Mensch ärgere Dich nicht! – in R

File: mad.rev, in: /home/wiwi/pwolf/lehre/simstudy

15. Dezember 2010

Inhalt

1 Einleitung **3**

2 Grundsätzliche Überlegungen **3**

3 Problemanalyse. **4**

4 Erste Umsetzungsgedanken **4**

4.1 Spielbrettdarstellung 5

4.2 Datenstrukturüberlegungen 6

4.3 Fertige Darstellung des Spielbrettes 7

5 Spiel mit einem Spieler und einem Kegel **10**

5.1 Abbildung des Kegels 11

5.2 Setzen des Kegels 11

5.3 Bedingungsüberprüfungen 12

5.4 Das Würfel-Werfen und Restarbeiten 12

5.5 Die Konstruktion einer Spielfunktion 13

6 Spiel mit einem Spieler und mehreren Kegeln **13**

6.1 Grobstruktur des Spiels mit einem Spieler 13

6.2 `melde()` 15

6.3 Initialisierung des Spiels mit einem Spieler 16

6.4 Handlungsbedingungen 16

6.5 Kegelbewegungen 17

6.6 Auswahlfragen 18

6.7 Eine Funktion zum *Mensch ärgere Dich nicht!*-Spielen 18

6.8 Modellkritik 19

6.9 Eine kleine Auswertung 19

7 *Mensch ärgere Dich nicht!* für mehrere Spieler **22**
7.1 Vorüberlegungen 22
7.2 Die Struktur des Spiels 22
7.3 Die Initialisierung relevanter Größen 24
7.4 Die Bestimmung des nächsten Spielers 25
7.5 Heraussetzen neuer Kegel 25
7.6 Abwicklung eines normalen Zuges 26
7.7 Bedingungen 28
7.8 Aktionen 28
7.9 Das Schlagen anderer Kegel 28
7.10 Testerei 29
7.11 Eine einfache Spielfunktion 32

8 Simulations-Experimente mit dem Spiel **33**
8.1 Simulationen zur Ermittlungen von Spieldauern 33
8.2 Experimente zur Visualisierung von Strähnen 35
8.3 Fazit zur Simulation 38

9 Zugabe: Ein Spiel zum Mitspielen **38**

10 Rückblick **39**

A Anhang **40**
A.1 Wartezeiten 40
A.2 Tcl/Tk-Update 41
A.3 Zusammenstellung der Funktionen 41
A.4 Eine Simulation mit Haltepunkten 41
A.5 Alternative Ideen zur Zustandsberechnung 42
A.6 Indizes 43

1 Einleitung

Gegenstand dieses Papiers ist die Implementation des Spiels *Mensch ärgere Dich nicht!* in R. Das Spiel hat seine Ursprünge in dem indischen Spiel Pachisi. Im Jahre 1910 hat Josef Friedrich Schmidt *Mensch ärgere Dich nicht!* auf den Markt gebracht. Durch eine geschickte Geschenkaktion von 3000 Exemplaren an Lazarette im ersten Weltkrieg wurde das Spiel bekannt und ist seitdem ein Dauerbrenner.[1]

Mit der Implementation des Spiels sollen mehrere Zwecke verfolgt werden. Aus der Welt des Spiels gibt es eine Reihe von Fragen, die die Spieler des Brettspiels interessieren. Wer kann beispielsweise beantworten, mit wie vielen Würfelwürfen zu rechnen ist, wenn zwei, drei oder vier Spieler an einer Partie teilnehmen. Wenn Sie, lieber Leser, schon einmal *Mensch ärgere Dich nicht!* gespielt haben, werden Sie sich auch schon geärgert haben. Offensichtlich erleiden die Spieler immer mal wieder Pechsträhnen, aber glücklicherweise sind auch Glückssträhnen anzutreffen. Gelingt es in diesem Zusammenhang, solche Strähnen aufzudecken, also Phasen optisch darzustellen, in denen die Würfel einem Spieler wohl oder nicht sehr wohl gesonnen sind?

Aus Sicht der Simulation interessiert zweitens der Weg zu einem passenden Simulationsmodell. In diesem Papier lässt sich verfolgen, wie man zu lauffähigen Modellen gelangt. Weiter ist es für eine Demonstration angebracht, das Spiel selbst und Spielverläufe zu visualisieren. Dadurch lassen sich schwierige Konstellationen überprüfen, und wir erhalten eine Möglichkeit, die Implementierung zu validieren. Möglicherweise kann das Vorgehen anderen Simulations-Projekten als Orientierung dienen.

Drittens zeigt dieses Papier eine Reihe von Aspekten der Programmierung mit R, und viertens sehen wir ein Beispiel für die Anwendung des literaten Programmierstils.[2] Der gewählte Stil ist schon von Ferne zu erkennen, denn der Leser findet eine Mischung von Texten, R-Code-Sequenzen und nummerischen oder graphischen Ergebnissen vor. Die Texte bilden die Dokumentation und die Code-Stücke, die wir auch als Code-Chunks bezeichnen, sind Implementierungen kleinerer Lösungsschritte. Der eingesetzte literate Stil erlaubt neben einer Verwebung von Gedanken und Technik die Zerlegung von größeren Probleme in kleinere, die separat kommentiert und gelöst werden können. Dabei dominiert primär der Gedankengang zur Lösung die Niederschrift, technische Aspekte bestimmen erst in zweiter Linie den Gang der Dinge.

2 Grundsätzliche Überlegungen

Zu Beginn eines solchen Projekts stellt sich immer die Frage des Anfangens. In Erinnerung an Lehrbuchbeispiele ist man geneigt, in Phasen zu denken: Problemanalyse, Entwurf, Codierung, Testen und Einsatz. So werden auch wir mit

[1] Quelle: `http://de.wikipedia.org/wiki/Mensch_ärgere_Dich_nicht`.
[2] Siehe: `http://www.literateprogramming.com`

ein paar Worten zum Problem beginnen. Entwurfsgedanken sind natürlich ihrer Umsetzung vorangestellt. Jedoch sind die Phasen nicht in großen zusammenhängenden, abgegrenzten Einheiten abgebildet. Vielmehr ist ein wiederholtes Entwerfen und Codieren zu erkennen, insbesondere auch wegen des verwendeten Programmierstils. Diese Art der Umsetzung erhöht hoffentlich die Verständlichkeit und bei den Lesern die Vorstellung vom Lösungsprozess.

3 Problemanalyse.

Wir beginnen mit ein paar Bemerkungen zum Problembereich. Bei dem vorliegenden Vorhaben sind die Nebenbedingungen durch Spielaufbau und Spielregeln vorgegeben. Der Spielaufbau wird als bekannt vorausgesetzt. Für die Regeln sei Wikipedia zitiert:

> *Wer eine Sechs würfelt, muss eine eigene Spielfigur aus der Startposition heraus auf sein Startfeld des Spielfeldes stellen (auch wenn er mit einer anderen Figur einen ihm nützlicheren Zug machen könnte). Danach darf er erneut würfeln und mit der Figur entsprechend viele Felder vorrücken. Das Startfeld muss so bald wie möglich wieder freigemacht werden. Hat er aber keine Figur mehr in der Startposition, so steht es ihm frei, die erwürfelten sechs Felder mit einer Figur seiner Wahl vorzurücken. Auch dann darf er erneut würfeln und einen weiteren Zug machen.*
>
> *Kommt beim Umlauf eine Spielfigur auf ein Feld, das bereits von einer gegnerischen Spielfigur besetzt ist, gilt die gegnerische Figur als geschlagen und muss zurück auf ihre Startposition. Eigene Figuren können nicht geschlagen werden – steht eine eigene Figur auf dem Zielfeld, ist der Zug unausführbar. Hat ein Spieler mehrere Spielfiguren im Umlauf, kann er entscheiden, mit welcher er ziehen möchte. Ein Würfelwurf darf allerdings nicht aufgeteilt werden.*
>
> `http://de.wikipedia.org/wiki/Mensch_ärgere_Dich_nicht` (Nov. 2010)

Zu der Frage, was das Produkt genau können soll, sei auf die Einleitung verwiesen. Zunächst wollen wir von dem Ziel ausgehen, ein Programm zu erschaffen, das gegen sich selbst spielen, die Spielverläufe zwecks Validierung visualisieren kann und letztlich Auswertungen über Spielverläufe erlaubt.

4 Erste Umsetzungsgedanken

Für die Umsetzung des Entwurfs stellt sich wieder die Frage, womit man anfangen sollte. Sollte man zunächst eine passende Visualisierung des Spielbretts finden oder aber sich an die Abbildung des Spiel-Prozesses machen. Weiter ist zu Beginn überhaupt nicht klar, welche Datenstrukturen geeignet sind. Die Wahl der Strukturen für die Daten ist sehr kritisch, da sie später die Schwierigkeit des Zugriffs auf Daten festlegen. Es war auch hier so, dass einige Strukturentscheidungen im Laufe der Entwicklung modifiziert wurden und einige Datenstrukturen weiter verbessert werden könnten.

4.1 Spielbrettdarstellung

Zunächst einmal wird die Darstellung des Spielbretts in Angriff genommen. Neben den Feldern, über die die einzelnen Kegel bewegt werden, müssen die vier Zielfelder jeder Partei und die vier Ausgangsfelder für den Vorrat der noch nicht im Spiel befindlichen Kegel erstellt werden. Das Spielbrett soll so auf einem graphischen Device so realisiert werden, dass unten links die vier blauen Ausgangsfelder, unten rechts die roten und oben rechts die grünen zu finden sind. Für die gelben bleibt damit die Ecke oben links übrig.

Zur technischen Umsetzung der Felder müssen Kreise gezeichnet werden, die ggf. mit einer passenden Farbe ausgefüllt werden müssen. Nach Wahl eines einfachen Rasters, das in x- wie in y-Richtung von 0 bis 12 geht, ergibt sich durch Auszählen folgende Skizze:

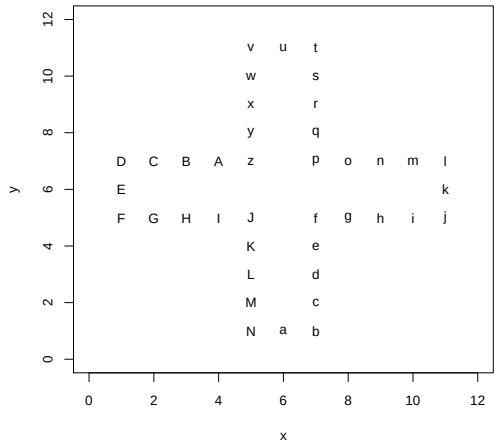
R: `<-`, `c()`, `rep()`, `":"`, `plot()`

1 ⟨ *erstelle Skizze für das Brett* 1⟩ ≡

```
x <- c(6,rep(7,5),8:10,rep(11,3),10:8,rep(7,5),
       6,rep(5,5),4:2, rep(1,3),  2:4,rep(5,5))
y <- c(rep(1,2),  2:4,rep(5,5),  6,rep(7,5),8:10,
       rep(11,3),10:8,rep(7,4),7:6,rep(5,5),4:1)
plot(x,y,xlim=c(0,12),ylim=c(0,12),pch=c(letters,LETTERS))
```

R: Wir erkennen in dem Code-Chunk die Konstruktion der beiden Vektoren `x` *und* `y`*. Sie entstehen durch Verkettung kleinerer Einheiten mittels der Funktion* `c()`*. Einzelne Einheiten sind entweder Zahlen, mit dem Sequenzoperator* `":"` *gebildete Zahlensequenzen oder aber mit* `rep()` *erzeugte Zahlenwiederholungen. Das Bild wird mit der High-Level-Plotfunktion* `plot()` *konstruiert, der die x- und y-Werte sowie die Grenzen der Darstellung mitgegeben werden. Über das Argument* `pch` *werden Symbole für die Punkte übergeben. Hierfür wird auf die in R enthaltenen Variablen* `letters` *und* `LETTERS` *zugegriffen, deren Bedeutung auf der Hand liegen dürfte.*

Die Ausführung der letzten Anweisungen liefert folgendes Bild:



4.2 Datenstrukturüberlegungen

Schön, die grobe Struktur passt zum Spielbrett. Weniger gut gefällt, dass wir mit den Punkten nicht mehr weiter arbeiten können. Später wird es notwendig sein, auf die einzelnen Spielfelder für die Kegel zugreifen zu können. Auch könnte sich ergeben, dass zu den Spielfeldern Eigenschaften verwaltet werden müssen. Wir wollen zunächst jedem Spielfeldpunkt durch Zeilen einer Matrix repräsentieren. Dazu wiederholen wir die Auszählung, jedoch beginnen wir jetzt bei dem Startfeld für die blauen Kegel (unten links) und fädeln die Felder in der Reihenfolge auf, wie sie die Kegel besuchen sollten. Der Matrix geben wir den Namen `fields`. In der ersten Spalte sind die x-Werte und in der zweiten die y-Werte zu finden.

R: `cbind()`, `#`

2 ⟨*generiere Felder* 2⟩ ≡ ⊂ 9, 24, 43, 64, 80, 88

```
## normale Felder
x <- c(rep(5,5),4:2, rep( 1,3), 2:4,rep(5,5),6,
       rep(7,5),8:10,rep(11,3),10:8,rep(7,5),6)
y <- c( 1:4,rep(5,5),6:7,rep(7,4),8:10,rep(11,3),
       10:8,rep(7,5),6,  rep(5,5),4:2, rep(1,2))
fields <- cbind(x,y)
```

R: Die Funktion `cbind()` *fügt Vektoren spaltenweise zu einer Matrix zusammen. Das Zeichen* `#` *ist übrigens das Kommentarzeichen von R.*

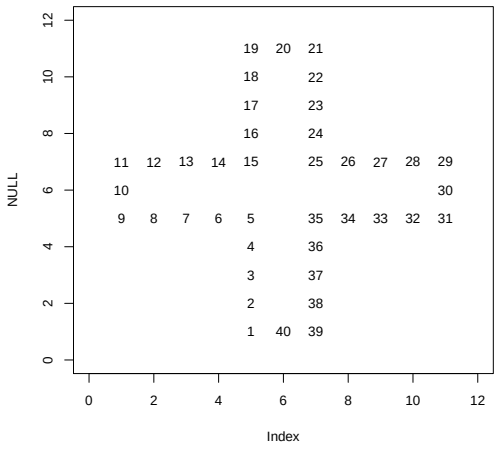
Es erfolgt die Umsetzung der Anweisungen zur Darstellung.

R: `NULL`, `text()`, `seq()`

3 ⟨*zeige Felder* 3⟩ ≡

```
plot(NULL,xlim=c(0,12),ylim=c(0,12)); text(x,y,seq(x))
```

R: Die Funktion `text()` *erzeugt Texte an den durch Koordinaten beschriebenen Punkten. Die Texte werden über das dritte Argument mitgeliefert. Im vorliegenden Fall handelt es sich um die Zahlen von 1 bis 40, die mit der Sequenzerzeugungsfunktion* `seq()` *generiert werden. Erhält diese Funktion einen Vektor als Argument, dann bildet sie den Vektor der möglichen Indizes des angegebenen Vektors.* `NULL` *charakterisiert ein leeres Objekt.*



Auf der Matrix `fields` sind nun die Koordinaten der Felder abgelegt. Entsprechend können wir auch mit den Vorratsfeldern für die vier Farben und deren Zielfeldern verfahren. Für die Zielfelder werden vier Matrizen mit Namen der Form *Farbe*`.ziel` eingerichtet, die die Koordinaten in den ersten beiden Spalten enthalten und in der dritten Spalte noch eine Zustandsinformation aufnehmen können. R: `";"`

4 ⟨*generiere Felder 2*⟩+ ≡ ⊂ 9, 24, 43, 64, 80, 88

```
## Zielfelder
x <- rep(6,4); y <- 2:5;        blue.ziel <- cbind(x,y,state=0)
x <- 10:7;     y <- rep(6,4);    red.ziel <- cbind(x,y,state=0)
x <- rep(6,4); y <- 10:7;      green.ziel <- cbind(x,y,state=0)
x <- 2:5;      y <- rep(6,4); yellow.ziel <- cbind(x,y,state=0)
```

R: Es geht nichts über schön niedergeschriebenen Code. Schönheit erhöht die Lesbarkeit und vermeidet viele Fehler! Mit dem `";"`*-Zeichen lassen sich innerhalb einer Zeilen einzelne R-Anweisungen trennen.*

Die Ausgangs- oder Vorratsfelder wollen wir ebenfalls mit Matrizen (mit den Namen *Farbe*`.start`) verwalten. Ihre Struktur entspricht derjenigen der Zielmatrizen. R: `"[]"`

5 ⟨*generiere Felder 2*⟩+ ≡ ⊂ 9, 24, 43, 64, 80, 88

```
## Startfelder
x       <- c(1.5,10.5,10.5,1.5); y       <- c(1.5,1.5,10.5,10.5)
x.shift <- c(0.5,0.5,-0.5,-0.5); y.shift <- c(-0.5,0.5,0.5,-0.5)
  blue.start <- cbind(x[1]+x.shift, y[1]+y.shift,state=0)
   red.start <- cbind(x[2]+x.shift, y[2]+y.shift,state=0)
 green.start <- cbind(x[3]+x.shift, y[3]+y.shift,state=0)
yellow.start <- cbind(x[4]+x.shift, y[4]+y.shift,state=0)
```

R: Im Prinzip zeigt uns der Chunk keine neuen R-Techniken. Die Koordinaten der gewünschten Kreise finden wir, indem wir die Koordinaten der Zentren der vier Vorratslager auf `x` *und* `y` *ablegen und jedes Zentrum dann mit Shift-Vektoren verschieben. Hierfür ist es sehr bequem, dass man zu Vektoren ein-elementige Vektoren addieren kann. Die ein-elementigen Objekte haben wir wie in anderen Sprachen durch Indexzugriff erhalten.*

4.3 Fertige Darstellung des Spielbrettes

Jetzt wollen wir uns einmal das Spielbrett ohne Kegel zusammenbauen. Für die Gesamtdarstellung muss ein Device geöffnet werden. Auf dieser Grundlage wollen wir die oben definierten Felder durch Kreise darstellen. R: `rect()`

6 ⟨*stelle Spielfeld dar 6*⟩ ≡ ⊂ 9, 24, 43, 64, 65, 80, 88, 89

```
plot(NULL,xlim=c(0,12),ylim=c(0,12),xlab="",ylab="",axes=FALSE)
rect(0,0,12,12,col="gray")
```

R: `rect()` *erstellt ein Rechteck, im vorliegenden Fall in grau.*

Da alle Felder ein ähnliches Design haben, wird für die Zeichnung der runden Felder eine kleine Funktion entworfen. Diese Funktion soll mit zwei Koordinaten-Vektoren (`x` und `y`) versorgt werden können, aber auch mit einer 2-spaltigen Matrix oder auch nur mit zwei Einzelwerten in einem Vektor. Es sollen Farben, die Größe und ggf. ein textlicher Inhalt für die Kreise (`txt`) angebbar sein.

`make.circle`	ergänzt Kreise in einer graphischen Darstellung
Input:	
`x`	Vektor der x-Werte, zweispaltige Matrix oder zweielementiger Vektor mit einer x- und einer y-Koordinaten
`y`	ggf. Vektor der y-Werte
`col`	Füllfarbe für Kreise
`size`	Kreisgröße
`txt`	ins Zentrum zu schreibender Text
Output:	
—	Kreise auf Device

Eine solche Zusammenfassung sollte man für jede Funktion erstellen.

7 ⟨*stelle Spielfeld dar* 6⟩+ ≡ ⊂ 9, 24, 43, 64, 65, 80, 88, 89

```
# eine Funktion, um Kreise zu zeichnen
make.circle <- function(x,y,col="white",size=4.5,txt=""){
  if(exists("make.no.circles")) return()
  if(missing(y)){
    if(is.matrix(x)){
      y<-x[,2]; x<-x[,1]
    } else {
      y<-x[ 2]; x<-x[ 1]
    }
  }
  points(x,y,cex=size,lwd=3)
  points(x,y,cex=size*0.9,col=col,pch=16)
  if(!is.null(txt)){
    if(""==txt[1]){
      text(x,y,seq(x))
    } else {
      text(x,y,txt)
    }
  }
}
```

R:
`function()`,
`points()`,
`is.matrix()`,
`missing()`,
`if()`,
`is.null()`,
`exists()`,
`return()`

R: Hier ist sind schon tiefere R-Tricks zu erkennen, denn man sieht, wie man mit `function()` *eine Funktion definieren kann. Der Kopf der Funktion legt die Argumente fest, im Rumpf sind die Handlungen beschrieben.* `missing()` *stellt im vorliegenden Fall fest, ob beim Aufruf ein Argument (hier:* `y`*) mitgeliefert wurde. Man erkennt weiter die Syntax einer if-then-else-Konstruktion in R. Es wird äußerst dringend empfohlen, immer geeignete Einrückungen zu verwenden. Die Funktion* `points()` *ergänzt in einem Plot Punkte, deren Größe über* `cex` *festgelegt wird. Das Argument* `lwd` *von* `points()` *steuert die Strichstärke von Linien. Mit* `is.matrix()` *und* `is.null()` *lassen sich feststellen, ob ein Objekt eine Matrix bzw. leer ist. Die Wirkung der geschachtelten if-Konstruktionen möge der Leser anhand ausgedachter Beispiele überlegen. Diese Konstruktionen sollen dafür sorgen entsprechend der Situation geeignete Maßnahmen umzusetzen. Die Funktion* `exists()` *überprüft die Existenz eines Objektes.* `return()` *unterbricht die weitere Abarbeitung der Funktion, im vorliegendem Fall, falls die*

Variable `make.no.circles` *nicht existiert.*

Die neu definierte Funktion kann nun mit den passenden Matrizen als Inputs aufgerufen werden.

8 ⟨*stelle Spielfeld dar* 6⟩+ ≡ ⊂ 9, 24, 43, 64, 65, 80, 88, 89

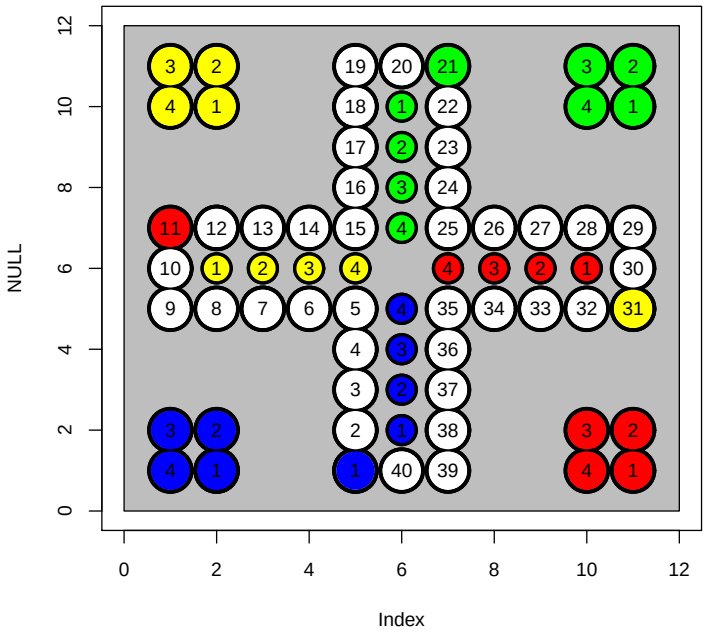
```
# Startfelder
make.circle(  blue.start,col="blue")
make.circle(   red.start,col="red")
make.circle( green.start,col="green")
make.circle(yellow.start,col="yellow")

# Felder
col.fields <- rep("white",40)
col.fields[c(1,11,21,31)] <- c("blue","yellow","green","red")
make.circle(fields,col=col.fields)

## Ziele:
make.circle(  blue.ziel,col="blue",  size=3,txt=letters[1:4])
make.circle(yellow.ziel,col="yellow",size=3,txt=letters[1:4])
make.circle( green.ziel,col="green", size=3,txt=letters[1:4])
make.circle(   red.ziel,col="red",   size=3,txt=letters[1:4])
```

R: Die neu definierte Funktion können wir offensichtlich mit mehr oder weniger Argumenten aufrufen.

Als Ergebnis bekommen wir folgendes Bild.



Damit ist die Konstruktion des Spielbretts abgeschlossen.

5 Spiel mit einem Spieler und einem Kegel

Ein Versuch, sofort das fertige Spiel umzusetzen, ist nicht zu empfehlen. Deshalb wollen auch wir zunächst ein ganz einfaches Spiel, ein Spiel mit nur einem einzigen blauen Kegel, entwerfen. Die grobe Struktur besteht darin, das Spielfeld aufzubauen, dann wiederholt zu würfeln und die Folgewirkungen umzusetzen. Für die Folgewirkungen werden die verschiedenen Situationen, die entstehen können, nach einander abgehandelt: Ärgern, Kegel heraussetzen, Kegel weitersetzen und Kegel ins Zielfeld stellen. Es werden jeweils Meldungen ausgegeben und notwendige Aktionen benannt, deren Bedeutungen durch die gewählten Namen klar werden müssten. Die durch spitze Klammern begrenzte und in italic gesetzten Anweisungen sind Namen von Code-Chunks, die in dem Papier an anderer Stelle definiert sein müssen. Für die Auswertung werden sie automatisch an der Stelle ihrer Verwendung eingefügt. Einige der schon erarbeiteten Code-Sequenzen finden Verwendung, andere werden weiter unten definiert. Der Kegel selbst wird in der selben Farbe wie die Brettpunkte gezeichnet, nur mit anderer Helligkeit und etwas kleiner.

R: `repeat, break, cat(), &&, !=, ==, next, TRUE`

9 ⟨*spiele einfachstes Spiel* 9⟩ ≡ ⊂ 22

```
⟨generiere Felder 2⟩
⟨stelle Spielfeld dar 6⟩
⟨initialisiere einfachstes Spiel 21⟩
repeat{
  ⟨verhindere Endlosschleife 25⟩
  if(fertig) break
  ⟨würfle den Würfel 20⟩
  if( WUERFEL != 6 && ⟨ist Kegel noch nicht im Spiel 16⟩ ){
    cat("Mensch ärgere Dich!"); next
  }
  if( WUERFEL == 6 && ⟨ist Kegel noch nicht im Spiel 16⟩ ){
    ⟨nehme Kegel aus Vorrat 11⟩
    ⟨stelle Kegel auf das Startfeld 12⟩
    cat("Juchhu 6 gefallen!\nSetze Kegel jetzt raus!"); next
  }
  if( ⟨ist Zielfeld ein normales Feld 17⟩ ){
    ⟨entferne Kegel 13⟩
    ⟨stelle Kegel auf neue Position 14⟩
    cat("Position neu\n",kegel); next
  }
  if( ⟨ist Zielfeld im Ziel 18⟩ ){
    ⟨entferne Kegel 13⟩
    ⟨stelle Kegel ins Ziel 15⟩
    cat("geschafft!"); fertig<-TRUE; next
  }
  next
}
```

R: Die große Handlung findet in einer Repeat-Schleife statt, deren Syntax unmittelbar zu erkennen ist. `next` *und* `break` *sind Schleifen spezifische Anweisungen.* `next` *löst sofort den nächsten Schleifendurchlauf aus.* `break` *sorgt für den Abbruch der Schleife. Als neue Funktion wird* `cat()` *eingesetzt; sie schreibt Texte zur Information auf den Bildschirm. Der Operator* `&&` *verknüpft zwei einzelne Wahrheitswerte per* Und. `==` *und* `!=` *untersuchen elementweise die Gleichheit*

bzw. Ungleichheit von zwei Objekten. `TRUE` *ist eine Konstante mit der naheliegender Bedeutung.*

Damit ist der zentrale Ablauf geklärt.

5.1 Abbildung des Kegels

Für die Repräsentation des Kegels führen wir die Variable `kegel` ein. Auf dieser legen wir die Nummer des Feldes ab, auf dem unser Kegel steht. Zur Kennzeichnung seines Platzes im Ziel verwenden wir seine Position im Zielbereich als negative Zahl. Ein Kegel, der noch nicht im Spiel ist, erhält als Code die Nummer 0. Mit unserer Funktion `make.circle()` können wir den Kegel bequem auf sein Feld im Vorratsbereich setzen.

10 ⟨*initialisiere Kegel* 10⟩ ≡ ⊂ 21

```
kegel <- 0
make.circle(blue.start[1,],col="lightblue",size=3, txt="1")
```

5.2 Setzen des Kegels

Wird ein Kegel durch Würfeln einer 6 aus dem Vorrat genommen, muss seine Darstellung im Vorrat gelöscht werden.

11 ⟨*nehme Kegel aus Vorrat* 11⟩ ≡ ⊂ 9

```
make.circle(blue.start[1,],col="blue",txt=1)
```

Nachdem der Kegel aus dem Vorrat genommen worden ist, muss er auf das Startfeld gestellt werden. Dazu weisen wir `kegel` den Wert 1 zu. Wir erhalten die neuen Koordinaten mit Hilfe der ersten Zeile von `fields` und können die Darstellung mit `make.circle()` umsetzen.

12 ⟨*stelle Kegel auf das Startfeld* 12⟩ ≡ ⊂ 9

```
kegel <- 1
make.circle(fields[kegel,],col="lightblue",size=3,txt=kegel)
```

Bei einem normalem Zug wird der Kegel zunächst von einen Feld entfernt.

13 ⟨*entferne Kegel* 13⟩ ≡ ⊂ 9

```
make.circle(fields[kegel,],col=col.fields[kegel],txt=kegel)
```

Ein Weitersetzen erfordert es, die neue Position zu berechnen und das Zielfeld neu zu färben.

14 ⟨*stelle Kegel auf neue Position* 14⟩ ≡ ⊂ 9

```
kegel <- kegel+WUERFEL
make.circle(fields[kegel,],col="lightblue",size=3,txt=1)
```

Ziel ist es, den Kegel im Zielbereich unterzubringen. Falls die angepeilte Zielstelle auf `pos.wish` abgelegt ist, kann der Wert von `kegel` neu gesetzt und die Farbveränderung des Zielfeldes durchgeführt werden. Weiterhin wird an dem Zielfeld die Information angebracht, dass es belegt ist.

15 ⟨*stelle Kegel ins Ziel* 15⟩ ≡ ⊂ 9

```
kegel <- -pos.wish
make.circle(blue.ziel[pos.wish,],col="lightblue",size=3,txt=1)
blue.ziel[pos.wish,"state"] <- 1
```

5.3 Bedingungsüberprüfungen

Bei der Abwicklung des einfachen Spiels sind eine Reihe von Bedingungen zu beachten.

16 ⟨*ist Kegel noch nicht im Spiel* 16⟩ ≡ ⊂ 9

```
0==kegel
```

Sofern die neue Feldnummer nicht größer als 40 ist, handelt es sich um einen normalen Zug.

17 ⟨*ist Zielfeld ein normales Feld* 17⟩ ≡ ⊂ 9

```
WUERFEL+kegel <= 40
```

Ob der Kegel ins Ziel gesetzt werden darf, hängt davon ab, ob die berechnete Zielstelle existiert und frei ist.

18 ⟨*ist Zielfeld im Ziel* 18⟩ ≡ ⊂ 9

```
(pos.wish <- WUERFEL+kegel-40)<=4 && blue.ziel[pos.wish,"state"]==0
```

R: Übrigens wird während der Bedingungsüberprüfung die mögliche Zielposition ermittelt und auf `pos.wish` *abgelegt. So richtig schön ist das nicht, zumal die Mischung von Zuweisung und Bedingungsprüfung einen Anfänger verwirren könnte.*

Mehr aus Sicherheits-Gründen soll eine Endlosschleife verhindert werden. Deshalb zählen wir in jedem Schleifendurchgang die Variable `w.max` herunter und brechen aus der Schleife aus, wenn der Wert unter 1 sinkt.

R: `Sys.sleep()`

19 ⟨*verhindere Endlosschleife für das Ein-Kegel-Spiel* 19⟩ ≡

```
Sys.sleep(sleep); w.max <- w.max-1; if(w.max<1) break;
```

R: `Sys.sleep()` *hält die weitere Abwicklung für die angegebenen Sekunden an.*

5.4 Das Würfel-Werfen und Restarbeiten

Zum Schluss muss noch das Würfeln umgesetzt werden.

20 ⟨*würfle den Würfel* 20⟩ ≡ ⊂ 9, 24, 43

```
WUERFEL <- sample(1:6,1)
```

Was fehlt noch zur Komplettierung des einfachen Spiels? Antwort: einige Initialisierungen.

21 ⟨*initialisiere einfachstes Spiel* 21⟩ ≡ ⊂ 9

```
sleep <- 1
⟨initialisiere Kegel 10⟩
fertig <- FALSE; w.max<-20
```

5.5 Die Konstruktion einer Spielfunktion

Für die leichtere Handhabung gießen wir die Lösungsschritte in eine Funktion mit dem Namen `play.mad.simple()`.

`play.mad.simple`	führt eine simples Spiel durch
Input:	
--	
Output:	
—	Darstellung des Bretts und des einfachen Spielverlaufs

22 ⟨*definiere Funktion zum einfachen Spiel:* `play.mad.simple()` 22⟩ ≡ ⊂ 23, 87

```
play.mad.simple <<- function(){
  ⟨spiele einfachstes Spiel 9⟩
}
```

23 ⟨*führe einfaches Spiel aus* 23⟩ ≡

```
⟨definiere Funktion zum einfachen Spiel: play.mad.simple() 22⟩
play.mad.simple()
```

6 Spiel mit einem Spieler und mehreren Kegeln

Aufbauend auf den Ideen des einfachen Spiels wollen wir im zweiten Schritt ein Spiel mit einem Spieler und mehreren Kegeln abbilden. Zentrale Konzepte, wie Spielaufbau, Spielablauf usw., können wir übernehmen. Jedoch müssen wir uns die einzelnen Punkte modifizieren, in denen Kegel vorkommen.

6.1 Grobstruktur des Spiels mit einem Spieler

Betrachten wir das fertige Rahmenwerk: R: `sum()`

24 ⟨*spiele Spiel mit nur einem Spieler* 24⟩ ≡ ⊂ 26, 39

```
⟨generiere Felder 2⟩
⟨stelle Spielfeld dar 6⟩
⟨initialisiere Spiel mit einem Spieler 28⟩
⟨stelle Funktion melde bereit 27⟩
WUERFEL<-0
melde("Initialisieren")
repeat{
  cat("Info: Augenzahl",WUERFEL,"Kegel-Felder",kegel)
  ⟨verhindere Endlosschleife 25⟩
  if(fertig) break
  ⟨würfle den Würfel 20⟩
  if( WUERFEL != 6 && ⟨muss neuer Kegel eingebracht werden 29⟩ ){
    cat("Info: im Kopf des Spieler: Mensch aergere Dich!"); next
  }
```

```
  melde("Aegern")
  if( WUERFEL == 6 && ⟨kann neuer Kegel eingebracht werden 30⟩ ){
    ⟨nehme einen Kegel aus dem Vorrat 33⟩
    ⟨stelle gewählten Kegel auf das Startfeld 34⟩
    cat("Info: Spieler: Juchhu 6 gefallen! Kegel raus!")
    next
  }
  melde("Raussetzen")
  ⟨wähle Kegel zum Setzen 38⟩
  melde("Kegel wählen",kegel)
  if( ⟨ist Zielfeld im Ziel des vordersten Kegels 32⟩ ){
    ⟨entferne Kegel mit Index idx 35⟩
    ⟨stelle Kegel mit Index idx ins Ziel 37⟩
    cat("Info: Spieler ruft: Kegel gerettet!")
    if(sum(kegel<0)==4){
      cat("Info: Spieler ist nicht zu bremsen: na also!")
      fertig<-TRUE
    }
    next
  }
  melde("Kegel ins Ziel stellen")
  if( ⟨ist Zielfeld des gewählten Kegels auf Spielfeld 31⟩ ){
    ⟨entferne Kegel mit Index idx 35⟩
    ⟨stelle gewählten Kegel auf neue Position 36⟩
    next
  }
  melde("Vorsetzen"); next
}
```

R: Die Funktion `sum()` *berechnet die Summe der Werte eines Vektors.*

Wir wollen schon an dieser Stelle mehrere Möglichkeiten zulassen, die Wartezeit zu setzen. Darum verwenden wir eine Referenz auf einen Code-Chunk, den wir erst weiter unten definieren.

25 ⟨*verhindere Endlosschleife* 25⟩ ≡ ⊂ 9, 24, 43, 80

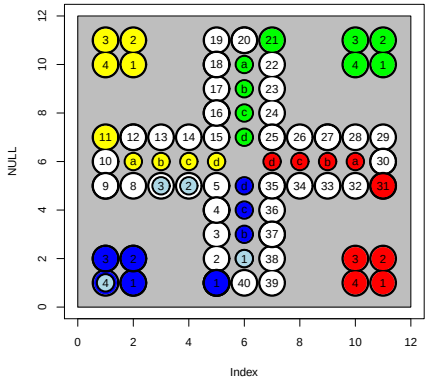
```
⟨stelle ggf. neue Wartezeit fest 85⟩
Sys.sleep(sleep); w.max <- w.max-1; if(w.max<1) break;
```

Bevor wir uns an die Umsetzung machen, wollen wir zeigen, welche Ausgaben wir beispielsweise erwarten. Hierzu starten wir folgenden Code-Chunk, was natürlich nur erfolgreich ist, wenn die verwendeten Chunks schon existieren.

26 ⟨*spiele Spiel mit einem Spieler* 26⟩ ≡

```
w.max<-20; set.seed(17)
⟨spiele Spiel mit nur einem Spieler 24⟩
```

Zu diesem Probelauf gehören folgender Endzustand und das daneben stehende Protokoll:




```
Info: Augenzahl 4 Kegel-Felder 0 0 0 0
Info: im Kopf des Spieler: Mensch aergere Dich!
Info: Augenzahl 1 Kegel-Felder 0 0 0 0
Info: Spieler: Juchhu 6 gefallen! Kegel raus!
Info: Augenzahl 6 Kegel-Felder 1 0 0 0
Info: Augenzahl 3 Kegel-Felder 4 0 0 0
Info: Augenzahl 5 Kegel-Felder 9 0 0 0
Info: Augenzahl 3 Kegel-Felder 12 0 0 0
Info: Augenzahl 4 Kegel-Felder 16 0 0 0
Info: Augenzahl 2 Kegel-Felder 18 0 0 0
Info: Augenzahl 2 Kegel-Felder 20 0 0 0
Info: Augenzahl 5 Kegel-Felder 25 0 0 0
Info: Augenzahl 2 Kegel-Felder 27 0 0 0
Info: Augenzahl 3 Kegel-Felder 30 0 0 0
Info: Augenzahl 1 Kegel-Felder 31 0 0 0
Info: Spieler: Juchhu 6 gefallen! Kegel raus!
Info: Augenzahl 6 Kegel-Felder 31 1 0 0
Info: Augenzahl 5 Kegel-Felder 31 6 0 0
Info: Spieler: Juchhu 6 gefallen! Kegel raus!
Info: Augenzahl 6 Kegel-Felder 31 6 1 0
Info: Augenzahl 6 Kegel-Felder 31 6 7 0
Info: Augenzahl 4 Kegel-Felder 35 6 7 0
Info: Augenzahl 2 Kegel-Felder 37 6 7 0
Info: Spieler ruft: Kegel gerettet!
Info: Augenzahl 4 Kegel-Felder -1 6 7 0
```

Wir stellen fest, dass im Prinzip die grobe Struktur sehr ähnlich der ersten Lösung ist. Auf drei Dinge sei jedoch besonders hingewiesen:

- Es gibt verschiedene Aufrufe einer Funktion mit dem Namen `melde()`. Diese diente in der Entwicklungsphase dazu, Meldungen auszugeben und die Fehlersuche zu erleichtern.
- Es wurde nach Abhandlung des Heraussetzen ein neuer Chunk mit dem Namen ⟨*wähle Kegel zum Setzen* 38⟩ eingefügt. Durch die Verkomplizierung gegenüber dem einfachsten Spiel erschien es zweckmäßig, zunächst über die Auswahl des zu bewegenden Kegels nachzudenken und in einem zweiten Schritt über die Kegelbewegungen.
- Die Namen der referenzierten Code-Chunks sind gegenüber der ersten Version größtenteils modifiziert, damit keine falschen Code-Stücke eingebaut werden.

6.2 `melde()`

Die oben schon motivierte Funktion `melde` muss definiert werden. Falls eine Variable mit dem Namen `Debug` existiert und den Wert `TRUE` besitzt, werden die Argumente der Aufrufe ausgegeben.

`melde`	schreibt Meldung über den Spielzustand auf die Ausgabe
Input:	
`what`	textliche Charakterisierung der Position im Ablauf
`...`	ggf. weitere Mitteilungen
Output:	
—	Text-Ausgaben auf den Bildschirm

R: `exists()`, `||`, `paste()`

27 ⟨*stelle Funktion* `melde` *bereit* 27⟩ ≡ ⊂ 24, 50
```
melde <- function(what="relax",...){
  if(!exists("Debug")||!Debug) return()
  print(paste("--> Position: [",what,"] erledigt"))
  if(!missing(...)) cat(...)
}
if(exists("NO.OUT") && NO.OUT) cat <- function(...){ "relax" }
```

R: `exists()` *prüft die Existenz eines Objektes. Die beiden Striche* `"||"` *sind der Operator der* Oder*-Operation. Mit* `paste()` *lassen sich Zeichenketten zusammenfügen. In der letzten Code-Zeile wird ein Trick angewandt, der nur sehr behutsam eingesetzt werden sollte. Und zwar wird dort eine Funktion mit einem Namen einer schon definierten Funktion definiert. Hierdurch wird die originale Schreibfunktion ausgebremst, und es kommt zu keinen Ausgaben, wenn* `NO.OUT` *den Wert* `TRUE` *besitzt.*

6.3 Initialisierung des Spiels mit einem Spieler

Wir müssen während der Initialisierung vier Kegel schaffen und auf das Brett stellen. Deshalb müssen auch vier Kegel initialisiert werden, und `kegel` wird deshalb zu einem Vektor der Länge 4 erweitert. Wie beim einfachen Spiel werden die technischen Variablen `w.max` und `sleep` gesetzt.

28 ⟨*initialisiere Spiel mit einem Spieler* 28⟩ ≡ ⊂ 24
```
kegel <- rep(0,4)
make.circle(blue.start,col="lightblue",size=3)
fertig <- FALSE
if(!exists("w.max")) w.max<-60
⟨setze Wartezeit sleep 82⟩
```

6.4 Handlungsbedingungen

Wann muss ein neuer Kegel ins Spiel gebracht werden? Natürlich muss sich ein Kegel im Vorrat befinden. Weiterhin darf sich kein Kegel auf den normalen Feldern des Brettes befinden, es darf also kein Kegel einen Wert zwischen 1 bis 40 besitzen.

R: `%in%`, `any()`

29 ⟨*muss neuer Kegel eingebracht werden* 29⟩ ≡ ⊂ 24
```
any(0==kegel) && !any(kegel %in% 1:40)
```

R: Hinter `%in%` *verbirgt sich der aus der Mathematik bekannte Element-Operator. Die Funktion* `any()` *stellt fest, ob irgend ein Wahrheitswert den Wert* `TRUE` *hat. Demgegenüber liefert* `all()` *nur dann ein* `TRUE`*, wenn alle Werte des Inputs* wahr *anzeigen.*

Ein neuer Kegel kann eingebracht werden, falls noch mindestens ein Stein im Vorrat ist sowie kein eigener Stein auf dem Startfeld steht. R: `all()`

30 ⟨*kann neuer Kegel eingebracht werden* 30⟩ ≡ ⊂ 24
```
any(0==kegel) && all(kegel!=1)
```

Das neue Feld für einen Kegel ergibt sich durch Addition der Kegelposition mit dem Würfelergebnis. Ist diese Summe kleiner als 40, ist das Ziel ein zulässiges Feld auf dem Brett. Hierbei wird davon ausgegangen, dass der zu bewegende Kegel bereits ausgewählt wurde und die Nummer `idx` trägt.

31 ⟨*ist Zielfeld des gewählten Kegels auf Spielfeld* 31⟩ ≡ ⊂ 24
```
(WUERFEL+kegel[idx]) <= 40
```

Damit sich das neue Feld im Zielbereich befindet, muss die Summe aus Würfelaugenzahl und Kegelposition abzüglich 40 in der Menge der Zahlen von eins bis vier liegen, und außerdem muss dieses Zielfeld (`pos.wish`) noch frei sein. `pos.wish` wird wie schon beim einfachsten Spiel später noch gebraucht, um den Kegel an die passende Zielstelle zu platzieren.

32 ⟨*ist Zielfeld im Ziel des vordersten Kegels* 32⟩ ≡ ⊂ 24
```
(pos.wish <- WUERFEL+kegel[idx]-40) %in% (1:4) &&
  (blue.ziel[pos.wish,"state"]==0)
```

6.5 Kegelbewegungen

Wenn ein Kegel ins Spiel gebracht werden muss, ist ein Stein aus dem Vorrat zu bestimmen. Es wird derjenige mit der niedrigsten Nummer gewählt. Die Nummer wird auf `idx` abgelegt und für das Platzieren benötigt.

33 ⟨*nehme einen Kegel aus dem Vorrat* 33⟩ ≡ ⊂ 24
```
idx <- (1:4)[0==kegel][1]
make.circle(blue.start[idx,],col="blue",txt=idx)
melde("Kegelauswahl","Nummer: ------------",idx)
```

Wird ein Kegel aus dem Vorrat ins Spiel gebracht, muss er auf das Feld 1 gestellt werden. Die Nummer des Kegels ist `idx` zu entnehmen.

34 ⟨*stelle gewählten Kegel auf das Startfeld* 34⟩ ≡ ⊂ 24
```
kegel[idx] <- 1
make.circle(fields[kegel[idx],],col="lightblue",size=3,txt=idx)
```

Das Aufnehmen des Kegels mit dem Index `idx` leistet folgender Chunk. Dazu muss die passende Nummer auf `idx` stehen.

35 ⟨*entferne Kegel mit Index* `idx` 35⟩ ≡ ⊂ 24
```
make.circle(fields[kegel[idx],],col=col.fields[kegel[idx]],
            txt=kegel[idx])
```

```
melde("Kegelentfernung","Nummer des Kegels (idx):",idx)
```

Bei einem normalen Zug wird ein Kegel aufgenommen und dann auf der neuen Position wieder auf das Brett gesetzt.

36 ⟨*stelle gewählten Kegel auf neue Position* 36⟩ ≡ ⊂ 24

```
kegel[idx] <- kegel[idx]+WUERFEL
make.circle(fields[kegel[idx],],col="lightblue",size=3,txt=idx)
```

Die Zielfelder werden durch negative Zahlen kodiert. Deshalb wird auf der Variablen `kegel` die negative gewünschte Position vermerkt. Ebenfalls wird in der Matrix der Zielfelder der Zustand angepasst.

37 ⟨*stelle Kegel mit Index* `idx` *ins Ziel* 37⟩ ≡ ⊂ 24

```
kegel[idx] <- -pos.wish
make.circle(blue.ziel[pos.wish,],col="lightblue",size=3,txt=idx)
blue.ziel[pos.wish,"state"] <- 1
```

6.6 Auswahlfragen

Welches ist der Kegel der im nächsten Zug gesetzt wird? Zunächst werfen wir einen Blick auf den vordersten Kegel. Falls jedoch ein Kegel auf dem Feld mit der Nummer 1 steht und außerdem noch ein Kegel im Vorrat ist, muss das Startfeld sofort geräumt werden. Jedoch geht das Räumen nur, wenn auf dem angepeilten neuen Feld kein eigener Kegel steht. Man sieht, es fügen sich eine Reihe von Bedingungen ineinander.

Gibt es mehrere Steine, die für normale Züge infrage kommen, wollen wir als einfache Strategie den vordersten weitersetzen.

R: `which.max()`

38 ⟨*wähle Kegel zum Setzen* 38⟩ ≡ ⊂ 24

```
idx <- which.max(kegel)
if(any(kegel==0) && any(kegel==1) && !any(kegel==(1+WUERFEL))){
  idx <- (1:4)[kegel==1] # Platte putzen
}
```

R: `which.max()` *ermittelt die Position des größten Elementes in einem Vektor. Damit können wir leicht den Kegel ermitteln, der maximal weit vorn ist.*

An dieser Stelle gibt es noch ein ungelöstes Problem: Was ist, wenn der vorderste Kegel nicht gezogen werden kann, weil man kein freies Zielfeld mit der realisierten Augenzahl treffen würde? Dann müsste man den Zweit-Vordersten betrachten. Im Moment wird jedoch auf ein Ziehen verzichtet. Man ahnt, dass dieser Chunk bei der nächsten Version an Relevanz gewinnen wird.

6.7 Eine Funktion zum *Mensch ärgere Dich nicht!*-Spielen

Wie oben kann es hilfreich sein, wenn das Spiel als eigenständige Funktion bereit steht. Deshalb sei eine solche hier eingeführt.

play.mad.one	erlaubt ein Spiel mit einem Spieler und 4 Kegeln
Input:	
nr	Start des Zufallszahlengenerators
Debug	zum Setzen der Variablen Debug
sleep	Pausierzeit pro Schleifendurchlauf
w.max	maximale Anzahl von Würfelwürfen
NO.OUT	falls TRUE werden Ausgaben unterbunden
Output:	
—	Text-Ausgaben auf den Bildschirm
—	Positionen der Kegel

R: return(), set.seed()

39 ⟨*definiere Funktion zum ein-Spieler-Spiel:* play.mad.one() 39⟩ ≡ ⊂ 40, 87

```
play.mad.one <<- function(nr=16,Debug=FALSE,sleep=0.25,
                          w.max=80,NO.OUT=FALSE){
  set.seed(nr)
  ⟨spiele Spiel mit nur einem Spieler 24⟩
  return(kegel)
}
# play.mad.one(NO.OUT=TRUE)
```

R: return() *führt zur Beendigung einer Funktion. Im vorliegenden Fall wird das Objekt* kegel *als Funktionswert zurückgegeben. Mit* set.seed() *wird ein Start für den Zufallszahlengenerator gewählt. Bei gleichen Startwerten werden die selben Zufalls-Zahlen gezogen, so dass Simulationsergebnisse reproduziert werden können.*

6.8 Modellkritik

In der vorliegenden Lösung ist noch nicht verarbeitet, dass man laut Regelwerk im Bereich der Zielfelder ebenfalls vorrücken kann. Auch wird – wie schon erwähnt – beim normalen Vorrücken nur der vorderste Kegel betrachtet.

6.9 Eine kleine Auswertung

Eine Auswertung mehrerer Spiele kann nun beginnen. Zum Beispiel könnte uns interessieren, welche Unterschiede die Spielergebnisse aufweisen, wenn ein Spiel auf 30 Runden begrenzt wird.

R: for(), t()

40 ⟨*simuliere mehrere Spiele mit einem Spieler* 40⟩ ≡

```
⟨definiere Funktion zum ein-Spieler-Spiel: play.mad.one() 39⟩
n.sim <- 20; set.seed(13)
result <- matrix(0,4,n.sim)
for(i in 1:n.sim){
  result[,i] <- play.mad.one(w.max=30,nr=i,sleep=0.001)
}
t(result)
```

R: In diesem Beispiel sehen wir eine for*-Schleife. In den runden Klammern*

wird die Menge der Werte festgelegt, die die Variable `i` *nach und nach annehmen soll. In dem Beispiel sind es gerade die Zahlen von 1 bis 20. Im folgenden Schleifenrumpf ist beschrieben, was in jedem Schleifendurchgang passieren soll. Dabei wird die Variable* `i` *immer einen anderen Wert besitzen.* `t()` *transponiert eine Matrix.*

Wir erhalten folgendes Ergebnis:

```
      [,1] [,2] [,3] [,4]
 [1,]    3   -1   31    1
 [2,]   -1    4   24    1
 [3,]   28    3    5    0
 [4,]   -2    4   27    1
 [5,]   -2    2   28    1
 [6,]   -3   39    2    1
 [7,]   -2    2   31    1
 [8,]   -1   14    2    0
 [9,]   38    4    7    1
[10,]   32    3    0    0
[11,]   -1    2   15    0
[12,]   -4    0    0    0
[13,]   -3    2   27    1
[14,]   -4   -3    2    1
[15,]   -3    6   37    1
[16,]   -1    2   32    1
[17,]   -1    6   40    1
[18,]   -4    2   11    1
[19,]   -4    3   16    1
[20,]   -2   27    7    0
```

Eine kleine Aufbereitung verdeutlicht die Ergebnisse.

R: `apply()`, `sort()`, `colnames()`, `rownames()`, `print()`, `noquote()`

41 ⟨*erledige eine einfache Simulation mit nur 30 Würfen* 41⟩ ≡

```
a <- result; a[a<0] <- 99; a <- apply(a,2,sort)
a[a==0] <- ""; a[a=="99"] <- "*"
colnames(a) <- 1:20; rownames(a) <- rep("Kegel:",4)
print(noquote(a[4:1,]))
```

R: In dieser Auswertung sind einige R-Leckerbissen zu sehen. Die Funktion `sort()` *sortiert einen Vektor.* `apply()` *wendet die Operation Sortieren auf jede Spalte der Matrix* `result` *an. Mit* `colnames()` *und* `rownames()` *werden die Namen von Spalten und Zeilen einer Matrix gesetzt.* `print()` *ist die universelle Funktion zum Ausdruck von Objekten.* `noquote()` *wird unterstützend eingesetzt, um im Ausdruck die Tüddelchen verschwinden zu lassen.*

Als Output bekommen wir die Positionen der Kegel für die 20 Läufe jeweils nach Beendigung der 30 Würfel-Würfe. `*` zeigen Kegel im Ziel an.

```
       1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
Kegel: *  *  28 *  *  *  *  *  38 32 *  *  *  *  *  *  *  *  *  *
Kegel: 31 24 5  27 28 39 31 14 7  3  15    27 *  37 32 40 11 16 27
Kegel: 3  4  3  4  2  2  2  2  4     2     2  2  6  2  6  2  3  7
Kegel: 1  1     1  1  1  1     1           1  1  1  1  1  1  1
```

Wir wollen noch einen experimentellen Vorschlag wagen, die Laufergebnisse zu bewerten. Dazu werden einem Kegel im Ziel 50 Punkte zugewiesen, die Kegel auf dem Brett bekommen die Feldnummern als Punkte gutgeschrieben.

R: `density()`, `summary()`

42 ⟨*werte einfaches Spiel aus* 42⟩ ≡

```
res <- sort(apply(result*(result>0),2,sum)+apply((result<0)*50,2,sum))
plot(density(res), main="Dichtespur")
print(res)
summary(res)
```

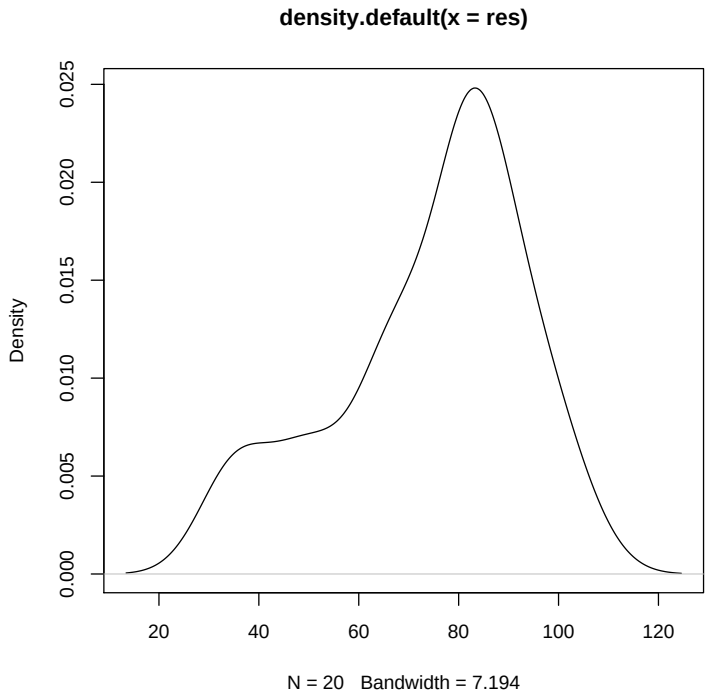
R: `density()` *zeichnet eine Dichtespur zu den genannten Daten, wogegen* `summary()` *einige zusammenfassende Statistiken eines Datensatzes berechnet.*

Dieses liefert folgende nummerischen Ergebnisse ...

```
[1]  35  36  50  50  64  66  67  70  79  80  81  82  84  84  85  85  92  94  97 103

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   35.0    65.5    80.5    74.2    85.0   103.0
```

... sowie das Bild:



Wir haben damit ein Spiel geschaffen, mit dem man schon einige Fragen zu dem Spiel *Mensch ärgere Dich nicht!* beantworten kann. Doch interessiert noch mehr ein Spiel mit mehreren Spielern.

7 *Mensch ärgere Dich nicht!* für mehrere Spieler

7.1 Vorüberlegungen

Aus Sicht der R-Technik dürften für das allgemeine Spiel gar nicht so viel neue Dinge auftauchen. Auch ist zu erwarten, dass die grobe Struktur weiter funktioniert. Welche Probleme sind gegenüber der letzten Version für den Fall mehrerer Spieler zu lösen?

- Für jeden Spieler müssen Steine verwaltet werden. Im Prinzip müsste die Lösung für einen Spieler auf die anderen übertragbar sein, jedoch liegen nur für den ersten Spieler die Zielfelder direkt hinter Feld Nummer 40.
- Bei mehreren Spielern muss die Abfolge verwaltet werden, nach denen diese würfeln dürfen. Dabei ist zu beachten, dass man nach einer 6 noch einmal würfeln darf, und falls kein Stein auf dem Feld bewegt werden kann, drei Versuche zum Heraussetzen erlaubt sind.
- Der spannenste Punkt besteht darin, dass man fremde Spieler herauskegeln muss, wenn dieses geht. Damit erfordert der Punkt *Auswahl des nächsten Spielers* besondere Konzentration.

7.2 Die Struktur des Spiels

Beginnen wir gleich wieder mit dem strukturierenden Code-Chunk. Er zeigt, dass die grobe Struktur unverändert geblieben ist. Um die Übersicht nicht zu verlieren, haben wir größere semantische Einheiten zu eigenen Chunks zusammen gebunden. Insgesamt ist vieles gleich geblieben, doch der Teufel steckt – wie immer – im Detail.

43 ⟨*spiele Spiel mit mehreren Spielern* 43⟩ ≡ ⊂ 67, 68, 74, 88

```
⟨generiere Felder 2⟩
⟨stelle Spielfeld dar 6⟩
⟨initialisiere Spiel mit mehreren Spielern 50⟩
melde("Initialisieren")
repeat{
  ⟨berichte über Zustand 44⟩
  ⟨verhindere Endlosschleife 25⟩
  if(fertig) break else counter<-counter+1
  ⟨bestimme Spieler, der dran ist 51⟩
  ⟨würfle den Würfel 20⟩
  ⟨handle Ärgern ab 46⟩
  ⟨handle Neuen-Kegel-Heraussetzen ab 47⟩
  ⟨wähle Kegel zum Setzen bei mehreren Spielern 56⟩
  ⟨handle Kegel-ins-Ziel-Bringen ab 48⟩
  ⟨handle Normales-Weitersetzen ab 49⟩
  next
}
```

In verschiedenen Situation sind Infos über den Prozess von nutzen.

44 ⟨*berichte über Zustand* 44⟩ ≡ ⊂ 43, 74, 80

```
cat("Counter:",counter,"/Spieler:", akt.player,"/Wurf:",WUERFEL,
    "\n  Kegel",kegel)
```

Es erschien eleganter, die Meldung nach einem Wurf dem passenden Chunk anzuhängen.

45 ⟨*würfle den Würfel* 20⟩+ ≡ ⊂ 9, 24, 43
```
melde("Wurf",WUERFEL,"count:",counter,"Player:",akt.player)
```

Das Spiel mit mehreren Spielern macht nur Sinn, wenn mindestens zwei Akteure an den Start gehen. Die Spielerzahl wird auf der Variablen `n.player` vermerkt.

Ärgern. Ein Vergleich mit der zweiten Version zeigt, dass es auf dieser Abstraktionsstufe keine Änderungen gegeben hat – abgesehen davon, dass wir uns auf den aktuellen Spieler `akt.player` beziehen müssen.

46 ⟨*handle Ärgern ab* 46⟩ ≡ ⊂ 43, 65, 80
```
if( WUERFEL != 6 && ⟨muss Spieler neuen Kegel einbringen 52⟩ ){
  cat("Info: ",akt.player,"denkt: Mensch aergere Dich!"); next
}
melde("Aegern","w.max",w.max,"akt.player",akt.player)
```

Kegel Heraussetzen. Bei der Frage des Heraussetzens eines neuen Kegels taucht die Novation auf, dass eventuell ein Kegel eines anderen Spielers entfernt werden muss.

47 ⟨*handle Neuen-Kegel-Heraussetzen ab* 47⟩ ≡ ⊂ 43, 65, 80
```
if( WUERFEL == 6 && ⟨kann Spieler neuen Kegel einbringen 53⟩ ){
  ⟨wähle einen Kegel aus Vorrat des Spielers aus 54⟩
  ⟨entferne ggf. fremden Kegel vom Brett 63⟩
  ⟨stelle gewählten Kegel des Spielers auf das Startfeld 55⟩
  cat("Info:", akt.player,"ruft: Juchhu 6 gefallen! Kegel",
      idx,"raus!"); next
}
melde("Raussetzen")
```

Kegel ins Ziel verschieben. Die Umsetzung des Verschiebens eines Kegels in den Zielbereich weist im Groben keine Besonderheiten auf. Unterschiede ergeben sich indes bei der Feinarbeit.

48 ⟨*handle Kegel-ins-Ziel-Bringen ab* 48⟩ ≡ ⊂ 43, 65, 80
```
if( ⟨ist neues Feld im Ziel des aktuellen Kegels und erlaubt 58⟩ ){
  ⟨entferne aktuellen Kegel 60⟩
  ⟨stelle aktuellen Kegel ins Ziel 61⟩
  cat("Info:",akt.player,"ruft: Kegel",idx,"gerettet!")
  if(sum(kegel[,akt.player]<0)==4){
    cat("Info:", akt.player,"ruft: geht doch / geschafft!")
    fertig<-TRUE; print(paste("dice counter:",counter))
  }
  next
}
melde("Kegel ins Ziel stellen")
```

Weitersetzen. Auch beim normalen Weitersetzen sieht alles ganz einfach aus:

49 ⟨*handle Normales-Weitersetzen ab* 49⟩ ≡ ⊂ 43, 65, 80

```
if( ⟨ist Zielfeld des aktuellen Kegels auf Spielfeld 59⟩ ){
  ⟨entferne aktuellen Kegel 60⟩
  ⟨entferne ggf. fremden Kegel vom Brett 63⟩
  ⟨stelle aktuellen Kegel auf neue Position 62⟩
  next
}
melde("Vorsetzen")
```

7.3 Die Initialisierung relevanter Größen

Was muss während der Initialisierung erledigt werden? Das Objekt `kegel` wird als Matrix angelegt, deren Spalten zu den einzelnen Spielern gehören und deren i-te Zeile dem jeweiligen i-ten Kegel zugeordnet ist. Die verwendeten Farben werden auf `cols.player` für die Felder und `cols.player.kegel` für die Kegel vermerkt. Konstante Fakten sollten immer nur an einer Stelle kodiert werden, so dass beispielsweise Farbänderungen schnell möglich sind. Mit den `cols`-Variablen wird diese Empfehlung erfüllt. Die Start- und Ziel-Info-Matrizen werden zur Erleichterung von Zugriffen auf Listen zusammengefasst. In der vorliegenden Version werden die vorbereiteten Spalten für Zustandsinformationen übrigens nicht mehr benutzt. Zur Verwaltung des Spielers, der aktuell an der Reihe ist, wird die Variable `akt.player` eingeführt. `anz.keine.sechs` übernimmt die Rolle des Zählens der Nicht-Sechsen. Mit der Variablen `counter` zählen wir, wie viele Würfel-Würfe durchgeführt worden sind. `fertig` wird anzeigen, wenn eine Ende-Bedingung für die Simulation erfüllt ist. `n.player` zeigt uns die Anzahl der Spieler an und auf `WUERFEL` wird wieder das letzte Würfelergebnis gespeichert werden.

R: `list()`, `[[]]`

50 ⟨*initialisiere Spiel mit mehreren Spielern* 50⟩ ≡ ⊂ 43, 64, 65, 80

```
kegel <- matrix(0,4,4)
cols.player <- c("blue","yellow","green","red")
cols.player.kegel <- c("lightblue","lightgoldenrodyellow",
                       "yellowgreen","mediumvioletred")
vorrat    <- list(blue.start,yellow.start,green.start,red.start)
ziel.koor <- list(blue.ziel, yellow.ziel, green.ziel, red.ziel)
for(i in 1:4){
  if(n.player >= i){
    idx <- c(1,3,2,4)[i]
    make.circle(vorrat[[idx]],col=cols.player.kegel[idx],size=3)
  }
}
akt.player <- 0; anz.keine.sechs <- 0; fertig <- FALSE
⟨setze Wartezeit sleep für play.mad 83⟩
counter <- 0; WUERFEL <- 0
⟨stelle Funktion melde bereit 27⟩
```

R: Listen sind Vektoren, deren Elemente Objekte von eventuell unterschiedlichem Typ sind. Sie werden generiert mit der Funktion `list()`*. Einzelne Elemente einer Liste erhält man durch Indexzugriffe mit doppelten eckigen Klammern.*

7.4 Die Bestimmung des nächsten Spielers

Auf `akt.player` ist die Nummer des aktuellen Spielers vermerkt; zu Beginn ist dort eine 0 abgelegt, so dass der Spielanfang erkennbar ist. Im Regelfall wird der jeweils nächste Spieler gewählt. Doch gilt dies nicht, wenn vorher eine 6 gewürfelt worden war. Dann bleibt der alte Spieler an der Reihe. Hat ein Spieler alle Kegel im Vorrat, darf er dreimal für eine 6 würfeln. `new.player` ist ein Flag, das zu Beginn der Spielerbestimmung auf `TRUE` gesetzt wird. Falls ein Spieler noch einmal würfeln darf, wird dieses Flag auf `FALSE` gesetzt. Damit ergibt sich eine schöne Zweiteilung: Erst wird die Situation analysiert, dann kommt der Handlungsteil. R: `ifelse()`

51 ⟨*bestimme Spieler, der dran ist* 51⟩ ≡ ⊂ 43, 80

```
if(akt.player==0) akt.player <- 1 else {
  new.player <- TRUE
  if(WUERFEL==6){ new.player <- FALSE }
  if(WUERFEL <6){
    if(all(kegel[,akt.player]<=0)){
      # Anzahl der bisherigen Fehl-Versuche inkrementieren:
      anz.keine.sechs <- anz.keine.sechs + 1
      if(anz.keine.sechs < 3) {
        new.player <- FALSE
      } else {                      # schon 3 x dran
        anz.keine.sechs <- 0
      }
    }
  }
  if(new.player){
     if(n.player==2) akt.player <- ifelse(akt.player==1, 3, 1)
     if(n.player==3) akt.player <- 1 + (akt.player%%n.player)
     if(n.player==4) akt.player <- 1 + (akt.player%%n.player)
  }
}
melde("akt.player:",akt.player)
```

R: `ifelse()` *wird mit drei gleich langen Vektoren aufgerufen und erzeugt einen Vektor gleicher Länge. Das erste Argument ist ein Vektor aus Wahrheitswerten, mit dessen Hilfe entschieden wird, ob für die Konstruktion des neuen Vektors ein Element aus dem zweiten oder dem dritten Input verwendet werden soll.* Übrigens dürfen bei 2 Spielern die Spieler 1 und 3 spielen, nicht aber der 2.

7.5 Heraussetzen neuer Kegel

Für das Heraussetzen-Müssen muss gegenüber dem einfachen Spiel nur der Indexzugriff angepasst werden.

52 ⟨*muss Spieler neuen Kegel einbringen* 52⟩ ≡ ⊂ 46

```
any(0==kegel[,akt.player]) &&
  !any(kegel[,akt.player] %in% 1:40)
```

Auch folgender Chunk ist bis auf die Index-Zugriffe bekannt.

53 ⟨*kann Spieler neuen Kegel einbringen* 53⟩ ≡ ⊂ 47

```
any(0==kegel[,akt.player]) &&
  all(kegel[,akt.player]!=c(1,11,21,31)[akt.player])
```

Dto.

54 ⟨*wähle einen Kegel aus Vorrat des Spielers aus* 54⟩ ≡ ⊂ 47

```
idx <- (1:4)[0==kegel[,akt.player]][1]
make.circle(vorrat[[akt.player]][idx,],
            col=cols.player[akt.player],txt=idx)
melde("Kegelauswahl","found: ------------",idx)
```

Die Startfelder der einzelnen Spieler sind unterschiedlich, nämlich 1, 11, 21 und 31. Das ist zu beachten.

55 ⟨*stelle gewählten Kegel des Spielers auf das Startfeld* 55⟩ ≡ ⊂ 47

```
kegel[idx,akt.player] <- c(1,11,21,31)[akt.player]
make.circle(fields[kegel[idx,akt.player],],
            col=cols.player.kegel[akt.player],size=3,txt=idx)
```

7.6 Abwicklung eines normalen Zuges

Für die Abwicklung eines normalen Zuges müssen die relevanten Regeln und eine Strategie umgesetzt werden. Dazu ermitteln wir die Nummer des Kegels `idx`, der zu setzen ist, nach folgenden Überlegungen ermittelt. Die Wirkungen einer 6 mit dem Einbringen eines neuen Kegels werden an dieser Stelle als schon erledigt angesehen.

1. Es wird geprüft, welche Kegel sich überhaupt bewegen dürfen.
2. Falls man Kegel ins Ziel bringen kann, werden diese Züge vorgezogen.
3. Falls ein anderer Spieler geschlagen werden kann, muss ein Zug ins Ziel verschoben werden.
4. Die schon genannten Regeln werden dominiert durch den Fall, dass sich mindestens ein Kegel im Vorrat befindet, ein Kegel auf dem Startfeld steht und ziehen kann.

R: `&`

56 ⟨*wähle Kegel zum Setzen bei mehreren Spielern* 56⟩ ≡ ⊂ 43, 65, 80

⟨*bestimme mögliche neue Felder* 57⟩

```
# Verarbeitung eines neuen Kegels ist bereits erledigt
anz.ok<-sum(zug.erlaubt); if(anz.ok==0) next
# Zufallsauswahl unter erlaubten Kegelbewegungen
idx<-which(zug.erlaubt)[sample(1:anz.ok,1)]
# Kegel ins Ziel retten: geht vor
if(any(cand.ziel)){ idx<-which(cand.ziel)[1] }
# Schlagen: geht vor
if(any(h<-(brett.feld[cand.brett] %in% kegel[,-akt.player]))){
  idx<-(1:4)[cand.brett][h][1]
}
# Zuerst ggf. Platte putzen: diese Regel dominiert die davor
if(any(kegel[,akt.player]==1 & zug.erlaubt)){
  idx<-which(kegel[,akt.player]==1 & zug.erlaubt)
}
melde("Kegel wählen",kegel,"Kegelnummer:",idx)
```

R: Den Operator `&&` *hatten wir bereits oben kennen gelernt. Er verknüpft zwei Wahrheitswerte mit einer* Und-*Operation. Das einfache* `&`*-Zeichen führt elementweise die* Und-*Operation für zwei logische Vektoren durch, so dass ein gleich langer neuer Vektor aus Wahrheitswerten entsteht.*

Voraussetzung für die letzten Überlegungen ist eine geeignete Vorarbeit. Bei dieser werden eine Reihe von Charakteristika der Situation erhoben, insbesondere welches die neuen, potentiellen Felder der Kegel sind. Genauer werden die in folgender Auflistung aufgeführten Variablen konstruiert. Es sei darauf hingewiesen, dass 0-Einträge immer negative Erkenntnisse kodieren wie: nicht zulässig, nicht erreichbar, nicht möglich, kein erlaubtes Feld vorhanden usw.

- `zug.erlaubt` zeigt, welche Kegel bewegt werden dürfen.
- `cand.brett` zeigt, ob sich Figuren auf ein normales Feld des Brettes ziehen dürfen.
- `cand.ziel` zeigt, ob sich Kegel in den Zielbereich bewegen dürfen.
- `cand.start` zeigt, ob sich Kegel auf das Startfeld bewegen dürfen.
- `ziel.feld` zeigt Nummern der Felder im Zielbereich an.
- `brett.feld` zeigt Nummern der Felder unter den normalen Brett-Feldern an.

57 ⟨*bestimme mögliche neue Felder* 57⟩ ≡ ⊂ 56

```
limit.ziel <- c(40,10,20,30)[akt.player]
# Betrachtung von Kegeln auf dem Brett
brett.feld <- ifelse(0<kegel[,akt.player],WUERFEL+kegel[,akt.player],0)
# Betrachtung von Kegeln aus dem Vorrat
if(WUERFEL==6){
  brett.feld <- ifelse(kegel[,akt.player]==0,
                       1+(limit.ziel%%40),brett.feld)
}

# Kandidaten -> Zielfeld
cand.ziel  <- 0 < kegel[,akt.player] &     ## Kegel auf Normalfeld?
              kegel[,akt.player] <= limit.ziel & ## Kegel vor Ziel?
              limit.ziel < brett.feld          ## Zielfeld im Ziel?
ziel.feld  <- (brett.feld*cand.ziel) %% 10     ## evtl. Zielfeld-Nr
# Zielkandidaten: kein Brettzug
brett.feld <- ifelse(cand.ziel,0,brett.feld)
ziel.feld  <- (ziel.feld %in% 1:4) * ziel.feld    ## Ziel existent?
cand.ziel  <- cand.ziel & ziel.feld!=0
cand.ziel  <- cand.ziel & !(ziel.feld %in% -kegel[,akt.player])
ziel.feld  <- ifelse(cand.ziel,ziel.feld,0)
# Kandidaten -> Brett
cand.brett <- (!(brett.feld %in% kegel[,akt.player]))&(brett.feld!=0)
brett.feld <- ifelse(cand.brett,1+((brett.feld - 1)%%40),0)
cand.start <- cand.brett & kegel[,akt.player]==0
# Zug erlaubt?
zug.erlaubt <- cand.ziel | cand.brett
```

Der gerade vorgestellte Code-Chunk erwies sich als sehr fehlerträchtig. Deshalb sollte am besten das zu lösende Teilproblem noch einmal mit noch klareren Strukturen gelöst werden. Ein alternativer Ansatz ist im Anhang skizziert.

7.7 Bedingungen

Es sind noch zwei Bedingungs-Chunks zu formulieren. Hierbei zeigt `idx` den ausgewählten Kegel an.

58 ⟨*ist neues Feld im Ziel des aktuellen Kegels und erlaubt* 58⟩ ≡ ⊂ 48
```
cand.ziel[idx]
```

59 ⟨*ist Zielfeld des aktuellen Kegels auf Spielfeld* 59⟩ ≡ ⊂ 49
```
cand.brett[idx]
```

7.8 Aktionen

Gegenüber den Vorarbeiten wirken die Umsetzungen der Aktionen so einfach, dass sie keiner weiteren Kommentierung bedürfen.

60 ⟨*entferne aktuellen Kegel* 60⟩ ≡ ⊂ 48, 49
```
make.circle(fields[kegel[idx,akt.player],],
            col=col.fields[kegel[idx,akt.player]],
            txt=kegel[idx,akt.player])
melde("Kegelentfernung","idx::",idx)
```

61 ⟨*stelle aktuellen Kegel ins Ziel* 61⟩ ≡ ⊂ 48
```
kegel[idx,akt.player] <- -ziel.feld[idx]
make.circle(hh<-ziel.koor[[akt.player]][ziel.feld[idx],],
            col=cols.player.kegel[akt.player],size=3,txt=idx)
```

62 ⟨*stelle aktuellen Kegel auf neue Position* 62⟩ ≡ ⊂ 49
```
kegel[idx,akt.player]<- 1+((kegel[idx,akt.player]+WUERFEL-1) %% 40)
make.circle(fields[kegel[idx,akt.player],],
            col=cols.player.kegel[akt.player],size=3,txt=idx)
```

7.9 Das Schlagen anderer Kegel

Das, was das Spiel würzt, ist bis zum Schluss aufgespart worden: das Schlagen fremder Kegel. Falls die Nummer des neuen Feldes eines zulässigen Zuges mit einer Nummer übereinstimmt, die auf `kegel` vermerkt ist, muss dort ein fremder Kegel stehen. In diesem Fall werden Halter und Kegelnummer ermittelt, das Feld optisch zurückgesetzt und der Kegel in seinen Vorratsraum zurückgestellt. Zum Schluss wird die Kegelverwaltung aktualisiert.

63 ⟨*entferne ggf. fremden Kegel vom Brett* 63⟩ ≡ ⊂ 47, 49
```
neu.feld <- WUERFEL+kegel[idx,akt.player]
if(any(neu.feld==kegel)){
  cat("ACHTUNG: Kegel fliegt!")
  player.pech <-(1:4)[apply(neu.feld == kegel, 2,any)][1]
  kegel.pech  <-(1:4)[apply(neu.feld == kegel, 1,any)][1]
```

```
  # praktischer Rauswurf (verzichtbar)
  make.circle(fields[kegel[kegel.pech,player.pech],],
              col=col.fields[neu.feld],txt=neu.feld)
  melde("Kegel-Rauswurf","idx::",idx)
  # Integration in den Vorrat
  make.circle(vorrat[[player.pech]][kegel.pech,],
              col=cols.player.kegel[player.pech],size=3,txt=kegel.pech)
  # Verwaltung aktualisieren
  kegel[kegel.pech,player.pech] <- 0
  melde("Kegel-Rauswurf umgesetzt","Feld/player.pech/kegel.pech",
        neu.feld,player.pech,kegel.pech)
}
```

7.10 Testerei

In einem solchen Spiel gibt es schon verschiedene Möglichkeiten, Fehler zu machen. Deshalb sollte man – und sei es zur Absicherung – Testsituationen überlegen und checken.

Zunächst entwerfen wir einen Chunk, um dass Spielfeld in Abhängigkeit von verschiedenen Spieleranzahlen zu testen:

64 ⟨*checke Aufbau* 64⟩ ≡

```
n.player <- 2
⟨generiere Felder 2⟩
⟨stelle Spielfeld dar 6⟩
⟨initialisiere Spiel mit mehreren Spielern 50⟩
```

Für den Test ganz spezifischer Situationen stellen wir noch einmal den Kern des Spiel als kleine Testumgebung so bereit, dass auf Basis einer Kegelkonstellation ein einziger Würfelwurf umgesetzt wird.

65 ⟨*checke Testsituation* 65⟩ ≡

```
w.max <- n.player <- 2
⟨stelle Spielfeld dar 6⟩
⟨initialisiere Spiel mit mehreren Spielern 50⟩
##kegel <- cbind(c(2,39,-4,-3), 0,0,0); ##kegel <- kegel1
kegel <- cbind(c(3,0,0,0),0,c(21,0,0,0),0)
WUERFEL <- 6; akt.player <- 3
⟨aktualisiere Darstellung des Spielbrettes 66⟩; print(kegel)
melde("Initialisieren")
for(i in 1){
  cat("Counter:",counter,"/Spieler:", akt.player,"/Wurf:",WUERFEL,
      "\n  Kegel",kegel)
  melde("Wurf",WUERFEL,"count:",counter,"Player:",akt.player)
  ⟨handle Ärgern ab 46⟩
  ⟨handle Neuen-Kegel-Heraussetzen ab 47⟩
  ⟨wähle Kegel zum Setzen bei mehreren Spielern 56⟩
  melde("Kegel wählen",kegel)
  ⟨handle Kegel-ins-Ziel-Bringen ab 48⟩
  ⟨handle Normales-Weitersetzen ab 49⟩
}
kegel
```

Das Ergebnis der Testsituation muss natürlich graphisch dargestellt werden. Dazu zeichnet folgender Chunk ein Brett mit Kegeln ausgehend von `kegel`.

66 ⟨*aktualisiere Darstellung des Spielbrettes* 66⟩ ≡ ⊂ 65, 89

```
for(i in 1:4){ # Kegel-Nummer
  for(j in 1:4){ # Spieler
    # Kegel auf dem Brett
    if(0 < kegel[i,j]){
      make.circle(fields[kegel[i,j],],
                  col=cols.player.kegel[j],size=3,txt=i)
    }
    # Kegel im Vorrat
    make.circle(vorrat[[j]][i,],col=cols.player[j],txt=i)
    if(kegel[i,j] == 0){
      if(j==1 || (j==2 & 2<n.player) || j==3 || (j==4 & 4==n.player)){
        make.circle(vorrat[[j]][i,],
                    col=cols.player.kegel[j],size=3,txt=i)
      }
    }
    # Kegel im Ziel
    if(kegel[i,j] < 0){
      pos<- -kegel[i,j]
      make.circle(ziel.koor[[j]][pos,],
                  col=cols.player.kegel[j],size=3,txt=i)
    }
  }
}
```

Bevor Simulationen mit dem Spiel losgehen können, sollten wir es erst einmal testen.

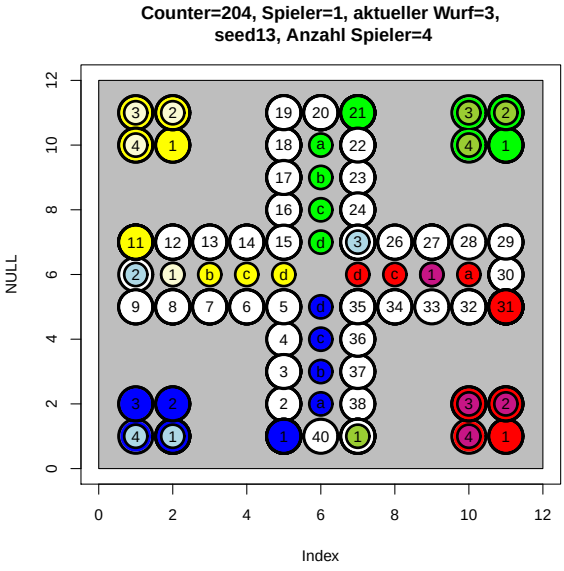
67 ⟨*teste das Spiel mit mehreren Spielern* 67⟩ ≡

```
w.max <- 205; n.player <- 4; Debug <- !TRUE; seed=13
set.seed(seed); sleep <- 0.01
⟨setze Wartezeit sleep 82⟩
⟨spiele Spiel mit mehreren Spielern 43⟩
title(paste("Counter=",counter,", Spieler=",akt.player,
            ", aktueller Wurf=",WUERFEL,
            ",\n seed",seed,", Anzahl Spieler=",n.player,sep=""))
list(Counter=counter,Spieler=akt.player,Wuerfel=WUERFEL,Kegel=kegel)
```

Dieser Testaufruf verschafft uns einen Eindruck von dem Meldungsprotokoll ohne weitere Debug-Infos. Schauen wir uns einmal den letzten Teil des Protokolls zum letzten Aufruf an:

```
Counter: 190 /Spieler: 4 /Wurf: 4
  Kegel 0 0 20 0 -1 24 0 0 31 25 0 0 -2 36 35 0
Info: 1 ruft: Juchhu 6 gefallen! Kegel 1 raus!
Counter: 191 /Spieler: 1 /Wurf: 6
  Kegel 1 0 20 0 -1 24 0 0 31 25 0 0 -2 36 35 0
Counter: 192 /Spieler: 1 /Wurf: 5
  Kegel 6 0 20 0 -1 24 0 0 31 25 0 0 -2 36 35 0
ACHTUNG: Kegel fliegt!
Counter: 193 /Spieler: 2 /Wurf: 1
  Kegel 6 0 20 0 -1 25 0 0 31 0 0 0 -2 36 35 0
ACHTUNG: Kegel fliegt!
Counter: 194 /Spieler: 3 /Wurf: 4
  Kegel 6 0 20 0 -1 25 0 0 35 0 0 0 -2 36 0 0
Counter: 195 /Spieler: 4 /Wurf: 3
```

```
  Kegel 6 0 20 0 -1 25 0 0 35 0 0 0 -2 39 0 0
ACHTUNG: Kegel fliegt!
Counter: 196 /Spieler: 1 /Wurf: 5
  Kegel 6 0 25 0 -1 0 0 0 35 0 0 0 -2 39 0 0
Info:  2 denkt: Mensch aergere Dich!
Counter: 197 /Spieler: 2 /Wurf: 5
  Kegel 6 0 25 0 -1 0 0 0 35 0 0 0 -2 39 0 0
ACHTUNG: Kegel fliegt!
Counter: 198 /Spieler: 3 /Wurf: 4
  Kegel 6 0 25 0 -1 0 0 0 39 0 0 0 -2 0 0 0
Info:  4 denkt: Mensch aergere Dich!
Counter: 199 /Spieler: 4 /Wurf: 5
  Kegel 6 0 25 0 -1 0 0 0 39 0 0 0 -2 0 0 0
Info:  4 denkt: Mensch aergere Dich!
Counter: 200 /Spieler: 4 /Wurf: 3
  Kegel 6 0 25 0 -1 0 0 0 39 0 0 0 -2 0 0 0
Info:  4 denkt: Mensch aergere Dich!
Counter: 201 /Spieler: 4 /Wurf: 1
  Kegel 6 0 25 0 -1 0 0 0 39 0 0 0 -2 0 0 0
ACHTUNG: Kegel fliegt!
Info: 1 ruft: Juchhu 6 gefallen! Kegel 2 raus!
Counter: 202 /Spieler: 1 /Wurf: 6
  Kegel 0 1 25 0 -1 0 0 0 39 0 0 0 -2 0 0 0
Counter: 203 /Spieler: 1 /Wurf: 6
  Kegel 0 7 25 0 -1 0 0 0 39 0 0 0 -2 0 0 0
$Counter
[1] 204
$Spieler
[1] 1
$Wuerfel
[1] 3
$Kegel
     [,1] [,2] [,3] [,4]
[1,]    0   -1   39   -2
[2,]   10    0    0    0
[3,]   25    0    0    0
[4,]    0    0    0    0
```



Das Bild zeigt den Zustand auf dem Brett nach 204 Würfelwürfen.

7.11 Eine einfache Spielfunktion

Für weitere Tests und für Simulationen muss das Spiel noch in Form einer Funktion bereitgestellt werden. Sie soll `play.mad` heißen.

`play.mad`	erlaubt ein Spiel zu spielen
Input:	
`n.player`	Anzahl der Spieler
`w.max`	maximale Anzahl von Würfelwürfen
`Debug`	zum Setzen der Variablen `Debug`
`sleep`	Pausierzeit pro Schleifendurchlauf
`seed`	Start des Zufallszahlengenerators
`NO.OUT`	falls `TRUE` werden Ausgaben unterbunden
`sleep.slider`	falls `TRUE` lässt sich Wartezeit interaktiv setzen
Output:	
—	Text-Ausgaben auf den Bildschirm
—	Positionen der Kegel

68 ⟨*definiere Funktion zum MAD-Spielen:* `play.mad()` 68⟩ ≡ ⊂ 69, 70, 87

```
play.mad <<- function(n.player = 2, w.max = 1000, Debug = !TRUE,
                      sleep = 0.2, seed = 13, NO.OUT = !FALSE,
                      sleep.slider = TRUE){
  set.seed(seed)
  ⟨spiele Spiel mit mehreren Spielern 43⟩
  title(paste("Counter=",counter,", winner=",akt.player,
              ", last throw=",WUERFEL,
              ",\n seed",seed,", n.players=",n.player,sep=""))
  return(list(Counter=counter,Spieler=akt.player,
              Wuerfel=WUERFEL,Kegel=kegel))
}
```

Folgende Situationen führten zu Fehlern. Deshalb wollen wir diese mit der gerade definierten Funktion ausprobieren.

69 ⟨*teste Mehrspielerspiel mit 2 Spielern* 69⟩ ≡

```
⟨definiere Funktion zum MAD-Spielen: play.mad() 68⟩
play.mad(w.max = 100, n.player = 2, Debug = !TRUE, sleep = 0.02, seed=13)
# play.mad(seed=14,n.player=2,w.max=150,NO.OUT=TRUE)
# play.mad(seed=14,n.player=4,w.max=150,NO.OUT=TRUE)
```

8 Simulations-Experimente mit dem Spiel

8.1 Simulationen zur Ermittlungen von Spieldauern

Jetzt wollen wir uns endlich daran machen, einige Fragen zum Spiel per Simulation anzugehen.

Wie lange muss man bis zum Ende des Spiels im Durchschnitt würfeln? Zur Beantwortung dieser Frage können wir die Funktion zur Simulation mehrmals aufrufen und die Ergebnisse graphisch und nummerisch auswerten.

70 ⟨*definiere Experiment zur Ermittlung von Spieldauern:* `exp.mad.dauern()` 70⟩ ≡ ⊂ 71, 72, 73, 87

```
exp.mad.dauern <<- function(wd=5,w.max=300,n.player=2){
  ⟨definiere Funktion zum MAD-Spielen: play.mad() 68⟩
  result<-NULL
  make.no.circles<-TRUE
  for(i in 1:wd){
    print(i)
    res<-play.mad(seed=i,n.player=n.player,w.max=w.max,NO.OUT=TRUE,
                  sleep=.001,sleep.slider = FALSE)
    result<-c(result,res$Counter)
  }
  opar <- par(mfrow=c(2,2),bty="n")
  plot(NULL,ylim=c(0,max(result)),xlim=c(0,wd+1),bty="n",
       xlab="index",ylab="tosses")
  segments(seq(wd),rep(0,wd),seq(wd),result)
  boxplot(result,horizontal=TRUE,
          ylim=c(0.9*min(result),1.2*max(result)))
  rug(result)
  plot(sort(result),xlab="run",ylab="tosses -- sorted",
       xlim=c(0,wd+1),type="h")
  points(sort(result))
  hist(result,probability=TRUE,xlim=c(0.9*min(result),1.2*max(result)))
  lines(density(result))
  par(opar)
  summary(result)
}
```

Achtung: diese Simulation dauert schon recht lange.

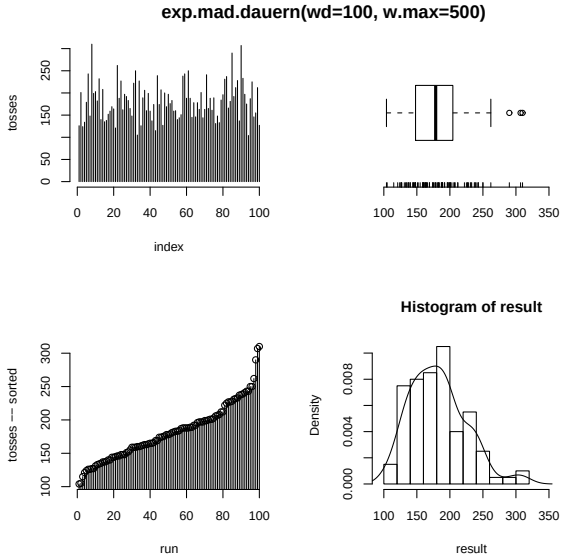
71 ⟨*ermittle Spieldauern für zwei Spieler* 71⟩ ≡

```
⟨definiere Experiment zur Ermittlung von Spieldauern: exp.mad.dauern() 70⟩
exp.mad.dauern(wd=100, w.max=500)
title("exp.mad.dauern(wd=100, w.max=600, n.player=3)")
```

Wir erhalten folgende zusammenfassenden Statistiken sowie folgende Graphiken.

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  104.0   148.0   178.5   181.9   203.8   310.0
```

Im Mittel muss man also 189 Mal würfeln, bis ein Spieler seine Kegel nach Haus gebracht hat.

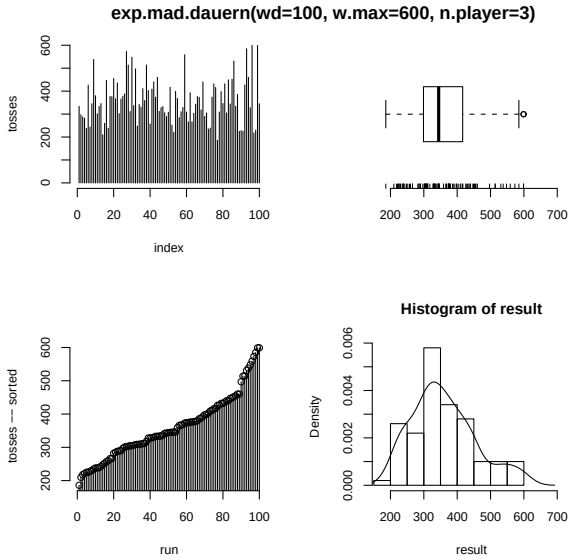


Jetzt wollen wir natürlich auch die Ergebnisse für 3 Spieler ermitteln.

72 ⟨*ermittle Spieldauern für zwei Spieler* 71⟩+ ≡
⟨*definiere Experiment zur Ermittlung von Spieldauern:* `exp.mad.dauern()` 70⟩
```
exp.mad.dauern(wd=100, w.max=600, n.player=3)
title("exp.mad.dauern(wd=100, w.max=600, n.player=3)")
```

Für drei Spieler ergeben sich, wie vielleicht erwartet, grob doppelt so lange Spieldauern.

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  186.0   300.0   344.5   359.1   416.2   599.0
```

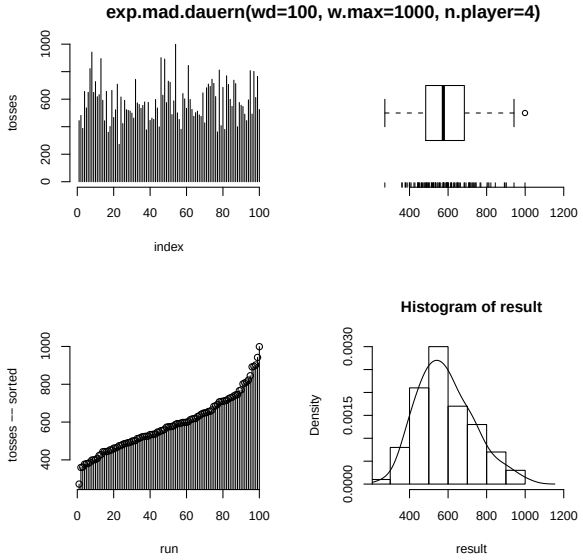


Als Krönung wird noch die Simulation für 4 Spieler durchgeführt.

73 ⟨*ermittle Spieldauern für zwei Spieler* 71⟩+ ≡
```
⟨definiere Experiment zur Ermittlung von Spieldauern: exp.mad.dauern() 70⟩
exp.mad.dauern(wd=100, w.max=1000, n.player=4)
title("exp.mad.dauern(wd=100, w.max=1000, n.player=4)")
```

Mit vier Spielern hat ein Spiel die vor festgelegte maximale Spieldauer überschritten. Nach einer Simulationszeit von zirka 15 Minuten liegt folgendes Ergebnis vor:

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 272.0   485.0   575.5   587.1   683.8   999.0
```



Im Mittel sind bei vier Spielern knapp 600 Würfe erforderlich, so dass jeder Spieler ungefähr 150 Mal würfeln muss, bis der erste Spieler alle seine Kegel im Ziel untergebracht hat. Auch ist interessant , dass die Verteilungder Dauern rechtsschief ist.

8.2 Experimente zur Visualisierung von Strähnen

Die Untersuchung von Verläufen erfordert es, dass wir während eines Simulationslaufes Daten über die Zustände erheben. Dazu definieren wir den Code-Chunk ⟨*berichte über Zustand* 44⟩ so, dass auf der Variablen `mad.verlaufs.ergebnis` nach jedem Schleifendurchlauf die Zustandsmatrizen `kegel` gesammelt werden. In dem Chunk ⟨*zeige Verlauf an* 76⟩ werden die Zustände nach Spielern getrennt dargestellt.

74 ⟨*definiere Experiment zur Verlaufsdarstellung* `exp.mad.prozess()` 74⟩ ≡ ⊂ 77, 78, 87
```
exp.mad.prozess <<- function(n.player=2, nr=13,Debug=FALSE,sleep=0.01,
                             w.max=300,NO.OUT=FALSE,report.process=TRUE){
```

```
  set.seed(nr)
  make.no.circles <- TRUE; sleep.slider <- FALSE
  ⟨spiele Spiel mit mehreren Spielern 43⟩
  ⟨zeige Verlauf an 76⟩
  ⟨berichte über Zustand 44⟩
  return(kegel)
}
```

Das Sammelobjekt ist eine Liste, die bei jedem Schleifendurchlauf verlängert wird.

75 ⟨*berichte über Zustand* 44⟩+ ≡ ⊂ 43, 74, 80

```
if(exists("report.process")){
  if(counter==0){
    mad.verlaufs.ergebnis <- NULL
  } else {
    mad.verlaufs.ergebnis[[counter]] <- kegel
  }
}
```

Die Zustände lassen sich auf sehr unterschiedliche Arten vermessen. Wir gehen hier den Weg, für jeden Spieler jeweils die Summe der zurückgelegten Felder als Indikator auszurechnen. Dabei muss man für alle bis auf den blauen Spieler die Feldnummern geeignet auf die Zahlen von 1 bis 40 abbilden. Kegel im Ziel erhalten den Betrag 41 zugeordnet.

76 ⟨*zeige Verlauf an* 76⟩ ≡ ⊂ 74

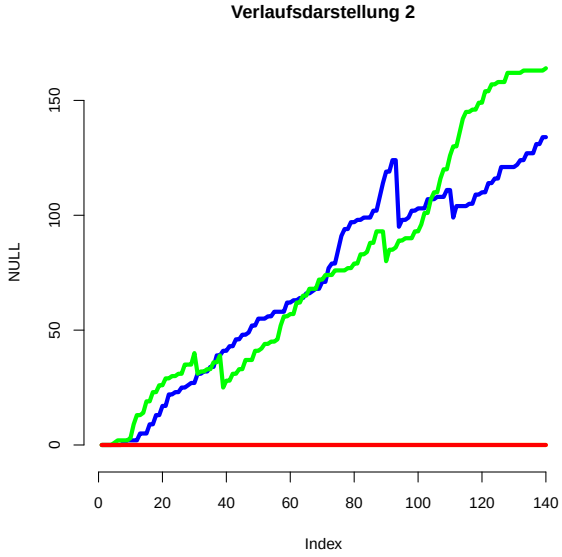
```
result <- mad.verlaufs.ergebnis
result <- lapply(result,function(x) {
                          x <- ifelse(x<0,41,x)
                          x[,2] <- ifelse(1 <= x[,2] & x[,2] <= 40,
                                          1+((x[,2]+30-1) %% 40), x[,2])
                          x[,3] <- ifelse(1 <= x[,3] & x[,3] <= 40,
                                          1+((x[,3]+20-1) %% 40), x[,3])
                          x[,4] <- ifelse(1 <= x[,4] & x[,4] <= 40,
                                          1+((x[,4]+10-1) %% 40), x[,4])
                          colSums(x)
                        }
          )
res<-t(matrix(unlist(result),nrow=4))
plot(NULL, xlim=c(1,length(result)), ylim=c(-5,max(res)),bty="n")
for(i in 1:4) lines(res[,i],col=cols.player.kegel[i],lwd=6)
lines(res[,2],lwd=1)
title(paste("Verlaufsdarstellung",n.player))
```

Die Simulation ist schnell angestoßen und ...

77 ⟨*simuliere Verläufe* 77⟩ ≡

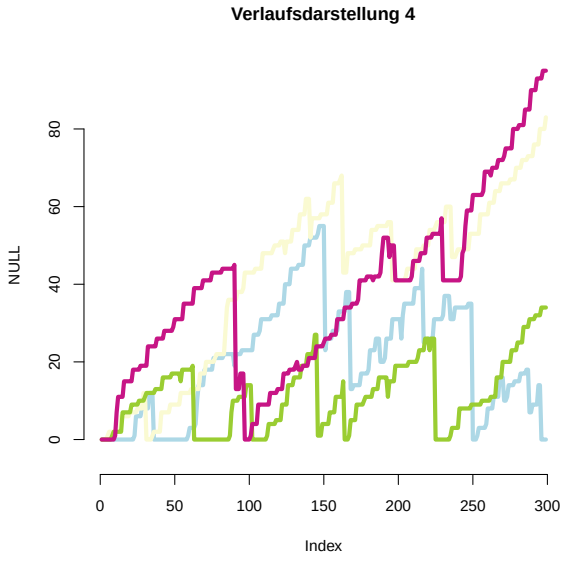
```
⟨definiere Experiment zur Verlaufsdarstellung exp.mad.prozess() 74⟩
result<-play.mad.process(NO.OUT=TRUE)
```

... wir erhalten als Ergebnis folgende exemplarischen Verläufe bei zwei Spielern:



Bei vier Spielern ergibt sich beispielsweise Folgendes:

78 ⟨*simuliere Verläufe* 77⟩+ ≡
⟨*definiere Experiment zur Verlaufsdarstellung* `exp.mad.prozess()` 74⟩
```
result<-play.mad.process(NO.OUT=TRUE,n.player=4)
```



Gerade beim letzten Bild erkennen wir, dass Rot zwei Glückssträhnen hat. Nach einem famosen Start folgt die Ernüchterung durch zweimaliges Schlagen, doch dann setzt eine lang anhaltende Glückssträhne ein, die auch zum Gewinn führt.

8.3 Fazit zur Simulation

Für die Untersuchung des Spiels *Mensch ärgere Dich nicht!* liegt nun ein Werkzeugkasten vor, mit dem wir einige kleinere Einsichten gewonnen haben. Bei Betrachtung der Simulations-Ergebnisse entstehen sofort neue Fragen und neue Ideen, die dann zu weiteren Simulationen führen. Jedoch soll das Experimentieren an dieser Stelle abgebrochen werden. In einem Folgepapier kann die Diskussion ja wieder aufgenommen werden. Dazu sei der Leser herzlich eingeladen.

9 Zugabe: Ein Spiel zum Mitspielen

Vielleicht kommt der Wunsch auf, dass der Anwender selbst mitspielen will. Hierzu brauchen wir nur das Würfeln des ersten Spielers dem Anwender überlassen. Die Funktion `play.with.me()` setzt diesen Gedanken um.

79 ⟨*definiere Funktion zum Mitspielen:* `play.with.me()` 79⟩ ≡ ⊂ 87

```
play.with.me <<- function(n.player = 2, w.max = 1000, Debug = !TRUE,
                          sleep = 0.2, seed = 13, NO.OUT = !FALSE,
                          sleep.slider = TRUE){
  set.seed(seed)
  ⟨spiele Spiel mit mehreren Spielern mit Anwender als ersten Spieler 80⟩
  title(paste("Counter=",counter,", winner=",akt.player,
              ", last throw=",WUERFEL,
              ",\n seed",seed,", n.players=",n.player,sep=""))
  return(list(Counter=counter, Spieler=akt.player,
              Wuerfel=WUERFEL, Kegel=kegel))
}
```

Der Rahmen ist identisch mit dem der Funktion `play.mad()`. Zu ändern ist der Code-Chunk, der den Kern des Spiels umsetzt, und natürlich sein Name.

80 ⟨*spiele Spiel mit mehreren Spielern mit Anwender als ersten Spieler* 80⟩ ≡ ⊂ 79

```
⟨generiere Felder 2⟩
⟨stelle Spielfeld dar 6⟩
⟨initialisiere Spiel mit mehreren Spielern 50⟩
melde("Initialisieren")
repeat{
  ⟨berichte über Zustand 44⟩
  ⟨verhindere Endlosschleife 25⟩
  if(fertig) break else counter <- counter+1
  ⟨bestimme Spieler, der dran ist 51⟩
  ⟨würfle den Würfel, aber nicht für Anwender 81⟩
  ⟨handle Ärgern ab 46⟩
  ⟨handle Neuen-Kegel-Heraussetzen ab 47⟩
  ⟨wähle Kegel zum Setzen bei mehreren Spielern 56⟩
  ⟨handle Kegel-ins-Ziel-Bringen ab 48⟩
  ⟨handle Normales-Weitersetzen ab 49⟩
  next
}
```

Im Unterschied zu der Funktion `play.mad()` muss der Anwender sein Würfelergebnis eingeben. Dazu ist der Chunk, in dem gewürfelt wird, zu modifizieren.

Die vom Anwender eingegebenen Augenzahlen müssen korrekt sein, also aus der Menge $\{1, 2, \ldots, 6\}$ stammen. Mit der Funktion `menu()` ist das ein leichtes Spiel. Falls der Anwender `Exit` wählt, erfolgt eine vorzeitige Beendigung der Funktion.

81 ⟨*würfle den Würfel, aber nicht für Anwender* 81⟩ ≡ ⊂ 80

```
WUERFEL <- sample(1:6,1)
melde("Wurf",WUERFEL,"count:",counter,"Player:",akt.player)
if(akt.player==1){
  h <- "Spieler 1 ist am Zug. Bitte Augenzahl eingeben!"
  WUERFEL <- menu(1:6,title=h)
  if(WUERFEL==0) return("Oh, vorzeitiges Ende: Abbruch durch Spieler 1")
}
```

Das war's schon. Übrigens könnte man die Eingaben von Spieler 1, also des Anwenders, sammeln und dann mit ein paar Tests auf Zufälligkeit checken, ob er nicht gemogelt hat. Diese Verbesserung überlassen wir gerne dem Leser.

10 Rückblick

Mit einer Bemerkung zum Lösungsstil wollen wir das Papier beenden. Denn die Beschäftigung mit dem Spiel *Mensch ärgere Dich nicht!* hatte wesentlich den Zweck, für die eingesetzten Techniken anhand eines anfassbaren Beispiels zu werben. So wird dem Leser in kleinen Häppchen und hoffentlich verdaulich R serviert. Der hier verwendete Stil der *literaten Programmierung* offenbart sich dem Leser unmittelbar beim Lesen. Er wird erkennen, dass auf dem ersten Blick schwer erscheinende Probleme greifbar werden. Der Leser kann durch das Studium des Textes unterstützt durch die Code-Chunk-Namen die Struktur der Lösung verstehen. Neu definierte Abstraktionsebenen werden explizit beschrieben, und die gewählte Zerlegung eines Problems in Teilprobleme lässt sich nachvollziehen. Der Leser kann ebenfalls überprüfen, ob die Übersetzung eines Teilproblems stimmig ist, und damit letztlich die Lösungen validieren. Dieser Schritt ist gerade für Simulationsvorhaben von herausragender Bedeutung.

Für den Entwickler bietet der Stil ebenfalls bedeutsame Vorteile. Wie vorgeführt lassen sich Code-Chunks, beispielsweise zur Abbildung des Spielbrettes, mehrfach verwenden. Diese Art der Wiederverwendung alter Lösungen bietet sich in vielen Fällen an, in denen der Einsatz von Funktionen unpassend erscheint. Denn notwendigerweise führen Funktionen zur Laufzeit zu Speicheroperationen, die aus Problemsicht nicht immer überzeugend sind. Wichtiger ist indes der Hinweis, dass der Stil während der Entwicklung Klarheit über die jeweilige Lösung bringt. Fehlersituationen lassen sich schnell einkreisen und damit dürfte auch bezüglich Wartungsfragen ein großen Plus zu vermerken sein.

Zusammenfassend betonen wir: Mit dem Stil des literaten Programmierens haben sich für die Simulation des Spiels *Mensch ärgere Dich nicht!* sehr schnell Erfolgserlebnisse eingestellt, und es liegen nun Lösungen zu verschiedenen Fragen vor, die schön, verständlich, überprüfbar und wartungsfreundlich sind.

A Anhang

Im Anhang sind einige ergänzende Bemerkungen notiert. Sie dienen der Komplettierung oder zeigen Weiterentwicklungen auf. Im letzten Teil findet der Leser eine Auflistung der Variablen und der Code-Chunk-Namen mit Angaben der Verwendungsorte.

A.1 Wartezeiten

Für Animationen müssen Wartezeiten gesetzt werden können. Die schon oben etablierte Idee ist, gewünschte Wartezeiten über Funktionsaufrufe anzugeben. Hierdurch erhielt oben die Variable `sleep` einen geeigneten Wert. Da in den verschiedenen Schleifen immer die Funktion `Sys.sleep` mit der Variablen `sleep` aufgerufen wird, ist es eine sichere Strategie, die zu setzen, sofern sie noch nicht existiert.

82 ⟨*setze Wartezeit* `sleep` 82⟩ ≡ ⊂ 28, 67, 83

```
if(!exists("sleep")) sleep <- 0.2
```

Wir wollen bei Fragen der Bequemlichkeit jedoch noch einen Schritt weiter gehen und eine interaktive Steuerung der Wartezeit mittels eines Sliders umsetzen. Für diesen Zweck ist die Funktion `slider` erforderlich sowie eine Variable in der Umgebung `slider.env`, auf der der aktuelle Wert von `sleep` gespeichert wird. Diese Variable gilt es ebenfalls zu initialisieren.

83 ⟨*setze Wartezeit* `sleep` *für* `play.mad` 83⟩ ≡ ⊂ 50

```
⟨setze Wartezeit sleep 82⟩
if(exists("slider.env")) slider(obj.name="sleep",obj.value=sleep)
```

Als dritten Schritt konstruieren wir einen Schieber.

84 ⟨*setze Wartezeit* `sleep` *für* `play.mad` 83⟩+ ≡ ⊂ 50

```
if(sleep.slider==TRUE && exists("slider")){
  slider(obj.name="sleep",obj.value=.2)
  set.sleep<-function(...){
    value <- as.numeric(slider(no=1))/10
    slider(obj.name="sleep",obj.value=value)
    cat("new sleep value",value)
    NULL
  }
  slider(sl.function=set.sleep, sl.names="sleeping time (in 0.1 sec)",
         sl.min=0, sl.max=40, sl.delta=1, sl.defaults=floor(sleep*10))
}
```

In den Schleifendurchläufen muss nun ein neu gesetzter Wartewunsch zum Tragen kommen.

85 ⟨*stelle ggf. neue Wartezeit fest* 85⟩ ≡ ⊂ 25

```
if(exists("slider.env")&&"sleep" %in% ls(env=slider.env))
  sleep<-slider(obj.name="sleep")
```

A.2 Tcl/Tk-Update

Leider muss auf manchen Systemen ein Impuls gegeben werden, damit der bei der Verwendung von `relax` Messages sofort angezeigt werden. Hierzu wird an den Würfel-Chunk eine hier nicht näher diskutierte `tcl`-Anweisung angehängt.

86 ⟨*würfle den Würfel* 20⟩+ ≡ ⊂ 9, 24, 43

```
if(length(grep("tcltk",search()))>0) tcl("update","idletasks")
```

A.3 Zusammenstellung der Funktionen

In diesem Abschnitt führen wir noch einmal alle definierten Funktionen auf.

87 ⟨*start* 87⟩ ≡

⟨*definiere Funktion zum einfachen Spiel:* `play.mad.simple()` 22⟩
⟨*definiere Funktion zum ein-Spieler-Spiel:* `play.mad.one()` 39⟩
⟨*definiere Funktion zum MAD-Spielen:* `play.mad()` 68⟩
⟨*definiere Experiment zur Ermittlung von Spieldauern:* `exp.mad.dauern()` 70⟩
⟨*definiere Experiment zur Verlaufsdarstellung* `exp.mad.prozess()` 74⟩
⟨*definiere Funktion mit Haltepunkten:* `play.mad.halt()` 88⟩
⟨*definiere Funktion zum Mitspielen:* `play.with.me()` 79⟩

A.4 Eine Simulation mit Haltepunkten

Es könnte sein, dass man nur bestimmte Phasen des Spieles sehen will. Mit Hilfe der Funktion `play.mad.halt()` können bestimmte Haltepunkte in der Darstellung realisiert werden.

88 ⟨*definiere Funktion mit Haltepunkten:* `play.mad.halt()` 88⟩ ≡ ⊂ 87, 90

```
play.mad.halt <<- function(n.player=4, seed=13,Debug=FALSE,sleep=0.01,
                     w.max=80,NO.OUT=FALSE,show.zuege=NA){
  ⟨generiere Felder 2⟩
  ⟨stelle Spielfeld dar 6⟩
  if(0<length(show.zuege) && !is.na(show.zuege[1])){
    make.stops <- TRUE; mc<-make.circle
    mc.no<-function(x, y, col = "white", size = 4.5, txt = "") "relax"
  }
  set.seed(seed)
  ⟨spiele Spiel mit mehreren Spielern 43⟩
  return(kegel)
}
```

Damit diese Idee funktioniert, muss ein passender Bericht angefordert werden.

89 ⟨*berichte über Zustand* 44⟩+ ≡ ⊂ 43, 74, 80

```
if(exists("make.stops")){
  if(counter %in% show.zuege){
    ⟨stelle Spielfeld dar 6⟩
    ⟨aktualisiere Darstellung des Spielbrettes 66⟩
    make.circle <- mc; title(counter)
  } else {
```

```
    make.circle <- mc.no
  }
}
```

Durchführung einer Simulation mit Haltepunkten gelingt mit folgendem Aufruf:

90 ⟨*zeige Simulation mit Haltepunkten* 90⟩ ≡
⟨*definiere Funktion mit Haltepunkten:* `play.mad.halt()` 88⟩
```
play.mad.halt(NO.OUT=TRUE,show.zuege=(1:5)*10,sleep=0.1)
```

A.5 Alternative Ideen zur Zustandsberechnung

Während der Entwicklung zeigte sich, dass die eingeschlagene Strategie zur Ermittlung des nächsten Kegels zu sehr unübersichtlichen Anweisungen führte, die leider auch in verschiedenen Fehler mündeten. Deshalb sei empfohlen, eine alternative Vorgehensweise zu wählen. Diese könnte bei dem Zug mit höchster Priorität anfangen und in etwa, wie folgt, umzusetzen sein.

91 ⟨*alternative Bestimmung möglicher neuer Felder* 91⟩ ≡
```
ziel.feld  <- brett.feld <- zug.erlaubt <-
  cand.brett <- cand.start <- cand.ziel <- rep(0,4)
limit.ziel <- 1 + 10*(akt.player-1); akt.kegel <- kegel[,akt.player]
if(any(ok<-(akt.kegel==start) & !(pos<-(WUERFEL+akt.kegel) %in% akt.kegel))){
  # Platte putzen
  cand.start <- ok; brett.feld <- pos*cand.start; idx <- (1:4)*cand.start
} else {
  ?? ueber.start <- (start <= pos) & (akt.kegel < start)
  if(any(ok<-pos %in% kegel[,-akt.player] && ??pos nicht nach Startfeld??)){
    # Schlagen
    cand.brett <- ok; brett.feld <- pos*cand.brett; idx <- (1:4)*cand.brett
  } else {
    ?? Zielfelder berechnen
    if(??){
      # Kegel ins Ziel
      cand.ziel <- ok; ziel.feld <- pos
    } else {
      # Normalezug
      cand.brett <- ok; brett.feld <- pos*cand.brett; idx <- (1:4)*cand.brett
    }
  }
}
```

A.6 Indizes

Object Index

akt.kegel ∈ 91
akt.player ∈ 44, 45, 46, 47, 48, 50, 51, 52, 53, 54, 55, 56, 57, 60, 61, 62, 63, 65, 67, 68, 79, 81, 91
anz.keine.sechs ∈ 50, 51
anz.ok ∈ 56
blue.start ∈ 5, 8, 10, 11, 28, 33, 50
blue.ziel ∈ 4, 8, 15, 18, 32, 37, 50
brett.feld ∈ 56, 57, 91
cand.brett ∈ 56, 57, 59, 91
cand.start ∈ 57, 91
cand.ziel ∈ 56, 57, 58, 91
col.fields ∈ 8, 13, 35, 60, 63
cols.player ∈ 50, 54, 66
cols.player.kegel ∈ 50, 55, 61, 62, 63, 66, 76
counter ∈ 43, 44, 45, 48, 50, 65, 67, 68, 75, 79, 80, 81, 89
Debug ∈ 27, 39, 67, 68, 69, 74, 79, 88
fertig ∈ 9, 21, 24, 28, 43, 48, 50, 80
fields ∈ 2, 8, 12, 13, 14, 34, 35, 36, 55, 60, 62, 63, 66
green.start ∈ 5, 8, 50
green.ziel ∈ 4, 8, 50
idx ∈ 24, 31, 32, 33, 34, 35, 36, 37, 38, 47, 48, 50, 54, 55, 56, 58, 59, 60, 61, 62, 63, 91
kegel ∈ 9, 10, 12, 13, 14, 15, 16, 17, 18, 24, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 44, 48, 50, 51, 52, 53, 54, 55, 56, 57, 60, 61, 62, 63, 65, 66, 67, 68, 74, 75, 79, 88, 91
kegel.pech ∈ 63
limit.ziel ∈ 57, 91
mad.verlaufs.ergebnis ∈ 75, 76
make.circle ∈ 7, 8, 10, 11, 12, 13, 14, 15, 28, 33, 34, 35, 36, 37, 50, 54, 55, 60, 61, 62, 63, 66, 88, 89
make.no.circles ∈ 7, 70, 74
make.stops ∈ 88, 89
mc ∈ 88, 89
mc.no ∈ 88, 89
melde ∈ 24, 27, 33, 35, 43, 45, 46, 47, 48, 49, 50, 51, 54, 56, 60, 63, 65, 80, 81
neu.feld ∈ 63
new.player ∈ 51
n.player ∈ 50, 51, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 76, 78, 79, 88
n.sim ∈ 40
opar ∈ 70
player.pech ∈ 63
pos ∈ 66, 91
red.start ∈ 5, 8, 50
red.ziel ∈ 4, 8, 50
res ∈ 42, 70, 76
result ∈ 40, 41, 42, 70, 76, 77, 78
set.sleep ∈ 84
sleep ∈ 19, 21, 25, 28, 39, 40, 50, 67, 68, 69, 70, 74, 79, 82, 83, 84, 85, 88, 90
sleep.slider ∈ 68, 70, 74, 79, 84
value ∈ 84
vorrat ∈ 50, 54, 63, 66
w.max ∈ 19, 21, 25, 26, 28, 39, 40, 46, 65, 67, 68, 69, 70, 71, 72, 73, 74, 79, 88
WUERFEL ∈ 9, 14, 17, 18, 20, 24, 31, 32, 36, 38, 44, 45, 46, 47, 50, 51, 57, 62, 63, 65, 67, 68, 79, 81, 91
x.shift ∈ 5
yellow.start ∈ 5, 8, 50
yellow.ziel ∈ 4, 8, 50
y.shift ∈ 5
ziel.feld ∈ 57, 61, 91
ziel.koor ∈ 50, 61, 66
zug.erlaubt ∈ 56, 57, 91

Code Chunk Index

⟨*aktualisiere Darstellung des Spielbrettes* 66⟩ ⊂ 65, 89 . . . . . p30
⟨*alternative Bestimmung möglicher neuer Felder* 91⟩ p42
⟨*berichte über Zustand* 44 ∪ 75 ∪ 89⟩ ⊂ 43, 74, 80 . . . . . p22
⟨*bestimme mögliche neue Felder* 57⟩ ⊂ 56 . . . . . p27
⟨*bestimme Spieler, der dran ist* 51⟩ ⊂ 43, 80 . . . . . p25
⟨*checke Aufbau* 64⟩ p29
⟨*checke Testsituation* 65⟩ p29
⟨*definiere Experiment zur Ermittlung von Spieldauern:* `exp.mad.dauern()` 70⟩ ⊂ 71, 72, 73, 87 . . . . . p33
⟨*definiere Experiment zur Verlaufsdarstellung* `exp.mad.prozess()` 74⟩ ⊂ 77, 78, 87 p35
⟨*definiere Funktion mit Haltepunkten:* `play.mad.halt()` 88⟩ ⊂ 87, 90 . . . . . p41
⟨*definiere Funktion zum einfachen Spiel:* `play.mad.simple()` 22⟩ ⊂ 23, 87 . . . p13
⟨*definiere Funktion zum ein-Spieler-Spiel:* `play.mad.one()` 39⟩ ⊂ 40, 87 . . . . . p19
⟨*definiere Funktion zum MAD-Spielen:* `play.mad()` 68⟩ ⊂ 69, 70, 87 . . . . . p32
⟨*definiere Funktion zum Mitspielen:* `play.with.me()` 79⟩ ⊂ 87 . . . . . p38
⟨*entferne aktuellen Kegel* 60⟩ ⊂ 48, 49 . . . . . p28
⟨*entferne ggf. fremden Kegel vom Brett* 63⟩ ⊂ 47, 49 . . . . . p28
⟨*entferne Kegel* 13⟩ ⊂ 9 . . . . . p11
⟨*entferne Kegel mit Index* `idx` 35⟩ ⊂ 24 . . . . . p17
⟨*erledige eine einfache Simulation mit nur 30 Würfen* 41⟩ p20
⟨*ermittle Spieldauern für zwei Spieler* 71 ∪ 72 ∪ 73⟩ p33
⟨*erstelle Skizze für das Brett* 1⟩ p5
⟨*führe einfaches Spiel aus* 23⟩ p13
⟨*generiere Felder* 2 ∪ 4 ∪ 5⟩ ⊂ 9, 24, 43, 64, 80, 88 . . . . . p6
⟨*handle Ärgern ab* 46⟩ ⊂ 43, 65, 80 . . . . . p23
⟨*handle Kegel-ins-Ziel-Bringen ab* 48⟩ ⊂ 43, 65, 80 . . . . . p23
⟨*handle Neuen-Kegel-Heraussetzen ab* 47⟩ ⊂ 43, 65, 80 . . . . . p23
⟨*handle Normales-Weitersetzen ab* 49⟩ ⊂ 43, 65, 80 . . . . . p24
⟨*initialisiere einfachstes Spiel* 21⟩ ⊂ 9 . . . . . p12
⟨*initialisiere Kegel* 10⟩ ⊂ 21 . . . . . p11
⟨*initialisiere Spiel mit einem Spieler* 28⟩ ⊂ 24 . . . . . p16
⟨*initialisiere Spiel mit mehreren Spielern* 50⟩ ⊂ 43, 64, 65, 80 . . . . . p24
⟨*ist Kegel noch nicht im Spiel* 16⟩ ⊂ 9 . . . . . p12
⟨*ist neues Feld im Ziel des aktuellen Kegels und erlaubt* 58⟩ ⊂ 48 . . . . . p28
⟨*ist Zielfeld des aktuellen Kegels auf Spielfeld* 59⟩ ⊂ 49 . . . . . p28
⟨*ist Zielfeld des gewählten Kegels auf Spielfeld* 31⟩ ⊂ 24 . . . . . p17
⟨*ist Zielfeld ein normales Feld* 17⟩ ⊂ 9 . . . . . p12
⟨*ist Zielfeld im Ziel* 18⟩ ⊂ 9 . . . . . p12
⟨*ist Zielfeld im Ziel des vordersten Kegels* 32⟩ ⊂ 24 . . . . . p17
⟨*kann neuer Kegel eingebracht werden* 30⟩ ⊂ 24 . . . . . p17
⟨*kann Spieler neuen Kegel einbringen* 53⟩ ⊂ 47 . . . . . p25
⟨*muss neuer Kegel eingebracht werden* 29⟩ ⊂ 24 . . . . . p16
⟨*muss Spieler neuen Kegel einbringen* 52⟩ ⊂ 46 . . . . . p25
⟨*nehme einen Kegel aus dem Vorrat* 33⟩ ⊂ 24 . . . . . p17
⟨*nehme Kegel aus Vorrat* 11⟩ ⊂ 9 . . . . . p11

⟨*setze Wartezeit* `sleep` 82⟩ ⊂ 28, 67, 83 ..... p40
⟨*setze Wartezeit* `sleep` *für* `play.mad` 83 ∪ 84⟩ ⊂ 50 ..... p40
⟨*simuliere mehrere Spiele mit einem Spieler* 40⟩ p19
⟨*simuliere Verläufe* 77 ∪ 78⟩ p36
⟨*spiele einfachstes Spiel* 9⟩ ⊂ 22 ..... p10
⟨*spiele Spiel mit einem Spieler* 26⟩ p14
⟨*spiele Spiel mit mehreren Spielern* 43⟩ ⊂ 67, 68, 74, 88 ..... p22
⟨*spiele Spiel mit mehreren Spielern mit Anwender als ersten Spieler* 80⟩ ⊂ 79 p38
⟨*spiele Spiel mit nur einem Spieler* 24⟩ ⊂ 26, 39 ..... p13
⟨*start* 87⟩ p41
⟨*stelle aktuellen Kegel auf neue Position* 62⟩ ⊂ 49 ..... p28
⟨*stelle aktuellen Kegel ins Ziel* 61⟩ ⊂ 48 ..... p28
⟨*stelle Funktion* `melde` *bereit* 27⟩ ⊂ 24, 50 ..... p16
⟨*stelle gewählten Kegel auf das Startfeld* 34⟩ ⊂ 24 ..... p17
⟨*stelle gewählten Kegel auf neue Position* 36⟩ ⊂ 24 ..... p18
⟨*stelle gewählten Kegel des Spielers auf das Startfeld* 55⟩ ⊂ 47 ..... p26
⟨*stelle ggf. neue Wartezeit fest* 85⟩ ⊂ 25 ..... p40
⟨*stelle Kegel auf das Startfeld* 12⟩ ⊂ 9 ..... p11
⟨*stelle Kegel auf neue Position* 14⟩ ⊂ 9 ..... p11
⟨*stelle Kegel ins Ziel* 15⟩ ⊂ 9 ..... p12
⟨*stelle Kegel mit Index* `idx` *ins Ziel* 37⟩ ⊂ 24 ..... p18
⟨*stelle Spielfeld dar* 6 ∪ 7 ∪ 8⟩ ⊂ 9, 24, 43, 64, 65, 80, 88, 89 ..... p7
⟨*teste das Spiel mit mehreren Spielern* 67⟩ p30
⟨*teste Mehrspielerspiel mit 2 Spielern* 69⟩ p32
⟨*verhindere Endlosschleife* 25⟩ ⊂ 9, 24, 43, 80 ..... p14
⟨*verhindere Endlosschleife für das Ein-Kegel-Spiel* 19⟩ p12
⟨*wähle einen Kegel aus Vorrat des Spielers aus* 54⟩ ⊂ 47 ..... p26
⟨*wähle Kegel zum Setzen* 38⟩ ⊂ 24 ..... p18
⟨*wähle Kegel zum Setzen bei mehreren Spielern* 56⟩ ⊂ 43, 65, 80 ..... p26
⟨*werte einfaches Spiel aus* 42⟩ p21
⟨*würfle den Würfel* 20 ∪ 45 ∪ 86⟩ ⊂ 9, 24, 43 ..... p12
⟨*würfle den Würfel, aber nicht für Anwender* 81⟩ ⊂ 80 ..... p39
⟨*zeige Felder* 3⟩ p6
⟨*zeige Simulation mit Haltepunkten* 90⟩ p42
⟨*zeige Verlauf an* 76⟩ ⊂ 74 ..... p36