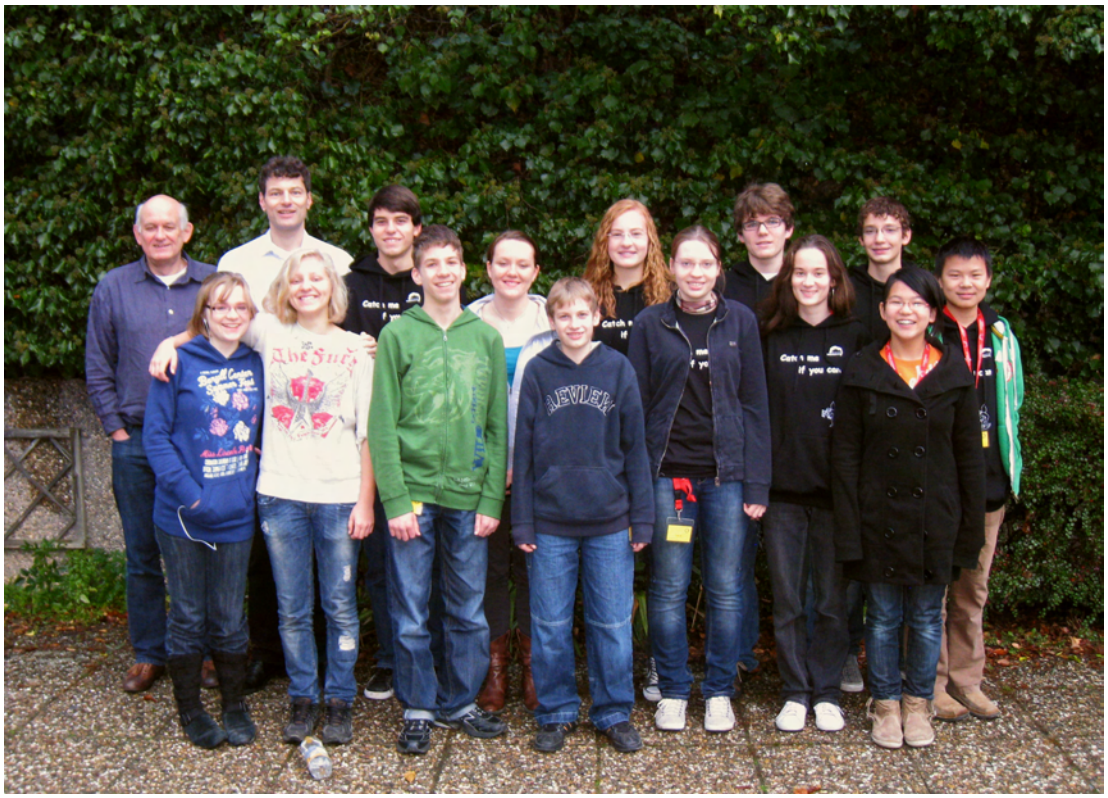
Kurs 4 – Pinball, Breakout & Friends



Wir ...

Samuel Samu kümmerte sich ständig um alles mögliche, er selbst war immer motiviert. Am Eröffnungswochenende jammerte er von Anfang an, er habe keine Ahnung vom Programmieren, doch im Sommer zeigte er uns wo der Programmierhammer hängt. Mit seinen Computerfähigkeiten wuchs auch seine Aufmüpfigkeit ;-) und er wurde zu einem vorantreibendem und lebhaftem Geist.

Wenke stellte sich den großen Herausforderungen mit Bravour und half überall mit. Jederzeit konnten wir auf ihre Zuverlässigkeit und Gewissenhaftigkeit zählen, mit der sie ihre Aufgaben erledigte. Wir vermissten sie sehr, da sie am Abschlusswochenende leider krank war. Es fiel uns nicht leicht ihr Fehlen auszugleichen. Als Leichtathletin war sie nicht nur beim Sportfest sehr gut zu gebrauchen, sondern auch zur Organisation unserer coolen Festlichkeiten.

Lucia war sehr nett und immer hilfsbereit. Außerdem konnte man sich voll und ganz auf sie verlassen. Ihre Fantasie kannte bei den Leveln keine Grenzen, und sie entwickelte uns drei schicke Designs.

Das Bergfest gestaltete sie mit anderen zusammen zu einer genialen Fete.

Kate unser jüngster Spross der Academy war zwar ruhig, aber sehr kreativ. Auch wenn sie es als jüngste nicht immer ganz leicht hatte, ging sie im Akademieleben nicht unter.

In unserem Kurs behauptete sie sich als eine solide Programmiererin, mag sie am Anfang auch manche Probleme mit der abstrakten Materie des Programmierens gehabt haben

– an dieser Stelle noch mal ein Lob an ihre Willensstärke sich durch diese Thematik durchzubeißen.

Larissa war zwar still, arbeitete aber trotzdem zügig und präzise.Wir konnten immer auf ihre Programme zählen. Mit ihrer souveränen Vortragekunst verzauberte sie uns alle, denn wenn sie vorne stand und ihr Thema präsentierte, verstand jeder gleich worum es ging.

Außerdem zeigt sie uns ihre Kontrolle über die Blockflöte am Hausmusikabend mit einer Kanon-Sonate.

Paul unser großer Koordinator, hat nie den Überblick verloren. Ohne ihn wären wir aufgeschmissen gewesen. Er fügte alle Klassen locker zusammen und sorgte für ein funktionsfähiges Spiel. Auch er trug zu mit viel Ideenreichtum und witzigen Sprüchen zum Erfolg unseres Kurses bei. Er scheute sich nicht Verantwortung zu übernehmen und zu sagen, was Sache ist.

Philipp war unser Crack, sowohl informatisch als auch physikalisch gesehen. Mit viel Spaß programmierte er die anspruchsvollen Animationen der Abschlusspräsentation und programmierte uns alle an die Wand.

Die Programmiersprache beherrschte er wie seine Muttersprache und mit seinem Fachwissen half er an allen Ecken und Enden.

Alex Klein kam er doch ganz groß raus, was er uns durch seine wunderbare Präsentationsfertigkeit bewies. Mit seiner aufgeweckten Art brachte er, wo auch immer er hin ging, gute Laune mit und trug damit super zur Atmosphäre bei.

Timo war der totale Physikjunkie, immer gut drauf und ließ sich oft von DanY und Mina nerven. Er hatte die Ruhe weg und meisterte früher oder später sämtliche Probleme.

Dank seiner Ruhe löste er nicht nur die verzwickten Rätsel, sondern minderte den Stress in manchen Situationen.

Dany war immer bester Stimmung, auch wenn ihr Magen laut knurrte. Sie war extrem gesprächig und heiterte alle anderen auf. Außerdem stand sie voll und ganz hinter einem PINKEN Breakout. Um uns eines ihrer Hobbies näher bringen zu können, hat sie eine Taekwando-KüA geleitet und Timo im persönlichen Schachspiel abgezogen.

Mina wusste immer, was zu tun war. Sie war sehr locker und ließ sich nicht aus der Ruhe bringen, stattdessen brachte sie frischen Wind in den Kurs. Zu jeder Situation fiel ihr ein guter Kommentar ein, mit dem sie alle zum Lachen brachte. Ihre freche und ehrliche Art macht sie einzigartig.

Kevin war neben Paul die Sammelzentrale unserer erarbeiteten Ergebnisse. Mit maßloser Vorfreude wartete er auf die fertigen Programmversionen und verpasste unserer Abschlusspräsentation ein fetziges Layout. Dabei ist er sämtliche Spielereien in OpenOffice auf den Grund gegangen.

Jörg unser junger Kursleiter, der uns immer mit Rat und Tat zur Seite stand. Auch in schwierigen Situationen behielt er den Überblick und war stets geduldig. Er motivierte uns auch über die längsten Durststrecken hinweg und fand immer die richtigen Worte, wenn er das benötigte „Hirnschmalz“ einforderte.

Matthias unser erfahrener und überaus kompetenter Kursleiter war sehr hilfsbereit und versuchte uns in jedem Moment die Welt der Informatik etwas näher zu bringen.

Er legte besonders Wert auf Ordnung im Quelltext und die Auflösung der Bilder, was uns oft in den Wahnsinn trieb. Er liebte es uns darauf hinzuweisen, dass wir unser Projekt „an die Wand fahren“. Seine Kaffeemaschine brachte uns sehr viele Besuche ein.

Melli war die beste Schülermentorin. Sie hat überall gute Stimmung verbreitet, so war es ständig lustig, wenn man auf sie getroffen ist.

Beim Hip-Hop war sie immer voller Power und Motivation und schaffte es auch im theoretischen Unterricht, uns schwierigen Physikstoff zu vermitteln.

Alle zusammen haben wir viel geschafft und jede Menge Spaß gehabt: Was sind wir? INFOS!

Einleitung

Paul Nickel

Breakout, Pinball und Friends – das war der Name unseres Kurses.

Zusammen mit unseren Kursleitern Jörg Richter, Matthias Taulien und Melina Becker haben wir uns in den Bereich der Computerspielentwicklung gewagt. Das Ziel des Kurses war es ein komplexes Computerspiel zu entwickeln, bei dem auch die Physik und die Mathematik eine wichtige Rolle spielt.

Um das Spiel auf den Bildschirm zu bringen erlernten wir die Programmiersprache Java.

Das Ziel des Kurses war es also ein physikalisch möglich korrektes Computerspiel zu entwickeln – aber es blieb die Frage, wie dieses Spiel aussehen sollte?

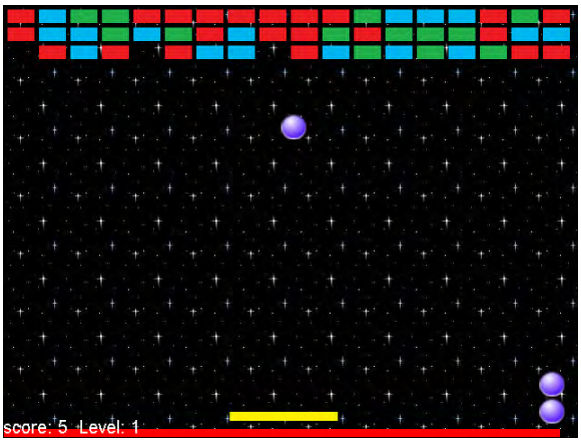
Angeregt von unserer Kursleitung entschieden wir uns für das Spiel Breakout.

Auf der Abbildung ist eine Vorabversion unseres Spieles zu sehen: Am oberen Spielfeldrand sind verschiedenfarbige Blöcke zu sehen, darunter befindet sich der Ball und das gelbgefärbte Paddel.

Ziel des Spieles ist es, alle Blöcke auf dem Spielfeld mithilfe des Balls zu zerstören. Der Ball wird mit dem Paddel im Spielfeld gehalten. Im Idealfall bewegt sich der Ball nach den physikalischen Gesetzmäßigkeit und wird an dem linken, rechten und oberen Spielfeldrand sowie an den Blöcken und (ganz wichtig) an dem Paddel reflektiert. Am unteren Spielfeldrand wird der Ball allerdings nicht reflektiert, sondern aus dem Spiel entfernt.

Diese Grundidee kann man um beliebig viele Modifikationen erweitern. So stehen einem in der Version, wie auf der Abbildung zu sehen ist, insgesamt drei Bälle zur Verfügung. Zudem haben die Blöcke mehrere Leben und besitzen eine gewisse Art Anziehungskraft. Am unteren Spielfeldrand ist eine rote Zeitleiste zu sehen. Ist diese abgelaufen wird eine neue Reihe von Blöcken erzeugt. Auf der linken, unteren Spielfeldseite wird der aktuelle Spielstand angegeben. Auf der rechten, unteren Spielfeldseite ist zu sehen, wie viele Bälle noch zur Verfügung stehen.

In unserem Spiel, das wir als Team zusammen programmiert haben, gibt es unter anderem eine ausführlichere Ausgabe des Spielstands, Buttons mit verschiedenen Funktionen, ein Hilfefenster, drei verschiedene Level, Bonusobjekte und vieles mehr.



Screenshot eines Breakout-Spieles

Aber nicht nur die Spielidee kann man beliebig modifizieren: Auch bei der physikalisch korrekten Umsetzung des Spieles kann man ‚wie Jörg immer so schön sagte, „beliebig viel Gehirnschmalz“ investieren. Bei insgesamt 3 794 Zeilen Code unseres Spieles machten die 1 139 Zeilen, die das Spiel physikalisch möglichst realistisch machen, einen erheblichen Anteil aus.

So haben wir uns in mehreren Einheiten mit verschiedenen physikalischen Themen auseinandergesetzt: Begonnen mit Vektorberechnung, über Gravitation bis hin zu Beschleunigung haben wir uns mit einigen Bereichen der Dynamik beschäftigt.

Objektorientierte Programmierung

Larissa Link

Unser Breakout-Spiel haben wir in Java, einer objektorientierten Programmiersprache entwickelt. Wichtige Aspekte der objektorientierten Programmierung und damit die größten Vorteile dieser Art der Programmierung sind: die hierarchische Klassenstruktur mit Vererbung, also die Aufteilung des Quelltextes in einzelne Klassen, und die Datenkapselung.

Klassen und Objekte

In unserem Spiel gibt es verschiedene Spielfiguren und Spielfunktionen, die unterschiedlich aussehen und verschiedene Eigenschaften und Fähigkeiten haben. In einem Programm ist für jedes dieser Elemente eine Klasse vorhanden. Klassen sind Baupläne, sozusagen die Bauanleitungen für die Objekte, die man dann im Spiel verwenden kann. Ein Objekt hat Fähigkeiten, bestimmte Aktionen auszuführen und hat typische Eigenschaften. Diese Eigenschaften – auch Attribute genannt – und die möglichen ausführbaren Handlungen, die man mithilfe von Methoden realisiert, sind in der jeweiligen Klasse aufgeführt. Jede dieser Klassen hat ihren eigenen Quelltext – das heißt eine Datei, in die man in der Programmiersprache Befehle hineinschreibt, um Methoden und Attribute zu definieren. Dies ist ein großer Vorteil der objektorientierten Programmierung: man hat nicht einen großen Programmtext, sondern jede Klasse hat einen Bereich, eine Datei für sich, in der programmiert werden kann. Das ist vor allem wichtig, wenn man in Gruppen arbeitet. Jede Arbeitsgruppe kann beispielsweise eine eigene Klasse programmieren, damit gleichzeitig an verschiedenen Funktionen des Programms gearbeitet werden kann. Später lassen sich dann alle Klassen zu einem gemeinsamen Programm zusammenführen. Sowohl die Attribute, als auch ihre Werte und die Methoden einer Klasse lassen sich nur innerhalb des Quelltextes genau dieser Klasse direkt verändern; nur durch Methoden kann man beim Programmieren auf andere „fremde“ Klassen zugreifen. Das nennt man Datenkapselung.

Hierarchie und Vererbung

Wenn man eine Klasse geschrieben hat, kann man beliebig viele Objekte dieser Klasse erzeugen. Diese haben dann die jeweiligen definierten Eigenschaften und können auch real ihre Methoden ausführen. Die Klassen eines Programms müssen aber nicht alle gleichgestellt sein. Man kann unter ihnen auch eine Hierarchie aufbauen. Das heißt, man kann von einer Klasse Unterklassen erzeugen. Nur durch diese Hierarchie unter den Klassen kann ein sehr wichtiger Aspekt der objektorientierten Programmierung zum Tragen kommen, nämlich die Vererbung. Wenn man von einer Klasse eine neue Unterklasse erstellt, und von dieser Objekte erzeugt, erben diese automatisch alle Attribute und Methoden der Oberklasse. Außerdem kann man in der Unterklasse nun noch zusätzliche Eigenschaften und Fähigkeiten hinzufügen. Man muss so nicht alle bereits in der Oberklasse vorhandenen Attribute und Methoden nochmal schreiben! Dies erspart Arbeit – die neue Unterklasse hat schon alle in der Oberklasse gegebenen „Voraussetzungen“ und lässt sich dazu noch weiter spezifizieren.

Wie funktioniert eine Animation am Computer?

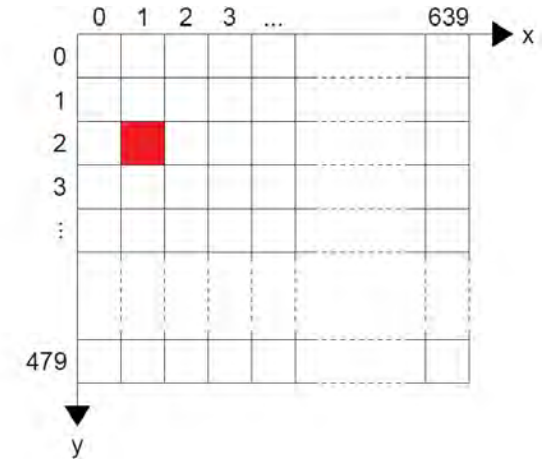
Lucia Constanze Grosse

Täglich, wenn man am Computer arbeitet oder nur im Internet surft, wird man mit sich bewegenden Grafiken konfrontiert. Das gleiche passiert auch in unserem Computerspiel, denn was wäre ein Breakout-Spiel ohne einen animierten Ball?

Bevor man ein Objekt bewegen kann, muss man es zunächst zeichnen können.

Zeichnen einer Grafik

Wenn man eine Grafik am Computer zeichnet, legt man ihre Position durch Koordinaten fest.



Dazu nutzt man ein normales zweidimensionales kartesisches Koordinatensystem, welches

sich von dem, das wir aus dem Mathematikunterricht kennen, ein wenig unterscheidet.

Die x-Achse zeigt zwar wie gewohnt von links nach rechts, die y-Achse jedoch von oben nach unten. Die markierte Kachel hat also die Koordinaten (1|2).

Damit ein Objekt erscheint, muss es genau wie auf einem Papier gezeichnet werden. Dafür benötigen wir eine Methode, da wir schließlich auch sichtbare Objekte verwenden. Die Methode `zeichneDich()` ist bei uns für das Zeichnen eines Objekts zuständig. Die Grafik wird mit einem Objekt namens `g2d` der Klasse `Graphics2D`, vergleichbar mit einem Multifunktionsstift, konstruiert und gezeichnet.

Mit der Methode `g2d.setColor()` legt man fest, welche Farbe das zu zeichnende Objekt erhalten soll.

Danach zeichnet man beispielsweise mit der Methode `g2d.fillRect(100, 200, 50, 10)` ein Rechteck, das 50 Pixel breit und 10 Pixel hoch ist; die linke obere Ecke des Rechtecks hat die Koordinaten (100|200)

In der Praxis setzen wir nicht sofort Zahlen ein, sondern Variablen. Dies ermöglicht Rechtecke mit unterschiedlicher Größe und Position zu zeichnen. Bei uns werden die Abmessungen sogar dynamisch an die Spielfeldgröße angepasst.

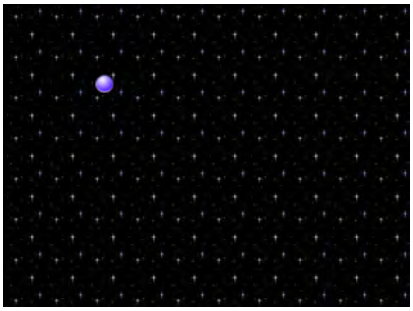
Animieren einer Grafik

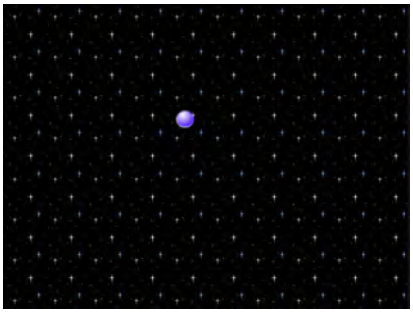
Für die Bewegung der Objekte ist die Methode `bewegeDich()` zuständig. Objekte werden vom Computer bewegt, indem man ihre Koordinaten verändert. Um ein Objekt zum Beispiel nach rechts zu verschieben, verändert man die x-Koordinate. Um es sowohl horizontal als auch vertikal zu verschieben, verändert man die x-Koordinate wie auch die y-Koordinate.

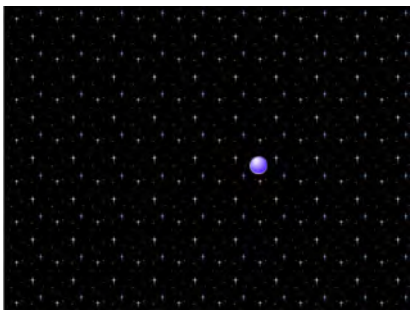
In der Methode `bewegeDich()` weisen wir dem x-Wert und dem y-Wert jeweils neue Werte zu. Das Gleichheitszeichen entspricht in diesem Fall nicht dem gebräuchlichen Gleichheitszeichen, sondern einer Zuweisung.

```
x = x + 20
y = y + 10
```

Wenn man diese Methode mehrmals hintereinander ablaufen lässt, befindet sich der Ball jedes Mal an einer anderen Position:







Je schneller man die Methode hintereinander ablaufen lässt, desto gleichmäßiger und realer wirkt die Bewegung, weil wir die Sprünge zu den neuzugewiesenen Koordinaten nicht mehr wahrnehmen. Dies kann man sich wie bei einem Daumenkino vorstellen.

Entwicklungsumgebungen Greenfoot und Eclipse

Alexander Quint, Kevin Wang

In unserem Kurs haben wir mit zwei sogenannten Entwicklungsumgebungen gearbeitet: Greenfoot und Eclipse. Beide basieren auf der Programmiersprache Java, unterscheiden sich aber deutlich.

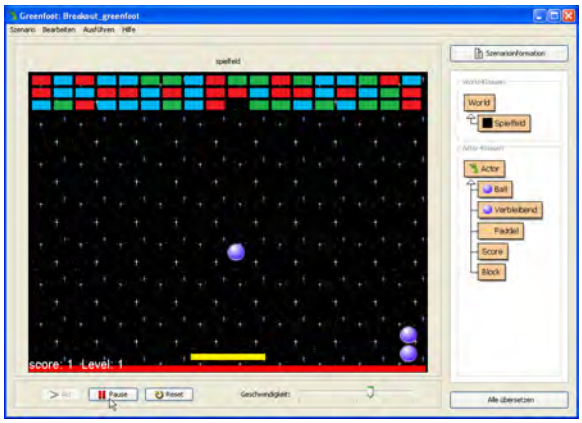
Eine Entwicklungsumgebung hilft dabei, Programme zu entwickeln. Sie formatiert den Code, zeigt Syntaxfehler an und macht Lösungsvorschläge zur Fehlerbehebung. Außerdem übersetzt der Compiler den Quellcode in Befehle, die vom Computer ausgeführt werden können.

Greenfoot ist zum Erlernen von Java gedacht.

Man kann eine Spielfeld-Welt mit graphischen Objekten bevölkern, deren Eigenschaften und Fähigkeiten man programmiert – ideal für ein kleines Computerspiel

Eclipse dagegen eine sehr mächtige Entwicklungsumgebung ist, die auch von professionellen Programmierern benutzt wird. In Eclipse ist kein Grundgerüst vorgegeben, also muss man alles selbst erstellen.

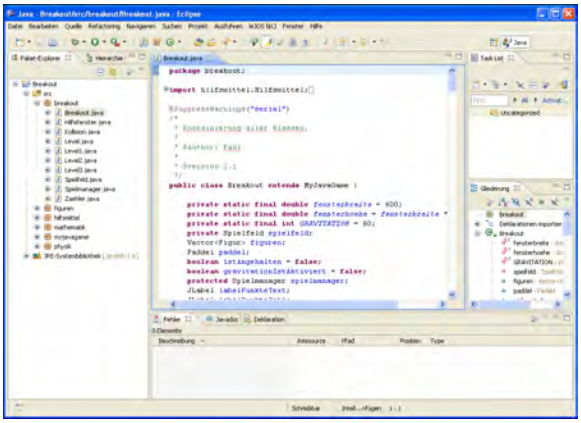
Greenfoot haben wir am Eröffnungswochenende und bei unserer Hausaufgabe verwendet – während der Sommerakademie jedoch haben wir mit Eclipse programmiert.



Hauptfenster von Greenfoot

Die obige Abbildung zeigt das Hauptfenster von Greenfoot. In der Mitte befindet sich das Spielfeld. In dieses kann man direkt Objekte einfügen, und man kann deren Methoden aufrufen oder Attribute inspizieren. Auf der rechten Seite sieht man den Bereich mit den verschiedenen Klassen und Unterklassen, z. B. die Klasse `Paddel`. Mit den Buttons „Run“ (vor Spielstart), „Pause“ (während des Spiels) und „Reset“ unterhalb des Spielfelds kann man das Spiel starten, stoppen und in den Anfangszustand versetzen. Mit dem Regler „Geschwindigkeit“ wird die Geschwindigkeit der beweglichen Objekte (Ball, Paddel) eingestellt.

Im untenstehenden Bild wird die Arbeitsplattform von Eclipse dargestellt. Hier sieht man, dass Eclipse deutlich komplexer als Greenfoot ist.



Arbeitsplattform von Eclipse

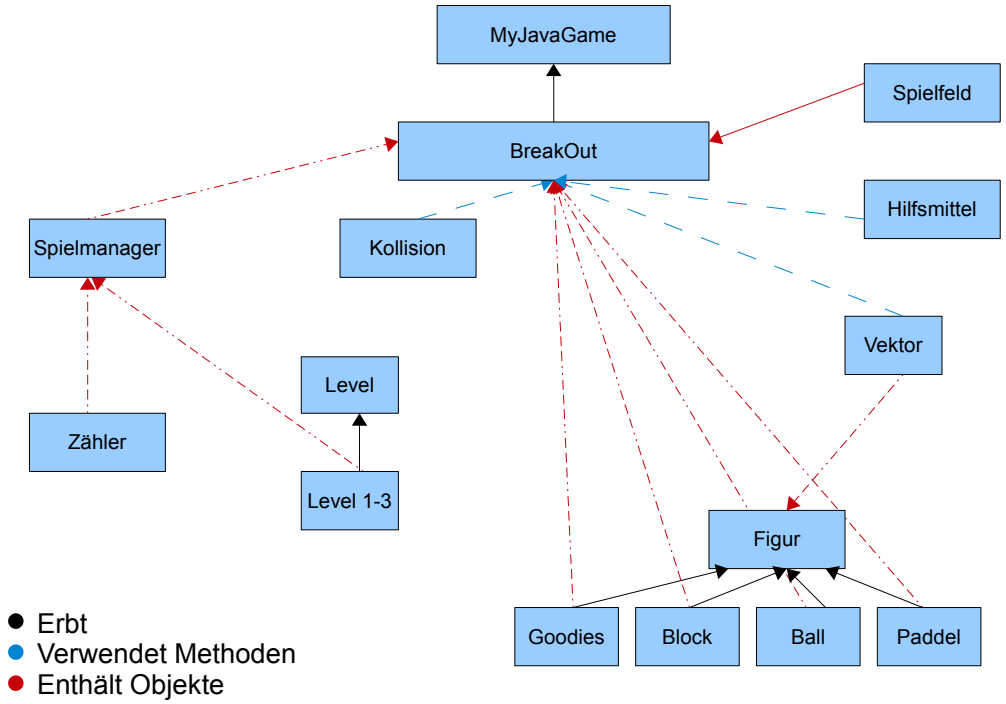
Auf der linken Seite sieht man einen Bereich, auf der sich alle Projekte, Pakete und Klassen befinden. In der Mitte ist das Editorfenster, in dem man den Quelltext für Programme erstellen kann. Auf der oberen Leiste gibt es ein einen grünen Button mit einem weißen Pfeil. Wenn man auf diesen klickt, wird das Spiel kompiliert und gestart. Unten gibt es einen Ausgabebereich für Fehlermeldungen etc. Auf der rechten Seite befindet sich eine Klassenhierarchie. Und wenn man einen Programmierfehler macht, wird dieser durch ein rotes Feld am Rand gekennzeichnet. Wenn man auf eine Glühbirne am Rand klickt, bekommt man Lösungsvorschläge.

Die Entwicklung unseres Spiels

Daniela Schäfer, Jasmina Vercrüsse

Für ein komplexes Spiel wie Breakout war es sinnvoll die Arbeit aufzuteilen.

Dazu mussten wir noch bevor wir anfingen zu programmieren, überlegen, wie unser Spiel zum Schluss aussehen sollte und eine Programmstruktur entwerfen. Die einzelnen Fachgruppen mussten sich dann Bereiche aussuchen und die damit verbundenen Klassen programmieren. Mit dieser Vorgehensweise erzielten wir einen großen Erfolg, weil sich jede Gruppe nur mit



ihrem Themengebiet und dessen Problemen beschäftigen musste. Teilweise wurde, zum Beispiel der Mathematikgruppe, spezielles Fachwissen übermittelt. Ein weiterer Vorteil war, dass

verschiedene Klassen gleichzeitig programmiert werden konnten, was uns sonst nicht möglich gewesen wäre.

Um das Programmieren der Klassen zu koordinieren, wurden zu Beginn Etappenziele festgelegt. Die unterschiedlichen Versionen wurden an einem gesammelten Speicherort zusammengefügt und mit den jeweiligen Versionsnummern bezeichnet. Nachdem alle Klassen in einer Versionsnummer übereingestimmt hatten, wurden sie zu einem gesamten Spiel mit eben dieser Versionsnummer zusammengefügt.

Ein temporäres Problem war, dass jede Fachgruppe unterschiedlich viele Aufgaben für die jeweiligen Versionen hatte und unterschiedlich schnell gearbeitet hat. Somit waren die einzelnen Bereiche oft verschieden weit entwickelt und es konnte nicht immer die neueste Version des Spiels erstellt werden, obwohl sie für einige Gruppen zur Weiterarbeit nötig gewesen wäre. Man musste deshalb provisorisch arbeiten: Hierzu sprach man sich vorher ab, wie die benötigte Methode heißen solle, und verwendete sie geeignete Platzhalter.

Ein weiteres Problem konnte sich in der Kommunikation ergeben: Es kam vor, dass Fachgruppen an Aufgaben arbeiteten, für die sie gar nicht zuständig waren oder die zuständige Fachgruppe erledigte ihre Aufgaben nicht, weil sie mit anderen Aufgaben beschäftigt war.

Trotz dieser Probleme bot die Gruppenarbeit viele Chancen und verringerte den Arbeitsaufwand für die gesamte Gruppe. Ohne diese Aufteilung wären wir mit unserem Spiel wohl nie so weit gekommen.

Das Herz des Spiels

Kevin Wang, Paul Nickel

Das Herz unseres Breakout-Spiels sind die Klassen `MyJavaGame`, `Breakout` und die Klasse `Spielmanager`. Die Klasse `MyJavaGame` enthält ein Grundgerüst für die Animation. `Breakout` koordiniert das gesamte Spiel, und die Klasse `Spielmanager` verwaltet die Angaben, die den Spielstand betreffen, wie zum Beispiel das aktelle Level, den Punktestand und die Anzahl der Leben.

Die Klasse `MyJavaGame`

Doch wie funktioniert das Zusammenspiel dieser Klassen? Die Klasse `MyJavaGame` wurde von unseren Kursleitern vorbereitet, um uns den Einstieg zu erleichtern. In dieser Klasse war die Rumpf-Animation bereits vorhanden: Zum Beispiel eine Methode, die immer wieder ausgeführt wird, und in der auch das (alledings noch leere) Spielfeld gezeichnet wird.

Die Klasse `Breakout`

`Breakout` definierten wir als eine Unterklasse von `MyJavaGame`, sodass `Breakout` alle Methoden von `MyJavaGame` ausführen kann. In der Klasse `Breakout` werden neue Spielfiguren und das Spielfeld erzeugt, ihre Positionen berechnet und gezeichnet – sie ist die zentrale Klasse unseres Spiels. Außerdem wird der Punktestand angezeigt und geprüft, ob eine Taste gedrückt wird.

Betrachten wir die Klasse `Breakout` näher. Der Quelltext der Klasse `Breakout` besteht im Wesentlichen aus vier Abschnitten:

Im Konstruktor stehen die Befehle, die beim Start des Spiel bearbeitet werden. In ihm werden zum Beispiel das Spielfeld, die Spielfiguren und die Spalte für den Spielstand erzeugt.

Im zweiten Abschnitt finden alle Berechnungen statt. Dort wird die aktuelle Position des Balls berechnet und geprüft, ob eine Kollision stattfindet.

Des weiteren gibt es den Bereich, die für das Zeichnen des Spiels zuständig ist.

Am Schluss befindet sich der Teil, der für die Behandlung der Tasten zuständig ist. In ihm ist die gesamte Steuerung des Spiels festgelegt: Wenn zum Beispiel die p-Taste gedrückt wird, wird das Spiel pausiert.

Man sieht, dass die Klasse `Breakout` ziemlich komplex ist. Damit Breakout nicht zu unübersichtlich wurde, haben wir andere wichtige Aufgaben in eigene Klassen ausgelagert.

Die Klasse `Spielmanager`

Die Berechnung des Spielstands (Punkte, Leben, Level) übernehmen deshalb die Klassen `Spielmanager` und `Zaehler`. Außerdem erzeugt der Spielmanager neue Level und Goodies. Goodies sind Objekte, die zufällig herunterfallen, wenn man einen Block zerstört hat. Sie können dann vom Spieler eingesammelt werden. Damit dies funktioniert, kommunizieren die Klassen `Spielmanager` und `Breakout` miteinander.

Um den aktuellen Spielstand zu berechnen, steht der Spielmanager in engem Kontakt mit der Klasse `Zaehler` und der Klasse `Kollision`. Ein Beispiel: Wenn ein Block zerstört wurde, ruft die Klasse `Kollision` den Zaehler auf, der dann den Punktestand erhöht. Der Punktestand wiederum wird dann vom `Spielmanager` abgefragt.

Soweit es uns möglich war, haben wir versucht, den Spielmanager unabhängig von der Klasse `Kollision` zu gestalten. In der Praxis sieht das dann so aus: Sollten keine Blöcke mehr auf dem Spielfeld sein, muss nicht die Klasse `Kollision` dem Spielmanager mitteilen, dass er ein neues Level erzeugen soll. Vielmehr prüft der Spielmanager selbsttätig, ob auf dem Spielfeld noch Blöcke sind. Ist dies nicht der Fall, erzeugt er automatisch ein neues Level. Somit entstehen zwei getrennte Abläufe, die die Fehlerbehebung vereinfachen und das Spiel übersichtlicher machen.

Insgesamt bilden die drei Klassen `MyJavaGame`, `Spielmanager` und `Breakout` das Herz unseres Spiels. Sie bekommen den Großteil aller Informationen übermittelt und entscheiden auf deren Basis, welche Aktion als nächstes ausgeführt werden soll.

Das Paket `figuren`

Kate Lau

Das Paket `figuren` beinhaltet die Klasse `Figur` und als deren Unterklassen `Ball`, `Block`, `Paddel` und `Goody`.

Somit ist das Paket `figuren` für die verschiedenen Objekte auf dem Spielfeld zuständig.

Die Klasse `Figur`

Die Klasse `Figur` enthält alle diejenigen benötigten Methoden und Eigenschaften, die alle Figuren gemeinsam haben: die Position, die Breite, die Höhe, die Geschwindigkeit, die Beschleunigung, die Farbe und das Abbild. Für die Position, die Beschleunigung und die Geschwindigkeit werden Vektoren benutzt, da man mit ihnen geschickter rechnen kann.

Für die Attribute sind Getter- und Settermethoden implementiert. Das sind Methoden, mit denen man die Werte der Attribute erfragen oder modifizieren kann.

Alle Figuren werden aufs Spielfeld gezeichnet, müssen also eine Methode `zeichneDich()` bereitstellen. Nun wird ein Ball natürlich anders gezeichnet als ein Block. Deshalb kann der Code von `zeichneDich()` nicht in der Klasse `Figur` stehen. Sie ist deshalb hier lediglich deklariert, dadurch ist sichergestellt, dass in allen Unterklassen eine solche Methode implementiert werden muss. In der Fachsprache nennt man solche Methoden „abstrakt“.

Die Klasse `Paddel`

Da die Klasse `Paddel` eine Unterklasse von `Figur` ist, erbt ein Paddel-Objekt alle Methoden und Attribute von der Klasse `Figur`. Deshalb braucht sie nur wenige zusätzliche Methoden. Zum Beispiel kennt das Paddel noch die beiden Methoden `getGanzLinks()` sowie `setGanzLinks()` und analog die Methoden `getGanzRechts()` und `setGanzRechts()`. Außerdem wird die Methode `zeichneDich()` mit Leben gefüllt.

Die Klasse `Ball`

Ein Ball hat drei Attribute mehr als eine Figur: Durchschlagskraft, Masse und Radius. Die Masse wird bei der Kollision von zwei Bällen benötigt. Durchschlagskraft ist ein boolscher Wert (ein Wahrheitswert, der entweder wahr oder falsch sein kann) und beschreibt, ob der Ball von den Blöcken reflektiert wird oder nicht.

Die Klasse `Block`

Ein Block besitzt noch das Attribut `anzLeben`. Es zählt, wie viele Leben ein Block hat, also wie oft ein Block vom Ball getroffen werden muss, bis er verschwindet.

Die Klasse `Block` hat mehrere Konstruktoren für die verschiedenen Blöcke, etwa für ruhende Blöcke oder sich bewegende Blöcke.

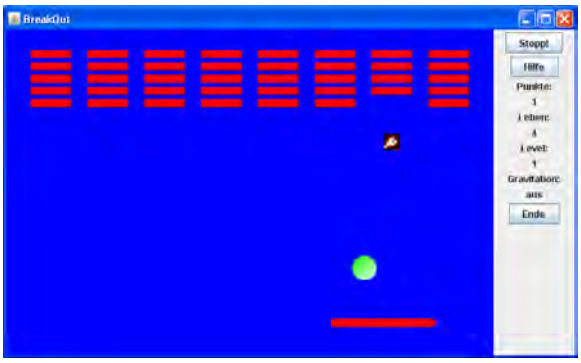
Sie kennt außerdem Methoden, um Leben hinzuzufügen bzw. abzuziehen und die Anzahl der Leben abzufragen.

Level

Lucia Constanze Grosse

Um für den nötigen Spielspaß zu sorgen, haben wir verschiedene Level entworfen, die unterschiedlichen Schwierigkeitsgraden entsprechen und besondere Funktionen, wie zum Beispiel Goodies besitzen.

Level 1



Wenn ein Block zerstört wird, kann mit einer gewissen Chance ein sogenanntes „Goody“ entstehen. Goodies sind, wie der Name schon verrät, Bonusobjekte. Um die Wirkung des Goodies auszulösen, muss man das Goody mit dem Paddel auffangen.

Die Goodies und ihre Wirkungen

Beim Auffangen dieses Goodies wird das Paddel breiter.

Dieses Goody sorgt dafür, dass der Ball eine Durchschlagskraft bekommt und so durch Blöcke hindurch fliegen kann und nicht von ihnen reflektiert wird.

Mit Auffangen dieses Goodies erhält man fünf Extrabälle.

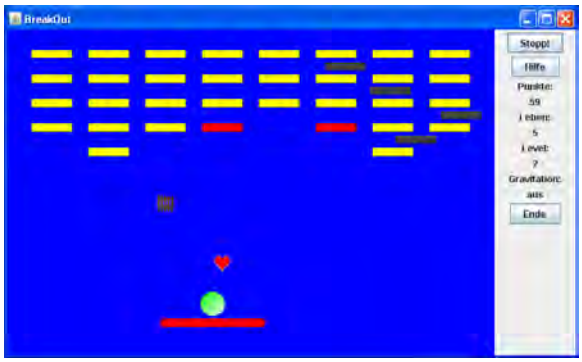
Unter dem Paddel entsteht eine Sicherheitsbande, die einen Ball reflektiert, falls es dem Spieler nicht gelingt, den Ball mit dem Paddel aufzufangen.

Dieses Goody führt dazu, dass man ein weiteres Leben bekommt.

Dieses Goody müsste man eigentlich als „Bady“ bezeichnen, da es bewirkt, dass das Paddel schmaler wird.

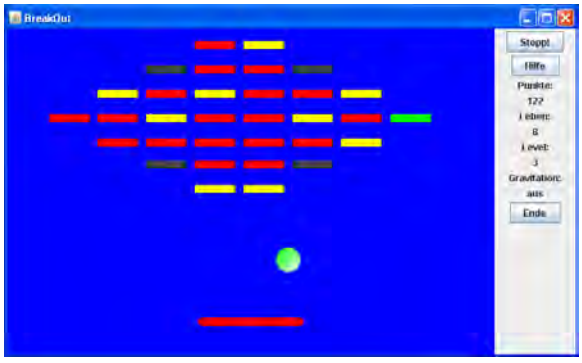
Level 2

Im zweiten Level kommen zu den Funktionen aus Level 1 noch „Steine“ hinzu. Das sind nichtverschwindende Blöcke, die den Ball dennoch normal reflektieren. In diesem Level bewegen sich die Steine horizontal hin und her. Außerdem gibt es Blöcke, die man 2 treffen muss, bevor sie verschwinden. Man sagt, sie haben 2 Leben.



Level 3

In unserem 3. Level gibt es Blöcke mit entweder einem, zwei oder drei Leben. Die Anzahl der Leben eines Blocks lässt sich an seiner Farbe erkennen. Rot steht für ein Leben, gelb für zwei und grün für drei Leben. Wieder gibt es Steine, die sich diesmal aber nicht bewegen.



Beschleunigung

Paul Nickel, Philipp Provenzano

Um das Spiel noch realistischer zu gestalten, sollten die sich bewegende Körper eine Schwerkraft erfahren.

Beim Fall eines Körpers zum Boden erfährt er eine konstante Geschwindigkeit.

Um die Gravitation nachvollziehen zu können, mussten wir uns aus diesem Grund erst einmal mit dem Thema Beschleunigung beschäftigen.

Beim Beschleunigen erfährt ein Körper eine fortwährende Zunahme der Geschwindigkeit – er wird schneller.

Die durchschnittliche Beschleunigung $\vec{a}$ definiert man als:

$$\vec{a} = \frac{\vec{v}_2 - \vec{v}_1}{t_2 - t_1}$$

Dabei sind $\vec{v}_1$ und $\vec{v}_2$ die Geschwindigkeit zu den Zeitpunkten t_1 und t_2. Für $t_2 - t_1$ schreiben wir kürzer Δt.

Mit dieser Formel berechnen wir dann die geänderte Geschwindigkeit des Körpers:

$$\vec{v}_{neu} = \vec{v}_{alt} + \vec{a} \cdot \Delta t$$

Die neue Geschwindigkeit verwenden wir wiederum zur Berechnung der neuen Position des Körpers:

$$\vec{s}_{neu} = \vec{s}_{alt} + \vec{v}_{neu} \cdot \Delta t$$

$\vec{s}$ ist bei uns die Position des Mittelpunkts eines Objekts.

Da durch diese Formel die Geschwindigkeit diskret beschrieben wird, ist die Berechnung nicht hunderprozentig genau – für unsere Zwecke jedoch vollkommen ausreichend.

Kollisionen . . .

Samuel Brielmaier

. . . am Spielfeldrand

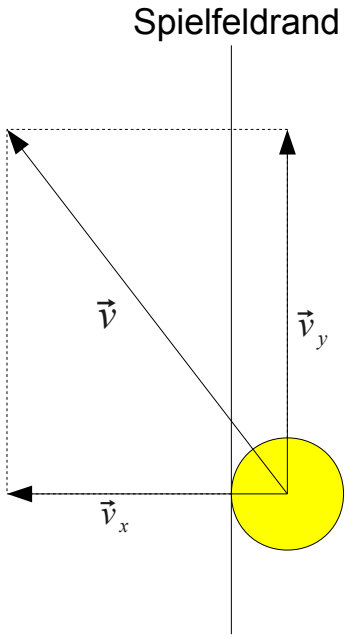
Damit der Ball bei unserem Spiel im Spielfeld gehalten wird, muss der Ball an den Rändern reflektiert werden. Um zu prüfen, ob der Ball

den Spielfeldrand berührt, wird jedes Mal, wenn die Figuren neu gezeichnet werden, abgefragt, ob der Abstand des Ballmittelpunkts zum Rand kleiner oder gleich wie sein Radius.

Wenn dies der Fall ist, wird ein Teilvektor (entweder die x-, oder die y-Komponente) des Geschwindigkeitsvektors gespiegelt.

Konkret bedeutet das bei dem unten zu sehenden Beispiel:

Die x-Richtung soll entgegengesetzt werden, deshalb wird die x-Komponente des Geschwindigkeitsvektors invertiert. Dadurch gilt die Gesetzmäßigkeit „Einfallswinkel ist gleich Ausfallwinkel“.

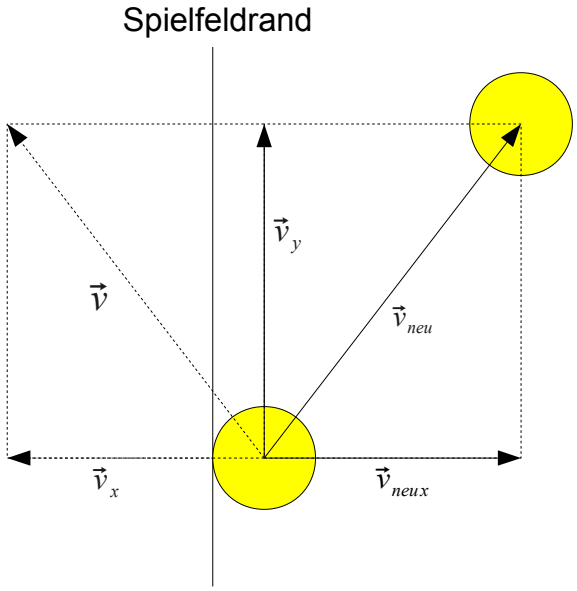


Reflexion am linken Spielfedrand: Der Abstand des Balles zum linken Rand ist hier genau der Radius. $\vec{v}$ steht für den Geschwindigkeitsvektor, $\vec{v}_x$ für seine x-Komponente und $\vec{v}_y$ für die y-Komponente.

Bei der Kollision des Balles am oberen Spielfeldrand wird statt der x-Komponente einfach die y-Komponente umgedreht.

. . . mit einem Block

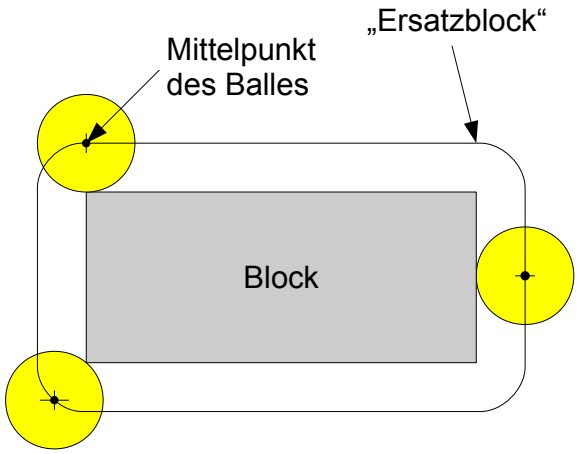
Dazu haben wir die Kollision in 8 verschiedene Fälle eingeteilt: Die Kollision mit der Ober- und Unterseite, der linken und rechten Seite und den vier Ecken bildet hierbei jeweils einen Fall.



Die invertierte x-Komponente der Geschwindigkeit ergibt mit der unveränderten y-Komponente den neuen Geschwindigkeitsvektor.

An der linken und rechten Seite verhält sich der Ball exakt genau so wie am linken und rechten Spielfeldrand, denn es wird immer nur der x-Vektor invertiert. Dementsprechend wird bei der Kollision an der Ober- und Unterseite der y-Vektor invertiert, so wie es auch beim oberen Spielfeldrand der Fall ist.

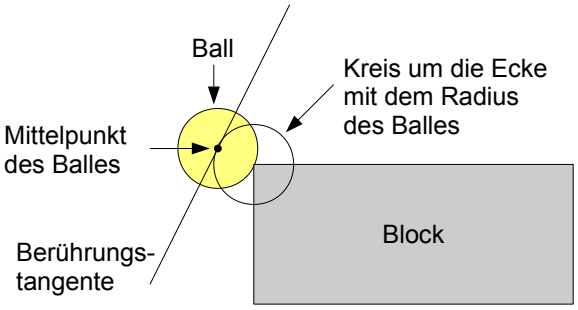
An den Ecken ist es dagegen erheblich komplizierter. Deshalb haben wir uns angesehen, wo der Mittelpunkt des Balles ist, wenn eine Kollision stattfindet.



Die Reflexion verhält sich genauso, als wenn man zum Berechnen der neuen Werte den Mittelpunkt des Balles an einem „Ersatzblock“ re-

flektieren lässt, der zum Rand des echten Blocks den Abstand des Radius des Balles hat.

Somit ist die Berechnung deutlich einfacher. Der Mittelpunkt des Balls wird im Kollisionsfall „Ecke“ an der Berührungstangente mit einem Kreis (um die Ecke und dem Radius des Balles) reflektiert.

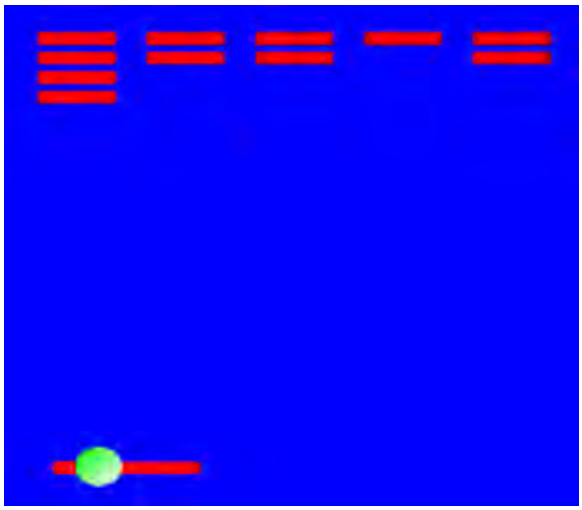


Der Ballmittelpunkt wird an einem Kreis um die Ecke reflektiert

Ball-Paddel-Problem

Wenke Grahneis

Als wir die ersten Versionen unserer Kollision des Balls am Paddel testeten, trat immer wieder ein Problem auf: Wenn das Paddel den Ball (mit einer Geschwindigkeit $\neq 0$) von der Seite traf, blieb der Ball im Paddel hängen, wie man hier erkennen kann:



Das wollten wir natürlich nicht und machten uns auf die Suche nach der Ursache. Dabei fanden wir folgendes heraus: Wenn der Ball nach der Reflexion am Paddel eine geringere Geschwindigkeit als das Paddel hat – das Paddel also schneller ist als der Ball – meldet es nach dem nächste Rechenschritt wieder Kontakt mit dem Ball. Das führt zu einer weiteren Reflexion des Balls. Da der Ball nun wieder in seine ursprüngliche Richtung fliegt, hat er im nächsten Rechenschritt höchstwahrscheinlich wieder Kontakt mit dem Paddel und wird reflektiert. Durch dieses dauernde Reflektieren des Balls hat es für uns den Anschein, als hätte das Paddel den Ball gefangen.

Unsere erste Idee zur Behebung des Problems war es, den Ball nach seiner Reflexion am Paddel noch zusätzlich einen Zeitschritt zurückzusetzen. Wir testeten die neue Version und merkten schnell, dass diese Lösung zwar bei niedrigen Geschwindigkeiten funktionierte, aber bei hohen Geschwindigkeiten des Paddels keinen Erfolg hatte. Wir stellten das Problem mit einem physikalischen Versuch nach, bei dem zwei Wagen kollidieren und erkannten, dass der Ball eigentlich vom Paddel nicht nur reflektiert, sondern, sofern das Paddel auch eine Geschwindigkeit besitzt, entsprechend dem Impulserhaltungssatz auch beschleunigt wird.

Für diese Rechnung muss man die Kollision aus dem „Paddelsystem“ betrachten, das heißt sich das Koordinatensystem aus Sicht des Paddels vorstellen. Wenn sich also ein Ball im eigentlichen Koordinatensystem mit einer Geschwindigkeit von -10 Pixel pro Sekunde nach links auf das Paddel zubewegt und das Paddel sich ihm mit einer Geschwindigkeit von 50 Pixel pro Sekunde nähert, würde er sich dem Paddel im „Paddelsystem“ mit einer Geschwindigkeit von -60 Pixel pro Sekunde nähern. Der Grund dafür ist, dass das Paddel im „Paddelsystem“ immer die Geschwindigkeit 0 hat und man dem Ball somit die Geschwindigkeit des Paddels abzieht. Nun wird der Ball am Paddel reflektiert. Das hat zur Folge, dass seine Geschwindigkeit invertiert wird, also aus -60 Pixel pro Sekunde, $+60$ Pixel pro Sekunde werden. Um diese Geschwindigkeit nun wieder in das normale Koordinatensystem umzurechnen, muss man die Geschwindigkeit des Paddels wieder dazu addieren: $60\,\mathrm{px/s} + 50\,\mathrm{px/s} = 110\,\mathrm{px/s}$. Also hat der Ball nach der Kollision mit dem Paddel eine Geschwindigkeit von 110 Pixel pro Sekunde.

Diese Rechnung bauten wir dann in unseren Quelltext der Klasse Kollision ein. Wir führten einen Test der neuen Version durch, und es zeigte sich, dass das Paddel den Ball erfreulicherweise nicht mehr fangen konnte. Leider trat nun ein neues neues Problem auf: Durch den starken Kick wurde der Ball oft sehr schnell – im obigen Beispiel hat sich die Geschwindigkeit mehr als verzehnfacht! Durch diese hohe Geschwindigkeit wurde der Ball für den Spieler unkontrollierbar. Wir beschlossen, die Methode so zu modifizieren, dass der Ball bei der Kollision einen geringeren Teil der Paddel-Geschwindigkeit übertragen wird, so als wäre das Paddel weich.

So hatten wir schließlich eine physikalisch korrekte und gleichzeitig spielbare Lösung gefunden!

Kollision von zwei Bällen

Philipp Provenzano, Timo Simnacher

Die Behandlung der Kollision zwischen zwei Bällen hat uns am meisten Kopfzerbrechen bereitet.

Bei der Berechnung der Position der Bälle haben wir uns bisher immer an dem Koordinatensystem des Spiels orientiert. Bei der Kollision mussten wir allerdings ein Koordinatensystem mit einem anderen Ursprung verwenden: Wir arbeiteten im Schwerpunktsystem.

In diesem System ist der Ursprung des Koordinatensystems der Schwerpunkt der beiden Bälle. Alle Bewegungen werden von dort aus betrachtet.

Der Ort des Schwerpunktes ist:

$$\vec{x}_{Sp} = \frac{m_1 \cdot \vec{x}_1 + m_2 \cdot \vec{x}_2}{m_1 + m_2}$$

Dabei sind m_1 und m_2 die Massen und $\vec{x}_1$ und $\vec{x}_2$ die Koordinaten der Mittelpunkte der beiden Kugeln.

Mit der Ableitung nach der Zeit kann man die Geschwindigkeit des Schwerpunkts berechnen. Aus

$$\frac{\mathrm{d}\vec{x}_1}{\mathrm{d}t} = \vec{v}_1 \text{ und } \frac{\mathrm{d}\vec{x}_2}{\mathrm{d}t} = \vec{v}_2$$

folgt:

$$\vec{v}_{Sp} = \frac{m_1 \cdot \vec{v}_1 + m_2 \cdot \vec{v}_2}{m_1 + m_2}$$

Dann ist die Geschwindigkeit des Schwerpunktes nach dem Stoß:

$$\vec{u}_{Sp} = \frac{m_1 \cdot \vec{u}_1 + m_2 \cdot \vec{u}_2}{m_1 + m_2}$$

Der Impulserhaltungssatz besagt:

$$m_1 \cdot \vec{v}_1 + m_2 \cdot \vec{v}_2 = m_1 \cdot \vec{u}_1 + m_2 \cdot \vec{u}_2$$

Daraus folgt: $\vec{u}_{Sp} = \vec{v}_{Sp}$ – die Geschwindigkeit des Schwerpunkts ist also erhalten!

Im Schwerpunktsystem ist die Geschwindigkeit des Schwerpunkts 0:

$$\vec{v}_{Sp} = 0$$

$$0 = \frac{m_1 \cdot \vec{v}_1 + m_2 \cdot \vec{v}_2}{m_1 + m_2}$$

$$0 = m_1 \cdot \vec{v}_1 + m_2 \cdot \vec{v}_2$$

Die rechte Seite der Gleichung ist der Gesamtimpuls – er ist im Schwerpunktsystem also immer null!

Elastische Stöße im Schwerpunktsystem

Wir berechnen nur elastische Stöße, also solche, bei denen keine Bewegungsenergie verloren geht, etwa durch Verformung. Sie sollen auch reibungsfrei sein, so dass die Bälle keinen Drall erhalten.

Zunächst überzeugt man sich davon, dass die Beträge der Geschwindigkeiten einzeln erhalten sind:

Wie wir gerade gesehen haben, ist der Gesamtimpuls im Schwerpunktsystem immer null. Dies formt man um:

$$m_1 \cdot \vec{v}_1 + m_2 \cdot \vec{v}_2 = 0$$

$$\vec{v}_2 = -\frac{m_1}{m_2} \cdot \vec{v}_1$$

Und ebenso nach dem Stoß:

$$m_1 \cdot \vec{u}_1 + m_2 \cdot \vec{u}_2 = 0$$

$$\vec{u}_2 = -\frac{m_1}{m_2} \cdot \vec{u}_1$$

Da wir elastische Stöße betrachten, ist die Gesamtenergie vor und nach dem Stoß gleich. Die Bewegungsenergie eines Körpers errechnet sich aus dem Produkt der Masse und dem Quadrat der Geschwindigkeit: $E = \frac{1}{2} \cdot m \cdot \vec{v}^2$. Es gilt also:

$$\frac{1}{2}m_1 \cdot \vec{v}_1^2 + \frac{1}{2}m_2 \cdot \vec{v}_2^2 = \frac{1}{2}m_1 \cdot \vec{u}_1^2 + \frac{1}{2}m_2 \cdot \vec{u}_2^2$$

Nun kann man für $\vec{v}_2$ und $\vec{u}_2$ einsetzten, was man oben erhalten hat:

$$m_1 \cdot \vec{v}_1^2 + m_2 \cdot \left(-\frac{m_1}{m_2} \cdot \vec{v}_1\right)^2 = m_1 \cdot \vec{u}_1^2 + m_2 \cdot \left(-\frac{m_1}{m_2} \cdot \vec{u}_1\right)^2$$

$$m_1 \cdot \vec{v}_1^2 + m_2 \cdot \frac{m_1^2}{m_2^2} \cdot \vec{v}_1^2 = m_1 \cdot \vec{u}_1^2 + m_2 \cdot \frac{m_1^2}{m_2^2} \cdot \vec{u}_1^2$$

$$\vec{v}_1^2 + \frac{m_1}{m_2} \cdot \vec{v}_1^2 = \vec{u}_1^2 + \frac{m_1}{m_2} \cdot \vec{u}_1^2$$

$$\left(1 + \frac{m_1}{m_2}\right) \cdot \vec{v}_1^2 = \left(1 + \frac{m_1}{m_2}\right) \cdot \vec{u}_1^2$$

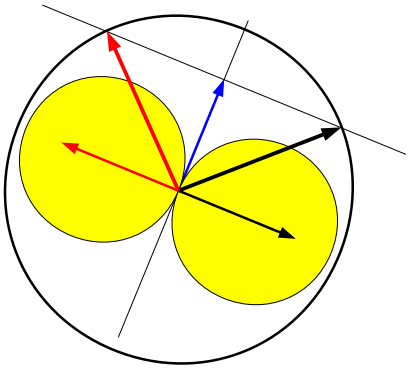
$$\vec{v}_1^2 = \vec{u}_1^2$$

$$|\vec{v}_1| = |\vec{u}_1|$$

Damit ist gezeigt, dass sich im Schwerpunktsystem der Betrag der Geschwindigkeit jedes Balls nicht ändert, seine Richtung möglicherweise schon.

Um die neue Richtung zu finden, zerlegt man die Ballgeschwindigkeit zu dem Zeitpunkt, zu dem sich die Bälle berühren, in eine Komponente parallel und eine Komponente orthogonal zur Berührungstangente.

Bei einem reibungsfreien Stoß wirkt keine Kraft tangential zur Oberfläche. Deshalb ändert sich die Parallelkomponente (blau) nicht. Auch der Betrag der Geschwindigkeit bleibt gleich. Dadurch muss auch der Betrag der Orthogonalkomponente gleich bleiben. Für die Orthogonalkomponente sind deshalb nur zwei verschiedene Möglichkeiten denkbar: die Orthogonalkomponente vor dem Stoß (dünner schwarzer

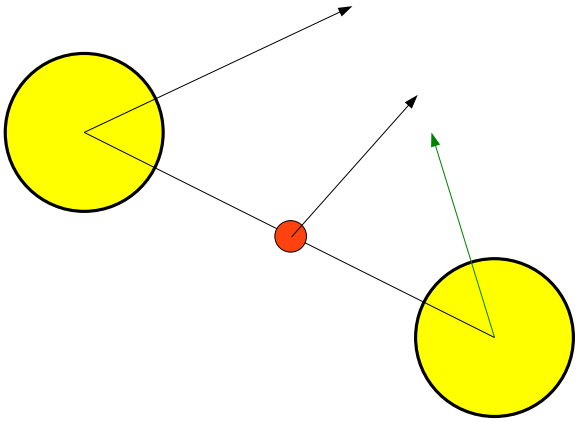


Pfeil) oder der Gegenvektor dazu (dünner roter Pfeil). Somit erhält man im Schwerpunktsystem die Geschwindigkeit nach dem Stoß (rot) indem man die ursprüngliche Geschwindigkeit (schwarz) einfach an der Berührtangente spiegelt.

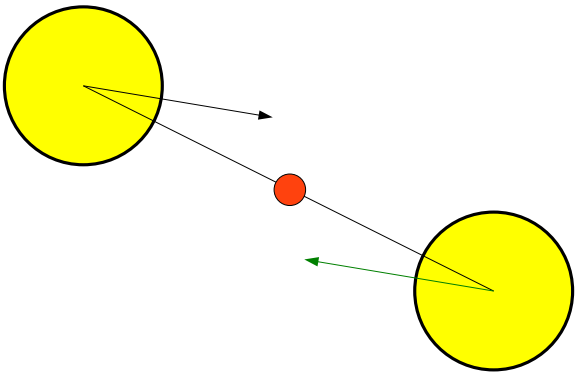
Stöße im Koordinatensystem unseres Spiels

Wie kann man mit diesem Wissen nun die Stöße in unserem Spielfeld-System berechnen? Ganz einfach: Man rechnet zunächst alles ins Schwerpunktsystem um, berechnet dort den Stoß und transformiert schließlich wieder zurück ins System des Spielfeldes.

Zunächst rechnen wir also die Geschwindigkeiten der Bälle in unsrem Koordinatensystem in das Schwerpunktsystem um. Dazu subtrahiert man die Geschwindigkeit des Schwerpunkts von den anderen Geschwindigkeiten.

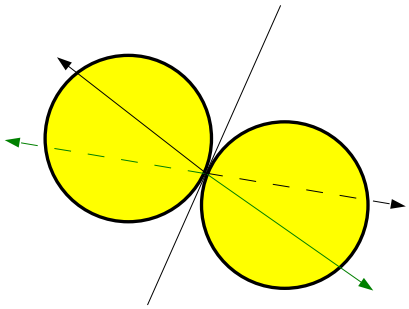


Bewegung von zwei Bällen und deren Schwerpunkt im „normalen“ Koordinatensystem



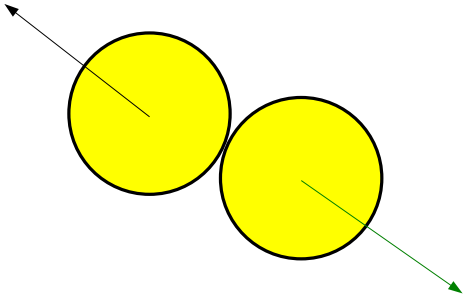
Bewegung von zwei Bällen im Schwerpunktsystem

Anschließend erfolgt die Berechnung der Geschwindigkeiten nach dem Stoß im Schwerpunktsystem.

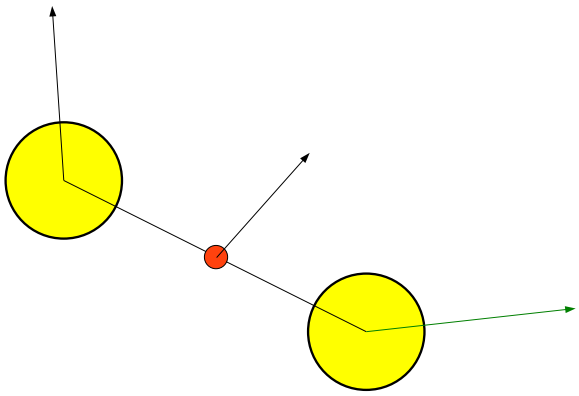


Spiegeln der Geschwindigkeitsvektoren der Bälle bei einer Kollision an der Berührungstangente (Schwerpunktsystem; gestrichelt die jeweils ursprüngliche Richtung)

Zum Schluss müssen wir die neuen Geschwindigkeitsvektoren wieder in unser „normales“ Koordinatensystem zurückrechnen, indem man die Geschwindigkeit des Schwerpunktes wieder zu den Ballgeschwindigkeiten hinzuaddiert.



Bewegung der Bälle nach der Kollision (Schwerpunktsystem)



Bewegung der Bälle nach der Kollision („normales“ Koordinatensystem)

Ausblick

Wenke Grahneis

Was hätten wir gemacht, wenn wir noch mehr Zeit gehabt hätten? – Eine Menge!

Nachdem wir alle die Grundlagen beherrschten und die einzelnen Arbeitsgruppen eingeteilt waren, wage ich zu sagen, dass es in jedem von uns angefangen hat „rumzuträumen“. Ein richtig professionelles Computerspiel, das man vielleicht später im Media Markt im Regal stehen sieht. Okay, das ist jetzt vielleicht die Extremversion, aber an Erweiterungsideen mangelte es sicher nicht. Zum Beispiel wollten wir in den vierten Level so etwas wie schwarze Löcher einbauen, die den Ball verschwinden lassen, oder auch Blöcke, die bei der Zerstörung das Spielfeld für einige Sekunden weiß werden lassen („flash“), waren beim Level-Team im Gespräch. Aber leider waren die zwei Wochen im Nu vorbei und es hieß: „Fertig werden, die Abschlusspräsentation steht an!“. Die letzten Verbesserungen wurden noch getätigt, und dann stand sie, unsere Version 2.0. Das heißt aber nicht, dass es nie eine Version 5.0 geben wird. Wir können jetzt ja programmieren (zumindest so grob) und hatten solchen Spaß dabei, dass es sich wahrscheinlich niemand nehmen lassen wird, unser „Breakout“ zu Hause zu erweitern und noch ein paar Special Features oder Goodies einzubauen. Denn nichts ist schöner, als wenn der Computer dann endlich macht, was er soll!