TOMMASO VISCONTI
tommaso@develer.com

Sviluppo di videogiochi in Go

develer

Durante il webinar vedremo insieme come, usando la libreria Ebiten, sia possibile sviluppare videogiochi 2D usando Go

È un modo interessante per imparare le basi del linguaggio e divertirsi, evitando la “solita” ToDo app

Il webinar sarà diviso in 3 parti.

In ognuna vedremo alcuni concetti di Ebiten e poi illustrerò come usare quanto spiegato per l'esercizio proposto (che potrete fare dopo il webinar)

Gli esempi, la mia versione del gioco e gli assets li potete trovare nel repository dedicato:

https://github.com/develersrl/webinar-go-game-development

Il videogioco che vi proporrò di sviluppare è un semplice punta-e-spara, come si vede da questa immagine:

AGENDA

- Introduzione a Ebiten
- Disegnare immagini
- Animazioni
- Spritesheets
- Input dall'utente
- Musica e suoni
- Fonts
- UI/UX e scene

Come funziona un gioco?

(introduzione semplice)

Un fondamentale pattern di programmazione nel mondo dei videogiochi è il **Game Loop**

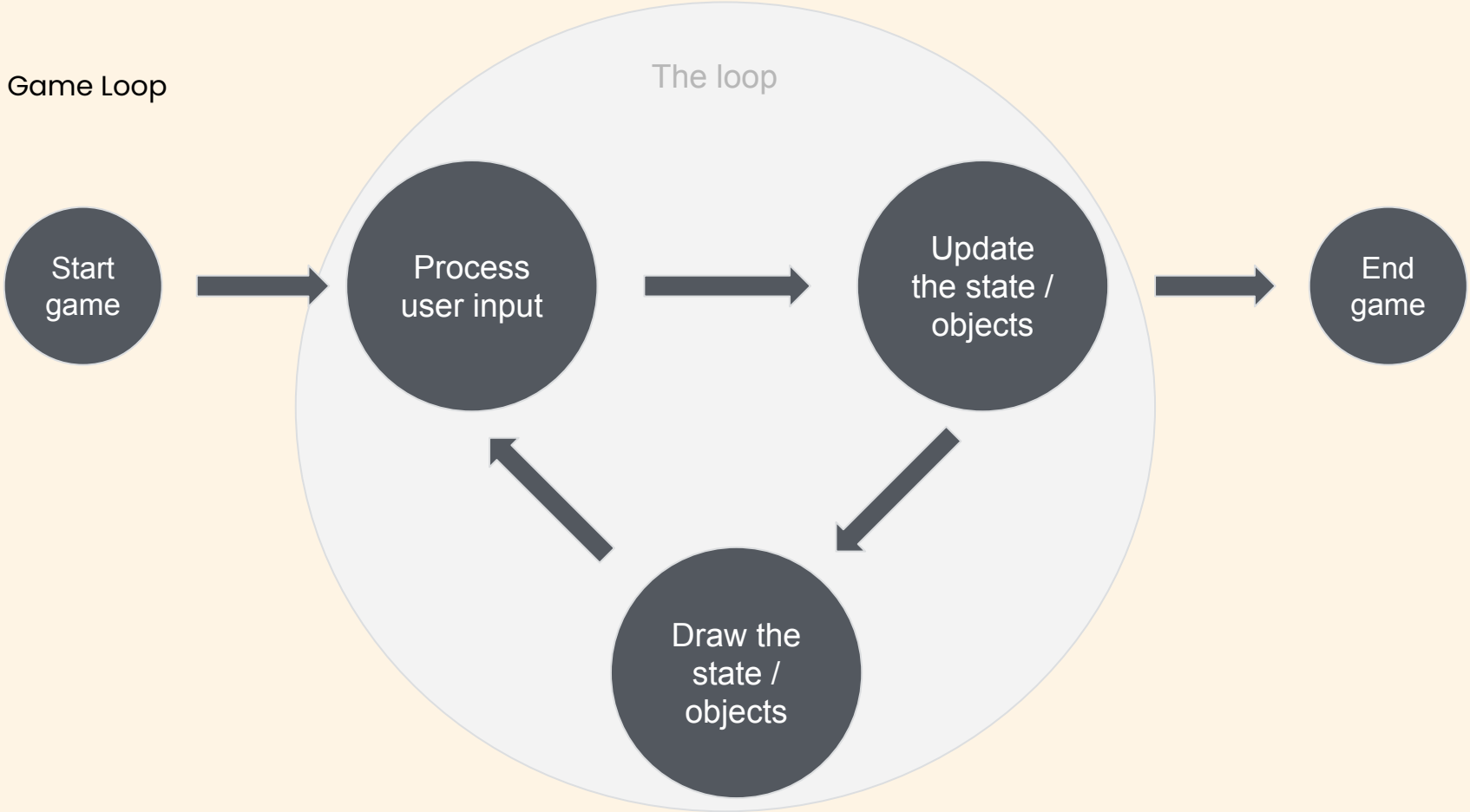
Per approfondimenti su altri pattern:
https://gameprogrammingpatterns.com/

Un gioco, come un qualsiasi altro programma, è un flusso di codice

Un gioco deve mostrare qualcosa e interagire con l'utente (tastiera, mouse, suoni, ecc)

Il Game Loop è un pattern di base per il funzionamento dei videogiochi

Game Loop

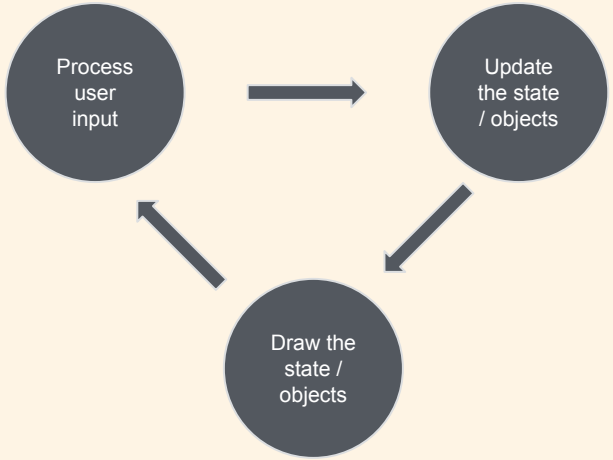

The loop
Start game
Process user input
Update the state / objects
End game
Draw the state / objects

Game Loop

I giochi cercano di mantenere almeno 60 FPS (1 frame ogni 16.6 ms)

La versione di base del Game Loop ha un problema: la velocità di gioco è dipendente da quanto gira velocemente il loop, il che, a sua volta, dipende dalla velocità del computer su cui viene eseguito.

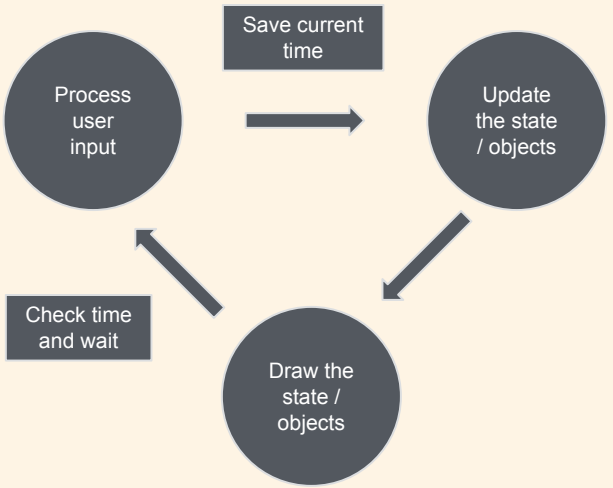
L'obiettivo è un'esperienza di gioco organica per tutti i giocatori



1° soluzione: **aggiungere un ritardo** al termine del loop prima di iniziare un nuovo giro

Ottimo per loop che girano troppo veloci, ma non per quelli lenti (>16.6 ms)

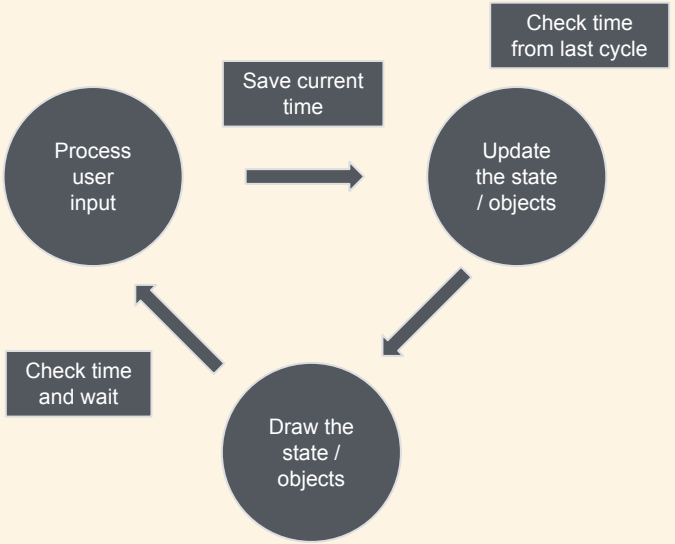
Se lo “sleep” è 0, allora il gioco deve per forza rallentare



2° soluzione: lo step di **update** conserva internamente il tempo passato dall'ultimo update e calcola lo stato in base a quello

Questo ha conseguenze terribili, se il loop è lento, quel che accade durante, "non esiste":

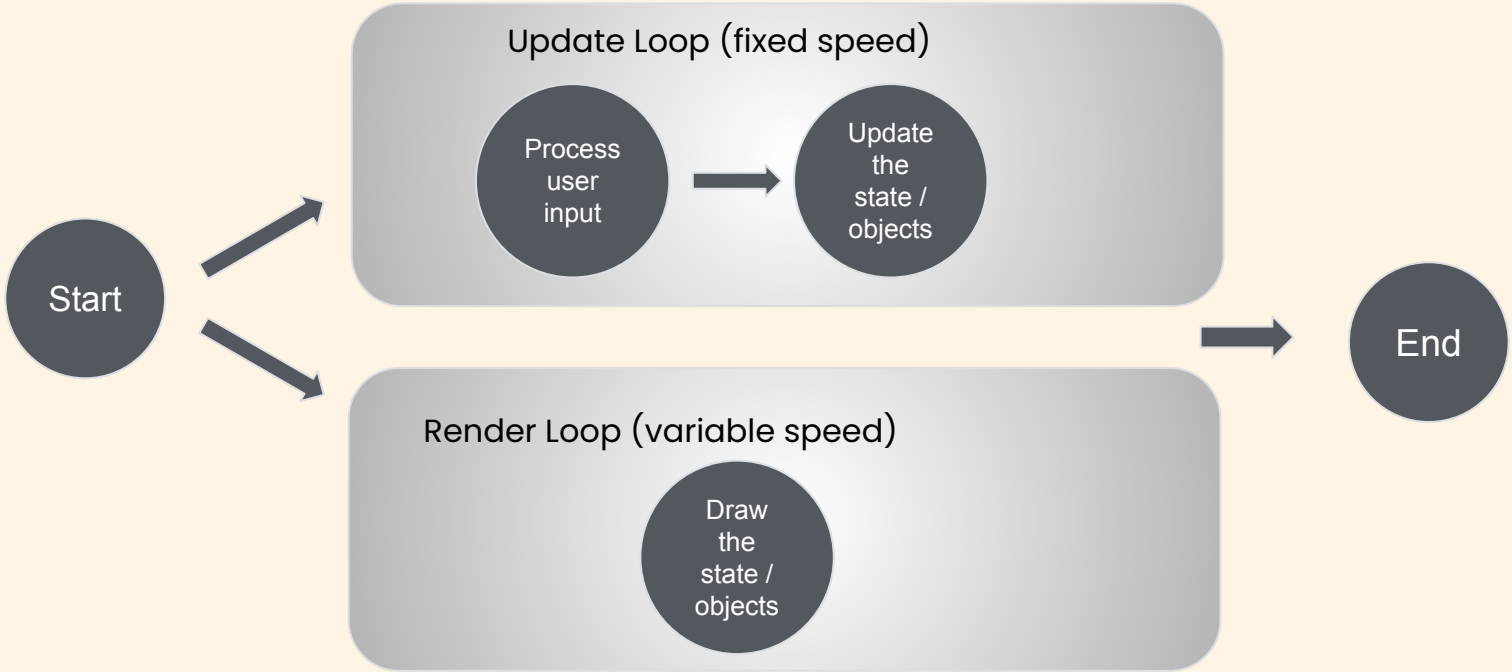
- ad esempio un personaggio può passare attraverso muri o una pallottola attraversa gli obiettivi senza colpirli



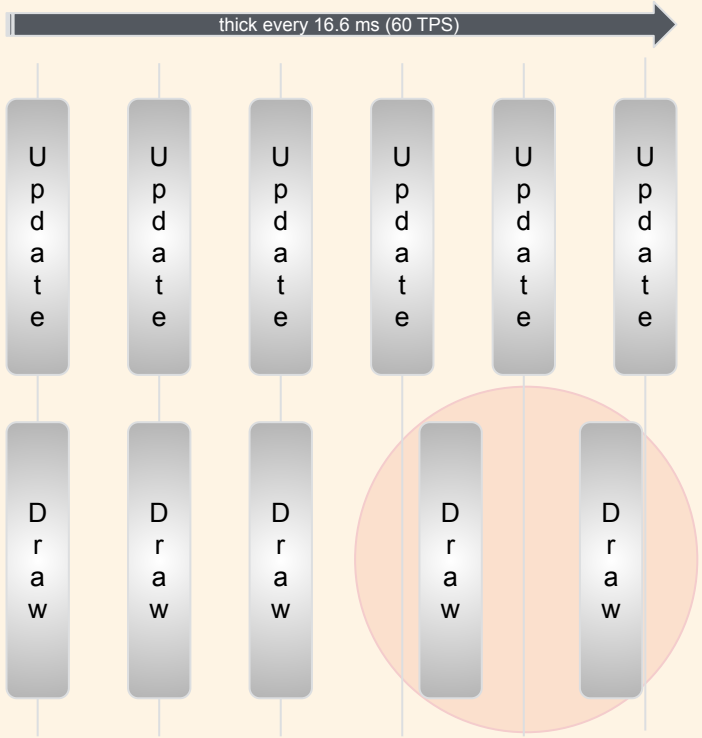
3° soluzione (usata anche da Ebiten):

la logica (input + update) viene eseguita in un loop a **velocità fissa** (60 FPS)

Il processo di rendering (draw) è in un loop a parte e **cerca** di mantenere i 60 FPS



Game Loop



Se il rendering è out-of-sync perché in ritardo, possono apparire delle aberrazioni grafiche, ma non intaccano la logica e lo stato

Ebiten (/ebíten/)

Ebiten

A dead simple 2D game library for Go

Ebiten è una libreria sviluppata da Hajime Hoshi che fa della semplicità il suo obiettivo principale.

In Ebiten quasi tutto è una “immagine” e la gran parte delle operazioni consiste nel disegnare e trasformare immagini. Ha un’API semplice.
È multipiattaforma: 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

Attenzione: disegnare in 2D non vuol dire poter sviluppare solo giochi “piatti”:

Info: http://clintbellanger.net/articles/isometric_math/

Per realizzare un gioco con Ebiten è sufficiente implementare l'interfaccia ebiten.Game e passare l'oggetto a ebiten.RunGame(*ebiten.Game):

```
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 {
    // ...
}
```

Vediamo in dettaglio l’esempio “Hello, World” che potete trovare anche sul sito di Ebiten

“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()

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

`Update()` avanza lo stato del gioco di 1 tick (60 ticks al secondo)

NON DISEGNA NULLA

Game interface
Draw()

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

`Draw()` disegna gli oggetti nello schermo, basandosi sullo stato del gioco

Non dovrebbe cambiare lo stato

Game interface
Layout()

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

`Layout()` riceve le dimensioni esterne (tipicamente la finestra di gioco) e ritorna la dimensione logica del gioco

Può ritornare un valore fisso o può adattarlo ad eventuali cambiamenti nel layout, basati sul device usato dall'utente

Immagini

In Ebiten **tutto è un'immagine** (a partire dallo "schermo") e l'operazione tipica è quella di disegnare immagini una sull'altra.

Esistono molte funzioni per creare immagini:

- `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)`

Per uno sguardo approfondito a `ebiten.Image` si faccia riferimento all'API:

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

Iniziamo con l'operazione più semplice, riempire un'immagine (in questo caso lo schermo intero) con un colore:

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

Creiamo adesso un'immagine rettangolare, riempiamola di colore e disegniamola sopra l'immagine dello schermo con `DrawImage()`:

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

Il secondo argomento di `DrawImage()` è `*DrawImageOptions{}`

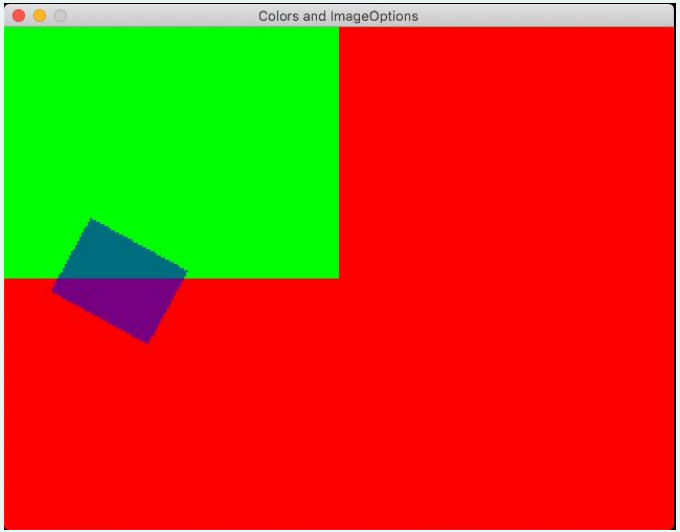
che possono trasformare le immagini cambiando colore, geometria, ecc.

Ad esempio `GeoM` permette di ruotare, scalare e muovere un'immagine:

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

Con quanto visto finora è possibile disegnare:



https://github.com/develersrl/webinar-go-game-development/tree/master/examples/01_colors_and_image_options

Disegniamo adesso un’immagine partendo da un file png:

Esistono molte opzioni per caricare un'immagine. Quella suggerita da Ebiten, per aumentare la portabilità, è quella di usare file2byteslice*:

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

Il file `assets.go` generato dal comando:

```
package main

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

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

A questo punto si può creare un'immagine Ebiten con:

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

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

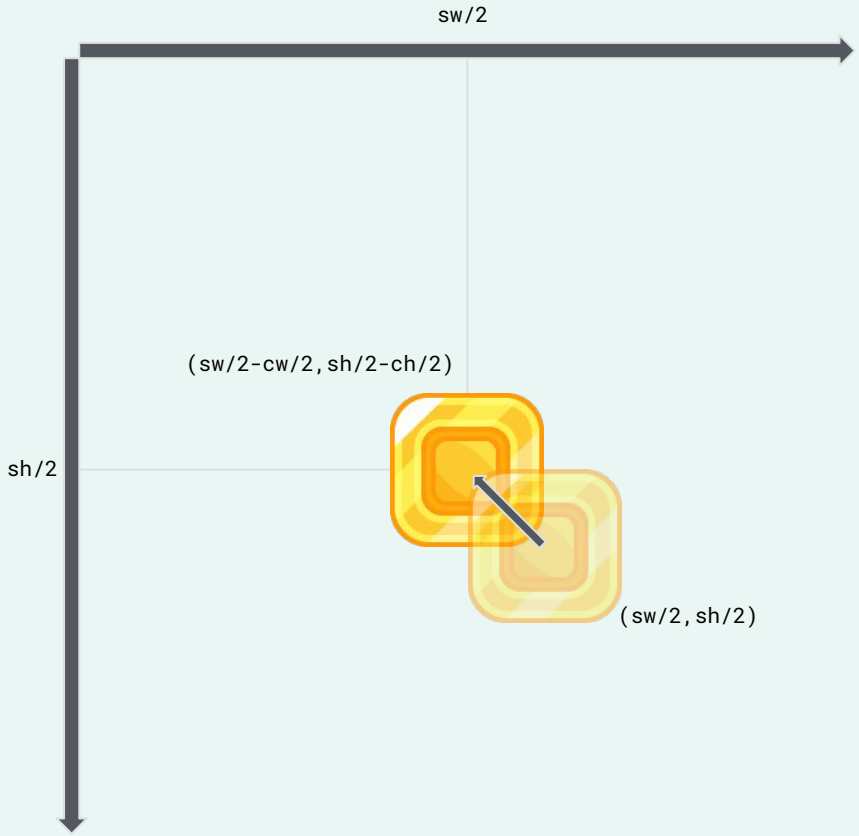
È necessario importare il decoder giusto per il formato di immagine

Usiamo, come in precedenza, `DrawImage()`, ma prima spostiamo la moneta al centro dello schermo:

```
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

Immagini da file



Il risultato sarà il seguente:

https://github.com/develersrl/webinar-go-game-development/tree/master/examples/02_images

Per semplificare e automatizzare la creazione degli asset vi consiglio l'uso dei **generatori** Go.

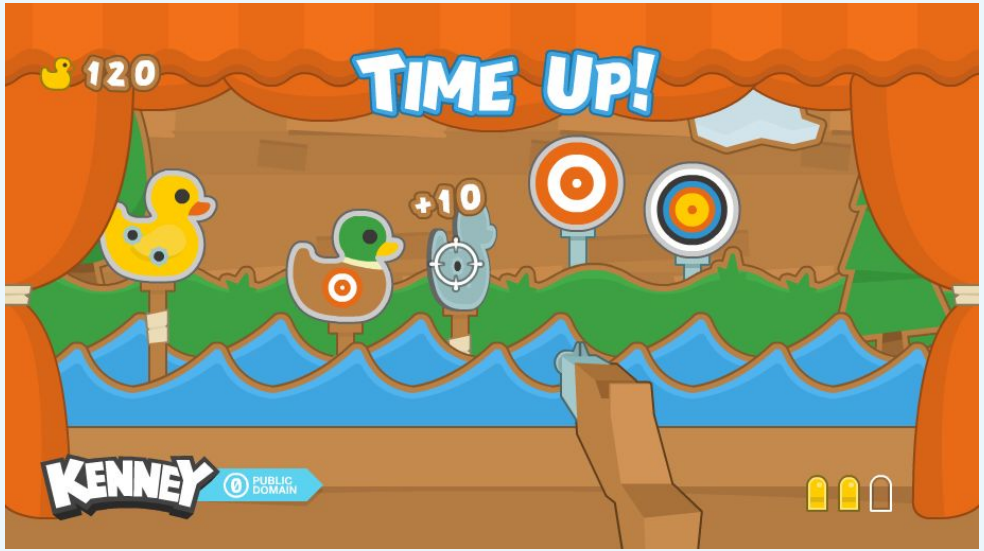
Basta creare un file `generate.go` con il comando (o i comandi):

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

Lanciando `**go generate .**` verranno eseguiti tutti i comandi

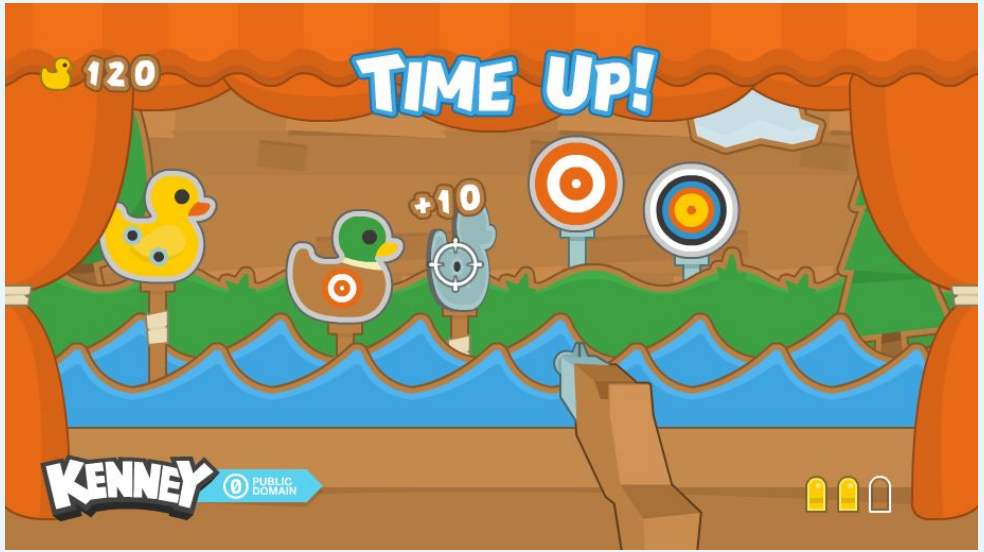
Esercizio n.1

Iniziamo a realizzare il videogioco:



Vediamo le varie parti:

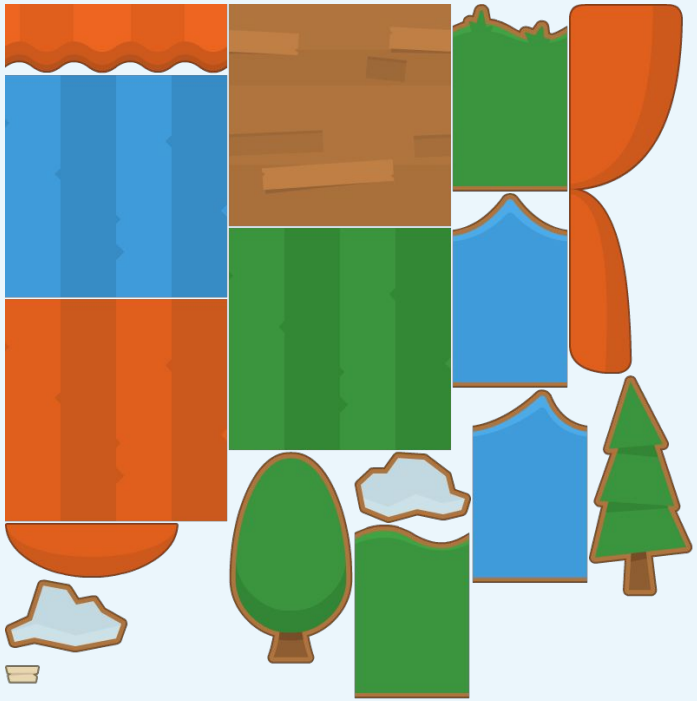
- Sfondo, tende e bancone sono **immagini statiche**
- Onde e papere si **muovono** in varie direzioni
- **Le papere** entrano a sinistra e vanno verso destra
- Il **mirino** mostra il mouse, si spara col click
- **Colpire** una papera dà 10 punti
- Viene mostrato il **punteggio**
- C'è una **musica** di sottofondo e i **suoni** dello sparo



Nel repository trovate una cartella **assets** con tutte le immagini, sia come file singoli che come spritesheets.

Iniziamo con la cartella **assets/PNG/Stall/** che contiene il necessario per questo esercizio:

- **bg_green.png** per lo sfondo
- **curtain.png** per le tende verticali
- **curtain_straight.png** per la tenda in alto
- **bg_wood.png** per il bancone



Primo esercizio
Immagini statiche

Il risultato

Cose su cui fare attenzione

- Le immagini dello sfondo, del bancone e della tenda in alto non sono abbastanza grandi per riempire tutto lo spazio ma sono fatte apposta per essere messe una accanto all'altra senza interruzioni. Bisogna calcolare quante volte ripeterle.
- La tenda a destra è specchiata rispetto a sinistra: `op.GeoM.Scale(-1,1)`
- Sopra al bancone ho creato un bordo. **Extra**: È possibile simulare un'ombra con rettangoli neri ripetuti e la trasparenza
- Il tutto si può realizzare solo con `Draw()`

Animazioni

Un’immagine animata è una sequenza di immagini statiche (frame).
Un metodo comune per creare animazioni è quella di usare delle spritesheet che contengono la sequenza di immagini e mostrare un’immagine alla volta

Realizziamo un'animazione introducendo uno "stato": un ticker che definisce a che punto del gioco siamo, avanzando di 1 ad ogni frame:

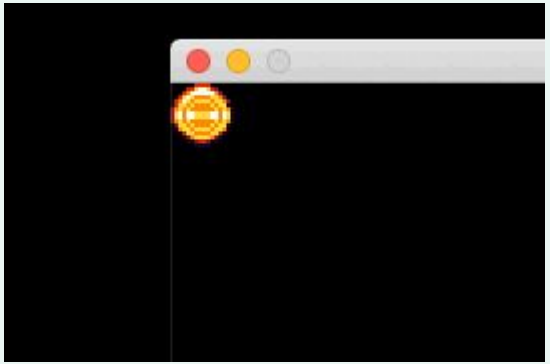
```
type Game struct {
   tick  uint64
}
func (g *Game) Update(screen *ebiten.Image) error {
   g.tick++
   return nil
}
```

Usando immagini di dimensioni fisse, per ogni frame la funzione `Draw()` deve disegnare una sotto-immagine muovendosi di una dimensione fissa, per poi ricominciare al termine del loop:

La funzione `Draw()` usa il ticker per calcolare il frame da visualizzare e quindi estrae dalla sprite il rettangolo corrispondente. `SubImage()` ritorna una interfaccia `image.Image` che necessita **type assertion** a `*ebiten.Image` per essere disegnata

```
const (
    imgSize   = 16 // size in pixels, square img
    numFrames = 8 // number of frames in the spreadsheet
)
func (g *Game) Draw(screen *ebiten.Image) {
    op := &ebiten.DrawImageOptions{}
    frameNum := g.tick % numFrames
    // move right in the spreadsheet
    frameX := int(frameNum * imgSize)
    rect := image.Rect(frameX, 0, frameX+imgSize, imgSize)
    subImg := coins.SubImage(rect)
    screen.DrawImage(subImg.(*ebiten.Image), op)
}
```

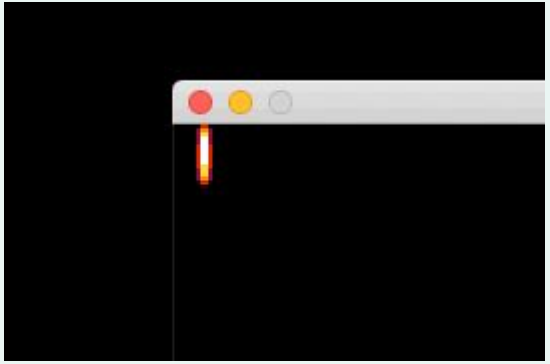
Adesso l'animazione è funzionante, ma ad una velocità eccessiva dato che il tick viene aumentato ~60 volte al secondo (la velocità di `Update()`):



Introduciamo quindi la variabile **velocità**:

```
type Game struct {
    tick  float64
    speed float64
}
func (g *Game) Draw(screen *ebiten.Image) {
    // ...
    frameNum := int(g.tick/g.speed) % numFrames
    // ...
}
```

Non resta quindi che calcolare il numero di frame che vogliamo visualizzare in 1 secondo N, con la formula **speed=FPS/N**, nel nostro caso **speed = 60/6 = 10**:



`https://github.com/develersrl/webinar-go-game-development/tree/master/examples/03_tiles_fixed_size`

Spritesheets

Ebiten

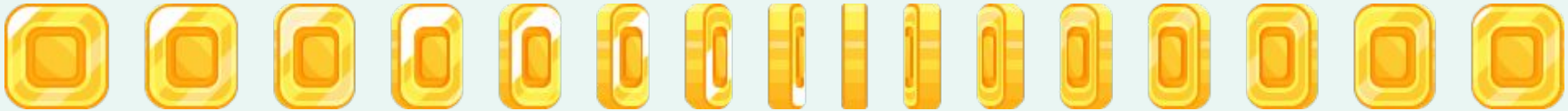
Spritesheets

Le spritesheet sono grandi immagini che sono formate da tante immagini, usate singolarmente o come parte di un’animazione.

Lo scopo è quello di ridurre la dimensione finale degli assets, ma così non si sa più dove sono le immagini che ci servono.

La conseguenza di avere un'animazione sparsa in una sprite è che ogni frame può avere dimensioni diverse (non, come prima, immagini tutte uguali).

Vediamo con un esempio come realizzare un'animazione di questa moneta:



https://github.com/develersrl/webinar-go-game-development/tree/master/examples/04_tiles_vars

Con le sprite, di solito*, si riceve anche un file (es. JSON) che definisce le coordinate di ogni sotto-immagine all'interno della sprite.

Con queste coordinate si possono definire le `image.Rect` necessarie a calcolare le `SubImage`.

*altrimenti te lo devi costruire (non te lo auguro) 😢

Questo è un esempio JSON di coordinate di una sprite:

```
{"frames": [
   { "x": 0, "y": 0, "w": 64, "h": 64 },
   { "x": 86, "y": 0, "w": 57, "h": 64 },
   { "x": 165, "y": 0, "w": 50, "h": 64 },
   ...
]}
```

Vediamo come “mappare” il file JSON su oggetti Go:

```go
type framesSpec struct {
    Frames []frameSpec `json:"frames"`
}

type frameSpec struct {
    X int `json:"x"`
    Y int `json:"y"`
    W int `json:"w"`
    H int `json:"h"`
}
```

```json
{"frames": [
    { "x": 0, "y": 0, "w": 64, "h": 64 },
    { "x": 86, "y": 0, "w": 57, "h": 64 },
    { "x": 165, "y": 0, "w": 50, "h": 64 },
    ...
]}
```

I frame vengono associati al nostro oggetto Game:

```
type Game struct {
    tick      float64
    speed     float64
    frames    []frameSpec
    numFrames int
}
```

Nota inserire i frame e il loro numero in Game è una semplificazione per l'esempio, a seconda dei bisogni ci saranno strutture più o meno complesse in cui gestirli

Definiamo una funzione buildFrames() per parsare il JSON:

```
func (g *Game) buildFrames(path string) error {
    j, _ := ioutil.ReadFile(path)
    fSpec := &framesSpec{}
    json.Unmarshal(j, fSpec)
    g.frames = fSpec.Frames
    g.numFrames = len(g.frames)
    return nil
}
```

main() si aspetta di ricevere come argomento il path al file JSON da passare a buildFrames():

```
func main() {
    if len(os.Args) < 2 {
        log.Fatal("missing json file arg")
    }
    g := &Game{}
    g.buildFrames(os.Args[1])
    ebiten.RunGame(g)
}
```

Draw() calcola il frame da disegnare e usa le coordinate per ritagliarlo:

```
func (g *Game) Draw(screen *ebiten.Image) {
   frameNum := int(g.tick/g.speed) % g.numFrames
   f := g.frames[frameNum]
   rect := image.Rect(f.X, f.Y, f.X+f.W, f.Y+f.H)
   subImg := coins.SubImage(rect).(*ebiten.Image)
   screen.DrawImage(subImg, &ebiten.DrawImageOptions{})
}
```

L’animazione funziona, ma ecco l’effetto di avere immagini di dimensioni diverse:

Per funzionare correttamente, ogni frame va centrato rispetto allo stesso punto (in questo caso il centro dello schermo):

```
x, y := screen.Size()
tx := x/2 - f.W/2
ty := y/2 - f.H/2
op := &ebiten.DrawImageOptions{}
op.GeoM.Translate(float64(tx), float64(ty))
```

Ed ecco il risultato, centrato rispetto allo schermo:

https://github.com/develersrl/webinar-go-game-development/tree/master/examples/04_tiles_vars

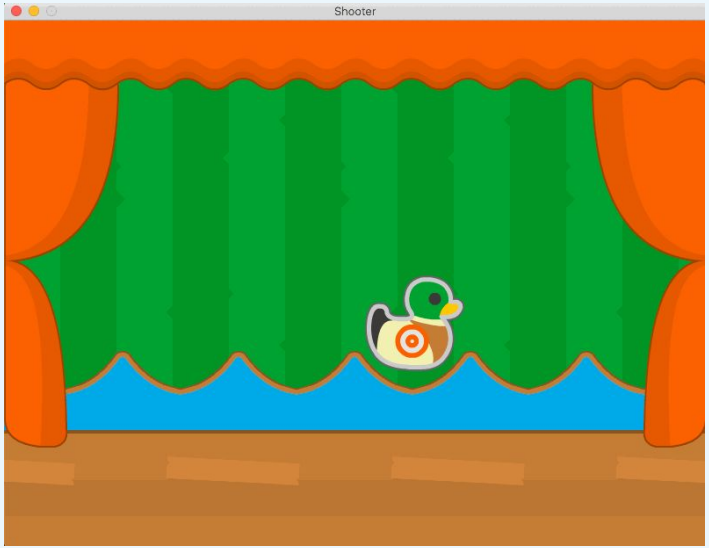
Esercizio n.2

Onde in movimento, generazione di papere

Secondo esercizio
Aggiungere animazioni

Dopo il 1° esercizio

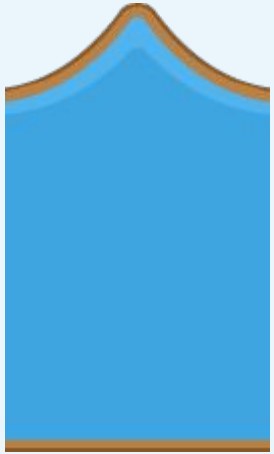
Al termine di questo esercizio



Secondo esercizio
Aggiungere animazioni

Le immagini necessarie:

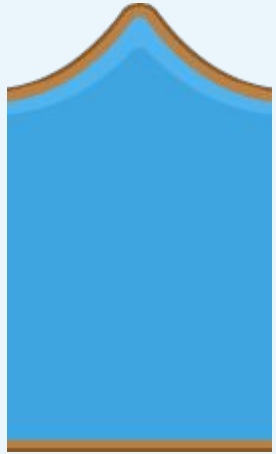
- PNG/Objects/duck_outline_target_white.png
- PNG/Stall/water1.png



Secondo esercizio
Aggiungere animazioni

Obiettivi:

- I movimenti sono in entrambe le direzioni, sù/giù e destra/sinistra. Per calcolare la direzione si possono usare dei moltiplicatori +1/-1
- Le onde vanno costruite ripetendo l'immagine per riempire lo schermo, ma dato che l'animazione si muove, sarà necessario aggiungere un'ulteriore immagine oltre il bordo (che sarà visibile durante l'animazione)



Secondo esercizio
Aggiungere animazioni

Le papere si muovono lentamente sù e giù e velocemente verso destra

Devono essere generate casualmente in `Update()`, come in questo esempio:

```
rand.Seed(time.Now().Unix())
// every second there's 30% possibilities to generate a missing duck
if len(visibleDucks) < maxDucks {
    if tick%60 == 0 && rand.Float64() < 0.3 {
        visibleDucks = append(l.ducks, newDuck())
    }
}
```

Secondo esercizio
Aggiungere animazioni

La posizione X della papera ci dice se è ancora visibile nello schermo. Una volta uscita dallo schermo, può essere rimossa dalla lista di papere “attive”:

```
n := 0
for _, duck := range visibleDucks {
    if duck.xPosition <= screenWidth {
        visibleDucks[n] = duck
        n++
    }
}
visibleDucks = visibleDucks[:n]
```

https://github.com/golang/go/wiki/SliceTricks#filter-in-place

Extra

Idee aggiuntive:

- usare le immagini dallo spritesheet invece di singoli file
  - astrarre la logica di estrazione immagine invece di farlo per ognuna
- raggruppare le costanti in un unico posto aiuta a regolare i valori quando si testa il gioco
- aggiungere un supporto sotto la papera
- le papere possono anche ruotare leggermente durante il movimento, si veda qui a destra:

User input

Tastiera

```
func (g *Game) Update(screen *ebiten.Image) error {
   if ebiten.IsKeyPressed(ebiten.KeyUp) {
       obj.moveUp()
   }
   return nil
}
```

```
ebiten.IsKeyPressed(k Key) bool
```

La funzione riceve **Key**, un nuovo **tipo** definito in Ebiten

Ebiten

Tastiera

```
type Key int
const (
    KeyX             Key = Key(driver.KeyX)
    KeyY             Key = Key(driver.KeyY)
    KeyZ             Key = Key(driver.KeyZ)
    KeyBackslash     Key = Key(driver.KeyBackslash)
    KeyBackspace     Key = Key(driver.KeyBackspace)
    // ...
)
```

Lista dei Key disponibili:
https://pkg.go.dev/github.com/hajimehoshi/ebiten/v2#Key

A proposito di **tipi**, vediamo come definire un tipo **direction** per aiutarci nella gestione della direzione degli oggetti:

```
type direction int
const (
    right direction = 1
    left  direction = -1
)
```

Possiamo anche aggiungere un metodo `invert()` al tipo:

```
func (d direction) invert() direction {
    return -d
}
```

Ecco come usarlo nel nostro gioco per muovere le papere:

```
type duck struct {
    yDirection     direction
}

if duck.yPosition >= duck.maxYPosition {
    duck.yDirection = duck.yDirection.invert()
}
```

Mouse

Come per la tastiera, anche per il mouse si può verificare un click:

```
if ebiten.IsMouseButtonPressed(ebiten.MouseButtonLeft) {
    obj.shoot()
}
```

La posizione del mouse si può ottenere con:

```
x, y := ebiten.CursorPosition()
```

La posizione è relativa allo `screen`:

(0,0) dello `screen` è (0,0) del mouse, anche se la finestra del gioco si muove nello schermo dell'utente

https://github.com/develersrl/webinar-go-game-development/tree/master/examples/05_inputs

Ebiten
Debounce

Gli input vengono rilevati ad ogni `Update()`, quindi ogni 16.6ms

Se si tiene premuto per un tempo superiore, il gioco rileva due input.
Non è quello che, di solito, si vuole

Va quindi aggiunto un debouncing per evitare i doppi input

Ebiten

Debounce

```
type game struct {
   lastClickAt time.Time // 0-value of time is 0001-01-01 00:00:00 +0000 UTC
}
const debouncer = 100 * time.Millisecond
func (g *game) Update(screen *ebiten.Image) error {
   if ebiten.IsKeyPressed(ebiten.KeyA) && time.Now().Sub(g.lastClickAt) > debouncer {
       log.Printf("A pressed")
       g.lastClickAt = time.Now()
   }
   return nil
}
```

Musica e suoni

Per riprodurre suoni in Ebiten è necessario definire un **audio context** che definisce il sample rate dei flussi audio.

Il sample rate deve essere lo stesso per tutti i flussi, **però** i decoder eseguono il resample automaticamente.

Multipli flussi audio vengono mixati automaticamente (ma troppi flussi contemporanei possono generare dei disturbi)

https://pkg.go.dev/github.com/hajimehoshi/ebiten@v1.12.1/audio

I file audio possono essere gestiti come gli altri assets, con file2byteslice

```
//go:generate file2byteslice -input ./hit.wav  -output hit.go -package assets -var Hit
```

Il contesto audio è facilmente generabile con:

```
var audioContext *audio.Context
func init() {
    var err error
    audioContext, err = audio.NewContext(44100)
}
```

I suoni (ad esempio la musica di fondo) possono essere riprodotti in loop infiniti.

Anche il decoder corretto per il flusso deve essere usato.

```
import "github.com/hajimehoshi/ebiten/audio/vorbis"

oggS, _ := vorbis.Decode(audioContext, audio.BytesReadSeekCloser(RagtimeSound))

s := audio.NewInfiniteLoop(oggS, oggS.Length())

player, _ := audio.NewPlayer(audioContext, s)
player.Play()
```

I “player” di singoli suoni devono essere riavvolti per essere usati più volte:

```
import "github.com/hajimehoshi/ebiten/audio/wav"

sound, _ := wav.Decode(audioContext, audio.BytesReadSeekCloser(src))
player, _ := audio.NewPlayer(audioContext, sound)
player.Rewind()
player.Play()
```

https://github.com/develersrl/webinar-go-game-development/tree/master/examples/06_sounds

Font

Per realizzare le scritte, oltre alle immagini, è anche possibile usare font a piacere, grazie al package `text`:

https://pkg.go.dev/github.com/hajimehoshi/ebiten@v1.12.1/text

Anche in questo caso, possiamo usare file2byteslice:

```
//go:generate file2byteslice -input ./penguin_attack/PenguinAttack.ttf  -output
font.go -package main -var FontAsset
package main
```

Il font dell'esempio è https://www.dafont.com/it/penguin-attack.font?l[]=10 (GPL)

https://github.com/develersrl/webinar-go-game-development/tree/master/examples/07_fonts

Il font può essere importato con:

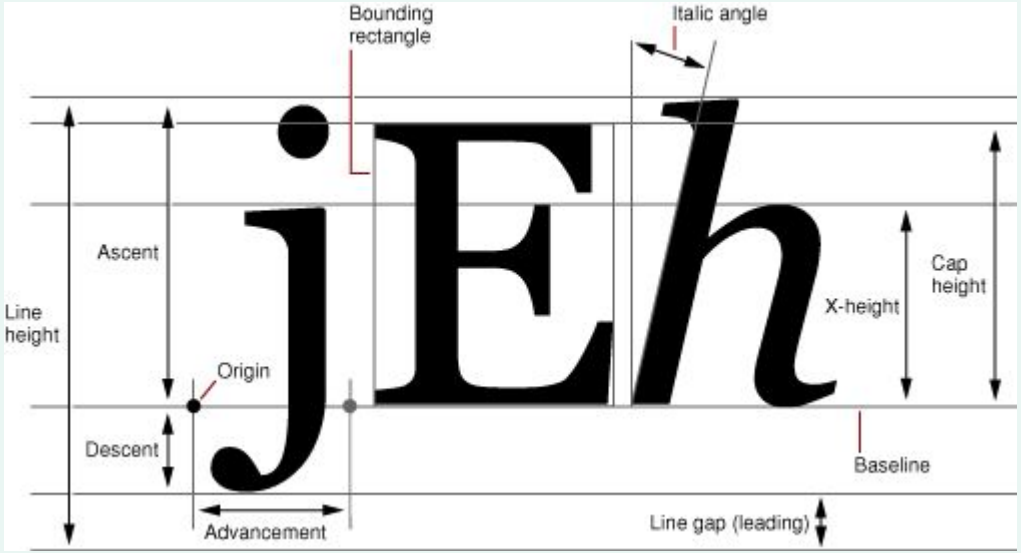
```
var myFont font.Face
func init() {
   tt, _ := truetype.Parse(FontAsset)

   myFont = truetype.NewFace(tt, &truetype.Options{
       Size:    36,
       DPI:     72,
       Hinting: font.HintingFull,
   })
}
```

Quindi `Draw()` può usare il font per realizzare dei testi:

```
func (g *game) Draw(screen *ebiten.Image) {
   // calculate the rectangle containing the text
   bounds := text.BoundString(myFont, "Hello, Gophers!")
   // write moving the text down by its height
   text.Draw(screen, "Hello, Gophers!", myFont, 10, bounds.Dy(), color.White)
}
```

Regola per il posizionamento (facile no?):

se il testo fosse un solo punto “.”, allora sarebbe posizionato nel punt x,y passato a `Draw()`



UI/UX e scene

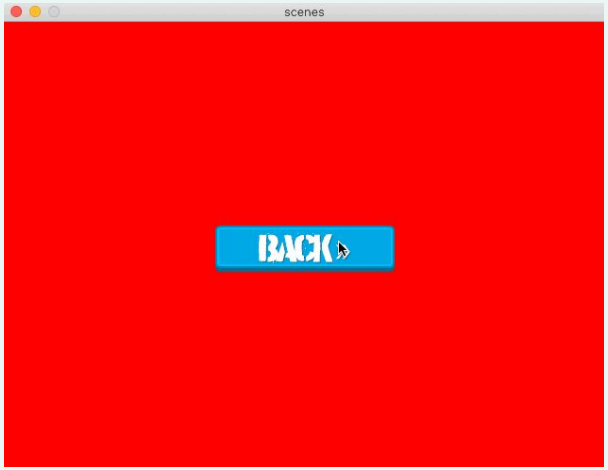

12345
NICKNAME
WINS: 12345
GLOBAL RANKING
TOP PLAYERS
CRAZY CLOWNS
START
TOTURIAL
NEXT TOURNAMENT
00h 00m
REGISTER
TOP TEAMS

Per realizzare un gioco “vero” è necessario anche pensare ad una UI e alla user-experience (quindi bottoni, scene, opzioni, ecc)

Quanto visto finora è comunque tutto quel che serve per disegnare una UI.

Altre cose (ad esempio il salvataggio del gioco su file) non sono gestite da Ebiten ma possono essere realizzate con codice Go

Vediamo un semplice esempio di “cambio di scena”



https://github.com/develersrl/webinar-go-game-development/tree/master/examples/08_scenes

Nell’esempio possiamo definire delle scene che includano tutto il necessario per essere disegnate, mentre `game` sa quale scena è attiva:

```
type scene struct {
   // add required elements
}
type game struct {
   scenes      map[string]*scene
   activeScene string
}
```

La scena include l'immagine del bottone, il colore di sfondo e il nome della scena da attivare al click:

```
type scene struct {
    img       *ebiten.Image
    nextScene string
    bg        color.Color
}
```

Al click, si cambia la scena:

```
func (g *game) Update(screen *ebiten.Image) error {
    s := g.scenes[g.activeScene]
    if ebiten.IsMouseButtonPressed(ebiten.MouseButtonLeft) {
        x, y := ebiten.CursorPosition()
        if isClicked(s.img) {
            g.activeScene = s.nextScene
        }
    }
    return nil
}
```

`Draw()` si limita a disegnare la scena attiva, senza sapere quale sia:

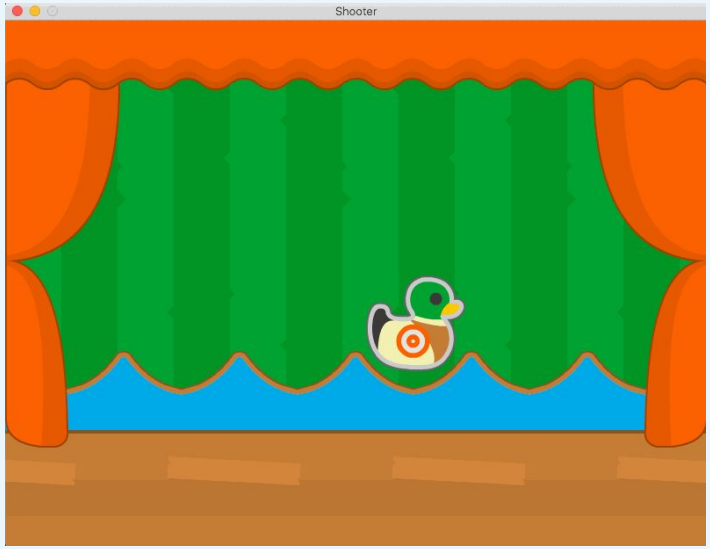
```
func (g *game) Draw(screen *ebiten.Image) {
    s, ok := g.scenes[g.activeScene]
    screen.Fill(s.bg)
    op := &ebiten.DrawImageOptions{}
    op.GeoM.Translate(float64(x), float64(y))
    screen.DrawImage(s.img, op)
}
```

Esercizio n.3

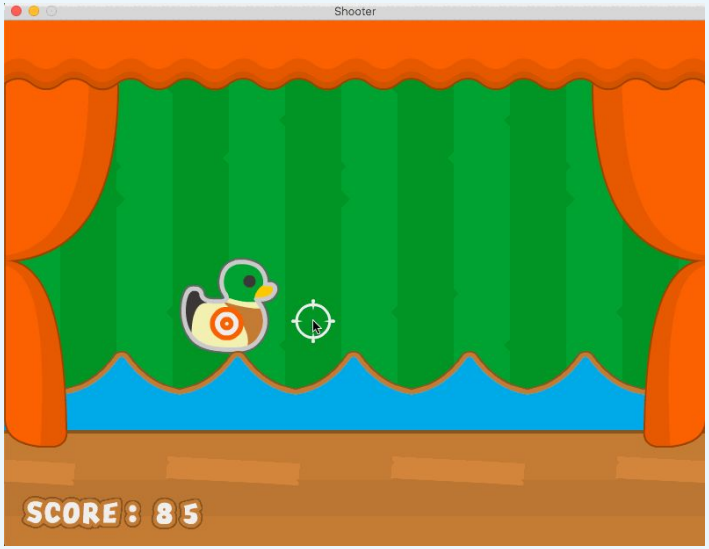
Mirino, spari, punteggi e suoni

Terzo esercizio

Dopo il 2° esercizio



Al termine di questo esercizio



Obiettivi:

- Aggiungere la musica di sottofondo
- Disegnare il mirino e muoverlo insieme al mouse
- Gestire il punteggio e mostrarlo (con testo o immagini)
- Al click, verificare se una papera è colpita (usare un rettangolo) e aggiungere 10 punti. Eseguire il suono.
- (extra) Scalare 5 punti per ogni papera mancata. Eseguire il suono.

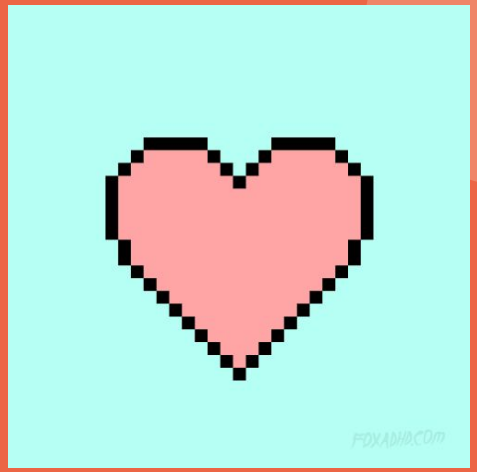
Terzo esercizio

I file necessari:

- PNG/HUD/crosshair_{white,red}_large.png
- Font personalizzati o PNG/HUD/text_*.png
- hit.wav e miss.wav
- ragtime.ogg (musica di sottofondo)

Extra:

- Aggiungere una scena iniziale e un bottone “Gioca” per iniziare il gioco
- Aggiungere una scena finale e un bottone “Gioca ancora”
- Creare una classifica: i più veloci a realizzare 100 punti? Più punti in 30 secondi?
- Al termine del gioco, i giocatori devono inserire il proprio nome per la classifica



Grazie per l'attenzione!

https://github.com/develersrl/webinar-go-game-development

Vuoi rimanere aggiornato sugli eventi Develer?
Seguici nei nostri canali social: