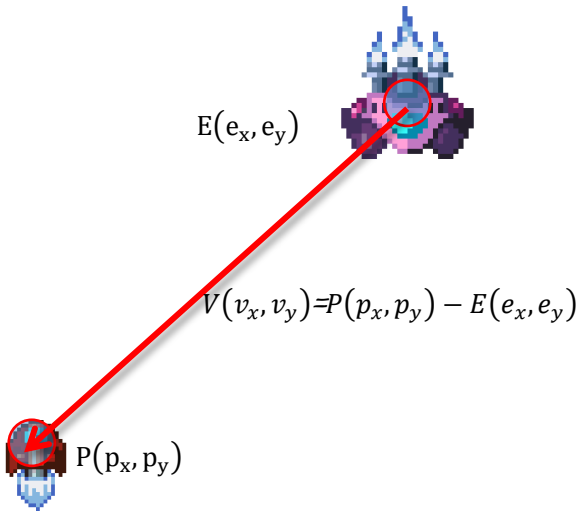
てな感じであとは適当に実装してもらえればいいです。もし本体を正規化したくなくて、正規化済みのベクトルを返したければ Normalized って関数を用意して

```
Vector2 Normalized(){
	Vector2 ret=*this;
	ret.Normalize();
	return ret;
}
```

でもいいんじゃないかなと思います。うん、作っといてください。

なんで正規化なんて必要なのかと言うと、例えばシューティングで「自機狙い弾」を作りたいとします。どうやって作りますか？

うん、まずは敵機から自機に向かうベクトルを作らなければならないので、



計算は終点から始点を引く…つまり自機の座標から、弾を発射する敵機の座標を引けば方向だけは合致したベクトルを作る事ができます。

ただし、今の速度ベクトルはあまりよろしくない…何故だか分かりますか？ベクトルの長さ」が大きすぎるんです。上図のベクトルの場合、敵から一瞬で殺されます。どういうことかわかりますか？敵の座標はＥでスピードはＶです。

そうなると、座標Ｅから出発した弾は、次の瞬間には…

$$E + V = E(e_x, e_y) + \ (P - E) = P$$

となってしまい、1 フレームあれば着弾してしまいます。流石にそんなんシューティングゲームが成り立つわけありません。

そこで正規化して、ベクトルの長さをまずは一律に1にしてしまいます。そうなると方向を保ったまま指定のスピードで進むことができます。例えば 1 フレームに 2 ピクセル数分のスピードで進みたいなら

```
Vector2 velocity=Subtract(playerpos,enemypos);
velocity.Normalize();
velocity.Scale(2.0f);

(中略)

pos = Add(pos,velocity);
```

てな感じで実装すれば自機狙い弾がすぐに作れます。なお、演算子オーバーロードを実装できる人なら

```
pos += velocity;
```

てな感じで、直感的な書き方ができると思います。だから僕個人的には演算子オーバーロードは使いこなせるようになって欲しいけど、難しいと感じる人もたくさんいそうなので、今回みたいな書き方をしています。

もし、演算子オーバーロード知らないけど、使いこなしたいという要望があれば教えますんで、放課後にでもとっ捕まえてください。

課題①簡単な弾幕避けゲーの作成

避けゲー要件:

1. 自機と敵機がいる
2. 敵が一定のタイミングで弾を発射する
3. ***敵は一度に複数の弾を発射する***
4. ***敵は一定時間ごとに発射する弾のパターンを変える***
5. ***弾のパターンは以下の通り***
   - ***(ア) 自機狙い3way弾***
   - ***(イ) 放射状弾***
   - ***(ウ) バラマキ弾***
6. ***その他自分なりにパターン組める人は組んでください。評価に入れます***
7. ***敵はパターンで動く(左右に適当に動けばいいです)***
8. ***自機に敵の弾が当たったら反応する(円の当たり判定)***

ガワはつくっておいてあげるので、赤い部分を自前で実装しておいてください。でもさっさと作る事をお勧めしておきます。頑張ろう。

提出期限を、お知らせします。7/17(金)18:00です。非リモートになってからでOKなので、学校のサーバに上げてください。

¥¥stfs¥apc_abcc クリエイティブ¥gakuseigame¥通常授業以降¥川野¥数学課題提出先¥シューティングゲーム
の中に、『学籍番号_氏名』のフォルダを作って、そこにexeと素材を上げてください。用件を書

きますね。

exeと素材単品で動く事

- 必要なものだけ提出していること(ソースコード、プロジェクトは不要)
- タイトルバーに「学籍番号_氏名」が表示されるようにしてください
- サーバ上で動かなければ動かないとみなします
- 前述の「赤文字の要件」を満たして実装していること
- 期日は厳守する事

とします。学校にこれない人は、メールだのクラウドだのに上げて提出してください。
要件を満たしてなければ、この課題の点数はありません(一切の理由は考慮しません、ただ、期限までに提出してください)。
よろしくお願いします。

行列による回転

プログラミングの世界では実は回転は行列で回転しています。DxLib においては DrawRotaGraph で回転できますが、実はこいつは「画像の回転」しかできません。

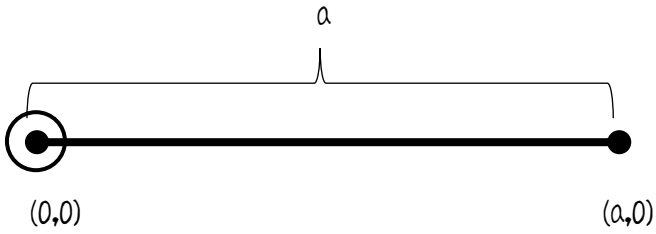
何の問題ですか？

という意見もあるかもしれないですね。それはそうだろう。誰だって意味もないのに新し知識や難しい知識を学びたくない(そうでもない人もいると思いますが)でしょう。
ところがねぇ、例えば

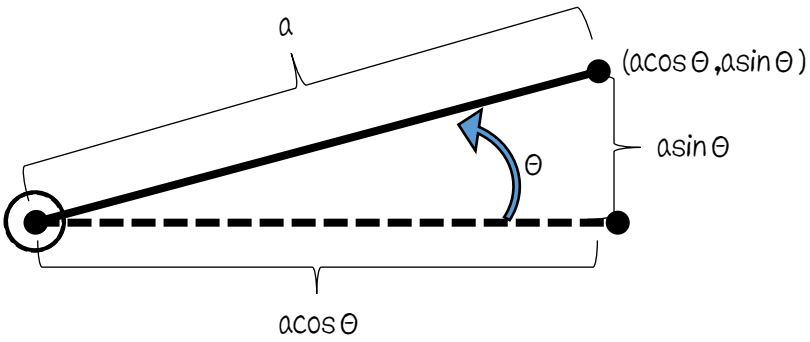
こいつに当たり判定があったとしよう…。おそらく今までみんなは回転のことを考慮しなくてもいい距離の当たり判定(円の当たり判定)や、そもそも回転を考慮しないルールで AABB(矩形の当たり判定)を使っていたことだろう。

回転おさらい

だが残念ながらそうもいかないんだよねえ…まぁ実際には当たり判定の話だけじゃなく、多関節のキャラを動かしたりするとき等に「線分」を回転させるとどうなるのか？を考えなければならないのだ、めんどくせえけど。つまり…

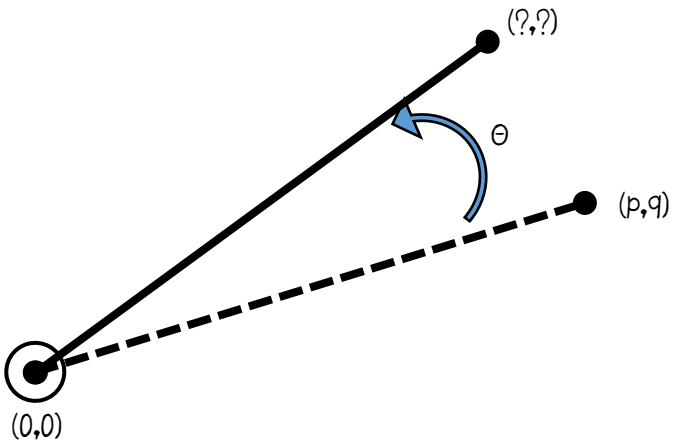


これを…



こうするとする。
そうなると(a,0)→(acosθ,asinθ)これはわかるだろう。

では次にこういう風に最初から傾いているものをさらに回転させることを考える。



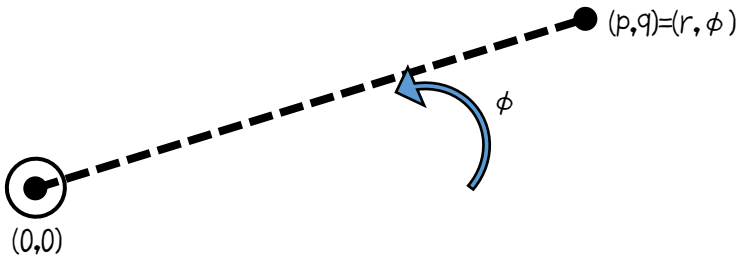
ちょっとこの辺から、**1年生までにやったところを理解してないとガチ難しい**のでよく聞いてください。回転した結果を？、？にしてますが、どうなるんでしょうね、これ。

ここで「極座標変換」という考え方を導入します。実は Photoshop や GIMP の中でも応用されている考え方なので、知っておきましょう。

一応さ、みんなが今まで使ってるような(x,y)みたいな座標表現を「直交座標」って言うんだ。X 軸と Y 軸が直交してるでしょ？

それに対して「極座標」ってのは、その XY 座標を、半径と角度で指定するものなんだ。で、角度は一応 X 軸からのなす角度で考える…と。角度をなんだろ、適当に角度を φ って記号であらわすとして、原点からの距離を r とすると、さっきの(p,q)を(r,φ)であらわす。これが極座標です。

何言ってんだ？って思うのかもしれませんが、そもそも(p,q)を(r,0)から φ 回転したものと考えるのです。



で、そうすると $p=r\cos\phi$、$q=r\sin\phi$ と表せます。この式覚えといてね？
この辺分かりますか？納得いかないなら言ってね。で、もう X を cos で Y を sin でって分かってるので、それを取っ払って表現したのが極座標表現です。

逆に言うと $r=\sqrt{p^2+q^2}$ なので相互に表現が行ったり来たりできるわけです。応用に関してはもう何よりもまず、Photoshop とか GIMP とか見てね。

さて、極座標として考えると(p,q)→(r,φ)としてみる事ができるというのを一応納得していただきたいのですが、納得したという前提で描きます。

この(r,φ)を Θ だけ回転させたらどうなるのか…それは当然(r,φ+Θ)です…そらそうですよね？

さて、ここで直交座標に戻してしまします。元々、$p=r\cos\phi$、$q=r\sin\phi$ でしたね？さて、回転後はどうなるのかというと

$p'=r\cos(\phi+\Theta)$

$q' = r\sin(\phi+\theta)$

となります。大丈夫？死んでない？こっからもっと死ぬんでまだ生きてろよ～。もう忘れてると思うけど、加法定理って、したことある？「ないです」とか言うなよ？数学のセンセー怒るぞ？

まま、ええわ。おさらいすとこ。

$\sin(\alpha\pm\beta)=\sin\alpha\cos\beta\pm\sin\beta\cos\alpha$
$\cos(\alpha\pm\beta)=\cos\alpha\cos\beta\pm(-\sin\alpha\sin\beta)$

こんなやつ。なんか覚えとりますかね？これも導出までやれるんですが、そんな時間ないので、知りたい人は個別に聞いてください。リクエスト多ければ解説しますけど。

さて、専門的な事はともかく、この加法定理をさっきの式に適用するんDA

$p' = r\cos(\phi+\theta)$
$q' = r\sin(\phi+\theta)$

クッソめんどくさそうなのが、わかるだろう？でもやってみるんだ。

$p' = r\cos\phi\cos\theta - r\sin\phi\sin\theta$
$q' = r\cos\phi\sin\theta + r\sin\phi\cos\theta$

右辺値がrでくくれちゃうのですが、まぁそれは置いといて、注目すべきは$r\cos\phi$と$r\sin\phi$です。

最初の定義
$p=r\cos\phi$、$q=r\sin\phi$
を思い出してください。これを↑の式に逆に当てはめると…こうなる

$p' = p\cos\theta - q\sin\theta$
$q' = p\sin\theta + q\cos\theta$

お分かりいただけただろうか…そう、コブラのマシンはサイコガンが…見えるだろう？

これで任意の線分を**「原点を中心に」回転**させることができる。のはお分かりいただけたかなと思う。

うん、ここまでが１年生の時に習った内容だと思うんだけど、ここからさらに次がある。

これではちょっと足りないのだ。残念ながら。何が足りないのかというとここまでの場合、「原点中心回転」という制約がついたままなのである。
ということで

任意座標回りの回転

について考慮する必要があるのだ。

Do you understand?

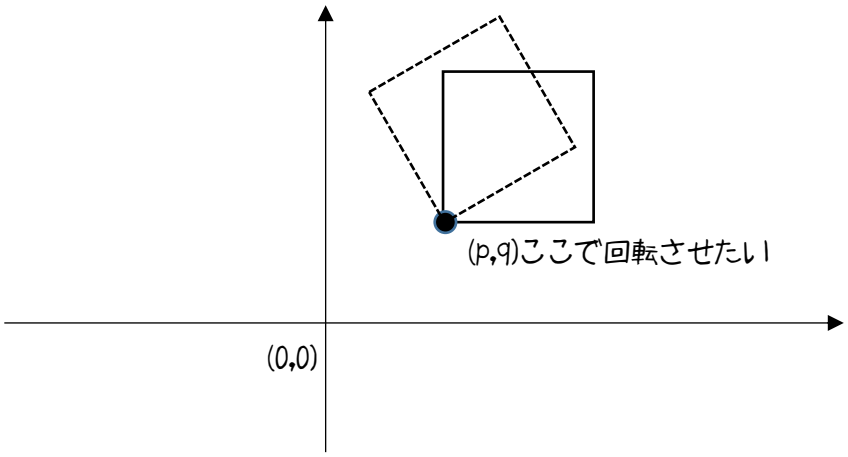
OK。オーイェー!
大したことじゃない。任意座標の回転は次の言葉を覚えてればいい。

平行移動に始まり平行移動に終わる

なんだっそら…。

と思うかもしれんが、とりあえず聞いてほしい。回転行列の回転というのは何をどう頑張っても「原点中心の回転」しかしないのである。もう、しゃあないねん。アキラメロン。

ということで、見とけ見とけよ～



たとえば、ある図形を原点から離れた座標(p,q)を中心に回転させたいとする。ただし先ほども書いたように、原点中心にしか回転できない。というわけで、どうするのかというと、無理やり原点に持って行くのである。

つまり

1. 原点に持って行くために(-p,-q)平行移動
2. Θ回転(コブラのマシンはサイコガン)
3. 元の座標に戻すために(p,q)平行移動

の三つの操作を行うのである。

大丈夫かな？息してるかな？頑張れ。俺も深夜にこれを書いてる。死にそう。

でさ、ちょっと俺が間に合ってないんだけど、もし授業でここまで来ちゃったらさ、やってほしい事があるんだ。

ちょっとさ、以下の関数を使って、原点から離れた場所にまず長方形を描画してくれない？

```
// 四角形を描画する
int DrawQuadrangle( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4, int Color, int FillFlag ) ;
```

あ、これ DxLib の隠し関数ね。で、それが書けたら、4つの頂点のどこかを中心に回転させてみて？

ここまでの説明で、できる人はできるはずなので…。

gakuseigame¥通常授業以降¥川野¥数学課題資料¥DrawQuadrangle.zip

を参考に、とある点を中心に回転した点を返すRotatePosition関数を完成させてください。一応、右クリと左クリで回転方向を変えるようにしています。

今のところこいつにクリックした点と、どっち方向に回るのかを渡しているためクリック点を中心に回ってるか確認してください。

失敗してたら正方形がどこかに吹っ飛ぶか、ゆがんだ正方形になるかと思います。正方形を保ったまま回転できるようにしてください。

で、やってみてもらえれば分かると思いますが、

```
//それぞれの頂点に回転変換を施す
for (auto& pos : positions) {
	pos = RotatePosition(Position2(mx, my), angle, pos);
}
```

全頂点にこの関数を適用しています。
まぁ、4頂点しかないので、今回は問題ないですが、頂点数が 1 万とかになるとちょっと辛いことになってきます。

ちなみに演算回数としては、1頂点あたりsin、cosがそれぞれ2回ずつ、足し算引き算もそれぞれ2回ずつ。このうちsinとcosがちょっと重い処理だしなんとかしたい。

そこで出てくるのが行列です。

行列による回転

表題がかぶっちゃってますが、行列による回転を考えます。行列は演算というか操作を記憶しておくことができます。さらに操作を合成して記憶しておけます。

つまり、原点に移動して回転して元の座標に戻すという操作を1つの行列に合成しておき、あとからこれを演算できます。数学的に書くと…まずは(-p,-q)移動するので

$$\begin{pmatrix} 1 & 0 & -p \\ 0 & 1 & -q \\ 0 & 0 & 1 \end{pmatrix}$$

こういう行列です。

まま、ここまでは大抵の人はわかるやろ…次に回転だけどこれは簡単で、コブラのマシンは

サイコガンやね。

$$\begin{pmatrix} cos\Theta & -sin\Theta & 0 \\ sin\Theta & cos\Theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

最後に元に戻す行列

$$\begin{pmatrix} 1 & 0 & p \\ 0 & 1 & q \\ 0 & 0 & 1 \end{pmatrix}$$

これ、これらを全部合成することができます。

$$\begin{pmatrix} 1 & 0 & p \\ 0 & 1 & q \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} cos\Theta & -sin\Theta & 0 \\ sin\Theta & cos\Theta & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & -p \\ 0 & 1 & -q \\ 0 & 0 & 1 \end{pmatrix}$$

計算すると…

$$\begin{pmatrix} cos\Theta & -sin\Theta & p - pcos\Theta + qsin\Theta \\ sin\Theta & cos\Theta & q - psin\Theta - qcos\Theta \\ 0 & 0 & 1 \end{pmatrix}$$

もう、気が狂うほど複雑なんじゃ

まぁ、これ、プログラムで組むと意外なほどに簡単(？)なのでやっていきましょう。

まず Matrix 構造体を作ります。中身は

```
struct Matrix {
	float m[3][3];//3x3行列ですやん
};
```

これを Geometry.h に書いといてください。

とりあえず、何もしない I 行列を返す IdentityMat 関数を作ってください。何もしない I 関数ってどういうのでしたっけ？そう、単位行列ですよね。

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

ひとまず、IdentityMat 関数だけ書いといてやるから、あとは自分で考えて
平行移動する TranslateMat 関数(float x,float y)
および原点中心回転する RotateMat 関数(float angle)
を作ってください。

```
Matrix IdentityMat(){
	Matrix ret={};
```

```
        ret[0][0]=ret[1][1]=ret[2][2]=1.0f;
}
```

これ参考にしてね。
あと行列同士の掛け算も考えるので、オペレータオーバーロードでもいいし、MultipeMatrix 関数でもいいので、ともかく行列乗算の関数を作ってください。

とりあえずヒント…

```
struct Matrix {
        float m[3][3];
};

///単位行列を返す
Matrix IdentityMat();

///平行移動行列を返す
///@param x X方向平行移動量
///@param y Y方向平行移動量
Matrix TranslateMat(float x, float y);

///回転行列を返す
///@param angle 回転角度
Matrix RotateMat(float angle);

///２つの行列の乗算を返す
///@param lmat 左辺値(行列)
///@param rmat 右辺値(行列)
///@attention 乗算の順序に注意してください
Matrix MultipleMat(const Matrix& lmat, const Matrix& rmat);

///ベクトルに対して行列乗算を適用し、結果のベクトルを返す
///@param mat 行列
///@param vec ベクトル
Vector2 MultipleVec(const Matrix& mat, const Vector2& vec);
```

はい、あとは頑張って実装してね～。え？どうやったらいいかわからない？
だって行列1年の時やってんじゃん。全然わからんとか言ったら真島の兄貴が黙っちゃいねえ

ぞ!!!(虎の威を借る狐)。

「習ってない」なんて寝言は聞きたかねえ!!!やれ!!!

まず乗算からやれ!!!

```
///2つの行列の乗算を返す
///@param lmat 左辺値(行列)
///@param rmat 右辺値(行列)
///@attention 乗算の順序に注意してください
Matrix MultipleMat(const Matrix& lmat, const Matrix& rmat);
```

この関数から実装だ!!やれっつってんだるぉおお!!数学の通りにやんだよ!!!

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\begin{pmatrix} k & l & m \\ o & p & q \\ r & s & t \end{pmatrix} = ?$$

これの計算はどうなるんだぁ!?言ってみろ!!!あ?こんな計算もできねえのか!?「6つの値をかけて足す」を合計9回やるだけだゾ。

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\begin{pmatrix} k & l & m \\ o & p & q \\ r & s & t \end{pmatrix} = \begin{pmatrix} ak+bo+cr & al+bp+cs & am+bq+ct \\ dk+eo+fr & dl+ep+fs & dm+eq+ft \\ gk+ho+ir & gl+hp+is & gm+hq+it \end{pmatrix}$$

ああ～、くっっっっっっっっっっそめんどくさいんじゃあ!!!!これをプログラムで書け。あくしろよ。

こんなクソみたいな数式書くより楽だろクソが!!!つべこべ言わずにやれ!!!

あ?プログラムをどう書いたらいいか分からん?なんてことを…。今まで何やってきたんで

すか…ふざけるな!!!(迫真)

しょうがねえなあ…じゃあ俺がいっちょ１行１列目だけ見せてやるか。見とけ見とけよ～。

```
Matrix ret;
ret.m[0][0]=lmat.m[0][0]*rmat.m[0][0]+lmat.m[0][1]*rmat.m[1][0]+lmat.m[0][2]*rmat.m[2][0];
```

あとは↑の1行を9個コピペして、それぞれの行と列に合わせた計算をやんだよ。

コピペする場合は1行目1～3列(まず3つコピペ)をまず全て作っておいて、さらにそれを行数分(3行)コピペすると楽でしょう。

まぁ、ここはそれなりに時間がかかるので、待ってあげるから頑張って自分で書きましょう。
自分で書かないと自分の腕が上がりませんよ？とりあえず「間違ってもいいや」って気分で、とにかく手を動かしましょう。君たちあまり自覚ないかもしれませんが、これを生業とするんですよ？分かったらとっととコーディングしましょう。

書けましたかね？

じゃあ次はベクトル演算関数書いてください！

```
///ベクトルに対して行列乗算を適用し、結果のベクトルを返す
///@param mat 行列
///@param vec ベクトル
Vector2 MultipleVec(const Matrix& mat, const Vector2& vec);
```

正直これは先ほどの行列同士の掛け算ができてたらすぐに終わるものです。ベクトルといっても結局は行列の「列がないやつ」だと考えればわかりやすいでしょう？

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax+by+c \\ dx+ey+f \\ gx+hy+i \end{pmatrix}$$

ともかくこうなるように関数を作ってください。ここまでやってんだから作れるでしょ？

なに？できない！？じゃあ1行目だけみせてやっかんしょうがねえなあ

```
Vector2 ret = {};
ret.x = mat.m[0][0] * vec.x + mat.m[0][1] * vec.y + mat.m[0][2];
```

あとは自分でやれ。さすがにわかるやろ…

で、あとは便利関数。もうわかるやろ。やれ。面倒だから詳しい解説書かない。

```
///単位行列を返す
Matrix IdentityMat();
```

これは

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

を返せ

```
///平行移動行列を返す
///@param x X方向平行移動量
///@param y Y方向平行移動量
Matrix TranslateMat(float x, float y);
```

これは

$$\begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$$

を返せ

```
///回転行列を返す
///@param angle 回転角度
Matrix RotateMat(float theta);
```

これは

$$\begin{pmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

を返せ。以上。

さて、ここまでやれたらあとは必要なものがそろったということで、元の RotatePosition 関数を

```
Matrix mat = MultipleMat(TranslateMat(center.x, center.y),
                         MultipleMat(RotateMat(angle),
                         TranslateMat(-center.x, -center.y)));
return MultipleVec(mat, pos);
```

こういう風に書き換えて、元通りに動くことを確認してください。
で、ここまでだと変換後の座標を返しているんですが、これだとあまり意味がないんですよね。ですので、関数を書き換えます。戻り値を Matrix にします。

```
Position2 RotatePosition(const Position2& center, float angle, Position2 pos) {
```

この関数を…

```
Matrix RotatePosition(const Position2& center, float angle) {
```

こうして、行列を返すようにします。なお、mat をそのまま返せばいいです。

で、呼び出し側もこう書き換えます。

```
Matrix mat = RotatePosition(Position2(mx, my), angle);
//それぞれの頂点に回転変換を施す
for (auto& pos : positions) {
        pos = MultipleVec(mat, pos);
}
```

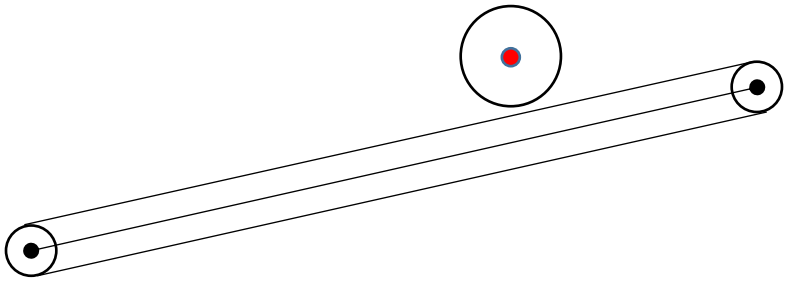
これで「行列一回作ってしまえばあとは適用するだけ」ということがお分かりいただけたであろうと思います。

カプセルと円の当たり判定

さて、なんで回転についてこんなに延々と回りくどいことを話してきたかというと、当たり判定と関連があるからです。単なる画像の回転ならそりゃ DrawRotaGraph 使っとけってところなんですが、当たり判定も回転させるとなるとそうはいかん。

ちなみに長方形の当たり判定のことを AABB(軸並行バウンディングボックス)というのだけれど、こいつを回転可能にしたものを OBB(回転バウンディングボックス)というんだ。

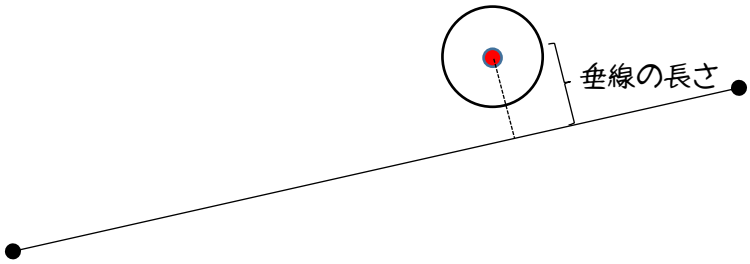
だが、こいつは少々扱い…アルゴリズムが厄介なのだ。本当に厄介。だから今回はもうちょっとだけ簡単で、とっても軽くて扱いやすい「カプセル型」を使用することにした。



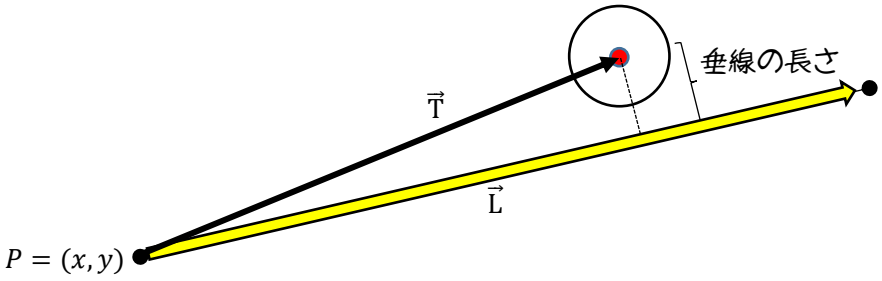
例えば、図のような円と、下に見える線分の当たり判定はどうやってやったらいいのだろうか？ちなみに円の当たり判定はもう大丈夫だよな？それ前提で話すぞ？

線分への最短距離

簡単に言うと「線分への最短距離」を求めればいい。「線分への最短距離」とは垂線の長さですよね？



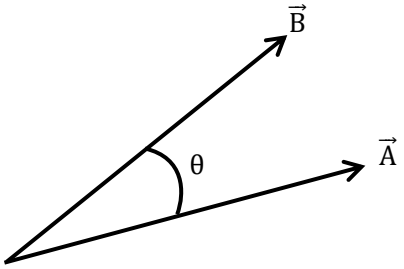
これをどうやって求めましょうか…一応求める際に必要になってくるのは、この線のベクトルと「線の起点、P」から、円の中心点までのベクトルの二つです。
線分のどちらかの端点(どちらでもいい)から、円の中心までのベクトルを$\vec{T}$とし、その線分のその端点からもう一方の端点までのベクトルを$\vec{L}$とします。



ここまではいいかな？

内積とcos

で、もう忘れてる…かもしれませんが、内積とcosθの関係を覚えておられますでしょうか？



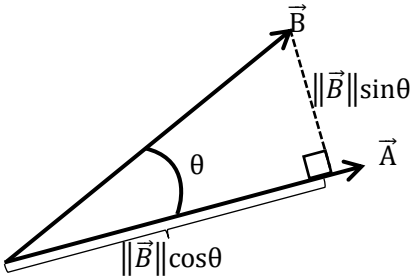
Aベクトルと Bベクトルがこのような位置関係の場合、内積とその間の角度θの関係は

$$\vec{A}\cdot\vec{B} = \|\vec{A}\|\|\vec{B}\|cos\theta$$

となります。$\|\vec{A}\|$はベクトル$\vec{A}$の大きさを表しています。これを式変形して

$$\frac{\vec{A}\cdot\vec{B}}{\|\vec{A}\|\|\vec{B}\|} = cos\theta$$

のようにcosθを求めることもできます。そこでもう一歩進んで Bの A への射影を考えてみよう。Bの端点から A へ垂線を下ろしましょう。



で、この二つのうち内積に関係があるのは Bcos 側ですよね？ということで

$$\frac{\vec{A}\cdot\vec{B}}{\|\vec{A}\|} = \|\vec{B}\|cos\theta$$

とすれば分かるように、内積を A(垂線をおろされた側のベクトル)で割ってやりゃ射影の長さがわかるわけです。ちょっとこれは覚えときましょう。

ちなみに外積という物もあって、演算記号「×」で表します。

$$\vec{A}\times\vec{B} = \|\vec{A}\|\|\vec{B}\|sin\theta$$

まぁ、混乱するだけなので、今回は使わないです。内積だけならともかく外積まで出てきたら

良くわかんないでしょ?(いやホンマは1年でやっとるはずなんで知っててしかるべきなんやけどな?)

ちなみに内積ですが、数式で書くと

$$\vec{A}\cdot\vec{B}=x_Ax_B+y_Ay_B$$

簡単ですね。それぞれの要素同士をかけて足すだけ!!以上!!終わり!!閉廷!!!!

一応なぜこいつが $\cos\theta$ と関係しているのかですが、またまた極座標を考えれば分かりますよね。

まず x_A, y_A, x_B, y_B を極座標で表してみます。

$$x_A=r_Acos\theta_A$$
$$y_A=r_Asin\theta_A$$
$$x_B=r_Bcos\theta_B$$
$$y_B=r_Bsin\theta_B$$

とします。そうすると先ほどの

$$\vec{A}\cdot\vec{B}=x_Ax_B+y_Ay_B$$

が別のものに見えてきませんか…?ほら…

$$\vec{A}\cdot\vec{B}=r_Acos\theta_Ar_Bcos\theta_B+r_Asin\theta_Ar_Bsin\theta_B$$

である。さて、ここでr_Ar_Bが全体にかけられていることに気づくだろうか…つまり

$$\vec{A}\cdot\vec{B}=r_Ar_B(cos\theta_Acos\theta_B+sin\theta_Asin\theta_B)$$

である。さらにかっこの中を見てほしい…cos,cos+sin,sin…何か思いださないかね?そう、cosの加法定理

$$r_Ar_B(cos\theta_Acos\theta_B+sin\theta_Asin\theta_B)=r_Ar_B\cos(\theta_A-\theta_B)$$

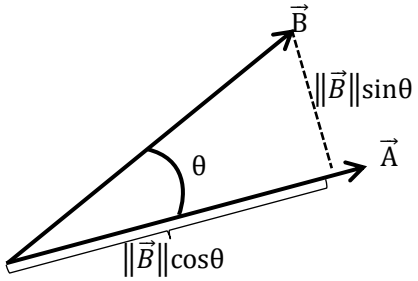
θ_A, θ_B ともにベクトルの向きを表す角度であり、それがマイナスされているということは「間の角度」といえる。つまり

$$r_Ar_B\cos(\theta_A-\theta_B)=r_Ar_B\cos\theta=\|\vec{A}\|\|\vec{B}\|cos\theta$$

というわけで、内積が $\cos\theta$ と関係があるということは…分かっただろう?

さて、こいつを当たり判定として使用するときのやり方だが…、まずは射影長を内積を利用して計算してみましょう。

射影長と最短距離



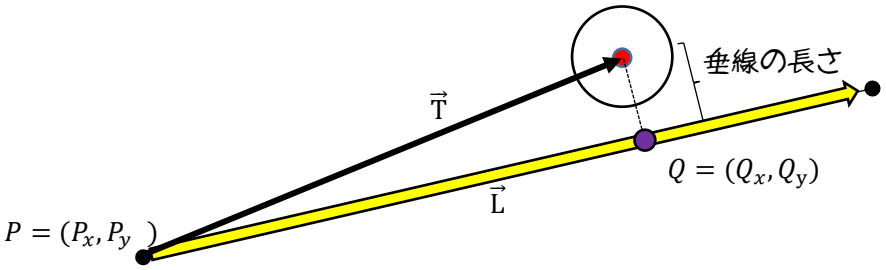
さて、ここまでの話で納得できたかどうかわからないが、ともかく射影の長さは

$$\frac{\vec{A}\cdot\vec{B}}{\|\vec{A}\|}=\|\vec{B}\|cos\theta$$

で計算できる。

ここまではいいだろうか？次は実際に線分と円の最短距離を射影長の長さを利用して図ってみよう。

まず、垂線を下ろしたところの座標を求めよう



垂線を下ろしたところの座標を仮にＱとしよう。このＱを求めたい。

座標Ｐとベクトル$\vec{T}$とベクトル$\vec{L}$を使って求めてみてくれ。どうなるかな？どうなるかな？

わからんのか？この戯けが!!

しようがねえなぁ、俺がいっちょ計算してやっか。

先ほども言った通り、Ｔから Ｌに対する射影長は

$$\frac{\vec{T}\cdot\vec{L}}{\|\vec{L}\|}$$

で求まります。長さはわかったが、方向が分からないとなぁ…でも、よく考えてほしい。方向はそりゃ点 Ｐ から$\vec{L}$の方向にある。だが、長さが邪魔である。長さ(大きさ)が邪魔な時はどうする

んだったかな？

…そう、正規化である。つまり

$$\frac{\vec{\mathrm{L}}}{\|\vec{\mathrm{L}}\|}$$

は「L の方向だけを持ったベクトル情報」となる。方向が分かったらあとは長さをかけてやるだけだ。

$$\frac{\vec{\mathrm{T}}\cdot\vec{\mathrm{L}}}{\|\vec{\mathrm{L}}\|}\frac{\vec{\mathrm{L}}}{\|\vec{\mathrm{L}}\|}$$

さらにこれに起点の P を足してやれば Q を求めることができる。つまり

$$\mathrm{Q}=\mathrm{P}+\frac{(\vec{\mathrm{T}}\cdot\vec{\mathrm{L}})\vec{\mathrm{L}}}{\|\vec{\mathrm{L}}\|^2}$$

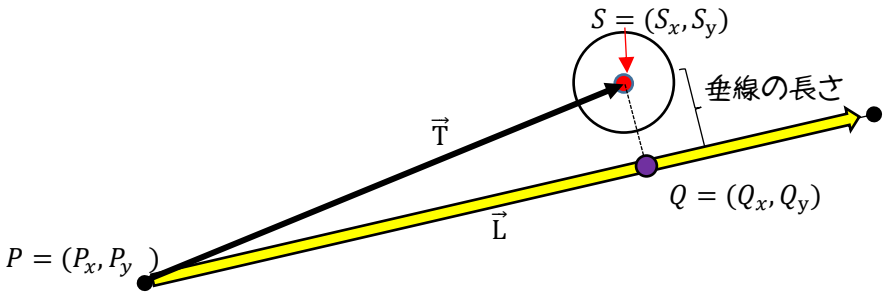
である。

もし、予め$\vec{\mathrm{L}}$が正規化されており、正規化済みの$\vec{\mathrm{L}}$を$\hat{\mathrm{L}}$とおくと

$$\mathrm{Q}=\mathrm{P}+(\vec{\mathrm{T}}\cdot\hat{L})\hat{\mathrm{L}}$$

となる。ずいぶんシンプルになった。このようにベクトルは予め正規化しておくと計算が楽になることが多い。覚えておこう。

さて、最短距離までもう少しだ。



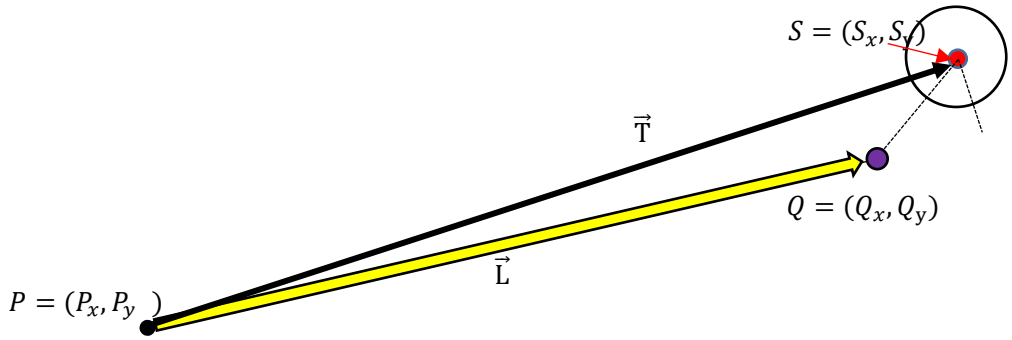
球体の中心座標を S と置くと、あとは S と Q の距離を求めればいい。S は既知であるので

$$\|\mathrm{Q}-\mathrm{S}\|=\sqrt{(Q_x-S_x)^2+\left(Q_y-S_y\right)^2}$$

で最短距離が求まります。さて、ではここまでに必要な武器をプログラムで書いてください。
内積は Dot 関数で作ればいいです。
もう…実装してたかな？ともかく、特定の線分と特定の点の最短距離を求めるプログラムを書いてください。

垂線とは書いたものの…

ちょっと罠があります。それはこういうパターンです。



こんな風に、垂線を下ろしても線分の上に乗っからず、線分の外に来ちゃうパターンです。こういう場合は線分の端点がQになります。で、どの式が役に立つのかというとこれ

$$\frac{(\vec{\mathrm{T}}\cdot\vec{\mathrm{L}})}{\|\vec{\mathrm{L}}\|^2}$$

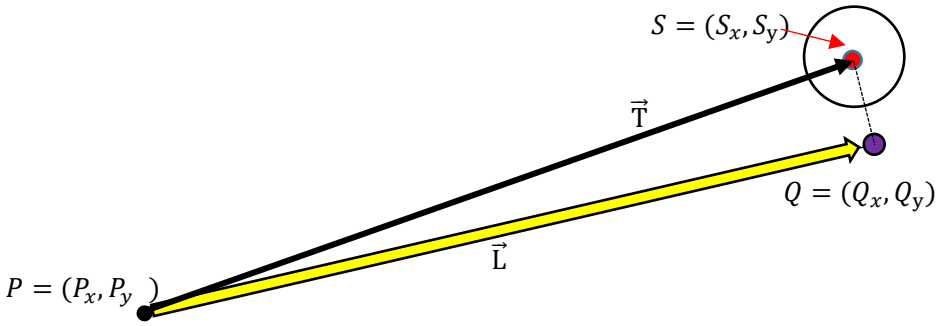
え？なんでこれが役に立つの？って思うかもしれませんが、この式、よ～～～く考えてください。役に立つんですが…難しいかな～。

うーん。さっきも話したように垂線下した部分への射影が

$$\frac{(\vec{\mathrm{T}}\cdot\vec{\mathrm{L}})}{\|\vec{\mathrm{L}}\|}$$

ですよね？

ここはご理解いただける？で、例えば射影が線分の長さ$\|\vec{\mathrm{L}}\|$と全く同じとき、どうなるだろう？



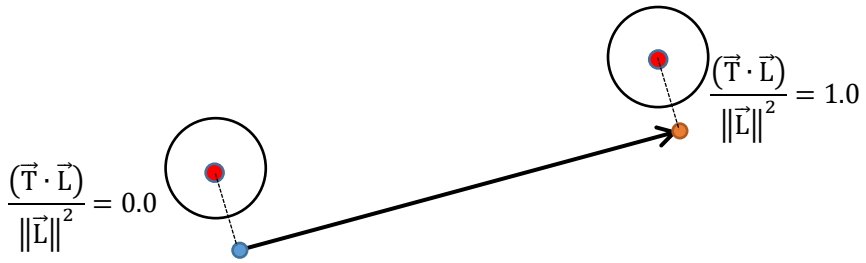
そう、その時は$\frac{(\vec{T}\cdot\vec{L})}{\|\vec{L}\|}=\|\vec{L}\|$なのだから

$$\frac{(\vec{T}\cdot\vec{L})}{\|\vec{L}\|^2}=\frac{(\vec{T}\cdot\vec{L})}{\|\vec{L}\|}\frac{1}{\|\vec{L}\|}=\frac{\frac{(\vec{T}\cdot\vec{L})}{\|\vec{L}\|}=\|\vec{L}\|}{\|\vec{L}\|}=\frac{\|\vec{L}\|}{\|\vec{L}\|}=1$$

となりますよね？
つまり、射影が線分の上に乗っかってる限り

$$0.0\leq\frac{(\vec{T}\cdot\vec{L})}{\|\vec{L}\|^2}\leq 1.0$$

が成り立つわけです。



では、射影が線分の外に出てしまった場合はどうすればいいのか？
それは射影長が 0 未満だったら 0 に、1 を超えたら 1 に補正してください。

ここはプログラム的に書くとわかりやすいかと思うので、プログラム的に書きます。すると

```
float t=Dot(T,L)/Dot(L,L);//(T・L)/(L^2);
t=min(max(0.0f,t),1.0f);
```

になります。おわかりいただけたであろうか…？ちなみに Dot は内積関数で、2つのベクトルを受け取ったら内積値を返す関数です。自分で作ってね。掛け算と足し算しかないから簡単ダルルオ!?

あと、min とか max は、二つの引数を受け取り、それぞれ小さいほうと大きいほうを返す関数です。即ち 0 と 1 の範囲内に補正しているということです。

さて、この t から射影点、もしくは端点、を求めるわけですが、そこは簡単です。点は必ず元の L ベクトル上にあるわけですから、この t を L にかけてやるだけでいいのです。

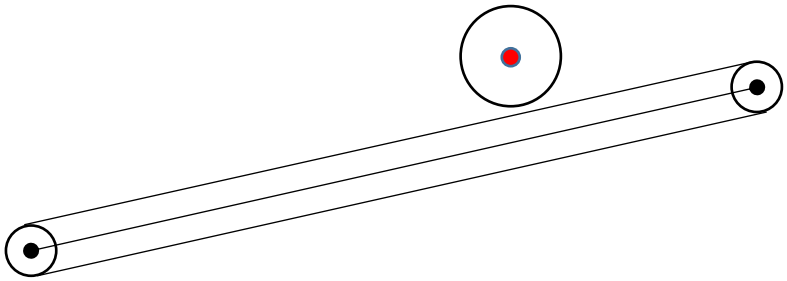
つまり L*t もしくは L.Scale(t)で、そのベクトルが求まります。なお、座標はそれに線分の始点を足してやればいいので、

Position2f pos=posa+L*t;

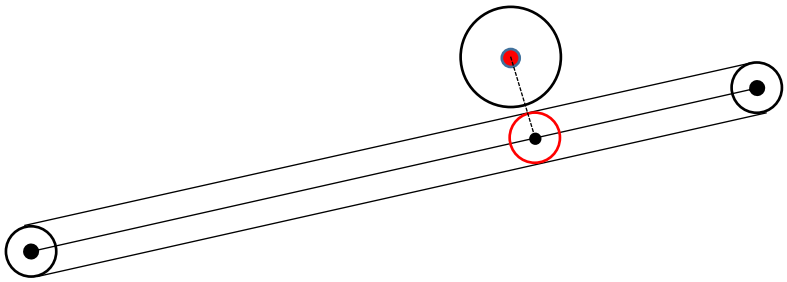
で補正済み射影点が求まります。

補正済み射影点と球体との距離を測り、当たり判定に利用する

さて、すでに補正済み射影点が出てますので、あとは球体(円)の中心点との距離を測り、その距離が半径と「線分の幅」を足したものよりも大きければ当たってない、小さければ当たっているということにします。



理屈自体は球体(円)の当たり判定と同じです。測る座標が違うだけです。先ほど計算した補正済み射影点、pos と円の中心例えば、circle.pos との距離を測ればいいのです。垂線を下ろした場所に円があると思えばいいのです。



そう考えると、簡単でしょう。すでに作った円どうしの当たり判定を適用できます。

実装

準備

まず、内積の関数を作ってください。内積の計算は「かけて足す」だけです。Geometry.cpp の中に Dot という関数があるので、それを実装してください。二つのベクトルのそれぞれの要素同士をかけて足せばいいんです。

内積が実装出来たら、さっきちょろっと書いた

```
float t = Dot(T,L)/Dot(L,L);//(T・L)/(L^2);
t = min(max(0.0f,t),1.0f);
```

を用いれば射影長を求められます。

あ、準備としてもちろん、円とカプセルの構造体は必要ですね。

```
//円
struct Circle {
	float radius;//半径
	Position2 pos; //中心座標
	Circle() :radius(0), pos(0, 0) {}
	Circle(float r, const Position2& p) :radius(r), pos(p) {}
};

//カプセル
struct Capsule {
	float radius;//半径
	Position2 posA; //端点A
	Position2 posB; //端点B
	Capsule() :radius(0), posA(0, 0), posB(0, 0) {}
	Capsule(float r, const Position2& A, const Position2&B) {}
};
```

これはもう用意しておきましたので、あとの必要なものは自分で用意してください。ここまできたらそれくらいできるでしょ。

要件

さて、ここから何を作ってほしいのかといいますと、今までの

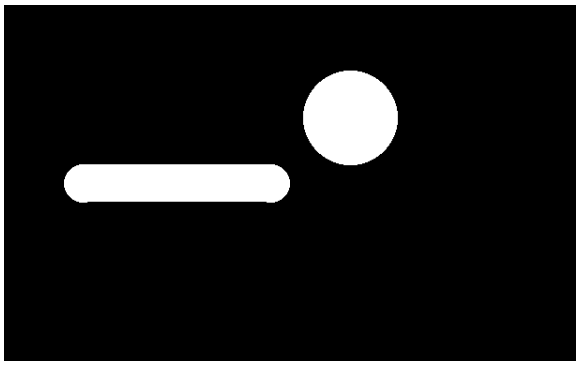
- 円の当たり判定
- 回転行列による回転
- 線分と円の距離

をもとに、カプセル&円の当たり判定をとっていただきたいと思います。
では要件です。

- 円は上下左右カーソルキーで移動すること
- カプセルは画面上をマウス押下で回転すること
- 通常時は円とカプセルは白色で描画すること

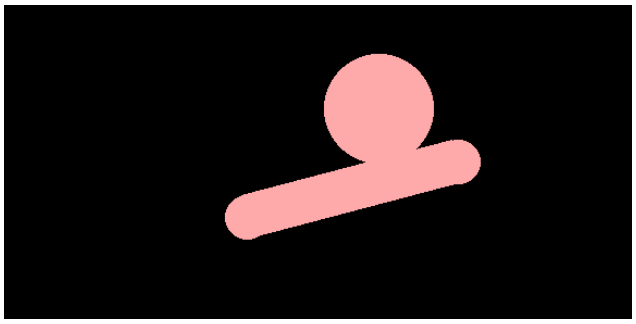
- 衝突時は円とカプセル両方ともに色を変更すること

カプセルは一応 DrawCapsule 関数を用意してますので、それを使ってもいいです。



当たってないとき～

そして…



当たったとき～

といった具合にしてみてください。

さぁ、れっつら GO

余裕のある人は円を増やしたり自動で動かしたりして、ゲームっぽくしてください。

課題その②

ちらちら見えてたと思いますが

これ。課題に使うものです。

サーバに「丸太はこびげー」があるので落として CarryLog.zip 解凍してください。で、表示するとマリオとルイージが丸太をもっていますが、これを回転させてゴールに導くゲームを作り

ます。

最終的に一番上にたどり着けばゴールですが、岩が降ってきたり、横から何か来たりするような感じで邪魔をさせてください。当たったら、何かしらアクションを起こしてください。

丸太ゲー要件

1. 回転させながら上に上がっていく(回転させすぎてはダメ)
2. 一番上に到達するとゲームクリア
3. マリオとルイージが垂直以上になると下に落ちてゲームオーバー
   (難しくなるので、これは無くてもいいかも…)
4. 上から岩が降ってくるので避ける
5. 左右から岩が飛んでくるので避ける
   (これもなくていいかも)

を作ってください。

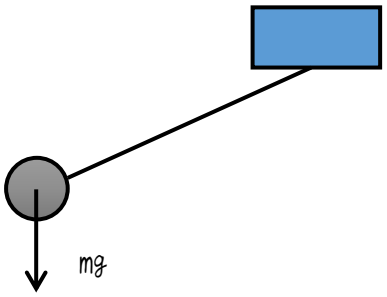
提出要件:

- 提出期限は 8/21(金)18:00 までに提出する事(いかなる理由があっても提出しないと単位ないので、できるだけ事前に提出してください。今日のうちに出しとくといいです。)
- 提出場所
  ¥¥stfs¥APC_ABCC クリエイティブ¥gakuseigame¥通常授業以降¥川野¥数学課題提出先¥丸太運びゲーム
- サーバでダブルクリックして動くこと
- 丸太の回転と当たり判定が実装されていること
- **タイトルバーに学籍番号と氏名を書いておくこと**
- 動作に必要な Exe とリソースファイル(画像と音)のみ提出し、ソースコードやプロジェクトはアップロードしないこと

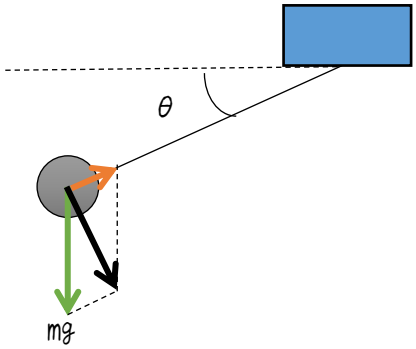
以上です。

振り子運動

初期位置が下図のようになっているとします。



当然重力が mg かかっているんですが、紐に繋がれているので「張力」ってのが発生します。「反作用」みたいなもんですけど、そのまま g に従って落ちると紐の長さが保てなくなるので、**紐の付け根方向に物体を戻そうとする力…それが張力**です。つまるところ作用反作用です。

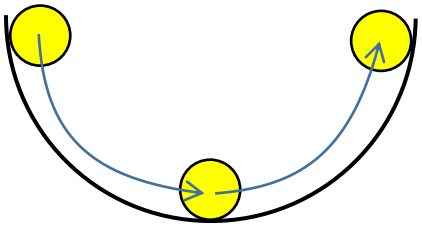


結局、図のようになり、結果として「紐の長さを中心とした円弧運動」を行います。そして加速度は g がかかるべきところが $g\cos\theta$ となります。

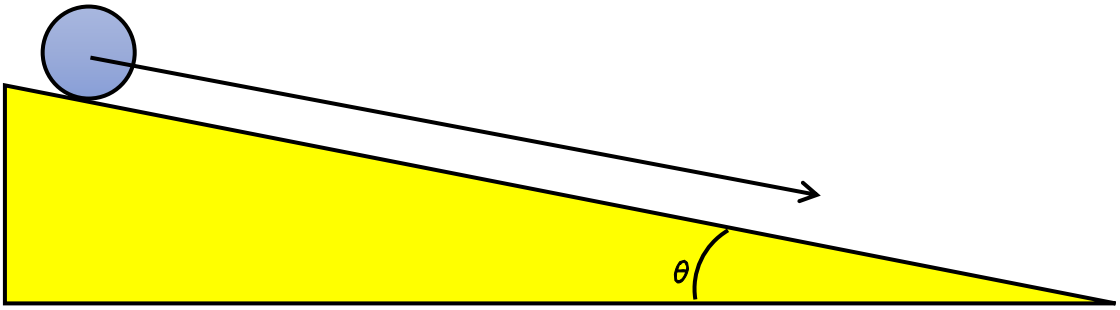
物体が支点から鉛直方向(つまりひもが縦に真っ直ぐの状態)にあるとき、加速度が 0 になります。なぜなら $\cos 90° = 0$ だから。

そして、最下点を通過すると今度は上に上がっていきますが、それは一度貯めこんでおいた速度が重力で少しずつ削られてまたゼロになっていくわけです。

なんか分かりづらいなーって人はお椀の中をボールが動いてる状況を考えてください。もしくは「ハーフパイプ」の中の状況を考えてください。



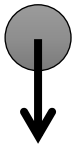
こうだとします。
そうすると結局は「連続して斜面の角度が変わる坂」にいるようなものです。ここで坂における球体の動きについて考えてみましょう。



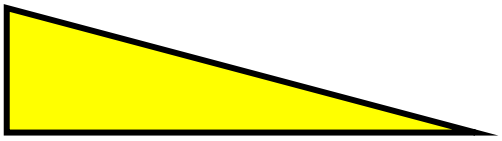
さて、斜面にボールを置くと、当然下に向かって転がりますね？それは重力がかかっているからです。**重力のみ**でこの坂を下る場合はどれくらいのスピードで下るんでしょうね？

最初は停止からゆっくり転がり、段々と早くなっていくことは想像がつくでしょう？それは大丈夫かな？ということは「加速」しているわけだ。加速しているということは何らかの「力」がかかっています。

この場合、その力は「重力」ですね？ニュートン氏に対し異論はないしいいね？
坂道を降りる時の速度加速度についてですが、確かに重力が推進力のもとになっていますが、普通重力は真下に向かってかかるもの…様子がおかしいですよね？



物理的な力としては「下に降りたい」わけで、重力の方向は真下向きなわけです。
ところが、障害物がありますよね？

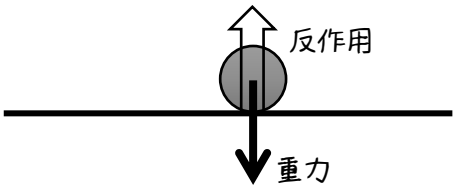


坂道です。そう、坂。

常識的にというか、経験的に考えて、動く物体は「動かない」障害物(壁、地面、床、その他)にあたったら停止しますよね？

これを物理学では「作用反作用の法則」という考え方で対応しています。これはなんとなくごぞんじですよね？

例えば地面に対してであれば、常に重力がかかっているのに皆さんが地面にめり

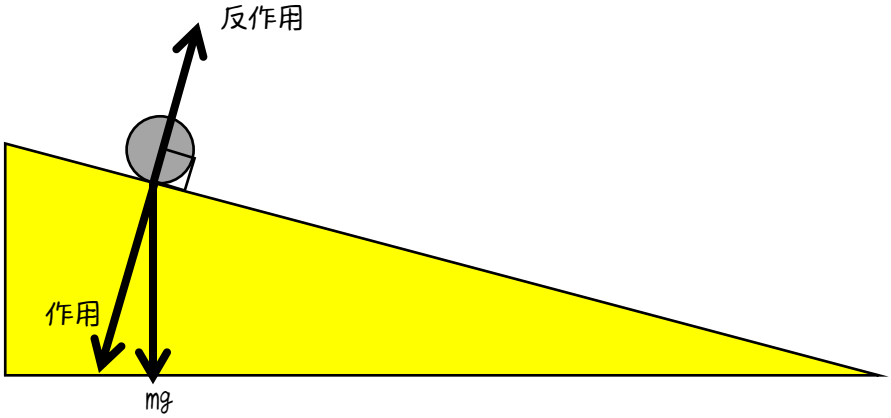


込まないのは反作用がかかっているからです。
多分これを初めて聞いたときは「なにそれ？そんな力感じませんけどwww」ってなったかもしれないけど、要は

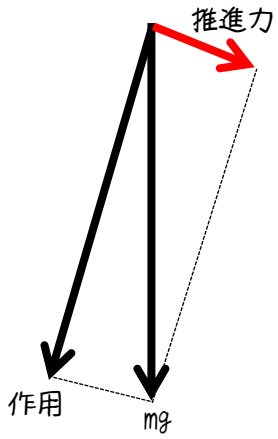
「力をかけているのに動かない(加速しない)」ということはその動かないという事実によって「かけている力をまるまる打ち消す反対方向の力がかかっている」と言うことがわかるわけです。

これがなければあなたがたは地の底深くまでズドーンです。だってF=maなんですから、力がかかっている以上加速しちゃうんです。本来。重力によって mg がみんなの体と地面の接地面にかかってるんです。でも落ちないということは床からmgの反作用を食らっているということです。

さて、このことを踏まえたうえで坂道のボールについて考えてみよう。重力は等しくかかるので、下向きに mg の力がかかっている。ちなみに「面における反作用」は面に垂直な方向にかかる(面に垂直な方向にしかかからない)。これは何となくイメージできるだろう？



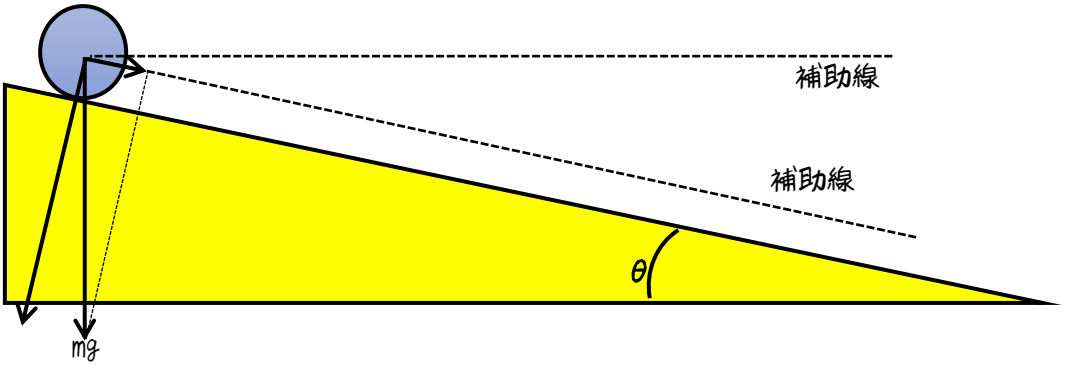
で、重力と反作用が同じ方向を向いていない以上、ある特定の方向の力が残ってしまう。
<u>この力が推進力となるのだ</u>。ちなみに坂道の例以外にも「反作用によって力が奪われて残った分が推進力になる」例は結構ある。ビリヤードの衝突なんかもそう。じゃあ実際の推進力はどれくらいになるのかって話だけど、それは「打ち消した力ベクトル」をもとのベクトル「重力」から引けばいい。面に垂直な成分はすべて奪われているのだから



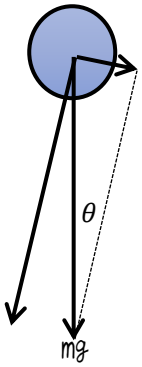
まぁ、これ逆に言うと、重力の「面に垂直な方向成分(面への作用)」と「推進力」をベクトル的に足し算をすると元の mg になります。

ということで、mg というベクトルを「作用」と「推進力」のように2つのベクトルにしてしまうことを「ベクトルの分解」といいます。

目的がなければ「ベクトルの分解」なんて無数に解が出てきてしまうのですが、今回は作用方向が「斜面に垂直」であることが分かっています。ここからよく考えてほしい。



ハァ…ハァ…どうよ!!
さぁこっちも苦労して補助線を引いてやったんだから(Word では意外とこういう図を描くのが難しい)、てめーらも頭と手を使って、反作用と推進力を考えてみてくれ…必要なのは推進力のみだがな!!



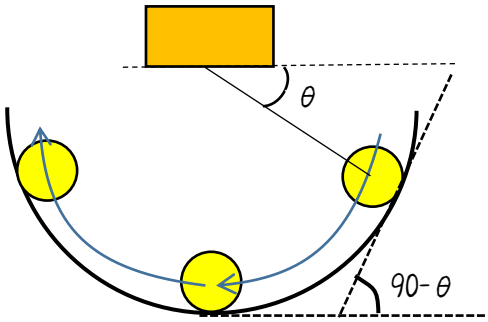
あ と は わ か る な?

よくわからない!?まま、ええやろ。良くわからない人のためにプログラムで練習できるようなコードを書いてみた。

それを頑張って実装すれば「ムズくて物理が分からないわ」に対する「これでわかるようになったろう」になるんじゃないかなと思います。

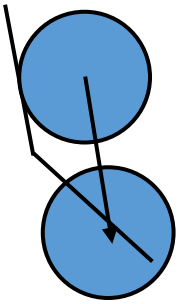
順序としては、
① 現在の床の傾きから角度を計算(atan,atan2)
② 角度をもとに加速度を計算
③ 速度に加速度を加算
④ 加算後の速度をX方向Y方向に分解
⑤ ④を座標に加算
となります。

そうだよ。垂直抗力だよ。つまり振り子だのなんだの難しいことのように考えガチだけど所詮はお椀のように連続して傾きが変わる斜面の内部で動いているのと同じなのだ



まぁ、別にどちらの角度を基準にしてもいいんですが、上の図のθであれば$g\cos\theta$もしくは$g\sin(90-\theta)$が加速度になります。
角度自体は分かったんですが、じゃあこの加速度を振り子ブロックに与えればうまくいくのでしょうか？実は微妙にうまくいきません。

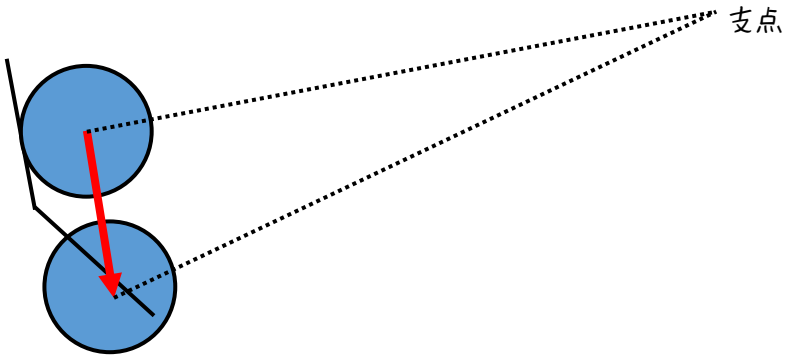
何故かというと「現在の状況」から加速して進むんですが、そのまま進めちゃダメなんです。ものごっついい拡大してみると



こういう風になっています。
数学をある程度分かってる人ならわかると思うんですが、これを繰り返すと面倒なことに

なる…のが分かりますか？

加速度は紐と垂直な方向に発生します。と、いうことは…？
御覧のようにピタゴラスの定理により、進むほど紐が長くなってしまいます。



結局、図のようになり、結果として「紐の長さを中心とした円弧運動」を行います。
そして加速度は g がかかるべきところが $g\cos\theta$ となります。

不思議ですねえ～。まぁ、簡単に言うと素直にやってしまうとドゥンドゥン長くなります。どういうことかというと、元の紐のベクトル(支点→重り)を$\vec{L}$としますよ？推進力は張力と直行する方向になりますので、推進力による変位を$\vec{M}$とすると

$$\vec{L}^2 + \vec{M}^2 > \vec{L}^2$$

は明らかですので、どんどんどんどん長くなります。これでは使い物になりません。
ということで補正します。

$$P' = \frac{\vec{L}+\vec{M}}{\|\vec{L}+\vec{M}\|}\|L\| + S$$

ただし S は支点座標。$\vec{L} = P - S$ で P' は補正後の座標です。ああ、ちなみに M 自体は

$$\vec{M} = \vec{G} - \frac{\vec{G}\cdot\vec{L}}{\|\vec{L}\|}\frac{\vec{L}}{\|\vec{L}\|}$$

で定義されるもので、今重りにかかっているベクトルです。

ひとまずはお椀型で練習しましょう。

そしたらだいたいわかってくるかなと思います。

それが終わったらいよいよ最後の課題です。

課題その③

サーバの振り子ゲームというフォルダにあるOJISWINGというのを解凍して、きちんと振り子運動をして、振り子の紐を切り離して、とおくに飛ばせるようにしてください。

提出期限はだいたい~~8/2(金)くらいとします~~。提出規約はいつもどおりです。

- 提出期限は今のところ未定です…授業でここまで来る頃には決まってるでしょう
- サーバでダブルクリックして動くこと
- とにかく振り子運動はやっておくこと
- **タイトルバーに学籍番号と氏名を書いておくこ**と
- 動作に必要なExeとリソースファイル(画像と音)のみ提出し、ソースコードやプロジェクトはアップロードしないこと