Digitale Ausgabe

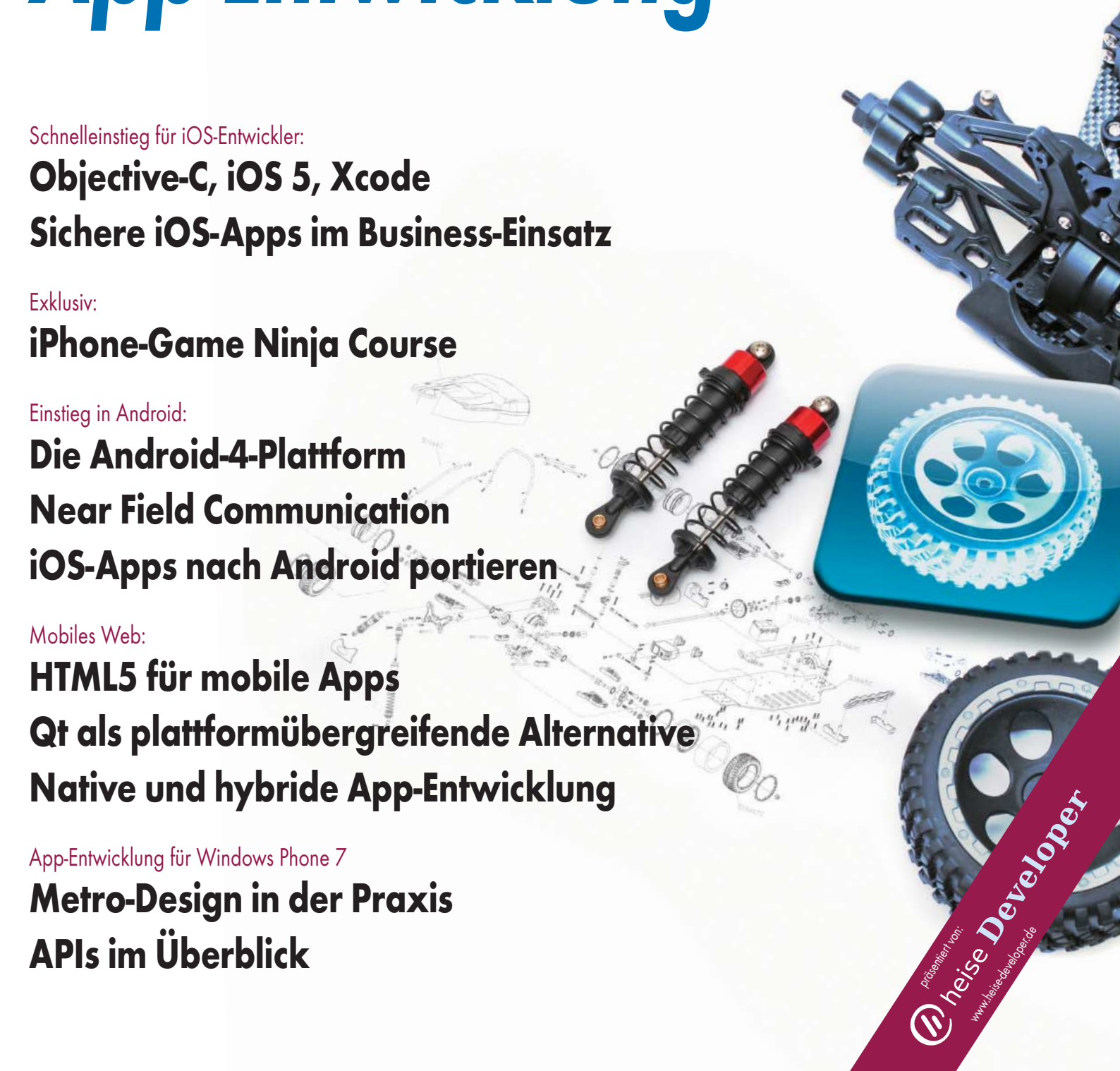
iX DEVELOPER

€ 8,99

3/2012

iOS, Android, Windows Phone und das mobile Web:

# App-Entwicklung



Schnelleinstieg für iOS-Entwickler:

**Objective-C, iOS 5, Xcode**

**Sichere iOS-Apps im Business-Einsatz**

Exklusiv:

**iPhone-Game Ninja Course**

Einstieg in Android:

**Die Android-4-Plattform**

**Near Field Communication**

**iOS-Apps nach Android portieren**

Mobiles Web:

**HTML5 für mobile Apps**

**Qt als plattformübergreifende Alternative**

**Native und hybride App-Entwicklung**

App-Entwicklung für Windows Phone 7

**Metro-Design in der Praxis**

**APIs im Überblick**

präsentiert von: heise Developer www.heise-developer.de

# Pferdewetten, Leithammel und Platzhirsche

Es klingt nach einer überschaubaren Ausgangslage: Nach viel Gerangel setzten sich neben Apple schließlich Microsoft (mit viel Geld und dem Zukauf von Nokia) sowie Google (mit noch mehr Geld und Kooperationen mit Samsung und Motorola) durch. Schnell hat sich gezeigt, dass der Teufel auch hier im Detail steckt. So verfolgen alle drei Systeme ganz eigene Designkonzepte, setzen eigene Programmiersprachen und Tools voraus und adressieren eine Vielzahl unterschiedlicher Hardwareumgebungen. Wer Pech hat, muss seine App jeweils noch auf verschiedene Betriebssystemversionen anpassen oder in Kauf nehmen, einen Teil seiner Kunden schlicht nicht bedienen zu können. Wirft man einen Blick auf die Zahlen, können iOS-Entwickler davon ausgehen, dass rund 70 Prozent ihrer Kunden bereits das aktuelle iOS 5.1 auf ihren Geräten installiert haben. Android-Entwickler haben es da schwerer – die derzeit aktuelle Version 4 (Ice Cream Sandwich) läuft auf gerade mal 5 Prozent aller Geräte, knapp 64 Prozent entfallen auf Android 2.3 (Gingerbread) und noch 21 Prozent laufen unter Android 2.2 (Froyo). Für Entwickler bedeutet das, dass sie genau abwägen müssen, auf welches System sie ihre Anwendungen anpassen wollen.

Web-Apps versprechen eine leistungsstarke Alternative zu nativen Anwendungen, die zudem gleich das Problem mit der Plattformfragmentierung erschlagen. Das Stichwort sind Standards wie HTML5 (nicht ganz sauber gerne als Oberbegriff für einen Technik-Mix aus HTML, CSS und JavaScript verwendet), die einen Zugriff auf Hardwarefunktionen erlauben und es Entwicklern ermöglichen, Look and Feel der gewohnten nativen Systemumgebung nachzuempfinden. Verschiedene Frameworks helfen, die Vielfalt zu beherrschen.

Für welche Plattform man sich entscheidet, liegt letztlich ganz in der Hand des Entwicklers. Apple bietet jüngsten Erhebungen nach das Ökosystem, in dem Kunden am ehesten bereit sind, für Apps zu bezahlen. Das mag sich in naher Zukunft ändern, wenn Google Play mit einer strengeren Freigabepolitik höherwertige Apps verspricht. Unter Android genießen Entwickler derzeit etwas mehr Freiheiten und können schon jetzt auf Techniken wie NFC zurückgreifen, der man im Zuge des „Mobile Payment“ in den kommenden Jahren einen Durchbruch vorhersagt. An Microsofts Windows Phone schließlich scheiden sich die Geister – ein herausragendes, aber gewöhnungsbedürftiges Designkonzept, ein noch junger, aber dafür von weniger Mitbewerbern bevölkerter Marketplace, eine wenig verbreitete, aber im Kontext von Windows 8 sicherlich aufkeimende Softwareplattform. Man wird abwarten müssen, wohin sich Microsofts System entwickelt. Da das Unternehmen die Plattform derzeit mit viel Nachdruck fördert, mag ein früher Einstieg lohnen. Ganz pragmatisch gesehen fällt die Wahl leichter: Mac-Entwickler werden sich traditionell bei iOS am schnellsten eingewöhnen, Java-Developer sich in die Java-nahe Android-Plattform einleben und .NET- und Silverlight-Entwickler sich zügig unter Windows Phone zurechtfinden. Wer mit Webanwendungen vertraut ist, kann prinzipiell jede Plattform bedienen.

Was die Zukunft bringt, ist ungewiss. Android hat mit der Version 4 seine Smartphone- und Tablet-Plattformen vereinheitlicht und strengere Regeln, unter anderem in puncto Design, vorgegeben. Und versucht, mit Google Play am Erfolg von Apples App Store anknüpfen zu können. Microsoft steht in den Startlöchern und vielleicht wird künftig Mozilla dazustoßen, mit einem komplett browserbasierten Betriebssystem, das Web-Apps auf eine Stufe mit nativen Anwendungen stellt. Wie immer kann dabei ein Blick über den Tellerrand nicht schaden: Prinzipiell verfolgt jedes System die gleichen Ziele, wenn auch auf ganz unterschiedlichen Wegen – wer hier offen auf Probleme und Ansätze ihm fremder Plattformen eingeht, wird letztlich auch mit seiner eigenen App davon profitieren.

ROBERT LIPPERT

## Gelddruckmaschine iOS

Apples iOS ist eine der Standardplattformen für App-Entwickler. Klar definierte Zielplattformen, ein stetig wachsendes Ökosystem und auf Seiten der Kunden die Bereitschaft, für Apps zu bezahlen, machen das System für Entwickler attraktiv. Zuletzt vorgestellte Erweiterungen wie das Automatic Reference Counting erleichtern die Softwareentwicklung und mit Tools wie der Xcode-IDE können App-Entwickler das iPad ebenso ansprechen wie iPhone und iPod touch.

ab Seite 7

## Durchstarter Android

Googles Android ist der Durchstarter unter den neuen Mobile-Betriebssystemen und macht in Sachen Verbreitung inzwischen selbst Apple spürbar zu schaffen. Das schnelle Wachstum hat jedoch seinen Preis, vorneweg die oft zitierte Gerätefragmentierung. Google macht mit Android 4 striktere Vorgaben an Hard- und Software und bietet Tools, die Entwicklern eine Anpassung an die diversen Plattformen erleichtern. Zudem erlaubt Android Near Field Communication – eine Schlüsseltechnologie für mobile Bezahlsysteme.

ab Seite 67

## App-Entwicklung mit iOS

**Games**
Spieleentwicklung für das iPhone 8

**Programmiersprachen**
Einführung in Objective-C 18

**Security**
Sichere Datenübertragung mit Location-Based Services, Gesten und Verschlüsselung 29

**App-Security**
Sichere iOS-Entwicklung im Unternehmensumfeld 34

**Plattformübersicht**
iOS 5 und Xcode 4.3.1 für Entwickler 43

**Multimedia**
HTTP Live Streaming: Grundlagen und Einsatzmöglichkeiten 48

**Speicherverwaltung**
Memory Management in iOS 52

**Literatur**
Literaturübersicht zu iOS-Programmierbüchern 56

**Backend**
Serverseitige Infrastruktur für komplexe iOS-Apps 60

## App-Entwicklung mit Android

**Softwaremigration**
iOS-Apps auf Android portieren 68

**Performance-Tuning**
Performancekritische Applikationen in Android 70

**Drahtlose Datenübertragung**
Near Field Communication unter Android 4 76

**Frameworks**
Schnelleinstieg in Qt: Warum Entwickler mit Nokias Framework nie danebengreifen 82

**App-Security**
Sicherheitsmechanismen für Android-Apps 88

## Außenseiter Windows Phone

Microsoft setzt mit Windows Phone auf einen komplett neuen Ansatz auf Grundlage der Metro-Designprinzipien. Strikte Vorgaben an Hardware und Software erinnern dabei ein wenig an Apple. Gute Aussichten für die Zukunft bietet die Nähe zum kommenden Windows 8. Wer nicht so lange warten möchte, findet schon jetzt interessante Voraussetzungen für die App-Entwicklung.

ab Seite 101

## Zukunftssichere Cross-Plattform-Entwicklung

Web-Apps sind der neue Trend in Sachen plattformübergreifender, mobiler Anwendungsentwicklung. Das Spektrum reicht von reinen Webanwendungen, über hybride Mischformen aus Web- und nativer App bis hin zu per Framework erzeugten „echten" nativen Apps. Der Durchbruch geht auf die Entwicklung des kommenden Web-Standards HTML5 zurück.

ab Seite 125

**Plattformübersicht**
Neuerungen in Android 4 — 94

**User Interface**
Schöne Benutzeroberflächen unter Android — 97

**Literatur**
Bücher zu Android, Windows Phone und Web-Apps — 100

### Entwickeln für Windows Phone 7

**Mobile-API**
Die Windows-Phone-Standard-APIs im Überblick — 102

**IDE**
Einstieg in Expression Blend und Visual Studio — 109

**Datenpersistenz**
Isolated Storage unter Windows Phone 7 — 114

**Design**
Metro UI Design in der Praxis: Neue Konzepte für die Interaktion zwischen App und Anwender — 120

### Web-Apps – HTML5 und Co.

**Softwareentwicklung**
Best Practices für die Cross-Plattform-App-Entwicklung — 126

**HTML5**
Systemunabhängige Web-Apps mit HTML5, CSS 3 und JavaScript — 132

**Cross-Plattform-Entwicklung**
Plattformunabhängige Web-Apps mit „The-M-Project" — 137

**App-Plattform**
Ein kleiner Überblick zu Appcelerator Titanium — 142

**Cross-Platform mit .NET**
Mono-Apps für iOS, Android und Windows Phone — 147

### Sonstiges

Editorial — 3
Inserentenverzeichnis — 154
Impressum — 154


# Auf der Heft-DVD

**video2Brain-Video-Tutorials:** Rund 3 Stunden Videomaterial für Android- und iOS-Entwickler, mit Frank Jüstel und Constantin Ehrenstein. Wer Apps für Apples mobiles Betriebssystem entwickeln möchte, lernt hier die Grundlagen zu Objective-C kennen (inklusive Speichermanagement und den Umgang mit der Xcode-IDE) sowie die Neuerungen aus iOS 5, darunter auch die Anbindung an die iCloud. Wer lieber mit Android arbeitet, bekommt einen Einstieg in die Java-Programmierung für Googles Plattform vermittelt.

## SDKs und IDEs

**Adobe Flex:** Open-Source-SDK für die Entwicklung mobiler Applikationen auf der Grundlage von Adobes Flash/AIR-Laufzeitumgebung. Das Software Development Kit enthält das Flex-Framework und den Flex-Compiler, mit denen sich Flex-Anwendungen unter einer IDE der Wahl entwickeln lassen.
**Android SDK:** Eclipse-Plug-in zum Programmieren für das Google-Betriebssystem. Das Starter-Paket enthält die nötigen Tools, um die Eclipse-IDE für die Entwicklung von Android-Apps vorzubereiten.
**Corona SDK:** Cross-Plattform-SDK für Mac OS X und Windows. Das speziell für die Spieleentwicklung konzipierte SDK erlaubt die Programmierung plattformübergreifender Apps und bietet einen eigenen Simulator.
**Eclipse Classic:** Die Basisumgebung für die Android-Entwicklung. Die Classic-Variante bietet das Grundgerüst der bekannten IDE und lässt sich über zahlreiche Extensions beliebig erweitern.
**RhoStudio:** IDE auf Eclipse-Basis für Mac OS X und Windows. Die Software erlaubt die Entwicklung nativer Business-Anwendungen auf Grundlage des quelloffenen Ruby-Frameworks Rhodes.
**Sencha Touch 2:** Mobile Development mit HTML5. Das Web-Framework stellt Entwicklern über 50 Komponenten, ein State-Management und ein integriertes MVC-System zur Verfügung und verspricht flüssiges Scrolling und adaptive Layouts für mobile Anwendungen.
**Windows Phone SDK 7.1:** SDK zum Erstellen von Anwendungen und Spielen für Windows-Phone-7.0- und Windows-Phone-7.5-Geräte. Umfasst unter anderem Visual Studio 2010 Express, den Windows-Phone-Emulator, das Silverlight-4-SDK und das Microsoft Expression Blend SDK für Windows Phone.

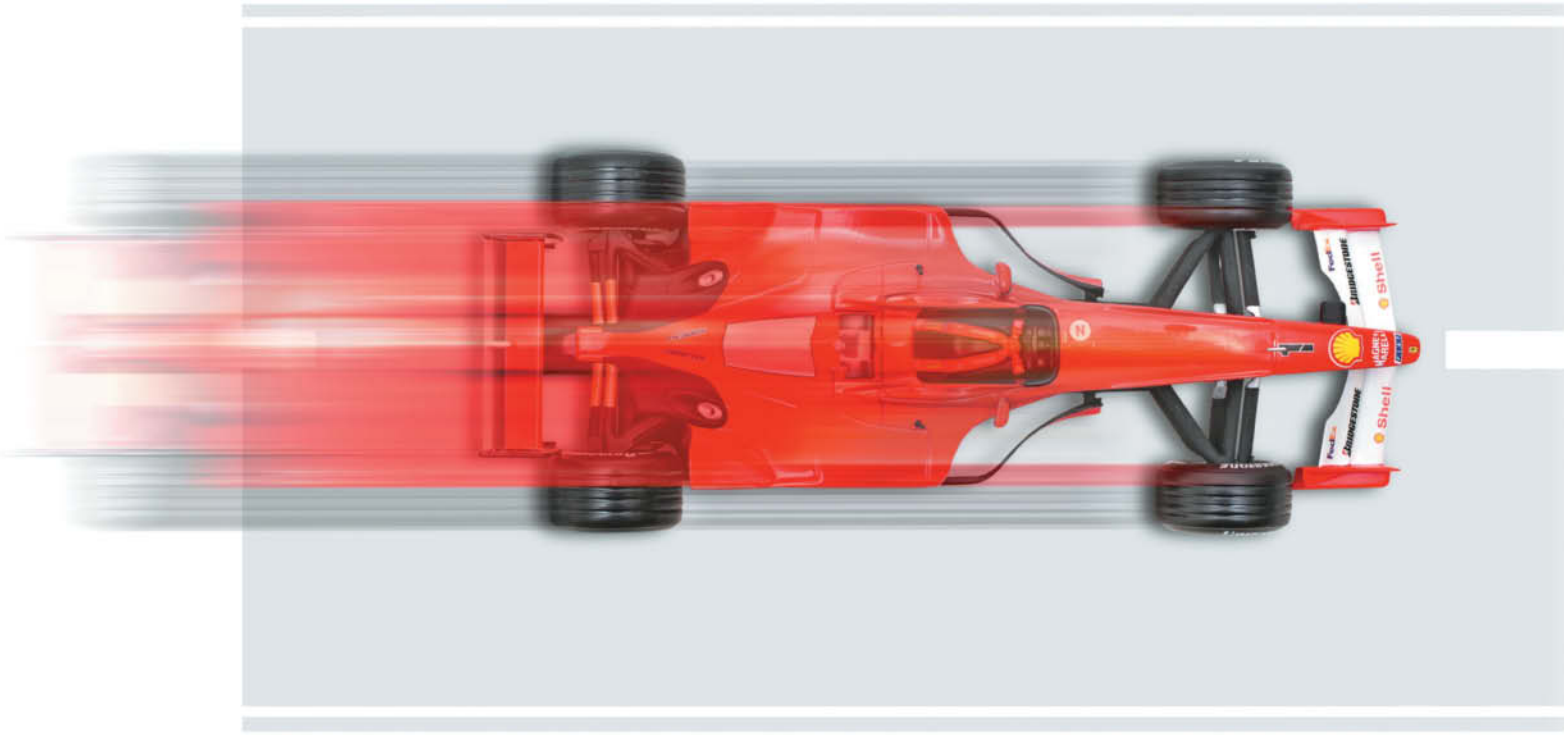
## Hinweis für Käufer

- PDF- und iPad-Version: In der *iX*-App finden Sie einen Button zum Download des DVD-Images.
- Adobe-Digital-Editions-Version: Das Image finden Sie unter dem Link *ftp://ftp.heise.de/pub/ix/developer/0212/appentwicklung.iso*

## Tools & Frameworks

**Cappuccino:** Open-Source-Framework für Mac OS X. Erlaubt die Entwicklung plattformübergreifender Webanwendungen mit Objective-J.
**MoSync 3.0:** Mobile SDK für native und hybride Cross-Plattform-Apps, seit Version 3.0 HTML5-fähig. Die Entwicklungsumgebung ermöglicht Apps auf Grundlage von C/C++ und JavaScript/HTML5.
**MonoTouch, Mono for Android:** Cross-Plattform-Entwicklung mit Objective-C respektive C#/.NET. Entwickler können bestehende .NET-Bibliotheken in ihre mobilen Projekte importieren, Visual Studio (Android) und Xcode nutzen oder auch die Mehrkern-Prozessoren des iPad 2 oder iPhone 4S ausreizen.
**PhoneGap:** Ermöglicht eine Cross-Plattform-Entwicklung mit HTML5. App-Entwickler können über PhoneGap aus Webanwendungen heraus die nativen APIs mobiler Geräte ansprechen.
**SproutCore:** Das quelloffene Framework bietet einen MVC-Ansatz und erlaubt performante clientseitige Ausführung von Code. Besonders im mobilen Kontext erlaubt das stabile Apps, selbst bei Verbindungsproblemen.
**The-M-Project:** Cross-Plattform-Entwicklung mit CSS3 und HTML5 nach dem MVC-Prinzip. Das Framework umfasst eine Build-Umgebung („Espresso“), die über die Kommandozeile eine native Paketierung von Web-Apps ermöglicht.
**Hammer.js:** JavaScript-Library für Multitouch-Gesten wie Tap, Double Tap, Hold, Drag und Pinch.
**jQuery/jQuery Mobile:** JavaScript-Klassenbibliothek zur DOM-Manipulation und -Navigation.
**Notepad++ 6:** Ein Quellcode-Editor für verschiedene Sprachen.

**Listings und Lizenzen** zu den Heftartikeln und den Softwarepaketen.



## iOS: Konstante im Mobil-Universum

Apples iOS mag nicht mehr das am weitesten verbreitete Smartphone-Betriebssystem sein – es ist aber, wie jüngste Studien zeigen, das System, mit dem sich als Entwickler das meiste Geld verdienen lässt. Die Gründe sind vielfältig – eine in weiten Teilen einheitliche Plattform, strenge Designvorgaben und eine Updatepolitik, die auch ältere Gerätegenerationen berücksichtigt, schaffen eine verlässliche Grundlage für die Entwicklung. Wer heutzutage eine (kommerzielle) App veröffentlichen will, kommt um Apples Plattform daher nicht herum.


| | |
|---|---|
| Spieleentwicklung: So einfach gelingt ein Casual Game | 8 |
| Grundlagen iOS-Entwicklung: Objective-C | 18 |
| Location-Based Services, Gesten und Verschlüsselung | 29 |
| Sichere iOS-Entwicklung im Unternehmensumfeld | 34 |
| iOS 5 und Xcode für Entwickler | 43 |
| Multimedia: HTTP Live Streaming | 48 |
| Memory Management unter iOS | 52 |
| Literaturübersicht für iOS-Entwickler | 56 |
| Serverseitige Infrastruktur für komplexe iOS-Apps | 60 |

Spieleentwicklung für das iPhone

# Ninja Course

**Patrick Völcker**


Wer hat nicht schon davon geträumt, ein eigenes Spiel zu entwickeln? Es muss ja nicht gleich ein 3D-Ego-Shooter sein, ein nettes Jump and Run auf dem eigenen iPhone tut es für den Beginn auch. Wer dessen grundsätzlichen Aufbau, inklusive Game Engine, Animation, Kollisionsabfrage und Scrolling verstanden hat, kann bald seine eigenen Ideen umsetzen. Der Artikel begleitet die komplette Entwicklung bis hin zum fertigen Spiel – Mitmachen erwünscht.

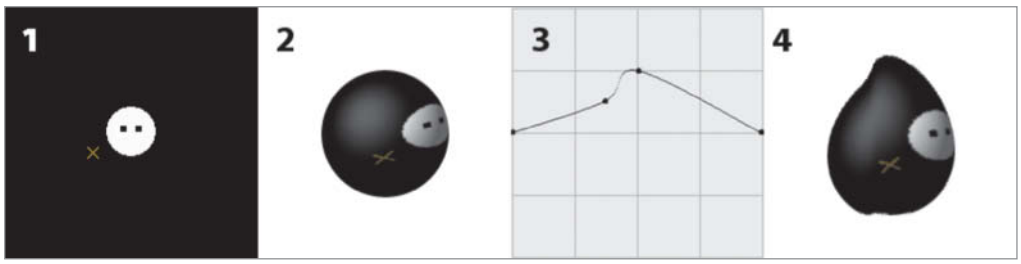
In dem Jump and Run „Ninja Course“ (Video des Gameplay unter [a]) wird ein kleiner Ninja auf seinem Trainingsparcours gesteuert. Während die Spielfigur immer schneller werdend über die Plattformen hetzt, muss der Spieler versuchen, durch geschicktes Springen (Berühren des Touchscreens) so lange wie möglich am Leben zu bleiben und gleichzeitig Sterne einzusammeln, die ihn mit der notwendigen Energie versorgen. Das Spiel ist beendet, wenn der Spieler keine Energie mehr hat, von einer Plattform ins Buschwerk gefallen ist oder ins Seitenaus gedrängt wurde.

Für das Projekt benötigen Entwickler zwei Tools: Zum einen die Open-Source-Bildbearbeitungssoftware GIMP [b] (oder eine Alternative nach Wahl) und natürlich die Entwicklungsumgebung Xcode. Da Programmierer gewöhnlich nur selten zugleich begnadete Grafiker sind, soll zunächst gezeigt werden, wie man auf einfache Weise ansprechende Game-Grafiken erstellt. Im zweiten Teil stellt der Artikel den dazugehörigen Code vor.

Nach dem Download von GIMP (deutsch) und der anschließenden Installation erstellt man sich seinen Ninja über eine 200 × 200 Pixel große schwarze Fläche, in deren Mitte man einen 40 × 40 Pixel großen weißen Kreis setzt und diesen mit zwei schwarzen Pünktchen (den Augen) ausstattet. Links daneben sei das Gesicht mit einem verwegenen Kreuz (Werkzeug: *Stift*, Pinsel: *Diagonal Star (11)*) verziert. Danach bildet man die so erstellte Textur auf einer Kugel ab (*Filter->Abbilden->Auf Objekt abbilden*):

- *Einstellungen*: Abbilden auf Kugel
- *Licht*: Punktlichtquelle x:-0.5 y:-0.5 z:1.5
- *Material*:
  - *Intensitätsstufen:*
    - Umgebung: 0.3, Diffuse: 1.0
  - *Reflexivität:*
    - Diffuse: 0.5, Spiegel: 0.3, Glanzlicht: 2.0
- *Ausrichtung*:
  - Position: x:0.5 y:0.5 z:0.0
  - Rotation: x:20 y:50 z:-10

Und schon ist der kugelrunde Ninja erstellt. Über *Filter ->Verzerren->Verbiegen* verleiht man ihm noch eine organischere Form, in dem man den Mittelpunkt der oberen Kurve ein Kästchen nach oben schiebt und einen neuen Kurvenpunkt einen halben Kasten links darunter setzt. Abbildung 1 illustriert die wesentlichen Arbeitsschritte. Anschließend beschränkt man die Grafik noch auf ihr notwendiges Maß




**Die Entwicklung des Ninja-Sprites: Die erste Grafik wird in GIMP auf einer Kugel abgebildet und anschließend über die obere Verbiegungskurve neu geformt (Abb. 1).**

(*Bild->Automatisch zuschneiden*) und verkleinert sie über *Bild->Bild skalieren* auf 39 × 50 Pixel (Interpolation: Sinc Lanczos3). Fertig ist die Ninja-Grafik *player.png*.

Für die Laufanimation benötigt man noch zwei Füße: Über *Bild->Leinwandgröße…* beschränkt man das Bild auf die oberen 5 Pixel, passt die Ebene an (*Ebene->Ebene auf Bildgröße*), schneidet das Bild noch einmal automatisch zu und speichert den Ausschnitt als *ninjafoot.png*. Im Spiel werden beide Füße über einen einfachen Algorithmus animiert, sodass man sich die Animation aus Einzelbildern in GIMP an dieser Stelle erspart.

Ähnlich wie die Spielfigur ist die benötigte Plattform zu erstellen, die man später im Code dynamisch skaliert, um mehrere Größen zu erhalten und das Spiel so abwechslungsreicher zu gestalten. Dafür fülle man ein 200 × 200 Pixel großes Bild mit dem Muster *pine* (Werkzeugkasten: *Füllen ->Füllart: Muster*) und bildet es wie folgt auf ein Objekt ab:

- *Einstellungen*: Abbilden auf Zylinder
- *Ausrichtung*:
  – Position: x:0.5 y:1.2 z:0.0
- *Rotation:* x:0 y:0 z:0
  – Zylinder: Radius: 0.5
  – Länge: 1.2

Auch hier schneidet man das Bild anschließend automatisch zu, verlängert es über *Ebene->LiquidRescale* (in der Mac-Version vorinstalliert, unter Windows über [c] nachrüstbar) auf 700 Pixel Höhe (Breite bleibt erhalten) und speichert die Grafik unter dem Namen *platform.png*.

Der Hintergrund ist schnell erledigt: Ein 480 × 320 Pixel großes Bild wird einfach komplett von oben nach unten gefüllt (Werkzeugkasten: *Farbverlauf*, Farbverlauf *Land 1*, Form *linear*) und unter *bg.png* abgespeichert.

Etwas aufwendiger ist die Dekoration: Das Buschwerk erzeugt man, in dem ein neues Bild (480 × 160 Pixel) mit transparentem Hintergrund bis auf einen kleinen, ausgefransten Rand oben komplett mit dem Werkzeug Pinsel (Pinsel *Vine*) bemalt wird. Über *Filter->Abbilden->Kacheln* verdoppelt man die Breite auf 960 Pixel und übermalt die unschöne Nahtstelle genau in der Mitte erneut mit *Vine*, bis sie einem gefällt. Anschließend reduziert man die Leinwandgröße über *Bild->Leinwandgröße…* zurück auf 480 × 160 Pixel (inklusive Zentrieren!) und verdoppelt die Breite über *Kacheln* erneut. Die nötigen Arbeitsschritte illustriert die Abbildung 2. Als Ergebnis erhält man eine Kachelgrafik, die in x-Richtung aus zwei nahtlos aneinander gefügten, identischen Kacheln besteht. Bleibt noch, die Datei abschließend unter *bush.png* zu speichern.

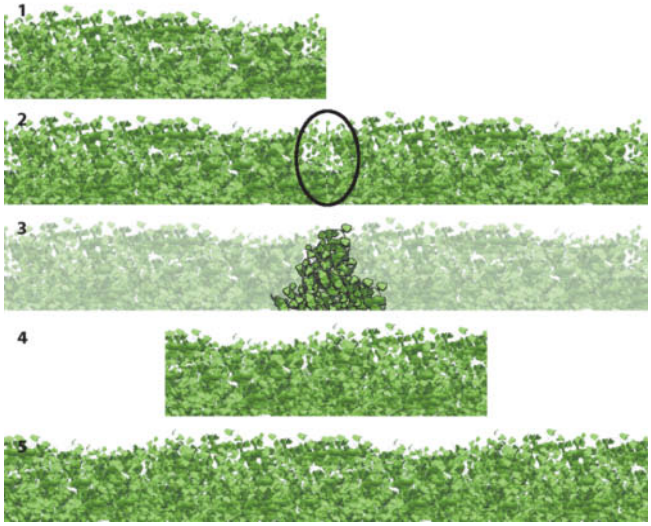
Zu guter Letzt benötigt der Ninja noch Items, die er einsammeln kann: In diesem Fall erstellt man sich einen Stern, der später den Energievorrat auffüllen soll. Dazu öffnet man in einem 100 × 100 Pixel großen und transparentem Bild den Menüpunkt *Filter->Licht und Schatten->Verlaufsaufhellung*. Hier selektiert man im Reiter *Auswahl* den *Bright_Star*, setzt diesen im Reiter *Einstellungen* auf die Bildmitte und wählt den Radius *25*. Für eine stärkere Leuchtkraft dupliziert man die Ebene zweimal (im Ebenenfenster das Kontextmenü der Ebene: *Ebene duplizieren*), schneidet das Bild wie üblich automatisch zu und speichert die Grafik unter *star.png*.

## Ran an den Code

Das sollte als grafische Vorleistung genügen, jetzt ist die eigentliche Programmierung an der Reihe. Für eine native App muss der Entwickler das Spiel in Objective-C anlegen. Wer JavaScript bevorzugt, dem sei ein Blick auf die Webseite des Autors [d] empfohlen, wo das Spiel aus diesem Artikel nach dem gleichen Schema noch einmal in HTML5 entwickelt wurde.

In Xcode erstellt man sich zunächst ein neues Projekt namens *ninjaCourse* als *Application->Single View Application*. Es ist darauf zu achten, dass man dabei das Automatic Reference Counting nutzt. Das ist nicht nur praktischer beim Programmieren, der Code wird sogar schneller ausgeführt. In einem ersten Schritt fügt man dann dem Projekt das QuartzCore-Framework und seine sechs Grafiken hinzu.

Der Autor vermeidet gerne sowohl Interface Builder als auch Storyboard Editor und erstellt seine Inhalte bevorzugt programmatisch. Um den Code etwas übersichtlicher – und hoffentlich für Nicht-Objective-C-Programmierer verständlich – zu gestalten, wurde eine rudimentäre *Sprite*-Klasse erstellt, die dem Entwickler oft benötigte Schritte abnimmt. Letztendlich macht diese Klasse nichts anderes, als den Zugriff auf ImageViews zu erleichtern, die hier als Sprites die-




**So entsteht eine Nahtloskachel: Das aufgemalte Buschwerk wird einfach gekachelt und die unschöne Nahtstelle übermalt, anschließend beschnitten und erneut gekachelt (Abb. 2).**

nen. Abgesehen davon werden alle wichtigen Berechnungen im *ViewController* erledigt.

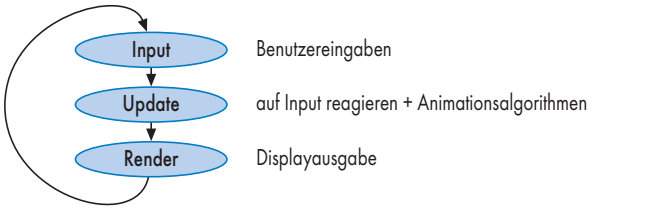
Im Event-Handler *viewDidLoad* lädt man sich nach Programmstart alle Grafiken in die App. Die meisten kann man dabei gleich als Sprite definieren, lediglich das statische Hintergrundbild *bg.png* wird direkt in der Hauptview *canvas* hineingeladen. Elemente wie die Plattformen, die mehr als einmal vorkommen, legt man übergreifend als MutableArray an, in das man die einzelnen Sprites speichert, um sie in der Game Engine über eine *for*-Schleife zu durchlaufen und zu manipulieren.

Nach dem Setzen der wichtigsten Werte (beispielsweise *score* auf null) für den Levelstart über die Methode *newGame* ist abschließend der Timer *callGameEngine* zu starten, der dafür sorgt, dass die Game Engine – der zentrale Punkt des Spiels – alle 33 Millisekunden aufgerufen wird und damit die Spielabläufe und Animationen am Laufen hält.

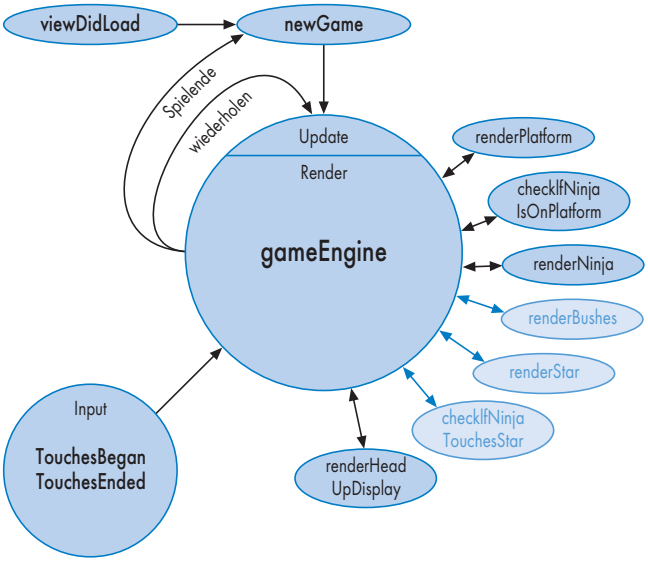
## Die Game Engine

Was nahezu alle Action-Spiele gemeinsam haben, ist eine zentrale Game Engine, oft Game-Loop genannt (Abbildung 3). Sie sorgt dafür, dass das Spiel flüssig wie ein Kinofilm läuft. Gewöhnlich werden in ihr drei Schritte durchlaufen: *Input*, *Update*, *Render*.

Da Objective-C Event-basiert ist, kann der Entwickler den Eingabe-Teil (die Berührung des Touchscreens) auslagern, muss sich also nur um die Updates (wie Animationsschritte, Kollisionserkennung et cetera) und die anschließende Ausgabe auf dem Screen kümmern. Um die Game Engine übersichtlich zu halten, ruft diese Untermethoden wie *renderPlatforms* oder *renderNinja* auf, die sich jeweils gesondert um die einzelnen Sprites und deren Darstellung kümmern. Abbildung 4 zeigt den klassischen Spielaufbau für Ninja Course.




**Der klassische Aufbau fast aller Game Engines: Benutzereingabe, Reaktion und Ausgabe (Abb. 3)**




**Der klassische Spielaufbau eines iPhone-Game. Die ausgegrauten Bereiche sind nicht optional (Abb. 4).**

## Parallax-Scrolling

Da hier ein Side-Scroller entwickelt werden soll, bewegt sich alles in die entgegengesetzte Richtung des Ninja-Sprite. Das heißt, statt die x-Koordinate des Ninjas nach rechts zu bewegen, setzt man alle anderen Sprites, sowohl im Vorder- als auch im Hintergrund, nach links. Sie befinden sich jeweils auf eigenen Ebenen, abhängig von der Reihenfolge, in der sie bei Programmstart nacheinander aufgerufen wurden (Abbildung 4). Je später ein Sprite initialisiert wird, desto weiter vorne (aus der Sicht des Spielers) befindet es sich auf dem Screen. Diese Tatsache kann man sich für einen schönen optischen Effekt zunutze machen: das Parallax-Scrolling.

Indem man hintere Ebenen beziehungsweise Sprites langsamer als vordere bewegt, erzeugt man einen räumlichen Effekt, der die grafisch dreidimensionale Gestaltung der Sprites noch weiter betont. Scrollt ein Sprite links aus dem Screen, wird es einfach rechts wieder angesetzt. Die Busch-Ebenen hingegen scrollen nur bis zur Hälfte (die exakt der Länge der Bildschirmbreite entspricht) und springen dann zurück auf Position null. Da diese Sprites aus zwei optisch identischen Hälften bestehen, wird der Spieler gar nicht bemerken, dass hier ein Positionswechsel stattfindet. Er hat vielmehr den Eindruck, als würden endlos lange Sprites verwendet. Für diesen Wechsel benötigt der Entwickler im Vergleich zu den anderen Sprites keine *if*-Abfrage, vielmehr kann er einen eleganten mathematischen Trick anwenden: Er teilt die Position des Sprites einfach durch die Hälfte von dessen Breite (beziehungsweise durch die Bildschirmbreite), über den Restwert dieser Division erhält er die neue x-Koordinate automatisch (Funktion *fmod* in Methode *renderBushs*).

## Kollisionsabfrage

Außer Scrolling passiert bislang nur wenig. Jetzt muss man den Ninja und die Plattformen in der Methode *checkIfNinjaIsOnPlattform* logisch miteinander verbinden, also daraufhin prüfen, ob sich der Ninja in der Luft befindet (keine Kollision stattfindet) oder eine der Plattformen berührt (Kollision). Passiert Letzteres, muss man prüfen, ob die Spielfigur diese von oben oder von links berührt. Von rechts und unten kann keine Berührung erfolgen, da das Sprite nicht rückwärts laufen kann und das Spiel längst verloren ist, bevor das Sprite eine Plattform von unten bespringen könnte. Findet die Berührung hingegen oben statt, ist alles in Ordnung: Der kleine Ninja wird sauber auf die Plattform gesetzt (leicht unterhalb des eigentlichen Sprite-Rahmens, um den optischen Effekt zu erreichen, als laufe das Sprite in der Mitte der Plattformfläche) und die Laufanimation in *renderNinja* wird gestartet.

Hat der Spieler die Plattform hingegen verfehlt und knallt dabei seitlich gegen die nachfolgende, ändert sich erstmals die x-Koordinate des Ninjas direkt: Er wird mit derselben Geschwindigkeit wie die Plattformen nach links aus dem Bildschirm geschoben. Schafft er es bis zum völligen Ver-

# Jetzt Windows 8-Entwickler werden!

Auf dem Mobile World Congress in Barcelona wurde Ende Februar die Windows 8 Consumer Preview offiziell vorgestellt. Diese bietet bereits jetzt einen sehr guten Ausblick auf die Funktionen und Features der kommenden Windows-Version – Steven Sinofsky, seines Zeichens President der Windows and Windows Live Division bei Microsoft, stellt diese Interessierten ausführlich im Blog „Building Windows 8" vor. Wer sich selbst vom kommenden Microsoft-Betriebssystem überzeugen möchte, kann die Windows 8 Consumer Preview kostenlos herunterladen und testen.

- **Download unter http://bit.ly/win8-ix1**
- **Windows 8-Blog: http://bit.ly/win8-ix2**

### Ab jetzt mit Apps: Windows Store

Gleichzeitig mit Veröffentlichung der Windows 8 Consumer Preview startete auch der Windows Store. Dort finden Sie schon während der Betaphase eine reichhaltige Auswahl an neuen Apps im Metro-Stil – sowohl von Drittanbietern wie auch von Microsoft. Mit der passenden Idee haben Entwickler die Möglichkeit, eigene Apps, noch vor der Veröffentlichung der finalen Version von Windows 8, im Windows Store zu platzieren und somit von Anfang an in einem neu entstehenden Markt dabei zu sein.

Unterstützung bekommt die Entwicklergemeinde hierbei von Microsoft Deutschland, zum einen durch Vermittlung des nötigen technischen Know-how und zum anderen mit dem vorzeitigen Zugang zum Store.

### Know-how aufbauen: Windows 8 Trainings

Diese kostenlosen Entwicklertrainings bieten Ihnen Einblicke in die Planung, Gestaltung und Programmierung von Apps für Windows 8. Dabei werden einzelne Bereiche der Windows 8-Plattform im Detail vorgestellt, erkundet und an Beispielen erarbeitet. Angeboten werden insgesamt drei verschiedene Arten von Trainings: Sie können zwischen Windows 8-Entwicklung mit dem Schwerpunkt .NET oder Windows 8-Entwicklung mit dem Schwerpunkt HTML5 und JavaScript wählen.

Dritte Variante sind die sog. Metro UX Trainings, die sowohl für Designer als auch für Entwickler gedacht sind. In einem Ganztagesworkshop wird dabei das Prinzip des Metro Style Designs und dessen Auswirkungen auf die Anwendungsarchitektur intensiv beleuchtet – eine wichtige Rolle spielen dabei auch die beiden Tools Visual Studio und Expression Blend.

Alle Trainingsvarianten finden in den kommenden Monaten regelmäßig im ganzen Bundesgebiet statt. Interessierte können sich unkompliziert auf MSDN Online für den passenden Termin anmelden.

- **Windows 8 Trainings: http://bit.ly/win8-ix3**

### Ab in den Store: Windows 8 App Excellence Labs

Über die Teilnahme an einem Windows 8 App Excellence Lab haben Sie die einmalige Chance, Ihre App bereits vorab im Windows 8 Beta Store, also noch vor dem endgültigen Release von Windows 8, zu veröffentlichen und so einer breiten Öffentlichkeit zugänglich zu machen.

Dort helfen Ihnen Windows 8-Experten von Microsoft, Ihrer App in Bezug auf UI/UX, Performance-Optimierung sowie technischer Problemlösungen/Debugging den letzten Schliff zu verleihen. Im Zuge der Veranstaltung werden Token ausgegeben, mit denen man sich im Windows 8 Beta Store kostenlos registrieren kann. Die Registrierung bleibt auch nach der Betaphase bestehen und ist für die ersten zwei Jahre kostenlos. Der Token ist nicht nur der Zugang zum Windows 8 Beta Store, sondern bietet auch die Möglichkeit, einen App-Namen zu registrieren. Somit können Sie zum Beispiel für eine bessere Auffindbarkeit für Ihre App bei der Store-Suche sorgen.

- **Windows 8 App Excellence Labs: http://bit.ly/win8-ix4**

### Jetzt loslegen und Apps entwickeln?

Auf jeden Fall, denn derzeit bietet sich Ihnen die einmalige Möglichkeit, sich in einem neu entstehenden Markt hervorragend zu platzieren und als einer der ersten im Windows 8 Store eine Applikation zu publizieren. Dadurch ist nicht nur eine hohe Sichtbarkeit für Ihre App, sondern auch für Ihr Unternehmen garantiert.

### Noch Fragen?

Sind sie neugierig geworden, haben Sie bereits eine erste tolle App-Idee, wollen Sie eine unserer Veranstaltungen besuchen oder können wir Ihnen ansonsten in Sachen Windows 8-Entwicklung weiterhelfen? Wir stehen Ihnen jederzeit zur Verfügung, besuchen Sie einfach MSDN Online und holen Sie sich die benötigten Informationen oder schreiben Sie uns einfach eine E-Mail! Wir freuen uns, Ihnen weiterhelfen zu dürfen.

- **Infos zur Windows 8-Entwicklung: www.msdn.de**
- **Schreiben Sie uns: win8de@microsoft.com**

Microsoft®

schwinden durch den Schwung noch auf die Plattform, lässt man ihn langsam wieder auf seine vertikale Ursprungsposition zurückwandern.

## Items prägen den Spielablauf

Die zweite wichtige Kollisionsabfrage findet in *checkIfNinjaTouchesStar* statt: Überschneiden sich die Rahmen beider Sprites (Abbildung 5), verschwindet der Stern und füllt dafür die Energie (*energy*) des Ninjas auf. In diesem Fall hat die Berührung des Sterns etwas Gutes.

**Listing 1: Sprite.h**

```
#import <UIKit/UIKit.h>

@interface Sprite : UIView {
    UIImageView *pic;
  double width;
  double height;
  double scale;
}

@property (nonatomic, strong) UIImageView *pic;

- (id) initWithImage: (NSString*) imageName
           addToView: (UIView*) parentView;

- (void) x: (double) x
         y: (double) y
     scale: (double) faktor;

- (void) flipX: (BOOL) x
         flipY: (BOOL) y;

- (CGPoint) position;
- (double) width;
- (double) height;
- (double) scale;
- (CGRect) frame;

@end
```

**Listing 2: Sprite.m**

```
#import "Sprite.h"

@implementation Sprite
@synthesize pic;

- (id) initWithImage: (NSString*) imageName
           addToView: (UIView*) parentView {
  if (self == [super init]) {
    UIImage* img = [UIImage imageNamed:imageName];

    pic = [[UIImageView alloc] initWithImage:img];
    [self addSubview:pic];
    [parentView addSubview:self];
    width = img.size.width;
    height = img.size.height;
  }
  return self;
}

- (void) x: (double) x
         y: (double) y
     scale: (double) faktor {
  scale = faktor;
   pic.frame = CGRectMake(x, y, width*faktor, height*faktor);
}

- (void) flipX: (BOOL) x
         flipY: (BOOL) y {
  pic.transform = CGAffineTransformMakeScale((x?-1:1),(y?-1:1));
}

- (CGPoint) position {return
CGPointMake(pic.frame.origin.x,pic.frame.origin.y);}
- (double) width {return width;}
- (double) height {return height;}
- (double) scale {return scale;}
- (CGRect) frame {return CGRectMake(pic.frame.origin.x,
pic.frame.origin.y, width*scale, height*scale);}

@end
```

Ohne Probleme könnte man den Stern auch in einen Gegner verwandeln: In dem Fall müsste der Entwickler die Grafik nur austauschen (beispielsweise in eine Libelle) und statt der Energiezufuhr einfach die Methode *newGame* aufrufen, um das Spiel zu beenden und das Ziel dahingehend umzuformulieren, dass der Spieler den Gegnern auf dem Parcours eben ausweichen muss. Dem Autor gefällt die Version mit den Energiesternchen allerdings besser, da er die Idee mag, dass der kleine Ninjakämpfer bei seiner Trainingsmeditation Kraft und Energie sammelt. Zudem ist der Code schon lang genug. Wenn der Leser das Spiel erweitern möchte, wäre die Option sowohl Items als auch gegnerische Sprites einzubauen, sicherlich eine Überlegung wert.

## Ein wenig Spielphysik

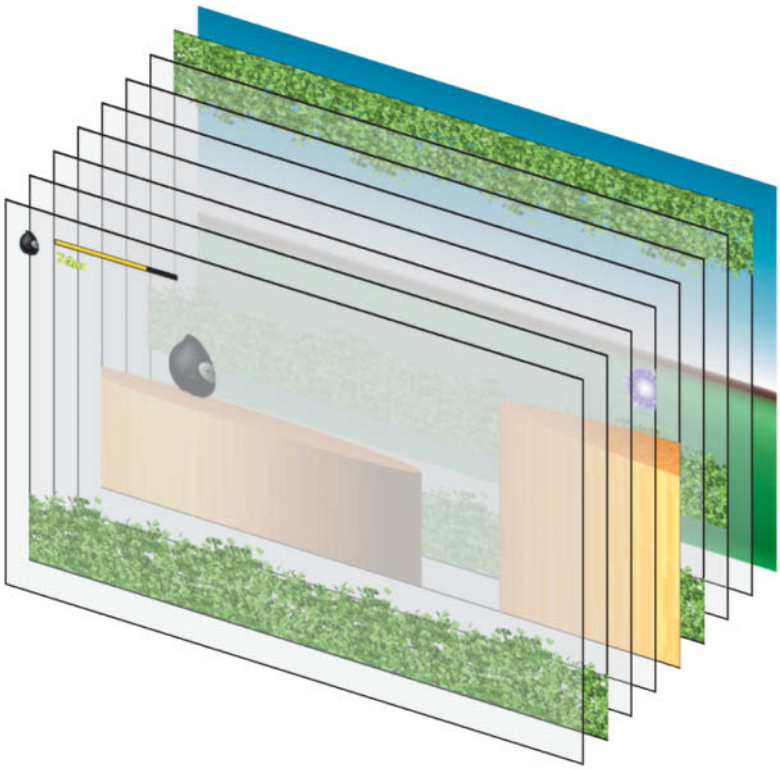
Selbst in einem kleinen Casual Game wie Ninja Course bedarf es ein wenig Physik, damit sich die Spielfigur einigermaßen agil durch das Spiel bewegt. In diesem Fall benötigt man sie beim Verhalten des Sprite, wenn sich der Ninja gerade in der Luft befindet. Das kann zwei Ursachen haben: Entweder der Ninja springt gerade eben hoch oder er befindet sich im freien Fall.

Die aktuelle y-Koordinate des Ninjas wird dabei in der Variablen *playerY* festgehalten. Ähnlich wie sich die Fortbewegung in x-Richtung bei jedem Durchlauf der Game Engine durch die Variable *speed* verändert, bestimmt die Variable *gravity* die Auf- und Abwärtsbewegung in der Luft. Da die Gravitation allerdings ständig auf den Körper des Ninjas einwirkt, wird prinzipiell in jedem Durchgang die Variable *playerY* um den Wert *gravity* (man hätte die Variable auch *speedY* nennen können) erhöht, egal ob dieser positiv (freier Fall) oder negativ ist. Lediglich bei einer Kollision mit einer Plattform von oben muss man *gravity* auf null setzen, da die Erdanziehungskraft *worldGravity* in diesem Fall durch die Plattform gestört wird. Um einen eleganten Sprung zu simulieren, muss man der Variablen lediglich einen hohen negativen Startwert geben, beispielsweise *-5*. Da *gravity* bei jedem Durchlauf um den Wert der Konstanten *worldGravity* (hier: *0.35*) erhöht wird, verlangsamt sich die Aufwärtsbewegung damit automatisch, bis nach 15 Durchläufen der Gravitationswert wieder positiv ist, und die ursprüngliche Sprungbewegung automatisch in einen freien Fall überleitet.

## Benutzereingaben

Da man nur einen sehr eingeschränkten Aktionsrahmen vorgibt – der Spieler kann über eine Berührung lediglich den Sprung des Ninjas starten –, hält sich der Eingabe-Teil des Spiels stark in Grenzen. Im Prinzip genügt es, wenn man in der bei einer Berührung des Touchscreens automatisch von iOS aufgerufenen Methode *touchesBegan* einfach die Variable *gravity = -5* setzt. Wie eben festgestellt, würde der Spieler durch diese einzelne Zeile einen kompletten Sprung von fester Höhe auslösen.

Allerdings hat die Sache einen Haken: Der Spieler könnte jederzeit springen, sogar dann, wenn sich der kleine Ninjakämpfer schon in der Luft befindet. Das Spiel würde damit ein wenig an Witz verlieren, man sollte daher klarstellen, dass sich das Spieler-Sprite nur dann in die Lüfte erheben darf, wenn es sich auf einer Plattform befindet, von der es sich abdrücken kann. Das ist der Fall, wenn *gravity* den Wert



**Die verschiedenen Sprite-Ebenen lassen sich unabhängig voneinander bewegen, was das Parallax-Scrolling vereinfacht (Abb. 5).**

null hat – man fragt den Wert daher vorher ab.

Theoretisch könnte *gravity* aber auch den Wert null annehmen, wenn sich das Sprite gerade in der Luft befindet, beispielsweise bei der Umkehr der Bewegung vom Sprung zum freien Fall. Wenn der Spieler diesen Moment geschickt abpasst, könnte er erneut zum Sprung ansetzen, was entweder ein gewolltes Feature oder einfach nur schlecht programmiert ist. Schützen kann man sich vor diesem Bug durch zwei Methoden: Entweder implementiert der Entwickler noch eine Variable, die in der Kollisionsabfrage mit den Plattformen zuvor gesetzt wird: Falls der kleine Ninja gerade normal läuft und zum Sprung ansetzen darf und diese zusätzlich in *touchesBegan* abgefragt wird. Oder aber man wählt – wie in diesem Fall – gleich zu Beginn Werte für *gravity* und *worldGravity*, die einen Nullpunkt in der Luft rein mathematisch gar nicht erst zulassen.

## Dynamische Sprunghöhe

Es gibt es in einem Jump and Run nichts Langweiligeres, als wenn das Sprite eine feste Sprunghöhe hat. Das ist vor allem ein Markenzeichen von Hobbyprogrammierern, die sich zum ersten Mal an ein solches Spiel wagen. Und gerade bei einem Spiel wie Ninja Course, bei dem gutes Timing und Gefühl für die richtige Sprunghöhe essenziell sind, wäre ein solches Manko mehr als nur hinderlich.

Doch wie kann man die Sprunghöhe über einen Touchscreen beeinflussen? Intuitiv wäre, wenn der kleine Ninja höher spränge, je fester der Anwender auf den Screen drückt. Man kann den Entwicklern bei Apple ja nicht unbedingt vorhalten, dass sie bei ihren mobilen Endgeräten Innovationen stark vernachlässigt hätten, aber einen Sensor, der die Kraft

**Wenn sich die Rahmen (Bounding-Box) beider Sprites überschneiden, findet eine Kollision statt (Abb. 6).**

**Listing 3: ViewController.h**

```
#import <UIKit/UIKit.h>
#import "Sprite.h"

@interface ViewController : UIViewController {
  // Generell:
  UIImageView *canvas;  // Gamescreen
  double z;             // Animationszaehler
  double screenWidth;   // Displaybreite
  double screenHeight;  // Displayhoehe

  // Ninja
  Sprite *player;       // Ninja-Hauptsprite
  NSMutableArray *foot; // beide Fusssprites
  BOOL ninjaJumps;      // springt?
  double playerX;       // x-Koordinate
  double playerY;       // y-Koordinate
  double speed;         // Bewegung in x-Richtung (laufen)
  double gravity;       // Bewegung in y-Richtung (fallen/springen)

  // andere Sprites
  NSMutableArray *platforms;
  NSMutableArray *bush;
  Sprite *star;

  // Head-Up-Display (HUD)
  Sprite *energyIcon;
  UIView *energyBar;
  UIView *energyBarBg;
  UILabel *score;
  double energy;    // Energie
  int distance;     // Fortschrittsanzeige
}

@property (nonatomic,strong) UIImageView *canvas;
@property (nonatomic,strong) Sprite *player;
@property (nonatomic,strong) Sprite *star;
@property (nonatomic,strong) Sprite *energyIcon;
@property (nonatomic,strong) UIView *energyBar;
@property (nonatomic,strong) UIView *energyBarBg;
@property (nonatomic,strong) UILabel *score;
@property (nonatomic, strong) NSMutableArray *platforms;
@property (nonatomic, strong) NSMutableArray *bush;
@property (nonatomic, strong) NSMutableArray *foot;

-(void)newGame;
-(void)gameEngine;
-(void)renderHeadUpDisplay;
-(void)renderBushs;
-(void)renderPlatforms;
-(void)checkIfNinjaIsOnPlatform;
-(void)renderStar;
-(void)checkIfNinjaTouchesStar;
-(void)renderNinja;

@end
```

ausliest, mit der der Spieler auf den Screen drückt, haben selbst sie nicht in ihre Produktpalette eingebaut. Man kann sich aber durch zwei Tricks behelfen. Der anspruchsvolle Ansatz ist es, den Beschleunigungssensor des Gerätes auszuwerten (beispielsweise ermittelt Apple in seiner App GarageBand über diesen Weg die Anschlagskraft auf die virtuellen Klaviertasten). Einfacher geht es nach folgendem Prinzip:

Je fester ein Spieler auf den Screen drückt, desto länger berührt er ihn auch. Das ist der Elastizität der menschlichen Haut geschuldet, die bei stärkerem Druck nachgibt, sodass der Berührpunkt stattdessen in die Breite geht. Dementsprechend länger dauert es, bis der Spieler den Finger nach starkem Druck wieder vom Screen entfernt hat. Diesen Zusammenhang von Druck zu Zeit kann man sich als Programmierer ebenfalls zu eigen machen: Je länger der Spieler drückt, desto höher springt das Sprite.

Im Programm verwendet man dazu einen weiteren Event-Handler namens *touchesEnded*, der erst auslöst, sobald eine Berührung endet. In ihm könnte man theoretisch die Zeit

**Listing 4: ViewController.m**

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>

#define worldGravity 0.35
#define maxJumpPower -10
#define maxFallingSpeed 16

@implementation ViewController
@synthesize canvas, player, star, platforms, bush, foot;
@synthesize energyIcon, energyBar, energyBarBg, score;

- (void)viewDidLoad {
  [super viewDidLoad];

  // Display: Vollbild und Dauerbeleuchtung an
  [[UIApplication sharedApplication] setStatusBarHidden:YES];
  [[UIApplication sharedApplication] setIdleTimerDisabled:YES];

  // Hintergrundbild vorbereiten
  canvas = [[UIImageView alloc] init];
  canvas.frame = [[UIScreen mainScreen] applicationFrame];
  canvas.image = [UIImage imageNamed:@"bg.png"];
  canvas.userInteractionEnabled = YES;
  self.view = canvas;

  // Displaygroesse auslesen (Vorsicht, mit Absicht vertauscht!)
  screenWidth = self.view.bounds.size.height; // = iPhone: 480 Pixel
  screenHeight = self.view.bounds.size.width; // = iPhone: 320 Pixel

  // Sprite: Buschwerk (Deko)
  bush = [[NSMutableArray alloc] init];
  for (int i = 0; i < 4; i++) {
    Sprite* tmp = [[Sprite alloc] initWithImage: @"bush.png"
                                      addToView: canvas];
    [bush addObject:tmp];
  }
  // spezielle Busch-Praeperationen
  [[bush objectAtIndex:0] x:0 y:-100 scale:1];
  [[bush objectAtIndex:1] x:0 y:140 scale:1];
  [[bush objectAtIndex:2] x:0 y:180 scale:1];
  [[bush objectAtIndex:3] x:0 y:250 scale:1];
  [[bush objectAtIndex:2] flipX:YES flipY:NO];
  [[bush objectAtIndex:0] flipX:YES flipY:YES];

  // Sprite: Plattformen
  platforms = [[NSMutableArray alloc] init];
  for (int i = 0; i < 3; i++) {
    Sprite* tmp = [[Sprite alloc] initWithImage: @"platform.png"
                                      addToView: canvas];
    [platforms addObject:tmp];
  }

  // Sprite: Energiesternchen
  star = [[Sprite alloc] initWithImage: @"star.png"
                             addToView: canvas];

  // Sprite: Ninja (besteht aus Koerper (player) und Fuessen

  // Fuesse
  foot = [[NSMutableArray alloc] init];
  for (int i = 0; i < 2; i++) {
    Sprite* tmp = [[Sprite alloc] initWithImage: @"ninjafoot.png"
                                      addToView: canvas];
    [foot addObject:tmp];
  }

  // Koerper
  player = [[Sprite alloc] initWithImage:@"ninja.png"
                               addToView:canvas];
  // Vorderen Fuss vor Koerper setzen
  [self.view bringSubviewToFront:[foot objectAtIndex:0]];
  // vordersten Busch ganz nach vorne setzen
  [self.view bringSubviewToFront:[bush objectAtIndex:3]];

  // Head-Up-Display (HUD)
  // Energieanzeige
  energyIcon = [[Sprite alloc] initWithImage:@"ninja.png"
                                   addToView:canvas];
  [energyIcon x:10 y:10 scale:0.4];

  // schwarzer Hintergrundbalken
  energyBarBg =[[UIView alloc] initWithFrame:CGRectMake(39,10,102,5)];
  energyBarBg.backgroundColor=[UIColor blackColor];
  [canvas addSubview:energyBarBg];

  // roter Energiebalken
  energyBar = [[UIView alloc] init];
  energyBar.backgroundColor=[UIColor yellowColor];
  [canvas addSubview:energyBar];
  // Fortschrittanzeige
  score = [ [UILabel alloc ] initWithFrame:CGRectMake(40,15,100,20)];
  score.textColor = [UIColor yellowColor];
  score.backgroundColor = [UIColor clearColor];
  score.font = [UIFont fontWithName:@"Arial Rounded MT Bold" size:12];
  [canvas addSubview:score];

  // Level initiieren
  [self newGame];

  // Timer fuer Game Engine starten (30fps)
  CADisplayLink *callGameEngine =
  [CADisplayLink displayLinkWithTarget:self
                              selector:@selector(gameEngine)];
  [callGameEngine setFrameInterval:2];
  [callGameEngine addToRunLoop:[NSRunLoop currentRunLoop]
                       forMode:NSDefaultRunLoopMode];
}

-(void)newGame {
  distance = 0;
  energy = 100;
  z = 0;
  speed = 3;
  playerX = 70;
  playerY = -player.height;
  gravity = 0;
  [star x:1000 y:20+arc4random()%80 scale:1];
  for (int i=0; i< [platforms count]; i++)
    [[platforms objectAtIndex:i] x:50+i*400
                                 y:80+arc4random()%160
                             scale:0.35+0.1*i];
}

-(void)gameEngine {
  // UPDATE
  z++; // unabhaengiger Animationszaehler

  // update NINJA
  // Ninja bis 20 automatisch beschleunigen
  if (speed<20) speed+=0.01;

  // Springen oder Fallen?
  if (ninjaJumps) {
    if (gravity>maxJumpPower) gravity-=1; else ninjaJumps=NO; // jump
  } else if (gravity<maxFallingSpeed) gravity+=worldGravity; // fallen

  // update HUD
  energy-=0.1;
  distance+=speed;


  // RENDER PLATTFORMEN
  [self renderPlatforms];

  // Kollision Ninja + Plattformen
  [self checkIfNinjaIsOnPlatform];

  // RENDER NINJA
  [self renderNinja];

  // RENDER BUSCHWERK
  [self renderBushs];

  // RENDER STERN
  [self renderStar];
  [self checkIfNinjaTouchesStar];
```

messen, die vergangen ist, seitdem der Spieler zu drücken angefangen hat, und davon die initiale Sprungkraft ableiten. Allerdings würde der Sprung nach dieser Methode erst dann eingeleitet, wenn die Berührung beendet wäre. Und welcher Spieler will schon ein Casual Game spielen, bei dem seine Heldenfigur erst in Aktion tritt, wenn er eine Interaktion beendet? Das muss dynamischer gehen, sprich: intuitiver.

Das Geheimnis liegt darin, dass der Startwert für die Sprungkraft immer identisch ist (in diesem Fall wird *gravity* auf *-3* gesetzt. Gleichzeitig setzt man auch die boolsche Variable *ninjaJumps*. In der Game Engine wird bei jedem Durchlauf im Update-Bereich vom Wert der Variablen *gravity* so lange ein Zähler subtrahiert (also die Sprungkraft während des Sprungs nachträglich erhöht), bis *ninjaJumps* in *touchesEnded* wieder auf *false* gesetzt wird. Um unendlich hohe Sprünge zu vermeiden, muss man die Variable allerdings auch zurücksetzen, falls eine bestimmte maximale Sprungkraft (*maxJumpPower*, hier: *-10*) erreicht ist. Und schon kann der Spieler beliebig durch seinen Parcours springen. Nun macht das tollste Spiel keinen Spaß, wenn der

**Listing 4: ViewController.m (Fortsetzung)**

```
  // RENDER HUD
  [self renderHeadUpDisplay];

  // SPIELENDE?
  if (playerY>600) [self newGame];
  if (playerX<-screenWidth/2) [self newGame];
  if (energy<=0) [self newGame];
}

-(void)renderBushs {
  for (int i = 0; i < [bush count]; i++) {
    Sprite* tmp = [bush objectAtIndex:i];

    // Buschwerk-Sprites unterschiedlich schnell bewegen und
    // nach einer Bildschirmbreite wieder zuruecksetzen
    [tmp x:fmod(tmp.position.x-speed*(0.3+i*0.3),screenWidth)
         y:tmp.position.y
     scale:tmp.scale];
  }
}

-(void)renderPlatforms {
  for (int i = 0; i < [platforms count]; i++) {
    Sprite* tmp = [platforms objectAtIndex:i];

    // Plattform bewegen
    [tmp x:tmp.position.x-speed y:tmp.position.y scale:tmp.scale];

    // Wenn Plattform vorbei...
    if (tmp.position.x+tmp.width*tmp.scale<0) {

      // ... hole Daten der hintersten Plattform...
      Sprite* last = [platforms objectAtIndex:(i+2)%3];
      double lastPos = last.position.x+last.width*last.scale;

      // ...und setze Plattform zufaellig dahinter
      // (je schneller das Spiel, desto weiter der Abstand)
      [tmp x: lastPos + fmod(arc4random(),(20*speed))
           y: 80 + arc4random()%160
       scale: 0.5 + 0.1*(arc4random()%5)]; // scale: 0.5 - 1.0
    }
  }
}

-(void)checkIfNinjaIsOnPlatform{
  // Alle Plattformen checken
  for (int i = 0; i < [platforms count]; i++) {
    Sprite* tmp = [platforms objectAtIndex:i];
    // Ueberschneiden sich deren Frames mit Ninja?
    if (CGRectIntersectsRect(player.frame, tmp.frame)) {
      // Falls Beruehrung von oben...
      if (playerY+player.height < tmp.position.y+32*tmp.scale) {
        // ...und der Ninja faellt gerade oder steht bereit...
        if (gravity>=0) {
          // ...dann Ninja fest auf Plattform setzen (kein Fallen!)
          gravity=0;
          playerY = tmp.position.y + 10*tmp.scale-player.height;
          // Falls Ninja nach links verschoben wurde, waehrend des
          // Laufens langsam wieder auf Ursprungsposition zurueck
          if (playerX < 100) playerX++;
        }
      } else {
        // andernfalls (Sprite beruehrt Plattform von links):
        // Ninja scrollt mit (bleibt an Plattformwand haengen)
        playerX-=speed;
        distance-=speed;
      }
    }
  }
}

-(void)renderStar{
  // schoene Flugbahn beschreiben (Sinus + Kosinus) + flimmern
  [star x:star.position.x-speed+sin(z/15)*2
        y:star.position.y+cos(z/20)*2
    scale:0.5+fabs(sin(z/25)*0.5)];

  // Falls Stern vorbei ist, hinten zufaellig neu ansetzen
  if (star.position.x<-100) {
    [star x:screenWidth+500+arc4random()%1000
          y:20+arc4random()%80
      scale:1];
  }
}

-(void)checkIfNinjaTouchesStar {
  if (CGRectIntersectsRect(player.frame, star.frame)) {
    // Stern hinten ansetzen
    [star x:screenWidth+500+arc4random()%1000
          y:20+arc4random()%80
      scale:1];
    // Energie erhoehen (maximal 100%)
    energy+=20;
    if (energy>100) energy=100;
  }
}

-(void)renderNinja {
  // nachtraeglich y-Koordinate updaten, da sie evtl.
  // durch Kollisionsabfrage geaendert wurde
  playerY+=gravity;

  // Ninja auf feste x-Position setzen
  // Sinusberechnung in y sorgt fuer kleine Koerperbewegung
  [player x:playerX y:playerY+sin(speed*20)*2 scale:1];

  // Fuesse animieren
  // Laufen (ist der Fall, wenn Gravitation gerade 0 ist)
  if (gravity==0) {
    // vorderer Fuss
    [[foot objectAtIndex:0] x:playerX+12-speed/2+sin(-speed*30)*7
                            y:playerY+player.height-5+cos(-speed*30)*3
                        scale:1];
    // hinterer Fuss (durch Pi um 180 Grad phasenverschoben)
    [[foot objectAtIndex:1] x:playerX+12-speed/2+sin(-speed*30+M_PI)*7
                        y:playerY+player.height-5+cos(-speed*30+M_PI)*3
                          scale:1];
  } else {
    // Sprunganimation
    // (Beine hinten bei Absprung, Beine vorne beim Landen)
    // vorderer Fuss
    [[foot objectAtIndex:0] x:playerX+9+gravity
                        y:playerY+player.height-6+fabs(gravity/3)
                          scale:1];
    // hinterer Fuss
    [[foot objectAtIndex:1] x:playerX+6+gravity
                        y:playerY+player.height-4+fabs(gravity/2)
                          scale:1];
  }
}

-(void)renderHeadUpDisplay {
  energyBar.frame = CGRectMake(40, 11, energy, 3);
  score.text = [NSString stringWithFormat: @"%dm", (int)distance/50];
}

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  // nur Springen, wenn Ninja gerade auf Plattform steht
  if (gravity==0) {
    gravity=-3;
    ninjaJumps = YES;
  }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  ninjaJumps = NO;
}

// Screen im Landscape-Modus
- (BOOL)shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation)interfaceOrientation{
  return (interfaceOrientation==UIInterfaceOrientationLandscapeLeft);
}

- (void)didReceiveMemoryWarning {[super didReceiveMemoryWarning];}

@end
```

**Onlinequellen**

| | | |
|---|---|---|
| [a] | Gameplay-Video zum hier entwickelten Ninja Course | goo.gl/aurzi |
| [b] | The GIMP | www.gimp.org/ |
| [c] | LiquidRescale für GIMP | liquidrescale.wikidot.com |
| [d] | Ninja Course in HTML5 | www.iosgames.com/html5 |
| [e] | Quellcode zum Artikel auf dem *iX*-FTP-Server | ftp://ftp.heise.de/pub/ix/developer/mobile_sonderheft_2012/voelcker_iOSgame.zip |

Spieler nichts erreichen kann. Da Ninja Course kein definiertes Ziel hat, benötigt der Spieler etwas anderes Messbares, mit dem er seine Erfolge vergleichen und sich gegebenenfalls darüber freuen kann. Hier kommt das Head-up-Display ins Spiel (Abbildung 7). Es bezeichnet die kleinen Icons am Displayrand, mit denen, je nach Spielgenre, üblicherweise die Anzahl der Leben, die Restenergie, die Anzahl der verbliebenen Schüsse und die aktuelle Levelnummer angegeben wird.

## Head-up-Display und Spielerbindung

Das einzige, was der Spieler in Ninja Course erreichen kann, ist eine maximale Strecke, die er zurücklegt, bis er eine Plattform verfehlt oder nicht mehr genügend Energie hat. Dem Spieler gibt man diese Messlatte, in dem man einfach eine Variable *distance* einführt, die sich – mathematisch einfach – mit jedem Durchlauf der Game Engine um den Wert *speed* erhöht und die, um die Größenordnung zu wahren, über eine kleine Formel im Label *score* samt Größeneinheit *m* umgerechnet und ausgegeben wird.

Da der kleine Ninja aber auch mit jedem Schritt Energie verliert, könnte der Spieler ein verstärktes Interesse an einen intuitiven Überblick über seine Restenergie haben. Man muss sie nicht plump als Zahl ausgeben, sondern kann sie grafisch in Form eines Balkens *energyBar* darstellen, der sich einfach durch eine einfache View mit rotem Hintergrund erzeugen lässt. Damit er einen virtuellen Vergleichswert hat, legt man einen zweiten Balken *energyBarBg* in Schwarz darunter, der den vollen Energieausschlag symbolisieren soll.

## Animation

Damit der Spieler den Balken als eigene Energie zu verstehen begreift, könnte man ihm entweder ein kleines Herz-Symbol (das ultimative Icon für Leben oder Lebensenergie) davor setzen, oder aber – um sich eine zusätzliche Grafik zu sparen – einfach seinen kleinen Ninja auf halbe Größe herunterskalieren und ihn stattdessen als Symbol verwenden.


**Mit einfachen Mitteln lässt sich ein ansprechendes Head-up-Display für den schnellen Überblick erzeugen (Abb. 7).**

Nun ist das Spiel fast fertig und es geht an die Feinarbeit: die Animation. Um eine *ImageView* (der internen Form des Sprite) in Objective-C animieren zu können, gibt es zwei Möglichkeiten. Entweder übergibt der Entwickler der *ImageView* ein Array mit mehreren Grafiken und bestimmt die Dauer der Gesamtanimation, oder er animiert die Bewegungsabläufe programmatisch (falls sie nicht zu komplex sind). Zuerst sei auf die erste Version (hier ohne die *Sprite*-Klasse) eingegangen:

```
UIImageView animation = [[UIImageView alloc] init];
animation.animationImages = [NSArray arrayWithObjects:
  [UIImage imageNamed:@"frame1.png"],
  [UIImage imageNamed:@"frame2.png"],
  [UIImage imageNamed:@"frame3.png"],
  [UIImage imageNamed:@"frame4.png"],
  nil]];
animation.animationDuration = 1;
animation.animationRepeatCount = 0;
[animation startAnimating];
[self.view addSubview: animation];
```

Hier wird zunächst die ImageView *animation* initialisiert und ihr ein Array aller Einzelbilder in der richtigen Animationsfolge übergeben. Die Dauer der Gesamtanimation gibt man in Sekunden an. Dann nennt man die Anzahl der Wiederholungen. Ist der Wert von *animationRepeatCount* gleich null, wird die Animation endlos wiedergegeben. Zuletzt startet man die Animation über *startAnimating* und projiziert sie auf die Hauptview. Das ist elegant und gibt dem Programmierer viel Freiheit, dafür muss sich der Grafiker sehr bemühen, damit die Animation rund läuft. Zugleich hat dieser Ansatz den Nachteil, dass er immer in derselben Geschwindigkeit abläuft. Zwar kann der Entwickler die *animationDuration* während der Animation ändern, aber man möchte bei Ninja Course noch etwas flexibler bleiben.

In GIMP wurden dafür extra die Füße des Ninjas nicht im Hauptbild, sondern als Extragrafik angelegt. Sie kommen nun in doppelter Version – für jeden Fuß dieselbe Ausgangsgrafik – zum Einsatz. Durch einfache trigonometrische Funktionen kann man die Füße in Abhängigkeit der aktuellen Geschwindigkeit (*speed*) anpassen. Dazu nutzt man eine kombinierte Sinus-Kosinus-Funktion für die Kreisberechnung: Wenn man von einem von *speed* abhängigen Wert den Sinuswert nimmt und diesen der aktuellen x-Koordinate eines Sprite hinzurechnet, und dasselbe mit dem Kosinus auf der y-Koordinate anwendet, erhält man eine Kreisfunktion. Allerdings ist diese minimal, da sich Sinus und Kosinus bekanntlich nur im Zahlenraum von -1 bis +1 bewegen. Aus diesem Grund multipliziert man die Werte mit der maximal gewünschten Amplitude. Damit erreicht man, dass sich die Füße rund um eine vorher definierte Position am unteren Ende des Ninja-Haupt-Sprites bewegen. Ist die x-Amplitude höher als die y-Amplitude, erhält man eine kleine, falsche Ellipse und der Fuß bewegt sich wie in einem Zeichentrickfilm.

Damit sich der andere Fuß nicht deckungsgleich bewegt, muss man ihn um eine halbe Phase verschieben, was damit erreicht wird, dass man zu dem von *speed* abhängigen Ausgangswert einfach ein π (hier: 3.14) addiert. Schon hat man seinem kleinen Ninjakämpfer Beine gemacht. Gleichzeitig kann man über eine Sinus-Funktion das Hauptsprite während der Laufbewegung dezent heben und senken, sodass es aussieht, als würde es atmen. Ebenso flexibel kann man schließlich in Abhängigkeit der y-Bewegung die Füße nach hinten und vorne verschieben, sodass bei einem Sprung mit