

Common Lisp
何れも#Lisp
? (trapean@)
Common Lisp
何れも#Lisp
? (trapean@)

目次

はじめに	3
第 1 章 開発環境の構築	5
1.1 処理系の選択	5
1.2 SBCL のインストール	5
1.3 Lispbuilder-SDL	5
1.4 Lispbuilder-SDL のインストール (Windows)	6
第 2 章 ゲームプログラムの第一歩	8
2.1 ゲームループ	8
2.2 最初の最小のプログラム	8
2.3 まとめ	9
第 3 章 画面に描く	10
3.1 描画するプログラム	10
3.2 まとめ	11
第 4 章 キー入力を拾う	12
4.1 まずはキーを拾ってみる	12
4.2 キー入力を管理する	13
4.3 抽象化を被せる	15
4.4 まとめ	16
第 5 章 「絵を動かす」から「自機を動かす」へ	17
5.1 絵を動かす	17
5.2 さっきのプログラムのまずさと対策	20
5.3 自機を作る	20
5.4 ゲーム画面クラスを作る	23
5.5 自機を動かす	24
5.6 まとめ	25
第 6 章 敵を出す	26
6.1 敵を作る	26
6.2 敵を動かす	27
6.3 敵を動かす (いっぱい)	29
6.4 成仏させる	32
6.5 まとめ	33

第 7 章	当たり判定	34
7.1	当たり判定の理屈	34
7.1.1	円形の当たり判定	34
7.1.2	矩形の当たり判定	35
7.1.3	どっちがいいの?	36
7.2	当たり判定を作る	36
7.3	当たり判定を埋め込む	37
7.4	当たり判定する	41
7.5	まとめ	43
第 8 章	弾を撃つ	44
8.1	引き金を引くのは誰?	44
8.2	弾を作る	44
8.3	弾を管理する	46
8.4	引き金を引かせる	48
8.4.1	自機弾の実装	48
8.4.2	敵弾の実装	49
8.5	撃たれる準備	51
8.6	弾を撃つ	51
8.7	まとめ	54
第 9 章	ローディングとステージ	55
9.1	画像読み込みの管理	55
9.1.1	なんで管理したいの?	55
9.1.2	管理しよう	56
9.2	ステージ	57
9.2.1	考え方	57
9.2.2	実装	57
9.3	ローディングとステージを使う	59
9.4	まとめ	60
Q and A		61

はじめに

この文書について

この文書は『「ゲームを作りたくてプログラミング言語を勉強しているけどどうやってゲームを作ればいいのか分からない人のための導入冊子」を作るために上級生からアドバイスを貰うためのプロトタイプ』です¹。ということで初心者の方がいきなりこの文書を読むのはお勧めしません。逆にゲーム作成の経験を積んでいる上級生の方にはこの文書を読んでニヤニヤするだけじゃなくて「もっとこの辺の文章を分かりやすくした方がいい」などのアドバイスを投げてくださいと助かります。

なぜ Common Lisp?

さっき言ったようにこの文書はプロトタイプです。いちいち時間かけてられません。だったらさっさと作れる言語がいいじゃないですか！ということで Common Lisp というチョイスです。

一応もう少しマシな理由もあるにはあります。例えば、

「どの言語ならゲームを作れますか」という質問を潰すため

ありますよね、こういう質問。ぶっちゃけ描画手段さえ確立されていれば大概の言語でゲームが作れます。僕も JavaScript でゲーム作ってますし。

でも世の中に出回ってるゲーム製作本って大抵 C/C++ で書いてますよね。そのせいなのかどうかはともかく、まるで特定の言語じゃないとゲームを作れないと思われている気がしてならないのです。その既成概念をぶち壊せそうなチョイスとして Common Lisp はありだと思えます。

上級生も知らない言語の方が的確なアドバイスを貰えそう

何か知識を教えるときは『知ってる人』に教えるよりも『知らない人』に教える方が分かりづらいたころがはっきりする、と考えました。

知っている言語をサンプルに使うと文章での解説が足りないのにコードだけで言いたいことが伝わってしまって、『初心者にとって』分かりにくいところを見逃してしまうのではないかと思います。その点、知らない言語をサンプルにすれば文章での解説を頼りに読むので分かりにくいところがはっきりするのではないかと思った次第です。

といった具合です。なんだか眉唾な感じですが。

ここまでのところ原稿書くのに都合がいいから Common Lisp 使ってるって話しかけてないので Common Lisp 自体のメリット/デメリットも書いてきましょう。

メリット

LL 並に書きやすく, LL の 10 ~ 100 倍速く動く

処理系にもよるけど普通に機械語レベルにコンパイルされるのでそこの LL より速いです。さすがに普通に書いて C/C++ 並のスピードは出せないけど、型指定したり実行時の型チェック外したりだのして C 並に危険なコードにすると C/C++ 並の速度が出せるらしいです。

¹ さっそく括弧が多くて幸先のいいスタートですね。

処理系のバージョン違いに振り回されにくい

2.x 系と 3.x 系の非互換に苦しむ pythonista やよく知らないけど苦しんでるらしい rubyist の方ならこのありがたさが分かりますよね？LL な言語は大抵言語開発者の独断で言語仕様が決まっています。一方 Common Lisp の言語仕様は ANSI の規格として定められていて、そうそう変わることはないです。なので安心して処理系をアップデートできる … はずです。

デメリット

キモい

ですよね。

身の回りに頼れる lisper がいない

困ったときに誰か詳しい人に質問できないのは悩ましいですね。

いたけど変態だった

ですよね。

日本語の資料が少ない

日本語の書籍で現在手に入るものと言ったら片手で数えられる程度でしょう。ネット上の資料も少なくましてゲーム製作となるとねえ …。

schemer の視線が怖い

これに対する対策は 2 つ。schemer の主張を論破して Common Lisp の優位性を示すか、scheme が好きだけど仕方なく Common Lisp を使ってるアピールするかです。

デメリットの方が多いですね。でも書きやすくて速いってのは十分魅力的だと思いますけど。

開発環境について

この文書では Common Lisp と SDL を使ってゲームを作ります。どちらもマルチプラットフォームなので同じソースで複数の OS をサポートできるはずですが、

参考までに僕の環境ですが、

- Windows7(64bit), SBCL 1.0.37(32bit)
- Ubuntu 10.04 LTS, SBCL 1.0.29

といった感じです。

エディタは Vim を使ってますが Lisp 系言語との相性で言うと Emacs の方がいいというのが定説です²。なんでも Emacs と SLIME っていうのを使うと対話処理系がいい感じになったり補完とかいい感じになるらしいです。一応 Vim にも slimv.vim というプラグインがあるらしいです。他のエディタはどうなんでしょうか、少なくとも括弧に対する処理や自動インデントが強いエディタじゃないと Lisp を書くのは厳しいと思います。

²でも Lisp 界で有名なポール・グレームは vi で書いてるらしいので Vim でも頑張れるでしょう、きっと。

第1章 開発環境の構築

Common Lisp でゲームを作ろう、ということではまず開発環境を用意しましょう。それには Common Lisp の処理系と SDL のバインディングライブラリが必要です。ではさっそく環境を作っていきます。

1.1 処理系の選択

Common Lisp の処理系はいくつかあります。SBCL, CLISP, Closure CL, CMUCL などいっぱいありますね。で、どの処理系がいいかという話ですが、SBCL と CLISP が割とメジャーです。加えてこの 2 つは Windows もサポートしているということでこの 2 つのどちらかでいいでしょう。今回は SBCL を採用します。

1.2 SBCL のインストール

Windows の場合は公式サイト (<http://www.sbcl.org/>) からインストーラ落として動かせば普通にインストールできます。ただ、C:\Program Files 以下にインストールすると後でアクセス制御まわりがめんどいので他のディレクトリにインストールした方がいいです¹。

コマンドプロンプトや Power Shell を立ち上げて

```
sbcl
```

と打ち込んでプロンプトが立ち上がれば OK です。

```
(quit)
```

と打ち込むと終了します。

Linux の場合はバイナリパッケージを落とせばよいでしょう。ソースからコンパイルする場合は別途 Common Lisp 処理系が必要になるので注意。

1.3 Lispbuilder-SDL

今回ゲームを作るにあたって SDL というライブラリを使います。これは DirectX や OpenGL など各種プラットフォームでのゲーム作成関連 API をラップしているライブラリです。SDL 自体は C 言語用のライブラリですが、様々な言語へのバインディングが存在します。

SDL の Common Lisp 向けのバインディングとしては Lispbuilder-SDL があります。今回はこれを使います。

¹僕の場合は C:\usr\bin\sbcl にインストールしました。

1.4 Lispbuilder-SDL のインストール (Windows)

今回は Lispbuilder-SDL(公式サイト<http://code.google.com/p/lispbuilder/wiki/LispbuilderSDL>) というライブラリを使います。このライブラリを Windows 上でインストールする手順を見ていきます²。

まずは SBCL をインストールしたディレクトリを確認しておきます。以降では SBCL をインストールしたディレクトリを [SBCL] と表記します。

次に Lispbuilder-SDL が依存しているライブラリをダウンロードします。必要なのは

CFFI

<http://common-lisp.net/project/cffi/releases/?M=D> から `cffi_*.***.tar.gz` をダウンロード。最新版で OK。

Alexandria

<http://www.cliki.net/Alexandria> から `alexandria-****-**-*.tar.gz` をダウンロード。

Babel

<http://www.cliki.net/Babel> から `babel_latest.tar.gz` をダウンロード。

trivial-features

<http://www.cliki.net/trivial-features> から `trivial-features_latest.tar.gz` をダウンロード。

です。これらの tar.gz ファイルを解凍して [SBCL]\site 以下に置いて下さい。site フォルダがなければ作って下さい。

次に Lispbuilder-SDL のダウンロードページ(<http://code.google.com/p/lispbuilder/downloads/list>) から 10 個のファイルをダウンロードします³。

- win32-lispbuilder-sdl-gfx-binaries-2.0.13.tgz
- win32-lispbuilder-sdl-ttf-binaries-2.0.9.tgz
- win32-lispbuilder-sdl-mixer-binaries-1.2.11.tgz
- win32-lispbuilder-sdl-image-binaries-1.2.10.tgz
- win32-lispbuilder-sdl-binaries-1.2.14.tgz
- lispbuilder-sdl-ttf-0.3.0.tgz
- lispbuilder-sdl-mixer-0.4.tgz
- lispbuilder-sdl-image-0.5.0.tgz
- lispbuilder-sdl-gfx-0.7.0.tgz
- lispbuilder-sdl-0.9.8.1.tgz

これらを全部展開して [SBCL]\site 以下に置いて下さい。ファイル上書きに関するダイアログが出るかもしれませんが構わず上書きして OK です。

諸々のファイルを置いていくと [SBCL]\site 以下のフォルダ構成は

²Linux ユーザーは自力で頑張れ。手順は公式サイト参照。CFFI のインストールは apt-get あたりから cl-cffi を入れるのがベターらしい。

³この冊子で触れる範囲だったら gfx とか ttf とか mixer とかはいらん気がしますが一応念のため …。

```
[SBCL]\site\alexandria\  
[SBCL]\site\babel_*.*.\  
[SBCL]\site\cffi_*.**.\  
[SBCL]\site\lispbuilder-sdl-gfx\  
[SBCL]\site\lispbuilder-sdl-image\  
[SBCL]\site\lispbuilder-sdl-mixer\  
[SBCL]\site\lispbuilder-sdl-ttf\  
[SBCL]\site\lispbuilder-sdl\  
[SBCL]\site\trivial-features_*.*.\  

```

のようになります。

これで必要なファイルはすべてインストールされましたが、SBCL を起動した時に読み込むようにするためにもう少し設定が必要です。以下の内容を記述したファイルを `sbclrc` というファイル名で [SBCL] 直下に保存して下さい。

List 1.1: `sbclrc`

```
(require :asdf)  
;; [SBCL]はSBCLをインストールしたディレクトリ  
(dolist (dir (directory "[SBCL]\\site\\*\\"))  
  (pushnew dir asdf:*central-registry* :test #'equal))  
  
;;load listbuilder-sdl  
(asdf:operate 'asdf:load-op :lispbuilder-sdl)  
(asdf:operate 'asdf:load-op :lispbuilder-sdl-binaries)
```

これでインストールは完了です。インストールがうまくいったか確認するには SBCL を起動して

```
(asdf:operate 'asdf:load-op :lispbuilder-sdl-examples)  
(sdl-examples:mandelbrot)
```

と打ち込めば OK です。正しくインストールされればサンプルアプリが動作します。

ロード時間を短縮する

これで Lispbuilder-SDL が使えるので OK と言えば OK なのですが、SBCL を起動するたびにライブラリのロードに時間を食われて精神的によろしくなかったりします。これを解決するためにコアファイルを生成して、SBCL を起動する際にはコアファイルを指定するようにします。

コアファイルを生成するには SBCL を起動して

```
(save-lisp-and-die  
  (merge-pathnames "main.core" (user-homedir-pathname))) :purify t)
```

と打ち込めば `C:\Users\ユーザー名\直下` に `main.core` というファイルが生成されます。

次回以降 SBCL を起動する際にはオプションで `--core C:\Users\ユーザー名\main.core` を指定すればコアファイルが読み込まれ、ライブラリのロードにかかる時間が短縮できます。やったね！

参考文献

この章は@lambda.sakura さんの記事 (<http://d.hatena.ne.jp/sakura-1/20110112>) と千葉大学松田研の記事 (<http://www.math.s.chiba-u.ac.jp/~matsu/cl/sbcl.html>) を参考に書きました。この場を借りて感謝します。

第2章 ゲームプログラムの第一歩

ゲーム作成のための環境がそろったのでさっそくゲームを作っていきます。まずはゲームプログラムってどんな構造なのか見ていきます。

2.1 ゲームループ

ゲームプログラムに最低限必要なものってなんでしょうか？それは「ユーザの入力に即座に反応すること」です¹。なんだかゲームに限らずどのプログラムにも当てはまりそうな気がします、案外そうでもないです。Linux のコマンドラインのツールの類は起動してから終了するまでユーザの入力を受け付けずに動きますよね²。GUI のプログラムも入力に対して即座に反応するとは限りませんよね。テキストボックスに文字を入力してもボタンを押すまで反応しなかったりとか。

そんな訳でゲームプログラムは入力に即座に反応するための構造を持っています。それがループです。ゲームプログラムの中心は以下の処理の繰り返しです。

- ユーザの入力を受け取る
- ゲームの状態を更新する
- ユーザへの出力を更新する

ユーザの入力というのはキー入力やマウス操作、ゲームパッドの入力などですね。ユーザへの出力は画面の表示や効果音の再生などです。こういったゲーム中のループ処理はゲームループと言います。ゲームプログラミングの全てはゲームループの中で行う3つのことと、ループに入る前の初期化処理に行き着くと言っていいでしょう。

2.2 最初の最小のプログラム

とりあえず Common Lisp でコードを書いてみましょう。ということで一番基本のプログラムを用意しました (List2.1)。このプログラムを実行するには SBCL を起動して出てくるプロンプトで

```
(load "main.lisp" :external-format :utf-8)
(main)
```

と打てば OK です。これでファイル “main.lisp” をロードして、関数 main を呼びます。ファイル名や文字コード³の指定は各自で合う値にしてください。

さて、これを実行したらどうなるかという真っ黒な画面のウィンドウが出てくるってそれだけです。まあたった 10 数行でできることなんてそんなもんですよね⁴。Esc キーを押すか終了ボタンを押すとウィ

¹Conway のライフゲームはどうなの？って話は無しの方で。

²シグナルがどうこうって話は無しの方で。

³正しくは文字エンコーディングですね。SBCL の場合 UTF-8 なら :utf-8、Shift_JIS なら :cp932、EUC-JP なら :euc-jp に設定します。この辺は Common Lisp 処理系ごとに違うらしいので注意。

⁴ちなみにこれと同じことを C 言語で直接 Win32API とか DirectX とか叩いてやると 100 行越えます。

List 2.1: main.lisp

```

1 (defun main ()
2   (sdl:with-init ()
3     (sdl:window 640 480 :title-caption "てすと") ; ウィンドウをつくる
4     (setf (sdl:frame-rate) 60) ; FPSを60に
5
6     (sdl:update-display) ; 画面更新
7
8     ;; ここからイベント処理
9     (sdl:with-events ()
10      (:quit-event () t)
11      (:key-down-event (:key key) ; キーを押したときの処理
12       (when (sdl:key= key :sdl-key-escape)
13         (sdl:push-quit-event)))
14      (:idle ()
15       ;; この中がゲームループ、今は空っぽだけ。
16       (sdl:update-display))))))

```

ンドウが閉じます。ウィンドウを閉じるとプロンプトに処理が戻るので気が済んだら (quit) で終了しましょう。

それではプログラム (List2.1) の中身を見ていきましょう。まず 2 行目の `sdl:with-init` ですが、これは SDL ライブラリの初期化と終了時の処理を一手に引き受けてくれるマクロです。3 行目で `sdl:window` で幅 640px, 高さ 480px のウィンドウを生成します。4 行目では FPS を 60 に設定しています。FPS の管理も SDL 側で面倒を見てくれるわけですね。5 行目の `sdl:update-display` 関数の呼び出しでとりあえず画面を更新しておきます。

9 行目からが本丸のコードです。ユーザが何か操作をしたりするとイベントというものが発生します。そのイベントを処理するのが 9 行目以降のコードです。イベントには種類があるのでそれぞれの種類に応じた処理をしていきます。まず 10 行目では終了ボタンを押すなどで終了要求のイベントが来たときの処理です。この処理は `t` を返すだけで、そのままプログラムを終了させます。11~13 行目はキーボードが押されたときに発生するイベントに対する処理です。押されたキーが Esc キーかどうか調べて、Esc キーのときは終了要求のイベントを投げます。

14 行目以降がこのプログラムのゲームループです。プログラムが起動している間は定期的にこの処理が実行されます。今回はまだ何もせず画面を更新するだけですが、これから先ゲームを作っていくときはここにゲームの状態更新や描画の処理を書いていきます。

このプログラムの内容はこれで以上です。

2.3 まとめ

この章のまとめはこんな感じでどうでしょうか。

- ゲームプログラム = 初期化 + ゲームループ
- ゲームループ = 「入力を受け取る」 + 「状態を更新する」 + 「出力する」

画面が黒いままでさみしいので次の章では画面に絵を描いていきます。

第3章 画面に描く

ゲームと名乗るならグラフィックが欲しいですね。ということでこの章では描画する方法を見ていきます。

3.1 描画するプログラム

ひとまずプログラムを用意しました (List3.1)。

このプログラムも前の章でやったのと同じ手順で動きます。実行してみると図 3.1 のように表示されます。Esc を押すと終了します。図形や画像、テキストを描画してますね。

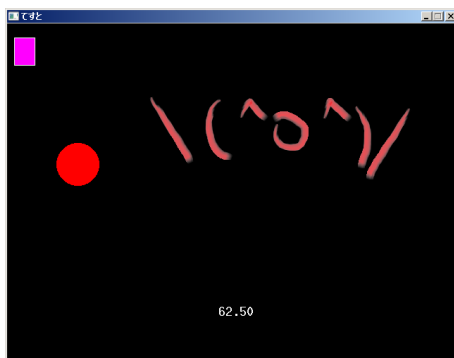


図 3.1: main.lisp の実行結果

List 3.1: main.lisp

```
1 (defun load-png-image (source-file)
2   (sdl:convert-to-display-format :surface (sdl:load-image source-file)
3     :enable-alpha t
4     :pixel-alpha t))
5
6 (defun main ()
7   (sdl:with-init ()
8     (sdl:window 640 480 :title-caption "てすと")
9     (setf (sdl:frame-rate) 60)
10    (sdl:initialise-default-font sdl:*font-10x20*) ; フォント初期化
11    (let ((img (load-png-image "test.png"))) ; 画像ファイルをロード
12
13      (sdl:update-display)
14
15      (sdl:with-events ()
16        (:quit-event () t)
17        (:key-down-event (:key key)
18          (when (sdl:key= key :sdl-key-escape)
19            (sdl:push-quit-event)))
20        (:idle ()
21          ; ; 描画する前に前に書いたものを消します
```

```


22     (sdl:clear-display sdl:*black*)
23     ;; 四角形を描画
24     (sdl:draw-box-* 10 ; 左上頂点のX座標
25                    20 ; 左上頂点のY座標
26                    30 ; 幅
27                    40 ; 高さ
28                    :color sdl:*magenta* ; 中の色
29                    :stroke-color sdl:*white*) ; 辺の色
30     ;; 円を描画
31     (sdl:draw-filled-circle-* 100 ; 中心のX座標
32                               200 ; 中心のY座標
33                               30 ; 半径
34                               :color sdl:*red*) ; 中の色
35     ;; 画像描画
36     (sdl:draw-surface-at-* img ; 画像
37                               200 ; 左上頂点のX座標
38                               100) ; 左上頂点のY座標
39     ;; 文字列描画
40     (sdl:draw-string-solid-* ;; FPSを取得して文字列に変換
41                               (format nil "~,2F" (sdl:average-fps))
42                               300 ; 左上頂点のX座標
43                               400) ; 左上頂点のY座標
44     (sdl:update-display))))))

```


今回のプログラムは前の章のプログラムに

- リソースの初期化 (画像の読み込みやフォントの初期化)
- ゲームループ中の描画処理

を追加したものです。では順番に見て行きましょう

まずリソースの初期化について見ていきます。ゲームループ中の描画処理では画像の描画とテキストの描画を行っていますが、これらを行うにはそれぞれ画像のロードとフォントの初期化が必要です。10行目ではテキストの描画に使うデフォルトのフォントを横10×縦20ピクセルのビットマップフォントに初期化します¹。11行目ではPNG画像ファイル“test.png”を読み込んで²変数に格納しています。画像を描画する際にはこの変数に格納した画像データを描画します。

画像を読み込む際に使っている関数 `load-png-image` は1~4行目で定義しています。このプログラムではアルファチャンネルを使用したPNG画像をロードしているのですが、この関数では画像のロードとアルファチャンネルを有効にする処理を行なっています。今後も画像をロードするときはこの関数を使っていきます。

次に描画処理について見て行きましょう。描画を行う前に21行目で画面を黒一色にクリアします。これによって前の時点で描画したものを消しています。24~29行目では四角形を描画しています。指定しているパラメータの意味はコメントを見れば明らかでしょう。補足しておく座標や長さの単位はすべてピクセルで整数値を指定します。座標は描画領域の左上を原点としてX軸は右方向が正、Y軸は下方向が正です。31~34行目は円の描画です。この関数は中の塗りつぶしだけやって縁取りはしません。36~38行目ではにロードした画像を描画しています。40~43行目ではこのプログラムのFPSをテキストに変換して描画しています。`sdl:average-fps` 関数で平均FPSを実数値で取得します。それを `format` 関数で小数点以下2桁までの表記の文字列に変換して描画しています。

3.2 まとめ

描画の方法をひと通り見てきました。次はユーザの入力を拾っていきます。

¹ここで設定したフォントはASCII文字だけをカバーしているので日本語のテキストは表示できません。

²ここで平然と (bmpではなく)PNGを読み込めるのはSDLImageの御利益です。

第4章 キー入力を拾う

前の章で描画の方法を学んだのでユーザへの出力はもうできますね。次はユーザから入力を拾いたいですね。ということでここではキーボードからの入力を拾う方法を見ていきます。

4.1 まずはキーを拾ってみる

キー入力を拾う話ですが、実は2章で書いたプログラムでキー入力を拾うコードを書いています。2で書いたプログラムではEscキーを押したときにプログラムを終了させる処理を書いています。あのときどうやってキー入力を拾ったかということ、イベント処理でキーを押したときのイベントに対して処理を書いていた。ということでキー入力の処理はキーに関するイベント処理で扱えばいいわけです。

それではキー入力を拾うプログラムを書いていきましょう。

List 4.1: main.lisp

```
1 (defun main ()
2   (sdl:with-init ()
3     (sdl:window 640 480 :title-caption "てすと")
4     (setf (sdl:frame-rate) 60)
5     (sdl:initialise-default-font sdl:*font-10x20*)
6     (let ((current-key nil))
7
8       (sdl:update-display)
9
10      (sdl:with-events ()
11        (:quit-event () t)
12        (:key-down-event (:key key) ; キーを押したときの処理
13          (if (sdl:key= key :sdl-key-escape)
14              (sdl:push-quit-event)
15              (setf current-key key))))
16        (:key-up-event () ; キーを放したとき処理
17          (setf current-key nil))
18        (:idle ()
19          (sdl:clear-display sdl:*black*)
20          ; どのキーが押されたか表示
21          (sdl:draw-string-solid-* (format nil "~A" current-key) 10 20)
22          (sdl:update-display))))))
```

このプログラム (List4.1) では現在押しているキーを画面に文字列で表示します。

キー入力を管理するためにkey-down-eventとkey-up-eventのイベントを監視します。key-down-eventはキーを押した時、key-up-eventはキーを放した時に発生するイベントです。キー入力のイベント処理では

```
(: イベント名 (:key 変数名)
; ; イベントの処理
)
```

という風にイベントの引数として (:key 変数名) を指定すると指定した変数名の変数にイベントを発生させたキー (key-down-event なら押したキー、key-up-event なら放したキー) の種類が格納されます。このプログラムではキーを押したときにそのキーが Esc キーの場合はプログラムの終了処理をして、そうでなければ変数 current-key に格納しています。キーを放したときは current-key を nil に戻しています。

さて、このプログラムですが1つ問題があります。2つのキーを同時押ししてみましょう。で、片方のキーを放してみると表示が nil になります。まだもう片方のキーが押されているのにこれでは困ります。

この問題の原因は1つの変数だけでどのキーを押しているかを管理しようとしているところです。1つの変数で単純に管理しようとするると同時に1つのキーしか管理できません。ゲームでの操作では同時押しが発生することが多いのでこれではダメですね。

ということで同時に複数のキーを管理するための仕組みを作る必要があります。

4.2 キー入力を管理する

複数のキーを管理したい訳ですが実現する方法は色々あります¹。今回は CLOS²でキー入力を管理するクラスを作ることになります。

実際にクラスを定義する前にどんなクラスがいいか、どういう風に使えるようにすればいいかを考えていきましょう。

ここで1つ考えておくべきことがあります。それはゲームの状態更新の部分を書いていく際には「どのキーを押したか」よりも「どの操作のキーを押したか」を知りたくなる、ということです。例えば Z キーを押すと弾を撃つという仕様でゲームを作っていたとしましょう。弾を撃つ処理を書くときに知りたいのは「弾を撃つ操作のキーを押したか」であって「Z キーを押したか」ではないということです。ゲームのロジックを書く際にはどの操作がどのキーに対応しているかを気にしなくてもいいように書けるのが望ましいです。そこで今回作成するクラスではどのキーがどの操作に割り当てられるかも管理して、キーの状態を知るときには「どの操作のキーが押されたか」を調べるような設計にします。

これらのことを念頭においてキー入力の管理クラスを作ってみました (List4.2)。

List 4.2: key-state.lisp

```
1 ;; 総称関数
2 (defgeneric update-key-state (key key-press key-state)
3   (:documentation "キーの状態を更新する"))
4
5 ;; キー入力の状態クラス
6 (defclass key-state ()
7   ((right
8     :initform nil
9     :documentation "右移動キーの状態")
10    (left
11     :initform nil
12     :documentation "左移動キーの状態"))))
13
14 ;; key-stateクラスのメソッド実装
15 (defmethod update-key-state (key key-press (key-state key-state))
16   (with-slots (right left) key-state
17     (cond ((sdl:key= key :sdl-key-right)
18            (setf right key-press))
19           ((sdl:key= key :sdl-key-left)
20            (setf left key-press)))))
```

¹ハッシュテーブルを使う、属性リストを使う、構造体を使うなどなど。

²Common Lisp Object System. Common Lisp でオブジェクト指向なコードを書く仕組みですね。

今回は「右移動のキー」と「左移動のキー」を管理するクラスということで作りました。6~12行目が入力管理クラスの定義です。各スロット³が各操作キーに対応していて、今そのキーが押されているかどうかを真偽値で持っています。15~20行目がキー入力の状態を更新するメソッド update-key-state の実装です。どのキーの状態が変わったか (key)、どう変わったか (押されたかどうかの真偽値 key-press) とキー入力管理クラスのオブジェクトを引数に取ります。cond で条件分岐して キーが押されたら right スロットを更新、 キーが押されたら left スロットを更新するようにしています。

このキー入力管理クラスを使ってプログラムを組んでみました (List4.3)。

List 4.3: main2.lisp

```

1 (load "key-state.lisp" :external-format :utf-8)
2
3 (defun main ()
4   (sdl:with-init ()
5     (sdl:window 640 480 :title-caption "てすと")
6     (setf (sdl:frame-rate) 60)
7     (sdl:initialise-default-font sdl:*font-10x20*)
8     (let ((current-key-state (make-instance 'key-state)))
9
10      (sdl:update-display)
11
12      (sdl:with-events ()
13        (:quit-event () t)
14        (:key-down-event (:key key) ; キーを押したときの処理
15          (if (sdl:key= key :sdl-key-escape)
16              (sdl:push-quit-event)
17              (update-key-state key t current-key-state)))
18        (:key-up-event (:key key) ; キーを放したとき処理
19          (update-key-state key nil current-key-state))
20        (:idle ()
21          (sdl:clear-display sdl:*black*)
22          ; どのキーが押されたか表示
23          (with-slots (right left) current-key-state
24            (sdl:draw-string-solid-* (format nil "right::~[no~;yes~]" right)
25                                     10 20)
26            (sdl:draw-string-solid-* (format nil "left::~[no~;yes~]" left)
27                                     10 40))
28          (sdl:update-display))))))

```

この章の始めで書いたプログラム (List4.1) との違いを見ていきます。

- 8行目で current-key の代わりに current-key-state を定義し、key-state クラスのオブジェクトで初期化している
- 17行目で current-key にキーの種類を setf する代わりに update-key-state 関数に t 渡してを呼んでいる
- 19行目で current-key に nil を setf する代わりに update-key-state 関数に nil を渡してを呼んでいる

違いを上げるとこんな感じでしょう。23~25行目でキーの状態を見る際には current-key-state の各スロットにアクセスしています⁴。

このプログラムを実行してみると キーと キーの同時押しにも対応できていることが分かります⁵。

³C++でいうメンバー変数、Java でいうフィールドです。

⁴CLOS ではスロットのカプセル化は行いません。なので with-slots マクロを使ってスロットを参照したりできます。

⁵キーボード自体が同時押しに対応していない可能性もありますが。

4.3 抽象化を被せる

さっきの節ではキー入力を管理するクラスを作ってみました。機能的には問題ないようですが、メンテナンス性はどうでしょうか。

さっきの例では2つのキーだけを管理していましたが、もっと多くのキーを管理してみましょう。

```
1 ;; キー入力の状態クラス
2 (defclass key-state ()
3   ((right
4     :initform nil
5     :documentation "右移動キーの状態")
6    (left
7     :initform nil
8     :documentation "左移動キーの状態")
9    (up
10    :initform nil
11    :documentation "上移動キーの状態")
12   (down
13    :initform nil
14    :documentation "下移動キーの状態"))))
15
16 ;; key-stateクラスのメソッド実装
17 (defmethod update-key-state (key key-press (key-state key-state))
18   (with-slots (right left up down) key-state
19     (cond ((sdl:key= key :sdl-key-right)
20            (setf right key-press))
21           ((sdl:key= key :sdl-key-left)
22            (setf left key-press))
23           ((sdl:key= key :sdl-key-up)
24            (setf up key-press))
25           ((sdl:key= key :sdl-key-down)
26            (setf down key-press))))))
```

新たに up と down を追加しそれぞれ キーと キーの状態を管理するようにしました。

クラス定義を書き足すだけではなく、メソッド update-key-state の定義も書き足す必要があります。今のキー入力管理クラスの実装では管理するキーが1つ増えるたびに5行ずつ⁶コードが増えていきます。メソッド定義で使っている with-slots マクロの引数にも追加が必要です。そう、今のキー入力管理クラスは管理するキーの追加に関してメンテナンス性が悪いのです。

メンテナンス性を改善するにはどうすればいいか、たいていの言語では効率を犠牲にして抽象化の層を追加するという解決になります。が、今回は Lisp です。Lisp にはマクロがあります。ということでマクロを使って効率を落とさずに抽象化の層を追加していきます。

マクロはコードを生成するプログラムです。今回はキー入力管理クラスの名前と「スロット名と対応するキーの組」のリストを受け取ってクラス定義のコードとメソッド update-key-state のコードを生成するマクロを作ることによってキー入力管理のメンテナンス性を上げていきます。

ということで書いてみたのがプログラム (List4.4) です。

6~15行目がキー入力管理クラスの定義とメソッド update-key-state の定義を生成するマクロです。18~20行目では定義したマクロを使って List4.2 で定義したのと同様のキー入力管理のコードを生成しています。List4.3 のプログラムで key-state.lisp の代わりに key-state2.lisp を読み込んで同じように動作します。

定義したマクロ defkeystate の実装の詳細については今すぐ理解できる必要はないでしょう⁷。ただ、このマクロが本当に正しく動いているのか気になるひともいるでしょう。そういうときは macroexpand-1 関数を使って確かめてみましょう。プロンプトで

⁶クラス定義で3行、メソッド定義で2行

⁷このコードを書いた本人もすんなりとは読めないです。

List 4.4: key-state2.lisp

```

1 ;; 総称関数
2 (defgeneric update-key-state (key key-press key-state)
3   (:documentation "キーの状態を更新する"))
4
5 ;; キー入力状態クラス用のマクロ
6 (defmacro defkeystate (name &rest key-maps)
7   '(progn
8     (defclass ,name ()
9       ,(loop for k in key-maps collect '(,(car k) :initform nil)))
10      ,(let ((key (gensym)) (key-press (gensym)) (key-state (gensym)))
11          '(defmethod update-key-state (,key ,key-press (,key-state ,name))
12            (with-slots ,(mapcar #'car key-maps) ,key-state
13              (cond ,(loop for k in key-maps
14                           collect '((sdl:key= ,key ,(cadr k))
15                                       (setf ,(car k) ,key-press))))))))))
16
17 ;; キー入力の状態クラス
18 (defkeystate key-state
19   (right :sdl-key-right)
20   (left :sdl-key-left))

```

```

(macroexpand-1 '(defkeystate key-state
                  (right :sdl-key-right)
                  (left :sdl-key-left)))

```

と打ち込めば List4.2 で定義したクラス定義やメソッド定義のコードとほぼ同じものが得られるはず⁸。

このマクロによって管理するキーを増やすのが簡単になりました。例えば上移動と下移動のキーを追加する場合は

```

1 (defkeystate key-state
2   (right :sdl-key-right)
3   (left :sdl-key-left)
4   (up :sdl-key-up)
5   (down :sdl-key-down))

```

のようになります。

4.4 まとめ

今回はキー入力を拾う方法を知るところからキー入力の管理、さらにコードのメンテナンス性を上げるためにマクロによる抽象化を被せていきました。

次からはこれまでに学んだ技術を組み合わせていよいよシューティングゲームを作っていきます。

⁸update-key-state の引数名が変わってたりするけど、これはユーザの書いたコードの変数名とバッティングするのを防ぐ為です。詳しくはマクロに触れている本を読んでください。『実践 Common Lisp』がおすすめ。

第5章 「絵を動かす」から「自機を動かす」へ

この章からいよいよシューティングゲームを作っていきます。まずは自機を表示するところから行きましょう。はじめに「とりあえず動く」コードを書いてみます。そのままではいろいろ問題があるので抱えている問題を潰しつつ、コードを整理していきます。

5.1 絵を動かす

シューティングゲームということでまずは自機を動かせるようになりたいですね。ということでまずは自機の絵を方向キーで動かすというミッションから始めていきます。

キー入力の管理については前の章で書いたコードを使えばよいでしょう。

List 5.1: key-state.lisp

```
1 ;; 総称関数
2 (defgeneric update-key-state (key key-press key-state)
3   (:documentation "キーの状態を更新する"))
4
5 ;; キー入力状態クラス用のマクロ
6 (defmacro defkeystate (name &rest key-maps)
7   '(progn
8     (defclass ,name ()
9       ,(loop for k in key-maps collect '(, (car k) :initform nil)))
10    ,(let ((key (gensym)) (key-press (gensym)) (key-state (gensym)))
11        '(defmethod update-key-state (,key ,key-press (,key-state ,name))
12          (with-slots ,(mapcar #'car key-maps) ,key-state
13            (cond ,(loop for k in key-maps
14                        collect '((sdl:key= ,key ,(cadr k))
15                                (setf ,(car k) ,key-press))))))))))
16
17 ;; キー入力の状態クラス
18 (defkeystate key-state
19   (up :sdl-key-up)
20   (down :sdl-key-down)
21   (right :sdl-key-right)
22   (left :sdl-key-left))
```

今回は上下左右の方向キーを使います。ということで List5.1 の 18~22 行目で各キーの割り当てを書いています。

キー入りに合わせて絵を動かすにはどうすればいいでしょうか？手順をまとめてみましょう。

1. 画像を描画する座標を変数として用意する
2. 方向キーの入力を受け取る
3. 方向キーの入力に合わせて描画する座標を更新する
4. 絵を描画する

こんな感じの手順でいけそうですね。ということでプログラム (List5.2) を書いてみました。

List 5.2: main.lisp

```

1  ;;; グローバル変数/定数定義
2  ;; 2。斜め移動時の補正に使う
3  (defconstant +sqrt-2+ (sqrt 2))
4
5  (load "key-state.lisp" :external-format :utf-8)
6
7  (defun load-png-image (source-file)
8    (sdl:convert-to-display-format :surface (sdl:load-image source-file)
9      :enable-alpha t
10     :pixel-alpha t))
11
12 (defun main ()
13   (sdl:with-init ()
14     (sdl:window 640 480 :title-caption "てすと")
15     (setf (sdl:frame-rate) 60)
16     (let ((current-key-state (make-instance 'key-state))
17           (x 0) ; 自機 X座標
18           (y 0) ; 自機 Y座標
19           (speed 5) ; 速さ。1フレームあたりの移動量
20           ;; 自機の画像をロード
21           (player-img (load-png-image "player.png")))
22
23       (sdl:update-display)
24
25       (sdl:with-events ()
26         (:quit-event () t)
27         (:key-down-event (:key key)
28           (if (sdl:key= key :sdl-key-escape)
29             (sdl:push-quit-event)
30             (update-key-state key t current-key-state)))
31         (:key-up-event (:key key)
32           (update-key-state key nil current-key-state))
33         (:idle ()
34           (sdl:clear-display sdl:*black*)
35           ;; 移動キーの操作を移動量に反映
36           (let ((dx 0) (dy 0)) ; X方向とY方向の移動量
37             (with-slots (up down right left) current-key-state
38               (cond (right (incf dx speed))
39                     (left (decf dx speed))
40                     (cond (up (decf dy speed))
41                           (down (incf dy speed))))
42             ;; 斜め移動の場合の処理。移動量を補正します
43             (when (and (/= dx 0) (/= dy 0))
44               (setf dx (/ dx +sqrt-2+) dy (/ dy +sqrt-2+)))
45             ;; 移動量を自機の座標に反映
46             (incf x dx)
47             (incf y dy))
48           ;; 自機を描画
49           (sdl:draw-surface-at* player-img (round x) (round y))
50           (sdl:update-display))))))

```

実行すると図 5.1 のようになります。

それではプログラムを見て行きましょう。

16~21 行目ではこれから使う変数を定義しています。x と y が手順 1. で言った描画する座標の変数です。この変数が表す座標に絵を描画することにします。speed は方向キーを押したときに絵が移動するスピードです。これは 1 フレーム何ピクセル動くかで表します。

27~32 行目はキー入力を拾う処理ですね。前の章でやったのと同じことをやってます。

33~50 行目がゲームループです。画面を消去して (34 行目)、描画する座標を更新して (36~47 行目)、

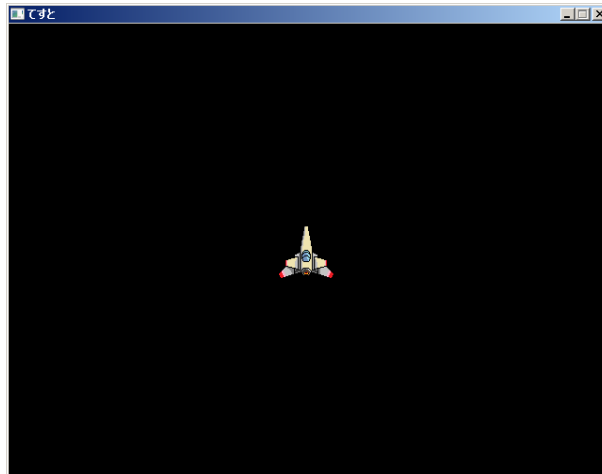


図 5.1: main.lisp の実行結果。キー操作で自機の絵が動く。

描画する (49 行目) といった流れになっています。

では描画する座標を更新するところについて詳しく見ていきましょう。まず次のフレームで X 方向と Y 方向にどれくらい移動するかをそれぞれ dx と dy に持たせることにします。どちらも 0 なら X 方向にも Y 方向にも移動しないことになります。38~39 行目では X 方向の移動量を求めています。右方向キーが押されていたら dx を増やす (右方向に移動する)、左方向キーが押されていたら dx を減らす (左方向に移動する) といった処理をしています¹。40~41 行目では X 方向の場合と同様にして Y 方向の移動量を求めています。

43~44 行目では斜め移動の時の補正をしています。はい、補正が必要なんですよ奥さん。

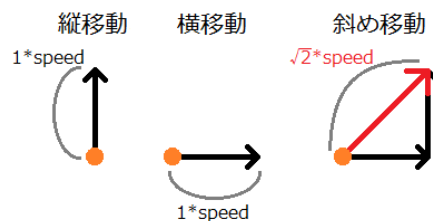


図 5.2: 斜め移動では補正が必要

そもそもどうやって斜め移動をやっているのかというと縦方向の移動処理と横方向の移動処理を独立にやることで実現しています。例えば上キーと右キーを同時押しした場合は、上キーの処理 (上方向に $speed$ ピクセル移動する) と右キーの処理 (右方向に $speed$ ピクセル移動する) を行います。で、そうやって処理すると結局今の位置から上方向に $speed$ 、右方向に $speed$ ずれたところが移動先の座標になります。さて、このとき元の位置からどれだけ動いたかということ $\sqrt{2} * speed$ なのです (図 5.2)。1 フレームあたりの移動量を $speed$ と決めてたのにその $\sqrt{2}$ 倍の距離を移動することになってしまいます。ということで斜め移動の場合は移動量が $speed$ になるようにするために dx と dy をそれぞれ $\sqrt{2}$ で割っています (44 行目)。

そんなこんなで移動量 dx と dy が求まったところで画像の座標を更新しているのが 46~47 行目です。今の座標に移動量を足すだけです。

49 行目では画像を描画しています。 x と y に $round$ 関数を噛ませて整数値に変換しています。これは

¹左右のキー両方押しした場合は右キーの処理が優先します。

sdl:draw-surface-at-*関数に渡す座標は整数じゃないといけないからです²。

ここまでひととおりプログラム (List5.2) について見てきました。このプログラムを実行すればちゃんと方向キーの入力に合わせて絵が動きます。

5.2 さっきのプログラムのまずさと対策

とりあえず絵を動かすプログラムが書けました。でもこのプログラムだけでこの章を終わらせる訳にはいきません。色々問題点があるからです。

まず挙げるべき点は座標更新のためのロジックがゲームループにベタ書きされていることです。この調子でゲーム製作を進めていくとどんどんゲームループに処理がベタ書きされてコードが読みづらくなります。処理に使う変数もどんどん増えていくので変数名のバツィングとかにも気をつけないといけなくなります。そんなわけで処理を切り分けることを考えましょう。

このプログラムを「絵を動かすプログラム」ではなく「自機を動かすプログラム」としてとらえてみましょう。そうすると座標を更新する処理も絵を描画する処理も「自機に対する処理」だということが分かります。また、描画する座標である x や y 、描画する画像 `player-image` が「自機のパラメータ」であることも見えてきます。そこで自機のクラスを用意することで自機に関する処理や変数を `main` 関数から切り離すことにします。

もうひとつ問題点があります。このプログラムでは画面外に移動できてしまいます。画面外に移動できないように座標更新のロジックを直してやる必要がありますね。ここで座標更新ロジックを修正する前に画面に関するパラメータをどうするか考えてみます。画面外に出ているかどうかを知るには画面のサイズを知っておく必要があります。こういった画面に関する情報もまとめておいた方がコードの見通しが良くなると思います。ということで画面に関する情報をクラスにまとめておくことにします。

5.3 自機を作る

自機に関するデータや処理を切り離そう、ということでここでは自機のクラスを作っていきます。

まず自機を表すのにどんな情報が必要か考えてみましょう。とりあえずさっきのプログラムの時点で描画する画像やら座標やら移動速度やらを扱っていたのでこれらの情報は必要でしょう。他にも必要な情報を考えてとりあえず自機クラスを作ってみました (List5.3)。

List 5.3: `player.lisp`:自機クラスの定義

```
7 ;; 自機クラス
8 (defclass player ()
9   ((x
10    :initarg :x
11    :initform 0
12    :documentation "中心X座標 ")
13    (y
14    :initarg :y
15    :initform 0
16    :documentation "中心Y座標 ")
17    (width
18    :initarg :width
19    :initform 64
20    :documentation "幅")
21    (height
22    :initarg :height
23    :initform 64
```

²このコードを書いているときにこの仕様を忘れていて「なぜエラーを吐くんだ!」と慌てました。

```

24     :documentation "高さ")
25   (speed
26     :initarg :speed
27     :initform 5
28     :documentation "速さ")
29   (image
30     :initarg :image
31     ;; 初期化するとき必ず画像ファイルを指定してください > <
32     :initform (error "自機画像ファイルを指定しろ")
33     :documentation "画像"))

```

自機座標として x と y 、移動速度として `speed`、自機画像として `image` のスロットを用意しました。この他に画像の幅 `width` や高さ `height` も用意しました。さっきのプログラムでは自機の座標として左上の点を取っていましたが、今回の自機クラスでは中心の点に変えています(図 5.3)。この方があとで当たり判定を考えると都合がいいからです。とりあえず今はそういうもんだと納得して下さい。

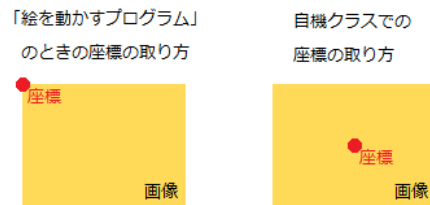


図 5.3: 座標の取り方の違い

各スロットの初期値は適当に決めてます。まあ大体 `make-instance` するとき明示的に指定すると思いますが一応、ということで。画像については明示的に指定しないとエラーになるようにしています。

定義した自機クラスに対してどんな処理が必要でしょうか。さきほどの絵を動かすプログラムでは自機に対して行なった処理は2つあります。「自機の状態を更新する処理」と「自機を描画する処理」です。今回もとりあえずこの2つの処理を用意しておけばいいでしょう。ということでそれぞれの処理に対応する総称関数を定義しておきます(List 5.4)。`update-player` が自機の状態を更新する処理、`draw-player` が自機を描画する処理ですね。

List 5.4: `player.lisp`: 総称関数の定義

```

1 ;; 総称関数
2 (defgeneric update-player (player key-state game-field)
3   (:documentation "自機の状態を更新する"))
4 (defgeneric draw-player (player game-field)
5   (:documentation "自機を描画する"))

```

それでは `update-player` メソッドの実装を定義していきましょう(List 5.5)。引数としては自機クラス `player` とキー入力の状態 `key-state`、ゲーム画面クラス `game-field` を取ることにします。ゲーム画面クラスの定義についてはあとで見えていきますが、今はとりあえずこのクラスから画面幅と画面高さが取得できることだけ分っていれば OK です。

List 5.5: `player.lisp`: `update-player` メソッドの実装

```

65 ;;; 自機の状態更新
66 (defmethod update-player ((player player)
67   (key-state key-state)
68   (game-field game-field))
69   (move-by-input player key-state)
70   (fix-position player game-field))

```

自機の更新処理は大きく分けて「キー入力に基づいて移動する処理」と「画面外にはみ出ないように位置を補正する処理」の2つです。それぞれの処理をメソッドに分けました³。

分けた2つの処理に対応するメソッドを見ていきましょう。

List 5.6: player.lisp:move-by-input メソッドの実装

```
35 ;; キー入力による移動処理
36 (defmethod move-by-input ((player player) (key-state key-state))
37   (let ((dx 0) (dy 0))
38     (with-slots (up down right left) key-state
39       (with-slots (speed) player
40         (cond (right (incf dx speed))
41               (left  (decf dx speed)))
42         (cond (up    (decf dy speed))
43               (down (incf dy speed))))))
44   ;; 斜め移動の場合の補正
45   (when (and (/= dx 0) (/= dy 0))
46     (setf dx (/ dx +sqrt-2+) dy (/ dy +sqrt-2+)))
47   (with-slots (x y) player
48     (incf x dx)
49     (incf y dy)))
```

move-by-input メソッド (List5.6) はキー入力に基づいて移動する処理を行うメソッドです⁴。コードの内容は絵を動かすプログラムのときにゲームループの中でやっていた処理 (List5.2 の 36 ~ 47 行目) とほとんど同じです。

List 5.7: player.lisp:fix-position メソッドの実装

```
51 ;; 画面外にはみ出たときの位置補正処理
52 (defmethod fix-position ((player player) (game-field game-field))
53   (with-slots (x y width height) player
54     (let ((field-width (game-field-width game-field))
55           (field-height (game-field-height game-field))
56           (width/2 (/ width 2))
57           (height/2 (/ height 2)))
58       (when (< (- x width/2) 0) (setf x width/2))
59       (when (< (- y height/2) 0) (setf y height/2))
60       (when (> (+ x width/2) field-width)
61         (setf x (- field-width width/2)))
62       (when (> (+ y height/2) field-height)
63         (setf y (- field-height height/2))))))
```

fix-position メソッド (List5.7) は画面からはみ出した場合に位置を補正する処理を行うメソッドです。画面からはみ出しについて調べるのでゲーム画面クラスのインスタンスを引数にとります。では処理の内容を見ていきましょう。54~55 行目で画面の幅と高さをゲーム画面クラスから取得しています。自機が画面からはみ出すパターンは4種類あります⁵(図 5.4)。58~63 行目ではそれぞれのパターンに応じて座標の補正を行なっています。

自機の更新に関する処理は以上です。

³最初はメソッドに分けずに書いていたけど後の章書いているうちにだんだんコードの見通しが悪くなってきたので分けました。

⁴move-by-input メソッドと fix-position メソッドに対しては総称関数を定義していないが、これは別に問題ない。総称関数を定義しない場合処理系によって自動的に総称関数の定義が生成される。まあ手抜きといえば手抜きなんだけど。

⁵同時に2つのパターンに該当するようにはみ出し方もある。X 方向にも Y 方向にもみ出している場合がこれに該当する。その場合は2つのパターンそれぞれの補正が行われる。

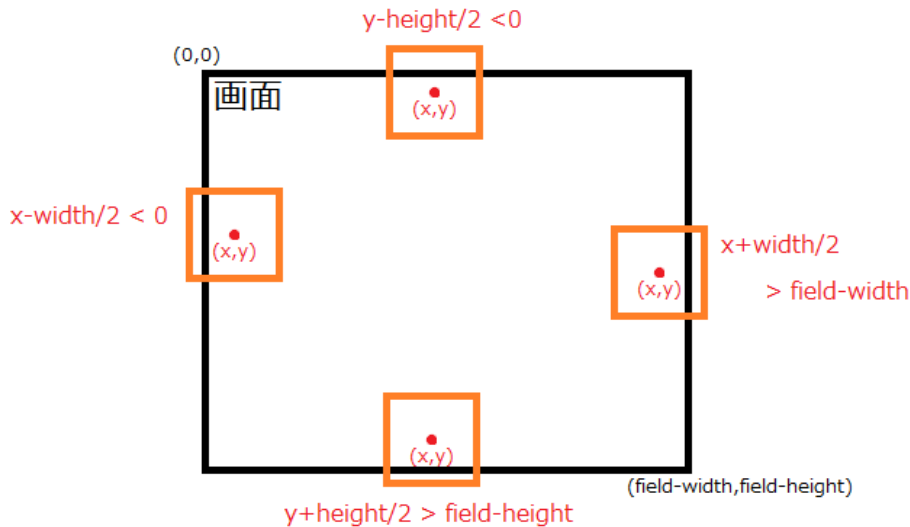


図 5.4: 自機が画面からはみ出すパターンは 4 つある

次に draw-player メソッドの実装を定義していきましょう (List5.8)。まあ画像を一枚描画するだけなんです。

List 5.8: player.lisp:draw-player メソッドの実装

```

72 (defmethod draw-player ((player player) (game-field game-field))
73   (with-slots (x y width height image) player
74     (sdl:draw-surface-at-* image
75       (round (+ (game-field-x game-field)
76                (- x (/ width 2))))
77       (round (+ (game-field-y game-field)
78                (- y (/ height 2)))))))

```

引数として自機クラスの他にゲーム画面クラスも受け取ることにしています。これは描画するときの原点をゲーム画面クラスで指定できた方がいいかな、という考えでこうしています⁶。そんなわけで 67 行目と 69 行目ではゲーム画面の原点座標を取得して自機の描画座標に足してます。また、自機の座標を画像の左上ではなく中心とするようにしたので、描画するときには左上の座標を計算しています (76 行目、78 行目)。

自機クラスまわりのコードについては以上です。

5.4 ゲーム画面クラスを作る

自機クラスの方は出来上がったのでゲーム画面クラスの方を定義していきます (List5.9)。今回は画面の幅と高さ、原点の位置 (X 座標と Y 座標) の 4 つのスロットを持たせます。これらのスロットはメソッド内以外でも扱うと思うのでそれぞれ:reader で読み出し用のメソッドを生成するようにしています。

List 5.9: game-field.lisp

```

1 ;; ゲーム画面クラス
2 (defclass game-field ()
3   ((width
4     :reader game-field-width
5     :initarg :width
6     :initform 0

```

⁶ゲーム画面とウィンドウの描画領域の間に余白を持たせたい、といったケースに対応しやすくなります。


```

7   :documentation "画面幅")
8   (height
9     :reader game-field-height
10    :initarg :height
11    :initform 0
12    :documentation "画面高さ")
13   (x
14     :reader game-field-x
15     :initarg :x
16     :initform 0
17     :documentation "画面左上X座標")
18   (y
19     :reader game-field-y
20     :initarg :y
21     :initform 0
22     :documentation "画面左上Y座標"))))

```

今のところゲーム画面クラスに対して何か処理をする必要はないので総称関数やメソッドは定義しません。

5.5 自機を動かす

ここまでで自機クラスとゲーム画面クラスができました。ということでこれらのクラスを使って自機をキー入力で動かすというミッションをこなしていきましょう。

List 5.10: main2.lisp

```

1  ;;; グローバル変数/定数定義
2  ;; 2. 斜め移動時の補正に使う
3  (defconstant +sqrt-2+ (sqrt 2))
4
5  (load "key-state.lisp" :external-format :utf-8)
6  (load "game-field.lisp" :external-format :utf-8)
7  (load "player.lisp" :external-format :utf-8)
8
9  (defun load-png-image (source-file)
10   (sdl:convert-to-display-format :surface (sdl:load-image source-file)
11                                   :enable-alpha t
12                                   :pixel-alpha t))
13
14  (defun main ()
15   (sdl:with-init ()
16     (sdl:window 640 480 :title-caption "てすと")
17     (setf (sdl:frame-rate) 60)
18     (let* ((current-key-state (make-instance 'key-state))
19            ;; ゲーム画面オブジェクトを生成
20            (game-field (make-instance 'game-field
21                                       :width 640
22                                       :height 480
23                                       :x 0
24                                       :y 0))
25            ;; 自機オブジェクトを生成
26            (player (make-instance 'player
27                                   :x (/ (game-field-width game-field) 2)
28                                   :y (/ (game-field-height game-field) 2)
29                                   :width 64
30                                   :height 64
31                                   :speed 5
32                                   :image (load-png-image "player.png")))))

```

```

33     (sdl:update-display)
34
35     (sdl:with-events ()
36       (:quit-event () t)
37       (:key-down-event (:key key)
38         (if (sdl:key= key :sdl-key-escape)
39             (sdl:push-quit-event)
40             (update-key-state key t current-key-state)))
41       (:key-up-event (:key key)
42         (update-key-state key nil current-key-state))
43       (:idle ()
44         (sdl:clear-display sdl:*black*)
45         ;; 移動キーの操作を移動量に反映
46         (update-player player current-key-state game-field)
47         ;; 自機を描画
48         (draw-player player game-field)
49         (sdl:update-display))))))

```

ではプログラムを見ていきましょう。20～24行目ではゲーム画面クラスのインスタンスを、26～32行目では自機クラスのインスタンスをそれぞれ生成しています。43～49行目がゲームループの処理です。自機関連の処理をゲームループから自機クラスの方に切り離したので、ゲームループ内の記述が簡潔になりました。

これから章を進めていくにつれてコードを構成する要素が増えていきますが、簡潔なコードになるように整理することを心がけましょう。

ひと通りプログラムが書けたので実際に動かしてみましょう。ちゃんと自機が画面外にはみ出ないようになってますね？ということでこの章はめでたしめでたし。

5.6 まとめ

絵を動かすプログラムを整理して色々問題を潰していったら自機を動かすプログラムができました⁷。やったね！

とりあえず自機を動かせるようになったんで次は敵を出しましょう。

⁷まあ自機を動かすプログラムを目指して整理していた訳ですけどね。

第6章 敵を出す

ゲームと言ったら敵が出てきて、それを倒していったなんぼのものだと思います¹。ということでこの章では敵を出すコードを書いていきます。

6.1 敵を作る

敵を出すって言うてはみたけど実際何をすればいいでしょうか。やることはこの2つに絞られるかと思います。

- 敵の状態 (位置とか) を更新する
- 敵を描画する

なんだかやることが自機を作るときと似たような感じがしますね。ええ、実際同じです。なので自機を作るのと同じような考え方で敵も作れそうですね。

ということで敵のクラスを作っちゃいましょう (List6.1)。

List 6.1: enemy.lisp:敵クラスの定義

```
9 ;; 敵クラス
10 (defclass enemy ()
11   ((x
12    :initarg :x
13    :initform 0
14    :documentation "中心X座標 ")
15    (y
16     :initarg :y
17     :initform 0
18     :documentation "中心Y座標 ")
19    (vx
20     :initarg :vx
21     :initform 0
22     :documentation "X方向速度 ")
23    (vy
24     :initarg :vy
25     :initform 0
26     :documentation "Y方向速度 ")
27    (width
28     :initarg :width
29     :initform 128
30     :documentation "幅")
31    (height
32     :initarg :height
33     :initform 128
34     :documentation "高さ")
35    (image
36     :initarg :image
37     :initform (error "敵画像ファイルを指定しろ")
38     :documentation "画像")))
```

¹パズルゲームとかはどうかのって話ですが、その場合ゲーム作成者が敵ですね。

中心座標なり画像なり幅や高さなりをスロットとして持っているあたりは自機クラスと同じですね。少し違うのは speed ではなく vx と vy を持っているあたりです。これは敵の場合自機とは違ってキー入力などユーザーの操作に頼らずに動くので、毎回敵の座標を更新する際にどの方向に動かすかを知っている必要があるからです²。

それでは総称関数の定義を見ていきます。

List 6.2: enemy.lisp:敵クラスの定義

```
1 ;; 総称関数
2 (defgeneric update-enemy (enemy)
3   (:documentation "敵の状態を更新する"))
4 (defgeneric draw-enemy (enemy game-field)
5   (:documentation "敵を描画する"))
```

敵クラスに対して行う処理は2つ、「敵の状態を更新する処理」と「敵を描画する処理」です。これは自機の場合と同じですね。1行目の update-enemy 関数ですが、自機と違ってキー入力を受け取る必要は無いので key-state を引数に取りません。また、状態更新の処理では画面からはみ出しについて考えないことにします。ということで game-field も引数に取りません。3行目の draw-enemy 関数については自機の描画関数 draw-player と同じ感じですね。

ではそれぞれのメソッド定義を見ていきましょう。

List 6.3: enemy.lisp:メソッド定義

```
40 (defmethod update-enemy ((enemy enemy))
41   (with-slots (x y vx vy) enemy
42     (incf x vx)
43     (incf y vy)))
44
45 (defmethod draw-enemy ((enemy enemy) (game-field game-field))
46   (with-slots (x y width height image) enemy
47     (sdl:draw-surface-at-* image
48       (round (+ (game-field-x game-field)
49                 (- x (/ width 2))))
50       (round (+ (game-field-y game-field)
51                 (- y (/ height 2)))))))
```

update-enemy の方は単に現在の座標に速度成分を足すだけです。キー入力やら画面のはみ出しやら補正やらは扱いません。draw-enemy の方も自機の描画とほとんど同じですね。

なんだかあっけない感じが、これでとりあえず敵クラスができました。

6.2 敵を動かす

敵ができたので動かしてみましょう (実行結果は図 6.1)。

List 6.4: main.lisp

```
1 ;;; グローバル変数/定数定義
2 ;; 2. 斜め移動時の補正に使う
3 (defconstant +sqrt-2+ (sqrt 2))
4
5 (load "key-state.lisp" :external-format :utf-8)
6 (load "game-field.lisp" :external-format :utf-8)
7 (load "player.lisp" :external-format :utf-8)
8 (load "enemy.lisp" :external-format :utf-8)
9
```

²自機が vx やら vy やらを持つ必要がないことが特殊だとも言えます。

```

10 (defun load-png-image (source-file)
11   (sdl:convert-to-display-format :surface (sdl:load-image source-file)
12     :enable-alpha t
13     :pixel-alpha t))
14
15 (defun main ()
16   (sdl:with-init ()
17     (sdl:window 640 480 :title-caption "てすと")
18     (setf (sdl:frame-rate) 60)
19     (let* ((current-key-state (make-instance 'key-state))
20           ;; ゲーム画面オブジェクトを生成
21           (game-field (make-instance 'game-field
22                                   :width 640
23                                   :height 480
24                                   :x 0
25                                   :y 0))
26           ;; 自機オブジェクトを生成
27           (player (make-instance 'player
28                                 :x (/ (game-field-width game-field) 2)
29                                 :y (/ (game-field-height game-field) 2)
30                                 :width 64
31                                 :height 64
32                                 :speed 5
33                                 :image (load-png-image "player.png")))
34           ;; 敵オブジェクトを生成
35           (enemy (make-instance 'enemy
36                                :x 320
37                                :y 0
38                                :width 64
39                                :height 64
40                                :vx 0
41                                :vy 3
42                                :image (load-png-image "enemy.png"))))
43     (sdl:update-display)
44
45     (sdl:with-events ()
46       (:quit-event () t)
47       (:key-down-event (:key key)
48         (if (sdl:key= key :sdl-key-escape)
49             (sdl:push-quit-event)
50             (update-key-state key t current-key-state)))
51       (:key-up-event (:key key)
52         (update-key-state key nil current-key-state))
53       (:idle ()
54         (sdl:clear-display sdl:*black*)
55         ;; 移動キーの操作を移動量に反映
56         (update-player player current-key-state game-field)
57         ;; 敵の状態を更新
58         (update-enemy enemy)
59         ;; 自機を描画
60         (draw-player player game-field)
61         ;; 敵を描画
62         (draw-enemy enemy game-field)
63         (sdl:update-display))))))

```

敵を出すコードは自機を出すコードと似ています。まず 35 ~ 42 行目で敵クラスのインスタンスを作って、それからゲームループの中で update-enemy 関数と draw-enemy 関数を呼んでいます (58 行目と 62 行目)。

うーん、やっぱりあっけないなー。なんでだろうね。

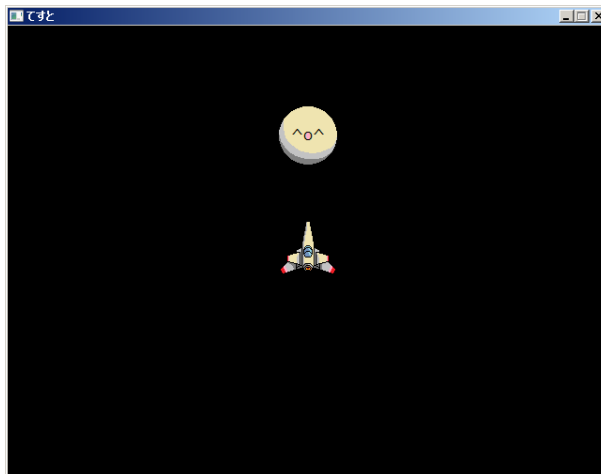


図 6.1: main.lisp の実行結果。

6.3 敵を動かす (いっぱい)

さっきからあっけないと感じる理由は自機を作るときとコードが似てて新鮮味がないというが1つですが、もう1つあります。それは敵が一匹しかいないことです。大抵敵っていっぱい出てきますよね。

ということで敵をいっぱい出しましょう。

ところで1匹の敵だけを扱うときは1つの変数に1つの敵インスタンスを放り込んでいましたが、敵が何匹もいるときはどうすればいいのでしょうか。Lispには複数のものを一括で扱うためのデータ構造³がいくつかあります。例えばリストやベクタなどがそうです。ここではリストで複数の敵を管理するというアイデアを基本にして、敵を管理するクラスを作っていきます⁴。

ということでまずは敵管理クラスを定義します。

List 6.5: enemy-manager.lisp:敵管理クラス定義

```

11 ;; 敵管理クラス
12 (defclass enemy-manager ()
13   ((enemies
14     :initform nil
15     :documentation "管理している敵たち")))
```

スロットは1個だけ、管理する敵をリストで持つことにします。最初は空のリストを持っていて後から管理する敵を追加していきます。

次に敵管理クラスに対して行う操作を定義していきます。

List 6.6: enemy-manager.lisp:敵管理クラス総称関数

```

1 ;; 総称関数
2 (defgeneric add-enemy (enemy-manager enemy)
3   (:documentation "管理する敵を追加する"))
4 (defgeneric update-enemies (enemy-manager)
5   (:documentation "敵の状態を更新する"))
6 (defgeneric draw-enemies (enemy-manager game-field)
7   (:documentation "敵を描画する"))
```

敵クラスに対してやることは「状態の更新」と「描画」でした。これらのことは敵管理クラスが敵クラスに対して行うことになります。加えて敵管理クラスでは管理する敵を追加していく操作が必要になります。ということで List6.6 のような定義になります。

³こういうのを一般的にコレクションとか言ったりします。

⁴当初はリストで直接敵を管理していたが、あとの章のコードを書いている内に面倒が増えてきたのでクラスを被せました。

ではメソッドの定義を書いていきましょう。

List 6.7: enemy-manager.lisp:敵管理クラスメソッド定義

```
17 (defmethod add-enemy ((enemy-manager enemy-manager) (enemy enemy))
18   (with-slots (enemies) enemy-manager
19     (push enemy enemies)))
20
21 (defmethod update-enemies ((enemy-manager enemy-manager))
22   (with-slots (enemies) enemy-manager
23     (dolist (enemy enemies)
24       (update-enemy enemy))))
25
26 (defmethod draw-enemies ((enemy-manager enemy-manager)
27                           (game-field game-field))
28   (with-slots (enemies) enemy-manager
29     (dolist (enemy enemies)
30       (draw-enemy enemy game-field))))
```

add-enemy は単に新たな敵を enemies スロットに追加していただくだけです。update-enemies と draw-enemies はそれぞれ enemies スロットの各要素 (敵クラスのインスタンスですね) に対して update-enemy メソッドや draw-enemy メソッドを呼んで、管理している各敵の状態を更新したり描画したりしています。

ここまでで複数の敵の管理ができるようになったのでプログラムを書いてみました (List6.8、実行結果:図 6.2)。

List 6.8: main2.lisp

```
1 ;;; グローバル変数/定数定義
2 ;; 2。斜め移動時の補正に使う
3 (defconstant +sqrt-2+ (sqrt 2))
4
5 (load "key-state.lisp" :external-format :utf-8)
6 (load "game-field.lisp" :external-format :utf-8)
7 (load "player.lisp" :external-format :utf-8)
8 (load "enemy.lisp" :external-format :utf-8)
9 (load "enemy-manager.lisp" :external-format :utf-8)
10
11 (defun load-png-image (source-file)
12   (sdl:convert-to-display-format :surface (sdl:load-image source-file)
13     :enable-alpha t
14     :pixel-alpha t))
15
16 (defun main ()
17   (sdl:with-init ()
18     (sdl:window 640 480 :title-caption "てすと")
19     (setf (sdl:frame-rate) 60)
20     (let* ((current-key-state (make-instance 'key-state))
21            ;; ゲーム画面オブジェクトを生成
22            (game-field (make-instance 'game-field
23                                     :width 640
24                                     :height 480
25                                     :x 0
26                                     :y 0))
27            ;; 自機オブジェクトを生成
28            (player (make-instance 'player
29                                  :x (/ (game-field-width game-field) 2)
30                                  :y (/ (game-field-height game-field) 2)
31                                  :width 64
32                                  :height 64
33                                  :speed 5
34                                  :image (load-png-image "player.png"))))
```

```

35     ;; 敵管理クラスを生成
36     (enemy-manager (make-instance 'enemy-manager)))
37 ;; 敵を生成
38 (dotimes (i 10)
39   (add-enemy enemy-manager
40     (make-instance
41       'enemy
42       :x (+ 32 (* i (/ (game-field-width game-field) 10)))
43       :y 0
44       :width 64
45       :height 64
46       :vx 1
47       :vy 1
48       :image (load-png-image "enemy.png"))))
49 (sdl:update-display)
50 (sdl:with-events ()
51   (:quit-event () t)
52   (:key-down-event (:key key)
53     (if (sdl:key= key :sdl-key-escape)
54         (sdl:push-quit-event)
55         (update-key-state key t current-key-state)))
56   (:key-up-event (:key key)
57     (update-key-state key nil current-key-state))
58   (:idle ()
59     (sdl:clear-display sdl:*black*)
60     ;; 移動キーの操作を移動量に反映
61     (update-player player current-key-state game-field)
62     ;; 敵の状態を更新
63     (update-enemies enemy-manager)
64     ;; 自機を描画
65     (draw-player player game-field)
66     ;; 敵を描画
67     (draw-enemies enemy-manager game-field)
68     (sdl:update-display))))))

```

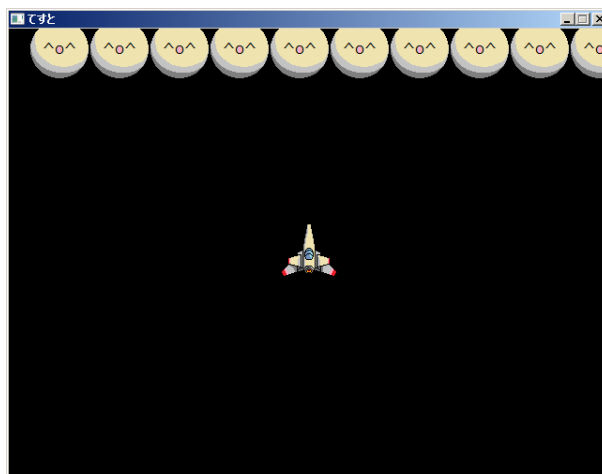


図 6.2: main2.lisp の実行結果。

それでは中身を見ていきましょう。

36 行目では敵管理クラスのインスタンスを持つ変数 `enemy-manager` を定義しています。38~48 行目では敵を生成して `enemy-manager` の管理下に追加しています。この操作を `dotimes` で 10 回繰り返しているので全部で 10 匹の敵が追加されています。63 行目では敵の状態の更新、65 行目では敵の描画を

行なっています。複数の敵に対する操作を enemy-manager が一手に引き受けているのでゲームループ内の記述が簡潔になっています。

これで敵をいっぱい出せますね。

6.4 成仏させる

ところで画面外に出ていった敵をいつまでも生かしておく必要はないですよね。今はまだ自機と10匹やそこらの敵しか出してませんが、これからゲームに色々な要素が増えていくと管理するものの数が増えていきます。そうすればメモリーをどんどん食い尽くすし、それぞれの要素に割く処理も増えるし…。

ということで画面外に出ていった敵を成仏させることを考えます。

まずは敵が画面外にいるかどうか判定する関数を用意します。まずは総称関数を定義します。

List 6.9: enemy.lisp:総称関数

```
6 (defgeneric out-of-field-p (enemy game-field)
7   (:documentation "敵が画面外にいるかどうか判定"))
```

引数には敵クラスとゲーム画面クラスのインスタンスを取ることになります。次はメソッド定義です。

List 6.10: enemy.lisp:メソッド定義

```
53 (defmethod out-of-field-p ((enemy enemy) (game-field game-field))
54   (with-slots (x y width height) enemy
55     (let ((width/2 (/ width 2))
56           (height/2 (/ height 2)))
57       (or (< (+ x width/2) 0)
58           (< (+ y height/2) 0)
59           (> (- x width/2) (game-field-width game-field))
60           (> (- y height/2) (game-field-height game-field)))))
```

敵が画面外にいる場合というのは図 6.3 にある4つのパターンのいずれか(1つないし2つ)です。4つのパターンの中に当てはまるものがあれば57~60行目の式が真を返します。それ以外は偽となります。

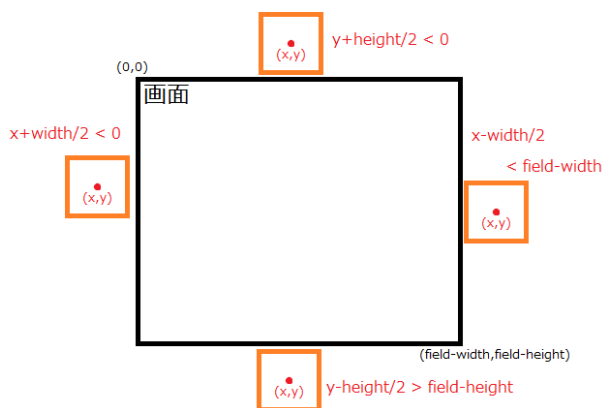


図 6.3: 敵が画面外に出るパターンは4つある

次に敵を管理するクラスの方を見ていきます。敵を管理するクラスなので敵を成仏させるのもこのクラスのお仕事です。ということで要らなくなった敵を成仏させる操作を新たに用意します。

List 6.11: enemy-manager.lisp:敵管理クラス総称関数

```
8 (defgeneric gc-enemies (enemy-manager game-field)
9   (:documentation "不要な敵を削除する"))
```

ではメソッドの中身を定義していきます。

List 6.12: enemy-manager.lisp:敵管理クラスメソッド定義

```
32 (defmethod gc-enemies ((enemy-manager enemy-manager) (game-field game-field))
33   (with-slots (enemies) enemy-manager
34     (setf enemies
35       (delete-if #'(lambda (enemy)
36                   (out-of-field-p enemy game-field))
37                 enemies))))
```

敵を成仏させるにはどうすればいいかという単に管理している敵のリストから削除すれば OK です⁵。ということで enemies スロットの各要素に対して out-of-field-p で画面外に出ているかどうか判定して、画面外に出た敵をリストから削除しています。

これでひと通り準備ができたので画面外に出た敵を成仏させるバージョンのプログラムを書いてみましょう。

List 6.13: main3.lisp:ゲームループ

```
59 (:idle ()
60   (sdl:clear-display sdl:*black*)
61   ;; 移動キーの操作を移動量に反映
62   (update-player player current-key-state game-field)
63   ;; 敵の状態を更新
64   (update-enemies enemy-manager)
65   ;; 画面外に出た敵を成仏
66   (gc-enemies enemy-manager game-field)
67   ;; 自機を描画
68   (draw-player player game-field)
69   ;; 敵を描画
70   (draw-enemies enemy-manager game-field)
71   ;; デバッグ用。今生きている敵の数を表示
72   (with-slots (enemies) enemy-manager
73     (sdl:draw-string-solid* (format nil "~a" (length enemies))
74                             500 400))
75   (sdl:update-display))))
```

ほとんど List6.8 と同じようなコードなのでゲームループの部分だけ見ていきます。

画面外に出た敵を削除する処理は 66 行目で行なっています。gc-enemies メソッドを呼び出すだけです。中でやっている処理は先に説明したとおり。72~74 行目では現在生きている敵の数を enemy-manager の enemies スロットの要素数から求めて表示しています。敵が画面外に出るとその分だけ数が減っていきます。

6.5 まとめ

敵を出せるようになりました。いっぱい敵を扱うために管理クラスを作りました。用済みの敵を処分しました。

といった感じで敵の管理もできるようになりました。次は当たり判定やります。

⁵敵のリストから削除すると、削除された敵はプログラムのどこからも参照されなくなるのでそのうちガベージコレクションによって成仏されるという理屈です。

第7章 当たり判定

自機ができました。敵もできました。でもまだ攻撃することもダメージを受けることもできません。悲しいですね。ぶつかったら何か反応が欲しいですね。ということで今回は当たり判定の話です。

7.1 当たり判定の理屈

敵と自機がぶつかったら何か起きてほしい。じゃあまずはぶつかったかどうかを検知しないといけませんね。これこそが当たり判定です。

当たり判定の難しさ(と効率)はどれくらい正確で精度の高い判定をするかに比例します。雑な精度でいいなら当たり判定も簡単です。ここでいう精度とは欲しい当たり判定の形にどれだけ似ているかということです。じゃあゲームでの当たり判定ってどの程度の精度が必要なのでしょうか。えっとですね、雑でもいいです。割とゆるい精度の当たり判定でもゲームの面白さってあんまり変わらんもんです。

そんな訳でここでは当たり判定の中でも簡単なやつを2つ紹介します。

7.1.1 円形の当たり判定

当たり判定の話は基本的に言葉で書くより図で示した方が早いのでまずは図を出しますね。

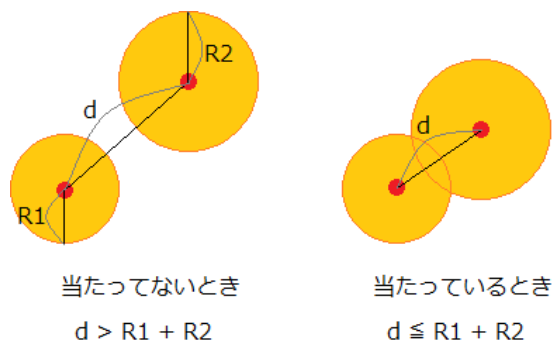


図 7.1: 円形対円形の当たり判定

円形の当たり判定は半径でサイズが決まります。2つの当たり判定の半径を R_1, R_2 、2つの当たり判定の中心の距離を d とすると

$$d \leq R_1 + R_2$$

のとき2つの当たり判定がぶつかっていると判定します(図 7.1)。

R_1 やら R_2 やらは始めから決まってるパラメータですが、 d はどうやって求めましょうか。2つの当たり判定の中心座標をそれぞれ $(x_1, y_1), (x_2, y_2)$ とすると3平方の定理から

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

ですね。

ということで当たり判定の式は

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \leq R_1 + R_2$$

と書けるのですが、実はこの大小関係は両辺を2乗しても変わりません。なので

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 \leq (R_1 + R_2)^2$$

で判定できます。円形の当たり判定を行うときはこの判定式を使います。

7.1.2 矩形の当たり判定

ここでは回転しない矩形同士の当たり判定を見ていきます¹。

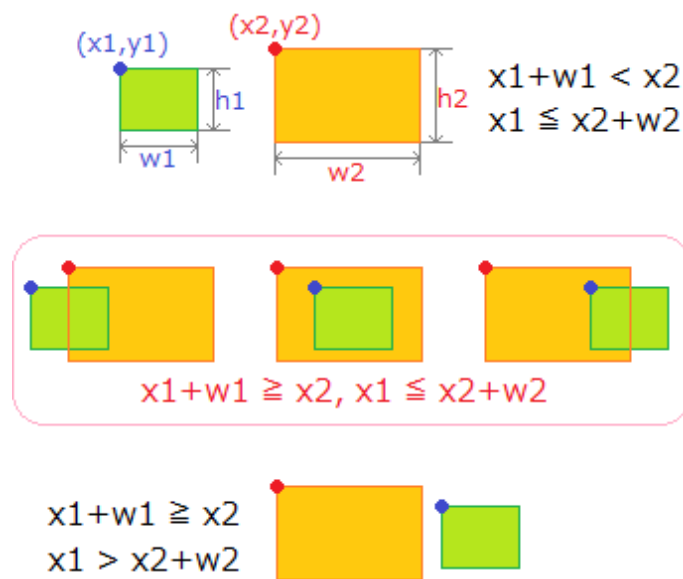


図 7.2: 矩形対矩形の当たり判定 (X 方向)

図 7.2 は 2 つの矩形の当たり判定について X 方向に配置を動かしていったときの関係です。ピンクの枠で囲まれているのがぶつかっている状態です。ぶつかっているときだけ $x_1 + w_1 \geq x_2, x_1 \leq x_2 + w_2$ が成り立つことが分かります。Y 方向についても同様にしてぶつかっているときだけ成り立つ条件が分かります。

2 つの矩形当たり判定があって左上頂点座標がそれぞれ $(x_1, y_1), (x_2, y_2)$ で幅と高さがそれぞれ $(w_1, h_1), (w_2, h_2)$ のとき、2 つの矩形当たり判定がぶつかっているのは

$$\begin{aligned} x_1 + w_1 &\geq x_2 \\ x_1 &\leq x_2 + w_2 \\ y_1 + h_1 &\geq y_2 \\ y_1 &\leq y_2 + h_2 \end{aligned}$$

という 4 つの条件が満たされる場合だけです。

¹回転を許すともっと複雑な判定になります。詳細はググれ。

7.1.3 どっちがいいの？

2つの当たり判定を紹介しましたが、実際ゲームで使うにはどっちがいいか悩みます。判定の効率に関して言えばどちらも大差ないです。となると後は実際に欲しい形に近いかが選択の鍵になります。

試しに自機の当たり判定について考えてみましょう。基本的な話ですが、当たり判定が見た目より大きくなるとまずいです。ということで見た目のサイズに収まるように当たり判定を設定します。今回は手を抜いて自機の座標と当たり判定の中心座標を一致させることにします²。さっき紹介した円形と矩形の当たり判定を自機にそれぞれ設定したのが図 7.3 です。

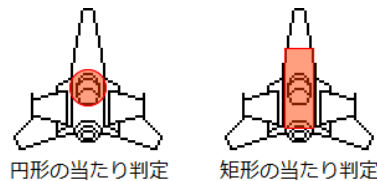


図 7.3: 自機に当たり判定を設定する例

どちらも全然自機の形と合っていないですね。翼の部分とか全然判定ないじゃないですか。はい、はじめに精度が悪い云々言ってたのはこういうことなんですよ。でも東方の当たり判定もこんなもんだったりするんでこんな適当な当たり判定でもゲームとしては十分なんです³。

自機の場合だけ見てもどっちの当たり判定がいいか分からないので他のオブジェクト (敵とか) の場合も見ながら考えましょう。今回の場合敵がおもいきり丸いので円形の当たり判定を採用することにします。

7.2 当たり判定を作る

今回は円形の当たり判定を使います。ということで当たり判定のクラスを定義していきます。

List 7.1: hit-test.lisp:クラス定義

```
4 ;; 当たり判定クラス。すべての当たり判定はこのクラスを継承して欲しい
5 (defclass hit-test-region ()
6   ())
7
8 ;; 円形当たり判定クラス
9 (defclass circle-region (hit-test-region)
10  ((x
11   :initarg :x
12   :initform 0
13   :accessor circle-x
14   :documentation "中心X座標 ")
15   (y
16   :initarg :y
17   :initform 0
18   :accessor circle-y
19   :documentation "中心Y座標 ")
20   (r
21   :initarg :r
22   :initform 0
23   :accessor circle-radius
24   :documentation "半径")))
```

²当たり判定の中心座標を自機の座標から少しずらしたりできるような設計の方が望ましいです。hit-test-region クラス代わりに Composite パターンを適用するとうまくいくかも。

³ゲームってぶっちゃけ正確さより面白さが大事なんですね。

今回はhit-test-regionクラスとcircle-regionクラスの2つのクラスを定義していて、circle-regionクラスはhit-test-regionクラスを継承しています。circle-regionは円形の当たり判定を表すクラスでhit-test-regionは任意の当たり判定を表すクラスです。今のところ円形の当たり判定だけを考えていますが、後々いろんな当たり判定が使われる場合を想定してhit-test-regionクラスを設けています。こういったクラスを用意しておいて全ての当たり判定クラスをhit-test-regionクラスの子クラスにすることによって、当たり判定全般に共通する操作を定義する際にはhit-test-regionに対する操作だけを定義してやればよくなります。現時点ではhit-test-regionクラス自体に用はないですが、一応作っておきました⁴。

ではcircle-regionクラスについて見ていきましょう。円形の当たり判定に必要な情報は円の中心座標と半径です。ということでそれに対応するスロットを用意しています(座標がxとy、半径がr)。それぞれのスロットは色々なところからアクセスされるのでアクセッサを用意しています。

それではこの当たり判定クラスに対する処理を見ていきます。まずは総称関数の定義から。

List 7.2: hit-test.lisp:総称関数定義

```
1 (defgeneric hit-test (region-1 region-2)
2   (:documentation "当たり判定を行う"))
```

2つの当たり判定クラスのインスタンスを受け取って当たり判定処理をする関数hit-testを定義しています。では次にメソッドの実装へ。

List 7.3: hit-test.lisp:メソッド定義

```
26 ;; 円形対円形の当たり判定クラス
27 (defmethod hit-test ((region-1 circle-region) (region-2 circle-region))
28   (let ((dx (- (circle-x region-1) (circle-x region-2)))
29         (dy (- (circle-y region-1) (circle-y region-2)))
30         (rs (+ (circle-radius region-1) (circle-radius region-2))))
31     (<= (+ (* dx dx) (* dy dy))
32         (* rs rs))))
```

2つの円形の当たり判定クラスのインスタンスを受け取って当たり判定処理を行います。やることは前の節で示した円形同士の当たり判定の条件式

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 \leq (R_1 + R_2)^2$$

が満たされているかどうかを判定することです。判定結果の真偽値がこの関数の戻り値です。

7.3 当たり判定を埋め込む

当たり判定クラスを作成したのでさっそく自機や敵に設定していきましょう。

まずは敵に当たり判定を設定していきます。敵には2つの当たり判定を持たせます。1つは自機に対してぶつかったときの当たり判定(攻撃判定)、もう1つは自機が撃った弾に被弾したときの当たり判定(ダメージ判定)⁵です。この2つの当たり判定を持たせるために敵クラスに新たなスロットを追加していきます。

List 7.4: enemy.lisp:敵クラス定義

```
35 (attack-region-radius
36   :initarg :attack-radius
```

⁴あとでどういう面倒が起きるか予見してコードを書けるかどうかプログラマの大事なセンスだと思います。こういうのはコードを書いたり読んだりして身につけるしかないのかなあ。

⁵まだ弾を撃つとかやってないですね…。

```

37     :initform 10
38     :documentation "攻撃判定半径")
39 (damage-region-radius
40   :initarg :damage-radius
41   :initform 10
42   :documentation "ダメージ判定半径")
43 (attack-region
44   :accessor attack-region
45   :documentation "攻撃判定")
46 (damage-region
47   :accessor damage-region
48   :documentation "ダメージ判定")

```

敵クラスには4つのスロットを追加しました。この4つのスロットの内訳は円形の当たり判定をもつスロット2つとその当たり判定の半径をもつスロット2つです。2つというのは攻撃判定用とダメージ判定用ということですね。当たり判定のスロットにはアクセッサを指定しています。

当たり判定のスロット (attack-region と damage-region) には circle-region クラスのインスタンスを格納する訳ですが、格納は敵クラスのインスタンスを生成する際に行います。

List 7.5: enemy.lisp:敵クラス初期化時の処理

```

54 ;; 敵クラス初期化処理
55 (defmethod initialize-instance :after ((enemy enemy) &key)
56   (with-slots (attack-region attack-region-radius x y) enemy
57     (setf attack-region (make-instance 'circle-region
58                                   :x x
59                                   :y y
60                                   :r attack-region-radius)))
61   (with-slots (damage-region damage-region-radius x y) enemy
62     (setf damage-region (make-instance 'circle-region
63                                   :x x
64                                   :y y
65                                   :r damage-region-radius))))

```

initialize-instance は敵クラスのインスタンスを make-instance で生成する際に呼び出されるメソッドです。このメソッドを呼ぶと attack-region と damage-region にそれぞれ circle-region クラスのインスタンスを生成して格納します。当たり判定のパラメータは中心座標については敵の中心座標を指定して、半径はそれぞれ attack-region-radius と damage-region-radius を指定します。

List 7.6: enemy.lisp:敵クラス更新メソッド

```

67 (defmethod update-enemy ((enemy enemy))
68   (flet ((update-hit-region (x y region)
69         (setf (circle-x region) x)
70         (setf (circle-y region) y)))
71     (with-slots (x y vx vy) enemy
72       (incf x vx)
73       (incf y vy)
74       ;; 当たり判定の座標更新
75       (update-hit-region x y (attack-region enemy))
76       (update-hit-region x y (damage-region enemy))))))

```

update-enemy にも手を加えます。敵の座標を更新したときに当たり判定の座標も更新するようにしました。具体的には flet で当たり判定の座標を更新する関数をローカルに定義して、この関数を使って攻撃判定とダメージ判定の座標を更新しています。

今回はデバッグのために当たり判定も描画することにします。ということで描画処理にも手を入れます。

List 7.7: enemy.lisp:敵クラス描画メソッド

```

78 (defmethod draw-enemy ((enemy enemy) (game-field game-field))
79   (flet ((draw-hit-region (region color)
80         (sdl:draw-circle-* (round (circle-x region) )
81                             (round (circle-y region) )
82                             (round (circle-radius region))
83                             :color color)))
84     (with-slots (x y width height image) enemy
85       (sdl:draw-surface-at-* image
86                               (round (+ (game-field-x game-field)
87                                           (- x (/ width 2))))
88                               (round (+ (game-field-y game-field)
89                                           (- y (/ height 2))))))
90     ;; デバッグ用。当たり判定描画
91     (draw-hit-region (damage-region enemy) sdl:*green*)
92     (draw-hit-region (attack-region enemy) sdl:*red*)))

```

update-enemy メソッドの時と同様に flet で当たり判定を描画する関数を定義して攻撃判定とダメージ判定を描画する処理を追加しています。攻撃判定が赤、ダメージ判定が緑の円で表示されます。

敵クラスに対する当たり判定の設定は以上です。

次は自機クラスの方に当たり判定を設定していきます。自機の場合は攻撃判定は持たなくてダメージ判定だけを持ちます⁶。

やることは敵クラスでやったことと同じです。自機クラスに当たり判定関連のスロットを追加して (List7.8)、初期化時の処理を書いて (List7.9)、更新と描画に手を加える (List7.10) といった感じです。

List 7.8: player.lisp:自機クラス定義

```

29 (damage-region-radius
30   :initarg :damage-radius
31   :initform 10
32   :documentation "ダメージ判定半径")
33 (damage-region
34   :accessor damage-region
35   :documentation "ダメージ判定")

```

List 7.9: player.lisp:自機クラス初期化時の処理

```

42 ;; 自機クラス初期化処理
43 (defmethod initialize-instance :after ((player player) &key)
44   (with-slots (damage-region damage-region-radius x y) player
45     (setf damage-region (make-instance 'circle-region
46                                       :x x
47                                       :y y
48                                       :r damage-region-radius))))

```

List 7.10: player.lisp:自機クラス更新と描画

```

81 ;; 当たり判定の座標更新処理
82 (defmethod update-region ((player player))
83   (with-slots (x y damage-region) player
84     (setf (circle-x damage-region) x)
85     (setf (circle-y damage-region) y)))
86
87 ;;; 自機の状態更新
88 (defmethod update-player ((player player)
89                           (key-state key-state)
90                           (game-field game-field))

```

⁶自機と敵がぶつかったら自機が死んで敵は死なないですね。自機は体当たりで攻撃できないんですね。


```

91 (move-by-input player key-state)
92 (fix-position player game-field)
93 (update-region player))
94
95 (defmethod draw-player ((player player) (game-field game-field))
96   (flet ((draw-hit-region (region color)
97           (sdl:draw-circle-* (round (circle-x region))
98                               (round (circle-y region))
99                               (round (circle-radius region))
100                                :color color)))
101     (with-slots (x y width height image) player
102       (sdl:draw-surface-at-* image
103                               (round (+ (game-field-x game-field)
104                                         (- x (/ width 2))))
105                               (round (+ (game-field-y game-field)
106                                         (- y (/ height 2))))))
107       ;; デバッグ用。当たり判定描画
108       (draw-hit-region (damage-region player) sdl:*green*)))

```

自機の更新処理の場合は当たり判定の座標更新を新たにメソッドとして定義していますが、結局やっていることは敵の場合と同じです。

敵には当たり判定を埋め込みましたが、敵の管理クラスの方にはまだ手を入れてなかったですね。敵に対する操作を行う際には敵管理クラスを挟むことになります。それは自機と敵との当たり判定を行うときでも同じです。

ということで敵管理クラスの方に当たり判定に関する処理を追加していきます。

List 7.11: enemy-manager.lisp:敵管理クラス総称関数

```

10 (defgeneric hit-test-enemies (enemy-manager region &key)
11   (:documentation "敵との当たり判定を行う。"))

```

追加する処理は1つ、与えられた当たり判定と管理している全ての敵との間で当たり判定処理を行うことです。第二引数で当たり判定を受け取ります。第二引数のあとに&keyを指定しておくことで他に追加でキーワード引数を指定できるようにしておきます。

ではメソッドの実装を見ていきます。

List 7.12: enemy-manager.lisp:敵管理クラスメソッド定義

```

40 (defmethod hit-test-enemies ((enemy-manager enemy-manager)
41                             (region circle-region)
42                             &key
43                             (region-type 'attack)
44                             (on-hit-func
45                              #'(lambda (enemy)
46                                  (declare (ignore enemy))
47                                  nil)))
48   (let ((region-getter (cond ((eq region-type 'attack) #'attack-region)
49                              ((eq region-type 'damage) #'damage-region))))
50     (with-slots (enemies) enemy-manager
51       (some #'(lambda (enemy)
52                 (if (hit-test (funcall region-getter enemy) region)
53                     (progn
54                       (funcall on-hit-func enemy)
55                       t)
56                     nil))
57             enemies))))

```

大雑把な流れでいうと管理している敵1つ1つに対してregionと当たり判定して1つでも当たったら真を返す、そうでなければ偽を返します。「1つでも~だったら真を返す、そうでなければ偽を返す」

というのを実現するために `some` 関数を使っています。

この関数ではキーワード引数として新たに2つの引数を取るようになっています。

1つは `region-type` です。これは敵の当たり判定として攻撃判定とダメージ判定のどちらを使うかを指定します。例えば敵と自機との当たり判定を行うときは攻撃判定を使うので `:region-type 'attack` を、敵と自機弾との当たり判定を行うときはダメージ判定を使うので `:region-type 'damage` を指定します。

もう1つは `on-hit-func` です。これは敵と当たったときに呼ぶ処理を指定します。この処理が呼び出されるときには当たった敵が引数として渡されます。今回は使いませんが、あとで自機弾との当たり判定をするときに使うことになると思うので用意しました。指定しなければ何もしない関数が呼び出されます⁷。

これで当たり判定処理の準備は万全です。

7.4 当たり判定する

ひと通り当たり判定処理の準備が整ったので実際に当たり判定処理を行うプログラムを書いていきます (List 7.13, 実行結果: 図 7.4)。

List 7.13: main.lisp

```
1 ;;; グローバル変数/定数定義
2 ;; 2。斜め移動時の補正に使う
3 (defconstant +sqrt-2+ (sqrt 2))
4
5 (load "key-state.lisp" :external-format :utf-8)
6 (load "game-field.lisp" :external-format :utf-8)
7 (load "hit-test.lisp" :external-format :utf-8)
8 (load "player.lisp" :external-format :utf-8)
9 (load "enemy.lisp" :external-format :utf-8)
10 (load "enemy-manager.lisp" :external-format :utf-8)
11
12 (defun load-png-image (source-file)
13   (sdl:convert-to-display-format :surface (sdl:load-image source-file)
14     :enable-alpha t
15     :pixel-alpha t))
16
17 (defun main ()
18   (sdl:with-init ()
19     (sdl:window 640 480 :title-caption "てすと")
20     (sdl:initialise-default-font sdl:*font-10x20*) ; フォント初期化
21     (setf (sdl:frame-rate) 60)
22     (let* ((current-key-state (make-instance 'key-state))
23            ;; ゲーム画面オブジェクトを生成
24            (game-field (make-instance 'game-field
25                                     :width 640
26                                     :height 480
27                                     :x 0
28                                     :y 0))
29            ;; 自機オブジェクトを生成
30            (player (make-instance 'player
31                                  :x (/ (game-field-width game-field) 2)
32                                  :y (/ (game-field-height game-field) 2)
33                                  :width 64
34                                  :height 64
```

⁷(declare (ignore enemy)) とあるのは引数 `enemy` を使わないという宣言です。宣言しないと警告が出ます。意図的に使わないのが、本当に使い忘れたのが区別できるようにこういう宣言は書いておきましょう。

```

35         :speed 5
36         :damage-radius 10
37         :image (load-png-image "player.png"))
38     ;; 敵管理クラスを生成
39     (enemy-manager (make-instance 'enemy-manager))
40     ;; 自機存命フラグ
41     (player-is-alive t))
42 ;; 敵を生成
43 (dotimes (i 10)
44     (add-enemy enemy-manager
45         (make-instance
46             'enemy
47             :x (+ 32 (* i (/ (game-field-width game-field) 10)))
48             :y 0
49             :width 64
50             :height 64
51             :vx 1
52             :vy 1
53             :attack-radius 30
54             :damage-radius 20
55             :image (load-png-image "enemy.png"))))
56 (sdl:update-display)
57
58 (sdl:with-events ()
59     (:quit-event () t)
60     (:key-down-event (:key key)
61         (if (sdl:key= key :sdl-key-escape)
62             (sdl:push-quit-event)
63             (update-key-state key t current-key-state)))
64     (:key-up-event (:key key)
65         (update-key-state key nil current-key-state))
66     (:idle ()
67         (sdl:clear-display sdl:*black*)
68         ;; 自機が生きている時だけゲーム状態を更新
69         (when player-is-alive
70             ;; 移動キーの操作を移動量に反映
71             (update-player player current-key-state game-field)
72             ;; 敵の状態を更新
73             (update-enemies enemy-manager)
74             ;; 画面外に出た敵を成仏
75             (gc-enemies enemy-manager game-field)
76             ;; 敵対自機の当たり判定
77             (when (hit-test-enemies enemy-manager
78                 (damage-region player)
79                 :region-type 'attack)
80                 (setf player-is-alive nil))))
81
82     ;; 自機を描画
83     (draw-player player game-field)
84     ;; 敵を描画
85     (draw-enemies enemy-manager game-field)
86     ;; 死んだら残念な表示
87     (when (not player-is-alive)
88         (sdl:draw-string-solid-* "You died! Push Esc key..."
89             220 240))
90     (sdl:update-display))))))

```

まず初期化に関して。自機とか敵とかを生成する際に当たり判定に関するパラメータ(半径)を渡しています(36行目と53~54行目)。それから自機が生きているかどうかのフラグ player-is-alive を用意しています(41行目)。

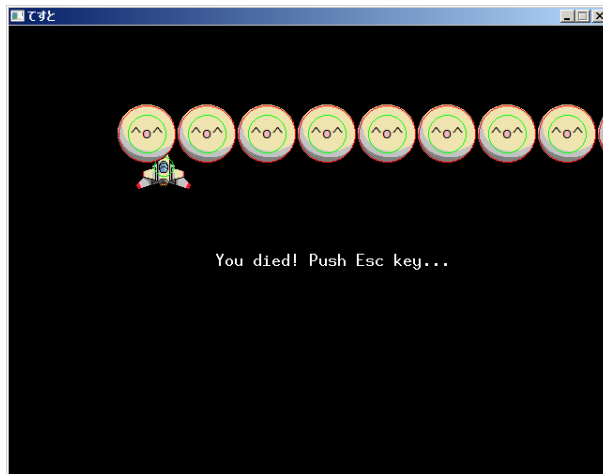


図 7.4: main.lisp の実行結果。敵にぶつくと自機は死ぬ。

次にゲームループでの処理について。ゲームループでの処理は自機が活着ているときとそうでないときに場合分けしています。自機が活着ているときはゲームの状態を更新して描画します。そうでないときはゲームの状態を更新せずに描画だけ行います。

当たり判定処理は 77~80 行目で行なっています。敵それぞれに対して自機のダメージ判定と敵の攻撃判定との間で当たり判定を行なって (77~79 行目)、当たっていた場合に自機の存命フラグを折ります (80 行目)。

87~89 行目では死亡時のテキスト表示を行なっています。ここでも自機の存命フラグを見えています。当たり判定の実装はこれで以上です。

7.5 まとめ

よく使う当たり判定は円形対円形と矩形対矩形。どちらを使うかはグラフィックとの相談。割と雑でもいい。

当たり判定ができあがったので次は弾を撃つことにします。

第8章 弾を撃つ

シューティングゲームと言うからには弾を撃たないといけませんね。ということで弾を撃つ処理を組んでいきます。

8.1 引き金を引くのは誰？

これから弾を撃つ処理を作っていくという訳ですが、実は弾の処理は敵の処理と似ています。弾も敵もユーザーの操作なしに動いて描画されて当たり判定を持っています。一度に複数個存在したりするところも同じですね。ということで弾の処理も似たような流れになることが想像できます。つまり弾のクラスと複数の弾を管理するクラスを作って、管理クラスを介して弾を制御することになります。

じゃあ敵と弾で違うところは何なんだ、って話になりますよね。それは誰によって生成されるかにあります。自機が放つ弾は自機が生成します。敵が撃つ弾は敵が生成します。弾は誰かが引き金を引いて放たれるものです。敵の場合はそうではありません。ゲームループの前の初期化の時点で生成していました。これが敵と弾との違いです。

という訳で弾を撃つ処理を作るにあたってはその弾を撃つ敵なり自機なりに弾を生成する処理を埋め込むことになります。

8.2 弾を作る

さっき弾と敵は処理が似てるって話がありましたね。これから弾のクラスを作っていきますが、ほとんど敵のクラスを作るときと同じノリで作っていきます。

List 8.1: bullet.lisp:弾クラスの定義

```
9 ;; 弾クラス
10 (defclass bullet ()
11   ((x
12    :initarg :x
13    :initform 0
14    :documentation "中心X座標 ")
15    (y
16     :initarg :y
17     :initform 0
18     :documentation "中心X座標 ")
19    (vx
20     :initarg :vx
21     :initform 0
22     :documentation "X方向速度 ")
23    (vy
24     :initarg :vy
25     :initform 0
26     :documentation "Y方向速度 ")
27    (width
28     :initarg :width
29     :initform 16
```

```

30     :documentation "幅")
31 (height
32   :initarg :height
33   :initform 16
34   :documentation "高さ")
35 (attack-region-radius
36   :initarg :attack-radius
37   :initform 10
38   :documentation "攻撃判定半径")
39 (attack-region
40   :accessor attack-region
41   :documentation "攻撃判定")
42 (alive
43   :accessor alive
44   :initform t
45   :documentation "弾の存命フラグ")
46 (image
47   :initarg :image
48   :initform (error "弾画像ファイルを指定しろ")
49   :documentation "画像"))

```

まずは弾クラスの定義です。ほとんど敵クラスと同じですね。違うのはダメージ判定系のスロットがないこと、alive というスロットがあることです¹。ダメージ判定系がないのは弾にダメージを与えることはないだろうということで納得できると思います。alive の方はなんなのかということこの弾がまだ必要かどうかを表しています。このフラグが折れている弾は画面外に出た敵のときと同様に管理クラスによって削除されます。

ではこの弾クラスに対してどんな操作を行うか見ていきます。

List 8.2: bullet.lisp:弾クラス総称関数

```

1 ;; 総称関数
2 (defgeneric update-bullet (bullet)
3   (:documentation "弾の状態を更新する"))
4 (defgeneric draw-bullet (bullet game-field)
5   (:documentation "弾を描画する"))
6 (defgeneric out-of-field-p (bullet game-field)
7   (:documentation "弾が画面外にいるかどうか判定"))

```

敵クラスとまるっきり同じです。次にメソッド定義を見ていきます。

List 8.3: bullet.lisp:弾クラスメソッド定義

```

51 ;; 弾クラス初期化処理
52 (defmethod initialize-instance :after ((bullet bullet) &key)
53   (with-slots (attack-region attack-region-radius x y) bullet
54     (setf attack-region (make-instance 'circle-region
55                                     :x x
56                                     :y y
57                                     :r attack-region-radius))))
58
59 (defmethod update-bullet ((bullet bullet))
60   (when (alive bullet)
61     (flet ((update-hit-region (x y region)
62             (setf (circle-x region) x)
63                   (setf (circle-y region) y)))
64       (with-slots (x y vx vy) bullet
65         (incf x vx)
66         (incf y vy)
67         (update-hit-region x y (attack-region bullet))))))

```

¹後で敵クラスの方にも alive スロットを追加するので、実質ダメージ判定系しか変わらないです。

```

68
69 (defmethod draw-bullet ((bullet bullet) (game-field game-field))
70   (with-slots (x y width height image) bullet
71     (sdl:draw-surface-at-* image
72       (round (+ (game-field-x game-field)
73                 (- x (/ width 2))))
74       (round (+ (game-field-y game-field)
75                 (- y (/ height 2)))))))
76
77 (defmethod out-of-field-p ((bullet bullet) (game-field game-field))
78   (with-slots (x y width height) bullet
79     (let ((width/2 (/ width 2))
80           (height/2 (/ height 2)))
81       (or (< (+ x width/2) 0)
82           (< (+ y height/2) 0)
83           (> (- x width/2) (game-field-width game-field))
84           (> (- y height/2) (game-field-height game-field))))))

```

これもほとんど敵クラスのと看と同じです。せいぜいダメージ判定関連の処理がなくなっているのと更新を行うのを alive フラグが立っているときだけにしているくらいしか違いませんね。alive フラグが折れているときに更新処理を行わないのは、既に不要な弾を更新するのは余計な手間だからサボってもいいだろうという考えでそうしています。

弾クラスについては以上です。

8.3 弾を管理する

弾クラスができたので次は弾を管理するクラスを作っていきます。この辺の考えも敵の管理のと看と同じですね。

List 8.4: bullet-manager.lisp:弾管理クラス定義

```

18 (defclass bullet-manager ()
19   ((bullets
20     :initform nil
21     :documentation "管理している弾"))

```

まずは弾管理クラスの定義から。これも敵管理クラスと同じで、管理する弾をリストで抱えます。このクラスに対して行う操作を挙げていきましょう。

List 8.5: bullet-manager.lisp:弾管理クラス総称関数

```

1 (defgeneric add-bullet (bullet-manager bullet)
2   (:documentation "弾を追加する"))
3 (defgeneric update-bullets (bullet-manager)
4   (:documentation "管理している弾を更新する"))
5 (defgeneric draw-bullets (bullet-manager game-field)
6   (:documentation "弾を描画する"))
7 (defgeneric gc-bullets (bullet-manager game-field)
8   (:documentation "不要になった弾を削除する"))
9 (defgeneric hit-test-bullets (bullet-manager region &key)
10  (:documentation "管理している弾との当たり判定"))

```

定義している操作も敵管理クラスと同じですね。

弾管理クラスではこれらに加えて以下のマクロを定義しておきます。

List 8.6: bullet-manager.lisp:弾管理クラス do-bullets マクロ

```

12 (defmacro do-bullets ((bullet-var bullet-manager) &body body)
13   (let ((bullets-symbol (gensym)))
14     `(with-slots ((,bullets-symbol bullets)) ,bullet-manager
15       (dolist (,bullet-var ,bullets-symbol)
16         ,@body))))

```

このマクロは弾管理クラスの bullets スロットに対する do-list のループをラップしたものです。要するに管理している弾1つ1つに対して何か処理する、というループを書くコードを書きやすくする代物です。メソッド定義や後の節のコードで使うために定義しました²。

それではメソッド定義を見ていきます。

List 8.7: bullet-manager.lisp:弾管理クラスメソッド定義

```

23 (defmethod add-bullet ((bullet-manager bullet-manager) (bullet bullet))
24   (with-slots (bullets) bullet-manager
25     (push bullet bullets)))
26
27 (defmethod update-bullets ((bullet-manager bullet-manager))
28   (do-bullets (bullet bullet-manager)
29     (update-bullet bullet)))
30
31 (defmethod draw-bullets ((bullet-manager bullet-manager)
32                          (game-field game-field))
33   (do-bullets (bullet bullet-manager)
34     (draw-bullet bullet game-field)))
35
36 (defmethod gc-bullets ((bullet-manager bullet-manager)
37                       (game-field game-field))
38   (with-slots (bullets) bullet-manager
39     (setf bullets
40           (delete-if #'(lambda (bullet)
41                         (or (not (alive bullet))
42                             (out-of-field-p bullet game-field)))
43                     bullets))))
44
45 (defmethod hit-test-bullets ((bullet-manager bullet-manager)
46                             (region circle-region)
47                             &key
48                             (on-hit-func
49                               #'(lambda (bullet)
50                                   (declare (ignore bullet))
51                                   nil))))
52   (with-slots (bullets) bullet-manager
53     (some #'(lambda (bullet)
54              (if (hit-test (attack-region bullet) region)
55                  (progn
56                    (funcall on-hit-func bullet)
57                    t)
58                  nil)))
59     bullets)))

```

各メソッドでやっている内容も敵管理クラスの場合とほとんど同じなので、どの辺が違うかを見ていきましょう。まず update-bullets(27行目) と draw-bullets(31行目) は処理の内容は敵管理クラスの場合と同じですが、さきほど定義した do-bullets マクロを使って書いています。次に gc-bullets(36行目) ですが、削除する対象として画面外の弾の他に alive スロットのフラグが折れている弾を加えて

²敵管理クラスで同じようなマクロを定義してもよかったんだけどメソッド定義以外で使うところがなげだだったので作りませんでした。

います (41 行目)。あとは hit-test-bullets(45 行目) のキーワード引数から region-type がなくなっています。これは弾が攻撃判定しか持っていないのでどの判定を使用するか指定する必要がないからです。弾管理クラスについてはこれで以上です。

8.4 引き金を引かせる

弾クラスも弾管理クラスもできたし弾を撃たせてみよう、言いたいところですがちょっと待ってください。冒頭にも言ったように弾を撃つには引き金を引く誰かが必要です。自機弾を撃つのは自機ですし、敵弾を撃つのは敵です。弾を撃たせるには自機なり敵なりに引き金を引けるようにしないとイケないですね。

8.4.1 自機弾の実装

まずは自機に弾を撃たせるようにします。この処理は自機の更新処理に埋め込みます。更新処理に弾の処理を埋め込む前に少し下準備をしておきます。

List 8.8: player.lisp:自機クラス定義

```
36 (update-frame
37   :accessor update-frame
38   :initform 0
39   :documentation "自機更新の時間軸")
40 (bullet-image
41   :initarg :bullet-image
42   :initform (error "自機弾画像ファイルを指定しろ")
43   :documentation "弾の画像")
```

まずは自機クラスに新たなスロットとして update-frame と bullet-image を追加します。update-frame は自機を更新するたびにカウントが 1 増えるスロットで、自機の更新に際して時間が関わってくるような処理を行うときには update-frame を時間として利用します。直接弾の処理に関わるスロットではないのですが、こういうスロットを用意した方が自機の更新に関する記述の幅が広がります。bullet-image は弾の画像を指定するスロットです。弾を撃つたびにファイルから画像をロードするのはしんどいので自機を生成する際に弾の画像もロードすることにします。

次に自機更新の総称関数について。

List 8.9: player.lisp:自機クラス総称関数

```
2 (defgeneric update-player (player key-state game-field bullet-manager)
3   (:documentation "自機の状態を更新する"))
```

今まで update-player は 3 つの引数を受け取っていましたが、新たに弾管理クラスのインスタンスを受け取るように変更します。

それではいよいよ自機更新メソッドに弾を撃つ処理を埋め込んでいきます。

List 8.10: player.lisp:自機クラス更新系

```
93 ;; 更新フレームを進める処理
94 (defmethod tick-update-frame ((player player))
95   (incf (update-frame player)))
96
97 ;; 弾を撃つ処理
98 (defmethod shoot ((player player) (bullet-manager bullet-manager))
99   (with-slots (x y bullet-image) player
100    (add-bullet bullet-manager
```

```

101         (make-instance 'bullet
102                       :x x
103                       :y y
104                       :vx 0
105                       :vy -10
106                       :width 16
107                       :height 16
108                       :attack-radius 8
109                       :image bullet-image)))
110
111 ;;; 自機の状態更新
112 (defmethod update-player ((player player)
113                          (key-state key-state)
114                          (game-field game-field)
115                          (bullet-manager bullet-manager))
116   (with-slots (shoot) key-state
117     (when (and shoot (zerop (mod (update-frame player) 4)))
118           (shoot player bullet-manager)))
119   (move-by-input player key-state)
120   (fix-position player game-field)
121   (update-region player)
122   (tick-update-frame player))

```

自機更新のメソッドに弾を撃つ処理 (116~18行目) と update-frame を更新する処理 (122行目) を追加しました。update-frame の更新はというと tick-update-frame メソッドの定義を見れば分かりますが、単に update-frame を 1 増やすだけです。

弾の発射の方の処理はというと shoot メソッドの定義の方に書いてあります。この中では弾クラスのインスタンスを生成して弾管理クラスの管理下に追加してやるということをやっています。つまりこの shoot メソッドを呼ぶと新たに弾が放たれる訳です。shoot メソッドは引き金なんですね。

対して引き金を引くのが自機更新の中の処理 (116~18行目) です。ここでは弾の発射に対応するキー入力があったときに shoot メソッドを呼んで弾を発射します。ただ、単に弾発射のキー入力押されているという条件だけではなくて「update-frame が 4 の倍数」という条件も満たすときだけ発射するようにしています。これは弾を断続的に撃つようにするための条件です。今回の場合だと自機の更新 4 回につき 1 発だけ撃つ、60FPS なら 1 秒間に 15 発までしか撃たなくなります。なぜこのように断続的にしているかということ、この方が弾を撃ってるっぽく見える、つまり表現上こうした方がいいという理由です。

8.4.2 敵弾の実装

自機だけ弾を撃てるのも不平等な気がするので敵も弾を撃てるようにします。やることは自機弾の場合と同じです。

ということでまずは敵クラスに update-frame スロットと bullet-image スロットを追加します。

List 8.11: enemy.lisp:敵クラス定義

```

53 (update-frame
54   :accessor update-frame
55   :initform 0
56   :documentation "敵更新の時間軸")
57 (bullet-image
58   :initarg :bullet-image
59   :initform (error "敵弾画像ファイルを指定しろ")
60   :documentation "弾の画像")

```

次に update-enemy の総称関数を変更して弾管理クラスを引数にとるようにします。

List 8.12: enemy.lisp:敵クラス総称関数

```

2 (defgeneric update-enemy (enemy bullet-manager)
3   (:documentation "敵の状態を更新する"))

```

そして update-enemy メソッドの定義に弾発射処理を埋め込みます。

List 8.13: enemy.lisp:敵クラス更新系

```

79 (defmethod update-enemy ((enemy enemy) (bullet-manager bullet-manager))
80   (flet ((update-hit-region (x y region)
81         (setf (circle-x region) x)
82               (setf (circle-y region) y)))
83     ;; 弾発射口ジックを組み込む
84     (when (zerop (mod (update-frame enemy) 100))
85       (with-slots (x y bullet-image) enemy
86         (add-bullet bullet-manager
87                     (make-instance 'bullet
88                                     :x x
89                                     :y y
90                                     :vx 0
91                                     :vy 2
92                                     :width 16
93                                     :height 16
94                                     :attack-radius 8
95                                     :image bullet-image))))
96     (with-slots (x y vx vy) enemy
97       (incf x vx)
98       (incf y vy)
99       (update-hit-region x y (attack-region enemy))
100      (update-hit-region x y (damage-region enemy))))
101   (incf (update-frame enemy)))

```

101 行目で update-frame スロットを 1 増やしているのは自機の場合と同じですね。

弾の発射処理は 84~95 行目で行なっています。特にキー入力を拾わないところ以外は自機弾のときと同じです。今回は 100 フレームに 1 回弾を撃つようにしています (84 行目)。

今のところそこまで見通しの悪いコードではないのでリファクタリングはしません。

敵クラスに関してはこれで OK ですが、敵の管理は敵管理クラスが行なっていて敵の状態更新も敵管理クラスが一手に引き受けています。ということで敵管理クラスの方にも手を入れる必要があります。とはいってもやることは単純で、update-enemy メソッドを呼ぶときに弾管理クラスも引数として渡すようにするというだけです。

List 8.14: enemy-manager.lisp:敵管理クラス総称関数

```

4 (defgeneric update-enemies (enemy-manager bullet-manager)
5   (:documentation "敵の状態を更新する"))

```

まずは update-enemies も弾管理クラスを引数にとるようにしておきます。

List 8.15: enemy-manager.lisp:敵管理クラス更新系

```

23 (defmethod update-enemies ((enemy-manager enemy-manager) (bullet-manager
24   bullet-manager))
25   (with-slots (enemies) enemy-manager
26     (dolist (enemy enemies)
27       (update-enemy enemy bullet-manager))))

```

そして update-enemies メソッドの中で update-enemy メソッドを呼ぶときに弾管理クラスを渡すようにします。これで敵管理クラスの方も OK です。

8.5 撃たれる準備

前のセクションでひと通り弾を撃つ処理はできました。これで弾を撃てます。が、まだ1つやり残していることがあります。それは弾に撃たれたときの処理の準備です。敵が弾に撃たれたら、その敵には消えてもらいたい。そのためには少し準備が必要です。

というわけでまずは敵クラスの方に手をいれます。

List 8.16: enemy.lisp:敵クラス定義

```
49 (alive
50   :accessor alive
51   :initform t
52   :documentation "存命フラグ")
```

新たに `alive` というスロットを追加しました。これは敵がまだ必要かどうかのフラグです。弾クラスの定義でも出てきましたね。敵が被弾したときにはこのフラグを折ってしまいます。

続いて敵管理クラスの方に参ります。

List 8.17: enemy-manager.lisp:敵管理クラス削除系

```
33 (defmethod gc-enemies ((enemy-manager enemy-manager) (game-field game-field))
34   (with-slots (enemies) enemy-manager
35     (setf enemies
36           (delete-if #'(lambda (enemy)
37                         (or (not (alive enemy))
38                             (out-of-field-p enemy game-field)))
39                       enemies))))
```

不要な敵を削除する処理ですが、ここで削除する対象として画面外の敵の他に `alive` スロットのフラグが折れている敵も対象にするようにしています(37行目)。この辺も弾管理クラスの場合と同じですね。以上で弾で撃たれる準備も整いました。

8.6 弾を撃つ

ひと通り弾に関する処理の準備が整ったので弾を撃つプログラムを書いていきます。

まずはキー入力に弾を撃つキーを追加しておきます。Zキーで弾を撃つようにしました。

List 8.18: key-state.lisp:キー入力管理クラス

```
17 ;; キー入力の状態クラス
18 (defkeystate key-state
19   (up      :sdl-key-up)
20   (down    :sdl-key-down)
21   (right   :sdl-key-right)
22   (left    :sdl-key-left)
23   (shoot   :sdl-key-z))
```

では本題のプログラムの方を書いていきます(実行結果:図 8.1)。

List 8.19: main.lisp

```
1 ;;; グローバル変数/定数定義
2 ;; 2. 斜め移動時の補正に使う
3 (defconstant +sqrt-2+ (sqrt 2))
4
5 (load "key-state.lisp" :external-format :utf-8)
6 (load "game-field.lisp" :external-format :utf-8)
```

```

7 (load "hit-test.lisp" :external-format :utf-8)
8 (load "bullet.lisp" :external-format :utf-8)
9 (load "bullet-manager.lisp" :external-format :utf-8)
10 (load "player.lisp" :external-format :utf-8)
11 (load "enemy.lisp" :external-format :utf-8)
12 (load "enemy-manager.lisp" :external-format :utf-8)
13
14 (defun load-png-image (source-file)
15   (sdl:convert-to-display-format :surface (sdl:load-image source-file)
16     :enable-alpha t
17     :pixel-alpha t))
18
19 (defun main ()
20   (sdl:with-init ()
21     (sdl:window 640 480 :title-caption "てすと")
22     (sdl:initialise-default-font sdl:*font-10x20*) ; フォント初期化
23     (setf (sdl:frame-rate) 60)
24     (let* ((current-key-state (make-instance 'key-state))
25            ;; ゲーム画面オブジェクトを生成
26            (game-field (make-instance 'game-field
27              :width 640
28              :height 480
29              :x 0
30              :y 0))
31            ;; 自機オブジェクトを生成
32            (player (make-instance 'player
33              :x (/ (game-field-width game-field) 2)
34              :y (/ (game-field-height game-field) 2)
35              :width 64
36              :height 64
37              :speed 5
38              :damage-radius 10
39              :image (load-png-image "player.png")
40              :bullet-image
41              (load-png-image "bullet-2.png"))))
42            ;; 敵管理クラスを生成
43            (enemy-manager (make-instance 'enemy-manager))
44            ;;;; 自機弾管理
45            (player-bullet-manager (make-instance 'bullet-manager))
46            ;;;; 敵弾管理
47            (enemy-bullet-manager (make-instance 'bullet-manager))
48            ;; 自機存命フラグ
49            (player-is-alive t))
50      ;; 敵を生成
51      (dotimes (i 10)
52        (add-enemy enemy-manager
53          (make-instance
54            'enemy
55            :x (+ 32 (* i (/ (game-field-width game-field) 10)))
56            :y 0
57            :width 64
58            :height 64
59            :vx 1
60            :vy 1
61            :attack-radius 30
62            :damage-radius 20
63            :image (load-png-image "enemy.png")
64            :bullet-image (load-png-image "bullet-1.png"))))
65      (sdl:update-display)
66
67      (sdl:with-events ()

```

```

68     (:quit-event () t)
69     (:key-down-event (:key key)
70      (if (sdl:key= key :sdl-key-escape)
71          (sdl:push-quit-event)
72          (update-key-state key t current-key-state)))
73     (:key-up-event (:key key)
74      (update-key-state key nil current-key-state))
75     (:idle ()
76      (sdl:clear-display sdl:*black*)
77      ;; 自機が生きている時だけゲーム状態を更新
78      (when player-is-alive
79       ;; 移動キーの操作を移動量に反映
80       (update-player player current-key-state
81        game-field player-bullet-manager)
82       ;; 敵の状態を更新
83       (update-enemies enemy-manager enemy-bullet-manager)
84       ;; 自機弾更新
85       (update-bullets player-bullet-manager)
86       ;; 敵弾更新
87       (update-bullets enemy-bullet-manager)
88       ;; 自機弾対敵当たり判定
89       (do-bullets (bullet player-bullet-manager)
90        (when (hit-test-enemies enemy-manager
91         (attack-region bullet)
92         :region-type 'damage
93         :on-hit-func
94         #'(lambda (enemy)
95             ;; 当たった敵は死ぬ
96             (setf (alive enemy) nil))))
97         ;; 敵を当てた弾を消す
98         (setf (alive bullet) nil)))
99       ;; 敵弾対自機当たり判定
100      (when (hit-test-bullets enemy-bullet-manager
101       (damage-region player))
102       (setf player-is-alive nil))
103      ;; 画面外に出た敵や死んだ敵を成仏
104      (gc-enemies enemy-manager game-field)
105      (gc-bullets player-bullet-manager game-field)
106      (gc-bullets enemy-bullet-manager game-field)
107      ;; 敵対自機の当たり判定
108      (when (hit-test-enemies enemy-manager
109       (damage-region player)
110       :region-type 'attack)
111       (setf player-is-alive nil)))
112
113      ;; 自機弾を描画
114      (draw-bullets player-bullet-manager game-field)
115      ;; 自機を描画
116      (draw-player player game-field)
117      ;; 敵を描画
118      (draw-enemies enemy-manager game-field)
119      ;; 敵弾を描画
120      (draw-bullets enemy-bullet-manager game-field)
121      ;; 死んだら残念な表示
122      (when (not player-is-alive)
123       (sdl:draw-string-solid-* "You died! Push Esc key..."
124        220 240))
125      (sdl:update-display))))))

```

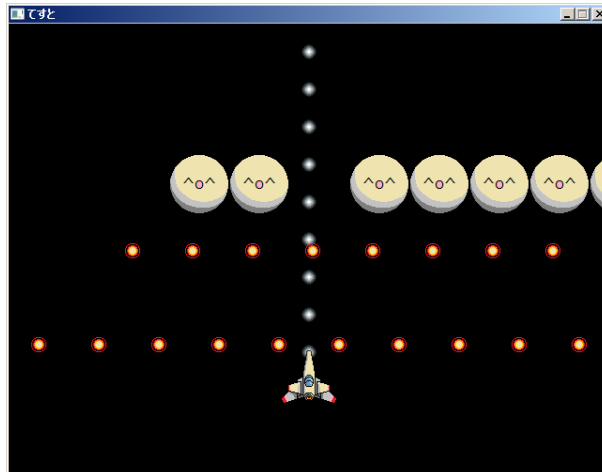


図 8.1: main.lisp の実行結果。

弾に関する処理は大きく分けて以下のようになります。

- 弾管理クラスのインスタンス生成
- 弾の更新
- 弾との当たり判定処理
- 不要な弾の削除
- 弾の描画

まず弾管理クラスのインスタンス生成ですが、これは 45 行目と 47 行目で行なっています。ここでは自機弾の管理用と敵弾の管理用とで 2 つのインスタンスを生成しています。

次に弾の更新ですが、これは 85 行目と 87 行目の `update-bullets` メソッドを呼ぶ所が該当します。更新に関する処理はすべてこのメソッドが担っています。

次に弾との当たり判定ですが、これには自機弾と敵との当たり判定、敵弾と自機との当たり判定という 2 種類があります。まずは自機弾と敵との当たり判定処理 (89~98 行目) を見ていきます。ここでは `do-bullets` を使って自機弾それぞれについて `hit-test-enemies` メソッドを呼ぶという形で記述しています。敵と自機弾が当たった際には当たった敵と弾の両方の `alive` スロットを `nil` にして、それぞれを削除します。次に敵弾と自機との当たり判定処理 (100~102 行目) を見ていきます。こちらは敵弾と自機が当たった時に `player-is-alive` を `nil` にしています。

弾に関する更新と当たり判定が終わったところで不要な弾を削除しているのが 105~106 行目の処理です。最後に 114 行目と 120 行目で弾を描画しています。描画するときの順番で絵の重なりを決めています。

8.7 まとめ

弾クラスやそれを管理するクラスを作って、弾を撃たせるための処理を自機や敵に埋め込んだ。これで弾を撃つプログラムも書けるようになった。

次はステージ組んだり画像のロードを管理したりします。

第9章 ローディングとステージ

敵を弾で撃ったりできるようになったのでこれでもうシューティングゲームを名乗ることはできるのですが、まだ色々問題があります。例えばこれまでの実装だと敵が出現するタイミングを変えられませんが、ゲーム開始時にいきなり全部の敵が出てきます。これはちょっと頂けませんね。敵の出現するタイミングについても管理する仕組みが必要ですね。他にも管理する必要があるものがあります。今まで画像ファイルのロードはその場その場で `load-png-image` 関数を呼んでいましたが、画像のローディングも一元管理できた方がいいです。ということでこの章ではこの2つの管理の話をしていきます¹。

9.1 画像読み込みの管理

9.1.1 なんで管理したいの？

具体的なコードの話の前に、なんで画像読み込みを管理したいか、という話をしておきます。

そもそも画像を何に使うかと言えば描画のために行います。つまり実際に画像を使って何かやるのは描画処理のときです。一方で画像を読み込みたいタイミングというのはゲームループに入る前、初期化する時です。そうなるとう画像を描画しようと思ったら、初期化のときにロードした画像を「どこか」に格納してゲームループに入り、描画するときにその「どこか」から取り出すといったことをすることになります。ではその「どこか」というのをどこにすればいいかという話になります。

これまでの自機や敵の描画の場合はそれぞれ自機クラスや敵クラスに格納していました。これらのクラスはゲームループに入る前の初期化の段階でインスタンスを生成していたので画像をロードしたらすぐにこの「どこか」に放り込めた訳です。

弾の描画の場合はどうだったか。弾の画像は弾クラスに格納されていましたが、弾クラスのインスタンスが生成されるのはゲームループの中です。画像をロードするタイミングである初期化から弾クラスのインスタンス生成までは時間が空いているのです。この空いている時間の間、ロードした弾画像をどこに格納していたかという弾を発射する敵や自機のクラスです。これによって画像をロードしてから弾を生成するまでの「どこか」をどうするかという問題を誤魔化すことができたのですが、あまり綺麗な解決ではないです。

この問題は敵の出現するタイミングを自由に変えられるようにした場合に致命的になります。敵の出現するタイミングを変えることは敵クラスのインスタンスが作られるタイミングを変えることと同じです。今までゲームループ前の段階で敵クラスのインスタンスが作られていたことによって敵画像や敵の弾画像に関する「どこか」の問題を解決していた訳ですが、この解決がうまくいなくなるのです。

画像を読み込んでから敵クラスなどのインスタンスを生成するまでの間に生じる「どこか」の問題、これを解決するのが画像読み込みの管理です。言ってしまうと画像を読み込んでから然るべきクラスのインスタンスが生成されるまで画像を預かっておくというのが画像読み込み管理のお仕事です。

¹なんでこの2つを1つの章でまとめてやるかということ、ステージを作る話を書こうとコードを書く `create-enemy-from-schedule` 関数を書いているあたりで画像のロード管理が欲しくなった。じゃあローディングもやろう、という流れでこうなりました。いい加減ですね。

9.1.2 管理しよう

画像読み込み管理したいって思いは伝わったと思うので、さっそくコードを書いてみます。

List 9.1: loader.lisp:画像読み込み管理

```
1 (defgeneric add-image (image-loader key image-file)
2   (:documentation "ロードする画像を登録する"))
3 (defgeneric get-image (image-loader key)
4   (:documentation "ロードした画像を取得する"))
5
6 (defclass image-loader ()
7   ((images
8     :initform nil
9     :documentation "ロードした画像"))))
10
11 (defun load-png-image (source-file)
12   (sdl:convert-to-display-format :surface (sdl:load-image source-file)
13     :enable-alpha t
14     :pixel-alpha t))
15
16 (defmethod add-image ((image-loader image-loader) key image-file)
17   (with-slots (images) image-loader
18     (if (getf images key)
19         (error "keyが重複してます。~a" key)
20         (setf (getf images key)
21               (load-png-image image-file)))))
22
23 (defmethod get-image ((image-loader image-loader) key)
24   (with-slots (images) image-loader
25     (getf images key)))
```

画像読み込み管理のコードはこれですべてです。では詳しく見ていきましょう。

画像読み込み管理を行うクラスとして `image-loader` を定義しました (6~9 行目)。このクラスは属性リストのスロット `images` を持っていて、この中にロードした画像を格納していきます²。このクラスに対して行う操作は2つ、読み込む画像を登録することと読み込んだ画像を取得することです。登録する方が `add-image` メソッドで取得する方が `get-image` メソッドです。具体的に何をやりたいのかというと `add-image` メソッドに読み込みたい画像ファイルとキーを指定して呼ぶと、あとで `get-image` メソッドにキーを渡して呼んだら読み込んだ画像が手に入るようにしたいのです。

`add-image` メソッドの定義は 16~21 行目ですね。引数には `image-loader` のインスタンスの他にキーと画像ファイル名を指定します。やっていることは指定したキーがすでに `images` スロットにあるかどうか調べて、あるときはエラーを吐いて終了、なければ画像を `load-png-image` 関数でロードして指定したキーで `images` に追加します。

`get-image` メソッドの定義は 23~25 行目です。単に引数で渡されたキーで `images` から対応する要素を引っ張ってくるだけです。ろくにエラー処理してないのはアレですがまあこれで必要な動作はするので放っておきます。

実際にこれらのコードを使うときはゲームループに入る前の初期化処理のときに

```
(add-image an-image-loader-instance :player-image "player.png")
```

のようにキーと画像ファイル名を指定して呼んで、読み込んだ画像を取得するときは

```
(get-image an-image-loader-instance :player-image)
```

という風に呼べば取得できます。

²属性リストよりハッシュテーブルの方がパフォーマンスがいい気がしますが、ゲーム全体のパフォーマンスから言ったら微々たる差だろうということで放置しています。実際管理する画像の数もたかが知れてるだろうし。

9.2 ステージ

敵が出てくるタイミングとか場所とか自由に弄れるようにしたい、というのはつまりステージを作れるようにしたいということです。ということでここからはステージを作るための仕組みを作っていきます。

9.2.1 考え方

タイミングを考えるとということは時間軸が必要ですね。ということで自機や敵の更新で導入したようなフレームカウントが必要になるだろうということが予想できます。ゲームループが回るたびにフレームカウントを増やして行って、ある値になったら敵を出すという風にすれば敵を出すタイミングを変えられそうです。敵を出すタイミングのフレームカウントとその敵のパラメータの組があればいつでも敵を出すかなど決められます。これらを組み合わせればステージが作れますね。

9.2.2 実装

考え方が伝わったところでステージまわりのコードを書いていきます。ステージを表すクラス `stage` を作ることにします。コードを書く前にこれから作る `stage` クラスをどう使うかを見ておきます。

まずインスタンスを生成するときには

```
(let ((stage (make-instance 'stage
                            :schedule
                            ;; :frame 敵が出現するタイミング
                            ;; :x     敵のX座標
                            ;; :y     敵のY座標
                            ;; :vx    敵のX方向速度
                            ;; :vy    敵のY方向速度
                            '((:frame 30 :x 0 :y 0 :vx 0 :vy 2)
                              (:frame 60 :x 320 :y 0 :vx 0 :vy 2))))
    ...
)
```

のように敵が出現するタイミングと位置などのパラメータの組を `:schedule` キーワード引数で渡します。ゲームループ中では

```
(update-stage stage enemy-manager image-loader)
```

のように `update-stage` に敵管理クラスのインスタンスと画像読み込み管理クラスのインスタンスを渡して呼ぶと、呼び出したタイミングに合わせて敵を追加していきます。

こういった使い方を想定した上で `stage` クラスまわりのコードを書いていきます。

List 9.2: `stage.lisp`:ステージクラスまわり

```
1 ;; 総称関数
2 (defgeneric update-stage (stage enemy-manager loader)
3   (:documentation "ステージの更新"))
4
5 ;; ステージクラス定義
6 (defclass stage ()
7   ((frame
8     :initform 0
9     :accessor frame
10    :documentation "ステージの時間軸")
11   (current-schedule-index
12     :initform 0
13     :documentation "現在のスケジュールを指すインデックス"))
```

```

14     (schedule
15       :initarg :schedule
16       :initform nil
17       :documentation "敵が出現するスケジュール"))
18
19 (defun get-frame-from-schedule (s)
20   (getf s :frame))
21
22 (defun create-enemy-from-schedule (s loader)
23   (make-instance
24     'enemy
25     :x (getf s :x)
26     :y (getf s :y)
27     :width 64
28     :height 64
29     :vx (getf s :vx)
30     :vy (getf s :vy)
31     :attack-radius 30
32     :damage-radius 20
33     :image (get-image loader :enemy)
34     :bullet-image (get-image loader :enemy-bullet)))
35
36 (defmethod initialize-instance :after ((stage stage) &key)
37   (with-slots (schedule) stage
38     (setf schedule (sort schedule #'< :key #'get-frame-from-schedule))))
39
40 (defmethod update-stage ((stage stage)
41                          (enemy-manager enemy-manager)
42                          (loader image-loader))
43   (with-slots (schedule frame current-schedule-index) stage
44     (loop for s in (nthcdr current-schedule-index schedule)
45           if (= (get-frame-from-schedule s) frame)
46             do (progn
47                 (incf current-schedule-index)
48                 (add-enemy enemy-manager
49                           (create-enemy-from-schedule s loader))))
49     (incf frame)))
50

```

まずはクラス定義 (6~17 行目) ですが、stage クラスは3つのスロットを持ちます。frame がステージの時間経過を表すフレームカウント、schedule が敵の出現するタイミングや位置などのパラメータを管理しているリストを持ちます。current-schedule-index は schedule を走査するための補助的なスロットです。

次にインスタンス生成時の初期化処理 (36~38 行目) を見ていきます。ここでは schedule スロットに入っているリストを:frane の小さい順にソートしておきます。なぜソートするかというと update-stage メソッドの処理は schedule がソートされていることを前提に書いているからです。

次は update-stage メソッドの定義 (40~50 行目) を見ていきましょう。ここでやっていることは2つ、frame を1増やすこと (50 行目) と schedule の中から:frane の値が frame スロットと一致するものを見つけ出して、そのパラメータから敵を生成すること (44~49 行目) です。敵を生成して追加する処理は 48~49 行目で行なっていて、敵生成の細かい内容は create-enemy-from-schedule 関数に投げられています。

create-enemy-from-schedule 関数は (:frame ~ :x ~ :y ~ :vx ~ :vy ~) といった形の属性リストを受け取って敵クラスのインスタンスを生成します。敵を生成するに当たって画像を取得する必要があるので前の節で定義した画像読み込み管理クラスを利用しています。

ステージに関する実装はこれで以上です。

9.3 ローディングとステージを使う

ここまででローディングやステージのための仕組みが整いました。では実際のゲームでこれらを使ってみます。

List 9.3: main.lisp

```
1 ;;; グローバル変数/定数定義
2 ;; 2. 斜め移動時の補正に使う
3 (defconstant +sqrt-2+ (sqrt 2))
4
5 (load "loader.lisp" :external-format :utf-8)
6 (load "key-state.lisp" :external-format :utf-8)
7 (load "game-field.lisp" :external-format :utf-8)
8 (load "hit-test.lisp" :external-format :utf-8)
9 (load "bullet.lisp" :external-format :utf-8)
10 (load "bullet-manager.lisp" :external-format :utf-8)
11 (load "player.lisp" :external-format :utf-8)
12 (load "enemy.lisp" :external-format :utf-8)
13 (load "enemy-manager.lisp" :external-format :utf-8)
14 (load "stage.lisp" :external-format :utf-8)
15
16 (defun main ()
17   (sdl:with-init ()
18     (sdl:window 640 480 :title-caption "てすと")
19     (sdl:initialise-default-font sdl:*font-10x20*)
20     (setf (sdl:frame-rate) 60)
21     ;; ローダー生成
22     (let ((loader (make-instance 'image-loader)))
23       ;; 画像を読み込んでいく
24       (add-image loader :player "player.png")
25       (add-image loader :enemy "enemy.png")
26       (add-image loader :player-bullet "bullet-2.png")
27       (add-image loader :enemy-bullet "bullet-1.png")
28       (let* ((current-key-state (make-instance 'key-state))
29              (game-field (make-instance 'game-field
30                                         :width 640
31                                         :height 480
32                                         :x 0
33                                         :y 0))
34              (player (make-instance 'player
35                                    :x (/ (game-field-width game-field) 2)
36                                    :y (/ (game-field-height game-field) 2)
37                                    :width 64
38                                    :height 64
39                                    :speed 5
40                                    :damage-radius 10
41                                    :image (get-image loader :player)
42                                    :bullet-image
43                                    (get-image loader :player-bullet)))
44              (enemy-manager (make-instance 'enemy-manager))
45              (player-bullet-manager (make-instance 'bullet-manager))
46              (enemy-bullet-manager (make-instance 'bullet-manager))
47              ;; ステージ
48              (stage (make-instance 'stage
49                                   :schedule
50                                   '((:frame 0 :x 0 :y 0 :vx 2 :vy 2)
51                                     (:frame 60 :x 0 :y 0 :vx 1.5 :vy 2)
52                                     (:frame 120 :x 0 :y 0 :vx 1 :vy 2)
53                                     (:frame 180 :x 0 :y 0 :vx 0.5 :vy 2))))
54         (player-is-alive t))
```

```

55     (sdl:update-display)
56
57     (sdl:with-events ()
58       (:quit-event () t)
59       (:key-down-event (:key key)
60         (if (sdl:key= key :sdl-key-escape)
61             (sdl:push-quit-event)
62             (update-key-state key t current-key-state)))
63       (:key-up-event (:key key)
64         (update-key-state key nil current-key-state))
65       (:idle ()
66         (sdl:clear-display sdl:*black*)
67         (when player-is-alive
68           ;; ステージ更新
69           (update-stage stage enemy-manager loader)
70           (update-player player current-key-state
71             game-field player-bullet-manager)
72           (update-enemies enemy-manager enemy-bullet-manager)
73           (update-bullets player-bullet-manager)
74           (update-bullets enemy-bullet-manager)
75           (do-bullets (bullet player-bullet-manager)
76             (when (hit-test-enemies enemy-manager
77                 (attack-region bullet)
78                   :region-type 'damage
79                   :on-hit-func
80                   #'(lambda (enemy)
81                       (setf (alive enemy) nil))))
82             (setf (alive bullet) nil)))
83           (when (hit-test-bullets enemy-bullet-manager
84                 (damage-region player))
85             (setf player-is-alive nil))
86           (gc-enemies enemy-manager game-field)
87           (gc-bullets player-bullet-manager game-field)
88           (gc-bullets enemy-bullet-manager game-field)
89           (when (hit-test-enemies enemy-manager
90                 (damage-region player)
91                   :region-type 'attack)
92             (setf player-is-alive nil)))
93
94         (draw-bullets player-bullet-manager game-field)
95         (draw-player player game-field)
96         (draw-enemies enemy-manager game-field)
97         (draw-bullets enemy-bullet-manager game-field)
98         (when (not player-is-alive)
99           (sdl:draw-string-solid-* "You died! Push Esc key..."
100             220 240))
101         (sdl:update-display))))))

```

新しい所としては、22行目で画像読み込み管理クラスのインスタンスを生成して、24~27行目で管理する画像を追加しています。読み込んだ画像の取得は43行目などで行なっています。それから48~53行目ではステージクラスのインスタンスを生成しています。ここでは敵が4匹、タイミングや速度をずらして出現させるように記述しています。ゲームループの中ではステージの更新を行います(69行目)。これによって適切なタイミングで敵が出現するようになります。

9.4 まとめ

ローディングとかステージの仕組みを組みました。もうこれで完成でいいよね…。

Q and A

ここではさっきまでの章で扱わなかった (扱えなかった?) 話について軽く紹介しておきます。

Q. タイトル画面とかゲームオーバー画面とか作りたい

A. 画面遷移の話ですね。今までのコードではゲームプレイ中の画面での処理しか扱っていませんでしたが実際のゲームでは他の画面もあるわけで、それぞれの画面で処理を切り替える必要があります。

画面遷移の話は状態遷移として考えるのが分かりやすいでしょう。状態遷移については例えばデザインパターンだと State パターンを使って実装するといったことが考えられます。ゲームループの中 (:idle ~) ですね) にベタに書いているコードを何かしらのクラスのメソッドとして切り離すことからスタートして組んでいくといいと思います。

Q. 音まわりの扱いが知りたい

A. 基本的に描画と同じように扱えばいいと思います。ただ、音にせよ描画にせよ割と使うライブラリに左右される話なので最終的には自分で調べるという話になってしまいますね。

Q. 被弾したときにアニメーションが欲しい

A. この辺の話も最終的には状態遷移の話に行き着きます。例えば敵が被弾した場合だと「被弾前 (生きてる状態) 被弾後 (被弾したアニメーションしてる状態) アニメーション後 (描画されない、不要になった状態)」といった遷移をすると考えます。各状態ごとに更新や描画の内容を切り替えるようにすれば OK だと思います。

Q. リプレイを作りたい

A. リプレイをどうやって実装するかというとゲーム開始から終了までのキー入力と入力したタイミングを保存して、リプレイ再生のときにどのキーを押したか、どのタイミングで押したかを再現することで実現します。この冊子のコードだと key-state クラスあたりをゴニョゴニョ弄れば実装できるはず

です。
ただ、1つ気をつけないといけないことがあります。それは乱数の使用です。ゲーム中で乱数を使用している場合、リプレイの記録時と再生時で違う乱数が使われたらプレイを再現できません。そこで必ず同じ乱数が使われるように乱数の種というものをリプレイデータに保存します。コンピュータが扱う乱数というのは何かしらのアルゴリズムによって生成されるものです。そのアルゴリズムに与えるパラメータが同じなら、同じ乱数 (列) を生成します。このパラメータが乱数の種です。プレイ記録時の乱数の種を再生時にも使えば同じ乱数が得られるのでプレイを再現できるようになるという訳です。