© Pavel Chagochkin / 123RF.com

Autorennen mit Go für den Desktop

Rasend schnell

Der schnellste Weg auf der Rennstrecke führt entlang der Ideallinie. Statt auf dem Nürburgring trainiert Mike Schilli seine Reflexe sicherheitshalber mit einer in Go geschriebenen Desktop-Applikation. Michael Schilli

Vor einigen Jahren durfte ich mal bei einem Sicherheitstraining die physikalischen Grenzen meines Honda Fits ausloten. Kurz darauf begann ich, mich für Autorennen zu interessieren. Zuedem ist es unter Angestellen im Silicon Valley ein nicht unübliches Hobby, seine aufgemotzten Privatboliden auf Rennstrecken wie Laguna Seca in Kalifornien von der Leine zu lassen, wohl vor allem deshalb, weil in Amerika auf den Freeways das strikte Tempolimit von meist 65 Meilen pro Stunde (104 km/h) gilt.

Beim Studieren der Thematik nahm ich überrascht zur Kenntnis, dass es keineswegs nur darum geht, den Bleifuß immer schön auf dem Gaspedal zu lassen. Wer

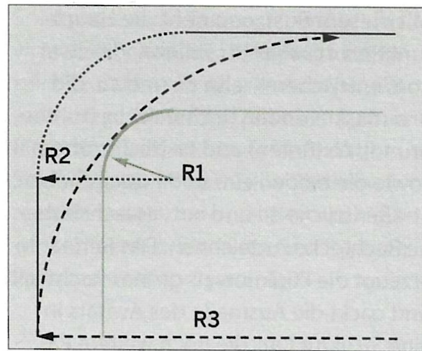
Rennbahnrekorde brechen will, muss exakt nach physikalischen Formeln durch die Kurven brausen und immer die Ideallinie finden, um so in jeder Runde kumulativ Sekunden einzusparen. Die physikalischen Grundlagen der Rennbahn erklärt das Standardwerk „Going Faster" von Carl Lopez. Es führt detailliert aus, wie schnell man in eine Kurve fahren kann, ohne dass das Auto zu schleudern beginnt, und in welchem Winkel und zu welchem Zeitpunkt der Rennfahrer das Lenkrad einschlagen muss, damit während der Kurvenfahrt möglichst wenig Zeit verstreicht.

Rasen lernen

Dabei ist die Ideallinie durch eine Kurve nicht der kürzeste Weg, der an der Innenseite entlangführt. Vielmehr geht es darum, auf einer Kreisbahn in einem möglichst großen Radius durch die Kurve zu fahren **1**. Vor der gezeigten 90-Grad-Rechtskurve fährt ein Fahrer vom Schlage eines Max Verstappen darum zunächst an den linken Fahrbahnrand und zieht danach scharf nach rechts zum inneren Rand. Auf diese Weise schrammt der Rennwagen nur knapp an der Innenseite der Kurve vorbei, um kurz darauf auf der horizontalen Strecke nach der Kurve wiederum auf die linke Fahrbahnseite zu ziehen. Damit ist der Radius, den das Auto fährt, deutlich größer als der der Kurve, und es kann viel schneller durch sie hindurch fahren, ohne dass die Reifen die Bodenhaftung verlieren und das Fahrzeug ins Schleudern kommt.

Welt aus Geometrie

Die Abbildung **2** zeigt die Simulation einer Kurvenfahrt als in Go geschriebenes Desktop-Spiel mit Rennanimation. Der als ein grünes Quadrat dargestellte Rennwagen flitzt auf die Kurve zu. Der Spieler muss das Fahrzeug mit den Tasten H und L nach links und rechts steuern, damit es bei dem Höllentempo nicht aneckt, sondern nach der Kurvenausfahrt wohlbehalten oben am rechten Fensterrand ankommt. Die Stoppuhr neben den beiden Schaltern läuft während der Animation immer mit und zeigt die verstrichene Rundenzeit in Sekunden mit Hundertsteln an.



1 Der schnellste Weg durch die Kurve nutzt den größtmöglichsten Radius.

Mit ein wenig Vorwissen aus Geometrie und Videospieltechnik klopft sich so ein simples 2D-Spiel schnell mit Go und dem Fyne-Framework zusammen. Das Programm durchläuft dazu eine Anzahl von Frames pro Sekunde, in denen es jeweils die aktuelle Lage der Spielfiguren errechnet und die Position dann in der Grafik auffrischt. Gleichzeitig fängt es User-Eingaben wie Tastendrücke oder Mausklicks ab und lässt diese Ereignisse in die Berechnungen einfließen, indem es zum Beispiel die Lenkung verstellt.

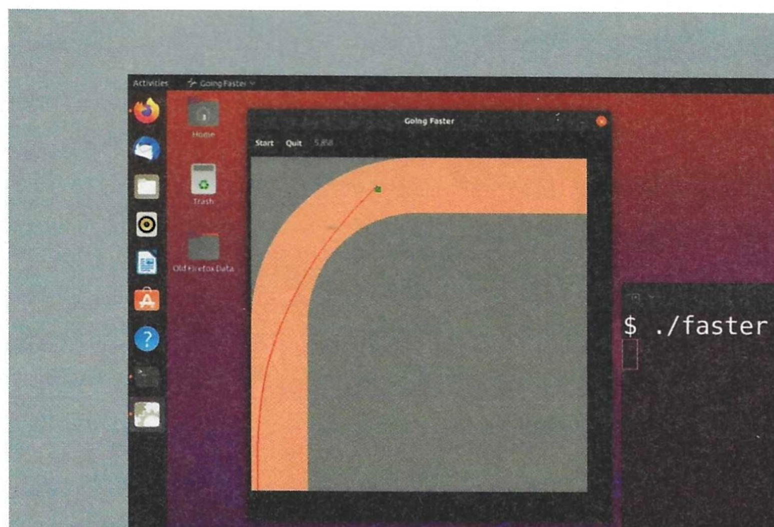
Am Anfang war der Kreis

Wie schreibt sich solch ein Spiel in Go? Zu allererst gilt es, die „Welt" des Spiels zu zeichnen. Und zwar so, dass das Programm später bei jedem durchlaufenen Video-Frame blitzschnell errechnen kann, ob die Spielfigur sich noch auf der Straße befindet oder schon die Konturen der Kurve verlassen hat und das Rennauto in den Büschen liegt.

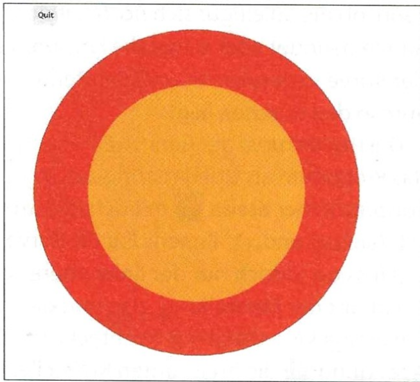
Die Rechtskurve der Rennstrecke malt das Programm als Überlappung zweier konzentrischer Kreise **3** mit den Radien „r2" (außen) und „r1" (innen). Für die Kurve interessiert jedoch nur der linke obere Quadrant der Darstellung, also maskieren einige klug platzierte Rechtecke in Abbildung **4** die irrelevanten Kreisteile. Die blauen, grauen und orangefarbenen Flächen verschwinden später in der Darstellung, und die beiden lachsfarbenen Rechtecke bestimmen die Einfahrt in die Kurve und deren Ausfahrt. Listing 1 stellt diese „Welt", wie es im Spielejargon heißt, mithilfe von `Circle()` und `Rectangle()`-Objekten auf einem Canvas-Objekt des Fyne-Frameworks dar.

Rundes und Eckiges

Da das Fyne-Framework etwas unhandliche Methoden zum Platzieren von Kreisen und Rechtecken hat, definieren die Funktionen `drawCircle()` und `drawRectangle()` ab den Zeilen 26 und 34 praktischere Schnittstellen. Die Kreisfunktion nimmt als ersten Parameter die Füllfarbe entgegen, gefolgt vom Mittelpunkt im x/y-Format und einem Radius `r`. Fyne platziert Circle-Objekte anhand der linken oberen Ecke eines imaginären Quadrats, das den Kreis umschließt, und bugsiert es mittels `Move()` dorthin, um es dann



2 Das Rennstreckenspiel in Aktion auf dem Desktop.



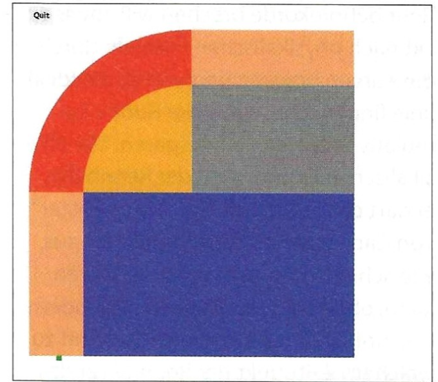
3 Zwei konzentrische Kreise bestimmen die 90-Grad-Kurve ...

mit `Resize()` auf die geforderte Größe aufzublasen. Um einen Kreis mit Radius `r` zu erhalten, weist das umschließende Quadrat eine Seitenlänge von 2*r auf. Die Schnittstelle für Rechtecke in Fyne ist logischer, und `drawRectangle()` kombiniert nur die Aufrufe der Methoden `Move()` und `Resize()`, damit die aufrufende Funktion alles erledigen kann.

Mit diesem Rüstzeug geht die Hauptfunktion `drawWorld()` daran, die zwei konzentrischen Kreise `ci` und `co`, die drei maskierenden Rechtecke `bg` (Hintergrund), `mb` (unten) und `mr` (rechts oben) sowie die beiden ein- und ausleitenden Straßenstücke `in` und `out` als lachsfarbene Rechtecke zu zeichnen. Das Rennauto erzeugt die Funktion als grünes Rechteck und packt die Ausmaße des Avatars in eine Struktur `Car`, die noch weitere Parameter wie Geschwindigkeit und Startposition enthält und später in Listing 2 definiert wird. Alle so weit erzeugten Grafik-Objekte packt Zeile 23 in einen Container, den `drawWorld()` mitsamt dem `Car`-Objekt an den Aufrufer zurückgibt, auf dass dieser sie der Grafik-Engine des Frameworks zur Verwaltung zuführe.

Auto als Struktur

Listing 2 zeigt das Hauptprogramm `main`, das ein Applikationsfenster mit fester Größe aufzieht und mit `drawWorld()` aus Listing 1 die Rennstrecke samt Wagen



4 ... und mit einigen Rechtecken als Masken entsteht die Rennbahn.

hineinzeichnet. Die Struktur vom Typ `Car` ab Zeile 10 definiert in `Ava` (wie in „Avatar"), wie das Auto in der Grafik-Welt dargestellt wird, nämlich als grünes Rechteck. Weiter schleppt die Struktur noch die Anfangskoordinaten des Fahrzeugs sowie die aktuelle Geschwindigkeit, die Fahrtrichtung und den Einschlagwinkel der Räder mit. In `Timer` enthält die Struk-

Listing 1: world.go

```
01 package main
02 import (
03   "fyne.io/fyne/v2"
04   "fyne.io/fyne/v2/canvas"
05   "fyne.io/fyne/v2/container"
06   col "golang.org/x/image/colornames"
07   "image/color"
08 )
```

```
20   car.Ava.Resize(fyne.NewSize(10, 10))
21   car.Ava.Move(car.StartPos)
22   objects := []fyne.CanvasObject{bg, co, ci, mb,
   mr, in, out, car.Ava}
23   play := container.NewWithoutLayout(objects...)
24   return play, car
25 }
26 func drawCircle(co color.RGBA, x, y, r float32)
   *canvas.Circle {
```

tur außerdem einen Zeitmesser, der die bis dato auf der Rennstrecke verstrichene Zeit bereithält.

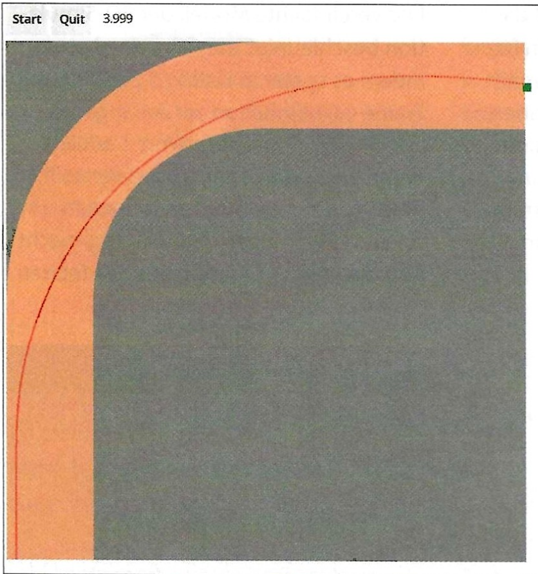
Kurven fahren

Schlägt der Fahrer des Rennwagens mithilfe des Lenkrads die Vorderräder ein, bewegt er sich auf einer Kreisbahn. Im Spiel steuern die Tasten „H" und „L" das virtuelle Lenkrad des Fahrzeugs ein Fitzelchen nach links oder rechts (gemäß „vi"-Konvention). Die Tastatureingaben fängt der Callback zur Fyne-Funktion `SetOnTypedKey()` ab Zeile 55 ab und reagiert darauf, indem er den Lenkradwinkel `TurnAng` sowie die Geschwindigkeit entsprechend verstellt.

Das vereinfachte Modell der 2D-Simulation beschleunigt das Fahrzeug konstant, indem es später in Listing 5 pro Video-Frame 0,01 Einheiten zur Anfangsgeschwindigkeit mit dem Wert 1 addiert. Wenn der Fahrer das Lenkrad verstellt, sinkt die Geschwindigkeit hingegen um 0,1 Einheiten. Jeder Lenkvorgang macht also die Beschleunigung aus den letzten

Listing 2: `faster.go`

```
01 package main
02 import (
03   "fyne.io/fyne/v2"
04   "fyne.io/fyne/v2/app"
05   "fyne.io/fyne/v2/canvas"
06   "fyne.io/fyne/v2/container"
07   "fyne.io/fyne/v2/widget"
08   "os"
09 )
10 type Car struct {
11   Ava      *canvas.Rectangle
12   StartPos fyne.Position
13   DriveAng float32
14   TurnAng  float32
15   Timer    Clock
16   Speed    float32
17 }
18 func main() {
19   a := app.New()
20   w := a.NewWindow("Going Faster")
21   w.Resize(fyne.NewSize(650, 700))
22   w.SetFixedSize(true)
23   var r1, r2 float32
24   r1 = 200
25   r2 = 300
26   play, car := drawWorld(r1, r2)
27   tracker := NewTracker()
28   tracker.StartPos = car.StartPos
29   tracker.R1 = r1
30   tracker.R2 = r2
31   ctrl := animation(&car, tracker)
32   quit := widget.NewButton("Quit",
33     func() { os.Exit(0) })
34   start := widget.NewButton("Start",
35     func() {
36       ctrl <- 1
37     })
38   car.Timer = NewClock()
39   display := widget.NewLabel("")
40   go func() {
41     for {
42       select {
43       case readout := <-car.Timer.UpdateCh:
44         display.SetText(readout)
45         display.Refresh()
46       }
47     }
48   }()
49
50 car.Timer.Reset()
51   car.Timer.Update()
52   buttons := container.NewHBox(start, quit,
display)
53   con := container.NewVBox(buttons, play)
54   w.SetContent(con)
55   w.Canvas().SetOnTypedKey(
56     func(ev *fyne.KeyEvent) {
57       key := string(ev.Name)
58       switch key {
59       case "L":
60         car.TurnAng += .001
61         car.Speed -= .1
62       case "H":
63         car.TurnAng -= .001
64         car.Speed -= .1
65       case "Q":
66         os.Exit(0)
67       case "S":
68         ctrl <- 1
69       }
70     })
71   w.ShowAndRun()
72 }
```



5 **Neue Bestzeit in der Kurve mit 3,999 Sekunden.**

10 Frames zunichte. Insofern ist es günstig, wenig hektisch zu lenken, genau wie auf einer echten Rennstrecke auch.

Listing 3: `timer.go`

```
01 package main
02 import (
03   "fmt"
04   "time"
05 )
06 type Clock struct {
07   Start    time.Time
08   UpdateCh chan string
09 }
10 func NewClock() Clock {
11   return Clock{
12     Start:    time.Now(),
13     UpdateCh: make(chan
   string),
14   }
15 }
16 func (t *Clock) Reset() {
17   t.Start = time.Now()
18 }
19 func (t Clock) Update() {
20   dur := time.Since(t.Start)
21   t.UpdateCh <- fmt.Sprintf
   ("%.03f", dur.Seconds())
22 }
```

Wie sich das Fahrzeug danach im Bild weiterbewegt, hängt von zwei Werten ab: In welchem Winkel das Fahrzeug sich bereits bewegt, gibt `DriveAng` in der `Car`-Struktur in Zeile 13 an. Dieser Wert beschreibt in Radianten-Graden, in welcher Himmelsrichtung das Fahrzeug unterwegs ist. Und wie weit das Lenkrad momentan eingeschlagen ist, steht in `TurnAng`, als Summe der durch den User initiierten Lenkradbewegungen. Die Summe beider Werte bestimmt anschließend in der Animation von Listing 5 die neue Richtung des Fahrzeugs.

Eine absolut realistische Simulation müsste hier allerdings die in Autos verbauten Vorder- und Hinterachsen einkalkulieren, von denen sich (normalerweise) nur die Räder der Vorderachse verstellen lassen. Statt dieses sogenannte Ackermann-Steering zu verwenden, begnügt sich das einfache Spiel jedoch damit, die beiden Winkel der Fahrtrichtung und der Lenkradeinstellung zu addieren und die nächste im Spielparcours angefahrene Koordinate später mit dem Sinus- beziehungsweise Cosinussatz aus der Schulmathematik zu errechnen **5**. Deshalb stellt das Spiel das Fahrzeug als kleines Quadrat dar, ein realistischeres Rechteck sähe komisch aus, wenn es unkorrigiert seitwärts aus der Kurve herausfahren würde.

Wachsames Auge

Ob das Auto noch auf der Rennstrecke fährt oder bereits wie in Abbildung **5** im Ziel ist, oder vielleicht auf halber Strecke von der Fahrbahn abgekommen ist, bestimmt das Objekt vom Typ `Tracker` ab Zeile 27 in Listing 2. Es kennt die Ausmaße des Parcours und kann später in `animation()` in Listing 5 errechnen, ob die gegenwärtige Koordinate noch auf der Straße liegt oder daneben. Die Implementierung dieser geometrischen Funktionen findet sich in Listing 4.

Ein Objekt vom Typ `Clock` misst die Rundenzeit ab Zeile 38 in Listing 2 und zeigt sie in Sekunden und Hundersteln in einem Fyne-Widget vom Typ `Label` an. Die GUI erhält bei jedem Frame des Videospiels die aktuell anzuzeigende Zeit

Listing 4: `tracker.go`

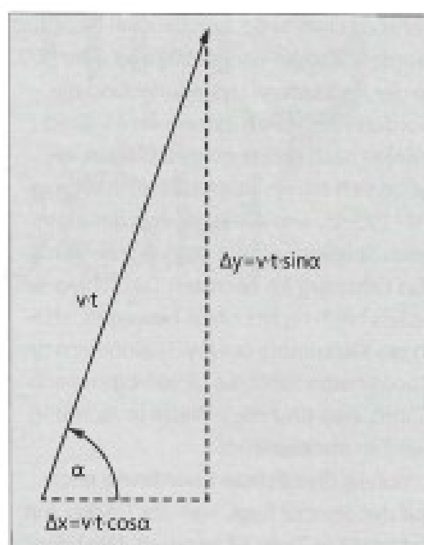
```
01 package main
02 import (
03   "fyne.io/fyne/v2"
04   "math"
05 )
06 type Tracker struct {
07   StartPos fyne.Position
08   R1, R2   float32
09 }
10 func NewTracker() Tracker {
11   tracker := Tracker{}
12   return tracker
13 }
14 func (t Tracker) OnRoad(x, y
   float32) bool {
15   // before curve
16   if y > t.R2 && x < t.R2-t.
   R1 && x > 0 {
17     return true
18   }
19   // in curve
20   if y <= t.R2 && x <= t.R2 {
21     h := t.R2 - y
22     w := t.R2 - x
23     r := float32(math.
   Sqrt(float64(h*h + w*w)))
24     if r <= t.R2 && r >= t.R1
   {
25       return true
26     }
27     return false
28   }
29   // after curve
30   if y <= t.R2-t.R1 && x >=
   t.R2 && x < 2*t.R2 && y > 0 {
31     return true
32   }
33   return false
34 }
```

über den Channel `UpdateCh`. Diesen liest die nebenläufig ausgeführte Go-Routine ab Zeile 40 stetig aus und frischt die Zeit im Stoppuhr-Widget auf, das so stetig vor sich hinrattert. Die Implementierung der Stoppuhr findet sich in Listing 3.

Zeit messen

Der Channel `ctrl`, den die Funktion `animate()` erzeugt hat und in Zeile 31 von Listing 2 an das Hauptprogramm zurückgibt, bestimmt, wann der Rennwagen startet und beschleunigt. Dies geschieht in Zeile 36 als Reaktion auf das Klicken des „Start"-Buttons mit der Maus oder auf das Drücken der Taste „S" in Zeile 68. Beide Male schiebt der Code eine „1" in den Channel, die Listing 5 später aufschnappt und das Auto anschieben wird.

Listing 3 definiert den Zeitmesser, dessen `Reset()`-Funktion die verstrichene Zeit auf der Rennstrecke auf null setzt, indem sie in `Start` die aktuelle Uhrzeit ablegt. Bei jedem Aufruf der Funktion `Update()` ab Zeile 19 bestimmt `Since()` die



6 Die geänderten x- und y-Koordinaten in Fahrtrichtung des Rennwagens.

Differenz aus der aktuellen Uhrzeit und der Startzeit. Die Funktion formatiert denWert als Sekunden und Hunderstelsekunden, um ihn so in den Channel `UpdateCh` zu schicken, wo ihn das Hauptprogramm im Label-Widget der GUI anzeigt.

Ob das Auto noch auf der Strecke fährt oder davon abgekommen ist, bestimmt Listing 4 mit dem `Tracker`-Objekt. In seinem Konstruktor speichert es die Radien „R1" und „R2" der 90-Grad-Kurve des Spiels und bestimmt daraus die Koordinaten des gültigen Spielraums. Die Funktion `OnRoad()` ab Zeile 14 berechnet zu einer vorgegebenen x/y-Koordinate, ob sie auf der Fahrbahn liegt oder daneben. Dazu teilt sie den Parcours in drei Bereiche auf: vor der Kurve, in der Kurve, und die Ausfahrt ins Ziel. Gibt `OnRoad()` einen wahren Wert zurück, ist der Wagen auf der Strecke, während ein falscher Wert signalisiert, dass er in den Büschen oder im Ziel ist und das Spiel deswegen endet.

Listing 5 schließlich steuert die Dynamik des Spielablaufs, vom Start des Reigens ab Zeile 13, nachdem das Kommando dazu auf dem Steuerkanal `ctrl` angekommen ist, bis zum Update in jedem einzelnen Spielframe, von denen 100 pro Sekunde durchrauschen.

PROBELESEN OHNE RISIKO
TESTEN SIE JETZT 3 AUSGABEN FÜR 19 €
OHNE DVD 15 €
linuxUSER
MARKE EIGENBAU
amazon
5 EURO-GUTSCHEIN
Abo-Vorteile
33% Rabatt
- Günstiger als am Kiosk
- Versandkostenfrei bequem per Post
- Pünktlich und aktuell
- Keine Ausgabe verpassen
SICHERN SIE SICH JETZT IHR GESCHENK!
EIN AMAZON-GUTSCHEIN ÜBER 5,00 €
- Telefon: 0911 / 993 990 98 - E-Mail: computec@dpv.de
Einfach bequem online bestellen: shop.linuxuser.de

Den Ablauf dieser Frames steuert der Timer in Zeile 29 von Listing 5, der genau 10 Millisekunden wartet, bevor der Code ab Zeile 30 zu laufen beginnt. Die Anweisungen frischen die Zeitanzeige auf und lesen die aktuelle Position des Rennwagens als x und y aus. Zeile 34 erhöht mit jedem Frame die Geschwindigkeit Speed um 0,01 (von einem Anfangswert von 1) und errechnet in den Zeilen 35 und 36 mit dem Sinus- beziehungsweise Cosinussatz die nächste Koordinate als x/y-Wert, entsprechend der Geometrie in Abbildung 6.

Fährt das Fahrzeug zum Beispiel Richtung Norden, also im Winkel 90 Grad oder Pi/2 in der Radianten-Darstellung, und die Vorderräder wären extrem im 45-Grad-Winkel nach rechts eingeschlagen, ergäbe sich ein resultierender Winkel von Pi/4 (90-45, also 45 Grad). Aus der aktuellen Spielkoordinate (x, y) würde sich das Fahrzeug im nächsten Taktschlag des Spiels nach rechts oben bewegen, also in die Koordinate (x+1, y-1) einfahren (y-Koordinaten zählt die UI von oben nach unten, also sind die y-Werte in Richtung Norden abnehmend).

Sofern die nächste Koordinate noch auf der Strecke liegt, was der Tracker mit `OnRoad()` in Zeile 37 feststellt, fährt der Auto-Avatar in Zeile 38 mit `Move()` dorthin. Falls nicht, ist das Auto im Ziel oder liegt im Graben, und `return` in Zeile 40 beendet die sonst endlos weiterlaufende `for`-Schleife. Fertig ist das Spiel.

Installation

Mit allen Listings in einem Verzeichnis führt Listing 6 zu einem ausführbaren Binary, das `faster` heißt. Es ist gar nicht so einfach, den Wagen nach dem Start und der anfänglichen Beschleunigung auf der Strecke zu halten. *(uba)*

Der Autor

Michael Schilli arbeitet als Software Engineer in der San Francisco Bay Area in Kalifornien. In seiner Kolumne forscht er seit 1997 jeden Monat nach praktischen Anwendungen verschiedener Programmiersprachen. Unter mschilli@perlmeister.com beantwortet er gern Ihre Fragen.

Dateien zum Artikel herunterladen unter

www.lm-online.de/dl/48693

Listing 6: Binary erzeugen

```
go mod init faster
got mod tidy
go build
```

Weitere Infos und interessante Links

www.lm-online.de/qr/48693

Listing 5: animate.go

```
01 package main
02 import (
03   "fyne.io/fyne/v2"
04   "math"
05   "time"
06 )
07 func animation(car *Car, tracker Tracker) chan
   int {
08   ctrl := make(chan int)
09   go func() {
10     for {
11       select {
12       case <-ctrl:
13         car.TurnAng = 0
14         car.DriveAng = 0
15         car.Ava.Move(car.StartPos)
16         car.Speed = 1
17         car.Timer.Reset()
18         car.Timer.Update()
19         run(car, ctrl, tracker)
20       }
21     }
22   }()
23   return ctrl
24 }
25 func run(car *Car, ctrl chan int, tracker
   Tracker) {
26   for {
27     select {
28     case <-ctrl:
29     case <-time.After(time.Duration(10) * time.
   Millisecond):
30       car.Timer.Update()
31       x := car.Ava.Position().X
32       y := car.Ava.Position().Y
33       car.DriveAng += car.TurnAng
34       car.Speed += 0.01
35       x += car.Speed * float32(math.
   Sin(float64(car.DriveAng)))
36       y -= car.Speed * float32(math.
   Cos(float64(car.DriveAng)))
37       if tracker.OnRoad(x, y) {
38         car.Ava.Move(fyne.NewPos(x, y))
39       } else {
40         return
41       }
42     }
43   }
44 }
```