Kapitel 8

Wanzenjagd

In diesem Kapitel:

Von der Kugel zur Wanze	156
Rotationen	160
De-Bugging	164
Zufallserscheinungen	169
Die Sache mit Pythagoras	170
Farbe und Transparenz	173
Es geht auch kleiner	175
Zusammenfassung	176

Hans-Georg Schumann: Spieleprogrammierung mit dem XNA Framework, 2009 (978-3-86645-527-6)

Auch mit Kugeln oder Bällen kann man Spiele programmieren. In diesem Kapitel soll es aber um lebendige Objekte gehen bzw. solche, die lebendig sein könnten. Damit kehren wir zurück zu den Kreaturen, mit denen wir uns schon einmal in früheren Kapiteln beschäftigt haben. Diesmal werden sie zum Opfer, denn wir betätigen uns hier als Jäger. Damit man auch etwas fürs Auge hat, kümmern wir uns um die passende Optik.

Von der Kugel zur Wanze

In dem Spiel, das wir planen, soll es darum gehen, auf dem Spielfeld herumlaufende Wanzen per Mausklick zu »plätten«. Das ist erst einmal die Grundidee. Wir werden sehen, was uns dann noch dazu einfällt.

Beginnen wir mit einer neuen Klasse, die denselben Namen trägt wie die aus dem 4. Kapitel: `Creature`. Allerdings ist sie erwachsen geworden und kann mit der vollen Unterstützung des XNA-Pakets rechnen.

HINWEIS Genau besehen hat diese neue Klasse mit der gleichnamigen alten überhaupt nichts mehr zu tun – bis darauf, dass ich mir von dieser Familie ein paar Bilder ausgeliehen habe.

Wir fangen natürlich nicht wieder bei Null an, sondern bedienen uns kräftig bei den Vereinbarungen der ganzen Kugel-Klassen (von `ABall` bis `DBall`). Allerdings werden wir einige Eigenschaften anders zusammenfassen.

Ich möchte unser neues Projekt gern *XBuggy* nennen und ihm auch gleich eine neue Klassendatei spendieren, der ich den Namen *Kreatur1.cs* gebe.

TIPP Vergessen Sie nicht, *jeder* neuen Klasse die beiden Zeilen hinzuzufügen, um eine Verbindung zum XNA Framework herzustellen und dessen Werkzeuge nutzen zu können:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

Hier ist der Hauptdarsteller des Spiels – jeweils in unversehrtem und in irreparablem Zustand:

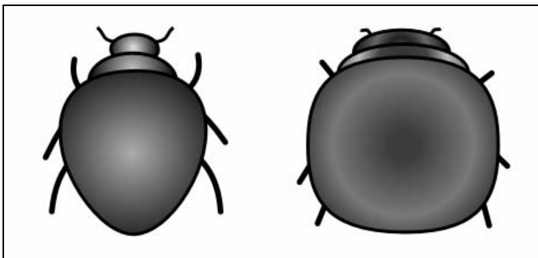

Abbildung 8.1 Unplatt und platt

Beginnen wir gleich mit dem Grundgerüst der `Creature`-Klasse samt Konstruktoren (→ *XBuggy1*, *Kreatur1.cs*):

```
public class Creature
{
  protected Rectangle Figur;
  protected Texture2D Textur;
  protected Viewport  Spielfeld;
```

```
  protected Vector2   Diff, Dpunkt;
  protected Vector4   Grenzen;
  protected float     Winkel;

  public Creature()
  {
    Figur = new Rectangle(0, 0, 50, 50);
    Diff  = new Vector2(5, 5);
  }

  public Creature(int xx, int yy, int bb, int hh)
  {
    Figur = new Rectangle(xx, yy, bb, hh);
    Diff  = new Vector2(5, 5);
  }
}
```

Am Anfang nichts Neues, alles alte Bekannte aus dem letzten Kapitel. Allerdings gibt es diese Änderungen:

```
protected Vector2 Diff, Dpunkt;
protected Vector4 Grenzen;
protected float   Winkel;
```

Ein `Vector2` umfasst die zwei Koordinaten eines Punkts. Visual C# bietet auch den Typ `Point` an, der jedoch nicht für die Xbox unterstützt wird. Statt `xDiff` und `yDiff` verwenden wir hier XNA-gerecht `Diff.X` und `Diff.Y`. Auf die neuen Eigenschaften `Dpunkt` und `Winkel` kommen wir später zu sprechen.

Die Grenzen des Spielfelds packen wir jetzt in einer Struktur vom Typ `Vector4` zusammen. Dabei soll Folgendes gelten:

xRight	xLeft	yTop	yBottom
Grenzen.W	Grenzen.X	Grenzen.Y	Grenzen.Z

In den beiden Konstruktoren bekommen dann `Figur` und `Diff` schon einmal etwas zugewiesen. Um `Grenzen` kümmert sich die Methode `SetLimits()`.

HINWEIS Auch weil unter XNA viel mit Gleitpunktzahlen statt mit ganzen Zahlen gerechnet wird, passen die `Vector`-Vereinbarungen hier besser. `Figur` bleibt unangetastet, weil hier z.B. die Übergabe als Parameter in der `Draw`-Methode umständlicher wäre. Außerdem sind die Buchstaben `W`, `X`, `Y` und `Z` weniger aussagekräftig als etwa `Width` und `Height`. (Bei `Grenzen` ist das akzeptabel, weil wir es hier mit den vier *Eck*werten zu tun haben.)

Hier soll ein Wesen wie z.B. eine Wanze auf der einen Seite des Spielfelds verschwinden, um auf der gegenüberliegenden wieder aufzutauchen. Würde das Tierchen einfach kehrtmachen oder abprallen, wäre sein weiterer Weg im Spiel zu leicht berechenbar.

Die Methode *SetLimits()* muss nun gegenüber früheren Methoden eine wichtige Erweiterung erfahren. Die Grenzen sollen jetzt nämlich auch außerhalb des Spielfelds liegen dürfen, damit die Krabbeltiere auch wirklich verschwinden.

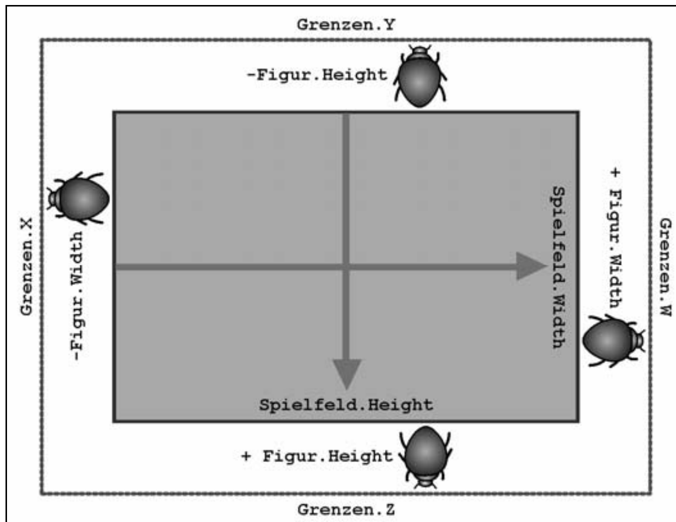

Abbildung 8.2 Grenzerweiterung

Wir wollen aber die *Abprallmethode* nicht einfach fallenlassen. Deshalb bekommt die Methode `SetLimits()` einen Parameter, der wahlweise `true` oder `false` sein kann.

```
public void SetLimits(bool Modus)
{
  if (Modus)
  {
    Grenzen.X = Figur.Width / 2;
    Grenzen.W = Spielfeld.Width - Figur.Width / 2;
    Grenzen.Y = Figur.Height / 2;
    Grenzen.Z = Spielfeld.Height - Figur.Height / 2;
  }
  else
  {
    Grenzen.X = -Figur.Width;
    Grenzen.W = Spielfeld.Width + Figur.Width;
    Grenzen.Y = -Figur.Height;
    Grenzen.Z = Spielfeld.Height + Figur.Height;
  }
}
```

Im ersten Fall (`true`) würde unser Versuchstier, wie in früheren Projekten auch schon der Ball, von den Spielfeldgrenzen abprallen. Im zweiten Fall (`false`) verschwindet die Wanze und taucht etwas später auf der anderen Seite wieder auf. (Höhe und Breite der Figur sind hier wie schon bei der Kugel gleich.)

Eine weitere Änderung betrifft die Methode `Update()`, die nun auch einen `bool`-Parameter benötigt, denn sie muss ja darauf reagieren, wie die Grenzen gerade gesetzt sind:

```
public void Update(bool Modus)
{
```

```
  if (Modus)
  {
    if (Figur.X < Grenzen.X) Diff.X = -Diff.X;
    if (Figur.X > Grenzen.W) Diff.X = -Diff.X;
    if (Figur.Y < Grenzen.Y) Diff.Y = -Diff.Y;
    if (Figur.Y > Grenzen.Z) Diff.Y = -Diff.Y;
  }
  else
  {
    if (Figur.X < Grenzen.X) Figur.X = (int)Grenzen.W;
    if (Figur.X > Grenzen.W) Figur.X = (int)Grenzen.X;
    if (Figur.Y < Grenzen.Y) Figur.Y = (int)Grenzen.Z;
    if (Figur.Y > Grenzen.Z) Figur.Y = (int)Grenzen.Y;
  }
  Figur.X += (int)Diff.X;
  Figur.Y += (int)Diff.Y;
}
```

Bei der Wanze zeigt sich ein Problem, das wir bei der Kugel nicht hatten: Wenn sie in verschiedene Richtungen läuft, sollte sie auch in verschiedene Richtungen schauen können. Brauchen wir aber da nicht vier verschiedene Bilder? Oder nicht noch viel mehr, für die schrägen Positionen? Wie wir das hinbekommen, klären wir später.

Im Hauptprogramm ändert sich gegenüber früheren Versionen nicht viel. Zunächst einmal verwenden wir hier statt einer Kugel das Objekt `Wanze1` vom Typ `Creature`:

```
Creature Wanze1;
// ...
protected override void Initialize()
{
  Wanze1 = new Creature(0, 0, 100, 100);
  base.Initialize();
}
```

Das hat auch zur Folge, dass die Datei nun *WanzeX.jpg* statt *KugelX.jpg* heißt (und im Ordner *bin\x86\Debug* oder *bin\x64\Debug* liegen muss):

```
protected override void LoadContent()
{
  spriteBatch = new SpriteBatch(GraphicsDevice);
  Wanze1.SetTexture
    (Texture2D.FromFile(GraphicsDevice, "WanzeX.jpg"));
  Wanze1.SetViewport(GraphicsDevice.Viewport);
  Wanze1.SetLimits(false);
  Wanze1.SetCenter(false);
}
```

DVD Auch die Wanzenbilder liegen auf der DVD außer in den Unterordnern *bin\x86\Debug* noch in einem Extraordner *Projekte\Bilder*.

Nicht zu vergessen, dass nun `Setlimits()` und `Update()` jeweils mit einem Parameterwert (also `false` oder `true`) aufgerufen werden. (Auf `SetCenter()` komme ich später.)

```
protected override void Update(GameTime gameTime)
{
  KeyboardState Taste = Keyboard.GetState();
  Wanze1.Update(false);
  if (Taste.IsKeyDown(Keys.Escape)) Exit();
  base.Update(gameTime);
}
```

Ich habe mir auch gleich erlaubt, in der `Draw`-Methode der Hintergrundfarbe einen neuen Anstrich zu geben und sie vom Kornblumenblau in ein Limonengrün zu ändern:

```
GraphicsDevice.Clear(Color.LimeGreen);
```

Außerdem bekommt das Spielfeld einen eigenen Titel:

```
Window.Title = "Debugging";
```

Rotationen

Wenn Sie jetzt das Programm starten, erscheint auch eine Wanze (bzw. meine grafische Version dieser Gattung), die nach rechts orientiert ist und so über das Spielfeld geschoben wird.

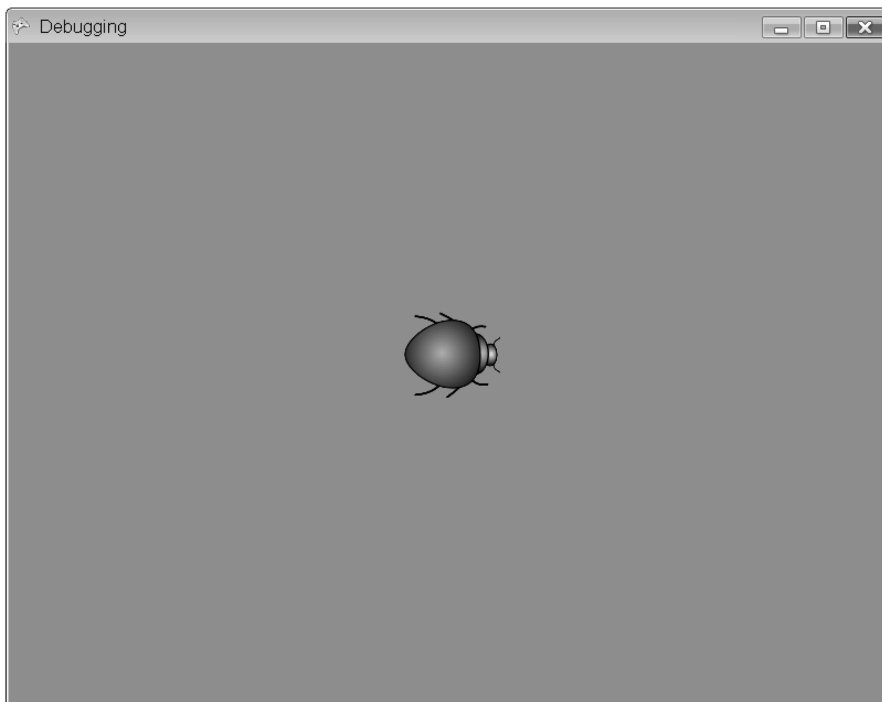

Abbildung 8.3 Nur eine Wanze

Störend dabei ist, dass die Wanze niemals ihre *Blickrichtung* ändert. Wir bräuchten also entweder einen ganzen Packen verschiedener Wanzenbilder oder eine Methode, die die Wanze in die passende Richtung dreht. Und die gibt es natürlich in XNA, allerdings nicht als Extramethode, sondern als Überladung von `Spritebatch.Draw()`. Schauen wir uns zuerst noch einmal die Version an, die wir bereits kennen:

```
SpriteBatch.Draw(Textur, Rechteck, Farbe);
```

Die drei Parameter dieser Methode kommen auch in fast allen anderen Varianten vor. Es gibt eine ganze Reihe davon, für unsere Zwecke ist diese gut geeignet:

```
SpriteBatch.Draw(Textur, Rechteck,
  Quelle, Farbe, Winkel, Drehpunkt, Effekte, Tiefe);
```

Das ist die Bedeutung der Parameter, die Sie noch nicht kennen:

`Quelle`	Falls ein Teil eines (anderen) Bilds verwendet werden soll, werden hier die Maße des Bildausschnitts angegeben. In unserem Falle null, weil wir das Rechteck verwenden, das im vorhergehenden Parameter steht.
`Winkel`	Der Winkel im Bogenmaß, um den das Sprite gedreht werden soll.
`Drehpunkt`	Der Punkt, um den das Sprite gedreht werden soll.
`Effekte`	Darstellungseffekte. Wir brauchen zurzeit keine, also None.
`Tiefe`	Es gibt zwei Ebenen, vorn (0) und hinten (1), unsere Kreatur läuft in der Ebene 0.

Womit dann unser Aufruf der `Draw`-Methode so aussieht – eingepackt in die gleichnamige Methode von `Creature`:

```
public void Draw(SpriteBatch spriteBatch)
{
  Winkel = (float)Math.Atan2(Diff.Y, Diff.X);
  Dpunkt = new Vector2(3*Figur.Width/4, 3*Figur.Height/4);
  spriteBatch.Draw (Textur, Figur,
    null, Color.White, Winkel, Dpunkt, SpriteEffects.None, 0);
}
```

Damit bei jeder Bewegung und Richtungsänderung auch die Wanze entsprechend ausgerichtet wird, muss der aktuelle Winkel berechnet werden. Und nun kommt die oben vereinbarte neue Eigenschaft `Winkel` ins Spiel.

Winkel werden meistens in Grad gemessen. Und Sie wissen wahrscheinlich, dass eine volle Drehung um die eigene Achse eine Drehung um 360° ist. Aber was ist ein *Bogenmaß*?

Das Bogenmaß leitet sich von der Zahl *Pi* (π) ab, die etwa den Wert 3,14 hat. Sie ist in C# als `Math.PI`, ein Element der Klasse `Math`, definiert. Ein Halbkreis entspricht *Pi*, ein Vollkreis 2*Pi. Für das Bogenmaß gibt es keine Maßeinheit (so wie das kleine *Kügelchen* für Grad). Die folgende Tabelle gibt einen kleinen Überblick:

Gradmaß	0°	90°	180°	270°	360°
Bogenmaß	0,0000	1,5708	3,1416	4,7124	6,2832
(mit π)	0	π/2	π	3π/2	2π
Richtung	rechts	unten	links	oben	rechts

0° und 360° bedeuten ebenso wie 0 und 2π dasselbe, auch wenn Grad- und Bogenmaß ein jeweils anderes sind: Nach einer vollen Drehung kommt man ja wieder in der Ausgangsposition an.

In der Ausgangsrichtung zeigt das Objekt nach rechts (oder Westen). Dort beginnt auch die Zählung mit 0° oder 0, nach unten (Süden) wird mit *positiven*, nach oben (Norden) mit *negativen* Werten weitergezählt:

Richtung	rechts	unten	links	oben
Bogenmaß	0,0000	1,5708	3,1416	–1,5708
(mit π)	0	π/2	π	–π/2

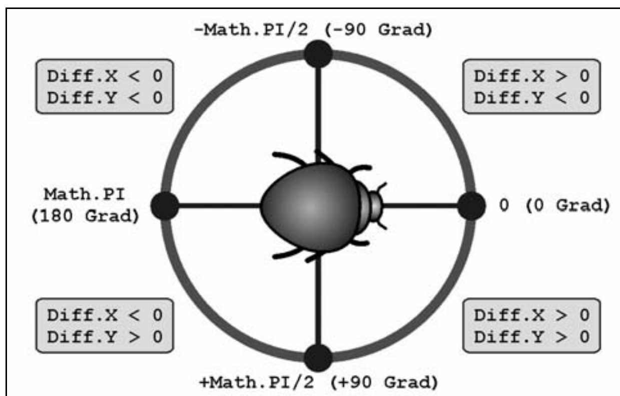

Abbildung 8.4 Immer im Kreis

HINWEIS XNA bietet mit `MathHelper` eine weitere mathematische Hilfsklasse an. Dort sind z.B. mit `Pi`, `PiOver2` und `PiOver4` (für Pi/2 und Pi/4) einige wichtige Eigenschaften definiert, die gegenüber `Math` diesen Vorteil haben: Während dort viele Eigenschaften wie z.B. `PI` als `double` vereinbart sind, wurden die genannten `Pi`-Eigenschaften von `MathHelper` als `float` definiert. Somit ist unter XNA, das `float`-Zahlen bevorzugt, ein Typecasting nicht nötig.

Wir müssen nun aus den Werten von `Diff.X` und `Diff.Y` den Winkel berechnen, der die aktuelle Bewegungsrichtung unserer Wanze anzeigt. Leider geht das nicht so einfach, indem man etwa den Quotienten der beiden Werte bildet. Aber das wäre schon mal ein Ansatz.

Denn damit haben wir die so genannte *Steigung*. Sie gibt schon über die Bewegungsrichtung Auskunft. Allerdings ist der Steigungswert leider nicht der Winkel, den die `Draw`-Methode als Parameter braucht. (Außerdem lässt sich gar nicht jeder Quotient berechnen, denn wie teilt man denn einen Wert durch 0?)

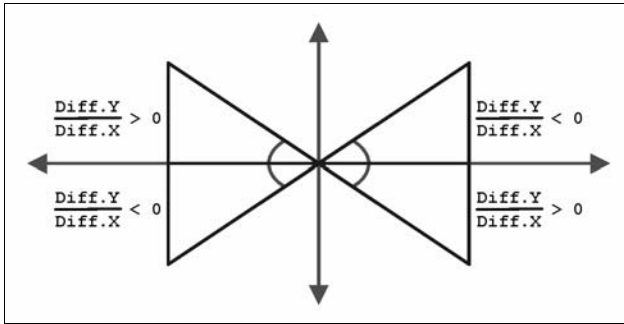

Abbildung 8.5 Tangens und Winkel

Mathematiker reden bei der Steigung auch vom *Tangens* (abgekürzt: Tan). Der hat etwas mit Winkeln zu tun. Also liegt es nahe, es damit zu versuchen. Hier würde gelten:

```
Tangens_eines_Winkels = Diff.Y / Diff.X;
```

Was wir brauchen, ist der Winkel selbst. Also müssten wir sozusagen einen *umgekehrten Tangens* einsetzen. So etwas gibt es tatsächlich, *Arcustangens* genannt (abgekürzt: Atan). Die Mathematik-Bibliothek von C# stellt uns da gleich zwei Methoden zur Verfügung, von denen diese hier am besten geeignet ist:

```
Winkel = (float)Math.Atan2(Diff.Y, Diff.X);
```

Hier werden die beiden `Diff`-Werte als Parameter übernommen, und die Methode `Atan2()` macht daraus einen Winkel im Bogenmaß. Mit dem können wir dann die Methode `Draw()` von `SpriteBatch` füttern. Zusätzlich brauchen wir noch den Punkt, um den die Figur gedreht werden soll:

```
Dpunkt = new Vector2(3*Figur.Width/4, 3*Figur.Height/4);
```

WICHTIG Nahe liegend wäre eigentlich der Mittelpunkt der Figur, aber damit klappt es bei mir nicht: Die Wanze dreht sich immer um einen Punkt, der irgendwo schräg neben dem Mittelpunkt liegt. Erst die oben stehende Korrektur brachte für mich die gewünschte Drehung.

Deshalb musste ich auch `SetCenter()`ein bisschen anpassen:

```
public void SetCenter(bool Modus)
{
  if (Modus)
  {
    Figur.X = (int)Grenzen.W / 2 + Figur.Width / 4;
    Figur.Y = (int)Grenzen.Z / 2 + Figur.Height / 4;
  }
  else
  {
    Figur.X = (int)Grenzen.W / 2 - Figur.Width / 2;
    Figur.Y = (int)Grenzen.Z / 2 - Figur.Height / 2;
  }
}
```

Der Parameter muss denselben Wert haben, den Sie bei `SetLimits()` setzen. Denn je nachdem, ob die Grenzen innerhalb oder außerhalb des Spielfelds liegen, muss in `SetCenter()` entsprechend korrigiert werden.

Und nun müsste alles passen und die Wanze dreht sich auch tatsächlich in die aktuelle Laufrichtung.

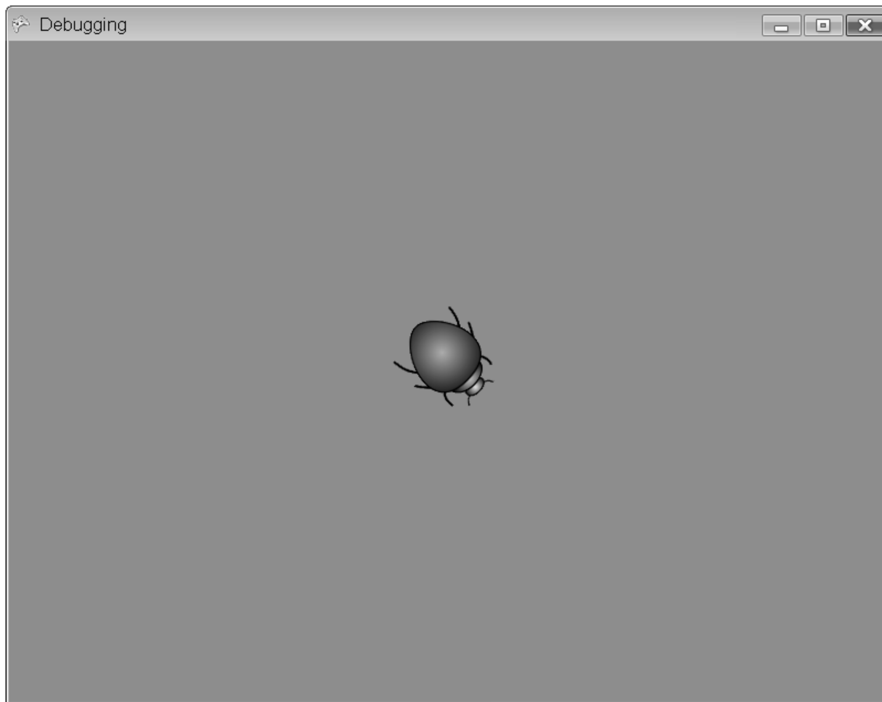

Abbildung 8.6 Richtungsänderung

TIPP Wenn Sie eine Wanze auch mal mit den Tasten steuern wollen, probieren Sie das Projekt *XBuggy1A* aus. Und in *XGame10* bekommt auch ein Ball eine sichtbare Drehrichtung.

De-Bugging

Wir haben zwar im Moment nur eine Wanze im Rennen, aber die sollte uns erst einmal als Opfer genügen. Damit erreichen wir ein Stadium, in dem es sozusagen um Leben und Tod geht. Einem Objekt der Klasse `Creature` sollten wir seine Unsterblichkeit lassen, aber wir vereinbaren eine neue Klasse namens `Bug`, die alles von `Creature` erbt und dann einige weitere Eigenschaften und Methoden bekommt. Weil der Quelltext in *Kreatur1.cs* schon ziemlich umfangreich ist, spendieren wir der neuen Klasse auch eine neue Datei.

1. Dazu klicken Sie (mal wieder) im *Projekt*-Menü auf *Klasse hinzufügen*.
2. Ändern Sie im Dialogfeld den vorgegebenen Namen z.B. in *Wanze1.cs*.
3. Dann bestätigen Sie das mit Klick auf *Hinzufügen*.
4. Erweitern Sie die `using`-Vereinbarungen um diese Zeilen:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

Und nun kümmern wir uns um das neue Kind von `Creature` (→ *XBuggy2, Wanze1.cs*):

```
public class Bug : Creature
{
  protected bool isKilled;
  public bool IsKilled
  {
    get { return isKilled; }
    set { isKilled = value; }
  }

  public Bug() : base()
  {
    IsKilled = false;
  }

  public Bug(int xx, int yy, int bb, int hh)
  : base(xx, yy, bb, hh)
  {
    IsKilled = false;
  }
}
```

Neben zwei Konstruktoren bekommt diese Klasse mit `IsKilled` eine *öffentlich* vereinbarte Eigenschaft, die zunächst auf `false` gesetzt wird, denn eine Wanze soll ja erst einmal ein bisschen herumlaufen dürfen, ehe wir sie dann per Mausklick »platt machen« und damit ins Jenseits befördern.

Property

Zweimal `IsKilled`? Ja, einmal als Variable `isKilled` (mit kleinem »i«), und einmal als eine Struktur, die uns bis jetzt noch nicht begegnet ist:

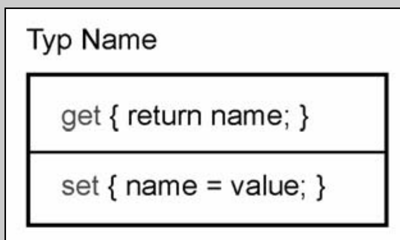

Abbildung 8.7 *get-set*-Eigenschaft (Property)

In den Anfangszeiten der Objektorientierten Programmierung (OOP) sprach man stets von Eigenschaften und Methoden und meinte mit den Eigenschaften damals vorwiegend Variablen. So hat sich das auch bei einigen Programmiersprachen gehalten.

Bei neueren Sprachen wie C# hielt eine neue Form von Eigenschaften Einzug. Diese Struktur verfügt über maximal zwei Methoden, deren Namen mit `get` und `set` festgelegt ist. Zur Unterscheidung kann man diese Art von Eigenschaft auch *Property* nennen, während man zur gewöhnlichen Form auch *Attribut* sagen kann.

Die Property `IsKilled` (ohne Klammern!) umfasst also das, was sonst dieses Methodenpaar erledigen müsste:

```
public void SetKilled(bool Tot)
{
  isKilled = Tot;
}
public bool GetKilled()
{
  return isKilled
}
```

Wir werden also künftig auch diese Struktur neben anderen Klassenelementen einsetzen.

Nun brauchen wir auch eine neue `Draw`-Methode, die die alte überschreibt:

```
public override void Draw(SpriteBatch spriteBatch)
{
  if (IsKilled)  // == true
    SetSpeed(0, 0);
  else
    Winkel = (float)Math.Atan2(Diff.Y, Diff.X);
  Dpunkt = new Vector2(Figur.Width, Figur.Height);
  spriteBatch.Draw(Textur, Figur, null, Color.White,
    Winkel, Dpunkt, SpriteEffects.None, 0);
}
```

Wichtig ist, dass außerdem die `Draw`-Methode der `Creature`-Klasse in *Kreatur1.cs* nun diesen Kopf erhält:

```
public virtual void Draw(SpriteBatch spriteBatch)
```

Wenn die Wanze erwischt wurde (`IsKilled == true`), wird ihre Geschwindigkeit auf Null herabgesetzt. Der Anzeigewinkel bleibt der der letzten Bewegung. Lebt die Wanze noch (`IsKilled == false`), dann wird der Winkel neu berechnet. Anschließend wird die Wanze im aktuellen Zustand dargestellt.

Das Aussehen der Wanze wird im Hauptprogramm bestimmt. Man kann sie einfach so unbeweglich auf dem Spielfeld liegen lassen, oder man verpasst ihr eine neue Textur, die ihren leblosen Zustand anzeigt. Oder brutaler: Die anzeigt, dass sie gerade per Mausklick geplättet wurde.

HINWEIS Zugegeben: Die Wanze könnte auch ein Käfer sein, aber nicht jeder traut sich, ohne Skrupel einen Käfer zu zerquetschen. Deshalb machen wir Jagd auf Wanzen. (Im Englischen wurde das »Gewissensproblem« einfach dadurch gelöst, dass man in beiden Fällen von *Bug* spricht.)

Schauen wir uns an, wie sich die `Update`-Methode im Hauptprogramm ändert (→ *XBuggy2, Game1.cs*):

```
protected override void Update(GameTime gameTime)
{
  KeyboardState Taste = Keyboard.GetState();
  MouseState Maus = Mouse.GetState();
  IsMouseVisible = true;
  Wanze1.Update(false);
  if (Maus.LeftButton == ButtonState.Pressed)
```

```
  {
    Rectangle Ziel = Wanze1.GetFigur();
    Point MPfeil = new Point(Maus.X + Ziel.Width/2, Maus.Y + Ziel.Height/2);
    if (Ziel.Contains(MPfeil))
    {
      Wanze1.SetTexture
        (Texture2D.FromFile(GraphicsDevice, "Wanze0.jpg"));
      Wanze1.IsKilled = true;
      Wanze1.Update(false);
    }
  }
  if (Taste.IsKeyDown(Keys.Escape)) Exit();
  base.Update(gameTime);
}
```

Zunächst wird wieder die Maus eingebunden und der Mauszeiger wird sichtbar gemacht:

```
MouseState Maus = Mouse.GetState();
IsMouseVisible = true;
```

Dann kommt das übliche Wanzen-Update:

```
Wanze1.Update(false);
```

Anschließend wird überprüft, ob die linke Maustaste gedrückt wurde:

```
if (Maus.LeftButton == ButtonState.Pressed)
```

Wenn ja, brauchen wir das Zielrechteck, in dem sich unser Opfer aufhält, und die Position des Mauspfeils:

```
Rectangle Ziel = Wanze1.GetFigur();
Point MPfeil = new Point(Maus.X + Ziel.Width/2, Maus.Y + Ziel.Height/2);
```

HINWEIS Auch die Klasse `Creature` muss natürlich ebenso wie schon die `Ball`-Klassen des letzten Kapitels über diese Methode verfügen:

```
public Rectangle GetFigur()
{
  return Figur;
}
```

Nun könnten wir überprüfen, ob sich der Mauspfeil über der Wanze (bzw. im umgebenden Rechteck) befindet:

```
if (Ziel.Contains(MPfeil))
```

In diesem Fall geht es darum, ob ein Punkt oder ein Rechteck in einem anderen komplett enthalten ist.

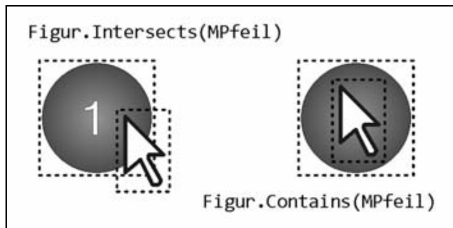

Abbildung 8.8 Überschneiden oder enthalten?

Während die Methode `Intersects` nur mit zwei Rechtecken funktioniert, kann man bei `Contains` auch einen Punkt übergeben – was wir hier ja auch tun. Für welche Methode man sich entscheidet, ist letztendlich Geschmackssache. Probieren Sie beide aus und verwenden Sie dabei auch einmal größere oder kleinere Rechtecke für den Mauszeiger.

Wurde eine Wanze mit einem Mausklick »erlegt«, so wird ein neues Bild geladen, dann wird ihr Lebenszustand auf tot gesetzt und die Anzeige wieder aufgefrischt:

```
Wanze1.SetTexture(Texture2D.FromFile(GraphicsDevice, "Wanze0.jpg"));
Wanze1.IsKilled = true;
Wanze1.Update(false);
```

Wenn Sie dieses Programm nun laufen lassen, könnte es sein, dass es nach einer Weile (oder schon nach kurzer Zeit) zu einem solchen Bild kommt:

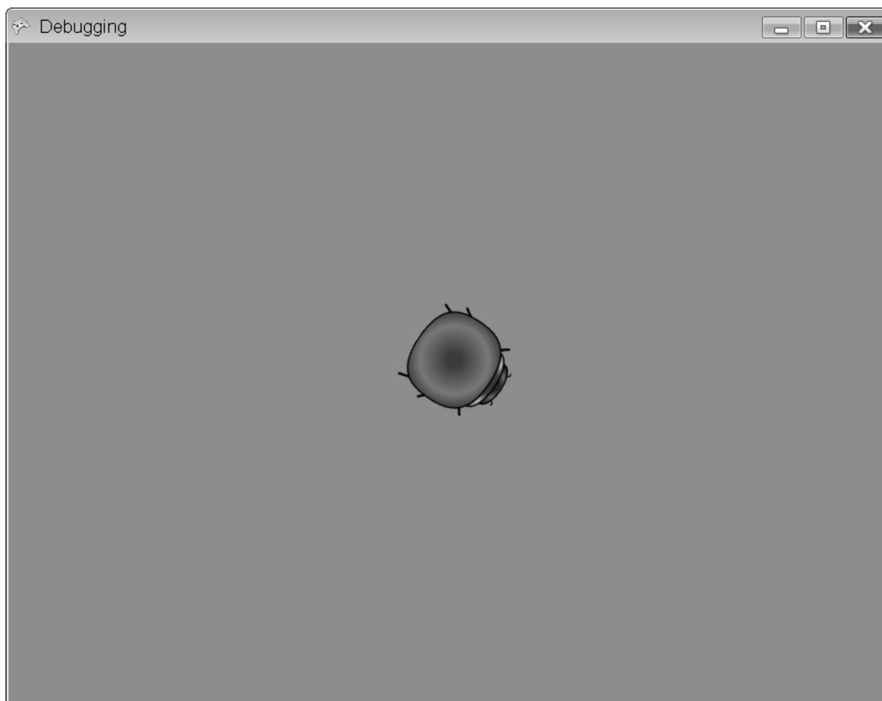

Abbildung 8.9 Wanzenplättung

Zufallserscheinungen

Nun erscheint unsere Wanze an einem Rand, läuft über das Spielfeld, und wenn Sie sie nicht erwischen oder ihr eine Gnadenfrist einräumen, verschwindet sie am anderen Ende, um dann an einer (meistens schräg) gegenüberliegenden Stelle wieder aufzutauchen.

Beobachtet man diesen Weg eine Weile, so kann man abschätzen, wo die Wanze das nächste Mal erscheint. Für unsere Wanze brauchen wir da etwas mehr Intelligenz. Sie soll also in der Lage sein, unvorhergesehen an einer anderen Stelle wieder aufzutauchen.

Dafür benötigt die Klasse `Bug` jetzt eine neue `Update`-Methode (→ *XBuggy3, Wanze1.cs*):

```
public void Update(int Typ)
{
  if ((Figur.X <= Grenzen.X) || (Figur.X >= Grenzen.W)
  || (Figur.Y <= Grenzen.Y) || (Figur.Y >= Grenzen.Z))
    SetRandPos(Typ);
  Figur.X += (int)Diff.X;
  Figur.Y += (int)Diff.Y;
}
```

Hier werden auf einen Schlag alle vier Grenzen getestet und wenn eine davon erreicht ist, wird eine Methode aufgerufen, die wir uns noch entwicklen müssen. `SetRandPos()` soll *Set Random Position* abkürzen, zu Deutsch soviel wie *Setze das Objekt auf eine Zufallsposition*. Die Kurzversion sieht so aus:

```
public void SetRandPos(int Typ)
{
  float XX = Grenzen.X + Figur.Width / 4;
  float XW = Grenzen.W - Figur.Width / 4;
  float YY = Grenzen.Y + Figur.Height / 4;
  float YZ = Grenzen.Z - Figur.Height / 4;
  do
  {
    Figur.X = Zufall.Next((int)Grenzen.X, (int)Grenzen.W);
    Figur.Y = Zufall.Next((int)Grenzen.Y, (int)Grenzen.Z);
  }
  while ((Figur.X >= XX) && (Figur.X <= XW) && (Figur.Y >= YY) && (Figur.Y <= YZ));
}
```

Zuerst werden vier *innere* Grenzwerte festgelegt. Denn die Wanze soll ja nicht mitten auf dem Spielfeld erscheinen, sondern unsichtbar außerhalb. Und dann erst soll sie vom Rand aus ins Feld laufen. Für die Zufallspositionen ist also das Spielfeld eine Art verbotene Zone. Und damit das Objekt nicht schon bei der Positionierung ins Feld hineinlugt, wurden die Grenzwerte etwas korrigiert.

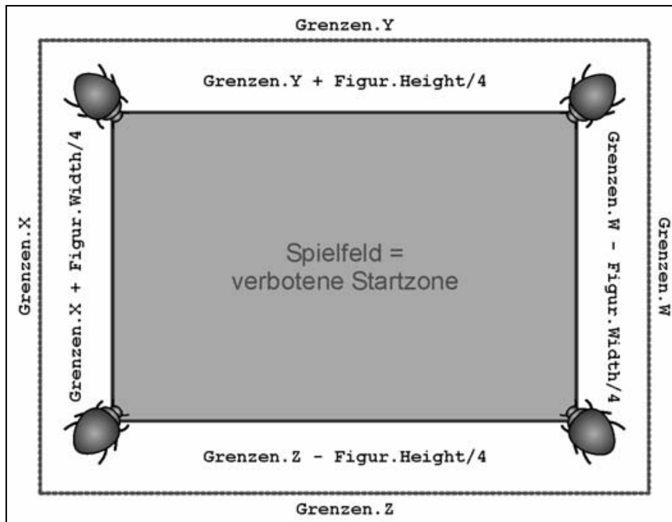

Abbildung 8.10 Startpositionen für die Wanze

In einer Schleife wird nun je ein Zufallswert für `Figur.X` und `Figur.Y` erzeugt. Dies geschieht so lange, bis beide Werte nicht mehr in der »verbotenen Zone« liegen:

```
do
{
  Figur.X = Zufall.Next((int)Grenzen.X, (int)Grenzen.W);
  Figur.Y = Zufall.Next((int)Grenzen.Y, (int)Grenzen.Z);
}
while ((Figur.X >= XX) && (Figur.X <= XW) && (Figur.Y >= YY) && (Figur.Y <= YZ));
```

Für `Zufall.Next()` sind als Parameter jeweils der niedrigste und der höchste erlaubte Wert für eine Zufallszahl angegeben. Das Typecasting ist hier nötig, um aus den Grenzwerten ganze Zahlen zu machen.

Die Sache mit Pythagoras

Wenn Sie das Spiel nun ausprobieren, kommt die Wanze aus verschiedenen, manchmal unverhofften Ecken. Allerdings behält sie stets die von Anfang an eingeschlagene Richtung bei.

Damit kommen wir zur ersten Erweiterung von `SetRandPos()`:

```
if (Typ == 1)
{
  int Drehung = Zufall.Next(3);
  if (Drehung == 0) SetSpeed(-Diff.X, Diff.Y);
  if (Drehung == 1) SetSpeed(Diff.X, -Diff.Y);
}
```

Wird beim Aufruf der Methode als Parameter eine Eins eingegeben, so wird zuerst eine zufällige Zahl erzeugt, die den Wert 0, 1 oder 2 haben kann. Diese `Drehung` bestimmt dann, ob die Richtung für den `X`-Wert oder den `Y`-Wert von `Diff` umgekehrt wird. Bei 0 verändert sich nichts.

Um diesen Effekt im Hauptprogramm zu aktivieren, ist ein entsprechender Aufruf der `Update`-Methode nötig:

```
Wanze1.Update(1);
```

Damit gibt es insgesamt vier verschiedene Richtungen, die unser Spielobjekt einschlagen kann. Sind Sie damit zufrieden? Oder wäre es Ihnen lieber, wenn die Wanze grundsätzlich *jede* Richtung einschlagen könnte, also nicht nur einfach den aktuellen X- oder Y-Wert wechseln würde? Dann schauen Sie sich die nächste Erweiterung von `SetRandPos()` an:

```
if (Typ == 2)
{
  do
  {
    Diff.X = (float)(2 * Zufall.Next(-6, +6));
    Diff.Y = (float)(2 * Zufall.Next(-4, +4));
  }
  while ((Diff.X == 0) || (Diff.Y == 0));
  SetSpeed(Diff.X, Diff.Y);
}
```

Hier werden für `Diff.X` und `Diff.Y` jeweils neue Zufallswerte erzeugt. Dies geschieht wieder in einer Schleife, um zu vermeiden, dass beide Werte gleichzeitig auf 0 gesetzt werden. Dann hätten wir ja Bewegungsstillstand.

Sie können die gegebenen Zufallsgrenzen ändern, probieren Sie eigene Werte aus. Auf jeden Fall sehen Sie gleich die Wirkung, wenn Sie den Parameterwert für die `Update`-Methode im Hauptprogramm ändern:

```
Wanze1.Update(2);
```

Nun läuft die Wanze in verschiedenste Richtungen, aber natürlich auch verschieden schnell. Denn `Diff` bestimmt sowohl die Geschwindigkeit als auch die Bewegungsrichtung. Sie könnten sich damit zufrieden geben, aber schön wäre doch eine dritte Möglichkeit, in der die gegebene Geschwindigkeit eingehalten und nur die Richtung geändert wird.

Dann ist allerdings etwas mehr Arbeit angesagt. Denn nun müssen wir die Mathematik wieder etwas mehr bemühen. Die Geschwindigkeit eines Objekts erkennt man daran, welche Strecke es in einem bestimmten Zeitraum zurücklegt.

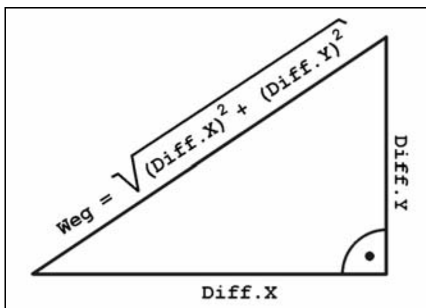

Abbildung 8.11 Pythagoras im Dreieck

Dieser Weg bildet zusammen mit `Diff.X` und `Diff.Y` ein rechtwinkliges Dreieck. Vielleicht erinnern Sie sich noch an den Namen *Pythagoras*? Genau der ist es, der uns hier weiterhelfen wird. Zuerst brauchen wir die Quadratwerte von `Diff.X` und `Diff.Y`, also `Diff.X * Diff.X` und `Diff.Y * Diff.Y`.

HINWEIS Es gibt in der Mathematikklasse von C# auch eine Methode `Math.Pow()`. `Pow` ist die Abkürzung für *Power*, was hier Potenz bedeutet. Dieser Methode wird als Erstes die Zahl übergeben, die potenziert werden soll, dann der entsprechende Exponent. Das Quadrieren der beiden Werte lässt sich demnach auch mit den Aufrufen `Math.Pow(Diff.X, 2)` bzw. `Math.Pow(Diff.Y, 2)` erledigen. Was dann frei übersetzt heißt: *Diff.X hoch 2* bzw. *Diff.Y hoch 2*.

Nach dem Addieren der beiden Quadrate brauchen wir nun daraus die Quadratwurzel. Das ist Aufgabe der Methode `Math.Sqrt()`:

```
Weg = Math.Sqrt(Diff.X * Diff.X + Diff.Y * Diff.Y);
```

Damit hätten wir dann jeweils den Weg, den die Wanze wirklich zurücklegt. Und weil beim Quadrieren einer Zahl immer ein *positiver* Wert herauskommt, werden die negativen Werte automatisch »verrechnet«, das Ergebnis für den Weg ist immer positiv (oder 0).

Nachdem dann zwei Zufallswerte für `Diff.X` und `Diff.Y` gebildet wurden, muss der eine an den anderen Wert angepasst werden, damit der zurückgelegte Weg der gleiche bleibt. Hier setzen wir wieder die Pythagoras-Formel ein, diesmal *rückwärts*:

```
Ausgleich = Math.Sqrt(Math.Abs(Weg * Weg - Diff.Y * Diff.Y));
```

Da hier ein Wert von einem anderen subtrahiert wird, kann es schon mal zu einem negativen Ergebnis kommen. Der Versuch, daraus dann die Quadratwurzel zu ziehen, führt zu Problemen. Deshalb sorgt eine weitere Methode aus dem Mathematikpaket, nämlich `Math.Abs()` dafür, dass der Absolutwert dieser Rechnung gebildet wird. Der Absolutwert einer Zahl ist immer positiv. Und damit ist der Weg für das Ziehen der Wurzel frei.

Den erhaltenen Wert können wir nun nicht einfach `Diff.X` zuweisen, nun wird es wieder wichtig, ob `Diff.X` vorher positiv oder negativ war:

```
if (Diff.X > 0) Diff.X = Ausgleich; else Diff.X = -Ausgleich;
```

Und nun der ganze Quelltext am Stück – inklusive der nötigen Typumwandlungen:

```
if (Typ == 3)
{
  float Weg = (float)(Math.Sqrt(Diff.X * Diff.X + Diff.Y * Diff.Y));
  do
  {
    Diff.X = (float)(2 * Zufall.Next(–6, +6));
    Diff.Y = (float)(2 * Zufall.Next(–4, +4));
  }
  while ((Diff.X == 0) && (Diff.Y == 0));
  float Ausgleich = (float)(Math.Sqrt(Math.Abs(Weg * Weg - Diff.Y * Diff.Y)));
  if (Diff.X > 0) Diff.X = Ausgleich; else Diff.X = -Ausgleich;
  SetSpeed(Diff.X, Diff.Y);
}
```

Wenn Sie das Programm jetzt mit dem passenden `Update`-Aufruf (also 3 als Parameter) starten, dann läuft Ihre Wanze auch mit gleich bleibender Geschwindigkeit. Mal sehen, ob Sie sie erwischen.

Farbe und Transparenz

Bleiben wir noch eine Weile bei einer einzigen Wanze. Wer kümmert sich um die Entsorgung der Wanze, wenn sie das Zeitliche gesegnet hat? Eine Möglichkeit wäre es, sie einfach verschwinden zu lassen. Das lässt sich leicht mit einer solchen Anweisung erledigen:

```
spriteBatch.Draw(Textur, Figur, Color.TransparentWhite);
```

Damit wird das betreffende Objekt sofort unsichtbar. (Das funktioniert auch mit `TransparentBlack`.)

Manchmal ist das durchaus sinnvoll, aber hier würde es bedeuten, dass wir von der platten Wanze gar nichts zu sehen bekommen. Wäre es da nicht besser, das Bild langsam auszublenden?

Das geht, und zwar ebenfalls über die `Draw`-Methode. Dazu müssen wir uns etwas näher mit dem `Color`-Parameter beschäftigen. Das, was Sie bisher z.B. als `CornFlowerBlue`, `LimeGreen`, `White` oder `Black` kennen, sind ja nur Namen, die für eine bestimmte Farbmischung stehen.

Sämtliche Farben, besser Farbtöne, setzen sich nämlich aus nur drei Grundfarben zusammen: Rot, Grün und Blau. Der Typ `Color` besteht in Wirklichkeit sogar aus vier Komponenten (kurz RGBA):

Farbe	Rotanteil	Grünanteil	Blauanteil	Alphakanal
Rot (Red)	255	0	0	255
Green (Grün)	0	255	0	255
Blau (Blue)	0	0	255	255
Gelb (Yellow)	255	255	0	255
Purpur (Magenta)	255	0	255	255
Türkis (Cyan)	0	255	255	255
Weiß (White)	255	255	255	255
Schwarz (Black)	0	0	0	255
TransparentWhite	255	255	255	0
TransparentBlack	0	0	0	0

Für alle Werte gilt: 255 gibt den Höchstanteil einer Farbe wieder, bei 0 hat diese Farbe keinen Anteil am Farbton. Der vierte Wert ermöglicht es, ein Objekt transparent darzustellen. Ist der Alphakanal 255, so wird das Objekt ganz normal in voller Farbpracht dargestellt, bei 0 ist es unsichtbar. Dazwischen zeigt es sich verschieden durchsichtig. Man kann so z.B. Glas oder Wasser simulieren. (Wie man an `TransparentWhite` und `TransparentBlack` sehen kann, sind diese beiden als komplett *durchsichtige* Farbwerte definiert.)

Man spricht hier auch vom *Alphablending*.

Ein Byte aus Nullen und Einsen

Warum reichen die Werte von 0 bis 255 und nicht z.B. von 0 bis 100, etwa als Wert für Prozent? Weil für jeden Farbanteil ebenso wie für den Alphakanal ein Speicherplatz (genannt *Byte*) reserviert ist, haben dort nur Werte von 0 bis 255 Platz. Das liegt an dem Zahlensystem, in dem ein Computer arbeitet. Er benutzt nicht unser Dezimalsystem (mit zehn verschiedenen Ziffern), sondern ein Binärsystem aus den Ziffern 0 und 1, und setzt daraus alle Zahlen zusammen, mit denen er zu tun hat.

In einem Byte sind acht Kombinationen von Nullen und Einsen möglich, macht 2 hoch 8 = 256 Möglichkeiten. Weil die Zählung bei 0 beginnt, kommen also die Dezimalwerte von 0 bis 255 (umgerechnet in Binärzahlen) infrage.

Für die möglichen Farbwerte bedeutet das dann insgesamt Rot*Grün*Blau = 256*256*256 = 16.777.216 Möglichkeiten. Mehr als unsere Augen unterscheiden können.

Wie setzen wir unsere Erkenntnisse nun in die `Draw`-Methode ein? Statt einer vorgegebenen Farbe können wir auch die vier Werte selbst übergeben, etwa so für `CornFlowerBlue` oder für `LimeGreen`:

```
spriteBatch.Draw(Textur, Figur, new Color(100, 149, 237, 255));
spriteBatch.Draw(Textur, Figur, new Color(50, 205, 50, 255));
```

Und wenn wir als *letzten* Wert z.B. 128 einsetzen, erscheinen diese Farben halbtransparent über dem Hintergrund.

Für unsere `Bug`-Klasse benötigen wir jetzt eine weitere Eigenschaft. Als Datentyp verwenden wir `byte`, dann ist später kein Typecasting nötig (→ *XBuggy4, Wanze1.cs*):

```
protected byte Alfa;
```

Die Eigenschaft setzen wir im jeweiligen Konstruktor auf den Höchstwert:

```
Alfa = 255;
```

Sobald eine Wanze tot ist, wird dieser Wert bis auf 0 heruntergezählt – bei jedem Aufruf der `Draw`-Methode:

```
if (IsKilled)
{
  SetSpeed(0, 0);
  if (Alfa > 0) Alfa--;
}
```

Anschließend wird die tote Wanze blasser und blasser, bis ihre sterblichen Überreste schließlich nicht mehr zu sehen sind:

```
spriteBatch.Draw(Textur, Figur, null,
  new Color(255, 255, 255, Alfa), Winkel, Dpunkt, SpriteEffects.None, 0);
```

Hans-Georg Schumann: Spieleprogrammierung mit dem XNA Framework, 2009 (978-3-86645-527-6)

Lebt die Wanze noch, wird `Alfa` wieder auf 255 gesetzt, denn lebendig soll sie ja sichtbar sein (das ist sogar nötig, wenn wir später im Spiel die Wanzen wieder zum Leben erwecken wollen):

```
else  // if (!IsKilled)
{
  Winkel = (float)Math.Atan2(Diff.Y, Diff.X);
  Alfa = 255;
}
```

HINWEIS Wie man eine tote Wanze aus- und eine lebendige wieder einblenden kann, zeigt das Beispiel *XBuggy4A*.

Es geht auch kleiner

Zum Schluss des Kapitels sollen Sie noch eine weitere Variante der unermüdlichen `Draw`-Methode von `SpriteBatch` kennen lernen. Dazu müssen wir allerdings als zweiten Parameter statt eines Rechtecks einen Punkt von Typ `Vector2` übergeben.

In diesem Falle soll die tote Wanze nicht ausgeblendet, sondern nur solange verkleinert werden, bis sie verschwunden ist. So etwas hatten wir schon mal, meinen Sie? Ja, aber anders; Während wir da die Figur tatsächlich haben immer kleiner werden lassen, können wir auch nur die Darstellungsgröße ändern.

Wenn wir der `Draw`-Methode einen *Skalierungsfaktor* übergeben, sorgt sie für die angepasste Darstellung.

Zuerst benötigen wir dazu eine entsprechende Variable, die wir in den Konstruktoren auf 1 setzen:

```
protected float Faktor;
// ...
Faktor = 1;
```

Eine Zahl, die größer als 1 ist, wäre ein Vergrößerungsfaktor. Hier brauchen wir deshalb eine Zahl, die unter 1 liegt. Genauer: Der aktuelle Wert 1 soll langsam bis 0 heruntergezählt werden. Hier die ganze `Draw`-Methode von `Bug` auf einen Blick (→ *XBuggy4B*):

```
public new void Draw(SpriteBatch spriteBatch)
{
  if (IsKilled)
  {
    SetSpeed(0, 0);
    if (Faktor > 0) Faktor-=0.01f;
  }
  else
    Winkel = (float)Math.Atan2(Diff.Y, Diff.X);
  Dpunkt = new Vector2(3*Figur.Width/4, 3*Figur.Height/4);
  spriteBatch.Draw(Textur, new Vector2(Figur.X, Figur.Y), null,
    Color.White, Winkel, Dpunkt, Faktor, SpriteEffects.None, 0);
}
```

Klar ist, dass der Wert, der subtrahiert werden muss, eine Gleitpunktzahl ist, und zwar eine ziemlich kleine, sonst geht es zu schnell mit dem Verschwinden:

```
if (Faktor > 0) Faktor-=0.01f;
```

HINWEIS Sie fragen sich, was das kleine »f« am Ende der Zahl zu suchen hat? Das kennzeichnet eine Gleitpunktzahl (die niemals als *Kommazahl* eingegeben werden darf) als `float`. Würde dieser Zusatz nicht da stehen, macht C# daraus automatisch den Typ `double`. Da wir hier aber vorwiegend mit `float` arbeiten, ist diese Kennzeichnung mit dem »f« nötig.

Wir hätten also ganz zu Anfang den Startwert für Faktor auch so festlegen können:

```
Faktor = 1.0f;
```

Der `Draw`-Aufruf übernimmt nun nur die *Position* von `Figur`, und hinter dem Drehpunkt taucht auch `Faktor` als Parameter auf:

```
spriteBatch.Draw(Textur, new Vector2(Figur.X, Figur.Y), null,
  Color.White, Winkel, Dpunkt, Faktor, SpriteEffects.None, 0);
```

Nun wird die platte Wanze kleiner. Dass es umgekehrt auch geht, können Sie sich wahrscheinlich denken. So würde diese Anweisungszeile die tote Wanze fast auf Spielfeldgröße aufblähen:

```
if (Faktor < 10) Faktor+=0.1f;
```

Zusammenfassung

Naja, ein echtes Spiel ist das Projekt *XBuggy* noch immer nicht, aber die eine Wanze hat Sie auch schon ganz schon auf Trab gehalten, oder? Und dass man die Wanze drehen und platt machen kann, ist ja zumindest ein optischer Fortschritt. Dies sind die Neuigkeiten des Kapitels:

- Eigenschaften lassen sich mit der `get`-`set`-Kombination auch außerhalb eines Objekts flexibel handhaben.
- Die `SpriteBatch`-Methode `Draw()` lässt sich auch dazu verwenden, Objekte gedreht darzustellen. Zur Berechnung des Winkels im Bogenmaß ist der Einsatz von `Math.PI` oder einer mathematischen Formel z.B. mit `Math.Atan()` nötig. Alternativ oder zusätzlich lassen sich auch `MathHelper.ToDegrees()` oder `MathHelper.ToRadians()` einsetzen.
- Damit sich die Geschwindigkeit nicht mit der Richtung ändert, muss mithilfe der Pythagoras-Formel der echte Weg ermittelt werden (also über Quadrieren und Wurzelziehen).
- Bei einem Kollisionstest für zwei Rechtecke können Sie überprüfen, ob das eine sich mit dem anderen überschneidet (`Intersects`) oder ob es sich darin befindet (`Contains`).
- Für die Transparenz eines Objekts oder einer Farbe ist das Alphablending zuständig. Dafür enthält der Typ `Color` neben den Farbanteilen (kurz: RGB) einen zusätzlichen vierten Wert für die Transparenz.
- Eine weitere Variante der Methode `SpriteBatch.Draw()` ermöglicht es, Bilder zu skalieren, also ihre Darstellungsgröße zu ändern.

Im nächsten Kapitel geht es weiter um Wanzen, diesmal aber bekommen Sie es mit einer ganzen Schar zu tun.

Hans-Georg Schumann: Spieleprogrammierung mit dem XNA Framework, 2009 (978-3-86645-527-6)