

KAPITEL

9

Kollisionserkennung: Asteroiden



Lernziele

Themen: Mehr zu Bewegung, Tastatursteuerung und Kollisionserkennung

Konzepte: Sammlungen (erneut), **for**-Schleife, **for-each**-Schleife (erneut), Felder (erneut)

In diesem Kapitel werden wir noch einmal auf die Schleifen zurückkommen (und eine neue Schleifenart einführen) und uns tiefer gehend mit der Kollisionserkennung beschäftigen. Wir arbeiten wieder mit Sammlungen (du wurdest ja gewarnt, dass sie wichtig werden würden!) und führen einige der bereits in vorherigen Kapiteln angesprochene Themen hier zusammen. Zu diesem Zweck kommen wir auf ein Szenario zurück, das wir bereits vom ersten Kapitel her kennen: die Asteroiden (Abbildung 9.1). Die Asteroiden-Version, die wir hier verwenden, unterscheidet sich ein wenig von der, die wir zuvor betrachtet haben. Sie verfügt über einige zusätzliche Features (wie eine Protonenwelle und einen Punktestandzähler), ist jedoch noch nicht vollständig implementiert.



Abbildung 9.1
Das neue Asteroiden-Szenario (mit Protonenwelle).

Wichtige Teile der Funktionalität fehlen noch, die zu implementieren in diesem Kapitel unsere Aufgabe sein soll.

Anhand dieses Beispiels wollen wir uns noch einmal mit Bewegung und Kollisionserkennung beschäftigen. Unser Ziel ist es, etwas mehr Routine beim Einsatz von Java-Programmierkonzepten wie Schleifen und Sammlungen zu gewinnen.

9.1 Analyse: Was ist vorhanden?

Wir sollten dieses Projekt damit beginnen, dass wir einen Blick in die vorhandene Codebasis werfen. Es handelt sich dabei um eine nur teilweise implementierte Lösung namens *asteroids-1*, die sich im Ordner *Kapitel07* der Buchszenarien befindet. (Achte darauf, dass du die Version aus dem Ordner *Kapitel07* verwendest und nicht die aus *Kapitel01*.)

Übung 9.1 Öffne das Szenario *asteroids-1* aus dem *Kapitel07*-Ordner der Buchprojekte. Experimentiere ein wenig herum, um festzustellen, was das Szenario kann bzw. nicht kann.

Übung 9.2 Erstelle eine Liste der Dinge, die diesem Projekt hinzuzufügen sind.

Übung 9.3 Mit welcher Taste auf der Tastatur kann ein Schuss abgefeuert werden?

Übung 9.4 Füge eine Explosion in ein gerade ausgeführtes Szenario ein (durch interaktive Erzeugung eines Objekts der Klasse **Explosion**). Funktioniert das? Was geschieht?

Übung 9.5 Füge eine Protonenwelle in ein Szenario ein. Funktioniert das? Was geschieht?

Beim Herumspielen mit dem aktuellen Szenario ist dir sicher aufgefallen, dass wichtige Funktionalität fehlt:

- Die Rakete bewegt sich nicht. Sie kann weder gedreht noch vorwärtsbewegt werden.
- Nichts passiert, wenn ein Asteroid mit der Rakete zusammenstößt. Er fliegt einfach durch die Rakete hindurch, anstatt sie zu zerstören.
- Folglich kannst du nicht verlieren. Das Spiel ist nie zu Ende und der endgültige Punktestand wird nie angezeigt.
- Die Klassen **ScoreBoard**, **Explosion** und **ProtonWave**, die im Klassendiagramm zu finden sind, scheinen in diesem Szenario nicht zum Einsatz zu kommen.

Was wir jedoch machen können, ist, Schüsse auf die Asteroiden abzufeuern. (Wenn du noch nicht herausgefunden hast, wie das geht, experimentiere weiter.)

Asteroiden brechen auseinander, wenn sie von einer Kugel getroffen werden, oder verschwinden ganz, wenn sie bereits sehr klein sind.

Ziel dieses Spiels ist es ganz offensichtlich, alle Asteroiden auf dem Bildschirm zu zerstören, ohne dass unsere Rakete selbst getroffen wird. Um das Ganze etwas interessanter zu gestalten, wollen wir noch eine weitere Waffe hinzufügen – die Protonenwelle. Und wir wollen den Punktestand während des Spiels speichern. Doch dazu bedarf es noch einer Menge Arbeit.

- Wir müssen die Bewegung für die Rakete implementieren. Zurzeit kann sie nur Schüsse abfeuern, aber sonst nichts. Sie soll sich aber auch vorwärtsbewegen und drehen können.
- Wir müssen sicherstellen, dass die Rakete explodiert, wenn sie von einem Asteroiden getroffen wird.
- Wenn die Rakete explodiert, soll eine Anzeigetafel mit unserem Endergebnis eingeblendet werden.
- Wir wollen eine Protonenwelle auslösen können. Die Protonenwelle sollte klein um die Rakete beginnen und sich allmählich ausdehnen, wobei sie alle davon berührten Asteroiden zerstört.

Doch bevor wir uns all diesen Funktionen zuwenden, beginnen wir mit einer kleinen kosmetischen Änderung: Wir zeichnen Sterne in unser Universum.

9.2 Sterne zeichnen

In all unseren vorherigen Szenarien haben wir ein fertiges Bild als Hintergrund für unsere Welt verwendet. Das Bild wurde in einer Bilddatei in unserem Dateisystem gespeichert.

In diesem Szenario wollen wir eine neue Technik einführen, um Hintergrundbilder zu erstellen: Wir zeichnen sie direkt bei ihrer Erzeugung.

Das Asteroiden-Szenario verwendet keine Bilddatei als Hintergrund. Eine Welt, der kein Hintergrundbild zugewiesen wurde, erhält standardmäßig ein automatisch erzeugtes, weißes Hintergrundbild.

Übung 9.6 Untersuche den Konstruktor der Klasse **Space** in deinem Szenario. Suche die Codezeilen, die den schwarzen Hintergrund erzeugen.

Wie ein Blick auf das Szenario *asteroids-1* verrät, ist der Hintergrund durchgehend schwarz. Im Konstruktor der Klasse **Space** finden wir dazu folgende drei Codezeilen:

```
GreenfootImage background = getBackground();  
background.setColor(Color.BLACK);  
background.fill();
```

Tipp!

Wenn du nur vorübergehend ein paar Codezeilen entfernen möchtest, ist es leichter, sie „auszukommentieren“ als sie zu löschen. Hierfür bietet dir der Greenfoot-Editor eine bequeme Funktion. Markiere einfach die betroffenen Zeilen und rufe vom Menü BEARBEITEN den Befehl EINKOMMENTIEREN (F8) oder AUSKOMMENTIEREN (F7) auf.

Übung 9.7 Entferne diese drei Zeilen aus deiner Klasse. Dazu musst du sie lediglich auskommentieren. Was kannst du beobachten? (Anschließend füge sie wieder ein.)

Die erste Zeile liefert das aktuelle Hintergrundbild unserer Welt zurück. Dies ist das automatisch erzeugte (weiße) Bild. Dadurch erhalten wir eine Referenz auf den Welthintergrund, der in der Variablen **background** gespeichert wird.

Das so gespeicherte Hintergrund-Objekt ist vom Typ der Klasse **GreenfootImage**, die uns bereits vorher begegnet ist.

Übung 9.8 Schau in die Dokumentation für die Klasse **GreenfootImage**. Wie lautet der Name der Methode, die ein Rechteck zeichnet? Was ist der Unterschied zwischen **drawOval** und **fillOval**?

Die zweite Zeile im obigen Codefragment setzt die Farbe auf Schwarz. Diese Zeile selbst hat noch keinen direkten Effekt (sie ändert nicht die Farbe des Bildes). Sie legt vielmehr die Farbe fest, die für alle folgenden Zeichenoperationen verwendet wird. Der Parameter ist eine Konstante der Klasse **Color**, die wir aus dem vorherigen Kapitel bereits kennen.

Übung 9.9 Schau noch einmal in die Dokumentation der Klasse **Color**. (Weißt du noch, in welchem Paket du suchen musst?) Für wie viele Farben definiert diese Klasse Konstanten?

Die dritte Zeile des Codefragments füllt unser Bild endlich mit der gewählten Farbe. Beachte, dass wir dieses Bild nicht erneut als Welthintergrund setzen müssen. Als wir dieses Bild anforderten (mit **getBackground()**), erhielten wir eine Referenz auf das Hintergrundbild, das immer noch unseren Welthintergrund bildet. Es wird nicht aus der Welt entfernt, nur weil wir jetzt eine Referenz darauf besitzen.

Wenn wir in dieses Bild zeichnen, zeichnen wir direkt in den Hintergrund der Welt.

Unsere Aufgabe soll es jetzt sein, einige Sterne in das Hintergrundbild zu zeichnen.

Übung 9.10 Erzeuge in der Klasse **Space** eine neue Methode namens **createStars**. Diese Methode soll einen Parameter namens **number** vom Typ **int** übernehmen, der die Anzahl der zu erzeugenden Sterne angibt. Ein Rückgabebetyp wird nicht benötigt. Der Rumpf der Methode sollte – jedenfalls vorläufig – leer bleiben.

Übung 9.11 Schreibe einen Kommentar für die neue Methode. (Der Kommentar sollte beschreiben, was die Methode macht und wofür der Parameter verwendet wird.)

Übung 9.12 Rufe diese neue Methode in deinem **Space**-Konstruktor auf. 300 Sterne sind für den Anfang eine ganz gute Zahl (obwohl du später durchaus andere Werte vorgeben kannst – je nachdem, was dir gefällt).

Übung 9.13 Kompiliere die Klasse **Space**. Zu diesem Zeitpunkt solltest du noch keinen Effekt beobachten können (da unsere neue Methode leer ist). Aber zumindest sollte sich die Klasse problemlos kompilieren lassen.

In der Methode **createStars** werden wir jetzt Code einfügen, der einige Sterne auf das Hintergrundbild malt. Die genaue Anzahl der Sterne wird im Parameter der Methode angegeben.

Hierfür wollen wir eine andere Schleife einsetzen: die **for**-Schleife.

Aus den vorherigen Kapiteln kennen wir bereits die **while**- und die **for-each**-Schleife. Die **for**-Schleife verwendet das gleiche Schlüsselwort wie die **for-each**-Schleife (**for**), weist aber eine andere Struktur auf:

```
for (initialisierung; schleifenbedingung; inkrement)
{
    schleifenrumpf;
}
```

Ein Beispiel für diese Schleife findest du in der Methode **addAsteroids** der Klasse **Space**.

Übung 9.14 Untersuche die Methode **addAsteroids** in der Klasse **Space**. Was macht sie?

Übung 9.15 Gehe zur **for**-Schleife in dieser Methode. Schreibe nieder, welcher Teil im Schleifenkopf die *Initialisierung*, die *Schleifenbedingung* und das *Inkrement* ist. (Siehe dazu die Definition der **for**-Schleife oben.)

Konzept

Die **for**-Schleife ist ein Java-Schleifenkonstrukt, das sich als besonders nützlich erweist, wenn die Anzahl der Iterationen fest vorgegeben ist.

Der Initialisierungsteil einer **for**-Schleife wird genau einmal ausgeführt, bevor die Schleife beginnt. Dann wird die Schleifenbedingung geprüft. Wenn sie wahr ist, wird der Schleifenrumpf ausgeführt. Am Ende, wenn der Schleifenrumpf vollständig ausgeführt wurde, wird der Inkrementteil des Schleifenkopfs ausgeführt. Anschließend beginnt die Schleife von vorn. Erneut wird die Bedingung ausgewertet und, wenn sie wahr ist, die Schleife ausgeführt. Dies geht so lange, bis die Schleifenbedingung **false** zurückliefert. Die Initialisierung wird nur einmal am Anfang und dann nicht mehr ausgeführt.

Eine **for**-Schleife lässt sich ganz einfach durch eine **while**-Schleife ersetzen. Eine **while**-Schleife, die einer **for**-Schleife entspricht, sähe folgendermaßen aus:

```
initialisierung;
while (schleifenbedingung)
{
    schleifenrumpf;
    inkrement;
}
```

Die **while**-Schleife hier und die **for**-Schleife oben machen genau das Gleiche. Der Hauptunterschied zwischen den beiden ist, dass bei der **for**-Schleife die Initialisierung und der Inkrementteil in den Schleifenkopf verschoben wurden. Dadurch befinden sich alle Elemente, die das Schleifenverhalten definieren, an einer Stelle, was das Lesen der Schleife leichter machen kann.

Eine **for**-Schleife ist vor allem dann besonders praktisch, wenn wir bereits vor Eintritt in die Schleife wissen, wie oft wir die Schleife ausführen wollen.

Die **for**-Schleife in der Methode **addAsteroids** lautet:

```
for (int i = 0; i < count; i++)
{
    int x = Greenfoot.getRandomNumber(getWidth()/2);
    int y = Greenfoot.getRandomNumber(getHeight()/2);
    addObject(new Asteroid(), x, y);
}
```

Dieser Code ist ein typisches Beispiel für eine **for**-Schleife:

- Der Initialisierungsteil deklariert und initialisiert eine Schleifenvariable. Diese Variable wird oft **i** genannt und mit **0** initialisiert.
- Die Schleifenbedingung prüft, ob unsere Schleifenvariable immer noch kleiner ist als der vorgegebene Wert (hier: **count**). Wenn ja, wird die Schleife fortgesetzt.
- Der Inkrementteil erhöht einfach die Schleifenvariable.

Die **for**-Schleife gibt es in verschiedenen Variationen, aber dieses Beispiel zeigt ein sehr typisches Format.

Übung 9.16 Wandle in deiner Klasse **Space** die **for**-Schleife in der Methode **addAsteroids** in eine **while**-Schleife um. Stelle sicher, dass sie das Gleiche macht wie vorher.

Übung 9.17 Ändere die Schleife in dieser Methode wieder zurück in eine **for**-Schleife.

Übung 9.18 Implementiere den Rumpf der Methode **createStars**, die du zuvor erzeugt hast. Diese Methode sollte folgende Aufgaben erledigen:

- Sie soll das Hintergrundbild der Welt anfordern.
- Sie soll eine **for**-Schleife ähnlich der in **addAsteroids** beinhalten. Der Grenzwert für die Schleife wird der Methode als Parameter übergeben.
- Sie soll im Rumpf der Schleife zufällige *x*- und *y*-Koordinaten erzeugen. Setze die Farbe auf Weiß und zeichne dann ein gefülltes Oval mit einer Breite und Höhe von zwei Pixel auf das Hintergrundbild.

Teste deinen Code! Werden in deiner Welt Sterne angezeigt? Wenn alles richtig war, solltest du Sterne sehen.

Übung 9.19 Erzeuge Sterne von beliebiger Helligkeit. Dazu musst du eine Zufallszahl zwischen 0 und 255 (dem zulässigen Bereich für RGB-Werte für Farben) erzeugen sowie ein **Color**-Objekt, das den gleichen Zufallswert für alle drei Farbkomponenten (rot, grün und blau) verwendet. Durch die Verwendung des gleichen Wertes für alle Farbkomponenten wird sichergestellt, dass die resultierende Farbe ein neutraler Grauton ist. Verwende diese Zufallsfarbe zum Zeichnen des Sterns. Achte darauf, dass du für jeden neuen Stern eine neue Farbe erzeugst.

Diese Aufgaben sind nicht ganz einfach, aber du solltest über das nötige Wissen verfügen, um sie zu lösen. Wenn du Probleme hast, kannst du einen Blick in unsere Lösung werfen. Du findest eine Implementierung hiervon in der Szenario-Version *asteroids-2*. Du kannst aber auch momentan diesen Abschnitt noch überspringen, mit der folgenden Aufgabe fortfahren und später hierher zurückkehren.

9.3 Drehen

Im vorangehenden Abschnitt haben wir viel Zeit und Mühe für das äußere Erscheinungsbild aufgewendet. Wir haben uns einer **for**-Schleife bedient, um Sterne im Hintergrund zu erzeugen. Das war viel Arbeit für wenig Effekt, doch unsere neu erworbenen Kenntnisse zur **for**-Schleife werden sich später noch als sehr nützlich erweisen.

Kommen wir nun zur eigentlichen Funktionalität. Wir wollen, dass sich die Rakete bewegt. Der erste Schritt soll darin bestehen, die Rakete in eine Drehbewegung zu versetzen, wenn wir die rechte oder linke Pfeiltaste auf der Tastatur drücken.

Übung 9.20 Untersuche die Klasse **Rocket**. Suche nach dem Code, der für die Tastatureingabe zuständig ist. Wie lautet der Name der Methode, die diesen Code enthält?

Übung 9.21 Füge eine Anweisung hinzu, die dafür sorgt, dass sich die Rakete nach links dreht, während die Taste  gedrückt wird. In jedem **act**-Schritt sollte sich die Rakete um 5 Grad drehen. Hierfür bieten sich die Methoden **getRotation** und **setRotation** der Klasse **Actor** an.

Übung 9.22 Füge eine Anweisung hinzu, die dafür sorgt, dass sich die Rakete nach rechts dreht, wenn die Taste  gedrückt wird. Teste deinen Code!

Wenn es dir gelungen ist, diese Übungen erfolgreich abzuschließen, sollte diese Rakete jetzt in der Lage sein zu drehen, wenn du die Pfeiltasten drückst. Da die Rakete in die Richtung feuert, in die sie schaut, kann sie in alle Richtungen feuern.

Die nächste Herausforderung besteht darin, die Rakete vorwärtszubewegen.

9.4 Vorwärtsfliegen

Unsere Klasse **Rocket** ist eine Unterklasse von **SmoothMover** – eine Klasse, die wir bereits im vorherigen Kapitel kennengelernt haben. Das bedeutet, dass sie über einen Geschwindigkeitsvektor verfügt, der die Bewegung bestimmt, sowie über eine Methode **move()**, die dafür sorgt, dass sich das Objekt gemäß dieses Vektors bewegt.

Unser erster Schritt besteht darin, diese **move()**-Methode aufzurufen.

Übung 9.23 Füge in die **act**-Methode von **Rocket** einen Aufruf der Methode **move()** ein (geerbt von **SmoothMover**). Teste deinen Code. Was kannst du beobachten?

Das Hinzufügen eines **move()**-Aufrufs in unsere **act**-Methode ist ein wichtiger erster Schritt, der jedoch an sich noch nicht viel bringt. Er ist dafür verantwortlich, dass sich die Rakete entsprechend dem Bewegungsvektor bewegt, aber da wir keine Bewegung initiiert haben, hat dieser Vektor zurzeit eine Länge von 0, sodass keine Bewegung zu verzeichnen ist.

Um dies zu ändern, wollen wir zuerst einen kleinen Wert für den automatischen Drift vorgeben, sodass die Rakete mit einer Anfangsbewegung startet. Dadurch wird das Spielen interessanter, da es die Spieler daran hindert, sich einfach für eine lange Zeit nicht zu bewegen.

Übung 9.24 Füge einen kleinen Wert für die Anfangsbewegung der Rakete ein. Erzeuge dazu einen neuen Vektor mit einer beliebigen Richtung und einer kleinen Länge (ich habe für meine Version 0.7 gewählt) und übergebe diesen Vektor dann der **SmoothMover**-Methode **addToVelocity** als Parameter, um diese Kraft auf die Rakete anzuwenden. Dies kannst du im Konstruktor der Rakete tun. (Achte darauf, einen **int**-Wert als ersten Parameter im Konstruktor von **Vector** zu verwenden, damit der korrekte Konstruktor aufgerufen wird.)

Teste deinen Code. Wenn du alles richtig gemacht hast, sollte die Rakete ganz von allein vorwärtsfliegen, wenn das Szenario beginnt. Wähle keine allzu schnelle Anfangsgeschwindigkeit. Experimentiere ein wenig herum, bis du eine passable Anfangsgeschwindigkeit gefunden hast, die langsam genug ist.

Als Nächstes wollen wir dem Spieler die Möglichkeit einräumen, die Bewegung zu steuern. Unser Plan ist, dass durch Drücken der -Taste ("up") der Raketen-Turbo gezündet wird und wir uns vorwärtsbewegen.

Für die Tastatureingabe zur Einleitung einer Drehbewegung hatte unser Code die folgende Form:

```
if (Greenfoot.isKeyDown("left"))
{
    turn(-5);
}
```

Der Code für die Vorwärtsbewegung folgt jedoch einem etwas anderem Schema. Der Grund dafür ist, dass wir bei der Rotation nur agieren müssen, wenn die Taste gedrückt wird.

Hier liegt der Unterschied zur Vorwärtsbewegung: Wenn wir zum Bewegen die -Taste drücken, wollen wir das Bild der Rakete durch das Bild der Rakete mit Turboantrieb ersetzen. Lassen wir die Taste dann wieder los, sollte wieder das normale Bild angezeigt werden. Es soll also etwas passieren, wenn die Taste gedrückt wird und wenn sie losgelassen wird. Wir benötigen also ungefähr folgendes Codeschema:

wenn die Taste "up" gedrückt wird:
ändere Bild und zeige Rakete mit Turboantrieb;
addiere Bewegung;

wenn Taste "up" losgelassen wird:
zeige wieder das normale Bild an;

Das Anzeigen der Bilder ist recht einfach. Im Szenario sind die beiden dafür benötigten Raketenbilder bereits enthalten: *rocket.png* und *rocketWithThrust.png*. Beide Bilder werden oben in der Klasse **Rocket** in Zustandsfelder geladen.

Da wir auf die beiden Fälle reagieren müssen, dass die -Taste gedrückt bzw. nicht gedrückt ist, werden wir eine eigene Funktion definieren und aufrufen, die sich dieser Funktionalität annimmt.

In **checkKeys** können wir den folgenden Methodenaufruf einfügen:

```
ignite(Greenfoot.isKeyDown("up"));
```

Diese Codezeile ruft als Erstes die Methode **isKeyDown** der **Greenfoot**-Klasse auf, die einen booleschen Wert zurückgibt. Dieser Wert wird dann als Parameter der Methode **ignite** übergeben (die wir nun noch schreiben müssen).

Dazu müssen wir uns um folgende Dinge kümmern:

- Die Methode übernimmt einen booleschen Parameter (nennen wir ihn **boosterOn**), der anzeigt, ob der Turbo an oder aus sein sollte. Dieser Parameter wird durch das Ergebnis des Aufrufs von **isKeyDown("up")** bestimmt (wenn die Taste gedrückt ist, dann hat **boosterOn** den Wert **true**, ansonsten **false**).
- Ist der Turbo eingeschaltet, setzt sie das Bild auf *rocketWithThrust.png* und fügt unter Verwendung von **addToVelocity** einen neuen Vektor hinzu. Dieser Vektor sollte seine Richtung von der aktuellen Rotation der Rakete (**getRotation()**) erhalten und eine kleine konstante Länge (sagen wir 0.3) aufweisen.
- Ist der Turbo nicht eingeschaltet, setzt sie das Bild auf *rocket.png*.

Übung 9.25 Füge in deine Methode **checkKeys** – wie oben beschrieben – den Aufruf der Methode **ignite** ein.

Übung 9.26 Lege für **ignite** eine Methode mit einem leeren Rumpf an. Diese Methode sollte einen booleschen Parameter übernehmen und vom Rückgabetyt **void** sein. Denke daran, einen Kommentar zu schreiben. Teste deinen Code! Der Code sollte sich kompilieren lassen (aber sonst noch nichts machen).

Übung 9.27 Implementiere den Rumpf der **ignite**-Methode in Anlehnung an die oben stehenden Aufzählungspunkte.

Für die Implementierung unserer **ignite**-Methode ist es okay, wenn das Bild bei jedem Aufruf der Methode gesetzt wird, auch wenn dies eigentlich nicht notwendig ist (zum Beispiel wenn der Turbo aus ist und auch vorher schon aus war, müssten wir das Bild eigentlich nicht erneut setzen, da es sich nicht geändert hat). Da jedoch das Setzen des Bildes – auch wenn es nicht notwendig ist – nur geringen Overhead verursacht, lohnt sich nicht die Mühe, das Neusetzen zu vermeiden.

Hast du all diese Übungen erfolgreich nachvollzogen, solltest du jetzt imstande sein, mit deiner Rakete durch das All zu fliegen und auf Asteroiden zu feuern. Welche Wirkung es hat, einen Bewegungsvektor zu verwenden anstatt nur die **move**-Methode des Akteurs aufzurufen, siehst du, wenn die Taste gedrückt ist: Die Rakete bleibt in Bewegung, auch wenn du nichts machst, und du musst eine Taste drücken, um die Bewegung zu *ändern*. Wir haben einen *Impuls* implementiert.

Eine Version des Projekts, welche die bisher in diesem Kapitel besprochenen Übungen implementiert, findest du in den Buchszenarien unter *asteroids-2*.

9.5 Mit Asteroiden kollidieren

Das auffälligste Manko unseres Asteroiden-Spiels auf dieser Stufe ist, dass wir direkt durch die Asteroiden hindurchfliegen können. Das nimmt dem Spiel seinen ganzen Reiz, da wir nicht verlieren können. Doch das wollen wir in diesem Abschnitt ändern.

Die Idee ist, dass unsere Rakete explodiert, wenn sie mit einem Asteroiden kollidiert. Wenn du die bisherigen Übungen dieses Kapitels gemacht hast, ist dir sicher aufgefallen, dass in unserem Projekt bereits eine voll funktionsfähige Klasse **Explosion** zur Verfügung steht. Platziere einfach eine Explosion in der Welt und du wirst einen entsprechenden Explosionseffekt auslösen.

Eine grobe Beschreibung der zu lösenden Aufgabe könnte demnach lauten:

```
if (wir mit einem Asteroiden kollidiert sind)
{
    platziere eine Explosion in der Welt;
    entferne die Rakete aus der Welt;
    zeige das Endergebnis an (Spielende);
}
```

Bevor wir uns mit den einzelnen Unteraufgaben beschäftigen, wollen wir unseren Quelltext auf die Implementierung dieser Aufgabe vorbereiten. Wir folgen dabei der gleichen Strategie wie zuvor: Da es sich um eine eigene Unteraufgabe handelt, definieren wir eine eigene Methode; so bleibt unser Code weiterhin gut strukturiert und lässt sich leicht lesen. In der Regel solltest du die Implementierung neuer Funktionalität immer auf diese Weise beginnen. Die nächste Übung ist dieser Aufgabe gewidmet.

Übung 9.28 Erzeuge eine neue Methode mit einem leeren Rumpf in der Klasse **Rocket**, die prüft, ob die Rakete mit Asteroiden kollidiert. Nenne sie **checkCollision**. Diese Methode kann **private** sein und muss weder Rückgabewert noch Parameter aufweisen.

Übung 9.29 Füge in die Methode **act** der Klasse **Rocket** einen Aufruf der Methode **checkCollision** ein. Stelle sicher, dass sich die Klasse kompilieren und ausführen lässt.

Als erste Unteraufgabe wollen wir prüfen, ob wir mit einem Asteroiden zusammengestoßen sind. In der Greenfoot-Klasse **Actor** gibt es eine Reihe von verschiedenen Methoden mit unterschiedlicher Funktionalität, die auf Kollisionen prüfen.

Wir haben eine davon – **getOneIntersectingObject** – bereits in den vorigen Kapiteln kennengelernt. Nun wollen wir uns auch die anderen noch ansehen.

Konzept

Greenfoot stellt mehrere Methoden zur **Kollisionserkennung** bereit. Du findest sie in der Klasse **Actor**.

Übung 9.30 Sieh dir die Dokumentation der **Actor**-Klasse an und schreibe alle Methoden auf, die mit Kollisionserkennung zu tun haben.

Übung 9.31 Du wirst bemerken, dass es auch Methoden gibt, die dich über andere Objekte in der Nähe informieren (selbst wenn du nicht mit ihnen kollidierst). Schreibe auch diese Methoden auf.

Anhang C bietet eine Übersicht über die verschiedenen Methoden zur Kollisionserkennung und beschreibt ihre jeweilige Funktionalität. Jetzt wäre vielleicht ein guter Moment, um einen kurzen Blick darauf zu werfen. Irgendwann einmal solltest du dich mit jeweils allen Methoden zur Kollisionserkennung vertraut machen.

Für unsere Zwecke scheint **getIntersectingObjects** ein geeigneter Kandidat. Zwei Objekte berühren sich, wenn sich ihre Bilder in einem beliebigen Pixel berühren. Das ist ziemlich genau das, was wir benötigen.

In **Kapitel 5** haben wir einen überraschenden Effekt gesehen: Wenn du ganz genau hinsiehst, kannst du erkennen, dass manchmal eine Kollision entdeckt wird, wenn sich die Objekte eigentlich gar nicht berühren. Dies wird durch transparente Pixel in den Akteur-Bildern hervorgerufen.

Bilder in Greenfoot sind immer rechteckig. Wenn wir es mit einem nicht rechteckigen Bild zu tun haben (wie unsere Rakete), liegt das daran, dass einige Pixel in dem Bild *transparent* (d.h. unsichtbar) sind, da sie keine Farbe enthalten. Soweit es unser Programm angeht, sind sie jedoch immer noch Teil des Bildes.

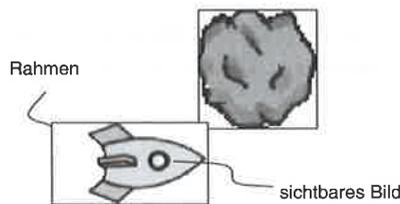
Abbildung 9.2 zeigt die Bilder für die Rakete und den Asteroiden mit ihren jeweiligen *Rahmen*. Der Rahmen bildet die Grenze des eigentlichen Bildes. (Das Bild der Rakete ist ein wenig größer als notwendig, damit es die gleiche Größe hat wie das zweite Raketen-Bild *rocketWithThrust*, das hinten im leeren Bereich die Flamme zeigt.)

Konzept

Der **Rahmen** eines Bildes ist das Rechteck, das das Bild umschließt.

Abbildung 9.2

Zwei Bilder von Akteuren und ihre jeweiligen Rahmen.



In **Abbildung 9.2** überschneiden sich die beiden Bilder, auch wenn sich ihre sichtbaren Teile nicht berühren. Die Methoden zur Kollisionserkennung werten dies als Überschneidung. Sie arbeiten mit den Rahmen und achten nicht auf die nicht transparenten Bereiche eines Bildes.

Folglich „trifft“ der Asteroid unsere Rakete, auch wenn auf dem Bildschirm zwischen den beiden noch ein kleiner Abstand vorhanden ist.

Für unser Asteroiden-Spiel wollen wir dies einmal außer Acht lassen. Zum einen ist der Abstand sehr klein, sodass er Spielern häufig gar nicht auffällt. Zum anderen ist

es nicht schwer, diesen Effekt zu begründen („Wenn du zu nahe an einen Asteroiden heranfährst, wird dein Raumschiff durch die Anziehungskraft zerstört“).

Manchmal ist es aber ganz nützlich zu prüfen, ob sich die eigentlich sichtbaren (nicht transparenten) Bereiche eines Bildes überschneiden. Dies ist möglich, jedoch ungleich schwieriger. Deshalb wollen wir hier davon absehen.

Nachdem wir uns entschieden haben, auf die Berührung von Objekten zu reagieren, können wir uns erneut den **Actor**-Methoden zuwenden. Es gibt zwei Methoden, die prüfen, ob sich Objekte berühren. Ihre Signaturen lauten:

```
List getIntersectingObjects(Class cls)
Actor getOneIntersectingObject(Class cls)
```

Beide Methoden übernehmen einen Parameter vom Typ **Class** (was bedeutet, dass wir auf Berührungen mit einer bestimmten Klasse von Objekten prüfen können, wenn wir wollen). Der Unterschied besteht darin, dass die eine Methode eine Liste aller Objekte zurückliefert, die wir gerade berühren, während die andere nur ein einzelnes Objekt zurückliefert. Falls wir mehr als ein Objekt berühren, wählt die zweite Methode eines davon nach dem Zufallsprinzip und liefert es zurück.

Für unsere Zwecke ist die zweite Methode ausreichend. Wie bei dem WBC-Szenario in **Kapitel 5**, macht es für das Spiel wirklich keinen Unterschied, ob wir mit einem Asteroiden oder mit zweien gleichzeitig kollidieren. Die Rakete wird auf alle Fälle explodieren. Die einzige Frage, die sich uns stellt, ist, ob wir überhaupt von einem Asteroiden „getroffen“ wurden.

Deshalb verwenden wir die zweite Methode. Da der Rückgabewert vom Typ **Actor** und nicht vom Typ **List** ist, lässt sich damit auch einfacher arbeiten. Der Code für unsere erste Aufgabe – überprüfen, ob wir mit einem Asteroiden kollidiert sind – ist dann derselbe wie eben:

```
Asteroid a = (Asteroid)getOneIntersectingObject(Asteroid.class);
if (a != null)
{
    ...
}
```

Übung 9.32 Füge in deine Methode **checkCollision()** Code ein, der wie oben gezeigt auf Kollision mit einem Asteroiden prüft.

Übung 9.33 Füge in den Rumpf der **if**-Anweisung Code ein, der an der Position der Rakete eine Explosion in die Welt einfügt und die Rakete aus der Welt entfernt. (Hierzu musst du die Methode **getWorld()** verwenden, um auf deren Methoden der Welt zum Hinzufügen und Entfernen von Objekten zuzugreifen.)

Für die letzte Übung können wir mit unseren eigenen Methoden **getX()** und **getY()** unsere aktuelle Position ermitteln. Diese Informationen können wir dann als Koordinaten für die Positionierung der Explosion nutzen.

Ein Lösungsversuch könnte beispielsweise folgendermaßen aussehen:

```
World world = getWorld();  
world.removeObject(this); // entfernt die Rakete aus der Welt  
world.addObject(new Explosion(), getX(), getY());
```

Dieser Code sieht auf den ersten Blick ganz vernünftig aus, funktioniert aber nicht.

Übung 9.34 Probiere den Code von oben aus. Lässt er sich kompilieren? Lässt er sich ausführen? An welcher Stelle läuft etwas schief und wie lautet die Fehlermeldung?

Dieser Code funktioniert nicht, weil wir die Methoden **getX()** und **getY()** aufrufen, *nachdem* wir die Rakete aus der Welt entfernt haben. Wenn ein Akteur aus der Welt entfernt wurde, hat er keine Koordinaten mehr – er verfügt nur über Koordinaten, solange er in der Welt existiert. Deshalb schlagen die Aufrufe der Methoden **getX()** und **getY()** in diesem Beispiel fehl.

Dieses Problem lässt sich jedoch ganz leicht beheben. Tausche einfach die beiden letzten Codezeilen: Füge zuerst die Explosion ein und entferne dann die Rakete aus der Welt.

Übung 9.35 Dies ist eine sehr fortgeschrittene Übung, die einige von euch vielleicht erst einmal überspringen sollten, um später darauf zurückzukommen:

Bei der hier verwendeten Explosion handelt es sich um eine relativ einfache Explosion. Sie soll für unsere Zwecke reichen, doch wenn du wirklich professionelle Spiele entwickeln willst, gibt es hier noch Verbesserungsmöglichkeiten. Eine anspruchsvollere Implementierung von Explosionen wird in einer Reihe von Tutorial-Videos mit dem Namen „Creating Explosions“ vorgestellt, das auf dem Greenfoot-YouTube-Kanal unter

<https://www.youtube.com/user/18km>

verfügbar ist. Erzeuge eine ähnliche Explosion für deine Rakete.

9.6 Spielende

Unser Spiel ist jetzt schon halbwegs spielbar. Vielleicht ist dir aufgefallen, dass die Punktezählung noch nicht funktioniert (dazu kommen wir später) und dass nichts passiert, wenn du verlierst. Als Nächstes werden wir am Spielende, wenn die Rakete zerstört wurde, einen großen „Spiel beendet“-Schriftzug einblenden.

Auch dies ist nicht besonders schwer: Es gibt im Projekt bereits eine Klasse **ScoreBoard**, die wir verwenden können.

Übung 9.36 Erzeuge ein Objekt der Klasse **ScoreBoard** und platziere es in der Welt.

Übung 9.37 Lies den Quelltext der Klasse **ScoreBoard**. Wie viele Konstruktoren weist sie auf? Wie unterscheiden sie sich?

Übung 9.38 Modifiziere die Klasse **ScoreBoard**: Ändere den angezeigten Text; ändere die Textfarbe; ändere die Hintergrund- und Rahmenfarbe; ändere die Schriftgröße, sodass der neue Text gut passt; passe die Breite der Anzeigetafel an deinen Text an.

Wie du siehst, zeigt die Anzeigetafel den Text „Spiel beendet“ und den Endstand an (auch wenn der Punktestand zurzeit noch nicht korrekt gezählt wird – doch darüber wollen wir uns später Gedanken machen).

Die Klasse **Space** verfügt bereits über eine Methode namens **gameOver**, die eine Anzeigetafel erzeugen und anzeigen soll.

Übung 9.39 Suche und analysiere die Methode **gameOver** in der Klasse **Space**. Was kann die derzeitige Implementierung?

Übung 9.40 Implementiere die Methode **gameOver**. Sie sollte mithilfe des Konstruktors, der einen **int**-Parameter für den Punktestand erwartet, ein neues **ScoreBoard**-Objekt erzeugen. Verwende im Moment 999 als Punktestand – wir werden dies später ändern und einen echten Punktestand verwenden. Schreibe Code, der die Anzeigetafel genau in der Mitte der Welt platziert.

Übung 9.41 Wenn du deine **gameOver**-Methode implementiert hast, teste sie. Denke daran: Du kannst die Methoden der Welt-Klasse aufrufen, indem du mit der rechten Maustaste den Hintergrund der Welt anklickst (siehe **Abbildung 8.3**).

Übung 9.42 Wie kannst du sicherstellen, dass die Anzeigetafel genau in der Mitte der Welt platziert wird, ohne dass du die Position hartcodierst (d.h. ohne die Zahlen 300 und 250 direkt als Koordinaten zu verwenden)? Hinweis: Verwende die Breite und Höhe der Welt. Implementiere dies in deinem Szenario.

Es scheint, als wenn uns der größte Teil der Arbeit bereits abgenommen wurde. Wir müssen zur Beendigung des Spiels nur noch die Methode **gameOver** aufrufen.

Den Code, der das Spiel beendet, wollen wir in der Methode **checkCollision** unserer Rakete unterbringen: Wenn wir eine Kollision feststellen, soll die Rakete explodieren (was wir bereits vorgesehen haben) und das Spiel ist zu Ende.

Übung 9.43 Füge den Aufruf der Methode **gameOver** in deine **checkCollision**-Methode ein. Kompiliere den Code. Was kannst du beobachten? Wenn du einen Fehler erhältst: wie lautet die Fehlermeldung?

Wenn du den Aufruf von **gameOver** ohne eine Typanpassung hinzugefügt hast, wirst du ein Problem bemerken. Wir haben in früheren Kapiteln gesehen, dass wir eine Typanpassung verwenden mussten, wenn wir unsere eigenen Methoden aus der Welt-Unterklasse aufrufen wollten (wie hier die **gameOver**-Methode). Die Erklärung, die wir dir dafür gegeben haben, als wir diesem Problem zum ersten Mal (in **Kapitel 5**) begegnet sind, war ein wenig oberflächlich und hat einige Details unter den Teppich gekehrt. Nun ist es an der Zeit, dieses Thema noch einmal aufzugreifen und näher zu beleuchten.

Wir wollen uns dazu als Erstes den bisherigen Code genau ansehen und annehmen, dass wir gerade einen Aufruf an die **gameOver**-Methode ohne Typanpassung eingefügt haben (Listing 9.1).

Listing 9.1:
Erster (fehlerhafter)
Versuch, die Methode
gameOver aufzu-
rufen.

```

/**
 * Prüft, ob wir mit einem Asteroiden kollidieren.
 */
private void checkCollision()
{
    Asteroid a = (Asteroid) getOneIntersectingObject(Asteroid.class);
    if (a != null)
    {
        World world = getWorld();
        world.addObject(new Explosion(), getX(), getY());
        world.removeObject(this);
        world.gameOver(); //Fehler: dies funktioniert nicht
    }
}

```

Wenn du versuchst, diesen Code zu kompilieren, erhältst du folgende Fehlermeldung:

cannot find symbol – method gameOver()

Diese Meldung versucht uns mitzuteilen, dass der Compiler keine Methode dieses Namens finden kann. Wir wissen jedoch, dass es eine solche Methode in unserer Klasse **Space** gibt. Wir wissen außerdem, dass der hier verwendete Aufruf **getWorld()** uns eine Referenz auf unser **Space**-Objekt liefert. Wo also liegt das Problem?

Das Problem ist, dass der Compiler nicht ganz so schlau ist, wie wir es gerne hätten. Die Methode **getWorld()** ist in der Klasse **Actor** definiert und ihre Signatur lautet:

World getWorld()

Wie wir sehen, wird hier behauptet, dass die Methode ein Objekt vom Typ **World** zurückliefert. Die Welt, die jedoch in unserem Fall zurückgeliefert wird, ist vom Typ **Space**.

Konzept

Objekte können mehr als einen Typ aufweisen: den Typ ihrer eigenen Klasse und den Typ ihrer übergeordneten Klasse (d.h. der Oberklasse).

Dies ist kein Widerspruch: Unser Welt-Objekt kann gleichzeitig vom Typ **World** und vom Typ **Space** sein, da **Space** eine Unterklasse von **World** ist (**Space ist eine World**, wir sprechen auch davon, dass der Typ **Space** ein *Untertyp* des Typs **World** ist).

Der Fehler rührt von den Unterschieden zwischen den beiden Typen: **gameOver** ist in der Klasse **Space** definiert, aber **getWorld** liefert uns ein Ergebnis vom Typ **World**. Der Compiler betrachtet nur den deklarierten Rückgabebetyp der von uns aufgerufenen Methode (**getWorld**). Deshalb sucht er die Methode **gameOver** auch nur im Typ **World**, wo er sie nicht finden kann. Und deshalb erhalten wir diese Fehlermeldung.

Um dieses Problem zu lösen, müssen wir dem Compiler explizit mitteilen, dass die Welt, die wir uns zurückliefern lassen, eigentlich vom Typ **Space** ist. Dies erreichen wir über eine *Typanpassung*, die wir bereits kennengelernt haben.

```
Space space = (Space) getWorld();
```

Den Typ anzupassen, ist eine Technik, um dem Compiler einen genaueren Typ für unser Objekt anzugeben, als er selbst feststellen kann. In unserem Fall kann der Compiler feststellen, dass das von **getWorld** zurückgelieferte Objekt vom Typ **World** ist, und wir teilen ihm jetzt mit, dass es eigentlich vom Typ der Klasse **Space** ist. Dazu schreiben wir den Klassennamen **(Space)** in Klammern vor den Methodenaufruf. Anschließend können wir das Ergebnis einer Variable vom Typ **Space** (anstelle einer **World**-Variable) zuweisen und dann Methoden aufrufen, die in **Space** definiert sind:

```
space.gameOver();
```

An dieser Stelle möchten wir betonen, dass die Typanpassung den Typ des Objekts nicht ändert. Unsere Welt ist eigentlich die ganze Zeit vom Typ **Space**. Das Problem ist nur, dass der Compiler dies nicht weiß. Mit der Typanpassung geben wir dem Compiler lediglich zusätzliche Informationen an die Hand.

Doch kehren wir zurück zu unserer Methode **checkCollision**. Sobald wir den Typ unserer Welt in **Space** umgewandelt haben, können wir alle Methoden darauf aufrufen: Sowohl die, die in **Space** definiert sind, als auch die vererbten, die in **World** definiert sind. Das bedeutet, dass nicht nur der Aufruf von **addObject** und **removeObject**, sondern auch der Aufruf von **gameOver** funktionieren sollte. Listing 9.2 zeigt die korrigierte Methode.

```
/**
 * Prüft, ob wir mit einem Asteroiden kollidieren.
 */
private void checkCollision()
{
    Asteroid a = (Asteroid) getOneIntersectingObject(Asteroid.class);
    if (a != null)
    {
        Space space = (Space) getWorld();
        space.addObject(new Explosion(), getX(), getY());
        space.removeObject(this);
        space.gameOver();
    }
}
```

Konzept

Typanpassung ist die Technik, einen genaueren Typ für unser Objekt anzugeben, als der Compiler selbst feststellt.

Listing 9.2

Eine korrekte Lösung, die Methode **GameOver** aufzurufen.

Übung 9.44 Implementiere den Aufruf der Methode **gameOver** und wandle den Typ des **World**-Objekts in **Space** um. Teste deinen Code. Er sollte jetzt funktionieren und die Anzeigetafel sollte eingeblendet werden, wenn die Rakete explodiert.

Übung 9.45 Was passiert bei einer falschen Typanpassung? Versuche einmal, das Welt-Objekt in sagen wir **Asteroid** anstelle von **Space** umzuwandeln. Funktioniert das? Was kannst du beobachten?

Der bisherige Aufwand hat dazu geführt, dass der Schriftzug „Spiel beendet“ eingeblendet wird, (wobei der Punktestand immer noch nicht stimmt). Um die Punktezahl kümmern wir uns erst gegen Ende dieses Kapitels. Wenn du dich damit schon jetzt befassen möchtest, kannst du zu den Übungen am Ende dieses Kapitels springen und zuerst damit fortfahren. Wir wollen hier erst einmal die Protonenwelle betrachten.

9.7 Feuerkraft hinzufügen: die Protonenwelle

Unser Spiel kann sich durchaus schon sehen lassen. Zum Schluss wollen wir in diesem Kapitel nur noch eine Sache ausführlich besprechen: die Implementierung einer Protonenwelle als zweite Waffe. Damit wollen wir das Spiel etwas abwechslungsreicher gestalten. Die Idee ist, dass sich unsere Protonenwelle, nachdem sie ausgelöst wurde, von der Rakete aus radial ausdehnt und dabei jeden Asteroiden, der davon berührt wird, zerstört. Da eine solche Welle in alle Richtungen gleichzeitig wirkt, ist sie eine viel mächtigere Waffe als unsere bisherigen Geschosse. Wir sollten deshalb im Spiel die Häufigkeit dieses Waffeneinsatzes beschränken, damit das Spiel nicht zu einfach wird.

Übung 9.46 Führe dein Szenario aus. Platziere eine Protonenwelle in deinem Szenario. Was kannst du beobachten?

Diese Übung zeigt uns, dass wir bereits über eine von **Actor** abgeleitete Klasse namens **ProtonWave** verfügen und dass deren Objekte die Welle in voller Größe anzeigen. Doch leider müssen wir feststellen, dass sich diese Welle zurzeit weder bewegt noch verschwindet noch den Asteroiden irgendeinen Schaden zufügt.

Unsere erste Aufgabe soll also sein, die Welle wachsen zu lassen. Wir wollen die Welle ganz klein beginnen und sie dann wachsen lassen, bis sie ihre bereits bekannte volle Größe annimmt.

Übung 9.47 Studiere den Quelltext der Klasse **ProtonWave**. Welche Methoden existieren bereits?

Übung 9.48 Welchen Zweck erfüllen die einzelnen Methoden? Lies die jeweiligen Kommentare und erweitere sie um detailliertere Erläuterungen.

Übung 9.49 Versuche zu erklären, was die Methode **initializeImages** macht und wie sie funktioniert. Halte deine Gedanken schriftlich fest und verwende bei Bedarf Diagramme.

9.8 Die Ausdehnung der Welle

Wir haben festgestellt, dass die Klasse **ProtonWave** über eine Methode namens **initializeImages** verfügt, die 30 Bilder der Welle in verschiedenen Größen erzeugt und sie in einem Feld speichert (Listing 9.3). Dieses Feld namens **images** hält das kleinste Bild an Index 0 und das größte an Index 29 (Abbildung 9.3). Die Bilder werden erzeugt, indem ein Basis-Bild (*wave.png*) geladen wird, von dem dann in einer Schleife Kopien erzeugt werden, die auf verschiedene Größen skaliert werden.

```

/**
 * Erzeugt die Bilder für die expandierende Welle.
 */
public static void initializeImages()
{
    if (images == null)
    {
        GreenfootImage baseImage = new GreenfootImage("wave.png");
        images = new GreenfootImage[NUMBER_IMAGES];
        int i = 0;
        while (i < NUMBER_IMAGES)
        {
            int size = (i+1) * ( baseImage.getWidth() / NUMBER_IMAGES );
            images[i] = new GreenfootImage(baseImage);
            images[i].scale(size, size);
            i++;
        }
    }
}

```

Listing 9.3

Die Bilder für die Protonenwelle initialisieren.

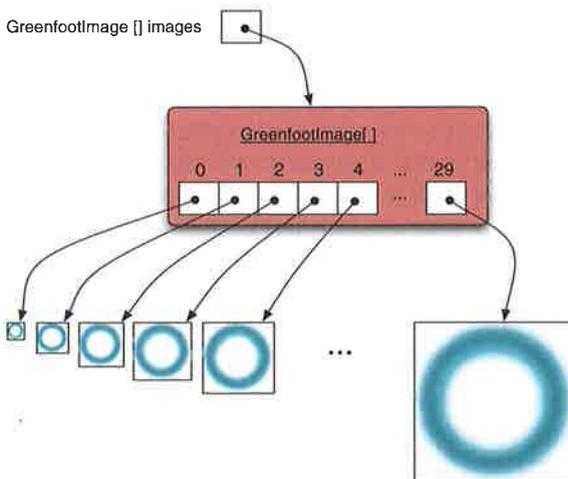


Abbildung 9.3

Ein Feld von Bildern (von denen einige aus Platzgründen nicht abgebildet sind).

Diese Methode greift für die Skalierung auf die Methode **scale** aus der Klasse **GreenfootImage** zu. Außerdem verwendet sie für die Iteration eine **while**-Schleife. Das Beispiel eignet sich jedoch auch durchaus für eine **for**-Schleife, wie wir sie am Anfang des Kapitels kennengelernt haben.

Übung 9.50 Schreibe die Methode **initializeImages** so um, dass sie anstelle der **while**-Schleife eine **for**-Schleife verwendet.

In der Praxis spielt es in diesem Fall keine große Rolle, welche Schleife wir verwenden. (Wir haben dich die Änderungen nur durchführen lassen, damit du mehr Übung im Aufsetzen von **for**-Schleifen bekommst.) Dennoch ist in diesem Fall die **for**-Schleife eine gute Wahl, da wir es mit einer bekannten Anzahl an Iterationen (die Anzahl der Bilder) zu tun haben und wir den Schleifenzähler gut zur Berechnung der Bildgrößen heranziehen können. Ein weiterer Vorteil der **for**-Schleife ist, dass sie alle Elemente der Schleife (Initialisierung, Bedingung und Inkrement) im Kopf vereint, sodass wir weniger Gefahr laufen, irgendetwas davon zu vergessen.

Das Zustandsfeld **images** sowie die Methode **initializeImages** sind statisch (beide verwenden das Schlüsselwort **static** in ihrer Definition). Wie wir bereits in **Kapitel 3** kurz angesprochen haben, bedeutet **static**, dass das Zustandsfeld **images** in der Klasse **ProtonWave** und nicht in den jeweiligen Objekten gespeichert ist. (Man nennt diese Art des Zustandsfelds auch Klassenvariable.) Folglich teilen sich alle Objekte, die wir von dieser Klasse erzeugen, diesen Satz Bilder und wir müssen nicht für jedes Objekt einen eigenen Satz erzeugen. Das ist viel effizienter, als jedes Mal einen separaten Satz Bilder zu verwenden.

Das Kopieren und Skalieren dieser Bilder dauert ziemlich lange (bis zu zwei zehntel Sekunde auf einem gängigen Computer). Dies scheint auf den ersten Blick nicht lang zu sein, aber lang genug für eine sichtbare und folglich ärgerliche Verzögerung, wenn wir diese Schritte mitten während des Spielens ausführten. Zur Lösung dieses Problems wird der Code dieser Methode in eine **if**-Anweisung gepackt:

```

if (images == null)
{
    ...
}

```

Diese **if**-Anweisung stellt sicher, dass der Hauptteil dieser Methode (der Rumpf der **if**-Anweisung) nur einmal ausgeführt wird. Beim ersten Aufruf hat **images** den Wert **null**, die Methode wird vollständig ausgeführt und das Zustandsfeld **images** wird zu einem Wert ungleich **null** initialisiert. Bei den nachfolgenden Aufrufen wird nur noch der Test der **if**-Anweisung ausgeführt und der Rumpf übersprungen. Eigentlich wird die Methode **initializeImages** jedes Mal (vom Konstruktor) aufgerufen, wenn eine Protonenwelle erzeugt wird, doch nur beim ersten Mal wird die eigentliche Arbeit geleistet.¹

Nachdem wir inzwischen eine ungefähre Vorstellung von dem Code und den bereits existierenden Zustandsfeldern haben, können wir uns an die Ausarbeitung des Codes machen.

¹ Genau genommen wird die Methode das erste Mal aus dem **Space**-Konstruktor heraus aufgerufen, sodass sie ausgeführt wird, bevor die erste Protonenwelle überhaupt erzeugt ist. Dies verhindert bereits bei der Auslösung der ersten Protonenwelle eine Verzögerung. Der Aufruf wurde nur aus zusätzlichen Sicherheitsgründen noch einmal in den Konstruktor der Protonenwelle aufgenommen: So ist sichergestellt, dass die Klasse auch dann funktioniert, wenn sie irgendwann in einem anderen Projekt verwendet wird, ohne dass die Methode **initializeImages** im Voraus aufgerufen wurde.

Konkret wollen wir Folgendes machen:

- Wir wollen die Welle mit dem kleinsten Bild beginnen lassen.
- Bei jedem **act**-Schritt soll die Welle größer werden (d.h., das nächstgrößere Bild wird angezeigt).
- Nachdem wir das größte Bild angezeigt haben, soll die Welle verschwinden (aus der Welt entfernt werden).

Dies erreichst du durch die folgenden Übungen.

Übung 9.51 Setze im Konstruktor der Klasse **ProtonWave** das Bild auf das kleinste Bild. (Du kannst **images[0]** als Parameter für die Methode **setImage** verwenden.)

Übung 9.52 Erzeuge ein Zustandsfeld namens **currentImage** vom Typ **int** und initialisiere es mit 0. Wir werden dieses Zustandsfeld dazu verwenden, um die Bilder zu durchlaufen. Der aktuelle Wert ist der Index des gerade angezeigten Bildes.

Übung 9.53 Erzeuge eine neue **private**-Methode namens **grow** mit einem leeren Rumpf. Diese Methode nimmt keine Parameter entgegen und liefert keinen Wert zurück.

Übung 9.54 Rufe die **grow**-Methode aus deiner **act**-Methode heraus auf (auch wenn sie zurzeit noch nichts macht).

Wir haben es fast geschafft. Das Einzige, was noch zu tun bleibt, ist die **grow**-Methode zu implementieren. Die Idee dahinter ist mehr oder weniger folgende:

**zeige das Bild mit der Indexposition *currentImage* an;
inkrementiere *currentImage*;**

Außerdem müssen wir eine **if**-Anweisung einfügen, die vorab prüft, ob **currentImage** das letzte Bild erreicht hat. In diesem Fall entfernen wir die Protonenwelle aus der Welt und sind fertig.

Übung 9.55 Implementiere die Methode **grow** wie vorstehend besprochen.

Übung 9.56 Teste deine Protonenwelle. Wenn du interaktiv eine Protonenwelle erzeugst und während der Ausführung des Szenarios in der Welt platzierst, solltest du den Effekt der Wellenausbreitung sehen.

Übung 9.57 Füge einen Sound hinzu. Du findest eine Sounddatei namens *proton.wav* bei dem Szenario. Du brauchst sie nur abzuspielen. Du kannst die Anweisung zum Abspielen des Sounds im Destruktor der Protonenwelle unterbringen.

Nachdem wir jetzt über eine funktionierende Protonenwelle verfügen, sollten wir unsere Rakete auch zum Auslösen der Protonenwelle befähigen.

Übung 9.58 Erzeuge in der Klasse **Rocket** eine leere Methode namens **startProtonWave** ohne Parameter. Muss diese Funktion etwas zurückliefern?

Übung 9.59 Implementiere diese Methode. Sie sollte ein neues Protonenwellenobjekt an den aktuellen Koordinaten der Rakete in der Welt platzieren.

Übung 9.60 Rufe diese Methode von der Methode **checkKeys** aus auf, wenn die Taste "z" gedrückt wird. Teste deinen Code.

Übung 9.61 Du wirst schnell merken, dass jetzt die Protonenwelle viel zu oft ausgelöst werden kann. Zum Abfeuern der Kugeln wurde in die Klasse **Rocket** (mithilfe der Konstanten **gunReloadTime** und dem Zustandsfeld **reloadDelayCount**) eine Verzögerung eingebaut. Gehe diesen Code sorgfältig durch und implementiere etwas Ähnliches für die Protonenwelle. Gib verschiedene Werte für die Verzögerung ein, bis du einen findest, der dir passabel erscheint. Protonenwellen sollten nicht sehr häufig zur Verfügung stehen.

9.9 Mit Objekten interagieren, die im Wirkungsbereich liegen

Wir verfügen nun über eine Protonenwelle, die wir auf Knopfdruck auslösen können. Das Problem ist jedoch, dass die Protonenwelle keine Auswirkung auf die Asteroiden hat.

Wir wollen also Code hinzufügen, der die Asteroiden zerstört, wenn sie von einer Protonenwelle getroffen werden.

Übung 9.62 Füge als vorbereitende Maßnahme für diese neue Funktionalität in die Klasse **ProtonWave** eine leere Methode namens **checkCollision** ein. Diese Methode übernimmt keine Parameter und liefert auch keinen Wert zurück. Rufe diese Methode von deiner **act**-Methode aus auf.

Übung 9.63 Der Zweck dieser neuen Methode ist es zu prüfen, ob die Welle einen Asteroiden berührt, und, wenn ja, diesen Asteroiden zu zerschlagen. Schreibe den Kommentar zu dieser Methode.

Dieses Mal wollen wir nicht die Methode **getIntersectingObjects** verwenden, da die unsichtbaren Bildbereiche an den Ecken der Protonenwelle (die Teil des Umgrenzungsrechtecks, aber nicht Teil des Kreises sind) ziemlich groß sind und Asteroiden zerstört würden, lange bevor die Welle sie zu berühren scheint.

Deshalb wollen wir in diesem Falle eine andere Kollisionserkennungsmethode verwenden: **getObjectsInRange**.

Die Methode **getObjectsInRange** liefert eine Liste aller Objekte zurück, die innerhalb eines gegebenen Radius des aufrufenden Objekts liegen (Abbildung 9.4). Ihre Signatur lautet

List getObjectsInRange(int radius, Class cls)

Beim Methodenaufruf können wir (wie zuvor) die Klasse der Objekte angeben, an denen wir interessiert sind. Außerdem können wir einen Radius (in Zellen) angeben. Diese Methode liefert daraufhin eine Liste aller Objekte der angeforderten Klasse zurück, die innerhalb des angegebenen Radius um das aufrufende Objekt liegen.

Um festzustellen, welche Objekte sich in dem Wirkradius befinden, werden die Mittelpunkte der Objekte verwendet. So befindet sich zum Beispiel ein Asteroid im Radius 20 einer Rakete, wenn die Entfernung seines Mittelpunkts zum Mittelpunkt der Rakete kleiner als 20 Zellen ist. Für diese Methode spielt die Größe des Bildes keine Rolle.

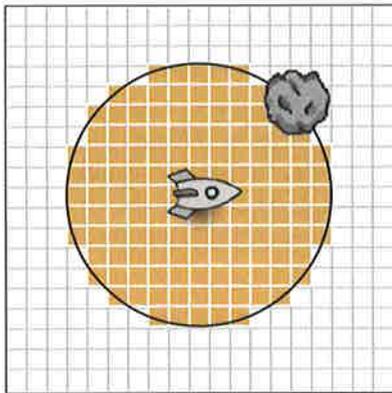


Abbildung 9.4
Der Bereich um einen Akteur mit einem gegebenen Radius.

Unter Verwendung dieser Methode können wir nun unsere Methode **checkCollision** implementieren.

Unsere Protonenwelle verfügt über Bilder unterschiedlicher Größe. Bei jedem **act**-Durchlauf können wir die Größe des aktuellen Bildes verwenden, um den Bereich für unsere Kollisionsprüfung festzulegen. Die Größe unseres aktuellen Bildes können wir mit folgendem Methodenaufruf herausfinden:

getImage().getWidth()

Wir können anschließend die Hälfte dieser Größe als unseren Bereich verwenden (da der Bereich als Radius und nicht als Durchmesser angegeben wird).

Übung 9.64 Dekлариere in **checkCollision** eine lokale Variable namens **range** und weise ihr den halben Wert der Bildbreite zu.

Übung 9.65 Füge einen Aufruf der Methode **getObjectsInRange** hinzu, die alle Asteroiden in dem berechneten Bereich zurückliefert. Weise das Ergebnis einer Variablen vom Typ **List<Asteroid>** zu. Denke auch daran, eine **import**-Anweisung für den **List**-Typ mit aufzunehmen.

Diese Übungen liefern uns eine Liste aller Asteroiden in dem Bereich der Protonenwelle zurück. Jetzt müssen wir diese Asteroiden nur noch alle zerstören.

Dafür gibt es in der Klasse **Asteroid** eine Methode namens **hit**, die bereits verwendet wird, um Asteroiden zu zerschlagen, die von einer Kugel getroffen wurden. Diese Methode lässt sich auch hier einsetzen.

Wir können eine **for-each**-Schleife verwenden, um die Liste aller Asteroiden zu durchlaufen, die uns die Methode **getObjectsInRange** zurückgeliefert hat. (Wenn du noch unsicher im Aufsetzen von **for-each**-Schleifen bist, blättere noch einmal zu **Kapitel 7.10** zurück.)

Übung 9.66 Suche die Methode **hit** in der Klasse **Asteroid**. Wie lauten ihre Parameter? Was liefert sie zurück?

Übung 9.67 In der Klasse **ProtonWave** wird ganz oben eine Konstante namens **DAMAGE** definiert, die festlegt, wie viel Schaden angerichtet werden soll. Suche die Deklaration dieser Konstanten. Wie lautet ihr Wert?

Übung 9.68 Schreibe in der Methode **checkCollision** eine **for-each**-Schleife, die über die von **getObjectsInRange** zurückgelieferte Asteroidenliste iteriert. Rufe im Schleifenrumpf für jeden Asteroiden **hit** auf, wobei du die Konstante **DAMAGE** verwendest, die die Höhe des verursachten Schadens angibt.

Wenn du all diese Übungen implementiert hast, teste deinen Code. Hast du alles richtig gemacht, solltest du über eine spielbare Version des Asteroiden-Zerstörers verfügen. Jetzt kannst du Asteroiden beschießen und mit den Protonenwellen viele Asteroiden sofort zerstören. Du wirst feststellen, dass es sinnvoller ist, die Zeit zum Neuladen der Protonenwelle ziemlich lang zu wählen, da das Spiel zu leicht wird, wenn du diese Waffe zu oft einsetzt.

Diese Version des Spiels einschließlich aller Änderungen, die wir in den letzten Abschnitten vorgenommen haben, findest du unter den Buchprojekten als *asteroids-3*. Du kannst dich dieser Version bedienen, um sie zum Vergleich mit deiner Version heranzuziehen oder um nach Lösungen zu suchen, wenn du bei einer der Übungen hängen geblieben bist.

9.10 Verbesserungsmöglichkeiten

Damit sind wir bereits am Ende unserer ausführlichen Diskussion eines peu à peu entwickelten Szenarios angekommen. Es gibt jedoch noch eine ganz Reihe von möglichen Verbesserungen an diesem Spiel. Einige liegen auf der Hand, andere magst du dir vielleicht selbst ausdenken.

Im Folgenden findest du in Form von Übungen einige Vorschläge, das Spiel weiter auszubauen. Viele davon sind unabhängig voneinander, d.h., sie müssen nicht in der angegebenen Reihenfolge erledigt werden. Wähle zuerst die aus, die dich am meisten interessieren, und denke dir möglichst auch selbst einige Erweiterungen aus.

Die Klasse Counter und andere Hilfsklassen

Die Klasse **Counter** aus diesem Szenario ist nur eine Klasse aus einer kleinen Sammlung von nützlichen Hilfsklassen, die Greenfoot anbietet. Du kannst jede dieser Klassen importieren, indem du die Funktion IMPORTIERE KLASSE aus dem BEARBEITEN-Menü im Hauptfenster von Greenfoot auswählst. Versuche es und finde heraus, welche anderen Klassen es hier gibt.

Wir könnten auch den Punktestand einfach auf den Hintergrund der Welt schreiben, wie wir es in **Kapitel 5** gemacht haben. Hier haben wir uns jedoch entschlossen, aus kosmetischen Gründen die **Counter**-Klasse zu verwenden: Wir wollten es ein wenig interessanter aussehen lassen. Wenn du die Punktezählung in diesem Szenario implementiert hast, wirst du feststellen, dass dieser Zähler nicht nur besser aussieht, sondern außerdem einen Animationseffekt enthält, wenn der Zählerstand erhöht wird.

Übung 9.69 Integriere einen Punktestandzähler. Du hast festgestellt, dass es bereits einen Punktestandzähler gibt, der jedoch noch nicht verwendet wird. Der Zähler wird durch die Klasse **Counter** definiert und in der Klasse **Space** wird ein Zähler-Objekt erzeugt.

Damit die Punktezählung funktioniert, musst du ungefähr folgendermaßen vorgehen: Füge in **Space** eine Methode namens, sagen wir, **countScore** hinzu, deren Aufgabe sein sollte, neue Punkte zu dem aktuellen Punktestand zu addieren. Rufe diese neue Methode immer dann von der Klasse **Asteroid** aus auf, wenn ein Asteroid getroffen wird (vielleicht möchtest du je nachdem, ob ein Asteroid zerschlagen oder gänzlich aus der Welt entfernt wird, unterschiedliche Punkte vergeben).

Übung 9.70 Füge neue Asteroiden ein, wenn alle entfernt wurden. Vielleicht soll das Spiel mit nur zwei Asteroiden beginnen und jedes Mal, wenn diese entfernt wurden, sollen neue erscheinen, wobei jedes Mal einer mehr erscheint. Das bedeutet, dass in der zweiten Runde drei Asteroiden zu beseitigen wären, in der dritten vier usw.

Übung 9.71 Füge einen Level-Zähler hinzu. Jedes Mal, wenn du alle Asteroiden zerstört hast, hast du ein höheres Level erreicht. Du kannst das auch damit verknüpfen, dass du mehr Punkte auf den höheren Levels vergibst.

Übung 9.72 Füge einen Sound für das Erreichen des Level-Endes hinzu. Dieser sollte jedes Mal abgespielt werden, wenn ein Level beendet wurde.

Übung 9.73 Füge einen Anzeiger für den Ladezustand der Protonenwelle hinzu, der dem Spieler anzeigt, wann die Protonenwelle wieder eingesetzt werden kann. Dies kann ein Zähler, aber auch eine grafische Repräsentation sein.

Übung 9.74 Füge einen Schutzschild hinzu. Wenn ein Schutzschild eingerichtet wird, sollte es für eine vorgegebene kurze Zeit aufrechtbleiben und die Rakete vor kollidierenden Asteroiden schützen. Der eingerichtete Schutzschild sollte auf dem Bildschirm angezeigt werden.

Es gibt natürlich noch zahllose Möglichkeiten, das Spiel zu erweitern. Denk dir ruhig selbst etwas aus, implementiere es und stelle deine Ergebnisse auf die Greenfoot-Website.



Zusammenfassung der Programmier Techniken

In diesem Kapitel haben wir uns mit der Vervollständigung des Asteroiden-Spiels befasst, das bereits teilweise in Codeform vorlag. Dabei sind uns einige wichtige Konstrukte wieder begegnet, einschließlich Schleifen, Listen und Kollisionserkennung.

Wir haben einen neuen Schleifentyp kennengelernt: die **for**-Schleife. Diese Schleife haben wir dazu verwendet, um die Sterne zu zeichnen und die Bilder der Protonenwelle zu erzeugen. Im Rahmen der Implementierung der Protonenwellenfunktionalität sind wir noch einmal auf die **for-each**-Schleife zu sprechen gekommen.

Es wurden zwei verschiedene Kollisionserkennungsmethoden verwendet: **getOneIntersectingObject** und **getObjectsInRange**. Beide haben – je nach Situation – ihre Vorteile. Die letztgenannte liefert eine Liste von Akteuren zurück, sodass wir uns noch einmal mit Listen beschäftigen mussten.

Je mehr Praxis du mit Listen und Schleifen bekommst, umso leichter wird es, sie einzusetzen. Nachdem du Listen und Schleifen erstmal eine Weile verwendet hast, wirst du dich fragen, warum du diese Konzepte am Anfang so kompliziert fandest.



Zusammenfassung der Konzepte

- Die **for-Schleife** ist ein Java-Schleifenkonstrukt, das sich besonders gut eignet, wenn die Anzahl der Iterationen fest vorgegeben ist.
- Greenfoot stellt mehrere Methoden zur **Kollisionserkennung** bereit. Du findest sie in der Klasse **Actor**.
- Der **Rahmen** eines Bildes ist das Rechteck, das das Bild umschließt.
- Objekte können **mehr als einen Typ aufweisen**: den Typ ihrer eigenen Klasse und den Typ ihrer übergeordneten Klasse (d.h. der Oberklasse).
- **Typanpassung** ist die Technik, einen genaueren Typ für unser Objekt anzugeben, als der Compiler selbst feststellt.

Vertiefende Aufgaben

Dieses Mal wollen wir die **for**-Schleifen weiter einüben und noch ein bisschen mehr mit Strings machen. Dazu verwenden wir ein anderes Szenario, *loop-practice* (du findest es im Ordner von **Kapitel 9**).

Öffne dieses Szenario sowie den Editor für die Klasse **ChalkBoard** (Kreidetafel). Du wirst eine Methode namens **practice** vorfinden. In den folgenden Übungen schreibst du den gesamten Code in diese Methode.

Die Methode besitzt anfangs nur einen einzigen Methodenaufruf, mit dem die Zahl 7 ausgegeben wird. Dies dient dazu, dir zu zeigen, wie etwas auf die Tafel geschrieben wird. Es ist leicht – du rufst einfach die Methode **write** mit einem **int**-, einem **char**- oder einem **String**-Parameter auf.

In allen Übungen wird erwartet, dass du die **write**-Methode aufrufst, um Zahlen oder Text auszugeben.

Übung 9.75 Lösche in der Methode **practice** den vorhandenen Methodenaufruf. Schreibe eine **for**-Schleife, die 10-mal den Buchstaben „a“ ausgibt.

Übung 9.76 Schreibe eine **for**-Schleife, die die Zahlen 0 bis 19 ausgibt.

Übung 9.77 Schreibe eine **for**-Schleife, die die Textschnipsel „a0“, „a1“, „a2“ usw. bis „16“ (ohne die Anführungszeichen) ausgibt.

Übung 9.78 Schreibe eine **for**-Schleife, die alle geraden Zahlen von 0 bis 24 ausgibt.

Übung 9.79 Schreibe eine **for**-Schleife, die alle Vielfachen von 5 von 15 bis 75 ausgibt.

Übung 9.80 Schreibe eine **for**-Schleife, die das Quadrat aller Zahlen von 1 bis 12 ausgibt.

Übung 9.81 Dekлариere eine lokale **String**-Variable und weise ihr deinen Namen zu. Gib diese Variable aus.

Übung 9.82 Anstatt die Variable als Ganzes auszugeben, schreibe eine **for**-Schleife, die jedes Zeichen dieses Strings einzeln ausgibt. Ziehe die Dokumentation der Klasse **String** zurate, um die Länge eines String herauszufinden und wie auf jedes Zeichen einzeln zugegriffen werden kann.

Übung 9.83 Schreibe eine Schleife, die jeden Buchstaben deines Namens zweimal ausgibt. Falls dein Name also zum Beispiel „Mira“ ist, dann sollte M M i i r r a geschrieben werden.

Übung 9.84 Schreibe eine Schleife, die jeden zweiten Buchstaben deines Namens ausgibt.

Übung 9.85 Schreibe eine Schleife, die alle Buchstaben deines Namens rückwärts ausgibt.

Übung 9.86 Weise deiner **String**-Variable das Wort „Greenfoot“ anstelle deines Namens zu. Schreibe eine Schleife, die jeden Buchstaben dieses Strings außer „r“ und „o“ ausgibt. Wenn die Schleife auf diese Buchstaben trifft, sollen sie einfach ignoriert werden.

Übung 9.87 Schreibe eine Schleife, die alle Buchstaben dieses Strings ausgibt, aber jedes „o“ durch ein „e“ und jedes „e“ durch ein „o“ ersetzt.

Die nächsten drei Übungen sind für die Neugierigen. Du musst hier selbst ein wenig recherchieren. Du benötigst Wissen über den Modulo-Operator (starte eine Internetsuche, wenn du niemanden direkt fragen kannst). Dieser Operator wird in Java mit dem Prozentzeichen (%) geschrieben. Du wirst diesen Modulo-Operator einsetzen müssen, um zu prüfen, ob eine Zahl ein Vielfaches einer anderen Zahl ist.

Übung 9.88 Schreibe eine Schleife, die die Zahlen von 0 bis 30 ausgibt, außer sie sind Vielfache von 5. In diesem Fall wird der Buchstabe „X“ anstelle der Zahl ausgegeben.

Übung 9.89 Verändere die Schleife aus der vorherigen Übung so ab, dass sie auch die Vielfachen von 3 ersetzt, und zwar durch den Buchstaben „O“.

Übung 9.90 Füge ein weiteres Detail zur Schleife hinzu: Falls die Zahl ein Vielfaches von 3 und von 5 ist, ersetze sie durch den Buchstaben „Z“.