A Self-Education Engineering Series

Game Engine Zero

Volume 1: Three Arcade Games

Make Games in Go, and the Engine They Need

Early Access edition. Volumes 1–3, 24 chapters. Built 2026-09-19.
Free to read online at kryolabs.duckdns.org/zero. Buying this file supports the writing of the remaining volumes.

VOLUME 1

Three Arcade Games

Go and Ebitengine from zero, ending in Pong, Snake and Breakout

Three arcade games you wrote yourself, from an empty Go module: a window and a game loop, drawing, input, sprites, text and sound, then Pong, Snake and Breakout, each a playable game in one chapter.

VOLUME 1 · CHAPTER 1

The Game Loop

Installing Go and Ebitengine, then a rectangle that moves across a window

1. The game loop

This chapter builds a window with a white rectangle in it. The rectangle moves from left to right, two pixels at a time, and comes back in from the left when it goes off the right edge.

A game loop is a loop that runs until the game closes. Each pass updates the game's state and then draws a picture of it. Ebitengine, the library this book uses, runs the loop. It calls your `Update` method sixty times a second and your `Draw` method each time the display wants a new picture. A third method, `Layout`, tells it how big the picture is.

2. Installing Go

⚙ TOOL — GO 1.26 AND A BASH TERMINAL

Download Go from `go.dev/dl` and follow the install steps at `go.dev/doc/install`. On Linux, unpack the archive into `/usr/local`, add `/usr/local/go/bin` to your `PATH`, and open a new terminal.

Every command in this book is a Bash command. On Windows, install Git for Windows and use its Git Bash terminal.

▣ BUILD · STAGE 1 — CHECK THE INSTALL

```
go version
mkdir -p ~/scratch
cd ~/scratch


// hello.go — create
package main

import (
        "fmt"
        "runtime"
)

func main() {
        fmt.Println("go runs on",
runtime.GOOS+"/"+runtime.GOARCH)
}
```

```
$ go version

go version go1.26.7 linux/amd64
$ go run hello.go

go runs on linux/amd64
```

Your `go version` line ends with your own patch number, operating system and processor: a Mac prints `darwin/arm64`, Windows prints `windows/amd64`. If the shell cannot find `go`, open a new terminal.

`package main` marks a file that belongs to a program. `func main` is where the program starts. The import block names the packages the file uses. `go run` compiles the file and runs it. If this program runs, Go is installed.

▣ BUILD · STAGE 2 — THE GEZ MODULE

```
cd ~
mkdir gez
cd gez
go mod init gez
go mod edit -go=1.26
cat go.mod
```

```
$ go mod init gez

go: creating new go.mod: module gez
$ cat go.mod

module gez

go 1.26
```

A Go project is a module: a directory with a `go.mod` file at its root. The file names the module and the Go version it is written for. `go mod init` writes your exact toolchain version, such as `1.26.7`; `go mod edit -go=1.26` changes it to the release. Every `go` command from here on is typed inside `~/gez`.

◆ NOTE — THE EDITOR

Any text editor works. VS Code with its Go extension marks errors as you type and formats on save.

3. What Ebitengine needs on your system

On Linux and macOS, Ebitengine is compiled against C libraries for the window system and the sound card. Install them before the next stage, or the first compile fails with an error about a missing header.

⚙ TOOL — SYSTEM PACKAGES FOR EBITENGINE

- **Linux** (Debian, Ubuntu): a C compiler and the OpenGL, X11 and ALSA headers.

```
sudo apt install gcc pkg-config libgl1-mesa-dev xorg-dev
libasound2-dev
```

  Other distributions' package names are at `ebitengine.org/en/documents/install.html`.
- **macOS**: the Xcode command-line tools, installed with `xcode-select --install`.

- **Windows**: nothing beyond Go.

4. The Game interface

Ebitengine runs any value that has three methods: `Update`, `Draw` and `Layout`. In Go, a set of method names and signatures is an interface; this one is `ebiten.Game`.

▣ BUILD · STAGE 3 — A WINDOW WITH ONE RECTANGLE

```
// cmd/window/main.go — create
package main

import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

// The picture the game draws is 320 pixels wide and 180 tall.
The window
// is a separate matter.
const (
        screenW = 320
        screenH = 180
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
)

// Game is what Ebitengine runs: one value with the three
methods below.
type Game struct{}

// Update runs sixty times a second. Nothing changes yet.
func (g *Game) Update() error {
        return nil
}

// Draw runs whenever the display wants a picture.
func (g *Game) Draw(screen *ebiten.Image) {
        screen.Fill(courtColor)
```

```
        vector.FillRect(screen, 158, 78, 4, 24, lineColor,
false)
}

// Layout is told the window's size and answers with the
picture's.
func (g *Game) Layout(outsideWidth, outsideHeight int) (int,
int) {
        return screenW, screenH
}

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("A window")
        ebiten.SetTPS(60)
        if err := ebiten.RunGame(&Game{}); err ≠ nil {
                log.Fatal(err)
        }
}


go get github.com/hajimehoshi/ebiten/v2@v2.9.11
go mod tidy
go mod vendor
ls vendor
cat go.mod
go vet ./...
go run ./cmd/window
```

```
$ go get github.com/hajimehoshi/ebiten/v2@v2.9.11

go: added github.com/ebitengine/gomobile v0.0.0-20250923094054-
ea854a63cce1
go: added github.com/ebitengine/hideconsole v1.0.0
go: added github.com/ebitengine/purego v0.9.0
go: added github.com/hajimehoshi/ebiten/v2 v2.9.11
go: added github.com/jezek/xgb v1.1.1
go: added golang.org/x/sync v0.21.0
go: added golang.org/x/sys v0.44.0
$ go mod tidy

$ go mod vendor

$ ls vendor

github.com
golang.org
modules.txt
$ cat go.mod

module gez
```

```
go 1.26

require github.com/hajimehoshi/ebiten/v2 v2.9.11

require (
        github.com/ebitengine/gomobile v0.0.0-20250923094054-
ea854a63cce1 // indirect
        github.com/ebitengine/hideconsole v1.0.0 // indirect
        github.com/ebitengine/purego v0.9.0 // indirect
        github.com/jezek/xgb v1.1.1 // indirect
        golang.org/x/sync v0.21.0 // indirect
        golang.org/x/sys v0.44.0 // indirect
)
$ go vet ./...
```

The game after stage 3: the court colour, with a white rectangle in the middle.

`go get` adds the named release of Ebitengine to `go.mod`, with the six modules it depends on marked `indirect`. Your run may print seven `go: downloading` lines first. `go mod tidy` gives Ebitengine its own `require` line. `go mod vendor` copies all seven modules into a `vendor` directory, and every build from now on uses that copy. `go vet ./...` checks every package in the module and prints nothing when it passes. Run it before every `go run` in this book.

`go run ./cmd/window` opens a window titled *A window*, 960 by 540, with the rectangle in the middle. The terminal prints nothing. Close the window to stop the program. Every chapter in this volume adds its own directory under `cmd/`.

`Draw` receives `screen`, an image 320 pixels wide and 180 tall, and paints it with two Ebitengine calls. `screen.Fill` paints the whole image one colour. `vector.FillRect` paints a rectangle with its top-left corner at column 158, row 78, 4 wide and 24 tall; the last argument, `false`, keeps the edges hard. `RunGame` takes a pointer to a new `Game`, opens the window and runs the loop until the window closes. The reference for every Ebitengine call is `pkg.go.dev/github.com/hajimehoshi/ebiten/v2`.

5. Ticks and draws

A tick is one call to `Update`. `SetTPS(60)` asks for sixty ticks a second, so a tick is one sixtieth of a second of game time. A draw is one call to `Draw`, made at the display's rate, which can be higher or lower than the tick rate.

▣ BUILD · STAGE 4 — MOVING THE RECTANGLE IN UPDATE

```
// cmd/window/main.go — replace
package main

import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

// The picture the game draws is 320 pixels wide and 180 tall.
The window
// is a separate matter.
const (
        screenW = 320
        screenH = 180
)
```

```
var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
)

// Game is what Ebitengine runs: one value with the three
methods below.
type Game struct {
        x float64 // the rectangle's left edge, in the picture's
pixels
}

// Update runs sixty times a second. Everything that changes,
changes here.
func (g *Game) Update() error {
        g.x += 2
        if g.x ⩾ screenW {
                g.x = -4 // off the right edge: come back in
from the left
        }
        return nil
}

// Draw runs whenever the display wants a picture, and paints
what Update decided.
func (g *Game) Draw(screen *ebiten.Image) {
        screen.Fill(courtColor)
        vector.FillRect(screen, float32(g.x), 78, 4, 24,
lineColor, false)
}

// Layout is told the window's size and answers with the
picture's.
func (g *Game) Layout(outsideWidth, outsideHeight int) (int,
int) {
        return screenW, screenH
}

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("A window")
        ebiten.SetTPS(60)
        if err := ebiten.RunGame(&Game{x: 158}); err ≠ nil {
                log.Fatal(err)
        }
}


go vet ./...
go run ./cmd/window
```

The game one second after it starts: sixty ticks have moved the rectangle from column 158 to column 278.

Four things changed: the field `x`, the body of `Update`, the first argument to `FillRect`, and the starting value in `main`. `Update` adds two to `x` on every tick. At 320 the rectangle is off the right edge, so `Update` sets `x` to -4 and the rectangle slides in from the left. `Draw` paints the rectangle where `Update` put it.

Everything that moves, moves in `Update`. `Draw` only paints what `Update` decided. That rule holds for every game in this book.

The rectangle moves 120 pixels a second, sixty ticks of two pixels, on every machine. `x` is a `float64` because later games move by half a pixel a tick. `FillRect` takes a `float32`, so the position is converted there.

⚠ WORKED FAILURE — THE RECTANGLE MOVED IN DRAW

The lines that move the rectangle compile and run in `Draw` too. Take them out of `Update` and put them at the top of `Draw`:

```
func (g *Game) Update() error {
        return nil
}
```

```
func (g *Game) Draw(screen *ebiten.Image) {
        g.x += 2
        if g.x ⩾ screenW {
                g.x = -4
        }
        screen.Fill(courtColor)
        vector.FillRect(screen, float32(g.x), 78, 4, 24,
lineColor, false)
}
```

On a 60 Hz display with nothing else running, the rectangle moves at about the same speed as before. The two pictures below are the same program after sixty ticks on two other displays: one that draws three times a tick, as a 180 Hz monitor does, and one that draws every other tick, as a busy laptop does.

Sixty ticks with the movement in Draw, three draws a tick: 180 steps of two pixels and one wrap put the rectangle at column 194.

The same sixty ticks, one draw every two ticks: thirty steps put the rectangle at column 218.

The rectangle now moves two pixels every draw, and the display decides how many draws there are. After sixty ticks it is at column 194 on one display and 218 on the other; with the movement in `Update` it is at 278 on every machine. The game has no speed of its own. Move the lines back to `Update`.

6. The picture and the window

The picture is 320 by 180. The window is 960 by 540, three times larger. Ebitengine calls `Layout` with the window's size, makes the image it hands to `Draw` at the size `Layout` returns, and scales that image up to fill the window. So the rectangle, 4 pixels wide in the picture, shows 12 pixels wide on screen. Drag the window's corner to any size and the picture scales with it; the game never learns the window changed. A game that counted in window pixels would have to rescale everything each time.

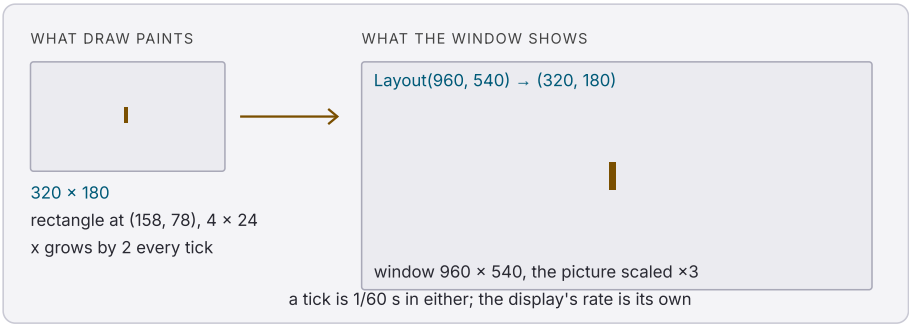


Figure 1.1 — the picture and the window. `Draw` paints 320 by 180. `Layout` returns that size whatever the window is. Ebitengine scales the picture to the window.

7. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Install Go and check it with `go version` and a small program.
- Create the `gez` module, add Ebitengine at `v2.9.11` and vendor it.
- Write a type with `Update`, `Draw` and `Layout`, hand it to `ebiten.RunGame`, and say how often each method is called.
- Move a rectangle two pixels a tick in `Update` and wrap it at the edge.
- Explain why the same lines in `Draw` put the rectangle in a different place on every display.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — a second rectangle. Add a second rectangle that starts at column 20 and moves one pixel a tick. Which one reaches the right edge first?**

Add a second field, a second `+=` in `Update` and a second `FillRect` in `Draw`. The first rectangle reaches 320 on tick 81, the second on tick 300, five seconds after the window opens.

▼ **Exercise 2 — a taller picture. Change `Layout` to return 320 by 360 and run the program. What happens to the window, and what happens to the rectangle?**

The window stays 960 by 540. Ebitengine fits the taller picture inside it, scaled by 1.5 and centred, with dark bands at the sides. The rectangle sits in the top quarter, six pixels wide on screen instead of twelve. `Update` and `Draw` did not change.

▼ **Exercise 3 — thirty ticks a second. Change `SetTPS(60)` to `SetTPS(30)`. Predict the rectangle's speed, then run it.**

Half the speed: sixty pixels a second, so the rectangle takes 2.7 seconds from the middle to the edge instead of 1.35. `Draw` still runs at the display's rate, so the rectangle moves in visible two-pixel steps thirty times a second.

VOLUME 1 · CHAPTER 2

Drawing

Rectangles, circles, lines, an image, and the matrix that places it

1. Drawing the court

This chapter draws the court the games in this volume are played on, one of each shape Ebitengine draws by hand, and a small image placed at any size and angle. The work goes in a new program, `cmd/shapes`.

The court is three calls: a fill, a border stroked two pixels wide just inside the picture's edge, and a row of dashes down the middle.

▣ BUILD · STAGE 1 — THE COURT, CREATE CMD/SHAPES/MAIN.GO

```
// cmd/shapes/main.go — create
package main

import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

const (
        screenW = 320
        screenH = 180
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
)

// drawCourt paints the court: the fill, a border two pixels
wide just
```

```
// inside the edge, and a net of dashes down the middle.
func drawCourt(screen *ebiten.Image) {
        screen.Fill(courtColor)
        vector.StrokeRect(screen, 1, 1, screenW-2, screenH-2, 2,
lineColor, false)
        for y := 0; y < screenH; y += 8 {
                vector.FillRect(screen, 159, float32(y), 2, 4,
lineColor, false)
        }
}

type Game struct{}

func (g *Game) Update() error {
        return nil
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawCourt(screen)
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Shapes")
        ebiten.SetTPS(60)
        if err := ebiten.RunGame(&Game{}); err ≠ nil {
                log.Fatal(err)
        }
}


go vet ./...
go run ./cmd/shapes
```

The window after stage 1: the court, its border and its net.

Each chapter's program gets its own directory under `cmd/`, so this one is `cmd/shapes`. The constants, the three methods and `main` are chapter 1's, with a new window title.

`screen.Fill` paints every pixel of the picture one colour. `vector.StrokeRect` paints a line along the outline of a rectangle. The stroke straddles that outline, half its width to either side. This one runs round the rectangle from (1, 1) to (319, 179) at two pixels wide, so it covers columns 0 and 1 on the left, 318 and 319 on the right, and the matching rows top and bottom. That is the two outermost pixels all round, and nothing further in.

The net is drawn with `vector.FillRect`, the call chapter 1 used. The loop runs from row 0 to row 176 in steps of eight, which is twenty-three dashes, two pixels wide and four tall. They sit on columns 159 and 160. The middle of a 320-wide picture is the line between those two columns, so a two-pixel dash sits on it exactly. Every game in this volume that has a court draws it with these three calls.

2. Circles, rectangles and lines

The `vector` package draws five shapes by hand: a filled and a stroked rectangle, a filled and a stroked circle, and a line. Add one of each to the left half of the court.

▣ BUILD · STAGE 2 — ONE OF EACH SHAPE, EXTEND CMD/SHAPES/MAIN.GO

```
// cmd/shapes/main.go — extend
var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
        dimColor   = color.RGBA{R: 60, G: 66, B: 80, A: 255}
        ballColor  = color.RGBA{R: 232, G: 120, B: 80, A: 255}
)

func (g *Game) Draw(screen *ebiten.Image) {
        drawCourt(screen)
        drawShapes(screen)
}

// drawShapes paints one of each shape on the left half of the
court.
func drawShapes(screen *ebiten.Image) {
        vector.FillCircle(screen, 40, 50, 20, ballColor, false)
        vector.StrokeCircle(screen, 100, 50, 20, 2, lineColor,
false)
        vector.FillRect(screen, 20, 100, 40, 24, lineColor,
false)
        vector.StrokeRect(screen, 80, 100, 40, 24, 2, lineColor,
false)
        vector.StrokeLine(screen, 20, 150, 140, 150, 1,
dimColor, false)
        vector.StrokeLine(screen, 20, 160, 140, 170, 3,
dimColor, false)
}


go vet ./...
go run ./cmd/shapes
```

The window after stage 2: a filled and a stroked circle, a filled and a stroked rectangle, and two lines, on the left half of the court.

This is the first listing marked `extend`, and the convention holds for the rest of the book. An extend listing prints whole declarations. Each one replaces the declaration of the same name in the file, or is added at the end if the file has none. Here the `var` block gains two colours, `Draw` gains a line, and `drawShapes` is new. The imports do not change.

Each call names its shape by numbers. A circle takes its centre and its radius. A rectangle takes its top-left corner, its width and its height. A line takes its two ends. A stroked shape takes a stroke width before its colour.

Every call ends in `false`. That argument declines antialiasing, which is the smoothing of a shape's edges. A pixel game declines it on purpose. At 320 by 180 each pixel is a visible square on screen, and a smoothed edge is a row of half-lit squares that reads as blur. The line at row 150 is one pixel wide and the sloping line below it is three, which is the difference between a hairline and a stroke at this size.

3. Making an image from text

Most of what a game draws is an image: a picture made once and copied onto the screen wherever the game wants it. An image is a rectangle of pixels. Each pixel is four bytes: red, green, blue, and alpha, which is how opaque the pixel is, 255 for solid. The bytes run left to right along a row, and the rows run top to bottom. So an eight-by-eight image is 256 bytes, and the pixel at column `x`, row `y` starts at byte 4 × (8`y` + `x`).

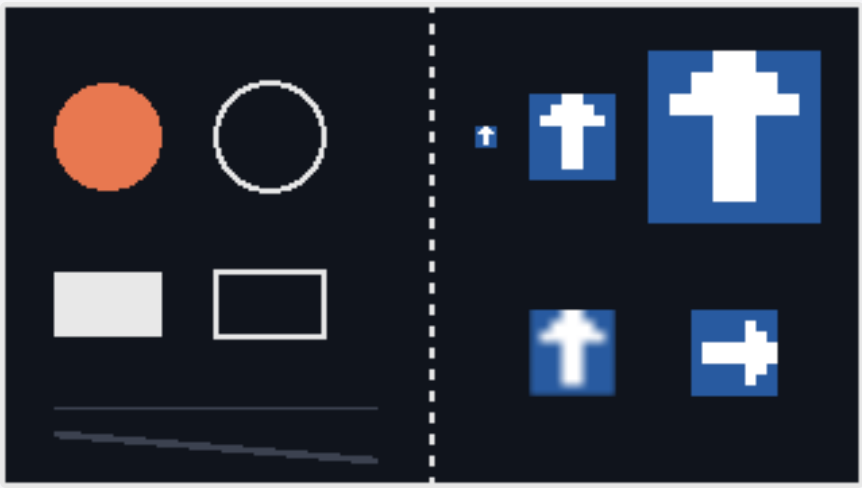
This stage builds an eight-by-eight arrow from eight lines of text and draws it five ways: at one, four and eight times its size, at four times through a different filter, and at four times turned a quarter turn.

▣ BUILD · STAGE 3 — AN ARROW DRAWN FIVE WAYS, EXTEND CMD/SHAPES/MAIN.GO

```
// cmd/shapes/main.go — extend
import (
        "image/color"
        "log"
        "math"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

type Game struct {
        glyph *ebiten.Image
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawCourt(screen)
        drawShapes(screen)
        drawGlyphs(screen, g.glyph)
}

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Shapes")
        ebiten.SetTPS(60)
        if err := ebiten.RunGame(&Game{glyph: newGlyph()}); err
≠ nil {
                log.Fatal(err)
        }
```

```
}

// glyphArt is an eight-by-eight picture as text: # is ink, . is
paper.
var glyphArt = []string{
        "...##...",
        "..####..",
        ".######.",
        "...##...",
        "...##...",
        "...##...",
        "...##...",
        "........",
}

var (
        inkColor   = color.RGBA{R: 255, G: 255, B: 255, A: 255}
        paperColor = color.RGBA{R: 40, G: 90, B: 160, A: 255}
)

// newGlyph turns glyphArt into an image: four bytes a pixel,
red, green,
// blue and alpha, the rows laid end to end.
func newGlyph() *ebiten.Image {
        w, h := len(glyphArt[0]), len(glyphArt)
        pix := make([]byte, 4*w*h)
        for y, row := range glyphArt {
                for x := 0; x < w; x++ {
                        c := paperColor
                        if row[x] == '#' {
                                c = inkColor
                        }
                        i := 4 * (y*w + x)
                        pix[i], pix[i+1], pix[i+2], pix[i+3] =
c.R, c.G, c.B, c.A
                }
        }
        img := ebiten.NewImage(w, h)
        img.WritePixels(pix)
        return img
}

// drawGlyph draws img scaled by s, turned by angle degrees
about its own
// centre, with the centre at (x, y), through the filter given.
func drawGlyph(screen, img *ebiten.Image, x, y, s, angle
float64, filter ebiten.Filter) {
        w, h := float64(img.Bounds().Dx()),
float64(img.Bounds().Dy())
        op := &ebiten.DrawImageOptions{}
        op.GeoM.Translate(-w/2, -h/2)
        op.GeoM.Scale(s, s)
        op.GeoM.Rotate(angle * math.Pi / 180)
        op.GeoM.Translate(x, y)
```

```
        op.Filter = filter
        screen.DrawImage(img, op)
}

// drawGlyphs draws the glyph on the right half of the court: at
one, four
// and eight times through the nearest-pixel filter, then at
four times
// through the linear filter, then at four times turned a
quarter turn.
func drawGlyphs(screen, glyph *ebiten.Image) {
        drawGlyph(screen, glyph, 180, 50, 1, 0,
ebiten.FilterNearest)
        drawGlyph(screen, glyph, 212, 50, 4, 0,
ebiten.FilterNearest)
        drawGlyph(screen, glyph, 272, 50, 8, 0,
ebiten.FilterNearest)
        drawGlyph(screen, glyph, 212, 130, 4, 0,
ebiten.FilterLinear)
        drawGlyph(screen, glyph, 272, 130, 4, 90,
ebiten.FilterNearest)
}


go vet ./...
go run ./cmd/shapes
```



The window after stage 3: the arrow at one, four and eight times through the nearest filter, at four times through the linear filter, and at four times turned a quarter turn.

`newGlyph` walks the eight strings and writes four bytes for each character into one 256-byte slice, at the offset the formula above gives. `ebiten.NewImage` makes an empty image of that size, and `WritePixels` copies the slice into it. That call is how pixels get into an image from anywhere; a file decoder does the same thing after reading a PNG.

The image is made once, in `main`, and kept in the `Game`. Making it again in every `Draw` would send 256 bytes to the graphics card sixty or more times a second for no gain.

`screen.DrawImage` is Ebitengine's call for copying one image onto another. It takes the image and a `DrawImageOptions`, whose `GeoM` field holds the placement and whose `Filter` field says how pixels are looked up. The next two sections take those in turn.

4. Scale, rotate, then move

`GeoM` is a matrix: a list of steps a point goes through. You build it by calling `Translate`, `Scale` and `Rotate` on it, and the steps are applied in the order you wrote them. `Scale` and `Rotate` both work about the origin, the point (0, 0).

An image's own coordinates start at its top-left corner, so `drawGlyph` has four steps. The first `Translate` moves the image four pixels up and left, which puts its centre on the origin. `Scale` multiplies every coordinate by `s`. `Rotate` turns the image about its centre. The last `Translate` carries the centre to (`x`, `y`).

Follow the four-times arrow centred on (212, 50). The image's corner (0, 0) goes to (-4, -4), then to (-16, -16), then stays put through a turn of zero, then lands at (196, 34). The opposite corner lands at (228, 66). That is a 32-pixel square with its centre where it was asked for.

Scaling and rotating first, then moving last, is what keeps those two steps working about the image's own centre. A move done earlier is scaled and turned by the steps that follow it.

∑ MATH INTERLUDE — WHERE A CORNER LANDS

Take the source corner (8, 8) of the four-times arrow centred on (212, 50). It becomes (4, 4) after the centring move, (16, 16) after the scale, stays put through a turn of zero, and ends at (228, 66) after the last move.

Now run the same steps with the last move done first. The corner becomes (4, 4), then (216, 54), then, scaled by four, (864, 216). That is more than twice the picture's width away. Scaling after moving scales the move.

`x, y` a point of the source image: column and row, from its top-left corner
`Translate(a, b)` add a to x and b to y: the point moves by (a, b)
`Scale(s, s)` multiply x and y by s: the point moves s times further from the origin
`Rotate(r)` turn the point about the origin by r, an angle in radians; degrees × π / 180
`radian` the angle unit the library counts in: a full turn is 2π, about 6.28, so a quarter turn is π/2

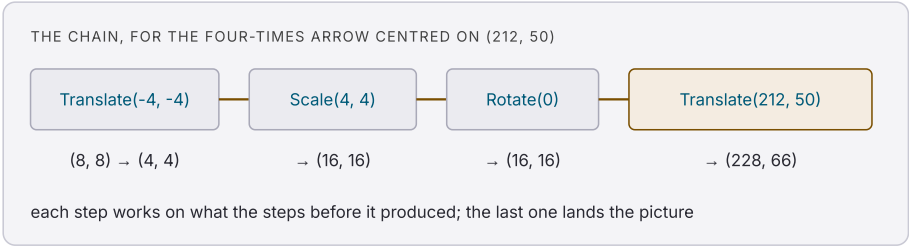


Figure 2.1 — the four steps of `drawGlyph`, with one corner of the source image followed through them: centred, scaled, turned, and moved to where it goes.

⚠ WORKED FAILURE — TRANSLATE BEFORE SCALE

Move the last `Translate` up two lines, so the glyph is moved to (`x`, `y`) and then scaled:

```
func drawGlyph(screen, img *ebiten.Image, x, y, s, angle
float64, filter ebiten.Filter) {
        w, h := float64(img.Bounds().Dx()),
float64(img.Bounds().Dy())
        op := &ebiten.DrawImageOptions{}
        op.GeoM.Translate(-w/2, -h/2)
        op.GeoM.Translate(x, y)
        op.GeoM.Scale(s, s)
        op.GeoM.Rotate(angle * math.Pi / 180)
        op.Filter = filter
        screen.DrawImage(img, op)
}
```

The same program with the move before the scale: four of the five arrows are gone.

It compiles, it vets, it runs, and four arrows are missing. Nothing reports an error. Reason from the arrow that is left.

The one-times arrow is where it was, and one is the only scale that leaves a position alone: (176, 46) multiplied by one is (176, 46). The four-times arrow's centre went to (212, 50) and was then multiplied by four, to (848, 200), which is off the right edge and off the bottom. The eight-times arrow went to

(2176, 400). The two on the lower row went to (848, 520) and (1088, 520), and the second of those was then turned a quarter turn about the origin, which swung it further away still.

Drawing outside the picture is legal, which is why there is no error: a sprite walking off the edge of the screen does it on purpose. Put the move back where it belongs, last.

5. The nearest and the linear filter

The filter decides what colour a screen pixel gets when it falls between source pixels, which happens whenever an image is scaled or turned.

`FilterNearest` gives the screen pixel the colour of the one source pixel it lands on, so a source pixel scaled by four becomes a crisp four-by-four square. `FilterLinear` blends the four nearest source pixels, which suits a photograph and spoils pixel art: it turns every edge into a fade. In the picture above, the four-times arrow through the linear filter is the blurred one.

Every image in this volume is drawn through the nearest filter, which is also Ebitengine's default. `drawGlyph` takes the filter as an argument only so that the two can be put side by side.

6. How DrawImage uses the matrix

`DrawImage` does not move pixels one at a time. It sends the image's four corners through the matrix, which gives a four-sided figure on the screen, and asks the graphics card to fill it. For each screen pixel inside that figure the card works out which point of the source it corresponds to and looks that point up through the filter.

Three things follow. Any chain of moves, scales and turns costs the same as a plain copy, because the matrix is applied to four corners

either way. The order of the steps is the whole meaning of the chain. And the filter belongs to the lookup, not to the image, so the same image can be drawn crisp in one call and blurred in the next.

7. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Draw the court with three calls, and say which pixels a two-pixel stroke covers and why the net sits on columns 159 and 160.
- Name the five `vector` calls, the numbers each one takes, and what the final `false` declines.
- Make an eight-by-eight image from text with `WritePixels`, and give the byte offset of any pixel in it.
- Build a `GeoM` that scales, turns and moves an image about its own centre, and follow one corner through the four steps by hand.
- Say what the nearest and the linear filter do to a scaled-up pixel, and which one a pixel game uses.
- Explain, from where the surviving arrow was, why a move before a scale sends the other arrows off the screen.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — a diagonal net. Replace the dashes with one line from the top-left corner to the bottom-right, one pixel wide, and run it. Which pixels does that line light on a 320-by-180 picture?**

`vector.StrokeLine(screen, 0, 0, screenW, screenH, 1, lineColor, false)`. The line is steeper across than down, so it lights roughly one pixel per column, 320 in all, stepping down a row every 1.78 columns. With the smoothing declined, each step is a hard one-pixel jog, and you can count the jogs.

▼ **Exercise 2 — the fade, measured. Draw the glyph at sixteen times through `FilterLinear`, on its own, and count how many screen pixels the fade between paper and ink spans.**

Sixteen. The blend runs from the centre of one source pixel to the centre of the next, which at sixteen times is sixteen screen pixels. The same edge through `FilterNearest` is one pixel wide. The fade is as wide as the scale, so linear filtering of pixel art gets worse the bigger the art is drawn.

▼ **Exercise 3 — a turning arrow. Give the game an `angle` field, add one to it in `Update`, and draw the eight-times arrow with it. How long does one full turn take, and where does the change belong?**

Six seconds: 360 ticks of one degree at sixty ticks a second. The `angle += 1` belongs in `Update`, by chapter 1's rule, and the draw call reads it. Put the increment in `Draw` and the arrow turns at whatever speed the display gives it.

VOLUME 1 · CHAPTER 3

Input

Keys held, keys just pressed, and a paddle under your hands

1. Reading input

This chapter builds a paddle you can move with W and S. Space serves a ball once per press, and a mouse click moves the paddle to the cursor's row.

Ebitengine provides input polling. `ebiten.IsKeyPressed(ebiten.KeyW)` answers whether W is held on this tick. A serve needs a different answer: whether Space went down on this tick. The program reads those answers once at the top of `Update` and passes plain booleans to the game rules.

2. Moving the paddle

▣ BUILD · STAGE 1 — A PADDLE UNDER W AND S

```
// cmd/paddle/main.go — create
package main

import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

const (
        screenW = 320
        screenH = 180
        border  = 2 // the court's border, which nothing crosses
```

```
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
        dimColor   = color.RGBA{R: 60, G: 66, B: 80, A: 255}
)

// drawCourt paints the court: the fill, a border two pixels
wide just
// inside the edge, and a net of dashes down the middle.
func drawCourt(screen *ebiten.Image) {
        screen.Fill(courtColor)
        vector.StrokeRect(screen, 1, 1, screenW-2, screenH-2, 2,
lineColor, false)
        for y := 0; y < screenH; y += 8 {
                vector.FillRect(screen, 159, float32(y), 2, 4,
lineColor, false)
        }
}

// The paddle is four pixels wide and twenty-four tall, on
column 8, and
// moves two pixels a tick.
const (
        paddleX     = 8
        paddleW     = 4
        paddleH     = 24
        paddleSpeed = 2
)

// keys is what the player is doing this tick: the keys held.
type keys struct {
        up, down bool // W and S, while held
}

// readKeys asks Ebitengine about the keyboard, once a tick.
func readKeys() keys {
        var k keys
        k.up = ebiten.IsKeyPressed(ebiten.KeyW)
        k.down = ebiten.IsKeyPressed(ebiten.KeyS)
        return k
}

type Game struct {
        paddleY float64 // the paddle's top edge
}

// clamp returns v held inside [lo, hi].
func clamp(v, lo, hi float64) float64 {
        if v < lo {
                return lo
        }
        if v > hi {
```

```
                return hi
        }
        return v
}

// step advances the game one tick with what the player is
doing.
func (g *Game) step(k keys) {
        if k.up {
                g.paddleY -= paddleSpeed
        }
        if k.down {
                g.paddleY += paddleSpeed
        }
        g.paddleY = clamp(g.paddleY, border, screenH-border-
paddleH)
}

func (g *Game) Update() error {
        g.step(readKeys())
        return nil
}

func (g *Game) drawPaddle(screen *ebiten.Image) {
        vector.FillRect(screen, paddleX, float32(g.paddleY),
paddleW, paddleH, lineColor, false)
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawCourt(screen)
        g.drawPaddle(screen)
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Paddle")
        ebiten.SetTPS(60)
        g := &Game{paddleY: (screenH - paddleH) / 2}
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}


go vet ./...
go run ./cmd/paddle
```

The window after stage 1, with W held for half a second: thirty ticks of two pixels have moved the paddle from row 78 to row 18.

Hold W and the paddle moves up two pixels a tick. Hold S and it moves down. Hold both and the moves cancel, so the paddle stays where it is. Let go and the paddle stops on that tick because `IsKeyPressed` is read again every tick.

`clamp` keeps the paddle inside the two-pixel border. Its top edge can be as high as row 2 and as low as row 154, which puts the bottom edge on row 177. The clamp runs after movement, so holding W at the top has no visible effect.

`Update` reads the keys and hands them to `step`. `step` receives a `keys` value, not the keyboard itself. That keeps the physical controls in `readKeys` and the paddle rules in `step`.

3. Reading a press once

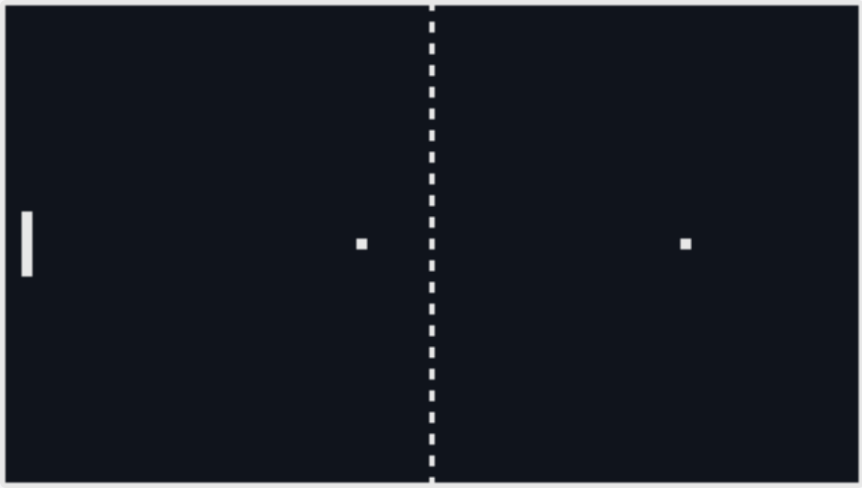
A serve should happen once per press. Ebitengine's `inpututil` package keeps last tick's keyboard state and compares it with this tick. `IsKeyJustPressed` is true only on the tick a key goes down.

▣ BUILD · STAGE 2 — A BALL SERVED ON SPACE, EXTEND

CMD/PADDLE/MAIN.GO

```
// cmd/paddle/main.go — extend
import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/inpututil"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

// keys is what the player is doing this tick: the keys held,
and the keys
// that went down this tick.
type keys struct {
        up, down bool // W and S, while held
        serve    bool // Space, on the tick it went down
}

// readKeys asks Ebitengine about the keyboard, once a tick.
func readKeys() keys {
        var k keys
        k.up = ebiten.IsKeyPressed(ebiten.KeyW)
        k.down = ebiten.IsKeyPressed(ebiten.KeyS)
        k.serve = inpututil.IsKeyJustPressed(ebiten.KeySpace)
        return k
}

type Game struct {
        paddleY float64 // the paddle's top edge
        balls   []ball  // every ball in flight, oldest first
}

// step advances the game one tick with what the player is
doing.
func (g *Game) step(k keys) {
        if k.up {
                g.paddleY -= paddleSpeed
        }
        if k.down {
                g.paddleY += paddleSpeed
        }
        g.paddleY = clamp(g.paddleY, border, screenH-border-
paddleH)
        if k.serve {
                g.balls = append(g.balls, ball{x: paddleX +
paddleW, y: g.paddleY + paddleH/2 - 2})
        }
        kept := g.balls[:0]
        for _, b := range g.balls {
```

```
                b.x += ballSpeed
                if b.x < screenW {
                        kept = append(kept, b)
                }
        }
        g.balls = kept
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawCourt(screen)
        g.drawPaddle(screen)
        g.drawBalls(screen)
}

// A ball served from the paddle flies right at six pixels a
tick.
const ballSpeed = 6

type ball struct {
        x, y float64
}

func (g *Game) drawBalls(screen *ebiten.Image) {
        for _, b := range g.balls {
                vector.FillRect(screen, float32(b.x),
float32(b.y), 4, 4, lineColor, false)
        }
}


go vet ./...
go run ./cmd/paddle
```



The window after stage 2, two thirds of a second after the first of two taps of Space a third of a second apart: two balls in flight, 120 pixels apart.

Tap Space and a four-pixel ball leaves the paddle's face. It starts level with the paddle's middle. Hold Space and only one ball leaves, because `k.serve` is true on one tick.

The balls live in a slice. A serve appends a ball. Each tick moves every ball six pixels and keeps only the balls still on the court. The keepers are appended to `g.balls[:0]`, which reuses the same backing array instead of making a new one.

The extend listing changes the import block, `keys`, `readKeys`, `Game`, `step` and `Draw`. The ball declarations are new and go at the end of the file.

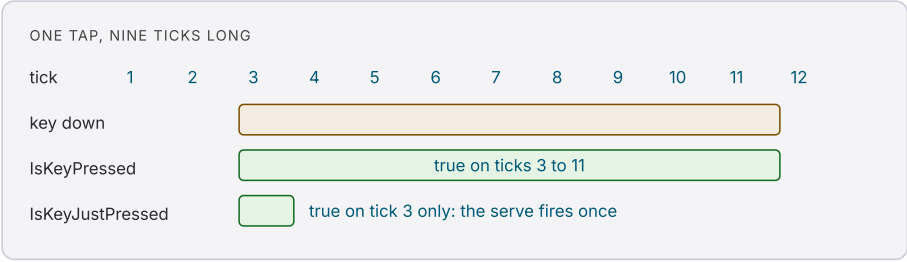


Figure 3.1 — a tap of Space seen tick by tick: held for nine ticks, and down on one. A serve wants the second answer, a paddle the first.

⚠ WORKED FAILURE — SPACE READ AS A HELD KEY

`IsKeyPressed` reads naturally for Space, but it gives the wrong kind of answer for a serve. Use it for the serve:

```
func readKeys() keys {
        var k keys
        k.up = ebiten.IsKeyPressed(ebiten.KeyW)
        k.down = ebiten.IsKeyPressed(ebiten.KeyS)
        k.serve = ebiten.IsKeyPressed(ebiten.KeySpace)
```

```
        return k
}
```

The same program with Space read as a held key, half a second after one tap of Space that lasted nine ticks: nine balls, six pixels apart.

One tap makes nine balls because the tap lasted nine ticks. On each of those ticks, `k.serve` was true. Each ball is six pixels behind the one before it because each one left the paddle one tick later.

A held Space key serves sixty balls a second. The program cannot tell a tap from a hold because `IsKeyPressed` answers only about this tick. `inpututil` remembers the previous tick, so it can answer whether the key went down now. Put `IsKeyJustPressed` back.

4. Reading the mouse

▣ BUILD · STAGE 3 — THE MOUSE, EXTEND CMD/PADDLE/MAIN.GO

```
// cmd/paddle/main.go — extend
// keys is what the player is doing this tick: the keys held,
```

```
the keys
// that went down this tick, and where the mouse is.
type keys struct {
        up, down bool // W and S, while held
        serve    bool // Space, on the tick it went down
        click    bool // the left mouse button, on the tick it
went down
        mx, my   int  // the cursor, in the picture's pixels
}

// readKeys asks Ebitengine about the keyboard and the mouse,
once a tick.
func readKeys() keys {
        var k keys
        k.up = ebiten.IsKeyPressed(ebiten.KeyW)
        k.down = ebiten.IsKeyPressed(ebiten.KeyS)
        k.serve = inpututil.IsKeyJustPressed(ebiten.KeySpace)
        k.click =
inpututil.IsMouseButtonJustPressed(ebiten.MouseButtonLeft)
        k.mx, k.my = ebiten.CursorPosition()
        return k
}

type Game struct {
        paddleY float64 // the paddle's top edge
        balls   []ball  // every ball in flight, oldest first
        mx, my  int     // where the cursor was on the last tick
}

// step advances the game one tick with what the player is
doing.
func (g *Game) step(k keys) {
        if k.up {
                g.paddleY -= paddleSpeed
        }
        if k.down {
                g.paddleY += paddleSpeed
        }
        if k.click {
                g.paddleY = float64(k.my) - paddleH/2 // the
paddle's middle to the cursor's row
        }
        g.paddleY = clamp(g.paddleY, border, screenH-border-
paddleH)
        g.mx, g.my = k.mx, k.my
        if k.serve {
                g.balls = append(g.balls, ball{x: paddleX +
paddleW, y: g.paddleY + paddleH/2 - 2})
        }
        kept := g.balls[:0]
        for _, b := range g.balls {
                b.x += ballSpeed
                if b.x < screenW {
                        kept = append(kept, b)
```

```
                }
        }
        g.balls = kept
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawCourt(screen)
        g.drawPaddle(screen)
        g.drawBalls(screen)
        g.drawCursor(screen)
}

// drawCursor marks the cursor with a small cross.
func (g *Game) drawCursor(screen *ebiten.Image) {
        x, y := float32(g.mx), float32(g.my)
        vector.StrokeLine(screen, x-4, y, x+5, y, 1, dimColor,
false)
        vector.StrokeLine(screen, x, y-4, x, y+5, 1, dimColor,
false)
}


go vet ./...
go run ./cmd/paddle
```

The window after stage 3, the tick after a click at column 200, row 60: the cross marks the cursor and the paddle's middle has moved to the cursor's row.

Move the mouse and the grey cross follows it in the picture's own pixels. Drag the window larger and the cross does not speed up.

Ebitengine maps the cursor into the 320-by-180 coordinates that `Layout` chose.

Click and the paddle jumps so its middle is on the cursor's row. The next line clamps it, so a click near the border cannot push it through the wall. The cursor's position is copied into the game in `Update` and drawn from that copy in `Draw`.

Before the cursor enters the window, Ebitengine reports the last position it knew. At the start, that is the top-left corner, so the cross sits there until the mouse moves.

⚙ TOOL — WHAT THE LIBRARY REPORTS ABOUT INPUT

`ebiten.IsKeyPressed(key)` is true while a key is held; the keys are constants such as `ebiten.KeyW`, `ebiten.KeySpace` and `ebiten.KeyArrowUp`, one per physical key. `inpututil.IsKeyJustPressed(key)` is true on the tick the key went down, and `inpututil.IsMouseButtonJustPressed(ebiten.MouseButtonLeft)` the same for a button. `ebiten.CursorPosition()` returns the cursor in the game's own coordinates; it can be outside the picture when the cursor is outside the window. None of these reports events: each reports the state at the moment it is asked, so `Update` asks once a tick. The reference is `pkg.go.dev/github.com/hajimehoshi/ebiten/v2/inpututil`.

5. Keeping input in one value

The operating system sees input as events: a key went down, then a key came up. Ebitengine turns that stream into answers the game can poll once per tick. `inpututil` adds the comparison with the previous tick.

Reading input at the top of `Update` gives the whole tick one set of answers. A key cannot be held in `step` and released in `Draw`, because `Draw` does not ask the keyboard. It draws the state that `Update` already decided.

6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Move a paddle two pixels a tick while a key is held and keep it inside a two-pixel border with a clamp, and say which rows its top edge can occupy.
- Read a key that should act once per press with `inpututil`, and explain, in ticks, what a tap of Space does when it is read as held instead.
- Pack the keys, the just-pressed keys and the cursor into one value at the top of `Update`, and say what that buys the rest of the tick.
- Read the cursor in the picture's coordinates and move something to its row on a click.
- Walk a slice of balls, moving each and dropping the ones that have left, without allocating.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — the arrows too. Make Up and Down move the paddle as well as W and S, without touching `step`.**

Two lines in `readKeys`: `k.up = ebiten.IsKeyPressed(ebiten.KeyW) || ebiten.IsKeyPressed(ebiten.KeyArrowUp)`, and the same for down. `step` sees a boolean and does not know two keys feed it, which is what the `keys` value is for.

▼ **Exercise 2 — a second paddle. Add a paddle on column 308 under the Up and Down arrows, and serve the balls from whichever paddle was moved last.**

A second `Y` field and two more booleans in `keys`; a `last` field set to 1 or 2 whenever a paddle's keys are held; the serve reads `last` to pick the paddle and the direction, and the ball type needs a velocity so that a ball from the right paddle flies left. Both paddles clamp to the same two rows.

▼ **Exercise 3 — the count in the title. Count the serves and show the count in the window's title.**

A `serves` field incremented where the ball is appended, and `ebiten.SetWindowTitle(fmt.Sprintf("Paddle: %d serves", g.serves))` on the same line, with `fmt` imported. Tap Space three times and the title bar reads 3; hold it and it still reads one more, which is the chapter's point, made by the window manager.

VOLUME 1 · CHAPTER 4

Sprites and Animation

A sheet of small pictures, cut into sub-images, and a frame counter in Update

1. Loading a sprite sheet

This chapter loads one PNG that holds the game's small pictures. The program cuts it into paddles, ball frames and digits, then animates the ball with a counter in `Update`.

The sheet is 64 by 32 pixels. It contains two paddles, four ball frames and the digits 0 to 9. Ebitengine draws the whole sheet through `DrawImage`, or it draws a sub-image that views one rectangle of the same sheet.

2. Showing the sheet

Download the sheet into the module, at `assets/pong-sheet.png`; the programs open it by that path, from the module's root, where every `go run` in this book is typed. The second file is the same sheet at four times its size on the court colour, for looking at; nothing loads it.

▣ BUILD · STAGE 1 — THE SHEET ON THE SCREEN

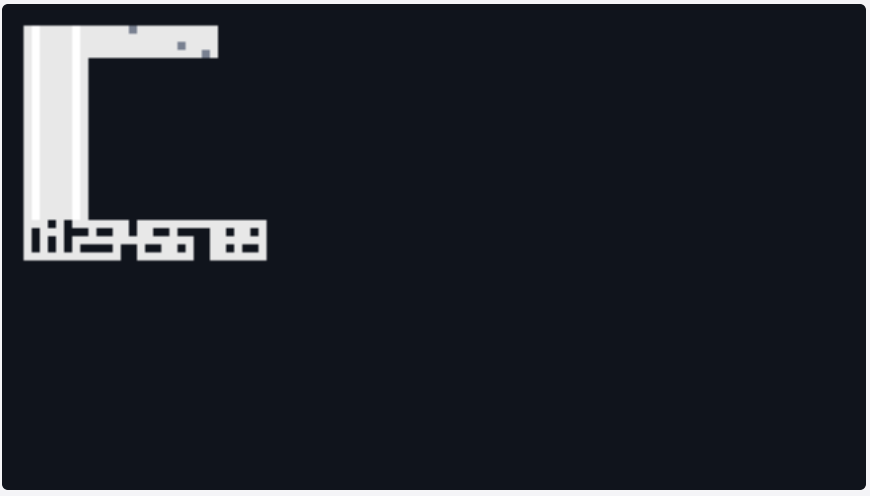
```
mkdir -p assets
```

pong-sheet.png (PNG, 64 × 32, transparent background; the sheet the games load)

pong-sheet-4x.png (PNG, 256 × 128; the same sheet at four times on the court colour)

```
// cmd/sprite/main.go — create
package main

import (
        "image/color"
        "image/png"
        "log"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
)

const (
        screenW = 320
        screenH = 180
)

var courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}

// loadImage reads a PNG file and hands its pixels to
Ebitengine.
func loadImage(path string) (*ebiten.Image, error) {
        f, err := os.Open(path)
        if err ≠ nil {
                return nil, err
        }
        defer f.Close()
        img, err := png.Decode(f)
        if err ≠ nil {
                return nil, err
        }
        return ebiten.NewImageFromImage(img), nil
}

// drawAt draws img scaled by s with its top-left corner at (x,
y).
func drawAt(screen, img *ebiten.Image, x, y, s float64) {
        op := &ebiten.DrawImageOptions{}
        op.GeoM.Scale(s, s)
        op.GeoM.Translate(x, y)
        screen.DrawImage(img, op)
}

type Game struct {
        sheet *ebiten.Image
}

func (g *Game) Update() error {
        return nil
```

```
}

// drawSheet draws the whole sheet at three times its size.
func (g *Game) drawSheet(screen *ebiten.Image) {
        screen.Fill(courtColor)
        drawAt(screen, g.sheet, 8, 8, 3)
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawSheet(screen)
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Sprites")
        ebiten.SetTPS(60)
        sheet, err := loadImage("assets/pong-sheet.png")
        if err ≠ nil {
                log.Fatal(err)
        }
        if err := ebiten.RunGame(&Game{sheet: sheet}); err ≠
nil {
                log.Fatal(err)
        }
}


go vet ./...
go run ./cmd/sprite
```



The window after stage 1: the sheet at three times its size, the court colour showing through its transparent pixels.

`loadImage` opens the file, decodes the PNG with `image/png` and hands the decoded image to `ebiten.NewImageFromImage`. Ebitengine then has an image it can draw.

The file is decoded once in `main`, before the window opens. Reading it in `Draw` would read the same file every time the screen was painted. The paddles sit at the sheet's top-left corner, the ball frames sit beside them, and the digits sit along the bottom.

`pong-sheet-4x.png`: the sheet at four times. Two paddles, each lit one column in from the side that will face the net; the ball's four frames, with a grey seam at the top, the right and the bottom of the last three; the ten digits along row 24.

⚙ TOOL — A PIXEL EDITOR, FOR A SHEET OF YOUR OWN

The sheet was made pixel by pixel, 362 painted pixels in all. Piskel (`piskelapp.com`) runs in a browser and exports a PNG. LibreSprite is a free installed editor.

Draw on a transparent background. Keep each piece at whole-pixel positions you write down. Export at one times, not scaled,

because the game does the scaling.

3. Cutting sub-images

A piece of the sheet is a rectangle: its left column, top row, width and height, in the sheet's own pixels. The paddles are at (0, 0) and (4, 0), four wide and twenty-four tall; the ball's frames start at column 8 and follow every four columns, four by four; the digits start at (0, 24) and follow every three columns, three wide and five tall. `SubImage` takes such a rectangle and returns an image that is the sheet seen through it: drawing the sub-image draws those pixels of the sheet and no others, and no pixel is copied to make it.

▣ BUILD · STAGE 2 — THE PIECES, CUT OUT AND LAID OUT, EXTEND CMD/SPRITE/MAIN.GO

```
// cmd/sprite/main.go — extend
import (
        "image"
        "image/color"
        "image/png"
        "log"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
)

type Game struct {
        sp *sprites
}

// drawSheet draws the whole sheet at three times its size.
func (g *Game) drawSheet(screen *ebiten.Image) {
        screen.Fill(courtColor)
        drawAt(screen, g.sp.sheet, 8, 8, 3)
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawSheet(screen)
        g.drawPieces(screen)
}

func main() {
        ebiten.SetWindowSize(960, 540)
```

```
        ebiten.SetWindowTitle("Sprites")
        ebiten.SetTPS(60)
        sheet, err := loadImage("assets/pong-sheet.png")
        if err ≠ nil {
                log.Fatal(err)
        }
        if err := ebiten.RunGame(&Game{sp: cutSprites(sheet)});
err ≠ nil {
                log.Fatal(err)
        }
}

// sprites is the sheet cut into the pictures a game draws. Each
piece is a
// sub-image: a view onto the one sheet, not a copy of its
pixels.
type sprites struct {
        sheet       *ebiten.Image
        left, right *ebiten.Image    // the paddles, four by
twenty-four
        ball        [4]*ebiten.Image // the ball's four spin
frames, four by four
        digits      [10]*ebiten.Image
}

// cutSprites cuts the pieces out of the sheet, where the
sheet's layout
// puts them: paddles at the top left, the ball's frames beside
them, the
// digits, three by five each, along row 24.
func cutSprites(sheet *ebiten.Image) *sprites {
        cut := func(x, y, w, h int) *ebiten.Image {
                return sheet.SubImage(image.Rect(x, y, x+w,
y+h)).(*ebiten.Image)
        }
        s := &sprites{sheet: sheet, left: cut(0, 0, 4, 24),
right: cut(4, 0, 4, 24)}
        for i := range s.ball {
                s.ball[i] = cut(8+4*i, 0, 4, 4)
        }
        for i := range s.digits {
                s.digits[i] = cut(3*i, 24, 3, 5)
        }
        return s
}

// drawPieces draws each piece the sheet was cut into, at four
times: the
// paddles to the right of the sheet, the ball's frames and the
digits in
// a row under it.
func (g *Game) drawPieces(screen *ebiten.Image) {
        drawAt(screen, g.sp.left, 216, 8, 4)
        drawAt(screen, g.sp.right, 240, 8, 4)
```

```
        for i, b := range g.sp.ball {
                drawAt(screen, b, float64(8+24*i), 116, 4)
        }
        for i, d := range g.sp.digits {
                drawAt(screen, d, float64(120+16*i), 116, 4)
        }
}


go vet ./...
go run ./cmd/sprite
```

The window after stage 2: the sheet, and beside and below it each of the sixteen pieces at four times, drawn apart.

`cut` is declared inside `cutSprites` so the rectangles stay close to the sheet layout. Each call names the left column, top row, width and height of one piece.

`SubImage` returns Go's `image.Image` interface. Ebitengine documents that a sub-image of an Ebitengine image is also an Ebitengine image, so the type assertion turns it back into `*ebiten.Image`.

No pixels are copied when the program cuts the sheet. Drawing sixteen pieces means sixteen `DrawImage` calls, all reading rectangles from the same loaded image.

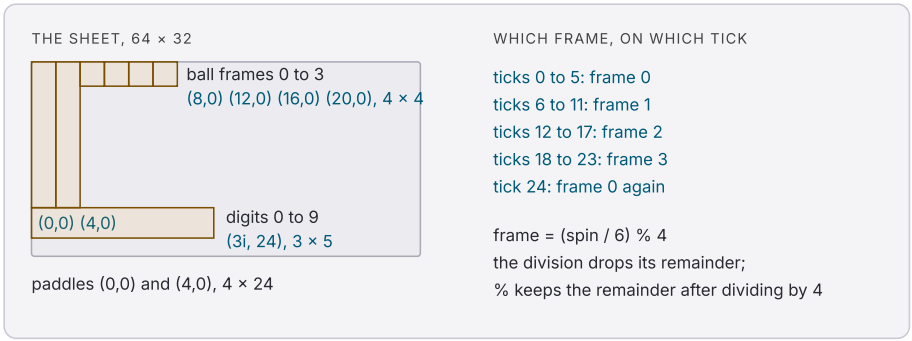


Figure 4.1 — the sheet's layout, as sixteen rectangles, and the counter that picks the ball's frame: six ticks a frame, four frames a turn.

4. Animating with a counter

▣ BUILD · STAGE 3 — THE SPIN, COUNTED IN UPDATE, EXTEND CMD/SPRITE/MAIN.GO

```
// cmd/sprite/main.go — extend
type Game struct {
        sp   *sprites
        spin int // ticks the ball has been turning
}

func (g *Game) Update() error {
        g.spin++
        return nil
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawSheet(screen)
        g.drawPieces(screen)
        g.drawSpin(screen)
}

// The ball turns one frame every six ticks: a full turn in
twenty-four
// ticks, two and a half turns a second.
const spinEvery = 6

// drawSpin draws the frame the counter picks: at eight times in
the top
// right corner, and at one, rolling along the bottom a pixel a
tick.
func (g *Game) drawSpin(screen *ebiten.Image) {
```

```
        frame := (g.spin / spinEvery) % 4
        drawAt(screen, g.sp.ball[frame], 272, 8, 8)
        drawAt(screen, g.sp.ball[frame],
float64(g.spin%screenW), 168, 1)
}


go vet ./...
go run ./cmd/sprite
```

The window fifteen ticks after stage 3 starts: the counter reads 15, fifteen divided by six is two, and frame 2 has the seam on the right. The small ball has rolled fifteen pixels along the bottom.

The big ball turns because `spin` chooses a different frame every six ticks. The small ball uses the same frame and moves one pixel per tick along the bottom. A full turn takes twenty-four ticks, so the small ball rolls twenty-four pixels per turn.

`Update` raises `spin` by one. `Draw` may read the counter and choose a frame, but it must not change the counter.

∑ MATH INTERLUDE — SIX TICKS A FRAME, FOUR FRAMES A TURN

The counter goes 0, 1, 2, … and the frame has to go 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, …, 3, 3, 3, 3, 3, 3, 0, 0, …: each frame held for

six ticks, then round again. Dividing the counter by six and dropping the remainder gives 0 for ticks 0 to 5, 1 for 6 to 11, 2 for 12 to 17, and keeps climbing; taking that number's remainder after dividing by four folds it back onto 0 to 3. On tick 15, 15 ÷ 6 is 2 remainder 3, so 2; 2 divided by 4 is 0 remainder 2, so frame 2. On tick 45, 45 ÷ 6 is 7; 7 divided by 4 leaves 3; frame 3. Go's `/` on integers drops the remainder and `%` keeps it, so the two steps are `(spin / 6) % 4`.

`spin` the counter: ticks the ball has been turning
`a / b` integer division: how many whole times b goes into a; 15 / 6 is 2
`a % b` the remainder after that division; 15 % 6 is 3, and 7 % 4 is 3
`frame` which of the four pictures to draw: (spin / 6) % 4, always 0 to 3

⚠ WORKED FAILURE — THE COUNTER IN DRAW

The counter is used by drawing, so it is easy to put the increment in `Draw`:

```
func (g *Game) Update() error {
        return nil
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.spin++
        g.drawSheet(screen)
        g.drawPieces(screen)
        g.drawSpin(screen)
}
```

The same program with the counter in Draw, fifteen ticks in, on a display that paints three pictures a tick: the counter reads 45, the seam is at the bottom, and the small ball has rolled 45 pixels.

Fifteen ticks after the window opens, this display has painted 45 pictures. The counter reads 45, so the big ball shows frame 3 instead of frame 2. The small ball is at column 45 instead of column 15.

A display that paints three pictures per tick makes the animation run three times as fast. A machine that skips pictures slows it down. The counter changes game state, so it belongs in `Update`.

5. Using views and counts

A sub-image is a rectangle into a sheet. Ebitengine still draws it with `DrawImage`; only the source rectangle changes.

Animation uses the same draw call with a different rectangle. The rectangle is chosen by an integer that `Update` changes. Longer animations use more frames or a different divisor, but the idea stays the same.

6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Load a PNG into an Ebitengine image in three steps, once, before the window opens.
- Cut a sheet into sub-images from a written-down layout, and say what a sub-image shares with the sheet and what it copies.
- Draw sixteen pieces of one sheet at four times and know that it is one image on the graphics card.
- Pick an animation frame from a counter with one division and one remainder, and work out the frame on any tick by hand.
- Say why the counter lives in `Update`, with the numbers a three-pictures-a-tick display gives when it does not.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — spin the other way. Make the small ball roll left, from column 319 down, and make its seam walk the other way round to match.**

Draw it at `319 - g.spin%screenW`, and pick the frame as `(4 - frame) % 4`: frame 1 becomes 3, 3 becomes 1, and 0 and 2 stay, so the seam goes bottom, right, top and behind, which is the turn a ball rolling left makes.

▼ **Exercise 2 — a slower turn. Change `spinEvery` to 12 and watch the big ball. How many turns a second now, and how far does the small ball roll per turn?**

One and a quarter turns a second: 48 ticks a turn at sixty ticks a second. The small ball rolls 48 pixels a turn, twice what it rolled per turn before, and looks as if it were sliding. A rolling ball wants its frames to match its distance.

▼ **Exercise 3 — a seventeenth piece. Open the sheet in a pixel editor, draw an eight-by-eight picture of your own at (32, 0), where the sheet is empty, save it over `assets/pong-sheet.png`, and cut it out and draw it at eight times.**

One more field in `sprites`, `cut(32, 0, 8, 8)` in `cutSprites`, and a `drawAt` somewhere free, such as (272, 60) at eight times. Every other piece is untouched, because their rectangles do not overlap yours, and the volume's games still load the file. Keep the original sheet beside it, as `pong-sheet-mine.png` if you like, so that the pictures in the book still match what you see.

VOLUME 1 · CHAPTER 5

Text and the HUD

Digits from the sheet, a TrueType face for words, and a line centred by measuring it

1. Drawing text

This chapter draws a score and a pause message. The score uses the sprite sheet's block digits. The words use Go Regular, a TrueType font carried inside the program.

A number's width comes from its digit count. A word's width comes from the font. In both cases, centring means subtracting half the width from the middle column and drawing at a whole-pixel left edge. The pause overlay then shows the draw-order rule: later calls paint over earlier calls.

2. Drawing sheet digits

Put the digit font in `internal/digits`. Go allows only code inside the `gez` module to import an `internal` package, so the package is private to this project. It loads the sheet and cuts the ten glyphs out of row 24.

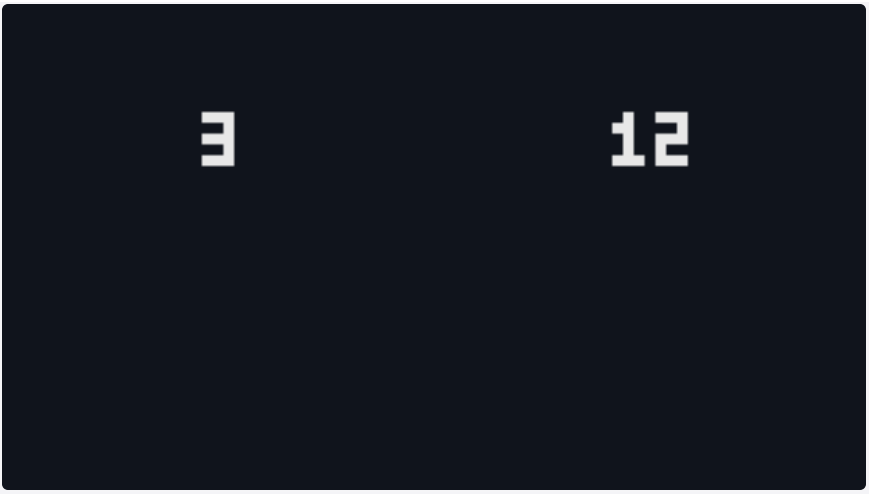
▣ BUILD · STAGE 1 — THE DIGIT FONT, AND TWO SCORES DRAWN WITH IT

```
// internal/digits/digits.go — create
// Package digits draws numbers with the sprite sheet's digit
font: ten
// glyphs, three pixels wide and five tall, on row 24 of the
sheet. Every
// game in this volume draws its score with it.
package digits
```

```
import (
        "image"
        "image/png"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
)

// Font is the ten digits, cut from the sheet.
type Font struct {
        glyph [10]*ebiten.Image
}

// The glyphs' size on the sheet, and the gap between two
digits, in the
// sheet's pixels; a scale multiplies all three.
const (
        GlyphW = 3
        GlyphH = 5
        Gap    = 1
)

// Load reads the sheet at path and cuts the ten digits out of
it.
func Load(path string) (*Font, error) {
        f, err := os.Open(path)
        if err ≠ nil {
                return nil, err
        }
        defer f.Close()
        img, err := png.Decode(f)
        if err ≠ nil {
                return nil, err
        }
        sheet := ebiten.NewImageFromImage(img)
        var font Font
        for i := range font.glyph {
                r := image.Rect(GlyphW*i, 24, GlyphW*i+GlyphW,
24+GlyphH)
                font.glyph[i] = sheet.SubImage(r).
(*ebiten.Image)
        }
        return &font, nil
}

// Width is how many pixels wide n is when drawn at scale.
func Width(n, scale int) int {
        count := 1
        for n ⩾ 10 {
                n ⁄= 10
                count++
        }
        return scale * (count*GlyphW + (count-1)*Gap)
```

```
}

// Draw draws n at scale with its left edge on column x and its
top on row y.
func (f *Font) Draw(dst *ebiten.Image, n int, x, y float64,
scale int) {
        var glyphs []int
        for n ⩾ 10 {
                glyphs = append([]int{n % 10}, glyphs...)
                n /= 10
        }
        glyphs = append([]int{n}, glyphs...)
        step := float64(scale * (GlyphW + Gap))
        for i, d := range glyphs {
                op := &ebiten.DrawImageOptions{}
                op.GeoM.Scale(float64(scale), float64(scale))
                op.GeoM.Translate(x+float64(i)*step, y)
                dst.DrawImage(f.glyph[d], op)
        }
}

// DrawCentred draws n at scale with its middle on column x and
its top on
// row y, rounding the left edge down to a whole pixel.
func (f *Font) DrawCentred(dst *ebiten.Image, n int, x, y
float64, scale int) {
        left := float64(int(x) - Width(n, scale)/2)
        f.Draw(dst, n, left, y, scale)
}


// cmd/score/main.go — create
package main

import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"

        "gez/internal/digits"
)

const (
        screenW = 320
        screenH = 180
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
)

type Game struct {
```

```
        font  *digits.Font
        left  int // the two scores on show
        right int
}

func (g *Game) Update() error {
        return nil
}

// drawScore draws the two numbers with the sheet's digits at
four times,
// centred on the middle of each half of the court.
func (g *Game) drawScore(screen *ebiten.Image) {
        screen.Fill(courtColor)
        g.font.DrawCentred(screen, g.left, 80, 40, 4)
        g.font.DrawCentred(screen, g.right, 240, 40, 4)
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawScore(screen)
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Score")
        ebiten.SetTPS(60)
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                log.Fatal(err)
        }
        g := &Game{font: font, left: 3, right: 12}
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}


go vet ./...
go run ./cmd/score
```



The window after stage 1: 3 and 12 in the sheet's digits at four times, each centred on the middle of its half.

`Draw` splits the number from the right. For 12, the remainder gives 2, then the next division leaves 1. The code prepends each digit so they draw in reading order.

Each glyph is drawn by scaling and then moving, the same `GeoM` order used for images. At four times scale, one digit takes twelve pixels and the gap takes four, so the next digit starts sixteen pixels to the right.

`Width` counts digits and applies the same arithmetic as the Math Interlude. `DrawCentred` subtracts half the width from the middle and rounds to a whole pixel. `0` is one glyph wide because a score of zero still needs to show one digit.

∑ MATH INTERLUDE — TWENTY-EIGHT PIXELS FOR TWO DIGITS

A number with two digits at four times is two glyphs of $3 \times 4 = 12$ pixels with one gap of $1 \times 4 = 4$ between them: $12 + 4 + 12 = 28$ pixels wide. Three digits are $12 + 4 + 12 + 4 + 12 = 44$, and one digit is 12: the gaps are one fewer than the digits. As a formula, for n digits at scale s, the width is $s \times (3n + (n - 1))$. Centred on

column 240, the 12 has its left edge at 240 − 28 ÷ 2 = 226 and its right edge at 254.

`n` how many digits the number has: 1 for 0 to 9, 2 for 10 to 99
`s` the scale: how many screen pixels one sheet pixel becomes; 4 on the scoreboard
`width` s × (3n + (n − 1)): glyphs three wide, gaps one wide, one fewer gap than glyphs
`left` the middle column less half the width, in whole pixels

3. Drawing TrueType text

Go Regular is the font published by the Go project. The package `golang.org/x/image/font/gofont/goregular` stores the TrueType file in a byte slice named `TTF`. Importing it puts the font bytes in the program. Add the module before extending `cmd/score`.

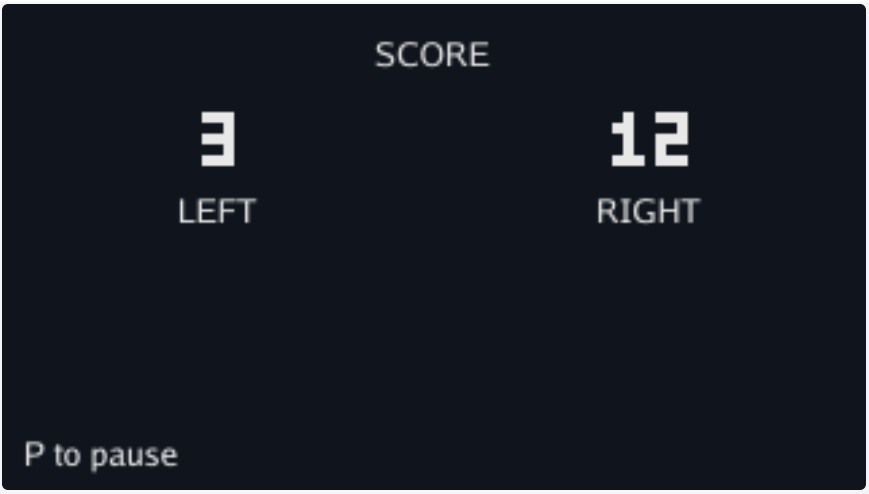
▣ BUILD · STAGE 2 — WORDS FROM A TRUETYPE FACE, EXTEND CMD/SCORE/MAIN.GO

```
// cmd/score/main.go — extend
import (
        "bytes"
        "image/color"
        "log"
        "math"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/text/v2"
        "golang.org/x/image/font/gofont/goregular"

        "gez/internal/digits"
)

type Game struct {
        font  *digits.Font
        face  *text.GoTextFace
        left  int // the two scores on show
        right int
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawScore(screen)
```

```
        g.drawWords(screen)
}

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Score")
        ebiten.SetTPS(60)
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                log.Fatal(err)
        }
        g := &Game{font: font, face: newFace(12), left: 3,
right: 12}
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}

// newFace parses the Go Regular font, which ships inside the
program as a
// Go package, and returns it at the size asked for, in pixels.
func newFace(size float64) *text.GoTextFace {
        src, err :=
text.NewGoTextFaceSource(bytes.NewReader(goregular.TTF))
        if err ≠ nil {
                log.Fatal(err)
        }
        return &text.GoTextFace{Source: src, Size: size}
}

// drawText draws s in the line colour with its left edge on
column x and
// its top on row y.
func drawText(dst *ebiten.Image, s string, face
*text.GoTextFace, x, y float64) {
        op := &text.DrawOptions{}
        op.GeoM.Translate(x, y)
        op.ColorScale.ScaleWithColor(lineColor)
        text.Draw(dst, s, face, op)
}

// drawCentred draws s with its middle on column x: the string
is measured
// and the left edge is half its width to the left, rounded down
to a pixel.
func drawCentred(dst *ebiten.Image, s string, face
*text.GoTextFace, x, y float64) {
        w, _ := text.Measure(s, face, 0)
        drawText(dst, s, face, math.Floor(x-w/2), y)
}

// drawWords draws the title and a line of small text under each
number.
func (g *Game) drawWords(screen *ebiten.Image) {
```

```
        drawCentred(screen, "SCORE", g.face, screenW/2, 12)
        drawCentred(screen, "LEFT", g.face, 80, 70)
        drawCentred(screen, "RIGHT", g.face, 240, 70)
        drawText(screen, "P to pause", g.face, 8, 160)
}


go get golang.org/x/image@v0.43.0
go mod tidy
go mod vendor
cat go.mod
go vet ./...
go run ./cmd/score
```

```
$ go get golang.org/x/image@v0.43.0

$ go mod tidy

$ go mod vendor

$ cat go.mod

module gez

go 1.26

require (
        github.com/hajimehoshi/ebiten/v2 v2.9.11
        golang.org/x/image v0.43.0
)

require (
        github.com/ebitengine/gomobile v0.0.0-20250923094054-
ea854a63cce1 // indirect
        github.com/ebitengine/hideconsole v1.0.0 // indirect
        github.com/ebitengine/purego v0.9.0 // indirect
        github.com/go-text/typesetting v0.3.0 // indirect
        github.com/jezek/xgb v1.1.1 // indirect
        github.com/rivo/uniseg v0.4.7 // indirect
        golang.org/x/sync v0.21.0 // indirect
        golang.org/x/sys v0.44.0 // indirect
        golang.org/x/text v0.38.0 // indirect
)
$ go vet ./...
```



The window after stage 2: the numbers from the sheet, and four lines of words from the TrueType face at twelve pixels.

`go get` prints nothing here because `golang.org/x/image` was already in the build list through Ebitengine's own tests. The command still writes it into `go.mod` as a direct requirement. `go mod tidy` finds the modules needed by the text package, and `go mod vendor` copies them into `vendor`.

`newFace` parses the font once in `main`. `text.GoTextFace` pairs the parsed font with a size in pixels. `text.Draw`, from Ebitengine's text package, takes draw options with a `GeoM` for position and a colour scale for the glyphs.

The TrueType words have soft edges because a curve crosses partial pixels. The sheet digits keep hard edges because they are tiny images drawn at an integer scale. Use the sheet for scores and the font for words.

`text.Measure` returns the width and height of a string in this face. `drawCentred` subtracts half the width and uses `math.Floor` so the glyphs start on a whole pixel.

4. Drawing an overlay

▣ BUILD · STAGE 3 — THE PAUSE OVERLAY, EXTEND CMD/SCORE/MAIN.GO

```
// cmd/score/main.go — extend
import (
        "bytes"
        "image/color"
        "log"
        "math"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/inpututil"
        "github.com/hajimehoshi/ebiten/v2/text/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
        "golang.org/x/image/font/gofont/goregular"

        "gez/internal/digits"
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
        shadeColor = color.RGBA{R: 0, G: 0, B: 0, A: 160}
)

type Game struct {
        font   *digits.Font
        face   *text.GoTextFace
        left   int // the two scores on show
        right  int
        paused bool
}

func (g *Game) Update() error {
        if inpututil.IsKeyJustPressed(ebiten.KeyP) {
                g.paused = !g.paused
        }
        return nil
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawScore(screen)
        g.drawWords(screen)
        g.drawPause(screen)
}

// drawPause darkens everything drawn so far and writes over it.
func (g *Game) drawPause(screen *ebiten.Image) {
```

```
        if !g.paused {
                return
        }
        vector.FillRect(screen, 0, 0, screenW, screenH,
shadeColor, false)
        drawCentred(screen, "PAUSED", g.face, screenW/2, 84)
}


go vet ./...
go run ./cmd/score
```

The window after stage 3 with P pressed once: the court, the numbers and the words under a shade, and PAUSED on top of it.

Press P and the court dims. Press P again and the picture comes back. The toggle uses `IsKeyJustPressed` so a held P changes the flag once.

The shade is one rectangle the size of the picture. Its alpha is 160 out of 255, so it covers about 63 percent of what is under it. The line colour at 232 comes out near 86 after the shade, dim but still visible.

A translucent colour blends with the pixel already on the screen. The word is drawn after the shade, so it stays at full brightness.

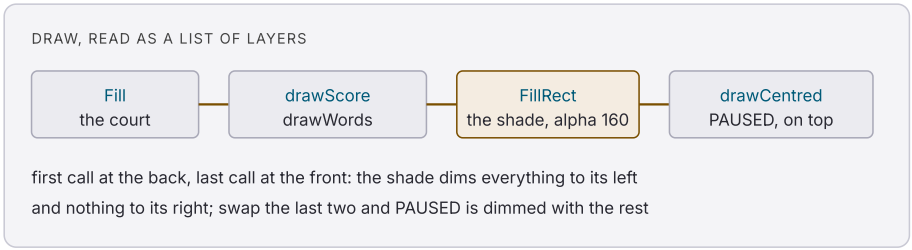


Figure 5.1 — the four calls of a paused frame in the order they run, which is the order of the layers, back to front.

⚠ **WORKED FAILURE — PAUSED DRAWN BEFORE THE SHADE**

Swap the two lines in `drawPause`, writing the word first and shading after:

```
func (g *Game) drawPause(screen *ebiten.Image) {
        if !g.paused {
                return
        }
        drawCentred(screen, "PAUSED", g.face, screenW/2, 84)
        vector.FillRect(screen, 0, 0, screenW, screenH,
shadeColor, false)
}
```

The same program with the word drawn first: PAUSED is under the shade, as faint as the score.

The word is there, but it is as dim as the numbers. The shade darkens whatever the pixels hold when it is drawn, and the word was already in those pixels.

`screen` is one image. Each call paints over what is already there. There is no separate layer called "top"; the order of the calls creates the order you see. Put the shade back before the word.

5. Ordering draw calls

`Draw` paints into one image. The fill at the top gives each picture a fresh background. Opaque colours replace pixels; translucent colours mix with the pixels already there.

Read `Draw` from back to front: background first, game objects after that, then scores, then overlays. If two things overlap, the later call is the one you see on top.

6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Cut a ten-glyph digit font from the sheet into a package every game can import, and draw any number with it at any scale.
- Work out the width of a number in the digit font by hand, and centre it on a column in whole pixels.
- Add `golang.org/x/image` to the module, and say why `go get` printed nothing and what `go mod tidy` added.
- Parse Go Regular from the bytes inside the program, make a face at a size, draw a string in the line colour, and centre it by measuring it.
- Draw a translucent shade and say, from the alpha, how bright the digits under it come out.

- Explain from the order of two calls why a word drawn before a shade is dim, and read any `Draw` as a back-to-front list.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — a score that climbs. Make the left score go up by one every sixty ticks, and watch what happens to the digits when it reaches 10 and 100.**

A tick counter in `Update`, and `g.left++` when it hits a multiple of sixty. At 10 the number gets a second glyph and its left edge moves eight pixels left, because `DrawCentred` re-centres it; at 100 it moves eight more. The right edge moves the same amount right. A scoreboard that kept its left edge fixed would drift toward the middle instead.

▼ **Exercise 2 — the height the face reports. Print the width and height `text.Measure` returns for "SCORE" at 12, then at 24, and compare them with the pixels on screen.**

Add a `fmt.Println` of the two values in `main`; at 12 the height is the face's line height, a little more than twelve, and the width is near 40; at 24 both double. The height is the room a line needs, not the height of the capitals, which is why the words are placed by their top and the number under them by a row chosen by eye.

▼ **Exercise 3 — a lighter shade. Change the shade's alpha to 60, then 250, and say what PAUSED looks like on each.**

At 60 the court is barely dimmed, to about three quarters, and the word sits on a picture that still reads as the game; at 250 the court is nearly black and the word floats alone. 160 is a

choice between the two, made so that a player can see the state they paused in.

VOLUME 1 · CHAPTER 6

Sound

One audio context, a player per sound, and a beep on the tick the game says

1. Playing a sound

This chapter plays a WAV file when Space is pressed. It also plays a second sound on a timer, once every sixty ticks.

Ebitengine's audio package provides the context and players. A program opens one context, decodes each sound once, makes one player for that sound and plays it by rewinding the player and starting it. The worked failure shows what Ebitengine reports when the program tries to make a new player from the same stream on each press.

2. Loading one player

Download the two sounds into `assets/`, beside the sheet. The first program plays one of them, on Space, and lights a square for ten ticks so that the tick it played on can be seen as well as heard.

▣ BUILD · STAGE 1 — A BEEP ON SPACE

hit.wav (WAV, 22,050 Hz mono 16-bit, 60 ms at 880 Hz; the hit)

point.wav (WAV, 22,050 Hz mono 16-bit, 180 ms at 440 Hz; the point)

```
// cmd/beep/main.go — create
package main

import (
        "bytes"
```

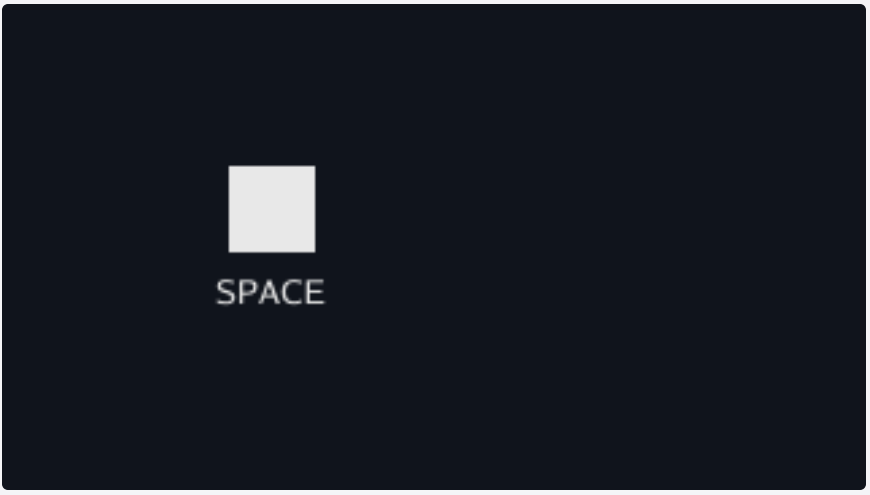
```
        "image/color"
        "log"
        "math"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/audio"
        "github.com/hajimehoshi/ebiten/v2/audio/wav"
        "github.com/hajimehoshi/ebiten/v2/inpututil"
        "github.com/hajimehoshi/ebiten/v2/text/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
        "golang.org/x/image/font/gofont/goregular"
)

const (
        screenW    = 320
        screenH    = 180
        sampleRate = 22050 // samples a second: the rate the
sounds were written at
        litFor     = 10    // ticks a light stays lit after its
sound plays
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
        dimColor   = color.RGBA{R: 60, G: 66, B: 80, A: 255}
)

func newFace(size float64) *text.GoTextFace {
        src, err :=
text.NewGoTextFaceSource(bytes.NewReader(goregular.TTF))
        if err ≠ nil {
                log.Fatal(err)
        }
        return &text.GoTextFace{Source: src, Size: size}
}

func drawCentred(dst *ebiten.Image, s string, face
*text.GoTextFace, x, y float64) {
        w, _ := text.Measure(s, face, 0)
        op := &text.DrawOptions{}
        op.GeoM.Translate(math.Floor(x-w/2), y)
        op.ColorScale.ScaleWithColor(lineColor)
        text.Draw(dst, s, face, op)
}

// keys is what the player is doing this tick.
type keys struct {
        hit bool // Space, on the tick it went down
}

func readKeys() keys {
        return keys{hit:
```

```
inpututil.IsKeyJustPressed(ebiten.KeySpace)}
}

type Game struct {
        face   *text.GoTextFace
        hit    *audio.Player // the hit sound, decoded once
        hitLit int           // ticks the light has left
}

// step advances the game one tick: a hit on Space.
func (g *Game) step(k keys) {
        if k.hit {
                g.hit.Rewind()
                g.hit.Play()
                g.hitLit = litFor
        }
        if g.hitLit > 0 {
                g.hitLit--
        }
}

func (g *Game) Update() error {
        g.step(readKeys())
        return nil
}

// drawLight draws a square that is lit while its sound is
fresh.
func drawLight(screen *ebiten.Image, x float64, lit int, label
string, face *text.GoTextFace) {
        c := dimColor
        if lit > 0 {
                c = lineColor
        }
        vector.FillRect(screen, float32(x)-16, 60, 32, 32, c,
false)
        drawCentred(screen, label, face, x, 100)
}

func (g *Game) Draw(screen *ebiten.Image) {
        screen.Fill(courtColor)
        drawLight(screen, 100, g.hitLit, "SPACE", g.face)
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Beep")
        ebiten.SetTPS(60)
        ctx := audio.NewContext(sampleRate)
```

```
        data, err := os.ReadFile("assets/hit.wav")
        if err ≠ nil {
                log.Fatal(err)
        }
        stream, err := wav.DecodeWithSampleRate(sampleRate,
bytes.NewReader(data))
        if err ≠ nil {
                log.Fatal(err)
        }
        hit, err := ctx.NewPlayer(stream)
        if err ≠ nil {
                log.Fatal(err)
        }
        g := &Game{face: newFace(12), hit: hit}
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}


go mod tidy
go mod vendor
cat go.mod
go vet ./...
go run ./cmd/beep
```

```
$ go mod tidy

$ go mod vendor

$ cat go.mod

module gez

go 1.26

require (
        github.com/hajimehoshi/ebiten/v2 v2.9.11
        golang.org/x/image v0.43.0
)

require (
        github.com/ebitengine/gomobile v0.0.0-20250923094054-
ea854a63cce1 // indirect
        github.com/ebitengine/hideconsole v1.0.0 // indirect
        github.com/ebitengine/oto/v3 v3.4.1 // indirect
        github.com/ebitengine/purego v0.9.0 // indirect
        github.com/go-text/typesetting v0.3.0 // indirect
        github.com/jezek/xgb v1.1.1 // indirect
        github.com/rivo/uniseg v0.4.7 // indirect
        golang.org/x/sync v0.21.0 // indirect
        golang.org/x/sys v0.44.0 // indirect
```

```
        golang.org/x/text v0.38.0 // indirect
)
$ go vet ./...
```



The window after stage 1, three ticks after a tap of Space: the light is lit, and the beep has just sounded.

Tap Space and the beep sounds. The square lights for a sixth of a second so you can see the tick that played it.

Tap Space twice quickly and the sound starts twice. The second tap cuts the first beep short because `Rewind` moves the player back to the first sample before `Play` starts it. A player at the end of its samples must be rewound before it can play again.

The audio packages bring in `oto`, the module Ebitengine uses to talk to the sound device. `go mod tidy` and `go mod vendor` add and vendor it.

`main` opens the context at 22,050 samples a second, the rate of these sounds. It reads the WAV file into memory. `wav.DecodeWithSampleRate` turns the bytes into samples the context can read, and `ctx.NewPlayer` makes the player kept in the game.

⚙ TOOL — AN AUDIO CONTEXT AND A PLAYER

`audio.NewContext(rate)` opens the machine's sound device at a sample rate; there is one per program, and the library reports a device it cannot open as the error `RunGame` returns. `wav.DecodeWithSampleRate(rate, r)` reads a WAV from a reader and returns a stream in the context's format. `ctx.NewPlayer(stream)` makes a player; `Rewind` puts it back to the start, `Play` starts it, and a player that is already playing when `Play` is called carries on. The reference is `pkg.go.dev/github.com/hajimehoshi/ebiten/v2/audio`.

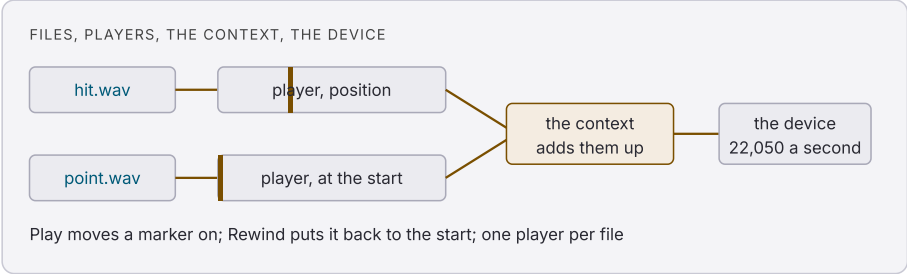


Figure 6.1 — two files decoded once into two players, one context adding up whatever is playing, and the device consuming a steady stream.

⚠ WORKED FAILURE — A NEW PLAYER ON EVERY PRESS

The player is used in `step`. Making it there from the decoded stream looks tidy, but it makes a second player from the same stream:

```
type Game struct {
        face   *text.GoTextFace
        ctx    *audio.Context
        stream *wav.Stream // the hit sound, decoded once
        hitLit int         // ticks the light has left
}

// step advances the game one tick: a hit on Space.
func (g *Game) step(k keys) {
        if k.hit {
```

```
                p, err := g.ctx.NewPlayer(g.stream)
                if err ≠ nil {
                        log.Fatal(err)
                }
                p.Play()
                g.hitLit = litFor
        }
        if g.hitLit > 0 {
                g.hitLit--
        }
}
```

with `main` keeping the context and the stream in the game instead of a player. The first tap beeps. The second tap, a quarter of a second later, ends the program:

```
$ go run ./cmd/beep

audio: audio error: audio: the same source must not be used by
multiple Player objects
```

That line follows the date and time printed by `log.Fatal`. `go run` then reports `exit status 1`.

The message names the problem. Two players were made from one stream. A stream is a position in the decoded samples, so two players would fight over where that position is. Ebitengine refuses the second player. Keep one player and rewind it.

3. Playing on a timer

The second sound plays once a second. At sixty ticks a second, that means every sixtieth tick. The counter that decides it lives in `Update`.

Move the loading code into `internal/sound`. The package opens the context when the first sound loads and hides the rewind inside `Play`.

▣ BUILD · STAGE 2 — THE SOUND PACKAGE, AND A POINT EVERY SECOND

```
// internal/sound/sound.go — create
// Package sound plays short WAV files: one audio context for
the whole
// program, one player for each sound, made once, and started
from the
// beginning on the tick a game says. Every game in this volume
plays its
// sounds with it.
package sound

import (
        "bytes"
        "os"

        "github.com/hajimehoshi/ebiten/v2/audio"
        "github.com/hajimehoshi/ebiten/v2/audio/wav"
)

// SampleRate is the rate the volume's sounds were written at,
and the rate
// the audio context is opened at.
const SampleRate = 22050

// context is the program's one audio context, opened by the
first Load.
// A program can open only one, and every player lives in it.
var context *audio.Context

// Sound is one sound a game can play.
type Sound struct {
        player *audio.Player
}

// Load reads the WAV file at path, decodes it, and makes a
player for it
// in the program's audio context, opening the context if this
is the first
// sound loaded.
func Load(path string) (*Sound, error) {
        if context == nil {
                context = audio.NewContext(SampleRate)
        }
        data, err := os.ReadFile(path)
        if err ≠ nil {
                return nil, err
        }
        stream, err := wav.DecodeWithSampleRate(SampleRate,
bytes.NewReader(data))
        if err ≠ nil {
                return nil, err
        }
        player, err := context.NewPlayer(stream)
        if err ≠ nil {
```

```
                return nil, err
        }
        return &Sound{player: player}, nil
}

// Play plays the sound from its start. A sound still playing
starts again.
func (s *Sound) Play() {
        s.player.Rewind()
        s.player.Play()
}


// cmd/beep/main.go — extend
import (
        "bytes"
        "image/color"
        "log"
        "math"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/inpututil"
        "github.com/hajimehoshi/ebiten/v2/text/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
        "golang.org/x/image/font/gofont/goregular"

        "gez/internal/sound"
)

const (
        screenW = 320
        screenH = 180
)

type Game struct {
        face       *text.GoTextFace
        hit, point *sound.Sound
        tick       int
        hitLit     int // ticks the hit light has left
        pointLit   int
}

// step advances the game one tick: a hit on Space, a point
every second.
func (g *Game) step(k keys) {
        g.tick++
        if k.hit {
                g.hit.Play()
                g.hitLit = litFor
        }
        if g.tick%pointEvery == 0 {
                g.point.Play()
                g.pointLit = litFor
        }
```

```
        if g.hitLit > 0 {
                g.hitLit--
        }
        if g.pointLit > 0 {
                g.pointLit--
        }
}

func (g *Game) Draw(screen *ebiten.Image) {
        screen.Fill(courtColor)
        drawLight(screen, 100, g.hitLit, "SPACE", g.face)
        drawLight(screen, 220, g.pointLit, "EVERY SECOND",
g.face)
}

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Beep")
        ebiten.SetTPS(60)
        hit, err := sound.Load("assets/hit.wav")
        if err ≠ nil {
                log.Fatal(err)
        }
        point, err := sound.Load("assets/point.wav")
        if err ≠ nil {
                log.Fatal(err)
        }
        g := &Game{face: newFace(12), hit: hit, point: point}
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}

// A light stays lit for ten ticks after its sound plays; a
point sounds
// every sixty ticks.
const (
        litFor     = 10
        pointEvery = 60
)


go vet ./...
go run ./cmd/beep
```

The window on the sixtieth tick after stage 2 starts: the point light lit and its lower tone sounding, the hit light dim.

The lower tone sounds once a second and the right light blinks with it. Space still plays the hit sound on the left. If a tap lands near a second boundary, both sounds play because the context mixes every player that is playing.

The extend listing removes `os`, `audio` and `wav` from `cmd/beep`, because `internal/sound` now does that work. It also moves the sample rate into the package and keeps the light and timer constants beside the code that uses them.

`sound.Load` reads a file, decodes it and makes one player. The context lives in a package variable, so the program loads a sound in one call and never handles the context directly.

4. Mixing players

The sound device receives a steady stream of samples. The context supplies that stream and adds together the samples from every player that is playing. Silence is zeros.

A player is a position in one decoded sound. Playing advances the position, reaching the end stops it, and rewinding moves it back to the start. Decoding once and rewinding keeps sound playback tied to the tick that caused it.

5. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Open the program's one audio context at the sounds' sample rate and say what a second `NewContext` does.
- Read a WAV file, decode it into a stream, and make one player from it, in the order those three happen.
- Play a sound on the tick a key goes down and on every sixtieth tick, with the counter where chapter 1 put it.
- Say why a new player per press ends the program, from the error's own words, and why rewinding the one player is the fix.
- Load a sound through `internal/sound` in two lines, and know what the package keeps that the game never sees.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — a faster beat. Change `pointEvery` to 20, run it, and count the tones in ten seconds.**

Thirty: three a second, because sixty ticks a second divided by twenty ticks a tone is three. The tone is 180 milliseconds long and the gap between starts is 333, so each finishes before the next begins and none is cut short.

▼ **Exercise 2 — a quieter point. Give `Sound` a `SetVolume(v float64)` method that passes `v` to the player, and play the point at a fifth of its volume.**

`func (s *Sound) SetVolume(v float64) { s.player.SetVolume(v) }`, and `point.SetVolume(0.2)` in `main` after the load. The volume is a scale from 0 to 1 on every sample the player contributes to the mix, and it stays set across rewinds, so the tone is quiet every second.

▼ **Exercise 3 — the two-key chord. Play the point on S as well as on the timer, and tap S and Space together.**

A second field in `keys` read with `IsKeyJustPressed(ebiten.KeyS)`, and `g.point.Play()` with the point light in `step` when it is set. Both tones sound at once, the higher one ending after sixty milliseconds and the lower one carrying on, which is the mixer adding two players' samples in the same stream.

VOLUME 1 · CHAPTER 7

Pong

Two paddles, a ball, a serve, a score, and a match — the whole game in one chapter

1. Building Pong

This chapter builds Pong as a complete game. It has two paddles, a ball, a serve, a score, a match to seven, solo mode and two sounds.

Put the rules in `Match`. Its `Step` method changes the match once per tick from the keys held on that tick. The window code reads the match, draws it with Ebitengine and plays sounds when the match reports an event.

2. Moving two paddles

▣ BUILD · STAGE 1 — THE MATCH'S FIRST TWO RULES, AND THE WINDOW THAT DRAWS THEM

```
// cmd/pong/match.go — create
package main

// The court is the whole picture, with a border two pixels wide
just
// inside its edge. Everything that moves stays inside the
border.
const (
        courtW = 320
        courtH = 180
        border = 2
)

// A paddle is four pixels wide and twenty-four tall, moves two
pixels a
// tick, and never leaves the court.
const (
```

```
        paddleW     = 4
        paddleH     = 24
        paddleSpeed = 2
        leftX       = 8                     // the left paddle's
column
        rightX      = courtW - 8 - paddleW // the right paddle's
column, 308
)

// Paddle is one side's bat: the position of its top-left
corner, in pixels.
type Paddle struct {
        X, Y float64
}

// newPaddle returns a paddle on column x, centred on the court.
func newPaddle(x float64) Paddle {
        return Paddle{X: x, Y: (courtH - paddleH) / 2}
}

// Move steps the paddle two pixels up, two pixels down, or not
at all, and
// keeps it inside the border.
func (p *Paddle) Move(up, down bool) {
        if up {
                p.Y -= paddleSpeed
        }
        if down {
                p.Y += paddleSpeed
        }
        p.Y = clamp(p.Y, border, courtH-border-paddleH)
}

// clamp returns v held inside [lo, hi].
func clamp(v, lo, hi float64) float64 {
        if v < lo {
                return lo
        }
        if v > hi {
                return hi
        }
        return v
}

// The ball is four pixels square. Its position is its top-left
corner and
// its velocity is in pixels a tick.
const ballSize = 4

// Ball is the ball: where it is and how far it moves each tick.
type Ball struct {
        X, Y   float64
        VX, VY float64
}
```

```
// newBall returns a ball at rest in the centre of the court.
func newBall() Ball {
        return Ball{X: (courtW - ballSize) / 2, Y: (courtH -
ballSize) / 2}
}

// Match is the state of one game, advanced one tick at a time
by Step.
type Match struct {
        Left  Paddle // W and S
        Right Paddle // Up and Down
        Ball  Ball
}

// newMatch returns a match at tick zero: both paddles centred,
the ball at
// rest in the middle of the court.
func newMatch() *Match {
        return &Match{Left: newPaddle(leftX), Right:
newPaddle(rightX), Ball: newBall()}
}

// movePaddles moves the left paddle by W and S and the right by
Up and
// Down.
func (m *Match) movePaddles(k keys) {
        m.Left.Move(k.w, k.s)
        m.Right.Move(k.up, k.down)
}

// Step advances the match by one tick with the keys held during
it.
func (m *Match) Step(k keys) {
        m.movePaddles(k)
}


// cmd/pong/main.go — create
package main

import (
        "image"
        "image/color"
        "image/png"
        "log"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
```

```
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
)

// keys is what the players are doing this tick: six keys, held
or not.
type keys struct {
        w, s, up, down, space, r bool
}

// readKeys asks Ebitengine about the six keys, once a tick.
func readKeys() keys {
        return keys{
                w:     ebiten.IsKeyPressed(ebiten.KeyW),
                s:     ebiten.IsKeyPressed(ebiten.KeyS),
                up:    ebiten.IsKeyPressed(ebiten.KeyArrowUp),
                down:  ebiten.IsKeyPressed(ebiten.KeyArrowDown),
                space: ebiten.IsKeyPressed(ebiten.KeySpace),
                r:     ebiten.IsKeyPressed(ebiten.KeyR),
        }
}

// newCourt paints the court once, into an image the game copies
to the
// screen every frame: the fill, a border two pixels wide just
inside the
// edge, and a net of dashes down the middle.
func newCourt() *ebiten.Image {
        img := ebiten.NewImage(courtW, courtH)
        img.Fill(courtColor)
        vector.StrokeRect(img, 1, 1, courtW-2, courtH-2, 2,
lineColor, false)
        for y := 0; y < courtH; y += 8 {
                vector.FillRect(img, 159, float32(y), 2, 4,
lineColor, false)
        }
        return img
}

// sprites is the sheet cut into the pictures the game draws.
type sprites struct {
        left, right *ebiten.Image
        ball        [4]*ebiten.Image
}

// loadSprites reads the sheet and cuts the paddles and the
ball's four
// spin frames out of it.
func loadSprites(path string) (*sprites, error) {
        f, err := os.Open(path)
        if err ≠ nil {
                return nil, err
        }
        defer f.Close()
        img, err := png.Decode(f)
```

```
        if err ≠ nil {
                return nil, err
        }
        sheet := ebiten.NewImageFromImage(img)
        cut := func(x, y, w, h int) *ebiten.Image {
                return sheet.SubImage(image.Rect(x, y, x+w,
y+h)).(*ebiten.Image)
        }
        s := &sprites{left: cut(0, 0, 4, 24), right: cut(4, 0,
4, 24)}
        for i := range s.ball {
                s.ball[i] = cut(8+4*i, 0, 4, 4)
        }
        return s, nil
}

// drawSprite draws img with its top-left corner at (x, y),
rounded down to
// the pixel the corner is in: the rules keep fractions of a
pixel and the
// screen has none.
func drawSprite(dst, img *ebiten.Image, x, y float64) {
        op := &ebiten.DrawImageOptions{}
        op.GeoM.Translate(float64(int(x)), float64(int(y)))
        dst.DrawImage(img, op)
}

// Game is Pong's window: the match, and everything drawn.
type Game struct {
        match *Match
        court *ebiten.Image
        sp    *sprites
}

// newGame builds the match and loads what the window draws.
func newGame() (*Game, error) {
        sp, err := loadSprites("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        return &Game{match: newMatch(), court: newCourt(), sp:
sp}, nil
}

// step advances the window's game one tick.
func (g *Game) step(k keys) {
        g.match.Step(k)
}

func (g *Game) Update() error {
        g.step(readKeys())
        return nil
}
```

```
// drawMatch draws what the rules say is where: the two paddles
and the
// ball.
func (g *Game) drawMatch(screen *ebiten.Image) {
        m := g.match
        screen.DrawImage(g.court, nil)
        drawSprite(screen, g.sp.left, m.Left.X, m.Left.Y)
        drawSprite(screen, g.sp.right, m.Right.X, m.Right.Y)
        drawSprite(screen, g.sp.ball[0], m.Ball.X, m.Ball.Y)
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawMatch(screen)
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Pong")
        ebiten.SetTPS(60)
        g, err := newGame()
        if err ≠ nil {
                log.Fatal(err)
        }
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}


go vet ./...
go run ./cmd/pong
```

The window after stage 1, half a second in, with W held for the first twenty ticks and Down for the first thirty: the left paddle 40 pixels up, the right 60 down, the ball at rest in the middle.

`match.go` imports nothing from Ebitengine. It holds the court size, the paddles, the ball and the rule that moves the paddles.

`main.go` handles the window. It reads six booleans into `keys`, draws the court image, cuts the sheet into sprites and draws them with `DrawImage`. `drawSprite` rounds positions to whole pixels so the rules can use `float64` positions without choosing a screen row.

The two paddles differ on the sheet by one lit column each, the one facing the net, so the sprites are cut as a left and a right and not one paddle drawn twice. `Move` is chapter 3's clamp with the paddle's own numbers, and `rightX` is worked out from the court's width so that the two paddles sit eight pixels in from either edge. Hold W and Up together and both paddles climb; the picture above was made by holding W for a third of a second and Down for half of one.

3. Bouncing the ball

▣ BUILD · STAGE 2 — THE BALL MOVES, AND SPINS AS IT GOES, EXTEND BOTH FILES

```
// cmd/pong/match.go — extend
// newMatch returns a match at tick zero: both paddles centred,
and the
// ball flying from the middle of the court, right and a little
down.
func newMatch() *Match {
        m := &Match{Left: newPaddle(leftX), Right:
newPaddle(rightX), Ball: newBall()}
        m.Ball.VX, m.Ball.VY = 2, 1
        return m
}

// Step advances the match by one tick with the keys held during
it.
func (m *Match) Step(k keys) {
        m.movePaddles(k)
        m.Ball.Move()
}

// Move advances the ball one tick and reflects it off the top
and bottom
// borders: whatever part of the step went past the border comes
back.
func (b *Ball) Move() {
        b.X += b.VX
        b.Y += b.VY
        const top, bottom = border, courtH - border - ballSize
        if b.Y < top {
                b.Y = 2*top - b.Y
                b.VY = -b.VY
        }
        if b.Y > bottom {
                b.Y = 2*bottom - b.Y
                b.VY = -b.VY
        }
}


// cmd/pong/main.go — extend
// Game is Pong's window: the match, and everything drawn.
type Game struct {
        match *Match
        court *ebiten.Image
        sp    *sprites
        spin  int // ticks the ball has been moving
}
```

```
// step advances the window's game one tick: the match, and the
ball's spin.
func (g *Game) step(k keys) {
        g.match.Step(k)
        if b := g.match.Ball; b.VX ≠ 0 || b.VY ≠ 0 {
                g.spin++
        }
}

// drawMatch draws what the rules say is where: the two paddles
and the
// ball's current spin frame.
func (g *Game) drawMatch(screen *ebiten.Image) {
        m := g.match
        screen.DrawImage(g.court, nil)
        drawSprite(screen, g.sp.left, m.Left.X, m.Left.Y)
        drawSprite(screen, g.sp.right, m.Right.X, m.Right.Y)
        drawSprite(screen, g.sp.ball[ballFrame(g.spin,
m.Ball.VX)], m.Ball.X, m.Ball.Y)
}

// The ball turns one frame every six ticks while it moves, the
other way
// round when it moves the other way.
const spinEvery = 6

// ballFrame picks the spin frame for a ball that has been
moving for spin
// ticks, turning with the direction it travels.
func ballFrame(spin int, vx float64) int {
        f := (spin / spinEvery) % 4
        if vx < 0 {
                f = (4 - f) % 4
        }
        return f
}


go vet ./...
go run ./cmd/pong
```

The window seventy ticks after stage 2 starts: the ball left the middle flying right and down, reached the bottom border on tick 66, bounced off it on tick 67, and is on its way up.

The ball starts in the middle and moves two pixels right and one pixel down each tick. When its top edge would pass the bottom border, `Move` reflects the extra distance back inside the court and flips the vertical speed.

On tick 67 the ball would be one pixel past the lowest row it may occupy. Reflection puts it one pixel inside instead. Stopping it on the border would lose part of the step and make faster balls hesitate at the wall.

The spin counter counts only while the ball moves, so a resting ball holds its frame. The counter belongs to the window code because it changes only the picture, not the match rules.

4. Returning from the paddle

A paddle returns the ball. Where the ball struck the paddle decides the angle it leaves at: straight back from the paddle's middle, up to sixty degrees off straight from either end, so that a player can aim by

where they meet it. Each return adds a quarter of a pixel a tick to the ball's speed, so a long rally gets faster. And the test for a strike is the one the rectangles of chapter 2 make easy: after the ball has moved, does its four-by-four rectangle overlap the paddle's four-by-twenty-four?

▣ BUILD · STAGE 3 — CONTACT, AND THE ANGLE OF THE RETURN, EXTEND CMD/PONG/MATCH.GO

```
// cmd/pong/match.go — extend
import "math"

// Step advances the match by one tick with the keys held during
it.
func (m *Match) Step(k keys) {
        m.movePaddles(k)
        m.fly()
}

// Rect is an axis-aligned rectangle: its top-left corner, width
and height.
type Rect struct {
        X, Y, W, H float64
}

// Overlaps reports whether two rectangles share any area.
Rectangles that
// only touch along an edge do not overlap.
func (r Rect) Overlaps(o Rect) bool {
        return r.X < o.X+o.W && o.X < r.X+r.W && r.Y < o.Y+o.H
&& o.Y < r.Y+r.H
}

// Rect is the paddle's rectangle.
func (p Paddle) Rect() Rect {
        return Rect{X: p.X, Y: p.Y, W: paddleW, H: paddleH}
}

// Rect is the ball's rectangle.
func (b Ball) Rect() Rect {
        return Rect{X: b.X, Y: b.Y, W: ballSize, H: ballSize}
}

// A paddle returns the ball at an angle set by where the ball
struck it, up
// to sixty degrees from straight, and a quarter of a pixel a
tick faster
// than it arrived.
const (
```

```
        maxAngle = 60.0 // degrees, at the paddle's ends
        speedUp  = 0.25 // pixels a tick added on every return
)

// Return sends the ball away from paddle p: dir is +1 to send
it right,
// off the left paddle, and -1 to send it left. The angle comes
from how
// far the ball's centre is from the paddle's centre, the speed
grows by
// speedUp, and the ball is placed against the paddle's face.
func (b *Ball) Return(p Paddle, dir float64) {
        offset := ((b.Y + ballSize/2) - (p.Y + paddleH/2)) /
(paddleH / 2)
        offset = clamp(offset, -1, 1)
        angle := offset * maxAngle * math.Pi / 180
        speed := math.Hypot(b.VX, b.VY) + speedUp
        b.VX = dir * speed * math.Cos(angle)
        b.VY = speed * math.Sin(angle)
        if dir > 0 {
                b.X = p.X + paddleW
        } else {
                b.X = p.X - ballSize
        }
}

// fly moves the ball one tick and returns it off a paddle it
met,
// reporting whether it met one.
func (m *Match) fly() bool {
        m.Ball.Move()
        if m.Ball.Rect().Overlaps(m.Right.Rect()) {
                m.Ball.Return(m.Right, -1)
                return true
        }
        if m.Ball.Rect().Overlaps(m.Left.Rect()) {
                m.Ball.Return(m.Left, +1)
                return true
        }
        return false
}


go vet ./...
go run ./cmd/pong
```

The window eighty ticks after stage 3 starts, in a rally between two players: the right paddle went down to meet the ball, returned it on tick 74, and six ticks later the ball is on its way back at an angle.

Hold Down as the ball comes and the right paddle meets it, and the ball comes back at an angle that depends on where on the paddle it landed: near the top of the paddle it goes up, near the middle it goes straight, and it is a quarter of a pixel a tick faster than it was. In the picture, the right paddle came down to meet the ball below its middle and sent it back down and to the left. The first file has its first import, `math`, for the cosine and sine that turn an angle into a velocity and the `Hypot` that turns a velocity back into a speed. A player who never moves never returns the ball, because the ball was launched at a row the centred paddle does not cover, and it leaves on the right; a serve is stage 5's job.

`Return` ends by placing the ball against the paddle's face. The overlap test fires when the ball is already partly inside the paddle, and a ball left there would overlap it again on the next tick, be returned again, and rattle. Putting its edge on the face means the next tick's step carries it clear. `fly` tests the right paddle first and the left second and stops at the first strike; the ball cannot be inside both, so the order never matters.

∑ MATH INTERLUDE — WHERE IT STRUCK, AS AN ANGLE

The paddle is 24 pixels tall, so its middle is 12 pixels from either end. A ball whose centre is 6 pixels below the paddle's middle is halfway to the end, and half of sixty degrees is thirty. Half of 12 is 0.5, and 0.5 × 60° = 30°; a ball 12 pixels off is at 1.0, the full sixty; a ball dead centre is at 0 and goes straight back. That fraction is `offset`, clamped to the range −1 to 1 for a ball that overlaps only the paddle's corner. The speed is the length of the velocity, by Pythagoras: a ball moving 2 across and 1 down is moving $\sqrt{(2^2 + 1^2)} = \sqrt{5} = 2.236$ pixels a tick, and after the return it moves 2.486. The new velocity is that speed pointed along the angle: across, the speed times the cosine, and down, the speed times the sine, so at thirty degrees and 2.25 pixels a tick the ball moves 1.949 across and 1.125 down every tick, and `dir` flips the across part to send it away from the paddle.

`offset` how far the ball's centre is from the paddle's centre, as a fraction of half the paddle: −1 at the top, 0 in the middle, 1 at the bottom

`angle` offset × 60°, in radians for the library: degrees × π / 180

`speed` how far the ball moves a tick, whatever its direction: $\sqrt{(VX^2 + VY^2)}$, which math.Hypot computes

`cos, sin` the across and down parts of a unit step along an angle: cos 30° = 0.866, sin 30° = 0.5

`dir` +1 for a return off the left paddle, which sends the ball right; −1 off the right paddle

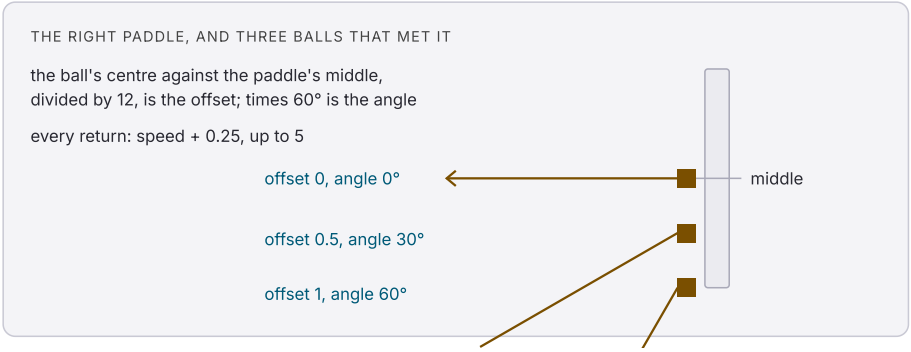


Figure 7.1 — three balls against the right paddle's face, and the direction each leaves in: straight from the middle, thirty degrees from halfway down, sixty from the end.

⚠ WORKED FAILURE — NO CAP ON THE SPEED, AND A BALL THAT STEPS THROUGH THE PADDLE

Stage 3's `Return` adds a quarter of a pixel a tick on every return and never stops adding. Change one line in `newMatch` so that the ball is launched straight along the middle row, `m.Ball.VX, m.Ball.VY = 2, 0`, and touch nothing: the ball meets the middle of each centred paddle, comes straight back, and speeds up. On the 32nd return it reaches ten pixels a tick, and thirty ticks later, on tick 1911, this is the picture:

The stage 3 program with the straight launch, on tick 1911: the ball is past the right paddle, having gone from column 302 to column 312 in one tick.

The ball's left edge was at column 302 on tick 1910 and at 312 on tick 1911, and the paddle occupies columns 308 to 311. For the overlap test to fire, the ball's left edge has to be past 304 and short of 312 at the end of some tick, a window eight pixels wide, and a ball moving ten pixels a tick can step over an eight-pixel window entirely, which it did. The paddle was there; the ball was

never inside it at the end of a tick, and the test asks only about the end of a tick. A point follows on tick 1912. The fix this game chooses is the one that keeps the test: cap the speed below the window. The window is the paddle's width plus the ball's, eight pixels, and a ball that never moves more than five a tick cannot cross it in one step. A game whose ball had to outrun its paddles' width would need a different test, one that asks about the path between two ticks and not the position at the end of one.

▣ BUILD · STAGE 4 — FIVE PIXELS A TICK, AND NO MORE, EXTEND CMD/PONG/MATCH.GO

```
// cmd/pong/match.go — extend
// A paddle returns the ball at an angle set by where the ball
struck it, up
// to sixty degrees from straight, and a quarter of a pixel a
tick faster
// than it arrived, up to five pixels a tick.
const (
        maxAngle = 60.0 // degrees, at the paddle's ends
        speedUp  = 0.25 // pixels a tick added on every return
        maxSpeed = 5.0  // pixels a tick, the cap
)

// Return sends the ball away from paddle p: dir is +1 to send
it right,
// off the left paddle, and -1 to send it left. The angle comes
from how
// far the ball's centre is from the paddle's centre, the speed
grows by
// speedUp up to maxSpeed, and the ball is placed against the
paddle's face.
func (b *Ball) Return(p Paddle, dir float64) {
        offset := ((b.Y + ballSize/2) - (p.Y + paddleH/2)) /
(paddleH / 2)
        offset = clamp(offset, -1, 1)
        angle := offset * maxAngle * math.Pi / 180
        speed := math.Min(math.Hypot(b.VX, b.VY)+speedUp,
maxSpeed)
        b.VX = dir * speed * math.Cos(angle)
        b.VY = speed * math.Sin(angle)
        if dir > 0 {
                b.X = p.X + paddleW
        } else {
                b.X = p.X - ballSize
```

```
        }
}


go vet ./...
go run ./cmd/pong
```

With the cap, the same straight launch rallies for as long as the window is open: the ball reaches five pixels a tick on the twelfth return and stays there, and after six thousand ticks, a hundred seconds, it has been returned 95 times and is still in play. Five is a design choice, and the reasons are the paddle's width, the ball's, and the arithmetic above; a paddle six pixels wide could afford a cap of nine. Put the launch back to `2, 1`.

5. Serving and scoring

A match is in one of three states. In *Serve* the ball rests in the middle until Space is held, when it is served toward the side that conceded the last point, at two pixels a tick and an angle drawn at random from within thirty degrees of straight. In *Play* the ball flies and a point can be scored, by the ball leaving through a side; the point puts the match back in Serve, or, when it is a side's seventh, in *Over*, where R starts another match. Four reasons to change state, one method that does it.

▣ BUILD · STAGE 5 — THREE STATES, EXTEND BOTH FILES

```
// cmd/pong/match.go — extend
import (
        "math"
        "math/rand/v2"
)

// Match is the state of one game, advanced one tick at a time
by Step.
type Match struct {
        Left  Paddle // W and S
        Right Paddle // Up and Down
        Ball  Ball
        State State
        Score [2]int  // points for the left side and the right
side
```

```
        Dir   float64 // which way the next serve goes: +1
right, -1 left

        rng *rand.Rand // the one source of random numbers: the
serve's angle
}

// newMatch returns a match at tick zero: both paddles centred,
the ball at
// rest in the middle of the court, the first serve toward the
right, and
// the serve's angles drawn from a generator seeded with seed.
func newMatch(seed uint64) *Match {
        return &Match{
                Left: newPaddle(leftX), Right:
newPaddle(rightX), Ball: newBall(),
                Dir: 1, rng: rand.New(rand.NewPCG(seed, 0)),
        }
}

// Step advances the match by one tick with the keys held during
it, and
// reports what happened that a player would hear.
func (m *Match) Step(k keys) event {
        m.movePaddles(k)
        switch m.State {
        case Serve:
                if k.space {
                        m.serve()
                }
        case Play:
                if m.fly() {
                        return hit
                }
                if side := m.Ball.Out(); side ≠ 0 {
                        m.point(side)
                        return point
                }
        case Over:
                if k.r {
                        m.restart()
                }
        }
        return nothing
}

// Out reports which side the ball has left the court through:
+1 past the
// right edge, -1 past the left, 0 while any of it is on the
court.
func (b Ball) Out() float64 {
        switch {
        case b.X > courtW:
                return 1
```

```
        case b.X+ballSize < 0:
                return -1
        }
        return 0
}

// A match is in one of three states, and moves between them for
exactly
// four reasons: Space ends Serve, a point ends Play, the
seventh point
// ends the match, and R starts another.
type State uint8

const (
        Serve State = iota // the ball waits at the centre until
Space is held
        Play               // the ball is in flight and a point
can be scored
        Over               // a side has seven points; R starts
a new match
)

// String names the state.
func (s State) String() string {
        switch s {
        case Serve:
                return "serve"
        case Play:
                return "play"
        }
        return "over"
}

// The serve and the match.
const (
        serveSpeed   = 2.0  // pixels a tick
        serveAngle   = 30.0 // degrees: a serve is drawn from
[-30, 30]
        winningScore = 7
)

// event is what a tick did that a player would hear: nothing, a
paddle
// hit, or a point.
type event uint8

const (
        nothing event = iota
        hit
        point
)

// serve puts the ball in flight from the centre toward the side
that
```

```
// conceded last, at serveSpeed and an angle drawn from the
generator.
func (m *Match) serve() {
        angle := (m.rng.Float64()*2 - 1) * serveAngle * math.Pi
/ 180
        m.Ball = newBall()
        m.Ball.VX = m.Dir * serveSpeed * math.Cos(angle)
        m.Ball.VY = serveSpeed * math.Sin(angle)
        m.State = Play
}

// point scores for the side the ball did not leave through,
puts the ball
// back at the centre, and ends the match on the seventh point.
func (m *Match) point(side float64) {
        if side > 0 {
                m.Score[0]++ // out on the right: the left
scores
        } else {
                m.Score[1]++
        }
        m.Dir = side // the next serve goes toward the side that
conceded
        m.Ball = newBall()
        m.State = Serve
        if m.Score[0] == winningScore || m.Score[1] ==
winningScore {
                m.State = Over
        }
}

// restart begins a new match on the same court: scores to zero,
paddles
// centred, the first serve toward the right. The generator
carries on.
func (m *Match) restart() {
        m.Score = [2]int{}
        m.Left, m.Right, m.Ball = newPaddle(leftX),
newPaddle(rightX), newBall()
        m.Dir = 1
        m.State = Serve
}


// cmd/pong/main.go — extend
import (
        "flag"
        "image"
        "image/color"
        "image/png"
        "log"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
```

```
        "github.com/hajimehoshi/ebiten/v2/vector"
)

// newGame builds the match from the flags and loads what the
window draws.
func newGame() (*Game, error) {
        sp, err := loadSprites("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        return &Game{match: newMatch(*seedFlag), court:
newCourt(), sp: sp}, nil
}

func main() {
        flag.Parse()
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Pong")
        ebiten.SetTPS(60)
        g, err := newGame()
        if err ≠ nil {
                log.Fatal(err)
        }
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}

var seedFlag = flag.Uint64("seed", 1, "the seed the serves are
drawn from")


go vet ./...
go run ./cmd/pong
go run ./cmd/pong -seed 7
```

The window opens with the ball at rest in the middle, and nothing happens until Space. Then the ball leaves toward the right at some angle, and if the right player misses it the ball goes out, the ball comes back to the middle, and the next Space sends it toward the right again, because the right conceded. Seven points to a side and Space does nothing; R starts over. The score is stored in `match.go`, where `Step` is now a `switch` on the state, and each state reads only the key that matters to it: Space in Serve, R in Over, and the ball's flight and exit in Play.

The serve's angle is the one random thing in the game, and it is drawn from a generator the match owns, seeded from a number on the command line, so that `-seed 7` gives the same sequence

of serves every time it is run and two runs with the same seed and the same keys are the same match. The default seed is 1. `rand.Float64()` gives a number from 0 up to but not including 1, and `×2 - 1` moves it to −1 up to 1, so the angle lies from −30 degrees up to 30. `Step` now returns an `event`, a hit or a point or nothing. The `String` method on `State` lets a state be printed as a word.

`main.go` changes in two places: the seed is a flag, parsed before anything else, and `newGame` hands it to the match. A flag declared at package level is registered before `main` runs, wherever in the file it sits.

6. Drawing the score and message band

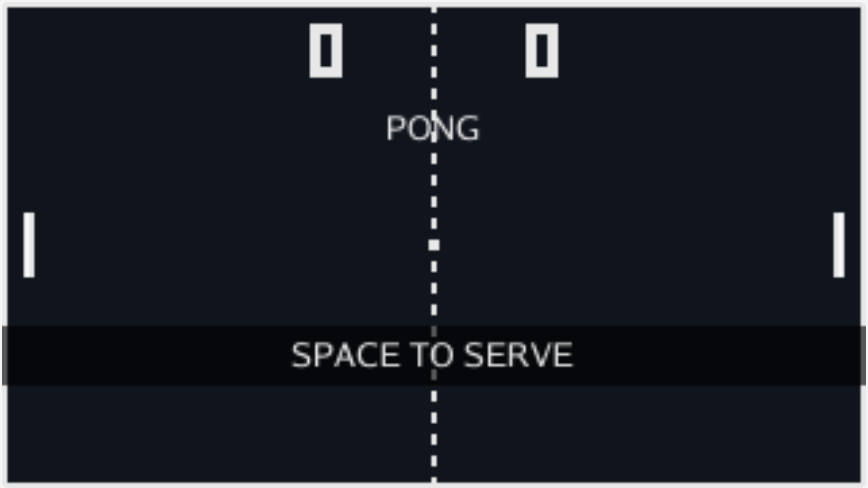
▣ BUILD · STAGE 6 — THE SCORE AND THE BAND, A TEXT FILE AND AN EXTENSION OF MAIN.GO

```
// cmd/pong/text.go — create
package main

import (
        "bytes"
        "log"
        "math"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/text/v2"
        "golang.org/x/image/font/gofont/goregular"
)

// face is the TrueType face the game's words are set in: Go
Regular,
// carried inside the program.
type face struct {
        f *text.GoTextFace
}

// newFace parses the font once and returns it at the size asked
for.
func newFace(size float64) *face {
        src, err :=
text.NewGoTextFaceSource(bytes.NewReader(goregular.TTF))
```

```
        if err ≠ nil {
                log.Fatal(err)
        }
        return &face{f: &text.GoTextFace{Source: src, Size:
size}}
}

// drawCentred draws s in the line colour with its middle on
column x and
// its top on row y.
func (fc *face) drawCentred(dst *ebiten.Image, s string, x, y
float64) {
        w, _ := text.Measure(s, fc.f, 0)
        op := &text.DrawOptions{}
        op.GeoM.Translate(math.Floor(x-w/2), y)
        op.ColorScale.ScaleWithColor(lineColor)
        text.Draw(dst, s, fc.f, op)
}


// cmd/pong/main.go — extend
import (
        "flag"
        "fmt"
        "image"
        "image/color"
        "image/png"
        "log"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"

        "gez/internal/digits"
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
        bandColor  = color.RGBA{R: 0, G: 0, B: 0, A: 160}
)

// Game is Pong's window: the match, and everything drawn.
type Game struct {
        match  *Match
        court  *ebiten.Image
        sp     *sprites
        spin   int // ticks the ball has been moving
        font   *digits.Font
        face   *face
        served bool // has the first serve happened? The title
shows until it has.
}
```

```
// newGame builds the match from the flags and loads what the
window draws.
func newGame() (*Game, error) {
        sp, err := loadSprites("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        return &Game{
                match: newMatch(*seedFlag), court: newCourt(),
sp: sp,
                font: font, face: newFace(12),
        }, nil
}

// step advances the window's game one tick: the match, and the
ball's spin.
func (g *Game) step(k keys) {
        g.match.Step(k)
        if b := g.match.Ball; b.VX ≠ 0 || b.VY ≠ 0 {
                g.spin++
        }
        if g.match.State == Play {
                g.served = true
        }
}

func (g *Game) Draw(screen *ebiten.Image) {
        g.drawMatch(screen)
        g.drawHUD(screen)
}

// The score is drawn with the sheet's digits at four times, and
the band
// is a dark strip behind the line that says what to do.
const (
        digitScale = 4
        scoreY     = 8
        scoreX     = 120 // the left score's middle; the right's
is mirrored
        bandY      = 120
        bandH      = 22
        lineY      = 124
        titleY     = 40
)

// drawHUD draws the score, and, while the match waits on a key,
a band
// with the line that says which key.
func (g *Game) drawHUD(screen *ebiten.Image) {
        m := g.match
```

```
        g.font.DrawCentred(screen, m.Score[0], scoreX, scoreY,
digitScale)
        g.font.DrawCentred(screen, m.Score[1], courtW-scoreX,
scoreY, digitScale)
        if m.State == Play {
                return
        }
        vector.FillRect(screen, 0, bandY, courtW, bandH,
bandColor, false)
        switch m.State {
        case Serve:
                if !g.served {
                        g.face.drawCentred(screen, "PONG",
courtW/2, titleY)
                }
                g.face.drawCentred(screen, "SPACE TO SERVE",
courtW/2, lineY)
        case Over:
                side := "LEFT"
                if m.Score[1] > m.Score[0] {
                        side = "RIGHT"
                }
                g.face.drawCentred(screen, fmt.Sprintf("%s WINS
-  R FOR ANOTHER", side), courtW/2, lineY)
        }
}


go vet ./...
go run ./cmd/pong
```



The window after stage 6, before the first serve: the score, the title, and the band.

The tick after the first point of a match in which nobody moved: the serve went right, the centred paddle was not on its row, and the left has a point.

The score is the digit font from chapter 5 at four times, each side's number centred on the middle of its half, and the band is chapter 5's shade cut down to a strip, drawn only while the match waits on a key, with the line that says which key. The title shows until the first serve and then never again; a `served` flag in the window, set the first time the match is seen in Play, is enough to know that. Both are drawn after the match, which puts the band over the ball, and the band is drawn before its words, which is chapter 5's lesson kept.

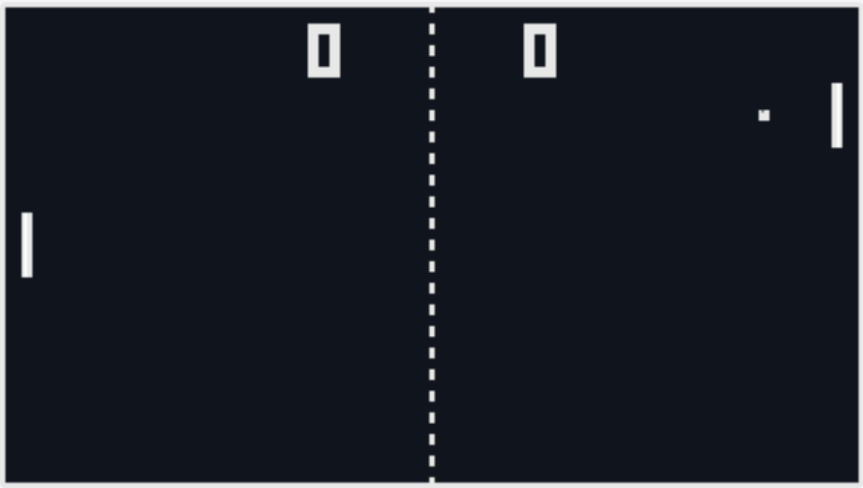
`text.go` is chapter 5's two functions folded into a type, so that `g.face.drawCentred` reads as one thing; Snake and Breakout each get a copy of this file with the same twenty lines. The line colour it draws in is `main.go`'s, because the two files are one package.

7. Adding solo mode

▣ BUILD · STAGE 7 — SOLO MODE, EXTEND BOTH FILES

```
// cmd/pong/match.go — extend
// Match is the state of one game, advanced one tick at a time
by Step.
type Match struct {
        Left  Paddle // W and S
        Right Paddle // Up and Down, or the program in solo mode
        Ball  Ball
        State State
        Score [2]int  // points for the left side and the right
side
        Dir   float64 // which way the next serve goes: +1
right, -1 left
        Solo  bool    // the right paddle follows the ball

        rng *rand.Rand // the one source of random numbers: the
serve's angle
}

// newMatch returns a match at tick zero: both paddles centred,
the ball at
// rest in the middle of the court, the first serve toward the
right, and
// the serve's angles drawn from a generator seeded with seed.
The right
// paddle is played by the program when solo is set.
func newMatch(seed uint64, solo bool) *Match {
        return &Match{
                Left: newPaddle(leftX), Right:
newPaddle(rightX), Ball: newBall(),
                Dir: 1, Solo: solo, rng:
rand.New(rand.NewPCG(seed, 0)),
        }
}

// movePaddles moves the left paddle by W and S and the right by
Up and
// Down, or, in solo mode, after the ball.
func (m *Match) movePaddles(k keys) {
        m.Left.Move(k.w, k.s)
        if m.Solo {
                m.Right.Follow(m.Ball)
        } else {
                m.Right.Move(k.up, k.down)
        }
}

// Follow moves the paddle toward the ball's row at the paddle's
speed and
// never faster: the program's opponent in solo mode.
func (p *Paddle) Follow(b Ball) {
        target := b.Y + ballSize/2 - paddleH/2
        dy := clamp(target-p.Y, -paddleSpeed, paddleSpeed)
```

```
        p.Y = clamp(p.Y+dy, border, courtH-border-paddleH)
}


// cmd/pong/main.go — extend
// newGame builds the match from the flags and loads what the
window draws.
func newGame() (*Game, error) {
        sp, err := loadSprites("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        return &Game{
                match: newMatch(*seedFlag, *soloFlag), court:
newCourt(), sp: sp,
                font: font, face: newFace(12),
        }, nil
}

var (
        seedFlag = flag.Uint64("seed", 1, "the seed the serves
are drawn from")
        soloFlag = flag.Bool("solo", false, "play against the
program")
)


go vet ./...
go run ./cmd/pong -solo
```



Ten seconds into a solo match: the right paddle has followed the ball up to its row while the left player, who served, waits below.

With `-solo` the right paddle plays itself: on every tick it moves toward the row that would put its middle on the ball's middle, two pixels at most, and stops when it is there. That is a paddle that never misses a slow ball and cannot keep up with a fast one at a steep angle, because the ball's vertical speed at sixty degrees and five pixels a tick is 4.3 and the paddle's is 2. The way to beat it is the way to beat a person who only follows: meet the ball near the end of your paddle, so that it leaves steeply, and do it again. The follower reads the ball's position from the match, in the match, so it is a rule and not a piece of the window; a second flag hands the choice to `newMatch`, and the two flags now share one `var` block.

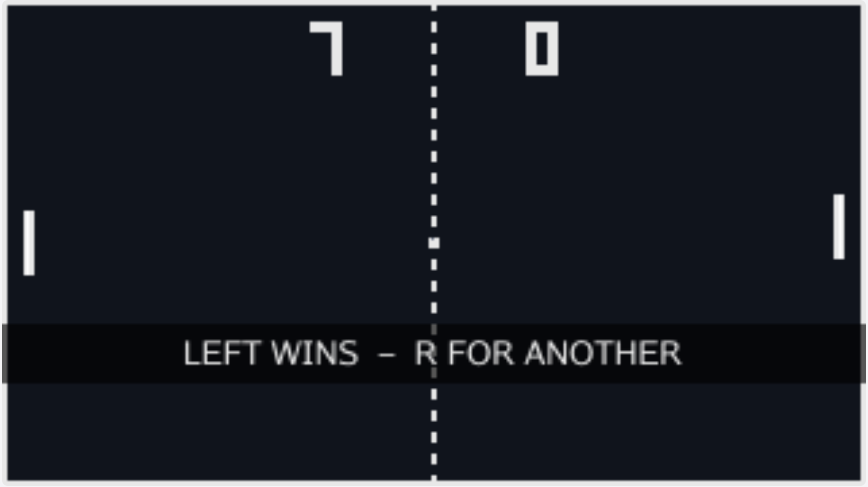
8. Playing hit and point sounds

▣ BUILD · STAGE 8 — A HIT AND A POINT, HEARD, EXTEND CMD/PONG/MAIN.GO

```
// cmd/pong/main.go — extend
import (
        "flag"
        "fmt"
        "image"
        "image/color"
        "image/png"
        "log"
        "os"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"

        "gez/internal/digits"
        "gez/internal/sound"
)

// Game is Pong's window: the match, and everything drawn or
heard.
type Game struct {
```

```
        match      *Match
        court      *ebiten.Image
        sp         *sprites
        spin       int // ticks the ball has been moving
        font       *digits.Font
        face       *face
        served     bool // has the first serve happened? The
title shows until it has.
        hit, point *sound.Sound
}

// newGame builds the match from the flags and loads what the
window
// draws and plays.
func newGame() (*Game, error) {
        sp, err := loadSprites("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        hit, err := sound.Load("assets/hit.wav")
        if err ≠ nil {
                return nil, err
        }
        point, err := sound.Load("assets/point.wav")
        if err ≠ nil {
                return nil, err
        }
        return &Game{
                match: newMatch(*seedFlag, *soloFlag), court:
newCourt(), sp: sp,
                font: font, face: newFace(12), hit: hit, point:
point,
        }, nil
}

// step advances the window's game one tick: the match, the
ball's spin,
// and a sound for what the match reports.
func (g *Game) step(k keys) {
        switch g.match.Step(k) {
        case hit:
                g.hit.Play()
        case point:
                g.point.Play()
        }
        if b := g.match.Ball; b.VX ≠ 0 || b.VY ≠ 0 {
                g.spin++
        }
        if g.match.State == Play {
                g.served = true
```

```
        }
}

go vet ./...
go run ./cmd/pong -solo
```



The end of a solo match, 7,456 ticks in: the left player beat the following paddle seven points to none, and the band waits for R.

A return beeps and a point sounds the lower tone. The match reports both as an `event`. The window plays the sound in `step`, and the match rules never import or call the sound package.

Pong now has two files for rules and window code, plus one file for text drawing. The picture above is a solo match that took 7,456 ticks. The follower lost every point to a player who aimed for the ends of the paddle.

9. Keeping rules separate from drawing

Every fact about a match is in one value. One method changes it, once per tick, from the keys for that tick. That made each stage a change to the rules first, with the window drawing the result.

This also makes the match reproducible. Two matches with the same seed and the same keys on the same ticks are the same match, to the pixel, because the rules do not read a wall clock, a mouse or a draw count. The spin counter lives outside the rules because it changes only the picture.

10. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Keep the rules of a game in a value with a `Step` method and the window in another, and say what each may touch.
- Reflect a ball off a border without losing any of its step, and say what stopping it at the border would look like at five pixels a tick.
- Turn where a ball struck a paddle into an angle and a velocity, by hand, for a ball six pixels off the middle at 2.25 pixels a tick.
- Explain, in columns, how a ball at ten pixels a tick steps over a four-pixel paddle, and why a cap of five cannot.
- Write a three-state match with a seeded serve, and say what the seed buys.
- Draw a score and a state band over the match, in the right order, and play a sound on what `Step` reports.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — to eleven, by two. Make the match end at eleven points, but only when the leader is two points clear.**

`winningScore = 11`, and the test in `point` becomes `lead ⩾ winningScore && lead-trail ⩾ 2` with `lead` and `trail` the larger and smaller score. At 10–10 the match goes on to 12–10. The band needs no change; it reads the scores when the state is Over.

▼ **Exercise 2 — a wider paddle, a higher cap. Make the paddles six pixels wide, drawn with `vector.FillRect` instead of the sprites, raise the cap to nine, and run the straight launch of the worked failure. Then take the cap off and find the speed the ball goes through at.**

`paddleW = 6`, two `FillRect` calls in `drawMatch` with the paddles' rectangles, and `maxSpeed = 9`. The overlap window is now ten pixels wide, and a ball at nine a tick cannot step over it, so the straight rally holds at nine for as long as you watch. With the cap removed the rally speeds up past ten and the ball goes through on the first return whose step happens to land either side of the window; count the returns to find the speed, and compare it with the chapter's 32.

▼ **Exercise 3 — the machine against itself. Make the left paddle follow the ball too when `-solo` is given, and watch a match nobody plays.**

In `movePaddles`, `m.Left.Follow(m.Ball)` under the same `if m.Solo`. Both paddles arrive centred on the ball, every return is straight, and the rally runs at five pixels a tick until you close the window: two followers never miss each other's returns, which is why the game gives the follower a human to lose to.

VOLUME 1 · CHAPTER 8

Snake

A grid, a timer, a body that is a queue, and a high score that outlives the window

1. Building Snake

This chapter builds Snake on a grid. The snake moves one cell every eight ticks, remembers one turn between moves, grows when it eats food and saves the best score.

The game counts in cells, not pixels. A body move writes one new head cell and drops the tail unless the snake is growing. The ring buffer makes that move change one slot instead of copying the whole body.

2. Moving on a grid timer

▣ BUILD · STAGE 1 — THE GRID, THE TIMER, AND A HEAD

```
// cmd/snake/grid.go — create
package main

// The grid is forty cells wide and twenty tall, of eight-pixel
cells, and
// fills the picture below a twenty-pixel strip for the score:
40 × 8 is
// 320 across and 20 × 8 is 160 down, and cell (0, 0) sits at
pixel (0, 20).
const (
        cols     = 40
        rows     = 20
        cellSize = 8
        top      = 20 // the strip above the grid, in pixels
)

// Cell is one square of the grid: column X, row Y.
```

```
type Cell struct {
        X, Y int
}

// Inside reports whether the cell is on the grid.
func (c Cell) Inside() bool {
        return c.X ⩾ 0 && c.X < cols && c.Y ⩾ 0 && c.Y < rows
}

// Pixel is the top-left corner of the cell on the screen.
func (c Cell) Pixel() (x, y int) {
        return c.X * cellSize, top + c.Y*cellSize
}

// Dir is a direction across the grid: one of the four below, or
the zero
// Dir, which is no direction at all.
type Dir struct {
        DX, DY int
}

var (
        up    = Dir{0, -1}
        down  = Dir{0, 1}
        left  = Dir{-1, 0}
        right = Dir{1, 0}
)

// Step is the cell one step from c in direction d.
func (c Cell) Step(d Dir) Cell {
        return Cell{c.X + d.DX, c.Y + d.DY}
}


// cmd/snake/game.go — create
package main

// The snake moves one cell every eight ticks: at sixty ticks a
second,
// seven and a half cells a second.
const startInterval = 8

// Snake is the state of one game: the head, its heading and its
timer,
// advanced one tick at a time by Step.
type Snake struct {
        Head     Cell // the cell the head is on
        Heading  Dir  // the way the next move goes
        Interval int  // ticks between moves
        Wait     int  // ticks until the next move
}

// newSnake returns a game at tick zero: the head in the middle
of the
```

```
// grid, heading right, the first move eight ticks away.
func newSnake() *Snake {
        return &Snake{Head: Cell{cols / 2, rows / 2}, Heading:
right, Interval: startInterval, Wait: startInterval}
}

// Step advances the game by one tick: the timer counts down,
and the head
// moves when it fires.
func (g *Snake) Step(k keys) {
        g.Wait--
        if g.Wait > 0 {
                return
        }
        g.Wait = g.Interval
        g.move()
}

// move steps the head one cell the way it is heading.
func (g *Snake) move() {
        g.Head = g.Head.Step(g.Heading)
}


// cmd/snake/main.go — create
package main

import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"
)

const (
        screenW = 320
        screenH = 180
)

var (
        courtColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
        dimColor   = color.RGBA{R: 60, G: 66, B: 80, A: 255}
        bodyColor  = color.RGBA{R: 120, G: 200, B: 120, A: 255}
        foodColor  = color.RGBA{R: 232, G: 120, B: 80, A: 255}
        bandColor  = color.RGBA{R: 0, G: 0, B: 0, A: 160}
)

// keys is what the player is doing this tick: the four arrows
and R,
// held or not.
type keys struct {
        up, down, left, right, r bool
```

```
}

// readKeys asks Ebitengine about the five keys, once a tick.
func readKeys() keys {
        return keys{
                up:    ebiten.IsKeyPressed(ebiten.KeyArrowUp),
                down:  ebiten.IsKeyPressed(ebiten.KeyArrowDown),
                left:  ebiten.IsKeyPressed(ebiten.KeyArrowLeft),
                right:
ebiten.IsKeyPressed(ebiten.KeyArrowRight),
                r:     ebiten.IsKeyPressed(ebiten.KeyR),
        }
}

// Game is Snake's window: the snake, and everything drawn.
type Game struct {
        snake *Snake
}

// newGame builds the snake.
func newGame() *Game {
        return &Game{snake: newSnake()}
}

// step advances the game one tick.
func (g *Game) step(k keys) {
        g.snake.Step(k)
}

func (g *Game) Update() error {
        g.step(readKeys())
        return nil
}

// drawCell fills a cell's square, one pixel smaller than the
cell, at the
// pixel the cell works out to.
func drawCell(dst *ebiten.Image, c Cell, clr color.RGBA) {
        x, y := c.Pixel()
        vector.FillRect(dst, float32(x), float32(y), cellSize-1,
cellSize-1, clr, false)
}

// drawGrid paints the background and the line under the score
strip.
func drawGrid(screen *ebiten.Image) {
        screen.Fill(courtColor)
        vector.FillRect(screen, 0, top-1, screenW, 1, dimColor,
false)
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawGrid(screen)
        drawCell(screen, g.snake.Head, lineColor)
```

```
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Snake")
        ebiten.SetTPS(60)
        if err := ebiten.RunGame(newGame()); err ≠ nil {
                log.Fatal(err)
        }
}


go vet ./...
go run ./cmd/snake
```

The window forty ticks after stage 1 starts: five moves have carried the head from the middle of the grid, cell (20, 10), to cell (25, 10).

A white square steps to the right, one cell every eight ticks. The grid code uses `Cell` for positions and `Dir` for movement. Stepping a cell means adding the direction to it.

The four directions are values because Go constants cannot be structs. The zero `Dir` means no direction. Nothing becomes a pixel until `Pixel` converts a cell to a screen position.

The timer is two integers. `Wait` counts down every tick and, on the tick it reaches zero, is reset from `Interval` and the head moves. Eight ticks between moves is the starting tempo. The rest of the program reads the interval instead of repeating the number.

3. Queuing a turn

An arrow key is held for a few ticks, and a move happens on one tick in eight. If the game only read the arrow on the tick of the move, most taps would be missed; if it changed the heading the moment the arrow was seen, two quick taps between moves would leave only the second, which is the same thing. So an arrow queues a turn, the queue holds one turn, the last one wins, and the move takes it.

▣ BUILD · STAGE 2 — THE QUEUED TURN, EXTEND CMD/SNAKE/GAME.GO

```
// cmd/snake/game.go — extend
// Snake is the state of one game: the head, its heading, a
queued turn and
// its timer, advanced one tick at a time by Step.
type Snake struct {
        Head     Cell // the cell the head is on
        Heading  Dir  // the way the next move goes, unless a
turn is queued
        Queued   Dir  // the turn the next move takes; the zero
Dir is none
        Interval int  // ticks between moves
        Wait     int  // ticks until the next move
}

// Step advances the game by one tick with the keys held during
it. An
// arrow held on any tick queues a turn; the turn is taken on
the tick the
// timer fires, and only the last one queued counts.
func (g *Snake) Step(k keys) {
        if d, ok := turn(k); ok {
                g.Queued = d
        }
        g.Wait--
        if g.Wait > 0 {
```

```
                return
        }
        g.Wait = g.Interval
        g.move()
}

// move takes the queued turn, if there is one, and steps the
head one cell.
func (g *Snake) move() {
        if g.Queued ≠ (Dir{}) {
                g.Heading = g.Queued
                g.Queued = Dir{}
        }
        g.Head = g.Head.Step(g.Heading)
}

// turn reads a direction off the arrow keys: the first held, in
the order
// up, down, left, right, and none when no arrow is held.
func turn(k keys) (Dir, bool) {
        switch {
        case k.up:
                return up, true
        case k.down:
                return down, true
        case k.left:
                return left, true
        case k.right:
                return right, true
        }
        return Dir{}, false
}


go vet ./...
go run ./cmd/snake
```

The window 56 ticks after stage 2 starts, with Up tapped on tick 20 and Right on tick 38: the head went right three cells, up two, and right three more, to cell (25, 8).

Tap Up and the snake turns up on its next move, whenever that is; tap Up and then Right before the move and it turns right, the first tap forgotten. In the picture, Up on tick 20 was taken by the move on tick 24 and Right on tick 38 by the move on tick 40, and a tap held for a single tick is enough, because `Step` reads the arrows on every tick and the queue keeps what it read. Two things about the code. `turn` returns a second value saying whether any arrow was held, so that a tick with no arrow leaves the queue alone instead of emptying it. And a struct can be compared with `≠` when all its fields can, so `g.Queued ≠ (Dir{})` asks whether a turn is queued; the parentheses keep Go from reading the braces as the start of a block.

4. Storing the body in a ring

The body is the cells the snake occupies, head first. Every move adds the new head to the front and, unless the snake is growing, drops the tail from the back, and the cells in the middle stay where they are. A slice with the head at index 0 would have to shift every cell on every

move. A ring buffer does not: it is a slice with a slot for every cell of the grid, 800 of them, an index saying which slot holds the head, and a count of how many slots are in use, running backwards from the head. A move writes the new head into the slot after the current one and moves the index; the count stays the same, which means the slot the tail was in stops being counted, and is written over eight hundred moves later.

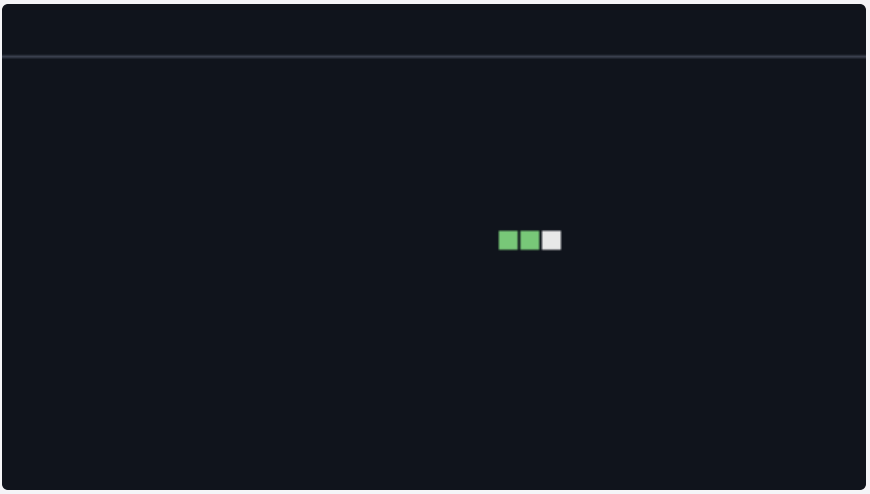
▣ BUILD · STAGE 3 — THE BODY AS A RING BUFFER, A NEW FILE AND TWO EXTENSIONS

```
// cmd/snake/body.go — create
package main

// Body is the snake's cells in a ring buffer: a slice with a
slot for every
// cell of the grid, the index of the slot the head is in, and
how many
// slots are in use. The cells run backwards from the head, so
the tail is
// n-1 slots behind it, wrapping round the end of the slice. A
move writes
// the new head into the slot after the old one; unless the
snake grew, the
// count stays the same, so the tail's slot simply stops
counting. Nothing
// is copied and nothing is allocated.
type Body struct {
        slots []Cell
        head  int // the slot the head is in
        n     int // slots in use, counting back from the head
}

// newBody returns a body of the cells given, head first, with a
slot for
// every cell of the grid.
func newBody(cells ...Cell) Body {
        b := Body{slots: make([]Cell, cols*rows), head:
len(cells) - 1, n: len(cells)}
        for i, c := range cells {
                b.slots[b.head-i] = c
        }
        return b
}

// Len is the number of cells in the body.
func (b *Body) Len() int {
        return b.n
```

```
}

// At is the i-th cell from the head: At(0) is the head,
At(Len()-1) the
// tail. Its slot is head-i, wrapped into the slice.
func (b *Body) At(i int) Cell {
        return b.slots[(b.head-i+len(b.slots))%len(b.slots)]
}

// Head is the first cell.
func (b *Body) Head() Cell {
        return b.At(0)
}

// Has reports whether c is one of the body's cells.
func (b *Body) Has(c Cell) bool {
        for i := 0; i < b.n; i++ {
                if b.At(i) == c {
                        return true
                }
        }
        return false
}

// Advance writes the new head into the slot after the current
one. When
// the snake grows the count goes up by one and the tail stays;
otherwise
// the count stays and the tail's slot is no longer counted.
func (b *Body) Advance(head Cell, grow bool) {
        b.head = (b.head + 1) % len(b.slots)
        b.slots[b.head] = head
        if grow {
                b.n++
        }
}


// cmd/snake/game.go — extend
// The snake moves one cell every eight ticks: at sixty ticks a
second,
// seven and a half cells a second. It starts three cells long.
const (
        startInterval = 8
        startLength   = 3
)

// Snake is the state of one game: the snake, its heading, a
queued turn
// and its timer, advanced one tick at a time by Step.
type Snake struct {
        Body     Body // the snake's cells, head first
        Heading  Dir  // the way the next move goes, unless a
turn is queued
```

```
        Queued   Dir  // the turn the next move takes; the zero
Dir is none
        Interval int  // ticks between moves
        Wait     int  // ticks until the next move
}

// newSnake returns a game at tick zero: three cells in the
middle of the
// grid, heading right, the first move eight ticks away.
func newSnake() *Snake {
        return &Snake{Body: startBody(), Heading: right,
Interval: startInterval, Wait: startInterval}
}

// move takes the queued turn, if there is one, and steps the
head one
// cell; the tail follows.
func (g *Snake) move() {
        if g.Queued ≠ (Dir{}) {
                g.Heading = g.Queued
                g.Queued = Dir{}
        }
        g.Body.Advance(g.Head().Step(g.Heading), false)
}

// startBody is three cells in the middle of the grid, heading
right.
func startBody() Body {
        head := Cell{cols / 2, rows / 2}
        cells := make([]Cell, startLength)
        for i := range cells {
                cells[i] = Cell{head.X - i, head.Y}
        }
        return newBody(cells...)
}

// Head is the cell the snake's head is on.
func (g *Snake) Head() Cell {
        return g.Body.Head()
}


// cmd/snake/main.go — extend
func (g *Game) Draw(screen *ebiten.Image) {
        drawGrid(screen)
        g.drawBody(screen)
}

// drawBody draws the snake, tail first, so that the head is
drawn last.
func (g *Game) drawBody(screen *ebiten.Image) {
        s := g.snake
        for i := s.Body.Len() - 1; i > 0; i-- {
                drawCell(screen, s.Body.At(i), bodyColor)
```

```
        }
        drawCell(screen, s.Head(), lineColor)
}


go vet ./...
go run ./cmd/snake
```



The window 56 ticks after stage 3 starts, with the same two taps as before: three cells, the head white and the body green, the tail two cells behind on the row the last turn began.

The snake is three cells now and turns as a snake does: the head goes round the corner and the body follows it cell by cell, because each cell of the body is where the head was one, two, three moves ago. That is the ring buffer read backwards from the head, and it is also why the tail is drawn first: the head is painted last so that it is on top if the two ever share a cell, which the next stages make a matter of life and death. `Head` is a field no longer but a method, the first cell of the body, and every use of it gains a pair of parentheses. `newBody` lays the starting cells backwards from the head slot so that the tail is behind the head in the ring the same way it will be after a hundred moves.

∑ MATH INTERLUDE — THE SLOT BEHIND THE HEAD

Start with 800 slots, the head in slot 2, and three cells in use: the head is in slot 2, the next cell in slot 1, the tail in slot 0. A move writes the new head into slot 3, and the three cells in use are now slots 3, 2 and 1; slot 0 still holds the old tail, but nothing counts it. After 797 more moves the head is in slot 799, and the next move needs a slot: (799 + 1) mod 800 is 0, the slot the first tail left, which has not been counted for 797 moves. Reading backwards wraps the same way: with the head in slot 0 and three cells in use, the cell one behind the head is in slot (0 − 1 + 800) mod 800 = 799, and the tail in 798. Adding 800 before taking the remainder keeps the number from going negative, because Go's % of a negative number is negative.

`N`the number of slots, cols × rows = 800: one for every cell the snake could ever occupy
`head`the slot the head is in, 0 to N−1
`n`how many slots are in use, counting back from the head: the snake's length
`(head + 1) mod N`the slot the next head goes in; N−1 wraps to 0
`(head - i + N) mod N`the slot of the cell i behind the head; the + N keeps it from going negative

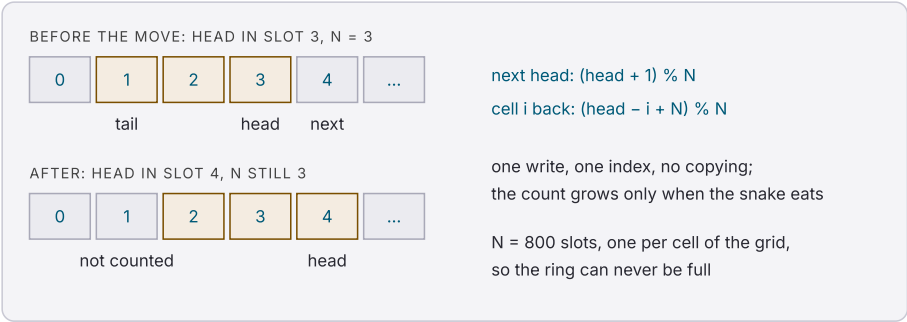


Figure 8.1 — the ring buffer across one move: the head's slot advances, the count stays, and the old tail's slot is left uncounted.

5. Placing food and scoring

▣ BUILD · STAGE 4 — FOOD, GROWTH, THE SCORE IN THE STRIP, AND A SOUND, EXTEND BOTH FILES

```
// cmd/snake/game.go — extend
import "math/rand/v2"

// The snake moves one cell every eight ticks to begin with,
seven and a
// half cells a second, and starts three cells long. Every fifth
food takes
// a tick off the interval, down to three.
const (
        startInterval = 8
        startLength   = 3
        foodsPerSpeed = 5
        minInterval   = 3
)

// Snake is the state of one game: the snake, its timer, the
food and the
// score, advanced one tick at a time by Step.
type Snake struct {
        Body     Body // the snake's cells, head first
        Heading  Dir  // the way the next move goes, unless a
turn is queued
        Queued   Dir  // the turn the next move takes; the zero
Dir is none
        Interval int  // ticks between moves
        Wait     int  // ticks until the next move
        Food     Cell // the one cell with food on it
        Score    int  // foods eaten this game
        Grow     int  // moves on which the tail stays put,
still owed

        rng *rand.Rand // the one source of random numbers:
where the food goes
}

// newSnake returns a game at tick zero: three cells in the
middle of the
// grid, heading right, the first move eight ticks away, and the
food drawn
// from a generator seeded with seed.
func newSnake(seed uint64) *Snake {
        g := &Snake{
                Body: startBody(), Heading: right, Interval:
startInterval, Wait: startInterval,
                rng: rand.New(rand.NewPCG(seed, 0)),
```

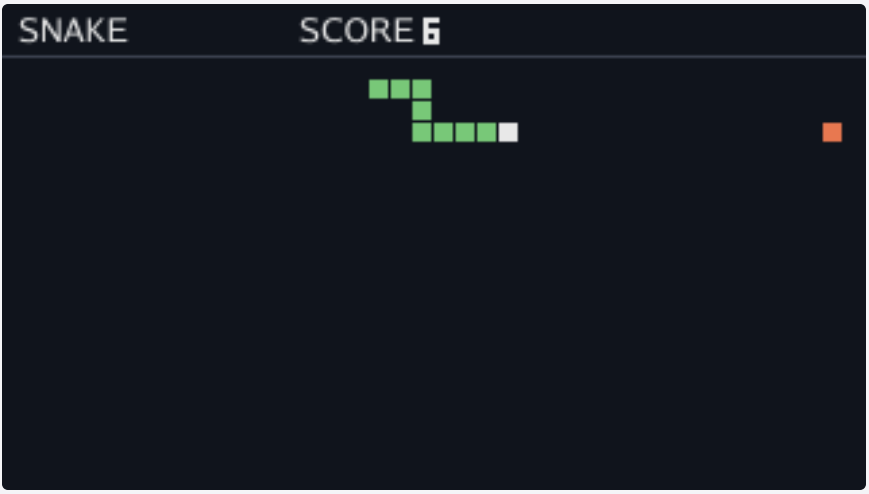
```
        }
        g.place()
        return g
}

// Step advances the game by one tick with the keys held during
it. An
// arrow held on any tick queues a turn; the turn is taken on
the tick the
// timer fires, and only the last one queued counts.
func (g *Snake) Step(k keys) event {
        if d, ok := turn(k); ok {
                g.Queued = d
        }
        g.Wait--
        if g.Wait > 0 {
                return nothing
        }
        g.Wait = g.Interval
        return g.move()
}

// move takes the queued turn, if there is one, and steps the
head one
// cell; a cell with food on it is eaten.
func (g *Snake) move() event {
        if g.Queued ≠ (Dir{}) {
                g.Heading = g.Queued
                g.Queued = Dir{}
        }
        next := g.Head().Step(g.Heading)
        grow := g.Grow > 0
        if grow {
                g.Grow--
        }
        g.Body.Advance(next, grow)
        if next == g.Food {
                g.eat()
                return ate
        }
        return nothing
}

// event is what a tick did that a player would hear: nothing,
or the
// snake ate.
type event uint8

const (
        nothing event = iota
        ate
)

// place puts the food on a cell the body does not hold: a
```

```
column and a
// row are drawn, and drawn again while the cell is taken.
func (g *Snake) place() {
        for {
                c := Cell{g.rng.IntN(cols), g.rng.IntN(rows)}
                if !g.Body.Has(c) {
                        g.Food = c
                        return
                }
        }
}

// eat scores the food under the head, grows the snake on its
next move,
// takes a tick off the interval every fifth food, and places
the next one.
func (g *Snake) eat() {
        g.Score++
        g.Grow++
        if g.Score%foodsPerSpeed == 0 && g.Interval >
minInterval {
                g.Interval--
        }
        g.place()
}


// cmd/snake/text.go — create
package main

import (
        "bytes"
        "log"
        "math"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/text/v2"
        "golang.org/x/image/font/gofont/goregular"
)

// face is the TrueType face the game's words are set in: Go
Regular,
// carried inside the program.
type face struct {
        f *text.GoTextFace
}

// newFace parses the font once and returns it at the size asked
for.
func newFace(size float64) *face {
        src, err :=
text.NewGoTextFaceSource(bytes.NewReader(goregular.TTF))
        if err != nil {
                log.Fatal(err)
```

```
        }
        return &face{f: &text.GoTextFace{Source: src, Size:
size}}
}

// draw draws s in the line colour with its left edge on column
x and its
// top on row y.
func (fc *face) draw(dst *ebiten.Image, s string, x, y float64)
{
        op := &text.DrawOptions{}
        op.GeoM.Translate(x, y)
        op.ColorScale.ScaleWithColor(lineColor)
        text.Draw(dst, s, fc.f, op)
}

// drawCentred draws s with its middle on column x and its top
on row y.
func (fc *face) drawCentred(dst *ebiten.Image, s string, x, y
float64) {
        w, _ := text.Measure(s, fc.f, 0)
        fc.draw(dst, s, math.Floor(x-w/2), y)
}


// cmd/snake/main.go — extend
import (
        "flag"
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"

        "gez/internal/digits"
        "gez/internal/sound"
)

// Game is Snake's window: the snake, and everything drawn or
heard.
type Game struct {
        snake *Snake
        font  *digits.Font
        face  *face
        eat   *sound.Sound
}

// newGame builds the snake from the flags and loads what the
window draws
// and plays.
func newGame() (*Game, error) {
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
```

```
        }
        eat, err := sound.Load("assets/hit.wav")
        if err ≠ nil {
                return nil, err
        }
        return &Game{snake: newSnake(*seedFlag), font: font,
face: newFace(12), eat: eat}, nil
}

// step advances the game one tick and plays what it reports.
func (g *Game) step(k keys) {
        switch g.snake.Step(k) {
        case ate:
                g.eat.Play()
        }
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawGrid(screen)
        g.drawFood(screen)
        g.drawBody(screen)
        g.drawScore(screen)
}

func main() {
        flag.Parse()
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Snake")
        ebiten.SetTPS(60)
        g, err := newGame()
        if err ≠ nil {
                log.Fatal(err)
        }
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}

var seedFlag = flag.Uint64("seed", 1, "the seed the food is
drawn from")

// drawFood draws the one cell with food on it.
func (g *Game) drawFood(screen *ebiten.Image) {
        drawCell(screen, g.snake.Food, foodColor)
}

// drawScore draws the name of the game and the score in the
strip.
func (g *Game) drawScore(screen *ebiten.Image) {
        g.face.draw(screen, "SNAKE", 6, 3)
        g.face.draw(screen, "SCORE", 110, 3)
        g.font.Draw(screen, g.snake.Score, 156, 5, 2)
}
```

```
go vet ./...
go run ./cmd/snake
```



The window ten seconds after stage 4 starts, steered toward each food as it appeared: six eaten, nine cells long, and the seventh food waiting on the right.

An orange cell appears somewhere on the grid; steer the head onto it and it beeps, the score in the strip goes up by one, the snake is one cell longer after its next move, and a new food appears elsewhere. Every fifth food takes a tick off the interval: eight ticks a move to begin with, seven after the fifth food, down to three after the twenty-fifth, which is twenty cells a second and about as fast as a person can steer. The food is drawn as an orange cell before the body, so that a head arriving on it is drawn over it.

Where the food goes is the one random thing in the game, drawn from a generator the snake owns and seeded from `-seed`, so that the same seed gives the same sequence of foods, and `place` draws again while the cell it drew is under the snake. That technique is called rejection sampling: draw from the whole grid, reject what is not allowed, draw again. It is the right tool while the snake covers little of the grid, and it is the wrong tool for a snake

that covers most of it, when almost every draw would be rejected; with 800 cells and a snake that dies long before it fills them, a second draw is rare and a third rarer.

Growth is a debt. Eating adds one to `Grow`, and the next move pays it: `Advance` is told to grow, the count goes up, and the tail stays where it was. The snake lengthens by one cell one move after it eats, which is when the head has moved off the food. `Step` now returns an `event`, as Pong's does, and the window plays the hit sound on `ate`. The strip at the top is chapter 5's two fonts, the words from the face and the score from the sheet's digits at twice their size, and `text.go` is Pong's with one more method, `draw`, for text placed by its left edge.

6. Ending and restarting the game

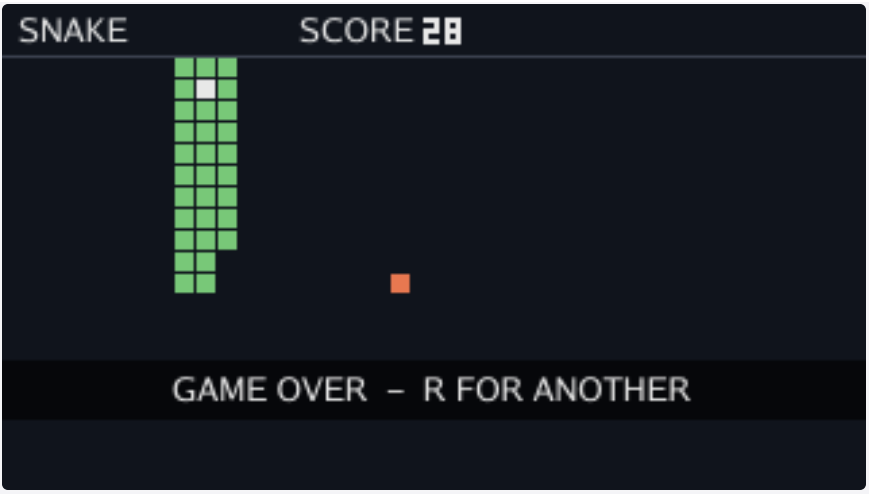
▣ BUILD · STAGE 5 — DEATH, AND ANOTHER GAME ON R, EXTEND BOTH FILES

```
// cmd/snake/game.go — extend
// Snake is the state of one game: the snake, its timer, the
food and the
// score, advanced one tick at a time by Step.
type Snake struct {
        Body     Body // the snake's cells, head first
        Heading  Dir  // the way the next move goes, unless a
turn is queued
        Queued   Dir  // the turn the next move takes; the zero
Dir is none
        Interval int  // ticks between moves
        Wait     int  // ticks until the next move
        Food     Cell // the one cell with food on it
        Score    int  // foods eaten this game
        Grow     int  // moves on which the tail stays put,
still owed
        Dead     bool // the head left the grid or ran into the
body; R restarts

        rng *rand.Rand // the one source of random numbers:
where the food goes
}
```

```
// newSnake returns a game at tick zero: three cells in the
middle of the
// grid, heading right, the first move eight ticks away, and the
food drawn
// from a generator seeded with seed.
func newSnake(seed uint64) *Snake {
        g := &Snake{rng: rand.New(rand.NewPCG(seed, 0))}
        g.restart()
        return g
}

// Step advances the game by one tick with the keys held during
it. An
// arrow held on any tick queues a turn; the turn is taken on
the tick the
// timer fires, and only the last one queued counts. A dead game
waits for R.
func (g *Snake) Step(k keys) event {
        if g.Dead {
                if k.r {
                        g.restart()
                }
                return nothing
        }
        if d, ok := turn(k); ok {
                g.Queued = d
        }
        g.Wait--
        if g.Wait > 0 {
                return nothing
        }
        g.Wait = g.Interval
        return g.move()
}

// move takes the queued turn, if there is one, and steps the
head one
// cell. A cell off the grid or on the snake's own body is
death, decided
// before anything moves; a cell with food on it is eaten.
func (g *Snake) move() event {
        if g.Queued ≠ (Dir{}) {
                g.Heading = g.Queued
                g.Queued = Dir{}
        }
        next := g.Head().Step(g.Heading)
        if !next.Inside() || g.Body.Has(next) {
                g.Dead = true
                return died
        }
        grow := g.Grow > 0
        if grow {
                g.Grow--
        }
```

```
        g.Body.Advance(next, grow)
        if next == g.Food {
                g.eat()
                return ate
        }
        return nothing
}

// event is what a tick did that a player would hear: nothing,
the snake
// ate, or the snake died.
type event uint8

const (
        nothing event = iota
        ate
        died
)

// restart begins a new game on the same grid: the body, the
timer and the
// score go back to the start, and the generator carries on, so
the food
// is not where it was last time.
func (g *Snake) restart() {
        g.Body = startBody()
        g.Heading, g.Queued = right, Dir{}
        g.Interval, g.Wait = startInterval, startInterval
        g.Score, g.Grow = 0, 0
        g.Dead = false
        g.place()
}


// cmd/snake/main.go — extend
// Game is Snake's window: the snake, and everything drawn or
heard.
type Game struct {
        snake    *Snake
        font     *digits.Font
        face     *face
        eat, die *sound.Sound
}

// newGame builds the snake from the flags and loads what the
window draws
// and plays.
func newGame() (*Game, error) {
        font, err := digits.Load("assets/pong-sheet.png")
        if err != nil {
                return nil, err
        }
        eat, err := sound.Load("assets/hit.wav")
        if err != nil {
```

```
                return nil, err
        }
        die, err := sound.Load("assets/point.wav")
        if err ≠ nil {
                return nil, err
        }
        return &Game{snake: newSnake(*seedFlag), font: font,
face: newFace(12), eat: eat, die: die}, nil
}

// step advances the game one tick and plays what it reports.
func (g *Game) step(k keys) {
        switch g.snake.Step(k) {
        case ate:
                g.eat.Play()
        case died:
                g.die.Play()
        }
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawGrid(screen)
        g.drawFood(screen)
        g.drawBody(screen)
        g.drawScore(screen)
        g.drawOver(screen)
}

// drawOver draws a band across the grid while the game waits
for R.
func (g *Game) drawOver(screen *ebiten.Image) {
        if !g.snake.Dead {
                return
        }
        vector.FillRect(screen, 0, 132, screenW, 22, bandColor,
false)
        g.face.drawCentred(screen, "GAME OVER  -  R FOR
ANOTHER", screenW/2, 136)
}


go vet ./...
go run ./cmd/snake
```



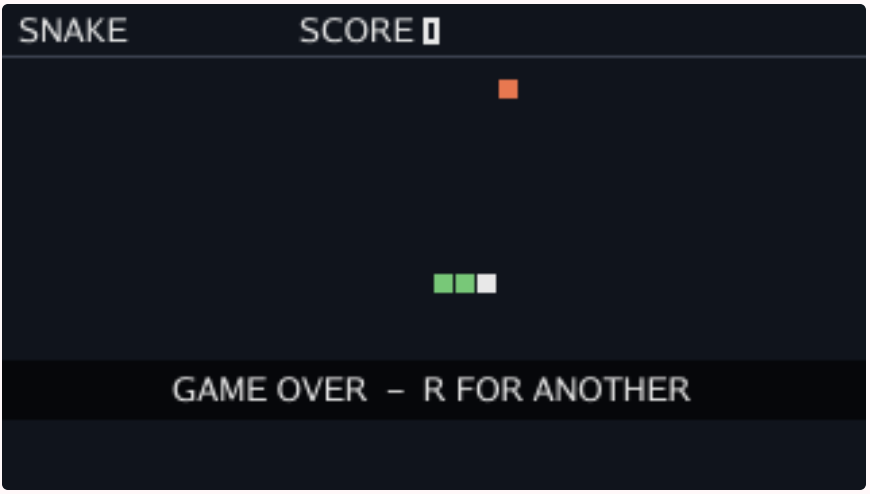
The end of the game that started in the last picture, 41 seconds in: 28 foods, 31 cells, a head with nowhere to go, and the band.

Run into the border or into the body and the lower tone sounds, the snake stops, and a band says so; R starts another game with the food somewhere new. Death is decided before the move, on the cell the head is about to enter, so a dead snake is drawn where it stopped and not one cell into the wall. The body test is `Has`, a walk along the ring from the head to the tail, thirty-one cells at the end of the game in the picture and never more than 800; the walk is eight hundred comparisons at the worst, once every three ticks, which the game does not notice. In that picture the snake was steered toward each food by the shortest path and folded itself into a block it could not leave, which is how most games of Snake end.

`restart` is where a new game begins, and `newSnake` calls it too, so that the first game and every game after start the same way, from the same lines. The generator is the one thing `restart` leaves alone: it carries on from wherever it was, so the food after R is not where the food was at the start. A dead game reads only R; the arrows are ignored until the snake is alive again.

⚠ WORKED FAILURE — A TURN INTO THE NECK

Stage 5's `Step` queues any arrow as a turn. Head right, tap Left:



Half a second after stage 5 starts, with Left tapped on tick 20: the snake is dead where it stood, at cell (22, 10), with a score of nought.

The snake dies on its next move without moving. Left was queued on tick 20, the move on tick 24 took it, and the cell one step left of the head was the neck, the second cell of the body, which `Has` found and called death. The rule is right; the turn is wrong, and no player means it: reversing a snake into itself is never what a tap of the opposite arrow is for. It is a common way to lose, on a keyboard where Left and Right are a finger apart, and the fix is to refuse the reversal in `Step`, before it is queued.

▣ BUILD · STAGE 6 — THE OPPOSITE OF THE HEADING IS NOT A TURN, EXTEND BOTH FILES

```
// cmd/snake/grid.go — extend
// Opposite is the direction that undoes d: up for down, left
for right.
func (d Dir) Opposite() Dir {
```

```
        return Dir{-d.DX, -d.DY}
}


// cmd/snake/game.go — extend
// Step advances the game by one tick with the keys held during
it. An
// arrow held on any tick queues a turn, unless it is the
opposite of the
// way the snake is heading; the turn is taken on the tick the
timer fires,
// and only the last one queued counts. A dead game waits for R.
func (g *Snake) Step(k keys) event {
        if g.Dead {
                if k.r {
                        g.restart()
                }
                return nothing
        }
        if d, ok := turn(k); ok && d ≠ g.Heading.Opposite() {
                g.Queued = d
        }
        g.Wait--
        if g.Wait > 0 {
                return nothing
        }
        g.Wait = g.Interval
        return g.move()
}


go vet ./...
go run ./cmd/snake
```

A tap of the opposite arrow now does nothing, and the snake carries on. The test is against the heading, the direction the snake is going, and not against the queued turn: heading right with Up queued, a tap of Left is refused, because the snake has not turned up yet and a left turn taken on the same move would still go into the neck. A tap of Down in the same situation is accepted and replaces the Up, which is the queue doing its job.

7. Saving the high score

A best score should outlive the window it was scored in, and the place for a small file a program keeps for its user is the directory the operating system sets aside for exactly that: `os.UserConfigDir`

returns it, which is `~/.config` on Linux, `~/Library/Application Support` on macOS and `AppData\Roaming` on Windows, and the game keeps `gez/snake.json` under it. The file holds one number, as JSON, so that it can be read by eye and edited by hand.

▣ BUILD · STAGE 7 — THE BEST SCORE, KEPT, A NEW FILE AND AN EXTENSION OF MAIN.GO

```
// cmd/snake/highscore.go — create
package main

import (
        "encoding/json"
        "errors"
        "io/fs"
        "os"
        "path/filepath"
)

// The high score lives in one small JSON file in the directory
the
// operating system keeps a user's settings in:
~/.config/gez/snake.json
// on Linux, and the equivalent elsewhere.
type highScore struct {
        High int `json:"high"`
}

// scorePath is where the file lives.
func scorePath() (string, error) {
        dir, err := os.UserConfigDir()
        if err ≠ nil {
                return "", err
        }
        return filepath.Join(dir, "gez", "snake.json"), nil
}

// loadHighScore reads the file. A missing file is a score of
zero and no
// error: the first game ever has nothing to beat.
func loadHighScore() (int, error) {
        path, err := scorePath()
        if err ≠ nil {
                return 0, err
        }
        b, err := os.ReadFile(path)
        if errors.Is(err, fs.ErrNotExist) {
                return 0, nil
        }
        if err ≠ nil {
```

```
                return 0, err
        }
        var hs highScore
        if err := json.Unmarshal(b, &hs); err ≠ nil {
                return 0, err
        }
        return hs.High, nil
}

// saveHighScore writes the file, making its directory if it is
missing.
func saveHighScore(high int) error {
        path, err := scorePath()
        if err ≠ nil {
                return err
        }
        if err := os.MkdirAll(filepath.Dir(path), 0o755); err ≠
nil {
                return err
        }
        b, err := json.Marshal(highScore{High: high})
        if err ≠ nil {
                return err
        }
        return os.WriteFile(path, append(b, '\n'), 0o644)
}


// cmd/snake/main.go — extend
// Game is Snake's window: the snake, the best score so far, and
everything
// drawn or heard.
type Game struct {
        snake    *Snake
        high     int // the best score, this session or from the
file
        font     *digits.Font
        face     *face
        eat, die *sound.Sound
}

// newGame builds the snake from the flags, reads the high
score, and loads
// what the window draws and plays.
func newGame() (*Game, error) {
        high, err := loadHighScore()
        if err ≠ nil {
                return nil, err
        }
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        eat, err := sound.Load("assets/hit.wav")
```

```
        if err ≠ nil {
                return nil, err
        }
        die, err := sound.Load("assets/point.wav")
        if err ≠ nil {
                return nil, err
        }
        return &Game{snake: newSnake(*seedFlag), high: high,
font: font, face: newFace(12), eat: eat, die: die}, nil
}

// step advances the game one tick, plays what it reports, and
keeps the
// high score when a game ends above it.
func (g *Game) step(k keys) {
        switch g.snake.Step(k) {
        case ate:
                g.eat.Play()
        case died:
                g.die.Play()
                if g.snake.Score > g.high {
                        g.high = g.snake.Score
                        if err := saveHighScore(g.high); err ≠
nil {
                                log.Print(err)
                        }
                }
        }
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawGrid(screen)
        g.drawFood(screen)
        g.drawBody(screen)
        g.drawScore(screen)
        g.drawBest(screen)
        g.drawOver(screen)
}

// drawBest draws the best score beside the score.
func (g *Game) drawBest(screen *ebiten.Image) {
        g.face.draw(screen, "BEST", 220, 3)
        g.font.Draw(screen, g.high, 256, 5, 2)
}


go vet ./...
go run ./cmd/snake
cat ~/.config/gez/snake.json
```

```
$ cat ~/.config/gez/snake.json
```

```
{"high":28}
```

The end of the same game with the best score in the strip: 28, kept in the file the moment the snake died.

The strip now reads BEST as well as SCORE, from the file at the start and from the game as it ends: on the tick the snake dies, if the score beats the best, the best is updated and written out, before the band is even drawn. The line quoted is the file after the game in the picture, on Linux. A game closed by the window button has already saved, because saving happens on death and not on exit, and a game that never dies had nothing to save.

`loadHighScore` treats a missing file as a score of zero and any other trouble as an error, because the first run of the game has no file and that is not a fault, while a file that exists and cannot be read is. `json.Marshal` turns the struct into `{"high":28}` using the name in the field's tag, and `Unmarshal` reads it back; the newline appended to the file is for the `cat` above. A failure to save is printed with `log.Print` and the game goes on, since a score that could not be written is no reason to stop playing.

8. Separating cells from pixels

Snake's rules use cells and whole numbers. A position is a cell, a move is one cell, and a turn is one of four directions. The timer turns ticks into moves at a tempo the game can change with one integer.

The ring buffer fits because the body changes only at its two ends. A new head is written. The tail is kept or dropped. The cells between them do not move.

The window code turns cells into pixels when it draws. The rules do not need to know how large a cell is on screen.

9. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Put a game on a grid of cells with a two-integer position and a two-integer direction, and move on a timer of two integers.
- Queue a turn between moves, keep only the last one, and say why the heading and not the queue decides whether a reversal is refused.
- Store a body in a ring buffer and give the slot of the cell four behind the head when the head is in slot 2 of 800.
- Place food on a free cell by drawing and rejecting, and say when that stops being a good way to do it.
- Grow a snake by owing a move, speed it up every fifth food, and end the game on the border or the body without moving into either.
- Keep one number in a JSON file in the user's configuration directory, read it at the start, and write it on the tick it changes.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — through the wall. Make the border wrap: a head that leaves on the right comes in on the left, and the same top to bottom.**

In `move`, after `next` is worked out, fold it: `next.X = (next.X + cols) % cols` and the same for rows, and drop `!next.Inside()` from the death test. The body test still applies, so the snake can still bite itself coming round; the picture shows the head appearing on the far side one move later.

▼ **Exercise 2 — how many draws. Count how many cells `place` draws over a whole game, and print the count when the snake dies.**

A `Draws` field incremented inside the loop and printed from the window's `step` on `died`. A second draw happens only when the first lands under the body, which covers a few percent of the grid in a game like the one in the pictures, so the count comes out a little above the number of placements; a snake would have to fill half the grid before draws doubled.

▼ **Exercise 3 — a faster start. Run with `startInterval` at 4 and then at 2 and say what changes about steering, not speed.**

At 4 a tap has four ticks to land before the move, and two quick taps for a U-turn usually both land in one move and only the second counts; at 2 the second tap of any pair lands after the move. The queue holds one turn, so the faster the tempo the more a player has to time taps to moves, which is the speed-up's difficulty and not the distance covered.

VOLUME 1 · CHAPTER 9

Breakout

A wall of bricks that come apart by the side they were hit on, three levels and three lives

1. Building Breakout

This chapter builds Breakout. The game has a paddle along the bottom, a ball, a wall of bricks, three levels, three balls and the same two short sounds.

The new rule is brick collision. An overlap says the ball is inside a brick, but it does not say which side the ball crossed. The game compares the overlap on x with the overlap on y, then reflects only the velocity across the smaller overlap. Reflecting both components sends the ball back the way it came.

2. Serving from the paddle

▣ BUILD · STAGE 1 — THE FIELD, THE PADDLE, THE BALL AND THE SERVE

```
// cmd/breakout/board.go — create
package main

import "math"

// The field is the picture below a twenty-pixel strip for the
score. The
// ball bounces off its left, right and top edges and is lost
past the
// bottom.
const (
        fieldW = 320
        fieldH = 180
        top    = 20 // the strip above the field, in pixels
```

```
)

// The paddle is thirty-two pixels wide and four tall on row
172, moved
// three pixels a tick by the arrows and never off the field.
The ball is
// four pixels square.
const (
        paddleW     = 32
        paddleH     = 4
        paddleY     = 172
        paddleSpeed = 3
        ballSize    = 4
)

// Rect is an axis-aligned rectangle: its top-left corner, width
and height.
type Rect struct {
        X, Y, W, H float64
}

// Overlaps reports whether two rectangles share any area.
Rectangles that
// only touch along an edge do not overlap.
func (r Rect) Overlaps(o Rect) bool {
        return r.X < o.X+o.W && o.X < r.X+r.W && r.Y < o.Y+o.H
&& o.Y < r.Y+r.H
}

// Ball is the ball: where it is and how far it moves each tick.
type Ball struct {
        X, Y   float64
        VX, VY float64
}

// Rect is the ball's rectangle.
func (b Ball) Rect() Rect {
        return Rect{X: b.X, Y: b.Y, W: ballSize, H: ballSize}
}

// paddleRect is the paddle's rectangle, from its left edge.
func paddleRect(x float64) Rect {
        return Rect{X: x, Y: paddleY, W: paddleW, H: paddleH}
}

// clamp returns v held inside [lo, hi].
func clamp(v, lo, hi float64) float64 {
        if v < lo {
                return lo
        }
        if v > hi {
                return hi
        }
        return v
```

```
}

// The serve leaves the paddle at two pixels a tick, three
across for every
// four up; the paddle returns the ball at an angle set by where
the ball
// struck it, up to sixty degrees from straight up.
const (
        serveSpeed = 2.0
        serveDX    = 0.6 // of the speed, to the right
        serveDY    = 0.8 // of the speed, upward
        maxAngle   = 60.0
)

// A game is in one of two states: waiting to serve, with the
ball on the
// paddle, or in play.
type State uint8

const (
        Serve State = iota // the ball rides the paddle until
Space is held
        Play               // the ball is in flight
)

// event is what a tick did that a player would hear: nothing,
the paddle
// struck, or the ball lost.
type event uint8

const (
        nothing event = iota
        paddle
        lost
)

// Board is the state of one game of Breakout, advanced one tick
at a time
// by Step.
type Board struct {
        Paddle float64 // the paddle's left edge; its row never
changes
        Ball   Ball
        State  State
}

// newBoard returns a game at tick zero: the paddle centred and
the ball on
// it, waiting to serve.
func newBoard() *Board {
        b := &Board{Paddle: (fieldW - paddleW) / 2}
        b.rest()
        return b
}
```

```
// rest puts the ball on the middle of the paddle, at rest, and
waits.
func (b *Board) rest() {
        b.Ball = Ball{X: b.Paddle + (paddleW-ballSize)/2, Y:
paddleY - ballSize}
        b.State = Serve
}

// Step advances the game by one tick with the keys held during
it, and
// reports what happened that a player would hear.
func (b *Board) Step(k keys) event {
        b.movePaddle(k.left, k.right)
        switch b.State {
        case Serve:
                b.Ball.X = b.Paddle + (paddleW-ballSize)/2 //
the ball rides the paddle
                if k.space {
                        b.serve()
                }
        case Play:
                if ev := b.fly(); ev ≠ nothing {
                        return ev
                }
                if b.Ball.Y ⩾ fieldH {
                        b.rest()
                        return lost
                }
        }
        return nothing
}

// movePaddle steps the paddle three pixels left, three right,
or not at
// all, and keeps it on the field.
func (b *Board) movePaddle(left, right bool) {
        if left {
                b.Paddle -= paddleSpeed
        }
        if right {
                b.Paddle += paddleSpeed
        }
        b.Paddle = clamp(b.Paddle, 0, fieldW-paddleW)
}

// serve sends the ball up and to the right at the serve speed.
func (b *Board) serve() {
        b.Ball.VX = serveDX * serveSpeed
        b.Ball.VY = -serveDY * serveSpeed
        b.State = Play
}

// fly moves the ball one tick, mirrors it off the three walls,
```

```
and returns
// it off the paddle, reporting what it met.
func (b *Board) fly() event {
        ball := &b.Ball
        ball.X += ball.VX
        ball.Y += ball.VY
        const right = fieldW - ballSize
        if ball.X < 0 {
                ball.X = -ball.X
                ball.VX = -ball.VX
        }
        if ball.X > right {
                ball.X = 2*right - ball.X
                ball.VX = -ball.VX
        }
        if ball.Y < top {
                ball.Y = 2*top - ball.Y
                ball.VY = -ball.VY
        }
        if ball.VY > 0 &&
ball.Rect().Overlaps(paddleRect(b.Paddle)) {
                b.returnBall()
                return paddle
        }
        return nothing
}

// returnBall sends the ball back up off the paddle at an angle
set by
// where it struck: the ball's centre against the paddle's, as a
fraction
// of half the paddle, times sixty degrees. The speed does not
change.
func (b *Board) returnBall() {
        ball := &b.Ball
        offset := ((ball.X + ballSize/2) - (b.Paddle +
paddleW/2)) / (paddleW / 2)
        offset = clamp(offset, -1, 1)
        angle := offset * maxAngle * math.Pi / 180
        speed := math.Hypot(ball.VX, ball.VY)
        ball.VX = speed * math.Sin(angle)
        ball.VY = -speed * math.Cos(angle)
        ball.Y = paddleY - ballSize
}


// cmd/breakout/main.go — create
package main

import (
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
```

```
        "github.com/hajimehoshi/ebiten/v2/vector"
)

var (
        fieldColor = color.RGBA{R: 16, G: 20, B: 28, A: 255}
        lineColor  = color.RGBA{R: 232, G: 232, B: 232, A: 255}
        dimColor   = color.RGBA{R: 60, G: 66, B: 80, A: 255}
        bandColor  = color.RGBA{R: 0, G: 0, B: 0, A: 160}
)

// keys is what the player is doing this tick: the two arrows,
Space and
// R, held or not.
type keys struct {
        left, right, space, r bool
}

// readKeys asks Ebitengine about the four keys, once a tick.
func readKeys() keys {
        return keys{
                left:  ebiten.IsKeyPressed(ebiten.KeyArrowLeft),
                right:
ebiten.IsKeyPressed(ebiten.KeyArrowRight),
                space: ebiten.IsKeyPressed(ebiten.KeySpace),
                r:     ebiten.IsKeyPressed(ebiten.KeyR),
        }
}

// The ball leaves a trail of its last twelve positions.
const trailLen = 12

// Game is Breakout's window: the board, and everything drawn.
type Game struct {
        board *Board
        trail []Ball // where the ball has been, oldest first
}

// newGame builds the board.
func newGame() *Game {
        return &Game{board: newBoard()}
}

// step advances the board one tick and keeps the trail.
func (g *Game) step(k keys) {
        g.board.Step(k)
        if g.board.State == Play {
                g.trail = append(g.trail, g.board.Ball)
                if len(g.trail) > trailLen {
                        g.trail = g.trail[1:]
                }
        } else {
                g.trail = g.trail[:0]
        }
}
```

```
func (g *Game) Update() error {
        g.step(readKeys())
        return nil
}

// fillRect fills a Rect, rounded down to the pixel its corner
is in.
func fillRect(dst *ebiten.Image, r Rect, clr color.RGBA) {
        vector.FillRect(dst, float32(int(r.X)),
float32(int(r.Y)), float32(r.W), float32(r.H), clr, false)
}

// drawField paints the background and the line under the score
strip.
func drawField(screen *ebiten.Image) {
        screen.Fill(fieldColor)
        vector.FillRect(screen, 0, top-1, fieldW, 1, dimColor,
false)
}

// drawBall draws the trail, oldest and dimmest first, then the
paddle and
// the ball.
func (g *Game) drawBall(screen *ebiten.Image) {
        for i, b := range g.trail {
                c := dimColor
                c.A = uint8(255 * (i + 1) / (trailLen + 1))
                fillRect(screen, Rect{X: b.X + 1, Y: b.Y + 1, W:
2, H: 2}, c)
        }
        fillRect(screen, paddleRect(g.board.Paddle), lineColor)
        fillRect(screen, g.board.Ball.Rect(), lineColor)
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawField(screen)
        g.drawBall(screen)
}

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

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Breakout")
        ebiten.SetTPS(60)
        if err := ebiten.RunGame(newGame()); err ≠ nil {
                log.Fatal(err)
        }
}
```

```
go vet ./...
go run ./cmd/breakout
```

The window after stage 1: the paddle on row 172 and the ball resting on it, waiting for Space.

Twenty-five ticks after a serve: the ball has climbed forty pixels and drifted thirty to the right, and its trail points back to where it left the paddle.

The ball sits on the paddle and slides with it as the arrows move it; Space sends it up and to the right, three pixels across for

every four up, and it comes off the left, right and top edges of the field and down to the paddle, which returns it at an angle set by where it struck, the way Pong's paddles do turned upright. Miss it, and it drops out of the bottom and comes back to rest on the paddle. Most of `board.go` is Pong's `match.go` with the axes swapped: `Rect`, `Overlaps`, `Ball`, `clamp`, the mirror off a wall, and a `returnBall` whose offset runs along the paddle and whose angle is measured from straight up, so that the sine goes across and the cosine goes up. A serve of 0.6 across and 0.8 up at two pixels a tick is a 3-4-5 triangle: 1.2 across, 1.6 up, two along the diagonal.

The ball rides the paddle while the game waits, by copying the paddle's position into the ball's on every tick of the Serve state, and `rest` is what puts it there and sets the state, from three places before the chapter is done. The window adds one thing Pong's did not: a trail, the ball's last twelve positions kept in a slice and drawn as two-by-two dots that fade toward the oldest, so that the ball's path can be seen in a still picture, which is what makes the worked failure below visible. The trail is a fact about the picture and lives in the window; it is emptied whenever the ball is not in flight.

3. Resolving brick hits

A brick is 28 pixels wide and 8 tall, and ten of them with four-pixel gaps are 316 pixels, two short of the field on either side. Rows start at row 24 with a two-pixel gap, up to eight of them. A wall is a small table of hit points, one number a brick, 0 for a gap or a brick that is gone, and a wall is written as text: ten characters a row, a digit for the hit points, a dot for a gap. The first wall goes in a file, which the game reads when it starts.

▣ BUILD · STAGE 2 — THE WALL, FROM A FILE, AND WHICH EDGE THE BALL CAME THROUGH

```
# configs/levels.txt — create
# Breakout's levels: one character a brick, a digit for its hit
points, a
# dot for a gap, ten characters a row, a blank line between
levels.
3333.33333
2222.22222
1111.11111
1111.11111


// cmd/breakout/bricks.go — create
package main

import (
        "fmt"
        "math"
)

// Bricks are twenty-eight pixels wide and eight tall, in ten
columns with
// four-pixel gaps: 10 × 28 + 9 × 4 is 316, centred with two
pixels each
// side. Rows start at row 24 with two-pixel gaps, up to eight
of them. A
// brick has one, two or three hit points.
const (
        cols     = 10
        maxRows  = 8
        brickW   = 28
        brickH   = 8
        gapX     = 4
        gapY     = 2
        margin   = (fieldW - (cols*brickW + (cols-1)*gapX)) / 2
// 2
        firstRow = 24
        maxHP    = 3
)

// Wall is the bricks' hit points by row and column: 1 to 3 for
a brick, 0
// for a gap or a brick that is gone.
type Wall [maxRows][cols]int

// brickRect is the rectangle of the brick in column col and row
row.
func brickRect(col, row int) Rect {
        return Rect{
                X: margin + float64(col)*(brickW+gapX),
                Y: firstRow + float64(row)*(brickH+gapY),
                W: brickW, H: brickH,
        }
}
```

```
// Left counts the bricks still standing.
func (w *Wall) Left() int {
        n := 0
        for _, row := range w {
                for _, hp := range row {
                        if hp > 0 {
                                n++
                        }
                }
        }
        return n
}

// parseRow reads one row of a wall from ten characters: a digit
for a
// brick's hit points, a dot for a gap.
func parseRow(row string) ([cols]int, error) {
        var out [cols]int
        if len(row) ≠ cols {
                return out, fmt.Errorf("%d characters, want %d",
len(row), cols)
        }
        for c, ch := range row {
                switch {
                case ch == '.':
                case ch ⩾ '1' && ch ⩽ '0'+maxHP:
                        out[c] = int(ch - '0')
                default:
                        return out, fmt.Errorf("column %d: %q is
not a digit 1-%d or a dot", c+1, ch, maxHP)
                }
        }
        return out, nil
}

// parseWall reads a wall from text: one string a row, one
character a
// brick. There are at most eight rows.
func parseWall(rows []string) (Wall, error) {
        var w Wall
        if len(rows) > maxRows {
                return w, fmt.Errorf("%d rows, at most %d
allowed", len(rows), maxRows)
        }
        for r, row := range rows {
                parsed, err := parseRow(row)
                if err ≠ nil {
                        return w, fmt.Errorf("row %d: %w", r+1,
err)
                }
                w[r] = parsed
        }
        return w, nil
```

```
}

// Centre is the middle of the rectangle.
func (r Rect) Centre() (x, y float64) {
        return r.X + r.W/2, r.Y + r.H/2
}

// Penetration is how far r reaches into o along each axis: the
smaller of
// the two overlaps on x, and the smaller of the two on y. Both
are
// positive when the rectangles overlap.
func (r Rect) Penetration(o Rect) (px, py float64) {
        px = math.Min(r.X+r.W-o.X, o.X+o.W-r.X)
        py = math.Min(r.Y+r.H-o.Y, o.Y+o.H-r.Y)
        return px, py
}

// hitBrick takes a hit point off the brick the ball overlaps,
or, when it
// overlaps two, off the one whose centre is nearer the ball's,
and bounces
// the ball off it.
func (b *Board) hitBrick() bool {
        ball := b.Ball.Rect()
        bx, by := ball.Centre()
        struck, best := Rect{}, -1.0
        col, row := 0, 0
        for r := range b.Bricks {
                for c := range b.Bricks[r] {
                        br := brickRect(c, r)
                        if b.Bricks[r][c] == 0 ||
!ball.Overlaps(br) {
                                continue
                        }
                        cx, cy := br.Centre()
                        if d := math.Hypot(cx-bx, cy-by); best <
0 || d < best {
                                struck, best, col, row = br, d,
c, r
                        }
                }
        }
        if best < 0 {
                return false
        }
        b.Bricks[row][col]--
        b.Score++
        b.bounce(struck)
        return true
}

// bounce reflects the ball off the brick it struck: along x
when the ball
```

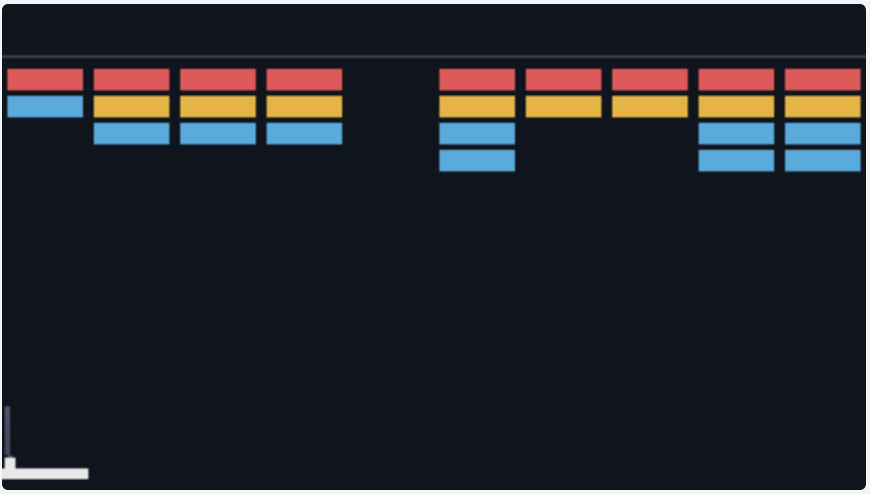
```
// went less far into the brick from a side than from the top or
bottom,
// along y otherwise, so a tie, which is a corner, counts as a
top or a
// bottom. One component flips and the other is left alone.
func (b *Board) bounce(brick Rect) {
        px, py := b.Ball.Rect().Penetration(brick)
        if px < py {
                b.Ball.VX = -b.Ball.VX
        } else {
                b.Ball.VY = -b.Ball.VY
        }
}


// cmd/breakout/levels.go — create
package main

import (
        "bufio"
        "fmt"
        "io"
        "os"
        "strings"
)

// parseLevels reads levels from text: the rows of a wall, a
blank line
// between levels, and lines beginning with # ignored. An error
names the
// line it was found on.
func parseLevels(r io.Reader) ([]Wall, error) {
        var levels []Wall
        var rows []string
        line := 0
        flush := func() error {
                if len(rows) == 0 {
                        return nil
                }
                w, err := parseWall(rows)
                rows = nil
                if err != nil {
                        return fmt.Errorf("line %d: %w", line,
err)
                }
                levels = append(levels, w)
                return nil
        }
        sc := bufio.NewScanner(r)
        for sc.Scan() {
                line++
                text := strings.TrimRight(sc.Text(), " \t")
                switch {
                case strings.HasPrefix(text, "#"):
```

```
                case text == "":
                        if err := flush(); err ≠ nil {
                                return nil, err
                        }
                default:
                        if _, err := parseRow(text); err ≠ nil
{
                                return nil, fmt.Errorf("line %d:
%w", line, err)
                        }
                        rows = append(rows, text)
                }
        }
        if err := sc.Err(); err ≠ nil {
                return nil, err
        }
        if err := flush(); err ≠ nil {
                return nil, err
        }
        if len(levels) == 0 {
                return nil, fmt.Errorf("no levels")
        }
        return levels, nil
}

// loadLevels reads the levels in the file at path.
func loadLevels(path string) ([]Wall, error) {
        f, err := os.Open(path)
        if err ≠ nil {
                return nil, err
        }
        defer f.Close()
        levels, err := parseLevels(f)
        if err ≠ nil {
                return nil, fmt.Errorf("%s: %w", path, err)
        }
        return levels, nil
}


// cmd/breakout/board.go — extend
// event is what a tick did that a player would hear: nothing, a
brick
// struck, the paddle struck, or the ball lost.
type event uint8

const (
        nothing event = iota
        brick
        paddle
        lost
)

// Board is the state of one game of Breakout, advanced one tick
```

```
at a time
// by Step.
type Board struct {
        Paddle float64 // the paddle's left edge; its row never
changes
        Ball   Ball
        Bricks Wall // hit points, by row and column
        State  State
        Score  int    // one point a hit
        Levels []Wall // the walls, in order; read at the start
and never changed
}

// newBoard returns a game of the levels given, at tick zero:
the paddle
// centred, the ball on it, the first wall up, waiting to serve.
func newBoard(levels []Wall) *Board {
        b := &Board{Levels: levels, Bricks: levels[0], Paddle:
(fieldW - paddleW) / 2}
        b.rest()
        return b
}

// fly moves the ball one tick, mirrors it off the three walls,
returns it
// off the paddle, and takes a hit point off a brick it struck,
reporting
// what it met.
func (b *Board) fly() event {
        ball := &b.Ball
        ball.X += ball.VX
        ball.Y += ball.VY
        const right = fieldW - ballSize
        if ball.X < 0 {
                ball.X = -ball.X
                ball.VX = -ball.VX
        }
        if ball.X > right {
                ball.X = 2*right - ball.X
                ball.VX = -ball.VX
        }
        if ball.Y < top {
                ball.Y = 2*top - ball.Y
                ball.VY = -ball.VY
        }
        if ball.VY > 0 &&
ball.Rect().Overlaps(paddleRect(b.Paddle)) {
                b.returnBall()
                return paddle
        }
        if b.hitBrick() {
                return brick
        }
```

```
        return nothing
}


// cmd/breakout/main.go — extend
// newGame reads the levels and builds the board.
func newGame() (*Game, error) {
        levels, err := loadLevels("configs/levels.txt")
        if err ≠ nil {
                return nil, err
        }
        return &Game{board: newBoard(levels)}, nil
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawField(screen)
        g.drawWall(screen)
        g.drawBall(screen)
}

func main() {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Breakout")
        ebiten.SetTPS(60)
        g, err := newGame()
        if err ≠ nil {
                log.Fatal(err)
        }
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
}

// brickColors is the colour of a brick by its hit points: index
1 to 3.
var brickColors = [maxHP + 1]color.RGBA{
        {},
        {R: 90, G: 170, B: 220, A: 255},
        {R: 230, G: 180, B: 70, A: 255},
        {R: 220, G: 90, B: 90, A: 255},
}

// drawWall draws every brick still standing, in the colour of
its hit
// points.
func (g *Game) drawWall(screen *ebiten.Image) {
        for r, row := range g.board.Bricks {
                for c, hp := range row {
                        if hp > 0 {
                                fillRect(screen, brickRect(c,
r), brickColors[hp])
                        }
                }
```

```
        }
}


go vet ./...
go run ./cmd/breakout
```



Fifteen seconds into a game with the first wall up: a red brick has been hit twice on the left, and blue ones have gone in both halves; the paddle has gone to the left edge to meet the ball there.

The wall is up, in three colours for three hit points, and every brick the ball strikes loses a point, turns the next colour, and finally goes; the fifth column is a gap the ball can fly through. The wall came from the file: `loadLevels` reads it, skips the comment lines, and hands `parseWall` one paragraph, which turns ten characters a row into ten hit points a row and refuses anything else, naming the line. A file that reads `configs/levels.txt: line 5: 11 characters, want 10` in the terminal has a row with a character too many on its fifth line, and the game does not start until it is fixed, which is better than a wall with a brick hanging off its edge.

`hitBrick` walks the whole wall, eighty bricks at most, for the ones the ball overlaps, and takes the hit off the one whose centre is nearest the ball's: a ball that lands in the gap between two

bricks overlaps both and should strike one. Then `bounce` decides the edge. `Penetration` is two subtractions an axis: how far the ball's right edge is past the brick's left, or the brick's right past the ball's left, whichever is smaller, and the same for the vertical, and the smaller of the two is the edge the ball came through. The recording the pictures come from strikes its first brick from the side on tick 706: the ball at (29.79, 60.89), moving up and slightly right, reaches 0.21 pixels into the leftmost brick of the fourth row on x and 1.11 on y, so it has come through the brick's right edge, and it leaves moving left, its vertical speed untouched.

∑ MATH INTERLUDE — THE SMALLER PENETRATION

The brick struck on tick 706 spans columns 2 to 30 and rows 54 to 62, and the ball, four wide, has its left edge at 29.79 and its top at 60.89. On x, the ball's right edge, 33.79, is 31.79 past the brick's left edge, and the brick's right edge, 30, is 0.21 past the ball's left: the smaller, 0.21, is how far the ball has come in from the right. On y, the ball's bottom, 64.89, is 10.89 past the brick's top and the brick's bottom, 62, is 1.11 past the ball's top: 1.11 from below. A ball that has come 0.21 in from one side and 1.11 in from another came through the side it is least inside, so it struck the right edge, and only the across velocity flips. A tie is a corner, and the rule calls it a top or a bottom.

`px` how far the ball reaches into the brick from a side: the smaller of (ball's right − brick's left) and (brick's right − ball's left)
`py` the same from the top or bottom: the smaller of (ball's bottom − brick's top) and (brick's bottom − ball's top)
`px < py` the ball came through a side: flip VX and leave VY; otherwise flip VY and leave VX
`min` the smaller of two numbers, math.Min in Go

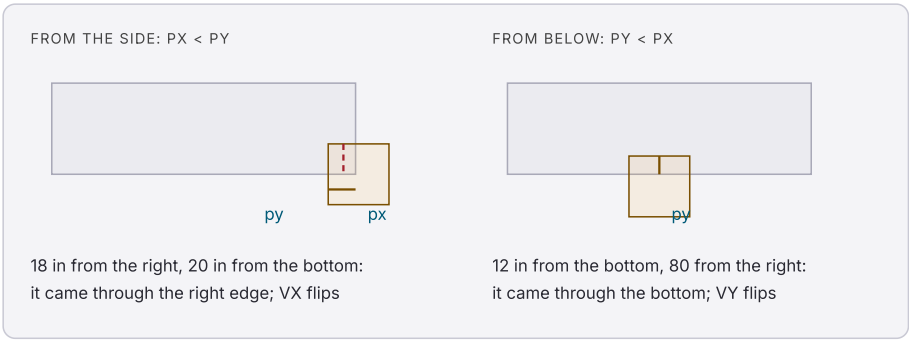
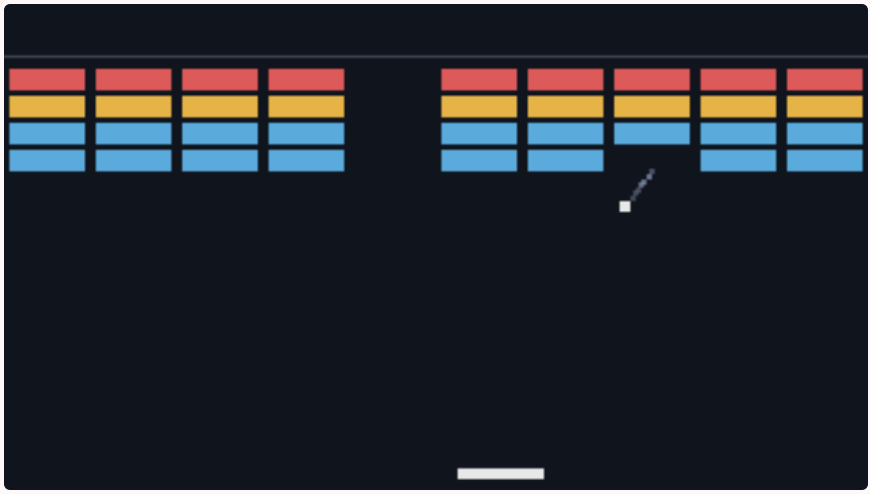


Figure 9.1 — the two penetrations of a ball into a brick, and the edge the smaller one names. Only the velocity across that edge is reflected.

⚠ WORKED FAILURE — BOTH COMPONENTS FLIPPED, AND A BALL THAT GOES BACK THE WAY IT CAME

The rule can be skipped. A ball that struck a brick has to leave it, and flipping both components leaves it every time, in one line:

```
func (b *Board) bounce(brick Rect) {
        b.Ball.VX, b.Ball.VY = -b.Ball.VX, -b.Ball.VY
}
```



Eight ticks after the first brick of a game, with both components flipped: the ball is going back down the line it came up, and its trail runs up to the brick

and back over itself.

The first brick of the game is struck on tick 82, from below, at 1.2 across and 1.6 up. On tick 81 the ball was at (237.20, 62.40); on tick 82 it struck at (238.40, 60.80) and left moving −1.2 across and 1.6 down; on tick 83 it was at (237.20, 62.40) again, on tick 84 at (236.00, 64.00), and so on down the same line it came up, to the same spot on the paddle it left. The trail in the picture is one line because the ball's last twelve positions include the way up and the way back down, and they coincide to the hundredth of a pixel. A player watching sees the ball come straight back at them off every brick, and a wall with a ball that retraces cannot be cleared from one place: the paddle has to move to change the angle, which it cannot do while the ball is in the air. A brick struck from below should send the ball back up and on, across, the way a ball does; the horizontal velocity was never the brick's to change. Put the penetration test back.

4. Loading levels

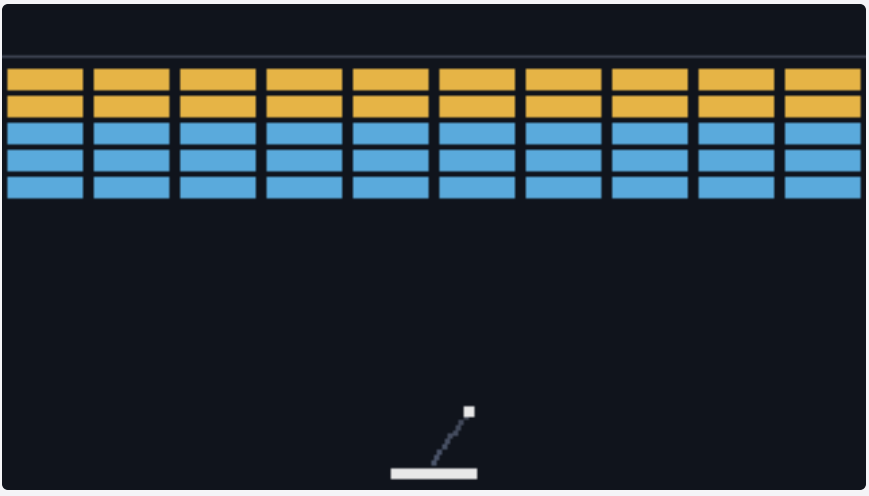
▣ BUILD · STAGE 3 — LEVELS, AND THE WALL THAT IS DOWN, A REPLACED FILE AND AN EXTENSION OF BOARD.GO

```
# configs/levels.txt — replace
# Breakout's levels: one character a brick, a digit for its hit
points, a
# dot for a gap, ten characters a row, a blank line between
levels.
3333.33333
2222.22222
1111.11111
1111.11111

2222222222
2222222222
1111111111
1111111111
1111111111

33.3333.33
```

```
22.2222.22
11.1111.11
11.1111.11
22.2222.22
33.3333.33


// cmd/breakout/board.go — extend
// The serve leaves the paddle at two pixels a tick, three
across for every
// four up; the paddle returns the ball at an angle set by where
the ball
// struck it, up to sixty degrees from straight up. Each level
serves a
// quarter of a pixel a tick faster, up to five.
const (
        serveSpeed = 2.0
        serveDX    = 0.6 // of the speed, to the right
        serveDY    = 0.8 // of the speed, upward
        maxAngle   = 60.0
        levelUp    = 0.25 // added to the serve speed by each
level
        maxSpeed   = 5.0
)

// A game is in one of three states: waiting to serve, with the
ball on
// the paddle; in play; or between levels, with a wall down and
the next
// waiting for Space.
type State uint8

const (
        Serve   State = iota // the ball rides the paddle until
Space is held
        Play                 // the ball is in flight
        Cleared              // a wall is down and the next
waits for Space
)

// event is what a tick did that a player would hear: nothing, a
brick
// struck, the paddle struck, the ball lost, or a wall down.
type event uint8

const (
        nothing event = iota
        brick
        paddle
        lost
        cleared
)

// Board is the state of one game of Breakout, advanced one tick
```

```
at a time
// by Step.
type Board struct {
        Paddle float64 // the paddle's left edge; its row never
changes
        Ball   Ball
        Bricks Wall // hit points, by row and column
        State  State
        Score  int     // one point a hit
        Speed  float64 // the speed the ball is served at on
this level
        Level  int     // which of the levels the wall is, from
0
        Levels []Wall  // the walls, in order; read at the start
and never changed
}

// newBoard returns a game of the levels given, at tick zero:
the paddle
// centred, the ball on it, the first wall up, waiting to serve.
func newBoard(levels []Wall) *Board {
        b := &Board{Levels: levels, Bricks: levels[0], Speed:
serveSpeed, Paddle: (fieldW - paddleW) / 2}
        b.rest()
        return b
}

// Step advances the game by one tick with the keys held during
it, and
// reports what happened that a player would hear.
func (b *Board) Step(k keys) event {
        b.movePaddle(k.left, k.right)
        switch b.State {
        case Serve:
                b.Ball.X = b.Paddle + (paddleW-ballSize)/2 //
the ball rides the paddle
                if k.space {
                        b.serve()
                }
        case Play:
                if ev := b.fly(); ev ≠ nothing {
                        if ev == brick && b.Bricks.Left() == 0 {
                                b.clear()
                                return cleared
                        }
                        return ev
                }
                if b.Ball.Y ⩾ fieldH {
                        b.rest()
                        return lost
                }
        case Cleared:
                if k.space {
                        b.next()
```

```
                }
        }
        return nothing
}

// serve sends the ball up and to the right at this level's
speed.
func (b *Board) serve() {
        b.Ball.VX = serveDX * b.Speed
        b.Ball.VY = -serveDY * b.Speed
        b.State = Play
}

// clear ends the level: the ball goes back to the paddle and
the game
// waits for Space.
func (b *Board) clear() {
        b.rest()
        b.State = Cleared
}

// next puts the next wall up, a quarter of a pixel a tick
faster.
func (b *Board) next() {
        b.Level++
        b.Bricks = b.Levels[b.Level]
        b.Speed = math.Min(serveSpeed+levelUp*float64(b.Level),
maxSpeed)
        b.rest()
}


go vet ./...
go run ./cmd/breakout
```



Forty ticks after the first wall came down and Space brought the second one up: five rows, no gap, and a serve at 2.25 pixels a tick.

Clear the wall and the ball comes back to the paddle and waits; Space brings up the second wall from the file, five rows and no gap, and serves a quarter of a pixel a tick faster; the third wall is the file's last paragraph. The game in the pictures took 8,637 ticks, two minutes and twenty-four seconds, to bring the first wall down. Two things changed in the rules. The wall's fall is checked on the tick of the hit that emptied it, in `Step`, where `Left` counts what stands; and the serve speed is a field now, set per level and capped at five for the same reason Pong's is. The file is read once, at the start, and `Levels` is never written after that: `next` copies a wall out of it, and the copy is what loses bricks. Go copies an array on assignment, which is what makes `Wall` an array and not a slice.

5. Adding lives, messages and sounds

▣ BUILD · STAGE 4 — LIVES, THE GAME'S END, THE STRIP AND THE SOUNDS, A TEXT FILE AND TWO EXTENSIONS

```
// cmd/breakout/text.go — create
package main

import (
        "bytes"
        "log"
        "math"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/text/v2"
        "golang.org/x/image/font/gofont/goregular"
)

// face is the TrueType face the game's words are set in: Go
Regular,
// carried inside the program.
type face struct {
        f *text.GoTextFace
}
```

```
// newFace parses the font once and returns it at the size asked
for.
func newFace(size float64) *face {
        src, err :=
text.NewGoTextFaceSource(bytes.NewReader(goregular.TTF))
        if err ≠ nil {
                log.Fatal(err)
        }
        return &face{f: &text.GoTextFace{Source: src, Size:
size}}
}

// draw draws s in the line colour with its left edge on column
x and its
// top on row y.
func (fc *face) draw(dst *ebiten.Image, s string, x, y float64)
{
        op := &text.DrawOptions{}
        op.GeoM.Translate(x, y)
        op.ColorScale.ScaleWithColor(lineColor)
        text.Draw(dst, s, fc.f, op)
}

// drawCentred draws s with its middle on column x and its top
on row y.
func (fc *face) drawCentred(dst *ebiten.Image, s string, x, y
float64) {
        w, _ := text.Measure(s, fc.f, 0)
        fc.draw(dst, s, math.Floor(x-w/2), y)
}


// cmd/breakout/board.go — extend
// The serve leaves the paddle at two pixels a tick, three
across for every
// four up; the paddle returns the ball at an angle set by where
the ball
// struck it, up to sixty degrees from straight up. Each level
serves a
// quarter of a pixel a tick faster, up to five, and a game has
three balls.
const (
        serveSpeed = 2.0
        serveDX    = 0.6 // of the speed, to the right
        serveDY    = 0.8 // of the speed, upward
        maxAngle   = 60.0
        levelUp    = 0.25 // added to the serve speed by each
level
        maxSpeed   = 5.0
        balls      = 3
)

// A game is in one of four states: waiting to serve, with the
```

```
ball on the
// paddle; in play; between levels, with a wall down and the
next waiting
// for Space; or over, with the last ball lost or the last wall
down.
type State uint8

const (
        Serve   State = iota // the ball rides the paddle until
Space is held
        Play                 // the ball is in flight
        Cleared              // a wall is down and the next
waits for Space
        Over                 // the third ball is lost, or the
last wall is down; R restarts
)

// Board is the state of one game of Breakout, advanced one tick
at a time
// by Step.
type Board struct {
        Paddle float64 // the paddle's left edge; its row never
changes
        Ball   Ball
        Bricks Wall // hit points, by row and column
        State  State
        Score  int     // one point a hit
        Speed  float64 // the speed the ball is served at on
this level
        Level  int     // which of the levels the wall is, from
0
        Balls  int     // balls left, counting the one in play
or on the paddle
        Levels []Wall  // the walls, in order; read at the start
and never changed
}

// newBoard returns a game of the levels given: the paddle
centred, the
// ball on it, the first wall up, three balls, waiting to serve.
func newBoard(levels []Wall) *Board {
        b := &Board{Levels: levels}
        b.restart()
        return b
}

// Step advances the game by one tick with the keys held during
it, and
// reports what happened that a player would hear.
func (b *Board) Step(k keys) event {
        b.movePaddle(k.left, k.right)
        switch b.State {
        case Serve:
                b.Ball.X = b.Paddle + (paddleW-ballSize)/2 //
```

```
the ball rides the paddle
                if k.space {
                        b.serve()
                }
        case Play:
                if ev := b.fly(); ev ≠ nothing {
                        if ev == brick && b.Bricks.Left() == 0 {
                                b.clear()
                                return cleared
                        }
                        return ev
                }
                if b.Ball.Y ⩾ fieldH {
                        b.lose()
                        return lost
                }
        case Cleared:
                if k.space {
                        b.next()
                }
        case Over:
                if k.r {
                        b.restart()
                }
        }
        return nothing
}

// clear ends the level, or the game when it was the last level.
func (b *Board) clear() {
        b.rest()
        b.State = Cleared
        if b.Level == len(b.Levels)-1 {
                b.State = Over
        }
}

// lose takes a ball away, and ends the game when it was the
last.
func (b *Board) lose() {
        b.Balls--
        b.rest()
        if b.Balls == 0 {
                b.State = Over
        }
}

// restart begins again at the first level with three balls and
no score.
func (b *Board) restart() {
        b.Score, b.Level, b.Balls = 0, 0, balls
        b.Bricks, b.Speed = b.Levels[0], serveSpeed
        b.Paddle = (fieldW - paddleW) / 2
```

```
        b.rest()
}


// cmd/breakout/main.go — extend
import (
        "fmt"
        "image/color"
        "log"

        "github.com/hajimehoshi/ebiten/v2"
        "github.com/hajimehoshi/ebiten/v2/vector"

        "gez/internal/digits"
        "gez/internal/sound"
)

// Game is Breakout's window: the board, and everything drawn or
heard.
type Game struct {
        board    *Board
        trail    []Ball // where the ball has been, oldest first
        font     *digits.Font
        face     *face
        hit, die *sound.Sound
}

// newGame reads the levels and loads what the window draws and
plays.
func newGame() (*Game, error) {
        levels, err := loadLevels("configs/levels.txt")
        if err ≠ nil {
                return nil, err
        }
        font, err := digits.Load("assets/pong-sheet.png")
        if err ≠ nil {
                return nil, err
        }
        hit, err := sound.Load("assets/hit.wav")
        if err ≠ nil {
                return nil, err
        }
        die, err := sound.Load("assets/point.wav")
        if err ≠ nil {
                return nil, err
        }
        return &Game{board: newBoard(levels), font: font, face:
newFace(12), hit: hit, die: die}, nil
}

// step advances the board one tick, keeps the trail, and plays
what the
// board reports.
func (g *Game) step(k keys) {
```

```
        switch g.board.Step(k) {
        case brick, paddle:
                g.hit.Play()
        case lost, cleared:
                g.die.Play()
        }
        if g.board.State == Play {
                g.trail = append(g.trail, g.board.Ball)
                if len(g.trail) > trailLen {
                        g.trail = g.trail[1:]
                }
        } else {
                g.trail = g.trail[:0]
        }
}

func (g *Game) Draw(screen *ebiten.Image) {
        drawField(screen)
        g.drawWall(screen)
        g.drawBall(screen)
        g.drawHUD(screen)
}

// drawHUD draws the level, the balls left and the score in the
strip,
// and a band across the field while the game waits between
levels or at
// its end.
func (g *Game) drawHUD(screen *ebiten.Image) {
        b := g.board
        g.face.draw(screen, "LEVEL", 6, 3)
        g.font.Draw(screen, b.Level+1, 50, 5, 2)
        g.face.draw(screen, "BALLS", 100, 3)
        g.font.Draw(screen, b.Balls, 146, 5, 2)
        g.face.draw(screen, "SCORE", 210, 3)
        g.font.Draw(screen, b.Score, 256, 5, 2)
        if b.State != Cleared && b.State != Over {
                return
        }
        vector.FillRect(screen, 0, 100, fieldW, 22, bandColor,
false)
        switch {
        case b.State == Cleared:
                g.face.drawCentred(screen, fmt.Sprintf("CLEARED
-  SPACE FOR LEVEL %d", b.Level+2), fieldW/2, 104)
        case b.Bricks.Left() == 0:
                g.face.drawCentred(screen, "EVERY WALL DOWN  -
R FOR ANOTHER", fieldW/2, 104)
        default:
                g.face.drawCentred(screen, "GAME OVER  -  R FOR
ANOTHER", fieldW/2, 104)
        }
}
```

```
go vet ./...
go run ./cmd/breakout
```

The tick after the first wall came down: 63 points for its 63 hit points, three balls still, and the band waiting for Space.

The end of the same game, 23,995 ticks in: every wall down, 229 points, one ball left.

Twenty seconds into a game in which the paddle never moved: three serves, three balls lost, five bricks struck on the way.

The strip shows the level, the balls left and the score, in the two fonts of chapter 5, and a band says what to do between levels and at the end; a brick or the paddle beeps, a lost ball or a fallen wall sounds the lower tone. A game has three balls, counting the one in play, and losing the third is the end; so is the last wall coming down, and the band says which. R begins again at the first level, from `restart`, which `newBoard` now calls too, as Snake's does. The three walls in the pictures took 229 hits, which is the sum of the hit points in the file: 63 in the first wall, 70 in the second and 96 in the third, one point each, so a score is also a count of how much wall has gone.

Breakout is three games' worth of pieces put together in a new order: Pong's ball and return, Snake's strip and states, and a wall that is data in a file. The file is the part to change first. A wall with a hole in the middle, a wall of single bricks, a wall eight rows deep, each is a paragraph of ten-character rows and no code, and a mistake in the paragraph is reported by line before the window opens.

6. Choosing the collision axis

An overlap test answers whether two rectangles share area. A bounce also needs to know which edge was crossed.

The two penetrations answer that question without storing the ball's previous position. If the ball's step is smaller than the brick, the axis with the smaller penetration is the axis it crossed. The cap of five pixels a tick keeps the step smaller than an eight-pixel brick.

7. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Turn Pong's paddle and ball upright, serve on a 3-4-5 triangle, and return the ball from a paddle along the bottom at an angle from straight up.
- Lay out a wall of bricks from ten-character rows in a file, name the line of a bad row, and draw each brick in the colour of its hit points.
- Compute the two penetrations of a ball into a brick by hand and say which edge it came through, for the strike on tick 706.
- Explain from three ticks' positions why flipping both components sends the ball back along its own trail.
- Move a game through Serve, Play, Cleared and Over with three balls and three walls, and bring the next wall up faster.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — a wall of your own. Add a fourth paragraph to `configs/levels.txt` with a hole in its middle and a row of threes at the bottom, and play to it.**

Six rows such as `1111111111`, `1111111111`, `111....111`, `111....111`, `2222222222`, `3333333333`, after a blank line.

The game reads four levels, the band after the third wall says `SPACE FOR LEVEL 4`, and the fourth serves at 2.75 pixels a tick. The hard bricks at the bottom take three hits each before the ball can reach the easy ones.

▼ **Exercise 2 — the corner rule, reversed. Change `px < py` to `px ⩽ py` and play a level. When does it matter?**

Only on an exact tie, a ball that has come the same distance in from a side and from the top or bottom, which is a corner struck dead on. With `⩽` a corner flips the horizontal velocity instead of the vertical; either choice is a rule, and the game plays the same nearly all the time, so a tie is the case to decide once and write down.

▼ **Exercise 3 — the trail as a ruler. Make the trail twenty-four positions long and count, from a still picture, how many pixels the ball moves a tick at each level.**

`trailLen = 24`, and the dots are one tick apart, so the distance between two dots is the speed: two pixels on the first wall, 2.25 on the second, 2.5 on the third, measured along the diagonal. A trail is a cheap way to see a velocity, and a debugging aid many games ship with a key to turn it on.

VOLUME 1 · CHAPTER 10

Testing Your Games (optional)

Record a match, replay it without a window, and let a test play it

1. Testing a game

This optional chapter is for checking Pong after you change it. It records the keys held on each tick, replays them with no window, runs that replay in a Go test and writes a PNG for a chosen tick.

Pong's rules learn the outside world through six booleans per tick. A recording is a text file containing those booleans as key names, one line per tick. If the replay uses the same seed and mode, a fresh match reaches the same result because `Step` receives the same inputs in the same order.

2. Recording keys

The recording's format is the simplest that can be read by eye: one line per tick, the names of the keys held on that tick separated by spaces, and an empty line for a tick with nothing held. A two-minute match is 7,000 lines and a few kilobytes. Two flags go on the program: `-record` names a file to write the run's keys to when the window closes, and `-replay` names a file to play with no window at all.

▣ BUILD · STAGE 1 — RECORD, AND THE FILE IT WRITES, A NEW FILE AND AN EXTENSION OF MAIN.GO

```
// cmd/pong/record.go — create
package main
```

```
import (
        "bufio"
        "flag"
        "fmt"
        "os"
        "strings"
)

// A recording is a text file with one line per tick: the names
of the keys
// held on that tick, separated by spaces, or an empty line when
none was.

var (
        record = flag.String("record", "", "write the keys held
on every tick of this run to the file named")
        replay = flag.String("replay", "", "play the recording
in the file named, with no window, and print the final score")
)

// String names the held keys, in a fixed order.
func (k keys) String() string {
        var names []string
        for _, key := range []struct {
                held bool
                name string
        }{{k.w, "w"}, {k.s, "s"}, {k.up, "up"}, {k.down,
"down"}, {k.space, "space"}, {k.r, "r"}} {
                if key.held {
                        names = append(names, key.name)
                }
        }
        return strings.Join(names, " ")
}

// parseKeys reads one line of a recording back into a keys
value.
func parseKeys(line string) (keys, error) {
        var k keys
        for _, name := range strings.Fields(line) {
                switch name {
                case "w":
                        k.w = true
                case "s":
                        k.s = true
                case "up":
                        k.up = true
                case "down":
                        k.down = true
                case "space":
                        k.space = true
                case "r":
                        k.r = true
```

```
                default:
                        return k, fmt.Errorf("%q is not a key",
name)
                }
        }
        return k, nil
}

// writeRecording writes one line per tick to the file at path.
func writeRecording(path string, ticks []keys) error {
        f, err := os.Create(path)
        if err ≠ nil {
                return err
        }
        w := bufio.NewWriter(f)
        for _, k := range ticks {
                fmt.Fprintln(w, k)
        }
        if err := w.Flush(); err ≠ nil {
                f.Close()
                return err
        }
        return f.Close()
}

// readRecording reads the file at path back into one keys value
per tick.
func readRecording(path string) ([]keys, error) {
        f, err := os.Open(path)
        if err ≠ nil {
                return nil, err
        }
        defer f.Close()
        var ticks []keys
        sc := bufio.NewScanner(f)
        for sc.Scan() {
                k, err := parseKeys(sc.Text())
                if err ≠ nil {
                        return nil, fmt.Errorf("%s line %d: %v",
path, len(ticks)+1, err)
                }
                ticks = append(ticks, k)
        }
        return ticks, sc.Err()
}

// replayMatch steps a fresh match through every tick of the
recording at
// path, with no window, and prints the final score.
func replayMatch(path string) error {
        ticks, err := readRecording(path)
        if err ≠ nil {
                return err
        }
```

```
        m := newMatch(*seedFlag, *soloFlag)
        for _, k := range ticks {
                m.Step(k)
        }
        fmt.Printf("%s: %d ticks, score %d-%d, %s\n", path,
len(ticks), m.Score[0], m.Score[1], m.State)
        return nil
}


// cmd/pong/main.go — extend
// Game is Pong's window: the match, and everything drawn or
heard.
type Game struct {
        match      *Match
        court      *ebiten.Image
        sp         *sprites
        spin       int // ticks the ball has been moving
        font       *digits.Font
        face       *face
        served     bool // has the first serve happened? The
title shows until it has.
        hit, point *sound.Sound
        recorded   []keys // the keys held on every tick so far,
when recording
}

func (g *Game) Update() error {
        k := readKeys()
        if *record ≠ "" {
                g.recorded = append(g.recorded, k)
        }
        g.step(k)
        return nil
}

func main() {
        flag.Parse()
        if *replay ≠ "" {
                if err := replayMatch(*replay); err ≠ nil {
                        log.Fatal(err)
                }
                return
        }
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("Pong")
        ebiten.SetTPS(60)
        g, err := newGame()
        if err ≠ nil {
                log.Fatal(err)
        }
        if err := ebiten.RunGame(g); err ≠ nil {
                log.Fatal(err)
        }
```

```
        if *record ≠ "" {
                if err := writeRecording(*record, g.recorded);
err ≠ nil {
                        log.Fatal(err)
                }
                fmt.Printf("%s: %d ticks recorded\n", *record,
len(g.recorded))
        }
}


mkdir -p tapes
go vet ./...
go run ./cmd/pong -solo -record tapes/mine.txt
head -30 tapes/mine.txt
```

Play a few points against the follower and close the window, and the terminal prints `tapes/mine.txt:` and the number of ticks the window was open, one line of the file for each; the lines are empty until the first key and then read `space`, `w` and `s` as the keys went down. The window's `Update` gained one line: before the keys go to `step`, they go into a slice, one entry a tick, when a recording was asked for. The keys are recorded where they are read and nowhere else, so what is in the file is what the match was given. `String` writes a `keys` value as words and `parseKeys` reads the words back, refusing a word that is not a key with the line it was on, and the rest of the file is the two functions that read and write a file of lines.

`replayMatch` is the whole of a headless run: read the file, make a match from the same flags the window would, step it once per line, and print how it ended. No window is opened; the program returns before `ebiten.RunGame` is reached. The recording holds the keys and nothing else, so the seed and the mode have to be given again on the command line, the same as they were when it was made; the worked failure below is what happens when they are not.

▣ BUILD · STAGE 2 — A RECORDING REPLAYED WITH NO WINDOW

The three recordings below are the ones this volume's pictures of Pong were made from: a rally between two program paddles, one

serve with nobody at the keys, and a whole match in solo mode. Put them in `tapes/`.

ch07-rally.txt (600 ticks: two program paddles, the ball launched as in chapter 7's stage 2)

ch07-serve.txt (300 ticks: Space on tick 2, nothing else)

ch07-match.txt (7,456 ticks: a solo match, seed 1, to seven points)

```
go run ./cmd/pong -replay tapes/ch07-match.txt -solo
```

```
$ go run ./cmd/pong -replay tapes/ch07-match.txt -solo

tapes/ch07-match.txt: 7456 ticks, score 7-0, over
```

The match that took two minutes to play replays without opening a window, and it ends 7–0 to the left, over. The 7,456 lines of the file, plus the seed and solo mode on the command line, are enough to reproduce the match.

The window, sprites, sounds and display clock do not affect the score. Change a rule in `match.go` and this printed line may change, which is why the replay belongs in a test.

⚠ WORKED FAILURE — THE SAME RECORDING WITHOUT -SOLO

```
$ go run ./cmd/pong -replay tapes/ch07-match.txt

tapes/ch07-match.txt: 7456 ticks, score 5-2, serve
```

The same file, and a different match: 5–2, and not over. The keys in the file are the left player's, recorded against a right paddle that followed the ball; played against a right paddle that reads Up and Down, which nobody in the recording pressed, the right paddle never moves, the ball goes past it or does not on different ticks, and from the first point that differs every later line of the file is a key pressed for a match that is no longer happening. A recording replays a match only with the match's other inputs, the

seed and the mode, given exactly as they were; the file does not carry them, so the command line has to.

3. Checking a replay in a test

▣ BUILD · STAGE 3 — THE TEST

```
// cmd/pong/pong_test.go — create
package main

import (
        "testing"

        "gez/internal/shot"
)

// TestReplay plays the solo match recorded in tapes/ch07-
match.txt through
// a fresh match, with no window, and checks how it ended. go
test runs in
// the package's directory, and the recordings live at the
module's root.
func TestReplay(t *testing.T) {
        shot.AtRoot()
        ticks, err := readRecording("tapes/ch07-match.txt")
        if err ≠ nil {
                t.Fatal(err)
        }
        m := newMatch(1, true)
        for _, k := range ticks {
                m.Step(k)
        }
        if m.State ≠ Over || m.Score ≠ [2]int{7, 0} {
                t.Fatalf("after %d ticks: score %d-%d, %s; want
7-0, over", len(ticks), m.Score[0], m.Score[1], m.State)
        }
        t.Logf("%d ticks: score %d-%d, %s", len(ticks),
m.Score[0], m.Score[1], m.State)
}


go test ./cmd/pong -run TestReplay -v
```

```
$ go test ./cmd/pong -run TestReplay -v

=== RUN   TestReplay
```

```
    pong_test.go:25: 7456 ticks: score 7-0, over
--- PASS: TestReplay (0.00s)
PASS
ok      gez/cmd/pong    0.00s
```

A Go test is a function in a file ending in `_test.go`, named `Test` and something, taking a `*testing.T`; `go test` compiles the package with its test files and runs every such function `-run` selects, and `-v` prints each test's name and its `t.Logf` lines instead of only the verdict. The two times on the last lines are yours and differ from run to run; everything else is the same on every machine. The test is in package `main`, beside the game, so it can call `newMatch` and `readRecording` directly and needs nothing exported. It steps the recording through a fresh solo match with seed 1, the flags written as arguments, and fails with both scores if the match did not end 7–0 and over.

Now change a rule. Make the cap six instead of five, or the paddle two pixels taller, run the test again, and it fails, with the score the changed rules produce in its message; the recording's left player was aiming at a paddle that is no longer there. That is what the test is for: a change to the rules that was meant to leave the game alone is caught by a match that ends differently, and a change that was meant to alter the game gets a recording of its own. The `os.Chdir` at the top is because `go test` runs a test in the package's own directory, `cmd/pong`, while the recordings live at the module's root beside `assets/`.

⚙ TOOL — GO TEST

`go test ./cmd/pong` runs every test in the package and prints one line, `ok` or `FAIL`; `-run Name` selects tests whose names match; `-v` shows each test; `-count=1` makes it run again instead of printing a cached result. A test passes unless it calls `t.Fatal`, `t.Fatalf` or `t.Errorf`. The reference is `go help test` and `pkg.go.dev/testing`.

4. Writing a frame image

A frame is the window's `Draw` called once after the match has been stepped to the tick wanted, and the pixels it painted read back and written as a PNG. Ebitengine only calls `Draw` inside a running window, so the program that makes a picture has to open one, run the ticks, draw, read the pixels, and close it; the package below does exactly that and nothing else, and every picture in this volume came out of it.

▣ BUILD · STAGE 4 — THE PICTURES, TWO NEW FILES

```
// internal/shot/shot.go — create
// Package shot makes pictures of a game without anyone at the
keyboard: it
// opens a window, advances a game a given number of ticks by
calling a
// function once per tick, draws one picture, and writes it to a
PNG file.
// Every picture in this volume was made with it.
package shot

import (
        "fmt"
        "image"
        "image/png"
        "os"
        "path/filepath"

        "github.com/hajimehoshi/ebiten/v2"
)

// The pictures are the size every game in this volume draws.
const (
        ScreenW = 320
        ScreenH = 180
)

// Frame is one picture to make: Step is called once for each of
Ticks
// ticks, then Draw paints the picture, and the picture is
written to Path.
type Frame struct {
        Path  string
        Ticks int
        Step  func()
        Draw  func(screen *ebiten.Image)
}
```

```
// runner is the game the window runs: it works through the
frames in
// order, all of one frame's ticks in a single Update, then
takes the
// picture in the Draw that follows.
type runner struct {
        frames []Frame
        i      int  // the frame being made
        ready  bool // the frame's ticks have all been stepped
        err    error
}

func (r *runner) Update() error {
        if r.err ≠ nil || r.i ⩾ len(r.frames) {
                return ebiten.Termination
        }
        if r.ready {
                return nil // stepped; wait for Draw to take the
picture
        }
        f := r.frames[r.i]
        for t := 0; t < f.Ticks; t++ {
                f.Step()
        }
        r.ready = true
        return nil
}

func (r *runner) Draw(screen *ebiten.Image) {
        if !r.ready || r.i ⩾ len(r.frames) {
                return
        }
        f := r.frames[r.i]
        f.Draw(screen)
        if err := save(screen, f.Path); err ≠ nil {
                r.err = err
                return
        }
        fmt.Printf("%s: tick %d, %dx%d\n", f.Path, f.Ticks,
screen.Bounds().Dx(), screen.Bounds().Dy())
        r.i++
        r.ready = false
}

func (r *runner) Layout(outsideWidth, outsideHeight int) (int,
int) {
        return ScreenW, ScreenH
}

// save writes the picture screen holds to path as a PNG: every
pixel, four
// bytes each, exactly what was drawn.
func save(screen *ebiten.Image, path string) error {
```

```
        w, h := screen.Bounds().Dx(), screen.Bounds().Dy()
        img := image.NewRGBA(image.Rect(0, 0, w, h))
        screen.ReadPixels(img.Pix)
        if err := os.MkdirAll(filepath.Dir(path), 0o755); err ≠
nil {
                return err
        }
        f, err := os.Create(path)
        if err ≠ nil {
                return err
        }
        if err := png.Encode(f, img); err ≠ nil {
                f.Close()
                return err
        }
        return f.Close()
}

// Run makes every frame in turn in one window, then closes it.
Ebitengine
// runs one window per process, so a test makes all of its
pictures in one
// call.
func Run(frames []Frame) error {
        ebiten.SetWindowSize(960, 540)
        ebiten.SetWindowTitle("shot")
        ebiten.SetTPS(60)
        r := &runner{frames: frames}
        if err := ebiten.RunGame(r); err ≠ nil {
                return err
        }
        return r.err
}

// AtRoot moves the process to the module's root, where go.mod
is, so that
// a test reads assets/ and tapes/ the way the programs do. go
test starts a
// test in its package's directory; a second test in the same
process finds
// the process already there.
func AtRoot() {
        for i := 0; i < 4; i++ {
                if _, err := os.Stat("go.mod"); err == nil {
                        return
                }
                os.Chdir("..")
        }
}


// cmd/pong/shot_test.go — create
package main
```

```
import (
        "testing"

        "gez/internal/shot"
)

// newTestGame builds the window's game the way main does, from
the flags'
// defaults or the seed and mode given.
func newTestGame(t *testing.T, seed uint64, solo bool) *Game {
        *seedFlag, *soloFlag = seed, solo
        g, err := newGame()
        if err ≠ nil {
                t.Fatal(err)
        }
        return g
}

// play returns a step function that feeds the game one tick of
the
// recording at a time, and nothing held once the recording runs
out.
func play(t *testing.T, g *Game, path string) func() {
        ticks, err := readRecording(path)
        if err ≠ nil {
                t.Fatal(err)
        }
        i := 0
        return func() {
                var k keys
                if i < len(ticks) {
                        k = ticks[i]
                }
                i++
                g.step(k)
        }
}

// launch puts the ball in flight the way chapter 7's stage 2
does, with no
// serve: flying right and a little down.
func launch(g *Game) {
        g.match.State = Play
        g.match.Ball.VX, g.match.Ball.VY = 2, 1
}

// firstEvent steps a fresh match through a recording and
returns the tick
// on which the event first happens, or -1.
func firstEvent(t *testing.T, m *Match, path string, want event)
int {
        ticks, err := readRecording(path)
        if err ≠ nil {
                t.Fatal(err)
```

```
        }
        for i, k := range ticks {
                if m.Step(k) == want {
                        return i + 1
                }
        }
        return -1
}

// TestCh07Frames makes chapter 7's pictures.
func TestCh07Frames(t *testing.T) {
        shot.AtRoot()
        // Stage 1: the paddles under the keys, W held twenty
ticks and Down thirty.
        paddles := newTestGame(t, 1, false)
        tick := 0
        movePaddles := func() {
                tick++
                paddles.step(keys{w: tick ⩽ 20, down: tick ⩽
30})
        }
        // Stage 2: the launched ball, after its first bounce
off the bottom.
        bounce := newTestGame(t, 1, false)
        launch(bounce)
        // Stage 3: the rally between two program paddles, six
ticks after the
        // first return.
        rallyTick := firstEvent(t, launchedMatch(), "tapes/ch07-
rally.txt", hit) + 6
        t.Logf("first return in tapes/ch07-rally.txt: tick %d",
rallyTick-6)
        returned := newTestGame(t, 1, false)
        launch(returned)
        // Stage 6: the title before the first serve, and the
band after the
        // first point of the serve recording.
        title := newTestGame(t, 1, false)
        pointTick := firstEvent(t, newMatch(1, false),
"tapes/ch07-serve.txt", point)
        t.Logf("first point in tapes/ch07-serve.txt: tick %d",
pointTick)
        served := newTestGame(t, 1, false)
        // Stage 7: the solo match, mid-rally and over.
        solo := newTestGame(t, 1, true)
        over := newTestGame(t, 1, true)
        matchTicks, err := readRecording("tapes/ch07-match.txt")
        if err ≠ nil {
                t.Fatal(err)
        }
        err = shot.Run([]shot.Frame{
                {Path: "assets/frames/ch07-paddles.png", Ticks:
30, Step: movePaddles, Draw: paddles.drawMatch},
                {Path: "assets/frames/ch07-bounce.png", Ticks:
```

```
70, Step: func() { bounce.step(keys{}) }, Draw:
bounce.drawMatch},
                {Path: "assets/frames/ch07-return.png", Ticks:
rallyTick, Step: play(t, returned, "tapes/ch07-rally.txt"),
Draw: returned.drawMatch},
                {Path: "assets/frames/ch07-title.png", Ticks: 0,
Step: func() {}, Draw: title.Draw},
                {Path: "assets/frames/ch07-point.png", Ticks:
pointTick + 1, Step: play(t, served, "tapes/ch07-serve.txt"),
Draw: served.Draw},
                {Path: "assets/frames/ch07-solo.png", Ticks:
600, Step: play(t, solo, "tapes/ch07-match.txt"), Draw:
solo.Draw},
                {Path: "assets/frames/ch07-over.png", Ticks:
len(matchTicks), Step: play(t, over, "tapes/ch07-match.txt"),
Draw: over.Draw},
        })
        if err ≠ nil {
                t.Fatal(err)
        }
}

// launchedMatch is a fresh match with the ball launched as
stage 2 does.
func launchedMatch() *Match {
        m := newMatch(1, false)
        m.State = Play
        m.Ball.VX, m.Ball.VY = 2, 1
        return m
}


go vet ./...
go test ./cmd/pong -run TestCh07Frames -v
```

```
$ go test ./cmd/pong -run TestCh07Frames -v

=== RUN   TestCh07Frames
    shot_test.go:76: first return in tapes/ch07-rally.txt: tick
74
    shot_test.go:83: first point in tapes/ch07-serve.txt: tick
87
assets/frames/ch07-paddles.png: tick 30, 320x180
assets/frames/ch07-bounce.png: tick 70, 320x180
assets/frames/ch07-return.png: tick 80, 320x180
assets/frames/ch07-title.png: tick 0, 320x180
assets/frames/ch07-point.png: tick 88, 320x180
assets/frames/ch07-solo.png: tick 600, 320x180
assets/frames/ch07-over.png: tick 7456, 320x180
--- PASS: TestCh07Frames (0.00s)
PASS
ok      gez/cmd/pong    0.00s
```

A window opens for a moment and closes, and `assets/frames/` holds seven pictures: the seven on chapter 7's page, made from the recordings above and the ticks the test names, in the order the page shows them. Open `ch07-over.png` beside the page's last picture; they are the same picture, and on most machines the same bytes, the soft edges of the band's words being the one place two graphics cards may differ by a shade.

`shot.Run` is a game in Ebitengine's sense with three methods, and it makes every frame of a list in one window, because Ebitengine runs one window per process. Its `Update` runs all of a frame's ticks in one call, by calling the frame's `Step` function that many times, and then holds still; the `Draw` that follows calls the frame's `Draw` function, reads the screen's pixels into a standard-library image with `ReadPixels`, encodes it as a PNG, and moves to the next frame. Stepping every tick inside one `Update` is what makes the picture a picture of a tick and not of a moment: the display's rate has nothing to say about which tick gets drawn.

The test file is the list. For each picture it builds the window's game the way `main` does, from the flags, and hands `shot.Run` a step function and a draw function. The step function is what stands in for the keyboard: `play` feeds the game one line of a recording a tick, and the paddles picture's function holds W and Down for a set number of ticks, since that picture needed no recording. The draw function is whichever of the window's drawing methods the picture wanted: `drawMatch` alone for the pictures of chapter 7's first stages, which had no score yet, and `Draw` for the rest. Two pictures are placed by an event rather than a tick, the first return and the first point; `firstEvent` steps a spare match through the recording to find the tick, and the test prints it, which is where chapter 7's 74 and 87 came from.

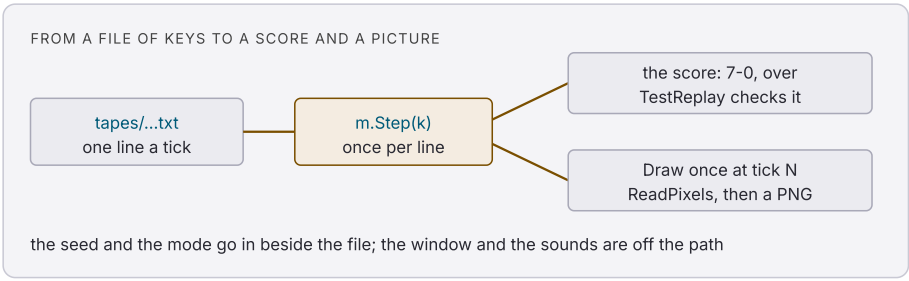


Figure 10.1 — a recording stepped through the rules, and the two things made from it: the final score a test can check, and any tick's picture.

5. Keeping tests on the rules

A recording replays because the match has no other input. The seed is fixed, the keys come from the file, and nothing in `match.go` reads a clock, a draw count, the mouse or the window.

The spin counter and the sounds live outside the rules, so they cannot change a score. A test needs a recording and an expected score. A frame image needs a recording, a tick and one call to `Draw`.

6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Record the keys of a match, one line a tick, where they are read, and say what a recording does and does not contain.
- Replay a recording with no window and get the score the match ended with, and explain from the failure box why the flags have to match.
- Write a Go test in package `main` that plays a recording and fails with the scores when a rule change alters the match.
- Make a picture of any tick of any of the three games with `shot.Run`, and say why the display's rate cannot change which tick is drawn.

- Name the one property of the rules that makes all three possible.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — your own match, checked. Record a solo match of your own to `tapes/mine.txt`, replay it, and add a test that plays it and checks the score you got.**

`go run ./cmd/pong -solo -record tapes/mine.txt`, play to seven, close the window; `go run ./cmd/pong -replay tapes/mine.txt -solo` prints your score; a second test in `pong_test.go` is `TestReplay` with the file name and the scores changed. Run both tests with `-run Replay`.

▼ **Exercise 2 — a rule change, caught. Make the paddles move three pixels a tick and run `TestReplay`.**

`paddleSpeed = 3` in `match.go`, and the test fails: the left paddle arrives at every row sooner than the recording's player expected, the returns come off different parts of the paddle, and the match ends 0–7, which the message prints. Put the speed back and it passes again.

▼ **Exercise 3 — a picture of Snake. Write `cmd/snake/shot_test.go` with one frame: the snake left to itself for two hundred ticks.**

`shot.AtRoot()`, a game from `newGame()`, and one `shot.Frame` with `Ticks: 200`, `Step: func() { g.step(keys{}) }` and `Draw: g.Draw`, written to `assets/frames/snake-200.png`. The picture is the game-

over band over a snake that ran into the right border on its twentieth move, at tick 160, and waited.