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 please ask questions in the chat.

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

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/

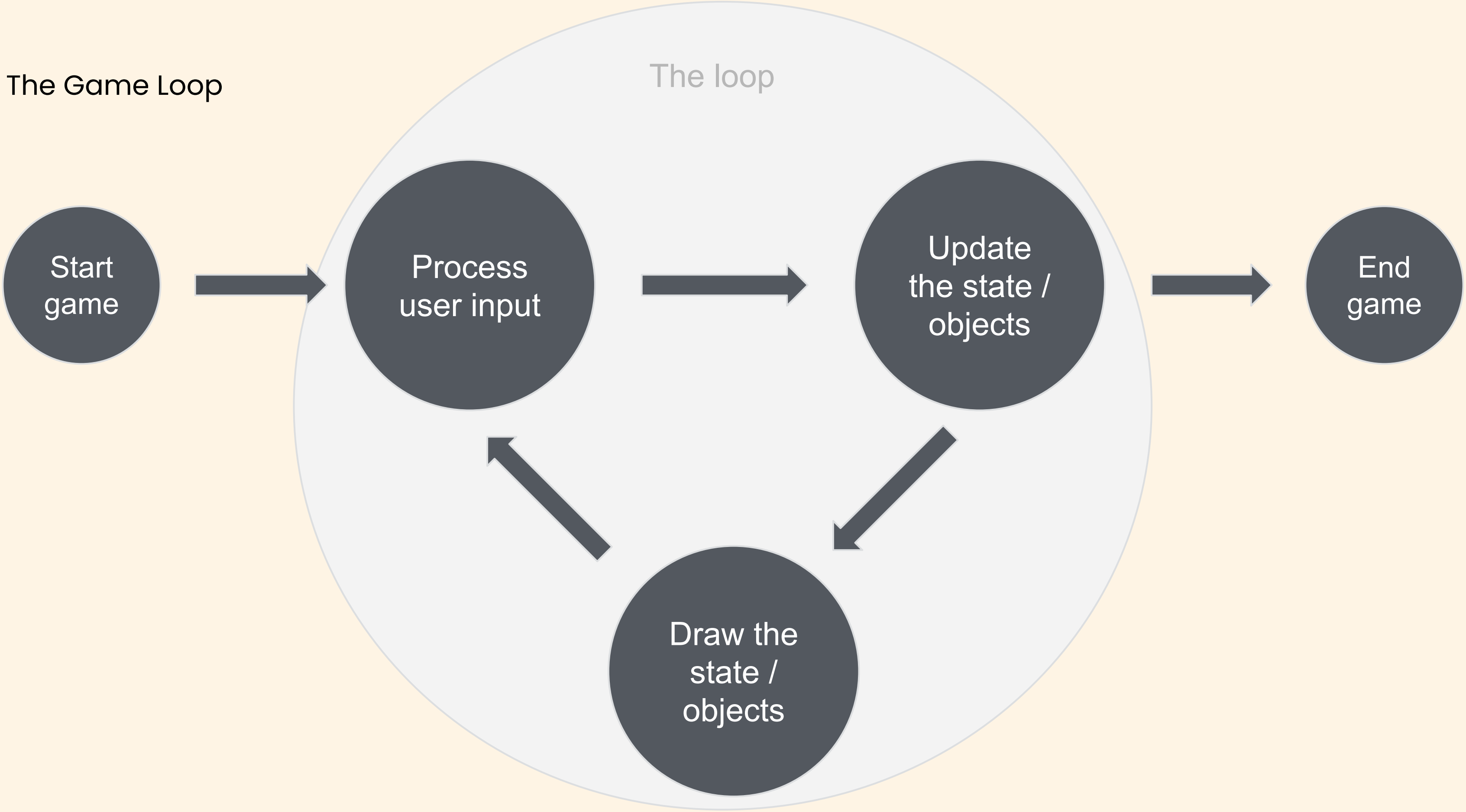
The Game Loop

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

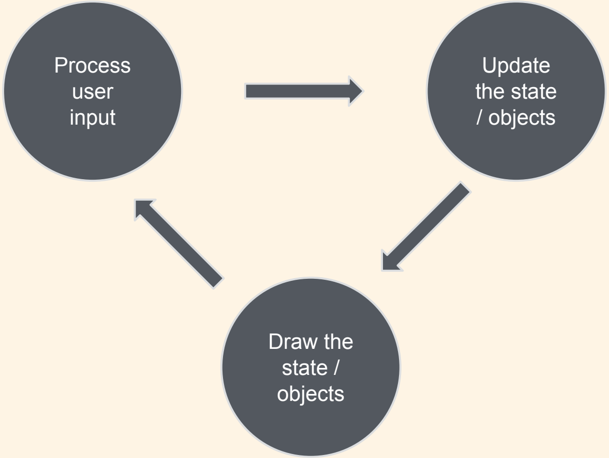


The Game Loop

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

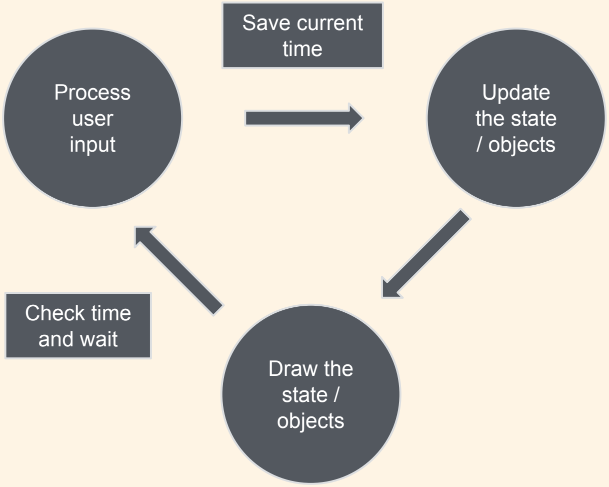


The Game Loop

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

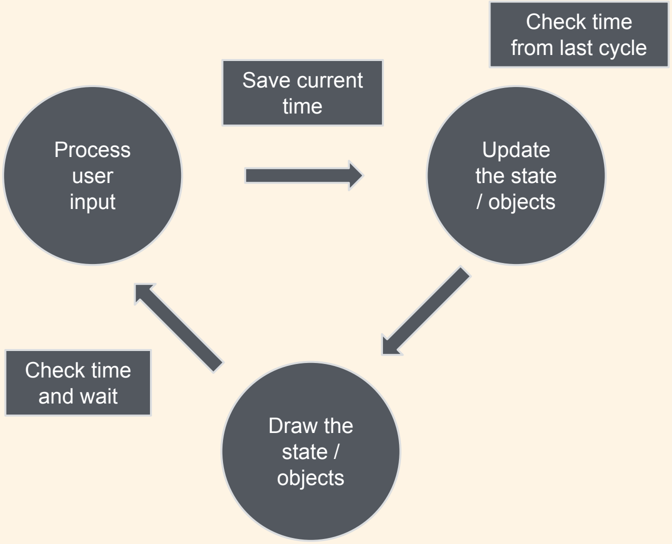


The Game Loop

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”:

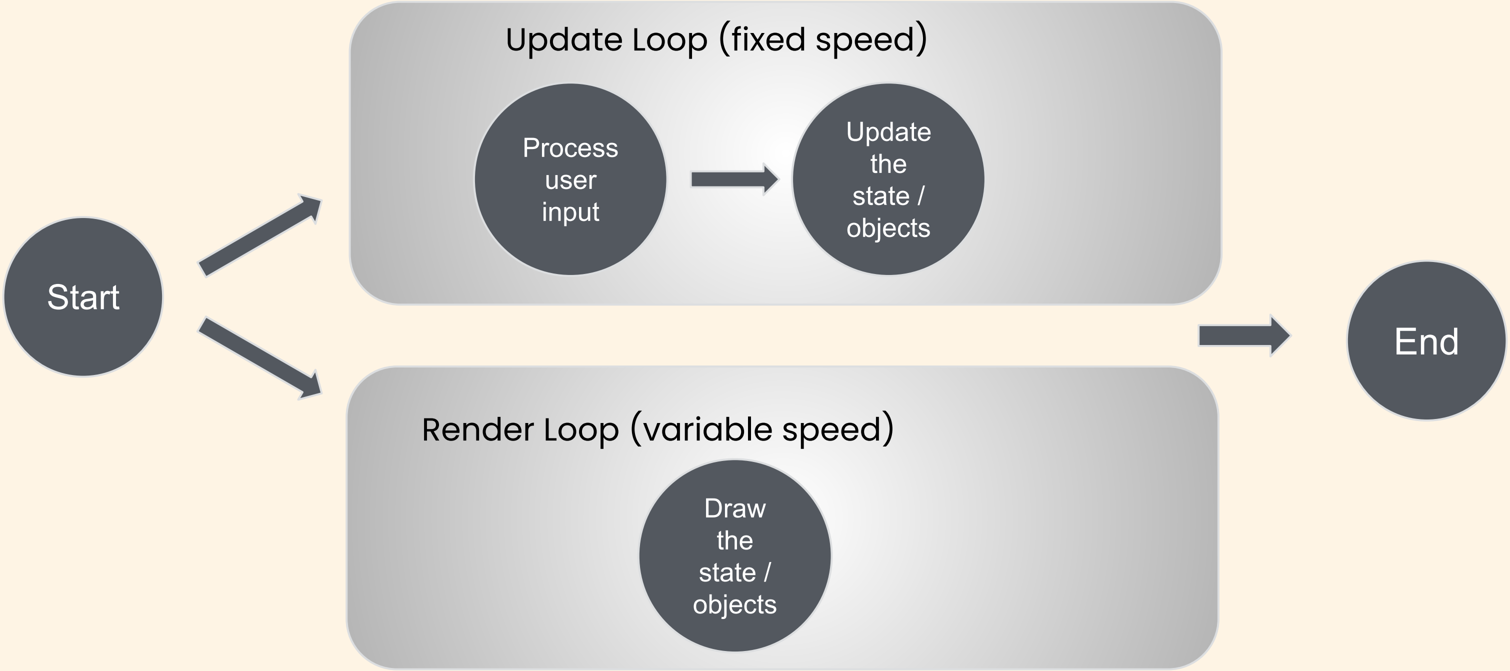
- a character not hitting a wall because the update step has been executed too late and the wall is below the character



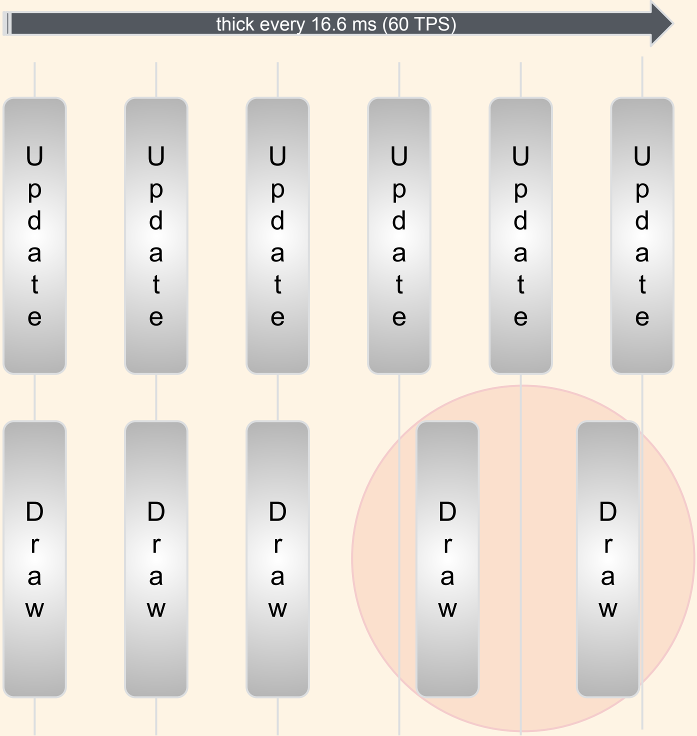
3rd solution (used by Ebiten):

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

The rendering process (draw) runs at its own speed


Update Loop (fixed speed)
Process user input
Update the state / objects
Start
End
Render Loop (variable speed)
Draw the state / objects

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 (/ebíteɴ/)

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.

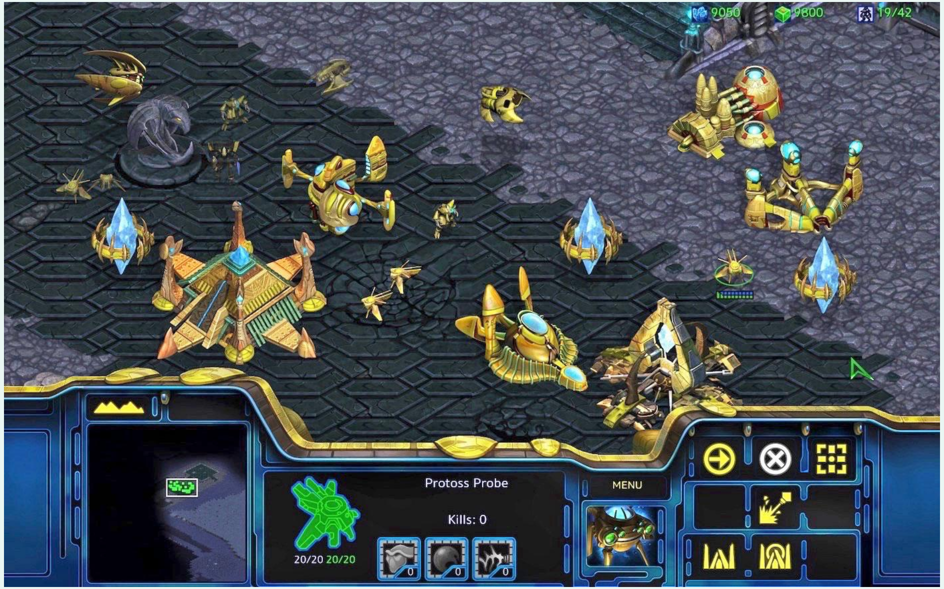
🌏 https://ebiten.org

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

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

Ebiten

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



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

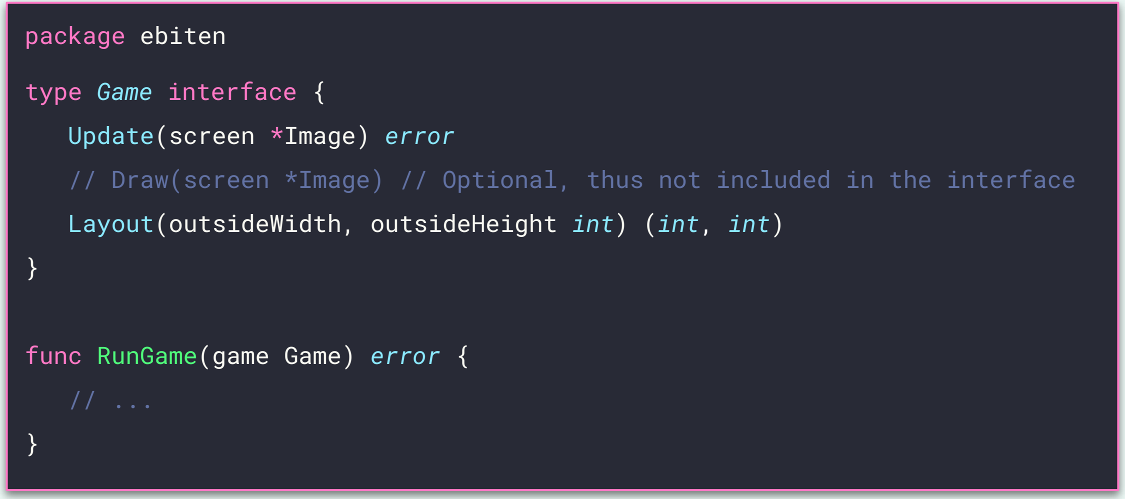
Ebiten

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.”

Ebiten

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



```go
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

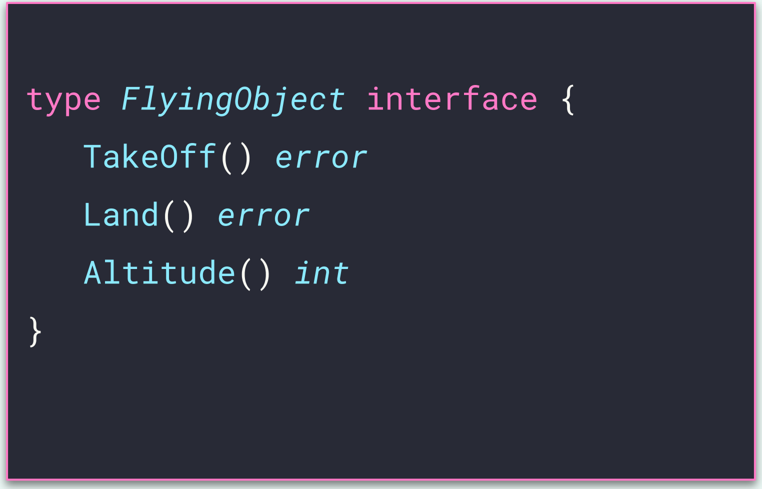
Go interfaces are named collections of method signatures.

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.

Go intefaces

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:



```go
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()
}
```

Ebiten

the game interface

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

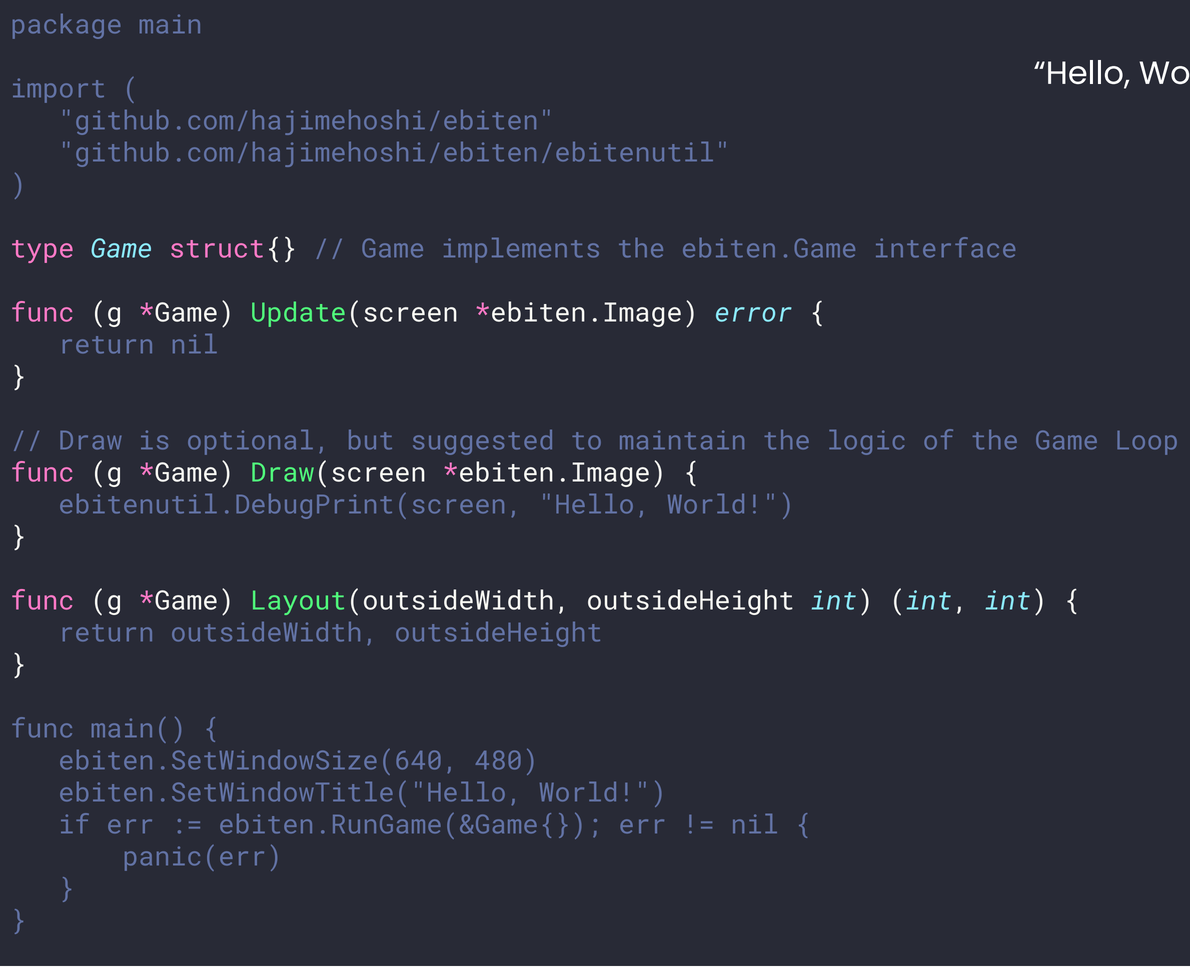
“Hello, World!” with Ebiten

```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



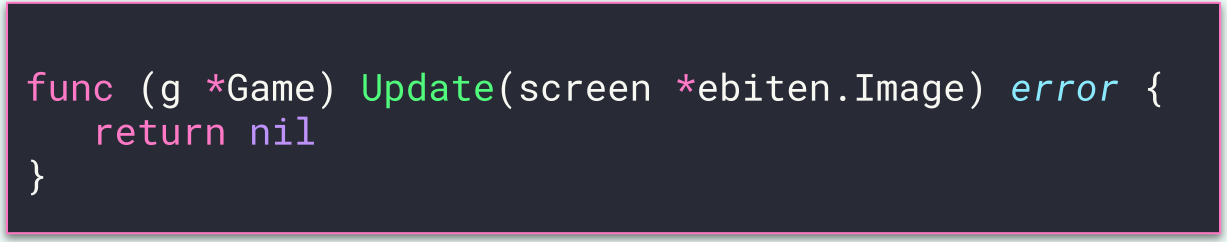
```go
package main

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

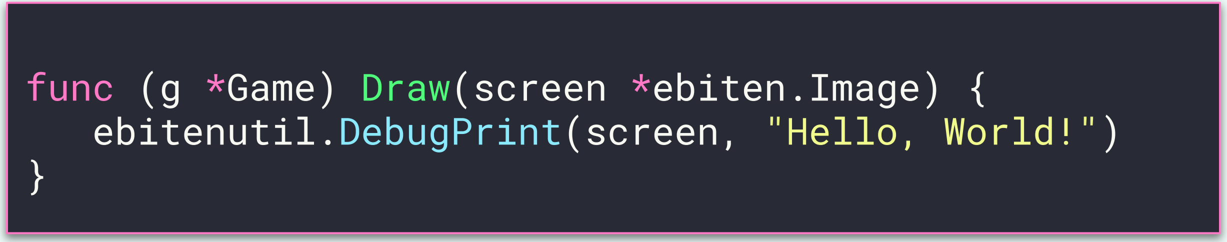


```go
func (g *Game) Update(screen *ebiten.Image) error {
    return nil
}
```

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

Game interface

Draw() function

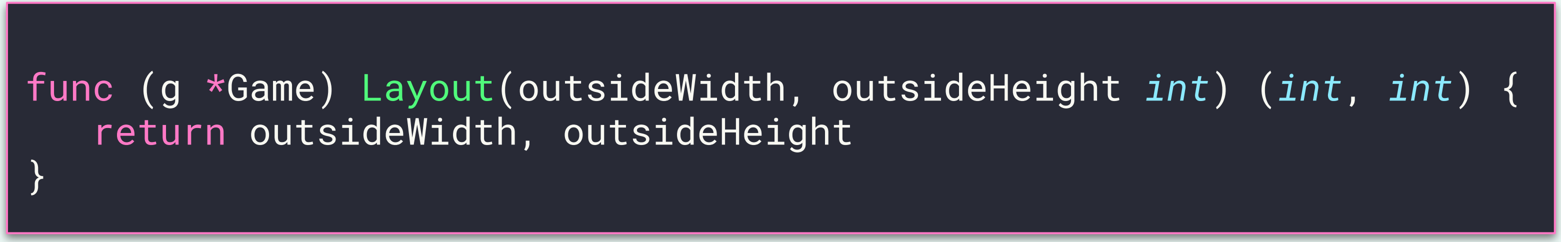


```go
func (g *Game) Draw(screen *ebiten.Image) {
   ebitenutil.DebugPrint(screen, "Hello, World!")
}
```

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

Game interface

Layout() function



```go
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

32

Ebiten

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

Images

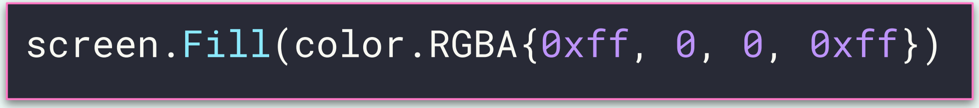
`ebiten.Image` has a lot of useful methods, full list at

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

Ebiten

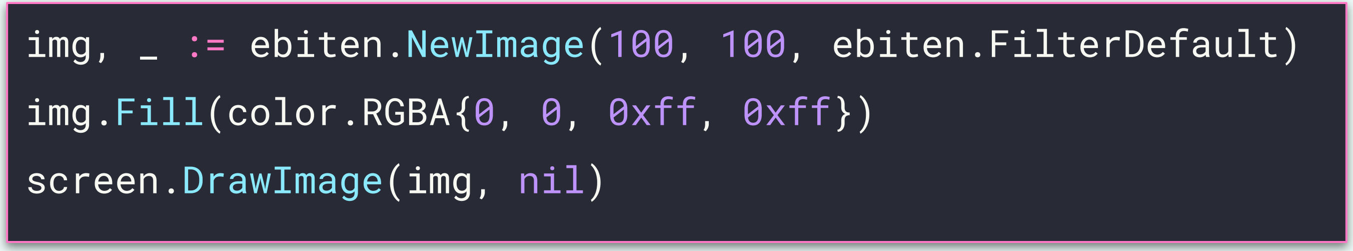
Images

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



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

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



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

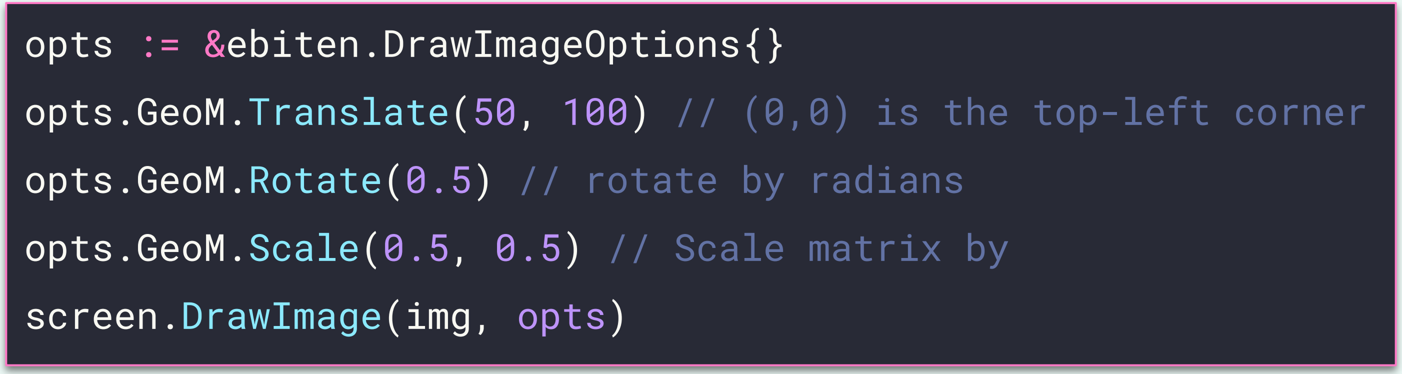
Ebiten

Image options

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:



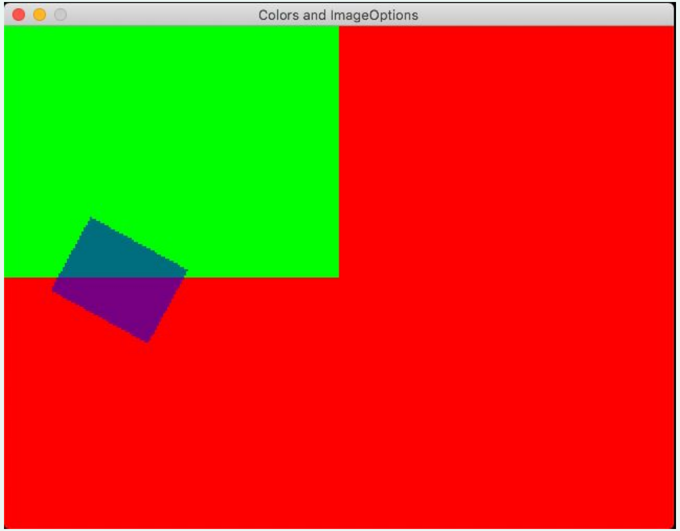
```go
opts := &ebiten.DrawImageOptions{}
opts.GeoM.Translate(50, 100) // (0,0) is the top-left corner
opts.GeoM.Rotate(0.5) // rotate by radians
opts.GeoM.Scale(0.5, 0.5) // Scale matrix by
screen.DrawImage(img, opts)
```

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

Ebiten

Image options

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

Ebiten

Images from files

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

Ebiten

Images from files

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:

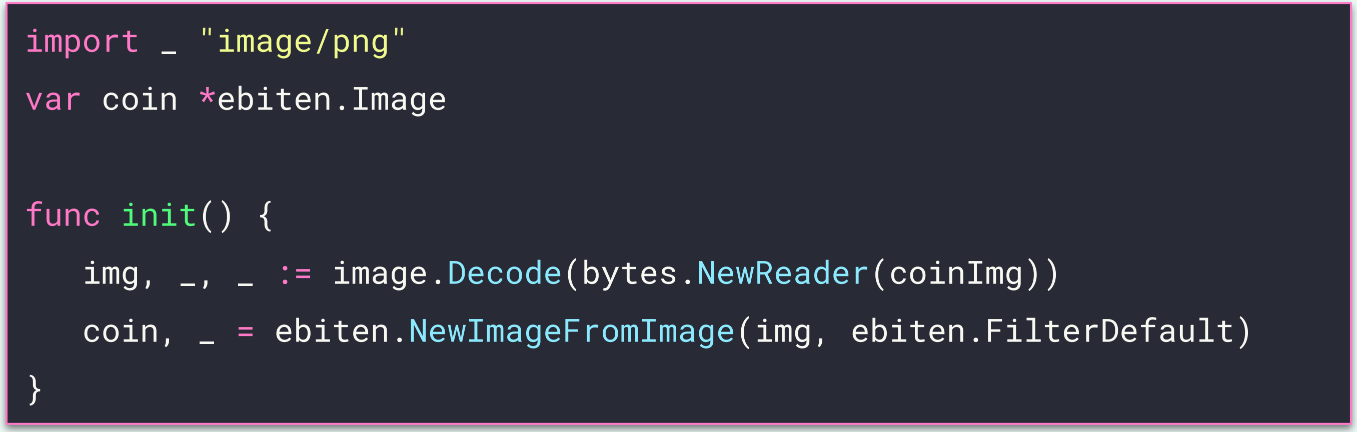
```go
package main

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

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

Ebiten

Images from files

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



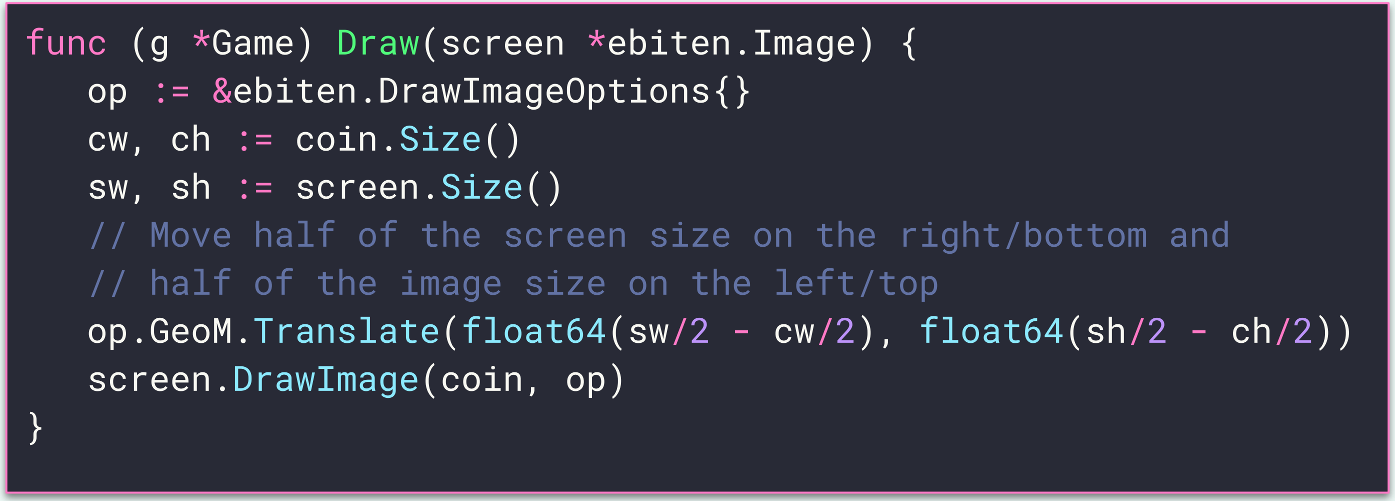
```go
import _ "image/png"
var coin *ebiten.Image

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

Ebiten

Images from files

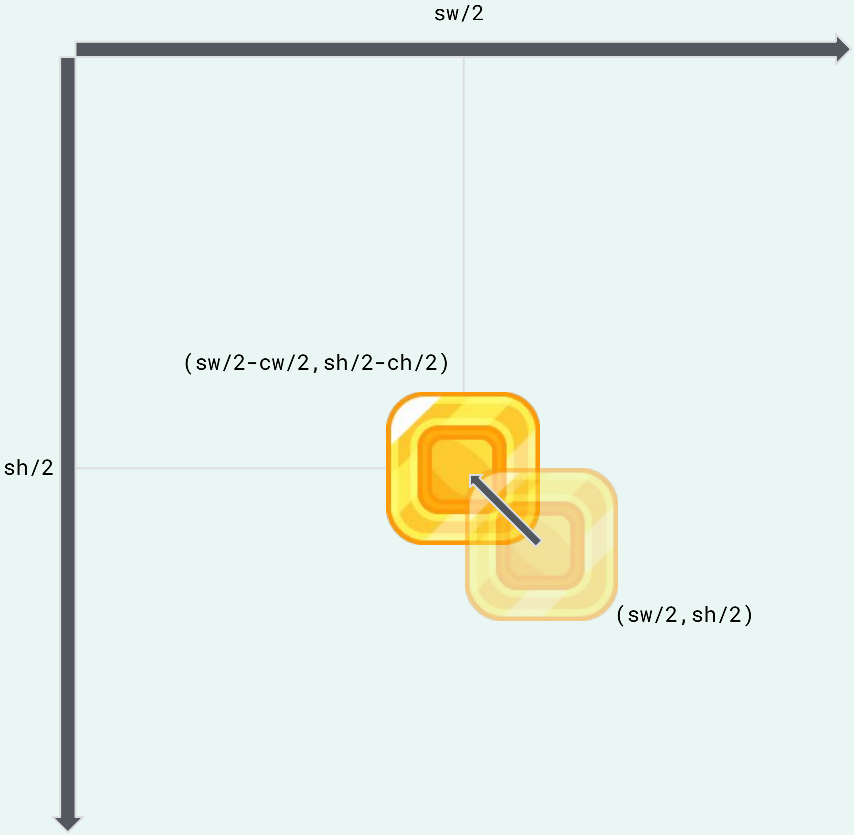
The Draw function just moves the image to the center of the screen:



```go
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

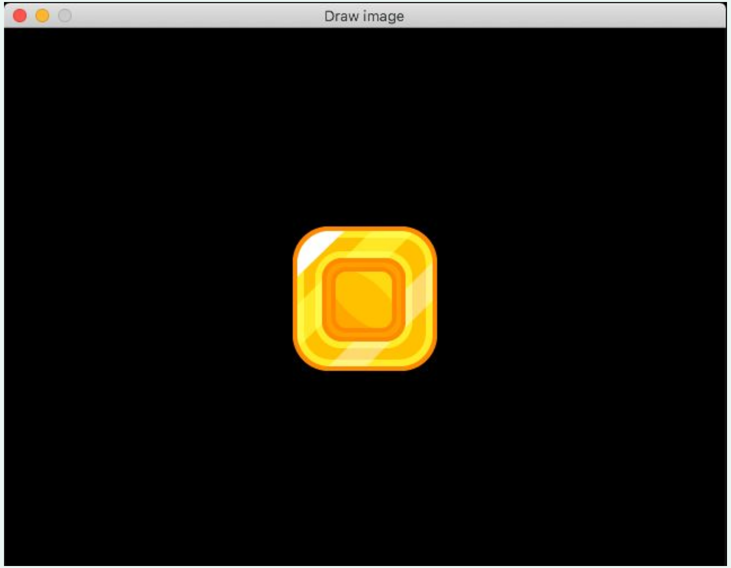
Images from files



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

Ebiten

Images from files

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

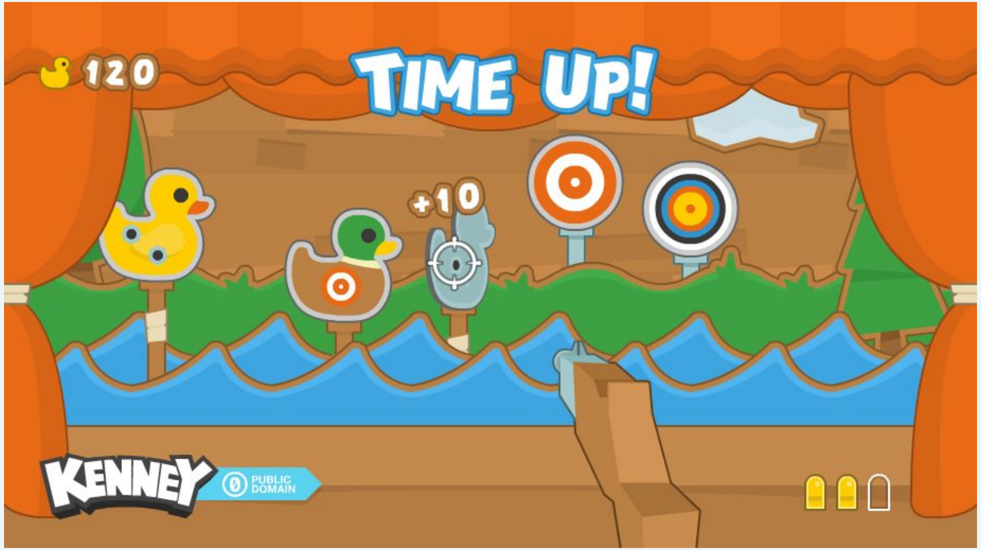
Create a `generate.go` file with this content (more lines can be added):

```
//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.

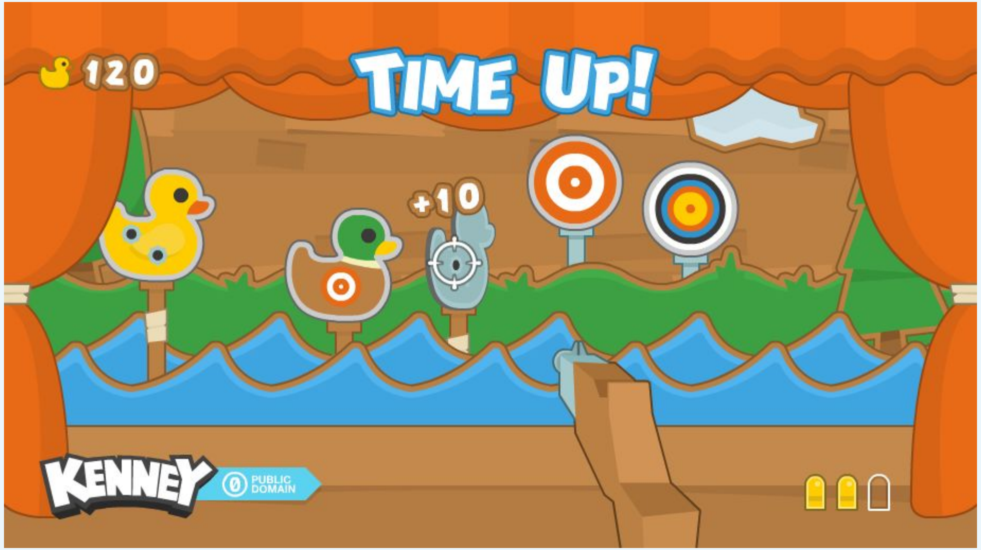
Exercise n.1

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**



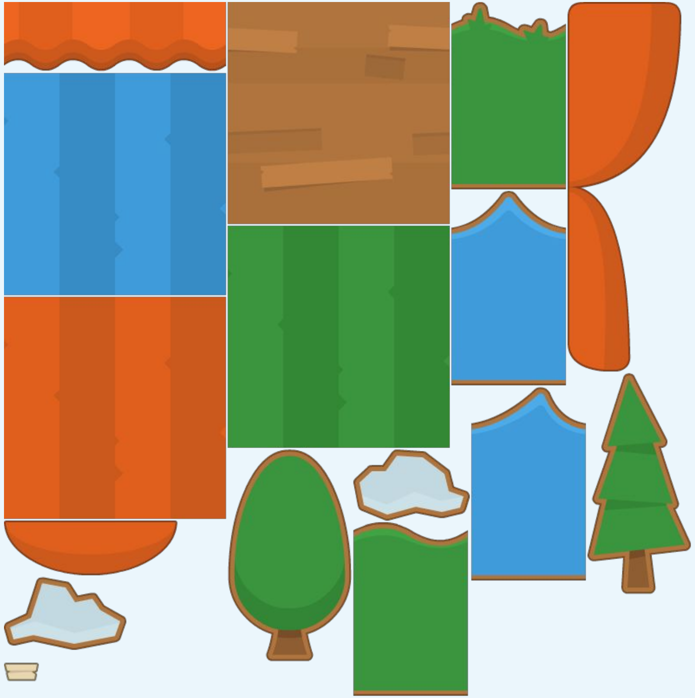
First exercise

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:

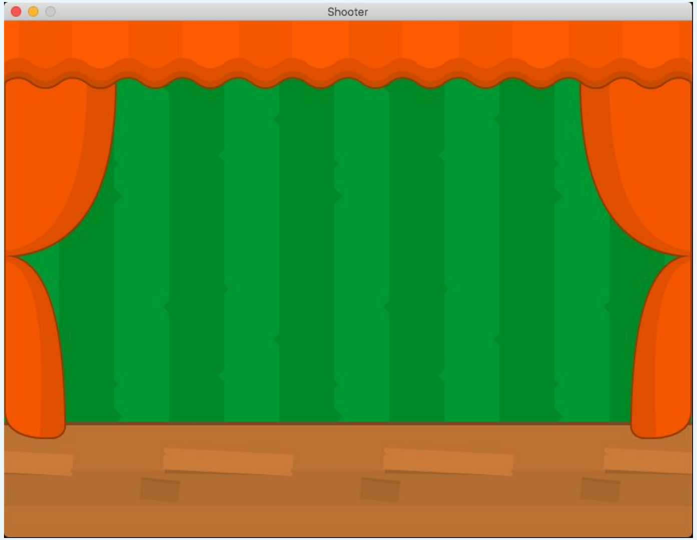
- **bg_green.png** for the background
- **curtain.png** for the right/left curtains
- **curtain_straight.png** for the top curtain
- **bg_wood.png** for the desk



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

First exercise

Static images

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

Some ideas:

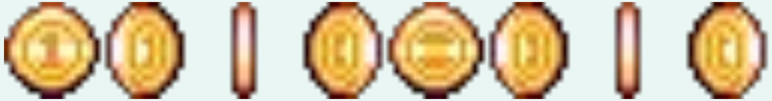
- Images can be represented by an Object interface with the Update and Draw functions and the Game can hold a list of objects (the order is important!) calling Update and Draw of all objects without knowing the type of each object
- Each object can have its own constructor, called by the main game constructor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

Animations

Ebiten

Image animation

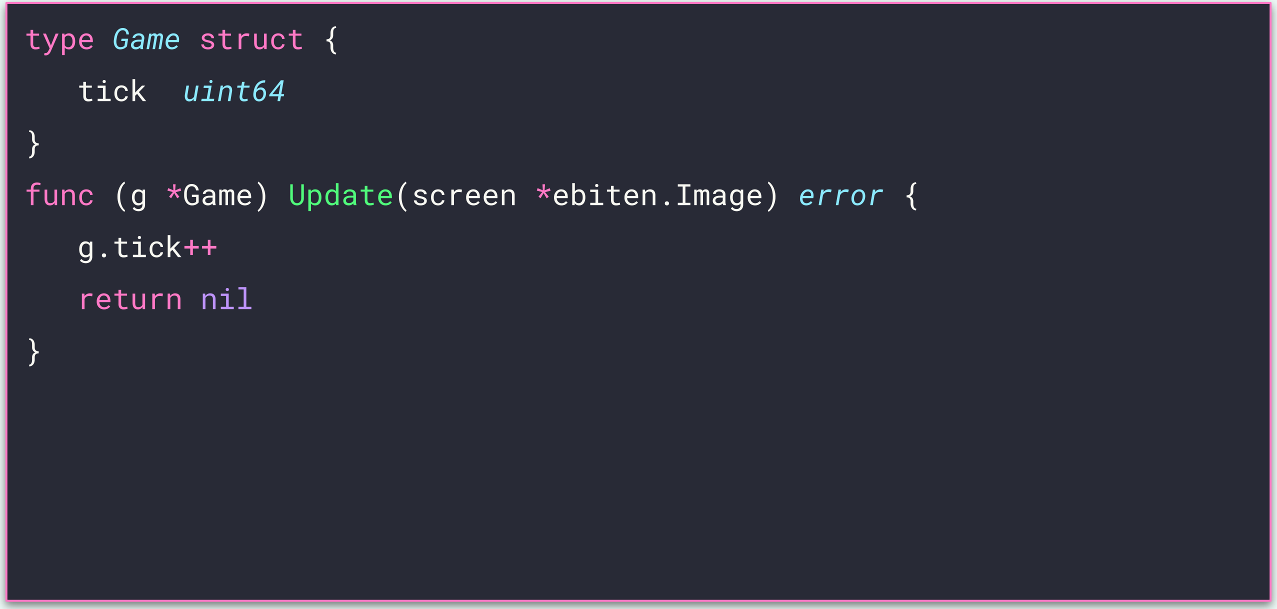
To animate an image the most popular way is to draw the frames of the animation using a spritesheet, so that only a single image must be loaded once:



Ebiten

Image animation

Use a simple “state” to know at which tick of the game we are:

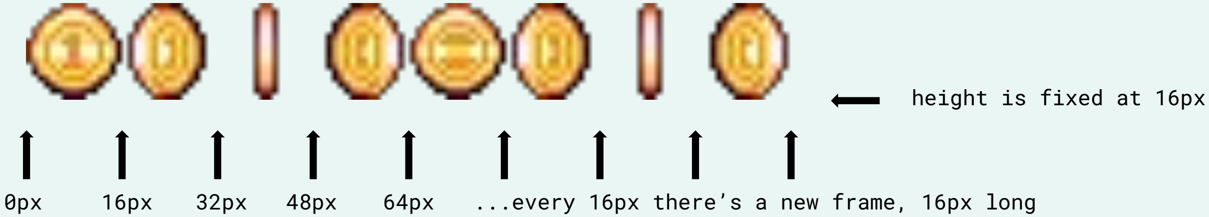


```go
type Game struct {
   tick  uint64
}
func (g *Game) Update(screen *ebiten.Image) error {
   g.tick++
   return nil
}
```

Ebiten

Image animation

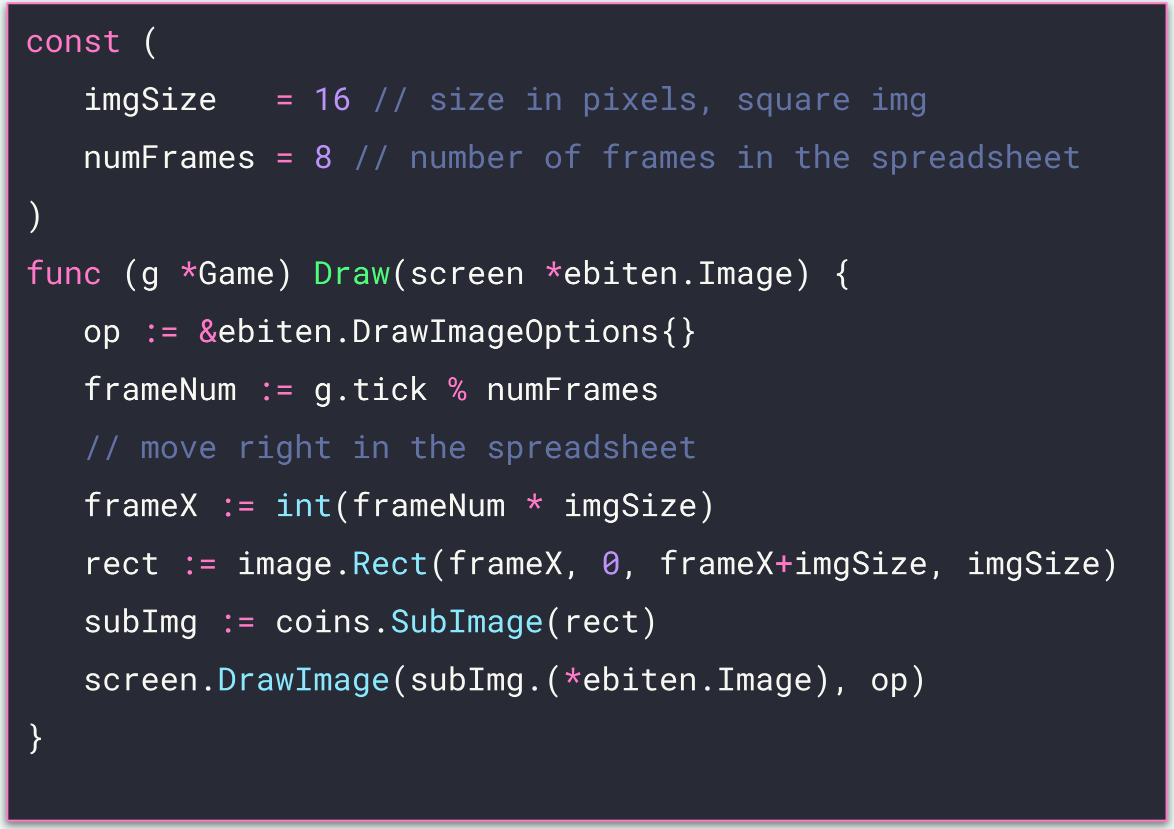
With fixed size images, each frame the Draw function must draw a sub-image moving the coordinates by the same amount, looping at the end:



Ebiten

Image animation

Each time Draw() is called, based on the tick we calculate the frame in the image and then we create a sub image calculating the rectangle to show. SubImage() returns an image.Image interface so we need a type assertion to draw.

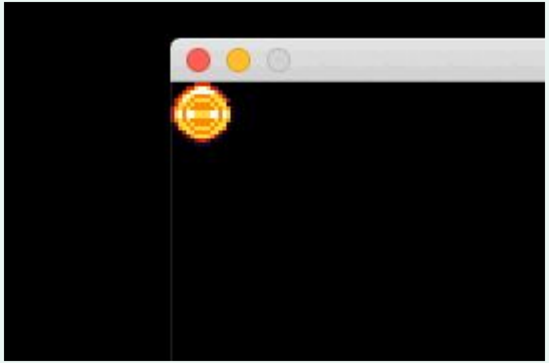


```
const (
    imgSize   = 16 // size in pixels, square img
    numFrames = 8 // number of frames in the spreadsheet
)
func (g *Game) Draw(screen *ebiten.Image) {
    op := &ebiten.DrawImageOptions{}
    frameNum := g.tick % numFrames
    // move right in the spreadsheet
    frameX := int(frameNum *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

Ebiten

Image animation

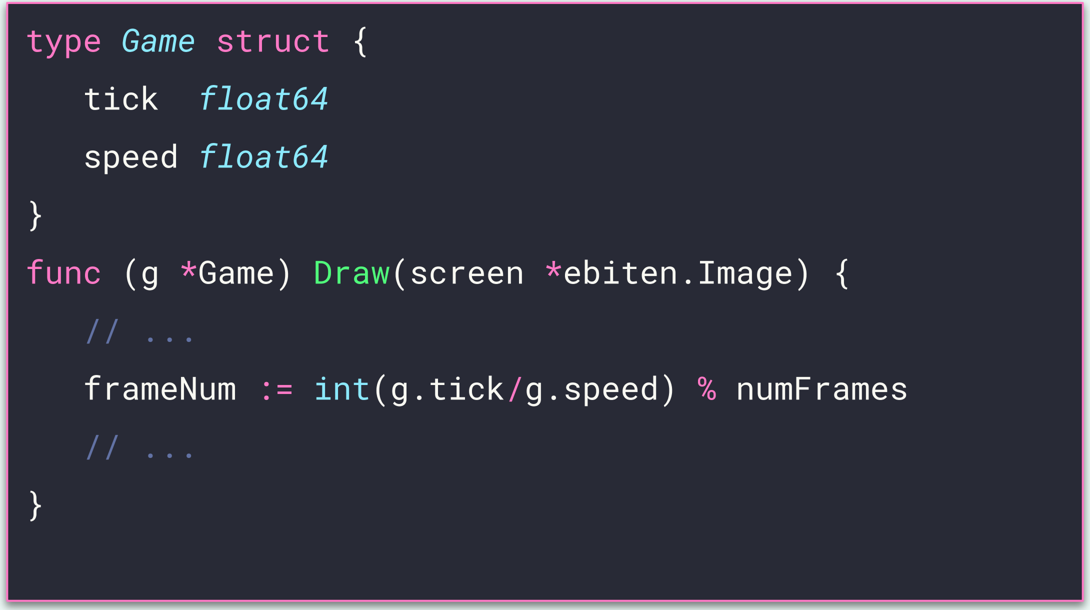
Almost done, except that the animation is too fast. In fact we're rendering ~60 ticks/frames per second (the Update() speed):



Ebiten

Image animation

Let’s add a speed value to the game:

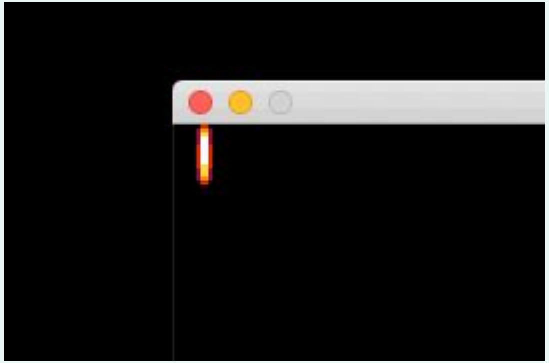


```go
type Game struct {
   tick  float64
   speed float64
}
func (g *Game) Draw(screen *ebiten.Image) {
   // ...
   frameNum := int(g.tick/g.speed) % numFrames
   // ...
}
```

Ebiten

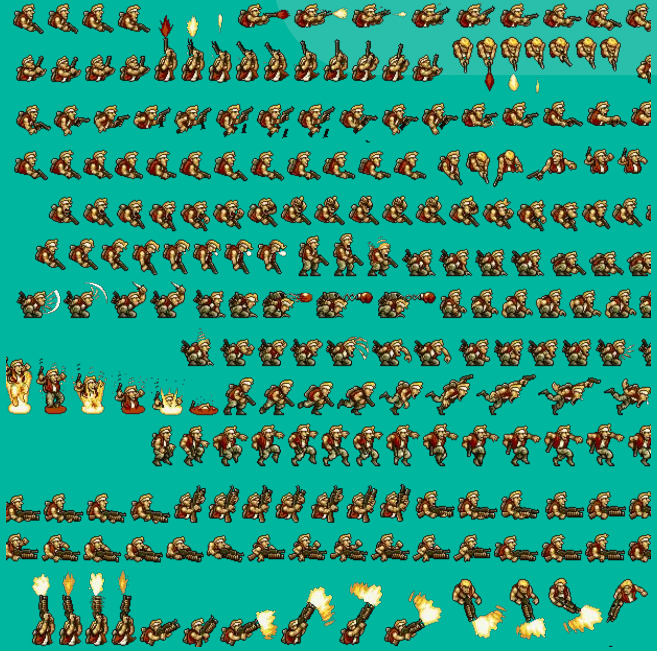
Image animation

Much better with speed at 60/6=10, or the number of TPS (60) divided by the number of frames that we want to show during 1 second:

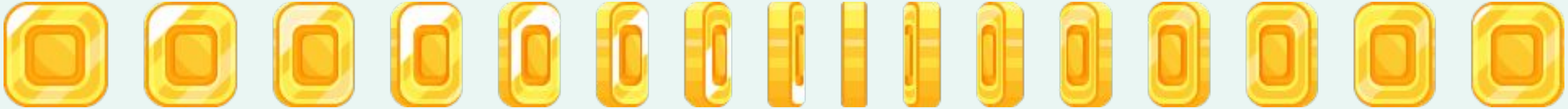


https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/03_tiles_fixed_size

Spritesheets



Things get a bit more complicated when frames have different sizes or are spread across a big image, in different positions (to optimize the final image size):



With images like this, you can* receive a specs file (like JSON) where for each frame you get x0 and y0 as well as width and height of the frame.

With those values you can build a image.Rect for each frame and use it to get a SubImage.

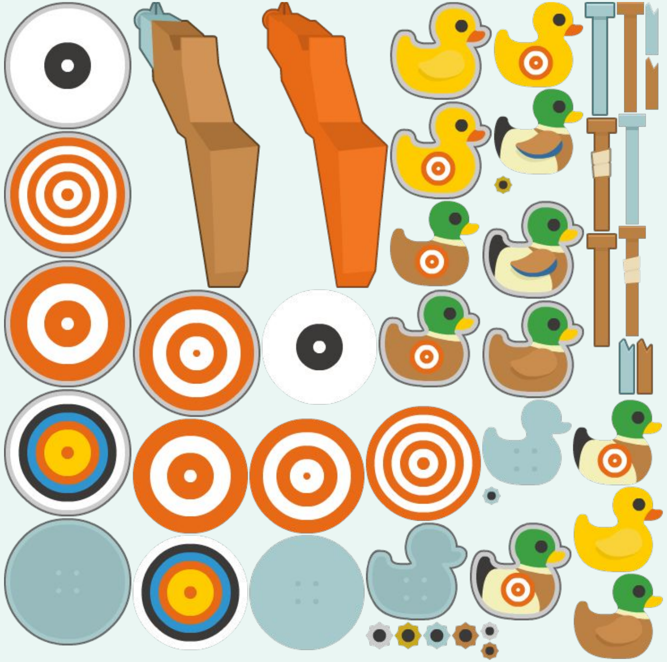
*if not, you probably need to build one yourself… 😨

Ebiten

Spritesheets

The spritesheet can contain frames of an animation or unique images (or both).

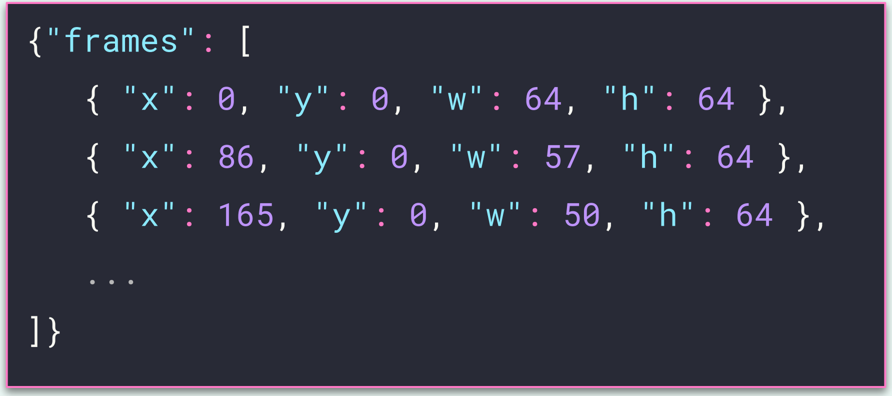
The use of the spritesheet reduces the final size (in bytes) required for all the assets.



Ebiten

Spritesheets

This is an example of JSON file with the spritesheet specs:

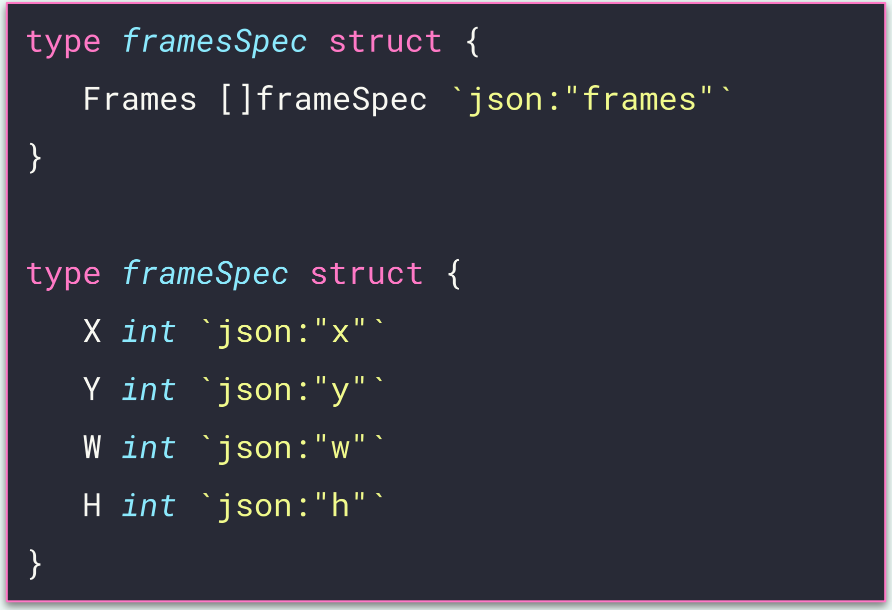


```
{"frames": [
   { "x": 0, "y": 0, "w": 64, "h": 64 },
   { "x": 86, "y": 0, "w": 57, "h": 64 },
   { "x": 165, "y": 0, "w": 50, "h": 64 },
   ...
]}
```

Let’s see how to process a spritesheet image with this JSON spec

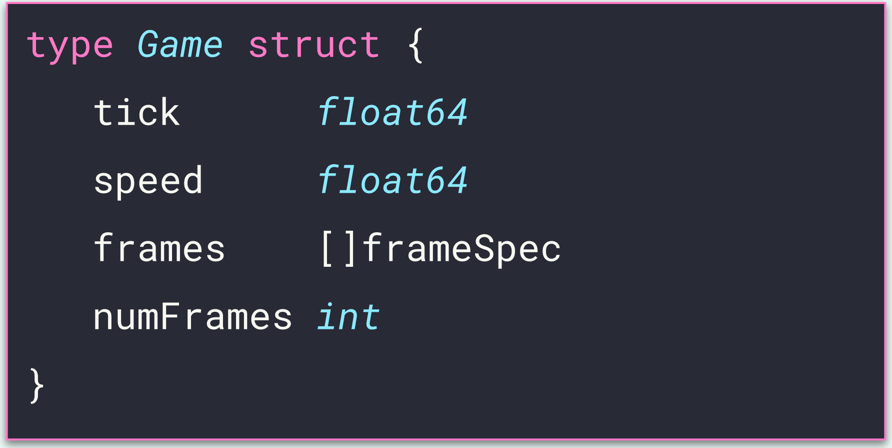
This is just an example, you can get different JSON structures and you can choose to parse them in different ways

We use 2 structs to “map” the JSON to Go objects:



```
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"`
}
```

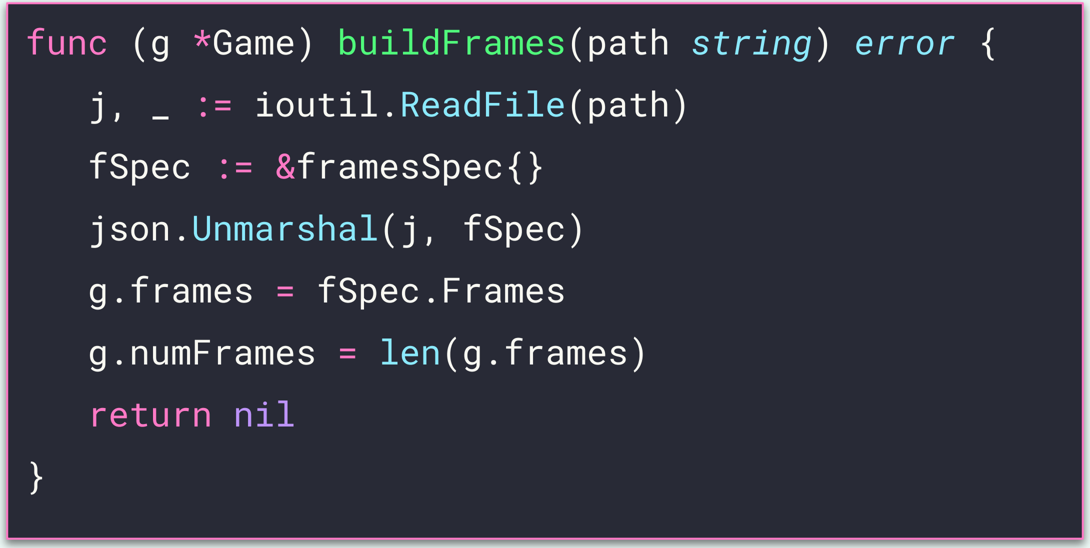
The Game gets the frames and their number:



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

Note that to make things simple I’m adding everything to the Game, but this obviously doesn’t scale and each image should have its own place

A new buildFrames() function parses the JSON specs to the Game frames:

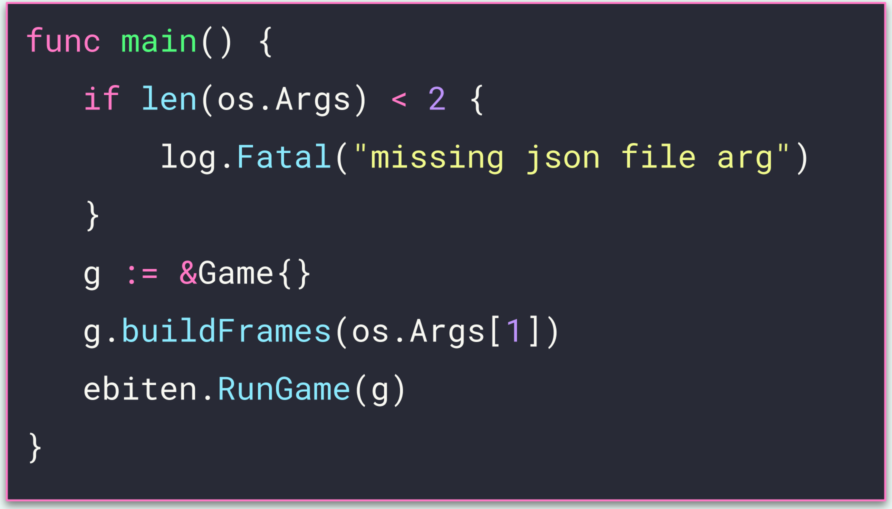


```
func (g *Game) buildFrames(path string) error {
    j, _ := ioutil.ReadFile(path)
    fSpec := &framesSpec{}
    json.Unmarshal(j, fSpec)
    g.frames = fSpec.Frames
    g.numFrames = len(g.frames)
    return nil
}
```

Ebiten

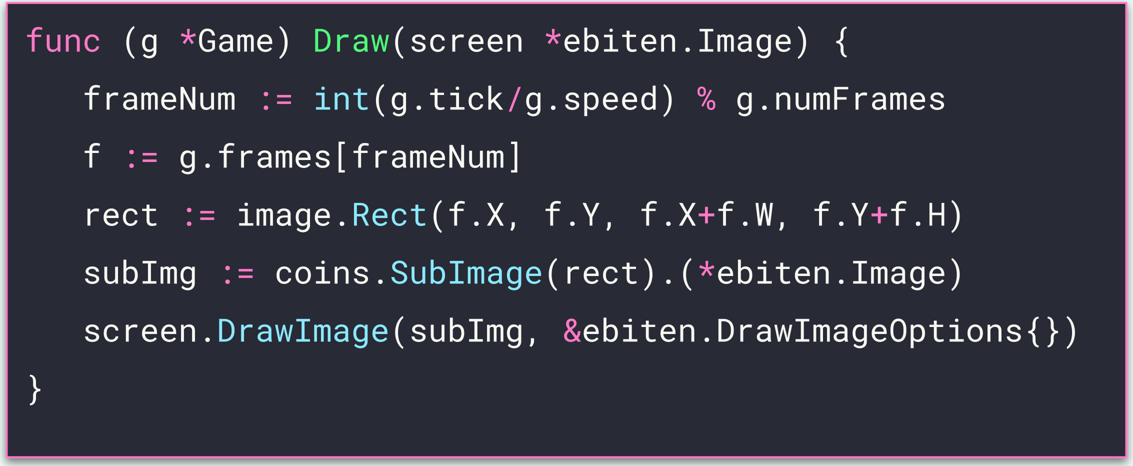
Spritesheets

The main() function gets the file as argument and passes it to buildFrames():



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

The Draw() function calculates the frame to show:

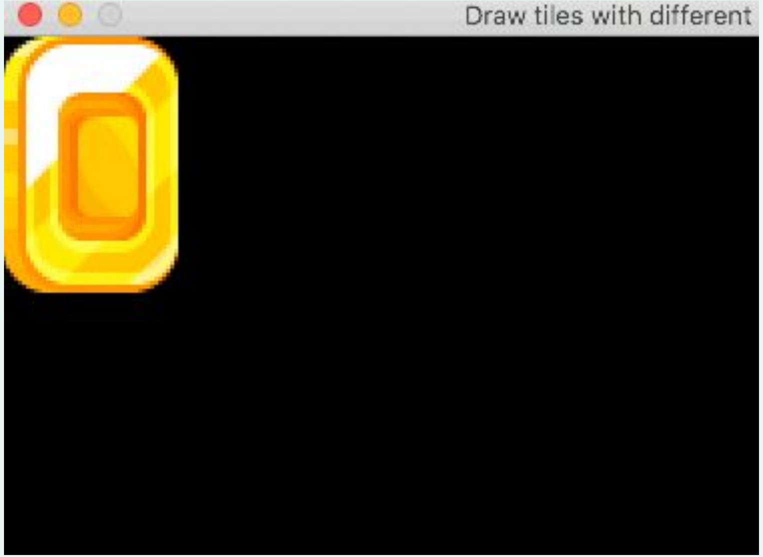


```go
func (g *Game) Draw(screen *ebiten.Image) {
   frameNum := int(g.tick/g.speed) % g.numFrames
   f := g.frames[frameNum]
   rect := image.Rect(f.X, f.Y, f.X+f.W, f.Y+f.H)
   subImg := coins.SubImage(rect).(*ebiten.Image)
   screen.DrawImage(subImg, &ebiten.DrawImageOptions{})
}
```

Ebiten

Spritesheets

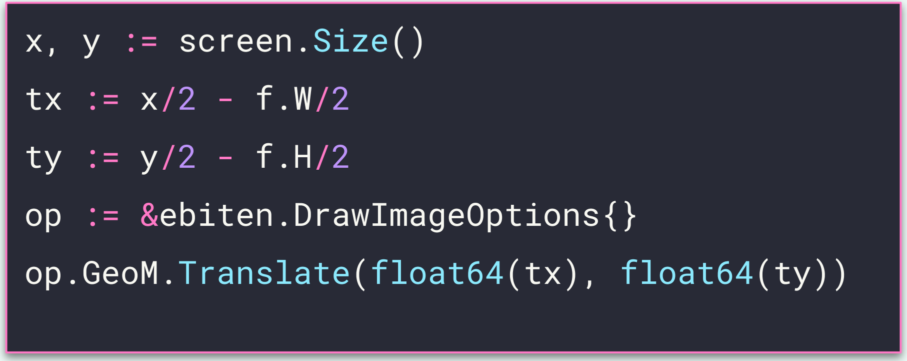
Almost there, but as the images have different sizes, the animation is wrong:



Ebiten

Spritesheets

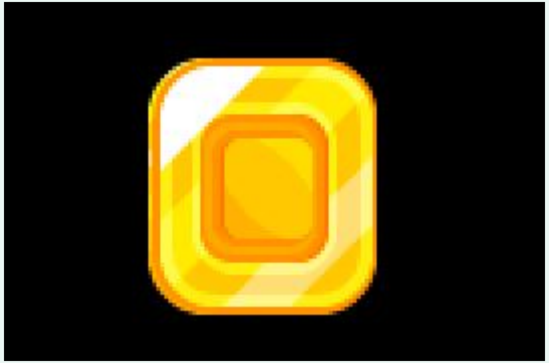
The solution is to move all images so they all have the same center:



```go
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))
```

The screen size can be replaced with any other position into it.

Now it is centered to the screen:



```
https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/04_tiles_vars
```

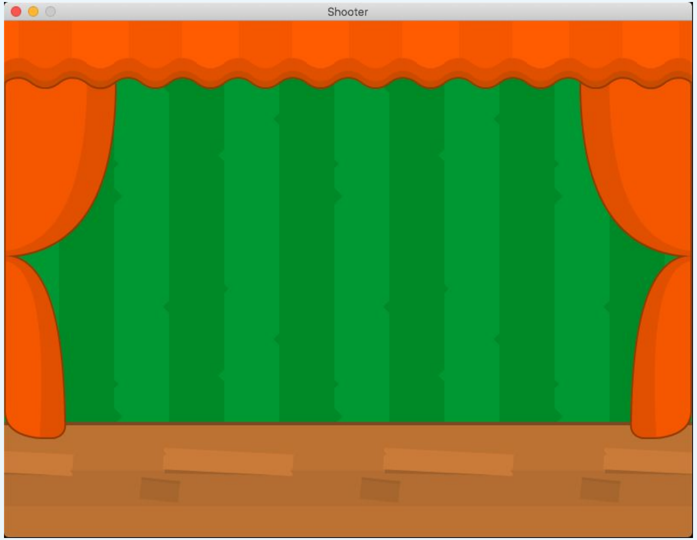
Exercise n.2

Add moving waves, generate ducks

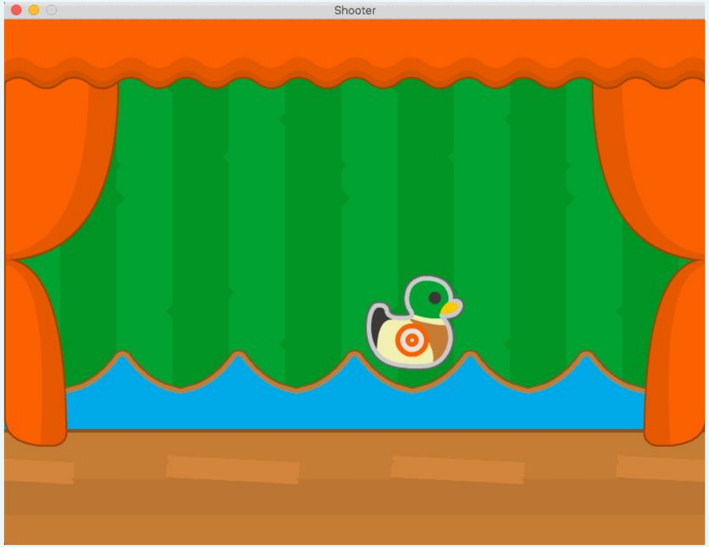
Second exercise

Add animations

What you have now



What you'll have then

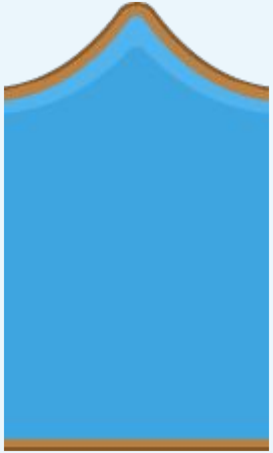


Second exercise

Add animations

Assets you need:

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

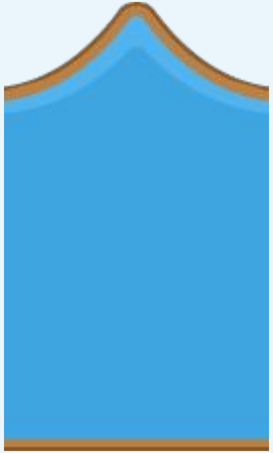


Second exercise

Add animations

Goals:

- Movements are now both in the x and y directions, up/down and right/left. You can use a +1/-1 multiplier to move on the opposite direction
- Waves must be glued horizontally to fill in the screen but also, as they move left and right, you must add extra images out of the screen, they'll become visible while moving

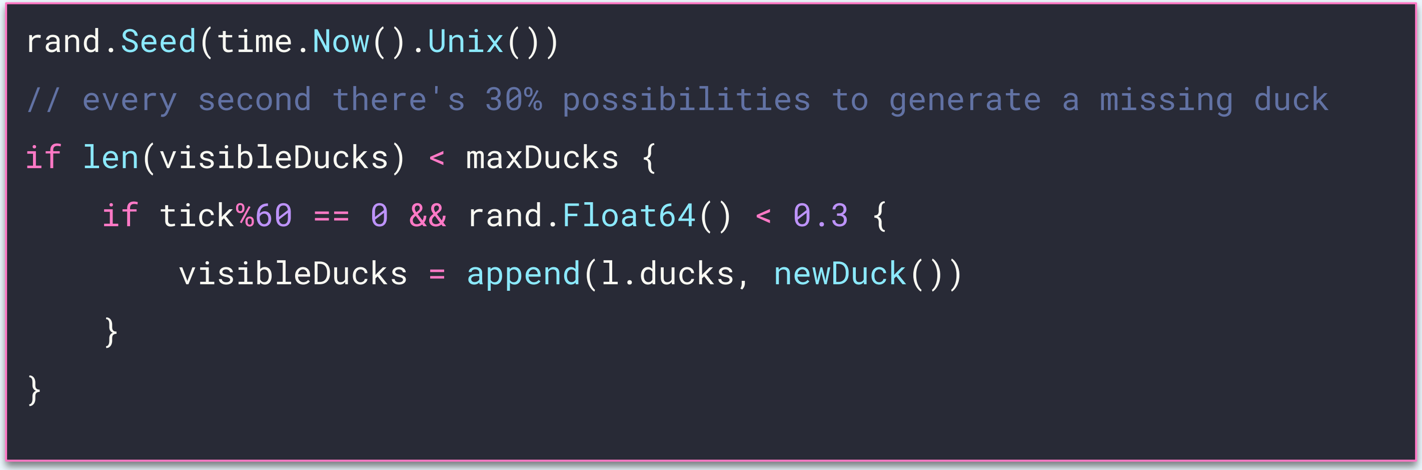


Second exercise

Add animations

Ducks move fast on the right, slow up and down

They can be generated “randomly” during `Update()`, this is an example

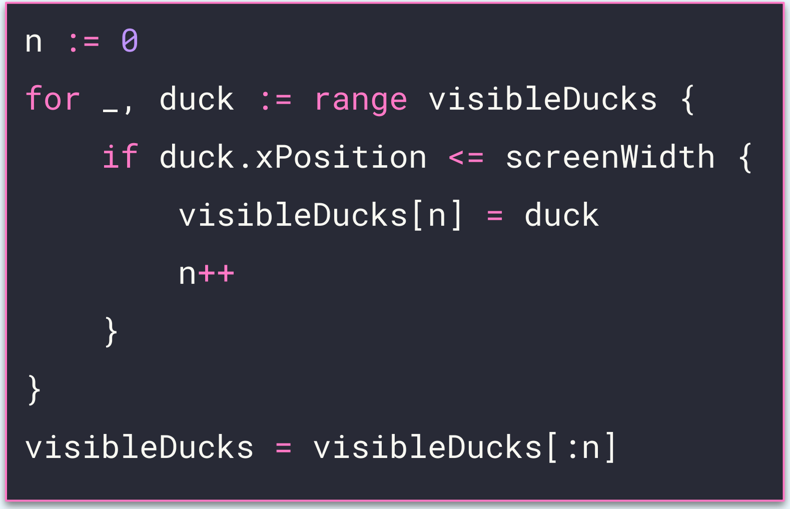


```go
rand.Seed(time.Now().Unix())
// every second there's 30% possibilities to generate a missing duck
if len(visibleDucks) < maxDucks {
    if tick%60 == 0 && rand.Float64() < 0.3 {
        visibleDucks = append(l.ducks, newDuck())
    }
}
```

Second exercise

Add animations

Check the X offset of the duck, when bigger than screen width, it’s off the screen and can be deleted:



```
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

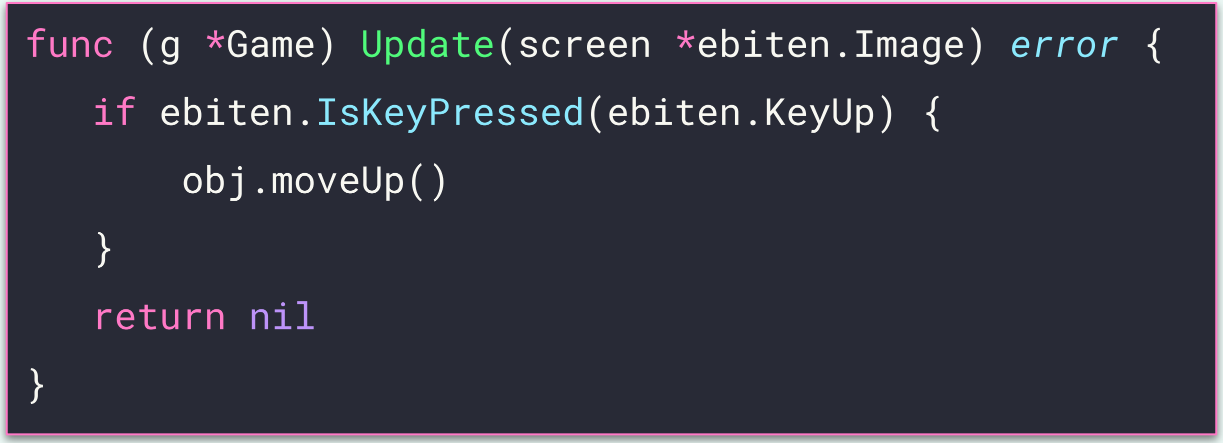
Extras

Some ideas:

- use images from spritesheets instead of single images
    - create a logic to get an image from spreadsheets using the image name
- constants (like speeds) could be extracted from functions to global constants, to ease adjusting their values
- add a stick below the duck, move them together
- ducks could also rotate a bit while moving

User input

Keyboard



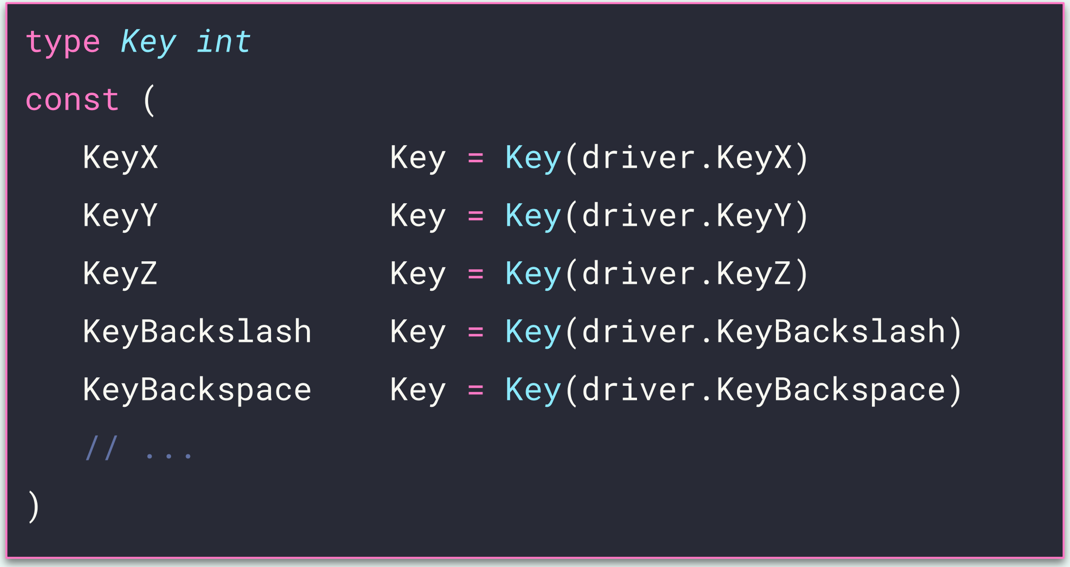
```
func (g *Game) Update(screen *ebiten.Image) error {
   if ebiten.IsKeyPressed(ebiten.KeyUp) {
       obj.moveUp()
   }
   return nil
}
```

`ebiten.IsKeyPressed(k Key) bool`

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

Ebiten

Keyboard input



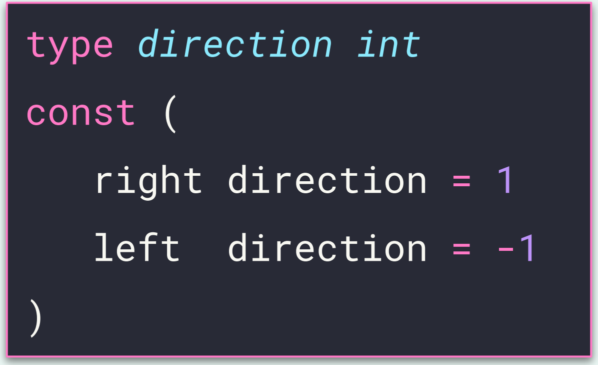
```go
type Key int
const (
    KeyX            Key = Key(driver.KeyX)
    KeyY            Key = Key(driver.KeyY)
    KeyZ            Key = Key(driver.KeyZ)
    KeyBackslash    Key = Key(driver.KeyBackslash)
    KeyBackspace    Key = Key(driver.KeyBackspace)
    // ...
)
```

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:

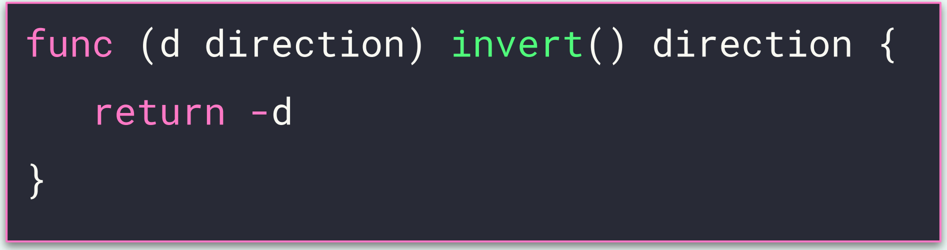


```go
type direction int
const (
    right direction = 1
    left  direction = -1
)
```

Ebiten

Define new types

We can also add behaviours to these types:



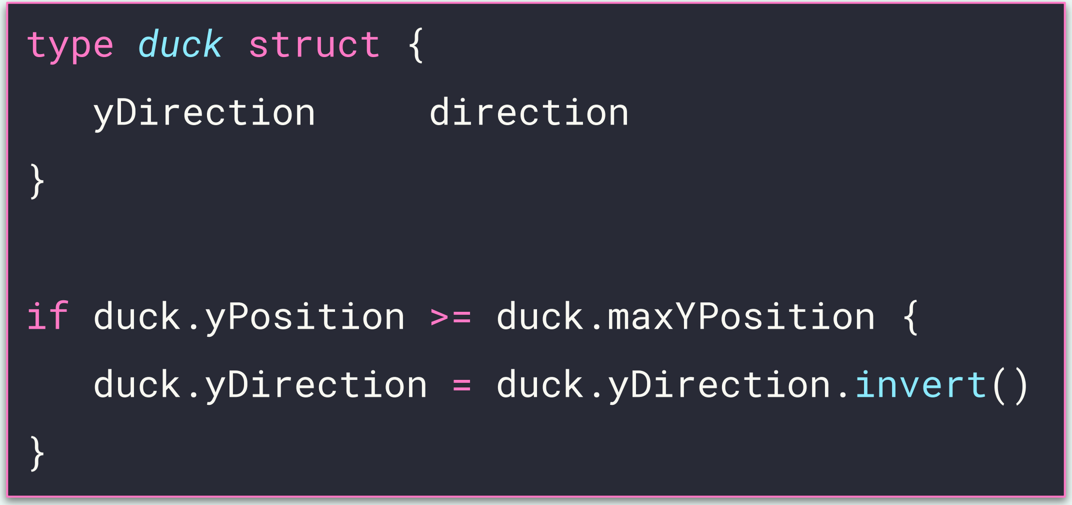
```go
func (d direction) invert() direction {
   return -d
}
```

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

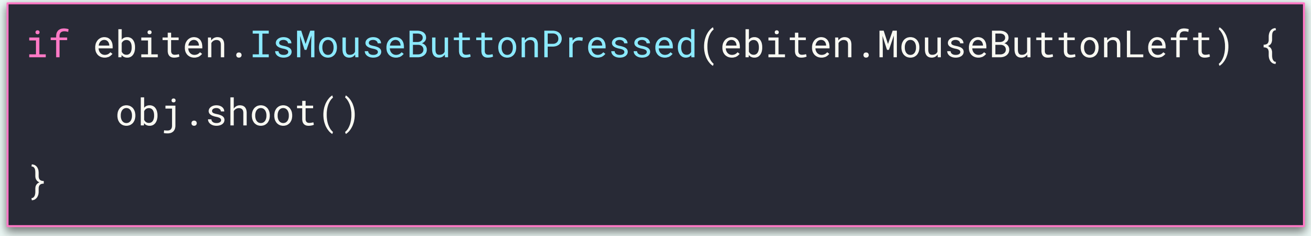
This is a small example that can apply to our game:



```
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:

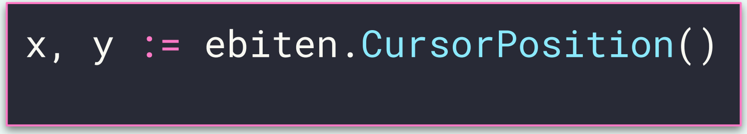


```go
if ebiten.IsMouseButtonPressed(ebiten.MouseButtonLeft) {
    obj.shoot()
}
```

Ebiten

Mouse input

The cursor position can be obtained with:



```
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`

Ebiten

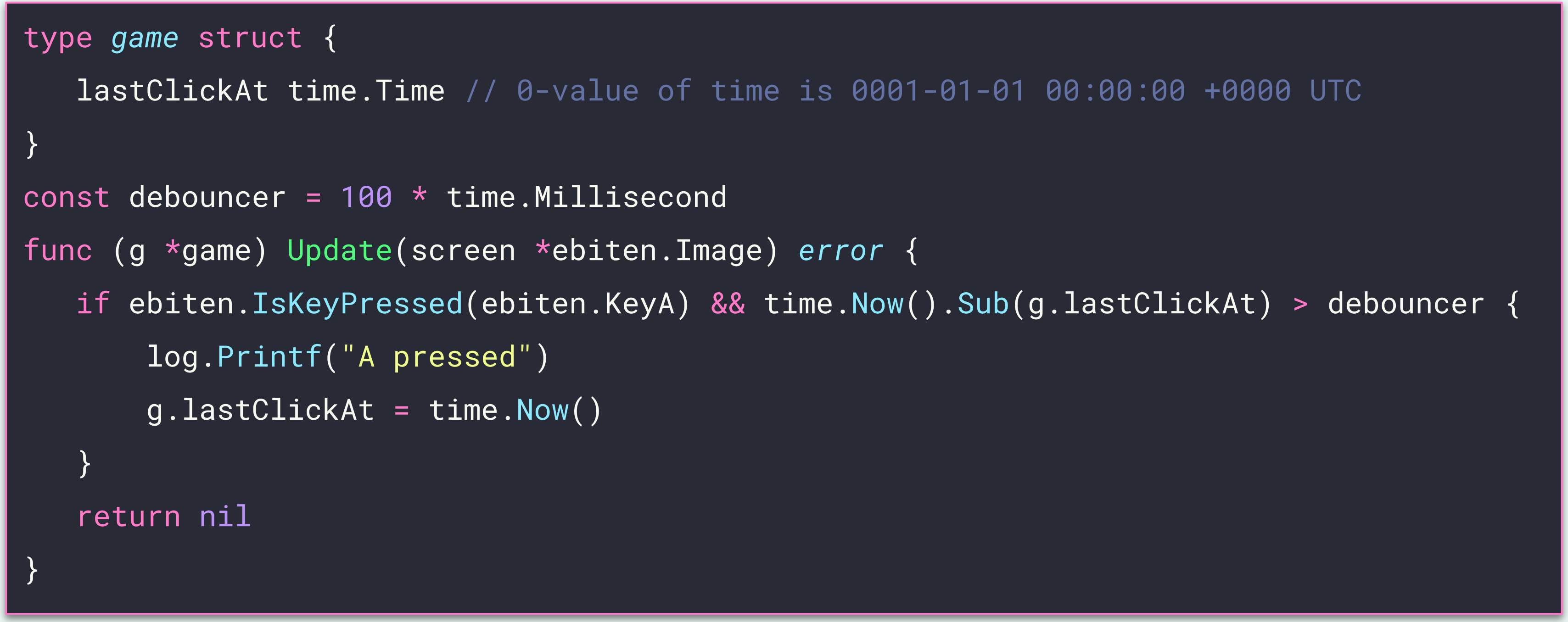
Mouse input

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



```go
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

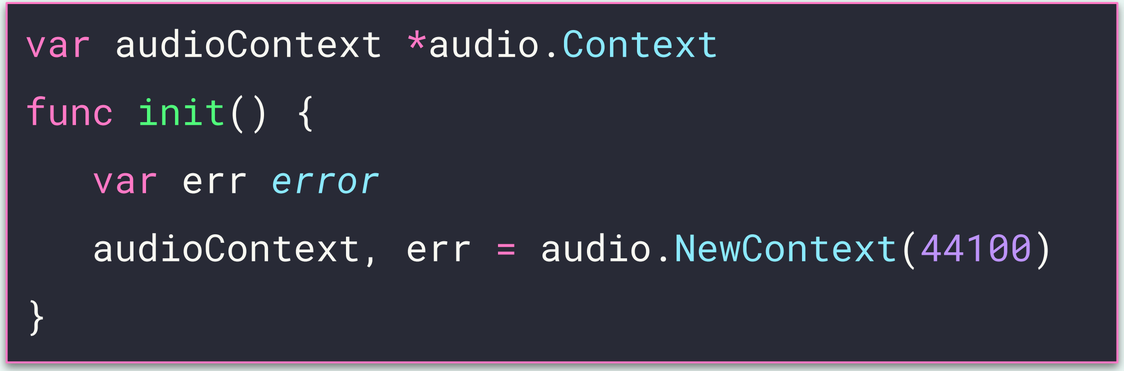
As for other assets, I suggest adding sounds as go files and using generators:

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

Ebiten

Sounds

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

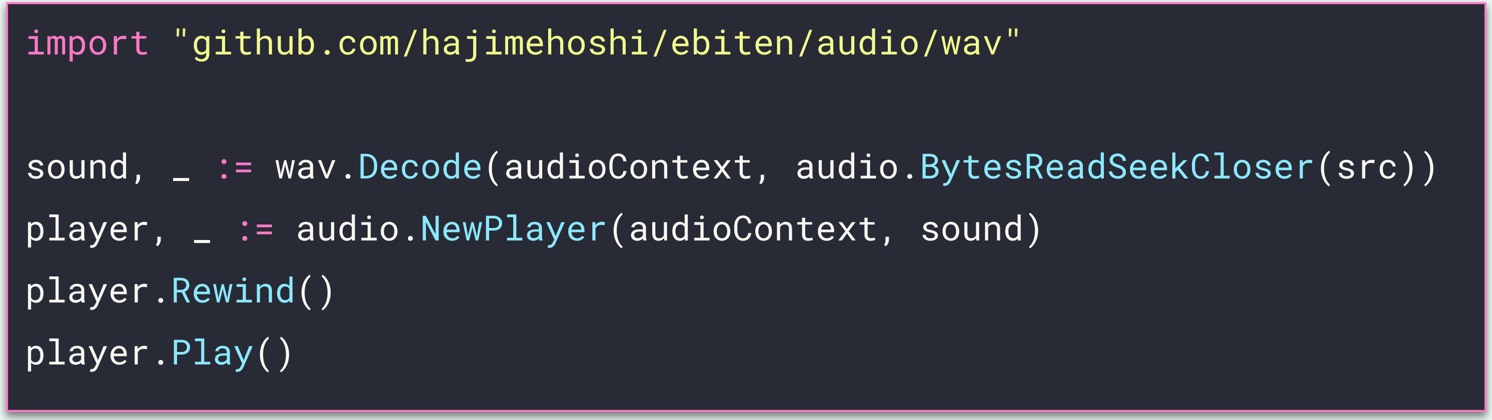
A background music could be played within an infinite loop, the file start-end must be mergeable without interruptions. Depending on the file, you’ll need different decoders.

```
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()
```

Ebiten

Sounds

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()
```

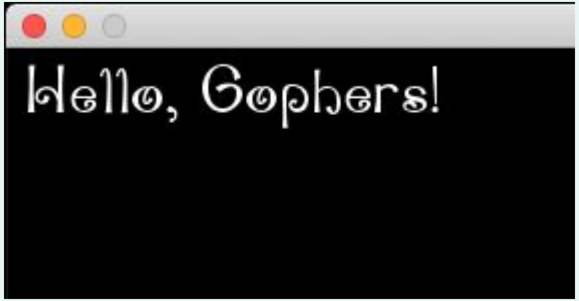
https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/06_sounds

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

Ebiten

Fonts

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
```

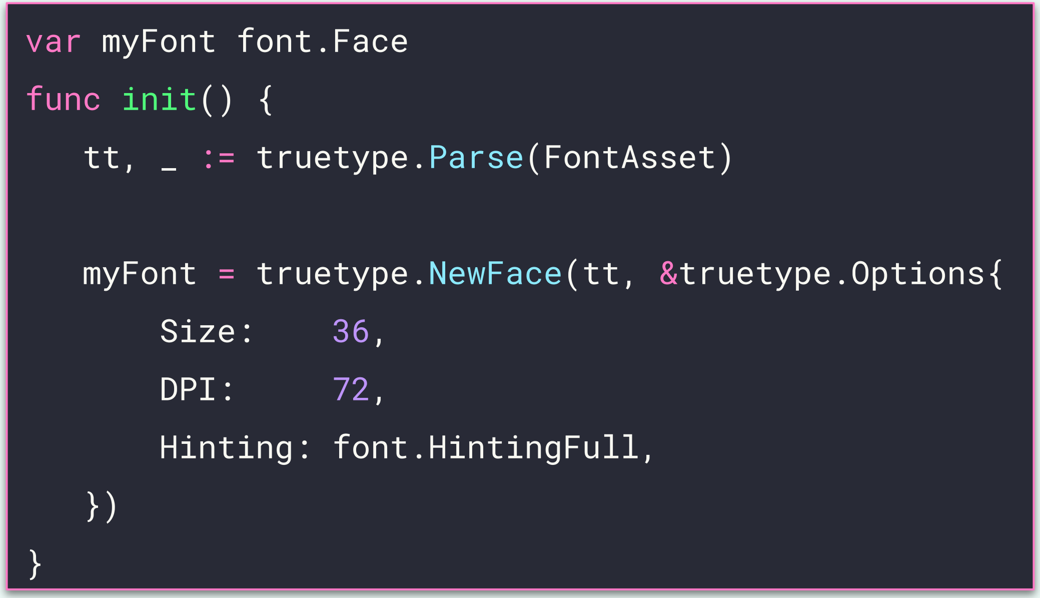
In my example the font is https://www.dafont.com/it/penguin-attack.font?l[]=10 (GPL)

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

Ebiten

Fonts

Then, load the font into the program:

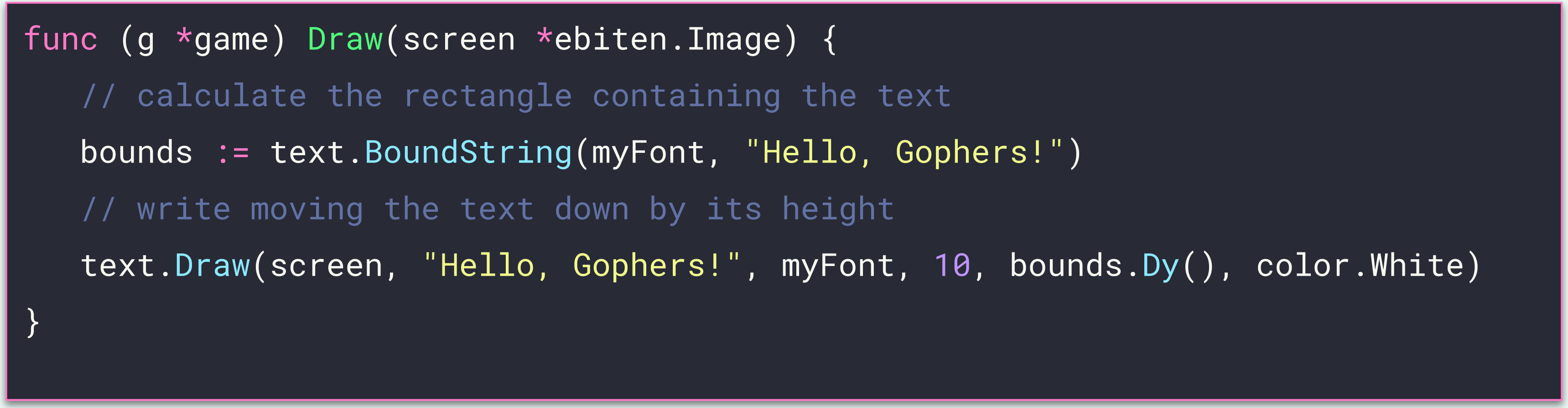


```go
var myFont font.Face
func init() {
    tt, _ := truetype.Parse(FontAsset)

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

Ebiten

Fonts

Now, we can write to the screen.

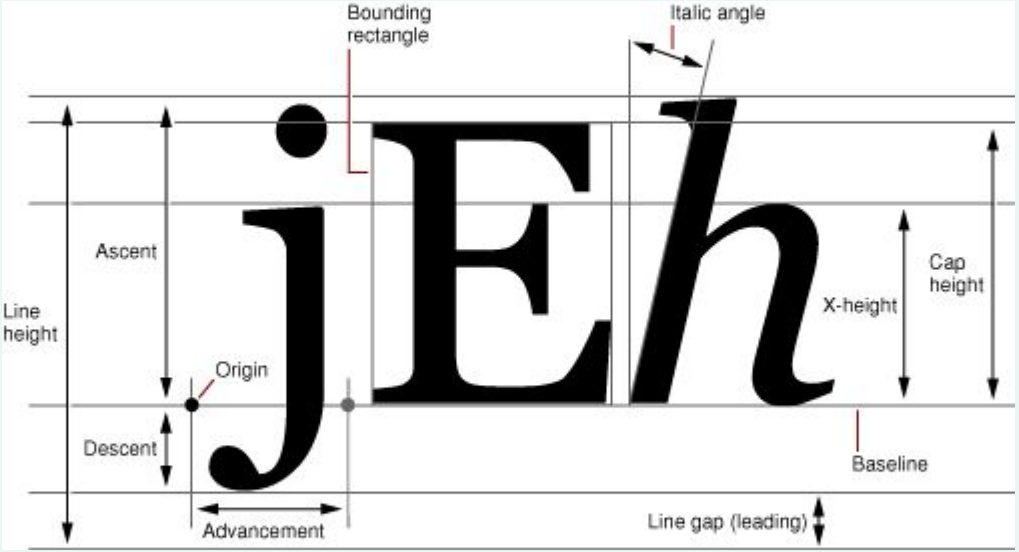


```go
func (g *game) Draw(screen *ebiten.Image) {
    // calculate the rectangle containing the text
    bounds := text.BoundString(myFont, "Hello, Gophers!")
    // write moving the text down by its height
    text.Draw(screen, "Hello, Gophers!", myFont, 10, bounds.Dy(), color.White)
}
```

`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)

Ebiten

Scenes

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

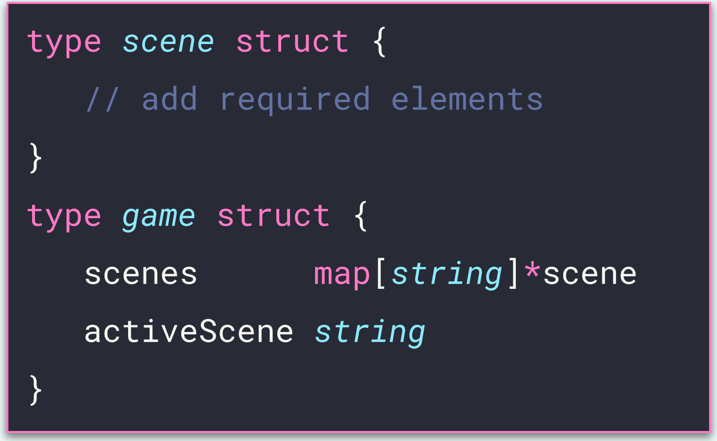
A scene completely changes the look and behaviour of the game and permits the user to move around

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

Ebiten

Scenes

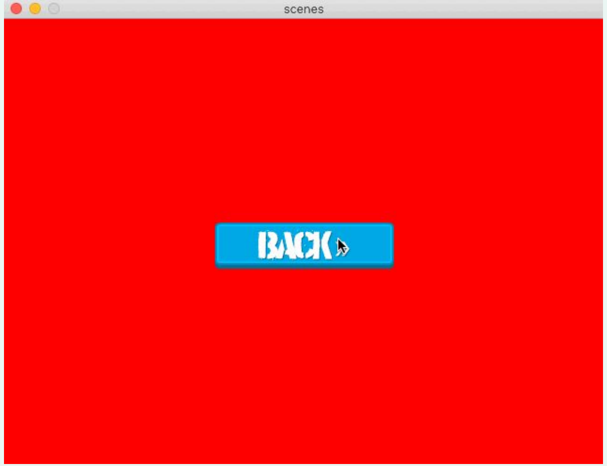
An idea could be to define a scene type with all you need to draw the scene and then leave the game to know which scene is active:



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

Ebiten

Scenes

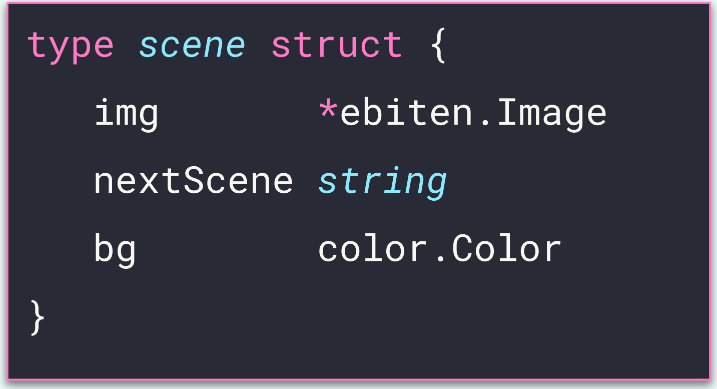


```
https://github.com/tommyblue/golab-2020-go-game-development/tree/master/examples/08_scenes
```

Ebiten

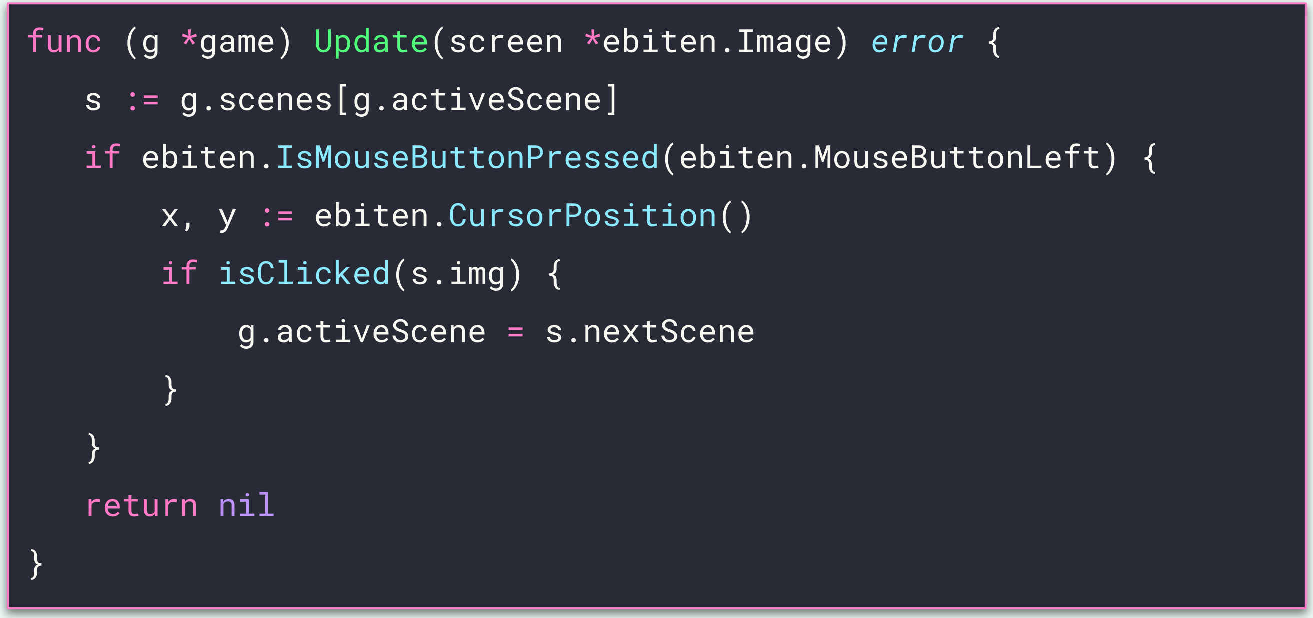
Scenes

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:

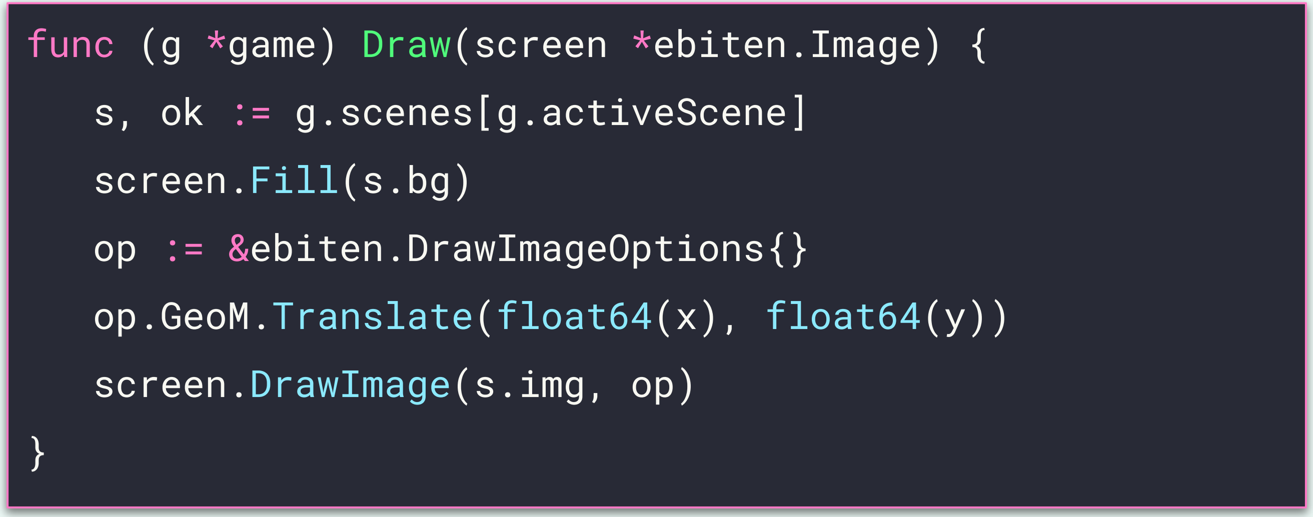


```go
func (g *game) Update(screen *ebiten.Image) error {
   s := g.scenes[g.activeScene]
   if ebiten.IsMouseButtonPressed(ebiten.MouseButtonLeft) {
       x, y := ebiten.CursorPosition()
       if isClicked(s.img) {
           g.activeScene = s.nextScene
       }
   }
   return nil
}
```

Ebiten

Scenes

`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)
}
```

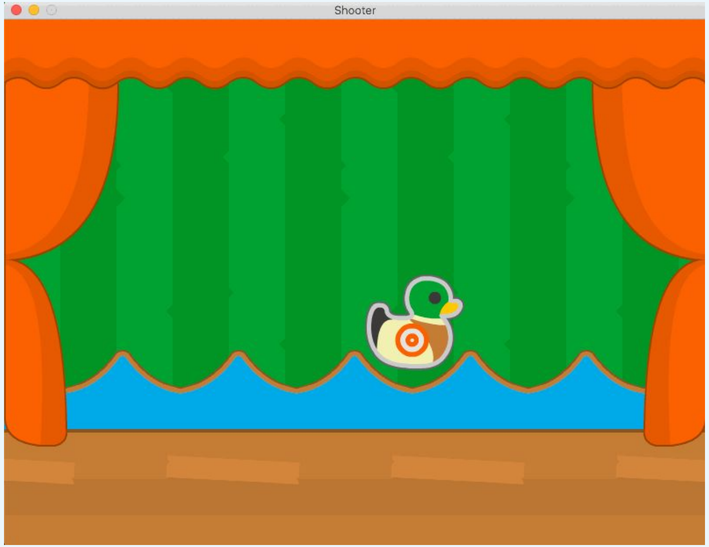
Exercise n.3

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

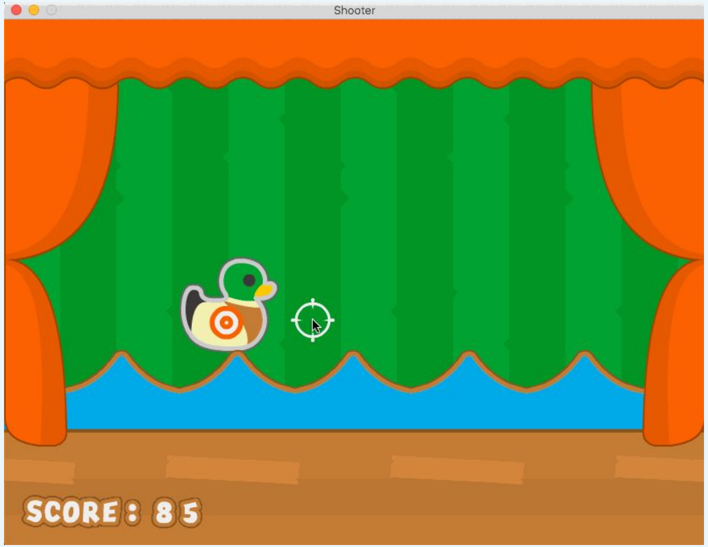
Third exercise

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:

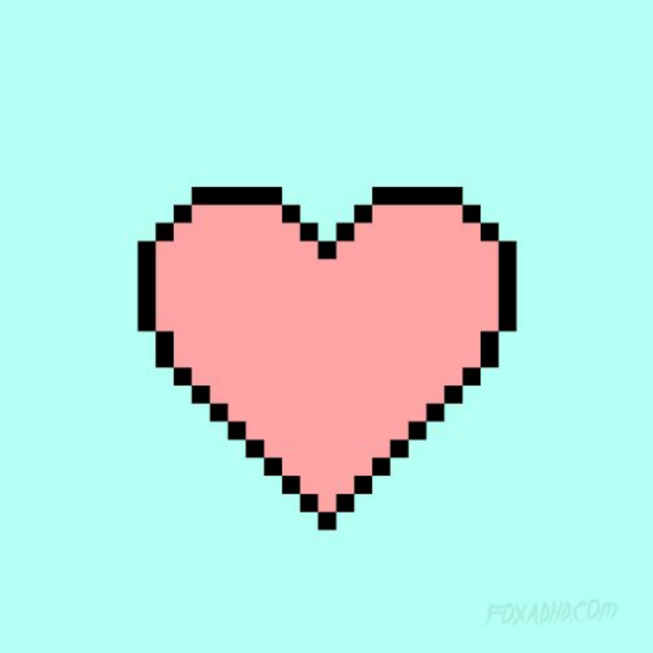
- PNG/HUD/crosshair_{white,red}_large.png
- Custom fonts or PNG/HUD/text_*.png
- hit.wav and miss.wav
- ragtime.ogg (background music)

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