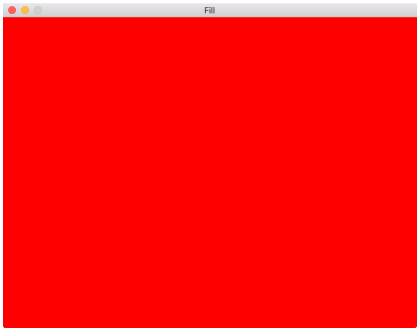


図 1　fill.go の実行結果 (35 ページ)

図 2　ebiten.png

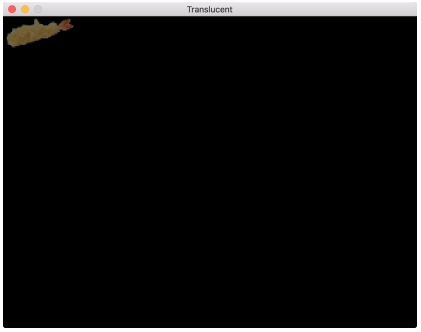


図 3　translucent.go の実行結果 (76 ページ)

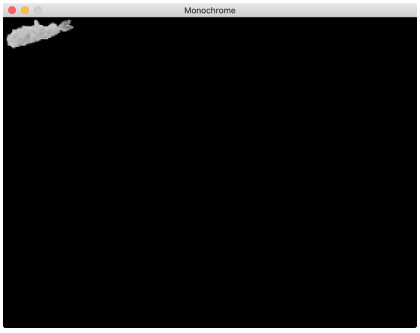


図 4　monochrome.go の実行結果 (78 ページ)

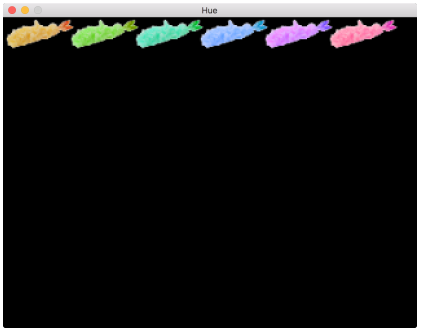


図 5　hue.go の実行結果 (81 ページ)

図 6　gophers_photo.jpg

図 7　sepia.go の実行結果 (83 ページ)

Ebiten

Go で作る 2D ゲーム

星一 著

はじめに

この本は拙作のゲームライブラリ Ebiten (https://github.com/hajimehoshi/ebiten) について解説する本です。

Ebiten はプログラミング言語 Go を用いて 2D ゲームを作ることが出来ます。スーパーファミコンのような 16bit 風のレトロな 2D ゲームを実装することを主目的としています。ただ、解像度の制限などは特になく、汎用的な 2D ゲームライブラリとしても使用できます。API がシンプルであること、マルチプラットフォームであることが特徴です。

本書では、Ebiten のインストール方法をはじめ、Ebiten の基本的な知識を各論的に紹介します。前提知識としては、プログラミング言語 Go の知識、高校数学程度の知識を仮定しています。各機能の解説には、なるべくその機能を使ったコードを掲載しています。

本書に掲載するコードは、断片的にせずに、なるべく自己完結するように努めました。前後の文脈を完全に理解しなくとも読めるようにするためです。記載のコードのままコンパイルして動くはずです。また、本書のコードは GitHub のリポジトリ (https://github.com/hajimehoshi/ebiten-book-code) にて公開しています。コードのライセンスは Unlicense (Public Domain) です。

本書が Ebiten に触れるきっかけになれば幸いです。

目次

はじめに **v**

第 1 章 Ebiten 概説 **1**

1.1 特徴 2
1.2 実例 3
1.3 パッケージ 7
1.4 Ebiten の背景 8
1.5 Ebiten のポリシー 9
1.6 コミュニティ 11

第 2 章 インストール **13**

2.1 事前準備 14
2.2 Ebiten のインストール 18
2.3 サンプルの実行 19
2.4 Android/iOS 21

第 3 章 ゲームループ **23**

3.1 最初の Ebiten プログラム 24
3.2 FPS 26
3.3 Hello, World! 28
3.4 ブラウザでの実行 30
3.5 更新 31

第 4 章 描画・基礎 **33**

4.1 ebiten.Image 構造体 34
4.2 塗りつぶし 35
4.3 画像 37
4.4 座標系 40
4.5 行列 41
4.6 描画位置 48
4.7 拡大・縮小 52
4.8 フィルタ 57
4.9 回転 59

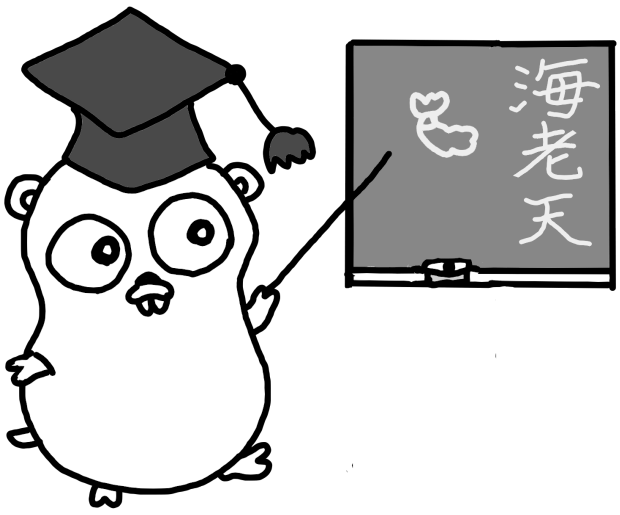
第 5 章 描画・応用 63
5.1 オフスクリーンレンダリング 64
5.2 オフスクリーンバッファの使い回し 68
5.3 モザイク 71
5.4 色変換行列 74
5.5 半透明 76
5.6 モノクロ 78
5.7 色相 81
5.8 セピア 83
5.9 部分描画 86
5.10 大量スプライト 90
5.11 ラスタスクロール 93
5.12 その他の機能 96

第 6 章 入力 97
6.1 キーボード 98
6.2 マウス 102
6.3 ゲームパッド 104
6.4 タッチ 107

第 1 章

Ebiten 概説

Ebiten はプログラミング言語 Go のシンプルな 2D ゲームライブラリです。この章では、Ebiten はどんなライブラリなのかについて概説します。また Ebiten を使った実例を紹介します。



1.1　特徴

Ebiten の特徴としては次の通りです。

シンプルな API　Ebiten の API は非常にシンプルです。たとえばピクセルの集合を表す構造体は `Image` 構造体しかなく、描画先および描画元はどちらもその構造体で表されます。また、ほぼあらゆる描画は `Image` オブジェクトから `Image` オブジェクトへの操作として表現されます。Ebiten は実装に OpenGL を使用していますが、Ebiten を使うのに OpenGL の知識は一切必要ありません。

マルチプラットフォーム　Windows、macOS、Linux といったデスクトップはもちろんのこと、Android、iOS 上で動くゲームが作れます。また、GopherJS (`http://www.gopherjs.org/`) というトランスパイラを使ってブラウザ上で動かすことも出来ます。

オープンソース　Apache License 2.0 を採用しました。Ebiten のソースコードはすべて GitHub リポジトリ (`https://github.com/hajimehoshi/ebiten`) 上で公開されています。ライセンスの範囲内で自由に改変可能です。コントリビューションも大歓迎です。

1.2 実例

1.2.1 いの べーしょん 2007

https://github.com/hajimehoshi/go-inovation

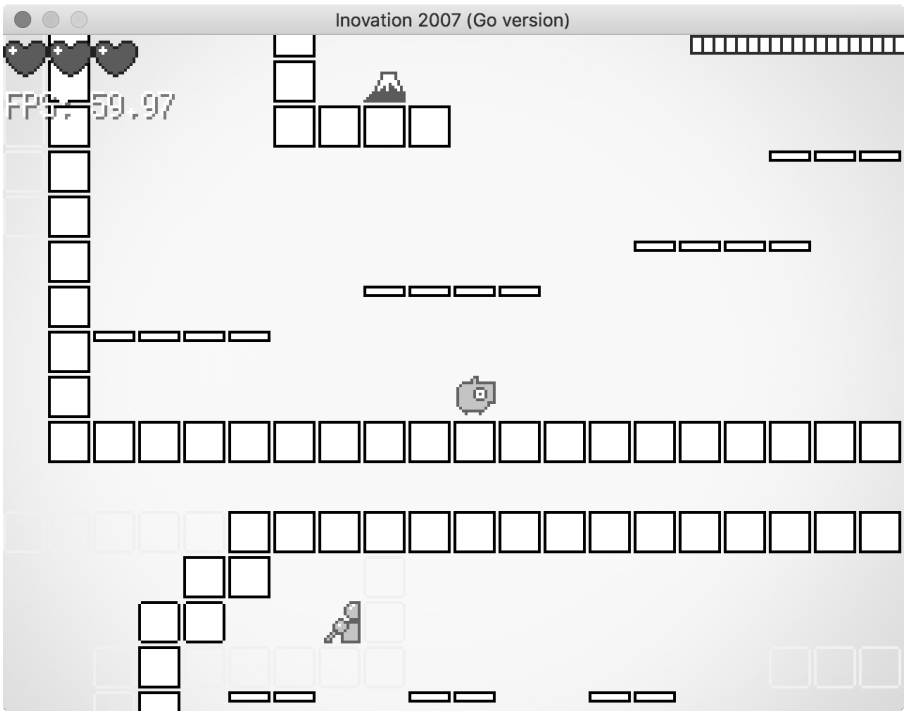


図 1.1 いの べーしょん 2007

「いの べーしょん 2007」は、猪を操作してアイテムを集めるアクションゲームです。無操作でも猪突猛進で進んでしまう猪を操作して、なるべく多くのアイテムを集めます。

オリジナルはおめが氏作のアクションゲームです (http://o-mega.sakura.ne.jp/product/ino.html)。元々 D というプログラミング言語で書かれていて、それが有志の手によって HTML5 に移植されました。著者の手で、さらにそれを Go で移植したのですが、その際に Ebiten を用いました。

ビルドすれば Windows、macOS、Linux で動きます。また GopherJS で Chrome、Firefox、Safari でも動きます。さらに Android や iOS 版もあります。それぞれアプリストア[1]にて公開しています。

[1] Android 版 は https://play.google.com/store/apps/details?id=com.hajimehoshi.

1.2.2　Wired Logic

https://github.com/martinkirsche/wired-logic

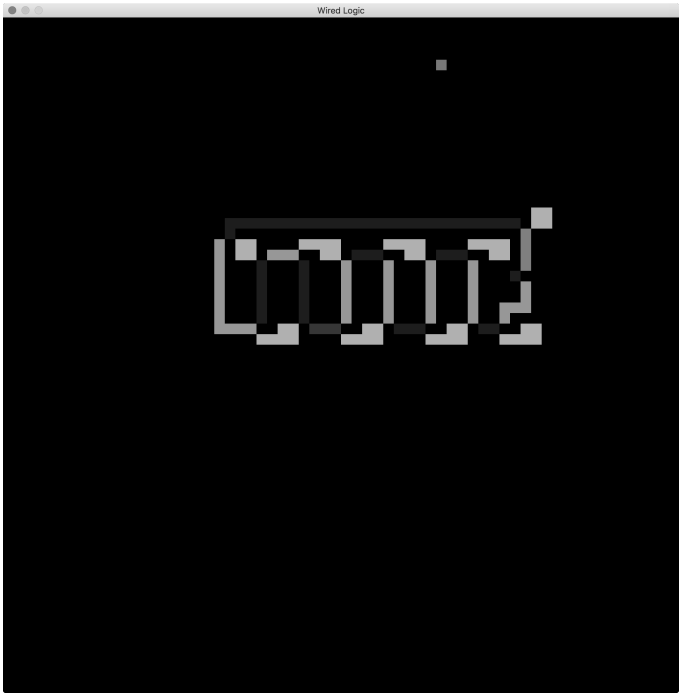


図 1.2　Wired Logic

Wired Logic は、Martin Kirsche 氏による回路シミュレータです。ブラウザ版もあります (http://www.martinkirsche.de/wired-logic/)。

1.2.3　Switches

https://github.com/hajimehoshi/switches

Switches は、筆者である星一作の迷路ゲームです。スイッチを切り替えてゴールを目指します。ブラウザ版もあります (https://hajimehoshi.github.io/switches/)。

1.2.4　Awakengine

https://github.com/DrJosh9000/awakengne/

---

goinovation、iOS 版は https://itunes.apple.com/us/app/id1132624266 で入手可能

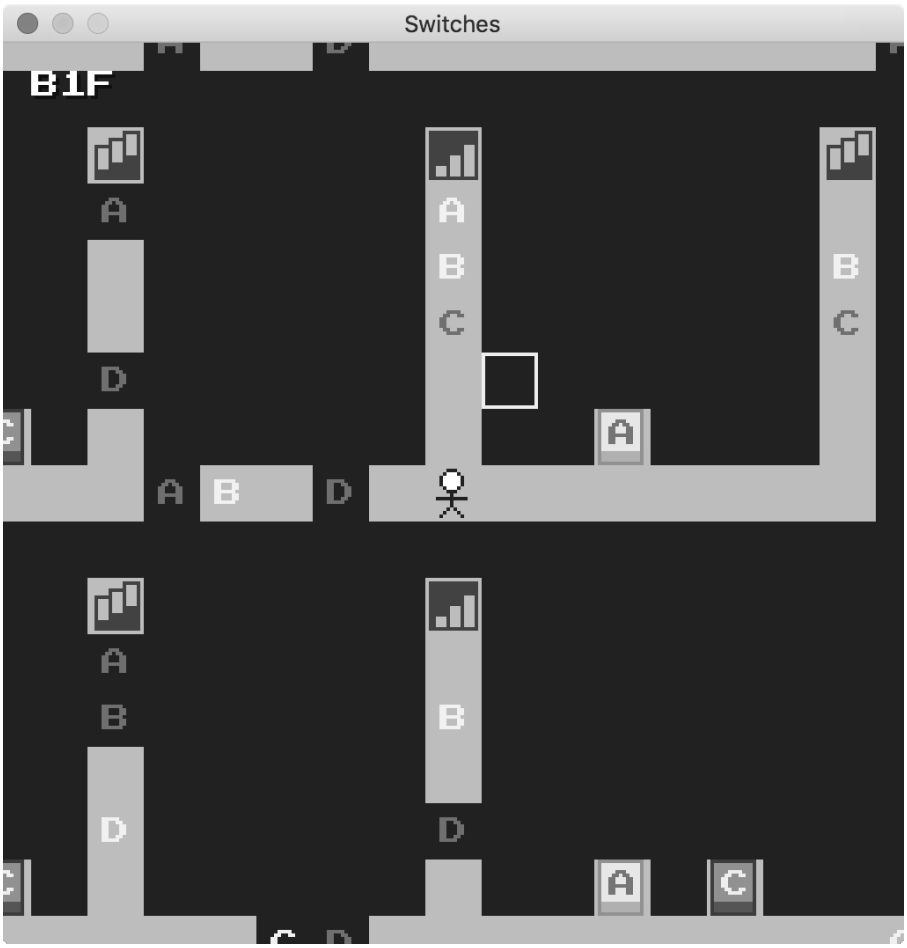


図 1.3 Switches

Awakeneinge は Josh Deprez 氏作のアドベンチャーゲームエンジンです。Awakengine で作られたゲームは、Web ブラウザ上に出力できることを利用して、日記形式で公開されています。日記としては https://awakeman.com/27/ および https://awakeman.com/40/ にて公開されています。

その他 Ebiten を使ったアプリケーションについては、Ebiten GitHub プロジェクトの Wiki にある、Works のページ (https://github.com/hajimehoshi/ebiten/wiki/Works) をご参照ください。

です。

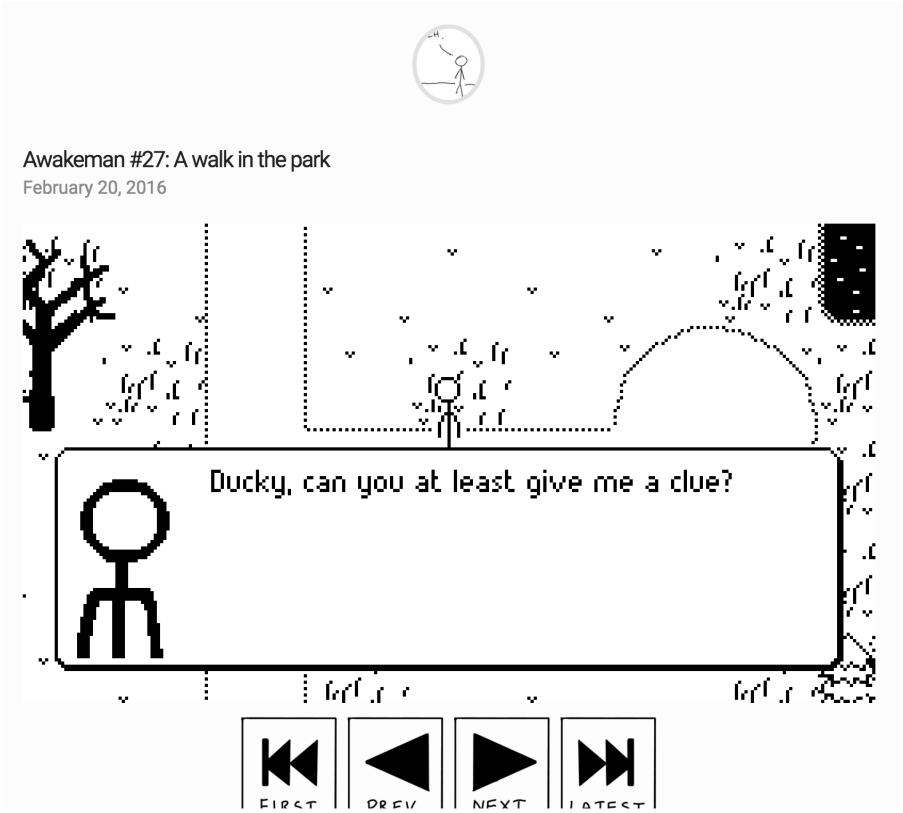


図 1.4　Awakeman #27

1.3 パッケージ

Ebiten を構成するパッケージについて紹介します。

1.3.1 github.com/hajimehoshi/ebiten パッケージ

Ebiten の中核を成すパッケージです。メインループ、描画機能、入力機能を提供します。本書は、主にこのパッケージの使い方について説明します。

1.3.2 github.com/hajimehoshi/ebiten/audio パッケージ

オーディオ関連の機能を提供するパッケージです。なぜこれが ebiten パッケージとは別パッケージになっているかというと、後から Ebiten にオーディオ機能が入ったという歴史的経緯もあるのですが、ebiten パッケージには依存せずに独立して使用することができるからです。本書ではこのパッケージについての紹介はしません。

1.3.3 github.com/hajimehoshi/ebiten/audio/vorbis パッケージ

Ogg/Vorbis デコーダを提供します。ブラウザでは、本来 Ogg/Vorbis に対応していない Safari でも使用することが出来ます。

1.3.4 github.com/hajimehoshi/ebiten/audio/wav パッケージ

WAV(RIFF 形式) のデコーダを提供します。

1.3.5 github.com/hajimehoshi/ebiten/ebitenutil パッケージ

デバッグ出力などのユーティリティを提供します。

1.4　Ebiten の背景

Ebiten を作った動機は、筆者の「スーファミっぽいレトロなゲームを作りたい」という考えからです。元々拙作の Star Ruby (`http://www.starruby.info/`) というゲームライブラリが前身としてありました。Star Ruby はプログラミング言語 Ruby のライブラリで、RPG ツクール XP というゲームエディタに搭載されていた RGSS という Ruby の拡張を参考に作ったものです[2]。Star Ruby をプログラミング言語 Go 用に直し、API と実装を洗練させたものが Ebiten になります。「スーファミっぽいレトロなゲーム」というのは作者の完全な趣味です。よくあるゲームライブラリとして「何でもできる」代わりに複雑な API セットを持つものはよくあります。Ebiten は逆に非常にシンプルなライブラリであり、出来ることは意図的に絞ってあります。その代わり非常にミニマムな API セットを定義することが出来ました。「スーファミっぽい」というのは API の判断基準になっていて、スーファミの機能、特にグラフィックスを実現するための必要最小限の API になっています。プログラミング言語 Go については、非常にシンプルな言語仕様ながら、ネイティブにコンパイルできて高速に動くバイナリを作れる点が非常に気に入りました。

前身のライブラリから考えると、Ebiten は最初の構想からすでに 10 年以上経過しています。では筆者は実際にゲームを作ったのかというというと、あまり作ったことはなくて (笑)、ゲームを作るよりもゲームライブラリを作るほうに時間を費やしてしまいました。とはいえ、2017 年 4 月現在で、ありがたいことに、簡単なゲームをいくつか作るのに Ebiten が使われています。また、筆者自身も知り合いと Ebiten を使ってスマホ向け RPG を製作中です。

[2] RPG ツクール XP は当時の株式会社エンターブレイン (現・株式会社 KADOKAWA エンターブレイン) から発売されたソフトウェアですが、実は Ebiten (海老天) という名前はエンターブレインへのリスペクトを込めてつけました。エンターブレインのネット上での愛称が「海老」だったからです☺

1.5　Ebiten のポリシー

Ebiten のポリシーは「スーファミっぽいゲームという要件を満たすための必要最小限の API を定義する」です。API をミニマムにするというのは、他のゲームライブラリあまりみられない点であると考えています。Ebiten は「どんなゲームでも実現できるライブラリ」ではありません。あくまで「スーファミっぽいゲームを作れる」ことを主眼においており、API 設計時における取捨選択はこの目的に沿っているかどうかで判断されます。

API が必要最小限であるということは、実現したいゲームの機能を実装するためにはある程度冗長に書かなければならないということでもあります。よくゲームライブラリの宣伝文句として「○○の機能が数行で実現できる」というのを見ますが、Ebiten はサンプルを見ていただければ分かる通り、数行で実現できることはほぼありません。

具体例を見てみます。単純に画像を回転させるのにもメインロジック部分で 10 行程度は要します。回転するためには画像を幾何行列で指定して描画するのですが、中心を定めて回転するためには、移動と回転の行列の掛け算を複数回行わなければならないからです。

```
func update(screen *ebiten.Image) error {
    count++
    w, h := gophersImage.Size()
    op := &ebiten.DrawImageOptions{}
    // 画像の中心で画像を回転させたい。まず画像の中心を原点（左上）に持ってくる。
    op.GeoM.Translate(-float64(w)/2, -float64(h)/2)
    // 回転する。
    op.GeoM.Rotate(float64(count%360) * 2 * math.Pi / 360)
    // 画面中心に再移動させる。
    op.GeoM.Translate(screenWidth/2, screenHeight/2)
    // 描画する。
    screen.DrawImage(gophersImage, op)
    return nil
}
```

「行列なんぞ使わないで、画像の中心と角度を指定して回転させる API を作る」というのも考えられます。それはそれで 1 つの解ですが、この場合例えば拡大と組み合わせるとどうなるかとか、任意の行列もオプションで指定したくなった場合どっちを優先するかなど、考慮しなければならない別のことが増えてしまいます。Ebiten ではあらゆる

幾何変形についてアフィン行列のみの指定にすることによって、矛盾がなく挙動が明確な API になっています。

Ebiten にはコンパクトさにかけるという意味でのエレガントさはないのですが、逆に言うとゲームのコードが非常に明確になるということでもあります。Ebiten ではエレガントに書けるかどうかは重要視していないのです。また最小限の API といっても、API は十分汎用であり、この最小限の API をうまく組み合わせれば、やりたいことが概ね実現できると考えています。

1.6 コミュニティ

1.6.1 GitHub

Ebiten のリポジトリは GitHub の https://github.com/hajimehoshi/ebiten ですが、そこの Issues で問題報告や質問を受け付けています。言語は英語です。

1.6.2 Slack

Gophers Slack (https://blog.gopheracademy.com/gophers-slack-community/) の#ebiten に筆者は常駐しています。Invite form (https://invite.slack.golangbridge.org/) から参加できます。言語は英語です。

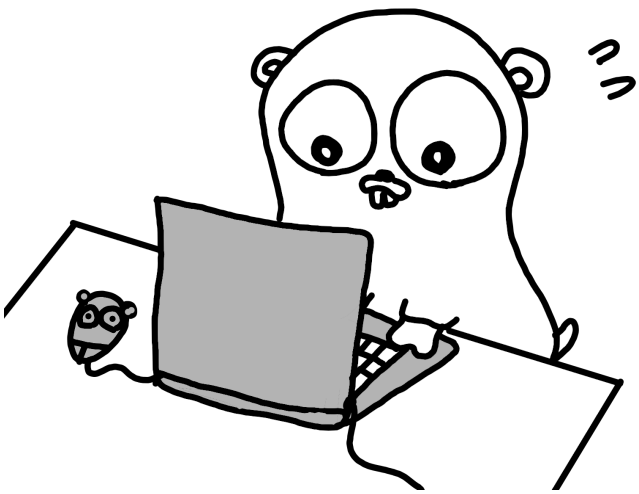
1.6.3 その他

筆者のメール hajimehoshi@gmail.com に質問いただければ、出来る範囲で個別に応対いたします。

第 2 章

インストール

本章では Ebiten のインストール方法を各プラットフォームごとに説明します。Go のライブラリは基本的に `go get` コマンド一発でインストールできるのですが、Ebiten はコンパイルに C コンパイラや他のライブラリも要求するため、準備を整えてから `go get` する必要があります。



2.1　事前準備

Ebiten をインストールするまでの事前準備は次の通りです。

1. Go をインストールする。
2. Git をインストールする。
3. C コンパイラをインストールする。
4. 依存ライブラリをインストールする (Linux のみ)。

2.1.1　Go

Go のインストール手順については公式ドキュメント (https://golang.org/doc/install) が詳しいです。ここではその概略を説明します。

まず Go をインストールします。https://golang.org/dl/ から最新安定版を落としてインストールしてください。

インストール後には PATH 環境変数の設定が必要です。デフォルトだと Go は以下のパスに入ります。

Windows　C:\Go[1]
macOS　/usr/local/go
Linux　/usr/local/go

Go のバイナリはそのパスの bin ディレクトリにはいっています。Windows ならば C:\Go\bin を、macOS および Linux ならば/usr/local/go/bin を、PATH 環境変数に追加してください。

コマンドプロンプトまたはターミナルで go version と打ち、Go のバージョンが表示されればインストール成功です。

次に、Go のワークスペース (https://golang.org/doc/code.html#Workspaces) のためのディレクトリを作ります。ワークスペースは、Go のパッケージのソースコードやコンパイル済みのバイナリを置いておく場所です。

Go のワークスペースは GOPATH という環境変数で指定します。Go1.7 以前の場合はこ

[1] 環境によって\は¥のように表示されているかもしれません。とりあえずこの 2 つの記号は同じ意味だと思ってもらって良いです。

の環境変数を必ず設定する必要がありました。Go1.8 以降の場合、GOPATH 環境変数が設定されていない場合はデフォルトの値を使います。デフォルト値は以下のとおりです。

Windows %USERPROFILE%\go
macOS $HOME/go
Linux $HOME/go

Windows における %USERPROFILE% や macOS/Linux における $HOME はそれぞれ環境変数の展開を表します。Windows であれば %USERPROFILE% は C:\Users\(ユーザー名) になっているはずです[2]。上記のディレクトリを作っておきましょう。

余談ですが、多くの Go のツールは、プログラムがこのワークスペースの中に入っていることを想定しています。相対パスのパッケージや main パッケージではない、絶対パスで表されるパッケージのソースファイルは、ワークスペースの決まった場所に置くことが想定されています。ある程度まとまったプロジェクトならば、ワークスペースの定位置内で作業してしまったほうが無難です。テンポラリに go run だけで実行される Go プログラムならば気にする必要はありません。

2.1.2 Git

続いて Git をインストールします。Git はバージョン管理システムですが、Go の場合パッケージを取得するのに使われます。

Windows

Git の公式サイトから Windows 版インストーラー (https://git-scm.com/download/win) を落とし、インストールしてください。また、次の値を PATH 環境変数に追加する必要があります。

32bit 環境の場合 C:\Program Files\Git\bin
64bit 環境の場合 C:\Program Files (x86)\Git\bin

git version と入力してバージョンが出れば成功です。

[2] コマンドプロンプトで echo %USERPROFILE% とすることで実際の内容を確認できます。

macOS

最新の macOS ならば Git はすでにインストール済みなので、新たな作業は不要です。

Linux

インストール方法は環境によって異なります。Ubuntu の場合、`apt-get` コマンドで入手できます[3]。

```
apt-get install git
```

2.1.3　C コンパイラ

Ebiten をコンパイルするためには、Go だけでなく C のコンパイラも必要です。Ebiten の依存ライブラリの一部が C で書かれているからです。これは `go get` コマンドや `go build` コマンドなどによって間接的に使用されます。実際に Ebiten を使用する際には C コンパイラが適切にインストールさえされていればよくて、ゲーム開発側が意識する必要はありません。

Windows

Windows の Go で使える C コンパイラは GCC です。インストール方法には複数方法がありますが、TDM GCC (`http://tdm-gcc.tdragon.net/`) がおすすめです。インストーラーをダウンロードしてインストールしてください。また環境変数の設定も自動でやってくれます。

`gcc -version` と入力してバージョンが出れば成功です。

macOS

C コンパイラである clang を入れる必要があります。最新の macOS ならば、ターミナルで `clang` と入力すると、clang がない場合にはインストールを促すダイアログが表示されるはずです。あとは指示に従ってインストールしてください。

---

[3] 大抵の場合、`apt-get install` にはルート権限が必要になります。その場合には `apt-get` コマンドを `sudo` コマンドで実行してください。

Linux

インストール方法は環境によって異なります。Ubuntu の場合、apt-get コマンドで入手できます。

```
apt-get install build-essential
```

2.1.4 依存ライブラリ

Linux では、Ebiten コンパイルのために依存ライブラリが必要です。Ubuntu の場合、以下の依存ライブラリを apt-get などでインストールしてください。

- libglu1-mesa-dev
- libgles2-mesa-dev
- libxrandr-dev
- libxcursor-dev
- libxinerama-dev
- libxi-dev
- libopenal-dev

2.2　Ebiten のインストール

コマンドプロンプトまたはターミナル上で go get して、Ebiten をインストールしてください。

```
go get github.com/hajimehoshi/ebiten/...
```

... は github.com/hajimehoshi/ebiten だけでなく、それ以下のパッケージ名のライブラリ、例えば github.com/hajimehoshi/ebiten/audio なども、すべてインストールすることを表します。

Web ブラウザ向けゲームを作りたい場合は、GopherJS もインストールします。GopherJS は Go を JavaScript に変換するツールです。Ebiten のブラウザ向けの機能はこれを使用しています。

```
go get github.com/gopherjs/gopherjs
go get github.com/gopherjs/webgl
```

2.3　サンプルの実行

インストール出来たら、サンプルを実行してみましょう。次のコマンドを実行してください。

Windows の場合

```
cd %GOPATH%\src\github.com\hajimehoshi\ebiten\examples\
go run -tags=example blocks/main.go
```

macOS、Linux の場合

```
cd $GOPATH/src/github.com/hajimehoshi/ebiten/examples/
go run -tags=example blocks/main.go
```

どちらもやっていることは同じで、ワークスペースの中にある github.com/hajimehoshi/ebiten/examples のソースコードがあるディレクトリに移動して、go run で Go プログラムを実行しているだけです。

よくある落ち物ゲームがプレイできたら起動成功です。おめでとうございます!

リソースファイルの位置の関係上、サンプルを実行できるディレクトリは決まっているので、そこまで移動する必要があります[4]。またビルドタグ example が必要です。

余談ですが、なぜビルドタグが必要なのかというと、go get で github.com/hajimehoshi/ebiten/... を指定したときに、examples まで一緒にインストールされてほしくないからです。go get する際、main パッケージのものは$GOPATH/bin にインストールされますが、... を指定することでそれ以下のディレクトリにある全ての main パッケージのものが対象に入ってしまいます。ビルドタグをつけることで、明示的に指定しない限りは go get でインストールされません。

[4] どこからでも実行可能なようにサンプルを修正する方法は、ないこともないのですが、サンプルのソースコードが複雑になってしまうので、あえてこのままにしています。

2.3.1　サンプルのブラウザでの実行

サンプルをブラウザで動かすことが出来ます。ただしタグの関係上、gopherjs serve[5]で動かすことが出来ません。そのため、Ebiten サンプルを動かすためのプログラムを用意してあります。examples ディレクトリ内で次のコマンドを実行してください。

```
go run _server/main.go
```

http://localhost:8000/?blocks にアクセスして、ゲームが実行できれば成功です。?以降の文字列を変えると、実行するサンプルを変えることが出来ます。

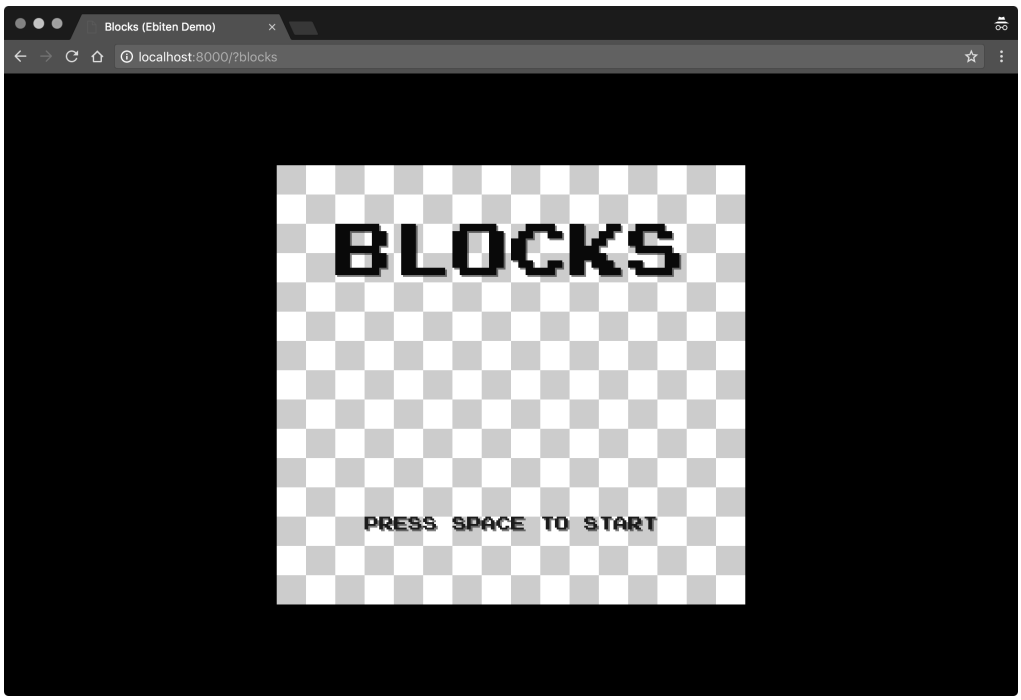


図 2.1　サンプルのブラウザでの実行例

[5] GopherJS の機能の一つ。ワークスペースの中の Go プログラムを自動的に JavaScript に変換してくれる HTTP サーバーを提供してくれます。localhost の特定のポート (デフォルトでは 8080) にアクセスするだけで、ワークスペースの中にある Go パッケージをそのままブラウザ上で実行できます。

2.4 Android/iOS

Ebiten は Android および iOS 向けのゲームを作ることが出来ます。モバイルにおける Ebiten の利用は、デスクトップのそれに比べると複雑です。ここでは概略だけ説明します。詳細については Wiki (`https://github.com/hajimehoshi/ebiten/wiki/Mobile`) を参照してください。

Go プログラムをモバイル向けにコンパイルするには、gomobile (`https://godoc.org/golang.org/x/mobile`) というツールを使います。gomobile は主に 2 つの機能があります。

`gomobile build` 単体アプリケーションを出力する。Android の場合は apk ファイル、iOS の場合は ipa ファイルになる。Go だけですべてのアプリケーションが書けるが、アプリケーションは OpenGL サーフェス 1 個で構成される物になる。

`gomobile bind` 共有ライブラリを出力する。Android の場合は aar ファイル、iOS の場合は framework ファイルになる。あくまで共有ライブラリなので、Java/Objective-C から呼び出す形で使用する。関数の相互呼び出しが可能。

さて Ebiten はこの 2 つのうち `gomobile bind` の共有ライブラリ出力しかサポートしていません。これはなぜかというと、`gomobile build` で出力されるアプリは制約が多く、ストアで公開できる実用的なアプリにすることが非常に難しいからです。たとえば OpenGL サーフェス 1 個しかなく広告などの追加が不可能であること、ソフトウェアキーボードを通じた文字入力ができないこと、署名を入れることが困難なことが挙げられます。一方 `gomobile bind` の出力は共有ライブラリであり、アプリとして利用するためには Java/Objective-C のグルーコードが必要になってしまいます。しかし `gomobile build` にあるような制約は一切なく、なんでも自由に出来る柔軟性を持ちます。

具体的に、モバイル向けパッケージを作るにはどうしたらよいか、どのような Java/Objective-C のグルーコードが必要になるか等については、前述の Wiki を参照してください。

第 3 章

ゲームループ

この章ではゲームの基本構造であるゲームループについて解説します。

ゲームは、一定時間のインターバルで少しずつ状態を更新しながら進みます。またそのときの状態を元に描画を行います。更新は非常に短い時間間隔で行われます。Ebiten においては 1/60 秒です。パラパラ漫画のように、短い時間感覚で少しずつ状態を変えていくことで、時間軸方向の変化を表現します。

Ebiten のゲームループのインターフェイスは非常に単純です。1 秒間に 60 回呼ばれる 1 個のコールバック関数を Run という関数に渡すだけで、あとは Ebiten がメインループを実行します。

3.1　最初の Ebiten プログラム

最初の Ebiten プログラムというとで、何も実行しないゲームを起動してみましょう。

```
// blank.go

package main

import (
    "log"

    "github.com/hajimehoshi/ebiten"
)

func update(screen *ebiten.Image) error {
    // ebiten.Run 関数の呼び出しにこの関数が渡される。
    // ebiten.Run 関数呼出し後、この関数は定期的に (1 秒間に 60 回) 呼ばれる。
    return nil
}

func main() {
    // ゲームのエントリーポイント。
    // Run は一度呼ばれると基本的には制御が戻ってこない。
    if err := ebiten.Run(update, 320, 240, 2, "Test"); err != nil {
        log.Fatal(err)
    }
}
```

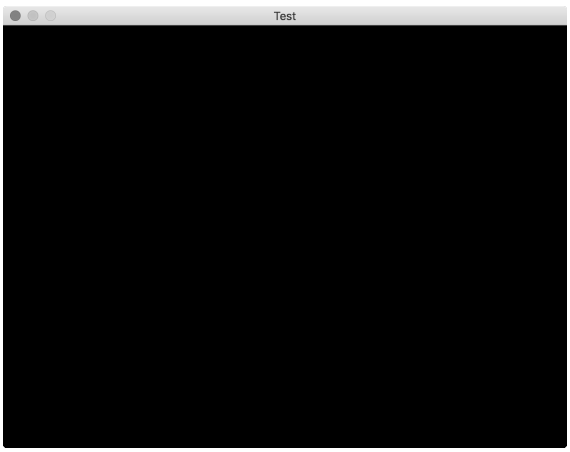


図 3.1　blank.go の実行結果

go run で実行して、真っ黒なウィンドウが表示されれば成功です。

```
func Run(f func(*Image) error,
    width, height int, scale float64, title string) error
```

Run 関数でゲームを開始し、与えられた関数 f を定期的に実行します。画面サイズは width、height、scale で指定されます。scale は単純に画面を拡大 (または縮小) するための数字です。scale には整数だけでなく小数も指定でき、その際もドット絵感が保たれたまま拡大を行います[1]。title はタイトルで、ウィンドウ名等に使われます。

f は引数として*ebiten.Image を取ります。これは画面を表す Image オブジェクトです。これに対して操作を行うことで描画を実現します。また、f の戻り値として error 型の値を返します。これが nil でない値の場合は、ゲームを終了し、Run 関数の戻り値となります。

Run 関数は一度実行すると基本的に制御が帰ってきません。ただし、以下の状況のときには制御を返して終了します。

- ウィンドウを閉じた (戻り値は nil になる)
- OpenGL エラーが発生した
- 関数 f が nil でないエラー値を返した

上記通り意図的にエラーを返すようにして終了させることも可能ですが、これを行う場合は、ゲームがデスクトップで動いていると分かっている場合に限定したほうが無難です。ブラウザやモバイルにはアプリケーションの終了という概念がなく、実際 Ebiten はそれを想定していません。終了後の処理が正しく行えるのはデスクトップ環境だけです。Run は基本的に「帰ってこないもの」として扱うのが無難です。

[1] 小数が指定された場合は次のような手順をとります。画面を「指定されたスケールに最も近い、その値以上である整数値」倍でニアレストフィルタで一旦拡大し、その結果を指定されたスケールに合わせてリニアフィルタで縮小します。

3.2 FPS

Run 関数に与えられた関数は 1 秒間に 60 回呼ばれます。いわゆる 60FPS (Frames Per Second) というわけです。Frame はゲームにおける時間の最小単位です。Ebiten の 60FPS というのは、どの OS でも、どのリフレッシュレートのディスプレイを使っても固定です。Ebiten においては、環境について気にすることなく、60FPS 前提でゲームを組んで問題ないのです。

これは f が 60 秒間呼ばれることが保証されるという意味であって、実際の描画は環境によっては 60FPS を下回る可能性があります。f は論理的なゲームの状態の更新と描画の更新両方を担います。論理的な状態の更新は常に 60FPS を仮定してよいのですが、描画結果は採用されずにスキップされる可能性があります。その結果描画がガクガクになるかもしれません。描画更新は論理更新よりも一般的に重いため、描画が完了するまでに 1/60 秒以上かかってしまう場合に、この現象が発生します。現在の処理が重く、描画スキップされることが予め分かっていれば、f 中で描画処理を行う必要はないわけです。このような状況を判別する関数として ebiten.IsRunningSlowly があります。この関数が true を返す場合は f 中で描画処理を行う必要がありません。

現在の「描画」の FPS については ebiten.CurrentFPS 関数で取得することが出来ます。

なお 60 という数字は ebiten.FPS という定数としても定義されています。

ebiten.IsRunningSlowly を考慮したプログラムの大枠は次のようになります。

```
// isrunningslowly.go

package main

import (
    "log"

    "github.com/hajimehoshi/ebiten"
)

func update(screen *ebiten.Image) error {
    // 状態更新処理
    if ebiten.IsRunningSlowly() {
        // IsRunningSlowly が true のときは、screen に対する
```

```
        // 描画結果は実際の画面に反映されない。
        // よって描画処理を行わず終了してよい。
        return nil
    }
    // 描画処理
    return nil
}

func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Test"); err != nil {
        log.Fatal(err)
    }
}
```

3.3　Hello, World!

真っ黒な画面では面白くないので、何か書いてみましょう。あくまでデバッグ用途ですが、文字を書くための関数があります。

```
// helloworld.go

// test

package main

import (
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    ebitenutil.DebugPrint(screen, "Hello, World!")
    return nil
}

func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Hello"); err != nil {
        log.Fatal(err)
    }
}
```

ebitenutil というパッケージを import する必要があります。ebitenutil は補助的なユーティリティ関数等が定義されているパッケージです。今回はその中の DebugPrint 関数を利用します。

```
func DebugPrint(image *ebiten.Image, str string) error
```

DebugPrint 関数は与えられた Image オブジェクト image に文字列 str を描画しま

図 3.2 helloworld.go の実行結果

す。文字列は日本語などは使えず、アルファベットである必要があります[2]。あくまでデバッグ用途なので大した機能はありませんが、ちょっとした文字列をとりあえず表示したい場合には便利です。

DebugPrint は error 値を返すことになっていますが、この値は常に nil です[3]。よって無視して構いません。

[2] 厳密には U+0020 から U+00FF (ÿ) までの範囲の (制御文字ではない) 文字が使え、それ以外の文字は単に無視されます。

[3] バージョン 1.5.0-alpha から、Ebiten の描画関数はすべてエラーを返さないことになりました。後方互換性のためにシグニチャはそのままにしています。

3.4　ブラウザでの実行

作ったゲームを gopherjs コマンドを利用します。たとえば作った Go ファイルが helloworld.go だとすると、JavaScript ファイルを出力するコマンドは次のようになります。

```
gopherjs build -o helloworld.js helloworld.go
```

出力は helloworld.js ファイルです。実際には helloworld.js ファイルだけでなく helloworld.js.map ファイルも出力されます。

このように作ったゲームを実行するには、次のような HTML を用意します。

```
<!DOCTYPE html>
<script src="helloworld.js"></script>
```

実際にこのファイルをブラウザで開いて、実行されていれば成功です。

ゲームがワークスペース[4]の中に正しく置かれているのであれば、gopherjs serve コマンドでも実行することが出来、しかもリロードの際に Go ファイルの更新に追従して、最新のものを実行してくれます。

注意点として、GopherJS の制約に気をつける必要があります。たとえばファイルにアクセスする os.Open などはブラウザ上では動きません[5]。ファイルにアクセスする処理は JavaScript 用に別途作る必要があります。なお、デスクトップとブラウザで両方使えるユーティリティ関数として、ebitenutil の NewImageFromFile 関数や OpenFile 関数があります。

なおこの HTML ファイルは、一個の script 要素以外に追加で書くことは、基本的に想定していません。何かのページに Ebiten のゲームを組み込みたい場合は、この HTML を iframe などで埋め込むと良いでしょう。

[4] 環境変数 GOPATH で指定される場所。ないしは、 (Go1.8 以降の場合) ホームディレクトリの中の go ディレクトリ。ソースやコンパイル結果が置かれる場所。

[5] os.Open に関しては Node.js では実行できます。

3.5 更新

文字が描画されたのはいいのですが、これでは本当にゲーム更新関数が定期的に呼ばれているのかいまいちわかりません。実際に更新関数が呼ばれていることは println 関数などを使ってコンソール上に文字を表示しても確認できます。しかしそれでは面白くないので、ゲーム画面で何か時間経過が分かるものを作ってみましょう。

```
// increment.go

package main

import (
    "fmt"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

// カウンター
var i = 0

func update(screen *ebiten.Image) error {
    // カウンターをインクリメントする。
    i++
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // カウンターを表示する。
    msg := fmt.Sprintf("%d", i)
    ebitenutil.DebugPrint(screen, msg)
    return nil
}

func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Increment"); err != nil {
        log.Fatal(err)
    }
}
```

整数変数 i を更新ごとに 1 加算しています。

実行すると数字が描画されますが、ものすごい勢いで数字が増えていくことがわかり

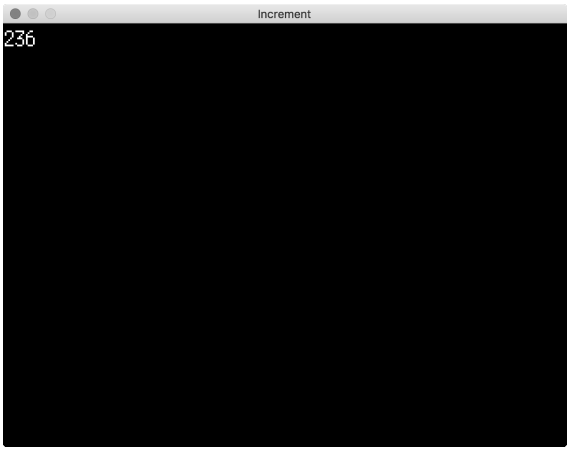


図 3.3　icrements.go の実行結果

ます。60FPS ですので、実際に 1 秒間に 60 増える計算になります。

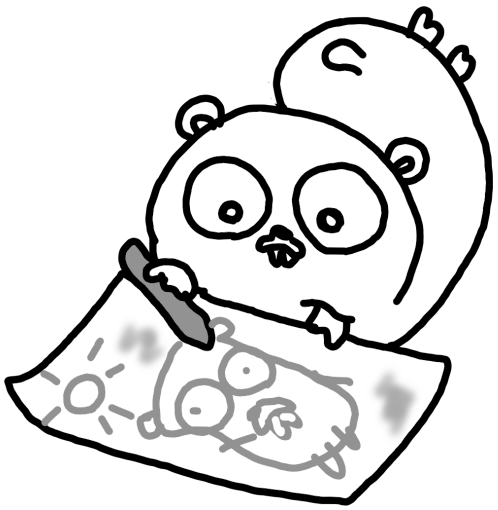
　また勘が鋭い方はお気づきかもしれませんが、update 関数に渡される screen は呼ばれる前に毎度クリアされています。クリアされていないとすると、同じ箇所に文字を書き続けた場合に文字が重なって表示されるはずですが、実際にはそうなっていません。

第 4 章

描画・基礎

この章では Ebiten の描画機能について解説します。

Ebiten における描画の基本単位は `ebiten.Image` オブジェクトです。この章では、このオブジェクトの使い方の基礎を紹介します。また、`ebiten.Image` オブジェクトを使いこなすための、「行列」と呼ばれる数学について、最小限の基礎知識を紹介します。



4.1　ebiten.Image 構造体

ebiten.Image 構造体は、Ebiten における画像を表す構造体です。2 次元平面矩形のピクセルの集合を表します。色はプリマルチプライドアルファな RGBA です。

例えるならば、ebiten.Image オブジェクトは、四角い透明フィルムのようなものです。Ebiten ではその透明フィルムに対して描画命令を記述します。たとえばフィルムを単色で塗りつぶしたり、他のフィルムを貼り付けたりなどです[1]。

ebiten.Image 構造体は、標準ライブラリの image パッケージの Image インターフェイスを実装しています。そのため、ebiten.Image を標準ライブラリの関数に渡すことが出来ます。たとえば ebiten.Image オブジェクトを PNG エンコーダにそのまま渡すことが出来ます。

いままで更新関数に渡される引数も ebiten.Image オブジェクトでした。またこれから扱う画像ファイル (から作る、描画元となるデータ) も ebiten.Image オブジェクトとして扱われます。Ebiten における描画命令は基本的に、ebiten.Image 構造体の関数に統一されているのです。

[1] この例えは若干不正確です。フィルムにフィルムを貼り付けると、貼り付けられるフィルムが二度と再利用できないような感じがします。ebiten.Image オブジェクトにおいて描画元はそういった制限がなく、複数の描画先に対して同じ描画元を何度も使いまわすことが出来ます。

4.2　塗りつぶし

Ebiten の描画操作の中で最も簡単なものは単色の「塗りつぶし」です。画面を塗りつぶすには Image の Fill 関数を使います。

```
func (i *Image) Fill(clr color.Color) error
```

実際に赤色で画面を塗りつぶしてみましょう。Ebiten では画面は Image オブジェクトですので、それに対して Fill 関数を呼んであげればよいだけです。色の指定は image/color パッケージの Color を指定します。

```
// fill.go

package main

import (
    "image/color"
    "log"

    "github.com/hajimehoshi/ebiten"
)

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // 画面 (screen) を赤色で塗りつぶす。
    // color.RGBA は (premultiplied-alpha な) 8bit カラーを表す。
    screen.Fill(color.RGBA{0xff, 0, 0, 0xff})
    return nil
}

func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Fill"); err != nil {
        log.Fatal(err)
    }
}
```

Fill を呼び出すと、対象となる Image に元々描かれていたものはすべて消えてしまいます。

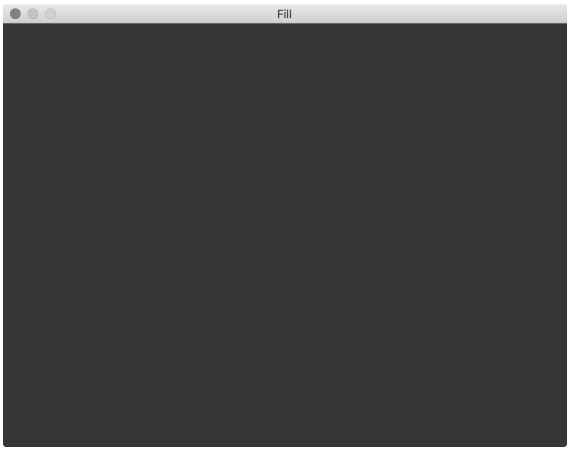


図 4.1　fill.go の実行結果 (カラー版は i ページ図 1)

`Fill` は `error` 値を返すことになっていますが、`ebitenutil.DebugPrint` と同様、この値は常に `nil` です。よって無視して構いません。

4.3 画像

いよいよ画像を描画していきます。画像の描画には DrawImage 関数を使います。

```
func (i *Image) DrawImage(image *Image, options *DrawImageOptions) error
```

Image である i を描画先として、別の Image である image を描画します。Ebiten では描画元も描画先もどちらもおなじ Image です。

今回は海老の天ぷらの絵を描画してみましょう。

図 4.2 ebiten.png (カラー版は i ページ図 2)

この画像のファイル名は ebiten.png で、ソースコードと同じディレクトリに置いてあるとします。この画像を表示するコードは次のとおりです。

```
// image.go

package main

import (
    // PNG ファイルを扱うため、デコーダを有効にする。
    // この、デコーダ登録のためだけにパッケージをインポートする手法については
    // 標準ライブラリの image パッケージを参照すること:
    // https://golang.org/pkg/image/
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
```

```
    // ebitenImage を画面に描画する。
    // 第 2 引数である DrawImageOptions は今回は指定を省略する (nil を指定する)。
    screen.DrawImage(ebitenImage, nil)
    return nil
}

func main() {
    var err error
    // ebitenutil パッケージの NewImageFromFile 関数を使って、
    // ファイル名から直接*ebiten.Image を作成する。
    // 2 つめの戻り値は*image.Image だが、今回は使わないので無視する。
    // フィルタとして今回は FilterNearest を使用した。
    // この値は画像の拡大縮小の際のフィルタを表すが、実際に画像を拡大縮小したり
    // しない限りは使用されないので、今回はとりあえずなんでも良い。
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Image"); err != nil {
        log.Fatal(err)
    }
}
```

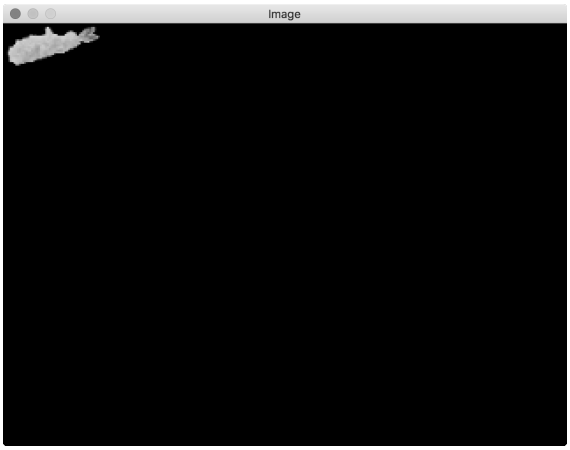


図 4.3　image.go の実行結果

画像ファイル ebiten.png から ebiten.Image オブジェクトを作り、それを画面上に描画します。今回は DrawImageOptions を省略したので、単に左上にそのまま描画されます。

DrawImage は常に nil を返しますので、返り値は無視してよいです。

画像ファイルから ebiten.Image オブジェクトを作るのに、今回は ebitenutil パッケージの NewImageFromeFile 関数を使いました。

```
func NewImageFromFile(path string, filter ebiten.Filter)
    (*ebiten.Image, image.Image, error)
```

余談ですが、なぜ NewImageFromFile 関数が ebiten パッケージではなく ebitenutil パッケージに入っているのかというと、ebiten パッケージはポータビリティのためにファイルシステムを仮定していないからです。実際 NewImageFromFile 関数はデスクトップとブラウザ[2]では動くのですが、モバイルでは動きません。画像などのリソースはファイルを使うより go-bindata (https://github.com/jteeuwen/go-bindata) などを使って埋め込むことを推奨しています。なお、ファイルを指定せずに、標準ライブラリの image.Image インターフェイスから ebiten.Image オブジェクトを作る関数として、ebiten.NewImageFromImage があります。

```
func NewImageFromImage(source image.Image, filter Filter) (*Image, error)
```

さて、画面に画像を描画できることはできましたが、このままでは指定した画像を左上に描画するだけです。これでは何もゲームになりません。画像を指定した位置に描画したり、回転させたりするにはどうしたらよいでしょうか。そのためには、Ebiten の座標系の理解と「行列」と呼ばれる数学の知識が必要になります。

[2] モバイルでは XmlHttpRequest を使った実装になっていて、パスは絶対 URL または相対 URL として解釈されます。

4.4　座標系

Ebiten は 2D グラフィックスを取り扱いますが、そのための座標系が定められています。ebiten パッケージの Image オブジェクト上に何かを描画する場合、その描画先となる Image オブジェクト上に座標系が定義されます。X 軸方向が右、Y 軸方向は下です。原点は左上です。

図 4.4　Ebiten における座標系。Y 軸が下向きであることに注意

座標系は ebiten.Image オブジェクトごとに存在します。ある ebiten.Image に何かを描画しようとした場合、その描画先の左上が原点になります。

前節で何のオプションも指定せずに画像を描画しましたが、その場合は原点の位置に描画されます。

4.5 行列

ebiten.Image は 2 次元平面矩形のピクセルの集合であると説明しました。Ebiten ではそのピクセルの一点一点について座標変換ルールを適用することで、単なる座標指定はもちろん、拡大縮小、回転など様々な描画方法を実現します[3]。その座標変換ルールとして、Ebiten では「行列」を用います。

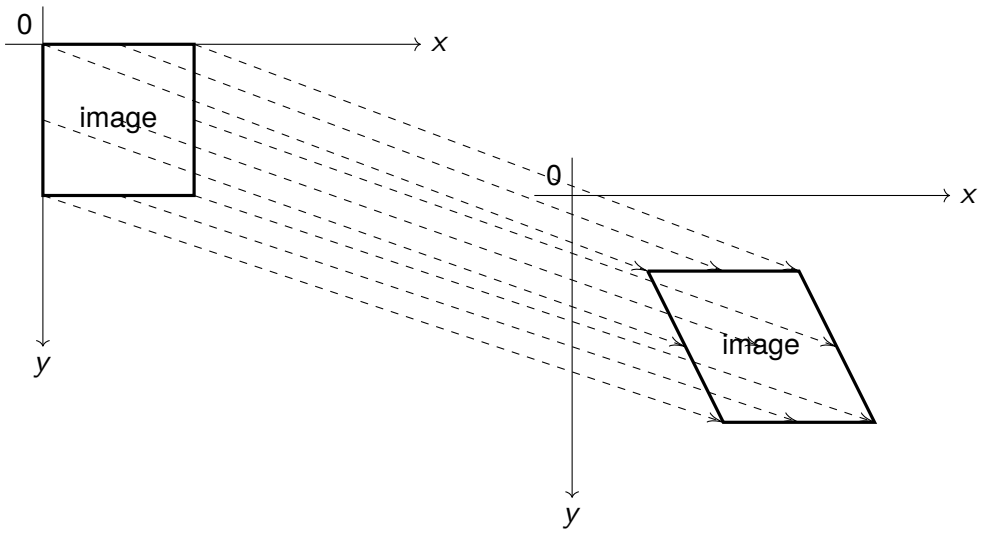


図 4.5 行列による幾何変換のイメージ。画像の一点一点について「行列」による変換を適用し、それぞれ違う座標に変換する。結果的に画像全体が移動・変形する

2 次元空間における点座標は 2 次ベクトル (x, y) で表されます[4]。これを数学的に変換して新しい 2 次ベクトルにするのが、(2 次) 行列というわけです。

また Ebiten では、座標の位置を変換する幾何変換以外に、色変換にも行列は用いられます。詳細は後述しますが、色を RGBA の 4 次元空間の一点とみなし、行列によって変換するのです。

---

[3] 勘の良い方はお気づきかもしれませんが、ピクセル一点一点について変換を施して単純に「拡大」すると、拡大結果はスカスカになってしまいます。また、縮小すると一部のピクセルの情報がなくなってしまいます。変な描画結果にならないようにピクセルを補完する必要があるのですが、どう補完するかについては先述の NewImageFromFile などが取るフィルタの値 (ebiten.FilteNearest など) によって決められます。

[4] ベクトルというと向きとか矢印のイメージがありますが、ここでは 2 つ組の数字以上の意味はありません。

ここでは、厳密な数学の議論ではなく、Ebiten を使うための最低限の知識を紹介します。

4.5.1　行列の定義

行列は数を矩形状に配列したものです。数学の線形代数という分野で使われます。

大きさが 2 の行列は次のようなものです。

$$\begin{pmatrix} 0.5000 & -0.8660 \\ 0.8660 & 0.5000 \end{pmatrix} \tag{4.1}$$

横方向を行、縦方向を列と呼びます。

行の要素数と列の要素数が同じ行列を正方行列と呼びます。Ebiten では正方行列しか扱いません。

大きさが 2 の正方行列は 2 次行列と呼ばれます。大きさが 3 の 3 次行列は次のようになります。

$$\begin{pmatrix} 0.2990 & 0.5870 & 0.1140 \\ -0.1687 & -0.3313 & 0.5000 \\ 0.5000 & -0.4187 & -0.0813 \end{pmatrix} \tag{4.2}$$

4.5.2　行列とベクトルの乗算

先程の行列とベクトル (x, y) との間で乗算を行うことができます。

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix} \tag{4.3}$$

行列が左側に、ベクトルが右側に来ています。逆にベクトルを行列の左から掛けることもありますが、Ebiten では扱いません。

行列はいわゆる「変換ルール」ですが、上の式の (x, y) は変換前の点の座標、乗算はルールの適用、$(ax + by, cx + dy)$ は変換後の点の座標になるわけです。

ちなみに 3 次の場合も同様です。

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix} \tag{4.4}$$

4.5.3 単位行列

ベクトルと掛け算しても何も変わらない行列を単位行列と呼びます。

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{4.5}$$

実際にベクトルと掛け算してみましょう。

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \cdot x + 0 \cdot y \\ 0 \cdot x + 1 \cdot y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.6}$$

与えられたベクトルと同じ結果が得られました。この行列では、2 次元空間上の点の位置を全く変えません。

4.5.4 拡大行列

原点を中心に、X 軸方向に s_x 倍、Y 軸方向に s_y 拡大するような行列は次のようになります。

$$\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \tag{4.7}$$

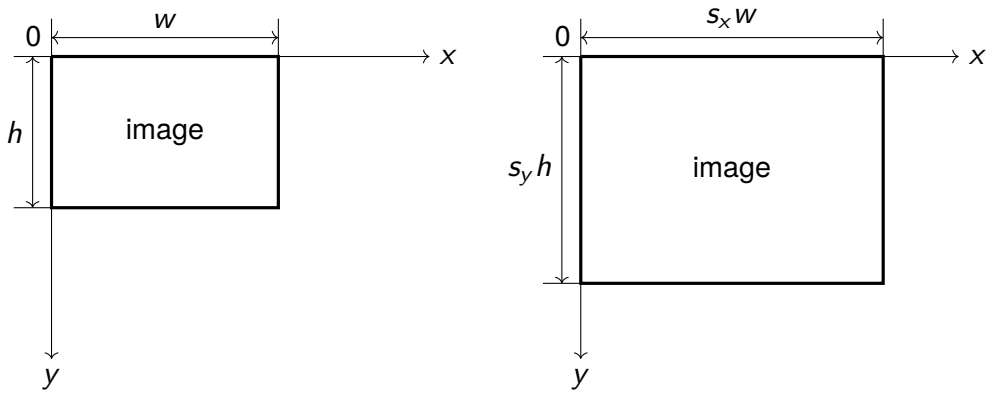


図 4.6 行列による拡大縮小変換

実際にベクトルと掛け算してみましょう。

$$\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_x \cdot x + 0 \cdot y \\ 0 \cdot x + s_y \cdot y \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \end{pmatrix} \tag{4.8}$$

4.5.5　回転行列

原点を中心に角度 θ だけ回転させる行列は次のようになります。三角関数を用います。

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{4.9}$$

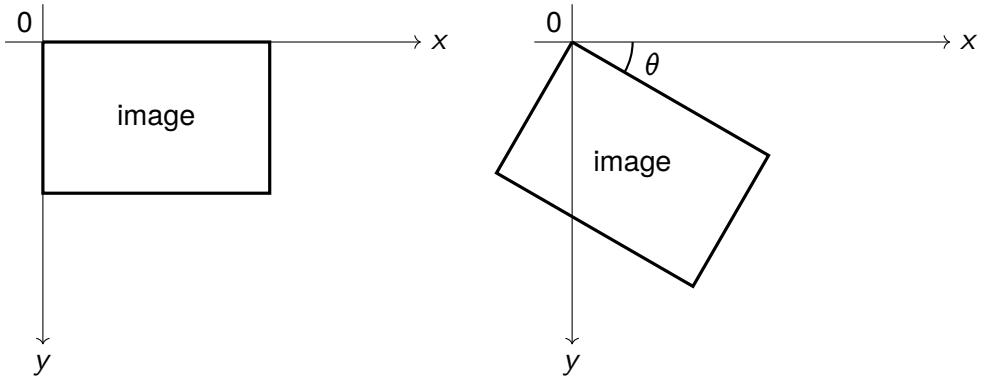


図 4.7　行列による回転変換

三角関数まで出てきてしまいました。Ebiten を使う上では、Go の関数一個で回転行列を適用できるので安心してください。

4.5.6　行列と行列の掛け算

たとえば拡大と回転を組み合わせたい場合はどうしたらよいでしょうか。結論から言うと、そういった変換ルールの合成も最終的に 1 つの行列で表すことができます。では実際に行列同士を組み合わせるとどうなるか見てみましょう。

ベクトルに対して行列を 2 回適用すると次のようになります。

$$\begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \begin{pmatrix} a_1x + b_1y \\ c_1x + d_1y \end{pmatrix} \tag{4.10}$$

$$= \begin{pmatrix} a_2(a_1x + b_1y) + b_2(c_1x + d_1y) \\ c_2(a_1x + b_1y) + d_2(c_1x + d_1y) \end{pmatrix} \tag{4.11}$$

$$= \begin{pmatrix} (a_2a_1 + b_2c_1)x + (a_2b_1 + b_2d_1)y \\ (c_2a_1 + d_2c_1)x + (c_2b_1 + d_2d_1)y \end{pmatrix} \tag{4.12}$$

$$= \begin{pmatrix} a_2a_1 + b_2c_1 & a_2b_1 + b_2d_1 \\ c_2a_1 + d_2c_1 & c_2b_1 + d_2d_1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{4.13}$$

なんだか物々しい式になってしまいました。この式の意味するところは、行列同士の乗算を次のように定義できるということです。

$$\begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} = \begin{pmatrix} a_2a_1 + b_2c_1 & a_2b_1 + b_2d_1 \\ c_2a_1 + d_2c_1 & c_2b_1 + d_2d_1 \end{pmatrix} \tag{4.14}$$

かくして、行列同士の合成が定義され、結果も行列になりました。ルール同士を合成した複雑なルールも、結局 1 つの行列で表されます。

なお行列同士の乗算について、式を一生懸命覚えなくても Ebiten は使えます[5]。ただ、ルール同士を合成した結果も 1 つの行列で表されるという事実は覚えておいたほうよいでしょう。

ちなみに 3 次行列の場合は次のようになります。

$$\begin{pmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ g_2 & h_2 & i_2 \end{pmatrix} \begin{pmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ g_1 & h_1 & i_1 \end{pmatrix} \tag{4.15}$$

$$= \begin{pmatrix} a_2a_1 + b_2d_1 + c_2d_1 & a_2b_1 + b_2e_1 + c_2h_1 & a_2c_1 + b_2f_1 + c_2i_1 \\ d_2a_1 + e_2d_1 + f_2d_1 & d_2b_1 + e_2e_1 + f_2h_1 & d_2c_1 + e_2f_1 + f_2i_1 \\ g_2a_1 + h_2d_1 + i_2d_1 & g_2b_1 + h_2e_1 + i_2h_1 & g_2c_1 + h_2f_1 + i_2i_1 \end{pmatrix} \tag{4.16}$$

注意しなければならないのは、行列同士の乗算は順序があるという点です。難しい言葉でいうと「非可換」なのです。行列 A と行列 B があったとして、AB と BA の結果は一般的には異なります。変換ルールの話で言うと、「回転してから拡大」するのと「拡大してから回転」するのは一般的には異なる、ということになります。

---

[5] 覚えておくと数学の試験には役に立つかもしれませんね☺ なお丸暗記する必要はなくて、ちゃんと規則性があります。乗算の結果は、左側の行列の行と右側の行列の列の各要素同士を掛けたものになっています。

4.5.7　アフィン変換行列

2 次元空間上の点を移動させるにはこれで十分なように見えますが、実は問題があります。原点 $(0, 0)$ に対してどんな 2 次行列を掛けても、原点は原点のままなのです。

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} a \cdot 0 + b \cdot 0 \\ c \cdot 0 + d \cdot 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{4.17}$$

これは何を意味するかというと、2 次行列を使っても画像は原点 (Ebiten では左上の座標) から絶対に移動しないことを意味します。平行移動に関しては行列で表現できないのでしょうか?

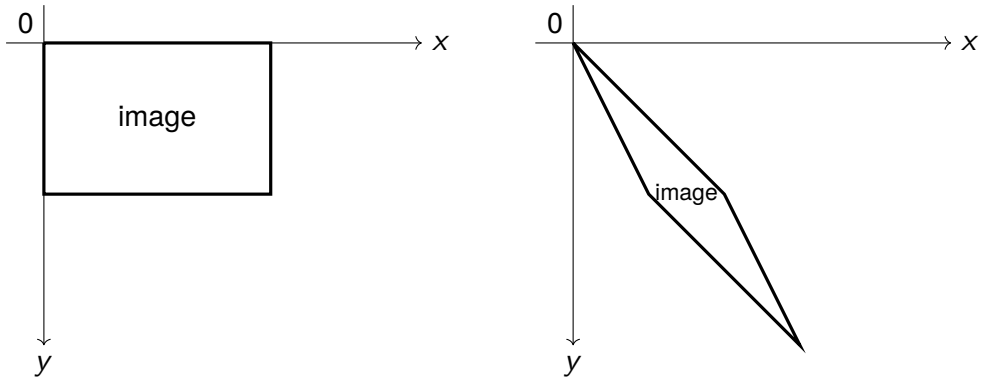


図 4.8　どんな行列変換を施しても原点は原点のまま?

そこでアフィン変換行列というのを使います。2 次ベクトルのためのアフィン変換行列は次のようになります。

$$\begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \tag{4.18}$$

なんと 3 次行列になってしまいました。ベクトルも 1 次元拡大し、3 つ目の値を常に 1 として扱います。いままで (x, y) だったのが $(x, y, 1)$ となるわけです。

平行移動する「だけ」のアフィン変換行列は次のようになります。

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \tag{4.19}$$

ためしにベクトル $(x, y, 1)$ と乗算させると、結果は X 成分と Y 成分だけ平行移動した $(x + t_x, y + t_y, 1)$ になることが分かります。

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot x + 0 \cdot y + t_x \cdot 1 \\ 0 \cdot x + 1 \cdot y + t_y \cdot 1 \\ 0 \cdot x + 0 \cdot y + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix} \tag{4.20}$$

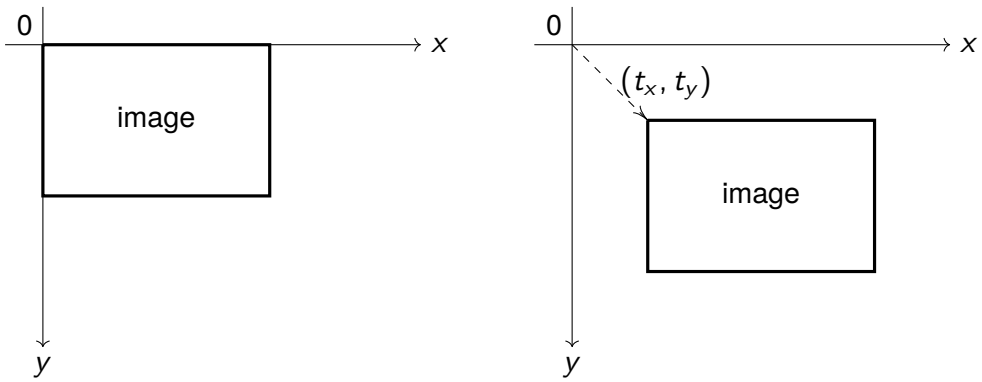


図 4.9 アフィン変換行列による平行移動

従来の拡大行列や回転行列は、t_x および t_y が 0 である行列になります。

なお証明は省略しますが、アフィン変換行列同士の乗算の結果は常にアフィン変換行列になります。これは何を意味するかというと、拡大縮小、回転、平行移動などのすべての変換、及びその組み合わせが、1 つのアフィン変換行列で表されるということです。この事実を利用して、Ebiten の幾何変換のための API は「行列を 1 つ指定する」という極めて単純なものになっています。なお Ebiten が扱う行列はすべてアフィン変換行列で、それ以外の行列を扱うことはありません。

4.6　描画位置

4.6.1　固定位置

アフィン変換行列について理解したところで、実際に使ってみましょう。単純に画像を原点とは違う位置に描画してみます。

```
// image_position.go

package main

import (
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // ebitenImage を位置 (20, 10) 画面に描画する。
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Translate(20, 10)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Image Position"); err != nil {
        log.Fatal(err)
    }
```

```
}
```

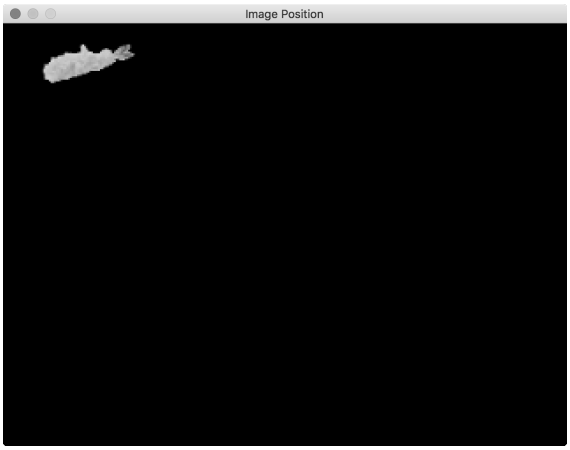


図 4.10 image_position.go の実行結果

ebiten.DrawImageOptions 構造体は、DrawImage の際の設定を表します。このオブジェクトの幾何行列などを指定するのです。

```
type DrawImageOptions struct {
    ImageParts    ImageParts    // 画像の一部分を描画する場合の指定
    GeoM          GeoM          // 幾何行列
    ColorM        ColorM        // 色行列
    CompositeMode CompositeMode // 合成方法
}
```

メンバーの GeoM は幾何行列を表します。デフォルトの状態では単位行列です。Transalte 関数を呼ぶことで平行移動成分を設定できますが、この関数は、現在の GeoM に平行移動のための行列を左から乗算します。

```
func (g *GeoM) Translate(tx, ty float64)
```

$$\begin{pmatrix} 1 & 0 & 20 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 20 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.21}$$

今回は Transalte を呼ぶ前は何もしていないので単位行列であり、結果的に GeoM

は単なる平行移動するだけのアフィン変換行列になります。

4.6.2　移動

毎フレーム少しずつ異なる GeoM を与えると、徐々に移動する画像を描画できます。

```
// image_move.go

package main

import (
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var (
    ebitenImage  *ebiten.Image
    ebitenImageX = 0 // X 座標
)

func update(screen *ebiten.Image) error {
    // 描画先の X 座標を更新する。
    ebitenImageX++
    if ebiten.IsRunningSlowly() {
        return nil
    }
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Translate(float64(ebitenImageX), 0)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Image Move"); err != nil {
```

```
        log.Fatal(err)
    }
}
```

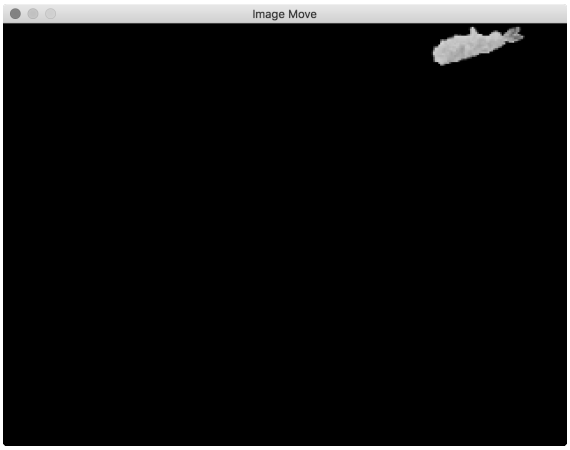


図 4.11　image_move.go の実行結果

実行すると、海老天の画像がどんどん右に移動していきます。なおしばらくすると海老天の画像が画面右端に行って、見えなくなってしまいます。

4.7　拡大・縮小

4.7.1　単純な拡大

```
// image_scale.go

package main

import (
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // ebitenImage を (原点中心に)2 倍拡大する。
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Scale(2, 2)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Image Scaling"); err != nil {
        log.Fatal(err)
    }
}
```

GeoM の Scale を使います。

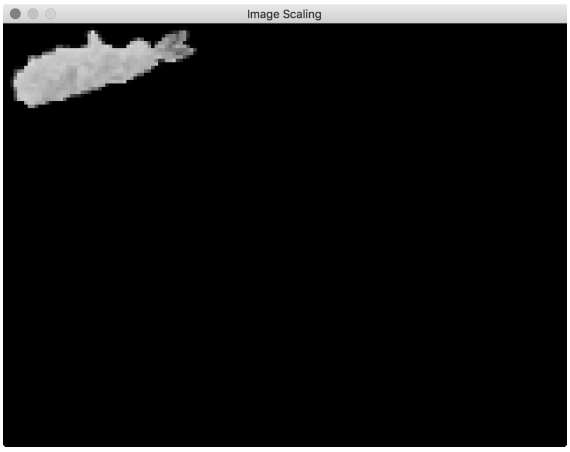


図 4.12　image_scale.go の実行結果

```
func (g *GeoM) Scale(x, y float64)
```

現在の行列に拡大行列を左から乗算します。x と y はそれぞれ X 軸方向、Y 軸方向の拡大率を表します。1.0 より大きい値ならば拡大、1.0 より小さい正の値ならば縮小になります。また負の値を指定すると鏡像になります。

行列の計算としては、次のようになります。GeoM のデフォルトは単位行列であり、それに対して Scale を呼んだだけなので、結果的に単なる拡大行列になります。

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.22}$$

4.7.2　平行移動と拡大

平行移動と拡大を組み合わせるとどうなるでしょうか。

```
// image_scale_translate.go

package main

import (
    _ "image/png"
    "log"
```

```
    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // ebitenImage を原点中心に 2 倍拡大した後、(20, 10) だけ平行移動する。
    // 順番を変えると違う結果になることに注意。
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Scale(2, 2)
    op.GeoM.Translate(20, 10)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2,
        "Image Scale and Translate"); err != nil {
        log.Fatal(err)
    }
}
```

幾何行列 GeoM に対して拡大行列、平行移動行列を左からこの順で掛けました。

$$\begin{pmatrix} 1 & 0 & 20 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 20 \\ 0 & 2 & 10 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.23}$$

注意しなければならないのは、この関数呼び出しの順番を入れ替えると違う結果になるということです。

図 4.13　image_scale_translate.go の実行結果

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 20 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 40 \\ 0 & 2 & 20 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.24}$$

拡大は「原点中心」に行うので、平行移動した後に拡大した場合は、平行移動成分もそれに合わせて拡大されます。

行列は左から掛かるので、操作の順番は行列積の右から左の順序になることに注意してください。式 4.23 (54 ページ) は「拡大してから平行移動」、式 4.24 (55 ページ) は「平行移動してから回転」です。

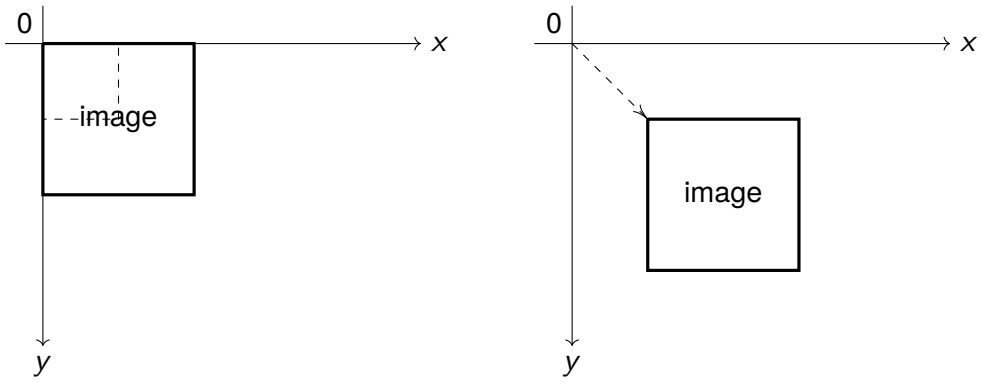


図 4.14　拡大してから平行移動

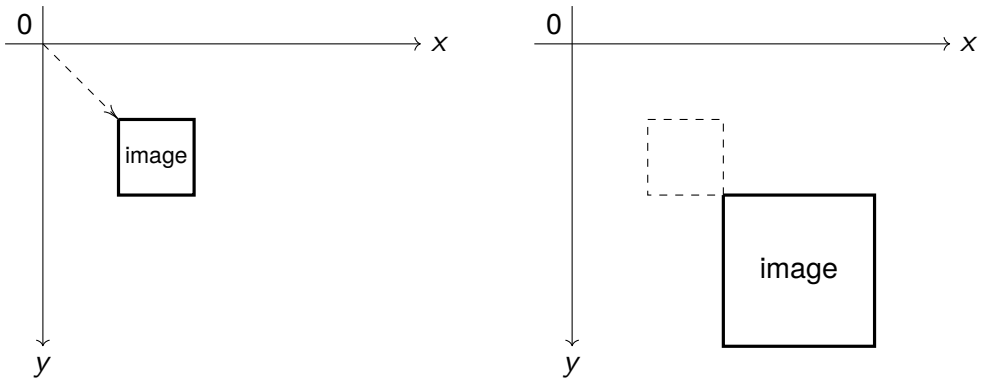


図 4.15　平行移動してから拡大。拡大等は原点中心に行われることに注意

4.8 フィルタ

いままでは ebiten.Image オブジェクト作成時に ebiten.FilterNearest が指定されていました。これは拡大縮小や回転などのときの補完方法です。Ebiten には以下の 2 種類のフィルタがあります。

ebiten.FilterNearest 最も単純な補完。拡大してもピクセルはそのまま大きな正方形になる。

ebiten.FilterLinear 線形補間を行う。拡大するとぼやけたようになる。

実際に拡大して ebiten.FilterLinear を使う例を見てみます。

```
// image_filters.go

package main

import (
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var (
    ebitenImageNearest *ebiten.Image
    ebitenImageLinear  *ebiten.Image
)

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // 4 倍拡大してそれぞれ描画する。
    // 大きめに拡大したのは、フィルタの効果をわかりやすくするため。
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Scale(4, 4)
    screen.DrawImage(ebitenImageNearest, op)
    op = &ebiten.DrawImageOptions{}
    op.GeoM.Scale(4, 4)
    op.GeoM.Translate(0, 100)
```

```
    screen.DrawImage(ebitenImageLinear, op)
    return nil
}

func main() {
    var err error
    // ebiten.Image オブジェクト作成時に FilterLinear を指定する。
    ebitenImageNearest, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    ebitenImageLinear, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterLinear)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Image Filters"); err != nil {
        log.Fatal(err)
    }
}
```

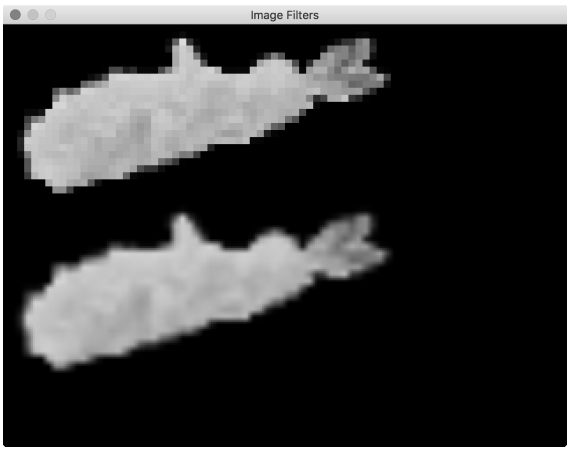


図 4.16　image_filters.go の実行結果

上の画像がニアレストフィルタ、下の画像がリニアフィルタです。ニアレストフィルタの方はギザギザに、リニアフィルタの方はぼやけた画像になったことがわかります。

なお本書では、断りがない限りは基本的に ebiten.FilterNearest を使用します。

4.9 回転

4.9.1 単純な回転

画像を時計回りに 45 度だけ傾けて描画してみます。

```
// image_rotate.go

package main

import (
    _ "image/png"
    "log"
    "math"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // ebitenImage を 45 度時計回りの向きに (原点中心に) 回転させる。
    // 45 度はラジアンでπ/4 である。
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Rotate(math.Pi / 4)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Image Rotating"); err != nil {
        log.Fatal(err)
    }
```

```
}
```

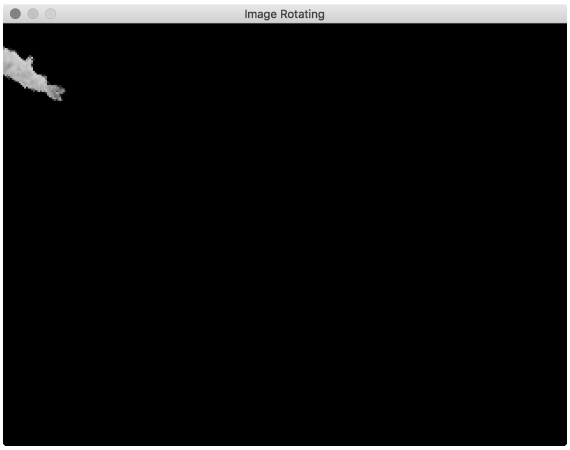


図 4.17　image_rotate.go の実行結果

先程の例と同じく、GeoM はデフォルトでは単位行列です。それに対して左から回転行列と平行移動行列を乗算する形になります。結果的には回転操作の後に平行移動する行列となります。

$$\begin{pmatrix} \cos\pi/4 & -\sin\pi/4 & 0 \\ \sin\pi/4 & \cos\pi/4 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\pi/4 & -\sin\pi/4 & 0 \\ \sin\pi/4 & \cos\pi/4 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.25}$$

Rotate 関数は与えられた角度の回転行列を左から乗算します。

```
func (g *GeoM) Rotate(theta float64)
```

指定する角度の単位はラジアンです。360 度系 (Degree) からラジアンに変換するには、以下の式を用います。

$$\theta_{\text{radian}} = \frac{\pi}{180}\theta_{\text{degree}} \tag{4.26}$$

指定した角度の分、時計回りの向きに原点中心で回転します。一般的な数学の座標系における回転は反時計回りで、これとは異なります。Ebiten の座標系だと Y 軸が下向きだからです。

4.9.2 回転中心

画像を回転できたのはいいのですが、いつでも原点が回転中心なのはいささか不便です。実は平行移動とうまく組み合わせることで、任意の位置を回転中心にすることができます。今回は画像の中心を回転中心にしてみましょう。

```
// image_rotate_center.go

package main

import (
    _ "image/png"
    "log"
    "math"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // 画像の大きさを得る。
    w, h := ebitenImage.Size()
    // ebitenImage を時計回りの向きに、画像の中心を回転中心として
    // 45 度回転させる。
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Translate(-float64(w)/2, -float64(h)/2)
    op.GeoM.Rotate(math.Pi / 4)
    op.GeoM.Translate(float64(w)/2, float64(h)/2)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
```

```
    }
    if err := ebiten.Run(update, 320, 240, 2, "Image Rotating"); err != nil {
        log.Fatal(err)
    }
}
```

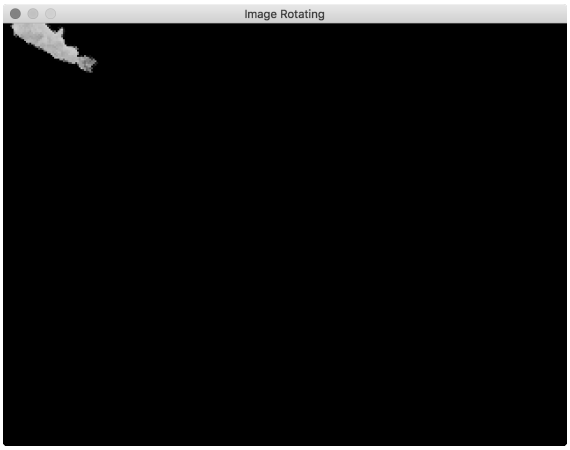


図 4.18　image_rotate_center.go の実行結果

回転中心を (c_x, c_y) として、その座標が原点の位置に来るように $(-c_x, -c_y)$ だけ一旦平行移動します。回転した後に、また元の位置に (c_x, c_y) だけ平行移動して戻してあげると、結果的に (c_x, c_y) を中心とした回転が得られます。

$$\begin{pmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{4.27}$$

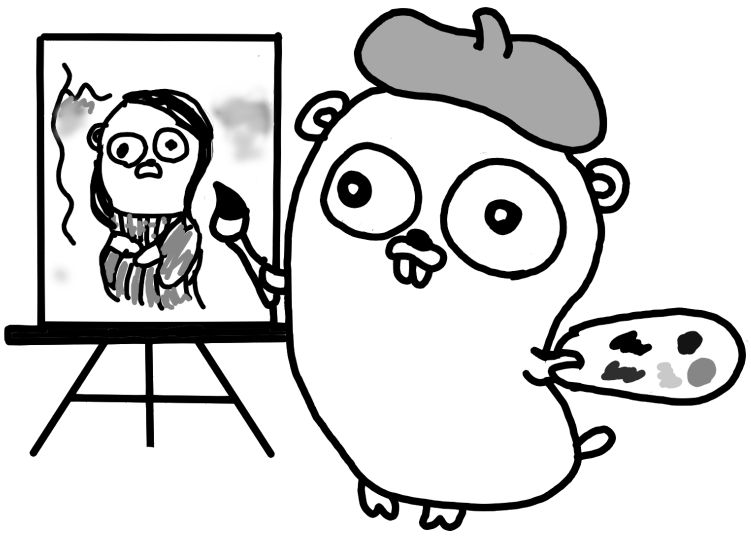
今回は画像の中心が回転中心です。画像の大きさを w、h とすると、回転中心は w/2、h/2 となります。プログラムを見ていただくと分かる通り、上記行列の計算と全く同じことをしています。前述のとおり、行列は左から掛かっているので、操作の順は行列積の右から左になります。

```
    op.GeoM.Translate(-float64(w)/2, -float64(h)/2)
    op.GeoM.Rotate(math.Pi/4)
    op.GeoM.Translate(float64(w)/2, float64(h)/2)
```

第 5 章

描画・応用

前章までの知識をベースに、応用的な描画について説明します。主にオフスクリーンを使った描画、色行列、部分描画機能を紹介します。



5.1　オフスクリーンレンダリング

オフスクリーンレンダリングとは、画面ではないところに対して描画を行い、その描画結果を利用して画面などに描画することです。この画面外の描画先をオフスクリーンバッファと呼びます。

Ebiten でオフクリーンバッファとなる ebiten.Image オブジェクトを作るには、NewImage 関数を使います。

```
func NewImage(width, height int, filter Filter) (*Image, error)
```

NewImage 関数のエラー値は常に nil になります。アンダースコア (_) などを使って握りつぶせます。NewImageFromImage 関数も同様です[1]。

実際にオフスクリーンバッファを使用する例を見てみましょう。

```
// offscreen.go

package main

import (
    _ "image/png"
    "log"
    "math"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var (
    ebitenImage    *ebiten.Image
    offscreenImage *ebiten.Image
    angle          int
)

func update(screen *ebiten.Image) error {
    angle++
    angle %= 360
```

[1] 似たような関数で ebitenutil パッケージの NewImageFromFile 関数がありますが、この関数のエラー値は無視できません。ファイル読み込みの IO エラー等が発生する可能性があるからです。

```
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // オフスクリーンバッファをクリアする。
    // screen と違い自動的にクリアされないので、明示的に呼ぶ必要がある。
    offscreenImage.Clear()
    // オフスクリーンバッファに対し、2 個海老天の画像を描画する。
    // うち 1 個は回転させる。
    op := &ebiten.DrawImageOptions{}
    offscreenImage.DrawImage(ebitenImage, op)
    op = &ebiten.DrawImageOptions{}
    w, h := ebitenImage.Size()
    // 画像の中心を回転中心として angle 度分回転する。
    op.GeoM.Translate(-float64(w)/2, -float64(h)/2)
    op.GeoM.Rotate(float64(angle) * math.Pi / 180)
    op.GeoM.Translate(float64(w)/2, float64(h)/2)
    op.GeoM.Translate(float64(h)/2, 0)
    offscreenImage.DrawImage(ebitenImage, op)
    // オフスクリーンバッファを画面に描画する。
    op = &ebiten.DrawImageOptions{}
    screen.DrawImage(offscreenImage, op)
    // 同じオフスクリーンバッファを拡大して描画する。
    op = &ebiten.DrawImageOptions{}
    op.GeoM.Scale(2, 2)
    op.GeoM.Translate(100, 100)
    screen.DrawImage(offscreenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    // オフスクリーンレンダリング用のバッファを作成する。
    // 今回は main 関数内で生成した。
    // なお ebiten.Image を生成する処理は重い処理であり、
    // update 関数内で毎フレーム生成するのは推奨されない。
    offscreenImage, err = ebiten.NewImage(100, 100, ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
```

```
    if err := ebiten.Run(update, 320, 240, 2, "Offscreen"); err != nil {
        log.Fatal(err)
    }
}
```

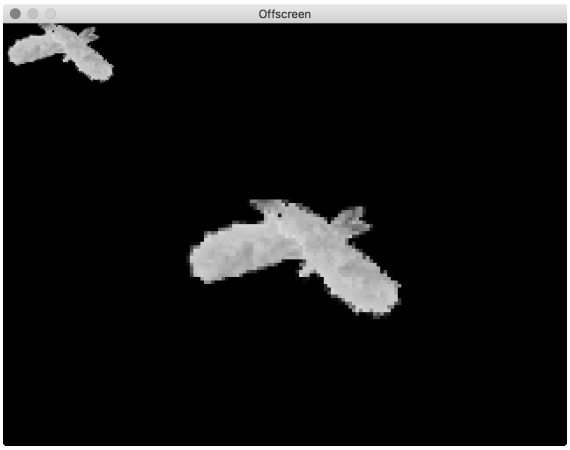


図 5.1　offscreen.go の実行結果

実際に動かしてみると、2 つのアニメーションが全く同じようにアニメーションしていることが分かります。2 つの海老天の画像は画面ではなく一旦オフスクリーンバッファに描画され、そのオフスクリーンバッファを 2 回画面に異なる方法で描画しています。オフスクリーンバッファは、正確には ebiten.Image オブジェクトは矩形なので、その範囲からはみ出る部分は描画されていないことも分かります。

Ebiten には画面、画像ファイル、オフスクリーンバッファの間の区別はなく、すべて ebiten.Image オブジェクトとして統一して扱われます。実は、ファイルから作成した ebiten.Image オブジェクトもオフスクリーンバッファとして転用することが可能です。この 3 つには実質上の区別がほぼないからです[2]。

update 関数に渡ってくる screen は使用時に毎度クリアされています。ここでいうクリアとは、すべてのピクセルの値が 0[3]になっていることです。しかしオフスクリーンバッファとしての ebiten.Image は明示的に Clear を呼ばないとクリアされません。

[2] 画面を表す ebiten.Image は毎フレーム自動的にクリアされます。実質上の唯一の違いはそれだけです。

[3] R、G、B、A の値がすべて 0 という意味。ちなみに、色のフォーマットはプリマルチプライドアルファなので、アルファ値が 0 ならば RGB の値も 0 になります。

毎フレームオフスクリーンバッファを更新するのであれば毎回 Clear を呼ぶ必要があります。しかしこのことを逆に利用して、オフスクリーンバッファをあえてクリアせず、描画結果のキャッシュとして利用することも可能です。

```
func (i *Image) Clear() error
```

Clear 関数は画像のピクセルをすべてゼロ値にします。Fill(color.Transparent) と結果は同じです。

Clear 関数は常に nil を返します。帰り値は無視して構いません。

注意しなければならないのは、オフスクリーンレンダリングバッファは 1 個しか作成していないことです。ebiten.NewImage などの ebiten.Image オブジェクトを生成する処理は重い処理なので、毎フレーム生成するなどといったことはしてはいけません。

5.2　オフスクリーンバッファの使い回し

前節で ebiten.Image オブジェクトをオフスクリーンバッファとして使う方法を説明しました。ではオフスクリーンバッファを使って画面に描画した後に、オフスクリーンバッファの内容を消してしまうとどうなるでしょうか。結論を言うと、画面の DrawImage を呼び出した時点での、オフスクリーンバッファの状態がそのまま採用されます。その後オフスクリーンバッファをどう操作しようとも、描画結果に影響を与えることはありません。このことを利用して、オフスクリーンバッファを違う描画内容の受け皿として使いまわすことが出来ます。例を見てみましょう。

```
// reusing_offscreen.go

package main

import (
    _ "image/png"
    "log"
    "math"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var (
    ebitenImage    *ebiten.Image
    offscreenImage *ebiten.Image
    angle          int
)

func update(screen *ebiten.Image) error {
    angle++
    angle %= 360
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // オフスクリーンバッファを初期化する。
    // 前節で見た通り、オフスクリーンは自前で初期化する必要がある。
    offscreenImage.Clear()
    // オフスクリーンバッファに対し、1 個海老天の画像を描画する。
    op := &ebiten.DrawImageOptions{}
```

```
    offscreenImage.DrawImage(ebitenImage, op)
    // オフスクリーンバッファを画面に描画する。
    op = &ebiten.DrawImageOptions{}
    screen.DrawImage(offscreenImage, op)

    // オフスクリーンバッファをまた消去する。
    // 直前の DrawImage については、その時点でのオフスクリーンバッファの
    // 内容が使われるので、影響はない。
    offscreenImage.Clear()
    // 海老天を回転して描画する。
    op = &ebiten.DrawImageOptions{}
    w, h := ebitenImage.Size()
    op.GeoM.Translate(-float64(w)/2, -float64(h)/2)
    op.GeoM.Rotate(float64(angle) * math.Pi / 180)
    op.GeoM.Translate(float64(w)/2, float64(h)/2)
    op.GeoM.Translate(float64(h)/2, 0)
    offscreenImage.DrawImage(ebitenImage, op)
    // オフスクリーンバッファを画面に別の場所に描画する。
    op = &ebiten.DrawImageOptions{}
    op.GeoM.Translate(50, 0)
    screen.DrawImage(offscreenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    offscreenImage, err = ebiten.NewImage(100, 100, ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Reusing Offscreen"); err != nil {
        log.Fatal(err)
    }
}
```

オフスクリーンバッファに内容を描いた後、それを画面に描画します。その直後にオフスクリーンバッファの内容を Clear で内容を消去しています。この消去操作は、直前の描画に影響を与えません。画面 screen に対する DrawImage を呼び出した時点での

図 5.2　reusing_offscreen.go の実行結果

オフスクリーンバッファがそのまま使われます。

5.3　モザイク

モザイクは画像をわざと粗く表示させる演出です。スーパーファミコンのゲームでよく使われます。Ebiten ではモザイクを直接実現する方法はありませんが、既存の仕組みの組み合わせでできます。ニアレストフィルターで縮小したものをオフスクリーンに描画したあと、それをまたニアレストフィルターで拡大描画するのです。

いままで使ってきた海老天の画像は、モザイクかけるにしては小さくてよく分からない画像です。今回は Go のマスコットである Gopher くんの画像[4]を用意して、モザイクをかけてみます。また結果がよく分かるようにモザイクをかける前とかけた後を横に並べてみます。

図 5.3　gopher.png

```
// mosaic.go

package main

import (
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
```

[4] https://github.com/egonelbre/gophers の witch-too-much-candy.png です。作者は Egon Elbre 氏、ライセンスは CC0 です。

```
)

var (
    // Gopher の画像。
    gopherImage *ebiten.Image
    // 一旦縮小した Gopher を描画するためのオフスクリーンバッファ。
    tmpImage *ebiten.Image
)

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    const scale = 4.0
    // Gopher をオフスクリーンバッファに縮小して描画する。
    tmpImage.Clear()
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Scale(1/scale, 1/scale)
    tmpImage.DrawImage(gopherImage, op)
    // Gopher を画面に描画する。
    op = &ebiten.DrawImageOptions{}
    screen.DrawImage(gopherImage, op)
    // オフスクリーンバッファ画面に違う位置に描画する。
    op = &ebiten.DrawImageOptions{}
    op.GeoM.Scale(scale, scale)
    op.GeoM.Translate(100, 0)
    screen.DrawImage(tmpImage, op)
    return nil
}

func main() {
    var err error
    gopherImage, _, err = ebitenutil.NewImageFromFile("./gopher.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    // オフスクリーンバッファを生成する。
    // 大きさは gopherImage の大きさ以上であればなんでも良い。
    // とりあえず画面と同じ大きさにした。
    // モザイクのためには、フィルタは Nearest フィルタである必要がある。
    tmpImage, err = ebiten.NewImage(320, 240, ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
```

```
    }
    if err := ebiten.Run(update, 320, 240, 2, "Mosaic"); err != nil {
        log.Fatal(err)
    }
}
```

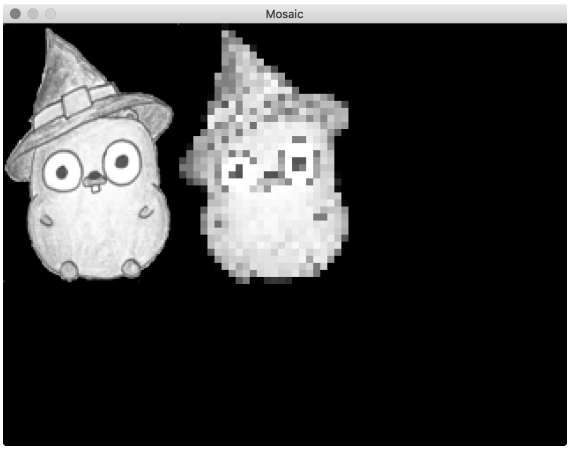


図 5.4　mosaic.go の実行結果

5.4　色変換行列

これまでは描画する画像の幾何変形を行ってきました。この節では行列による画像の色変形を行います。

5.4.1　色

Ebiten で扱う色は赤、緑、青、アルファ値の 4 つの数値で表されます。それぞれの Red、Green、Blue、Alpha の頭文字をとって RGBA と呼びます。

RGB はそれぞれ光の三原色である赤、青、緑を表します。あらゆる色はこの 3 色の合成で表されます。たとえば黄色は赤、緑を最大に、青をゼロにすることで得られます。

アルファ値は「不透明度」を表す値です。最大値ならば完全に不透明、0 ならば完全に透明、その他の値ならば半透明になります。これは画像の描画などの際に色を合成する時の計算に使用されます。

各成分の値の範囲は 0 から 1 です。Go 上の表現だと 0 から 255 だったり 0 から 65535 だったりしますが、これはあくまでプログラム上の表現であり、数学的には 0 から 1 の値として計算されます。

Ebiten では `image/color` パッケージの `RGBA` 型を主に扱います。実際に `ebiten.Image` オブジェクトの内部状態はそれで表され、また `ColorModel` 関数は `color.RGBAModel` を返します。

5.4.2　プリマルチプライドアルファ

`color.RGBA` はプリマルチプライドアルファと呼ばれる形式です。これは、RGB の値にアルファ値の値が乗算された値になっている形式です。なぜこの形式になっているかの詳しい説明は、この本の趣旨から外れるため割愛します[5]。逆にアルファ値が乗算されていない普通の RGBA 値をストレートアルファといいます。たとえばストレートアルファで $(0, 0.5, 1, 0.5)$ と表される色があったとします。その場合、プリマルチプライド

[5] プリマルチプライドアルファを採用する理由は端的には「そうでなければならないケースがあるため」です。RGBA 同士の合成の計算を行う際、プリマルチプライドアルファならば OpenGL の `glBlendFunc` の指定だけで出来ますが、ストレートアルファだと `glBlendFunc` だけで正確に行うのは不可能です。また、拡大時のアルファ値の補完の計算もプリマルチプライドアルファでないと不正確なものになります。

アルファ形式だと $(0, 0.25, 0.5, 0.5)$ となります。

5.4.3 色変換行列

ところで色も、幾何変換のときに扱った 2 次ベクトルの座標と同じく、4 次ベクトルとして扱うことが出来ます。R、G、B、A の 4 つの数値で構成されるので 4 次になるわけです。幾何行列の場合はアフィン変換行列である 3 次行列を使いました。色の場合、アフィン変換行列だと 5 次行列になります。

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} & t_r \\ m_{21} & m_{22} & m_{23} & m_{24} & t_g \\ m_{31} & m_{32} & m_{33} & m_{34} & t_b \\ m_{41} & m_{42} & m_{43} & m_{44} & t_a \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r \\ g \\ b \\ a \\ 1 \end{pmatrix} \tag{5.1}$$

$$= \begin{pmatrix} m_{11}r + m_{12}g + m_{13}b + m_{14}a + t_r \\ m_{21}r + m_{22}g + m_{23}b + m_{24}a + t_g \\ m_{31}r + m_{32}g + m_{33}b + m_{34}a + t_b \\ m_{41}r + m_{42}g + m_{43}b + m_{44}a + t_a \\ 1 \end{pmatrix} \tag{5.2}$$

行列に適用する色はプリマルチプライドアルファではなくストレートアルファ形式になります。行列適用前に、プリマルチプライドアルファ形式の色は一旦ストレートアルファに変換されます。最終結果は再びプリマルチプライドアルファに戻されます。

色行列は強力で、半透明処理などの単純な処理はもちろん、モノクロやセピア変換、色相変換などが行列だけで出来てしまいます。

5.5　半透明

画像を半透明で描画してみます。

```
// translucent.go

package main

import (
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // 色行列を使用する。
    // Scale 関数でアルファ値だけ半分にする。
    op := &ebiten.DrawImageOptions{}
    op.ColorM.Scale(1, 1, 1, 0.5)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Translucent"); err != nil {
        log.Fatal(err)
    }
}
```

ColorM 構造体が色行列を表す構造体です。GeoM の時と同じく、関数を呼ぶことで乗

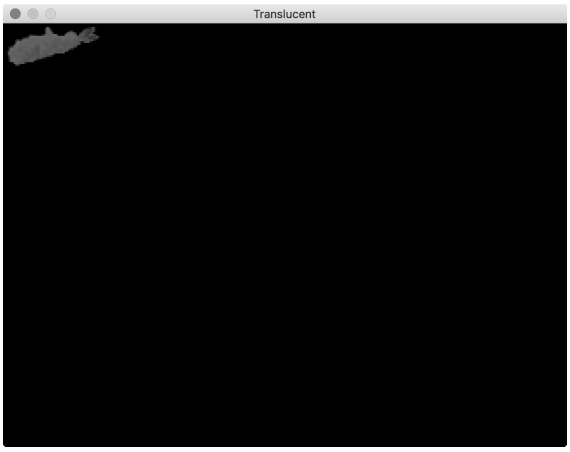


図 5.5　translucent.go の実行結果 (カラー版は i ページ図 3)

算していきます。Scale 関数は現在の色行列に拡大行列を乗算する関数です。

```
func (c *ColorM) Scale(r, g, b, a float64)
```

それぞれ R、G、B、A の成分に対する乗算値を指定します。今回はアルファ値だけ 0.5 倍します。また ColorM の初期値は単位行列なので、乗算の結果は結局次のようになります。

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.3}$$

数字がたくさん出てきて物々しいですが、出てきた結果はアルファ値を 0.5 倍するだけの行列です。

5.6　モノクロ

今度は色調を変換してみましょう。色を白黒にして海老天の画像を描画してみます。

```
// monochrome.go

package main

import (
    _ "image/png"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // 2 番目の引数の彩度を 0 にして白黒にする。
    op := &ebiten.DrawImageOptions{}
    op.ColorM.ChangeHSV(0, 0, 1)
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Monochrome"); err != nil {
        log.Fatal(err)
    }
}
```

ChagneHSV は色相、彩度、明度を変更する行列を乗算する関数です。

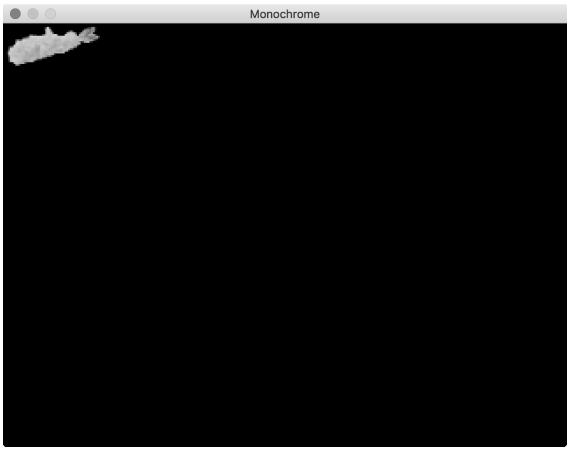


図 5.6 monochrome.go の実行結果 (カラー版は i ページ図 4)

```
func (c *ColorM) ChangeHSV(hueTheta float64, saturationScale float64,
        valueScale float64)
```

第 1 引数の hueTheta は色相を調節する値で、色相環の角度をラジアンで指定します。0 を指定すると無変更になります。

第 2 引数の saturationScale は彩度を調節する値で。彩度は色の鮮やかさを表します。0 だと完全に白黒、1 だと何も変化しません。1 より大きい値を指定することも可能で、より鮮やかな色になります。

第 3 引数の valueScale は明度を調節する値です。明度は色の明るさを表します。0 だと完全に真っ暗に、1 だと何も変化しません。1 より大きい値を指定すると白に近づきます。

今回は saturationScale として 0 を指定したので、完全に白黒になりました。

ChangeHSV 関数が呼ばれる時、行列の計算としては若干複雑な計算をします。まず次の行列を掛けて、RGB 色空間を YCbCr という色空間に変換します。

$$\begin{pmatrix} 0.2990 & 0.5870 & 0.1140 & 0 & 0 \\ -0.1687 & -0.3313 & 0.5000 & 0 & 0 \\ 0.5000 & -0.4187 & -0.0813 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.4}$$

続いて、色相変換角度を θ として、次の行列を乗算します。

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 & 0 \\ 0 & \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.5}$$

彩度のスケールを s、明度のスケールを v として、次の行列を乗算します。

$$\begin{pmatrix} v & 0 & 0 & 0 & 0 \\ 0 & sv & 0 & 0 & 0 \\ 0 & 0 & sv & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.6}$$

最後に、次の行列を乗算して YCbCr 色空間から RGB 色空間に戻します。

$$\begin{pmatrix} 1 & 0 & 1.40200 & 0 & 0 \\ 1 & -0.34414 & -0.71414 & 0 & 0 \\ 1 & 1.77200 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.7}$$

このように内部実装は実に複雑なことをしているのですが、すべてを理解する必要はありません。彩度や色相の変換も、行列の乗算演算だけで出来てしまうというのがポイントです[6]。

[6] HSV 変換は行列を使うやり方だけではなく、より正確なやり方も存在します。ただ、より正確な方法を採用すると、非線形変換となり、行列で表現できなくなります。Ebiten では正確性を妥協して、行列だけで変換出来る方法を採用しました。逆に言うと、Ebiten における HSV 変換は多少の誤差を伴います。

5.7 色相

```
// hue.go

package main

import (
    _ "image/png"
    "log"
    "math"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // 色々色相を変えて画像を描画する。
    for i, a := range []int{0, 60, 120, 180, 240, 300} {
        op := &ebiten.DrawImageOptions{}
        op.GeoM.Translate(float64(i)*50, 0)
        op.ColorM.RotateHue(float64(a) * math.Pi / 180)
        screen.DrawImage(ebitenImage, op)
    }
    return nil
}

func main() {
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Hue"); err != nil {
        log.Fatal(err)
    }
}
```

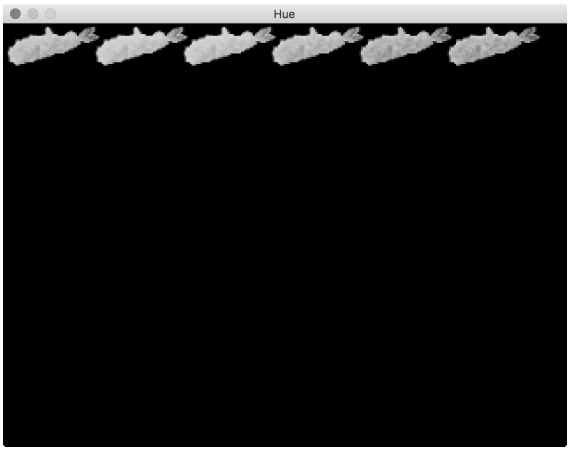


図 5.7　hue.go の実行結果 (カラー版は i ページ図 5)

今回は RotateHue 関数を使いました。

```
func (c *ColorM) RotateHue(theta float64)
```

RotateHue(theta) は ChangeHSV(theta, 1, 1) と効果は同じで、色相だけ変化させます。このサンプルでは 6 種類の角度を指定して、横位置を少しずつずらして描画します。レインボー海老天の出来上がりです。

5.8　セピア

色行列の強力さを観て見るために他の例を見てみましょう。Gopher くんの写真[7]をセピア色にします。セピア色はゲームでよく使われる表現ですね。

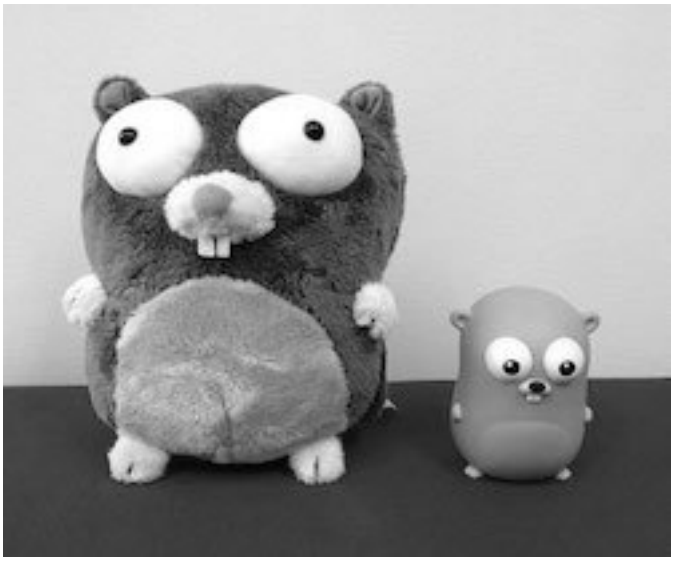


図 5.8　gophers_photo.jpg (カラー版は ii ページ図 6)

セピア色効果をわかりやすくするため、セピア色にする前後の写真を交互に描画します。

```
// sepia.go

package main

import (
    _ "image/jpeg"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)
```

[7] http://blog.golang.org/go-programming-language-turns-two より、Chris Nokleberg 氏の写真です。ライセンスは Creative Commons Attribution 3.0 です。

```
var (
    tick         = 0
    gophersImage *ebiten.Image
)

func update(screen *ebiten.Image) error {
    // tick は 1 フレームごとに 1 増える値である。
    // 120 フレーム (2 秒) で 0 に戻す。
    tick++
    tick %= 120
    if ebiten.IsRunningSlowly() {
        return nil
    }
    op := &ebiten.DrawImageOptions{}
    // 1 秒ごとにセピア色にする。
    if 60 <= tick {
        op.ColorM.ChangeHSV(0, 5.0/15.0, 1)
        op.ColorM.Translate(2.0/15.0, -2.0/15.0, -4.0/15.0, 0)
    }
    screen.DrawImage(gophersImage, op)
    return nil
}

func main() {
    var err error
    gophersImage, _, err = ebitenutil.NewImageFromFile("./gophers_photo.jpg",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Sepia"); err != nil {
        log.Fatal(err)
    }
}
```

ColorM 構造体の Translate を用いました。

```
func (c *ColorM) Translate(r, g, b, a float64)
```

GeoM 構造体の Translate と同じ要領で、色味を指定した分だけ平行移動させます。色の値の範囲は 0 から 1 であることに注意してください。

今回は ChangeHSV 関数を呼んだあとに Translate 関数を呼びました。ColorM も行

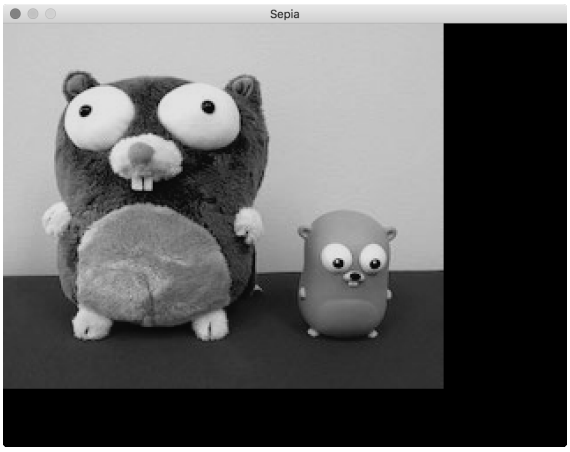


図 5.9　sepia.go の実行結果 (カラー版は ii ページ図 7)

列なので、乗算の順序に意味があります。この呼び出し順序を逆にすると、結果が変わってしまいます。

5.9　部分描画

今までの描画では、一つの ebiten.Image オブジェクトの全体を 1 回描画するだけでした。では、その Image の一部分を描画したい場合はどうすればよいでしょうか。Ebiten では DrawImageOptions の ImageParts を指定します。ImageParts に指定する ImageParts インターフェイスの定義は次のとおりです。

```
type ImageParts interface {
    Len() int
    Dst(i int) (x0, y0, x1, y1 int)
    Src(i int) (x0, y0, x1, y1 int)
}
```

Len 関数は描画するパートの数を表します。

Dst 関数は i 番目のパートに対応する描画先の座標、Src 関数は描画元の座標を返します。それぞれ $(x0, y0)$ は左上の座標、$(x1, y1)$ は右下の座標を表します。

ImageParts はインターフェイスなので、これを満たすように型を実装していきます。具体例を見てみましょう。

```
// parts.go

package main

import (
    _ "image/jpeg"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var gophersImage *ebiten.Image

// ebiten.ImageParts インターフェイスを実装する構造体。
// 4 つのパーツに分割して描画する。
type fourParts struct {
    // 描画対象となる画像。
    // この構造体内ではサイズを取得するためだけに使う。
```

```
    image *ebiten.Image
}

// 描画するパーツの数を返す関数。
func (f *fourParts) Len() int {
    // 今回は 4 つのパーツを描画するので、定数 4 を返す。
    return 4
}

// 描画元矩形の範囲を返す関数。
func (f *fourParts) Src(i int) (int, int, int, int) {
    // Len が 4 を返すので、i の範囲は 0 から 3 である。
    // それぞれに対応した描画元矩形の範囲を返す。
    // それぞれ左上、右上、左下、右下のパーツに分解している。
    w, h := f.image.Size()
    switch i {
    case 0:
        return 0, 0, w / 2, h / 2
    case 1:
        return w / 2, 0, w, h / 2
    case 2:
        return 0, h / 2, w / 2, h
    case 3:
        return w / 2, h / 2, w, h
    default:
        panic("not reach")
    }
}

// 描画先矩形の範囲を返す関数。
func (f *fourParts) Dst(i int) (int, int, int, int) {
    // Len が 4 を返すので、i の範囲は 0 から 3 である。
    w, h := f.image.Size()
    // 少し隙間を開けて表示する。
    const margin = 10
    switch i {
    case 0:
        return 0, 0, w / 2, h / 2
    case 1:
        return w/2 + margin, 0, w + margin, h / 2
    case 2:
        return 0, h/2 + margin, w / 2, h + margin
    case 3:
        return w/2 + margin, h/2 + margin, w + margin, h + margin
```

```
    default:
        panic("not reach")
    }
}

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // ImageParts として fourParts を指定する。
    op := &ebiten.DrawImageOptions{}
    op.ImageParts = &fourParts{gophersImage}
    screen.DrawImage(gophersImage, op)
    return nil
}

func main() {
    var err error
    gophersImage, _, err = ebitenutil.NewImageFromFile("./gophers_photo.jpg",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Parts"); err != nil {
        log.Fatal(err)
    }
}
```

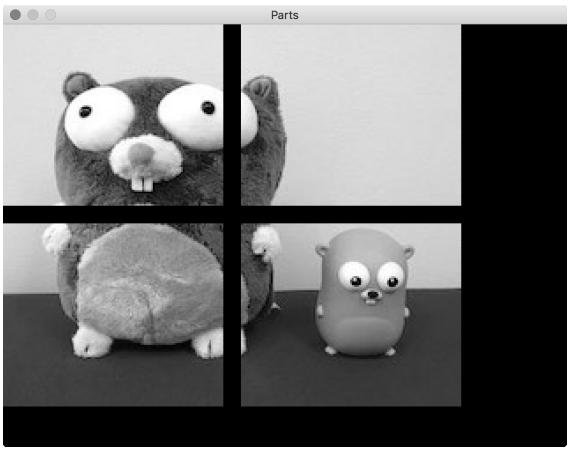


図 5.10　parts.go の実行結果

写真がバラバラになって表示されました。今回は画像を 4 つのパーツに分解して描画しました。このように ImageParts は描画元の一部分だけを描画するのに便利です。また後述しますが、描画元が同じならば一度に大量のパーツを描画しても負荷は変わらないので、たとえばマップタイルを描画する等の目的にもぴったりです。

もちろんこの描画に GeoM や ColorM を指定して描画することも可能です。

5.10　大量スプライト

ebiten.ImageParts は「一部分描画する」用途以外に、同じ Image を連続して大量に描画するという処理にも使えます。実装の都合上、DrawImage を複数回大量に呼ぶよりも、ImageParts を指定して DrawImage を呼ぶのを 1 回だけに済ますほうが、実行効率が高くなります。ドローコールの回数が 1 回で済むからです[8]。

大量に海老天の画像を描画する例を見てみましょう。

```
// sprites.go

package main

import (
    _ "image/jpeg"
    "log"
    "math/rand"
    "time"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var ebitenImage *ebiten.Image

// 座標を表す構造体。
type position struct {
    x int
    y int
}

// ebiten.ImageParts インターフェイスを実装する。
type sprites struct {
    image     *ebiten.Image
    positions []position
}
```

[8] これは実装の都合でしかないので、本来はそのような使い分けを考慮する必要がない API のほうが望ましいのです。しかしながら、複数回 DrawImage を呼ぶと、内部処理の関係でどうしてもパフォーマンスが出ないため、妥協しています。将来的には、DrawImage を複数回大量に呼んでも問題がないようにしたいと考えています。2017 年 4 月現在もこの目標を念頭に Ebiten のチューニングを進めています。

```
func (s *sprites) Len() int {
    return len(positions)
}

func (s *sprites) Src(i int) (int, int, int, int) {
    // 描画元矩形は画像そのまま使う。
    w, h := s.image.Size()
    return 0, 0, w, h
}

func (s *sprites) Dst(i int) (int, int, int, int) {
    // X 座標と Y 座標を i に合わせて変更する。
    w, h := s.image.Size()
    p := s.positions[i]
    return p.x, p.y, p.x + w, p.y + h
}

var positions []position

func init() {
    // 乱数シードを設定する。
    rand.Seed(time.Now().UnixNano())
    // 各スプライトの座標を乱数で決定する。
    positions = make([]position, 100)
    for i := range positions {
        positions[i].x = rand.Intn(320)
        positions[i].y = rand.Intn(240)
    }
}

func update(screen *ebiten.Image) error {
    if ebiten.IsRunningSlowly() {
        return nil
    }
    op := &ebiten.DrawImageOptions{}
    op.ImageParts = &sprites{
        image:     ebitenImage,
        positions: positions,
    }
    screen.DrawImage(ebitenImage, op)
    return nil
}

func main() {
```

```
    var err error
    ebitenImage, _, err = ebitenutil.NewImageFromFile("./ebiten.png",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    if err := ebiten.Run(update, 320, 240, 2, "Sprites"); err != nil {
        log.Fatal(err)
    }
}
```

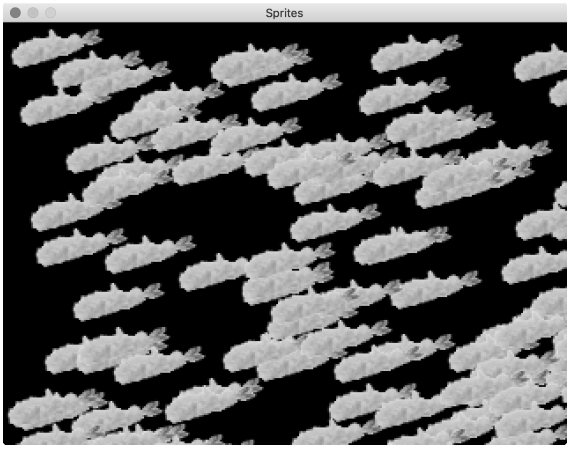


図 5.11　sprites.go の実行結果

大量の海老天の画像が描画されました。ebiten.ImageParts を用いているので、DrawImage の呼び出し 1 回でまとめて全てが描画できます。

なお乱数のシードは現在時刻で初期化しています。そのため起動する度に結果が変わるはずです。

5.11　ラスタスクロール

ebiten.ImageParts を使った応用例として、ラスタスクロールがあります。ラスタスクロールとは、スーパーファミコンなどの昔のゲームに見られた表現で、ビデオ信号の走査タイミングに合わせて描画内容を制御すること得られる描画効果です。Ebiten では、画像を 1 ラインごとに細かいパーツに分解して、位置をずらして描画することで実現できます。

```
// raster_effect.go

package main

import (
    _ "image/jpeg"
    "log"
    "math"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

var gophersImage *ebiten.Image

type rasterEffectParts struct {
    image *ebiten.Image
    angle int
}

func (r *rasterEffectParts) Update() {
    // 角度を更新する。
    // なお、この更新方法で角速度が決まる。
    r.angle++
    r.angle %= 360
}

func (r *rasterEffectParts) Len() int {
    // 画像を横に細長い形に分割する。
    // パーツの高さは 1 ピクセルであり、数も高さのピクセル数と同じである。
    _, h := r.image.Size()
    return h
}
```

```
func (r *rasterEffectParts) Src(i int) (int, int, int, int) {
    // 描画元矩形を i に合わせて指定する。
    // 高さは 1 ピクセルであることに注意。
    w, _ := r.image.Size()
    return 0, i, w, i + 1
}

func (r *rasterEffectParts) Dst(i int) (int, int, int, int) {
    // 波形の振幅。
    const amplitude = 8
    w, _ := r.image.Size()
    // 波形の位相。
    // 位相の計算に i を使うことで、各行ごとに微妙に異なるズレを生じさせる。
    a := r.angle + 4*i
    // ズレ d を計算する。
    d := int(math.Floor(math.Sin(float64(a)*math.Pi/180) * amplitude))
    return d, i, w + d, i + 1
}

var parts *rasterEffectParts

func update(screen *ebiten.Image) error {
    parts.Update()
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // ImageParts として rasterEffectParts を指定する。
    op := &ebiten.DrawImageOptions{}
    op.ImageParts = parts
    screen.DrawImage(gophersImage, op)
    return nil
}

func main() {
    var err error
    gophersImage, _, err = ebitenutil.NewImageFromFile("./gophers_photo.jpg",
        ebiten.FilterNearest)
    if err != nil {
        log.Fatal(err)
    }
    parts = &rasterEffectParts{
        image: gophersImage,
    }
```

```
    if err := ebiten.Run(update, 320, 240, 2, "Raster Effect"); err != nil {
        log.Fatal(err)
    }
}
```

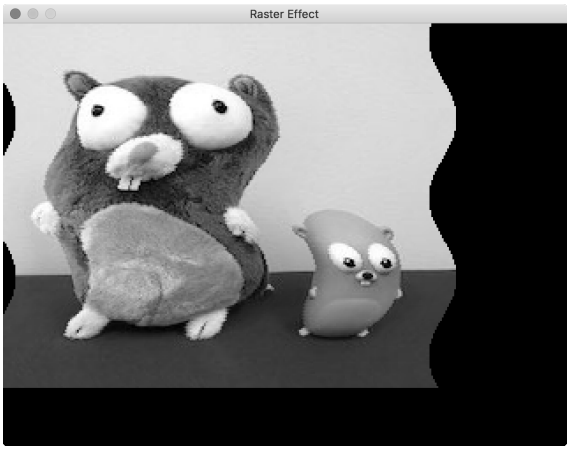


図 5.12　raster_effect.go の実行結果

写真がうねうねと波打ちます。画像を横に細長い棒状の集合とみなし、それぞれのパーツをちょっとずつずらして描画しました。

これを応用して、疑似 3D などを行うこともできます。

5.12　その他の機能

5.12.1　コンポジション

DrawImageOptions 構造体の CompositeMode を変更することで、加算合成、アルファ値によるマスク、XOR 合成などが可能になります。詳しくは API ドキュメントの CompositeMode の項 (https://godoc.org/github.com/hajimehoshi/ebiten#CompositeMode) を参考にしてください。

5.12.2　ピクセルの置き換え

ebiten.Image の ReplacePixels 関数で、その画像のピクセルをまるごと変えることが出来ます。

```
func (i *Image) ReplacePixels(p []uint8) error
```

指定する配列は R、G、B、A をそれぞれ 8bit 整数で表したプリマルチプライドアルファ形式のバイト列です。

ピクセル単位で操作した場合などに便利です。例えばこれでライフゲームが実装できます[9]。

この操作は非常に遅いので、毎フレーム呼び出すのは 1 回までにしておくのが無難です。

[9] 実際にライフゲームを実装したものとして、https://hajimehoshi.github.io/ebiten/examples/life.html があります。

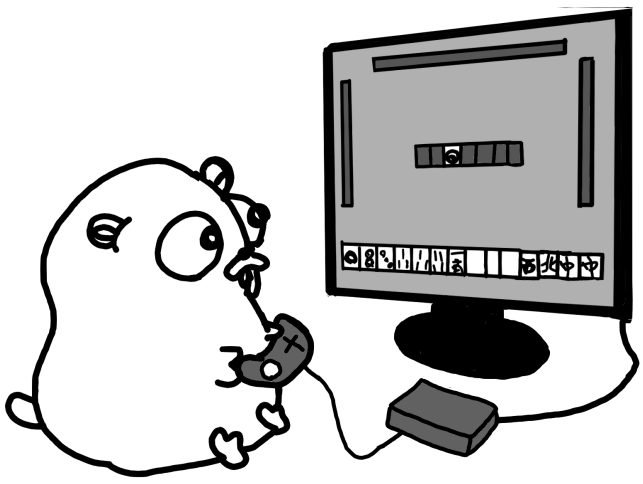
第 6 章

入力

この章ではユーザー入力をハンドルする方法について説明します。いままで描画方法について説明してきましたが、ユーザー入力を受け付けないため、ユーザーはただ画面を黙ってみているだけでした。入力をハンドルすることにより、インタラクティブなゲームを作ることが出来ます。

Ebiten の入力は、キーボード、マウス、ゲームパッド、タッチを扱うことが出来ます。

機能としては非常に単純で、基本的に「特定のキーが現在押されているかどうか」を判断する関数しかありません。



6.1　キーボード

キーボードのキーが押されているかどうかを判定するには、IsKeyPressed 関数を用います。

```
func IsKeyPressed(key Key) bool
```

Key 型は、キーボードのキーを表す定数の型です。たとえば A のキーに対応する定数は KeyA です。使用できるキーは以下のとおりです。

英字　KeyA、KeyB、…、KeyZ
数字　Key0、Key1、…、Key9
ファンクションキー　KeyF1、KeyF2、…、KeyF12
方向キー　KeyDown、KeyLeft、KeyRight、KeyUp
修飾キー　KeyAlt、KeyControl、KeyShift
その他　KeyBackspace、KeyCapsLock、KeyComma、KeyDelete、KeyEnd、KeyEnter、KeyEscape、KeyHome、KeyInsert、KeyPageDown、KeyPageUp、KeyPeriod、KeySpace、KeyTab

ポータビリティのためにいろいろな制約があります。たとえば、修飾キーは左右を区別しません。

6.1.1　入力テスト

実際にキー入力を判別するコードをみてみます。

```
// key.go

package main

import (
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)
```

```
func update(screen *ebiten.Image) error {
    // 入力は描画ではなく論理更新の範疇なので、
    // IsRunningSlowly を呼ぶ前に IsKeyPressed を呼ぶ。

    // A キーが押されたかどうかの判別を行う。
    if !ebiten.IsKeyPressed(ebiten.KeyA) {
        return nil
    }
    if ebiten.IsRunningSlowly() {
        return nil
    }
    ebitenutil.DebugPrint(screen, "A key is pressed!")
    return nil
}

func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Key"); err != nil {
        log.Fatal(err)
    }
}
```

図 6.1　key.go の実行結果

A キーを押したときにだけメッセージが出るはずです。

6.1.2　トリガー

前述のとおり、Ebiten には「現在キーが押されているかどうか」を判定する関数しかありません。「いままで押されていなかったが、今回新たに押されたかどうか」を判別する手段は用意されていないのです。

では「今はじめて押された」かどうかの判別はどうしたら良いでしょうか。どれくらい長い間そのキーが押されていたかを記録し、その値が 1 のときに「新しく押された」とみなすことで、新規入力を判別できます。

```
// key_trigger.go

package main

import (
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

// A キーの状態を表す整数値。
// 具体的には、どのくらい長い間押されたかを表す値。
var keyAState = 0

func update(screen *ebiten.Image) error {
    if ebiten.IsKeyPressed(ebiten.KeyA) {
        keyAState++
    } else {
        keyAState = 0
    }
    if ebiten.IsRunningSlowly() {
        return nil
    }
    if keyAState != 1 {
        return nil
    }
    // keyAState が 1 のときにのみ文字を描画する。
    ebitenutil.DebugPrint(screen, "A key is triggered!")
    return nil
}
```

```
func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Key Trigger"); err != nil {
        log.Fatal(err)
    }
}
```

A キーを押したときに一瞬だけ文字が出ます。

今回は 1 つのキーだけの判別を行いました。複数キーを使用したい場合は、キーごとに状態を持ち更新する必要があります。

6.2　マウス

Ebiten はマウスのようなポインティングデバイスを扱えます。なおトラックパッドも共通の API を使います。

マウスの現在座標を取得するには、ebiten.CursorPosition 関数を使います。

```
func CursorPosition() (x, y int)
```

座標は左上原点とした座標です。スケールの値に合わせて自動的に調整されるので、画面スケールの指定がどうであれゲームのコードを変える必要がありません。

またマウスのボタンが押されているかの判別には、ebiten.IsMouseButtonPressed を使います。

```
func IsMouseButtonPressed(mouseButton MouseButton) bool
```

引数の MouseButton 型の値として MouseButtonLeft、MouseButtonRight、MouseButtonMiddle があり、それぞれ左、右、中央ボタンを表します。

キーボードと同じく「現在押されているかどうか」しか判別できないため、新規に押されたかどうかを判別するためには、前述と同様の工夫が必要になります。

実際にマウスを使う例を見てみましょう。

```
// mouse.go

package main

import (
    "fmt"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

func update(screen *ebiten.Image) error {
    // 現在のマウスカーソル座標を取得する。
    x, y := ebiten.CursorPosition()
```

```
    msg := fmt.Sprintf("(%d, %d)\n", x, y)
    // マウス左ボタンが押されていたら、メッセージを追加する。
    if ebiten.IsMouseButtonPressed(ebiten.MouseButtonLeft) {
        msg += "Left button is pressed"
    }
    if ebiten.IsRunningSlowly() {
        return nil
    }
    ebitenutil.DebugPrint(screen, msg)
    return nil
}

func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Mouse"); err != nil {
        log.Fatal(err)
    }
}
```

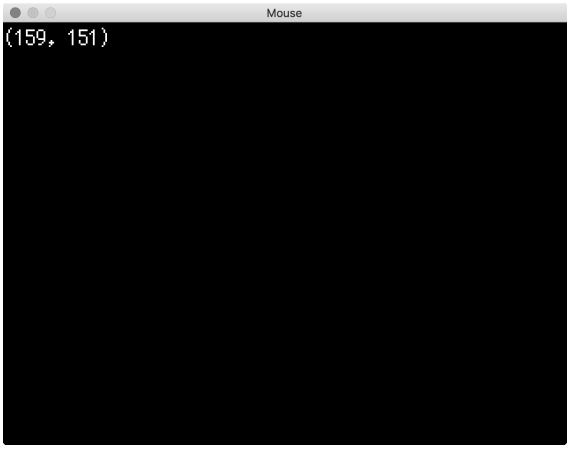


図 6.2　mouse.go の実行結果

6.3　ゲームパッド

Ebiten はゲームパッドを扱うことも出来ます。デスクトップはもちろん、ブラウザでも動作します[1]。また、複数のゲームパッドを処理することが出来ます。入力に使えるのは、ボタンの押下の有無、および軸の位置です。

ボタンを判別するには、IsGamepadButtonPressed 関数を使います。

```
func IsGamepadButtonPressed(id int, button GamepadButton) bool
```

id は 0 始まりのゲームパッドの番号を表します。Ebiten バージョン 1.5.0 時点ではゲームパッドの数の取得や、抜き差しタイミングによるハンドルは出来ません。

GamepadButton 型はゲームパッドのボタンを表す型で、GamepadButton0 から GamepadButton31 まで定義されています。関数の戻り値は、ボタンが押されていれば true、そうでなければ false です[2]。

軸の傾きを取得するには、GamepadAxis 関数を使います。また、軸の数を取得するには GamepadAxisNum 関数を使います。

```
func GamepadAxis(id int, axis int) float64
```

```
func GamepadAxisNum(id int) int
```

GamepadAxis 関数の戻り値の範囲は-1 から 1 です。id 引数は先程と同じくゲームパッドの番号を指定します。axis 引数は 0 始まりの軸の番号を取得します。

ゲームパッドおよび環境によって、物理ボタンに対応する GamepadButton の値は異なることに注意してください。たとえ同じゲームパッドを使っていても、ブラウザが違うだけでボタンが異なることが有り得ます。ボタン番号を仮定せず、ゲーム側でキーコンフィギュレーションを提供するのが望ましいでしょう。

---

[1] Gamepad API (https://www.w3.org/TR/gamepad/) に対応しているブラウザのみ対応。

[2] ボタンによっては、押されているかいないかという状態だけではなく、徐々に押されていくという状態を持つものがあります。PlayStation4 コントローラの L2/R2 ボタンなどがそうです。Web の Gamepad API では取得可能ですが、デスクトップ実装に使用している GLFW3.2 がそれに未対応なため、現在対応予定はありません。

ゲームパッド API を実際に使う例をみてみます。お手持ちのゲームパッドを指して実行してみてください。

```
// gamepad.go

package main

import (
    "fmt"
    "log"

    "github.com/hajimehoshi/ebiten"
    "github.com/hajimehoshi/ebiten/ebitenutil"
)

func update(screen *ebiten.Image) error {
    // ゲームパッド番号。今回は 0 番で決め打ちする。
    const gamepadID = 0
    // 現在押されているボタンを取得する。
    buttons := []ebiten.GamepadButton{}
    for b := ebiten.GamepadButton0; b <= ebiten.GamepadButtonMax; b++ {
        if ebiten.IsGamepadButtonPressed(gamepadID, b) {
            buttons = append(buttons, b)
        }
    }
    // 現在の軸の状態を取得する。
    axes := make([]float64, ebiten.GamepadAxisNum(gamepadID))
    for i := range axes {
        axes[i] = ebiten.GamepadAxis(gamepadID, i)
    }
    if ebiten.IsRunningSlowly() {
        return nil
    }
    // ボタンの状態を文字列化する。
    msg := "Buttons: "
    for _, b := range buttons {
        msg += fmt.Sprintf("%d ", b)
    }
    // 軸の状態を文字列化する。
    // 軸の傾きは-1 から 1 の浮動小数点で表される。
    msg += "\nAxes:\n"
    for _, a := range axes {
        msg += fmt.Sprintf("  %0.6f\n", a)
```

```
    }
    ebitenutil.DebugPrint(screen, msg)
    return nil
}

func main() {
    if err := ebiten.Run(update, 320, 240, 2, "Gamepad"); err != nil {
        log.Fatal(err)
    }
}
```

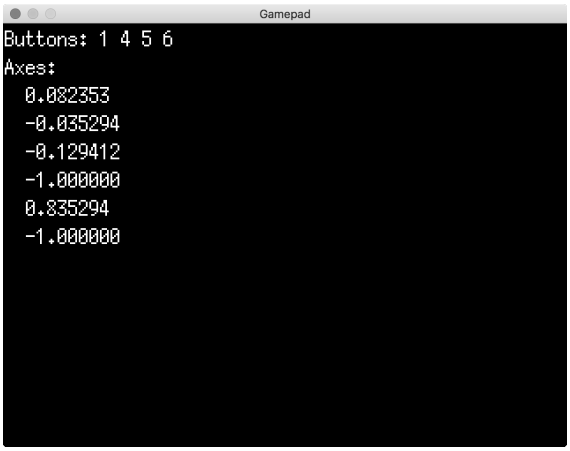


図 6.3　gamepad.go の実行結果。筆者は PlayStation4 のコントローラを用いた

6.4 タッチ

Ebiten はタッチも扱うことが出来ます。現在のタッチの状態は Touches 関数で取得します。

```
func Touches() []Touch
```

Touches 関数は Touch インターフェイスの配列を返します。タッチは 1 個だけではなく同時に複数存在することがあるからです。Touch インターフェイスの定義は次のとおりです。

```
type Touch interface {
    ID() int
    Position() (x, y int)
}
```

ID 関数はストロークを識別する整数を返します。指を画面に押してから離すまでの連続した動作がストロークです。Position 関数はそのストロークの現在の座標を返します。

著者紹介

星一 (ほしはじめ)

1985 年生まれ。ゲームを作ること、ゲームを作るための何かを作ることに興味がある。

URL https://star.one/
メールアドレス hajimehoshi@gmail.com

Ebiten
Go で作る 2D ゲーム

2017/04/09　初版第 1 刷発行 (技術書典 2)

サークル　star.one
著者　星一
イラスト　ぴぽごう
印刷　ちょ古っ都製本工房 (http://www.chokotto.jp/)
