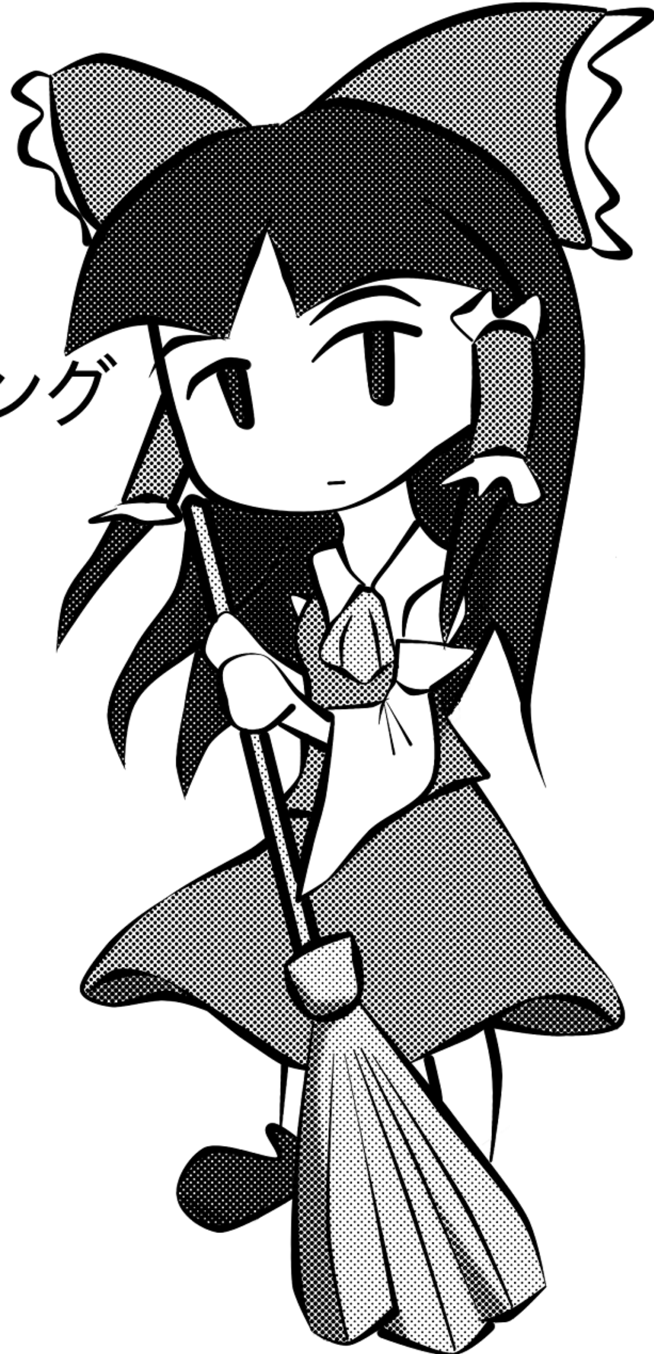
花映塚AIをつくる

花 AI 塚ではじめる

STG AI プログラミング



まえがき

花映塚を遊んだことがある人なら、花映塚の AI の動きに魅せられたことがあるでしょう。電動歯ブラシと名高いその動きを素晴らしいと思うか、それとももっとうまくできると思うかは人それぞれですが、いずれにせよ、いつか自分で花映塚 AI を作りたいと思ったはずです (この本を手に取ったということはそうでしょう)。この本では拙作のツール『花 AI 塚』を使って花映塚 AI を作っていきます。

東方界隈において AI を自作するという試みはこれまでにも行われてきました[*1]。しかしこれまでの AI はもっぱら C++ などでハードコーディングされていて、いざ自分で AI を作ろうと思ったらゲームの状態を取得するための仕組みを作るところからはじめなければなりませんでした。つまり、AI を作ろうと思ったら AI を動かすための土台から作る必要があったのです。

花 AI 塚は花映塚上で AI を動かすための土台を提供します。自機の位置や弾の当たり判定など AI に必要な情報の取得を一手に担い、誰もがスクリプトで手軽に AI を作れます。自作した AI と対戦したり、AI 同士で対戦させたりすることができます。

AI を作るために、土台を作る知識と手間を背負う必要はありません。この本を手に、花 AI 塚を動かせば今すぐにでも花映塚 AI を作ることができます。この本を通じて花映塚 AI を作ることをもっと気軽に楽しめるようになればと思います。

なお、本書に出てくるスクリプトは以下のリポジトリで配布しています。

- `https://bitbucket.org/ide_an/start_writing_kaeiduka_ai_script/`

[*1] trial-run 氏の『AI にダブルスポイラーをプレイさせる本』(http://trial-run.net/archives/2235) や@aki33524 氏の紅魔郷 AI(http://www.slideshare.net/aki33524/ai-32089294) などがある。

目次

第 1 章　はじめる 5
1.1 花 AI 塚の導入 5
1.2 はじめての AI 6
1.3 ゲームの情報を拾う 8
1.3.1 自機 8
1.3.2 敵 9
1.3.3 弾 10
1.3.4 EX アタック 10
1.3.5 アイテム 11

第 2 章　避ける 12
2.1 簡単な回避 AI 12
2.1.1 何を避ける? 12
2.1.2 どう避ける? 12
2.1.3 AI を書く 13
2.1.4 回避 AI の問題 17
2.2 もっと先を読む AI 17
2.2.1 N フレーム先を予測する 17
2.2.2 自機の取り得る位置 18
2.2.3 被弾リスクの計算 19
2.2.4 AI を書く 20
Column: ログを取る 22

第 3 章　撃つ 23
3.1 弾を撃つ 23
3.2 チャージアタックを撃つ 24
3.3 避けつつ撃つ 25
3.4 狙った敵を撃ちに行く 26

3.4.1 アルゴリズムを考える 26
3.4.2 AI を書く 27
3.4.3 AI の戦果 30
Column: 花 AI 塚の仕組み 30

第 1 章

はじめる

1.1　花 AI 塚の導入

花 AI 塚は以下のページで配布されています。

- http://www.usamimi.info/~ide/programe/touhouai/

インストールは zip を解凍するだけです。フォルダには以下のファイルがあります。

ファイル名	説明
ka_ai_duka.exe	花 AI 塚を起動するプログラム
ka_ai_duka.ini	設定ファイル
setting.exe	設定ファイル編集用のプログラム
inject.dll	花 AI 塚の本体。花映塚のプロセスにロードされる
readme.txt	説明書
LICENSE.txt	ライセンス関連
doc/	API リファレンス
sample-scripts/	サンプル AI 集

表 1.1　花 AI 塚の配布ファイル

花 AI 塚を動かすに当たって 2 つのことを設定する必要があります。

- 東方花映塚のプログラム (th09.exe) へのパス
- 動かす AI のスクリプトへのパス

これらを設定するには setting.exe を起動して、それぞれファイルパスを指定すれば OK です (図 1.1)。今回は動かす AI のスクリプトとして sample-scripts/random-walk/main.lua を指定することにします。

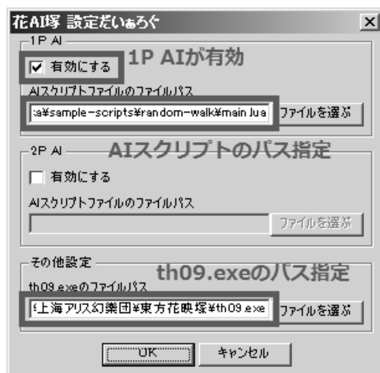


図 1.1　setting.exe での設定

設定を保存したら、ka_ai_duka.exe を起動します。すると東方花映塚が立ち上がります。この状態でストーリーモードやマッチモード (スクリプト AI 側を人間にする) でゲームを始めるとランダム・ウォークする AI が動きます。

1.2　はじめての AI

ここからは花 AI 塚での AI の作り方を見ていきます。ひとまず左右を往復するだけの AI を作ることにします。

一般にシューティングゲームの AI は以下のことを毎フレーム繰り返しています (図 1.2)。

1. 現在のゲームの状態 (自機の位置や弾の当たり判定など) を取得する
2. 取得した情報にもとづいて、次に行うべき自機操作を決める
3. 自機操作をゲームへの入力 (キー入力など) として送る

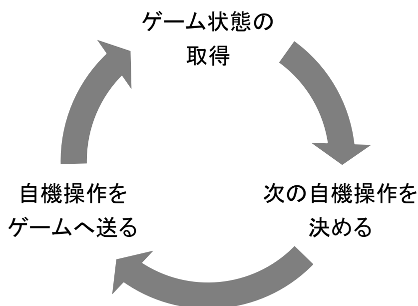


図 1.2　シューティングゲーム AI の流れ

まずは自機操作をキー入力として送るところから始めてみましょう。

ソースコード 1.1　chap1/main.lua 左移動キーを送るだけ

```
function main()
  sendKeys(0x40)
end
```

花 AI 塚では毎フレーム main 関数が呼び出されます。AI の処理は基本的にこの中に書いていきます。main 関数の中で呼ばれている sendKeys 関数は、ゲームへの入力を送る関数です。

sendKeys 関数には花映塚へどんなキーを送るかを表す引数を渡します。引数の各 bit が花映塚の操作のどのキーを押しているかに対応しています (表 1.2)。たとえば上のコードでは引数として 0x40 を渡しているので下から 7bit 目に対応するキー、すなわち左移動キーが押された状態がゲームへの入力として送られます。

表 1.2　sendKeys の引数

bit	説明	bit	説明
1bit 目	射撃 (Z キー)	5bit 目	上移動 (↑キー)
2bit 目	緊急ボム (X キー)	6bit 目	下移動 (↓キー)
3bit 目	低速移動 (Shift キー)	7bit 目	左移動 (←キー)
4bit 目	未使用	8bit 目	右移動 (→キー)

左右往復する AI を書き進めていきましょう。画面端に行くまで横に動き続け、画面端にたどり着いたら反対向きに動くようにします。自機が今どっちの方向に動いているかを覚えておく必要があるので、変数 is_moving_right を用意します。

ソースコード 1.2　chap1/main.lua 移動方向を覚えておく

```
local is_moving_right = true
function main()
  if(is_moving_right --[[ and 右端にぶつかった ]]) then
    -- 画面右端にぶつかったので切り返す
    is_moving_right = false
  elseif (not(is_moving_right) --[[ and 左端にぶつかった]]) then
    -- 画面左端にぶつかったので切り返す
    is_moving_right = true
  end
  if (is_moving_right) then
    sendKeys(0x80) -- 右移動のキーを送る
  else
    sendKeys(0x40) -- 左移動のキーを送る
  end
end
```

自機が画面端にぶつかったかどうかを知るには自機の位置を知る必要があります。花 AI 塚ではいくつかのグローバル変数を通してゲームの状態を知ることができます。たとえば 1P 側の自機の情報を取得するには game_sides[1].player を参照します。また、AI が 1P 側か 2P 側かは player_side を参照すれば分かります。

```
local my_side = game_sides[player_side] -- player_sideは1Pなら1，2Pなら2
local player_x = my_side.player.x -- 自機のX座標を取得
```

左右往復する AI の完成版は以下のようになります。

ソースコード 1.3　chap1/main.lua 完成版

```
local X_MAX = 136
local X_MIN = - X_MAX
local is_moving_right = true
function main()
  local my_side = game_sides[player_side]
  local player_x = my_side.player.x
  if(is_moving_right and X_MAX <= player_x) then
    -- 画面右端にぶつかったので切り返す
    is_moving_right = false
  elseif (not(is_moving_right) and player_x <= X_MIN) then
    -- 画面左端にぶつかったので切り返す
    is_moving_right = true
  end
  if (is_moving_right) then
    sendKeys(0x80) -- 右移動のキーを送る
  else
    sendKeys(0x40) -- 左移動のキーを送る
  end
end
```

それでは実際にこの AI を動かしてみましょう。setting.exe で chap1/main.lua を読み込むように設定した後、ka_ai_duka.exe を起動すればこの AI が動かせます。

1.3　ゲームの情報を拾う

1.2 節では AI を作るために自機の座標を取得していました。花 AI 塚では他にも様々な情報を取得できるので、ここで紹介しておきます。

1.3.1　自機

1.2 節でも出てきたように、自機に関する情報は `game_sides[player_side].player` から取得できます。`player` からは以下の情報が得られます (図 1.3)。

```
local player = game_sides[player_side].player
player.x                -- 自機座標
player.y                --
player.life             -- ライフ
player.hitBodyRect      -- 当たり判定。矩形、円形、対アイテム用などいくつかある
player.currentCharge    -- チャージゲージのチャージ量
player.currentChargeMax -- チャージ上限
player.chargeSpeed      -- チャージ速度
player.character        -- どのキャラクターか?
player.speedFast        -- 移動速度(高速時)
player.speedSlow        -- 移動速度(低速時)
player.combo            -- コンボ数
player.spellPoint       -- スペルポイント
player.cardAttackLevel  -- カードアタックレベル
player.bossAttackLevel  -- ボスカードアタックレベル
```

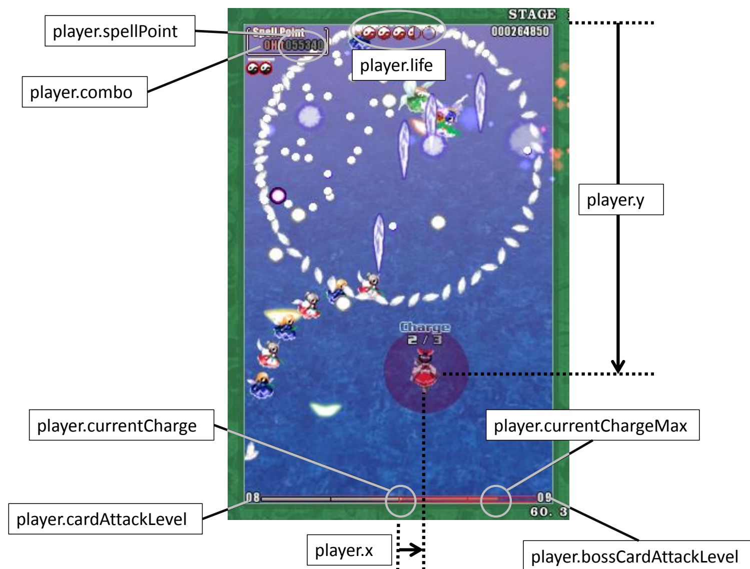


図 1.3 自機の情報

1.3.2 敵

敵に関する情報は `game_sides[player_side].enemies` から得られます。自機と違って敵は複数存在することがあります。なので `enemies` は配列になっています。`enemies` の要素からは以下の情報が得られます。

```
local enemy = game_sides[player_side].enemies[1]
enemy.id                   -- ID
enemy.x                    -- 座標
enemy.y                    --
enemy.vx                   -- 速度
enemy.vy                   --
enemy.isSpirit             -- 幽霊かどうか
enemy.isActivatedSpirit -- 活性化した幽霊かどうか
enemy.isBoss               -- ボスかどうか
enemy.isLily               -- リリー・ホワイトかどうか
enemy.isPseudoEnemy        -- 擬似的な敵かどうか
enemy.hitBody              -- 当たり判定
```

敵や弾、EX アタックには一意な ID が振られています。1 フレーム前にいた敵が次のフレームでどこに行ったかを調べるときにはこの ID が一致する敵を探します。

敵にはいくつか種類があり、`enemy.is`〜のフィールドを見ることで区別することができます (図 1.4)。擬似的な敵 (`enemy.isPseudoEnemy == true` な敵) は画面上では見えない敵です。擬似的な敵は当たり判定がなく、倒すこともできないのですが、EX アタックやカードアタックの弾を配置するなどの仕事をしており、こういった攻撃を回避するために使えます。

図 1.4　敵の情報

1.3.3　弾

弾に関する情報は game_sides[player_side].bullets から配列で得られます。bullets の要素からは以下の情報が得られます。

```
local bullet = game_sides[player_side].bullets[1]
bullet.id           -- ID
bullet.x            -- 座標
bullet.y            --
bullet.vx           -- 速度
bullet.vy           --
bullet.isErasable   -- 消せる弾かどうか
bullet.hitBody      -- 当たり判定
```

花映塚では敵を撃破すると周囲の白弾を消すことができます。bullet.isErasable はそのように敵の撃破で消せる弾かどうかを識別するためのフィールドです。通常の弾の他、レーザーも bullets の要素として管理されています。

1.3.4　EX アタック

EX アタックとはいい感じに幽霊を倒したり Ex アイテムを拾ったときに発生する、自陣から相手陣への攻撃です。霊夢の陰陽玉やチルノの氷柱などがこれに該当します。EX アタックに関する情報は game_sides[player_side].exAttacks から配列で得られます。exAttacks の要素からは以下の情報が得られます。

```
local exAttack = game_sides[player_side].exAttacks[1]
exAttack.id        -- ID
exAttack.x         -- 座標
exAttack.y         --
exAttack.vx        -- 速度
exAttack.vy        --
exAttack.type      -- 種類
exAttack.hittable  -- 当たり判定が有効かどうか
exAttack.hitBody   -- 当たり判定
```

EX アタックはキャラによって固有で、変則的な作用をもたらすものもあります。

- メディスン・メランコリーの毒霧 (`exAttack.type == ExAttackType.Medicine`) は当たっている間自機の速度が遅くなります。
- 四季映姫ヤマザナドゥの EX アタック (`exAttack == ExAttackType.Eiki`) は白弾が当たると新たな弾を生成します。

また、EX アタック自体が当たり判定を持たずに他のオブジェクトの生成だけを行うこともあります。

- 霧雨魔理沙の EX アタック (`exAttack.type == ExAttackType.Marisa`) は擬似的な敵を生成します。この敵はレーザーを生成したのち消滅します。
- 小野塚小町の EX アタック (`exAttack.type == ExAttackType.Komachi`) は大量の弾を生成したのち消滅します。

このように EX アタックの振る舞いはキャラクターによって様々なため、本格的に EX アタックに対処する AI を作るにはキャラクターごとに対策を考える必要があります。

1.3.5　アイテム

花映塚ではリリー・ホワイトやボスを倒したときにアイテムが降ってきます。アイテムに関する情報は `game_sides[player_side].items` から配列で得られます。`items` の要素からは以下の情報が得られます。

```
local item = game_sides[player_side].items[1]
item.id       -- ID
item.x        -- 座標
item.y        --
item.vx       -- 速度
item.vy       --
item.type     -- 種類
item.hitBody  -- 当たり判定
```

以降の章ではゲームの状態に関するこれらの情報をもっと駆使して動く AI を作っていきます。

第 2 章

避ける

花映塚では常に敵や弾を避ける必要があります。1.2 節の AI はただ左右に動くだけでしたが、もっとゲームの情報を活用して敵や弾を避ける AI を作っていきます。

2.1 簡単な回避 AI

2.1.1 何を避ける?

AI を作るにあたって、まず何を避ける必要があるかはっきりさせておきます。以下の 3 つです。

- 弾 (`game_sides[player_side].bullets`)
- 敵 (`game_sides[player_side].enemies`)
- EX アタック (`game_sides[player_side].exAttacks`)

これらはいずれも当たり判定を持っていて、ぶつかるとダメージを受けるので避けないといけません[*1]。

2.1.2 どう避ける?

何を避けるかが定まったので、次はどう避けるかを考えます。避け方には大きく 2 つのアプローチが考えられます。

- 積極的に避ける: 敵や弾の少ない場所を見つけて、そこに向かって動いていく
- 消極的に避ける: 敵や弾に当たりそうなときだけ動いて避ける

積極的に避けるには移動先の決定や移動経路の決定、実際の移動処理などを考えないといけません。なのでまずは消極的に避けることにします。

[*1] たまに当たり判定を持たないものもある。その場合は避けなくて OK

消極的に避けるには、敵や弾に当たりそうかどうかを知る必要があります。そのためには敵や弾がどう動いていくかをシミュレートします。シミュレートした結果、数フレーム後に被弾すると分かったら避ける動作を選ぶというのが消極的な回避の流れです。

まずは 1 フレーム先だけをシミュレートして避ける AI を作ることにします。花 AI 塚では敵や弾の速度を取得できます。なので 1 フレーム後の敵や弾の位置は

$$(1\text{ フレーム後の位置}) = (\text{現在の位置}) + (\text{現在の速度})$$

で求まります。敵や弾を 1 フレーム後の位置にずらしてから当たり判定処理をすれば、1 フレーム後に被弾するかどうかが分かります。

では被弾すると分かったら、どのように動けばいいでしょうか。自機の移動操作は全部で 17 通りあります[*2]。花 AI 塚では自機の移動速度もわかるので、それぞれの移動操作について 1 フレーム後にどこに移動しているかをシミュレートできます。よって、17 通りの移動操作それぞれについて 1 フレーム後に被弾するかを調べ、被弾しない移動操作を選べば良いということになります。

以上のことをアルゴリズムとしてまとめると下のようになります。

1. 自機の移動操作を列挙する。それぞれの操作の被弾リスクを 0 とする
2. 1 フレーム後の敵や弾、EX アタックの位置を予測する
3. 列挙した自機の移動操作それぞれについて、1 フレーム後の自機の位置を求め、その位置で敵や弾、EX アタックとの当たり判定処理を行う
    - 衝突している場合はその移動操作の被弾リスクを増やす
4. 最も被弾リスクの少ない移動操作を採用する

ここでは被弾リスクというものを導入して、移動操作を選ぶときの基準にしています。

2.1.3　AI を書く

では実際に AI のコードとして実装していきます。トップダウンで `main` 関数から書いていきます。

ソースコード 2.1　chap2-1/main.lua main 関数

```
function main()
  local myside = game_sides[player_side]
  local player = myside.player
  -- 選択する移動操作の候補を生成
  local candidates = generateCandidates(player)
  -- それぞれの移動操作の被弾リスクを計算
  calculateHitRisk(candidates, myside.enemies, player)
  calculateHitRisk(candidates, myside.bullets, player)
```

[*2] 移動方向が上下左右斜めで 8 通り、それぞれについて高速移動と低速移動の 2 通り、くわえて移動しない (停止) という操作があるので合わせて 8 * 2 + 1 = 17 通り。

```
  calculateHitRisk(candidates, myside.exAttacks, player)
  -- 被弾リスクが一番低い移動操作を選ぶ
  local choice = choose(candidates)
  -- キー入力として送信
  sendKeys(choice.key)
end
```

generateCandidates 関数は自機の移動操作の列挙、calculateHitRisk 関数は各移動操作の被弾リスクを計算します。被弾リスクを元に choose 関数で次に自機が行う移動操作を決めて sendKeys 関数で実際のゲームに反映しています。

まずは generateCandidates 関数について見ていきましょう。

ソースコード 2.2　chap2-1/main.lua generateCandidates 関数

```
local function generateCandidates(player)
  local candidates = {}
  local dxs = {0, 1, -1, 0, 0}
  local dys = {0, 0, 0, 1, -1}
  -- キー入力。停止、→、←、↓、↑
  local keys = {0x0,0x80,0x40,0x20,0x10}
  for i=1, #keys do
    candidates[i] = {
      key = keys[i],
      dx = player.speedFast * dxs[i],
      dy = player.speedFast * dys[i],
      hitrisk = 0
    }
  end
  return candidates
end
```

1 フレーム後に自機が取り得る位置は 17 通りありますが、今回は簡単のため以下の 5 通りだけを考えます。

- 上下左右の高速移動 (斜め移動はなし)
- 停止

generateCandidates 関数の中では、「停止」「右」「左」「下」「上」の順に自機の移動操作を列挙しています。それぞれの移動操作について、自機が今の位置からどれくらい動くか (dx と dy) とキー入力 (key)、被弾リスク (hitrisk) をフィールドに持たせています。

次は calculateHitRisk 関数を見ていきますが、その前に main 関数での呼び出しをもう一度見ておきましょう。

```
  calculateHitRisk(candidates, myside.enemies, player)
  calculateHitRisk(candidates, myside.bullets, player)
  calculateHitRisk(candidates, myside.exAttacks, player)
```

敵、弾、EX アタックの順に被弾リスクの計算をしています。enemies や bullets などはいずれも配列です。1.3 節で見たように敵、弾、EX アタックでそれぞれ持ってるフィールドが違いますが、座標や速度、当たり判定の取得方法は共通しています。なので calculateHitRisk 関数は敵、弾、EX アタックを区別せずに呼び出せます。

それでは calculateHitRisk 関数を見ていきましょう。

ソースコード 2.3　chap2-1/main.lua calculateHitRisk 関数と補助関数

```
local function adjustX(x)
  return math.max(-136, math.min(x, 136)) -- -136 <= x <= 136
end

local function adjustY(y)
  return math.max(16, math.min(y, 432)) -- 16 <= y <= 432
end

local function calculateHitRisk(candidates, objects, player)
  for i, obj in ipairs(objects) do
    local obj_body = obj.hitBody
    if obj_body ~= nil then
      -- 1フレーム後のオブジェクトの座標を求める
      obj_body.x = obj_body.x + obj.vx
      obj_body.y = obj_body.y + obj.vy
      -- 自機の当たり判定はオブジェクトの当たり判定の種類によって使い分ける
      local player_body
      if obj_body.type == HitType.Circle then
        player_body = player.hitBodyCircle
      else
        player_body = player.hitBodyRect
      end
      -- それぞれの移動操作でぶつかるかどうか調べる
      for j, cnd in ipairs(candidates) do
        player_body.x = adjustX(player.x + cnd.dx)
        player_body.y = adjustY(player.y + cnd.dy)
        if hitTest(player_body, obj_body) then
          cnd.hitrisk = cnd.hitrisk + 1
        end
      end
      -- 座標を元に戻す
      obj_body.x = obj.x
      obj_body.y = obj.y
      player_body.x = player.x
      player_body.y = player.y
    end
  end
end
```

大雑把に何をやっているかは以下のとおりです。

- objects の各要素 (敵や弾、EX アタック) について
  1. その要素の当たり判定 (obj_body) を取得する
  2. obj_body の座標に速度を足す (これが 1 フレーム後の位置)
  3. 自機の当たり判定 (player_body) を取得する
  4. 移動操作の候補それぞれについて
     1. player_body の座標を移動後の座標にする
     2. player_body と obj_body との当たり判定処理を行う
     3. 当たっていたら被弾リスクを 1 足す
  5. player_body と obj_body の座標を元に戻す

当たり判定にはいくつか種類があり、それに応じて player_body を何にするかを決めます (表 2.1)。obj_body が矩形なら自機の当たり判定も矩形の判定を使い、円形なら自機の当たり判定も円形を使います。当たり判定は座標の情報を持っており、この座標を書き換えることで当たり判定

の位置を変えています。

表 2.1　obj_body の当たり判定の種類と、対応する自機の当たり判定

obj_bodyの当たり判定の種類	自機の当たり判定
HitType.Rect	player.hitBodyRect
HitType.RotatableRect	player.hitBodyRect
HitType.Circle	player.hitBodyCircle

calculateHitRisk 関数の引数 candidates には generateCandidates 関数で生成した自機移動操作の候補が渡されます。candidates の各候補 (つまり自機移動操作) について、1 フレーム後の自機の座標を求め、obj_body との当たり判定処理を行います。

```
        player_body.x = adjustX(player.x + cnd.dx)
        player_body.y = adjustY(player.y + cnd.dy)
        if hitTest(player_body, obj_body) then
          cnd.hitrisk = cnd.hitrisk + 1
        end
```

player_body の座標を更新する際、単に cnd.dx や cnd.dy を足すだけでなく、adjustX 関数や adjustY 関数を適用しています。これは自機が画面外に出られないことを考慮して自機移動をシミュレートするためです。adjustX 関数や adjustY 関数は渡された座標が画面外になっているときに画面内に収まるように補正する関数です。この補正によって自機移動のシミュレートを誤らないようにしています。

当たり判定処理は hitTest 関数で行います。この関数は花 AI 塚のシステムで定義されているもので、2 つの当たり判定を渡すとその 2 つがぶつかっているかどうかを返します。今回のコードではぶつかっているときには被弾リスク (cnd.hitrisk) を増やしています。

最後に choose 関数を見ていきます。

ソースコード 2.4　chap2-1/main.lua choose 関数

```
local function choose(candidates)
  local min_risk = 99999999
  local min_i = -1
  for i, cnd in ipairs(candidates) do
    if cnd.hitrisk < min_risk then
      min_risk = cnd.hitrisk
      min_i = i
    end
  end
  return candidates[min_i]
end
```

candidates には自機移動操作の候補が渡されます。choose 関数のやることはシンプルで、candidates の中から最も被弾リスクの低い候補を選んで返すだけです。

回避 AI のコードはこれですべてです。

2.1.4 回避 AI の問題

この AI を実際に動かしてみると、少しは弾を避けることもあるのですが、真上から来た弾や幽霊を避けずに被弾してしまうことが多いです。なぜでしょうか? 理由はいくつか考えられます。

- 回避する方向が悪い
  - 真上から来た弾に対して後ずさりして避けようとして詰むケースがある。弾の移動方向に対して直角に避ければよいはず。
- 先読みフレーム数が少ない
  - 幽霊と衝突するケースはこれに該当する。弾の速度が速い場合、1 フレーム前までは停止していれば被弾しなかったが、次のフレームではもうどの方向に動いても被弾してしまうという状況が起こりうる。もっと先のフレームまで予測した上で自機が動けば回避できる可能性が高い。

1 フレーム先だけでなく、もっと先まで予測すれば後者の問題は解決します。この解決法で前者の問題もある程度は解決できるでしょう。後ずさりして画面下に追い詰められた後でも被弾まで数フレームの余裕があれば別の方向への回避動作に切り替えることが期待できるからです。

2.2 もっと先を読む AI

さきほどの 1 フレーム先読みの AI を改良して、もっと先まで読む AI を作ります。

2.2.1 N フレーム先を予測する

さきほどは 1 フレーム先の敵や弾などの位置を予測しましたが、今度は N(>1) フレーム先の位置を予測します。1 フレーム先の予測では単純に現在の速度を足しました。花 AI 塚では現在の速度を正確に取得できるのでこの予測は正確です。しかし N フレーム先を正確に予測するのは困難です。というのも 1 フレーム、2 フレーム先など将来の速度を知ることができないからです。

とはいえここで匙を投げるわけにもいきません。予測が誤差を含むことを覚悟の上で敵や弾などの移動モデルを考えることにします。シンプルな移動モデルとしては以下のものが考えられます。

- 等速直線運動
  - ずっと同じ速度で動き続ける (速度 0 なら止まり続ける)
- 等加速度運動
  - ずっと同じ加速度で動き続ける (加速度 0 なら等速直線運動と同じ)

等速直線運動は現在の速度と 1 フレーム、2 フレーム先の速度が等しいと仮定することです。こ

の場合 N フレーム後の位置は単純に現在の速度の N 倍を足せば得られます。

$$(\text{N フレーム後の位置}) = (\text{現在の位置}) + N * (\text{現在の速度})$$

等加速度運動では等速直線運動よりも条件を緩めて、速度がだんだん変わってもいい、ただし速度の変化量 (加速度) は一定としています。花 AI 塚では直接加速度を求めることはできませんが、1 フレーム前の速度を記録して、現在の速度との変化量を求めることで加速度を知ることができるでしょう。

今回の AI では簡単のため、等速直線運動のモデルを採用します。

2.2.2　自機の取り得る位置

さきほどの AI では自機の移動操作が上下左右移動 + 停止の 5 通りだったので、1 フレーム後の自機の位置も 5 通りでした。では 2 フレーム後は何通りかというと 13 通りです。以降、フレーム数の 2 乗に比例して増えていきます (図 2.1)。

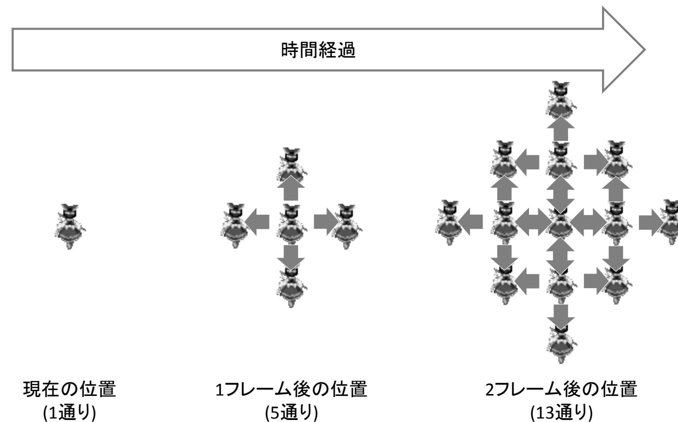


図 2.1　自機の取り得る位置はフレーム数の 2 乗に比例して増える

さきほどの AI では各移動操作の被弾リスクを求めるために 1 フレーム後の位置での当たり判定処理を行っていましたが、これを素直に N フレーム後に取り得る位置に拡張するのは計算量的に厳しそうです。なので N フレーム後に自機が取り得る位置について縛りを設けることにします。

ここで人間が花 AI 塚をプレイするときのことを考えてみましょう。自機を 1 フレームだけ右移動して停止する、といったようなフレーム単位での制御はできないでしょう。つまり一度右移動を始めたら数フレームは右移動を続けるはずです。

これと同じ縛りを AI にも適用することにします。つまり現在のフレームで移動操作を選んだら、N フレームの間その移動操作を続けるというわけです。そうすると N フレーム後の自機の位

置は移動操作の種類と同じで 5 通りとなります。N が増えるにつれて増加するということはありません (図 2.2)。

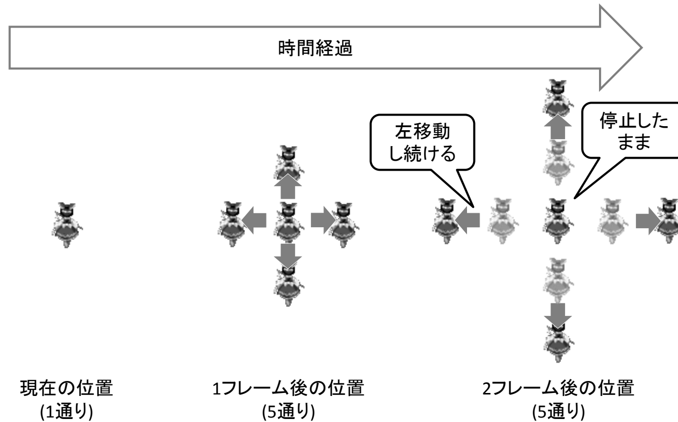


図 2.2 自機の取り得る位置を縛った場合、何フレーム経っても取り得る位置は 5 通り

N が大きくなると N フレームの間同じ移動操作を続けるという仮定は不自然になってきますが、数フレーム程度の先読みであれば問題ないでしょう。

2.2.3 被弾リスクの計算

1 フレームの先読みのときは、1 フレーム先の敵や弾などの位置を予測し、各移動操作について 1 フレーム先の位置での当たり判定処理を行い、ぶつかっていればその度に被弾リスクを 1 足すというものでした。N フレームの先読みの場合は、これを 1〜N フレーム先の各フレームについて行います。

敵や弾などの予測には誤差があることを前に述べました。誤差は時間が経つにつれて大きくなります。これはすなわち未来であればあるほど、被弾の予測の確度が低くなるということです。被弾リスクの計算もこのことを反映すべきでしょう。ここでは未来であればあるほど、加算する被弾リスクを小さくします。

N フレーム先読みの被弾リスクの計算量は 1 フレーム先読みの N 倍あります。敵や弾が合わせて 100 個あるとしたら、$100 * 5(=$自機の取り得る位置の数$) * N = 500N$ 回当たり判定処理を行わないといけません。このままでは AI が処理落ちする可能性が高いので、計算量を削りましょう。たとえば N フレーム後にぶつかる恐れのある敵や弾というのは、1 フレーム後の時点ですでにそこそこ自機に近いところにいるものだけと考えられます[*3]。2 フレーム後〜N フレーム後の被弾リスクの計算では、自機に近いものとだけ当たり判定処理をするようにすれば安全に計算量を落とせるでしょう。

[*3] N があまり大きくない時の話。

2.2.4　AI を書く

もっと先読みをする AI を実際に書いてみます。位置の予測や被弾リスクの計算だけを変えるので、さきほどの AI の calculateHitRisk 関数の実装を変えるだけで実装できます。

ソースコード 2.5　chap2-2/main.lua calculateHitRisk 関数

```
local N = 5 -- 5フレーム先までの先読み
local function calculateHitRisk(candidates, objects, player)
  for i, obj in ipairs(objects) do
    -- 1フレーム後のオブジェクトの座標を求める
    local obj_body = obj.hitBody
    if obj_body ~= nil then
      -- 自機の当たり判定はオブジェクトの当たり判定の種類によって使い分ける
      local player_body
      if obj_body.type == HitType.Circle then
        player_body = player.hitBodyCircle
      else
        player_body = player.hitBodyRect
      end
      if shouldPredict(obj_body, player_body) then
        for frame=1,N do
          obj_body.x = obj_body.x + frame * obj.vx
          obj_body.y = obj_body.y + frame * obj.vy
          -- それぞれの移動操作でぶつかるかどうか調べる
          for j, cnd in ipairs(candidates) do
            player_body.x = adjustX(player.x + frame * cnd.dx)
            player_body.y = adjustY(player.y + frame * cnd.dy)
            if hitTest(player_body, obj_body) then
              cnd.hitrisk = cnd.hitrisk + (1/2)^frame
            end
          end
          -- 座標を元に戻す
          obj_body.x = obj.x
          obj_body.y = obj.y
          player_body.x = player.x
          player_body.y = player.y
        end
      end
    end
  end
end
```

以前の calculateHitRisk 関数と較べると for ループが 1 つ増えています。これは 1 フレーム先から N フレーム先までの各フレームで被弾リスクを計算するためです。敵、弾などはずっと同じ速度で移動すると仮定しています。なので frame フレーム後の位置は

```
          obj_body.x = obj_body.x + frame * obj.vx
          obj_body.y = obj_body.y + frame * obj.vy
```

のように速度の frame 倍を足した値になります。自機も N フレームの間同じ方向に移動し続けると仮定しているので敵などと同様に

```
            player_body.x = adjustX(player.x + frame * cnd.dx)
            player_body.y = adjustY(player.y + frame * cnd.dy)
```

のようになります。

frame フレーム先で敵や弾などとぶつかっていた場合は被弾リスクを $\frac{1}{2^{frame}}$ だけ足します。frame が大きければ被弾リスクの増加分は小さくなります。未来であればあるほど弾の位置などの

予測の確度が低いのでリスクの増加分も少なくしているのです。

```
            if hitTest(player_body, obj_body) then
              cnd.hitrisk = cnd.hitrisk + (1/2)^frame
            end
```

`calculateHitRisk` 関数では `frame` について for ループを回す前にそもそも被弾リスクの計算をすべきかどうかの判定を if 文で行っています。ここで呼び出している `shouldPredict` 関数では、いま注目している敵や弾が自機の近くにあるかどうかを調べます。

ソースコード 2.6　chap2-2/main.lua shouldPredict 関数

```
local function shouldPredict(obj_body, player_body)
  local prev_width, prev_height, prev_radius
  local size_rate = 10
  if obj_body.type ~= HitType.Circle then
    prev_width = player_body.width
    prev_height = player_body.height
    player_body.width = prev_width * size_rate
    player_body.height = prev_height * size_rate
  else
    prev_radius = player_body.radius
    player_body.radius = prev_radius * size_rate
  end
  local ret = hitTest(obj_body, player_body)
  if obj_body.type ~= HitType.Circle then
    player_body.width = prev_width
    player_body.height = prev_height
  else
    player_body.radius = prev_radius
  end
  return ret
end
```

この関数の中でやっていることは以下の通りです。

- 自機の当たり判定 (`player_body`) の大きさを `size_rate` 倍にする
- `obj_body` と当たり判定処理を行う
- `player_body` の大きさを戻した後、当たり判定処理の結果を返す

敵や弾が自機に近いかを調べるなら距離を求めてもよさそうですが、レーザーなど当たり判定の大きなものだと弾の位置座標は遠くても自機の近くまで当たり判定が届くことがあるので、当たり判定を使った判定の方が確実です。

先読みする AI のコードはこれでひと通り揃いったので実際に動かしてみましょう。1 フレームだけの先読みと較べてはるかに長生きします。弾を撃たないので勝つことはないですが。

前よりもよく避けるようになったとはいえ、まだまだ問題はあるでしょう。たとえば

- 動きが機械的
- 画面端に追い込まれがち
- 当たりそうになるまで動かない
- レーザーに当たりやすい[*4]

[*4] 原因は 2 つある。1 つはレーザーの予告線を検知できないという花 AI 塚そのものの問題、もう 1 つは弾のサイズの

- 射命丸の EX アタック (加速するやつ) に当たりやすい

こういった問題は弾の予測などのモジュールレベルの問題もあれば、消極的な回避 AI そのものの問題もあります。まずは小さくて直しやすい問題から取り組んでみてはいかがでしょうか。

Column: ログを取る

AI を書いていると変数の実際の値が見たくなることがあります。標準の Lua なら print 関数を呼ぶところですが、花 AI 塚には print 関数は用意されていません。花 AI 塚ではセキュリティや実装上の都合から標準ライブラリのいくつかが実装されていません。
花 AI 塚ではファイルの読み書きができるので、変数の値を出力したい場合はファイルに書き込みます。下のコードでは毎フレーム自機の X 座標を log.txt に書き込みます。

```
local f = io.open("log.txt","w")
function main()
  local player  = game_sides[player_side].player
  f:write(tostring(player.x).."\n")
end
```

ちなみに花 AI 塚では AI が動いている間も人間のキー入力操作を受け付けるので、この AI を動かした状態で人間がプレイすれば、人間の動きに沿った移動のログを取ることができます。
ファイルの読み書きができるファイルパスには制限があります。詳しくは花 AI 塚に付属のリファレンスマニュアルを参照下さい。

変化を考えていない弾予測の問題だ。レーザーが伸びたり太くなることを予想せずに留まり、被弾することが多い。

第 3 章

撃つ

花映塚は避けるだけでは勝てません。敵を撃ち、アタックを仕掛け、相手陣に弾の雨を降らせなければなりません。この章では攻撃を仕掛ける AI を作っていきます。

3.1 弾を撃つ

今まで自機の操作として移動操作のみを扱ってきましたが、これからは攻撃操作も扱っていきます。移動操作は sendKeys 関数でキー入力を送ることで実現しましたが、射撃などの攻撃操作も同様にして行います。

花映塚では射撃操作は射撃キー (キーボードだと Z キー) を連打して行います。射撃キーを押し続けるとチャージになってしまうので、押したり離したりを繰り返す必要があります。

射撃操作する AI は以下のコードで実現できます。

```
local pressed = false
function main()
  pressed = not(pressed)
  if pressed then
    sendKeys(1) -- 射撃キーは1bit目に対応する
  else
    sendKeys(0)
  end
end
```

ただ、このコードでは射撃のためにわざわざグローバル変数を作っていて、あまり綺麗ではありません。また、キーのオンオフを繰り返すループがループ構文の形をしていないのは不自然でもあります。AI のコードが増えると射撃キーのためのループが他の処理のコードに埋もれていき、ループであることが分かりづらくなるでしょう。ということで手直しが必要です。

花 AI 塚の AI を書くのに使っている Lua にはコルーチンというものがあります。コルーチンは関数のようなものですが、普通の関数と違って戻り値を return ではなく coroutine.yield で返します。そしてコルーチンが次に呼ばれるときには前回 yield した場所から処理が再開します。以下のコードは射撃操作する AI をコルーチンを使って書いたものです。

```
local generateShootKey = coroutine.wrap(function ()
```

```
  while true do
    coroutine.yield(1)
    coroutine.yield(0)
  end
end)

function main()
  local keys = generateShootKey()
  sendKeys(keys)
end
```

この generateShootKey の場合、呼び出すたびに 1 と 0 とが交互に返ってきます。キーのオンオフの繰り返しが generateShootKey の中のループとして切り出されたので先ほどより読みやすいと思います。

3.2　チャージアタックを撃つ

花映塚では射撃キー長押しでチャージし、チャージした状態で離すとチャージアタックとなります。チャージアタックも射撃キーを sendKeys 関数で送りつけることで実現できます。

チャージアタックを AI の操作に組み込むに当たって以下の情報が欲しくなるでしょう。

- いまチャージできる最大値はいくつか
- いまどこまでチャージしているか
- チャージにどれくらい時間がかかるか

これらはそれぞれ game_sides[player_side].player の以下のフィールドから分かります。

- player.currentChargeMax
- player.currentCharge
- player.chargeSpeed

currentChargeMax や currentCharge は 0〜400 の範囲の値で、100 以上で C1、200 以上で C2、300 以上で C3、400 で C4 が撃てるようになります。chargeSpeed は 1 フレームでどれだけチャージが溜まるかを持っています。

下の AI は C2 以上を撃てる時はチャージして C2 を撃つ、それ以外の時は通常弾を撃つというものです。チャージするかどうかを判定するために currentCharge や currentChargeMax を参照しています。

```
local generateShootKey = coroutine.wrap(function ()
  while true do
    coroutine.yield(1)
    coroutine.yield(0)
  end
end)

function main()
  local myside = game_sides[player_side]
  local player = myside.player
  local should_charge = false
```

```
  if player.currentChargeMax > 200 and -- C2以上チャージできる
     player.currentCharge < 200 then -- まだC2までチャージしてない
    should_charge = true
  end

  local keys = 0
  if should_charge then
    keys = 1 -- 射撃キー押しっぱなしにしとく
  else
    keys = generateShootKey()
  end
  sendKeys(keys)
end
```

花映塚ではボムに相当するものとしてチャージアタックの他に緊急ボムもありますが、これは緊急ボムのキー (X キー) を sendKeys 関数で送るだけで実現できます。

3.3　避けつつ撃つ

射撃やチャージアタックなどの攻撃と 2 章の回避アルゴリズムを組み合わせることで避けながら攻撃する AI が作れます。

ソースコード 3.1　chap3-1/main.lua main 関数

```
function main()
  local myside = game_sides[player_side]
  local player = myside.player
  -- 選択する移動操作の候補を生成
  local candidates = generateCandidates(player)
  -- それぞれの移動操作の被弾リスクを計算
  calculateHitRisk(candidates, myside.enemies, player)
  calculateHitRisk(candidates, myside.bullets, player)
  calculateHitRisk(candidates, myside.exAttacks, player)
  -- 被弾リスクが一番低い移動操作を選ぶ
  local choice = choose(candidates)

  local should_charge = false
  if player.currentChargeMax > 200 then -- C2以上チャージできる
    if player.currentCharge < 200 then -- まだC2までチャージしてない
      should_charge = true
    end
  end

  local keys = choice.key
  if should_charge then
    keys = keys + 1 -- 射撃キー押しっぱなしにしとく
  else
    keys = keys + generateShootKey() -- 射撃キー押したり離したりする
  end
  -- キー入力として送信
  sendKeys(keys)
end
```

main 関数は単に 2 章の main 関数と 3.2 節の main 関数を組み合わせたものです。射撃キーと移動操作キーを組み合わせるために keys = keys + generateShootKey() のように足し算をしています。本来は or のビット演算をしたいのですが、Lua にはビット演算が用意されていないので

足し算で代用しているのです[*1]。

AI の動きは「当たりそうなら避ける」、「C2 撃てる時は撃つ」、「それ以外は射撃し続ける」の 3 つだけですが、これでも Normal のマッチモード程度ならそこそこ勝てます。

3.4　狙った敵を撃ちに行く

避けつつ撃つ AI でもぼちぼちゲームはできるのですが、プレイを見ていると以下の問題点に気づきます。

- 射軸の近くに敵がいても倒さない
- 無駄にチャージアタックを撃ち続ける
- 敵の連鎖爆発を考慮していないので敵の隊列を分断してしまう

攻撃もむやみやたらにすればいいものではなく、タイミングやターゲットに狙いを定めるべきです。ここでは敵を狙って追いかけるように AI を改良していきます。

3.4.1　アルゴリズムを考える

敵を狙って追いかけて撃つ、とはどういうことなのか分解してみましょう。

1. 狙う敵を決める
2. 狙った敵に近づく
3. 狙った敵が射軸上に来たら撃つ

まずどの敵を狙うべきか、これ自体戦術としての幅があります。たとえば「一番近くの敵を狙う」、「隊列の先頭を狙う」、「連鎖爆発が繋がりそうな敵を狙う」といったものが考えられます。今回は簡単に、一番近くの敵を狙うことにします。

狙った敵に近づくシンプルな方法は、常に狙った敵との距離が縮む移動操作を選ぶというのがあります[*2]。つまり敵が自機より右にいれば右へ、自機より左にいれば左に動くというものです。ただ、STG では狙った敵に近づく以外にも敵や弾を避けないといけません。回避 AI の実装では各移動操作の被弾リスクを計算して、最も被弾リスクが小さい移動操作を選んでいました。この枠組を拡張して、被弾リスクと狙った敵までの距離を元にいい感じのスコアを計算して、スコアが最小の移動操作を選ぶことにします。

射軸上に来たかどうかは単純に自機の X 座標と狙っている敵の X 座標の差が小さいかどうかで判定できます。

[*1] `1 | 2` のように立っているビットが重なっていなければ or と足し算は同じ結果になる。逆に `1 | 3` のようにビットが重なっているとダメ。今回の AI ではビットが重ならないのでうまくいく。

[*2] 他にも敵を待ちぶせできる位置へ移動するという方法なども考えられる。

3.4.2　AI を書く

それでは AI のコードを書いていきます。まずは main 関数から。

ソースコード 3.2　chap3-2/main.lua main 関数

```
function main()
  local myside = game_sides[player_side]
  local player = myside.player

  -- 移動候補
  local candidates = generateCandidates(player)
  -- 追跡する敵
  local target_enemy = getCurrentTarget(player, myside.enemies)
  -- 追跡コスト計算
  calculateFollowingCost(candidates, player, target_enemy)
  -- それぞれの移動操作の被弾リスクを計算
  calculateHitRisk(candidates, myside.enemies, player)
  calculateHitRisk(candidates, myside.bullets, player)
  calculateHitRisk(candidates, myside.exAttacks, player)
  -- 総合的なコストが最小の手を選ぶ
  local choice = choosemin(candidates, function (candidate)
    return candidate.hitrisk * 10000 + candidate.followingCost
  end)

  local should_charge = false
  if player.currentChargeMax > 200 and
     player.currentCharge < 200 then
    should_charge = true
  end

  local keys = choice.key
  if should_charge then
    keys = keys + 1
  else
    -- 狙ってる敵と軸が合ったときに撃つ
    if target_enemy ~= nil and isNear(player, target_enemy) then
      keys = keys + generateShootKey()
    end
  end
  sendKeys(keys)
end
```

回避 AI のコードに以下の処理を組み合わせています。

- 現在追跡している敵を取得する (getCurrentTarget 関数。target_enemy に格納)
- target_enemy を追跡するコストを各移動操作について計算する (calculateFollowingCost 関数)
- 追跡コストと被弾リスクをもとに移動操作を選ぶ
- target_enemy が自機の射軸の近くにいたら射撃する

移動操作の選択には choosemin 関数を使っています。これは choose 関数を一般化したもので、引数 func には関数を取ります。func に candidates の各要素を食わせ、func の戻り値が最小になる要素を返すというものです[*3]。便利なので main 関数以外でも使っています。

[*3] 数学だと arg min に相当する。

ソースコード 3.3　chap3-2/main.lua choosemin 関数

```
local function choosemin(candidates, func)
  local min = 99999999
  local min_i = -1
  for i, cnd in ipairs(candidates) do
    local v = func(cnd)
    if  v < min then
      min = v
      min_i = i
    end
  end
  return candidates[min_i]
end
```

移動操作の選択ではそれぞれの移動操作のスコアを (被弾リスク) ∗ 10000 + (追跡コスト) で計算し、このスコアが最も小さい移動操作を選んでいます。今回のスコアの定義は被弾リスクを重視しています (なので被弾リスクの係数が大きいです)。追跡コストを重視すると敵を追いかけるために被弾リスクの高い移動操作を選ぶようになってしまうからです。

今回、追跡コストを計算するので `generateCandidates` 関数には追跡コストを格納する `followingCost` フィールドが追加されています。

ソースコード 3.4　chap3-2/main.lua generateCandidates 関数

```
local function generateCandidates(player)
  local candidates = {}
  local dxs = {0, 1, -1, 0, 0}
  local dys = {0, 0, 0, 1, -1}
  -- キー入力。停止、→、←、↓、↑
  local keys = {0x0,0x80,0x40,0x20,0x10}
  for i=1, #keys do
    candidates[i] = {
      key = keys[i],
      dx = player.speedFast * dxs[i],
      dy = player.speedFast * dys[i],
      followingCost = 0,
      hitrisk = 0
    }
  end
  return candidates
end
```

`getCurrentTarget` 関数は現在狙っている敵を返します。基本的には 1 つ前のフレームで狙っていた敵を返しますが、さっきまで狙っていた敵がいなくなった場合や 1 つ前のフレームでは敵を狙っていなかった場合は自機の射軸に一番近い敵を狙います。

ソースコード 3.5　chap3-2/main.lua getCurrentTarget 関数

```
local target_enemy_id = nil
local function getCurrentTarget(player, enemies)
  local target_enemy = findEnemyById(enemies, target_enemy_id)
  if target_enemy == nil then
    -- 今狙っている敵がいないときは、射軸に一番近い敵を選ぶ
    target_enemy = choosemin(enemies, function(enemy)
      -- 幽霊や擬似的な敵やボスは無視
      if enemy.isSpirit or enemy.isPseudoEnemy or enemy.isBoss then
        return 99999999
      end
      return (enemy.x - player.x)^2
    end)
  end
  if target_enemy ~= nil then
```

```
    target_enemy_id = target_enemy.id
  else
    target_enemy_id = nil
  end
  return target_enemy
end
```

1.3.2 節でも触れましたが、花 AI 塚から得られる敵の情報には ID が振られています。1 フレーム前に狙った敵の ID を覚えておけば、この ID を使って狙っている敵を enemies の中から探し出せるわけです。ID が target_enemy_id に一致する敵を見つけるために以下のような findEnemyById 関数を定義して使っています。

ソースコード 3.6　chap3-2/main.lua findEnemyById 関数

```
local function findEnemyById(enemies, id)
  for i, enemy in ipairs(enemies) do
    if enemy.id == id then
      return enemy
    end
  end
  return nil
end
```

getCurrentTarget 関数では射軸に近い敵を求めるために choosemin 関数を使っています。現在狙っている敵がいないときは、自機の射軸に近くてかつ幽霊や擬似的な敵やボスでない敵を次に狙う敵として選びます。幽霊やボスを無視しているのはこれらを倒すのに時間がかかり、その間他の敵を倒せないのがしんどいからです。擬似的な敵はそもそも倒せないので無視しています。

main 関数の解説に戻ります。狙う敵を target_enemy に格納した後、calculateFollowingCost 関数を呼んでいます。この関数では各移動操作について、target_enemy を追跡するのにかかるコストを計算します。今回追跡コストは単純に自機と target_enemy との X 方向の距離 (つまり射軸との距離) を使っています。

ソースコード 3.7　chap3-2/main.lua calculateFollowingCost 関数

```
local function calculateFollowingCost(candidates, player, target)
  for i, cnd in ipairs(candidates) do
    local dx = 0
    if target ~= nil then
      dx = target.x - adjustX(player.x + cnd.dx)
    end
    cnd.followingCost = math.abs(dx)
  end
end
```

また main 関数の解説に戻って、狙った敵が自機の近くに来た時の処理を見ていきます。

```
    -- 狙ってる敵と軸があったときに撃つ
    if target_enemy ~= nil and isNear(player, target_enemy) then
      keys = keys + generateShootKey()
    end
```

現在狙っている敵がいて (target_enemy~=null)、その敵が自機の射軸上にいるときに射撃を行うようにしています。射軸上にいるかの判定を isNear 関数で行っています。

ソースコード 3.8　chap3-2/main.lua isNear 関数

```
local function isNear(player, target)
  return (target.x - player.x)^2 < 32^2
end
```

isNear 関数は単に敵と自機の X 方向の距離が一定値以下かどうかを判定しています。

敵を追いかけながら避ける AI のコードはこれで以上です。

3.4.3　AI の戦果

実際にこの AI を動かしてみたところ、Normal のストーリーモードをクリアできました[*4]。この AI はまだまだ神主 AI と比較しても見劣りするところがあるでしょう。とはいえ STG AI の目標の 1 つを達成できたと思います。

花映塚 AI がこなすべきことはまだまだあります。この AI はまだアイテムを拾いに行きませんし、チャージアタックのタイミングもいい加減です。人間のプレイと較べて欠けている要素を少しずつでも AI に組み込んでいくとよいでしょう。

Column: 花 AI 塚の仕組み

花 AI 塚は自機や弾の位置など花映塚 AI を作るのに必要な情報を集める仕事をしています。花 AI 塚ではこういった情報を集めるために、花映塚のプロセスのメモリを解析しています。

一般にプログラムが使っているデータはメモリ上に配置されます。プログラムが使っているメモリを読むことができて、さらにメモリ上にどうデータが配置されるかが分かれば、そのプログラムが使うデータを正しく読み取ることができるのです。

花 AI 塚では DLL インジェクションを使って花映塚プロセスに寄生しています。こうすることで通常のポインタを介したアクセスで花映塚プロセスのメモリを読むことができます。また、メモリへの書き込みも通常のポインタと同様にできるので、自機操作の反映は自機操作関連のデータを直接書き換えて行っています。

花 AI 塚の配布元には花 AI 塚の仕組みや開発の経緯など技術資料がまとまっているので、詳しくはそちらを参照下さい。

[*4] 霊夢自機にて。最も苦戦したのは vs 射命丸文。

あとがき

本文 @ide_an： 花映塚に限った話ではないですが、AI はかわいいです。自分で作った AI ならなおさら。この本を通して花映塚 AI を愛でる趣味が広がれば幸いです。

表紙 @ide_an： 画力が欲しい。

花映塚 AI をつくる 花 AI 塚ではじめる STG AI プログラミング

発行日	2014 年 12 月 30 日
著者	@ide_an
サークル	いで庵 (http://usamimi.info/~ide/)
連絡先	ideanweb@gmail.com
印刷所	株式会社 栄光