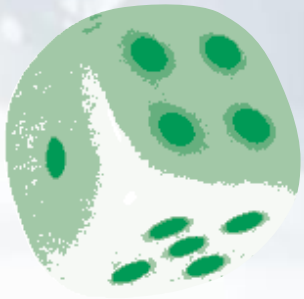
Der Computer-spieler

von Christian Schmitz

Serie Realbasic, Folge 2 In dieser Folge programmieren wir den Computergegner für die Computer-Simulation von „Mensch-ärgere-Dich-nicht“. Am Schluss spielt diese komplett automatisch mit allen vier Spielern gegen sich selbst



EINSTIEG

Die aktuelle Serie zeigt, wie man in Realbasic ein Brettspiel à la „Mensch ärgere Dich nicht“ erstellt. In der ersten Folge haben wir die Spieloberfläche entworfen, in dieser Folge entwickeln wir vier Computerspieler, die gegeneinander „Mensch ärgere Dich nicht“ spielen.

ZUNÄCHST SORGEN wir dafür, dass wir die Felder auf dem Spielbrett im Programmcode besser unterscheiden können. Dafür legen wir unter der Superklasse „Feld“ drei Klassen an „Hausfeld“, „Zielfeld“ und „Brettfeld“. Im Fenster ändern wir passend die Superklassen der Felder für die Häuser, die Zielfelder und die Felder des Bretts.

Wenn jetzt alle Felder nicht mehr von der Klasse „Feld“ abgeleitet werden, dann kann man später im Programmcode einfach testen zu welcher Art ein Feld gehört. Nehmen wir an, dass wir eine Variable „f“ vom Typ Feld haben, in der irgendein Feld abgelegt ist. Mit der Zeile „If f isa Zielfeld Then“ kann man testen, ob dieses Feld von der Klasse Zielfeld stammt. Natürlich stimmt „If f isa Feld Then“ auch.

Figuren-Klasse

Für die Daten der Spielfiguren legen wir eine Klasse namens „Figur“ an. Sie bekommt eine Eigenschaft „Farbe as Color“ für die Farbe der Spielfigur. Dazu dann noch eine Eigenschaft „Feld as Feld“ für einen Verweis auf das aktuelle Feld, auf dem die Figur steht.

Wenn wir im Programmcode eine Figur erzeugen, dann sollten wir direkt beim Erzeugen der Figur eine Farbe angeben. Für eine Zeile wie „f = New Figur(Farbe)“, bei der wir direkt die Farbe übergeben, brauchen wir einen so genannten Konstruktor. Ein Konstruktor ist ein kleines Unterprogramm, dass aufgerufen wird, wenn ein Objekt aus einer Klasse erzeugt wird. Passend dazu wird ein Destruktor aufgerufen, wenn das Objekt aus dem Speicher aufgelöst wird. In Realbasic benennt man den Konstruktor genauso wie die Klasse. Vor den Namen des Destruktors stellt man zusätzlich ein „~“. Für unsere Figur heißen damit der Konstruktor „Figur“ und der Destruktor „~Figur“.

Nun erstellen wir in der Klasse „Figur“ eine neue Methode mit dem Namen „Figur“ und dem Parameter „c as Color“ (Menübefehl „Neue Methode…“ im Menü „Bearbeiten“). Mit der Zeile „Farbe = c“ in dieser Methode sorgen wir dafür, dass man keine Figur erzeugen kann, ohne Ihr eine Farbe zu geben.

Es ist eine gute Angewohnheit, Variablen, die nur lokal beschränkt verwendet werden, einen kurzen Namen zu geben. Dabei orientiert man sich am Namen des Variablentyps und zählt bei Bedarf im Alphabet weiter. Wer also ein „i“ oder ein „j“ sieht, kann davon ausgehen, dass es ein Integer ist oder ein „c“ eine

Serie: Realbasics für Profis

1	Interface	Heft 12/2001
2	Computerspieler	Heft 01/2002
3	Mensch spielt mit	Heft 02/2002
4	Netzwerk	Heft 03/2002
5	Kompilieren und Release	Heft 04/2002

Farbe (Color) und ein „s“ ein Text (String). Dabei sollte man aber direkt auf einen Blick sehen können, wo sie deklariert wurden. Bei Variablen mit größerer Reichweite nimmt man längere beschreibende Namen wie zum Beispiel „Farbe“ in der Klasse „Figur“.

Spieler-Klasse

Um alle Spielerdaten zusammen zu halten, erzeugen wir eine neue Klasse namens „Spieler“. Diese Klasse bekommt mehrere Eigenschaften. Eine Farbe mit „Farbe as Color“, der Name des Spielers mit „Name as String“ und drei Zahlenwerte mit „Würfel as integer“, „Runden as integer“ und „Offset as integer“. In „Würfel“ speichern wir die zuletzt gewürfelte Zahl des Spielers. In der Eigenschaft „Runden“ halten wir die noch verbleibenden Würfelversuche für den Spieler fest. Am Anfang ist das eine drei, bis der Spieler eine Figur im Spiel hat. Bei einer gewürfelten sechs bekommt er einen Versuch mehr. Mit „Offset“ vermerken wir, ab welchem Index in den Feldern des Spielbretts das jeweilige Heimfeld anfängt.

Immer der Reihe nach!

Die Spieler sollen der Reihe nach spielen. Dazu setzen wir einen Timer in das Hauptfenster. Wir ziehen also das Steuerelement mit dem Bild einer Uhr aus der Toolbar in das Hauptfenster. Ein Doppelklick auf dieses Steuerelement im Fenster bringt uns in den Code-Editor. Dort öffnen wir den „Action“-Event des Timers. Als Code fügen wir nur eine Zeile mit dem Inhalt „Nextplayer“ ein. Die Methode „Nextplayer“ wird den nächsten Spieler auswählen und seinen Spielzug veranlassen.

Wir legen nun eine Methode „Nextplayer“, eine Eigenschaft „Currentplayer as Spieler“ und eine Eigenschaft „SpielerCounter as integer“ an. Currentplayer hält einen Verweis auf den aktuellen Spieler. SpielerCounter speichert, welcher Spieler an der Reihe ist.

Die erste Aufgabe von Nextplayer ist es, herauszufinden, ob ein Spieler schon im Spiel ist und noch eine Runde zu spielen hat. Das passiert zum Beispiel nachdem er eine sechs gewürfelt hat. Mit der Zeile „If Currentplayer = Nil or currentplayer.runden < = 0 Then“ prüfen wir, ob noch kein Spieler ausgewählt wurde. Der Vergleich „currentplayer.runden < = 0“ kontrolliert, ob der aktuelle Spieler keinen Wurf mehr hat. In beiden Fällen gehen wir zum nächsten Spieler über.

Falls noch kein Spieler dran war (Currentplayer ist noch Nil), nehmen wir einfach den Spieler 0 („Currentplayer = Spieler(0)“). Ansonsten zählen wir den SpielerCounter eins höher. Wenn SpielerCounter größer als die Liste der Spieler wird, setzen wir den Zähler wieder auf 0. Die Zeile „Currentplayer = Spieler(SpielerCounter)“ wählt den aktiven Spieler aus und mit der Zeile „Spielername.Text = Currentplayer.Name“ schreiben wir den Namen des aktiven Spielers in das Fenster.

Spieler rekrutieren

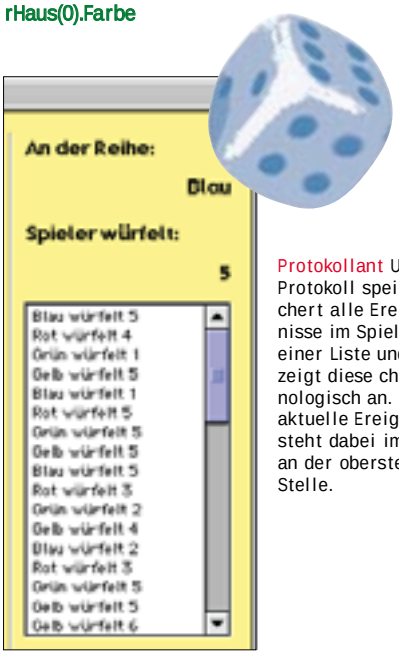
Noch existiert kein Spieler. Das ändern wir aber schnell, indem wir eine neue Eigenschaft „Spieler(3) as Spieler“ anlegen. „Spieler“ ist jetzt ein Feld von 0 bis 3 für insgesamt 4 Spieler. Eine neue Methode „NeuesSpiel“ im Hauptfenster mit folgenden Zeilen legt vier Spieler an:

```
Spieler(0)=New Spieler("Blau", RGB(0,0,255),0)
Spieler(1)=New Spieler("Gelb",
RGB(255,255,0),10)
Spieler(2)=New Spieler("Grün", RGB(0,255,0),20)
Spieler(3)=New Spieler("Rot", RGB(255,0,0),30)
```

Damit existiert nun pro Farbe ein Spieler mit dem Namen der Farbe als Spielername, der eigentlichen Farbe und der jeweiligen Position im Spielfeld. Doch den Konstruktor der Klasse „Spieler“ gibt es noch nicht. Dazu legen wir in der Klasse Spieler eine neue Methode mit dem Namen „Spieler“ und den Parametern „n as String“, „c as Color“, „o as Integer“ an. Die übergebenen Werte speichern wir dann in den Variablen Name, Farbe und Offset ab.

Im Open Event vom Fenster sollten wir jetzt noch zum Starten des Spieles die Methode „NeuesSpiel“ aufrufen. Außerdem sollte man dort noch festlegen, dass die jeweiligen Startfelder der Spieler die selbe Farbe wie die Hausfelder haben:

```
Brett(0).Farbe = bHaus(0).Farbe
Brett(10).Farbe = yHaus(0).Farbe
Brett(20).Farbe = gHaus(0).Farbe
Brett(30).Farbe =
rHaus(0).Farbe
```



Protokollant Unser Protokoll speichert alle Ereignisse im Spiel in einer Liste und zeigt diese chronologisch an. Das aktuelle Ereignis steht dabei immer an der obersten Stelle.

Wenn wir das Programm nun starten, wechselt oben rechts in dem Textfeld sekündlich der Name des Spielers, der an der Reihe ist.

Alea iacta est – Der Würfel ist gefallen

Nun kommt der Zufall ins Spiel. Zunächst bekommt die Klasse Spieler eine neue Methode namens „Würfeln“. Diese Methode enthält die Würfelformel: „Würfel = rnd*6+1“. Rnd generiert eine Fließkomma-Zufallszahl, die größer gleich 0 und kleiner 1 ist. Wir nehmen diese Zahl mal sechs und addieren eins dazu. Da Würfel vom Typ „Integer“ ist, schneidet Realbasic die Nachkommastellen ab. Wir bekommen so gleichförmig verteilte Zufallszahlen von Eins bis Sechs.

Im Hauptfenster in der Methode „Nextplayer“ schreiben wir nun weiter. Zuerst versichern wir uns mit einem „If Currentplayer < > Nil Then“, dass ein Spieler an der Reihe ist. Dieser Spieler bekommt als erstes eine Runde abgezogen „currentplayer.runden = currentplayer.runden-1“. Anschließend rufen wir die Würfeln-Routine dieses Spielers auf und lassen ihn würfeln.

Das Ergebnis veröffentlichen wir zuerst im Textfeld unter dem Spielername mit der Zeile „Würfel.Text = Str(Currentplayer.Würfel)“. Als nächstes ergänzen wir das Protokoll um dieses Ergebnis. „Log Currentplayer.Name + ” würfelt “ + Str(Currentplayer.Würfel)“.

Wer eine Sechs würfelt, sollte noch mal würfeln. Dazu testen wir, ob „Currentplayer.Würfel“ gleich sechs ist. Wenn ja, setzen wir „Currentplayer.Runden“ wieder um eins rauf.

Jedem Spieler seine Figuren

Jeder Spieler braucht vier Figuren, die wir am Anfang in die vier Häuser setzen. Zuerst bekommt die Klasse Spieler eine neue Eigenschaft namens „Figur(3) as Figur“. Im Konstruktor füllen wir die Figuren mit den Zeilen:

```
Figur(0)=New Figur(Farbe)
Figur(1)=New Figur(Farbe)
Figur(2)=New Figur(Farbe)
Figur(3)=New Figur(Farbe)
```

Im Hauptfenster in der Methode „NeuesSpiel“ fehlen noch einige Variablen, denen wir Startwerte übergeben. Vor dem bisherigen Code sollten wir für gleich schon einmal eine Zählvariable für die For-Next-Schleife deklarieren: „Dim I as integer“. Anschließend löschen wir die Protokoll-Listbox mit „Liste.DeleteAllRows“, löschen den Verweis zum aktuellen Spieler mit „Currentplayer = Nil“, setzen den SpielerCounter auf Null.

Dies alles dient dazu, einen Menüeintrag „Neues Spiel“ zu erzeugen und damit die Reste vom letzten Spiel zu löschen. ►

Figuren ins Haus!

Nun schreiben wir noch einen weiteren Code, um jedem Spieler seine Häuser zu zeigen und dort seine Spielfiguren hinzustellen. Mit einer Schleife von null bis drei bekommt jeder Spieler seine Häuser und die Zielfelder zugewiesen:

```
For i=0 to 3
   Spieler(0).haus(i)=bHaus(i)
   Spieler(1).haus(i)=yHaus(i)
   Spieler(2).haus(i)=gHaus(i)
   Spieler(3).haus(i)=rHaus(i)
   Spieler(0).ziel(i)=bZiel(i)
   Spieler(1).ziel(i)=yZiel(i)
   Spieler(2).ziel(i)=gZiel(i)
   Spieler(3).ziel(i)=rZiel(i)
Next
```

Als nächstes stellt jeder Spieler seine Figuren ins Haus. Dazu rufen wir die Methoden „MoveHaus“ der Spieler auf, die wir noch anlegen.

```
Spieler(0).MoveHaus
Spieler(1).MoveHaus
Spieler(2).MoveHaus
Spieler(3).MoveHaus
```

Nun legen wir in der Klasse Spieler eine Methode „MoveHaus“ an. Diese Methode benutzt eine For-Next-Schleife, um allen Figuren den Spieler und das Haus zuzuweisen.

```
Dim i as integer
For i=0 to 3
   Figur(i).Spieler = me
   Figur(i).MoveTo haus(i)
Next
```

In der Klasse Spieler fehlt noch die Eigenschaften „Haus(0) as Hausfeld“ und „Ziel(3) as Zielfeld“. Damit kann jeder Spieler direkt auf seine Felder zugreifen.

Figur, setz dich da hin!

Wir übermitteln der Figur zuerst die Information, dass sie zu einem bestimmten Spieler gehört. Dafür deklarieren wir in der Klasse Figur die Eigenschaft „Spieler as Spieler“ und die Eigenschaft „Feld as Feld“, um einen Verweis auf das aktuelle Feld zu speichern.

Nun folgt die Methode „MoveTo“ mit dem Zielfeld als Parameter „f as Feld“. Dank der objektorientierten Programmierung akzeptiert MoveTo auch ein Ziel- oder Hausfeld.

Wenn das Zielfeld nicht undefiniert ist („If f < > Nil Then“), sehen wir nach, ob diese Figur schon auf einem Feld ist („If Feld < > Nil Then“). In diesem Fall löschen wir den Verweis auf die Figur in diesem Feld und lassen es neu zeichnen („Feld.Redraw“). Jetzt setzen wir die aktuelle Figur mit „f.Figur = me“ in das neue Feld. Mit „f.Redraw“ zeichnen wir das Feld neu und speichern es mit „Feld = f“ . Eben haben wir eine Eigenschaft vom Feld benutzt, die es noch nicht gibt. Wir ergänzen also in der Klasse Feld die Eigenschaft „Figur as Figur“. So weiß jedes Feld, was auf ihm sitzt.

Jedes Feld besitzt eine Draw-Methode und in dieser darf die Figur gezeichnet werden. Wenn also eine Figur zum Feld gehört („If Figur < > Nil Then“), dann setzen wir die Malfarbe der Grafikumgebung auf die Farbe der Figur „g.ForeColor = Figur.Farbe“. Wir zeichnen ein gefülltes Oval mit dem Befehl „g.Fill Oval 3, 3, width-6, height-6“. Dieses Oval ist kreisrund und hält zum Rand vom Feld einen Abstand von 3 Pixel. Wir wechseln die Malfarbe nun mit „g.ForeColor = RGB(0,0,0)“ auf Schwarz und zeichnen mit „g.DrawOval 3, 3, width-6, height-6“ über das gefüllte Oval ein ungefülltes als Rand.

Methoden für die Figur

Unsere Figuren sind jetzt noch recht unbeweglich. Manchmal muss eine Figur zurück ins Haus. Dazu geben wir der Klasse Figur eine Methode „MoveHaus“. Wir deklariereren eine Laufvariable mit „Dim i as integer“. In der For-Next-Schleife prüfen wir die vier Hausfelder des Spielers („For i = 0 to 3“), ob dort Platz ist („If Spieler.Haus(i).Figur = Nil Then“). Stoßen wir auf ein Hausfeld, das keine Figur hat, bewegen wir die Figur mit der Zeile „MoveTo Spieler.Haus(i)“ in das Haus. Mit dem Befehl „Exit“ brechen wir die Schleife ab.

Eine weitere Hilfsfunktion ist „ShowAt“. Wenn die Figur später wandert, sollen für einen kurzen Moment die Felder auf dem Wanderweg aufleuchten. Dies erreichen wir, indem wir auf dem Feld kurz (1/6tel Sekunde) die Figur gegen unsere austauschen. Wir deklarieren eine Methode „ShowAt“ mit dem betreffenden Feld als Parameter „f as Feld“.

In der Methode brauchen wir eine Variable, um die Figur des Feldes für den Tauschvorgang zwischen zu lagern („Dim old as Figur“). Zusätzlich benötigen wir noch eine Variable für die Startzeit („Dim I as integer“). Bis das Zielfeld definiert ist („If f < > Nil Then“), speichern wir den Verweis auf die aktuelle Figur des Feldes in der Variable Old („Old = f.Figur“). Danach setzen wir unsere aktuelle Figur mit „f.Figur = me“ in das Feld und zeichnen es mit „f.Redraw“ neu.

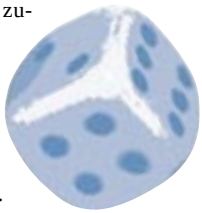
Eine Pause von 1/6tel Sekunde erhalten wir mit einer leeren Schleife. Wir messen die aktuelle Zeit in Ticks (ein Tick entspricht einer 1/60tel Sekunde) und zählen 10 Ticks hinzu. Wir benutzen eine While-Schleife, die so lange läuft, bis I größer als Ticks wird. Der Code dazu sieht so aus:

```
I = Ticks +10
While I > Ticks
Wend
```

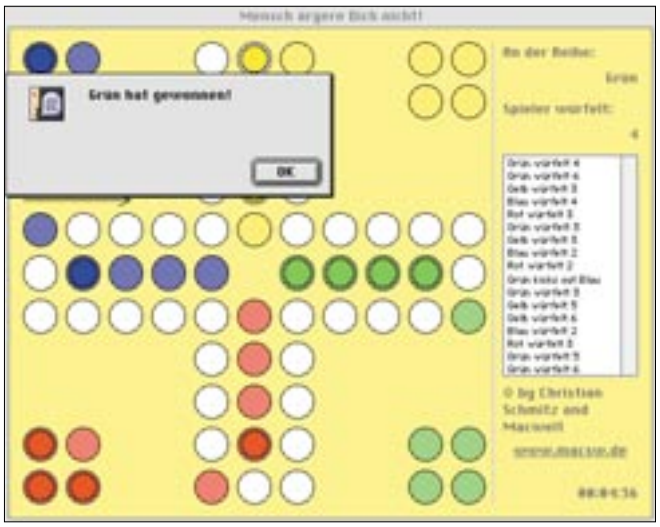
Zum Schluss restaurieren wir wieder mit „f.Figur = Old“ und „f.Redraw“.

Methoden für den Spieler

Auch die Klasse Spieler benötigt noch ein paar Befehle. Hilfsroutinen, die wir später aufrufen. Zuerst wäre da die Funktion „Alleimhaus“, die ein Boolean-Ergebnis zurückliefert. Es wird True, wenn der Spieler alle Figuren im Haus hat, sonst bleibt es False. Wir deklarieren zwei Variablen I und Z vom Typ Integer. I für die Schleife und Z als Zähler.



Für jede Figur („For I = 0 to 3“) prüfen wir, ob ihr Feld ein Hausfeld ist „If Figur(i).Feld isa Hausfeld Then“. Der Test mit isa klappt immer. Wenn das Feld nicht definiert ist (Wert Nil), dann kommt Falsch als Ergebnis zurück. Aber wenn es stimmt, zählen wir den Zähler um eins weiter („Z = Z + 1“).



Selbstläufer Das in dieser Folge fertiggestellte Programm läuft vollständig mit allen Regeln des Spiels. Vier Computerspieler spielen dabei gegeneinander. In der nächsten Folge kommt dann auch ein Mensch ins Spiel.

Mit der Zeile „Return Z = 4“ melden wir, ob alle vier Figuren im Haus sind.

In der Spieler-Klasse benötigen wir noch eine Funktion, die eine Figur aus dem Haus zurückgibt. Wir legen also eine Methode „FigurImHaus“ mit dem Rückgabetyp „Figur“ an.

In einer Schleife von 0 bis 3 gehen wir die Hausfelder durch und schauen, ob eine Figur drin ist („If haus(i).Figur < > Nil Then“). Wenn ja, melden wir diese als Ergebnis zurück: „Return Haus(i).Figur“.

Außerdem deklarieren wir schon für später eine Variable „ZielCount as integer“ und eine Methode „GeheFelder“ mit einem Parameter „c as integer“. Diese Methode wird die Figur auf dem Spielbrett bewegen. Die Variable „ZielCount“ zählt dann die Anzahl der Figuren im Zielbereich für jeden Spieler.

Bewegung auf dem Spielbrett

Den Code zum Würfeln schreiben wir in die Methode Nextplayer des Hauptfensters. Wir deklarieren dazu zwei Variablen „n“ und „war“ vom Typ „Figur“.

Wenn eine Sechs gewürfelt wird, setzen wir „Currentplayer.Runden“ auf eins. So lange ein Spieler noch Figuren im Haus hat, muss er bei einer Sechs eine Figur aufs Spielfeld setzen. Dazu rufen die Funktion „FigurImHaus“ des Spielers am Zug auf, lassen uns einen Verweis auf eine Figur im Haus geben und speichern diesen in „n“. Wenn wir keine Figur im Haus mehr finden, rufen wir „Currentplayer.GeheFelder 6“ auf und bewegen so die Figur um sechs Felder weiter.

Falls wir eine neue Figur auf das Brett stellen müssen, prüfen wir vorher, ob auf dem Startfeld vor dem Haus schon eine Figur steht. Die Zeile „War = Brett (Currentplayer.Offset).Figur“ fragt die Figur des Startfelds ab und kopiert einen Verweis in die Variable War. Wenn War ein gültiger Verweis auf eine Figur ist und diese Figur nicht dem aktuellen Spieler gehört („If War < > Nil and War.Spieler < > Currentplayer Then“), schreiben wir ins Protokoll: „log Currentplayer.Name +“kicks out“ + War.Spieler.Name“, schicken die Figur mit „War.MoveToHaus“ heim und leeren das Feld mit. „Brett(Currentplayer.Offset).Figur = Nil“.

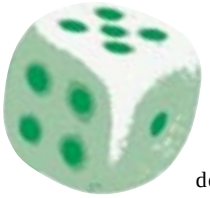
Nun prüfen wir noch einmal, ob das Feld leer ist „If Brett(Currentplayer.Offset).Figur = Nil Then“, und bewegen unsere Figur N mit „n.MoveTo Brett(Currentplayer.Offset)“ dort hin. Falls vorher auf dem Feld noch eine Figur steht, muss es die eigene sein und wir lassen den Spieler mit „Currentplayer.GeheFelder 6“ einen normalen Zug machen.

Falls keine Sechs gewürfelt wurde, soll die Figur die gewürfelte Anzahl an Feldern vorrücken. Die Zeile

Mit Uhrzeit In der finalen Fassung unserer Simulation bauen wir auch noch eine Zeitanzeige ein, für die wir in dieser Ausgabe leider keinen Platz mehr haben.

„Currentplayer.GeheFelder CurrentPlayer.Würfel“ erledigt dies. Der Methode „GeheFelder“ fehlt noch eine Funktion, die prüft, wie das nächste gültige Feld heißt.

Suche nach dem nächsten Feld



Wir erzeugen dazu eine neue Methode im Hauptfenster mit dem Namen „NextFeld“, den Parametern „S as Spieler, F as Feld“ und dem Rückgabetyp „Feld“. Wir wollen hiermit für den Spieler S vom Feld f aus das nächste freie Feld für die Figur suchen.

Auch hier brauchen wir eine Laufvariable („Dim i as integer“). Wir prüfen, ob das Feld f ein Brettfeld ist („If f isa Brettfeld Then“). Innerhalb der 40 Brettfelder zählen wir nun mit „I = f.Index +1“ hoch. „Index“ ist dabei der Index des Steuerelements.

Hier stoßen wir auf ein Problem. I kann den Wert 40 erreichen, doch Brett(40) existiert nicht mehr. Falls das I also 40 erreicht, setzen wir es einfach wieder auf Null.

Wenn I mit dem Offset des Spielers identisch ist, also der Nummer des Feldes, bei dem der Spieler anfängt, dann hat diese Figur eine Runde geschafft und wir schauen in die Zielfelder. In einer Schleife gehen wir die Zielfelder durch und suchen einen leeren Platz:

```
For i=0 to 3
    If s.Ziel(i).Figur = Nil Then
        Return s.Ziel(i)
    End If
Next
```

Falls wir ein Feld finden, auf dem keine Figur steht, geben wir dieses Feld als Ergebnis der Funktion zurück. Wenn der Spieler noch nicht fertig war, dann melden wir mit „Return Brett(i)“ das nächste Brettfeld zurück. Falls F jedoch kein Brettfeld ist, geben wir mit „Return F“ das selbe Feld wieder zurück.

Achtung! In dieser Funktion benutzen wir die Variable i für zwei unabhängige Dinge. Normalerweise schafft man sich damit sehr schnell Bugs, aber es spart nun mal ein paar Bytes Arbeitsspeicher.

Gehe-Felder

Jetzt endlich legen wir die Methode „GeheFelder“ mit dem Parameter „C as integer“ in der Klasse Spieler an. Mit dieser Methode bewegt der Spieler seine Figur um C Felder.

Wir legen zuerst zwei Variablen vom Typ Figur mit den Namen „F“ und „War“ an. Dann noch ein Feld mit dem Namen „B“ und ein Integer „i“ für die Schleifen.

Nun gehen wir alle Figuren des Spielers durch und suchen eine Figur auf dem Brett. Falls wir eine finden, legen wir sie in F ab.

Haben wir eine Figur gefunden, kopieren wir in B das aktuelle Feld der Figur mit „B = F.Feld“. Dann rufen wir die Methode NextFeld von 1 bis C im Hauptfenster auf rücken mit „B = window1.nextfeld (Me, B)“ jedesmal um ein Feld weiter. Wir kennen damit in B das Zielfeld unserer Wanderung. Falls dieses Zielfeld definiert ist („If B < > Nil Then“), legen wir zuerst die Figur, die dort steht, in der Variable War ab. Wenn War leer ist bzw. nicht zu einer der Figuren des Spielers gehört („If War = Nil or War.Spieler < > me Then“), können wir den Spielzug zu Ende führen.

Dazu starten wir wieder mit „B = F.Feld“ und gehen wie oben die Felder entlang. Diesmal rufen wir jedoch nach dem NextFeld-Aufruf die Methode „F.ShowAt b“ auf und zeigen damit kurz die Spielfigur auf dem Feld.

Wenn das letzte „B“ Feld ein Zielfeld sein sollte („If b isa Zielfeld Then“), dann hat der Spieler einen Punkt gemacht, den wir in der Variable „ZielCount“ mitzählen.

Falls nun in der Variable War eine Figur definiert ist („If War < > Nil Then“), dann dürfen wir diese Figur Heim schicken. Mit „window1.log me.Name+“kicks out“ + War.Spieler.Name“ dokumentieren wir das und führen es mit „War.MoveToHaus“ aus. Schließlich stellen wir die Figur mit „f.MoveTo b“ an Ihren Zielort.

Falls der Spieler alle vier Figuren im Ziel versammelt hat, dann hat er gewonnen! Wir schreiben „If me.zielcount=4 Then“ und zeigen den Sieger mit. „Msgbox me.name + “hat gewonnen!“ “ in einer Dialogbox an.

Fazit

Wir sind fertig mit dem Computerspieler. Mit der 3D-Umgebung aus Realbasic 3.5 könnte man dem Spiel auf Wunsch auch räumliche Tiefe verleihen. In der nächsten Folge darf dann der Mensch mitspielen. *cm*