ソフトゼミＢ　第４回

あたり判定

今回はあたり判定です。いよいよゲームが完成します。今回も穴埋め形式ですが、それが終わって文章もだいたい読んだ人は自主的にゲームをいじって、自分好みの魔改造をしましょう。

✤あたり判定の実装

今回までに書いたプログラムをみて、コメントアウトしてあるプログラムの部分があれば、全部外しましょう。おそらく順調に進んでいれば judge 系と move_item 関数を実装してないと思うので、それらを実装してください。前回と同じく穴４ は合計で二つあります。update.cpp においてある関数の実行部と function.h の宣言部分も忘れずに開放しておいてください。それが終われば完成です。おつかれさまでした。

✤解説

とはいえ、これで本当に終わってはあんまりなので、解説をします。今回はあたり判定の関数に絞って説明していきます。次回に全体のシステムの流れを解説しますが今回で作業は終わりなので、暇な人は適当にシステムを魔改造すると良いでしょう。

judge 関数

judge.cpp に入っている judge 関数の役割は、あたり判定です。厳密にいうとすれば、プレイヤー、敵、弾、アイテムといった画面上に表示される複数の画像とで重なりがあるかどうかを判定する、といったところでしょうか。

その都合上どのように重なりを判定するかが重要です。おおむね次の方針があります。

1. 座標の大小を直接比較する
2. 距離を取って、ある点の中心との近さによって判定
3. その他数学的な手法を用いる

それぞれいいところ、悪いところがあります。

一つ目の方法は単純で分かりやすいですが、長方形以外のものに応用が利かせにくいのがデメリットです。

二つ目は円形のあたり判定を作るときに重宝します。math.h に距離を取る関数があるのでこれを使うとかなり良いです。

三つ目の方法はプログラムをするのが難しくなりがちなのと、そもそもそういった判定のボキャブラリーが必要なのが難点です。

また、判定の行い方も二種類あります。

1. ある範囲をまとめて判定
2. 座標一点一点を一つ一つ判定

一つ目の方法はおおざっぱに形を判定するのには向いていますが、複雑な図形を判定するにはやはり二つ目の方法が良いでしょう。とはいえ、二つ目の方法はプログラムの手間がかかる上、パソコンに負荷がかかりがちなので積極的に行いたくはありません。

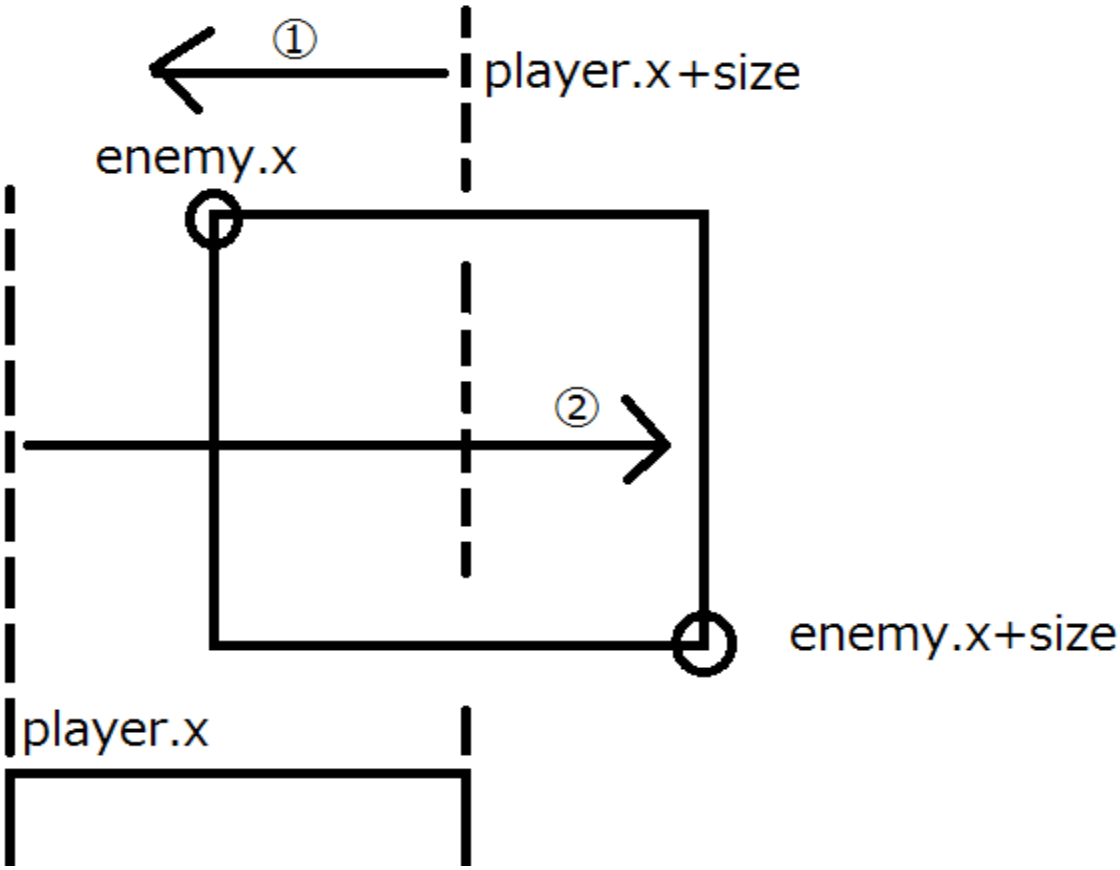
この judge 関数では判定は座標を直接比較し、まとめて判定しています。実際に判定部分を見てみましょう。

以下は敵の弾と自機のあたり判定を行う関数です。

```
//敵の弾と自機
void judge_enemybullet_to_player()
{----------------------------------中略-----------------------------------
                                        //存在しているものが
                                        if (enemybullet[i][j].hp)
                                        {
                                                //自機と重なっていたら
if (enemybullet[i][j].x < player.x + player.size_x &&
enemybullet[i][j].x + enemybullet[i][j].size_x > player.x &&
enemybullet[i][j].y < player.y + player.size_y &&
enemybullet[i][j].y + enemybullet[i][j].size_y > player.y)
                                                {
enemybullet[i][j].hp = 0; //弾の存在を消し
player.hp -= enemybullet[i][j].atk; //自機の HP から弾の攻撃力分減らして
```

```
if (player.hp < 0)
{
player.hp = 0;//自機の HP が 0 未満にならないようにする
}
----------------------------------以下略----------------------------------
```

重要なのは判別式の部分です。文字の形だとわかりにくいので図に表します。



上の図はｘ座標のみを判定していますが、ｙ座標も同様に行っています。

① が　enemybullet[i][j].x < player.x + player.size_x
② が　enemybullet[i][j].x + enemybullet[i][j].size_x > player.x

に対応しています。もし図の左側に出ていれば①が、右側に出ていれば②が範囲から出ることが確認できると思います。ｙ座標でも同様に考えて範囲の判定を行っています。

create_enemybullet 関数内の自機狙い弾について

create_enemybullet 関数には自機狙いにするための記述があります。ここも確認してみましょう。

```
//自機狙いにしてみた
//ang に自機と敵との角度を入れる
//ang = atan2((double)((player.y + player.size_y / 2) - (enemy[i].y + enemy[i].size_y)), (double)((player.x + player.size_x / 2) - (enemy[i].x + enemy[i].size_x / 2)));
//enemybullet[i][j].vx = cos(ang) * 5.0;
//enemybullet[i][j].vy = sin(ang) * 5.0;
```

atan2 という関数は math.h に格納されてる数学関数の一つです。アークタンジェントを返してくれます。引数は（(自分を基準に置いた目標に対しての）y 座標,x 座標）で使えます。

これだけだと自機に向かうだけですが、この自機に向かうという処理を敵の弾生成時に行うのではなく、敵の弾の移動の際に処理してやったり、create_enemy 関数の引数を増やして自機を狙う敵と自機を狙わない敵の両方を同時に作ることができます。というわけで皆さん順調に進んでいるので、今回はこういったシステム実装のノルマを課そうかと思います。

✤ ノルマ

1.自機狙い弾の実装
2.ホーミング弾の実装
（ホーミング弾 ： 弾が発射された後も、弾が目標に向かって向きを変えて進むような弾）
3.自機狙い弾、ホーミング弾の stage1 関数内でのスイッチ化。敵を生成する際に自機狙い弾にするか、ホーミング弾にするか、それとも何も狙わないようにするかを選択できるようにしてください。