

2年生の数学(数学応用)

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 も軽しくは触れると思いますが、まあどうせ後期もあるしええんちゃうかな。

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 ゲームに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` があるんだよなあ…)
- ラジアン(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わりを持つんだ。重要★)
 - 外積(これは 3D かな…? まあ 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(まあ A 地点と B 地点の中間地点がどこかってやつね)
- ベジエ曲線(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やねーかな。まあ別に確率統計やらんでもガチャの確率とかわかるじやろ? 時間余ったらやろうかなあ…。だいたいこんな感じのをを使って、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。

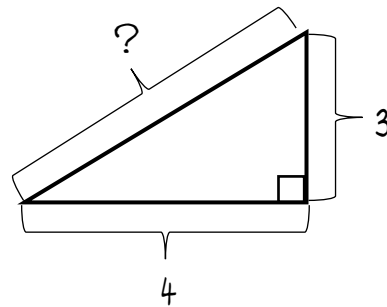
確認テスト

(‘ω’)ノ ヤ、ようこそみんなが嫌がる数学の問題へ。この問題はサービス問題だからまずは落ちついて取り組んでみてほしい。別にこの問題で成績をつけようなんてケチな事は思っていない。ぼくの評価基準はプログラミングできてるかどうか、それだけだ。

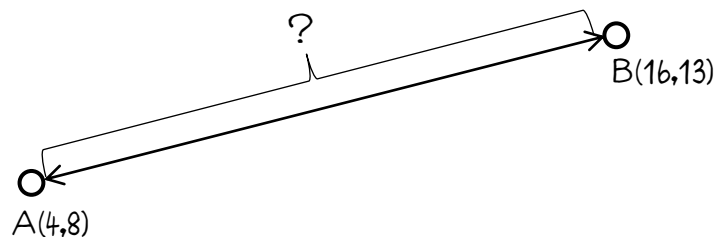
でもいくつか問題を解いてみたとき、君は、きつと言葉では言い表せない『ときめき』みたいなものを微塵も感じないとは思ってたが、この殺伐とした世の中でたまには黙々と数学やるって時間があってもいい。そう思ってこの問題を作ったんだ。じゃあ、解いてもらおうか。

新2年生のための…おさらい問題①

- (1) はい、三平方(ピタゴラス)の定理です。底辺が4で高さが3の直角三角形があります。これの斜辺の長さを求めましょう。単位はいらないです。



- (2) 座標平面上の2点 $A(4,8)$ と $B(16,13)$ の直線距離を求めましょう。これまた単位は不要!

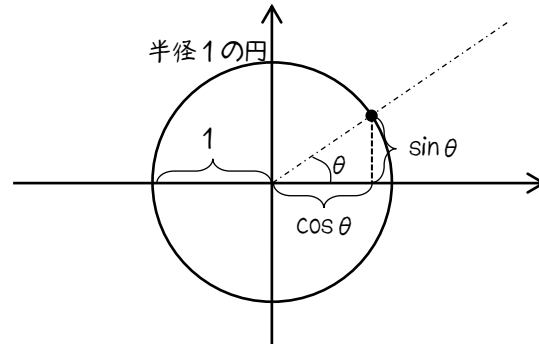


- (3) 度数法(degree)で 30° の角度は弧度法(radian)では何分の何 π かな?

- (4) 弧度法(radian)で $2\pi/3$ は度数法(degree)では何 $^\circ$ になるかな?

- (5) 度数法で 315° は弧度法では何分の何 π になるかな?

三角関数の基本は『円』です。三角なのに円とか話が違うじゃないかい!! 1加減にしる! と言いたいんですが、まあ半径1の円を基準にしつつ、そこにできる直角三角形のそれぞれの



辺の長さを規定したものだから…ま、多少はね。

上の図のように半径1の円に対して、円の中心からx軸から角度 θ だけ回転した向きに直線を伸ばす。そして交点を求める。その交点のx座標が $\cos \theta$ でy座標が $\sin \theta$ と規定する。これが基本の考え方ですね。あとは『相似と比率』について理解できれば三角関数は大丈夫です。最低でもどっちがsinでどっちがcosかは覚えておきましょう。

(b) それを前提に…問題です。

$\sin(45^\circ)$ と $\cos(45^\circ)$ を求めましょう。

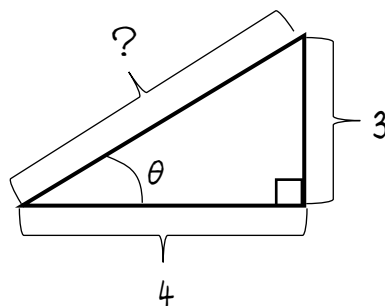
(7) $\sin(30^\circ)$ と $\cos(30^\circ)$ を求めましょう。

(8) $\sin(90^\circ)$ と $\cos(90^\circ)$ を求めましょう。

(9) $\sin(120^\circ)$ と $\cos(120^\circ)$ を求めましょう。

(10) $\sin(210^\circ)$ と $\cos(210^\circ)$ を求めましょう。

(11) 以下の図形の $\sin \theta$ および $\cos \theta$ を求めましょう。



(12) ベクトル $A(2,3)$ とベクトル $B(3,1)$ の内積を求め、AとBの間の角度を θ としたときの $\cos \theta$ を求めましょう。

新2年生のための…おさらし問題②

(1)x 方向に 4, y 方向に 5, z 方向に 6 移動する 4×4 行列を書きなさい(左の空白ね)

$$\begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+4 \\ y+5 \\ z+6 \\ 1 \end{pmatrix}$$

(2)二次元において 180° 回転の後、x 方向に 5, y 方向に 10 移動する 3×3 行列を書きなさい

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

(3)時速 36km は秒速何メートルですか？

(4)時速 4km の A ちゃんと時速 8km の B さんが、12km 離れた場所から同時に向かい合って進むとき、2 人が出会うのは何分後ですか？

(5) 時速 36km で走っている車が、ブレーキを踏んだら 5 秒で停止できました。ブレーキを踏んでいる間の加速度は何 m/s^2 ですか？

※これ以降の問題における重力加速度は 10m/s^2 とします。また、空気抵抗などは無視します。
解答にルート(平方根)が混ざっても構いません。そのまま解答してください。

(6)A くんがジャンプします。ジャンプした瞬間の上向き速度が 30m/s だとするとジャンプ最高点は地面から何 m の位置にありますか？

(7)A くんが前斜め 30° でジャンプします。ジャンプした瞬間の速度が 30m/s だとすると A くんの着地点は、ジャンプ地点から前方向に何 m 離れていますか？

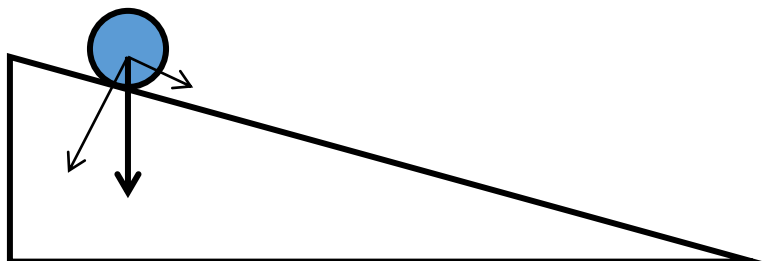
(8)点 A(80m, 220m, 320m) から、点 B(200m, 100m, 200m) へ 4 秒で変位したときの速度ベクトルを答えなさい

(9) 二次元平面上において速度ベクトル $(30\text{m/s}, 40\text{m/s})$ で動いていたものが 5 秒後に速度ベクトルが $(-40\text{m/s}, 30\text{m/s})$ になった。この時の『加速度』をベクトルを用いて表そう。

(10) 原点 $(0,0,0)$ から $(10\text{ m/s}, 30\text{ m/s}, 15\text{ m/s})$ のベクトルでカラーボールを射出します。着地点座標を求めなさい(※カラーボールは重力の影響を受けます)。

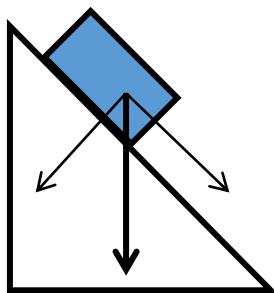
以下の図のようにボールが坂を転がる運動を考えてください。

坂の角度を 30° とします。坂は無限に続いているとします



(11) ボールが坂を転がる時の加速度は何 m/s^2 ですか？

(12) 最初に坂にボールを静止させた状態から始めた場合、坂を転がり始めてから 5 秒後のスピードを求めましょう(摩擦はないものとします)



次は図のように、 45° の斜面の上に四角い板を置くとする。

(13) 坂と板の間の動摩擦係数を 0.4 とし、板の重さを 10Kg だとすると、板が斜面を滑り落ちる加速度は何 m/s^2 ですか？

(14) とある滑り台において、最下点からの高さが 45m の場所から滑り下りました。滑り台との間には摩擦もなく、何かから押されることもなく、重力の力だけで滑り降りるとすると、最下点でのスピードは何 m/s になりますか？

(15) 原点 $(0,0)$ から座標 $(120,90)$ までの直線で定義された壁がある。この壁に対してベクトル $\vec{v}=(30,10)$ の速度で衝突した物体が反射した。反射されたベクトルを求めましょう。

実践編

ここからは実践編です。実際のゲーム的なところで役に立つものをしていきます。まずは手始めに簡単そうなところから…

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。

あつ…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

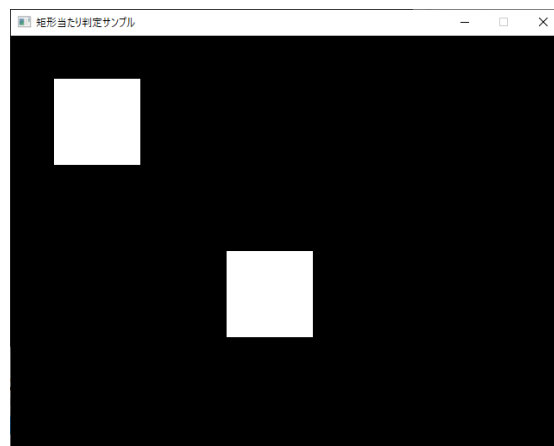
```
struct Rect{  
    int x, y, w, h;  
};
```

```
struct Circle{  
    int x,y;  
    float r;  
}
```

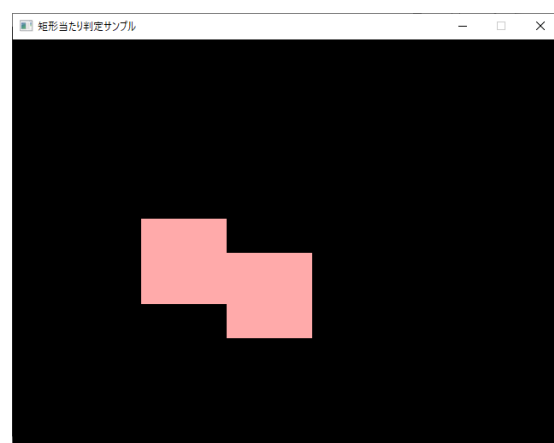
という風に定義します。もし余裕があるならこれらの構造体は Geometry.h というヘッダを作って、そこに置きましょう。

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ろ。

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;

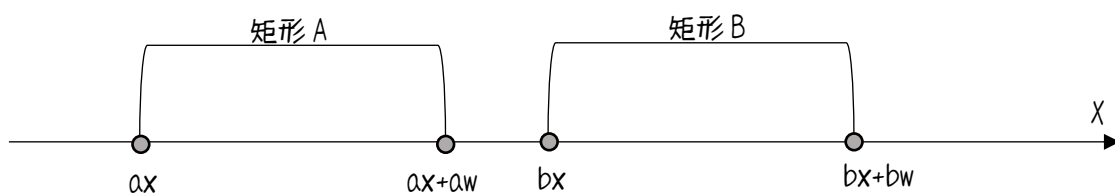
    int color = 0xffffffff; //すっげえ白くなってる。はっきりわかんだね
    if (IsHit(rcA, rcB)) {
        color = 0xffaaaa; //赤くなってんぜ…
    }
    rcA.Draw(color);
    rcB.Draw(color);
    ScreenFlip();
}

```

ま、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…？分かりませえん？

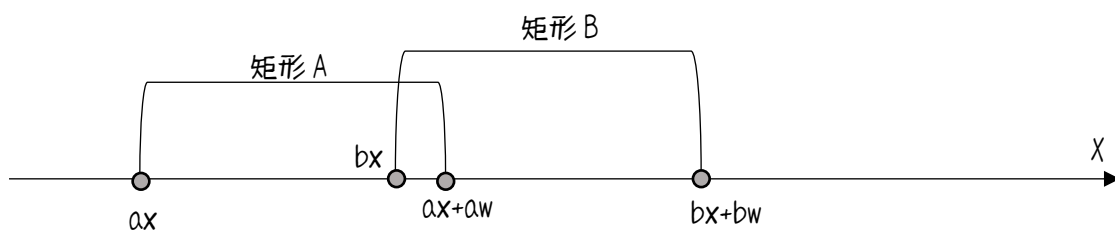
矩形の当たり判定…というか AABB(AxisAlignedBoundingBox)はまず「軸ごとに分けて考える」が鉄則でございます。軸ごとに分けられればあとは数直線…算数の世界。非常に簡単でリーズナブル。良心的アルゴリズムでございます。

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



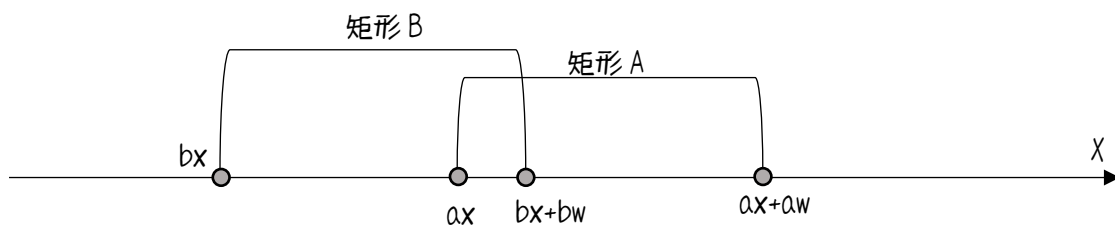
これは…当たってないよね？

という事は「当たっている」とは



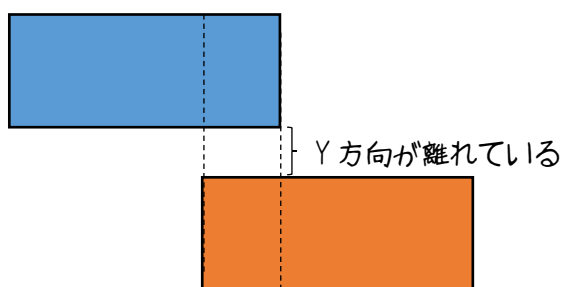
この図のように $bx \leq ax+aw$ になっている①

それか、もしくは



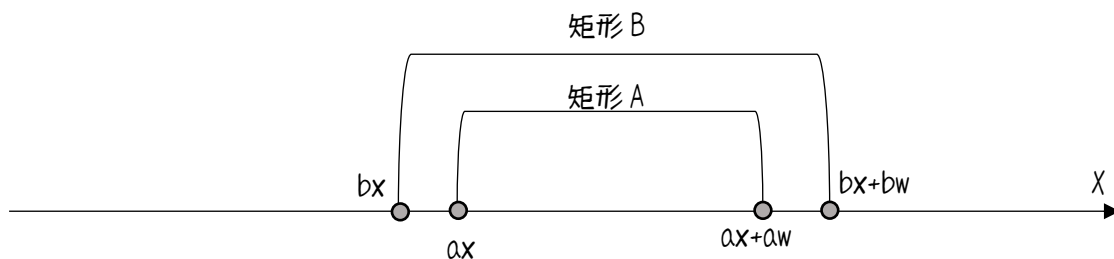
この図のように $ax \leq bx+bw$ になっている②

…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、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



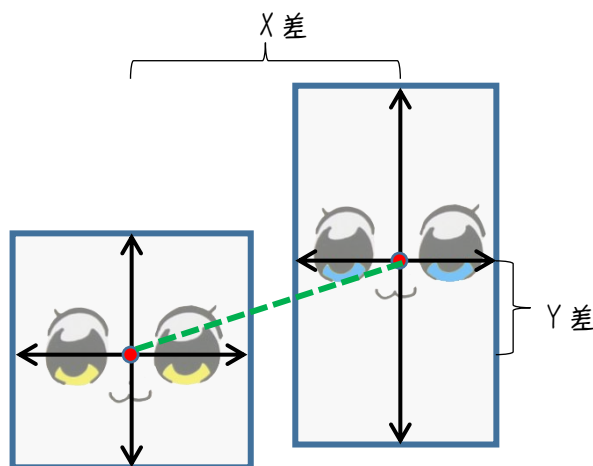
こういう場合もあったりしてこれもうわかんねえなってなりそうなんだけど、
よく考えて『当たってない』条件に注目するともうちょっとシンプルに考えられますね。当たってない当然の条件とは、片方の右よりもう片方の左が右にある時つまり

$\max(ax+aw, bx+bw) < \min(ax, bx)$

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

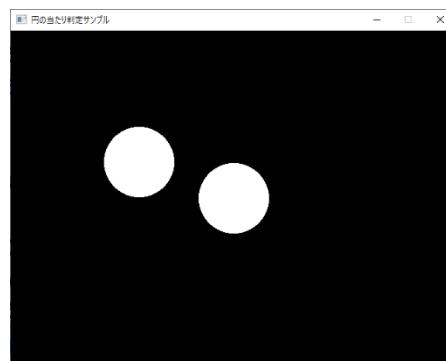


このような位置関係になるため、円の当たり判定の時のように『中心点間の距離』より幅および高さが離れているかどうかで判定でき、より概念的にわかりやすいと思います。

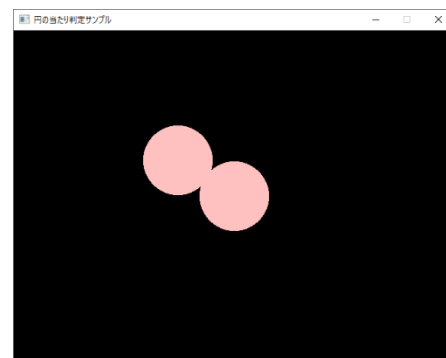
`return abs(bx-ax) < aw+bw && abs(by-ay) < ah+bh;`

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

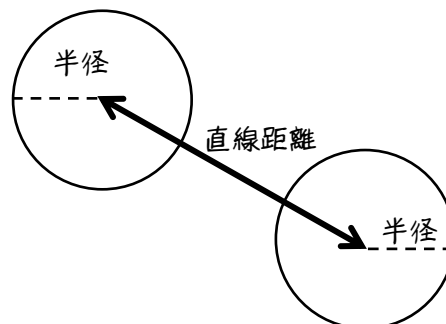


あたってない



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、特に気にしないでいいでしょう。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、それ知ってさえいけばすぐに終わると思います。

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。

っさあ〜頑張ろう!!!

ああ、くつつつそ余裕がある人は、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ろですが、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ツノ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。

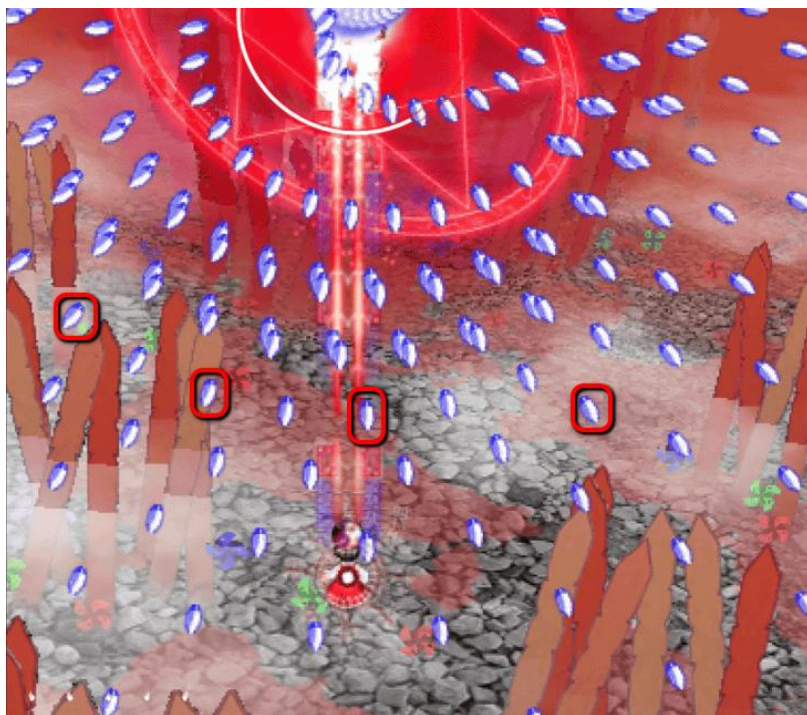
どっちが cos でどっちが sin かだけ区別がつけば十分です。逆に言うところの区別だけはつけてください…ほんとに困るんで…どっちか覚えれば他は自動的にわかるレベルなんだからさ～覚えてくれよな～頼むよ～。

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。

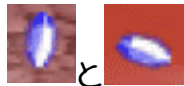
例えばシューティングなどで



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？

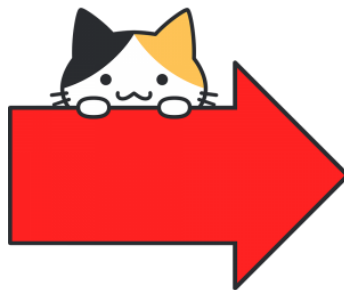
専門的なことはともかく…回転しているん DA

分かりやすい例だと以下の例ですが

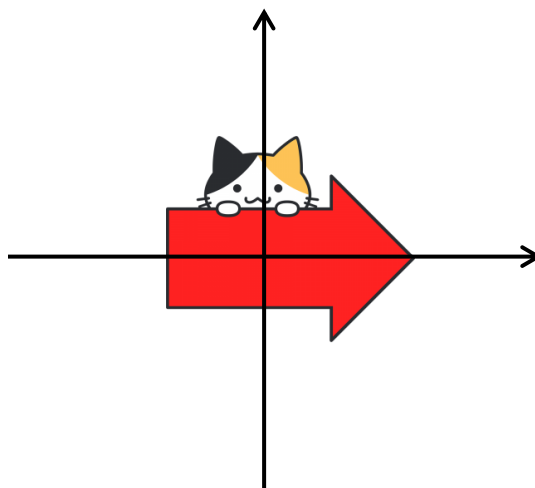


と は明らかに角度が違いますね？つまり画像が回転している。DxLib において画像を回転させるには DrawRotatGraph を使用するのですが、どれくらい回転させればいいのか…。

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か。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角…例えば原点(0,0)から見て(3,4)の方角に向きたいとします。

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 ってところです。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。



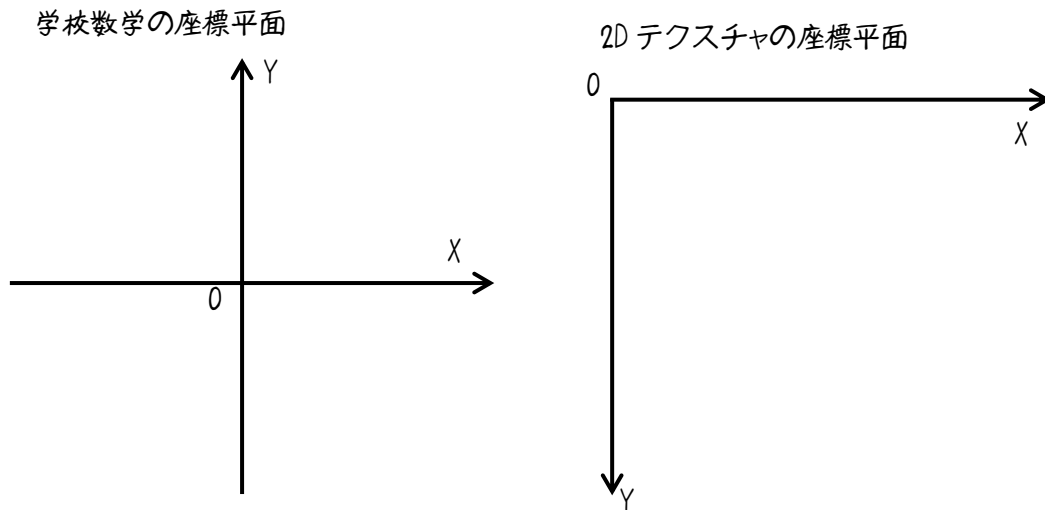
あれ？(3,4)って上向きじゃないの？

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、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ックスにおける 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。

ホントに面倒なんですけど、『そういうルール』故…致し方なし



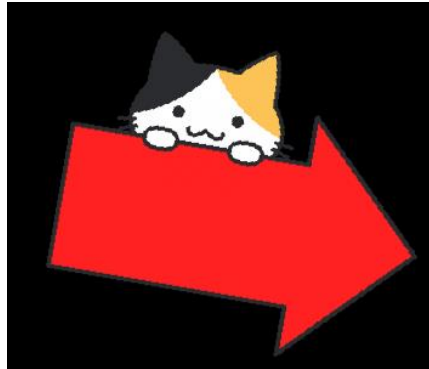
Y 軸方向だけ、向きが違う。もうちょっとというと原点の座標も違う…。というわけで向きが反対になるのです。しかし、安心するがよい。どの道、 atan2 に代入する数値がそもそも 2D テクスチャ座標平面であるから、その値をそのまま使用して RotaGraph で回転させてもきちんと辻褄は合うのだ。

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。そうすると向きは『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ことがあります。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、しゃあない…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 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)。さあ、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って思います。いわゆるベクトルの話なんですけど、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。まだオペレーター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)という関数を構造体内に書きちゃいます。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。

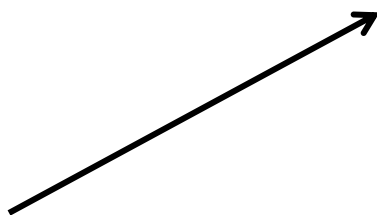
```
float Magnitude(const Vector2& v){
    //ベクトル v の大きさを返してください
}
```

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、どうやって返すかわからねえんだけど？

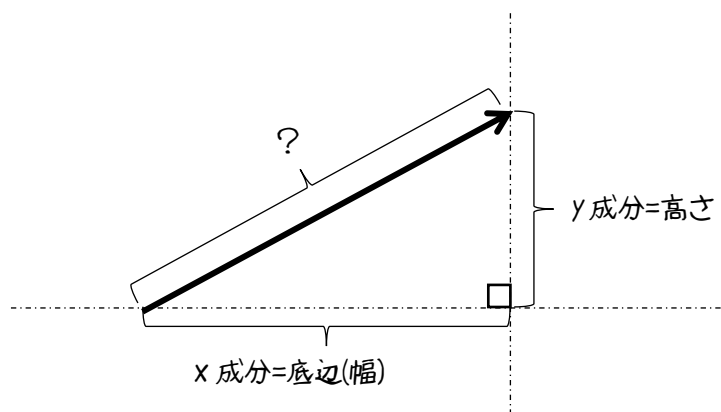
え？習ってない？習ってない？本当に？それホントに M センサーの前で言えんの？

ま、ええわ、もっかい説明したろ。

ベクトルってのはさ、まあ、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いるので、簡単っすねえ〜。ベクトルを 1 にするっていうことは、ベクトルの大きさを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();
};
```

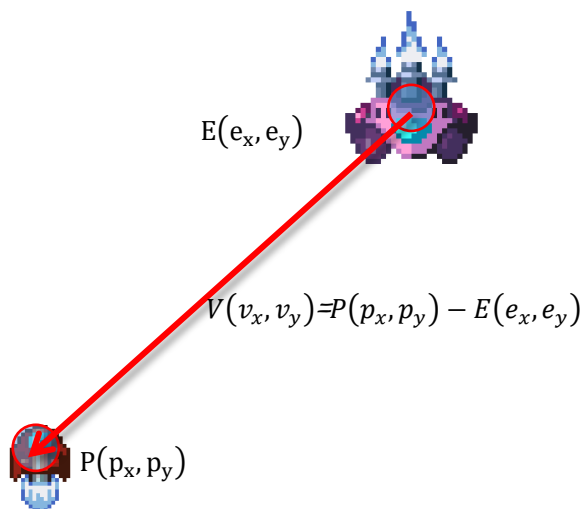
てな感じであとは適当に実装してもらえればいいです。もし本体を正規化しなくて、正規化済みのベクトルを返したければ Normalized って関数を用意して

```
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、作っといってください。

なんで正規化なんて必要なのかというと、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ことができます。

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。

そうすると、座標Eから出発した弾は、次の瞬間には…

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

となってしまう、1 フレームあれば着弾してしまいます。流石にそんな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。そうす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てください。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、赤い部分を自前で実装しておいてください。期限はえーつと6月末くらいかな。でもさっさと作る事をお勧めしておきます。頑張ろう。

ガワは

[¥¥stfs¥APC_ABCC](#) クリエイティブ¥gakuseigame¥||野部屋¥数学¥弾幕

の中にZIP置いてますのでそれを利用してください。自分で1から作れる人は1から作って構いません。

あつ。提出期限を6/28(金)にしておきます。直前で焦る事のないよう計画的に作ってください。

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ますが)でしょう。

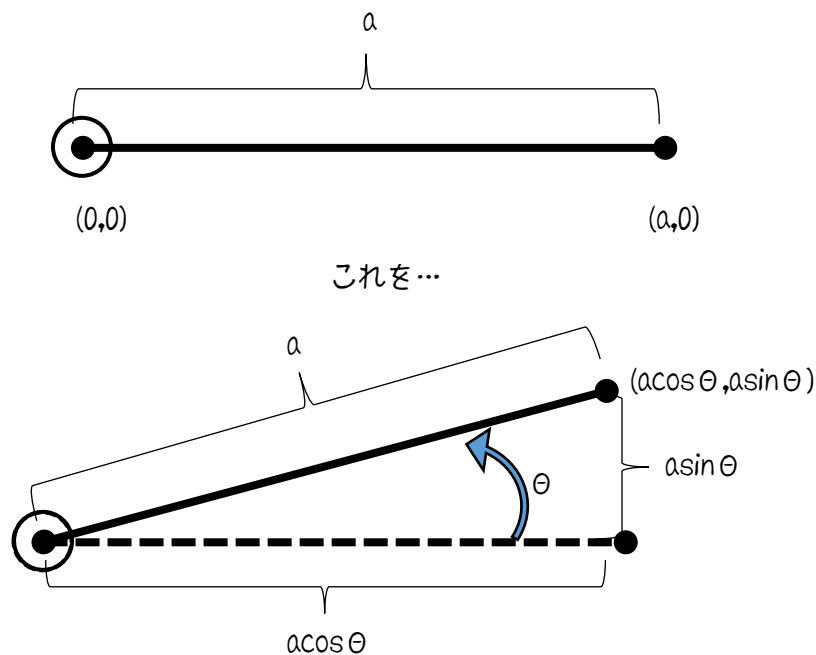
ところがねえ、例えば



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

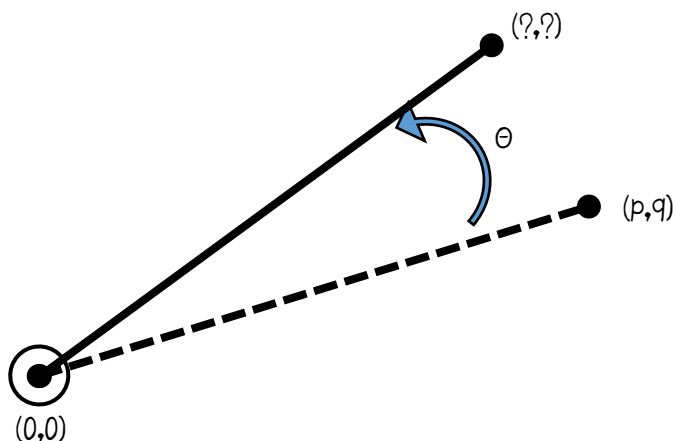
だが残念ながらそうもいかないんだよねえ…。まあ実際には当たり判定の話だけじゃなくて、多関節のキャラを動かしたりするとき等に『線分』を回転させるとどうなるのか？を考えなければならぬのだ、めんどくせえけど。つまり…



こうするとする。

そうなる $(a,0) \rightarrow (a\cos\theta, a\sin\theta)$ 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。



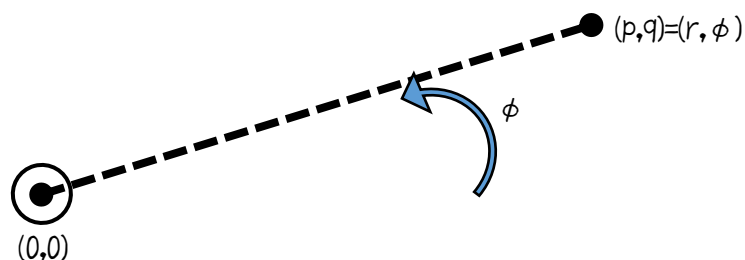
ちよつとこの辺から、1年生までにやったところを理解しないとガチ難しいのでよく聞いてください。回転した結果を $?, ?$ にしていますが、どうなるんでしょうね、これ。

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られるのです。



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 $(p, q) \rightarrow (r, \phi)$ 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 $(r, \phi + \theta)$ 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います。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るよ〜。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。おさらいすところ。

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

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

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、まあそれは置いといて、注目すべきは $r \cos \phi$ と $r \sin \phi$ です。

最初の定義

$$p = r \cos \phi, \quad q = r \sin \phi$$

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でしょうか…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ろまでの場合、『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!

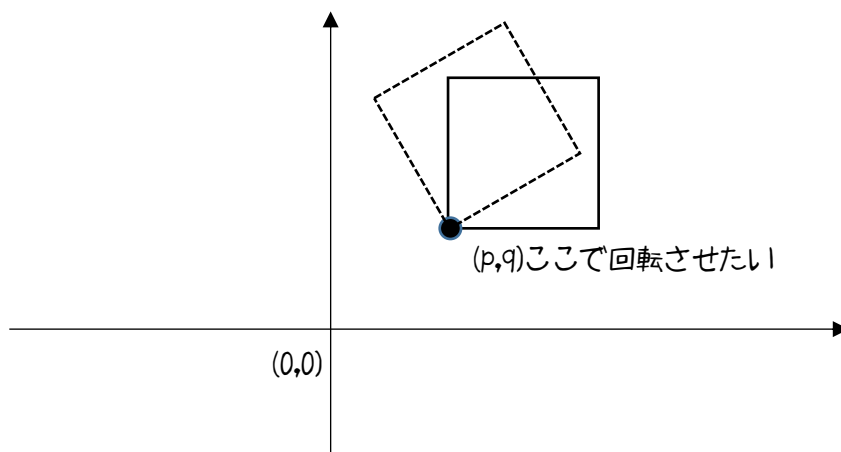
大したことじゃない。任意座標の回転は次の言葉を覚えてればいい。

平行移動に始まり平行移動に終わる

なんだっそら…。

と思うかもしれないが、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いないんだけど、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 DxDlib 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…。

川野部屋¥数学¥回転の話¥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。

まあ4頂点しかないの、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。

ままだ、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…

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

もう、気が狂うほど複雑なんじゃ

まあ、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 に書いといってください。

とりあえず、何もしない1行列を返す IdentityMat 関数を作ってください。何もしない関数って

どういのでしたっけ？そう、単位行列ですよ。

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

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!!やれつつてんだるおおお!!!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あ!?やってみろ!!!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〜くつつつつつつつつつ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わからん?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。

まあ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、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よ。

```
///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。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);  
}
```

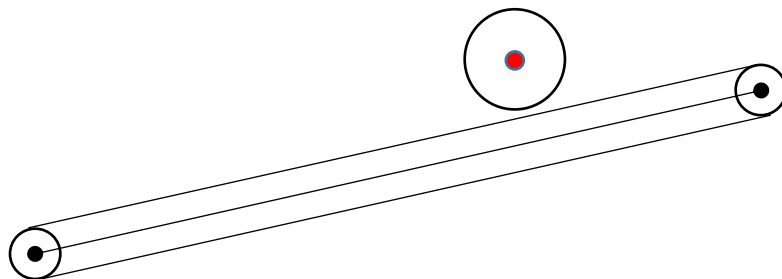
これで『行列一回作ってしまえばあとは適用するだけ』ということがお分かりいただけたであろうと思います。

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使っとけてところなんですけど、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)というのだけけど、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。

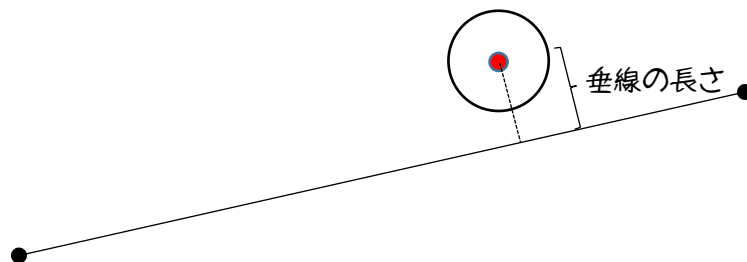
だが、こいつは少々扱い…アルゴリズムが厄介なのだ。本当に厄介。だから今回はもうちょっとだけ簡単で、とっても軽くて扱いやすい『カプセル型』を使用することにした。



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。

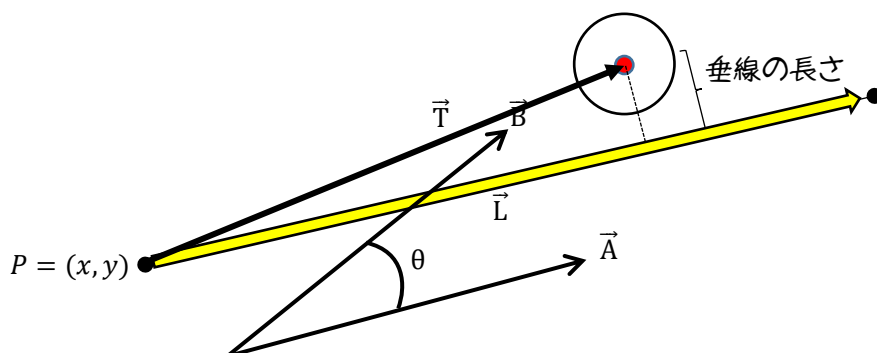


これをどうやって求めましょうか…一応求める際に必要になってくるのは、この線のベクトルと『線の起点P』から、円の中心点までのベクトルの二つです。

線分のどちらかの端点(どちらでもいい)から、円の中心点までのベクトルを \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 $\cos\theta$ の関係を覚えておられますでしょうか？

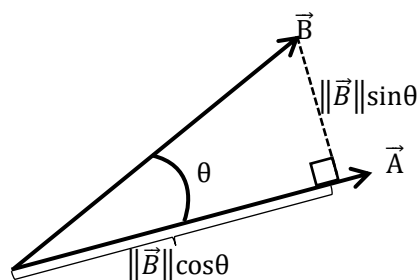
A ベクトルと B ベクトルがこのような位置関係の場合、内積とその間の角度 θ の関係は

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

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 $\cos\theta$ 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 $B\cos$ 側ですよね？ということ

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

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『 \times 』で表します。

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

まあ、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年でやっとす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

$$\vec{A} \cdot \vec{B} = x_A x_B + y_A y_B$$

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。

$$x_A = r_A \cos \theta_A$$

$$y_A = r_A \sin \theta_A$$

$$x_B = r_B \cos \theta_B$$

$$y_B = r_B \sin \theta_B$$

とします。そうすると先ほどの

$$\vec{A} \cdot \vec{B} = x_A x_B + y_A y_B$$

が別のものに見えてきませんか…?ほら…

$$\vec{A} \cdot \vec{B} = r_A \cos \theta_A r_B \cos \theta_B + r_A \sin \theta_A r_B \sin \theta_B$$

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

$$\vec{A} \cdot \vec{B} = r_A r_B (\cos \theta_A \cos \theta_B + \sin \theta_A \sin \theta_B)$$

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

$$r_A r_B (\cos \theta_A \cos \theta_B + \sin \theta_A \sin \theta_B) = r_A r_B \cos(\theta_A - \theta_B)$$

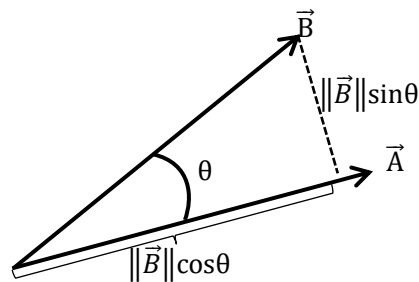
θ_A, θ_B とともにベクトルの向きを表す角度であり、それがマイナスされているということは『間の角度』といえる。つまり

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

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



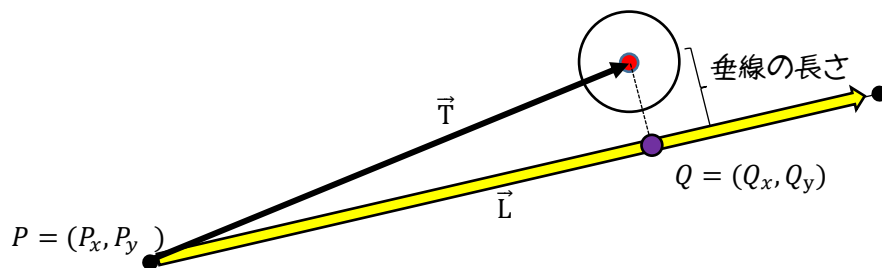
さて、ここまでの話で納得できたかどうかはわからないが、とにかく射影の長さは

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

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!!

しょうがねえなあ、俺がいつちよ計算してやっか。

先ほども言った通り、 \vec{T} から \vec{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あ…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だけかな？

…そう、正規化である。つまり

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

は『 \vec{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。

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

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

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

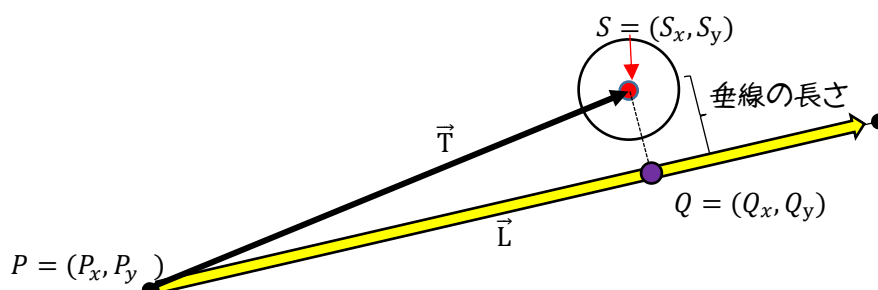
である。

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

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

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な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た。



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

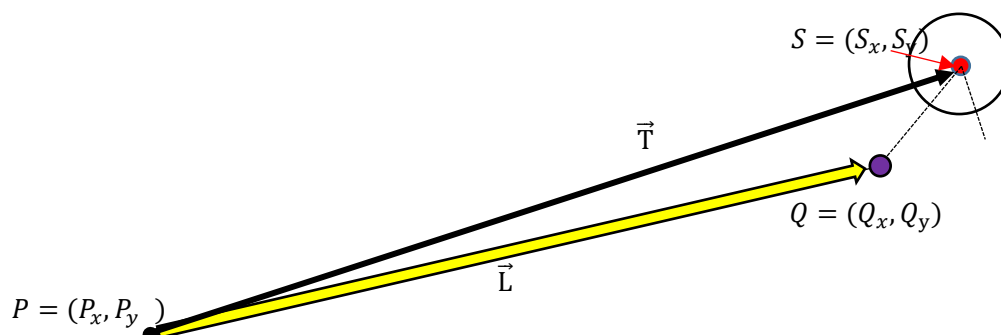
$$\|Q - S\| = \sqrt{(Q_x - S_x)^2 + (Q_y - S_y)^2}$$

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…

ちょっと罣があります。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ことこれ

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

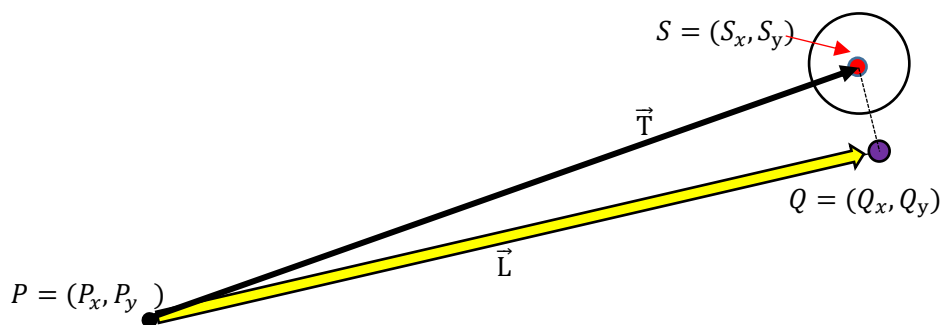
え？なんでこれが役に立つの？って思うかもしれませんが、この式、よ〜〜〜く考えてください。役に立つんですが…難しいかな〜。

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

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

ですよ？

ここはご理解いただける？で、例えば射影が線分の長さ $\|\vec{L}\|$ 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

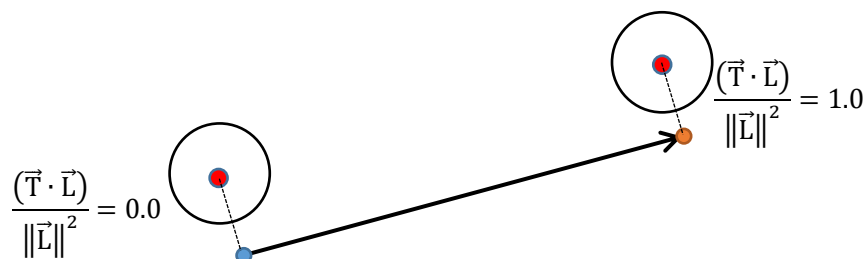
$$\frac{(\vec{T} \cdot \vec{L})}{\|\vec{L}\|^2} = \frac{(\vec{T} \cdot \vec{L})}{\|\vec{L}\|} \frac{1}{\|\vec{L}\|} = \frac{\|\vec{L}\|}{\|\vec{L}\|} = \frac{\|\vec{L}\|}{\|\vec{L}\|} = 1$$

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。

ここはプログラマ的に書くとわかりやすいかと思うので、プログラマ的に書きます。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だ！

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。

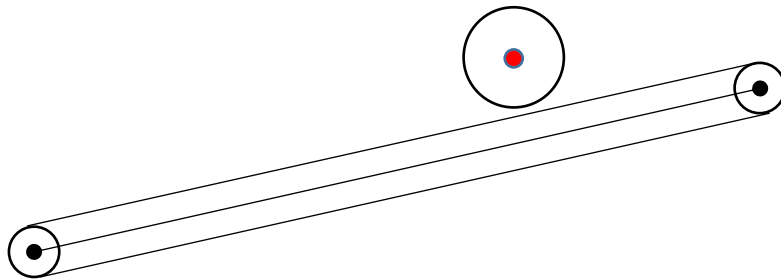
つまり $L * t$ もしくは $L.Scale(t)$ で、そのベクトルが求まります。なお、座標はそれに線分の始点を足してやればいいので、

```
Position2f pos=posa+L*t;
```

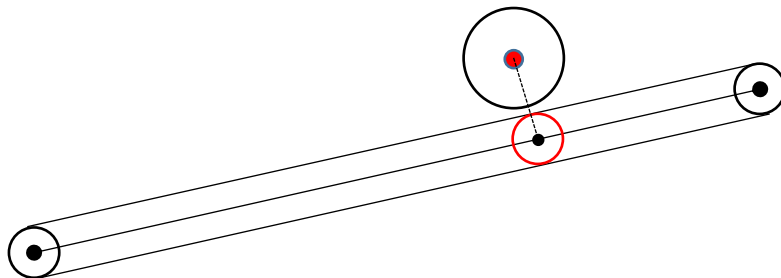
で補正済み射影点が求まります。

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、さっきちよろっと書いた

```
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。ここまできたならそれくらいできるでしょ。

要件

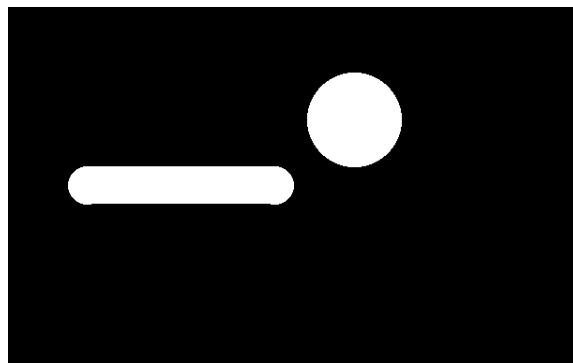
さて、ここから何を作ってほしいのかといいますと、今までの

- 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。

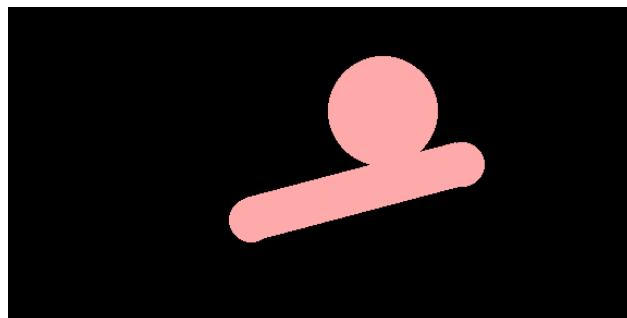
- 円は上下左右カーソルキーで移動すること
- カプセルは画面上をマウス押下で回転すること
- 通常時は円とカプセルは白色で描画すること
- 衝突時は円とカプセル両方ともに色を変更すること

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。

さあれっつら GO