Animations

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:

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

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

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:

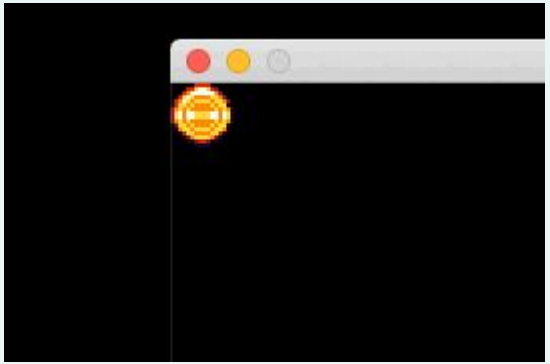
Ebiten

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)
}
```

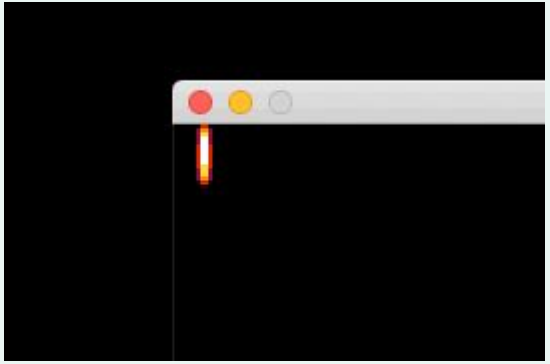
Almost done, except that the animation is too fast. In fact we're rendering ~60 ticks/frames per second (the Update() speed):



Let’s add a speed value to the game:

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

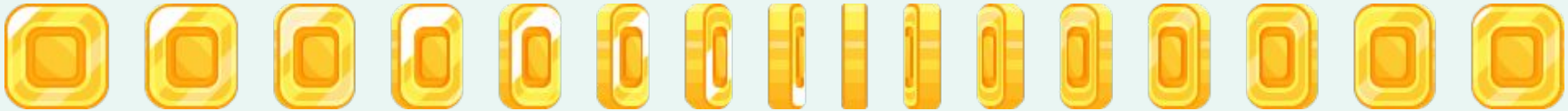
Much better with speed at 60/6=10, or the number of TPS (60) divided by the number of frames that we want to show during 1 second:



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… 😢

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.

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:

```
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
}
```

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

```
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:

```
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{})
}
```

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

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

```
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

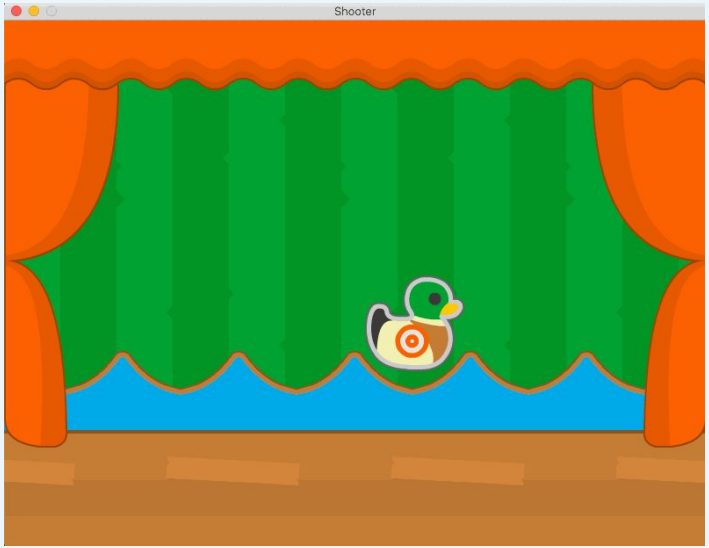
Exercise n.2

Add moving waves, generate ducks

Second exercise
Add animations

What you have now

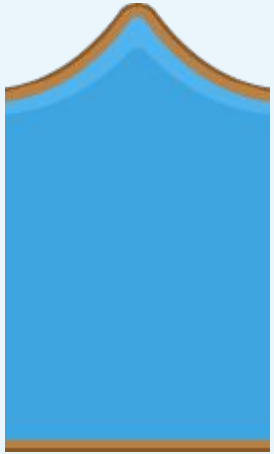
What you'll have then



Second exercise
Add animations

Assets you need:

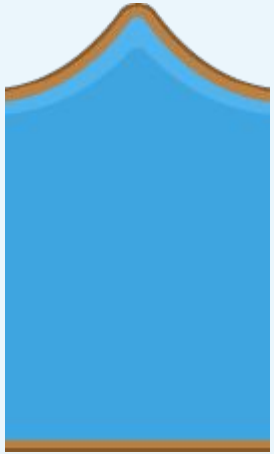
- PNG/Objects/duck_outline_target_white.png
- PNG/Stall/water1.png



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

```
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