```
// z-Index sortieren von oben nach unten
// damit Staemme von hoeheren Plattformen
// die Plattformen darunter nicht verdecken!
for (int zIndex = 0; zIndex < screenHeight; zIndex +=rasterY) {

  for (int sortCurrentVisiblePlatform = 0;
       sortCurrentVisiblePlatform < [platforms count];
       sortCurrentVisiblePlatform++) {

    Sprite *existingPlatform =
      [platforms objectAtIndex:sortCurrentVisiblePlatform];
    if ((existingPlatform.frame.origin.y <= zIndex) &&
        (existingPlatform.frame.origin.y > zIndex-rasterY))
      [self.view bringSubviewToFront:existingPlatform];
  }
}
// z-Index sortieren Ende
```

4.5.4 Detaillierte Animationen

Spannender sind die Möglichkeiten, wie wir unsere Animationen noch »realistischer« gestalten können, sofern man das überhaupt von einer Animation bei einem niedlich gezeichneten 2D-Jump'n'Run behaupten kann. Schließlich bewegt unsere Heldin – ich habe sie als unsere erste »menschliche« Spielfigur überhaupt auf den Namen »Sally« getauft – ihre Füße immer gleichmäßig schnell, egal mit welcher Geschwindigkeit sie läuft. Diese Eigenschaft zu verbessern mag bei diesem Spiel ein wenig pedantisch wirken, aber – hey! – gelernt ist gelernt. Es ist auch kein großer Akt, die Animationsschritte so zu steuern, dass die Figur dabei mit den Fußsohlen nicht wie auf Eis gleitet, sondern richtige Schritte ausführt. Dazu schauen wir uns die Einzelbilder unserer animierten Spielfigur genauer an:

Abb. 4–57 *Pro Animationsschritt wird der Ballen des Standfußes jeweils um 2 Pixel nach hinten verschoben.*

Mit jedem Einzelbild wird der jeweilige Standfuß – derjenige, der von vorne nach hinten wandert – um je 2 Pixel nach hinten versetzt. Würde sich unser Sprite gleichzeitig nun pro Schritteinzelbild um je 2 Pixel fortbewegen (oder wie hier: der Hintergrund entgegenbewegen), dann hätten wir die perfekte Animation für die Geschwindigkeit 2 Pixel/Bildwechsel. Bewegt sich das Sprite zwischen zwei Bildwechseln nur um ein Pixel voran, sieht es aus, als würde es einen Moonwalk auf der Stelle tanzen. Bewegt es sich hingegen mehr als 2 Pixel pro Bildwechsel weiter, entsteht der Eindruck, als gleite das Sprite auf einem glattem Untergrund.

Man kann einem Dschungel ja viel vorwerfen, aber Glatteis gehört nicht gerade zu seinen typischen Eigenschaften. Zumindest in den nächsten 100 Jahren kann man davon ausgehen, dass die Plattentektonik handelsübliche Dschungel noch nicht weit genug in Richtung Antarktis verschoben hat, um dort Eisblöcke zum Standardrepertoire gehören zu lassen.

Um unser Sprite nun also auch bei langsamerer und schnellerer Fortbewegung synchrone Fußarbeit leisten zu lassen, müssen wir von dem Prinzip der einmaligen Animationsdefinition über `animation-Images` und `animationDuration` bzw. innerhalb unserer Klasse `Sprite` über `setAnimationTyp` weggehen und die Einzelbilder je nach Fortschritt »von Hand« wechseln.

Dazu deaktivieren wir erst einmal die beiden Stellen in `ViewController.m`, in denen die Heldin animiert wird. Am einfachsten finden wir diese, indem wir in Xcode nach dem Vorkommnis von »`[player setAnimationTyp` « suchen und die Fundstellen auskommentieren:

```
/*
 [player setAnimationTyp:[animationSource objectAtIndex:walk]
              spriteTyp:0
               duration:1.0
                 repeat:0];
*/
[…]

// Animation an Verhalten anpassen
/*
int playerLookedLikeBefore = playerLooksLike;

// Stehen
[…]

// Animation nur aendern, wenn nicht schon gesetzt!
if (playerLooksLike!=playerLookedLikeBefore) {
  […]
}
*/
```

Nun kommt uns wieder die `zaehler`-Variable zu Hilfe, die ja für nichts anderes als die Position der Figur (bzw. eigentlich des Viewports) innerhalb des Levels steht und je nach Richtung zunimmt oder kleiner wird. Das heißt aber auch, für jede

Position auf der x-Achse innerhalb des Levels hat unsere Spielfigur eine unique ID, also einen bestimmten, unverwechselbaren Wert. Da unsere Animation aus 15 Einzelbildern besteht, ordnen wir also nun jeder Position im Level einen bestimmten Animationsframe zu. Das heißt, an der Stelle, an der unsere Heldin gerade den rechten Fuß nach vorne nimmt, wird sie immer den Fuß nach vorne nehmen, egal, ob sie sich am Boden oder auf einer höher gelegenen Plattform befindet. Die einzige Ausnahme soll sein, wenn sie springt oder fällt. Und wenn ich jetzt noch das Wort »Modulo« fallen lasse, dürfte Ihnen nun auch ungefähr die Methode klar sein, auf die ich hinausmöchte.

Die `animationSource` steht ja bereits, darauf können wir nun über den Index zugreifen. An der Stelle, an der wir oben eben noch den Code auskommentiert haben, setzen wir folgende Zeilen hin:

```
// Animation an Verhalten anpassen

// springen und fallen
if (playerSpeedY!=0) [player setSpriteTyp:
                        [[animationSource objectAtIndex:walk]
                          objectAtIndex:3] typ:0];
// rechts
else if (playerSpeedX>0) [player setSpriteTyp
                          [[animationSource objectAtIndex:walk]
                           objectAtIndex:(zaehler/2+100000)%15] typ:0];
// links
else if (playerSpeedX<0) [player setSpriteTyp:
                          [[animationSource objectAtIndex:walk]
                           objectAtIndex:14-(zaehler/2+100000)%15] typ:0];
// stehen
else if (playerSpeedX==0) [player setSpriteTyp:
                           [[animationSource objectAtIndex:walk]
                            objectAtIndex:0] typ:0];
/*int playerLookedLikeBefore = playerLooksLike;
// Stehen
```

Die entscheidende Zeile ist in der für die Rechtsbewegung zu finden:

```
(zaehler/2+100000)%15
```

Da sich der Standfuß unserer Heldin von Einzelbild zu Einzelbild jeweils um 2 Pixel fortbewegt, müssen wir also nur alle 2 Pixel einen Bildwechsel erfolgen lassen. Deswegen teilen wir `zaehler` (welche für die Position steht) erst durch `2`. Bevor wir von der ganzen Summe den Modulo von 15 nehmen, damit wir eine ständig fortlaufende und wiederholende Zahlenfolge von 0 bis 14 erreichen, addieren wir noch 100.000 zum Dividenden. Damit verhindern wir, dass wir negative Werte erhalten, wenn wir mit unserer Spielfigur zu weit nach links laufen, weil wir in unserem Animations-Array nicht auf negative Werte zugreifen können. Leider kann man das Problem nicht über `fabs` (für den absoluten Wert ohne Vorzeichen) lösen, da von der 0 aus nach links gesehen wieder der Wert 1

statt der 14 errechnet würde, womit unser Sprite plötzlich rückwärts liefe, weil die Animation verkehrtherum abgespielt werden würde.

Dieser Effekt tritt leider auch auf, wenn unser Sprite nach links läuft. Weil wir die Einzelbilder nur spiegeln, die `zaehler`-Variable aber nicht umdrehen können, läuft unsere Heldin anscheinend rückwärts. Dem wirken wir entgegen, indem wir für die Linksbewegung den für nach rechts berechneten Wert einfach von 14 abziehen.

Unser erklärtes Ziel für diese Aktion war ja eigentlich, die Animation realistischer zu machen. Wenn wir die App nun allerdings kompilieren und anspielen, sieht die Animation noch viel schlimmer aus. Warum? Weil unser Sprite nun völlig hyperaktiv durch die Level flitzt und bei besonders hoher Geschwindigkeit einige Einzelbilder sogar ausgelassen werden. Das war also ein Griff in die Kiste.

Jetzt können wir natürlich noch ein Weilchen mit den Werten experimentieren, aber um es kurz zu machen: Ersetzen wir in den seitlichen Bewegungen die 2 durch die 5, nehmen wir unserer Animation zwar die Exaktheit, aber dafür läuft unsere Heldin angenehm geschmeidig. Verrückt! Um die Animation »realistischer« wirken zu lassen, müssen wir sie künstlich verfälschen.

Das hat zum einen die Ursache, dass das Auge der exakten Fußposition bei der Geschwindigkeit nicht folgen kann, und zum anderen, dass wir durch zahlreiche schlechte Cartoons im Fernsehen und anderen Casual Games, bei denen genauso verfahren wurde, »abgehärtet« wurden. Wie dem auch sei, es funktioniert. Also nutzen wir es!

4.5.5 Parallax-Scrolling – ein Hauch von 3D

So! Seitdem wir das erste Mal über Scrolling gesprochen haben, sind nun fast 150 Seiten vergangen, bis wir endlich, endlich zur schönsten Form des Scrollings gekommen sind. Das heißt, eigentlich ist Parallax-Scrolling keine eigenständige Scroll-Technik, sondern nur eine Erweiterung, aber was für eine! Ohne auch nur einen Hauch von 3D-Technik zu programmieren, können wir damit eine räumliche Tiefe schaffen. Gleichzeitig wirkt unser Spiel dadurch detailreicher und auch professioneller. Allein dafür lohnt sich doch das Durchlesen dieses Unterkapitels! Zuerst einmal benötigen wir folgende drei Grafiken:

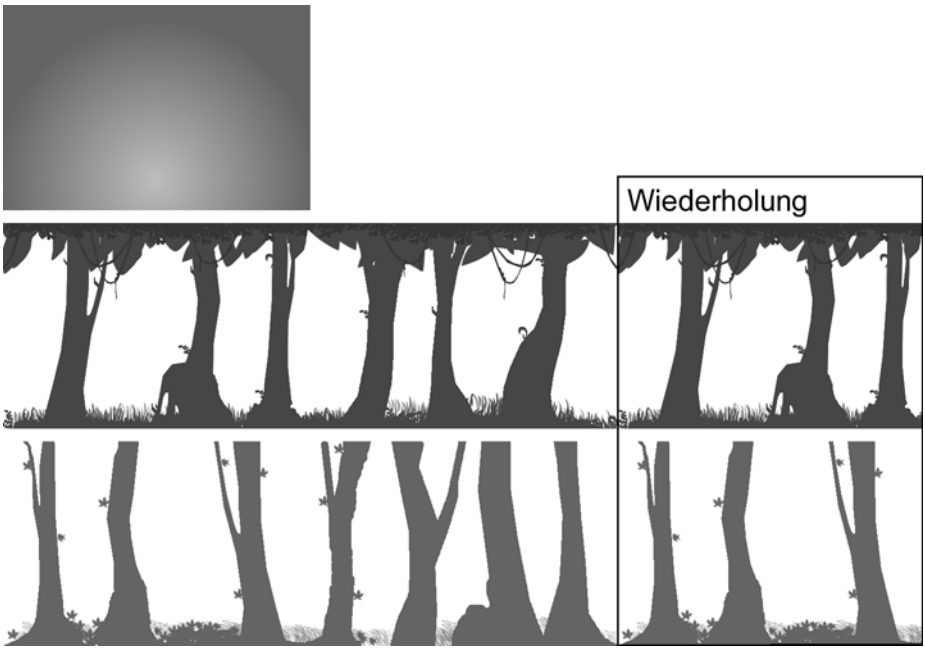

Abb. 4–58 *bg.png (480×320), layer0.png (1440×320) und layer1.png (1440×320)*

Bei den beiden Grafiken `layer0.png` und `layer1.png` ist jeweils darauf zu achten, dass diese aus drei Abschnitten à 480×320 bestehen, wobei der linke und der rechte Abschnitt jeweils optisch identisch sind. Das ist notwendig, damit wir diese Hintergrundebenen später endlos durchscrollen können.

Wir bauen also unseren Dschungel optisch noch ein wenig auf, allerdings nur im Hintergrund, so dass dieser lediglich der Dekoration unseres Spiels dient. Bis auf den Hintergrundverlauf `bg.png`, der fix auf seiner Position bleiben wird, verhalten sich `layer0.png` und `layer1.png` wie unterschiedlich schnelle Ebenen, wobei die vordere der beiden (`layer0.png`) die schnellste sein wird und dabei trotzdem noch langsamer als das Spiel im Vordergrund sein muss. In der Kombination gibt dieses Ebenenspiel allerdings schon schön was her:

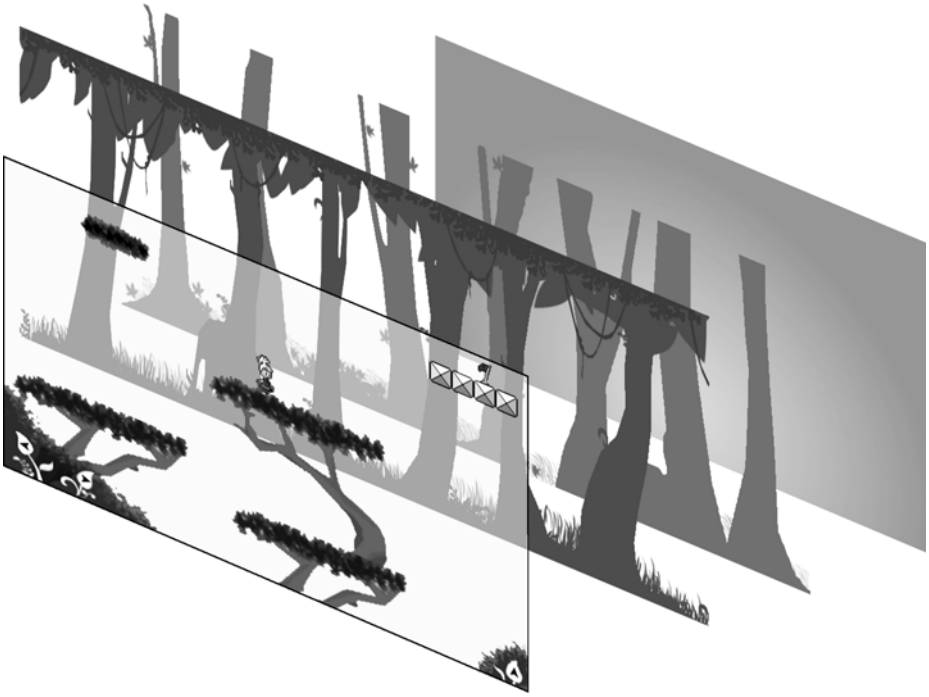

Abb. 4–59 *Die vier Schichten: Canvas (inkl. Spiel), Hintergrund1 (Wald, beweglich), Hintergrund2 (Wald, beweglich) und der eigentliche Hintergrund (Farbverlauf, fix)*

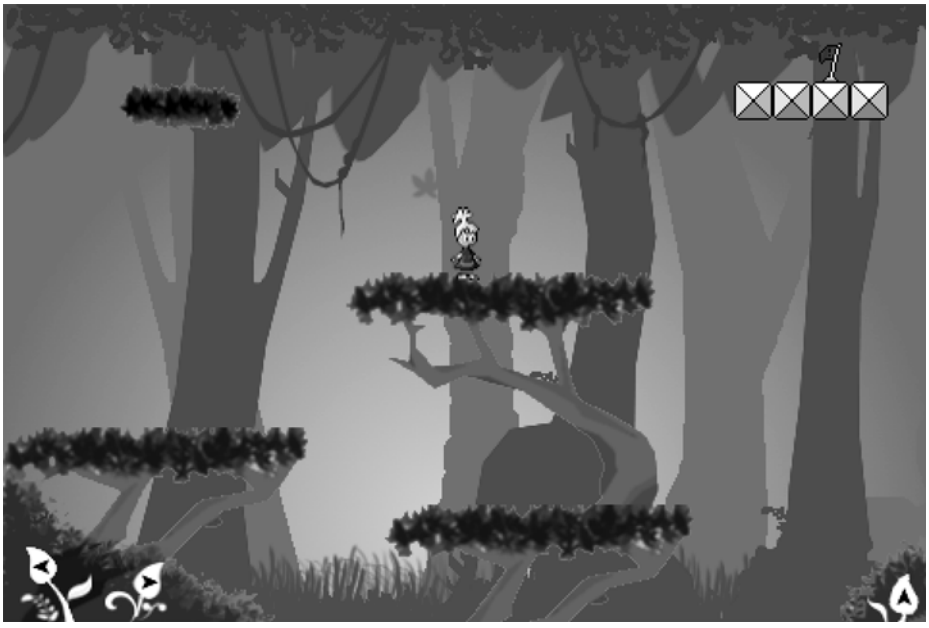

Abb. 4–60 *Die vier Schichten im Viewport: Die Baumstämme im Vordergrund können sich optisch vom Hintergrund noch nicht so richtig absetzen.*

Allerdings vermischen sich alle Ebenen farblich so stark, dass auf den ersten Blick nicht sofort erkennbar ist, was zum Vordergrund und was zum Hintergrund gehört. Zwar sind die Plattformen durch ihre dunklen Farbtöne und ihre klaren Strukturen im Vergleich zu den nur angedeuteten und einfach gehaltenen Bäumen im Hintergrund deutlich abgesetzt, aber die Bäume und Äste, die diese Blätterplattformen halten, vermengen sich mit dem Hintergrund.

Um den Vordergrund deutlicher herauszuarbeiten, könnten wir den Plattformen noch schwarze Outlines mitgeben oder alternativ die drei Hintergrundebenen heller gestalten.

Um allerdings noch einen schöneren Effekt kennenzulernen, arbeiten wir mit einem Layer `fog.png`, der als Nebelschicht zwischen Spiel und Hintergrund gesetzt wird. Dieser ist halbtransparent (bereits im PNG festgelegt und nicht über den Alphakanal der `UIImageView`, weil der Effekt sonst zu sehr zu Lasten der Performance geht), mit ein paar weißen Schleierwolken und vereinzelten Strukturen, die dem Spiel eine magische und märchenhafte Aura verleihen. Der Layer könnte z. B. so aussehen:

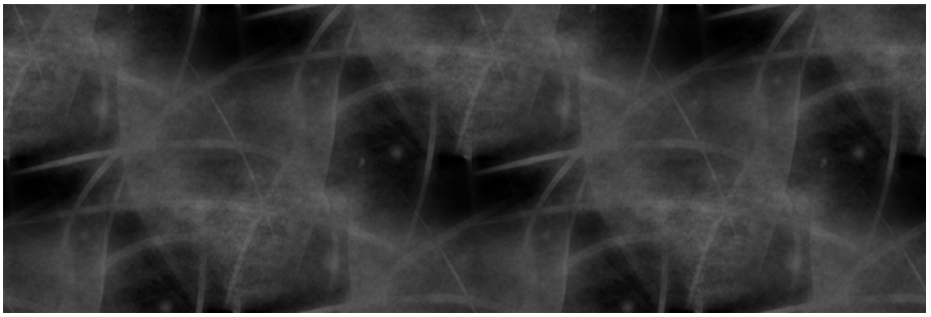

Abb. 4–61 *Märchenhafter Schleier: fog.png (zur besseren Sichtbarkeit auf schwarzem Hintergrund)*

Wichtig ist allein, dass das Tile zweigeteilt ist, wobei sich beide Hälften optisch bis aufs Pixel gleichen! Wenn diese zusätzliche Schicht hinter der Spielebene, aber noch vor der ersten Hintergrundebene fix gesetzt wird, erhalten wir folgenden Effekt, der eine größere Distanz zwischen Vorder- und Hintergrund herstellt (s. Abb. 4–62).

Um den gesamten Hintergrund mit allen Ebenen nun langsam mitscrollen zu lassen, nehmen wir erneut unsere Variable `zaehler` zu Hilfe, an deren Wert sich bereits das Scrollen der Spielebene orientiert. So können wir die Synchronität der insgesamt vier scrollenden Ebenen gewährleisten.

Abb. 4–62 *So soll unser Spiel anschließend aussehen, die Hintergrundebenen scrollen dabei in unterschiedlichen Geschwindigkeiten mit.*

Im Code sieht das Ganze wie folgt aus: Zuallererst müssen wir die `UIView canvas` in eine `UIImageView` umwandeln. Die drei Ebenen (2×Baum und 1×Nebel) entwerfen wir jeweils als Objekt der Klasse `Sprite`:

```
@interface ViewController : UIViewController {
  // Das duerfte bekannt sein:
  UIImageView *canvas;

  […]
  // Parallax-Scrolling
  Sprite *layer0;
  Sprite *layer1;
  Sprite *fogLayer;
  double fogMove;
}

@property (nonatomic,strong) UIImageView *canvas;
[…]
@property (nonatomic,strong) Sprite *layer0;
@property (nonatomic,strong) Sprite *layer1;
@property (nonatomic,strong) Sprite *fogLayer;
@end
```

Listing 4–29 *ViewController.h*

```
#import "ViewController.h"
#import "Sprite.h"
#include "constants.h"

@implementation ViewController
@synthesize canvas, joypad, jumpButton, player, imageSource;
@synthesize animationSource, platforms, levelSource;
@synthesize platformGraphicSource;
@synthesize layer0, layer1, foglayer;
 […]

- (void)loadView {

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

  // Hintergrund vorbereiten
  canvas = [[UIImageView alloc] initWithFrame:[[UIScreen mainScreen]
                                                applicationFrame]];
  [canvas setImage:[UIImage imageNamed:@"bg.png"]];
  canvas.userInteractionEnabled = YES;
  canvas.multipleTouchEnabled = YES;
  self.view = canvas;


  // Breite und Hoehe speichern (vorsicht: landscape!)
  screenWidth = self.view.bounds.size.height;
  screenHeight = self.view.bounds.size.width;

  // Parallax-Ebenen

  // erst hinterste Ebene
  layer1 = [[Sprite alloc] initWithImage: [UIImage imageNamed:@"layer1.png"]
                spriteTyp:0
               parentView:canvas];

  // dann vordere Ebene
  layer0 = [[Sprite alloc] initWithImage: [UIImage imageNamed:@"layer0.png"]
                spriteTyp:0
               parentView:canvas];

  // zuletzt die Nebel-Ebene
  fogLayer = [[Sprite alloc] initWithImage: [UIImage
imageNamed:@"fairyfog.png"]
                spriteTyp:0
               parentView:canvas];
  […]
}

[…]
```

```
- (void)gameEngine {

  if (normalAction) {

    // Zaehler fuer Animationen
    zaehler+=playerSpeedX;

    [self platformEngine];
    [self parallaxScrolling];
    [self playerEngine];
    [player moveBy:0 y: playerSpeedY];

    […]
  }
}
```

Listing 4–30 *ViewController.m*

Und jetzt die erschreckend herrlich kurze Methode `parallaxScrolling`, die an Übersichtlichkeit und Logik nur extrem schwer zu überbieten ist:

```
- (void)parallaxScrolling {
  fogMove+=0.5;
  [layer0 setCenter:-((960 + zaehler)/2) % 960 + layer0.frame.size.width/2
                  y:screenHeight/2];
  [layer1 setCenter:-((960 + zaehler)/3) % 960 + layer1.frame.size.width/2
                  y:screenHeight/2];
  [fogLayer setCenter:-fmod((480 + zaehler + fogMove)/2, 480) +
                      fogLayer.frame.size.width/2
                    y:screenHeight/2];
}
```

Inhaltlich alles klar? Die vielleicht nicht sofort verständlichen Zahlen `960` und `480` stehen in diesem Fall einfach für die Breite der jeweiligen Ebene. Während die Baum-Ebenen jeweils 960 Pixel (+ weitere 480 Pixel, die mit dem linken Drittel identisch sind) breit sind, haben wir die Nebel-Ebene auf 480 Pixel (plus die gleiche Anzahl identischer Pixel) Breite belassen. Ansonsten passiert in den vier Codezeilen Folgendes: `layer0` und `layer1` werden zu einem Bruchteil der normalen Scroll-Geschwindigkeit fortbewegt, und zwar so weit, bis das rechte Drittel der Ebene vollständig im Viewport sichtbar ist. Genau in diesem Moment wird sie wieder auf ihre Ursprungsposition ganz nach links gebracht. Der Spieler merkt diesen Unterschied nicht, weil die beiden Drittel optisch identisch sind, sich darauf die gleichen Bäume, Blätter und Gräser an exakt denselben Positionen befinden. Lediglich das mittlere Drittel kann frei gestaltet werden. Dieses hätten wir auch weglassen und uns auf zwei Hälften beschränken können (wie wir es z.B. bei der Nebel-Ebene gemacht haben), aber bei den Bäumen fallen sich schnell wiederholende Strukturen eher auf als bei einem verschwommenen Nebelmotiv.

Abb. 4–63 *Das linke und das rechte Drittel sehen identisch aus.*

Beim Nebel-Layer sorgen wir dafür, dass er sich auch während Standphasen der Heldin weiterbewegt. Dafür haben wir die kleine Hilfsvariable `fogMove` in den Code gemogelt, die für eine geringfügige, aber stete Bewegung sorgt.

An dieser Stelle käme bei einer interaktiven Multimediapräsentation ein Tusch, gefolgt von den Fanfaren und Standing Ovations, denn das war schon alles, was wir über Parallax-Scrolling wissen müssen!

4.5.6 Einäugige Sumpfmonster

Bis jetzt muss man schon ganz schön viel Geschick an den Tag legen, um die Level zu meistern, aber noch spannender wird es natürlich, wenn es Gegenspieler gibt.

Bei »Neptune Patrol« waren die Aliens noch relativ leicht zu programmieren, schließlich hatten wir eine durchgängig inkrementierende Zählervariable. Wenn sich diese nun sowohl erhöhen als auch verkleinern oder gar negative Werte annehmen kann, dann fliegen unsere Aliens, deren Choreografie ja auf `zaehler` basierte, durchaus auch rückwärts oder bleiben mitten in der Luft stehen. Das wirft auf die Aliens dann ein intellektuell recht ungünstiges Licht.

Natürlich können wir einfach eine zweite Zählvariable einführen und die Choreografien daran ausrichten, aber sind wir mal ehrlich: In Jump'n'Runs dominiert eine andere Sorte an Choreografie: die der dumm vor sich hindümpelnden Plattformaufundabläufer. Und dafür sind einäugige Sumpfmonster doch die absolute Idealbesetzung:

Abb. 4–64 *Sumpfmonster (Bufo zyklopensis, engl. Bogeye = »Sumpfauge«) das fieseste Geschöpf im Dschungel*

Also fügen wir diese unseren Leveldateien hinzu, indem wir auf die Plattformen, auf denen ein Sumpfmonster sein träges und tragisches Dasein pflegen soll, ein kleines @-Zeichen setzen:

```
-------------------------
----------------------X-
------0--------------1111
-------------------------
-------------------------
-------------------------
-------------@-----------
-----------2-------------
-------------------------
----#--------------------
-------------------------
--3----------------------
-------------------------
------------3------------
-------------------------
-------------------------
```

Listing 4–31 *level1.lvl (mit Monstern)*

In `loadLevel` müssen wir dieses neue Symbol natürlich abfangen:

```
- (bool)loadLevel: (int)currentLevel {

  […]
  if ([part isEqualToString:@"^"]) spriteTyp=spring; // Sprungfeder
  if ([part isEqualToString:@"X"]) spriteTyp=flag; // Zielflagge
  if ([part isEqualToString:@"*"]) spriteTyp=extralife; // Extraleben
  if ([part isEqualToString:@"@"]) spriteTyp=bogeye; // Sumpfmonster
  […]
}
```

Damit auch die korrekte Grafik dafür angezeigt wird, müssen wir sie in `loadView` dem dafür verantwortlichen Array bekannt machen (und zwar an letzter Stelle):

```
[platformGraphicSource addObject:[UIImage imageNamed:@"bogeye0.png"]];
```

Die Null im Dateinamen deutet schon darauf hin, dass wir das Monsterchen animieren wollen, und in der Tat, als letztes Element des Arrays `animationSource` wird unsere Monsterbewegung hinzugefügt:

```
// Sumpfmonster
[animationSource addObject:[NSArray arrayWithObjects:
  [UIImage imageNamed:@"bogeye0.png"],
  [UIImage imageNamed:@"bogeye1.png"],
  [UIImage imageNamed:@"bogeye2.png"],
  [UIImage imageNamed:@"bogeye3.png"],
  [UIImage imageNamed:@"bogeye4.png"],
  [UIImage imageNamed:@"bogeye5.png"],
  nil]];
```

Abb. 4–65 *Das animierte Sumpfauge*

In den Konstanten (`constants.h`) definieren wir nun noch

```
//Animation
[…]
#define bogeyewalk 4
[…]
// Monster
#define bogeye 8
#define bogeyeSpeed 2
```

damit wir nicht über irgendwelche Zahlen, die wir auswendig lernen müssen, darauf zugreifen können, und schon geht's los mit der Implementierung. In der `gameEngine` sorgen wir für einen Aufruf der Enemy-Engine mittels `[self enemyEngine]`, und programmieren diese wie folgt:

```
- (void)enemyEngine {

  // Plattformcheck START
  for (int currentEnemy=0;currentEnemy<[platforms count];currentEnemy++) {

    Sprite *enemySprite = [platforms objectAtIndex:currentEnemy];
    int spriteId = [enemySprite sId];
   int sTyp= [[[levelSource objectAtIndex:spriteId] objectAtIndex:0] intValue];

    // Monster bewegen
    if (sTyp==bogeye) {

      // Monster nach vorne setzen
      [self.view bringSubviewToFront:enemySprite];

      // auf Kollision mit allen anderen Plattformen pruefen
      for (int otherVisiblePlatforms=0;
           otherVisiblePlatforms < [platforms count];
           otherVisiblePlatforms++) {

        if (otherVisiblePlatforms!=currentEnemy) {
         Sprite *otherPlatform=[platforms objectAtIndex:otherVisiblePlatforms];

          // Beruehrung mit Plattform?
          if ([enemySprite detectCollisionWith:otherPlatform]) {

            // Monster an Plattformrand?
            if (enemySprite.center.y < otherPlatform.frame.origin.y) {

              if (enemySprite.center.x < otherPlatform.frame.origin.x) {
                if ([enemySprite movingDirection]==right) {
                  // Kehrtwende
                  [enemySprite setMovingDirection:
                              -[enemySprite movingDirection]];
                  [enemySprite mirrorSprite:left];
                }
             } else if (enemySprite.center.x > otherPlatform.frame.origin.x +
                         otherPlatform.frame.size.width) {
                if ([enemySprite movingDirection] == left)  {
                  // Kehrtwende
                  [enemySprite setMovingDirection:
                              -[enemySprite movingDirection]];
                  [enemySprite mirrorSprite:right];
                }
              } else {
                [enemySprite setCenter:enemySprite.center.x
                                  y:otherPlatform.frame.origin.y -
                                     enemySprite.frame.size.height/3];
              }
            }
          }
```

```
            // Beruehrung mit Plattform ENDE
          }
        }
        // Sprite bewegen
        [enemySprite moveBy:-bogeyeSpeed*[enemySprite movingDirection] -
                            playerSpeedX y:0];

      }
    }
    // Plattformcheck ENDE

}
```

Das Vorgehen ist schnell erklärt: Aus dem Code wird ersichtlich, dass wir – ähnlich wie bei »Noodle Jump« – Monster als eine spezielle Art von Plattformen ansehen, die sich durch ihr Verhalten von diesen abgrenzen. In diesem Fall ist es ihre Beweglichkeit.

Dafür laufen wir in einer Schleife alle sichtbaren Plattformen einmal durch und prüfen für den Fall, dass eine davon vom Typ `bogeye` ist, diese in einer Unterschleife auf eine Kollision mit allen anderen Plattformen. Ist dies der Fall und das Monster befindet sich **auf** der Plattform (was wir anhand eines y-Koordinatenvergleichs feststellen können), dann soll sich das Monster fortbewegen.

Durch die Kombination von Kollisionsabfrage und y-Koordinate definieren wir für das Sprite einen Raum, in dem es sich aufhalten kann. In diesem kann es sich vom linken bis zum rechten Rand hin- und herbewegen.

Nun haben wir den Dschungel zwar mit Sumpfmonstern bevölkert, aber noch machen sie das Spiel nicht komplizierter, sondern verhalten sich eher wie reine Dekoelemente. Eine Kollisionsabfrage ist allerdings schnell eingebaut. Für Jump'n'Runs üblich gestalten wir diese so, dass es bei seitlicher Bewegung für die Spielfigur ungünstig endet, wohingegen bei einem gezielten Sprung auf das Monster dieses nicht nur die Plattform, sondern auch sein hiesiges Dasein verlässt:

```
- (void)enemyEngine {

  // Plattformcheck START
  […]

  // Plattformcheck ENDE


  // Kollision mit Player START
  for (int currentEnemy=0;currentEnemy<[platforms count];currentEnemy++) {

    Sprite *enemySprite = [platforms objectAtIndex:currentEnemy];
    int spriteId = [enemySprite sId];
   int sTyp= [[[levelSource objectAtIndex:spriteId] objectAtIndex:0] intValue];
```

```
        // Monster?
        if (sTyp==bogeye) {

          if ([enemySprite detectCollisionWith:player]) {

            // Player kollidiert mit Sumpfmonster?
            if ([player detectCollisionWith:enemySprite]) {

              // wenn in Fallbewegung: Sumpfmonster entfernen
              if (playerSpeedY>0) {

                // kleiner Sprung
                playerSpeedY=-6;
                [platforms removeObject:enemySprite];

                [UIView transitionWithView:self.view
                                  duration:0.5
                                   options:UIViewAnimationOptionCurveEaseIn
                                animations:^{
                  [enemySprite moveBy:0 y:100];
                  [enemySprite setAlpha:0.0];
                } completion:^(BOOL finished) {
                  [enemySprite removeFromSuperview];
                }];
              // ansonsten: futsch! Neustart
              } else [self restartLevel];
            }

          }
        }
      }
    }
```

So weit nichts Neues für Sie im Code, diese Zeilen sollten lediglich unser Spiel komplettieren.

Mit dieser Idee, aus Plattformen durch den Zusatz von Bewegung Monster zu gestalten, können wir natürlich auch bewegliche Plattformen in Form von Aufzügen, Booten oder Rolltreppen schaffen. Das ist in diesem Fall bei Bäumen als Plattformen etwas unglaubwürdig, aber hey, wir befinden uns in der Spielewelt! Da gelten eigene Gesetze, oder haben Sie sich nie gefragt, warum ausgerechnet der Frosch in »Frogger« ein Leben verliert, wenn er ins Wasser fällt?