Flash MX ActionScript

Unser Online-Tipp
für noch mehr Wissen ...

... aktuelles Fachwissen rund
um die Uhr – zum Probelesen,
Downloaden oder auch auf Papier.

www.InformIT.de

PEARSON Education

ADDISON-WESLEY

Jan Brücher
Marc Hugo

Flash MX ActionScript

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.

10 9 8 7 6 5 4 3 2 1

06 05 04 03

3-8273-1972-2

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

Printed in Germany

Inhaltsverzeichnis

Vorwort

Die Skriptsprache ActionScript erweitert die Funktionen, die Ihnen Flash bei der Gestaltung von interaktiven und animierten Webangeboten bietet, beträchtlich. Die Programmversion MX stellt gerade im Hinblick auf ActionScript noch einmal eine wesentliche Weiterentwicklung dar. Allerdings ist der Umgang mit dem Programm und der zugehörigen Skriptsprache mittlerweile vergleichsweise komplex und damit erklärungsbedürftig.

Wir verfolgen mit diesem Buch insofern mehrere Anliegen:

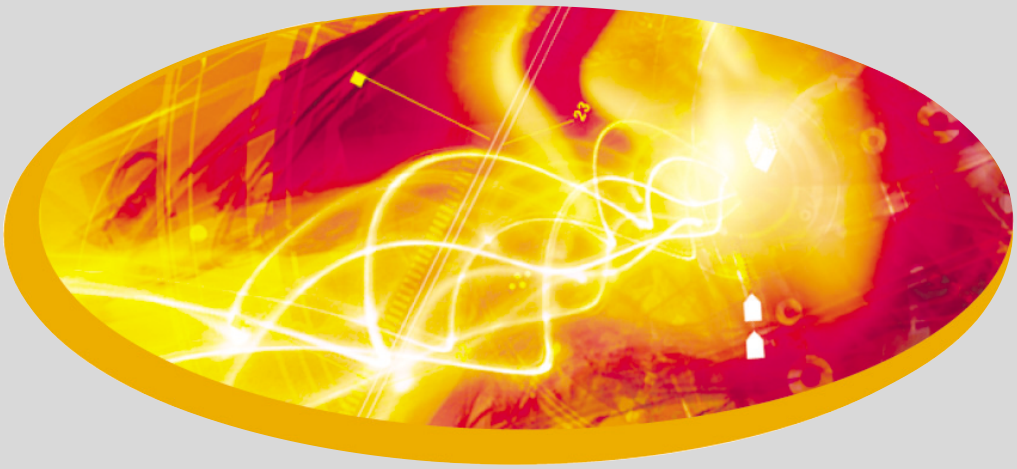
- Im ersten Teil führen wir systematisch in die Programmierung mit MX ActionScript, die Syntax und die verschiedenen Elemente der Skriptsprache ein. Kurze Beispiele veranschaulichen die Funktionsweise der einzelnen Befehle. Sie erhalten so einen Überblick über die Struktur von ActionScript und seine Besonderheiten sowie über die Neuerungen in der Version MX. Zudem gehen wir auch auf die Prinzipien des objektorientierten Programmierens ein. Diejenigen, die noch gar keine Programmiererfahrung besitzen, werden Schritt für Schritt mit ActionScript vertraut gemacht. Leser mit Programmiererfahrung haben die Möglichkeit, rasch nachzuschlagen, um ihre Kenntnisse zu vertiefen.

- Den zweiten Schwerpunkt des Buches stellen praktische Anwendungen dar: In Teil 2 zeigen wir Ihnen anhand zahlreicher Praxisbeispiele, was Sie mit ActionScript erreichen können. Hierbei beginnen wir mit Grundelementen, wie Sie sie in der Praxis täglich benötigen – etwa Maskeneffekte, Textfelder, Schaltflächen, Preolader, die Einbindung von Sound oder die Arbeit mit Komponenten. Anschließend behandeln wir dann eine Reihe von anspruchsvolleren Aufgabenstellungen: vom Arbeiten mit Shared Libraries über das Zusammenspiel mit serverseitigen Skriptsprachen bis zum Generieren von 3D-Effekten und zur Spieleprogrammierung (wobei auch hier ein Anwendungsbeispiel zur objektorientierten Programmierung nicht fehlen darf). Wir haben diese Anwendungen selbst entwickelt. Sie können sie Schritt für Schritt nachvollziehen und dank der beigegebenen CD-ROM auch selbst einsetzen und weiterentwickeln.
- Abschließend erhalten Sie eine Reihe von nützlichen Hinweisen zur Arbeit mit Flash – so zum Debugger, zum Zusammenspiel von Flash und HTML oder auch zu ASnative.

Wie wir aus eigener Erfahrung und der Beteiligung in den einschlägigen Flash-Foren wissen, bleiben bei der Arbeit mit ActionScript nicht selten Fragen offen. Deshalb finden Sie zu diesem Buch auch eine Internet-Seite: Auf der Website *www.DieFlasher.de* stehen wir Ihnen bei Problemen zur Seite. Das Passwort für den internen Bereich finden Sie am Ende des letzten Kapitels.

Wir wünschen Ihnen viel Spaß beim Arbeiten mit diesem Buch.

Jan Brücher, Marc Hugo Oktober 2002



Teil I: Grundlagen

Grundlagen Programmierung/ActionScript

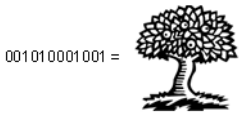
Flash MX bietet Ihnen mit ActionScript eine vollwertige Programmiersprache. Seitdem in Version 5 von Flash die Syntax von ActionScript grundlegend überarbeitet wurde und sich an den ECMA-Standard 245 anlehnt, ist das Programmieren mit Flash um einiges komplexer geworden. Während man in Flash 4 durch Zusammenklicken zu einem akzeptablen ActionScript-Code gelangte, muss man nun in den meisten Fällen den Code per Hand modifizieren. ActionScript bietet fast alle Vorzüge einer modernen Programmiersprache.

Selbst objektorientiertes Programmieren ist möglich. Trotzdem bietet Flash für Programmieranfänger die Option, den Code zusammenzuklicken, durch den Normalmodus im ActionScriptfenster. Gerade für Einsteiger bietet dies einen leichten Einstieg in die doch recht komplexe Programmierwelt.

Falls Sie noch wenig Programmiererfahrung haben, sollten Sie sich dieses Kapitel aufmerksam durchlesen, da es das Grundgerüst für den Einstieg in ActionScript bildet. Alle anderen können dieses Kapitel getrost überspringen, zumindest die ersten Unterpunkte, da es sich hierbei wirklich nur um eine grundlegende Einführung handelt.

1.1 Der Einstieg

Die Kunst des Programmierens liegt darin, die Sprache des Menschen in eine für den Computer verständliche Syntax umzusetzen. Ein Programmierer ist also quasi nur ein „Übersetzer“.



Angenommen Sie wollen ein kleines Spiel erstellen, bei dem der Benutzer (User) über die Cursortasten (Pfeiltasten) ein Auto steuern kann. Ihre Aufgabenstellung würde lauten:

„Ein Auto kann über die Pfeiltasten bewegt werden, realisieren Sie dies."

User
Ein User ist derjenige, der später vor dem Computer sitzt und Ihr Programm bedient.

Wenn Sie dies dem Computer so mitteilen könnten, fehlen jedoch grundlegende Daten:

- Was ist ein Auto?
- Was passiert bei welcher Pfeiltaste?
- Was ist Bewegung?

Mittels ActionScript können Sie dem Computer (bzw. dem Interpreter) genau sagen, was er zu tun hat. ActionScript ist eindeutig definiert, so dass es vonseiten des Rechners zu keinen Missverständnissen kommt. Trotzdem wird nicht immer alles so laufen wie Sie es geplant haben. In diesem Fall benutzen Sie den Debugmodus. Für eine kleinere und kompaktere Fehlersuche bietet Flash, schon seit der Version 4 zusätzlich zum Debugmodus den Trace() – Befehl.

Mit diesem kann man eine einfache, nur für den Programmierer sichtbare Ausgabe erzeugen, indem der Parameter des Tracebefehls angezeigt wird. So lässt sich auf einfache Weise bei gewissen Events der Inhalt einer Variablen ausgeben.

Da die Anwendung dieses Befehls sich so einfach gestaltet, werden wir diesen nutzen, um mit Ihnen zusammen das erste kleine Programm zu erstellen.

Es gibt insgesamt drei Möglichkeiten, ActionScript in Flash zu integrieren:

Zum einen in einem Frame der Zeitleiste – dieser Befehl wird ausgeführt, sobald der Abspielkopf das entsprechende Frame erreicht und abspielt.

Frame
Ein einzelnes Bild der Zeitleiste

Keyframe
Ein Schlüsselbild in der Zeitleiste, erkennbar an dem kleinen Kreis links unten.

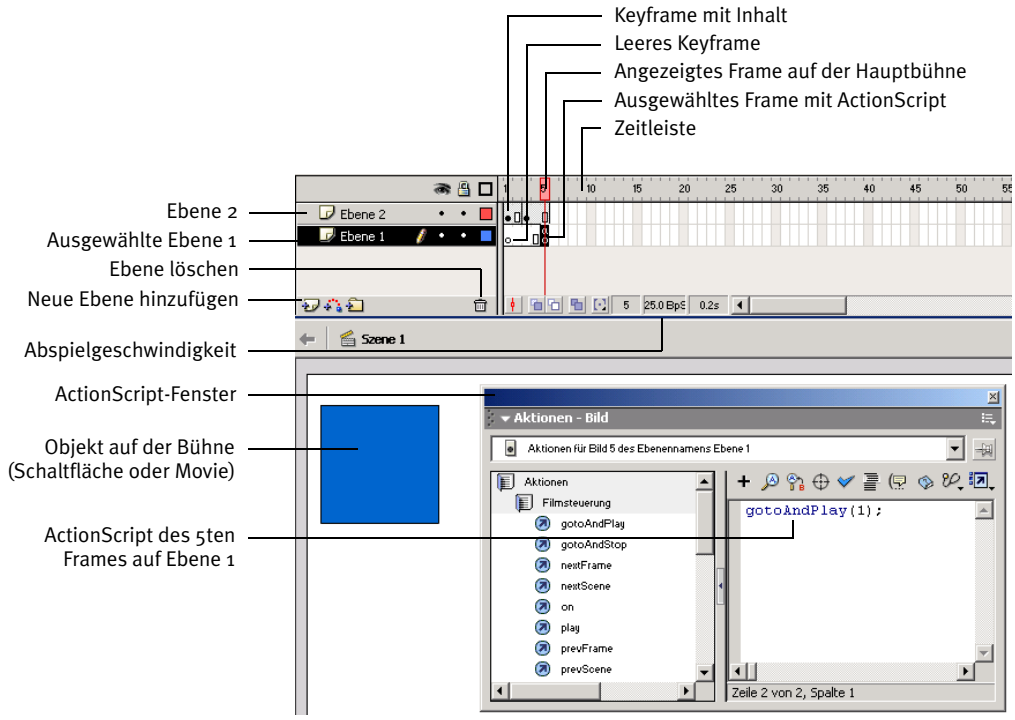


Abbildung 1.1: Flash MX-Bühne mit ActionScript-Fenster

Zum anderen kann man ein ActionScript für Schaltflächen (Buttons) definieren. Dabei findet man das ActionScript direkt auf der jeweiligen Schaltfläche auf der Bühne – wann es ausgeführt wird, hängt von der Einstellung ab. Jedoch wird dieses ActionScript immer erst auf eine bestimmte Aktion der Maus ausgeführt (Mausklick, Mausover usw. siehe Kapitel 4.3).

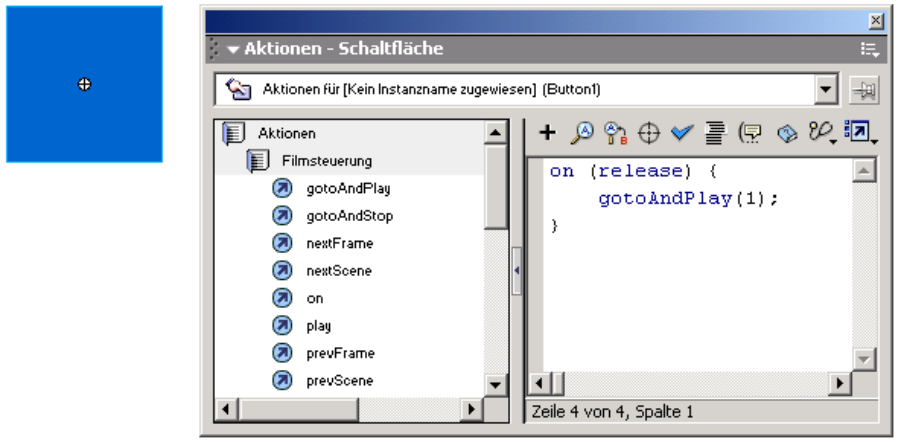


Abbildung 1.2: ActionScript eines Buttons

Die dritte Möglichkeit ist, ein ActionScript auf eine Filmsequenz (Movie) zu legen. Hierfür gibt es verschiedene Einstellungen, die bestimmen, wann das Script ausgeführt wird. Allerdings warten diese nicht zwingend auf eine Aktion des Benutzers, sondern sind an die Aktionen der Filmsequenz gebunden.

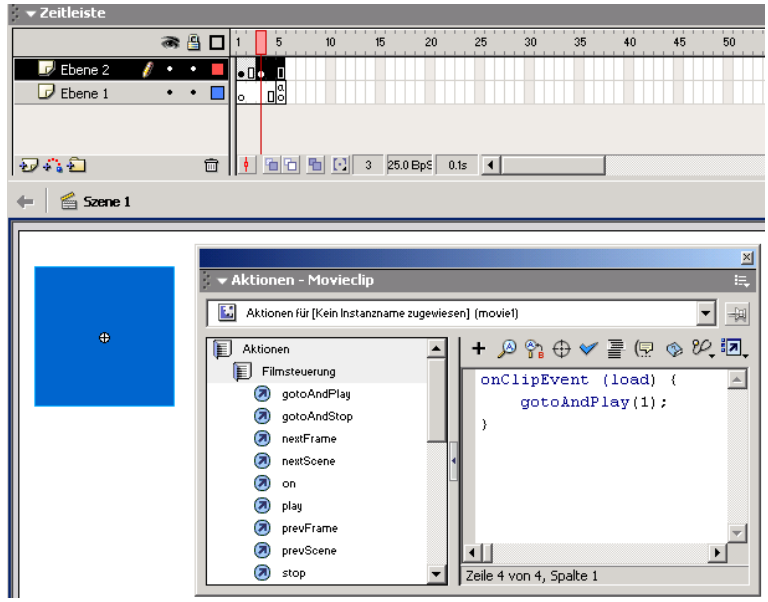


Abbildung 1.3: ActionScript eines Movies

Für das erste kleinere Programm mit dem Befehl `trace()` fügen Sie das ActionScript in das erste Frame in der Hauptzeitleiste ein. So wird das ActionScript bzw. der Befehl direkt beim Start Ihres Programms ausgeführt.

Hauptzeitleiste

Jedes Movie hat seine eigene Zeitleiste. Da es nur eine permanente Zeitleiste auf der Hauptbühne gibt, bezeichnet man diese als Hauptzeitleiste.

Um das ActionScript dort einzufügen, wählen Sie das erste Frame aus und gehen in das ActionScript-Fenster.

Ihre Aufgabe ist, das Wort „Hallo“ auf dem Bildschirm auszugeben. Für eine Ausgabe benutzen Sie den Befehl `trace()`, dieser benötigt genau einen Parameter.

Theoretisch könnte man auf die Idee kommen `trace(Hallo)` zu schreiben, allerdings würde Flash dann nach einer Variablen suchen, die Hallo heißt. Da Sie einen String (eine Zeichenkette) ausgeben möchten, müssen Sie das Wort (die Zeichen, im Beispiel: „Hallo“) in Anführungszeichen setzen.

```
trace(„Hallo");
```

Listing 1.1: Ausgabe eines Strings in einem nur für den Programmierer sichtbaren Fenster

In Flash wird jeder Befehl durch ein Semikolon beendet.

Wenn Sie nun diesen Befehl so in das erste Frame eingefügt haben und Ihr Programm testen (Strg+↵ oder Steuerung › Film testen), sehen Sie, wie Flash ein Ausgabefenster für Sie öffnet und Ihnen Ihre Botschaft (Ihr „Hallo“) mitteilt. Dieses Ausgabefenster wird nur direkt unter Flash mit angezeigt. Wenn Sie Ihr kleines Programm erstellen bzw. veröffentlichen (Strg+F12 oder Datei › Veröffentlichen) und es danach einzeln starten (über den Browser), sehen Sie, dass Flash dort nun kein extra Fenster öffnet. Dies liegt daran, dass der Befehl `trace()` zur Fehlersuche (Debuggen) gedacht ist und so nur direkt unter Flash für den Programmierer selber sichtbar wird.

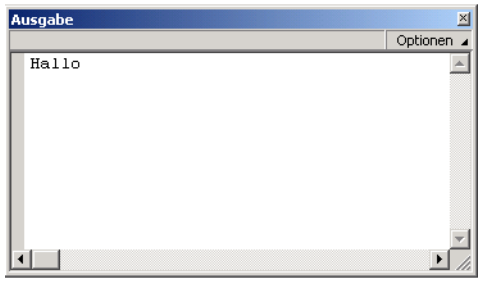


Abbildung 1.4: Erzeugtes Ausgabefenster (nur in Flash sichtbar)

Falls Sie wirklich auf Ihrer Internetseite ein solches Fenster öffnen möchten, geht dies über JavaScript im Zusammenhang mit dem alert-Befehl oder mit einem von Ihnen direkt unter Flash erstellten Fenster (dieses wird allerdings nur im Flash-Bereich mit angezeigt und ist nicht wirklich ein neues Fenster). Dazu jedoch in den folgenden Kapiteln mehr.

1.2 Variablen

Wie oben schon angedeutet, gibt es in Flash Variablen. Variablen sind eine Art Speicher bzw. Container für einen Wert. Stellen Sie sich vor, Sie haben gerade einen Termin vereinbart und notieren sich diesen nun auf einem Zettel. Genauso speichert man dies in Flash in Variablen, um sich den Wert quasi zu merken.

Eine gängige Metapher ist, die Variablen mit Schubladen zu vergleichen. Sie speichern den Wert in einer Variable, und Flash legt diesen für Sie in einer Schublade ab. Wenn Sie nun wieder auf die Variable zugreifen, öffnet Flash die Schublade und gibt Ihnen den entsprechenden Wert wieder aus. Damit Flash immer die richtigen Schubladen für Sie öffnet, müssen diese eindeutig benannt sein. So dürfen Variablen nicht mit Ziffern beginnen, keine Leerzeichen oder Sonderzeichen enthalten, Ausnahme sind Unterstriche. Variablen dürfen nicht wie eines der reservierten

Schlüsselwörter lauten (ein Variablenname `load` geht z.B. nicht). Außerdem sind Variablennamen in Flash nicht case-sensitive – die Groß- und Kleinschreibung ist irrelevant.

Eine ausführliche Liste mit allen Schlüsselwörtern finden Sie in Anhang A.1.

Variablennamen müssen mit einem Buchstaben beginnen und dürfen keine Sonderzeichen wie Umlaute etc. enthalten.

Verwenden Sie nicht die gleichen Namen für Instanzen, Textfelder, Objekte, Funktionen oder Variablen, wenn diese in demselben Movie liegen. Grundsätzlich sollte man aber immer unterschiedliche und möglichst eindeutige (selbst sprechende) Namen verwenden.

Variablen deklarieren

Alle möglichen Variablentypen können unter Flash durch das Schlüsselwort „var“ deklariert werden, müssen dies aber nicht zwingend. „var“ bedeutetet, dass die Variable nur lokal z.B. in einer Funktion gehalten wird. Mehr dazu später in „Gültigkeitsbereich (Scoping)“ ab Seite 93.

Schlüsselwörter

Schlüsselwörter sind „var“ oder „trace“. Diese sind in der ActionScript-Syntax fest eingebunden. Schlüsselwörter können nicht mit neuen Bedeutungen belegt werden und können auch nicht als Variablen-, Funktions-, Instanz-, Objektnamen oder Ähnliches benutzt werden. Sie sind feste Bestandteile der Programmiersprache.

Man muss bei der Deklaration von Variablen nicht angeben, um welchen Typ es sich handelt. Eigentlich ist es nicht nötig, neu eingeführte Variablen mit „var“ zu deklarieren. Variablen, die deklariert und nicht initialisiert (es erfolgt eine Wertezuweisung) werden, erhalten automatisch den Typ `undefined`.

Da man Variablen normalerweise immer zu Beginn eines ActionScripts deklariert, liegt der Vorteil darin, dass man direkt erkennt, welche Variablen im gesamten Script benutzt werden.

Der Preis des Features, dass Variablen nicht mit einem Schlüsselwort deklariert werden müssen, ist aber sehr hoch. Diese Eigenschaft ist allerdings in Flash implementiert und lässt sich nicht ändern. In Programmiersprachen, in denen Variablen deklariert werden müssen, gibt der Compiler bzw. Interpreter eine Fehlermeldung aus, falls während des Programms eine nicht eingeführte Variable angesprochen wird. Dies geschieht bei Flash nicht, da hier keine Variablen zwingend eingeführt werden müssen!

Wenn man sich z.B. bei einem größeren ActionScript im späteren Verlauf der Programmierung bei einem Variablennamen verschreibt und so statt `geschwindigkeit` versucht eine Variable namens `gschwindigkeit` (das „e“ fehlt) anzusprechen, gibt Flash dabei keine Fehlermeldung aus, sondern erstellt sogar noch eine Variable mit dem falschen Namen.

Solche Fehler können schwer aufzufinden sein. Hilfe bietet hier der Debugger, in dem alle benutzten Variablen aufgeführt sind. Mehr zum Debugger erfahren Sie in Kapitel 9.2 ab Seite 324.

Es gibt in Flash, wie zuvor schon erwähnt, verschiedene Typen von Variablen. Oben in dem ersten Beispiel haben Sie das Wort „Hallo“ ausgegeben. Das Wort „Hallo“ gehört zum Typ `String` (Zeichenkette). Dies erkennen Sie daran, dass es in Anführungszeichen geschrieben ist. Wenn Sie diesen String nun einer Variablen zuweisen, spricht man, wenn man sich direkt auf die Variable bezieht, auch von einem String, da sie diesen enthält.

```
var var1="Hallo";
```

Listing 1.2: Deklaration einer Variable mit Zuweisung eines Strings

`Var1` steht in diesem Buch immer für einen x beliebigen Variablennamen. Falls mehrere Variablen benötigt werden, heißen diese `var1`, `var2` usw. Ausnahmen bilden Index- bzw. Zähler-Variablen, die in Schleifen (Kapitel 1.6) benutzt werden, denen die Werte `i`, `j`, `k` zugewiesen werden.

Mögliche Typen von Variablen

In Flash gibt es drei grundlegend unterscheidbare Typen von Variablen:

- Numerisch
- String
- Boolean

Darüber hinaus kann eine Variable noch die folgenden Zustände besitzen:

- Undefined
- Null

Diese beiden Zustände werden auch als eigener Typ geführt, obwohl es sich dabei auch dann um einen Typ von Variable handelt, der nur einen Wert (Zustand) annehmen kann. Da eine Variable vom Typ `undefined` oder `null` ist, automatisch auch diesen Wert enthält, also `undefined` oder `null`, weitere Werte sind nicht zulässig.

▶ Numerisch

Kommen wir nun zum einfachsten Typ, den man einer Variable zuweisen kann: dem Typ *numerisch*. Anders als in gängigen Programmiersprachen wie C++ oder Java, werden in Flash die verschiedenen numerischen Variablentypen nicht noch weiter durch deren Größe in Byte unterteilt. So ist es in Flash irrelevant, ob man eine Variable für Gleitkommazahlen oder ganzzahlige Werte (z.B. integer) einführt. Wenn man eine Variable explizit als `Number` einführt, wird dieser automatisch der Wert `0` zugewiesen.

`Number` ist hierbei das Schlüsselwort und dient dazu, dem Interpreter mitzuteilen, dass man eine Variable vom Typ *numerisch* definieren möchte.

```
var var1=Number(var1);
trace(var1);
```

Listing 1.3: Explizite Einführung einer Variablen als Number

Bildschirmausgabe: `0`

Konstruktorfunktion
Die Konstruktorfunktion wird aufgerufen, um ein neues Objekt zu bilden. Flash ruft diese meist automatisch mit auf, wenn man z.B. eine bisher nicht definierte Variable anspricht. Man kann diese aber auch explizit aufrufen durch Verwenden des Operators `new` und des entsprechenden Klassennamens. Mehr zu Objekten und Klassen in Kapitel 1.8.

Casten
Casten ist ein Oberbegriff für das Umwandeln einer Variablen von einem Typ (Number, String) zu einem anderen. Siehe auch „Typenwandlung (Casting)“ ab Seite 35.

typeof ()
Mit `typeof()` ermittelt man den Typ der Variablen, ob es sich z.B. um einen String, eine Number-Variable handelt. In Klammern als Parameter wird die Variable mit angegeben, als Rückgabewert erhält man den Typ.

Normalerweise sollte man eine Variable vom Typ Number auch über die Konstruktorfunktion erzeugen können. Leider gibt Flash in diesem Fall immer den Typ `object` statt `number` an. Deshalb muss man sich mit dem obigen Trick des Castens behelfen, falls man eine Variable leer einführen möchte und evt. anschließend mit der `typeof()`-Funktion darauf zugreift.

```
var1 = new Number();
trace(typeof(var1)+" "+var1);
```

Listing 1.4: Deklaration einer Numbervariablen mit Hilfe des Konstruktors von Number

Bildschirmausgabe: `object 0`
Man erkennt, dass Flash der Variable vom Typ `number` den Number-Standardwert (Defaultwert) `0` zugewiesen hat. Wenn man allerdings mit `typeof()` den Typ der Variablen abfragt, wird fälschlicherweise `object` ausgegeben!

Variablenoperationen
Numerische Werte können einfach addiert werden.

```
var var1=3;
var var2=4;
var var3 = var1 + var2;
```

Listing 1.5: Deklaration und anschließende Addition von zwei Variablen

Der Wert der Addition wurde in der Variable `var3` gespeichert, diese hat nun den Wert `7`. Links steht immer die Variable, der ein Wert zugewiesen wird. Auf der rechten Seite hinter dem Gleichzeichen steht der Term, dessen Ergebnis der linken Variablen zugewiesen werden soll. Natürlich kann man genauso gut multiplizieren, subtrahieren usw. Die verschiedenen Operatoren stellen wir Ihnen noch gesondert in Kapitel 1.3 ab Seite 37 vor.

Eine Mehrfachzuweisung ist in Flash auch möglich.

```
var1=var2=5;
```

Listing 1.6: Mehrfachzuweisung in Flash

Der Wert der Variablen `var1` und `var2` wäre nach so einer Zuweisung `5`.

Korrekt ausgedrückt müsste es lauten: Der Wert 5 wird der Variablen `var2` zugewiesen. Der Wert der Variablen `var2` wird der Variablen `var1` zugewiesen. Der Interpreter arbeitet diese Art von Anweisungen immer von rechts nach links ab.

Binär-Werte

Wie zuvor in dem kleinen Exkurs beschrieben, ist das Binärsystem das grundlegende System des Computers. Es besitzt zwei Zustände: 1 und 0. Wenn man also nun eine Zahl darstellte, sieht sie evt. wie folgt aus:

```
0011 0010
```

Die Unterteilung in Vierer-Blöcke haben wir nur wegen der Übersicht vorgenommen. Genau vier Blöcke, da diese exakt zu einer Hexadezimalzahl zusammengefasst werden können.

Man liest die Zahlen immer von rechts nach links, umrechnen lässt sich das Ganze über die Basis, die im Binärsystem 2 ist. Wenn man die Zahlen von oben senkrecht untereinander schreibt, erhält man folgendes Bild:

Binär	zur Basis	Wert	Dezimal
0	2^0	1	0 (=0*1)
1	2^1	2	2 (=1*2)
0	2^2	4	0 (=0*4)
0	2^3	8	0 (=0*8)
1	2^4	16	16 (=1*16)
1	2^5	32	32 (=1*32)
0	2^6	64	0 (=0*64)
0	2^7	128	0 (=0*128)

Zahlensysteme und Bits und Bytes

Es gibt noch andere Möglichkeiten, Dinge zu zählen, als die gängige Methode. Für uns ist es selbstverständlich, dass nach der 9 die 10 folgt, doch was passiert, wenn man beschließt, nicht mehr bis 9, sondern nur noch bis 7 zu zählen? Und nach der 7 kommt die 10.

Alles würde komplett anders aussehen.

Die Art bis 9 zu zählen, bzw. eine Basis von 10 zugrunde zu legen (zehn Werte {0,1,2,3,4,5,6,7,8,9}), beschreibt das Dezimalsystem.

Wenn man nur noch bis 7 zählen würde, beinhaltet das System acht Werte {0,1,2,3,4,5,6,7}, deshalb spricht man von einem Octal-System. Es gibt theoretisch unendlich viele Systeme – 3-Systeme oder 52´er. Die gängigen sind allerdings: 2 (binär), 8 (octal), 10 (dezimal), 16 (hexadezimal).

Sicherlich fragen Sie sich, wofür man die verschiedenen Zahlensystem benötigt. Gerade Programmierer müssen öfters mit anderen Zahlensystemen umgehen. Der Rechner kann „intern“ nur mit 0 und 1 rechnen, hierbei hat man also ein System, das nur zwei Werte beinhaltet, deshalb heißt dieses auch Binär-System. Wenn man über einen dieser Wertezustände spricht, nennt man diesen „Bit“.

8 Bit werden zusammengefasst zu 1 Byte.

Ein Byte kann 256 verschiedene Werte annehmen. (Wie man dies errechnet, wird im Abschnitt „Octal-Werte“ ab Seite 23 erläutert)

So ist es auch verständlich, dass der normale ASCII-Code (Zeichensatzstandard, siehe Anhang A) auch genau 256 Zeichen beinhaltet. Alles in der Welt des Computers stützt sich darauf. Genauso wie das Octal- und vor allen Dingen auch das Hexadezimal-System eine wichtiger Rolle spielen, da sie ein Vielfaches von 2 sind.

Dementsprechend lassen sich auch die Zahlen für die verschiedenen Farbtiefen erklären, seien es

- 1 Bit; 2 Farben (schwarz weiß)
- 3 Bit; 8 Farben (CGA)
- 4 Bit; 16 Farben (EGA)
- 8 Bit; 256 Farben (VGA)
- 16 Bit; 65.536 Farben (Super VGA / High Color)
- 24 Bit; 16,7 Millionen Farben (True Color 24 Bit)
- 32 Bit; 42.949,7 Millionen Farben (True Color 32 Bit)

In Truecolor lassen sich 42.949,7 Millionen verschiedene Kombinationen – Farben – darstellen.

Diese Anzahl der Bits ist nicht zu verwechseln mit der eines 32-Bit-Betriebssystems. Diese Zahl steht dabei im Zusammenhang mit der Bandbreite zur CPU. Also für die innerhalb eines Taktzyklus abgearbeitete Anzahl an Bits.

8 Bit nennt man auch „Byte“ und 2 Byte wiederum ein „Wort“. 2 „Wort“ ergeben ein „Langwort“, auch bekannt als „DoppelWort“.

DoppelWort * DoppelWort ergibt 1 kByte. Anders als von vielen Anfängern angenommen, entsprechen 1 kByte daher nicht 1000 Byte, sondern 1024 Byte, dies wären wiederum 8192 Bit. 1 kByte * 1 kByte ergibt MByte (MegaByte). 1MB enthält genau 1.048.576 Byte (1024 Byte *1024 Byte) und 1 GByte (GigaByte) wiederum 1 kByte * 1 kByte * 1 kByte also 1.073.741.824 Byte. Dies entspricht 8.589.934.592 Bit. Wenn Sie das Wort „Flash“ auf Ihrer Festplatte speichern, benötigt dieses 5 Byte Speicherplatz. Sie können sich nun ausrechnen, wie oft Sie dieses Wort auf Ihrer Festplatte abspeichern können.

Der Binärwert wird mit dem Wert in der Zeile „Wert“ multipliziert (Hier spielt dies zwar noch keine Rolle, man hätte auch einfach sagen können, dass dieser übernommen wird. Aber bei einem anderen Zahlensystem, wo mehr als nur 1 oder 0 zu Beginn stehen, sieht dies schon wieder ganz anders aus.). Wenn man nun alle Zahlen aus dem Feld „Dezimal“ addiert erhält man den Dezimalwert für die Zahl.

0011 0010 entspricht in Dezimalschreibweise 50.
Eine 7 entspricht in Binärschreibweise 0111.
Eine 4 entspricht 0100.

Wofür man Binäre Zahlen direkt in Flash nutzen kann, wird in Kapitel 1.3, Abschnitt „Bitweise Operatoren“ ab Seite 47 beschrieben.

Octal-Werte

Das Octal-System unterscheidet sich vom Binärsystem darin, dass es mehr Werte annehmen kann als 2, nämlich 8 {0,1,2,3,4,5,6,7).

Im Octalsystem zählt man:

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20

Dementsprechend gibt es z.B. eine 20 auch im Octalsystem, die aber eine ganz andere Wertigkeit besitzt als die 20 im Dezimalsystem.

Wenn man eine Zahl ins Dezimalsystem umrechnen möchte, kann man die Zahl zuerst im Binärsystem ableiten und dann wieder im Dezimalsystem oder direkt, wie zuvor schon bei den Binärzahlen – nur dieses Mal zu einer anderen Basis:

Octal	zur Basis	Wert	Dezimal
3	8^0	1	3 (=3*1)
2	8^1	8	16 (=2*8)
4	8^2	64	256 (=4*64)
6	8^3	512	3072 (=6*512)

Die Octalzahl 6423 entspricht also 3347.

In Flash werden Octalzahlen durch eine vorangestellte `0` gekennzeichnet.

```
trace(06423);
```

Listing 1.7: Ausgabe einer Octalzahl

Dies würde folgende Ausgabe ergeben: 3347

Es ist auch möglich, Octalzahlen direkt in eine Binärzahl zu wandeln, da eine Octalzahl genau acht Kombinationen besitzt. Acht Kombinationsmöglichkeiten hat man im Binärbereich durch 3 Bits. Daraus folgt, dass eine Octalzahl 3 Bits entspricht. Man wandelt also einfach Ziffer für Ziffer um und setzt die einzelnen binären drei Blöcke später wieder aneinander.

So entspricht die Octalzahl	6	4	2	3
der Binärzahl	110	100	010	011
Zusammengesetzt also	1101	0001	0011	

Hexadezimal-Werte

Das Hexadezimalsystem unterscheidet sich nicht großartig von den bisher vorgestellten Systemen. Der Unterschied liegt darin, dass man bis 16 zählt. Da das normale Dezimalsystem aber „nur“ zehn „Symbole“ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} besitzt, muss man für die restlichen sechs Stellen neue Symbole einführen {A, B, C, D, E, F}.

Wenn man diese Zählweise dem Dezimalsystem gegenüberstellt, sieht dies wie folgt aus:

Dezimal	Hexadezimal	Dezimal	Hexadezimal
0	0	10	A
1	1	11	B
2	2	12	C
3	3	13	D
4	4	14	E
5	5	15	F
6	6	16	10
7	7	17	11
8	8	18	12
9	9	19	13
		20	14

Diese Gegenüberstellung soll verdeutlichen, dass man mit A, B, C usw. normal weiter zählt.

Die Umrechnung ins Dezimalsystem kann wie zuvor auch schon durchgeführt werden.

Hex	zur Basis	Wert	Dezimal
A	16^0	1	10 =(10*1)
4	16^1	16	64 =(4*16)
2	16^2	256	512 =(2*256)
1	16^3	4096	4096 =(1*4096)

Die Hex-Zahl 124A entspricht demnach der Dezimalzahl 4682.

In Flash werden Hexadezimalzahlen durch ein vorangestelltes `0x` gekennzeichnet.

```
trace(0x124A)
```

Listing 1.8: Ausgabe einer Hexadezimalzahl

Dies erzeugt folgende Bildschirmausgabe: 4682

Es ist auch möglich, Hexadezimalzahlen direkt in eine Binärzahl zu wandeln, da eine Hexadezimalzahlen genau 16 Kombinationen besitzt. 16 Kombinationsmöglichkeiten hat man im Binärbereich durch 4 Bits. Daraus folgt, dass eine Hexadezimalzahlen 4 Bits entspricht. Man wandelt also auch diese wie zuvor bei den Octalzahlen Ziffer für Ziffer um und setzt die einzelnen binären vier Blöcke später wieder aneinander.

So entspricht die Hexadezimalzahl	1	2	4	A
der Binärzahl	0001	0010	0100	1010

Zusammengesetzt also 0001 0010 0100 1010

▶ Strings

Ein weiterer Variablentyp, den wir zu Beginn schon vorgestellt haben, ist der Typ `String`. Dieser speichert alle Zeichen in einer Variablen. Es ändert sich nichts bei der Zuweisung oder der Deklaration der Variablen zum vorangegangen Beispiel. Allerdings spricht man bei einer Variablen, die eine Zeichenkette enthält, von einem String.

```
var var1=String(var1);
trace(var1);
```

Listing 1.9: Explizite Deklaration einer Variablen als String.

Man achte darauf, dass `trace()` eine leere Zeichenkette "" ausgegeben hat und nicht undefined.

```
var var1="Hallo";
```

Listing 1.10: Zuweisung des Wortes „Hallo“ zu der neu eingeführten Variable var1

Man kann diese Variable jetzt auch über `trace()` ausgeben.

```
var var1="Hallo";
Trace(var1);
```

Listing 1.11: Ausgabe des Wortes „Hallo“ in einem nur für den Programmierer sichtbaren Fenster

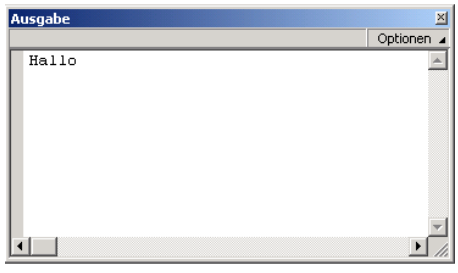


Abbildung 1.5: Erzeugtes Ausgabefenster (nur in Flash sichtbar)

Natürlich konnte man die beschriebenen numerischen Variablen genauso mit `trace()` ausgeben.

Strings erkennt man immer daran, dass sie in Anführungszeichen stehen. Eine weitere mögliche Notation für einen String ist das '-Zeichen.

```
"Dies ist ein String"
```

funktioniert dementsprechend genauso gut wie

```
'Dies ist ein String'
```

Der grundlegende Unterschied liegt also darin, dass in Strings, die durch Anführungszeichen definiert wurden, Hochkommata zulässig sind und dass durch Hochkommata definierte Strings Anführungszeichen enthalten können.

Wenn Sie also den Text

Heute ist ein „schöner Tag“

ausgeben möchten, benutzen Sie die Hochkommata-Notation, da Flash sonst Ihr zweites Anführungszeichen, dass Sie benutzen, um die Worte „schöner Tag“ hervorzuheben, als Ende des String ansieht. Flash versucht dann, den restlichen Teil des Strings zu interpretieren, und da dieser nicht der Syntax entspricht, würden Sie eine Fehlermeldung bekommen.

Stringvar

Bei `Stringvar` handelt es sich nicht um ein Schlüsselwort von Flash, es ist ein Variablenname, der andeuten soll, dass es sich bei der Variablen um einen String handelt. So wie bei den vorangegangenen Beispielen schon `var1`, `var2` als Variablenname für Numberwerte dienten.

Listing 1.12: *Ein String mit Hochkommata-Notation*

```
var Stringvar;
Stringvar = 'Heute ist ein "schöner Tag"';
```

Ein ähnliches Beispiel mit einem Hochkomma im String:

Listing 1.13: *Ein String in normaler Notation kann Hochkommata enthalten*

```
var Stringvar
Stringvar = "Fehler des Computers:  'Tastatur nicht gefunden:
Drücken Sie bitte F1'";
```

Es ist darauf zu achten, dass es sich bei dem benutzten geraden Hochkomma um das Hochkomma links neben der Returntaste handelt (') und nicht um die Taste, die rechts oben neben dem Fragezeichen sitzt (´).

Bei dieser Art der Notation ergibt sich trotzdem noch ein Problem. Stellen Sie sich vor, Sie haben einen Satz, der Hochkommata und Anführungszeichen enthält.

Fehler des Computers: „Tastatur nicht gefunden: Drücken Sie bitte F1“

Wenn Sie Hochkommata und Anführungszeichen in einem Satz benutzen müssen oder ein anderes Sonderzeichen benutzen, können Sie auf das Fluchtzeichen zurückgreifen. Das Fluchtzeichen ist der Backslash (\). Dieser ermöglicht es Ihnen, bestimmte Zeichen darzustellen.

```
var Stringvar;
Stringvar = "Fehler des Computers:  \"Tastatur nicht gefunden:
Drücken Sie bitte F1\" ";
```

Listing 1.14: Ein String mit Hochkommata und Anführungszeichen

Hinter dem Fluchtsymbol (\) schreiben Sie einfach das darzustellende Zeichen, so weiß Flash, dass es sich nur um ein Zeichen handelt und nicht um ein Schlüsselwort bzw. einen Teil der Syntax. Das Fluchtsymbol ist nicht Teil der Ausgabe.

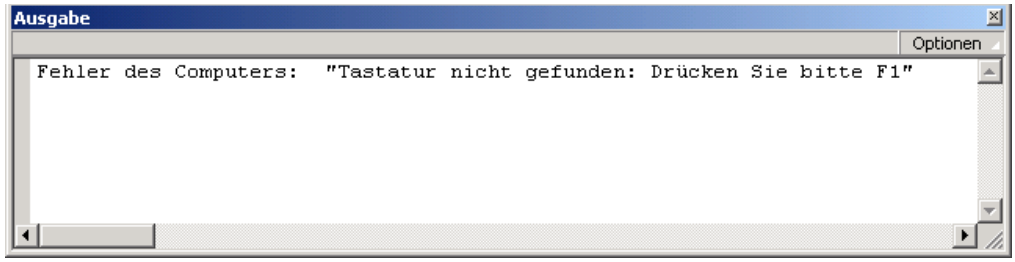


Abbildung 1.6: Ausgabefenster

Es gibt für das Fluchtsymbol noch folgende Notationen:

Fluchtsymbol	Bedeutung
\'	Ausgabe eines Hochkommas
\"	Ausgabe des Anführungszeichens
\t	Tabulator (ASCII 9)
\r	carriage return, Zeilenumbruch (ASCII 13)
\n	newline, Zeilenumbruch (ASCII 10)

Tabelle 1.1: Fluchtsymbol-Syntax

So können Sie ohne Probleme auch einen Zeilenumbruch damit herbeiführen.

```
var Stringvar;
Stringvar = "Dieser Text enthält einen \n Zeilenumbruch";
```

Listing 1.15: String mit Zeilenumbruch

Eine weitere Möglichkeit, Sonderzeichen darzustellen, existiert über den Unicode. Dies ist ein universeller und in letzter Zeit der am meisten anerkannte Zeichencodierungsstandard. Das Einbetten des Unicodes in Flash geschieht auch über das Fluchtsymbol, gefolgt von einem u und einem 4-stelligen Zahlencode.

Listing 1.16: String mit Unicode-Zeichen

```
var Stringvar;
Stringvar = "\u0041";
```

`Stringvar` würde jetzt ein „A“ enthalten.

Flash MX unterstützt nun den gesamten Zeichensatz des Unicodes (Tabelle siehe Anhang A.1)

Weitere Informationen zum Unicode finden Sie unter:

http://www.unicode.org/unicode/standard/translations/german.html

oder auch direkt unter

http://www.unicode.org/

Weitere Möglichkeiten, Sonderzeichen darzustellen bestehen über die Stringfunktion `String.fromCharCode(Character1, Character2, ..., ...)` oder über die URL-Encoding-Tabelle. Dazu jedoch später mehr unter Strings und im ActionScript-Referenzteil.

Strings verketten

So wie man numerische Werte addieren kann, kann man Strings verketten. Hierbei wird String1 an den String2 angehängt:

Listing 1.17: Verketten von Strings

```
var var1="Hallo";
var var2="Welt";
var var3=var1 + var2;
trace (var3);
```

Es würde der String "`HalloWelt`" auf dem Bildschirm ausgegeben. Dies ist etwas unschön da dieser kein Leerzeichen zwischen den zwei Worten enthält. Sie hätten dies extra mit angeben müssen:

```
var var3 = var1 + " " + var2;
```

Das Leerzeichen muss natürlich auch, da es sich um einen String handelt, in Anführungszeichen gesetzt werden. Wie Sie sehen, kann man Strings auch direkt angeben und muss diese vorher nicht in einer Variable gespeichert haben. Sie können auch Strings und numerische Werte verketten. Flash wandelt dabei automatisch den Wert in einen String um, so wird zum Beispiel aus 4 → „4“.

Listing 1.18: Verketten von Strings und numerischen Werten

```
trace(3+4+"hallo");
trace("hallo"+3+4);
```

Der erste Befehl gibt den String „7hallo“ aus. Der zweite ergibt allerdings „hallo34“ und nicht, wie man vielleicht erwartet, „hallo7“.

Dies liegt daran, dass Flash den Befehl von links nach rechts abarbeitet. Im ersten Fall stößt Flash zuerst auf die Werte 3 und 4, die dann addiert werden sollen. Danach soll an den Wert 7 der String „hallo“ angehängt werden. Flash macht aus dem Wert 7 einen String „7“ und hängt den String „hallo“ an.

Im zweiten Fall jedoch findet Flash zuerst den String und wandelt dementsprechend zuerst den numerischen Wert 3 in einen String, um diesen an „hallo“ anzuhängen. Danach soll an diesen String „hallo3“ noch die Zahl 4 angehängt werden. Flash wandelt den Wert in einen String um und hängt diesen auch an.

Daran erkennt man, dass Flash die Befehle immer von links nach rechts abarbeitet. Wenn Sie dies umgehen möchten, gibt es den Klammeroperator. Wie in der Mathematik, in der schon das Assoziativgesetz gilt, trifft dies in der Programmierung natürlich genauso zu. Wenn man allerdings Strings mit in die „Rechnung“ einbezieht, kann Flash zu einem anderen Ergebnis kommen, da es sich bei dem Pluszeichen hinter dem String für Flash nicht um eine mathematische Operation handelt, sondern um einen Verkettungsbefehl. Dieser Verkettungsbefehl ist auch als `add` bekannt.

So kann man durch die Schreibweise mit Klammern

```
var var3="hallo"+(3+4);
```

erreichen, dass der String „hallo7“ ausgegeben wird.

Deutlicher wird der Unterschied zwischen den Verkettungsoperanden + und dem Befehl `add` an dem folgenden Beispiel:

```
var var3= 2 add 3;
```

Bildschirmausgabe: `23`

Listing 1.19: Stringverkettung mit Hilfe von add

Obwohl es sich bei 2 und 3 um numerische Werte handelt, werden diese trotzdem verkettet, da dies explizit durch den Befehl `add` bewerkstelligt wird.

Stringfunktionen und Eigenschaften

In Flash 4 gab es `substring` und ähnliche Befehle in einer anderen Syntax als in Flash 5, auf diese Befehle wollen wir nicht mehr eingehen. Diese älteren Flash-4-Befehle wurden schon in Flash 5 durch die neueren String-Methoden ersetzt. Diese Methoden haben den Vorteil, dass sie sich auch an die Punktsyntax von Flash halten und so „intuitiver“ angewendet werden können.

In Flash 5 wurde die Stringmethode erweitert, so bekommt man nun in Flash MX für fast jede gewünschte Funktion direkt die passende Methode. Doch was machen Stringmethoden eigentlich? Sie erleichtern uns die Arbeit! Statt selbst zu programmieren, kann man mit einem Methodenaufruf sagen, was man möchte.

Mit der `charAt()`-Methode können Sie sich beispielsweise einen bestimmten Buchstaben aus einem String ausgeben lassen, bzw. diesen abfragen.

String.methode()

`String` ist hierbei die Klasse. Wenn man die Funktion hinterher anwendet, schreibt man nicht `String.charAt()`, sondern ersetzt die Klasse `String` durch das gebildete Objekt von der Klasse `String`. Mehr zu Klassen und Objekten in Kapitel 1.8.

Listing 1.20: Abfrage eines bestimmten Buchstabens innerhalb eines Strings

```
var var1="Zeichenkette";
var2=var1.charAt(7);
```

Die Variable `var2` würde nur die Zeichenkette „k“ enthalten, da dies der Buchstabe an der siebten Stelle des Wortes „Zeichenkette“ ist. Wenn Sie sich selbst von der Funktionsweise der Methode überzeugen wollen, können Sie dies über den `trace()`-Befehl bewerkstelligen:

```
var var1="Zeichenkette";
trace(var1.charAt(7));
```

Hier nun einmal kurz alle Stringmethoden im Überblick:

- String.charAt(index)
- String.charCodeAt(index)
- String.concat(String1, String2,... , ...)
- String.fromCharCode(Character1, Character2, ... , ...)
- String.indexOf(wert, start)
- String.lastIndexOf(wert, start)
- String.slice(start, ende)
- String.split(Trennzeichen)
- String.substr(start, laenge)
- String.substring(start, ende)
- String.toLowerCase()
- String.toUpperCase()

Die Anwendung dieser Methoden erfolgt immer ähnlich. Zuerst kommt der Name des Strings, gefolgt von einem Punkt und dem entsprechenden Methodennamen. Welche Methode genau welche Aufgabe erfüllt, sehen Sie in der Befehlsreferenz zu Flash. Dort sind alle Methoden mit einem zugehörigen Beispiel ausführlich aufgelistet.

Hier sei noch erwähnt, dass jeder String genau eine Eigenschaft besitzt. Dies ist die Länge und sie kann über `String.length` abgefragt werden.

Listing 1.21: Ausgabe der Länge eines Strings

```
var var1="Zeichenkette";
Trace(var1.length);
```

Bildschirmausgabe: `12`

Die Ausgabe unter Flash ist eine 12, da das Wort „Zeichenkette“ genau zwölf Buchstaben besitzt.

Im Folgenden noch ein Praxisbeispiel zu den Stringmethoden. Dies ist etwas anspruchsvoller und setzt gewisse Kenntnisse im Bereich Vergleichsoperatoren und If-Abfragen voraus. Falls Sie diese noch nicht besitzen, überspringen Sie dieses Beispiel, in Kapitel 1.4, Abschnitt

„Verkürzte If-Else-Abfrage“ ab Seite 63 verweisen wir wieder darauf zurück und dann sollte alles für Sie verständlich sein.

Die Datei zum Beispiel finden Sie auf der CD unter *Kapitel1/oo_email.fla*

Es kommt oft vor, dass man einen String auf gewisse Anzeichen untersucht. Sei es bei einem Ratespiel, ob der User die richtige Antwort eingetippt hat, oder ob eine eingegebene E-Mail-Adresse Gültigkeit besitzt. Bei Letzterem müsste man zum Beispiel wissen, an welcher Position sich das @-Zeichen oder der letzte Punkt befindet.

Im Folgenden geht es also um ein Programm, das die eingegebene E-Mail auf die Richtigkeit überprüft. So muss jede E-Mail-Adresse nach einer gewissen Syntax aufgebaut sein.

x@xxx.xx

x steht für ein beliebiges Zeichen.

Dies wäre die kürzeste mögliche E-Mail-Adresse. Daran erkennt man, dass vor dem @-Zeichen mindestens ein Zeichen stehen muss, zwischen @ und Punkt mindestens drei Zeichen und nach dem Punkt entweder zwei oder drei Zeichen.

Dies kontrolliert das Script. Zuerst erstellen Sie einen Button und zwei Textfelder, eines für die Eingabe, das die Variable `email` anspricht, und eines für die Ausgabe mit der Variablen `Ausgabe`.

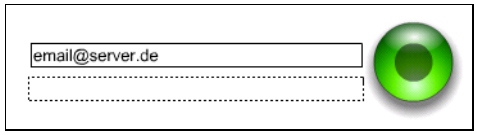


Abbildung 1.7: Layout des E-Mail-Adressen-Validators

Nachdem das Grundgerüst des Programms erstellt wurde, kann man sich nun dem ActionScript widmen. Da das Script auf den Button gelegt wird, muss es in ein Event-Model eingefügt werden. Mehr dazu in Kapitel 2 ab Seite 135. Die folgende Codezeile sorgt dafür, dass beim Drücken des Buttons das Script ausgeführt wird.

```
on (release) {   }
```

Als Erstes sollte man die grundlegenden Informationen, die man vergleichen möchte, aus dem String „herausfiltern“ und in Variablen speichern. Dies ist nicht zwingend notwendig, verschafft aber viel Übersicht!

```
var laenge = email.length;
```

Die Länge des Strings `email` wird ermittelt und in der Variablen `laenge` gespeichert.

```
var PunktPosition=email.lastIndexOf(".") ;
```

Die Position des letzten Punktes im String wird in der Variablen `PunktPosition` gespeichert.

```
var AtPosition=email.indexOf("@")==email.lastIndexOf("@")?
email.indexOf("@"):false ;
```

Dies ist eine geschickte Lösung: Es wird abgefragt ob das erste @-Zeichen von links genau an der gleichen Position ist wie das erste @-Zeichen von rechts. Dadurch wird sichergestellt, dass nur ein @-Zeichen im String existiert. Wenn es nur ein @-Zeichen gibt, ist die Bedingung, `email.indexOf("@")==email.lastIndexOf("@")`, `true` und somit wird `AtPosition`, `email.indexOf("@")`, also die Position des @-Zeichens zugewiesen. Falls zwei @-Zeichen existieren, wird die Bedingung `false` und `AtPosition` wird `false` zugewiesen.

In Kapitel 1.4, Abschnitt „Verkürzte If-Else-Abfrage“ ab Seite 63 gehen wir noch einmal ausführlich auf die verkürzte If-Else-Bedingung ein.

Nachdem alle relevanten Daten des Strings in Variablen zwischengespeichert sind, vergleichen Sie, ob die eingegebenen Daten den gewünschten Anforderungen entsprechen.

x@xxx.xx

Als Erstes wird überprüft, ob sich vor dem @-Zeichen mindestens noch ein Zeichen befindet, dementsprechend vergleicht man, ob die @-Position größer als 1 ist.

```
AtPosition>1
```

Als Nächstes haben wir verglichen, ob sich zwischen der @-Position und dem Punkt mindestens drei Zeichen befinden.

x@**xxx**.xx

```
(AtPosition+3)<PunktPosition
```

Die @-Position + 3 muss also kleiner als die Punktposition sein.

Danach wird die Länge des Strings nach dem Punkt überprüft, dieser muss zwei oder drei Zeichen lang sein, deshalb muss die Punktposition +2 oder +3 der Länge des gesamten Strings entsprechen.

x@xxx.**xx** oder x@xxx.**xxx**

```
((PunktPosition+4) == laenge || (PunktPosition+3) == laenge)
```

Alle diese Vergleiche müssen `true` ergeben, damit die E-Mail-Adresse gültig ist.

Dies müssen Sie dann durch das logische UND zusammenfassen, und mit einer If-Abfrage dem Ausgabe-Textfeld ein Ergebnis zuweisen. Ingesamt sieht der Code für den Button dann wie folgt aus:

```
on (release) {
    var laenge = email.length;
    var AtPosition=email.indexOf("@")
        ==email.lastIndexOf("@")? email.indexOf("@"):false ;
    var PunktPosition=email.lastIndexOf(".") ;
    if (AtPosition>1 && (AtPosition+3)<PunktPosition
        && ((PunktPosition+4) == laenge || (PunktPosition+3)
        == laenge)) {
        Ausgabe = "Eingabe gültig";
    } else {
        Ausgabe = "Ungültige E-Mail-Adresse";
    }
}
```

Listing 1.22: Das ActionScript, um eine E-Mail-Adresse auf bestimmte Formalien zu überprüfen

▶ Boolean

Der Typ `Boolean` besitzt nur zwei zulässige Werte:

- true (Wahr)
- false (Falsch)

Wenn eine Variable von Typ `Boolean` direkt deklariert wird, wird dieser automatisch der Wert `false` zugewiesen.

```
var var1=Boolean(var1);
trace(var1);
```

Listing 1.23: Explizite Deklaration einer Boolean-Variable

Bildschirmausgabe: `false`

Diesen Typ benutzt man, um einfache Optionen abzufragen, z.B. wenn etwas nur zwei Zustände hat, wie eine Tür, die offen oder geschlossen sein kann.

```
var tuer1=false;
var tuer2=false;
var tuer3=false;
```

Listing 1.24: Drei Variablen werden deklariert und Booleanwerte zugewiesen.

Hier im Beispiel werden drei Variablen initialisiert. Diese sollen den Zustand von drei Türen beschreiben. Im Anfangszustand sind alle Türen geschlossen, dementsprechend sind alle Variablen `false`. Wenn man nun die erste Tür öffnet, muss die Variable `tuer1` den Wert `true` annehmen.

Bei der hardwarenahen Programmierung spricht man bei Variablen, die nur zwei Zustände anzeigen (z.B. dass beim Addieren der Speicher übergelaufen ist) von Flags. Vergleichen kann man die Booleanvariablen noch mit Jumpern bzw. Dip-Schaltern auf dem Motherboard. Auch diese können jeweils nur einen Zustand von zwei möglichen haben: offen und geschlossen, an oder aus, – 1 oder 0.

Sicherlich braucht man diese Art Variable nicht oft, aber indirekt kommt sie bei jeder Vergleichsoperation vor. Mehr dazu in Kapitel 1.4, Abschnitt „If-Else-Abfrage“ ab Seite 62.

▶ Undefined

Anders als `Number`, `String` oder `Boolean` besitzt `Undefined` nicht mehrere Werte. Eine Variable vom Typ `undefined` enthält auch automatisch diesen Wert, der besagt, dass eine Variable deklariert wurde, ihr aber kein Wert zugewiesen ist. Da in Flash keine Notwendigkeit besteht, Variablen zu deklarieren, besitzen demzufolge auch alle nicht deklarierten Variablen den Typ `undefined`.

Listing 1.25: Ausgabe von undefined

```
var var1;
trace(var1);
trace(var2);
```

Ausgabe in Flash MX

```
undefined
undefined
```

In Flash 5 war das Ausgabefenster hingegen noch leer.

Durch diesen Unterschied können aber auch Probleme zwischen Flash MX und Flash 4 auftreten. Da in Flash 4 kein `undefined` existierte, wurde dort immer auf "" (einen leeren String) geprüft. Diese Art der Prüfung funktioniert dementsprechend in Flash MX nicht mehr. In Flash 5 war dies jedoch noch möglich, da dort aus Abwärtskompatibilitätsgründen deklarierte Variablen nicht direkt mit dem Wert `undefined` belegt wurden, obwohl dies damals auch schon so vorgesehen war. Wer also noch ein älteres Flash-4-Script besitzt oder in Flash 5 noch die alte Programmierweise benutzt hat, sollte auf diese Änderung achten!

▶ Null

Genauso wie `Undefined` enthält auch `null` nur einen zulässigen Wert, nämlich sich selbst.

`Null` und `undefined` entsprechen sich. So ergibt ein Abfrage auf Gleichheit `true`.

Der Unterschied zu `undefined` besteht darin, dass `null` explizit einer Variablen zugewiesen werden muss. `Null` symbolisiert einen Sonderwert, den man einer Variablen zuweist, wenn eine spezielle Ausnahmebehandlung ausgeführt werden soll. Sicherlich kann man auch einen speziellen Wert zuweisen und sagen, wenn die Variable den Wert `xyz` hat, mache dies und das, aber genau für diesen Fall existiert `null` – sozusagen als standardisierter Sonderwert. Außerdem steckt hinter `null` noch mehr: In gängigen Programmiersprachen kann man jeder Variablen nur bestimmte Typen zuweisen. So kann man einer Variablen die auf ein Array verweist, ausschließlich Verweise auf Arrays zuweisen. Die Ausnahme bildet `null`. In den gängigen Programmiersprachen weist man, wenn ein Array oder ein Objekt gelöscht wird, man die Referenzvariable aber noch braucht, dieser den Wert `null` zu. Diese gängige Praxis sollte man auch in Flash so beibehalten, auch wenn dies durch die Syntax von

Flash nicht zwingend vorgegeben ist, da Flash die Variablen-Typen zwar unterscheidet, aber immer automatisch castet (den richtigen Typ zuweist).

`Null` wird zudem noch dafür benutzt, um existierende Funktionen zu löschen. Falls man also einem *enterFrame-Event* einen Inhalt zugewiesen hat und diesen wieder löschen will, weist man diesem `null` zu.

```
Moviename.onEnterFrame=null;
```

Listing 1.26: Löschen des Inhalts eines EnterFrame-Events

Mehr zu Funktionsaufrufen und dem Event-Model finden Sie in den Kapiteln 1.7, 1.8 und 2.

Typenwandlung (Casting)

Wie schon angedeutet, weist Flash Variablen immer automatisch den richtigen Typ zu, so dass man nie direkt darauf Einfluss nehmen muss. Trotzdem ist es natürlich auch möglich, eine Typenwandlung zu erzwingen. Dadurch kann man erreichen, dass statt zu addieren verkettet wird. Oder dass man mit einem String, der eine Zahl enthält, rechnen kann.

```
var1=3;
var2=4;
var3 = var1+var2;
trace(var3);

var3 = String(var1)+var2;
trace(var3);
```

Listing 1.27: Typenwandlung von Number zu String

Bildschirmausgabe:

```
7
34
```

Im ersten Beispiel addiert Flash die Werte ganz normal, beim zweiten jedoch wurde `var1` zu einem String gewandelt, so dass dieser nun verkettet wird. Eigentlich ist dies etwas unsauber, denn die erste Variable wird strikt durch `String()` gecastet und die zweite Variable muss Flash automatisch zu einem String casten. Wenn man schon auf die cast-Operatoren zurückgreift, kann man diese auch direkt überall einsetzen.

```
var3 = String(var1)+String(var2);
```

Natürlich hätte man in dem Fall auch `add` statt + als Operator benutzen können. Flash hätte für `add` dann wieder automatisch die Typenwandlung vorgenommen. Genauso muss man das Ergebnis nicht immer in einer Variablen zwischenspeichern, man hätte es auch direkt mit in den Ausgabe-Befehl schreiben können.

```
trace(String(var1)+var2);
```

Das Vorangegangene sollte der besseren Übersicht dienen. Die Umwandlung von String zu Number oder Boolean erfolgt genauso. Erwähnenswert ist, dass nur `0` `false` entspricht und jeder andere Wert `true`. Dies ist so festgelegt!

Listing 1.28: Umwandlung des Wertes 3 zu Boolean

```
var1=3;
var3 = Boolean(var1);
trace(var3);
```

Bildschirmausgabe: `true`

Listing 1.29: Umwandlung des Wertes -3 zu Boolean

```
var1=-3;
var3 = Boolean(var1);
trace(var3);
```

Bildschirmausgabe: `true`

Listing 1.30: Umwandlung des Wertes 0 zu Boolean

```
var1=0;
var3 = Boolean(var1);
trace(var3);
```

Bildschirmausgabe: `false`

Andersherum werden Booleanvariablen in 0 und 1 gewandelt. Alle Zeichenketten werden zu Number gewandelt. Dadurch wird aus den meisten Werten `NaN` (ein Sonderwert für Variablen vom Typ `Number`), was auch wieder `false` ergibt.

Bei Strings, die zu `Number` gewandelt werden, werden alle zulässigen Numberformate erkannt. Falls der String diesen entspricht, werden Sie auch umgewandelt. Falls es kein zulässiges Numberformat ist, beinhaltet die anschließende Variable von Typ `Number` den Wert `NaN`.

Listing 1.31: Umwandlung eines Strings, der einen Wert enthält, zu Number

```
var1="345.78";
var3 = Number(var1);
trace(var3);
```

Bildschirmausgabe: `345.78`

Listing 1.32: Umwandlung eines Strings zu Number

```
var1="345,-";
var3 = Number(var1);
trace(var3);
```

Bildschirmausgabe: `NaN`

1.3 Operatoren

Operatoren haben Sie bereits in den vorangegangenen Beispielen kennen gelernt. Operatoren dienen als Bindeglied zwischen zwei Variablen, sie geben Aufschluss darüber, was mit den zwei Variablen geschehen soll. Meistens beziehen sich Operatoren auf zwei Variablen. Dies ist allerdings nicht immer der Fall: So gibt es beispielsweise den Punktoperator, um auf den Inhalt eines Objektes zuzugreifen. Oder das Ausrufezeichen als logischen Negator, um aus `true` – `false` und umgekehrt zu erzeugen. Dabei benutzt man jeweils nur eine Variable. Jeder Operator besitzt eine gewisse Wertigkeit. So wie in der Mathematik „Punkt vor Strich“ gilt, existiert auch hier eine Rangreihenfolge. Diese können Sie der Tabelle mit den *Wertigkeiten der Operatoren* in Anhang A entnehmen. Grundsätzlich lassen sich die Operatoren entsprechend ihrer Aufgabenbereiche gliedern. Als Erstes gehen wir auf die numerischen Operatoren ein, welche zum Rechnen benutzt werden.

Numerische (arithmetische) Operatoren

Zu den numerischen Operatoren gehören:

- + Additionsoperator
- - Subtraktionsoperator
- * Multiplikationsoperator
- / Divisionsoperator
- % Modulooperator
- - Negationsoperator

▶ Der Additionsoperator +

Wie in den obigen Beispielen angedeutet, dient dieser Operand der Addition von zwei Numbervariablen oder zum Verketten von Strings. Über die Addition von Zahlen finden Sie genug in Kapitel 1.2, Abschnitt „Numerisch“ ab Seite 19.

Es gibt auch eine verkürzte Schreibweise, um eine Variable um 1 zu erhöhen. So kann man statt

```
var1 = var1+1;
```

auch

```
var1 ++;
```

schreiben. Beide Befehle haben den gleichen Effekt und sorgen dafür, dass der Inhalt von `var1` um 1 erhöht wird. Bei dieser Schreibweise mit dem Inkrement gibt es zwei Möglichkeiten: einmal als Prä-Inkrement

(++var1;) und als Post-Inkrement (var1++;). Dazu jedoch später mehr in Kapitel 1.3, Abschnitt „Zuweisungsoperatoren“ ab Seite 40.

- **Der Subtraktionsoperator -**

Anders als sein Gegenstück, der Additionsoperand, lässt sich der Subtraktionsoperator nur auf Variablen vom Typ Number anwenden bzw. er castet alle anderen Typen automatisch zu Number. Dementsprechend kann man mit dem Minusoperanden auch nur das machen, was man eigentlich erwartet, nämlich Numberwerte subtrahieren.

Listing 1.33: Anwendung des Minusoperator

```
trace(10-4);
```

Bildschirmausgabe: 6

Auch bei dem Subtraktionsoperator gibt es wieder eine verkürzte Schreibweise, um eine Variable um den Wert 1 zu verringern. So kann man statt

```
var1 = var1-1;
```

auch

```
var1 --;
```

schreiben. Beide Befehle haben den gleichen Effekt und bewirken, dass der Inhalt von var1 um 1 verringert wird. Bei dieser Dekrement-Schreibweise gibt es zwei Möglichkeiten. Einmal als Prä-Dekrement (--var1;) und als Post-Dekrement (var1--;). Dazu jedoch später mehr in Kapitel 1.3, Abschnitt „Zuweisungsoperatoren“ ab Seite 40.

- **Multiplikationsoperator * und Divisionsoperator /**

Die Anwendung des Multiplikationsoperators und des Divisionsoperators verstehen sich analog.

Listing 1.34: Anwendung des Multiplikations- und Divisionsoperators

```
trace(10*5/2)
```

Bildschirmausgabe: 25

Man beachte hierbei auch die Notation für den Divisionsoperanden: ein Querstrich (Slash) / und kein Doppelpunkt (:). Der Doppelpunkt wird unter anderem dazu benötigt, um bei Objekten eine Initialisierung, sprich eine Deklaration, direkt mit einer Wertezuweisung zu ermöglichen. Dazu jedoch später mehr.

Kommen wir nun noch einmal auf die numerischen Operatoren zurück: Alle bis auf den Additionsoperator haben die Eigenschaft, automatisch Strings in Number zu casten. Wenn Sie zum Beispiel versuchen, den Minusoperator – auf einen String anzuwenden, würde Flash dem Ergebnis NaN zuweisen, vorausgesetzt der String enthält kein gültiges Numberformat.

Der Additionsoperand bildet deshalb eine Ausnahme, weil er der einzige überladene Operator ist, das heißt, er ist der einzige, der bei unterschiedlichen Typen von Variablen unterschiedliche Aktionen ausführt.

Zum einen wird das Pluszeichen zum Addieren von zwei Zahlen verwendet (Additionsoperand), zum anderen zum Verketten von zwei Strings (Verkettungsoperand).

▶ Der Modulooperator %

Es handelt sich bei diesem Operator um keine der gebräuchlichen vier Rechenzeichen und dennoch ist der Modulooperator sehr nützlich. Er dividiert den ersten Wert durch den zweiten und gibt als Ergebnis den Rest aus.

```
trace(11%4);
```

Bildschirmausgabe: `3`

Listing 1.35: Anwendung des Modulooperators

Es würde eine 3 ausgegeben, da 11 - (2*4) = 3 ergibt. Vielleicht ist es auch anschaulicher zu sagen: 11 durch 4 ergibt 2, Rest 3.

Um zu überprüfen, ob eine Zahl ungerade oder gerade ist, ist dieser Operator nützlich.

```
var var1=5;
trace(var1%2);
```

Bildschirmausgabe: `1`

Listing 1.36: Überprüfen einer Zahl auf gerade oder ungerade

Die Variable `var1` dient hier nur als Platzhalter. Indem man eine Zahl mit Modulo auf die Teilbarkeit durch 2 überprüft, kann man prüfen, ob die Zahl gerade (Ergebnis 0) oder ungerade (Ergebnis 1) ist.

▶ Der Negationsoperator -

Dieser Operand dient dazu, bei der entsprechenden Variablen einen Vorzeichenwechsel herbeizuführen.

```
var var1=5;
trace(-var1);
```

Bildschirmausgabe: `-5`

Listing 1.37: Negation einer Variablen

Im Vergleich zu den vorangegangenen Operatoren bezieht sich dieser nur auf eine Variable (und nicht auf Zwei)! Es wird nur das Vorzeichen geändert. Mathematisch entspricht dies der Multiplikation mit –1.

String-Operatoren

Es existiert nur ein einziger String-Operator und diesen kennen Sie bereits aus dem Einführungsteil.

▶ Der Verkettungsoperand +

Es handelt sich beim Verkettungsoperanden um das Pluszeichen (+). Mit diesem können Sie zwei Strings verketten (aneinander hängen, zusammenfügen).

Listing 1.38: Ausgabe von zwei verketteten Strings

```
var1 = "Hallo" + " " + "Welt";
trace(var1);
```

Bildschirmausgabe: `Hallo Welt`

Wie schon erwähnt ist das Pluszeichen überladen durch die Funktion als Additions- und Verkettungsoperand. Welche Funktion angewandt wird, hängt von den bestehenden Variablen ab. Wenn eine Variable den Typ `String` besitzt, wird die andere Variable auch zu `String` gecastet. Weitere Beispiele zum Verketten von Strings, oder zum Casten von Variablen finden Sie in Kapitel 1.2 in den Abschnitten „Strings verketten“ ab Seite 28 und „Typenwandlung (Casting)“ ab Seite 35.

Zuweisungsoperatoren

Einen Zuweisungs- bzw. Zuordnungs-Operator haben Sie bereits in den vorangegangenen Beispielen benutzt: das Gleichheitszeichen (=). Mit diesem Operator weisen Sie der links stehenden Variable den Inhalt des rechten Ausdruckes bzw. der dort stehenden Variable zu. Zuweisungsoperatoren werden also immer dazu benutzt, um einer Variablen einen Wert zuzuweisen. Ob es sich dabei um ein neu berechnetes Ergebnis handelt oder um einen Wert, der schon in einer anderen Variable steht, ist erst einmal irrelevant. Wichtig ist zu wissen, dass erst durch die Zuweisungsoperatoren Werte in Variablen gespeichert werden. Jedes Mal wenn Sie einen Zuweisungsoperator in einem Skript sehen, wird der davon links stehenden Variablen der Wert des rechten Ausdruckes zugeordnet.

▶ Einfache Zuweisung =

Die Anzahl der zur Verfügung stehenden Zuweisungsoperatoren scheint zwar groß, aber im Grunde agieren alle sehr ähnlich. Der Vater aller Zuweisungsoperatoren ist das Gleichheitszeichen (=). Hier noch einmal ein simples Beispiel:

```
var1 = 4;
var1 = var1 / 2 ;
```

Listing 1.39: Beispiel für eine Wertezuweisung

Der Variablen `var1` wird durch den Zuordnungsoperator (=) der Wert 4 zugewiesen. Anschließend wird der Variablen `var1` der durch 2 geteilte Wert der Variablen `var1` zugewiesen, sprich der Inhalt der Variable `var1` wird durch 2 geteilt. So enthält am Ende die Variablen `var1` den Wert 2.

▶ Erweiterte Zuweisung

Vielleicht ist es Ihnen bei dem oberen Beispiel direkt aufgefallen. Wir wollten dort den gespeicherten Wert in der Variable `var1` einfach nur durch 2 teilen und haben dies mit folgendem Code bewerkstelligt:

```
var1 = var1 / 2 ;
```

Die Lösung ist relativ umständlich, denn man möchte nur den Wert einer Variablen durch 2 teilen und nicht extra schreiben müssen: Teile die Variable `var1` durch 2 und weise das Ergebnis `var1` zu. Um den Inhalt einer Variablen direkt zu verändern bzw. den Wert zu ändern und automatisch das Ergebnis wieder der Variable zuzuweisen, gibt es Zuweisungsoperatoren, die alle nach demselben Schema aufgebaut sind. So kann man statt

```
var1 = var1 / 2 ;
```

auch

```
var1 /= 2 ;
```

schreiben.

Beide Codezeilen erzielen denselben Effekt. Diese verkürzte Schreibweise gibt es nicht nur für das Dividieren, sondern auch für alle anderen Operatoren. Man schreibt den Operator, dessen Funktion man nutzen möchte, vor das Gleichheitszeichen und dahinter die Zahl, mit der gerechnet werden soll.

Die einzige Ausnahme bildet hier der Negationsoperand, da dieser auch doppelt belegt ist und beide Male nur für Numbervariablen zuständig ist.

Wenn Sie nur das Vorzeichen einer Variablen umkehren wollen, geht dies nicht über

```
var1 -=  ;
```

Das sieht ungewöhnlich aus, da hinter dem Gleichheitszeichen der Wert fehlt. Wenn Sie dort einfach eine 1 einsetzen

```
var1 -= 1;
```

würden Sie den Inhalt der Variable `var1` um 1 verringern. Um einfach nur das Vorzeichen einer Variable zu ändern, müssen Sie diese mit –1 multiplizieren.

Die kürzeste Schreibweise dafür ist folgende:

Listing 1.40: Umkehrung des Vorzeichens einer Variablen

```
var1 *= -1;
```

Wenn die Variable `var1` also vorher den Wert 4 hatte, hätte sie danach den Wert –4.

Alternativ wäre auch

Listing 1.41: Umkehrung des Vorzeichens einer Variablen

```
var1 = -(var1);
```

möglich. Diese Lösung benutzt keinen erweiterten Zuweisungsoperator.

Im Folgenden noch einmal alle Zuweisungsoperatoren. Die Funktionsweise des eigentlichen Operators entnehmen Sie bitte den Abschnitten „Numerische (arithmetische) Operatoren“ ab Seite 37, „String-Operatoren“ ab Seite 40 und „Bitweise Operatoren“ ab Seite 47.

Zuweisungsoperator	Funktion
=	Zuweisung
+=	Zuweisung mit Addition
-=	Zuweisung mit Subtraktion
*=	Zuweisung mit Multiplikation
/=	Zuweisung mit Division
%=	Zuweisung mit Modulo
&=	Zuweisung mit Bitweise Und
|=	Zuweisung mit Bitweise Oder
^=	Zuweisung mit Bitweise Xor (Antivalenz)
<<=	Zuweisung mit Bitweise Verschiebung nach links
>>=	Zuweisung mit Bitweise Verschiebung nach rechts
<<<=	Zuweisung mit Bitweise vorzeichenlose Verschiebung nach links

Tabelle 1.2: Zuweisungsoperatoren

Vergleichsoperatoren

Mit Hilfe der Vergleichsoperatoren kann man zwei Variablen vergleichen. Als Ergebnis liefern diese Operatoren immer einen Booleschen Wert zurück, also `true` (wahr) oder `false` (falsch). Boolesche Werte siehe auch Kapitel 1.2, Abschnitt „Boolean“ ab Seite 33.

Benutzt werden diese Abfragen dann später meistens im Zusammenhang mit Schleifen oder Abfragen, dazu jedoch später mehr.

▶ Gleichheit ==

Die einfachste Weise etwas zu vergleichen, ist es, zwei Werte auf Gleichheit zu überprüfen. Dazu bedient man sich zweier Gleichheitszeichen. Man achte auf zwei Gleichheitszeichen, da sonst bei einem Gleichheitszeichen eine Zuweisung erfolgt und Flash auch keine Fehlermeldung ausgibt, da alles syntaktisch richtig ist! Die Flash-4-User unter Ihnen werden sich noch teilweise schmerzhaft an den Umstieg zu Flash 5 erinnern, da in Flash 4 nach Definition nur ein Gleichheitszeichen zum Vergleich genutzt wurde!

```
var1 = "Hallo Welt";
var2 = "Hallo Welt";
trace( var1 == var2 );
```

Listing 1.42: Prüft zwei Variablen auf Gleichheit

Bildschirmausgabe: `true`

Sie können nicht nur Werte vergleichen, sondern auch Strings oder sonstige Variablentypen.

Es wird auf „einfache Gleichheit“ geprüft. Sprich, es ist irrelevant, von welchem Typ die Variable ist. Falls nötig, wird gecastet, so entspricht der String „5“ dem Wert 5, wie das folgende Beispiel verdeutlicht.

```
trace("5" == 5);
```

Listing 1.43: Vergleicht einen String mit einem Wert

Bildschirmausgabe: `true`

▶ Strikte Gleichheit ===

Um zu verhindern, dass eine Zahl automatisch gecastet wird, nutzt man diesen Operator. Falls es wirklich erforderlich sein sollte, auch auf denselben Typ der Variable zu prüfen, bietet Flash MX den Operand === für strikte Gleichheit an.

```
trace("5" === 5);
```

Listing 1.44: Vergleicht einen String mit einem Wert

Bildschirmausgabe: `false`

▶ Ungleich !=

Dinge können auch auf Ungleichheit geprüft werden.

```
var1 = "Hallo";
var2 = "Welt";
trace( var1 != var2 );
```

Listing 1.45: Vergleicht, ob die zwei Variablen ungleich sind und liefert ein positives Ergebnis

Bildschirmausgabe: `true`

Als Ergebnis wird true geliefert, da der String „Hallo“ nicht dem String „Welt“ entspricht.

Listing 1.46: Vergleicht einen String mit einem Wert

```
var1 = "4";
var2 = 4;
trace( var1 != var2 );
```

Bildschirmausgabe: `false`

Hier erkennt man wieder, dass der Vergleich typen-unabhängig ist und dass automatisch intern gecastet wird. Im Folgenden wieder der strikte Vergleich.

- **Strikt Ungleich !==**

Wie schon zuvor bei der Überprüfung auf Gleichheit, existiert auch für Ungleich ein typenabhängiger Operator.

```
var1 = "4";
var2 = 4;
trace( var1 !== var2 );
```

Bildschirmausgabe: `true`

Obwohl beide Variablen eine 4 enthalten, ist das Ergebnis, dass die Variablen ungleich sind, da `var1` vom Typ String und `var2` vom Typ Number ist.

- **kleiner als <**

Mit diesem Operator überprüfen Sie, ob eine Variable kleiner ist als die andere. Dies ist nicht auf Werte beschränkt, sondern die Abfrage geht auch mit Strings. Bei Strings wird die Größe des ASCII-Zeichens verglichen. Die Zeichen werden dabei von links nach rechts verglichen, bei Gleichheit werden die nachfolgenden Zeichen verglichen. So ist

```
"aa" < "ab"
```

`true`, aber

```
trace("aa" < "aa");
```

nicht, da beide Strings gleich groß sind.

Zahlenvergleiche sind natürlich auch möglich:

Listing 1.47: Vergleicht, ob 5 kleiner ist als 6

```
trace(5 < 6);
```

Bildschirmausgabe: `true`

Bei gemischten Vergleichen mit Strings und Werten wird versucht, den String in eine Zahl zu casten. Falls dies nicht möglich ist, ergibt der gesamte Term undefined.

- **Größer als >**

Dieser Operator ist das Gegenstück zu dem „Kleiner als“-Operator. Er vergleicht, ob der linke Ausdruck größer ist als der rechte.

```
trace (5 > 6)
```

Bildschirmausgabe: `false`

Listing 1.48: Vergleicht, ob 5 größer ist als 6

- **Kleiner gleich <=**

Damit überprüfen Sie, ob eine Zahl kleiner oder gleichgroß einer anderen Zahl ist. Um nicht prüfen zu müssen, ob diese Zahl garantiert größer ist und das Ergebnis dann negieren zu müssen, gibt es den „Kleiner gleich“-Operanden.

Er wird angewendet wie die anderen Operanden auch.

```
trace (5 <= 5)
```

Bildschirmausgabe: `true`

Listing 1.49: Vergleicht, ob 5 kleiner oder gleich 5 ist

- **Größer gleich >=**

Dieser Operator ist das Gegenstück zu dem „Kleiner gleich“-Operator. Er vergleicht, ob der linke Ausdruck größer oder gleich groß dem rechten Ausdruck ist.

```
trace (5 >= 5);
```

Bildschirmausgabe: `true`

Listing 1.50: Prüft, ob 5 größer oder gleich groß 5 ist

Logische Operatoren

Die logischen Operatoren stellen eine Erweiterung der Vergleichsoperatoren dar. Während man mit den Vergleichsoperatoren immer nur zwei Variablen vergleichen kann, kann man mit Hilfe der logischen Operatoren diese Abfrage erweitern.

Logische Operatoren vergleichen immer zwei Boolean-Werte. Da Vergleichsoperatoren als Ergebnis Boolean-Werte liefern, kann man diese Abfragen also gut kombinieren und so mehrere Dinge in einer Abfrage prüfen!

So kann man mit den logischen Operatoren z.B. folgende Abfrage realisieren:

„Gib true aus, wenn Vergleich1 und Vergleich2 wahr sind.“

Insgesamt gibt es genau drei logische Operatoren:

- && Und
- || Oder
- ! Nicht

Genauso wie in der Mathematik „Punkt vor Strich“ gilt, lautet die Regel hier „UND vor ODER“. Der logische Negator – das Ausrufezeichen (!) – hat allerdings die höchste Priorität.

▶ Der UND-Operator &&

Um zwei logische Ausdrücke zusammenzufassen und zu überprüfen, ob beide wahr sind, nutzt man den Und-Operator. Dies könnte z.B. wie folgt aussehen:

Listing 1.51: Anwendung des UND-Operators

```
trace( (3<5) && (5>2) );
```

Bildschirmausgabe: `true`

Die einzelnen Ausdrücke `(3<5)` und `(5>2)` müssen nicht zwingend geklammert sein. Da beide Ausdrücke wahr sind, ergibt die Abfrage insgesamt auch true.

Gesprochen hört sich der Code wie folgt an: „Wenn 3 kleiner 5 und 5 größer als 2 ist, gib true aus, ansonsten false“. Hier ein kurzer Überblick der Logik:

Wert1 && Wert2

Wert 1	Wert 2	Ergebnis
true	true	true
true	false	false
false	true	false
false	false	false

Tabelle 1.3: Wertetabelle des logischen UND

▶ Der Oder-Operator ||

Wie schon der UND-Operator verbindet auch der ODER-Operator zwei logische Ausdrücke miteinander. Im Gegensatz zum UND-Operator wird das Ergebnis beim ODER-Operator true, sobald einer der zwei Ausdrücke wahr ist.

Listing 1.52: Anwendung des ODER-Operators

```
trace(  (3<5) || (1>2) );
```

Bildschirmausgabe: `true`

Wie man sieht, ist der rechte Ausdruck nicht wahr. Da aber der linke Ausdruck wahr ist, reicht dies, um insgesamt ein `true` zu erzeugen.

Man spricht: „ Wenn 3 kleiner 5 oder 1 größer 2, gib true aus, ansonsten false.“

Hier noch einmal eine kleine Übersichtstabelle, vergleichen Sie die Ergebnisse mit denen des UND-Operators, dann wird Ihnen der Unterschied schnell bewusst.

Wert1 || Wert2

Wert 1	Wert 2	Ergebnis
true	true	true
true	false	true
false	true	true
false	false	false

Tabelle 1.4: Wertetabelle des logischen ODER

- **Das logische NOT (NICHT) !**

Während der UND- oder der ODER-Operator immer zwei Ausdrücke verbinden, bezieht sich das logische NICHT auf nur einen Ausdruck. Das Ausrufezeichen (!) setzt man vor den jeweiligen Ausdruck, und so wird aus einem `true` ein `false` bzw. aus einem `false` ein `true`.

```
trace( !(3<4) );
```

Hierbei ist die Klammerung des eigentlichen Ausdrucks wichtig, da der NICHT-Operator eine höhere Priorität besitzt als das „Kleiner-als“. Deshalb muss man klammern, damit das Ergebnis des Vergleichs 3 < 4 negiert wird und nicht die 3 (was nicht direkt möglich ist). Aber wie schon erwähnt, sieht Flash alle Zahlen außer 0 als `true` an! Boolesche Variablen siehe auch Kapitel 1.2, Abschnitt „Boolean“ ab Seite 33.

! Wert

Wert	Ergebnis
true	false
false	true

Tabelle 1.5: Anwendung des NICHT-Operators

Bitweise Operatoren

Wie schon im Kapitel 1.2, Abschnitt „Numerisch“ ab Seite 19 erwähnt, existieren verschiedene Arten der Zahlendarstellung. Intern arbeiten alle Computer mit Bits (1 und 0), also einem Wert, der nur zwei Zustände kennt. Flash bietet mit den bitweisen Operatoren die Möglichkeit, auch direkt damit zu rechnen. Leider bringt dies in Flash – anders als bei gängigen Programmiersprachen – keinen Geschwindigkeitsvorteil.

Beispielsweise kann man, anstatt eine Zahl mit 2 zu multiplizieren, die Bits der Zahl um zwei Stellen nach links verschieben.

Hier kurz das Beispiel – lassen Sie sich von dem ersten etwas komplexeren Script nicht gleich abschrecken:

Listing 1.53: Beweise dafür, dass Bitshifting in Flash nicht schneller als eine normale Multiplikation ist

```
zeit1=0;
zeit2=0;
wert=10;
st=getTimer();
for(i=0; i< 99999;i++)
e=wert*4;
zeit1=getTimer()-st;
st=getTimer();
for(i=0; i< 99999;i++)
e2=wert<<2;
zeit2=getTimer()-st;
trace("Ergebnis der Rechnung: "+ e +" "+ e2);
trace("Die Multiplikation dauerte: "+zeit1+ "\n
Das bitweise Verschieben benötigte: "+zeit2  );
```

Bildschirmausgabe:

```
Ergebnis der Rechnung: 40 40
Die Multiplikation dauerte: 2979
Das bitweise Verschieben benötigte: 3065
```

Das erste für Sie noch unbekannte Element dürfte die Funktion `getTimer()` sein. Sie liefert die vergangene Zeit seit Start des Movies. Lassen Sie sich nicht von der ungewohnten Schreibweise mit den Klammern irritieren. Die vergangene Zeit seit Beginn des Movies (bis zum Aufruf der Funktion `getTimer()`) wird in der Variablen `st` gespeichert. Die for-Schleife überspringen wir an dieser Stelle. Sie ist für den Gesamtzusammenhang irrelevant und wird nur benötigt, um überhaupt ein messbares Ergebnis zu erzielen. Mehr zu Schleifen erfahren Sie in Kapitel 1.6 ab Seite 81.

Es geht in dem Script um die Messung der Zeit, die Flash benötigt, um den Befehl e=wert*4; bzw. e2=wert<<2; auszuführen.

Dementsprechend wird – bevor der Befehl (mehrmals durch die for-Schleife) ausgeführt wird – die Zeit gestoppt bzw. der Zeitpunkt des Starts in der Variablen `st` festgehalten. st=getTimer();

Nachdem der jeweilige Befehl (e=wert*4; bzw. e2=wert<<2;) ausgeführt wurde, wird die Differenz des aktuellen Zeitpunktes zum Startzeitpunkt gebildet und in der jeweiligen Variablen `zeit1` bzw. `zeit2` gespeichert.

```
zeit1=getTimer()-st;
```

bzw.

```
zeit2=getTimer()-st;
```

Nach den Berechnungen erfolgt die Ausgabe des Ergebnisses durch `trace()`. Dies zeigt dann, dass die bitweisen Operatoren nicht schneller sind:

```
Ergebnis der Rechnung: 40 40
Die Multiplikation dauerte: 2979
Das bitweise Verschieben benötigte: 3065
```

Durch die erste Ausgabe sollte nur sichergestellt werden, dass beide Operationen auch wirklich das gleiche Ergebnis liefern. An der zweiten Ausgabe erkennt man, dass beide Berechnungen (fast) die gleiche Zeit benötigen.

Noch ein zweites Beispiel, welches sich folgende Beziehung zu Nutze macht:

`Math.pow(2,10);` entspricht `(2<<9);`

2^{10} entspricht dem bitweise Verschieben der Bits der Ziffer 2 um neun Stellen nach links.

Listing 1.54: Weiterer Geschwindigkeitsvergleich zwischen Bitshifting und normaler Berechnung

```
zeit1=0;
zeit2=0;
st=getTimer();
for(i=0; i< 99999;i++)
e=Math.pow(2,10);
zeit1=getTimer()-st;
st=getTimer();
for(i=0; i< 99999;i++)
e2=(2<<9);
zeit2=getTimer()-st;
trace("Ergebnis der Rechnung: "+e +" "+ e2);
trace("Die normale Berechnung dauerte: "+zeit1+ "\n
Das bitweise Verschieben benötigte: "+zeit2  );
```

Bildschirmausgabe:

```
Ergebnis der Rechnung: 1024 1024
Die normale Berechnung dauerte: 2449
Das bitweise Verschieben benötigte: 2476
```

Für Berechnungen bringt das Bitshifting, wie die Beispiele zeigen, keinen Vorteil. Wenn man aber bei Berechnungen/Vergleichen nur boolesche Werte benutzt, also die Zuständen `true` oder `false`, kann man sich die Arbeit vereinfachen.

Schauen Sie sich noch einmal das schon im Kapitel 1.2, Abschnitt „Boolean“ ab Seite 33 vorgestellte Beispiel mit den Türen genau an. Hier leicht erweitert durch eine Tür:

```
var tuer1=false;
var tuer2=false;
var tuer3=false;
var tuer4=false;
```

Es ist nicht unbedingt notwendig, für jede Tür eine Variable zu benutzen. Man kann alle Zustände der Türen auch in Variablen speichern. Erinnern Sie sich noch an die Binärschreibweise von Zahlen. Jede Zahl konnte genau zwei Zustände annehmen, genauso wie der Typ Boolean oder hier im Beispiel unsere Türen.

Man schafft sich eine Variable vom Typ `Number`. Man definiert, dass alle Türen geschlossen sind, wenn diese Variable den Wert 0 besitzt. Beim Wert 1 ist nur die erste Tür auf. Beim Wert 2 nur die zweite. Beim Wert 3 nur die erste und die zweite usw.

Sie haben den Zusammenhang dieser kleinen Reihe nicht gefunden? Es ist einfacher als es scheint, die Türen werden einfach im Binärsystem codiert und dementsprechend hochgezählt. Die nachfolgende Tabelle sollte alle Unklarheiten beseitigen (0 entspricht geschlossen, 1 entspricht offen).

Tür 4	Tür 3	Tür 2	Tür 1	Binärzahl	Dezimalwert
0	0	0	0	0000	0
0	0	0	1	0001	1
0	0	1	0	0010	2
0	0	1	1	0011	3
0	1	0	0	0100	4
0	1	0	1	0101	5
0	1	1	0	0110	6
0	1	1	1	0111	7
1	0	0	0	1000	8
1	0	0	1	1001	9
1	0	1	0	1010	10
1	0	1	1	1011	11
1	1	0	0	1100	12
1	1	0	1	1101	13
1	1	1	0	1110	14
1	1	1	1	1111	15

Tabelle 1.6: Zustandstabelle mit Überführung in Dezimalschreibweise

Der Vorteil dieser Schreibweise liegt auf der Hand. Man besitzt nun eine Variable, in der die gesamten Informationen der Zustände der Türen enthalten sind. Enthält die Variable z.B. den Wert 9, weiß man sofort, dass Tür 1 und 4 offen sind und Tür 2 und 3 geschlossen.

Bei den Dezimalzahlen sehen Sie, dass alle Türen einen Wert besitzen – nicht nur die Zustände, sondern auch die Türen, und zwar das jeweils binär zugeordnete Bit – dazu Tabelle 1.7.

	Binärwert	Dezimalwert
Tür1	0001	1
Tür2	0010	2
Tür3	0100	4
Tür4	1000	8

Tabelle 1.7: Türen und zugeordnete Werte

Sie besitzen jetzt eine sehr praktikable Handhabungsmöglichkeit für das Zustandsproblem. Wenn man z.B. vom Grundzustand 0 (alle Türen geschlossen) in den Zustand „Tür 3 offen" übergehen möchte, muss man zu der Variable nur den Wert der jeweiligen Tür addieren.

```
zustand = 0;      // Alle Türen geschlossen
zustand += 4;     // Tür 3 wird geöffnet
```

Der Wert der Variablen `zustand` wird um 4 (den Wert der geöffneten Tür) erhöht. Wenn sich noch eine weitere Tür öffnet, addiert man den Wert wieder dazu (hier Tür1 = offen).

```
zustand+=1;       // Tür 1 wird geöffnet
```

Somit enthält die Variable `zustand` den Wert 5. Gemäß unserer obigen Definition (siehe Tabelle 0.4) entspricht dies dem Fall, dass die Türen 1 und 3 offen sind. Falls eine Tür wieder geschlossen wird, muss der jeweilige Wert der Tür wieder dividiert werden.

Wenn man davon absieht, dass der Lösungsansatz dieses Problems über das binäre Zahlensystem verläuft, hat es ansonsten nicht viel mit binären oder Bit-Operationen zu tun gehabt. Dies ändert sich nun. Die oben beschriebene Lösung ist praktizierbar, allerdings müssten Sie sich die Werte der Türen merken und wissen, welche Dezimalzahl zu welcher Türenstellung passt. Die Werte der unteren Tabelle kann man sich über die Potenzen von 2 (1,2,4,8,16...|| 2^1, 2^2, 2^3,2^4,2^5) herleiten. Sie kämen aber nicht darum herum, die obere Wertetabelle aufzuzeichnen. Bei diesem kleinen Beispiel mit vier Türen ist dies noch relativ problemlos zu bewerkstelligen. Bei größeren Räumen mit 8 oder 16 Türen werden die Probleme aber sofort sichtbar. Bei Letzterem wären dies 2^{16} (n^k) = 65536 Kombinationen.

n = Anzahl der Stellungen eines Objektes
k = Anzahl der Objekte

Ohne die Wertetabelle muss man die Werte in das binäre Zahlensystem umrechnen, um z.B. zu erfahren, ob Tür 3 auf ist, wenn der Zustand 15 beträgt.

Sie brauchen hier keine Umrechnung mehr per Hand nachzuvollziehen. Dies macht Flash automatisch. Über das Binärsystem bzw. mit Hilfe der Bit-Operatoren können Sie überprüfen, ob die dritte Tür auf ist.

Normalerweise würde man folgendes Vorgehen wählen:

```
zustand=15;
zustand &  0100
```

0100, da die dritte Stelle von rechts betrachtet dem Zustand der dritten Tür entspricht.

Mit dem bitweisen &-Operator wird jedes Bit einzeln verglichen. Die binäre Schreibweise von 15 lautet: 1111

```
zustand & 0100
```

wäre also gleichbedeutend mit

	1111
&	0100
	0100

Als Ergebnis werden nur die Bits (Stellen) 1 bleiben, bei denen in beiden Ausdrücken eine 1 vorkam.

Dementsprechend erhält man bei dem Vergleich 0100 als Ergebnis. Leider bietet Flash keine Möglichkeit, Binärzahlen direkt einzugeben. Daher muss man statt 0100 in Binärform zur dezimalen Schreibweise wechseln (octal oder hexadezimal wären auch möglich gewesen, aber weniger sinnvoll). Da man als Ergebnis lieber `true` oder `false` haben möchten, castet man dies noch nach Boolean. Folglich entspricht dies dann in Flash folgendem Code:

Listing 1.55: Anwendung des bitweisen UND

```
zustand = 15;
tuer3= Boolean (zustand & 4);
trace(tuer3);
```

Bildschirmausgabe: `true`

Die 4 ergibt sich, indem Sie die Binärzahl 0100 ins Dezimalsystem überführen – wie zuvor schon in der „Wertetabelle der Türen“ geschehen. Das Casten in den Typ Boolean klappt nur deshalb so gut, weil – falls die dritte Tür zu sein sollte – 0000 als Ergebnis herauskommen würde. Dies kommt daher, weil bei einem bitweisen UND und der Zahl 0100 nur die dritte Stelle mit der 1 überprüft wird, da die anderen 0 sind.

Jetzt wissen Sie zwar, wie Sie überprüfen, ob eine Tür geöffnet ist, aber obiges Beispiel, bei dem zu dem Zustand der jeweilige Dezimalwert der Tür addiert wurde, um sie zu öffnen, hat noch ein kleineres Problem.

Denn was passiert, wenn man die erste Tür aufmachen möchte (also zustand+=1), aber die Tür schon geöffnet ist? Nach unserem Schema wird dann die erste Tür geschlossen und stattdessen die zweite geöffnet.

Deshalb muss man bei der oben beschriebenen Methode immer überprüfen, ob eine Tür schon geöffnet ist. Sicherlich wäre eine Lösung dieses Problems mit dem bitweisen UND-Operator möglich, aber auch umständlich, wenn man in Betracht zieht, was das bitweise ODER leistet.

Kehren wir zur folgenden Ausgangssituation zurück: alle Türen bis auf die erste sind geschlossen. Das Programm versucht jetzt die erste Tür wieder zu öffnen. Für das Öffnen einer Tür wird aber nun der bitweise ODER-Operator benutzt.

Listing 1.56: Beispiel für bitweises ODER

```
zustand=1; //Nur die erste Tür geöffnet
zustand = zustand | 1; //Erste Tür soll geöffnet werden
trace(zustand);
```

Bildschirmausgabe: 1

Statt zustand = zustand | 1; zu schreiben, ist es auch möglich, die kurze Schreibweise zu wählen: Operator + Zuweisungsoperator – wie bereits in Kapitel 1.3, Abschnitt „Zuweisungsoperatoren“ ab Seite 40 vorgestellt. Wir haben hier, des Verständnisses wegen, die längere Form benutzt. Möglich wäre jedoch auch

```
zustand |= 1;
```

In dem Beispiel steht eine 1 hinter dem ODER, weil 1 der Wert der ersten Tür ist. Bei der zweiten Tür wäre dies 2. Bei der dritten 4 usw. (wie in der Tabelle mit den Werten der Türen aufgezeigt).

Die binäre Schreibweise von 1 bzw. wenn nur die erste Tür geöffnet ist, lautet 0001

zustand | 1

wäre also gleichbedeutend mit

	0001
|	0001
	0001

Als Ergebnis sind alle Bits (Stellen) 1, bei denen in einem der beiden Ausdrücke eine 1 vorkam.

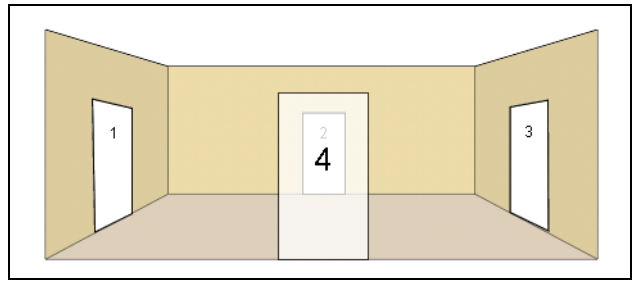
Falls man versucht, über diese Methode eine Tür zu öffnen, die schon geöffnet ist, passiert nichts. Ansonsten wird diese normal geöffnet.

So weit zur Anwendung der bitweisen UND- und ODER-Operatoren. Mit Hilfe des bitweisen NOT lassen sich auch entsprechend umgekehrt funktionierende Logiken erstellen.

Erweitern wir nun das Beispiel um folgende Definitionen:

- Die vier Türen führen alle in den gleichen Raum.
- Die gegenüber liegenden Türen dürfen nie gleichzeitig geöffnet werden.

Abbildung 1.8: Skizze zum Beispiel



Die Aufgabe lautet ein Programm zu schreiben, das erkennen muss, wenn die gegenüber liegenden Türen gleichzeitig geöffnet sind.

Dies sind die Zustände

10, 11, 14, 15 für Tür 2 und 4 sowie
5, 7, 13, 15 für Tür 1 und 3.

Diese alle einzeln abzufragen ist sicherlich die einfachste Möglichkeit, da wir uns jetzt auch schon die Mühe gemacht haben, die einzelnen Werte der Variablen `zustand` für die entsprechenden Ereignisse herauszusuchen. Aber normalerweise ist dies am intelligentesten auch wieder mit Hilfe der bitweisen Operatoren zu lösen.

Warum sollte man alles einzeln vergleichen, wenn man mit den logischen Operatoren die relevanten Informationen aus der Variable `zustand` gewinnen kann: Nämlich zu überprüfen, ob Tür 1 und 3 oder 2 und 4 zur gleichen Zeit geöffnet sind.

Dafür gibt es zwei Möglichkeiten: Entweder man überprüft mit dem logischen UND und erhält den Wert der zwei Türen (falls sie auf sind, hier 1 und 3)

	Ausgangsposition
&	0101
	0101 (falls Tür 1 und 3 geöffnet sind)

oder man benutzt das logische ODER und erhält den maximalen Wert der Binärzahl (falls beide Türen auf sind, hier 2 und 4)

	Ausgangsposition
|	0101
	1111 (falls Tür 2 und 4 geöffnet sind)

Es ist einfacher, das bitweise ODER zu benutzen, weil dann das Ergebnis bei geöffneten Türen 1111 entspricht. Dies kann man anschließend in eine oooo umkehren. Wenn man dies nun in den Typ Boolean wandelt, kommt für den Fall, dass beide Türen offen sind, `false` heraus.

Dementsprechend kann man für die Türen 2 und 4 folgendes ActionScript erstellen:

```
(zustand|5)
```

Die 5 steht hierbei für den binären Wert 0101. Wie oben schon erklärt, nimmt der gesamte Ausdruck, falls beide Türen offen sind, den binären Wert 1111 an.

Binärer Aufbau einer Zahl

In Kapitel „Binär-Werte“ ab Seite 21 haben wir den binären Aufbau einer Zahl beschrieben und im Exkurs dazu erwähnt, dass der Rechner diese auch so abspeichert. Dies stimmt natürlich nur bedingt. Jeder Rechner reserviert für eine Zahl eine bestimmte Länge an Bits im Arbeitsspeicher. In Flash direkt können Sie diese Länge nicht beeinflussen, so wird für jede Zahl immer die gleiche Menge an Bits reserviert. Das Maximum der reservierten Bits variiert natürlich zwischen den jeweiligen Rechnersystemen. Zur Abspeicherung der Informationen im Arbeitsspeicher werden diese abermals neu geordnet (Little-Endian-Adressierung bei Intel/Alpha, Big-Endian-Adressierung bei Motorola/Spark). Dies sei aber nur nebenbei erwähnt, für die Benutzung von Flash ist es vollkommen irrelevant. Wir wollen damit nur andeuten, dass Sie nie direkt in Ihrem Arbeitsspeicher Ihre Bitketten wieder finden.

Flash MX arbeitet mit bis zu 15-stelligen Dezimalzahlen, ohne zu runden. Bei größeren Zahlen benutzt Flash die Exponentialschreibweise, und es treten die üblichen Rundungsfehler auf:

```
trace(9999999999999999-9999999999999998);
```

Listing 1.57: Rundungsfehler bei 16-stelligen Dezimalzahlen

Bildschirmausgabe: `0`

Dies zeigt auch auf, dass Flash intern mit größeren Werten als 32 Bit rechnen kann. Allerdings muss man dann bei der Benutzung der Bitoperatoren aufpassen, denn die Bitoperatoren wandeln alle Zahlen in eine 32-Bit-vorzeichenlose-Ganzzahl um, so dass dies zu folgenschweren Fehlern führen kann!

```
trace(10000000000);
trace(10000000000|0);
```

Listing 1.58: Anwendung von Bitoperatoren auf Zahlen größer als 32 Bit

Bildschirmausgabe:

```
10000000000
1410065408
```

Ihnen sollte bewusst sein, dass die Anwendung eines bitweisen ODER mit 0 keinen Einfluss auf eine Zahl hat. Theoretisch sollte das Ergebnis also zweimal 10000000000 enthalten. Denken Sie immer daran, wenn Sie mit den Bitoperatoren arbeiten, dass die Zahlen im 32-Bit-Bereich liegen müssen, da die Operation sonst zu einem falschen Ergebnis führt!

Es existiert aber auch noch eine Formatierung innerhalb der jeweiligen Bitkette. Dort muss nämlich auch abgespeichert werden, ob es sich um einen positiven oder negativen Wert handelt. Dafür ist das erste Bit von links reserviert. Ist dieses Bit 0, ist die Zahl positiv, bei einer 1 wiederum negativ. Dies ist wichtig zu wissen, wenn man sich einmal genauer das bitweise NOT anschaut, auf das wir im Folgenden kurz eingehen.

Analog dazu – Tür 1 und 3:

```
(zustand|10)
```

Falls einer der vorgegebenen Zustände eintritt, enthält der Ausdruck den binären Wert 1111. Falls nur eine Tür offen ist, stehen dort z.B. Werte, wie 1011 oder 1110. Wie oben schon erläutert, muss man, um den Wert sinnvoll in einen Boolean casten zu können, diesen negieren. Leider erfüllt das bitweise NOT (~) nicht diese Funktion, da es sich dabei um eine komplette Negation aller Bits handelt, einschließlich des Vorzeichen-Bits.

Wie im Exkurs beschrieben, existiert bei jeder Zahl ein Vorzeichenbit. Das bitweise NOT negiert das Vorzeichenbit und subtrahiert von dem Gesamtausdruck 1. Bei negativen Zahlen wird 1 addiert.

Wir wollten Ihnen nur aufzeigen, dass es für das obige Problem nicht benutzt werden kann. Stattdessen benutzt man das bitweise XOR mit einer entsprechend langen binären Zahl mit Einsen.

Das bitweise XOR vergleicht zwei Bitketten und dort, wo diese sich unterscheiden, erfolgt im Ergebnis eine 1.

Listing 1.59: Beispiel für bitweises XOR

```
trace(3 ^ 6); // 0011 ^ 0110
```

Bildschirmausgabe: `5 // 0101`

Hier noch einmal die Bitketten direkt untereinander geschrieben:

```
0011 ^
0110
0101
```

Kehren wir wieder zurück zum Beispiel, da die Variable `zustand` 2^4 Kombinationen hat, dementsprechend den dezimal Bereich von 0 bis 15 abdeckt, muss man das bitweise XOR mit 15 benutzen.

Dies sieht für den ersten Ausdruck wie folgt aus:

```
((zustand | 5) ^ 15)
```

Wie bei der Beschreibung des bitweisen ODER erwähnt, nimmt der innere Ausdruck (zustand | 5) für den Fall, dass die Türen 2 und 4 geöffnet sind, den binären Wert 1111 an.

Wenn man nun darauf das bitweise XOR mit der binären Zahl 1111 (also Dezimal 15) anwendet, wird der komplette Ausdruck 0000. Allerdings nur für den Fall, dass die Türen 2 und 4 offen sind, ansonsten bleibt dort an einer Stelle eine 1 stehen!

Das Gleiche passiert für den anderen Ausdruck

```
((zustand|10)^15)
```

Wenn nun die Situation eintritt, dass die Türen 1 und 3 oder 2 und 4 geöffnet werden, nimmt einer der beiden Ausdrücke – ((zustand | 5) ^ 15) oder ((zustand|10)^15) – den binären Wert 0000 an. Man negiert diese Ausdrücke und castet sie auch somit zu dem Typ Boolean.

```
!((zustand|5)^15)
!((zustand|10)^15)
```

Anschließend kann man die zwei booleschen Variablen noch durch ein logisches ODER verbinden und erhält damit direkt die Lösung zu der gestellten Aufgabe.

```
trace(  !((zustand|5)^15)  || !((zustand|10)^15)   );
```

Wer sich in Logik auskennt, erkennt sofort, dass man diesen Ausdruck nach dem Theorem von De Morgan noch umstellen kann.

Theorem von De Morgan

Augustus De Morgan (1806 – 1871) begründete die logische Theorie der Relationen. Von ihm stammt auch die Erkenntnis des folgenden Satzes:

!(a || b) == !a && !b

!(a && b) == !a || !b

In Worten bedeutet dies, dass zwei einzeln negierte Ausdrücke dem gleichen gesamt negierten Ausdruck entsprechen, wenn man das ODER durch ein UND ersetzt oder umgekehrt. Oder auf die Elektrotechnik bezogen: Ein ODER-Gatter mit negierten Eingängen entspricht einem UND-Gatter mit negierten Ausgängen und umgekehrt.

Im Folgenden noch einmal zwei Wertetabellen, um dies zu verdeutlichen.

A	B	!(A && B)	!A || !B
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Tabelle 1.8: Beweis für !(A && B) == !A || !B

A	B	!(A || B)	!A && !B
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Tabelle 1.9: Beweis für !(A || B) == !A && !B

Dementsprechend ist

```
trace(  !((zustand|5)^15)  || !((zustand|10)^15)   );
```

gleichbedeutend mit

```
trace(  !(((zustand|5)^15)&&((zustand|10)^15))   );
```

Restliche Operatoren

▶ ( ) Gruppierung

Der Gruppierungsoperator kann dafür genutzt werden, um komplette Ausdrücke zu klammern und diese so zu verschachteln. Die Funktionsweise ähnelt dem Klammern aus der Mathematik. So ist 6 * (5 + 6) analog zu var1 && (var2 || var3). Der Gruppierungsoperator erlaubt es also, die Rangreihenfolge der Operatoren zu verändern, indem der Ausdruck innerhalb der Klammern immer zuerst ausgewertet werden muss.

So wird bei der Rechnung zuerst 5 + 6 addiert und das Ergebnis mit 6 multipliziert, anstatt 6 * 5 + 6 zu rechnen. Genau das Gleiche passiert bei dem Vergleich, so wird zuerst `var2` ODER `var3` ausgewertet und anschließend erst die UND-Verknüpfung betrachtet. Ohne Klammern hätte der UND-Operator eine höhere Priorität und würde zuerst ausgeführt, was zu einem anderen Ergebnis führen würde.

Siehe auch Anhang A „Operatorliste“ ab Seite 333.

▶ ( ) Funktionsaufruf

Anders als beim Gruppierungsoperator dienen die Klammern, wenn sie einem Namen folgen, als Funktionsaufruf. Der Befehl `trace()` der zu Beginn vorgestellt wurde, stellt zum Beispiel einen Funktionsaufruf der Funktion `trace()` dar.

Mehr zu Funktionen und Funktionsaufrufen finden Sie in Kapitel 1.7 ab Seite 89.

▶ [ ] ArrayElement

Die Schreibweise mit den eckigen Klammern im Anschluss an einen Namen dient dazu, ein Array anzusprechen. Alternativ kann der Array-Operator auch zum Auswerten von Ausdrücken benutzt werden.

Mit dem Befehl var1[1] könnten Sie beispielsweise auf das erste Array-Element des Arrays mit dem Namen `var1` zugreifen. Anders hingegen kann man mit i=1; var2= _root[„var“+i]; der Variablen `var2` den Inhalt der Variablen `var1` zuweisen, wobei `var1` hier dynamisch erzeugt wurde. Ähnliche Befehlsfolgen hat man früher in Flash 4 mit `eval()` benutzt, als Ersatz für Arrays. Im letzten Fall ähnelt die Funktionsweise des Array-Operators der von `eval( )`, der innen liegende Ausdruck wird ausgewertet.

Mehr über die Funktionsweise von Arrays und des Arrayzugriff-Operators erfahren Sie in Kapitel 1.5 ab Seite 67.

▶ . Strukturelement

Seit Flash 5 existiert die Möglichkeit, durch die Punktsyntax auf die Eigenschaften bzw. Variablen eines Objektes zuzugreifen. Des Weiteren dient der Punktoperator als Strukturelement und hat somit den in Flash 4 benutzten / Schrägstrich (Slash) abgelöst. Mehr zu dem Aufbau von Flash und zur Punktsyntax finden Sie in den Kapiteln 1.8 und 2.

▶ new Objektzuweisung

Mit `new` wird ein neues Objekt einer Klasse gebildet. Mehr zum Erstellen von Objekten finden Sie in Kapitel 1.8 ab Seite 102.

▶ delete

Mit dem Delete-Operator können Sie Variablen oder Objekte, z.B. auch Arrays, löschen.

```
delete var1;
```

Listing 1.60: Löscht die Variable var1

Der komplette Ausdruck ergibt ein `true`, wenn das Löschen der Variable erfolgreich geschehen ist. Falls die Variable `var1` nicht existierte, ergibt dies `false`.

▶ typeOf

Mit dem typeOf-Operator kann man den Typ einer Variablen abfragen.

```
var1="Hallo Welt";
trace(typeOf(var1));
```

Listing 1.61: Abfrage des Typs der Variable var1

Bildschirmausgabe: `String`

Mehrere Beispiele dazu finden Sie im Abschnitt „Mögliche Typen von Variablen“ ab Seite 19.

▶ ?: Bedingt

Dieser Operator bietet eine kürzere, alternative Schreibweise zu einer If-Else-Abfrage.

Siehe Abschnitt „If-Else-If-Abfrage“ ab Seite 64.

▶ , Mehrfache Auswertung

Mit dem Komma-Operator können Sie mehrzeilige Ausdrücke zu einer Zeile zusammenfassen. Dies kann immer dann sinnvoll sein, wenn Ihnen die benutzte Funktion/Bedingung/Schleife nur eine Zeile zur Verfügung stellt. In Abschnitt „If-Else-If-Abfrage“ ab Seite 64 wurde dieser Operator z.B. benötigt.

1.4 Bedingungen (Abfragen)

Bisher haben wir Ihnen nur Mittel zum Erstellen eines stur ablaufenden Programmtextes gezeigt; nun kommen wir zum ersten Mittel, mit dem Sie den Ablauf Ihres Programms beeinflussen können. Sie stellen quasi eine Weiche und bringen so den Interpreter dazu, einen alternativen Programmtext auszuführen.

if-Abfrage

Eine If-Abfrage benötigt man bei einer Vielzahl von Anwendungen. Auch wenn wir bei den bisherigen Beispielen bewusst darauf verzichtet haben, ist sie doch eine der grundlegendsten Programmierwerkzeuge.

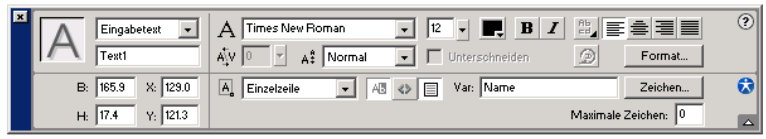
Als Beispiel bedienen wir uns diesmal einer Passwortabfrage. Es sei zuvor noch erwähnt, dass die beschriebene Passwortabfrage keine Relevanz in der Praxis hat, da diese Abfrage unsicher ist. So braucht jeder Besucher das SWF nur herunterzuladen und sich einmal den Quellcode mit einem Editor (z.B. dem Notepad) anzuschauen; dort findet er prompt das entsprechende Passwort. Eine sichere Passwortabfrage können Sie erstellen, indem Sie die Daten z.B. über ein PHP (Skriptsprache, die serverseitig ausgeführt wird) an Flash übermitteln.

Nichtsdestotrotz ist eine Passwortabfrage ein gutes Beispiel für eine If-Bedingung.

Die Datei zum Beispiel finden Sie auf der CD unter *Kapitel1/01_if.fla*.

Zuerst erzeugt man ein Eingabefeld für das Passwort bzw. den Namen. In Kapitel 4.2 ab Seite 169 wird der Umgang mit Textfeldern erläutert. Wenn Sie das Textfeld erstellt haben, sollte dies wie folgt aussehen.

Abbildung 1.9: Eingabefeld



Wichtig ist dabei, dass links oben Eingabetext ausgewählt wird, sowie der Variablenname unten rechts vergeben und der Button für den Rahmen aktiviert wurde.

Nachdem Sie zwei Textfelder für Name und Passwort erstellt haben, beschriften Sie diese noch und erstellen einen „Okay“-Button. Zeichnen Sie dazu ein Rechteck in Größe des späteren Buttons auf die Bühne, markieren Sie es und drücken Sie anschließend F8 (Einfügen › In Symbol konvertieren). Eine kurze Einführung und Erläuterung zu Buttons finden

Sie in Kapitel 4.3 ab Seite 179. Dies alles sollten Sie auf der Hauptbühne im ersten Frame erstellt haben. In den zweiten Frame fügen Sie Ihre eigentliche Seite ein. Damit der Flashplay beim Abspielen des Movies direkt im ersten Frame stoppt, müssen Sie dort in das Keyframe noch ein `stop();` platzieren. Dadurch bleibt der Abspielkopf im ersten Frame stehen und dem Besucher wird Ihre Passwortabfrage angezeigt.

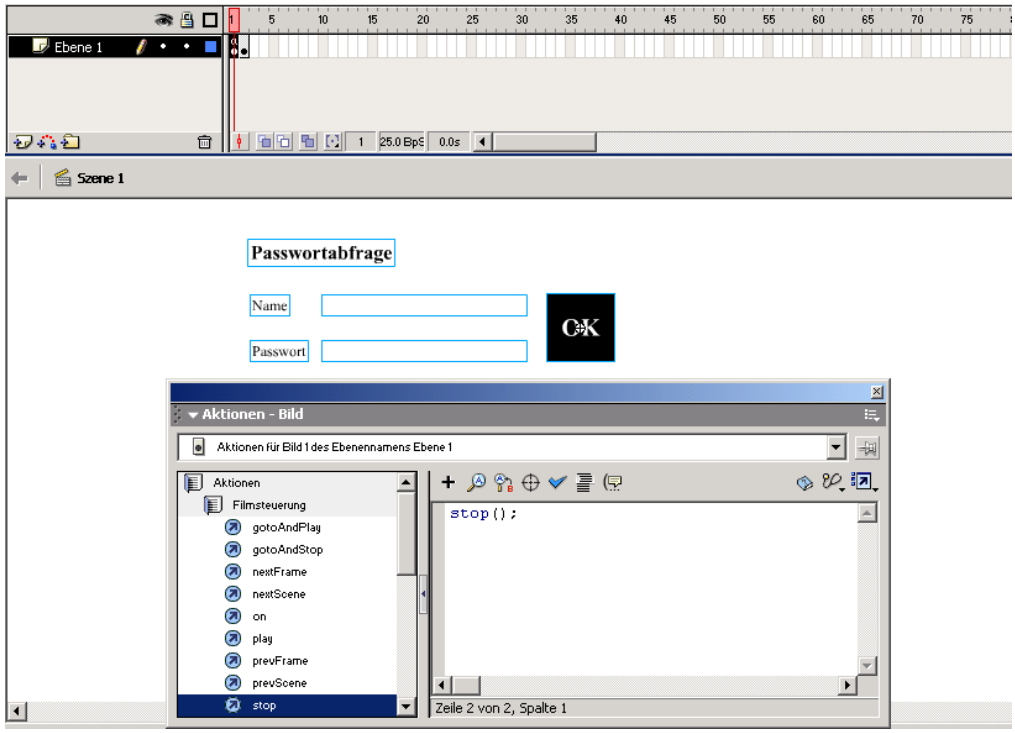


Abbildung 1.10: Passwortabfrage, Frame 1

Nun schreiben Sie noch das eigentliche ActionScript. Wir erwarten von dem Besucher, dass er das Passwort und seinen Namen eingibt und anschließend auf den Button drückt.

Wenn der Button gedrückt wird, muss Flash vergleichen, ob das Passwort und der Name zulässig sind. Wenn den Eingabetextfeldern die Variablennamen `Name` und `Passwort` zugeordnet sind, kann man diese direkt abfragen und vergleichen.

Die Auswirkungen des folgenden Codesegmentes sollten Sie kennen.

```
Trace( Name == "Admin" && Passwort=="Gott" );
```

Es wird verglichen, ob die Variable `Name` den Inhalt „Admin“ hat und ob die Variable `Passwort` den Inhalt „Gott“ hat; falls dies beides zutrifft, wird ein true ausgegeben.

If-Abfragen reagieren nur auf `true` oder `false`, also auf Boolean-Werte. Wenn eine If-Abfrage ein `true` enthält, wird der Inhalt des folgendes Blocks (ein Block ist ein mit geschweiften Klammern `{ }` umschlossenes Codesegment) ausgeführt, bei `false` wird dieser einfach übergangen. Falls ein Block nur eine Codezeile enthält, können die geschweiften Klammern entfallen, der Übersicht wegen ist aber ratsam, sie trotzdem zu benutzen!

Listing 1.62: Beispiel für eine simple Passwortabfrage mit Hilfe einer If-Bedingung

```
on(release){
    if(  Name == "Admin" && Passwort=="Gott" ){
        gotoAndStop(2);
    }
}
```

Dieses Mal benutzten wir kein `trace()` mehr, um das Ergebnis sichtbar zu machen. Dies ist nicht mehr notwendig, da bei richtiger Eingabe der Film direkt in den zweiten Frame springt und bei falscher Eingabe nichts passiert. Hier noch einmal der Code in Worten: „Wenn die Variable `Name` den String „Admin“ UND die Variable `Passwort` den String „Gott“ enthält, bewege den Abspielkopf zu Frame 2.“

Letzteres ist natürlich sehr unschön, da der Besucher auf die Idee kommt, er hätte den Knopf nicht richtig gedrückt (oder Ähnliches).

Bei einer If-Abfrage mit dem Vergleichsoperator == ist darauf zu achten, auch wirklich zwei Gleichheitszeichen zu benutzen. Bei einem einzelnen Gleichheitszeichen interpretiert dies Flash als Zuweisung und gibt keine Fehlermeldung aus!

If-Else-Abfrage

Um noch eine Alternative ausführen zu können, wenn die If-Abfrage false ergibt, existiert der Else-Zweig.

Listing 1.63: Beispiel für eine If-Abfrage mit Else-Zweig

```
var1=true;
if(var1){
    trace(„Wahr");
}else{
    trace(„Unwahr");

}
```

Wenn die Variable `var1` `true` ist, wird „Wahr“ ausgegeben, ansonsten „Unwahr“. Wenn letzteres eintritt, muss die Variable `var1` nicht zwingend `false` enthalten! Es bedeutet lediglich, dass sie nicht `true` enthält.

Variablen werden automatisch für den Vergleich zu Boolean gecastet! So ist z.B. 1 auch `true` !

Die Datei zu dem Beispiel finden Sie auf der CD unter *Kapitel1/02_else.fla*.

Wenn Sie nun das Script von der oberen Passwortabfrage um den Else-Zweig ergänzen, könnte dies wie folgt aussehen:

Listing 1.64: If-Abfrage mit Else-Zweig

```
on(release){
    if( Name == "Admin" && Passwort=="Gott"){
        gotoAndStop(2);
    }else{
        Name="";
        Passwort="Unzulässige Eingabe";
    }
}
```

Falls der Besucher ein unzulässiges Passwort eingibt, wird die gesamte Eingabe gelöscht, und in dem Textfeld für das Passwort erscheint der String „Unzulässige Eingabe“. Dies ist natürlich sehr unschön, da der Besucher dann alle Daten wieder erneut eingeben muss. Man hätte genauso gut noch ein zusätzliches dynamisches Textfeld weiter unten anordnen können, in dem dann der String „Unzulässige Eingabe“ ausgegeben wird.

Verkürzte If-Else-Abfrage ?:

Wenn Sie nur eine If-Abfrage mit einem Else-Zweig brauchen, gibt es noch eine zweite Notationsmöglichkeit. Diese ist um einiges kürzer, dafür aber auch ein wenig unübersichtlicher:

```
If(Bedingung){
    Code1();
}else{
    Code2();
}
```

Normale If-Abfrage mit Else-Zweig. Die folgende Zeile dient dem gleichen Zweck:

```
Bedingung ? Code1() : Code2();
```

Bedingung steht hierbei für einen Vergleich, der ein Boolean als Ergebnis liefert. Code1() und Code2() stehen für ein beliebiges einzeiliges ActionScript. Zuerst kommt die Abfrage, wenn diese `true` ergibt, wird der Befehl vor dem Doppelpunkt ausgeführt, ansonsten der Befehl, der danach folgt.

Hier noch einmal das zuvor erwähnte ActionScript aus der If-Else-Abfrage

Listing 1.65: Normale If-Abfrage mit Else-Zweig

```
on(release){
    if(  Name == "Admin" && Passwort=="Gott"){
        gotoAndStop(2);
    }else{
        Name="";
        Passwort="Unzulässige Eingabe";
    }
}
```

und im Folgenden die entsprechend kürzere Version

Listing 1.66: If-Else-Abfrage in kurzer Schreibform

```
(Name == "Admin" && Passwort=="Gott") ?    gotoAndStop(2):
    Name="",Passwort="Unzulässige Eingabe";
```

Die Datei zu diesem Beispiel finden Sie auf der CD-ROM unter *Kapitel1/03_elseK.fla*.

Da der Else-Zweig zwei Befehle enthält – nämlich `Name="";` `Passwort="Unzulässige Eingabe";` – muss man diesen für die gekürzte Schreibform der If-Else-Abfrage einzeilig schreiben. Dies geschieht mit Hilfe des Komma-Operators: `Name="",Passwort="Unzulässige Eingabe";`

In Flash werden einzelne Befehle immer durch ein Semikolon am Ende gekennzeichnet. Dementsprechend handelt es sich bei dem Befehl `Name="";` `Passwort="Unzulässige Eingabe";` eigentlich – auch wenn dieser in einer Zeile steht – um zwei Befehle.

Eine verkettete Ausführung mehrerer Befehl in einem Schritt kann man durch den Kommaoperator erzeugen.

Ein weiteres Beispiel zu der verkürzten If-Else-Bedingung finden Sie am Ende des Kapitels 1.2 im Abschnitt „Stringfunktionen und Eigenschaften“ ab Seite 29 in dem Beispiel des E-Mail-Validators.

If-Else-If-Abfrage

Wenn man mehrere User zulassen möchte, kann man die If-Bedingung einfach erweitern:

```
if(  Name == "Admin" && Passwort=="Gott" || Name == "Admin2"
    && Passwort=="Liebe" ){
        gotoAndStop(2);

    }
```

Dies wäre ohne Probleme so realisierbar, auch ohne die einzelnen Ausdrücke zu klammern, da die UND-Operatoren stärker binden als der ODER-Operator. Was jedoch tun, wenn Sie die unterschiedlichen User auch auf unterschiedliche Seiten verweisen möchten?

Dies ist sehr einfach und übersichtlich mit der Else-If-Abfrage machbar, so dass keine If-Abfragen verschachtelt werden müssen:

```
on(release){
    if(  Name == "Admin" && Passwort=="Gott"){
        gotoAndStop(2);
    }else if(Name == "Admin2" && Passwort=="Liebe"){
        gotoAndStop(3);
    }else{
        Name="";
        Passwort="Unzulässige Eingabe"
    }
}
```

Listing 1.67: Beispiel für eine Else-If()-Abfrage

Die Datei zu diesem Beispiel finden Sie auf der CD-ROM unter *Kapitel1/04_else1.fla*.

Dies kann man unendlich erweitern, da jede If-Abfrage unendliche viele `else if()` enthalten kann. In Worten heißt der obere Quelltext so viel wie: „Wenn die Variable `Name` den String „Admin“ UND die Variable `Passwort` den String „Gott“ enthält, bewege den Abspielkopf zu Frame 2, ansonsten vergleiche, ob die Variable `Name` den String „Admin2“ UND die Variable `Passwort` den String „Liebe“ enthält, und bewege den Abspielkopf zu Frame 3. Ansonsten weise der Variablen `Name` eine leere Zeichenkette und der Variablen `Passwort` den String „Unzulässige Eingabe“ zu.“

Switch-Case

Für den Fall, dass man eine Variable oft auf bestimmte Inhalte überprüfen muss, bietet Flash MX nun eine Switch-Case-Abfrage an.

Die Datei zum Beispiel finden Sie auf der CD unter *Kapitel1/05_switc.fla*.

Als Beispiel haben wir ein kleines Rechenprogramm für Kinder erstellt.

Der Aufbau ist ähnlich dem der Passwortabfrage, nur dieses Mal existiert nur ein Eingabefeld für die Lösung und ein Ausgabefeld mit einer kleinen Hilfestellung, falls ein falsches Ergebnis eingegeben wurde.

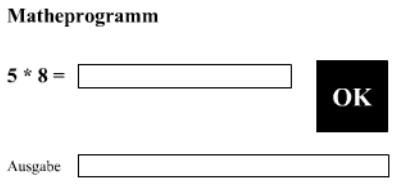


Abbildung 1.11: Bildschirm des Matheprogramms

Am besten gehen wir direkt auf das auf dem Button liegende ActionScript ein.

Listing 1.68: Switch-Case-Abfrage für kleines Matheprogramm

```
on (release) {
    switch (ergebnis) {
    case "5" :
        ausgabe = "1 mal 5 ergibt 5";
        break;
    case "8" :
        ausgabe = "1 mal 8 ergibt 8";
        break;
    case "10" :
        ausgabe = "2 mal 5 ergibt 10";
        break;
    case "15" :
        ausgabe = "3 mal 5 ergibt 15";
        break;
    case "16" :
        ausgabe = "2 mal 8 ergibt 16";
        break;
    case "20" :
        ausgabe = "4 mal 5 ergibt 20";
        break;
    case "24" :
        ausgabe = "3 mal 8 ergibt 24";
        break;
    case "25" :
        ausgabe = "5 mal 5 ergibt 25";
        break;
    case "30" :
        ausgabe = "6 mal 5 ergibt 30";
        break;
    case "32" :
        ausgabe = "4 mal 8 ergibt 32";
        break;
    case "35" :
        ausgabe = "7 mal 5 ergibt 35";
        break;
    case "40" :
        gotoAndStop(2);
        break;
    default :
        ausgabe = "Falsches Ergebnis :o( Versuche es erneut!";
    }
}
```

Wenn der User in das Eingabefeld eine 5 eingibt, wird im Ausgabefeld der Text „1 mal 5 ergibt 5“ als Hilfestellung mit ausgegeben. Bei der Switch-Case-Anweisung gibt man also bei `switch` die Variable an, auf die man sich bezieht, und kann anschließend mit `case` Fallunterscheidungen bei verschiedenen Zuständen erzeugen. Hier im Beispiel wurde jedes `case` durch ein `break` beendet. Wenn man nicht möchte, dass die nachfolgenden Fallunterscheidungen auch noch mit ausgeführt werden, muss

man dies so programmieren. Falls Sie z.B. bei dem letzten case-Befehl das `break` entfernen und dieser aufgerufen wird (`eingabe` muss dafür 40 sein), wird zuerst dieser Befehl ausgeführt und danach noch alle folgenden – in dem Fall also der Inhalt aus dem default-Block.

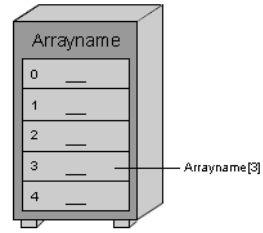
Die angegebenen Werte hinter dem `case` stehen in Anführungszeichen, da diese direkt aus dem Eingabetextfeld stammen und es sich deshalb um einen String handelt. Man kann aber mit switch-case auch Number- oder Boolean -Vergleiche durchführen.

Falls keine der vorgesehenen Fallunterscheidungen eintritt, wird `default` aufgerufen.

1.5 Arrays

Die Begriffe String, Number, Boolean usw. können Sie zuordnen, doch was ist ein Array? Wieder ein neuer Typ? Wohl kaum. Ein Array ist vielmehr als übergeordnete Struktur zu sehen, er kann alle existierenden Variablentypen in sich speichern.

Anders als bei den gängigen Programmiersprachen, in denen in einem Array immer nur Daten eines Variablentyps gespeichert werden können, kann man unter Flash in einem Array alles speichern: Strings oder Nummern, selbst die Reihenfolge spielt keine Rolle. Die Arrays in Flash sind also schon vergleichbar mit Strukturen (C++) bzw. Records (Pascal). Allerdings sind Arrays in Flash nicht auf eine feste Länge begrenzt. Im Folgenden wird noch eingehend erläutert, was Arrays machen und wofür man sie nutzen kann.



Wir haben Ihnen anfangs erklärt, dass Sie sich eine Variable wie eine Schublade vorstellen sollen, die einen Namen besitzt und in der ihr Wert abgelegt wird. Ein Array können Sie sich, um bei dieser Metapher zu bleiben, als einen Schrank mit vielen Schubladen vorstellen. Die Schubladen besitzen jetzt allerdings eine durchgängige Nummerierung (einzelne Namen sind zwar auch möglich, davon wollen wir aber erst einmal absehen). Jeder Schrank bekommt einen Namen zugewiesen, so können Sie dann später direkt jede Schublade eindeutig bezeichnen. Schrank „Arrayname“, Schublade Nummer 3.

Arrays eignen sich hervorragend dazu, Daten aus Listen zu speichern. Gehen Sie einmal davon aus, es ginge um ein Projekt, in dem Sie die Namen von Leuten speichern müssen.

Wenn Sie dies mit einzelnen Variablen bewerkstelligen, haben Sie zuerst das Problem, dass Sie sich für identische Sachen unterschiedliche Variablennamen einfallen lassen müssen.

```
var name="Jan";
var zweitername="Marc";
```

usw.

Dies ist sehr unpraktisch, da Sie sich immer auch den Variablennamen „merken“ müssen.

Wer ein wenig über das Problem nachdenkt, kommt bestimmt auch zu folgender Lösung

```
var name1 = "Jan";
var name2 = "Marc";
```

usw.

Diese Durchnummerierung war in Flash 4 üblich, da es dort keine Arrays gab. Mit Hilfe von `eval ("name"+i)` (i sei hier ein Zahlenwert zwischen 1 und 2) konnte man dann die „Namenliste“ ansprechen. Seit der fünften Version kann Flash auch mit Arrays umgehen. Dies vereinfacht die Handhabung solcher Listen.

Wie bei allen anderen Variablen-Typen besteht auch hier wieder die Möglichkeit, über die Konstruktorfunktion ein leeres Array zu erstellen. Der Name `arr` ist hierbei ein beliebiger Name für das Array.

```
arr = new Array( );
```

Des Weiteren kann man der Kostruktorfunktion auch Parameter mitgeben, die dann dem Array direkt als Werte zugewiesen werden.

```
arr=new Array("Jan","Marc");
```

Der Inhalt dieses Arrays würde der oben zuvor erstellten Liste entsprechen.

Man kann die Schreibweise aber auch noch weiter vereinfachen, indem man erst gar nicht die Konstruktorfunktion benutzt.

```
arr=["Jan","Marc"];
```

Der Vorteil gegenüber der Variante, mit einzelnen Variablen zu arbeiten, wird schon beim Initialisieren der Werte sichtbar. Es ist nicht nur einfacher, sondern auch übersichtlicher!

Wenn man nun auf den ersten Listenwert zugreifen möchte, schreibt man einfach

```
Arrayname[Listenfeldnummer]
```

Da Arrays immer von 0 an durchnummeriert werden, würde man, um den Namen „Jan“ abzufragen, folgende Notation benutzen:

```
arr[0]
```

Im Folgenden noch einmal ein Beispiel, damit Sie dies selbst testen können:

```
arr = [ „Jan", „Marc", „Frank", „Fabian"]; //erstellt ein Array
    namens arr
trace( arr[0] ); //Gibt den Inhalt des ersten Arrayelements aus
trace( arr[1] ); //Gibt den Inhalt des zweiten Arrayelements aus
trace( arr[2] ); //Gibt den Inhalt des dritten Arrayelements aus
trace( arr[3] ); //Gibt den Inhalt des vierten Arrayelements aus
```

Listing 1.69: Kleines Beispiel zur Erstellung und Ausgabe eines Arrays

Bildschirmausgabe:

```
Jan
Marc
Frank
Fabian
```

Jedes Array enthält genau eine Eigenschaft – die Länge, die Sie über `arrayname.length` abfragen können. Achten Sie darauf, nicht aus Versehen noch Klammern zu setzen, wie es bei der Längenabfrage bei Strings nötig ist. Es handelt sich bei Arrays um eine Eigenschaft, die aufgerufen wird, und nicht um eine Methode! Wenn Sie z.B. das obere Beispiel noch um folgende Zeile am Ende erweitern, wird ihnen die Array-Länge mit ausgegeben.

```
trace(arr.length);
```

Listing 1.70: Erweitertes Beispiel mit Ausgabe der Länge des Arrays

Bildschirmausgabe:

```
Jan
Marc
Frank
Fabian
4
```

Die Länge des Arrays wird immer entsprechend der Menge der Inhalte gesetzt. Sie können aber auch mit Hilfe der Konstruktorfunktion ein Array mit einer bestimmten Länge definieren. Dafür geben Sie der Konstruktorfunktion als Parameter einen Numberwert mit auf den Weg.

```
arr = new Array(20);
```

Listing 1.71: Erstellung eines Arrays der Länge 20

Falls das Array größer sein muss oder Sie auf das 21. Element zugreifen, erweitert Flash dementsprechend automatisch das Array.

In Arrays speichert man also Daten ab. Dies bedeutet aber nicht zwingend, dass Sie eine Adressverwaltung oder Ähnliches in Flash realisieren. Arrays gehören zu den grundlegenden Programmiertechniken und man braucht sie öfter, als man glaubt.

Array-Methoden

Das Arbeiten mit einem Array bringt neben der Übersicht und der einfacheren Schreibweise noch einen weiteren enormen Vorteil. In Flash existieren Array-Methoden, die speziell für den Umgang mit Arrays erstellt wurden, ähnlich der String-Methoden.

Hier vorab eine Übersicht aller Methoden.

Methodenaufruf	Funktion
arrayname.concat(arr1, arr2 ....arrN)	verbindet zwei oder mehr Arrays durch Aneinanderhängen.
arrayname.join(Zeichen)	erzeugt einen String, in dem die Arrayelemente hintereinander, getrennt durch das angegebene „Zeichen“, eingefügt sind.
arrayname.pop()	entfernt das letzte Element des Arrays und gibt den Inhalt des Elementes zurück.
arrayname.push(var1, var2...varN)	fügt ein oder mehrere Elemente am Ende des Arrays hinzu und gibt die neue Länge des Arrays zurück.
arrayname.reverse()	kehrt die Reihenfolge der Array-Elemente um.
arrayname.shift()	enfernt das erste Element des Arrays und gibt den Inhalt des Elementes zurück.
arrayname.slice(index1, index2)	gibt den Inhalt der Elemente zwischen index1 und index2 wieder.
arrayname.sort([funktion])	sortiert das Array alphabetisch. Falls eine Funktion mit angegeben wird, muss diese –1 (kleiner), 0 (gleich groß) oder 1 (kleiner) als `return` Wert liefern.
arrayname.sortOn(Objektvariablenname)	wenn ein Array mehrere Objekte vom gleichen Typ enthält, kann man das Array nach einer Variable des Objektes alphabetisch sortieren lassen.
arrayname.splice(Index1, [Anzahl], [var1, varN])	löscht die Anzahl der angegebenen Elemente von Index1 (einschließlich). Oder/Und fügt die angegebenen Werte `var1...varN` ab Index1 ein.
arrayname.toString()	gibt die Arrayelemente in einem String zurück, getrennt durch ein Komma.
arrayname.unshift(var1, var2 ... varN)	fügt ein oder mehrere Elemente am Anfang des Arrays hinzu und gibt die neue Länge des Arrays zurück.

Notation

Eckige Klammern

`arrayname.sort( [funktion] )` bedeutet nicht, dass der Funktionsname in eckigen Klammern geschrieben werden muss, sondern dass die Angabe eines Funktionsnamens optional ist.

Punkte – N

`var1, var2 ... varN` bedeutet, dass Sie mindestens ein Element angeben müssen, aber theoretisch auch unendlich viele (n) dort angeben können.

Im Folgenden ein kleines Beispiel zur Anwendung der Methode `sort()`.

```
arr = [ "Jan", "Marc", "Frank", "Fabian"]; //erstellt ein Array
    namens arr
trace( arr[0] ); //Gibt den Inhalt des ersten Arrayelements aus
trace( arr[1] ); //Gibt den Inhalt des zweiten Arrayelements aus
trace( arr[2] ); //Gibt den Inhalt des dritten Arrayelements aus
trace( arr[3] ); //Gibt den Inhalt des vierten Arrayelements aus
arr.sort();
trace("\nsortiert:");
trace( arr[0] ); //Gibt den Inhalt des ersten Arrayelements aus
trace( arr[1] ); //Gibt den Inhalt des zweiten Arrayelements aus
trace( arr[2] ); //Gibt den Inhalt des dritten Arrayelements aus
trace( arr[3] ); //Gibt den Inhalt des vierten Arrayelements aus
```

Listing 1.72: Unsortierte und sortierte Ausgabe eines Arrays

Bildschirmausgabe:

```
Jan
Marc
Frank
Fabian

sortiert:
Fabian
Frank
Jan
Marc
```

Selbstverständlich ist es auch ohne Probleme möglich, Methoden zu verschachteln.

Im Folgenden benutzen wir die Stringfunktion `split()` mit dem Parameter „x“ und trennen so den String „Text“ in ein Array mit den Inhalten „Te“ und „t“. Danach wenden wir auf das Array die Methode `join()` mit dem Parameter „s“ an. So wird das Array wieder zu einem String zusammengesetzt und an der zusammengefügten Stellen jeweils ein „s“ eingesetzt.

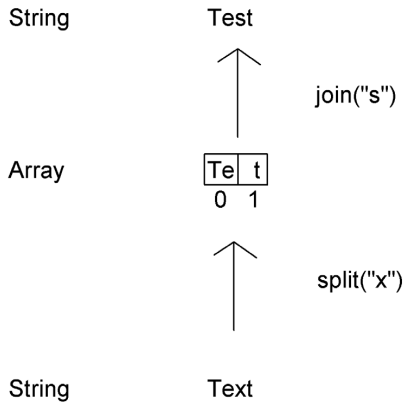


Abbildung 1.12: Beispiel für eine einfache replace-Aktion

So erhält man die Eigenschaften einer replace()-Funktion. In diesem Beispiel wurde das „x“ durch ein „s“ ersetzt.

Listing 1.73: Beispiel für mehrfache Anwendung von Methoden

```
var1 = "Text";
var1=var1.split("x").join("s")
trace(var1);
```

Bildschirmausgabe: `Test`

Kommen wir nun wieder zu einem Beispiel, das Sie mitbauen können. Es geht darum, drei Movies oder Bilder auf Knopfdruck ein- bzw. auszublenden. Erschwerend hinzu kommt, dass immer zufällig ein Movie ausgewählt werden soll, welches ein- oder ausgeblendet wird.

Die Datei zum Beispiel finden Sie auf der CD unter *Kapitel1/06_Aname.fla*.

Angenommen alle drei Filme werden angezeigt:

Abbildung 1.13: Zustandsaufnahme der späteren Ausgabe, wenn alle drei Filme angezeigt werden

So kann der User auf den linken Button klicken, und es soll automatisch einer der drei Filme ausgeblendet werden. Ein Klick auf den rechten Button hätte keine Wirkung, da maximal drei Movies angezeigt werden können.

Falls der User auf den linken Button klickt, wird ein Movie von Flash zufällig ausgewählt und ausgeblendet. Angenommen Flash würde das erste Movie ausblenden, sähe der Bildschirm wie folgt aus:

Abbildung 1.14: Anzeige von drei Filmen, von denen der erste ausgeblendet ist.

Wenn der User nun wieder auf den linken Button klickt, um ein weiteres Movie auszublenden, wählt Flash aus Film eins und zwei zufällig einen aus und blendet diesen aus. Anders herum sollte es für das Einblenden eines Films funktionieren.

Nachdem die Aufgabenstellung geklärt ist (falls nicht, starten Sie kurz das SWF und schauen sich die Funktionsweise einmal an) kommen wir nun zur Programmierung.

Als Erstes erstellen Sie die drei Filme. Bei uns im Beispiel werden diese nur ein- und ausgeblendet.

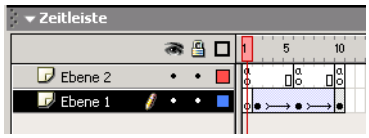


Abbildung 1.15:
Zeitleiste eines der drei Filme

In Frame 1 ist der jeweilige Film gar nicht sichtbar. Damit dies zu Beginn auch so bleibt, steht dort ein `stop()` im ersten Frame. Von Frame 2 bis 5 wird der Film (hier der Schriftzug) eingeblendet und stoppt danach in Frame 6. In Frame7 bis 10 wird der Film wieder ausgeblendet und springt anschließend wieder zu Frame 1.

Die drei Filme haben wir in einem weiteren Film (Container) erstellt. Dies wäre nicht nötig gewesen, man hätte sie genauso gut direkt auf der Hauptbühne erstellen können.

Wenn Sie nun Ihre drei Filme erstellt haben, müssen Sie diesen noch Instanznamen zuweisen, damit man die Filme auch mit ActionScript ansprechen kann. Wir haben den Filmen die Instanznamen: „film1“, „film2“ und „film3“ gegeben.

Dann erstellen Sie noch zwei Buttons, und es könnte so aussehen:

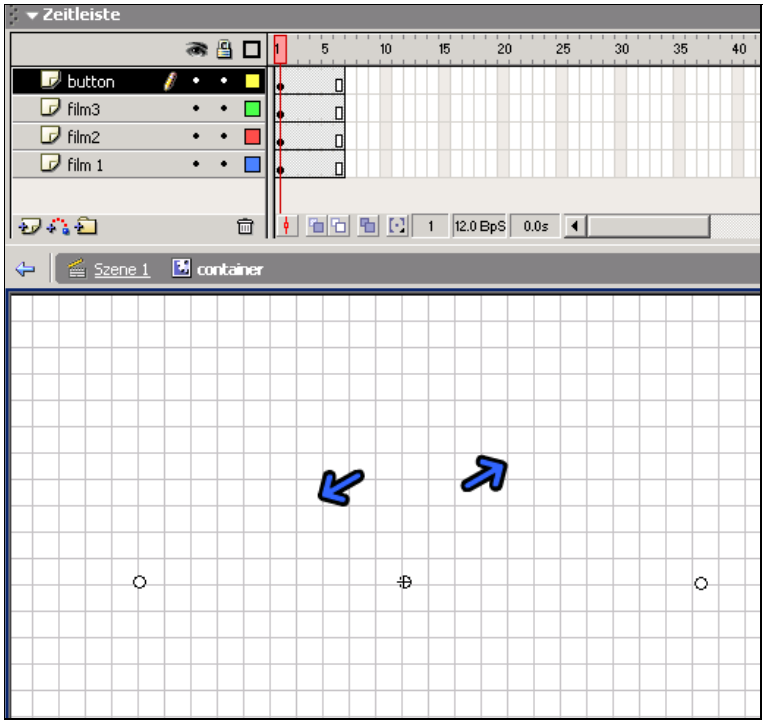


Abbildung 1.16:
Layout des späteren Beispiels unter Flash

Wie Sie sehen, befindet man sich bei uns im Movieclip „container“. Falls Sie Ihre Sachen jetzt auf der Hauptbühne liegen haben, lassen Sie sich davon nicht irritieren, es ist irrelevant!

Nachdem man das Layouttechnische alles erstellt hat, kann es an die Programmierung gehen. Damit Flash weiß, ob ein Film gerade eingeblendet ist, speichern wir seinen Instanznamen in einem Array. Sicherlich wäre es auch möglich gewesen, jedes Mal jeden Film einzeln anzusprechen und abzufragen, aber es geht hier um die Benutzung von Arrays, und sicherlich ist diese Lösung eleganter und mit weniger Aufwand verbunden.

Zu Beginn müssen die Arrays erstellt werden. Für den Fall, dass Sie auch ein zusätzliches Movie erstellt haben, in dem sich alles befindet, können Sie dies auch mit dem `onclipEvent(load)` bewerkstelligen, ansonsten kopieren Sie den Inhalt des Events in den ersten Frame auf Ihrer Bühne.

Listing 1.74: Initialisierung der Arrays

```
onClipEvent (load) {
      anzeigbar=["film1","film2","film3"];
      loeschbar=new Array();
}
```

Da Flash sich immer merken muss, welche Movieclips angezeigt werden, benutzt man zwei Arrays. In einem werden alle Movieclips gespeichert, die ausgeblendet sind (`anzeigbar`), in dem anderen alle, die eingeblendet sind (`loeschbar`). Wie Sie sehen, haben wir nur die Instanznamen der jeweiligen Movies als String in den Arrays gespeichert.

Gleichzeitig definiert das auch den Anfangszustand alle – Movieclips sind ausgeblendet. Wenn also nun ein Movie eingeblendet werden soll, wird der Name des Movies aus dem Array `anzeigbar` in den Array `löschbar` „verschoben“. (Es gibt keine direkte Methode für das Verschieben, stattdessen muss man den Namen löschen und im anderen wieder einfügen)

Wenn der User das Beispiel öffnet, bekommt er keinen Movieclip angezeigt, also muss er auf den rechten Button klicken. Nun soll Flash aus den Filmen, die nicht angezeigt werden – also alle, die im Array `anzeigbar` enthalten sind – einen per Zufall auswählen und einblenden.

Um eine zufällige Zahl zu ermitteln, gibt es in Flash zwei Möglichkeiten: die aus Flash 5 bekannte random()–Funktion:

- arrNr = random(anzeigbar.length);
  `arrNr` würde dann Ganzzahlen zwischen 0 und die Länge des Arrays –1 enthalten. Die Funktion gilt allerdings als veraltet und stattdessen soll die neue random-Methode des Matheobjektes verwendet werden.
- Math.random();
  Diese zweite Methode liefert immer einen zufälligen Wert zwischen 0 und 1 und nimmt keinen Parameter entgegen. Dementsprechend muss man den Wert selber in das gewünschte Intervall umrechnen.

```
arrNr = Math.floor(Math.random()*anzeigbar.length);
```

Es sollte verständlich sein, warum wir uns trotzdem für die „veraltete“ Methode entschieden haben.

Die zufällige Zahl wird über die Länge des Arrays gebildet. Wenn das Array eine Länge von 3 hat, kann die ermittelte Zahl nur die Werte 0, 1 oder 2 annehmen. Dieses Ergebnis nutzen wir direkt, um auf ein Array-Element zuzugreifen. Dementsprechend wurde auch der Variablenname gewählt: `arrNr` Arraynummer.

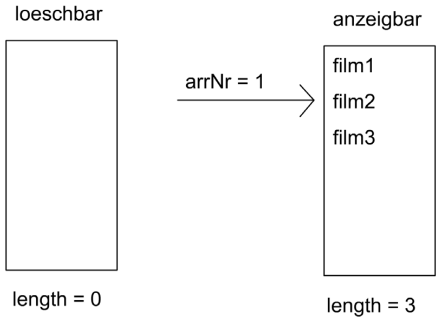


Abbildung 1.17:
Zustand der Arrays zu Beginn

Wie man in Abbildung 1.17 gut erkennt, enthält `anzeigbar` zu Beginn alle drei Instanznamen.

Für `arrNr` haben wir in diesem Beispiel den Wert 1 angenommen, dementsprechend würde es auf das zweite Array-Element verweisen. Theoretisch verweist `arrNr` natürlich nicht darauf, wir haben dies nur für uns definiert, da wir `arrNr` als „Zugriffszeiger" für das Array benutzen.

Nachdem Flash zufällig ein Movie ausgewählt hat, müssen wir dafür sorgen, dass dieses auch eingeblendet wird. Dafür braucht man den Instanznamen, um das Movie direkt ansprechen zu können. Dementsprechend greifen wir über den Inhalt von `arrNr` auf das Array `anzeigbar` zu und speichern den Instanznamen in einer Variablen, hier `movieName`.

```
movieName = anzeigbar[arrNr];
```

Jetzt kann man über den Instanznamen das Movie ansprechen und die Einblendesequenz abspielen lassen.

```
this[movieName].gotoAndPlay(2);
```

Man beachte, dass bei dem Befehl zwar auch die Array-Zugriffsoperatoren [ ] benutzt werden, es sich hierbei aber nicht um ein Array handelt! Es wird im Beispiel nur dafür gesorgt, dass der String, der in der Variablen `moviename` steht, als Pfadname benutzt wird. Man hätte an der Stelle auch eine If-Abfrage für alle drei Movies einbinden können, um so immer direkt film2.gotoAndPlay(2) zu schreiben. Eine weitere Alternative wäre der Befehl `eval( )`, der genau das Gleiche bewirkt.

```
eval(moviename).gotoAndPlay(2);
```

Nachdem wir das Movie eingeblendet haben, muss man die Zustände in den Arrays noch richtig setzen. Da der zweite Film nun eingeblendet ist und dadurch nicht mehr in dem Array `anzeigbar` stehen darf, sondern stattdessen im Array `loeschbar` mit eingetragen werden muss.

Das Eintragen des Films in `loeschbar` geht sehr leicht mit der Array-Methode `push( )`:

```
loeschbar.push(movieName);
```

Nach dem Ausführen dieses Befehls weisen die Arrays folgende Inhalte auf:

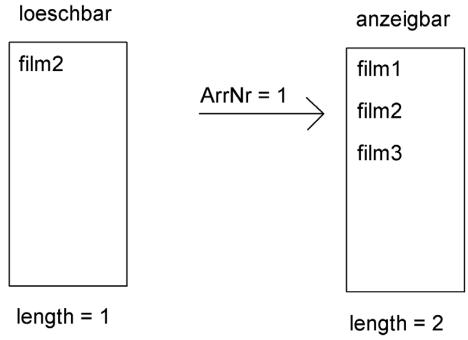


Abbildung 1.18: Array-Zustand während des Ausführens des Scripts

„film2“ muss also noch aus dem Array `anzeigbar` gelöscht werden. Dafür haben wir die Array-Methode `splice( )` benutzt. Als Parameter benötigt die Methode die Zahl des Array-Elements, also `arrNr` und die Anzahl der darauf zu löschenden Elemente. Da wir immer nur ein Element löschen möchten, ist diese 1.

```
anzeigbar.splice(arrNr,1);
```

Wenn dieser Befehl auch noch ausgeführt wurde, ist das ActionScript beendet. film2 wird angezeigt und die Zustände in den Arrays sind richtig zugeordnet.

Abbildung 1.19: Zustand des Beispiels, wenn nur der zweite Film angezeigt wird

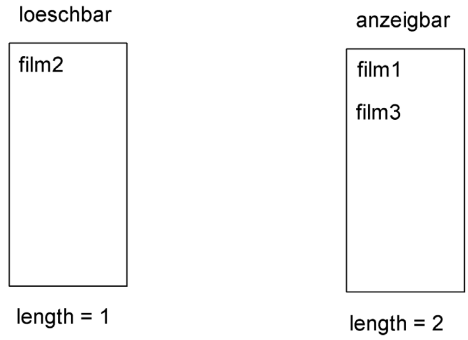


Abbildung 1.20: Passend zu Abbildung 1.19 die Zustände der Arrays

Im Folgenden noch einmal das komplette ActionScript auf einen Blick:

```
arrNr = random(anzeigbar.length); //zufälliges Movie aus den
    anzeigbaren auswählen
movieName = anzeigbar[arrNr]; //Moviename des Movies bestimmen
this[movieName].gotoAndPlay(2); //das Movie einblenden
loeschbar.push(movieName); //Den Movienamen in das
    löschbar-Array schreiben
anzeigbar.splice(arrNr,1); //Das eingeblendete Movie aus dem
    anzeigbar-Array entfernen
```

Falls der User öfter als dreimal auf den Button klickt, werden Felder mit leerem Inhalt in das `loeschbar`-Array eingefügt. Dies stört das ganze Konzept. Damit dies nicht passiert, überprüft man vor dem Ausführen des Skripts zuerst, ob noch ein Movie existiert, das eingeblendet werden kann. Alle anzeigbaren Movies stehen im Array `anzeigbar`. Falls dieses also kleiner als 1 ist, gibt es keine weiteren Movies mehr zum Einblenden, und das Script sollte nicht ausgeführt werden. Insgesamt sieht das Script des Buttons dann so aus:

Listing 1.75: Script des Buttons zum Einblenden eines Movies

```
on (release) {
    //movie einblenden
    if(anzeigbar.length >0){ //Nur Movie einblenden,
        wenn mindestens noch eins ausgeblendet ist
        arrNr = random(anzeigbar.length); //zufälliges Movie
            von den anzeigbaren auswählen
        movieName = anzeigbar[arrNr]; //Moviename des Movies
            bestimmen
        this[movieName].gotoAndPlay(2); //das Movie einblenden
        loeschbar.push(movieName); //Den Movienamen in das
            löschbar-Array schreiben
        anzeigbar.splice(arrNr,1); //Das eingeblendete Movie
            aus dem anzeigbar-Array entfernen
    }
}
```

Für alle diejenigen, die immer Wert auf ein möglichst kurzes Script legen, hier die gekürzte Version:

Listing 1.76: Gekürzte Version des Scripts aus dem Listing 1.75

```
on (release) {
    if(anzeigbar.length >0){
        arrNr = random(anzeigbar.length);
        this[anzeigbar[arrNr]].gotoAndPlay(2);
        loeschbar.push(anzeigbar.splice(arrNr,1));
    }
}
```

Das Skript für das Ausblenden eines Movies sieht analog zu diesem aus und benötigt keine ausführliche Beschreibung:

Listing 1.77: Script zum Ausblenden eines Movies

```
on (release) {
    //movie ausblenden
    if(loeschbar.length >0){ //Nur Movie ausblenden,
        wenn mindestens noch eins eingeblendet ist
        arrNr = random(loeschbar.length); //zufälliges Movie
            von den löschbaren auswählen
        movieName = loeschbar[arrNr]; //Moviename des Movies
            bestimmen
        this[movieName].gotoAndPlay(7); //das Movie einblenden
        anzeigbar.push(movieName);//Den Movienamen in das
            anzeigbar-Array schreiben
        loeschbar.splice(arrNr,1); //Das ausgeblendete Movie
            aus dem löschbar-Array entfernen
    }
}
```

Eine leicht geänderte Version zu dem Beispiel finden Sie auf der CD unter *Kapitel1/06_ANr.fla*.

Es sei erwähnt, dass die Movies über ein Tweening ein- und ausgeblendet werden. Viele sind der Ansicht, dass gerade in ActionScript-Büchern Tweenings gar nichts verloren haben, da es ohne weiteres auch möglich ist, ein kleines ActionScript zu erstellen, das über den Alphawert des jeweiligen Movies dieses ein- und ausblendet. Wir haben dies aus Gründen der Übersichtlichkeit und Verständlichkeit anders gelöst. Außerdem ist die Variante mit dem Tweening praktikabler, da man so die Movies nicht nur ein- oder ausblenden, sondern auch hereinfliegen lassen kann oder Ähnliches.

Mehrdimensionale Arrays

Wie ein eindimensionales Array aussieht, sollten Sie sich gut vorstellen können, denken Sie nur an die Metapher zum eindimensionalen Array mit dem Schrank, oder stellen Sie sich dafür eine einspaltige Tabelle vor. (Folgende ist nur wegen der Beschriftung zweispaltig.)

Arr (Arrayname)

Index	Inhalt
0	"Jan"
1	"Marc"
2	"Frank"
3	"Fabian"

Tabelle 1.10: Eindimensionales Array

Bei einem zweidimensionalen Array besitzt diese Liste mehrere Spalten, die über einen Index laufen.

Erweitern wir das Beispiel um Wohnort und Telefonnummer:

Arr2 (Arrayname)

Index1 \ Index2	0	1	2
0	"Jan"	"Wuppertal"	567988
1	"Marc"	"Kamen"	543809
2	"Frank"	"Unna"	543478
3	"Fabian"	"Magdeburg"	538064

Tabelle 1.11: Zweidimensionales Array

Der kürzeste Quelltext dazu sieht wie folgt aus

```
arr2=[["Jan","Wuppertal",567988],
["Marc","Kamen",543809],
["Frank","Unna",543478],
["Fabian","Magdeburg",538064]];
```

Dies ist zwar die Anwendung eines zweidimensionalen Arrays, dennoch ist es in diesem Fall angebracht, mit einem Objekt zu arbeiten, das die Variablen `Name`, `Ort` und `Telefonnummer` enthält. Diese Objekte sollten dann in einem eindimensionalen Array gespeichert werden. Für diese Art der Speicherung existiert die Sortiertfunktion `sortOn(Objektvariable)`. Dazu jedoch mehr im Kapitel 1.8 ab Seite 102. Ein zweidimensionales Array kann man beispielsweise für eine Straßenübersicht benutzen, bzw. für ein Spiel, um festzulegen, welche Felder betreten werden dürfen.

Dafür legt man vorab fest, dass Felder mit dem Inhalt kleiner 5 (beliebiger Wert) nicht betreten werden dürfen. Wir haben hier direkt fünf Unterscheidungen für nicht belegte Felder zugelassen, um so später das Spielfeld mit verschiedenen Objekten aufbauen zu können. Wenn man auf diese Art ein zweidimensionales Array mit der Größe von 10 * 10 Feldern erstellt, sieht dies eventuell wie folgt aus:

```
spielfeld=[
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[ 0, 6, 6, 6, 6, 6, 6, 6, 6, 0],
[ 0, 6, 0, 0, 0, 0, 0, 0, 6, 0],
[ 0, 6, 0, 0, 0, 0, 0, 0, 6, 0],
[ 0, 6, 0, 0, 0, 0, 0, 0, 6, 0],
[ 0, 6, 6, 6, 6, 6, 6, 6, 6, 0],
[ 0, 6, 0, 0, 0, 0, 0, 0, 6, 0],
[ 0, 6, 0, 0, 0, 0, 0, 0, 6, 0],
[ 0, 6, 6, 6, 6, 6, 6, 6, 6, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
];
```

Bei dieser Schreibweise erkennt man fast direkt im Quelltext, wie das Spielfeld später aussehen wird. 10 * 10 Felder, mit einer 6, auf der sich später jemand bewegen kann.

Das Abfragen von Werten aus zweidimensionalen Arrays erfolgt analog zu eindimensionalen. Während Sie dort `arr[indexnummer]` geschrieben haben, müssen Sie nun zwei Indexnummern angeben `arr[Indexnummer1][Indexnummer2]`.

Wenn man dies einmal auf das obere Spielfeld anwendet:

```
trace(spielfeld[0][0]);
trace(spielfeld[2][1]);
```

Beim ersten Array wird die linke obere Ecke ausgegeben – folglich eine 0. Es gilt der Merksatz, „Zeile zuerst, Spalte später". Beim zweiten Array wird die dritte Zeile und die zweite Spalte angesprochen; dementsprechend ergibt dies eine 6. Sie müssen beim Umgang mit Arrays immer daran denken, von 0 beginnend, also inklusive 0, zu zählen und nicht wie gewohnt bei 1.

So, wie man Felder abfragt, können natürlich auch einzelne Felder gesetzt werden:

```
spielfeld[5][5]=6;
```

Der Befehl würde in der sechsten Zeile und sechsten Spalte den Wert auf 6 setzen. Auf das obere Array bezogen, würde sich nichts ändern, da dieser Wert bereits den Wert 6 enthält.

Wenn Sie ein zweidimensionales Array mit den Konstruktoraufrufen erstellen wollen, evtl. ohne Werte zuzuweisen, geht dies natürlich auch.

```
spielfeld = new Array(3 ); //erzeugt ein eindimensionales Array
    mit der Länge 3
spielfeld[0]=new Array(3); // erzeugt in der ersten Zeile ein
    weiteres Array (die zweite Dimension) mit der Länge 3
spielfeld[1]=new Array(3); // analog zu 0 nur für Zeile 2
spielfeld[2]=new Array(3); // analog zu 0 nur für Zeile 3
```

So hätten Sie ein zweidimensionales Array erstellt, das 3*3 Felder groß ist. Mit Schleifen ist dies um einiges einfacher zu bewerkstelligen – mehr dazu im nächsten Kapitel.

Wenn Sie ein zwei- oder mehrdimensionales Array so erstellen, können Sie direkt auf die einzelnen Elemente zugreifen:

```
spielfeld[1][2]="Text"; //Zuweisung eines Strings zu Zeile 2
    Spalte 3
trace(spielfeld[1][2]);
```

gibt dementsprechend „Text" aus. Wenn Sie allerdings versuchen auf einen Teil des Arrays zuzugreifen, den Sie nicht definiert haben, funktioniert dies nicht. Flash erweitert zwar eindimensionale Arrays automatisch, dies geschieht aber bei mehrdimensionalen logischerweise nicht mehr.

```
spielfeld[5][2]="Text"; //Zuweisung eines Strings zu Zeile 6
    (die nicht existiert), Spalte 3
trace(spielfeld[5][2]);
```

Dies führt dementsprechend zu einem Fehler. Flash meldet keinen Syntaxfehler, da alles richtig interpretiert werden kann. Aber statt der Ausgabe des Strings „Text“ wird `undefined` ausgegeben, da die Zeile in dem Array nicht definiert wurde.

1.6 Wiederholungen (Schleifen)

Schleifen werden dazu benutzt, einen Quelltext entsprechend einer bestimmten Bedingung beliebig oft zu wiederholen. Wie im letzten Kapitel 1.5 angedeutet kann man dies unter anderem dazu benutzen, mehrdimensionale Arrays zu erstellen, da beim Erstellen von Arrays immer wieder fast derselbe Quelltext benötigt wird.

Schleifen werden innerhalb eines Frames ausgeführt! Wenn man also versucht, in einer Schleife einen Movieclip auf der Bühne zu bewegen _x++; (bewegt das Movie nach rechts) und dies in einer Schleife ausführt, wird man später beim Aufrufen des Scripts nur einen Sprung sehen und kein langsames Scrollen. Dies liegt daran, dass Schleifen innerhalb eines Frames ausgeführt werden, und so wird die Position immer wieder versetzt. Dies ist aber nicht sichtbar, da die Aktionen von Flash erst bei einem Framewechsel auf der Bühne aktualisiert werden. Für ein langsames Scrollen oder dergleichen muss man ein enterFrame-Event benutzen – dazu mehr in Kapitel 2.

Der Schleifenkopf muss immer aus einer Bedingung bestehen, die im Schleifenkörper geändert wird, so dass die Schleife nach einer gewissen Zeit terminiert (endet). Wenn eine Schleife nicht endet, es sich also um eine Endlosschleife handelt, kann die ausführende Umgebung (Flashplayer, Flash oder der Browser) abstürzen. Flash gibt aber auch in Fällen, bei denen mehr als 20000 Aktionen innerhalb eines Frames stattfinden, eine Anfrage an den User auf, ob das Script nicht besser abgebrochen werden solle.

Abbildung 1.21: Warnung von Flash, falls mehr als 20 000 Aktionen innerhalb eines Frames ausgeführt werden

while-Schleife

Die Syntax einer while-Schleife sieht wie folgt aus:

```
while (Bedingung) {
    Code();
}
```

Solange die Bedingung `true` ergibt, wird die Schleife ausgeführt. Wenn beim ersten Ausführen der Schleife die Bedingung sofort `false` ist, wird der Schleifeninhalt gar nicht ausgeführt!

Kommen wir noch einmal auf Kapitel 1.5, Abschnitt „Mehrdimensionale Arrays“ ab Seite 78 zu sprechen, dort ging es um das Erstellen von mehrdimensionalen Arrays. Um ein 3 mal 3 Felder großes Array zu erstellen, hatten wir folgenden Quelltext benutzt:

```
spielfeld = new Array(3 ); //erzeugt ein eindimensionales Array
    mit der Länge 3
spielfeld[0]=new Array(3); // erzeugt in der ersten Spalte ein
    weiteres Array (die zweite Dimension) mit der Länge 3
spielfeld[1]=new Array(3); // analog zu 0 nur für Spalte 2
spielfeld[2]=new Array(3); // analog zu 0 nur für Spalte 3
```

Wie man sieht, ist dies relativ viel Aufwand, gar nicht vorzustellen, wenn man ein 20 mal 20 Felder großes Array benötigt.

Dieses Problem lässt sich aber leicht mit einer Schleife lösen:

Listing 1.78: Erzeugt ein 20 mal 20 Felder großes Array

```
i=0;
spielfeld = new Array(20);
while(i<spielfeld.length){
    spielfeld[i]=new Array(spielfeld.length);
    i++;
}
```

Man sieht sofort den einfachen Lösungsansatz, dementsprechend gleich der Quelltext für ein dreidimensionales Array:

Listing 1.79: Erstellt ein dreidimensionales Array

```
i=0;
spielfeld = new Array(20);
while(i<spielfeld.length){
    spielfeld[i]=new Array(spielfeld.length);
    k=0;
    while(k<spielfeld.length){
        spielfeld[i][k]=new Array(spielfeld.length);
        k++;
    }
    i++;
}
```

Die Schleifen wurden einfach verschachtelt, dadurch wird der innen liegende Quelltext 20-mal ausgeführt, wegen der inneren Schleifen. Die komplette Schleife wird ebenfalls 20-mal ausgeführt.

Daher wird der Befehl `spielfeld[i][k]=new Array(spielfeld.length);` 40*40 also tausendsechshundertmal ausgeführt. Wenn Sie dieses Array ohne eine Schleife hätten erzeugen wollen, hätten Sie also 1600 Zeilen Code dafür benötigt!

do-while-Schleife

Eine do-while-Schleife ähnelt einer while-Schleife.

```
do {
    code();
} while (Bedingung);
```

Bei dieser Art von Schleife steht die Bedingung am Ende. Sie wird deshalb immer erst im Rumpf auf die Abbruchbedingung geprüft, was zur Folge hat, dass der Schleifenkörper, also der von der Schleife umfasste Quelltext, mindestens einmal ausgeführt wird.

Im Folgenden einmal eine direkte Gegenüberstellung von while- und do-while-Schleife

Listing 1.80: do-while-Schleife und while-Schleife

```
startwert=1;
i=startwert;
do {
    i++;
    trace("do-while");
} while (i<3);

i=startwert;
while (i<3){
    i++;
    trace("while");
}
```

Bildschirmausgabe:

```
do-while
do-while
while
while
```

Man sieht, dass beide Schleifen gleich funktionieren und deren Inhalte gleich oft ausgeführt werden. Der Unterschied der beiden Schleifentypen wird dann sichtbar, wenn man das Abbruchkriterium direkt auf false setzt, also den `startwert` erhöht.

Listing 1.81: do-while-Schleife und while-Schleife

```
startwert=3;
i=startwert;
do {
    i++;
    trace("do-while");
} while (i<3);

i=startwert;
while (i<3){
    i++;
    trace("while");
}
```

Bildschirmausgabe: `do-while`

In diesem Fall wird, obwohl die Bedingung von Anfang an `false` ist, der Inhalt der do-while-Schleife einmal ausgeführt.

for-Schleife

Die Bedingungsabfrage der for-Schleife ähnelt der while-Schleife, sprich die Abfrage findet zu Beginn statt und nicht am Ende.

```
for (Startaktion; Bedingung ;  Wiederholung){
    Code();
}
```

Anders als die bisherigen Schleifen enthält diese neben der Bedingung noch eine Startaktion und ein weiteres Feld für das wiederholte Ausführen von Aktionen. Wie Sie oben schon bei der while-Schleife gesehen haben, haben wir dort eine Variable eingeführt, die bei jedem Schleifen-Durchlauf um 1 erhöht wurde. Speziell für diesen Fall existiert aber eigentlich die for-Schleife.

Wir haben eben bei der while-Schleife die Variable `i` benutzt. Dies geschah nicht willkürlich, sondern ist ein ungeschriebener Standard, für solche Zähler- oder Indexvariablen die Namen `i, j, k... i2, j2, k2` usw. zu verwenden.

Während man diese Variablen bei einer while-Schleife teilweise außerhalb o setzen und innerhalb erhöhen muss, werden diese bei einer for-Schleife oben im Kopf mit angegeben.

```
for( i=0; i< Wert; i++){
    code();
}
```

Dies ist die Art von Schleife, die am häufigsten Verwendung findet.

Angenommen, Sie wollen das Ergebnis von 5^6 ausrechnen. Selbstverständlich bietet Flash hierfür mit `pow()` eine Methode aus dem Matheobjekt an, aber wir wollen darauf bewusst verzichten und das Ergebnis mit Hilfe einer for-Schleife erzeugen.

Als Erstes benötigt man zwei Variablen für die Basis und den Exponenten. Wir weisen diesen einen festen Wert zu – man hätte diese auch wie in den vorangegangenen Beispielen über Eingabefelder einlesen können. Vergessen Sie dabei nicht, die eingelesenen Werte von String nach Number zu casten! Hier also die zwei Variablen:

```
basis = 5;

exponent = 6;
```

Die Ausgabe des Ergebnisses erfolgt wieder mit Hilfe von `trace()`. Sicherlich wäre es auch direkt möglich:

```
trace(5* 5* 5* 5* 5* 5);
```

zu schreiben, aber das war nicht der Sinn der Übung. Wenn man 5^6 rechnet, muss man die Zahl 5 sechs Mal mit sich selbst multiplizieren.

Listing 1.82: Berechnungsbeispiel für 5^6

```
basis = 5;
exponent = 6;
ergebnis=basis;
for (i=1;i<exponent;i++){
    ergebnis*=basis;
}
trace(ergebnis);
```

Bildschirmausgabe: `15625`

Die Schleife würde bei jedem Durchlauf den Wert des Ergebnisses mit der Basis multiplizieren und dies der Größe des Exponenten entsprechend.

Der Wert des Ergebnisses würde sich, der Schleifendurchläufe entsprechend, wie folgt ändern:

```
25
125
625
3125
15625
```

Jedes Mal wenn man auch eine Zählervariable benötigt, sollte man eine for-Schleife einsetzen. Hier noch einmal der Quelltext zum Erstellen eines dreidimensionalen Arrays:

Listing 1.83: Erstellt ein dreidimensionales Array mit Hilfe von while-Schleifen

```
i=0;
spielfeld = new Array(20);
while(i<spielfeld.length){
    spielfeld[i]=new Array(spielfeld.length);
    k=0;
    while(k<spielfeld.length){
        spielfeld[i][k]=new Array(spielfeld.length);
        k++;
    }
    i++;
}
```

Zum direkten Vergleich hier noch einmal die gleiche Funktion nur mit for-Schleifen:

Listing 1.84: Erstellt ein dreidimensionales Array mit Hilfe von for-Schleifen

```
spielfeld = new Array(20);
for(i=0;i<spielfeld.length,i++){
    spielfeld[i]=new Array(spielfeld.length);
    for(k=0; k<spielfeld.length; k++){
        spielfeld[i][k]=new Array(spielfeld.length);
    }
}
```

for-in-Schleife

Ein for-in-Schleife läuft über die Eigenschaften eines Objektes. Objekte und Klassen werden in Kapitel 1.8 beschrieben. Sie können sich mit dieser Schleife alle Eigenschaften eines Objektes auflisten lassen.

Im Folgenden lassen wir uns alle Eigenschaften des Movies in `_level0` anzeigen:

Listing 1.85: Gibt alle Eigenschaften des aktuellen Objektes aus

```
for (var  in this) {
    trace(a + " = "+ this[a]);
}
```

Bildschirmausgabe: `$version = WIN 6,0,21,0`

Da es zu dem Zeitpunkt keine Variablen gibt, wird nur die Standardvariable, die Versionsnummer, aufgelistet. Wenn wir vorher allerdings Variablen initialisieren oder deklarieren, werden diese auch mit aufgeführt:

Listing 1.86: Gibt alle Eigenschaften des aktuellen Objektes aus

```
auto= true;
autofarbe= "grün";
autoanzahl = new Number();
for (a in this) {
    trace(a + " = "+ this[a]);
}
```

Bildschirmausgabe:

```
autoanzahl = 0
autofarbe = grün
auto = true
$version = WIN 6,0,21,0
```

break

Mit `break` kann man eine Schleife vorzeitig beenden. Wie man allerdings bei der Switch-Case-Anweisung gesehen hat, wird es nicht nur für die Schleifen benutzt. Switch-Case bildet allerdings hier auch die einzige Ausnahme. Man sollte es vermeiden, zu viele breaks in Schleifen zu benutzen, da es den Programmablauf unübersichtlicher gestaltet. Meistens lässt sich das Anwenden von `break` durch geschickte Programmierung umgehen.

Kommen wir noch einmal auf das Beispiel von Seite 85 zurück:

Listing 1.87: Berechnungsbeispiel für 5^6

```
basis = 5;
exponent = 6;
ergebnis=basis;
for (i=1;i<exponent;i++){
    ergebnis*=basis;
}
trace(ergebnis);
```

Angenommen, es sind für Sie nur Ergebnisse kleiner als 1000 relevant, sprich sobald ein Ergebnis größer als 1000 wird, ist es egal, ob 2000 oder 1 000 000.

Um Rechenzeit zu sparen, können Sie die Schleife in diesem Fall abbrechen.

Listing 1.88: Berechnungsbeispiel für Potenzen mit einem Ergebnis kleiner oder gleich 1000

```
basis = 5;
exponent = 6;
ergebnis=basis;
for (i=1;i<exponent;i++){
    ergebnis*=basis;
    if(ergebnis >1000){
        break;
        }
}
trace(ergebnis);
```

Die Schleife läuft normal, das Ergebnis nimmt wieder die folgenden Zustände an:

```
25
125
625
3125
```

Doch hier wird die Schleife durch das `break` abgebrochen, da das Ergebnis größer als 1000 ist. So erfolgt die letzte Rechnung, die als Ergebnis 15625 hat; nicht mehr.

continue

Das Gegenteil von `break` ist nicht `continue`. Wie sollte es auch funktionieren, soll eine Schleife, die komplett beendet, ist wieder von vorne starten?

Wir werden die Funktionsweise von `continue`, der von `break` gegenüberstellen.

Hier noch einmal das Beispiel mit dem `break`, dieses Mal wird zusätzlich die Anzahl der Schleifendurchläufe mit ausgegeben:

Listing 1.89: Lösung mit break

```
basis = 5;
exponent = 6;
ergebnis=basis;
for (i=1;i<exponent;i++){
trace(i+" Schleifendurchlauf");
    ergebnis*=basis;
    if(ergebnis >1000){
        break;
    }

}
trace(ergebnis);
```

Bildschirmausgabe:

```
1 Schleifendurchlauf
2 Schleifendurchlauf
3 Schleifendurchlauf
4 Schleifendurchlauf
3125
```

Das gleiche Beispiel nur mit `continue`:

Listing 1.90: Lösung mit continue

```
basis = 5;
exponent = 6;
ergebnis=basis;
for (i=1;i<exponent;i++){
trace(i+" Schleifendurchlauf");
    if(ergebnis >1000){
        continue;
        }
    ergebnis*=basis;
}
trace(ergebnis);
```

Bildschirmausgabe:

```
1 Schleifendurchlauf
2 Schleifendurchlauf
3 Schleifendurchlauf
4 Schleifendurchlauf
5 Schleifendurchlauf
3125
```

Wie Sie sehen, wurde auch bei der Lösung mit `continue` die Berechnung nicht mehr ausgeführt, wohl aber der trace-Befehl. Dies liegt daran, dass ein `continue` dafür sorgt, dass der nachfolgende Inhalt der Schleife übersprungen wird und die Schleife weiter normal durchläuft.

Man sieht, dass `break` und `continue` nichts gemeinsam haben, wenn man davon absieht, dass beide die Eigenschaften einer Schleife ändern. Während `break` also dafür sorgt, dass eine Schleife sofort beendet wird, überspringt der Interpreter, wenn er auf ein `continue` stößt; nur den Rest der Schleife und kehrt wieder zum Schleifenanfang zurück. (Die Schleife wird dadurch nicht neu gestartet, die Laufvariablen werden normal weiter erhöht und nicht neu auf 0 gesetzt oder dergleichen.)

1.7 Funktionen

Funktionen funktionieren ähnlich wie Schleifen, denn auch Funktionen dienen grundlegend dazu ein und den selben Quelltext mehrfach auszuführen. Funktionen bieten hierbei allerdings einen Vorteil: Während bei den Schleifen der Quelltext, der wiederholt wird, immer direkt hintereinander ausgeführt wird, haben Sie bei Funktionen den Vorteil, an beliebigen Stellen in Ihrem Programm diese genauso oft aufzurufen, wie Sie sie benötigen.

Gehen wir einmal näher auf die Syntax ein: Jede Funktion besitzt einen Namen, wie eine Variable. Wenn man die Funktion an einer späteren Stelle in seinem Script wieder aufrufen möchte, schreibt man einfach den Namen der Funktion und zwei Klammern () dahinter.

```
function Funktionsname ( ){
    code( );
}
```

So sieht eine Funktionsdeklaration aus. Der Aufruf dieser Funktion geschieht dann über den Funktionsnamen:

```
Funktionsname( );
```

Im Folgenden ein kleines Beispiel:

```
function Ausgabe(){
    trace("Hallo Welt");
    }
```

Listing 1.91: Die Funktion Ausgabe wird erstellt.

Wenn man dieses Script so aufruft, passiert gar nichts, da nur die Funktion erstellt, aber nicht aufgerufen wird. Damit auf dem Bildschirm „Hallo Welt“ ausgegeben wird, muss man die Funktion aufrufen.

```
function Ausgabe(){
    trace("Hallo Welt");
    }

ausgabe();
ausgabe();
```

Listing 1.92: Erstellen und zweimaliges Aufrufen der Funktion Ausgabe

Bildschirmausgabe:

```
Hallo Welt
Hallo Welt
```

Man sieht, dass man die Funktion beliebig oft aufrufen kann.

Parameter

Parameter nennt man Variablen, die man an Funktionen übergibt. Man übergibt Variablen an Funktionen, indem man beim Funktionsaufruf die entsprechenden Variablen oder direkt die Werte in die Klammern des Funktionsaufrufes schreibt.

Damit man diese Variablen aber direkt in der Funktion ansprechen kann, muss bei der Definition der Funktion schon festgelegt werden, wie die übergebenen Variablen später in der Funktion heißen sollen.

Funktionen können auch auf die sie umgebenden Variablen zugreifen:

Listing 1.93: Ausführliches Beispiel zur Variablenübergabe an Funktionen

```
wert1="Anfangswert";
wert2="Anfangswert2";
function FKT1(wert1){
    trace("FKT1 wert1 : "+ wert1 + " |this.wert1 : " +
this.wert1);
    trace("FKT1 wert2 : "+ wert2 + " |this.wert2 : " +
this.wert2);
}

wert2="geänderter Anfangswert2";
FKT1("Übergebene Variable");
```

Bildschirmausgabe:

```
FKT1 wert1 : Übergebene Variable |this.wert1 : Anfangswert
FKT1 wert2 : geänderter Anfangswert2 |this.wert2 : geänderter
Anfangswert2
```

Zuerst werden die Variablen `wert1` und `wert2` mit Strings belegt. Direkt danach wird die Funktion `FKT1` definiert. Dieser wird die in Klammern geschriebene Variable als Parameter übergeben. Die Funktion selbst erzeugt nur eine kleine Bildschirmausgabe. Der Wert der Variablen `wert2` wird anschließend noch einmal geändert und dann wird die Funktion aufgerufen.

Vielleicht sollten wir noch kurz auf das Schlüsselwort `this` eingehen. Es verweist auf das aktive Objekt. Bisher haben wir noch nichts zum Gültigkeitsbereich bzw. das Scoping von Variablen geschrieben. Eine genaue Beschreibung dazu finden Sie in Kapitel 2.

Ansonsten schauen Sie sich einmal das Beispiel genau an.

Der Funktion wird als Parameter ein String übergeben, der dann in der Variablen `wert1(Funktionslokal)` gespeichert wird. Wie man an dem Beispiel sieht, existiert die Variable `wert1` bereits auf der Hauptzeitleiste.

In einem solchen Fall spricht man vom Verdecken. Wenn man sich in der Funktion befindet, verdeckt die Variable `wert1` die schon vorhandene Variable `wert1` auf der Hauptbühne.

Mit `this` gelangt man in so einem Fall auf die außen liegende Variable, wie das Beispiel auch zeigt:

```
trace("FKT1 wert1 : "+ wert1 + " |this.wert1 : " + this.wert1);
```

Der Befehl hatte folgende Ausgabe zur Folge:

```
FKT1 wert1 : Übergebene Variable |this.wert1 : Anfangswert
```

Mit `wert1` wurde also die Variable, die an die Funktion mit übermittelt wurde, ausgegeben, und mit `this.wert1` wurde die Variable auf der Hauptzeitleiste bzw. in dem aktuellen Movie (also dem aktuellen Objekt) ausgegeben.

```
trace("FKT1 wert2 : "+ wert2 + " |this.wert2 : " + this.wert2);
```

Die zweite Zeile, die für eine Ausgabe sorgt, soll nur verdeutlichen, dass man, wenn man auf Variablen zugreifen möchte, die außerhalb der Funktion definiert wurden, nicht zwingend mit `this` darauf zugreifen muss. Zudem zeigt es, dass auf die aktuell gültigen Variablenwerte zugegriffen wird und nicht auf diejenigen, die beim Zeitpunkt der Funktionsdefinition gültig waren. Im Übrigen ist es auch vollkommen egal, an welcher Stelle in Ihrem Script Sie die Funktion erstellen, so könnte diese z.B. auch erst ganz am Ende erstellt werden:

```
wert1="Anfangswert";
wert2="Anfangswert2";
wert2="geänderter Anfangswert2";
FKT1("Übergebene Variable");
function FKT1(wert1){
    trace("FKT1 wert1 : "+ wert1 + " |this.wert1 : " +
          this.wert1);
    trace("FKT1 wert2 : "+ wert2 + " |this.wert2 : " +
          this.wert2);
}
```

Rückgabewert

Jetzt wo Sie wissen, wie Sie Werte an Funktionen übermitteln, können Sie sich auch denken, dass es einen Weg zurück gibt. Funktionen können nämlich auch Werte zurückgeben – allerdings immer nur einen. Falls man unbedingt mehrere Ergebnisse zurückgeben muss, kann man ein Array oder ein Objekt zurückgeben.

Das Schlüsselwort zum Zurückgeben von Werten lautet `return` und wird gefolgt von einem Wert, einer Variable oder einem Objekt, das man zurückgeben möchte.

Wenn man zum Beispiel eine Funktion erstellen möchte, die zurückgibt, ob eine Zahl gerade oder ungerade ist, bekommt man als Parameter eine Zahl und als Rückgabewert gibt man `true` oder `false` aus.

Auf die Problematik, ob eine Zahl gerade oder ungerade ist, sind wir schon in Kapitel 1.3 im Abschnitt „Numerische (arithmetische) Operatoren“ ab Seite 37 beim Modulo-Operator eingegangen. Man muss einfach nur auf die entsprechende Zahl ein Modulo 2 anwenden und erhält als Ergebnis 1 oder 0, das falls nötig, problemlos in `true` oder `false` gecastet werden kann.

Listing 1.94: Funktion, um eine Zahl auf gerade oder ungerade zu überprüfen

```
function istGerade(wert1) {
    var ergebnis = wert1 % 2;
    if (ergebnis == 0) {
        return true;
    } else {
        return false;
    }
}
trace(istGerade(4));
trace(istGerade(5));
trace(istGerade(6));
trace(istGerade(7));
```

Bildschirmausgabe:

```
true
false
true
false
```

Vielleicht hätten Sie diese auch so erstellt. Es ist allerdings auch möglich direkt die Operation bei der Rückgabe mit anzugeben, so dass dort erst das Ergebnis berechnet wird.

Listing 1.95: Gekürzte istGerade-Funktion

```
function istGerade(wert1) {
    return Boolean(!wert1 % 2);
}
trace(istGerade(4));
trace(istGerade(5));
trace(istGerade(6));
trace(istGerade(7));
```

Bildschirmausgabe:

```
true
false
true
false
```

Man hätte nicht unbedingt mit `Boolean` den Ausdruck noch casten müssen, da Flash das Ergebnis auch automatisch casten würde, wenn dies für eine Bedinung benutzt wird.

Sobald eine Funktion mit `return` einen Wert zurück gibt, endet diese Funktion auch automatisch.

```
function istGerade(wert1) {
    return Boolean(!wert1 % 2);
    trace("Wird nicht mehr ausgeführt");
}
istGerade(8);
```

So hätte dieses ActionScript keine Ausgabe zur Folge.

Gültigkeitsbereich (Scoping)

Bei dem Beispiel in Kapitel 1.7 im Abschnitt „Parameter“ ab Seite 90 sind wir schon kurz auf die Gültigkeitsbereiche von Variablen eingegangen. So hatten wir dort schon erwähnt, dass man aus Funktionen heraus auf die umliegenden Variablen zugreifen kann. Falls die Variablennamen verdeckt sind, kann man in dem Fall das Schlüsselwort `this` benutzen, um auf die verdeckten Variablen zuzugreifen.

Wir sind aber zum Beispiel nicht auf die Lebensdauer der als Parameter übergebenen Variablen eingegangen. Dies existiert nämlich nur in der Funktion und wird nach Beenden derselben gelöscht.

```
function FKT1(wert1) {
    trace("Ausgabe in der Funktion: "+wert1);
}
FKT1("Ein beliebiger String");
trace("Ausgabe nach Aufruf der Funktion: "+wert1);
```

Listing 1.96: Variablen, die als Parameter übergeben werden, existieren nur so lange wie die Funktion.

Bildschirmausgabe:

```
Ausgabe in der Funktion: Ein beliebiger String
Ausgabe nach Aufruf der Funktion:
```

Natürlich kann man auch innerhalb von Funktionen Variablen erstellen:

```
function FKT1() {
    wert1="Ein beliebiger String";
    trace("Ausgabe in der Funktion: "+wert1);
}
FKT1();
trace("Ausgabe nach Aufruf der Funktion: "+wert1);
```

Listing 1.97: Erstellte Variablen in Funktionen sind auch nach dem Ende der Funktion weiterhin existent.

Bildschirmausgabe:

```
Ausgabe in der Funktion: Ein beliebiger String
Ausgabe nach Aufruf der Funktion: Ein beliebiger String
```

Diese Variablen sind auch noch nach Beendigung der Funktion, in dem Objekt, in dem die Funktion erstellt wurde, bekannt.

Um Variablen nur funktionslokal zu erstellen, benutzen Sie das Schlüsselwort `var`. Damit erstellte Variablen werden funktionslokal gehalten wie zuvor schon die Parametervariablen.

```
wert1="Der alte Wert";
function FKT1() {
    var wert1="Ein beliebiger String";
    trace("Ausgabe in der Funktion: "+wert1);
    trace("Ausgabe in der Funktion mit this: "+this.wert1);
}
FKT1();
trace("Ausgabe nach Aufruf der Funktion: "+wert1);
```

Listing 1.98: Erstellung von lokalen Variablen in Funktionen

Bildschirmausgabe:

```
Ausgabe in der Funktion: Ein beliebiger String
Ausgabe in der Funktion mit this: Der alte Wert
Ausgabe nach Aufruf der Funktion: Der alte Wert
```

Zum einen zeigt das Beispiel, wie man mit `var` funktionslokale Variablen erstellt, zum anderen wie man mit `this` auf die verdeckten Variablen zugreift.

Zu erkennen ist auch, dass die beiden Variablen mit dem Namen `wert1` während des Aufrufs der Funktion parallel existieren und sich nicht beeinflussen.

Kommen wir nun zu einer Sache die sich in Flash MX im Vergleich zu Flash 5 geändert hat. Mit `this` greift man normalerweise immer auf das augenblicklich aktive Objekt zu (oder anders gesagt, auf das Objekt, in dem man sich gerade befindet). Funktionen stellen allerdings normalerweise keine Objekte dar. Dies hatte unter Flash 5 zur Folge, dass man sich mit `this` immer auf das Objekt bezog, aber wenn man z.B. eine Funktion in der anderen erstellt nicht auf die Variablen der äußeren Funktion zugreifen kann.

Am besten schaut man sich dazu direkt einmal den Quellcode an:

Listing 1.99: Beispiel für Funktionsdefinitionen in Funktion

```
wert1 = "Wert1 auf _level0";
function FKT1 (wert1) {
    trace ("FKT1 wert1 : "+wert1+" |this.wert1 : "+this.wert1);
    function FKT2 () {
        trace ("FKT2 wert1 : "+wert1+" |this.wert1 : "
               +this.wert1);
    }
    FKT2();
    zweiteFunktion = FKT2;
}
FKT1("Übergebener Parameter an FKT1");
zweiteFunktion();
wert1 = "Neu gesetzter Wert1 auf _level0";
zweiteFunktion();
```

Man erkennt, dass die Funktion `FKT2` in der Funktion `FKT1` erstellt wird. Um von außen Zugriff auf die innenliegende Funktion zu haben, mussten wir eine zusätzliche Variable `zweiteFunktion` mit einer Referenz auf die Funktion `FKT2` erstellen.

Ausgabe von Flash 5:

```
FKT1 wert1 : Übergebener Parameter an FKT1 |this.wert1 : Wert1
auf _level0
FKT2 wert1 : Wert1 auf _level0 |this.wert1 : Wert1 auf _level0
FKT2 wert1 : Wert1 auf _level0 |this.wert1 : Wert1 auf _level0
FKT2 wert1 : Neu gesetzter Wert1 auf _level0 |this.wert1 : Neu
gesetzter Wert1 auf _level0
```

In der Ausgabe von Flash 5 sieht man, dass die erste Ausgabe von FKT1 wie erwartet aussieht. Mit wert1 greift man auf den übergebenen Parameter zu, mit this.wert1 auf die Variable im umliegenden Objekt. Die Ausgabe von FKT2 sollte Ihnen aber zu denken geben. Alle Werte, die ausgegeben werden, sind die von _level0, die des aktuellen Objektes. Es ist nicht möglich, an die Werte der Variablen von FKT1 heranzukommen, obwohl FKT2 in FKT1 liegt.

Dies wurde nun in Flash MX behoben: So kennt die Funktion FKT2 alle Variablen aus FKT1. Mit this verweist man dann immer auf das Objekt, von dem die jeweilige Funktion aufgerufen wurde. Letzteres erkennt man an der zweiten Zeile, die Funktion wurde aus der zweiten Funktion aufgerufen und gibt mit this.wert1 die Variable aus FKT1 an. In der vorletzten und letzten hingegen wurde FKT2 von _level0 aus aufgerufen und spricht somit mit this.wert1 auch die Variable auf _level0 an.

Ausgabe von Flash MX:

```
FKT1 wert1 : Übergebener Parameter an FKT1 |this.wert1 :
Wert1 auf _level0
FKT2 wert1 : Übergebener Parameter an FKT1 |this.wert1 :
Übergebener Parameter an FKT1
FKT2 wert1 : Übergebener Parameter an FKT1 |this.wert1 :
Wert1 auf _level0
FKT2 wert1 : Übergebener Parameter an FKT1 |this.wert1 :
Neu gesetzter Wert1 auf _level0
```

Man erkennt, dass aufgerufene Funktionen unter Flash ähnlich wie Objekte behandelt werden. Selbst wenn die eigentliche Funktion FKT1 schon beendet ist, kann FKT2 noch auf die Variablen von FK1 zugreifen.

Rekursion

Von einer Rekursion spricht man, wenn eine Funktion sich selbst wieder aufruft.

In den meisten Büchern wird eine Schleife rekursiv nachgebildet. Allerdings sei erwähnt, dass dies nicht der eigentliche Sinn der Rekursion ist. Wenn das alles wäre, könnte man genauso gut eine Schleife benutzen, dies wäre noch um einiges übersichtlicher.

Um Ihnen den logischen Ansatz der Rekursion verständlich zu machen, hier die for-Schleife, die wir schon in Kapitel 1.6 im Abschnitt „for-Schleife“ ab Seite 84 beschrieben haben.

Listing 1.100: Potenzberechnung mit einer for-Schleife

```
basis = 5;
exponent = 6;
ergebnis=basis;
for (i=1;i<exponent;i++){
    ergebnis*=basis;
}
trace(ergebnis);
```

Das gleiche nur mit einem rekursiven Ansatz.

Listing 1.101: Potenzberechnung durch Rekursion

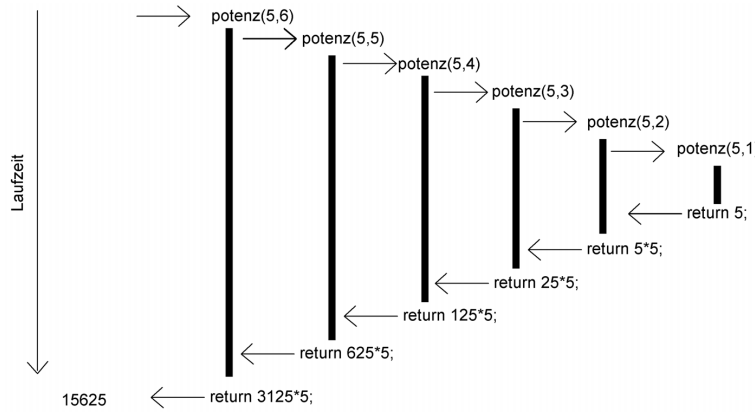
```
function potenz(basis, exponent){
    if(exponent>1){
        return basis* potenz(basis, exponent-1 );
    }
    return basis
}
trace(potenz(5,6));
```

Die rekursive Version muss nicht komplizierter oder länger aussehen, allerdings dürfte es schwieriger sein, sie zu verstehen.

Wir haben eine Funktion erstellt mit dem Namen `potenz`, diese benötigt zwei Parameter, die Basis und den Exponenten. Danach erfolgt sofort eine Abfrage, ob der Exponent größer als 1 ist. Da dies zutrifft, wird der If-Anweisungsblock ausgeführt. Dieser besitzt direkt eine return-Anweisung, welche allerdings nicht sofort ausgeführt werden kann, da der Rückgabewert erst errechnet werden muss. Zur Berechnung wird wieder die Funktion aufgerufen. Allerdings mit einem um 1 kleineren Exponenten.

Dies geschieht so lange, bis der Exponent nicht mehr größer als 1 ist. Für diesen Fall würde dann nicht der If-Anweisungsblock ausgeführt, sondern stattdessen der Wert der Variablen `basis` zurückgegeben. Dadurch lässt sich das Ergebnis der vorherigen Rechnung vervollständigen, da die letzte Funktion einen fest definierten `return`-Wert, nämlich die Basis hat. Sobald also die letzte Funktion den Wert der Basis zurückliefert und keine neue Funktion aufruft, lösen sich alle verschachtelten Funktionen nacheinander wieder auf. Die folgende Grafik soll dies verdeutlichen.

Abbildung 1.22: Berechnungsverlauf der rekursiven Lösung



Für diejenigen, die lieber Fakultäten mögen:

```
function fac( n )
{
   if( n > 1 ) return n * fac(n-1);
   return 1;
}
trace(fac(6));
```

Listing 1.102: Berechnung von Fakultäten durch Rekursion

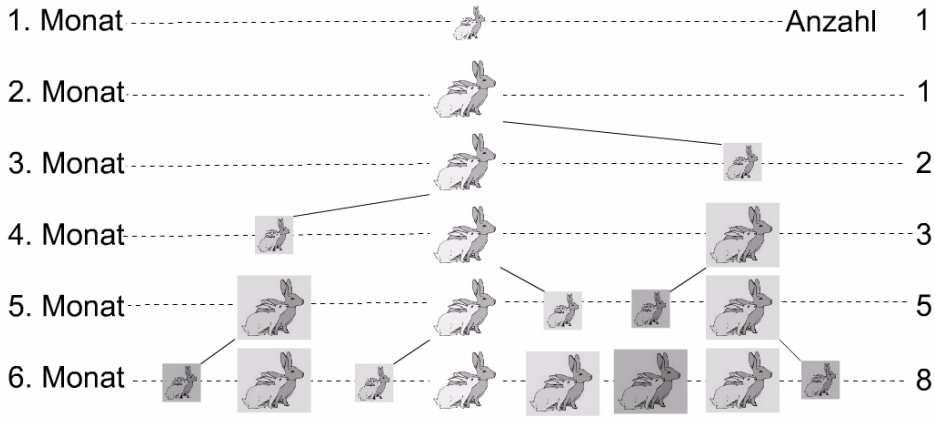
Solche Dinge lassen sich aber auch ohne Probleme mit for-Schleifen lösen. Der eigentliche Trick der Rekursion tritt erst dann ein, wenn man wirklich beim Lösen in Bäume verzweigen muss, da man dann auf den einzelnen Ebenen ohne Problem Werte im Speicher halten kann.

Dies ist auch gleichzeitig der größte Nachteil. Wie man an dem Bild erkennt, ruft die Rekursion immer wieder neue Abbilder der Funktion auf – Stück für Stück wird immer mehr Speicher belegt. Bei großen Projekten sollten Sie den enormen Speicherbedarf berücksichtigen.

Ein vernünftiges Beispiel für die Rekursion ist das Traversieren eines Baumes. Leider würde das Beispiel hier jegliche Rahmen sprengen, deshalb beschränken wir uns auf ein Problem aus dem Jahre 1202, dessen Lösung mit Hilfe der Fibonaccis-Zahlenreihe gelöst wird.

Die Rätselaufgabe war folgende: Wenn man ein Paar Hasen (Männchen und Weibchen) unter folgenden Annahmen aussetzt, wie viele Hasen existieren dann nach einem Jahr?

- Jedes neu geborene Paar braucht zwei Monate bis zum ersten Wurf. Danach werfen Sie monatlich.
- Jeder Wurf enthält genau ein Paar (Männchen und Weibchen).
- Jedes Hasenpaar braucht einen Monat bis zur Geschlechtreife. Erst im zweiten Monat ist das erste Hasenpaar Geschlechtsreif und kann so erst im dritten Monat ein weiteres Hasenpaar werfen, welches aber wiederum erst im 4 Monat geschlechtsreif ist und somit erst im fünften Monat ein weiteres Paar werfen kann.



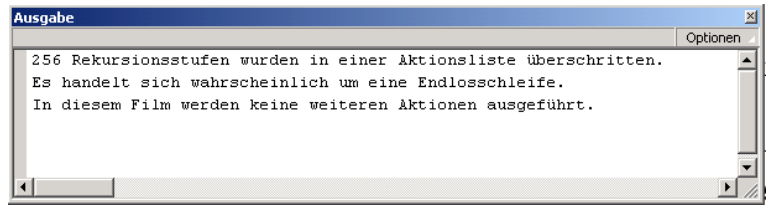
Wie man sieht, ruft die Funktion sich nicht nur einmal selbst wieder auf, sondern gleich zweimal. Ansonsten ähnelt das Ganze den voran gegangenen Beispielen.

Listing 1.103: Fibonacci-Funktion

```
function fib( n )
{
   if( n < 1 ) return 1;
   return fib(n-1)+fib(n-2) ;
}
monat=12;
trace(fib(monat-1));
```

Flash MX bietet nicht nur neue Befehle, sondern auch neue Fehlermeldungen, wenn man mehr als 256 rekursive Aufrufe tätigt.

Abbildung 1.23: Nach 256 Rekursionsstufen wird automatisch abgebrochen.



Methoden

Flash MX bietet im Gegensatz zu Flash 5 zwei Methoden für Funktionen an. Da jede Funktion in Flash MX durch ein Objekt der Klasse Funktion dargestellt wird, besitzen alle Funktionen diese beiden Methoden.

▶ apply()

Eine Funktion unter Flash 5 aufzurufen, wenn der Funktionsname nur als String bekannt ist, ist nicht möglich. Selbst die besten Versuche mit `eval` scheitern.

```
NameDerFunktion = ("FKT1");

eval(NameDerFunktion+"()"); //funktioniert nicht
```

Damit dies nicht weiterhin unmöglich ist, bietet Flash MX die Methode `apply()` an:

Listing 1.104: Anwendung der Methode apply()

```
NameDerFunktion = eval("FKT1");
function FKT1(wert1,wert2) {
    trace("FKT1 wurde ausgeführt");
}

NameDerFunktion.apply();
```

Bildschirmausgabe: `FKT1 wurde ausgeführt`

Man wendet `eval` auf den bestehenden String an und wendet danach die Methode `apply` darauf an. So wird die jeweilige Funktion, sofern sie existiert, aufgerufen.

Es ist möglich, `apply` zwei Parameter mit auf den Weg zu geben. Der erste gibt an, aus welchem Objekt diese aufgerufen wird (dementsprechend ändert sich der Inhalt der Variable `this` in der Funktion), der zweite Parameter ist ein Array mit Werten, der der aufgerufenen Funktion als Parameter übergeben wird.

Listing 1.105: Aufruf der Funktion aus dem aktuellen Objekt (this)

```
wert3="schön dass es euch gibt!";
NameDerFunktion = eval("FKT1");

function FKT1(wert1,wert2) {
    trace("FKT1 wurde ausgeführt mit den Werten: "+wert1
    +" "+wert2+" " + this.wert3);
}
NameDerFunktion.apply(this, ["Hallo","Welt,"]);
```

Bildschirmausgabe:

```
FKT1 wurde ausgeführt mit den Werten: Hallo Welt, schön dass es
euch gibt!
```

Die Funktion konnte problemlos mit `this` auf die Variable in der Zeitleiste zugreifen. Dies hätte die Funktion zwar auch ohne `this` gekonnt, aber es sollte hier im Beispiel zeigen, das `this` dementsprechend auf das aktuelle Objekt verweist. Im nächsten Beispiel wird als Parameter für das Objekt, von dem die Funktion aus gestartet wird, `null` angegeben. Deshalb kann man in der Funktion auch nicht mit `this` auf die Variable `wert3` zugreifen.

Listing 1.106: Aufruf der Funktion aus einem nicht definierten Objekt (null)

```
wert3="schön dass es euch gibt!";
NameDerFunktion = eval("FKT1");

function FKT1(wert1,wert2) {
    trace("FKT1 wurde ausgeführt mit den Werten: "+wert1
    +" "+wert2+" " + this.wert3);
}
NameDerFunktion.apply(null, ["Hallo","Welt,"]);
```

Bildschirmausgabe:

```
FKT1 wurde ausgeführt mit den Werten: Hallo Welt,
```

▶ call()

Wie bei der Methode `apply()` bietet dieser Befehl die Möglichkeit, eine Funktion auf ein beliebiges Objekt zu legen bzw. diese aus einem beliebigen Objekt heraus zu starten. Um den Umweg über den String als Funktionsnamen zu vermeiden, existiert diese Funktion. Sie ermöglicht das Aufrufen einer bekannten Funktion unter Angabe eines neuen Bestimmungsortes.

Listing 1.107: Beispiel für die Methode call()

```
function MeinObjekt() {
}
function FKT1(obj) {
  trace("this == obj ? " + (this == obj));
}
var obj = new MeinObjekt();
FKT1.call(obj, obj);
```

Bildschirmausgabe: `this == obj ? true`

In dem Beispiel wird zuerst ein Objekt erzeugt, `MeinObjekt`, mit Namen `obj` bzw. die Variable `obj` referenziert ein Objekt des Typs `MeinObjekt`. Die erstellte Funktion FKT1 wird unter Verwendung der Methode `call` aufgerufen. Dem Methodenaufruf werden zwei Parameter übergeben. Zuerst wird wieder das Objekt, aus dem die Funktion ausgeführt werden soll, angegeben. Hier wird die Funktion aus dem erstellten Objekt `obj` aufgerufen. Als weitere Parameter können Werte angegeben werden, die `1 : 1` an die aufzurufende Funktion weitergeleitet werden. In dem Beispiel wird eine Referenz auf das Objekt `obj` übergeben, dadurch kann man innerhalb der Funktion überprüfen, ob `this`, `obj` entspricht. Dies trifft zu, da der Vergleich `true` ergibt.

Das Objekt arguments

Jede aufgerufene Funktion besitzt ein Objekt namens `arguments`. Dieses Objekt besitzt vier Informationen, die wir im Folgenden beschreiben.

- **arguments.callee**

Diese Eigenschaft des `arguments`-Objektes verweist auf die Funktion selbst, so können Sie mit `arguments.callee()` die Funktion aus sich selbst heraus wieder aufrufen, ohne den Namen der Funktion zu kennen.

Listing 1.108: Beispiel für die Benutzung von arguments.callee

```
function potenz(basis, exponent){
    if(exponent>1){
        return basis* arguments.callee(basis, exponent-1 );
    }
    return basis
}
trace(potenz(5,6));
```

Dieses Beispiel haben wir Ihnen schon auf Seite 96 vorgestellt. Hier besitzt es allerdings eine kleine Modifikation, so dass der Funktionsname durch `arguments.callee` ersetzt wurde.

▶ arguments.caller

Diese Eigenschaft ähnelt `callee`. Während `callee` auf die aktuelle Funktion verweist, verweist `caller` auf die Funktion, aus der die Funktion heraus aufgerufen wurde.

Listing 1.109: Beispiel für die Anwendung von arguments.caller

```
function FKT1(i) {
        trace("FKT1 wurde ausgeführt, i = "+i);
        arguments.caller(i+0.5);
}
function FKT2(i) {
    trace("FKT2 wurde ausgeführt, i = "+i);
    if(i<3){
        FKT1(i+1);
    }
}
FKT2(1);
```

Bildschirmausgabe:

```
FKT2 wurde ausgeführt, i = 1
FKT1 wurde ausgeführt, i = 2
FKT2 wurde ausgeführt, i = 2.5
FKT1 wurde ausgeführt, i = 3.5
FKT2 wurde ausgeführt, i = 4
```

Das Beispiel soll nur zeigen, wie die Funktion `FKT1` die Funktion `FK2` über die Eigenschaft `arguments.caller` aufruft.

▶ arguments[]

Indem Sie auf das Objekt `arguments` wie auf ein Array zugreifen, erhalten Sie die an die Funktion übergebenen Parameter.

Listing 1.110: Ansprechen von übergebenem Parameter ohne direkten Namensbezug

```
function FKT1() {
    trace(arguments[0] + " " + arguments[1] + " "
    + arguments[2]);
}
FKT1("wert1", "wert2", "wert3");
```

Bildschirmausgabe: `wert1 wert2 wert3`

▶ arguments.length

Wenn man auf das Objekt `arguments` schon wie auf ein Array zugreifen kann, um an die einzelnen Parameter der Funktion heranzukommen, besitzt dieses natürlich auch das Attribut `length`, wie man es schon von Arrays her kennt.

Durch diese zwei Dinge kann man Funktionen erstellen, die auf eine flexible Anzahl von Parametern reagieren können.

Listing 1.111: Eine Funktion, die alle übergebenen Parameter mit trace() einzeln ausgibt

```
function FKT1() {
    for(i=0;i<arguments.length;i++){
        trace(arguments[i]);
    }
}
FKT1("wert1", "wert2", "wert3", "wert4", "wert5");
```

Bildschirmausgabe:

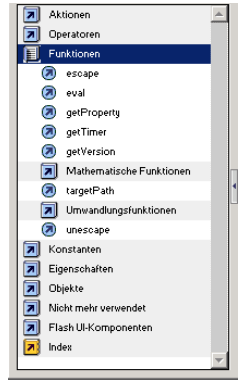
```
wert1
wert2
wert3
wert4
wert5
```

Standardfunktionen

Flash hat auch schon Grundfunktionen mit implementiert. Darunter fallen zum einen alle Umwandlungsfunktionen (Casting-Funktionen) wie `Boolean()`, `Number()` usw. so wie `getTimer()`.

In Abbildung 1.24 sehen Sie die Funktionen.

Abbildung 1.24: Auswahlmenü des ActionScript-Fensters



1.8 Objektorientiertes Programmieren (OOP)

Jeder spricht davon, überall wird es als das neue große Feature angezettelt, doch was verbirgt sich dahinter? Braucht man ab sofort keine Arrays oder Variablen mehr ?

Wohl kaum! Während wir in den vorangegangenen Kapiteln in den meisten Fällen einen strikt ablaufenden Programmtext geboten haben, wird sich dieses nun ändern.

OOP bietet vom Grundprinzip her nur eine Alternative zur Problemlösung an. Bei einer normalen Programmierung existiert theoretisch nur

eine Abarbeitungsrichtung des Programmtextes, zwar „springt“ man auf dieser hin und her, aber es wird immer strikt von oben nach unten eine Befehlsreihenfolge abgearbeitet. Dies kann bei kleineren Problemen zu einer sehr übersichtlichten Lösung führen, sobald Sie aber versuchen komplexere Anwendungen damit zu beschreiben, wird es sehr unübersichtlich.

Es bietet sich dann an, objektorientiert zu programmieren. Dabei wird die Problemstellung in kleinere Teilprobleme zerlegt, welche dann separat behandelt werden. Ein weiterer Vorteil ist, dass man ähnlichen Problemen, die Lösung des Teilproblems dort wieder einsetzen kann, da automatisch alles modular programmiert wird.

Dies hört sich kompliziert an, erleichtert aber die Arbeit um Einiges. Allerdings muss man dafür auch willig sein, es zu benutzen.

Objekte, Instanzen, Exemplare
Es handelt sich jeweils um dasselbe. Alle Ausdrücke sind gebräuchlich, wobei Objekt im Zusammenhang mit Klasse wohl am meisten gebraucht wird.

Eigenschaften
Von Eigenschaften spricht man, wenn man sich auf Variablen bezieht die zu einem Objekt gehören.

Attribute
Bei Attributen handelt es sich um Variablen, die in einer Klasse erstellt wurden.

Objekte

Ein Objekt, ist der Grundstein der objektorientierten Programmierung. Um zu beschreiben, wozu ein Objekt dient, gehen wir noch einmal auf das Beispiel aus dem Kapitel 1.5, Abschnitt „Mehrdimensionale Arrays“ ab Seite 78 ein.

Arr2 (Arrayname)

Index1 \ Index2	0	1	2
0	"Jan"	"Wuppertal"	567988
1	"Marc"	"Kamen"	543809
2	"Frank"	"Unna"	543478
3	"Fabian"	"Magdeburg"	538064

Tabelle 1.12: Zweidimensionales Array

Der kürzeste Quelltext dazu sieht wie folgt aus:

```
arr2=[["Jan","Wuppertal",567988],
["Marc","Kamen",543809],
["Frank","Unna",543478],
["Fabian","Magdeburg",538064]];
```

Wie man sieht, wurden Vorname, Ort und Telefonnummer von verschiedenen Personen in einem zweidimensionalen Array abgespeichert.

Sicherlich nicht die allerschlechteste Variante, um Userdaten abzuspeichern. Aber es fällt doch sehr störend auf, dass die einzelnen Spalten durchnummeriert sind und keine Namen besitzen.

Deshalb erzeugt man für jede Person ein Objekt. Dementsprechend wäre das erste Objekt `Jan`, das zweite `Marc` usw.

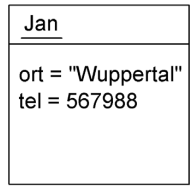
Listing 1.112: Erstellen eines Objektes

```
jan = new Object( );
jan.ort="Wuppertal";
jan.tel=567988;

marc= new Object( );
marc.ort = "Kamen";
marc.tel = 543809;

usw.
```

Abbildung 1.25: Objekt: Jan



Hier wurde wieder die Konstruktorfunktion zum Erstellen eines Objektes benutzt. Eine alternative und kürzere Schreibweise wäre die folgende:

Listing 1.113: Erstellung eines Objektes in kurzer Schreibform

```
jan = { ort:"Wuppertal", tel: 567988 };
marc = { ort:"Kamen", tel: 543809};
```

Zugriff erlangt man auf den Inhalt der Objekte durch den Punktoperator:

```
trace( jan.ort);
```

Bildschirmausgabe: `Wuppertal`

Die Objekte nach den Namen zu benennen, ist ungeschickt, da man jeweils den Namen der Person wissen muss, um auf die weiteren Daten zugreifen zu müssen. Deshalb wählt man als Namen immer einen Oberbegriff. Da wir hier Personen verwalten, also „Personen“. Allerdings kann man nicht alle Objekte Person oder Personen nennen, da diese ja unterschiedliche Namen benötigen.

Man würde also wieder anfangen, durchzunummerieren: `Person1`, `Person2`.

Zu Beginn des Kapitels zu den Arrays haben wir anhand eines ähnlichen Beispiels erklärt, warum man Arrays benutzt. Dementsprechend wollen wir dies auch hier tun, es ist nämlich ohne Probleme möglich, Objekte in Arrays zu speichern.

```
Personen= [ { ort:"Wuppertal", tel: 567988 }, { ort:"Kamen",
tel: 543809}];
```

Wir haben jetzt zum Erstellen des Arrays und des Objektes jeweils die kurze Notation gewählt. Lassen Sie sich dadurch nicht abschrecken, das Beispiel genauer anzuschauen.

Das erste Arrayelement enthält die Daten von Jan, das zweite die Daten von Marc. Etwas unpraktisch ist, dass die Namen verschwunden sind. Deshalb werden wir die Namen auch als Eigenschaften des Objektes mit abspeichern.

```
Personen= [ { name:"Jan", ort:"Wuppertal", tel: 567988 }, {
name:"Marc", ort:"Kamen", tel: 543809}];
```

Die Reihenfolge ist irrelevant bei der Angabe der Objekteigenschaften.

Auf die einzelnen Inhalte kann man nun einfach über das Array zugreifen.

Personen[0].name wäre zum Beispiel "`Jan`", da man auf das erste Feld des Arrays und dadurch auf das erste Objekt zugreift und von diesem die Eigenschaft `name` anspricht.

Wie man sieht, kann man ein Objekt als eine Art Container ansehen, in dem man alles abspeichern kann.

Klassen

Folgen wir direkt dem zuvor beschriebenen Beispiel:

```
Personen= [ { name:"Jan", ort:"Wuppertal", tel: 567988 }, {
name:"Marc", ort:"Kamen", tel: 543809}];
```

Dies erstellte also ein Array namens `Personen`, welches Objekte mit weiteren Daten enthält. Sehr nachteilig ist allerdings, dass man sich durch das immer wieder neue Eingeben des Eigenschaftennamens schnell vertippt.

Um alles in „standardisierte“ Formen zu bringen, gibt es Klassen. Klassen sind eine Beschreibung für ein Objekt. Sie definieren vorab z.B., dass jede Person einen Namen, einen Ort, eine Telefonnummer und eine E-Mail-Adresse besitzt.

Auch wenn Ihnen dies nicht bewusst war, haben wir im vorangegangenen Beispiel schon eine Klasse benutzt, nämlich die Klasse `Object`. Von dieser Klasse haben wir leere Objekte erhalten, die wir anschließend mit Eigenschaften gefüllt haben. Wenn man die kurze Schreibweise wählt, geschieht genau das gleiche, auch wenn nicht das Schlüsselwort `Object` auftaucht.

Sie haben also eben schon Exemplare von der Klasse `Object` gebildet. Da die Klasse `Object` aber nur Attribute und Methoden besitzt, die wir nicht benutzt haben, ist dies nicht weiter aufgefallen. Eine vordefinierte Klasse bringt in diesem Fall nicht so viele Vorteile, deshalb erstellt man

sich seine auf den jeweiligen Fall angepasste Klasse. Von dieser kann man dann die benötigte Anzahl an Exemplare bilden kann.

Auch wenn der folgende Code aussieht, als würde eine normale Funktion erstellt: Ignorieren Sie es! Da es in ActionScript kein spezielles Schlüsselwort wie `class` gibt, erstellt man eine Funktion, die dann einer Klasse entspricht. Gängig ist auch die Formulierung „Konstruktorfunktion“, da die Funktion beim Erstellen eines Exemplars aufgerufen wird. Die Funktion bildet jedoch selbst schon die Klasse:

Listing 1.114: Eine Klasse wird erstellt.

```
Klassenname = function()
{
  this.var1 = "Variable 1";
  this.var2 = "Variable 2";
};
```

Klassenname

Für Klassennamen gibt es noch einige unausgesprochene Regeln, an die man sich halten sollte.

Der Name einer Klasse sollte immer mit einem Großbuchstaben beginnen.

Außerdem sollte der Name einer Klasse im Singular stehen, also Einzahl sein, da eine Klasse immer für den Inhalt eines Objektes steht. Wenn Sie also eine Klasse mit dem Namen Bücher erstellen, würde diese Klasse theoretisch mehrere Bücher verwalten und nicht nur eins. Sie könnten von dieser Klasse noch mehrere Objekte bilden, so würden Sie nicht nur einmal Bücher verwalten, sondern noch öfter. Sie hätten quasi eine Klasse, die zum Verwalten von Bücherverwaltungsobjekten dient. Zumindest liegt die Vermutung nahe, die Namensgebung hat natürlich keinen praktischen Einfluss auf die Funktionalität der Klasse.

Die selbst erstellte Klasse für unser Beispiel mit den Personen könnte so aussehen:

```
Person = function(var1, var2, var3)
{
  this.name = var1;
  this.ort = var2;
  this.tel = var3;
};
```

Man erkennt, dass die Klasse direkt beim Aufruf Parameter entgegennimmt. Dies hat den Vorteil, dass man direkt beim Erstellen den Inhalt des späteren Objektes mit angeben kann und so das Objekt direkt mit den richtigen Inhalt erstellt.

```
person1 = new Person("Jan", "Wuppertal", 567988  );
trace(person1.name);
```

Bildschirmausgabe: `Jan`

Natürlich kann man auch die Objekte der selbst erstellten Klasse direkt in einem Array erzeugen.

```
personen= [ new Person("Jan", "Wuppertal", 567988  ),
            new Person("Marc" , "Kamen", 543809)];
trace(personen[1].name);
```

Bildschirmausgabe: `Marc`

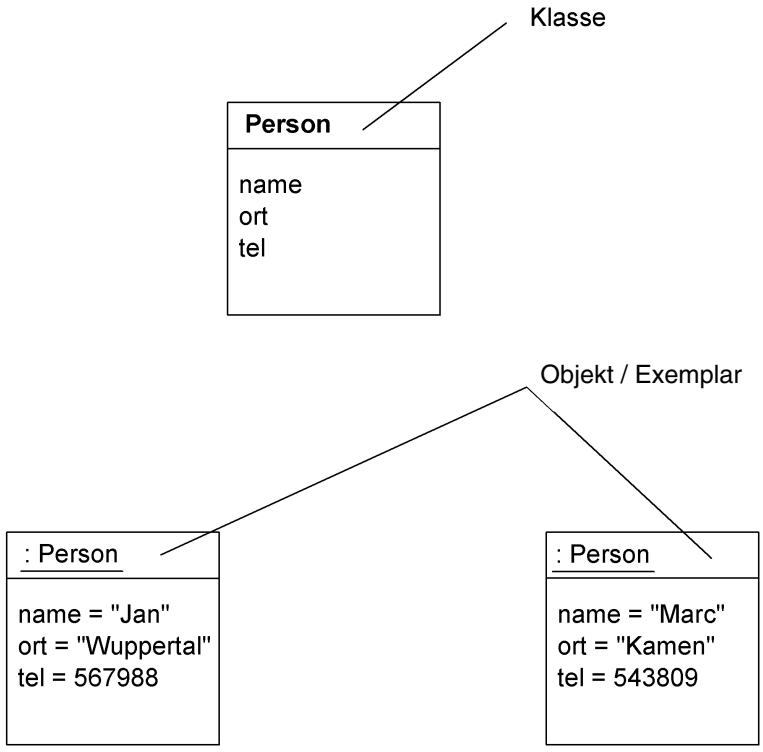


Abbildung 1.26: Die Klasse Person und zwei Exemplare

Objekt- und Klassendiagramme

Die von uns benutzte Bebilderung der Klassen und Objekte erfolgt nicht willkürlich, sondern richtet sich nach dem UML-Standard (Unified Modeling Language). Falls Sie die genaue Definition der UML interessiert, finden Sie weitere Informationen im Internet auf den Seiten:

http://www.uml.org/
http://www.omg.org/
http://www.odmg.org/
http://www.rational.com/uml/

Listing 1.115: Änderungen von Attributen nach Erstellung der Konstruktorfunktion sind nicht möglich.

```
Klassenname = function( )
{
    this.var1="Variable";
};
objekt1 = new Klassenname;
trace(objekt1.var1);
Klassenname.var1="geändertes Attribut"; //GEHT NICHT !!
objekt2 = new Klassenname;
trace(objekt1.var1);
trace(objekt2.var1);
objekt1.var1="In dem Exemplar geänderte Variable";
trace(objekt1.var1);
trace(objekt2.var1);
```

Bildschirmausgabe:

```
Variable
Variable
Variable
In dem Exemplar geänderte Variable
Variable
```

Anhand des Beispieles erkennt man, dass Attribute einer Klasse, die in der Konstruktorfunktion gesetzt werden, nicht durch `Klassenname.var1` geändert werden können. (Dies erscheint logisch, wenn man noch einmal an sein Wissen über Funktionen denkt oder es in Kapitel 1.7 ab Seite 89 nachliest.) Hingegen kann man die Eigenschaften in Exemplaren (`objekt1`) ganz normal verändern.

Es gibt drei grundlegende Möglichkeiten, Variablen einer Klasse oder einem Objekt mit auf den Weg zu geben.

1. In der Konstruktorfunktion
   Diese Möglichkeit haben Sie oben schon kennen gelernt.

   ```
   Klassenname= function() {
       this.var1="Inhalt";
   };
   ```

   Wenn Sie von dieser Klasse ein Objekt bilden, enthält jedes Objekt eine eigene Variable var1 mit dem Inhalt „Inhalt“.

2. Über den Klassennamen (static)
   Diese Version haben wir auch eben schon angewendet, um eine Variable aus der Konstruktorfunktion zu ändern – was natürlich nicht ging. Auf diese Weise kann man aber statische Variablen erzeugen, die nur in der Klasse bekannt sind. Diese Variablen werden nicht an das Objekt übergeben!

   ```
   Klassenname.var2 = "Statische Variable";
   ```

3 Über das prototype-Exemplar
Auf diese Art sind wir noch nicht eingegangen, da sie die komplexeste ist. Der Quellcode würde wie folgt aussehen.

```
Klassenname.prototype.var3 = "Vererbte Variable";
```

Hier noch einmal für die Beispiele die entsprechende Ausgabe:

```
ukObjekt = new Klassenname();
trace("----------------------");
trace(ukObjekt.var1);//existiert
trace(ukObjekt.var2);//NA, da statisch
trace(ukObjekt.var3);//existiert
trace("----------------------");
trace(Unterklasse.var1);//NA
trace(Unterklasse.var2);//existiert
trace(Unterklasse.var3);//NA
```

prototype

Jede erstellte Klasse erhält automatisch von Flash ein Exemplar `prototype`.

Jedes Mal, wenn man durch das Schlüsselwort `new` ein neues Objekt von einer Konstruktorfunktion erzeugt, erzeugt Flash für jede Eigenschaft des prototype-Exemplars eine Referenz in dem neuen Objekt auf die jeweilige Eigenschaft des prototype-Exemplars. Es handelt sich hierbei um eine Vererbung. Dadurch ist man in der Lage, auch noch die „Eigenschaft" der Klasse im späteren Verlauf des Programms zu ändern. Oben hatten wir definiert, dass alle Variablen einer Klasse „Attribute" heißen und alle Variablen eines Objektes „Eigenschaften". Dies liegt daran, dass man Werte der Klasse nicht ändert, sondern die Werte des Exemplars mit dem Namen `prototype` – aber es hat fast die gleiche Auswirkung. Warum nur „fast", soll das folgende Beispiel erläutern:

Listing 1.116: Zuweisung eines Klassenattributs mit prototype

```
Klassenname = function( )
{
    //Leere Klasse
};
Klassenname.prototype.var1="Klassenattribut";

objekt1 = new Klassenname;
objekt2 = new Klassenname;
trace(objekt1.var1);
trace(objekt2.var1);
Klassenname.prototype.var1="geändertes Klassenattribut";
trace(objekt1.var1);
trace(objekt2.var1);
objekt1.var1="geänderte Objekteigenschaft";
trace(objekt1.var1);
trace(objekt2.var1);
```

Bildschirmausgabe:

```
Klassenattribut
Klassenattribut
geändertes Klassenattribut
geändertes Klassenattribut
geänderte Objekteigenschaft
geändertes Klassenattribut
```

Mit der Zeile Klassenname.prototype.var1="Klassenattribut"; fügen wir dem Objekt `prototype` noch eine Eigenschaft hinzu: die Variable `var1`. Danach erzeugen wir zwei Objekte der Klasse. Jedes Objekt enthält als Eigenschaft eine Referenz mit dem Namen `var1`, die auf `prototype.var1` verweist. Wie das Beispiel zeigt, kann man so später `Klassenname.prototype.var1` verändern, und es ändern sich auch die Werte in den Objekten.

Referenzvariable
Eine Referenz ist eine Variable, die direkt den Inhalt einer anderen Variablen anzeigt. Diese Referenzvariable besitzt als Inhalt nur die Adresse oder den Namen der anderen Variablen, deren Inhalt referenziert wird.

Leider sind Referenzen unter Flash noch problematisch in der Handhabung, da es weder einen Zeiger noch einen Adressoperator gibt. Dadurch ist es nicht möglich, selbst Referenzen zu erstellen, und man muss diejenigen benutzen, die Flash bietet.

Wenn man einen Referenzverweis einer Variablen löscht, wie im Beispiel geschehen. objekt1.var1="geänderte Objekteigenschaft"; ist es nicht möglich, diese wieder zu erstellen. Zudem zeigt es, dass man über die Referenz in Flash keinen Einfluss auf die referenzierte Variable nehmen kann.

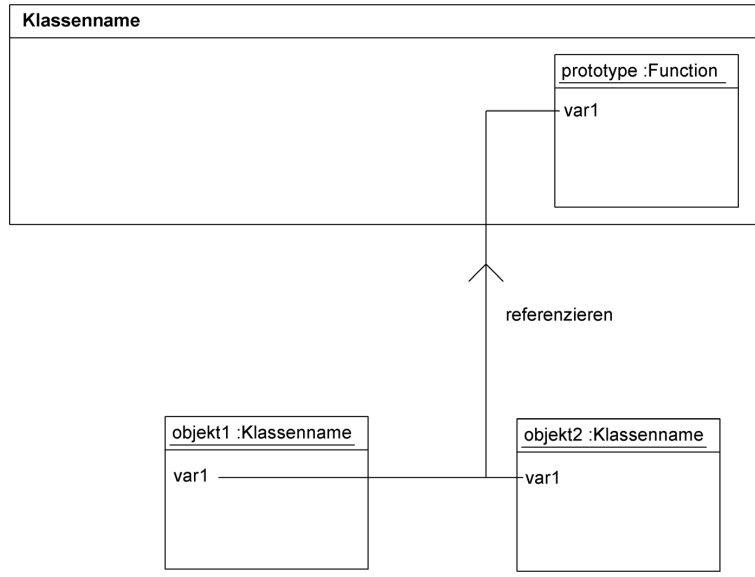


Abbildung 1.27: *Die Objekte der Klasse Klassennamen haben als Eigenschaft eine Referenz auf Klassenname.prototype.var1*

Hinweis zur Grafik: Klassennamen sind fett gedruckt. Objekte beinhalten „Objektnamen :Klassennamen“ und sind unterstrichen.

Der Klassenname, von dem das Exemplar `prototype` gebildet wurde, ist unbekannt, da dies Flash-intern geschieht. Daher die drei Fragezeichen beim Klassennamen von `prototype`. An der Grafik soll man erkennen, dass die Eigenschaft `var1` der erstellten Objekte von `Klassenname` Referenzen auf `Klassenname.prototype.var1` sind. Demzufolge kann man `var1` auch zu dem Zeitpunkt als Klassenattribut bezeichnen. Es existiert nur einmal in der Klasse selbst und wird nicht an die erstellten Objekte mit weitergegeben. Es wird nur die Referenz weitergegeben, wenn man ein Klassenattribut mit einer Zahl belegt und versucht, auf die Referenz in einem Objekt den Operator ++ anzuwenden. Dann wandelt Flash die Referenz in eine Numbervariable um und weit dieser den um eins erhöhten Wert der zuvor referenzierten Zahl zu.

Man kann in Flash also nicht über eine Referenz den referenzierten Wert verändern!

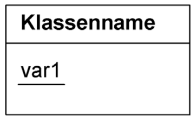
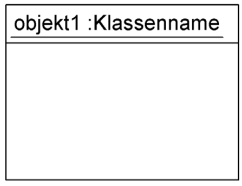
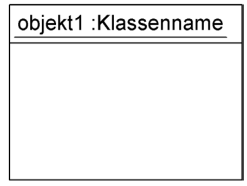


Abbildung 1.28: Klassendiagramm mit einer statischen Variable





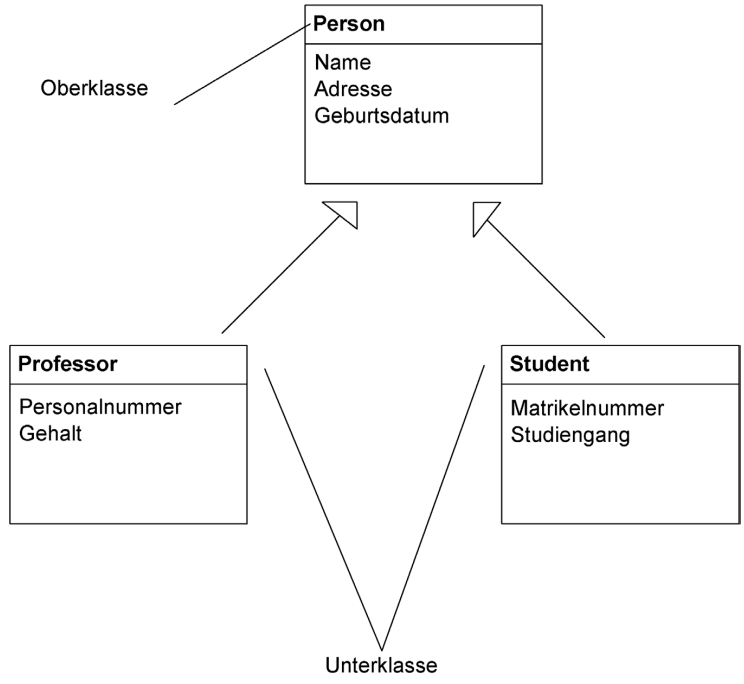
Zum Vergleich hier noch einmal der Unterschied zu einer Klassenvariable: Normalerweise würde es sich, wenn man nur die Grafik betrachtet, um eine statische Variable (Klassenvariable) mit dem Namen `var1` handeln. Statische Variablen sind durch den Unterstrich gekennzeichnet. Dazu würde folgendes ActionScript gehören:

```
Klassenname.var1="Inhalt";
```

Vererbung

Bei der Vererbung geht es darum, dass eine Klasse (Unterklasse) die Attribute einer anderen Klasse (Oberklasse) übernimmt/erhält. Das folgende Klassendiagramm soll dies verdeutlichen.

Abbildung 1.29: Vererbungsschema



Beide Klassen, sowohl `Student` als auch `Professor`, würden nach der Vererbung die Attribute `Name`, `Adresse` und `Geburtsdatum` der Klasse `Person` besitzen. Bei der Vererbung handelt es sich also jeweils um eine Spezialisierung. Hier im Beispiel von Person zu Student oder Professor. Andere Beispiele sind:

Auto:
- LKW
- PKW

Eier
- Spiegelei
- Osterei
- Rührei

Tier
- Katze
- Hund

usw.

Diese Art von Beispielen (Spezialisierung) ist nicht zu verwechseln mit den Strukturen.

Strukturen sind:

Auto:
- Rad
  - Felgen
    - Radschrauben
- Tür
  - Türgriff

Haus:
- Wand
  - Steine
  - Fenster
    - Griff
    - Glas

Strukturen kann man durch Assoziationen darstellen.

Kommen wir jedoch zurück zur Vererbung: In speziellen Programmiersprachen muss die Vererbung auch eingesetzt werden, damit ein Objekt Typen von der Oberklasse referenzieren kann. Dies ist aber in Flash nicht nötig, da man den Typ der Variablen nicht festlegen muss bzw. auch nicht zwingend festlegen kann. In Flash setzt man die Vererbung also nur dafür ein, Klassen zu spezialisieren.

Sobald man eine Klasse vererben möchte, kommt man um `prototype` nicht herum. Wie im Abschnitt „prototype“ ab Seite 109 beschrieben, erzeugt Flash beim Erstellen eines neuen Objektes für alle Methoden des `prototype`-Exemplars Referenzen in dem neuem Objekt.

Genau diese Eigenschaft nennt man Vererben. Wobei alle vererbten Objekte unter Flash zuerst nur als Referenz in der Unterklasse vorhanden sind. Erst wenn die Werte der Variablen in der Unterklasse geändert werden, ersetzt Flash die Referenz durch eine „richtige“ Variable, die objektbezogen ist. Es ist nicht möglich, direkt Variablen als objektbezogen zu definieren.

Oberklasse, Mutterklasse, Vaterklasse, Elternklasse

Alle diese Begriffe umschreiben immer die Klasse, von der die andere Klasse (Unterklasse) erbt. Wir benutzen im Folgenden das Wort Oberklasse.

Unterklasse, Tochterklasse, Sohnklasse, Kindklasse

Alle Begriffe umschreiben immer die Klasse, die von einer andere Klasse (Oberklasse) erbt. Wir benutzen im Folgenden das Wort Unterklasse.

Das in Abbildung 1.29 beschriebene Beispiel sieht unter Flash wie folgt aus:

Listing 1.117: Vererbungsstruktur in ActionScript

```
Person = function( )
{
    this.name="John Dooh";
    this.adresse="";
    this.geb=19000101;
};
Professor = function( )
{
    this.personalnummer=0001;
    this.Gehalt=0;
};
Professor.prototype = new Person();
Student = function( )
{
    this.matrikelnummer=0001;
    this.studiengang=null;
};
Student.prototype = new Person();
stud1= new Student;
trace(stud1.name);
```

Bildschirmausgabe: `John Dooh`

Durch die Zeilen Student.prototype = new Person(); oder Professor.prototype = new Person(); wurde das automatisch erstellte `prototype`-Exemplar gelöscht und durch ein Exemplar der Klasse `Person` ersetzt.

Die Vererbung der Eigenschaften des Exemplars, das `prototype` ersetzt hat, funktioniert genauso wie zuvor bei `prototype` selbst. In dem Beispiel haben wir anschließend noch ein Objekt von `Student` gebildet und die vererbte Eigenschaft `name` ausgegeben.

Methoden

Das Beispiel aus Kapitel 1.8, Abschnitt „Vererbung“ ab Seite 111 zeigt zwar die Vererbung, einen wirklichen Zweck kann man aber nur erahnen.

Dementsprechend wollen wir das Beispiel erweitern.

Jede Klasse kann Methoden besitzen. Methoden sind ähnlich wie Funktionen aufgebaut und erfüllen auch denselben Zweck. Der Unterschied besteht darin, dass eine Methode immer einer Klasse zugeordnet ist, was bei Funktionen nicht der Fall ist.

Es gibt zwei Arten, Methoden in einer Klasse zu erstellen: Entweder Sie erstellen diese direkt im Konstruktor.

```
Person = function( )
{
    this.name="John Dooh";
    this.straße="nirgendwo";
    this.ort="irgendwo";
    this.plz="???";
    this.geb=19000101;
    this.getAdr = function(){
        return (this.name+newline+this.straße+newline+this.plz
            +" "+this.ort);
        };
};
```

was zur Folge hat, dass jedes von dieser Klasse erstellte Exemplar eine eigene Methode `getAdr()` hat,

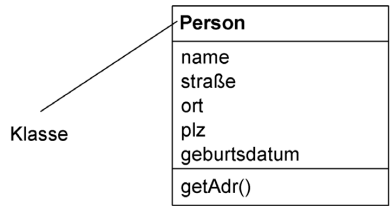
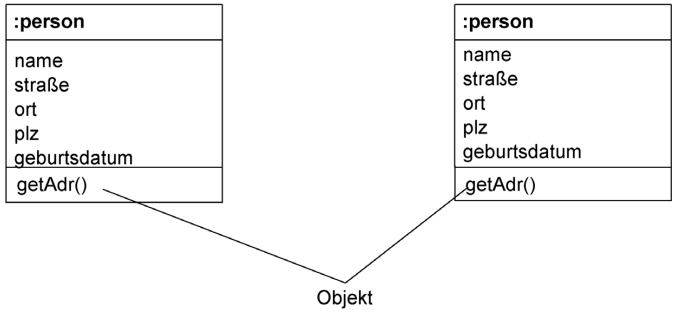


Abbildung 1.30: Jedes Exemplar hat eine eigene Methode getAdr().

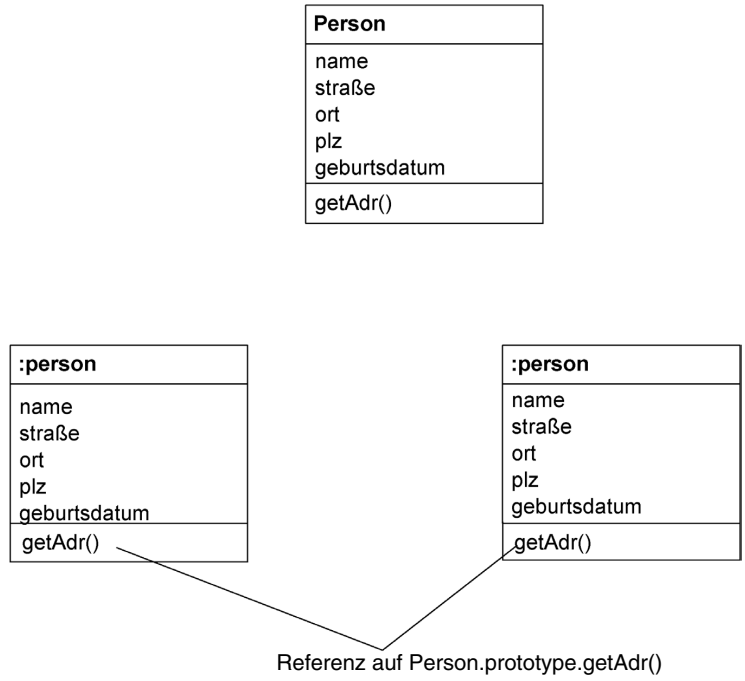


oder Sie „vererben“ es, indem Sie mit `prototype` arbeiten:

```
Person = function () {
    this.name = "John Dooh";
    this.straße = "nirgendwo";
    this.ort = "irgendwo";
    this.plz = "???";
    this.geb = 19000101;
};
person.prototype.getAdr = function() {
    return (this.name + newline + this.straße + newline
        + this.plz + " " + this.ort);
};
```

Letzteres hat den Vorteil, dass die Methode nur einmal in der Klasse (in `prototype`) erstellt wird. Wenn man dann ein Exemplar dieser Klasse bildet, bekommt jedes Objekt eine Referenz mit dem Methodennamen zugewiesen.

Abbildung 1.31: Referenzierte Methode



super

`super` ist ein neues Schlüsselwort und dient dazu, aus der Unterklasse Objekte oder Methoden der Oberklasse aufzurufen.

```
Person = function () {
    this.name = "John Dooh";
    this.straße = "nirgendwo";
    this.ort = "irgendwo";
    this.plz = "???";
    this.geb = 19000101;
};
person.prototype.getAdr = function() {
    return (this.name + newline + this.straße + newline
        + this.plz + " " + this.ort);
};
```

Bei der zuletzt erstellten Klasse `Person` fällt auf, dass den Attributen `name`, `straße` usw. einfach ein Wert zugewiesen wird. Dies braucht man meist in der Praxis nicht, sondern man möchte, wie wir es auch schon im Ab-

schnitt „Klassen“ ab Seite 105 vorgestellt haben, die Werte mit in der Konstruktorfunktion übergeben.

Listing 1.118: Klasse Person mit einer Methode

```
Person = function (name, strasse, plz, ort, geb) {
    this.name = name;
    this.strasse = strasse;
    this.plz = plz;
    this.ort = ort;
    this.geb = geb;
};
person.prototype.getAdr = function() {
    return (this.name + newline + this.straße + newline
        + this.plz + " " + this.ort);
};
```

Gehen wir nun wieder einen Schritt weiter und vererben die Klasse `Person` an die Klasse `Student`:

Listing 1.119: Student erbt die Klasse Person

```
Person = function (name, strasse, plz, ort, geb) {
    this.name = name;
    this.strasse = strasse;
    this.plz = plz;
    this.ort = ort;
    this.geb = geb;
};
person.prototype.getAdr = function() {
    return (this.name + newline + this.strasse + newline
        + this.plz + " " + this.ort);
};

Student = function (name, strasse, plz, ort, geb, matrnr,
    studgang) {
    //super(name, strasse, plz, ort, geb);
    this.matrikelnummer=matrnr;
    this.studiengang=studgang;
};
Student.prototype = new Person();
stud1 = new Student("Peter Silie","Rosenberg 5", 55555,
    "Nirgendwo", 19900101, 4545345, "AI");
trace(stud1.getAdr());
```

Wenn man dies so erstellt, bleibt das Ausgabefenster leer, da die Daten, die in der geerbten Klasse `Person` gespeichert werden sollen, dort nicht „ankommen“. Man muss dafür sorgen, dass die Daten aus der Konstruktorfunktion der Klasse `Student` an die Konstruktorfunktion der Klasse `Person` weitergeleitet werden.

Normalerweise geschieht dies beim Aufruf der Konstruktorfunktion beim Erstellen eines neuen Objektes. Dies wäre die Zeile Student.prototype = new Person(); . Zu dem Zeitpunkt wird zwar auch ein neues Exemplar von `Person` gebildet, aber nur, um das `prototype` Exemplar von `Student` zu überschreiben und so die Vererbung erst zu ermöglichen.

Mit den bisher bekannten Befehlen ist schlecht an die Konstruktorfunktion von `Person` heranzukommen. Deshalb gibt es `super`, das auf die Klasse, von der geerbt wurde, also auf die Oberklasse, verweist. Wir haben das vorangegangen Beispiel leicht abgeändert, um die Funktionsweise deutlich zu machen: `super` verweist nicht direkt auf `Person`, sondern auf `Person.prototype`.

Listing 1.120: Beispiel für die Benutzung von super

```
Person = function (name, strasse, plz, ort, geb) {
    this.name = name;
    this.strasse = strasse;
    this.plz = plz;
    this.ort = ort;
    this.geb = geb;
    this.test="Hallo Peter 1";
};
person.prototype.test2="Hallo Peter 2";
person.prototype.getAdr = function() {
    return (this.name + newline + this.strasse + newline
        + this.plz + " " + this.ort);
};

Student = function (name, strasse, plz, ort, geb, matrnr,
    studgang) {
    super(name, strasse, plz, ort, geb);
    trace(super.test);
    trace(super.test2);
    this.matrikelnummer=matrnr;
    this.studiengang=studgang;
};
Student.prototype = new Person();
stud1 = new Student("Peter Silie","Rosenberg 5", 55555,
    "Nirgendwo", 19900101, 4545345, "AI");
trace(stud1.getAdr());
```

Bildschirmausgabe:

```
undefined
Hallo Peter 2
Peter Silie
Rosenberg 5
55555 Nirgendwo
```

Gehen wir zuerst auf die Klasse `Person` ein. Diese besitzt noch zwei zusätzliche Variablen: zum einen die Variable `test`, welche im Konstruktor gesetzt wird, und zum anderen die Variable `test2`, welche sich im `prototype` Exemplar befindet.

Wenn anschließend ein Exemplar von der Klasse `Student` gebildet wird, welche die Klasse `Person` geerbt hat, wird mit `super()` der Konstruktor von `Person` aufgerufen. Im Beispiel übergeben wird dort die entsprechenden Variablen, so dass später die Methode `getAdr()` eine richtige Ausgabe liefert.

Mit super kann man nur auf die Variablen und Methoden in dem prototype-Exemplar von Person zugreifen, nicht aber auf die Variablen, die in der eigentlichen Klasse bzw. in der Konstruktorfunktion stehen, da diese sogar gleichnamige Unterklassenvariablen überschreiben. Dies erkennen Sie daran, dass die Ausgabe von super.test ein undefined liefert und die Ausgabe von super.test2 den Wert von Person.prototype.

Man kann mit this. und dem Namen der Variablen oder der Methode direkt auf die Oberklassenattribute und Methoden zugreifen. Der Einsatz von super macht nur dann Sinn, wenn die Namensräume sich überlagern und so z.B. this.test auf eine existierende Variable in Student zeigt.

Wie im folgenden Ausschnitt der Student-Klasse:

Listing 1.121: Anwendung von super bei Namensüberlagerung

```
Student = function (name, strasse, plz, ort, geb, matrnr,
    studgang) {
    super(name, strasse, plz, ort, geb);
    this.test2="Test";
    trace(this.test2);
    trace(super.test2);
    this.matrikelnummer=matrnr;
    this.studiengang=studgang;
};
```

Bildschirmausgabe:

```
Test
Hallo Peter 2
```

this.test2 gibt die davor noch erstellte Variable aus, während man mit super.test2 auf die Variable in Person.prototype zugreift (bzw. auf eines der gebildeten Exemplare, das prototype aus Student überschrieben hat).

Variablen, die in einem Konstruktor einer Oberklasse erstellt werden, überschreiben die gleichnamigen Variablen der Unterklasse !

```
Student = function (name, strasse, plz, ort, geb, matrnr,
    studgang) {
    this.test="Test";
    super(name, strasse, plz, ort, geb);
    trace(this.test);
    this.matrikelnummer=matrnr;
    this.studiengang=studgang;
};
```

Die Ausgabe von this.test ergibt „Hallo Peter“, die zuvor erstellte Variable this.test mit dem Inhalt „Test“ wird also überschrieben. In jeder gängigen Programmiersprache würde die Variable test allerdings nicht überschrieben!

Wenn man einer Unterklasse Methoden über das prototype-Exemplar zuweisen möchte, wird prototype durch das Vererben der Klasse Person überschrieben. Deshalb kann man selbst erstellte Methoden erst nach dem Überschreiben erstellen.

```
Student.prototype.getMatr = function() {
    return this.matrikelnummer+" "+super.getName() ;
};
Student.prototype = new Person();
```

Dies wäre genau die falsche Reihenfolge.

Auch als Abschluss des Kapitels zur Vererbung noch einmal der zuletzt erstellte komplette Quelltext, bei dem noch eine zusätzliche Methode für die Klasse Student erstellt wurde:

Listing 1.122: Beispiel für eine Vererbungsstruktur

```
Person = function (name, strasse, plz, ort, geb) {
    this.name = name;
    this.strasse = strasse;
    this.plz = plz;
    this.ort = ort;
    this.geb = geb;
};
person.prototype.getAdr = function() {
    return (this.name + newline + this.strasse + newline
        + this.plz + " " + this.ort);
};
person.prototype.getName = function() {
    return this.name;
};
Student = function (name, strasse, plz, ort, geb, matrnr,
    studgang) {
    super(name, strasse, plz, ort, geb);
    this.matrikelnummer = matrnr;
    this.studiengang = studgang;
};
Student.prototype = new Person();
Student.prototype.getMatr = function() {
    return this.matrikelnummer + " " + super.getName();
};
stud1 = new Student("Peter Silie", "Rosenberg 5", 55555,
    "Nirgendwo", 19900101, 4545345, "AI");
trace(stud1.getMatr());
```

Bildschirmausgabe: 4545345 Peter Silie

__proto__

__proto__ ist eine Referenzvariable, die jedes erstellte Objekt und jede erstellte Klasse automatisch enthält. __proto__ verweist bei Objekten auf das prototype-Exemplar der Klasse, von der das Objekt gebildet wurde.

```
Klassenname = function () {
    // Leere Klasse
};
Objekt1 = new Klassenname();
trace(Objekt1.__proto__ == Klassenname.prototype);
```

Listing 1.123: __proto__ verweist auf das prototype-Exemplar aus der Klasse, aus der das Exemplar gebildet wurde.

Bildschirmausgabe: true

Dieser Vergleich ist unnötig, da es in Flash MX jetzt auch die Funktion instanceof() gibt.

```
trace( objekt1 instanceof Klassenname );
```

Dies ergibt auch true, allerdings leistet die Funktion instanceof noch mehr als unser oberer Vergleich. Wenn man mit object vergleicht, erhält man auch ein true.

```
trace( objekt1 instanceof object );
```

Dies liegt daran, dass die Funktion dem __proto__-„Baum“ bis zum Ende folgt.

Alle Objekte und Klassen, die man erzeugt, erben automatisch die Attribute und Methoden von Object, da dies die höchste Klasse ist.

Der Code von instanceof sieht wie folgt aus:

```
function instanceof (theObject, theClass){
     while ((theObject = theObject.__proto__) != null) {
           if (theObject == theClass.prototype) {
                 return true;
           }
     }
     return false;
}
```

Listing 1.124: ActionScript der Funktion instanceof

An diesem ActionScript erkennt man, dass das Attribut von Object __proto__ null entsprechen muss, da darauf die while-Schleife aufgebaut ist.

__proto__ wird nur für den Interpreter erstellt, damit dieser, ähnlich wie oben die while-Schleife, den Pfad eines Objektes entlang laufen kann, um zu sehen, welche Methoden jenes Objekt erbt.

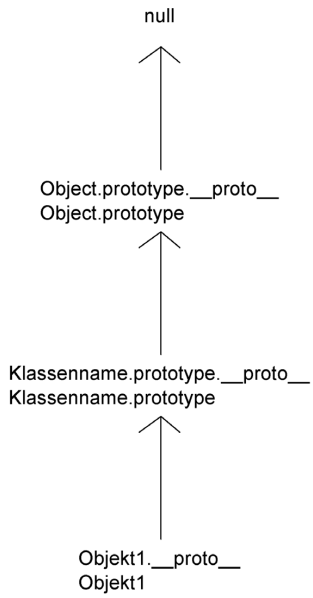


Abbildung 1.32: __proto__ -Baum

Die `__proto__`-Eigenschaft ist nicht schreibgeschützt (readonly). Dementsprechend kann man erstellte Objekte einfach einer anderen Klasse zuweisen. Dies geht natürlich in keiner ernst zu nehmenden Programmiersprache. Da Flash aber gerade in Bezug auf die Variablentypen sehr flexibel ausgelegt ist, bereitet dies überhaupt keine Probleme.

Wir erweitern das Beispiel mit den Klassen `Student` und `Person` um eine Klasse `Professor`.

```
Professor = function (name, strasse, plz, ort, geb, pnr,
    gehalt) {
    super(name, strasse, plz, ort, geb);
    this.personalnummer = pnr;
    this.Gehalt = gehalt;
};
Professor.prototype = new Person();
Professor.prototype.getPnr = function() {
    return "Pnr: "+this.personalnummer + " Name: "
        + super.getName();
};
```

Nach den Klassendefinitionen folgt das eigentlich Interessante:

```
stud1 = new Student("Peter Silie", "Rosenberg 5", 55555,
    "Nirgendwo", 19900101, 4545345, "AI");
trace(stud1.getMatr());
```

Es wird ein Objekt vom Typ `Student` erzeugt, dessen Methode `getMatr()` aufgerufen und das Ergebnis ausgegeben:

```
Mtr: 4545345 Name: Peter Silie
```

Als Nächstes ändern wir von dem Objekt den Verweis von __proto__, das auf Student verwies, auf ein neues Objekt der Klasse Professor.

```
stud1.__proto__ = new Professor("","","","","",677678,6000);
trace(stud1.getMatr()); //geht nicht mehr
trace(stud1.getPnr());
trace(stud1.getAdr());
```

Bildschirmausgabe:

```
undefined
Pnr: 677678 Name: Peter Silie
Peter Silie
Rosenberg 5
55555 Nirgendwo
```

Der __proto__-Verweis des Objekts stud1 wird geändert und zeigt nun auf ein Exemplar vom Typ Professor. Man macht also aus einem Studenten einen Professor.

Die ersten Parameter sind überflüssig: Dies sind die Personendaten, die von der Klasse Professor an den Konstruktor der Klasse Person weiter geleitet werden.

Diese Daten sind deshalb irrelevant, da das Objekt stud1 sie schon besitzt. Die neuen Daten personalnummer und gehalt werden in dem neu erstellten Exemplar mit abgespeichert.

Da Flash aber immer den __proto__-Baum entlangläuft, sieht es so aus, wie wenn unsere Daten direkt in stud1 stünden.

Hier merkt man schon, dass diese Lösungsvariante eigentlich nicht so gedacht ist. Es funktioniert auch nur, weil man ein zusätzliches Exemplar erstellt, welches sich in den __proto__-Baum „einschmiegt“ und von dem stud1 dadurch auch alle Eigenschaften und Methoden erbt.

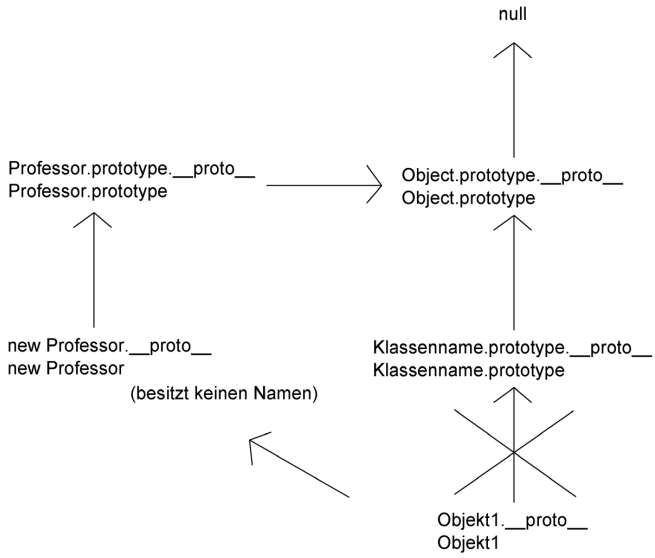


Abbildung 1.33: Manuelle Änderung des __proto__-Verweises

Dieses zweifelhafte Feature sollte nicht benutzt werden. Folglich ist es doch nicht so einfach möglich, aus einem `Student` einen `Professor` zu machen. Programmiertechnisch gesehen ist dies direkt natürlich gar nicht möglich. Der beste Lösungsansatz wäre zu dem `Student` ein neues `Professor`-Objekt zu erstellen und die einzelnen Daten mit Get- und Set-Methoden zu übertragen.

Das Beispiel soll deutlich zeigen, dass man nicht den `__proto__`-Verweis eines Objektes auf ein anderes Objekt verweisen lassen kann.

Man kann `__proto__` aber auch sinnvoll abändern, wenn man sich an die Regeln hält. So kann man ohne Probleme Klassen in den `__proto__`-Baum mit einbinden, um deren Attribute an weiter unten in der Hierarchie stehende Klassen oder Objekte zu vererben. Für diesen Fall gibt es zwar schon die Klasse `Object`, von der man das `prototype` Exemplar erweitern kann. Aber `Object` wird automatisch an alle Klassen und Exemplare vererbt. Mit der folgenden Methode kann man das Vererben sogar auf einzelne Objekte beschränken!

Da dies eine der grundlegende Sachen in Fash ist, bietet Flash MX die Methode `registerClass()`. Sie erspart den Umgang mit den `__proto__`-Verweisen. Die Anwendung dieser Methode haben wir im Praxisbeispiel in Kapitel 8.5 beschrieben. Im nachfolgenden gehen wir weiter auf die `__proto__`-Verweise und deren Manipulation ein.

Die folgende Grafik verdeutlicht, wie in Flash das Objekt- und Klassenmodell aufgebaut ist.

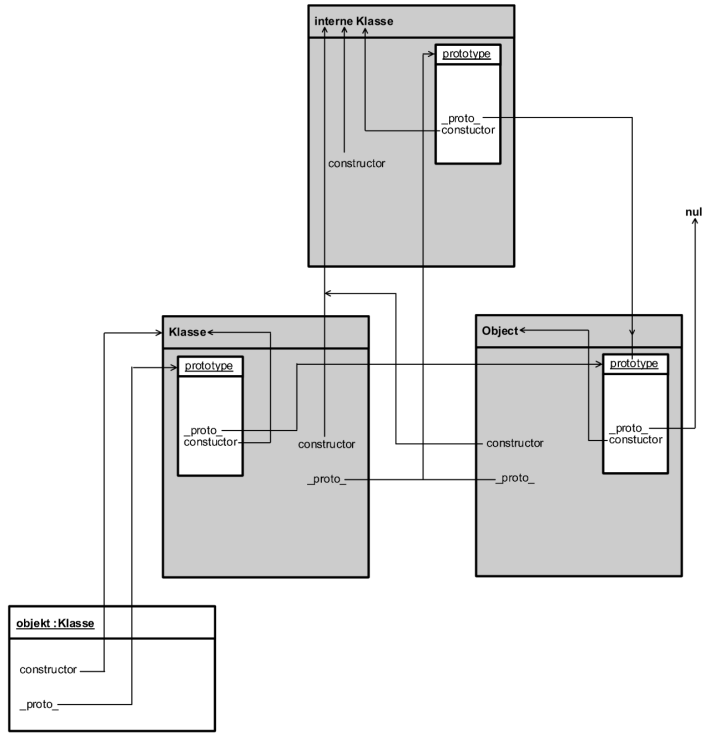


Abbildung 1.34: Aufbau des Objekt- und Klassenmodells in Flash

Zuerst sehen Sie eine Referenz namens `constructor`, die wir noch gar nicht erwähnt haben. Die `constructor`-Variable zeigt immer auf die Konstruktorfunktion, mit der das Exemplar gebildet wurde. Klassen besitzen zwar auch `__proto__`- und `constructor`- Variablen/Referenzen, diese sind aber nur der Vollständigkeit halber mit aufgeführt.

Wenn man zum Beispiel folgende Methode einer Klasse hinzufügt, kann man anschließend über deren Aufruf noch einmal die Konstruktorfunktion der Klasse aufrufen.

Listing 1.125: Aufruf der Konstruktorfunktion

```
Klassenname.prototype.rufeKon = function( ){
    this.constructor();
}
```

Oder man kann aus einem Objekt heraus, ohne den Klassennamen direkt zu erwähnen oder zu wissen, ein neues Objekt bilden:

```
objekt2 = new objekt1.__proto__.constructor();
```

oder auch

```
objekt2 = new objekt1.constructor();
```

Beide Befehle sind identisch, da jedes Mal der gleiche Konstruktor angesprochen wird.

Den `__proto__`-Baum per Hand zu verändern ist nur unter der Bedingung sinnvoll, dass man ein Objekt hat und speziell nur diesem Objekt etwas vererben möchte, bzw. einer Gruppe von Objekten.

Eines der Objekte, auf die man bei der Erstellung kaum Einfluss hat, ist das `movieclip`-Objekt. Man kann zwar der Klasse `Movieclip` noch Methoden unter Nutzung des `prototype`-Objektes vererben. Aber diese Methode würden dann automatisch alle Objekte referenzieren.

Im Folgenden erstellen wir eine Klasse `Fenster` mit einigen Methoden.

Die Datei zu diesem Beispiel finden Sie auf der CD-ROM unter *Kapitel1/09_proto.fla*.

Listing 1.126: Beispiel für Vererbung durch Änderung des __proto__-Baums

```
Fenster = function (name, x1, y1, x2, y2, farbe) {
    var referenz = this.createEmptyMovieClip(name,
        --Fenster.maxw);
    referenz.__proto__ = Fenster.prototype;
    referenz.beginFill(farbe, 100);
    referenz.moveTo(x1, y1);
    referenz.lineTo(x2, y1);
    referenz.lineTo(x2, y2);
    referenz.lineTo(x1, y2);
    referenz.endFill();
};
Fenster.prototype.__proto__ = movieclip.prototype;
Fenster.maxw = 999; //Klassenglobal
Fenster.prototype.onEnterFrame = function() {
    this._x++;
};
```

```
Fenster.prototype.onMouseDown = function() {
    if (this.hitTest(_root._xmouse, _root._ymouse)) {
        this.startDrag();
        this.drag = true;
    }
};
Fenster.prototype.onMouseUp = function() {
    if (this.drag) {
        this.stopDrag();
        drag = false;
    }
};
Fenster("w1", 50, 50, 100, 100, 0x999999);
Fenster("w2", 150, 150, 160, 160, 0x111111);
Fenster("w3", 160, 160, 200, 250, 0xFF0000);
```

Normalerweise wäre die Konstruktorfunktion von `Fenster` leer, da wir die Klasse nur zum Vererben benutzen und nicht, um Exemplare davon zu erstellen. Aber in dem Beispiel haben wir uns erlaubt, die eigentliche Konstruktorfunktion wieder als normale Funktion zu benutzen. Man hätte aber genauso gut eine zweite Funktion erstellen können.

Die erstellte Klasse `Fenster` wird durch `Fenster.prototype.__proto__ = movieclip.prototype;` zu der `Movieclip`-Klasse verlinkt, so dass `Fenster` alle Eigenschaften von `Movieclip` erbt. Durch `var referenz = this.createEmptyMovieClip(name, --Fenster.maxw);` wird ein Objekt vom Typ `Movieclip` erzeugt. Damit dieses nun die Methoden aus unserer erstellten Klasse `Fenster` erbt, müssen wir dessen `__proto__`-Verweis auf den `prototype` von `Fenster` ausrichten.

```
referenz.__proto__ = Fenster.prototype;
```

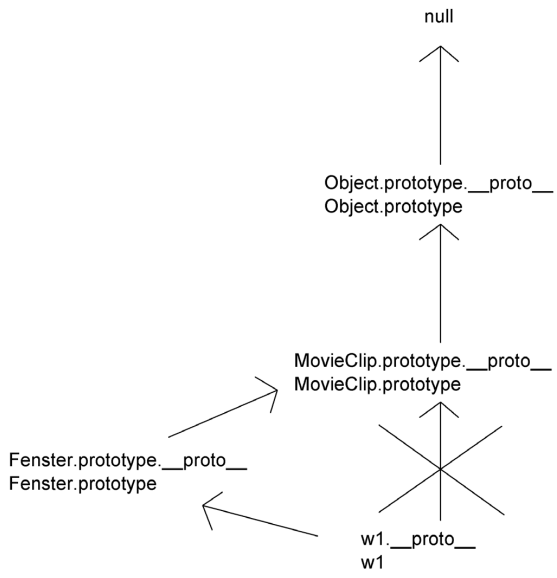


Abbildung 1.35: Einfügen einer Klasse in den __proto__-Pfad

Wir wollten Ihnen dieses interessante Feature von Flash nicht vorenthalten. Da Macromedia sich aber mit ActionScript an dem ECMA-245-Standard richten will, ist es fraglich, ob dies in späteren Versionen so noch zu programmieren ist. Wahrscheinlich wird `__proto__` in Zukunft schreibgeschützt sein.

Macromedia selbst verweist aber darauf, dass diese Praxis unüblich ist und `__proto__` unbedingt als schreibgeschützt betrachtet werden sollte! Für die Vererbung soll man die herkömmliche Methode benutzen, die wir Ihnen in Abschnitt „Vererbung“ ab Seite 111 vorgestellt haben und um Objekte von MovieClip einer selbstdefinierten Klasse zuzuordnen, wie hier im Beispiel, existiert die Methode `registerClass()`; siehe Kapitel 8.5.

Vordefinierte Klassen und Objekte

Flash beinhaltet von Haus aus implementierte Klassen und Objekte. Diese waren auch auf den vorangegangenen Grafiken erkennbar. Im Folgenden gehen wir auf die wichtigsten Klassen ein und zeigen ihre Vorteile.

▶ Object

Die oberste aller Klassen ist `Object`, jede andere Klasse, die wir erstellen, wird von `Object` abgeleitet. Zudem erben alle schon bestehenden Klassen von `Object`. `Object` selbst ist fast leer und besitzt die zwei Standardmethoden `toString()` und `valueOf()`, die auch alle anderen Klassen erben. Seit Flash MX besitzt die Klasse `Object` noch die Methoden `watch()`, `unwatch()`, `registerClass()` und `addProperty()`.

Durch den Zugriff auf `Object.prototype` kann man jedem Objekt, das man später erstellt, noch Variablen oder Methoden vererben. Im Folgenden erweitern wir die Klasse `Object` um eine Methode namens `traceAll()`. Die Methode gibt mit Hilfe einer for-in-Schleife alle Variablen eines Objektes oder einer Klasse aus.

Listing 1.127: Vererben einer Methode an Object

```
Object.prototype.traceAll = function(){
    trace("------------------------");
    for(var i in this)
        trace(i +" = "+ this[i]);
    trace("------------------------");
    return i;
}

ASSetPropFlags(Object.prototype, ["traceAll"], 1);

FKT1= function() {
    this.var1="Variable aus dem Konstructor";
};
FKT1.var2 = "Statische Variable";
Object1=new FKT1();
```

```
Object1.traceAll();
FKT1.traceAll();
```

Bildschirmausgabe:

```
------------------------
traceAll = [type Function]
var1 = Variable aus dem Konstructor
------------------------
------------------------
traceAll = [type Function]
var2 = Statische Variable
------------------------
```

Selbst definierte Methoden werden auch in for-in Schleifen mit aufgelistet. Dies ist natürlich unpraktisch, da man seine selbst geschriebene Methode nicht sehen möchte. Um dies zu verhindern, kann man sich einer ASNative Funktion bedienen, welche genau dies verhindert.

Was ASNative ist und welche Funktionen noch existieren, erfahren Sie in Kapitel 9.3.

Listing 1.128: Erstellen einer Methode in Object, die nicht in for-in-Schleifen angezeigt wird

```
Object.prototype.traceAll = function(){
    trace("------------------------");
    for(var i in this)
        trace(i +" = "+ this[i]);
    trace("------------------------");
    return i;
}
ASSetPropFlags(Object.prototype, ["traceAll"], 1);

FKT1= function() {
    this.var1="Variable aus dem Konstructor";
};
FKT1.var2 = "Statische Variable";
Object1=new FKT1();
Object1.traceAll();
FKT1.traceAll();
```

Bildschirmausgabe:

```
------------------------
var1 = Variable aus dem Konstructor
------------------------
------------------------
var2 = Statische Variable
------------------------
```

▶ Movieclip

Genauso wie man Object-Methoden über `prototype` vererbt, kann man dies auch bei der Klasse `Movieclip` bewerkstelligen. Dadurch bekommt jeder Movieclip diese Methode vererbt. Allerdings nicht selbst erstellte Objekte oder Klassen vom Typ `Object`!

Die Datei zum Beispiel finden Sie auf der CD unter *Kapitel1/10_revPl.fla*.

Diese Methode ist sehr nützlich: Es geht darum, einen Movieclip rückwärts abspielen zu lassen.

Vom Prinzip her ist das ActionScript relativ simpel, es muss in einem bestimmten Abstand immer wieder die Funktion `prevFrame()` aufgerufen werden. Dann braucht man noch eine Methode, die das Rückwärtsabspielen des Films wieder stoppt. Dementsprechend nennen wir die zwei Methoden `reversePlay()` und `reverseStop()`.

Listing 1.129: Methode zum Rückwärtsabspielen eines Movies

```
Movieclip.prototype.reversePlay = function(){
    if(this.reversePlayStarted==undefined){
        if (!(typeof (this.as) == "movieclip")) {
            this.reversePlayStarted=
                this.createEmptyMovieClip("AS", 20000);
        }else{
        this.reversePlayStarted=this.as;
        }
        this.reversePlayStarted.onEnterFrame = function() {
            if(_currentframe>1)
                prevFrame();
            else
                gotoAndStop(_totalframes);
        }
    }
}
ASSetPropFlags(Movieclip.prototype, ["reversePlay"], 1);
```

Wir haben uns erlaubt, eine Variable in dem Movie mit dem Namen `reversePlayStarted` zu „reservieren“. Wenn einmal die Methode gestartet wurde, muss man dafür sorgen, dass sie kein zweites Mal auf dasselbe Movie angewendet wird. Dementsprechend erfolgt zuerst die if-Abfrage auf `reversePlayStarted`. Zudem nutzen wir die Variable auch noch dafür, eine Referenz auf das erstellte Movie zu speichern. Wenn die Methode also noch nicht gestartet wurde, wird ein neuer (unsichtbarer) Movieclip erstellt, mit dem Namen `AS` und einer Tiefe von 20 000, vorausgesetzt, dass dieser noch nicht existiert. Danach erfolgt eine Zuweisung eines `onEnterFrame-Events`, das den eigentlichen Movieclip bei jedem Framewechsel um ein Frame zurückversetzt.

Listing 1.130: Methode zum Stoppen des rückwärts ablaufenden Movies

```
Movieclip.prototype.reverseStop = function(){
    this.reversePlayStarted.onEnterFrame = null;
    delete this.reversePlayStarted;
}
ASSetPropFlags(Movieclip.prototype, ["reverseStop"], 1);
```

Wie man sieht, setzt die Methode zuerst über die Referenz `reversePlayStarted`, die auf das erstellte Movie verweist, das `onClipEvent(enterFrame)` auf Null und löscht anschließend die Referenz. Dadurch wird das Starten der Methode `reversePlay()` wieder freigegeben.

Möglich ist es auch, die neue Funktion `setInterval` zu benutzen. Damit kann sogar die Abspielgeschwindigkeit verändert werden.

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel1/11_revP2.fla*.

Listing 1.131: Methoden mit variabler Abspielgeschwindigkeit

```
Movieclip.prototype.reversePlay = function(){
    if(this.reversePlayStarted==undefined){
        if(arguments.length<1){
            if (!(typeof (this.as) == "movieclip")) {
                this.reversePlayStarted=
                    this.createEmptyMovieClip("AS", 20000);
            }else{
            this.reversePlayStarted=this.as;
            }
            this.reversePlayStarted.onEnterFrame = function() {
                if(_currentframe>1)
                    prevFrame();
                else
                    gotoAndStop(_totalframes);
            }
        }else{
            this.reversePlayStarted=setInterval(function(){
                if(_currentframe>1)
                    prevFrame();
                else
                    gotoAndStop(_totalframes);
                updateAfterEvent();
            }, 1/arguments[0]*1000);
        }
    }
}
ASSetPropFlags(Movieclip.prototype, ["reversePlay"], 1);
Movieclip.prototype.reverseStop = function(){
    if((typeof (reversePlayStarted) == "movieclip")) {
        this.reversePlayStarted.onEnterFrame = null;
        delete this.reversePlayStarted;
    }else if(reversePlayStarted != undefined){
        clearInterval(this.reversePlayStarted);
        delete this.reversePlayStarted;
    }
}
ASSetPropFlags(Movieclip.prototype, ["reverseStop"], 1);
```

Der sicherlich interessante Block dürfte der Einsatz der Interval-Funktion sein:

```
this.reversePlayStarted=setInterval(function(){
```

Hiermit wird das Interval gesetzt, die Variable `reversePlayStarted` bekommt die Nummer der Interval-Funktion zugewiesen. Die Funktion endet mit den folgenden Parametern.

```
}, 1/arguments[0]*1000);
```

Es handelt sich dabei nur um einen weiteren Parameter, und zwar einer Zahl, die in Millisekunden die Intervalllänge angibt. Mit `arguments[0]` greifen wir auf den ersten übergebenen Parameter zu, der der Methode `reversePlay()` übergeben wurde. Das Teilen durch 1 und Multiplizieren mit 1000 sorgen für eine Umrechnung von bps bzw. fps in die benötigte Zeit in Millisekunden.

```
updateAfterEvent();
```

Das `updateAfterEvent` ist nötig, um das Movie schneller als die augenblicklich eingestellte Framerate laufen zu lassen. Es sorgt dafür, dass das Interval so oft wie eingestellt aufgerufen wird. Ohne `UpdateAfterEvent` würde die Intervallfunktion bei kleineren Werten als die Framerate des Movies nur einmal pro Framewechsel aufgerufen!

Für die Vorwärtsrichtung sieht es fast genauso aus.

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel1/12_ReFor.fla*.

Listing 1.132: Methoden zum beschleunigten Abspielen eines Movies

```
Movieclip.prototype.forwardPlay = function(){
    if(this.reversePlayStarted==undefined){
        if(arguments.length<1){
        this.reversePlayStarted=this.createEmptyMovieClip
        ("AS", 20000); if (!(typeof (this.as) == "movieclip")) {
            }else{
            this.reversePlayStarted=this.as;
            }
            this.reversePlayStarted.onEnterFrame = function() {
                if(_currentframe<=_totalframes)
                    nextFrame();
                else
                    gotoAndStop(0);
            }
        }else{
            this.reversePlayStarted=setInterval(function(){
                if(_currentframe<=_totalframes)
                    nextFrame();
                else
                    gotoAndStop(0);
                updateAfterEvent();
            }, 1/arguments[0]*1000);
        }
    }
}
ASSetPropFlags(Movieclip.prototype, ["forwardPlay"], 1);

Movieclip.prototype.forwardStop = function(){
    if((typeof (reversePlayStarted) == "movieclip")) {
        this.reversePlayStarted.onEnterFrame = null;
        delete this.reversePlayStarted;
```

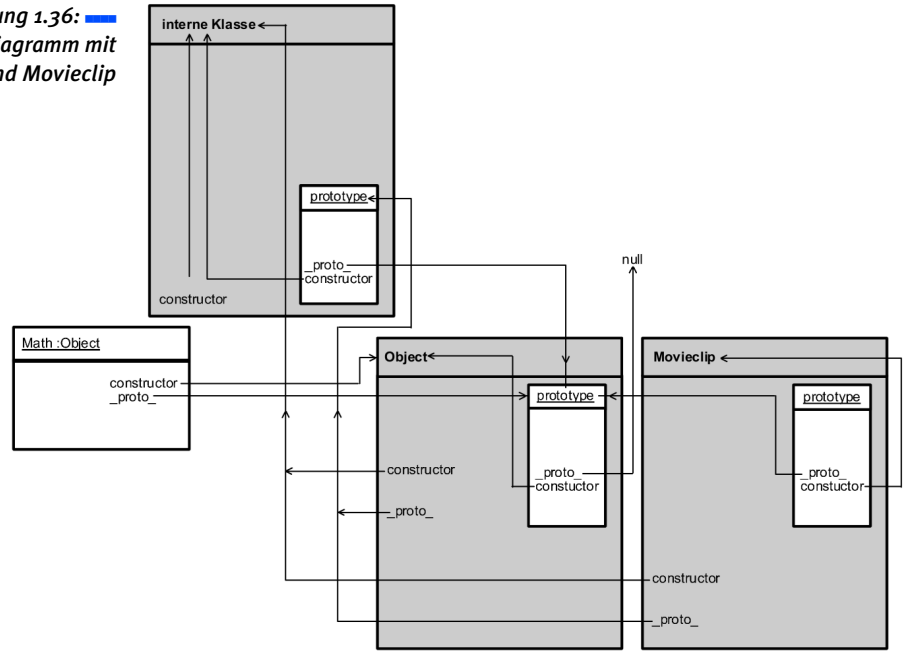
```
        }else if(reversePlayStarted != undefined){
            clearInterval(this.reversePlayStarted);
            delete this.reversePlayStarted;
        }
    }
    ASSetPropFlags(Movieclip.prototype, ["forwardStop"], 1);
```

Die Methoden bzw. das Programm sind analog zu dem vorangegangenen Beispiel aufgebaut.

Leider ist es nicht möglich, direkt die Funktionen von `play()` und `stop()` zu ändern, nur über Umwege, wie am Ende des __proto__-Kapitels beschrieben.

Dementsprechend war es einfacher, andere Namen für die Methoden zu wählen.

Abbildung 1.36: Klassendiagramm mit Object und Movieclip



▶ Diverse

Neben `Movieclip` und `Object` gibt es noch `Color`, `Math`, `Sound` und noch jede Menge andere Objekte und Klassen. Wir gehen im Folgenden auf das Mathe-Objekt ein. Da wir in Kapitel 1.5 ab Seite 67 in dem Beispiel „Probleme“ mit der random-Funktion hatten, wollen wir diese nun genau auf unsere Ansprüche umschreiben.

Die normale random-Methode des `Math`-Objektes liefert einen Floatwert zwischen 0 und 1.

In dem Beispiel wollten wir aber eine Zahl vorgeben, z.B. 4, und dementsprechend zufällig eine Zahl aus der Folge {0,1,2,3} bekommen.

```
newrandom = function(wert){
return Math.floor(Math.random()*wert);
}
```

Dies wäre die einfachste Art. Wenn man nun aber die random-Methode des `Math`-Objektes ersetzt, kann man dies natürlich auch direkt so geschickt erstellen, dass bei keiner Parameterübergabe die normale random-Methode ausgeführt wird, und mit Parameter unsere spezielle Funktion. Man kann dies auch noch so erweitern, dass bei zwei Parametern Ober- und Untergrenze gesetzt werden. Realisiert sieht es wie folgt aus:

Listing 1.133: Verbesserte random-Funktion

```
newrandom = function(){
    switch (arguments.length){
        case 0:
            return Math.random();
        case 1:
            return Math.floor(Math.random()*arguments[0]);
        case 2:
            if(arguments[0]<arguments[1])
                return Math.floor(Math.random()*(arguments[1]
                    -arguments[0]+1)+arguments[0]);
            return Math.floor(Math.random()*(arguments[0]
                -arguments[1]+1)+arguments[1]);
        default:
            trace("Error: Zuviele Parameter bei
                Math.random() aufruf!");
    }
}
```

Vielleicht interessant zu sehen ist, dass bei dem switch-case-Block keine Breaks existieren, da die return-Anweisung das Break quasi beinhaltet.

Für den Fall, dass zwei Parameter angegeben werden, müsste man eigentlich Oberwert und Unterwert angeben. Dementsprechend wird hier verglichen und immer der richtige Wert als Ober- und Unterwert ermittelt.

Das Problem ist, dass man die Methode `random` aus dem `Math`-Objekt nicht löschen kann. Da es sich aber um ein Objekt handelt, kann man ein neues Objekt erzeugen und dieses mit Hilfe von Object weiter vererben.

Man erzeugt also ein Objekt, das `Math` ersetzen soll. Dieses sollte alle Eigenschaften von `Math` besitzen. Dementsprechend zeigt der `__proto__`-Verweis des neuen Objektes auf das `Math`-Objekt. Man sieht wieder, dass ein Objekt auf ein Objekt verweist, also genau das passiert, was man eigentlich vermeiden wollte.

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel1/12_rando.fla*.

Listing 1.134: Überschreiben der Standard-random-Methode aus dem Math-Objekt

```
math2 = new Object();
  math2.__proto__=Math;
  math2.random2=Math.random;
  ASSetPropFlags(math2, ["random2"], 1);
  math2.random=function(){
    switch (arguments.length){
        case 0:
            return Math.random2();
        case 1:
            return Math.floor(Math.random2()*arguments[0]);
        case 2:
            if(arguments[0]<arguments[1])
                return Math.floor(Math.random2()*(arguments[1]-
                    arguments[0]+1)+arguments[0]);
            return Math.floor(Math.random2()*(arguments[0]
                -arguments[1]+1)+arguments[1]);
        default:
            trace("Error: Zu viele Parameter bei
                Math.random() aufruf!");
    }
};
ASSetPropFlags(math2, ["random"], 1);
Object.prototype.Math=math2;
ASSetPropFlags(Object.prototype, ["math"], 1);
```

Natürlich darf man auf keinen Fall vergessen, die Methoden für die for-in-Schleifen mit ASNative wieder „unsichtbar“ zu machen, da man irgendwann sonst sehr viel „Müll“ hätte und for-in-Schleifen überhaupt keinen Sinn mehr machten.

Die einfachere Version wäre es allerdings, direkt einen anderen Namen für die neue Methode zu wählen und diese dem `Math`-Objekt anzuhängen, wie im folgenden Beispiel. Dort heißt die erweiterte random-Methode dann `Random2`.

Listing 1.135: Erweitert das Math-Objekt um die Methode random2

```
math.random2=function(){
    switch (arguments.length){
        case 0:
            return Math.random();
        case 1:
            return Math.floor(Math.random()*arguments[0]);
        case 2:
            if(arguments[0]<arguments[1])
                return Math.floor(Math.random()*(arguments[1]
                    -arguments[0]+1)+arguments[0]);
            return Math.floor(Math.random()*(arguments[0]
                -arguments[1]+1)+arguments[1]);
        default:
            trace("Error: Zu viele Parameter bei
                Math.random() aufruf!");
    }
};
ASSetPropFlags(Math, ["random2"], 1);
```

Aufbau von Flash

Nachdem Sie im ersten Kapitel sehr ausführlich über alle Grundelemente von ActionScript informiert wurden, wollen wir nun weiter ins Detail gehen. Sicherlich haben wir auch schon in dem ein oder anderen Beispiel einen Button oder ein Movie benutzt. Dieses Kapitel beschreibt nun die Vorteile von Flash und seine optimale Anwendung.

Wenn wir in diesem Kapitel von Objekten reden, beziehen wir uns dabei nicht mehr auf durch ActionScript erzeugte Objekte oder Klassen, sondern auf Linien, Quadrate und Objekte, die auf der Bühne liegen. Dies können auch Movies oder Buttons sein.

2.1 Allgemeiner Aufbau

Zuerst erfolgt eine kurze Erläuterung der Grundelemente. Falls Ihnen diese bereits vertraut sind, lesen Sie am besten direkt im Kapitel 2.2 ab Seite 138 weiter, dort gehen wir dann auf das Wesentliche – ActionScript – ein.

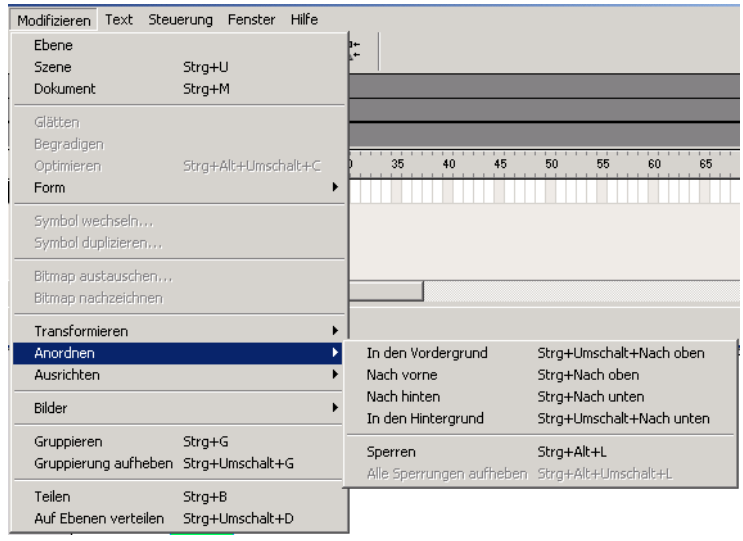
Hauptzeitleiste

Das kleinste Segment, in dem man etwas platzieren kann, ist ein Frame. Mehrere Frames bilden die Zeitleiste und ein Movie. Das Movie, das man beim Start eines neuen flas angezeigt bekommt, nennt man Hauptmovie. Die Frames des Hauptmovies kann man in Szenen unterteilen. Bei dieser Unterteilung werden die Frames – allerdings nur für den Flash-User, der die Anwendung erstellt – sichtbar, so dass man eine bessere Gliederung erhält. Es lässt sich jeweils immer nur eine Szene gleichzeitig darstellen. So kann man zum Beispiel in Szene 1 einen Preloader einbauen, während Szene 2 die Hauptseite beinhaltet. In Flash liegen die Szenen und damit die Frames direkt hintereinander, so erklärt sich auch, warum nach Ablauf von Szene 1 Szene 2 folgt. Jede Szene sieht aus wie eine neue Hauptzeitleiste, dies ist aber nicht der Fall. In Wirklichkeit unterteilt man die Hauptzeitleiste durch Szenen nur in einzelne Abschnitte.

Die Hauptzeitleiste besteht aus Frames, die verschiedenen Ebenen zugeteilt sind. Wenn Sie nur mit einer Ebene arbeiten, können Sie maximal ein Frame der Hauptzeitleiste zeitgleich anzeigen.

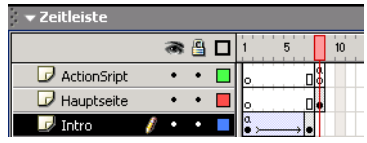
Jedes Objekt, das auf der Bühne angezeigt wird, liegt in einem Frame. Innerhalb eines Frames gibt es eine interne (nur auf das Frame bezogene) Tiefenverteilung für den Fall, dass mehrere Objekte in einem Frame liegen. Diese Tiefenverteilung können Sie über das Menü Modifizieren › Anordnen verändern.

Abbildung 2.1: Änderung der Tiefenreihenfolge von Objekten innerhalb eines Frames



Wenn man also mehrere Objekte in einem Frame erstellt, ist es möglich, die Tiefenreihenfolge zu verändern, allerdings wird diese nicht noch einmal extra optisch in Flash angezeigt. Um dies zu umgehen und um auch mehrere Frames zeitgleich anzeigen zu können, existieren die Ebenen.

Abbildung 2.2: Ebenenstruktur eines Movies



Die oberen Objekte in den höher liegenden Ebenen werden logischerweise auch auf der Hauptbühne über den tiefer liegenden Objekten angezeigt. Zudem kann man die Ebenen auch beschriften und so für mehr Übersicht im Film sorgen. Des Weiteren gilt: Hat man in zwei Ebenen z.B. im ersten Frame ein ActionScript liegen, wird das obere ActionScript zuerst ausgeführt.

Wenn Sie also die Zeitleiste betrachten, können Sie einen linear ablaufenden Film produzieren.

MovieClip

Die Hauptzeitleiste ist ähnlich aufgebaut wie ein MovieClip, dementsprechend hat man in einem Movie auch wieder Ebenen und Frames. Diese werden synchron (mit gleicher fps-Zahl) zu denen der Hauptzeitleiste abgespielt. Man benutzt ein Movie dazu, um z.B. ein Objekt zu bewegen, wenn sich diese Bewegung immer wieder wiederholen soll, wie z.B. ein sich drehendes E-Mail-Logo.

Die zugehörige fla-Datei finden Sie auf der CD unter *Kapitel2/oo_movie.fla*.

Die relative Tiefe des MovieClips zur Hauptzeitleiste hängt davon ab, wo dieser Movie auf der Hauptzeitleiste platziert wurde. Die Elemente in dem Movie selbst bilden eine eigene Gruppe, die dann relativ zu den restlichen Objekten platziert wird. Jeder Movie besitzt ein eigenes Koordinatensystem, nach dem die Objekte in diesem Movie ausgerichtet werden. Dementsprechend hat auch jeder Movie einen eigenen Nullpunkt. Dieser Nullpunkt ist durch einen kleinen Kreis gekennzeichnet, den man nun in Flash MX einfach bewegen kann, ohne direkt die innen liegenden Objekte mit zu verschieben. Dafür muss aber das Auswahlwerkzeug angewählt sein.

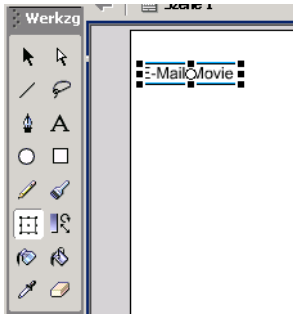


Abbildung 2.3:
Einfacheres Verschieben des Nullpunkts in Flash MX

Für den normalen Gebrauch in Flash ist der Nullpunkt eines Movies nicht so wichtig, wenn man allerdings mit ActionScript Objekte versetzt, gibt man die Koordinaten immer relativ zu dem Nullpunkt des Movies an.

Schaltflächen

Schaltflächen besitzen intern auch mehrere Ebenen, aber nur eine gewisse Anzahl an Frames. Der Nullpunkt der Schaltflächen kann analog zu dem Nullpunkt der Movies verändert werden.

Schaltflächen bieten im Gegensatz zu Movies automatisch Frames, die `mouseOver` oder `mouseOut` angezeigt werden, dafür aber keine Frames, die nacheinander abgespielt werden.

Eine genauere Beschreibung zu Schaltflächen finden Sie in Kapitel 4.3 ab Seite 179.

Grafiken

Um niemanden zu verwirren, sei an dieser Stelle erwähnt, dass es auch Grafiken gibt. Dies sind Objekte, die mehrere andere Objekte einfach nur zusammenfassen. Man benötigt sie, wenn man auf eine Zeichnung, z.B. ein Rechteck, einen Effekt anwenden möchte, wie z.B. Alpha. Außerdem muss man die besagte Zeichnung in eine Grafik einfügen, wenn man ein Bewegungs-Tweening erstellen möchte.

2.2 Objekt

Jeder Button oder jedes Movie, der/das sich auf der Hauptbühne befindet, ist ein Objekt von `MovieClip`. Leider kann man z.B. durch `new MovieClip` kein neues Movie erzeugen. Dies ist dafür aber Pproblemlos mit `createEmptyMovieClip()` möglich. Ähnliche Funktionalität bieten noch `attachMovieClip()` und `duplicateMovieClip()`, die jeweils eine Kopie eines in der Bibliothek oder auf der Bühne befindlichen Movies erzeugen.

Tiefenstruktur

Jedem Objekt in Flash ist eine bestimmte Tiefe zugewiesen. Wenn alle Objekte auf derselben Ebene, bzw. in demselben Objekt liegen, entspricht der geringsten Tiefe immer das hinterste Objekt.

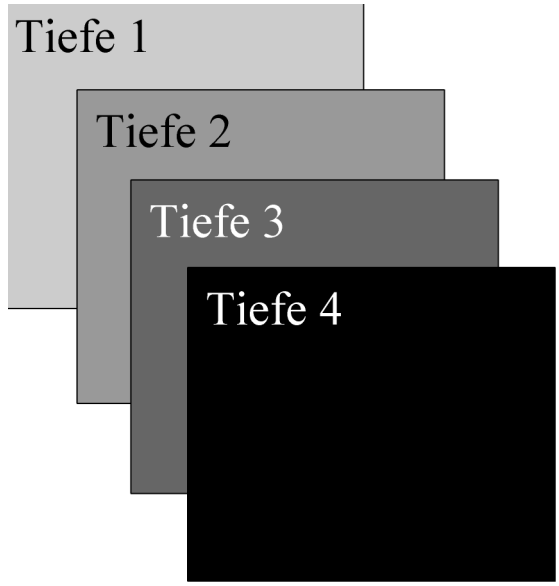


Abbildung 2.4: Tiefenanordnung bei gleicher Ebene/ gleichem Objekt

Diese Tiefen unterscheiden sich allerdings. So kann ein Objekt, das in einem Movie liegt und die Tiefe 8 hat, trotzdem oberhalb eines Objekts angezeigt werden, das noch eine höhere Tiefe hat, da die Tiefe des Movies ausschlaggebend ist, in dem dieses Objekt liegt.

Movie1 Tiefe 1
    - Button1 Tiefe 8
Movie2 Tiefe 2
    - Button2 Tiefe 1
Movie3 Tiefe 3

Obwohl der Button2 eine höhere Tiefe hat, wird er unterhalb von Button1 angezeigt.

Es gilt also: Die Tiefe des (Haupt-)Objekts, in dem sich ein Objekt befindet, ist ausschlaggebend für dessen Tiefe.

Die Elemente zur Verschachtelung von Objekten sind:

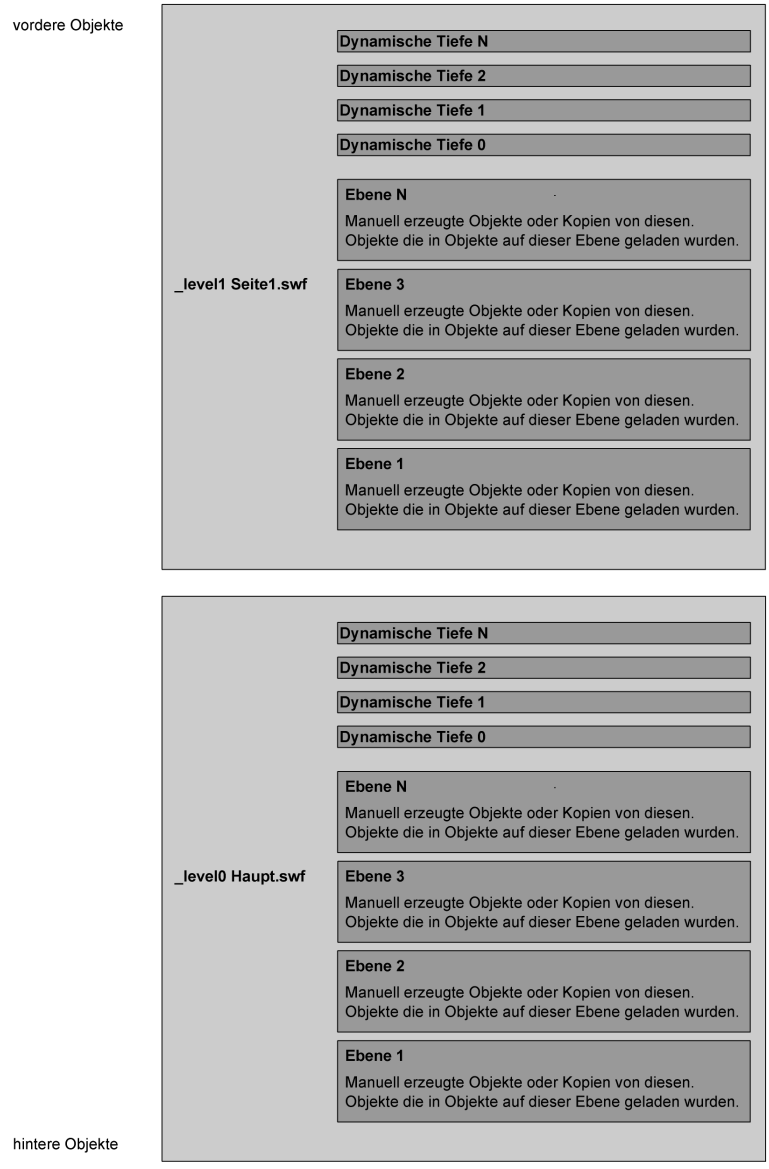
- Ebenen
- Movies oder Buttons
- Levels (SWF)

So enthält ein Level, bzw. das SWF, das in den Level geladen wurde, immer einen Hauptfilm (die Hauptzeitleiste), dieser besteht aus Ebenen. In den Ebenen kann man wieder neue Movies oder Buttons platzieren, die wiederum Ebenen enthalten. So kann man diese Dinge also theoretisch unendlich tief verschachteln.

Movies, die Sie dynamisch erzeugen, liegen nicht, wie man annehmen könnte, parallel zu den Objekten auf der Hauptbühne, sondern sie sind diesen übergeordnet. Solange man ein Movie ohne Bestimmungsort einlädt (selbstverständlich unter direkter Angabe, dass ein Movie in ein anderes Movie eingeladen werden soll, welches einen festen Platz in einer Ebene hat), übernimmt das eingeladene Movie dessen Tiefeposition.

Alle anderen dynamisch erzeugten Movies liegen über den Ebenen des Hauptmovies, egal wie niedrig deren Tiefe ist. Die folgende Grafik soll dies verdeutlichen.

Abbildung 2.5: Tiefenstruktur in Flash



Bei dem seite1.swf muss es sich in dem Fall um ein SWF handeln, das von dem Haupt.swf durch `loadMovie` in `_level1` eingeladen wurde. Noch zu erwähnen ist, dass Objekte in Flash MX eine Eigenschaft `_depth` besitzen, mit der man die Tiefe dieses Objekts abfragen kann. Mit dem Befehl `swapDepth()` kann man auch auf die Tiefe eines MovieClips Einfluss nehmen.

ActionScript-Struktur

Jedem Objekt kann ein Instanzname zugewiesen werden. Dies ist nun sogar bei Buttons oder Textfeldern möglich. In Flash 5 ging das noch nicht! Erst wenn ein Objekt (Movie, Button, Textfeld) einen Instanznamen besitzt, kann man dieses per ActionScript ansprechen. Den Instanznamen weisen Sie über die Eigenschaftsleiste zu. Es handelt sich um das zweite Textfeld von links oben. Eine nähere Beschreibung zu der Eigenschaftsleiste finden Sie in Kapitel 4.2 ab Seite 169.

Wenn Sie einen neuen Flashfilm erstellen, sind Sie auf der Hauptzeitleiste. Diese Hauptzeitleiste befindet sich (vorausgesetzt es handelt sich um dasjenige SWF, das später auch als Erstes gestartet wird) in `_level0`. Wenn Sie nun Variablen dieses SWFs ansprechen wollen, geht dies mit Hilfe der Punktsyntax.

```
_level0. variablenname
```

Wenn Sie ein anderes SWF in ein anderes Level, z.B. 5, einladen, können Sie auch dessen Variablen ansprechen.

```
_level5.variablenname
```

Hierbei handelt es sich um die Variablen, die sich in der Hauptzeitleiste befinden. Die jeweilige Hauptzeitleiste des aktuellen SWF (in dem Sie sich befinden) können Sie auch über `_root` erreichen. Dementsprechend wäre, wenn Sie sich in dem SWF befinden, das später in `_level0` zu finden ist (dies ist immer das SWF, das später zuerst gestartet wird), `_level0` identisch mit `_root`.

`_root` referenziert immer die aktuelle Hauptzeitleiste des SWFs, in dem der Befehl benutzt wurde.

Wenn Sie nun auf Ihrer Hauptzeitleiste ein Movie erstellen und diesem den Instanznamen „Movie1“ zuweisen, können Sie mit

```
_root.Movie1.variablenname
```

auf dessen Variablen zugreifen.

Selbstverständlich können Sie nicht nur Variablen setzen. Sie können dieses Movie auch stoppen, und zwar durch den Befehl

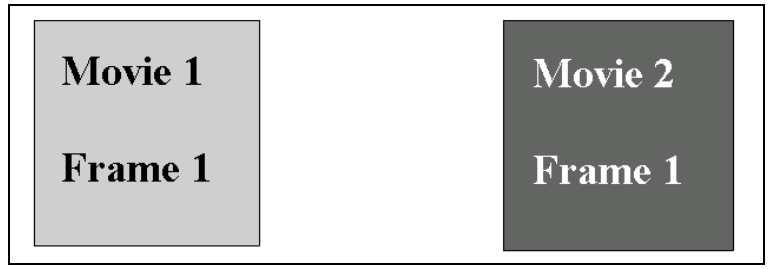
```
_root.Movie1.stop();
```

Dies sind jetzt immer genaue Angaben. Sie können aber auch relative Angaben machen. Z.B. wenn sich Ihr Script in der Hauptzeitleiste in einem Frame befindet, könnten Sie genauso gut das `_root` oder das `_level0` vor dem Aufruf weglassen, da Sie sich schon dort befinden.

Als Beispiel haben wir einmal auf der Hauptbühne eines Films zwei Movies platziert. Diese Movies enthalten jeweils drei Frames. Wir benutzen im Folgenden auch noch ein neues Flash MX Feature für die Zuweisung von Buttonereignissen. Dadurch kann man die Movies in unserem Beispiel gleich anklicken. Falls Sie sich jetzt fragen, wofür man denn dann noch Buttons bräuchte – ganz einfach: Wenn es nur Movies gäbe,

müsste man jedes Mal ein Script schreiben, damit der Button sich bei `mouseOver` ändert. Wenn man aber direkt unter Flash einen Button statt eines Movies erstellt, existieren dort Felder wie `mouseOver` oder `HitArea`. Mehr dazu in Kapitel 4.3 ab Seite 179.

Abbildung 2.6: Bild der zwei Movies



Die fla-Datei zu diesem Beispiel finden Sie auf der CD unter *Kapitel2/01_AS.fla*.

Wenn man nun also auf das linke Movie klickt, soll dieses dafür sorgen, dass das rechte Movie einen Frame weiter springt, und umgekehrt.

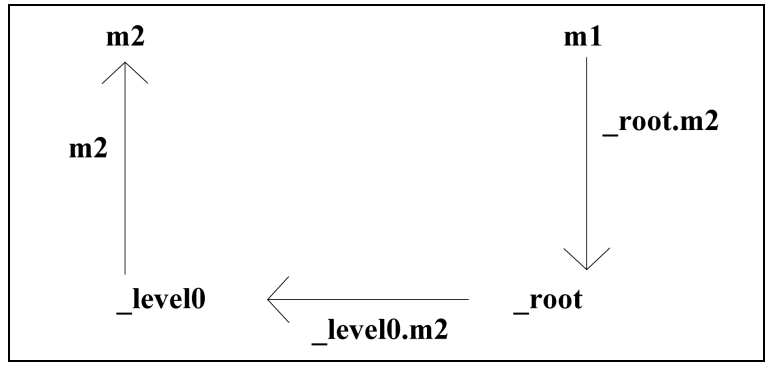
Den Movies haben wir die Instanznamen `m1` und `m2` zugewiesen. Das ActionScript von Movie1 sieht wie folgt aus:

```
onClipEvent(load){
    stop();
}
on(release){
    _root.m2.nextFrame();
}
```

Das ClipEvent sorgt dafür, dass Movie 1 im ersten Frame stoppt. Standardmäßig werden alle Movies abgespielt; da wir dies nicht wollen, haben wir es mit dem Befehl `stop()` gestoppt.

`on(release)` ist das Buttonereignis, wenn man also auf den Movie klickt und wieder loslässt, wird das darin liegende ActionScript ausgeführt. Dieses ActionScript setzt den Film, der sich auf der Hauptbühne dieses SWFs befindet und den Instanznamen `m2` hat, um einen Frame weiter.

Abbildung 2.7: Absolute Pfadangabe in Flash



Es handelt sich hierbei also um eine absolute Pfadangabe, da unabhängig davon, von wo dieses ActionScript ausgeführt wird, immer wieder der Movie `m2` auf der Hauptbühne angesprochen wird.

Das ActionSript des zweiten Movies könnte natürlich absolut identisch aussehen, nur dass statt `m2` `m1` angesprochen wird. Wir haben aber die Gelegenheit genutzt und bei dem Movie2 das ActionScript so abgeändert, dass es eine relative Pfadangabe benutzt.

```
onClipEvent(load){
    stop();
}
on(release){
    _parent.m1.nextFrame();
}
```

Mit `_parent` verweist man auf das eine Stufe höher liegende Objekt. Dies wäre in dem Fall also `_root`, deshalb führen beide Scripts auch quasi dasselbe aus.

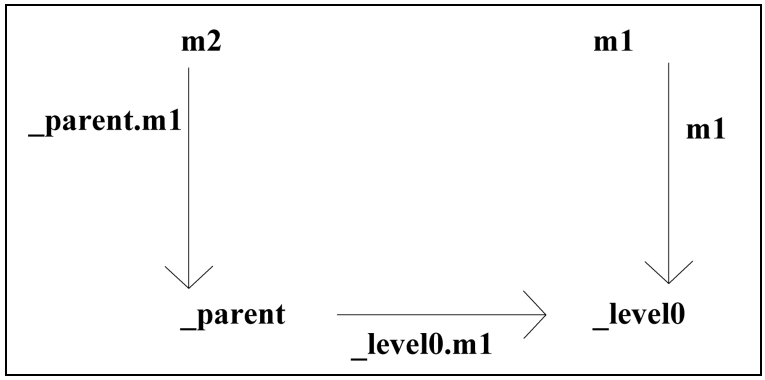


Abbildung 2.8: Relative Pfadangabe in Flash

Eine relative Pfadangabe benutzt man meistens dann, wenn es einfach kürzer ist, als die ganzen verschachtelten Instanznamen davor noch einmal aufzuzählen, oder dann, wenn es darum geht, ein Script flexibel zu halten. Der Nachteil einer absoluten Pfadangabe ist nämlich, dass man daran nichts mehr ändern darf. Man kann nicht einfach hier im Beispiel den rechten Film in noch einen weiteren MovieClip packen. Dann würde dieser nicht mehr das linke Movie ansprechen können.

Fassen wir noch einmal zusammen: `_root` referenziert also immer die Hauptbühne des SWFs, aus dem der Befehl aufgerufen wird, und `_parent` verweist immer auf das eine Stufe höhere liegende Objekt.

Jedes Movie besitzt auch bestimmte Eigenschaften, diese sind erkenntlich an den Unterstrichen vor den Namen. `_root`, `_parent` und `_level` bilden hier die Ausnahme.

Wenn Sie zum Beispiel einmal das obige Beispiel aufrufen und das ActionScript des ersten Movies wie folgt ändern, verschieben Sie das rechten Movie nach rechts.

```
onClipEvent(load){
    stop();
}
on(release){
    _root.m2._x+=5;
}
```

Das ActionScript des zweiten Buttons könnte man auch ändern um diesen wieder zurückzuverschieben.

```
onClipEvent(load){
    stop();
}
on(release){
    this._x-=5;
}
```

Beide Scripts sprechen den rechten MovieClip an und versetzen dessen x-Position jeweils um 5 Pixel.

Variablen und Eigenschaften von Movies sind immer erreichbar über den jeweiligen Pfad.

Es ist nicht möglich, Variablen Attribute wie *private* oder ähnlich zuzuweisen. Alle erstellten Variablen sind grundsätzlich dort gültig, wo sie auch erstellt wurden. Alle erstellten Variablen sind immer *lokal*. Es ist möglich, über `_global` globale Variablen zu erstellen.

```
_global.globalvariable="Inhalt";
```

2.3 Eventmodel

Sobald man ein etwas anspruchsvolleres Script bauen möchte, benötigt man ein Eventmodel. Wenn man ein Event benutzt, spricht man auch allgemein von einer Ereignissteuerung, da ein Script immer erst nach dem Eintreten eines bestimmten Ereignisses gestartet wird.

In Flash MX wurde das Eventmodel noch einmal drastisch erweitert. Damit besitzt nun so ziemlich jedes Objekt auch seine eigenen callBack-Funktionen. Außerdem kann man auf diese Weise fast allen Objekten nun auch listener zuordnen.

Filmereignisse (ClipEvent)

Jedem MovieClip, den Sie erstellen, können Sie ein ClipEvent zuweisen. Dieses Script wird nicht wie die bisherigen Scripts in der Zeitleiste nur beim Anzeigen des entsprechenden Frames, sondern beim Eintreffen des jeweiligen Ereignisses ausgeführt. Wie der Name schon sagt, sind diese Ereignisse alle an den Film gebunden.

Mit Hilfe von ClipEvents ist es also möglich, Scripts durch bestimmte Filmereignisse zu starten. Die Syntax sieht wie folgt aus:

```
onClipEvent(Ereignis){
    Code ();
}
```

Als Ereignisse existieren die folgenden:

- data wird ausgeführt beim Senden und Empfangen von Daten mit Hilfe von loadVariables oder loadMovie.
- enterFrame wird ausgeführt entsprechend der eingestellten BPS (FPS)-Zahl. Tritt immer beim Wechsel in ein neues Frame (Bild) ein.
- keyDown wird ausgeführt, sobald eine Taste der Tastatur (Keyboard) gedrückt wird.
- keyUp wird ausgeführt, sobald eine Taste der Tastatur (Keyboard) losgelassen wird.
- load wird ausgeführt, wenn der MovieClip in dem Movie erscheint, normalerweise also einmal zu Beginn, allerdings auch, wenn der Keyframe, in dem der Movie liegt, neu angesprochen wird.
- mouseDown wird beim Drücken der linken Maustaste ausgeführt.
- mouseMove wird ausgeführt, sobald die Maus bewegt wird.
- mouseUp wird beim Loslassen der linken Maustaste ausgeführt.
- unload wird beim Entladen des Movies ausgeführt, wenn dieser also nicht mehr in den angezeigten Frames auf der Hauptzeitleiste existiert.

Ein Beispiel zu dem load-Event finden Sie am Ende des Kapitels 2.2 im Abschnitt „ActionScript-Struktur“ ab Seite 141.

An dieser Stelle eine leicht erweiterte Version dieses Beispiels.

```
onClipEvent(load){
    stop();
    var1=2;
}
onClipEvent(enterFrame){
    this._x+=var1;
}
on(release){
    var1=-var1;
}
```

Listing 2.1: Verschiebt das Movie um 2 Pixel auf der x-Achse, durch Klicken kann die Richtung geändert werden

Erklären muss man dazu wohl nicht viel, das ActionScript in dem `onClip-Event(enterFrame)` wird immer wieder ausgeführt, dadurch werden die MoviesClips auf der Bühne verschoben. Schauen Sie sich am besten direkt einmal das Beispiel an.

Die fla-Datei zu diesem Beispiel finden Sie auf der CD unter *Kapitel2/o2_AS.fla*.

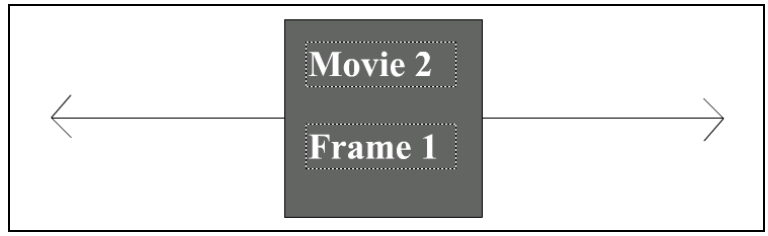


Abbildung 2.9: MovieClip wird auf der x-Achse durch ActionScript verschoben

Mausereignisse

So wie es mit dem `onclipEvent` filmgesteuerte Ereignisse gibt, existieren auch mausgesteuerte Ereignisse.

Mit Hilfe von `on(Event)` ist es also möglich, Scripts durch bestimmte Mausereignisse zu starten. Die Syntax sieht wie folgt aus:

```
on (Ereignis){
    Code ();
}
```

Als Ereignisse existieren die folgenden:

- dragOut wird ausgeführt, wenn die Maus über der Schaltfläche gedrückt und anschließend außerhalb losgelassen wird.
- dragOver wird ausgeführt, wenn die Maus über der Schaltfläche gedrückt und anschließend runter und wieder auf die Schaltfläche gezogen und losgelassen wird.
- keyPress (Tastencode) wird ausgeführt, wenn die angegebene Taste gedrückt wird. Die zulässigen Tastencodes finden Sie in Anhang A.
- press wird ausgeführt, wenn sich die Maus über der Schaltfläche befindet und die Maustaste gedrückt wird.
- release wird ausgeführt, wenn die Maus sich über der Schaltfläche befindet und die Maustaste losgelassen wird.
- releaseOutside wird ausgeführt, wenn die Maus über der Schaltfläche gedrückt und anschließend außerhalb der Schaltfläche losgelassen wird.
- rollOut wird ausgeführt, wenn der Mauszeiger den Schaltflächenbereich verlässt.
- rollOver wird ausgeführt, sobald die Maus über den Schaltflächenbereich gezogen wird.

Ein Beispiel zu den Mausevents zeigte bereits das Beispiel im vorigen Kapitel zu den ClipEvents. Mausevents können in Flash MX nun auch auf MovieClips angewendet werden!

Weitere Informationen und Beispiele zu Schaltflächen finden Sie in Kapitel 4.3 ab Seite 179.

callBack-Functions (Rückruf-Funktionen)

Hierbei handelt es sich um ein neues Feature von Flash MX. Dadurch besitzen jetzt bestimmte Klassen vordefinierte Methoden. Diese Methoden sind leer implementiert und werden zu bestimmten Ereignissen automatisch von Flash aus aufgerufen. Wenn man diese Methoden mit einer Funktion mit Inhalt überschreibt, hat dies zur Folge, dass Flash diese Funktion beim Eintreten des Ereignisses automatisch aufruft. Da man dafür einem Objekt eine Funktion mit übergibt und das Objekt eigentlich bestimmt, wann diese Funktion ausgeführt wird, spricht man von Rückruf-Funktionen (callBack-Funktionen).

Die Handhabung dieses neuen Features gestaltet sich sehr leicht. Man spricht einfach eine vordefinierte Ereignisprozedur an, wie z.B. `onChanged` eines Textfelds, und weist diesem eine Funktion zu. Im Folgenden haben wir ein Beispiel mit zwei Textfeldern erstellt und einer winzigen „Übersetzungsdatenbank“ vom Deutschen ins Englische bzw. umgekehrt.

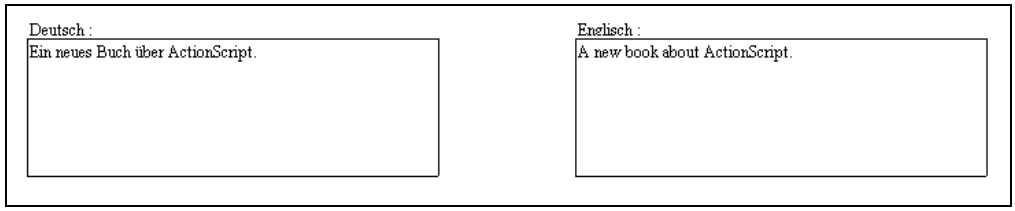


Abbildung 2.10: Ausgabebildschirm des Beispiels

Die fla-Datei zu diesem Beispiel finden Sie auf der CD unter *Kapitel2/o3_callb.fla*.

Zuerst wurden Schriftart, -farbe und -größe definiert.

```
stFormat = new TextFormat();
stFormat.color = 0x000000;
stFormat.font = "Arial";
stFormat.size = 16;
```

Danach wurden die vier Textfelder erstellt.

```
createTextField("T1u", 3, 100, 84, 300, 16);
T1u.selectable = false;
T1u.setTextFormat(stFormat);
T1u.text = "Deutsch :";

createTextField("T1", 1, 100, 100, 300, 100);
T1.type = "input";
T1.multiline = true;
T1.border = true;
T1.setTextFormat(stFormat);
```

```
createTextField("T2u", 4, 500, 84, 300, 16);
T2u.selectable = false;
T2u.setTextFormat(stFormat);
T2u.text = "Englisch :";

createTextField("T2", 2, 500, 100, 300, 100);
T2.type = "input";
T2.multiline = true;
T2.border = true;
T2.setTextFormat(stFormat);
```

Weitere Informationen zu Textfeldern finden Sie in Kapitel 4.2 ab Seite 169.

```
T1.onChanged = function() {
var Texteng=["A ","new ","book ","about "];
var Textdeu=["Ein ","neues ","Buch ","über "];
var text2=T1.text;
for(i=0;i<Textdeu.length;i++)
     text2 = text2.split(Textdeu[i]).join(Texteng[i]);
T2.text = text2;
}
```

Hier wurde der Rückruf-Funktion `onChanged` von Textfeld `T1` eine Funktion zugewiesen. Zuerst werden zwei Arrays mit den englischen und deutschen Wörtern erstellt.

Danach wird der Text des ersten Textfeldes `T1` in einer Variable `text2` zwischengespeichert. Eine for-Schleife durchläuft die Arrayinhalte und tauscht falls zutreffend die Wörter in der Variable `text2` aus. Dies würde man natürlich, wenn man wirklich eine Datenbank zum Übersetzen bauen wollte, nicht so programmieren, aber es dient hier als Beispiel für die Rückruf-Funktion.

Diese Funktion wird nicht immer wieder aufgerufen, sondern nur dann, wenn der User einen Text in das erste Textfeld `T1` eingibt oder an dem bestehenden Text etwas ändert.

```
T2.onChanged = function() {
var Texteng=["A ","new ","book ","about "];
var Textdeu=["Ein ","neues ","Buch ","über "];
var text2=T2.text;
for(i=0;i<Textdeu.length;i++)
     text2 = text2.split(Texteng[i]).join(Textdeu[i]);
T1.text = text2;
}
```

Fast genau die gleiche Funktion wird Textfeld `T2` zugewiesen, mit dem Unterschied, dass hier die Arrays vertauscht werden, da man ja nun genau umgekehrt übersetzen möchte.

Es existiert noch eine ganze Menge an callBack-Funktionen. Textfelder besitzen z.B. auch noch die callBack-Funktionen

- onKillFocus
- onScroller
- onSetFocus

MovieClips

- onData
- onDragOut
- onDragOver
- onEnterFrame
- onKeyDown
- onKeyUp
- onKillFocus
- onLoad
- onMouseDown
- onMouseUp
- ... und noch viele mehr.

Welche callBack-Funktionen vorhanden sind, sehen Sie unter Flash bei dem jeweiligen Objekt unter dem Punkt Ereignisse.

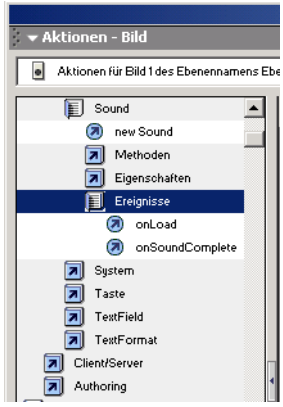


Abbildung 2.11:
callBack-Funktionen des Soundobjekts

Listener

Listener ermöglichen es, Objekte einem anderen Objekt „zuzuordnen“. Hierbei „hört“ das eine Objekt dann auf bestimmte Ereignisse des anderen Objekts. Es stellt zum Teil eine Erweiterung für die callBack-Funktionen dar.

Während man einer callBack-Funktion immer nur eine Funktion zuweisen kann, ist es mit Listenern möglich, mehrere Funktionen aufrufen zu lassen.

Die callBack-Funktion `onChanged` des Textfeldes kennen Sie bereits aus dem vorangegangenen Kapitel. Nun werden wir ein Objekt mit Hilfe der Listener dem Textfeld zuordnen.

```
stFormat = new TextFormat();
stFormat.color = 0x000000;
stFormat.font = "Arial";
stFormat.size = 16;

createTextField("T1u", 3, 100, 84, 300, 16);
T1u.selectable = false;
T1u.setTextFormat(stFormat);
T1u.text = "Eingabe :";

createTextField("T1", 1, 100, 100, 300, 100);
T1.type = "input";o
T1.multiline = true;
T1.border = true;
T1.setTextFormat(stFormat);
```

Zuerst wird wieder ein Textfeld für die Eingabe erzeugt.

```
obj1=new Object();
obj1.onChanged = function() {
trace("obj1");
}
T1.addListener(obj1);
```

Hier wird ein Objekt `obj1` erstellt, welches die Methode `onChanged` enthält. Damit diese Methode bei Änderung des Textinhaltes von Textfeld `T1` aufgerufen wird, muss man das Objekt mit `addListener` von `T1` aufrufen.

```
obj2=new Object();
obj2.onChanged = function() {
trace("obj2");
}
T1.addListener(obj2);
```

Das Gleiche geschieht noch einmal mit einem zweiten Objekt.

Wenn Sie sich das Beispiel einmal anschauen, sehen Sie, dass automatisch auch beide Methoden onChanged der zwei Objekte aufgerufen werden, sobald sich der Text in dem Textfeld `T1` ändert.

Die fla-Datei zu diesem Beispiel finden Sie auf der CD unter *Kapitel2/o4_liste.fla*.

Neuerungen in Flash MX

3.1 Grundlegendes

Nebst neuer Namensgebung in der sechsten Version hat Flash MX mit einer Menge Neuerungen aufzuwarten. Die Oberfläche hat ein neues Design erfahren, wie man auf Anhieb merkt, und sollte dieses nicht gefallen, so kann man es sogar selbst noch variieren. Viele ActionScript-Funktionen und Berechnungen sind schneller geworden, zumeist sogar um ganze Faktoren.

Im Folgenden werde ich kurz auf die neuen Funktionalitäten eingehen, damit Sie einen Überblick erhalten.

What's new

- **Vereinheitlichtes Design**

In Flash MX wurden viele Bedienfelder nach Zugehörigkeit gruppiert, so gehört zum Beispiel zum Textwerkzeug nun ein sehr umfangreiches und übersichtliches Bedienfeld. Musste man sich vorher noch mehrere Dialogfelder öffnen um in alle Einstellungen Einsicht zu erlangen, so ist diese Übersicht nun sehr schön zusammengeführt.

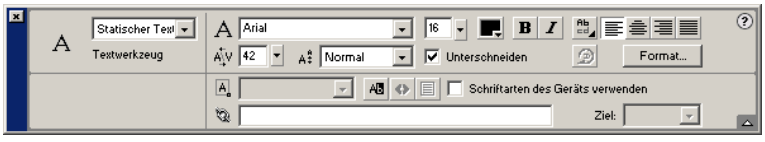
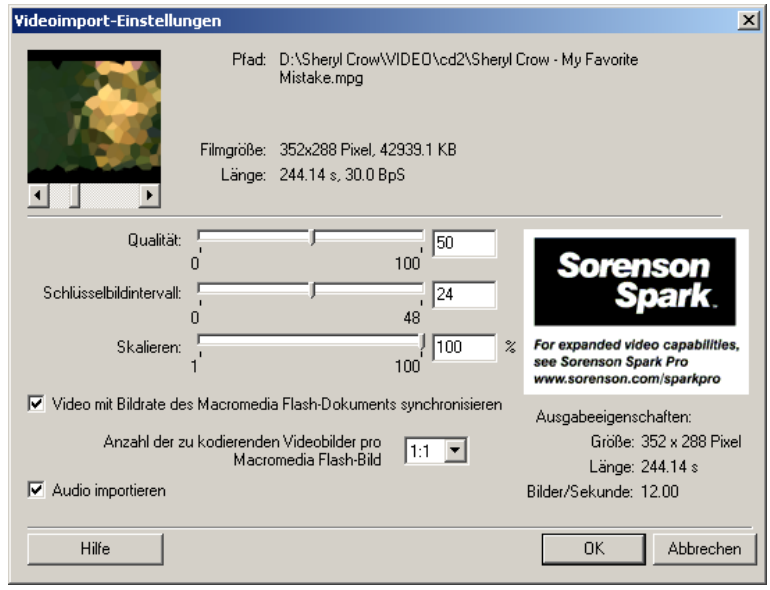


Abbildung 3.1: Alles auf einen Blick – eines der vielen neuen Dialogfelder

- **Erweiterte Videoimportmöglichkeiten**

Flash bietet nun die Möglichkeit Videos verschiedener Formate einzubinden. Das im Folgenden abgebildete PlugIn ermöglicht es dem Benutzer, Videodateien neu zu formatieren (Auflösung, Framerate, Qualität zu ändern).

Abbildung 3.2: Der neue Videoimport in Flash MX

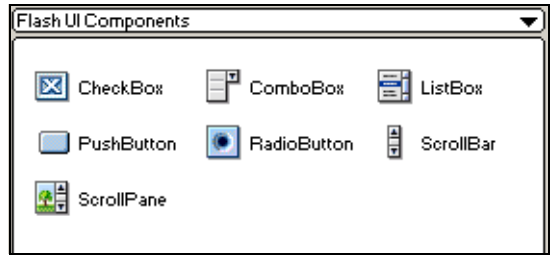


Das Ergebnis ist ein der Videodauer entsprechend langes Schlüsselbild, das sich einfach einbinden lässt.

▶ Flash-Komponenten

Die Komponenten sind im Grunde schon Bekannte aus Flash 5, auch wenn es hier Erweiterungen gab und die Liste der Komponenten größer wurde. Sie lassen sich einfach per Drag&Drop platzieren und die Eigenschaften gewohnt einfach editieren.

Abbildung 3.3: Die Komponenten erlauben das einfache und schnelle Erstellen von Formularen



Haben Sie eine Komponente platziert und klicken Sie diese an, so sehen Sie unter dem Dialogfeld EIGENSCHAFTEN alle einstellbaren Optionen. Dies sollte es Ihnen ermöglichen, ohne viel Aufwand z.B. Formulare zu erstellen. Natürlich sind auch andere Funktionen denkbar, so kann man z.B. das Dropdown auch zu Navigationszwecken gebrauchen. Wie gewohnt ist hier auch der Onlinesupport von Macromedia sehr gut, so dass Sie einige neue Komponenten dort auch herunterladen können.

▶ Erweiterter Debugger

Schon Flash 5 hatte einen erweiterten Debugger, was die Fehlersuche erheblich erleichterte (man erinnert sich noch gerne an die `trace()`-Funktion aus den Flash5-Vorgängern).

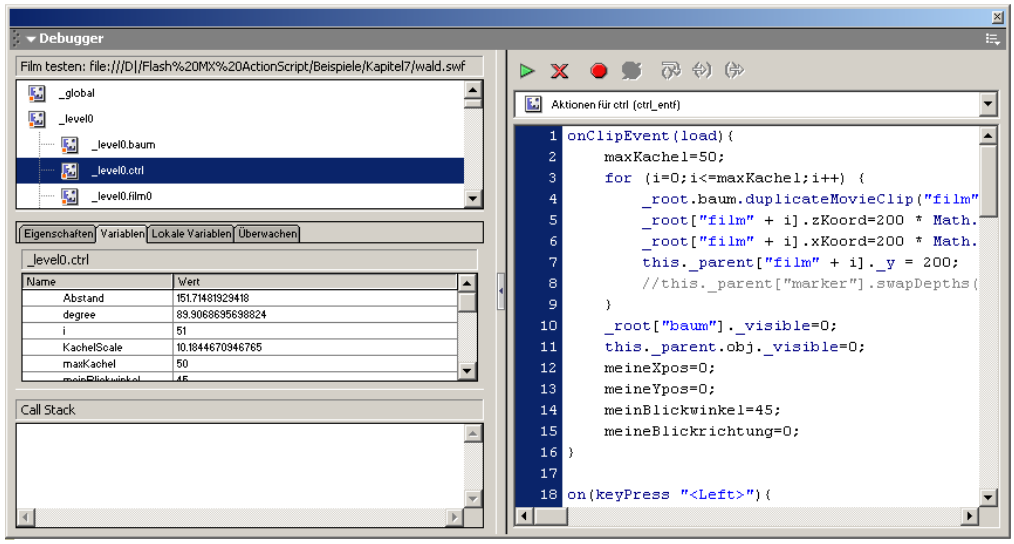


Abbildung 3.4: Der neue Debugger

Der Debugger, auf den noch in Kapitel 9.2 ab Seite 324 näher eingegangen wird, bietet Ihnen eine Vielzahl von Möglichkeiten Fehler zu suchen, wenn es sein muss auch online. Manchmal lassen sich nur so fehlende Referenzierungen nachvollziehen. Eingestellt in den EINSTELLUNGEN FÜR VERÖFFENTLICHUNG kann man natürlich auch einen Passwortschutz festlegen, möchte man doch nicht, dass jeder Einblick in den Quellcode erhält.

▶ Ebenenordner

Das Anlegen von Ebenenordnern ermöglicht eine verbesserte Übersichtlichkeit der Zeitlinien.

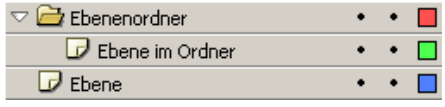


Abbildung 3.5: Ebenenordner zur verbesserten Übersichtlichkeit der Zeitleiste

Oft legt man verschiedene Seiten eines Webs versetzt in verschiedene Zeitleisten, dort kommen dann jeweils ja noch Extra-Zeitleisten hinzu, möchte man z.B. Tweenings dort ablaufen lassen. Hier lernt man schnell die Ordnerstruktur zu schätzen.

▶ Dynamisches Laden von Bildern

Schon lange auf der so genannten „Wishlist“, hat diese Funktion endlich die Aufnahme in Flash geschafft. Bislang war man auf andere Programme angewiesen, damit Bilder, die z.B. für eine Bildergalerie zur Verfügung stehen, erst in SWF umgewandelt werden oder diese schon von Anfang an in Flash eingebunden/importiert sind. Mit dieser Funktion ist es nun möglich, Bilder in Flash zu laden ohne diese vorher konvertieren zu müssen. Die linke obere Ecke des Bildes liegt später im Ursprungspunkt der Instanz, in die dieses Bild geladen wurde, was ein Zentrieren von Bildern erschwert, aber vielleicht wird es hierfür Funktionen in der nächsten Programmversion geben.

Haben Sie in eine Instanz ein Bild geladen, also einen Aufruf in Form von `_root.bild.loadMovie("C:\MeinBild.JPG");` so wird das `duplicateMovieClip()` auf die Instanz „Bild“ angewendet nicht richtig funktionieren, die Kopie der Instanz wird leer sein.

▶ Dynamisches Laden von MP3

Genauso wie in der zuvor beschriebenen Funktion lässt sich auch mittels `loadMovie` MP3 in Flash einbinden. Auf die Steuerung von Sound wird in Kapitel 5.5 ab Seite 224 noch ausführlich eingegangen. Dieses Feature ermöglicht es, eine Online-Jukebox zu erstellen, ohne Sounds vorher in das SWF-Format zu konvertieren.

▶ Rund um das ActionScript

Neben einer Erweiterung des Sprachumfangs und diversen Funktionen, die schon lange auf der Wunschliste standen (dynamisches Laden von MP3 und JPEG), wurde auch die Entwicklungsumgebung den neuen Gegebenheiten angepasst. Bei Eingabe von Befehlen werden alle Parameter, die zur Auswahl stehen, in einer DropDown-Liste angeboten. Sollten Sie schon mal mit Visual Basic gearbeitet haben, so wird Ihnen das bestimmt bekannt vorkommen. Eine Sache, die es leider noch nicht gibt, ist die Anzeige der Parameter für selbst definierte Funktionen, auch ist der Typ nicht angezeigt, etwas Vorwissen, was z.B. „Ziel“ sein kann, sollte man schon haben.

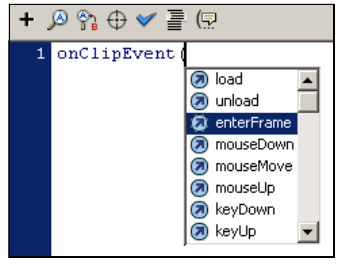


Abbildung 3.6: Gibt es eine begrenzte Auswahl von Parametern, so werden diese zur Auswahl angezeigt

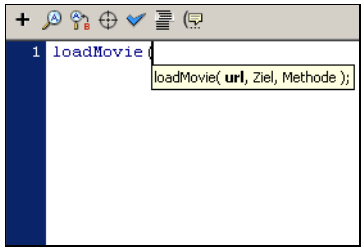


Abbildung 3.7:
Die Anzahl und Art der Parameter wird angezeigt, sollten mehrere Werte erlaubt sein

Videos in Flash

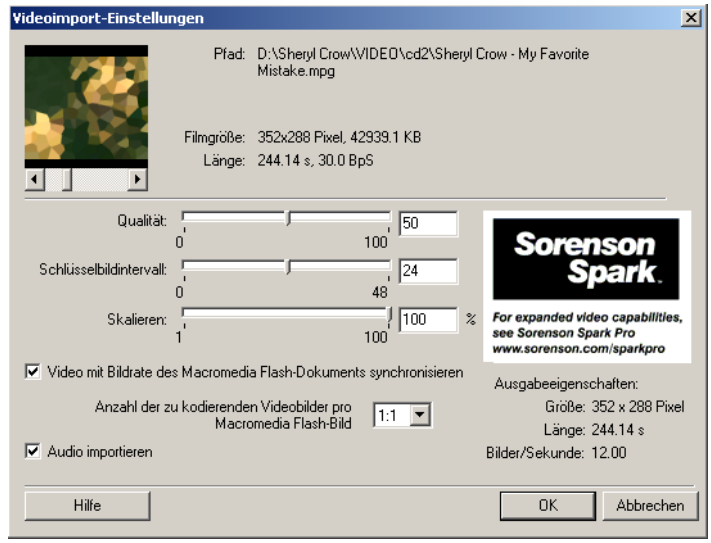
Auch wenn das Einbinden von Videos nicht unbedingt in ein ActionScript-Buch passen mag, so ist es auch ein Buch für fortgeschrittene Anwender. Videosequenzen im richtigen Maße können sehr schöne Effekte erzeugen. Auf unserer Webseite haben wir in der zurzeit aktuellen Version davon auch Gebrauch gemacht – auch wenn unser Video komplett aus dem Computer kommt.

Wollen Sie Videos in Flash einbinden, so haben Sie grundsätzlich hierfür zwei Möglichkeiten von Flash aus. Zum einen können Sie den mitgelieferten Import nutzen, Flash rechnet Ihnen das Video in einzelne Bilder um, oder Sie erzeugen eine eigene Bildsequenz (fortlaufend nummeriert), importieren diese in Flash (was zunächst das gleiche Ergebnis liefert wie der Import) und lassen dann alle Bilder nachzeichnen unter Modifizieren › Bitmap nachzeichnen. Dies müssen Sie dann für alle Bilder wiederholen, was aber angesichts der Funktion, eigene Tastenkürzel anzulegen, kein großes Problem ist. Im Folgenden werde ich auf diese beiden Möglichkeiten eingehen.

▶ Videos importieren (pixelbasiert)

Sind alle Einstellungen einmal vorgenommen, läuft das Programm zügig durch. Das Ergebnis ist ein einziger Keyframe in der Länge des Videos. Das Video selbst ist pixelbasiert, das heißt wird es skaliert, nimmt die Qualität ab.

Abbildung 3.8: Der Videoimport-Filter – hier werden alle Einstellungen einmalig vorgenommen

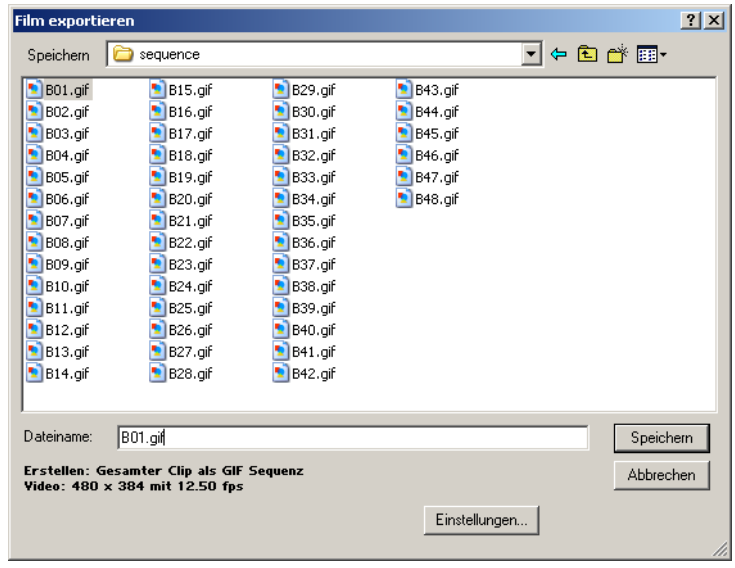


Da pixelbasierte Videos nicht immer in Flash passen, erfahren Sie im folgenden Teil, wie man Videos vektorisiert und so nette Kunsteffekte erhält.

▶ Videos importieren (vektorisiert)

Zuerst muss man sich eine geeignete Sequenz aussuchen. Da Vektorengrafiken ja aus Farbflächen bestehen, sollte man auf kontrastreiche Aufnahmen achten. Diese importiert man sich z.B. in Adobe Premiere® und exportiert von dort aus eine Bilderfolge in geeigneter Größe. Unter Datei › Clip exportieren finden Sie die Möglichkeit auch .gif als Format anzugeben.

Abbildung 3.9: Der Dialog zum Bildexport, hier mit schon erzeugter Liste



Der nächste Schritt ist optional, aber zu empfehlen. Wenn Sie etwas vertraut mit Adobe Photoshop® sind, dann werden Sie auch die Stapelverarbeitung kennen. Zunächst legen Sie eine Aktion an, die das Bild in große Farbflächen unterteilt. Dies macht es später Flash einfacher, das Bitmap nachzuzeichnen, und sieht im Ergebnis auch effektiv besser aus.

Die Aktionen finden Sie direkt neben dem Protokoll im Photoshop, die Aktion muss von Ihnen angelegt werden.

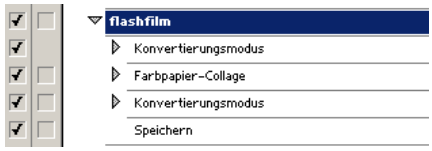


Abbildung 3.10: Die Aktion in Adobe Photoshop®

Als Nächstes wenden Sie auf alle Bilder diese Aktion an und erhalten somit eine bearbeitete Bildsequenz.

Den Dialog erhalten Sie unter Datei › Automatisieren › Stapelverarbeitung.

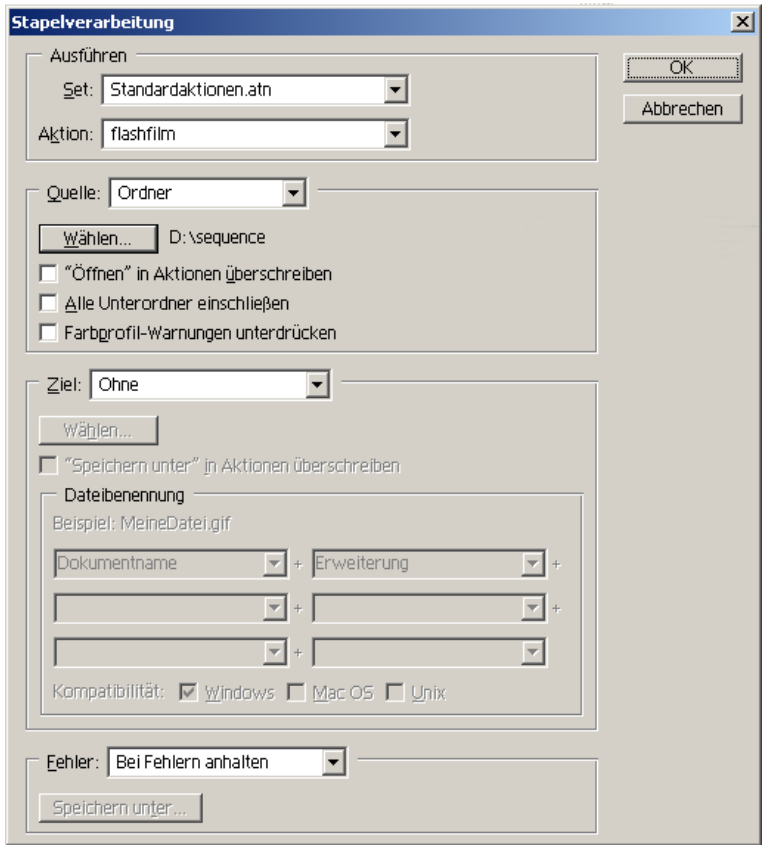
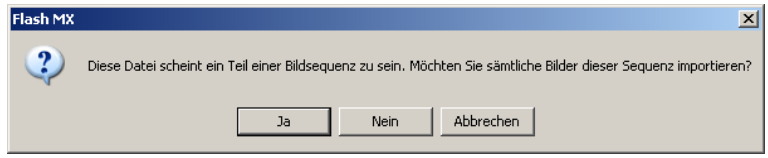


Abbildung 3.11: Der Name „flashfilm“ entspricht dem von Ihnen gewählten Namen

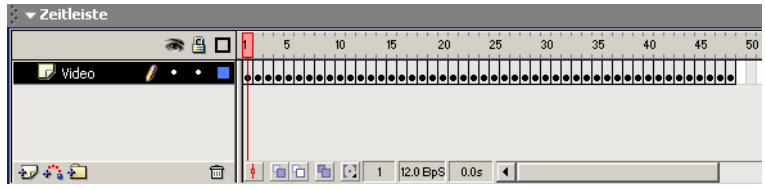
Importieren Sie jetzt das erste Bild in Flash, folgender Dialog wird wegen der durchlaufend nummerierten Bildfolge nun erscheinen.

Abbildung 3.12: Natürlich möchten Sie die ganze Sequenz importieren



Folgendes Ergebnis werden Sie nun erhalten, und auch wenn es nach viel Arbeit aussieht – dank Shortcuts reduziert sich die Arbeit zu einer Fingerübung.

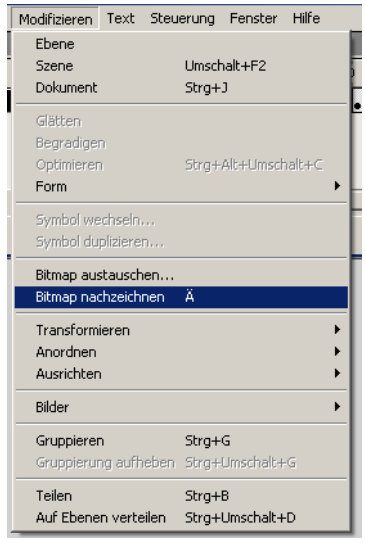
Abbildung 3.13: Diese Bilder müssen alle von Flash nachgezeichnet werden

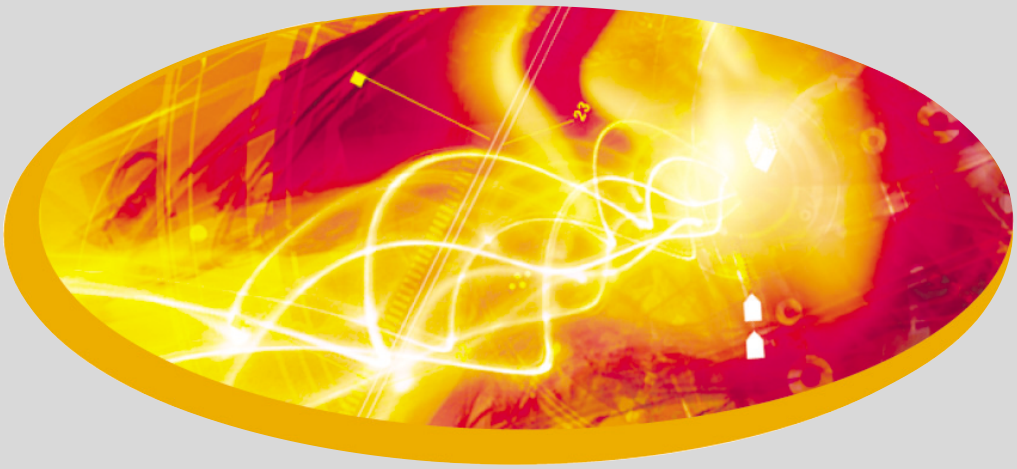


Folgend bietet sich der Shortcut auf der Taste „Ä“ an, da nur noch ↵ gedrückt werden muss, wenn die Einstellungen für BITMAP NACHZEICHNEN stimmen– hier ist „Ä“ nahe liegend.

Haben Sie alle Bilder vektorisiert, importieren Sie einfach noch den Sound, wenn gewünscht, und fertig ist Ihr Video!

Abbildung 3.14: Angelegte Shortcuts werden im Dialog angezeigt





Teil II: Anwendungen

Basic Elements

4.1 Maskeneffekte

Grundlagen

Masken bieten Ihnen die Möglichkeit dynamisch nur Ausschnitte eines Objektes anzuzeigen. Hierfür wird eine eigene Ebene auf der Zeitleiste in der Flash-Entwicklungsumgebung angelegt. Die Maske selbst definiert den sichtbaren Bereich, also das, was von der Ebene, auf die diese Maske angewendet wird, sichtbar sein soll. Neu in Flash MX ist erstmals, dass Flash nun auch eine ActionScript-Steuerung dieser Masken anbietet. Zuerst möchte ich auf den einfachen Fall einer Ebenenmaske eingehen. Folgende Grafik veranschaulicht Ihnen noch einmal die Funktionsweise von Masken generell, denn nicht nur in Flash, sondern auch in anderen Programmen werden Masken so verstanden.

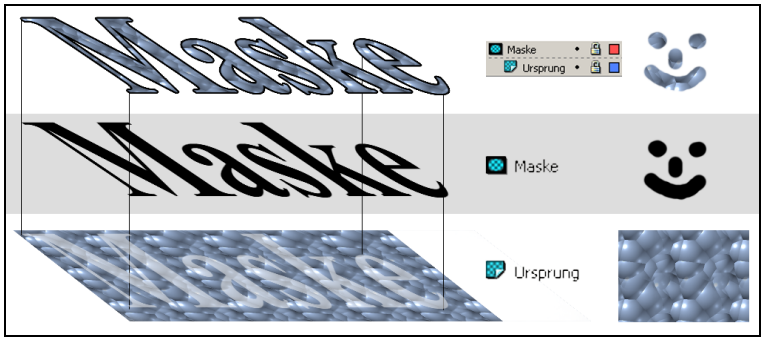


Abbildung 4.1: Hier ein kurzer Überblick über die Struktur von angewendeten Masken

Im Gegensatz zu Adobe Photoshop z.B. besteht in Flash keine Möglichkeit Verläufe durch Masken darzustellen. Hier gibt es allerdings noch Tricks, die zum gewünschten Ergebnis führen, auf die zu einem späteren Zeitpunkt in diesem Kapitel eingegangen wird.

Texteinblendungen

Möchten Sie einen Schriftzug einblenden, bei dem sich die Buchstaben „auffüllen“, bietet sich die Verwendung einer einfachen Maske an. Anhand dieses Beispiels werden Sie schnell erkennen, wie Masken funktionieren.

Der Text „Texteinblendung“ wird zuerst auf einer Extra-Ebene angelegt und als Maske definiert. Um eine Ebene zu einer Ebenenmaske zu machen wählen Sie einfach die Ebene per Cursor und führen einen Rechtsklick aus. Folgendes Menü erscheint, aus diesem wählen Sie die Maske.

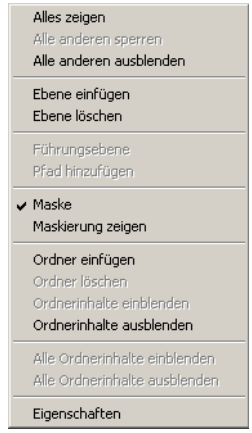


Abbildung 4.2: Hier dargestellt ist das Menü, nachdem die Ebene als Maske deklariert wurde.

Wenn Sie den Film nun starten, werden Sie feststellen, dass kein Schriftzug lesbar ist; dies ist einfach die Konsequenz aus dem Fehlen der zu maskierenden Ebene. Zeichnen Sie beliebige Objekte in die Ursprungsebene, so wird nur die Schnittmenge beider angezeigt, d.h. die Stellen, an denen sich Maske und Ursprung überschneiden.

Da Sie mit Flash MX allerdings noch mehr Möglichkeiten haben, möchte ich dieses Beispiel noch etwas weiter führen, der Text soll so aussehen, als liefen die Buchstaben mit Wasser voll. Hierfür muss auf Tweenings zurückgegriffen werden. Sollten Sie mit diesen nicht vertraut sein, so können Sie entweder die Einführung in Tweenings in diesem Kapitel vorziehen oder dieses Beispiel weiterführen, sobald Sie dort angelangt sind.

Erstellen Sie zunächst mittels eines Tweenings einen stehenden Wellengang. Ein Ergebnis könnte wie folgt aussehen.

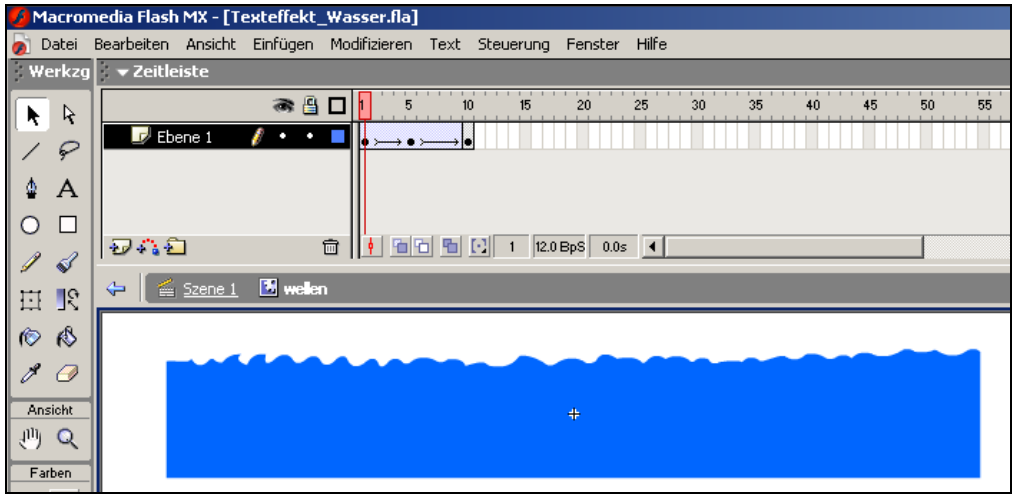


Abbildung 4.3: MovieClip „wellen“.

Den gerade eben angelegten MovieClip legen Sie in die Ursprungsebene auf der Hauptzeitleiste. Dort platzieren Sie den MovieClip unter dem Schriftzug so, dass diese keine Deckung haben. Ein weiteres Tweening der Wellen vermittelt später den Eindruck des steigenden Wassers. Setzen Sie in das letzte Schlüsselbild des Tweenings ein `stop();`, so dass der Text gefüllt stehen bleibt.

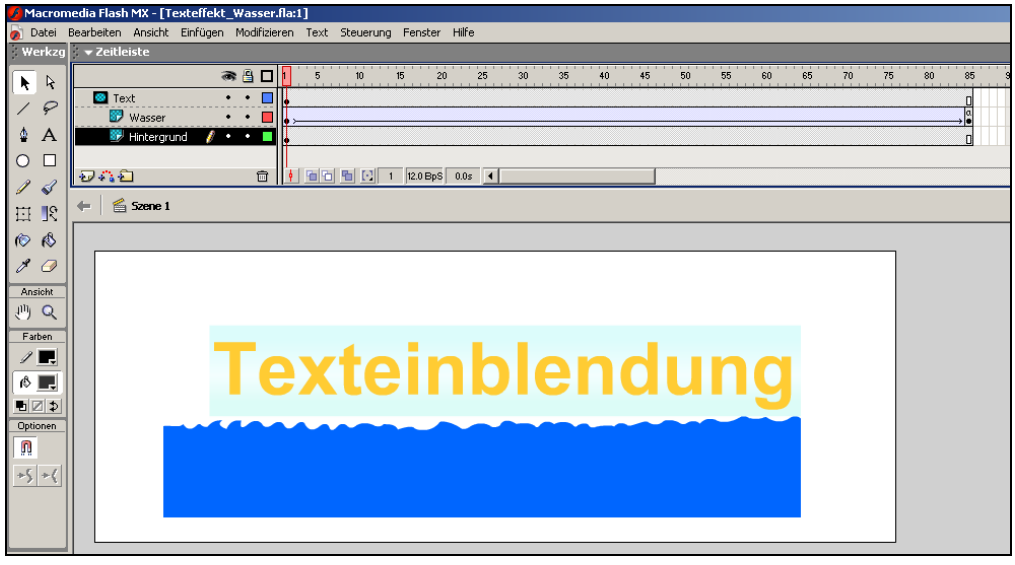


Abbildung 4.4: So sieht eine mögliche Anordnung auf der Hauptzeitleiste aus.

Der Texteffekt erzeugt folgende Animation:

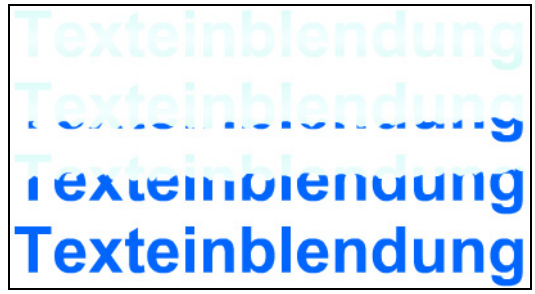


Abbildung 4.5: Abgebildet im Zeitverlauf von oben nach unten.

Alphawerte und Masken

Im Folgenden wird nun erläutert, wie man den Effekt von „weichen Kanten“ erzeugen kann. Wie schon kurz erläutert, können Masken alleine dies nicht leisten, jedoch hat Flash die Möglichkeit PNG-Dateien zu importieren. Dieses Grafikformat speichert neben dem Farbwert pro Pixel auch den so genannten Alpha-Wert, also den Wert, der besagt, inwiefern dieser Pixel darunter liegende überdeckt. Gültige Werte, die als Eigenschaft angegeben werden, können zwischen 0 (vollständig transparent) und 100 (vollständig deckend) liegen. Diese Alphawerte haben allerdings keinen Einfluss auf Masken, hier ist das Shape allein entscheidend. Folgendes Beispiel kombiniert die Möglichkeiten einer Maske mit den Eigenschaften einer PNG-Grafik. Das Ergebnis soll so aussehen:

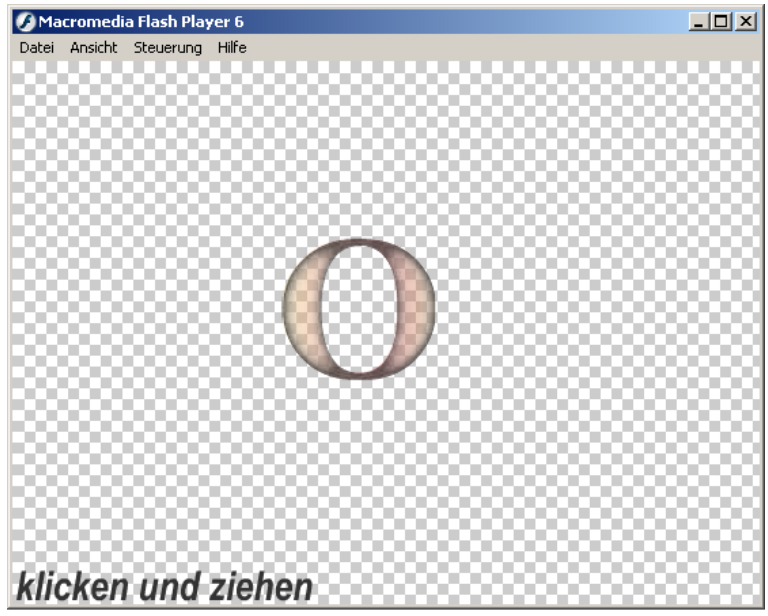


Abbildung 4.6: Maskeneffekt mit weichen Kanten

Wie Sie bei der Abbildung schon erkennen, sind die Randbereiche der Form „o“ nicht mehr vollständig transparent, so dass der Einruck von Dreidimensionalität entstehen kann. Ähnlich einem Glas sind die Randbereiche weniger durchlässig, was in Zusammenhang mit den Brechungswinkeln im Glas steht, aber lassen Sie uns vorne anfangen.

Zuerst muss eine entsprechende Grafik erstellt werden, dies werde ich kurz an einem Beispiel in Photoshop 6.0® erklären. Erstellen Sie zunächst dort ein neues Dokument in entsprechender Größe mit transparentem Hintergrund. Legen Sie auf einer neuen Ebene die gewünschte Grafik an ohne zu versuchen diese Ränder selbst zu zeichnen. In diesem Beispiel wurde einfach der Buchstabe „o“ in eine Ebene gerastert. Erzeugen Sie aus diesem eine Ebene, in der genau diese Grafik ausgespart wird. Dies geht verhältnismäßig einfach, indem Sie eine neue Ebene mit einer beliebigen Farbe füllen und die Ebene, in der die gewünschte Grafik liegt, mit gedrückter Strg-Taste auswählen. Sie haben nun eine Auswahl des Umrisses der Grafik erzeugt. Wählen Sie jetzt direkt wieder die gefüllte Ebene und drücken Sie die Entf-Taste, hiermit schneiden Sie das Shape aus. Wenden Sie dann für diese Ebene einen Ebeneneffekt an, in diesem Beispiel „Schein nach außen“, Farbe Schwarz, Größe 12px, Überfüllen 20%. Wiederholen Sie nun den ersten Auswahlschritt und deaktivieren Sie auch die Sichtbarkeit der Vorlage-Ebene. Drücken Sie jetzt Strg+⇧+C um aus allen Ebene zu kopieren. Hier wird nun auch der Effekt mitkopiert. Legen Sie ein neues leeres Dokument an und kopieren Sie das, was Sie aus dem zuerst angelegten Dokument kopiert haben, hier mit Strg+V hinein. Exportieren Sie diese Grafik jetzt als PNG-Datei unter Hilfe › Transparentes Bild exportieren. Haben Sie dies erfolgreich ausgeführt, können Sie die eben erzeugte Datei in Flash importieren (Strg+R). Das Ergebnis könnte wie folgt aussehen:

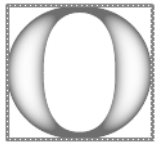


Abbildung 4.7:
Die in Flash importierte Grafik

Legen Sie nun die auf die Grafik zugeschnittene Maske an und wenden Sie diese auf einen beliebigen Hintergrund an. Platzieren Sie die Grafik passend eine Ebene über die Maskenebene, so dass Maske und Grafik deckungsgleich sind. Diese Anordnung in einem MovieClip sieht dann wie folgt aus:

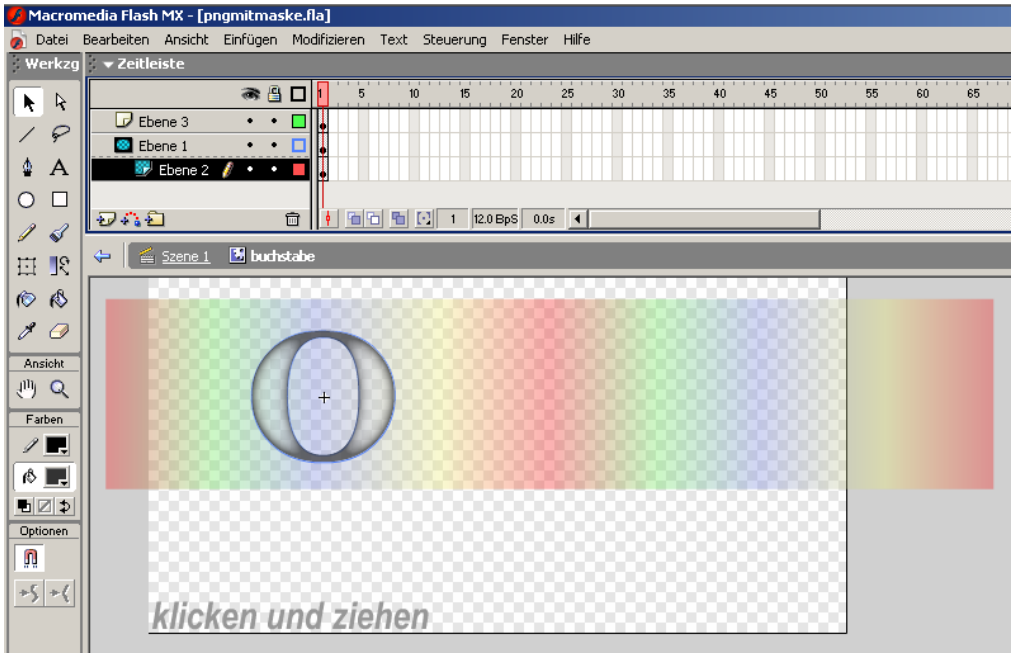


Abbildung 4.8: Eine Anordnung von Maske, Hintergrund und Grafik

Nun wenden Sie auf den soeben erzeugten MovieClip noch ein ActionScript an, das es Ihnen ermöglicht, es vor dem Hintergrund zu bewegen, auf diese Weise wird der transparente Charakter des Objektes besonders hervorgehoben.

```
onClipEvent(mouseDown){
    this.startDrag();
}
onClipEvent(mouseUp){
    this.stopDrag();
}
```

Anstelle dieses „Regenbogen"-Tweenings können dort beliebige Inhalte eingefügt werden.

Masken mit ActionScript

Masken können in Flash MX nun auch dynamisch erzeugt werden, bzw. vorhandene Objekte (MovieClips) anderen, bestehenden Objekten als Maske zugewiesen werden. Dies wird mittels des Befehls `.setMask()` gemacht. Vor dem Punkt sollte das zu maskierende Objekt syntaktisch korrekt referenziert sein, in der Klammer wird als Parameter die Referenzierung auf das Maskenobjekt erwartet.

4.2 Textfelder

Will man in Flash einen Text einbinden, ist man auf die Nutzung von Textfeldern angewiesen, vorausgesetzt man fügt diesen Text nicht als Grafik ein. Im folgenden Abschnitt erläutern wir die verschiedenen Arten von Textfeldern und ihre spezifischen Eigenschaften.

Erzeugen Sie Text möglichst mit Hilfe der Textfelder. Texteffekte können Sie auch mit Hilfe von Flash erzeugen. Importierte Grafiken mit Text sind auf jeden Fall immer größer als ein von Flash erzeugtes Textfeld und auch meistens nicht so klar zu lesen.

Textfelder werden für jegliche Art von Schrift benötigt. Wählen Sie das Textwerkzeug aus, um auf der Bühne ein Textfeld anzulegen.

Abbildung 4.9: Textwerkzeug

Legen Sie ein Textfeld an, indem Sie mit gedrückter Maustaste auf die Bühne klicken und ein Feld entsprechender Breite aufziehen. Die Breite kann im Nachhinein geändert werden. Wenn der Text über die maximale Breite des Textfeldes hinausgeht, wird der eingegebene Text nicht vollständig angezeigt. Mit einem einfachen Klick mit der linken Maustaste erstellt Flash ein Textfeld, das sich der Texteingabe automatisch anpasst.

Abbildung 4.10: Statisches Textfeld mit fester Breite

Unterschieden werden diese beiden Arten von Textfeldern durch das kleine Symbol auf der rechten Seite: Ein Rechteck am oberen Rand kennzeichnet ein statisches Feld mit einer vorgegebenen Breite, ein Kreis am unteren Rand hingegen steht für ein Textfeld mit variabler Breite.

Durch einen Doppelklick auf dieses Symbol am rechten Rand des Textfeldes können Sie von einer Variante zur anderen wechseln.

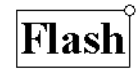
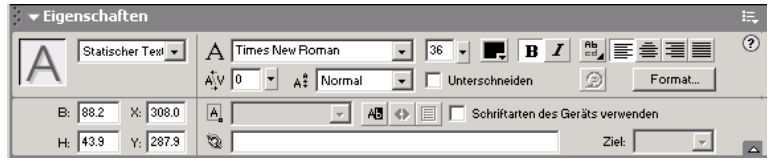


Abbildung 4.11: Dynamisches Textfeld, dessen Breite sich dem eingegebenen Text anpasst.

Durch Klicken mit der rechten Maustaste auf das Textfeld und die Auswahl des Menüpunktes Eigenschaften kann man die Schriftgröße, Schriftart, Schriftfarbe und den Zeichenabstand ändern. Zudem kann man dort auch die Optionen fett, kursiv, unterstrichen, hochgestellt oder tiefgestellt einstellen.

Abbildung 4.12: Aufruf des Eigenschaftsfensters für ein Textfeld

Abbildung 4.13: Eigenschaftsfenster für ein Textfeld

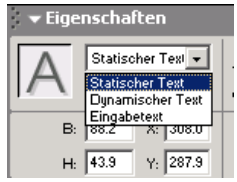


Im unteren mittigen Feld geben Sie die Ziel-URL an, mit der eine markierte Textpassage verlinkt werden soll. Mit der Option Unterschneiden ist gewährleistet, dass der Zeichenabstand zwischen bestimmten Buchstabenkombinationen, z.B. „A“ und „w“, verringert wird, um eine bessere optische Wirkung zu erzielen.

In dem Fenster Absatz können Sie den Text linksbündig, rechtsbündig, in Blocksatz oder zentriert ausrichten. Über die darunter liegenden Felder ist es möglich, einen Einzug vom linken bzw. rechten Rand oder einen Erstzeileneinzug einzustellen. Im Feld rechts unten können Sie den Zeilenabstand verändern.

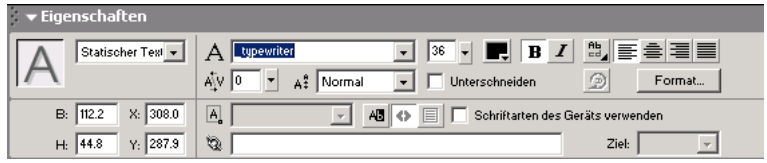
Im linken oberen Fenster legen Sie fest, ob es sich um einen statischen oder dynamischen Text oder um eine Texteingabe handelt.

Abbildung 4.14: Auswahl der Art des Textfeldes



Statischer Text

Abbildung 4.15: Eigenschaftsfenster bei statischem Text



Statischer Text wird als Standardtext gebraucht. Wenn man einen Text eingeben möchte, der sich während des gesamten Films nicht ändern soll, wählt man diese Option. Dieses Textfeld kann man mit Tweenings bewegen und skalieren, den Inhalt aber nicht mehr ändern. Wenn Sie eine andere Schriftart wählen, die nicht zu den Flash-Geräteschriftarten gehört (_serif, _sans, _typewriter), bettet Flash die jeweilige Schriftart ein und speichert diese mit in dem SWF. Dies ist immer sinnvoll, wenn Sie eine ungewöhnliche Schriftart wählen, da Sie nicht voraussetzen können, dass jeder Besucher diese Schrift installiert hat. Das Einbetten der Schriftart erhöht jedoch die Dateigröße. Um zu unterbinden, dass Flash Schriften einbettet, aktivieren Sie das Kästchen „Schriftarten des

Geräts verwenden“. Falls Sie dieses Kästchen aktiviert haben und der Besucher die ausgewählte Schriftart nicht installiert hat, wählt Flash eine auf dem Gerät des Anwenders installierte Schrift aus, die jener Schriftart am nächsten kommt. Ist jedoch eine vergleichbare Schrift nicht vorhanden, so ist nicht auszuschließen, dass dies zu einem nicht lesbaren Ergebnis führt. Die standardmäßig installierten Schriftarten sind bei kleineren Schriftgrößen unter Umständen schärfer bzw. besser lesbar und sollten deshalb bevorzugt verwendet werden.

Durch die Aktivierung des Kästchens Auswahl ermöglichen Sie es dem Besucher, die jeweilige Textpassage in die Zwischenablage zu kopieren und in anderen Programmen weiter zu nutzen.

Dynamischer Text

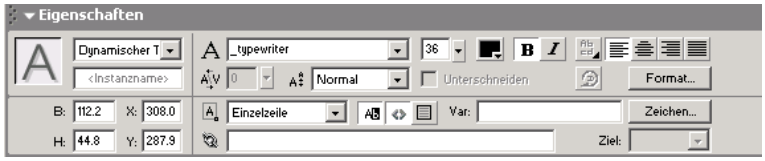


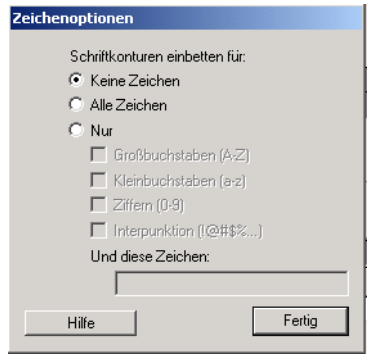
Abbildung 4.16: Eigenschaftsfenster eines dynamischen Textfeldes

Dynamischer Text wird dazu benötigt, um Variablen auf dem Bildschirm auszugeben, die als Ausgabeboxen eingesetzt werden können. Es kann eingestellt werden, dass ein Rahmen gezeichnet wird und ob der Text ein- oder mehrzeilig dargestellt werden soll. Legen Sie bei leeren Textfeldern immer einen Rahmen an, damit sie als solche für den Besucher auch erkennbar sind. Die Option „Auswählbar“ hat genau die gleiche Funktion wie zuvor beim statischen Text: Sie ermöglicht es dem Besucher, Text in die Zwischenablage zu kopieren. In dem Eingabefeld „Var“ geben Sie den Variablennamen an, der dem Textfeld zugewiesen werden soll. Wenn Sie dort *Textfeld1* eingeben und dieser Variable anschließend im Verlauf des Films den Wert 1 zuweisen, erscheint eine „1“ in diesem Textfeld. Sie können natürlich auch längere Texte eingeben wie z.B. `Textfeld1 = "Dieses hier ist das erste Textfeld";` , dabei sollten Sie darauf achten, dass der komplette Text in das Ausgabefeld passt.

Wenn Sie das HTML-Kästchen aktivieren, erwartet Flash einen HTML-Text. Diesen können Sie aus einer Variablen, die Sie zuvor schon gesetzt haben, oder aus einer externen Datei einlesen.

Abbildung 4.17: HTML-Option

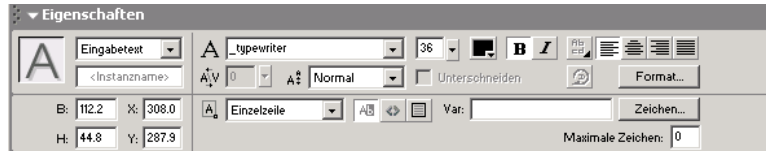
Abbildung 4.18: Textkonturenfenster



Mit dem Button Zeichen können Sie die komplette Schriftart einbetten oder nur Teilstücke, Großbuchstaben, Kleinbuchstaben, Zahlen oder Sonderzeichen zulassen. In der darunter liegenden Eingabebox können Sie die eingebetteten Zeichen selbst angeben. Die Zeichen werden durch Komma getrennt eingegeben.

Eingabetext

Abbildung 4.19: Eigenschaften eines Eingabetextfeldes



Der Eingabetext wird dazu benötigt, um dem Besucher die Möglichkeit zu geben, einen Text oder einen Wert an Flash zu übermitteln.

Die Optionen sind vergleichbar mit denjenigen beim dynamischen Text – bis auf die neue Eingabebox „Maximale Zeichen“: Dort können Sie die maximale Anzahl der Zeichen eingeben, die das Textfeld enthalten kann. Überzählige Zeichen werden dann nicht angenommen.

Neu in Flash MX ist abgesehen von der anderen Oberfläche auch, dass man Textfelder Instanznamen zuweisen kann. Dafür dient jeweils das zweite Feld von links oben gesehen. Wenn Sie einem Textfeld einen Instanznamen zuweisen, können Sie dieses mit ActionScript direkt ansprechen.

Tabulatoren und Textfelder

Wenn man ein Eingabeformular, bestehend aus mehreren Textfeldern, erstellt, ist es wünschenswert, dass der Besucher mit Hilfe der Tabulatortaste zwischen den Textfeldern wechseln kann.

Zu diesem Zweck besitzen Textfelder eine Menge an Eigenschaften, die mit der Auswahl zu tun haben.

Flash generiert von Anfang an eine vordefinierte Tabulatorreihenfolge abhängig von den Objekten auf der Bühne. Diese richtet sich nach den x- und y-Koordinaten der Objekte.

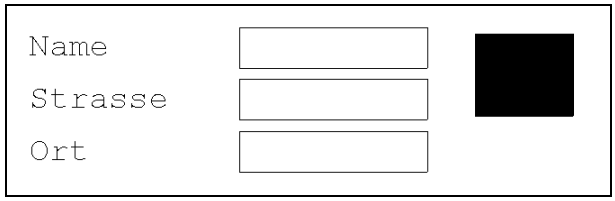


Abbildung 4.20: Aufbau des Beispiels

Wenn Sie sich das Beispiel einmal anschauen, würde man durch Drücken der Tabulatortaste zuerst das oberste Textfeld auswählen, als Zweites den Button (dies sollte eigentlich nicht geschehen).

Um dies zu verhindern bietet Flash die Eigenschaft `tabEnabled`. Dieser werden bei dem Button deaktivieren und somit diesen aus der automatisch generierten Tabulatorreihenfolge entfernen. Der Button hat den Instanznamen *b1*.

Die Fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel4/02_tab.fla*.

```
b1.tabEnabled = false;
```

Listing 4.1: Deaktiviert die automatische Tabulatorreihenfolge für den Button

Mit diesem Befehl würde die Tabulatorreihenfolge für dieses Beispiel zufrieden stellend funktionieren. Wenn man allerdings das Beispiel auch nur leicht erweitert, stößt man wieder auf ein Problem.

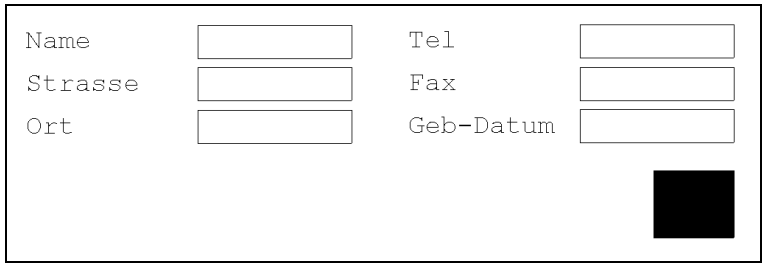


Abbildung 4.21: Erweitertes Layout des Beispiels

Flash erstellt für diese Textfelder eine sehr unschöne Tabulatorreihenfolge. Falls Sie uns dies nicht glauben, können Sie sich gerne selbst einmal davon überzeugen.

Die Fla-Datei zu dem zweiten noch nicht funktionierenden Beispiel finden Sie auf der CD unter *Kapitel4/03_tab2a.fla*.

Da Flash diese Tabulatorreihenfolge nicht schön setzt, muss man sich nun selbst darum kümmern.

Zum einen gibt es dafür den Befehl

```
Selection.setFocus(t1);
```

Wir haben den Textfeldern die Instanznamen t1-t6 zugewiesen. Wenn kein Instanzname zugewiesen ist, kann man die Textfelder über die zugewiesenen Variablennamen ansprechen.

Der obere Befehl würde also den Fokus auf Textfeld 1 richten. Wenn man nun die Tabulator-Taste drückt, soll der Fokus von Textfeld 1 auf Textfeld 2 versetzt werden. Dies würde der folgende Befehl bewerkstelligen.

```
Selection.setFocus(t2) ;
```

Dies soll natürlich möglichst automatisch geschehen.

In Flash 5 musste man noch mühsam sich selbst einen „Tabulatorbaum“ bilden und sich da durchhangeln. Flash MX hingegen bietet die Eigenschaft `tabIndex`, wodurch man einfach eine neue Tabulatorreihenfolge erstellen kann.

Listing 4.2: Erstellung einer selbst definierten Tabulatorreihenfolge

```
TF = new Array("t1", "t2", "t3","t4", "t5", "t6");
for(i=0; i<TF.length;i++){
    this[TF[i]].tabIndex=i+1;
}
```

Eigentlich spricht das Beispiel schon fast für sich selber, dennoch ein paar Worte dazu. Zuerst wird ein Array erstellt, welches die Instanznamen der Textfelder beinhaltet.

Die for-Schleife durchläuft dieses Array und weist jedem Textfeld die Eigenschaft `tabIndex` zu. Zu Beginn ist diese Eigenschaft standardmäßig nicht definiert. Das erste Textfeld erhält den Wert 1, das zweite 2 usw. Flash MX erledigt dann automatisch den Rest. Sogar die Tastenkombination ⇧+⇆ funktioniert direkt ohne unser Zutun!

Wer das doch etwas umständlichere Script aus Flash 5 dagegen kennt, wird dieses neue Feature schnell zu schätzen wissen und nicht mehr missen wollen.

Die Fla-Datei zu dem zweiten funktionierenden Beispiel finden Sie auf der CD unter *Kapitel4/03_tab2b.fla*.

Dynamisches Kreieren von Textfeldern

Das dynamische Kreieren und Steuern von Textfeldern bietet Ihnen eine Vielzahl von Möglichkeiten, eine von ihnen sind Texteffekte. Rekapitulieren wir kurz noch mal die unterschiedlichen Arten der Textfelder.

Unterschieden wird in drei Arten:

- statische Textfelder
- dynamische Textfelder
- Eingabetextfelder

Statische Textfelder sind, wie der Name schon verrät, statisch und ihre Inhalte können im späteren Programmablauf nicht mehr verändert werden. Hierin unterscheidet es sich von den beiden anderen Arten und bie-

tet sich eigentlich für Programmier- bzw. ActionScripting-Zwecke nicht an. In der Flash-Oberfläche erkennen Sie ein statisches Textfeld einfach an der Anfass-Ecke, die sich hier oben rechts im Textrahmen befindet.

Abbildung 4.22: Statisches Textfeld in der Entwicklungsumgebung

Im Gegensatz zum statischen Textfeld kann man den Text eines dynamischen Textfeldes nachträglich bzw. zur Laufzeit verändern. Dynamische Textfelder sehen in der Flashumgebung wie folgt aus:

Abbildung 4.23: Dynamisches Textfeld in der Entwicklungsumgebung

Abbildung 4.24: Eingabetextfeld in der Entwicklungsumgebung

Um die Texte steuern zu können wird in der sechsten Flash-Version, Flash MX, eine neue Option angeboten, die es erlaubt, Textfelder wie normale Filminstanzen zu steuern, doch da diese nicht mit der Funktion `duplicateMovieClip` funktioniert, muss im Zweifelsfall doch wieder darauf zurückgegriffen werden, die Textfelder in MovieClips zu kopieren.

In dem folgenden Beispiel wurde aus genau diesem Grund das Textfeld in einen MovieClip kopiert. Der MovieClip hat den Instanznamen *letter*, das Textfeld den Instanznamen *character* und der Variablenname ist *wert*. Um diese Namensgebung noch mal kurz zu rekapitulieren:

Die Textinstanzen können mit duplicateMovieClip nicht kopiert werden.

- „letter“ → MovieClip, in dem sich unser Textfeld befindet, wird gebraucht um mehrere Instanzen des Textes bzw. der Buchstaben für den Effekt zu erzeugen.
- „character“ → Instanzname des Textfeldes, liefert uns später wichtige Eigenschaften des Textfeldes zurück um den Buchstaben zu platzieren.
- „wert“ → Zugriffsvariable, setzt man sie auf einen Wert, erscheint dieser als Text in dem Feld.

Das Ganze wird in der Entwicklungsumgebung wie folgt angezeigt:

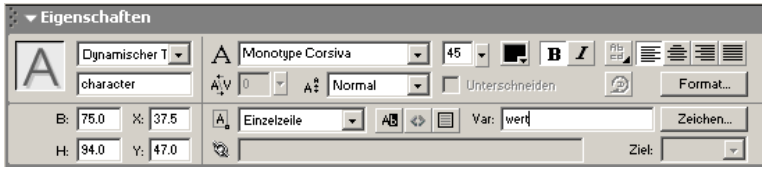


Abbildung 4.25: Angaben im Eigenschaftsfenster für Textfelder

Flash ist stellenweise recht pfiffig programmiert, was einen Entwickler oft zwingt in die Trickkiste zu greifen um einen gewünschten Effekt zu erzeugen. Ebenso hier, wenn Sie den Film auf Ihrem System durchtesten und verschiedene Werte in das Textfeld eintragen, werden Sie kaum Probleme haben, diese korrekt angezeigt zu bekommen, doch lassen Sie sich nicht täuschen, intern passiert hier mehr, als man vermutet. Flash optimiert standardmäßig alle Buchstaben eines Schriftsatzes so, dass nur die notwendigen mit in das spätere .swf-File übernommen werden. Das heißt im Klartext für Sie: Wenn die Schrift nicht im .swf liegt, wird die nächstliegende aus dem System verwendet. Da Sie auf dem Entwicklungssystem prüfen, werden Sie den Unterschied nicht bemerken. Klicken Sie um dieses zu vermeiden auf „Zeichen“ und aktivieren Sie die Option Alle Zeichen im Dialogfenster.

Abbildung 4.26: Alle Zeichen sollen für das Textfeld mit in den .swf übernommen werden.

Da wir nun das Textfeld recht variabel und komfortabel gestaltet haben, können wir uns an den eigentlichen Texteffekt machen, der Ihnen das dynamische Kreieren und Steuern von Textfeldern veranschaulichen soll. Folgende Grundüberlegung stellen wir zunächst an:

- Alle Buchstaben werden einzeln animiert
- Jeder Buchstabe folgt dem gleichen Lauf
- Der Text wird erst ausgeblendet, wenn der Schriftzug komplett ist

Zunächst wird ein MovieClip mit dem Instanznamen „fontanim“ angelegt. Hier befindet sich der MovieClip „letter“, der animiert werden soll. An dieser Stelle steht es Ihnen frei, eine Animationsmöglichkeit zu wählen, hier ist jedoch ein Pfadtweening angebracht.

Das Tweening selbst hat drei Stellen, an denen ein ActionScript direkt in die Schlüsselbilder eingetragen werden muss, die sich auch aus den oben genannten Punkten ergeben.

- Der erste Frame muss einen Stop beinhalten, damit die Buchstaben nicht unkontrolliert einblenden.
- In der Mitte muss der Buchstabe warten und so lange anzeigen, bis er wieder ausblenden darf.
- Nach Ablauf der Animation muss der Film stoppen, die Instanz aus dem Speicher entfernt werden.

Die Animation könnte also wie folgt aussehen:

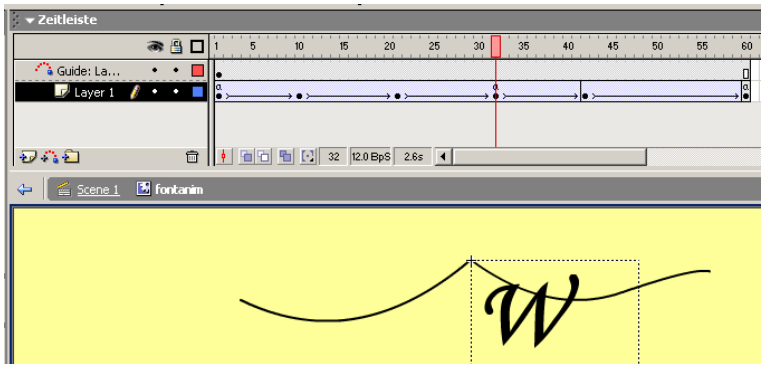


Abbildung 4.27: So kann die Animation u.U. aussehen.

Nun werden nur noch Steuerungsfunktionen gebraucht um den Ablauf zu kontrollieren und alles an der richtigen Stelle zu platzieren. Da der Texteffekt später einfach per Übergabe eines Textes und passender Zielkoordinaten funktionieren soll, muss die erste Funktion die Zeichenkette zuerst in ihre Bestandteile, die einzelnen Zeichen, auflösen und entsprechend die Instanzen kopieren.

Hier bietet es sich an, auf der Hauptbühne einen MovieClip anzulegen, der alle Funktionen beinhaltet. Aus einer Gewohnheit ist dieser mit *ctrl* benannt und enthält zunächst folgendes ActionScript:

```
onClipEvent(load){
    k=0;
    l=0;
    sText="";
    counter=0;
    counter_2=0;

    function startTextEffect(StartwertX, StartwertY, Abstand,
        spText){
        _root.stop();
        k=0;
        l=0;
        counter_2=0;
        sText=spText;
        nextspace=0;
        for (i=0;i<sText.length;i++){
            _root["letteranim"].duplicateMovieClip("font" +
                i, i);
            _root["font" + i].stop();
            _root["font" + i].letter.wert=
                 sText.substring(i,i+1);
            StartwertX += nextspace;
            nextspace = _root["font" +
                i].letter.character.textWidth * Abstand
            _root["font" + i]._x = StartwertX;
```

```
                _root["font" + i]._y = StartwertY;
            }
            counter=0;
        }
    }
```

Wundern Sie sich nicht über Variablen, die initialisiert werden, obwohl sie hier noch nicht gebraucht werden. Ruft man die Funktion `startTextEffect()` auf, z.B. so:

```
_root["ctrl"].startTextEffect(50,70,1.2,"Willkommen auf
    DieFlasher.de");
```

fängt die Funktion an, den Text „Willkommen auf DieFlasher.de“ zu zerlegen. Zuerst wird mit dem Befehl

```
_root["letteranim"].duplicateMovieClip("font" + i, i);
```

eine Kopie der `letteranim`-Instanz erstellt, der Name für den ersten Buchstaben ist somit *font0*. Dieser wird im Folgenden zuerst noch mal gestoppt, dann der Buchstabe ermittelt und in das Feld kopiert. „nextspace“ ist der Abstand zum nächsten Buchstaben, er ermittelt sich aus einer neuen Eigenschaft, die Flash MX zur Verfügung stellt, genannt `textWidth`. Nun ist auch klar, warum auch das Textfeld einen Instanznamen brauchte.

Folgend müssen für den kompletten Schriftzug die anderen Buchstaben geladen werden. Hierzu wird im *ctrl*-movie folgender Quellcode eingefügt.

```
onClipEvent(enterFrame){
    if (k<sText.length){
        _root["font" add k].play();
        k++;
    }

    if(counter==sText.length){
        if (l<sText.length){
            _root["font" add l].play();
            l++;
        }
    }
}
```

Die Variablen „k“ und „l“ sind mit o vorbelegt, so dass die erste Schleife alle Buchstabenanimationen hintereinander startet. Da die Animationen ja auf der Hälfte der Animation den Buchstaben anzeigen und dort stehen bleiben, wird in einer zweiten Schleife bei Erreichen der Zeichenlänge – also wenn der komplette Schriftzug geladen ist – das Ausblenden, der zweite Teil der Animation, gestartet.

Die Funktion um die Animation zu stoppen liegt, wie alle Funktionen, im *ctrl*-movie und wird durch `_root["ctrl"].stopme(this);` aufgerufen.

```
function stopme(obj){
    counter++;
    obj.stop();
}
```

Es ist an dieser Stelle zwingend erforderlich, die Anzahl der gestoppten Buchstaben zu zählen, da man es ansonsten ja nicht mitbekommt, wenn der fertige Schriftzug steht.

Im letzten Keyframe der Animation liegt der Funktionsaufruf `_root["ctrl"].finishme(this);` durch den das Beenden einer Buchstabenanimation an die Funktion gemeldet wird. Diese hat nun die Aufgabe die Instanz aus dem Speicher zu laden und zu überprüfen, ob es der letzte Buchstabe einer Zeichenkette war.

```
function finishme(obj){
    removeMovieClip(_root[obj._name]);
    counter_2++;
    if (counter_2==sText.length) {
        _root.play();
    }
}
```

Nachdem auch die letzte Instanz aus dem Speicher geladen wurde, wird mittels `_root.play();` das Feld geräumt und die nächste Textanimation kann gestartet werden.

4.3 Schaltflächen

Grundlagen

Eine Schaltfläche ist eines der drei grundlegenden Objekte in Flash. Wenn Sie Strg+F8 drücken, kommen Sie in ein Auswahlmenü, wo Sie entscheiden können, was Sie anlegen möchten. Zur Auswahl stehen

- MovieClip (oft auch als Filmsequenz oder kurz mc bezeichnet)
- Schaltfläche (langläufig auch als Button bezeichnet)
- Grafik

Natürlich können Sie auch erst Elemente platzieren, diese mit gedrückter ⇧-Taste markieren und dann nur F8 drücken. Hier wird Ihnen der gleiche Dialog angeboten, nur wird direkt aus der Auswahl ein entsprechendes Objekt erzeugt.

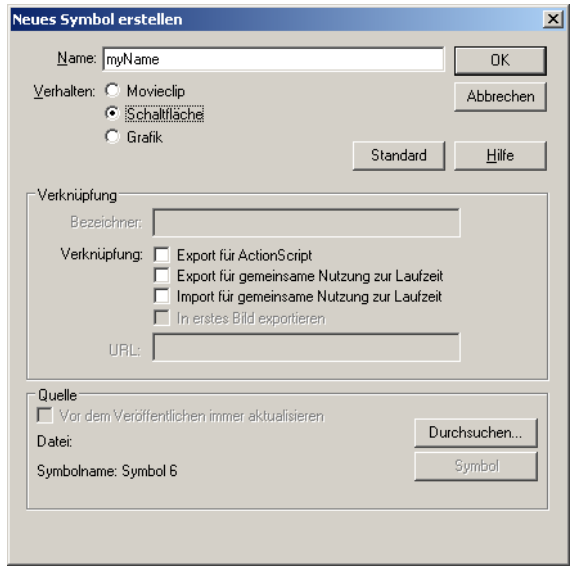


Abbildung 4.28: Der Dialog zur Erstellung eines Objektes

Während das ActionScript, das mit einem MovieClip verbunden ist, immer in einem `onClipEvent()` liegen muss, versteht das Schaltflächenobjekt diese Anweisung nicht, hier wird eine `on()`-Abfrage verlangt. In die Klammern ist der jeweilige zu überwachende Event einzutragen und aus Gründen der Vollständigkeit ist hier noch zu erwähnen, dass Grafiken kein ActionScript beinhalten können. Auch unterscheidet sich der Button in seiner doch recht restriktiven Vorgabe vom MovieClip. Zwar können Ebenen und Masken genutzt werden, aber die Anzahl der Frames ist auf die Darstellung der drei Zustände und einen aktiven Bereich beschränkt. Hier werden die Grafiken oder auch MovieClips eingefügt, die das Äußere der Schaltfläche bestimmen – auf diese Weise ist es auch möglich, animierte Schaltflächen zu erzeugen. Folgende vier Frames werden für die Schaltfläche benötigt:

- Normal
- Maus über der Schaltfläche (aktiver Bereich)
- Maus gedrückt über der Schaltfläche (aktiver Bereich)
- Aktiver Bereich

In der Entwicklungsumgebung sieht das wie folgt aus:

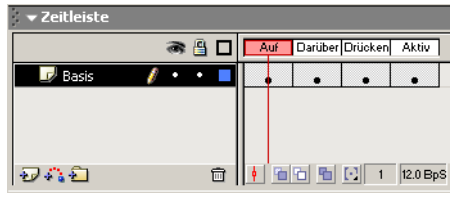


Abbildung 4.29: Die vier zu definierenden Frames

Nebenbei bemerkt: Falls Sie in der Eile z.B. einen Schriftzug zur Schaltfläche erklären und den aktiven Bereich nicht definieren, wird einfach das Element, welches zur Verfügung steht (in diesem Falle „Auf"), benutzt. Das mag zwar in der Testumgebung wunderbar funktionieren, hat aber einen Haken, der nicht auf Anhieb auffallen muss: Der aktive Bereich ist wirklich nur auf den Buchstaben, d.h. bei recht filigraner Schriftart kann es bei der Benutzung schwierig werden.

Überlassen Sie nichts dem Zufall, definieren Sie den aktiven Bereich selbst!

Blinde Schaltflächen

Als blinde Schaltflächen bezeichnet man Schaltflächen, die nur durch einen „aktiven Bereich" ausgezeichnet sind. Um diese zu erstellen muss man weiter nichts tun als die restlichen drei Zustände der Schaltfläche zu entfernen. Um in der Entwicklungsumgebung trotzdem Ihre Elemente wiederzufinden ist der aktive Bereich hier türkis und halbdurchlässig markiert, wie im Folgenden dargestellt.

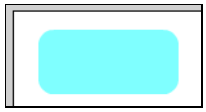


Abbildung 4.30:
Eine blinde Schaltfläche

Sie werden sich vielleicht selbst fragen, wozu man blinde Schaltflächen gebrauchen kann, aber das ist recht kurz erläutert. Möchte man z.B. gewisse Ereignisse überwachen, z.B. ob der Benutzer mit dem Cursor über einen gewissen Bildbereich fährt, so ist es das Einfachste diesen Bereich zu zeichnen. Doch jetzt müssen die Ereignisse auch überwacht werden, es wird auf das so genannte `onMouseOver` gewartet. Nun bieten uns Grafiken als Objekte keine Eventhandler, und die MovieClips können das `onMouseOver` leider nicht in dieser Form abfragen, übrig bleibt hier zwingendermaßen die Schaltfläche. Doch da es müßig wäre, eine fertige Grafik zu zerteilen und die einzelnen Schaltflächen wieder aneinander zu fügen, wählt man typischerweise eine blinde Schaltfläche. Folgendes Beispiel wird Ihnen das noch mal veranschaulichen.

▶ Hinweismeldungen am Cursor

Das oben schon angesprochene Beispiel finden Sie auch wie alle Beispiele auf der beigefügten CD. Zunächst importiert man eine beliebige Grafik; an dieser Stelle habe ich eine Aufnahme, die ich selbst gemacht habe, gewählt. Dann erstellen Sie blinde Schaltflächen in verschiedenen Formen und Größen, je nach Anforderung. Diese können Sie nun über das Bild verteilen, überall dort, wo später ein Hinweisfeld erscheinen soll. Das kann dann wie folgt aussehen:

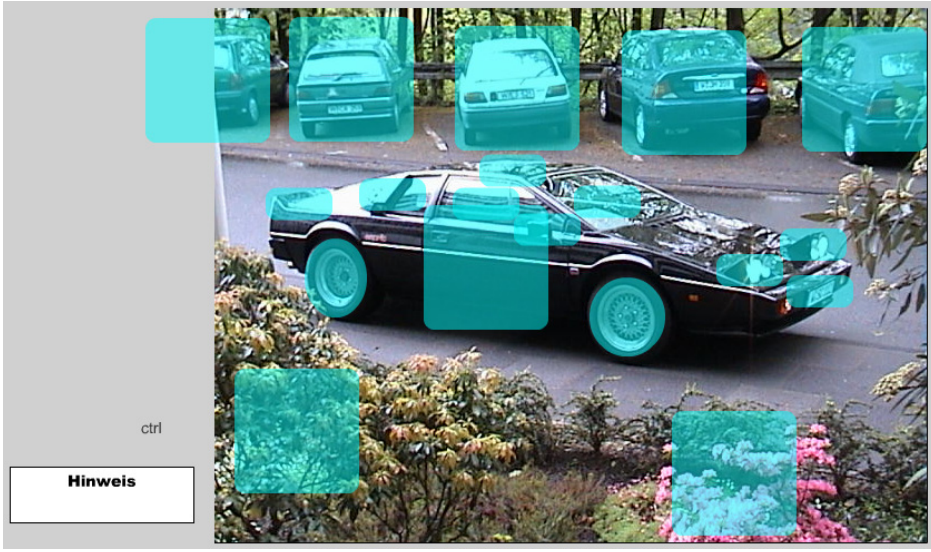


Abbildung 4.31: Platzierte blinde Schaltflächen auf der Grafik in der Entwicklungsumgebung

Wie Sie wahrscheinlich schon bemerkt haben, sind auf der Hauptbühne zwei von mir noch nicht erwähnte, aber notwendige Elemente platziert: zum einen ein MovieClip mit dem Hinweisfeld („hinweis“), zum anderen ein MovieClip „ctrl“.

Der einzige Frame der Hauptzeitleiste enthält ein kurzes ActionScript mit zwei Anweisungen.

```
_root["hinweis"]._visible=0;
stop();
```

Hier wird zum einen das Hinweisfeld ausgeblendet, zum anderen der Film (und damit die automatische Wiederholung) gestoppt.

Von diesem Stop nicht betroffen sind ja alle Instanzen, die in diesem Frame liegen, und somit wird trotzdem unser MovieClip *ctrl* fortlaufend wiederholend ausgeführt. Dieser enthält folgendes ActionScript.

```
onClipEvent(enterFrame){
    _root["hinweis"]._x=_root._xmouse;
    _root["hinweis"]._y=_root._ymouse;
}
```

Die Eigenschaft der Mauskoordinaten am besten immer mit vorangestelltem _root abfragen!

Dieses kurze Script bewirkt, dass der MovieClip *hinweis* immer dem Maus-Cursor folgt. Wichtig ist hier an der Stelle auch ein vorangestelltes `_root` bei der Eigenschaft der Cursor-Koordinaten, da diese sonst nicht definiert referenziert werden, was so viel heißt wie dass für die Koordinatenbestimmung kein festgelegter Nullpunkt vorhanden ist.

Das Textfeld im MovieClip heißt *text*, alle Zeichen sind eingebunden. Nun sind schon alle Vorbereitungen getroffen und den Schaltflächen kann ihr ActionScript zugewiesen werden. Doch bevor es an das Codieren geht,

machen wir eine kurze Pause und überlegen – was muss wann passieren? Folgendes lässt sich festhalten:

- im Moment des Eintritts in den aktiven Bereich soll der Text angezeigt werden
- der Hinweistext soll zu 30% durchlässig sein (optional)
- der Textinhalt soll dynamisch vergeben werden
- das Hinweisfeld soll bei Verlassen des aktiven Bereichs wieder ausgeblendet werden

Die beiden zu unterscheidenden Events heißen hier also `rollOver` und `rollOut`, passend hierzu sieht dann das ActionScript der entsprechenden Schaltfläche aus.

```
on(rollOver){
    _root["hinweis"].text = "Esprit S3 Fahrerseite ;-)";
    _root["hinweis"]._visible=1;
    _root["hinweis"]._alpha=70;
}

on(rollOut){
    _root["hinweis"]._visible=0;
    _root["hinweis"].text="";
}
```

Der Text ist hier in dem Beispiel natürlich frei wählbar, der MovieClip, der unsichtbar dem Cursor folgt, wird nun sichtbar geschaltet und mit einem „alpha“-Wert versehen. Hier ist er noch zu 70% sichtbar. Bei Verlassen der Schaltfläche wird der Text wieder gelöscht, das Hinweisfeld wieder ausgeblendet. Das Ergebnis sieht in diesem Fall so aus:

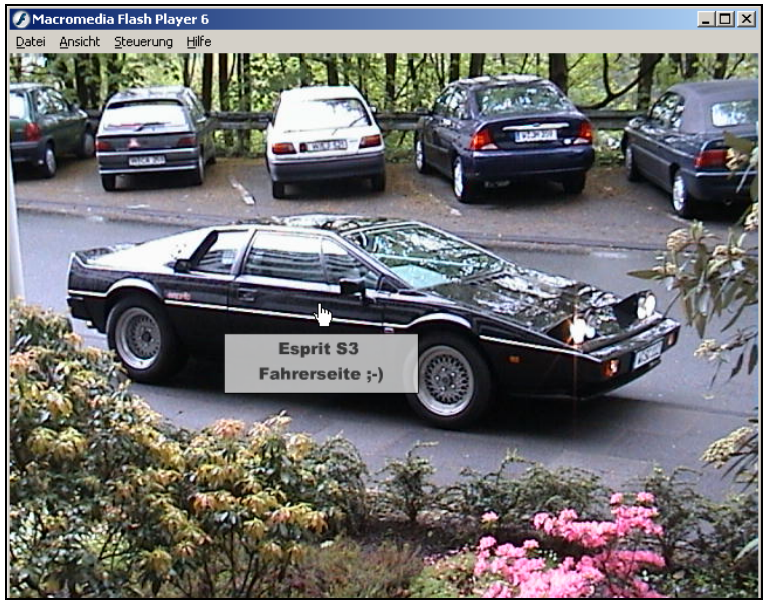


Abbildung 4.32: Das fertige Ergebnis, von den blinden Schaltflächen ist nun nichts mehr zu sehen.

Im nun folgenden Abschnitt lernen Sie, wie Sie das Beispiel noch um ein weiteres interessantes Feld erweitern. Grundsätzlich hat dieses Beispiel seinen Hauptanwendungsbereich bei Schaubildern, Produktpräsentationen oder auch interaktiven CD-ROMs.

Wollen Sie Schaltflächen etwas beleben, müssen Sie dort Filmsequenzen einbauen. Im Folgenden erfahren Sie, wie das geht.

Animierte Schaltflächen

Abbildung 4.33: Die Schaltfläche

Erstellt man in einem MovieClip eine einfache Animation (meistens in Anlehnung an die vorhandene Schaltfläche), so kann man diese in einen der Schaltflächenzustände hineinkopieren.

Das auch auf der CD vorhandene Beispiel zeigt eine recht unspektakuläre Schaltfläche, die beim Überfahren mit dem Cursor animiert wird.

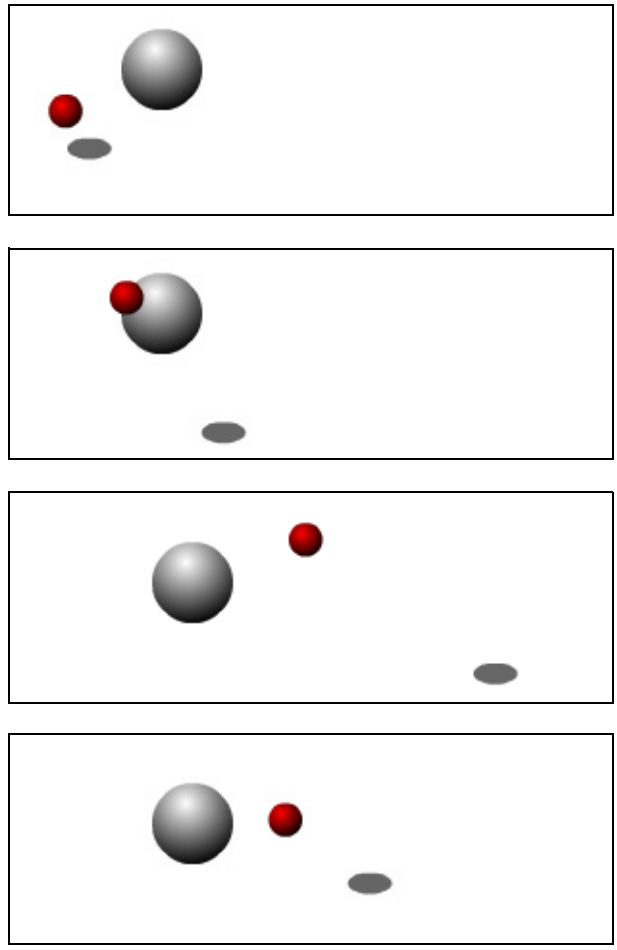


Abbildung 4.34: So sieht die fertige Animation aus.

Um diese Animation zu erstellen sind nur eine Handvoll Frames und Ebenen notwendig, was in der Flash-Umgebung so aussieht:

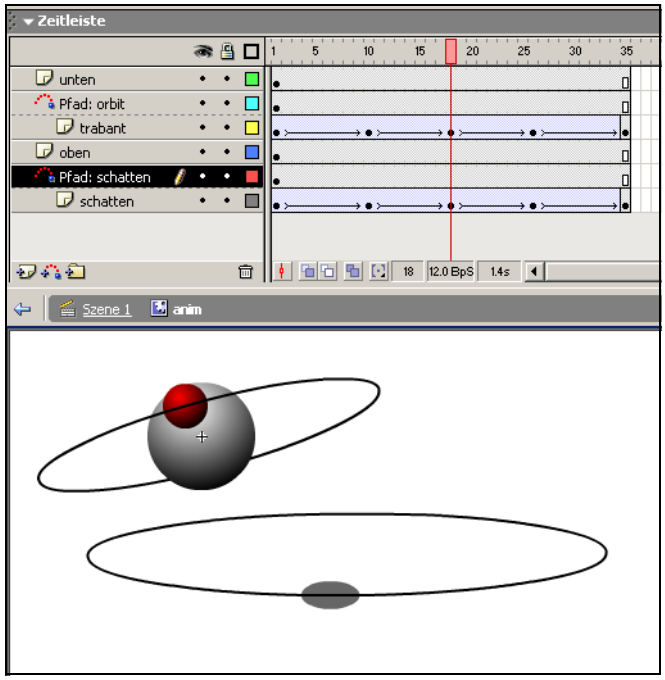


Abbildung 4.35: Die Animation im Detail

Wie Sie im Einzelnen das Tweening oder jegliche andere Animation gestalten, ist Ihnen überlassen; wie man Animationen dieser Art erstellt, finden Sie an anderer Stelle in diesem Buch.

Flash MX Buttons

Eine der Neuerungen in Flash MX ist, dass man den Schaltflächen Instanznamen geben kann und somit eine bessere Kontrolle über sie hat. Das entsprechende Dialogfenster sieht folgendermaßen aus:

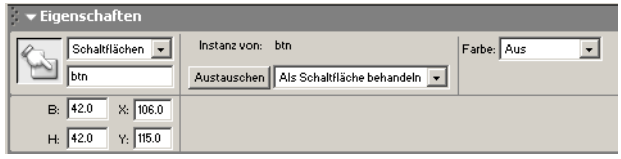


Abbildung 4.36: Eigenschaften-Panel in Flash MX

Die in der Abbildung ausgewählte Schaltfläche wurde mit dem Instanznamen *btn* (steht für Button) versehen. Nun lässt sich über diesen Instanznamen die Schaltfläche ansteuern. In einer Erweiterung zum obigen Beispiel habe ich einfach noch einen MovieClip mit dem Namen *ctrl*

eingefügt, der nichts anderes macht, als die Drehung des Buttons laufend zu ändern.

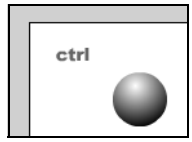


Abbildung 4.37: Control-MovieClip mit Schaltfläche.

In diesem MovieClip ist folgendes ActionScript enthalten:

```
onClipEvent(enterFrame){
    _root["btn"]._rotation+=4;
}
```

So kommt es, dass die Schaltfläche fortlaufend rotiert. Nicht zu vergessen ist ein ergänzendes `stop();` im Frame auf der Hauptzeitleiste. Das Endergebnis ist eine sich fortlaufend drehende Schaltfläche, aber dies soll Ihnen nur die Möglichkeiten, die Sie hier haben, veranschaulichen.

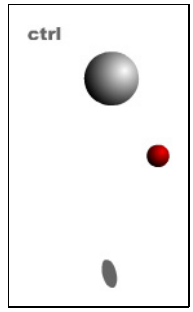


Abbildung 4.38: Das Ergebnis wird bei einem MouseOver besonders deutlich.

4.4 Preloader

Einen Preloader benötigen Sie, wenn Ihre Internetseite so groß ist, dass eine längere Ladezeit entsteht. Im Folgenden erfahren Sie, wie die Ladezeit abgefragt und dem Besucher angezeigt werden kann. Wir gehen hierbei von einer Website mit einer aufwändigen Navigation mit mehreren in Flash erstellten Untermenüs aus. Je nachdem, wie groß diese Untermenüs sind, sollte man verschiedene Preloadertechniken einsetzen. Im Folgenden zeigen wir, welche Technik sich in welchem Fall am besten eignet und worin die Vor- und Nachteile bestehen.

Flash-Seiten sollten so aufgebaut sein, dass sie keinen Preloader benötigen. Denn Flash wurde als *Streamingtool* konzipiert, so dass beim Öffnen der Seite sofort eine Bildschirmausgabe erfolgt und die restlichen Daten im Hintergrund eingeladen werden. Bei Seiten mit längeren Ladezeiten kann es jedoch passieren, dass der Besucher auf einen Button

drückt, der auf eine noch nicht eingeladene Szene verweist. Flash springt dann entweder zu einem bereits eingeladenen Frame, was zur Folge hat, dass die gesamte Seite nicht mehr richtig reagiert, oder der User klickt und es passiert gar nichts. Für diesen Fall gibt es die `ifFramesLoaded`-Abfrage.

Die Abfrage direkt in den Button einzubauen, ist eine Möglichkeit, aber weniger nützlich, weil die Aktion nur beim Klicken einmal ausgeführt wird. Dem Besucher wird zwar gezeigt, dass die Seite geladen wird, aber nicht, wann sie komplett eingeladen ist. Als Alternative bietet es sich an, einen Preloader einzubauen. Die folgenden Abschnitte zeigen Ihnen verschiedene Möglichkeiten auf.

Bevor Sie sich für einen Preloader entscheiden, sollten Sie sich über die Größe Ihrer geplanten Website im Klaren sein.

Für kleinere Internetseiten, in die hauptsächlich Flash-Grafiken und nur wenig oder gar kein Sound integriert wurden, reicht ein einfacher Preloader aus. Falls Sie eine etwas größere Seite planen, besteht die Möglichkeit diese in einzelne SWF zu unterteilen und jene jeweils auch mit einem Standard-Preloader zu bestücken.

Wenn man aber die Seite hingegen in einzelne SWFs unterteilt, werden die verschiedenen Dateien bzw. Untermenüs einzeln eingeladen. Und dies meistens erst, wenn der Besucher auf einen Unterpunkt klickt; so entsteht noch eine weitere störende Wartezeit für den Besucher. Um dem entgegenzuwirken werden wir Ihnen einen objektorientierten Preloader zeigen. Der Aufbau wird vielleicht zuerst etwas komplex erscheinen, dafür ist die Anwendung umso leichter.

Trotz alldem sollten Sie nicht vergessen, dass der Preloader nicht der wichtigste Bestandteil einer Seite ist. Das Layout und der Inhalt sollten immer die zwei Punkte sein, in die Sie am meisten Arbeit investieren!

Abfragemöglichkeiten

Flash MX bietet im Gegensatz zu Flash 5 noch einige interessante Möglichkeiten Files einzuladen.

Trotzdem wollen wir Ihnen die „älteren“ Flash 5-Befehle nicht vorenthalten. Denn sie bilden den Grundstein für einen einfachen Preloader.

▶ _totalframes-Eigenschaft

Die `totalframes`-Eigenschaft gibt aus, wie viele Frames das jeweilige SWF enthält. Dadurch erfährt man die Anzahl der gesamten Frames des Movies, in dem der Befehl ausgeführt wird.

```
if (_framesloaded >=_totalframes)
   {
    gotoAndPlay ("Scene 1", 1);
   }
```

Listing 4.3: Frame2: Sehr einfacher Preloader

Für die Benutzung von _totalframes im Zusammenhang mit Szenen ist jedoch noch eine Kleinigkeit wichtig. Flash setzt die Szenen einfach aneinander. Wenn man einen Film mit mehreren Szenen erstellt, in dem kein ActionScript enthalten ist, wird zuerst die erste Szene komplett abgespielt, dann die zweite usw.

Flash behandelt die Szenen wie einen großen Film, es unterteilt die Szenen also nicht richtig, sondern zeigt diese nur unterteilt und übersichtlicher an. Für Flash sind Szenen nichts anderes als hintereinander liegende Frames.

Wenn man die _totalframes-Eigenschaft auf einen Film mit mehreren Szenen anwendet, gibt Flash folglich nicht die Anzahl der Frames der jeweiligen Szene wieder, sondern die des ganzen Films (alle Frames in allen Szenen). Szenen sind hierbei nicht zu verwechseln mit eingebetteten Movies, diese sind nicht in der _totalframes-Abfrage beinhaltet, da es sich um eigenständige Filme handelt.

_totalframes gibt die gesamte Anzahl der Frames innerhalb eines SWFs an, egal ob diese durch Szenen getrennt sind oder nicht.

Für einen Preloader für mehrere Szenen kann man deshalb nicht die _totalframes-Abfrage verwenden. In diesem Fall würde man den ganzen Film preloaden und könnte nicht jede Szene einzeln abfragen.

▶ _framesloaded-Eigenschaft

Diesen Befehl kann man zum Beispiel auch dazu benutzen, um abzufragen, ob ein bestimmter Frame schon eingeladen wurde. Sollte dies zutreffen, kann man die davor liegenden Frames abspielen lassen, während im Hintergrund die restliche Hauptseite eingeladen wird.

Möglich ist auch, einen else-Befehl mit einzubinden bzw. eine Verschachtelung mit anderen Befehlen, z.B. im Zusammenhang mit der _totalframes-Eigenschaft.

Mit den Eigenschaften _frameloaded und _totalframes kann man auch eine prozentuale Darstellung erzeugen.

Listing 4.4: Prozentuale Berechnung des Ladestatus mit _framesloaded und _totalframes

```
ProzentAusgabe= int((_level0._framesloaded/
_level0._totalframes)*100);
balken.gotoAndStop(ProzentAusgabe);
if ( _root._framesloaded == _root._totalframes ) {
   _root.gotoAndPlay(2);
} else {
   _root.gotoAndStop(1);
}
```

Da bei der Berechnung der Prozentzahl die Anzahl der eingeladenen Frames zugrunde gelegt wird, funktioniert die Prozentanzeige leider nur dann genau, wenn alle Frames in etwa die gleiche Datenmenge beinhalten. Wenn beispielsweise der Film aus 100 Frames besteht und der 100ste Frame eine große Grafikdatei enthält, die vorher nicht vorkam, würde der Preloader sehr schnell anzeigen, dass die ersten 99% einge-

laden wären und dann lange bei 99% stehen bleiben, um diese Grafik einzuladen.

▶ getBytesLoaded() und getBytesTotal()

Da eine exakte prozentuale Ladeanzeige mittels `_framesloaded` und `_totalframes` nicht möglich ist, muss man für eine genaue Prozentangabe die bereits eingeladenen Bytes und die Gesamtdateigröße abfragen. Dies geht mit den Eigenschaften `getBytesLoaded()` und `getBytesTotal()`.

Die Berechnung erfolgt natürlich analog zum `_framesloaded` und `_totalframes`.

```
ProzentAusgabe =
       int((_root.getBytesLoaded()/_root.getBytesTotal())*100);
```

Im Folgenden haben wir dieses Script einmal zu einem brauchbaren Preloader erweitert.

Standard-Preloader

Man kann das Script in das erste Frame eines Movies einsetzen, die darauf folgenden Frames werden erst abgespielt, wenn alle eingeladen sind.

Ob man gerade diesen leicht erweiterten Preloader als Standard-Preloader ansieht, ist natürlich fraglich. Flash MX bietet mittlerweile so viele verschiedene Programmierarten etwas zu realisieren, dass es sehr viele Lösungen für ein Problem gibt. Trotzdem kann man den folgenden Preloader als Standard ansehen, da sich alle ähneln und dieser hier alle Grundelemente enthält.

Die Fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel4/o4_prelo.fla*.

Listing 4.5: Preloader eines beliebigen Films, der im ersten Frame platziert wird.

```
stop();
this.createEmptyMovieClip("preloader", 1);
//Ersatz für onClipEvent(load) von Preloader
with(this.preloader){
    rechteck = function (name, tiefe, farbe, x1, y1, x2, y2) {
        var referenz = createEmptyMovieClip(name, tiefe);
        referenz.beginFill(farbe, 100);
        referenz.moveTo(x1, y1);
        referenz.lineTo(x2, y1);
        referenz.lineTo(x2, y2);
        referenz.lineTo(x1, y2);
        referenz.endFill();
    };
    scrwidth = System.capabilities.screenResolutionX;
        //Bildschirmbreite
    scrheight = System.capabilities.screenResolutionY;
        //Bildschirmhoehe
```

```
    _x = scrwidth / 2;
    _y = scrheight / 2;
    stFormat = new TextFormat();
    stFormat.color = 0x000000;
    stFormat.font = "Arial";
    stFormat.size = 16;

    createTextField("T1", 2, 0 - 50, 0 - 8, 100, 16);
    T1.selectable = false;
    T1.autoSize = "center";
    T1.setTextFormat(stFormat);

    createTextField("T2", 3, 0 - 50, 0 + 8, 100, 16);
    T2.selectable = false;
    T2.autoSize = "center";
    T2.setTextFormat(stFormat);
}
//Ersatz für OnClipEvent(enterFrame) von preloader
this.preloader.onEnterFrame = function() {
    this.insgesamt = _root.getBytesTotal();
    this.geladen = _root.getBytesLoaded();
    this.prozent =
        Math.round(this.geladen / this.insgesamt * 100);
    rechteck("R1", 1, 0x00FFFF, 0 - 50, 0 - 8,
        this.prozent - 50, 16 - 8);
    this.T1.text = this.prozent + " %";
    this.T2.text = Math.round(this.geladen / 1024) + " kB von "
        + Math.round(this.insgesamt / 1024) + " kB eingeladen";
    if(this.prozent==100){
        this.unloadMovie();
        play();
    }
};
```

Zum einfachen Testen können Sie mit Strg+↵ (Steuerung › Film testen) den Film starten und danach den Streaming-Modus aktivieren. Dies geht durch weiteres Drücken von Strg+↵ oder Ansicht › Streaming-Modus. Für eine gute Ansicht sollten Sie unter Fehlersuche die simulierte Übertragungsrate auf 14,4 stellen.

Damit zuerst der Preloader eingeladen wird, muss das erste Frame des Movies leer sein. Das Movie wird anschließend direkt angehalten.

```
stop();
```

Es wird ein extra Movie erzeugt mit dem Namen *preloader* und der Tiefe 1. Dieses Movie hat den Vorteil, dass man darin alle Teile des Preloaders platziert und, wenn alles eingeladen ist, einfach das Movie wieder löscht und wirklich alles wieder entladen ist! Zudem braucht man auch ein Movie für ein `enterFrame`-Ereignis.

```
this.createEmptyMovieClip("preloader", 1);
```

Man hätte das Movie auch per Hand schon erstellen können und dann dem `onClipEvent(load)` folgende Zeile einfügen können. Da wir aber das Movie per ActionScript erzeugt haben, stellt der folgende Codeblock den `onClipEvent(load)`-Ersatz dar.

Mit Hilfe des with-Befehls werden alle enthaltenen Befehle ausgeführt, als wenn sie in dem MovieClip *preloader* stünden.

```
//Ersatz für onClipEvent(load) von Preloader
with(this.preloader){
```

Als Erstes fügen wir eine kleine selbst geschriebene Funktion ein, die ein Rechteck zeichnet. Damit diese Funktion überall zugänglich ist, wird sie mit Hilfe von `prototype` an `MovieClip` vererbt. Um ein Rechteck zu zeichnen benötigt die Funktion den Namen und die Tiefe zum Erstellen eines neuen Movies, die Farbe des späteren Rechtecks und die Koordinaten der linken oberen und rechten unteren Ecke.

```
MovieClip.prototype.rechteck = function (name, tiefe,
    farbe, x1, y1, x2, y2) {
    var referenz = createEmptyMovieClip(name, tiefe);
    referenz.beginFill(farbe, 100);
    referenz.moveTo(x1, y1);
    referenz.lineTo(x2, y1);
    referenz.lineTo(x2, y2);
    referenz.lineTo(x1, y2);
    referenz.endFill();
};
```

Damit die später erzeugte Anzeige mittig auf dem Bildschirm erscheint, wird das Movie *preload* um die Hälfte der Bühnenbreite und die Hälfte der Bühnenhöhe versetzt.

```
scrwidth = System.capabilities.Stage.width;
scrheight = Stage.height;
_x = scrwidth / 2;
_y = scrheight / 2;
```

Da man den Preloadstatus in einem Textfeld ausgeben möchte, definiert man ein Ausgabeformat für die Textfelder. Hier die Farbe Schwarz, die Schriftart Arial und die Schriftgröße 16.

```
stFormat = new TextFormat();
stFormat.color = 0x000000;
stFormat.font = "Arial";
stFormat.size = 16;
```

Nach der Definition für die Ausgabe des Textes kann man das eigentliche Textfeld erzeugen. Dieses bekommt den Namen *T1* und die Tiefe 2 zugewiesen. Die anschließenden Koordinaten beziffern die x- und y-Koordinate sowie die Breite und Höhe. Die „krummen“ Werte erklären sich dadurch, dass das Textfeld mittig angezeigt werden soll. Da es eine Breite von 100 und eine Höhe von 16 hat, wird es um 50 Pixel nach links und 8 Pixel nach

oben verschoben. Mit den nachfolgenden Zeilen wird festgelegt, dass das Textfeld nicht auswählbar ist und die Schrift zentriert angezeigt wird. Zum Schluss weisen wir diesem Textfeld noch die Ausgabeformatdefinitionen zu, die wir zuvor erstellt hatten.

```
createTextField("T1", 2, 0 - 50, 0 - 8, 100, 16);
preloader.T1.selectable = false;
preloader.T1.autoSize = "center";
preloader.T1.setTextFormat(stFormat);
```

Da ein Textfeld für die Ausgabe nicht reichte, haben wir noch ein zweites erstellt, das sich direkt unter dem ersten befindet, dementsprechend (-8)+16= +8 bei den y-Koordinaten.

```
createTextField("T2", 3, 0 - 50, 0 + 8, 100, 16);
T2.selectable = false;
T2.autoSize = "center";
T2.setTextFormat(stFormat);

}
```

Nachdem das *preload*-Movie und die Textfelder darin erstellt und ausgerichtet sind, muss man noch dafür sorgen, dass die Textfelder mit Inhalt gefüllt werden. Da dies eine Aktion ist, die immer wieder geschehen muss, steht sie in einem `onClipEvent(enterFrame)`bzw. da wir das Movie dynamisch erzeugt haben, entsprechend in einer `onEnterFrame`-Prozedur.

```
//Ersatz für OnClipEvent(enterFrame) von preloader
this.preloader.onEnterFrame = function() {
```

Zuerst werden die eingeladenen Bytes und die Anzahl der gesamten Bytes abgefragt und daraus die entsprechende Prozentzahl ermittelt.

```
this.insgesamt = _root.getBytesTotal();
this.geladen = _root.getBytesLoaded();
this.prozent = Math.round(this.geladen / this.insgesamt *
100);
```

Da man nun die aktuelle Prozentzahl errechnet hat, kann man ein Rechteck (den Ladebalken) zeichnen. Wie zuvor bei den Textfeldern wird dieses auch wieder mittig gesetzt und als rechter unterer Endpunkt für die x-Koordinaten die Prozentzahl mit einbezogen.

```
rechteck("R1", 1, 0x00FFFF, 0 - 50, 0 - 8,
    this.prozent - 50, 16 - 8);
```

In das obere Textfeld wird die Prozentzahl mit einem Prozentzeichen ausgegeben.

```
this.T1.text = this.prozent + " %";
```

Für das untere Textfeld errechnet man die eingeladenen kB und gibt dies dann schön formatiert aus.

```
        this.T2.text = Math.round(this.geladen / 1024) + " kB von "
            + Math.round(this.insgesamt / 1024) + " kB eingeladen";
```

Damit, wenn der komplette Film eingeladen ist, auch der Preloader deaktiviert wird, fragt man mit einer if-Abfrage immer ab, ob `prozent` gleich 100 entspricht, und wenn dies zutrifft, entlädt man den Film *preloader* und spielt den eigentlichen Hauptfilm ab.

```
        if(this.prozent==100){
            this.unloadMovie();
            play();
        }
    };
```

Objektorientierter Preloader für mehrere SWFs

Mit Hilfe von OOP kann man relativ komplexe Scripts sehr übersichtlich gestalten, so auch bei diesem Preloader. Der Wunsch war es, eine Liste mit Dateien anzugeben, und Flash sollte diese dann automatisch nacheinander einladen.

Einsatz findet dieses Preloader-Konzept z.B. auf unserer Seite *www.DieFlasher.de*. Zuerst werden damit verschiedene Soundsamples und das Intro eingeladen. Während das Intro dann abgespielt wird, lädt der Preloader weiter im Hintergrund die Hauptseite, weitere Soundsamples und erweiterte Effekte für die Hauptseite ein. So war es möglich, unsere Internetseite auf sehr viele Dateien zu verteilen.

Dies hat den Vorteil, dass ein Besucher dieser Seite, der über eine relativ langsame Internetverbindung verfügt, trotzdem so schnell als möglich auf die Hauptseite gelangt und Sounds oder weitere Animationen, die man nicht direkt benötigt, später automatisch nachgeladen werden.

Gehen wir einmal auf das etwas komplexere Script ein. Wer noch nicht so viel Erfahrung mit Funktionen oder Objekten besitzt, diesen Preloader aber unbedingt einsetzen möchte, liest entweder noch mal die zugehörigen Teile in Kapitel 1 oder überspringt die Erklärungen zum Aufbau und schaut sich dann die doch relativ leichte Modifizierung zum Schluss dieses Kapitels an.

Die Fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel4/05_prelo.fla*.

Leider funktioniert die von Macromedia in Flash installierte Streaminganzeige für diese Art des Einladens nicht! Deshalb ist es nicht direkt möglich, das Beispiel offline zu testen. Damit Sie nicht erst das File auf einen Server hochladen müssen um es zu testen, haben wir es mit auf unseren Server gelegt. Sie finden es unter: *http://www.DieFlasher.de/FlashMXBuchBeispiele/Kapitel4/05_prelo.html*.

```
MovieClip.prototype.rechteck = function (name, tiefe, farbe,
    x1, y1, x2, y2) {
    var referenz = createEmptyMovieClip(name, tiefe);
    referenz.beginFill(farbe, 100);
    referenz.moveTo(x1, y1);
    referenz.lineTo(x2, y1);
    referenz.lineTo(x2, y2);
    referenz.lineTo(x1, y2);
    referenz.endFill();
};
```

Erstellt wieder die Methode zum Zeichnen eines Rechtecks in MovieClip.prototype.

```
MovieClip.prototype.Obj = function(info, name, link, level,
    funktion, startf, endef) {
    this.info = info;
    this.name = name;
    this.link = link;
    this.level = level;
    this.funktion = funktion;
    this.startf = startf;
    this.endef = endef;
}
```

Es wird eine Klasse namens `Obj` erstellt. Diese dient als „geordneter Zwischenspeicher" der Daten eines einzuladenden Movies, Bilds oder Musikfiles. Die Variable `info` enthält einen Infotext zu dem Film. In `name` ist der Name des dafür erstellten MovieClips gespeichert. `link` enthält den Namen des einzuladenden Films. In `level` steht der Level für das neu erstellte Movie. Die letzten drei Variablen dienen als Speicher für die Referenzen der später definierten Funktionen. Diese sind ähnlich callback-Funktionen anzusehen, da das später erstellte Script diese automatisch zu bestimmten Zeitpunkten aufruft. Dazu später mehr.

Gehen wir an dieser Stelle erst einmal auf die durch `Obj` erstellte Liste ein. Diese ist nicht Teil des stetigen Preloaders und muss jeweils immer für den speziellen angepasst werden, je nachdem, was man einladen möchte.

```
objliste = [
new Obj("Bild1", "b1", "02_01.jpg", 11, ausgabe, stdout, FEnde),
new Obj("Bild2", "b2", "02_02.jpg", 12, ausgabe, stdout, FEnde2),
new Obj("Musik", "HG", "stdpop.mp3", x, ausgabe, stdout, MusikEnde),
new Obj("Bild2", "b4", "02_02.jpg", 14, ausgabe, stdout, FEnde4),
new Obj("Bild3", "b3", "02_03.jpg", 13, ausgabe, stdout, FEnde3)
];
```

Dies ist die Liste aus der Beispieldatei, dort laden wir fünf Dateien ein. Zuerst zwei Bilder, danach ein MP3 und noch weitere zwei Bilder. Bei den Parametern handelt es sich um folgende:

1. um einen Informationstext, der beim Einladen mit angezeigt wird (werden kann),
2. um den Namen des späteren Movies oder Objektes,
3. um den Namen des einzuladenden Files,
4. um den Level des erstellen Movies, in den das SWF oder JPG eingeladen wird (bei MP3 ist die Angabe irrelevant,
5. um eine Funktion, die kontinuierlich während des Einladens des Files aufgerufen wird,
6. um eine Funktion, die einmal beim Start des Ladevorgangs des jeweiligen Files gestartet wird,
7. um eine Funktion, die nach Beendigung des Ladevorgangs einmal ausgeführt wird.

```
MovieClip.prototype.Preloader = function(oliste) {
    this.oliste = oliste;
    this.actu = 0;
    if (!(typeof (_root.as) == "movieclip")) {
        _root.createEmptyMovieClip("AS", 20000);
    }
    this.lade();
}
```

Es wird eine Klasse `Preloader` erstellt, die die Basis des Preloaders bildet. In der Klasse werden die Objektliste `oliste` und eine Variable `actu`, die den aktuellen Stand des Ladestatus enthält, gespeichert. Außerdem wird geprüft, ob schon ein MovieClip namens *as* auf `_root` existiert, falls nicht, wird dieser erstellt. Danach wird noch die Methode `lade` aufgerufen, welche den Start des Einladevorganges initialisiert.

```
MovieClip.prototype.preloader.prototype.lade = function() {
    var name = this.oliste[this.actu].name;
    var link = this.oliste[this.actu].link;
    var level = this.oliste[this.actu].level;
    var funktion = this.oliste[this.actu].funktion;
    var endef = this.oliste[this.actu].endef;
    var info = this.oliste[this.actu].info;
    var upper = this;
```

Es wird die Klasse `Preloader` um die Methode `lade` erweitert. Im Konstruktor von `lade` werden als Erstes alle Daten aus der Objektliste `oliste` in lokalen Variablen gespeichert. Zusätzlich wird noch eine Variable `upper` mit einer Referenz auf `Preloader` erzeugt, um später aus einer weiteren Funktion in `lade` noch auf die Variablen in dem `Preloader`-Objekt zugreifen zu können

```
    var anzahl = this.oliste.length;
    var modi = link.slice(link.length - 3).toLowerCase();
    this.oliste[this.actu].startf(info);
```

Außerdem wird noch die Länge der Liste mit den einzuladenden Objekten abgefragt und in der Variable `anzahl` gespeichert. Da man zwischen dem Einladen eines SWF oder JPG zu einem MP3 unterscheiden muss,

speichern wir die letzten drei Zeichen, also die Endung des Filenamens in der Variable `modi`. Gleichzeitig werden die Zeichen durch die Funktion toLowerCase noch in Kleinbuchstaben gewandelt. Die dritte Zeile sieht vielleicht zuerst etwas erschreckend aus. Im Grunde wird dort aber nur auf die Liste zugegriffen und von dem aktuellen Objekt die durch `startf` referenzierte Funktion gestartet.

```
if (modi == "swf" || modi == "jpg") {
```

Es muss, wie zuvor schon erwähnt, eine Fallunterscheidung zwischen JPG, SWF und MP3 gemacht werden. Hier befindet sich also der Zweig für JPG und SWF.

```
_root.createEmptyMovieClip(name, level);
```

Es muss ein MovieClip erstellt werden, in dem das SWF oder das JPG eingeladen wird, da es sehr ungünstig wäre, es direkt auf die Hauptbühne zu setzen.

```
_root.AS.onEnterFrame = function() {
```

Da die Abfrage für den Ladestatus immer wieder erfolgen muss, benutzen wir den im Konstruktor erstellten MovieClip um ein `enterFrame`-Event zu erzeugen.

```
funktion(_root[name].getBytesLoaded(),
    _root[name].getBytesTotal(), info);
```

Dies benutzen wir dann zum einen um die erste übergebene Funktion, die in dem lokalen Variablennamen `funktion` gespeichert ist, immer wieder aufzurufen. Dieser Funktion werden direkt als Parameter die `getBytesLoaded` und `getBytesTotal` sowie die Filminformationen mit übergeben. Vielleicht ist es Ihnen aufgefallen: Unter Flash 5 hätte die innere Funktion `_root.AS.onEnterFrame` unmöglich auf die Variablen der außen liegenden Funktion zugreifen können. Dies geht jetzt in Flash MX problemlos.

```
if ((_root[name].getBytesLoaded() >=
    _root[name].getBytesTotal())
    && (_root[name].getBytesTotal() > 5)) {
```

Danach folgt innerhalb des `enterFrame`-Events jedes Mal noch eine Abfrage, ob der Ladevorgang schon beendet ist. Dies geschieht durch `((_root[name].getBytesLoaded() >= _root[name].getBytesTotal())`. Da es, wenn man mit Flash Files einlädt, immer zuerst einen Moment dauert, bis Flash überhaupt anfängt, diese Files einzuladen, muss man, damit die if-Abfrage nicht direkt zu Beginn wahr wird, diese noch um den folgenden Ausdruck erweitern `&& (_root[name].getBytesTotal() > 5)`. Da die JPG- und SWF-Header einer Datei schon größer als 5 Bytes sind, stellt diese Abfrage auch kein Problem dar.

```
_root.AS.onEnterFrame = null;
upper.actu++;
endef(info,name);
```

Wenn ein File komplett eingeladen ist, wird die `enterFrame`-Funktion wieder gleich null gesetzt (so dass nichts mehr ausgeführt wird, sie wird gelöscht). Außerdem wird die Variable `actu` in dem `Preloader`-Objekt, welches durch `upper` referenziert wird, um eins erhöht. Damit wird gespeichert, dass der erste Film eingeladen ist und der aktuelle Film der um eins höher liegende in der Liste ist.

```
                if (upper.actu < anzahl) {
                    upper.lade();
                }
            }
        };
```

Als letzte Aktion des `enterFrame`-Events wird noch überprüft, ob es überhaupt noch ein weiteres Objekt einzuladen gibt. Falls dies zutrifft, ruft sich die Funktion *lade*, in der man sich ja indirekt immer noch befindet, selbst wieder auf.

```
        _root[name].loadMovie(link);
```

Als letzte Aktion der Verzweigung für SWF und JPG wird noch das entsprechende File in das zuvor erstellte Movie geladen. Es folgt die if-Abfrage für den Fall, dass ein MP3 eingeladen werden soll.

```
    } else if (modi == "mp3") {
        _root.AS.onEnterFrame = function() {
            funktion(_root[name].getBytesLoaded(),
                _root[name].getBytesTotal());
            if (_root[name].getBytesLoaded() >=
                _root[name].getBytesTotal()
                && _root[name].getBytesTotal() > 5) {
                _root.AS.onEnterFrame = null;
                upper.actu++;
                endef(info,name);
                if (upper.actu < anzahl) {
                    upper.lade();
                }
            }
        };
        _root[name].loadSound(link, false);
```

Wie man vielleicht erkennen kann, verhält sich der Zweig für das Einladen eines MP3 fast identisch zu dem vorherigen. Allerdings muss man beim Einladen eines Musikfiles kein leeres Movie erstellen, damit der Sound in einem Sound-Objekt gespeichert wird.

```
    } else {
        trace("Ungültiges Format!\nEs sind nur SWF,JPG und
            MP3 zulässig!");
    }
};
```

Zu guter Letzt haben wir noch einen else-Zweig mit einer Fehlerausgabe für den Benutzer des Preloader erstellt, so dass bei einem ungültigen Datei-Format eine Meldung ausgegeben wird.

Im Folgenden ist der Quellcode aufgeführt, den Sie erstellen müssten, wenn Sie diesen Preloader benutzen möchten. Wir müssten jetzt extra für jedes Bild eine eigene Funktion schreiben um dieses richtig auf der Bühne zu platzieren. Bei einer normalen Internetseite ist dies aber meistens nicht der Fall, dort lädt man mehrere Soundsamples ein oder Hintergrundanimationen, also SWF, die direkt schon richtig positioniert sind, so dass man in den meisten Fällen immer die gleiche Funktion benutzen kann.

```
FEnde = function (info,name) {
    _root.aktuell = "" + info + " erfolgreich eingeladen\n";
    _root[name]._x+=15;
_root[name]._y+=200;
};
FEnde2 = function (info,name) {
    _root.aktuell = "" + info + " erfolgreich eingeladen\n";
    _root[name]._x+=300;
    _root[name]._y+=80;
};
FEnde3 = function (info,name) {
    _root.aktuell = "" + info + " erfolgreich eingeladen\n";
    _root[name]._x+=600;
    _root[name]._y+=200;
};
FEnde4 = function (info,name) {
    _root.aktuell = "" + info + " erfolgreich eingeladen\n";
    _root[name]._x+=600;
    _root[name]._y+=450;
};
MusikEnde = function (info,name) {
    _root.aktuell = "" + info + " erfolgreich eingeladen\n";
    _root[name].start(0, 1000);
};
```

Dies sind die Funktionen, die nach Beendigung des Ladevorgangs gestartet werden, welche Funktion zu welchem File gehört, wird mit in der Objektliste angegeben.

```
stdout = function (info) {
    _root.aktuell = "Lade " + info + "\n";
};
```

Dies ist die Standardausgabe-Funktion, die beim Einladen eines Files gestartet wird. Mit Hilfe dieser Funktion wird der Infotext in dem dafür vorgesehenen Textfeld auf der Hauptbühne angezeigt.

```
ausgabe = function (geladen, gesamt, info) {
    _root.state = int(geladen) + " von " + gesamt +
        " Bytes geladen; " + int(geladen * 100 / gesamt) +
        " % von " + info;
    rechteck("R1", 1, 0x00FFFF, 0 , 0 ,
        int(geladen * 100 / gesamt)*2, 16);
    };
```

Die Funktion `ausgabe` wird innerhalb des `enterFrame`-Events immer wieder ausgerufen, sie sorgt für die dynamische Anzeige des Ladestatus und erzeugt auch den Ladebalken.

```
objliste = [
new Obj("Bild1", "b1", "02_01.jpg", 11, ausgabe, stdout, FEnde),
new Obj("Bild2", "b2", "02_02.jpg", 12, ausgabe, stdout, FEnde2),
new Obj("Musik", "HG", "stdpop.mp3", x, ausgabe, stdout, MusikEnde),
new Obj("Bild2", "b4", "02_02.jpg", 14, ausgabe, stdout, FEnde4),
new Obj("Bild3", "b3", "02_03.jpg", 13, ausgabe, stdout, FEnde3)
];
```

Dies ist noch einmal die schon zu Beginn vorgestellte Liste (ein Array) mit den Objekten, die die Werte für die einzuladenden Filme gespeichert haben.

```
standardpreloader = new Preloader(objliste);
```

Um den Preloadvorgang zu starten muss man ein Objekt der Klasse `Preloader` bilden, dies hat auch den Vorteil, dass man mehrere Preloader gleichzeitig oder hintereinander starten könnte. Wenn man nun einmal die Reihenfolge der Schritte verfolgt, die der Preloader nach seiner Erstellung durchläuft, sehen diese wie folgt aus:

① Erstellung eines `Preloader`-Objektes
② Dementsprechend Konstruktoraufruf der Klasse `Preloader`
③ Der Konstruktor erstellt falls noch nicht vorhanden ein Movie namens AS
④ Setzt `actu` auf 0 und ruft die Methode `lade` auf
⑤ Die Methode lade benutzt `actu` um zu erfahren welches Objekt aus der Liste geladen werden soll
⑥ Da `actu` 0 ist, fängt die Methode `lade` vorne an
⑦ `lade` ruft die selbst definierte Funktion auf, die beim Start des Einladevorgangs des ersten Objektes gestartet werden soll, auf
⑧ `lade` überprüft, ob es sich um ein SWF, JPG oder um ein MP3 handelt, und erstellt für Ersteres ggf. ein Movie
⑨ `lade` weist dem zuerst erstellten Movie As eine enterFrame-Funktion zu bzw. belegt die Funktion mit Inhalt.
⑩ `lade` lädt dann noch das entsprechende File ein und ist damit beendet
⑪ Die erstellte `enterFrame`-Funktion hingegen überprüft immer den `lade`-Vorgang und ruft immer wieder die selbst erstellte Funktion, die dafür vorgesehen ist und in der Liste als Referenz eingetragen wurde, auf

⑫ Des Weiteren überprüft die `enterFrame`-Funktion, ob das File fertig eingeladen ist, falls dies zutrifft, wird geprüft, ob dies das letzte File in der Liste war, dann ist der Preloadervorgang beendet. Wenn noch weitere Objekte in der Liste existieren, die eingeladen werden sollen, wird die Methode `lade`aus dem gebildeten `Preloader`-Objekt noch einmal gestartet und alles beginnt wieder ab Schritt fünf aufs Neue.

4.5 Sound

Mit Flash ist es möglich, verschiedene Aktionen mit Sound zu unterlegen. Auch längere Musikstücke kann man ohne Probleme im Hintergrund laufen lassen. Wie man dies am besten in Szene setzt, wird im Folgenden beschrieben.

Die Bibliothek Ihres Films können Sie mit Strg + L aufrufen (L steht für Library).

Flash MX bringt eine Menge an Soundsamples mit. Sie finden sie unter Fenster › Allgemeine Bibliotheken › Sounds. Per Drag&Drop können Sie die Samples der *Allgemeinen Bibliothek* in die Bibliothek Ihres Films verschieben.

Sound sollte so spät wie möglich importiert werden, da dieser den Testlauf zum Erstellen eines SFW erheblich verlängert. Falls Sie den Sound vorab benötigen, sollten Sie die Qualität erst einmal auf die niedrigste Stufe setzen (Datei › Einstellungen für Veröffentlichung › Flash oder direkt in der Bibliothek Ihres Filmes unter „Eigenschaften“ des jeweiligen Sounds).

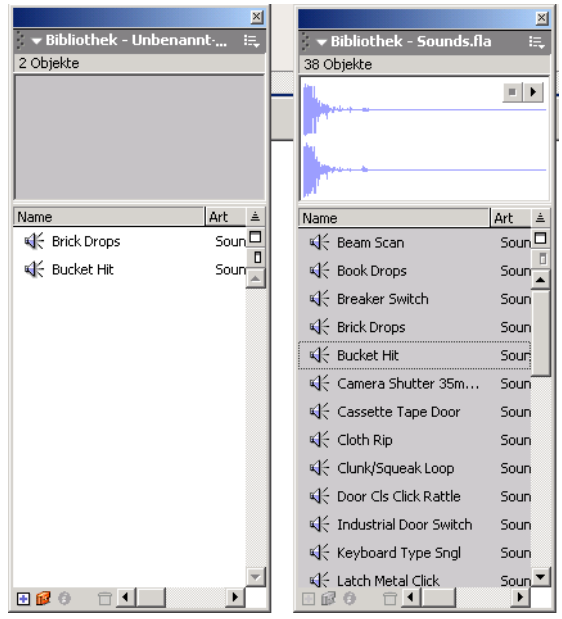


***Abbildung 4.39:** Zwei offene Bibliotheken*

Sie können die Samples auch direkt auf die Bühne Ihres Films ziehen, dann werden diese direkt dem jeweiligen Frame zugewiesen und an dieser Stelle abgespielt. Ein anderes Soundsample importieren Sie über Datei › Importieren.

Ein einem Frame zugeordnetes Sample wird in der Frameleiste wie folgt angezeigt:

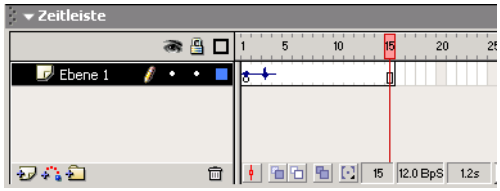


Abbildung 4.40: Frameleiste mit Sound

Wenn Sie sich das Eigenschaftsfenster des Frames anschauen, dem Sie den Sound zugewiesen haben, werden dort die aktuellen Sound-Einstellungen angezeigt.

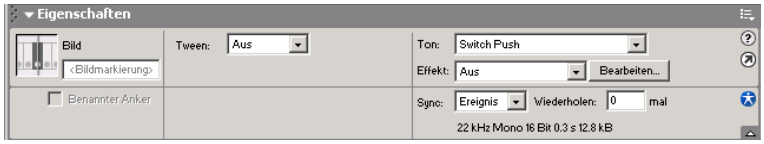


Abbildung 4.41: Eigenschaften eines Frames mit zugewiesenem Sound

Unter dem Menü *Ton* können Sie eines der bisher importierten/eingefügten Soundsamples dem jeweiligen Frame zuordnen.

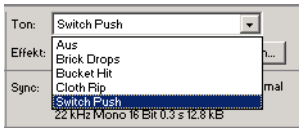


Abbildung 4.42: Sound-Auswahl

Alle dem Film zugewiesenen Sounds sind auswählbar.

In dem darunter liegenden Feld *Effekt* können Sie die Eigenschaften des Sounds festlegen, z.B. ob dieser eingeblendet oder ausgeblendet werden soll.

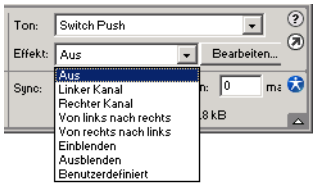


Abbildung 4.43: Anzeige der verschiedenen vordefinierten Effekte

In dem daneben liegenden Feld *Bearbeiten* können Sie die Soundeigenschaften selbst definieren. Den Lautstärkepegel für den rechten und linken Kanal können Sie variieren, so dass Sie eine eigene Hüllenkurve für den jeweiligen Sound erstellen können.

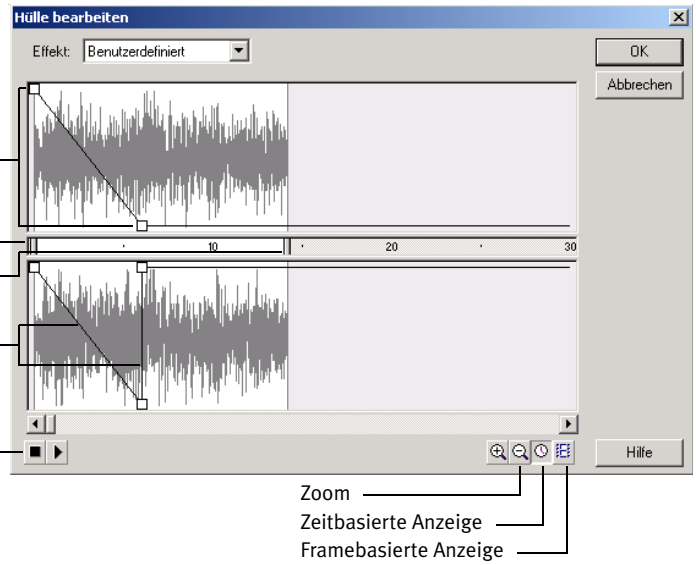


Abbildung 4.44: *Anpassung der Hüllenkurve eines Sounds*

Wenn Sie unterschiedliche Hüllenkurven erzeugen, erhalten Sie unterschiedliche Soundvarianten. Auf diese Weise brauchen Sie eventuell keinen neuen Sound zu importieren und Ihr SWF wird kleiner.

Durch Klicken auf die Soundsamples erzeugen Sie neue Lautstärkepegelbegrenzer, die Sie beliebig, für jeden Kanal einzeln, verschieben können. Über die Start- und Stoppunkte können Sie einzelne Sequenzen wegschneiden, so dass nur bestimmte Teile im Loop laufen und immer nur ein Teil des Soundsamples abgespielt wird.

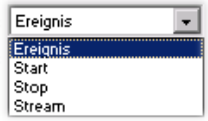


Abbildung 4.45: *Abspielverhalten des Soundsamples*

In dem Feld *Sync* können Sie das Abspielverhalten des Soundsamples einstellen.

- „Ereignis“ bedeutet, dass der Sound an der Position komplett abgespielt wird.
- Mit „Start“ wird das Soundsample hingegen nur so lange abgespielt, wie der Key-Frame, in dem sich das Soundsample befindet, nicht durch ein neues Sample unterbrochen wird.
- Mit „Stop“ wird das angegebene Soundsample gestoppt.
- „Stream“; ist für eine Veröffentlichung im Internet gedacht. Hier braucht nur der Anfang des Soundsamples eingeladen zu sein, um abgespielt zu werden. Der Sound wird immer passend zu den Frames abgespielt. Wenn die Frames nicht schnell genug eingeladen werden, werden sie übersprungen. Falls die Brandbreite des Users zu

gering ist, um den Sound im Voraus einzuladen, kann dies dazu führen, dass der Film kurz anhält.

In dem Feld *Wiederholungen* kann angegeben werden, wie oft das jeweilige Soundsample hintereinander abgespielt werden soll.

Die Größe des Soundpuffers für den Streaming-Sound kann man durch einen ActionScript-Befehl ändern. Standardmäßig steht dieser Soundpuffer auf 5 Sekunden. Das heißt, es müssen fünf Sekunden Laufzeit des Soundsamples eingeladen sein, bevor dieses abgespielt wird. Dieser Wert kann durch den Befehl `_soundbuftime` geändert werden. Hier wird z.B. die Soundbuffertime auf 10 Sekunden erhöht:

```
_soundbuftime = 10;
```

Man sollte diesen Wert bei Sounds, die in sehr guter Qualität für die Übertragung bereitgestellt werden, erhöhen – ansonsten würde es beim Abruf durch Besucher mit geringen Datenübertragungsraten zu Aussetzern kommen.

4.6 Komponenten anwenden

Flash bietet nun auch fertige Komponenten an, die Ihnen viel Arbeit ersparen können, sie entsprechen in etwa dem, was Sie vielleicht aus Flash 5 als SmartClips kannten. Folgende Komponenten stehen Ihnen schon ab der Installation von Flash MX zur Verfügung:

- RadioButton (Felder werden abhängig voneinander ausgewählt)
- CheckBox (Felder werden unabhängig voneinander ausgewählt)
- ComboBox (Ein Feld kann aus einem Dropdown ausgewählt werden)
- ListBox (Felder können aus einer Liste ausgewählt werden)
- PushButton (fertig designte Schaltfläche)
- ScrollBar (Laufbalken für ein Textfeld)
- ScrollPane (Laufbalken für ein Bild)

Klappen Sie das entsprechende Register in der Entwicklungsumgebung auf, so haben Sie alle oben genannten Komponenten zur Auswahl und können diese per Drag&Drop im Movie platzieren.

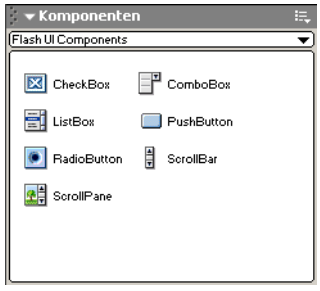


Abbildung 4.46:
Auflistung aller Standardkomponenten

Im Folgenden wird nur kurz auf die Verwendung der Komponenten eingegangen, da vieles selbsterklärend ist.

Abbildung 4.47: Ein RadioButton mit der Standard-Beschriftung „Radio Button“

RadioButton

Eigenschaften und ihre Funktionen:

- Label → Beschriftung des Elements
- Initial State (Boolean) → Status bei der ersten Anzeige (true = mit Häkchen)
- Group Name → Gruppe, in der sich der Status bedingt, pro Gruppe kann nur ein Element gewählt werden (Geschlecht m/w zum Beispiel)
- Data → Wert des Elements
- Label Placement → Ausrichtung der Beschriftung (Links, Mitte, Rechts)
- Change Handler → wird ausgelöst durch „onChange“

Ein RadioButton eignet sich, wenn in Formularen aus einer Reihe von Möglichkeiten nur eine Auswahl getroffen werden soll. Anhand der Namensvergabe der Gruppe lassen sich diese Auswahlmöglichkeiten verschieden, je nach Bedarf, zusammenfassen.

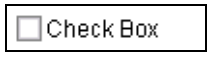


Abbildung 4.48: Eine CheckBox mit der Standard-Beschriftung „Check Box“

CheckBox

Eigenschaften und ihre Funktionen:

- Label → Beschriftung des Elements
- Initial State (Boolean) → Status bei der ersten Anzeige (true = mit Häkchen)
- Label Placement → Ausrichtung der Beschriftung (Links, Mitte, Rechts)
- Change Handler → wird ausgelöst durch „onChange“

Eine CheckBox bietet die Möglichkeit einen Status zu setzen bzw. eine Auswahl zu treffen. Im Unterschied zum RadioButton bestehen bei der Checkbox nicht zwingend Abhängigkeiten zu anderen Elementen.

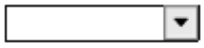


Abbildung 4.49: Eine ComboBox

ComboBox

Eigenschaften und ihre Funktionen:

- Editable (Boolean) → Eingabe möglich
- Labels (Array) → Beschriftung der Elemente
- Data (Array) → zugehörige Werte der Elemente
- Row Count → Zeilenanzahl
- Change Handler → wird ausgelöst durch „onChange“

Entspricht dem, was man weitläufig als DropDown bezeichnet. Genauso wie beim RadioButton wird hier nur eine Auswahl getroffen, jedoch werden in dieses Element alle Einträge der dem RadioButton entsprechenden Gruppe eingetragen.

ListBox

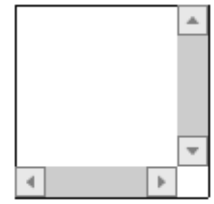


Abbildung 4.50: Eine ListBox

Eigenschaften und ihre Funktionen:

- Labels (Array) → Beschriftung der Elemente
- Data (Array) → zugehörige Werte der Elemente
- Select Multiple (Boolean) → mehrere Auswahlen können getroffen werden
- Change Handler → wird ausgelöst durch „onChange“

Die ListBox ist ein Feld, das eine Liste von Items anzeigt. Man kann hier einstellen, ob eines oder mehrere gewählt werden können.

PushButton

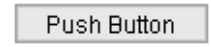


Abbildung 4.51: Ein PushButton

Eigenschaften und ihre Funktionen:

- Label → Beschriftung des Elements
- Change Handler → wird ausgelöst „onChange“

Entspricht einem normalen Button, wie man ihn auf herkömmliche Weise auch selbst erstellen kann, dieser hier ist leicht zu editieren und hilft bei einheitlicher Gestaltung.

ScrollBar

Abbildung 4.52: Die vertikale Scrollbar

Eigenschaften und ihre Funktionen:

- Target TextField ⟶ Textfeld, das der Scrollbalken steuert
- Horizontal (Boolean) ⟶ horizontaler Scrollbalken wird angezeigt

Dies ist ein Scrollbalken, der einem Textfeld zugewiesen werden kann, je nach Umbrucheinstellungen des Textfeldes sogar mit horizontalem Scrollbalken.

ScrollPane

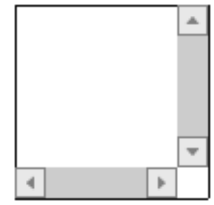


Abbildung 4.53: Der ScrollPane-Container

Eigenschaften und ihre Funktionen:

- Scroll Content ⟶ zu scrollender Inhalt
- Horizontal Scroll (Boolean) ⟶ horizontaler Scrollbalken wird angezeigt
- Vertical Scroll (Boolean) ⟶ vertikaler Scrollbalken wird angezeigt
- Drag Content (Boolean)

Dies ist ein Container, in den man einen Inhalt, z.B. ein Bild, laden kann. Ist dieses Bild u.U. größer als der Bereich, der für dieses Element vorgesehen ist, so werden Scollbars angeboten, alternativ kann man auch die Navigation durch Drag&Drop hinzuschalten.

Zugriff auf die Elemente

Später müssen diese Werte, die mit Hilfe der Komponenten eingetragen wurden, ja ausgewertet werden. Hierzu gibt es ein paar Zugriffsmöglichkeiten. Im Folgenden ist „Element“ der Instanzname des auszuwertenden Elementes.

- Element.getValue(); → liest den Wert eines Elementes
- Element.getSelectedItem().label; → Beschriftung des aktuell ausgewählten Elementes
- Element.getSelectedIndex(); → ganzzahliger Wert, gibt an, der wievielte Eintrag gewählt ist
- Element.setValue(); → setzt den Wert eines Elementes
- Element.setSelectedIndex(); → legt fest, das wievielte Element auf „ausgewählt“ gestellt wird

4.7 Mailfunktion

Durch den Befehl `mailto` ist es möglich, den Standard-E-Mail-Client des Besuchers aufzurufen und diesem Texte zu übermitteln. Der Besucher kann den Text gegebenenfalls ändern und diesen anschließend mit seinem E-Mail-Programm verschicken.

Falls Sie diesen Befehl im Flashdebugging-Modus laufen lassen, öffnet sich ein neues Browserfenster und erst danach wird das E-Mail-Programm gestartet. Dies ist ein normaler Vorgang, da der `mailto`-Befehl mit `get URL` aufgerufen wird. Normalerweise wird die Seite schon in einem Browser angezeigt und deshalb kein neues Fenster geöffnet.

```
GetURL(mailto:IhreEmailAdresse@Server.de);
```

Dieser Befehl ruft direkt den Standard-E-Mail-Client des Besuchers auf. Sie können diesen Befehl direkt in der Frameleiste einbauen, so dass, wenn der Besucher einen gewissen Frame erreicht hat, automatisch das E-Mail-Programm aufgerufen wird. Die Mehrzahl der Besucher wird es allerdings als unhöflich empfinden, wenn plötzlich ohne Abfrage ihr E-Mail-Client geöffnet wird. Deshalb ist es sicherlich ratsam, den Befehl in Verbindung mit einem davor geschalteten Dialogfeld zu verwenden.

Listing 4.6: Ruft den E-Mail-Client des Besuchers auf Knopfdruck auf

```
on (release) {
GetURL(mailto:IhreEmailAdresse@Server.de)
}
```

Mit diesem Befehl wird ein leeres E-Mail-Formular mit der Empfängeradresse *IhreEmailAdresse@Server.de* geöffnet. Wenn Sie nun direkt den Betreff oder einen Inhalt vorgeben möchten, sieht dieser Befehl folgendermaßen aus:

```
on (release) {
GetURL("mailto:IhreEmailAdresse@Server.de?subject=Allgemeine
Anfrage zu den Produkten &body=Geben Sie hier bitte Ihre Anfrage
ein");
}
```

Listing 4.7: Aufruf des E-Mail-Clients mit Vorgabe für den Inhalt

Die Anführungszeichen können entfallen, wenn man die Zeile nicht als Ausdruck deklariert. Diese Zeile braucht nicht als Ausdruck deklariert zu werden, da Flash keine Variablen mit an das E-Mail-Programm übergibt.

Die komplette Befehlszeile:

```
mailto:IhreEmailAdresse@Server.de?subject=Allgemeine Anfrage zu
    den Produkten &body=Geben Sie hier bitte Ihre Anfrage ein
```

ist keine Flash-Syntax, sondern stammt von HTML ab.

Wenn man hingegen *Betreff* und *Inhalt* mit Hilfe einer Variablen übergeben möchte, muss man den Text als Ausdruck bezeichnen und den für Flash „unwichtigen" Teil, den HTML-Befehl, in Anführungszeichen schreiben. Dadurch arbeitet Flash diesen Teil nicht ab und es kommt zu keiner Syntaxfehlermeldung.

```
on (release) {
GetURL("mailto:DeineEmailAdresse@Server.de?subject=" add Betreff
    add "&body=" add Inhalt);
}
```

Listing 4.8: Aufruf des E-Mail-Clients mit Übergabe des Inhalts von Variablen

Der Befehl enthält die zwei Variablennamen `Betreff` und `Inhalt`, was man auch daran erkennt, dass diese außerhalb der Anführungszeichen stehen. Flash sieht den Text außerhalb der Anführungszeichen als „Befehle" an und den Text innerhalb als String. Mit dem Befehl `add` bzw. `+` verkettet man Strings und Variablen.

Die zwei Variablen sollten aber auch einen Text übergeben. Entweder gibt man diesen direkt mit `SetVariable` vor, oder man ermöglicht es dem Besucher, die Variable durch ein Eingabefeld zu ändern.

Vielleicht auch interessant zu diesem Thema zu betrachten ist der E-Mail-Validator, den wir in Kapitel 1.2 im Abschnitt „Stringfunktionen und Eigenschaften" ab Seite 29 beschrieben haben.

4.8 Tweenings

Von Tweenings haben Sie vielleicht noch nie zuvor gehört oder Sie gehören zu der Sorte Anwender, die schon seit Flash 3 dabei sind. An dieser Stelle möchte ich zunächst auf einige grundlegende Dinge betreffend Tweenings eingehen und später auch ein paar Tricks und Kniffe verraten, die auch versierte Anwender vielleicht noch nicht kennen.

Unterschieden wird in zweierlei Tweenings:

- Bewegungstweeing
- Formtweening

Ein Tweening ist eine kleine Animation, in der man von zwei Schlüsselbildern als Grundlage ausgeht. Sofern beide Schlüsselbilder aus dem gleichen Objekt (einer Kopie) erzeugt wurden, kann Flash eigene Zwischenschritte berechnen und als Animation ausgeben. Hier sind die Namen eigentlich selbsterklärend, aber noch mal für das Protokoll:

Bewegungstweenings – hier berechnet Flash die Bewegung eines Objektes. Optional können folgende Einstellungen noch vorgenommen werden:

- Pfad – die Bewegung folgt nicht der kürzesten Verbindung der zwei Punkte, sondern orientiert sich an dem Pfad (hier optional auch noch zusätzlich daran ausgerichtet)
- Drehung – Anzahl der Rotationen, die das Objekt auf dem Weg von A nach B zurücklegen soll
- Abbremsen – Geschwindigkeitskontrolle der Bewegung (abbremsendes Auto)

Das Bewegungstweening

Objekte, die getweent werden, sollten zumindest gruppiert sein!

Um ein Bewegungstweening zu erstellen braucht man natürlich ein Objekt, das bewegt werden soll; ich habe in diesem Fall einen kleinen Stern gezeichnet, der nun von Punkt A nach B wandern soll. Hierfür habe ich die Zeichnung gruppiert (Strg+G), was sich immer anbietet, zumal es Flash ansonst selbst übernimmt – zu erkennen an Objekten in der Objektbibliothek, benannt „TweenXX“. Dann habe ich ein zweites Schlüsselbild (F6) erzeugt – quasi eine Kopie des ersten – und das Objekt an die Zielposition verschoben. Zwischen die beiden Schlüsselbilder kann man nun noch ein paar Frames legen, je nach Bedarf. Macht man einen Rechtsklick auf diese, kann man die Erstellung eines Bewegungstweens auswählen.

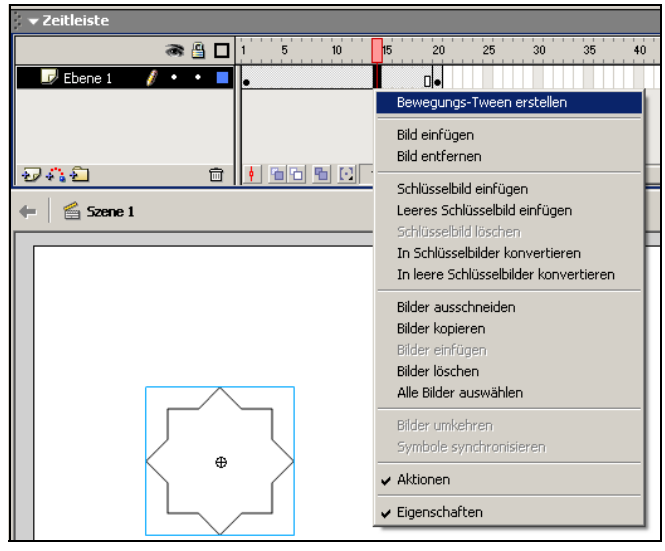


Abbildung 4.54: *Hier nimmt das Tween seinen Anfang*

Das Ergebnis ist in der Hauptzeitleiste zu erkennen, die beiden verbundenen Schlüsselbilder sind durch einen Pfeil auf violettem Hintergrund verbunden.

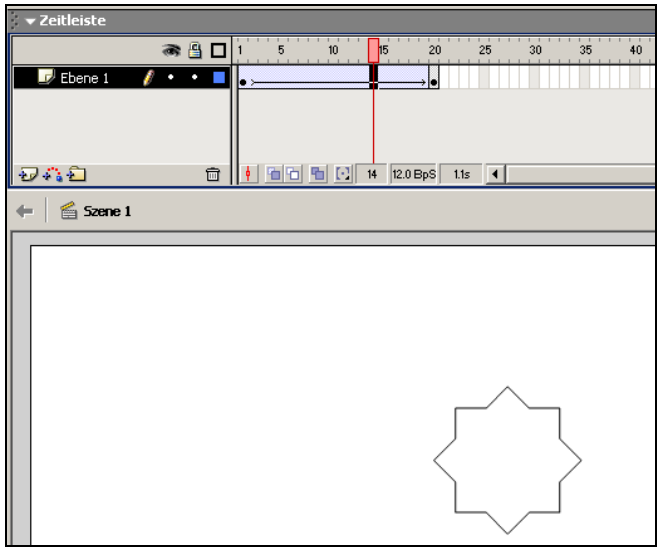


Abbildung 4.55: Das fertige Tween

Was neben dem Unterschied auf der Hauptzeitleiste gleichzeitig auch eine sofortige Erfolgskontrolle ist – man sieht es hier im Vorher/Nachher-Vergleich –, das Objekt hat sich bewegt. Nun bietet uns Flash noch eine ganze Reihe von Optionen, was welchen Weg und vor allem wie sich das Objekt bewegen soll. Eine recht verbreitete Methode ist das Tweenen mit einem zusätzlichen Pfad, der den Weg bezeichnet, an dem sich die Animation orientiert. Folgend nochmals das gleiche Tweening nur mit einem Pfad.

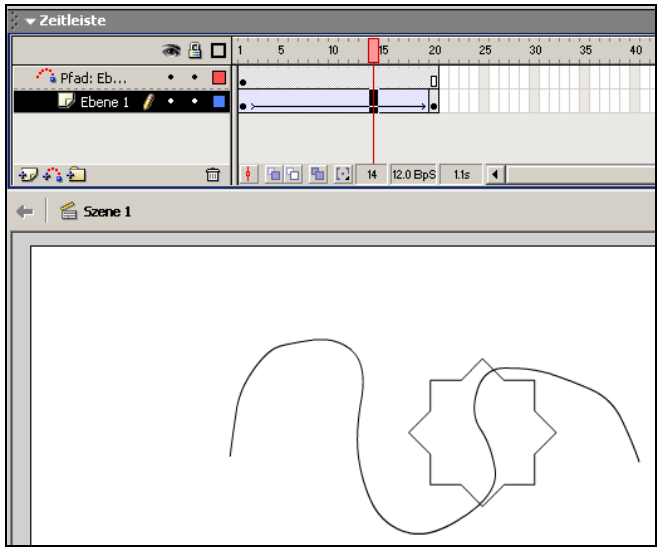


Abbildung 4.56: Das Tweening entlang eines Pfades

Um auf Nummer sicher zu gehen, dass das Objekt auch dem Pfad folgt, probieren Sie die Animation einfach kurz durch Drücken von ↵ aus. Um noch ein paar andere Spezialitäten nicht unerwähnt zu lassen erläutere ich kurz das zum Tweening gehörige Dialogfeld.

Abbildung 4.57: Eigenschaften-Dialogfeld eines Tweenings.

Das Kontrollkästchen Skalieren besagt, ob eine Größenveränderung mitgetweent werden soll. Ist diese Kästchen nicht angehakt und die Objekte der beiden Schlüsselbilder weisen unterschiedliche Größen auf, so entsteht ein Sprung. Der Regler Abbremsen ermöglicht es, den Beschleunigungsvorgang zu definieren, von -100 am Anfang bis 100 am Ende. Im Dropdown Drehen können Sie auswählen, ob Sie die Drehung der Objektes Flash überlassen, ob es nicht gedreht werden soll oder – wenn Sie es kontrollieren wollen – wie oft in welche Richtung. Die unteren Kontrollkästchen sind für das Pfadtweening von Bedeutung, je nachdem, wie das Objekt am Pfad ausgerichtet werden soll (im gleich bleibenden Winkel zur Tangente in dem Punkt, an dem sich das Objekt befindet). Auf die tontechnischen Besonderheiten wird an anderer Stelle näher eingegangen, das Tweening selbst kann man mit einem Ton verknüpfen, d.h. beide werden zeitgleich ausgeführt.

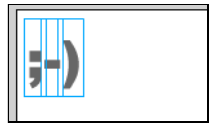
Das Formtweening

Ein Formtweening verändert im Gegensatz zum Bewegungstweening nicht die Position, sondern die Form eines Objektes. Sie erstellen für ein Formtweening wieder zunächst zwei Schlüsselbilder, dann brechen Sie deren Inhalte aus Shapes herunter.

Abbildung 4.58: Markiertes Textfeld.

Haben Sie z.B. einen Text markiert, drücken Sie einfach Strg+B mit der Folge, dass Sie nun mehrere Textfelder, je Buchstaben eines, haben.

Abbildung 4.59: Ein auf mehrere Textfelder heruntergebrochenes Textfeld.



Dies reicht für unsere Zwecke nicht aus, durch erneutes Drücken der Tastenkombination wird aus dem Textfeld ein Shape gebildet.

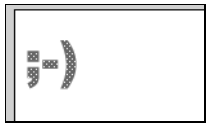


Abbildung 4.60:
In Shapes gewandelte Zeichen

Nun legen Sie einfach zwei Schlüsselbilder mit Shapes dieser Form an, klicken auf die Frames, die zwischen den beiden Schlüsselframes liegen, und wählen unten im Eigenschaften-Dialog unter Tween › Form aus.

Das von Flash errechnete vorgeschlagene Tweening verhält sich nun aber vielleicht nicht so, wie Sie es sich vorstellten. Hier haben Sie die Möglichkeit, synonym zu den Pfaden im Bewegungstween, einzugreifen. Mit Hilfe von Formmarken können Sie gezielt Ursprungs- und Zielpunkt innerhalb der beiden Shapes festlegen. Formmarken fügen Sie mit Strg + ⇧ + H ein und platzieren sie einfach per Drag&Drop.

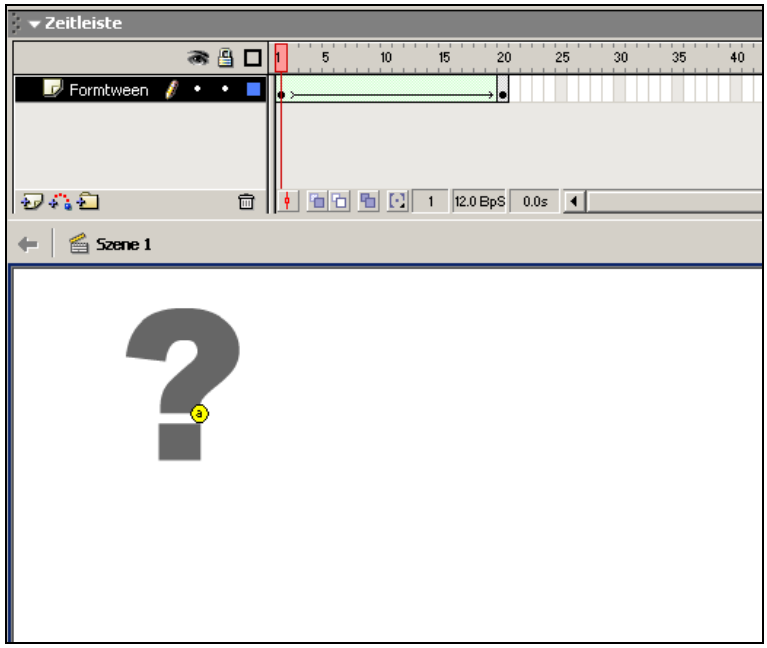


Abbildung 4.61:
Formmarke im Ursprungsbild festgelegt.

Hier legen Sie eine Formmarke an das erste Shape an. Sie merken schnell, ob Sie einen gültigen Bereich getroffen haben, da die Marken automatisch magnetisch an den Umriss des Objektes springen.

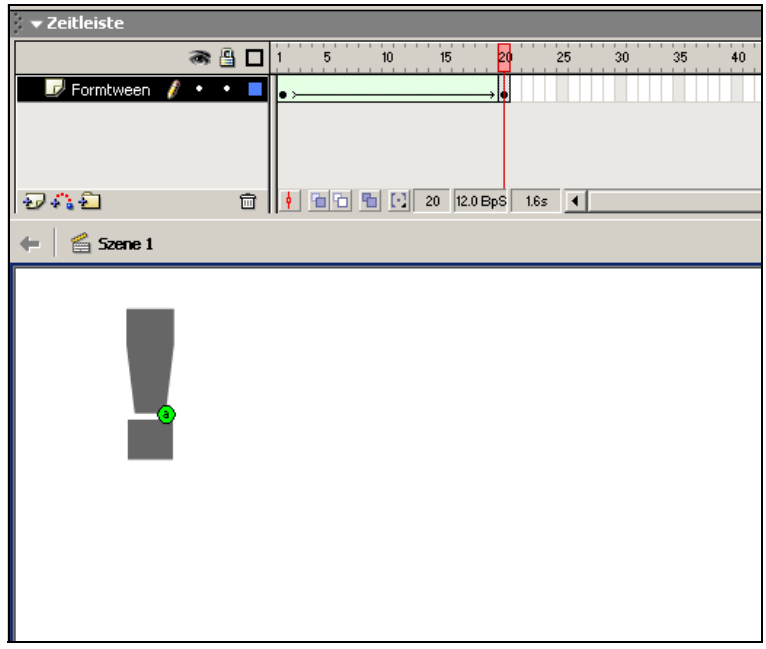


Abbildung 4.62: Formmarke im Zielbild festgelegt.

Gleiches wiederholen Sie für das Ziel-Shape, somit haben Sie für Flash verbindlich festgelegt, dass diese Bereiche aus Start- und Ziel-Shape zueinander geführt werden.

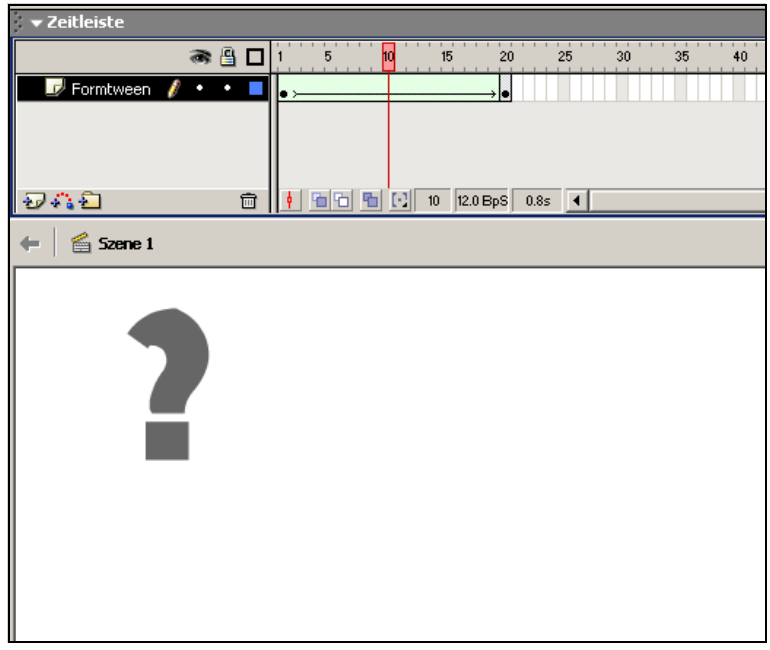


Abbildung 4.63: Zwischenergebnis mit festgelegten Formmarken.

Hier im Zwischenergebnis – da Flash alles schon in der Entwicklungsumgebung errechnet – können Sie schnell und einfach überprüfen, ob der Verlauf Ihrer Vorstellung entspricht. Erlaubt sind 26 Formmarken pro Tweening, von A-Z gekennzeichnet.

Kombinationsmöglichkeiten

Haben Sie die Idee von Tweenings verinnerlicht, stellen Sie sich vielleicht die Frage, wie man komplexere Animationen erstellt. Nicht immer lassen sich Bewegungsabläufe so einfach klassifizieren, meistens ist es eine Kombination aus Form und Bewegungstweening. An folgendem Beispiel möchte ich Ihnen veranschaulichen, wie man eine solche Kombination erstellt.

In unserer Szene soll ein Vogel Richtung Sonnenuntergang fliegen und eine Wolke am Himmel vorbeiziehen. Hier werden die Bewegungsabläufe wie folgt unterteilt:

- Der Flügelschlag des Vogels wird durch ein sich wiederholendes Formtweening erzeugt
- Die Flugbahn des Vogels durch ein Bewegungstweening mit Pfadangabe
- Die Größenveränderung (Aufgrund der zunehmenden Entfernung) wird an das Bewegungstweening gekoppelt
- Die vorbeiziehende Wolke kann man durch ein einfaches Bewegungstweening darstellen (ohne Pfadangabe)

Zuerst wird der Flügelschlag modelliert. Im Folgenden sind die verschiedenen Keyframes angezeigt, wie Sie sie zu setzen haben, wenn Sie den gleichen Effekt erzielen möchten.

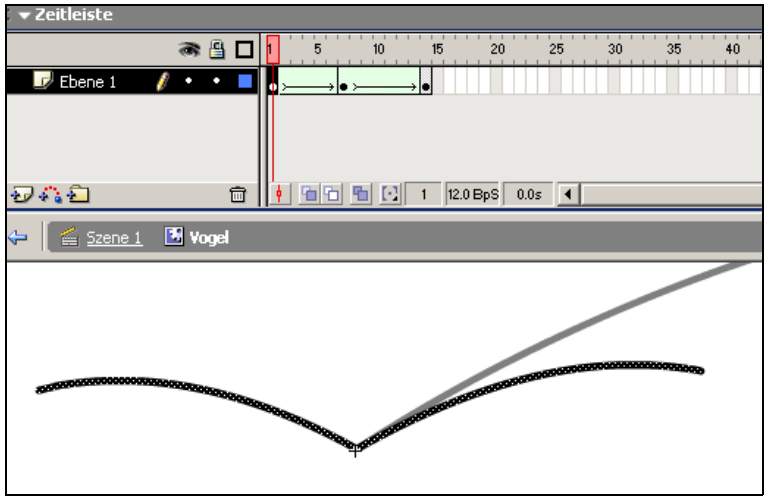


Abbildung 4.64: Bild 1 des Flügelschlags

Abbildung 4.65: Bild 2 des Flügelschlags

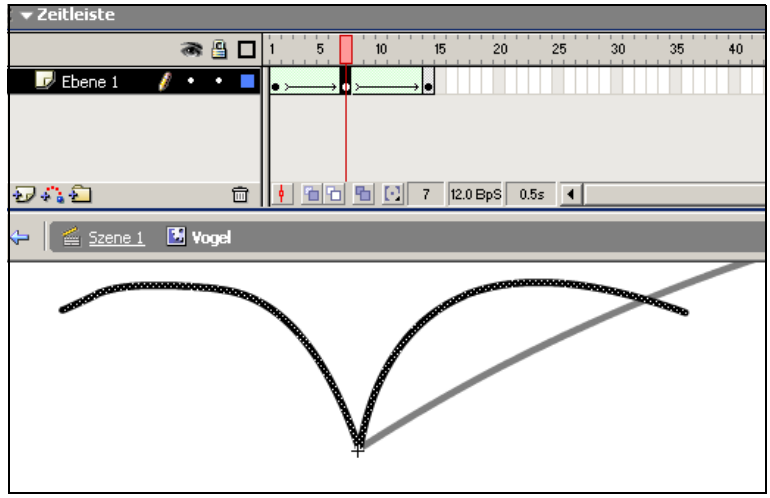
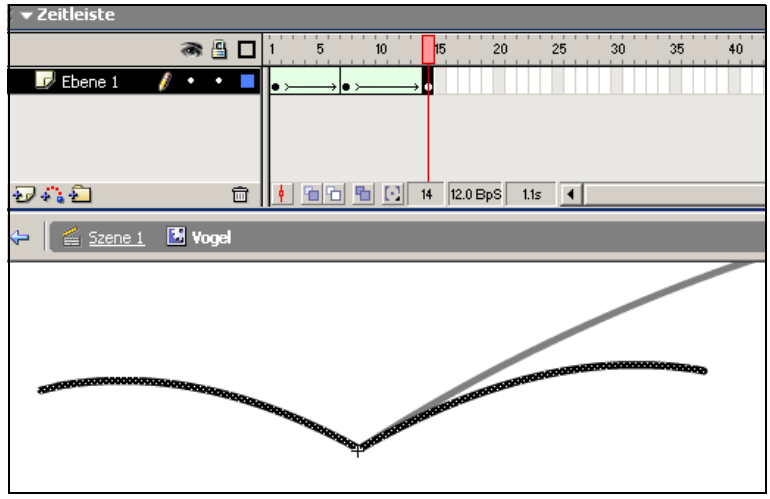


Abbildung 4.66: Bild 3 des Flügelschlags



Wenn Sie nun diese Animation noch in eine eigene Instanz legen, können Sie diese auf der Hauptbühne noch an einem Pfadtweening ausrichten und damit eine Flugbahn festlegen. Das Ganze könnte dann wie folgt angeordnet sein.

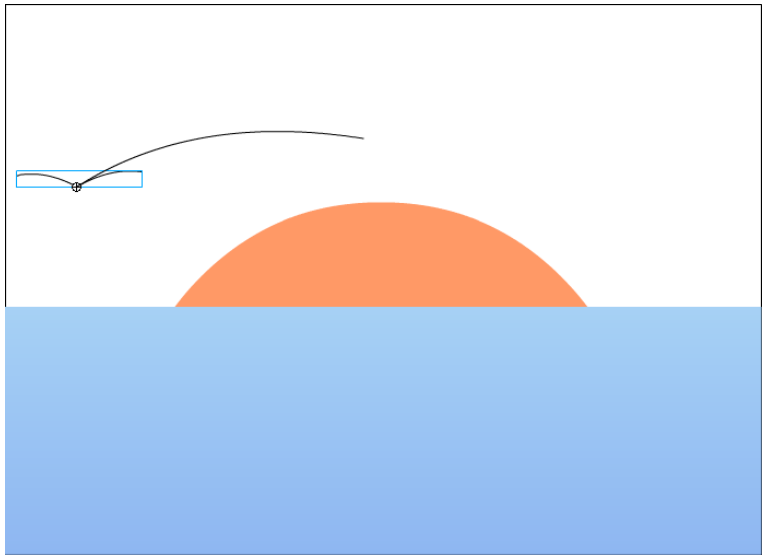


Abbildung 4.67: Möwe im Anflug

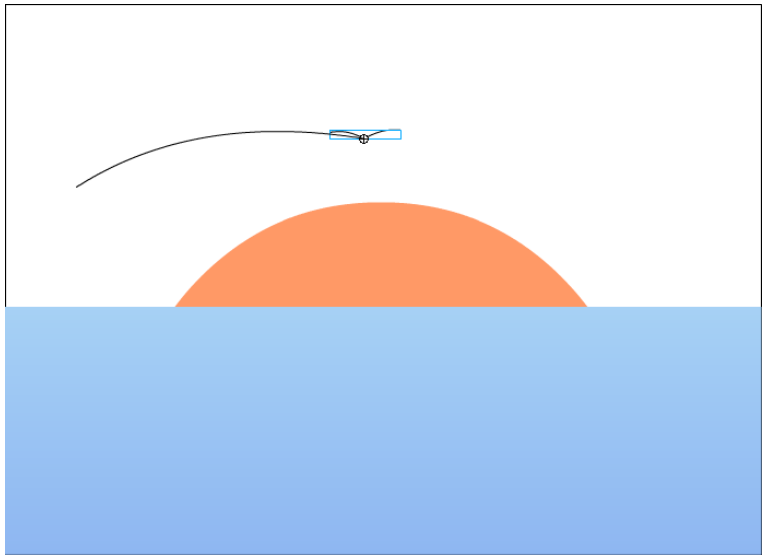


Abbildung 4.68: Möwe verschwindet am Horizont

Die fertige Animation zum direkten Nachvollziehen des Beispiels finden Sie auch auf der beiliegenden CD.

Advanced Elements

5.1 Externe Dateien einbinden

Es ist möglich, in Flash „externe“ Daten per ActionScript einzuladen. Diese Daten müssen allerdings nach einer bestimmten Syntax aufgebaut sein. Des Weiteren erläutern wir das Einlesen von HTML-Dateien. Wie man XML-Dateien importiert, erfahren Sie in Kapitel 6.3.

Das Einbinden von JPG-, MP3- oder SWF-Dateien per ActionScript wird in Kapitel 4.4 im Abschnitt „Objektorientierter Preloader für mehrere SWFs“ ab Seite 193 erläutert.

Textdateien

Durch den Befehl `loadVariablesNum("http://Server.de/Verzeichnisse/datei.txt", 0)`; lesen Sie Daten aus einer Datei ein. Der Dateiname oder die Dateiendung ist dabei unwichtig. Die Datei muss lediglich einer gewissen Syntax entsprechen und sich auf demselben Server bzw. in derselben Subdomain befinden. Flash setzt die entsprechenden Variablen durch das Einlesen der Datei in den entsprechenden Level oder Movie. o bedeutet, dass die Variablen in `_Level0` eingeladen werden. Sie können die Variablen aber auch direkt in einen Film einladen, um so eine bessere Übersicht über die Variablen zu erhalten.

```
loadVariables ("http://www.Server.de/Name.txt", "instanzname");
Data.txt:
inhalt=Das hier ist der Text aus der Textdatei. Mit 2 Sonder-
zeichen : %26, %25
&inhalt2=Das hier ist der 2´te Text aus der Textdatei mit ä,ü
und ö
&Variable=1&
&Variable2=1&
```

Man muss alle Variablen in der TXT-Datei direkt mit zwei &-Zeichen versehen. Dadurch wird diese direkt als String übergeben. Wenn man allerdings mit diesem Wert rechnen möchte, muss man ihn erst mit `parseFloat()` in eine Zahl umwandeln.

Wenn Sie Sonderzeichen darstellen möchten, zum Beispiel ein &-Zeichen, welches für die Verkettung der Variablen reserviert ist, gibt es einen entsprechenden Code, bei dem Sie das jeweilige Zeichen durch das entsprechende Codezeichen ersetzen (wie oben in der TXT-Datei schon angedeutet). Dabei haben wir uns auf die URL-Encoding-Tabelle bezogen. Eine bessere Art wäre es aber, die Sonderzeichen mit Hilfe des Unicodes darzustellen. Mehr Informationen zum Unicode finden Sie in Kapitel 1.2 im Abschnitt „Numerisch“ ab Seite 19.

Wenn Sie eine TXT-Datei öfters von einem Server herunterladen müssen, weil sich deren Inhalt geändert hat, entsteht das Problem, dass der Browser diese Datei aus dem Cache liest. Um dies zu vermeiden, besteht die Möglichkeit, dem Aufruf der Datei unter Flash ein „?“ anzuhängen. Dadurch teilt man dem Browser mit, dass man Variablen mit an die Datei übergeben möchte. Dies unterlassen wir zwar, aber der Browser liest dadurch die TXT-Datei nicht mehr aus dem Cache, sondern lädt sie neu herunter:

```
loadVariables ("http://www.Server.de/Name.txt?", "instanzname");
```

Leider funktioniert dies nur teilweise, da es sein kann, dass der Besucher über einen Proxy-Server auf Ihre Internetseite zugreift. Auf dem Proxy-Server wird die TXT-Datei zwischengespeichert und man hätte wieder das Problem, dass die Datei aus einem Cache geladen wird. Man kann den Proxy-Server mit einer HTML-Meta-Angabe anweisen, die Dateien nicht zwischenzuspeichern.

```
<mate-http-equiv="pragma" content="no-cache">
```

Wenn es also um eine TXT-Datei mit dynamischem Inhalt geht, ist es unter Umständen unproblematischer, wenn Sie dieser stattdessen über eine serverseitige Scriptsprache wie CGI oder PHP die Variablen übergeben.

HTML-Dateien

HTML-Dateien können nun direkt in Flash abgebildet werden. Man kann sie über folgenden Befehl einladen:

```
loadVariablesNum ("text.html", 0);
```

Die eingelesenen Daten werden in einer Variable auf Level o gespeichert. Diese Variable muss in dem HTML-File vor dem Text stehen, z.B.: Text = (HTML-Text).

Die Darstellung des HTML-Textes erfolgt in einem entsprechenden Textfeld. Wichtig sind hierbei die Einstellungen: *HTML*, *Dynamischer Text* und *Mehrere Zeilen*.

Mehr Informationen zu Textfeldern finden Sie in Kapitel 4.2.
Die Farbe sowie die Größe und die Schriftart übernimmt Flash aus dem HTML-File. Auch Links zu anderen Seiten lassen sich so leicht erzeugen. E-Mail-Links mit *Mailto* sind auf diese Weise ebenfalls möglich. Ohne

Probleme kann ein Wort unterstrichen, kursiv oder fett geschrieben sein. Hierfür sind folgende HTML-Tags erlaubt:

`<a href=""> </a>`	Linkaufrufe
`<B> </B>`	fett
`<I> </I>`	kursiv
`<U> </U>`	unterstrichen
`< font color="#008000"> </font>`	Farbe
`<font face="Braggadocio"> </font>`	Schriftart
`<font size="+100"> </ font >`	Schriftgröße
`<p align='left'>`	linksbündig
`<p align='center'>`	mittig
`<p align='right'>`	rechtsbündig

Außerdem werden auch noch `LEFTMARGIN`, `RIGHTMARGIN`, `ALIGN`, `INDENT` und `LEADING` unterstützt.

Texte als Blocksatz auszurichten ist leider hiermit nicht möglich, genauso wenig kann man direkt Umlaute darstellen oder ein Wort durchstreichen. Sonderzeichen können aber über die URL-Encoding-Tabelle oder über den Unicode dargestellt werden.

Ein HTML-File mit den entsprechenden Modifikationen:

```
HTML=<font color="#FF0000">Dies hier ist der Text aus dem
HTML-File. Die <font color="#008000">Farbe</font> sowie die
<font size="+100">Groesse</font> und die <font face=
"Braggadocio">Schriftart</font> uebernimmt Flash aus dem
HTML-File. Auch Links lassen sich so leicht erzeugen :
<a href="http://www.DieFlasher.de"><u>Die Flasher</u></a>
<font color="#000000"> Ohne Probleme kann auch ein Wort
<u>unterstrichen</u>, <i>kursiv</i> oder <b>fett</b>
geschrieben sein. Texte rechtsbuendig oder als Blocksatz
auszurichten geht leider nicht, genauso wenig kann man ein
Ae, Ue oder Oe darstellen oder ein Wort durchstreichen. Die
Sonderzeichen aus der URL-Encoding-Tabelle kann man aber auch
hier anwenden.
<a href="mailto:e.mail@server.de"><u>E-mail-Links mit Mailto
sind auch moeglich.</u> </a>
</font>
</font>
```

5.2 Komponenten erstellen

Komponenten sind schon seit der letzten Flash-Version dabei und bieten uns die Möglichkeit häufig verwendete Elemente schnell wiederzuverwenden. Im folgenden Beispiel habe ich den Texteffekt, der bereits vorgestellt wurde, leicht modifiziert um eine Komponente zu erhalten.

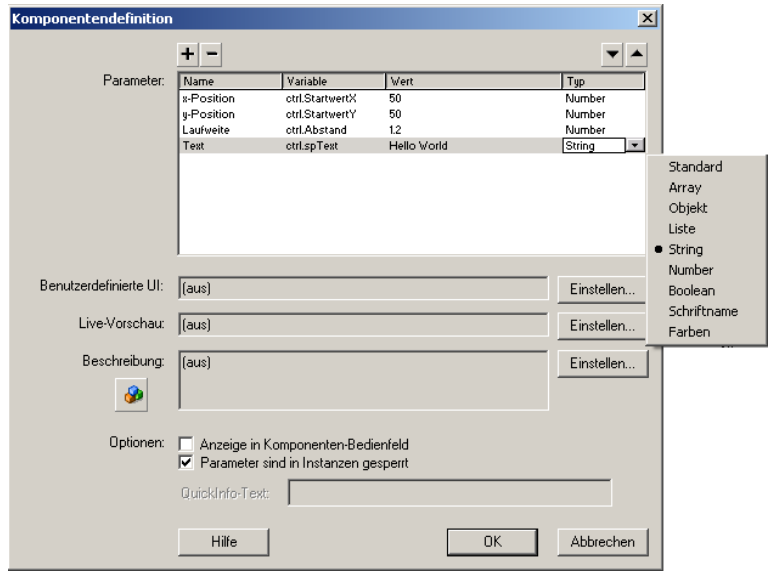
Referenzierungen

Zunächst müssen Sie darauf achten, dass alle Referenzen innerhalb des Filmes relativ sind. Im Beispiel des Texteffektes musste ich nachträglich alle _root-Verweise durch _parent ersetzen. Den Aufruf des Texteffektes habe ich mit in den Bereich „load“ innerhalb des Clips ctrl aufgenommen. Anschließend wurden alle notwendigen Elemente in eine Filminstanz verschoben, so dass ich in der Bibliothek ein neues Element erhalte, das Texteffekt heißt.

Parameter angeben

Folgend müssen die Parameter für diese Instanz angegeben werden. Hierzu wechselt man in die Bibliothek (Strg+L) und wählt dort den Film „Texteffekt“ an. Im Dialog der Bibliothek (oben rechts) finden Sie die Option Komponentendefinition... . Hier geben Sie die Referenzen direkt zu den Parametern in der Instanz „ctrl“ an.

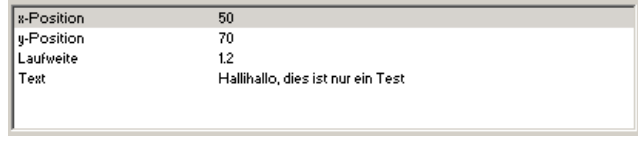
Abbildung 5.1: Komponentenerstellung im Dialog



Testen

Nun können Sie Ihre neue Komponente natürlich beliebig testen, der Dialog zur Änderung der Parameter sieht ja wie gewohnt aus.

Abbildung 5.2: Der Dialog der selbst erstellten Komponente



5.3 Shared Librarys

In Flash ist es möglich, Objekte aus anderen SWFs zu importieren. Wie das Exportieren und Importieren von Objekten funktioniert, wird hier erläutert.

In Flash gibt es nun die Möglichkeit, einzelne Elemente eines Films zu exportieren und so für andere Filme verfügbar zu machen. Dies bietet den Vorteil, immer wieder auf die gleichen Elemente zugreifen zu können, die der User schon eingeladen hat, und das Objekt bei Änderungen nur an einer Stelle aktualisieren zu müssen.

Um ein Objekt für andere Benutzer freizugeben, klicken Sie es in der Bibliothek mit der rechten Maustaste an und wählen den Menüpunkt Verknüpfung aus.

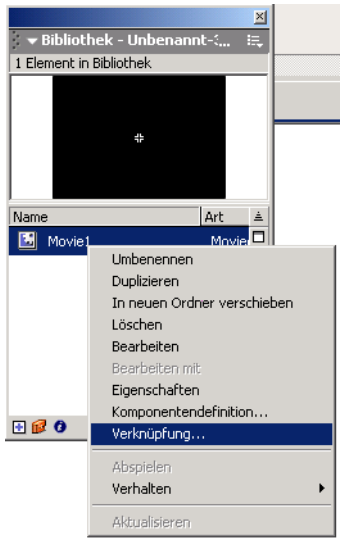


Abbildung 5.3: Popup-Menü eines Films

Wählen Sie in der Dialogbox „Export für gemeinsame Nutzung zur Laufzeit“ und geben Sie bei „Bezeichner“ einen Namen ein.

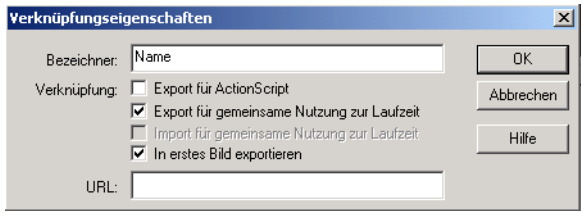
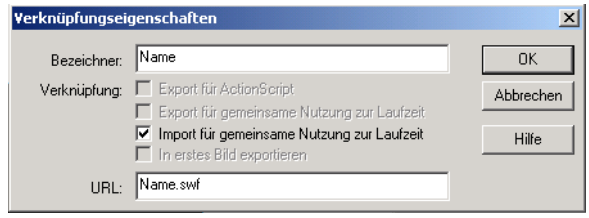


Abbildung 5.4: Verknüpfungseigenschaften zum Exportieren

Über diesen Weg können Sie in einem Flashmovie die Bibliothek (Library) eines anderen Movies über Datei › Als Bibliothek öffnen öffnen und per Drag&Drop das entsprechende Objekt in die Bibliothek importieren.

In der Bibliothek, in die das Objekt importiert wurde, werden die Verknüpfungsparameter automatisch eingestellt.

Abbildung 5.5: Verknüpfungseigenschaften zum Importieren



Dies setzt voraus, dass beide SWFs im gleichen Verzeichnis liegen. Wenn Sie die Files später auf einen Server legen, ist es günstiger, direkt mit absoluten URLs zu arbeiten.

Wenn Sie auf der Festplatte testen, benutzen Sie relative Pfadangaben, wobei Sie aber darauf achten sollten, in den Verzeichnisnamen keine Sonderzeichen oder Überlängen zu verwenden.

Der große Nachteil bei diesem Import liegt darin, dass man gar keine Kontrolle hat, ob das File schon eingeladen ist oder nicht.

Zwar kann man in Flash MX über das Deaktivieren des Kontrollkästchens „In erstes Bild exportieren“ dafür sorgen, dass ein File nicht direkt zu Beginn „unkontrolliert“ eingeladen wird, wirklich gut wird die Lösung dadurch aber auch nicht!

Flash MX bietet mit dem sehr flexiblen Befehl `loadMovie` an, SWF, JPG und MP3 direkt einzuladen. Es empfiehlt sich daher die Nutzung dieses Befehls!

Ein Praxisbeispiel dazu, direkt mit einem Preloader verbunden, finden Sie in Kapitel 4.4 im Abschnitt „Objektorientierter Preloader für mehrere SWFs“ ab Seite 193.

5.4 Flash Printing

Mit dem Einzug der Serveranbindungsmöglichkeiten in Flash sind auch Funktionen wie das Drucken von Inhalten notwendig geworden.

Druckbare Seiten festlegen

Wählen Sie ein Schlüsselbild an und geben Sie dort als Bildmarkierung im Dialog Eigenschaften ein #p an. Nun weiß Flash, dass dies gedruckt werden kann.

Druckbaren Bereich festlegen

Wollen Sie den druckbaren Bereich genauer festlegen, wenn Sie zum Beispiel nur einen Ausschnitt drucken wollen, so legen Sie zunächst ein weiteres Schlüsselbild an. Dieses enthält dann ein Rechteck, welches genau diesen Bereich, der druckbar sein soll, umreißt. Die Bildmarkierung dieses Bildes bezeichnen Sie nun mit #b. Bereiche können pro Zeitleiste nur einmal festgelegt werden. Wollen Sie nicht jedes Mal den Bereich selbst festlegen, so können Sie bei der Aktion Print anstelle des selbst definierten druckbaren Bereiches auch folgende Optionen angeben:

```
"bmax"
```

Hier werden alle Objekte des angegebenen Films mit maximaler Größe auf das Blatt Papier angepasst.

```
"bmovie"
```

Hier wird die Hauptbühne als Referenzgröße genommen, alle Objekte erscheinen in entsprechender Größe auf dem Stück Papier.

```
"bframe"
```

Hier wird der Größenbezug zu dem Frame gebildet, dementsprechend ist die Ausrichtung.

Daten rastern

Wollen Sie Effekte, die mit Hilfe von Alphawerten oder Einfärbungen erreicht wurden, ausdrucken, so müssen diese vorher gerastert bzw. „als Bitmap“ gedruckt werden. In Flash MX ist diese Option im Standard ausgeschaltet, da in den meisten Fällen Texte ausgedruckt werden und dies dort nicht notwendig ist.

Print anwenden

Dies ist zunächst das einfache Ausdrucken einer Ebene. Der Hauptfilm liegt im Level 0; sollten Sie einmal per loadMovie einen Film in die Ebene 0 laden, werden Sie feststellen, dass Sie Ihren Hauptfilm verlassen.

```
printNum (level)
```

Optional können Sie den druckbaren Bereich noch mit angeben, also einfach die Instanz, die Sie mit #b bezeichnet haben.

```
printNum (level, "druckbarerBereich")
```

Anstelle eines Levels kann man auch direkt den MovieClip angeben, sofern dieser entsprechend bezeichnet ist.

```
print ("MovieClip")
```

Auch hier ist der druckbare Bereich optional, wollen Sie es z.B. dem Anwender ersparen, jedes Mal Ihr aufwändiges Logo mitzudrucken.

```
print ("MovieClip", "druckbarerBereich")
```

Im Folgenden sind noch mal alle Varianten aufgelistet mit der Ergänzung „AsBitmap“, das heißt dass die Daten gerastert an den Drucker gesendet werden. Dies kann man ggf. auch an der Datenmenge im Spool-Verzeichnis des Druckers nachvollziehen.

```
printAsBitmapNum (level)

printAsBitmapNum (level, "druckbarerBereich")

printAsBitmap ("MovieClip")

printAsBitmap ("MovieClip", "druckbarerBereich")
```

Auch wenn „AsBitmap“ qualitative Einbußen hat, so ist es die einzige Möglichkeit alle Effekte und somit ein genaues Abbild der Bildschirmdarstellung zu erhalten.

5.5 Soundobjekt

Jedes Mal, wenn man einen Sound per ActionScript abspielen möchte, muss man das Soundobjekt benutzen. Nachdem die wichtigsten Grundkenntnisse schon im Kapitel 4.5 ab Seite 200 erläutert wurden, kommen wir nun direkt auf das Wesentliche des Soundobjektes.

Um das Soundobjekt überhaupt benutzen zu können, müssen Sie das entsprechende Sample vorab diesem Soundobjekt zuweisen. Dies geht in Flash MX auf zwei Arten. Entweder Sie laden ein MP3 per `loadMovie`, wie in Kapitel 4.4 im Abschnitt „Objektorientierter Preloader für mehrere SWFs“ ab Seite 193, ein oder Sie benutzen den Befehl `attachSound`, auf den wir nachfolgend auch eingehen werde. Um `attachSound` anwenden zu können müssen Sie einem Sample einen Verknüpfungsnamen geben.

- Importieren Sie ein Sample in Flash.
- Öffnen Sie die Bibliothek Ihres Filmes (Strg+L).
- Klicken Sie mit der rechten Maustaste auf das Sample in der Bibliothek Ihres Filmes und wählen Sie anschließend die Option Verknüpfung.
- Wählen Sie den Menüpunkt Verknüpfung › Export für ActionScript aus.
- Bei Bezeichner geben Sie den Namen, den die Verknüpfung haben soll, ein.
- Bestätigen Sie mit OK.

In diesem Beispiel wurde dadurch dem Sample der Verknüpfungsname *rotor* zugewiesen.

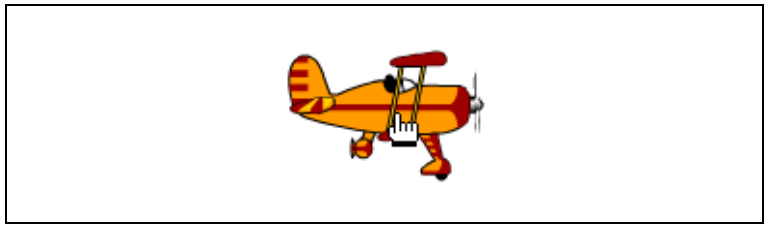


Abbildung 5.6: Layout des Beispiels, ein Flugzeug kann per Drag&Drop verschoben werden.

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel5/oo_flugz.fla*.

Um den Sound zu starten, erstellen Sie zunächst ein Soundobjekt. Dieses Soundobjekt wird wie alle anderen Objekte auch mit `new name` erstellt. Da das Objekt am besten direkt zu Beginn hinzugefügt wird, macht man dies über `onClipevent(load)`.

Listing 5.1: Start eines Samples

```
onClipEvent (load) {
    _root.rotorgeraeusch = new Sound();
    _root.rotorgeraeusch.attachSound("rotor");
    _root.rotorgeraeusch.start(0.2,999);
}
```

`onClipevent(load)` besagt, dass der folgende Befehl einmal ausgeführt wird – und zwar dann, wenn der MovieClip zu Beginn eingeladen wird.

Der erste Befehl erzeugt ein Soundobjekt mit dem Namen *rotorgeraeusch*, welches sich in der Hauptebene, also in *_root*, befindet. Der zweite Befehl übergibt das Sample mit dem Verknüpfungsnamen *rotor* an das Soundobjekt. Durch den Befehl `start` wird der Sound gestartet, die zwei Parameter besagen, dass dieser erst nach 0.2 Sekunden gestartet und *999*-mal wiederholt werden soll.

In dem Clipevent `enterFrame` setzen wir die Pan (Balance)- und Lautstärkeneinstellungen entsprechend der Position des Flugzeuges. Über `setVolume` wird die Lautstärke gesetzt, welche einen Wert zwischen 0-100 annehmen kann, und über `setPan` wird die Balance gesetzt, welche von -100 (links) bis +100 (rechts) geht.

Listing 5.2: Zuweisung von Pan und Volumen und Skalierung des Flugzeuges

```
onClipEvent (enterFrame) {
    _root.rotorgeraeusch.setVolume(100+_root.flaeche.flugzeug._y);
    _root.rotorgeraeusch.setPan(_root.flaeche.flugzeug._x);
    _root.flaeche.flugzeug._xscale=100+_root.flaeche.flugzeug._y;
    _root.flaeche.flugzeug._yscale=100+_root.flaeche.flugzeug._y;
}
```

Dieses ActionScript sorgt dafür, dass sich bei Bewegung des Flugzeuges der Sound mit ändert und das Flugzeug auf der y-Achse nach oben hin kleiner wird.

Auf dem Button des Flugzeuges befindet sich folgendes ActionScript um das Flugzeug per Drag&Drop verschieben zu können.

Listing 5.3: Drag&Drop eines Movies

```
on (press) {
    startDrag(this, true, -100, -90, 100, -10);
}
on (release, releaseOutside) {
    stopDrag();
}
```

`startDrag` sorgt dafür, dass das Movie, aus dem der Befehl aufgerufen wird (da `this` benutzt wurde), an die Maus geheftet wird. Als zweiten Parameter benötigt `startDrag` einen Booleanwert, wenn dieser true ist, wird das gedraggte Movie unter der Maus zentriert, bei false nicht. Wir haben uns hier also für das zentrierte „Draggen“ entschieden. Die letzten vier Parameter geben das Maximum der Fläche an, auf der das Flugzeug bewegt werden darf.

Beim Loslassen der Maustaste soll der Drag natürlich wieder enden, deshalb `on release` bzw. `onreleaseOutside stopDrag`.

duration, position und onSoundComplete()

In Flash MX besitzt das Soundobjekt zwei neue Eigenschaften: `duration` und `position`. Mit `duration` kann man die Länge eines Sounds abfragen, was es ermöglicht, eine Art Sequenzer für beliebige Soundsamples zu bauen, ohne jeweils die Soundlänge per Hand eintragen zu müssen, und `position` gibt die aktuelle Soundposition an; wenn man z.B. ein Sample nur stoppen möchte, kann man so die Soundposition zuerst abfragen, in einer Variable zwischenspeichern, und wenn das Soundfile wieder gestartet werden soll, ab der Position weiter spielen.

Wir haben für Sie einmal ein kleines Beispiel für die Pause-Funktion gebaut.

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel5/o1_pause.fla*.

Das Beispiel besitzt nur einen Schalter, auf Knopfdruck wird der Sound gestartet. Durch nochmaliges Drücken wird dieser wieder gestoppt. Auf den dritten Knopfdruck hin wird die Wiedergabe wieder eingeschaltet, von der Position aus, wo der Sound gestoppt wurde. Bei dem gewählten Musiksample handelt es sich übrigens um die Hintergrundmusik unserer Homepage *www.DieFlasher.de*.

Zuerst wird in der Hauptzeitleiste ein Soundobjekt gebildet.

Listing 5.4: Erzeugt ein Soundobjekt und weist diesem das Sample DieFlasher.de zu

```
_root.stdpop = new Sound();
_root.stdpop.attachSound("DieFlasher.de");
```

Das ActionScript des Buttons sieht wie folgt aus.

```
on (release) {
    if (_root.stdpop.position != 0 && !_root.pause) {
        _root.stdpop.stop();
        _root.pause = true;
    } else {
        _root.stdpop.start(int(_root.stdpop.position/1000),1);
        _root.pause = false;
    }

}
```

Beim Klick des Schalters wird zuerst festgestellt, ob der Pausemodus nicht aktiv ist und das Lied sich nicht am Anfang befindet. Falls dies nicht zutrifft, wird das Lied gestartet, anderenfalls gestoppt. Um jeweils immer den richtigen Zustand zu finden wurde die Variable `pause` mit eingeführt, welche einen Booleanwert enthält.

Wie man sieht wird durch das Script das Lied immer nur einmal ausgeführt. Dies lässt sich auch so direkt nicht anders regeln, da, wenn man in dem Startbefehl eine 999 für die Anzahl eingeben würde, das Lied immer wieder von der letzten Pause-Position abgespielt würde.

Damit das Lied trotzdem immer wieder von vorn abgespielt wird, kann man aber die neue CallBack-Funktion `onSoundComplete()` benutzen. Diese wird automatisch von Flash aufgerufen, wenn das Abspielen des Musikstückes beendet wurde. Wenn man also in dem Musikstück direkt angibt, dass dieses wieder von vorne gestartet werden soll, wird dies, sobald das Lied endet, ausgeführt.

```
_root.stdpop.onSoundComplete = function() {
    _root.stdpop.start(0,1);
    _root.pause = false;
};
```

Listing 5.5: Ausnutzung der CallBack-Funktion, um ein Clip im Loop laufen zu lassen

6 Flash MX in Interaktion

6.1 Grundüberlegungen

Flash bietet viele schöne Möglichkeiten Interaktionen anregend zu gestalten, natürlich ist es dann auch interessant zu erfahren, wie man externe Daten in Flash einlesen kann, um diese dort darzustellen. Möchte man auf Informationen aus einer Datenbank z.B. zurückgreifen, so geht es nicht darum, einzelne Werte, sondern ganze Datensätze einzulesen. Wie man dieses Problem elegant lösen kann, werden Sie in den folgenden Teilen erfahren.

6.2 Datenaustausch

Flash kann Daten aktiv nur über URL-Aufrufe verschicken, wobei man per VB-Script auch auf Variablen des Flashfilms zugreifen kann. Ausgehend von Flash selbst ist dies aber erstmal nebensächlich.

Der Datenaustausch hat sich so, wie man ihn im folgenden Schema dargestellt sieht, bewährt.

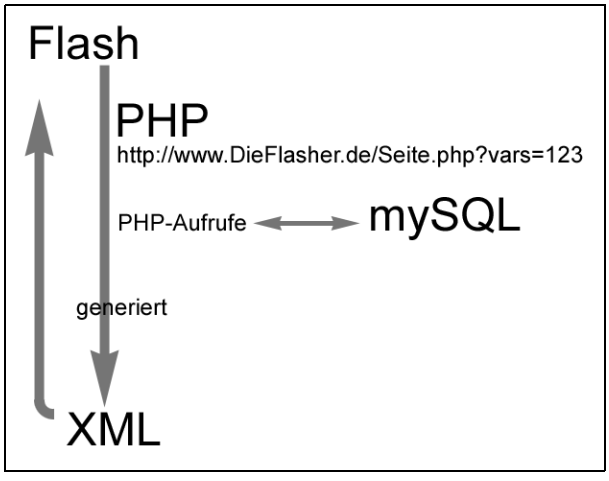


Abbildung 6.1: Datenaustausch von Flash mit einer Datenbank (Schema)

Falls Sie schon mal mit Datenbanken gearbeitet haben sollten, werden Ihnen Programme wie Query Analyser vertraut sein, sie ermöglichen die direkte Eingabe von SQL-Befehlen (Datenbankabfragen) und liefern eine Ergebnistabelle zurück. Ein solches Programm wird hier später vorgestellt, doch zunächst sollen Sie erfahren, wie man XML-Daten in Flash einliest. Hierzu soll ein kleines Beispiel eines einfach editierbaren Menüs genügen.

6.3 Flash und XML

Pflegt man eine Seite, die Flashelemente enthält, so werden es wahrscheinlich schon viele von Ihnen so empfunden haben, dass die Inhalte nicht immer einfach zu pflegen sind, da jedes Mal Teile in Flash geöffnet und generiert werden müssen, auch wenn sich nur eine paar Links ändern. Um diese Pflege für eine Navigation zu vereinfachen kann man alle relevanten Inhalte einfach in eine XML-Struktur überführen. Dies sieht in unserem Beispiel dann so aus.

```
<links>
    <site displayname="Home" inhalt="home.htm"
header="header/hd_home.htm" icon="icon/home.swf"></site>
    <site displayname="Forum" inhalt="forum/index.php?viewcat=1"
header="header/hd_forum.htm" icon="icon/forum.swf"></site>
    <site displayname="Galerie" inhalt="galerie.htm"
header="header/hd_galerie.htm" icon="icon/galerie.swf"></site>
    <site displayname="News" inhalt="news.htm"
header="header/hd_news.htm" icon="icon/news.swf"></site>
    <site displayname="Biografie" inhalt="bio.htm"
header="header/hd_bio.htm" icon="icon/bio.swf"></site>
    <site displayname="Discografie" inhalt="disco.htm"
header="header/hd_disco.htm" icon="icon/disco.swf"></site>
    <site displayname="Trading" inhalt="trade.htm"
header="header/hd_trade.htm" icon="icon/trading.swf"></site>
    <site displayname="Downloads" inhalt="down.htm"
header="header/hd_down.htm" icon="icon/down.swf"></site>
    <site displayname="Gaestebuch" inhalt="gaeste.htm"
header="header/hd_gaeste.htm" icon="icon/gaeste.swf"></site>
    <site displayname="Links" inhalt="links.htm"
header="header/hd_links.htm" icon="icon/links.swf"></site>
</links>
```

Es liegen verschiedene Datensätze vor, die jeweils die Attribute „displayname“, „inhalt“, „header“ und „icon“ beinhalten. Das folgende Bild zeigt das spätere Ergebnis, „displayname“ entspricht hier dem Untertitel des Bildes, „inhalt“ und „header“ sind jeweils die Links, nach dem zwei verschiedene Frames geupdated werden sollen, und „icon“ gibt das entsprechende „.swf“ an, welches das Icon enthält.

Abbildung 6.2: Fertige Navigation

Die oben dargestellte XML-Datei ist natürlich nur exemplarisch und muss für Ihre Bedürfnisse angepasst werden. Diese Datei ist im Beispiel „menu.xml“. Auf der Hauptbühne befinden sich 4 Objekte, eine Filminstanz, die alles kontrolliert, eine Instanz des Navigationsknopfes und natürlich die beiden „skip“-Knöpfe.

Das ActionScript der beiden Knöpfe ist recht einfach, „nach-links“ beinhaltet

```
on(press){
    if (!_root["ctrl"].fwd && !_root["ctrl"].rwd) {
        _root["ctrl"].zurueck();
    }
}
```

und das ActionScript des „nach-rechts“ enthält

```
on(press){
    if (!_root["ctrl"].fwd && !_root["ctrl"].rwd) {
        _root["ctrl"].vor();
    }
}
```

Diese Aufrufe referenzieren ActionScript aus der Kontrollinstanz („ctrl“ benannt), dessen ActionScript wie folgt aussieht.

```
onClipEvent(load){
    i = 0;
    step=10;
    offset = 275;
    l_ende = offset;
    _root.myXML = new XML();
    _root.myXML.load("menu.xml");
    fwd = flase;
    rew = false;
    buttonwidth=110;
}
```

Gleich zu Beginn müssen alle Variablen initialisiert werden. Deren Funktion werden Sie im Detail dem Skript entnehmen können. Wichtig ist an dieser Stelle, dass die Datei „menu.xml“ in ein intern angelegtes XML-Objekt geladen wird.

```
onClipEvent(enterFrame){
    if (_root.myXML.loaded && i < 1){
        i = 1;
```

Diese Bedingung fragt ab, ob die XML-Datei schon vollständig eingeladen wurde, und verhindert gleichzeitig durch den Zähler in „i“, dass jeder Menüpunkt öfter als einmal eingeladen wird.

```
    k=1;
while(!_root.myXML.firstChild.childNodes[i].nextSibling ==
    false){
        _root.display.duplicateMovieClip("button" + k,k);
        _root["button" + k ].displtxt =
            _root.myXML.firstChild.childNodes[i].
            attributes.displayname;
        _root["button" + k ].icon.loadMovie("" +
            _root.myXML.firstChild.childNodes[i].
            attributes.icon);
        _root["button" + k ].inhaltURL =
            _root.myXML.firstChild.childNodes[i].
            attributes.inhalt;
        _root["button" + k ].headerURL =
            _root.myXML.firstChild.childNodes[i].
            attributes.header;
        k++;
        i=i+2;
```

Eine Kuriosität des XML in Flash ist die Vergabe der Indizes der childNodes. Wie auch das HTML, so hat das XML eine baumartige Struktur, die erste Ebene besteht nur aus dem „Ast“ „links“, der wiederum enthält weitere Verzweigungen an „site“. Mit firstChild zeige ich auf den ersten und einzigen Ast in unserem Fall („links“), und dessen childNodes, also untergeordneten Äste, enthalten die Daten, die wir laden wollten (site).

```
    }
    r_ende = offset - buttonwidth * (k-2);
    _root.display._visible=0;
}
```

Im Folgenden werden alle Navigationspunkte positioniert, da die Übergänge fließend sein sollen. Wenn man vom einen zum anderen Punkt wechselt, werden die Positionen laufend für alle Punkte berechnet. Sind Punkte zu weit links oder rechts, so werden diese ausgeblendet, bzw. auch skaliert. Somit erhält man einen netten dreidimensionalen Effekt.

```
for (m=1;m<=k;m++){
    _root["button" + m ]._x = offset + buttonwidth * (m-1);
    _root["button" + m ]._y = 50;
    scale = 100 - Math.abs(_root["button" + m ]._x-275)/3;
    _root["button" + m ]._xscale=scale;
    _root["button" + m ]._yscale=scale;
    if (scale<0) {
        _root["button" + m ]._visible=0;
    } else {
        _root["button" + m ]._visible=1;
    }
}
```

Hier werden Status gesetzt bzw. abgefragt um zu überwachen, ob noch Punkte in Bewegung sind oder wann diese am Ziel sind.

```
    if (fwd && offset>zieloffset){
        offset=offset-step;
    } else if(fwd){
        offset = zieloffset;
        fwd = false;
    }

    if (rwd && offset<zieloffset){
        offset=offset+step;
    } else if (rwd){
        offset = zieloffset;
        rwd = false;
    }

    function vor(){
        if ((offset-buttonwidth)>=r_ende) {
            fwd=true;
            zieloffset = offset-buttonwidth;
        }
    }

    function zurueck(){
        if ((offset+buttonwidth)<=l_ende) {
            rwd=true;
            zieloffset = offset+buttonwidth;
        }
    }
}
```

Um die Befüllung der Buttons zu verstehen werde ich im Folgenden noch auf die Struktur der Punkte eingehen.

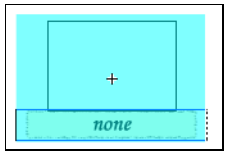


Abbildung 6.3:
Ein Navigationspunkt in der Entwicklungsumgebung

Um die Links setzen zu können enthält die Schaltfläche folgendes Script.

```
on(press){
    getURL(inhaltURL,"inhalt");
    getURL(headerURL,"header");
}
```

Da die Navigation die URLs in die entsprechenden Frames „inhalt“ und „header“ lädt, muss dies für ihre Verwendung noch geändert werden, wenn Sie ein anderes Frameset benutzen.

6.4 Flash und SQL

Im Folgenden werden Sie erfahren, wie man Datenbankabfragen in recht allgemeiner Form durchführen kann. Ein Query Analyser ist hier das Klassenziel – hierbei handelt es sich um eine Oberfläche, die SQL-Befehle an die Datenbank weitergibt und das Ergebnis anzeigt.

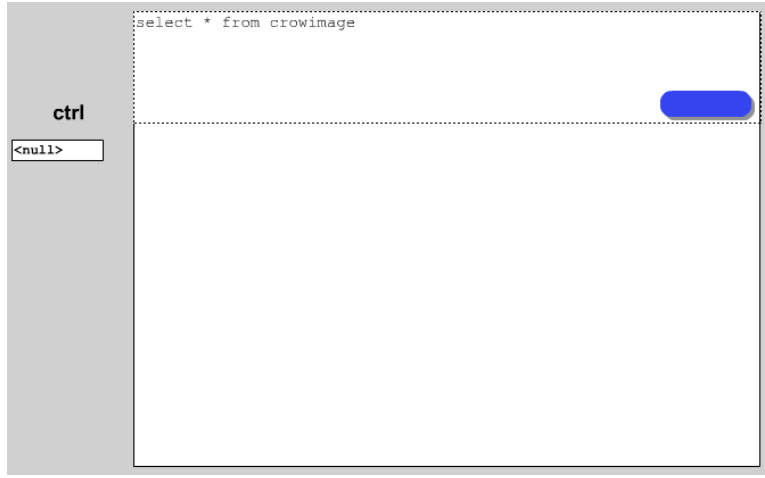


Abbildung 6.4: Der Query Analyser in der Entwicklungsumgebung

Die Kontrollinstanz enthält folgendes ActionScript, hier werden alle wesentlichen Aktionen zur Ausgabe der Daten gesteuert.

```
onClipEvent(load){
i=0;
function doTable(rows, cols) {
    c=0;
    i=0;
    while (c<cols){
        r=0;
        while (r<rows){
            i++;
            _root["cell"].duplicateMovieClip("cell" + r + "_"
                + c,i);
            _root["cell" + r + "_" + c]._x = 80*c;
            _root["cell" + r + "_" + c]._y = 100 + 18*r;
            r++;
        }
        c++;
    }
}

function ladeXML(sSQL){
    _root.myXML = new XML();
    _root.myXML.load("myXMLload.php?sql=" + sSQL);
```

```
    }
}

onClipEvent(enterFrame){
    if (_root.myXML.loaded && i < 1){
        i = 1;
        k=1;
        while(!_root.myXML.firstChild.childNodes[i].nextSibling
            == false){
            if (i==1) {
                linkURL =
                    _root.myXML.firstChild.attributes.link;
                linkTarget =
                    _root.myXML.firstChild.attributes.target;
            }
            l=0;
            while(_root.myXML.firstChild.childNodes[i]
                .attributes[l]) {
                l++;
            }
            k++;
            i=i+2;
        }
    }
}
```

Die Funktion zum Laden der Daten werden Sie wahrscheinlich wiedererkennen, neu sind die Funktionen, die zum einen die Tabelle und zum anderen die XML-Datei erzeugen.

Um die XML-Datei zu erzeugen wird hier ein kurzes PHP-Skript vorgestellt, das auf eine mySQL-Datenbank passt. Da diese genauso wie PHP frei verfügbar sind, schien dies angebracht.

```
<liste>
<?php
    $link = mysql_connect("localhost", "DATENBANKNAME",
        "DATENBANKPASSWORT")
        or die ("Could not connect");
    mysql_select_db ("DATENBANKNAME")
        or die ("Could not select database");

    if (isset($sql)) {
    $query = $sql;

    $result = mysql_query ($query)
        or die ("<row attr=\"failed\"></row>");

while($line = mysql_fetch_row($result)){
    print "<row ";
    $i = 1;
```

```
        while(list($col_name, $col_value) = each($line)){
            print " attr" . $i . "=\"$col_value\"";
            $i = $i + 1;
        }
        print " len=\"$i\"></row>\n";
    }

    }
        mysql_close($link);
    ?>
    </liste>
```

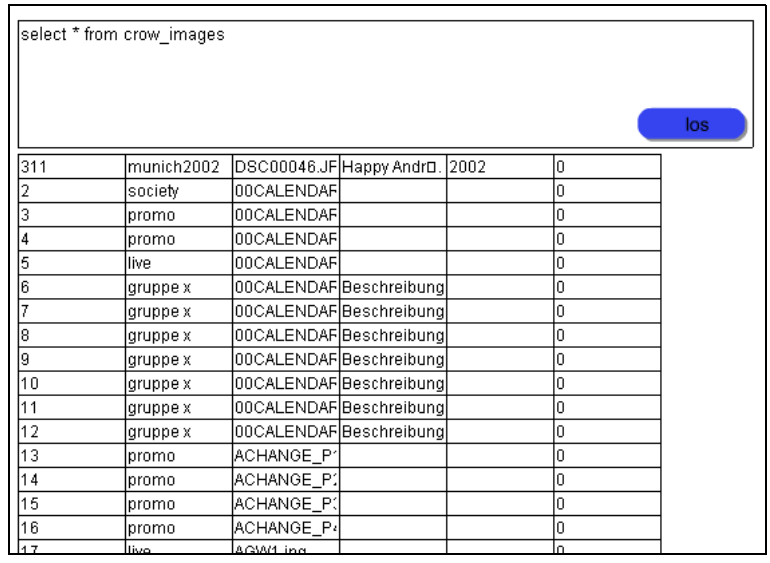
Dies erzeugt eine recht allgemein gehaltene XML-Datei, die von der Funktion `doTable()` ausgewertet wird. Der Aufruf der PHP-Datei wird einfach als Link auf den Button gelegt, der folgendes ActionScript enthält.

```
on (press){
    _root["ctrl"].ladeXML(_root.sqlquery);
}
```

Zu beachten ist, dass „sqlquery“ unter „var:“ bei dem Eingabetextfeld einzutragen ist.

Das Ergebnis sieht dann u.U. wie folgt aus:

Abbildung 6.5: Das Ergebnis einer Datenbankabfrage



311	munich2002	DSC00046.JF	Happy Andr□.	2002	0
2	society	00CALENDAF			0
3	promo	00CALENDAF			0
4	promo	00CALENDAF			0
5	live	00CALENDAF			0
6	gruppe x	00CALENDAF	Beschreibung		0
7	gruppe x	00CALENDAF	Beschreibung		0
8	gruppe x	00CALENDAF	Beschreibung		0
9	gruppe x	00CALENDAF	Beschreibung		0
10	gruppe x	00CALENDAF	Beschreibung		0
11	gruppe x	00CALENDAF	Beschreibung		0
12	gruppe x	00CALENDAF	Beschreibung		0
13	promo	ACHANGE_P			0
14	promo	ACHANGE_P			0
15	promo	ACHANGE_P			0
16	promo	ACHANGE_P			0
17	live	AGW1.jpg			0

7

Die dritte Dimension

7.1 Grundüberlegungen

Um den Eindruck einer dreidimensionalen Umgebung zu erzeugen muss zuerst ein Regelwerk von Verhaltensweisen erstellt und die Frage geklärt werden: Was verleiht einer zweidimensionalen Darstellung eine dritte Dimension?

Als ich mich mit der Fragestellung konfrontiert sah, musste ich unweigerlich an meinen Kunstunterricht damals in der Schule denken, dort mussten damals Fluchtpunktzeichnungen angefertigt werden. Wenn man diese in einem Foto entsprechend nachzeichnet, kann ich das, was ich damals erlernte, noch mal veranschaulichen. Der Punkt, in dem die „Gassen“ am Horizont zusammenlaufen, nennt sich Fluchtpunkt, die Linien, an denen man die Fluchten verfolgen kann, heißen Fluchtlinien.

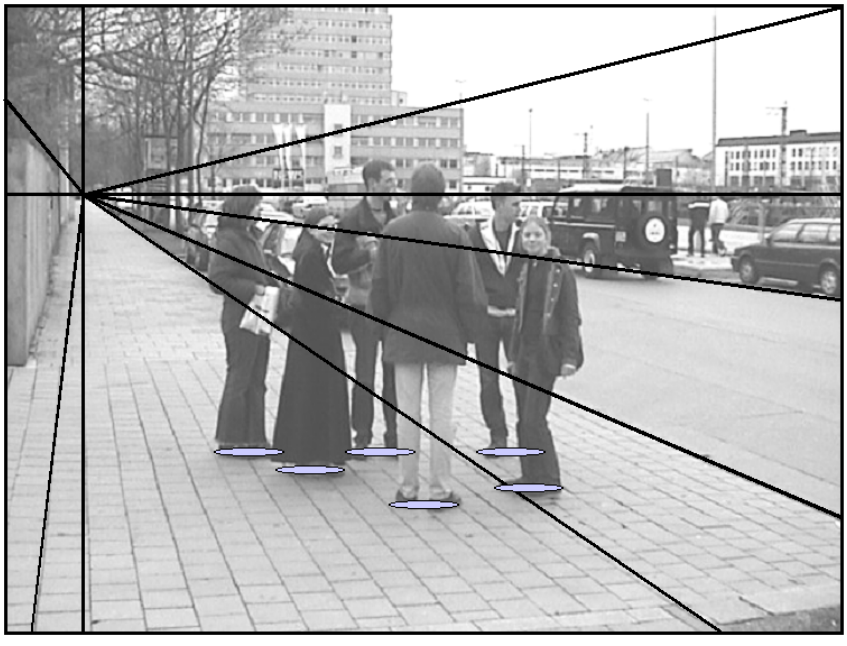
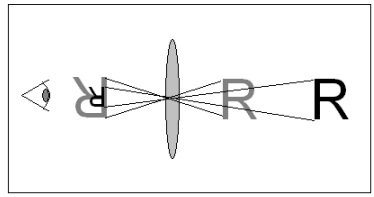


Abbildung 7.1: Ein Foto mit Fluchtlinien.

Anhand dieser Linien kann man die Größenverhältnisse, die Perspektive nachvollziehen. Es ist offensichtlich, dass Größe und Entfernung aneinander gekoppelt sind. Doch nun folgt die nächste Herausforderung. Wenn man Perspektive erzeugen will, muss man auch das Verhältnis Abstand – Größenrelation kennen. Einige von Ihnen erinnern sich vielleicht noch an den Physikunterricht und einige wenige haben vielleicht auch schon mal so etwas wie eine Lochkamera gebaut, ein Schema des Aufbaus mit Linse bringt in dieser Frage die entscheidende Einsicht.

Abbildung 7.2: Entfernungs- und Größenkorellation



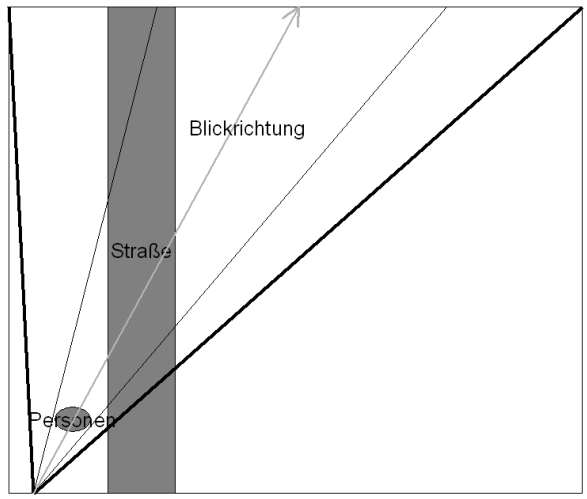
Wie hier zu erkennen ist, ist die Größenrelation antiproportional zur Entfernung, oder um es in einfachere Worte zu fassen: Ist ein Objekt doppelt so weit entfernt wie ein anderes gleicher Größe, so misst dies für den Betrachter auch nur noch entsprechend die halbe Höhe/Breite.

Dies erklärt auch Effekte wie z. B.die Tatsache, dass man ein Flugzeug mit ggf. 800 km/h vom Boden aus ohne Probleme beobachten kann, während ein Auto mit 80 km/h Geschwindigkeit vom Straßenrand aus nicht viel Zeit für eingehendere Betrachtung lässt.

All diese Annahmen ermöglichen uns zu verstehen, wie man ein 3D-Bild errechnet, doch wenn eine „Landschaft" vorliegt, muss zuerst geklärt werden: Was soll sichtbar sein, was liegt in meinem Blickfeld, welche Objekte verdecken einander?

Hier bietet es sich an, das Geschehen aus der Vogelperspektive, also von oben, zu betrachten.

Abbildung 7.3: Die schematische Darstellung des Fotos aus der Vogelperspektive, oder auch „Draufsicht".

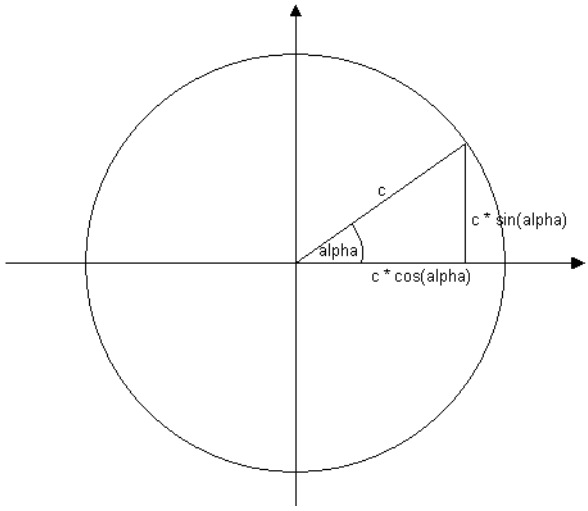


Nun kann man erkennen, dass die Positionierung auf der Bühne von drei Faktoren abhängig ist: Entfernung, Winkel zur Blickrichtung und Blickwinkel (Sichtfeld). Um die Winkel zu erhalten muss man Dreiecke bilden, in denen man am besten alle drei Seiten kennt. Die horizontalen und vertikalen Abstände lassen sich noch verhältnismäßig leicht errechnen, da die eigene Position und die Objektposition ja in Form von Koordinatenpaaren vorliegen und man nur die Differenz errechnen muss. Die Entfernung ist mit Hilfe von Pythagoras, einem alten Griechen aus einer Zeit vor Flash, recht einfach zu berechnen. Da die Koordinaten im rechten Winkel zueinander stehen, kann man einfach die Wurzel aus der Summe der Quadrate der beiden Katheten (die kurzen Seiten in einem Dreieck mit rechtem Winkel) errechnen.

Es gilt also:

$$Entfernung = \sqrt{xEntfernung^2 + zEntfernung^2}$$

Um den Winkel zu errechnen muss man wieder ein paar grundsätzliche Dinge kennen, wie z.B. den Einheitskreis.



$$c \cdot \sin(alpha) = zEntfernung$$
$$c \cdot \cos(alpha) = xEntfernung$$
$$\Rightarrow$$
$$\frac{zEntfernung}{c} = \sin(alpha)$$
$$\frac{xEntfernung}{c} = \cos(alpha)$$
$$\Rightarrow$$
$$alpha = \sin^{-1}\left(\frac{zEntfernung}{c}\right)$$
$$alpha = \cos^{-1}\left(\frac{xEntfernung}{c}\right)$$

Abbildung 7.4: Die Darstellung der Winkel- und Streckenverhältnisse – hat „c“ die Länge 1, spricht man vom Einheitskreis.

Flash selbst rechnet im Bogenmaß, das bedeutet, dass eine andere Aufteilung vorgenommen wird. Während im Gradmaß ein Kreisumfang in 360° aufgeteilt wird, ist im Bogenmaß der Kreisumfang des sog. Einheitskreises entscheidend, also 2 * PI. Die Umrechnung gestaltet sich entsprechend einfach:

$$xGradma\beta \cdot \frac{\pi}{180} = xBogenma\beta$$

Auch wenn es scheint, man könne nun problemlos alle Winkel zwischen den Objekten berechnen, so ist das aufgrund eines misslichen Nebeneffekts der Umkehrfunktion der Sinusfunktion nicht ganz so einfach. Die Umkehrfunktion liefert nur Werte zwischen 0° und 180° zurück, so dass Objekte, die hinter dem Betrachtungspunkt liegen, als davor liegend ausgegeben werden. Hier ist die einzig praktikable Lösung in der Vektorrechnung zu finden. Kreuzt man zwei Vektoren im Dreidimensionalen, so ist der Ergebnisvektor jener, der senkrecht (in den Raum) auf beiden steht, sofern diese nicht parallel verlaufen. Verläuft ein Vektor rechts gerichtet zum anderen, so ist das Ergebnis immer positiv, der Vektor zeigt, bildlich gesprochen, nach oben. Zeigt der andere Vektor nach links, so ist das Ergebnis immer negativ. Das Kreuzprodukt eines Vektors errechnet sich wie folgt:

$$A \times B = \begin{Bmatrix} a_1 \\ b_1 \\ c_1 \end{Bmatrix} \times \begin{Bmatrix} a_2 \\ b_2 \\ c_2 \end{Bmatrix} = \begin{Bmatrix} b_1 \cdot c_2 - c_1 \cdot b_2 \\ c_1 \cdot a_2 - a_1 \cdot c_2 \\ a_1 \cdot b_2 - b_1 \cdot a_2 \end{Bmatrix}$$

Da wir uns nur dafür interessieren, ob der Vektor zum Objekt vor oder hinter uns liegt, müssen wir nur die letzte Zeile auswerten, wenn gilt

$$A = \begin{Bmatrix} -\sin\left((meineBlickrichtung + 90) \cdot \frac{\pi}{180}\right) \\ \cos\left((meineBlickrichtung + 90) \cdot \frac{\pi}{180}\right) \\ 0 \end{Bmatrix}$$

$$B = \begin{Bmatrix} xEntfernung \\ zEntfernung \\ 0 \end{Bmatrix}$$

Nun haben wir alle notwendigen Informationen um ein einfaches Beispiel herzustellen.

▶ Der Wald

Auch wenn Sie an diesem Punkt vielleicht glauben, in grade diesem zu stehen, hoffe ich, dass folgendes Beispiel die Annahmen verdeutlicht und warum diese notwendig waren.

Zuerst brauchen wir einen Hintergrund. Diesen können Sie recht einfach mit dem Verlauf-Werkzeug erzeugen, das Ergebnis könnte so aussehen:

Abbildung 7.5:
Der Horizont – unendliche Weiten ...

Im Folgenden erstellen Sie bitte noch zwei weitere MovieClips; zunächst einen, der den Baum (oder das Objekt, das Sie verteilen möchten) enthält und auch „baum“ heißt (sieht dann ggf. wie folgt aus).

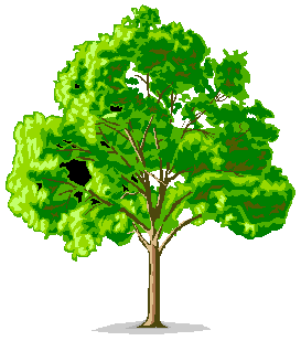


Abbildung 7.6: Die Grafik eines Baumes

Der zweite MovieClip ist leer, er enthält nur das ActionScript und erhält den Namen „ctrl“ (steht für Control).

Das einzige ActionScript (bis auf ein `stop();` im ersten und einzigen Frame der Hauptbühne) liegt in diesem MovieClip, beide MovieClips liegen auf der Hauptbühne.

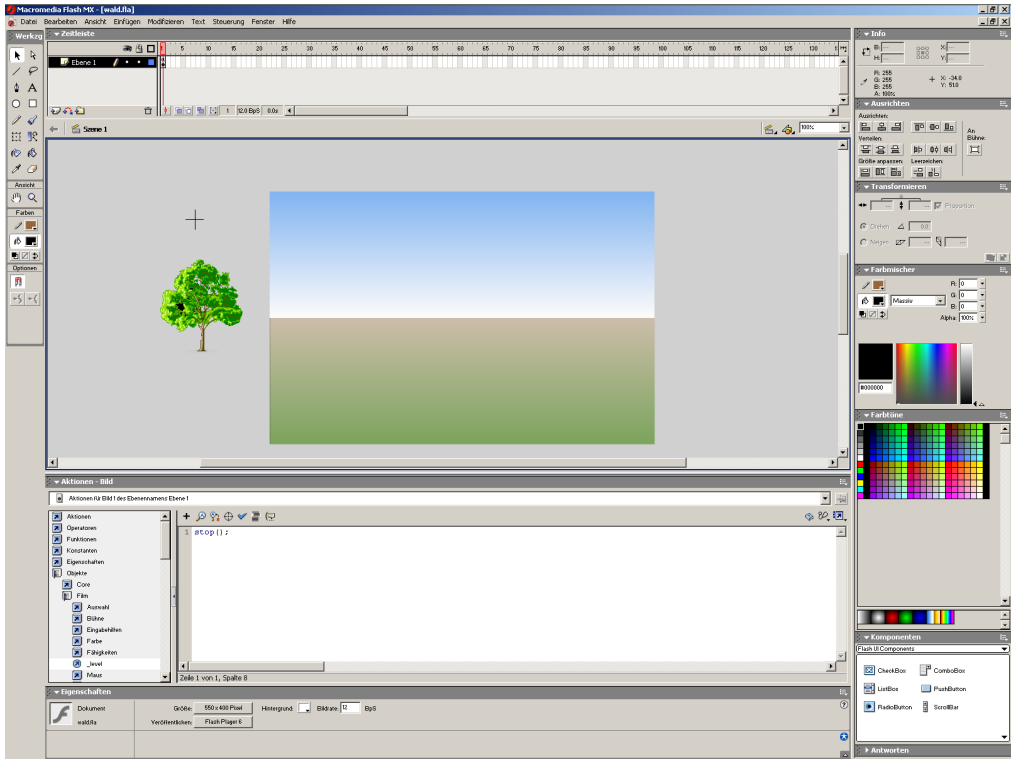


Abbildung 7.7: Dies ist eine mögliche Anordnung der Elemente auf der Hauptbühne.

Das Herzstück des Effektes ist in einem einzelnen MovieClip gelegen, der hier als Kreuz sichtbar ist. Vollständig „leere“ MovieClips anzulegen ist nicht zu empfehlen, da diese bei der späteren Bearbeitung nicht einfach auszuwählen sind.

(ActionScript des „crtl“)

```
onClipEvent(load){
  maxObj=50;
  for (i=0;i<=maxObj;i++) {
    _root.baum.duplicateMovieClip("film" + i,i+2);
    _root["film" + i].zKoord=200 * Math.random();
    _root["film" + i].xKoord=200 * Math.random();
    this._parent["film" + i]._y = 200;
  }
  _root.baum._visible=0;
  meineXpos=0;
  meineYpos=0;
  meinBlickwinkel=45;
  meineBlickrichtung=0;
}
```

Es werden hier zu Beginn die Anzahl der Objekte festgelegt und der „Baum“ (bzw. der MovieClip mit einem Bild dessen) entsprechend oft aus dem „Original“ dupliziert. Die x-z-Koordinaten werden hier zufällig auf Werte zwischen 0 und 200 gesetzt. Blickrichtung sowie Blickwinkel werden auch gleich festgelegt.

```
on(keyPress "<Left>"){
  meineBlickrichtung-=4;
  meineBlickrichtung=meineBlickrichtung%360;
}
```

Wird die Taste [←] gedrückt, so verändert sich die Blickrichtung um -4 Grad, wir drehen uns also. Um später keine Winkel › 360 zu erhalten wird hier ein Modulo mit eingefügt.

```
on(keyPress "<Right>"){
  meineBlickrichtung+=4;
  meineBlickrichtung=meineBlickrichtung%360;
}
```

Wird die Taste [→] gedrückt, so verändert sich die Blickrichtung um +4 Grad, wir drehen uns also. Um später keine Winkel › 360 zu erhalten wird auch hier ein Modulo mit eingefügt.

```
on(keyPress "<Down>"){
  meineXpos+=Math.sin((meineBlickrichtung-90)*Math.PI/180)*2;
  meineZpos-=Math.cos((meineBlickrichtung-90)*Math.PI/180)*2;
}
```

Wird die Taste [↓] gedrückt, verändern sich meine x-z-Koordinaten wie oben berechnet.

```
on(keyPress "<Up>"){
  meineXpos-=Math.sin((meineBlickrichtung-90)*Math.PI/180)*2;
  meineZpos+=Math.cos((meineBlickrichtung-90)*Math.PI/180)*2;
}
```

Wird die Taste [↑] gedrückt, verändern sich meine x-z-Koordinaten wie oben berechnet.

```
onClipEvent(enterFrame){
  for(i=0;i<=maxObj;i++){
```

Bei Aufruf des Movies (enterFrame) werden alle Objekte neu berechnet, ob und wo diese platziert werden sollen.

```
zDist = this._parent["film" + i].zKoord - meineZpos;
xDist = this._parent["film" + i].xKoord - meineXpos;
```

Hier werden die Koordinatenabstände berechnet, die man erhält, wenn man aus einer Vogelperspektive auf die Positionen der einzelnen Objekte schaut.

```
Abstand = Math.sqrt(zDist*zDist + xDist*xDist);
```

Der eigentliche Abstand, den man zwischen eigenem Blickpunkt und dem Objekt hat, errechnet sich über den Satz des Pythagoras. Da die Koordinatenabstände ja im rechten Winkel zueinander stehen und der Abstand mit diesen ein rechtwinkliges Dreieck bildet, kann man über die zwei bekannten Strecken die dritte (den Abstand) berechnen.

```
if (Abstand>5) {
```

Nahe stehende Objekte werden hier ausgeblendet, dies kann aber individuell angepasst werden.

```
if (meineBlickrichtung<0){
  meineBlickrichtung += 360;
}
degree = Math.asin(zDist/Abstand)/Math.PI*180;
plusXpos =
    (-Math.sin((meineBlickrichtung+90)*Math.PI/180))*2;
PlusZpos =
    Math.cos((meineBlickrichtung+90)*Math.PI/180)*2;
XVektor=(-plusZpos);
ZVektor=plusXpos;
upDown = (XVektor * zDist) - (ZVektor * xDist);
```

Hier wird ein Vektor gebildet, der im rechten Winkel zu unserer Blickrichtung steht. Wir bilden hier den so genannten Kreuzvektor der beiden oben beschriebenen Vektoren. Je nachdem ob sich das Objekt (aus der Vogelperspektive betrachtet) links oder rechts von der Achse, also vor oder hinter uns befindet, ist das Ergebnis „K“ positiv oder negativ, zeigt der Ergebnisvektor nach „oben“ oder „unten“.

$$\begin{Bmatrix} XVektor \\ ZVektor \\ 0 \end{Bmatrix} \times \begin{Bmatrix} xDist \\ zDist \\ 0 \end{Bmatrix} = \begin{Bmatrix} ? \\ ? \\ \pm K \end{Bmatrix}$$

```
if (upDown>0.5) {
```

Das Objekt liegt vor uns, eine weitere Überprüfung folgt.

```
if (xDist>0) {
  degree=90 + (-degree + 90);
} else if (degree<0) {
  degree=270 + (degree + 90);
}
relDeg=(Math.round(meineBlickrichtung)
    + Math.round(degree) + 90)%180;
```

Die hier eingeführte Variable „relDeg“ bezeichnet den zum Objekt relativen Blickwinkel abhängig von unserer Blickrichtung.

```
if((relDeg-90)>(-meinBlickwinkel)
    && (relDeg-90)<(+meinBlickwinkel)) {
```

Das Objekt liegt nun mit entgültiger Sicherheit innerhalb unseres Blickfeldes, nun geht es darum, es zu positionieren, zu skalieren und in eine entsprechende Reihenfolge zu bringen.

```
KachelScale = 2000/Abstand;
this._parent["film"+i]._xscale = KachelScale;
this._parent["film"+i]._yscale = KachelScale;
this._parent["film"+i].swapDepths(Math.round
    (-Abstand));
this._parent["film"+i]._x = 275
    + Math.cos(relDeg*Math.PI/180)*270;
```

Nun sind alle Berechnungen abgeschlossen, das Objekt mit Blickrichtung auf die Bühnenmitte (275px) ausgerichtet.

```
this._parent["film" + i]._visible=1;
```

Dieses Objekt wird angezeigt.

```
} else {
  this._parent["film" + i]._visible=0;
```

Dieses Objekt wird nicht angezeigt.

```
  }
} else {
  this._parent["film" + i]._visible=0;
```

Dieses Objekt wird nicht angezeigt.

```
  }
} else {
  this._parent["film" + i]._visible=0;
```

Dieses Objekt wird nicht angezeigt.

```
  }
 }
}
```

Das Endergebnis sieht dann so aus:

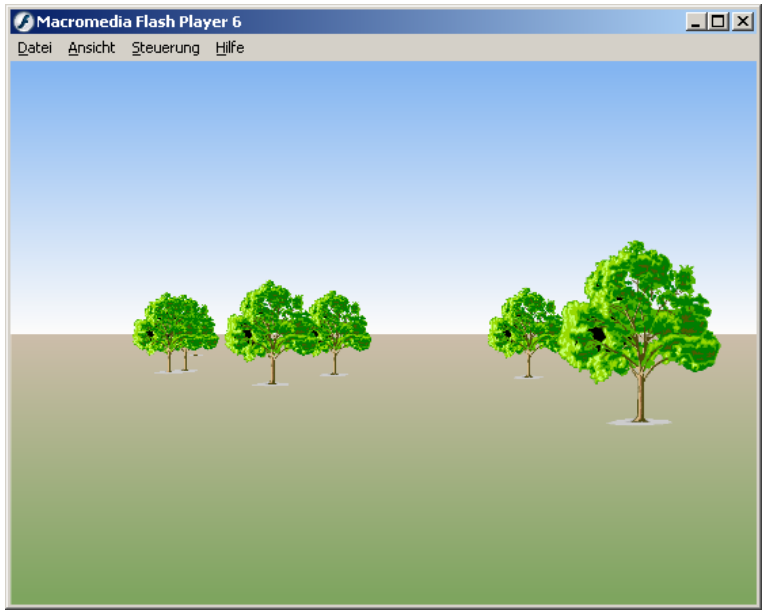


Abbildung 7.8: Der Wald im Flash Player

Um es ganz genau zu nehmen sieht ein mögliches Ergebnis so aus, da die Positionen ja bei jedem Start zufällig verteilt werden.

7.2 3D-Würfel

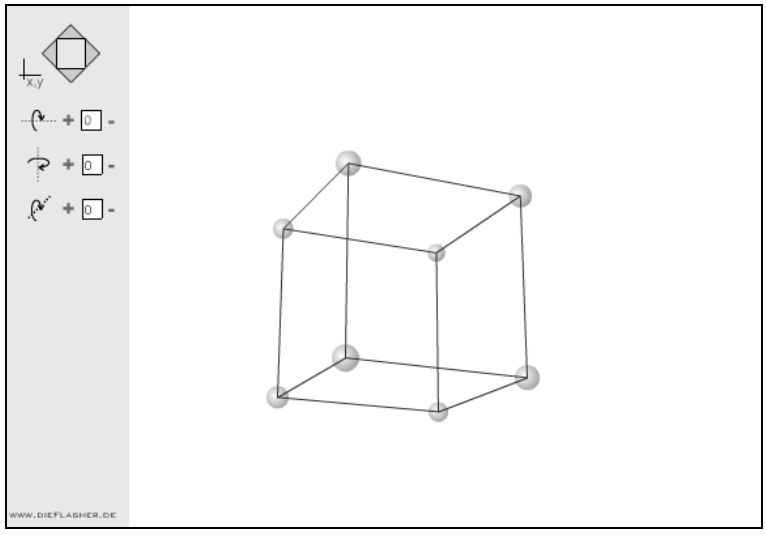


Abbildung 7.9: Ziel dieser Lektion – Erstellung eines Wireframe-Würfels

Möchten Sie in Flash – gegebenenfalls unter dem Aspekt der Spieleprogrammierung – immer wieder neue Effekte auf der Seite zeigen, bietet sich der folgende Effekt an. Da das Internet nicht nur informiert, sondern Besucher auch unterhält, ist es nahe liegend, interaktive Spielereien auf Ihrer Site unterzubringen. Sie bieten nicht nur Einblicke in den technischen Standard Ihrer Site, sondern sogar Einflussmöglichkeiten und Interaktivität. Die Informationspflicht ist aber dessen ungeachtet oberstes Gebot im Internet und darf nicht von Effekten zugedeckt werden.

Da Sie mit Flash mittlerweile etwas vertrauter sein dürften, werden wir alle benötigten Elemente auflisten. Sollten Sie als Quereinsteiger Verständnisprobleme haben, können Sie in den entsprechenden vorangegangenen Kapiteln nachschlagen.

Alle Elemente werden in einem MovieClip erstellt, der sich im ersten und einzigen Frame auf der Hauptbühne befindet.

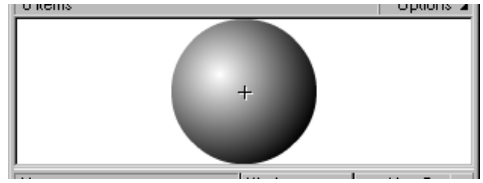
Zunächst brauchen wir einen Hintergrund, damit sich die Navigation abhebt. Maße und Farben können wir an dieser Stelle nicht verbindlich empfehlen; die Farbe, die wir gewählt haben, ist hellgrün, die Maße sind 100 x 500 Pixel.

Abbildung 7.10: Hintergrund der Marginalspalte

Dann muss ein Kreis (3D-Effekt durch radialen Farbverlauf) in einem MovieClip angelegt werden. Der vollständige Kreis soll einen Durchmesser von 17 Pixeln haben.

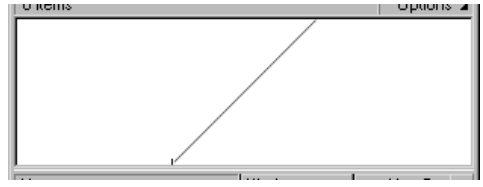
Die Instanz heißt „Kreis“.

Abbildung 7.11: Eckpunkte



Anschließend ist eine Linie als MovieClip anzulegen. Die Linie verläuft vom MovieClip-Mittelpunkt im 45°-Winkel nach rechts oben und wird mit den Maßen 100 x 100 Pixel dargestellt.

Abbildung 7.12: Verbindungslinie



Des Weiteren werden nun einige Buttons benötigt. Den „Kontaktbutton“ erstellen Sie wie gewohnt. Der Inhalt *www.DieFlasher.de* soll auf die Autoren des Flashfilms verweisen.

Abbildung 7.13: Vorlage für den Button

Wir brauchen jetzt noch weitere Buttons, um die Rotation zu kontrollieren. Wir werden später die Möglichkeit haben, den Würfel im Ursprung zu verschieben sowie Rotationen um die drei Achsen auszuführen. Um den Würfel zu verschieben, bieten sich Pfeile an. Da Flash es erlaubt, ein Objekt an mehreren Stellen gleichzeitig zu benutzen, soll uns ein Pfeil-Button genügen. Dieser kann wie folgt aussehen:

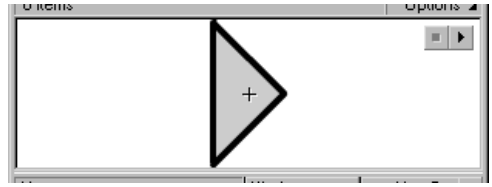


Abbildung 7.14: Richtungspfeil

Des Weiteren „navigieren“ wir den Würfel noch um drei Achsen. Da wir die Rotation mittels der Winkelgeschwindigkeiten kontrollieren, bietet es sich an, + und – zu nehmen, da wir auch „negativ“, sprich in beide Richtungen, drehen können. Dafür werden jetzt zwei Buttons erstellt und in den MovieClip des ersten Frames der Hauptbühne gelegt. Die Buttons können wie folgt aussehen:

Abbildung 7.15: Pluszeichen

Abbildung 7.16: Minuszeichen

Danach brauchen wir noch drei Textfelder zur Ausgabe der Winkelgeschwindigkeiten. Ziehen Sie sich dafür einfach ein kleines Textfeld mit der Option „Dynamischer Text“ auf. Kopieren Sie es und fügen Sie es noch zweimal ein. Benennen Sie die Felder logisch, d.h. für jeden verständlich. In diesem Fall heißen die drei Textfelder „xachse“, „yachse“, „zachse“.

Nun ordnen Sie die Navigationselemente an. Der „Pfeil“-Button muss noch dreimal eingefügt werden, um eine Ursprungsverschiebung auf der x/y-Ebene zu realisieren. Fertig kann das dann wie in Abbildung 7.17 aussehen:

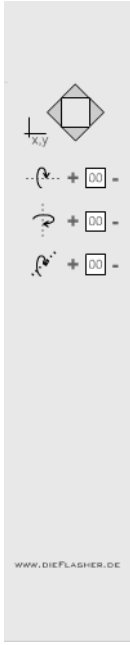


Abbildung 7.17: Die Marginalspalte

Nun fügen Sie in die Clip-Action des MovieClips, in dem alle anderen Elemente sind, zunächst folgende Initialisierungsdaten ein:

```
// Diese Aktion, also alles innerhalb der
{}-Klammern, wird einmalig ausgeführt.
onClipEvent (load) {
    // Hier werden die Punktkoordinaten
    initialisiert
    px0 = 70;
    py0 = 70;
    pz0 = 70;

    px1 = 70;
    py1 = 70;
    pz1 = -70;

    px2 = -70;
    py2 = 70;
    pz2 = -70;

    px3 = -70;
    py3 = 70;
    pz3 = 70;

    px4 = -70;
    py4 = -70;
    pz4 = 70;

    px5 = 70;
    py5 = -70;
    pz5 = 70;

    px6 = 70;
    py6 = -70;
    pz6 = -70;

    px7 = -70;
    py7 = -70;
    pz7 = -70;
```

```
    // Hier wird die maximale Punktanzahl festgelegt: sieben,
    weil Null mitgezählt wird.
    maxp = 7;
    // i, j , k sind die anfänglichen Winkelgeschwindigkeiten.
    Wenn Sie den Film ausführen, steht der Würfel.
    i = 0;
    j = 0;
    k = 0;
    // Hier wird der x/y-Ursprung des Würfels festgelegt.
    xoffset = 300;
    yoffset = 225;
    // Die Stärke der Perspektive.
    perspect = 5;
    // Die Maximalgröße der Punkte.
    zoom = 100;
    // Eine Geschwindigkeitskontrolle kann individuell ange-
    passt werden, je nach Anzahl der zu berechnenden Punkte.
    pitchx = 30;
    pitchy = 30;
    pitchz = 30;
    // Der Weg, die acht Punkte mit den zwölf Linien zu verbin-
    den.
    knoten = new Array();
    knoten[0]=5;
    knoten[1]=6;
    knoten[2]=7;
    knoten[3]=4;
    knoten[4]=5;
    knoten[5]=6;
    knoten[6]=7;
    knoten[7]=4;
    knoten[8]=1;
    knoten[9]=2;
    knoten[10]=3;
    knoten[11]=0;
    // Entfernung in Z. Kann optional geändert werden.
    entf=-800;
    // Hier werden die acht Eckpunkte des Würfels generiert.
    for (c=0; c<maxp+1; c++) {
        duplicateMovieClip (kreis, "kreis"+c, c);
    }
    // Hier werden die zwölf Verbindungslinien generiert.
    for (lin=0; lin<12; lin++) {
        duplicateMovieClip (_level0.line, "line"+lin, lin+50);
    }
}
```

Jetzt, wo die Daten initialisiert wurden, kann man zum zweiten Part wechseln. Dieser wird eingeleitet durch ein

```
onClipEvent (enterFrame) {
    // Hier wird xr, yr, zr (X, Y, Z - Achse Rotation)
    initialisiert.
    xr = i/pitchx;
    yr = j/pitchy;
    zr = k/pitchz;
    // Die Funktion calc (Calculate = berechne) wird definiert.
    Übergeben wird ein Wert.
    function calc (wert) {
        //Folgend wird mittels Polarkoordinaten die Punkttrans-
        formation im Raum berechnet.
        eval("zn" add wert) = Math.sin(xr)*eval("py" add
        wert)+Math.cos(xr)*eval("pz" add wert);
        eval("py" add wert) = Math.cos(xr)*eval("py" add wert)-
        Math.sin(xr)*eval("pz" add wert);
        eval("pz" add wert) = eval("zn" add wert);
        eval("zn" add wert) = Math.cos(yr)*eval("pz" add wert)-
        Math.sin(yr)*eval("px" add wert);
        eval("px" add wert) = Math.cos(yr)*eval("px" add
        wert)+Math.sin(yr)*eval("pz" add wert);
        eval("pz" add wert) = eval("zn" add wert);
        eval("yn" add wert) = Math.cos(zr)*eval("py" add wert)-
        Math.sin(zr)*eval("px" add wert);
        eval("px" add wert) = Math.cos(zr)*eval("px" add
        wert)+Math.sin(zr)*eval("py" add wert);
        eval("py" add wert) = eval("yn" add wert);
        // Hier wird mittels pz die „Tiefe" des Punktes im Raum
        gemessen. Um den Punkt in Fluchtpunktnähe zu verschie-
        ben, wird hier ein Korrekturfaktor berechnet, der die
        gleich zugeordneten Werte auf der x/y-Ebene mehr Rich-
        tung Mitte verschiebt.
        eval("deep" add wert)=entf/(entf-eval("pz" add wert));
        // Hier werden die Punkte selbst skaliert, ihre Größe
        der „Tiefe", also Entfernung zum Betrachter, entspre-
        chend verkleinert.
        setProperty ("kreis" add wert, _xscale, zoom-eval("pz"
        add wert)/perspect);
        setProperty ("kreis" add wert, _yscale, zoom-eval("pz"
        add wert)/perspect);
        // Hier werden die Koordinaten dem Punkt zugeordnet.
        setProperty ("kreis" add wert, _x, eval("px" add
        wert)*eval("deep" add wert)+xoffset);
        setProperty ("kreis" add wert, _y, eval("py" add
        wert)*eval("deep" add wert)+yoffset);
        setProperty ("kreis" add wert, _alpha, 40);
    }
```

```
        // Hier wird die Berechnung jedes einzelnen Punktes
        aufgerufen.
        for (c=0; c<maxp+1; c++) {
            calc(String(c));
        }
        // Folgend werden die Verbindungslinien zwischen den
        einzelnen Punkten berechnet.
        for (lin=0; lin<12; lin++) {
            li=knoten[lin];
            kn=lin;
            // Durch das „kn %= 8" wird kn automatisch für Werte
            größer als acht um acht subtrahiert.
            kn %= 8;
            // Die Position der Linie wird an die Position des
            Punktes angeglichen.
            eval("line"+lin)._x=eval("kreis" + kn)._x;
            eval("line"+lin)._y=eval("kreis" + kn)._y;
            // Da die Linie die Maße 100 x 100 Pixel hat, hat
            _xscale bzw. _yscale gleichzeitig die realen Pixel-
            werte. Es werden durch Berechnen des Abstandes die
            Streckungs- bzw. Stauchungsfaktoren berechnet.
            eval("line"+lin)._xscale=-(eval("kreis" + kn)._x-
                eval("kreis" + li)._x);
            eval("line"+lin)._yscale=eval("kreis" + kn)._y-
                eval("kreis" + li)._y;
        }
    }
```

Jetzt haben Sie (hoffentlich) alle Scripts richtig eingefügt, die Instanznamen richtig vergeben und die Filme in der richtigen Hierarchie eingefügt. Nun muss der Navigation noch Leben eingehaucht werden. Im Folgenden nummerieren wir die Elemente durch, das jeweils dazugehörige ActionScript werden Sie darunter finden.

Im Folgenden das dazugehörige ActionScript:

1
```
on (press){
  yoffset=yoffset-5;
}
```

2
```
on (press){
  xoffset=xoffset+5;
}
```

3
```
on (press) {
  yoffset=yoffset+5;
}
```

4
```
on ( press ) {
  xoffset=xoffset-5;
}
```

5
```
on(press){
  i=i+1;
  xachse=i;
}
```

6
```
on(press){
  i=i-1;
  xachse=i;
}
```

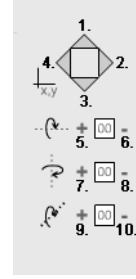


Abbildung 7.18: Übersicht über die Schaltflächen

```
7
    on (press){
      j=j+1;
      yachse=j;
    }
```

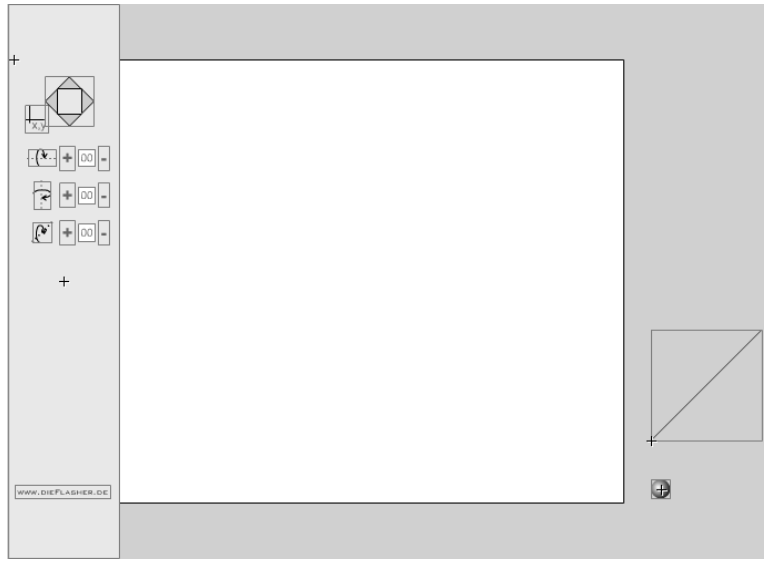
```
8
    on (press){
      j=j-1;
      yachse=j;
    }
```

```
9
    on (press){
      k=k+1;
      zachse=k;
    }
```

```
10
    on (press){
      k=k-1;
      zachse=k;
    }
```

Wenn alle Elemente richtig platziert wurden, darf die Hauptbühne nur einen MovieClip beinhalten, der aber die komplette Berechnung per ActionScript in sich trägt. Das sieht auf der Hauptbühne dann wie folgt aus:

Abbildung 7.19: Anordnung in der Entwicklungsumgebung



Wenn Sie überlegen, an Stelle des Würfels z.B. eine Pyramide oder ein aus Punkten zusammengesetztes Objekt zu „importieren“, so können Sie dies. Nur müssen Sie den `Array()` wegen der Linien etc. manuell anpassen. Ansonsten fügen Sie einfach fortlaufend die Punkte ein, ändern zu guter Letzt noch `maxp`, die Variable, in der die Anzahl der Punkte im Raum gespeichert ist. Diese Engine halten wir für durchaus erweiterbar, sie kann nun individuellen Kundenwünschen angepasst werden. Die Koordinatensteuerung kann z.B. alternativ per Mausbewegung abgefragt werden und die Punkte können in Buttons verwandelt werden.

7.3 Schein-3D

3D-Fläche

Wie man in den vorangegangenen Beispielen sehen konnte, nimmt man Entfernung durch verschiedene Dinge wahr wie Größenveränderung, Unschärfe oder Verschiebung zu einem gemeinsamen Fluchtpunkt. Um Shapes zu verformen bietet Flash zwar ein paar Funktionen an, aber das perspektivische Verzerren fehlt. An dieser Stelle kann man sich helfen, wenn man den Eindruck der dritten Dimension erzeugen möchte, folgendes Beispiel wird dieses erläutern.

Das Problem bei einer perspektivischen Verzerrung ist, dass eine Ungleichmäßigkeit vorliegt, ein Teil der Grafik wird weniger stark verzerrt als ein anderer. Da Flash diese Funktion leider nicht bietet, ist man als Anwender auf seinen Einfallsreichtum angewiesen – oder eine entsprechende Anleitung, wie man sich so etwas selbst erstellt.
Für eine perspektivische Verzerrung ist folgende Transformation charakteristisch:

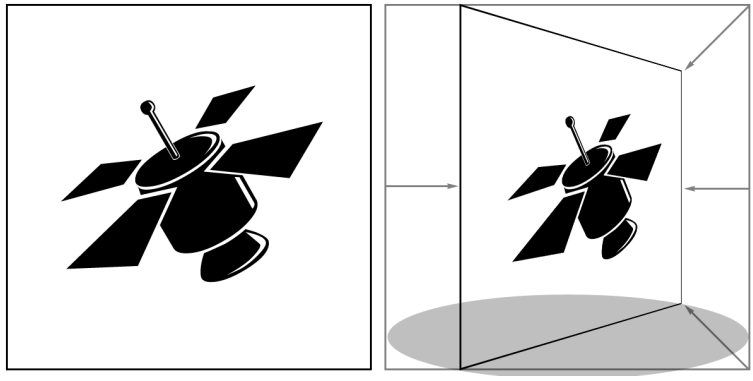
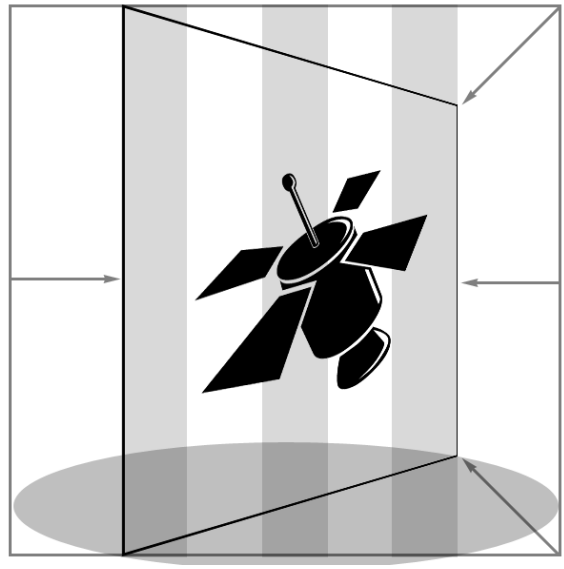


Abbildung 7.20: Perspektivische Verzerrung

Wie zu erkennen ist, wird der weiter entfernte Teil des Bildes nicht nur über die horizontale Achse, sondern auch über die vertikale Achse gestaucht. Leider kann man in Flash Objekte nicht dementsprechend asymmetrisch verzerren, weshalb nur die Alternative bleibt, das Bild so zu zerlegen, dass Flash es verarbeiten kann um den gleichen Effekt zu erzielen. Da die horizontale Stauchungsachse in diesem Fall gleich bleibt, bietet es sich an, das Bild in Streifen zu zerlegen.

Abbildung 7.21: Das Bild zerlegt in Streifen



Um diese Streifen zu erzeugen muss man das Originalbild entsprechend oft duplizieren und mit Ebenenmasken versehen. Diese einzelnen Teile muss man dann dem Grad der Rotation entsprechend stauchen bzw. strecken. Um eine Drehung um mehr als 180° zu erzeugen muss man den MovieClip, der das Bild enthält, manuell im Entwurf des Filmes spiegeln und als zweiten MovieClip ablegen.

Zuerst wird die „Kachel“, die rotiert werden soll, angelegt, im Beispiel sieht das dann wie folgt aus.

Abbildung 7.22: Das zu rotierende Bild

Im Folgenden wird ein MovieClip mit der Bezeichnung „dreid“ angelegt, der zum einen eine Instanz dieses Bildes unter „kachel“ und eine unter dem Namen „spiegelkachel“ enthält. Die Instanz „spiegelkachel“ ist zum Original nur gespiegelt, wie der Name schon vermuten ließ. Des Weiteren enthält der MovieClip „dreid“ einen Streifen, der als Maske fungiert, auch hier ist der Name der Instanz „maske“ nahe liegend. Zuletzt wird neben dem Bild, der gespiegelten Instanz und der Maske noch ein anderer MovieClip angelegt, der das ActionScript enthält. Dieser Clip erhält den Namen „ctrl“, folgendes ActionScript ist enthalten.

```
onClipEvent(load){
  _parent["maske"]._visible=0;
  _parent["kachel"]._visible=0;
  _parent["spiegelkachel"]._visible=0;
  _parent["ctrl"]._visible=0;
  for (i=0;i<7;i++){
    this._parent["maske"].duplicateMovieClip("maske" + i, i);
  }
}

onClipEvent(enterFrame){
  step=20*(Math.cos((this._parent.degree) * Math.PI / 180));
  if (this._parent.degree>90 && this._parent.degree<270) {
    for (i=0;i<7;i++){
      this._parent["kachel"].duplicateMovieClip("kachel"
        + i,i+7);
    }
  } else {
    for (i=0;i<7;i++){
      this._parent["spiegelkachel"].duplicateMovieClip ("kachel"
        + i,i+7);
    }
  }
  if (step<0) {
    k=-1;
  } else {
    k=1;
  }
  for (i=0;i<7;i++){
    this._parent.degree=this._parent.degree%360;
    this._parent["maske" + i]._width = Math.abs(step+k);
    this._parent["maske" + i]._x = (step * (i + 0.5))-(step * 3.5);
    this._parent["kachel" + i]._width=Math.abs(step * 7);
    this._parent["kachel" + i]._x=0;
    this._parent["maske" + i]._y=0;
    this._parent["kachel" + i]._y=0;
    this._parent["kachel" + i]._yscale = 100 + 10
      * Math.sin(this._parent.degree * Math.PI / 180))
      * (1-(i + 1) / 3.5);
    this._parent["kachel" + i].setMask(this._parent["maske"
      + i]);
  }
}
```

Es wird der Wert „degree“ ausgewertet, den man durch einfache Zuweisung in den MovieClip setzen kann. Um die Performance zu testen und auch eine Bewegung zu erzeugen legt man einfach auf der Hauptbühne einen weiteren MovieClip „zaehler“ an, der folgendes ActionScript enthält.

```
onClipEvent(load){
  var iMax=1;
  this._visible=0;
  for (i=1;i<=iMax;i++) {
    _root.dreid.duplicateMovieClip("dreid" + i,i+2);
    _root["dreid" + i].degree=30*i;
    _root["dreid" + i]._x=i*80;
    _root["dreid" + i]._y=i*10;
  }
  _root["dreid"]._visible=0;
}

onClipEvent(enterFrame){
  for (i=1;i<=iMax;i++) {
    _root["dreid" + i].degree = _root["dreid" + i].degree + 6;
  }
}
```

Beim ersten Laden werden iMax Kopien erzeugt und diese je um 30 Grad in der Drehung, 80 Pixel in der Horizontalen und 10 Pixel in der Vertikalen versetzt angelegt. Um die Rotation nun zu „animieren“ muss man nur den Wert der Drehung fortlaufend ändern, dies passiert jeweils beim Aufruf des MovieClips – immer in 6-Grad-Schritten.

Das Ergebnis der Rotation sieht dann wie folgt aus.

Abbildung 7.23: Das perspektivisch verzerrte Ergebnis

7.4 3D-Animation (Import)

Gast-Autor: Aaron Kreis – Beispiel: www.DieFlasher.de

Flash unterstützt kein echtes 3D-Format. Es besteht jedoch die Möglichkeit Sequenzen in 3D-Programmen zu erstellen und als Bildsequenz in Flash zu importieren.

Bevor Sie jedoch mit der Erstellung von 3D-Animationen beginnen, sollten Sie sich über die große Vielfalt der 3D-Programme informieren.

Weiterhin ist darauf zu achten, dass der Renderer eine Exportfunktion in das Standard *.3DS-Format besitzt. Dieses Format wird benötigt, um später die jeweiligen Objekte durch Standalone-Tools wie z.B. Swift3D oder Vecta3D in Vektoren umrechnen zu lassen.

***Abbildung 7.24:** Intro Die Flasher*

Um einen Vorspann wie auf der *www.DieFlasher.de*-Internetseite zu erstellen, sind jedoch gewisse Grundkenntnisse der 3D-Verarbeitung notwendig.

- Modellierung
- Material
- Licht
- Animation

bilden die Kerngebiete der 3D-Animation.

Modellierung ist der erste Schritt bei der Erstellung einer Szene. Durch verschiedene Modellierungsverfahren wie z.B. Polygon- oder Nurbsmodellierung und die besonderen Werkzeuge entsprechender 3D-Programme sind alle vorstellbaren Formen realisierbar.

Während die Geometrie die räumliche Definition von Objekten bestimmt, beschreibt das Material das Aussehen von Objekten. In der realen Welt wird das Aussehen von Objekten durch die Oberflächeneigenschaft in Verbindung mit Licht bestimmt.

In 3D-Programmen müssen die unterschiedlichen Attribute für Farbe, Transparenz, Reflektionen etc. erst definiert werden.

Der Grund, sich für eine einfach texturierte Oberfläche zu entscheiden, ist ganz einfach zu erklären. Durch das Importieren von Bildsequenzen erreichen Flash-Dateien eine enorme Größe. Um die Dateigröße zu reduzieren haben wir uns dazu entschlossen, die einzelnen Bilder zu vektorisieren.

Diese Vorgehensweise ist nur sinnvoll, solange sich durch das Vektorisieren nicht zu viele Vektoren bilden.

Abbildung 7.25: Vektorisierte Grafik

Um ein Bitmap zu vektorisieren gehen Sie folgendermaßen vor:

Legen Sie eine neue Ebene an und wählen Sie Datei › Import. Suchen Sie das Bild, welches Sie bearbeiten möchten, und bestätigen Sie mit OK.

Das ausgesuchte Bitmap wird nun auf der Bühne und der Bibliothek eingefügt. Durch Drücken der Tastenkombination AltGr+L können Sie die Bibliothek aufrufen. An dem Baum-Icon ist zu erkennen, dass es sich hierbei um ein Bitmap handelt. Beim Doppelklicken auf das Icon öffnet sich der Dialog für Bitmaps.

Abbildung 7.26: Eigenschaftsfenster eines Bildes

Die wichtigsten Bilddateiformate in der Übersicht:

Formatname	Kurz-form	Kompri-mierung	Vergleich zur Originalgröße von 4,74 MB	Anmerkung	Zweck/Anwendung
Bitmap	BMP RLE	nein ja – als RLE-Datei mit 256 Farben	4,74 MB als RLE: ‹ 1,59 MB	als Windows- und OS/2- Format	Windows-Programme
Encapsulated Postscript	EPS	nein	13,12 MB	ASCII- Codierung mit 8-Bit Vorschau	DTP
Compuserve „Graphics Interchange Format“	GIF	immer	1,30 MB	maximal 8 Bit (256 Farben) Farbtiefe! verschiedene Besonder-heiten: Transparenz, Ani-mation, Interlaced, ...	kompatibel zu vielen Programmen einer der Internet-Standards für Logos, Buttons, Grafiken mit geringer Farbtiefe
JPEG	JPG	immer	1,62 MB 0,23 MB 0,05 MB	Option „sehr gut“ Option „gut“ Option „mäßig“ je nach Komprimierung wird das Bild verändert, bei starker Vergrößerung stören sogenannte „Fractale“	Speicherplatz sparen einer der Internet-Standards für Photos, Grafiken mit hoher Farbtiefe
Photo CD	PCD	ab Auflösung 1024 x 1536	6 Auflösungen möglich	Bild muß zur Speiche-rung konvertiert werden	nur zum Lesen von Photo-CDs
Pixar	PXR	nein	4,74 MB	für High-End-Workstations (inkompatibel zu PCs)	Austausch mit Prepress-Systemen
Pict(ure)	PIC	variable	‹ 3,79 MB	Hausformat von Apple Macintosh	kompatibel zu Mac-Programmen
Paintbrush PCX	PCX	nein	4,8 MB	verliert vermutlich an Bedeutung	Bildverarbeitung und -austausch
Photoshop	PSD	ja, mit variablem Faktor	‹ 4,62 MB	konzentriert sich auf Pho-toshop	mit zusätzlichen Photoshop-spezifischen Optionen (z.B. Bildebenen)
Targa	TGA	nein	4,74 MB		sehr kompatibel, Belichtungsdienste
TIFF (Tagged Image File Format)	TIF	nein	4,74 MB		kompatibel zu vielen EBVs, schnelles Öffnen und Speichern
TIFF, LZW-komprimiert	TIF	ja	‹ 2,73 MB	komprimiert ohne Informationsverluste	kompatibel zu vielen EBVs
TIFF 8 Bit (256 Farben)	TIF	nein	‹ 1,58 MB	reduzierte Farbinformation	kompatibel zu vielen EBVs
TIFF 8 Bit, LZW-komprimiert	TIF	ja	‹ 0,71 MB	komprimiert gegenüber der 8-Bit-Version verlustfrei	kompatibel zu vielen EBVs

Darin werden Informationen wie letztes Speicherdatum, Auflösung, Farbtiefe usw. angezeigt. Falls Sie zwischenzeitlich innerhalb eines Grafikprogramms eine Veränderung an der Bitmap vorgenommen haben sollten, können Sie diese durch Aktualisieren erneut importieren.

Durch das Häkchen bei Glätten werden vergrößerte Bitmaps weich gezeichnet (einzelne Pixel werden nicht mehr erkannt).

Rufen Sie nun über Modifizieren/Bitmap nachzeichnen die Kompressionseinstellung auf.

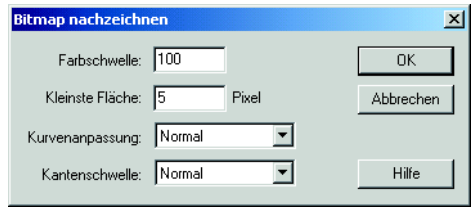


Abbildung 7.27: Eigenschaftsfenster für das Nachzeichnen

- Farbschwellwert gibt an, wie unterschiedlich zwei nebeneinander liegende Pixel sein dürfen, damit sie als gleichfarbig interpretiert werden. Je kleiner der Wert, desto detallierter das Ergebnis.
- Kleinste Fläche gibt die Anzahl der Pixel an, die zu einer Farbfläche zusammengefasst werden sollen.
- Kurvenanpassung bestimmt, wie stark die Kurven geglättet werden.
- Kantenschwelle bestimmt den Umgang mit den Kanten.

Es gibt keine ideale Voreinstellung, so bleibt Ihnen nichts anderes übrig, als mit den jeweiligen Einstellungen zu experimentieren.

Beim Einrichten einer Szene spielen Lichtquellen eine ganz wesentliche Rolle. Erst sie ermöglichen es, Objekte zu sehen. 3D-Programme bieten meist unterschiedliche Lichttypen zur Beleuchtung.

Für die *www.DieFlasher.de*-Homepage wurden zwei verschiedene Lichttypen verwendet.

▶ Ambient Light

Strahlt das Licht gleichmäßig in alle Richtungen. Damit kann die Grundhelligkeit einer Szene gesteuert werden.

Abbildung 7.28: Farbschwellwert = 10; kleinste Fläche = 10 ; Kurvenanpassung = sehr glatt; Kantenschwelle = viele Ecken

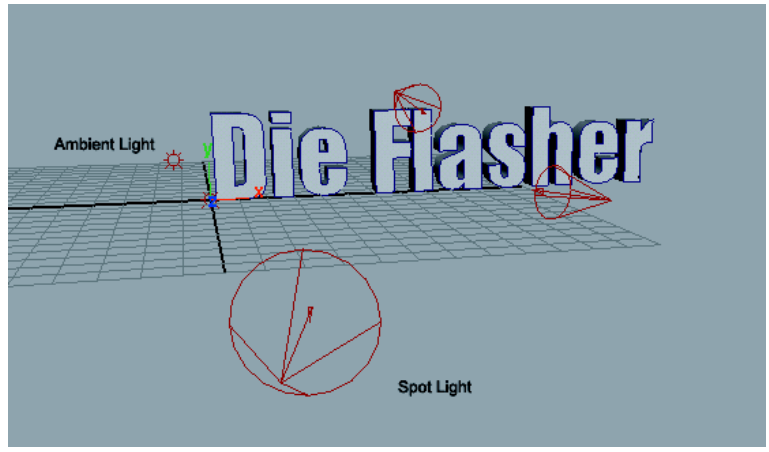
Abbildung 7.29: Farbschwellwert = 50; kleinste Fläche = 50; Kurvenanpassung = normal; Kantenschwelle = normal

Abbildung 7.30: Farbschwellwert = 100; kleinste Fläche = 50; Kurvenanpassung = normal; Kantenschwelle = wenige Ecken

- **Spot Light**

Strahlt das Licht gerichtet von einem bestimmten Punkt im Raum, in einem beschränkten Lichtkegel. (Es handelt sich um ein typisches Theater-Licht mit kegelförmigem Strahl.)

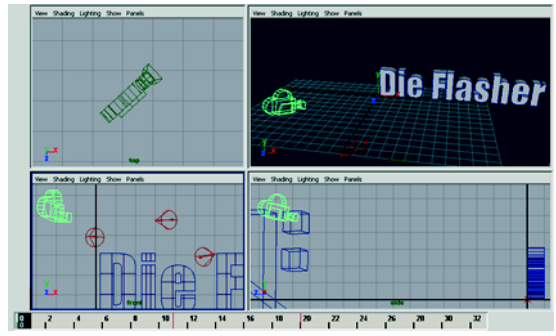
Abbildung 7.31: Spot Lights



Um die Animation der Homepage zu erstellen, haben wir uns für eine einfache Kamerafahrt entschieden. Das Prinzip ist recht einfach.

Hierbei werden innerhalb des 3D-Programms die Kamerapositionen verändert und durch Keyframes festgesetzt. Die einzelnen Zwischenschritte (Tweening) werden wie bei Flash automatisch berechnet.

Abbildung 7.32: Auf der Zeitleiste befinden sich in Frame 0, 11 und 20 die Keyframes.



Export aus 3D-Programm:
Beim Rendern von Bitmap-Animationen sollte das *.BMP-Format verwendet werden, denn Bitmaps komprimiert Flash besser als *.JPG.

Für den Export in einen 3D Standalone-Vektorisierer muss das *.3DS-Format ausgewählt werden. Die Bewegungen werden mit exportiert und können z.B. in Swift 3D oder Vektra3D später angepasst werden.

Importieren von Bitmaps nach Flash:
Über Datei › Import haben Sie nun die Möglichkeit, die im 3D-Programm erstellte Bildsequenz in Flash zu importieren. Flash erkennt dabei, ob es sich um eine Sequenz (eine Folge von Dateien mit gleichem Namen) handelt.

Um eine Sequenz zu importieren, müssen Ihre Bitmaps den gleichen Namen tragen und durchnummeriert sein, z.B.: Bild1.bmp, Bild2.bmp, Bild3.bmp usw.

Beim Import werden die Bitmaps in der Bibliothek abgelegt und als Einzelobjekt auf die aktuelle Ebene kopiert. Vektorgrafiken werden als Gruppen importiert. Sequenzen wiederum werden als aufeinander folgende Bilder in die aktuelle Ebene eingefügt.

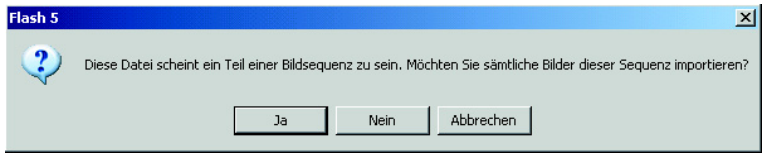


Abbildung 7.33: Meldung beim Import einer Bildsequenz

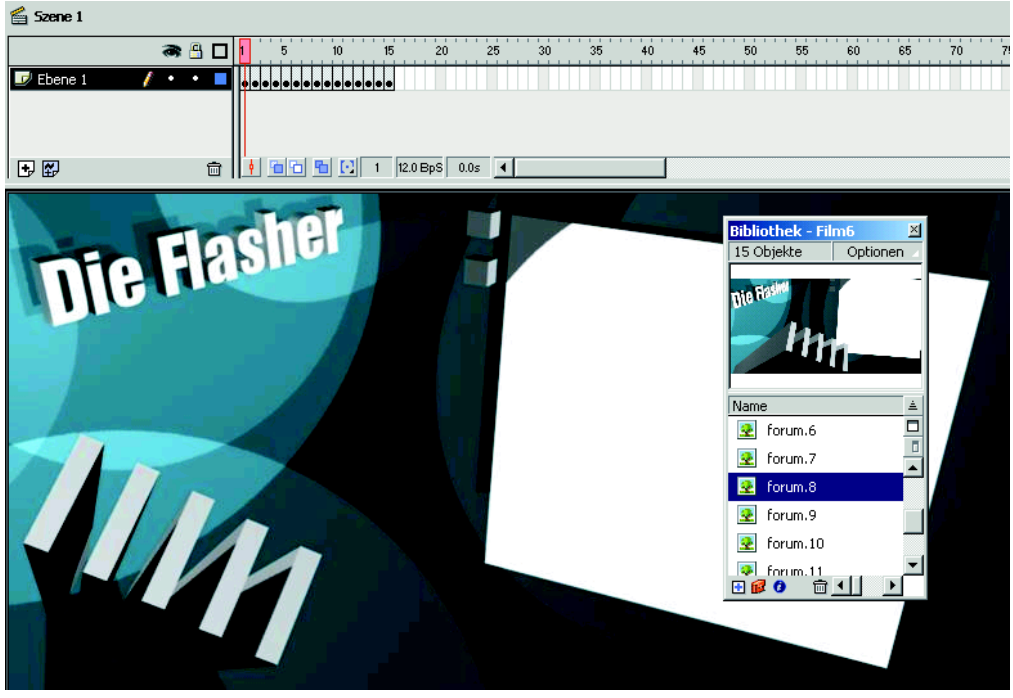


Abbildung 7.34: Die Internetseite www.DieFlasher.de

Ein Beispiel für eine solche Bild-für-Bild-Animation finden Sie im Verzeichnis zu diesem Kapitel auf der CD-ROM unter *Kapitel7/forum.fla*.

Falls Sie sich dafür entscheiden sollten, ein einfaches 3D-Logo zu erstellen, reicht bereits der Umgang mit Standalone-Tools wie z.B. Swift 3D.

Swift 3D ist eine Software, die 3D-Bilder und -Animationen konvertiert und direkt in Macromedia Flash (SWF)-, AI-, EPS- und SVG-Dateiformate exportiert. Sie erzeugt 3D-Vektoranimationen in verschiedenen Größenordnungen, die trotz allem nur geringe Dateigrößen erzeugen. Es ist auch möglich, als sequentielle EPS und im Adobe Illustrator-Dateiformat abzuspeichern, um sie in Adobe LiveMotion und anderen 2D-Vektorgrafikprogrammen zu verwenden.

In dem nun folgenden Beispiel möchten wir Ihnen zeigen, wie Sie mit Swift 3D ein 3D-Logo erstellen können. In unserem Beispiel soll sich der Schriftzug „DieFlasher“ horizontal um die eigene Achse drehen.

Nachdem Sie das Programm gestartet haben, sehen Sie das folgende Bild.

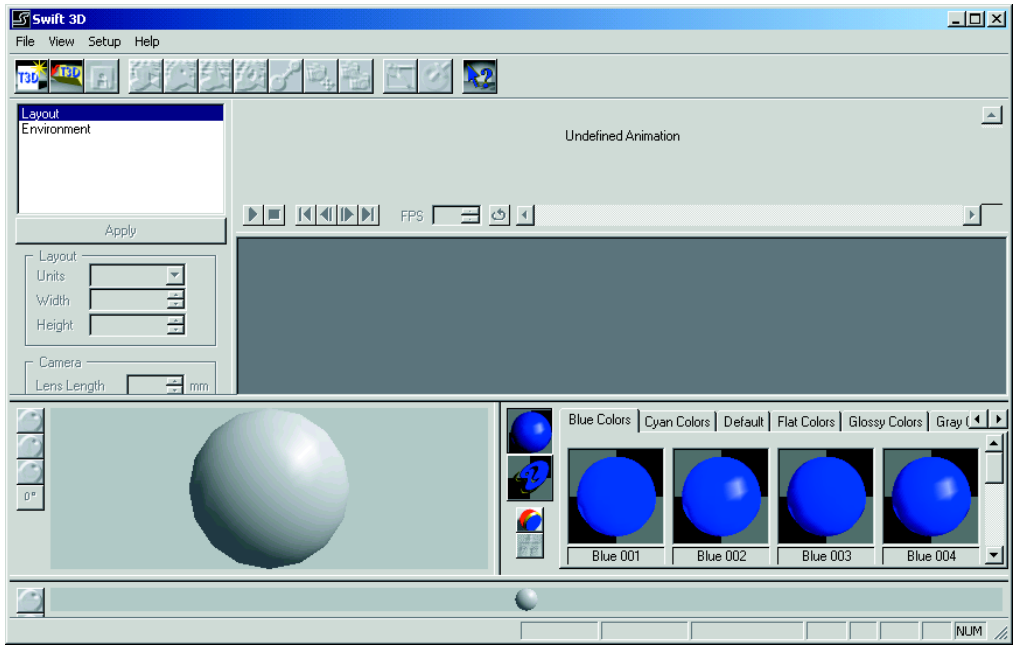


Abbildung 7.35: Swift-3D-Fenster

Drücken Sie bitte File › New › Create a new empty Swift 3D document um ein neues Projekt zu starten.

Oben in der Leiste befindet sich das Text-Werkzeug.

Abbildung 7.36: Symbolleiste

Klicken Sie auf das linke Symbol um das Textfenster zu öffnen.

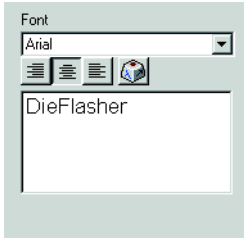


Abbildung 7.37: Textfenster

Innerhalb des Textfensters können Sie den gewünschten Inhalt eintragen.

Unter Font können Sie sich für eine andere Schriftart wie z.B. „Times New Roman“ entscheiden.

Unten rechts befindet sich die Farbpalette. Um die Farbe des Textes zu ändern, müssen Sie unten rechts auf eine Farbe klicken und mit gedrückter linker Maustaste die gewünschte Farbe auf den Text ziehen. In unserem Fall haben wir uns für ein Orange entschieden.

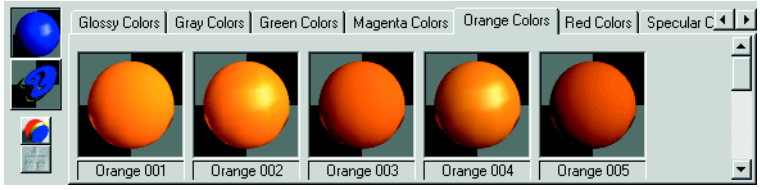


Abbildung 7.38: Farbeinstellungen

Unten rechts befindet sich der Show Animation-Knopf. Hier können Sie sich für eine Bewegung entscheiden.

Abbildung 7.39: Animationseinstellungen

Ziehen Sie nun mit gedrückter linker Maustaste das Horizontal Right-Symbol auf den Text.

Den Verlauf der Animation können Sie anhand der Zeitleiste erkennen. Durch Drücken des Play Animation-Knopfes können Sie die Animation starten.

Als letzten Schritt muss die Animation exportiert werden. Drücken Sie File › Export um das Exportfenster zu öffnen.

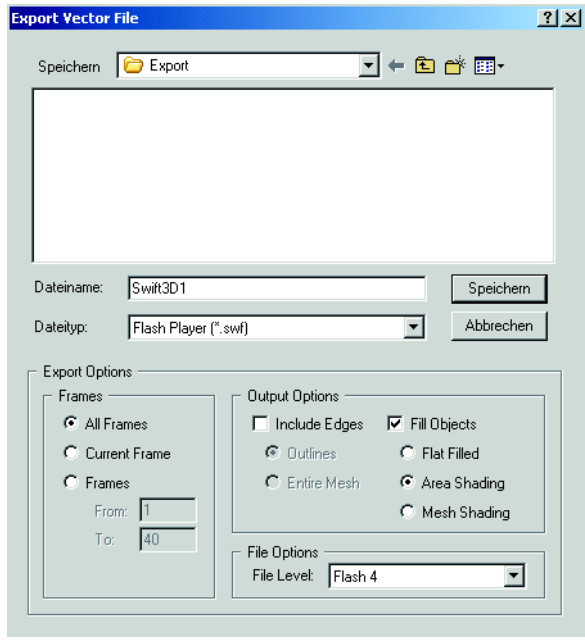


Abbildung 7.40: Exportfenster

Durch Drücken der Speichern-Taste wird die Animation gerendert und als *.swf-File auf der Festplatte gespeichert. Das Rendern kann je nach Rechenleistung einige Minuten in Anspruch nehmen.

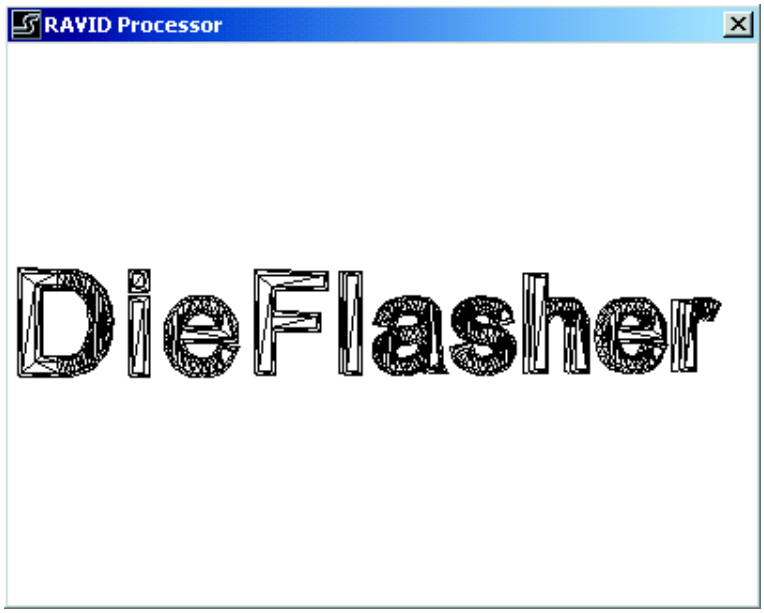


Abbildung 7.41: Drahtgitter-Modell

Abbildung 7.42: Das fertige SWF

Die fertige Animation befindet sich auf der CD in *Kapitel7/Swift.3D.swf*.

8 Spieleprogrammierung

Flash 5 hat bereits bewiesen, dass Flash auch für die Spieleprogrammierung eingesetzt werden kann. Wer jedoch an den Entwurf einer 3D-Engine unter Flash denkt, der muss seine Erwartungen zügeln. Selbst leistungsfähige PCs kann Flash überfordern. Zum einen durch das Duplizieren von mehreren großflächigen Movies mit Alphatweenings oder durch das Ansprechen und Versetzen von mehr als 30-40 Movies innerhalb eines Frames. Dabei kann die Framezahl sogar weit unter 5 BPS sinken (Standard 12 BPS). Seien Sie sich dieser Grenzen, die Flash setzt, bewusst!

Flash MX besitzt eine Menge neuer Funktionen. Besonders im Blick auf die objektorientierte Programmierung hat sich viel getan, so dass man dieses Mittel nun ohne große Bedenken auch unter Flash einsetzen kann, wie wir nachher in Kapitel 8.5 auch beweisen werden.

Erst durch die objektorientierte Programmierung ist es wirklich möglich, auch größere und vor allen Dingen auch komplexere Probleme strukturiert und übersichtlich zu lösen.

Denk-, Brett- oder Strategiespiele sind relativ problemlos unter Flash zu erstellen, da sich nicht viel auf dem Bildschirm bewegt. Bei Jump & Run oder ähnlichen Arcade-Spielen, bei denen sich mehrere Objekte auf dem Bildschirm bewegen, muss man schon teilweise ein wenig „geschickter“ programmieren um keine überflüssigen Prozeduren zu starten, die den Rechner unnötig belasten, da Flash wie schon erwähnt sehr schnell den Rechner ausbremst.

Die Euphorie zu Beginn der Entwicklung eines neuen Spiels ist groß. Beachten Sie aber: Je umfangreicher ein Spiel ist, desto unübersichtlicher wird es. Gehen Sie deshalb gerade auch beim Erstellen eines Spieles geplant vor! Entwerfen Sie zuerst am besten ein Pflichtenheft mit den Muss- und Kann-Kriterien. Danach sollten Sie sich in Ruhe überlegen, inwieweit das Niedergeschriebene wirklich unter Flash realisierbar ist. Sicherlich müssen Sie hierfür, für den Fall, dass es sich um ein großes Projekt handelt, schon eine gewisse Programmiererfahrung in Flash mitbringen. Aber schließlich hat auch niemand behauptet, dass das Programmieren eines anspruchvollen Spiels leicht wäre!

Nachdem also Ihr Spielkonzept steht und Sie sich auch schon eingehend Gedanken über die Realisierbarkeit gemacht haben, sollten Sie Ihre Gedanken wieder festhalten, indem Sie ein Klassendiagramm zeichnen.

Danach sollten Sie die auftretenden „Probleme“ in Teilstücke aufteilen und getrennt zu den Problemen die programmiertechnische Lösung suchen. Bei den meisten Spielen wird es aber letztendlich immer so sein, dass Sie zuerst mit der Programmierung des Spielfeldes beginnen müssen, um danach die einzelnen Objekte darauf zu erstellen.

Bevor wir jetzt direkt in die Programmierung eines komplexen Spiels einsteigen, wollen wir Ihnen vorab ein paar Grundelemente aus der Spieleentwicklung zeigen, mit denen Sie dann schon selbst ein kleines Spiel erstellen können.

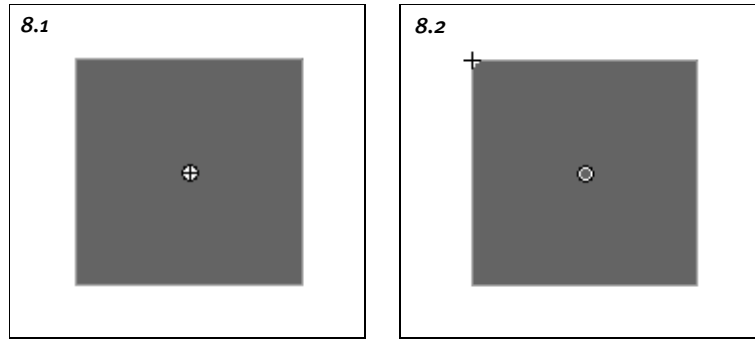
Bei den Grundelementen dreht es sich vorab erst einmal um die Bewegung von Objekten, um die Simulation von Schwerkraft und um die Kollisionserkennung sowie die damit verbundene Energieübertragung.

8.1 Bewegung

Um Objekte in Flash per ActionScript zu bewegen, muss man sie ansprechen können. Dafür muss es einen Instanznamen besitzen. Instanznamen können Movies, Schaltflächen und Textfeldern zugewiesen sein. Wir werden im Folgenden immer ein Movie benutzen. Um das Movie zu bewegen greift man auf die Eigenschaften _x und _y zu. Dies sind die jeweiligen Koordinaten des Objektes auf dem Bildschirm. Diese Koordinaten werden immer relativ zum Nullpunkt des Objektes ermittelt.

Abbildung 8.1: Objekt mit zentriertem Nullpunkt

Abbildung 8.2: Objekt mit dem Nullpunkt in der linken oberen Ecke



Dies ist wichtig zu wissen! Wenn Sie einem Objekt, das den Nullpunkt in der Mitte hat, die Koordinaten o,o zuweisen, würde dies zu ¾ außerhalb der Bühne liegen.

Angenommen das Objekt hat den Instanznamen Rechteck. Dann würde man über rechteck._x=o und rechteck._y=o den o-Punkt des Objektes gleich dem o-Punkt der Bühne setzen und so das Objekt auf der Hauptbühne versetzen.

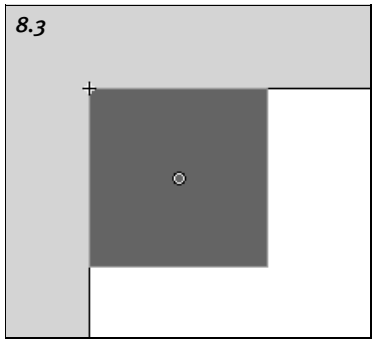


Abbildung 8.3: Objekt, dessen Nullpunkt links oben in der Ecke liegt

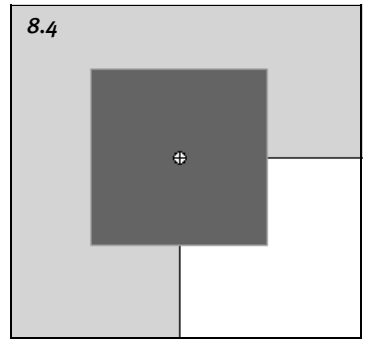


Abbildung 8.4: Objekt mit zentriertem Nullpunkt

Lassen Sie sich bitte nicht durch den Punkt irritieren, dieser ist für das ActionScript unbedeutend und nur für die Positionierung in Flash mittels des Info-Bedienfeldes relevant. Siehe auch in Kapitel 2.1 im Abschnitt „MovieClip“ ab Seite 137.

Linear

Eine lineare Bewegung bedeutet, dass sich ein Objekt mit gleichbleibender Geschwindigkeit über den Bildschirm bewegt. Dies wäre vergleichbar mit einem Auto, das die ganze Zeit über direkt eine konstante Geschwindigkeit fährt.

Dieses Mittel wird z.B. oft bei Jump & Run eingesetzt um Felsbrocken über den Bildschirm zu bewegen.

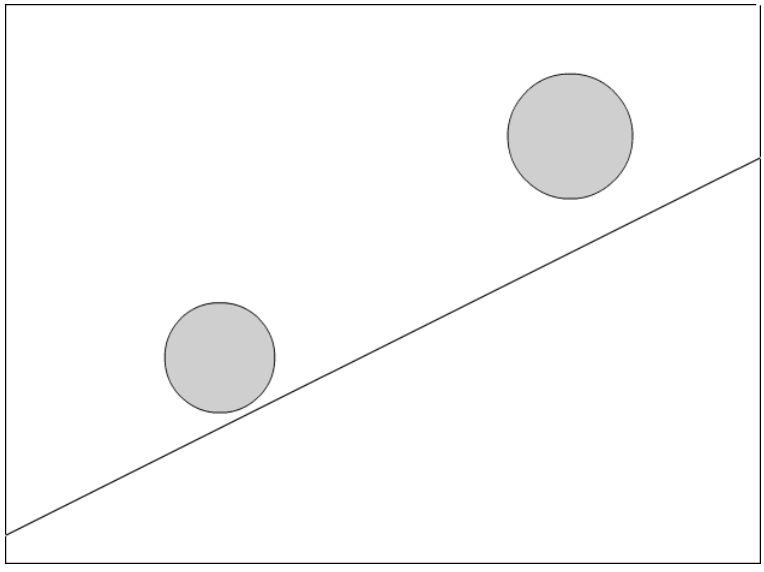


Abbildung 8.5: Springende Felsbrocken

Vielleicht erinnert sich der eine oder andere von Ihnen noch an die doch schon etwas in die Jahre gekommenen Spiele des Atari. Wenn man genau darüber nachdenkt, wird man sich sicherlich auch noch daran erinnern können, dass die Felsbrocken sehr unrealistisch den Abhang herunter kamen. Dies wird bei uns im Folgenden nicht anders sein!

Dies liegt zum einen daran, dass Felsbrocken, die einen Berg hinunterrollen, natürlich auch beschleunigen, dies war damals nicht der Fall und so wird es in unserem Beispiel auch sein. Die Felsbrocken haben eine konstante Geschwindigkeit, damit der spätere Spieler besser abschätzen kann, wann er springen oder sich ducken muss.

Bevor wir nun direkt auf die Programmierung eingehen, eines noch vorab. In diesem Kapitel haben wir fast ausschließlich mit Befehlen gearbeitet, die es auch schon in Flash 5 gab. Sicherlich wäre es auch möglich gewesen, einen leeren MovieClip zu generieren und in diesem mit den neuen Zeichenfunktionen einen Kreis als Fels zu zeichnen usw. Es geht uns hierbei jedoch nicht darum, die neuen Befehle zu demonstrieren, sondern darum, in dem Beispiel auf möglichst einfache Weise eine lineare Bewegung mit ActionScript darzustellen. Wie man letzten Endes etwas programmiert, variiert von Programmierer zu Programmierer und es wird immer mehrere gleich gute, aber selten eine perfekte Lösung zu einem Problem geben!

Lassen wir jetzt jedoch erst einmal einen Felsbrocken den Berg hinabrollen. Dazu haben wir einen MovieClip mit einem Kreis erzeugt und diesen rechts oben am „Berg“, der aus einer Linie besteht, platziert. Der MovieClip hat den Instanznamen *Fels*.

Listing 8.1: Lineare Bewegung eines Objektes

```
onClipEvent(enterFrame){
  this._x-=4;
  this._y+=2;
}
```

Wenn man dieses ActionScript auf den Fels legt, wird jener immer um zwei Pixel nach unten und vier Pixel nach links verschoben. Dies erweckt dann den Eindruck, als wenn der Felsbrocken der Berg hinunterrollt.

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/01_fels.fla*.

Damit es nicht nur bei einem Fels bleibt, dupliziert man den ihn. Es gibt verschiedene Methoden um eine vernünftige Aufteilung der Felsen auf dem Bildschirm zu erzeugen. Man kann zum einen nach Zeitabständen handeln oder nach der Position. Wir haben die letztere Variante gewählt und duplizieren jeweils einen neuen Fels, wenn ein Fels zum ersten Mal unter eine x-Position von 700 kommt. Damit dies wirklich nur einmalig pro Fels passiert, besitzt jeder die Variable `dup`, die `true` gesetzt wird, sobald dieser Fels einen neuen dupliziert hat. Sowie der neu erzeugte Fels wieder unter eine x-Position von 700 kommt, würde dieser auch wieder einen neuen Fels erzeugen.

```
onClipEvent (enterFrame) {
  this._x -= 4;
  this._y += 2;
  if (this._x < 700 && !this.dup) {
    this.dup = true;
    with (_root) {
      _root.i++;
      Fels.duplicateMovieClip("Fels" + _root.i, _root.i);
      _root["Fels" + i]._x = 1015;
      _root["Fels" + i]._y = 177;
      _root["Fels" + i].ha = i;
    }
  }
}
```

Listing 8.2: Lineare Bewegung mit Duplizieren ab bestimmter X-Position

Damit Ihnen bewusst wird, was nun eigentlich passiert, haben wir die neu erzeugten Felsen einfach einmal durchnummeriert. *ha* ist nur der Variablenname, der dem Textausgabefeld zugeordnet ist.

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o2_fels.fla*.

Wenn man sich das Beispiel einmal anschaut, erkennt man schnell, dass immer wieder neue Felsen erzeugt werden und die alten links unten aus dem Bildschirm verschwinden. Interessant dabei ist zu sehen, dass dies Flash von der Performance her gar nichts ausmacht. Man merkt also, dass Objekte, die außerhalb des Bildschirms versetzt werden, irrelevant sind. Wir haben das Beispiel einmal leicht abgeändert und sehr dicht beieinander liegende Objekte erzeugt, einfach nur um zu zeigen, wo Flash seine Schwächen hat. Man merkt sofort, dass die Performance des kompletten Systems zusammenbricht. Bitte achten Sie beim Starten dieses Beispiels darauf, dass Sie es auf langsamen Systemen wieder schnell genug schließen, da es sonst unter Umständen sogar dazu führen kann, dass der Rechner keine Befehle mehr entgegennimmt!

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o2_felsB.fla*.

Es ist natürlich sehr unschön, immer mehr Objekte zu erzeugen, auch wenn Flash dies nichts ausmacht, da diese außerhalb des Bildschirmbereiches liegen. Es ist aber doch eine Ressourcen-Verschwendung! Damit dies nicht passiert, setzen wir die Objekte, die links unten den Bildschirm verlassen, wieder nach rechts oben und duplizieren nur eine bestimmte Anzahl an Objekten.

```
onClipEvent (enterFrame) {
  this._x -= 4;
  this._y += 2;
  if (this._x < 700 && _root.i < 2 && !this.dup) {
    this.dup = true;
    _root.i++;
```

Listing 8.3: Komplettes Script für die Felsen

```
        with (_root) {
          Fels.duplicateMovieClip("Fels" + i, i);
          _root["Fels" + i]._x = 1015;
          _root["Fels" + i]._y = 177;
          _root["Fels" + i].ha=i;
        }
      }
      if (this._x < 0) {
        this._x = 1015;
        this._y = 177;
      }
    }
```

Die obere if-Bedingung wurde nur durch einen „Zähler“ erweitert, dass nun nur zwei Objekte dupliziert werden. Zudem sorgt die nachfolgende if-Bedingung dafür, dass ein Objekt, wenn es den Bildschirmbereich verlässt, seine x-Position also kleiner als o ist, wieder nach rechts oben gesetzt wird und wieder neu den Berg hinunter„rollt“.

Da der Fels aber nicht nur rollen, sondern eigentlich springen soll, muss man dies auch noch hinzufügen. Dies wollen wir hier aber mal wieder auf eine sehr einfache Weise lösen, nämlich mit einem Tweening. Sicherlich wäre dies auch mit ActionScript möglich, aber wenn man sich den Aufwand anschaut, den so ein ActionScript verursacht, und dies dann mit der Tweening-Version vergleicht, ist das Tweening doch um einiges einfacher bei fast gleicher Lösung.

Zu dem Tweening braucht man wohl keine großen Worte zu verlieren, es wird einfach nur der Fels nach oben und wieder zurück bewegt.

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/03_fels.fla*.

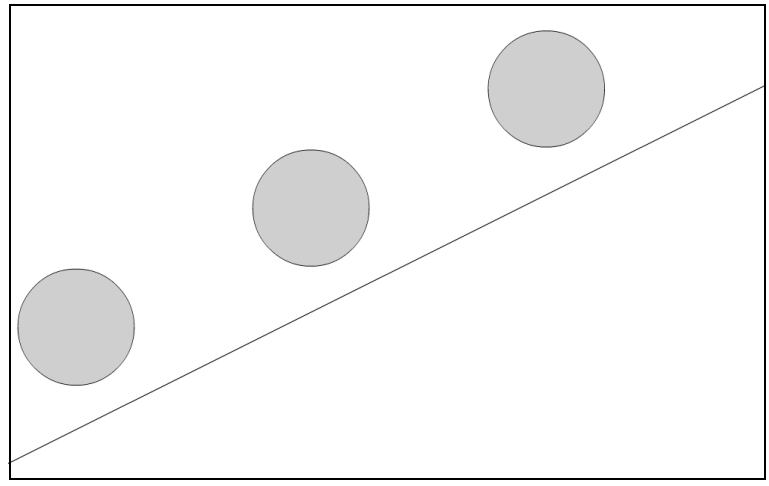


Abbildung 8.6: ***Bild des Beispiels 03_fels***

Wie man sieht, sind alle Felsen auf der gleichen Höhe in der Luft und alle besitzen auch die gleiche Größe. Damit dem nicht mehr so ist, muss man noch ein paar Werte zufällig erzeugen lassen und erhält ein Bild wie zu Beginn dieses Kapitels gezeigt.

Die entsprechenden Zeilen wären:

```
this._yscale=this._xscale=Math.random2(20,90);
this.gotoAndPlay(Math.random2(40));
```

Mit der ersten verringert man den MovieClip. Dies funktioniert hier auch nur, weil der Nullpunkt des Movies sich rechts unten direkt am Fels befindet.

Mit dem zweiten wird zu Beginn einmal zu einer anderen Stelle gesprungen, so dass nicht alle Felsen wirklich synchron „springen“.

Zu erwähnen ist auch noch, dass wir hier die selbst erstellte random-Methode benutzt haben, die am Ende des ersten Kapitels beschrieben wurde.

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o4_fels.fla*.

Wenn man nun eine Figur sich den Berg entlangbewegen lassen möchte, handelt es sich hierbei auch um eine lineare Bewegung, da sich die Figur immer gleich schnell bewegt, nachdem eine Taste gedrückt wurde.

Damit die Figur sich den Berg raufbewegt, muss man eine Funktion erstellen, die den Berg beschreibt. Dies ist in unserem Fall relativ leicht, da es sich nur um eine Gerade handelt.

Wer sich noch an den Grundaufbau einer Funktion erinnert, kommt schnell auf

```
f(x) = m * x + b;
```

Wobei m die Steigung und b die Verschiebung auf der y-Achse ist.

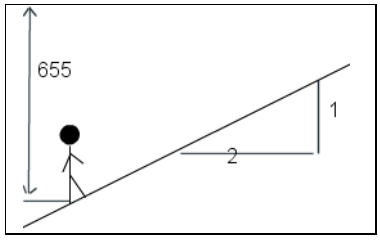


Abbildung 8.7:
Werte für die Funktion

Die Werte für die Steigung bekommt man über ein Steigungsdreieck, den Wert der Figur kann man fast direkt ablesen, statt 655 hätte man natürlich auch 768-113 rechnen können, also Bildschirmhöhe weniger die Position der Figur. Dies wirkt vielleicht ein wenig ungewohnt, da man die Koordinaten nach links oben abmisst und nicht nach links unten. Dies liegt daran, dass Flash den Nullpunkt eines Movies standardmäßig links

oben platziert, so also auch auf der Hauptbühne. Deshalb muss man später bei dem Ergebnis auch das Vorzeichen umkehren, um einen negativen Wert für y herauszubekommen.

Wenn man die Werte in eine Funktion einsetzt, würde dies wie folgt aussehen.

$$f(x) = -(½ * x + 655);$$

Eine ähnliche Berechnung haben wir übrigens auch auf unserer Internetseite *www.DieFlasher.de* benötigt, um die Würfel, die als Slider benutzt werden, richtig zu skalieren.

Nachdem man die Formel erstellt hat, kann man diese auch direkt in das ActionScript einbauen.

***Listing 8.4:* ActionScript zum Bewegen der Figur**

```
onClipEvent (enterFrame) {
  if (Key.isDown(39)) {
    _x += 2;
  }
  if (Key.isDown(37)) {
    _x -= 2;
  }
  _y = (-0.5 * _x) + 655;
}
```

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o5_figur.fla*.

Es gibt viele Möglichkeiten den Druck einer Taste abzufragen. Dies hier ist die für ein Spiel am besten geeignete, da alle anderen Abfragen über das Key-Objekt mit `onKeyDown` oder dergleichen eine Verzögerung nach dem ersten Tastendruck beinhalten! Dies ist kein Fehler, sondern so gewollt. Denken Sie daran, wenn Sie mit Word arbeiten und eine Taste länger gedrückt halten, zuerst kommt nur ein Buchstabe, dann folgen nach einem kurzen Zeitabstand die restlichen sehr schnell. Dies benötigen wir aber für das Spiel nicht!

Einziger Verbesserungspunkt wäre, die komplette Abfrage in ein selbst definiertes Intervall zu legen, da diese Abfrage in dem `ClipEvent(EnterFrame)` frameabhängig ist und dadurch nur indirekt zeithabhängig, da die Framezahl auf langsameren Rechnern sinken kann.

Mit `Key.isDown(wert)` fragt man eine bestimmte Taste ab, in unserem Beispiel die Tasten 37 und 39, also Pfeil rechts und Pfeil links.

Dieses Beispiel könnte man jetzt auch noch erweitern, so dass die Figur springen kann und sich duckt, dies würde man am leichtesten wieder mit Tweenings realisieren.

Beschleunigung

Während man im vorherigen Kapitel die x- und y-Werte einfach direkt gesetzt hat, um ein Objekt linear zu bewegen, kann man einen „ähnlichen" Trick nun auch wieder bei der Beschleunigung anwenden. Durch eine Multiplikation mit einem Wert lässt man die x- und y-Werte zum Versetzen des Objektes gleichmäßig ansteigen. Dadurch wird das Objekt jedes Mal um eine weitere Strecke versetzt und es sieht für den Betrachter so aus, als wenn es schneller würde.

Wir bauen dabei wieder auf das allererste Beispiel des Felsens auf.

Listing 8.5: Simulation einer beschleunigten Bewegung

```
onClipEvent(load){
  x=4;
  y=2;
  a=1.2;
}
onClipEvent(enterFrame){
  x*=a;
  y*=a;
  this._x-=x;
  this._y+=y;
}
```

Das Objekt in diesem Beispiel beschleunigt nun aber fast schon exponentiell und nicht linear, wie man es erwarten würde.

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o6_besch.fla*.

Für eine lineare Beschleunigung muss man sich erst einmal vor Augen führen, was denn nun eigentlich genau geschieht.

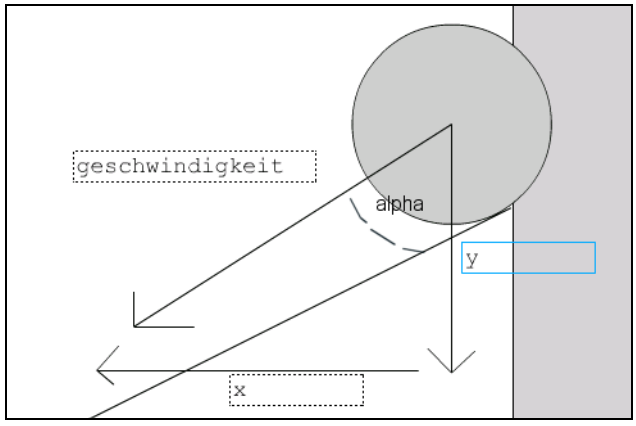


Abbildung 8.8: Geschwindigkeitsberechnung

Wie man in der Abbildung 8.8 sieht, ist die Geschwindigkeit die Hypotenuse in einem gebildeten Dreieck aus den x- und y-Werten. Dementsprechend haben wir bisher also noch gar nicht mit der Geschwindigkeit gerechnet. Dies wird aber nötig, sobald man eine Beschleunigung hat, da die Beschleunigung die Geschwindigkeit in Abhängigkeit zur Zeit immer erhöht. Wir benutzen hier als Zeit die Framerate! Dementsprechend kann man bei den Maßeinheiten der Werte auch nicht von Meter pro Sekunde reden. Da es hier aber um die Realisierung eines Spiels geht, ist dies wohl auch relativ irrelevant.

Wie schon gesagt, man muss erst einmal die Geschwindigkeit ausrechnen!

Dies geht einfach über Pythagoras:

$$a^2 = b^2 + c^2$$

Wobei a die Hypotenuse, also hier die Geschwindigkeit ist.

```
geschwindigkeit=Math.sqrt(x*x+y*y);
```

Somit hätte man die Geschwindigkeit ermittelt. Zu dieser Geschwindigkeit rechnet man bei jedem Durchlauf die Beschleunigung, in diesem Beispiel die Variable `a`, hinzu.

```
geschwindigkeit+=a;
```

Dadurch muss man nun aber die x- und y-Werte wieder neu ermitteln.

Da man zu dem Zeitpunkt nur die Werte der Hypotenuse kennt, kann man so nicht weiter rechnen. Deshalb muss man sich noch einen Wert berechnen. Wir haben uns für den Winkel *alpha* entschieden.

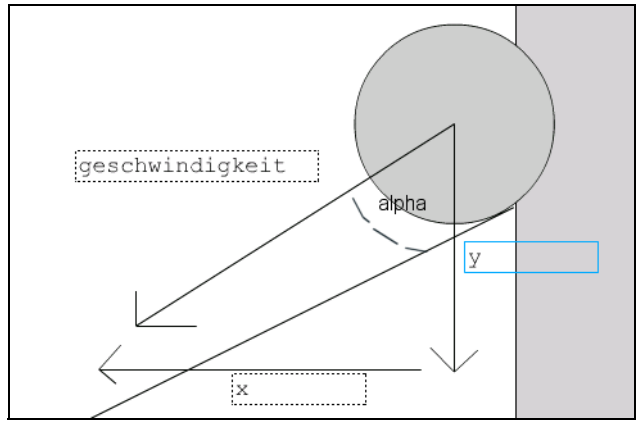


Abbildung 8.9: Geschwindigkeitsdiagramm mit Winkel alpha

Wenn man *alpha* und den Wert der geänderten Geschwindigkeit kennt, kann man über die Winkelfunktion die neuen Werte für *x* und *y* ausrechnen.

Erweiterte Mathe-Funktionen

Es sei zu erwähnen, dass wir die Mathe-Funktionen erweitert haben, so dass uns nun direkt Funktionen für die Winkelberechnung in Grad zur Verfügung stehen. Im Grunde übernehmen diese Funktionen nur die „lästige“ Umrechnung von Grad in Degree, eigentlich nur die Multiplikation mit PI durch 180.

```
Math.sind = function(wert) {return Math.sin(wert * Math.PI / 180);};
Math.cosd = function(wert) {return Math.cos(wert * Math.PI / 180);};
Math.tand = function(wert) {return Math.tan(wert * Math.PI / 180);};
Math.asind = function(wert) {return Math.asin(wert) * 180 / Math.PI;};
Math.acosd = function(wert) {return Math.acos(wert) * 180 / Math.PI;};
Math.atand = function(wert) {return Math.atan(wert) * 180 / Math.PI;};
```

Listing 8.6: Beschleunigte Bewegung

```
onClipEvent(load){
  x=4;
  y=2;
  a=2;
  geschwindigkeit=Math.sqrt(x*x+y*y);
  alpha = Math.asind(x/geschwindigkeit);
}
onClipEvent(enterFrame){
  geschwindigkeit+=a;
  x=Math.sind(alpha)*geschwindigkeit;
  y=Math.sqrt((geschwindigkeit*geschwindigkeit)-(x*x));
  this._x-=x;
  this._y+=y;
}
```

Die fla-Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o6_bescA.fla*.

Das oben aufgeführte ActionScript lässt sich aber noch leicht verbessern. Wie man sieht, wird dort der Winkel alpha ausgerechnet, dieser wird aber eigentlich gar nicht benötigt. Das zeigen die zwei Formeln, wenn man die eigentlichen Inhalte einmal durch eine Variable ersetzen würde.

```
var1= x/geschwindigkeit;
alpha = Math.asind(var1);
x=Math.sind(alpha)*geschwindigkeit;
```

Wie man sieht, wird zuerst von *var1* der *asin* berechnet um dies anschließend durch den *sin* wieder rückgängig zu machen. Dies zeigt also, dass man die *sin*-Berechnung auch direkt entfallen lassen kann. Das liegt daran, dass man für die Rechnung eigentlich nicht den Winkel benutzt, sondern nur das Verhältnis der zwei Kantenlängen zueinander.

Die verbesserte Version würde wie folgt aussehen.

Listing 8.7: Beschleunigung mit verbesserter Berechnung

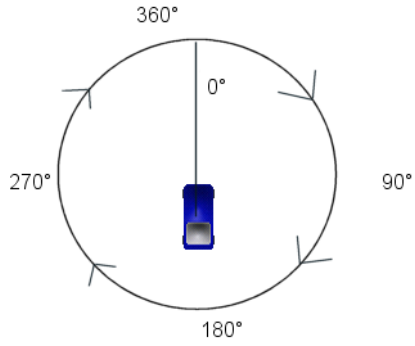
```
onClipEvent(load){
  x=4;
  y=2;
  a=2; //beschleunigung....hmmm, geht nicht !?
  geschwindigkeit=Math.sqrt(x*x+y*y);
  verh = x/geschwindigkeit;
}
onClipEvent(enterFrame){
  geschwindigkeit+=a;
  x=verh*geschwindigkeit;
  y=Math.sqrt((geschwindigkeit*geschwindigkeit)-(x*x));
  this._x-=x;
  this._y+=y;
}
```

Die fla-Datei zu dem verbesserten Beispiel finden Sie auf der CD unter *Kapitel8/06_bescB.fla*.

Rotation (Drehung)

Auch wenn Sie jetzt vielleicht erwarten, dass wir in diesem Kapitel den Fels zum Rollen bringen, werden wir dies nicht tun. Stattdessen wollen wir Ihnen zeigen, wie Sie ein Auto mittels der Pfeiltasten steuern bzw. bewegen. Diese Steuerung findet später in Kapitel 8.5 Anwendung.

Abbildung 8.10: Drehung eines Objektes



Wie man an der Grafik erkennt, wird der Winkel stets zu einer Geraden gemessen, die senkrecht nach oben verläuft. Diesen Winkel kann man über die Eigenschaft `_rotation` abfragen und setzen!

Das Drehen eines solchen Objektes ist nicht schwer. Wir haben deshalb direkt eine kleine komplette Steuerung für das Auto entworfen.

```
onClipEvent (load) {
  this.geschwin = 0;
  this.winkel = 0;
}
onClipEvent (enterFrame) {
  if (Key.isDown(Key.UP)) {
    this.geschwin += 1;
  } else if (Key.isDown(Key.DOWN)) {
    this.geschwin -= 1;
  }
  if (Key.isDown(Key.LEFT)) {
    this.winkel -= this.geschwin;
    if (this.winkel < 0) {
      this.winkel += 360;
    }
  } else if (Key.isDown(Key.RIGHT)) {
    this.winkel += this.geschwin;
    if (this.winkel > 360) {
      this.winkel -= 360;
    }
  }
  _x += Math.sind(this.winkel) * this.geschwin;
  _y -= Math.cosd(this.winkel) * this.geschwin;
  this._rotation = this.winkel;
}
```

Listing 8.8: ActionScript zur Steuerung eines Autos

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/07_rotat.fla*.

Zu Beginn werden die Variablen `geschwin` und `winkel` gesetzt, welche jeweils die aktuelle Geschwindigkeit und den Richtungswinkel enthalten. Beim Drücken der Pfeil-hoch- oder -runter-Taste wird die Geschwindigkeit um eins erhöht bzw. um eins vermindert. Gleiches geschieht für den Richtungswinkel, wobei dort zusätzlich noch dafür gesorgt wird, dass der Wert des Winkels in einem Intervall zwischen 0 und 360 bleibt. Dies wäre aber für `_rotation` nicht zwingend notwendig!

Die Werte für die x- und y-Verschiebung werden wieder über die Winkelsätze ausgerechnet.

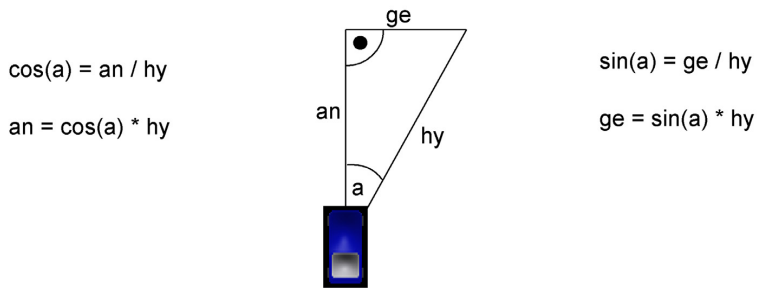


Abbildung 8.11: Berechnung der Wegstrecke über die Winkelfunktionen

Das Fahrverhalten des Fahrzeuges ließe sich natürlich noch individuell anpassen. Wir haben in dem Beispiel schon einen „kleinen“ Trick benutzt und die Drehung in Abhängigkeit von der Geschwindigkeit gesetzt.

```
this.winkel -= this.geschwin;
```

Dadurch ist es nur möglich, das Fahrzeug zu steuern, wenn es auch in Bewegung ist. Tauschen Sie die zwei Zeilen einmal gegen folgende aus.

```
this.winkel -= 1;
```

Sie merken zwar, dass das Fahrzeug einfacher zu steuern ist, aber das Fahrverhalten unrealistisch wirkt!

Das obere Beispiel hat bei dem Fahrverhalten auch noch eine kleine Eigenart, und zwar bleibt der Wenderadius immer gleich groß. Man kann so schnell fahren, wie man möchte, man benötigt nicht mehr Platz.

Wen dies stört, der kann an der Stelle eine entsprechende Funktion entwickeln, die je nach Geschwindigkeit einen größeren oder kleineren Wenderadius erzeugt.

```
this.winkel -= Math.sqrt (Math.abs(this.geschwin*4))
```

Die Datei zu dem Beispiel mit dem verbesserten Wenderadius finden Sie auf der CD unter *Kapitel8/07_rotat.fla*.

Sicherlich wäre es eventuell realistischer, wenn das Auto bei Kurven, die zu schnell gefahren werden, aus der Spur rutscht. Wir wollten Ihnen hiermit aber nur eine Einführung in das Erstellen einer Fahrzeugsteuerung geben. Sie sollten jetzt auch in der Lage sein die eigentlich Fahrphysik eines Fahrzeuges selbst bestimmen zu können.

8.2 Schwerkraft

Als Ausgangspunkt für das Einbinden einer „simulierten“ Schwerkraft soll das erste Beispiel aus Kapitel 8.1 im Abschnitt „Linear“ ab Seite 271 dienen. Dieses haben wir allerdings direkt leicht abgeändert, so dass man nun den „Ball“ per Drag&Drop bewegen kann und diesem dadurch auch eine Geschwindigkeit verleiht. Zudem haben wir noch eine sehr primitive „Kollisionsabfrage“ erstellt, so dass der Ball nicht vom Bildschirm verschwindet.

Der geänderte Quelltext sieht wie folgt aus.

Listing 8.9: Lineare Bewegung mit Drag&Drop

```
onClipEvent (load) {
  x = 1;
  y = 1;
}
onClipEvent (enterFrame) {
  if (!drag) {
    if (_x - _width / 2 < 0 || _x + _width / 2 > 1024) {
      x = -x;
    }
    if (_y + _width / 2 > 700 || _y - _width / 2 < 0  ) {
      y = -y;
    }
    this._x -= x;
    this._y += y;
  } else {
    xapos = xnewpos;
    yapos = ynewpos;
    xnewpos = _root._xmouse;
    ynewpos = _root._ymouse;
  }
}
on (press) {
  drag = true;
  startDrag(this, true);
}
on (release, releaseOutside) {
  x = xapos - xnewpos;
  y = ynewpos - yapos;
  drag = false;
  stopDrag();
}
```

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o8_gravi.fla*.

Um nun eine Schwerkraft zu simulieren, die den Ball nach unten bewegen würde, addiert man auf den y-Wert, der die y-Geschwindigkeit beinhaltet, einfach immer einen konstanten Wert.

Dadurch wird die y-Geschwindigkeit immer größer und der Ball würde sich Richtung Boden bewegen. Dort angekommen, wird die Energie umgekehrt, hier durch einen einfachen Vorzeichenwechsel, wodurch der Ball sich wieder nach oben bewegt. In dem Fall ist der Wert von y negativ. Da diesem aber noch immer ein konstanter Wert aufaddiert wird, geht y also linear gegen o, wodurch der Ball wieder langsamer nach oben fliegt, bis schließlich der Wert von y wieder ins Positive wechselt, der Ball wieder nach unten bewegt wird und die ganze Routine wieder von vorne beginnt. Hier einmal der Teil des Quelltextes, welcher sich zum vorangegangenen Code geändert hat.

```
onClipEvent (load) {
  x = 1;
  y = 1;
  g=1;
}
onClipEvent (enterFrame) {
  if (!drag) {
    if (_x - _width/2 < 0 or _x + _width/2 > 1024) {
      x = -x;
    }
    if ( _y + _width/2 > 700) {
      y = -y;
    }else{
      y=y+g;
    }
    this._x -= x;

    this._y += y;
} else {
```

An der Stelle `y=y+g;` wird die Konstante auf y bei jedem Durchlauf addiert.

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o8_grav2.fla*.

So weit, so gut. Allerdings fehlt noch ein wenig die Verlustberechnung, da es ja nicht so sein soll, dass der Ball unendlich hin- und herspringt. Am einfachsten realisiert man dies durch das simple Abziehen eines prozentualen Wertes der Geschwindigkeit.

```
if (!drag) {
  if (_x - _width / 2 < 0 or _x + _width / 2 > 1024) {
    x = -x;
  }
  if (_y + y / 2 + _width / 2 > 700) {
    y = -y;
    y -= y * 0.2;
    y -= y * Math.exp(-0.3 * Math.abs(y));
    x -= x * 0.2;
    x -= x * Math.exp(-0.5 * Math.abs(x));
  } else {
    y += g;
    y -= y * 0.01;
    x -= x * 0.01;
  }
  this._x -= x;
  this._y += y;
} else {
```

Bei jedem Durchlauf wird die Geschwindigkeit des Balls schon einmal um 1% verlangsamt. Dies soll z.B. den Luftwiderstand simulieren. Das wäre natürlich unter realen Aspekten betrachtet nicht richtig, soll aber für die bloße Simulation reichen. Im Quelltext handelt es sich dabei um die folgenden zwei Zeilen.

```
y -= y * 0.01;
x -= x * 0.01;
```

Um den Aufprallverlust zu simulieren dienen die folgenden Zeilen.

```
y -= y * 0.2;
y -= y * Math.exp(-0.3 * Math.abs(y));
x -= x * 0.2;
x -= x * Math.exp(-0.5 * Math.abs(x));
```

Die x- und y-Werte werden schon mal jeweils um 20% verringert, so dass die Geschwindigkeit eindeutig abnimmt. Die anderen zwei Zeilen mit der Logarithmusfunktion sorgen für ein realistischeres Verhalten bei kleineren Werten. Da die prozentuale Verringerung gerade im unteren, kleineren Bereich sehr stark abnimmt, sorgt dies dafür, dass der Ball sehr oft zum Schluss hin auf- und wieder abprallen würde, wie eine Eisenkugel auf einem dicken Eisenblock. Um dies zu vermeiden haben wir mit den zwei Funktionen dafür gesorgt, dass die Geschwindigkeit bei kleineren Werten stetig abnimmt.

Die erstellten Funktionen haben die Eigenschaft, für x-Werte kleiner eins von Null an exponentiell gegen 1 zu laufen (von rechts betrachtet).

Im Grunde genommen ist es auch egal, was die Funktionen nun genau machen oder welche Funktion man benutzt. Sie sollte nur jeweils nicht zu komplex werden, um die Berechnungsdauer nicht unnötig in die Höhe zu treiben, und für einen realistischen Effekt sorgen.

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/o8_grav3.fla*.

8.3 Kollisionserkennung

Die Kollisionserkennung ist das grundlegende Element der meisten Arcade-Spiele. Selbstverständlich existiert auch eine Kollisionserkennung im dreidimensionalen Bereich, wir wollen hier im Folgenden aber erst einmal „nur“ auf den 2D-Bereich eingehen und zeigen, dass vieles nicht so einfach ist, wie man sich dies vielleicht vorstellt. Wer sich das Kapitel 7 durchgelesen hat, weiß, dass das Erzeugen einer wirklichen 3D-Umgebung unter Flash nicht unbedingt leicht ist; um es nicht unnötig komplizierter zu machen, bleiben wir deshalb im 2D-Bereich. Zumal sich dieser analog zum 3D-Bereich verhält, nur eben mit einer Variable weniger.

Sobald man anfängt ein simples zweidimensionales Spiel zu programmieren, benötigt man eine Kollisionserkennung. Sei es um zu erfahren, wann eine Rakete einen Gegner trifft oder wann das Raumschiff einen Gegenstand aufsammelt oder die Wand berührt.

Man erkennt schnell, dass die Kollisionserkennung eines der grundlegenden Elemente eines Spiels ist und immer wieder abgefragt werden muss. Gerade deshalb ist es auch wichtig, dass diese Passage des Programms gut programmiert ist. Eine schlechte Programmierung würde eventuell dazu führen, dass während des Spiels unnötige Abfragen getätigt werden, die die gesamte Performance herunterziehen. Gerade in Flash, wo man bei der „normalen" Programmierung auch schon darauf angewiesen ist, auf die Performance zu achten, ist es umso wichtiger, wiederkehrende Codeabschnitte des Programms besonders sauber und gut zu programmieren.

Selbst eigentlich von der Programmierung recht „harmlos" wirkende Spiele, benötigen sobald Sie mit Flash realisiert sind, einen leistungsfähigen Computer um überhaupt flüssig zu laufen.

Es existieren zwei Gründe, warum eine Kollisionserkennung langsam laufen kann. Entweder vergleicht man zu viele Objekte miteinander, eventuell sogar Objekte, bei denen man auf einfachere Art schon vorhersagen kann, dass sich diese niemals treffen werden. Oder der eigentliche Code der Kollisionsabfrage ist zu aufwändig.

Bei Ersterem werden Sie vielleicht direkt sagen, dass Ihnen so etwas niemals passieren würde, doch es geschieht schneller, als man vermutet.

Stellen Sie sich vor, Sie haben ein Spiel mit drei Objekten. Wie viele Abfragen wären also für die Kollisionserkennung nötig?

Drei Abfragen sind nötig. Man muss Objekt 1 mit Objekt 2, Objekt 1 mit Objekt 3 und Objekt 2 mit Objekt 3 vergleichen. Alle anderen Abfragen wären überflüssig, da sie diesen entsprechen. So liefert ein Vergleich zwischen den Objekten 3 und 1 natürlich das gleiche Ergebnis wie ein Vergleich zwischen 1 und 3.

Für eine Kollisionsabfrage zwischen zwei Objekten ist also ein Vergleich notwendig, bei drei Objekten sind drei Vergleiche nötig und bei vier Objekten sechs Vergleiche.

Dies lässt sich nach folgender Formel berechnen:

$$(n^2 - n) / 2$$

n steht hierbei für die Anzahl der Objekte.

Verringerung der Kollisionsabfrage

Wie man sieht, steigt die Anzahl der nötigen Vergleiche mit der Anzahl der Objekte sehr schnell an. Dies führt in Flash sehr, sehr schnell zu einem Problem, da Flash nicht wirklich performant ist. Deshalb sollten Sie als Programmierer von Anfang an darauf achten, so wenig Kollisionsvergleiche wie möglich zu nutzen.

Ziel muss es sein, die Anzahl der Kollisionsvergleiche vorab schon so klein wie möglich zu halten. Bei normalen Kollisionsabfragen ist das Ergebnis fast immer negativ und es gibt meistens mögliche Hinweise, die darauf schließen lassen, dass dies so ist. Ihre Aufgabe besteht darin, diese zu erkennen und mit in das Spiel zu implementieren.

Sie selbst als der Erbauer des späteren Spiels sind für dessen Funktionsumfang zuständig. Es gibt verschiedene Ansätze die Anzahl der Kollisionsabfragen zu verringern.

Zum einen kann man die Objekte in verschiedene Abschnitte anhand der x- und y- Koordinaten ordnen und so von vornherein ausschließen, dass diese Objekte kollidieren.

Im Beispiel in Kapitel 8.5 benutzen wir eine ähnliche Methode im Zusammenhang mit den Menschen, die sich auf dem Spielfeld bewegen.

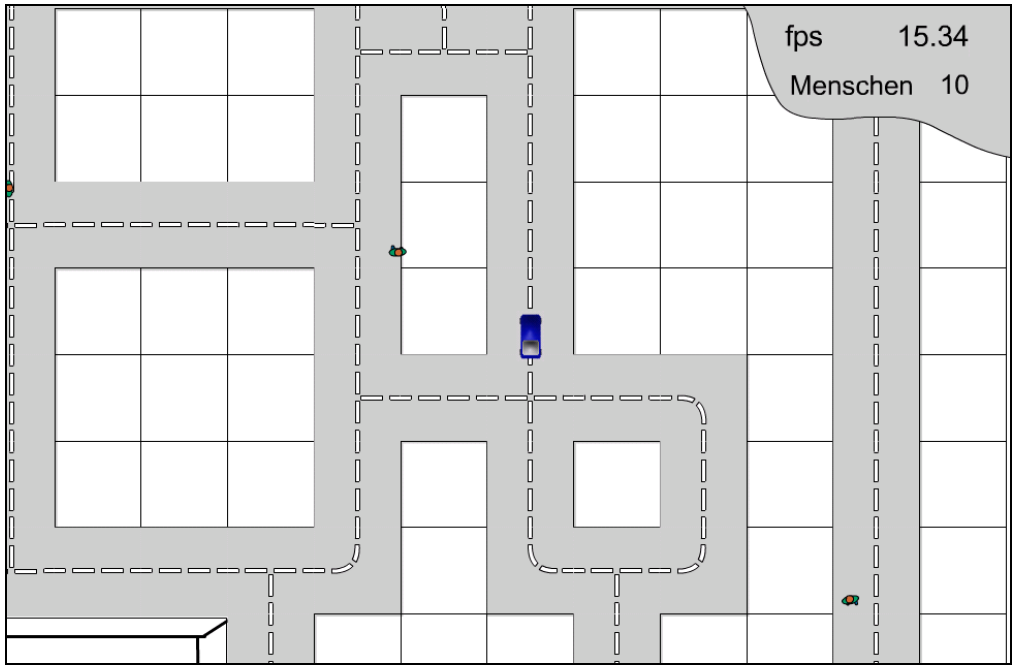


Abbildung 8.12: Bild des Spiels aus Kapitel 8.5

Jede Figur ist direkt einem Feld zugeordnet, so dass man, wenn man eine Kollisionsabfrage zwischen dem Auto und den Figuren macht, zuerst aus den umliegenden Feldern um das Auto die in Frage kommenden Figuren ermittelt und nur noch mit diesen einen Kollisionstest durchführt. Diese Methode verringert die Anzahl der Kollisionsabfragen von n auf ein Minimum, evtl. sogar auf o!

Eine weitere und durchaus einfache Methode wäre, dass Sie als Programmierer von vornherein festlegen, dass sich Objekte nicht berühren. Oder dass diese Objekte sich vielleicht berühren, aber nichts geschieht.

Letzteres wurde wieder bei dem Beispiel in Kapitel 8.5 genutzt, wenn eine Person mit einer anderen Person zusammenstößt, geschieht nichts.

Wenn man allerdings mit mehreren Objekten hantiert, z.B. bei dem Spiel Asteroid, muss jeweils immer zwischen allen Asteroiden und dem Schiff ein Kollisionstest durchgeführt werden.

Hier ist es sinnvoll, von vornherein Kollisionen auszuschließen, indem man die Objekte sortiert und mit den Tests bei den näher liegenden Objekten beginnt. So kann man evtl. ausschließen, dass dahinter liegende Objekte getroffen werden bzw. das Schiff berühren. Man muss in diesem Fall natürlich dafür sorgen, dass das Sortieren der Objekte schnell geschieht! Aber das sollte nicht das Problem sein.

Außerdem kann man vielleicht vorab schon das Eintreffen einer Kollision auf ein bestimmtes Zeitintervall einschränken, zum Beispiel, dass ein Objekt ein anderes Objekt nur alle 5 Sekunden treffen kann oder dass es nach dem Erscheinen eines neues Objektes noch 10 Sekunden dauert, bis eine Kollision möglich ist. Dies sollte man auf gar keinen Fall vernachlässigen, auch wenn man mit letzterem Effekt nur eine temporäre Verringerung erzielt hat, führt ersterer auf jeden Fall zu einer sichtlichen Verbesserung der Performance des Spiels.

Grundsätzlich sind Kollisionstests in Flash nur auf die Grenzen der Objekte basiert und nicht auf die Pixel! Wenn man unbedingt eine genauere Kollisionsabfrage braucht, sollte man wiederum zuerst über die einfachere Kollisionsabfrage testen, ob die Objekte sich überhaupt berühren, und erst wenn dies der Fall ist, kann man immer noch eine komplexere Abfrage benutzen! Man grenzt also das Zielgebiet mit Kollisionstests langsam ein und wird später immer genauer.

Schnelle Kollisionstests sind der beste Weg zu einem guten Spiel!

Rechteck

Wie schon erwähnt ist eine gute Kollisionserkennung genauso wichtig wie die geringe Anzahl der Vergleiche. Flash bringt von Haus aus direkt eine simple Kollisionsabfrage mit namens HitTest. Diese überprüft, ob zwei Objekte sich berühren. Das funktioniert allerdings nicht pixelgenau, sondern die Objekte werden zu einem kleinstmöglichen Rechteck erweitert. Dementsprechend funktioniert diese Kollisionsabfrage zwar relativ schnell, aber auch relativ ungenau. Es ist mit HitTest auch möglich, eine pixelgenaue Abfrage zu erzeugen, allerdings nur in Abhängigkeit von dem Mauszeiger. Zum Vergleich zwischen zwei Movies wird um jedes Objekt ein Rechteck gezogen. Dieses Rechteck, mit dem der HitTest ausgeführt wird, wird nicht mit gedreht, wenn man ein Objekt mit `_rotate` dreht. Stattdessen wird das Rechteck vergrößert und neu um das gedrehte Objekt gezogen.

Abbildung 8.13:
Rechteck mit richtiger Hittest-Abfrage (der graue Bereich)

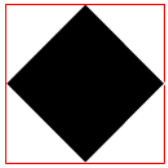


Abbildung 8.14:
Rechteck mit falscher HitTest-Abfrage

Der HitTest-Befehl existiert seit Flash 5 und bietet die Möglichkeit einer einfachen Kollisionsabfrage. Um zu zeigen, dass der Befehl nicht sehr komplex ist, wollen wir die Funktionsweise einmal „per Hand" nachprogrammieren. Um so eine Kollisionsabfrage zu bauen, benötigt man die x- und y-Position sowie die Höhe und Breite der jeweiligen Objekte. Diese Daten liefert Flash über die Objekteigenschaften _x, _y, _width und _height. Um dann eine Kollision zu erkennen vergleicht man die Position, also die Ecken der Objekte miteinander und prüft, ob diese überlappen. Die linke obere Ecke eines Objektes erhält man automatisch über _x und _y (vorausgesetzt der Nullpunkt im Objekt ist auch richtig angelegt und nicht verschoben). Den rechten unteren Punkt erreicht man über das Addieren der Breite sowie der Höhe des Objektes zu dem linken oberen Punkt.

Dementsprechend kann man alle Eckkoordinaten des Objektes berechnen. Da man dementsprechend aber auch nur die Eckpunkte auf eine Kollision abfragen wird, entsteht ein rechteckiger Kollisionsbereich und keiner, der genau dem angegebenen Objekt entspricht.

Es tritt also das oben schon beschriebene Problem auf, dass nur rechteckige Objekte abgefragt werden.

```
if (this, hitTest(_root.m2)) {
  trace("HitTest-Abfrage ergibt true");
}
if ((_root.m2._y < this._y + this._height && _root.m2._y
  + _root.m2._height > this._y) && (_root.m2._x < this._x
  + this._width && _root.m2._x + _root.m2._width > this._x)) {
  trace("Selbst erstellte Abfrage ergibt true");
}
```

Listing 8.10:
Anwendung von HitTest und eines selbst erstellten Vergleichs

Sie können dies selbst einmal testen, wir haben Ihnen ein entsprechendes File auf der CD-ROM bereitgestellt unter *Kapitel8/09_recht.fla*.

Als weiteres Beispiel zu der HitTest-Funktion finden Sie auf der CD unter *Kapitel8/09_Koll2.swf* das erweiterte Spiel aus Kapitel 8.1, Abschnitt „Linear" ab Seite 271. Die Kollisionsbereiche sind rot gekennzeichnet!

Es ist einerseits verständlich, dass Flash nur den einfachen Vergleich unterstützt, da alles andere, wie zum Beispiel ein pixelgenauer Vergleich, eine relativ aufwändige Abfrage mit sich bringen würde. Macromedia scheint in dem Punkt aber auch eher von den Benutzern zu lernen, denn die HitTest-Abfrage existiert erst seit Flash 5, zuvor wurde diese wie oben beschrieben erzeugt. Was man allerdings in diesem Punkt wirklich sagen kann, ist, dass die HitTest-Abfrage noch „leicht" zu verbessern ist und dass es wirklich schade ist, dass Macromedia dies noch nicht getan hat!

Von großem Vorteil wäre, wenn die HitTest-Abfrage auch nach dem Rotieren eines Objektes den rechteckigen Hit-Bereich mit rotiert. Dies hört sich zwar jetzt einfach an, ist aber in der Praxis dann doch schon ein wenig aufwändiger, aber keineswegs unlösbar!

Wenn man ein Objekt rotiert, kann man über die Winkelfunktionen konkret jeden Punkt errechnen.

Um dies einmal an einem Beispiel zu verdeutlichen schauen Sie sich die Datei *Kapitel8/10_vHitTe.swf* an.

In dem Beispiel haben wir ein „Auto", ein rechteckiges Objekt, das über die Pfeiltasten zu steuern ist, und ein Rechteck als Haus, das als zweites Objekt dient.

Über die „Ampel" kann man den Zustand der HitTest-Abfrage erkennen. Für die rechte obere und linke untere Ecke haben wir die HitTest-Abfrage verbessert, so dass sie richtig funktioniert. Bei den zwei anderen Kanten wird auf die Standard-HitTest-Abfrage zurückgegriffen, die in diesem Fall ein nicht befriedigendes Ergebnis liefert.

Es ist uns bewusst, dass Sie sicherlich neugierig sind, wie man so etwas realisiert, aber genau aus diesem Grund zeigen wir Ihnen jetzt erst, wie es eigentlich funktioniert. Haben Sie selbst schon versucht eine Lösung zu finden? Wie schon angedeutet, lässt sich alles über die Winkelfunktionen berechnen.

Als Erstes gibt es zwei Ansätze. Da der Mittelpunkt des Objektes für die Rotation des Autos mittig sitzen muss (die Alternative wäre um einiges umständlicher!), man zum Berechnen der Kollision aber den größten bzw. kleinsten x- und y-Wert des Objektes benötigt, muss man die Höhe und Breite des gedrehten Autos ermitteln! Wir sprechen hier von der Länge in der x- und y-Achse, die eigentliche Länge des Autos sollte sich natürlich nicht ändern!

Der erste Ansatz wäre, einfach zwei MovieClips zu verschachteln, den inneren zu rotieren, um dann über den äußeren diese Werte geschickt und einfach ablesen zu können. Diese Methode haben wir hier übrigens auch gewählt. Wenn man allerdings wirklich später eine universelle HitTest-Funktion erstellen möchte, sollte man von dieser Variante eher absehen.

Der andere Ansatz wäre, die Länge und Breite des Autos zu Beginn einmal abzufragen oder direkt in einer Variablen im ActionScript anzugeben. Darüber und über die Drehung des Autos, sprich den Winkel, welchen man über `_rotation` erhält, kann man mit Hilfe der Winkelsätze dann

die Höhe und Breite errechnen. Wobei man vorzugsweise nur einen Wert, also entweder die Höhe oder die Breite, über die Winkelsätze errechnet und den anderen Wert mit Hilfe des Strahlensatzes ermittelt, da die Berechnung mit den Winkelfunktionen (Sinus und Cosinus usw.) unter Flash nicht wirklich schnell erfolgt!

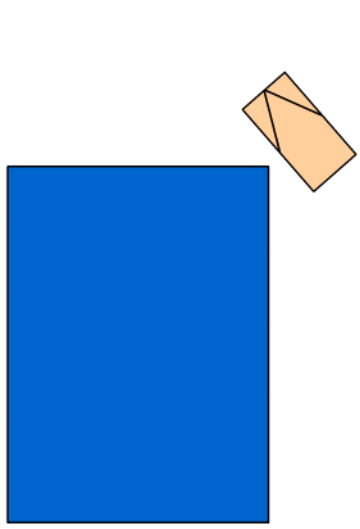


Abbildung 8.15:
Mit der verbesserten Kollisionserkennung wird erkannt, dass die Objekte sich nicht berühren.

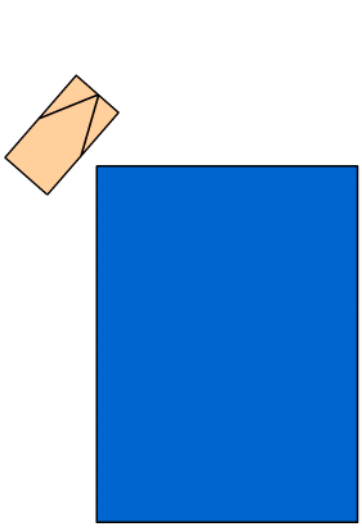
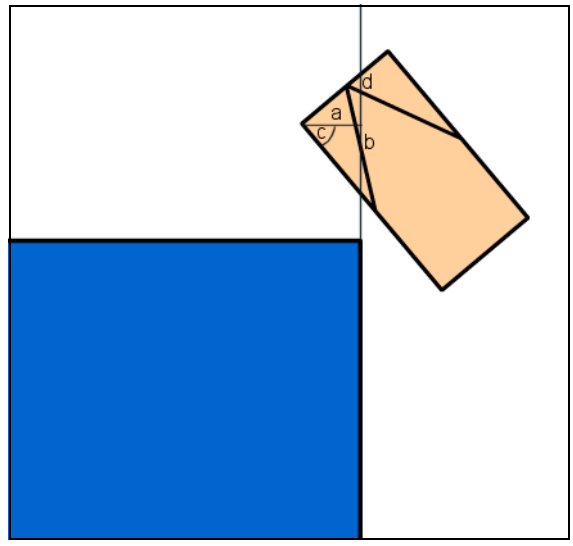


Abbildung 8.16:
Bei der Standard-HitTest-Abfrage wird fehlerhafterweise bereits eine Kollision im Randbereich erkannt!

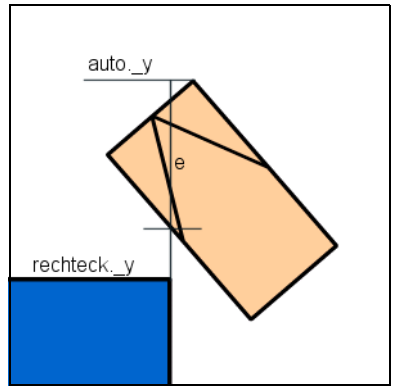
In welchem Zusammenhang `_rotation` mit der Drehung eines Objektes steht und wie man damit Berechnungen durchführt, wurde in Kapitel 8.1 im Abschnitt „Rotation (Drehung)“ ab Seite 280 verdeutlicht.

Abbildung 8.17: Berechnungsskizze für die Höhe des Autos



Da man die jeweilige x-Koordinate der rechten Kante des Rechteckes und der linken Kante des Autos kennt, kann man über die Differenz die Länge der Strecke a errechnen. Der Winkel c lässt sich über die Rotation des Autos bestimmen und zusammen mit der Strecke a kann man so die Länge der Strecke b berechnen. Als weiteren Wert benötigt man allerdings nicht die Strecke d, sondern die gesamte Höhe von der Strecke a bis zur oberen Kante des Objektes. Dies lässt sich aber einfach ermitteln, da man die Breite des Autos kennt. Danach addiert man diesen Wert mit dem Wert von b und erhält so die eigentliche Höhe des Objektes, hier als e bezeichnet, an einer bestimmten x-Position.

Abbildung 8.18: Berechnungsskizze für anschließend verbesserte Kollisionserkennung



Der nun noch nötige Vergleich ist relativ simpel, man bildet die Differenz aus `auto._y` und `rechteck._y`, und wenn dieser Wert größer sein sollte als die Länge der ermittelten Strecke e, berührt das Auto wirklich das Rechteck.

Die war jetzt allerdings nur der Vergleich für eine bestimmte Situation und für eine Ecke, das grundlegende Verfahren sollte aber deutlich erkennbar sein. Wenn man dies für die anderen Kanten auch so erzeugen würde, erhielte man eine verbesserte HitTest-Funktion. Man erkennt also, dass es möglich ist, den HitTest zu verbessern. Es kommt natürlich immer auf die Problemstellung an, welche Lösungen sich bieten.

In den folgenden Abschnitten werden wir noch auf die häufigsten und gebräuchlichsten Lösungen eingehen.

Kreis

Wie man an dem Beispiel des erweiterten Spiels aus Kapitel 8.1, Abschnitt „Linear“ ab Seite 271 erkennen kann (auf der CD unter *Kapitel8/09_Koll2.fla*), eignet sich die normale HitTest-Abfrage nicht für eine Kollisionserkennung bei runden Objekten.

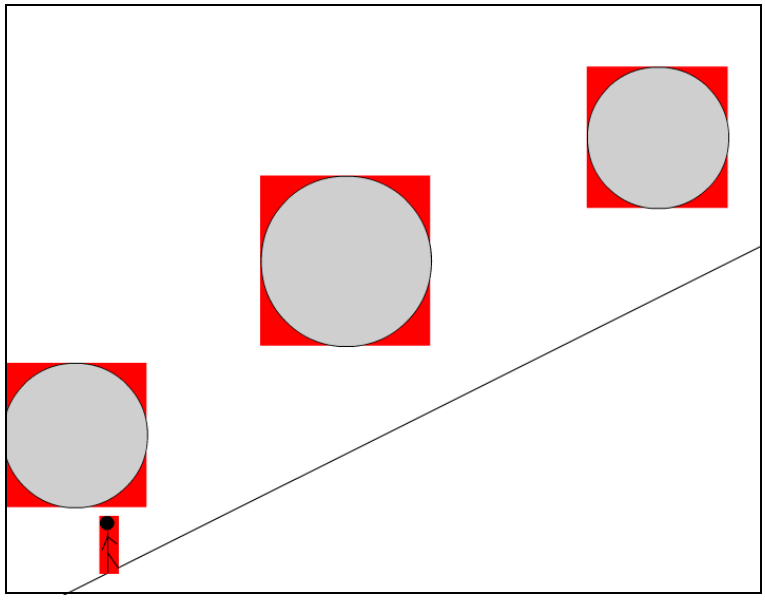
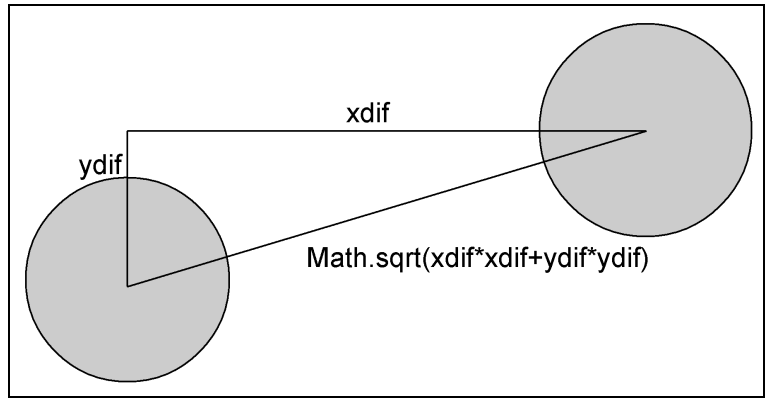


***Abbildung 8.19:** Die HitTest-Abfrage produziert auch bei runden Objekten einen rechteckige Kollisionsbereich (hier grau unterlegt).*

Speziell für runde Objekte kann man aber auf sehr einfache Art eine Kollisionsabfrage erstellen. Wenn man die zwei Mittelpunkte der entsprechenden Objekte kennt, kann man über die Division von x- und y-Position die Entfernung auf x- und y-Achse bezogen errechnen. Über Pythagoras errechnet man die Länge der direkten Entfernung zwischen den Mittelpunkten. Da man den Radius beider Kreise kennt, kann man über die Addition und den Vergleich mit der direkten Entfernung ermitteln, ob sich die zwei Kreise berühren.

Abbildung 8.20: Berechnung der Entfernung der Mittelpunkte



Im Folgenden sehen Sie das ActionScript, dort wird zuerst *xdif* und *ydif* errechnet, also die x- und y-Differenz. Anschließend werden diese Werte für die weitere Berechnung der Länge einer gradlinigen Verbindung der zwei Mittelpunkte benötigt. Über eine if-Abfrage wird dann in dem Beispiel ein Signal entsprechend gesetzt.

Listing 8.11: Kollisionserkennung für runde Objekte

```
onClipEvent (enterFrame) {
  xdif= _root.ball1._x-_root.ball2._x;
  ydif= _root.ball1._y - _root.ball2._y;
  if ( Math.sqrt(xdif*xdif+ydif*ydif)< _root.ball1._height/2
    +_root.ball2._height/2){
    _root.tr.gotoAndStop(2);
  }else{
    _root.tr.gotoAndStop(1);
  }
}
```

Die Datei zu dem Beispiel finden Sie auf der CD-ROM unter *Kapitel8/11_kreis.fla*.

8.4 Physikalische Fragen

Nach dem Feedback, das uns nach dem letzten Buch erreichte, scheint der nun folgende Exkurs in die Physik angebracht. In der Spielprogrammierung sollen sich ja viele Dinge „natürlich“ bewegen, doch diese Bahnen werden erst durch die Physik beschrieben.

Energieübertragung

Kollidieren zwei Objekte (elastisch), tauschen diese ihre Energie bzw. einen Impuls aus. Der Impuls berechnet sich aus Masse mal Geschwindigkeit, wobei die Geschwindigkeiten immer in die gleiche Richtung zeigen, das Vorzeichen sei gleichgültig. Wir haben also *M1*G1=I1* (also Masse von Objekt1 multipliziert mit Geschwindigkeit von Objekt 1 ergibt den Impuls von Objekt 1). Nun addiert man die Impulse und verteilt das Ergebnis abhängig von den Massen auf die beiden Objekte und erhält somit wieder die Geschwindigkeiten.

Fallbeschleunigung

Um Körper in einem Spiel „fallen“ zu lassen muss man auch hier die Zusammenhänge der Physik kennen.

$$\text{Fallstrecke} = \tfrac{1}{2}\, g * t^2$$

demzufolge:

$$\text{Fallgeschwindigkeit} = \tfrac{1}{2}\, g * t$$

Die Geschwindigkeit und Strecke sind parallel zur Anziehungskraft *g* (auf der Erde ~9,81 m/s^2). Von Strecke auf Geschwindigkeit lässt sich sowohl durch Herauskürzen der Zeit *t* als auch durch die erste Ableitung schließen.

Abprallwinkel

Die Berechnung des Abprallwinkels bedarf jeweils angepasster Lösungen, von daher gibt es hier auch nur wieder etwas Theorie. Treffen sich zwei Objekte in Bewegung, so zerlegt man die Bewegungsrichtungen in der Dimension entsprechend viele Einzelvektoren und errechnet die neuen Geschwindigkeiten. Prallt ein Objekt von einer Wand (unelastisch) ab, so muss man die gleiche Rechnung durchführen, doch alle Energie auf das bewegliche Objekt zurückübertragen. Da sich hier nur die Bewegungsrichtung senkrecht zur Wand ändert, jene aber parallel gleich bleibt, gilt das im Volksmund bekannte „Eintrittswinkel gleich Austrittswinkel“. Treffen Kugeln aufeinander, so gilt der Winkel der beiden Tangenten im Berührungspunkt.

8.5 OOP-Spiel

In diesem Kapitel wollen wir Ihnen anhand eines Beispiels demonstrieren, wie man ein objektorientiertes Spiel programmiert. Damit Sie eine Vorstellung des Spiels haben, schauen Sie sich dieses einmal am besten an. Sie finden es auf der CD unter *Kapitel8/12_spiel.swf*.

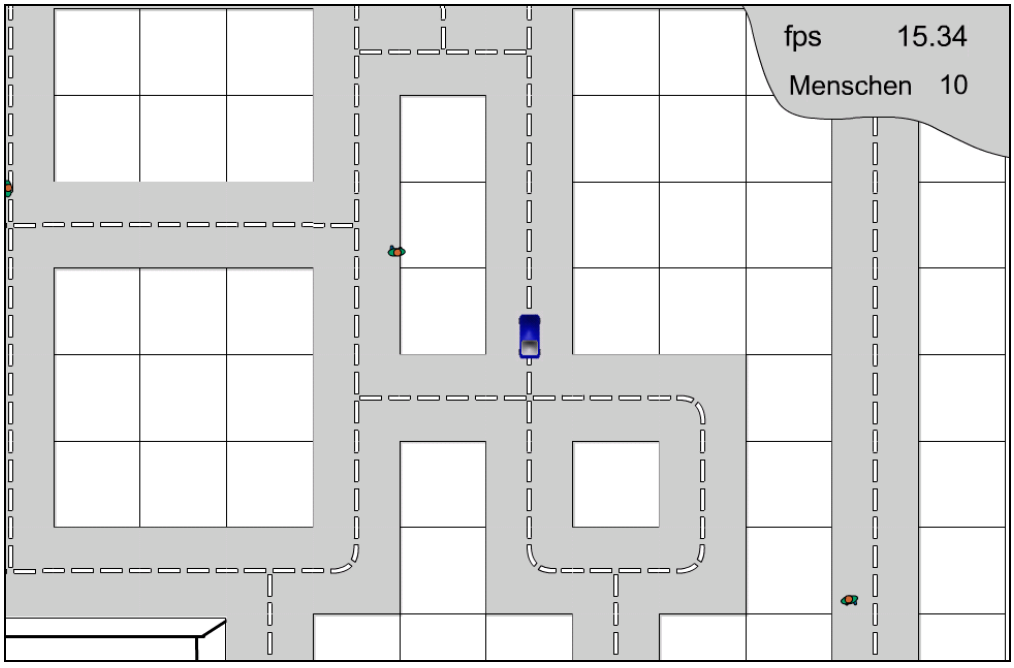


Abbildung 8.21:
Bild des Spiels

Das Auto ist durch die Pfeiltasten steuerbar. Anders als bei einer „normalen“ Steuerung bewegt sich hierbei allerdings nicht das Auto, sondern der Hintergrund. Zusätzlich befinden sich auf den Straßen noch Menschen, die sich eigenständig zufällig bewegen.

Analyse und Entwurf

Bevor man anfängt sich erste Gedanken zu der Programmierung des Spiels zu machen, erstellt man zuerst ein Pflichtenheft mit den Kann- und Muss-Kriterien. Danach kann man sich erste Gedanken zum Programmentwurf machen und ein OOA-Klassendiagramm erstellen. Geschäftsprozessdiagramme, Zustandsautomaten oder Sequenzdiagramme wollen wir an dieser Stelle einmal vernachlässigen, da die meisten Projekte in Flash noch klein genug und überschaubar bleiben dürften. Aus Gründen der Überschaubarkeit für Außenstehende und um das Projekt nicht un-

nötig zu verkomplizieren haben wir auch auf ein 3-Schichten- oder ein ModelViewController-Modell verzichtet.

Zum Erstellen des OOA-Klassendiagramms schaut man sich am besten die Grafik des Spiels noch einmal an. Welche Klassen/Objekte lassen sich daraus erkennen?

Man benötigt jeweils eine Klasse für das Auto, die Menschen, das Spielfeld und, da das Spielfeld dynamisch kreiert wird, auch noch eine Klasse für jedes einzelne Feld.

Zusätzlich haben wir noch eine Klasse für die Map, in der der Levelaufbau gespeichert ist, so dass man das Objekt einfach austauschen könnte, und direkt einen anderen Level erstellt.

Zudem besitzt unsere Realisierung noch eine Oberklasse `Spielfigur`, von der `Mensch` erbt. Dies Klasse soll alle Elemente enthalten, die alle Objekte auf dem Spielfeld gemeinsam haben. Da die einzigen beweglichen Objekte im Moment auf dem Spielfeld die Menschen sind, erscheint diese Klasse vielleicht zuerst überflüssig. Wenn man aber das Spiel noch um Tiere, Züge und andere Autos erweitern möchte, ist so eine Klasse sehr sinnvoll.

Relationen zwischen Klassen/Klassendiagrammen

Klassen stehen zueinander in Beziehung. Wenn man oben einmal auf das Beispiel eingeht, existiert eine Klasse `Spielfeld`, die das gesamte Spielfeld beschreibt, und eine Klasse `Feld`, die ein einzelnes Feld beschreibt.

Ein Spielfeld besteht aus 0 bis unendlich vielen Feldern.

Ein Feld jedoch muss immer genau einem Spielfeld zugeordnet sein.

Die Beziehung zwischen den Klassen wird durch eine Linie dargestellt, die Relationen schreibt man auf diese Linie, wobei 0 bis unendlich durch einen (*) Stern dargestellt wird.

Abbildung 8.22: Ein Spielfeld besitzt o bis unendlich viele Felder. Ein Feld ist immer einem Spielfeld zugeordnet.

Das Klassendiagramm unseres Spiels sieht wie folgt aus:

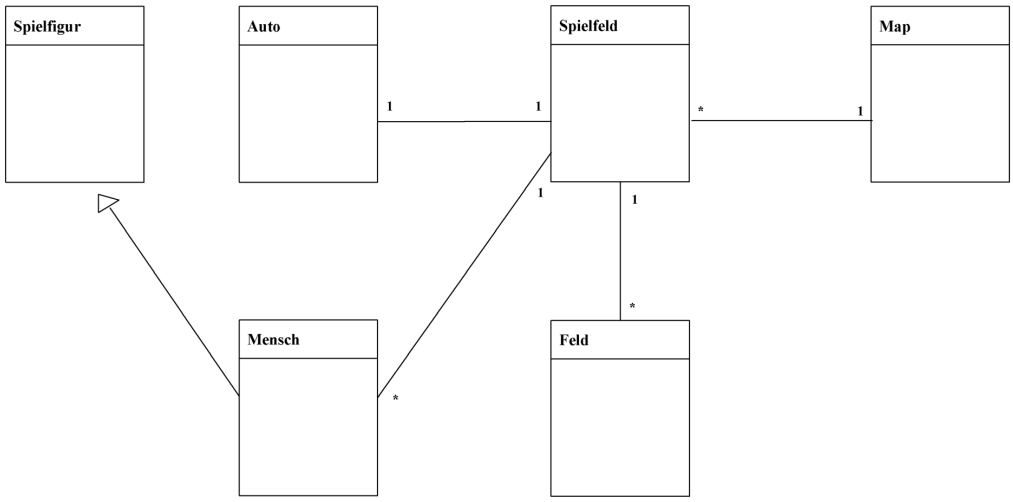


Abbildung 8.23: Klassendiagramm ohne Variablen und Methoden

Wie man sieht, besitzt die Klasse Spielfeld die meisten Beziehungen zu anderen Klassen. So besitzt diese immer genau ein Auto und eine Map und o bis unendlich viele Menschen und Felder.

Wenn man den Programmtext nach diesem Diagramm anfertigt, kann man ein paar Variablen und Arrays direkt ablesen, die man erstellt. So muss die Klasse `Feld` eine Variable enthalten, die auf das jeweilige `Spielfeld` verweist, da jedes Feld genau einem Spielfeld zugeordnet ist. Bei uns im Quelltext heißt die Variable `spielfeld`. Das Spielfeld hingegen sollte ein Array enthalten, in dem alle Felder aufgelistet sind. Dieses Array heißt bei uns `Felder`. Nach diesem Schema kann man dies für alle Klassen vollziehen. Auto enthält eine Referenzvariable auf das `Spielfeld` (Name: `spielfeld`), `Spielfeld` enthält wiederum eine Variable mit der Referenz auf das `Auto` (Name: `auto`).

In unserem Projekt haben wir dies natürlich auch genauso gelöst, allerdings ist es nicht immer notwendig, diese Verweise zu erstellen. Z.B. besitzt bei uns die Klasse Spielfeld keine Verweise auf `Mensch`, da das, wie sich herausgestellt hat, bisher nicht notwendig war. So etwas kann sich natürlich schnell ändern, wenn man das Projekt erweitert, dann müsste man diese Verweise noch nachträglich einfügen.

Standard-Bibliothek-Erweiterungen

Bevor Sie sich über neue Methoden oder Funktionsaufrufe in den folgenden Quelltexten wundern, gehen wir vorab auf die Erweiterungen ein.

Die grundlegendste ist die Methode zur Vererbung. Wir haben Ihnen diese bereits ausführlich in Kapitel 1.8 im Abschnitt „Vererbung“ ab Seite 111 vorgestellt.

Listing 8.12: Vererbungsmethode

```
Function.prototype.extend = function (obj){
    this.prototype.__proto__ = obj.prototype;
  }
```

Sie sorgt dafür, dass die angegebene Klasse von der übermittelten Klasse erbt bzw. erweitert wird.

Des Weiteren haben wir noch das Array-Objekt um eine Methode ergänzt. Ob dies unbedingt sinnvoll ist und ob man diese Methode an anderer Stelle noch einmal braucht, sei erst einmal dahingestellt. Die Methode wird auf ein zweidimensionales Array angewendet, in dem Zahlen gespeichert sind. Die Methode durchläuft dieses Array, wobei alle Felder von `startnr` beginnend durchnummeriert werden. Von allen Feldern, die kleiner als `content` sind, werden die laufenden Nummern in einem Stapel (einem Array) gespeichert und dieses zurückgegeben.

Listing 8.13: Array-Methode lessThen

```
Array.prototype.lessThen = function(content,startnr){
var stapel=new Array();
var elem=startnr;
for(i=0;i<this.length;i++){
 for(j=0;j<this[0].length;j++){
  if(this[i][j]>content){
     stapel.push(elem);
    }
    elem++;
 }
}
return stapel;
};
```

Ansonsten werden noch die erweiterte *random2*-Funktion aus Kapitel 1.8, Abschnitt „Diverse“ ab Seite 132 sowie die erweiterten Mathewinkel-Funktionen für Grad aus Kapitel 8.1, Abschnitt „Beschleunigung“ ab Seite 277 verwendet.

Die Klasse Map

Wir gehen als Erstes einmal auf die einfachste Klasse, die Klasse `Map`, ein.

Listing 8.14: Klasse Map

```
Map = function (moveMax, level) {
  this.moveMax=moveMax;
  this.level=level;
  this.playerfelder = new Array();
  this.playerfelder = this.level.lessThen(this.moveMax,1);
  this.hoehe = this.level.length;
  this.breite = this.level[0].length;
  this.felderanzahl = this.breite*this.hoehe;
};
```

Die Konstruktorfunktion der Klasse erwartet zwei Parameter, als hinteren ein Array, welches die Map beschreibt, und als vorderen eine Zahl, welche angibt, bis welche Zahl maximal die Felder betretbar sind.

Wenn man so eine Map erzeugt, muss man also zuerst ein Array erstellen, welches den eigentlichen Levelaufbau beinhaltet.

Zuerst legt man fest, welche Zahlen welche Bedeutung haben:

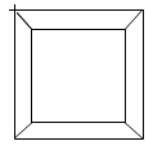


*Abbildung 8.24: 2 = 1*1 Haus*

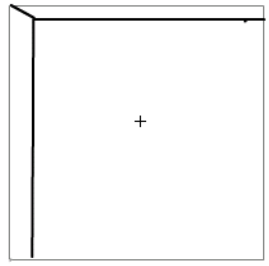


*Abbildung 8.25: 3 = 2*2 Haus*

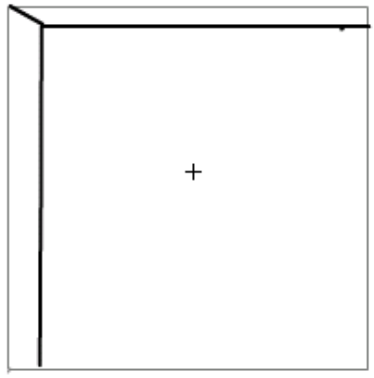


*Abbildung 8.26: 4 = 4*4 Haus*

5 = leer

Abbildung 8.27: 20 = Straße vertikal

Abbildung 8.28: 21 = Straße horizontal

Abbildung 8.29: 22 = 3 Kreuzung horizontal, oben

Abbildung 8.30: 23 = 3 Kreuzung horizontal, unten

Abbildung 8.31: 24 = 3 Kreuzung vertikal, links

Abbildung 8.32: 25 = 3 Kreuzung vertikal, rechts

Abbildung 8.33: 26 = 4 Kreuzung

Abbildung 8.34: 27 = Kurve links, oben

Abbildung 8.35: 28 = Kurve rechts, unten

Abbildung 8.36: 29 = Kurve rechts, oben

Abbildung 8.37: 30 = Kurve links, unten

Wir haben die Grafiken alle in einem Movie abgelegt und jede Grafik ist in dem der Zahl entsprechenden Frame zu finden. Wie man sieht, sind alle Objekte unter 19 für ein Auto nicht befahrbar. Dementsprechend ist die Zahl für MoveMax 19.

Es ist immer ratsam, bei allen Dingen etwas Platz zu lassen, dementsprechend sind nicht alle Zahlen unter 19 genutzt.

Nachdem man die Definitionen erstellt hat, kann man einen Level designen. Wenn man dies geschickt formatiert, erkennt man diesen Level fast schon direkt in der Array-Deklaration.

Listing 8.15: Aufbau eines Levels

```
level =
      [[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
      [ 2,28,21,21,23,21,21,21,23,23,23,21,21,21,30, 2],
      [ 2,20, 3, 5,20, 2, 2, 2,25,22,24, 2, 2, 2,20, 2],
      [ 2,20, 5, 5,20, 2, 2, 2,20, 2,20, 2, 2, 2,20, 2],
      [ 2,25,21,21,26,21,21,21,24, 2,20, 2, 2, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,20, 2,20, 2, 2, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,25,21,26,21,30, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,20, 2,20, 2,20, 2,20, 2],
      [ 2,29,23,21,22,21,21,23,27, 2,29,23,27, 2,20, 2],
      [ 2, 2,20, 4, 5, 5, 5,20, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,20, 5, 5, 5, 5,20, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,20, 5, 5, 5, 5,20, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,20, 5, 5, 5, 5,20, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,29,21,21,21,23,24, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2, 2, 2, 2, 2,25,22,21,21,21,27, 2, 2,20, 2],
      [ 2, 2, 2, 2, 2, 2,20, 2, 2, 2, 2, 2, 2, 2,20, 2],
      [ 2,28,21,21,23,21,22,21,30, 2, 2, 2, 2, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,25,21,30, 2, 2, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,20, 2,20, 2, 2, 2,20, 2],
      [ 2,25,21,21,26,21,21,21,24, 2,20, 2, 2, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,20, 2,20, 2, 2, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,25,21,26,21,30, 2,20, 2],
      [ 2,20, 2, 2,20, 2, 2, 2,20, 2,20, 2,20, 2,20, 2],
      [ 2,29,23,21,22,21,21,21,27, 2,29,23,27, 2,20, 2],
      [ 2, 2,20, 2, 2, 2, 2, 2, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,20, 2, 2, 2, 2, 2, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,20, 2, 2, 2, 2, 2, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,20, 2, 2, 2, 2, 2, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,20, 2, 2, 2, 2, 2, 2, 2, 2,20, 2, 2,20, 2],
      [ 2, 2,29,21,21,21,21,21,21,21,21,22,21,21,27, 2],
      [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]];
```

Wenn man das Array erstellt hat, kann man ein Exemplar der Klasse `Map` erzeugen.

Listing 8.16: Erstellen eines Map-Exemplares

```
map1= new Map (19, level);
```

Durch die Konstruktorfunktion wurden sofort auch alle weiteren Variablen mit erstellt, so besitzt das Objekt `map1: map1.breite, map1.hoehe, map1.playerfelder` usw.

Nach dieser Map muss nun ein Level erstellt werden. Diese Funktion soll die Klasse `Spielfeld` für uns erledigen.

Doch gehen wir zuerst auf die kleinere Klasse Feld ein.

Die Klasse Feld

Ein Feld beschreibt ein Spielfeld, wobei es sich hierbei auch um einen MovieClip handelt, da später natürlich dieses Feld auch am Bildschirm angezeigt werden soll!

Der MovieClip mit den verschiedenen Feldinhalten in den Frames, getreu der oben in der Map festgelegten Spezifikationen, befindet sich in der Bibliothek und ist verlinkt mit dem Namen `Block`.

Um diesen MovieClip in der Bibliothek (wohlbemerkt es handelt sich hierbei auch um eine Klasse!) mit unserer Klasse `Feld` zu „verbinden", reicht es, wenn dieser MovieClip von `Feld` erbt. Wie schon in Kapitel 1.8 im Abschnitt „__proto__" ab Seite 121 ausführlich beschrieben geht dies, indem man den `__proto__` -Baum per Hand verändert. Nachteil ist dabei, dass man dies wirklich selber für jedes erzeugte Objekt machen muss und dass der Konstruktor von der Klasse `Feld` nicht automatisch mit aufgerufen würde, sondern man diesen explizit ansprechen müsste.

Dies klingt nicht nur umständlich, sondern ist es auch, deshalb schafft Flash MX mit der neuen Methode `registerClass` Abhilfe. Mit dieser Methode kann man eine Klasse dem MovieClip in der Bibliothek zuordnen, und sobald man ein Exemplar dieses MovieClips bildet, wird auch ein Exemplar der verbundenen Klasse gebildet, welchem das MovieClip-Exemplar zugeordnet wird! Einzige Bedingung, die es zu beachten gilt: Die selbst geschriebene Klasse muss von der Klasse `MovieClip` erben! Ansonsten sind in dem Exemplar von MovieClip keine MovieClip-Methoden wie `gotoAndStop` oder ähnliche vorhanden!

```
Object.registerClass("Block", Feld);
```

Zuvor muss natürlich `Feld` von `MovieClip` geerbt haben. Dies geht über

```
Feld.extend(MovieClip);
```

Oder alternativ nach Standard über

```
Feld.prototype = new MovieClip;
```

Bei Letzterem wird aber ein neues „überflüssiges" Exemplar erstellt. In diesem Fall von MovieClip wäre das hier nicht weiter schlimm. Wenn man sich aber z.B. `Feld` anschaut:

```
Feld = function () {
  Feld.obj.push(this);
};
Feld.obj = new Array();
```

Listing 8.17: Klasse Feld

so haben wir diese Klasse so programmiert, dass alle erstellten Objekte automatisch in dem Array `Feld.obj` abgespeichert werden,. Wenn Feld nun mit der zuletzt beschriebenen Methode vererbt wird, würde in der Liste dieses überflüssige Objekt auftauchen, was sehr störend beim späteren Programmieren sein kann.

Die Konstruktorfunktion von `Feld` hat keine Parameter, da diese automatisch beim Erzeugen eines neuen Objektes der verbundenen Klasse `MovieClip` aufgerufen wird. Leider ist es nicht möglich, diese Werte für den Konstruktor der verbundenen Klasse mit auf den Weg zu geben, so dass man, wenn eine Klasse über `registerClass` verbunden ist und ein Objekt darüber erzeugt wird, keine Möglichkeit hat diesen Parameter für den Konstruktor mitzuliefern!

Deshalb besitzt die Klasse `Feld` auch noch eine Methode `setWerte()`.

Listing 8.18: setWerte-Methode von Feld

```
Feld.prototype.setWerte = function(spielfeld, nr, x, y, breite,
  hoehe) {
  this.spielfeld=spielfeld;
  this.breite = breite;
  this.hoehe = hoehe;
  this.zeile = Math.ceil(Nr/spielfeld.map.breite);
  this.spalte = (Nr-1)%(spielfeld.map.breite)+1;
  this.nr = nr;
  this.xpos = x;
  this.ypos = y;
  this._x = x;
  this._y = y;
};
```

Die Felder werden wie schon beschrieben von `Spielfeld` anhand der erstellten Map erstellt.

Die Klasse Spielfeld

Listing 8.19: Die Klasse Spielfeld

```
Spielfeld = function (name, map, feldbreite, feldhoehe, felddif)
  {
  Spielfeld.obj.push(this);   //Objektzähler
  this.feldhoehe=feldhoehe;
  this.feldbreite=feldbreite;
  this.map=map;
  this.name = name;
  this.movie = _root.createEmptyMovieClip(name,
    spielfeld.obj.length); // Referenz auf das Movie
  // setzt die Felder des Spielfeldes
  this.felder=new Array;
  this.setFelder(map,feldbreite,feldhoehe,felddif);
};
```

Der spätere Konstruktoraufruf dieser Klasse gestaltet sich wie folgt:

```
spielfeld1 = new Spielfeld("karte", map1, 100, 100, 0);
```

Listing 8.20: Konstruktoraufruf der Klasse Spielfeld

Als Erstes wird der Name für das erzeugte Spielfeld übergeben, danach das Map-Exemplar, die Pixelgröße der Spielfelder, also deren Breite und Höhe, sowie der Abstand zwischen den Feldern.

Die wichtigen Daten werden in dem Objekt gespeichert, zudem wird ein MovieClip erzeugt. Dies ist die spätere Bühne, auf der alles andere platziert wird. Dieser MovieClip wird für die Steuerung benutzt. Da alles auf der Bühne verschoben werden muss, wenn sich das Auto bewegt, ist es einfacher, alles in einem Movie anzuordnen und dieses Movie zu verschieben.

Die Konstruktorfunktion ruft die Methode `set.Felder` auf und übergibt an diese die Map, also den Levelaufbau, sowie die felderspezifischen Daten.

Wie man sieht, wird ein Array mit dem Namen *felder* erstellt, dieses Array wird von der `set.Felder`-Methode gefüllt.

Listing 8.21: setFelder-Methode der Klasse Spielfeld

```
Spielfeld.prototype.setFelder = function(map,breite,hoehe,dif) {
  var l = 1;
  var k = 0;
  for (var i = 0; i<map.hoehe; i++) {
    for (var j = 0; j<map.breite; j++) {
      var xpos = (breite+dif)*k;
      var ypos = (hoehe+dif)*i;
      actu = _root[this.name].attachMovie("Block", "feld"
        +(Feld.obj.length+1), 20+Feld.obj.length);
      this.felder.push(actu);
      actu.setWerte(this,l, xpos, ypos, breite, hoehe);
      actu.gotoAndStop(this.map.level[i][k]);
      k++;
      l++;
    }
    k = 0;
  }
};
```

Die Methode erzeugt sehr viele Exemplare von `Feld`. `Feld` wurde `Block` zugewiesen; durch das erstellen eines Block-Exemplares durch `attachMovie` wird automatisch auch ein `Feld`-Exemplar erzeugt.

Die Felder werden innerhalb der Bühne, innerhalb des neu erstellten Movies „Karte“ platziert. Ausgerichtet werden diese Felder durch die `setWerte`-Methode von `Feld`. Über die Zeile `actu.gotoAndStop(this.map.level[i][k])`; wird das jeweilige Feld angesprochen und veranlasst zu dem jeweiligen Frame zu springen, dessen Grafik laut `Map` (dem Levelplan) angezeigt werden soll.

Wenn man dies so weit hat, werden die Felder schon entsprechend der in der Map definierten Position auf der Bühne ausgerichtet und angezeigt.

Zusätzlich besitzt `Spielfeld` noch eine `getPos`-Methode, diese erwartet die Nummer eines Feldes und gibt die Referenz auf dieses Feld zurück.

Listing 8.22: getPos-Methode von Spielfeld

```
Spielfeld.prototype.getPos = function(nummer) {
  for (var i = 0; i<this.felder.length; i++) {
    if (this.felder[i].nr == nummer) {
      return this.felder[i];
    }
  }
  return null;
};
```

Es sieht an dieser Stelle vielleicht etwas überdimensioniert aus, da extra die ganze Liste durchsucht wird, obwohl man bei dem jetzigen Beispiel weiß, dass Feld45 in Arrayelement 44 steht, aber im späteren Verlauf werden wir nicht für alle Felder wirklich Movies erzeugen und dann ist die Liste nicht mehr vollständig. Der Suchalgorithmus ließe sich allerdings noch verbessern, eine binäre Suche ist bei einer sortierten Liste effizienter.

Die Klasse Auto

Listing 8.23: Die Klasse Auto

```
Auto = function () {
  Auto.obj.push(this);
};
Auto.obj = new Array();
```

Jedem Objekt von `Auto` kann man nur ein Spielfeld zuordnen und jedem Spielfeld auch nur ein Objekt von `Auto`. Deshalb wird beim Erstellen des `Auto`-Exemplars direkt das Spielfeld im Konstruktor mit angegeben und das Auto speichert sowohl in sich selbst als auch im Spielfeld durch die `set`-Methode die Information, dass dies das zugehörige Auto-Exemplar ist.

Da das Auto auch an einen MovieClip in der Bibliothek gehängt wird, ist die eigentliche Konstruktorfunktion zweitrangig.

Listing 8.24: Auto erbt von MovieClip und Autom wird mit der Klasse Auto verbunden

```
Auto.extend(MovieClip);
Object.registerClass("Autom", Auto);
```

Wichtiger ist die Klassenglobal-Methode `Auto.newOne`, sie erzeugt ein Exemplar von dem MovieClip *autom* und somit auch automatisch ein Exemplar von `Auto` und übergibt die entsprechenden Parameter an die `Init`-Methode von Auto `Auto`.

Listing 8.25: Klassenglobale newOne-Methode von Auto

```
auto1 = Auto.newOne(spielfeld1,_root.scrwidth/2,
  _root.scrheight/2,59);//Setzt ein Auto auf das Spielfeld
Auto.newOne = function(spielfeld, x, y, feldnummer) {
  var actu = _root.attachMovie("autom", "auto"
    + Auto.obj.length, 1000 + Auto.obj.length);
  actu.init(spielfeld, x, y, feldnummer);
  return actu;
};
```

Wie man erkennt, wird das Auto nicht auch in dem MovieClip des Spielfeldes erstellt, sondern direkt auf der Hauptbühne. Dies liegt daran, dass sich das Auto nicht mit der Bühne bewegen soll.

Die init-Methode von Auto setzt auch nur die entsprechenden Variablen des Autoobjektes und ruft die Funktion setCar auf.

Listing 8.26: Init-Methode der Klasse Auto

```
Auto.prototype.init = function(spielfeld, x, y, feldnummer) {
  this.spielfeld = spielfeld;
  spielfeld.setAuto(this);
  this.spielfeldmovie = spielfeld.movie;
  this._x = x;
  this._y = y;
  this.geschwin = 0;
  this._rotation = 0;
  this.winkel = 0;
  this.wandverz = 0.8;
  this.maxv = 14;
  this.maxr = 8;
  this.setcar(feldnummer);
  this.xpos = this.spielfeldmovie._x;
  this.ypos = this.spielfeldmovie._y;
};
```

Die setCar-Methode benutzt die getPos-Methode von Spielfeld um eine Referenz auf das Spielfeld mit der angegebenen Nummer zu bekommen.

Danach fragt es von dem Spielfeld die Eigenschaften ab und setzt das Auto (bzw. das Spielfeld) entsprechend.

Listing 8.27: setcar-Methode von Auto

```
Auto.prototype.setcar = function(feld) {
  var actu = this.spielfeld.getPos(feld);
    //gibt Referenz auf Feldobjekt zurück
  this.spielfeld.movie._x = _root.scrwidth / 2 - actu.xpos
    - actu.breite / 2;
  this.spielfeld.movie._y = _root.scrheight / 2 - actu.ypos
    - actu.hoehe / 2;
  this.feld = feld;
  //Ein Feld entspricht dem internen Maß 1
  //mittig -> plus die Hälfte
  this.zeile = actu.zeile - 0.5;
  this.spalte = actu.spalte - 0.5;
};
```

Da nicht immer direkt feststeht, wie das Auto bewegt werden soll, haben wir dies flexibler gestaltet, so dass man dem Auto, falls programmiert, verschiedene „Steuerungen“ zukommen lassen kann.

Deshalb muss man, nachdem man ein Auto-Objekt über die newOne-Methode erzeugt hat, diesem noch eine Methode für die Steuerung zuweisen.

Listing 8.28: Erzeugt ein Autoobjekt mit Steuerungsmethode

```
auto1 = Auto.newOne(spielfeld1,_root.scrwidth/2,
  _root.scrheight/2,59);
auto1.onEnterFrame=Auto.spielfeldmove;/
```

In der ersten Zeile wird ein Objekt von der Klasse `Auto` erzeugt, welches sich auf Spielfeld1 auf dem 59. Feld befinden soll, wobei das Auto auf dem Schirm mittig dargestellt wird.

Durch die zweite Zeile wird dem EnterFrame-Event des Autos die Methode `spielfeldmove` zugewiesen. Diese Methode sorgt dafür, dass durch Drücken der Pfeiltasten das Spielfeld bewegt oder das Auto gedreht wird.

Die Methode zur Steuerung, `spielfeldmove`, ist vom Grundprinzip her sehr ähnlich der in Kapitel 8.1 im Abschnitt „Rotation (Drehung)“ ab Seite 280 beschriebenen Steuerung.

Einziger Unterschied: Später wird anstatt des Autos das Spielfeld entgegengesetzt bewegt.

Listing 8.29: Der Teil der spielfeldmove-Methode, der zur Steuerung des Autos dient

```
Auto.spielfeldmove = function() {
  if (Key.isDown(Key.UP)) {
    this.geschwin += (this.maxv - this.geschwin) / 16;
  } else if (Key.isDown(Key.DOWN)) {
    this.geschwin -= (this.maxr + this.geschwin) / 16;
  }
  if (Key.isDown(Key.LEFT)) {
    if (this.geschwin < 0) {
      this.winkel += Math.sqrt(Math.abs(this.geschwin * 2));
    } else {
      this.winkel -= Math.sqrt(Math.abs(this.geschwin * 2));
    }
  } else if (Key.isDown(Key.RIGHT)) {
    if (this.geschwin < 0) {
      this.winkel -= Math.sqrt(Math.abs(this.geschwin * 2));
    } else {
      this.winkel += Math.sqrt(Math.abs(this.geschwin * 2));
    }
  }
  if (Key.isDown(Key.SPACE)) {
    this.geschwin = 0;
  }
  this.x = Math.sind(-(this.winkel)) * -(this.geschwin);
  this.y = Math.cosd(-(this.winkel)) * -(this.geschwin);
  if (this.winkel < 0) {
    this.winkel += 360;
  }
  if (this.winkel > 360) {
    this.winkel -= 360;
  }
```

Der zweite Teil der Methode dient der Kollisionserkennung mit den Feldern. Es wird dabei die Position des Autos proportional zu den Zeilen/Spalten heruntergerechnet, bis sich ein Maß von 1 ergibt, so dass darüber auch direkt die Arrayposition angesprochen werden kann. Im späteren Verlauf wird dann verglichen, ob eine Kollision erfolgt, und wenn dies

zutrifft, in welcher Richtung, um dementsprechend einen ungefähren Abprallwinkel zu berechnen.

Listing 8.30: Kollisionserkennung zwischen Auto und Feld

```
var autoz = int(this.zeile
  + this.y / this.spielfeld.feldhoehe);
var autos = int(this.spalte
  + this.x / this.spielfeld.feldbreite);
this.feld = (autoz * this.spielfeld.map.breite + autos);
stopp = false;
if (this.spielfeld.map.level[autoz][autos] >
  this.spielfeld.map.moveMax) {
  //wegfrei
  if (this.altesfeld != (autoz * this.spielfeld.map.breite
    + autos)) {
    this.altesfeld = this.feld;
  }
  this.letzteF = "";
} else {
  //Kollision
  if (this.feld - 1 == this.altesfeld
    && this.spielfeld.map.level[autoz][autos
    - 1] > this.spielfeld.map.moveMax) {
    this.block = "rechts";
    //direkte Kollision rechts
    if (this.winkel >= 90) {
      this.rechts = true;
    } else {
      this.rechts = false;
    }
    this.y *= this.wandverz;
    this.letzteF = "y";
    //aufprallwinkel bestimmen
    this.awinkelP =
      (90 - Math.abs(90 - this.winkel)) / 90 * 100;
    this.x = 0;
  } else if (this.feld - this.spielfeld.map.breite
      == this.altesfeld && this.spielfeld.map.level[autoz
      - 1][autos] > this.spielfeld.map.moveMax) {
    this.block = "unten";
    //direkte Kollision unten
    if (this.winkel >= 180) {
      this.rechts = true;
    } else {
      this.rechts = false;
    }
    this.x *= this.wandverz;
    this.letzteF = "x";
    //aufprallwinkel bestimmen
    this.awinkelP =
      (90 - Math.abs(180 - this.winkel)) / 90 * 100;
    this.y = 0;
```

```
} else if (this.feld + 1 == this.altesfeld
    && this.spielfeld.map.level[autoz][autos
    + 1] > this.spielfeld.map.moveMax) {
  this.block = "links";
  //direkte Kollision links
  if (this.winkel >= 270) {
    this.rechts = true;
  } else {
    this.rechts = false;
  }
  this.y *= this.wandverz;
  this.letzteF = "y";
  //aufprallwinkel bestimmen
  this.awinkelP =
    (90 - Math.abs(270 - this.winkel)) / 90 * 100;
  this.x = 0;
} else if (this.feld + this.spielfeld.map.breite
    == this.altesfeld && this.spielfeld.map.level[autoz
    + 1][autos] > this.spielfeld.map.moveMax) {
  this.block = "oben";
  //direkte Kollision oben
  if (this.winkel < 90) {
    this.rechts = true;
  } else {
    this.rechts = false;
  }
  this.x *= this.wandverz;
  this.letzteF = "x";
  //aufprallwinkel bestimmen
  //Abfrage für 0 oder 360
  if (this.winkel < 90) {
    this.awinkelP = (90 - Math.abs(this.winkel)) / 90 * 100;
  } else {
    this.awinkelP =
      (90 - Math.abs(360 - this.winkel)) / 90 * 100;
  }
  this.y = 0;
} else {
  //falls keine direkte Kollision erkennbar
  //ist das Auto direkt an einer Kante
  if (this.letzteF == "x") {
    // letzte Kollision war horizontal
    if ((this.spielfeld.map.level[autoz
      + 1][autos] > this.spielfeld.map.moveMax
      && this.block == "oben") ||
      (this.spielfeld.map.level[autoz
      - 1][autos] > this.spielfeld.map.moveMax
      && this.block == "unten")) {
      //nur wenn die Felder nebenan auch frei sind!
      this.y = 0;
    } else {
```

```
        stopp = true;
      }
    } else if (this.letzteF == "y") {
      //letzte Kollision war vertikal
      if ((this.spielfeld.map.level[autoz][autos
        - 1] > this.spielfeld.map.moveMax
        && this.block == "rechts") ||
        (this.spielfeld.map.level[autoz][autos
        + 1] > this.spielfeld.map.moveMax
        && this.block == "links")) {
        //nur wenn die Felder untereinander auch frei sind!
        this.x = 0;
      } else {
        stopp = true;
      }
    } else {
      //Kollision direkt mit einer Kante
      //wird direkt gestoppt
      stopp = true;
    }
  }
  //Abprallwinkel setzen
  if (this.rechts) {
    if(this.geschwin>0){
    this.winkel += 2 * this.geschwin * this.awinkelP / 100;
    this.geschwin /= 1.1;
    }else{
    //rückwärts ohne AW, müsste extra berechnet werden
    this.geschwin /= 1.1;
    }
  } else {

    if(this.geschwin>0){
    this.winkel -= 2 * this.geschwin * this.awinkelP / 100;
    this.geschwin /= 1.1;
    }else{
    //rückwärts ohne AW, müsste extra berechnet werden
    this.geschwin /= 1.1;
    }
  }
}
//Auto setzen
if (!stopp) {
  this.xpos -= this.x;
  this.ypos -= this.y;
  this.spielfeld.movie._y = this.ypos;
  this.spielfeld.movie._x = this.xpos;
  this.zeile += this.y / this.spielfeld.feldhoehe;
  this.spalte += this.x / this.spielfeld.feldbreite;
}
this._rotation = this.altwinkel = this.winkel;
```

Die Klasse Spielfigur

Wie schon kurz erwähnt gibt es als Oberklasse von `Mensch` die Klasse `Spielfigur`. Diese enthält alle Eigenschaften und Methoden, die alle Spielfiguren gemeinsam haben.

Da bisher nur der Mensch als Spielfigur existiert, ist diese Klasse auch noch nicht besonders erweitert.

Listing 8.31: Die Klasse Spielfigur

```
Spielfigur = function (spielfeld, feldnr) {
  this.spielfeld = spielfeld;
  this.setplayer(feldnr);
  Spielfigur.obj.push(this);
};
Spielfigur.obj = new Array();
```

Wie man sieht, ruft der Konstruktor die `setplayer`-Methode auf. Diese setzt alle möglichen Eigenschaften des Spielers.

Listing 8.32: setplayer-Methode der Klasse Spielfigur

```
Spielfigur.prototype.setplayer = function(feld) {
  var content = this.spielfeld.getpos(feld);
  this.zeile = content.zeile - 0.5;
  this.spalte = content.spalte - 0.5;
  this._x = content.xpos + this.spielfeld.feldbreite / 2;
  this._y = content.ypos + this.spielfeld.feldhoehe / 2;
  this.xpos = this._x;
  this.ypos = this._y;
};
```

Die `setplayer`-Methode benutzt, um die Koordinaten des Feldes zu bekommen, die `getpos`-Methode des Spielfeldes.

Da alle Klassen, bzw. Spielfiguren, später von dieser Klasse, `Spielfigur`, erben werden und es sich bei den erbenden Klassen um Klassen handelt, die mit `registerClass` an MovieClips gebunden werden, muss `Spielfigur` von `MovieClip` erben. Theoretisch müsste z.B. die Klasse `Mensch` von `MovieClip` erben. Da `Mensch` aber schon von `Spielfigur` erbt und keine direkte Mehrfachvererbung möglich ist, realisiert man dies so.

Listing 8.33: Spielfigur erbt von Movieclip

```
Spielfigur.extend(MovieClip);
```

Die Klasse Mensch

Als Letztes kann man dann noch ein paar Menschen mit auf das Spielfeld setzen. Diese Klasse ist natürlich nach dem gleichen Schema aufgebaut wie die anderen auch.

Als Besonderheit ist hier hervorzuheben, dass diese Klasse nicht direkt von `MovieClip` erbt, sondern wie schon erwähnt von `Spielfigur`, welche zuvor von `MovieClip` geerbt hat.

```
Mensch = function (x, y) {
  Mensch.obj.push(this);
};
Mensch.obj = new Array();
```

Listing 8.34: Die Klasse Mensch

Wie bereits erläutert erbt `Mensch` von `Spielfigur`, und der MovieClip `menschm` wird mit der Klasse `Mensch` verbunden.

```
Mensch.extend(Spielfigur);
Object.registerClass("menschm", Mensch);
```

Listing 8.35: Vererbung von Mensch

Die Klasse `Mensch` besitzt auch eine `newOne`-Methode, die der Funktionsweise der Methode von `Auto` ähnelt.

```
Mensch.newOne = function(spielfeld, feldnr) {
  var actu = (spielfeld.movie).attachMovie("menschm", "menschm"
    + Mensch.obj.length, 10000 + Mensch.obj.length);
  actu.init(spielfeld, feldnr);
  return actu;
};
```

Listing 8.36: newOne-Methode von Mensch

Die Methode übernimmt eine Referenz auf `Spielfeld`, auf die der Mensch gesetzt werden soll, und eine Feldnummer, auf welchem Feld der Mensch positioniert werden soll.

Kurzer Code und Übersichtlichkeit

Es gibt in Flash, so wie in vielen Programmiersprachen, sehr viele Möglichkeiten einen Quelltext kürzer zu formulieren. Viele sind der Meinung, dass die kürzeste Schreibweise auch immer die beste ist, doch sollte man abwägen, was einem mehr nutzen bringt, ein kurzer Quellcode oder die Übersichtlichkeit des Quellcodes. Sicherlich trägt ein kurzer Quellcode auch zu dessen Übersichtlichkeit mit bei, aber schauen Sie sich einmal folgendes Beispiel einer Methode an. Die genaue Funktion dieses Quelltextes im Zusammenhang mit dem gesamten Projekt soll an dieser Stelle erst einmal irrelevant sein.

```
var actu=(spielfeld.movie).attachMovie("menschm", "menschm"+1, 2000005);
actu.init(spielfeld,feldnr);
return actu;
```

Listing 8.37: Inhalt der Methode newOne von Mensch

Es wird eine Variable mit der Referenz auf das neu erzeugte Movie erstellt. Über die Variable wird anschließend die Methode `init` aufgerufen und die Referenz als Rückgabewert zurückgeliefert.

Wie man erkennt, dient diese Variable wirklich nur als Zwischenspeicher. Es ist möglich, auf die zusätzliche Variable `actu` zu verzichten, doch kann man wirklich noch auf Anhieb erkennen, welche Funktion der folgende Code haben könnte. (Die Funktionsweise ist natürlich der des oberen entsprechend!)

```
return (spielfeld.movie).attachMovie("menschm", "menschm"+1, 2000005).init(spielfeld,feldnr);
```

Dies funktioniert nur, weil `init` als Rückgabewert `this`, bzw. auch den Wert von `actu` enthält.

Trotzdem dient es wohl als abschreckendes Beispiel für eine zu kurze Schreibweise!

Es wird ein Movie von menschm erzeugt und dessen init-Methode aufgerufen.

Da das erstellte Movie von Mensch geerbt hat, ist die init-Methode, die aufgerufen hat, die von Mensch, bzw. die des Menschexemplares, das mitgebildet wurde!

Listing 8.38: Init-Methode von Mensch

```
Mensch.prototype.init = function(spielfeld, feldnr) {
  super.constructor(spielfeld, feldnr);
  this.dif=Math.random2(2000)+2000;
  this.geschwin=Math.random2(3)+2;
  this.onEnterFrame = this.move;
  return this;
};
```

Die init-Methode ruft den Oberklassenkonstruktor (super.constructor) (den Konstruktor von Spielfigur) auf und übergibt die entsprechenden Parameter. Außerdem werden noch die restlichen Daten in dem Exemplar gespeichert. Erwähnenswert ist auch noch, dass dem onEnterFrame-Event die Methode move zugewiesen wird.

Listing 8.39: Methode move von Mensch

```
Mensch.prototype.move = function() {
  //Bewegung
  if (!this.start) {
    this.startwert = getTimer();
    this.start = true;
    this.winkel = Math.random2(359) + 1;
  }
  if (this.startwert + this.dif < getTimer()) {
    this.start = false;
  }
  this._rotation = this.winkel;
  //Kollisionsabfrage ähnlich wie beim Auto, hier etwas
    vereinfacht
  this.x = Math.sind(-(this.winkel)) * -(this.geschwin);
  this.y = Math.cosd(-(this.winkel)) * -(this.geschwin);
  var z = int(this.zeile + this.y / this.spielfeld.feldhoehe);
  var s = int(this.spalte + this.x / this.spielfeld.feldbreite);
  this.feld = (z * this.spielfeld.map.breite + s);
  if (this.spielfeld.map.level[z][s] >
    this.spielfeld.map.Movemax) {
    //Bewegen
    this.xpos += this.x;
    this.ypos += this.y;
    this._y = this.ypos;
    this._x = this.xpos;
    this.zeile += this.y / this.spielfeld.feldhoehe;
    this.spalte += this.x / this.spielfeld.feldbreite;
  } else {
    //Bei Kollision drehen
    this.startwert = getTimer();
    this.winkel -= Math.random2(35) + 5;
```

```
    this.x = Math.sind(-(this.winkel)) * -(this.geschwin);
    this.y = Math.cosd(-(this.winkel)) * -(this.geschwin);
  }
  //grobe Kollisionsabfrage
  //gleich,links,rechts,drüber,drunter,oben links,oben rechts,
    unten links, unten rechts
  if(this.feld   == this.spielfeld.auto.feld ||
     this.feld   == this.spielfeld.auto.feld-1 ||
     this.feld   == this.spielfeld.auto.feld +1 ||
     this.feld   == this.spielfeld.auto.feld
       -this.spielfeld.map.breite ||
     this.feld   == this.spielfeld.auto.feld
       +this.spielfeld.map.breite ||
     this.feld   == this.spielfeld.auto.feld
       -this.spielfeld.map.breite -1 ||
     this.feld   == this.spielfeld.auto.feld
       -this.spielfeld.map.breite +1 ||
     this.feld   == this.spielfeld.auto.feld
       +this.spielfeld.map.breite -1 ||
     this.feld   == this.spielfeld.auto.feld
       +this.spielfeld.map.breite +1
     ){
    this.treffer(); //ruft genauere Kollisionsabfrage auf
    }
};
```

Zu Beginn wird über eine if-Abfrage in Kombination mit `getTimer()` ein zufällig ablaufender Zeitwert erstellt. Nach Ablauf dieser Zeit ändert die Figur zufällig ihre Bewegungsrichtung.

Die Kollisionsabfrage entspricht vom Funktionsprinzip der von der Klasse `Auto`, interessant dürfte evtl. noch die allerletzte if-Abfrage sein, welche die Methode `treffer()` aufruft. Es handelt sich hierbei um eine grobe if-Abfrage, wobei jedes Objekt die umliegenden Felder des Autos vergleicht um festzustellen, ob es sich in dessen Nähe befindet. Falls dies zutrifft, erfolgt die genauere HitTest-Abfrage in der Methode `treffer()`. Es handelt sich hierbei also um eine grobe Vorabinformationsnutzung, die die Performance etwas steigert. Ausführlich sind solche Ansätze in Kapitel „Verringerung der Kollisionsabfrage“ ab Seite 286 beschrieben.

Da man meistens mehrere Menschen auf dem Spielfeld haben möchte und diese gerne zufällig verteilt sein dürfen, existiert noch eine der Methode `newOne` übergeordnete Methode.

Listing 8.40: setToSpielfeld-Methode von Mensch

```
Mensch.setToSpielfeld = function(spielfeld, anzahl) {
  _root.spieler = anzahl;
  for (i = 0; i < anzahl; i++) {
    Mensch.newOne(spielfeld1, spielfeld.map.playerfelder
      [Math.random2(spielfeld.map.playerfelder.length)]);
  }
};
```

Übergeordnet in dem Sinne, dass diese Methode sich der `newOne`-Methode bedient. Die Methode `setToSpielfeld` erwartet eine Referenz auf das Spielfeld und die Anzahl von Menschen, die auf dieses Spielfeld gesetzt werden sollen. Damit die Menschen immer zufällig auf ein Stück Straße gesetzt werden, ermittelt die Methode über das Array `spielfeld.map.playerfelder` die Felder, die betretbar sind, und verteilt auf diesen zufällig die Menschen.

Resümee

Das Klassendesign haben wir bewusst direkt etwas umfangreicher erstellt, so kann man nun theoretisch ohne großen Programmieraufwand noch z.B. rechts oben eine verkleinerte Übersichtskarte einblenden, auf der sich auch Menschen oder ein Auto befinden, wobei sich dort dann auch evtl. das Auto bewegt und nicht der Plan. Leider zieht das eigentliche Spiel aber schon so viel an Performance, dass jedes zusätzliche Feature besser vermieden werden sollte.

Um die Performance des Spiels zu verbessern gibt es mehrere Ansätze, der einfachste und schnellste ist die Steuerung zu ändern, so dass die ganzen Spielfelder nicht bewegt werden, sondern nur das Auto. Das Spiel würde sofort flüssig laufen.

Ein anderer Ansatz wäre die Spielfelder zu vergrößern bzw. die Häuser, so dass nicht mehr so viele Objekte auf dem Bildschirm verschoben werden müssen. Dies haben wir in der Datei *13_spiel.swf* auf der CD in dem Verzeichnis *Kapitel8* gespeichert.

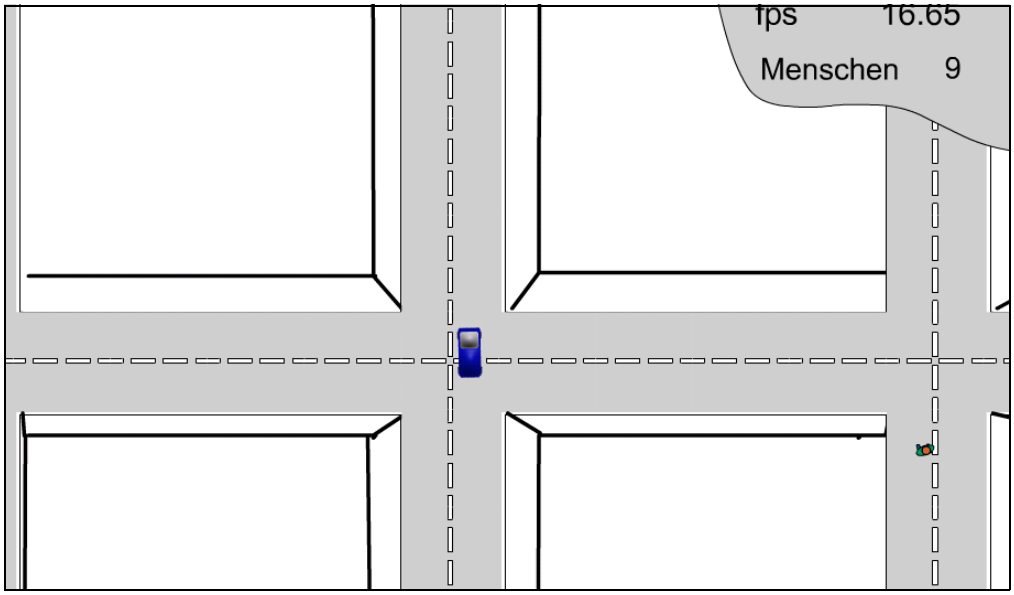


Abbildung 8.38: Leicht abgeänderte Version des Spiels, 13_spiel.swf

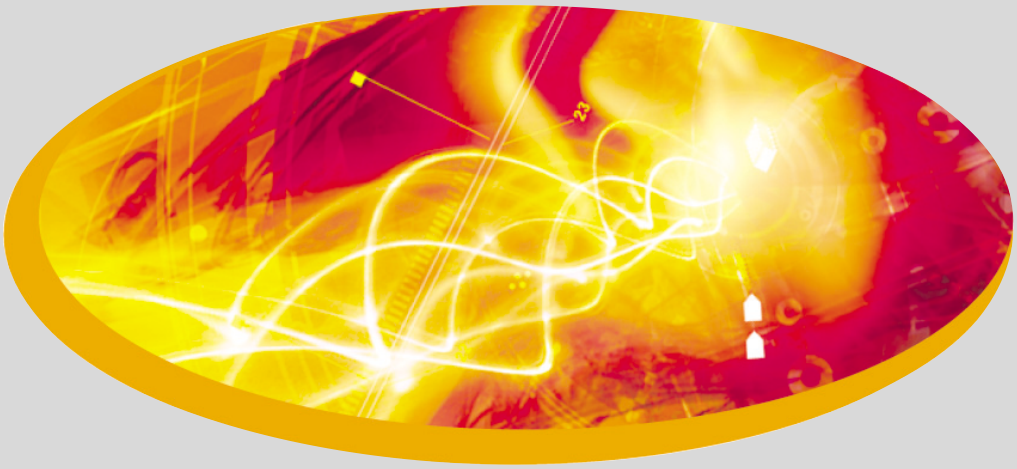
Wie man sieht, haben wir das Aussehen der Häuser leicht abgeändert und diesen durch ein kleines ActionScript mehr Dynamik verliehen, so dass sie nun dreidimensionaler wirken und sich der Position des Autos anpassen.

Die großen Häuser haben den Vorteil, dass statt der 16 (4*4) Objekte jetzt nur noch ein Objekt verschoben werden muss. Dies bringt einen leichten Geschwindigkeitsvorteil.

In der Datei *14_spiel.swf* haben wir alle Objekte gegen eine Grafik ausgetauscht, um zu zeigen, dass das eigentliche Problem das Verschieben der großflächigen Grafik darstellt.

Alle zugehörigen flas finden Sie natürlich auch auf der CD im Verzeichnis *Kapitel8:*

12_spiel.fla
13_spiel.fla
14_spiel.fla



Teil III: Arbeitserleichterungen

9 Arbeiten mit Flash

9.1 Benutzerdefinierte Oberfläche

Flash MX bietet eine neue, sehr benutzerfreundliche Option, nämlich das Speichern von Benutzeroberflächen.

Äußerlichkeiten

Mittels Drag&Drop können alle Fenster an die von Ihnen gewünschten Positionen bewegt werden.

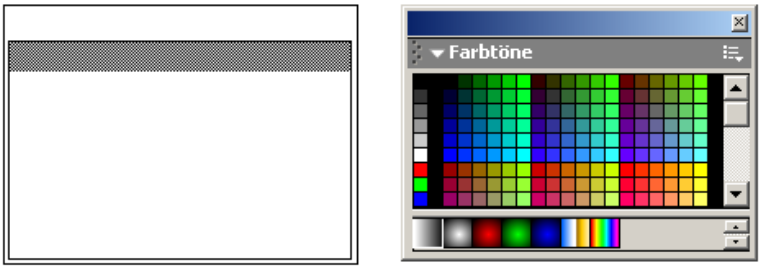
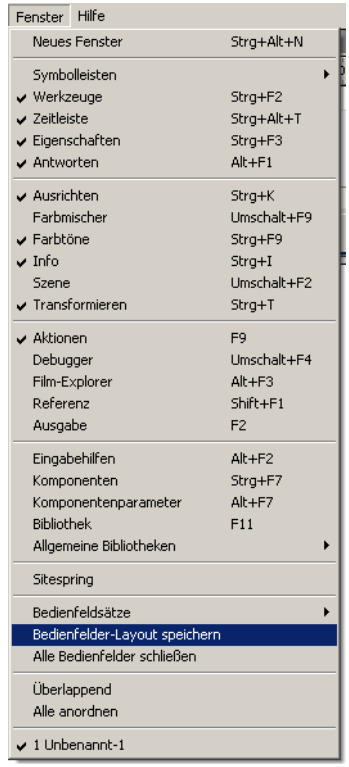


Abbildung 9.1: Zieht man ein Feld, bekommt man einen Rahmen als Anhaltspunkt, wie viel Platz dieses Feld braucht.

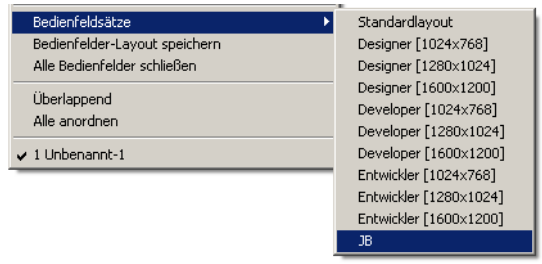
Sitzt alles perfekt, so können Sie Ihre Einstellungen abspeichern und sich so sehr viel Zeit ersparen, wenn Sie mal einen Kollegen an Ihren Arbeitsplatz lassen.

Abbildung 9.2: Nun muss nur noch ein geeigneter Name her ...



Haben Sie Ihre Einstellungen gespeichert, dann können Sie fortan diese Fensterpositionen und -größen einfach unter Bedienfeldsätze abrufen. Der Name Ihrer Einstellung ist dort nun gespeichert.

Abbildung 9.3: ... und wenn man den Platz schon mal verleiht, bieten sich Initialen an



Die inneren Werte

Flash MX bietet neben dem Speichern der Bedienfelder auch noch die Möglichkeit Shortcuts aufzunehmen. Dies kann Ihre Arbeit erheblich vereinfachen, sofern die Funktion nicht schon per Tastenkürzel zu erreichen ist.

Unter Bearbeiten › Tastenkombination findet man folgenden Dialog.

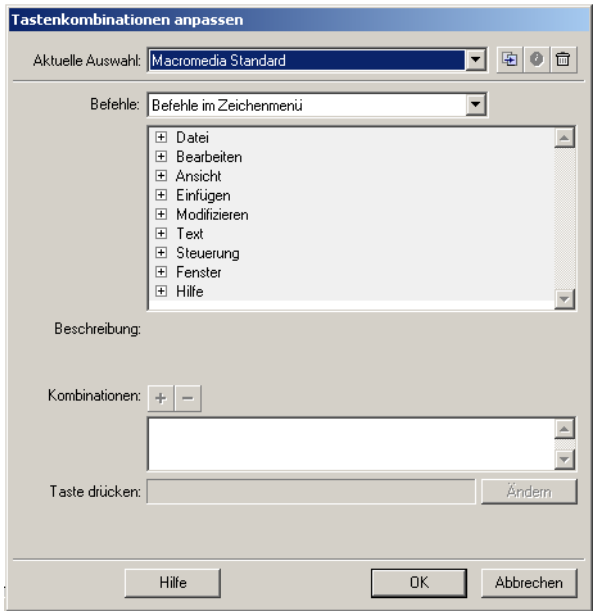


Abbildung 9.4: Hier haben Sie die Auswahl zwischen verschiedenen Sätzen

Es empfiehlt sich, an dieser Stelle eine Kopie des bestehenden Tastenlayouts zu machen und auf einer Kopie die optimalen Tastenkürzel auszuprobieren. Aus welchem Dialog der Befehl stammt, können Sie im Dropdown-Menü Befehle auswählen.

An dieser Stelle haben Sie gesehen, wie man sowohl das Aussehen als auch die Eingabe von Flash seinen individuellen Bedürfnissen anpassen kann – was will man mehr?

9.2 Der Debugger

Aufgabe und Funktion eines Debuggers ist die Überwachung von Variablen zur Nachverfolgung von Programmabläufen. Seit den frühen Flash-Versionen gibt es bereits Funktionen wie `trace()` um Werte aus Variablen nach außen zu geben, aber Aufschluss über die Anzahl der Instanzen z.B. konnte dieses einfache Werkzeug nicht geben. Seit Flash 5 gibt es hier doch umfangreichere Funktionen, die mit Flash MX noch mal ordentlich aufgestockt wurden.

Debugger starten

Um den Debugger zu nutzen müssen Sie diesem zuerst einmal gestatten in das aktive Geschehen Einblick zu nehmen.

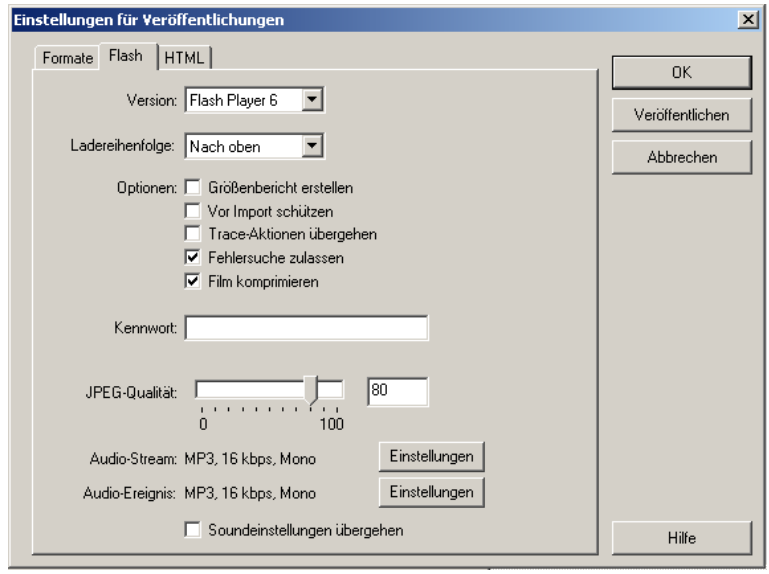


Abbildung 9.5: Einstellungen für Veröffentlichung, „Fehlersuche zulassen“

Bei dieser Einstellung ist ein Passwort optional, für den Hausgebrauch ist es aber durchaus auch in Ordnung, wenn man den Flashfilm nochmals ohne die Option „Fehlersuche zulassen“ veröffentlicht. Klicken Sie nun beim Abspielen des Films mit der rechten Maustaste in den Film, so haben Sie dort eine neue Option „Debugger“.

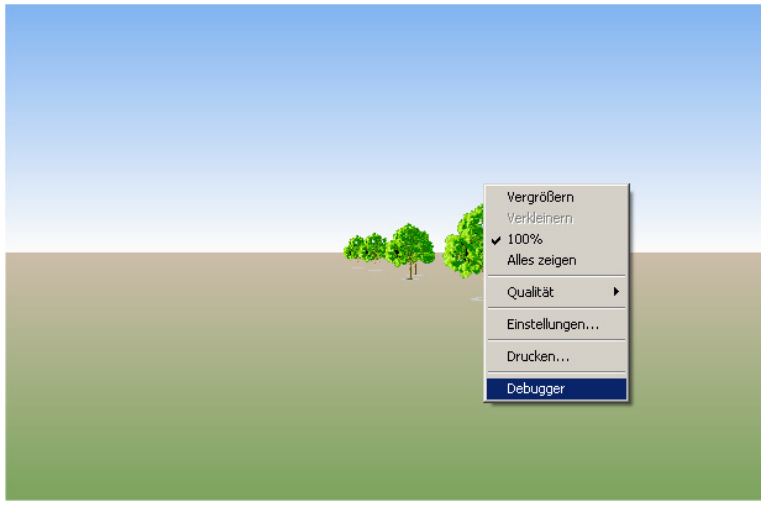


Abbildung 9.6: Fehlersuche im Wald

Wenn Sie einen Passwortschutz eingestellt haben, kommt nach der Auswahl „Debugger“ ein Dialogfeld, in dem Sie nach dem Passwort gefragt werden. Haben Sie dieses eingegeben, werden Sie folgende Debuggeroberfläche vorfinden.

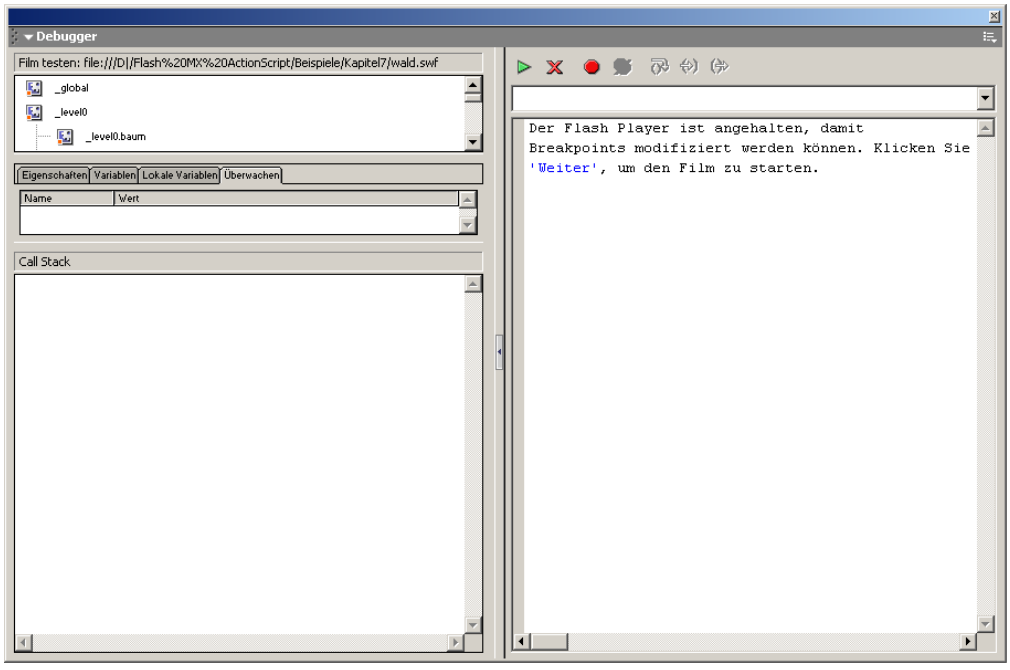


Abbildung 9.7: So gestaltet sich die Oberfläche des Debuggers.

Dort können Sie einzelne Werte Ihrer Filminstanzen überwachen und sehr einfach nachvollziehen, ab wann Werte aus von Ihnen erwarteten Bereichen kommen oder ob Instanzen überhaupt mit dem richtigen Namen erzeugt wurden und wo Sie diese dann finden.

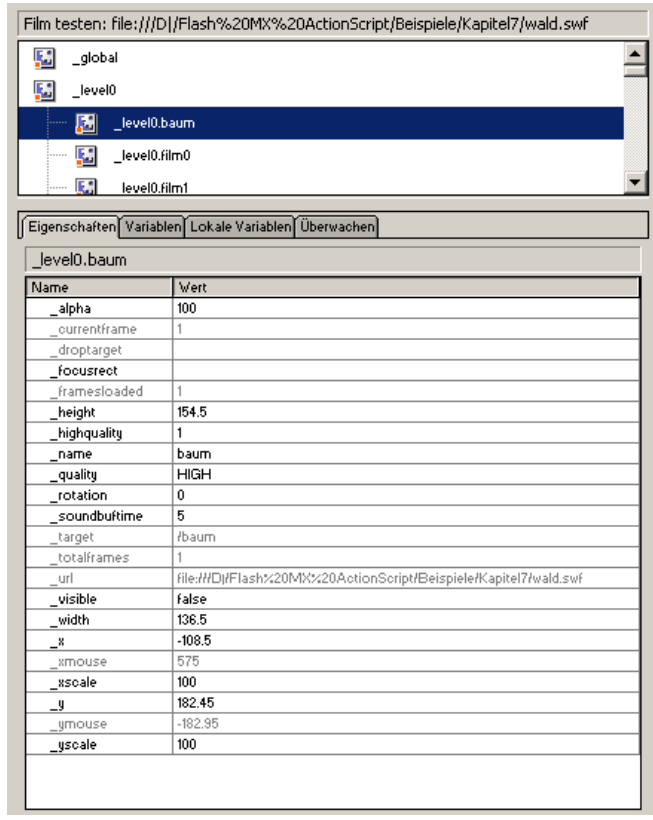


Abbildung 9.8: Fehlerüberwachung des Waldes. Hier kann man alle Standardeigenschaften des Baumes ablesen.

Natürlich kann es auch sein, dass Sie erst auf ein bestimmtes Ereignis warten um dort gezielt Werte abzufragen, aber auch hier kann Ihnen geholfen werden. Die Funktion der Breakpoints erweist sich hier als sehr nützlich, der Film, um genauer zu sein das ActionScript, wird an der Stelle des Breakpoints angehalten. So ist es Ihnen möglich, den Film bis zu einer Bedingung laufen zu lassen und erst bei Bedarf anhalten zu lassen.

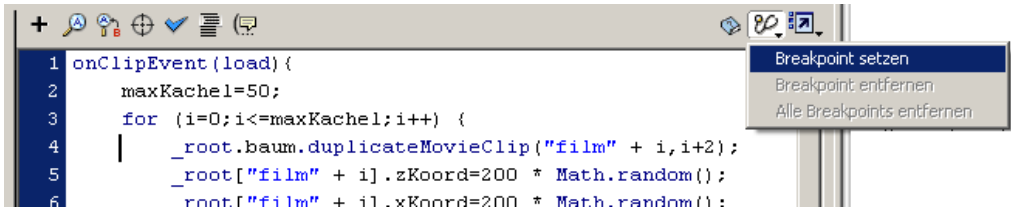


Abbildung 9.9: Im Dialog des ActionScript-Fensters haben Sie die Möglichkeit Breakpoints zu setzen.

Haben Sie den Debugger gestartet, wird bei Erreichen des Breakpoints das Script angehalten und im Debuggerdialog angezeigt.

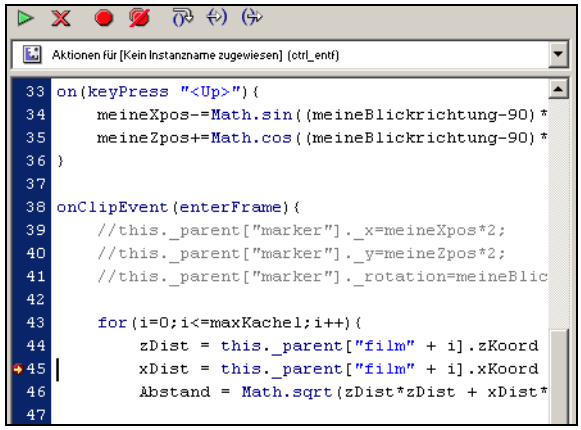


Abbildung 9.10: Breakpoints sind mit roten Kreisen in der Zeilennummerleiste markiert.

Nun kann man alle Werte, die dort mitgeladen werden, auswerten und ggf. auf die Fehler zurückschließen. Diese Liste der Variablen ist im Gegensatz zu den Eigenschaften natürlich von Ihrer Programmierung abhängig. Um diese allerdings zu erhalten sollte Ihrer Instanz ein Name zugeordnet sein, da die Werte sonst nicht überwacht werden können.

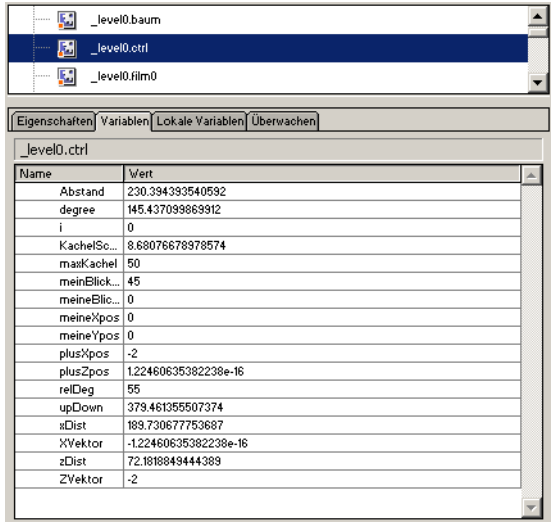
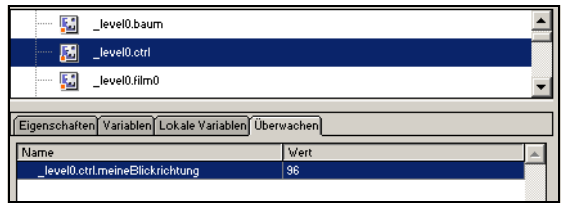


Abbildung 9.11: Liste der Variablen der Instanz

Um alle wichtigen Werte im Überblick zu halten können Sie diese auch überwachen lassen, das sieht dann wie folgt aus:

Abbildung 9.12: Ein Nachteil: Variablen werden absolut referenziert



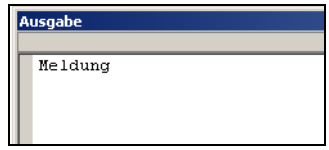
Ein Hauch von Nostalgie

Wem dieser Debugger dann für einfache Start-Stopp-Scripts zu überdimensioniert erscheint, der kann immer noch auf das gute alte `trace()` zurückgreifen. Innerhalb des Testmodus wird ein Popup erscheinen mit den übergebenen Werten. Die Syntax

```
trace("Meldung");
```

führt zu folgendem Ergebnis:

Abbildung 9.13: Eine alte Methode des Debuggings



Sollte das Script aber fortlaufende Werte erzeugen, die Sie überwachen wollten, ist klar vom trace abzuraten, da diese Werte meist so schnell vorbeilaufen, dass man zu oft den entscheidenden Punkt verpasst.

9.3 ASNative

ASNative ist ein von Macromedia nicht dokumentiertes Feature von Flash. Es handelt sich dabei um Funktionen, die vor dem eigentlichen ActionScript in Flash stehen. Alle Funktionen, Methoden, Objekte in Flash selber wurden darüber erzeugt. Dadurch, dass man selber mit ASNative programmiert, ist es also möglich, auch alle vorhandenen Funktionen zu überschreiben bzw. zu ändern. ASNative ist undokumentiert und dessen nutzbare Anwendung meist etwas schwierig aufgrund der fehlenden Informationen.

Ein kleines Beispiel:

```
ASSetPropFlags(MovieClip.prototype,null,0,1);
for(a in MovieClip.prototype)
trace (a + " "+this[a]);
```

Listing 9.1: Kurzes Beispiel für einen ASNative-Befehl

Bildschirmausgabe:

```
createTextField [type Function]
clear [type Function]
endFill [type Function]
lineStyle [type Function]
curveTo [type Function]
lineTo [type Function]
moveTo [type Function]
beginGradientFill [type Function]
beginFill [type Function]
createEmptyMovieClip [type Function]
stopDrag [type Function]
startDrag [type Function]
removeMovieClip [type Function]
duplicateMovieClip [type Function]
gotoAndStop [type Function]
gotoAndPlay [type Function]
prevFrame [type Function]
nextFrame [type Function]
stop [type Function]
play [type Function]
setMask [type Function]
getDepth [type Function]
attachVideo [type Function]
attachAudio [type Function]
getBytesLoaded [type Function]
getBytesTotal [type Function]
getBounds [type Function]
hitTest [type Function]
globalToLocal [type Function]
localToGlobal [type Function]
swapDepths [type Function]
attachMovie [type Function]
loadMovie [type Function]
loadVariables [type Function]
unloadMovie [type Function]
getURL [type Function]
meth [type Function]
tabIndex
enabled true
useHandCursor true
__proto__ [object Object]
constructor [type Function]
```

Wenn Sie dieses Script ausführen, gibt Ihnen Flash alle Methoden der Klasse `MovieClip` aus, z.B. auch die nicht dokumentierte Methode `meth( )`.

Dies würde einem allerdings nicht viel weiterhelfen, da man nicht weiß, was diese Methode macht.

Um die Neugier zu befriedigen sei gesagt, dass dies lediglich eine interne Methode von Flash ist, die benutzt wird, um zu überprüfen, ob die POST- und GET-Anweisungen richtig gesetzt wurden.

Wir haben uns in dem Beispiel der Funktion `ASSetPropFlags` bedient. Im ersten Kapitel haben wir die Funktion auch schon benutzt um die selbst geschriebenen Methoden vor for-in-Schleifen zu verstecken. Hier sind wir nun den umgekehrten Weg gegangen. Eine genauere Dokumentation zu dem Befehl finden Sie unter:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?ASSetPropFlags

ASNative bietet aber direkt über eine Matrix, welche über eine Funktion angesprochen werden kann, Zugang zu sämtlichen dokumentierten Funktionen von Flash und noch mehr!

So ist es möglich, die dritte Maustaste oder auch das Mausrad abzufragen. Leider funktioniert dies nie sehr zuverlässig oder gar nicht bei Macintosh-Usern, so dass dies wohl auch der Grund ist, warum Macromedia solche Features nicht direkt mit implementiert hat.

Im Folgenden sehen Sie den Aufruf der `Trace()`-Funktion über ASNative:

Listing 9.2: Aufruf der Trace-Funktion über ASNative

```
var1 = ASNative(100, 4);
var1("Hallo");
ASNative(100, 4)("Welt");
```

Bildschirmausgabe:

```
Hallo
Welt
```

Im ersten Teil wurde eine Referenzvariable auf den ASNative-Befehl (100,4) erzeugt. Dies ist der Trace-Befehl. Im zweiten Teil wurde der Befehl direkt mit dem entsprechenden Parameter aufgerufen.

Weitere Informationen zu ASNative-Befehlen finden Sie unter:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?ASNative

Weitere undokumentierte Funktionen von Flash MX:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?Undocumented%20Features

Anhang

A.1 Tastaturcode

▶ **Buchstaben von A bis Z und Ziffern 0 bis 9:**

Taste	Code	Taste	Code
A	65	0	48
B	66	1	49
C	67	2	50
D	68	3	51
E	69	4	52
F	70	5	53
G	71	6	54
H	72	7	55
I	73	8	56
J	74	9	57
K	75		
L	76		
M	77		
N	78		
O	79		
P	80		
Q	81		
R	82		
S	83		
T	84		
U	85		
V	86		
W	87		
X	88		
Y	89		
Z	90		

▶ **Tasten auf dem numerischen Ziffernblock:**

Tasten	Code	Tasten	Code
0	96	8	104
1	97	9	105
2	98	Multiplizieren	106
3	99	Plus	107
4	100	Eingabe	108
5	101	Minus	109
6	102	Komma	110
7	103	Dividieren	111

▶ **Funktionstasten:**

Tasten	Code	Tasten	Code
F1	112	F7	118
F2	113	F8	119
F3	114	F9	120
F4	115	F10	121
F5	116	F11	122
F6	117	F12	123

▶ **Andere Tasten:**

Tasten	Code	Tasten	Code
←	8	→	39
⇆	9	↓	40
Entf	12	Einfg	45
↵	13	Rückschritt	46
⇧	16	Hilfe	47
Strg	17	Num	144
Alt	18	;:	186
⇩	20	=+	187
Esc	27	-_	189
Leertaste	32	/?	191
Bild ↑	33	`-	192
Bild ↓	34	[{	219
Ende	35	|	220
Pos1	36	]}	221
←	37	“’	222
↑	38		

- **Schlüsselwörter in Flash:**

break	else	instanceof	typeof
case	for	new	var
continue	function	return	void
default	if	switch	while
delete	in	this	with

A.2 Operatorliste

In dieser Tabelle sind alle ActionScript-Operatoren und ihre Assoziativität von der höchsten bis zur niedrigsten Priorität aufgeführt.

Operator	Beschreibung	Assoziativität
höchste Priorität		
+	Unäres Plus	Von rechts nach links
-	Unäres Minus	Von rechts nach links
~	Bit-Komplement	Von rechts nach links
!	Logisches NOT	Von rechts nach links
not	Logisches NOT (wie in Flash 4)	Von rechts nach links
++	Post-Inkrement	Von links nach rechts
--	Post-Dekrement	Von links nach rechts
( )	Funktionsaufruf	Von links nach rechts
[ ]	Array-Element	Von links nach rechts
.	Strukturelement	Von links nach rechts
++	Prä-Inkrement	Von rechts nach links
--	Prä-Dekrement	Von rechts nach links
new	Objekt zuweisen	Von rechts nach links
delete	Objektzuweisung aufheben	Von rechts nach links
typeof	Objekttyp	Von rechts nach links
void	Gibt einen undefinierten Wert zurück	Von rechts nach links
*	*	Von links nach rechts
/	/	Von links nach rechts
%	Modulo	Von links nach rechts
+	+	Von links nach rechts
add	Stringverkettung (früher &)	Von links nach rechts
-	-	Von links nach rechts
<<	Bitweise Verschiebung nach links	Von links nach rechts
>>	Bitweise Verschiebung nach rechts	Von links nach rechts

Operator	Beschreibung	Assoziativität
>>>	Bitweise Verschiebung nach rechts (ohne Vorzeichen)	Von links nach rechts
<	Kleiner als	Von links nach rechts
<=	Kleiner als oder gleich	Von links nach rechts
>	Größer als	Von links nach rechts
>=	Größer als oder gleich	Von links nach rechts
lt	Kleiner als (für Strings)	Von links nach rechts
le	Kleiner oder gleich (für Strings)	Von links nach rechts
gt	Größer als (für Strings)	Von links nach rechts
ge	Größer oder gleich (für Strings)	Von links nach rechts
==	Gleich	Von links nach rechts
!=	Ungleich	Von links nach rechts
eq	Gleich (für Strings)	Von links nach rechts
ne	Ungleich (für Strings)	Von links nach rechts
&	Bitweises AND	Von links nach rechts
^	Bitweises XOR	Von links nach rechts
|	Bitweises OR	Von links nach rechts
&&	Logisches AND	Von links nach rechts
and	Logisches AND (Flash 4)	Von links nach rechts
||	Logisches OR	Von links nach rechts
or	Logisches OR (Flash 4)	Von links nach rechts
?:	Bedingt	Von rechts nach links
=	Zuweisung	Von rechts nach links
*=, /=, %=, +=, -=, &=, |=, ^=, <<=, >>=, >>>=	Zusammengesetzte Zuweisung	Von rechts nach links
,	Mehrfache Auswertung	Von links nach rechts
niedrigste Priorität		

▶ ASCII-Zeichensatz inklusive Latin I

Code	Zeichen	Code	Zeichen	Code	Zeichen
			-	32	Leerzeichen
33	!	34	"	35	#
36	$	37	%	38	&
39	'	40	(	41	)
42	*	43	+	44	,
45	-	46	.	47	/
48	0	49	1	50	2
51	3	52	4	53	5

Code	Zeichen	Code	Zeichen	Code	Zeichen
54	6	55	7	56	8
57	9	58	:	59	;
60	‹	61	=	62	›
63	?	64	@	65	A
66	B	67	C	68	D
69	E	70	F	71	G
72	H	73	I	74	J
75	K	76	L	77	M
78	N	79	O	80	P
81	Q	82	R	83	S
84	T	85	U	86	V
87	W	88	X	89	Y
90	Z	91	[	92	\
93	]	94	^	95	_
96	`	97	a	98	b
99	c	100	d	101	e
102	f	103	g	104	h
105	i	106	j	107	k
108	l	109	m	110	n
111	o	112	p	113	q
114	r	115	s	116	t
117	u	118	v	119	w
120	x	121	y	122	z
123	{	124	|	125	}
126	~	127	-	128	_
129	_	130	‚	131	ƒ
132	„	133	…	134	†
135	‡	136	ˆ	137	‰
138	≤	139	‹	140	Œ
141	_	142	_	143	_
144	_	145	‘	146	’
147	“	148	”	149	•
150	–	151	—	152	˜
153	™	154		155	›
156	œ	157	_	158	_
159	Ÿ	160		161	¡
162	¢	163	£	164	¤
165	¥	166	√	167	§
168	¨	169	©	170	ª

Code	Zeichen	Code	Zeichen	Code	Zeichen
171	«	172	¬	173	-
174	®	175	¯	176	°
177	±	178	Δ	179	ffl
180	´	181	µ	182	¶
183	·	184	¸	185	
186	º	187	»	188	≠
189	∞	190	≥	191	¿
192	À	193	Á	194	Â
195	Ã	196	Ä	197	Å
198	Æ	199	Ç	200	È
201	É	202	Ê	203	Ë
204	Ì	205	Í	206	Î
207	Ï	208	ð	209	Ñ
210	Ò	211	Ó	212	Ô
213	Õ	214	Ö	215	Σ
216	Ø	217	Ù	218	Ú
219	Û	220	Ü	221	∫
222	Ω	223	ß	224	à
225	á	226	â	227	ã
228	ä	229	å	230	æ
231	ç	232	è	233	é
234	ê	235	ë	236	ì
237	í	238	î	239	ï
240	Π	241	ñ	242	ò
243	ó	244	ô	245	õ
246	ö	247	ffi	248	ø
249	ù	250	ú	251	û
252	ü	253	fi	254	fl
255	ÿ				

▶ Unicode Tabelle

Unicode	Zeichen
0x00	NULL
0x01	START OF HEADING
0x02	START OF TEXT
0x03	END OF TEXT
0x04	END OF TRANSMISSION
0x05	ENQUIRY

Unicode	Zeichen
0x06	ACKNOWLEDGE
0x07	BELL
0x08	BACKSPACE
0x09	HORIZONTAL TABULATION
0x0A	LINE FEED
0x0B	VERTICAL TABULATION
0x0C	FORM FEED
0x0D	CARRIAGE RETURN
0x0E	SHIFT OUT
0x0F	SHIFT IN
0x10	DATA LINK ESCAPE
0x11	DEVICE CONTROL ONE
0x12	DEVICE CONTROL TWO
0x13	DEVICE CONTROL THREE
0x14	DEVICE CONTROL FOUR
0x15	NEGATIVE ACKNOWLEDGE
0x16	SYNCHRONOUS IDLE
0x17	END OF TRANSMISSION BLOCK
0x18	CANCEL
0x19	END OF MEDIUM
0x1A	SUBSTITUTE
0x1B	ESCAPE
0x1C	FILE SEPARATOR
0x1D	GROUP SEPARATOR
0x1E	RECORD SEPARATOR
0x1F	UNIT SEPARATOR
0x20	SPACE
0x21	EXCLAMATION MARK
0x22	QUOTATION MARK
0x23	NUMBER SIGN
0x24	DOLLAR SIGN
0x25	PERCENT SIGN
0x26	AMPERSAND
0x27	APOSTROPHE
0x28	LEFT PARENTHESIS
0x29	RIGHT PARENTHESIS
0x2A	ASTERISK
0x2B	PLUS SIGN
0x2C	COMMA

Unicode	Zeichen
0x2D	HYPHEN-MINUS
0x2E	FULL STOP
0x2F	SOLIDUS
0x30	DIGIT ZERO
0x31	DIGIT ONE
0x32	DIGIT TWO
0x33	DIGIT THREE
0x34	DIGIT FOUR
0x35	DIGIT FIVE
0x36	DIGIT SIX
0x37	DIGIT SEVEN
0x38	DIGIT EIGHT
0x39	DIGIT NINE
0x3A	COLON
0x3B	SEMICOLON
0x3C	LESS-THAN SIGN
0x3D	EQUALS SIGN
0x3E	GREATER-THAN SIGN
0x3F	QUESTION MARK
0x40	COMMERCIAL AT
0x41	LATIN CAPITAL LETTER A
0x42	LATIN CAPITAL LETTER B
0x43	LATIN CAPITAL LETTER C
0x44	LATIN CAPITAL LETTER D
0x45	LATIN CAPITAL LETTER E
0x46	LATIN CAPITAL LETTER F
0x47	LATIN CAPITAL LETTER G
0x48	LATIN CAPITAL LETTER H
0x49	LATIN CAPITAL LETTER I
0x4A	LATIN CAPITAL LETTER J
0x4B	LATIN CAPITAL LETTER K
0x4C	LATIN CAPITAL LETTER L
0x4D	LATIN CAPITAL LETTER M
0x4E	LATIN CAPITAL LETTER N
0x4F	LATIN CAPITAL LETTER O
0x50	LATIN CAPITAL LETTER P
0x51	LATIN CAPITAL LETTER Q
0x52	LATIN CAPITAL LETTER R
0x53	LATIN CAPITAL LETTER S

Unicode	Zeichen
0x54	LATIN CAPITAL LETTER T
0x55	LATIN CAPITAL LETTER U
0x56	LATIN CAPITAL LETTER V
0x57	LATIN CAPITAL LETTER W
0x58	LATIN CAPITAL LETTER X
0x59	LATIN CAPITAL LETTER Y
0x5A	LATIN CAPITAL LETTER Z
0x5B	LEFT SQUARE BRACKET
0x5C	REVERSE SOLIDUS
0x5D	RIGHT SQUARE BRACKET
0x5E	CIRCUMFLEX ACCENT
0x5F	LOW LINE
0x60	GRAVE ACCENT
0x61	LATIN SMALL LETTER A
0x62	LATIN SMALL LETTER B
0x63	LATIN SMALL LETTER C
0x64	LATIN SMALL LETTER D
0x65	LATIN SMALL LETTER E
0x66	LATIN SMALL LETTER F
0x67	LATIN SMALL LETTER G
0x68	LATIN SMALL LETTER H
0x69	LATIN SMALL LETTER I
0x6A	LATIN SMALL LETTER J
0x6B	LATIN SMALL LETTER K
0x6C	LATIN SMALL LETTER L
0x6D	LATIN SMALL LETTER M
0x6E	LATIN SMALL LETTER N
0x6F	LATIN SMALL LETTER O
0x70	LATIN SMALL LETTER P
0x71	LATIN SMALL LETTER Q
0x72	LATIN SMALL LETTER R
0x73	LATIN SMALL LETTER S
0x74	LATIN SMALL LETTER T
0x75	LATIN SMALL LETTER U
0x76	LATIN SMALL LETTER V
0x77	LATIN SMALL LETTER W
0x78	LATIN SMALL LETTER X
0x79	LATIN SMALL LETTER Y
0x7A	LATIN SMALL LETTER Z

Unicode	Zeichen
0x7B	LEFT CURLY BRACKET
0x7C	VERTICAL LINE
0x7D	RIGHT CURLY BRACKET
0x7E	TILDE
0x7F	DELETE
0x80	
0x81	
0x82	
0x83	
0x84	
0x85	
0x86	
0x87	
0x88	
0x89	
0x8A	
0x8B	
0x8C	
0x8D	
0x8E	
0x8F	
0x90	
0x91	
0x92	
0x93	
0x94	
0x95	
0x96	
0x97	
0x98	
0x99	
0x9A	
0x9B	
0x9C	
0x9D	
0x9E	
0x9F	
0xA0	NO-BREAK SPACE
0xA1	INVERTED EXCLAMATION MARK

Unicode	Zeichen
0xA2	CENT SIGN
0xA3	POUND SIGN
0xA4	CURRENCY SIGN
0xA5	YEN SIGN
0xA6	BROKEN BAR
0xA7	SECTION SIGN
0xA8	DIAERESIS
0xA9	COPYRIGHT SIGN
0xAA	FEMININE ORDINAL INDICATOR
0xAB	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
0xAC	NOT SIGN
0xAD	SOFT HYPHEN
0xAE	REGISTERED SIGN
0xAF	MACRON
0xB0	DEGREE SIGN
0xB1	PLUS-MINUS SIGN
0xB2	SUPERSCRIPT TWO
0xB3	SUPERSCRIPT THREE
0xB4	ACUTE ACCENT
0xB5	MICRO SIGN
0xB6	PILCROW SIGN
0xB7	MIDDLE DOT
0xB8	CEDILLA
0xB9	SUPERSCRIPT ONE
0xBA	MASCULINE ORDINAL INDICATOR
0xBB	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
0xBC	VULGAR FRACTION ONE QUARTER
0xBD	VULGAR FRACTION ONE HALF
0xBE	VULGAR FRACTION THREE QUARTERS
0xBF	INVERTED QUESTION MARK
0xC0	LATIN CAPITAL LETTER A WITH GRAVE
0xC1	LATIN CAPITAL LETTER A WITH ACUTE
0xC2	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
0xC3	LATIN CAPITAL LETTER A WITH TILDE
0xC4	LATIN CAPITAL LETTER A WITH DIAERESIS
0xC5	LATIN CAPITAL LETTER A WITH RING ABOVE
0xC6	LATIN CAPITAL LETTER AE
0xC7	LATIN CAPITAL LETTER C WITH CEDILLA
0xC8	LATIN CAPITAL LETTER E WITH GRAVE

Unicode	Zeichen
0xC9	LATIN CAPITAL LETTER E WITH ACUTE
0xCA	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
0xCB	LATIN CAPITAL LETTER E WITH DIAERESIS
0xCC	LATIN CAPITAL LETTER I WITH GRAVE
0xCD	LATIN CAPITAL LETTER I WITH ACUTE
0xCE	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
0xCF	LATIN CAPITAL LETTER I WITH DIAERESIS
0xD0	LATIN CAPITAL LETTER ETH (Icelandic)
0xD1	LATIN CAPITAL LETTER N WITH TILDE
0xD2	LATIN CAPITAL LETTER O WITH GRAVE
0xD3	LATIN CAPITAL LETTER O WITH ACUTE
0xD4	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
0xD5	LATIN CAPITAL LETTER O WITH TILDE
0xD6	LATIN CAPITAL LETTER O WITH DIAERESIS
0xD7	MULTIPLICATION SIGN
0xD8	LATIN CAPITAL LETTER O WITH STROKE
0xD9	LATIN CAPITAL LETTER U WITH GRAVE
0xDA	LATIN CAPITAL LETTER U WITH ACUTE
0xDB	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
0xDC	LATIN CAPITAL LETTER U WITH DIAERESIS
0xDD	LATIN CAPITAL LETTER Y WITH ACUTE
0xDE	LATIN CAPITAL LETTER THORN (Icelandic)
0xDF	LATIN SMALL LETTER SHARP S (German)
0xE0	LATIN SMALL LETTER A WITH GRAVE
0xE1	LATIN SMALL LETTER A WITH ACUTE
0xE2	LATIN SMALL LETTER A WITH CIRCUMFLEX
0xE3	LATIN SMALL LETTER A WITH TILDE
0xE4	LATIN SMALL LETTER A WITH DIAERESIS
0xE5	LATIN SMALL LETTER A WITH RING ABOVE
0xE6	LATIN SMALL LETTER AE
0xE7	LATIN SMALL LETTER C WITH CEDILLA
0xE8	LATIN SMALL LETTER E WITH GRAVE
0xE9	LATIN SMALL LETTER E WITH ACUTE
0xEA	LATIN SMALL LETTER E WITH CIRCUMFLEX
0xEB	LATIN SMALL LETTER E WITH DIAERESIS
0xEC	LATIN SMALL LETTER I WITH GRAVE
0xED	LATIN SMALL LETTER I WITH ACUTE
0xEE	LATIN SMALL LETTER I WITH CIRCUMFLEX
0xEF	LATIN SMALL LETTER I WITH DIAERESIS

Unicode	Zeichen
oxFo	LATIN SMALL LETTER ETH (Icelandic)
oxF1	LATIN SMALL LETTER N WITH TILDE
oxF2	LATIN SMALL LETTER O WITH GRAVE
oxF3	LATIN SMALL LETTER O WITH ACUTE
oxF4	LATIN SMALL LETTER O WITH CIRCUMFLEX
oxF5	LATIN SMALL LETTER O WITH TILDE
oxF6	LATIN SMALL LETTER O WITH DIAERESIS
oxF7	DIVISION SIGN
oxF8	LATIN SMALL LETTER O WITH STROKE
oxF9	LATIN SMALL LETTER U WITH GRAVE
oxFA	LATIN SMALL LETTER U WITH ACUTE
oxFB	LATIN SMALL LETTER U WITH CIRCUMFLEX
oxFC	LATIN SMALL LETTER U WITH DIAERESIS
oxFD	LATIN SMALL LETTER Y WITH ACUTE
oxFE	LATIN SMALL LETTER THORN (Icelandic)
oxFF	LATIN SMALL LETTER Y WITH DIAERESIS

A.3 CD-ROM/Website

Auf der CD-ROM finden Sie zu jedem im Buch erwähnten Beispiel das passende File. Zudem enthält die CD-ROM Tryout-Versionen ausgewählter Tools, die Flash hervorragend ergänzen.

Sie können uns im Internet auf der Seite ***www.DieFlasher.de*** besuchen. Auf dieser Website gibt es einen gesonderten Bereich speziell für die Käufer dieses Buches. Mit den untenstehenden Angaben haben Sie Zugang zu diesem Bereich. Hier können Sie Aktualisierungen, Verbesserungen und zusätzliche Informationen abrufen. Für den Fall, dass noch die eine oder andere Frage offen geblieben ist, haben wir dort eigens auch ein Forum eingerichtet.

Name: Gast
Passwort: 167645832134

(Bitte beachten Sie Groß und Kleinschreibung, Eingabe ohne Leer- und Trennzeichen)

Aktuelle Informationen zu weiteren Publikationen finden Sie unter *http://www.addison-wesley.de*.

Index

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.