Das Lingo Labor

Die Autoren

Joachim Gola ist technischer Leiter der Hamburger Multimedia-Firma 4=1 und Autor mehrerer Bücher zu Macromedia Director.

Johannes Asmus ist Maler und Kunsterzieher. Er lehrt an der Hochschule für angewandte Wissenschaft in Hamburg.

Johannes Asmus
Joachim Gola

Das Lingo Labor

Skripting-Experimente mit Macromedia Director

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

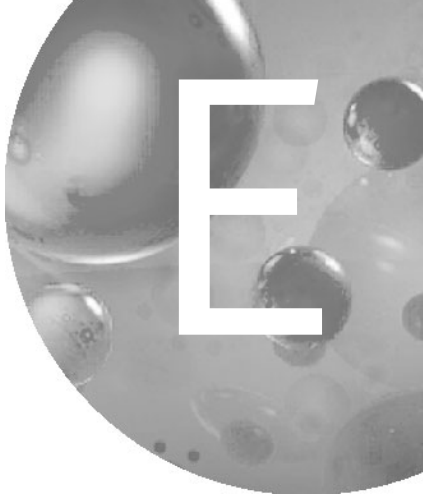
05 04 03 02

ISBN 3-8273-1909-9

Einbandgestaltung:	Marco Lindenbeck, Parsdorf b. München
Lektorat:	Klaus Hofmann, khofmann@pearson.de
Korrektorat:	Simone Meisner, Fürstenfeldbruck
Herstellung:	Anna Plenk, aplenk@pearson.de
CD Mastering:	Gregor Kopietz, gkopietz@pearson.de
Satz:	mediaService – Siegen (www.media-service.tv)
Druck und Verarbeitung:	Kösel, Kempten (www.KoeselBuch.de)

Printed in Germany

Inhaltsverzeichnis



Willkommen im Labor

Mit diesem Buch laden wir Sie ein, mit uns etwas zu entdecken: die Faszination nämlich, die das „allmähliche Verfertigen der Gedanken in Code“ auf Sie ausüben kann. Das Lingo Labor nimmt Ihren Spieltrieb, Ihren Wunsch, herumzuprobieren, zu „daddeln“, Alternativen auszuprobieren, als Ausgangspunkt und baut Ihnen Experimente auf, die Sie nicht lernen sollen, sondern mit denen Sie arbeiten und die Sie nach Belieben erweitern, verändern und fortführen können. Experimente, so hoffen wir, die Director und Lingo erfahrbar machen.

Die Autoren dieses Buchs sind ein ungleiches Team: Ein Anfänger, der die Faszination Director seit wenigen Monaten entdeckt, und der erfahrene Director-Anwender, der jahrelang seinen Lebensunterhalt mit diesem Werkzeug verdient hat. Diese Kombination stellte uns vor Herausforderungen und hat die Themenauswahl und die Art der Umsetzung in den Experimenten bestimmt.

Unter *„Experiment“* verstehen wir Folgendes:

1. Wir stellen einen definierten Ausgangszustand her. In der Praxis kann das ganz Unterschiedliches bedeuten: Vielleicht müssen wir nur das Director-Nachrichtenfenster öffnen, um dort Lingo-Befehle einzugeben, oder wir bereiten einen kleinen Director-Film vor, indem wir Bilder importieren und diese auf der Bühne arrangieren. Diesen Abschnitt nennen wir *„Versuchsaufbau“*. Zu Vorbereitung gehört manchmal auch eine kurze Einführung in ein Lingo-Themengebiet.
2. Jedes Experiment hat eine *„Durchführung“*, in der wir das Ausgangsmaterial nehmen und etwas daraus entwickeln. Wo immer möglich, tun wir dies mehrfach und auf unterschiedliche Weise. Sie als Laborant(in) sind gefordert, einmal mit uns das Experiment durchzuführen und dann selbst Variationen und Messreihen zu erstellen.
3. Wo nötig, findet im Anschluss (oder zwischendurch) eine *„Diskussion“* der Ergebnisse, Alternativen oder Probleme statt, die sich bei der Durchführung ergeben haben.

Das Lingo Labor kommt bewusst spielerisch daher – aber wir werden Sie nicht aus der Laborarbeit entlassen, ohne Sie mit allen wichtigen Skriptingtechniken in Director bekannt gemacht zu haben. Dieses Buch vermittelt viele spannende Ansätze – Animationstechniken, Imaging-Lingo, Lingo-generierte Director3D-Welten – und Skripting-Grundlagen

gleichermaßen. Dabei setzen wir *nicht* voraus, dass Sie Director bereits perfekt beherrschen, sondern gehen im Versuchsaufbau wie bei den Skripting-Experimenten Schritt für Schritt vor.

Dieses Buch stellt die Director-Grundlagen – beispielsweise auch die zahlreichen Hilfsmittel zum Animieren und Platzieren von Bühnenelementen, die Organisation von Besetzungen, Fragen des Imports von Medien und vieles mehr – stets dann dar, wenn wir ein Experiment aufbauen: Das Erstellen des Ausgangsfilms ist eine Aufgabe des Lingo Laboranten, ebenso wie die Durchführung der anschließenden Versuche.

Die Anlage des Buchs soll zwei Dinge von vornherein klarmachen:

- Wir wollen Ihnen Einstiegspunkte bieten, die zum eigenen Experimentieren einladen. Das Buch will nicht Director vollständig erklären – dafür gibt es andere Bücher.
- Ohne Ihre Lust, selbst etwas auszuprobieren, wird es nicht funktionieren: *Sie* führen die Experimente durch, das Buch ist nur das Laborhandbuch, das Sie dazu anleitet!

Eine wichtige Hilfe kann das auf CD-ROM mitgelieferte Material sein. Ein CD-Icon in der Randspalte weist darauf hin, dass sich auf der beiliegenden CD im jeweiligen Kapitel-Verzeichnis ein Beispielfilm befindet. Wir haben auch einen Endzustand des jeweiligen Beispielfilms mit auf die CD-ROM gepackt, denken aber, dass Sie mehr Spaß an der Laborarbeit haben werden, wenn Sie einen leeren Film (oder den jeweiligen *Start*-Film von der CD-ROM) nehmen und das gesamte Experiment in Director nachvollziehen.

Ein Hinweis auf *Beispiel.dir* im Text ist auf der CD-ROM beispielsweise so umgesetzt:

```
Lingolabor
  Kapitel01
    Beispiel_Start.dir   (öffnen Sie zu Beginn des Experiments)
    Beispiel_1.dir       (zeigt einen Zwischenstand)
    Beispiel_Ende.dir    (zeigt den Film am Ende des Experiments)
    Beispiel_Material    (Material-Verzeichnis zum Film)
      Bild.bmp
      Sound.mp3
      Video.mov
```

Alle Beispiele sind in Director 7, 8, 8.5 und neueren Versionen nutzbar – es sei denn, Sie finden eines der beiden in der Randspalte abgedruckten Icons, das die mindestens erforderliche Director-Version bezeichnet. Die Beispielfilme auf CD-ROM sind wo immer möglich im Format von Director 7 gesichert und können so in allen genannten Director-Versionen benutzt werden.

E.1 Warum Director?

Das Authoring-Programm Director war lange Jahre das „Flaggschiff“ der Softwarefirma Macromedia und ist auch heute noch weitgehend konkurrenzlos, was das Erstellen von Multimedia-Anwendungen für CD-ROM angeht. Director ist ein großer „Integrator“ und bietet Ihnen zahllose Möglichkeiten, unterschiedlichste Medienformate zusammenzuführen, zu steuern und zu einem Multimedia-Erlebnis zu machen. Directors integrierte Skripting-Sprache Lingo ist ein idealer Einstiegspunkt für Skripting-Sprachen überhaupt: In Director haben Sie sofortige visuelle Rückmeldung auf jede Ihrer Skriptzeilen, was beim Lernen und Experimentieren eine immense Hilfe ist.

Director lässt sich auch ohne Lingo benutzen: Wer nur animieren möchte, kann die zahlreichen Hilfsmittel für Drehbuch-Animationen nutzen, und wer mit den mitgelieferten „Bibliotheksskripten“ auskommt, kann Standard-Funktionen implementieren, ohne den Skripttext eines Blickes zu würdigen. Wirkliches „scriptless authoring“ kann es aber nicht geben, einfach weil die vorgefertigten Skripte schon bei einfachsten Anwendungen zu unflexibel sind – und man natürlich immer genau das benötigt, was sich nicht in der Bibliothekspalette, sprich in den mitgelieferten Skripten, befindet.

Sobald Interaktivität ins Spiel kommt, geht in Director ohne Lingo (fast) nichts, und schon um den Abspielkopf anzuhalten (der ansonsten immer weiter laufen würde – so weit, wie Sie Elemente im Drehbuch platziert haben) benötigen Sie ein kleines Skript. In den Experimenten werden Sie immer wieder von diesem „Stopper-Skript“ lesen.

Director hat ein größeres Inventar an „Keywords“ (Befehlen, Eigenschaften, Konstanten, Programmobjekten) als die meisten anderen Skriptingsprachen. Das ist eine direkte Folge der „Breite“ und Heterogenität des Programms: Ein Großteil der weit über 2.000 Lingo-Keywords sind medienspezifische Funktionen und Eigenschaften, die auch nur in Bezug auf einen bestimmten oder einige wenige Medientypen (z.B. Textdarsteller, Quicktime-Video, MP3-Sound, Shockwave3D-Darsteller etc.) zum Tragen kommen.

E.2 Konventionen

Die Skripte sind, wie sie im Buch abgedruckt sind, direkt in Director nutzbar. Im Druck mussten wir nur einen kleinen Kompromiss eingehen: Da auf der Druckseite Skripte häufig einmal in die nächste Zeile umbrechen, mussten wir uns für eines der beiden möglichen Zeichen für den Zeilenwechsel in mehrzeiligen Skriptzeilen entscheiden. Director 7 verwendet „¬“, Director 8 (und alle folgenden Versionen) dagegen „\“. In diesem Buch werden Sie immer „\“ als Zeilenumbruchszeichen finden. Wenn Sie das Skript selbst im Skripteditor schreiben, lassen Sie das Fortsetzungszeichen einfach weg und schreiben die folgende Zeile mit in die erste.

Menübefehle werden stets für die Plattformen Windows und MacOS angegeben; normalerweise unterscheiden sich die Kurzbefehle nur bei der Verwendung der Strg- (Windows) bzw. der ⌘-Taste (Mac). Der Befehl Strg/⌘+⇧+U (Filmskript öffnen) heißt unter Windows demnach Strg+⇧+U, am Mac aber ⌘+⇧+U.

Und nun: Viel Spaß im Labor!

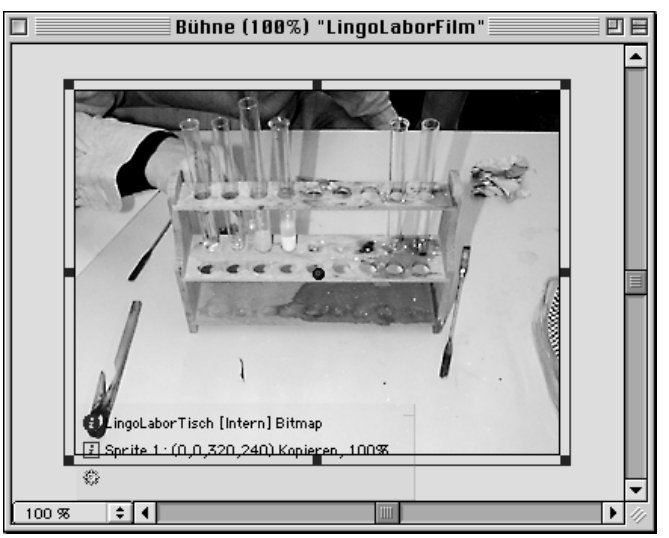


Abbildung E.1: Die Laborarbeit beginnt!

Spaß am Experimentieren

Um ein Gefühl dafür zu bekommen, was Director kann und wie das Programm arbeitet, muss man keine tausend Seiten eines Handbuchs gelesen haben, und man muss auch kein Medieninformatik-Studium absolviert haben. Unsere Überzeugung ist, dass ein gehöriges Maß an Neugier ausreicht, um loszulegen.

Zu jeder der folgenden Thesen wollen wir zum Einstieg ein kleines Experiment durchführen – als Einstimmung auf Director und als Appetithappen für die Experimente in den folgenden Kapiteln.

1. *Fehlermeldungen sind positiv!* Mit jeder Fehlermeldung, die Sie produzieren, wächst Ihr Verständnis des Programms.
2. *Tun Sie zunächst Dinge, die Sie können!* Beim Experimentieren ist es gut, zunächst Dinge auszuprobieren, bei denen Sie das Ergebnis kennen. Themen aus dem Schulunterricht (Mathematik, Physik) sind da gut geeignet.
3. Director – egal, welche Version Sie benutzen – ist ein Werkzeug, das *viel zu viel kann*. Deshalb kann eine Annäherung an Director nicht bedeuten, zwei- bis dreitausend Lingo-Befehle zu lernen. Gehen Sie von Ihrem momentanen Interesse aus, und erforschen Sie einen kleinen Ausschnitt der Möglichkeiten von Director.
4. Bauen Sie sich eine kleine *Testumgebung* und probieren Sie eine Reihe ähnlicher Dinge mit minimalen Änderungen am jeweiligen Skript aus.

1.1 Experiment 1: Hallo Welt!

Ein kleines Programm, das als Ausgabe die Nachricht „Hallo Welt!“ produziert, gehört zu den beliebten „Klassikern“ für alle Programmiersprachen. Selbstverständlich ist es auch in Director einfach herzustellen. Wir wollen aber einmal sehen, was man dabei alles falsch machen kann – und was man aus den Fehlern lernen kann ...

Versuchsaufbau

Starten Sie Director und ignorieren Sie die vielen Fenster, die sich da öffnen. Öffnen Sie mit Strg+M unter Windows beziehungsweise ⌘+M am Macintosh das Nachrichtenfenster.

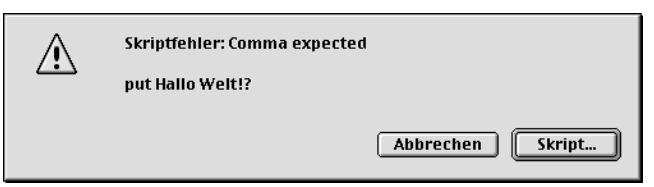
Durchführung

Tippen Sie das Folgende im Nachrichtenfenster:

```
put Hallo Welt!
```

Und senden Sie den Befehl mit der ↵-Taste ab. Was geschieht? Sie erhalten eine Fehlermeldung:

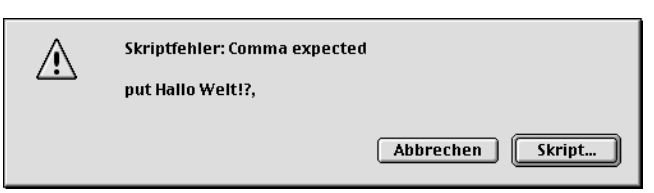
Abbildung 1.1: Fehlermeldung Nr. 1



Ein Komma fehlt? Vielleicht dort, wo in der Fehlermeldung das Fragezeichen steht? Vielleicht wird's ja so besser:

```
put Hallo Welt!,
```

Abbildung 1.2: Fehlermeldung Nr. 2



Die gleiche Fehlermeldung, und das Fragezeichen steht vor dem Komma. Aha, wir schließen: Das Ausrufezeichen könnte ein Problem sein. Und wir probieren gleich ein paar weitere Aufrufe im Nachrichtenfenster:

```
put Hallo Welt,
-- <Void> <Void>
put Hallo,Welt,
-- <Void> <Void>
put Hallo Welt
-- <Void> <Void>
```

Können Sie mit den Ergebnissen etwas anfangen? Wir sehen immerhin, dass `put`, so wie wir es erwarten, Ergebnisse in das Nachrichtenfenster schreibt. Wir ahnen, dass `put` die beiden Wörter `Hallo` und `Welt` getrennt behandelt und für beide einen Wert zurückliefert, und zwar unabhängig davon, ob zwischen den Wörtern ein Leerzeichen oder ein Komma steht. Wir wissen jetzt schon, dass ein Ausrufezeichen einen richtigen Fehler – und nicht nur eine unverständliche Rückgabe – produziert.

Ein ähnlicher Befehl wie `put` ist `alert`: Damit wird allerdings die Rückgabe nicht ins Nachrichtenfenster geschrieben, sondern eine Dialogbox angezeigt. Versuchen wir dies:

```
alert Hallo Welt
```

Abbildung 1.3: Fehlermeldung Nr. 3

Director erwartet einen String, und wir haben etwas anders geliefert? Versuchen wir es mit Anführungszeichen, um anzuzeigen, dass „Hallo Welt“ ein String ist:

```
alert "Hallo Welt"
```

Jawohl: Ein Dialogfenster öffnet sich, und Director sagt „Hallo Welt“. Ein Erfolg. Ob das auch mit dem `put`-Befehl funktioniert?

```
put "Hallo Welt"
-- "Hallo Welt"
put "Hallo Welt!"
-- "Hallo Welt!"
```

Ein Erfolg auf der ganzen Linie! `"Hallo"` ist demnach ein String, `Hallo` dagegen nicht. Wir können vermuten, dass Director `Hallo` als Variablennamen auffasst, da der Wert, der mit `put` ausgegeben wird, `<Void>`, also undefiniert ist. Wir versuchen, die Variable `Hallo` zu definieren:

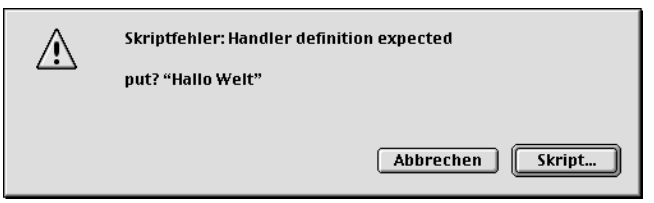
```
Hallo = "Hallo Welt!"
put Hallo
-- "Hallo Welt!"
```

Im Nachrichtenfenster können wir nun unser „Hallo Welt!“ erzeugen – aber ist das schon ein Programm? Vielleicht sollten wir ein Filmskript anlegen: Skripte sind Darsteller in Director und werden beim Sichern dann natürlich mitgesichert. Strg/⌘+0 öffnet das Skriptfenster, und wir schreiben unser im Nachrichtenfenster getestetes Skript:

```
put "Hallo Welt!"
```

Da steht es nun und tut keinem etwas. Wie erhalten wir die nächste Fehlermeldung? Wir versuchen es mit der Kompilierungstaste (dem Blitz-Icon oben im Skriptfenster), und tatsächlich:

Abbildung 1.4: Fehlermeldung Nr. 4



Director erwartet mal wieder etwas anderes von uns. Eine Handler-Definition, also etwas, was mit `on` anfängt: Wie wäre es mit `on hello`?

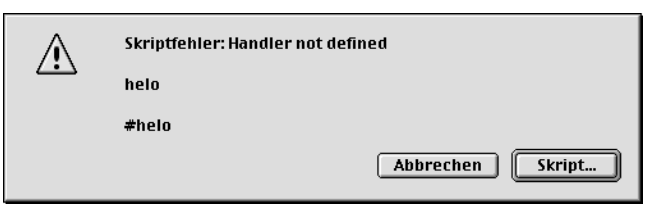
```
on hello
  put "Hallo Welt!"
end
```

Wieder die Kompilierungstaste drücken: Keine Fehlermeldung! Nun bleibt nur die Frage, wie wir unser Mini-Programm ausführen?

Wir versuchen es im Nachrichtenfenster:

```
helo
```

Abbildung 1.5: Und Fehlermeldung Nr. 5!



Der Handler ist nicht definiert? Kein Wunder, wir haben uns vertippt. Neuer Versuch:

```
hello
-- "Hallo Welt!"
```

Gelungen! Das Filmskript reagiert auf einen Aufruf aus dem Nachrichtenfenster. (Falls es das bei Ihnen nicht tut: Überprüfen Sie, ob das Skript tatsächlich vom Typ „Filmskript“ ist, indem Sie mit Strg/⌘+I die Darstellerinformation aufrufen.)

Was aber, wenn wir nicht immer ins Nachrichtenfenster tippen wollen, um unser Programm auszuführen? Wir benötigen einen Event-Handler, etwas, das unseren `hello`-Handler automatisch ausführt. Wir können unser Filmskript wie folgt erweitern:

```
on startMovie
  hello
end

on hello
  alert "Hallo Welt!"
end
```

Wenn Sie jetzt den Film starten Strg+Alt+P / ⌘+⌥+P, erhalten Sie keine Fehlermeldung, sondern den „Hallo Welt!"-Alert. Geschafft!

Sie finden den Film *hello.dir* auf der CD-ROM.

Diskussion

Die Fehlermeldungen in Director mögen häufig nicht sehr aussagekräftig sein – sie führen einen doch meist zur Fehlerquelle. Wir haben im Experiment eine Reihe von Meldungen provoziert, während sie im „Skripting-Alltag" ganz von alleine kommen: Undefinierte Variablen und Handler-Aufrufe, Tippfehler, unvollständige Konstrukte (wie `if … then … end if`) und vieles mehr können Fehlerquellen sein.

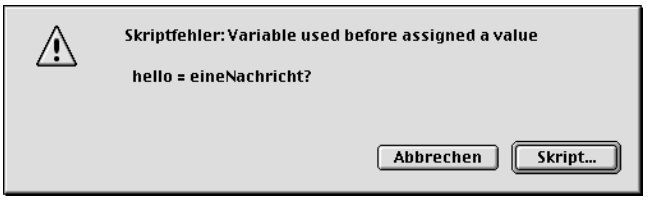
Zu folgenden Zeitpunkten meldet Director Skriptfehler:

- Bei der Skripteingabe: Automatische Einrückungen und Syntax-Kolorierung sind Hinweise darauf, dass Director Ihre Skript-Terminologie und bestimmte Konstrukte erkannt hat. Skripte in diesem Buch sind ebenso eingerückt (wenn auch leider nicht koloriert).
- Beim Kompilieren des Skripts: Die meisten Tippfehler werden bereits dann angemahnt, wenn Sie die Kompilierungstaste drücken (empfehlenswert) oder das Skript schließen. Einige Beispiele sahen Sie im Experiment.

  Ein beliebter Fehler ist, Variablen zu benutzen, die nicht definiert sind. Das folgende Filmskript wird beim Aufruf einen Fehler produzieren:

  ```
  on test
    hello = eineNachricht
    put hello
  end
  ```

Abbildung 1.6: Und noch eine Fehlermeldung



`eineNachricht` ist nicht definiert, soll aber der Variablen `hello` zugewiesen werden. So funktioniert es, da `eineNachricht` zunächst definiert wird:

```
on test
  eineNachricht = "Hallo Welt!"
  hello = eineNachricht
  put hello
end
```

Ein Fragezeichen im Lingotext der Fehlermeldung weist häufig auf die Stelle hin, die von Director nicht interpretiert werden konnte.

- Beim Ausführen des Skripts: Auch wenn sich ein Skript anstandslos kompilieren lässt, kann es noch Fehler enthalten, die Director erst bemerkt, wenn ein Skript ausgeführt wird.

  Das folgende Skript kompiliert, wirft aber einen so genannten Runtime-Fehler (vgl. Abbildung 1.5):

```
on startMovie
  helo
end

on hello
  alert "Hallo Welt!"
end
```

1.2 Experiment 2: Bekanntes wieder finden

Je nach Ihren Erfahrungen gibt es unterschiedliche Dinge, von denen Sie einfach wissen, wie sie funktionieren: Mathematische Funktionen beispielsweise verhalten sich gemeinhin wie erwartet, sie werden in den verschiedenen Programmiersprachen nur unterschiedlich ausgedrückt.

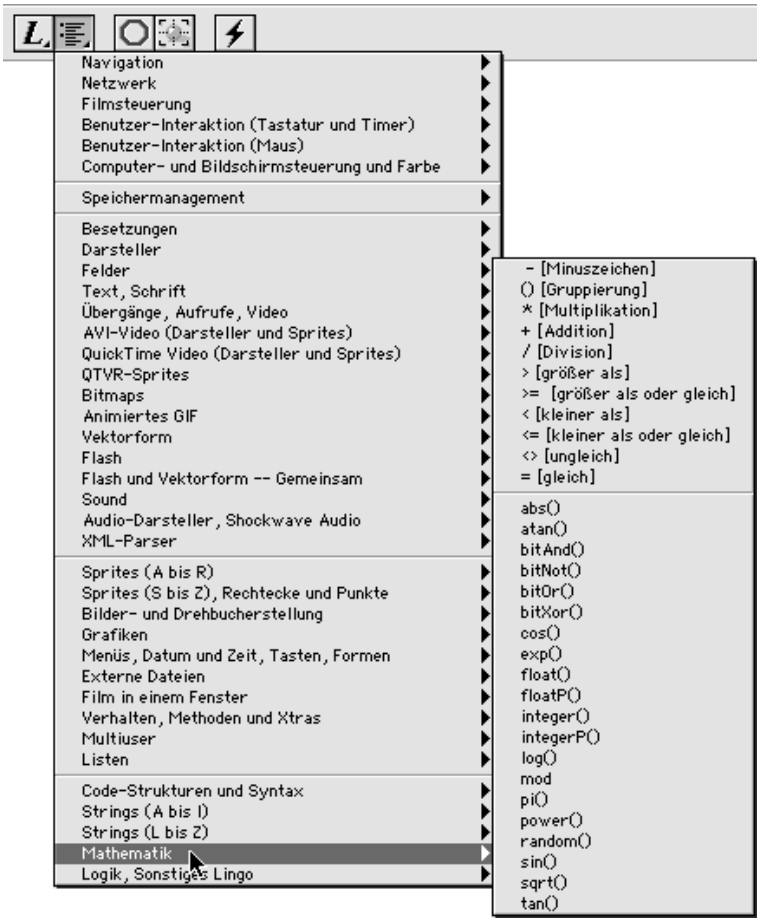


Abbildung 1.7: Im Lingo-Menü finden Sie thematisch sortierte Aufklapper, die Sie einmal auf Bekanntes hin durchforsten sollten.

Fügen Sie aus dem Lingo-Menü (vgl. Abbildung 1.7) einen Befehl ins Skript- oder Nachrichtenfenster ein, so erhalten Sie auch einen Hinweis auf die Verwendung: Die Funktion `power()` wird beispielsweise als `power(Basis, Exponent)` eingefügt. Sie können also im Nachrichtenfenster den mathematischen Term 2^8 wie folgt testen:

```
put power(2, 8)
-- 256.0000
```

Weitere Beispiele aus dem Mathematik-Menü:

```
put (7 + 3.5)*10
-- 105.0000
put abs(-7.5)
-- 7.5000
put float("1")
-- 1.0000
put integer(1.532)
-- 2
```

```
put 8 mod 3
-- 2
put sqrt (256)
-- 16
put pi()
-- 3.1416
put cos(0)
-- 1.0000
put cos(pi())
-- -1.0000
put sin(90 * pi()/180)
-- 1.0000
put tan(0)
-- 0.0000
put tan(pi()/2)
-- 1.63312393531954e16
```

Versuchsaufbau

Ziehen Sie auf der Bühne zwei Felddarsteller auf; das entsprechende Werkzeug befindet sich in der Werkzeugleiste. Stellen Sie das obere Feld im Darstellerinfo (Strg / ⌘+I) auf editierbar und nennen Sie es „Eingabe“. Das untere Feld soll „Ausgabe“ heißen. Legen Sie außerdem mit dem Button-Werkzeug einen einfachen Button mit der Beschriftung „berechnen“ an.

Durchführung

Unser Mini-Taschenrechner (*Minicalc.dir*) soll die Formel, die wir ins obere Feld eintragen, berechnen und das Ergebnis ins untere Feld stellen. Dafür benötigen wir nur ein kleines Skript auf dem „berechnen“-Button:

```
on mouseUp me
  ergebnis = value(member("Eingabe").text)
  member("Ausgabe").text = string(ergebnis)
end
```

`member("Eingabe").text` ist der Textinhalt des oberen Felddarstellers. Die Funktion `value()` interpretiert die eingegebene Formel, als ob es sich dabei um ein Lingo-Skript handelt: Der Variable `ergebnis` wird also das Ergebnis einer Lingo-Berechnung zugewiesen. Um dieses im Darsteller „Ausgabe“ sichtbar zu machen, müssen wir es zunächst in einen String (eine Zeichenkette) umwandeln. Dazu dient die Funktion `string()`.

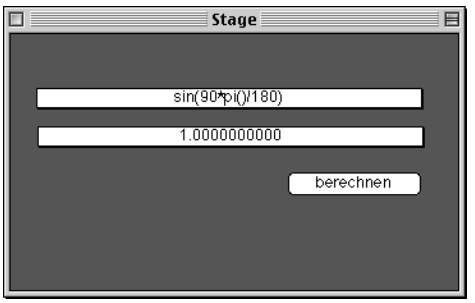


Abbildung 1.8: Ein kleiner selbst gebauter „Taschenrechner“

Standardmäßig stellt Director bei Fließkommazahlen 4 Nachkommastellen dar. Wollen Sie diese Voreinstellung ändern, so setzen Sie die `floatPrecision` auf einen höheren Wert (z.B. in einem Filmskript):

```
on startMovie
  the floatPrecision = 10
end
```

Das beeinflusst nur die Darstellung, nicht die Rechengenauigkeit des Programms.

Falls Sie Lust bekommen haben, einen „richtigen“ Taschenrechner zu bauen, finden Sie einen ersten Ansatz im Film *Calc.dir*, der sich ebenfalls auf der CD-ROM befindet.

1.3 Experiment 3: Kleine Bereiche beackern

In Experiment 2 haben wir die Darstellereigenschaft `text` verwendet, um den Inhalt des Felddarstellers „Eingabe“ zu lesen und das Ergebnis unserer Berechnungen in den Darsteller „Ausgabe“ zu schreiben:

```
einWert = member("Eingabe").text
member("Ausgabe").text = string(andererWert)
```

Feld- und Textdarsteller sind zwei Darstellertypen, bei denen es viel zu entdecken gibt: ein Forschungsgebiet, das wir in diesem Experiment ein bisschen anreißen wollen.

Im Lingo-Menü finden Sie die entsprechenden Einträge in der Liste der Darstellertypen (vgl. Abbildung 1.9).

Abbildung 1.9:
Ausschnitt aus dem Lingo-Menü im Skript- und Nachrichtenfenster: Darsteller-Typen

Feld- und Textdarsteller

Ein bisschen Historie hilft uns beim Verständnis des Unterschieds zwischen den beiden Darstellertypen. Bis Director 5 gab es zur Darstellung von Text nur Felddarsteller. Ein neuer Darstellertyp „Text" bot ab Director 6 weich gezeichneten („anti-aliased") Text, gleichzeitig geriet das Lingo-Inventar zur Steuerung der beiden Darstellertypen kräftig durcheinander. In den Versionen 7 bis 8.5 ist es immer noch so, dass manche Eigenschaften nur für Feld-, andere nur für Textdarsteller gültig sind. Allerdings hat Macromedia sich bemüht, die Unterschiede zu verringern.

Wozu gibt es denn dann überhaupt unterschiedliche Darstellertypen? Weil sie unterschiedlichen „Ballast" mit sich schleppen: Felddarsteller sind geeignet als einfache Ein- und Ausgabefelder, bei denen das Aussehen, der Font und die Typografie nicht wichtig sind. Sie benötigen nicht viel mehr Speicherplatz als die Textinformation, die sie beinhalten. Textdarsteller dagegen enthalten eine Vielzahl an Zusatzinformationen und sind längst nicht so „schlank" wie Felder, bieten aber eine entsprechende Zahl an Einstellungsmöglichkeiten. Textdarsteller werden von der Grafik-Engine von Director dargestellt (auch verzerrt und rotiert, s.u.), während Felddarsteller die Möglichkeiten des Betriebssystems nutzen.

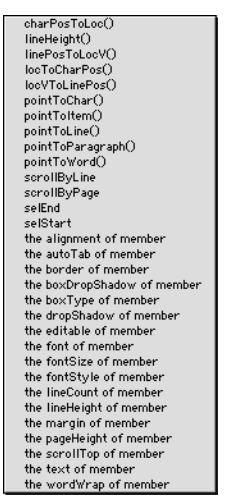


Abbildung 1.10:
Das Aufklappmenü unter „Felder" im Lingo-Menü: Teilweise gelten die Eigenschaften und Funktionen aber auch für Textdarsteller.

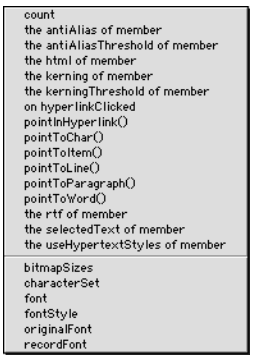


Abbildung 1.11:
Das Ausklappmenü „Text, Schrift“; die Einträge sind nur für Textdarsteller gültig

Versuchsaufbau

Wir benötigen nicht viel für diesen Versuch: Bitte legen Sie in einem leeren Film einen Felddarsteller an (z.B. mit Strg/⌘+8), nennen Sie ihn „Feld1“ und schreiben Sie einen kurzen Text hinein. Mit Strg/⌘+I erhalten Sie die Darstellereigenschaften: Achten Sie bitte darauf, dass „Rahmen: Anpassen“ ausgewählt ist.

Bitte duplizieren Sie den Darsteller (Menü Bearbeiten/Duplizieren) und benennen Sie die Kopie „Feld2“. Dann legen Sie einen Textdarsteller an (Strg/⌘+6), nennen ihn „Text“ und setzen die Option „Rahmen“ ebenfalls auf „Anpassen“. Außerdem wollen wir die Option „Anti-Alias“ (also die Weichzeichnung der Textkanten) auf „Gesamter Text“ setzen.

Wir haben schließlich alle drei Darsteller auf die Bühne gezogen (vgl. Abbildung 1.12).

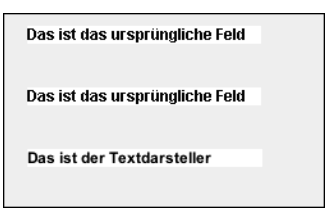


Abbildung 1.12:
Zustand der Darsteller zu Beginn des Experiments

Durchführung

Nun wollen wir einige der Darsteller-Eigenschaften aus den Lingo-Menüs im Nachrichtenfenster ausprobieren. Um nicht so viel tippen zu müssen, speichern wir in der Variablen `f` eine Referenz auf den Darsteller „Feld2“. Nach dieser Zuweisung sind `f.font` und `member("Feld2").font` gleichbedeutend.

```
f = member("Feld2")
f.font = "Verdana"
f.fontsize = 14
```

```
f.fontstyle = "plain"
f.fontstyle = "italic"
f.border = 1
f.margin = 5
f.boxdropshadow = 2
f.text = "Das ist das veränderte Feld"
```

Das Ergebnis sehen Sie in Abbildung 1.14. Nun dasselbe mit dem Textdarsteller:

```
t = member("Text")
t.font = "Verdana"
t.fontsize = 14
t.fontstyle = [#italic]
t.border = 1
```

Beim letzten Aufruf erhalten Sie eine Fehlermeldung wie in Abbildung 1.13

```
t.text = "Das ist der veränderte Textdarsteller"
```

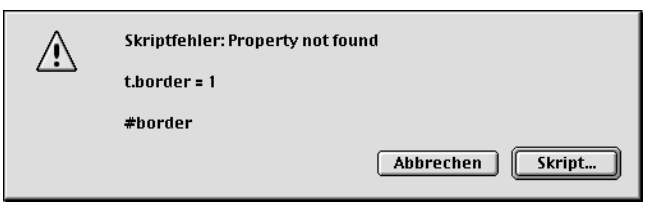


Abbildung 1.13: Aus dieser und entsprechenden Fehlermeldungen sehen Sie, dass bestimmte Eigenschaften bei Textdarstellern nicht existieren. Das gilt neben `border` auch für `margin` und `boxDropshadow`.

Wir erhalten ein Ergebnis ähnlich wie das in Abbildung 1.14 – und haben mit diesem Experiment einige Eigenschaften von Feld- und Textdarstellern kennen gelernt.

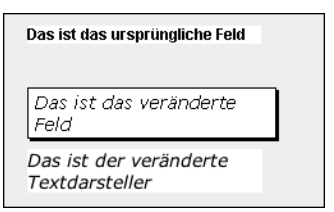


Abbildung 1.14: Die Darsteller am Ende des Experiments

Zu ergänzen bleibt noch, dass Sie gleich weitermachen können: Probieren Sie andere Darstellertypen aus, oder testen Sie einmal alle Eigenschaften, die im Aufklapper „Darsteller“ stehen, in Ihrem Beispielfilm mit den Text- und Felddarstellern.

1.4 Experiment 4: Eine Testumgebung für die Animation von Sprite-Eigenschaften

Sprites sind ein wichtiger Baustein in der Programmarchitektur von Director. Sie sind nicht dasselbe wie ein anderer wichtiger Baustein: die Darsteller. Die Zusammenhänge sind wie folgt:

- Die umfassendste Einheit ist die Director-Produktion (die aus einem oder mehreren Filmen besteht, evtl. auch noch aus externen Besetzungen und externen Medien)
- Ein Film besteht aus einer oder mehreren Besetzungen, die intern oder extern sein können.
  - Eine Besetzung enthält zumeist zahlreiche Darsteller: Texte, Bilder, SWF-Dateien, Sounds, Links zu Video-Dateien, Skripte u.a.m.
- Ein Film hat aber auch ein Drehbuch, in dem meist zahlreiche Sprites enthalten sind.
  - Ein Sprite ist ein Repräsentant eines Darstellers im Drehbuch, eine „Instanz". Wird ein Sprite verändert, so ändert sich der Darsteller *nicht*.
  - In der Logik Directors ist es nie der Darsteller, der im Drehbuch auftaucht, sondern ein Sprite, das zum Zeitpunkt x einen bestimmten Darsteller repräsentiert. (Die Eigenschaft `member` des Sprites enthält eine Referenz auf einen bestimmten Darsteller.)
- Die Bühne ist die Visualisierung des Drehbuchs zu einem bestimmten Zeitpunkt x – nämlich des Frames, in dem sich der Abspielkopf des Drehbuchs gerade befindet.

Während also die Besetzung die vielen Darsteller bereithält und organisiert, sind das Drehbuch und die Bühne die Domäne der Sprites: Sobald ein Darsteller auf die Bühne (oder ins Drehbuch) gezogen wird, wird eine Reihe von Informationen ins Drehbuch eingetragen, unter anderem die Informationen, dass das Sprite in Kanal x den Darsteller y enthält, dass dieses Sprite von Frame 1 bis 28 sichtbar ist und dass es an einer bestimmten Position auf der Bühne angezeigt werden soll.

Wenn wir ein Skript aus der Besetzung auf ein Element auf der Bühne ziehen, so wird es stets dem Sprite, nicht dem Darsteller zugeordnet. (Die veralteten Darstellerskripte werden uns in diesem Buch nicht beschäftigen).

Was also anfangen mit den Sprites? Wir wollen in diesem Experiment einige ihrer Eigenschaften kennen lernen und dynamisch verändern. Damit wir nicht jedes Mal von vorn anfangen, bauen wir uns ein „Animationskorsett", in dem wir dann unsere Tests ausführen.

Versuchsaufbau

Bitte legen Sie in einem leeren Film einen Textdarsteller mit dem Namen „Text“ an (Strg / ⌘+6). Schreiben Sie einen kurzen Text hinein. Dann legen Sie einen Bitmap-Darsteller mit den Malwerkzeugen im Malfenster (Strg / ⌘+5) an. Das Ergebnis muss keinen künstlerischen Ansprüchen genügen!

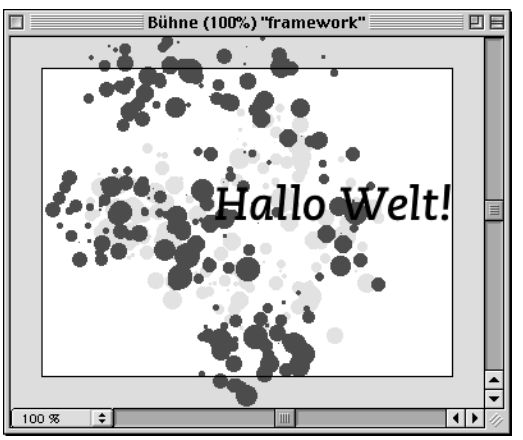


Abbildung 1.15: Die Bühne unseres Versuchsfilms. Für beide Sprites ist der Inkmodus „Hintergund Transparent“ eingeschaltet (mit Strg / ⌘-Klick auf das Sprite).

Einige der Eigenschaften, die wir gleich per Skript verändern werden, können Sie im Eigenschaften- bzw. Spriteinspektor (vgl. Abbildung 1.16) ausprobieren, wenn Sie die Darsteller als Sprites auf die Bühne ziehen.

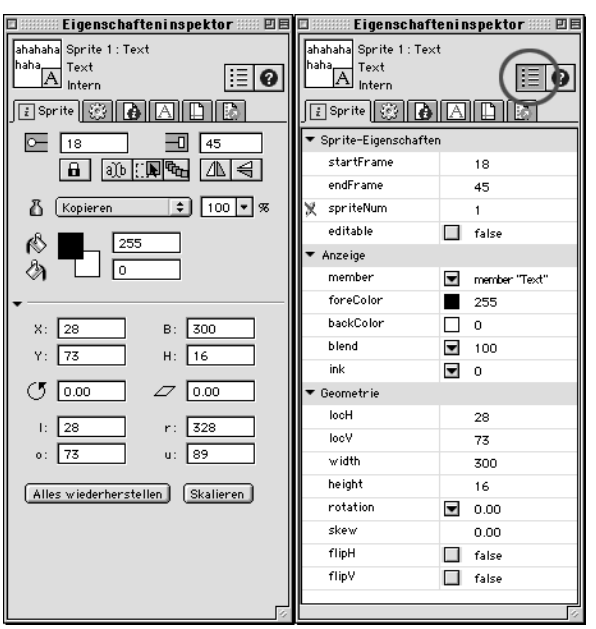


Abbildung 1.16: Der Eigenschaftsinspektor in Director 8 (oder der Spriteinspektor in früheren Versionen) zeigt Ihnen einige wichtige Sprite-Eigenschaften. In der Listendarstellung (rechts) sehen Sie auch die Lingo-Äquivalente zu den Symbolen links.

Durchführung

Mit dem folgenden Skriptgerüst wollen wir eine Reihe von Sprite-Eigenschaften animieren. Stören Sie sich zum jetzigen Zeitpunkt nicht an Dingen, die Ihnen unverständlich vorkommen. Wichtig ist hier lediglich die jeweilige Sprite-Eigenschaft, die wir in der Folge an die gekennzeichnete Position im Film *framework.dir* schreiben werden.

Das Gerüst:

```
property minWert, maxWert, step
property val

on beginSprite me
  minWert = 0
  maxWert = 360
  step = 3
  val = minWert
end

on prepareFrame me
  sprite(me.spriteNum).rotation = val
  ---------------------^^^^^^^^-------
  val = val + step
  val = max(minWert, val mod (maxwert + step))
end
```

Das Skript dient dazu, die Eigenschaft `rotation` des betreffenden Sprites kontinuierlich zu animieren, und zwar indem ein Zähler `val` zwischen den Werten `minWert` und `maxWert` mit der Schrittweite `step` hochgezählt wird und die Spriteeigenschaft bei jedem Framedurchlauf auf den Wert von `val` gesetzt wird. `minWert`, `maxWert` und `step` sind konfigurierbar; außerdem werden wir an der im Skript markierten Stelle auch andere Sprite-Eigenschaften austesten.

Schreiben Sie das Skript in ein Behavior! Diesen Skript-Typ legen Sie an, indem Sie auf das Behavior-Icon des Sprites auf der Bühne klicken oder indem Sie mit Strg / ⌘+O den Skripteditor öffnen und in den Skripteigenschaften (Strg / ⌘+I) den Skripttyp auf „Behavior“ stellen.

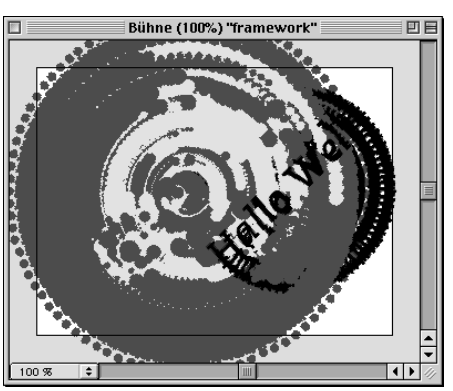


Abbildung 1.17:
Hier haben wir das Rotations-Behavior auf beiden Sprites platziert und zusätzlich die Sprite-Eigenschaft „Spuren“ (trails) eingeschaltet.

Probieren Sie auch einmal folgende Kombinationen aus:

- Spriteeigenschaft: `blend` (Opazität), `minWert`: 0, `maxWert`: 100, `step`: 5
  Das Skript sieht dann so aus:

```
property minWert, maxWert, step
property val

on beginSprite me
  minWert = 0
  maxWert = 100
  step = 5
  val = minWert
end

on prepareFrame me
  sprite(me.spriteNum).blend = val
  val = val + step
  val = max(minWert, val mod (maxwert + step))
end
```

- Spriteeigenschaft: `locH` (horizontale Position), `minWert`: 0, `maxWert`: 300, `step`: 1
- Spriteeigenschaft: `locV` (vertikale Position), `minWert`: 50, `maxWert`: 150, `step`: 5
- Spriteeigenschaft: `width` (Breite), `minWert`: 50, `maxWert`: 300, `step`: 5
- Spriteeigenschaft: `height` (Höhe), `minWert`: 50, `maxWert`: 300, `step`: 1
- Spriteeigenschaft: `color` (Vordergrundfarbe), `minWert`: 0, `maxWert`: 255, `step`: 1
  Bitte ändern Sie dafür die folgende Zeile:

```
sprite(me.spriteNum).color = paletteIndex(val)
```

  und probieren Sie auch einmal das Folgende aus:

```
sprite(me.spriteNum).color = rgb(val, 255, 255)
```

- Spriteeigenschaft: `loc` (Position), `minWert`: 0, `maxWert`: 200, `step`: 1, und ändern Sie die Zeile wiederum in:

```
sprite(me.spriteNum).loc = point (val, val)
```

  Um beispielsweise eine Sinuskurve auf die Bühne zu zeichnen, ändern Sie das Skript nur minimal ab:

```
property minWert, maxWert, step
property val

on beginSprite me
  minWert = 0
  maxWert = 320
  step = 2
  val = minWert
end
```

```
on prepareFrame me
  sprite(me.spriteNum).locH = val
  sprite(me.spriteNum).locV = 100 + 50 * sin(val * pi()/100.0)
  val = val + step
  val = max(minWert, val mod (maxwert + step))
end
```

Schalten Sie dabei die Eigenschaft „Spuren“ des Sprites ein, so ergibt sich ein Bild wie in Abbildung 1.18.

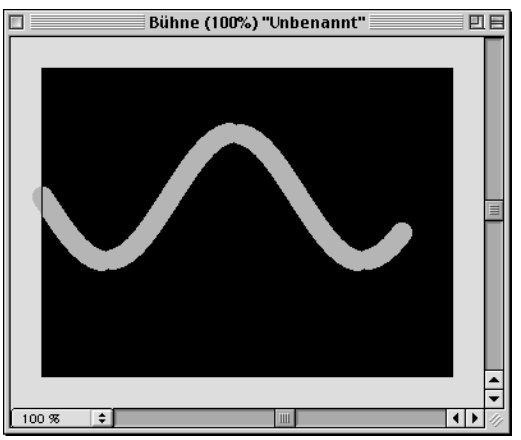


Abbildung 1.18: Hier wurde ein heller Punkt auf einer Sinus-Bahn bewegt.

Bei Ihren Versuchen sollten Sie auch an folgenden Parametern Änderungen vornehmen:

- Ändern Sie im Steuerpult (vgl. Abbildung 1.19) die Geschwindigkeit Ihres Films: Probieren Sie einmal 10, einmal 100 und einmal 1000 Bilder pro Sekunde (fps).

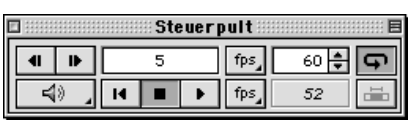


Abbildung 1.19: Die Bildrate des Films wirkt sich direkt auf die Geschwindigkeit der Animation aus.

- Ändern Sie auch die Sprite-Eigenschaften „Spuren“ (`trails`) und den Farbeffekt (`ink`). Beide Einstellungen finden Sie im Eigenschaften- bzw. Spriteinspektor.

Auf der CD-ROM finden Sie im Film *SpriteProps.dir* einige Beispiele. Weitere Experimente rund um Sprite-Properties enthält Kapitel 9.

Mausklicks

Mausklicks sind das A und O eines Multimedia-Projekts. Mit den wenigen Möglichkeiten, die der User mit seiner Maus hat, möchte er auf dem Bildschirm auch etwas ausrichten. Die Reaktion auf Mausklicks ist folglich eine der wichtigsten Aufgaben für Sie als „Lingo Laborant" – und sie ist denkbar einfach umzusetzen, so dass dieses Thema unser Einstieg in die Laborarbeit sein soll.

Was aber ist ein Mausklick? Bewirkt jeder Klick dasselbe? Wie legen Sie Bereiche auf der Bühne fest, die für Mausklicks sensibel sind, und wie regeln Sie es, dass im einen Fall auf einen Mausklick nur einmal, in einem anderen Fall aber bei jedem wiederholten Klick erneut reagiert wird?

In den folgenden Beispielen sollen die User-Klicks Animationen in Gang setzen. Wenn Sie einen Darsteller auf der Bühne platziert haben, wird eine Ihrer ersten Entscheidungen sein, festzulegen, wohin der User klicken soll, um seine Animation zu starten: auf den Darsteller selbst oder irgendwohin auf die Bühne? Und soll die Animation bei einem Klick lediglich einmal ausgelöst werden, oder kann sie immer wieder starten?

Damit ein Skript auf Mausklicks reagieren kann, muss uns Director eine Möglichkeit zur Verfügung stellen, diese Eingaben des Anwenders zu bemerken und zu verarbeiten. Tatsächlich löst jeder Klick programmintern „Events" aus, die wir mit unseren Skripten „auffangen" können. Der Ort, wo wir das tun können, sind die Event-Handler (wie `on mouseUp` oder `on mouseDown`). Außerdem gibt es auch noch Systemeigenschaften (wie `the mouseDown`), mit denen wir an jeder beliebigen Stelle eines Skriptes den Zustand der Maus überprüfen können.

Wollen wir auch noch unterscheiden, ob zum ersten oder bereits zum wiederholten Mal geklickt wurde, müssen wir in unserem Skript über Mausklicks Buch führen. Für diese Aufgabe werden wir uns mit zwei Variablentypen auseinander setzen.

Die folgenden Experimente benutzen alle denselben Versuchsaufbau. Da dies unsere erste Versuchsreihe ist, werden wir uns mit dieser Vorbereitung Zeit lassen und Schritt für Schritt die Bestandteile des Filmes zusammenfügen, mit dem wir anschließend unsere Mausklick-Experimente durchführen wollen. Das Ausgangsmaterial finden Sie im *Material*-Verzeichnis zu Kapitel 2 auf der CD-ROM.

2.1 Versuchsaufbau

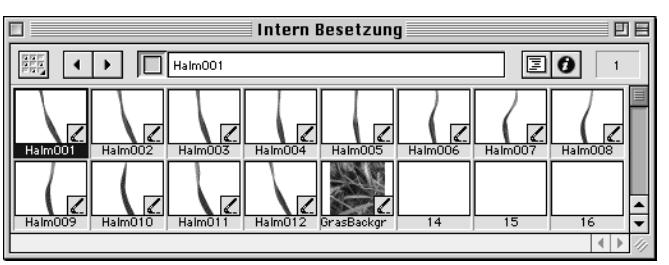
Um einen neuen Film in Director zu öffnen, gehen Sie im DATEI-Menü unter NEU/FILM Strg/⌘+N. Den neuen Film konfigurieren Sie im Dialogfeld FILMEIGENSCHAFTEN Strg/⌘+⇧+D und geben Sie diese Werte ein: Breite: 250, Höhe: 235, Bühnenposition: „Zentriert“.

Abbildung 2.1: Das Dialogfeld Filmeigenschaften

In Ihren leeren Film importieren Sie nun eine Serie von Einzelbildern, die wir in den Experimenten bei Mausklick animieren wollen.

Der Import Ihrer Darsteller geschieht am praktischsten bei geöffnetem Besetzungsfenster Strg/⌘+3. Importieren Sie die 12 Einzelbilder aus dem Verzeichnis *Material* mit Strg/⌘+R oder dem Menübefehl DATEI/IMPORTIEREN... Daraufhin erscheint ein Dialogfeld, in welchem Sie den Ordner mit den Darstellern auswählen und mit Klick auf „Importieren“ in die Besetzung des Films übertragen. Sie können die Darsteller auch einfach aus ihrem Verzeichnis in das Besetzungsfenster ziehen.

Abbildung 2.2: Das Besetzungsfenster nach dem Import der Darsteller



Sie können die Animation (die unterschiedlichen Phasen des Grashalm-Bilds) schon einmal testen, indem Sie auf den Darsteller „Halmoo1“ doppelklicken und im Malfenster über die Pfeil-Buttons von Bild zu Bild springen.

Öffnen Sie dann das Drehbuch Strg / ⌘+4 und ziehen Sie den Wiesenhintergrund (Darsteller 13) aus der Besetzung in den ersten Spritekanal und den ersten Grashalm (Darsteller 1) aus der Besetzung in den zweiten Spritekanal.

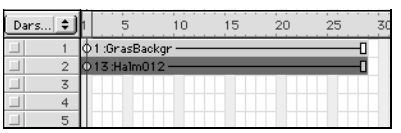


Abbildung 2.3: Das Drehbuch nach dem Einfügen der Darsteller

Director legt automatisch von jedem Darsteller ein Sprite mit 28 Frames ins Drehbuch. Falls Sie diesen voreingestellten Wert ändern wollen, gehen Sie im Datei-Menü unter Voreinstellungen/Sprite; im erscheinenden Dialog können Sie den gewünschten Wert eingeben.

Öffnen Sie im Drehbuch das Aufklappmenü für die Sprite-Anzeige (vgl. Abbildung 2.4).

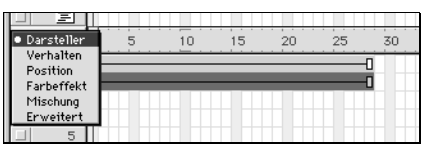


Abbildung 2.4: Hier wählen Sie aus, was Ihnen in den Sprite-Kanälen angezeigt werden soll.

Wenn Sie hier auf „Darsteller“ schalten, werden die Sprites mit den Darsteller-Namen versehen. Nach dem Import der beiden Darsteller müsste Ihr Drehbuch aussehen wie in Abbildung 2.5.

Abbildung 2.5: Die Darstellernamen werden als Sprite-Beschriftung angezeigt.

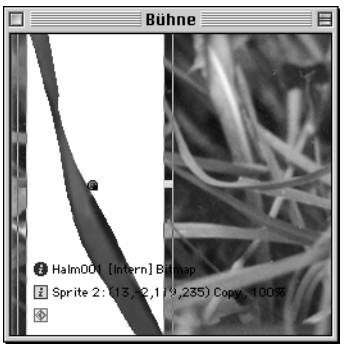


Abbildung 2.6:
Die Bühne nach dem Einfügen der Darsteller

Auf der Bühne (vgl. Abbildung 2.6) erscheint die rechteckige Einfassung des Grashalm-Hintergrunds weiß, anstatt den Blick auf den Bühnenhintergrund freizugeben. Wenn Sie mit gedrückter Strg / ⌘-Taste den Grashalm auf der Bühne anklicken, öffnen Sie ein PopUp-Menü, aus welchem Sie „Hintergrund transparent“ auswählen.

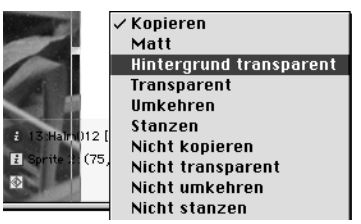


Abbildung 2.7:
Auswahl eines transparenten Hintergrunds

Damit ist der Versuchsaufbau abgeschlossen, und wir wollen uns noch um einige grundlegende Fragen der Skripterstellung kümmern.

Sprites mit „Verhalten“ ausstatten

Wie gehen Sie nun vor, um an eines der eben erstellten Sprites ein Skript anzufügen? Solche Sprite-Skripten heißen in Director Verhalten (Behaviors), und sie werden im Verhaltensinspektor (Director 7) oder im Eigenschaften-Inspektor (Director 8) erstellt. Wählen Sie einmal testweise den zweiten Spritekanal (Grashalm) aus und öffnen Sie den Verhaltensinspektor (Fenster/Inspektoren/Verhalten oder * auf der numerischen Tastatur). Klicken Sie im Verhaltensinspektor auf das Pluszeichen und wählen Sie „Neues Verhalten“.

Abbildung 2.8:
Drücken Sie diese Taste, um ein neues Verhalten zu erstellen.

Benennen Sie das Verhalten im erscheinenden Dialog (z.B.„test1“). Wenn Sie den Dialog bestätigen, erscheint das Skript-Fenster, in welches Sie einen Skripttext eintippen können.

In diesem Fenster gibt es diverse Erleichterungen zum Erstellen von Skripten. So können Sie z.B. Event-Handler aus einem vorgefertigten Lingo-Vokabular abrufen. Drücken Sie dazu die Taste mit dem Skript-Symbol und gehen Sie dort unter BENUTZER-INTERAKTION (MAUS) zu `on mouseDown`.

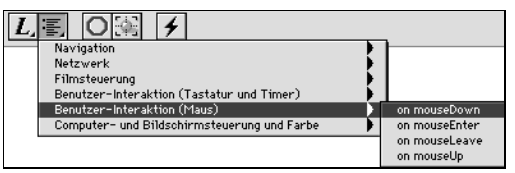


Abbildung 2.9: In das Skriptfenster können Sie Lingo-Begriffe aus dem thematisch sortierten Lingo-Menü einfügen.

Beim Loslassen erscheint der erste Teil Ihres Skripts schon geschrieben im Skript-Fenster:

```
on mouseDown

end mouseDown
```

Ergänzen Sie folgende Zeile und drücken Sie zum Schluss die Kompilierungstaste (Blitz):

```
on mouseDown
  beep
end mouseDown
```

Abbildung 2.10: Befinden sich in Ihrem Skript Syntax-Fehler, werden Sie nach dem Drücken dieser Taste darauf hingewiesen.

Sichern Sie den Film und spielen Sie ihn mit der ↵-Taste ab. Wenn Sie jetzt auf das Grashalm-Sprite klicken, ertönt der System-Beep-Ton.

Wenn Sie später einmal an dem Behavior weiterarbeiten möchten, markieren Sie den betreffenden Eintrag im Verhaltens- oder Eigenschafteninspektor und drücken Sie dann die Taste mit dem Skriptsymbol.

Abbildung 2.11:
Die Skripttaste auf dem Behavior-Tab des Eigenschaftsinspektors öffnet in Director 8 das Skript im Skriptfenster.

Haben Sie ohnehin ein Skript im Skriptfenster geöffnet, können Sie alle Skripte auch über die Pfeil-Tasten erreichen.

Abbildung 2.12:
Im Skriptfenster können Sie mit den Pfeiltasten durch alle Skripte blättern.

Wenn Sie das Skript „Test1“, das wir eben angelegt haben, im Verhaltensinspektor anzeigen lassen (Director 7 und 8) und den mittleren Bereich des Inspektors öffnen, sehen Sie eine kombinierte Darstellung, die die Bereiche „Ereignisse“ (Events) und „Aktionen“ umfasst (vgl. Abbildung 2.13). Hier bietet sich Ihnen ein weiterer Weg, einfache Skripts durch die Auswahl aus den beiden Aufklappmenüs (Plus-Buttons) zusammenzustellen. Abbildung 2.14 zeigt einige der Möglichkeiten, die die Menüs zur Verfügung stellen.

Abbildung 2.13:
Im mittleren Bereich des Verhaltensinspektors können Sie auf einfache Weise „Ereignissen“ Skriptaktionen zuordnen.

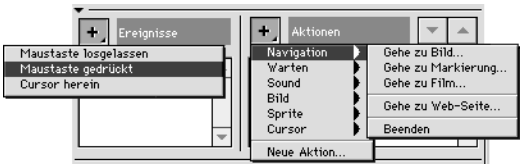


Abbildung 2.14: Einige der Ereignisse und Aktionen, aus denen Sie einfache Skripte erstellen können

Die Bibliothekspalette

Director wird mit einer großen Anzahl „fertiger" Skripte ausgeliefert, die eine Vielzahl an Anwendungsfällen abdecken und zumeist von Ihnen konfiguriert werden können. Sie finden diese Skripte in der Bibliothekspalette (Menü Fenster/Bibliothekspalette). Diese Skripte können Sie direkt aus der Palette auf ein Sprite auf der Bühne oder im Drehbuch ziehen, um es zu nutzen.

Abbildung 2.15: Die Taste im Bibliotheksfenster oben links eröffnet Ihnen verschiedene Bereiche, aus denen Sie je nach Bedarf auswählen.

Ereignisse und Event-Handler

Wenn ein Director-Film abläuft, produzieren sowohl Director als auch der Anwender „Ereignisse" oder Events. Ein Event-Handler ist ein Skriptblock, der diese Ereignisse nutzt, um Skriptaktionen auszuführen. Mit Event-Handlern entscheiden Sie also über den Zeitpunkt, zu dem das Programm etwas tun soll:

- Soll ein Skript durch einen Mausklick ausgelöst werden, wählen Sie `on mouseDown` oder `on mouseUp` als Event-Handler.
- Wollen Sie etwas einmalig ausführen, wenn ein Sprite zum ersten Mal angezeigt wird, so ist `on beginSprite` der geeignete Event-Handler.
- Soll während des ganzen Ablaufs, also von Frame zu Frame, etwas überprüft werden oder bei jedem Framewechsel eine Änderung herbeigeführt werden, so sind `on exitFrame` oder `on prepareFrame` die geeigneten Event-Handler.

Mehr zu diesem Thema finden Sie in den folgenden Kapiteln.

Behaviors wiederfinden

Häufig ist es für Director-Einsteiger verwirrend, wie das Programm mit Skripten umgeht. Alle wichtigen Skripttypen sind nämlich Darsteller, die in der Besetzung angelegt werden wie andere Darsteller (Text, Bild, Video etc.) auch. Wenn Sie ein Skript bearbeiten wollen, genügt daher ein Doppelklick auf den entsprechenden Darsteller in der Besetzung.

Andererseits sind Behaviors häufig auch Sprites zugeordnet, und Sie können sie daher auch aufrufen, indem Sie das Behavior-Icon im Info-Overlay eines Sprites anklicken (vgl. Abbildung 2.16). Auch im Drehbuchfenster können Sie im Kontextmenü (Rechts-/Ctrl-Klick auf ein Sprite) die Befehle Skript oder Verhalten aufrufen, um das erste zugeordnete Skript oder die Liste der Skripten angezeigt zu bekommen. In Director 8 ist der Behavior-Tab des Eigenschafteninspektors der naheliegendste Weg, die Behaviorliste eines Sprites zu inspizieren.

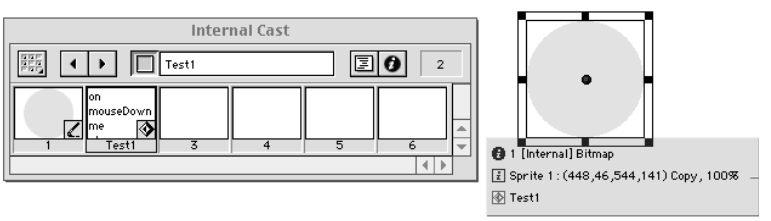


Abbildung 2.16: Behaviors sind Darsteller (die Sie in der Besetzung finden können); und sie sind häufig Sprites zugeordnet, wo Sie sie durch einen Klick auf das unterste kleine Icon im Sprite-Info-Overlay aktivieren können.

Ein Behavior kann auch im Skriptkanal des Drehbuchs platziert werden: Dann genügt ein Doppelklick auf den Eintrag im Skriptkanal, um das Skript zu öffnen.

2.2 Experiment 1: Irgendwo klicken

Im ersten Versuch soll erreicht werden, dass die Animation abläuft, sobald der Anwender irgendwo auf die Bühnenfläche klickt (und erneut bei jedem Klick).

Doppelklicken Sie auf den Skriptkanal (z.B. im Frame 5), um ein neues Behavior anzulegen. Benennen Sie das Behavior mit „Klickvariation“.

Zu Beginn des Behaviors deklarieren Sie eine Property-Variable, die später die abzuspielenden Bilder der Bildfolge zählt und für eine planmäßige Abspielreihenfolge sorgt. Deswegen heißt sie hier `counter`.

```
property counter
```

Sie stellen dann `counter` auf einen Wert ein, nämlich den Startwert der Animation (die Darstellernummer des ersten Bildes):

```
counter = 1
```

Das Zählen des Counters veranlasst die folgende Zeile, die `counter` um 1 nach oben setzt:

```
counter = counter + 1
```

Den neuen Counter nutzen wir, um den Darsteller unseres Grashalm-Sprites (`Sprite 2`) auszutauschen:

```
sprite(2).membernum = counter
```

Was jetzt fehlt, sind die so genannten Ereignisprozeduren oder "Event-Handler", mit denen festgelegt wird, wann welche Aktion erfolgen soll.

So soll der Counter zu Beginn – also wenn das Sprite erstmalig geladen und initialisiert wird – auf einen Startwert gesetzt werden:

```
on beginSprite me
 counter = 12
```

Diese Anweisungen müssen mit einem `end` abgeschlossen werden:

```
on beginSprite me
 counter = 12
end
```

Bei jedem Frame-Wechsel sollen die Veränderung der Property `counter` und der Darstellerwechsel stattfinden:

```
on exitFrame me
 counter = counter + 1
 sprite(2).membernum = counter
end
```

Allerdings wollen wir nicht endlos hochzählen: Das `exitFrame`-Skript soll nur ausgeführt werden, solange `counter` kleiner als 12, also kleiner als die höchste Darstellernummer der Grashalm-Animation, ist:

```
on exitFrame me
  if counter < 12 then
    counter = counter + 1
    sprite(2).membernum = counter
  end if
end
```

Zu Beginn haben wir `counter` auf 12 gesetzt: Es wird also keine Animation stattfinden, da die Bedingung `if counter < 12` nicht zutrifft. Erst ein Mausklick soll zum Abspielen der Animation führen:

```
on mouseUp me
  counter = 1
end
```

So sieht das Behavior „Klickvariation“ im Film *BuehnenklicksWiederholt.dir* im Zusammenhang aus:

```
property counter

on beginsprite me
  counter = 12
end

on mouseUp me
  counter = 1
end

on exitframe me
  if counter < 12 then
    counter = counter + 1
    sprite(2).membernum = counter
  end if
  go to the frame
  go to the frame hält den Abspielkopf am aktuellen Frame an.
end
```

Zum Anschauen des Film stellen Sie den Abspielkopf mit der [0]-Taste der numerischen Tastatur an den Filmanfang und drücken [↵] auf der numerischen Tastatur (oder: [Strg]/[⌘]+[Alt]+[P] für „Play“). Wann immer Sie irgendwo auf die Bühne klicken, wird die Grashalm-Animation abgespielt.

2.3 Experiment 2: Sprite-Klicks

Wenn die Animation immer wieder beim Klick auf ein bestimmtes Sprite ablaufen soll, genügt eine kleine Variation des ersten Experiments. In „SpriteklicksWiederholt.dir“ wurde das Behavior „Klickvariation“ aus dem Skriptkanal entfernt und dann aus der Besetzung auf Sprite 2 gezogen: Wenn auf den Grashalm geklickt wird, bewegt er sich – auf der Bühne bleiben die Klicks folgenlos.

Um unsere Behaviors gleich von Anfang an aufgeräumt zu halten, haben wir allerdings die Zeile `go to the frame` aus dem Behavior „Klickvariation“ gelöscht und in ein Stopper-Skript in den Skriptkanal verfrachtet. Dieses Skript sieht wie folgt aus:

```
on exitFrame
 go to the frame
end
```

Wir werden es immer wieder benötigen, um den Abspielkopf an einem bestimmten Frame festzuhalten. Die Systemeigenschaft `the frame` liefert die momentane Framenummer zurück, wenn wir beim Verlassen des Bildes auf die aktuelle Bildnummer springen (`go to`), bleiben wir also auf dem betreffenden Frame stehen.

Den Director-Film finden Sie als *SpriteklicksWiederholt.dir* auf der CD-ROM 

2.4 Experiment 3: Einmal reagieren

Dieses Experiment modifiziert Experiment 1 geringfügig: Jetzt soll nämlich die Animation nicht bei jedem Klick erneut, sondern nur einmal ablaufen. Weiterhin soll es egal sein, wohin geklickt wird: Wir können also das Skript „Klickvariation“ im Skriptkanal weiterverwenden und wo nötig modifizieren.

Wir führen eine zweite Property-Variable `geklickt` ein, die nur zwei Zustände kennt: `false` (es wurde noch nicht geklickt) und `true` (es wurde bereits einmal geklickt). `geklickt` wird zunächst als Property deklariert und im `beginSprite`-Handler auf den Startwert gesetzt.

```
property counter
property geklickt

on beginsprite me
  counter = 12
  geklickt = false
end
```

Nun müssen wir noch dafür sorgen, dass die Mausaktion nur dann ausgeführt wird, wenn zuvor noch nicht geklickt wurde, und wir setzen `geklickt` dann entsprechend auf `true`:

```
on mouseUp me
  if geklickt = false then
    counter = 1
    geklickt = true
  end if
end

on exitframe me
  if counter < 12 then
    counter = counter + 1
    sprite(2).membernum = counter
  end if
  go to the frame
end
```

Das Skript funktioniert dann genauso wie das in Experiment 1 – aber eben nur einmal. Bei allen weiteren Klicks ist `geklickt` schon `true` und der `mouseUp`-Handler wird nicht ausgeführt.

Den Director-Film finden Sie als *BuehnenklicksEinmalig.dir* auf der CD-ROM.

2.5 Experiment 4: Einmal reagieren nach Sprite-Klick

Soll Ihre Animation nur einmal ablaufen, wenn mit der Maus auf ein bestimmtes Sprite geklickt wird, so können Sie das Skript aus Experiment 3 wiederum aus dem Skriptkanal löschen und – ohne die Zeile `go to the frame` – entsprechend einem Sprite zuweisen. Es wird dann nur noch Klicks im Sprite-Bereich als Auslöser akzeptieren, sich ansonsten aber genauso wie in Experiment 3 verhalten.

Den Director-Film finden Sie als *SpriteklicksEinmalig.dir* auf der CD-ROM.

2.6 Experiment 5: Maus gedrückt halten

Eine Aktion genauso lange auszuführen, wie die Maus gedrückt gehalten wird – diese Anforderung soll uns nun beschäftigen. Wir wollen zwei Ansätze vorstellen, die die Anforderung unterschiedlich gut erfüllen. Ein Stopper-Skript soll den Abspielkopf beispielsweise auf Frame 5 anhalten.

Variante 1 fasst das „Gedrückthalten" der Maus als Zeit zwischen zwei Events auf: Mit `mouseDown` beginnt der Zeitraum, mit `mouseUp` (und `mouseUpOutside`, falls es sich um ein Sprite-Skript handelt) endet er. Die Skript-Lösung setzt daher eine Property-Variable (`mymousedown`) bei `mouseDown` auf `true`, bei den anderen Events auf `false`. Die Animation im `exitFrame`-handler wird nur ausgeführt, wenn `mymousedown = true` ist und – wie in den vorangehenden Beispielen – wenn die Animation noch nicht zu Ende ist. Bei einem erneuten Drücken der Maus wird die Animation wieder von vorn abgespielt, da wir `counter` im `mouseDown`-Handler auf 0 setzen.

```
property counter
property mymousedown

on beginSprite me
  mymousedown = false
end

on mouseDown me
  mymousedown = true
  counter = 0
end

on mouseUp me
  mymousedown = false
end

on mouseUpOutside me
  mymousedown = false
end

on exitFrame me
  if mymousedown = true then
    if counter < 12 then
      counter = counter + 1
      sprite(2).membernum = counter
    end if
  end if
end
```

Im Beispielfilm *MausGedruecktSprite.dir* haben wir ein weiteres Grashalm-Sprite in den dritten Drehbuchkanal eingefügt und darauf folgendes Behavior platziert, das fast die gleiche Auswirkung hat wie das oben stehende Skript:

```
property counter

on mousedown me
  counter = 0
end

on exitframe me
  if the mouseDown = true then
    if counter < 12 then
      counter = counter + 1
      sprite(3).membernum = counter
    end if
  end if
end
```

Die Systemeigenschaft `the mousedown` liefert dann den Wert `true`, wenn die Maus gedrückt ist. Wenn wir sie nutzen, so könnten die beiden Event-Handler für `mouseUp` und `mouseUpOutside` wegfallen, und auch die Eigenschaft `mymousedown` wäre überflüssig. Allerdings gibt es ein Problem mit diesem Skript: `the mousedown` ist wahr, auch wenn die Maus an einer beliebigen Stelle auf der Bühne gedrückt wird. Wenn tatsächlich auf unser Sprite geklickt wird, funktioniert das Skript also wie erwartet; wenn zu Beginn oder zu einem Zeitpunkt, zu dem `counter` kleiner als 12 ist, daneben geklickt wird, produziert das Skript ein Ergebnis, das wir nicht wollen: Dann nämlich wird die Animation abgespielt.

Dieses Verhalten lässt sich zwar durch weitere Maßnahmen in den Griff kriegen – wir könnten im `beginSprite`-Handler `counter` wieder auf 12 setzen, wir könnten die Systemeigenschaften `the clickOn` und `the rollover` nutzen – aber das wären nur unschöne Workarounds, die an unsere erste Lösung nicht heranreichen.

2.7 Experiment 6: Kontinuierlich animieren, wenn die Maus gedrückt ist

Eine kleine Erweiterung an unserem ersten, auf Events basierenden Skript aus Experiment 5 wollen wir noch vornehmen: Das Grashalm-Sprite soll kontinuierlich so lange animiert werden, wie wir die Maus drücken.

Dafür müssen wir lediglich die Property `counter` wieder auf 1 setzen, wenn Sie beim letzten Darsteller angekommen ist, also bei 12. Wir tun dies im `exitFrame`-Handler, der Rest des Skripts bleibt unverändert:

```
property counter
property mymousedown

on beginSprite me
  mymousedown = false
end

on mouseDown me
  mymousedown = true
  counter = 0
end

on mouseUp me
  mymousedown = false
end

on mouseUpOutside me
  mymousedown = false
end
```

```
on exitFrame me
  if mymousedown = true then
    if counter < 12 then
      counter = counter + 1
    else
      counter = 1
    end if
    sprite(2).membernum = counter
  end if
end
```

Den Director-Film finden Sie als *MausGedruecktSpriteWieder.dir* auf der CD-ROM 

2.8 Experiment 7: Doppelklick und Einfachklick erkennen

Im letzten Experiment zu Mausklicks soll ein markantes Klicksignal – der Doppelklick – eingesetzt werden. Um das Doppelklicken auf einem Element zu erkennen, reicht es aus, mit der Systemeigenschaft the doubleClick zu überprüfen, ob dieses Ereignis stattgefunden hat:

```
on mouseup me
  if the doubleClick = true then put "Doppelklick"
end
```

`the doubleClick` bleibt nach einem Doppelklick so lange `true`, bis ein neues Klick-Event auftritt. Sie sollten diese Systemeigenschaft also nur in einem `mouseUp`-Skript überprüfen.

Sowohl Einmal- als auch Doppelklicks zu erkennen und unterschiedlich zu verarbeiten, ist etwas komplizierter. Im Behavior „Einmal- und Doppelklick“ des Beispielfilms *Doppelklick.dir* wird ein Timer eingerichtet, der unmittelbar nach dem ersten Klick prüft, ob noch ein weiterer Klick folgt. Die Zeit wird in Director in Ticks oder in Millisekunden gemessen, wobei 1 Tick 1/60 Sekunde entspricht. 

Wir haben eine ganze Reihe weiterer Grashalm-Sprites auf die Bühne platziert, die aber mit folgendem kleinen Behavior zu Beginn unsichtbar geschaltet werden. Wundern Sie sich nicht über den Ausdruck `me.spriteNum`: Er bezeichnet die Spritenummer des Sprites, auf dem das Skript liegt, und wird uns im Verlauf der weiteren Experimente noch häufig begegnen.

```
on beginSprite me
 sprite(me.spriteNum).visible = false
end
```

Klick und Doppelklick auf Sprite 7 sollen nun jeweils die Sichtbarkeit verschiedener Sprites an- und ausschalten.

```
on mouseUp me
```

Wir reagieren auf das mouseUp*-Event. Da ein Doppelklick dieses Event zweimal produziert, wir aber unsere eigene Doppelklick-Erkennung haben, brechen wir das Skript dann ab, wenn es nach Doppelklick (zum zweiten Mal) aufgerufen wird:*

```
  if the doubleClick then exit
  startTimer
  repeat while the timer < 20
    if the mouseDown then
```

*Beim Klick läuft für einen Sekundenbruchteil (*the timer *zählt in Ticks) eine Schleife, die überprüft, ob ein erneutes* mouseDown *erfolgt. Falls das so ist, wird* doppelKlick *aufgerufen und die weitere Bearbeitung dieses Skripts abgebrochen, andernfalls wird nach Ende der Schleife* einmalKlick *ausgeführt.*

```
      doppelKlick
      exit
    end if
  end repeat
  einmalKlick
end

on doppelKlick
  sprite(2).visible = 1
  sprite(3).visible = 1
  sprite(4).visible = 1
  sprite(5).visible = 1
  sprite(6).visible = 0
end

on einmalKlick
  sprite(2).visible = 0
  sprite(3).visible = 0
  sprite(4).visible = 0
  sprite(5).visible = 0
  sprite(6).visible = 1
end
```

2.9 Diskussion

Wir haben in unseren Experimenten verschiedene Klickaktivitäten des Anwenders kennen gelernt und mit Lingo-Mitteln diese Aktivitäten unterscheiden gelernt. Dabei sind wir mit einem Skripttypus und zwei wichtigen Konzepten von Director in Berührung gekommen:

- Das „Verhalten“ (Behavior) ist der wichtigste Skripttypus in Director. Es kann in zwei Variationen auftreten: als Frameskript im Skriptkanal des Drehbuchs, und als Spriteskript auf Sprites, also auf Elementen, die Sie auf der Bühne angeordnet haben.
- Event-Handler bieten uns die Möglichkeit, unsere Skripte an Programm- oder Anwender-generierte „Ereignisse“ (Events) zu koppeln.
- Property-Variablen sind Datencontainer, die dazu dienen, Zustände in einem Behavior für alle Event-Handler verfügbar zu machen: Die Properties werden quasi zu „Eigenschaften“ des Sprites, auf dem das Skript liegt.

Variablen, Properties und Co.

Director kennt verschiedene Datencontainer: lokale Variablen, Property-Variablen und globale Variablen. Sie unterscheiden sich vor allem in ihrem Gültigkeitsbereich (Scope) und in der Art ihrer Deklaration.

Eine Variable, die irgendwo in einem Skript verwendet wird, indem ihr ein Wert zugewiesen wird, gilt nur innerhalb dieses Skripts und darin nur innerhalb des Event-Handlers oder der Funktion, die sie benutzt. In den Experimenten in diesem Kapitel haben wir keine solche lokale Variable benutzt.

Property-Variablen (wie `counter`, `geklickt` oder `mymousedown` in diesem Kapitel) deklarieren wir, üblicherweise am Anfang des Skripts, mit dem Schlüsselwort `property`. Diese „Eigenschaften“ sind dann Datencontainer, auf die wir von jedem Handler des Skripts aus zugreifen können, die wir also an jeder Stelle des Skripts lesen und verändern können. Neben Behaviors können auch Parent-Skripte Properties haben.

Tatsächlich ähneln Property-Variablen den (Director-eigenen) Eigenschaften von Sprites, Darstellern und anderer „Objekte“. Indem wir `counter` in unseren Skripten verändern, verändern wir eine (selbst definierte) Eigenschaft desjenigen Sprites, auf dem das Skript liegt. Ist der Abspielkopf der Director-Engine an einem Platz, wo das Sprite nicht mehr existiert, so gibt es auch die Property nicht mehr.

Ein weiterer Variablentyp ist die globale Variable: Sie wird mit dem Schlüsselwort `global` deklariert und kann in allen Skripten, die eine entsprechende Deklaration beinhalten, benutzt werden.

Globale Variablen sind also skript-, frame-, sprite- und auch filmübergreifende Datencontainer, mit denen man sparsam umgehen sollte, da sie nicht automatisch gelöscht werden.

Ganz unabhängig vom Variablentyp ist übrigens der Variableninhalt. In Director kann jede Variable jeden Datentyp aufnehmen, also beispielsweise Strings (Text), Integers oder Floats (Ganz- und Fließkommazahlen), Listen und eine Vielzahl weiterer komplexer Objekte wie Sprite-Referenzen oder Media von Darstellern.

Hinweis zur Animation

In den Experimenten 1 bis 6 dieses Kapitels haben wir durchgängig einen Weg zur Animation des Grashalms benutzt: Wir haben einen `counter` auf `exitFrame` hochgezählt und mit dem jeweiligen `counter`-Wert eine Bildänderung (Darstellerwechsel) bewirkt.

Damit haben wir die Animation an die Bildrate des Director-Films gekoppelt: Pro Framewechsel soll genau eine Bildänderung erfolgen.

Es gibt einen scheinbar einfacheren Weg, eine solche Animation zu programmieren: eine Programmschleife mit `repeat`.

```
on mouseDown
  repeat with counter = 1 to 12
    sprite(2).memberNum = counter
  end repeat
end mouseDown
```

`counter` wird hier von 1 bis 12 hochgezählt; die Darstelleränderung wird innerhalb der Schleife erledigt.

Wenn Sie dieses Skript ausprobieren, werden Sie nur das erste und das letzte Bild der Animation sehen, da Director während der Skriptausführung den Bildschirminhalt nicht neu zeichnet. Folgende Erweiterung zwingt die Abspielengine zu einem Neuzeichnen bei jedem Zählschritt:

```
on mouseDown
  repeat with counter = 1 to 12
    sprite(2).memberNum = counter
    updateStage
  end repeat
end mouseDown
```

Auch jetzt werden wir außer einem kurzen Zucken nicht wirklich etwas sehen – es sei denn, wir haben einen sehr langsamen Rechner. Tatsächlich ist die repeat-Schleife für Animationen denkbar ungeeignet, da die Ausführung dieser Schleifen immer mit der maximalen Geschwindigkeit erfolgt (und damit extrem rechnerabhängig ist). Bei längeren Vorgängen kommt hinzu, dass repeat die Applikation „lahm legt“: Kein anderes Skript, keine Mausabfrage o.Ä. kann parallel erfolgen.

In den Experimenten ist der Darstellertausch deshalb an den Framewechsel gebunden. Die Geschwindigkeit der Animation ist damit von der Geschwindigkeitseinstellung des Director-Films abhängig, die Sie im Steuerpult (vgl. Abbildung 2.17) oder im Tempokanal des Drehbuchs beeinflussen können.

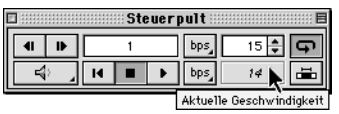


Abbildung 2.17: Die Tempo-Einstellung Ihres Films nehmen Sie im Steuerpult vor (Strg/⌘+2).

Die Ziffer im Feld des Steuerpults rechts unten gibt den tatsächlich erreichten Wert an, der niedriger als der Zielwert sein kann, wenn Director die Zielvorgabe nicht erreicht. Das kann zum Beispiel bei einer rechenaufwändigen Animation oder bei komplizierten Lingo-Skripten, die bei jedem Framedurchlauf ausgeführt werden, der Fall sein.

Und weiter ...

Die Maus und die Reaktion auf Mausaktivitäten stehen auch in den folgenden Kapiteln im Mittelpunkt. In Kapitel 7 geht es noch einmal intensiv um die Director-Events.

Wahrheit und Wiederholung: grundlegende Lingo-Strukturen

Bestimmte Strukturen sind so grundlegend für jede Art von „Programm“, dass sie in jeder Programmierlogik existieren. Stellen Sie sich eine Waschmaschine vor, die keine Zustandswerte überprüfen könnte („Schließ die Tür, *wenn das Programm startet*“, „Heize, *bis das Wasser 40 Grad warm ist*“, usw.) und keine Wiederholungen kennt (wie „Schleudere *17-mal* in die eine und *17-mal* in die andere Richtung“).

Auch in Director wollen wir immer wieder Bedingungen stellen (die Ausführung von Programmteilen von Zuständen anderer Teile unserer Anwendung abhängig machen), und wir werden Programmteile mehrfach durchlaufen wollen, schon weil es ökonomischer ist, als 10-mal denselben Code zu schreiben.

In diesem Kapitel wollen wir uns mit diesen Grundgegebenheiten des Programmierens auseinander setzen.

3.1 Experiment 1: Ein und Aus

Die einfachste Zustandsbeschreibung ist „Ein oder Aus“, „Ja oder Nein“, „1 oder 0“, „Schwarz oder Weiß“. Die Schalterpositionen eines Lichtschalters oder die Färbung eines Schwarz-Weiß-Bildes an einem bestimmten Bildpunkt sind Eigenschaften, die diese Zustandswerte annehmen können.

Auch im Computer werden alle Informationen in dieser binären Form gespeichert und verarbeitet. Ungeachtet der Komplexität, die damit erreichbar ist: Zahlreiche Zustände sind tatsächlich mit einfachen Ja/Nein-Fragen beantwortbar. Ist der Ton eingeschaltet (`the soundEnabled`)? Ist Quicktime zum Abspielen von Videos vorhanden (`the quicktimePresent`)? Spielt der Sound in Kanal 1 noch (`soundbusy(1)` oder `sound(1).isBusy()`)? Ist eine Netzwerkoperation abgeschlossen (`netDone(netID)`)?

Wenn Sie solche Eigenschaften abfragen, erhalten Sie als Ergebnis stets `1` (bzw. `TRUE`) als positive oder `0` (bzw. `FALSE`) als negative Antwort. Das können Sie im Nachrichtenfenster ausprobieren (Strg / ⌘+M):

```
put the soundEnabled
-- 1
put the quicktimePresent
-- 1
put soundbusy(1)
-- 0
netid = getNetText("http://www.lingolabor.de")
put netDone(netID)
-- 0
put netDone(netID)
-- 1
```

Wir schließen: Der Rechner, auf dem dies ausprobiert wurde, hat Sound und kann Quicktime-Videos abspielen. Momentan läuft kein Sound in Kanal 1. Eine Netzwerkaktion wurde gestartet und war bei der ersten Abfrage noch nicht abgeschlossen, bei der zweiten dann schon. Weiter geht's:

```
put voidP(netzString)
-- 1
netzString = netTextresult(netid)
put voidP(netzString)
-- 0
put netzString
-- "<HTML><HEAD><TITLE>Lingolabor-Homepage</TITLE> (...)"
```

Auch hier fragen wir wieder nach einem Zustand: Ist die Variable `netzString` undefiniert, also `VOID`? Da wir sie tatsächlich nicht definiert hatten, ist dies der Fall, folglich ist die Antwort „Ja“, `1`. Wenn wir die Variable dann definieren, indem wir ihr den Rückgabewert der eben gestarteten

Netzwerkaktion zuweisen (`netzstring = netTextresult(netid)`), liefert die Überprüfung von `voidP(netzString)` das Ergebnis `0` – die Variable `netzString` ist also nicht mehr undefiniert. Zur Kontrolle geben wir `netzString` aus, und die Variable enthält tatsächlich Text.

3.2 Experiment 2: Wahr oder Falsch

Allerdings ist nicht jede Information so einfach gestrickt, dass sie durch „Ja“ oder „Nein“ ausgedrückt werden kann. Vielmehr gibt es hunderte von Eigenschaften in Director, die komplexere Datentypen zurückliefern.

Versuchsaufbau

Legen Sie in einem leeren Film ein Sprite an (zum Beispiel mit dem Oval-Werkzeug in der Werkzeugpalette), und positionieren Sie den Abspielkopf im Drehbuch beispielsweise auf Frame 17. Nun können wir einige Eigenschaften abfragen, die nicht mit „Ja“ und „Nein“ beantwortet werden können.

Durchführung

Wir testen weiterhin einige Eigenschaften im Nachrichtenfenster. `the frame` gibt Ihnen beispielsweise den Frame an, in dem der Abspielkopf gerade steht; es wird eine Integer-Zahl zurückgeliefert. Und wenn Sie nach dem Darsteller des gerade angelegten Sprites fragen, erhalten Sie eine Referenz – einen „Zeiger“ – auf einen Darsteller in der Besetzung:

```
put the frame
-- 17

put sprite(1).member
-- (member 1 of castLib 1)
```

Färben Sie das eben angelegte Sprite einmal rot ein, indem Sie es auf der Bühne auswählen und die Farbe mit dem Farbwähler der Werkzeugpalette zuweisen. Auch diese Eigenschaft können wir in Lingo überprüfen:

```
put sprite(1).color
-- rgb( 255, 0, 0 )

put sprite(1).color.red
-- 255
```

Hier wird ein Farbobjekt (`rgb(255,0,0)`) bzw. ein Integer (der Rotanteil der Farbe) zurückgeliefert. In Lingo werden Sie es mit einer ganzen Reihe weiterer Datentypen zu tun haben, die wir im Laufe der „Laborarbeit“ erwähnen werden.

Auch in diesen Fällen interessieren uns aber häufig Aussagen, die eine eindeutige „Ja/Nein“-Antwort erlauben:

Ist der Abspielkopf schon hinter Frame 25? Wurde dem Sprite tatsächlich eine Farbe zugewiesen, die Rot enthält? Wir formulieren also Aussagen, die wieder „wahr“ oder „falsch“ sein können.

```
put (the frame > 25)
-- 0
put (the frame = 0)
-- 0
put (the frame = 17)
-- 1
put (sprite(1).color = rgb(255,0,0))
-- 1
put (sprite(1).color.red > 128)
-- 1
put (sprite(1).color.green + sprite(1).color.blue < 128)
-- 1
put (sprite(1).member.name <> "")
-- 0
```

Mit den Vergleichsoperatoren „kleiner als“ (<), „größer als“ (>), „kleiner oder gleich“ (<=), „größer oder gleich“ (>=), „gleich“ (=) und „ungleich“ (<>) können Sie die tatsächlichen Werte mit Ihren Grenz- oder Zielwerten vergleichen. Die Klammern, die wir in den obigen Beispielen verwendet haben, sind nicht notwendig; sie heben aber den Teil des Lingo-Ausdrucks, der eine „Wahrheitsaussage“ enthält, hervor.

Eine Anmerkung noch für Lingo Laboranten, die andere Skript- oder Programmiersprachen kennen: In Lingo sind der Zuweisungsoperator (=) und der Vergleichsoperator (=) identisch, was für Umsteiger gelegentlich zu Verwirrung führt:

```
myname = "Hans"
put (myname = "Hans")
-- 1
```

Die erste Zeile weist der Variablen `myname` den Wert `"Hans"` zu. Die zweite überprüft den Wahrheitswert der Aussage „Die Variable `myname` hat den Wert `"Hans"`“.

3.3 Experiment 3: Bedingungen

Selbstverständlich ist uns nicht geholfen, wenn wir Wahrheitswerte nur ins Nachrichtenfenster ausgeben. Hier wollen wir nun Lingo-Code von Bedingungen abhängig machen.

Versuchsaufbau

Wir benutzen den Film aus Experiment 2 weiter und legen mit der Tastenkombination Strg / ⌘+⇧+U ein neues Filmskript an. Beim Start des Films wollen wir eine Reihe von Bedingungen überprüfen und eventuell eine Warnmeldung ausgeben. Der erste Handler, der beim Start des Films ausgeführt wird, ist `on prepareMovie`, sodass wir als Grundgerüst schon Folgendes eingeben können:

```
on prepareMovie

end
```

Durchführung

Die Bedingungen, auf die wir nun unser Abspielsystem prüfen wollen, sind folgende:

- Ist die Grafikausgabe auf „True Color“ (mindestens 32 Bit Farbtiefe) eingestellt?
- Ist Quicktime vorhanden?
- Ist Quicktime in Version 6 vorhanden?
- Ist der Ton eingeschaltet?
- Ist die Systemlautstärke 3 oder höher (auf einer Skala von 0 bis 7)?

Wenn mindestens eine der Bedingungen nicht zutrifft, wollen wir eine Dialogbox anzeigen, die eine entsprechende Warnung enthält. Die Aussage, die Farbtiefe ist kleiner als 32 Bits, lässt sich beispielsweise so übersetzen:

```
(the colordepth < 32)
```

Um abhängig vom Wahrheitswert dieser Aussage einen Lingo-Befehl auszuführen, können wir mit `if … then … end if` ein Bedingungskonstrukt nutzen:

```
if (AUSSAGE) then
   BEFEHLE
end if

if (the colordepth < 32) then
   -- ...
end if
```

Befehle, die innerhalb dieser „Klammer“ geschrieben sind, werden nur ausgeführt, wenn die Aussage wahr (TRUE) ist, die Bedingung also zutrifft.

Bestimmte Aussagen benötigen wir in negativer Form: Wir wollen beispielsweise nicht (the quicktimePresent) als Bedingung nutzen (wenn diese Aussage wahr ist, gibt es vielmehr keine Meldung), sondern die Negation: not(the quicktimePresent). Das Schlüsselwort not() kehrt Wahrheitswerte um: TRUE wird FALSE, FALSE wird TRUE.

Wenn die jeweiligen Bedingungen zutreffen, wollen wir jeweils spezifische Meldungszeilen erzeugen und die lokale Variable meldungzeigen auf TRUE setzen:

```
on prepareMovie
  meldungzeigen = 0
    Die Variable meldungzeigen wird auf FALSE gesetzt. Sie wird
    unten überprüft.
  if (the colorDepth < 32) then
    colormeldung = "Bitte stellen Sie eine größere Farbtiefe
      ein." & Return
    meldungzeigen = 1
  end if
  if not(the quicktimePresent) then
    qtmeldung = "Bitte installieren Sie Quicktime." & Return
    meldungzeigen = 1
  end if
  if (quicktimeVersion() < 6) then
    qtmeldung = "Bitte installieren Sie Quicktime 6." & Return
    meldungzeigen = 1
  end if
  if not (the soundEnabled) then
    soundmeldung = "Bitte aktivieren Sie den Ton." & Return
    meldungzeigen = 1
  end if
  if (the soundLevel < 3) then
    soundmeldung = "Bitte stellen Sie die Lautstärke lauter."
    & Return
    meldungzeigen = 1
  end if
    Wenn meldungzeigen auf TRUE (1) gesetzt wurde, dann zeige
    nun die Meldung:
  if (meldungzeigen) then
    meldung = colormeldung & qtmeldung & soundmeldung
    Die Meldung wird aus den einzelnen Meldungszeilen zusammen-
    gesetzt.
    alert meldung
    Der Befehl alert erzeugt eine einfache Dialogbox mit dem
    Meldungstext.
  end if
```

```
  if (the runmode = "Author") then
    Im Authoringmodus soll die Meldung zusätzlich ins Nachrich-
    tenfenster geschrieben werden.
    authormeldung = "Sie spielen den Film im Authoring-Modus ab."
    meldung = colormeldung & qtmeldung & soundmeldung
      & authormeldung
    put meldung
  end if
end
```

Sie finden den Film *bedingung.dir* auf der CD-ROM. Spielen Sie den Film ab. Sie erhalten zu Beginn eine Meldung, die wie in Abbildung 3.1 aussehen wird, und im Nachrichtenfenster wird eine entsprechende Meldung ausgegeben:

```
-- "Bitte installieren Sie eine neuere Version von Quicktime.
Bitte stellen Sie die Lautstärke lauter.
Sie spielen den Film im Authoring-Modus ab."
```

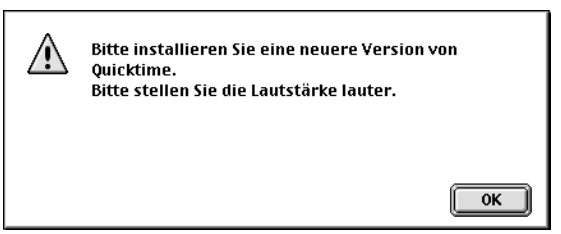


Abbildung 3.1: Selbst erzeugte Warnmeldung

Weitere Möglichkeiten, Bedingungen auszudrücken

Die Grundstruktur `if … then … end if`, die wir gerade verwendet haben, lässt sich noch modifizieren und erweitern.

- Die Grundform:
  Die Lingo-Struktur wird stets über mehrere Zeilen geschrieben, wobei `if (Aussage) then` die Klammer eröffnet und `end if` sie abschließt. Innerhalb der Klammer kann eine beliebige Anzahl Lingo-Befehle stehen:

  ```
  if (Aussage) then
    Befehl1
    Befehl2
    (...)
  end if
  ```

- Die Kurzschreibweise:
  Wollen Sie nur einen einzigen Lingo-Befehl von einer Bedingung abhängig machen, so können Sie die einzeilige Schreibweise ohne abschließendes `end if` benutzen:

```
if (Aussage) then Befehl
```

Werden mehrere Bedingungen ineinander verschachtelt, so ist diese Schreibweise evtl. nicht möglich. Näheres im nächsten Abschnitt.

- Die Erweiterung mit `else`:
  Wollen Sie einen Lingo-Befehl ausführen, wenn die überprüfte Aussage „wahr“ ist, und einen anderen Lingo-Befehl, wenn sie „falsch“ ist, so können Sie folgende Konstruktionen verwenden:

```
if (Aussage) then
  Befehl1
else
  Befehl2
end if

if (Aussage) then Befehl1
else Befehl2
```

  Die erste Variante erlaubt selbstverständlich wieder beliebig viele Lingo-Befehle innerhalb der beiden „Klammern“.
- Die Erweiterung mit `else if`:
  Die folgende Variante erlaubt es, mehrere Bedingungen in Folge zu untersuchen. Wenn eine Bedingung zutrifft, dann werden die nachfolgenden Überprüfungen nicht mehr vorgenommen. Folglich ist es wichtig, in welcher Reihenfolge Sie die Wahrheitsaussagen überprüfen:

```
if (Aussage1) then
  Befehl1
else if (Aussage2) then
  Befehl2
else if (Aussage3) then
  Befehl3
else
  Befehl4
end if
```

  Wenn beispielsweise `Aussage1` nicht zutrifft, aber `Aussage2` , dann wird `Befehl2` ausgeführt. Wenn keine der Aussagen 1 bis 3 zutrifft, so wird `Befehl4` in der abschließenden `else`-Klammer ausgeführt.
- Die Alternative `case`:
  Das letzte Beispiel lässt sich mit anderen Lingo-Mitteln so nachbilden, dass genau dasselbe Ergebnis erreicht wird:

```
case true of
    Aussage1: Befehl1
    Aussage2: Befehl2
    Aussage3: Befehl3
  otherwise
    Befehl4
end case
```

Auch hier ist wichtig, in welcher Reihenfolge die Aussagen aufgeführt werden: `case` bricht bei der ersten zutreffenden Aussage ab. Sie können immer auch mehrere Lingo-Befehle ausführen lassen:

```
case true of
    Aussage1:
      Befehl1a
      Befehl1b
      (...)
    Aussage2: Befehl2
end case
```

Auch wenn wir hier mit `case` nur den Wahrheitswert von Aussagen überprüfen, ist das `case`-Konstrukt eigentlich universeller: Es erlaubt, beliebige „Identitäten“ festzustellen, indem ein Testwert mit einer Reihe anderer Werte verglichen wird:

```
case Testwert of
    Vergleichswert1, Vergleichwert2: Befehl1
    Vergleichswert3: Befehl2
  otherwise: Befehl3
end case
```

Oder in einem echten Code-Beispiel:

```
on prepareMovie
  case (the colorDepth) of
      1, 2, 4, 8: nachricht = "Ihr Monitor
      stellt 256 oder weniger Farben dar."
      16: nachricht = "Ihr Monitor stellt tausende Farben dar."
      24, 32: nachricht = "Ihr Monitor
      stellt Millionen Farben dar."
  end case
  put nachricht
end
```

Mit `if` lässt sich diese Überprüfung selbstverständlich auch ausdrücken, allerdings würde man die „ökonomische“ Variante wählen und die Wahrheitsaussagen etwas anders fassen:

```
if (the colorDepth < 16) then
  nachricht = "Ihr Monitor stellt 256 oder weniger Farben dar."
else if (the colorDepth > 16) then
  nachricht = "Ihr Monitor stellt Millionen Farben dar."
else
  nachricht = "Ihr Monitor stellt tausende Farben dar."
end if
put nachricht
```

Bedingungen kombinieren: Schachtelung sowie AND und OR

Wenn mehrere Bedingungen zutreffen sollen, bevor Ihr Lingo-Skript ausgeführt wird, so haben Sie die Möglichkeit, mehrere `if`- (oder auch `case`-) Statements ineinander zu verschachteln:

```
if (Aussage1) then
  if (Aussage2) then
    Befehl1
  end if
end if
```

Hier wird Befehl1 nur ausgeführt, wenn Aussage1 und Aussage2 wahr sind. Das entspricht einer logischen Verknüpfung mit `AND`:

```
if (Aussage1 AND Aussage2) then Befehl1
```

Die Gesamtaussage `Aussage1 AND Aussage2` ist genau dann „wahr“, wenn beide Teilaussagen „wahr“ sind. Sonst ist die Gesamtaussage „falsch“:

Aussage1	Aussage2	Aussage1 AND Aussage2
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
FALSE	FALSE	FALSE

Werden Aussagen dagegen mit `OR` verbunden, so ist die Gesamtaussage dann wahr, wenn mindestens eine Teilaussage wahr ist:

Aussage1	Aussage2	Aussage1 OR Aussage2
TRUE	TRUE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE

Unser `case`-Beispiel von eben könnten wir also auch folgendermaßen übersetzen:

```
case (the colorDepth) of
    24, 32: nachricht = "Ihr Monitor stellt
    Millionen Farben dar."
end case
```

wird zu:

```
if ((the colorDepth = 24) OR (the colorDepth = 32)) then
  nachricht = "Ihr Monitor stellt Millionen Farben dar."
end if
```

AND und OR sowie das schon erwähnte NOT() sind das Handwerkzeug, das wir nutzen können, um Wahrheitswerte beliebiger Aussagen so zu verknüpfen, wie wir es für unsere Programmlogik benötigen. NOT() verknüpft dabei nicht zwei Aussagen, sondern negiert den Wahrheitswert *einer* Aussage:

Aussage1	NOT (Aussage1)
TRUE	FALSE
FALSE	TRUE

3.4 Experiment 4: Booles Schalter

In diesem Experiment wollen wir einige Zustandsänderungen auf der Bühne ermöglichen (ein Schalter wird eingeschaltet, ein Sound gestartet). Sind beide Aktionen erfolgt, so soll eine weitere Aktion „freigeschaltet" werden: Eine Grafik soll dann frei auf der Bühne bewegt werden können.

Hierbei helfen uns die Lingo-Mittel, die wir in diesem Kapitel kennen gelernt haben:

```
IF (BEDINGUNG1 AND BEDINGUNG2) THEN
  ERLAUBE_AKTION_3
END IF
```

Diese Verfahren, mit Wahrheitswerten und Bedingungen umzugehen, gehen auf den englischen Mathematiker George Boole zurück. Mit seinem Werk der „Algebra der Logik" schuf er 1854 die Grundlagen digitaler Rechen- und Steuerschaltungen.

Versuchsaufbau

Im Film *Boole.dir* platzieren wir eine Reihe von Grafik-Sprites: eine Hintergrundgrafik (Sprite 1), den „Aus"-Zustand eines Schalters (Sprite 2), eine ausgeschaltete Lampe (Sprite 3), ein Radio (Sprite 4) und ein Abbild von Herrn Boole.

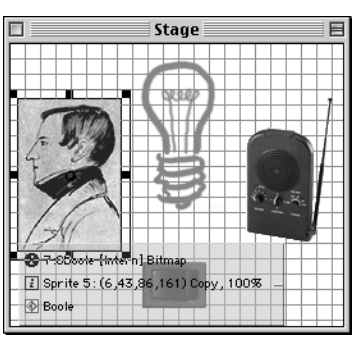


Abbildung 3.2: Die Bühne unseres Beispielfilms Boole.dir

Ein Klick auf den Schalter soll ihn umlegen und das Licht einschalten, der Klick auf das Radio einen Sound-Darsteller („Boole“) abspielen. Erstellen Sie die nötigen Skripte auf den entsprechenden Sprites.

Eine mögliche Umsetzung ist folgende: auf Sprite 4:

```
on mouseUp me
  puppetSound 1, member("Boole")
end
```

Auf Sprite 2, dem Schalter:

```
on mouseUp me
  sprite(me.spriteNum).member = member("SchalterAn")
  sprite(3).member = member("LampeAn")
end
```

Durchführung

Wenn Sie das Skript auf Sprite 2 ausprobiert haben, haben Sie sicherlich bemerkt, dass es die Lampe ein-, aber nicht wieder ausschaltet.

Eine Erweiterung ist nötig, die abhängig vom aktuellen Zustand des Schalters und der Lampe unterschiedliche Bildwechsel herbeiführt:

```
property eingeschaltet

on mouseUp me
  if (not(eingeschaltet)) then
    sprite(me.spriteNum).member = member("SchalterAn")
    sprite(3).member = member("LampeAn")
    eingeschaltet = TRUE
  else
    sprite(me.spriteNum).member = member("SchalterAus")
    sprite(3).member = member("LampeAus")
    eingeschaltet = FALSE
  end if
end
```

Für die Lebensdauer des Sprites hält die Property-Variable `eingeschaltet` den momentanen Zustand des Schalters. Ist `eingeschaltet` nicht definiert oder `FALSE`, so wird der erste Teil der `if … then … else … end if`-Klammer ausgeführt; hat eingeschaltet den Wert `TRUE`, so wird der zweite Teil genutzt.

Gleichzeitig haben wir eine Property-Variable definiert, die wir auch von anderen Skripten aus abfragen können – allerdings nur, wenn der Film läuft, da Sprite-Properties erst initialisiert werden, wenn der Abspielkopf auf das Sprite trifft.

```
put sprite(2).eingeschaltet
-- 0
```

ist ein Beispiel für die Abfrage der Property-Variablen `eingeschaltet`.

Auch das Skript auf Sprite 4 wollen wir durch eine solche Variable erweitern:

```
property TonGespielt

on mouseUp me
  puppetSound 1, member("Boole")
  TonGespielt = TRUE
end
```

Nun haben wir zwei Eigenschaften, die wir in unserem Hauptskript auf dem Sprite 5, dem Bild von Boole, nutzen können.

```
property myPos

on beginSprite me
  sprite(me.spriteNum).blend = 50
  mypos = sprite(me.spriteNum).loc
end

on exitFrame me
  test = (sprite(4).Tongespielt AND sprite(2).eingeschaltet)
  if test then
    sprite(me.spritenum).moveableSprite = 1
    sprite(me.spriteNum).blend = 100
  else
    sprite(me.spritenum).moveableSprite = 0
    sprite(me.spriteNum).blend = 50
    sprite(me.spriteNum).loc = mypos
  end if
end
```

Im `beginSprite`-Handler wird das Sprite auf 50% Transparenz eingestellt; außerdem merken wir uns in der Property-Variablen `mypos` die aktuelle Position (das ist hier die Position, die das Sprite im Drehbuch bzw. auf der Bühne hat).

Im `exitFrame`-Handler wollen wir regelmäßig überprüfen, ob die beiden Eigenschaften `sprite(4).Tongespielt` und `sprite(2).eingeschaltet` „wahr“ sind. In `test` merken wir uns den Wahrheitswert, der sich aus der `AND`-Verknüpfung der beiden Wahrheitswerte ergibt.

Wenn `test` „wahr“ ist, wird das Sprite beweglich (`moveableSprite`) gemacht und auf 100% Opazität eingestellt; falls `test` nicht „wahr“ ist, wird die Beweglichkeit wieder aufgehoben, die Opazität auf 50% reduziert und das Sprite auf die Ausgangsposition (`mypos`) gesetzt.

Wir können das Sprite also auf der Bühne herumziehen, sobald (und solange) der Lichtschalter eingeschaltet ist und wenn das Radio einmal angeklickt wurde. Durch die Änderung der Eigenschaft `blend` des Sprites wird uns optisch signalisiert, dass sich mit Herrn Boole etwas getan hat.

Wollen Sie das Skript so modifizieren, dass nicht das einmalige Anklicken des Radios, sondern der laufende Sound als Bedingung genommen wird, so müssen Sie nur die Zeile `test = …` modifizieren.

```
test = (soundbusy(1) AND sprite(2).eingeschaltet)
```

Hier ist `test` nur dann „wahr“, wenn in Kanal 1 ein Sound läuft (nämlich unser Sound „Boole“) UND wenn die Lampe eingeschaltet ist.

3.5 Experiment 5: Der Reiz der Wiederholung

Manch einen Skriptbefehl wollen wir mehrfach ausführen, zum Beispiel, wenn wir dieselbe Änderung für mehrere Sprites bewirken wollen. Director bietet mit `repeat` eine Kontrollstruktur, die solche Wiederholungen in Lingo implementiert.

Die Grundstruktur dieses so genannten `repeat`-Loops ist folgende:

```
repeat (Zähler oder Auswahl oder Abbruchbedingung)
 Befehle
end repeat
```

Die Befehle, die zwischen `repeat` und `end repeat` stehen, werden so oft ausgeführt, wie es der zur `repeat`-Syntax gehörige Zähler oder die Auswahl oder die Abbruchbedingung erlauben. Schreiben Sie die folgenden Beispiele in ein Filmskript in einen `startMovie`-Handler; mehrzeilige Konstrukte wie `repeat`-Loops lassen sich im Nachrichtenfenster nicht sinnvoll testen.

- Mit Zähler: Sie können eine beliebig benannte Zählervariable zwischen Startwert und Endwert hochzählen lassen:

  ```
  on startMovie
    repeat with meinZaehler = 5 to 27
      put meinZaehler
    end repeat
  end
  ```

  Wenn Sie den Film mit diesem Skript starten, so wird die Variable `meinZaehler` von 5 bis 27 hochgezählt (und die entsprechenden Werte werden mit `put` ins Nachrichtenfenster geschrieben).
  Dasselbe rückwärts bewirkt `downto`:

  ```
  on startMovie
    repeat with meinZaehler = 27 downto 5
      put meinZaehler
    end repeat
  end
  ```

Statt eines langen Variablennamens werden häufig die Buchstaben `i`, `j`, `k` etc. als Zählervariablen verwendet, so dass Sie auch im Lingolabor häufig solche `repeat`-Loops sehen werden:

```
on startMovie
  repeat with i = 5 to 27
    put i
    -- bzw. Skripte, die die Zählervariable i benutzen
  end repeat
end
```

- Mit Auswahl-Elementen als Liste: In die Zähler-Variable werden nacheinander alle Elemente einer linearen Liste eingefügt:

```
on startMovie
  meineListe = [5,7,19,21,27]
  repeat with meinElement in meineListe
    put meinElement
  end repeat
end
```

- Mit Abbruchbedingung: Der Loop wird ausgeführt, solange die angegebene Bedingung gültig ist:

```
on startMovie
  counter = 0
  repeat while (counter < 5)
    put counter
    counter = counter + 1
  end repeat
end
```

Dies ist die gefährliche Variante des `repeat`-Loops, da Sie selbst dafür sorgen müssen, dass die Bedingung irgendwann „falsch“ wird – sonst wird sich Director in einer endlosen Schleife aufhängen.

In diesem Versuch wollen wir ein Einsatzbeispiel für einen `repeat`-Loop ausprobieren und eine Alternative zeigen, die `repeat`-Loops in den allermeisten Fällen unnötig macht.

Versuchsaufbau

In der Besetzung des Filmes *Fensterframe.dir* haben wir zwei Grafikdarsteller, die ein geöffnetes und ein geschlossenes Fenster symbolisieren (die Darsteller „fensterOffen“ und „fensterZu“). Wir platzieren das geschlossene Fenster vier Mal auf die Bühne (Sprites 1 bis 4).

Ein Frame-Skript im Skript-Kanal soll zunächst nur das obligatorische `go to the frame` in einem `exitFrame`-Handler enthalten.

Bei Klick auf eines der Fenster soll das angeklickte Fenster geöffnet werden, und falls bereits ein anderes Fenster offen ist, dieses geschlossen werden.

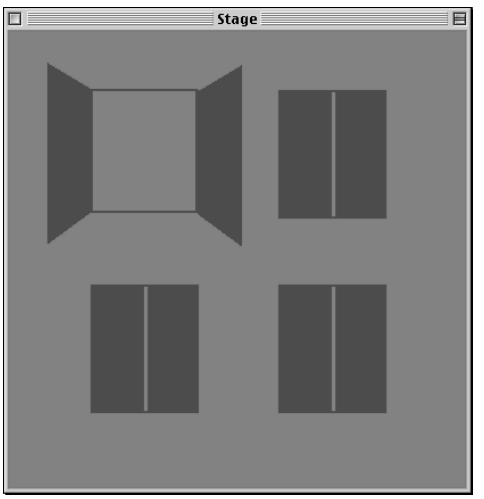


Abbildung 3.3: FensterRepeat.dir in Aktion

Wir versuchen unterschiedliche Umsetzungen, zunächst als Frame-Skript, dann als Sprite-Behavior.

Durchführung

Das folgende Frame-Skript kommt dem Ziel schon relativ nahe, und das mit sehr einfachen Mitteln:

Wir fragen im `exitFrame`-Handler des Frame-Skripts, ob die Maus gedrückt ist. Falls dies der Fall ist (also `the mouseDown` den Wert `TRUE` hat), dann überprüfen wir für die Sprites 1 bis 4, ob sich die Maus über dem betreffenden Sprite befindet (`if (rollover(i)) then …`). Ist dies der Fall, so wird der Darsteller des Sprites auf den Darsteller „FensterOffen“ gesetzt. Falls nicht, wird er wieder auf „FensterZu“ zurückgetauscht.

Der Clou dieses Skriptes ist demnach, dass es nur ausgeführt wird, wenn die Maus gedrückt ist. Dies überprüfen wir bei jedem Framedurchgang, also 15- oder 30-mal pro Sekunde, je nach Tempoeinstellung unseres Films. Die `repeat`-Schleife schließt dann einfach alle Fenster, nur das, über dem sich die Maus befindet, bleibt offen. Hier ist das Skript:

```
on exitFrame me
  if (the mouseDown) then
    repeat with i = 1 to 4
      if (rollover(i)) then
        sprite(i).member = member("FensterOffen")
      else
        sprite(i).member = member("FensterZu")
      end if
    end repeat
  end if
  go to the frame
end
```

Das Skript ist zwar sehr kurz und durch die Verwendung des `repeat`-Loops „effektiv“ gecodet, aber es hat einige gravierende Nachteile:

- Bei niedriger Bildrate wird nicht jeder Klick auf eines der Fenster erkannt – man muss die Maus einen Moment gedrückt *halten*, bis die `mouseDown`-Erkennung funktioniert.
- Das Skript nutzt den falschen Event: Statt auf ein `mouseDown` oder `mouseUp` zu reagieren, überprüfen wir (ineffektiverweise) bei jedem `exitFrame` den Zustand der Maus.
- Wir können mit diesem Konstrukt keine sinnvolle Lösung implementieren, mit der ein zweiter Klick auf ein geöffnetes Fenster dieses wieder schließt.

Deshalb versuchen wir es ein zweites Mal: Dieses Mal wollen wir den `mouseUp`-Event nutzen und ein Behavior erstellen, das auf alle Sprites platziert wird. Auf der CD finden Sie diesen Ansatz im Film *Fenster.dir*.

Überlegen wir uns zunächst, was das Öffnen und das Schließen des Fensters eigentlich bedeutet:

```
property istOffen

on oeffne me
  sprite(me.spriteNum).member = member("FensterOffen")
  istOffen = true
end

on schliess me
  sprite(me.spriteNum).member = member("FensterZu")
  istOffen = false
end
```

Diese beiden Handler des Behaviors bewirken alleine noch gar nichts; sie beinhalten aber die Lingo-Befehle, die wir benötigen, um die „FensterOffen“-Grafik bzw. die „FensterZu“-Grafik anzuzeigen und uns den Zustand des aktuellen Sprites in der Property `istOffen` zu merken.

Beachten Sie die Verwendung von `me.spriteNum`: Da wir das Behavior auf alle vier Sprites platzieren wollen, müssen wir die Spritenummer „neutral“ formulieren. In der Eigenschaft `me.spriteNum` ist die Spritenummer desjenigen Sprites enthalten, auf dem das Behavior liegt. Für ein Skript auf Sprite 3 ist `me.spriteNum` folglich 3.

Damit haben wir die Hauptfunktionalität des Behaviors schon definiert. Was noch fehlt, ist, diese Handler in einem (oder mehreren) Event-Handlern aufzurufen. Zum Beispiel so:

```
property istOffen

on beginSprite me
  schliess me
end
```

```
on mouseUp me
  if istOffen then
    schliess me
  else
    repeat with i = 1 to 4
      sprite(i).schliess()
    end repeat
    oeffne me
  end if
end

on schliess me
  sprite(me.spriteNum).member = member("Fensterzu")
  istOffen = false
end

on oeffne me
  sprite(me.spriteNum).member = member("FensterOffen")
  istOffen = true
end
```

Im `beginSprite`-Handler werden alle Fenster geschlossen: Das heißt, jedes Sprite führt, wenn das Sprite aktiviert wird, den `schliess`-Handler aus.

Der `mouseUp`-Handler des angeklickten Sprites muss nun eine ganze Reihe von Dingen erledigen: War das Fenster offen, so schließt er es; andernfalls schließt er zunächst alle Fenster (`repeat`-Schleife um den Aufruf `sprite(i).schliess()`) und öffnet dann das „eigene“, sprich, er ruft den eigenen `oeffne`-Handler auf, in dem die eigene Grafik auf den Darsteller „FensterOffen“ gesetzt wird.

Eleganter geht das Schließen der Fenster, wenn man

```
repeat with i = 1 to 4
  sprite(i).schliess()
end repeat
```

durch

```
sendAllSprites(#schliess)
```

ersetzt. Diese Director-Funktion sendet ein `schliess`-Event an alle Sprites; hat ein Sprite-Behavior einen entsprechenden Handler, so führt es ihn aus. `sendAllSprites()` erspart uns also den `repeat`-Loop (und auch die genaue Adressierung des `schliess`-Befehls: Verschieben wir die Sprites in andere Drehbuchkanäle, so wäre der `repeat`-Loop anzupassen, `sendAllSprites()` funktioniert weiterhin).

Hier nochmals das kommentierte Skript. Bitte platzieren Sie es auf *alle* vier Fenster-Sprites.

```
property istOffen

    istOffen wird gesetzt in oeffne() und schliess() und dient
    im mouseUp-Handler zur Unterscheidung, ob das Fenster zu
    öffnen oder zu schließen ist.

on beginSprite me
  schliess me
end

    Der beginSprite-Handler schließt alle Fenster und stellt
    so einen definierten Ausgangszustand her. istOffen ist
    danach FALSE.

on mouseUp me
  if istOffen then
    schliess me

    Wenn das Fenster, auf das geklickt wurde, offen ist,
    schließ es.

  else
    sendAllSprites(#schliess)
    oeffne me

    Sonst schließ alle und öffne mich.

  end if
end

on schliess me
  sprite(me.spriteNum).member = member("Fensterzu")
  istOffen = false
end

on oeffne me
  sprite(me.spriteNum).member = member("FensterOffen")
  istOffen = true
end
```

Diskussion

Die Kontrollstrukturen für Bedingungen (`if … then` und `case`) und für Wiederholungen (`repeat`) sind Lingo-Elemente, die wir immer wieder benötigen werden. Während es für die Überprüfung von Wahrheitswerten mit `if` oder `case` kaum eine Alternative gibt, lohnt sich bei `repeat` allerdings häufig ein kurzes Nachdenken. Director bietet eine ganze Reihe von Konstruktionen, die die Wiederholung von Code-Teilen ermöglichen. Dazu zählen:

- die hier vorgestellte Lösung mit `sendAllSprites()`,
- die Nutzung von `exitFrame` und `prepareFrame` für Lingo-Skripte, die Eigenschaften von Sprites verändern (zum Beispiel für Animationen, vgl. besonders Kapitel 1 und 9).

Versuchsreihe Lokalisierung der Maus – oder: Positionen und Richtungen

Wo ist die Maus gerade, wohin bewegt sie sich – dies herauszufinden, stellt eine Möglichkeit dar, dem Anwender Rückmeldung über seine Aktivität zu geben, ihn auf interessante Stellen hinzuweisen oder die Mausbewegung für entsprechende Programm-Reaktionen zu nutzen. Es gibt verschiedene Wege, den Ort der Maus auf der Bühne aufzuspüren bzw. die Lage der Maus über diversen Elementen auf der Bühne zu bestimmen. Vielleicht möchten Sie auch erfahren, ob der User mit der Maus an eine bestimmte Stelle geklickt hat oder mit ihr in eine bestimmte Region der Bildfläche gezogen ist. Oder Sie möchten herausfinden, ob der User die Maus eine Strecke weit fortbewegt hat und in welche Richtung.

Diese Fragen werden wir in den folgenden Experimenten erörtern. Damit werden auch die Grundlagen für die beiden folgenden Kapitel gelegt, wo es um nah verwandte Themen – nämlich Useraktivität und Buttons – gehen wird.

4.1 „Globale Detektoren“ für die Mausposition

Director bietet einige Systemeigenschaften und Funktionen, die jederzeit und unabhängig vom Skripttyp Rückmeldung darüber geben, an welcher Position (und möglicherweise über welchem Sprite) sich die Maus befindet, wo zuletzt geklickt wurde und was sich gerade unter der Maus befindet.

Folgende Systemeigenschaften werden wir in Experiment 1 nutzen:
`the mouseLoc` – Mausposition in Bühnenkoordinaten
`the mouseMember` – Darsteller unter der Maus
`rollover()` – Spritenummer des an der Mausposition zuoberst liegenden Sprites
`rollover(5)` – Angabe, ob sich Sprite 5 momentan unter der Maus befindet, als Boolean (`true` oder `false`)
`the clickLoc` – Position des letzten Mausklicks in Bühnenkoordinaten
`the clickOn` – SpriteNummer des zuletzt angeklickten Sprites (nur Sprites, die ein Skript enthalten, werden erkannt)
`the mouseChar`, `the mouseWord`, `the mouseLine` – Buchstaben-, Wort- und Zeilennummer des Felddarstellers unter der Maus

Beachten Sie, dass die Rückgabewerte ganz unterschiedliche Datentypen umfassen: Positionen werden als `point(xWert, yWert)`, Spritenummern als Zahl, Darstellerangaben als Darstellerreferenz wie beispielsweise `member 1 of castlib 1` und „Wahrheitswerte“ als `true` oder `false` zurückgeliefert.

4.2 Experiment 1: Globale Positionsangaben

Wir wollen in diesem Experiment Status-Informationen, die wir mit den aufgelisteten Systemeigenschaften und Funktionen erhalten, in einem Felddarsteller auf der Bühne anzeigen. Den zugehörigen Film finden Sie als *Mauspositionen.dir* auf der CD-ROM.

Versuchsaufbau

Sie brauchen für dieses Experiment einige Sprites auf der Bühne, um herausfinden zu können, über welchen Sprites sich die Maus befindet. Im 360 x 360 Pixel großen Beispielfilm *Mauspositionen.dir* sind daher einige Süßigkeiten (Sprites 2-4) vor einer Hintergrundfläche (Sprite 1) verteilt. Ein Felddarsteller (im Sprite 5) dient als Display, das über die Klick- bzw. Aufenthaltsorte der Maus Auskunft gibt.

Felddarsteller ziehen Sie auf der Bühne mit dem entsprechenden Tool aus der Werkzeug-Palette auf. Damit der Felddarsteller eine feste Größe hat, stellen Sie die Eigenschaft Rahmen auf „Fest“ (im Dialog „Darsteller-Eigenschaften“, den Sie bei ausgewähltem Felddarsteller mit der Tastenkombination Strg/⌘-I erhalten).

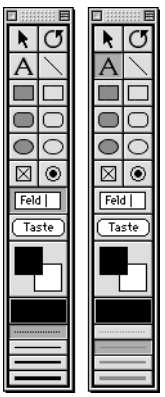


Abbildung 4.1:
In der Werkzeug-Palette ist das Tool für Felddarsteller (links) bzw. für Textdarsteller (rechts) ausgewählt.

Der Film enthält ein „Dummyskript“, das wir auf Sprite 4 platzieren. Es ist ein Behavior mit folgendem Inhalt, dessen Funktion wir später sehen werden:

```
on mouseDown me
  --
end
```

Im Skriptkanal platzieren Sie auf Bild 5 ein Skript, das den Abspielkopf anhält:

```
on exitFrame me
  go to the frame
end
```

Durchführung

Zunächst wollen wir die Mausposition ins „Ausgabe“-Feld schreiben lassen. In die Variable `Status` schreiben wir bei jedem Bilddurchgang (`on exitFrame`) diese Eigenschaft. Mit der Zeile `member("Ausgabe").text = Status` weisen wir die Variable dem Felddarsteller zu:

```
on exitFrame
  Status = "Mausposition:     " & the mouseLoc
  member("Ausgabe").text = Status
  go to the frame
end
```

Dieses Skript erweitern wir um weitere Eigenschaften:

```
on exitFrame
  Status = "Mausposition:   " & the mouseLoc & Return
  Status = Status & "Darsteller:   " & the mouseMember & \
   Return
  if not(voidP(the mouseMember)) then Status = Status & \
   "Darstellername:" && the mouseMember.name & Return
  Status = Status & "Rollover-Sprite:   " & rollover() & \
   Return
  Status = Status & "Rollover über 1:   " & rollover(1) & \
   Return
  Status = Status & "Letzte Klickposition:   " & \
   the clickLoc & Return
  Status = Status & "Letztes angeklicktes Sprite:   " & \
   the clickOn & Return
  Status = Status & "Wortnummer:   " && the mouseWord & Return
  if the mouseWord > 0 then Status = Status & "Wort:   " & \
   (the mouseMember).word[the mouseWord] & Return
  Status = Status & "Buchstabennummer:   " & the mouseChar & \
   Return
  if the mouseChar > 0 then Status = Status & "Buchstabe:   " \
   & (the mouseMember).char[the mouseChar] & Return
  member("Ausgabe").text = Status
  --
  go to the frame
end
```

Hier wird nach und nach eine umfangreiche Textausgabe zusammengestellt: In jeder Zeile schreiben wir einen Bezeichner und die entsprechende Lingo-Eigenschaft sowie `Return`, um einen Zeilenumbruch einzufügen. Mit dem Zeichen „\“ (in Director 7 muss das durch „¬“ ersetzt werden) können wir lange Skriptzeilen „umbrechen“, d.h. auf mehrere Zeilen verteilen. Bestimmte Eigenschaften können wir nur abfragen, wenn bestimmte Bedingungen erfüllt sind:

```
if not(voidP(the mouseMember)) then Status = Status & \
   "Darstellername:" && the mouseMember.name & Return
```

Wenn sich kein Darsteller unter der Maus befindet, liefert die Eigenschaft `the mouseMember` `VOID` zurück. Die Funktion `voidP()` überprüft, ob dies der Fall ist – sie liefert `true` zurück, wenn das Argument `VOID` ist. Mit der Überprüfung `if not(voidP(the mouseMember)) then` sagen wir also, dass der folgende Befehl nur ausgeführt werden soll, wenn sich tatsächlich ein Darsteller unter der Maus befindet. Ganz ähnlich verhält es sich mit

```
if the mouseWord > 0 then
```

und

```
if the mouseChar > 0 then
```

Beide Eigenschaften liefern `-1` zurück, wenn sich die Maus nicht über dem Textfeld befindet, und einen Zahlenwert `> 0`, wenn sich tatsächlich Text unter der Maus befindet.

Abbildung 4.2 und Abbildung 4.3 zeigen unterschiedliche Zustände unseres Versuchs.

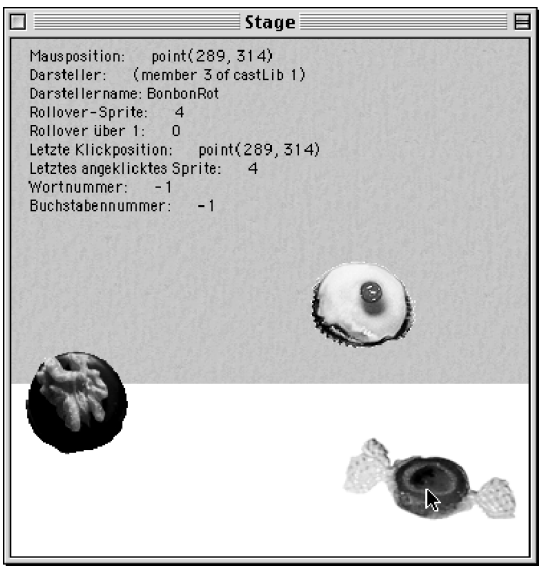


Abbildung 4.2: Die Maus befindet sich über dem Bonbon.

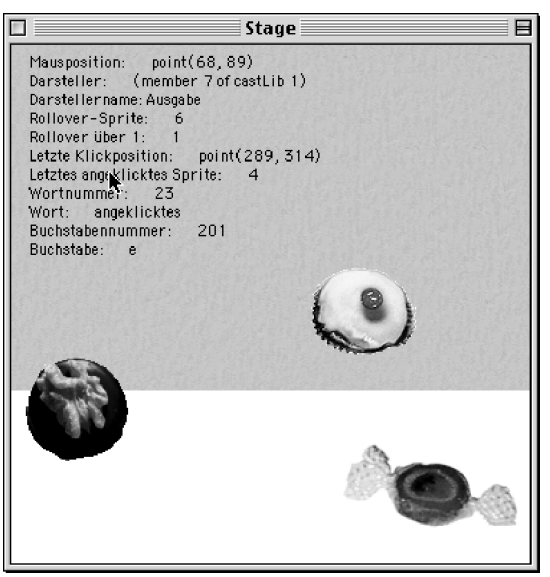


Abbildung 4.3: Die Maus befindet sich über dem Felddarsteller.* `mouseWord` *und* `mouseChar` *werden ausgegeben.

Da `mouseWord` und `mouseChar` nur die Wort- bzw. Buchstabennummer zurückgeben, lassen wir uns in den Zeilen

```
Status = Status & "Wort:      " & \
   (the mousemember).word[the mouseWord] & Return
```

und

```
Status = Status & "Buchstabe:     " & \
   (the mousemember).char[the mouseChar] & Return
```

zusätzlich noch das entsprechende Wort bzw. den entsprechenden Buchstaben aus dem Text anzeigen: `einFeldoderTextdarsteller.word[n]` liefert das `n`-te Wort im Darsteller zurück.

Diskussion

Mit `the mouseLoc` haben wir eine Systemeigenschaft an der Hand, die wir immer wieder einsetzen werden. Ist nur die x- oder nur die y-Koordinate der Mausposition gefragt, so können wir auch `the mouseH` (horizontaler, also x-Wert) und `the mouseV` (vertikaler, y-Wert) nutzen.

Interessant ist auch die Funktion `rollover()`: Ohne Argument verwendet, liefert sie das zuoberst liegende Sprite unter der Maus. Übergeben wir aber ein Argument (im Skript: `rollover(1)`), dann wird überprüft, ob sich die Maus über Sprite 1 befindet – unabhängig davon, ob noch andere Sprites in höheren Kanälen liegen. Der Rückgabewert ist dann `1` (`true`) oder `0` (`false`).

`the clickOn` hingegen dürfte im Experiment für Verwirrung gesorgt haben: Es liefert nämlich (fast) immer `0` zurück – wie sehr wir auch auf die Süßigkeitensprites oder den Hintergrund klicken. Einzige Ausnahme: Sprite 4 wird offenbar erkannt. Der Grund hierfür ist das Dummy-Skript, das wir auf Sprite 4 platziert haben. Das Verhalten von `the clickOn` ist dann doch konsequent: Nur Sprites, die ein Skript mit einem Mausevent (z.B. `on mouseDown`, `on mouseUp`) enthalten, werden als anklickbar „wahrgenommen“. Alle anderen Sprites werden von `the clickOn` ignoriert.

`the ClickOn`, `the mouseMember`, `rollover()` sind zwar sehr komfortabel einsetzbar, aber in vielen Fällen veraltet und überholt. Diese Eigenschaften fragen wir häufig in Sprite-Behaviors ab (vgl. die folgenden Experimente und das Kapitel „Buttons“): Dort haben wir dann direktere Möglichkeiten, einen Klick auszuwerten und das aktuelle Sprite bzw. seinen Darsteller zu benennen.

Ebenfalls teilweise veraltet sind die sehr komfortablen Eigenschaften `the mouseWord`, `the mouseChar`, `the mouseItem` und `the mouseLine`. Sie funktionieren nur mit Feld-, nicht mit Textdarstellern, und die entsprechenden Eigenschaften für Textdarsteller sind ebenfalls immer Spritebezogen:

```
the mouseChar     sprite(s).pointToChar(the mouseLoc)

the mouseWord     sprite(s).pointToWord(the mouseLoc)

the mouseItem     sprite(s).pointToItem(the mouseLoc)

the mouseLine     sprite(s).pointToLine(the mouseLoc)
                  sprite(s).pointToParagraph(the mouseLoc)
```

In einem Sprite-Behavior könnten Sie beispielsweise `pointToWord()` so einsetzen:

```
on mouseWithin me
  test = sprite(me.spriteNum).pointToWord(the mouseLoc)
  if sprite(me.spriteNum).member.word[test] contains
  "wichtig" then alert "Wichtig!"
  end if
end
```

Auch the `mouseMember` wird hier ersetzt (durch `sprite(me.spriteNum).member`). Sie sehen, der Behavior-Ansatz ist nicht einfacher und auch nicht platzsparender. Dennoch bietet er Vorteile, wenn wir daran gehen, die Funktion eines Sprites mit einem Behavior auf diesem Sprite zu verbinden.

Von Director 8 an ist die `pointTo…`-Syntax auch auf Felddarsteller anwendbar.

4.3 Experiment 2: Auf Mauspositionen reagieren

Ziel dieses Versuchs ist es, optische Veränderungen eintreten zu lassen, wenn sich der User mit der Maus über eine bestimmte Grenze hinwegbewegt hat, und die Veränderung wieder rückgängig zu machen, wenn er die Maus zurückzieht. Sie finden den Beispielfilm *MausH.dir* auf der CD-ROM.

Versuchsaufbau

Im 360 x 240 Pixel großen Film *MausH.dir* haben wir im Spritekanal 1 eine Hintergrundfläche installiert, im Spritekanal 2 den Darsteller „Praline“ und in Spritekanal 3 den Formdarsteller „Schraffur“.

Die auf der Bühne befindliche Praline soll animiert werden, wenn die Maus die Grenze von 70 Pixeln überschreitet. Außerdem wird der Bereich, in dem die Maus eine Aktion auslöst, mit einer Schraffur versehen.

Um einen solchen Effekt zu erzeugen, brauchen Sie ein Rechteck, das dann mit dem gewünschten Effekt gefüllt wird. Hier haben wir ein gefülltes Rechteck mit dem Rechteckwerkzeug aus der Werkzeugpalette mit der Linienstärke „keine Linie“ aufgezogen, und dann ein Muster wie in Abbildung 4.4 zugewiesen.

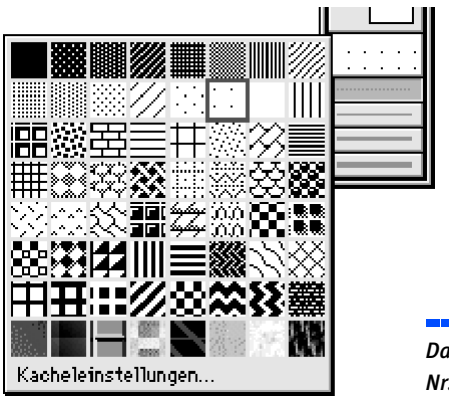


Abbildung 4.4:
Das Muster-Magazin von Director; Nr.14 ist ausgewählt.

Weisen Sie dem Sprite (über das Aufklappmenü nach Strg / ⌘-Klick) den Ink-Effekt „Hintergrund transparent“ zu. Um die Schraffur für das Experiment wieder unsichtbar zu machen, wählen wir im Muster-Magazin die weiße Fläche aus (Nr. 15).

Durchführung

Wir wollen zwei Wege ausprobieren, die simple „Grenzüberschreitung“ in Lingo zu implementieren. Das folgende Skript liegt auf Sprite 2.

Soll einfach nur überprüft werden, ob die Maus die 70-Pixel-Linie auf der x-Achse überschritten hat, so können wir die Systemeigenschaft `the mouseH` abfragen:

```
on exitFrame me
  if the mouseH > 70 then
    sprite(2).flipH = TRUE
    member("Schraffur").pattern = 14
  end if
end
```

Sobald die Maus rechts der 70 Pixel ist, wird die Praline gespiegelt und das Punktraster eingeschaltet. Der Effekt ist noch nicht sehr überzeugend, da er nicht rückgängig gemacht werden kann. Wir sollten folglich dem `if`-statement ein `else` hinzufügen und auch den Effekt (hier mit einer kontinuierlichen Rotation) aufpeppen:

```
on exitFrame me
  if the mouseH > 70 then
    sprite(2).rotation = sprite(2).rotation + 30
    member("Schraffur").pattern = 14
  else
    sprite(2).rotation = 0
    member("Schraffur").pattern = 0
  end if
end
```

Rotation und Pattern werden ausgeschaltet, sobald sich die Maus wieder links der 70-Pixel-Linie befindet. Abbildung 4.5 und Abbildung 4.6 zeigen die beiden Zustände.

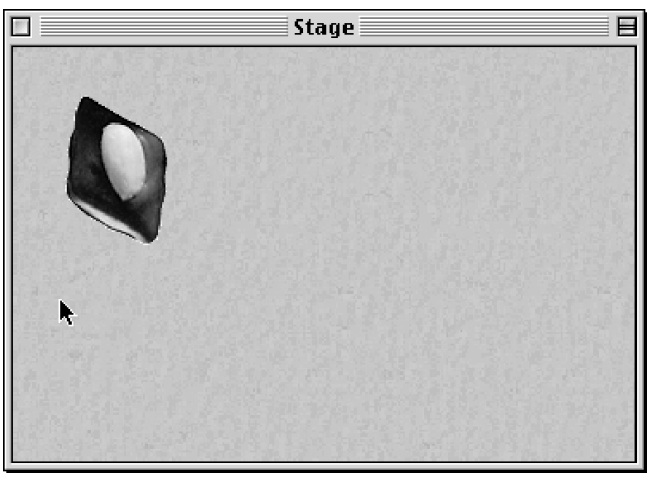


Abbildung 4.5: Die Maus ist links von der 70-Pixel-Linie.

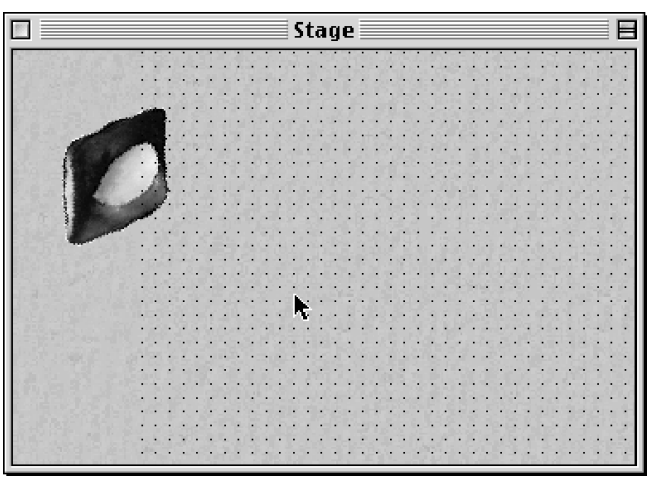


Abbildung 4.6: Die Maus hat die 70-Pixel-Linie überschritten.

Diskussion

Sie werden es bemerkt haben: Die Systemeigenschaft `the mouseH` liefert Werte, auch wenn sich die Maus außerhalb der Bühne befindet. Das ist in unserem Beispiel nicht weiter schlimm. Wenn Sie allerdings nur auf Mausbewegungen im Bühnenbereich reagieren wollen, so sollten Sie die Abfrage verfeinern:
Ändern Sie einmal testweise die Zeile

```
if the mouseH > 70 then
```

in

```
if inside(the mouseLoc, sprite(3).rect) then
```

Damit wird der Effekt nurmehr ausgelöst, wenn sich die Maus innerhalb des Sprites 3, also des Schraffurbereiches, befindet. Eine sinnvolle Einschränkung. Die Funktion `inside(einPunkt, einRechteck)` überprüft, ob sich `einPunkt` innerhalb von `einRechteck` befindet.

Ob sich die Maus in einem bestimmten Bühnenbereich befindet, ist insbesondere für Buttons und andere Klickbereiche wichtig. Im Kapitel Buttons finden Sie daher weitere Ansätze, die die Lingo-Events `on mouseWithin`, `on mouseEnter`, `on mouseLeave` verwenden.

Unser kleines Beispiel ließe sich mit zwei kleinen Sprite-Behaviors so umsetzen (im Film MausH_mod auf der CD-ROM):
Sprite 3 (Schraffur-Bereich):

```
on mouseEnter me
  sprite(me.spriteNum).member.pattern = 14
  sprite(me.spriteNum - 1).animate = true
end
on mouseleave me
  sprite(me.spriteNum).member.pattern = 0
  sprite(me.spriteNum - 1).animate = false
end
```

Sprite 2 (Praline):

```
property animate

on prepareFrame me
  if animate then
    sprite(me.spriteNum).rotation = \
      sprite(me.spriteNum).rotation + 30
  end if
end
```

Diese Umsetzung bewirkt genau das Gleiche wie die Variante mit `if inside(the mouseLoc, sprite(3).rect)`. Sie ist aber so formuliert, dass sie auf alle möglichen Sprites angewendet werden kann; einzige Bedingung ist, dass sie in aufeinander folgenden Spritekanälen liegen, wie in unserem Beispiel Sprite 2 und 3.

4.4 Experiment 3: Abstände ermitteln

Jetzt soll ermittelt werden, ob die Maus irgendwo auf der Bühne eine waagerechte Strecke von 70 Pixeln zurückgelegt hat. Für dieses Experiment nehmen wir die gleiche Darstelleranordnung wie im vorangegangenen Beispiel.

Durchführung

Sobald die Maus im Beispielfilm *MausStrecke.dir* eine horizontale Strecke von mehr als 70 Pixeln zurückgelegt hat, wird die Praline zu rotieren beginnen. Geht die Maus zurück in Richtung ihres Ausgangspunktes, springt die Praline in ihren Ausgangszustand zurück. Das Behavior „Strecke“ haben wir auf Sprite 2, der Praline, platziert.

```
property startH
on beginSprite me
  startH = the mouseH
end
```

Der x-Wert der Maus wird zu Beginn in der Variablen startH *festgehalten.*

```
on exitFrame me
  if abs(the mouseH - startH) > 70 then entfernt = TRUE
  else entfernt = FALSE
```

abs() *ist eine mathematische Funktion, die den Betrag aus mathematischen Ausdrücken ermittelt. Hier wird die Differenz aus horizontaler Maus-Ursprungsposition und aktueller Mausposition (in der Horizontalen) errechnet und mit* abs() *der absolute Wert (ohne Vorzeichen) ermittelt.*

```
  if entfernt then
    sprite(2).rotation = (sprite(2).rotation + 30) mod 360
  else
    sprite(2).rotation = 0
  end if
end
```

Modifikation

Was aber, wenn wir nicht den waagrechten Abstand, sondern die tatsächliche Strecke zwischen einem Startpunkt und der aktuellen Mausposition benötigen? Ein bisschen Schulmathematik hilft weiter – Sie erinnern sich sicher an Pythagoras?

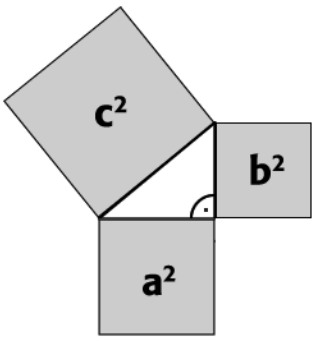


Abbildung 4.7:
Der Satz des Pythagoras: Im rechtwinkligen Dreieck ist die Summe der Quadrate über den Katheten gleich dem Quadrat über der Hypothenuse.

Wollen wir Abbildung 4.7 auf unseren Bildschirmraum übertragen, so wäre a ein Streckenabschnitt in x-Richtung, b ein solcher in y-Richtung. c ist der gesuchte Abstand. Um eine beliebige Strecke auf der Bühne zu vermessen, genügt es also, die Wurzel aus der Summe der Quadrate des x- und y-Abstandes zu ziehen, oder in Lingo:

```
on getDiff punkt1, punkt2
  vect = punkt1 - punkt2
  diff = sqrt(vect[1]*vect[1] + vect[2]*vect[2])
  return diff
end
```

An diese Funktion werden zwei Parameter übergeben: zwei Punkte, deren Abstand wir ermitteln wollen. Beispielsweise hat `punkt1` den Wert `point(100,100)`, `punkt2` ist `point(127, 33)`.

Die erste Zeile der Funktion ermittelt die Differenz zwischen beiden Punkten in Form eines Vektors: `vect = point(100, 100) - point(127,33) = point(-27, 67)`.

Um die Differenz als Länge der Strecke zwischen beiden Punkten zu ermitteln, nehmen wir den x-Abstand ins Quadrat `vect[1]*vect[1]`, ebenso den y-Abstand `vect[2]*vect[2]`, und ziehen die Wurzel (Squareroot: `sqrt()`) aus der Summe der beiden Ausdrücke. Am Ende liefert die Funktion mit `return` das Ergebnis zurück.

Schreiben Sie dieses Skript in ein Filmskript, damit es an jeder Stelle des Filmes aufgerufen werden kann.

Im Nachrichtenfenster können Sie dann testen:

```
put getdiff(point(100,100), point(127,33))
-- 72
```

Das Ergebnis ist für unsere Zwecke – nämlich die Berechnung des Abstandes in Pixeln – ausreichend. Wenn Sie die Nachkommastellen benötigen, müssen Sie `sqrt()` mit einer Fließkommazahl aufrufen:

```
diff = sqrt(float( vect[1]*vect[1] + vect[2]*vect[2]) )
```

Der Test im Nachrichtenfenster ergibt dann:

```
put getdiff(point(100,100), point(127,33))
-- 72.2357
```

Mit dieser Funktion im Handgepäck wollen wir das Ausgangsbeispiel so verändern, dass

1. als Startpunkt die Position des Sprites auf der Bühne genommen wird,
2. der tatsächliche Abstand zwischen der Mausposition und dem Startpunkt berechnet wird, und
3. die Animation stattfindet, wenn die Maus sich nahe (in einem 55-Pixel-Radius beispielsweise) am Sprite befindet.
4. Außerdem verwenden wir statt fester Spritenummern `me.spriteNum`, um das Behavior einem beliebigen Sprite zuordnen zu können.

Den Director-Film finden Sie als *MausAbstand.dir* auf der CD-ROM. Achten Sie auf die Kommentare, um zu sehen, wo wir die einzelnen Punkte umsetzen:

```
property startPos

on beginSprite me
  startPos = sprite(me.spriteNum).loc  -- (1, 4)
end

on prepareFrame me
  if getDiff(startPos, the mouseLoc) > 55 then --(2,3)
    entfernt = TRUE
  else
    entfernt = FALSE
  end if

  if entfernt then
    sprite(me.spriteNum).rotation = 0 -- (4)
  else
    sprite(me.spriteNum).rotation = \
     (sprite(me.spriteNum).rotation + 30) mod 360 -- (4)
  end if
end
```

Im zweiten Teil des Filmes (vgl. Abbildung 4.8) sehen Sie eine Umsetzung dieses Skriptes mit vielen rotierenden Sprites auf der Bühne.

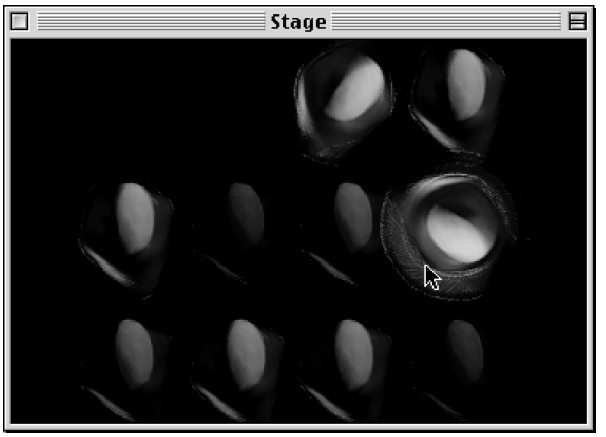


Abbildung 4.8: Das Sprite rotiert, sobald die Maus ihm nahe kommt.

Folgende Änderungen haben wir für diesen kleinen Effekt vorgenommen:

1. Im Skript haben wir die Rotation verlangsamt (+ 1 statt + 30)
2. Im Sprite- bzw. Eigenschafteninspektor haben wir allen Sprites einen Opazitätswert (`blend`) von nur 5 (also 5% Deckung) und „Spuren“ (`trails`) zugewiesen.
3. Das Skript haben wir so erweitert, dass zusätzlich abgefragt wird, ob die Maus gedrückt ist; ist dies der Fall, wird die Animation nicht abgespielt; stattdessen wird die Bühne neu gezeichnet und die Spuren werden entfernt (`the stagecolor = the stagecolor`).

So sieht das modifizierte Skript aus; die Funktion `getDiff()` ist in einem Filmskript definiert:

```
property startPos

on beginSprite me
  startPos = sprite(me.spriteNum).loc
end

on prepareFrame me
  if the mouseDown then
    the stagecolor = the stagecolor
  else
    if getDiff(startPos, the mouseLoc) > 55 then entfernt = TRUE
    else entfernt = FALSE
    if entfernt then
      sprite(me.spriteNum).rotation = 0
    else
      sprite(me.spriteNum).rotation = \
       (sprite(me.spriteNum).rotation + 1) mod 360
    end if
  end if
end
```

Diskussion

Die zuletzt gewählte Lösung bringt uns kreisförmige „Hotspots" um einen beliebigen Bühnenpunkt, wenn der Bereich kleiner als ein gewählter Abstand betrachtet wird. Ebenso können wir Bereiche definieren, die kreisförmige Löcher enthalten, wenn ein Abstand größer als ein bestimmter Wert gelten soll. Der Vorteil dieser Herangehensweise ist, dass es wirklich ein beliebiger Punkt und ein beliebiger Abstand sein können, die wir im Skript definieren. Alternativ könnten wir natürlich auch Sprites an die entsprechenden Stellen legen und fragen, ob sich die Maus über dem Sprite befindet oder nicht. Dieser Ansatz ist nicht ganz so flexibel, aber immerhin haben wir dann eine visuelle Repräsentation des Klick- oder Hotspot-Bereichs im Drehbuch.

Die in der Modifikation vorgestellte Lösung ließe sich beispielsweise sehr gut mit einem Behavior realisieren, das die jeweiligen Sprite-Begrenzungen als Auslösebereich nimmt:

```
property animate

on mouseEnter me
  animate = TRUE
end

on mouseLeave me
  animate = FALSE
end

on prepareFrame me
  if animate then
    sprite(me.spriteNum).rotation = \
     (sprite(me.spriteNum).rotation + 1) mod 360
  else
    sprite(me.spriteNum).rotation = 0
  end if
end
```

Diese Lösung finden Sie ebenfalls im Film *Mausabstand.dir*. Sie benötigt keine Entfernungsberechnung, da der Sprite-Bereich als Auslöser genommen wird.

4.5 Experiment 4: Bewegungsrichtung ermitteln

Wohin wird die Maus bewegt? Nicht die absolute Position, sondern die Richtung der Bewegung soll nun ermittelt werden. Dieser Versuch zeigt einen einfachen Weg, Richtungsangaben wie „oben“, „rechts“ oder „links unten“ zu erhalten. Den Director-Film finden Sie als *MausRichtung.dir* auf der CD-ROM.

Versuchsaufbau

Der Film besteht nur aus einer Hintergrundfläche im ersten Spritekanal, auf der einige Gegenstände abgebildet sind, die den User dazu bringen sollen, mit der Maus hin- und herzufahren.

Zur Kontrolle der Abläufe in diesem Experiment werden Sie Directors Nachrichtenfenster nutzen. Mit Hilfe des Befehls put veranlassen Sie in Ihrem Skript die Ausgabe von Werten bzw. eines Status von Abläufen. Das Nachrichten-Fenster öffnen Sie mit Strg / ⌘+M. Da es keine Taste gibt, um das Nachrichtenfenster zu leeren, löschen Sie seinen Inhalt mit Strg / ⌘+A und dann ← bzw. Entf.

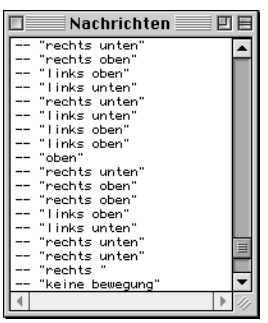


Abbildung 4.9:
Im Nachrichtenfenster erscheinen im laufenden Film veränderte Werte, deren Abfrage Sie mit *put* ***angeordnet haben.***

Wenn Sie mehrere put-Anweisungen im Skript haben, kann der Output im Nachrichtenfenster verwirren. Kommentieren Sie die put-Anweisungen aus, die Sie nicht in erster Linie brauchen. Markieren Sie die entsprechenden Zeilen im Skript und drücken Sie dann die Auskommentierungstaste im Skriptfenster. Um die Zeilen wieder zu aktivieren, drücken Sie die Aktivierungstaste (vgl. Abbildung 4.10).

Abbildung 4.10: Mit den beiden Buttons mit dem Kommentarzeichen (--) im Skriptfenster können Sie Skriptbereiche auskommentieren bzw. den Kommentar wieder aufheben.

Durchführung

Öffnen Sie den Beispielfilm *Mausrichtung.dir*, und lassen Sie ihn bei geöffnetem Nachrichtenfenster abspielen. Beobachten Sie die Änderungen im Nachrichtenfenster. Statt einer Ausgabe in Nachrichtenfenster würde in einem realen Projekt eine Anweisung zu Aktionen erfolgen (z.B. Text einblenden, Sound abspielen, Richtungspfeil rotieren). Das Behavior „Richtung“ befindet sich auf Sprite 1.

```
property lastloc, counter
on beginsprite me
  lastloc = the mouseloc
```

Eingangs wird die Mausposition in die Eigenschaftsvariable lastLoc *eingetragen.*

```
end
on exitFrame me
  -- counter = counter + 1
  -- if counter = 15 then
  --   counter = 0
```

Aktivieren Sie diese drei Zeilen und das schließende end if *am Skriptende, so wird der zeitliche Abstand der Überprüfungen erhöht (vgl. Text).*

```
  newloc = the mouseloc
  richtung = ""
```

Bei jenem Bildwechsel wird die Mausposition in der lokalen Variablen newLoc *aktualisiert. Die Funktion* the mouseLoc *speichert die aktuelle Mauszeigerposition als Punkt (mit den Punkt-Koordinaten x, y). Die lokale Variable* richtung *wird mit einem leeren String* "" *initialisiert.*

```
  if newloc[1] > lastloc[1] then richtung = "rechts "
```

Hier werden die Punktvariablen newloc *und* lastloc *verglichen, und zwar x-Wert und y-Wert getrennt. Die Angabe des Listenindex* [1] *entspricht dem x-,* [2] *dem y-Wert.*

```
  if newloc[1] < lastloc[1] then richtung = "links "
  if newloc[2] > lastloc[2] then richtung = richtung & "unten"
  if newloc[2] < lastloc[2] then richtung = richtung & "oben"
  if richtung = "" then richtung = "keine bewegung"
```

Wenn keine der vorangehenden Bedingungen zutraf, ist richtung *weiterhin* "". *In diesem Fall wird* "keine Bewegung" *eingetragen.*

```
    put richtung
    lastloc = newloc
```

Hier wird die letzte aktuelle Position in lastLoc *eingetragen; sie wird im nächsten Frame als Vergleichsposition genommen, um dann ggf. eine neue Mausbewegung festzustellen und um neue Richtungsergebnisse anzuzeigen.*

```
  -- end if
  go to the frame
end
```

Diskussion

Erklärungsbedürftig ist möglicherweise, warum wir in diesem und anderen Beispielen (vgl. Experiment 3 in diesem Kapitel) auf Punktvariablen zugreifen, als ob es Listen wären.

Tatsächlich ist ein Punkt (z.B. `point(120,70)`) eine spezielle Form einer Liste, und wir können mit der normalen Listensyntax auf die beiden Positionen zugreifen. Vergleichen Sie hier den Zugriff auf eine Punkt-Variable und eine lineare Liste mit zwei Elementen:

```
p = point(120,70)
put p[1]
-- 120
put p[2]
-- 70

liste = [119, 69]
put liste[1]
-- 119
put liste[2]
-- 69
```

Eine weitere Frage bzgl. unserer Umsetzung dürfte dann auftreten, wenn Sie eine hohe Bildrate einstellen: Das Skript wird dann nämlich fast nur noch "`keine Bewegung`" zurückmelden. Der Grund ist, dass die Überprüfung (und damit die Neuzuweisung `lastloc = newloc`) zu häufig stattfindet. Mit den hier auskommentierten Skriptzeilen richten Sie einen Zähler ein, der mit der passenden `if`-Abfrage dafür sorgt, dass die Überprüfung nur noch alle 15 Framedurchgänge stattfindet. Das ist ein sinnvoller Wert für eine Bildrate von 15 Bildern pro Sekunde: Damit findet einmal pro Sekunde eine Überprüfung statt. Auf diesem Weg reduzieren Sie die Genauigkeit, aber Sie erhalten aussagekräftigere Ergebnisse, da die Bewegung dann über einen längeren Zeitraum untersucht wird.

Wenn Sie – anstatt das Ergebnis ins Nachrichtenfenster auszugeben – ein Sprite in Form eines Pfeils in die Richtung zeigen lassen wollen, die Sie als Bewegungsrichtung der Maus ermitteln, so könnten Sie wie folgt vorgehen (das Skript legen Sie auf das zu drehende Sprite, der Pfeil zeigt anfangs nach oben):

```
property lastloc, counter
on beginsprite me
  lastloc = the mouseloc
end
on prepareFrame me
  newloc = the mouseloc
  vect = newloc - lastloc
  if vect[1] = 0 then
    if vect[2] = 0 then
      sprite(me.spriteNum).blend = 25
      return -- Skriptabbruch!
    else if vect[2] > 0 then
      rot = 180
    else if vect[2] < 0 then
      rot = 360
    end if
  else if vect[1] > 0 then
    if vect[2] > 0 then rot = 135
    else if vect[2] = 0 then rot = 90
    else if vect[2] < 0 then
      rot = 45
    end if
  else
    if vect[2] > 0 then rot = 225
    else if vect[2] = 0 then rot = 270
    else if vect[2] < 0 then
      rot = 315
    end if
  end if
  lastloc = newloc
  sprite(me.spriteNum).blend = 100
  sprite(me.spriteNum).rotation = rot
end
```

Der Pfeil zeigt in 45-Grad-Schritten in die Richtung, in die sich die Maus bewegt hat. Findet keine Bewegung statt, so wird er gedimmt (`sprite(me.spriteNum).blend = 25`) und zeigt weiter in die zuletzt ermittelte Richtung. Die Zutaten zu dieser Lösung finden Sie ebenfalls in *Mausrichtung.dir*.

4.6 Experiment 5: Die Bildschirmposition für Stereoeffekte nutzen

Seit Director 8 können Sie mit der Eigenschaft `pan` Positionswechsel der Maus auch mit einem Sound begleiten, der auf veränderte Positionen Bezug nimmt. Wenn der User also auf der Bühne ein Element hin- und herbewegt, kann der Sound so eingestellt werden, dass er scheinbar mehr von links oder von rechts ertönt. Im Extremfall erlaubt `pan` einen Schwenk von ganz links bis ganz rechts, indem Sie die Werte -100 bis 100 zuweisen. Selbstverständlich hören Sie diesen Effekt nur, wenn Ihr Rechner Stereo-Sound wiedergibt.

Versuchsaufbau

Im Beispiel *RichtungUndSound.dir* benötigen wir ein Motorengeräusch (ein importiertes AIFF, Mono, dessen Eigenschaft „Loop" im Propertyinspektor auf `TRUE` gesetzt wird). Ein Auto soll über die gesamte Breite der Bühne ziehbar sein. Die Ziehbarkeit können wir im Property-Inspektor einschalten, oder mit Lingo:

```
on beginSprite
  sprite(3).moveableSprite = TRUE
end
```

Das Auto weist zunächst nach rechts, soll aber (mit der Eigenschaft `flipH`) gespiegelt werden, wenn es in die andere Richtung gezogen wird. In den Spritekanal 1 legen wir eine Hintergrundfläche, „Asphalt". Das Cabriolet parken wir in Sprite 3, den Sound „Alfa" im ersten Soundkanal.

In den Skriptkanal kommt auf Bild 5 das Stopper-Skript:

```
on exitFrame me
  go to the frame
end
```

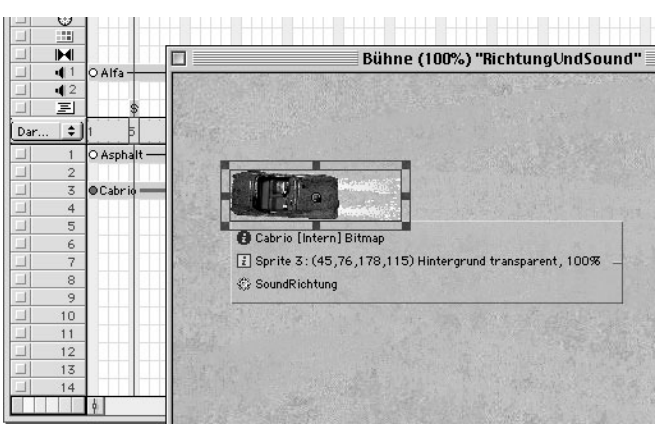


Abbildung 4.11: Der Aufbau des Experiments "Richtung und Sound"

Durchführung

Das Motorengeräusch soll beim Abspielen je nach Position des Autos aus dem entsprechenden Lautsprecher ertönen. Das regeln wir mit dem Behavior „SoundRichtung“ auf dem Cabrio (Sprite 3):

```
property mausZuletzt, Mitte

on beginSprite
  sprite(3).moveableSprite = TRUE
  Mitte = (the stage).rect.width / 2.0
```

In die Property Mitte *wird die Hälfte der Bühnenbreite als Fließkommazahl eingetragen.*

```
end

on exitFrame me
  if the mouseDown then
    if mouseH() - MausZuletzt < -3 then
```

Um zu verhindern, dass das Auto bei der kleinsten Bewegung herumflippt, wird eine Mindestbewegung von drei Pixeln angenommen - sobald die Schwelle überschritten ist, ändert sich die „Fahrtrichtung" des Autos. Bei kleinsten Bewegungen ändert sich nichts.

```
      sprite(3).flipH = 1
    else if mouseH() - MausZuletzt > 3 then
      sprite(3).flipH = 0
    end if
  end if

  mauszuletzt = the mouseH
  test = sprite(3).locH
  mypan = ((test - Mitte) / Mitte) * 100
```

*Die Teilstrecke (*test - Mitte*) geteilt durch die Gesamtstrecke (*Mitte*) ergibt einen Prozentwert, der sich im rechten Bühnenbereich zwischen 0 und 100 Prozent bewegt, im linken zwischen -100 und 0 Prozent. Durch die Multiplikation mit 100 erhalten wir die für* pan *notwendigen Werte zwischen -100 und 100.*

```
  sound(1).pan = mypan
end
```

Wenn Sie im Beispielfilm das Auto ziehen, die Pan-Wirkung beobachten und dann die Maustaste loslassen, erscheint eine Abbildung, die Ihnen die verschiedenen Streckenmaße veranschaulicht.

Modifikation

Im folgenden Skript erweitern wir das vorangegangene Beispiel wieder leicht, wir ändern aber vor allem den Modus, wie das Auto auf der Bühne gezogen wird.

In diesem Fall soll

1. das Behavior wieder von festen Spritenummern unabhängig gemacht werden (`me.spriteNum` statt 3),
2. das Auto nur waagrecht bewegt werden können (`moveableSprite = FALSE`, dafür aber binden Sie die Sprite-Position an die horizontale Mausposition),
3. sich im Drehbuch kein Sound befinden, sondern mit Director 8-Soundlingo das Starten und Stoppen des Soundes erreicht werden (`sound(1).play(member("Alfa"))`, `sound(1).stop()`),
4. wenn das Auto sich außerhalb der Bühne bewegt, die Sound-Lautstärke heruntergefahren werden (`sound(1).volume`).

Das folgende Skript setzt dies um. Die Property `fahren` wird bei `mouseDown` auf `TRUE` gesetzt; ist sie `TRUE`, so findet im `exitFrame`-Skript die Animation statt. Da wir nur den `locH` des Sprites verändern, kann das Sprite nur in der Waagrechten bewegt werden. `mouseUp` und `mouseUpOutside` sollen identische Auswirkung haben: den Sound stoppen und die Animation unterbinden (`fahren = false`):

```
property myLocH, LocZuletzt, Mitte
property fahren

on beginSprite me
  sprite(me.spriteNum).moveableSprite = FALSE
  Mitte = (the stage).rect.width / 2.0
end

on mouseDown me
  sound(1).play(member("alfa"))
  fahren = true
end

on mouseUp me
  sound(1).stop()
  fahren = false
end

on mouseUpOutside me
  mouseUp me
end

on exitFrame me
  if fahren then
    mylocH = the MouseH
```

```
    sprite(me.spriteNum).locH = myLocH
    --  (1) flipH oder nicht?
    if mylocH - LocZuletzt < -3 then
      sprite(me.spriteNum).flipH = 1
    else if mylocH - LocZuletzt > 3 then
      sprite(me.spriteNum).flipH = 0
    end if
    locZuletzt = myLocH
    -- (2) Pan-Berechnung
    mypan = ((mylocH - Mitte) / Mitte) * 100
    sound(1).pan = mypan
    -- (3) Volume-Veränderung
    deltavol = abs(mypan) - 100
    if deltavol > 0 then
      sound(1).volume = 100 - deltavol
    else
      sound(1).volume = 100
    end if
  end if
end
```

Die Skriptteile (1) und (2) entsprechen funktional denen in obigem Experiment. Teil (3) kommt hinzu: Er hat den Effekt, dass die Lautstärke verringert wird, wenn das Auto sich rechts oder links von der Bühne wegbewegt. Für die Berechnung von `deltavol` nutzen wir die schon vorhandene Property `mypan`. Sie hat an den Bühnenkanten die Werte -100 respektive 100, sodass `deltavol = abs(mypan) - 100` an den Bühnenkanten 0 ergibt und einen höheren Wert, wenn sich die Maus weiter nach außen bewegt, egal ob rechts oder links der Bühne. Außerhalb der Bühne wird die Lautstärke folglich mit `sound(1).volume = 100 - deltavol` verringert – je weiter weg, desto mehr. Auf der Bühne wollen wir die maximale Soundlautstärke haben, also `sound(1).volume = 100`.

Diskussion

Wie Sie gesehen haben, lässt sich die Eigenschaft `pan` für Director-Sounds per Lingo verändern, unabhängig davon, ob der Sound im Soundkanal oder via Skript abgespielt wird. Der Grund dafür ist das so genannte Sound-Objekt, das die Kontrolle über einen Soundkanal unabhängig von der Abspielart übernimmt. Diese Art der Behandlung von Sound wurde mit Director 8 eingeführt und macht eine Reihe von älteren Soundbefehlen (beispielsweise `puppetSound Kanalnummer, Darsteller`) überflüssig.

Faulheit oder Eifer

Was tut der User beim Durchstöbern Ihres Films? Etwa nichts? Dann möchten Sie ihn möglicherweise dazu bringen. Vielleicht können Sie ihn nach einer bestimmten Zeit einfach daran erinnern, aktiv zu werden, oder Sie veranlassen selbst einen Bildwechsel oder eine andere Aktion. Vielleicht möchten Sie auch nur in Erfahrung bringen, ob der User an eine bestimmte Stelle geklickt hat oder wie oft er bestimmte Elemente angeklickt hat.

Die Experimente dieser Versuchsreihe geben Ihnen einige Mittel an die Hand, wie Sie die User-Aktivität messen können.

5.1 Experiment 1: Filmskripte und „bevorzugte Event-Handler“

Zwei der Experimente dieser Versuchsreihe nutzen Filmskripte. Wieso überhaupt ein Filmskript im Unterschied zu einem Behavior einrichten?

Zunächst einmal bietet Director eine Reihe von Events, die dazu dienen können, einen Film zu konfigurieren (`on prepareMovie`, `on startMovie`) bzw. bei Filmwechsel oder -ende aufzuräumen (`on stopMovie`). Alle diese Events nutzen Sie in Filmskripten. Beispielsweise können Sie in einem `startMovie`-Handler globale Variablen initialisieren, in einem `prepareMovie`-Handler Sprites, die im Drehbuch liegen, aber beim Filmstart nicht sichtbar sein sollen, ausblenden, und in einem `stopMovie`-Handler eine Voreinstellungsdatei schreiben, die beim nächsten Filmstart wieder genutzt werden kann.

Zur „typischen“ Nutzung von Filmskripten gehört auch, dass Sie hier „globale“, also allgemein nutzbare Funktionen definieren können. Soll also nicht nur ein Behavior eine bestimmte Funktion nutzen können, sondern eine Reihe von Skripten, so würden Sie die Funktion üblicherweise in einem Filmskript anlegen.

Recht speziell ist eine Verwendungsweise, die wir in Experiment 2 nutzen werden. Director bietet nämlich eine Reihe von „primary event handlers“, also von bevorzugten, primären Skripten, die bei bestimmten Events aufgerufen werden. Diese werden über Systemeigenschaften konfiguriert:

- `the mouseDownScript`
- `the mouseUpScript`
- `the keyDownScript`
- `the keyUpScript`
- `the timeoutScript`

Diesen Eigenschaften weisen Sie als String einen Handlernamen zu. Ebendieser Handler – womit wir wieder beim Thema „Filmskript“ wären – muss als selbst definierte Funktion in einem Filmskript stehen.

Ein (erstes) Filmskript können Sie mit der Tastenkombination ⌘ / Strg + ⇧ + U öffnen. Es lässt sich aber auch jedes Skript über den Eigenschaften-Dialog des Skriptes in ein Filmskript umwandeln (vgl. Abbildung 5.1 und Abbildung 5.2).

Abbildung 5.1: Im Fenster „Skriptdarsteller-Eigenschaften“ können Sie den Skripttyp in Director 7 verändern.

Abbildung 5.2:
Die entsprechende Einstellung im Eigenschafteninspektor von Director 8

Ein weiterer Unterschied zwischen Behaviors und Filmskripten liegt in ihrer Lokalisierung und Erkennung in der Besetzung. Behaviors werden auf der Bühne auf dem Sprite-Overlay (und in der Skript-Preview des Drehbuchfensters) angezeigt. Filmskripte finden Sie nur in der Besetzung. Dort setzen sich Filmskripte und Behaviors durch verschiedene Icons voneinander ab:

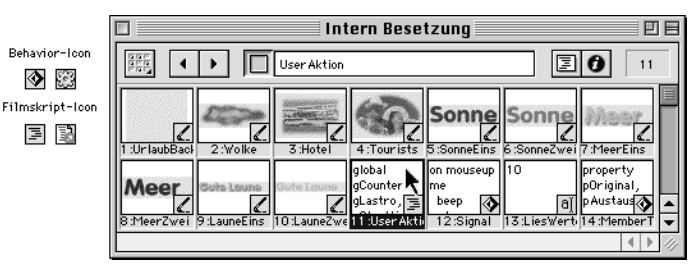


Abbildung 5.3: Die Skript-Icons haben sich zwischen den Director-Versionen 7 und 8 etwas verändert.

Wenn Sie das Filmskript in der Besetzung doppelklicken, öffnet sich direkt das Skriptfenster.

Versuchsaufbau

Legen Sie in einem leeren Film mit dem Button-Werkzeug der Werkzeugleiste einen Button an und beschriften Sie ihn (z.B. mit „mousedown“).

Legen Sie auf dem Button ein Behavior an mit folgendem Inhalt:

```
on mouseDown me
  put "mouseDown im Buttonskript"
end
```

Geringfügig modifizierte Skripte kommen als Frameskript in den Skriptkanal des Drehbuchs:

```
on mouseDown me
  put "mouseDown im Frameskript"
end

on exitFrame me
  go to the frame
end
```

und in ein Filmskript:

```
on mouseDown
  put "mouseDown im Filmskript"
end
```

Das Filmskript erweitern wir noch um folgende Zeilen. Die Funktion `mymousedown` ist unser eigenes kleines Skript, mit dem wir im ganzen Programm auf ein `mouseDown` reagieren wollen. Es steht im Filmskript, damit wir den Funktionsnamen der Systemeigenschaft `the mouseDownScript`, dem `primary event handler`, zuweisen können.

```
on startMovie
  the mouseDownscript = "mymousedown"
end

on mymousedown
  put "mouseDown im primary event handler "
end
```

Damit haben wir uns eine kleine Testumgebung geschaffen, die es uns ermöglicht, zu überprüfen, wie ein spezielles Event, nämlich `mouseDown`, in der Objekthierarchie von Director behandelt wird. Den Film *Eventtest.dir* finden Sie auf der CD-ROM.

Übrigens können Sie jedes Event so testen: einfach ausprobieren!

Durchführung

Wenn wir den Film starten und einmal auf den Button, einmal auf den Bühnenhintergrund klicken, werden folgende unterschiedlichen Ergebnisse ins Nachrichtenfenster geschrieben:
Bei Klick auf den Button:

```
-- "mouseDown im primary event handler"
-- "mouseDown im Buttonskript"
```

Bei Klick auf die Bühne:

```
-- "mouseDown im primary event handler"
-- "mouseDown im Frameskript"
```

Wir sehen zweierlei:

- Das `mouseDownScript` (primary event handler) wird in beiden Fällen zuerst ausgeführt; es fängt aber das Event nicht ab, sondern leitet es einfach durch.
- Wenn auf den Button geklickt wird, wird das Frameskript nicht mehr ausgeführt; in beiden Fällen wird der `mouseDown`-Handler im Filmskript gar nicht ausgeführt. Event-Handler in Behaviors und Frameskripten sorgen also dafür, dass das entsprechende Event nicht mehr weitergeleitet wird: Sie „schlucken“ das Event.

Um diesem Sachverhalten noch etwas weiter auf den Grund zu gehen, legen wir ein zweites Behavior auf den Button mit folgendem Skript:

```
on mouseDown me
  put "mouseDown im 2. Buttonskript"
  put " -- pass --"
  pass
end
```

Das erste Buttonskript und das Frameskript erweitern wir ebenfalls um ein `pass`:

```
on mouseDown me
  put "mouseDown im Frameskript"
  put " -- pass --"
  pass
end
```

Wir erhalten folgende Ergebnisse im Nachrichtenfenster. Beim Klick auf den Button:

```
-- "mouseDown im primary event handler "
-- "mouseDown im Buttonskript"
-- " -- pass --"
-- "mouseDown im 2. Buttonskript"
-- " -- pass --"
-- "mouseDown im Frameskript"
-- " -- pass --"
-- "mouseDown im Filmskript"
```

Beim Klick auf die Bühne:

```
-- "mouseDown im primary event handler "
-- "mouseDown im Frameskript"
-- " -- pass --"
-- "mouseDown im Filmskript"
```

Wir sehen, dass dank des `pass` das `mouseDown`-Event die ganze Event-Hierarchie durchwandert:

Zunächst zum primary event handler, dann zum (ersten) Sprite-Behavior, dann zum zweiten, falls beide ein `pass` enthalten weiter zum Frameskript, falls dieses ein `pass` enthält weiter zum Filmskript.

Berücksichtigen Sie dabei, dass `pass` unmittelbar ausgeführt wird. Alles was nach dem `pass` noch im Skript steht, wird nicht mehr beachtet.

Die Sonderstellung des primary event handlers werden wir uns im Experiment 2 – am Beispiel von `the timeoutScript` – zunutze machen.

5.2 Experiment 2: Timeout-Skripte

Wie lange ist der letzte Klick her? Wie lange die letzte Mausbewegung? Wie können wir mit unterschiedlichen Skripten eine festgelegte Zeit nach der letzten Aktion reagieren? Um diese Fragen geht es in unserem Experiment.

Versuchsaufbau

Der Film *timeout.dir* ist 360 x 240 Pixel groß. Auf Bild 1 und auf Bild 29 platzieren Sie bitte jeweils einen Marker „Erste Szene“ und „Zweite Szene“. Im Skript-Kanal platzieren Sie auf Bild 28 und 56 jeweils das schon bekannte „Stopper-Skript“:

```
on exitFrame me
  go to the frame
end
```

In der ersten Szene wollen wir drei Süßigkeiten auf der Bühne verstreuen (Sprites 2, 3 und 4), damit der Anwender etwas anzuklicken hat – diese Aktivität soll in diesem Experiment ermittelt werden. Zwei Anzeigen (Felddarsteller in den Spritekanälen 5 und 6) richten sich an den User: die Aufforderung, sich zu einem Klick zu entscheiden, und ein Zählwerk, das die Anzahl der Klicks wiedergibt. Auch ein entsprechender Sound („Bitte klicken Sie auf ein Feld“) ist in der Besetzung vorhanden.

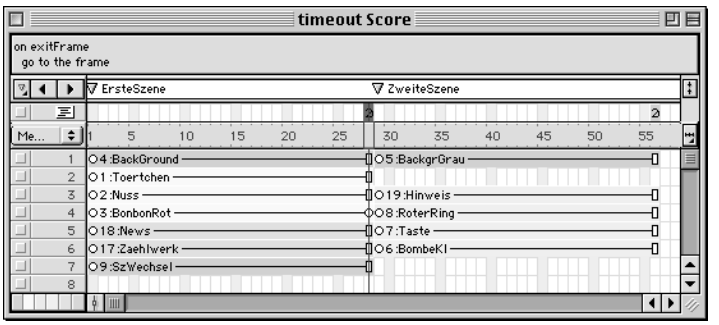


Abbildung 5.4: Das Drehbuch von timeout.dir.

Ein Button „Zweite Szene“ führt in den zweiten Bereich des Films. Das Skript ist entsprechend einfach:

```
on mouseUp me
  go "ZweiteSzene"
end mouseEnter
```

Kommt der User in der zweiten Szene mit der Maus in einen besonders markierten Bereich (Sprite 4, Darsteller „RoterRing“), so ertönt ein Signal. Das Tonabspielen bewerkstelligt das Behavior „SignalAusgeben“ auf Sprite 4:

```
on mouseEnter me
  puppetSound 1, "Signal"
end mouseEnter

on mouseLeave
  puppetSound 1, 0
    Sobald sich der User aus dem markierten Bereich herausbegibt,
    wird der Ton wieder ausgeschaltet.
end mouseLeave
```

Kommt er mit der Maus direkt über die Taste in Sprite 5, erscheint ein Warnhinweis als Grafik. Das folgende Skript versteckt den roten Kreis und die Taste und zeigt stattdessen den Warnhinweis (Sprite 6) an. Außerdem wird auch hier der Sound ausgeschaltet:

```
on mouseEnter
    puppetSound 1, 0
    sprite(4).loc = point(-1000,-1000)
    sprite(5).loc = point(-1000,-1000)
    sprite(6).loc = point(180,112)
end
```

Auf dem Warnhinweis in Sprite 6 liegt das Behavior „FilmAus“, das den Film (in der Authoringumgebung) anhält.

```
on mouseUp me
  halt
end mouseUp

on beginSprite
  sprite(6).loc = point(-1000,-1000)
end
```

Im `beginSprite`-Handler wird der Sprite mit dem Warnhinweis weit außerhalb der Bühne platziert. Dies hat einen ähnlichen Effekt wie die Verwendung der Sprite-Eigenschaft `visible`, allerdings ohne die Nachteile derselben. Der Befehl `sprite(eineSpriteNummer).visible = FALSE` hat nämlich zur Folge, dass der gesamte Spritekanal unsichtbar geschaltet wird, bis Sie die Eigenschaft wieder auf `TRUE` setzen. Wenn Sie die Position eines Sprites verändern, wirkt sich diese Änderung dagegen nur auf das gerade angesprochene Sprite aus.

Wir werden in der Durchführung einige weitere Skripte anlegen, die auch globale Variablen verwenden.

▶ Globale Variablen im Watcher verfolgen

Globale Variablen erzeugen Sie mit dem vorangestellten Schlüsselwort `global`. Wie die Property-Variablen deklarieren Sie diesen Variablen-Typus zu Beginn eines Skripts; üblicherweise wird globalen Variablen ein kleines „g“ vorangestellt, um sie von anderen Variablen unterscheiden zu können.

Ein nützliches Tool zum Überprüfen von globalen Variablen und von Director-eigenen Properties ist der Watcher. Sie erreichen dieses Fenster über das Menü Fenster/Watcher. Die simpelste Prozedur zum Einsatz des Watchers: Wählen Sie eine globale Variable in einem Skript aus, und klicken Sie auf den Button „Ausdruck verfolgen“. Sie können dann die Veränderungen von Werten Ihrer globalen Variablen beim Abspielen des Films verfolgen.

Abbildung 5.5: Der Button „Ausdruck verfolgen“ im Skriptfenster schickt den im Skript markierten Lingo-Ausdruck (wie globale Variablen, System- und Sprite-Eigenschaften) in den Watcher.

Für unser Experiment ist es besonders nützlich, wenn Sie den Watcher geöffnet halten und einige der verwendeten Lingo-Ausdrücke so im Auge behalten (vgl. Abbildung 5.6).

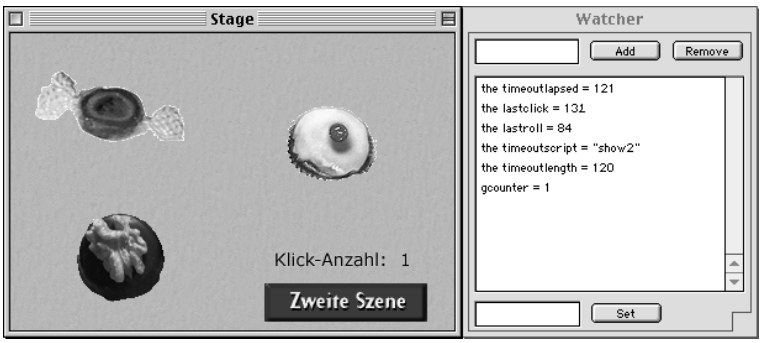


Abbildung 5.6: Im Watcher wollen wir bei der Durchführung des Experiments einige Ausdrücke verfolgen.

Durchführung

Drei Systemeigenschaften geben uns Auskunft darüber, wie lange der letzte Klick, die letzte Mausbewegung und die letzte Tastatur-Betätigung her sind: `the lastClick`, `the lastRoll` und `the lastKey`. Alle liefern die Zeit in Ticks, also 60stel Sekunden, zurück. Um beispielsweise nach 3 Sekunden der Inaktivität von Szene 2 in Szene 1 zurückzuwechseln, müssen wir uns fragen, welche Art von Inaktivität wir meinen. Geht es uns um die Mausbewegung, so genügt die Abfrage von `the lastRoll`, da diese Eigenschaft bei Klick ebenfalls auf 0 gesetzt wird:

```
on exitFrame me
  if the lastRoll > 3*60 then go "ErsteSzene"
end

on endSprite me
  puppetSound 1, 0
end
```

Dieses Skript haben wir auf Sprite 1 der zweiten Szene platziert. Es könnte allerdings auf jedem Sprite liegen. Beim Verlassen des Sprites wird auch ein eventuell noch spielender Sound in Kanal 1 ausgeschaltet.

Eine weitere Möglichkeit, die Zeit der Inaktivität zu messen, ist die Verwendung von Timeouts. Sobald wir die Systemeigenschaft `the timeoutLength` auf einen Zahlenwert (wiederum in Ticks) setzen, generiert Director so genannte `timeout`-Events; indem wir in einem Filmskript einen `on timeout`-Handler platzieren, können wir diese Events nutzen.

Versuchen Sie es einmal so:

```
on startMovie
  sprite(2).loc = point (-1000, -1000)
  the timeoutlength = 3*60 -- 3 Sekunden
end

on timeout
  put "timeout!"
  sprite(2).loc = point (260,89)
end
```

Ein zweiter Weg, der die oben vorgestellten „primary event handlers“ benutzt, funktioniert so:

```
on startMovie
  sprite(2).loc = point (-1000, -1000)
  the timeoutlength = 3*60 -- 3 Sekunden
  the timeoutscript = "show"
end

on show
  put "show!"
  sprite(2).loc = point (260,89)
end
```

Beide Skripte tun exakt dasselbe: Sie verstecken bei `startMovie` ein Sprite, und nach drei Sekunden der Inaktivität setzen sie es auf die Bühne und schreiben einen Kommentar ins Nachrichtenfenster. Die Eigenschaft `the timeoutLapsed` zeigt Ihnen an, wie viel Zeit seit dem letzten Mausklick bzw. der letzten Tastaturbetätigung vergangen ist. Mausbewegungen setzen die Timeout-Zeit nicht zurück.

In unserem Beispielfilm wollen wir nun Folgendes tun:

- Nach zwei Sekunden der Inaktivität soll ein verstecktes Sprite angezeigt werden.
- Danach soll nach weiteren zwei Sekunden Inaktivität der Sound „KlickenSie“ abgespielt werden und der Hinweis „Bitte klicken Sie auf ein Feld“ im Darsteller „News“ erscheinen.

Wir setzen demnach

```
the timeoutLength = 2 * 60
```

und benötigen einen Weg, wie wir zwei Aktionen mit dem `timeout`-Event auslösen können. Ein `on timeout`- Handler wäre hier recht unkomfortabel, besser kommen wir mit `the timeoutScript` zum Ziel:

```
global gcounter

on startMovie
  sprite(2).loc = point (-1000,-1000)
  member("Zaehlwerk").text = ""
```

```
  member("News").text = ""
  gcounter = 0
    Bei Start des Films werden die Anzeigen gelöscht und die
    Globale gcounter, die die Zahl der Klicks auf die Süßig-
    keiten-Sprites zählen soll, auf 0 gesetzt.
  the timeoutLength = 2*60
  the timeoutScript = "show"
    Das erste timeoutScript soll show sein.
end

on show
  sprite(2).loc = point (260,89)
  the timeoutscript = "show2"
    Das zweite timeoutScript soll show2 sein.
end

on show2
  if the frame < marker("ZweiteSzene") then
    Dieses Skript soll nur in Szene 1 ausgeführt werden.
    member("News").text = "Bitte klicken Sie auf ein Feld:"
    puppetsound 1, "KlickenSie"
    the timeoutScript = ""
    Dann wird das timeoutScript deaktiviert (durch Zuweisen
    von "").
  end if
end
```

Neben der Inaktivität des Users sollen auch seine Aktionen registriert werden. Klickt der User irgendwo auf eines der Süßigkeiten-Sprites, so werden seine Klicks gezählt. Die Klicks werden in einer Anzeige (Darsteller „Zaehlwerk“ in Sprite 6) sichtbar gemacht, und der Text im Darsteller „News“ wird gelöscht. Außerdem muss jeder Klick auch das `timeoutScript` wieder auf `"show"` setzen, damit zeitverzögert wieder die Aufforderung zum Klicken kommt. Das folgende Behavior liegt auf den Süßigkeiten-Sprites:

```
global gcounter

on mouseUp me
  gcounter = gcounter + 1
  member("Zaehlwerk").text = string(gcounter)
  member("News").text = ""
  the timeoutscript = "show"
end
```

5.3 Experiment 3: Mausaktivität

In diesem Beispiel wird die Aktivität des Users anhand seiner Rollovers über diverse Felder auf der Bühne gemessen. Zu einem bestimmten Zeitpunkt soll die Messung gestoppt und bewertet werden: Haben sich beispielsweise nach zehn Sekunden zehn Rollovers ereignet, wird als Kommentar „Ganz ordentlich“ in einem Textfeld ausgegeben.

Versuchsaufbau

Im Beispielfilm *Mausaktivitaet.dir* sind drei Bildelemente und drei Schriftelemente (Bitmap-Darsteller) vor einer Hintergrundfläche angeordnet: „UrlaubHintergrund“ (Sprite 1), „Wolke“ (Sprite 2), „Hotel“ (Sprite 3), „Touristen“ (Sprite 4), „SonneSchrift“ (Sprite 5), „MeerSchrift“ (Sprite 6), „LauneSchrift“ (Sprite 7). Ein kleiner Felddarsteller „Werte“ (Sprite 8) zeigt die gemessene Aktivität an. Die Schriftdarsteller reagieren auf Rollover mit einem Tausch auf einen Rollover-Darsteller.

Dieser Wechsel wird durch das Behavior „MemberTausch“ auf den Schrift-Elementen organisiert. Es definiert im `beginSprite`-Handler die Properties `pOriginal` (der auf der Bühne sichtbare Darsteller) und `pAustausch` (der bei einem Rollover sichtbare Darsteller); letzterer soll in der Besetzung im auf den Originaldarsteller folgenden „Slot“ liegen, also der Darsteller `member(pOriginal.number + 1)` sein. Bei `mouseEnter` und `mouseLeave` werden dem Sprite die entsprechenden Darsteller zugewiesen.

```
property pOriginal, pAustausch

on beginSprite me
  pOriginal = sprite(me.spriteNum).member
  pAustausch =  member(pOriginal.number + 1)
end

on mouseEnter me
  sprite(me.spriteNum).member = pAustausch
end

on mouseLeave me
  sprite(me.spriteNum).member = pOriginal
end
```

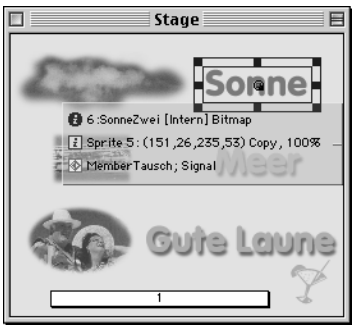


Abbildung 5.7:
Die Bühne des Films Mausaktivität.dir

Bilder- und Schriftelemente reagieren auf das Anklicken mit einem System-Ton (Behavior „Signal“):

```
on mouseup me
  beep
end
```

Durchführung

Unser Textfeld „Werte“ soll nun 10 Sekunden lang die Zahl der Rollovers anzeigen. Nach zehn Sekunden bewerten wir die Aktivität: Es wird zusätzlich ein Kommentar in das Textfeld geschrieben.

Da es sich hier um ein Skript handelt, das nicht speziell einem Sprite zugeordnet ist, könnten wir es im Skriptkanal des Drehbuchs (als Frame-Skript) oder als Filmskript realisieren. Wir haben uns für ein Filmskript entschieden, müssen dabei aber im Hinterkopf haben, dass Events wie `exitFrame` nur dann im Filmskript ankommen, wenn Sie unterwegs nicht aufgefangen werden. Wollen wir ein Stopper-Skript im Drehbuch platzieren, so sollte dies ein `pass` enthalten, damit das `exitFrame`-Event auch an das Filmskript weitergeleitet wird:

```
on exitFrame me
  go to the frame
  pass
end
```

Das Filmskript initialisiert in einem `startMovie`-Handler die verwendeten globalen Variablen und führt auf `exitFrame` die Überprüfung der Rollovers durch.

```
global gCounter, gLastro, gStarttime, gMachDenTest
global testtime
on startMovie
  gCounter = 0
  gStarttime = the ticks
  gMachDenTest  = TRUE
  testtime = 10*60 -- in Ticks, entspricht 10 Sekunden
end
```

```
on exitframe
  if (gMachDenTest) then
        Wenn der Test noch nicht gemacht wurde...
    if (rollover() <> gLastro) then
        ...und wenn sich das Rollover-Sprite geändert hat...
      gLastro = rollover()
      if (rollover() > 1 AND rollover() < 8) then
        ...und wenn auch noch das aktuelle Rollover eines der
        Sprites 2 bis 7 betrifft:
        gCounter = gCounter + 1
        member("Werte").text = string(gCounter)
        dann wird der Zähler gCounter um 1 erhöht und das
        Ergebnis als String dem Textdarsteller „Werte"
        zugewiesen.
      end if
    end if
    if the ticks > gStarttime + testtime then
        Wenn die zehn Sekunden vorbei sind...
      case true of
        ...schreibe den Zähler und einen Kommentar (abhängig
        von der Zahl der Rollovers) in den Ausgabedarsteller...
        (gCounter < 8): rueck = gCounter & ": ganz schön lahm"
        (gCounter < 15): rueck = gCounter & ": ganz ordentlich"
        otherwise:
          rueck = gCounter & ": Hektiker!"
      end case
      member("Werte").text = rueck
      gMachDenTest  = 0
        ...und setze gMachDenTest jetzt auf FALSE. Auf diese
        Weise wird die Untersuchung nicht wiederholt, da schon
        die erste Bedingung if (gMachDenTest) then nicht mehr
        zutrifft.
      gStarttime = 0
    end if
  end if
end
```

Falls Sie dasselbe Skript einmal als Frameskript verfassen wollen, versuchen Sie es bitte! (Kleiner Tipp: Statt globaler Variablen sollten Sie Property-Variablen verwenden!). Zur Überprüfung Ihrer Lösung finden Sie ein entsprechendes Skript im Film *Mausaktivitaet.dir* (Darsteller „FrameLoesung“).

Buttons

Director-Projekte sind meistens in einzelne „Szenen“ und einzelne Filme gegliedert, und zu Ihren Aufgaben bei der Gestaltung gehört es, Überleitungen zwischen Szenen/Einzelfilmen herzustellen und auf der Bühne Elemente anzuordnen und zu verwalten, über welche der Anwender zu den verschiedenen Schauplätzen kommt. Die Eingaben, die der User auf der Bildfläche macht, werden meistens von Buttons oder Schaltflächen entgegengenommen. Dass es sich bei den Bühnenelementen überhaupt um anklickbare Flächen handelt, weiß der Anwender nicht automatisch: Er ist es gewohnt, es in irgendeiner Weise signalisiert zu bekommen, zum Beispiel durch unterschiedliche Button-Zustände: einen Normal-, einen Hilite- und einen Down-Zustand.

Von der Gestaltung und Logik Ihres Angebots hängt es ab, wie gut und umfassend der User Ihre Produktion kennen lernt. In „Dust or Magic“ formuliert es Bob Hughes so: „Sobald eine Anwendung geöffnet ist, legen wir uns einige Theorien darüber zurecht, was hier los ist (was unbewusst geschieht). Wenn eine Abbildung auf dem Screen beim Anklicken als Trickfilm abläuft, erwarten wir das auch von allen anderen Abbildungen – jedenfalls solange sich die nicht-anklickbaren Abbildungen nicht anders verhalten.“

Bei einem Rollover verändert sich die Erscheinung des Buttons oder Cursors, beim Anklicken gibt es eine weitere Rückmeldung. Möglicherweise soll bei einem Szenenwechsel ein laufender Sound ausfaden oder der Szenenwechsel mit einem Übergangseffekt (Transition) versehen werden.

Es sind verschiedene Anforderungen, die auf Sie zukommen. Wenn Sie viele Buttons/Schaltflächen haben, ist die Anordnung nicht nur eine Frage der Optik auf dem Screen, sondern ein zunehmend komplexerer Skriptvorgang hinter den Kulissen. Das schließt auch alle Fragen der Navigation in Ihrer Anwendung ein: Wir werden in diesem Kapitel nicht nur Buttons per Skript zum Leben erwecken, wir wollen uns in den folgenden Kapiteln auch einige Konzepte vergegenwärtigen, wie Navigationsstrukturen in einem Director-Film aussehen können.

6.1 Alles kann ein Button sein ...

Die Wortbedeutung von „Button“ ist „Knopf“: ein Schalter auf dem Bildschirm, durch den der Anwender etwas erreichen kann. In Director können sehr viele Elemente als Buttons eingesetzt werden. Praktisch alle, die eine visuelle Repräsentation auf der Bühne haben: Bitmaps, Videos, Flash-Filme, Vectorshapes und Textdarsteller, um nur die wichtigsten zu nennen.

Gemeinsam dürfte all diesen Medientypen sein, dass ein Sprite auf der Bühne zu finden ist, auf das wir ein Behavior platzieren, das eine etwaige Bildänderung und eine Aktion auslöst. In Experiment 4 sehen Sie aber, dass es Medienformate gibt, die eigene Möglichkeiten bieten, eine Buttonoptik umzusetzen und sogar Aktionen festzulegen.

Wer sich nicht mit Lingo auseinander setzen möchte, findet einige fertige Button-Behaviors im Bereich „Steuerungen“ („Controls“) in der Behavior-Bibliothek. Da die Anpassung dieser Behaviors an eigene Anforderungen aber recht kompliziert ist, sollten Sie besser von den Beispielen in diesem Kapitel ausgehen und evtl. nötige Änderungen an ihnen vornehmen.

6.2 Experiment 1: Einfacher Sprungbutton

Versuchsaufbau

Der Film *EinfacherButton.dir* ist 260 x 220 Pixel groß. Damit darin zwischen verschiedenen „Szenen“ gewechselt werden kann, haben wir mehrere Markierungen angelegt und die folgenden Objekte im Drehbuch angeordnet.

1. „Hauptmenue“: Bilder 1 bis 10, Hintergrundfläche
2. „Erste Szene“: Bilder 11 bis 20, Hintergrundfläche
3. „Zweite Szene“: Bilder 21 bis 30, Hintergrundfläche
4. „Dritte Szene“: Bilder 31 bis 40, AppleBackground
5. „SchlussSzene“: Bilder 41 bis 50, SchlussHintergrund

Auf Bild 1, 11, 21, 31 und 41 platzieren Sie jeweils einen Marker „Hauptmenue“, „Erste Szene“, „Zweite Szene“, „DritteSzene“ und „SchlussSzene“. Sie klicken im Marker-Kanal in der Höhe des betreffenden Frames: automatisch wird eine Markierung gesetzt, der Sie sofort einen Namen geben sollten.

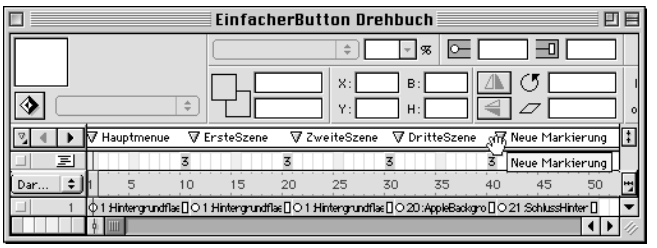


Abbildung 6.1: Das Setzen einer Markierung im Marker-Kanal

Überflüssige Markierungen packen und ziehen Sie ein Stück nach unten aus dem Marker-Kanal. Beim Loslassen verschwindet die Markierung.

Das Markierungsmenü sieht dann so aus wie in Abbildung 6.2. In den vierten Sprite-Kanal des Hauptmenü-Abschnitts legen wir die Schaltfläche „StartButton“. In die Besetzung fügen wir nach dem Darsteller „StartButton“ eine Rollover-Version des Button-Darstellers ein. In den Drehbuchabschnitt der ersten Szene legen wir in Sprite-Kanal 2 den Darsteller „Gurke“, der als Klick-Fläche dienen wird, und einen Textdarsteller („KlickHinweis“), der die Szenenwechsel-Prozedur erläutert: „Wenn Sie nicht auf die Gurke klicken, wandert der Abspielkopf nach 8 Sekunden weiter.“. Auf die zweite Szene kornmt der Darsteller „Orange“ als weiteres Anklick-Objekt. Die Szene ergänzt wieder ein Textdarsteller zur Erläuterung des Vorgangs: „Von hier geht es automatisch nach 3 Sekunden weiter.“

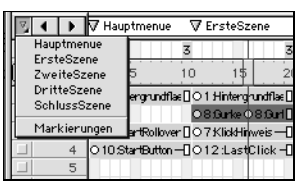


Abbildung 6.2: Ein kleines Aufklappmenü zeigt alle Markierungen.

Wenn Sie nicht-rechteckige Objekte als Anklick-Objekte verwenden wollen, klicken Sie mit gedrückter Strg / ⌘-Taste den Darsteller auf der Bühne an und wählen Sie den Effekt „Matt“ aus. Mausklicks werden bei diesem Inkeffekt nicht im gesamten Rechtecksbereich des Sprites registriert, sondern nur innerhalb des abgebildeten Objekts.

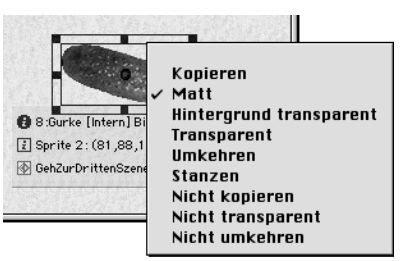


Abbildung 6.3: Der Ink-Effekt „Matt“ hat für Buttons eine wichtige Funktion.

Im Skript-Kanal plazieren Sie auf Bild 10, 20, 30, 40 und 50 jeweils das „Stopper-Skript“:

```
on exitFrame me
  go to the frame
end
```

Der Aufbau des Gesamtfilms sieht nun so aus wie in Abbildung 6.4.

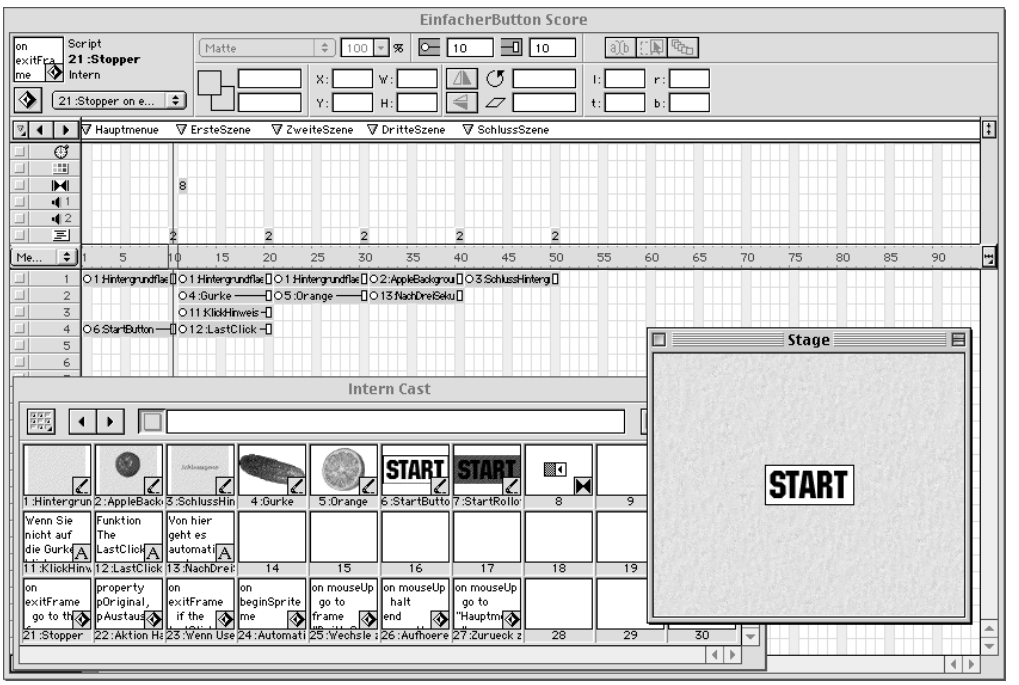


***Abbildung 6.4: Drehbuch, Bühne und Besetzung des Filmes** EinfacherButton.dir*

Durchführung

Im Beispielfilm *EinfacherButton.dir* wollen wir im Bereich „Hauptmenue“ einen „Start“-Button mit einfachem Rollover einrichten. Vom „Hauptmenue“ aus geht es durch Anklicken des Buttons weiter zu drei Folgeszenen und einer Schluss-Szene. Auf der ersten Szene kann der nächste Szenenwechsel ein weiteres Mal durch den Klick auf eine Schaltfläche herbeigeführt werden (nach Szene 3); sollte der User aber nichts tun, findet nach acht Sekunden ein automatischer Szenenwechsel statt (nach Szene 2). Auf Szene 3 angekommen, geht es nach drei Sekunden automatisch zur Schluss-Szene weiter. Szenenwechsel von Hand und automatisierter Szenenwechsel werden also variiert. Abbildung 6.5 zeigt die Szenen-Verzweigung, in einem Flow-Chart-Programm skizziert.

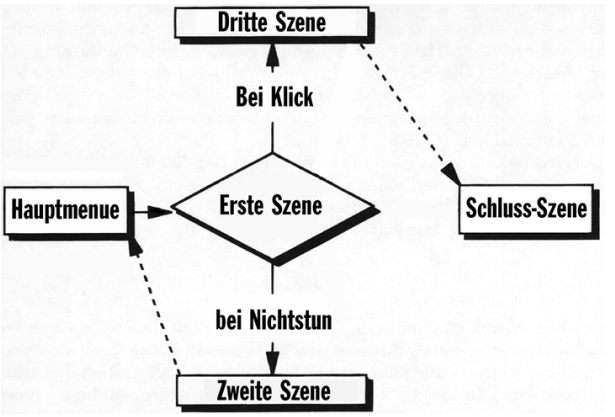


Abbildung 6.5: Die Navigationsstruktur des Films EinfacherButton.dir

Die Funktionalität eines solchen Szenenwechsels ist einfach zu erzeugen. Wir statten zunächst die Sprites mit Behaviors für den Szenenwechsel aus. Drei Behaviors regeln den Szenenwechsel für den Fall, dass der User klickt – im Hauptmenü („Aktion Hauptmenue“; Sprite 4):

```
on mouseUp
  go to "ErsteSzene"
end mouseUp
```

in der ersten Szene („Wechsle zur dritten Szene“; Sprite 2)

```
on mouseUp
  go to "DritteSzene"
end mouseUp
```

und in der zweiten Szene („Zurueck zum Menue“; Sprite 2):

```
on mouseUp
  go to "Hauptmenue"
end mouseUp
```

Zwei weitere Behaviors regeln den Szenenwechsel automatisch – in der zweiten Szene („Wenn User nichts tut“; Sprite 1)

```
on exitFrame
  if the lastClick > 8*60 then go to "ZweiteSzene"
  else go to the frame
end
```

und in der dritten Szene („Automatischer Szenenwechsel“; Sprite 1):

```
on beginSprite me
  startTimer
end

on exitFrame me
  if the timer < 3*60 then go to the frame
  else go to "SchlussSzene"
end
```

Im ersten Fall überprüft die Funktion `the lastClick` , ob der letzte Klick länger als 8 Sekunden zurückliegt, und veranlasst dann gegebenenfalls den Szenenwechsel. Statt `the lastClick` könnten Sie auch `the lastEvent` einsetzen, dann würden allerdings auch Mausbewegungen/Tastatureingaben den Wechsel verzögern.

Im zweiten Fall wird nach 3 Sekunden die Szene gewechselt, da bei Beginn ein Timer gestartet wird, der nach dem Verstreichen von drei Sekunden den Szenenwechsel veranlasst.

Der Szenenwechsel ist jetzt in allen Teilen komplett: Die Darsteller sind im Drehbuch entsprechend angeordnet, Marker und Stopper-Skript regeln Szenenbeginn und Ende. Die Behaviors für den anwendergesteuerten und den automatischen Szenenwechsel sind an ihrem Platz.

Der „Start“-Button im „Hauptmenue“ reagiert allerdings noch nicht auf ein Rollover. Damit der Schrift-Hintergrund beim Überfahren mit der Maus rot aufleuchtet, wird im Behavior „Aktion Hauptmenue“ ein Darstellerwechsel ergänzt:

```
property pOriginal, pAustausch

on beginSprite me
  pOriginal = sprite(me.spriteNum).member
  pAustausch =  member(pOriginal.number + 1)
end

on mouseEnter me
  sprite(me.spriteNum).member = pAustausch
  Befindet sich die Maus über dem Sprite, wird ein Tausch
  des Darstellers gegen den in der Besetzung nachfolgenden
  Darsteller herbeigeführt.
end

on mouseLeave me
  sprite(me.spriteNum).member = pOriginal
  Verlässt der User den Darsteller, wird der Tausch rückgängig
  gemacht.
end
```

Beim Rollover der Maus über den Darsteller sieht es jetzt so aus, als würde der Schrifthintergrund des Buttons rot aufleuchten. Das Skript ist so angelegt, dass der Austauschdarsteller in der Besetzung direkt im Anschluss an den Originaldarsteller liegen muss: Der `beginSprite`-Handler kann dann einfach `pAustausch = member(pOriginal.number + 1)` setzen, also die (interne) Darstellernummer des Originaldarstellers plus 1. (In Director 8 und neuer müssen Sie in die Normalansicht der Besetzung schalten, um Darsteller neu anordnen zu können. In der Listenansicht ist dies nicht möglich).

Kleiner Exkurs: Transitions

Beim Drücken des „Start"-Buttons im fertigen Beispielfilm hat man den Eindruck, als würde das Hauptmenü nach links weggeschoben und der Blick auf die „Erste Szene" darunter freigegeben. Es wurde hier einer von Directors Übergängen eingesetzt. Sie können aus einer ganzen Reihe solcher Übergänge (Transitions) wählen und damit einen Szenenwechsel besonders akzentuieren. Übergänge richten Sie im Übergangskanal des Drehbuchs ein (vgl. Abbildung 6.6).

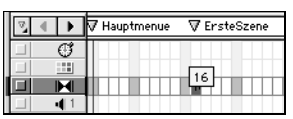


Abbildung 6.6:
Der Übergangskanal
(hier dunkel hervorgehoben)

Wenn der Übergangskanal nicht zu sehen ist, betätigen Sie eine kleine Taste am rechten Drehbuchrand auf der Höhe des Markerkanals (vgl. Abbildung 6.7).

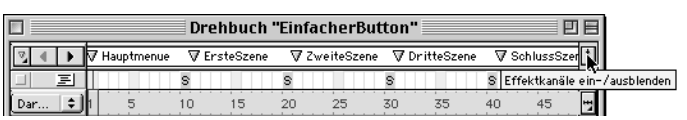


Abbildung 6.7: Taste zum Einblenden/Ausblenden der Effektkanäle

Im Übergangskanal können Sie sowohl Übergänge für die ganze Fläche der Bühne als auch nur für Elemente einer Szene regeln. Ist ein Übergang zwischen zwei Szenen geplant, platzieren Sie die Anweisung dafür im ersten Bild der zweiten Szene. Wählen Sie zunächst das entsprechende Bild im Übergangskanal aus. Ein Doppelklick auf das Bild öffnet den Dialog „Bildeigenschaften: Übergang" (oder: Menü Modifizieren/Bild/Übergang).

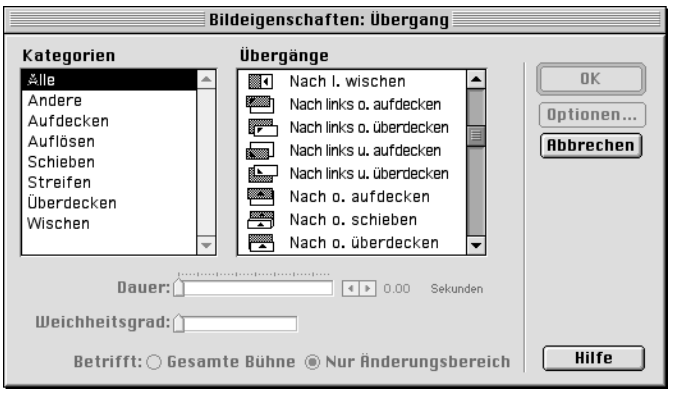


Abbildung 6.8: Ein Doppelklick auf einen Frame im Übergangskanal öffnet den Übergang-Dialog.

Wählen Sie die gewünschte Dauer und Art des Übergangs (z.B. die Richtung, in welche verschoben, das Muster, nach welchem aufgelöst werden soll). Bei Ihrer Übergangsauswahl müssen Sie allerdings beachten, dass manche auflösende Transitions (pixel- bzw. bitweises Auflösen) bei 32-Bit-Farbtiefe nicht funktionieren.

Übrigens: Auch wenn Sie Transitions per Lingo aufrufen wollen, ist es ein empfehlenswerter Weg, wie oben gezeigt einen Transition-Darsteller anzulegen und diesen dann im Skript zu verwenden. Benennen Sie dafür den im Übergangskanal generierten Transition-Darsteller (er wird automatisch in der gerade aktiven Besetzung angelegt); und rufen Sie die Transition im Skript so auf:

```
puppetTransition (member("trans17"))
```

Diskussion

Für jede Aktion ein eigenes Skript: Sie ahnen bereits, dass die Vorgehensweise in diesem Abschnitt noch nicht der Weisheit letzter Schluss ist. Wir werden in diesem und dem folgenden Kapitel noch Mittel kennen lernen, mit denen Sie komplexere Buttonskripte verfertigen, die dann auch mehrere Aktionen an unterschiedlichen Stellen im Drehbuch übernehmen können.

Vor der Generalisierung steht allerdings häufig genau der hier verfolgte Ansatz: das schnelle und unkomplizierte Umsetzen der gewünschten Navigationsstruktur.

6.3 Experiment 2: Ein Vectorshape als Button

Versuchsaufbau

Mit Strg / ⌘+⇧+V öffnen wir das Vektorform-Fenster und skizzieren mit dem Stift einen Pfad, der durch Ankreuzen der Checkbox „Gesch“ geschlossen und durch Ziehen an den Anfassern manipuliert werden kann. Selbstverständlich können Sie auch eine der geometrischen Grundformen zum Ausgangspunkt nehmen. Legen Sie einen möglichst unregelmäßig geformten Button an. Mit dem PopUpMenü der Taste „Strichbreite“ unter der Werkzeug-Palette im Vektorform-Fenster kann eine Kontur für die gezeichnete Form ausgewählt werden.

Die Vektorform wird direkt auf der grau eingefärbten Bühne platziert. Um der Bühne einen Farbton zu unterlegen, öffnen wir mit Strg / ⌘+⇧+D den „Filmeigenschaften“-Dialog und wählen unter „Bühnenfarbe“ einen passenden Farbton.

Den Director-Film finden Sie als *VectorButton.dir* auf der CD-ROM.

Durchführung

Wir weisen dem Vectorshape-Sprite den Ink-Effekt „Transparent“ (oder „Hintergrund Transparent“) zu. Mit folgendem Skript auf dem Sprite wollen wir zunächst testen, wo genau der aktive Bereich des Shapes ist:

```
on mouseWithin me
  beep
end
```

Bewegen Sie dann die Maus auf der Bühne an den Button heran: Das Skript spielt den System-Beep genau dann ab, wenn sich die Maus über dem opaken, also undurchsichtigen Bereich des Vectorshapes befindet.

Ändern Sie den Inkeffekt zu „Copy“ (oder einem beliebigen anderen Inkeffekt außer den beiden schon genannten), so ist das umschließende Sprite-Rechteck der „aktive Bereich“. Der Inkeffekt „Transparent“ ist für uns daher der weitaus spannendere.

Mit Vectorshapes lassen sich auch sehr effektvolle Rollover-Effekte generieren. Bereiten Sie Ihren Vectorshape-Darsteller dahingehend vor, dass Sie das kleine Verlauf-Symbol anklicken und mit den Interface-Elementen am Kopf des Vectorshape-Fensters einen hübschen Verlauf festlegen (vgl. Abbildung 6.9). Wenn Sie fertig sind, stellen Sie den Darsteller wieder um auf die Füllungs-Einstellung „Solid“.

Abbildung 6.9: Die Verlaufseinstellungen im Vectorshape-Editor finden Sie am oberen Rand.

Das folgende Skript sorgt dafür, dass eine Mausbewegung auf den Button den Darsteller von der „soliden“ zur Verlaufsfüllung umschaltet. Bei `mouseDown` wird zudem eine Animation gestartet, die auf `prepareFrame` ausgeführt wird: Die Anzahl der `fillCycles`, also der Wiederholungen unseres Verlaufs, wird kontinuierlich von 1 bis 7 hochgezählt. (Der Operator `mod` – Modulo – liefert in der Zeile

```
ergebnis = wert mod 7
```

den Rest bei Teilung durch 7 zurück (also je nach `wert` Zahlenwerte im Bereich von 0 bis 6).

```
property animate
on mouseEnter me
  member("Schaltflaeche").fillmode = #gradient
end

on mouseleave me
  member("Schaltflaeche").fillmode = #solid
end

on mouseDown me
  animate = 1
end
```

```
on mouseUp me
  animate = 0
end

on mouseUpOutside me
  animate = 0
end

on prepareFrame me
  if animate then
    member("Schaltflaeche").fillcycles = \
     (member("Schaltflaeche").fillcycles ) mod 7 + 1
  end if
end
```

Eine Besonderheit wird Ihnen auffallen, wenn Sie denselben Darsteller nochmals auf der Bühne liegen haben: Da das Skript eine Darstelleränderung bewirkt, sind alle Instanzen des Darstellers betroffen – auch solche, von denen die Maus weit entfernt ist. Das Skript eignet sich also nur, wenn für jeden Button ein eigener Vectorshape-Darsteller angelegt wird.

Diskussion

Vectorshapes bieten als Buttons vor allem einen Vorteil: Sie können (fast) beliebige Formen annehmen (ab Director 8 auch mehrere voneinander unabhängige Formen in einem Darsteller), und der Klickbereich (bzw. der aktive Bereich des Buttons, der auf `mouseEnter` etc. reagiert) ist entsprechend variabel, wenn Sie einen der „Transparent"-Ink-Effekte zuweisen.

Die Nachteile sind indes nicht von der Hand zu weisen: Die Erstellung von Vectorshapes ist umständlich, da sie nicht direkt auf der Bühne gezeichnet werden können. Vectorshapes bieten zudem nur geringe Gestaltungsmöglichkeiten und sind häufig nicht in ein Screendesign einzupassen. Als unsichtbare Hotspot-Bereiche (mit `sprite(s).blend = 0`) sind sie aber auch dann noch nützlich.

Eine weitere Besonderheit ist interessant: Mit Vectorshapes können Sie auch linienförmige Buttons anlegen. Stellen Sie dafür die Liniendicke beispielsweise auf den Maximalwert (12 Punkt) und den Füllmodus auf ungefüllt. Weisen Sie wieder das einfache Beep-Behavior zu. Nur die Linie ist nun aktiv, da sie der opake Bereich des Buttons ist.

6.4 Experiment 3: Eine Clickmap erstellen

Die folgende Buttonform erfordert einige Vorarbeit, aber sie bietet auch eine spannende Funktionalität: Millionen von Hotspots sind damit (theoretisch) auf einem Darsteller realisierbar. Die Idee ist einfach: Zu einem Darsteller, den wir auf der Bühne anzeigen, existiert in der Besetzung ein weiterer, der spezielle Farbinformationen enthält. Jeder Farbwert entspricht dabei einem Hotspot.

Damit die Erstellung dieser Clickmap einfacher ist, beschränken wir uns hier auf 256 Farben: Bei Bitmaps in 8 Bit Farbtiefe (das sind 256 Farben) ist es ja bekanntlich so, dass jeder Farbe ein eindeutiger Indexwert zugewiesen ist. Sie werden das beim Versuchsaufbau sehen (vgl. Abbildung 6.13).

Die Einsatzbereiche für eine Clickmap sind mannigfaltig: Überall dort, wo Buttons in großer Zahl und mit möglicherweise unregelmäßigen und unzusammenhängenden Formen vonnöten sind, ist es sinnvoll, sie einzusetzen. Aber auch ganz andere Einsatzzwecke sind denkbar: Wenn Sie ein Sprite auf der Bühne frei bewegen wollen und beispielsweise die Größe von der Position auf der Bühne abhängt und vielleicht auch noch bestimmte Bereiche der Bühne gesperrt sind, ist diese Anforderung mit einer „Tiefen-“ oder „Höhen-Map“ ähnlich der hier vorgestellten Clickmap lösbar.

Bitte beachten Sie, dass das Experiment mit einer Imaging-Lingo-Funktion (`getPixel()`) arbeitet. Imaging Lingo wurde offiziell mit Director 8 eingeführt, hat mit Einschränkungen aber auch in Director 7 bereits funktioniert. Die Anleitung hier bezieht sich nur auf Director 8 oder neuer. Wollen Sie das Beispiel in Director 7 erstellen, informieren Sie sich bitte unter *http://www.directorworkshop.de/workshop/jowork1.html* über die Director 7-Spezifika.

Versuchsaufbau

Im Film *clickmap.dir* reagieren die einzelnen Zitronen-Formen unterschiedlich auf Rollovers. Um sie dafür empfangsbereit zu machen, ist ihnen eine Clickmap hinterlegt worden. In der Clickmap definieren Sie die anklickbaren Felder in einem dem Ausgangsdarsteller in der Besetzung folgenden 8-Bit-Darsteller. So gehen Sie vor:

Legen Sie mit Strg / ⌘+5 einen Bitmap-Darsteller an, den Sie beispielsweise „Colormap“ benennen und in der Besetzung (in der Normalansicht) an die Position ziehen, die direkt auf den Darsteller „Zitronen-Papier“ folgt.

Abbildung 6.10:
Stellen Sie die Farbtiefe des Bitmapdarstellers auf 8 Bit.

Blenden Sie den Ausgangsdarsteller im Hintergrund mit Hilfe der Zwiebelschichten (Menü Ansicht/Zwiebelschichten-Effekt) ein.

Abbildung 6.11: Mit der links hervorgehobenen Taste aktivieren Sie die Zwiebelschichten-Funktion.

Wählen Sie nun deutlich unterscheidbare Farben aus der aktuellen Palette und zeichnen Sie die gewünschten Bereiche – Ihre Hotspots – als Farbflächen nach. Notieren Sie sich beim Einzeichnen der Klickfelder die Farb-Indizes (vgl. Abbildung 6.13), denn diese Werte werden Sie gleich im Skript verwenden.

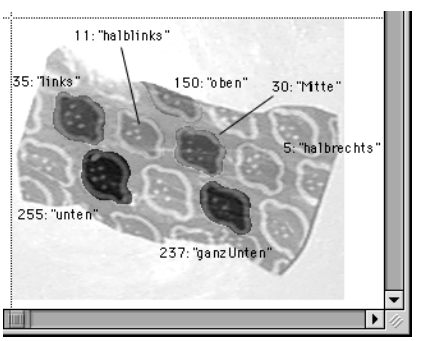


Abbildung 6.12: Durch den Zwiebelschichten-Effekt sehen Sie das Bonbonpapier durchscheinen und die aufgetragenen Klickfelder. Außerdem sind die später benötigten Farbwerte und Bezeichnungen eingeblendet.

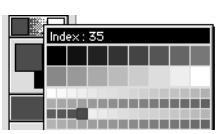


Abbildung 6.13:
Die ausgewählte Farbe hat den Indexwert 35.

Zeichnen Sie zum Schluss ein Rechteck, das genauso groß ist wie der Ausgangsdarsteller, als Rand Ihrer Clickmap ein, so dass die beiden Darsteller dasselbe interne Koordinatensystem haben (Abbildung 6.14). Das Rechteck muss eine Farbe haben, die im „Colormap“-Darsteller noch nicht vorkommt.

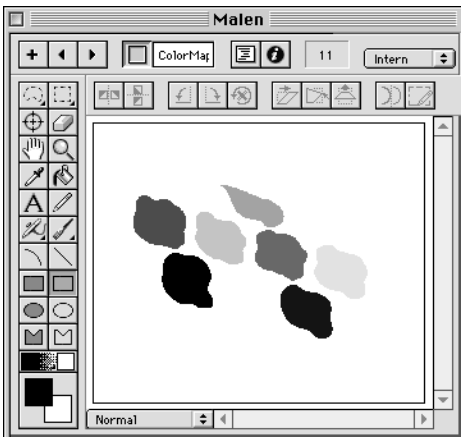


Abbildung 6.14: Im Malfenster versehen Sie Ihre Clickmap mit einem schwarzen Rand, damit die Bildkoordinaten von Ausgangsdarsteller und Clickmap identisch ausfallen.

Durchführung

Im dazugehörigen „ClickmapSkript“, das auf Sprite 1 platziert wird, kommen folgende Variablen zum Einsatz:

`visMember` ist der auf der Bühne sichtbare Darsteller.

`Farbkarte` enthält das Image des auf `visMember` in der Besetzung folgenden Darstellers.

`Farbliste` ist eine Propertyliste, die als Propertybezeichnung die Integer-Farbwerte, als Property-Wert den jeweils anzuzeigenden Text enthält.

`Farbkarte` und `Farbliste` werden als Property-Variablen definiert, da wir im `mouseWithin`-Handler wieder auf sie zugreifen wollen. Die anderen Variablen sind lokale Variablen, die nur im Handler, in dem sie benutzt werden, definiert sind.

`DarstellerOrt` erhalten wir, indem wir mit `mapStageToMember()` die Mausposition auf der Bühne in eine Darstellerkoordinate umrechnen.

Der Variablen `Farbauswahl` wird mit der Imaging-Lingo-Funktion `getPixel()` der Farbwert an der Position `Darstellerort` im Darsteller zugewiesen.

Mit Listensyntax greifen wir dann auf `Farbliste` zu und versuchen, den zum Farbwert passenden Eintrag zu finden. Ist er nicht vorhanden, so wird `VOID` zurückgeliefert: Wir setzen das Ergebnistextfeld auf "". Wenn ein Ergebniswert existiert (`if NOT(voidP(Auswahltext)) then` ...), so wird er im Darsteller „Anzeige“ angezeigt.

```
property Farbkarte, Farbliste

on beginsprite me
  vismember = sprite(me.spritenum).member
  Farbkarte = member(vismember.number +1).image
  Farbliste = [35: "links", 11: "halblinks", 30: "Mitte", \
   5: "halbrechts", 129: "unten", 237: "ganzUnten",150: "oben"]
  member("Anzeige").text = ""
end

on mousewithin me
  DarstellerOrt = \
   sprite(me.spritenum).mapStageToMember(the mouseloc)
  Umrechnung Bühnenposition in Darstellerkoordinate
  if voidP(DarstellerOrt) then return
  Diese Zeile bricht die Bearbeitung ab (return), falls
  DarstellerOrt VOID ist. VOID wird zurückgegeben, wenn die
  Maus auf die "Nulllinie" (point(0,n) bzw. point (n,0) des
  Darstellers bewegt wird.
  Farbauswahl = Farbkarte.getpixel(DarstellerOrt, #integer)
  Abfrage des Farbwertes im Image Farbkarte an der Position
  DarstellerOrt.  #integer ist ein optionaler Parameterwert,
  der dafür sorgt, dass nicht ein Farbobjekt (palette-
  Index(30)), sondern der entsprechende Integerwert (30)
  zurückgeliefert wird.
  Auswahltext = Farbliste.getaprop(Farbauswahl)
  Im Gegensatz zu getProp() verursacht getaprop() keine Fehler-
  meldung, wenn der gesuchte Wert nicht in der Propertyliste
  vorhanden ist. Enthält Farbauswahl beispielweise die Hinter-
  grund- oder Randfarbe, so erhält Auswahltext den Wert VOID.
  if NOT(voidP(Auswahltext)) then
    member("Anzeige").text = Auswahltext
  else member("Anzeige").text = ""
end
```

In der Zeile

```
DarstellerOrt = \
 sprite(me.spritenum).mapStageToMember(the mouseloc)
```

werden die Mauskoordinaten in Koordinaten im Bezugssystem des Darstellers umgerechnet. Nur wenn der Darsteller unskaliert ist und die ganze Bühne ausfüllt, entsprechen sich Darsteller-Koordinaten und Bühnenkoordinaten, ansonsten nicht.

Vergegenwärtigen Sie sich die Differenz aus Bühnen- und Darstellerkoordinaten in Abbildung 6.15. `mapStageToMember()` kann auch Darstellerkoordinaten errechnen, wenn das Sprite skaliert, rotiert oder anderweitig verzerrt ist.

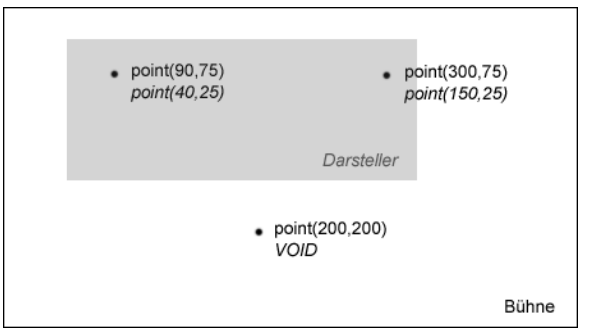


Abbildung 6.15: Bühnenkoordinaten (oberer Wert) und Darstellerkoordinaten (unterer, kursiver Wert), wie mit `mapStageToMember()` errechnet – hier für einen unskalierten und unverzerrten Darsteller

6.5 Experiment 4: Flash-Buttons

Den Director-Film zu diesem Experiment finden Sie als *Flashbutton.dir* auf der CD-ROM. Außerdem benötigen Sie Flash 5 oder neuer; eine entsprechende Trialversion finden Sie im Verzeichnis *Demos*.

Flash-Buttons zur Verwendung in Director können Sie in Flash 3 oder neueren Versionen erstellen. Unser Experiment zeigt den Weg in Flash 5.

Flash bietet einen eigenen Button-Editor, der komfortabel das Anlegen der unterschiedlichen Button-Zustände erlaubt (vgl. Abbildung 6.16). Mit der Flash-Aktion `GetURL` können Sie direkt Funktionen im Directorfilm aufrufen, wenn Sie eine wichtige Einschränkung (s.u.) beachten.

Für erste Tests sind die Buttons unter Fenster/Allgemeine Bibliotheken/Schaltflächen gut geeignet.

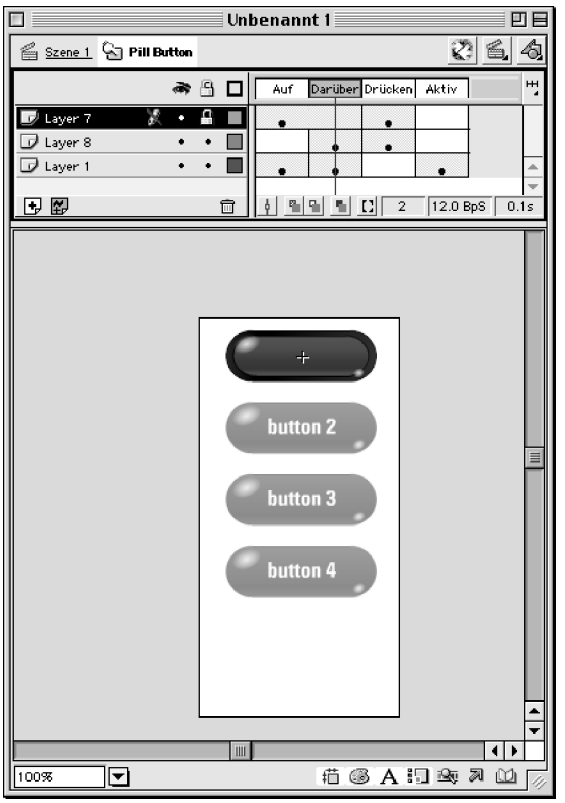


Abbildung 6.16: Button-Symbole in Flash 5: Aktiviert ist der oberste Button; sein „Over"-Zustand ist gerade sichtbar.

Versuchsaufbau

Legen Sie zunächst in Flash 5 einen Film mit vier Buttons an. Wir haben (im Beispiel-Flashfilm *Flashbut.fla*) aus der Bibliothek „Schaltflächen" den „Pill Button" benutzt, über den wir Textfelder mit den Beschriftungen „button 1" bis „button 4" gelegt haben.

Mit Rechts-/Ctrl-Klick auf den Button wählen Sie die Menüoption Aktionen aus und weisen den Buttons in Flash bereits Aktionen zu (vgl. Abbildung 6.17).

Die flexibleste Form der Kommunikation mit Director ist, die Aktion `GetURL` mit dem Protokoll `event:` zu verwenden. Der Actionscript-Text auf dem ersten Button ist dann beispielweise:

```
on (release) {
    getURL ("event:SetUpGo \"button1\"");
}
```

Er ruft die Funktion `setupGo` mit dem Parameter "`button1`" auf. Die anderen Buttons sollten entsprechend die Parameter "`button2`" bis "`button4`" enthalten.

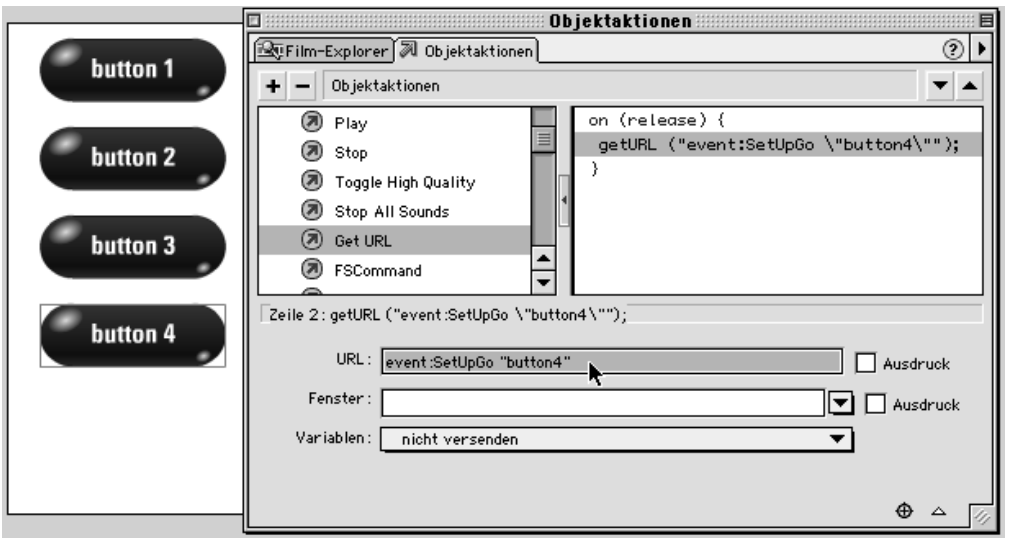


Abbildung 6.17: In Flash nutzen Sie die Aktion `GetURL` und geben unter URL `event:` oder `lingo:` als Protokoll an.

Exportieren Sie Ihren Flash-Film als *.swf*-Datei (DATEI/VERÖFFENTLICHEN): Damit schaffen Sie die Voraussetzung, dass der Film in Director importiert werden kann.

Legen Sie einen weiteren Flash-Film an für einen Button mit der Beschriftung „zurück“, und weisen Sie ihm die gleiche Aktion mit dem Parameter „button5“ zu.

Unsere Director-Datei *Flashbutton.dir* ist in Director 7 erstellt. Das funktioniert auch mit in Flash 5 erstellten SWFs, solange Sie keine speziellen Flash-5-Features verwenden. Die Faustregel lautet hier: Wenn Sie Ihren Flash-Film ohne Funktionalitätsverlust im Flash 3-Format exportieren können, so wird er auch in Director 7 funktionieren. Director 8 unterstützt die Flash-4-Erweiterungen, Director 8.5 kommt auch mit Flash 5-Features zurecht. Eine entsprechende Erweiterung für Flash MX ist angekündigt; informieren Sie sich unter *http://www.macromedia.com/support/director/downloads.html*.

Importieren Sie die *swf*-Datei (*flashbut.swf* und *zurueck.swf*) als Darsteller in den Director-Film, und ziehen Sie die beiden an unterschiedliche Stellen ins Drehbuch (vgl. Abbildung 6.18). Um den Hintergrund der Flash-Sprites transparent zu machen, weisen Sie den Ink-Effekt „Hintergrund transparent“ zu. Im Skriptkanal ist zudem durchgängig ein `go the frame`-Stopperskript platziert.

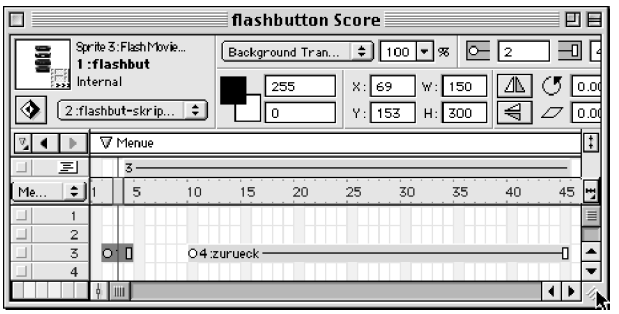


Abbildung 6.18: Das Drehbuch zum Film Flashbutton.dir

Durchführung

Beide Flash-Darsteller rufen Aktionen im Director-Film mit

```
event:setupGo "einParameter"
```

in der `GetURL`-Aktion auf. Ein solches Event können wir mit einem Behavior auf dem Flash-Sprite auffangen und uns – entsprechend dem übergebenen Parameter – verhalten:

```
property frameToGo
on setupGo me, param
  case param of
    "button1": frameToGo = 11
    "button2": frameToGo = 21
    "button3": frameToGo = 31
    "button4": frameToGo = 41
    otherwise
      frameToGo = "Menue"
  end case
end

on exitFrame me
  if not ( voidP(frameToGo) ) then
    go to frameToGo
  end if
end
```

Der `setupGo`-Handler setzt nur eine Property, er veranlasst nicht direkt einen Sprung auf einen anderen Frame. Dieses Vorgehen ist immer dann notwendig, wenn das aufrufende Flash-Sprite nicht durchgängig im Drehbuch liegt bzw. wenn ein anderer Director-Film aufgerufen werden soll.

Der `exitFrame`-Handler überprüft, ob `frameToGo` gesetzt ist (`not(voidP(frameToGo))`), und springt auf den entsprechenden Frame, wenn dem so ist. Da das Sprite nicht durchgängig im Drehbuch liegt, ist `frameToGo` bei einem erneuten Aufruf des Menü-Frames wieder `VOID`.

Modifikation

So schön das bereits funktioniert mit den in Flash gefertigten Buttons – es erscheint doch umständlich, die Buttonbeschriftungen jeweils in Flash zu ändern. Das folgende Experiment nutzt nur noch einen einzigen Flash-Button und die Director-Funktion `setVariable()`, um die Beschriftung des Buttons zu ändern.

Dafür legen wir einen neuen Flash-Film an (*FlashbutDyn.fla*), bei dem das Textfeld als „Dynamischer Text“ mit dem Variablennamen „but“ definiert wird (vgl. Abbildung 6.19). Das Button-Skript in Flash ist ebenfalls einfacher geworden, da wir einen größeren Teil der Logik im Director-Film skripten wollen:

```
on (release) {
    getURL ("event:SetUpGo");
}
```

Abbildung 6.19: Die Button-Beschriftung wird nun als „Dynamischer Text“ definiert.

Der Flash-Darsteller muss dann für das Menü mit 4 Buttons viermal auf der Bühne platziert werden, ebenso können wir ihn für den „Zurück“-Button nutzen. Das folgende Behavior setzt die gewünschte Funktionalität um. Es erwartet die Menü-Buttons in den Sprites 3 bis 6, den Zurück-Button im Sprite 7.

```
property frameToGo

on beginSprite me
  msp = sprite(me.spriteNum)
  case me.spriteNum of
    3: msp.setVariable("but", "Erster")
    4: msp.setVariable("but", "Zweiter")
    5: msp.setVariable("but", "Dritter")
    6: msp.setVariable("but", "Vierter")
    7: msp.setVariable("but", "zur¸ck")
  end case
end

on setupGo me
  case me.spriteNum of
    3: frameToGo = 11
    4: frametoGo = 21
    5: frameToGo = 31
```

```
      6: frameToGo = 41
      7: frameToGo = "Menue"
    end case
  end

  on exitFrame me
    if not ( voidP(frameToGo) ) then
      if frameToGo <> "Menue" then puppettransition mem-
  ber("pushleft")
      else puppettransition member("pushright")
      go to frameToGo
    end if
  end
```

Nun werden im `beginSprite`-Handler aus Director heraus die Buttonbeschriftungen im eingebetteten Flash-Film gesetzt – und zwar abhängig von der Sprite-Nummer. Auch die `setupGo`-Aktion führt abhängig von der Nummer des Sprites, das den Event erzeugt, unterschiedliche Aktionen aus.

Den Director-Film finden Sie als *FlashButtonDyn.dir* auf der CD-ROM.

Mit den Umlauten hat die Flash-Integration in Director einige Probleme, deshalb die „falsche“ Schreibung von „zurück“ im Skript. Falls Sie bei Ihren Versuchen auf ähnliche Hindernisse stoßen, so finden Sie im Film *FlashbuttonDyn.dir* die Ersetzungen für die deutschen Umlaute und „ß“.

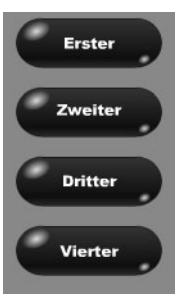


Abbildung 6.20:
Die Button-Beschriftungen im Flash-Film wurden per Skript zugewiesen.

Diskussion

Die Verwendung von Flash-Filmen (SWFs) innerhalb Director bietet einen großen Vorteil: Es lassen sich komplexe Bildänderungen und Animationen „kapseln“, so dass der in Director nötige Aufwand minimiert wird: Hier verwalten Sie nur noch *einen* Darsteller und *ein* Sprite.

Gleichzeitig ist die Einbindung so flexibel, dass Sie – wie in unserem modifizierten Beispiel – „Vorlagen“ erstellen können, die Sie mit Lingo-Skripten für unterschiedliche Anwendungszwecke anpassen können.

Zur Zwei-Wege-Kommunikation zwischen Director und SWF-Sprite (ab Director 7.02 / Flash 4) dienen die Befehle

- `sprite(flashsprite).setVariable(PfadZurFlashVariable, Wert)`
- `sprite(flashsprite).getVariable(PfadZurFlashVariable)`

sowie ab Director 8/Flash 5:

- `sprite(flashsprite).setFlashProperty(PfadZurFlashProperty, Wert)`
- `sprite(flashsprite).getFlashProperty(PfadZurFlashProperty)`

Die vorgestellte Methode des Buttonaufrufes mit `getURL()` funktioniert auch in Director 6.5 und mit Flash 3-Filmen schon.

6.6 Experiment 5: Schieberegler zur Filmsteuerung

Ein Schieberegler kann vielfältig eingesetzt werden. In diesem Experiment wollen wir die Position des Abspielkopfes im Drehbuch entsprechend der Position des Schiebereglers verändern: Wir implementieren also eine einfache Filmsteuerung.

Versuchsaufbau

Wir legen zwei Marker an: „Start“ und „End“. Das Skript wird diese Markerpositionen nutzen und die Navigation auf diesen Bildbereich einschränken. Den Bühnenhintergrund bilden ein Bildrahmen und ein Textdarsteller, in den wir lediglich die aktuelle Bildnummer schreiben wollen (s.u.). Wir ziehen alle Sprites über einen großen Drehbuchbereich (vgl. Abbildung 6.21) auf.

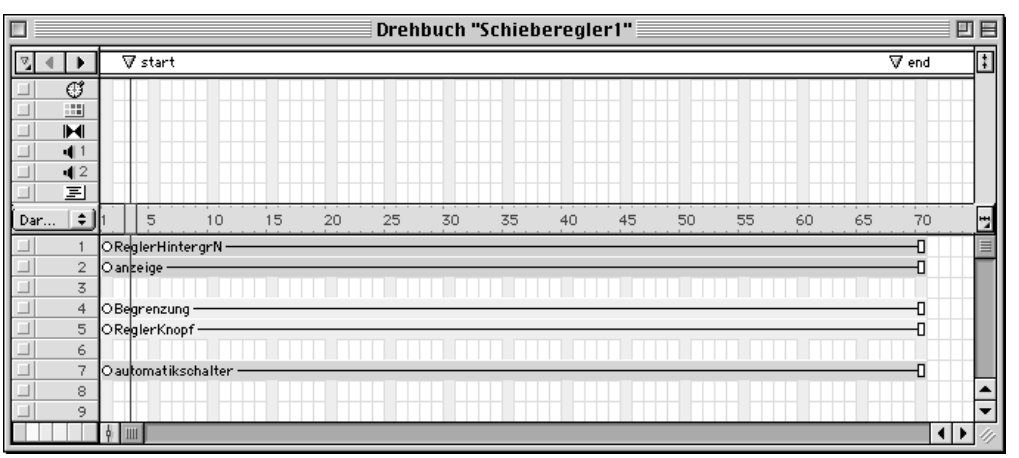


Abbildung 6.21: Die Drehbuchansicht unseres Beispielfilmes

Der eigentliche Schieberegler besteht aus einem Anfasser und einer Schiene, die die Beweglichkeit des Anfassers einschränkt. Den Anfasser platzieren wir in Sprite 5; für die Schiene ziehen wir ein Rechteck-Shape mit der Linienstärke 0 auf, so dass es genauso groß ist wie die Schiene im Hintergrundbild. Dieses Shape kommt in Sprite 4. Beide werden im Drehbuch auf die Länge von „Start“ bis „End“ aufgezogen.

Wählen Sie dann das Shape-Sprite nochmals aus und stellen Sie es im Sprite- bzw. Property-Inspektor auf eine Höhe von 0 Pixeln. Da es die Bewegungsfreiheit des Anfassers einschränken soll, darf es in senkrechter Richtung keine Erstreckung haben (vgl. Abbildung 6.22). Platzieren Sie den Anfasser genau auf die gleiche Y-Koordinate wie das unsichtbare Shape. Auch dabei ist der Sprite-/Propertyinspektor nützlich.

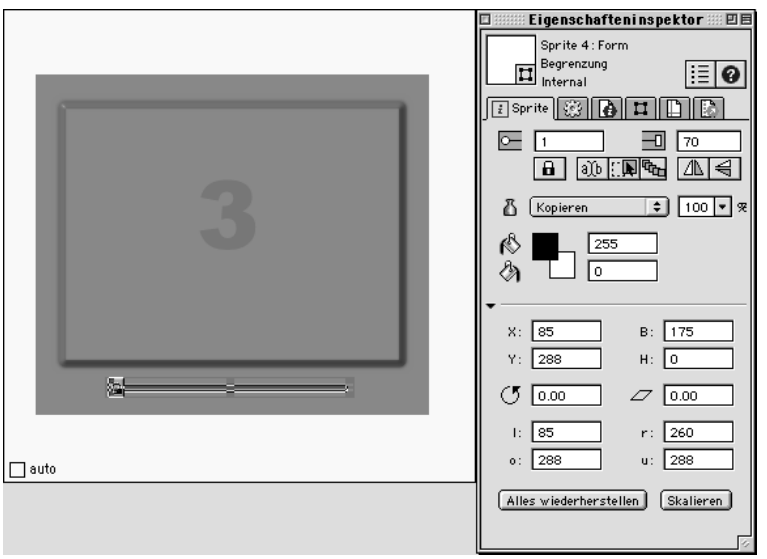


Abbildung 6.22: Ein unsichtbarer Formdarsteller (Shape) begrenzt die Beweglichkeit des Anfassers. Stellen Sie die Höhe auf 0 Pixel ein, und platzieren Sie den Anfasser genau auf denselben Y-Wert wie das Shape.

▶ **Autotext:**

Ein kleines Skript soll – zur Kontrolle unserer Navigation mit dem Schieberegler – die aktuelle Bildnummer in den Textdarsteller in Sprite 2 schreiben.

```
on prepareFrame me
  if the frame >= marker("start") AND
  the frame <= marker("end")
  then sprite(me.spritenum).
  member.text = string(the frame)
  else
    sprite(me.spritenum).member.text = ""
  end if
end
```

Den Director-Film finden Sie als *Schieberegler.dir* auf der CD-ROM.

Durchführung

Das folgende Skript auf dem Anfasser des Schiebereglers übernimmt die gesamte Steuerung auf der Zeitachse des Director-Skriptes. Bei jedem `exitFrame` wird ein `go`-Befehl ausgegeben und damit der Abspielkopf positioniert. Unser übliches „Stopper"-Skript ist daher in diesem Experiment ausnahmsweise überflüssig.

Wir werden einen einfachen Dreisatz bemühen müssen, um die Position des Anfassers in eine Bildposition im Drehbuch umrechnen zu können. Dabei gilt folgendes Prinzip:

PosAnfasser /BreiteRegler = BildPosition /Navigationsbereich

Ausführlich formuliert: „Die Position des Anfassers verhält sich zur Gesamtbreite des Reglers wie die aktuelle Bildposition zum gesamten navigierbaren Bildbereich im Drehbuch." Wenn wir die Bildposition suchen, kann die Gleichung so aufgelöst werden:

*BildPosition = PosAnfasser * Navigationsbereich /BreiteRegler*

Um das nun in Lingo umzusetzen, müssen wir bedenken, dass

- die Position des Anfassers durch seine tatsächliche Position auf der Bühne (`locH`) und den horizontalen Versatz des Reglers (bzw. des unsichtbaren Shapes, das für uns den Bewegungsbereich markiert) bestimmt ist,
- der Navigationsbereich durch die Marker „Start" und „Ende" bestimmt ist,
- die Breite des Reglers tatsächlich die Breite unseres unsichtbaren Shapes ist
- und die Bildposition, die wir als Ergebnis der Berechnung erhalten, noch in den Bereich zwischen den Markern „Start" und „Ende" verschoben werden muss.

Das folgende Behavior-Skript „nav" enthält die notwendigen Berechnungen:

```
property startloc, breite
property pos_start, pos_total, pos_aktuell

on beginsprite me
  sprite(me.spritenum).moveableSprite = 1
  sprite(me.spriteNum).constraint = me.spriteNum - 1
  -- Der Anfasser wird beweglich gemacht und seine Beweglichkeit
  -- auf die Größe des Sprites im um 1 niedrigeren Spritekanal
  -- eingeschränkt.
  breite = sprite(me.spritenum - 1).width
  startloc = sprite(me.spriteNum - 1).locH
  -- Breite und horizontaler Versatz des Reglers
```

```
  pos_start = marker("start")
  pos_total = marker("end") - pos_start + 1
```

Startposition im Drehbuch und die „Breite" des gesamten Navigationsbereichs

```
  pos_aktuell = marker("start")
  sprite(me.spritenum).locH = startloc
```

*Zu Beginn wird der Anfasser auf die Nullposition des Reglers (*startLoc*) verschoben.*

```
end

on exitFrame me
  x = sprite(me.spriteNum).locH - startloc
```

Korrigierter x-Wert des Anfassers

```
  pos_aktuell = pos_start + \
   min(pos_total - 1, x * pos_total / breite)
```

*Dreisatz pos_aktuell = x * pos_total/breite, und zusätzliche Korrekturen für den Versatz des „Start"-Markers (Addition von* pos_Start*) und zur Beschränkung am oberen Ende (*min()*)*

```
  go pos_aktuell
```

go-Befehl, der den Abspielkopf am berechneten Frame festhält

```
end
```

▶ Modifikation

Während nun die interaktive Steuerung wie erwartet funktioniert, fehlt uns noch so etwas wie ein „Automatik-Modus": Der Director-Film läuft, und der Schieberegler wird per Skript auf den der Abspielposition entsprechenden Wert gesetzt.

Das folgende Skript ergänzt unseren ersten Ansatz; es muss in der Behavior-Reihenfolge vor das Skript oben gesetzt werden, damit es den `exitFrame`-Event je nach Wert der Property `auto` weitergibt oder ausfiltert.

```
property msp, auto
property oldauto
```

In diesem Skript werden nur die Properties definiert, die wir hier benötigen. Die Properties des Behaviors „nav" nutzen wir zwar, greifen aber über sprite(me.spritenum).propertyName *darauf zu.*

```
on beginsprite me
```

Die folgende Zeile erspart uns viel Tipperei: msp *fungiert dann als „Shortcut" für die Sprite-Referenz* sprite(me.spritenum)*, also für das Sprite, auf dem sich das Skript befindet.*

```
  msp = sprite(me.spritenum)
end

on exitFrame me
  if not(auto) then
    -- Wenn der Automatikmodus ausgeschaltet ist, lass den Event
    -- zum zweiten Behavior durch...
    pass
  else
    -- ...ansonsten berechne die aktuelle Position des
    -- Schiebereglers - dafür lösen wir den Dreisatz von oben
    -- (pos_aktuell = x * pos_total/breite) nach x auf:
    -- x = pos_aktuell * breite / pos_total und korrigieren wie
    -- oben auch. Achten Sie auf die Verwendung von msp.(...),
    -- um auf die Properties des zweiten Behaviors zuzugreifen.
    pos_aktuell = the frame
    x = msp.startloc + (pos_aktuell - msp.pos_start + 1) \
     * msp.breite / msp.pos_total
    sprite(me.spriteNum).locH = x
    -- Die folgende Zeile bewirkt, dass am Marker „end" auto-
    -- matisch auf den Marker „Start" gesprungen wird.
    if pos_aktuell = marker("end") then go msp.pos_start
    -- stopevent bewirkt, dass das exitFrame-Event nicht an das
    -- zweite Behavior weitergegeben wird.
    stopEvent
  end if
end

on setAuto me, val
  -- Mit diesem Handler können wir von außen die Property auto
  -- ein- und ausschalten.
  auto = val
end
```

Die Umschaltung zwischen Normal- und Automatikmodus können wir mit einem beliebigen Button machen. Im Beispielfilm verwenden wir ein so genanntes Markierungsfeld (Ankreuzfeld), das auf `mouseUp` den Handler `setAuto` im Sprite 5 – dem Sprite des Anfassers – aufruft. Als Parameter wird der Wert der Eigenschaft `hilite` des Markierungsfeldes übergeben. `hilite` ist `true`, wenn das Feld angekreuzt ist, `false`, wenn dies nicht der Fall ist.

```
on beginsprite me
  mouseup me
end

on mouseup me
  sendsprite(5, #setAuto, sprite(me.spritenum).member.hilite)
end
```

6.7 Experiment 6: Schieberegler zur Video-Steuerung

Was ist zu tun, damit unser Schieberegler aus Experiment 5 statt des Director-Filmes ein digitales Video (zum Beispiel einen Quicktime-Film) steuert?

Nicht viel: Wir müssen uns lediglich klarmachen, welche Werte für die Gesamtdauer des Videos und für seine aktuelle Position stehen, nämlich `sprite(VideoSprite).member.duration` und `sprite(VideoSprite).movieTime`. Die Dauer `duration` ist eine Darstellereigenschaft; die aktuelle Abspielposition des Videos `movieTime` dagegen eine playbackbezogene, also Sprite-Eigenschaft.

Der Dreisatz

PosAnfasser / BreiteRegler = BildPosition / Navigationsbereich

heißt bezogen auf unser Video-Sprite:

PosAnfasser / BreiteRegler = SpriteMovietime / MemberDuration

Damit sollte es Ihnen eigentlich möglich sein, nach dem Versuchsaufbau direkt mit einer eigenen Skriptumsetzung loszulegen!

Versuchsaufbau

Importieren Sie den Beispielfilm (oder ein beliebiges anderes Video) in die Director-Besetzung des Filmes *SchiebereglerVideo.dir*. Die Eigenschaften „Pause“ (`pausedAtStart`) und Schleife (`loop`) werden wir gleich im Skript setzen.

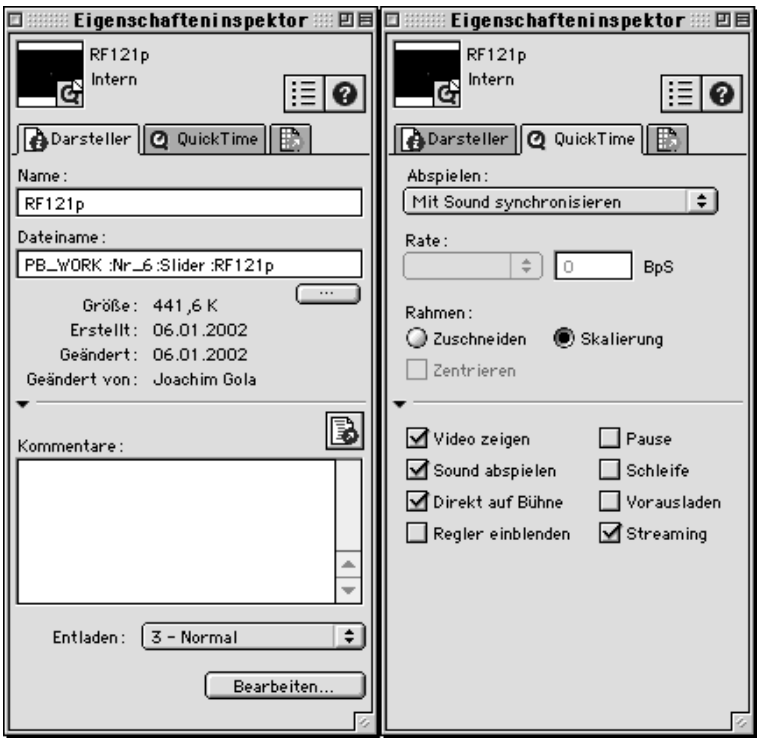


Abbildung 6.23: Im Eigenschafteninspektor von Director 8 finden Sie die wichtigsten Eigenschaften von Quicktime-Darstellern. In Director 7 erreichen Sie die entsprechenden Einstellungsfenster, wenn Sie den Darsteller in der Besetzung auswählen und Strg/⌘+I aufrufen.

Sie können die Eigenschaften aber auch in den Darstellereigenschaften aktivieren. In Director 8 markieren Sie den Film in der Besetzung und nehmen die Einstellung im Eigenschafteninspektor vor. Hier können Sie auch andere Präferenzen eingeben, die zum Abspielen von QuickTime-Filmen wichtig sind: Durch Ankreuzen entscheiden Sie, ob nur der Ton oder nur das Bild gespielt werden. Soll der Film erst einmal auf dem ersten Bildfeld verharren, wählen Sie „Pause“, möchten Sie die QuickTime-Bedientasten nutzen, kreuzen Sie „Regler einblenden“ an. „Direkt auf die Bühne“ ist eine Wiedergabeoption, die Sie immer dann wählen sollten, wenn kein anderes Element vor dem Quicktime-Film angezeigt wird. Wird die Option deaktiviert, so sinkt die Performance Ihres Video-Filmes.

Durchführung

Die erste Umsetzung (ohne „Auto-Modus“) könnte so aussehen; im Vergleich zu Experiment 5 sind nur wenige Zeilen verändert:

```
property startloc, breite
property pos_start, pos_total, pos_aktuell

on beginsprite me
  sprite(me.spritenum).moveableSprite = 1
  sprite(me.spriteNum).constraint = me.spriteNum - 1
  breite = sprite(me.spritenum - 1).width
  startloc = sprite(me.spriteNum - 1).locH
  Wie Experiment 5
  pos_start = 0
  pos_total = member("RF121P").duration
  Die gesamte Dauer des Quicktime-Films soll angezeigt werden.
  pos_aktuell = 0
  sprite(me.spritenum).locH = startloc
  Eigenschaften des Quicktime-Darstellers einstellen:
  member("RF121P").pausedAtStart = 1
  member("RF121P").loop = 1
end

on exitFrame me
  x = sprite(me.spriteNum).locH - startloc
  pos_aktuell = pos_start +
  min(pos_total - 1, x * pos_total / breite)
  Wie Experiment 5
  sprite(2).movieTime = pos_aktuell
  Die Eigenschaft movieTime des Video-Sprites wird auf
  pos_aktuell eingestellt.
end
```

Wollen wir den Auto-Modus in dasselbe Skript integrieren, so ist es wie folgt zu erweitern:

```
property startloc, breite
property pos_start, pos_total, pos_aktuell
property auto

on beginsprite me
  sprite(me.spritenum).moveableSprite = 1
  sprite(me.spriteNum).constraint = me.spriteNum - 1
  breite = sprite(me.spritenum - 1).width
  startloc = sprite(me.spriteNum - 1).locH
```

```
  pos_start = 0
  pos_total = member("RF121P").duration
  pos_aktuell = 0
  sprite(me.spritenum).locH = startloc
  member("RF121P").pausedAtStart = 1
  member("RF121P").loop = 1
end

on exitFrame me
  if (not(auto)) then
    Normalmodus
    x = sprite(me.spriteNum).locH - startloc
    pos_aktuell = pos_start +
    min(pos_total - 1, x * pos_total / breite)
    sprite(2).movieTime = pos_aktuell
  else
    Auto-Modus, x-Position des Anfassers wird aus der aktuellen
    movieTime und der duration (pos_total) errechnet.
    pos_aktuell = sprite(2).movietime
    x = startloc +
    (pos_aktuell - pos_start) * breite / pos_total
    sprite(me.spriteNum).locH = x
  end if
end

on setAuto me, val
  auto = val
  if (auto) then
    sprite(2).movierate = 1
  else
    sprite(2).movierate = 0
  end if
end
```

Auch diese Umsetzung folgt der Logik aus Experiment 5; um das Video-Sprite (das ja im beginSprite-Handler angehalten wurde) selbsttätig abspielen zu lassen, setzen wir `sprite(2).movierate = 1`.

Der `setAuto`-Handler wird wie gehabt im Behavior des Auto-Umschalt-Buttons aufgerufen.

Den fertigen Film finden Sie als *SchiebereglerVideo.dir* auf der CD-ROM.

Rollover: Signal zum Anklicken

Bereits in den Beispielen des Kapitel 6 gibt es teilweise Signale, die den Button als anklickbare Fläche ausweisen: In Experiment 2 ändert sich die Füllung des Vectorshapes, und der Flash-Button aus Experiment 4 zeigt ebenfalls unterschiedliche Zustände für Normal, Mouseover und Mousedown an. In diesem Abschnitt wollen wir das Thema der visuellen Rückmeldungen vertiefen: Wie müssen Darstelleränderungen bei einem Button organisiert werden, dass sich dieser Button „richtig“, also so wie alle anderen Buttons auf Ihrem Betriebssystem auch, verhält? Wie können wir die Grafik-Änderungen organisieren, so dass dasselbe Skript für viele Buttons eingesetzt werden kann? Wie schaffen wir es, dass jeder Button eine andere Aktion ausführen kann, ohne dass wir die gleich bleibenden Skriptteile immer wieder wiederholen?

7.1 Experiment 1: Buttons mit drei Zuständen

In diesem Experiment wollen wir uns Schritt für Schritt einer „idealtypischen“ Umsetzung eines Buttons mit drei optischen Zuständen nähern. Dabei geht es einerseits natürlich um den Button, andererseits aber auch darum, wie wir sukzessive ein Skript allgemeiner formulieren und damit universeller einsetzbar machen können. Ein paar Riegel Schokolade als Nervennahrung sind erlaubt ;-)

Versuchsaufbau

Wir benötigen zunächst nicht mehr als drei unterschiedliche, aber genau gleich große Button-Grafiken. Diese Bedingung ist wichtig: Variiert die Größe nur um wenige Pixel, so werden wir an den Button-Rändern unschöne Zitter-Effekte erhalten.

Im Beispielfilm *Buttonbehavior.dir* dienen zunächst drei einfache Farbflächen als Buttons.

Durchführung

▶ **Ein erster Ansatz:**

Einen einfachen Darstellertausch, der bereits alle drei Zustände verwendet, bietet das folgende Skript („Button1“ im Beispielfilm):

```
property spriteNum
property m_nl, m_hi, m_dn

on beginSprite me
  m_nl = member("nl")
  m_hi = member("hi")
  m_dn = member("dn")
end

on mouseEnter me
  sprite(spritenum).member = m_hi
end

on mouseLeave me
  sprite(spritenum).member = m_nl
end

on mouseDown me
  sprite(spritenum).member = m_dn
end
```

```
on mouseUp me
  sprite(spritenum).member = m_hi
  domyStuff me
end

on domyStuff me
  beep
end
```

Im `beginSprite`-Handler werden die Darsteller für die drei Zustände in die Property-Variablen `m_nl` (Null-), `m_hi` (Hilite-) und `m_dn` (Down-Zustand) geschrieben.

Die folgenden Event-Handler bewirken jeweils eine Darsteller-Änderung: Bei `mouseEnter` soll der Hilite-Darsteller, bei `mouseLeave` wieder der Null-Darsteller angezeigt werden. Entsprechendes bei `mouseDown` und `mouseUp`: `mouseDown` zeigt den Down-Darsteller, `mouseUp` den Hilite-Darsteller (da ja die Maus noch über dem Button ist).

Im Handler `domyStuff` ist nur eine minimale Button-Aktion angegeben: `beep`. Hier sollten Sie natürlich etwas für Ihren Zweck Sinnvolleres eintragen.

Dieser Button funktioniert, aber er verhält sich nicht in jedem Fall so, wie wir es von Windows oder MacOS gewohnt sind. Die folgenden beiden Spezialfälle sind noch abzufangen, und leider wird unser Skript damit etwas komplizierter:

1. Keine Bildänderung, wenn außerhalb des Buttons ein `mouseDown` erfolgt und die Maus dann über den Button gezogen wird. Die folgende Version führt die Property `pMousedDown` ein, mit der überprüft wird, ob das `mouseDown` auf diesem Button erfolgt ist. `pMousedDown` wird im `beginSprite`-Handler auf `0` gesetzt; auf `mouseDown` erhält die Property den Wert `1`, bei `mouseUp` und `mouseUpOutside` wird sie wieder `0`.
2. Wenn das `mouseDown` auf dem Button erfolgt und dann die Maus den Button verlässt, so muss der Down-Zustand angezeigt werden, wenn die Maus wieder über den Button zurückkehrt. Die Systemeigenschaft `the stillDown` erlaubt es (ebenso wie `the mouseDown`) abzufragen, ob die Maus gerade gedrückt ist.

Der `mouseUp`-Event muss nicht gesondert abgefangen werden: Er wird nur dann ausgelöst, wenn auch das `mouseDown` auf demselben Sprite erfolgt ist. Dies gilt für jedes Sprite-Skript.

Hier das veränderte Button-Behavior („Button2“ im Beispielfilm):

```
property spriteNum, pMousedDown
property m_nl, m_hi, m_dn

on beginSprite me
  pMousedDown = 0
  m_nl = member("nl")
  m_hi = member("hi")
  m_dn = member("dn")
end
```

```
on mouseDown me
  pMousedDown = 1
  sprite(spritenum).member = m_dn
end

on mouseUp me
  pMousedDown = 0
  sprite(spritenum).member = m_hi
  domyStuff me
end

on mouseUpOutside me
  pMousedDown = 0
  sprite(spritenum).member = m_nl
end

on mousewithin me
  mouseEnter me
end

on mouseEnter me
  if pMousedDown then
    sprite(spritenum).member = m_dn
  else if not(the stillDown) then
    sprite(spritenum).member = m_hi
  end if
end

on mouseLeave me
  sprite(spritenum).member = m_nl
end

on domyStuff me
  beep
end
```

Der `mouseEnter`-Handler lässt sich wie folgt paraphrasieren: Wenn das `mouseDown` innerhalb des Buttons erfolgt ist (`pMousedDown` ist `TRUE`), dann zeige den Down-Darsteller, wenn das nicht der Fall ist und die Maus auch nicht gedrückt ist (`not(the stillDown)`), dann zeige den Hilite-Darsteller. Wenn die Maus also außerhalb gedrückt wurde, findet bei `mouseEnter` keine Bildänderung statt.

- **Generalisierung:**

Mit folgender Modifikation wollen wir das Behavior noch etwas generel-ler machen: die Darsteller-Änderung ist nämlich nicht der einzige Weg, die Button-Zustände zu visualisieren. Ebenso könnten wir die Sprite-Farbe (`sprite(s).color`), die Transparenz (`sprite(s).blend`) oder andere

Eigenschaften des Sprites ändern wollen. Auf jeden Fall ist es ein Vorteil, die visuellen Effekte nicht im ganzen Skript verteilt zu haben, sondern in einem eigenen Handler zu konzentrieren, der dann von den Event-Handlern aufgerufen werden kann. Zunächst setzen wir das obige Skript nach dieser Maßgabe neu um („Button3“ im Beispielfilm):

```
property pMousedDown

-- Event-Handler:

on beginSprite me
  pMousedDown = 0
end

on mouseDown me
  pMousedDown = 1
  change me, #down
end

on mouseUp me
  pMousedDown = 0
  change me, #hi
  domyStuff me
end

on mouseUpOutside me
  pMousedDown = 0
  change me, #nl
end

on mousewithin me
  mouseEnter me
end

on mouseEnter me
  if pMousedDown then
    change me, #dn
  else if not(the stillDown) then
    change me, #hi
  end if
end

on mouseLeave me
  change me, #nl
end

-- Eigene Handler:

on change me, ziel
```

```
  case ziel of
    #nl:
      sprite(me.spriteNum).member = member("nl")
    #hi:
      sprite(me.spriteNum).member = member("hi")
    #dn:
      sprite(me.spriteNum).member = member("dn")
  end case
  updateStage
end

on domyStuff me
  beep
end
```

Der Effekt dieses Buttons ist der gleiche wie der des vorangehenden; allerdings haben wir uns eine Möglichkeit geschaffen, einfach einen anderen Weg zu implementieren, den Buttonzustand auszudrücken.

- **Modifikation:**

Zum Beispiel können wir den `change`-Handler wie folgt ändern, um statt des Darsteller-Tauschs Veränderungen an der Transparenz und Farbigkeit des Sprites vorzunehmen:

```
on change me, ziel
  case ziel of
    #nl:
      sprite(me.spriteNum).blend = 100
      sprite(me.spriteNum).color = rgb(0, 0, 0)
    #hi:
      sprite(me.spriteNum).blend = 80
      sprite(me.spriteNum).color = rgb(128, 0, 0)
    #dn:
      sprite(me.spriteNum).blend = 50
      sprite(me.spriteNum).color = rgb(0,0,128)
  end case
  updateStage
end
```

- **Darstellerorganisation, erste Möglichkeit:**

Aber auch der Darsteller-Tausch lässt sich genereller formulieren. Wenn wir uns beim Erstellen von Buttons beispielsweise an die Konvention halten, dass stets der Nullzustand ins Drehbuch gelegt wird und in der Besetzung der Hilite- und der Downzustand direkt auf den Nullzustand folgen, so können wir den `beginSprite`-Handler so umformulieren, dass er stets selbstständig die Darsteller-Referenzen anlegt (vgl. „Button4“ im Beispielfilm):

```
property pMousedDown
property m_nl, m_hi, m_dn

-- Event-Handler:
on beginSprite me
  pMousedDown = 0
  m_nl = sprite(me.spriteNum).member
  m_hi = member(m_nl.number + 1)
  m_dn = member(m_nl.number + 2)
end

-- (...) andere Skriptteile ausgelassen

-- Eigene Handler:
on change me, ziel
  case ziel of
    #nl:
      sprite(me.spriteNum).member = m_nl
    #hi:
      sprite(me.spriteNum).member = m_hi
    #dn:
      sprite(me.spriteNum).member = m_dn
  end case
  updateStage
end
```

Die etwas ungewöhnliche Notation `m_hi = member(m_nl.number + 1)` nutzt die (interne, eindeutige) `number` eines Darstellers, nicht die `memberNum`, die bei mehreren Besetzungen mehrfach vorkommen könnte. Als Hilite-Darsteller `m_hi` wird demnach der Darsteller genommen, dessen interne Nummer um eins höher ist als die des Nulldarstellers `m_nl`).

- **Darstellerorganisation, zweite Möglichkeit:**

Eine Konvention, die ebenfalls häufig benutzt wird, ist, die Darsteller mit `"einName_nl"`, `"einName_hi"`, `"einName_dn"` zu benennen. Auch dies könnte im `beginSprite`-Handler entsprechend analysiert werden. In diesem Fall ist die Reihenfolge der Darsteller in der Besetzung egal, allerdings dürfen gleiche Darstellernamen nicht zweimal vorkommen:

```
on beginSprite me
  pMousedDown = 0
  olditemdelimiter = the itemdelimiter
  the itemdelimiter = "_"
  m_nl = sprite(me.spriteNum).member
  wurzel = m_nl.name.item[1]
  m_hi = member(wurzel & "_hi")
  m_dn = member(wurzel & "_dn")
  the itemdelimiter = olditemdelimiter
end
```

Beachten Sie, dass der Darstellername des Null-Darstellers hier in zwei Items zerlegt wird, nachdem die Systemeigenschaft `the itemdelimiter` auf "_" gesetzt wurde. Item 1 ist der uns interessierende erste Teil des Darstellernamens, den wir für den Hilite- und Down-Darsteller dann mit den entsprechenden Endungen versehen. Damit `the itemdelimiter` nicht dauerhaft verstellt wird, merken wir uns die Einstellungen zu Beginn des Skriptes und setzen die Systemproperty am Ende auf den gemerkten Wert zurück.

- **Segmentierung:**

Eine weitere Optimierung betrifft die Button-Aktion. Bisher haben wir den visuellen Effekt und die Button-Aktion in einem Skript untergebracht. Was aber, wenn mehrere Buttons den gleichen Effekt, aber unterschiedliche Aktionen haben sollen?

Dann sollten wir den `domyStuff`-Handler in ein eigenes Behavior auslagern: Streichen Sie die Zeile `domyStuff me` im `mouseUp`-Handler und den gesamten `domyStuff`-Handler, und legen Sie ein neues Behavior an, das ebenfalls auf den Button gezogen wird:

```
on mouseup me
  domyStuff me
end

on domyStuff me
  beep
end
```

oder, da hier ja kaum eine Möglichkeit für Unübersichtlichkeit besteht, auch einfach:

```
on mouseup me
  beep
end
```

Diese Aufteilung ist im „Button4"-Skript im Beispielfilm umgesetzt.

Damit haben wir, was die Organisation der Darstellerwechsel und die Aufteilung der Funktionalität angeht, unseren „Idealbutton" umgesetzt. Welche Form der Darstellerorganisation Sie bevorzugen, bleibt Ihnen überlassen: Das Wichtigste ist, dass Sie sich auf eine der Konventionen festlegen.

Lediglich die Buttonaktionen sind mit dem unverbindlichen `beep` noch etwas dürr ausgefallen. Dem wollen wir zum Abschluss des Experiments abhelfen.

▶ **Konfigurierbares Behavior für Sprungadressen:**

Unser Behavior, das wir zusätzlich zum „Button4"-Behavior auf die Button-Sprites legen, sieht bisher so aus:

```
on mouseup me
  beep
end
```

Es soll über kurz oder lang so aussehen:

```
on mouseup me
  go to frame einEinstellbaresSprungziel
end
```

Das Einstellen des Sprungziels wiederum soll so geschehen wie bei den Behaviors in der Behavior-Bibliothek: Wenn Sie das Behavior auf ein Sprite ziehen, soll sich ein kleines Dialogfenster öffnen und dort sollen Sie das Sprungziel einstellen können.

Davon trennen uns noch zwei Schritte:

1 Wir definieren eine Property für das Springziel.

```
property Sprungziel
on mouseup me
  go Sprungziel
end
```

2 Wir definieren den Dialog, der angezeigt werden soll.

```
property Sprungziel
on mouseup me
  go to frame Sprungziel
end

on getPropertyDescriptionList me
  mlist = [:]
  mlist[#Sprungziel] = [#comment: "Bitte ein Sprungziel
  angeben:", #format: #marker, #default: "none"]
  return mlist
end
```

Mit diesem Skript wird über einen Konfigurationsdialog der Property `Sprungziel` ein Wert zugewiesen. Da wir als `#format` in der Propertydefinition im `getPropertyDescriptionList`-Handler das Format `#marker` eingegeben haben, bekommen wir im Dialog automatisch alle Marker des aktuellen Films sowie die „Standard-Sprungziele" `#previous`, `#loop` und `#next` angezeigt (vgl. Abbildung 7.1). In der Datei *Buttonbehavior_mod.dir* finden Sie diese Lösung auf CD-ROM.

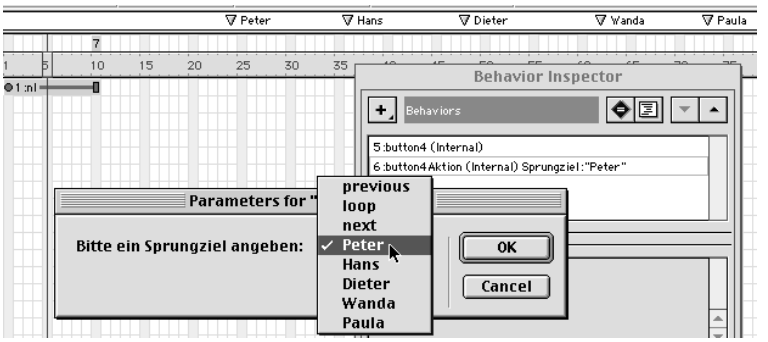


Abbildung 7.1: Eigene Konfigurationsdialoge für Behaviors zu erstellen ist einfach!

Selbstverständlich gibt es Alternativen zu dieser Vorgehensweise: in Experiment 4 in Kapitel 6 haben wir die Spritenummer genutzt, um unterschiedliche Sprungziele anzugeben:

```
on mouseup me
  case me.spritenum of
    5: go "Peter"
    6: go "Hans"
    7: go "Dieter"
    8: go "Wanda"
    9: go "Paula"
    otherwise
       go 1
  end case
end
```

Und es ist auch denkbar, das Sprungziel aus dem Namen des Darstellers abzuleiten – z.B. bei Menü-Buttons, die so benannt sind wie die Sprungziele. Wenn die Nullzustände der Buttons beispielsweise `"Peter_nl"`, `"Hans_nl"` etc. heißen, könnte das Skript so aussehen:

```
property Sprungziel
on beginSprite me
  olditemdelimiter = the itemdelimiter
  the itemdelimiter = "_"
  Sprungziel = item 1 of sprite(me.spriteNum).member.name
  the itemdelimiter = olditemdelimiter
end

on mouseUp me
  go Sprungziel
end
```

Für welchen dieser Wege Sie sich auch entscheiden: Es ist in jedem Fall notwendig, dass Sie sich bei der Konzeption Ihres Director-Films an Konventionen halten. Im ersten Fall die Konvention, dass alle Sprungziele mit einem Marker ausgezeichnet werden müssen, im zweiten, dass Ihre Buttons in bestimmten Spritekanälen liegen müssen, im dritten, dass

die Darsteller-Benennung und die gewählten Markernamen korrespondieren müssen.

7.2 Experiment 2: Ein „inaktiver“ Button-Zustand

Wir benutzen den letzten Stand des „Buttons mit drei Zuständen“ (*Buttonbehavior.dir*, „Button4“) und fügen ihm einen inaktiven Zustand hinzu.

Was heißt das genau? Welche Anforderungen müssen umgesetzt werden?

1. Zum einen gibt es eine vierte Grafik, die signalisieren soll, dass der Button *nicht* angeklickt werden kann.
2. Zum Zweiten benötigen wir einen Auslöser, eine wie auch immer geartete Bedingung, die dafür sorgt, dass der Button vom aktiven in den inaktiven Zustand übergeht (und idealerweise auch eine Möglichkeit, ihn wieder aktiv zu schalten). Im Experiment ist dieser Auslöser gleich der erste Klick auf den Button. Er soll also kein zweites Mal betätigt werden können.
3. Wir müssen für den Button im inaktiven Zustand alle Bildänderungen und alle Klickaktionen unterbinden.
4. Und – das ist die am wenigsten offensichtliche Anforderung – wir müssen darüber nachdenken, was passieren soll, wenn wir durch einen Sprung das Sprite verlassen und später wieder an die Stelle zurückkommen: In unserem Experiment soll der Button dann weiter inaktiv bleiben, er soll sich seinen Zustand also merken, auch wenn wir das Sprite verlassen.

Wir wollen die zusätzliche Funktionalität so implementieren, dass die bisherige Lösung unverändert bleibt: Die Sprites, die einen inaktiven Zustand bekommen, sollen einfach nur ein zusätzliches Behavior erhalten.

Versuchsaufbau

In den Film *ButtonInaktiv.dir* haben wir die Buttongrafiken (Farbflächen) aus Experiment 1 kopiert. Wir benötigen auch noch einen vierten Button-Zustand, den wir nach Null-, Hilite- und Down-Zustand in die folgende Position in der Besetzung platzieren.

Außerdem haben wir folgende Skripte aus Experiment 1 übernommen:

- das Behavior „Button4“, das den Rollover-Effekt produziert
- das Behavior „Button4Aktion“, das bei `mouseUp` einen konfigurierbaren Sprung zu einem Marker ausführt

Legen Sie die Null-Grafik des Buttons auf die Bühne, und weisen Sie ihm zunächst die ersten beiden Behaviors zu. Die Reihenfolge ist dabei egal. Funktioniert der Rollover-Effekt? (Falls nicht: Bedenken Sie, dass die Button-Grafiken in der Reihenfolge Null-Grafik, Hilite-Grafik, Down-Grafik in der Besetzung liegen müssen). Können Sie Sprungziele definieren? (Falls nicht: Sie benötigen einige Marker im Drehbuch, die als Sprungziele genutzt werden können.)

Wenn das alles mit einem einzelnen Button funktioniert, wollen wir dem Button-Sprite schon ein weiteres Behavior zuweisen, das zunächst noch leer ist, aber den Namen „Button4Inaktiv“ erhalten soll. Wählen Sie dafür das Sprite aus und betätigen Sie Neues Behavior im Aufklappmenü des Verhalten-Inspektors.

Da wir mit diesem Behavior auch verhindern wollen, dass Aktionen aus den beiden anderen Behaviors ausgeführt werden, muss es in der Reihenfolge der Behaviors auf dem Sprite an erster Stelle stehen (vgl. Abbildung 7.2). Benutzen Sie die Pfeile rechts oben im Verhalten-Inspektor (bzw. im Behavior-Tab des Eigenschafteninspektors in Director 8 und neuer), um das Skript „Button4Inaktiv“ ganz nach oben zu verschieben.

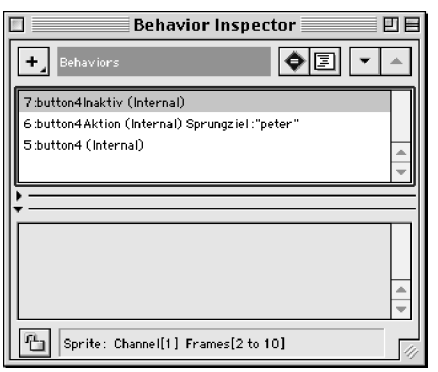


Abbildung 7.2: Das Behavior „button4Inaktiv“ haben wir mit den Pfeiltasten nach oben bewegt.

Wenn Sie einen Button so eingerichtet haben, reduzieren Sie seine Sprite-Länge auf ca. zehn Frames und duplizieren Sie ihn im Drehbuch mehrfach (mit Alt-Ziehen).

Platzieren Sie einen Marker „Start“ in Sprite 2 und einige weitere Marker in höheren Framenummern. In Sprite 10 und bei allen Markern außer „Start“ soll ein Stopper-Skript platziert werden. Geben Sie allen Buttons im Startbereich im Behavior „Button4Aktion“ ein Sprungziel. Im Drehbuchbereich nach Frame 10 legen Sie durchgängig einen Button an, der als Sprungziel den Marker „Start“ erhält. Er soll *nicht* das Behavior „Button4Inaktiv“ erhalten.

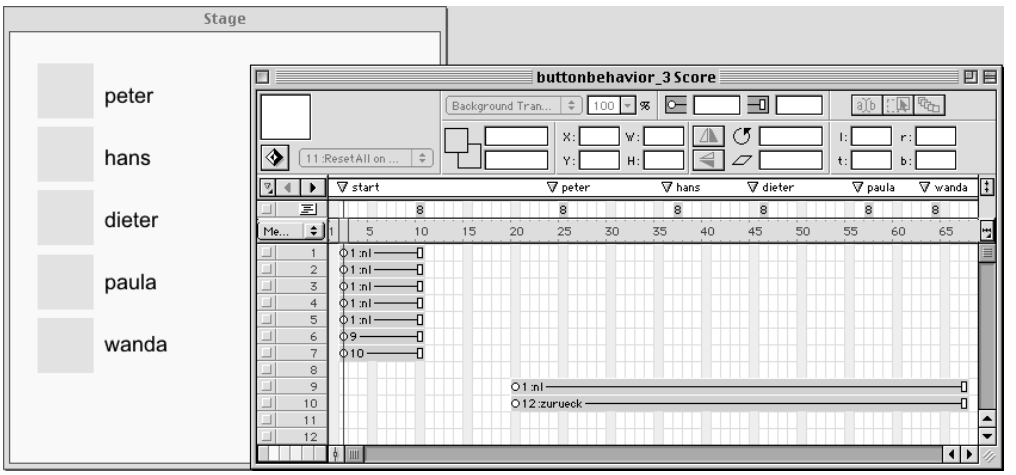


Abbildung 7.3: Bühne und Drehbuchansicht unseres Experiments

Drehbuch und Bühne sehen nun aus wie in Abbildung 7.3.

▸ Events:

Am Beispiel eines `mouseUp`-Events wollen wir den Weg dieser Nachricht durch die „Event-Hierarchie“ von Director beschreiben. Wenn wir auf ein Sprite klicken, sendet Director das `mouseUp`-Event in folgender Reihenfolge an Behaviors und andere Skripte (vgl. Tabelle 7.1).

Event-Hierarchie bei Klick auf Sprite	**• `mouseUp` wird ausgeführt** **x `mouseUp` wird nicht ausgeführt, auch wenn `on mouseUp`-Skript vorhanden** **– kein `on mouseUp`-Skript vorhanden**			
(Primärer Eventhandler: `the mouseUpScript`)	(•)	(•)	(•)	(•)
Sprite 1. Behavior 2. Behavior ...	• • •	– – –	– – –	– – –
Darstellerskript	x	•	–	–
Frameskript	x	x	•	–
Filmskript	x	x	x	•

Tabelle 7.1: Eventhierarchie bei `mouseUp`. Dies gilt entsprechend für alle Mausevents (`mouseDown`, `mouseEnter`, `mouseLeave`, `mouseWithin`).

Ein `on mouseUp` in einem Behavior sorgt also dafür, dass evtl. vorhandene Eventhandler in Darsteller-, Frame- und Filmskripten nicht ausgeführt werden. Ist in mehreren Behaviors auf einem Sprite aber ein `on mouseUp`-

Eventhandler, so werden *alle*, und zwar in der Reihenfolge der Behaviors, ausgeführt. Um diese Eventabfolge zu unterbrechen, bietet Director den Befehl `stopEvent`.

```
on mouseUp me
  stopEvent
end
```

Dieses Skript bewirkt, dass der Event auch von den folgenden Behaviors auf demselben Sprite – selbst, wenn dort `on mouseUp`-Eventhandler vorhanden sind – nicht mehr empfangen werden kann: Der Event wird abgeblockt.

Den Befehl `stopEvent` werden wir nutzen, um mit dem Behavior „Button4Inaktiv“ die Weitergabe von Mausevents zu unterbinden.

- **„Lebensdauer“ von Properties:**

Eine Property lebt so lange wie das Sprite, in dessen Behavior sie definiert wurde. Sobald das Sprite nicht mehr im Drehbuch liegt (weil wir beispielsweise einen Sprung auf einen anderen Frame vorgenommen haben), ist der Wert der Property verloren. Wenn wir Property-Werte eines Sprites dauerhaft behalten wollen, muss das Sprite also immer im Drehbuch vorhanden sein (was häufig nicht wünschenswert ist), oder wir müssen nach Wegen suchen, Sprite-Zustände außerhalb des Sprites „aufzubewahren“ und wiederherzustellen, wenn das Sprite wieder angezeigt wird. Für dieses Problem werden wir in diesem Experiment ebenfalls eine Lösung suchen.

Durchführung

In diesem Skript werden wir eine Property `pActive` benutzen, die signalisiert, ob der Button aktiv (`pActive = TRUE`) oder inaktiv (`pActive = FALSE`) ist. Die Property `m_in` soll, analog zu `m_nl`, `m_hi` und `m_dn` aus dem „Button4“-Behavior, den Darsteller enthalten, der für den inaktiven Zustand angezeigt wird:

```
m_in = member(sprite(me.spriteNum).member.number + 3)
```

also den Darsteller, der an der dritten Position nach dem Nulldarsteller in der Besetzung liegt.

- **Eine erste Umsetzung:**

Die erste Formulierung des Skriptes sieht so aus:

```
property m_in
property pActive

-- Selbst-Konfiguration
on beginsprite me
  pActive = TRUE
  m_in = member(sprite(me.spriteNum).member.number + 3)
end

-- Alle Events abblocken
on mouseUp me
  if not(pActive) then stopevent
  else pActive = FALSE
end
```

Alle Maus-Events, die in den Behaviors „Button4" und „Button4Aktion" genutzt werden, werden mit stopEvent *abgeblockt, wenn* pActive = FALSE *ist:* mouseUp, mouseDown, mouseWithin, mouseEnter *und* mouseleave. pActive *wird bei* mouseUp *auf* FALSE *gesetzt.*

```
on mouseDown me
  if not(pactive) then stopEvent
end
on mouseWithin me
  if not(pactive) then stopEvent
end
on mouseEnter me
  if not(pactive) then stopEvent
end
on mouseLeave me
  if not(pactive) then stopEvent
end

-- Eigene Bildänderung
on prepareFrame me
  if not(pActive) then sprite(me.spriteNum).member = m_in
```

Der inaktive Darsteller m_in *wird angezeigt, falls* pActive=FALSE *ist.*

```
end
```

Dieses Behavior hat noch kein „Gedächtnis“ – wenn Sie also einen Effekt sehen wollen, müssen Sie es bis zu der Stelle hin aufziehen, zu der Sie mit dem „Button4Aktion“-Behavior springen. In Abbildung 7.4 haben wir ein Sprite verlängert: Nur dieses Sprite kann den inaktiven Zustand anzeigen. Die anderen Sprites haben dieselben Skripte; da sie aber nicht durchgängig im Drehbuch liegen, erhalten sie beim Rücksprung auf den Marker „Start“ wieder ihren Standardwert.

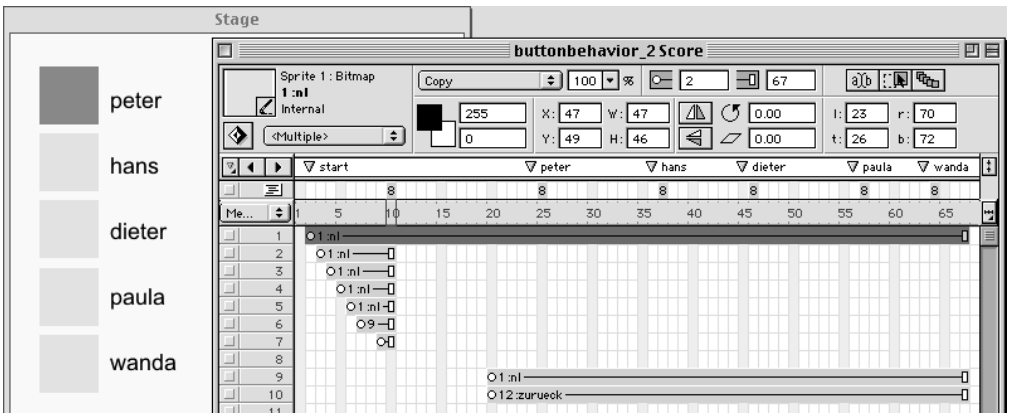


Abbildung 7.4: Nur Sprite 1 zeigt den inaktiven Zustand an: Die anderen Sprites „vergessen“ ihre Property-Werte, da sie nicht durchgehend im Drehbuch liegen.

▶ Der Button mit Gedächtnis:

Eine mögliche Lösung dieses Problems liegt darin, im `endSprite`-Handler den Wert der Property `pActive` in einer globalen Variablen zu speichern und bei `beginSprite` ihn von dort wieder einzulesen. Da wir mehrere Buttons derselben „Bauart“ haben, ist es sinnvoll, eine Propertyliste (`activeList`) als globale Variable zu nutzen. Jeder Button bekommt eine eindeutige `buttonID`, und er sichert (und findet) seinen `pActive`-Wert unter dieser Identifikation in der Propertyliste.

```
global activeList
```

activeList ist der globale Container, in den wir die Buttonzustände sichern.

```
property m_in
property buttonID
property pActive, startActive
```

startActive ist der Startwert von pActive, falls kein Wert in activeList gesichert wurde.

```
-- Selbst-Konfiguration
on beginsprite me
  startActive = TRUE
  buttonID = symbol("b_" & me.spriteNum & "_" & \
   sprite(me.spriteNum).startFrame)
```

buttonID wird aus der Spritenummer und dem ersten Frame des Sprites zusammengefügt und dann in ein Symbol verwandelt.

```
  m_in = member(sprite(me.spriteNum).member.number + 3)
  getActive me
```

Mit getActive me holt sich das Skript den Wert von pActive aus activeList. Die Funktion ist am Ende des Skripts definiert.

```
end

-- Events abblocken
on mouseUp me
  if not(pActive) then stopEvent
  else pActive = FALSE
end
on mouseDown me
  if not(pactive) then stopEvent
end
on mouseWithin me
  if not(pactive) then stopEvent
end
on mouseEnter me
  if not(pactive) then stopEvent
end
on mouseLeave me
  if not(pactive) then stopEvent
end

-- Eigene Bildänderung
on prepareFrame me
  if not(pActive) then sprite(me.spriteNum).member = m_in
end

-- Aktivitätszustand aus activeList lesen
on getActive me
  if voidP(activeList) then activeList = [:]
```

Falls activeList *noch nicht definiert wurde, wird die Variable hier als Propertyliste initialisiert.*

```
  test = activeList[buttonID]
```

Wir lesen aus activeList *die Property* buttonID *aus. Falls sie nicht existiert, wird* VOID *zurückgeliefert...*

```
  if voidP(test) then pActive = startActive
```

...was wir auch gleich überprüfen. Wenn test *den Wert* VOID *hat, nutzen wir stattdessen* startActive *als Wert für* pActive, *ansonsten nutzen wir* test.

```
  else pActive = test
end

-- Aktivitätszustand in activeList speichern
on endSprite me
  activeList[buttonID] = pActive
```

Wenn das Sprite verlassen wird (z.B. bei einem Sprung auf einen anderen Frame), sichern wir den aktuellen Wert von pActive *in die globale Variable* activeList.

```
end
```

```
-- von außen verändern
on changeActive me, val
```

Dieser Handler dient dazu, den Button von außen aktiv oder inaktiv zu machen. Er kann z.B. mit `sendAllSprites(#changeActive, TRUE)` *aufgerufen werden, um alle Sprites wieder aktiv zu schalten.*

```
  pActive = val
  if pActive then sendsprite(me.spriteNum, #change, #nl)
```

Hier wird ein Handler aus dem „Button4"-Behavior aufgerufen, wenn der Button auf aktiv geschaltet wird. Der Aufruf bewirkt, dass der Nulldarsteller angezeigt wird.

```
end
```

Die Werte der `activeList` können Sie im Watcher (Menü Fenster/Watcher) beobachten. Abbildung 7.5 zeigt eine Momentaufnahme.

Da die Buttons ja bereits bei der ersten Betätigung inaktiv werden, mag es sinnvoll sein, sich ein Skript zu erstellen, das alle Buttons wieder aktiviert. In unserem Behavior ist alles dafür vorbereitet: Der Handler `changeActive` schaltet den Zustand des Buttons um.

Das Skript für den Button „alle aktivieren" ist einfach:

```
on mouseup me
  sendallSprites(#changeActive, TRUE)
end
```

Abbildung 7.5: *Im Watcher können Sie die Werte von* `activeList` *verfolgen.* `#b_2_3` *und* `#b_4_5`*, also die Buttons, die mit „hans" und „paula" beschriftet sind, sind inaktiv.*

Diskussion

Es gibt selbstverständlich weitere Lösungsmöglichkeiten für die zu Beginn genannten Anforderungen an einen Button mit einem „inaktiven" Zustand. Der Vorteil der hier gewählten Lösung liegt darin, dass sie modular ist und zusammen mit fast beliebigen Skripten für die Darstellerumschaltung und für die Buttonaktion genutzt werden kann. Dafür mussten wir allerdings recht tief in die Director-Trickkiste greifen und uns mit der Event-Hierarchie auseinander setzen. Auch die Kommunikation zwischen Sprites sowie zwischen einzelnen Behaviors eines Sprites war Thema der Durchführung. Die Funktionen `sendSprite(Spritenummer, #Befehl, Parameter)` und `sendAllSprites(#Befehl, Parameter)` sind essentiell für diese Kommunikation.

7.3 Experiment 3: Jeweils ein Button einer Buttongruppe bleibt gedrückt

Wenn Sie gerade Experiment 2 durchgearbeitet haben, sollte Ihnen eine Lösung für die folgende Anforderung nicht schwerfallen:

Mehrere Buttons sollen sich aufeinander abgestimmt verhalten. Wird einer gedrückt, so sollen alle anderen nicht gedrückt sein. Ein gedrückter Button soll inaktiv sein.

Experiment 5 in Kapitel 3 zeigt Lösungen für eine ähnliche Fragestellung.

Diese Aufgabenstellung ist ähnlich wie die in Experiment 2 – und wir müssen nur formulieren, wie (und auch wo) die zusätzliche Funktionalität zu implementieren ist.

Unser Vorschlag: Wir lassen die Behaviors aus Experiment 2 („Button4", „Button4Aktion", „Button4Inaktiv") unangetastet, und suchen nach einem Weg, mit einem weiteren Behavior die Koordination der Buttons, die einer Gruppe zugehören, zu erreichen.

Versuchsaufbau

Im Beispielfilm *StickyButton.dir* haben wir den letzten Stand aus Experiment 2 übernommen und nur eine kleine Änderung vorgenommen: Die Buttons sind auf der Bühne in zwei Gruppen auseinander gerückt, und die Sprungziele der Buttons der oberen Gruppe sind durchgängig auf den Marker „Start" geändert, um das Testen zu vereinfachen.

Durchführung

Das folgende Skript muss in der Behaviorreihenfolge des Sprites oberhalb der Button-Aktion, aber unterhalb des Inaktiv-Skripts liegen (vgl. Abbildung 7.6). Nur so ist sichergestellt, dass die Event-Blockade des Inaktiv-Skripts auch hier funktioniert, andererseits aber die Status-Änderungen dieses Skriptes ausgeführt werden, bevor der Abspielkopf mit einer Button-Aktion möglicherweise das Sprite verlässt.

```
property grupname

on mouseUp me
  sendAllSprites(#grupActive, grupname)
```

grupActive *ist ein selbst definierter Handler in diesem Skript. Siehe unten.*

```
  sendsprite(me.spriteNum, #changeActive, FALSE)
```

Hiermit wird im Behavior „Button4Inaktiv" des aktuellen Sprites der Handler changeActive *mit dem Parameter* FALSE *aufgerufen: Der Button wird inaktiv gemacht.*

```
end

on grupActive me, val
  if val = grupname then \
   sendsprite(me.spriteNum, #changeActive, TRUE)
```

Nur wenn der übergebene Wert val *dem eigenen Gruppennamen entspricht, erhält das aktuelle Sprite mit* #changeActive, TRUE *die Aufforderung, das Sprite aktiv zu machen. Durch die vorgeschaltete Bedingung ist es möglich, mehrere Buttongruppen gleichzeitig auf der Bühne zu haben; sie werden sich nicht gegenseitig beeinflussen.*

```
end

on getpropertydescriptionlist me
```

Der Handler definiert den Eingabedialog für die Property grupname. *Als Gruppenname kann jeder String eingegeben werden.*

```
  mlist = [:]
  mlist[#grupname] = [#comment: "Gruppenname des Buttons:", \
   #format: #string, #default: ""]
  return mlist
end
```

Wir ziehen das Behavior auf alle fünf Button-Sprites, definieren aber unterschiedliche Gruppennamen (z.B. für die oberen drei Sprites den Namen „peterhansdieter", für die beiden unteren den Namen „paulawanda".

Abbildung 7.6:
Die ersten drei Buttons werden mit dem Behavior „Button4Group“ als Gruppe zusammengefasst. Sie erhalten über den Parameter-Dialog des Behaviors denselben Gruppennamen. Das Behavior wurde mit den Pfeiltasten an die zweite Position in der Behaviorreihenfolge geschoben.

7.4 Experiment 4: Der „deaktivierbare“ Button – ein einfaches Beispielprojekt

In diesem Experiment wollen wir die Ergebnisse aus den Experimenten 1 bis 3 einsetzen – und gleichzeitig etwas vereinfachen. Ziel ist es, von einem Ausgangsbild aus mit zwei Buttons auf andere Bilder zu springen. Sobald das geschehen ist, soll der entsprechende Button inaktiv werden, so dass man kein zweites Mal dorthin springen kann.

Die Umsetzung weicht bewusst von Experiment 2 ab, denn wir wollen eine möglichst einfache Behavior-Struktur umsetzen.

Versuchsaufbau

Der Beispielfilm *ButtonDeaktivierbar.dir* auf der CD-ROM enthält die Marker „Start“, „oben“ und „unten“ und die „Stopper“-Skripte an den Szenenenden.

Zwei Buttons für die Sprünge nach „oben“ bzw. „unten“ liegen in jeweils vier Zuständen vor (vgl. Abbildung 7.7). Legen Sie die Motive für die einzelnen Szenen (die Darsteller „fig1“ bis „fig3“) jeweils in Sprite 1 des Drehbuchs.

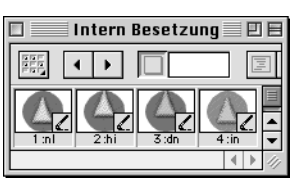


Abbildung 7.7:
Die vier Gesichter des Buttons in Button-Deaktivierbar.dir: Standard-, Rollover-, Maus-Gedrückt- und deaktivierter Button

Die Buttons für die Sprünge in die Szenen kommen in die Startszene: Legen Sie jeweils den Nulldarsteller ins Drehbuch (Darsteller „nl“ bzw. „2_nl“). Abbildung 7.9 zeigt den Drehbuchaufbau.

Außerdem benötigen wir auch zwei Transition-Darsteller. Legen Sie den ersten an, indem Sie im Transition-Kanal des Drehbuchs an beliebiger Stelle doppelklicken und die Einstellungen wie in Abbildung 7.8 vornehmen. Wechseln Sie dann in die Besetzung, benennen Sie den automatisch generierten Darsteller mit dem Namen „up“ und duplizieren Sie ihn (Menü Bearbeiten/Duplizieren). Benennen Sie die Kopie mit „down“ und stellen Sie nach einem Doppelklick den Transition-Typ auf „Aufdecken/Nach unten aufdecken“.

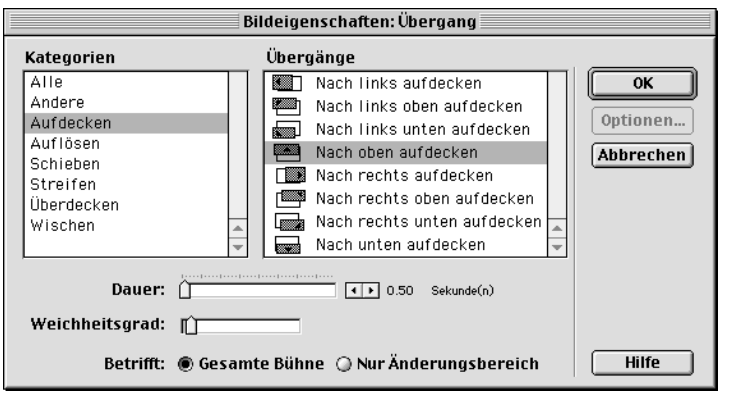


Abbildung 7.8: Transition-Einstellungen für den Darsteller „up“

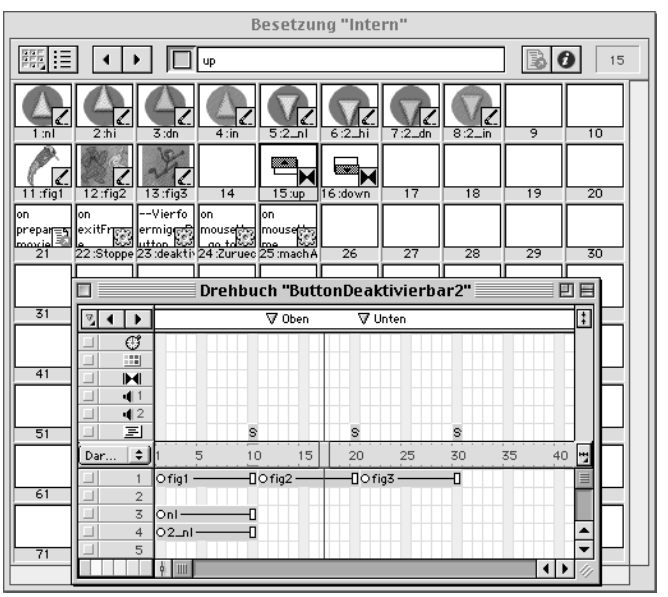


Abbildung 7.9: Besetzung und Drehbuch des Beispielfilms

Durchführung

Bei der Umsetzung nehmen wir uns den „Button4“ aus Experiment 5 als Ausgangspunkt – und ergänzen ihn um folgende Aufgaben:

1. Der Button muss wissen, ob er gerade aktiv oder inaktiv ist. Dazu dient eine Property-Variable `pActive`.
2. Ist der Button inaktiv, so müssen Bildänderungen und -aktionen unterbunden werden. Wir werden also an den entsprechenden Stellen überprüfen müssen, ob der Button aktiv ist: `if pActive then (...)`.
3. Der Button soll sich auch bei Szenenwechsel merken, ob er schon einmal geklickt wurde. Da die Werte von Properties verloren gehen, sobald das Sprite verlassen wird, speichern wir in der Listenvariable `gwarschonda` die Spritenummern der angeklickten Buttonsprites.
4. Bei Rückkehr auf die „Start“-Szene soll der richtige Buttonzustand eingestellt werden: Unser Skript muss entsprechend `gwarschonda` auswerten und sich dann entweder als „aktiv“ oder als „inaktiv“ (`pActive = 1` oder `pActive = 0`) ausweisen.
5. Ein eigener Handler soll ermöglichen, dass der Button (bzw. alle Buttons einer Szene) von einem anderen Skript aus wieder aktiviert wird. Diesen Handler wollen wir `makeActive` nennen.

Das folgende Skript setzt diese Anforderungen um. Die eingeschobenen Kommentare weisen Sie auf die Abweichungen zu „Button4“ hin:

```
property pMousedDown
property m_nl, m_hi, m_dn, m_in
```

pActive hält die Information, ob der Button gerade aktiv (Wert: `true`*) oder inaktiv (*`false`*) ist.*

```
property pActive
```

Die globale Variable `gwarschonda` *wird im* `prepareMovie`*-Handler als leere Liste* `[]` *initialisiert. Da sie in diesem Skript benutzt werden soll, wird sie auch hier als global deklariert.*

```
global gwarschonda

-- Initialisierung
on beginSprite me
  pMousedDown = 0
  m_nl = sprite(me.spriteNum).member
  m_hi = member(sprite(me.spriteNum).member.number + 1)
  m_dn = member(sprite(me.spriteNum).member.number + 2)
  m_in = member(sprite(me.spriteNum).member.number + 3)
  pActive = 1
```

Wenn die aktuelle Spritenummer in der Liste `gwarschonda` *vorhanden ist (sprich: ihre Position größer als 0 ist), dann mache den Button inaktiv, sonst mache ihn aktiv.*

```
  if gwarschonda.getPos(me.spritenum) > 0 then
    change me, #in
  else
    change me, #nl
  end if
end

-- Eigene Handler
on buttonaktion me
```

Fügt die aktuelle Spritenummer zur Merkliste gwarschonda *hinzu.*

```
  gwarschonda.add(me.spritenum)
```

Die Buttonaktionen haben wir ebenfalls direkt ins Skript geschrieben; wir unterscheiden nach der Spritenummer: In Sprite 3 liegt z.B. der Button, der zur Szene „oben" springen und die Transition „up" nutzen soll.

```
  if me.spritenum = 3 then
    puppetTransition member("up")
    go to "Oben"
  else if me.spritenum = 4 then
    puppetTransition member("down")
    go to "unten"
  end if
end

on makeActive me
```

Dieser Handler soll „von außen", also von anderen Skripten aufgerufen werden. Er setzt pActive *auf* true*, löscht die eigene Spritenummer aus* gwarschonda *und „aktiviert" den Button durch Aufruf des* change*-Handlers. Erläuterungen zur Listensyntax siehe unten.*

```
  pActive = 1
  listpos = gwarschonda.getpos(me.spritenum)
  if listpos > 0 then gwarschonda.deleteAt(listpos)
  change me, #nl
end

on change me, ziel
  if pActive then
```

Die folgenden Änderungen sollen nur ausgeführt werden, wenn pActive = TRUE *ist.*

```
    case ziel of
      #in:
        sprite(me.spriteNum).member = m_in
        pActive = 0
      #nl:
```

```
        sprite(me.spriteNum).member = m_nl
      #hi:
        sprite(me.spriteNum).member = m_hi
      #dn:
        sprite(me.spriteNum).member = m_dn
    end case
    updateStage
  end if
end

-- Event-Handler
```

Vgl. Experiment 1

```
on mouseDown me
  pMousedDown = 1
  change me, #dn
end

on mouseUp me
  pMousedDown = 0
  change me, #hi
  if pActive then buttonaktion me
end

on mouseUpOutside me
  pMousedDown = 0
  change me, #nl
end

on mousewithin me
  mouseEnter me
end

on mouseEnter me
  if pMousedDown then
    change me, #dn
  else if not(the stillDown) then
    change me, #hi
  end if
end

on mouseLeave me
  change me, #nl
end
```

Zwei Skripte benötigen wir noch: Zum einen muss die globale Variable `gwarschonda` initialisiert werden, damit wir später mit Listensyntax auf sie zugreifen können. Enthält eine Variable keine Liste, so führt ein Zugriff mit `getPos()` oder anderen Listenbefehlen zu Skriptfehlern. Die Initialisierung machen wir zum Beispiel in einem Filmskript im `prepareMovie`-Handler, so dass sichergestellt ist, dass die Variable den korrekten Startwert hat, sobald das erste Sprite existiert. Öffnen Sie dafür mit Strg / ⌘+⇧+U ein neues Filmskript und geben Sie die folgenden Zeilen ein:

```
on preparemovie
  global gwarschonda
  gwarschonda = []
end
```

Außerdem haben wir vor, auf die kleine Figur in der „Start"-Szene ein Behavior zu legen, das die beiden Buttons wieder aktiv schaltet. Es ist nach der Vorarbeit im Buttonbehavior denkbar einfach:

```
on mouseUp me
  sendAllSprites(#makeActive)
end
```

Es wird damit in allen Sprites der `makeActive`-Handler ausgeführt, falls ein solcher existiert.

Diskussion

Im Behavior der beiden Navigationsbuttons haben wir einige Listenbefehle benutzt, um einer linearen Liste ein Element hinzuzufügen (`aList.add()`), ein Element an einer bestimmten Position zu löschen (`aList.deleteAt(ListenPosition)`) und um die Position eines bestimmten Wertes in einer Liste zu bestimmen (`aList.getPos(einWert)`). Da `getPos()` den Wert `0` zurückliefert, falls das gesuchte Element in der Liste nicht existiert, haben wir diese Funktion auch benutzt, um die Existenz eines Wertes in der Liste zu überprüfen:

```
listpos = gwarschonda.getpos(me.spritenum)
if listpos > 0 then gwarschonda.deleteAt(listpos)
```

In `listpos` schreiben wir hier zunächst die Position der aktuellen Spritenummer in der Liste `gwarschonda`. Nur wenn dieses Listenelement existiert (`if listpos > 0 then`), löschen wir es mit `deleteAt()` aus der Liste. Ohne den vorgeschalteten Test würden wir Skriptfehler erhalten, wenn wir versuchen würden, nicht-existente Listenelemente zu löschen.

Sie können die Listenbefehle auch einfach im Nachrichtenfenster testen:

```
alist = []
aList.add(3)
aList.add(4)
put alist
-- [3, 4]
```

Erst wenn eine Variable als Liste initialisiert ist (aList = []*), können wir Listenbefehle auf die Variable anwenden.*

```
put aList.getPos(4)
-- 2
```

Der Wert 4 wurde an der zweiten Listenposition gefunden.

```
aList.deleteAt(aList.getPos(3))
put aList
-- [4]
```

Der Wert 3 wurde gelöscht.

```
put aList.getPos(3)
-- 0
```

getPos *ergibt nun* false.

Navigation: Konzepte und Skriptumsetzungen

In den folgenden Experimenten wollen wir einige der Lösungen, die wir in den beiden vorangehenden Kapiteln entwickelt haben, in kleinen Projekten einsetzen. Vorab allerdings sollten wir uns noch überlegen, was „Navigation“ in einem Multimedia-Projekt denn eigentlich bedeutet.

Die Verbindung der Einzelteile Ihres Projekts kann auf sehr unterschiedliche Arten geschehen – unterschiedlich in Hinblick auf das „Konzept“, das Sie Ihrer Anwendung zugrunde legen, aber auch sehr unterschiedlich, was die Umsetzung in Director angeht.

Das Navigations„konzept“ einer Director-Anwendung (und anderer Multimedia-Anwendungen wie einer Website) kann beispielsweise eine einfache lineare Abfolge von Szenen/Bildern/Frames sein: eine Diashow, bei der ein Bild nach dem anderen gezeigt wird. Die einzige Wahlmöglichkeit des Anwenders ist, jeweils zur nächsten Szene zu springen.

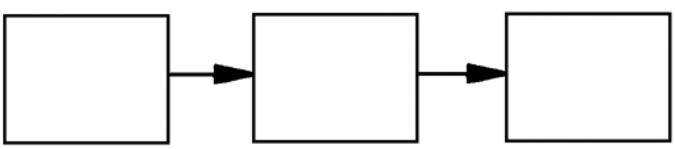
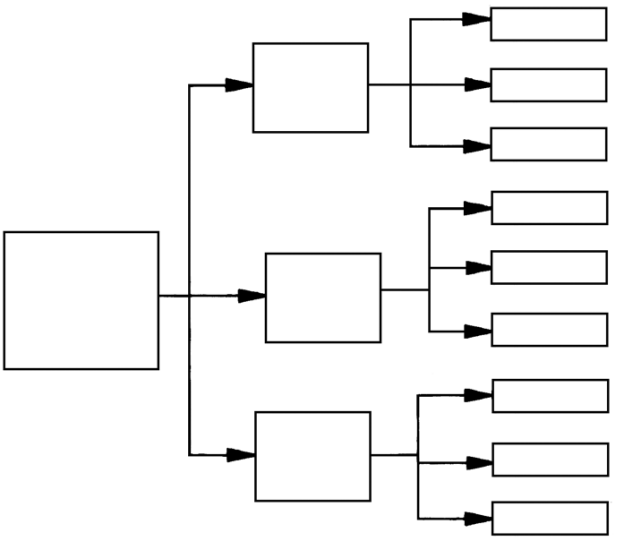


Abbildung 8.1: Lineare Navigationsstruktur

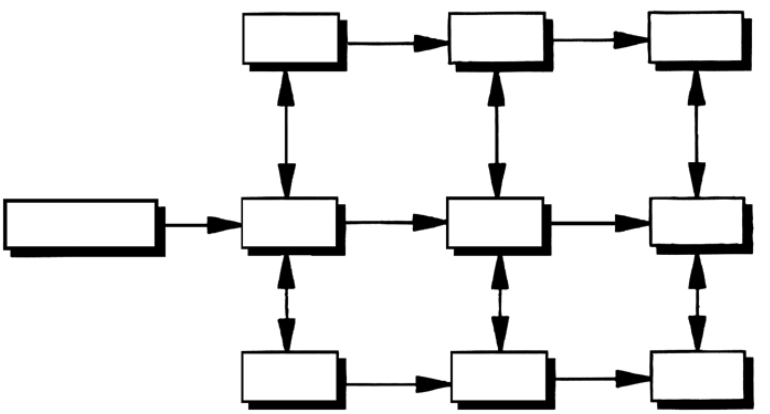
Mehr Wahlmöglichkeiten eröffnen Sie dem User bei hierarchischen oder parallelen Strukturen. Die allermeisten Websites und auch der überwiegende Teil von Multimedia-Produktionen haben eine hierarchische Grundstruktur: Der Anwender kann sich für Bereiche entscheiden, im Bereich wiederum für Unterbereiche usw. Ein solcher hierarchischer Aufbau ist sehr dienlich für informationslastige Anwendungen, bei denen dem Anwender Mittel an die Hand gegeben werden sollen, die Information, die ihn interessiert, möglichst geradlinig zu finden. Die typischen Auswahlmenüs für die Bereiche mit Untermenüs für die Subbereiche sind Ausdruck einer solchen Struktur.

Abbildung 8.2: Hierarchische Navigationsstruktur



Eine „parallele“ Struktur entsteht, wenn mehrere lineare Stränge zusammengeschaltet werden, so dass Übergänge von einem Strang zum anderen möglich sind. Hier steht die Entscheidung des Anwenders für den einen oder anderen Weg im Mittelpunkt: Es ist ein Konzept, das beispielsweise in Spielen zum Einsatz kommt. Es eignet sich, Wege komplizierter zu machen, und weniger zur gezielten Suche nach Sachinformation.

Abbildung 8.3: „Parallele“ Navigationsstruktur



Treibt man die „parallele“ Struktur auf die Spitze, so könnte eine netzartige Struktur entstehen: Jede „Szene“ kann mit beliebig vielen anderen verbunden werden, lineare Stränge sind möglich, aber nicht nötig. Das Labyrinth ist eine netzartige Struktur.

Diese Überlegungen haben zunächst noch gar nichts mit der Umsetzung in Director zu tun: Sie dienen – auch wenn Sie für Ihr eigenes Projekt eine Struktur entwerfen – nur dazu, dass Sie sich selbst darüber klar werden, wie Ihre Anwendung funktionieren soll. In Director haben Sie nämlich kein Werkzeug, eine solche Struktur zu erstellen oder abzubilden – ein großes Manko im Vergleich zu anderen Programmen wie Macromedia Authorware oder Asymetrix Toolbook. Das heißt: Die Struktur existiert in Ihrem Kopf (und vielleicht auch auf Papier), und um sie in Director umzusetzen, haben Sie prinzipiell zwei Möglichkeiten:

- Umsetzung im Drehbuch: Die Navigationsstruktur wird in eine lineare Abfolge von „Szenen“ oder Bildern transformiert, und Ihr Lingo-Skripting, die Funktionen Ihrer Buttons etc. implementieren die gemäß Ihrer Navigationsstruktur notwendigen (nichtlinearen) Sprünge.
- Umsetzung in Lingo: Sie suchen nach einem Modell, das Ihre Navigationsstruktur abbilden kann: Für das Labyrinth könnte das beispielsweise ein zweidimensionales Raster sein, das Wege und Hindernisse beschreibt. Sie bilden das Modell in Lingo ab: z.B. mit dem Datentyp Liste. Sie generieren anhand der Lingo-Umsetzung Ihres Modells die Sprungadressen zu „Szenen“ im Drehbuch, oder Sie bleiben gleich auf einem Frame und erzeugen auch die Bildänderungen per Skript.

Für beide Wege finden Sie Experimente in diesem Kapitel. Mischformen sind natürlich möglich und nötig, schließlich wollen Sie nicht vollständig auf die Arbeitshilfen, die Director bietet, verzichten (und Ihre Anwendung vollständig „in Code“ erstellen), und vielleicht wollen Sie auch nicht jedem Button Ihrer Anwendung manuell ein Sprungziel zuweisen, wenn die Anwendung komplexer wird.

8.1 Experiment 1: Was sich hinter dem Klingelbrett verbirgt...

Dieses Experiment soll eine einfache hierarchische Struktur im Drehbuch umsetzen: Von einem Startscreen (dem Klingelbrett) geht es in fünf Folgescreens (Wohnräume). Von allen Wohnräumen aus kann man wieder zum Klingelbrett zurückkehren oder zu einem Schluss-Screen springen.

Versuchsaufbau

Der 260 x 220 Pixel große Film *Szenenwechsel.dir* hat sechs Szenen: „markero“, „marker1“, „marker2“ bis „marker 5“ und eine Schluss-Szene „marker6“, die betreten wird, wenn der User aufhören will. In Szene „markero“ liegt im ersten Spritekanal eine Hintergrundfläche mit Klingelschildern und fünf Klingelknöpfen (Sprite 2-6). In den Szenen „marker1“

bis „marker5“ befinden sich verschiedene Innenraum-Abbildungen im zweiten Spritekanal sowie ein Button zum Zurückkehren in die Eingangsszene „markero“ (Sprite 3) und ein Button, der in die Schlussszene „marker6“ (Sprite 4) führt. Dort findet sich ein Kommentar zum Aufhören und die Schluss-Taste (Sprite 7). Drückt der User diese Taste, schaltet das Behavior „Schluss“ den Film aus:

```
on mouseUp
  halt
end
```

Der Befehl `halt` sorgt nur für eine Unterbrechung des Abspielens. Director wird nicht ausgeschaltet, und Sie können den Film weiterbearbeiten. Wird der Film im Projektor abgespielt, so beendet `halt` den Projektor.

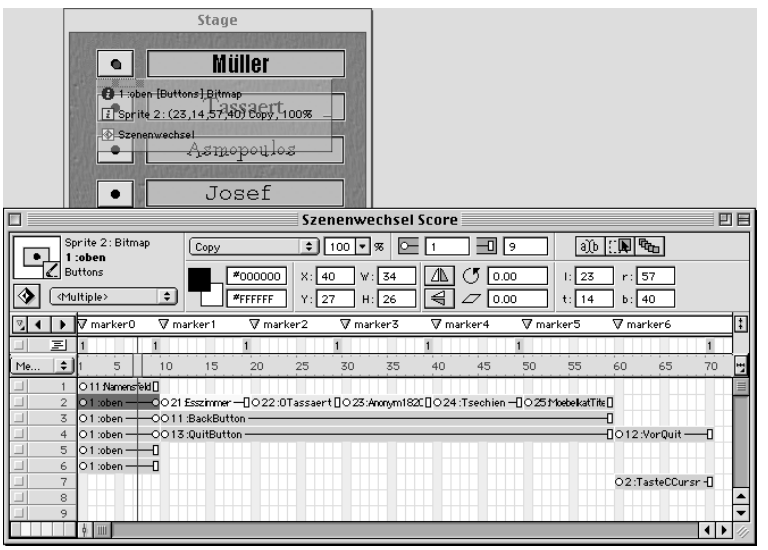


Abbildung 8.4: Bühne und Drehbuch unseres Beispielfilms

Bevor Sie die Funktionalität der anderen Buttons mit Skripten organisieren, einige Hinweise, wie Sie die Klingelknöpfe auf der Bühne aufreihen und für Buttons gegebenenfalls Extra-Besetzungen einrichten. Außerdem erfahren Sie, wie Sie Cursor-Varianten in Director erstellen.

Ausrichten

Haben Sie viele kleinteilige Elemente auf der Bühne anzuordnen, ist das Ausrichtungs-Werkzeug Strg / ⌘+K hilfreich. Die betreffenden Sprites müssen ausgewählt sein, dann können Sie über die PopUp-Menüs des Werkzeugs verschiedene Ausrichtungsvarianten wählen.

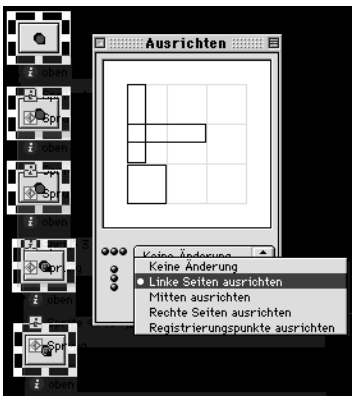


Abbildung 8.5: Gehen Sie ins Menü Modifizieren/Ausrichten*, um das Ausrichten-Werkzeug zu öffnen. In Director 8 bietet das Werkzeug noch weitere Optionen.*

- **Darstellerverwaltung:**

Haben Sie serienweise Buttons und Schaltflächen in einem Film zu verwalten, ist es angebracht, für die verschiedenen Darsteller-Typen Extra-Besetzungen anzulegen.

Abbildung 8.6: Mit dem Menü Datei/Neu/Besetzung *können Sie Besetzungen einrichten und Ordnung in Ihrer Darsteller-Sammlung schaffen.*

In Ihren Skripten müssen Sie beim Verweis auf einen Darsteller seinen Standort in einer bestimmten Besetzung berücksichtigen. Lingo kann in verschiedenen Formulierungen auf einen Darsteller verweisen. Die folgenden Zeilen meinen alle Darsteller Nr. 6, „blau“, in der zweiten Besetzung „Buttons“:

```
member("blau") -- *1
member(6,2)
member("blau",2)
member(6, "Buttons")
member 6 of castLib 2
member "blau" of castlib 2
member 6 of castLib "Buttons"
member(131078) -- *2
```

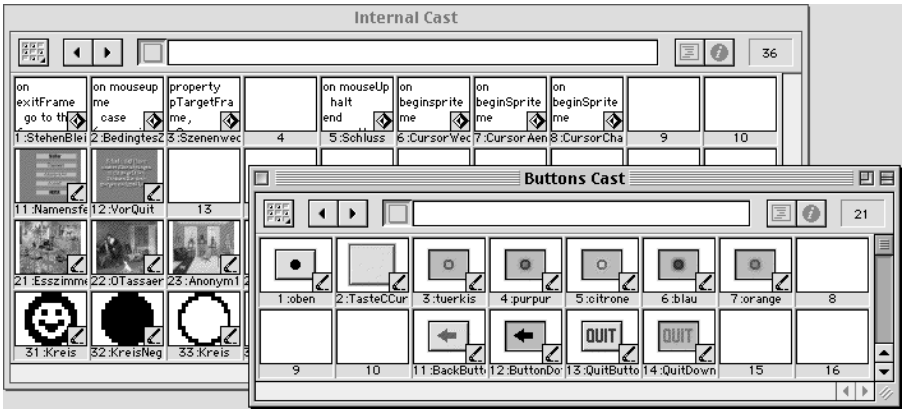


Abbildung 8.7: Mehrere Castlibs erhöhen die Übersichtlichkeit.

Die mit *1 markierte Schreibweise ist nur dann zuverlässig, wenn kein zweiter Darsteller mit diesem Namen existiert – Director wird stets den ersten zutreffenden Darsteller auswählen. Die Schreibweise *2 nutzt die Eigenschaft `member(m).number`, also die interne Nummer des Darstellers. Diese Nummerierung ist konsistent über alle Besetzungen eines Films hinweg und eignet sich daher, um relative Darstellerpositionen anzugeben wie in dem folgenden Skriptausschnitt:

```
m_nl = sprite(me.spritenum).member
m_hi = member(m_nl.number + 1)
```

Hier ist `m_nl.number + 1` eine solche interne Darstellernummer. Denselben Darsteller (mit Angabe der Castlib) bezeichnet

```
m_hi = member(m_nl.memberNum + 1, m_nl.castlibNum)
```

also der folgende Darsteller in derselben Castlib wie `m_nl`.

▶ **Cursor:**

Fünf Cursor-Formen tauchen im Beispielfilm *Szenenwechsel.dir* auf:

- der Standard-Cursor (Pfeil)
- der Zeigefinger-Cursor über den Tasten 1, 2, 4
- der Bleistift-Cursor über Taste 3
- ein selbst gebauter Cursor (Custom-Cursor) über Taste 5 (Smiley)
- ein Totenkopf-Cursor über der Schluss-Taste in der Schluss-Szene

Kommt der User in der Schluss-Szene mit der Maus über die Schluss-Taste, den Quit-Button, nimmt der Cursor eine Totenkopf-Form an. Wie beim Smiley-Cursor über Taste 5 handelt es sich dabei um einen selbst gebauten Cursor. Das sind 16 mal 16 Pixel große 1-Bit-Darsteller, die mit dem Bleistift im Malfenster Directors gezeichnet wurden. 1-Bit-Darsteller bestehen nur aus der Information derjenigen Bildteile, die eine Form bezeichnen. An den unbezeichneten Stellen kann der Untergrund durch-

scheinen. Möchten Sie das bei einem selbst gestalteten Cursor vermeiden, maskieren Sie diese Stellen. Eine Maske besteht eben aus genau denjenigen Teilen der Form, die den Blick auf den Untergrund freigeben. Bitmaps können Sie in Director mit Hilfe von Bitmaps transformieren... im Modifizieren-Menü in 1-Bit-Darsteller umwandeln:

Abbildung 8.8:
Wählen Sie aus dem Menü Modifizieren/ Bitmaps transformieren..., um 1-Bit-Darsteller aus Bitmaps zu erzeugen.

Schauen Sie sich dazu die beiden Elemente für den Cursor von Taste 5 im Malfenster an. Öffnen Sie das Besetzungsfenster und doppelklicken Sie auf den Darsteller Nummer 31 (klicken Sie im Malfenster oben in die Pfeil-Rechts-Taste, um auf den Masken-Darsteller – Nr. 32 – zu wechseln): Im Malfenster erscheinen daraufhin die Cursor.

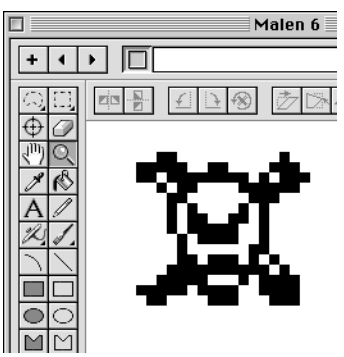


Abbildung 8.9:
Ein Doppelklick auf Bild-Darsteller öffnet die Darsteller im „Malfenster".

Es wird Ihnen aufgefallen sein, dass der Cursor, der über der Taste in der Schluss-Szene erscheint, ohne Maske angefertigt ist.

Zur Arbeit an Custom-Cursors im Malfenster sind Sie auf die Zoomfunktion angewiesen. Wenn Sie im Malfenster die Zoomeinstellung mit dem Lupenwerkzeug verändern, erscheint Ihr Objekt – neben der eingezoomten Wiedergabe – im Fenster oben rechts in normaler Größe. Um wieder auszuzoomen, klicken Sie in das kleine Fenster mit der Gesamtdarstellung.

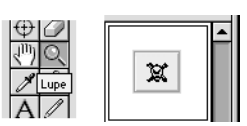


Abbildung 8.10:
Das Lupen-Werkzeug verwenden Sie zum Einzoomen im Malfenster (links). Ausgezoomt wird mit einem Klick auf die Gesamtansicht (rechts).

Durchführung

Für den Ausbau der Tasten mit Funktionalität für den Sprung zu neuen Szenen/Markierungen und für die optischen Änderungen (Cursoränderung und Darstellerwechsel) bei verschiedenen Mausaktionen bietet Director verschiedene Lösungen an, deren Wahl sich zu bedenken lohnt – insbesondere dann, wenn Sie eine ganze Serie von ähnlichen Buttons erstellen. Wir nutzen hier zwei einfachere Behaviors als die in Kapitel 7 entwickelten. Selbstverständlich können Sie dieses Experiment aber auch mit dem „Button4"-Skript umsetzen. Im Beispielfilm *Szenenwechsel.dir* sind zwei Varianten vorgestellt.

Die erste finden Sie im Behavior „BedingtesZiel" auf den Rückkehr-Buttons der einzelnen Szenen, die andere im Behavior „Szenenwechsel" auf den Klingelknöpfen der Eingangsszene (Sprite 2-6).

▶ **Spritenummer für Sprungziele nutzen:**

Das Behavior „BedingtesZiel" unterscheidet verschiedene Situationen – für den Fall, dass ein `mouseUp` auf Sprite 3 oder 4 stattfindet, veranlasst es einen Wechsel auf die Szenen markero bzw. marker6:

```
on mouseup me
  case (me.spritenum) of
    3: pZielszene = "marker0"
    4: pZielszene = "marker6"
  end case
  go pZielszene
end
```

▶ **Buttonparameter über Einstellungsdialog konfigurieren:**

Vergleichen Sie dazu die ähnliche Lösung in Kapitel 7

Ein grundsätzlich anderer Weg ist die Auswahl der Sprungziele (Marker), der Austausch-Darsteller und der Sprite-Cursor mit einem Parameterdialog. Hier ist eingangs ein gewisser Mehraufwand erforderlich. Sie generieren ein Dialogfeld, welches Ihnen später die Arbeit erleichtert, wenn Sie das Behavior auf eine gewisse Anzahl an Buttons ziehen. Das heißt, Sie schreiben ein einziges Behavior, das für alle Buttons gültig ist, und können im Dialogfeld individuelle Zuweisungen für die Properties des Behaviors vornehmen. Alle über einen solchen Parameterdialog zu konfigurierenden Eigenschaften müssen daher im Behavior als Properties deklariert werden. Mit Hilfe des Handlers `getPropertyDescriptionList` gestalten Sie Ihr Dialogfeld.

```
property pButtonStandard
property pZielszene, pCursor, pButtonGedrueckt -- GPDL
```

In den ersten Zeilen legen Sie die Eigenschaftsvariablen fest, die von allen folgenden Event-Handlern verwendet werden können. Die Werte für Properties in der zweiten Zeile kommen vom getPropertyDescriptionList*-Handler.*

```
on beginSprite me
  sprite(me.spritenum).cursor = pCursor
  pButtonStandard = sprite(me.spritenum).member
```

Zu Beginn wird in die Eigenschaftsvariable pButtonStandard *die Darstellerreferenz des Darstellers geschrieben, der auf dem Sprite liegt, auf dem das Behavior platziert ist.*

```
end

on Szenenwechsel me
```

Der selbst definierte Handler Szenenwechsel *löst die Bewegung zum Marker* pZielszene *aus.*

```
  go to frame pZielszene
end

on mouseDown me
  sprite(me.spriteNum).member  = pButtonGedrueckt
```

Darstellerwechsel bei mouseDown*. Welcher Darsteller angezeigt wird, hängt von Ihrer Auswahl im Parameter-Dialog ab.*

```
end

on mouseUpOutside me
  sprite(me.spriteNum).member = pButtonStandard
```

Beim Loslassen der Maus außerhalb des Buttons wird auf den Standard-Button zurückgestellt. Das gibt dem User die Chance, seine Absicht zu ändern, nachdem er die Maus schon gedrückt hat.

```
end mouseLeave

on mouseUp me
```

Beim Loslassen der Maus wird auf pButtonStandard *zurückgewechselt und ein Signal an den selbst definierten Handler* Szenenwechsel *ausgegeben. Damit die Bildänderung noch vor dem Sprung sichtbar wird, wird die Bühne mit* updateStage *neu gezeichnet.*

```
  sprite(me.spriteNum).member = pButtonStandard
  updateStage
  Szenenwechsel me
end
```

```
on getpropertydescriptionlist me
  Diese Zeile bewirkt, dass beim Erstellen der Szene im
  Autorenmodus ein Dialogfeld aufgerufen werden kann, mit
  dessen Hilfe Sie die Eigenschaften der Buttons je nach
  Sprite gesondert setzen.
  mlist = [:]
  Mit dieser Zeile wird eine leere Property-Liste angelegt.
  mlist[#pZielszene] = [#comment: "Einen Marker wählen:", \
   #format: #marker, #default: VOID]
  Für die Property pZielszene werden der Dialogtext (#comment),
  das Format (das in diesem Fall dafür sorgt, dass ein Aufklapp-
  menü aller Marker angezeigt wird) und der Standardwert
  (#default) angegeben.
  mlist[#pCursor] = [#comment: "Einen Cursor wählen:", \
   #format: #cursor, #default:"Arrow"]
  #format:cursor bewirkt, dass ein Aufklappmenü aller zur
  Verfügung stehenden Systemcursor angezeigt wird. Obwohl die
  Anzeige in der Sprache Ihrer Director-Version erfolgt, ist
  als Standardwert der englische Begriff einzutragen.
  mlist[# pButtonGedrueckt] = [#comment: "Gedrückter Button:", \
   #format: #graphic, #default: VOID]
  #format: #graphic zeigt im Aufklappmenü die Darstellernamen
  (und falls ein Darsteller keinen Namen hat, eine Darsteller-
  referenz in der Form member 35 of castlib 1) an. Es werden
  nicht alle Darsteller aufgelistet, sondern lediglich solche,
  die als Sprite benutzt werden können. Stattdessen könnten Sie
  auch die Formate #member (mehr Einträge) oder #bitmap (weni-
  ger Einträge) verwenden.
  return mlist
end
```

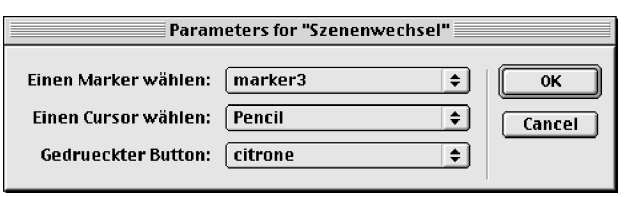


Abbildung 8.11: Wie Sie es in *getPropertyDescriptionList* konfiguriert haben, erscheinen im Dialogfeld PopUp-Menüs, mit denen Sie Sprungziel, Cursor und Erscheinung des gedrückten Buttons auswählen.

Abbildung 8.12: Die System-Cursor in Director

Möchten Sie eine einmal entschiedene Kombination ändern, wählen Sie das betreffende Sprite aus, markieren das dazugehörige Skript im Verhaltens- oder Eigenschaftsinspektor und klicken das Parameter-Icon an (vgl. Abbildung 8.13).

Abbildung 8.13: Parameter-Icon im Verhaltensinspektor von Director 7 und 8 (dort auch im Eigenschaftsinspektor)

Ihr selbst definiertes Dialogfenster erscheint und Sie können Änderungen vornehmen. Hier das Dialogfenster im Einsatz:

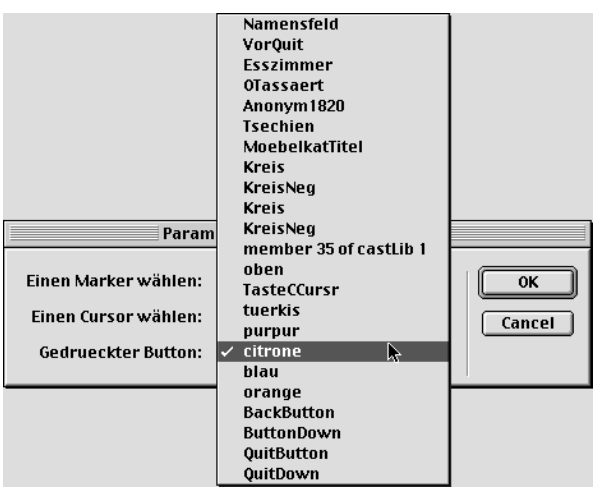


Abbildung 8.14: Ein Klick auf das Parameter-Icon im Verhaltensinspektor öffnet den Custom-Dialog „Parameter“; hier mit geöffnetem PopUp-Menü für die Buttonversion.

Zieht man das gleiche Behavior auf alle Button-Sprites, wird bei jedem Sprite einen Dialog geöffnet, in welchem der Status des gedrückten Buttons gesondert benannt und das Sprungziel sowie die Cursorform gesondert ausgewählt werden können.

- **Weitere kleine Skripte:**

Das Auswechseln der Cursor über den Tasten 1-4 wird von den Einträgen in den Parameter-Dialogen für diese Tasten geregelt, weil die Cursor-Form in der PropertyDescriptionList aufgenommen ist. Das ist mit Custom-Cursors nicht möglich. Der Cursor der Taste 5 in Sprite 6 wird durch das Behavior „CursorWechsel“ ausgetauscht; dabei wird der maskierte Cursor mit zwei Darstellernummern bezeichnet:

```
on beginSprite me
  sprite(me.spriteNum).cursor = [31,32]
```

Die erste Ziffer steht für die Darsteller-Nummer in der Besetzung, die zweite Ziffer für die darauf folgende Masken-Nummer in der Besetzung. Stattdessen können Sie auch Darsteller-Bezeichnungen verwenden: `sprite(me.spriteNum).cursor = [member(31), member(32)].`

```
end
```

Das Behavior „Schluss“ für den unmaskierten Cursor, der beim Rollover der Schluss-Taste (Sprite 7) erscheint, zeigt den Totenkopf-Cursor an:

```
on beginSprite me
  sprite(me.spriteNum).cursor  = [35]
end
```

8.2 Experiment 2: Süßes und Saures

Hier wollen wir zwei neue Navigationsmöglichkeiten entdecken: Zum einen nutzen wir zwei Filme und springen von einem zum anderen. Zum anderen wollen wir überprüfen, ob der Anwender tatsächlich alle Sprungziele des ersten Films gesehen hat, wenn er zum zweiten wechseln will, und ihm eine entsprechende Meldung anzeigen, falls das nicht der Fall sein sollte.

Das Experiment kombiniert lineare Elemente (Abfolge von Filmen) mit einer einfachen Hierarchie im ersten Film.

Versuchsaufbau

Beide Filme, *Startfilm.dir* und *Folge.dir*, sind 260 x 220 Pixel groß. Beide Filme enthalten mehrere Szenen.

Startfilm.dir beginnt mit einer Szene, auf welcher dem User Bonbon, Törtchen und Praline zum Anklicken angeboten werden. Er kann zu diesen Szenen springen und von dort aus zurück auf die Süßigkeiten-Szene. Hat er genug, klickt er auf den Button „Sauer“, der auf den zweiten Film *Folge.dir* verzweigt.

Falls er noch nicht alles gesehen hat, wird er gefragt, die noch nicht angeklickten Süßigkeiten auszutesten. Dafür wird ein Textfeld mit entsprechendem Text gefüllt. Wenn er das nicht möchte, kann er auf den mit der Nachfrage erschienenen „Nee-Button“ klicken und der zweite Film erscheint.

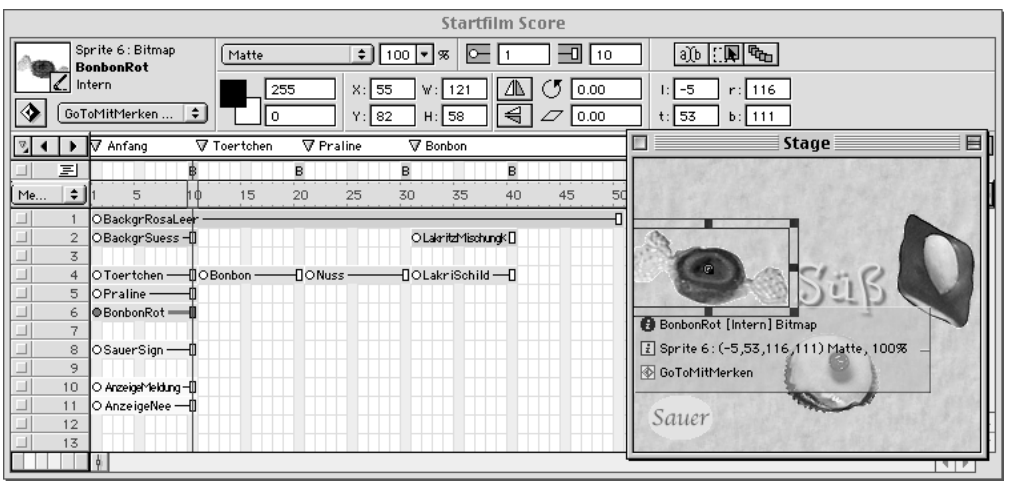


Abbildung 8.15: Drehbuch und Bühne im startfilm.dir

Im zweiten Film *Folge.dir* hat der User in der ersten Szene die Auswahl unter Saurem: Gurke, Fischbrötchen und Zitrone. In der zweiten Szene soll eine Textanzeige erscheinen, die ihm mitteilt, worauf er zuerst geklickt hat.

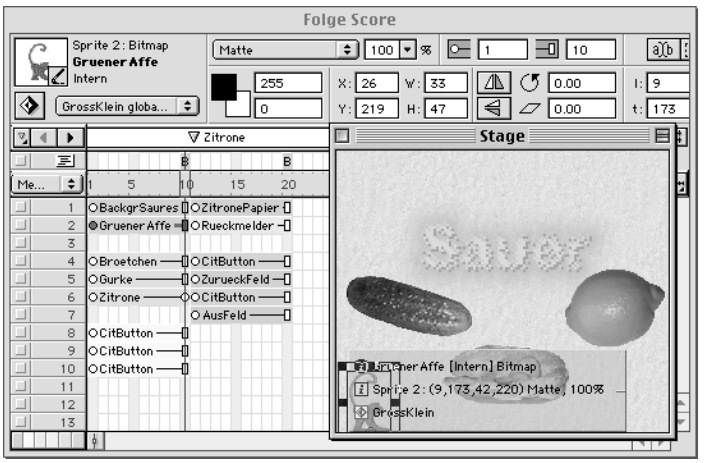


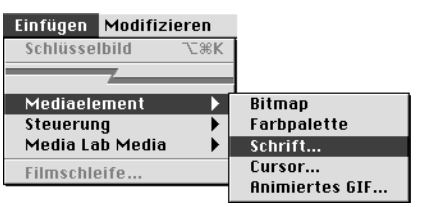
Abbildung 8.16: Drehbuch und Bühne in Folge.dir

Der generelle Aufbau der Szenen folgt den gleichen Prozeduren wie in den vorausgegangenen Versuchen. Damit zwischen verschiedenen Szenen gewechselt werden kann, haben wir mehrere Szenen aus Hintergrundflächen und verschiedenen Anklick-Objekten angelegt. Wir beschränken die ausführliche Beschreibung hier auf die Neuerungen:

- die Einrichtung eines Gedächtnisses
- unterschiedliche visuelle Rückmeldungen auf die User-Aktivität über Schaltflächen

Die Hinweise in den Textdarstellern sind in beiden Filmen etwas komfortabler ausgebaut als in den vorangehenden Experimenten. So ist der Text mit einer Schrift wiedergegeben, die dem Film als Darsteller beigefügt ist. Damit ist die Schrift verfügbar, ohne dass der User sie auf seinem Rechner installiert haben muss. Eingebettete Schrift und Originalfonts werden allerdings nicht immer identisch reproduziert. Ihre Texte sollten Sie daher unbedingt mit den importierten Fonts erstellen und auf Abweichungen testen.

Abbildung 8.17: Zum Import von Fonts gehen Sie unter Einfügen/Media-Element/Schrift



Einige der Textdarsteller geben Text farbig wieder. Das können Sie im Text-Inspektor (Menü Fenster/Inspektoren/Text-Inspektor) so veranlassen:

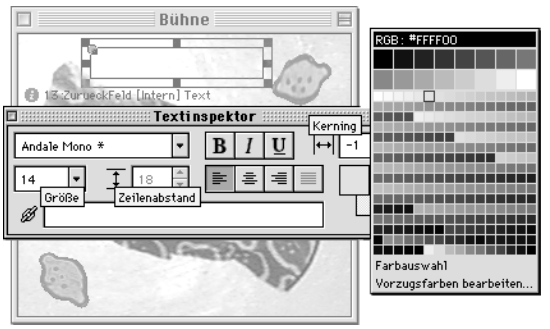


Abbildung 8.18: Den Text-Inspektor öffnen Sie auch mit Strg /⌘ + T.

In verschiedenen Eingabefeldern können Sie Schriftart, Größe, Zeilenabstand, Buchstabenweite und Schriftfarbe Ihren Wünschen entsprechend eingeben.

Durchführung

Vom Beispielfilm *Startfilm.dir* mit süßem Inhalt soll zu *Folge.dir* mit saurem Inhalt gewechselt werden. Ein solcher Filmwechsel lässt sich leicht mit folgenden Skriptzeilen herbeiführen:

```
on mouseUp
  go to movie "Folge.dir"
end mouseUp
```

Vor dem Themenwechsel soll gefragt werden, ob der User schon auf allen Süßigkeiten-Szenen war (Bonbon, Praline, Törtchen).

Damit Director sich die Wege des Users in einem Film merken kann, benötigen Sie ein „Gedächtnis“, eine Instanz, die den ganzen Film über gültig ist und die benötigten Zustände speichern kann. Wie auch schon in Kapitel 7 (Experiment 2 und 4) verwenden wir dafür wieder eine globale Variable. Mit dieser Variablen wollen wir überprüfen, ob der Anwender bereits alle drei Süßigkeiten angeklickt hat (dann geht es weiter zu *Folge.dir*). Außerdem wollen wir einen Hinweis zeigen, der ihm sagt, wo er noch hingehen könnte: Wir überprüfen also, worauf der User noch nicht geklickt hat, und zeigen einen entsprechend aussagekräftigen String im Textfeld an.

Die Information, welche Elemente angeklickt wurden, könnten wir als lineare Liste speichern: `[1, 1, 0]` könnte beispielsweise besagen, dass das erste und zweite Element angeklickt wurden, das dritte aber nicht. Wir müssten uns dann eine Konvention überlegen, welches Element der Liste zu welchem Sprite (Törtchen, Bonbon, Praline) gehört.

Eine Alternative ist die Speicherung in einer Propertyliste. `[#bonb: 1, #tort:1, #pral:0]` enthält die gleichen Informationen wie die lineare Liste, aber jede Information ist einer Property zugeordnet. Dies ist offensichtlich ein komfortablerer Weg.

```
on preparemovie
  global gSceneList
  gSceneList = [#bonb: 0, #tort:0, #pral:0]
end
```

Damit weisen wir beim Filmstart der globalen Variablen `gScenelist` eine Propertyliste mit den Ausgangswerten (alle nicht angeklickt, also `0`) zu.

Bei einem Buttonklick setzen wir eine bestimmte Property der Liste (hier für das Bonbon):

```
on mouseUp me
  global gSceneList
  gSceneList.bonb = 1
end
```

`gSceneList.bonb = 1` lässt sich auch als `gSceneList[#bonb] = 1` schreiben. Um zu überprüfen, ob das Bonbon schon geklickt wurde, fragen wir `if gSceneList.bonb = 1 then`; um alle drei Bedingungen zu überprüfen, verbinden wir sie mit einem logischen `AND`:

```
if (gSceneList.bonb AND gSceneList.tort) AND gSceneList.pral
 then ...
```

oder wir addieren die Propertywerte:

```
if gSceneList.bonb + gSceneList.tort + gSceneList.pral = 3
 then ...
```

In beiden Fällen wissen wir, dass alle drei Süßigkeiten angeklickt wurden, wenn die Gesamtbedingung `TRUE` ergibt.

Im Beispielfilm wird `gSceneList` in einem Filmskript initialisiert, d.h. auf die Ausgangswerte gesetzt. Außerdem werden hier die beiden Textdarsteller leergeräumt:

```
global gSceneList

on preparemovie
  gSceneList = [#tort: 0, #bonb: 0, #pral: 0]
  member("AnzeigeMeldung").text = ""
  member("AnzeigeNEE").text = ""
end
```

Folgendes Skript liegt auf den Süßigkeiten-Sprites:

```
global gSceneList
property myname

on beginSprite me
  case me.spriteNum of
```

Das Sprite erhält abhängig von der Spritenummer einen Bezeichner, der auch als Property in `gSceneList` *dienen soll.*

```
    4: myname = #tort
    5: myname = #pral
    6: myname = #bonb
  end case
end

on mouseUp
  gSceneList[myname] = 1
    Die gSceneList wird an der für das Sprite passenden
    Position auf 1 (geklickt) gesetzt.
  case myname of
    Hier folgen die Sprungbefehle.
    #tort: go "Toertchen"
    #pral: go "Praline"
    #bonb: go "Bonbon"
  end case
end mouseDown
```

Die globale `gSceneList` wird beim Klick auf den Button „Sauer“ ausgewertet. Er trägt folgendes Behavior:

```
global gSceneList

on mouseUp
  if (gSceneList[#tort] + gSceneList[#pral] + \
   gSceneList[#bonb]) = 3 then
    Wenn alle drei Werte auf 1 stehen, können wir direkt zum
    Folgefilm springen.
    member("AnzeigeMeldung").text = ""
    member("AnzeigeNEE").text = ""
    go to movie "Folge.dir"

  else
    Andernfalls setzen wir den Textdarsteller "AnzeigeNEE" auf
    „Nee" (und machen ihn damit sichtbar und anklickbar für ei-
    nen Direktsprung zu Folge.dir) und konstruieren sukzessive
    einen Ausgabetext, der den Anwender auf die verpassten
    Süßigkeiten hinweisen soll.
    member("AnzeigeNEE").text = "nee"
    myreturnstring = "Sie wollten doch eigentlich noch "
    filler = ""

    if gSceneList[#tort] <> 1 then
      myreturnstring = myreturnstring & filler & "ein Törtchen"
      filler = " und "
    end if
```

```
    if gSceneList[#pral] <> 1 then
      myreturnstring = myreturnstring & filler & "die Praline"
      filler = " und "
    end if

    if gsceneList[#bonb ] <> 1 then
      myreturnstring = myreturnstring & filler & "Bonbons"
    end if
    myreturnstring = myreturnstring & " probieren ?!?!"
    Schließlich wird der Ausgabestring dem Textdarsteller
    "AnzeigeMeldung" zugewiesen.
    member("AnzeigeMeldung").text = myreturnstring
  end if
end
```

Hat der Anwender alle Süßigkeiten angeklickt oder klickt er nach der Meldung auf den Button „Nee“, so wird der Sprung in den Film *Folge.dir* ausgelöst. Was geschieht mit unserer globalen Variable `gSceneList` mit diesem Filmwechsel? Die Antwort ist einfach: Gar nichts, sie steht im nächsten Film weiter zur Verfügung und ist auch noch vorhanden, wenn wir von dort in den *startfilm.dir* zurückspringen. Dann allerdings greift wieder der `prepareMovie`-Handler von oben, in dem `gSceneList` auf den Ausgangswert gesetzt wird. In diesem Fall ist das wohl sinnvoll – es wäre aber auch denkbar, diese Neuinitialisierung zu unterlassen, um die in `gSceneList` gespeicherten Zustände weiterverwenden zu können.

In diesem Fall würden Sie im `prepareMovie`-Handler statt

```
gSceneList = [#tort: 0, #bonb: 0, #pral: 0]
```

eher die Zeile

```
if voidP(gScenelist) then gScenelist = [#tort: 0, #bonb: 0,
#pral: 0]
```

benutzen: Nur, wenn `gSceneList` noch `VOID` ist, wird die Variable initialisiert. Wenn sie, auch nach einem Rücksprung aus einem anderen Film, bereits existiert, bleibt sie unverändert.

Der Fortsetzungsfilm dient einerseits als Sprungziel, er soll andererseits aber auch eine weitere Form von „Gedächtnis“ im Director-Film illustrieren. Wundern Sie sich nicht, wenn Sie beim Sprung von einem zum anderen Film eine Aufforderung zum Sichern erhalten: Das ist beim Authoring normal, wird aber beim Abspielen im Projektor oder als Shockwave nicht mehr auftauchen.

In der ersten Szene von *Folge.dir* soll ermittelt werden, worauf der User zuerst geklickt hat und wie oft geklickt wurde.

```
global gKlickCount

on startmovie
  gKlickCount = 0
end
```

In einem startMovie*-Skript wird die Variable* gKlickCount*, die später die Summe der Klicks auf die unterschiedlichen Buttons enthalten soll, initialisiert, auf Null gestellt.*

Auf den drei anklickbaren Sprites (Fischbrötchen, Zitrone und Gurke) wird mit folgendem Behavior die Variable `gKlickCount` bei jedem Klick um `1` erhöht und beim allerersten Klick ein Sprite-spezifischer Meldungstext in den Darsteller „Rueckmelder“ geschrieben. Dieser Textdarsteller ist erst in der zweiten Szene des Filmes sichtbar – auch ein Textdarsteller kann also als Behälter für Dinge dienen, die der Film „im Gedächtnis behalten“ soll.

```
global gKlickCount

on mouseUp me
  gKlickCount = gKlickCount + 1
  if gKlickCount = 1 then
    Beim ersten Klick wird...
    case sprite(me.spriteNum).member.name of
    ...je nachdem, welches Sprite wir angeklickt haben (das
    wird anhand des Darstellernamens des Sprites überprüft)...
      "Zitrone": mytext = "Sie haben zuerst auf die Zitrone \
       geklickt"
      "Broetchen": mytext = "Sie haben zuerst auf das Brötchen \
       geklickt"
      "Gurke": mytext = "Sie haben zuerst auf die Gurke \
       geklickt"
    end case
    member("Rueckmelder").text = mytext
    ...ein Meldungstext erzeugt und in den Darsteller
    „Rueckmelder" geschrieben.
  end if
  sprite(me.spriteNum).blend = 60
  sprite(me.spriteNum + 4).loc = \
   point(40 + (me.spriteNum)*30, 23)
    Während die anklickbaren Sprites in den Kanälen 4, 5 und 6
    liegen, gibt es in den Kanälen 8, 9 und 10 drei kleine Zi-
    tronen-Sprites, die zunächst außerhalb der Bühne platziert
    werden. Bei einem Klick auf Sprite 4, 5 oder 6 wird jeweils
    eines der Zitronen-Sprites ins Blickfeld gerückt, indem
    seine loc-Eigenschaft gesetzt wird (vgl. Abbildung 8.19).
end mouseUp
```

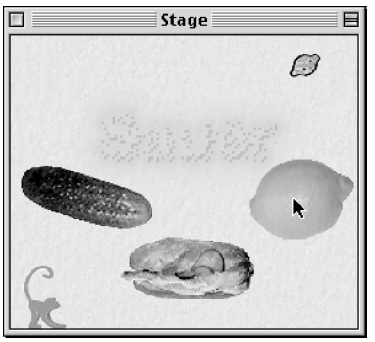


Abbildung 8.19:
Ein Klick auf eines der unteren Sprites reduziert die Deckkraft und zeigt am oberen Rand eine kleine Zitrone an.

Angeklickte Darsteller werden dem User durch eine verminderte Deckkraft des Elements auf der Bühne visuell verändert präsentiert. Auf diese Weise wird klar, wo der Anwender schon war.

Um von der Eingangsszene des Fortsetzungsfilms auf seine zweite Szene zu gelangen, dient eine Figur unten links als Button. Beim Rollover macht sie auf sich aufmerksam durch eine leichte Vergrößerung. Betätigt man diesen Button direkt, ohne auf die anderen Objekte geklickt zu haben, wird in der zweiten Szene eine Frage gestellt. Das folgende Behavior liegt auf der Figur:

```
global gKlickCount
property myrect

on beginSprite me
  myrect = sprite(me.spritenum).rect
```

In der Property myrect *merkt sich das Skript die Ausgangsgröße des Sprites.*

```
end

on mouseEnter me
  sprite(me.spritenum).rect = myrect + rect(-2, -4, 2, 0)
```

Kommt die Maus über das Sprite, wird ein einfacher visueller Effekt erzeugt, indem das rect *des Sprites um 2 nach links, um 4 nach oben und um 2 nach rechts erweitert wird.*

```
end

on mouseLeave me
  sprite(me.spritenum).rect = myrect
```

Verlässt die Maus das Sprite, so wird wieder die Ausgangsgröße hergestellt.

```
end
```

```
on mouseUp me
  if gKlickCount = 0 then
    Falls noch nicht geklickt wurde, wird der Meldungstext der
    nächsten Szene verändert.
     member("Rueckmelder").text = "Warum haben Sie nicht \
      geklickt!?"
  end if
  go to "Zitrone"
    Und schließlich springen wir zur nächsten Szene.
end
```

8.3 Experiment 3: Netzartige Verzweigungen – ein erster Versuch im Drehbuch

Dieser Versuch soll zeigen, wie Sie netzartige Verzweigungen im Drehbuch realisieren können. In Experiment 4 sehen Sie dann eine andere Lösung, die das Drehbuch weit weniger nutzt als unser jetziges Experiment und weitgehend lingo-basiert ist.

Hier versuchen wir, die Navigationsstruktur (vgl. Abbildung 8.20) durch Sprünge zu Markern im Drehbuch umzusetzen. Von einem „zentralen“ Punkt aus („Hauptmenue“, das weiße Karree in der Mitte) geht es in vier „Richtungen“ weiter („A“, „B“, „C“, „D“). Von zweien dieser Punkte soll es weitere Verzweigungen geben („C1“, „C2“, „C3“, „D1“, „D2“, „D3“).

Wir wollen den Sprung dieses Mal nicht erst bei Mausklick, sondern bereits dann auslösen, wenn sich die Maus in den Randbereich der Bühne bewegt. Zusammen mit einem entsprechenden Übergangseffekt („Schieben“ mit Richtungsangabe) entsteht so der Eindruck, dass wir mit der Mausbewegung die Bühne in diese Richtung verschieben.

Auf der CD-ROM finden Sie auch einen Film (*NetzdrehbuchFirst.dir*), der eine frühe und wenig „abstrakte“ Umsetzung beinhaltet und für jeden Drehbuchsprung ein eigenes Skript verwendet. In der Durchführung dieses Experiments gehen wir einen anderen Weg; nichtsdestotrotz können Sie natürlich mit dem folgenden Versuchsaufbau auch zu einer Lösung wie in *NetzdrehbuchFirst.dir* kommen.

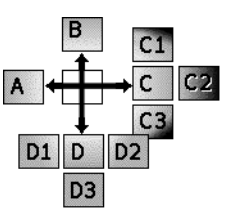


Abbildung 8.20:
Schema für die Szenenwechsel

Versuchsaufbau

In der Besetzung des Beispielfilmes *NetzDrehbuch.dir* finden Sie die Abbildungen einer Reihe von Werkzeugen, deren Darstellernamen mit den Positionen der Navigationsstruktur (vgl. Abbildung 8.20) korrespondieren.

Der erster Schritt beim Aufbau Ihres Director-Filmes ist, die Netzstruktur in eine lineare Abfolge von „Szenen“ zu übersetzen. Die Markierungen in Abbildung 8.21 zeigen Ihnen eine Möglichkeit. „Hauptmenue“ und die weiteren Marker beginnen auf Frame 11, damit zu Beginn des Filmes noch ein Eingangsscreen mit einem „Start“-Button angezeigt werden kann. Ein „Stopper“-Skript im Skriptkanal des Drehbuchs hält den Abspielkopf jeweils einige Frames nach den Markern an.

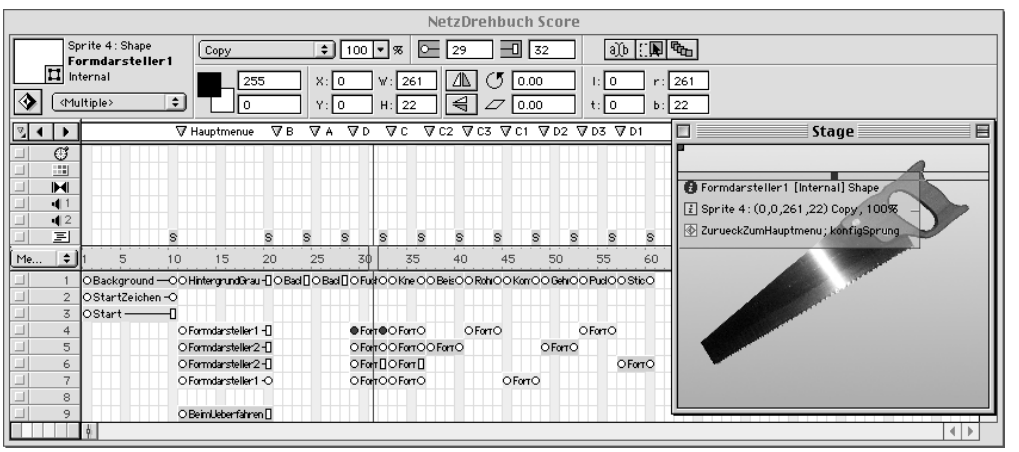


Abbildung 8.21: Drehbuch und Bühne unseres Beispielfilms

Im Sprite 1 haben wir unsere Hintergrundgrafiken (die Werkzeugabbildungen) platziert. Die Sprites 4 bis 7 sollen die unsichtbaren Rollover-Bereiche enthalten, die die Sprünge auf andere Szenen auslösen. Dafür haben wir mit dem „ungefüllten Rechteck“-Werkzeug der Werkzeugpalette zwei Rechtecke auf der Bühne aufgezogen, ein Rechteck für oben bzw. unten und eines für rechts bzw. links. Die Linienstärke stellen Sie dabei auf „unsichtbar“ (die gepunktete Linie, vgl. Abbildung 8.22).

Abbildung 8.22:
Werkzeugpalette: Unsichtbare Hotspots können Sie mit dem ungefüllten Rechteck und der Linienstärke 0 erstellen.

Im „Hauptmenue“ und in den Szenen „C“ und „D“ können Sie sich in alle vier Richtungen bewegen: Entsprechend legen Sie in alle vier Randbereiche ein entsprechendes Hotspot-Sprite. Auf den anderen Bildern („C1“ bis „D3“) sind die Bewegungsmöglichkeiten eingeschränkt: Anhand des Navigationsschemas in Abbildung 8.20 können Sie aber genau sehen, in welche Richtung eine Bewegung möglich sein muss.

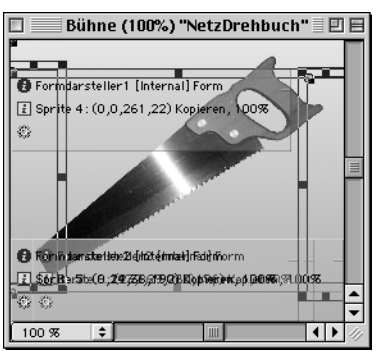


Abbildung 8.23:
In Szene „D“ sind 4 unsichtbare Hotspots anzulegen

Die Bereiche „A“ und „B“ erhalten keine unsichtbaren Hotspots: von hier geht es nur per Klick zurück zum Hauptmenü.

Legen Sie außerdem eine Transition an, indem Sie im Transition-Kanal des Drehbuchs an beliebiger Stelle doppelklicken. In Abbildung 8.24 sehen Sie die Konfiguration. Benennen Sie den Transition-Darsteller in der Besetzung mit „UP“. Duplizieren Sie ihn dreimal, und stellen Sie nach Doppelklick in der Besetzung entsprechend die anderen Schieberichtungen ein. Die Darsteller sollen „DOWN“, „LEFT“ und „RIGHT“ heißen, und die jeweilige Schieberichtung soll dieser Benennung entgegengesetzt sein. Aus dem Drehbuch können Sie die angelegte Transition dann entfernen, da wir die Bildübergänge mit Lingo aufrufen wollen.

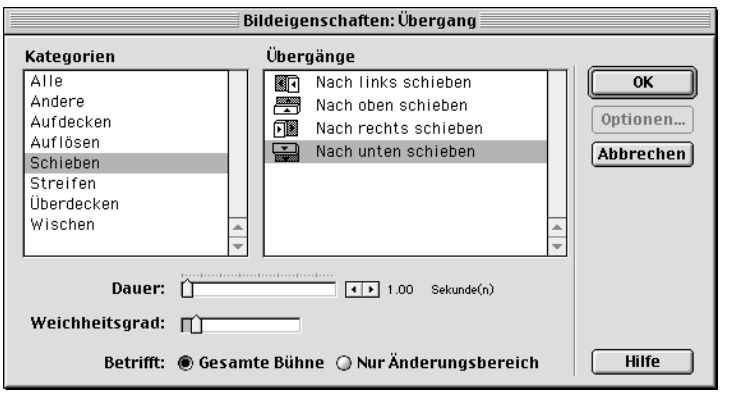


Abbildung 8.24: Die Eigenschaften des Transition-Darstellers „UP“

Durchführung

Die spannende Frage bei diesem Experiment ist, wie wir die Drehbuchsprünge und die dazugehörigen Transitions in unseren Skripten organisieren.

Eine Möglichkeit besteht darin, für jeden Sprung ein angepasstes Skript zu verfassen. Diese Lösung finden Sie im Film *Netzdrehbuch-First.dir*. Sie ist hier – mit einem guten Dutzend Skripten – gerade noch möglich. Sinnvoller ist allerdings, ein konfigurierbares Behavior zu verfassen, das für jedes der unsichtbaren Hotspot-Sprites nutzbar ist. Wir wollen dabei das Sprungziel und die Richtung, in der das Sprungziel zu finden ist, festlegen können. Den Richtungsparameter nutzen wir für die Auswahl der richtigen Transition.

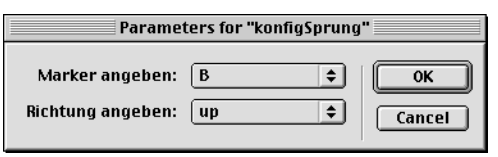


Abbildung 8.25: Konfigurationsdialog für das Behavior „konfigSprung“

Das folgende Skript erzeugt einen Einstellungsdialog wie in Abbildung 8.25 und löst bei `mouseEnter` die gewünschten Aktionen aus:

```
property pMarker, pTransition

on mouseEnter
  puppetTransition member(pTransition)
  go to pMarker
end mouseEnter

on getpropertydescriptionlist me
  mlist = [:]
  mlist[#pMarker] = [#comment: "Marker angeben:", \
   #format: #marker, #default: "Hauptmenue"]
```

```
  mlist[#pTransition] = [#comment: "Richtung angeben:", \
   #format: #string, #default: "up", \
   #range: ["up", "down", "right", "left"]]
  return mlist
end
```

Die beiden Properties des Behaviors, `pMarker` und `pTransition`, werden im `getPropertyDescriptionList`-Handler des Behaviors konfiguriert. Wir nutzen `#format: #marker`, um eine automatisch generierte Markerliste im Konfigurationsdialog anzeigen zu können. Die Transition hingegen wählen wir nicht direkt aus, sondern definieren für den Konfigurationsdialog `#format: #string` und eine Auswahl an möglichen Werten (`#range: ["up", "down", "right", "left"]`). Entsprechend benannte Transition-Darsteller haben wir bereits angelegt, so dass `member(pTransition)` den Transition-Darsteller bezeichnet – zumindest, wenn kein anderer Darsteller „up“ oder „left“ heißt!

Im `mouseEnter`-Handler werden die Hotspot-Aktionen ausgelöst: zunächst die Transition, dann der Sprung zum Marker. Diese Reihenfolge ist sinnvoll, denn eine Transition, die mit Lingo gestartet wird, wird erst beim nächsten Neuzeichnen der Bühne – also nach einem `prepareFrame`-Event – angezeigt.

Dieses Behavior ziehen Sie auf alle unsichtbaren Hotspots und konfigurieren die Sprungziele entsprechend dem Schema in Abbildung 8.20.

Um von allen Frames direkt zum Hauptmenü springen zu können, erhalten alle Hintergrundgrafiken (Sprite 1) das folgende Behavior:

```
on mouseUp me
  go to "Hauptmenue"
end
```

Den Einstieg ins Hauptmenü bewirkt der Klick auf einen der Start-Buttons zu Beginn des Filmes. Die beiden Start-Buttons reagieren auf Rollover und Klick mit unterschiedlichen Versionen der Buttonform und unterschiedlichen Sounds. Es handelt sich um ein abgewandeltes Behavior des „Buttons mit drei Zuständen“, den wir in Experiment 5 entwickelt haben. Die Veränderungen beziehen sich auf folgende Punkte:

- Da als Button-Aktion nur ein Sprung auf einen festgelegten Frame in Frage kommt, kann die Aktion direkt ins Behavior eingetragen werden.
- Konfigurierbar sind neben den Buttongrafiken für Rollover und Mousedown auch noch drei Sounds (`mouseEnter`, `mouseDown` und `mouseUp`).

Im Beispielfilm sind die Buttons unterschiedlich konfiguriert.

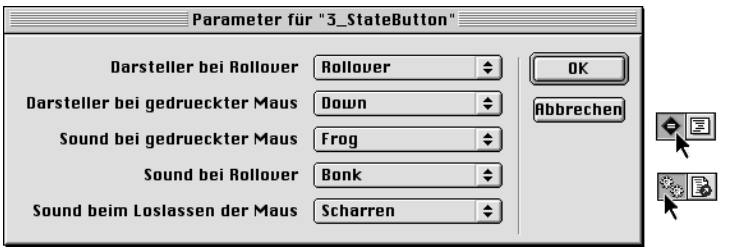


Abbildung 8.26: Das Parameterfenster für den Start-Button der Eingangsszene. Um das Fenster zu öffnen, markieren Sie das Sprite im Drehbuch und drücken dann die Parametertaste im Verhaltensinspektor (rechts in der Abbildung: oben Director 7, unten Director 8).

Die Skript-Umsetzung sieht folgendermaßen aus:

```
property pMousedDown
```

In der Property pMousedDown *wird eingetragen, ob ein* mouseDown *auf dem Button erfolgt ist.*

```
property m_nl, m_hi, m_dn
property s_dn, s_hi, s_up
```

*Für Standard- (*m_nl*), Rollover- (*m_hi*) und Down-Darsteller (*m_dn*) und die entsprechenden Sounds bei* mouseDown *(*s_dn*), bei Rollover (*s_hi*) und* mouseUp *(*s_up*) werden Property-Variablen eingerichtet.*

```
on getPropertyDescriptionList me
```

Definition des Konfigurationsdialoges. Das Skript nimmt den aktuellen Darsteller des Sprites als Ausgangspunkt und setzt die #default*-Werte für* m_hi *und* m_dn *auf die beiden folgenden Darsteller.*
Achten Sie auf die Verwendung von #format:#bitmap *für die Buttongrafiken und* #format:#sound *für die Sounds.*

```
  list = [:]
  nullDarsteller = sprite(the currentspriteNum).member.number
  list[#m_hi] = [#comment: "Darsteller bei Rollover", \
    #format: #bitmap, #default: member(nullDarsteller + 1)]
  list[#m_dn] = [#comment: "Darsteller bei gedrueckter Maus",\
    #format: #bitmap, #default: member(nullDarsteller + 2)]
  list[#s_dn] = [#comment: "Sound bei gedrueckter Maus", \
    #format: #sound, #default: ""]
  list[#s_hi] = [#comment: "Sound bei Rollover", \
    #format: #sound, #default: ""]
  list[#s_up] = [#comment: "Sound beim Loslassen der Maus", \
    #format: #sound, #default: ""]
  return list
end
```

Das folgende Grundgerüst ist dasselbe wie in Experiment 1 in Kapitel 7. Die Unterschiede werden erst mit anderen Befehlen im change*-Handler umgesetzt.*

```
on beginSprite me
  pMousedDown = 0
  m_nl = sprite(me.spriteNum).member
```

m_nl wird nicht über den getPropertyDescriptionList*-Handler konfiguriert und daher hier auf den Darsteller des Sprites gesetzt.*

```
  change me, #nl
end

on mouseDown me
  pMousedDown = 1
  change me, #dn
end

on mouseUp me
  pMousedDown = 0
  change me, #up
end

on mouseUpOutside me
  pMousedDown = 0
  change me, #nl
end

on mouseEnter me
  if pMousedDown then
    change me, #dn
  else if not(the stillDown) then
    change me, #hi
  end if
end

on mouseLeave me
  change me, #nl
end
```

change *setzt Darstellerwechsel und Sound-Aufruf um:*

```
on change me, ziel
  case ziel of
    #nl:
      sprite(me.spriteNum).member = m_nl
      puppetsound 1, 0
    #hi:
      sprite(me.spriteNum).member = m_hi
      puppetsound 1, s_hi
```

```
    #up:
      sprite(me.spriteNum).member = m_hi
      puppetsound 1, s_up
      go to "Hauptmenue"
    #dn:
       sprite(me.spriteNum).member = m_dn
       puppetsound 1, s_dn
  end case
  updateStage
end
```

Je nach Parameter führt der `change`-Handler unterschiedliche Aktionen aus: Er zeigt den passenden Darsteller an, und er spielt einen Sound im Kanal 1 (z.B. `puppetSound 1, s_up`) oder er stoppt den im Soundkanal möglicherweise spielenden Sound (`puppetSound 1, 0`). `s_up` ist dabei die Property, die über den `getPropertyDescriptionList`-Handler definiert wurde. Wird der Parameter `#up` übergeben, so löst das Skript außerdem die Buttonaktion aus, also den Sprung zum Marker `Hauptmenue`.

8.4 Experiment 4: Ein Lingo-Netz

Das folgende Experiment soll eine Netzstruktur umsetzen, die wir in Abbildung 8.27 visualisiert haben: In diesem Raster gibt es Felder mit einem schwarzen Punkt (wie beispielsweise „A1“): dies sind die Bilder, die Director anzeigen soll. Es gibt damit auch „Wege“ durch die Anwendung: Man kann von „A1“ zu „A2“ und weiter zu „A3“ gehen, aber man kann beispielsweise nicht von „A2“ zu „B2“ gehen. Wege sind also als schwarze Punkte, nicht betretbare Positionen als ungefüllte Kreise und die Startposition „B3“ als Kreuz markiert.

Keine Frage: Dieses Navigationsmodell lässt sich mit Markern ebenso umsetzen wie in Experiment 3 gezeigt. Sie legen die Marker „A1“ bis „E5“ an und nutzen das oben vorgestellte Behavior für die Navigation.

Was wir hier aber versuchen wollen, ist eine stärker lingo-orientierte Lösung: Wir wollen unser „Navigations-Schachbrett“ in eine Lingo-Struktur überführen und alle notwendigen Bildwechsel ohne Sprünge im Drehbuch realisieren. Die Lösung, zu der wir kommen werden, ist unabhängig von der Größe des Schachbretts: Ein 20x20-Raster ist damit ebenso umsetzbar wie unser 5x5-Beispiel.

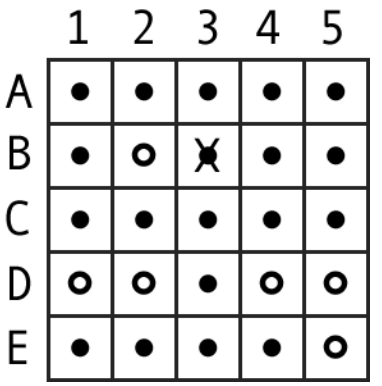


Abbildung 8.27:
Die Netzstruktur des Beispielfilms NetzLingo.dir als Wegeplan: Startposition (X), begehbare (•) und nicht begehbare Positionen (°)

Die Struktur, die in Lingo „geordnete" Daten ermöglicht, sind Listen. Wir können beispielsweise die Reihen „A" und „B" folgendermaßen darstellen, wenn wir festhalten wollen, ob dort ein gefüllter (1) oder ein ungefüllter Kreis (0) zu finden ist:

```
A = [1, 1, 1, 1, 1]
B = [1, 0, 1, 1, 1]
```

Auf die zweite Listenposition von `A` können wir so zugreifen:

```
put A[2]
-- 1
```

Wollen wir den ganzen 5x5-Plan umsetzen, so definieren wir zunächst die Reihen und fassen sie anschließend wieder in einer Liste zusammen:

```
A = [1, 1, 1, 1, 1]
B = [1, 0, 1, 1, 1]
C = [1, 1, 1, 1, 1]
D = [0, 0, 1, 0, 0]
E = [1, 1, 1, 1, 0]
plan = [A, B, C, D, E]
put plan
-- [[1, 1, 1, 1, 1], [1, 0, 1, 1, 1], [1, 1, 1, 1, 1],
   [0, 0, 1, 0, 0], [1, 1, 1, 1, 0]]
```

Wie kommen wir nun an die vierte Reihe und darin an die dritte Position?

```
put plan[4]
-- [0, 0, 1, 0, 0]
put plan [4][3]
-- 1
```

Mit dem Operator `[]` greifen wir auf Listenpositionen einer linearen Liste zu. Anders als beispielsweise in JavaScript fängt Director dabei mit `1` an zu zählen (die erste Position in der Liste ist der Listenindex 1). `plan[4]` ist also der Inhalt der vierten Position in `plan`, und `plan[4][3]` ist „die dritte Listenposition in derjenigen Liste, die an der vierten Listenposition in `plan` steht".

Eine Liste, wie wir sie beispielsweise in `A` abgelegt haben, ist eine normale (eindimensionale) lineare Liste. In `plan` befindet sich auch eine lineare Liste. Da jedes Element von `plan` aber wieder eine lineare Liste ist, können wir von einer zweidimensionalen Liste sprechen. Wir haben also das (zweidimensionale) Schachbrett in eine „zweidimensionale" Lingostruktur übersetzt. Genau das ist das Ziel dieses Experiments: eine Lingo-Entsprechung für eine Erscheinung der „realen Welt" zu finden.

Versuchsaufbau

Der Beispielfilm *NetzLingo.dir* soll zwei Bereiche haben: eine Eingangssequenz von wenigen Bildern und die Szene „Netz", die unsere Navigationslogik enthalten soll. Das Drehbuch ist entsprechend einfach aufgebaut (vgl. Abbildung 8.28). Am Ende beider Bereiche ist selbstverständlich ein „Stopper-Skript" zu finden.

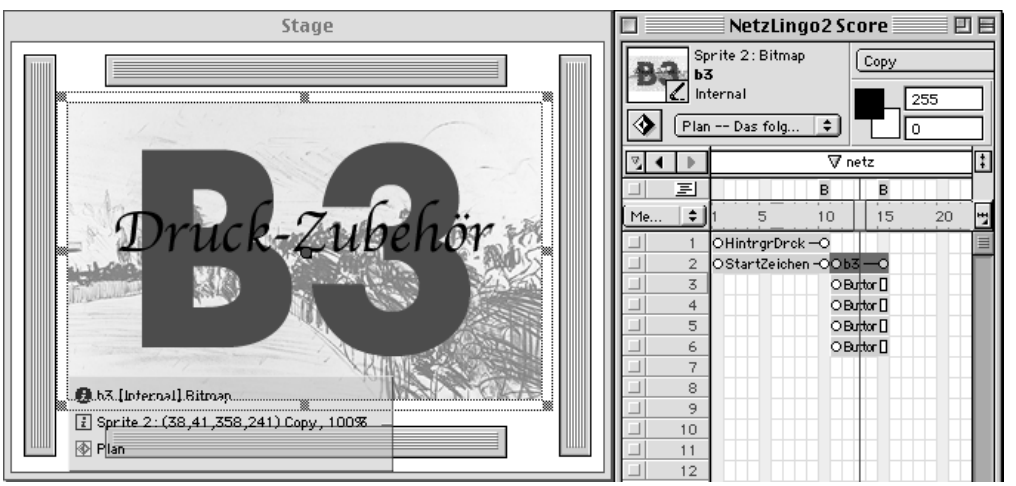


Abbildung 8.28: Bühne und Drehbuch von netzlingo.dir

In der Eingangsszene soll es lediglich ein Hintergrundbild und einen „Start"-Button geben. Nutzen Sie für den Button einen unserer selbst gemachten Behaviors, oder ziehen Sie aus der Bibliothekspalette (Menü Fenster/Bibliothekspalette) das Behavior „Taste" aus der Gruppe „Steuerungen" auf das Sprite. Die Buttonaktion soll lediglich aus einem `go to "Netz"` bestehen.

Im Drehbuchbereich „Netz" legen wir die vier balkenähnlichen Buttons wie in Abbildung 8.28 aus, und in der Mitte das Startbild unserer Netznavigation.

Vier Transitions mit den Namen „up", „down", „left" und „right" werden wie im vorangehenden Experiment als Darsteller angelegt.

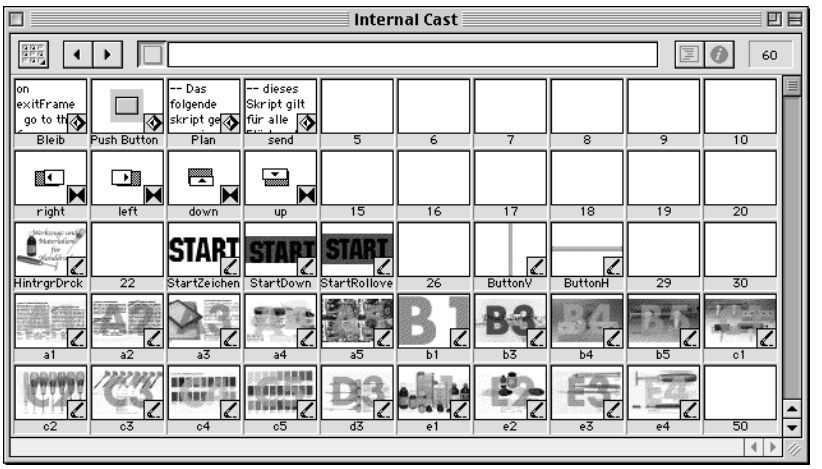


Abbildung 8.29: Besetzung von netzlingo.dir

Durchführung

Zwei Skripte arbeiten an der Funktionalität der Navigation in diesem Film zusammen. In dem Behavior auf den Navigations-Buttons wird die Bewegungsrichtung festgelegt und an das Hauptskript, das auf dem Szenenbild (Sprite 2 in Abbildung 8.28) liegt, gesendet.

Die vier balkenartigen Buttons haben im Grunde die gleiche Funktion: lediglich die gewünschte Navigationsrichtung unterscheidet sich. Wir legen daher ein Behavior „Send“ für alle Buttons an und unterscheiden anhand der Spritenummer, welche Richtungsparameter für das betreffende Sprite gültig sein sollen. Mit `sendSprite()` rufen wir schließlich eine Funktion im Hauptskript auf.

```
on mouseUp me
  case me.spriteNum of
    3:
      mydiff = [0,-1]
      mydir = "left"
```

`mydiff` *beschreibt die gewünschte Bewegungsrichtung in Form einer linearen Liste, deren erster Wert die senkrechte Bewegung (-1: oben, 0: keine Bewegung, 1: unten) und deren zweiter Wert die waagrechte Bewegung beschreibt. In* `mydir` *wird zudem der Name des Transitiondarstellers gespeichert, der beim Bildübergang benutzt werden soll.*

```
    4:
      mydiff = [0,1]
      mydir = "right"
    5:
      mydiff = [-1,0]
```

```
      mydir = "up"
    6:
      mydiff = [1,0]
      mydir = "down"
  end case
  sendsprite(2, #changePosition, mydiff, mydir)
end
```

Mit dem abschließenden `sendSprite()` rufen wir eine Funktion im Behavior „Plan“ auf Sprite 2 auf, das die gesamte Navigationslogik enthält:

```
property plan, rows, cols
property startpos, curpos

on beginSprite me
  rows = 5
  cols = 5
```

Die Properties rows *und* cols *enthalten die Höhe und Breite unseres Schachbrettes.*

```
  A = ["A1", "A2","A3", "A4", "A5"]
  B = ["B1", void,"B3", "B4", "B5"]
  C = ["C1", "C2","C3", "C4", "C5"]
  D = [void, void, "D3", void, void]
  E = ["E1", "E2","E3", "E4", void]
  plan = [A, B, C, D, E ]
```

Wie oben beschrieben, wird hier eine zweidimensionale Liste erzeugt. An den „gefüllten" Listenpositionen stehen die Darstellernamen der Bilder, die angezeigt werden sollen. Nicht begehbare Positionen des Navigationsplanes werden durch VOID *ausgedrückt.* plan *enthält schließlich den gesamten Navigationsplan inklusive der Information, welche Bilder an den einzelnen Positionen angezeigt werden sollen.*

```
  startpos = [2,3]  -- (B3)
  setPosition (me, startpos)
```

Die Startposition wird in Form einer linearen Liste gesetzt und dann an den Handler setPosition *übergeben.* [2, 3] *soll die dritte Position in der zweiten Reihe bedeuten, also „B3".*

```
end
```

Der Handler setPosition *erwartet eine Position im Raster als Übergabewert und holt sich aus* plan *den anzuzeigenden Darsteller. Außerdem kann eine Richtung für die Transition mitübergeben werden.*

```
on setPosition me, val, dir
  zeile = val[1]
  spalte = val[2]
  mymem = plan [zeile][spalte]
```

```
  if voidP(mymem) then
    Falls in plan an dieser Stelle void steht...
    beep
  else
    ...ansonsten zeige die Transition an, setze den Darsteller
    und merke dir die aktuelle Position in curpos.
    if not(voidP(dir)) then puppettransition member(dir)
    sprite(me.spriteNum).member = mymem
    curpos = val
  end if
end

    Der changePosition-Handler wird von den Buttons aufgerufen.
    Zunächst wird die Bewegungsrichtung in val überprüft, indem
    wir sie zur momentanen Position addieren. Liegt eine der
    Listenpositionen außerhalb des gültigen Bereichs, wird die
    Bearbeitung des Skriptes mit return abgebrochen. Ist die
    Zielposition gültig, so wird setPosition mit dem Zielpara-
    meter temp und dem Transition-Parameter dir aufgerufen.
on changePosition me, val, dir
  temp = curpos + val
  if (temp[1] < 1 OR temp[1] > rows) OR (temp[2] < 1 OR \
   temp[2] > cols) then return
  setPosition me, temp, dir
end
```

Diskussion

In diesem Beispiel wird ein Darsteller gewechselt, indem sein Name aus eine Liste gelesen und einem Sprite zugewiesen wird. Für kompliziertere Vorgänge stoßen Sie mit dieser Methode schnell an Grenzen, zum Beispiel, wenn Sie in einer Szene auch Animationen oder Interaktionen zwischen Akteuren auf der Bühne haben wollen. Das Beispiel zeigt aber auch den Vorteil gegenüber Experiment 3: Sie behalten eine gute Übersicht und verzetteln sich nicht in Markierungen, Transitions und unsichtbaren Überleger-Sprites.

Der Szenenwechsel mit einer Liste ist geeignet, wenn Sie eine Diashow-artige Anwendung realisieren möchten, in welcher zwischen den verschiedenen Bildern kreuz und quer gewechselt werden kann. Eine alternative Diashow mit einem linearen Ablauf wird in den Experimenten 5 und 6 vorgestellt.

- **Erweiterung des Behaviors „send“:**

Der nächste Handler stellt fest, ob die aktuelle Position einer Randposition im Planquadrat entspricht, und macht die Tasten, die dann das Planquadrat verlassen würden, unsichtbar. In die Variable `test` wird der Wert der Property-Variablen `curpos` (das heißt die Position innerhalb der Liste `plan`) eingetragen.

```
on prepareFrame me
  test = sprite(2).curpos
  sprite(me.spriteNum).blend = 100
  case me.spriteNum of
    3: if test[2] < 2 then sprite(me.spriteNum).blend = 0
    -- Wenn der zweite Wert in curpos kleiner ist als 2, also die
    -- linke Spalte erreicht ist, dann...
    4: if test[2] > (sprite(2).cols - 1) then \
       sprite(me.spriteNum).blend = 0
    -- Wenn der zweite Wert in curpos größer als 4 ist, also die
    -- unterste Reihe erreicht ist, dann...
    5: if test[1] < 2 then sprite(me.spriteNum).blend = 0
    -- Wenn der erste Wert in curpos kleiner als 2 ist, also die
    -- oberste Reihe erreicht ist, dann...
    6: if test[1] > (sprite(2).rows -1) then \
       sprite(me.spriteNum).blend = 0
    -- Wenn der erste Wert in curpos größer ist als vier, also die
    -- rechte Spalte im Planquadrat erreicht ist, dann...
  end case
end
```

Das Unsichtbarmachen des Sprites erfolgt über die Sprite-Eigenschaft `blend`: Das Behavior stellt zunächst das eigene sprite auf `blend = 100`, um dann abhängig von den einzelnen Bedingungen den Wert auf `0` zu setzen.

- **Rechnen mit Listen:**

Wir haben in diesem Experiment für zahlreiche unterschiedliche Dinge lineare Listen verwendet: als Speicher für den Navigationsplan, für Richtungs- und für Positionsangaben. Geht man hier konsistent vor, so lassen sich bestimmte Dinge sehr effektiv erreichen, wie beispielsweise die Berechnung der neuen Position aus alter Position und Richtung:

```
temp = curpos + val
```

Nehmen wir an, `curpos` ist „B3“, so entspricht das in unserer Konvention `[2, 3]`. Bei einer Bewegung nach unten (`[1, 0]`) ist die Summe:

```
temp = [2, 3] + [1, 0]
temp = [3, 3]
```

Mit Listen lässt sich also auch rechnen, zum Beispiel so wie in den folgenden Beispielen im Nachrichtenfenster:

```
x = [3, 5]

put x * 5
-- [15, 25]

put x * 5.3
-- [15.9000, 26.5000]

put x mod 2
-- [1, 1]

put x * [7, 2]
-- [21, 10]

put x mod [7,2]
-- [3, 1]
```

Alle Grundrechenarten und der Operator `mod` (Modulo) können direkt auf lineare Listen angewandt werden. Ist der zweite Wert dabei eine Zahl (Integer oder Fließkommazahl), so werden alle Elemente der Liste entsprechend verändert (multipliziert etc.). Ist der zweite Operator eine Liste, so wird das erste Element mit dem ersten, das zweite mit dem zweiten multipliziert etc. Die Ergebnisliste hat stets so viele Elemente wie die kürzere der beiden Listen.

Entsprechendes gilt auch für Propertylisten, allerdings sollten Sie vorsichtig sein, wenn Sie für mathematische Operationen Property- und lineare Listen gemischt verwenden.

8.5 Experiment 5: Lineare Diashow

Dieses Experiment ist denkbar einfach: Wir wollen alle Darsteller einer Besetzung nacheinander anzeigen. Der Anwender soll mit Tastaturbefehlen in der Reihe der Bilder weiter- und auch zurückspringen können, und vom letzten Bild soll man wieder zum ersten gelangen.

Welche Optionen haben wir?

- Wir können alle Bilder im Drehbuch auslegen und mit einem einfachen Frameskript die Steuerung übernehmen.
- Wir können aber auch die gesamte Diashow auf einem einzigen Frame realisieren und wie im Experiment 4 eine Darstellerliste benutzen, um die Bilder auszutauschen.

Versuchsaufbau

Legen Sie mit dem Menübefehl Modifizieren/Film/Besetzungen eine neue Besetzung an, die Sie „Media“ benennen (vgl. Abbildung 8.30). Importieren Sie eine Reihe von Bildern in diese Besetzung, indem Sie die Dateien aus ihrem Verzeichnis in die geöffnete Besetzung ziehen, oder mit dem Menübefehl Datei/Importieren... (Strg / ⌘+R).

Abbildung 8.30: Eine neue interne Besetzung legen Sie mit dem Menübefehl Modifizieren/Film/Besetzungen an.

In der Standard-Besetzung „Intern“ legen wir wieder zwei Transition-Darsteller an – und zwar eine für Vorwärts-, eine für Rückwärtsblättern. Wir haben dafür die Übergänge „Nach links aufdecken“ bzw. „Nach rechts aufdecken“ verwendet.

Abbildung 8.31: In der Media-Besetzung sollen alle in der Diashow darzustellenden Bilder liegen.

Nun sollten wir noch dafür sorgen, dass die Bilder auch gut zur Geltung kommen. Wir haben in den Filmeigenschaften die Größe auf 800x600 Pixel eingestellt und die Hintergrundfarbe auf Schwarz.

Für den ersten Teil des Experiments sollen alle Darsteller im Drehbuch liegen. Um nicht alle einzeln platzieren zu müssen, wählen wir alle

Darsteller in der Besetzung „Media“ aus und ziehen sie bei gedrückter Alt/⌥-Taste ins Drehbuch. Damit entsteht *ein* Sprite, das aber alle Darstellerwechsel enthält. Wollen Sie lieber Keyframes bei jedem Darstellerwechsel haben, so lassen Sie beim Ziehen die Taste weg (alle Bilder werden in Spritekanäle untereinander platziert), wählen die entstandenen Sprites im Drehbuch aus und betätigen den Menübefehl Modifizieren/Bild in Kanal...

Durchführung

Ein Frameskript soll die Steuerung der Diashow übernehmen. Was muss es leisten?

- Den Abspielkopf anhalten: Nur bei Tastendruck soll weiter- bzw. zurückgeschaltet werden.
- Die Tastaturabfragen durchführen: Vorwärts- und Rückwärtspfeil sollen die entsprechende Aktion auslösen
- Das Ende und den Anfang der Diashow besonders behandeln: nach dem letzten Bild kommt wieder das erste, vor dem ersten das letzte.
- Transitions als Übergänge von Bild zu Bild auslösen (diese können wir nicht im Drehbuch platzieren, wenn wir für Vor- und Rückwärtsblättern unterschiedliche Transitions sehen wollen).

Das folgende Skript legen Sie im Skriptkanal des Drehbuchs an und ziehen es auf die volle Länge der Diashow. Die ersten beiden Anforderungen sehen Sie in den Event-Handlern für `exitFrame` und `keyUp` erfüllt. Der `keyUp`-Handler ruft die Funktion `blaetter` mit einem Richtungsparameter auf.

```
on exitFrame me
  go the frame
end

on keyUp me
  case charToNum(the key) of
    28: blaetter (me, -1)     -- Linkspfeil
    29, 32: blaetter (me, 1)  -- Rechtspfeil und Leertaste
    otherwise
      beep
  end case
end

on blaetter me, val
  if val > 0 then puppettransition member("next")
  else puppettransition member("prev")
  newframe = (the frame + val)
  if (newframe < 1) then go to the lastframe
  else if (newframe > the lastframe) then go to 1
  else go to newframe
end
```

Im `blaetter`-Handler wird in Abhängigkeit vom Richtungsparameter `val` einer der beiden vorbereiteten Transition-Darsteller aufgerufen. Der Richtungsparameter dient sodann auch dazu, den neuen Frame zu errechnen (`newframe = (the frame + val)`). Um die ungültigen Werte abzufangen, wird `newFrame` sodann überprüft: Hier nutzen wir die Systemeigenschaft `the lastFrame`, da nur unsere Diashow im Drehbuch liegt. Haben Sie dort in anderen Bildbereichen noch weitere Elemente, so müssen Sie `1` und `the lastFrame` durch die tatsächlichen Grenzen Ihrer Diashow ersetzen.

Damit ist eine erste Umsetzung der Diashow abgeschlossen.

Modifikation

Mit dem in der Versuchsvorbereitung erstellten Film wollen wir nun eine Drehbuch-unabhängige Umsetzung versuchen. Als „Platzhalter" für die Bilddarsteller legen wir einen Darsteller an, der nur einen 1 x 1 Pixel großen schwarzen Punkt enthält, und platzieren diesen im Drehbuch. Unser bekanntes Stopperskript (`on exitFrame / go to the frame`) soll den Abspielkopf anhalten, und die Diashow-Funktionalität soll in einem Behavior direkt auf dem „Platzhalter"-Sprite erstellt werden.

Das neue Skript ähnelt weitgehend dem der ersten Umsetzung. Im `beginSprite`-Handler generieren wir allerdings eine `bildliste` aus Darsteller-Referenzen: eine `repeat`-Schleife schreibt alle Bitmap-Darsteller der Besetzung „Media" in die Liste. Dafür nutzen wir die Listenfunktion `add()` sowie die Eigenschaft `type` eines Darstellers. Als Startbild der Präsentation wird das erste Bild der `bildliste` genutzt (`curPos = 1`).

Die Bildänderung schließlich ist nicht mehr ein Sprung im Drehbuch, sondern die Zuweisung eines neuen Darstellers. Im `beginSprite-` und im `blaetter`-Handler wird mit der Zeile

```
sprite(1).member = bildliste[curpos]
```

der jeweils aktuelle Darsteller aus der `bildliste` dem Sprite 1 zugewiesen.

```
property bildliste, curPos, maxbild

on beginSprite me
  maxbild = the number of members of castlib ("media")
  bildliste = []
  repeat with i = 1 to maxbild
    mem = member(i, "media")
    if (mem.type = #bitmap) then bildliste.add(mem)
  end repeat
  maxbild=bildliste.count()
  curPos = 1
  sprite(1).member = bildliste[curpos]
end
```

```
on keyUp me
  case CharToNum(the key) of
    28: blaetter (me, -1)
    29, 32: blaetter (me, 1)
    otherwise
      beep
  end case
end

on exit Frame me
  go to the Frame
end

on blaetter me, val
  if val > 0 then puppettransition "next"
  else puppettransition "prev"
  curpos = (curpos + val)
  if (curpos < 1) then curpos = maxbild
  else if (curpos > maxbild) then curpos = 1
  sprite(1).member = bildliste[curpos]
end
```

Im `blaetter`-Handler musste außerdem die Überprüfung angepasst werden. `curpos` wird auf `maxbild` gesetzt, wenn die Property nach der Berechnung `curpos = (curpos + val)` kleiner als `1` ist (beim Rückwärtsblättern von Position 1), bzw. auf `1`, wenn sie größer als `maxbild` ist.

- **Tastatur-Abfragen:**

Beim Abfragen von Tastatur-Eingaben haben wir drei Möglichkeiten:

- `the key`: Die Systemeigenschaft liefert die zuletzt gedrückte Taste zurück, funktioniert allerdings nicht für Spezialtasten (wie die Pfeiltasten).
- `the keycode`: Liefert einen Zahlenwert zurück, der je nach Tastaturbelegung unterschiedlich ist (z.B. bei deutscher und amerikanischer Tastaturbelegung).
- `charToNum(the key)`: Liefert einen Zahlenwert zurück; bis zum Zahlenwert 127 ist er auf MacOS und Windows identisch, darüber erhalten wir je nach Plattform unterschiedliche Werte (z.B. für die deutschen Umlaute).

Um diese Werte zu testen, erstellen Sie ein Frameskript (in einem ansonsten leeren Film):

```
on exitFrame me
  go to the frame
end
```

```
on keyUp me
  put "key: " & the key
  put "keycode: " & the keycode
  put "charToNum(the key): " & charToNum(the key)
  put "-----------------------"
end
```

Sie finden einen entsprechenden Film als *Keytester.dir* auf der CD-ROM.

Spielen Sie den Film ab, öffnen Sie das Nachrichtenfenster und bringen Sie die Bühne wieder in der Vordergrund (sonst werden Ihre Eingaben ins Nachrichtenfenster geschrieben). Sie erhalten eine Ausgabe ähnlich dieser:

```
-- Welcome to Director --
-- "key: a"
-- "keycode: 0"
-- "charToNum(the key): 97"
-- "-----------------------"
-- "key: b"
-- "keycode: 11"
-- "charToNum(the key): 98"
-- "-----------------------"
-- "key: c"
-- "keycode: 8"
-- "charToNum(the key): 99"
-- "-----------------------"
-- "key: _"
-- "keycode: 123"
-- "charToNum(the key): 28"
-- "-----------------------"
-- "key: _"
-- "keycode: 124"
-- "charToNum(the key): 29"
-- "-----------------------"
-- "key: A"
-- "keycode: 0"
-- "charToNum(the key): 65"
-- "-----------------------"
-- "key:  "
-- "keycode: 49"
-- "charToNum(the key): 32"
-- "-----------------------"
-- "key: ä"
-- "keycode: 39"
-- "charToNum(the key): 138"
-- "-----------------------"
-- "key: ß"
-- "keycode: 27"
-- "charToNum(the key): 167"
-- "-----------------------"
```

Die Ausgaben `charToNum(the key): 28` und `charToNum(the key): 29` sind die Pfeiltasten links und rechts: `the key` alleine ergibt hier kein sinnvolles Ergebnis. `key: A` und `key: a` haben denselben `keyCode`, aber unterschiedliche ASCII-Werte (`charToNum()`). `Keycode: 49` ist die Leertaste, die man mit (`the key = SPACE`) oder `the key = " "` überprüfen kann. Die Werte für `key: ä` und `key: ß` sind am Mac abgenommen und werden unter Windows andere Werte für `charToNum(the key)` ergeben, da ihr ASCII-Wert größer als 127 ist.

Um bestimmte andere Tasten (z.B. ↵ oder alle Tastenkombinationen mit Strg / ⌘) zu überprüfen, müssen Sie Ihren Film als Projektor sichern, da manche Tasten in Director bereits belegt sind und so nicht bis zu Ihrem Skript vordringen. Schreiben Sie das Skriptergebnis statt ins Nachrichtenfenster in einen (nicht editierbaren) Textdarsteller, sichern Sie den Film und erstellen Sie mit dem Menübefehl Datei/Projektor erstellen... einen Projektor.

Um die Ergebnisse in den Felddarsteller „Ausgabe“ zu schreiben, ist das Skript wie folgt anzupassen:

```
on exitFrame me
  go to the frame
end

on keyUp me
  putstring = "key: " & the key & return
  putstring = putstring & "keycode: " & the keycode & return
  putstring = putstring & "charToNum(the key): " &
  charToNum(the key)& return
  member("Ausgabe").text = putstring
end
```

Im Verzeichnis „Keytester“ finden Sie einen entsprechenden Projektor und den offenen (und noch erweiterten) Ausgangsfilm *Keytester2.dir*.

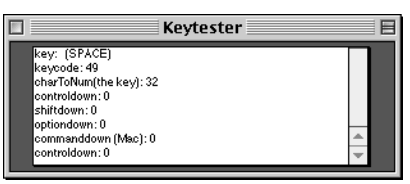


Abbildung 8.32:
Der Projektor Keytester liefert Ihnen erweiterte Angaben zu den gedrückten Tasten.

Uhr und Buch: Animation mit Rotation

Eine Uhr hat nicht viel mit einem Buch gemein, zugegeben. Wollen wir allerdings eine analoge Uhr und das Blättern einer Buchseite in Director umsetzen, so landen wir bei ganz ähnlichen Mitteln der Lingo-Umsetzung.

Rotation und Positionsbestimmungen mit trigonometrischen Funktionen (das sind Sinus, Cosinus und Tangens) hängen eng zusammen und bilden auch für dieses Kapitel die inhaltliche Klammer. Daneben wollen wir einige Animationstechniken kennen lernen: Neben der Sprite-Rotation sind dies die Manipulation der Darsteller-Eigenschaften von Vectorshapes und das Quadding, sprich die freie Verzerrung der Eckpunkte von Bitmap- (und anderen) Sprites.

In diesem Kapitel lernen wir zwei Arten kennen, in Lingo Winkelwerte auszudrücken. Die eine wird beispielsweise bei der Sprite-Eigenschaft `rotation` genutzt: die Gradangabe mit einem Wertebereich von 0 bis 360 Grad. Bei der Sinus-Berechnung allerdings (und bei allen trigonometrischen Funktionen) benötigt Director Winkelangaben in Radians. Die beiden Winkelangaben lassen sich mit einem Dreisatz ineinander umrechnen:

Gradwert / 360 = Radianswert / 2 * pi()

D.h.: Die volle Umdrehung (360 Grad) entspricht dem Umfang eines Kreises mit dem Radius 1 ($2*\pi$). Wollen Sie also einen Gradwert in Radians umrechnen, so lösen Sie die Formel wie folgt auf:

```
on gradToRad gradval
  return gradval * pi() / 180
end
```

und in die entgegengesetzte Richtung geht es mit:

```
on radToGrad radval
  return radval * 180 / pi()
end
```

Packen Sie die beiden Skripte in ein Filmskript und rufen Sie sie testweise im Nachrichtenfenster auf:

```
put gradToRad(180)
-- 3.1416

put radToGrad(2.1)
-- 120.3211
```

9.1 Experiment 1: Stunden-, Minuten- und Sekundenzeiger

In diesem Experiment wollen wir eine analoge Uhr – mit richtigen Zeigern! – auf die Bühne zaubern. Voraussetzung dafür ist, dass wir die Lingo-Funktionen kennen lernen, die uns Zugriff auf die Systemuhr des Anwenders gewähren.

Datum und Uhrzeit in Lingo

Director 7 bietet nur einen sehr einfachen Zugriff auf Datums- und Uhrzeitinformationen. Die Systemeigenschaften `the date` und `the time` liefern uns Strings (Zeichenketten) zurück, wie die folgenden Zeilen im Nachrichtenfenster zeigen.

```
put the date
-- "01.06.2002"
put the time
-- "18:57 Uhr"
```

Für eine Uhr mit Sekundenzeiger ist das noch etwas dünn; aber wir haben noch ein paar weitere Optionen:

```
put the date
-- "30.06.2002"
put the long date
-- "Dienstag, 30. Juni 2002"
put the short date
-- "30.06.2002"
put the abbreviated date
-- "Die, 30. Jun 2002"
put the time
-- "22:59 Uhr"
put the long time
-- "22:59:23 Uhr"
put the short time
-- "22:59 Uhr"
put the abbreviated time
-- "22:59 Uhr"
```

Die Eigenschaft `the long time` enthält alle nötigen Informationen (Stunden, Minuten und Sekunden), die wir für die Umsetzung unserer Uhr benötigen. Andere zeitbezogene Eigenschaften in Director 7 sind `the ticks` und `the milliseconds` (Sechzigstelsekunden bzw. Millisekunden seit Start der Anwendung) sowie `the timer` (Ticks seit Aufruf des Befehls `startTimer`). In Experiment 4 werden wir diese verwenden; für unsere Analoguhr sind sie nicht geeignet.

Die Uhrzeit liegt in `the long time` als String vor; wir müssen also auf Teile des Strings zugreifen, um die Einzelinformationen (Stunde, Minute, Sekunde) zu erhalten. Hier einige Herangehensweisen:

```
lt = the long time
put lt.char[1..2]
-- "22"
```

Mit dieser Syntax greifen wir auf die Buchstaben 1 bis 2 des Strings `lt` zu und erhalten folglich die Stunden. Alle Zeit- und Datumseigenschaften liefern den String allerdings so zurück, wie es der Anwender für sein Betriebssystem konfiguriert hat – eine zweistellige Stundenanzeige ist daher gar nicht garantiert.

Sicherer ist es, auf die „Items“ zwischen den Doppelpunkten zuzugreifen, sich also auf das Trennzeichen zu verlassen. Die Systemeigenschaft `the itemdelimiter` setzt in Director das Trennzeichen, an dem Director Strings in „Items“ zerteilt. Standardmäßig ist das das Komma.

```
put the itemdelimiter
-- ","
the itemdelimiter = ":"
lt = the long time
put lt.item[1]
-- "22"
put lt.item[2]
-- "59"
put lt.item[3]
-- "23 Uhr"
put lt.item[3].word[1]
-- "23"
```

Das `item[3]` enthält alles nach dem letzten Doppelpunkt: also auch Leerzeichen und „Uhr“. Wir machen uns mit dem Aufruf `lt.item[3].word[1]` den Umstand zunutze, dass im String "23 Uhr" zwei „Wörter“ enthalten sind, und greifen auf das erste Wort zu, um dann endlich die Sekunden zu erhalten.

Tabelle 9.1 stellt die Lingo-Eigenschaften, die zum Zugriff auf Teile von Strings dienen, und ihre jeweiligen Trennzeichen gegenüber.

Chunk-Expression	Trennzeichen
line/paragraph	RETURN
word	Leerzeichen, RETURN
item	the itemdelimiter
char	(jeder Buchstabe)

Tabelle 9.1: Zugriff auf Teil-Strings („Chunks“)

Seit Director 8 gibt es ein so genanntes Datumsobjekt, das es ganz ohne String-Manipulationen ermöglicht, Tag, Monat und Jahr zu erhalten. Für unsere Zeitangaben benötigen wir allerdings noch etwas Mathematik. Hier zunächst die Abfragen der Eigenschaft `the systemDate` sowie der Bestandteile des Datumsobjekts:

```
put the systemDate
-- date( 2002, 6, 30 )
t = (the systemDate)
put t.year
-- 2002
put t.month
-- 6
put t.day
-- 30
put t.seconds
-- 7026
```

`the systemDate` ist ein Objekt mit den Eigenschaften `year`, `month`, `day` und `seconds`. `seconds` liefert dabei stets die für den jeweiligen Tag vergangenen Sekunden. Um die Stunden zu erhalten, müssen wir durch 3600 (60 Minuten à 60 Sekunden) teilen:

```
put t.seconds/3600
-- 1
```

Die Minuten erhalten wir, wenn wir den Rest der Teilung durch 3600 mit dem Operator `mod` ermitteln und das Ergebnis durch 60 teilen:

```
put (t.seconds mod 3600) / 60
-- 57
```

Entsprechend sind die Sekunden dann der Rest, der bei Teilung durch 60 übrig bleibt, wiederum mit `mod` ermittelt:

```
put (t.seconds mod 60)
-- 6
```

Die Uhrzeit wurde also um 1:57:06 abgenommen.

Versuchsaufbau

Die erste Uhr, die wir herstellen wollen, ist eine klassische, analoge Uhr mit Zeigern. Einige Vorüberlegungen:

- Zeiger rotieren um einen Mittelpunkt. Wir sollten also versuchen, mit unserer Umsetzung dem zu entsprechen und mit der Rotation von Sprites zu arbeiten.
- Bitmaps zu rotieren ist in Director sehr unbefriedigend, da die Kanten rotierter Bitmap-Sprites „ausfransen“, selbst dann, wenn wir die Darsteller mit Alphakanal importieren.
- Sehr gut geeignet für Rotation sind Vectorshapes und Flash-Darsteller. Wir haben uns hier für Vectorshapes entschieden, aus dem einfachen Grund, dass wir diese direkt in Director erstellen können (und da wir in den folgenden Experimenten auch noch auf das Innenleben dieses Darstellertyps zugreifen wollen).

Abbildung 9.1 zeigt unsere erste Vectorshape-Uhr. Auch Verläufe sind in Vektorshapes einfach einzusetzen, was uns einige interessante visuelle Möglichkeiten an die Hand gibt.

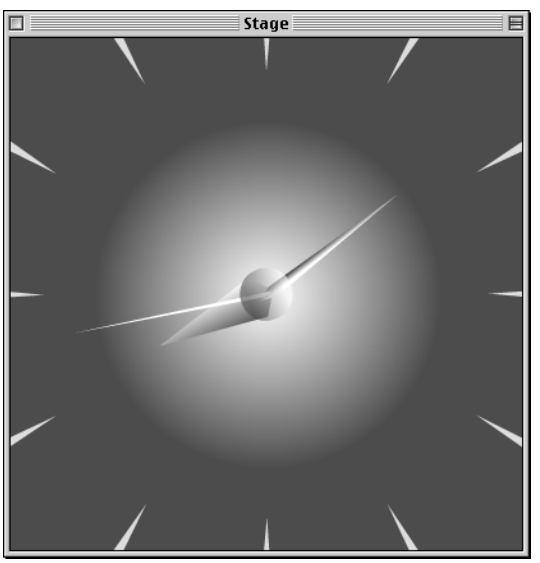


Abbildung 9.1: Analoge Uhr, die erste: Mit Vectorshapes und wenig Lingo können wir dieses Beispiel erstellen.

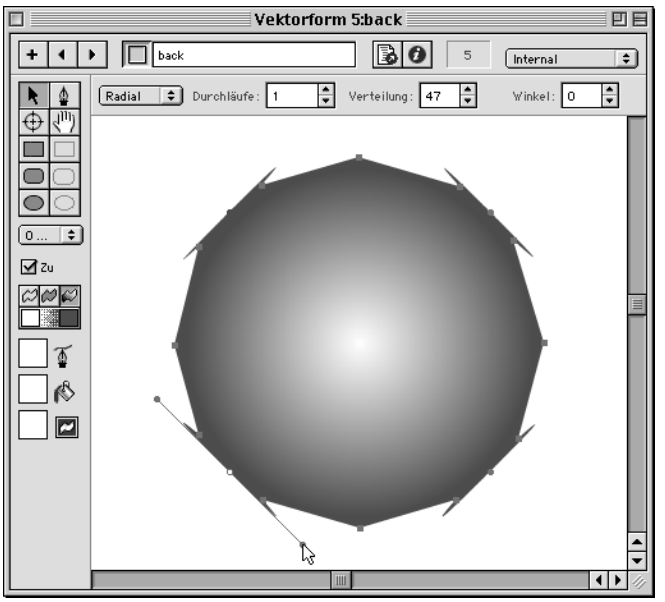


Abbildung 9.2: Vectorshape-Editor: Hier wurde ein Kreis mit einer Verlaufsfüllung angelegt; mit dem „Stift"-Werkzeug lassen sich weitere Punkte auf der Außenlinie platzieren, mit dem Pfeilwerkzeug können Sie Punkte auswählen und mit den Anfassern die Form zusätzlich modifizieren.

In einem quadratischen (zum Beispiel 400 x 400 Pixel großen) Film legen wir zunächst die nötigen Vectorshapes an. Ein runder Darsteller mit einem kreisförmigen, zentrierten Verlauf soll das „Zifferblatt" der Uhr bilden (vgl. Abbildung 9.2). Nutzen Sie das gefüllte Kreis-Werkzeug im Vectorshape-Fenster (Strg / ⌘+⇧+V) und achten Sie auf die folgenden Einstellungen:

- Die Einstellung „Zu" bzw. „Geschl." (Die Außenlinie soll von Director geschlossen werden)
- Die Einstellung „Verlauf" und die Wahl von Verlaufsfarben (zum Beispiel Weiß und Rot)
- Die Verlaufseinstellungen (obere Leiste): „Radial", Durchläufe: 1, Verteilung: zum Beispiel 47, Winkel unerheblich, X- und Y-Offset: jeweils 0 (die letzten Einstellungen erreichen Sie nur, wenn Sie das Vectorshape-Fenster weit aufziehen)
- Die Einstellung der Außenlinie auf 0 Pixel – damit wird sie unsichtbar

Für die drei Zeiger legen wir drei weitere Vectorshapes an; Abbildung 9.3 zeigt den Stundenzeiger mit einem linearen, genau waagrechten Verlauf. (Die Winkeleinstellung erreichen Sie nur, wenn Sie das Vectorshape-Fenster weit aufziehen.)

Die Abbildung zeigt auch die Position des Registrierungspunktes, die sehr wichtig ist für die spätere Rotation aller Zeiger-Sprites: Um den Punkt, den Sie hier festlegen, wird das Sprite gedreht.

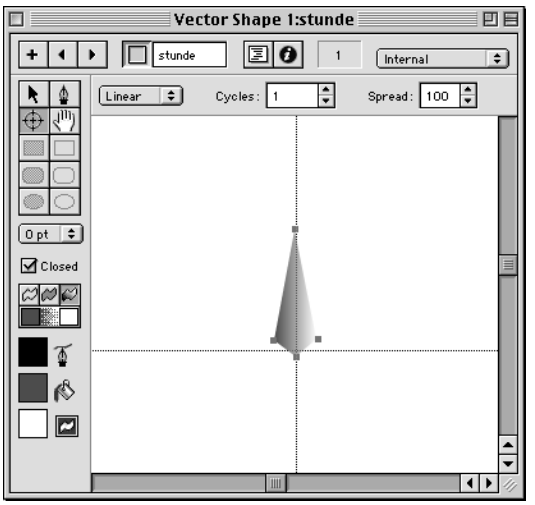


Abbildung 9.3: Der Stundenzeiger wurde direkt mit dem Stiftwerkzeug gezeichnet.

Die weiteren Zeiger legen Sie – mit unterschiedlichen Dicken und Längen – wie den Stundenzeiger an.

Als kleinen „Unruheherd“ wollen wir außerdem in der Mitte der Uhr einen kleinen Kreis rotieren lassen (Abbildung 9.4). Dieses Element hat auch die Funktion, die „Aufhängepunkte“ der Zeiger-Sprites zu verdecken, und wird im höchsten Spritekanal platziert (Sprite 5, vgl. Abbildung 9.5). Die Animation haben wir im Drehbuch mit einer Tweening-Animation eingerichtet; damit diese immer wieder abläuft, ist im Skriptkanal in Frame 20 kein „Stopper-Skript“, sondern das folgende Skript platziert:

```
on exitFrame me
  go to frame 1
end
```

Beim Abspielen des Films wird der Abspielkopf also immer wieder die Strecke zwischen Frame 1 und 20 durchlaufen.

Das „Zifferblatt“ und die drei Zeiger-Sprites werden genau auf die gleiche Position in der Mitte der Bühne (x: 200, y: 200) platziert. Nehmen Sie den Sprite- oder (in Director 8) den Property-Inspektor zu Hilfe, damit Sie die Position numerisch angeben können.

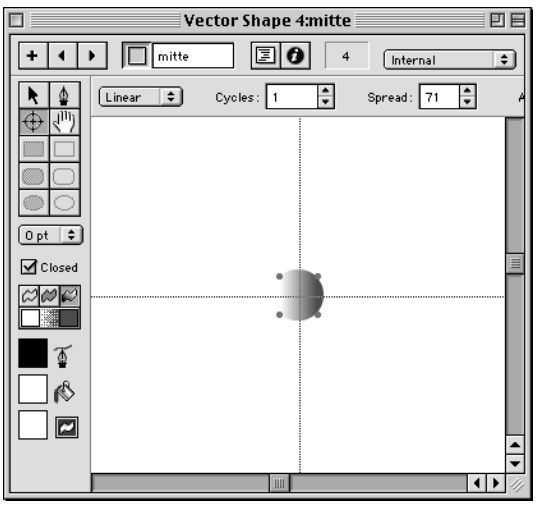


Abbildung 9.4: Die „Unruhe“ unserer Uhr soll im Mittelpunkt rotieren; außerdem verdeckt sie die Rotationspunkte unserer Zeiger.

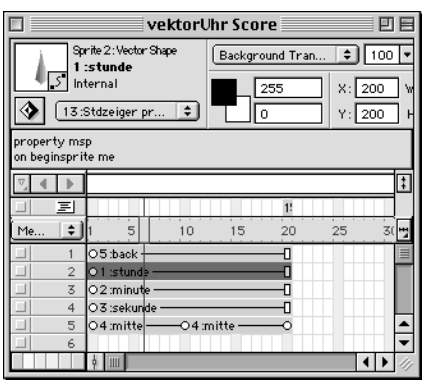


Abbildung 9.5: Das Drehbuch des Beispielfilms

Den Beispielfilm *VektorUhr.dir* finden Sie auf der CD-ROM.

Durchführung

Wir wollen jedem der Zeiger-Sprites ein eigenes Behavior-Skript zuweisen. Der Sekundenzeiger erhält dieses Skript:

```
on exitframe me
  the itemdelimiter = ":"
  lt = the long time
  sek = lt.item[3].word[1]
  rot = integer(sek)*6
  sprite(me.spritenum).rotation = rot
end
```

Wie oben gezeigt, extrahieren wir die Sekundenzahl aus der Systemeigenschaft `the long time`. Dies ist der Director 7-kompatible Weg (der aber

selbstverständlich in höheren Versionen auch funktioniert). Die Berechnung der Rotation folgt der Überlegung, dass eine volle Umdrehung eines Zeigers einer Rotation um 360 Grad entspricht und dass diese Bewegung beim Sekundenzeiger 60 Sekunden dauert. Jede Sekunde entspricht also einer Rotation um 6 Grad. Die Zeile `rot = integer(sek)*6` wandelt den String `sek`, der die Zahl der Sekunden enthält, in ein Integer um und multipliziert es mit 6. Das Ergebnis `rot` dient dazu, die Rotation des betreffenden Sprites einzustellen.

Den Minutenzeiger versuchen wir ebenso umzusetzen (als Behavior auf dem entsprechenden Sprite):

```
on exitframe me
  the itemdelimiter = ":"
  lt = the long time
  min = lt.item[2]
  rot = integer(min)*6
  sprite(me.spritenum).rotation = rot
end
```

Und schließlich die Stunde:

```
on exitframe me
  the itemdelimiter = ":"
  lt = the long time
  std = lt.item[1]
  min = lt.item[2]
  rot = integer(std) *30 + integer(min)/2.0
  sprite(me.spritenum).rotation = rot
end
```

Zwölf Stunden sind ein Umlauf (360 Grad): Jede Stunde entspricht einer Rotation von 30 Grad.

Damit der Stundenzeiger sich nicht nur einmal zur vollen Stunde bewegt, muss außerdem auch die Minutenzahl mit einfließen: In 60 Minuten bewegt sich der Stundenzeiger um 30 Grad, pro Minute also um ein halbes Grad. Das ist dann auch genug der Genauigkeit, das Einrechnen der Sekunden ersparen wir uns.

Allerdings können wir das Skript für den Minutenzeiger auch noch entsprechend modifizieren, denn momentan springt er jeweils zur vollen Minute eine Minute weiter. Eine kontinuierliche Bewegung erreichen Sie, wenn Sie wiederum die Sekunden mit berechnen (pro Sekunde bewegt sich der Minutenzeiger um 1/10 Grad):

```
on exitframe me
  the itemdelimiter = ":"
  lt = the long time
  min = lt.item[2]
  sek = lt.item[3].word[1]
  rot = integer(min)*6 + integer(sek)/10.0
  sprite(me.spritenum).rotation = rot
end
```

Mit dem Date-Objekt (ab Director 8) vereinfachen sich die Skripte, das Prinzip bleibt aber dasselbe:
Sekundenzeiger:

```
on exitFrame me
  sek = (the systemDate).seconds mod 60
  sprite(me.spritenum).rotation = sek * 6
end
```

Minutenzeiger:

```
on exitFrame me
  min = ((the systemDate).seconds mod 3600) / 60.0
  sprite(me.spritenum).rotation = min * 6
end
```

Stundenzeiger:

```
on exitFrame me
  std = (the systemDate).seconds / 3600.0
  sprite(me.spritenum).rotation = std * 30
end
```

Beachten Sie, dass wir bei der Berechnung von `min` und `std` nicht wie oben im Abschnitt „Datum und Uhrzeit in Lingo“ durch Integerzahlen dividieren, sondern durch Fließkommazahlen: Damit ist das Ergebnis auch eine Fließkommazahl, die Nachkommastellen gehen nicht verloren und werden bei der Rotation auch gleich mit dargestellt. Am deutlichsten ist dieser Effekt bei den Stunden:

```
t = 7000      -- Beispiel: 7000 Sekunden
put t/3600       -- bei Teilung durch Integer
-- 1          -- entspricht: 1 Stunde
put t/3600.0     -- bei Teilung durch Fließkommazahl
-- 1.9444     -- entspricht: fast 2 Stunden
```

9.2 Experiment 2: Visuelle Effekte für die Vektor-Uhr

In diesem Experiment wollen wir die Optik unserer Uhr verändern. Als Ausgangsfilm benutzen wir *VektorUhr_Ende.dir*, den letzten Stand des Films aus Experiment 1.

Erweiterung 1: Zeichnen mit trails und rotation

Die Stundeneinteilung, die in Abbildung 9.1 zu sehen ist, ist im Ausgangsfilm noch nicht vorhanden. Welche Möglichkeiten haben wir?

- Wir könnten selbstverständlich in Photoshop oder einem anderen Grafik-Programm ein hübsches Ziffernblatt gestalten und als Hintergrund in den Director-Film einsetzen.
- Wir könnten auch zwölf kleine dreieckige Vectorshapes entsprechend platzieren.
- Wir suchen nach einer Skriptlösung, um die Stunden- und evtl. auch die Minutenstriche zu zeichnen.

Schon in Kapitel 1 haben wir die Sprite-Eigenschaft „Spuren“ (`trails`) benutzt, um mit einem Bitmap-Sprite eine Sinuskurve auf die Bühne zu zeichnen. Dasselbe Prinzip wollen wir hier anwenden: Wir benutzen einen Vektorshape-Darsteller, um dynamisch die Stunden-Einteilung mit Trails auf die Bühne zu malen.

Diese Lösung hat eine große Einschränkung: Die „Spuren“ dürfen nicht in den Bereich hineinragen, der von den Zeiger-Sprites überstrichen wird, da sie dann von den Zeigern gelöscht würden.

Der Registrierungspunkt des Vektordarstellers, den wir zum Zeichnen benutzen wollen, ist so weit aus der Mitte herausverlagert, dass der Sprite auf der Bühne – wie auch alle anderen Elemente – auf den Punkt `point(200,200)` platziert werden kann. Abbildung 9.6 zeigt die Platzierung des Registrierungspunktes. Kontrollieren Sie Ihre Einstellung, indem Sie den Darsteller als Sprite auf der Bühne platzieren.

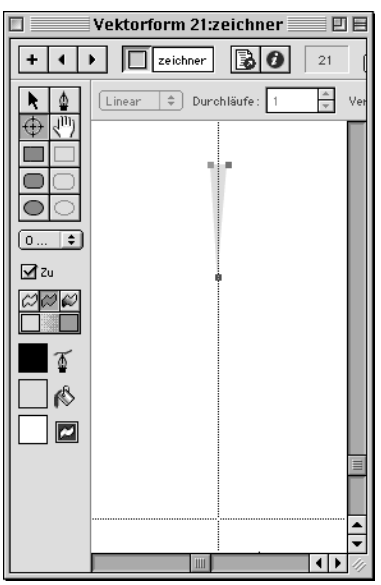


Abbildung 9.6:
Ein dreieckiges Vektorshape dient zum Zeichnen der Stunden- und Minuteneinteilung.

Das folgende Skript soll als Behavior auf dem Dreieck-Sprite angelegt werden. Es zeichnet die Stunden-Einteilung und „versteckt“ das Sprite anschließend:

```
on zeichnen me
  msp = sprite(me.spritenum)
  msp.trails = 1
  msp.loc = point(200,200) -- Mitte der Bühne
  repeat with i = 1 to 12
    msp.rotation = i*30
    updatestage
  end repeat
  msp.loc = point(-1000,-1000)
end

on beginSprite me
  zeichnen me
end
```

In der `repeat`-Schleife wird das Sprite 12-mal, jeweils um 30 Grad weitergedreht, platziert. `updateStage` sorgt dafür, dass das Sprite tatsächlich gezeichnet wird – innerhalb eines `repeat`-Loops geschieht dies ansonsten nicht.

Alternativ können Sie auch eine Stunden- und Minutenskala generieren (vgl. Abbildung 9.7). Wir haben neben dem schon vorhandenen Darsteller „Zeichner“ einen zweiten, leicht versetzten Darsteller angelegt („zeichner2“) und wechseln die beiden aus, je nachdem, ob es sich um einen Minuten- oder einen Stunden-Strich handelt:

```
on zeichnen me
  msp = sprite(me.spritenum)
  msp.trails = 1
  msp.loc = point(200,200) -- Mitte der Bühne
  repeat with i = 1 to 60
    msp.rotation = i*6
    if i mod 5 = 0 then
      msp.member = "zeichner2"
      msp.color = rgb(0,0,255)     -- zusätzliche Farbänderung \
                                   -- für die Stunden
    else
      msp.member = "zeichner"
      msp.color = rgb(255,255,255) -- Farbe zurücksetzen
    end if
    updatestage
  end repeat
  msp.loc = point(-1000,-1000)
end
```

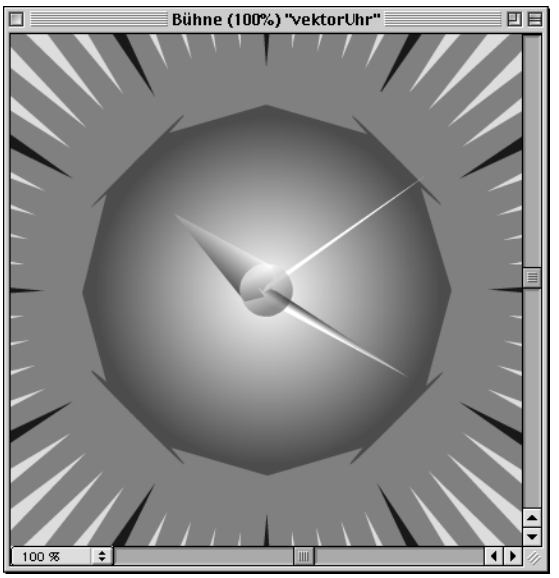


Abbildung 9.7: Die Minuteneinteilung ist ebenfalls per Skript generiert.

Erweiterung 2: Verlaufsfüllungen manipulieren

Vectorshapes sind vollständig programmierbar. Jede Form, jede Linien-, Farb- und Füllungseigenschaft hat ein Lingo-Äquivalent. Führen Sie einmal den Befehl `showProps()` für einen Vectorshape-Darsteller im Nachrichtenfenster aus. Das folgende Listing enthält die Eigenschaften des Darstellers für den Stundenzeiger (vgl. Abbildung 9.1):

```
put member("stunde").showprops()
directToStage:       0
centerRegPoint:      0
regPoint:            point(18, 95)
defaultRect:         rect(0, 0, 37, 102)
imageEnabled:        1
antialias:           1
scale:               100.0000
originMode:          #center
originPoint:         point(0, 0)
originH:             0.0000
originV:             0.0000
viewScale:           100.0000
viewPoint:           point(0, 0)
viewH:               0.0000
viewV:               0.0000
broadcastProps:      1
scaleMode:           #autoSize
```

```
static:              1
defaultRectMode:     #flash
closed:              1
strokeWidth:         0.0000
fillMode:            #gradient
gradientType:        #linear
fillScale:           100.0000
fillDirection:       0.0000
fillOffset:          point(0, 0)
fillCycles:          1
strokeColor:         rgb( 0, 0, 0 )
fillColor:           rgb( 255, 0, 0 )
backgroundColor:     rgb( 255, 255, 255 )
endColor:            rgb( 255, 255, 255 )
vertexList:          [[#vertex: point(1, 49)],
                      [#vertex: point(-17, 36)],
                      [#vertex: point(0, -51)],
                      [#vertex: point(18, 35)],
                      [#vertex: point(1, 49)]]
flashRect:           rect(0, 0, 37, 102)
```

Sie können jede dieser Eigenschaften im Nachrichtenfenster ausprobieren und ändern damit direkt den Darsteller.

Die Zeilen

```
member("back").fillmode = #solid
member("back").fillcolor = rgb(255,0,0)
```

setzen den Darsteller „back“ (den Uhrenhintergrund) auf eine einfarbige Füllung in der Farbe Rot; die folgenden Befehle füllen das Shape mit einem radialen (kreisförmigen) Verlauf (`fillMode`, `gradientType`) von Weiß (`fillColor`) bis Rot (`endColor`). Der Verlauf soll fünf Durchläufe (`fillCycles`) haben und auf 25% skaliert sein (`fillScale`). Außerdem soll er nicht im Darsteller zentriert, sondern auf einen Punkt verschoben sein (`fillOffset`). Die Umgrenzungslinie des Shapes soll 0.5 Pixel breit (`strokeWidth`) und weiß (`strokeColor`) sein. Das Ergebnis sehen Sie in Abbildung 9.8.

```
member("back").fillMode = #gradient
member("back").gradientType = #radial
member("back").fillColor = rgb(255,255,255)
member("back").endColor = rgb(255,0,0)
member("back").fillCycles = 5
member("back").fillScale = 25
member("back").fillOffset = point(-107, 48)
member("back").strokeWidth = 0.5
member("back").strokeColor = rgb(255,255,255)
```

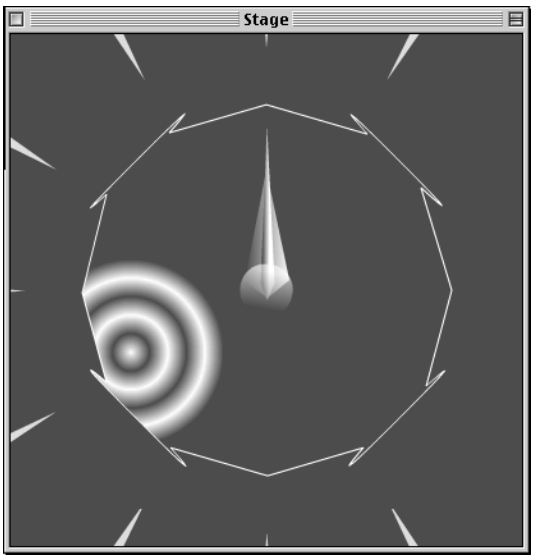


Abbildung 9.8: Manipulation des Vectorshapes „back“

Nach dem obigen Test liegt es nahe, statt eines Sekundenzeigers doch einmal einen Punkt im Verlauf des Vectorshapes als Sekundenanzeige zu verwenden: die Eigenschaft `fillOffset` erlaubt es ja offenbar, das Verlaufszentrum eines radialen Verlaufs irgendwo im Darsteller zu platzieren.

Die nötige Positionsangabe ist in unserem Fall vom Mittelpunkt des Darstellers aus gerechnet; wir benötigen einen Wert aus x- und y-Koordinaten.

Die mathematische Aufgabe lautet demnach, die Sekunden in die Positionsangabe umzurechnen. Der erste Schritt ist wie oben, die Sekunden in eine Winkelangabe umzurechnen, die wir für die weiteren Berechnungen aber in Radians, nicht in Grad benötigen:

```
rot = integer(sek)*6
alpha = rot * pi()/180 -- Winkel in Radians
```

Aus Abbildung 9.9 würde folgen:

```
x = sin(alpha) * radius
y = cos(alpha) * radius
```

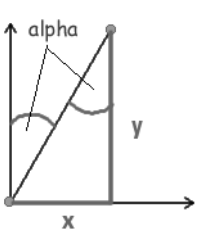


Abbildung 9.9:
Punkte auf einer Kreisbahn berechnen

Da aber die y-Achse in Director (im Vergleich zum Koordinatensystem der Mathematik) genau gespiegelt ist, müssen wir den y-Wert noch mit -1 multiplizieren und können dann den `fillOffset` des Vectorshapes wie folgt zuweisen:

```
x = sin(alpha) * radius
y = -1 * cos(alpha) * radius
member("back").fillOffset = point(x,y)
```

Das folgende Behavior, das Sie am besten direkt auf das Sprite 1 platzieren, setzt die gesamte Funktionalität um:

```
property radius

on beginSprite me
  member("back").fillMode = #gradient
  member("back").gradientType = #radial
  member("back").fillColor = rgb(255,255,255)
  member("back").endColor = rgb(255,0,0)
  member("back").fillCycles = 3
  member("back").fillScale = 3.5
  member("back").strokeWidth = 0.5
  member("back").strokeColor = rgb(255,255,255)
  --
  radius = 125
end

on exitframe me
  the itemdelimiter = ":"
  lt = the long time
  sek = lt.item[3].word[1]
  rot = integer(sek)*6
  alpha = rot * pi()/180
  x = sin(alpha) * radius
  y = -1 * cos(alpha) * radius
  member("back").fillOffset = point(x,y)
end
```

Die Angaben im beginSprite-Handler, die den Darsteller verändern, sind überflüssig, wenn Sie den Darsteller im Vectorshape-Editor bereits entsprechend eingerichtet haben. Die Eigenschaft `radius` legt fest, wie weit das Zentrum des Verlaufs vom Mittelpunkt des Darstellers entfernt sein soll.

Formel für kreisförmige Lingo-Animation

Sicher haben Sie es bemerkt, dass wir gerade eine Formel entwickelt haben, die keineswegs auf Vectorshapes begrenzt ist. Vielmehr können wir auch Sprites um beliebige Punkte auf der Bühne kreisen lassen:

```
property deg, mycenter, radius, step

on beginSprite me
  sprite(me.spriteNum).loc = point(-1000,-1000)
  radius = 100
  mycenter = point(150,200)
  step = 1
end

on exitFrame me
  deg = (deg + step) mod 360
  alpha = deg * pi() / 180
  x = sin(alpha) * radius
  y = -1 * cos(alpha) * radius
  sprite(me.spriteNum).loc = mycenter + point(x,y)
end
```

Sie können im `beginSprite`-Handler einen Radius für die Kreisbewegung, einen Mittelpunkt und eine Schrittweite (die angibt, wie viele Gradschritte pro `exitFrame` weitergezählt werden soll) vorgeben, die der `exitframe`-Handler in bekannter Manier benutzt, um die Position des Sprites zu verändern. Probieren Sie unterschiedliche Bildraten für die Geschwindigkeit der Animation aus!

Auch dieses Skript ist ein schönes Experimentierfeld, mit dem Sie allerhand andere Formen als „nur" einen schnöden Kreis erzeugen können. Einfach sind z.B. Ellipsen (z.B. `x = 2 * sin(alpha) * radius`) und Achten (z.B. `x = cos(alpha) * sin(alpha) * radius`) zu erreichen. Probieren Sie es aus!

9.3 Experiment 3: Timer für alle Zwecke

Während eine Uhr quasi unendlich läuft, gibt es durchaus auch Aktionen, die einen Startpunkt oder einen Endpunkt haben, auf die wir uns bei Zeitangaben beziehen können. Die Sekunden zu zählen seit Zeitpunkt x oder die verbleibende Zeit bis zum Spielende anzuzeigen, ist mit Lingo-Mitteln (zumindest seit Director 8) einfach.

Das Datums-Objekt, das wir oben vorgestellt haben, hilft uns dabei. Auch in Director 7 lassen sich – mit den oben gezeigten Mitteln, auf die String-Rückgabewerte von `the time` und `the date` zuzugreifen – Zeitdifferenzen berechnen. Sobald allerdings größere Zeiträume betroffen sind, sind schon einige Zusatzbedingungen (Schaltjahre!) mitzubedenken.

Leichter hat man es dann mit dem Datumsobjekt: Damit ist die Bestimmung der Differenz von zwei Datumsangaben ein Kinderspiel. Die Differenz (in Tagen) zwischen dem 17.3.1973 und dem 24. 7. 2002 errechnen Sie wie folgt:

```
d1 = date (1973, 3, 17)
d2 = date (2002, 7, 24)
put abs(d1 - d2)
-- 10721
```

Die folgende Funktion nimmt ein Datumsobjekt entgegen und gibt die Differenz zum heutigen Datum zurück:

```
on getDiff einDatum
  return abs(the systemdate - einDatum)
end
```

Das folgende Behavior startet im `beginSprite`-Handler einen Timer und zeigt in einem Textdarsteller an, wie viele Sekunden noch bis zur einstellbaren `maxTime` übrig sind (zum Beispiel 10 Minuten, die man zur Lösung einer Aufgabe zur Verfügung hat):

```
property maxTime

on beginSprite me
  maxTime = 10*60*60 -- 10 Minuten in Ticks
  startTimer
end

on prepareFrame me
  diff = (maxTime - the timer) / 60
  if diff <> lastdiff then
    sprite (me.spriteNum).member.text = diff
  end if
end
```

9.4 Experiment 4: Seitenrotation, oder: Ein Buch zum Blättern

Was hat das Blättern eines Buches mit Rotation zu tun? Nun, wenn Sie den Weg der Seiten-Ecken beim Blättern beobachten, dann beschreiben Sie tatsächlich einen Halbkreis. Diese Bewegung wollen wir mit Lingo nachbilden.

Versuchsaufbau

Unser Bilderbuch besteht aus 10 Seiten: Dafür haben wir im Beispielfilm eine zweite Besetzung „Media“ angelegt und zehn beliebige Abbildungen in ebenfalls beliebigen Größen in die ersten zehn Positionen gelegt.

In der Hauptbesetzung liegen die weiteren Bestandteile des Experiments: ein Buchhintergrund und ein in Fireworks produziertes Bild mit einem schmalen Verlauf von Schwarz zu Transparent, der die Buchmitte etwas räumlich wirken lassen soll. Dieses Sprite sollte im höchsten Drehbuchkanal zu liegen kommen (vgl. Abbildung 9.10), der Buchhintergrund liegt in Sprite 1.

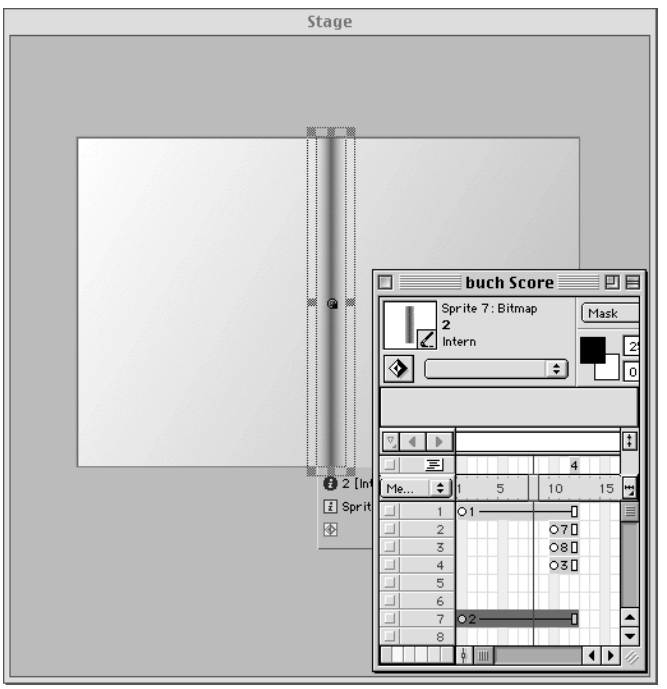


Abbildung 9.10: Drehbuch und Bühne unseres Experiments

Außerdem platzieren wir drei „Seiten“ auf der Bühne: Wir ziehen drei der Darsteller aus der Besetzung „Media“ als Sprites in die Kanäle 2, 3 und 4 und skalieren sie auf die einheitliche Größe von 194x253 Pixel (vgl. Abbildung 9.11). Die beiden „Seiten“ in Sprite 2 und 3 sollen folgende kleinen Behaviors erhalten:
Sprite 2 (links):

```
on mouseup me
  sendsprite(4,#animate, -1)
end
```

Sprite 3 (rechts):

```
on mouseup me
  sendsprite(4,#animate, 1)
end
```

Die beiden Sprites senden also an Sprite 4 lediglich die Aufforderung zur Animation sowie einen Richtungsparameter: 1 für Vorwärts-, -1 für Rückwärtsblättern.

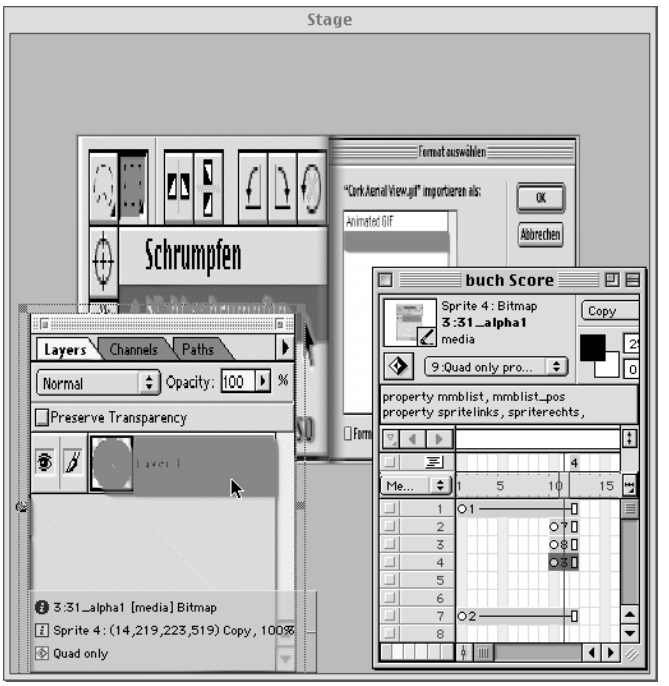


Abbildung 9.11: Die drei Seiten-Sprites werden auf eine einheitliche Größe skaliert.

Auf der CD finden Sie den vorbereiteten Ausgangsfilm *Buch_Start.dir*.

Durchführung

Zwei neue Ansätze wollen wir in diesem Experiment ausprobieren: zeitbasierte Animation und die Verzerrung von Bitmap-Sprites mit Quads.

- **Zeitbasierte Animation:**

Animationen werden meist an die Director-eigenen Frame-Events gekoppelt. Ein Zähler inkrementiert bei jedem `prepareFrame` oder `exitFrame`, und die Animation nutzt die Zählerwerte zur Veränderung von Sprite-Positionen, Farbwerten o.Ä. Nun wollen wir einer Animation eine feste Dauer geben – unabhängig davon, welche Bildrate eingestellt ist. Dazu berechnen wir – wiederum in einem `prepareFrame`- oder `exitFrame`-Handler – den Anteil der Gesamtdauer, der bereits verstrichen ist, nach folgender Formel:

Prozentsatz = ZeitDifferenz_seit_Start / Gesamtdauer

Mit diesem Prozentsatz berechnen wir unsere Animation. Im Falle der Blätter-Animation würde ein vollständiges Umblättern 100% der Animation entsprechen.

Anders als bei der framebasierten Animation mit einem Counter, wo eine höhere Bildrate einer schnelleren Animation entspricht, erreichen wir hier mit einer höheren Bildrate mehr Zwischenbilder, also eine „feinere Auflösung" der Animation. Die Dauer dagegen wird von uns als feste Zeitangabe festgelegt.

- **Quads:**

Die Eigenschaft `rect` eines Sprites beschreibt – anhand von zwei Punktangaben oder 4 Integerwerten – Größe und Position des Sprites:

```
put sprite(1).rect
-- rect(23, 18, 239, 211)
```

Für ein unverzerrtes Sprite enthält die Eigenschaft `quad` dieselben Informationen, allerdings etwas anders formuliert:

```
put sprite(1).quad
-- [point(23.0000, 18.0000), point(239.0000, 18.0000),
point(239.0000, 211.0000), point(23.0000, 211.0000)]
```

Es ist eine lineare Liste aus vier Punktangaben. Mit dem Verzerren-Werkzeug in Directors Werkzeugpalette können Sie das Sprite direkt auf der Bühne rotieren und neigen: Bewegen Sie die Maus bei ausgewähltem Verzerren-Werkzeug über das Sprite, dann zeigt Ihnen Director die unterschiedlichen Funktionen durch einen Cursorwechsel an.

Wenn Sie ein Sprite verzerren, beschreibt `rect` weiterhin das umschließende Rechteck um das verzerrte Sprite; `quad` hingegen die tatsächlichen Positionen der Eckpunkte:

```
put sprite(2).rect
-- rect(256, 6, 499, 235)

put sprite(2).quad
-- [point(333.6971, 6.8542), point(498.4035, 103.5642),
point(421.4406, 234.6394), point(256.7342, 137.9295)]
```

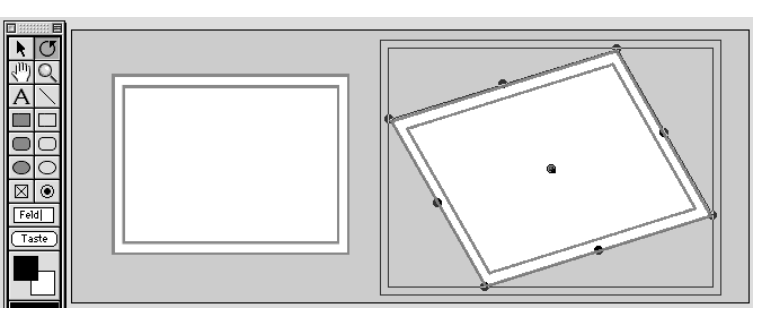


***Abbildung 9.12:** Das Verzerren-Werkzeug erlaubt es, Sprites auf der Bühne zu rotieren und zu neigen – und damit die Eigenschaft `quad` des Sprites zu manipulieren.*

Mit Lingo haben Sie noch weiter gehende Möglichkeiten als nur Rotation (`rotation`) und Neigung (`skew`): Sie können per Skript jede Punktangabe des Quads manipulieren. Im Nachrichtenfenster haben wir 2 Punkte des Quads von Sprite 2 zusätzlich verzerrt. Das Ergebnis des folgenden Skripts sehen Sie in Abbildung 9.13.

```
sprite(2).quad = [point(281.0254, 5.0), \
 point(482.6218, 16.1850), point(567.9300, 163.2310), \
 point(200.0, 225.5198)]
updatestage
```

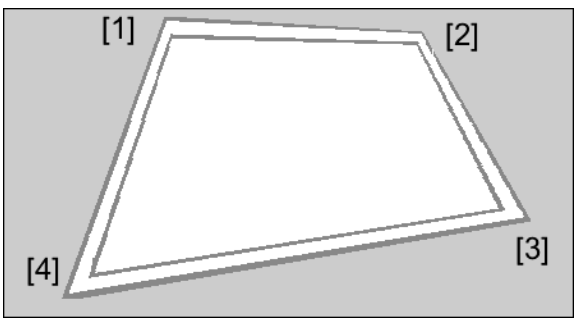


Abbildung 9.13: Ein mit Lingo verzerrtes Sprite. Die Zahlen bezeichnen die Listenindizes in der Eigenschaft *quad*.

Wenn Sie das Quad eines Sprites verändern, müssen Sie stets eine vollständige Liste aus vier Positionen (als `point()`) zuweisen.

▶ Zeitbasierte Blätteranimation mit Quads:

Wenn wir eine Seite unseres Buches blättern, so können wir diese Bewegung als eine Sinuskurve der beiden Buchecken auffassen, die einen horizontalen und einen vertikalen Weg beschreibt (vgl. Abbildung 9.14). `horizWeg` ist zweimal die Seitenbreite, `vertWeg` ist ein Wert, der festlegt, wie stark die perspektivische Verzerrung der geblätterten Seite sein soll. Sie sollten einige unterschiedliche Werte ausprobieren.

Wir nutzen die Sinuswerte für Winkel zwischen 0 und pi() für die Animation der Buchecke. Das Statement `sinus(pi()*proz)` nutzt unseren Prozentwert für die vergangene Zeit, um den Winkel für die Sinusberechnung entsprechend zu „dynamisieren“. Das Ergebnis der Sinusfunktion liegt stets zwischen 0 und 1 und muss dann mit einem Skalierungsfaktor (`vertWeg` im Skript) multipliziert werden.

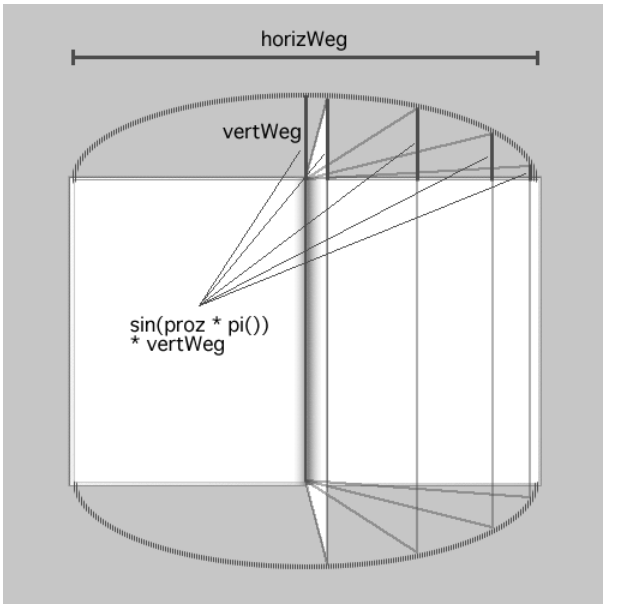


Abbildung 9.14: Blättern: Zwei Ecken der zu blätternden Seite bleiben fix, zwei werden auf Sinuskurven von rechts nach links oder entgegengesetzt bewegt.

Ein erster Skriptansatz implementiert die nötigen Properties und ihre Initialisierung im `beginSprite`-Handler, den Handler `animate`, der die Blätteranimation auslösen soll, sowie die eigentliche Animation im `prepareFrame`-Handler:

```
property spritelinks, spriterechts, spritemitte
property orig_q, dur, startticks
property myproz, vertWeg, horizWeg
property mydir

on beginsprite me
```

Initialisierung der Properties: Spritereferenzen, Dauer der Animation, Ausgangsquad für die zu blätternde Seite, horizontaler und vertikaler Weg für die Animation. `startticks` *wird bei Animationsstart auf* `the ticks` *gesetzt. Solange die Eigenschaft den Wert 0 hat, wird nicht animiert.*

```
  spritelinks = sprite(2)
  spriterechts = sprite(3)
  spritemitte = sprite(me.spriteNum)
  dur = 40
  orig_q = duplicate(spriterechts.quad)
  horizWeg = 2 * spriterechts.width
  -- Weg des Eckpunkts horizontal
  vertWeg = 50 -- "Blätterverzerrung" vertikal
  startticks = 0
end
```

```
on animate me, direction
```

Handler, der die Animation startet. Er wird von Sprite 2 und 3 aus mit sendSprite() *aufgerufen. Falls bereits eine Animation läuft (*startticks *also größer 0 ist), soll er nicht ausgeführt werden.*

```
  if startticks > 0 then return
  -- Abbruch: Animation läuft schon
  mydir = direction
  startticks = the ticks
end

on prepareframe me
  if startticks > 0 then
```

Berechnung des Prozentwertes der Animation aus verstrichener Zeit und Gesamtdauer. In Abhängigkeit von der Blätterrichtung mydir *wird der Prozentwert direkt verwendet oder umgekehrt (von 100% abgezogen).*

```
    mytime = the ticks - startticks
    if mydir = 1 then myproz = mytime / float(dur)
    else myproz = 1 - (mytime / float(dur))

    --
    if myproz <= 1 AND myproz >= 0 then
```

Berechnung des x- und y-Abstandes für die beiden Seitenecken. In der Funktion do_quad*, die wir weiter unten im Skript definieren, wird dann das tatsächliche Quad der Blätterseite berechnet und zugewiesen.*

```
      myYdelta = vertWeg * sin(pi()*myproz)
      myXdelta = horizWeg * myproz
      do_quad(me, orig_q, myXdelta, myYdelta)
    else
```

Am Ende der Animation wird die Funktion mit do_quad *mit den Zielwerten aufgerufen und wir setzen die Eigenschaft* startticks *auf 0, um die Animation zu stoppen.*

```
      if mydir = 1 then
        do_quad(me, orig_q, horizWeg,0)
      else
        do_quad(me, orig_q, 0,0)
      end if
      startticks = 0
    end if
  end if
end
```

```
on do_quad me, aquad, ax, ay
```

*Die Funktion nimmt ein Ausgangsquad (*aquad*) und zwei Differenzwerte (*ax *und* ay*) entgegen, modifiziert das Ausgangsquad entsprechend und weist dem mittleren Sprite das neue Quad zu.*

```
  ziel_q = aquad - [point(0,0),point(ax,ay), \
   point(ax, -1*ay), point(0,0)]
  spritemitte.quad = ziel_q
end
```

Bitte probieren Sie diesen Zwischenstand aus; es ist wichtig zu sehen, welche Teilfunktionalität wir damit erreicht haben. Dieses Skript finden Sie auf der CD-ROM im Film *Buch_quadonly.dir*.

Tatsächlich animiert nämlich unser Seitenquad schon sehr schön, aber wir haben auch noch einiges zu tun:

- Im linken Bildbereich wird unsere Blätterseite noch gespiegelt: Hier müssen wir unsere Quad-Berechnung modifizieren.
- Es finden noch keine Darstellerwechsel statt. Wir müssen beim Blättern sowohl dem Blättersprite als auch den beiden Sprites rechts und links die richtigen Darsteller zuordnen.

Die nötigen Änderungen sind im folgenden – vollständigen – Skript kommentiert:

```
property mmblist, mmblist_pos
property spritelinks, spriterechts, spritemitte
property orig_q, dur, startticks
property myproz, vertWeg, horizWeg
property mmb_r0, mmb_l0, mmb_r1, mmb_l1, mydir
property spiegeln

on beginsprite me
```

Wir generieren zunächst aus allen Bitmap-Darstellern der Besetzung „Media" eine Darstellerliste mmblist*. Falls dies keine gerade Anzahl von Darstellern ergibt, halten wir den Film an und warnen mit einem Alert.*

```
  mmblist = []
  repeat with i = 1 to the number of members of castLib "media"
    if member(i, "media").type = #bitmap then \
     mmblist.add(member(i, "media"))
  end repeat
  if (mmblist.count() mod 2 > 0) then
    alert "Bitte eine gerade Anzahl von Seiten in der \
     Besetzung MEDIA anlegen!"
    halt
  end if
```

Die aktuelle Listenposition, die stets den Darsteller für das linke Sprite angibt, wird auf 1 gesetzt. Den Sprites werden die entsprechenden Darsteller zugewiesen und die Quads werden initialisiert.

```
  mmblist_pos = 1
  spritelinks = sprite(2)
  spriterechts = sprite(3)
  spritemitte = sprite(me.spriteNum)
  spritelinks.member = mmblist[mmblist_pos]
  spritelinks.quad = spritelinks.quad
  spriterechts.member = mmblist[mmblist_pos + 1]
  spriterechts.quad = spriterechts.quad
  spritemitte.loc=point(-1000, -1000)
  --
  dur = 40
  startticks = 0
  orig_q = duplicate(spriterechts.quad)
  horizWeg = 2 * spriterechts.width
  -- Weg des Eckpunkts horizontal
  vertWeg = 50 -- "Blätterverzerrung" vertikal
end

on prepareframe me
  if startticks > 0 then
    mytime = the ticks - startticks
    if mydir = 1 then myproz = mytime / float(dur)
    else myproz = 1 - (mytime / float(dur))

    --
    if myproz <= 1 AND myproz >= 0 then
```

*Hier setzen wir nun - abhängig vom erreichten Zustand der Animation und der Animationsrichtung - die Darsteller für das Blättersprite (*mycurrmember*) und für das rechte bzw. linke Sprite (*rechtsmember *bzw.* linksmember*). Außerdem wird die Eigenschaft* spiegeln *eingeführt, um im* do_quad*-Handler die Quadberechnung modifizieren zu können.*

```
      if mydir = 1 AND myproz < 0.5 then
        mycurrmember = mmblist[mmblist_pos+1]
        rechtsmember = mmblist[mmblist_pos+3]
        spiegeln = 0
      else if mydir = 1 AND myproz > 0.5 then
        mycurrmember = mmblist[mmblist_pos+2]
        spiegeln = 1
      else if mydir = -1 AND myproz > 0.5 then
        mycurrmember = mmblist[mmblist_pos]
        linksmember = mmblist[mmblist_pos-2]
        spiegeln = 1
      else if mydir = -1 AND myproz < 0.5 then
        mycurrmember = mmblist[mmblist_pos-1]
        spiegeln = 0
      end if
```

Die Darsteller für das rechte und linke Sprite werden den Sprites zugewiesen.

```
      if not(voidp(rechtsmember)) then
        spriterechts.quad = spriterechts.quad
        spriterechts.member = rechtsmember
      end if
      if not(voidp(linksmember)) then
        spritelinks.quad = spritelinks.quad
        spritelinks.member = linksmember
      end if
      myYdelta = vertWeg * sin(pi()*myproz)
      myXdelta = horizWeg * myproz
      spritemitte.member = mycurrmember
      do_quad(me, orig_q, myXdelta, myYdelta)
    else
      -- Animation Ende
      mmblist_pos = mmblist_pos + 2*mydir
      if mydir = 1 then
        spiegeln = 1
        spritelinks. member = mmblist[mmblist_pos]
        do_quad(me, orig_q, horizWeg,0)
      else
        spriterechts. member = mmblist[mmblist_pos + 1]
        do_quad(me, orig_q, 0,0)
      end if
      startticks = 0
    end if
  end if
end

on do_quad me, aquad, ax, ay
  ziel_q = aquad - [point(0,0),point(ax,ay), \
   point(ax, -1*ay),point(0,0)]
```

Wenn es nötig ist, das Quad zu spiegeln, werden die einzelnen Quadpunkte in ziel_q *umsortiert.*

```
  if spiegeln = 1 then ziel_q = [ziel_q[2], ziel_q[1], \
   ziel_q[4], ziel_q[3]]
  spritemitte.quad = ziel_q
end

on animate me, direction
  if startticks > 0 then return
  -- Abbruch: Animation läuft schon
  mydir = direction
```

Hier überprüfen wir, ob wir in die angegebene Richtung mydir *überhaupt noch blättern dürfen. Falls nicht, wird die Skriptbearbeitung mit* return *abgebrochen.*

```
  temp = mmblist_pos + 2 * mydir
  if temp < 0 OR temp > count(mmblist) then return
  startticks = the ticks
end
```

Virtuelle Realität

Auch wenn alles, was wir am Rechner produzieren, mehr oder weniger eine „virtuelle Realität“ darstellt, so steht der Begriff zumeist doch für besonders „immersive“ Verfahren, in denen der Anwender das Gefühl hat, sich inmitten einer virtuellen Welt zu befinden. Ein beliebtes Verfahren, ein solches „Eintauchen“ zu simulieren, ist der Rundumblick: Der Anwender, das Auge, die Kamera steht im Mittelpunkt einer Szene, und durch Drehung (bzw. Ziehen mit der Maus) wird der Blickwinkel verändert.

Mit QuicktimeVR gibt es eine Technologie, die sich diesem Prinzip verschrieben hat. In Director können wir QuicktimeVR-Filme sehr einfach einbinden und interaktiv nutzen, was wir im zweiten Experiment auch machen werden. Allerdings bietet das Programm Alternativen: Wir können den „Panorama-Effekt “ auch mit Sprites erreichen (und haben den Vorteil, nicht auf Quicktime angewiesen zu sein) – und wir können seit Director 8.5 3D-Lingo nutzen, um unsere eigenen virtuellen Räume zu erstellen. Beide Ansätze wollen wir in diesem Kapitel ausprobieren.

Für die ersten beiden Experimente finden Sie vorbereitete Panoramen in unterschiedlichen Auflösungen und Formaten im Verzeichnis *Panos* auf der CD-ROM. Das Panoramafoto „Port of San Francisco“ ist ein Beispiel-Projekt des Programmes The VR Worx; es wurde aus 16 Einzelbildern zusammengefügt und als JPEG-Bild sowie als QuicktimeVR-Panorama exportiert (vgl. Abbildung 10.1).

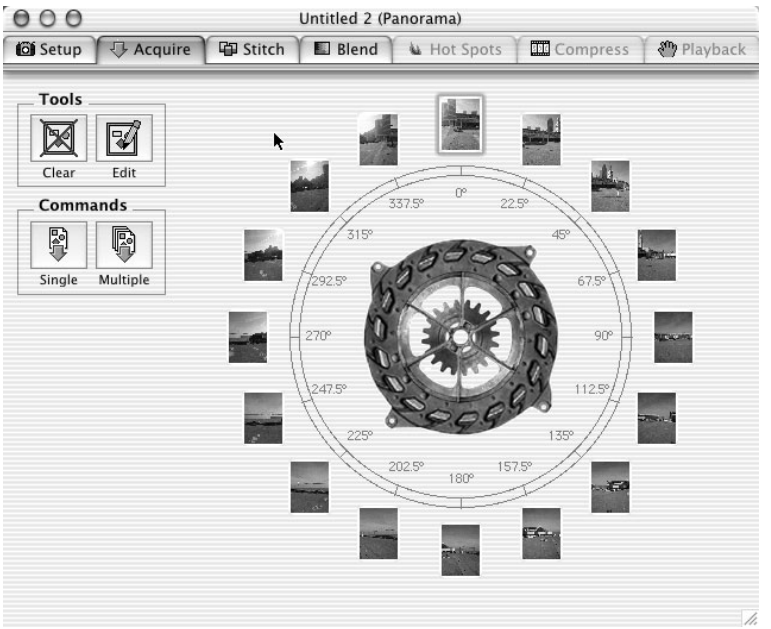


Abbildung 10.1: The VR Worx ist eines von vielen nützlichen QuicktimeVR-Tools.

10.1 Experiment 1: Panorama-Effekt mit Bitmap-Sprites

Zunächst wollen wir eine vorbereitete Bitmap als Panorama auf der Bühne bewegen. Wir können schon ahnen, dass das mit einem einzigen Sprite nicht möglich sein wird, denn das würden wir unweigerlich rechts oder links von der Bühne schieben. Die Idee ist daher, zwei Sprites zu benutzen, und dasjenige, das sich auf der einen Seite aus dem Bildbereich herausschiebt, auf der anderen wieder anzusetzen.

Versuchsaufbau

Im Beispielfilm *Lingopano.dir* haben wir drei Beispielpanoramen für Ihre Experimente vorbereitet. Platzieren Sie die Bitmap zweimal im Drehbuch, in direkt übereinander liegenden Drehbuchkanälen. Achten Sie darauf, dass die vertikale Position der beiden Sprites identisch ist.

Ein Stopperskript soll den Abspielkopf im Bereich der beiden Sprites anhalten. Das Drehbuch und die Bühne sehen dann aus wie in Abbildung 10.2.

Abbildung 10.2: Bühne und Drehbuch des Pano-Experiments

Durchführung

In Abbildung 10.3 sind drei Stationen unserer Panorama-Sprites für die Bewegung nach rechts veranschaulicht. Das Double ist in dieser Übersicht zur Verdeutlichung grau dargestellt:

- Station 1: Beide Sprites bewegen sich nach rechts.
- Station 2: Sobald sich das nachfolgende Sprite zur Hälfte im Bühnenbereich befindet...
- Station 3: ...wird es gegen das andere Sprite ausgetauscht. Beide bewegen sich dann weiter nach rechts.
- Station 4: (nicht abgebildet, da identisch zu Station 1) Das andere Sprite springt wieder nach links.

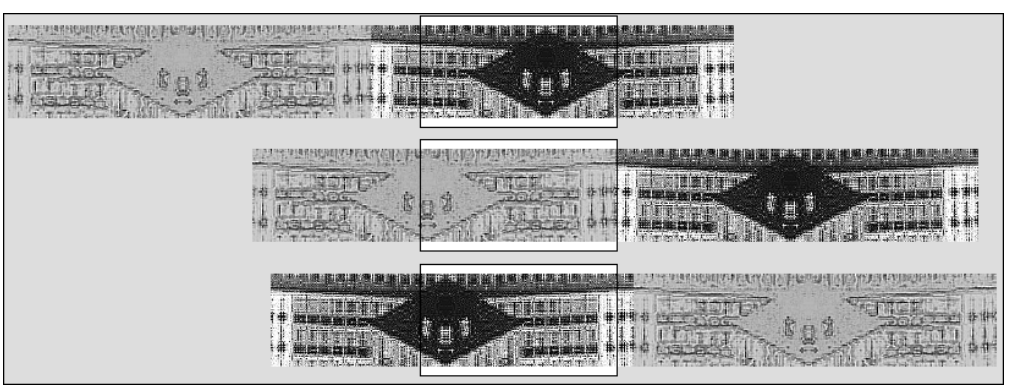


Abbildung 10.3: Die beiden Panorama-Darsteller wechseln die Position, so dass der Eindruck einer kontinuierlichen Bewegung entsteht.

Nun geht es darum, eine Anweisung zu formulieren, die eine kontinuierliche Bewegung im Bühnenausschnitt sicherstellt. Es muss u.a. eine Instanz geschaffen werden, welche das Maß der Vorwärtsbewegung („Schub“) steuert. Sprite- und Bühnengröße sind in ein Verhältnis zu bringen. Abbildung 10.4 zeigt alle Ortseigenschaften, die für die Horizontale relevant sind.

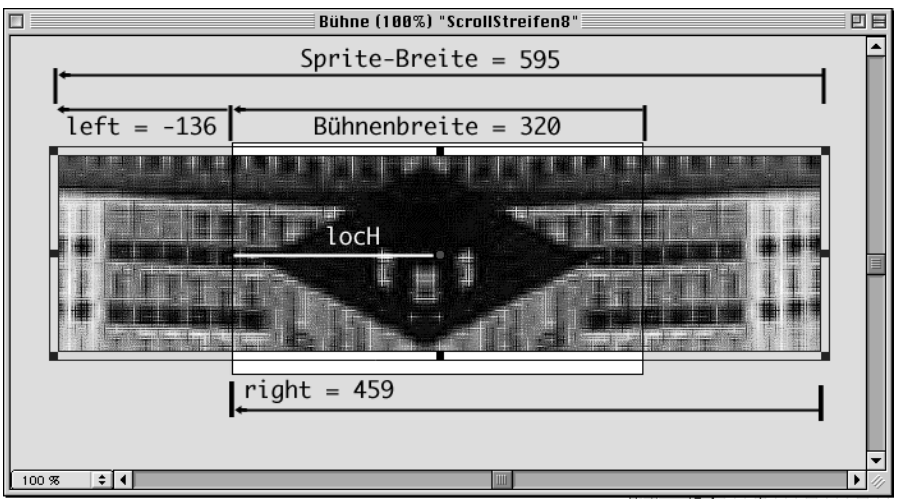


Abbildung 10.4: Maßgebliche Ortsangaben in ScrollStreifen.dir

Die Einheiten in Abbildung 10.4 werden in Lingo so ausgedrückt: die Sprite-Breite als `sprite(me.spritenum).rect.width`, die Bühnenbreite als `(the stage).rect.width`, der Abstand von linker Bühnenbegrenzung bis zur linken Spritekante als `sprite(me.spritenum).rect.left`, der Abstand der rechten Sprite-Kante vom linken Bühnenrand als `sprite(me.spritenum).rect.left` und der Abstand des Registrierungspunktes von der linken Bühnenkante als `sprite(me.spriteNum).locH`.

Das folgende Behavior implementiert die kontinuierliche Bewegung von *beiden* Sprites. Wir platzieren es auf Sprite 3. Das eigene Sprite wird als `msp` angesprochen, das andere (in Sprite 2) als `osp`:

```
property msp, osp
property bBreite, sBreite
property step

on beginSprite me
  msp = sprite(me.spritenum)
  osp = sprite(me.spritenum - 1)
  step = 1
  bBreite = (the stage).rect.width
  sBreite = msp.rect.width
end
```

```
on prepareFrame me
  if step <> 0 then
    newloc = msp.locH + step
    if newloc > bBreite/2 then
      osp.locH = newloc - sBreite
      if msp.rect.left > bBreite then msp.locH = newloc - sBreite
      else msp.locH = newloc
      --
    else
      osp.locH = newloc + sBreite
      if msp.rect.right < 0 then msp.locH = newloc + sBreite
      else msp.locH = newloc
    end if
  end if
end
```

Im `prepareFrame`-Handler wird zunächst `step` überprüft. Wenn die Property den Wert 0 hat, muss das Skript nicht weiter ausgeführt werden. In der lokalen Variable `newLoc` speichern wir den neuen Ort, den das Hauptsprite `msp` einnehmen soll. Ob es tatsächlich dahin platziert wird (oder um eine Spritebreite verschoben platziert wird), hängt davon ab, wo genau `newLoc` sich befindet.

Das folgende Behavior platzieren wir ebenfalls auf Sprite 3: Es steuert die Interaktion mit dem Panorama-Bild. Wir merken uns bei `mouseDown` die horizontale Mausposition in der Property `startH` und setzen im `prepareFrame`-Handler die Eigenschaft `step` unseres Panorama-Skriptes. Das ganze soll nur ausgeführt werden, solange die Maus gedrückt ist: Die Property `pressed` wird entsprechend bei `mouseDown`, `mouseUp` und `mouseUpOutside` verändert und im `prepareFrame`-Handler überprüft.

```
property startH, pressed

on mouseDown me
  startH = the mouseH
  pressed = true
end

on mouseup me
  pressed = false
end

on mouseUpOutside me
  mouseUp me
end

on prepareFrame me
  if pressed then
    sprite(me.spriteNum).step =  (the mouseH - startH) / 10.0
```

```
  else
    sprite(me.spriteNum).step = 0
  end if
end
```

Wenn Sie die Zeile

```
    sprite(me.spriteNum).step = 0
```

auskommentieren, dann läuft die Bewegung mit konstanter Geschwindigkeit weiter, auch wenn wir die Maus loslassen.

Da auch Sprite 2 angezeigt wird (und somit auch angeklickt werden kann), müssen wir sicherstellen, dass die Mausevents, die Sprite 2 erhält, an Sprite 3 weitergeleitet werden. Das folgende Skript liegt auf Sprite 2:

```
on mouseDown me
  sendSprite(me.spriteNum + 1, #mouseDown)
end

on mouseUp me
  sendSprite(me.spriteNum + 1, #mouseUp)
end
```

Es tut nichts anderes, als alle `mouseDown`- und `mouseUp`-Events an das Sprite weiterzuleiten, dessen Spritenummer um 1 höher ist als die eigene.

- **Erweiterung:**

Einen Vorschlag zur Erweiterung der Panorama-Lösung haben wir noch: Ändern Sie den Cursor! Mit folgendem Skript zeigen wir eine (offene) Hand an, wenn sich die Maus über dem Sprite befindet, und einen Rechts-Links-Pfeil, wenn die Maus gedrückt ist (zu den Cursorformen vgl. auch Kapitel 8, Experiment 1).

```
on beginSprite me
  sprite(me.spriteNum).cursor = 260
end

on mouseDown me
  sprite(me.spriteNum).cursor = 285
end

on mouseUp me
  sprite(me.spriteNum).cursor = 260
end
```

10.2 Experiment 2: QuicktimeVR – Swing-Bewegung kontrollieren

In Experiment 1 mussten wir noch einigen Lingo-Aufwand treiben, um das Panorama an die Mausbewegung zu koppeln. Mit QuicktimeVR geht das von alleine: Sie importieren einen QuicktimeVR-Film in Director und ziehen ihn auf die Bühne – das ist alles.

QuicktimeVR bietet neben den Drehbewegungen auch Zoom-In und -Out, sodass Sie Bildausschnitte heranzoomen oder die Ansicht im Weitwinkel betrachten können.

Eine automatische Drehung allerdings gibt es auch hier nur mit Lingo.

Versuchsaufbau

Legen Sie eines der QuicktimeVR-Panoramen aus dem Verzeichnis *Panos* auf der CD-ROM in Sprite 1 Ihres Films. Platzieren Sie ein Stopper-Skript im Skriptkanal und starten Sie den Film.

Öffnen Sie den Watcher und tragen Sie folgende Eigenschaften über das „Add“-Feld zur Beobachtung ein:

```
sprite(1).pan
sprite(1).fieldofview
sprite(1).tilt
```

Bevor wir mit eigenen Skripten loslegen: Benutzen Sie den QuicktimeVR-Film (Maus-Ziehen, ⇧- und Strg-Taste) und beobachten Sie die Werte im Watcher.

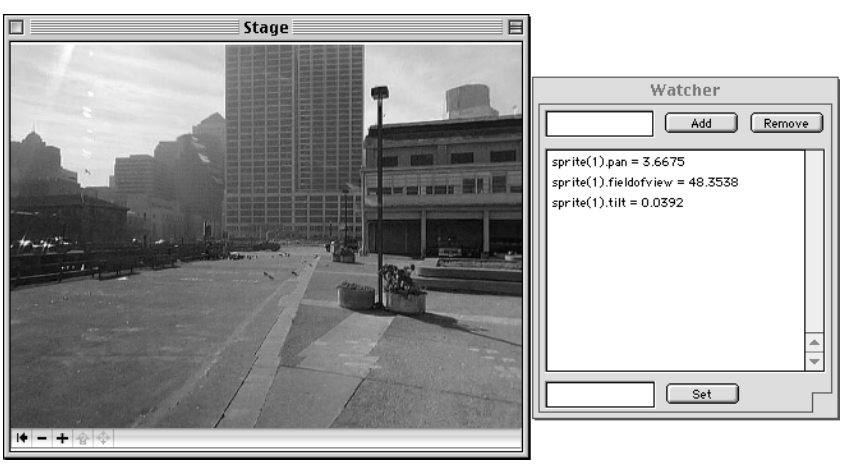


Abbildung 10.5: Im Watcher können Sie Eigenschaften des QuicktimeVR-Sprites beobachten.

Durchführung

Sie haben dank des Watchers sicher bereits ein Gefühl dafür bekommen, in welchen Wertebereichen sich `pan`, `fieldofview` und `tilt` bewegen. Je nach Vorgaben beim Erstellen des QuicktimeVR-Filmes können dies unterschiedliche Bereiche sein. `pan` bezeichnet den Rotationswinkel (häufig 0 - 360 Grad, also eine vollständige Umdrehung). `fieldofview` ist das Gesichtsfeld, kleine Werte zeigen einen sehr engen, große einen weiten Sichtbereich an; die Werte liegen häufig zwischen 1 und 60. `tilt` ist die Neigung, also die Abweichung aus der Horizontalen. Bei kleinem `fieldofview` sind höhere `tilt`-Werte möglich, Werte unter Null bezeichnen eine Neigung nach unten, Werte größer als Null entsprechend einen Blick in den Himmel.

Mit dem Befehl `swing(pan, tilt, fieldofview, speed)` können wir nun animierte Kamerafahrten veranstalten – auch im Nachrichtenfenster. Starten Sie den Film und geben Sie folgende Zeilen ein:

```
sprite(1).swing(60, 0, 40, 5)
sprite(1).swing(200, 10, 10, 5)
sprite(1).swing(0, -10, 60, 5)
```

Sie sehen, dass der Swing-Befehl eine Animation hin zu einer Ziel-Position ist. Diese Position wird durch `pan`, `tilt` und `fieldofview` eindeutig beschrieben; die Geschwindigkeit der Animation kann im vierten Parameter im Wertebereich von 1 (langsam) bis 10 (schnell) angegeben werden.

Eine 360-Grad-Drehung ist dabei aber nicht möglich: Quicktime sucht stets den kürzesten Weg zu einem Zielpunkt, so dass die Bewegung am besten in drei 120-Grad-Drehungen zerlegt wird. Das folgende Skript ermöglicht eine kontinuierliche Drehung:

```
property zielpan, myspeed
property msp
property counter, abweichung

on beginSprite me
  msp = sprite(me.spriteNum)
  msp.mouselevel = #shared
  myspeed = 4
  zielpan = (msp.pan + 120) mod 360
  abweichung = 0.25
  doSwing me
end

on doSwing me
  mytilt = msp.tilt
  myfov = msp.fieldofview
  msp.swing( zielpan, mytilt, myfov, myspeed )
end
```

```
on prepareFrame me
  if msp.pan >= (zielpan - abweichung) then
    if zielpan < 120 AND msp.pan > 240 then return
    counter = counter + 1
    -- put "Drehung: " & zielpan
    zielpan = (msp.pan + 120) mod 360
    doSwing me
  end if
end

on mouseup me
  doswing me
end
```

Das Skript setzt jeweils ein `zielpan` fest, das um 120 Grad höher ist als das aktuelle `pan` des Sprites. Die Funktion `doSwing()` übernimmt die aktuellen Werte für `tilt` und `fieldofview` und fügt in den Aufruf der Quicktime-Funktion `swing()` lediglich das neue `pan` ein. Da es keine automatische Benachrichtigung gibt, wenn der `swing()`-Befehl beendet ist, prüfen wir im `prepareFrame`-Handler, ob das `zielpan` (abzüglich einer geringen `abweichung`) erreicht ist. Falls ja, wird wiederum ein neues `zielpan` errechnet und `doSwing()` aufgerufen.

Die Bedingung `if msp.pan >= (zielpan - abweichung)` muss dann noch gesondert betrachtet werden, wenn es sich um den Bereich zwischen 0 und 120 Grad handelt. Dann ist das aktuelle `pan` nämlich unter Umständen höher als das `zielpan`, da `pan`-Werte bei 360 Grad wieder auf 0 springen. Diesen Sonderfall fangen wir in der Zeile

```
if zielpan < 120 AND msp.pan > 240 then return
```

ab: Wenn `zielpan` zwischen 0 und 120 liegt, dann werden hohe `pan`-Werte ignoriert: Die Skriptbearbeitung wird abgebrochen.

Sie finden diesen Film als *QTVRpano.dir* auf der CD-ROM.

10.3 Experiment 3: Shockwave3D – Leben im Würfel

Director 3D, wie es seit Director 8.5 möglich ist, geht bei der Implementierung virtueller Realitäten am weitesten. In diesem Experiment wollen wir den Aufbau eines Shockwave3D-Darstellers verstehen lernen – und unsere ursprüngliche Fragestellung einmal in dieser Technologie umsetzen. Wir nutzen dabei nur die Director-eigenen Mittel, so genannte „Primitives“ programmgesteuert zu erzeugen und zu animieren; das Experiment kommt also ohne Rückgriff auf (teure) 3D-Modelling- und Animationsprogramme aus.

Eigene 3D-Skripte erstellen

Das Lingo-Inventar, das für einen schnellen Einstieg ins 3D-Skripting nötig ist, ist überschaubar. Allerdings ist schon beim Umgang mit Primitives – den geometrischen Grundformen, die direkt aus Lingo erzeugt werden können – ein Verständnis der Hierarchien im Shockwave3D-Darsteller nützlich.

▶ **Paletten:**

Ein 3D-Darsteller enthält – in so genannten „Paletten" organisiert – Objekte der folgenden Typen:

- Modelresource
- Model
- Shader
- Texture
- Motion
- Light
- Camera
- Group

Diese „Paletten" sind im Darsteller ähnlich wie Besetzungen des Director-Films organisiert. Sie liegen in der Hierarchie direkt unterhalb des Darstellers und können entsprechend adressiert werden. Mit den Funktionen `newOBJEKTTYP()` und `deleteOBJEKTTYP()` werden entsprechende Einträge für alle Objekttypen erzeugt bzw. entfernt.

Legen Sie mit dem Menübefehl Einfügen/Mediaelement/Shockwave3D einen leeren 3D-Darsteller an. Nennen Sie ihn „Shockwave3D" und ziehen Sie ihn auf die Bühne.

Die folgenden Zeilen können Sie im Nachrichtenfenster ausprobieren:

```
mmb = member("Shockwave3D")
put mmb.model.count
-- 0
Res = mmb.newModelResource("Kugelres",#sphere)
Mod = mmb.newModel("Kugel", Res)
updateStage
put mmb.model("Kugel")
-- model("Kugel")
put mmb.model[1]
-- model("Kugel")
put mmb.modelresource[1]
-- modelResource("Kugelres")
mmb.deleteModel(1)
mmb.deleteModelResource("Kugelres")
updateStage
```

Sie sehen, dass wir durchgängig auf den Darsteller – nicht etwa auf das Sprite – zugegriffen haben. Aus einer (kugelförmigen) Modelresource

wurde ein Modell „Kugel“ erzeugt. Die Paletten „Model“ und „Modelresource“ enthalten jeweils ein Element, und wir können per Listensyntax darauf zugreifen. Im Sprite auf der Bühne werden die Änderungen sichtbar, wenn Sie ein `updateStage` senden.

▶ **Hierarchie:**

Neben dem „Paletten“-Modell, das es ermöglicht, Objekte im 3D-Darsteller zu erzeugen, zu verändern und zu löschen, sind die Abhängigkeiten der Objekte untereinander wichtig.

Die Hierarchie für Modelle – die uns hier zunächst hauptsächlich beschäftigen soll – sieht so aus:

Model
    Modelresource
    Shader(s)
        Texture(s)

Das heißt, ein Model beruht auf einer Modelresource (diese ist zum Erzeugen des Models notwendig), und dem Model werden sodann Shader zugewiesen (die seine Oberflächeneigenschaften und die Art des Renderings beschreiben). Einem Shader wiederum können Texturen zugewiesen werden, also Bitmap-Bilder, die innerhalb eines Shaders unterschiedliche Funktionen (z.B. als *Reflection Map*) erfüllen können.

Die „Welt“ des 3D-Darstellers (*World*) ist die oberste Hierarchieebene. Es ist die Gruppe, die immer vorhanden ist und alle weiteren Objekttypen enthalten kann:

Group (World) = Darsteller
    Model
        Shader
        Motion
    Model
        ...
    Group
        Model
        Light
        Camera
        Motion
        ...
    Camera
    Light
    ...

Das Sprite – also die Visualisierung eines 3D-Darstellers auf der Bühne – ist der Blick durch eine (oder mehrere) Kameras des Darstellers. Die Standard-Einstellung für `sprite(s).camera` ist dabei stets die erste Kamera des Darstellers.

▶ **Das Grundgerüst für einfache 3D-Behaviors:**

Änderungen, die wir mit Lingo an Shockwave3D-Darstellern vornehmen, werden nicht in der Director-Datei gespeichert. Das hat zur Folge, dass wir Shockwave3D-Darsteller, die wir mit Lingo erstellen und/oder manipulieren, im Normalfall jedes Mal neu erzeugen müssen.

Das folgende Skript-Gerüst tut dies im `beginSprite`-Handler. Es ist in folgende Abschnitte gegliedert:

1. Darsteller vorbereiten
2. Modelresource erzeugen und einstellen
3. Model erzeugen und einstellen
4. Shader erzeugen und einstellen
5. Textur(en) erzeugen und einstellen
6. Licht(er) erzeugen und einstellen
7. Kamera einstellen

Für jedes Modell, das wir in einem Darsteller erzeugen, wiederholen sich die Schritte 2 bis 5.

Im folgenden Skript finden Sie die Schritte am einfachsten Beispiel – ein Primitive ohne Textur wird in die Mitte der „Welt" platziert – exemplarisch aufbereitet. Der dazugehörige Beispielfilm ist der Film *3DBasis.dir*.

Besondere Beachtung verdienen die Funktion `resetworld()` in Schritt 1 sowie die Überprüfungen `if voidP(…) then …` bei jedem der folgenden Schritte. `resetworld()` setzt den Zustand des 3D-Darstellers auf den im Darsteller gespeicherten Zustand zurück. In unserem Fall – wir erzeugen einen leeren Darsteller in Director – ist dies also der leere Zustand.

Die `voidP()`-Überprüfungen stellen sicher, dass eine gleichnamige Ressource in der entsprechenden Palette noch nicht existiert. Dies wird dann interessant, wenn Sie mehrere Behaviors dieses Typs auf ein 3D-Sprite platzieren, oder wenn eine importierte w3d-Datei Grundlage Ihrer Versuche ist. Das Skript erzeugt demnach nur Modelresources, Models, Shaders, Textures und Lights, wenn gleichnamige Elemente noch nicht existieren.

```
on beginsprite me
  msp = sprite(me.spritenum)
  mmb = msp.member

  -- -------------------------
  -- 1. Darsteller vorbereiten
  -- -------------------------
  mmb.resetworld()
  -- World-Properties einstellen
  mmb.bgcolor = rgb(0,0,0)
  mmb.ambientcolor = rgb(128,128,128)
  mmb.directionalpreset = #none
  mmb.directionalcolor = rgb(255,0,0)
```

```
  -- --------------------------
  -- 2. Modelresource anlegen
  -- --------------------------
  if voidP(mmb.modelresource("resKugel")) then
    Res = mmb.newModelResource("resKugel", #sphere)
  else
    Res = mmb.modelresource("resKugel")
  end if

  -- --------------------------
  -- 3. Model anlegen
  -- --------------------------
  if voidP(mmb.model("Model1")) then
    mymod = mmb.newModel("Model1", Res)
  else
    mymod = mmb.model("Model1")
  end if

  -- --------------------------
  -- 4. Shader anlegen und zuweisen
  -- --------------------------
  if voidP(mmb.shader("shad")) then
    myShad = mmb.newShader("shad",#standard)
    myShad.diffuse = rgb(255,0,0)
    myShad.renderStyle = #fill
    myShad.texturelist = VOID
  else
    myShad = mmb.shader("shad")
  end if
  mmb.model("Model1").shaderList = mmb.shader("shad")

  -- --------------------------
  -- 5. Textur anlegen und zuweisen
  -- --------------------------

  -- --------------------------
  -- 6. Light anlegen
  -- --------------------------
  if voidp(mmb.light("li1")) then
    mylight = mmb.newlight("li1", #point)
    myLight.transform.position = \
     vector( -66.0000, -66.0000, 44.0000 )
    mylight.color = rgb( 255, 255, 255 )
    mylight.specular = 1
  end if

  -- --------------------------
  -- 7. Kamera einstellen
  -- --------------------------
  mmb.camera[1].transform.position = \
   vector( 0.0000, 0.0000, 100.0000 )
end
```

Schritt 1 setzt den Darsteller auf seinen Ausgangszustand zurück und setzt die Welteigenschaften, die auch im Property-Inspektor eingestellt werden können: Hintergrundfarbe und Ambientcolor sowie ein Spotlight und dessen Farbe (hier mit `#none` ausgeschaltet).

Schritt 2 legt eine Modelresource mit dem Namen `"resKugel"`, dem Typ `#sphere` und den Standard-Einstellungen an.

Schritt 3 generiert aus der Modelresource ein Modell mit dem Namen „Model1“. Da keine Position angegeben wird, wird das Modell im Koordinatenursprung platziert (`vector(0,0,0)`).

Schritt 4 legt einen neuen Shader `"shad"` mit folgenden Besonderheiten an: Er ist gefüllt, hat keine Textur (auch nicht das Standard-Schachbrettmuster), sondern lediglich eine Farbe.

Schritt 5 ist daher leer gelassen: Wir generieren in diesem Beispiel noch keine Textur.

In Schritt 6 legen wir ein eigenes (weißes und punktförmiges) Licht an, das an einen durch einen Vektor ausgedrückten Platz `vector(-66.0000, -66.0000, 44.0000)` verschoben wird, so dass es einen Lichtfleck auf der Kugel erzeugt. Einen ähnlichen Effekt erreichen Sie, wenn Sie in Schritt 1 die Eigenschaft `directionalpreset` beispielsweise auf `#bottomleft` setzen.

Schritt 7 schließlich modifiziert Kamera 1, also die dem Darsteller zugehörige `camera("DefaultView")`. Die Standard-Einstellung dieser Kamera ist `vector(0, 0, 250)`; mit `vector (0, 0, 100)` verringern wir den Abstand der Kamera vom Weltmittelpunkt (und damit von der Kugel, die dort platziert wurde).

Das ganze Skript wird auf `beginSprite` ausgeführt; Sie müssen also den Film starten, um etwas zu sehen.

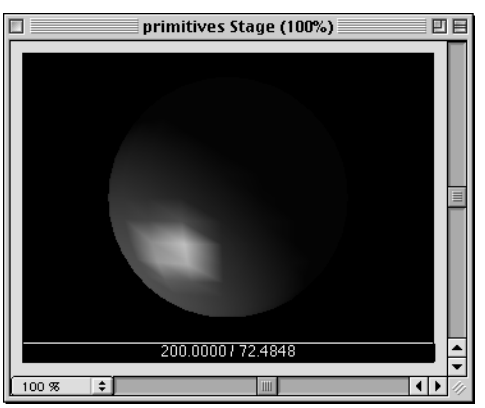


Abbildung 10.6: Das Ergebnis des ersten 3D-Skripts

▶ **Variationen:**

Verändern Sie den `renderStyle` des Shaders, um die Kugel in Wireframe- oder Punkt-Darstellung anzuzeigen:

```
myShad.renderStyle = #wire
```

oder

```
myShad.renderStyle = #point
```

In beiden Fällen nutzt die Anzeige die durch die Eigenschaft `diffuse` vorgegebene Farbigkeit.

Mit Veränderungen in Schritt 4 und 5 können Sie eine texturierte Kugel erzeugen:

```
-- -------------------------
-- 4. Shader anlegen und zuweisen
-- -------------------------
if voidP(mmb.shader("shad")) then
  myShad = mmb.newShader("shad",#standard)
  myShad.renderStyle = #fill
  myshad.texturemode = #none
  myshad.textureTransform.scale(1/8.0, 1/8.0, 1.0)
end if
mmb.model("Model1").shader = mmb.shader("shad")

-- -------------------------
-- 5. Textur anlegen und zuweisen
-- -------------------------
if voidP(mmb.texture("textu")) then
  mytextu = mmb.newTexture("textu", \
   #fromCastMember, member("text1"))
end if
mmb.shader("shad").texture = mytextu
```

Der Shader, der hier generiert wird, ist so skaliert, dass die Textur, die ihm zugewiesen wird, in x- und y-Richtung genau 8-mal wiederholt wird (Einstellung `textureTransform.scale()`). Als Textur dient ein Darsteller. Das Ergebnis dieses Skripts sehen Sie in Abbildung 10.7. Der entsprechende Beispielfilm ist *3DTextur.dir*.

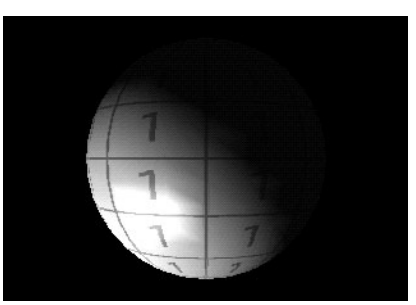


Abbildung 10.7:
Die Kugel mit einer gekachelten Bitmap-Textur

Soll die Textur nicht gekachelt sein, sondern genau einmal um den Körper herumprojiziert werden, so entfernen Sie die Zeile

```
myshad.textureTransform.scale(1/8.0, 1/8.0, 1.0)
```

Beachten Sie, dass wir hier durchgängig die „einfache Notation“ für Shader- und Textureigenschaften verwenden, da nur eine Textur bzw. ein Texturlayer angelegt wurden. Ein Shader kann bis zu 8 Textur-Layer enthalten, die jeweils unterschiedliche Funktionen haben. Die folgenden Statements:

```
myshad.textureTransform.scale(1/8.0, 1/8.0, 1.0)
mmb.model("Model1").shader = mmb.shader("shad")
mmb.shader("shad").texture = mytextu
```

sind gleichbedeutend mit:

```
myshad.textureTransformList[1].scale(1/8.0, 1/8.0, 1.0)
mmb.model("Model1").shaderList[1] = mmb.shader("shad")
mmb.shader("shad").textureList[1] = mytextu
```

Versuchsaufbau

Nutzen Sie den Film *3Dbasis.dir*, den wir mit dem Basis-Behavior (s.o.) für Lingo-generierte Primitives „fit“ gemacht haben.

Legen Sie bitte sechs unterschiedliche Texturen (Bitmapdarsteller) an. Idealerweise sollten sie quadratisch und beispielsweise 128x128 oder 256x256 Pixel groß sein. Andere Größen sind aber auch möglich, da Director Texturen auf nutzbare Maße skaliert.

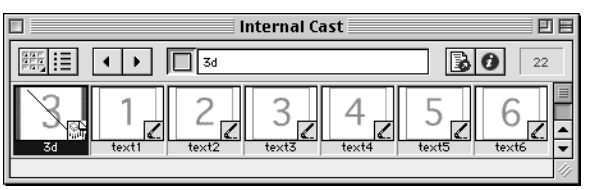


Abbildung 10.8: Texturdarsteller

Durchführung

Das Ziel dieses Experiments ist es, einen Würfel mit sechs unterschiedlichen Texturen zu erzeugen und Kamera und Licht in das Würfelinnere zu verlegen. Anschließend werden wir uns um die Kamera-Bewegung kümmern, damit wir uns in dem Würfel-Raum drehen können.

Zunächst modifizieren wir das 3D-Basis-Behavior. Die Kommentare im Skript zeigen Ihnen, wo wir Änderungen vorgenommen haben:

```
property a, mmb

on beginsprite me
  msp = sprite(me.spritenum)
  mmb = msp.member

  -- -------------------------
  -- 1. Darsteller vorbereiten
  -- -------------------------
  mmb.resetworld()
  -- World Properties einstellen
  mmb.bgcolor = rgb(0,0,0)
  mmb.ambientcolor = rgb(128,128,128)
  mmb.directionalpreset = #none
  mmb.directionalcolor = rgb(255,255,255)

  -- -------------------------
  -- 2. Modelresource anlegen
  -- -------------------------
  if voidP(mmb.modelresource("resBox")) then
```

Wir erzeugen eine Modelresource vom Typ #box*, deren Innenseiten (bzw. Rückseiten,* #back*) sichtbar sind.*

```
    Res = mmb.newModelResource("resBox", #box, #back)
  else
    Res = mmb.modelresource("resBox")
  end if

  -- -------------------------
  -- 3. Model anlegen
  -- -------------------------
  if voidP(mmb.model("Model1")) then
    mymod = mmb.newModel("Model1", Res)
  else
    mymod = mmb.model("Model1")
  end if

  -- -------------------------
  -- 4./5 Shader und Texturen anlegen und zuweisen
  -- -------------------------
```

Da wir mehrere Shader mit mehreren Texturen benötigen, kombinieren wir hier Schritt 4 und 5.
Die Liste textureMemberList *enthält die Darstellernamen der sechs Textur-Darsteller.*

```
  textureMemberList = ["text3", "text2", "text4", "text5", \
   "text1","text6"] -- hinten rechts vorn links oben unten
```

```
-- Für jede der sechs Seiten einen Shader und eine Textur
   erzeugen
repeat with a = 1 to 6
```

Für jede der sechs Würfelseiten generieren wir einen Shader und eine Textur.

```
  thisName = textureMemberList[a]
  if voidP(mmb.shader(thisname)) then
    myShad = mmb.newShader(thisName,#standard)
    myShad.ambient = rgb(255,255,255)
    myShad.diffuse = rgb(255,255,255)
    myShad.renderStyle = #fill
    myShad.specular = rgb(255,255,255)
  else
    myShad = mmb.shader(thisname)
  end if
  if voidP(mmb.texture(thisName)) then
    mytextu = mmb.newTexture(thisName, #fromCastMember, \
     member(thisName))
  else
    mytextu = mmb.texture(thisName)
  end if
```

Die Textur wird dem Shader zugewiesen. Sie wird dabei leicht skaliert und verschoben, um „Blitzer" an den Kanten der Würfelseiten zu vermeiden.

```
  myshad.texture = mytextu
  myshad.texturetransform.scale=vector(1.01,1.01,1.01)
  myshad.texturetransform.position= \
   vector(-0.005,-0.005, -0. 005)
```

Der Shader wiederum wird einer Würfelseite des Modells zugewiesen.
Die shaderlist *des Modells wird am Ende der* repeat*-Schleife also sechs Shader an den Positionen 1 bis 6 enthalten.*

```
  mymod.shaderList[a] = myshad
end repeat

-- -------------------------
-- 6. Light anlegen
-- -------------------------
```

Wir legen zwei Lichter an: ein Ambient- und ein Spot-Licht. Das Ambient-Licht ist dunkelgrau und wird die Szene nur wenig erhellen.

```
if voidp(mmb.light("li1")) then
  mylight = mmb.newlight("li1", #ambient)
  mylight.color = rgb( 32,32,32 )
  mylight.specular = 1
end if
```

```
  Das zweite Licht ist fast weiß; beim Lichttyp #spot können
  wir den spotAngle (Öffnungswinkel des Spots) und die
  Eigenschaft spotDecay einstellen. Ist spotDecay 0, so werden
  die Ränder des Lichts nicht weich gezeichnet; beim Wert 1
  geschieht dies.
if voidp(mmb.light("li2")) then
  mylight = mmb.newlight("li2", #spot)
  myLight.transform.position = \
   vector(0.0000, 0.0000, 180.0000)
  myLight.transform.rotation = \
   vector(0.0000, 0.0000, 0.0000)
  mylight.color = rgb( 238, 238, 238 )
  mylight.specular = 1
  mylight.spotAngle = 90.0000
  mylight.spotDecay = 1
else
  mylight = mmb.light("li2")
end if
  Das Spot-Licht wird an die Kamera gebunden: Wenn wir später
  die Kamera drehen, wird das Licht diese Drehung mitmachen
  (und die Szene immer aus der Kamera-Perspektive beleuchten,
  da auch der Rotationsvektor des Lichts vector(0,0,0) ist).
mylight.parent = mmb.camera("DefaultView")
-- --------------------------
-- 7. Kamera einstellen
-- --------------------------
  Die Kamera soll genau im Weltmittelpunkt stehen. Die Eigen-
  schaft fieldofView legt fest, wie groß der Sichtbereich der
  Kamera ist.
mmb.camera[1].transform.position = \
 vector(0.0000, 0.0000, 0.0000)
mmb.camera[1].fieldofView = 80.0
end
```

Bevor wir weiter experimentieren, wollen wir auch schon mal etwas von unserer Würfel-Welt sehen. Wenn Sie den Film starten, ist der Eindruck allerdings noch sehr eingeschränkt, da der Würfel noch nicht beweglich ist. Öffnen Sie die Palette „Bibliothek“ (Menü Fenster/Bibliothek) und ziehen Sie das Behavior „Drag Model To Rotate“ (in 3D/Actions) und das Auslöser-Behavior „Mouse Left“ (in 3D/Triggers) auf das 3D-Sprite auf der Bühne. Abbildung 10.9 und Abbildung 10.10 zeigen die Einstellungen für die beiden Behaviors.

Sie finden diesen Stand als *3Dwuerfel.dir* auf der CD-ROM.

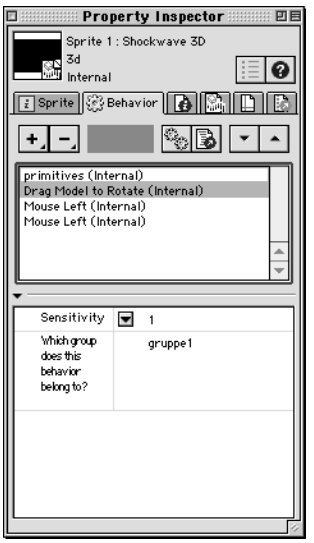


Abbildung 10.9: Einstellungsmöglichkeiten des Behaviors „Drag Model to Rotate“

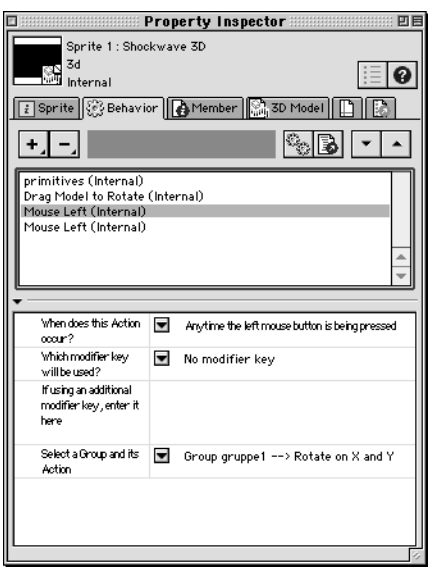


Abbildung 10.10: Das „Trigger“-Behavior „MouseLeft“

▸ Modifikationen

Probieren Sie andere Einstellungen für das Spot-Licht und für die Kamera aus. Wenn Sie die Kameraposition aus dem Zentrum herausbewegen, wird das deutliche Auswirkungen auf den Bewegungseindruck haben.

Ändern Sie im Auslöser „Left Mouse“ die Aktion, die das Behavior bewirken soll. Probieren Sie insbesondere die Einstellungen „Rotate on X axis“, „Rotate on Y axis“ und „Rotate on Z axis“ aus.

▶ Drehung implementieren

Statt vorgefertigte Behaviors für die Drehung zu verwenden, wollen wir uns nun ein kleines Skript schreiben, das die Kamera dreht. Löschen Sie die Bibliotheksbehaviors von unserem 3D-Sprite, und legen Sie ein neues, leeres Behavior auf ihm an.

Was soll das Skript leisten?

- Wie in den vorangehenden Experimenten soll das Ziehen mit der Maus eine Bewegung der Kamera in die entsprechende Richtung bewirken.
- Eine größere Mausbewegung soll eine schnellere Drehung in die entsprechende Richtung bewirken.

Das Skript ist sehr übersichtlich: Wir merken uns bei `mouseDown` die Klickposition in der Property `mloc`; im `prepareFrame`-Handler berechnen wir regelmäßig die Differenz zur aktuellen Mausposition (`diff`), zerlegen diese in x- und y-Differenzen (`diffx`, `diffy`), teilen sie durch einen Skalierungsfaktor (10.0) und multiplizieren mit -1, da die Bewegungsrichtung entgegen der Ziehbewegung sein soll. Der Befehl `rotate()` schließlich lässt die Kamera mit den errechneten Differenzwerten um die x- und y-Achse rotieren. Eine Rotation um die x-Achse entspricht dabei einer Bewegung in y-Richtung, entsprechend ist die Rotation um die y-Achse eine Bewegung in x-Richtung. Den dritten Parameter (Rotation um die z-Achse) setzen wir auf 0. Da unsere Kamera im Zentrum der Welt platziert ist, ist es egal, ob wir als vierten Parameter `#world` oder `#self` übergeben. Zur Kontrolle lassen wir uns den Rotationsvektor der Kamera ins Nachrichtenfenster ausgeben.

```
property pressed, mloc
property theCam

on mouseDown me
  pressed = true
  mloc = the mouseloc
  theCam = sprite(me.spriteNum).member.camera("DefaultView")
end

on mouseUp me
  pressed = false
end

on prepareFrame me
  if pressed then
    diff = the mouseLoc - mloc
    diffx = diff[1]
    diffy = diff[2]
    --
    stepx = -1 * diffx / 10.0
    stepy = -1 * diffy / 10.0
    --
```

```
      theCam.rotate(stepy, stepx, 0, #self)
      put theCam.transform.rotation
    end if
  end
```

In der Property `theCam` haben wir eine Abkürzung geschaffen, die es uns erspart, immer wieder `sprite(me.spriteNum).member.camera("DefaultView")` zu schreiben. Die Property `pressed` schließlich hält fest, ob die Maus auf dem Sprite gedrückt wurde; nur dann soll die Animation im `prepareFrame`-Handler stattfinden. `mouseUp` (und falls das Sprite nicht die ganze Bühne einnimmt auch `mouseUpOutside`) setzt `pressed` auf `false`.

Das Skript setzt die Anforderungen umfassend um – fast schon zu umfassend, denn innerhalb kürzester Zeit werden x- und y-Rotation dazu führen, dass wir über Kopf oder anderweitig verdreht in unserem Würfelraum stehen.

▶ Modifikation

Wir schaffen uns noch eine Möglichkeit, die einzelnen Bewegungen einzuschränken: Wenn wir bestimmte Tasten drücken, sollen einzelne Step-Werte gezielt verändert werden. Ersetzen Sie die Zeile `theCam.rotate (stepy, stepx, 0, #self)` durch folgende fünf Zeilen:

```
    if the shiftDown then stepx = 0
    if the optionDown then stepy = 0
    if the commandDown then stepz = 5
    else stepz = 0
    theCam.rotate(stepy, stepx, stepz, #self)
```

Die ⇧-Taste setzt also die x-Bewegung außer Kraft, die Alt- bzw. ⌥-Taste die y-Bewegung. Wenn wir Strg bzw. ⌘ drücken, rotiert die Kamera mit fester Schrittweite um die y-Achse.

Auch eine Reset-Möglichkeit wäre praktisch. Wenn wir die Taste R nutzen wollen, um in den Ausgangszustand zurückzukommen, so können wir zum Beispiel im `prepareFrame`-Handler folgende Zeile platzieren:

```
if keyPressed("r") then theCam.transform.rotation = \
 vector(0,0,0)
```

oder in einem `keyDown`-Handler:

```
on keyDown me
  if the key = "r" then theCam.transform.rotation = \
   vector(0,0,0)
end
```

Auch den abschließenden Stand finden Sie als *3Ddrehung.dir* auf der CD-ROM.

Diskussion

Das Grundgerüst, das Sie in diesem Experiment ausprobiert haben, kann Ihnen bei Ihren eigenen Versuchen in Director-3D-Skripting gute Dienste leisten. Im nächsten Kapitel werden wir es wieder nutzen, und wir zeigen Ihnen zusätzlich ein wichtiges Werkzeug, mit dem 3D-Authoring einfacher wird – und durch das die hunderte von Lingo-Befehlen (vgl. Abbildung 10.11) interaktiv erfahrbar werden.

Haben Sie sich einmal den Spaß gemacht, die hier zwischenzeitlich verwendeten Bibliotheks-Behaviors genauer anzuschauen? Sie sehen hunderte von Codezeilen, die gut kommentiert sind, aber in ihrer Komplexität kaum verständlich. Nutzen Sie diese Behaviors als schnelle Lösung, wenn Sie die Funktion selbst noch nicht geskriptet haben oder nicht skripten wollen. Als Ausgangspunkt für eigene Skripte sind sie denkbar ungeeignet.

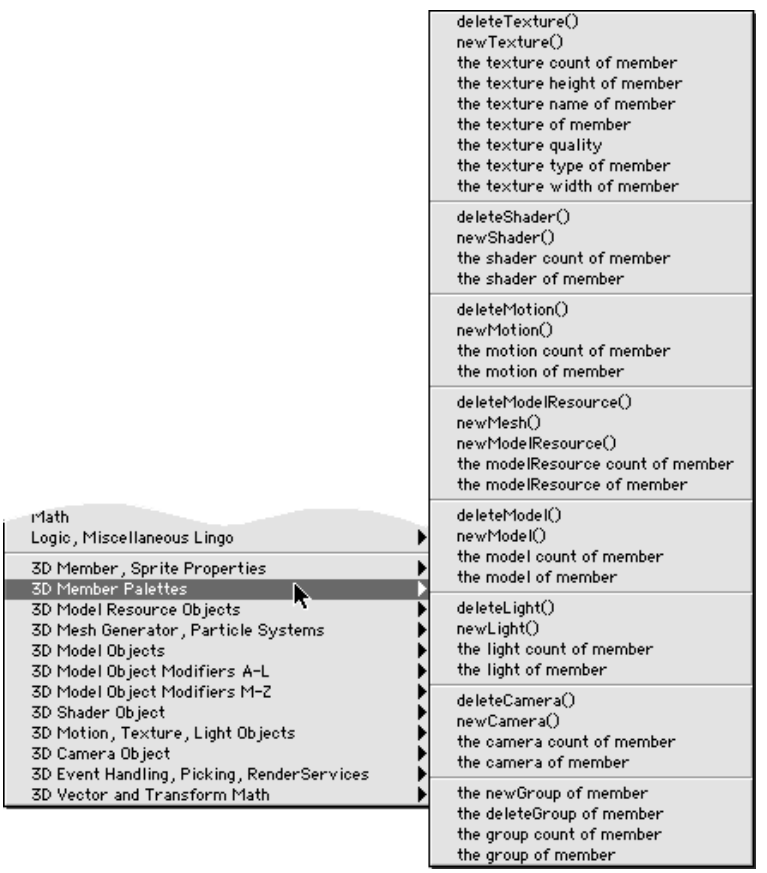


Abbildung 10.11: 3D-Befehle im Lingo-Menü

11

Die Taschenlampe

Sie kennen den Effekt: Eine Lampe wandert über ein abgedunkeltes Bild, und im Lichtkegel wird ein kleiner Ausschnitt sichtbar. Der Effekt ist – wenn man dem Anwender die Kontrolle über die „Lampe“ in die Hand gibt – eine hervorragende Möglichkeit, ihn für die Szene zu interessieren und zum Entdecken zu verführen. Auch eine Suchaufgabe oder zu entdeckende Klickbereiche sind denkbar.

Um den Taschenlampeneffekt umzusetzen, haben wir – je nach Director-Version – unterschiedliche Möglichkeiten:

1. Wir decken das Bild mit einer (schwarzen) Überleger-Grafik ab, durch die wir – mit dem Ink-Effekt „Hintergrund transparent“ – das Bild sichtbar machen (alle Director-Versionen); ab Director 7 kann der Überleger auch Transparenz-Informationen (einen Alphakanal) enthalten, so dass wir auf den Ink-Effekt verzichten können. Experiment 1 und 2 beruhen auf diesem Ansatz; bei beiden kommt nur einfaches Skripting zum Einsatz.
2. Auch der Ink-Effekt „Maske“ – in Kombination mit einem Maskendarsteller – führt zum Ziel; die Maske blendet große Teile des Darstellers aus, so dass der (schwarze) Bühnenhintergrund sichtbar wird. Dieser Ansatz – in Experiment 3 – ist ebenfalls mit überschaubarem Lingo-Inventar realisierbar.
3. Wir können ab Director 8 mit Imaging Lingo direkt den Alpha-Kanal des angeleuchteten Bildes verändern; auch so machen wir Teile des Bildes unsichtbar. In Experiment 4 werden wir uns also mit diesem spannenden und relativ neuen Bereich von Lingo auseinander setzen.
4. Wir können seit Director 8.5 in einem Shockwave3D-Darsteller direkt eine Lichtquelle über das dunkle Bild bewegen. Für dieses Experiment (Nr. 5) sind die Lingo-Anforderungen am höchsten.

Sie sehen, die Taschenlampe ist ein lohnendes Thema, um mit ganz unterschiedlichen grafischen Möglichkeiten von Director zu experimentieren. Die Arbeit am Darsteller und im Drehbuch greift hier wiederum eng ineinander mit der Lingo-Programmierung.

11.1 Versuchsaufbau

Der Film *Taschenlampe.dir* ist 320x240 Pixel groß und hat einen schwarzen Bühnenhintergrund. Auf Bild 1 und auf Bild 6 platzieren wir jeweils einen Marker: „Start“ und „Spielfiguren“. Im Spritekanal 1 werden in Bild 1-5 das zu entdeckende Bild (Darsteller „Schaufenster“) und in Bild 6-10 ein beliebiges anderes Bild platziert. Der Darsteller „Schaufenster“ ist genauso groß wie die Bühne und liegt an Position `point(0,0)` – dies ist eine Vorgabe, die wir beim weiteren Experimentieren noch verändern werden.

Im Skriptkanal platzieren wir auf Bild 5 und 10 jeweils unser „Stopper-Skript“:

```
on exitFrame me
  go to the frame
end
```

In Sprite-Kanal 3 im ersten Drehbuchabschnitt (1-5) platzieren wir über eine der Figuren ein kleines Shape, das wir mit dem ungefüllten Rechteck-Tool in der Werkzeugpalette und mit unsichtbarer Umrandung erzeugen: Es wird also vollkommen unsichtbar sein.

Abbildung 11.1:
Werkzeugpalette: Mit den Werkzeugen für ungefüllte Shapes können Sie unsichtbare Hotspots erstellen, wenn Sie die Liniendicke auf 0 (die gepunktete Linie) stellen.

Bei Klick auf dieses Shape soll zum Marker „Spielfiguren“ gesprungen werden. Das folgende Behavior „goSpielfiguren“ wird auf dem unsichtbaren Hotspot angelegt:

```
on mouseUp me
  go "Spielfiguren"
end
```

Da die Taschenlampe nicht durch den Maus-Cursor gestört werden soll, blenden wir diesen aus, solange wir uns im ersten Drehbuchbereich befinden. Dazu dient das Skript „Cursor weg“, das wir auf das Sprite 1 ziehen:

```
on beginSprite me
  sprite(me.spritenum).cursor = 200
end

on endSprite me
  sprite(me.spritenum).cursor = -1
end
```

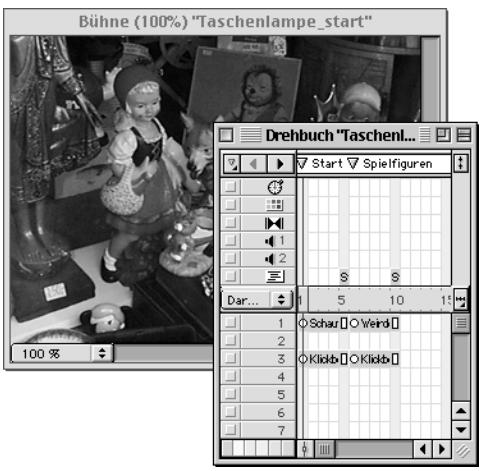


Abbildung 11.2: Der Ausgangszustand des Beispielfilms für die Experimente in diesem Kapitel

Damit wird der `cursor` des Sprites bei `beginSprite` auf die Nummer des unsichtbaren Cursors gesetzt, bei `endSprite` wieder auf den Standard-Cursor. Gegenüber der Syntax `cursor 200` hat dieses Skript den Vorteil, dass die Cursor-Änderung a) auf den Bereich des Sprites begrenzt ist und b) auch vorbei ist, sobald der Abspielkopf das Sprite verlässt.

Im zweiten Drehbuchbereich ist eine Rücksprungmöglichkeit – ebenfalls durch ein Skript auf Sprite 3 – vorbereitet.

11.2 Durchführung

Experiment 1: Überleger und Ink-Effekt

Die einfachste Umsetzung – was die Lingomittel angeht – erfordert zunächst ein bisschen Gedankenakrobatik: Anstatt nämlich etwas heller zu machen (wie die Taschenlampe), dunkeln wir das Bild mit einem schwarzen Überleger ab und bohren ein „Loch“ in diesen Überleger, durch das wir auf das dahinter liegende Bild schauen können. Die Lingo-Umsetzung beschränkt sich dann auf das Bewegen des Überlegers.

Dies sind die Vorbereitungen, die Sie im Malfenster von Director treffen müssen:

1. Zeichnen Sie ein schwarzes, gefülltes Rechteck. Die Größe ist zunächst noch egal.
2. Malen Sie mit dem Oval-Werkzeug ein kleineres, weißes Oval in das Schwarz.

Das Ergebnis sehen Sie in Abbildung 11.3. Platzieren Sie den Darsteller in Sprite 2 im ersten Drehbuchbereich (Frames 1-5).

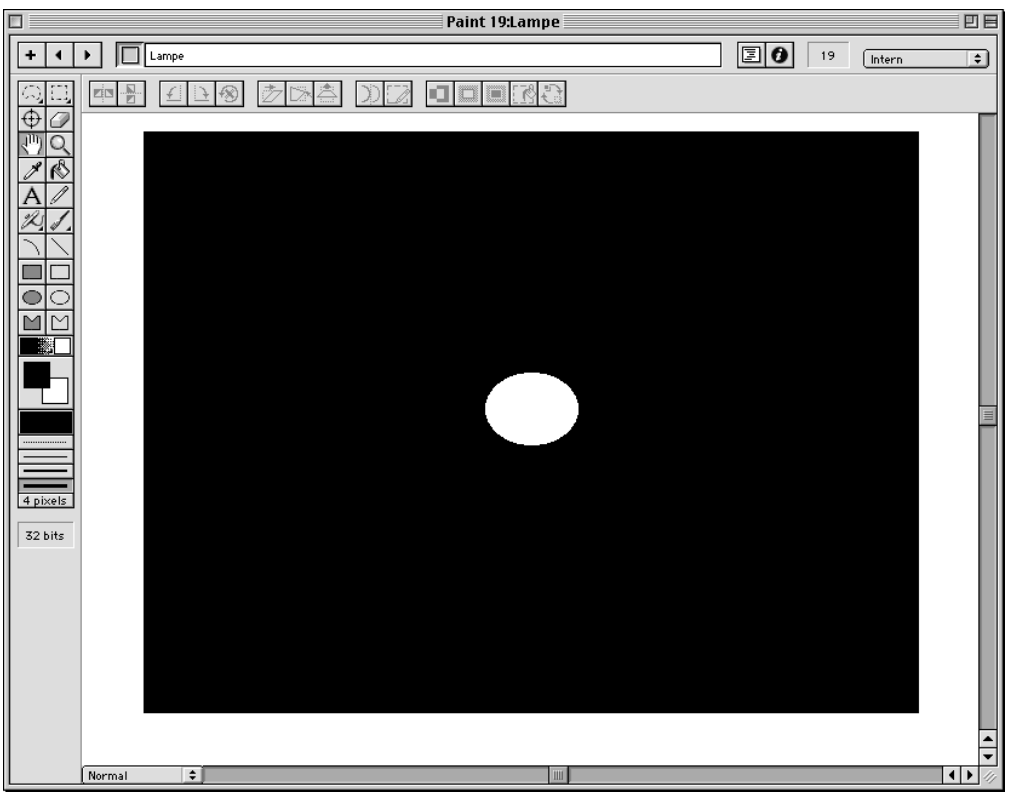


Abbildung 11.3: Der Darsteller „Lampe“: schwarze Fläche und weißer Fleck mit harten Kanten

Damit wir durch das Sprite hindurchschauen können, müssen wir die weiße Farbe durchscheinend machen. Dazu dienen Ink-Effekte (bzw. „Farbeffekte“): Sie verändern nicht den Darsteller, sondern die Art und Weise, wie der Darsteller im Sprite erscheint.

Probieren Sie auch die anderen Ink-Effekte aus!

Ein geeigneter Ink-Effekt ist „Hintergrund Transparent“. Sie können den Effekt über den Sprite- bzw. Property-Inspektor zuweisen oder ihn einfach im Aufklappmenü auswählen, das sich bei Strg / ⌘-Klick auf das Sprite öffnet.

Um das Sprites ziehbar zu machen, platzieren wir folgendes Behavior auf ihm:

```
on prepareFrame me
  sprite(2).loc = the mouseloc
end
```

Wenn Sie den Film abspielen, sehen Sie, dass das Ergebnis zwar schon in die richtige Richtung geht, aber noch nicht ausreichend ist: Der Überleger ist möglicherweise zu klein, oder er kann so weit auf eine Seite gezogen werden, dass das dahinter liegende Bild daneben sichtbar wird.

Es heißt also, sich über die Größe und den Bewegungsbereich des Überlegers Gedanken zu machen. Die Bild- und Bühnengröße ist 320 x 240 Pixel; wenn wir das „Loch", also die weiße Fläche in die Mitte des Überlegers platzieren, muss der Überleger genau doppelt so hoch und doppelt so breit sein wie die Bühne, damit wir uns mit der Maus an alle Bühnenränder bewegen können.

Wir müssen also unseren Überleger entsprechend korrigieren: Um eine genau 640x480 Pixel große schwarze Fläche zu erstellen, skalieren wir den bereits erstellten Darsteller mit dem Menübefehl Modifizieren/Bitmap Transformieren... auf diese Größe; dann malen Sie die Fläche am besten wieder ganz schwarz, zentrieren den Registrierungspunkt und malen mit dem weißen Oval wiederum ein weißes „Loch".

Eine weitere Vorkehrung müssen wir in unserem Skript einrichten: Wenn die Maus über den Bühnenrand bewegt wird, darf die „Lampe" ihr nicht weiter folgen als bis genau zum Rand. Eine ausführliche Formulierung dieser Bedingung finden Sie im folgenden Skript:

```
on prepareFrame me
  mPoint = the mouseLoc
  if the mouseH < 0 then mpoint[1] = 0
  else if the mouseH > 320 then mpoint[1] = 320
  if the mouseV < 0 then mpoint[2] = 0
  else if the mouseV > 240 then mpoint[2] = 240
  sprite(2).loc = mpoint
end
```

In der letzten Zeile wird die Position von Sprite 2 nicht mehr auf die Mausposition gesetzt, sondern auf `mPoint` – eine Punktangabe, die wir in den vorangehenden Zeilen selbst festlegen.

Die Punktangabe – in unserem Fall die Mausposition in der Form `point (117, 245)` – ist ein spezielles Lingo-Objekt, auf dessen Einzelwerte wir wie auf Listenelemente zugreifen können. `mPoint[1]` ist demnach der erste, der x-Wert des Punktes, `mPoint[2]` der zweite, y-Wert. Wenn die Maus in der Horizontalen links vom linken Bühnenrand (`mouseH < 0`), rechts vom rechten Rand (`mouseH > 320`) und entsprechend über oder unterhalb der Bühne ist, dann wird die entsprechende Angabe in mPoint geändert. `mPoint` wird damit an der ersten Position immer Werte zwischen 0 und 320, an der zweiten zwischen 0 und 240 enthalten.

Probieren Sie es aus – damit funktioniert unsere erste Taschenlampe...

Wenn Sie damit nicht zufrieden sind, können Sie das Skript mit den mathematischen Funktionen `min()` und `max()` deutlich verkürzen:

```
on prepareFrame me
  mpoint = point(min(320, max(0, the mouseH)), \
   min(240, max(0, the mouseV)))
  sprite(2).loc = mpoint
end
```

`min (3,5)` ergibt 3, `max(3,5)` ergibt 5 – um also einen Wert `x` auf den Bereich zwischen 1 und 10 einzuschränken müssen wir die beiden folgenden Bedingungen kombinieren: `min(x,10)` (nimm den `x`-Wert, solange er kleiner 10 ist, sonst nimm 10) und `max(1,x)` (nimm den `x`-Wert, wenn er größer 1 ist, sonst nimm 1). Die Kombination sieht so aus wie im obigen Skript: `min(10, max(1,x))` oder `max(1, min(5,x))`.

Den fertigen Film für dieses Experiment finden Sie als Director 7-Film *Taschenlampe_Ink.dir* auf der CD-ROM.

Falls Sie nicht die ganze Bühne abdecken wollen, sondern ein Sprite, das an beliebiger Stelle auf der Bühne liegt, können Sie die Berechnung von `mpoint` wie folgt verändern:

```
on prepareFrame me
  mpoint = the mouseLoc - point (sprite(1).rect.left, \
   sprite(1).rect.top)
  mpoint = point(min(320, max(0, mPoint[1])), min(240, \
   max(0, mPoint[2])))
  sprite(2).loc = mpoint
end
```

Die Berechnung von `mPoint` wird ergänzt durch die Verschiebung des Sprites auf der Bühne, indem wir die Punktangabe `point (sprite(1).rect.left, sprite(1).rect.top)` von der Mausposition abziehen, um die Darstellerkoordinaten zu erhalten. Danach begrenzen wir den Wertebereich wieder mit `min()` und `max()`.

Der Überleger müsste doppelt so groß sein wie das abzudeckende Sprite, und in höheren Spritekanälen würden Sie die Bühnenfläche um das „Schaufenster"-Sprite mit anderen Elementen abdecken (vgl. Abbildung 11.4).

Abbildung 11.4: Platzieren Sie Abdecker in höheren Spritekanälen, wenn der „Schaufenster"-Darsteller nicht auf der Nullposition der Bühne steht.

Experiment 2: Überleger und Darsteller mit Alpha

Wenn wir ein externes Bildbearbeitungsprogramm zu Hilfe nehmen, können wir die Ränder des Lichtkegels auch weich verlaufen lassen.

Legen Sie dafür in Fireworks (oder Photoshop oder einem anderen Bildbearbeitungsprogramm) eine 640 x 480 Pixel große Datei mit transparentem Hintergrund an, die Sie mit Schwarz füllen. Erstellen Sie eine runde Auswahl in der Mitte mit weichem Rand, und löschen Sie den ausgewählten Bereich. Das Bild besteht nun aus einer schwarzen Fläche und einem transparenten Bereich in der Mitte.

Sichern Sie die Datei als PNG (oder PSD oder Pict mit Alphakanal), und importieren Sie sie als Darsteller. Eine entsprechende Beispieldatei *Lampe.PNG* finden Sie auch auf der CD-ROM.

Setzen Sie den neuen Darsteller an Stelle des alten Lampendarstellers ein (z.B., indem Sie das Sprite auf der Bühne anwählen und mit ⌘+E / Strg+E den Darsteller austauschen).

Das Skript bleibt dasselbe – Ihre zweite Taschenlampe ist sofort einsetzbar.

▸ Diskussion:

Die Einschränkungen der beiden bisherigen Ansätze liegen auf der Hand: Wenn wir den Effekt nicht auf der ganzen Bühne einsetzen wollen, müssen

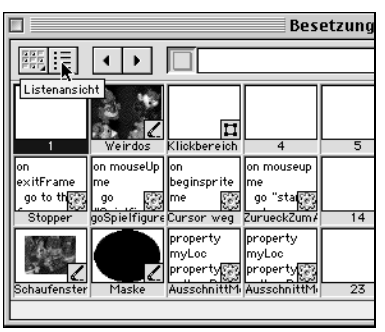
wir den Überleger mit weiteren Sprites, die in höheren Sprite-Kanälen liegen, abdecken. Das Bewegen des Lampen-Sprites wird außerdem bei größeren Bildern als in unserem Beispiel relativ viel Rechnerperformance erfordern. Es lohnt sich also, noch weitere Möglichkeiten auszuprobieren...

Experiment 3: Maskierung

Ebenfalls ein Ink-Effekt soll beim nächsten Experiment zum Einsatz kommen, nämlich der Modus „Maske“. Er hat eine Besonderheit: Obwohl er auf ein Sprite angewendet wird, benötigen wir nämlich eine bestimmte *Anordnung der Darsteller* in der Besetzung.

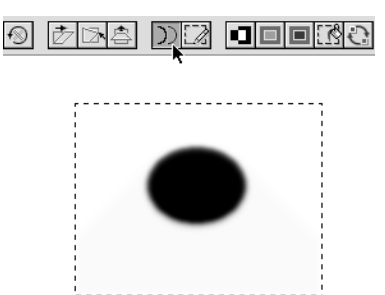
Nehmen Sie wieder den Ausgangsfilm (*Taschenlampe.dir*) und ziehen Sie den Darsteller „Schaufenster“ in der Besetzung auf eine Position, deren nachfolgender Darsteller leer ist. (Falls Sie in Director 8 oder neuer arbeiten, müssen Sie zuvor bei der Darstellung der Besetzung die „Listenansicht“ ausschalten, vgl. Abbildung 11.5) Im nachfolgenden Darsteller legen wir im Malfenster die Maske an.

Abbildung 11.5: In der Miniaturansicht können Sie die Darsteller der Besetzung neu anordnen.



Zeichnen Sie dafür einen schwarzen Kreis: Das maskierte Bild ist da sichtbar, wo der Maskendarsteller schwarz ist, und unsichtbar, wo er weiß ist. Mit dem Weichzeichnungstool im Malfenster können wir den Kreis auch gleich von seinen harten Kanten befreien (vgl. Abbildung 11.6).

Abbildung 11.6: Weichzeichnen im Malfenster



Auf der Bühne oder im Drehbuch weisen Sie dem Sprite 1 – also dem zu entdeckenden Bild – den Ink-Effekt „Maske“ zu. Sofort ist nur noch ein Teilbereich des Bildes sichtbar, ansonsten sieht man den schwarzen Bildschirmhintergrund.

Die Schwierigkeit liegt nun darin, die Maske entsprechend der Mausbewegung zu animieren. Da die Maske nicht im Drehbuch auftaucht, fällt der einfache Weg über `sprite(s).loc` weg.

Die Positionierung der Maske in Relation zum maskierten Darsteller hängt vielmehr vom Registrierungspunkt des Maskendarstellers ab.

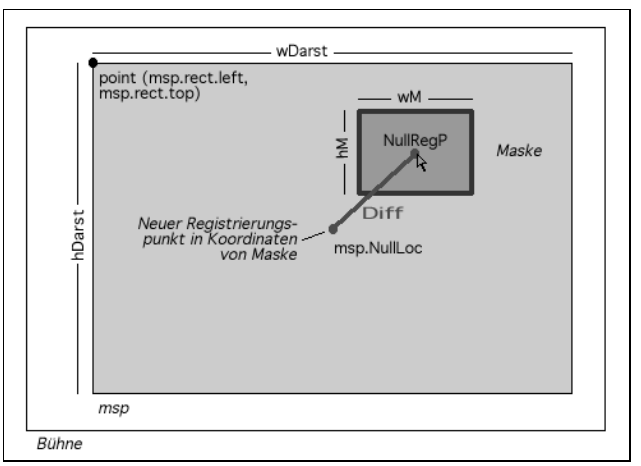


Abbildung 11.7: Um die Maske an der Stelle des Mauszeigers anzuzeigen, wird ihr Regpoint in Gegenrichtung verschoben und zwar genau so weit, dass er im NullLoc (also in der Mitte des Darstellers) des Bildsprites liegt.

Das folgende Skript bewegt den Registrierungspunkt des Maskendarstellers:

```
property msp
property myLoc
property nullregP, nullLoc

on beginsprite me
  msp = sprite(me.spriteNum)
  wM = member("Maske").rect.width
  hM = member("Maske").rect.height
  wDarst= msp.member.rect.width
  hDarst = msp.member.rect.height
  nullregP = point(wM/2, hM/2)
   -- Mitte des Maskendarstellers
  nullLoc = point(wDarst/2,hDarst/2)
   -- Mitte des Bilddarstellers
end
```

```
on prepareframe me
  mpoint = the mouseloc - point(msp.rect.left, msp.rect.top)
  if mPoint <> myLoc then
    member("Maske").regpoint = nullregP + (nullLoc - mpoint)
    myLoc = mPoint
  end if
end
```

Das Skript setzt im `beginSprite`-Handler die Properties `nullRegP` und `nullLoc` auf die berechneten Mittelpunkte des Maskendarstellers und des Bilddarstellers im Sprite: Der Mittelpunkt eines Darstellers ist eine Punktangabe in der Form `point(halbeBreite, halbeHöhe)`.

Im `prepareFrame`-Handler berechnen wir in `mPoint` die Mausposition in Relation zur linken, oberen Ecke des Sprites. Wenn sich `mPoint` geändert hat (`if mPoint <> myLoc` then), wird der Registrierungspunkt des Maskendarstellers verschoben. `nullLoc - mPoint` ist dabei die am Mittelpunkt des Bilddarstellers gespiegelte Position der Maus in Darstellerkoordinaten (des Bilddarstellers). Um sie als neuen Registrierungspunkt für den Maskendarsteller nutzen zu können, müssen wir diesen „negativen Versatz" zum Zentrum des Maskendarstellers addieren (vgl. Abbildung 11.7).

Diese Lösung ist ebenfalls als Director-7-Datei (*Taschenlampe_Mask.dir*) auf der CD-ROM zu finden. Wird das Sprite auf der Bühne skaliert/rotiert oder anderweitig verzerrt, sollten Sie die Bestimmung von `mpoint` wie folgt ändern:

```
mpoint = mapstageToMember(the mouseloc)
```

Näheres hierzu im nächsten Experiment.

Experiment 4: Imaging Lingo

Eine elegante Lösung, die Imaging Lingo nutzt, wollen wir in der Folge vorstellen. Imaging Lingo gibt es seit der Director-Version 8. Einige wenige Funktionen waren auch schon in Director 7 verfügbar, nicht allerdings die hier verwendeten.

Diesmal bewegen wir nicht einen Überleger und auch eine separate Maske wie in Experiment 3 existiert nicht. Imaging Lingo erlaubt es uns, direkt den Alphakanal und damit die Transparenz eines Darstellers zu manipulieren. Die Bildbearbeitung, die wir in Experiment 2 für den Überleger in einem externen Programm erledigt haben, wollen wir nun direkt mit Lingo erreichen.

Imaging Lingo ist der Weg dahin. Zuvor sollten wir uns über einige Grundkonzepte klar werden.

Zunächst ist es sehr einfach, mit Imaging Lingo im Programmspeicher ein „Bild" anzulegen und dieses – z.B. mit einer einfarbigen Fläche – zu füllen:

```
bild = image(320,240,8,0,#grayscale)
bild.fill(rect(0,0,320, 240), paletteIndex(128))
neuerDarst = new(#bitmap)
neuerDarst.image = bild
```

Das Skript erzeugt im Speicher ein Bild von 320 Pixeln Breite und 240 Höhe, in 8 Bit Farbtiefe mit der Farbpalette `#grayscale`, also Graustufen. Die Funktion `fill()` füllt sodann die mit einem `rect()` angegebene Fläche: hier also ein Rechteck, das die gesamte Bildfläche ausfüllt. Die Füllfarbe muss mit einem so genannten Farbobjekt angegeben werden, also entweder als `paletteIndex(n)` oder als `rgb(Rotwert, Grünwert, Blauwert)`. Palettenindizes werden bei Bildern in 8 Bit-Farbtiefe verwendet und bezeichnen eine bestimmte Farbe aus der Palette (0-255); die `rgb()`-Farbangabe enthält die Farbanteile in Werten von 0 bis 255.

Das so bearbeitete `image`, das in der Variablen `bild` gespeichert ist, wird in den nächsten beiden Schritten einem neu generierten Darsteller zugewiesen.

In Director haben zahlreiche Darsteller-Typen eine `image`-Property, d.h. wir können auf die visuelle Repräsentation des Darstellers direkt über Lingo zugreifen. Versuchen Sie Folgendes einmal im Nachrichtenfenster, und ändern Sie in der ersten Zeile die Darsteller-Bezeichnung, um auf Bitmap-, Text-, Flash- oder Vectorshape-Darsteller zuzugreifen:

```
bild = member("einTextDarst").image
neuerDarst = new(#bitmap)
neuerDarst.image = bild
```

Die erste Zeile extrahiert das Image eines Text-Darstellers und schreibt es in die Variable `bild`. Dann wird ein neuer Darsteller vom Typ `#bitmap` angelegt, dem in der letzten Zeile das zwischengespeicherte Bild zugewiesen wird. Je nach Darsteller-Typ, den Sie umwandeln, sehen Sie im neu kreierten Darsteller Unterschiedliches; Text-Darsteller ergeben beispielsweise eine schwarze Fläche; wird diese auf die Bühne gezogen, erscheint dennoch die Schrift.

Wir können daraus schließen, dass mit Imaging Lingo auch die Transparenz-Informationen, also der so genannte Alphakanal, kopiert bzw. mit anderen Befehlen auch manipuliert werden kann. Tatsächlich werden wir – da wir nun die wichtigsten Imaging-Befehle kennen gelernt haben – für unser Experiment den Alphakanal des „Schaufenster“-Darstellers dynamisch ändern.

Die wichtigsten Schritte sind dabei folgende:

1. Wir teilen dem Darsteller mit, dass er seine Alpha-Informationen benutzen soll:

   ```
   member("Schaufenster").useAlpha = 1
   ```

2. Wir legen im Speicher wiederum ein Image (in der Größe unseres „Schaufenster“-Darstellers) an, das wir später als Alphakanal für das

„Schaufenster“-Bild benutzen wollen. Es hat 8 Bit Farbtiefe und die Graustufen-Farbpalette:

```
myalpha = image(320,240,8,0,#grayscale)
```

3 Wir nutzen die Funktion `fill()`, um ein gefülltes Oval in das `image` zu zeichnen. Das Oval wird durch das Rechteck beschrieben, in das es eingepasst wird (hier: `lamprect`), und durch die `#shapeType`-Angabe:

```
myalpha.fill(lamprect, paletteIndex(255), [#shapetype: #oval])
```

Hier sehen wir auch bereits den Ansatzpunkt, wie wir die Animation des Lampenkegels bewerkstelligen können: Wenn wir das `lamprect` jeweils in Abhängigkeit von der Mausposition neu berechnen, können wir das Oval dorthin setzen, wo die Maus gerade über dem Bild steht.

4 Wir weisen dem Image des „Schaufenster“-Darstellers mit der Funktion `setAlpha()` den so erstellten Alpha-Kanal zu:

```
member("Schaufenster").image.setalpha(myalpha)
```

Die ersten beiden Schritte müssen wir nur einmal machen; sie werden sinnvollerweise also im `beginSprite`-Handler unseres Behaviors erledigt. Schritt 3 und 4 dagegen soll bei jeder Mausbewegung ausgeführt werden. Wir nutzen – wie schon in den vorangehenden Experimenten – den `prepareFrame`-Handler.

Die erste Version unseres Skriptes sieht wie folgt aus. Die Höhe und Breite des Lampenovals ist über Properties konfigurierbar:

```
property myalpha
property breite, hoehe  -- Breite und Höhe des Ovals

on beginsprite me
  msp = sprite(me.spritenum)
  msp.member.useAlpha  =  1
  myalpha = image(320,240,8,0,#grayscale)
  hoehe = 70
  breite = 90
end

on prepareFrame me
  mymousePos = the mouseLoc
  if oldmousePos <> mymousePos then
    lamprect = rect(mymousepos - point(breite/2,hoehe/2), \
     mymousepos + point(breite/2,hoehe/2))
    myalpha.fill(lamprect, paletteIndex(255), \
     [#shapetype: #oval])
    msp.member.image.setalpha(myalpha)
    oldmousePos = mymousePos
  end if
end
```

Bitte probieren Sie diesen Zwischenschritt auch aus. Wir haben nämlich – durch ein Versäumnis im Skript – eine andere interessante Anwendung geschaffen: Mit diesem Skript können wir ein Bild mit der Maus nach und nach *freilegen*. Die Maus hinterlässt sozusagen eine Lichtspur. Das ist nicht das, was wir wollten – aber auch eine sinnvolle Anwendung (vgl. Abbildung 11.8, Film auf CD: *Taschenlampe_Rubbel.dir*).

Abbildung 11.8: ***Rubbelbild: Nicht das erwünschte Ergebnis, aber auch interessant***

Damit wirklich nur der Lichtkegel auf dem Bild herumwandert, müssen wir, bevor wir das Oval zeichnen, erst einmal den Alpha-Kanal wieder zurücksetzen:

```
(...)
if oldmousePos <> mymousePos then
  myalpha.fill(rect(0,0,320, 240), paletteIndex(0))
(...)
```

`paletteIndex(0)` ist weiß. Damit wird das vorangehende schwarze Oval gelöscht, bevor es an der neuen Position neu gezeichnet wird.

▶ Modifikation:

Eine weitere kleine Modifikation erlaubt es, dass das „Schaufenster"-Sprite an jeder Position auf dem Bildschirm platziert werden kann. Statt nämlich einfach `the mouseloc` in die Variable `mymousePos` zu tun, korrigieren wir die Punktangabe um die Verschiebung des Sprites:

```
on prepareFrame me
  mymousePos = the mouseLoc - point(msp.rect.left, msp.rect.top)
```

Die Punktangabe `point(msp.rect.left, msp.rect.top)` beschreibt den oberen linken Punkt des „Schaufenster"-Sprites. Sie muss vom `mouseloc` abgezogen werden, um die Bühnen- in Darsteller-Positionsangaben zu transformieren.

Eigentlich gibt es eine eigene Lingo-Funktion für diesen Zweck: `mapStageToMember()`. Diese hat den Vorteil, dass sie auch mit auf der Bühne verzerrten (skalierten, rotierten, gequaddeten...) Sprites umgehen kann.

Der Nachteil ist, dass sie für Mauspositionen außerhalb des Sprites den Rückgabewert `VOID` liefert, also undefiniert ist. Ihr Einsatz wäre so möglich:

```
on prepareFrame me
  mymousepos = msp.mapstageToMember(the mouseloc)
  if oldmousePos <> mymousePos AND not (voidP(mymousePos)) then
```

Die Imaging-Operationen werden dann nur durchgeführt, wenn sich in `mymousePos` ein gültiger Wert befindet (`not (voidP (mymousePos))`). Sobald die Maus das Sprite verlässt, bleibt die Taschenlampe auf ihrer letzten Position. Diese Funktion wird im Beispielfilm *Taschenlampe_Image.dir* nicht genutzt, da das „Schaufenster"-Sprite nicht verzerrt ist und es uns wichtig war, dass der Lichtkegel das Bild auch verlassen kann.

▶ **Modifikation:**

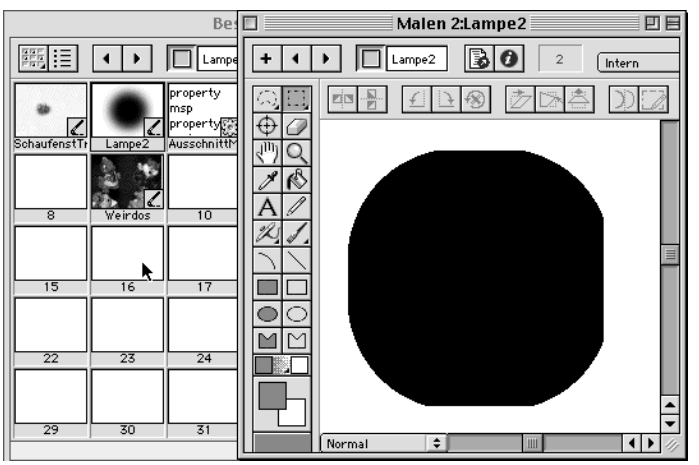


Abbildung 11.9: Lampe2.png als Darsteller importiert: Im Malfenster ist der Alphakanal nicht sichtbar, in der Vorschau in der Besetzung (und auf der Bühne) schon.

Eine andere Modifikation ist allerdings sehr sinnvoll. Hierfür sollten wir uns mit einer weiteren Imaging-Lingo-Funktion beschäftigen, der Funktion `copyPixels()`. Mit ihr können wir ein Bild in ein anderes kopieren – ideal, um einen weich gezeichneten, teiltransparenten Kreis (vgl. Abbildung 11.9; der Darsteller wurde in Fireworks erstellt und befindet sich als *Lampe2.PNG* auf der CD-ROM) aus einem Darsteller zu extrahieren und ihn dann mit `copyPixels()` in unser Alpha-`image` zu transferieren. Die Syntax ist wie folgt:

```
zielbild.copypixels(zuKopierendesBild, zielRect, AusgangsRect,
[Optionen])
```

In unserem Fall ergänzen wir eine Property `maskimage` am Beginn des Skripts, definieren im `beginSprite`-Handler diese Variable, indem wir eine Kopie des `image` des neuen Darstellers „`lampe2`“ einlesen, und ersetzen den bisherigen `fill()`-Befehl durch den `copyPixels()`-Aufruf. Die entsprechenden Stellen sind im Skript gefettet:

```
property myalpha
property breite, hoehe  -- Breite und Höhe des Ovals
property maskimage

on beginsprite me
  msp = sprite(me.spritenum)
  msp.member.useAlpha  =  1
  myalpha = image(320,240,8,0,#grayscale)
  hoehe = 70
  breite = 90
  maskimage = member("lampe2").image.duplicate()
end

on prepareFrame me
  mymousePos = the mouseLoc - point(msp.rect.left, msp.rect.top)
  if oldmousePos <> mymousePos then
    myalpha.fill(rect(0,0,320, 240), paletteIndex(0))
    lamprect = rect(mymousepos - point(breite/2,hoehe/2), \
     mymousepos + point(breite/2,hoehe/2))
    myalpha.copypixels(maskimage,  lamprect,  maskimage.rect, \
     [#dither: true])
    msp.member.image.setalpha(myalpha)
    oldmousePos = mymousePos
  end if
end
```

Das Duplizieren des Images von „Lampe2“ mit `image.duplicate()` bringt Performance-Vorteile, da wir dann immer auf eine unabhängige Kopie des Images im Arbeitsspeicher zugreifen können, nicht auf eine Referenz auf ein Darsteller-Image.

Diesen Film (*Taschenlampe_copy.dir*) finden Sie wiederum auf CD-ROM.

▶ **Variation – Kerzenlicht:**

Der unruhige Schein eines Kerzenlichts lässt sich ausgehend vom letzten Skript ebenfalls mit wenigen Modifikationen erreichen:

- Wir machen die Form des „Lichts“ variabel (mit einer Zufallsfunktion und der Sprite-Eigenschaft `quad`).
- Und wir berechnen bei jedem Framedurchlauf einen neuen Zufallswert für die Eigenschaft `myrandlight`, die sowohl für die Dunkelheit (entsprechend „nach unten“ transformiert) als auch für den Lichtkegel benutzt wird.

Das entsprechend modifizierte Skript sieht so aus:

```
property msp
property myimg, myalpha
property mymousepos
property breite, hoehe, maskimage

on beginsprite me
  msp = sprite(me.spritenum)
  msp.member.useAlpha = 1
  myalpha = image(320,240,8,0,#grayscale)
  hoehe = 90
  breite = 90
  maskimage = member("lampe2").image.duplicate()
end

on prepareFrame me
  mymousepos = the mouseloc
  myrand = random(150,200)/150.0   -- random für Quad
  myrandlight = random(128,255)    -- und lichtstärke
  lampquad = \
   [mymousepos + point(-breite/myrand, -hoehe*myrand), \
    mymousepos + point(breite/myrand,-hoehe*myrand), \
    mymousepos + point(breite/myrand,hoehe)/2, \
    mymousepos + point(-breite/myrand,hoehe)/2]
  myalpha.fill(rect(0,0,320, 240), \
    paletteIndex((myrandlight - 128) / 4))
  myalpha.copypixels(maskimage, lampquad, maskimage.rect, \
    [#dither: true, #blendlevel:myrandLight])
  msp.member.image.setalpha(myalpha)
end
```

Das Skript macht sich zunutze, dass die Funktion `copyPixels()` nicht nur mit einem Ziel-`rect`, sondern auch mit einem Ziel-`quad` genutzt werden kann. Das heißt, das aus dem Darsteller „lampe2“ duplizierte Maskenimage wird aus seiner rechteckigen Form (`maskimage.rect`) beim Kopieren mit `copyPixels()` in eine verzerrte Quadform gefüllt; das Quad wird zuvor mit dem Zufallswert `myrand` für jeden Framedurchlauf neu berechnet: Es wird also wirklich zittern. Auch für die Lichtintensität (Parameter `#blendLevel` im `copyPixels()`-Aufruf für die Opazität beim Einkopieren) benutzen wir eine Zufallszahl, `myrandLight`. Entsprechend wird die Funktion `fill()` bei jedem Framedurchlauf mit dem Wert `paletteIndex((myrandlight - 128) / 4))` aufgerufen, die Opazität des umgebenden Bereichs liegt also zwischen 0 und 31.

Die Zufallszahlen ermitteln wir mit der Funktion `random()`, die in zwei Varianten in Director existiert: `random (wert1, wert2)` liefert eine Zufallszahl, die größer oder gleich `wert1` und kleiner oder gleich `wert2` ist, `random(wert)` dagegen eine Zahl zwischen 0 und `wert-1`.

```
myrand = random(150, 200)/150.0
```

wird demnach 51 unterschiedliche Fließkommazahlen zwischen 1 und 1.3333 liefern können, die wir dann als Divisor oder Multiplikationsfaktor bei der Berechnung des „Lichtquads“ benutzen.

Den Film finden Sie als *Taschenlampe_Kerze* auf der CD-ROM.

- **Variation – Negativ in Positiv:**

Eine effektvolle Variation lässt sich mit allen Skripten dieses Experiments kombinieren: Legen Sie hinter das Schaufenster-Sprite eine Kopie des Darstellers, die im Malfenster in ein Negativ umgewandelt wurde (vgl. Abbildung 11.10). Besonders interessant ist es, wenn Sie die Variante „Rubbelbild“ (s.o.) mit diesem Effekt kombinieren: Dann wird das Negativ weggerubbelt und das Positiv erscheint mit Ihren Mausbewegungen.

Abbildung 11.10: Erstellen eines Negativs im Malfenster

Experiment 5: Lichtquelle im 3D-Darsteller bewegen

Während wir bisher nur über Umwege zum Ziel kamen – Überleger, Masken, Alphakanäle –, verspricht der Ansatz, in einem Director-3D-Darsteller eine Lichtquelle vor einer Wand zu bewegen, endlich eine „realistische“ Möglichkeit zu werden, den Taschenlampen-Effekt in Director umzusetzen. 3D-Fähigkeiten hat Director seit Version 8.5; mit älteren Versionen können Sie dieses Experiment daher nicht nachvollziehen.

▶ Hilfsmittel 3DPI:

Um zu einem vertieften Verständnis zu kommen, wie Director-3D funktioniert und wie ein Shockwave3D-Darsteller aufgebaut ist, bedienen wir uns eines genialen Sharewaretools der österreichischen Director-Entwicklerin Ullala: des 3DPI. Der 3D-Property-Inspektor erlaubt es, 3D-Darsteller und ihre zahlreichen Elemente (Modelresources, Models, Texturen, Shader, Lichter, Kameras etc.) anzulegen und direkt einzustellen, sodass sich eine kleine Szene wie unser Taschenlampen-Beispiel konstruieren lässt, ohne dass wir uns zunächst um die Lingo-Umsetzung kümmern müssen.

Abbildung 11.11 zeigt die 3D-Szene auf der Bühne, die Lichteinstellungen des 3DPI und vom 3DPI generierten Lingo-Code (nämlich die Lichteigenschaften). Das ist der Endzustand, den wir mit dem 3DPI erreichen wollen; über die „Trace"-Funktion des Tools (das ist der kleine Button oben rechts im 3DPI) können wir uns die jeweiligen Einstellungen ins Nachrichtenfenster ausgeben lassen.

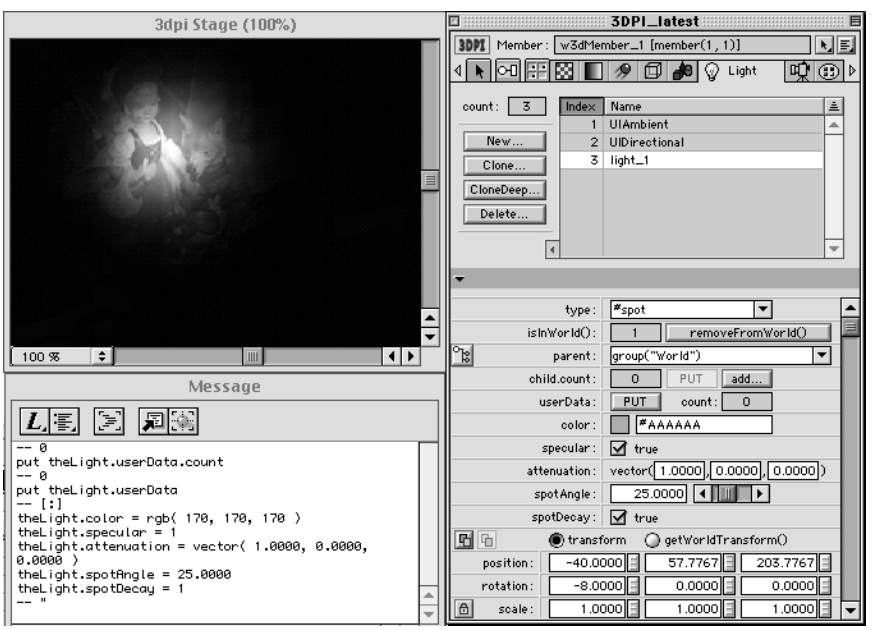


Abbildung 11.11: So soll es aussehen: Konstruktion einer 3D-Szenerie mit dem 3DPI (Shockwave3D-Propertyinspektor) von Ullala.

Untypischerweise lassen sich die beim Authoring in einem Shockwave3D-Darsteller gemachten Änderungen nämlich *nicht* direkt sichern. Das heißt, unsere Arbeit mit dem 3DPI ist überaus vergänglich – sobald wir den Film schließen, sind die Einstellungen hinüber. Aber der 3DPI liefert uns die Daten, mit denen wir im zweiten Schritt des Experiments ein Skript erstellen werden, um den Darsteller mit Lingo so einzurichten, wie wir es jetzt mit dem 3DPI versuchen werden.

Die jeweils neueste Version des 3DPI ist online unter *http://www.3dpi-director.com/* erhältlich und kann natürlich erst ab Director 8.5 eingesetzt werden. Legen Sie die Datei *3dpi.dcr* in das *Xtras*-Verzeichnis im Programm-Verzeichnis von Director 8.5. Nach einem Neustart des Programms steht der Eintrag 3DPI im Menü XTRAS in Director zur Verfügung.

▶ Erste Schritte mit dem 3DPI:

Legen Sie einen neuen Film (320x240 Pixel) in Director 8.5 an und importieren Sie den Bitmap-Darsteller „Schaufenster“ aus *Taschenlampe.dir*. Öffnen Sie den 3DPI über das Menü XTRAS.

- Wählen Sie den Darsteller-Tab des Inspektors und legen Sie einen neuen Shockwave3D-Darsteller an (vgl. Abbildung 11.12; alternativ lässt sich das natürlich auch über das Menü EINFÜGEN/MEDIA-ELEMENT/SHOCKWAVE3D erledigen). Platzieren Sie den Darsteller auch gleich auf der Bühne, damit Sie die weiteren Schritte im Bühnenfenster kontrollieren können.

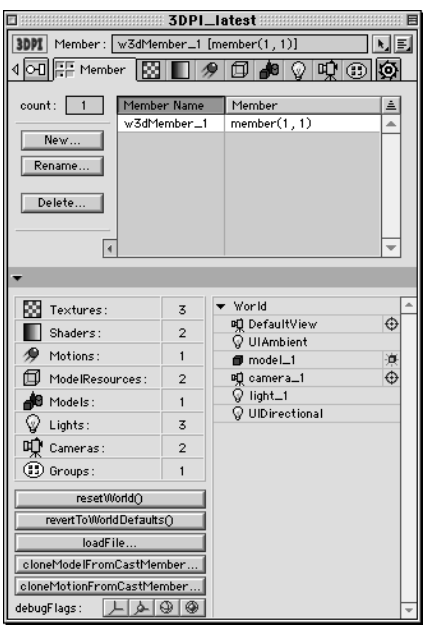
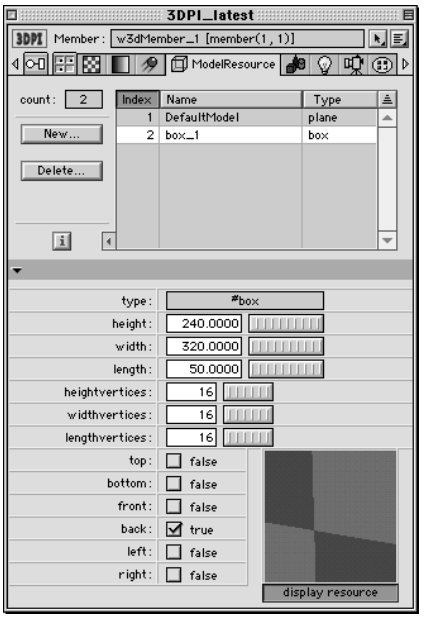


Abbildung 11.12: Neuen Shockwave3D-Darsteller erzeugen

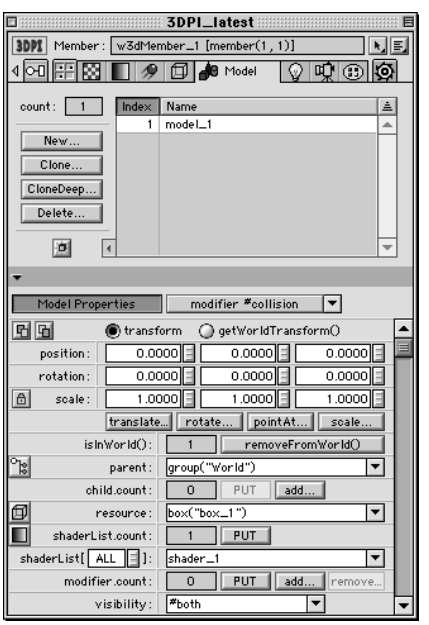
- Legen Sie eine Modelresource an, indem Sie im Tab „Modelresource“ den Button „New...“ anklicken. Wir haben uns für den Typ „Box“ entschieden und die Höhe und Breite unserer Bitmap „Schaufenster“ entsprechend gewählt (vgl. Abbildung 11.13).

Abbildung 11.13: Neue Modelresource erzeugen



- Legen Sie ein neues Model an, indem Sie im Tab „Model“ den Button „New...“ anklicken. Als Modelresource für das Model geben Sie die gerade generierte Box an (vgl. Abbildung 11.14). Sie sehen das Model bereits im Darsteller auf der Bühne: Natürlich nutzt er noch nicht die Bitmap „Schaufenster“, sondern erscheint mit einem Standard-Karo-Muster.

Abbildung 11.14: Neues Model erzeugen



- Legen Sie eine neue Textur an, indem Sie im Tab „Texture“ den Button „New...“ anklicken. Wählen Sie unter „type“ die Einstellung „#fromCastMember“ und geben Sie den Darsteller „Schaufenster“ unter „member“ an (vgl. Abbildung 11.15).

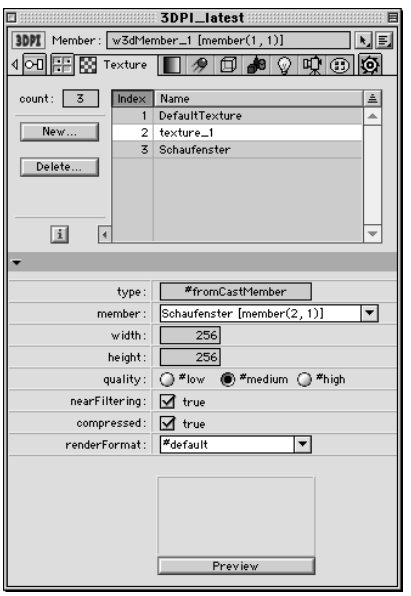


Abbildung 11.15:
Neue Textur erzeugen

- Legen Sie einen neuen Shader an, indem Sie im Tab „Shader“ den Button „New...“ anklicken. Wählen Sie unter „textureList“ die gerade erstellte Textur „texture_1“ für den Index 1 (vgl. Abbildung 11.16).

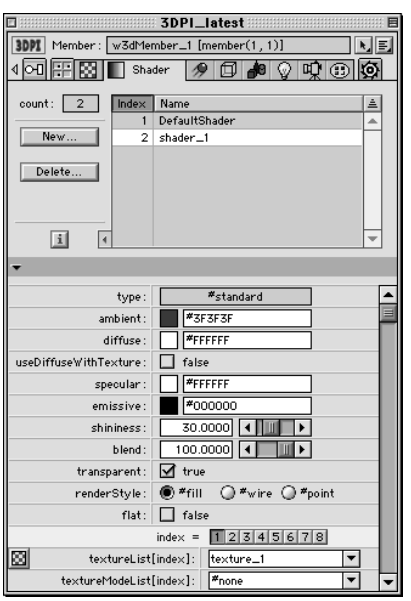


Abbildung 11.16:
Neuen Shader erzeugen

- Weisen Sie den Shader dem Model zu (Eintrag „shaderList“ im Model-Tab, vgl. Abbildung 11.14)
  Auf der Bühne können Sie nun das „Schaufenster“-Bild, das auf den Quader gemappt wurde, erkennen. Noch sind der Bildausschnitt und die Beleuchtung nicht richtig; das werden wir in den nächsten Schritten einstellen. Im Tab Modelresource können Sie schon einmal die nicht benötigten „Wände“ unseres Quaders ausblenden (Einstellungen vgl. Abbildung 11.13).
- Legen Sie eine neue Kamera an, indem Sie im Tab „Camera“ den Button „New...“ anklicken. Weisen Sie im Sprite-Tab (vgl. Abbildung 11.17) dem Sprite die neu generierte Kamera zu – dann sehen Sie die weiteren Änderungen direkt im 3D-Sprite auf der Bühne.

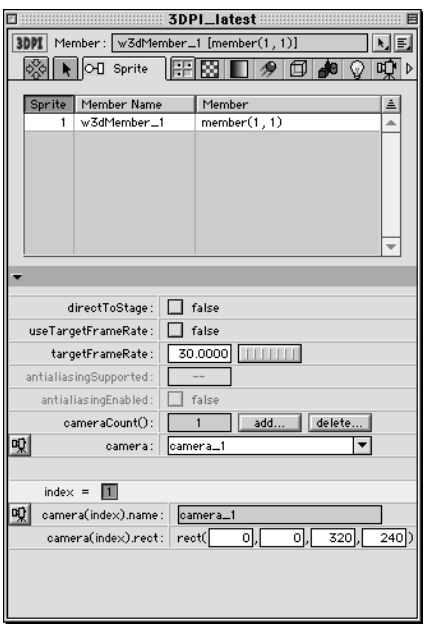


Abbildung 11.17: Kamera dem Sprite zuweisen

- Ändern Sie in der Kamera-Einstellung unter „position“ den dritten, also z-Wert so, dass der Bildausschnitt möglichst genau das „Schaufenster“-Bild anzeigt (vgl. Abbildung 11.18).

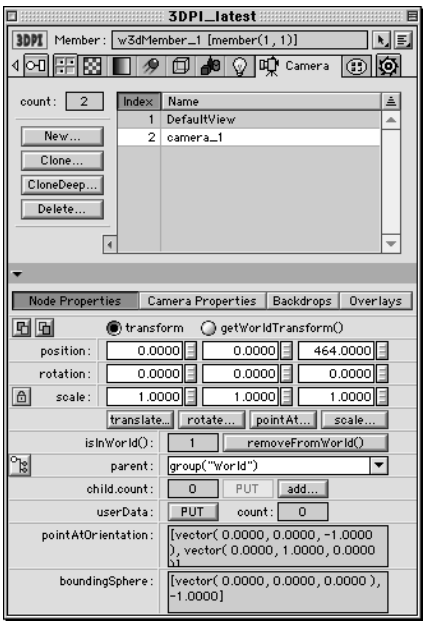


Abbildung 11.18:
Neue Kamera erzeugen

- Legen Sie eine neues Licht an, indem Sie im Tab „Light“ den Button „New...“ anklicken. Wählen Sie unter „type“ den Typ „#spot“, und ändern Sie die Werte für „spotAngle“, „position“ und „scale“, bis Sie einen auf das Bild fokussierten Spot haben (vgl. Abbildung 11.19).

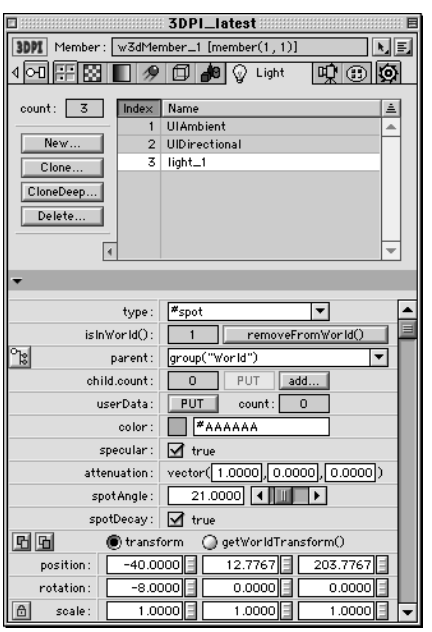


Abbildung 11.19:
Neues Licht erzeugen

- Die Lichtfarbe des Standard-Lichtes „UIAmbient“ haben wir auf ein dunkles Blau (#000044) gesetzt, damit der Spot besser zur Geltung kommt. Wir haben außerdem die Eigenschaft „spotDecay“ unseres Lichtes „light_1“ eingeschaltet (was ein weiches Auslaufen der Lichtränder bewirkt). Und selbst in der Definition der Modelresource (vgl. Abbildung 11.13) können wir noch etwas für unser Licht tun: Wir stellen die Anzahl der Vertices (also der Unterteilungen, die beim Rendern der Lichteffekte auf der Textur genutzt werden) von 2 auf 16.

Selbstverständlich ist das Ziel der Arbeit mit dem 3DPI aber nicht, genau das Ergebnis zu erreichen, das Abbildung 11.11 zeigt. Das Werkzeug bietet unendlich viele Möglichkeiten, mit dem Shockwave3D-Darsteller zu „spielen“: Probieren Sie es aus!

▶ **Lingo-Umsetzung:**

Wir rekapitulieren noch einmal: Shockwave3D-Darsteller, die in Director erzeugt werden, sind bis auf einige Standard-Elemente leer. Wenn wir Änderungen mit dem 3DPI vornehmen, sehen wir diese zwar im Darsteller, sie werden beim Sichern des Films aber nicht mitgesichert. Was ist also zu tun?

Wir müssen nun – durchaus mit Hilfe des 3DPI, dessen Trace-Funktion uns die benötigten Informationen liefert – ein Initialisierungsskript für unser Shockwave3D-Sprite schreiben, um bei Aufruf des Sprites auf der Bühne dessen Darsteller wieder so einzurichten, wie wir es gerade mit dem 3DPI getan haben. Wir folgen dabei den Schritten, die wir auch im 3DPI vornehmen mussten.

Außerdem ist unsere Lichtquelle natürlich noch statisch, und wir wollen sie an die Mausbewegung des Anwenders koppeln. Im 3DPI können wir diese Bewegung simulieren (Eigenschaft „position“ im Licht-Tab, wo Sie die x-, y- und z-Werte des Positionsvektors verändern können); damit werden wir auch ein entsprechendes Skript für die User-Interaktion generieren können.

Wir gehen wieder vom „Grundgerüst für Shockwave3D-Behaviors“ aus, das wir im vorangehenden Kapitel vorgestellt haben. Die meisten Daten konnten wir aus dem Trace des 3DPI direkt ins Skript einfügen. Diesen Trace (d.h. die Ausgabe des 3DPI im Nachrichtenfenster) finden Sie als Textdatei auf der CD-ROM, ebenso wie den Film *Taschenlampe3D.dir*, der den Endzustand dieses Experiments dokumentiert.

```
property msp, mmb, mylight

on beginsprite me
  msp = sprite(me.spritenum)
  mmb = msp.member
  -- -------------------------
  -- 1. Darsteller vorbereiten
  -- -------------------------
  mmb.resetworld()
  mmb.bgcolor = rgb(0,0,0)
  mmb.ambientcolor = rgb(0,0,0)
```

Die Einstellungen des Standard-Lichtes haben wir dem 3DPI-Trace entnommen:

```
  uiLight = mmb.light("UIDirectional")
  uiLight.transform.rotation = \
   vector(-54.7356, -30.0000, 35.2644)
  uiLight.color = rgb( 0, 0, 68 )
  uiLight.specular = 1
  -- -------------------------
  -- 2. Modelresource anlegen
  -- -------------------------
  if voidP(mmb.modelresource("box_1")) then
```

Die Einstellungen der Modelresource haben wir ebenfalls dem 3DPI-Trace entnommen:

```
    Res = mmb.newModelResource("box_1", #box, #back)
    Res.height = 240.0000
    Res.width = 320.0000
    Res.length = 50.0000
    Res.heightVertices = 16
    Res.widthVertices = 16
    Res.lengthVertices = 16
    Res.top = 0
    Res.bottom = 0
    Res.front = 0
    Res.back = 1
    Res.left = 0
    Res.right = 0
  else
    Res = mmb.modelresource("box_1")
  end if
  -- -------------------------
  -- 3. Model anlegen
  -- -------------------------
  if voidP(mmb.model("model_1")) then
    mymod = mmb.newModel("model_1", Res)
```

Die folgende Einstellung ist wichtig, sonst sehen wir gar nichts:

```
  mymod.visibility = #back
else
  mymod = mmb.model("model_1")
end if
-- --------------------------
-- 4./5 Shader und Texturen anlegen und zuweisen
-- --------------------------
if voidP(mmb.shader("shader_1")) then
```

Aus dem 3DPI-Trace:

```
  myShad = mmb.newShader("shader_1",#standard)
  myShad.ambient = rgb( 63, 63, 63 )
  myShad.diffuse = rgb( 255, 255, 255 )
  myShad.useDiffuseWithTexture = 0
  myShad.specular = rgb( 255, 255, 255 )
  myShad.emissive = rgb( 0, 0, 0 )
  myShad.shininess = 30.0000
  myShad.blend = 100.0000
  myShad.transparent = 1
  myShad.renderStyle = #fill
  myShad.flat = 0
else
  myShad = mmb.shader("shader_1")
end if
if voidP(mmb.texture("texture_1")) then
  mytextu = mmb.newTexture("texture_1", #fromCastMember, \
   member("Schaufenster"))
else
  mytextu = mmb.texture("texture_1")
end if
myshad.texture = mytextu
myshad.textureMode = #wrapPlanar
-- Shader zuweisen
mymod.shaderList = [myshad]
-- --------------------------
-- 6. Light anlegen
-- --------------------------
if voidp(mmb.light("light_1")) then
```

Aus dem 3DPI-Trace:

```
  mylight = mmb.newlight("light_1", #spot)
  mylight.transform.position = \
   vector(-40.0000, 57.7767, 203.7767)
  mylight.transform.rotation = \
   vector(-8.0000, 0.0000, 0.0000)
  mylight.transform.scale = vector(1.0000, 1.0000, 1.0000)
  mylight.color = rgb( 170, 170, 170 )
  mylight.specular = 1
  mylight.attenuation = vector(1.0000, 0.0000, 0.0000)
  mylight.spotAngle = 25.0000
```

```
    mylight.spotDecay = 1
  else
    mylight = mmb.light("light_1")
  end if
  -- -------------------------
  -- 7. Kamera einstellen und zuweisen
  -- -------------------------
  if voidP(mmb.camera("camera_1")) then
    mycam = mmb.newcamera("camera_1")
  else
    mycam = mmb.camera("camera_1")
  end if
```

Aus dem 3DPI-Trace:

```
  mycam.transform.position = vector(0.0000, 0.0000, 464.0000)
  mycam.transform.rotation = vector(0.0000, 0.0000, 0.0000)
  mycam.transform.scale = vector(1.0000, 1.0000, 1.0000)
  mycam.fieldofView = 30.0
  mycam.pointAtOrientation = [vector(0.0000, 0.0000, -1.0000),\
   vector(0.0000, 1.0000, 0.0000)]
  -- Kamera dem Sprite zuweisen; erst dann sehen wir die
  -- Szene durch die Kamera!
  msp.camera = mycam
end

on prepareFrame me
```

Maus-Position nutzen, um die Transform-Informationen des Lichts mylight *zu verändern:*

```
  mymouse = (the mouseLoc - point(160,120)) * point(1, -1)
  mylight.transform.position = \
   vector(mymouse[1], mymouse[2], mylight.transform.position[3])
end
```

- **Mausposition umrechnen:**

Am letzten Skriptabschnitt sehen Sie, wie einfach es ist, die gewünschte Interaktivität in das Skript einzufügen, wenn der Shockwave3D-Darsteller erst einmal initialisiert ist. Die Interaktivität besteht nämlich lediglich darin, die Mausposition zu nutzen, um den Transformationsvektor des Lichts `mylight`, das wir oben als Property definiert haben, zu manipulieren.

Um die Zeile

```
mymouse = (the mouseLoc - point(160,120)) * point(1, -1)
```

zu verstehen, schreiben Sie bitte zunächst einmal

```
mymouse = the mouseLoc
```

ins Skript und probieren dies aus. Der Lichtkegel erscheint verschoben.

Offensichtlich ist das Koordinatensystem des Darstellers eines, das vom Mittelpunkt der dargestellten Szene ausgeht; da der Darsteller 320 x 240 Pixel groß ist, ziehen wir also `point(160,120)` vom `mouseLoc` ab:

```
mymouse = the mouseLoc - point(160,120)
```

Nun sehen wir, dass die x-Koordinate schon richtig dargestellt wird (das Licht folgt in der Waagrechten unserer Maus), die y-Ausrichtung aber offensichtlich an der Mitte gespiegelt ist. Das können wir einfach korrigieren, indem wir die Mausposition ebenfalls an y spiegeln – durch Multiplikation mit `point(1, -1)`:

```
mymouse = (the mouseLoc - point(160,120)) * point(1, -1)
```

Während `mymouse` nun eine entsprechend manipulierte Punktangabe enthält, benötigen wir im 3D-Raum einen Vektor, den wir der Eigenschaft `mylight.transform.position` zuweisen können. X- und y-Werte des Vektors lesen wir mit Listensyntax aus Position 1 und 2 von `mymouse`, den z-Wert lassen wir unverändert, d.h. wir lesen ihn von der dritten Position der `mylight.transform.position`. Damit können wir unsere Skriptvorgaben für die Interaktivität wie folgt zusammenfassen. `-- (…)` bezeichnet Auslassungen im Vergleich zum oben stehenden Skript:

```
property mylight -- (…)
on beginSprite me
  --(...)
  mylight = mmb.newlight("light_1", #spot)
  --(...)
end

on prepareFrame me
  mymouse = (the mouseLoc - point(160,120)) * point(1, -1)
  mylight.transform.position = vector( mymouse[1], mymouse[2], \
   mylight.transform.position[3])
end
```

11.3 Diskussion

Die Frage, was denn nun der beste Weg ist, die „Taschenlampe“ (und ähnliche Aufgabenstellungen) in Director umzusetzen, lässt sich nicht allgemeingültig beantworten. Mehrere Aspekte kommen ins Spiel:

- die Director-Version,
- der Lingo-Aufwand, auch der Aufwand, sich in neue Bereiche von Lingo (wie Imaging und 3D) einzuarbeiten,
- die Abspielplattform, die erreicht werden soll (da die Lösungen unterschiedliche Anforderungen stellen, was Rechner-Performance oder Shockwave-Version angeht).

Mit Director 7 haben Sie keine Möglichkeit, die in den Experimenten 4 und 5 vorgestellten Techniken anzuwenden. Sie sind folglich auf Überleger oder Maskierung (Experimente 1-3) angewiesen. Überleger sind am einfachsten zu realisieren, haben aber Einschränkungen, wenn Sie die Technik mit weiteren Elementen auf der Bühne kombinieren wollen. Hier heißt es, in höheren Spritekanälen wiederum Sprites zu platzieren, die den Überleger verdecken, wo er nicht zu sehen sein soll. Auch die Performance-Anforderungen sind höher als bei der Maskierungstechnik.

Der Imaging-Lingo-Ansatz erfordert die Director-Version 8 und ist, was das benutzte Lingo angeht, nicht gerade trivial. Von den vorgestellten Techniken gehört diese (zusammen mit der Maskierungstechnik) zu denjenigen, die die geringsten Anforderungen an die Performance der Abspielplattform stellen.

3D-Lingo bringt das „realistischste“ Ergebnis, es ist aber ein bisschen so, wie mit Kanonen auf Spatzen zu schießen. Die Performance-Anforderungen sind hoch, der Lingo-Aufwand ebenso – aber wenn Sie Spaß am Tüfteln haben, ist das ein einfacher Einstieg in ein spannendes Arbeitsgebiet.

Wer ist eigentlich „me“?

Immer wieder, in fast jedem Skript, benutzen wir das Wörtchen `me`. Einem Lingo Laboranten sollte der nicht hinterfragte Gebrauch eines Lingo-Begriffes ein Dorn im Auge sein, und so wollen wir zum Abschluss empirisch ergründen, wer – oder was – dieses geheimnisvolle *ICH* (oder *mich*?) eigentlich ist.

Das ist eine philosophische Frage? Ja, da könnten Sie Recht haben...

Das Lingo-Handbuch gibt sich zwar nicht gerade wortkarg, aber so richtig verständlich ist nicht, was wir da über `me` lesen können:

> *Diese spezielle Variable wird in Parent-Skripts und Verhalten verwendet, um auf das aktuelle Objekt zu verweisen, das eine Instanz des Parent-Skripts bzw. des Verhaltens oder eine Variable ist, die die Speicheradresse des Objekts enthält.*
>
> *Der Begriff selbst hat in Lingo keine vordefinierte Bedeutung. Der Begriff* `me` *wird wie allgemein üblich verwendet.*

Angesichts dieser Erklärung haben wir zwei Möglichkeiten: einen theoretischen Einstieg in die objektorientierte Programmierung oder unseren bewährten Laboransatz. Wir entschließen uns für letzteren und legen mit einigen „Proben“ los.

12.1 Experiment 1: Proben

Die folgenden Beispiele finden Sie im Film *me.dir* auf der CD-ROM.

Die Weglassprobe

In den Skripten in diesem Buch haben wir `me` häufig wie hier verwendet:

```
property myname

on beginSprite me
  myname = "hans"
end

on mouseup me
  put myname
  changecolor me, random(0,255)
end

on changecolor me, newcolor
  sprite(me.spritenum).color = paletteIndex(newcolor)
end
```

Was tut dieses Skript? Es definiert eine Property `myname`, und bei `mouseUp` gibt es die Property ins Nachrichtenfenster aus und ruft einen weiteren, selbst definierten Handler auf, der die Farbe des Sprites auf einen zufälligen Farbwert setzen soll. So weit, so gut. Was passiert, wenn wir das `me` weglassen?

```
property myname

on beginSprite
  myname = "hans"
end

on mouseup
  put myname
  changecolor random(0,255)
end

on changecolor newcolor
  sprite(spriteNum).color = paletteIndex(newcolor)
end
```

Alle Erscheinungen von `me` wurden getilgt – und schon beim Kompilieren erhalten wir eine Fehlermeldung (vgl. Abbildung 12.1).

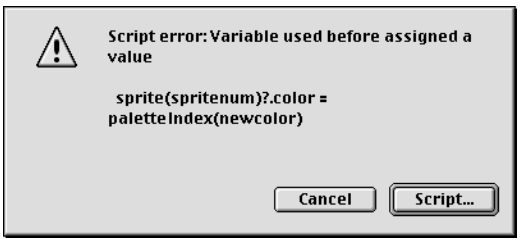


Abbildung 12.1: Director weist darauf hin, dass die Variable spritenum undefiniert ist.

`spriteNum` ist undefiniert – kein Wunder, wir haben sie nicht als Property deklariert, auch nicht als globale Variable. Tatsächlich reicht es aus, eine entsprechende Property-Deklaration ins Skript einzufügen, und unser Behavior funktioniert ohne jedes `me`:

```
property spriteNum
property myname
-- (…)
```

Die Umbenennungsprobe

Was passiert, wenn wir einfach ein anderes Wort an die Stelle von `me` schreiben? Zum Beispiel `ich` – wie es der Handbuchauszug suggeriert?

Das folgende Skript funktioniert wie unser obiges Beispiel mit `me`:

```
property myname

on beginSprite ich
  myname = "hans"
end

on mouseup ich
  put myname
  changecolor ich, random(0,255)
end

on changecolor ich, newcolor
  sprite(ich.spriteNum).color = paletteIndex(newcolor)
end
```

Erstaunliches tritt zutage: `ich` – obwohl bisher nirgends definiert – hat plötzlich eine Eigenschaft `spriteNum`, auf die wir mit Dot-Syntax (`ich.spriteNum`) zugreifen können. Offenbar sollten wir uns doch den Inhalt von `me` (oder `ich`) einmal genauer anschauen.

Die Inhaltsprobe

Das folgende Behavior haben wir „inhalt“ benannt und auf Sprite 1 gezogen.

```
on beginsprite me
  myname = "hans"
  put me
end

on mouseup ich
  put ich
end
```

Die Ausgabe im Nachrichtenfenster, sobald wir den Film starten und einmal auf das Sprite klicken, ist folgende:

```
-- <offspring "inhalt" 4 1a01a260>
-- <offspring "inhalt" 4 1a01a260>
```

Halten wir den Film an und starten wir ihn erneut, so ist die Zahlen-/Buchstabenkombination rechts eine andere. Die Ausgaben für das `me` und das `ich` sind aber jeweils identisch. Offenbar ist nur entscheidend, dass ein erster Parameter im Event-Handler abgefragt wird, aber nicht, wie wir diesen Parameter nennen.

Das `me` enthält offensichtlich eine Information darüber, von wem es abstammt: von dem Behavior „inhalt“ in diesem Fall. Die wilde Buchstaben-/Zahlenkombination am Ende ist eine Speicheradresse, deren Inhalt für uns nicht weiter interessant ist. Wichtig zu sehen ist allerdings, dass die Speicheradresse bei jedem Start des Films (bzw. bei jedem `beginSprite`) eine andere ist: Das Sprite und das Behavior werden neu initialisiert, d.h. sie erhalten neue „Plätze“ im Speicherraum von Director.

Wir erweitern unser kleines Skript um eine Property und weitere `put`-Befehle:

```
property myname

on beginsprite me
  put me
  myname = "hans"
end

on mouseup me
  put "--------"
  put me.count
  put me[1]
  put me[2]
  put me.myname
  put me.spriteNum
  put me[#myname]
  put me[#spritenum]
end
```

Im Nachrichtenfenster erscheint beim ersten Klick auf das Sprite Folgendes:

```
-- <offspring "inhalt2" 4 1a019d24>
-- "---------"
-- 2
-- "hans"
-- 4
-- "hans"
-- 4
-- "hans"
-- 4
```

Offensichtlich lässt sich auf `me` wie auf eine Liste – sogar eine Propertyliste – zugreifen: Die Eigenschaft `count` liefert die Anzahl der Listenelemente, mit `[n]` können wir auf eine Listenposition, mit `.propname` oder `[#propname]` auf eine Eigenschaft der Propertyliste zugreifen. Gleichwohl ist der Typ von `me` nicht Liste. Ergänzen Sie das `mouseUp`-Skript um folgende Zeilen:

```
  put "---------"
  put listP(me)
  put objectP(me)
  put ilk(me)
```

Was folgende Ausgabe im Nachrichtenfenster ergibt:

```
-- "---------"
-- 0
-- 1
-- #instance
```

`me` gilt also nicht als Liste (was wir mit `listP()` überprüft haben). `me` ist Objekt, und der Typ von `me` (`ilk()`) ist `#instance`. Was aber ist ein Objekt, was eine Instanz? Es scheint so etwas wie ein Eigenschaftscontainer zu sein, vergleichbar einem Sprite, das ja auch zahlreiche Eigenschaften (Properties) besitzt.

Die Differenzprobe

Was ist `me`, wenn wir zweimal dasselbe Behavior auf ein Sprite platzieren? Mit folgendem Minimalskript:

```
on beginSprite me
  put me.spriteNum
  put me
end
```

und der Ausgabe im Nachrichtenfenster:

```
-- 5
-- <offspring "diff" 4 1a019d24>
-- 5
-- <offspring "diff" 4 1a01a620>
```

können wir schließen, dass die beiden Behaviors jeweils unterschiedliche mes besitzen, da die Speicheradressen der beiden Instanzen unterschiedlich sind. Gemeinsam ist beiden natürlich die Referenz auf das zugrunde liegende Behavior.

Thesen, Fragen

Wenn wir versuchen, das bisher Festgestellte auf einen Nenner zu bringen, stoßen wir auf Schwierigkeiten: Sollte me tatsächlich überflüssig sein, oder im besten Fall eine Formulierungserleichterung, die uns erspart, spritenum als Property auszuweisen (Weglassprobe)? Ist me der Einstieg in jene sagenumwobenen Welten der objektorientierten Programmierung, in denen es von Objekten und Instanzen nur so wimmelt (Inhaltsprobe) und wo die Kapselung von Informationseinheiten (Differenzprobe) so wichtig ist?

Beides ist richtig.

Versuchen wir es mit einem Bild: Nehmen wir an, der Director-Film (sobald er gestartet wird) ist eine Sandkiste. Nehmen wir weiter an, die vielen netten Förmchen, die wir haben, befinden sich in einem Eimer außerhalb der Sandkiste. Wenn wir eines der Förmchen nehmen und in den Sand drücken, „initialisieren“ wir es. Drücken wir es mehrfach in den Sand, so haben wir von derselben Form mehrere „Instanzen“ geschaffen. Jeder Abdruck ist klar unterscheidbar. Jeder einzelne der Abdrücke hat die Form seines Förmchens, ist aber nicht das Förmchen.

Das me gehört in diesem Szenario eindeutig zu den Abdrücken im Sand, das Behavior zum Eimer (Besetzung) und die Tätigkeit des Sandkastenspielers ist die des Director-Drehbuchs, das Instanzen von Behaviors erzeugt. Irgendwo in diesem Bild muss auch das Sprite seinen Ort haben. Im Experiment 2 kümmern wir uns darum.

▶ Gegenprobe: Ist me wirklich überflüssig?

Tatsächlich könnte me in einigen Behaviors in diesem Buch einfach weggelassen werden (in den meisten müssten Sie dann aber property spritenum nachtragen). Dass wir trotzdem beharrlich an jeden Event-Handler und jeden selbst definierten Handler in Behaviors ein me angehängt haben, liegt daran, dass wir *eine* konsistente Schreibweise in allen Behaviors haben wollten. Denn es gibt durchaus Fälle, in denen wir das me benötigen.

Legen Sie folgendes Skript auf ein Sprite:

```
on myputter inhalt
  put inhalt
end

on mouseUp
  myputter "Hallo Welt"
end
```

Bei Klick funktioniert das Skript: „Hallo Welt“ wird ins Nachrichtenfenster geschrieben.

Der folgende Aufruf im Nachrichtenfenster schickt den Befehl `myputter()` an alle Sprites:

```
sendAllSprites(#myputter, "Hallo Welt")
-- <offspring "putter" 4 1a01a300>
```

Statt „Hallo Welt“ wird allerdings etwas anderes ausgegeben – es sieht aus wie unser `me`! Tatsächlich benötigen Befehle wie `sendAllSprites()`, `sendSprite()` und `call()` das `me` als ersten Parameter des aufgerufenen Handlers.

Mit folgendem Behavior auf Sprite 6:

```
on myputter me, inhalt
  put inhalt
end

on mouseUp me
  myputter "Hallo Welt"
end
```

funktionieren alle folgenden Aufrufe:

```
sendallSprites(#myputter, "Hallo Welt")
-- "Hallo Welt"

sendSprite(6, #myputter, "Hallo Welt")
-- "Hallo Welt"

call(#myputter, sprite(6), "Hallo Welt")
-- "Hallo Welt"

sendallSprites(#mouseUp)
-- "Hallo Welt"

sendSprite(6, #mouseUp)
-- "Hallo Welt"

call(#mouseUp, sprite(6))
-- "Hallo Welt"
```

Das me scheint also nicht nur eine Eigenschaftssammlung zu sein, sondern auch eine Adresse, über die die Handler eines Skripts für die Kommunikation von außen erreichbar sind.

12.2 Experiment 2: me und Sprites

Sprites haben viele Funktionen – die vorangehenden Kapitel haben davon reichlich Gebrauch gemacht. Sie stellen Darsteller auf der Bühne dar – oder (falls Sie Gefallen an der objektorientierten Begrifflichkeit gefunden haben) Sprites sind automatisch gebildete *Instanzen* von Darstellern auf der Bühne. In Bezug auf Skripte haben Sprites eine andere, ebenfalls sehr wichtige Funktion: Sie sorgen dafür, dass Behaviors automatisch initialisiert werden, dass (s.o.) *Instanzen* von Behaviors gebildet werden.

new()

Dieser Vorgang ist ein sehr einfacher: Wenn der Abspielkopf im Drehbuch ein Sprite erreicht, auf dem ein Behavior liegt, so wird unbemerkt und automatisch der new()-Befehl an das Skript gesendet. Mit folgendem Skript können Sie das nachweisen:

```
on new me
  put "new:" & me
end

on beginsprite me
  put "beginsprite:" & me
end
```

Im Nachrichtenfenster erscheint:

```
-- "new:<offspring "newTest" 3 1a01a134>"
-- "beginsprite:<offspring "newTest" 4 1a01a134>"
```

Können wir eine solche Instanz auch selbst erzeugen? Ja! Duplizieren Sie das obige Behavior, nennen Sie es „newTest2“ und modifizieren Sie es wie folgt:

```
on new me
  put "new:" & me
  return me
end

on beginsprite me
  put "beginsprite:" & me
end
```

Dann geben Sie im Nachrichtenfenster Folgendes ein:

```
x = new(script "newTest2")
-- "new:<offspring "newtest2" 2 1a01a6e8>"
put x
-- <offspring "newtest2" 2 1a01a6e8>
```

Selbstverständlich erhalten Sie keine Ausgabe für `beginSprite`, da wir kein Sprite haben, das dieses Event empfangen könnte. Aber mit dem Befehl `new()` haben Sie eine Instanz eines Behaviors erzeugt, ebenso wie es ein Sprite mit den mit ihm verbundenen Behaviors tut.

Properties

In den vorausgehenden Kapiteln haben wir vereinfachend gesagt, dass in Behaviors definierte Properties zu Sprite-Properties werden. Wenn das Behavior die Property `myname` so definiert:

```
property myname

on beginSprite me
  myname = "hans"
end
```

dann können wir von außen (oder von einem anderen Behavior auf demselben Sprite) so auf die Eigenschaft zugreifen:

```
put sprite(1).myname
-- "hans"
```

In den allermeisten Fällen – nämlich dann, wenn wir einen Propertynamen in den verschiedenen Behaviors auf einem Sprite nur einmal verwenden – sind die Sprite-Property und die Property in der Instanz des Behaviors tatsächlich identisch.

Das folgende Skript wollen wir nun mehrfach auf einem Sprite platzieren; damit wir dennoch unterschiedliche Werte für die Property `myname` angeben können, definieren wir mit `on getPropertyDescriptionList` einen Konfigurationsdialog (auf der CD-ROM im Film *sprite2.dir*):

```
property myname

on beginsprite me
  put "---------"
  put sprite(me.spritenum)
  put me
  put me.myname
  put myname
  put sprite(me.spritenum).myname
  put sprite(me.spritenum).scriptinstancelist[1].myname
  put sprite(me.spritenum).scriptinstancelist[2].myname
  put (me = sprite(me.spritenum).scriptinstancelist[1])
end
```

```
on getPropertyDescriptionList
  l = [:]
  l[#myname] = [#comment: "Dein Name:", #default: "", \
   #format: #string]
  return l
end
```

Ziehen Sie das Behavior zweimal auf Sprite 1 und tragen Sie jeweils unterschiedliche Werte für `myname` ein.

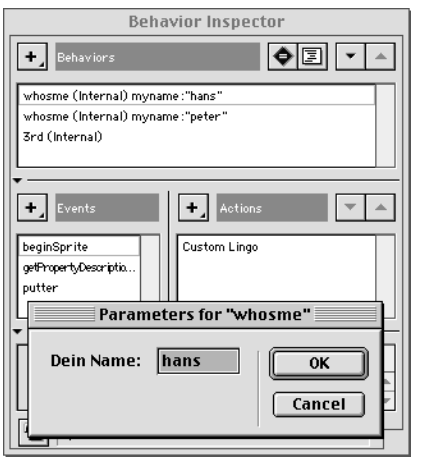


Abbildung 12.2: Ein Konfigurationsdialog ermöglicht, unterschiedliche Startwerte für Properties zuzuweisen.

Starten Sie den Film, dann sollten Sie die folgende Ausgabe im Nachrichtenfenster erhalten:

```
-- "---------"
-- (sprite 1)
-- <offspring "whosme" 4 1a01aa80>
-- "hans"
-- "hans"
-- "hans"
-- "hans"
-- "peter"
-- 1
-- "---------"
-- (sprite 1)
-- <offspring "whosme" 4 1a01a490>
-- "peter"
-- "peter"
-- "hans"
-- "hans"
-- "peter"
-- 0
```

Wir können feststellen:

- `sprite(me.spritenum)` ist für beide Behavior-Instanzen identisch; die Instanz `me` ist dagegen unterschiedlich.
- `me.myname` und `myname` liefern in beiden Instanzen unterschiedliche Werte zurück; innerhalb einer Instanz liefern beide Aufrufe aber denselben Wert.
- Interessant ist der Rückgabewert von `sprite(me.spritenum).myname`: Er liefert nämlich in beiden Instanzen den Wert der Property der ersten Instanz („hans“).
- Mit `sprite(me.spritenum).scriptinstancelist[index].myname` können wir gezielt auf die einzelnen Instanzen zugreifen.
- In der letzten Zeile des Skripts vergleichen wir schließlich `me` mit `scriptinstancelist[1]`. Erwartungsgemäß liefert die erste Instanz des Behaviors `TRUE`, die zweite `FALSE` zurück.

Der Ausdruck `sprite(1).myname` referiert nicht zwangsläufig auf die erste Behavior-Instanz. Vielmehr geht Director alle Behavior-Instanzen durch, bis das Programm eine zutreffende Property findet.

Wie wir mit

```
sprite(1).myname = "otto"
put sprite(1).scriptinstancelist[1].myname
-- "otto"
put sprite(1).scriptinstancelist[2].myname
-- "peter"
```

nachweisen können, gilt das auch beim Setzen der Properties.

Legen wir noch ein drittes Behavior auf unser Sprite, das die Property `dritte` enthält, so sehen wir aus

```
put sprite(1).dritte
-- "Im dritten Behavior"
put sprite(1).scriptinstancelist[1][#dritte]
-- <Void>
put sprite(1).scriptinstancelist[3][#dritte]
-- "Im dritten Behavior"
```

dass Director sich tatsächlich bis ins dritte Behavior durchhangelt, um die Property zu finden. (Wenn die Property in einer bestimmten Behaviorinstanz nicht existiert, würde der Zugriff mit `.dritte` einen Skriptfehler verursachen, der Zugriff über den Property-Namen in eckigen Klammern dagegen nicht. Deshalb ist hier die Schreibweise `[#dritte]` verwendet.)

Behaviors ohne Sprite: the actorlist, on stepFrame

Wenn wir tatsächlich mit `new()` unsere Behavior-Instanzen selbst initialisieren können – können wir dann auch Sprite-unabhängige Skript-Instanzen haben? Wir können!

Das Folgende können Sie mit dem Skripttyp „Parent“ ebenso bewerkstelligen wie mit einem normalen Behavior.

Legen Sie ein Behavior „newtest“ auf einem Sprite an, und zwar mit folgendem Skript:

```
property counter
on new me
  put "new:" & me
end
on beginsprite me
  put "beginsprite:" & me
  counter = 0
end
on prepareFrame me
  counter = counter + 1
  put "prepareFrame: " & counter
end
```

Ein zweites Behavior „newtest2“ soll lediglich in der Besetzung liegen:

```
property counter
on new me
  put "new:" & me
  counter = 0
  return me
end
on stepFrame me
  counter = counter + 1
  put "stepFrame: " & counter
end
```

`stepFrame` ist ein Event, der an alle Skriptinstanzen in der Systemproperty `the actorlist` gesendet wird. Er entspricht dem `prepareFrame`-Event für Sprites – nur eben ohne Sprite.

Beginnen Sie bei zunächst angehaltenem Film und geben Sie den ersten Teil im Nachrichtenfenster ein. Unsere selbst gebildete Behaviorinstanz wird mit dem Listenbefehl `add` in die `actorlist` eingetragen. Die Schreibweise `add the actorlist, x` ist in Director 7 nötig, in späteren Versionen können Sie auch `(the actorlist).add(x)` schreiben.

Im Nachrichtenfenster:

```
x = new(script "newtest2")
-- "new:<offspring "newtest2" 2 1a01a7b0>"
add the actorlist, x
put the actorlist
-- [<offspring "newtest2" 2 1a01a7b0>]
```

Film wird gestartet.

```
-- "new:<offspring "newTest" 3 1a019b80>"
-- "beginsprite:<offspring "newTest" 4 1a019b80>"
-- "stepframe:1"
-- "prepareFrame: 1"
-- "stepframe:2"
-- "prepareFrame: 2"
-- "stepframe:3"
-- "prepareFrame: 3"
-- "stepframe:4"
-- "prepareFrame: 4"
-- "stepframe:5"
-- "prepareFrame: 5"
```

Film wird angehalten und erneut gestartet.

```
-- "new:<offspring "newTest" 3 1a01a134>"
-- "beginsprite:<offspring "newTest" 4 1a01a134>"
-- "stepframe:6"
-- "prepareFrame: 1"
-- "stepframe:7"
-- "prepareFrame: 2"
-- "stepframe:8"
-- "prepareFrame: 3"
-- "stepframe:9"
-- "prepareFrame: 4"
-- "stepframe:10"
-- "prepareFrame: 5"
```

Nach dem Neustart des Films wird das Behavior erneut initialisiert (`new` und `beginSprite` werden ausgeführt, der `counter` steht wieder bei 0); unser Behavior in der `actorlist` dagegen hat das Anhalten des Films unbeschadet überstanden und zählt den `counter` einfach weiter.

12.3 Diskussion

Mit diesem kleinen Ausflug in die Bedeutung des Wortes `me`, das wir bisher ohne es zu hinterfragen in fast allen Skripts verwendet haben (nur in Filmskripten hat es keine Funktion), sind wir in einem sehr spannenden Gebiet gelandet: bei den objektorientierten Grundlagen des Programmierens mit Lingo. Die wichtigste Erkenntnis dieser Experimente sollte sein, dass bereits mit ganz normalen Sprites und Behaviors wesentliche Ideen der Objektorientierung in Director bereitliegen und ohne theoretische Verrenkungen nutzbar sind. Der Schritt zu Sprite-unabhängigen Skriptinstanzen ist dann nur ein sehr kleiner. Entscheiden Sie selbst, ob Sie mit diesen zusätzlichen Mitteln mehr – oder anderes – erreichen können als mit Sprite- und Frame-Behaviors.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z