TOMMASO VISCONTI
tommaso@develer.com

Game development with Go

develer

This workshop wants to give you inspiration to learn Go while also learning how to write games with it.

We'll see the Ebiten game library and its features and we'll stop on some Go core concepts along the way.

I divided the workshop into 3 parts, each part ends with a practical exercise.

We'll start with a theoretical part (~20 mins) then the exercise (~30 mins). The last 10 minutes will be used for Q&A + pause.

During exercises we'll have private or public chats/calls if you have blocking questions.

Code examples, assets and my version of the game can be found here:

https://github.com/tommyblue/golab-2020-go-game-development

AGENDA

- The game loop
- Introduction to Ebiten
- How to draw images
- Animations
- Spritesheets
- User input
- Music and sounds
- Fonts
- UI/UX and scenes

But first, how does a game work?

(simple introduction)

Game development has many well-known programming patterns

The most famous one is **the Game Loop**, that is the foundation of most games and frameworks

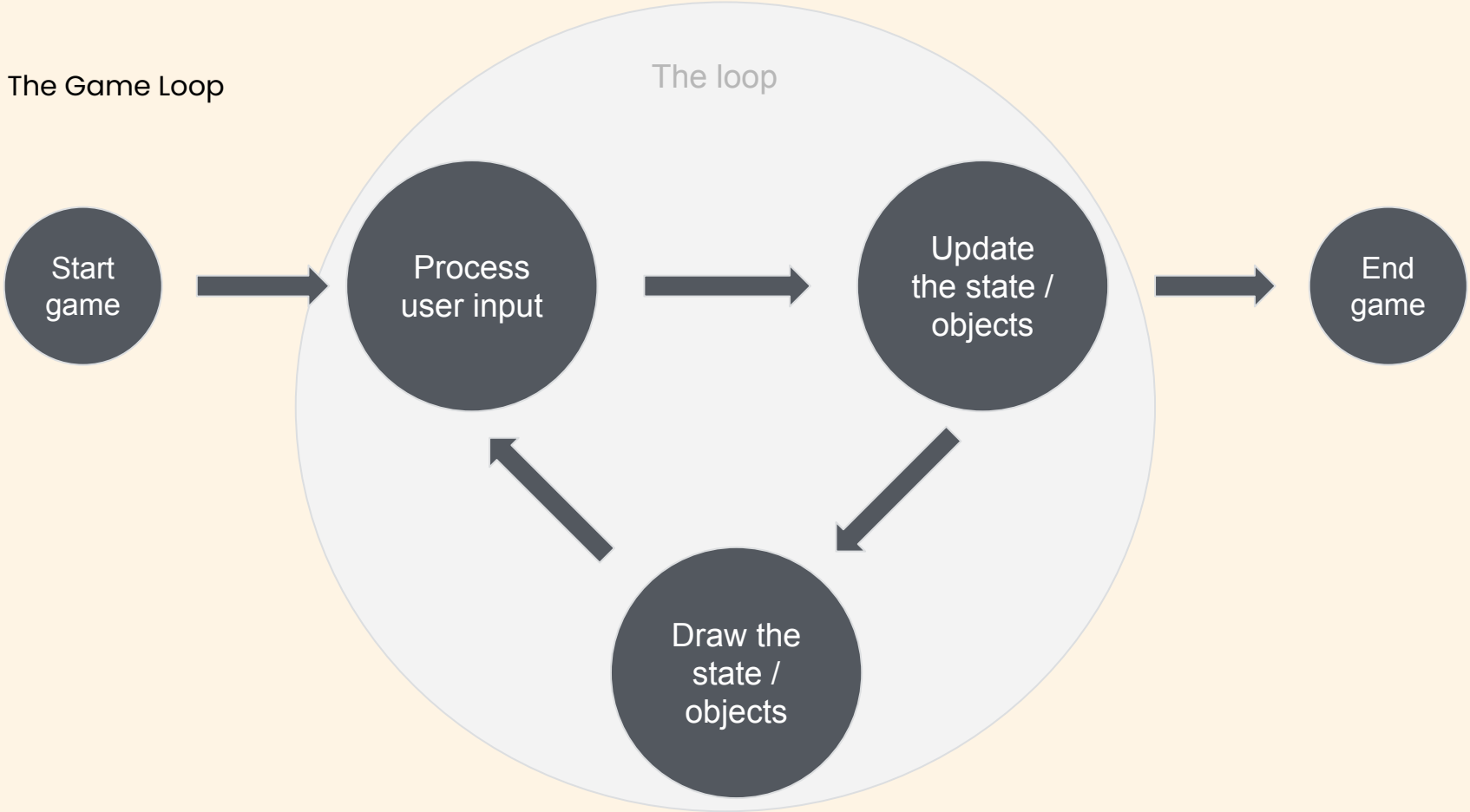
If you want to learn more about game patterns:
https://gameprogrammingpatterns.com/

As any other program, a game is a flow of code

A game must show something and interact with the user (keyboard, joystick, sound, etc)

The Game Loop is a simple but fundamental pattern to make a game work

The Game Loop

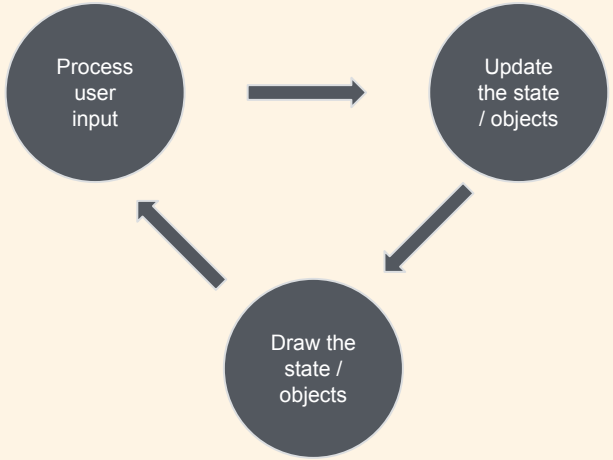


The Game Loop

Each game will try to run at 60 FPS (1 frame every 16.6 ms)

The main problem with the Game Loop is that the game speed depends on the underlying hardware. Fast computers will run faster games (not optimal for physics simulations :)

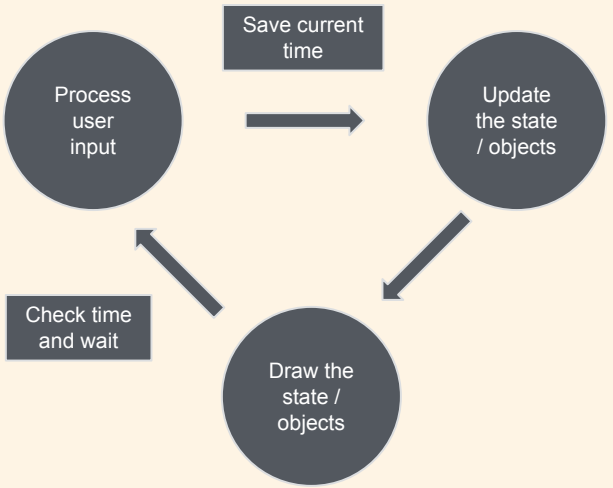
Old games was designed to know the HW speed and didn't work well on newer computers



1st solution: **add a delay** at the end of each loop to "wait" before the next cycle

Good solution for fast loops, but what for slow loops (>16.6 ms)?

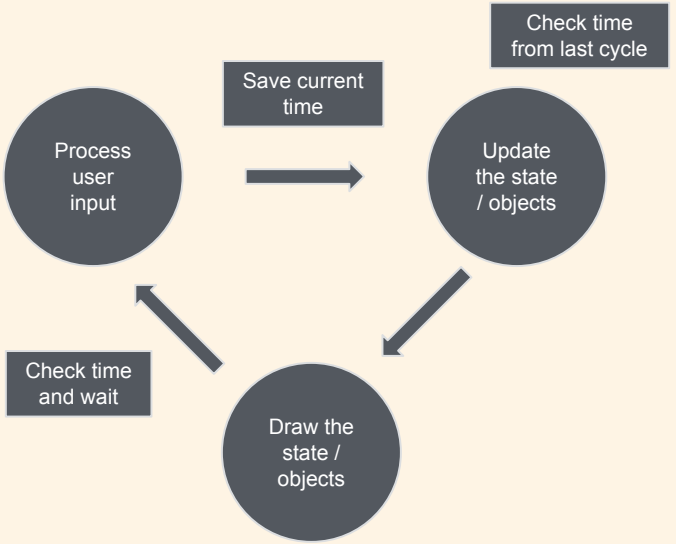
When the "sleep" time is below zero, it means that the game is too slow and then the game slows down



2nd solution: the **update** step knows how long is elapsed since the last loop and makes state calculation based on elapsed time

What happened in between “doesn’t happen”:

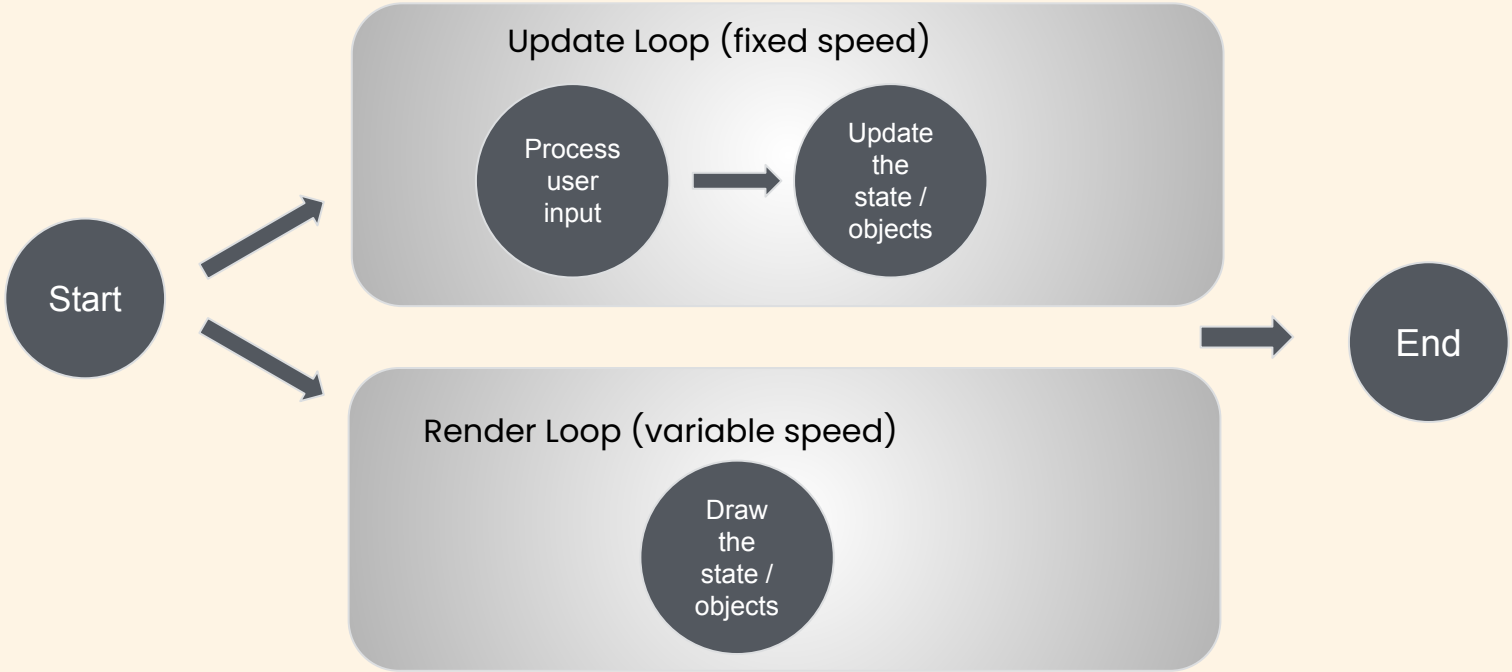
- a character not hitting a wall because the update step has been executed too late and the wall is below the character



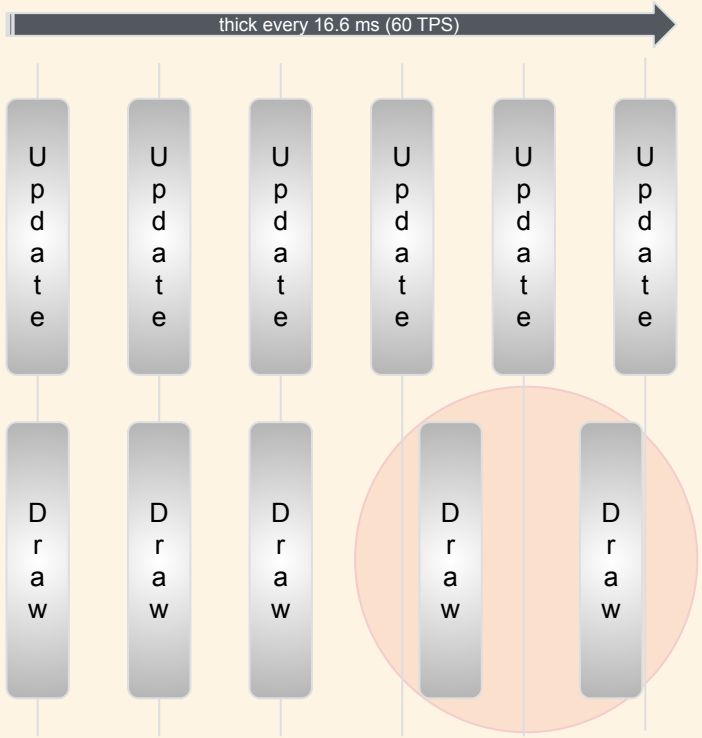
3rd solution (used by Ebiten):

the game logic (inputs + update) run in a separate loop at fixed speed (60 FPS)

The rendering process (draw) runs at its own speed



The Game Loop



When drawing is out-of-sync, we can see “glitches”, but they are only drawing glitches, the game logic is always correct.

Ebiten (/ebíten/)

Ebiten

A dead simple 2D game library for Go

Ebiten is a game library developed by Hajime Hoshi based on simplicity.

Everything is an image and most operations consist in drawing and moving images. API is simple and clear. Works on desktop, web, mobile.

🌏 https://ebiten.org

API Reference: https://pkg.go.dev/github.com/hajimehoshi/ebiten

Help: https://gophers.slack.com/app_redirect?channel=ebiten

Ebiten is a 2D game library, but even with 2D we can simulate a 3D world:

More info: http://clintbellanger.net/articles/isometric_math/

Ebiten uses the last described version of the game loop, with fixed updates (at 60 TPS*) and variable rendering speed.

*TPS (ticks per second) is different from FPS (frames per second), as well described by Hajime Hoshi:
“A frame represents a graphics update. This depends on the refresh rate on the user’s display. Then FPS might be 60, 70, 120, and so on. **This number is basically uncontrollable**. Ebiten can just turn on or off vsync. If vsync is turned off, Ebiten tries to update graphics as much as possible, then FPS can be 1000 or so.
A tick represents a logical update. TPS means how many times the update function is called per second. This is fixed as 60 by default.”

To run an Ebiten game, it's enough to implement a ebiten.Game interface and pass it to the ebiten.RunGame(*ebiten.Game) function:

```
package ebiten

type Game interface {
    Update(screen *Image) error
    // Draw(screen *Image) // Optional, thus not included in the interface
    Layout(outsideWidth, outsideHeight int) (int, int)
}

func RunGame(game Game) error {
    // ...
}
```

Go interfaces are named collections of method signatures.

Interfaces describe how an object can behave. Similar objects can have similar behaviours and then they can be described by the same interface.

Objects implement methods that are described by the interface.

To make an example, a superhero and a rocket can both fly. So they can TakeOff and Land, as well as returning their current altitude:

```
type FlyingObject interface {
   TakeOff() error
   Land() error
   Altitude() int
}
```

```go
type SuperHero struct {
   altitude int // field
}

func (s *SuperHero) TakeOff() error {
   if s.altitude != 0 {
       return fmt.Errorf("Already flying")
   }
   s.altitude = 10
   return nil
}
```

```go
func (s *SuperHero) Land() error {
   if s.altitude == 0 {
       return errors.New("Already landed")
   }
   s.altitude = 0
   return nil
}

// use a value receiver instead of a
// pointer receiver because it doesn't
// need to change the value
func (s SuperHero) Altitude() int {
   return s.altitude
}
```

```go
package main

import "fmt"

type FlyingObject interface {
   TakeOff() error
   Land() error
   Altitude() int
}

func main() {
   s := &SuperHero{}
   manageFly(s) // s is a *SuperHero that implements FlyingObject
   r := &Rocket{}
   manageFly(r) // same for the *Rocket
}

func manageFly(f FlyingObject) { // the f argument has the interface type
   f.TakeOff()
   fmt.Println("Altitude:", f.Altitude())
   f.Land()
}
```

Now let’s go back to Ebiten and the ebiten.Game interface and see how the “Hello, World” example works

“Hello, World!” with Ebiten

```go
package main

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

type Game struct{} // Game implements the ebiten.Game interface

func (g *Game) Update(screen *ebiten.Image) error {
    return nil
}

// Draw is optional, but suggested to maintain the logic of the Game Loop
func (g *Game) Draw(screen *ebiten.Image) {
    ebitenutil.DebugPrint(screen, "Hello, World!")
}

func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
    return outsideWidth, outsideHeight
}

func main() {
    ebiten.SetWindowSize(640, 480)
    ebiten.SetWindowTitle("Hello, World!")
    if err := ebiten.RunGame(&Game{}); err != nil {
        panic(err)
    }
}
```

“Hello, World!” with Ebiten

```go
package main

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

type Game struct{} // Game implements the ebiten.Game interface

func (g *Game) Update(screen *ebiten.Image) error {
    return nil
}

// Draw is optional, but suggested to maintain the logic of the Game Loop
func (g *Game) Draw(screen *ebiten.Image) {
    ebitenutil.DebugPrint(screen, "Hello, World!")
}

func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
    return outsideWidth, outsideHeight
}

func main() {
    ebiten.SetWindowSize(640, 480)
    ebiten.SetWindowTitle("Hello, World!")
    if err := ebiten.RunGame(&Game{}); err != nil {
        panic(err)
    }
}
```

“Hello, World!” with Ebiten

```go
package main

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

type Game struct{} // Game implements the ebiten.Game interface

func (g *Game) Update(screen *ebiten.Image) error {
    return nil
}

// Draw is optional, but suggested to maintain the logic of the Game Loop
func (g *Game) Draw(screen *ebiten.Image) {
    ebitenutil.DebugPrint(screen, "Hello, World!")
}

func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
    return outsideWidth, outsideHeight
}

func main() {
    ebiten.SetWindowSize(640, 480)
    ebiten.SetWindowTitle("Hello, World!")
    if err := ebiten.RunGame(&Game{}); err != nil {
        panic(err)
    }
}
```

“Hello, World!” with Ebiten

```go
package main

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

type Game struct{} // Game implements the ebiten.Game interface

func (g *Game) Update(screen *ebiten.Image) error {
    return nil
}

// Draw is optional, but suggested to maintain the logic of the Game Loop
func (g *Game) Draw(screen *ebiten.Image) {
    ebitenutil.DebugPrint(screen, "Hello, World!")
}

func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
    return outsideWidth, outsideHeight
}

func main() {
    ebiten.SetWindowSize(640, 480)
    ebiten.SetWindowTitle("Hello, World!")
    if err := ebiten.RunGame(&Game{}); err != nil {
        panic(err)
    }
}
```

Game interface

Update() function

```
func (g *Game) Update(screen *ebiten.Image) error {
    return nil
}
```

`Update()` updates the game logic by 1 tick (60 ticks per second)

Game interface

Draw() function

```
func (g *Game) Draw(screen *ebiten.Image) {
   ebitenutil.DebugPrint(screen, "Hello, World!")
}
```

`Draw()` draws the screen based on the current game state

Game interface

Layout() function

```
func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
    return outsideWidth, outsideHeight
}
```

`Layout()` gets the outside size (like the window size) and returns the game logical screen size

Can be fixed or can perform calculations to adapt the game to the user's device size

Images

In Ebiten **everything is an image** (starting from the screen) and what you'll always do is to draw images, one over the other

Images in Ebiten can be created in different ways:

- `ebiten.NewImage(width, height int, filter Filter) (*Image, error)`
- `ebiten.NewImageFromImage(source image.Image, filter Filter) (*Image, error)`
- `(*ebiten.Image).SubImage(r image.Rectangle) image.Image`
- `ebitenutil.NewImageFromFile(path string, filter ebiten.Filter) (*ebiten.Image, image.Image, error)`
- `ebitenutil.NewImageFromUrl(url string) (*ebiten.Image, error)`

`ebiten.Image` has a lot of useful methods, full list at

https://pkg.go.dev/github.com/hajimehoshi/ebiten#Image

The simplest thing you can do on an image is to fill it with a color:

```
screen.Fill(color.RGBA{0xff, 0, 0, 0xff})
```

An Image (a rectangle in this case) can be drawn over another with `DrawImage()`:

```
img, _ := ebiten.NewImage(100, 100, ebiten.FilterDefault)
img.Fill(color.RGBA{0, 0, 0xff, 0xff})
screen.DrawImage(img, nil)
```

The second argument of `DrawImage()` is a `*DrawImageOptions{}`

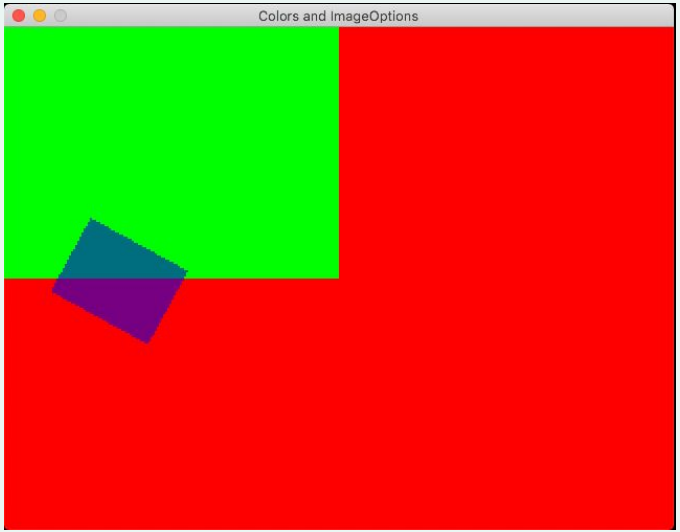
Image options can change color, geometry, composition and filtering of an image.

`GeoM` can be used to rotate, scale and move an image:

```
opts := &ebiten.DrawImageOptions{}
opts.GeoM.Translate(50, 100) // (0,0) is the top-left corner
opts.GeoM.Rotate(0.5) // rotate by radians
opts.GeoM.Scale(0.5, 0.5) // Scale matrix by
screen.DrawImage(img, opts)
```

GeoM functions: https://pkg.go.dev/github.com/hajimehoshi/ebiten#GeoM

What we've seen so far can be used to draw the image below:



https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/01_colors_and_image_options

Let’s draw a static image from a png file, a coin:

To load the image, there are multiple options. The most portable one is to save the image as a byte slice with file2byteslice*:

```
file2byteslice -input ./coin.png  -output assets.go -package main -var coinImg
```

The command above will generate the assets.go file:

```
package main

var coinImg = []byte("...")
```

*https://github.com/hajimehoshi/file2byteslice

Once the assets have been generated, the image can be created during initialization:

```
import _ "image/png"
var coin *ebiten.Image

func init() {
   img, _, _ := image.Decode(bytes.NewReader(coinImg))
   coin, _ = ebiten.NewImageFromImage(img, ebiten.FilterDefault)
}
```

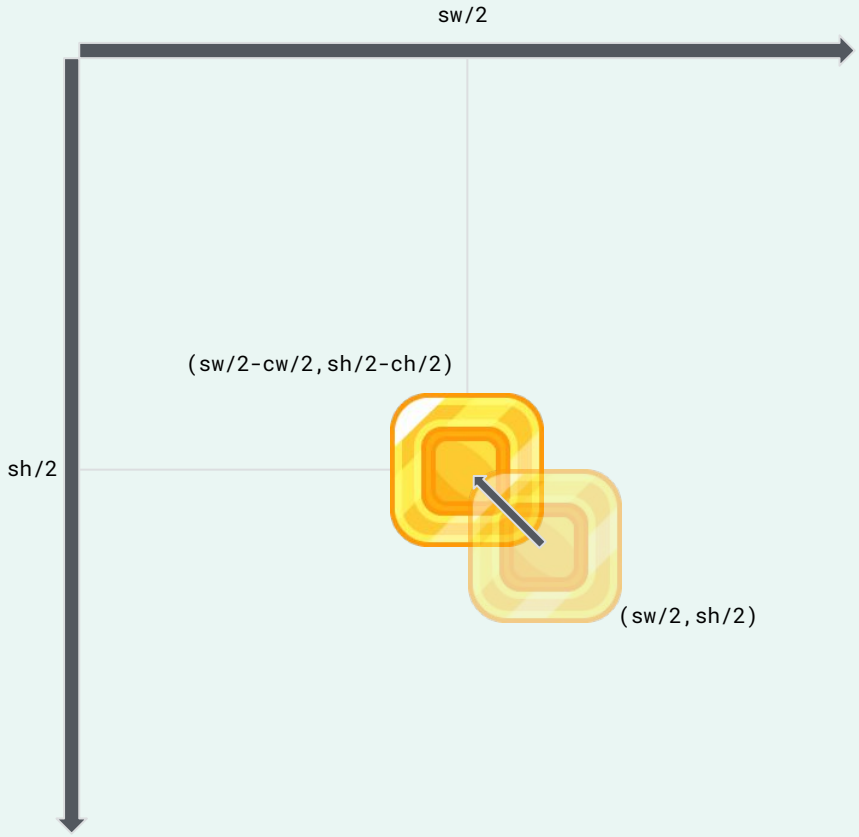
Depending on the image format, the correct decoder must be imported

The Draw function just moves the image to the center of the screen:

```
func (g *Game) Draw(screen *ebiten.Image) {
    op := &ebiten.DrawImageOptions{}
    cw, ch := coin.Size()
    sw, sh := screen.Size()
    // Move half of the screen size on the right/bottom and
    // half of the image size on the left/top
    op.GeoM.Translate(float64(sw/2 - cw/2), float64(sh/2 - ch/2))
    screen.DrawImage(coin, op)
}
```

Ebiten

Images from files



The result is an image like this one:

https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/02_images

To simplify/automate the generation process we can use go **generators**, an easy way to generate go files.

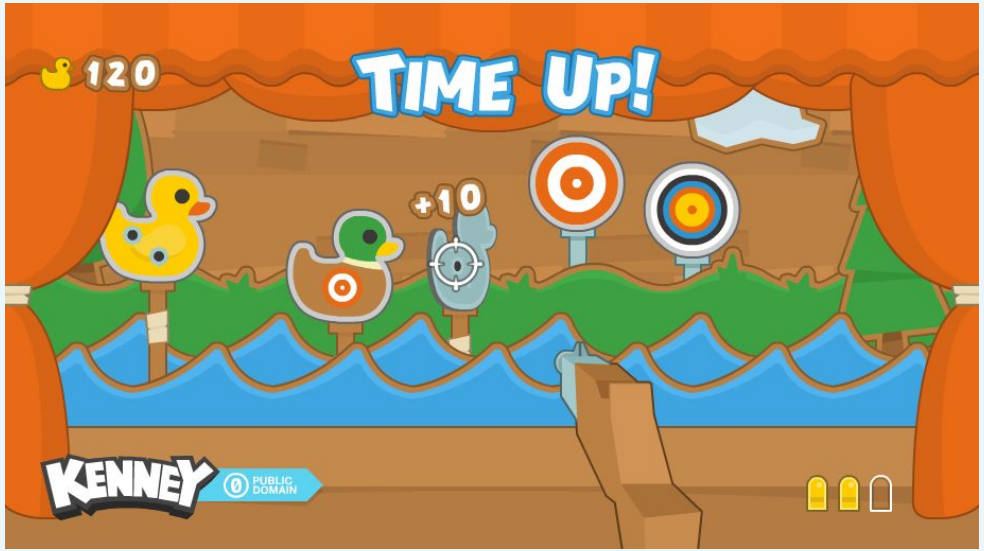
Create a `generate.go` file with this content (more lines can be added):

```
//go:generate file2byteslice -input ./coin.png  -output assets.go -package main -var coinImg
package main
```

Running **go generate .** will execute the commands in the file.

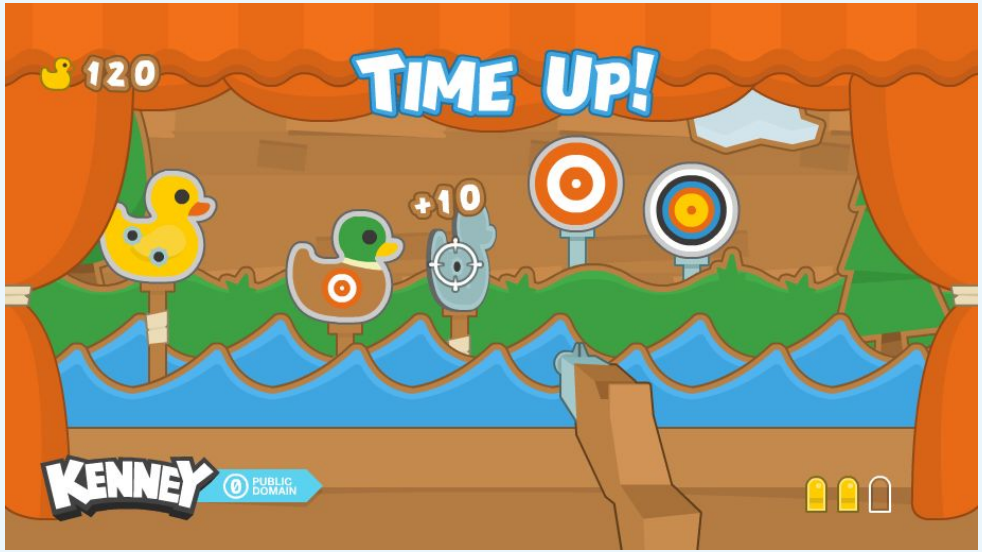
Exercise n.1

During the workshop you'll build a shooter game like this one:



What do we have here:

- Background, curtains and desk are **static images**
- Waves **move** in a wobbling way (right-left and up-down)
- **Ducks** appear from left and go right
- The **crosshair** shows the mouse position, left-click pulls the trigger
- **Hit** ducks gives 10 points, the **score** is shown
- Background **music** and shoot hit/miss **sounds**

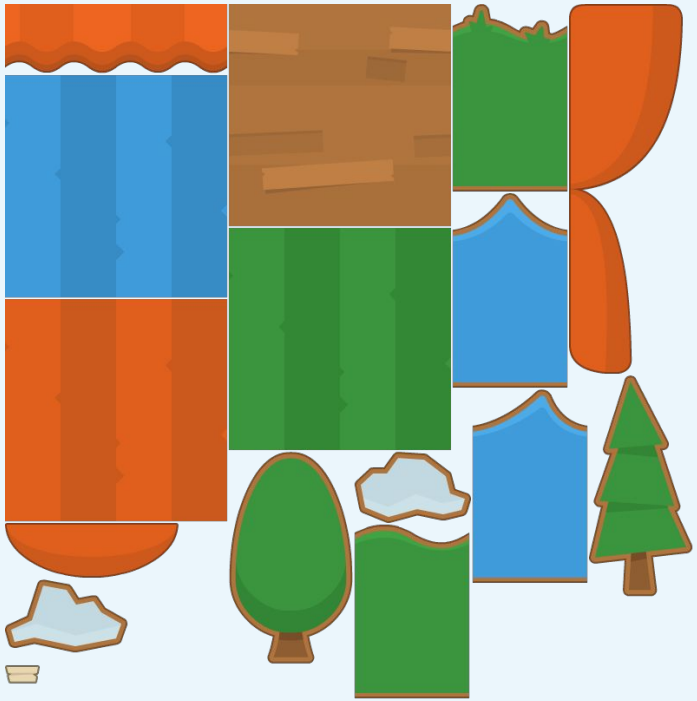


First exercise
Static images

In the repository you'll find an assets folder/package with the images both as single files or spritesheets with json file specs. Let's start with single images.

The **assets/PNG/Stall/** folder contains what you need to build the stage:

- **bg_green.png** for the background
- **curtain.png** for the right/left curtains
- **curtain_straight.png** for the top curtain
- **bg_wood.png** for the desk



First exercise
Static images

The result

Things to note

- background, desk and top curtain images are not big enough to fill the screen. They must be repeated many times (desk and curtain only in x, background both x and y). You need to calculate the amount of times to repeat to fill in the screen
- The curtain on the right is a mirrored image: `op.GeoM.Scale(-1,1)`
- I added a little brown border in the desk. It's just a rectangle. **Extra**: if you want, you can add a small shadow below it (play with black rectangles on 1px height and change the transparency)
- You can use only the `Draw()` function to do this

Spare time? How did you organize the code? Can you improve it?

Some ideas:

- Images can be represented by an Object interface with the Update and Draw functions and the Game can hold a list of objects (the order is important!) calling Update and Draw of all objects without knowing the type of each object
- Each object can have its own constructor, called by the main game constructor at startup
- Common logic should be shared between objects