User input

Keyboard

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

`ebiten.IsKeyPressed(k Key) bool`

The function get **Key**, which is a type defined by Ebiten

Ebiten

Keyboard input

```go
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)
    // ...
)
```

For the list of available keys:
https://pkg.go.dev/github.com/hajimehoshi/ebiten/v2#Key

Ebiten

Define new types

Defining a new type is something we've already seen when defining structs, but we can define types also on other base types:

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

Ebiten

Define new types

We can also add behaviours to these types:

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

The direction type can be used in our game to define the direction of the objects, and we can easily invert their movement (we're mixing abstraction and math in a "smart" way)

Ebiten

Define new types

This is a small example that can apply to our game:

```
type duck struct {
   yDirection     direction
}

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

Mouse

As for the keyboard, we can check also mouse clicks:

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

Ebiten
Mouse input

The cursor position can be obtained with:

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

The position is always relative to the game screen:

(0,0) in the screen is (0,0) of the cursor, also if you move the game window around

`https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/05_inputs`

Both for keyboard and mouse clicks, note that if the user clicks for a long time, you'll see the clicks for multiple Update() calls.

This is not wrong per-se, but depending on the game, you could add a debouncer to avoid duplicated inputs:

Ebiten

Debounce input

```
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
}
```

Ebiten
More inputs

Ebiten also manages touch inputs and gamepads

Music and sounds

Ebiten can easily play sounds. All sounds must share an **audio context** that defines a sample rate of the streams.

The sample rate must be the same for all streams, **however** decoders automatically resample the streams, so we don’t really need to care.

Once a context is defined, streams can be played on it. Multiple streams are automatically mixed (too many can create distortions)

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

Ebiten
Sounds

As for other assets, I suggest adding sounds as go files and using generators:

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

Creating the audio context is straightforward:

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

I'm using global vars here but you would want to add it to your Game object

Ebiten

Sounds

A background music could be played within an infinite loop, the file start-end must be mergeable without interruptions. Depending on the file, you'll need different decoders.

```go
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()
```

One-time sounds are are simpler to initialize and need to be rewinded every time:

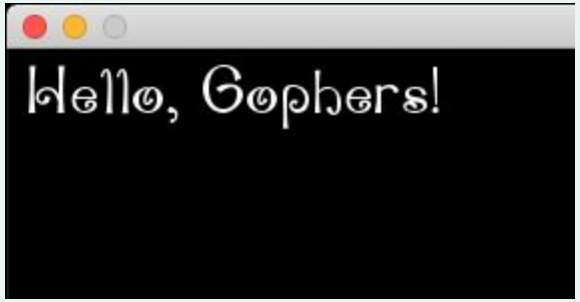
```go
import "github.com/hajimehoshi/ebiten/audio/wav"

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

https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/06_sounds

Fonts

Ebiten
Fonts

It is possible to use custom fonts instead of images, using the `text` package:



Hello, Gophers!

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

The font can be easily transformed to an asset with:

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

In my example the font is https://www.dafont.com/it/penguin-attack.font?l[]=10 (GPL)

`https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/07_fonts`

Then, load the font into the program:

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

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

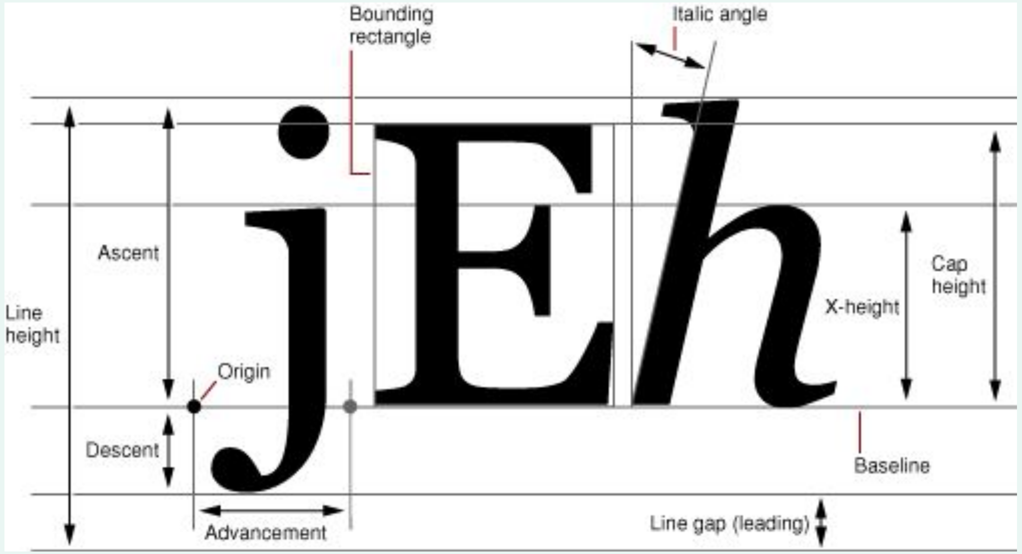
Now, we can write to the screen.

```go
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)
}
```

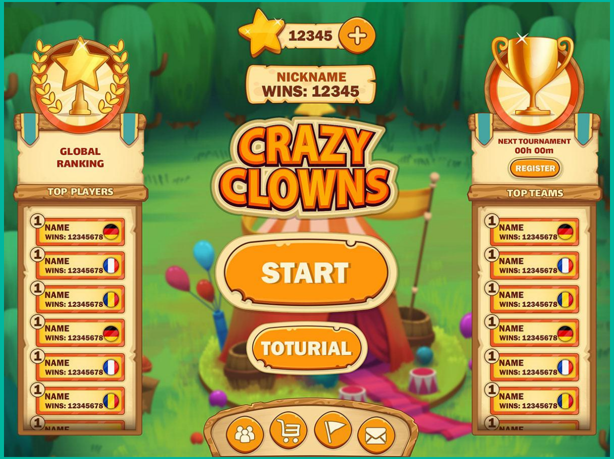
`BoundString` and `Draw` are the only functions in the package, easy.

Note on positioning, the rule is:

if the text is just a dot “.”, it will be drawn in the x,y point passed to `Draw()`



UI/UX and scenes



UI/UX are what transform a "draft" game to something more complex, with buttons, options, etc.

Adding a UI doesn't require more than what we've seen until now: images (or fonts) and user inputs.

You could decide to store scores on local files (but we won't see this now)

When thinking to a more complex game, we'll probably need multiple scenes

A scene completely changes the look and behaviour of the game and permits the user to move around

There's not a golden rule to add scenes to a game

Ebiten

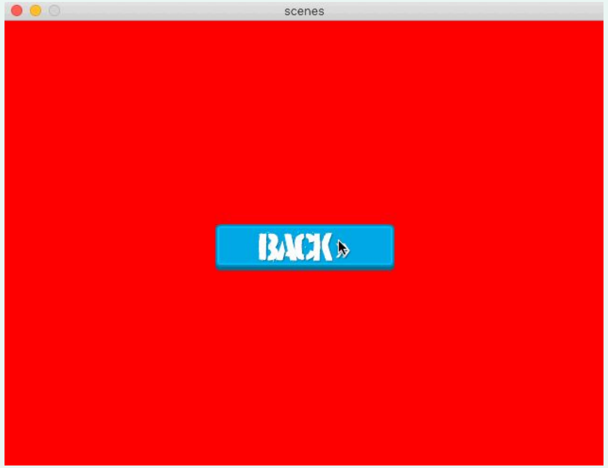
Scenes

An idea could be to define a scene type with all you need to draw the scene and then leave the game to know which scene is active:

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

Ebiten

Scenes



https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/08_scenes

The scene includes button img, background color and next scene (after click):

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

When the button is clicked, we change the scene:

```
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()` doesn't know about the scene, just draws:

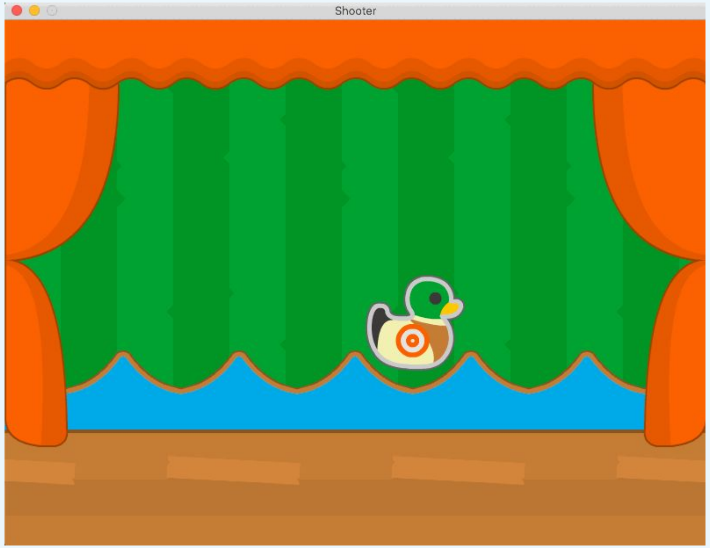
```go
func (g *game) Draw(screen *ebiten.Image) {
   s, ok := g.scenes[g.activeScene]
   screen.Fill(s.bg)
   op := &ebiten.DrawImageOptions{}
   op.GeoM.Translate(float64(x), float64(y))
   screen.DrawImage(s.img, op)
}
```

Exercise n.3

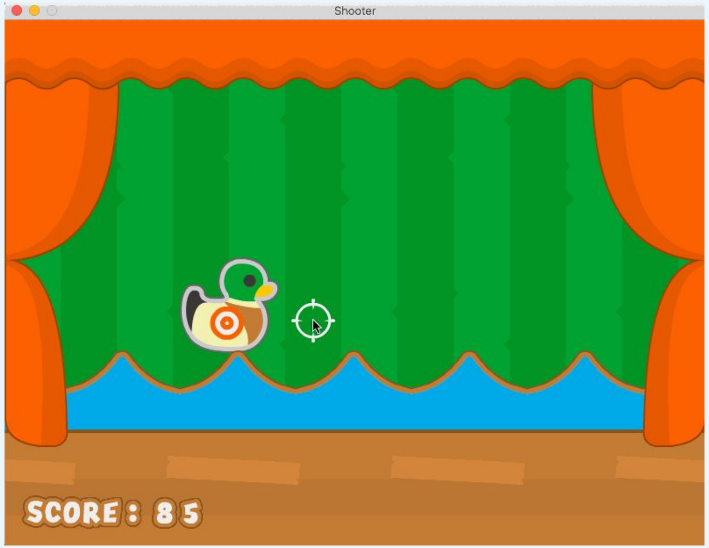
Mouse crosshair and clicks, add score, add sounds and background music

Third exercise
Music and user interaction

What you have now



What you'll have then (+ sound)



Third exercise
Music and user interaction

Goals:

- Add a background music
- Draw the crosshair, move it with the mouse cursor
- Define a global score
- On click, check if a duck has been hit (the cursor is on the duck rectangle). Add 10 points. Hit sound
- (optional) Remove 5 points when missed. Miss sound
- Write the score using images or custom font

Third exercise

Music and user interaction

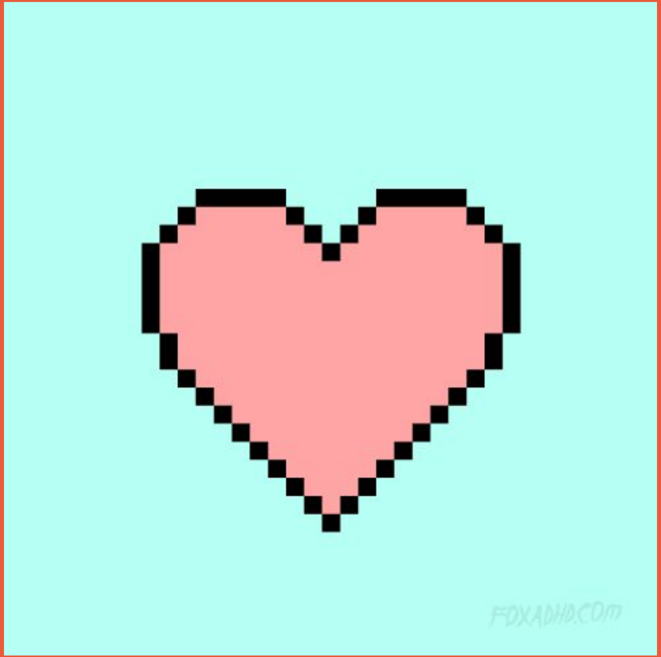
Assets you need:

- PNG/HUD/crosshair_{white,red}_large.png
- Custom fonts or PNG/HUD/text_*.png
- hit.wav and miss.wav
- ragtime.ogg (background music)

Third exercise
Music and user interaction

Extras:

- Add an initial scene with a “Play” button
- Add an end scene, with “Play again” button
- Create a leaderboard: the fastest to reach 100 points? The game lasts 30 secs?
- At the end of the game, the user is asked to insert their name for the leaderboard



That’s all folks!

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