3k

Entwicklung eines Computerspiels

Maturarbeit von Julian Dunskus, 6e

Kantonsschule Hohe Promenade, Gymnasium, Zürich

Schuljahr 2014/15

Referent: Stefan Müller

Korreferent: Michael Liebich

1 Anfang

1.1 Motivation

Schon als ich 9 Jahre alt war, zeigte mein Vater mir "Scratch" (www.scratch.mit.edu), eine auf Anfänger im Programmieren ausgelegte Plattform, um einfache Anwendungen und Spiele zu entwickeln, indem man farbige Blöcke zusammenzieht, um so den Code visuell aufzubauen. Ich war von Scratch begeistert, da ich endlich selber das machen konnte, was ich schon lange bewundert hatte. Später kaufte er mir ein Buch, "Java mit Eclipse für Kids" von Hans-Georg Schumann, was die Grundlagen von Java kinderfreundlich erklärt. Durch dieses Buch lernte ich die Syntax von Java kennen, hörte aber wieder auf mit Java, da ich Scratch zugänglicher und ansprechender fand.

Im September 2010 zeigte mir ein Freund "Minecraft", ein inzwischen sehr bekanntes Spiel, was einem eine Welt aus 1x1x1m-Blöcken bereitstellt, in der man gegen Monster kämpfen, Bäume fällen, Stein minen, Höhlen erforschen und alles erdenkliche bauen kann. Minecraft wurde mein absolutes Lieblingsspiel. Das wichtigste an dieser ganzen Geschichte ist aber, dass Minecraft in Java programmiert ist, und dass man, wenn man das Spiel mit einem Werkzeug dekompiliert hat, den Code bearbeiten kann. Dadurch kann man neue Blöcke hinzufügen, verrückte Sachen mit der Mechanik anstellen und alles ändern. Von diesen Möglichkeiten hingerissen tauchte ich in den Code ein und konnte mein durch Scratch und das Buch angeeignete Wissen anwenden. Ich schaute auf YouTube Anleitungen, die mir beibrachten, wie ich bestimmte Sachen tun konnte, zum Beispiel einen neuen Block hinzuzufügen. Durch diese Auseinandersetzung mit dem Code erlernte ich die Mehrheit meines heutigen Programmierkönnens. Schliesslich fing ich parallel dazu an, auch eigene Spiele und Anwendungen in Java zu programmieren.

1.2 Themenwahl

Bei den Präsentationen der Maturarbeiten wurde mir schnell klar, dass ich als Maturarbeit etwas programmieren wollte. Mir war aber noch nicht klar, ob ich ein Spiel oder eine sonstige Anwendung entwickeln wollte, bis ich während einer Mathestunde die Idee zu 3k hatte. Ich zeichnete dann gleich eine Skizze vom fertigen Spiel. Als ich wieder zuhause war, setzte ich diese Skizze dann gleich in Photoshop um (Abbildung 1). Ich schickte dann eine Anfrage an meinen damaligen Chemielehrer, Michael Liebich, der aber schon besetzt war und mich nicht noch dazunehmen konnte. Er leitete die Anfrage aber für mich weiter an zwei Mathematiklehrer, von denen Stefan Müller sich meiner Arbeit annahm. Nach unserer ersten Besprechung machte ich mich gleich an die Arbeit und kreierte die Dateien, die ich für dieses Spiel brauchen würde.

Abb. 1: Der erste Entwurf, mit Photoshop skizziert

2 Spielkonzept

2.1 Grundkonzept

Das Spiel läuft in einem sechseckigen Feld – mit Platz für 19 kleinere Sechsecke – ab. Diese Sechsecke haben entweder eine Zahl drauf, oder sie sind Powerups (mehr dazu später). Die Zahlen auf allen Sechsecken sind definierbar durch die Formel $b \cdot 2^n$; b wird ab hier als "Basis" bezeichnet, n als "Exponent". Am Anfang sind schon einige Sechsecke da, danach kommt nach jedem Zug ein neues dazu, mit Exponent 0 oder 1. In einem Zug gibt man allen Sechsecken eine Richtung an, in die sie sich bewegen, bis sie entweder auf den Rand des Spielfelds oder ein anderes Sechseck treffen. Wenn ein Sechseck auf ein anderes mit derselben Zahl trifft, schieben sie sich zu einem neuen zusammen, das nun die Summe der beiden Sechsecke als Zahl hat. Durch intelligentes Zusammenschieben der richtigen Sechsecke kommt man so schnell auf hohe Zahlen. Wenn man zwei 1536er zusammenschiebt, bekäme man 3072, was als 3k (Tausend) abgekürzt wird (daher der Name des Spiels). Zweimal 1536k gibt 3m (Millionen). Danach, obwohl man praktisch nie so weit käme, geht es dann mit 3b (Billionen), 3t (Trillionen) und 3q (Quadrillionen). Durch diese Kurzformen verliert man auch bei hohen Zahlen nicht den Überblick. Ziel des Spiels ist, auf möglichst hohe Zahlen zu kommen. Es ist vorbei, wenn keine Bewegung mehr möglich ist.

2.2 Schwierigkeitsstufen

Es gibt in 3k vier verschiedene Schwierigkeitsstufen: leicht, mittel, schwer und verrückt. In der leichtesten Schwierigkeitsstufe hat man nur Sechsecke mit der Basis 3, und um das Ganze noch einen Tick schwieriger zu machen, ist bei dieser Stufe in der Mitte des Spielfelds eine Blockade, wo man nichts durchschieben kann (Abbildung 2). Ohne diese Blockade ist diese Schwierigkeitsstufe bloss witzlos einfach. Bei der mittleren Stufe kommen zu den Sechsecken mit der Basis 3 auch noch welche mit Basis 2 dazu. Dadurch wird das Spiel sofort viel schwieriger. Die schwierige Stufe hat noch die Basis 5 dabei. Auf dieser Stufe darf man nichts falsch machen, sonst ist das Spiel praktisch sofort vorbei. In der letzten Stufe, wo die Basis 7 zusätzlich noch hinzukommt, ist es fast unmöglich, lange zu überleben. Man hat schon Glück, wenn man es auch nur schafft, ein Powerup zu aktivieren.

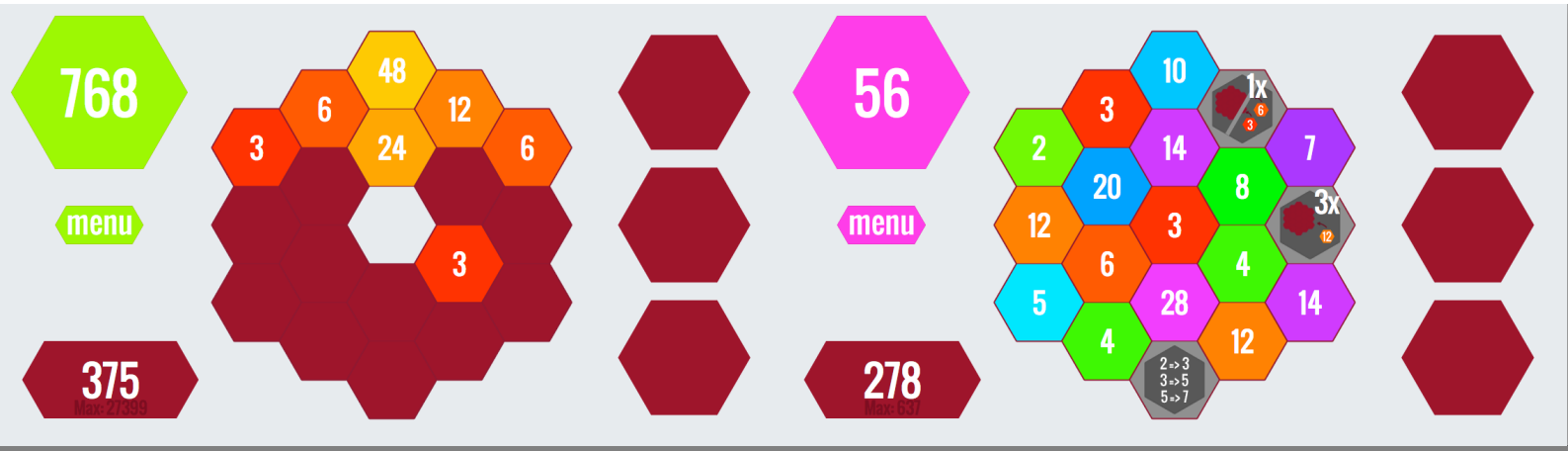


Abb. 2: Links das Spiel mit Blockade, rechts die verrückte Stufe

2.3 Powerups

Zusätzlich zu den normalen Sechsecken gibt es auch noch Powerups. Powerups werden beliebig statt normaler Sechsecke nach einem Zug zum Spielfeld hinzugefügt. Ein Powerup entfaltet erst seine Wirkung, wenn man es aktiviert, indem man neben ihm zwei gleiche Sechsecke zusammenschiebt. Es gibt acht verschiedene Powerups, die alle unterschiedliche Auswirkungen haben. Ab hier werden alle Powerups und ihre Funktionen erklärt. Bei den variablen Powerups – denen, wo 3x dabeisteht – ist 3 nur ein Beispiel. Es werden je nach Powerup beliebige Zahlen generiert. In der Beschreibung wird diese Zahl mit n umschrieben.

In den nächsten n Zügen verdoppelt sich das Ergebnis, wenn man zwei Sechsecke zusammenschiebt.

In den nächsten n Zügen kommen nur Sechsecke mit Exponent 2 dazu (12 / 12, 8 / 12, 8, 20 / 12, 8, 20, 28).

Alle Sechsecke mit dem tiefsten Exponenten werden verdoppelt (also 3 / 3, 2 / 3, 2, 5 / 3, 2, 5, 7).

Alle zusammenpassenden Sechsecke, egal wo, werden zusammengefügt.

In den nächsten n Zügen kommen keine neuen Sechsecke dazu.

Es werden die Zahlen von n beliebigen Sechsecken verdoppelt.

Alle Sechsecke mit dem tiefsten Exponenten werden gelöscht.

Die Basis b von allen Sechsecken wird um einen Schritt erhöht.

Abb. 3: Alle im Spiel enthaltene Powerups

3 Grafik

3.1 Design

Als ich den ersten Entwurf des Spiels in Adobe Photoshop entwickelte, hatte ich die Idee, Photoshop "Styles", im Grunde genommen grafische Formatvorlagen, für die verschiedenen Elemente zu verwenden. "Styles" können verschiedene grafische Effekte auf Formen und Bilder, die der User macht, anwenden. Diese erkennt man gut unten im linken Bild: die leeren Formen sind schattiert, die Sechsecke haben abgeschrägte Kanten, wodurch sie leicht dreidimensional anmuten, und der ganze Hintergrund ist ein einziges Rechteck mit einer "Style" drauf. Also gestaltete ich das Spiel in dem Stil (Abb. 4, links). Nachher überlegte ich mir jedoch, das Spiel auch auf iPhone und co. zu bringen, weswegen ich eine Test-App machte, für die ich keinen Code schreiben musste, ein sogenanntes "proof of concept." Diese Test-App zeigte einfach verschiedene Bilder an, die durch Knöpfe miteinander verbunden waren. Als ich jedoch mein Spiel in diesem Stil auf meinem Handy erblickte, bemerkte ich, das dies seit iOS 7 gar nicht mehr zum Stil des Betriebssystems passen würde, was auch einige Freunde gleich empfanden. Also überlegte ich mir etwas anderes. Ich nahm die Styles raus und gestaltete das gesamte Spiel neu, ohne unnötige Dekoration (Abb. 4, rechts). Es gibt jetzt nur noch einfarbige Flächen, die den Fokus auf den Inhalt lenken. So gefällt es mir viel besser: es sieht sauberer und aufgeräumter aus.

Abb. 4: Links das erste Design, rechts das moderne "flache" Design

3.2 Powerups

Im ersten Entwurf hatte ich schon ein solches Powerup (Abbildung 5), was aber im Nachhinein ziemlich hässlich aussieht, sicherlich nicht nach etwas, was man unbedingt einsammeln will. Also probierte ich einiges. Meine erste Idee war, einen Kreis in dem Sechseck zu ziehen. Dies setzte sich insofern durch, als ich mich schlussendlich auf ein um 90° gedrehtes Sechseck im grösseren festlegte. Ausserdem entschied ich mich gegen den gelben Farbton, da es sonst normale Sechsecke mit fast genau derselben Farbe geben konnte. Somit legte ich mich auf zwei Rottöne fest. Ich überlegte mir noch, ob ich einen inneren Schatten in der inneren Form haben sollte, um sie eingelassen aussehen zu lassen, aber entschied mich dagegen, was chronologisch mit der Änderung im Gesamtstil auf flache Farben einherging. Dann baute ich das Powerup

erstmals ins Spiel ein. Später entschied ich mich für eine neue Schriftart – mehr dazu im nächsten Unterkapitel – und ausserdem änderte ich dabei auch gleich das Farbschema des Powerups, weil das Rot einem leeren Feld zu ähnlich aussah.

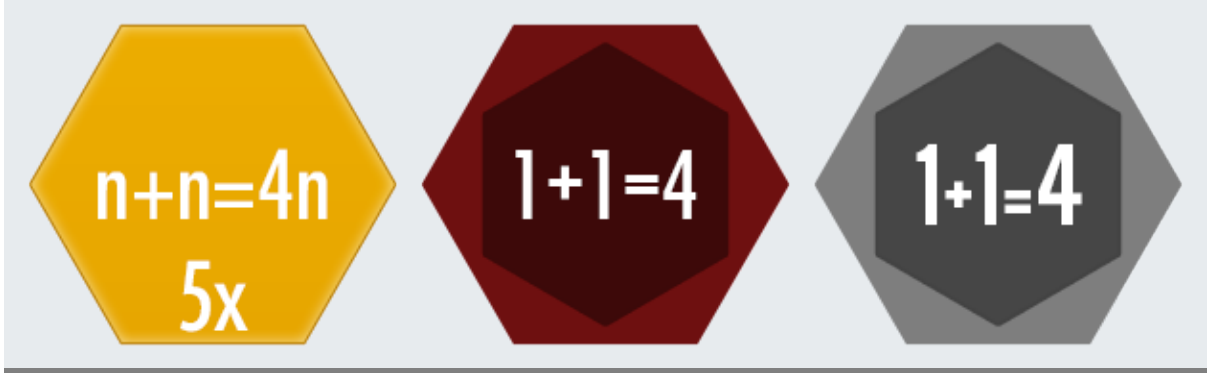


Abb. 5: Der Entwurf, die erste Implementierung, das Endprodukt (v.l.n.r.)

3.3 Schriftart

Als ich anfangs nach einer Schriftart für das Spiel suchte, legte ich mich sehr schnell auf **Futura Condensed Medium** fest (Abbildung 6, links). Aber später, als ich mir überlegte, dass ich das Spiel auch öffentlich machen und verteilen wollen würde, fiel mir auf, dass ich diese nicht vorinstallierte Schrift mitliefern müsste, was aber mit dem Kopierrecht in Konflikt geraten würde. Also suchte ich nach einer ähnlichen gratis-Schrift, die mir für das Spiel gefiel. So kam ich auf die Schriftart **Oswald** (Abbildung 6, rechts), die man auf Google Fonts gratis herunterladen und die ich mit dem Spiel weiterverteilen kann. Ein weiterer Vorteil von Oswald ist, dass es verschiedene Gewichtsstufen gibt: alles von "extra light" (besonders dünn) bis "heavy" (besonders fett). So kann ich unterschiedliche Stufen für verschiedene Elemente benutzen, wie ich es auch in dieser Dokumentation mache. Rückblickend sieht die ursprüngliche Schriftart ein bisschen professioneller aus, aber ich bin mit meiner zweiten Wahl auch zufrieden.

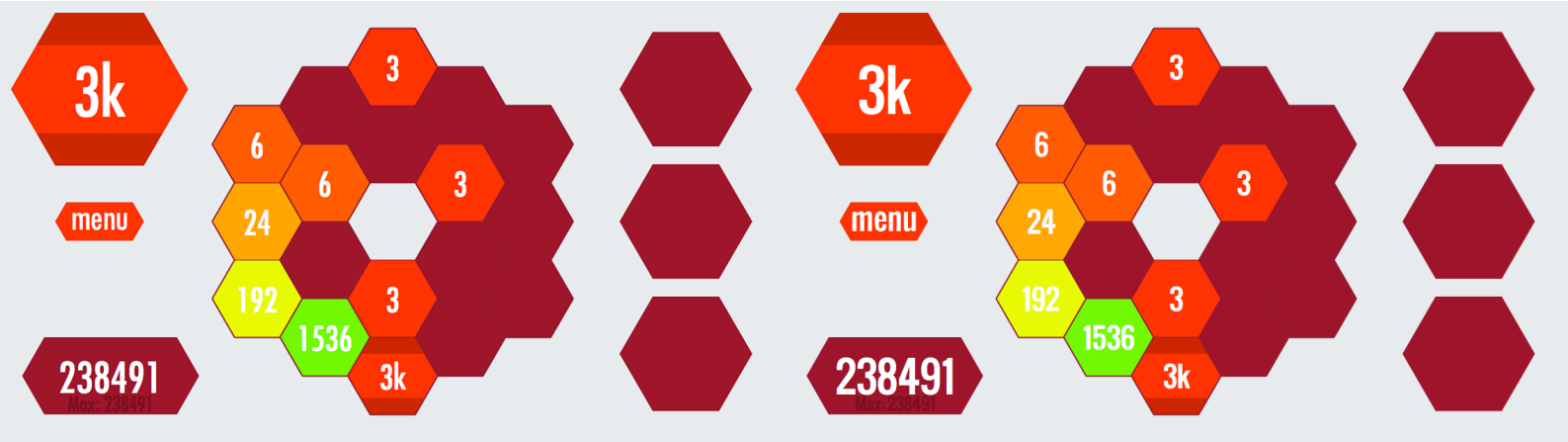


Abb. 6: Links die seriöse ursprüngliche Schrift, rechts die verspielte "Oswald"

4 Spielentwicklung

4.1 Spritesheets

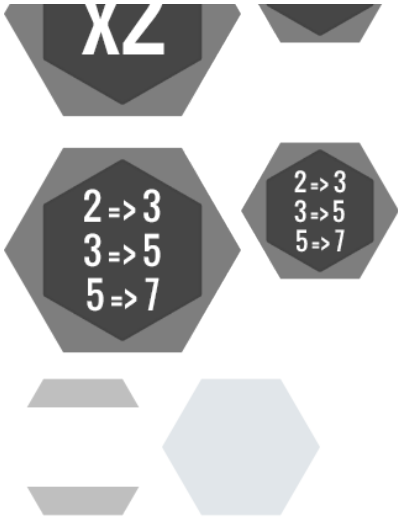


Abb. 7: Ausschnitt eines Spritesheets

Das erste, was ich tat, war, mir sogenannte Spritesheets zu machen. Spritesheets sind Bilddateien, die lauter kleinere Bilder beinhalten, welche das Spiel benutzt. So muss man nicht ganz viele kleine Dateien abspeichern, sondern kann alles, was zusammengehört, in ein einziges solches Spritesheet konsolidieren. Links ist ein Ausschnitt des Spritesheets, was ich für besondere Sechsecke benutze; Die Datei heisst "Hexagons_Special.png". In meinem Code lade ich das Bild, dann schneide ich mit verschieden Methoden, die ich mir dafür bereitgelegt habe, die Elemente heraus, und speichere sie in programmatische Bildvariablen ein. So isoliere ich dann in diesem Fall die Powerups in grosser und kleiner Auflösung, die Blockade (das graue Sechseck) und die Überlagerungen für Sechsecke ab der zehnten Stufe, die sie von einfachen Sechsecken leichter zu unterscheiden machen. Ich packe diese sogenannten Sprites dann in "Arrays", im Grunde genommen Listen von Objekten, durch die ich schnell und geordnet auf das Sprite zugreifen kann, was ich brauche, um es dann auf dem Bildschirm anzuzeigen. Eine solche Liste enthält zum Beispiel alle 36 Farben von Sechsecken mit einer Grösse von 256x256 Pixeln, die ich benutze, um oben links das höchste bisher erreichte Sechseck zu präsentieren. Eine andere enthält diese Sechsecke in 128x128 Pixeln, um sie auf dem Spielfeld anzuzeigen. Noch zwei andere enthalten je alle 10 Powerups in 192x192 und 128x128, jeweils für die Anzeige rechts und das Spielfeld, und eins enthält die Beschreibungen für die Powerups.

4.2 Programmierung

Als ich die ersten Spritesheets beisammen hatte, fing ich mit programmieren an. Das erste, was ich schrieb, war die Klasse[1], die als erste ausgeführt wird, wenn man das Spiel startet. Diese Klasse macht ein Fenster auf dem Bildschirm für das Spiel, delegiert das Anzeigen seines Inhalts an die "Renderer"-Klasse und lädt alle anderen Klassen. Nachher bekam diese Klasse natürlich noch viel mehr Funktionalität, wie z.B. die "InputManager"-Klasse als "Handler" für Mausklicks, -bewegung und Tastaturbefehle zu registrieren, aus dem "Save File" die gespeicherten Spieldaten zu lesen, oder sie dort reinzuschreiben. Als nächstes machte ich eine "Resources"-Klasse, die die Bilder, die ich für das Spiel brauche, lädt und sie anderen Klassen bereitstellt, wie zum Beispiel der "Renderer"-Klasse, welche ich als nächstes kreierte. Bisher hatte ich also drei Klassen, die zusammen das Hintergrundbild des laufenden Spiels in einem neuen Fenster anzeigten.

[1] Eine Klasse in Java ist eine Textdatei, die im Grunde genommen ein Bauplan für ein Objekt ist, was zur Laufzeit des Programms danach erstellt werden kann. Dieser Bauplan gibt dem Objekt bestimmte Methoden, die man ausführen kann, und Variablen, deren Wert man ändern kann. Mehr Informationen in der Codeanalyse.

Nach einigen Tagen hatte ich einen Prototypen, der zwar halbwegs funktionierte, aber noch sehr viele Fehler hatte. Zum Beispiel konnten Sechsecke nur ein Feld auf einmal verschoben werden, und die Animation hatte auch noch viele Probleme. Ausserdem rutschte immer mal wieder ein Sechseck über den Rand hinaus und verursachte Programmfehler, die das ganze Spiel abstürzen liessen. Aber dadurch, dass ich jetzt eine fast schon spielbare Version meines Spiels hatte, war ich extrem motiviert, daran weiterzuarbeiten. Oftmals erwischte ich mich dabei, wie ich noch nach Mitternacht am Programmieren war, obwohl ich am nächsten Tag in die Schule musste. Der Grossteil der Entwicklung geschah über die Sommerferien, wo ich fast täglich bis nach Mitternacht arbeitete und einen grossen Schritt weiterkam. Ich bin selten so produktiv, wie ich bei der Entwicklung dieses Spiels war.

4.3 Schwierigkeiten

Die erste Schwierigkeit, die ich antraf, war die Steuerung. Ich musste einen Weg finden, Eingaben ("Input") von Tastatur oder Maus in eine von sechs Richtungen umrechnen. Die erste Idee, die mir kam, war, jeder Richtung eine Buchstabentaste auf der Tastatur zuzuweisen. Ich zentrierte sie um die Taste "J", und machte auch gleich eine Alternative für Linkshänder um "F" (Abbildung 8). Nach kurzem Testen empfand ich diese Methode jedoch als nicht intuitiv und unpraktisch. Also entwickelte ich zwei Alternativen: die Pfeiltasten und die Maus. Beide Methoden waren etwa gleich schwierig zu implementieren. Die Maussteuerung läuft über Ziehbewegungen, bei denen ich anhand des Winkels vom Endpunkt zum Startpunkt des Zugs die Richtung bestimme. Das läuft wie folgt: anhand des Unterschieds zwischen Start- und Endposition der Ziehbewegung berechne ich erst einmal, ob überhaupt weit genug gezogen wurde, um eine Bewegung auszulösen. (So kann man sie auch wieder abbrechen, wenn man aus Versehen angefangen hat zu ziehen.) Wenn der Abstand gross genug ist, rechne ich weiter, in welchem der umgebenden Sechstel der Endpunkt liegt. Durch einige Tricks wird das sehr einfach und effizient auszurechnen. Zuerst rechne ich mit den Beträgen der Koordinaten, sodass alle Punkte im oberen rechten Viertel liegen. Ich errechne, ob der Punkt oberhalb der 60°-Linie, d.h. im oberen Sechstel liegt. Liegt er darin, muss ich nur noch herausfinden, ob er im oberen oder unteren Sechstel liegt (auf Abbildung 8 entspricht dies *T* und *C*). Wenn nicht, gibt es nur noch vier Möglichkeiten (*R*, *G*, *D* und *V*), wobei ich anhand von einfachen Vergleichen (*grösser als*, *kleiner als*) herausfinden kann, welche es ist.

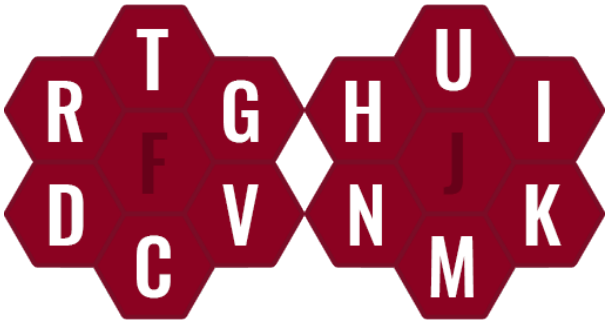


Abb. 8: Steuerung mit Buchstabentasten

Die Steuerung mit den Pfeiltasten hat zwar weniger mit Mathematik zu tun, aber um dafür zu sorgen, dass sich die Sechsecke immer in die Richtung bewegen, die man erwartet, ist einiges an Logik erforderlich. In Java bekommt man sogenannte "Events", – Ereignisse – wenn eine Taste gedrückt oder losgelassen wird. Durch solche Events erfährt man, welche Taste gedrückt bzw. losgelassen wurde, und kann gleich Code ausführen, um damit zu arbeiten. Rauf und runter sind einfach: einfach die jeweilige Pfeiltaste drücken und wieder loslassen. Aber bei den anderen vier Richtungen muss viel getüftelt werden, um die Steuerung intuitiv zu halten. Ich demonstriere dies anhand einer Bewegung nach oben rechts. Zuerst kommen einem vier Möglichkeiten in den Sinn (Tabelle 1):

Erstes Ereignis	Zweites Ereignis	Drittes Ereignis	Viertes Ereignis
↑ gedrückt	→ gedrückt	↑ losgelassen	→ losgelassen
↑ gedrückt	→ gedrückt	→ losgelassen	↑ losgelassen
→ gedrückt	↑ gedrückt	→ losgelassen	↑ losgelassen
→ gedrückt	↑ gedrückt	↑ losgelassen	→ losgelassen

Tabelle 1

Es ist aber nicht immer so einfach. Vielleicht entscheidet sich der Benutzer während einer Bewegung um, dann läuft das folgendermassen ab (Tabelle 2):

↑ gedrückt	← gedrückt	← losgelassen	→ gedrückt	→ losgelassen	↑ losgelassen
↑ gedrückt	← gedrückt	← losgelassen	→ gedrückt	↑ losgelassen	→ losgelassen
← gedrückt	↑ gedrückt	← losgelassen	→ gedrückt	↑ losgelassen	→ losgelassen
← gedrückt	↑ gedrückt	← losgelassen	→ gedrückt	→ losgelassen	↑ losgelassen

Tabelle 2

Theoretisch kann man dies unendlich mit Wechseln zwischen rechts und links erweitern, aber der fertige Algorithmus sollte damit gleich gut umgehen können wie mit den obigen Fällen. Jedenfalls sollte sich in all diesen acht Fällen schlussendlich eine Bewegung nach oben rechts ergeben, und die anderen Richtungen müssen auch funktionieren, ohne dass es Überschneidungen gibt. Um dies zu erreichen, benutze ich verschiedene "Booleans" (wahr/falsch), deren Werte beim Drücken oder Loslassen von Pfeiltasten schlau gesetzt und abgefragt werden, damit sich die erwartete Bewegung ergibt.

Als nächstes kam die eigentliche Bewegung. Ich musste irgendwie ausrechnen, welche Sechsecke sich kombinieren würden, über die nächste Zeit anzeigen, wie sie sich verschieben, und bei dem ganzen Durcheinander darauf achten, dass sich keine schon kombinierten Sechsecke noch im selben Zug weiterkombinieren können, Powerups aktiviert werden, und die Punktzahl neu berechnet wird. Die schlussendliche Lösung zu diesem Problem ist im nächsten Kapitel, der Codeanalyse, dokumentiert.

Die letzte Schwierigkeit war, am Ende eines Zuges herauszufinden, ob der Spieler noch eine Bewegung machen kann. Dies zu programmieren habe ich monatelang aufgeschoben, da ich es einerseits für nicht sehr wichtig hielt und andererseits nicht genau wusste, wie ich es machen wollte. Mitte November hatte ich dann die Idee, die ich ab hier erkläre. Ich muss für jedes Sechseck herausfinden, ob es sich in irgendeine Richtung bewegen kann. Der letzte Schritt, der mir so lange gefehlt hatte, war, dass ich nur die direkt anliegenden Sechsecke anschauen musste, um herauszufinden, ob sich das Sechseck bewegen kann, denn es ist nur dann keine Bewegung mehr möglich, wenn das Spielfeld voll ist und keine Sechsecke mit derselben Zahl nebeneinander liegen. So gehe ich einfach durch alle Sechsecke durch, und sobald eines sich bewegen kann, weiss ich, dass eine Bewegung möglich ist und breche den Rest der Anweisung ab. Nur wenn alle Sechsecke erfolglos geprüft wurden, wird das Spiel für verloren erklärt.

5 Codeanalyse

5.1 Java

Das gesamte Programm ist in der Programmiersprache Java geschrieben. Ein fertiges Java-Programm besteht aus kompilierten Dateien mit der Endung ".class". Solche Dateien werden vom Java-Compiler aus ".java"-Dateien gemacht, die ganz normale Textdateien sind. Der darin enthaltene Text ist nach speziellen Regeln formatiert, wodurch er vom Java-Compiler interpretiert werden kann. Diese Regeln nennt man auch die Syntax von Java. Meistens ist so eine Textdatei eine sogenannte Klasse. Eine solche Klasse ist im Grunde genommen ein Plan, nach dem zur Laufzeit des Programms ein Objekt erstellen werden kann. Man kann sich das vorstellen wie ein Hausbau: Ein Architekt fertigt einen Bauplan für das Haus an, wonach dann das Haus gebaut wird. Man kann beliebig viele dieser Objekte erstellen, genauso wie man beliebig viele solche Häuser bauen kann, die immer gleich rauskommen. Ein Objekt wird erstellt, wenn in der zuständigen Klasse ein sogenannter Konstruktor aufgerufen wird – man kann sich diesen als Bautrupp vorstellen. Der Konstruktor kann auch einige Argumente entgegennehmen und abhängig von diesen die Klasse unterschiedlich erstellen. Im Hausbeispiel wären diese Argumente einige kritischen Masse für das Haus. Klassen haben meistens einige Variablen und Methoden, die von danach erstellen Objekten natürlich übernommen werden. Variablen können entweder konstant – das heisst, sie können nicht mehr geändert werden, wenn das Programm läuft – oder variabel sein. Variablen können ganze Zahlen (*int* für "integer", oder *long* für grössere Zahlen), Dezimalzahlen (*float* für "floating point", *double* für doppelte Genauigkeit), Booleans (wahr oder falsch), Objekte oder alles erdenkliche andere sein. Methoden können, müssen aber nicht, Variablen als Argumente entgegennehmen.

Ein Beispiel einer solchen Methode:

```
1       //get the text to be displayed on the tile
2       public static String getDisplayedString(int b, int e)
3       {
4               return (int) (b * Math.pow(2, e % 10)) + SUFFIXES[e / 10];
5       }
```

Zeile 1 ist ein Kommentar: alles nach // wird ignoriert. Kommentare helfen dabei, den Code zu verstehen. Zeile 2 ist die Deklaration der Methode. Hier wird erklärt, dass die Methode eine öffentliche (*public*) Methode ist, die in der Klasse aufgerufen werden kann, ohne ein Objekt davon zu haben (*static*). Die Methode heisst *getDisplayedString*, nimmt die Ganzzahlen (*int*) *b* und *e* als Argumente (*()*) entgegen und gibt eine Zeichenfolge (*String*, im Grunde genommen ein Textstück) zurück. Die geschweiften Klammern auf Zeile 3 und 5 öffnen und schliessen den Inhalt der Methode. Zeile 4 sorgt dafür, dass die Methode etwas zurückgibt (*return*), nämlich alles ab *return* bis zum Semikolon (*;*), der die Anweisung abschliesst. Die zurückzugebende Zeichenfolge setzt sich zusammen aus zwei Teilen, die mit einem Plus zusammengefügt werden. Das eingeklammerte *int* anfangs des ersten Teils ist ein sogenannter Typecast, das heisst das danach eingeklammerte wird zum in Klammern angegebenen Typen umgewandelt. Dies ist nötig, da *Math.pow* generell ein *double* zurückgibt, was textual mit mindestens einer Kommastelle ausgegeben wird, selbst wenn die Zahl eigentlich ganz ist, z.B. 5.0 oder 183.0. Dieses Spiel heisst aber nicht 3.0k sondern 3k, weswegen das *double* auf *int* "gecastet" wird. Der Inhalt jener zweiten Klammer ist rein mathematisch: *b* *

$2^{e\%10}$. *e % 10* ist der Rest einer Division von *e* durch 10, *b* und *e* sind die beiden durchgegebenen Argumente. An das Ganze wird dann der zweite Teil angefügt, nämlich das "e/10"-te Glied der Kette *SUFFIXES*, einer Auflistung von Zeichenfolgen. Der Name der Zeichenfolge ist grossgeschrieben, da sie konstant ist. Sie ist oben in der Klasse folgendermassen definiert:

```
public static final String[] SUFFIXES = {"", "k", "m", "b", "t", "q"};
```

Hier treten einige neuen Elemente auf: *final* macht die Variable konstant, *String[]* sagt, dass es sich um einen *Array* (Kette, Auflistung, Reihe) von *Strings* handelt. Durch das = wird der vorangehenden Variable der nachstehende Wert zugewiesen, in diesem Fall eine Definition eines *Arrays* mit folgenden *Strings*: eine leere String (*""*), k (*"k"*), m, b, t und q. Diese werden dann den Zahlen angehängt, wenn ihr Exponent mindestens 10 ist. Dies sorgt für die Darstellung von 3k, 6k, 3m, etc.

5.2 Überblick

Das Spiel wird gestartet, wenn in Panel3k.java die Methode *main* aufgerufen wird. Diese Methode haben fast alle Java-Programme gemeinsam, denn sie wird vom System aufgerufen, um ein Programm zu starten. Bei der Deklaration der Klasse Panel3k wird *extends JPanel* angehängt. Das heisst, dass Panel3k alle öffentlichen Methoden und Variablen von *JPanel* übernimmt. Das einzige, was in der *main*-Methode passiert, ist, dass eine neue Instanz (ein aus der Klasse erschaffenes Objekt) in eine statische Variable eingespeichert wird. Dadurch können alle anderen Klassen und Objekte darauf zugreifen. Ausserdem wird dafür der Konstruktor aufgerufen, der den Wagen erst wirklich ins Rollen bringt. Der Konstruktor erstellt einen *JFrame*, ein Objekt, das ein Fenster in der Benutzeroberfläche darstellt (siehe hierzu auch Abbildung 9). *Panel3k* stellt bei diesem Fenster einiges ein, u.a. die Grösse, den Titel, die Position im Bild, und setzt sich dann als *JPanel* in den *JFrame* ein. Ein *JPanel* nimmt einen bestimmten Platz in einem Fenster ein. Durch diese Einsetzung ist z.B. die Mausbewegung unabhängig von der Höhe der Titelleiste des Fensters. Zudem macht der Konstruktor Instanzen von anderen wichtigen Klassen und speichert sie in statische Variablen ein, unter anderem von *Field.java*. *Field* ist das eigentliche Spielfeld. Es ist verantwortlich für die Verwaltung der Sechsecke, für die Bewegung (siehe nächstes Unterkapitel) und für die Aktivierung von Powerups, um einige Sachen zu nennen. *Field* hat eine Variable, die eine Matrix (Abb. 10) von *Hexagons ist*. Alle Sechsecke werden mit einem *Hexagon*-Objekt repräsentiert. Powerups und die Blockade sind Unterklassen von *Hexagon*,

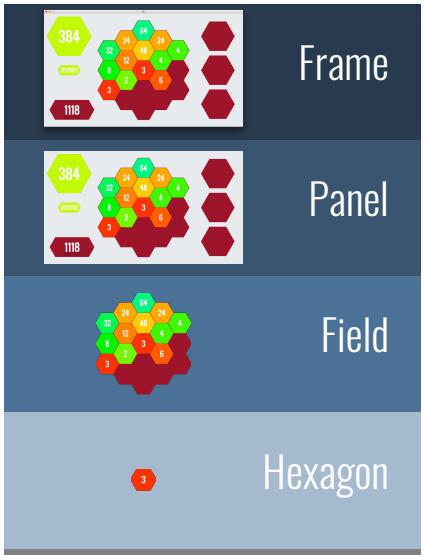


Abb. 9: Aufbau des Fensters

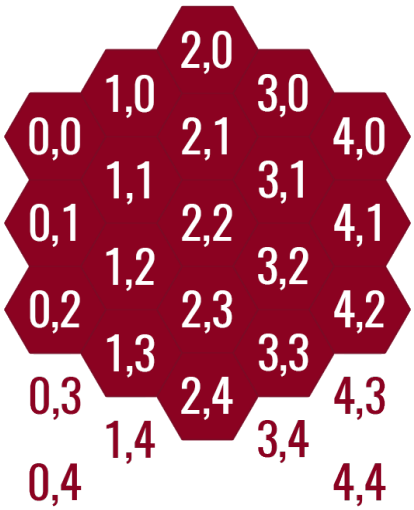


Abb. 10: Positionen der Felder in der Feldmatrix

die andere Bewegungsalgorithmen und einige Sonderfunktionen haben. Dann gibt es noch *InputManager*, der für Inputs aller Art verantwortlich ist. Alle Mausbewegung, -klicke und Tastaturereignisse werden von *Panel3k* an ihn weitergeleitet. Er kümmert sich um solche Ereignisse und leitet sie in Spezialfällen, z.B. Menüs oder der Anleitung, weiter. Wie bereits in den Schwierigkeiten erklärt, läuft auch die Bewegung über ihn. Wenn *InputManager* ein Ereignis abfängt, das eine Bewegung auslösen könnte, berechnet er die Richtung, in die bewegt werden soll. Als Beispiel: die Maustaste wird losgelassen. Ist der Abstand von Startpunkt zu Endpunkt kleiner als der Mindestradius, beträgt die Richtung -1, also wird gar nicht bewegt. Ansonsten wird ausgerechnet, in welchem der umliegenden Sechstel der Endpunkt relativ zum Startpunkt liegt und eine Zahl zugewiesen (Abb. 11). Diese Zahl wird dann durch verschiedene Methoden durchgegeben, was in einer Bewegung resultiert. Abbildung 12 stellt dar, wie alle Klassen zusammenhängen. Abbildung 13 gibt eine Übersicht der Methoden, die für eine Bewegung gebraucht werden, um deren Code es im nächsten Unterkapitel geht.

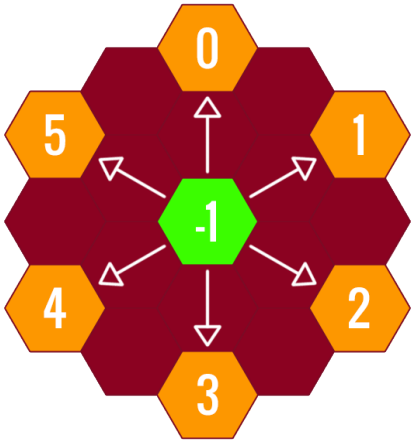


Abb. 11: Bewegungsrichtungen

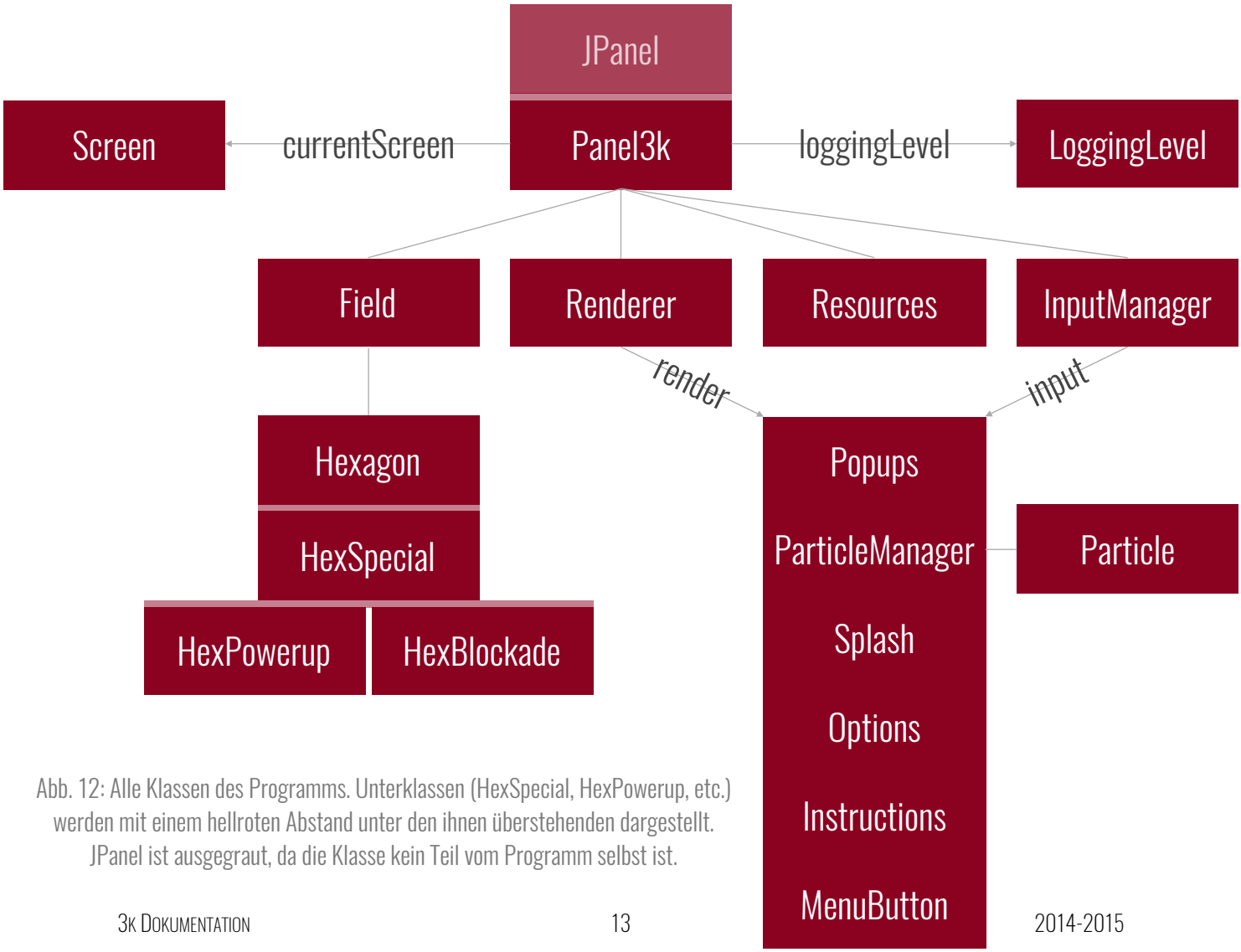
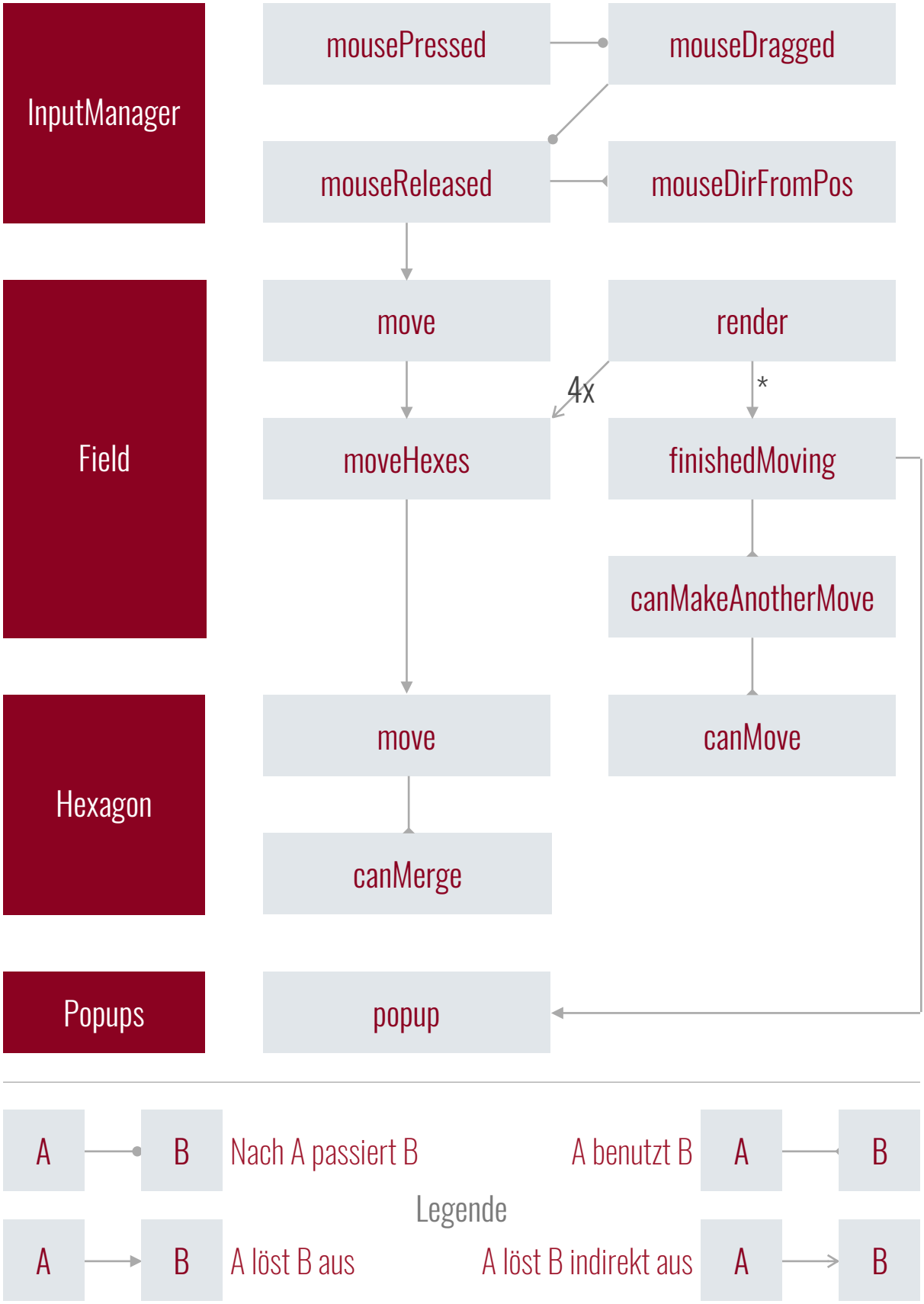


Abb. 12: Alle Klassen des Programms. Unterklassen (HexSpecial, HexPowerup, etc.) werden mit einem hellroten Abstand unter den ihnen überstehenden dargestellt. JPanel ist ausgegraut, da die Klasse kein Teil vom Programm selbst ist.



*: finishedMoving wird erst nach der vierten Iteration der Bewegung ausgelöst

Abb. 13: Ablauf einer Bewegung. Links die Klassen, deren Methoden (rechts davon) benutzt werden. Einige Methoden wurden weggelassen, da sie für die Erklärung nicht wichtig sind.

5.3 Bewegung

Eine Bewegung nach oben rechts kann drei Auslöser – Maus, Pfeiltasten, Buchstabentasten – haben; aus Platzgründen wird hier nur auf ersteren eingegangen. In diesem Unterkapitel wird Zeile um Zeile der Code erklärt, der am Schluss zu einer Bewegung nach oben rechts führt. Es fängt in *InputManager.java* folgendermassen an:

```
1       @Override
2       public void mousePressed(MouseEvent arg0)
3       {
4               Panel3k.log("Pressed at " + arg0.getX() + ", " + arg0.getY());
5               startX = arg0.getX();
6               startY = arg0.getY();
7               dragDuration = 0;
8       }
```

Zeile 1: *@Override* sagt, dass eine Funktion der überliegenden Klasse oder Interface (in diesem Fall letztere) überschrieben wird, und warnt, wenn es keine solche Methode gibt. Dies ist wichtig, wenn zum Beispiel die überschriebene Methode umbenannt wird. Ohne *@Override* würde man nicht darauf aufmerksam, dass dieser Teil des Codes höchstwahrscheinlich nicht mehr wie geplant funktioniert.

Zeile 2: Die Methode wird eingeführt, *public* sorgt dafür, dass sie von allen anderen Klassen auch aufgerufen werden kann; *void* sagt, dass es eine Methode ist, die keinen Wert zurückgibt; *mousePressed* ist der name der Methode; *(MouseEvent arg0)* erklärt, dass die Methode einen Wert verlangt, nämlich einen *MouseEvent*, der nachher unter arg0 in der Methode verwendet werden kann. *InputManager.java* wurde vorher so registriert, dass diese Methode aufgerufen wird, wenn der Benutzer eine Maustaste runterdrückt.

Zeile 3: *{* öffnet den Inhalt der Methode. Alles danach – bis *}* – ist Teil der Methode.

Zeile 4: *Panel3k.log* ruft in der Klasse *Panel3k* die Methode *log* mit den Argumenten in der Klammer auf, nämlich *"Pressed at " + arg0.getX() + ", " + arg0.getY()*, was im Grunde genommen Text zusammensetzt. *Panel3k.log loggt* den gegebenen Text. *Loggen* heisst, etwas zur Konsole auszugeben, zum Beispiel Fehlermeldungen, oder Informationen zum Verlauf des Programms, die sehr wichtig sind, wenn man einen Fehler finden muss. Der Text setzt sich zusammen aus einem statischen Stück *"Pressed at "*, der x-Koordinate des *MouseEvents*, in diesem Fall des Startpunkts der Ziehbewegung und seiner y-Koordinate, getrennt mit Komma und Leerschritt. *;* beendet die Anweisung.

Zeilen 5-6: Hier wird der Startpunkt der Bewegung in öffentliche Variablen eingespeichert. Diese werden nachher benutzt, um dem Benutzer eine Vorschau der Bewegung am Startpunkt zu geben.

Zeile 7: Die Dauer der Bewegung, *dragDuration*, wird auf 0 gesetzt. Diese Variable wird nachher jedes Mal, wenn das Bild gerendert wird, um 1 erhöht. Wenn sie über 20 ist, wird mit einem Pfeil bzw. Ring angezeigt, wohin die Bewegung ginge, wenn der Benutzer in dem Moment loslassen würde.

Zeile 8: *}* schliesst die Methode.

Während der Ziehbewegung wird folgendes ausgeführt:

```
1       @Override
2       public void mouseDragged(MouseEvent arg0)
3       {
```

```
4               mousePosX = arg0.getX();
5               mousePosY = arg0.getY();
6               dir = mouseDirFromPos(startX, startY, 50);
7       }
```

Zeilen 1-3: Siehe oben. Diese Methode wird immer wieder ausgeführt, wenn die Maus mit gehaltener Maustaste bewegt wird.

Zeilen 4-5: Die momentane Position der Maus wird in öffentliche Variablen eingespeichert.

Zeile 6: Setzt die Richtung der Bewegung auf den Wert, den die Methode *mouseDirFromPos* – anhand der Argumente *startX*, *startY* und dem Mindestradius 50 – ausrechnet, die Trigonometrie verwendet, um herauszufinden, in welche Richtung bewegt werden soll. Interessierte können im Anhang nachlesen.

Zeile 7: Siehe oben.

Beim loslassen der Maustaste wird diese Methode ausgeführt:

```
1       @Override
2       public void mouseReleased(MouseEvent arg0)
3       {
4               mousePosX = arg0.getX();
5               mousePosY = arg0.getY();
6               if(Panel3k.currentScreen == Screen.PAUSE_MENU)
7                       Panel3k.instance.pauseButtonPressed(mouseDirFromCenter(112));
8               if(Panel3k.currentScreen == Screen.OPTIONS)
9                       Panel3k.options.mouseReleased(arg0.getX(), arg0.getY());
10              if(Panel3k.currentScreen == Screen.INSTRUCTIONS)
11                      Panel3k.instructions.mouseReleased(arg0.getX(), arg0.getY());
12              if(Panel3k.currentScreen == Screen.SPLASH)
13                      Panel3k.splash.mouseReleased(arg0.getX(), arg0.getY());
14
15              if(Options.controlByDrag)
16              {
17                      //refresh direction
18                      dir = mouseDirFromPos(startX, startY, 50);
19                      //debugging
20                      Panel3k.log("Released at " + arg0.getX() + ", " + arg0.getY());
21                      Panel3k.log("Direction: " + dir);
22                      Panel3k.field.move(dir);
23              }
24
25              Panel3k.popups.mouseClicked();
26              Panel3k.menuButton.mouseClicked();
27
28              dragDuration = -1;
29              dir = -1;
30      }
```

Zeilen 1-5: Siehe oben.

Zeilen 6-13: Delegiert (leitet weiter) in diversen Spezialfällen – Nacheinander an Pausemenü, Optionen, Anleitung, Startbildschirm – die Mausbewegung an die jeweils dafür zuständigen Klassen. Weil der Inhalt der *if-Statements* aus nur einer Zeile besteht, werden keine geschweiften Klammern benötigt.

Zeile 14: Diese leere Zeile wird ignoriert. Sie ist aus ästhetischen Gründen platziert.

Zeile 15: *if* leitet ein *if-Statement* ein, was seinen Inhalt nur ausführt, wenn die in Klammern gegebene Kondition erfüllt ist, in diesem Fall *Options.controlByDrag*: ob die Bewegung per Maus aktiviert ist.

Zeile 16: *{* öffnet den Inhalt des *if-Statements*, der ausgeführt wird, wenn die Kondition erfüllt ist.

Zeile 17: *//* sorgt dafür, dass der Rest der Zeile auskommentiert ist und vom Programm ignoriert wird.

Zeile 18: Siehe vorige Methode, Zeile 6.

Zeilen 20-21: Einige Informationen zur Mausbewegung werden geloggt.

Zeile 22: Die Methode *move* von *field* (*Field.java*), worauf durch *Panel3k* zugegriffen wird, wird mit dem Argument *dir* – der Richtung der Bewegung – aufgerufen. Darauf, was diese Methode macht, wird gleich eingegangen.

Zeile 23: *}* schliesst das *if-Statement*.

Zeilen 25-26: *popups* (*Popups.java*) und *menuButton* (*MenuButton.java*), beides durch *Panel3k* verfügbar gemachte Klassen, wird gesagt, dass der Benutzer mit der Maus geklickt hat.

Zeilen 28-29: Variablen *dragDuration* und *dir* werden wieder auf -1 zurückgesetzt.

Zeile 30: Siehe oben.

Durch obigen Code wird in der Klasse *Field.java* die Methode *move* mit der Bewegungsrichtung ausgeführt:

```
1       public void move(int dir)
2       {
3               if(currentDir > -1 || dir == -1)
4                       return;
5               currentDir = dir;
6               //depending on direction of movement, go through all hexes in a specific
order, and tell each of them to move
7               moveTimes = TIMES_TO_MOVE;
8               moveTick = MAX_TICK;
9               moveHexes(dir);
10      }
```

Zeilen 1-2: Das Übliche. Öffnet die Methode *move*, die als Argument die Bewegungsrichtung *dir* nimmt.

Zeilen 3-4: Ein *if-Statement*, dessen Kondition aus zwei mit einem inklusiven "Oder" (||) verbundenen Unterkonditionen besteht. Das heisst, die Kondition ist erfüllt, wenn mindestens eine der Unterkonditionen erfüllt ist, aber auch, wenn es beide sind. In diesem Fall dient es dazu, die Methode abzubrechen (*return;*), wenn schon eine Bewegung im Gange ist oder gar keine Bewegung ausgelöst wurde (*dir == -1*). Dieses *if-Statement* braucht keine geschweiften Klammern (*{}*), da der Inhalt nur eine Zeile Code ist: *return;*.

Zeile 5: Die der Methode gegebene Richtung *dir* wird in *currentDir* eingespeichert. *currentDir* wird danach von anderen Methoden verwendet.

Zeilen 7-8: *moveTimes* bzw. *moveTick* werden auf die vorher gesetzten Konstanten *TIMES_TO_MOVE* UND *MAX_TICK* gesetzt, die zur Laufzeit des Programms nicht mehr verändert werden. Aus Konvention werden solche Konstanten mit Grossbuchstaben bezeichnet.

Zeile 9: Die Methode *moveHexes* wird mit dem Argument *dir* aufgerufen. Gleich mehr dazu.

Zeile 10: Die übliche schliessende Klammer.

Von Zeile 9 wird in derselben Klasse, *Field.java*, folgende Methode aufgerufen:

```
1       public void moveHexes(int dir)
2       {
3               nextField = new Hexagon[5][5];
4               if(dir == 1 || dir == 2) //moving right
5                       for(int i = 4; i > -1; i--)
6                       {
7                               if(dir == 1) //upper
8                                       for(int j = 0; j < 5; j++)
9                                       {
10                                              Hexagon h = field[i][j];
11                                              if(h != null)
12                                                      h.move(dir);
13                                      }
14                              else //lower: dir is 2
15                                      for(int j = 4; j > -1; j--)
16                                      {
17                                              Hexagon h = field[i][j];
18                                              if(h != null)
19                                                      h.move(dir);
20                                      }
21                      }
22              else //moving left or vertically
23                      for(int i = 0; i < 5; i++)
24                      {
25                              if(dir == 3 || dir == 4) //lower
26                                      for(int j = 4; j > -1; j--)
27                                      {
28                                              Hexagon h = field[i][j];
29                                              if(h != null)
30                                                      h.move(dir);
31                                      }
32                              else //upper: dir is 0 or 5
33                                      for(int j = 0; j < 5; j++)
34                                      {
35                                              Hexagon h = field[i][j];
36                                              if(h != null)
37                                                      h.move(dir);
38                                      }
39                      }
40      }
```

Zeilen 1-2: Wie immer. Diese Methode heisst *moveHexes* und wird mit dem Argument *dir* aufgerufen.

Zeile 3: Eine leere Matrix von *Hexagons wird in die Variable nextField* eingespeichert.

Zeilen 4-39: Eine komplexe Struktur von *if-else-Statements*, die darauf hinauslaufen, dass das Feld entgegen der Bewegungsrichtung durchlaufen wird, wobei in jedem Platz der Matrix geschaut wird, ob er ein *Hexagon* enthält und, wenn dies so sein sollte, in jenem die Methode *move* mit der gegebenen Richtung aufgerufen wird. Auf diese Methode *move* wird gleich weiter eingegangen.

Zeile 40: Eine ganz normale schliessende Klammer.

Weiter geht es mit *move* in *Hexagon.java*:

```
1       public void move(int dir)
2       {
3               startX = posX;
4               startY = posY;
5               targetX = posX;
6               targetY = posY;
7               try
```

```
8               {
9                       switch(dir)
10                      {
11-28                           [...]
29                              case 1: //UP RIGHT
30                                      if(startX < 4 && startY > (startX < 2 ? -1 : 0))
31                                              if(Field.nextField[startX+1][startY - (startX <
                                                        2 ? 0 : 1)] == null)
32                                              {
33                                                      targetY = startY - (startX < 2 ? 0 : 1);
34                                                      targetX = startX+1;
35                                              }
36                                              else
37                                              {
38                                                      Hexagon hex = Field.nextField[startX+1]
        [startY - (startX < 2 ? 0 : 1)];
39                                                      if(canMerge(hex))
40                                                      {
41                                                              doubleHex();
42                                                              if(Panel3k.activePowerups[0] > 0)
43                                                                      doubleHex();
44                                                              targetY = startY - (startX < 2 ?
        0 : 1);
45                                                              targetX = startX+1;
46                                                              hasMerged = true;
47                                                      }
48                                              }
49                                      break;
50-138                          [...]
139                     }
140                     Hexagon h = new Hexagon(targetX, targetY, base, exponent, startX,
                                startY, targetX, targetY);
141                     if(startX != targetX || startY != targetY)
142                     {
143                             Panel3k.log("Generation " + generation + " hex "
                                        getDisplayedString() + " moved. X: " + startX + "=>" +
                                        targetX + ", Y: " + startY + "=>" + targetY);
144                             if(hasMerged)
145                                     checkIfHighestHex();
146                             hasMoved = true;
147                     }
148                     h.hasMoved = hasMoved;
149                     h.hasMerged = hasMerged;
150                     h.generation = generation + 1;
151                     h.flash = flash;
152                     Field.nextField[targetX][targetY] = h;
153             } catch (Exception e)
154             {
155                     e.printStackTrace();
156                     Panel3k.log(LoggingLevel.ERROR, "Exception caused! Hex " +
                                getDisplayedString() + " moved. X: " + startX + "=>" + targetX
                                + ", Y: " + startY + "=>" + targetY + ", dir: " + dir);
157             }
158     }
```

Zeilen 1-2: Wie immer: Methode *move* wird mit dem Argument *dir* aufgerufen.

Zeilen 3-6: *startX*, *startY*, *targetX* und *targetY* werden auf die momentanen Positionen gesetzt.

Zeile 7: Ein *try-Statement* wird eingeleitet. Wenn im Inhalt eines *try-Statements* etwas schief läuft und einen der mit *catch()* abgefangenen Fehler aufwirft, führt das Programm den Code im *catch*-Block aus und läuft trotz des Fehlers weiter.

Zeile 9: Einleitung eines *switch-Statements*. Ein *switch-Statement* nimmt ein Argument und geht damit durch die mit *case* angegebenen Fälle durch. Wenn ein Fall zutrifft, wird der Code ab da bis zum nächsten *break* ausgeführt.

Zeilen 11-28: Code für die Bewegung nach oben. Aus Platzgründen weggelassen.

Zeile 29: Wenn die durchgegebene Variable gleich 1 ist, wird der Code ab hier ausgeführt.

Zeile 30: Hier wird geprüft, ob sich das Sechseck überhaupt nach oben rechts bewegen kann. Wenn er rechts oder oben am Rand ist, wird an dieser Stelle abgebrochen und das Sechseck bleibt an Ort und Stelle. Die y-Koordinate wird mithilfe von etwas ausgedrückt, was im Grunde genommen ein *if-else-Statement* ist: *startX < 2 ? -1 : 0*. Dabei ist *startX < 2* die Kondition; wenn sie erfüllt ist, ist der Wert das nach dem *?* gegebene, also hier *-1*; wenn nicht, ist er das nach dem *:* gegebene, also hier *0*. *&&* verbindet zwei Konditionen mit einem logischen *Und*. Das heisst, es müssen beide Konditionen erfüllt sein, damit die Verbindung es ist.

Zeile 31: Wenn der oben rechts benachbarte Platz frei ist, wird das Sechseck einfach dorthin geschoben.

Zeilen 33-34: Die Zielposition, *targetX* und *targetY*, wird auf das leere Feld gesetzt.

Zeile 36: Wenn der Platz schon besetzt ist, wird es komplizierter.

Zeile 38: Das Sechseck, was den Platz besetzt, wird lokal gespeichert.

Zeile 39: Es wird geprüft, ob das Sechseck mit dem bestehenden Sechseck kombinierbar ist. *canMerge* vergleicht einfach die Basis und den Exponenten des Sechsecks mit denjenigen des gegebenen, schaut, dass es kein Powerup oder die Blockade ist und stellt sicher, dass das Sechseck nicht in diesem Zug schon mit einem anderen kombiniert wurde.

Zeile 41: Verdoppelt den Wert des Sechsecks. *doubleHex* erhöht einfach den Exponenten um 1.

Zeilen 42-43: Wenn das Powerup *1+1=4* aktiv ist, wird der Wert nochmals verdoppelt, sodass er insgesamt vervierfacht wird.

Zeilen 44-45: Setzt die Zielposition auf das Sechseck, mit dem dieses kombiniert wird.

Zeile 46: Merkt sich, dass das Sechseck in diesem Zug schon mit einem anderen kombiniert wurde, damit ein Sechseck nicht in einem Zug mit mehreren kombiniert werden kann.

Zeile 49: *break* beendet diesen Fall des *switch-Statements*.

Zeilen 50-138: Weitere Fälle für die Bewegung in die anderen 4 Richtungen.

Zeile 139: *}* beendet das *switch-Statement*.

Zeile 140: Ein neues Sechseck wird erstellt. Das Sechseck ist genau gleich wie das momentane, bloss dass es an der Zielposition erstellt wird.

Zeile 141: Frag ab, ob sich das Sechseck bewegt hat.

Zeile 143: Loggt verschiedene Daten zur Bewegung und zum Sechseck.

Zeilen 144-145: Wenn das Sechseck sich mit einem anderen zusammengeschoben hat, wird geschaut, ob seine Zahl höher als der bestehende Rekord ist. Wenn der Rekord gebrochen ist, wird er mit dem neuen überschrieben.

Zeile 146: Setzt *hasMoved* auf *true*. Der Wert dieser Variable wird nachher benötigt, wenn nachgeschaut wird, ob ein neues Sechseck hinzugefügt werden soll.

Zeilen 148-151: Setzt diverse Variablen des neuen Sechsecks relativ zu denen des momentanen.

Zeile 152: Speichert das kreierte Sechseck in die nächste Feldmatrix ein.

Zeile 153: Fängt alle Fehler ab, die während der Bewegung passieren können. Wenn ein Fehler auftreten sollte, wird der folgende Code ausgeführt.

Zeile 155: Loggt den *Stack Trace* des Fehlers. Dies beinhaltet die Art des Fehlers und die Zeilen Code der verschiedenen Klassen, die dazu geführt haben.

Zeile 156: Loggt einige Daten zur versuchten Bewegung. Dadurch, dass sie mit einem *LoggingLevel* names *ERROR* geloggt werden, werden sie auch dann geloggt, wenn loggen eigentlich ausgestellt ist.

An dieser Stelle ist es wichtig, zu erklären, wie die Bewegung in *Field.java* abläuft: bisher wurde *dir*, die Bewegungsrichtung, in *currentDir* eingespeichert. Ausserdem wurden *moveTimes* und *moveTick* auf ihre Maximalwerte gesetzt, die als Konstanten gespeichert waren. Wenn das Feld gerendert wird (*render* in *Field.java*), passiert unter anderem folgendes:

```
1               if(moveTick > 0)
2               {
3                       moveTick -= Panel3k.renderer.getSPF();
4                       if(moveTick <= 0)
5                       {
6                               moveTimes--;
7                               exchangeFields();
8                               moveHexes(currentDir);
9
10                              if(moveTimes > 0)
11                                      moveTick = MAX_TICK;
12                              else
13                                      finishedMoving();
14                      }
15              }
```

Zeile 1: Schaut, ob gerade eine Bewegung im Gange ist.

Zeile 3: Verringert *moveTick* um die Zeit, die der Renderer für ein Mal Rendern braucht. Dadurch braucht eine Bewegung immer unabhängig von der Renderzeit gleich lange.

Zeile 4: Schaut, ob die erste Iteration der Bewegung abgeschlossen ist.

Zeile 6: Verringert *moveTimes* um 1.

Zeile 7: Ruft *exchangeFields* auf, welches *nextField* in *field* einspeichert.

Zeile 8: Bewegt die Sechsecke einen weiteren Schritt in die Bewegungsrichtung.

Zeilen 10-13: Wenn die letzte Iteration vollendet ist, wird *finishedMoving* aufgerufen, sonst wird *moveTick* wieder auf den maximalWert gesetzt und dadurch die Bewegung weitergeführt.

Weiter geht es in der oben genannten Methode *finishedMoving*:

```
1       public void finishedMoving()
2       {
3               Panel3k.log("Finished moving!");
4
5               if(hasAnythingMoved())
6               {
7                       addRandomHex();
8                       if(Panel3k.activePowerups[0] > 0)
9                               Panel3k.activePowerups[0]--;
10                      if(Panel3k.activePowerups[2] > 0 && Panel3k.activePowerups[1] == 0)
11                              Panel3k.activePowerups[2]--;
12                      if(Panel3k.activePowerups[1] > 0)
13                              Panel3k.activePowerups[1]--;
14              }
15
16              moveNumber++;
17              if(moveNumber == 100 && Options.difficulty == 0)
18                      Panel3k.popups.popup(1);
19
20              for(int i = 0; i < 5; i++)
21                      for(Hexagon h : field[i])
22                              if(h != null)
23                                      h.doneMoving();
24
25              currentDir = -1;
26
27              Panel3k.instance.writeSaveFile();
28
29              if(!canMakeAnotherMove())
30                      Panel3k.popups.popup(2);
31      }
```

Zeile 1: Das übliche Spiel. Diese Methode heisst *finishedMoving* und nimmt keine Argumente.

Zeile 3: Loggt, dass die Bewegung abgeschlossen wurde.

Zeile 5: Schaut, ob sich irgendwas bewegt hat. *hasAnythingMoved* geht einfach durch das Feld durch und fragt für jedes Sechseck ab, ob es sich bewegt hat.

Zeile 7: Fügt ein neues beliebiges Sechseck zum Feld hinzu. Diese Methode macht mit einer gewissen Wahrscheinlichkeit auch stattdessen Powerups und berücksichtigt allfällige aktive Powerups, die dafür sorgen können, dass gar keine neuen Sechsecke, oder nur solche mit Exponent 2, dazukommen.

Zeilen 8-13: Die Laufzeiten der momentanen Powerups werden verringert. Wenn gleichzeitig das Powerup an ist, was dafür sorgt, dass keine neuen Sechsecke dazukommen, und das, was dafür sorgt, dass nur solche mit Exponent 2 dazukommen, wird nur die Laufzeit des ersteren verringert, da auch nur das angewendet wird.

Zeile 16: Erhöht *moveNumber* um 1. ++ kann man vor oder nach einen Variablennamen stellen, um sie um 1 zu erhöhen. Zum Verringern kann man −− benutzen.

Zeile 17: Prüft, ob der Spieler 100 Bewegungen gemacht hat und die Schwierigkeit immer noch auf dem tiefsten Niveau ist.

Zeile 18: Zeigt ein Popup an, welches den Spieler daran erinnert, dass es weitere Schwierigkeitsstufen gibt.

Zeilen 20-23: Ruft in jedem Sechseck *doneMoving* auf, wodurch Powerups aktiviert werden, wenn es mit einem anderen zusammengeschoben wurde, und *hasMoved* und *hasMerged* zurückgesetzt werden.

Zeile 25: Setzt die Bewegungsrichtung auf -1 zurück.

Zeile 27: Speichert den Spielstand in eine Textdatei ab.

Zeile 29: Schaut, ob der Spieler keine Bewegung mehr machen kann.

Zeile 30: Zeigt ein Popup an, welches dem Spieler sagt, dass er keine Bewegung mehr machen kann und deshalb verloren hat. Darauf, wie das etwas funktioniert, wurde schon im Kapitel "4.3: Schwierigkeiten" eingegangen. Ein tieferer Einblick ist im angehängten kompletten Code möglich.

6 Fazit

6.1 Rückblick

Dadurch, dass ich so früh mit der Entwicklung des Spiels anfing, hatte ich viel Zeit für Details. Am Ende der Sommerferien hatte ich das Spiel praktisch fertig. Es war angenehm spielbar und es funktionierte alles, was funktionieren sollte. Was ich ab da machte, waren nur noch kleine Änderungen. Ich finde aber, dass die Qualität eines Spiels, neben den offensichtlichen Kriterien, ganz immens von diesen kleinen aber feinen Details abhängt. Das ist für mich der Unterschied zwischen einem Spiel, was man spielt, bis man ein Ziel erreicht, wonach man es nie wieder anrührt, und einem Spiel, was man einfach spielt, weil es so angenehm zu spielen ist. Für so detaillierte Sachen haben grosse Spielhersteller keine Zeit, ausserdem lohnt es sich für sie gar nicht, so tief reinzugehen. Genau diese Liebe zum Detail, gekoppelt mit den durch das Internet neu eröffneten Möglichkeiten, Medien an die Leute zu bringen, ohne sich an ein grosses Label zu verkaufen, hat der Welt der sogenannten Indie-Games[2] ermöglicht, so stark zu wachsen, wie sie es in den letzten paar Jahren tat. Zum Beispiel beim Indie-Game "Thomas Was Alone"[3], einem minimalistischen Rätselspiel, was ich nur empfehlen kann. In diesem Spiel steuert man Rechtecke in einer Welt herum und benutzt ihre verschiedenen Eigenschaften schlau, um das Level abzuschliessen. Der Entwickler hat mehrere Wochen fast nur daran gearbeitet, den Sprungmechanismus dieser Rechtecke absolut perfekt zu machen. Und genau dies fühlt man, wenn man das Spiel spielt: man spürt die Arbeit, die in dem Spiel steckt. Diese Perfektion ist das Ziel, worauf ich mit all diesen kleinen Änderungen hingearbeitet habe.

Rückblickend gibt es wirklich nur eine grössere Sache, die ich besser hätte machen können. Wie im ersten Unterkapitel der Codeanalyse erklärt, ist das Feld so ausgerichtet, dass es in der Mitte einen Knick macht. Dies sorgt für viele aufwändige Spezialfälle, einen derer ich bei der Analyse einer Bewegung in Zeile 30 der *move*-Methode der Klasse *Hexagon.java* erläutere. Einer meiner Klassenkameraden brachte nach der Vollendung des Spiels die geniale Idee, die Matrix einfach schräg über das Feld zu legen (Abb. 14), wodurch ich bei der Bewegung unabhängig von der x-Position die y-Position eines anliegenden Felds bestimmen könnte. Mehr Informationen dazu, wie es mit der verwendeten Methode funktioniert, gibt es in der Codeanalyse der Bewegung.

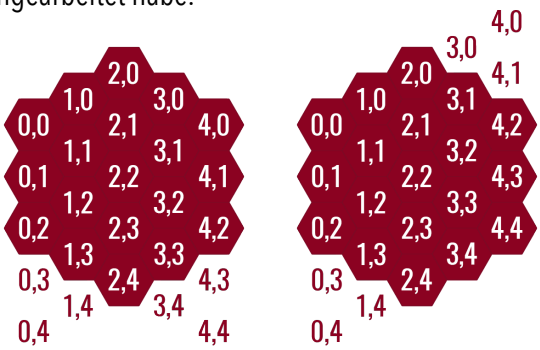


Abb. 14: Links die verwendete Methode, rechts die bessere Matrixauslegung.

Alles zusammengezählt müsste ich sagen, dass in die Entwicklung des Spiels an sich etwa doppelt so viel Zeit gesteckt habe wie in die Dokumentation. Jedoch kommt es mir eher so vor, als hätte ich gleich lange für beides gebraucht, da ich beim Programmieren so konzentriert bin, dass die Zeit viel schneller vergeht.

---

[2] "Indie" von engl. "independent": unabhängig. Hier: Spiele, die nicht unter Auftrag einer grossen Firma, die sie unter einen strengen Zeitplan mit genauen Bestimmungen stellen würde, entwickelt wurden.

[3] http://www.mikebithellgames.com/thomaswasalone/

6.2 Ausblick

Das Aufwendigste, was ich mit diesem Konzept noch vorhabe, ist, daraus eine App für iOS zu machen. Ich habe inzwischen schon einige kleinere Apps entwickelt[4], aber noch nicht eine so aufwendige. Das Icon für diese App habe ich bereits kreiert (Abb. 15). Die App ermöglicht mir, das Spiel weiter zu verteilen und – durch Platzierung von Werbung in der kostenfreien Version und eine Version, die ein oder zwei Franken kostet – vielleicht sogar Profit aus ihr zu schlagen. Einer Sache bin ich mir sicher: in der App werde ich die Feldmatrix optimal ausrichten.

Abb. 15: Icon für iOS

[4] Unter anderem Toucan Flight (www.toucanflight.tk) und Color Picker, wofür ich keine Website gemacht habe. Dafür einfach auf dem App Store nach "Color Picker Game" suchen. Herausgegeben sind die Apps bisher unter dem Namen der Firma meines Vaters: Aristo Consulting GmbH.

7 Bemerkungen

7.1 Danksagungen

Ich danke meinen Klassenkameraden und Eltern, ohne deren Feedback und Testen diese Arbeit nie geworden wäre, wie sie nun ist. Besonders herzlich danke ich meinem Vater, der sich meine fertige Arbeit durchgelesen und mich auf einiges aufmerksam gemacht hat, was ich sonst übersehen hätte, und ohne dessen Ansporn ich möglicherweise nie zum Programmieren gekommen wäre.

7.2 Links

http://play3k.tk – Offizielle Website zum Spiel mit Download, YouTube-Video und Updates

https://dl.dropbox.com/u/20470051/3k.app.zip – Direkter Link zum Download der Mac App

http://adf.ly/uwQqd – Gekürzter Link dazu

https://dl.dropbox.com/u/20470051/3k.jar – Direkter Link zum Download der JAR-Datei

http://adf.ly/uwQtQ – Gekürzter Link dazu

http://www.youtube.com/watch?v=zqP121n7i80 – Video zum Spiel auf YouTube

http://adf.ly/uwQv9 – Gekürzter Link dazu

https://dl.dropbox.com/u/20470051/3k%20CD.zip – Direkter Link zum Download des Inhalts der CD

http://adf.ly/vBquX – Gekürzter Link dazu

7.3 Quellen

http://eclipse.org – Eclipse ist die Anwendung, mit der ich das gesamte Spiel programmiert habe.

http://jhlabs.com/ip/blurring.html – Daher habe ich (mit Erlaubnis) den Code für den Unschärfe-Effekt des Hintergrunds im Pausemenü.

http://stackoverflow.com – Wenn ich beim Programmieren Probleme antraf, war dies meine "go-to"-Seite, um eine Lösung zu finden. Fand ich die Lösung hier nicht, habe ich einfach Google benutzt, und wenn selbst das nicht funktionierte, habe ich auf StackOverflow mein Problem dargelegt und Hilfe bekommen.

7.4 Verwendete Programme

Eclipse (Kepler) zum Programmieren, Kompilieren und Exportieren

Adobe Photoshop CS6 für Bilddateien

Pages (5.5) zum Schreiben dieser Arbeit

Anhang: CD

Auf der CD enthalten ist:

Der gesamte Quellcode

Eine ausführbare JAR-Datei

Eine Mac-Application

Ein PDF dieser Dokumentation